MASTER EN SCIENCES INFORMATIQUES

Mesures de performances de UNIX à partir d'un modèle mathématique

Cornil, Dirk

Award date:
1981

Awarding institution:
Universite de Namur

FACULTES
UNIVERSITAIRES
N.D. DE LA PAIX

NAMUR

INSTITUT D'INFORMATIQUE

RUE GRANDGAGNAGE, 21, B - 5000 NAMUR (BELGIUM)

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX,
NAMUR.

INSTITUT D'INFORMATIQUE.

MESURES DE PERFORMANCES DE UNIX A PARTIR D'UN MODELE MATHEMATIQUE

Promoteur : M. Noirhomme

Dirk Cornil

Mémoire présenté en vue de l'obtention du grade de
LICENCIE ET MAITRE EN INFORMATIQUE.

Année académique 1980 - 1981.

REMERCIEMENTS.

Je tiens tout d'abord à remercier Madame M. Noirhomme qui a accepté de diriger ce mémoire et qui m'a permis , par ses conseils judicieux , de mener ce travail à son terme.

J'exprime également toute ma gratitude au Professeur A. S. Tanebaum ainsi qu'à Messieurs J. Stevenson , S. Mullender et R. Wiggers qui m'ont très bien accueillis à la Vrije Universiteit d'Amsterdam où ils m'ont aidés à me familiariser avec le système d'exploitation UNIX

Je remercie aussi tout particulièrement Monsieur J-P. Adans sans l'aide duquel ce travail n'aurait pu se réaliser.

Je remercie Monsieur P. Lambion dont les avis éclairés à propos de la réalisation théorique du travail me furent d'un grand secours ; sa patience fut mise à contribution.

Enfin , ma plus vive reconnaissance va à tous ceux qui , d'une manière ou d'une autre m'ont aidés au cours de cette année , et , plus particulièrement encore à mon entourage.

TABLE DES MATIERES.

INTRODUCTION

Nous avons tentè d'èlaborer un outil , et plus prècisèment un modèle , de mesures de performances pour le système d'exploitation UNIX (*) tournant sur le PDP 11/45 des Facultès universitaires de Namur.

La mise au point d'un tel outil est complexe et il ne nous a pas toujours ètè possible d'approfondir tous les dètails de sa rèalisation avec les mêmes soins que ceux attribuès par une èquipe de chercheurs.

Nous nous sommes ègalement attardès à la rèalisation d'un mesureur dont le but est de fournir les donnèes d'entrèe nècessaires au modèle ainsi que de comparer les rèsultats obtenus avec ceux du modèle.

Bien sûr , avant d'en arriver là , notre ètude s'est dèroulèe en plusieurs ètapes.

Tout d'abord , il nous a fallu acquèrir les notions de mètrologie nècessaires au dèmarrage du projet. Nous avons donc synthètisè quelques règles de base de la mètrologie dans notre chapitre premier.

Parallèlement , nous devions ègalement acquèrir une certaine connaissance de UNIX . Ce fut le but du stage qui s'est dèroulè dans la division mathèmatique de la Vrije Universiteit d'Amsterdam. Les particularitès de UNIX nècessaires pour la rèalisation du modèle se trouvent rèsumèes dans le chapitre deux.

Le chapitre trois est consacrè à l'èlaboration et l'explicitation du modèle choisi.

L'utilitè des rèsultats d'un modèle est nulle s'ils ne sont pas comparès à certaines valeurs de rèfèrence. Le chapitre quatre contient une description du mesureur utilisè pour vèrifier la validitè du modèle.

L'èlaboration du modèle ayant nècessitè un certain nombre d'hypothèses , celles-ci sont examinèes dans le chapitre cinq.

Le dernier chapitre est consacrè à l'analyse statistique des rèsultats du modèle.

Unix est une marque dèposèe par les Bell Laboratories et a ètè mis au point par D. M. Ritchie et K. Thompson.

I. LES MESURES DE PERFORMANCES.

1. Buts des mesures de performances.

Les mesures de performances de systèmes informatiques sont souvent désirées par les responsables de ces systèmes afin :

- de mieux comprendre le fonctionnement du système en vue de son amélioration éventuelle.

  L'intérêt économique d'une bonne utilisation des ressources est certain car le coût et la complexité du matériel et du logiciel ont tendance à augmenter.

- d'élaborer les critères de choix d'un nouveau système.

  Le choix d'un nouveau système nécessite la connaissance des conditions d'exploitation et des performances que l'on désire obtenir.

- etc ...

2. Objet des mesures.

Voici une liste non exhaustive de quelques objets que l'on peut désirer mesurer:

- le rendement du système en terme de taux d'utilisation

- le temps de réponse moyen ou le nombre de dépassements d'un temps limite de réponse lors d'interactions types aux terminaux d'un système conversationnel

- la fréquence des pannes de divers sous-ensembles matériels

- l'état des files d'attente (premier arrivé-premier servi, dernier arrivé-dernier servi, priorités,...)

et, pour les chercheurs en modélisation ou en simulation

- les fréquences d'arrivées d'événements de nature logique ou physique

- les durées de services aux différents serveurs

- le comportement des files d'attente.

3. Les méthodes de mesures.

La méthode la plus souvent utilisée pour etudier le comportement d'un système est l'élaboration d'un modèle représentant une description formalisée de la compréhension que nous avons de la réalité.

Il existe deux types de modèles:

- le modèle mathématique.

  Ce type de modèle est caractérise par un ensemble de relations liant l'évolution d'un certain nombre de variables typiques du système observé. Il permet de prédire le comportement du système lorsqu'il est soumis à des sollicitations d'un type déterminé.

  Malheureusement, un modèle mathématique ne permet pas de prendre en considération le système global: il demande un certain nombre de simplifications et d'hypothèses non réalistes. De plus, il est souvent difficile de donner une représentation précise de la charge à laquelle est soumise le système, or la validité des résultats fournis par le modèle dépend justement de celle-ci.

- le simulateur.

  Dans un simulateur, les relations arithmétiques sont remplacées par des algorithmes. Il est donc possible de modéliser le système global de facon plus précise , ceci à condition d'introduire des algorithmes proches de la réalité.

  Mais un simulateur coute cher à l'exécution et sa complexité entraîne une grande difficulté de mise au point.

Les résultats fournis par les modèles ne sont que des données brutes qui devront encore être interprétées par une analyse statistique. La validité du modèle dépendra de l'exactitude de l'interprétation.

4. Techniques et outils d'extraction.

Un système informatique en exécution est, en fait, un ensemble d'objets dont les valeurs évoluent avec le temps et une suite d'évènements correspondants aux changements d'état résultant de l'utilisation de ces objets ou de l'arrivée de signaux. Suivant le niveau auquel nous nous plaçons dans le système, ces entités (objets et évènements) ont un caractère de nature physique ou logique.

L'objectif d'un extracteur (ou "mesureur") est de prélever le maximum d'informations sur le matériel, mais aussi et surtout sur le logiciel.

Nous avons donc le choix entre deux catégories d'outils d'extraction:

- outils logiciels: appropriés pour l'observation d'entités de type logique

- outils matériels: appropriés pour l'observation d'entités de type physique

Outre le fait qu'il doive être approprié au type d'entité que l'on veut mesurer , le choix d'un extracteur est également guidé par les paramètres suivants:

- adaptabilité et portabilité face à de nouvelles mesures ou de nouveaux systèmes

- degré de perturbation induite dans l'utilisation des différentes ressources du système (unité centrale, mémoire)

- facilité de mise en oeuvre et d'emploi

- ensemble dés couts engendrés.

4.1. Les outils d'extraction logiciels(mesureurs "programmés").

4.1.1. Introduction.

S'il est d'utilisation souple, un extracteur devrait permettre à l'utilisateur:

- de lancer l'extraction et de l'arrêter au moment de son choix

- d'utiliser les moyens de stockage de son choix

- de choisir ses mesures en fonction de ses besoins parmi les possibilites qui lui sont offertes soit au lancement de l'extraction, soit dynamiquement durant celle-ci (ceci est possible lorsque l'outil est conçu selon une architecture modulaire)

Il devrait également être facilement extensible, c'est-à-dire que l'utilisateur devrait pouvoir écrire ses propres programmes de collecte et les inclure dans l'outil existant.

L'extracteur devrait également introduire une perturbation minimale, c'est-à-dire que lorsque l'on accède aux variables, cela devrait pouvoir être fait sans risques pour le fonctionnement du système . En fait, il faudrait toujours s'assurer de ne pas trop modifier les performances du système.

Malheureusement, l'objectif de non perturbation est, en général en contradiction avec les objectifs précédemment définis, à savoir généralité et facilité de mise en oeuvre. Il faudra donc tenter de trouver un compromis entre la précision des mesures et la perturbation introduite.

4.1.2. Les fonctions de base d'un outil d'extraction logiciel.

Afin d' éviter une perturbation trop importante du système, l'extracteur doit s'exécuter sur un niveau prioritaire afin d'empêcher une autre tache de prendre le contrôle de l'unité centrale lorsqu'une prise de mesures est en cours.

4.1.2.1. Les techniques d'activation de l'extracteur.

Il existe deux techniques d'activation:

a. Sur évènements.

Lorsque l'exécution atteint un certain point de mesure, l'extracteur prend le contrôle de l'unité centrale afin de prélever les variables désirées.

Inconvénients de cette technique:

- il est rarement possible de prévoir le nombre de fois que l'on passera par tel ou tel point de mesure

- la mise au point(localisation des sondes et des points de mesures) risque d'être difficile si l'outil n'est pas conçu en même temps que le système pour lequel il est prévu.

b. Par échantillonage.

Cette technique permet de prélever des variables quantitatives ou qualitatives servant à la gestion de l'état du système.

Le problème principal lors de la mise en oeuvre d'une telle technique est le choix de la période d'echantillonage. En effet, il faut arriver à établir un compromis entre la finesse des observations, le volume des données et le temps passé dans l'extracteur.

4.1.2.2. La sortie des données.

a. Sortie de données non analysées.

Elle permet un minimum de perturbations car les données sont très peu élaborées. L'analyse différée peut être très détaillée car elle travaille sur des données brutes.

b. Sortie de données deja élaborées.

Cela permet de réagir sur le système, et plus précisement sur sa charge, en cours d'exécution. Ceci peut se faire:

- soit automatiquement

L'extracteur fournit les données à un système de pilotage qui modifie les paramètres dynamiques du système en fonction des valeurs de celles-ci.

- soit indirectement

L'opérateur pouvant connaitre à tout moment l'état du système ou d'une tache donnée, a donc la possibilité de modifier la charge.

4.1.2.3. Architecture de l'extracteur.

La souplesse d'utilisation et d'extension est largement liée a l'utilisation d'une structure modulaire qui permet de ne charger que les collecteurs que l'on désire.

Il existe deux architectures possibles pour un extracteur:

a. L'extracteur peut être implanté sans modifier le système.

Les extracteurs de ce type sont basés uniquement sur l'echantillonage et sont considérés comme des taches indépendantes mais privilegiées du système.

b. Il peut également se présenter comme une série de modifications ou d'ajouts au système.

4.1.2.4. Les différentes phases d'une session de mesures.

a. L'initialisation

- détermination des mesures à effectuer

- introduction des sondes aux points de mesures définis

- actions spécifiques dues aux mesures lancées.

b. Extraction.

b. Terminaison.

- sortie d'enregistrements

- restauration de l'état antérieur du système.

4.2. Les outils d'extraction matériels.

On utilise un outil d'extraction matériel pour mesurer un système lorsqu'il s'est avéré que toutes les autres méthodes étaient insuffisantes en qualité.

Son domaine d'activité est l'espace physique constitué de l'unité centrale, des canaux et des différents périphériques qui lui sont connectés.

Malgré sa portabilité et le fait qu'il introduise une perturbation minimum, un tel outil est assez difficile à mettre en oeuvre car il exige une connexion physique au système. De plus, il est surtout adapté au comptage d'évènements et ne permet pas de prélever des résultats en mémoire centrale.

5. Outils d'analyse et de dépouillements des résultats.

Les informations fournies par une série de mesures ne sont jamais utilisables directement, ceci de par leur volume et leur nature (origines diverses, structure hétérogène, rangement séquentiel en fonction du temps, ...). De plus, il est difficile d'effectuer des calculs trop approfondis lors de la phase de mesures, à moins d'introduire une perturbation dans les performances du système.

En fait, il y a deux démarches possibles:

a. l'analyseur est incorporé dans l'extracteur.

Dans ce cas, les mesures fournissent des informations directement disponibles ce qui permet une possibilité d'action immédiate sur le système.

Par contre, les traitements effectués sur les mesures sont très rudimentaires, ceci à cause des perturbations.

b. l'analyseur est séparé de l'extracteur.

Il travaille donc en différé avec la prise de mesures et évite donc les perturbations.

Mais, il supprime la possibilité d'agir immédiatement sur le système.

En fait, l'idéal serait un analyseur construit en deux parties, l'une effectuant un depouillement élémentaire lors de la prise de mesures, l'autre effectuant un traitement plus approfondi à la demande de l'opérateur.

Les spécifications d'un bon outil d'analyse sont donc les suivantes:

- lors d'une première passe, à partir du fichier de mesures, on extrait les informations essentielles. Ces informations sont traitées et placées dans un autre fichier.

- ensuite, on peut procéder à une analyse plus détaillée de ces fichiers à l'aide de programmes classiques.

6. La charge

Les mesures de performances s'inscrivent toujours dans un certain environnement constitué du système informatique et de sa charge. Celle-ci peut être définie comme étant un ensemble de travaux mobilisant des ressources tant matérielles que logicielles.

Les résultats fournis par les mesures et la charge sont donc fortement liés car si celle-ci n'est pas représentative de ce qui se passe réellement dans le système, les résultats obtenus ne sont d'aucun intérêt et risquent en plus d'être mal utilisés.

Nous avons choisi d'effectuer les mesures sur la charge réelle et non sur une charge synthétique, ceci pour deux raisons:

- l'élaboration d'une charge représentative risquait de prendre un certain temps car elle nécessite une étude complète et précise de la charge réelle.

- la charge pouvant varier très fortement d'un jour à l'autre, notamment à cause des travaux pratiques effectués sur le système, il nous a semblé plus utile de mettre au point un outil de mesure qui pouvait être utilisé pour comparer les performances réelles à des moments et pour des charges différentes.

II. L'ENVIRONNEMENT EXPERIMENTAL

1. L'environnement matériel.

Nous disposions pour effectuer nos mesures du PDP 11/45 de DIGITAL EQUIPEMENT installé aux Facultés et possédant entre autres :

- le memory management unit (mémoire virtuelle)
- un processeur virgule flottante
- 128 Kbytes de mémoire core et de 128 Kbytes de mémoire MOS
- un contrôleur PMDC 11/80 relié a un disque PMDS 11/80 de 62,4 Mégabytes
- un contrôleur RK 10 relié à 3 disques cartouches RK 05 de 2,5 Mégabytes chacun
- un contrôleur de bande magnétique relié à 2 drivers TU 10 (l'un disposant d'une densité de 800 bpi et de 9 pistes, l'autre de 800 bpi également mais de 7 pistes)
- un contrôleur LP 11 relié à une imprimante PMLP /300
- 14 terminaux parmi lesquels:
  - 1 LA 34 (console)
  - 5 VT 100
  - 2 Microbee DM 20
  - 2 Northstar Horizon
  - 3 Apple 2
  - 1 Sanders varioprinter

2. L'environnement logiciel: UNIX.

2.1. Description générale de UNIX.

UNIX est un système d'exploitation en temps partagé, développé par les BELL Laboratories sur des minis-ordinateurs de la gamme de DIGITAL EQUIPEMENT.

Largement répandu dans les centres de recherches, UNIX dispose d'un important logiciel de base (compilateurs d'une douzaine de langages, formatteurs de textes, ...).

Ce système développé initialement pour les machines de la série PDP 11 a été étendu ensuite sur d'autres machines: VAX 11/780, INTERDATA 8/32, LSI 11/23,
Z 8000, ...

L'architecture de UNIX s'inspire des systèmes CTSS et MULTICS développés par le MASSACHUSSETS INSTITUTE OF TECHNOLOGY (MIT) et repose sur les concepts suivants:

- création dynamique de processus suivant la relation père-fils
- communication entre processus
- système de fichiers hiérarchisé, chaque fichier étant défini comme un ensemble de caractères (la notion d'enregistrement n'existe pas au niveau du système)
- les entrées / sorties sont synchronisées avec l'exécution du processus de traitement.

L'essentiel du langage est écrit dans un langage de haut niveau, la langage C. Celui-ci est assez proche du PASCAL (on retrouve les notions de types structurés; instructions "if-then-else", "while", "case",...) et il permet en plus l'arithmétique sur des objets de type pointeurs.

Nous allons maintenant nous attarder sur quelques aspects de UNIX qui nous semblent importants pour la bonne compréhension de la suite du mémoire.

2.2. Les processus.

Un processus correspond à l'exécution d'un programme en mémoire.

La totalité du programme doit être chargée en mémoire centrale pour qu'il puisse être exécuté et il y reste jusqu'à ce que l'arrivée d'un programme plus prioritaire entraîne son transfert sur disque (la gestion de la mémoire se fait au moyen de l'algorithme de "swapping" décrit au point 2.3.)

2.2.1. Mémoire allouée à un processus.

A un processus sont associées 3 zones de mémoire:

1. Un segment de texte contenant le code à exécuté.

Il est accessible uniquement en lecture et sa copie se trouvant sur disque restera donc à jour. Celà permet d'éviter le transfert de ce segment sur disque ("swap-out").

2. Un segment de données réservé à l'utilisateur.

La création d'un processus génère un certain nombre de données qui doivent rester strictement privées à celui-ci. Elles seront donc contenues dans un segment de données réservé au processus et à son utilisateur.

3. Un segment de données réservé au système.

Ce segment de longueur fixe est réservé aux données créées par le système et contient donc toutes les données dont le système doit disposer pour la gestion du processus correspondant (descripteurs de fichiers ouverts, ...).

L'utilisateur ne peut accéder à ce segment.

Les segments de données utilisateurs et systèmes sont placés de façon contigüe en mémoire afin de reduire le temps de transfert entre mémoire primaire et secondaire.

2.2.2. Arrivée d'un nouveau processus dans le système.

A l'exception de la phase initiale de chargement du système ("bootstrap"), tous les nouveaux processus sont crées par la primitive système "fork". Le nouveau processus est une copie du processus courant, c'est-à-dire de celui qui est actif au moment de la création.

En fait, lors de la création d'un processus, tous les segments accessibles en écriture appartenants au processus courant sont recopiés pour le nouveau processus. De même, les fichiers qui étaient ouverts pour le processus courant restent ouverts pour le nouveau.

Les processus "père" et "fils" sont ensuite informés de leur participation dans une relation et choisissent leur propre destination.

2.2.2.1. Réservation d'une entrée dans le tableau des processus.

Lorsqu'un nouveau processus est créé dans le système, un processus "système" se charge de parcourir le tableau des processus afin de vérifier s'il existe encore une entrée libre dans celui-ci. Ce tableau qui réside en permanence en mémoire centrale contient des informations utiles pour la gestion des processus (état du processus , priorité, ...).

Le nombre de processus que ce tableau peut contenir est

paramètrable. Sur la version de UNIX tournant aux Facultès ce nombre a ètè fixè à 50 et signifie donc que le système ne peut supporter plus de 50 processus en concurrence.

S'il n'y a plus de place disponible dans ce tableau, le système signale à l'utilisateur qu'il doit essayer de relancer sa commande plus tard (espèrant que d'autres processus aient terminès leur session d'ici là).

2.2.2.2. Allocation de mèmoire pour un processus.

Le processus nouvellement crèè demande la même part de mèmoire que celle qui a ètè attribuèe au processus courant.

S'il y a suffisamment de place en mèmoire centrale, celle-ci est attribuèe au nouveau processus et on signale dans le tableau qu'il est chargè en mèmoire et prêt à être exècutè. Sinon ce processus, ou plus exactement la copie du processus courant est transfèrèe sur disque.

Ensuite , le processus actif au moment de la crèation redevient actif.

2.3. La gestion de la mèmoire.

En gènèral, il y a trop peu de place en mèmoire centrale pour contenir toutes les donnèes associèes aux diffèrents processus. Donc , il faudra parfois transfèrè des zones de donnèes sur disque.

L'allocation de la mèmoire centrale et secondaire est rèalisèe par le même algorithme "first-fit", c'est-à-dire que dès que l'on trouve une zone de mèmoire assez grande pour contenir le code et les donnèes associèes au processus, on la lui alloue. L'autre algorithme utilisè par certains systèmes d'exploitation est celui du "best-fit" (on optimise l'allocation car on choisit une zone mèmoire suffisante mais la plus petite possible).

Description du processus de transfert (ou processus de "swapping").

C'est un processus "système" qui gère les transferts entre la mèmoire centrale et le disque. Celui-ci examine la table des processus cherchant un processus se trouvant sur disque et qui serait prêt à tourner s'il disposait de la mèmoire centrale. Il lui alloue de la mèmoire et effectue le transfert en mèmoire où il pourra maintenant se disputer l'usage du processeur avec d'autres processus.

S'il n'y a pas de mèmoire centrale disponible, le processus de "swapping" va essayer d'en libèrer en choisissant dans la table un processus pouvant être transferè sur disque. Ce processus est transferè sur disque ("swap-out") et l'algorithme de "swapping" en choisit alors

un autre à transférer en mémoire centrale ("swap-in").

Il y a donc deux algorithmes spécifiques au processus de "swapping":

1. Quel est parmi les processus se trouvant sur disque celui qui est susceptible d'être transféré en mémoire centrale afin d' y être exécuté?

   Le processus le plus ancien et dont l'état est prêt est choisit le premier. Il y a une légère pénalité liée à la taille.

2. Quel est parmi les processus chargés en mémoire centrale celui qui est susceptible d'être transféré sur disque afin de libérer de la place?

   Les processus en attente d'événements à faible priorité(c'est-à-dire les processus qui ne sont pas actifs ou qui n'attendent pas la fin d'une entrée/ sortie)sont choisis les premiers, les plus anciens d'abord en tenant compte également de pénalités liées à la taille. Les autres processus sont examinés par le même algorithme mais on ne les retire pas de la mémoire avant qu'ils aient atteints un certain âge.

Le processus de "swapping" n'a aucun impact sur l'exécution des processus résidants.

Le processus de "swapping" est donc équivalent a un serveur FIFO.

2.4. Le "scheduling" du processeur.

La synchronisation des processus est realisée au moyen d'événements.

La signalisation d'un événement qui n'est attendu par aucun processus n'a aucun effet. De même, la signalisation d'un événement attendu par plusieurs processus réveille tous ces processus.

Comme il n'y a pas de mémoire associée aux événements(contrairement aux opérations P et V de Dijkstra), on ne peut ajouter une notion de "grandeur" à ceux-ci. Supposons, par exemple, que des processus désirant de la mémoire attendent un événement associé à l'allocation de la mémoire. Lorsque une partie de la mémoire sera liberée, l'événement sera signalé. Tous les processus en compétition pour la mémoire seront donc reveillés et se disputeront la nouvelle part de mémoire alors qu'en fait elle ne sera peut être pas assez grande pour n'importe lequel de ces processus.

Si un événement se produit entre l'instant ou un processus

décide de se mettre en attente et l'instant où ce processus se met en attente, il attendra un événement qui s'est déjà produit et qui risque de ne plus se reproduire. Ceci est du au fait qu'il n'y a pas de mémoire associée a l'événement, mémoire qui permettrait d'indiquer que l'événement s'est déjà produit; la seule action résultant d'un événement est la modification de l'état des processus.

Supposons maintenant qu'un processus actif se mette en attente d'un événement, le processus qui disposera maintenant du processeur sera un processus dont l'événement aura déjà été signalé et qui est aussi a l'état prêt.

Quel processus choisir parmi les processus qui sont prêts?

A chaque processus est associé une priorité. La priorité des processus "systèmes" est assignée par un certain code liés aux événements. Les événements associés a tout ce qui concerne le disque ont une priorité elevée, les événements associés aux vidéos ont une priorité moins elevée et ceux associés au temps la priorité la plus faible. Les priorités des processus "utilisateurs" sont toutes moins elevées que la moins elevée des priorités associées aux processus "systèmes".

La priorité associée aux processus utilisateurs est le rapport temps d'exécution sur temps réel pour un certain intervalle de temps. Un processus qui a utilisé beaucoup le processeur lors du dernier intervalle de temps réel aura donc une faible priorité.

L'algorithme de "scheduling" choisit d'abord les processus les plus prioritaires ,exécutant d'abord les processus "systèmes" et ensuite les processus "utilisateurs". Les ratios "temps d'exécution / temps réel" sont mis a jour toutes les secondes.

L'algorithme de "scheduling" s'arrange aussi pour qu'un processus disposant d'une priorité elevée ne monopolise pas le processeur. En effet dans ce cas sa priorité decroit. De même si un processus à faible priorité est ignoré depuis longtemps, sa priorité va s'accroitre.

Les processus chargés de la gestion du système peuvent interrompre le processeur a tout instant car ils exécutent des tâches hautement prioritaires. Mais, en terme de théorie des files d'attentes, on peut néanmoins considérer le processeur comme étant un serveur à discipline prioritaire sans préemption. En effet, les processus "systèmes" se placeront en avant dans la file d' attente devant le processeur mais n'interrompront pas le service en cours.

2.5. Les entrées/sorties.

Les entrées sorties dans UNIX sont conçues pour éliminer les

différences entres les divers types de périphériques et méthodes d'accès.

Du point de vue de l'utilisateur, les opérations de lecture et d'écriture de fichiers sont des opérations synchrones et immédiates (sans tamponnage).

Mais, en réalité, le système assure une gestion complexe de tampons pour réaliser une mémoire cache entre la mémoire centrale et le disque. Cette mémoire cache permet d'augmenter l'efficacité du système en conservant en mémoire les zones auxquelles on accède fréquemment.

La cache est gérée de façon à retarder les écritures et si le système s'arrête subitement, il y aura des entrées/sorties logiquement complètes mais physiquement incomplètes. Il existe une primitive qui permet la mise à jour de la mémoire secondaire mais une utilisation fréquente de celle-ci ne résoud que partiellement le problème. Il y aura donc des moments où les structures de données sur disques sont incohérentes même si le logiciel exécute les entrées/sorties dans le bon ordre.

En fait,UNIX dispose de deux systèmes d'entrée/sortie, un système d'entrée/sortie par blocs (ou structuré) et un autre par caractère (ou non structuré).

Le système d'entrées/sorties par caractère regroupe tous les périphériques qui ne tombent pas dans la classe des entrées/sorties par blocs: il s'agit donc de tous les périphériques typiquement caractères tels que les bandes papier perforées, les vidéos les imprimantes mais aussi les bandes et les disques qui ne sont pas utilisés de manière stéréotypée (par exemple des enregistrements de 80 bytes sur bande).

La désynchronisation des entrées / sorties physiques due à la présence de la mémoire cache a très peu d'effets sur la discipline des files d'attente des serveurs d'entrées / sorties qui peut être considérée comme étant de type FIFO

III. LE MODELE.

1. Objectifs du modèle.

Le but des mesures étant l'étude du système d'exploitation UNIX soumis à une charge type universitaire, nous allons élaborer un modèle de capable de nous fournir des résultats représentatifs du comportement de celui-ci. Ces résultats seront des estimations qu'il nous faudra encore comparer avec les mesures afin de valider ou de rejeter le modèle.

L'outil choisi est le modèle mathématique plutôt que le simulateur car il nous a semblé plus facile à mettre au point vu que certaines formules de modèles de réseaux de files d'attente se retrouvent telles quelles dans la littérature.

Lorsque nous disposerons d'un modèle fiable, c'est-à-dire quand il fournira des valeurs proches de celles mesurées, nous pourrons utiliser ses résultats comme étant représentatifs de la réalité.

Suivant les valeurs fournies, nous pourrons peut-être envisager de faire varier certains paramètres du système afin d'étudier l'évolution de son comportement.

D'autres modifications sont également envisageables, comme par exemple la redistribution de l'espace disque parmi les différentes catégories d'utilisateurs en fonction des taux d'utilisations des "pseudos-disques".

2. Représentation du modèle.

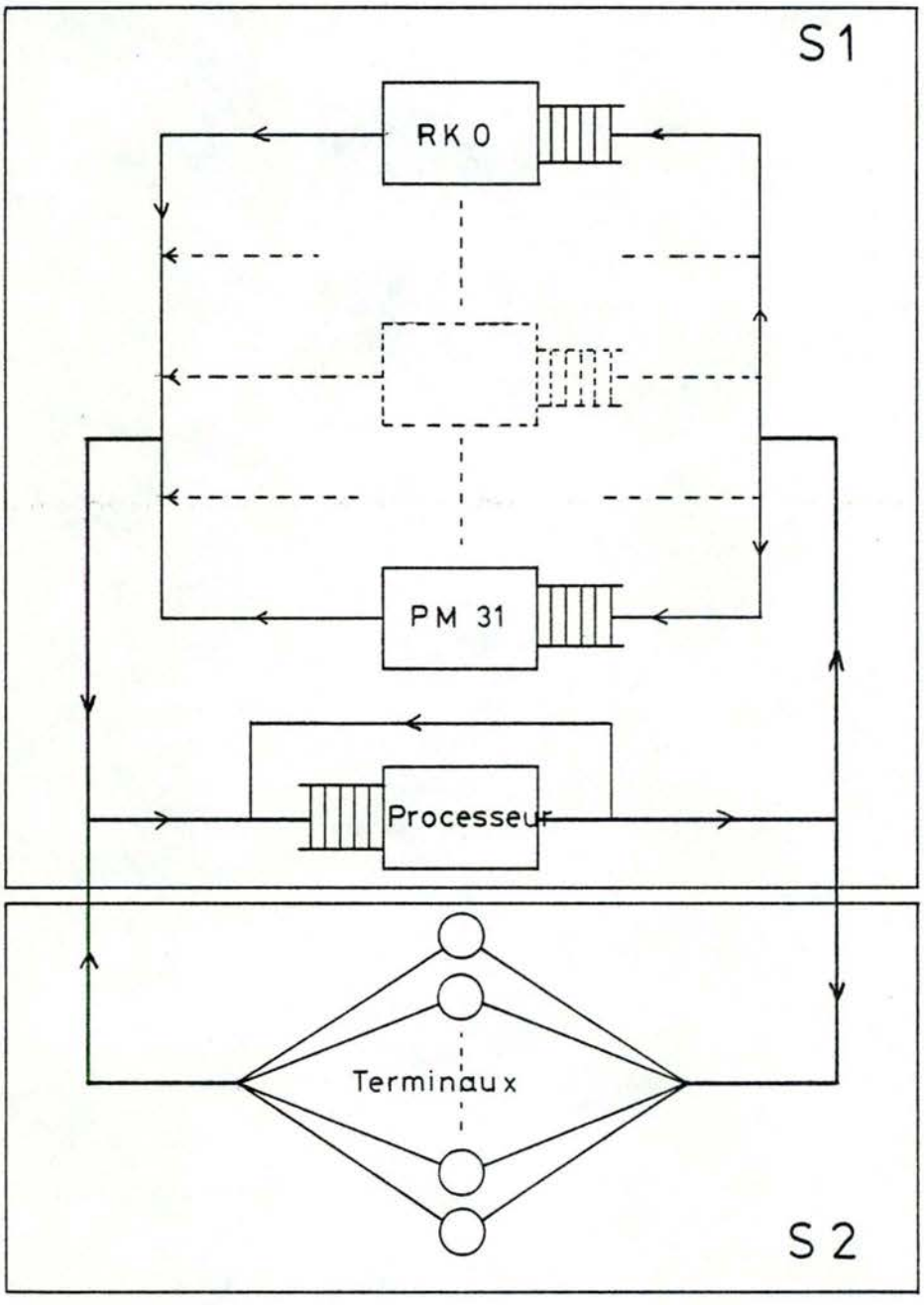


fig. 3.1.

Nous avons choisi de représenter le modèle sous forme d'un graphe orienté car il s'agit de la manière la plus simple de représenter un réseau de files d'attente. Les rectangles représentent les serveurs et les arcs indiquent les différents chemins que les processus peuvent suivre lorsqu'ils se déplacent dans le réseau.

Ce modèle représente en fait le comportement d'un processus dans un univers de multiprogrammation: il est caractérisé par une alternance de périodes d'utilisation processeur et de périodes d'entrées / sorties, précédées en général par des attentes devant ces serveurs.

Les processus lancés à partir des terminaux passent exclusivement par le processeur qui sera donc l'intermédiaire par lequel se feront les échanges entre les terminaux et le reste du système: nous appelerons donc le processeur serveur d'échanges.

Les différentes transitions entre les serveurs sont dues aux particularités du scheduling et des entrées / sorties de UNIX.

3. Notions de la théorie des files d'attentes.

Le modèle que nous nous proposons d'élaborer sera constitué d'un certain nombre de serveurs (processeur, entrées-sorties, défaut de pages, ...), de files d'attente précédant les serveurs et de chemins de transitions entre ceux-ci.

- Le serveur.

Il s'agit d'une ressource non partageable du système.

Dans notre étude, nous avons considéré que le disque PM était constitué de plusieurs serveurs car chacun de ceux-ci sert en fait un espace physique différent: le "pseudo-disque" constitué d'un certain nombre de cylindres. Ceci nous permettra de mesurer les taux d'utilisations du contrôleur de chaque "pseudo-disque" ainsi que le nombre d'accès à chacun de ceux-ci.

- La file d'attente.

Plusieurs demandes de services peuvent se présenter lorsqu'un serveur est déjà requis par un autre processus(le serveur est à l'état actif), on adjoint donc au serveur une file d'attente dans laquelle viendront se placer tous les processus dont il ne pourra satisfaire la demande de service immédiatement.

Les files d'attente pourront être gérées de façon différente suivant le serveur auquel elles appartiennent. Parmi les disciplines de files d'attente nous citons les plus couramment utilisées:

i. FIFO(first in / first out).

Les processus sont servis dans l'ordre de leur arrivée dans la file d'attente.

ii. LIFO(last in / first out).

Les processus sont servis dans l'ordre inverse de leur arrivée dans la file d'attente c'est-à-dire que le dernier processus arrivé est le premier servi.

iii. Priorité.

- non préemptive

Les processus de plus haute priorité seront servis d'abord, mais sans interrompre le service en cours.

- préemptive

Les processus disposant de la priorité la

plus élevée seront servis les premiers et les nouveaux processus entrants dans la file peuvent interrompre le service du processus en cours s'ils disposent d'une priorité plus élevée que celui-ci.

Il existe bien sur un algorithme qui met à jour les priorités des processus en attente, ceci afin d'éviter qu'un processus reste éternellement dans la file.

- Réseau de files d'attente.

Lorsque les processus doivent passer successivement dans plusieurs systèmes composés d'un serveur et d'une file d'attente, nous nous trouvons devant un réseau de files d'attente. Celui-ci est caractérisé par sa structure, c'est-à-dire par les liaisons qui existent entre les sous-systèmes qui le composent et, par la nature de ces sous-systèmes.

On peut distinguer principalement deux types de réseaux:

- réseau ouvert

Un réseau est ouvert lorsqu'il est alimenté par une source extérieure disposant d'un nombre infini de clients.

- réseau fermé

Un réseau est fermé lorsque le nombre total de processus dans le système est fini et, en général constant.

La théorie des files d'attente nous permet d'estimer un certain nombre de grandeurs caractéristiques du système observé, comme par exemple:

- le taux d'utilisation des différents serveurs c'est-à-dire le temps d'activité du serveur par rapport au temps total d'activité du système.

- le nombre moyen de processus se trouvant dans les diverses files d'attente.

- etc...

4. Le modèle.

4.1. Hypothèses.

Nous devons poser plusieurs hypothèses avant de pouvoir nous lancer dans l'étude du modèle:

- nous avons à faire à des serveurs:

  + obéissants à une discipline FIFO.

  + jacksonniens

    Un serveur est jacksonnien lorsque son débit dépend uniquement du nombre de clients se trouvant dans sa file d'attente.

  + exponentiels.

    Un serveur est exponentiel lorsque les durées de service sont indépendantes les unes des autres et distribuées suivant une loi exponentielle.

- il existe un régime permanent

  Si $p_n(t)$ est la probabilité de l'état n à l'instant t,

  avec t = 0, 1, 2, ...

  Alors, lorsque $t \to \infty$

  $\lim p_n(t) = p_n \quad n = 0, 1, 2, \ldots$

  avec l'un au moins des p_n non nul.

  $\Sigma\ p_n = 1$

4.2. Décomposition du modèle.

4.2.1. Notion de décomposition et de système équivalent.(B[3])

Système équivalent.

A l'intérieur d'un même réseau, deux systèmes d'attente sont équivalents si les distributions conjointes du nombre de clients dans chacun de ces systèmes et du nombre total dans les autres centres du réseau sont identiques.

Décomposition.

Un système est presque complètement décomposable si les interactions entre sous-systèmes sont suffisamment faibles

4.2.2. Substitution du modèle par un modèle équivalent.

Les interactions internes du système composé du serveur d'échange et des serveurs d'entrées / sorties sont nombreuses comparées aux interactions de ce système avec le monde extérieur(cfr annexe 1: les probabilités de transitions), nous allons donc décomposer le système global en deux sous-systèmes:

- les terminaux d'une part (système S2).
- le serveur d'échanges et les serveurs d'entrées / sorties d'autre part(système S1).

4.2.2.1. Etude du système global.

Le système global de la figure 3.1. sera remplacé par celui de la figure 3.2.

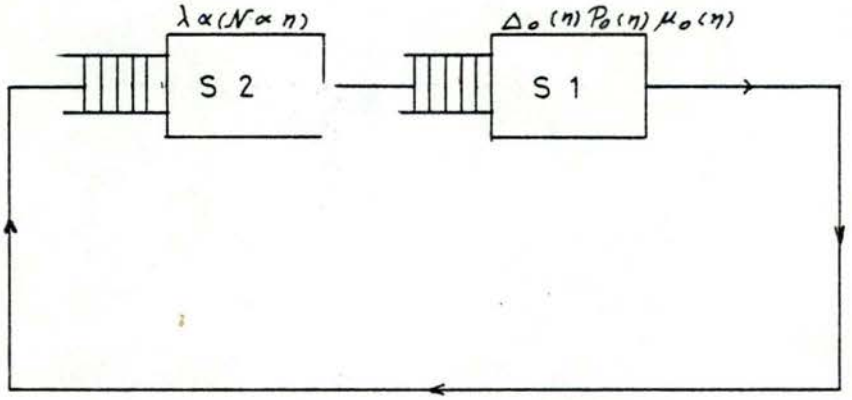


fig. 3.2.

Nous allons, dans cette décomposition, supposé que l'ensemble des terminaux puisse être remplacé par un serveur unique de taux de service $\lambda * (N - n)$ avec

$1/\lambda$: temps de réflexion au terminal.

N - n: nombre de terminaux en cycle de réflexion.

En général, les auteurs considèrent que l'on peut attacher au plus un processus à un terminal. Ceci n'étant pas le cas avec UNIX, nous choisissons:

N = 50 (nombre d'entrées dans la table des processus)
n = nombre de processus dans le système S1

Ainsi, (N - n) représentera le nombre de processus pouvant encore se présenter dans S1 sans bloquer le système. Nous considérons que cela fournit une bonne représentation du nombre de terminaux "logiques" se trouvant en cycle de réflexion.

Le sous-système S1 composé des serveurs d'entrées / sorties "jacksonniens" et du serveur d'échange exponentiel de débit de service $u_0(n)$, avec un processus d'arrivée poissonien de paramètre (n), lorsqu'il y a au total n processus dans le système, est équivalent, à l'état stationnaire, s'il existe, à un seul serveur exponentiel de taux de service $A_0(n)p_0(n)u_0(n)$

avec $A_0(n)$: probabilité conditionelle stationnaire que le serveur d'échange soit actif étant donné que le nombre de processus dans le sous-système est n.

$p_0(n)$: probabilité que le processus qui finit son service au serveur d'échange quitte le système lorsque le nombre total de processus présents dans ce dernier est n.

Ce théorème est démontré dans (B[3]).

Nous obtenons alors un réseau de 2 files d'attente exponentielles de taux de service dépendant de n et nous pouvons calculer la ditribution de probabilité de ce réseau(cfr. B[14]).

$$p(n) = \frac{1}{H(N)} \frac{(\lambda c)^n}{(N-n)! \prod_{i=1}^{n} A_0(i)}$$

avec c: temps moyen de calcul d'un processus

$$H(N): \sum_{n=0}^{N} \frac{(\lambda c)^n}{(N-n)! \prod_{i=1}^{n} A_0(i)}$$

Le temps moyen de rèponse dans ce système est donnè par :

$$\bar{t} = \frac{\bar{n}}{\lambda * (N - \bar{n})}$$

4.2.2.2. Etude du système S1.

Si le système S1 interagit faiblement avec son environnement, nous pouvons considèrer que la probabilitè conditionnelle d'activitè du serveur d'èchanges $A_0(n)$ ne devrait pas être trop diffèrente de la probabilitè que le serveur d'èchanges soit actif dans un rèseau fermè obtenu en coupant les liens entre S1 et le monde extèrieur.

Les èquations du rèseau de la fig. 3.1. s'obtiennent facilement grâce à l'hypothèse que les temps de services sont des variables alèatoires indèpendantes et distribuèes selon une loi exponentielle (serveurs exponentiels).

Nous ne dèvelopperons pas ces èquations qui, sous forme matricielle s'ècrivent:

$P(t+1) = P(t) * Q$

avec P(t) : vecteur des probabilités d'état $p(n_0,...,n_T,t)$
(n_i = nombre de processus au serveur i, i=0,..,l
et n_T = nombre de processus au serveur T
T représentant le serveur des terminaux
et l le nombre de serveurs)

Q = A + I, A étant la matrice des taux de transitions
et I la matrice unité.

Chaque valeur de l'indice j (j = 1,...,$\binom{l+1+N}{l+1}$) de la matrice Q se rapporte à un état différent (n_0 ,n_1 ,...,n_l ,n_T) du réseau. Nous choisissons un arrangement des lignes et des colonnes de Q tel que n soit la variable qui varie le plus lentement dans (n_0,n_1,...,n_l,n_T), c'est-à-dire que nous considérons d'abord tous les états possibles avec n_T= 0, ensuite avec n_T= 1, ...

Nous désignons par Q(N,K) , K = l + 1, N > 0, K > 1 la matrice Q d'ordre $\binom{K}{K+N}$ définie par cette numérotation des états.

Voici maintenant 2 lemmes et un théorème démontrés dans (B[3]):

Lemme 1.

Les éléments de la matrice Q(N,K) peuvent être partagés en (N+1) sous- matrices principales $Q'(n_T)$,n_T = 0,...,N de façon que l'ensemble des éléments non diagonaux de chaque sous-matrice $Q'(n_T)$, n_T= X, soit l'ensemble de toutes les probabilités de transition entre tout couple d'états distincts parmi les $\binom{K+1+N-X}{K-1}$ états avec n_T = X, c'est-à-dire n = N - X.

Lemme 2.

Tout élément non diagonal de Q(N,1) situé en dehors des sous-matrices principales Q'(n) vaut soit zéro,

soit $\alpha(n_0 + 1) * u_0(n + 1) * p_{0T}(n + 1)$
ou bien $\alpha(n_T + 1) * \lambda_0(n_T + 1) * (1 - p_{TT}(n_T + 1))$

avec $\alpha(j) = 0$ si j = 0 ou N+1
= 1 sinon

où n = N - n_T,n_T correspondant à la colonne considérée dans la matrice.

Théorème.

Si

$$\alpha(n_T)\lambda(n_T)(1 - p_{TT}(n_T)) + \alpha(n_0)u_0(n)p_{0T}(n)$$
$$<< \sum_{i=1}^{K} \alpha(n_i)u_i(n_i)(1-p_{ii}(n_i))+(n_0)u_0(n)\sum_{m=1}^{K} p_{om}(n)$$
$$\text{pour } n_T = 0 , \ldots , N$$

La matrice stochastique Q(N,K), N > 0, K > 1, définit un système presque complètement décomposable en (N+1) systèmes qui peuvent être représentés par des matrices stochastiques Q(X) d'ordre $\binom{N+K-X-1}{K-1}$, X = 0,...,N correspondant à toutes les interactions possibles entre les états $(n_0,\ldots,n_k)$ avec n_T= X.

Ce théorème fournit une condition suffisante pour considérer le réseau de la fig. 3.1. comme étant presque complètement décomposable. Les matrices stochastiques Q(X) décrivent en fait le comportement d'un réseau fermé obtenu en supprimant les liens entre les sous-systèmes S1 et S2. Nous déduisons de la définition des systèmes presque complètement décomposables que la probabilité conditionnelle $A_0(n)$ que le serveur d'échanges soit actif étant donné qu'il y a n clients dans le sous-système S1 est approximativement égale à la probabilité que le serveur d'échanges soit actif dans le réseau fermé décrit par Q(N-n) (cfr fig. 3.3.). Le calcul de cette probabilité ne pose aucun problème le réseau correspondant à Q(N-n) étant Jacksonnien.

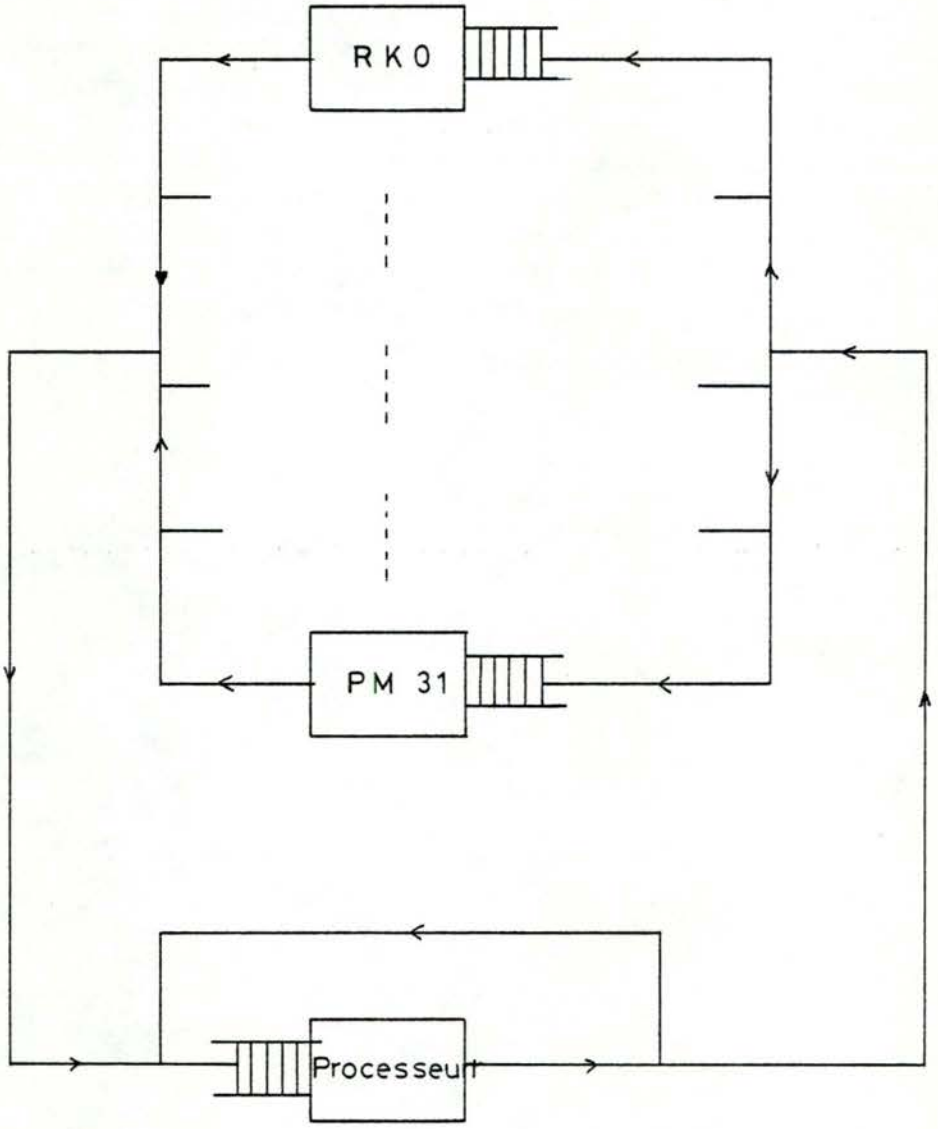


fig. 3.3.

Nous allons donc étudier ce modèle et nous intéresser tout particulièrement aux probabilités d'activité de ses différents serveurs.

- calcul de la probabilité d'activité du serveur d'échange.

Posons A_0, la probabilité que le serveur d'échange soit occupé.
Si $p(n_0, n_1, n_2 \ldots, n_l)$ est la probabilité qu'il y ait conjointement n processus au serveur d'échange et $n_i (1 \leq i \leq l)$ aux autres serveurs.

$$\text{avec } p(n_0, n_1 \ldots, n_l) = 0 \text{ si } \Sigma n_i = N$$
$$\text{et } \sum_{\substack{n_0, n_1, \ldots, n_l \\ \Sigma n_i = N}} p(n_0, n_1 \ldots, n_l) = 1$$

Le serveur est actif s'il y a au moins un processus dans sa file d'attente , ce qui peut se traduire en termes de probabilités par:

$$A_0 = \sum_{n_0 \geq 1} p(n_0, n_1, \ldots, n_l)$$

$$= \sum_{\substack{l \\ \Sigma n_i < N - 1 \\ i = 1}} p(n_0, n_1, \ldots, n_l)$$

Or, le théorème de Jackson nous dit que dans un réseau exponentiel, en régime permanent, la superposition des régimes permanents des systèmes composants donne encore un régime permanent global.

Nous avons donc:

$$A_0 = \sum_{\sum_{i=1}^{l} n_i < N-1} \frac{1}{G(N)} \prod_{i=1}^{l} \left(\frac{p_i u_0}{u_i} \right)^{n_i}$$

$$\text{car } p(n_0, n_1, \ldots, n_l) = \frac{1}{G(N)} \prod_{k=1}^{l} \left(\frac{u_0 p_i}{u_i} \right)^{n_i}$$

$$A_0 = \frac{G(N-1)}{G(N)}$$

avec u_i : débit de service du serveur i
p_i : probabilité de choisir le ième serveur après un service au serveur d'échange.

$$G(N) = \sum_{\sum_{i=1}^{l} n_i \leq N} \prod_{i=1}^{l} \left(\frac{p_i \, u_0}{u_i} \right)^{n_i}$$

(Constante de normalisation)

l : nombre de serveurs(36)
N : nombre de processus dans le système fermé.

- les autres serveurs.

Posons A_j, la probabilité d'activité du serveur j ($1 \leq j \leq l$).

$$A_j = \sum_{n_j \geq 1} p(n_0, n_1, \ldots, n_l)$$

$$= \sum_{\substack{\sum n_i \leq N \\ n_j \geq 1}} \frac{1}{G(N)} \prod_{i=1}^{l} \left(\frac{p_i u_0}{u_i} \right)^{n_i}$$

$$A_j = \frac{1}{G(N)} \left(\frac{p_j u_0}{u_j} \right)^{\ell} \sum_{i=1}^{36} n_i < N-1 \prod_{i=1} \left(\frac{p_i u_0}{u_i} \right)^{n_i}$$

$$= \left(\frac{p_j u_0}{u_j} \right) \frac{G(N-1)}{G(N)} = \left(\frac{p_j u_0}{u_j} \right) A_0$$

Ce qui fournit la loi de conservation des flux qui est vérifiée même si les serveurs ne sont pas exponentiels.

$$A_j \, u_j = A_0 \, u_0 \, p_j$$

- calcul de la constante de normalisation.

Elle peut être calculée aisément en utilisant l'algorithme suivant(cfr. B[14]).

Soit $\vec{n} = (n_0, n_1, \ldots, n_l)$

$$X_i = \frac{p_i u_0}{u_i}$$

et $S(N,l) = \left\{ (n_0, n_{1=/=} \ldots, n_l) : \sum_{i=0}^{l} n_i = N \text{ et } n_i > 0 \ \forall i \right\}$

Avec ces notations:

$$G(N) = \sum_{\vec{n}\ S(N,l)} \prod_{i=1}^{l} X_i^{n_i}$$

Notons $g(n,m) = \sum_{\vec{n} \in S(n,m)} \prod_{i=1}^{m} X_i^{n_i}$

Alors $g(n,l) = G(N)$

et $g(n,m) = g(n,m-1) + X_m g(n-1,m)$

avec $g(0,m) = 1 \quad m = 1, \ldots, l$

$g(n,0) = 1 \quad n = 0, \ldots, N$

La principale difficultè consistera à èvaluer N, car il est peu probable que le nombre de processus dans le système rèel soit constant. Nous pensons donc que le nombre moyen de jobs dans le système pendant la pèriode d'observation peut fournir une bonne approximation de N (nous pouvons également travailler avec un intervalle de valeurs centrès en N).

4.2.2.3. Etude du système S2.

Nous supposons que l'ensemble des terminaux puisse être remplacè par un serveur unique de taux de service $\lambda * (N-n)$.

Ceci reste à vèrifier mais cette hypothèse peut trouver une justification dans le fait que les terminaux constituent tous des postes sans attente : nous pensons donc pouvoir les remplacer par un seul poste sans attente.

IV. LE MESUREUR.

1. Objectif.

Les mesures réalisées sur un système réel au moyen d'un mesureur permettent , tout comme celles obtenues avec un modèle , d'étudier les performances. De plus, si celà s'avère nécessaire , le système peut exploiter celles-ci dynamiquement (exemple : suppression des fichiers temporaires lorsque l'espace disque atteint un certain seuil).

Le modèle élaboré étant destiné à nous fournir des résultats aussi proches que possibles de la réalité , le but du mesureur sera de vérifier la validité du modèle par comparaison des résultats du mesureur et de ceux du modèle.

2. Nature des mesures.

Les données élémentaires collectées par les mécanismes de mesures sont , pour la plupart , des contenus de mémoire , des événements ou des comptages d'événements.

Nous allons nous attarder principalement sur 3 groupes de données:

1. Intervalles de temps.

Pour mesurer par exemple le taux d'utilisation des différents serveurs , nous mesurerons les intervalles d'activité des serveurs ainsi que le temps total nécessité par les mesures.

Le rapport temps d'activité du serveur sur le temps d'activité du mesureur nous fournira le taux d'utilisation du serveur.

2. Grandeurs instantanées autres que le temps.

Parmi celles-ci , nous pouvons par exemple inclure la longueur de la file d'attente du processeur.

3. Dénombrements dans le temps.

Il s'agit de fréquences d'événements dans le temps comme par exemple les transitions entre les différents serveurs.

3. Mécanisme des mesures.

Pour d'évidentes raisons de simplicité , nous n'utiliserons pas les mesures cablées et nous mettrons donc au point un extracteur logiciel ou "programmé" qui nécessitera bien sûr une connaissance approfondie du système.

Le mesureur se présentera comme une série d'ajouts ou de modifications à UNIX et nous permettra d'accéder à un certain nombre de variables du système.

Nous distinguerons deux méthodes d'acquisition de données :

- les mesures non synchronisées avec les changements d'états du système (ou activation des sondes par échantillonage).

  Ce sont les mesures périodiques comme celle mesurant le nombre de processus se trouvant dans le système et la longueur de la file d'attente du processeur (valeurs mesurées toutes les 90 secondes).

- les mesures synchronisées avec l'évolution du système (ou act vation des sondes sur évènements).

  L'évolution du système est caractérisée aussi bien par les changements d'états (synchronisation des mesures des temps d'activité sur l'état de ceux-ci : le compteur n'est incrémenté que lorsque le serveur est actif) que par les transitions des processus d'un serveur à l'autre (probabilités de transition).

Les deux méthodes seront donc utilisées dans le mesureur et le choix de l'une plutôt que l'autre dépendra de la fréquence demandée ainsi que de la nature de la mesure.

4. Les grandeurs à mesurer.

- pour utiliser le modèle.

+ Nombre de processus dans le système.

Ayant supposé que nous nous trouvions dans un système fermé , le nombre de processus dans le système devrait rester constant.

Ceci peut parfois se vérifier en période d'utilisation normale mais en cas de fortes charges se nombre peut varier considérablement.

Nous considérerons donc le nombre moyen de processus durant la période de mesures comme étant représentatif du nombre de processus dans le système fermé.

+ Probabilités de transition (p_i).

Les différents chemins de transitions entre serveurs comprendront des compteurs qui seront incrémentés chaque fois qu'un processus passera par un de ces chemins.

Les probabilités de transition sont obtenues en divisant la valeur du compteur associé à un chemin par la somme des compteurs de tous les chemins.

+ Taux ou débits de service (u_i).

Le taux de service d'un serveur correspond au nombre de processus servis par celui-ci par unité de temps.

Nous comptabiliserons donc le nombre d'activations du serveur et diviserons celui-ci par le temps nécessité pour les mesures.

+ Taux d'utilisation.

Ils sont fournis par le rapport temps d'activité du serveur sur temps nécessité pour les mesures.

+ Temps de réflexion au terminal.

Il représente le temps mis par l'utilisateur pour préparer son prochain message après réception du précédant.

- pour vérifier le modèle.

+ Longueur de la file d'attente du processeur.

Elle sera utile pour vérifier si nous nous trouvons bien dans le cas d'un régime permanent (cfr V) et est mesurée en sélectionnant les processus disposant d'un certain état et de certaines "étiquettes" dans le tableau des processus.

+ Périodes d'activité des différents serveurs.

Ces valeurs seront utilisées pour vérifier si les serveurs sont exponentiels. (cfr. V)

5. Introduction des sondes.

Nous avons introduits , aux points de mesures définis précédemment , un certain nombre de sondes destinées à nous fournir des renseignements sur le comportement du système. Celles-ci sont essentiellement des compteurs et des "étiquettes" nous permettant de mesurer les grandeurs définies au point 4.

Cette étape a necessité une étude complète et détaillée de UNIX car l'introduction des sondes a entraîné un certain nombre de modifications du système.

UNIX étant composé d'une série de programmes qui sont ensuite compilés ensembles afin de fournir une version exécutable , nous dressons ici une liste indicative des programmes auxquels nous avons apporté des modifications. Les personnes désirant des renseignements complémentaires sont libres de consulter les sources du système.

Main.c : diverses initialisations de UNIX
Slp.c : scheduling.
Clock.c : gestion des interruptions en provenance de l'horloge.
Sys1.c : appels systèmes.
Bio.c : gestion des entrées / sorties par blocs.
Pm.3.c : contrôleur du disque PMDS 11/80.
Rk.c : contrôleur des disques RK 05.
Tty.4.c. : contrôleur des terminaux.
Alloc.c : allocation des espaces disques.
Fio.c & rdwri.c

6. Lancement du mesureur.

Comme le mesureur fait partie de UNIX , il sera donc réinitialisé à chaque redémarrage de celui-ci.

L'acquisition des données se fera donc tout au long de la période d'activité du système. L'accès au PDP 11/45 n'étant pas autorisé durant la nuit , les différentes sondes seront réinitialisées lors du redémarrage matinal , la collecte des résultats se fera le soir lors de la mise en veilleuse , ou si besoin , durant la journée.

Nous pourrions également mesurer le système pour un intervalle de temps donné , par exemple lors des travaux pratiques , car l'initialisation des sondes et la collecte des résultats peuvent avoir lieu n'importe quand.

7. Collecte des résultats.

Le système d'exploitation UNIX dispose d'un fichier (/dev/kmem) qui est une représentation du contenu de la mémoire attribuée au système. En accédant à ce fichier , nous pouvons donc connaître à tout moment la valeur des variables du système.

Le programme collectant les résultats du mesureur pourra donc sélectionner les valeurs qui l'intéressent dans ce fichier , il effectuera certains traitements sur celles-ci et , ensuite , passera la main au modèle. La perturbation introduite à ce niveau sera minimale car nous ne risquons pas de modifier le fonctionnement du système étant donné que le programme de collecte s'exécute comme un programme ordinaire.

Les valeurs fournies par les mesures et les valeurs estimées par le modèle seront placées dans un fichier afin de pouvoir leur appliquer des tests statistiques plus complets.

8. Perturbation introduite par les mesures.

La perturbation introduite par un mesureur est fonction de la durée et de la fréquence des mesures.

Ces deux données n'étant pas faciles à observer étant donné la nature du mesureur , nous ne nous sommes pas attardé d'avantage sur ce point , constatant de visu que le système était fiable et que la surcharge introduite n'apparaissait pas à l'usager derrière son terminal.

V. VERIFICATION DES HYPOTHESES RELATIVES AU MODELE.

1. Introduction

Lors du choix de notre modèle , nous avons été obligés de poser un certain nombre d'hypothèses dont il nous reste à vérifier la véracité. Celà sera chose faite dans ce chapitre cinq.

Il nous semble utile , à ce stade de notre étude de faire une mise en garde . En effet , malgré le bien-fondé des hypothèses , la réalisation et la vérification de celles-ci ne doit pas être un but en soi.

Dans le cas où elles ne seraient pas vérifiées , nous exécuterons malgré tout notre modèle et l' un des attraits de celui-ci sera alors de nous amener à réfléchir aux divergences entre les mesures et les estimations ainsi qu'à leurs causes.

2. Le régime permanent.

En termes de théorie des files d'attente et d'après la définition du régime permanent (cfr. III.4.1) , nous pouvons considérer :

soit x : longueur d'une file d'attente du réseau mesurée en un instant t quelconque.

si Ex (longueur moyenne de la file) , var(x) et $cov(x_t, x_{t+s})$ sont indépendantes du temps

alors , nous avons un processus stationnaire.

Nous avons donc mesuré la taille de la file d'attente d'un certain serveur , en l'occurence le processeur . Nous avons choisi celui-ci car il nous semblait le plus représentatif de l'état du système et car il suffisait d'accéder au tableau des processus pour comptabiliser le nombre de processus composant sa file d'attente.

La longueur de cette file est mesurée régulièrement à intervalles constants. A la fin de la journée , nous calculons la longueur moyenne aux différents instants t , t + 60' , t + 120' avec t = 0 , i , i + i , ... (i étant l'intervalle de mesure). Nous procédons ensuite à une comparaison de ces moyennes.

Nous pouvons remarquer sans peine sur les résultats de la fig. 5.1. que les variations entre longueurs moyennes sont faibles et donc nous pouvons espérer que que l'état de la file d'attente du processeur est stationnaire.

Le processeur étant , si nous pouvons nous exprimer ainsi , le système nerveux du système , nous pensons pouvoir estimer que si l'état de sa file d'attente est stationnaire , l'état des autres files d'attente du réseau a de fortes probabilités d'être stationnaire également. Dans ce cas , nous nous trouvons donc bien dans l'hypothèse d'un régime permanent

Nous aurions , bien sûr , pu réaliser des tests statistiques beaucoup plus complets (épreuves d'hypothèses ou régions de confiance par exemple). Malheureusement , le temps qui nous a été imparti s'est écoulé inexorablement et nous a empêché de nous attarder trop sur certains points .

Mean cpu queue lenght
at t, t + 60´,t + 120´,... with t = 0, 3, 6, 9, ...

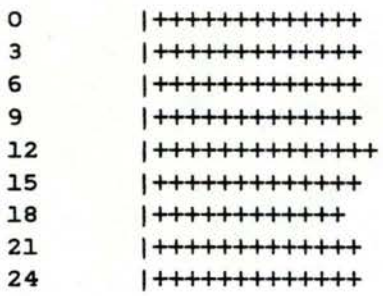


fig. 5.1.

3. Distribution du temps de service des serveurs.

3.1. Présentation.

L'hypothèse émise concernant la distribution des temps de services des différents serveurs revient à vérifier si ces temps de services suivent une loi exponentielle de fonction de répartition :

$$F(a) = Pr[x \leq a] = 1 - e^{-ma}$$

avec m = moyenne de l'échantillon.

Si le coefficient de variation est supérieur à 2 , il n'est pas nécessaire d'effectuer un test plus précis car la distribution n'est certainement pas exponentielle.

Par contre , s'il est inférieur à 2 , nous utiliserons le test de Kolmogorov-Smirnov qui consiste à comparer la distribution observée du temps de service d'un certain serveur à une distribution exponentielle théorique de même moyenne.

3.2. Le test de Kolmogorov-Smirnov

Ce test permet de comparer une distribution observée avec une distribution théorique donnée et détermine si les valeurs observées peuvent ou non appartenir à la population définie par la distribution théorique.

Le test de Kolmogorov-Smirnov est plus puissant que le test du Chi-carré car il est , contrairement à ce dernier , applicable dans tous les cas indépendamment du nombre d'observations et surtout de la découpe en classes.

Voici un résumé de la démarche à suivre :

1. Choix de la distribution théorique.

   Si x est une certaine valeur , $F_0(x)$ représente la proportion des éléments ayant une valeur égale ou inférieure à x.

2. Construire le diagramme cumulatif de l'échantillon.

Si x est une certaine valeur observée , $S_N(x)$ représente la proportion des éléments ayant une valeur égale ou inférieure à x.

3. A chaque $S_N(x)$ observé , associer un $F_0(x)$.

4. Calcul des ! $S_N(x) - F_0(x)$!

5. D = max ! $S_N(x) - F_0(x)$!

6. Consultation des tables avec D ainsi obtenu et pour un certain niveau de signification.

3.3. Choix des serveurs testés.

Avec une précision de l'ordre du 1/60 de seconde , limite qui nous était fixée par le système , nous nous sommes attardés à récolter les temps de services de 3 serveurs : le processeur , un serveur d'entrées / sorties et un terminal.

Le nom de serveur d'échanges attribué au processeur , justifie en quelque sorte le rôle primordial joué par celui-ci dans le réseau et il était donc tout à fait logique que nous testions la distribution de son temps de service.

Nous n'avons mesuré que les temps de services d'un seul serveur d'entrées / sorties pour plusieurs raisons qui semblent pouvoir nous permettre d'étendre nos considérations à tous ces serveurs :

- la philosophie du contrôleur de PM est identique à celle du RK
(cfr. Comparaison de pm.3.c et rk.c)
- la taille des "pseudos-disques" définis sur le disque PM correspond exactement à la taille d'un disque-cartouche RK
- la fréquence et le nombre de mesures effectuées risquaient , si nous n'y prenions garde , de perturber le système qui était déjà soumis à une forte charge
- le manque de temps.

Nous avons également testé les "temps de services" d'un terminal , car nous avions supposé dans notre modèle , que l'ensemble des terminaux pouvait être remplacé par un seul serveur de taux de service $\lambda * (N - n)$ (avec $1/\lambda$: temps de réflexion au terminal) . Nous allons donc vérifier si la distribution du temps de service de ce terminal est également exponentielle.

3.4. Les résultats.

Les résultats obtenus pour les différents serveurs se trouvent résumés en fig. 5.2. , 5.3. et 5.4.

Distribution of cpu active time slices
****

Unit of time : (1/60) sec.
Number of observations : 3300

+ = 64 observations.

```
1       ++++++++++++++++++++++++++++++++++++++++++++++
2       +++
3       ++
4       ++
5       +
6       +
7       +
8       +
9
10      +
```

Coefficient of variation: 0.7642

Signification level 0.20

Distribution is not exponential

fig. 5.2.

Distribution of rk active time slices
****

Unit of time : (1/60) sec.
Number of observations : 5400

+ = 64 observations

```
0       ++++++++++++++++++
1       ++++++++
2       ++++++++++++++++++++++++++
3       ++++++++++++++++
4       ++++++++++++
5       ++++++
6       ++
7       +
8
9
10      +
```

Coefficient of variation: 18.3900
cv is greater than 2
Distribution is certainly not exponential

fig. 5.3.

Distribution of ttye active time slices
****

Unit of time : (1/60) sec.
Number of observations : 745

Coefficient of variation: 5.8177
cv is greater than 2
Distribution is certainly not exponential

fig. 5.4.

Nous constatons sans peine que les distributions des temps de services des terminaux et des serveurs d'entrées / sorties ne sont certainement pas exponentielles car , déjà , leur coefficient de variation est fort eloignée de 1.

Par contre , lorsque nous observons la distribution du temps de service du processeur , nous constatons que celle-ci n'est pas très éloignée d'une distribution exponentielle bien que l'hypothèse d'égalité soit rejetée par le test de Kolmogorov - Smirnov.

4. Discipline des serveurs.

Normalement , les différentes files d'attente constituant le réseau doivent obéir à une discipline FIFO , c'est-à-dire que la file est gérée de façon à ce que les processus soient servis dans leur ordre d'arrivée dans la file.

Malheureusement , nous avons pu voir dans le chapitre présentant UNIX , que si celà s'avérait exact pour les différents serveurs d'entrées / sorties ainsi que pour le serveur de défauts de pages , il n'en était pas de même pour le processeur dont la file suit une discipline à priorités.

5. Conclusion

La mise en garde formulée au début de ce chapitre prend tout son sens lorsque nous sommes obligés de constater que peu d'hypothèses sont vérifiées dans leur entierete.

Arrivés à ce stade de notre étude , nous pouvions difficilement choisir d'élaborer un autre modèle plus complexe dont la mise au point risquerait d'être plus longue encore.

En effet , l'hypothèse d'une distribution exponentielle des serveurs facilite l'analyse du système car les $N(t) = (n_0, \ldots, n_l)$ (l étant le nombre de serveurs) forment un processus Markovien c'est-à-dire que l'état du système ne dépend nullement de son passé.

La méthode généralement choisie pour traiter les réseaux disposant de distributions de services non exponentielles est la "Method of stages" développée par Kleinrock (Cfr. B[11]).

Brièvement , cette méthode représente en fait un serveur quelconque par une combinaison de serveurs exponentiels . Si le processus d'arrivée au serveur est Poissonnien , le système peut être décrit au moyen d'un processus Markovien et la distribution du temps de service de ce serveur sera approchée par une distribution Coxienne.

Cette méthode a été appliquée avec succès par Kleinrock (Cfr. B[11] p. 217).

Nous avons donc décidé de poursuivre les mesures avec notre modèle et d'étudier sa validité par rapport aux mesures réelles au moyen d'analyses statistiques.

VI. ANALYSE DES RESULTATS.

1. Introduction.

Ce n'est pas tout d'élaborer un modèle et de mettre au point un mesureur , encore faut-il comparer leurs résultats . Le but de notre mesureur étant d'ailleurs de valider ou de rejeter le modèle.

Comme annoncé précédemment , l'objectif de ce dernier chapitre sera justement de comparer ces résultats au moyen de tests statistiques. Malheureusement , des contraintes temporelles nous ont empêché de développer ces tests et nous avons dû nous résoudre à réaliser certains tests élémentaires.

Malgré cela , leurs résultats peuvent quand même nous fournir certaines indications sur l'orientation que prendrait une analyse plus précise.

2. Choix des données testées.

Nous avons inséré , en annexe 1 , un exemple de résultats fournis par le modèle. Parmi ceux-ci nous retrouvons :

- les taux de service des différents serveurs
- le nombre moyen de jobs dans les files d'attente de ces différents serveurs.
- le taux de service au serveur T , serveur unique remplacant les terminaux dans notre modèle
- le taux de service du système S1 , composé du serveur d'échanges et des serveurs d'entrées / sorties
- le temps moyen de réponse du système.

Nous allons donc comparer les résultats fournis par le modèle aux résultats du mesureur. Malheureusement , nous ne pourrons effectuer toutes les comparaisons car certaines données ne sont pas accessibles par notre mesureur.

En effet , le choix du modèle nous a fait introduire dans ce dernier un certain nombre de concepts logiques propres à celui-ci comme , par exemple , la découpe en sous-systèmes S1 et S2 ainsi que la substitution de l'ensemble des terminaux par un erveur unique.

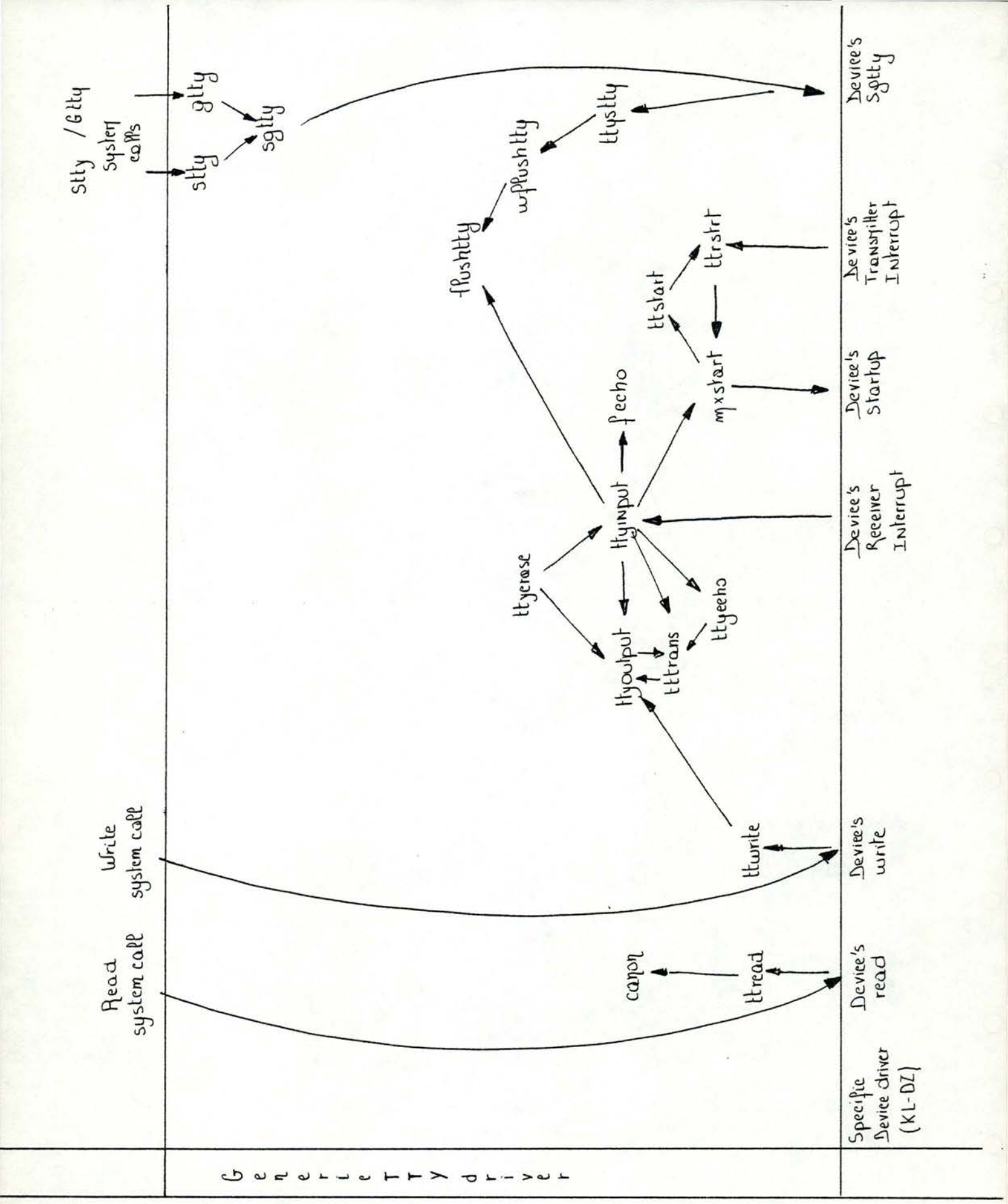
D'autres données sont également inaccessibles par le mesureur pour des motifs différents :

- le nombre de jobs se trouvant dans les files d'attente autres que celle du processeur ne peuvent être mesurés de façon précise au moyen de techniques simples.
- la complexité de la gestion du contrôleur des terminaux (cfr fig. 6.1.) ne nous a pas permis d'avoir une idée précise du temps de réponse du système.

Nous en sommes donc réduits à analyser uniquement le nombre de jobs dans la file d'attente du processeur ainsi que les taux d'utilisation des serveurs les plus utilisés à savoir ceux correspondants au processeur et aux "pseudos-disques":

pm02 , pm06 & pm08 : fichiers et programmes des mémorants
pm10 : disque de swapping
pm11 : disque système
pm00 , pm14 , pm15 & pm 17 : fichiers et programmes d'autres utilisateurs
pm26 : fichiers temporaires.

Ce qui nous donne donc au total 12 variables à analyser.



Interactions entre routines du contrôleur de terminaux.
fig. 6.1.

3. Choix des techniques statistiques.

3.1. La statistique multivariée.

Nous avons obtenu un certain nombre de résultats groupés en échantillons et issus du modèle et du mesureur . Ceux-ci correspondent chacun en termes statistiques à une population.

Nous allons , pour tenter de valider le modèle , effectuer un test d'homogénéité à 12 variables. Ce test peut être considéré comme une généralisation des tests d'homogénéités classiques à une seule variable et , fait partie de la statistique multivariée.

Tout comme un test t de Student permet de contrôler dans certaines conditions la validité d'une hypothèse d'égalité des moyennes de deux populations , le test décrit ci-après permet de comparer les moyennes de 2 populations dont les échantillons sont groupés par paires.

3.2. Le test (cfr. B[5] p. 245).

Comme à une dimension , lorsque les échantillons dont on dispose sont associés par paires , le test d'égalité des moyennes de deux populations se confond avec le test de conformité à zéro de la moyenne des différences.

Nous devons donc calculer les différences observées relatives à chaque variable et pour les 2 populations.

$$d_{ij} = x_{1ij} - x_{2ij} \quad (i=1,\ldots,p;j=1,\ldots,n)$$

avec p nombre de variables
n nombre d'observations.

Nous allons en déduire les moyennes $E(d_{ij})$, les placer dans un vecteur d et calculer la matrice de somme des carrés et produit des écarts A_d , ainsi que la valeur :

$$F_{obs} = \frac{n*(n-p)}{p} d' * A^{-1} * d$$

Pour un population normale a 2 dimensions et pour un echantillon aléatoire et simple , le rejet de l'hypothèse nulle :

$$H_0 : m_1 = m_2 \text{ ou } d = m_1 - m_2 = 0$$

doit intervenir lorsque la valeur F_{obs} est supérieure ou égale à la quantité $F_{1-\alpha}$ trouvée dans les tables et relative à la distribution de Snédécor à p et n-p degrés de liberté , et au niveau de signification α .

4. Les tests.

4.1. Les données.

Mean observed values

U0	:	0.44559442
U1	:	0.00818617
U2	:	0.00378955
U3	:	0.00972174
U4	:	0.02990461
U5	:	0.04237264
U6	:	0.05515625
U7	:	0.00315108
U8	:	0.00246024
U9	:	0.00851554
U10	:	0.00895396
N11	:	6.37361791

Mean measured values

U0	:	0.30238845
U1	:	0.00733690
U2	:	0.00231957
U3	:	0.00629608
U4	:	0.01905677
U5	:	0.02784481
U6	:	0.03306426
U7	:	0.00208500
U8	:	0.00212501
U9	:	0.00711506
U10	:	0.00552213
N11	:	11.46299002

Mean of differences between observed and measured values.

d[0] : 0.14320597
d[1] : 0.00084927
d[2] : 0.00146998
d[3] : 0.00342566
d[4] : 0.01084784
d[5] : 0.01452783
d[6] : 0.02209200
d[7] : 0.00106608
d[8] : 0.00033523
d[9] : 0.00140048
d[10] : 0.00343182
d[11] : -5.08937211

Standard deviations for those means

ec[0] : 0.11895076
ec[1] : 0.00186948
ec[2] : 0.00301386
ec[3] : 0.00733639
ec[4] : 0.01286188
ec[5] : 0.01196143
ec[6] : 0.04225373
ec[7] : 0.00097501
ec[8] : 0.00058831
ec[9] : 0.00182582
ec[10] : 0.00445875
ec[11] : 2.98592097

4.2. Mise en oeuvre du test de comparaison des moyennes.

Le test décrit en 3.2. a tout d'abord été réalisé avec 27 observations et 12 variables.

Nous avons calculé les d_{ij} et nous avons obtenu le vecteur d. Ensuite, nous avons calculé la matrice de somme des carrés et produit des écarts A_d dont nous avons pris l'inverse au moyen de la routine LINV3F de IMSL (*). Les éléments de cette matrice étant assez petits (de l'ordre de 10^{-4}), nous avons eu des problèmes de précision qui ont entraîné des erreurs dans la matrice inversée (non-symétricité de celle-ci). Nous avons donc appliqué à la matrice de départ un coefficient correcteur de 10^{4} que nous avons déduit après l'inversion de celle-ci.

Nous avons alors obtenu :

$F_{obs} = 428.875$

à 12 et 15 degrés de liberté.

Ce résultat étant franchement mauvais, nous avons cherché à savoir d'où provenaient les divergences car les résultats fournis par le modèle (cfr. Annexe 1) nous semblaient plutôt proches de ceux du mesureur.

Nous allons donc pour chaque d_i effectué un test t de Student.

4.3. Elaboration d'une région de confiance pour chaque d_i.

4.3.1. Région de confiance

Si

$$\bar{d}_i - \frac{\sigma_i Q_{t_{n-1}}(1-\alpha/2)}{\sqrt{(n-1)}} \leq 0 \leq \bar{d}_i + \frac{\sigma_i Q_{t_{n-1}}(1-\alpha/2)}{\sqrt{(n-1)}}$$

avec $\sigma_i = \sqrt{\dfrac{\sum_{j=1}^{n} (d_{ij} - d_i)^2}{n}}$ $\quad i = 1, \ldots, p$

$Q_{t_{n-1}}(1-\alpha/2)$: quantile t de student à n - 1 degrés de liberté pour un niveau de signification

---

(*) IMSL est une librairie mathématique dont dispose le DEC 2060 des Facultés.

Alors nous ne pouvons rejeter l'hypothèse d'égalité des moyennes m_{1i} et m_{2i} au niveau de signification α .

4.3.2. Le test.

Si nous établissons une région de confiance pour la moyenne des différences entre les observations et les mesures , nous obtenons les résultats suivants pour les différents serveurs :

* Cpu :	rejet de l'hyp.	$m_{1,01} = m_{2,01}$	avec	α	= 0.001
* Pm00 :	non rejet "	$m_{1,02} = m_{2,02}$	avec	α	= 0.005
* Pm02 :	non rejet "	$m_{1,03} = m_{2,03}$	avec	α	= 0.005
* Pm06 :	non rejet "	$m_{1,04} = m_{2,04}$	avec	α	= 0.005
* Pm08 :	non rejet "	$m_{1,05} = m_{2,05}$	avec	α	= 0.005
* Pm10 :	rejet "	$m_{1,06} = m_{2,06}$	avec	α	= 0.001
* Pm11 :	non rejet "	$m_{1,07} = m_{2,07}$	avec	α	= 0.005
* Pm14 :	rejet "	$m_{1,08} = m_{2,08}$	avec	α	= 0.001
* Pm15 :	non rejet "	$m_{1,09} = m_{2,09}$	avec	α	= 0.005
* Pm17 :	non rejet "	$m_{1,10} = m_{2,10}$	avec	α	= 0.005
* Pm26 :	non rejet "	$m_{1,11} = m_{2,11}$	avec	α	= 0.005
* File :	rejet "	$m_{1,12} = m_{2,12}$	avec	α	= 0.001

Les hypothèses 1 , 6 et 8 bien qu non acceptées au niveau de signification le plus large , peuvent à la limite ne pas être rejetées car , les valeurs fournies comme bornes pour l'intervalle sont malgré tout assez proche de 0.

Cece n'est malheureusement pas le cas pour l'hypothèse 12 et nous allons donc recommencer le test décrit en 3.2. en éliminant les observations correspondant à cette variable.

4.4. Reprise du test de comparaison des moyennes.

Si nous reprenons le test décrit en 3.2. avec les 11 premières variables nous obtenons :

$F_{obs} = 8.439$
à 11 et 16 degrés de liberté

et nous devons donc toujours rejeter l'hypothèse d'égalité des moyennes. Mais nous obtenons une valeur beaucoup plus proche de celles fournies dans les tables.

5. Conclusion.

De ces analyses statistiques élémentaires , nous pouvons déduire que les valeurs estimées pour les différentes taux d'utilisation semblent malgré tout représenter une bonne estimation de la réalité.

Nous avons vu , dans le chapitre précédent , que les hypothèses concernant les distributions des temps de services et les disciplines des files d'attente des serveurs n'étaient pas toutes vérifiées.

Néanmoins , les estimations des différents taux d'utilisations semblent fiables et nous pouvons en conclure qu'elles ne dépendent pas des hypothèses du modèle. Ceci se trouve confirmé par la loi de conservation des flux :

$$A_0 * u_0 * p_j = A_j * u_j$$

qui signifie qu'à partir du moment où nous avons déterminé les flux d'une partie du système , nous avons également déterminé les flux dans le reste du système.

Par contre , il semblerait que , pour obtenir des estimations plus proches de la réalité pour le nombre de jobs dans la file d'attente du processeur , il faille modifier les formules permettant d'obtenir les Q_i ($1 \leq i \leq 1$) dans notre modèle.

Mais , ceci implique , en partie , une modification de notre modèle , modification qu'il ne nous est plus possible de réaliser à l'heure actuelle.

CONCLUSION.

Ce mémoire nous a permis d'étudier , dans certains de ses détails , un système d'exploitation très bien conçu . Nous pensons d'ailleurs devoir rendre hommage à D. M. Ritchie et à K. Thompson ainsi qu' à toutes les personnes qui ont collaboré à la mise au point de UNIX. En effet , UNIX a été réalisé dans le souci d'en faciliter l'accès par un néophyte.

Nous avons également eu l'occasion , lors de notre stage à la Vrije Universiteit d'Amsterdam , de cotoyer des personnes connaissant très bien UNIX , pour l'avoir modifier à maintes reprises , et qui nous ont donné la possibilité de travailler sur une version de UNIX différente de celle des Facultés.

Lors de notre retour en Belgique , nous avons du nous adapter à cette version ce qui a retardé le choix de notre modèle d'autant plus qu' à l'origine les modifications avaient été conçues pour un modèle très simple.

Si nous mettons tant l'accent sur cette partie du travail , cela est du au fait que ce sont les diverses manipulations du système qui ont occupés le plus clair du temps et qui ont donc empiétés sur d'autres parties de notre travail.

D'ailleurs , il apparait à la lecture des deux derniers chapitres que beaucoup de choses n'ont pu être réalisées à cause de contraintes temporelles.

De plus , sur la fin , nous avons encore été retardés par divers problèmes et nous avons notamment perdu beaucoup de temps pour la réalisation des transferts entre le PDP 11/45 et le DEC 2060 ainsi que pour l'utilisation adéquate des routines de la librairie mathématique IMSL.

S'il nous fallait donner suite à ce mémoire , nous orienterions nos recherches vers le développement d'un modèle plus proche encore de la réalité ainsi que sur la réalisation d'un contrôleur de terminaux particulier permettant de mesurer le temps de réponse du système de façon aisée car celui-ci reste , malgré tout , la "valeur étalon" lorsqu' on parle de performances des systèmes informatiques.

BIBLIOGRAPHIE.

1. Adans Jean-Paul, Vercheval Jean

Mesures de performances de UNIX sous une charge universitaire.
Mémoire présenté en vue de l'obtention du diplôme de licencié et maître en informatique.
Année académique 1978-1979.

2. Bell telephone laboratories, Incorporated.

UNIX programmer's manual.
Seventh edition, volume 1, 2A & 2B.
Murray Hill, New Jersey.

3. Brandwajn A.

Equivalence et décomposition dans les réseaux de serveurs d'échanges avec application à un modèle de système à partage de ressources.
IRIA, laboratoire de recherche en informatique et automatique.
Rapport de recherche 101, février 1975.

4. Buzen Jeffrey P., Denning Peter J.

The operationnal analysis of queuing network models.
ACM Computing surveys, volume 10.

5. Dagnelie Pierre.

L'analyse statistique à plusieurs variables.
Les presses agronomiques de Gembloux.

6. Digital equipement corporation.

PDP 11/45
Processor handbook.

7. Fishman George S.

Concepts and methods in discrete event digital simulation.
John Wiley & sons.

8. Gelenbe E., Mitrani I.

Analysis and synthesis of computer system models.
Academic press, London.

9. Graham G. Scott.

Queuing network models of computer system performance.
ACM Computing surveys, volume 10.

10. Kernighan Brian W., Ritchie Dennis M.

The C programming language.
Prentice Hall software series.

11. Kleinrock Leonard.

Queuing systems.
Volume 2.
John Wiley & sons.

12. Lions J.

A comment on the UNIX operating system.
Department of computer science.
The university of New South Wales.

13. Martin Bernard.

Un système d'exploitation conversationnel en temps partag' :
UNIX.
Minis et micros numéro 140.

14. Noirhomme-Fraiture M.

Systèmes d'exploitation.
Modèles et performances (partim).

15. Noirhomme - Fraiture M.

Statistiques.
Notes de cours.

16. de Rivet Ph.

Postes de services et systèmes d'attente.
Notes de cours.

17. Rose Clifford A.

A Measurement procedure for queuing network models of computer systems.
ACM Computing surveys, volume 10.

18. Siegel Sidney.

Non parametric statistics for the behavioral sciences.
McGraw-Hill Kogakusha, LTD.

19. Sup'aréo école nationale supérieure de l'aéronautique et de l'espace

Mesures des systèmes informatiques.
Cépadues éditions, Toulouse.

20. IMSL library reference manual.

21. Biometrika tables for statisticians.

Volume 1 , third edition.
Edited by E. S. Pearson and H. O. Hartley.
Cambridge.
Published for the Biometrika Trustees at the University Press.

ANNEXE 1

Transition probabilities
****

CPU -> OUT	:	0.0042
CPU -> CPU	:	0.6055
CPU -> RK0	:	0.0046
CPU -> RK1	:	0.0000
CPU -> RK2	:	0.0555
CPU -> PM0	:	0.0000
CPU -> PM1	:	0.0000
CPU -> PM2	:	0.0000
CPU -> PM3	:	0.0000
CPU -> PM4	:	0.0000
CPU -> PM5	:	0.0000
CPU -> PM6	:	0.0314
CPU -> PM7	:	0.0000
CPU -> PM8	:	0.0168
CPU -> PM9	:	0.0187
CPU -> PM10	:	0.1406
CPU -> PM11	:	0.0962
CPU -> PM12	:	0.0038
CPU -> PM13	:	0.0000
CPU -> PM14	:	0.0036
CPU -> PM15	:	0.0007
CPU -> PM16	:	0.0000
CPU -> PM17	:	0.0015
CPU -> PM18	:	0.0000
CPU -> PM19	:	0.0000
CPU -> PM20	:	0.0000
CPU -> PM21	:	0.0000
CPU -> PM22	:	0.0000
CPU -> PM23	:	0.0000
CPU -> PM24	:	0.0000
CPU -> PM25	:	0.0000
CPU -> PM26	:	0.0077
CPU -> PM27	:	0.0093
CPU -> PM28	:	0.0000
CPU -> PM29	:	0.0000
CPU -> PM30	:	0.0000
CPU -> PM31	:	0.0000

Transition probabilities
****

CPU ->	OUT :	0.0036
CPU ->	CPU :	0.6100
CPU ->	RK0 :	0.0036
CPU ->	RK1 :	0.0000
CPU ->	RK2 :	0.0050
CPU ->	PM0 :	0.0000
CPU ->	PM1 :	0.0000
CPU ->	PM2 :	0.0411
CPU ->	PM3 :	0.0000
CPU ->	PM4 :	0.0000
CPU ->	PM5 :	0.0000
CPU ->	PM6 :	0.0000
CPU ->	PM7 :	0.0000
CPU ->	PM8 :	0.0559
CPU ->	PM9 :	0.0007
CPU ->	PM10 :	0.1591
CPU ->	PM11 :	0.0824
CPU ->	PM12 :	0.0000
CPU ->	PM13 :	0.0000
CPU ->	PM14 :	0.0065
CPU ->	PM15 :	0.0155
CPU ->	PM16 :	0.0000
CPU ->	PM17 :	0.0000
CPU ->	PM18 :	0.0000
CPU ->	PM19 :	0.0000
CPU ->	PM20 :	0.0000
CPU ->	PM21 :	0.0000
CPU ->	PM22 :	0.0000
CPU ->	PM23 :	0.0000
CPU ->	PM24 :	0.0000
CPU ->	PM25 :	0.0000
CPU ->	PM26 :	0.0032
CPU ->	PM27 :	0.0134
CPU ->	PM28 :	0.0000
CPU ->	PM29 :	0.0000
CPU ->	PM30 :	0.0000
CPU ->	PM31 :	0.0000

Résultats fournis par le modèle.

RESULTS FOR SUBSYSTEM S1

SERVER	MEASURED UTILIZATION	ESTIMATED UTILIZATION
cpu	0.3060	0.4022
rk0	0.0000	0.0012
rk1	0.0000	0.0000
rk2	0.0000	0.0000
pm00	0.0000	0.0000
pm01	0.0000	0.0000
pm02	0.0039	0.0051
pm03	0.0000	0.0000
pm04	0.0000	0.0000
pm05	0.0000	0.0000
pm06	0.0216	0.0284
pm07	0.0005	0.0006
pm08	0.0084	0.0111
pm09	0.0000	0.0000
pm10	0.0316	0.0416
pm11	0.0151	0.0198
pm12	0.0000	0.0000
pm13	0.0000	0.0000
pm14	0.0023	0.0030
pm15	0.0005	0.0006
pm16	0.0000	0.0000
pm17	0.0000	0.0000
pm18	0.0000	0.0000
pm19	0.0000	0.0000
pm20	0.0000	0.0000
pm21	0.0000	0.0000
pm22	0.0000	0.0000
pm23	0.0000	0.0000
pm24	0.0000	0.0000
pm25	0.0000	0.0000
pm26	0.0018	0.0024
pm27	0.0042	0.0056
pm28	0.0000	0.0000
pm29	0.0000	0.0000
pm30	0.0000	0.0000
pm31	0.0000	0.0000

```
*************************
*SERVER * Nb JOBS       *
*************************
*  cpu  *  6.52         *
*  rk0  *  0.00         *
*  rk1  *  0.00         *
*  rk2  *  0.00         *
*  pm00 *  0.00         *
*  pm01 *  0.00         *
*  pm02 *  0.01         *
*  pm03 *  0.00         *
*  pm04 *  0.00         *
*  pm05 *  0.00         *
*  pm06 *  0.07         *
*  pm07 *  0.00         *
*  pm08 *  0.03         *
*  pm09 *  0.00         *
*  pm10 *  0.11         *
*  pm11 *  0.05         *
*  pm12 *  0.00         *
*  pm13 *  0.00         *
*  pm14 *  0.01         *
*  pm15 *  0.00         *
*  pm16 *  0.00         *
*  pm17 *  0.00         *
*  pm18 *  0.00         *
*  pm19 *  0.00         *
*  pm20 *  0.00         *
*  pm21 *  0.00         *
*  pm22 *  0.00         *
*  pm23 *  0.00         *
*  pm24 *  0.00         *
*  pm25 *  0.00         *
*  pm26 *  0.01         *
*  pm27 *  0.01         *
*  pm28 *  0.00         *
*  pm29 *  0.00         *
*  pm30 *  0.00         *
*  pm31 *  0.00         *
* total *  6.82         *
*************************
```

Measured number of jobs in CPU queue : 13.3533

TOTAL RESULTS

Terminal service rate:5.0906

System S2 service rate:0.0602

Mean response time:4.5181

FACULTES
UNIVERSITAIRES
N.D. DE LA PAIX

NAMUR

INSTITUT D'INFORMATIQUE

RUE GRANDGAGNAGE, 21, B - 5000 NAMUR (BELGIUM)

FACULTES UNIVERSITAIRES NOTRE-DAME DE LA PAIX,

NAMUR.

INSTITUT D'INFORMATIQUE.

MESURES DE PERFORMANCES DE UNIX A PARTIR D'UN MODELE MATHEMATIQUE.

Annexes 2 & 3.

Promoteur : M. Noirhomme

Dirk Cornil

Mémoire présenté en vue de l'obtention du grade de LICENCIE ET MAITRE EN INFORMATIQUE.

Année académique 1980-1981

ANNEXE 2.

MODIFICATIONS APPORTEES A UNIX.

DESCRIPTION DES METHODES DE MESURES.

Nous allons maintenant decrire en quelques mots , la facon dont nous avons effectue nos mesures.

Nous expliquons notamment comment nous avons mesure :

- les taux d'utilisation des differents serveurs.
- les probabilites de transition
- les taux de service des differents serveurs
- le temps de reflexion a un certain terminal
- le nombre de jobs dans le systeme.

1. Mesure des taux d'utilisation des differents serveurs.

Le taux d'utilisation d'un serveur est fourni par le rapport du temps total d'uilisation du serveur durant la periode de mesures sur le temps pris par cette periode de mesures.

Nous avons donc comptabilise le temps d'activite des differents serveurs (processeur & serveurs d'entrees-sorties) ainsi que le temps total necessite par une session de mesures.

Pour realiser cela , nous avons introduit dans la routine "clock()" du fichier clock.c un certain nombre de compteurs correspondant chacun a un serveur. Ces compteurs sont incrementes tous les 1/10 eme de seconde si le "flag" qui leur est associe est positionne , c'est-a-dire s'il est different de la valeur nulle. En realite , la routine "clock()" est activee tous les 1/60 eme de seconde mais nous avons choisi de travailler avec une precision de l'ordre du 1/10 eme de seconde afin de ne pas trop surcharger le systeme.

1.1. Modification des "flags".

- Processeur.

Dans la routine chargee du scheduling du processeur (routine "swtch()" du fichier slp.c) , nous retrouvons les instructions suivantes :

```
loop    :

        /*
         * if no process is runnable , idle.
         */


        if(p == NULL)
            {
            p = rp;
            MISE A 0 DU FLAG ASSOCIE AU PROCESSEUR;
            idle();
            MISE A 1 DU FLAG ASSOCIE AU PROCESSEUR;
            goto loop;
            }
```

Nous pouvons donc constater qu'en modifiant la valeur du flag de cette maniere, le compteur comptabilisant le temps d'activation du processeur ne sera incremente que lorsque le processeur sera actif.

- Serveurs d'entrees-sorties (rk et pm).

Nous croyons utile de rappeler ici que le PDP 11/45 des Facultes dispose de 3 disques cartouches RK 05 et d'un disque PM DS 11/80 qui a ete subdivise en un certain nombre de "pseudos-disques" de facon a assurer la compatibilite entre la taille d'un RK et d'un "pseudo-disque".

Lorsque, dans la suite de cette description, nous parlons de rk ou de pm, il s'agit en fait du serveur d'entrees-sorties associe a l'un des RK ou a l'un des "pseudos-disques" du PM.

+ Mise a zero du flag.

Celle-ci se fait soit dans la routine "rkintr()" du fichier rk.c lorsque le serveur d'entrees-sorties considere est un rk, soit dans la routine "pmintr()" du fichier pm.3.c lorsqu'il s'agit d'un pm.

Ces routines sont destinees a traiter les interruptions entrainees par la fin d'une operation d'entree-sortie physique sur disque.

La mise a zero du flag est realisee a la fin de cette routine apres les differents controles et apres la liberation des ressources necessaires pour une operation d'entree-sortie physique ("iodone()").

+ Positionnement du flag.

Celui-ci se fait dans la routine "devstart()" du fichier bio.c et ceci aussi bien pour les rk que pour les pm.

Nous avons place les "flags" de facon a ce qu'ils soient positionnes avant la prise en main de l'operation d'entree-sortie par le hardware.

1.2. L'incrementation des compteurs.

Organigramme d'incrementation :

```
if(t_loop == 1/10 sec)
  {
  t_loop = 0;

  if(cpu != idle)
    t_cpu++;

  for(i=0; i<3; i++)
    if(rk[i] != idle)
      t_rk[i]++;

  for(i=0; i<32; i++)
    if(pm[i] != idle)
      t_pm[i]++

  t_real++;
  }
else t_loop++;
```

avec t_loop : precision de 1/10 sec
t_cpu : temps d'activite processeur
t_rk[i]: temps d'activite RKi (i = 1, 2, 3)
t_pm[i]: temps d'activite PMi (i = 1, ..., 32)
t_real : temps d'activite total.

1.3. Calcul des taux d'utilisation.

Ceux-ci se calculent lors de la collecte des resultats et , par exemple pour le processeur , il suffira de calculer le rapport t_cpu sur t_real.

2. Les probabilites de transition.

Lorsqu'un processus quitte un serveur , il se deplace dans le reseau en suivant un certain chemin. Les differents chemins que peuvent suivre les processus se trouvent representes sur le graphe de la fig. 3.1. Nous appellons pr(i,j) la probabilite que le processus aille au serveur j , une fois qu'il a fini son service au serveur i. (Dans le reseau que nous avons etudie , tous les echanges se font par l'intermediaire du processeur et seuls les pr(i,j) avec le serveur i = processeur sont differents de 1 ou de 0).

2.1. Incrementation des compteurs.

- Chemin : processeur -> processeur.

A chaque appel de la routine "swtch()" du fichier slp.c , nous incrementons un compteur. En effet , cette routine est chargee du scheduling du processeur:

+ elle remplace le processus courant par un processus se trouvant dans la file d'attente du processeur
+ elle replace le processus courant dans la file d'attente du processeur.

- Chemin : processeur -> monde exterieur.

Nous comptabilisons le nombre de jobs quittant le systeme en incrementant un compteur a chaque appel de la routine "exit()" du fichier des appels systemes sys1.c.

- Chemin : processeur -> serveurs d'entrees-sorties.

Nous avons introduit dans differents fichiers (bio.c, rdwrix.c, sys3.c, alloc.c, fio.c) des compteurs qui sont incrementes a chaque entree-sortie physique c'est-a-dire chaque fois qu'apparait une ligne :

```
(*bdevsw[dev.d_major].d_xxx)()
                ou
(*cdevsw[dev.d_major].d_xxx)()
```

avec xxx : strategy
read
write
open
close

Chaque fois qu'apparait une telle ligne , nous calculons le "major device number" (permet de voir si l'on travaille avec un rk ou un pm) et le "minor device number" (permet de selectionner l'un des rk ou l'un des pm). Disposant de ces 2 informations , nous pouvons maintenant incrementer le compteur correspondant au serveur selectionne.

2.2. Calcul des probabilites de transition.

Pour calculer les probabilites , nous effectuons d'abord la somme des compteurs associes a chaque chemin et la probabilite de transition d'un chemin est donc le rapport du compteur associe a ce chemin par cette somme. Ceci sera fait lors de la collecte des resultats des mesures.

3. Calcul des taux de service.

Le taux de service d'un serveur est defini comme etant le nombre d'unites servies par unite de temps. Comme nous disposons deja des temps d'utilisation des differents serveurs, il nous reste uniquement a calculer le nombre d'activations de ces serveurs. Le taux de service sera calcule lors de la collecte des resultats en effectuant le rapport nombre d'activations sur temps d'activation.

- Nombre d'activations du processeur.

Ce nombre correspond en fait au nombre de processus qui ont quitte le processeur lors de la periode de mesures. Ce nombre a deja ete calcule et correspond a la somme des compteurs associes aux differents chemins de transition quittant le processeur.

- Nombre d'activations des serveurs d'entrees-sorties.

Les routines "pmintr"() ou "rkintr"() sont appelees a chaque fin d'operation d'entree-sortie physique. Une fois selectionne le rk ou le pm (utilisation du "minor device number") , il ne nous reste qu'a incremente le compteur qui lui est associe.

4. Calcul du temps de reflexion au terminal.

Le temps de reflexion au terminal est defini comme etant le temps ecoule entre le moment ou l'ordinateur renvoie le "prompt" et le moment ou l'utilisateur renvoie une nouvelle commande. Il s'agit donc du temps que l'utilisateur met pour preparer sa commande. Pour plus de facilite et pour eviter de surcharger le systeme , nous ne calculerons le temps de reflexion que sur un seul terminal.

Comme pour le calcul des taux d'utilisation , nous utilisons un "flag". Lorsque celui-ci est positionne nous incrementons le compteur correspondant dans la routine "clock()" du fichier clock.c (avec une precision de 1/60 eme de seconde).

- mise a 1 du "flag".

  Si le terminal est logiquement actif , la routine "ttread" de tty.4.c , qui a pour sous-routine "canon()" , attend que le terminal lui envoie des caracteres. Nous positionnons donc le flag juste avant que cette routine ce mette en attente.

- mise a 0 du "flag".

  Nous remettons le "flag" a zero des que le terminal est logiquement inactif et des qu'il envoie des caracteres dans les tampons (routine "canon()").

Nous avons place ces temps de reflexion dans un tableau qui est transfere dans l'espace utilisateur chaque fois qu'il est rempli. Pour effectue ce transfert de l'espace noyau vers l'espace utilisateur , nous avons cree l'appel systeme "svtime".

5. Nombre de processus dans le systeme.

Nous calculons le nombre de processus dans le systeme au moyen d'un programme utilisateur ("p_accnt.c") qui effectue un acces a la table des processus (cfr proc.h) et comptabilise tous les processus dont l'etat n'est pas SSTOP.

p_bio.c

```
#
/*
 */

#include "../param.h"
#include "../user.h"
#include "../buf.h"
#include "../conf.h"
#include "../systm.h"
#include "../proc.h"
#include "../seg.h"
#include "../perfo.h"

/*
 * This is the set of buffers proper, whose heads
 * were declared in buf.h.  There can exist buffer
 * headers not pointing here that are used purely
 * as arguments to the I/O routines to describe
 * I/O to be done-- e.g. swbuf, just below, for
 * swapping.
 */
char    buffers[NBUF][514];
struct  buf     swbuf;

/*
 * Declarations of the tables for the magtape devices;
 * see bdwrite.
 */
int     tmtab;
int     httab;
int     asswait[2];     /* number of processes waiting for an associated buffe
int     freewait[2];    /* number of processes waiting for a free buffer */

/*
 * The following several routines allocate and free
 * buffers with various side effects.  In general the
 * arguments to an allocate routine are a device and
 * a block number, and the value is a pointer to
 * to the buffer header; the buffer is marked "busy"
 * so that no on else can touch it.  If the block was
 * already in core, no I/O need be done; if it is
 * already busy, the process waits until it becomes free.
 * The following routines allocate a buffer:
 *      getblk
 *      bread
 *      breada
 * Eventually the buffer must be released, possibly with the
 * side effect of writing it out, by using one of
 *      bwrite
 *      bdwrite
 *      bawrite
 *      brelse
 */

/*
 * Read in (if necessary) the block and return a buffer pointer.
 */
bread(dev, blkno)
{
```

```
        register struct buf *rbp;

        register drname, drno;

        rbp = getblk(dev, blkno);
        if (rbp->b_flags&B_DONE)
                return(rbp);
        rbp->b_flags =| B_READ;
        rbp->b_wcount = -256;

        /* system measures */

        drname = dev.d_major;
        drno = dev.d_minor & 037;

        if(drname == MRK)
                addd(rkrout[drno], 1);
        else if(drname == MPM)
                addd(pmrout[drno], 1);

        (*bdevsw[dev.d_major].d_strategy)(rbp);
        iowait(rbp);
        return(rbp);
}

/*
 * Read in the block, like bread, but also start I/O on the
 * read-ahead block (which is not allocated to the caller)
 */
breada(adev, blkno, rablkno)
{
        register struct buf *rbp, *rabp;
        register int dev;
        int drname, drno;

        dev = adev;
        rbp = 0;
        if (!incore(dev, blkno)) {
                rbp = getblk(dev, blkno);
                if ((rbp->b_flags&B_DONE) == 0) {
                        rbp->b_flags =| B_READ;
                        rbp->b_wcount = -256;

                        /* system measures */

                        drname = adev.d_major;
                        drno = adev.d_minor & 037;

                        if(drname == MRK)
                                addd(rkrout[drno], 1);
                        else if(drname == MPM)
                                addd(pmrout[drno], 1);

                        (*bdevsw[adev.d_major].d_strategy)(rbp);
                }
        }
        if (rablkno && !incore(dev, rablkno)) {
                rabp = getblk(dev, rablkno);
                if (rabp->b_flags & B_DONE)
                        brelse(rabp);
                else {
```

```
                        rabp->b_flags =| B_READ|B_ASYNC;

                        rabp->b_wcount = -256;

                        /* system measures */

                        drname = adev.d_major;
                        drno = adev.d_minor & 037;

                        if(drname == MRK)
                                addd(rkrout[drno],1);
                        else if(drname == MPM)
                                addd(pmrout[drno],1);

                        (*bdevsw[adev.d_major].d_strategy)(rabp);
                }
        }
        if (rbp==0)
                return(bread(dev, blkno));
        iowait(rbp);
        return(rbp);
}

/*
 * Write the buffer, waiting for completion.
 * Then release the buffer.
 */
bwrite(bp)
struct buf *bp;
{
        register struct buf *rbp;
        register flag;
        int drname, drno;

        rbp = bp;
        flag = rbp->b_flags;
        rbp->b_flags =& ~(B_READ | B_DONE | B_ERROR | B_DELWRI);
        rbp->b_wcount = -256;

        /* system measures */

        drname = rbp->b_dev.d_major;
        drno = rbp->b_dev.d_minor & 037;

        if(drname == MRK)
                addd(rkrout[drno],1);
        else if(drname == MPM)
                addd(pmrout[drno],1);

        (*bdevsw[rbp->b_dev.d_major].d_strategy)(rbp);
        if ((flag&B_ASYNC) == 0) {
                iowait(rbp);
                brelse(rbp);
        } else if ((flag&B_DELWRI)==0)
                geterror(rbp);
}

/*
 * Release the buffer, marking it so that if it is grabbed
 * for another purpose it will be written out before being
 * given up (e.g. when writing a partial block where it is
```

```
 * assumed that another write for the same block will soon follow).
 * This can't be done for magtape, since writes must be done
 * in the same order as requested.
 */
bdwrite(bp)
struct buf *bp;
{
        register struct buf *rbp;
        register struct devtab *dp;

        rbp = bp;
        dp = bdevsw[rbp->b_dev.d_major].d_tab;
        if (dp == &tmtab || dp == &httab)
                bawrite(rbp);
        else {
                rbp->b_flags |= B_DELWRI | B_DONE;
                brelse(rbp);
        }
}

/*
 * Release the buffer, start I/O on it, but don't wait for completion.
 */
bawrite(bp)
struct buf *bp;
{
        register struct buf *rbp;

        rbp = bp;
        rbp->b_flags |= B_ASYNC;
        bwrite(rbp);
}

/*
 * release the buffer, with no I/O implied.
 */
brelse(bp)
struct buf *bp;
{
        register struct buf *rbp, **backp;
        register int sps;

        rbp = bp;
        if (rbp->b_flags&B_WANTED)
                wakeup(rbp);
        if (bfreelist.b_flags&B_WANTED) {
                bfreelist.b_flags &= ~B_WANTED;
                wakeup(&bfreelist);
        }
        if (rbp->b_flags&B_ERROR)
                rbp->b_dev.d_minor = -1;  /* no assoc. on error */
        backp = &bfreelist.av_back;
        sps = PS->integ;
        spl6();
        rbp->b_flags &= ~(B_WANTED|B_BUSY|B_ASYNC);
        (*backp)->av_forw = rbp;
        rbp->av_back = *backp;
        *backp = rbp;
        rbp->av_forw = &bfreelist;
        PS->integ = sps;
```

p_bio.c

```
}

/*
 * See if the block is associated with some buffer
 * (mainly to avoid getting hung up on a wait in breada)
 */
incore(adev, blkno)
{
        register int dev;
        register struct buf *bp;
        register struct devtab *dp;

        dev = adev;
        dp = bdevsw[adev.d_major].d_tab;
        for (bp=dp->b_forw; bp != dp; bp = bp->b_forw)
                if (bp->b_blkno==blkno && bp->b_dev==dev)
                        return(bp);
        return(0);
}

/*
 * Assign a buffer for the given block.  If the appropriate
 * block is already associated, return it; otherwise search
 * for the oldest non-busy buffer and reassign it.
 * When a 512-byte area is wanted for some random reason
 * (e.g. during exec, for the user arglist) getblk can be called
 * with device NODEV to avoid unwanted associativity.
 */
getblk(dev, blkno)
{
        register struct buf *bp;
        register struct devtab *dp;
        extern lbolt;

        if(dev.d_major >= nblkdev)
                panic("blkdev");

    loop:
        if (dev < 0)
                dp = &bfreelist;
        else {
                dp = bdevsw[dev.d_major].d_tab;
                if(dp == NULL)
                        panic("devtab");
                for (bp=dp->b_forw; bp != dp; bp = bp->b_forw) {
                        if (bp->b_blkno!=blkno || bp->b_dev!=dev)
                                continue;
                        spl6();
                        if (bp->b_flags&B_BUSY) {
                                bp->b_flags =| B_WANTED;
                                addd(asswait, 1);
                                sleep(bp, PRIBIO);
                                spl0();
                                goto loop;
                        }
                        spl0();
                        notavail(bp);
                        return(bp);
                }
        }
```

p_bio.c

```
        spl6();

        if (bfreelist.av_forw == &bfreelist) {
                bfreelist.b_flags =| B_WANTED;
                addd(freewait, 1);
                sleep(&bfreelist, PRIBIO);
                spl0();
                goto loop;
        }
        spl0();
        notavail(bp = bfreelist.av_forw);
        if (bp->b_flags & B_DELWRI) {
                bp->b_flags =| B_ASYNC;
                bwrite(bp);
                goto loop;
        }
        bp->b_flags = B_BUSY | B_RELOC;
        bp->b_back->b_forw = bp->b_forw;
        bp->b_forw->b_back = bp->b_back;
        bp->b_forw = dp->b_forw;
        bp->b_back = dp;
        dp->b_forw->b_back = bp;
        dp->b_forw = bp;
        bp->b_dev = dev;
        bp->b_blkno = blkno;
        return(bp);
}

/*
 * Wait for I/O completion on the buffer; return errors
 * to the user.
 */
iowait(bp)
struct buf *bp;
{
        register struct buf *rbp;

        rbp = bp;
        spl6();
        while ((rbp->b_flags&B_DONE)==0)
                sleep(rbp, PRIBIO);
        spl0();
        geterror(rbp);
}

/*
 * Unlink a buffer from the available list and mark it busy.
 * (internal interface)
 */
notavail(bp)
struct buf *bp;
{
        register struct buf *rbp;
        register int sps;

        rbp = bp;
        sps = PS->integ;
        spl6();
        rbp->av_back->av_forw = rbp->av_forw;
        rbp->av_forw->av_back = rbp->av_back;
        rbp->b_flags =| B_BUSY;
```

```
        PS->integ = sps;
```


p_bio.c

```
}

/*
 * Mark I/O complete on a buffer, release it if I/O is asynchronous,
 * and wake up anyone waiting for it.
 */
iodone(bp)
struct buf *bp;
{
        register struct buf *rbp;

        rbp = bp;
        if(rbp->b_flags&B_MAP)
                mapfree(rbp);
        rbp->b_flags =| B_DONE;
        if (rbp->b_flags&B_ASYNC)
                brelse(rbp);
        else {
                rbp->b_flags =& ~B_WANTED;
                wakeup(rbp);
        }
}

/*
 * Zero the core associated with a buffer.
 */
clrbuf(bp)
int *bp;
{
        register *p;
        register c;

        p = bp->b_addr;
        c = 256;
        do
                *p++ = 0;
        while (--c);
}

/*
 * Initialize the buffer I/O system by freeing
 * all buffers and setting all device buffer lists to empty.
 */
binit()
{
        register struct buf *bp;
        register struct devtab *dp;
        register int i;
        struct bdevsw *bdp;

        bfreelist.b_forw = bfreelist.b_back =
            bfreelist.av_forw = bfreelist.av_back = &bfreelist;
        for (i=0; i<NBUF; i++) {
                bp = &buf[i];
                bp->b_dev = -1;
                bp->b_addr = buffers[i];
                bp->b_back = &bfreelist;
                bp->b_forw = bfreelist.b_forw;
                bfreelist.b_forw->b_back = bp;
```

```
                bfreelist.b_forw = bp;

                bp->b_flags = B_BUSY;
                brelse(bp);
        }
        i = 0;
        for (bdp = bdevsw; bdp->d_open; bdp++) {
                dp = bdp->d_tab;
                if(dp) {
                        dp->b_forw = dp;
                        dp->b_back = dp;
                }
                i++;
        }
        nblkdev = i;
}

/*
 * Device start routine for disks
 * and other devices that have the register
 * layout of the older DEC controllers (RF, RK, RP, TM)
 */
#define IENABLE 0100
#define WCOM    02
#define RCOM    04
#define GO      01
devstart(bp, devloc, devblk, hbcom)
struct buf *bp;
int *devloc;
{
        register int *dp;
        register struct buf *rbp;
        register int com;
        int drname, drno;

        dp = devloc;
        rbp = bp;
        *dp = devblk;                   /* block address */
        *--dp = rbp->b_addr;            /* buffer address */
        *--dp = rbp->b_wcount;          /* word count */
        com = (hbcom<<8) | IENABLE | GO |
                ((rbp->b_xmem & 03) << 4);
        if (rbp->b_flags&B_READ)        /* command + x-mem */
                com =| RCOM;
        else
                com =| WCOM;

        /* system measures */

        drname = rbp->b_dev.d_major;
        drno = rbp->b_dev.d_minor & 037;

        if(drname == MRK)
                rkrun[drno] = SACTIVE;
        else if(drname == MPM)
                pmrun[drno] = SACTIVE;

        *--dp = com;
}

/*
```

p_bio.c

```
 * startup routine for RH controllers.
 */
#define RHWCOM  060
#define RHRCOM  070

rhstart(bp, devloc, devblk, abae)
struct buf *bp;
int *devloc, *abae;
{
        register int *dp;
        register struct buf *rbp;
        register int com;

        dp = devloc;
        rbp = bp;
        if(cputype == 70)
                *abae = rbp->b_xmem;
        *dp = devblk;                           /* block address */
        *--dp = rbp->b_addr;                    /* buffer address */
        *--dp = rbp->b_wcount;                  /* word count */
        com = IENABLE | GO |
                ((rbp->b_xmem & 03) << 8);
        if (rbp->b_flags&B_READ)                /* command + x-mem */
                com =| RHRCOM; else
                com =| RHWCOM;
        *--dp = com;
}

/*
 * 11/70 routine to allocate the
 * UNIBUS map and initialize for
 * a unibus device.
 * The code here and in
 * rhstart assumes that an rh on an 11/70
 * is an rh70 and contains 22 bit addressing.
 */
int     maplock;
mapalloc(abp)
struct buf *abp;
{
        register i, a;
        register struct buf *bp;

        if(cputype != 70)
                return;
        spl6();
        while(maplock&B_BUSY) {
                maplock =| B_WANTED;
                sleep(&maplock, PSWP);
        }
        maplock =| B_BUSY;
        spl0();
        bp = abp;
        bp->b_flags =| B_MAP;
        a = bp->b_xmem;
        for(i=16; i<32; i=+2)
                UBMAP->r[i+1] = a;
        for(a++; i<48; i=+2)
                UBMAP->r[i+1] = a;
        bp->b_xmem = 1;
```

p_bio.c

```
}

mapfree(bp)
struct buf *bp;
{

        bp->b_flags =& ~B_MAP;
        if(maplock&B_WANTED)
                wakeup(&maplock);
        maplock = 0;
}

/*
 * swap I/O
 */
swap(blkno, coreaddr, count, rdflg)
{
        register int *fp;
        register drname, drno;

        fp = &swbuf.b_flags;
        spl6();
        while (*fp&B_BUSY) {
                *fp =| B_WANTED;
                sleep(fp, PSWP);
        }
        *fp = B_BUSY | B_PHYS | rdflg;
        swbuf.b_dev = swapdev;
        swbuf.b_wcount = - (count<<5);   /* 32 w/block */
        swbuf.b_blkno = blkno;
        swbuf.b_addr = coreaddr<<6;      /* 64 b/block */
        swbuf.b_xmem = (coreaddr>>10) & 077;

        /* system measures */

        drname = swbuf.b_dev.d_major;
        drno = swbuf.b_dev.d_minor & 037;

        if(drname == MRK)
                addd(rkrout[drno], 1);
        else if(drname == MPM)
                addd(pmrout[drno], 1);

        (*bdevsw[swapdev>>8].d_strategy)(&swbuf);
        spl6();
        while((*fp&B_DONE)==0)
                sleep(fp, PSWP);
        if (*fp&B_WANTED)
                wakeup(fp);
        spl0();
        *fp =& ~(B_BUSY|B_WANTED);
        return(*fp&B_ERROR);
}

/*
 * make sure all write-behind blocks
 * on dev (or NODEV for all)
 * are flushed out.
 * (from umount and update)
 */
```

```
{
        register struct buf *bp;

loop:
        spl6();
        for (bp = bfreelist.av_forw; bp != &bfreelist; bp = bp->av_forw) {
                if (bp->b_flags&B_DELWRI && (dev == NODEV||dev==bp->b_dev)) {
                        bp->b_flags =| B_ASYNC;
                        notavail(bp);
                        bwrite(bp);
                        goto loop;
                }
        }
        spl0();
}

/*
 * Raw I/O. The arguments are
 *      The strategy routine for the device
 *      A buffer, which will always be a special buffer
 *        header owned exclusively by the device for this purpose
 *      The device number
 *      Read/write flag
 * Essentially all the work is computing physical addresses and
 * validating them.
 */
physio(strat, abp, dev, rw)
struct buf *abp;
int (*strat)();
{
        register st uct buf *bp;
        register ch r *base;
        register int nb;
        int ts;

        bp = abp;
        base = u.u_base;
        /*
         * Check odd base, odd count, and address wraparound
         */
        if (base&01 || u.u_count&01 || base>=base+u.u_count)
                goto bad;
        ts = (u.u_tsize+127) & ~0177;
        if (u.u_sep)
                ts = 0;
        nb = (base>>6) & 01777;
        /*
         * Check overlap with text. (ts and nb now
         * in 64-byte clicks)
         */
        if (nb < ts)
                goto bad;
        /*
         * Check that transfer is either entirely in the
         * data or in the stack: that is, either
         * the end is in the data or the start is in the stack
         * (remember wraparound was already checked).
         */
        if ((((base+u.u_count)>>6)&01777) >= ts+u.u_dsize
```

p_bio.c

```
&& nb < 1024-u.u_ssize)

                goto bad;
        spl6();
        while (bp->b_flags&B_BUSY) {
                bp->b_flags =| B_WANTED;
                sleep(bp, PRIBIO);
        }
        bp->b_flags = B_BUSY | B_PHYS | rw;
        bp->b_dev = dev;
        /*
         * Compute physical address by simulating
         * the segmentation hardware.
         */
        bp->b_addr = base&077;
        base = (u.u_sep? UDSA: UISA)->r[nb>>7] + (nb&0177);
        bp->b_addr =+ base<<6;
        bp->b_xmem = (base>>10) & 077;
        bp->b_blkno = lshift(u.u_offset, -9);
        bp->b_wcount = -((u.u_count>>1) & 077777);
        bp->b_error = 0;
        u.u_procp->p_flag =| SLOCK;
        (*strat)(bp);
        spl6();
        while ((bp->b_flags&B_DONE) == 0)
                sleep(bp, PRIBIO);
        u.u_procp->p_flag =& ~SLOCK;
        if (bp->b_flags&B_WANTED)
                wakeup(bp);
        spl0();
        bp->b_flags =& ~(B_BUSY|B_WANTED);
        u.u_count = (-bp->b_resid)<<1;
        geterror(bp);
        return;
    bad:
        u.u_error = EFAULT;
}

/*
 * Pick up the device's error number and pass it to the user;
 * if there is an error but the number is 0 set a generalized
 * code.  Actually the latter is always true because devices
 * don't yet return specific errors.
 */
geterror(abp)
struct buf *abp;
{
        register struct buf *bp;

        bp = abp;
        if (bp->b_flags&B_ERROR)
                if ((u.u_error = bp->b_error)==0)
                        u.u_error = EIO;
}
```

p_pm.3.c

```
#

/*
 *      PM Disk Driver
 *
 *      Modified version of the original UNIX 6.0 RP11 driver
 *
 *      This driver has been modified to support the extended
 *      capacity of the PM DD11/80 Disk Drive. The PM DC11/80
 *      Disk controller performs an address conversion to en-
 *      sure complete software compatibility with older DEC
 *      controllers.
 *
 *      Data organization on the Disk is as follows:
 *
 *      Seen by software        actually
 *              612                823             cylinders
 *               20                  5             surfaces
 *               10                 30             sectors
 *
 *      This allows the use of 62.4 MBytes CDC9877 disk packs.
 *
 *      Minor device numbers 0 thru 31 refer to drive number 1,
 *      Minor device numbers 32 thru 63 refer to drive number 2,
 *      and so on.
 *      Since there are NPM drives in the configuration, up to
 *      NPM*32 minor's are legal.
 *
 *      jpa.fun'0180
 *
 *      Modified 05.09.80 to reduce size of pseudo-disks
 *      to 5000 blocks. /etc/mkfs them with 4872 blocks to
 *      have the same size as an RK cartridge.
 *      jpa.FUN
 */

#include "../param.h"
#include "../buf.h"
#include "../conf.h"
#include "../user.h"
#include "../perfo.h"

/* REGISTER LAYOUT */

#define PMADDR  0176710         /* starting address of registers */

struct{
        int     pmds;           /* 0176710: drive status */
        int     pmer;           /* 0176712: error */
        int     pmcs;           /* 0176714: control & status */
        int     pmwc;           /* 0176716: word count */
        int     pmba;           /* 0176720: bus address */
        int     pmca;           /* 0176722: cylinder address */
        int     pmda;           /* 0176724: disk address */
};

#define NPM     1               /* number of drives in config */
#define NPPM    28              /* number of minors per drive */

/* DATA ORGANIZATION */
```

p_pm.3.c

```
struct {
        char    *nblocks;       /* size in blocks */
        int     cyloff;         /* starting cylinder of minor */
} pm_sizes[] {

/*   size (blocks)      cylinder offset          minor   */

        5000,           0,              /*      0       */
        5000,           25,             /*      1       */
        5000,           50,             /*      2       */
        5000,           75,             /*      3       */
        5000,           100,            /*      4       */
        5000,           125,            /*      5       */
        5000,           150,            /*      6       */
        5000,           175,            /*      7       */
        5000,           200,            /*      8       */
        5000,           225,            /*      9       */
        5000,           250,            /*      10      */
        5000,           275,            /*      11      */
        5000,           300,            /*      12      */
        5000,           325,            /*      13      */
        5000,           350,            /*      14      */
        5000,           375,            /*      15      */
        5000,           400,            /*      16      */
        5000,           425,            /*      17      */
        5000,           450,            /*      18      */
        5000,           475,            /*      19      */
        5000,           500,            /*      20      */
        5000,           525,            /*      21      */
        5000,           550,            /*      22      */

/*      some bigger disks overlaying the other ones.    */

        7400,           575,            /*      23      */
        12400,          550,            /*      24      */
        17400,          525,            /*      25      */

/*      and some smaller ones for temporaries           */

        2400,           338,            /*      26      */
        2600,           325,            /*      27      */

};

/*      note that one cylinder equals 200 blocks        */

struct devtab   pmtab;
struct buf      rpmbuf;

#define GO      01
#define RESET   0
#define HSEEK   014             /* Home Seek */

#define IENABLE 0100            /* Interrupt Enable */
#define READY   0200

#define SUFU    01000           /* Selected Unit File Unsafe */
#define SUSU    02000           /* Selected Unit Seek Underway */
#define SUSI    04000           /* Selected Unit Seek Incomplete */
```

p_pm.3.c

```
#define HNF     010000          /* Header Not Found */

/*
 *      use av_back to save track+sector
 *      use b_resid for cylinder
 */

#define trksec  av_back
#define cylin   b_resid


/* STRATEGY ROUTINE */

pmstrategy(abp)
struct buf *abp;
{
        register struct buf *bp;
        register char *p1, *p2;

        bp = abp;
        if(bp->b_flags&B_PHYS)
                mapalloc(bp);    /* if we got a 11/70 */
        p1 = &pm_sizes[bp->b_dev.d_minor&037];
                /* bits 0 thru 5 give the actual minor */
                /* bits 6, 7 give the drive select */
        if(bp->b_dev.d_minor >= NPPM ||
           bp->b_blkno >= p1->nblocks) {
                bp->b_flags =| B_ERROR;
                iodone(bp);
                return;
        }
        bp->av_forw = 0;
        bp->cylin = p1->cyloff;
        p1 = bp->b_blkno;
        p2 = lrem(p1, 10);
        p1 = ldiv(p1, 10);
        bp->trksec = (p1%20)<<8 | p2;
        bp->cylin =+ p1/20;
        spl5();
        if((p1 = pmtab.d_actf) == 0)
                pmtab.d_actf = bp;
        else{
                for(; p2 = p1->av_forw; p1 = p2) {
                        if( p1->cylin <= bp->cylin
                         && bp->cylin <  p2->cylin
                         || p1->cylin >= bp->cylin
                         && bp->cylin >  p2->cylin)
                                break;
                }
                bp->av_forw = p2;
                p1->av_forw = bp;
        }
        if(pmtab.d_active == 0)
                pmstart();
        spl0();
}

/* START ROUTINE */

pmstart()
```

```
{
        register struct buf *bp;

        if((bp = pmtab.d_actf) == 0)
                return;
        pmtab.d_active++;
        PMADDR->pmda = bp->trksec;
        devstart(bp, &PMADDR->pmca, bp->cylin, bp->b_dev.d_minor>>5);
}

/* INTERRUPT ROUTINE */

pmintr()
{
        register struct buf *bp;
        register int ctr;
        register int drno;

        if(pmtab.d_active == 0)
                return;
        bp = pmtab.d_actf;
        pmtab.d_active = 0;
        if (PMADDR->pmcs < 0){              /* error bit */
                deverror(bp, PMADDR->pmer, PMADDR->pmds);
                if(PMADDR->pmds & (SUFU|SUSI|HNF)){
                        PMADDR->pmcs.lobyte = HSEEK|GO;
                        ctr = 0;
                        while((PMADDR->pmds&SUSU) && --ctr);
                }
                PMADDR->pmcs = RESET|GO;
                ctr = 0;
                while((PMADDR->pmcs&READY) == 0 && --ctr);
                if(++pmtab.d_errcnt <= 10) {
                        pmstart();
                        return;
                }
                bp->b_flags =| B_ERROR;
        }
        pmtab.d_errcnt = 0;
        pmtab.d_actf = bp->av_forw;
        bp->b_resid = PMADDR->pmwc;
        iodone(bp);

        /* system measures */

        drno = bp->b_dev.d_minor & 037;

        if(drno >= MPM00 && drno <= MPM31)
                {
                pmrun[drno] = SIDLE;
                addd(n_pmrun[drno], 1);
                }

        pmstart();
}

/* READ */
pmread(dev)
{
        if(pmphys(dev))
```

p_pm.3.c

```
        physio(pmstrategy, &rpmbuf, dev, B_READ);
}

/* WRITE */
pmwrite(dev)
{
        if(pmphys(dev))
        physio(pmstrategy, &rpmbuf, dev, B_WRITE);
}

/* PHYSIOLOGY */
pmphys(dev)
{
        register c;

        c = lshift(u.u_offset, -9);
        c =+ ldiv(u.u_count+511, 512);
        if(c > pm_sizes[dev.d_minor & 0371.nblocks) {
                u.u_error = ENXIO;
                return(0);
        }
        return(1);
}
```

p_tty.4.c

```
#
/*
Modified 4/7/76 to handle EAI and VT52  in similar ways
                em

Modified 11/16/76 to include RARE mode (same as RAW mode but recognizes
  the "signal" characters and generates the corresponding signal),
  to introduce a filter for EAI terminals
  and to correct a bug related to RAW mode for "REVERSE"
Modified 14/03/77 to structure "erase" and "kill" programming in ttyinput
  and to include the General Purpose signal (SIGGP).
Modified 12/20/77 to implement the TRANSP(arent) mode.
Modified 11/13/78 to implement XON/XOFF independently of output mechanism

Modified on Aug 79 : communication between UNIX and MPX .
  (either kl or mpx)
 */

/*
 *      Modified 020580 to remove EAI handling.
 *      simply because we don't have such beasts!
 *      jpa.fun
 *      Also modified to include MONITOR mode for
 *      use with BARCO monitors : put a tab after each cr
 */

/*
 *      Modified 290781 to measure reflexion time at ttye
 *
 *      Cornil Dirk.
 */

/*
 * general TTY subroutines
 */
#include "/binar/usr/sys/param.h"
#include "/binar/usr/sys/systm.h"
#include "/binar/usr/sys/user.h"
#include "/binar/usr/sys/tty.nh"
#include "/binar/usr/sys/proc.h"
#include "/binar/usr/sys/inode.h"
#include "/binar/usr/sys/file.h"
#include "/binar/usr/sys/reg.h"
#include "/binar/usr/sys/conf.h"
#include "/binar/usr/sys/drmxp.h"
#include "/binar/usr/sys/perfo.h"

/*
 * Input mapping table-- if an entry is non-zero, when the
 * corresponding character is typed preceded by "\" the escape
 * sequence is replaced by the table value.  Mostly used for
 * upper-case only terminals.
 */
char    maptab[]
{
        000,000,000,000,004,000,000,000,
        000,000,000,000,000,000,000,000,
        000,000,000,000,000,000,000,000,
        000,000,000,000,000,000,000,000,
        000,'|',000,'#',000,000,000,'~',
```

```
        '{', '}', 000, 000, 000, 000, 000, 000,
        000, 000, 000, 000, 000, 000, 000, 000,
        000, 000, 000, 000, 000, 000, 000, 000,
        '@', 000, 000, 000, 000, 000, 000, 000,
        000, 000, 000, 000, 000, 000, 000, 000,
        000, 000, 000, 000, 000, 000, 000, 000,
        000, 000, 000, 000, 000, 000, '~', 000,
        000, 'A', 'B', 'C', 'D', 'E', 'F', 'G',
        'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O',
        'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W',
        'X', 'Y', 'Z', 000, 000, 000, 000, 177,
};

/*
 * The actual structure of a clist block manipulated by
 * getc and putc (mch.s)
 */
struct cblock {
        struct cblock *c_next;
        char info[6];
};

/* The character lists-- space for 6*NCLIST characters */
struct cblock cfree[NCLIST];
/* List head for unused character blocks. */
struct cblock *cfreelist;

/*
 * The routine implementing the gtty system call.
 * Just call lower level routine and pass back values.
 */
gtty()
{
        int v[3];
        register *up, *vp;

        vp = v;
        sgtty(vp);
        if (u.u_error)
                return;
        up = u.u_arg[0];
        suword(up, *vp++);
        suword(++up, *vp++);
        suword(++up, *vp++);
}

/*
 * The routine implementing the stty system call.
 * Read in values and call lower level.
 */
stty()
{
        register int *up;

        up = u.u_arg[0];
        u.u_arg[0] = fuword(up);
        u.u_arg[1] = fuword(++up);
        u.u_arg[2] = fuword(++up);
        sgtty(0);
}
```

```
/*
 * Stuff common to stty and gtty.
 * Check legality and switch out to individual
 * device routine.
 * v  is 0 for stty; the parameters are taken from u.u_arg[].
 * c  is non-zero for gtty and is the place in which the device
 * routines place their information.
 */
sgtty(v)
int *v;
{
        register struct file *fp;
        register struct inode *ip;

        if ((fp = getf(u.u_ar0[R0])) == NULL)
                return;
        ip = fp->f_inode;
        if ((ip->i_mode&IFMT) != IFCHR) {
                u.u_error = ENOTTY;
                return;
        }
        (*cdevsw[ip->i_addr[0].d_major].d_sgtty)(ip->i_addr[0], v);
}

/*
 * Wait for output to drain, then flush input waiting.
 */
wflushtty(atp)
struct tty *atp;
{
        register struct tty *tp;

        tp = atp;
        spl5();
        while (tp->t_outq.c_cc) {
                tp->t_state =| ASLEEP;
                sleep(&tp->t_outq, TTOPRI);
        }
        flushtty(tp);
        spl0();
}

/*
 * Initialize clist by freeing all character blocks, then count
 * number of character devices. (Once-only routine)
 */
cinit()
{
        register int ccp;
        register struct cblock *cp;
        register struct cdevsw *cdp;

        ccp = cfree;
        for (cp=(ccp+07)&~07; cp <= &cfree[NCLIST-1]; cp++) {
                cp->c_next = cfreelist;
                cfreelist = cp;
        }
        ccp = 0;
        for(cdp = cdevsw; cdp->d_open; cdp++)
```

```
                ccp++;
        nchrdev = ccp;
}

/*
 * flush all TTY queues
 */
flushtty(atp)
struct tty *atp;
{
        register struct tty *tp;
        register int sps;

        tp = atp;
        while (getc(&tp->t_canq) >= 0);
        while (getc(&tp->t_outq) >= 0);
        wakeup(&tp->t_rawq);
        wakeup(&tp->t_outq);
        sps = PS->integ;
        spl5();
        while (getc(&tp->t_rawq) >= 0);
        tp->t_delct = 0;
        PS->integ = sps;
}

/*
 * transfer raw input list to canonical list,
 * doing erase-kill processing and handling escapes.
 * It waits until a full line has been typed in cooked mode,
 * or until any character has been typed in raw mode.
 */
canon(atp)
struct tty *atp;
{
        register char *bp;
        char *bp1;
        register struct tty *tp;
        register int c;
        int tyname, tyno, j;                    /* system measures */

        tp = atp;

        /* system measures */

        tyname = tp->t_dev.d_major & 027;
        tyno = tp->t_dev.d_minor & 017;

        spl5();
        while (tp->t_delct==0) {
                if ((tp->t_state&CARR_ON)==0)
                        {

                        /* system measures */

                        tyflag1 = REFNO;
                        ts_tty = 0;
                        return(0);
                        }

                sleep(&tp->t_rawq, TTIPRI);
```

```
                /* system measures */

                if(tyname == 16 && tyno == 2 && tyflag1 == REFYES)
                        {
                        tyflag1 = REFNO;

                        if(tycnt < SMAX)
                                {
                                tytime[tycnt] = ts_tty;
                                tycnt++;
                                }
                        else    {
                                tyflag2 = 1;
                                for(j=0; j<SMAX; j++)
                                        {
                                        tysave[j] = tytime[j];
                                        tytime[j] = 0;
                                        }
                                tycnt = 0;
                                }

                        ts_tty = 0;
                        }
        }
        spl0();

        if  (tp->t_flags & TRANSP) {
                if ((c=getc(&tp->t_rawq)) < 0) return(0);
                putc(c, &tp->t_canq);
                tp->t_delct--;
                return(1);
        }

loop:
        bp = &canonb[2];
        while ((c=getc(&tp->t_rawq)) >= 0) {
                if (c==0377) {
                        tp->t_delct--;
                        break;
                }
                if ((tp->t_flags&RAW)==0 && (tp->t_flags&RARE)==0) {
                        if (bp[-1]!='\\') {
                                if (c==tp->t_erase) {
                                        if (bp > &canonb[2])
                                                bp--;
                                        continue;
                                }
                                if (c==tp->t_kill)
                                        goto loop;
                                if (c==CEOT)
                                        continue;
                        } else
                        if (maptab[c] && (maptab[c]==c || (tp->t_flags&LCASE)
                                if (bp[-2] != '\\')
                                        c = maptab[c];
                                bp--;
                        }
                }
                *bp++ = c;
```

```
                if (bp>=canonb+CANBSIZ)
                        break;
        }
        bp1 = bp;
        bp = &canonb[2];
        c = &tp->t_canq;
        while (bp<bp1)
                putc(*bp++, c);
        return(1);
}

/*
 * Place a character on raw TTY input queue, putting in delimiters
 * and waking up top half as needed.
 * Also echo if required.
 * The arguments are the character and the appropriate
 * tty structure.
 */

char    fresc1[]        "\033:x\033*";
char    fresc2[]        "\033:\033x\033*";

ttyinput(ac, atp)
struct tty *atp;
{
        register int t_flags, c;
        register struct tty *tp;
        int *t_state;

        tp = atp;
        c = ac;
        t_flags = tp->t_flags;
        t_state = &tp->t_state;

        /* handle inncoming XOFF and XON (CTRL-S and CTRL-Q) to switch
           output stream off and on temporarily
        */


        if((t_flags & RAW) == 0 && (t_flags & TRANSP) == 0) { /* previously
                switch(c & 0177) {

                        case XOFF :     *t_state =| HOLD;
                                        return;

                        case XON :      *t_state =& ~HOLD;
                                        mxstart(tp);   /* restart output if n
                                        return;

                        default :       if(*t_state &HOLD) return;
                }
        }

                if (tp->t_rawq.c_cc>=TTYHOG) {
                        flushtty(tp);
                        return;
                }


        if  (tp->t_flags & TRANSP) {
```

```
                wakeup(&tp->t_rawq);
                if(putc(c, &tp->t_rawq) == 0)
                        tp->t_delct++;
                return;
        }

        if ((c =& 0177) == '\r' && t_flags&CRMOD)
                c = '\n';
/*
        else if((c == LARROW) && (t_flags&EAI)) c = BACKSP ;
                 because of EAI's odd conventions
 removed 020580 jpa
*/
        else if(t_flags&APPLE) {
                if(c == 013) c = '[';    /* ^K */
                else if(c == 002) c = '\\';      /* ^B */
                else if(c == 027) c = '_';       /* ^W */
        }
                /* this was added 12-feb-81 jpa'fun because the APPLE-II */
                /* has no complete ASCII keyboard. So we generate chars */
                /* we need as we can ! */
        else if ( (t_flags&RAW)==0 || (t_flags&RARE) )  {
                if  (c==CQUIT || c==CINTR) {
                        signal(tp, c==CINTR? SIGINT:SIGQIT);
                        flushtty(tp);
                        return;
                }
                if((t_flags&FRENCH)&&((t_flags&RARE)==0)) {
                        if(c == tp->t_kill || c == tp->t_erase) {
                                /*simulate backslash*/
                                putc('\\', &tp->t_rawq);
                                if(t_flags&ECHO) {
                                        *t_state =| BSLSW;
                                        if((*t_state&ESCSW) == 0)
                                                ttyecho('\\', tp);
                                }
                        }
                }
                if (c == ESC) {  *t_state =| ESCSW; return; }
                if (*t_state & ESCSW) {
                        /*general purpose signal*/
                        /*sorry, but a D CHAR on the BDM20 generates a ESC P
                        if((c=='P'&&(t_flags&FRENCH==0))|| c=='Q'|| c=='R'){
                                signal(tp, SIGQP);
                                goto escfull;
                        }
                        if((t_flags&FRENCH) && ((t_flags&RARE) == 0)) {
                                if(c=='M') { c = tp->t_kill; goto escpart; }
                                if(c=='P') { c = tp->t_erase; goto escpart;
                                /*special echo of other escape sequences*/
                                putc(ESC,&tp->t_rawq); putc(c,&tp->t_rawq);
                                *t_state =& ~ESCSW;
                                if(t_flags &ECHO) {
                                        fresc2[3] = c;
                                        frecho(fresc2, tp);
                                }
                                return;
                        }
escfull:
                        putc(ESC, &tp->t_rawq);
```

```
escpart:
                        *t_state =& ~ESCSW;
                }
        }

        if (t_flags&LCASE && c>='A' && c<='Z')
                c =+ 'a'-'A';

        putc(c, &tp->t_rawq);
        if (t_flags&RAW || t_flags&RARE || c=='\n' || c==CEOT) {
                wakeup(&tp->t_rawq);
                if (putc(0377, &tp->t_rawq)==0)
                        tp->t_delct++;
        }

        if (t_flags&ECHO) {
                if(t_flags&FRENCH) {
                        if(c==BACKSP || c == FF) {
                                fresc1[2] = c;
                                frecho(fresc1, tp);
                                return;
                        }
                }
                if ((t_flags&REVERSE) && (t_flags&RAW)==0 && (t_flags&RARE)==
                        /* erase and kill characters treatment */
                        if( c == '\\')
                                *t_state =| BSLSW; /* New switch ... 4/18/75*
                        else {
                                if ( c==tp->t_erase || c==tp->t_kill) {
                                        if ( (*t_state & BSLSW) == 0 ) {
                                                /* erase or kill */
                                                if ( c==tp->t_erase ) {
                                                        ttyoutput(BACKSP, tp
                                                        ttyerase(tp, 1);
                                                } else {
                                                        ttyoutput('\r', tp);
                                                        ttyerase(tp, 0);
                                                }
                                                goto noecho;
                                        }
                                        else if((*t_state&ESCSW) == 0)
                                                ttyoutput(BACKSP, tp);
                                                /*erase backslash*/
                                }
                                *t_state =& ~BSLSW;
                        }
                ttyecho(c, tp);
noecho:         mxstart(tp);
        }
}

/*
 *      additionnal routine for special echo
 *      on Micro Bee DM20 terminal
 *      (FRENCH mode)
 */
frecho(s, tp)
char *s; struct tty *tp;
{
        register char *p;
```

```
        p = s;
        while(*p)
                putc(*p++, &tp->t_outq);
        mxstart(tp);
}

/*
 * put character on TTY output queue, adding delays,
 * expanding tabs, and handling the CR/NL bit.
 * It is called both from the top half for output, and from
 * interrupt level for echoing.
 * The arguments are the character and the tty structure.
 */
ttyoutput(ac, atp)
struct tty *atp;
{
        register int c;
        register struct tty *tp;

        tp = atp;
        c = ac;
        if(tp->t_flags & TRANSP)
                putc(c, &tp->t_outq);
        else
                tttrans(c&0177, tp); /* was ttyswtch */
}


ttyecho(ac, tp)
struct tty *tp;
{
        struct tty *rtp;

        rtp = tp;
        ac =| 0200;
        if(rtp->t_state&ERMOD) {
                putc('\n', &rtp->t_outq);
                /* this is linefeed only, no newline !! */
                rtp->t_state =& ~ERMOD;
        }
        tttrans(ac, tp);  /* was ttyswtch */
}

/* comment out this routine ...

ttyswtch(ac, tp)
struct tty *tp;
{
        if(( tp ->t_flags&HCT302) && ((tp->t_flags & SCREEN) == 0)) {
                if(ac & 0200) {
                        COMMENT: We're echoing something
                        if( tp->t_state & BLACK ) {
                                tp->t_state =& ~BLACK;
                                tttrans(ESC, tp);
                                tttrans('3', tp);
                        }
                } else {
                        if((tp->t_state & BLACK) == 0) {
                                tp->t_state =| BLACK;
```

```
                                tttrans (ESC, tp);
                                tttrans('4', tp);
                        }
                }
        }
        tttrans(ac, tp);
}

 end of comment */

tttrans(ac, tp)
struct tty *tp;
{
        register int c;
        register struct tty *rtp;
        register char *colp;
        int ctype;

        rtp = tp;
        c = ac&0177;
        /*
         * Ignore EOT in normal mode to avoid hanging up
         * certain terminals.
         */
        if (c==CEOT && (rtp->t_flags&RAW)==0 && (rtp->t_flags&RARE)==0)
                return;
        /*
         * Turn tabs to spaces as required
         */
        if (c=='\t' && rtp->t_flags&XTABS) {
                do
                        ttyoutput(' ', rtp);
                while (rtp->t_col&07);
                return;
        }
        /*
         * for upper-case-only terminals,
         * generate escapes.
         */
        if (rtp->t_flags&LCASE) {
                colp = "({)}!|^~'`";
                while(*colp++)
                        if(c == *colp++) {
                                ttyoutput('\\', rtp);
                                c = colp[-2];
                                break;
                        }
                if ('a'<=c && c<='z')
                        c =+ 'A' - 'a';
        }
        /*
         * turn <nl> to <cr><lf> if desired.
         */
        if (c=='\n' && rtp->t_flags&CRMOD)
                ttyoutput('\r', rtp);
/*
         * take the odd EAI conventions into account
         * i.e. the left movement on the screen isn't
         * the ASCII BS character but LARROW
         *
```

```
        if((c == BACKSP) && (rtp->t_flags&EAI)) c = LARROW;
* removed 025080 jpa
*/
        if (putc(c, &rtp->t_outq))
                return;

        /* Take the stupid lack of a complete ASCII char set on the
         * APPLE-II keyboard into account and echo some controls
         * by chars we need. Sorry for this patch but this
         * should be only temporary. We'll try to fix this inside
         * the apple system and clean up UNIX with such garbage. */
        /* (I hope so) */
        if(rtp->t_flags&APPLE) {
                if(c == 013) c = '[';    /* ^K */
                else if(c == 002) c = '\\';      /* ^B */
                else if(c == 027) c = '_';       /* ^W */
        }
        /*
         * Calculate delays.
         * The numbers here represent clock ticks
         * and are not necessarily optimal for all terminals.
         * The delays are indicated by characters above 0200,
         * thus (unfortunately) restricting the transmission
         * path to 7 bits.
         */
        colp = &rtp->t_col;
        ctype = partab[c];
        /* Calculate correct tabs for the Cable Print - ESC x takes
         * no room at all or ESC is -1 column wide
         */
        if(c == ESC && rtp->t_flags&HCT302) (*colp)--;
        c = 0;
        switch (ctype&077) {

        /* ordinary */
        case 0:
                (*colp)++;

        /* non-printing */
        case 1:
                break;

        /* backspace */
        case 2:
                if (*colp)
                        (*colp)--;
                break;

        /* newline */
        case 3:
                ctype = (rtp->t_flags >> 8) & 03;
                if(ctype == 1) { /* tty 37 */
                        if (*colp)
                                c = max((*colp>>4) + 3, 6);
                } else
                if(ctype == 2) { /* vt05 */
                        c = 6;
                }
                *colp = 0;
                break;
```

```
        /* tab */
        case 4:
/*********************************************
 *              ctype = (rtp->t_flags >> 10) & 03;
 *              if(ctype == 1) {  COMMENT tty 37
 *                      c = 1 - (*colp | ~07);
 *                      if(c < 5)
 *                              c = 0;
 *              }
 ****************************removed 04.24.80 for FRENCH mode ** jpa */
                *colp =| 07;
                (*colp)++;
                break;

        /* vertical motion */
        /* carriage return */
        case 6:
                ctype = (rtp->t_flags >> 12) & 03;
                if(ctype == 1) { /* tn 300 */
                        c = 5;
                } else
                if(ctype == 2) { /* ti 700 */
                        c = 10;
                }
                *colp = 0;
        }
        if(c)
                putc(c|0200, &rtp->t_outq);
}

/*
 * Restart typewri er output following a delay
 * timeout.
 * The name of the routine is passed to the timeout
 * subroutine and it is called during a clock interrupt.
 */
ttrstrt(atp)
{
        register struct tty *tp;

        tp = atp;
        tp->t_state =& ~TIMEOUT;
        mxstart(tp);
}

/*
 * Start output on the typewriter. It is used from the top half
 * after some characters have been put on the output queue,
 * from the interrupt routine to transmit the next
 * character, and after a timeout has finished.
 * If the SSTART bit is off for the tty the work is done here,
 * using the protocol of the single-line interfaces (KL, DL, DC);
 * otherwise the address word of the tty structure is
 * taken to be the name of the device-dependent startup routine.
 */
ttstart(atp)
struct tty *atp;
{
        register int *addr, c;
```

```
        register struct tty *tp;
        int i;
        char *first;

        tp = atp;
        addr = tp->t_addr;
        c = tp->t_state;          /* c = aux. variable */

        if (c&TIMEOUT || c&HOLD)
                return(0);

        if ((c=getc(&tp->t_outq)) >= 0) {       /* now c = char. */
                if(c<=0177  ||  (tp->t_flags&TRANSP)) {
                        if ((tp->t_flags & TRANSP) == 0) {
                    /*
                                if ( (tp->t_flags&EAI) && ((c==010) || (c==01
                                                           (c==016) || (c==01
                                        c = 0;   * filter for EAI *
                                else
                     * removed 020580 jpa
                     */
                                        c =| (partab[c]&0200);
                        }
                        addr->tttcsr =| IENABLE;          /*  always turn intr
                        addr->tttbuf = c;
                } else {
                        timeout(ttrstrt, tp, c&0177);
                        tp->t_state =| TIMEOUT;
                }
                return(1);
        }
        else
                return(0);
}


/*
 * Called from device's read routine after it has
 * calculated the tty-structure given as argument.
 * The pc is backed up for the duration of this call.
 * In case of a caught interrupt, an RTI will re-execute.
 */
ttread(atp)
struct tty *atp;
{
        register struct tty *tp;
        register tyname, tyno;                  /* system measures */

        tp = atp;

        /* system measures */

        tyname = tp->t_dev.d_major & 027;
        tyno = tp->t_dev.d_minor & 017;

        if ((tp->t_state&CARR_ON)==0)
                return;

        /* system measures */
```

```
        if(tyname == 16 && tyno == 2)
                tyflag1 = REFYES;

        if (tp->t_canq.c_cc || canon(tp))
                while (tp->t_canq.c_cc && passc(getc(&tp->t_canq))>=0);
}

/*
 * Called from the device's write routine after it has
 * calculated the tty-structure given as argument.
 */
ttwrite(atp)
struct tty *atp;
{
        register struct tty *tp;
        register int c;

        tp = atp;
        if ((tp->t_state&CARR_ON)==0)
                return;
        while ((c=cpass())>=0) {
                spl5();
                while (tp->t_outq.c_cc > TTHIWAT) {
                        mxstart(tp);
                        tp->t_state =| ASLEEP;
                        sleep(&tp->t_outq, TTOPRI);
                }
                spl0();
                ttyoutput(c, tp);
        }
        spl5();  /* see UNIX News 2:8 */
        mxstart(tp);
        spl0();
}

/*
 * Common code for gtty and stty functions on typewriters.
 * If v is non-zero then gtty is being done and information is
 * passed back therein;
 * if it is zero stty is being done and the input information is in the
 * u_arg array.
 */
ttystty(atp, av)
int *atp, *av;
{
        register  *tp, *v;

        tp = atp;
        if(v = av) {
                *v++ = tp->t_speeds;
                v->lobyte = tp->t_erase;
                v->hibyte = tp->t_kill;
                v[1] = tp->t_flags;
                return(1);
        }
        wflushtty(tp);
        v = u.u_arg;
        tp->t_speeds = *v++;
        tp->t_erase = v->lobyte;
        tp->t_kill = v->hibyte;
```

p_tty.4.c

```
        tp->t_flags = v[1];
        return(0);
}
ttyerase(tp,c)
struct  tty *tp;
        {
        register int i;
        register struct tty *rtp;

        rtp = tp;
        if(rtp->t_flags&HCT302){
                ttyoutput('X', rtp); ttyoutput(BACKSP, rtp);
                ttyoutput('O', rtp); ttyoutput(BACKSP, rtp);
                rtp->t_state =| ERMOD;
        } else {
                if(c) {
                        ttyoutput(SPACE,rtp);
                        ttyoutput(BACKSP,rtp);
                } else
                {
                        if(rtp->t_flags&FRENCH) {
                                ttyoutput(ESC, rtp);
                                ttyoutput('M', rtp);
                        } else {
                                for(i=0; i<80; i++) ttyoutput(' ', rtp);
                                ttyoutput('\r', rtp);
                        }
                }
        }
}
/*
 * Sort out tty terminals for non-interrupt output
*/
mxstart(adp) struct tty *adp; {
        register struct tty *dp;
        register int *addr;
        struct { int (*func)(); } ;

        dp = adp;

        addr = dp->t_addr;

        if (dp->t_state & SSTART) {
                (*addr.func)(dp);
                return;
        }

        if ((addr->tttcsr & DONE) == 0)
                return;
        ttstart(dp);
}
```

p_rdwrix.c

```
#
/*
 */

#include "../param.h"
#include "../inode.h"
#include "../user.h"
#include "../buf.h"
#include "../conf.h"
#include "../systm.h"
#include "../seg.h"
#include "../proc.h"
#include "../reg.h"
#include "../perfo.h"


/*
 * Read the file corresponding to
 * the inode pointed at by the argument.
 * The actual read arguments are found
 * in the variables:
 *      u_base          core address for destination
 *      u_offset        byte offset in file
 *      u_count         number of bytes to read
 *      u_segflg        read to kernel/user
 */
/*      Amended to allow reading of zero characters on a
 *      character device (needed for semaphores), by moving
 *      the test for null u.u_count after the call to the
 *      d_read routine of cdevsw; this is consistent with
 *      what happens in writei.
 */

/*
 * function  modification : May 1979/pdc
 *
 *
 *    If the read command must transfer a number of bytes larger
 *    than 512, and if some of the physical blocks which are
 *    involved happen to be contiguous, as many bytes as
 *    possible are directly read in at
 *    the base address u.u_base by nfisio() instead of first being
 *    read into buffers and then transfered one at a time to the
 *    base address.
 *
 *    Function bmapc is used to compute 'u.u_rdblkc', i.e. the number
 *    of contiguous blocks which may be read by one single read operation.

 *    This requires the addition of a new variable to the _u structure:
 *    u.u_rdblkc.
 */
struct buf rxkbuf;
int ffflag;

readi(aip)
struct inode *aip;
{
        int *bp;
        int lbn, bn, on;
        register  n;
```

```
        int dn;
        int drname, drno;                       /* system measures */
        register struct inode *ip;


        ip = aip;

        ip->i_flag =| IACC;
        if((ip->i_mode&IFMT) == IFCHR) {

                /* system measures */

                drname = ip->i_addr[0].d_major;
                drno = ip->i_addr[0].d_minor & 037;

                if(drname == MRK)
                        addd(rkrout[drno],1);
                else if(drname == MPM)
                        addd(pmrout[drno],1);

                (*cdevsw[ip->i_addr[0].d_major].d_read)(ip->i_addr[0]);
                return;
        }
        if(u.u_count == 0) {
                return;
        }

        do {
                lbn = bn = lshift(u.u_offset, -9);
                on = u.u_offset[1] & 0777;
                n = min(512-on, u.u_count);
                if((ip->i_mode&IFMT) != IFBLK) {
                        dn = dpcmp(ip->i_size0&0377, ip->i_size1,
                                u.u_offset[0], u.u_offset[1]);
                        if(dn <= 0) {
                                return;
                        }
                        n = min(n, dn);

/* procedure 'bmapc()' replaces 'bmap()' of the original unix */

                        if ((bn = bmapc(ip,lbn,on)) == 0) {
                                return;
                        }
                        dn = ip->i_dev;
                }
                else {
                        dn = ip->i_addr[0];
                        rablock = bn+1;
                        u.u_rdblkc = ((u.u_count + 511)<<9);
                }

/* 'u.u_rdblkc' contains the number of contiguous blocks counting from
 * the returned block address.
 */

                if((u.u_rdblkc > 1) && (u.u_count > 512)) {
                        nfisio(bn,dn);
                }
                else {
```

```
                        if (ip->i_lastr+1 == lbn)
                                bp = breada(dn, bn, rablock);
                        else
                                bp = bread(dn, bn);
                        ip->i_lastr = lbn;
                        iomove(bp, on, n, B_READ);
                        brelse(bp);
                }

        } while(u.u_error==0 && u.u_count!=0);
if (u.u_error)
        printf("er=0%o;\n", u.u_error);
}

/* ---------------------  */


/* 1) Like the original bmap(), function bmapc() converts the logical
 *    block number ( parameter 'bn') of the block to be read
 *    into a physical block number which it returns. The
 *    parameter 'on' is the offset of the first byte to be read
 *    in this block.

 * 2) At the same time, bmapc() counts the number of blocks
 *    following the block to be read and which are contiguous on
 *    the block device ; it stores this result into 'u.u_rdblkc'.
 */

bmapc(ip, bn, on)

struct inode *ip;
int bn;
int on;

{
        int snb, ssnb, scount;
        int firstt, d;
        int *bp;
        register i, nb;
        register *bap;

/* initialisations */

        u.u_rdblkc = 1;
        firstt = ssnb = 0;
        d = ip->i_dev;

/* read offset inside block not equal to 0 or odd base */


        if ((on) || (u.u_base&01)) {
                return(bmap(ip,bn));
        }

        scount = (u.u_count - 1) >> 9;

/* small file */
        if ((ip->i_mode&ILARG) == 0) {
                ssnb = snb = ip->i_addr[bn];
```

```
                scount = min(bn+scount, 7);
                for(bn=bn+1; bn<=scount; bn++)
                        if (++snb != ip->i_addr[bn])
                                return(ssnb);
                        else
                                u.u_rdblkc++;
                return(ssnb);
        }

/* large file algorithm:
 * determine which i_addr[] should be used; each i_addr[] points
 * to 256 physical block numbers
 */

nxti:

        i = bn>>8;
        if (bn & 0174000)
                i = 7;
        nb = ip->i_addr[i];
        bp = bread(d, nb);
        bap = bp->b_addr;

/* case of double indirection in the addressing of physical blocks */

        if (i == 7) {
                i = ((bn>>8) & 0377) - 7;
                nb = bap[i];
                brelse(bp);
                bp = bread(d, nb);
                bap = bp->b_addr;
        }

/* b_addr points to the block containing the physical block number
 * of 'bn';
 */


        if (firstt == 0) {
                firstt = 1;
                i = bn&0377;
                ssnb = snb = bap[i];
                bn++;
                u.u_rdblkc = 1;
        }

/* loop counts the number of contiguous blocks in the 'mapping' block */

loop:

        for(i = bn&0377; i < 256; i++) {
                if ((u.u_rdblkc >= scount) || (++snb != bap[i])) {

                        /* if block count limit is reached
                         * or current block not contiguous
                         * to the previous one: return;
                         */

                        rablock = 0;
                        if ((i<255) && (u.u_rdblkc<=1))
```

```
                                rablock = bap[i+1];
                        brelse(bp);
                        return(ssnb);
                }
                else {
                        bn++;
                        u.u_rdblkc++;
                }
        }

/* if the count limit has not been reached
 * read the block containing the next 256 block numbers
 */

        brelse(bp);
        if (u.u_rdblkc >= scount) {
                rablock = 0;
                return(ssnb);
        }
        goto nxt1;
}

/* ------------------------- */


/*
 *  Called by readi() to read 'u.u_rdblkc' contiguous physical blocks
 * directly into main storage at the address u.u_base;
 * the global I/O parameters: u.u_base, u.u_count, and u.u_offset
 * are updated accordingly;
 */

nfisio(b,dev)
int b,dev;

{
        register struct buf *bp;
        register char *base;
        register int nb;
        int n;
        int drname, drno;                       /* system measures */


        bp = &rxkbuf;
        base = u.u_base;
        nb = (base>>6)&01777;

if (base&01) {
        printf("badbase");
        goto bad;
}

                                        /* this value must be even */


        spl6();
        while (bp->b_flags&B_BUSY) {
                bp->b_flags =| B_WANTED;
                sleep(bp,PRIBIO);
        }
```

```
        bp->b_flags = B_BUSY | B_PHYS | B_READ;
        bp->b_dev = dev;

/* compute the base address in core for the read operation;
 * it will always be situated in user space !
 */

        bp->b_addr = base&077;
        base = (u.u_sep? UDSA: UISA)->r[nb>>7] + ( nb&0177);
        bp->b_addr =+ base<<6;
        bp->b_xmem = (base>>10) & 077;

        bp->b_blkno = b;
        n = (u.u_rdblkc << 9) & 077777;
        if (n > u.u_count) n = u.u_count;
        bp->b_wcount = -(n >> 1);
        bp->b_error = 0;
        u.u_procp->p_flag =| SLOCK;

        /* system measures */

        drname = dev.d_major;
        drno = dev.d_minor & 037;

        if(drname == MRK)
                addd(rkrout[drno], 1);
        else if(drname == MPM)
                addd(pmrout[drno], 1);

        (*bdevsw[dev.d_major].d_strategy)(bp);

        spl6();
        while ((bp->b_flags&B_DONE) == 0)
                sleep(bp, PRIBIO);
        u.u_procp->p_flag =& ~SLOCK;
        if (bp->b_flags&B_WANTED)
                wakeup(bp);

        bp->b_flags =& ~(B_BUSY|B_WANTED);
        geterror(bp);
        spl0();

/* update I/O control variables in the '_u' structure */
        u.u_count =- n;
        u.u_base =+ n;
        dpadd(u.u_offset, n);

        return;
bad:
        u.u_error = EFAULT;
}

/*
 * Write the file corresponding to
 * the inode pointed at by the argument.
 * The actual write arguments are found
 * in the variables:
 *      u_base          core address for source
 *      u_offset        byte offset in file
```

```
 *      u_count         number of bytes to write
 *      u_segflg        write to kernel/user
 */
writei(aip)
struct inode *aip;
{
        int *bp;
        int n, on;
        int drname, drno;                       /* system measures */
        register dn, bn;
        register struct inode *ip;

        ip = aip;
        ip->i_flag =| IACC|IUPD;
        if((ip->i_mode&IFMT) == IFCHR) {

                /* system measures */

                drname = ip->i_addr[0].d_major;
                drno = ip->i_addr[0].d_minor & 037;

                if(drname == MRK)
                        addd(rkrout[drno], 1);
                else if(drname == MPM)
                        addd(pmrout[drno], 1);

                (*cdevsw[ip->i_addr[0].d_major].d_write)(ip->i_addr[0]);
                return;
        }
        if (u.u_count == 0)
                return;

        do {
                bn = lshift(u.u_offset, -9);
                on = u.u_offset[1] & 0777;
                n = min(512-on, u.u_count);
                if((ip->i_mode&IFMT) != IFBLK) {
                        if ((bn = bmap(ip, bn)) == 0)
                                return;
                        dn = ip->i_dev;
                } else
                        dn = ip->i_addr[0];
                if(n == 512)
                        bp = getblk(dn, bn); else
                        bp = bread(dn, bn);
                iomove(bp, on, n, B_WRITE);
                if(u.u_error != 0)
                        brelse(bp); else
                if ((u.u_offset[1]&0777)==0)
                        bawrite(bp); else
                        bdwrite(bp);
                if(dpcmp(ip->i_size0&0377, ip->i_size1,
                  u.u_offset[0], u.u_offset[1]) < 0 &&
                  (ip->i_mode&(IFBLK&IFCHR)) == 0) {
                        ip->i_size0 = u.u_offset[0];
                        ip->i_size1 = u.u_offset[1];
                }
                ip->i_flag =| IUPD;
        } while(u.u_error==0 && u.u_count!=0);
}
```

```
/*
 * Return the logical maximum
 * of the 2 arguments.
 */
max(a, b)
char *a, *b;
{

        if(a > b)
                return(a);
        return(b);
}

/*
 * Return the logical minimum
 * of the 2 arguments.
 */
min(a, b)
char *a, *b;
{

        if(a < b)
                return(a);
        return(b);
}

/*
 * Move 'an' bytes at byte location
 * &bp->b_addr[o] to/from (flag) the
 * user/kernel (u.segflg) area starting at u.base.
 * Update all the arguments by the number
 * of bytes moved.
 *
 * There are 2 algorithms,
 * if source address, dest address and count
 * are all even in a user copy,
 * then the machine language copyin/copyout
 * is called.
 * If not, its done byte-by-byte with
 * cpass and passc.
 */
iomove(bp, o, an, flag)
struct buf *bp;
{
        register char *cp;
        register int n, t;

        n = an;
        cp = bp->b_addr + o;
        if(u.u_segflg==0 && ((n | cp | u.u_base)&01)==0) {
                if (flag==B_WRITE)
                        cp = copyin(u.u_base, cp, n);
                else
                        cp = copyout(cp, u.u_base, n);
                if (cp) {
                        u.u_error = EFAULT;
                        return;
                }
                u.u_base =+ n;
```

```
                dpadd(u.u_offset, n);
                u.u_count -= n;
                return;
        }
        if (flag==B_WRITE) {
                while(n--) {
                        if ((t = cpass()) < 0)
                                return;
                        *cp++ = t;
                }
        } else
                while (n--)
                        if(passc(*cp++) < 0)
                                return;
}
```

```
#
/*
 */

#include "../param.h"
#include "../systm.h"
#include "../reg.h"
#include "../buf.h"
#include "../filsys.h"
#include "../user.h"
#include "../inode.h"
#include "../file.h"
#include "../conf.h"
#include "../perfo.h"

/*
 * the fstat system call.
 */
fstat()
{
        register *fp;

        fp = getf(u.u_ar0[R0]);
        if(fp == NULL)
                return;
        stat1(fp->f_inode, u.u_arg[0]);
}

/*
 * the stat system call.
 */
stat()
{
        register ip;
        extern uchar;

        ip = namei(&uchar, 0);
        if(ip == NULL)
                return;
        stat1(ip, u.u_arg[1]);
        iput(ip);
}

/*
 * The basic routine for fstat and stat:
 * get the inode and pass appropriate parts back.
 */
stat1(ip, ub)
int *ip;
{
        register i, *bp, *cp;

        iupdat(ip, time);
        bp = bread(ip->i_dev, ldiv(ip->i_number+31, 16));
        cp = bp->b_addr + 32*lrem(ip->i_number+31, 16) + 24;
        ip = &(ip->i_dev);
        for(i=0; i<14; i++) {
                suword(ub, *ip++);
                ub =+ 2;
        }
```

```
        for(i=0; i<4; i++) {
                suword(ub, *cp++);
                ub =+ 2;
        }
        brelse(bp);
}

/*
 * the dup system call.
 */
dup()
{
        register i, *fp;

        fp = getf(u.u_ar0[R0]);
        if(fp == NULL)
                return;
        if ((i = ufalloc()) < 0)
                return;
        u.u_ofile[i] = fp;
        fp->f_count++;
}

/*
 * the mount system call.
 */
smount()
{
        int d;
        int drname, drno;
        register *ip;
        register struct mount *mp, *smp;
        extern uchar;

        d = getmdev();
        if(u.u_error)
                return;
        u.u_dirp = u.u_arg[1];
        ip = namei(&uchar, 0);
        if(ip == NULL)
                return;
        if(ip->i_count!=1 || (ip->i_mode&(IFBLK&IFCHR))!=0)
                goto out;
        smp = NULL;
        for(mp = &mount[0]; mp < &mount[NMOUNT]; mp++) {
                if(mp->m_bufp != NULL) {
                        if(d == mp->m_dev)
                                goto out;
                } else
                if(smp == NULL)
                        smp = mp;
        }
        if(smp == NULL)
                goto out;

        /* system measures */

        drname = d.d_major;
        drno = d.d_minor & 037;
```

```
        if(drname == MRK)
                addd(rkrout[drno], 1);
        else if(drname == MPM)
                addd(pmrout[drno], 1);

        (*bdevsw[d.d_major].d_open)(d, !u.u_arg[2]);
        if(u.u_error)
                goto out;
        mp = bread(d, 1);
        if(u.u_error) {
                brelse(mp);
                goto out1;
        }
        smp->m_inodp = ip;
        smp->m_dev = d;
        smp->m_bufp = getblk(NODEV);
        bcopy(mp->b_addr, smp->m_bufp->b_addr, 256);
        smp = smp->m_bufp->b_addr;
        smp->s_ilock = 0;
        smp->s_flock = 0;
        smp->s_ronly = u.u_arg[2] & 1;
        brelse(mp);
        ip->i_flag =| IMOUNT;
        prele(ip);
        return;

out:
        u.u_error = EBUSY;
out1:
        iput(ip);
}

/*
 * the umount system call.
 */
sumount()
{
        int d;
        int drname, drno;                       /* system measures */
        register struct inode *ip;
        register struct mount *mp;

        update();
        d = getmdev();
        if(u.u_error)
                return;
        for(mp = &mount[0]; mp < &mount[NMOUNT]; mp++)
                if(mp->m_bufp!=NULL && d==mp->m_dev)
                        goto found;
        u.u_error = EINVAL;
        return;

found:
        for(ip = &inode[0]; ip < &inode[NINODE]; ip++)
                if(ip->i_number!=0 && d==ip->i_dev) {
                        u.u_error = EBUSY;
                        return;
                }

        /* system measures */
```

```
        drname = d.d_major;
        drno = d.d_minor & 037;

        if(drname == MRK)
                addd(rkrout[drno], 1);
        else if(drname == MPM)
                addd(pmrout[drno], 1);

        (*bdevsw[d.d_major].d_close)(d, 0);
        ip = mp->m_inodp;
        ip->i_flag =& ~IMOUNT;
        iput(ip);
        ip = mp->m_bufp;
        mp->m_bufp = NULL;
        brelse(ip);
}

/*
 * Common code for mount and umount.
 * Check that the user's argument is a reasonable
 * thing on which to mount, and return the device number if so.
 */
getmdev()
{
        register d, *ip;
        extern uchar;

        ip = namei(&uchar, 0);
        if(ip == NULL)
                return;
        if((ip->i_mode&IFMT) != IFBLK)
                u.u_error = ENOTBLK;
        d = ip->i_addr[0];
        if(ip->i_addr[0].d_major >= nblkdev)
                u.u_error = ENXIO;
        iput(ip);
        return(d);
}
```

```
#
/*
 */

/*
 * Everything in this file is a routine implementing a system call.
 */

#include "../param.h"
#include "../user.h"
#include "../reg.h"
#include "../inode.h"
#include "../systm.h"
#include "../proc.h"
#include "../perfo.h"

getswit()
{

        u.u_ar0[R0] = SW->integ;
}

gtime()
{

        u.u_ar0[R0] = time[0];
        u.u_ar0[R1] = time[1];
}

stime()
{

        if(suser()) {
                time[0] = u.u_ar0[R0];
                time[1] = u.u_ar0[R1];
                wakeup(tout);
        }
}

setuid()
{
        register uid;

        uid = u.u_ar0[R0].lobyte;
        if(u.u_ruid == uid.lobyte || suser()) {
                u.u_uid = uid;
                u.u_procp->p_uid = uid;
                u.u_ruid = uid;
        }
}

getuid()
{

        u.u_ar0[R0].lobyte = u.u_ruid;
        u.u_ar0[R0].hibyte = u.u_uid;
}

setgid()
{
```

```
        register gid;

        gid = u.u_ar0[R0].lobyte;
        if(u.u_rgid == gid.lobyte || suser()) {
                u.u_gid = gid;
                u.u_rgid = gid;
        }
}

getgid()
{

        u.u_ar0[R0].lobyte = u.u_rgid;
        u.u_ar0[R0].hibyte = u.u_gid;
}

getpid()
{
        u.u_ar0[R0] = u.u_procp->p_pid;
}

sync()
{

        update();
}

nice()
{
        register n;

        n = u.u_ar0[R0];
        if(n > 20)
                n = 20;
        if(n < 0 && !suser())
                n = 0;
        u.u_procp->p_nice = n;
}

/*
 * Unlink system call.
 * panic: unlink -- "cannot happen"
 */
unlink()
{
        register *ip, *pp;
        extern uchar;

        pp = namei(&uchar, 2);
        if(pp == NULL)
                return;
        prele(pp);
        ip = iget(pp->i_dev, u.u_dent.u_ino);
        if(ip == NULL)
                panic("unlink -- iget");
        if((ip->i_mode&IFMT)==IFDIR && !suser())
                goto out;
        u.u_offset[1] -= DIRSIZ+2;
        u.u_base = &u.u_dent;
        u.u_count = DIRSIZ+2;
```

```
        u.u_dent.u_ino = 0;
        writei(pp);
        ip->i_nlink--;
        ip->i_flag =| IUPD;

out:
        iput(pp);
        iput(ip);
}

chdir()
{
        register *ip;
        extern uchar;

        ip = namei(&uchar, 0);
        if(ip == NULL)
                return;
        if((ip->i_mode&IFMT) != IFDIR) {
                u.u_error = ENOTDIR;
        bad:
                iput(ip);
                return;
        }
        if(access(ip, IEXEC))
                goto bad;
        iput(u.u_cdir);
        u.u_cdir = ip;
        prele(ip);
}

chmod()
{
        register *ip;

        if ((ip = owner()) == NULL)
                return;
        ip->i_mode =& ~07777;
        if (u.u_uid)
                u.u_arg[1] =& ~ISVTX;
        ip->i_mode =| u.u_arg[1]&07777;
        ip->i_flag =| IUPD;
        iput(ip);
}

chown()
{
        register *ip;

        if (!suser() || (ip = owner()) == NULL)
                return;
        ip->i_uid = u.u_arg[1].lobyte;
        ip->i_gid = u.u_arg[1].hibyte;
        ip->i_flag =| IUPD;
        iput(ip);
}

/*
 * Change modified date of file:
 * time to r0-r1; sys smdate; file
```

```
 * This call has been withdrawn because it messes up
 * incremental dumps (pseudo-old files aren't dumped).
 * It works though and you can uncomment it if you like.

smdate()
{
        register struct inode *ip;
        register int *tp;
        int tbuf[2];

        if ((ip = owner()) == NULL)
                return;
        ip->i_flag =| IUPD;
        tp = &tbuf[2];
        *--tp = u.u_ar0[R1];
        *--tp = u.u_ar0[R0];
        iupdat(ip, tp);
        ip->i_flag =& ~IUPD;
        iput(ip);
}
*/

ssig()
{
        register a;

        a = u.u_arg[0];
        if(a<=0 || a>=NSIG || a ==SIGKIL) {
                u.u_error = EINVAL;
                return;
        }
        u.u_ar0[R0] = u.u_signal[a];
        u.u_signal[a] = u.u_arg[1];
        if(u.u_procp->p_sig == a)
                u.u_procp->p_sig = 0;
}

kill()
{
        register struct proc *p, *q;
        register a;
        int f;

        f = 0;
        a = u.u_ar0[R0];
        q = u.u_procp;
        for(p = &proc[0]; p < &proc[NPROC]; p++) {
                if(p == q)
                        continue;
                if(a != 0 && p->p_pid != a)
                        continue;
                if(a == 0 && (p->p_ttyp != q->p_ttyp || p <= &proc[1]))
                        continue;
                if(u.u_uid != 0 && u.u_uid != p->p_uid)
                        continue;
                f++;
                psignal(p, u.u_arg[0]);
        }
        if(f == 0)
                u.u_error = ESRCH;
```

```
}

times()
{
        register *p;

        for(p = &u.u_utime; p  < &u.u_utime+6; ) {
                suword(u.u_arg[0], *p++);
                u.u_arg[0] =+ 2;
        }
}

profil()
{

        u.u_prof[0] = u.u_arg[0] & ~1;   /* base of sample buf */
        u.u_prof[1] = u.u_arg[1];        /* size of same */
        u.u_prof[2] = u.u_arg[2];        /* pc offset */
        u.u_prof[3] = (u.u_arg[3]>>1) & 077777; /* pc scale */
}

reboot()
{
        printf("system halt.  \n");
        update();                   /* perform a sync */
        aloop();                    /* jump to hardware boot */
                                    /* this is machine dependent and asks for
                                    a little modification in ../conf/l.s */
}

        /* system measures */

svtime()
{
        register j;
        register *p;

        p = &tyflag2;
        suword(u.u_arg[0],*p);
        u.u_arg[0] =+ 2;

        for(j=0; j<SMAX; j++)
                {
                p = &tytime[j];
                suword(u.u_arg[0],*p);
                u.u_arg[0] =+ 2;
                }
        tyflag2 = 0;
}
```

p_sysent.c

```
#
/*
 */

/*
 * This table is the switch used to transfer
 * to the appropriate routine for processing a system call.
 * Each row contains the number of arguments expected
 * and a pointer to the routine.
 */
int     sysent[]
{
        0, &nullsys,                    /*  0 = indir */
        0, &rexit,                      /*  1 = exit */
        0, &fork,                       /*  2 = fork */
        2, &read,                       /*  3 = read */
        2, &write,                      /*  4 = write */
        2, &open,                       /*  5 = open */
        0, &close,                      /*  6 = close */
        0, &wait,                       /*  7 = wait */
        2, &creat,                      /*  8 = creat */
        2, &link,                       /*  9 = link */
        1, &unlink,                     /* 10 = unlink */
        2, &exec,                       /* 11 = exec */
        1, &chdir,                      /* 12 = chdir */
        0, &gtime,                      /* 13 = time */
        3, &mknod,                      /* 14 = mknod */
        2, &chmod,                      /* 15 = chmod */
        2, &chown,                      /* 16 = chown */
        1, &sbreak,                     /* 17 = break */
        2, &stat,                       /* 18 = stat */
        2, &seek,                       /* 19 = seek */
        0, &getpid,                     /* 20 = getpid */
        3, &smount,                     /* 21 = mount */
        1, &sumount,                    /* 22 = umount */
        0, &setuid,                     /* 23 = setuid */
        0, &getuid,                     /* 24 = getuid */
        0, &stime,                      /* 25 = stime */
        3, &ptrace,                     /* 26 = ptrace */
        0, &nosys,                      /* 27 = x */
        1, &fstat,                      /* 28 = fstat */
        0, &nosys,                      /* 29 = x */
        1, &nullsys,                    /* 30 = smdate; inoperative */
        1, &stty,                       /* 31 = stty */
        1, &gtty,                       /* 32 = gtty */
        0, &nosys,                      /* 33 = x */
        0, &nice,                       /* 34 = nice */
        0, &sslep,                      /* 35 = sleep */
        0, &sync,                       /* 36 = sync */
        1, &kill,                       /* 37 = kill */
        0, &getswit,                    /* 38 = switch */
        0, &pinit,                      /* 39 = pinit; system measures */
        1, &svtime,                     /* 40 = svtime; system measures */
        0, &dup,                        /* 41 = dup */
        0, &pipe,                       /* 42 = pipe */
        1, &times,                      /* 43 = times */
        4, &profil,                     /* 44 = prof */
        0, &nosys,                      /* 45 = tiu */
        0, &setgid,                     /* 46 = setgid */
        0, &getgid,                     /* 47 = getgid */
```

p_sysent.c

```
        2, &ssig,                       /* 48 = sig */
        0, &nosys,                      /* 49 = x */
        0, &nosys,                      /* 50 = x */
        0, &nosys,                      /* 51 = x */
        0, &nosys,                      /* 52 = x */
        0, &nosys,                      /* 53 = x */
        0, &nosys,                      /* 54 = x */
        0, &nosys,                      /* 55 = x */
        0, &nosys,                      /* 56 = x */
        0, &nosys,                      /* 57 = x */
        0, &nosys,                      /* 58 = x */
        0, &nosys,                      /* 59 = x */
        0, &nosys,                      /* 60 = x */
        0, &nosys,                      /* 61 = x */
        0, &setmagtape,                 /* 62 = setmagtape */
        0, &nosys                       /* 63 = x */
};
```

p_clock.c

```
#
#include "../param.h"
#include "../systm.h"
#include "../user.h"
#include "../proc.h"
#include "../perfo.h"

#define UMODE   0170000
#define SCHMAG  10

/*
 * clock is called straight from
 * the real time clock interrupt.
 *
 * Functions:
 *      reprime clock
 *      copy *switches to display
 *      implement callouts
 *      maintain user/system times
 *      maintain date
 *      profile
 *      tout wakeup (sys sleep)
 *      lightning bolt wakeup (every 4 sec)
 *      alarm clock signals
 *      job the scheduler
 */
clock(dev, sp, r1, nps, r0, pc, ps)
{
        register struct callo *p1, *p2;
        register struct proc *pp;
        int i;

        /*
         * restart clock
         */

        *lks = 0115;

        /*
         * display register
         */

        display();

        /*
         * callouts
         * if none, just return
         * else update first non-zero time
         */

        if(callout[0].c_func == 0)
                goto out;
        p2 = &callout[0];
        while(p2->c_time<=0 && p2->c_func!=0)
                p2++;
        p2->c_time--;

        /*
         * if ps is high, just return
         */
```

```
        if((ps&0340) != 0)
                goto out;

        /*
         * callout
         */

        spl5();
        if(callout[0].c_time <= 0) {
                p1 = &callout[0];
                while(p1->c_func != 0 && p1->c_time <= 0) {
                        (*p1->c_func)(p1->c_arg);
                        p1++;
                }
                p2 = &callout[0];
                while(p2->c_func = p1->c_func) {
                        p2->c_time = p1->c_time;
                        p2->c_arg = p1->c_arg;
                        p1++;
                        p2++;
                }
        }

        /*
         * lightning bolt time-out
         * and time of day
         */

out:
        /* system measures */

        if(t_loop == CTICKS)
                {
                t_loop = 0;

                if(cpurun == SACTIVE)
                        addd(t_cpu, 1);

                for(i=0;i<3;i++)
                        if(rkrun[i] == SACTIVE)
                                addd(t_rk[i], 1);

                for(i=0;i<32;i++)
                        if(pmrun[i] == SACTIVE)
                                addd(t_pm[i], 1);

                addd(t_real, 1);
                }
        t_loop++;

        if(tyflag1 == REFYES)
                ts_tty++;

        if((ps&UMODE) == UMODE) {
                u.u_utime++;
                if(u.u_prof[3])
                        incupc(pc, u.u_prof);
        } else
                u.u_stime++;
```

```
        pp = u.u_procp;
        if(++pp->p_cpu == 0)
                pp->p_cpu--;
        if(++lbolt >= HZ) {
                if((ps&0340) != 0)
                        return;
                lbolt -= HZ;
                if(++time[1] == 0)
                        ++time[0];
                spl1();
                if(time[1]==tout[1] && time[0]==tout[0])
                        wakeup(tout);
                if((time[1]&03) == 0) {
                        runrun++;
                        wakeup(&lbolt);
                }
                for(pp = &proc[0]; pp < &proc[NPROC]; pp++)
                if (pp->p_stat) {
                        if(pp->p_time != 127)
                                pp->p_time++;
                        if((pp->p_cpu & 0377) > SCHMAG)
                                pp->p_cpu -= SCHMAG; else
                                pp->p_cpu = 0;
                        if(pp->p_pri > PUSER)
                                setpri(pp);
                }
                if(runin!=0) {
                        runin = 0;
                        wakeup(&runin);
                }
                if((ps&UMODE) == UMODE) {
                        u.u_ar0 = &r0;
                        if(issig())
                                psig();
                        setpri(u.u_procp);
                }
        }
}

/*
 * timeout is called to arrange that
 * fun(arg) is called in tim/HZ seconds.
 * An entry is sorted into the callout
 * structure. The time in each structure
 * entry is the number of HZ's more
 * than the previous entry.
 * In this way, decrementing the
 * first entry has the effect of
 * updating all entries.
 */
timeout(fun, arg, tim)
{
        register struct callo *p1, *p2;
        register t;
        int s;

        t = tim;
        s = PS->integ;
        p1 = &callout[0];
        spl7();
```

```
        while(p1->c_func != 0 && p1->c_time <= t) {
                t =- p1->c_time;
                p1++;
        }
        p1->c_time =- t;
        p2 = p1;
        while(p2->c_func != 0)
                p2++;
        while(p2 >= p1) {
                (p2+1)->c_time = p2->c_time;
                (p2+1)->c_func = p2->c_func;
                (p2+1)->c_arg = p2->c_arg;
                p2--;
        }
        p1->c_time = t;
        p1->c_func = fun;
        p1->c_arg = arg;
        PS->integ = s;
}
```

p_sys1.c

```
#
/*
 */

#include "../param.h"
#include "../systm.h"
#include "../user.h"
#include "../proc.h"
#include "../buf.h"
#include "../reg.h"
#include "../inode.h"
#include "../perfo.h"

/*
 * exec system call.
 * Because of the fact that an I/O buffer is used
 * to store the caller's arguments during exec,
 * and more buffers are needed to read in the text file,
 * deadly embraces waiting for free buffers are possible.
 * Therefore the number of processes simultaneously
 * running in exec has to be limited to NEXEC.
 */
#define EXPRI   -1

exec()
{
        int ap, na, nc, *bp;
        int ts, ds, sep;
        register c, *ip;
        register char *cp;
        extern uchar;

        /*
         * pick up file names
         * and check various modes
         * for execute permission
         */

        ip = namei(&uchar, 0);
        if(ip == NULL)
                return;
        while(execnt >= NEXEC)
                sleep(&execnt, EXPRI);
        execnt++;
        bp = getblk(NODEV);
        if(access(ip, IEXEC) || (ip->i_mode&IFMT)!=0)
                goto bad;

        /*
         * pack up arguments into
         * allocated disk buffer
         */

        cp = bp->b_addr;
        na = 0;
        nc = 0;
        while(ap = fuword(u.u_arg[1])) {
                na++;
                if(ap == -1)
                        goto bad;
```

```
                u.u_arg[1] =+ 2;
                for(;;) {
                        c = fubyte(ap++);
                        if(c == -1)
                                goto bad;
                        *cp++ = c;
                        nc++;
                        if(nc > 510) {
                                u.u_error = E2BIG;
                                goto bad;
                        }
                        if(c == 0)
                                break;
                }
        }
        if((nc&1) != 0) {
                *cp++ = 0;
                nc++;
        }

        /*
         * read in first 8 bytes
         * of file for segment
         * sizes:
         * w0 = 407/410/411 (410 implies RO text) (411 implies sep ID)
         * w1 = text size
         * w2 = data size
         * w3 = bss size
         */

        u.u_base = &u.u_arg[0];
        u.u_count = 8;
        u.u_offset[1] = 0;
        u.u_offset[0] = 0;
        u.u_segflg = 1;
        readi(ip);
        u.u_segflg = 0;
        if(u.u_error)
                goto bad;
        sep = 0;
        if(u.u_arg[0] == 0407) {
                u.u_arg[2] =+ u.u_arg[1];
                u.u_arg[1] = 0;
        } else
        if(u.u_arg[0] == 0411)
                sep++; else
        if(u.u_arg[0] != 0410) {
                u.u_error = ENOEXEC;
                goto bad;
        }
        if(u.u_arg[1]!=0 && (ip->i_flag&ITEXT)==0 && ip->i_count!=1) {
                u.u_error = ETXTBSY;
                goto bad;
        }

        /*
         * find text and data sizes
         * try them out for possible
         * exceed of max sizes
         */
```

```
	ts = ((u.u_arg[1]+63)>>6) & 01777;
	ds = ((u.u_arg[2]+u.u_arg[3]+63)>>6) & 01777;
	if(estabur(ts, ds, SSIZE, sep))
		goto bad;

	/*
	 * allocate and clear core
	 * at this point, committed
	 * to the new image
	 */

	u.u_prof[3] = 0;
	xfree();
	expand(USIZE);
	xalloc(ip);
	c = USIZE+ds+SSIZE;
	expand(c);
	while(--c >= USIZE)
		clearseg(u.u_procp->p_addr+c);

	/*
	 * read in data segment
	 */

	estabur(0, ds, 0, 0);
	u.u_base = 0;
	u.u_offset[1] = 020+u.u_arg[1];
	u.u_count = u.u_arg[2];
	readi(ip);

	/*
	 * initialize stack segment
	 */

	u.u_tsize = ts;
	u.u_dsize = ds;
	u.u_ssize = SSIZE;
	u.u_sep = sep;
	estabur(u.u_tsize, u.u_dsize, u.u_ssize, u.u_sep);
	cp = bp->b_addr;
	ap = -nc - na*2 - 4;
	u.u_ar0[R6] = ap;
	suword(ap, na);
	c = -nc;
	while(na--) {
		suword(ap=+2, c);
		do
			subyte(c++, *cp);
		while(*cp++);
	}
	suword(ap+2, -1);

	/*
	 * set SUID/SGID protections, if no tracing
	 */

	if ((u.u_procp->p_flag&STRC)==0) {
		if(ip->i_mode&ISUID)
			if(u.u_uid != 0) {
```

```
                                u.u_uid = ip->i_uid;
                                u.u_procp->p_uid = ip->i_uid;
                        }
                if(ip->i_mode&ISGID)
                        u.u_gid = ip->i_gid;
        }

        /*
         * clear sigs, regs and return
         */

        c = ip;
        for(ip = &u.u_signal[0]; ip < &u.u_signal[NSIG]; ip++)
                if((*ip & 1) == 0)
                        *ip = 0;
        for(cp = &regloc[0]; cp < &regloc[6];)
                u.u_ar0[*cp++] = 0;
        u.u_ar0[R7] = 0;
        for(ip = &u.u_fsav[0]; ip < &u.u_fsav[25];)
                *ip++ = 0;
        ip = c;

bad:
        iput(ip);
        brelse(bp);
        if(execnt >= NEXEC)
                wakeup(&execnt);
        execnt--;
}

/*
 * exit system call:
 * pass back caller's r0
 */
rexit()
{

        u.u_arg[0] = u.u_ar0[R0] << 8;
        exit();
}

/*
 * Release resources.
 * Save u. area for parent to look at.
 * Enter zombie state.
 * Wake up parent and init processes,
 * and dispose of children.
 */
exit()
{
        register int *q, a;
        register struct proc *p;

        u.u_procp->p_flag =& ~STRC;
        p = u.u_procp;
        if((p->p_sig != 0)  &&  ((q=p->p_ttyp) != 0))
                /* exit on a signal => flush the tty queues */
                flushtty(q);
        for(q = &u.u_signal[0]; q < &u.u_signal[NSIG];)
                *q++ = 1;
```

```
        for(q = &u.u_ofile[0]; q < &u.u_ofile[NOFILE]; q++)
                if(a = *q) {
                        *q = NULL;
                        closef(a);
                }
        iput(u.u_cdir);
        xfree();
        a = malloc(swapmap, 1);
        if(a == NULL)
                panic("out of swap");
        p = getblk(swapdev, a);
        bcopy(&u, p->b_addr, 256);
        bwrite(p);
        q = u.u_procp;
        mfree(coremap, q->p_size, q->p_addr);
        q->p_addr = a;
        q->p_stat = SZOMB;

        /* system measures */

        addd(out_rt,1);

loop:
        for(p = &proc[0]; p < &proc[NPROC]; p++)
        if(q->p_ppid == p->p_pid) {
                wakeup(&proc[1]);
                wakeup(p);
                for(p = &proc[0]; p < &proc[NPROC]; p++)
                if(q->p_pid == p->p_ppid) {
                        p->p_ppid  = 1;
                        if (p->p_stat == SSTOP)
                                setrun(p);
                }
                swtch();
                /* no return */
        }
        q->p_ppid = 1;
        goto loop;
}

/*
 * Wait system call.
 * Search for a terminated (zombie) child,
 * finally lay it to rest, and collect its status.
 * Look also for stopped (traced) children,
 * and pass back status from them.
 */
wait()
{
        register f, *bp;
        register struct proc *p;

        f = 0;

loop:
        for(p = &proc[0]; p < &proc[NPROC]; p++)
        if(p->p_ppid == u.u_procp->p_pid) {
                f++;
                if(p->p_stat == SZOMB) {
                        u.u_ar0[R0] = p->p_pid;
```

```
                        bp = bread(swapdev, f=p->p_addr);
                        mfree(swapmap, 1, f);
                        p->p_stat = NULL;
                        p->p_pid = 0;
                        p->p_ppid = 0;
                        p->p_sig = 0;
                        p->p_ttyp = 0;
                        p->p_flag = 0;
                        p = bp->b_addr;
                        u.u_cstime[0] =+ p->u_cstime[0];
                        dpadd(u.u_cstime, p->u_cstime[1]);
                        dpadd(u.u_cstime, p->u_stime);
                        u.u_cutime[0] =+ p->u_cutime[0];
                        dpadd(u.u_cutime, p->u_cutime[1]);
                        dpadd(u.u_cutime, p->u_utime);
                        u.u_ar0[R1] = p->u_arg[0];
                        brelse(bp);
                        return;
                }
                if(p->p_stat == SSTOP) {
                        if((p->p_flag&SWTED) == 0) {
                                p->p_flag =| SWTED;
                                u.u_ar0[R0] = p->p_pid;
                                u.u_ar0[R1] = (p->p_sig<<8) | 0177;
                                return;
                        }
                        p->p_flag =& ~(STRC|SWTED);
                        setrun(p);
                }
        }
        if(f) {
                sleep(u.u_procp, PWAIT);
                goto loop;
        }
        u.u_error = ECHILD;
}

/*
 * fork system call.
 */
fork()
{
        register struct proc *p1, *p2;

        p1 = u.u_procp;
        for(p2 = &proc[0]; p2 < &proc[NPROC]; p2++)
                if(p2->p_stat == NULL)
                        goto found;
        u.u_error = EAGAIN;
        goto out;

found:
        if(newproc()) {
                u.u_ar0[R0] = p1->p_pid;
                u.u_cstime[0] = 0;
                u.u_cstime[1] = 0;
                u.u_stime = 0;
                u.u_cutime[0] = 0;
                u.u_cutime[1] = 0;
                u.u_utime = 0;
```

```
                return;
        }
        u.u_ar0[R0] = p2->p_pid;

out:
        u.u_ar0[R7] =+ 2;
}

/*
 * break system call.
 *  -- bad planning: "break" is a dirty word in C.
 */
sbreak()
{
        register a, n, d;
        int i;

        /*
         * set n to new data size
         * set d to new-old
         * set n to new total size
         */

        n = (((u.u_arg[0]+63)>>6) & 01777);
        if(!u.u_sep)
                n =- nseg(u.u_tsize) * 128;
        if(n < 0)
                n = 0;
        d = n - u.u_dsize;
        n =+ USIZE+u.u_ssize;
        if(estabur(u.u_tsize, u.u_dsize+d, u.u_ssize, u.u_sep))
                return;
        u.u_dsize =+ d;
        if(d > 0)
                goto bigger;
        a = u.u_procp->p_addr + n - u.u_ssize;
        i = n;
        n = u.u_ssize;
        while(n--) {
                copyseg(a-d, a);
                a++;
        }
        expand(i);
        return;

bigger:
        expand(n);
        a = u.u_procp->p_addr + n;
        n = u.u_ssize;
        while(n--) {
                a--;
                copyseg(a-d, a);
        }
        while(d--)
                clearseg(--a);
}

        /* system measures */

/*
```

p_sys1.c

```
/*
 * pinit system call.
 * Initialization of counters needed for system performance measurement
 */


pinit()
{
        register int i;

        t_cpu[0] = 0;
        t_cpu[1] = 0;
        cpurt[0] = 0;
        cpurt[1] = 0;

        for(i=0; i<3; i++)
                {
                t_rk[i][0] = 0;
                t_rk[i][1] = 0;
                n_rkrun[i][0] = 0;
                n_rkrun[i][1] = 0;
                rkrout[i][0] = 0;
                rkrout[i][1] = 0;
                }

        for(i=0; i<32; i++)
                {
                t_pm[i][0] = 0;
                t_pm[i][1] = 0;
                n_pmrun[i][0] = 0;
                n_pmrun[i][1] = 0;
                pmrout[i][0] = 0;
                pmrout[i][1] = 0;
                }

        t_real[0] = 0;
        t_real[1] = 0;
        out_rt[0] = 0;
        out_rt[1] = 0;

}
```

p_slp.c

```
#
/*
 */

#include "../param.h"
#include "../user.h"
#include "../proc.h"
#include "../text.h"
#include "../systm.h"
#include "../file.h"
#include "../inode.h"
#include "../buf.h"
#include "../perfo.h"

/*
 * Give up the processor till a wakeup occurs
 * on chan, at which time the process
 * enters the scheduling queue at priority pri.
 * The most important effect of pri is that when
 * pri<0 a signal cannot disturb the sleep;
 * if pri>=0 signals will be processed.
 * Callers of this routine must be prepared for
 * premature return, and check that the reason for
 * sleeping has gone away.
 */
sleep(chan, pri)
{
        register *rp, s;

        s = PS->integ;
        rp = u.u_procp;
        if(pri >= 0) {
                if(issig())
                        goto psig;
                spl6();
                rp->p_wchan = chan;
                rp->p_stat = SWAIT;
                rp->p_pri = pri;
                spl0();
                if(runin != 0) {
                        runin = 0;
                        wakeup(&runin);
                }
                swtch();
                if(issig())
                        goto psig;
        } else {
                spl6();
                rp->p_wchan = chan;
                rp->p_stat = SSLEEP;
                rp->p_pri = pri;
                spl0();
                swtch();
        }
        PS->integ = s;
        return;

        /*
         * If priority was low (>=0) and
         * there has been a signal,
```

```
	 * execute non-local goto to
	 * the qsav location.
	 * (see trap1/trap.c)
	 */
psig:
	aretu(u.u_qsav);
}

/*
 * Wake up all processes sleeping on chan.
 */
wakeup(chan)
{
	register struct proc *p;
	register c, i;

	c = chan;
	p = &proc[0];
	i = NPROC;
	do {
		if(p->p_wchan == c) {
			setrun(p);
		}
		p++;
	} while(--i);
}

/*
 * Set the process running;
 * arrange for it to be swapped in if necessary.
 */
setrun(p)
{
	register struct proc *rp;

	rp = p;
	rp->p_wchan = 0;
	rp->p_stat = SRUN;
	if(rp->p_pri < curpri)
		runrun++;
	if(runout != 0 && (rp->p_flag&SLOAD) == 0) {
		runout = 0;
		wakeup(&runout);
	}
}

/*
 * Set user priority.
 * The rescheduling flag (runrun)
 * is set if the priority is higher
 * than the currently running process.
 */
setpri(up)
{
	register *pp, p;

	pp = up;
	p = (pp->p_cpu & 0377)/16;
	p =+ PUSER + pp->p_nice;
	if(p > 127)
```

```
                p = 127;
        if(p < curpri)
                runrun++;
        pp->p_pri = p;
}

/*
 * The main loop of the scheduling (swapping)
 * process.
 * The basic idea is:
 *  see if anyone wants to be swapped in;
 *  swap out processes until there is room;
 *  swap him in;
 *  repeat.
 * Although it is not remarkably evident, the basic
 * synchronization here is on the runin flag, which is
 * slept on and is set once per second by the clock routine.
 * Core shuffling therefore takes place once per second.
 *
 * panic: swap error -- IO error while swapping.
 *      this is the one panic that should be
 *      handled in a less drastic way. Its
 *      very hard.
 */
sched()
{
        struct proc *p1;
        register struct proc *rp;
        register a, n;

        /*
         * find user to swap in
         * of users ready, select one out longest
         */

        goto loop;

sloop:
        runin++;
        sleep(&runin, PSWP);

loop:
        spl6();
        n = -1;
        for(rp = &proc[0]; rp < &proc[NPROC]; rp++)
        if(rp->p_stat==SRUN && (rp->p_flag&SLOAD)==0 &&
            rp->p_time > n) {
                p1 = rp;
                n = rp->p_time;
        }
        if(n == -1) {
                runout++;
                sleep(&runout, PSWP);
                goto loop;
        }

        /*
         * see if there is core for that process
         */
```

```
        spl0();
        rp = p1;
        a = rp->p_size;
        if((rp=rp->p_textp) != NULL)
                if(rp->x_ccount == 0)
                        a =+ rp->x_size;
        if((a=malloc(coremap, a)) != NULL)
                goto found2;

        /*
         * none found,
         * look around for easy core
         */

        spl6();
        for(rp = &proc[0]; rp < &proc[NPROC]; rp++)
        if((rp->p_flag&(SSYS|SLOCK|SLOAD))==SLOAD &&
            (rp->p_stat == SWAIT || rp->p_stat==SSTOP))
                goto found1;

        /*
         * no easy core,
         * if this process is deserving,
         * look around for
         * oldest process in core
         */

        if(n < 3)
                goto sloop;
        n = -1;
        for(rp = &proc[0]; rp < &proc[NPROC]; rp++)
        if((rp->p_flag&(SSYS|SLOCK|SLOAD))==SLOAD &&
           (rp->p_stat==SRUN || rp->p_stat==SSLEEP) &&
            rp->p_time > n) {
                p1 = rp;
                n = rp->p_time;
        }
        if(n < 2)
                goto sloop;
        rp = p1;

        /*
         * swap user out
         */

found1:
        spl0();
        rp->p_flag =& ~SLOAD;
        xswap(rp, 1, 0);
        goto loop;

        /*
         * swap user in
         */

found2:
        if((rp=p1->p_textp) != NULL) {
                if(rp->x_ccount == 0) {
                        if(swap(rp->x_daddr, a, rp->x_size, B_READ))
                                goto swaper;
```

```
                        rp->x_caddr = a;
                        a =+ rp->x_size;
                }
                rp->x_ccount++;
        }
        rp = p1;
        if(swap(rp->p_addr, a, rp->p_size, B_READ))
                goto swaper;
        mfree(swapmap, (rp->p_size+7)/8, rp->p_addr);
        rp->p_addr = a;
        rp->p_flag =| SLOAD;
        rp->p_time = 0;
        goto loop;

swaper:
        panic("swap error");
}

/*
 * This routine is called to reschedule the CPU.
 * if the calling process is not in RUN state,
 * arrangements for it to restart must have
 * been made elsewhere, usually by calling via sleep.
 */
swtch()
{
        static struct proc *p;
        register i, n;
        register struct proc *rp;

        if(p == NULL)
                p = &proc[0];
        /*
         * Remember stack of caller
         */
        savu(u.u_rsav);
        /*
         * Switch to scheduler's stack
         */
        retu(proc[0].p_addr);

loop:
        runrun = 0;
        rp = p;
        p = NULL;
        n = 128;
        /*
         * Search for highest-priority runnable process
         */
        i = NPROC;
        do {
                rp++;
                if(rp >= &proc[NPROC])
                        rp = &proc[0];
                if(rp->p_stat==SRUN && (rp->p_flag&SLOAD)!=0) {
                        if(rp->p_pri < n) {
                                p = rp;
                                n = rp->p_pri;
                        }
                }
```

```
        } while(--i);
        /*
         * If no process is runnable, idle.
         */
        if(p == NULL) {
                p = rp;
                cpurun = SIDLE;                 /* system measures */
                idle();
                cpurun = SACTIVE;               /* system measures */
                goto loop;
        }
        rp = p;
        curpri = n;
        /*
         * Switch to stack of the new process and set up
         * his segmentation registers.
         */

        retu(rp->p_addr);
        sureg();
        /*
         * If the new process paused because it was
         * swapped out, set the stack level to the last call
         * to savu(u_ssav).  This means that the return
         * which is executed immediately after the call to aretu
         * actually returns from the last routine which did
         * the savu.
         *
         * You are not expected to understand this.
         */
        if(rp->p_flag&SSWAP) {
                rp->p_flag =& ~SSWAP;
                aretu(u.u_ssav);
        }
        /*
         * The value returned here has many subtle implications.
         * See the newproc comments.
         */

        addd(cpurt, 1);                 /* system measures */

        return(1);
}

/*
 * Create a new process-- the internal version of
 * sys fork.
 * It returns 1 in the new process.
 * How this happens is rather hard to understand.
 * The essential fact is that the new process is created
 * in such a way that appears to have started executing
 * in the same call to newproc as the parent;
 * but in fact the code that runs is that of swtch.
 * The subtle implication of the returned value of swtch
 * (see above) is that this is the value that newproc's
 * caller in the new process sees.
 */
newproc()
{
        int a1, a2;
```

```
        struct proc *p, *up;
        register struct proc *rpp;
        register *rip, n;

        p = NULL;
        /*
         * First, just locate a slot for a process
         * and copy the useful info from this process into it.
         * The panic "cannot happen" because fork has already
         * checked for the existence of a slot.
         */
retry:
        mpid++;
        if(mpid < 0) {
                mpid = 0;
                goto retry;
        }
        for(rpp = &proc[0]; rpp < &proc[NPROC]; rpp++) {
                if(rpp->p_stat == NULL && p==NULL)
                        p = rpp;
                if (rpp->p_pid==mpid)
                        goto retry;
        }
        if ((rpp = p)==NULL)
                panic("no procs");

        /*
         * make proc entry for new proc
         */

        rip = u.u_procp;
        up = rip;

        rpp->p_stat = SRUN;
        rpp->p_flag = SLOAD;
        rpp->p_uid = rip->p_uid;
        rpp->p_ttyp = rip->p_ttyp;
        rpp->p_nice = rip->p_nice;
        rpp->p_textp = rip->p_textp;
        rpp->p_pid = mpid;
        rpp->p_ppid = rip->p_pid;
        rpp->p_time = 0;

        /*
         * make duplicate entries
         * where needed
         */

        for(rip = &u.u_ofile[0]; rip < &u.u_ofile[NOFILE];)
                if((rpp = *rip++) != NULL)
                        rpp->f_count++;
        if((rpp=up->p_textp) != NULL) {
                rpp->x_count++;
                rpp->x_ccount++;
        }
        u.u_cdir->i_count++;
        /*
         * Partially simulate the environment
         * of the new process so that when it is actually
         * created (by copying) it will look right.
```

```
         */
        savu(u.u_rsav);
        rpp = p;
        u.u_procp = rpp;
        rip = up;
        n = rip->p_size;
        a1 = rip->p_addr;
        rpp->p_size = n;
        a2 = malloc(coremap, n);
        /*
         * If there is not enough core for the
         * new process, swap out the current process to generate the
         * copy.
         */
        if(a2 == NULL) {
                rip->p_stat = SIDL;
                rpp->p_addr = a1;
                savu(u.u_ssav);
                xswap(rpp, 0, 0);
                rpp->p_flag =| SSWAP;
                rip->p_stat = SRUN;
        } else {
        /*
         * There is core, so just copy.
         */
                rpp->p_addr = a2;
                while(n--)
                        copyseg(a1++, a2++);
        }
        u.u_procp = rip;
        return(0);
}

/*
 * Change the size of the data+stack regions of the process.
 * If the size is shrinking, it's easy-- just release the extra core.
 * If it's growing, and there is core, just allocate it
 * and copy the image, taking care to reset registers to account
 * for the fact that the system's stack has moved.
 * If there is no core, arrange for the process to be swapped
 * out after adjusting the size requirement-- when it comes
 * in, enough core will be allocated.
 * Because of the ssave and SSWAP flags, control will
 * resume after the swap in swtch, which executes the return
 * from this stack level.
 *
 * After the expansion, the caller will take care of copying
 * the user's stack towards or away from the data area.
 */
expand(newsize)
{
        int i, n;
        register *p, a1, a2;

        p = u.u_procp;
        n = p->p_size;
        p->p_size = newsize;
        a1 = p->p_addr;
        if(n >= newsize) {
                mfree(coremap, n-newsize, a1+newsize);
```

p_slp.c

```
                return;
        }
        savu(u.u_rsav);
        a2 = malloc(coremap, newsize);
        if(a2 == NULL) {
                savu(u.u_ssav);
                xswap(p, 1, n);
                p->p_flag =| SSWAP;
                swtch();
                /* no return */
        }
        p->p_addr = a2;
        for(i=0; i<n; i++)
                copyseg(a1+i, a2++);
        mfree(coremap, n, a1);

        retu(p->p_addr);
        sureg();
}
```

p_alloc.c

```
#
/*
 */

#include "../param.h"
#include "../systm.h"
#include "../filsys.h"
#include "../conf.h"
#include "../buf.h"
#include "../inode.h"
#include "../user.h"
#include "../perfo.h"

/*
 * iinit is called once (from main)
 * very early in initialization.
 * It reads the root's super block
 * and initializes the current date
 * from the last modified date.
 *
 * panic: iinit -- cannot read the super
 * block. Usually because of an IO error.
 */
iinit()
{
        register *cp, *bp;
        int drname, drno;                       /* system measures */


        /* system measures */

        drname = rootdev.d_major;
        drno = rootdev.d_minor & 037;

        if(drname == MRK)
                addd(rkrout[drno], 1);
        else if(drname == MPM)
                addd(pmrout[drno], 1);

        (*bdevsw[rootdev.d_major].d_open)(rootdev, 1);
        bp = bread(rootdev, 1);
        cp = getblk(NODEV);
        if(u.u_error)
                panic("iinit");
        bcopy(bp->b_addr, cp->b_addr, 256);
        brelse(bp);
        mount[0].m_bufp = cp;
        mount[0].m_dev = rootdev;
        cp = cp->b_addr;
        cp->s_flock = 0;
        cp->s_ilock = 0;
        cp->s_ronly = 0;
        time[0] = cp->s_time[0];
        time[1] = cp->s_time[1];
}

/*
 * alloc will obtain the next available
 * free disk block from the free list of
 * the specified device.
```

p_alloc.c

```
 * The super block has up to 100 remembered
 * free blocks; the last of these is read to
 * obtain 100 more . . .
 *
 * no space on dev x/y -- when
 * the free list is exhausted.
 */
alloc(dev)
{
        int bno;
        register *bp, *ip, *fp;

        fp = getfs(dev);
        while(fp->s_flock)
                sleep(&fp->s_flock, PINOD);
        do {
                if(fp->s_nfree <= 0)
                        goto nospace;
                bno = fp->s_free[--fp->s_nfree];
                if(bno == 0)
                        goto nospace;
        } while (badblock(fp, bno, dev));
        if(fp->s_nfree <= 0) {
                fp->s_flock++;
                bp = bread(dev, bno);
                ip = bp->b_addr;
                fp->s_nfree = *ip++;
                bcopy(ip, fp->s_free, 100);
                brelse(bp);
                fp->s_flock = 0;
                wakeup(&fp->s_flock);
        }
        bp = getblk(dev, bno);
        clrbuf(bp);
        fp->s_fmod = 1;
        return(bp);

nospace:
        fp->s_nfree = 0;
        prdev("no space", dev);
        u.u_error = ENOSPC;
        return(NULL);
}

/*
 * place the specified disk block
 * back on the free list of the
 * specified device.
 */
free(dev, bno)
{
        register *fp, *bp, *ip;

        fp = getfs(dev);
        fp->s_fmod = 1;
        while(fp->s_flock)
                sleep(&fp->s_flock, PINOD);
        if (badblock(fp, bno, dev))
                return;
        if(fp->s_nfree <= 0) {
```

```
                fp->s_nfree = 1;
                fp->s_free[0] = 0;
        }
        if(fp->s_nfree >= 100) {
                fp->s_flock++;
                bp = getblk(dev, bno);
                ip = bp->b_addr;
                *ip++ = fp->s_nfree;
                bcopy(fp->s_free, ip, 100);
                fp->s_nfree = 0;
                bwrite(bp);
                fp->s_flock = 0;
                wakeup(&fp->s_flock);
        }
        fp->s_free[fp->s_nfree++] = bno;
        fp->s_fmod = 1;
}

/*
 * Check that a block number is in the
 * range between the I list and the size
 * of the device.
 * This is used mainly to check that a
 * garbage file system has not been mounted.
 *
 * bad block on dev x/y -- not in range
 */
badblock(afp, abn, dev)
{
        register struct filsys *fp;
        register char *bn;

        fp = afp;
        bn = abn;
        if (bn < fp->s_isize+2 || bn >= fp->s_fsize) {
                prdev("bad block", dev);
                return(1);
        }
        return(0);
}

/*
 * Allocate an unused I node
 * on the specified device.
 * Used with file creation.
 * The algorithm keeps up to
 * 100 spare I nodes in the
 * super block. When this runs out,
 * a linear search through the
 * I list is instituted to pick
 * up 100 more.
 */
ialloc(dev)
{
        register *fp, *bp, *ip;
        int i, j, k, ino;

        fp = getfs(dev);
        while(fp->s_ilock)
                sleep(&fp->s_ilock, PINOD);
```

p_alloc.c

```
loop:
        if(fp->s_ninode > 0) {
                ino = fp->s_inode[--fp->s_ninode];
                ip = iget(dev, ino);
                if (ip==NULL)
                        return(NULL);
                if(ip->i_mode == 0) {
                        for(bp = &ip->i_mode; bp < &ip->i_addr[8];)
                                *bp++ = 0;
                        fp->s_fmod = 1;
                        return(ip);
                }
                /*
                 * Inode was allocated after all.
                 * Look some more.
                 */
                iput(ip);
                goto loop;
        }
        fp->s_ilock++;
        ino = 0;
        for(i=0; i<fp->s_isize; i++) {
                bp = bread(dev, i+2);
                ip = bp->b_addr;
                for(j=0; j<256; j+=16) {
                        ino++;
                        if(ip[j] != 0)
                                continue;
                        for(k=0; k<NINODE; k++)
                        if(dev==inode[k].i_dev && ino==inode[k].i_number)
                                goto cont;
                        fp->s_inode[fp->s_ninode++] = ino;
                        if(fp->s_ninode >= 100)
                                break;
                cont:;
                }
                brelse(bp);
                if(fp->s_ninode >= 100)
                        break;
        }
        fp->s_ilock = 0;
        wakeup(&fp->s_ilock);
        if (fp->s_ninode > 0)
                goto loop;
        prdev("Out of inodes", dev);
        u.u_error = ENOSPC;
        return(NULL);
}

/*
 * Free the specified I node
 * on the specified device.
 * The algorithm stores up
 * to 100 I nodes in the super
 * block and throws away any more.
 */
ifree(dev, ino)
{
        register *fp;
```

```
        fp = getfs(dev);
        if(fp->s_ilock)
                return;
        if(fp->s_ninode >= 100)
                return;
        fp->s_inode[fp->s_ninode++] = ino;
        fp->s_fmod = 1;
}

/*
 * getfs maps a device number into
 * a pointer to the incore super
 * block.
 * The algorithm is a linear
 * search through the mount table.
 * A consistency check of the
 * in core free-block and i-node
 * counts.
 *
 * bad count on dev x/y -- the count
 *      check failed. At this point, all
 *      the counts are zeroed which will
 *      almost certainly lead to "no space"
 *      diagnostic
 * panic: no fs -- the device is not mounted.
 *      this "cannot happen"
 */
getfs(dev)
{
        register struct mount *p;
        register char *n1, *n2;

        for(p = &mount[0]; p < &mount[NMOUNT]; p++)
        if(p->m_bufp != NULL && p->m_dev == dev) {
                p = p->m_bufp->b_addr;
                n1 = p->s_nfree;
                n2 = p->s_ninode;
                if(n1 > 100 || n2 > 100) {
                        prdev("bad count", dev);
                        p->s_nfree = 0;
                        p->s_ninode = 0;
                }
                return(p);
        }
        panic("no fs");
}

/*
 * update is the internal name of
 * 'sync'. It goes through the disk
 * queues to initiate sandbagged IO;
 * goes through the I nodes to write
 * modified nodes; and it goes through
 * the mount table to initiate modified
 * super blocks.
 */
update()
{
        register struct inode *ip;
        register struct mount *mp;
```

p_alloc.c

```
        register *bp;

        if(updlock)
                return;
        updlock++;
        for(mp = &mount[0]; mp < &mount[NMOUNT]; mp++)
                if(mp->m_bufp != NULL) {
                        ip = mp->m_bufp->b_addr;
                        if(ip->s_fmod==0 || ip->s_ilock!=0 ||
                           ip->s_flock!=0 || ip->s_ronly!=0)
                                continue;
                        bp = getblk(mp->m_dev, 1);
                        ip->s_fmod = 0;
                        ip->s_time[0] = time[0];
                        ip->s_time[1] = time[1];
                        bcopy(ip, bp->b_addr, 256);
                        bwrite(bp);
                }
        for(ip = &inode[0]; ip < &inode[NINODE]; ip++)
                if((ip->i_flag&ILOCK) == 0) {
                        ip->i_flag =| ILOCK;
                        iupdat(ip, time);
                        prele(ip);
                }
        updlock = 0;
        bflush(NODEV);
}
```

p_fio.c

```
#
/*
 */

#include "../param.h"
#include "../user.h"
#include "../filsys.h"
#include "../file.h"
#include "../conf.h"
#include "../inode.h"
#include "../reg.h"
#include "../perfo.h"

/*
 * Convert a user supplied
 * file descriptor into a pointer
 * to a file structure.
 * Only task is to check range
 * of the descriptor.
 */
getf(f)
{
        register *fp, rf;

        rf = f;
        if(rf<0 || rf>=NOFILE)
                goto bad;
        fp = u.u_ofile[rf];
        if(fp != NULL)
                return(fp);
bad:
        u.u_error = EBADF;
        return(NULL);
}

/*
 * Internal form of close.
 * Decrement reference count on
 * file structure and call closei
 * on last closef.
 * Also make sure the pipe protocol
 * does not constipate.
 */
closef(fp)
int *fp;
{
        register *rfp, *ip;

        rfp = fp;
        if(rfp->f_flag&FPIPE) {
                ip = rfp->f_inode;
                ip->i_mode =& ~(IREAD|IWRITE);
                wakeup(ip+1);
                wakeup(ip+2);
        }
        if(rfp->f_count <= 1)
                closei(rfp->f_inode, rfp->f_flag&FWRITE);
        rfp->f_count--;
}
```

p_fio.c

```
/*
 * Decrement reference count on an
 * inode due to the removal of a
 * referencing file structure.
 * On the last closei, switchout
 * to the close entry point of special
 * device handler.
 * Note that the handler gets called
 * on every open and only on the last
 * close.
 */
closei(ip, rw)
int *ip;
{
        register *rip;
        register dev, maj;
        int drno;

        rip = ip;
        dev = rip->i_addr[0];
        maj = rip->i_addr[0].d_major;

        /* system measures */

        drno = rip->i_addr[0].d_minor & 037;

        if(rip->i_count <= 1)
        switch(rip->i_mode&IFMT) {

        case IFCHR:

                /* system measures */

                if(maj == MRK)
                        addd(rkrout[drno], 1);
                else if(maj == MPM)
                        addd(pmrout[drno], 1);

                (*cdevsw[maj].d_close)(dev, rw);
                break;

        case IFBLK:

                /* system measures */

                if(maj == MRK)
                        addd(rkrout[drno], 1);
                else if(maj == MPM)
                        addd(pmrout[drno], 1);

                (*bdevsw[maj].d_close)(dev, rw);
        }
        iput(rip);
}

/*
 * openi called to allow handler
 * of special files to initialize and
 * validate before actual IO.
 * Called on all sorts of opens
```

```
 * and also on mount.
 */
openi(ip, rw)
int *ip;
{
        register *rip;
        register dev, maj;
        int drno;                               /* system measures */

        rip = ip;
        dev = rip->i_addr[0];
        maj = rip->i_addr[0].d_major;

        /* system measures */

        drno = rip->i_addr[0].d_minor & 037;

        switch(rip->i_mode&IFMT) {

        case IFCHR:
                if(maj >= nchrdev)
                        goto bad;

                /* system measures */

                if(maj == MRK)
                        addd(rkrout[drno], 1);
                else if(maj == MPM)
                        addd(pmrout[drno], 1);

                (*cdevsw[maj].d_open)(dev, rw);
                break;

        case IFBLK:

                /* system measures */

                if(maj == MRK)
                        addd(rkrout[drno], 1);
                else if(maj == MPM)
                        addd(pmrout[drno], 1);

                if(maj >= nblkdev)
                        goto bad;
                (*bdevsw[maj].d_open)(dev, rw);
        }
        return;

bad:
        u.u_error = ENXIO;
}

/*
 * Check mode permission on inode pointer.
 * Mode is READ, WRITE or EXEC.
 * In the case of WRITE, the
 * read-only status of the file
 * system is checked.
 * Also in WRITE, prototype text
 * segments cannot be written.
```

```
 * The mode is shifted to select
 * the owner/group/other fields.
 * The super user is granted all
 * permissions except for EXEC where
 * at least one of the EXEC bits must
 * be on.
 */
access(aip, mode)
int *aip;
{
        register *ip, m;

        ip = aip;
        m = mode;
        if(m == IWRITE) {
                if(getfs(ip->i_dev)->s_ronly != 0) {
                        u.u_error = EROFS;
                        return(1);
                }
                if(ip->i_flag & ITEXT) {
                        u.u_error = ETXTBSY;
                        return(1);
                }
        }
        if(u.u_uid == 0) {
                if(m == IEXEC && (ip->i_mode &
                        (IEXEC | (IEXEC>>3) | (IEXEC>>6))) == 0)
                                goto bad;
                return(0);
        }
        if(u.u_uid != ip->i_uid) {
                m =>> 3;
                if(u.u_gid != ip->i_gid)
                        m =>> 3;
        }
        if((ip->i_mode&m) != 0)
                return(0);

bad:
        u.u_error = EACCES;
        return(1);
}

/*
 * Look up a pathname and test if
 * the resultant inode is owned by the
 * current user.
 * If not, try for super-user.
 * If permission is granted,
 * return inode pointer.
 */
owner()
{
        register struct inode *ip;
        extern uchar();

        if ((ip = namei(uchar, 0)) == NULL)
                return(NULL);
        if(u.u_uid == ip->i_uid)
                return(ip);
```

```
        if (suser())
                return(ip);
        iput(ip);
        return(NULL);
}

/*
 * Test if the current user is the
 * super user.
 */
suser()
{

        if(u.u_uid == 0)
                return(1);
        u.u_error = EPERM;
        return(0);
}

/*
 * Allocate a user file descriptor.
 */
ufalloc()
{
        register i;

        for (i=0; i<NOFILE; i++)
                if (u.u_ofile[i] == NULL) {
                        u.u_ar0[R0] = i;
                        return(i);
                }
        u.u_error = EMFILE;
        return(-1);
}

/*
 * Allocate a user file descriptor
 * and a file structure.
 * Initialize the descriptor
 * to point at the file structure.
 *
 * no file -- if there are no available
 *      file structures.
 */
falloc()
{
        register struct file *fp;
        register i;

        if ((i = ufalloc()) < 0)
                return(NULL);
        for (fp = &file[0]; fp < &file[NFILE]; fp++)
                if (fp->f_count==0) {
                        u.u_ofile[i] = fp;
                        fp->f_count++;
                        fp->f_offset[0] = 0;
                        fp->f_offset[1] = 0;
                        return(fp);
                }
        printf("no file\n");
```

p_fio.c

```
        u.u_error = ENFILE;
        return(NULL);
}
```

p_main.c

```
#
#include "../param.h"
#include "../user.h"
#include "../systm.h"
#include "../proc.h"
#include "../text.h"
#include "../inode.h"
#include "../seg.h"
#include "../perfo.h"

#define CLOCK1  0177546
#define CLOCK2  0172540
/*
 * Icode is the octal bootstrap
 * program executed in user mode
 * to bring up the system.
 */
int     icode[]
{
        0104413,        /* sys exec; init; initp */
        0000014,
        0000010,
        0000777,        /* br . */
        0000014,        /* initp: init; 0 */
        0000000,
        0062457,        /* init: </etc/init\0> */
        0061564,
        0064457,
        0064556,
        0000164,
};

/*
 * Initialization code.
 * Called from m40.s or m45.s as
 * soon as a stack and segmentation
 * have been established.
 * Functions:
 *      clear and free user core
 *      find which clock is configured
 *      hand craft 0th process
 *      call all initialization routines
 *      fork - process 0 to schedule
 *           - process 1 execute bootstrap
 *
 * panic: no clock -- neither clock responds
 * loop at loc 6 in user mode -- /etc/init
 *      cannot be executed.
 */
main()
{
        extern schar;
        register i, *p;

        /*
         * zero and free all of core
         */

        updlock = 0;
        i = *ka6 + USIZE;
```

```
	UISD->r[0] = 077406;
	for(;;) {
		UISA->r[0] = i;
		if(fuibyte(0) < 0)
			break;
		clearseg(i);
		maxmem++;
		mfree(coremap, 1, i);
		i++;
	}
	if(cputype == 70)
	for(i=0; i<62; i=+2) {
		UBMAP->r[i] = i<<12;
		UBMAP->r[i+1] = 0;
	}
	printf("unix tss --- measurement system --- V6.3 --- mem= %l\n", maxmem*
	maxmem = min(maxmem, MAXMEM);
	mfree(swapmap, nswap, swplo);

	/*
	 * determine clock
	 */

	UISA->r[7] = ka6[1]; /* io segment */
	UISD->r[7] = 077406;
	lks = CLOCK1;
	if(fuiword(lks) == -1) {
		lks = CLOCK2;
		if(fuiword(lks) == -1)
			panic("no clock");
	}

	/*
	 * set up system process
	 */

	proc[0].p_addr = *ka6;
	proc[0].p_size = USIZE;
	proc[0].p_stat = SRUN;
	proc[0].p_flag =| SLOAD|SSYS;
	u.u_procp = &proc[0];

	/* system measures */

	cpurun = SACTIVE;

	/*
	 * set up 'known' i-nodes
	 */

	*lks = 0115;
	cinit();
	binit();
	iinit();
	rootdir = iget(rootdev, ROOTINO);
	rootdir->i_flag =& ~ILOCK;
	u.u_cdir = iget(rootdev, ROOTINO);
	u.u_cdir->i_flag =& ~ILOCK;

	/*
```

```
		 * make init process
		 * enter scheduling loop
		 * with system process
		 */

		if(newproc()) {
			expand(USIZE+1);
			estabur(0, 1, 0, 0);
			copyout(icode, 0, sizeof icode);
			/*
			 * Return goes to loc. 0 of user init
			 * code just copied out.
			 */
			return;
		}
		sched();
}

/*
 * Load the user hardware segmentation
 * registers from the software prototype.
 * The software registers must have
 * been setup prior by estabur.
 */
sureg()
{
		register *up, *rp, a;

		a = u.u_procp->p_addr;
		up = &u.u_uisa[16];
		rp = &UISA->r[16];
		if(cputype == 40) {
			up -= 8;
			rp -= 8;
		}
		while(rp > &UISA->r[0])
			*--rp = *--up + a;
		if((up=u.u_procp->p_textp) != NULL)
			a -= up->x_caddr;
		up = &u.u_uisd[16];
		rp = &UISD->r[16];
		if(cputype == 40) {
			up -= 8;
			rp -= 8;
		}
		while(rp > &UISD->r[0]) {
			*--rp = *--up;
			if((*rp & WO) == 0)
				rp[(UISA-UISD)/2] -= a;
		}
}

/*
 * Set up software prototype segmentation
 * registers to implement the 3 pseudo
 * text,data,stack segment sizes passed
 * as arguments.
 * The argument sep specifies if the
 * text and data+stack segments are to
 * be separated.
```

p_main.c

```
 */
estabur(nt, nd, ns, sep)
{
        register a, *ap, *dp;

        if(sep) {
                if(cputype == 40)
                        goto err;
                if(nseg(nt) > 8 || nseg(nd)+nseg(ns) > 8)
                        goto err;
        } else
                if(nseg(nt)+nseg(nd)+nseg(ns) > 8)
                        goto err;
        if(nt+nd+ns+USIZE > maxmem)
                goto err;
        a = 0;
        ap = &u.u_uisa[0];
        dp = &u.u_uisd[0];
        while(nt >= 128) {
                *dp++ = (127<<8) | RO;
                *ap++ = a;
                a =+ 128;
                nt =- 128;
        }
        if(nt) {
                *dp++ = ((nt-1)<<8) | RO;
                *ap++ = a;
        }
        if(sep)
        while(ap < &u.u_uisa[8]) {
                *ap++ = 0;
                *dp++ = 0;
        }
        a = USIZE;
        while(nd >= 128) {
                *dp++ = (127<<8) | RW;
                *ap++ = a;
                a =+ 128;
                nd =- 128;
        }
        if(nd) {
                *dp++ = ((nd-1)<<8) | RW;
                *ap++ = a;
                a =+ nd;
        }
        while(ap < &u.u_uisa[8]) {
                *dp++ = 0;
                *ap++ = 0;
        }
        if(sep)
        while(ap < &u.u_uisa[16]) {
                *dp++ = 0;
                *ap++ = 0;
        }
        a =+ ns;
        while(ns >= 128) {
                a =- 128;
                ns =- 128;
                *--dp = (127<<8) | RW;
                *--ap = a;
```

```
        }
        if(ns) {
                *--dp = ((128-ns)<<8) | RW | ED;
                *--ap = a-128;
        }
        if(!sep) {
                ap = &u.u_uisa[0];
                dp = &u.u_uisa[8];
                while(ap < &u.u_uisa[8])
                        *dp++ = *ap++;
                ap = &u.u_uisd[0];
                dp = &u.u_uisd[8];
                while(ap < &u.u_uisd[8])
                        *dp++ = *ap++;
        }
        sureg();
        return(0);

err:
        u.u_error = ENOMEM;
        return(-1);
}

/*
 * Return the arg/128 rounded up.
 */
nseg(n)
{

        return((n+127)>>7);
}
```

ANNEXE 3.

PROGRAMMES DE TESTS
ET
DE RESOLUTION DU MODELE.

p_accnt.c

```
#
        /*
         *      Program which collects :
         *                - the number of jobs in the system
         *                - the number of jobs in the CPU queue
         *                - reflexion time at ttye
         * every 5' and puts observations in work files.
         *
         * Author : Cornil Dirk.
         */

#define SMAX    256
#define NPROC   50
#define SET     1

        /* stat codes

        SSLEEP  1               sleeping on high priority
        SWAIT   2               sleeping on low priority
        SRUN    3               running
        SIDL    4               intermediate state in process creation
        SZOMB   5               intermediate state in process termination
        SSTOP   6               process being traced
        */


struct
{
        char    p_stat;
        char    p_flag;
        char    p_pri;          /* priority, negative is high */
        char    p_sig;          /* signal number sent to this process */
        char    p_uid;          /* user id, used to direct tty signals */
        char    p_time;         /* resident time for scheduling */
        char    p_cpu;          /* cpu usage for scheduling */
        char    p_nice;         /* nice for scheduling */
        int     p_ttyp;         /* controlling tty */
        int     p_pid;          /* unique process id */
        int     p_ppid;         /* process id of parent */
        int     p_addr;         /* address of swappable image */
        int     p_size;         /* size of swappable image (*64 bytes) */
        int     p_wchan;        /* event process is awaiting */
        int     *p_textp;       /* pointer to text structure */
} proc;


main()
{
        register i;
        int njobs, cpuj;
        int a_proc;
        int pid;
        int fd1, fd2, fd3, fd4, fd5, mem, r;

        struct
                {
                char text[8];
                int type;
                int value;
                } nl[1];
```

```
struct
        {
        int flag;
        int time[SMAX];
        } buf;

setup( &nl[0] , "_proc");

nlist("/perform",nl);

if(nl[0].type == 0)
        {
        printf("No namelist !!\n");
        exit();
        }

a_proc = nl[0].value;

if((mem = open("/dev/kmem", 0)) <= 0)
        {
        printf("Can't open kernel memory\n");
        exit();
        }

if((fd1 = creat("/mnt/mem/cornil/usr/adm/daylist",0666)) == -1)
        {
        printf("Can't open /mnt/mem/cornil/usr/adm/daylist\n");
        exit();
        }

if((fd2 = open("/mnt/mem/cornil/usr/adm/joblist",1)) == -1)
        {
        printf("Can't open /mnt/mem/cornil/usr/adm/daylist\n");
        exit();
        }

if((fd3 = open("/mnt/mem/cornil/usr/adm/ttyacc",1)) == -1)
        {
        printf("Can't open /mnt/mem/cornil/usr/adm/ttyacc\n");
        exit();
        }

if((fd4 = creat("/mnt/mem/cornil/usr/adm/fpid",0666)) == -1)
        {
        printf("Can't open /mnt/mem/cornil/usr/adm/fpid\n");
        exit();
        }

if((fd5 = creat("/mnt/mem/cornil/usr/adm/cpujbs",0666)) == -1)
        {
        printf("Can't create /mnt/mem/cornil/usr/adm/cjobs\n");
        exit();
        }

pid = getpid();

write(fd4,&pid,2);

seek(fd2,0,2);
```

```
        seek(fd3, 0, 2);

        for(;;)
                {
                seek(mem, a_proc, 0);
                njobs = 0;
                cpuj = 0;

                for(i=0; i<NPROC; i++)
                        {
                        if((r = read(mem, &proc, sizeof(proc))) == -1)
                                {
                                printf("Error while reading kernel memory\n");
                                exit();
                                }

                        if(proc.p_stat > 0 && proc.p_stat < 6)
                                njobs++;

                        if(proc.p_stat > 0 && proc.p_stat < 4)
                                {
                                if((proc.p_flag & 07) != 0)
                                        cpuj++;
                                }

                        }

                write(fd1, &njobs, 2);
                write(fd2, &njobs, 2);
                write(fd5, &cpuj, 2);

                svtime(&buf);

                if(buf.flag == SET)
                        write(fd3, &buf, sizeof(buf));

                sleep(300);
                }

        close(fd1);
        close(fd2);
        close(fd3);
        close(fd4);
        close(fd5);
        close(mem);
        exit();
}

setup(p, s)
char *p, *s;
{
        while(*p++ = *s++);
}
```

p_cmpmean.c

```
#

        /*
         *      Conformity test for 2 populations with normal distribution
         * and dependant variables.
         *
         * Input : fdiff in bit mode (mean of differences)
         *         fad in bit mode (sum of squares and product of deviates
         *                          for the differences)
         *
         * Output : F Snedecor to be compare with value in Biometrika tables.
         *
         * Author : Cornil Dirk.
         *
         */

#define NVAR 12
#define READ 0

main(argc,argv)
int argc;
char **argv;
{

        register i, j;
        int n, p;
        int fd1, fd2;
        double line[NVAR];
        double diff[NVAR] ,sum[NVAR];
        double ad[NVAR][NVAR];
        double num, res;

        if(argc != 3)
                {
                printf("p_cmpmean fdiff fad\n");
                exit();
                }

        printf("Number of observations for population ?\t");
        if((n = readint(0)) <= 0)
                {
                printf("Wrong value\n");
                exit();
                }

        printf("Number of variables ?\t");
        if((p = readint(0)) <= 0)
                {
                printf("Wrong value\n");
                exit();
                }

        fd1 = fop(argv[1],READ);
        fd2 = fop(argv[2],READ);

        read(fd1,&diff,96);

        i = 0;

        while(read(fd2,&line,96) > 0)
```

```
                {
                for(j=0; j<p; j++)
                        ad[i][j] = line[j];

                i++;
                }

        close(fd1);
        close(fd2);


        for(i=0; i<p; i++)
                for(j=0; j<p; j++)
                        sum[i] =+ (diff[j] * ad[i][j]);

        res = 0.0;

        for(i=0; i<p; i++)
                res =+ sum[i] * diff[i];

        num = (n * (n - p)) * res;
        printf("\n\n\tF Snedecor : \t%.8f\n",(num / p));
        printf("\twith parameters %d & %d\n",p,(n - p));

        exit();
}
```

p_kill.c

```
/*
 *    Kills batch process started with initialisation of
 * the measurement system.
 *
 * Author : Cornil Dirk
 */

main()
{
        int fd, pid;

        if((fd = open("/mnt/mem/cornil/usr/adm/fpid",0)) == -1)
                {
                printf("Can't open /mnt/mem/cornil/usr/adm/fpid\n");
                exit();
                }

        read(fd,&pid,2);
        close(fd);

        if(kill(pid,9) != 0)
                {
                printf("Can't kill p_cnt\n");
                exit();
                }

        printf("p_cnt killed\n");
        exit();
}
```

p_prodmat.c

```
#

        /*
         *      Computes product of matrices :
         *              - matrice 1 is written in lines
         *              - matrice 2 is written in columns
         *              - result is printed on tty
         *
         * Author : Cornil Dirk.
         *
         */

#define NVAR 12
#define READ 0

main(argc, argv)
int argc;
char **argv;
{
        register i, j, k;
        int fd1, fdc;
        double line[NVAR], col[NVAR];
        double result[NVAR];

        if(argc != 3)
                {
                printf("prodmat lmat cmat\n");
                exit();
                }

        fd1 = fop(argv[1], READ);
        fdc = fop(argv[2], READ);

        for(i=0; i<NVAR; i++)
                {
                read(fd1, &line, 96);
                for(j=0; j<NVAR; j++)
                        {
                        read(fdc, &col, 96);
                        result[j] = 0.0;

                        for(k=0; k<NVAR; k++)
                                result[j] =+ line[k] * col[k];

                        printf("%.8f\n", result[j]);
                        }
                seek(fdc, 0, 0);
                }

        close(fdc);
        close(fd1);

        exit();
}
```

```
#

#define MAX     12


        /*
         * Converts integer from bit mode to ASCII
         *
         * Output : ASCII integer
         *
         * Author : Adans Jean-Paul
         */

readint(file)
int file;
{
        char buf[5];
        register char *b,*c;
        register i;
        b = buf;
        for(i=0;i < 5;i++){
                if(read(file, c, 1) == -1){
                        printf("EOF\n");
                        exit();
                }
                if(*c == '\n'){
                        buf[i] = '\0';
                        return(atoi(b));
                }
                else buf[i] = *c;
        }
}



        /*
         * Converts floating numbers from bit mode to ASCII
         *
         * Output : floating ASCII
         *
         * Author : Adans Jean-Paul
         */

float readfloat(file)
int file;
{
        float atof();

        char buf[30];
        register char *b,*c;
        register i;

        b = buf;

        for(i=0;i < 30;i++){
                if(read(file, c, 1) == -1){
                        printf("EOF\n");
                        exit();
                }
                if(*c == '\n'){
```

```
                        buf[i] = '\0';
                        return(atof(b));
                }
                else buf[i] = *c;
        }
}

        /*
         *      Computes x ** y
         */

float expon(number,exp)
float number;
int exp;
{
        register int i;
        float prod;

        prod = 1.0;

        for(i=exp;i>0;i--)
                prod =* number;

        return(prod);
}

setup(p,s)
char *p, *s;
{
        while(*p++ = *s++);
}

        /*
         * Copies an array of 2 integers from kernel space into a
         * double variable in user space.
         *
         * Author : Cornil Dirk.
         */

double kulcp(fd,k_adress)
int fd;
int k_adress;
{
        register int r;
        int value[2];
        double result;
        double atof();

        seek(fd,k_adress,0);

        if((r = read(fd,&value,4)) == -1)
                {
                printf("Error while reading kernel memory\n");
                exit();
                }

        result = atof(locv(value[0],value[1]));
        return(result);
}
```

```
        /*
         *      Copies matrix on a file in order to save memory
         * This matrix is put on 2 files : one recorded in lines
         * and the other in colons.
         *      It has been done to make easier matrix manipulations
         *
         * Author : Cornil Dirk
         */

fsave(fdl, fdc, a)
int fdl, fdc;
double a[MAX][MAX];
{
        register i, j;
        double line[MAX];

        seek(fdc, 0, 0);

        for(i=0; i<MAX; i++)
                {
                for(j=0; j<MAX; j++)
                        line[j] = a[i][j];
                write(fdc, &line, 96 * MAX);
                }

        close(fdc);
        seek(fdl, 0, 0);

        for(i=0; i<MAX; i++)
                {
                for(j=0; j<MAX; j++)
                        line[j] = a[j][i];
                write(fdl, &line, 96 * MAX);
                }

        close(fdl);

        return;
}

        /*
         *      Opens file whose name is given with given flag ,
         * checks validity and returns file descriptor
         *
         * Author : Cornil Dirk.
         */


fop(name, flag)
char *name;
int flag;
{
        int fd;

        if((fd = open(name, flag)) == -1)
                {
                printf("Can't open %s\n", name);
                exit();
                }
```

```
        return(fd);
}


        /*
         *      Creates file whose name is given with given flag ,
         * checks validity and returns file descriptor
         *
         * Author : Cornil Dirk.
         */

fcre(name)
char *name;
{
        int fd;

        if((fd = creat(name,0666)) == -1)
                {
                printf("Can't create %s\n",name);
                exit();
                }

        return(fd);
}
```

p_station.c

```
#

        /*
         *      Checks stationnarity of cpu queue lenght
         *
         * Author : Cornil Dirk.
         *
         */

#define BMAX    20

main()
{
        register n, i, j;
        int fd;
        int njobs[BMAX];
        double tmp;
        double sum[BMAX];

        if((fd = open("/mnt/mem/cornil/usr/adm/cjobs",0)) == -1)
                {
                printf("Can't open /mnt/mem/cornil/usr/adm/cjobs\n");
                exit();
                }

        n = 0;
        j = 0;

        for(i=0;i<BMAX;i++)
                sum[i] = 0.0;

        while(read(fd,&njobs,40) > 0)
                {

                for(i=0;i<BMAX;i++)
                        sum[i] =+ njobs[i];

                n++;
                }

        close(fd);
        printf("\n\n\tDistribution of mean cpu queue lenght");
        printf("\n\t at t, t + 60', t + 120' ... with t = 3, 6, 9, ...\n\n");

        for(i=0;i<BMAX;i++)
                sum[i] =/ n;

        for(i=0;i<n;i++)
                {
                tmp = sum[i];
                printf("%d\t|",j);
                j =+ 3;

                while(tmp>0.0)
                        {
                        printf("+");
                        tmp =- 1.0;
                        }

                printf("\n");
```

```
                }

        printf("\n\n");
        exit();
}
```

p_reflx.c

```
#
        /*
         * Computes mean reflexion time at terminal ttyh and
         * standard deviation.
         *
         * Author : Cornil Dirk.
         */

#define SMAX    256

main()
{
        register i, n;
        int fd, tmp;
        double mean, sum, var;
        double sum2, ec, tmp1;
        double sqrt();

        struct  {
                int flag;
                int time[SMAX];
                } buf;

        if((fd = open("/mnt/mem/cornil/usr/adm/ttyacc",0)) == -1)
                {
                printf("Can't open /mnt/mem/cornil/usr/adm/ttyacc\n");
                exit();
                }

        sum = 0.0;
        sum2 = 0.0;
        n = 0;

        while(read(fd,&buf,sizeof(buf)) > 0)
                {
                for(i=0;i<SMAX;i++)
                        {
                        if(buf.time[i] > 0)
                                {
                                tmp = buf.time[i];
                                sum =+ tmp;
                                n++;
                                }
                        }
                }

        seek(fd,0,0);
        mean = sum / n;

        while(read(fd,&buf,sizeof(buf)) > 0)
                for(i=0;i<SMAX;i++)
                        if(buf.time[i] > 0)
                                sum2 =+ ((buf.time[i] - mean) * (buf.time[i] - m

        mean = sum / n;
        var = sum2 / n;
        ec = sqrt(var);
        printf("Mean reflexion time at ttyh :\t%.4f 1/10 sec\n",mean);
        printf("Standard deviation :\t\t%.4f 1/10 sec\n",ec);
        printf("cv :\t\t\t%.4f\n",(ec/mean));
```

```
        close(fd);
        exit();
}
```

p_jlread.c

```
/*
 *      Reads accounting file and prints the mean number of
 * jobs in the system since starting measures.
 *      It prints also the standard deviation.
 *
 * Author : Cornil Dirk.
 */
main()
{
        int rfd, isum, njobs;
        double ec, mean, jsum, csum;

        if((rfd = open("joblist",0)) <= 0)
                {
                printf("Can't open joblist\n");
                exit();
                }

        jsum = 0.0;
        csum = 0.0;
        isum = 0;

        while(read(rfd,&njobs,2) > 0)
                {
                jsum =+ njobs;
                csum =+ (njobs * njobs);
                isum++;
                }

        mean = jsum / isum;
        ec = (((1/isum) * csum) - (mean * mean));
        ec = (( csum / isum) - (mean * mean));
        printf("NOMBRE MOYEN DE JOBS DANS LE SYSTEME :\t%.2f\n",mean);
        printf("ECART-TYPE :\t\t\t%.2f\n",ec);

        close(rfd);
        exit();
}
```

p_jcpu.c

```
#

        /*
         *      Collects number of jobs in cpu queue every 3' in
         * order to check stationnarity.
         *
         * Author : Cornil Dirk.
         *
         */

#define NPROC   50

        /* stat codes

        SSLEEP  1               sleeping on high priority
        SWAIT   2               sleeping on low priority
        SRUN    3               running
        SIDL    4               intermediate state in process creation
        SZOMB   5               intermediate state in process termination
        SSTOP   6               process being traced
        */


        /* flag codes

        SLOAD   01              in core
        SSYS    02              scheduling process
        SLOCK   04              process cannot be swapped
        */

struct
{
        char    p_stat;
        char    p_flag;
        char    p_pri;          /* priority, negativ  is high */
        char    p_sig;          /* signal number sent to this process */
        char    p_uid;          /* user id, used to direct tty signals */
        char    p_time;         /* resident time for scheduling */
        char    p_cpu;          /* cpu usage for scheduling */
        char    p_nice;         /* nice for scheduling */
        int     p_ttyp;         /* controlling tty */
        int     p_pid;          /* unique process id */
        int     p_ppid;         /* process id of parent */
        int     p_addr;         /* address of swappable image */
        int     p_size;         /* size of swappable image (*64 bytes) */
        int     p_wchan;        /* event process is awaiting */
        int     *p_textp;       /* pointer to text structure */
} proc;


main()
{
        register i, j;
        int njobs[20];
        int mem, fd, r;
        int a_proc;

        struct
                {
                char text[8];
```

```
                int type;
                int value;
                } nl[1];

        setup( &nl[0] , "_proc");

        nlist("/perform",nl);

        if(nl[0].type == 0)
                {
                printf("No namelist !!\n");
                exit();
                }

        a_proc = nl[0].value;

        if((mem = open("/dev/kmem", 0)) <= 0)
                {
                printf("Can't open kernel memory\n");
                exit();
                }

        if((fd = creat("/mnt/mem/cornil/usr/adm/cjobs",0666)) == -1)
                {
                printf("Can't create /mnt/mem/cornil/usr/adm/cjobs\n");
                exit();
                }

        for(;;)
                {
                for(j=0;j<20;j++)
                        {
                        seek(mem,a_proc,0);
                        njobs[j] = 0;

                        for(i=0;i<NPROC;i++)
                                {
                                if((r = read(mem,&proc,sizeof(proc))) == -1)
                                        {
                                        printf("Error while reading kernel memor
                                        exit();
                                        }

                                if(proc.p_stat>0 && proc.p_stat<4)
                                        if((proc.p_flag & 07) != 0)
                                                njobs[j]++;
                                }

                        sleep(180);
                        }
                write(fd,&njobs,40);
                }

        close(fd);
        close(mem);
        exit();
}

setup(p,s)
char *p, *s;
```

p_jcpu.c

```
{
        while(*p++ = *s++);
}
```

p_kolsmi.c

```
#

        /*
         * Use cc -O kolsmi.c lib.c
         *
         * Apply test of Kolmogorov-Smirnov to the server active
         * time slices (in 1/60 sec) of file argv[1].
         *
         *    It is concerned here with the degree of agreement between
         * the distribution of a set of sample values (observed scores)
         * of server active times and an exponential theoretical distri-
         * bution. It determines wether the scores in the sample can
         * reasonably be tought to have come from a population having
         * the theoretical distribution.
         *
         * Author : Cornil Dirk
         *
         */

#define CMAX    256
#define NULL    0


struct
        {
        int flag;
        int nserver;
        int time[150];
        } buf;

main(argc,argv)
int argc;
char **argv;
{
        register i, j, n;
        int fd, vmax;
        double lim[CMAX];
        double sum, mean, tmp, result, ind;
        double max , sum2, ec, var, cv;
        double deviation[CMAX], ocum[CMAX], thcum[CMAX];
        double exp(), fabs(), sqrt();

        /* Check matching ... */

        if(argc != 3)
                {
                printf("kolsmi file server\n");
                exit();
                }

        /* Opens accounting file */

        if((fd = open(argv[1],0)) == -1)
                {
                printf("Can't open %s\n",argv[1]);
                exit();
                }

        printf("\n\n\t Test of Kolmogorov-Smirnov\n");
        printf("\t*************************\n\n");
```

```
/* Initialisations */

sum = 0.0;
sum2 = 0.0;
n = 0;
vmax = 0;

for(i=0;i<CMAX;i++)
        lim[i] = 0.0;

/* Reads accounting file */

while(read(fd,&buf,sizeof(buf)) > 0)
        {
        for(i=0;i<150;i++)
                {
                if(buf.time[i] < CMAX)
                        {
                        sum =+ buf.time[i];
                        n++;
                        if(buf.time[i] > vmax)
                                vmax = buf.time[i];
                        }
                }
        }

if(vmax >= CMAX)
        {
        printf("Arrays are out of range\n");
        printf("PLEASE : redefines CMAX = %d\n",vmax);
        exit();
        }
mean = sum / n;

/* Computes number of observations of the values */

seek(fd,0,0);

while(read(fd,&buf,sizeof(buf)) > 0)
        {
        for(i=0;i<150;i++)
                {
                if(buf.time[i] < CMAX)
                        {
                        sum2 =+ (buf.time[i] - mean) * (buf.time[i] - me
                        for(j=vmax;j>=0;j--)
                                {
                                if(buf.time[i] <= j)
                                        lim[j] =+ 1.0;
                                else
                                        break;
                                }
                        }
                }
        }

printf("\n\n\tDistribution of %s active time slices\n\n",argv[2]);

distprint(vmax,fd);
```

```
/*
 * If cv(standard deviation / mean) greater than 2
 * it is certainly not an exponential
 */

var = sum2 / n;
ec = sqrt(var);
cv = ec / mean;
printf("\n\n\tCoefficient of variation: \t%.4f\n", cv);

if(cv >= 2.0)
        {
        printf("cv is greater than 2\n");
        printf("Distribution is certainly not exponential\n");
        exit();
        }


/* Computes observed frequencies & theoretical frequencies */

for(i=0; i<=vmax; i++)
        {
        tmp = (-mean) * i;
        ocum[i] = lim[i] / n;
        thcum[i] = 1 - exp(tmp);
        tmp = ocum[i] - thcum[i];
        deviation[i] = fabs(tmp);
        }

/*
 * Computes maximum divergence between observed frequencies an
 * theoretical frequencies.
 */

ind = n + 0.0;
max = 0.0;

for(i=0; i<vmax; i++)
        if(deviation[i] > max)
                max = deviation[i];



/* Computes critical values of the maximum divergence */

if(n > 35)
        {
        printf("\n\n");
        printf("\tSignification level 0.20\n\n");

        result = 1.07 / sqrt(ind);
        if(result <= max)
                {
                printf("\t\tDistribution is not exponential\n\n");
                exit();
                }
        else
                {
                printf("\t\tDistribution is exponential\n\n");
```

```
/*
 * If cv(standard deviation / mean) greater than 2
 * it is certainly not an exponential
 */

var = sum2 / n;
sc = sqrt(var);
cv = sc / mean;
printf("\n\n\tCoefficient of variation: \t%.4f\n", cv);

if(cv >= 2.0)
        {
        printf("cv is greater than 2\n");
        printf("Distribution is certainly not exponential\n");
        exit();
        }


/* Computes observed frequencies & theoretical frequencies */

for(i=0; i<=vmax; i++)
        {
        tmp = (-mean) * i;
        ocum[i] = lim[i] / n;
        thcum[i] = 1 - exp(tmp);
        tmp = ocum[i] - thcum[i];
        deviation[i] = fabs(tmp);
        }

/*
 * Computes maximum divergence between observed frequencies and
 * theoretical frequencies.
 */

ind = n + 0.0;
max = 0.0;

for(i=0; i<vmax; i++)
        if(deviation[i] > max)
                max = deviation[i];



/* Computes critical values of the maximum divergence */

if(n > 35)
        {
        printf("\n\n");
        printf("\tSignification level 0.20\n\n");

        result = 1.07 / sqrt(ind);
        if(result <= max)
                {
                printf("\t\tDistribution is not exponential\n\n");
                exit();
                }
        else
                {
                printf("\t\tDistribution is exponential\n\n");
```

```
                exit();
                }

        printf("\tSignification level 0.15\n\n");

        result = 1.14 / sqrt(ind);
        if(result <= max)
                {
                printf("\t\tDistribution is not exponential\n\n");
                exit();
                }
        else
                {
                printf("\t\tDistribution is exponential\n\n");
                exit();
                }

        printf("\tSignification level 0.10\n\n");

        result = 1.22 / sqrt(ind);
        if(result <= max)
                {
                printf("\t\tDistribution is not exponential\n\n");
                exit();
                }
        else
                {
                printf("\t\tDistribution is exponential\n\n");
                exit();
                }

        printf("\tSignification level 0.05\n\n");

        result = 1.36 / sqrt(ind);
        if(result <= max)
                {
                printf("\t\tDistribution is not exponential\n\n");
                exit();
                }
        else
                {
                printf("\t\tDistribution is exponential\n\n");
                exit();
                }

        printf("\tSignification level 0.01\n\n");

        result = 1.63 / sqrt(ind);
        if(result <= max)
                {
                printf("\t\tDistribution is not exponential\n\n");
                exit();
                }
        else
                {
                printf("\t\tDistribution is exponential\n\n");
                exit();
                }
        }
else    {
```

```
                printf("Maximum divergence:\t%.4f\n",max);
                printf("See in tables for rejection or acceptation\n");
                }

        exit();
}

        /*
         * Prints observed distribution.
         */

distprint(max,fd)
int max;
int fd;
{
        register i, j, cnt;
        double value;
        double obs[CMAX];
        int sum;


        sum = 0;

        for(i=0;i<CMAX;i++)
                obs[i] = 0.0;

        /* Points to first character of the file */

        seek(fd,0,0);

        while(read(fd,&buf,sizeof(buf)) > 0)
                {
                for(i=0;i<150;i++)
                        {
                        for(j=0;j<=max;j++)
                                {
                                if(buf.time[i] < CMAX)
                                        {
                                        if(buf.time[i] == j)
                                                {
                                                sum++;
                                                obs[j] =+ 1.0;
                                                break;
                                                }
                                        }
                                }
                        }
                }


        value = 0.0;

        for(i=0;i<=max;i++)
                if(obs[i] > value)
                        value = obs[i];

        cnt = 1;

        while(value > 50.0)
                {
```

```
                cnt =* 2;
                value =/ 2;
                }

        printf("\n\t Number of observations : %d\n",sum);
        printf("\n\t + = %d observations \n\n",cnt);

        for(i=0;i<=max;i++)
                {
                printf("%d\t",i);

                obs[i] =/ cnt;
                while(obs[i] > 0.0)
                        {
                        printf("+");
                        obs[i] =- 1.0;
                        }

                printf("\n");
                }

        printf("\n\n");
        return;
}
```

p_result.c

```
#
        /*
         * Prints the last results obtained by activation of the model.
         *
         * Author : Cornil Dirk
         */

#define NSERVERS 36

struct  {
        double m_util[NSERVERS];                /* Measured utilizations */
        double s_util[NSERVERS];                /* Estimated utilizations */
        double s_njobs[NSERVERS];               /*
                                                 * Estimated number of jobs
                                                 * in the server queues
                                                 */
        double m_cpuj;                          /*
                                                 * Measured number of
                                                 * jobs in CPU queue.
                                                 */
        int njobs;                              /* Number of jobs in the system
        double lambda;                          /* System arrival rate */
        double s2utrat;                         /* System 2 utilization rate */
        double resp;                            /* System response time */
        } buf;

char *nservers[]
        {
        "cpu ",
        "rk0 ",
        "rk1 ",
        "rk2 ",
        "pm00",
        "pm01",
        "pm02",
        "pm03",
        "pm04",
        "pm05",
        "pm06",
        "pm07",
        "pm08",
        "pm09",
        "pm10",
        "pm11",
        "pm12",
        "pm13",
        "pm14",
        "pm15",
        "pm16",
        "pm17",
        "pm18",
        "pm19",
        "pm20",
        "pm21",
        "pm22",
        "pm23",
        "pm24",
        "pm25",
        "pm26",
        "pm27",
```

```
        "pm28",
        "pm29",
        "pm30",
        "pm31"
        };

main()
{

        int fd;
        double tmp;

        if((fd = open("/mnt/mem/cornil/usr/adm/res1",0)) == -1)
                {
                printf("Can't open /mnt/mem/cornil/usr/adm/res1\n");
                exit();
                }

        while(read(fd,&buf,sizeof(buf)) > 0);


        printf("\n\n\n\t\t RESULTS FOR SUBSYSTEM S2\n\n");

        pr_util();
        pr_njobs();

        printf("\n\n\t\t TOTAL RESULTS\n");
        printf("Terminal service rate:\t%.4f\n",(buf.lambda * (50 - buf.njobs)))
        printf("System S2 service rate:\t%.4f\n",buf.s2utrat);

        tmp = buf.njobs / (buf.lambda * (50 - buf.njobs));
        printf("Mean response time:\t%.4f\n",tmp);

        close(fd);

        exit();
}

        /* Print server utilizations */

pr_util()
{
        register int i;

        printf("\n\n\n");
        printf("\t*****************************************************************
        printf("\t* SERVER\t* MEASURED UTILIZATION\t* ESTIMATED UTILIZATION *\n"
        printf("\t*****************************************************************

        for(i=0;i<NSERVERS;i++)
                printf("\t*  %s\t\t*  %.4f\t\t*  %.4f\t\t*\n",nservers[i],buf.m_

        printf("\t*****************************************************************

        return;
}

        /* Print number of jobs in the server queues */

pr_njobs()
```

p_result.c

```
{
        register int i;
        double sum;

        printf("\n\n\n");
        printf("\t**************************\n");
        printf("\t*SERVER\t* Nb JOBS\t*\n");
        printf("\t**************************\n");

        sum = 0.0;

        for(i=0;i<NSERVERS;i++)
                {
                printf("\t*  %s\t*  %.2f\t\t*\n",nservers[i],buf.s_njobs[i]);
                sum =+ buf.s_njobs[i];
                }

        if(sum < 10.0)
                printf("\t* total\t*  %.2f\t\t*\n",sum);
        else
                printf("\t* total\t*  %.2f\t*\n",sum);

        printf("\t**************************\n");
        printf("\n\n\t Measured number of jobs in CPU queue :\t%.4f\n",buf.m_cpu

        return;
}
```

p_sinit.c

```
/*
 * Initialisation of system counters and accounting of
 * server utilisations.
 *
 * Author : Cornil Dirk
 *
 */

int mem;

main()
{
        int fd1, fd2;
        int at_cpu, at_rk, at_pm, at_real;
        int tvec[2];

        struct  {
                double treal;
                double tcpu;
                double trk[3];
                double tpm[32];
                } tbuf;

        struct  {
                char text[8];
                int type;
                int value;
                } nl[4];

        setup( &nl[0] , "_t_cpu");
        setup( &nl[1] , "_t_rk");
        setup( &nl[2] , "_t_pm");
        setup( &nl[3] , "_t_real");

        nlist("/perform",nl);

        if(nl[0].type == 0)
                {
                printf("No namelist !!!\n");
                exit();
                }

        if((mem = open("/dev/kmem",0)) == -1)
                {
                printf("Can't open kernel memory\n");
                exit();
                }

        if((fd1 = open("/mnt/mem/cornil/usr/adm/tacc",1)) == -1)
                {
                printf("Can't open /mnt/mem/cornil/usr/adm/tacc\n");
                exit();
                }

        if((fd2 = open("/mnt/mem/cornil/usr/adm/tuti",0)) == -1)
                {
                printf("Can't open /mnt/mem/cornil/usr/adm/tuti\n");
                exit();
                }
```

```
        tbuf.treal = lkucp(at_real);
        tbuf.tcpu = lkucp(at_cpu);
        alkucp(tbuf.trk,at_rk,3);
        alkucp(tbuf.tpm,at_pm,32);

        /* Reinitialisation of system counters */

        pinit();

        seek(fd1,0,2);
        time(tvec);
        write(fd1,&tvec,4);

        seek(fd2,0,2);
        write(fd2,&tbuf,sizeof(tbuf));

        close(fd1);
        close(fd2);

        exit();
}

setup(p,s)
char *p, *s;
{
        while(*p++ = *s++);
}

        /* Copies an array of 2 integers from kernel space into a
         * double variable in user space.
         */

double lkucp(k_adress)
int k_adress;
{
        register int r;
        int value[2];
        double result;
        double atof();

        seek(mem,k_adress,0);

        if((r = read(mem,&value,4)) == -1)
                {
                printf("Error while reading counters\n");
                exit();
                }

        result = atof(locv(value[0],value[1]));
        return(result);

}

        /* Copies an array of n integers from address "k_address" in kernel
         * space into double array buf in user space.
         */

alkucp(buf,k_adress,n)
int k_adress, n;
double *buf;
```

p_sinit.c

```
{
        register int r, i;
        int value[2];
        double atof();

        seek(mem, k_adress, 0);

        for(i=0; i<n; i++)
                {
                if((r = read(mem, &value, 4)) == -1)
                        {
                        printf("Error while reading counters\n");
                        exit();
                        }

                buf[i] = atof(locv(value[0], value[1]));
                }

        return;
}
```

```
/*
 * Ce programme fournit les taux d'utilisation des differents
 * serveurs depuis la derniere initialisation des compteurs
 * ou depuis la creation du fichier d'accounting.
 */

int mem;

main()
{
        int i, fd1, fd2, resp;
        int tvec[2];
        int at_cpu, at_rk, at_pm, at_real;
        double rktime[3], pmtime[32];
        double cputime, realtime;
        double trealsum, tcpusum, trksum[3], tpmsum[32];
        double lkucp();

        struct
                {
                char text[8];
                int type;
                int value;
                } nl[4];


        struct
                {
                double treal;
                double tcpu;
                double trk[3];
                double tpm[32];
                } tbuf;

        setup( &nl[0] , "_t_cpu");
        setup( &nl[1] , "_t_rk");
        setup( &nl[2] , "_t_pm");
        setup( &nl[3] , "_t_real");

        nlist("/perform",nl);

        if(nl[0].type == 0)
                {
                printf("No namelist !!\n");
                exit();
                }

        at_cpu = nl[0].value;
        at_rk = nl[1].value;
        at_pm = nl[2].value;
        at_real = nl[3].value;

        if((mem = open("/dev/kmem",0)) == -1)
                {
                printf("Can't open kernel memory\n");
                exit();
                }
```

```
        if((fd1 = open("/mnt/mem/cornil/usr/adm/tacc",0)) == -1)
                {
                printf("Can't open /mnt/mem/cornil/usr/adm/tacc\n");
                exit();
                }

        if((fd2 = open("/mnt/mem/cornil/usr/adm/tuti",0)) == -1)
                {
                printf("Can't open /mnt/mem/cornil/usr/adm/tuti\n");
                exit();
                }

        cputime = lkucp(at_cpu);
        realtime = lkucp(at_real);
        alkucp(rktime,at_rk,3);
        alkucp(pmtime,at_pm,32);

        if(realtime <= 0.0)
                {
                printf("Realtime is null or negative\n");
                exit();
                }

        printf("Do you want server utilizations\n");
        printf("\t1. from the last initialization\n");
        printf("\t2. from the creation of the accounting file\n");
        printf("\n\t\tYour choice?\t");

        resp = getchar();

        if(resp == '1')
                {
                while(read(fd1,&tvec,4) > 0);

                printf("\t Date of last initialization: %s\n\n\n",ctime(tve ));

                printf("\n\n\t\t Total active time :\t\t%.4f sec\n",(realtime /

                printf("\t CPU :\t%.2f\n",(cputime / realtime));

                for(i=0;i<3;i++)
                        printf("\t RK%d :\t%.2f\n",i,(rktime[i] / realtime));

                for(i=0;i<32;i++)
                        printf("\t PM%d :\t%.2f\n",i,(pmtime[i] / realtime));
                }
        else if(resp == '2')
                {
                while(read(fd2,&tbuf,296) > 0)
                        {
                        trealsum =+ tbuf.treal;
                        tcpusum =+ tbuf.tcpu;

                        for(i=0;i<3;i++)
                                trksum[i] =+ tbuf.trk[i];

                        for(i=0;i<32;i++)
                                tpmsum[i] =+ tbuf.tpm[i];
                        }
```

```
                read(fd1,&tvec,4);

                printf("\t Date of last initialization: %s\n\n\n",ctime(tvec));

                printf("\n\n\t\t Total active time :\t\t%.4f sec\n",(( realtime

                printf("\t CPU :\t%.2f\n",(( cputime + tcpusum) / (realtime + tr

                for(i=0;i<3;i++)
                        printf("\t RK%d :\t%.2f\n",i,((rktime[i] + trksum[i]) /

                for(i=0;i<32;i++)
                        printf("\t PM%d :\t%.2f\n",i,((pmtime[i] + tpmsum[i]) /

                }
        else    {
                printf("Wrong choice !\n");
                exit();
                }

        close(fd1);
        close(fd2);
        close(mem);

        exit();
}

setup(p,s)
char *p, *s;
{
        while(*p++ = *s++);
}

        /* Copies an array of 2 integers from kernel space into a
         * double variable in user space.
         */

double lkucp(k_adress)
int k_adress;
{
        register int r;
        int value[2];
        double result;
        double atof();

        seek(mem,k_adress,0);

        if((r = read(mem,&value,4)) == -1)
                {
                printf("Error while reading counters\n");
                exit();
                }

        result = atof(locv(value[0],value[1]));
        return(result);

}

        /* Copies an array of n integers from address "k_address" in kernel
         * space into double array buf in user space.
```

```
        */

alkucp(buf,k_adress,n)
int k_adress;
int n;
double buf[];
{
        register int i,r;
        int value[2];
        double atof();

        seek(mem,k_adress,0);

        for(i=0;i<n;i++)
                {
                if((r = read(mem,&value,4)) == -1)
                        {
                        printf("Error while reading counters\n");
                        exit();
                        }

                buf[i] = atof(locv(value[0],value[1]));
                }

        return;
}
```

p_model.c

```
#

        /*
         *       Model for UNIX system performance measures.
         *
         * Measures :    - service rates
         *               - utilization rates
         *               - transition probabilities
         *               - ttye reflexion time
         *               - mean number of jobs in the system
         *               - mean number of jobs in CPU queue.
         *
         * Estimates :   - utilization rates
         *               - number of jobs in the server queues
         *               - response time.
         *
         * Author : Cornil Dirk
         */

#define NRK      3
#define NPM      32
#define NSERVERS 36
#define MAXPROC 50
#define BUFSIZE 512
#define SMAX     256
#define NULL     0

        /* stat codes

        SSLEEP   1                  sleeping on high priority
        SWAIT    2                  sleeping on low priority
        SRUN     3                  running
        SIDL     4                  intermediate state in process creation
        SZOMB    5                  intermediate state in process termination
        SSTOP    6                  process being traced
        */


struct
{
        char    p_stat;
        char    p_flag;
        char    p_pri;          /* priority, negative is high */
        char    p_sig;          /* signal number sent to this process */
        char    p_uid;          /* user id, used to direct tty signals */
        char    p_time;         /* resident time for scheduling */
        char    p_cpu;          /* cpu usage for scheduling */
        char    p_nice;         /* nice for scheduling */
        int     p_ttyp;         /* controlling tty */
        int     p_pid;          /* unique process id */
        int     p_ppid;         /* process id of parent */
        int     p_addr;         /* address of swappable image */
        int     p_size;         /* size of swappable image (*64 bytes) */
        int     p_wchan;        /* event process is awaiting */
        int     *p_textp;       /* pointer to text structure */
} proc;

int mem;
struct  {
        double m_util[NSERVERS];        /* Measured utilizations */
```

```
        double s_util[NSERVERS];        /* Estimated utilizations */
        double s_njobs[NSERVERS];       /*
                                         * Estimated number of jobs
                                         * in the server queues.
                                         */
        double m_cpuj;                  /*
                                         * Measured number of jobs in
                                         * CPU queue.
                                         */
        int njobs;                      /* Number of jobs in the system */
        double lambda;                  /* System arrival rate */
        double s2utrat;                 /* System 2 utilization rate */
        double resp;                    /* System response time */
        } buf;

double sutrat[NSERVERS];                /* Server utilisations rates */
double x[NSERVERS];
double tr_prob[37];                     /* Transition probabilities */

char *nservers[]
        {
        "cpu ",
        "rk0 ",
        "rk1 ",
        "rk2 ",
        "pm00",
        "pm01",
        "pm02",
        "pm03",
        "pm04",
        "pm05",
        "pm06",
        "pm07",
        "pm08",
        "pm09",
        "pm10",
        "pm11",
        "pm12",
        "pm13",
        "pm14",
        "pm15",
        "pm16",
        "pm17",
        "pm18",
        "pm19",
        "pm20",
        "pm21",
        "pm22",
        "pm23",
        "pm24",
        "pm25",
        "pm26",
        "pm27",
        "pm28",
        "pm29",
        "pm30",
        "pm31"
        };

main()
```

```
{

        int fd;
        int status;

        if((fd = open("/mnt/mem/cornil/usr/adm/res1", 1)) == -1)
                {
                printf("Can't open /mnt/mem/cornil/usr/adm/res1\n");
                exit();
                }

        collect();

        xcompute();

        sutil();

        njserv();

        s2server();

        siserver();

        if(fork() == 0)
                {
                execl("p_sinit", 0);
                printf("Can't execute sinit\n");
                }

        wait(&status);

        seek(fd, 0, 2);
        write(fd, &buf, sizeof(buf));

        fcopy();

        close(mem);
        close(fd);

        exit();
}

collect()
{
        register int i, j;
        int at_cpu, at_rk, at_pm, at_real;
        int aout_rt, acpurt, arkrout, apmrout;
        int an_rkrun, an_pmrun, a_proc;
        double rktime[NRK], rkpr[NRK], rkact[NRK];
        double pmtime[NPM], pmpr[NPM], pmact[NPM];
        double cputime, realtime;
        double cpupr, outpr, sumpr;
        double tmp;
        double lkucp();
        double olread();
        double reflx();
        double cjcpu();

        struct
                {
```

```
        char text[8];
        int type;
        int value;
        } nl[11];

setp( &nl[0] , "_t_real");
setp( &nl[1] , "_t_cpu");
setp( &nl[2] , "_t_rk");
setp( &nl[3] , "_t_pm");
setp( &nl[4] , "_out_rt");
setp( &nl[5] , "_cpurt");
setp( &nl[6] , "_rkrout");
setp( &nl[7] , "_pmrout");
setp( &nl[8] , "_n_rkrun");
setp( &nl[9] , "_n_pmrun");
setp( &nl[10] , "_proc");

nlist("/perform",nl);

if(nl[0].type == 0)
        {
        printf("No namelist !!\n");
        exit();
        }

at_real = nl[0].value;
at_cpu = nl[1].value;
at_rk = nl[2].value;
at_pm = nl[3].value;
aout_rt = nl[4].value;
acpurt = nl[5].value;
arkrout = nl[6].value;
apmrout = nl[7].value;
an_rkrun = nl[8].value;
an_pmrun = nl[9].value;
a_proc = nl[10].value;

if((mem = open("/dev/kmem",0)) == -1)
        {
        printf("Can't open kernel memory\n");
        exit();
        }

realtime = lkucp(at_real);
cputime = lkucp(at_cpu);
cpupr = lkucp(acpurt);
outpr = lkucp(aout_rt);

alkucp(rkpr,arkrout,NRK);
alkucp(pmpr,apmrout,NPM);
alkucp(rktime,at_rk,NRK);
alkucp(pmtime,at_pm,NPM);
alkucp(rkact,an_rkrun,NRK);
alkucp(pmact,an_pmrun,NPM);

if(realtime <= 0.0)
        {
        printf("Realtime is null or negative\n");
        exit();
        }
```

```
        if(cputime <= 0.0)
                {
                printf("CPUtime is null or negative\n");
                exit();
                }

        sumpr = outpr + cpupr;

        for(i=0;i<NRK;i++)
                sumpr =+ rkpr[i];

        for(i=0;i<NPM;i++)
                sumpr =+ pmpr[i];

        if(sumpr <= 0.0)
                {
                printf("No interactions!\n");
                exit();
                }

        tr_prob[0] = cpupr / sumpr;
        buf.m_util[0] = cputime / realtime;
        sutrat[0] = (sumpr * 10.0) / cputime;

        for(j=0;j<NRK;j++)
                {
                i = j + 1;
                tr_prob[i] = rkpr[j] / sumpr;
                buf.m_util[i] = rktime[j] / realtime;

                if(rktime[j] <= 0.0)
                        sutrat[i] = 0.0;
                else
                        sutrat[i] = (rkact[j] * 10.0) / rktime[j];
                }

        for(j=0;j<NPM;j++)
                {
                i = j + 4;
                tr_prob[i] = pmpr[j] / sumpr;
                buf.m_util[i] = pmtime[j] / realtime;

                if(pmtime[j] <= 0.0)
                        sutrat[i] = 0.0;
                else
                        sutrat[i] = (pmact[j] * 10.0) / pmtime[j];
                }


        tr_prob[36] = outpr / sumpr;

        buf.lambda = reflx();
        buf.m_cpuj = cjcpu();

        buf.njobs = 0;
        tmp = olread();

        while(tmp > 0.0)
                {
```

```
                buf.njobs++;
                tmp =- 1.0;
                }

        return;
}

setp(p,s)
char *p, *s;
{
        while(*p++ = *s++);
}

        /* Copies an array of 2 integers from kernel space into a
         * double variable in user space.
         */

double lkucp(k_adress)
int k_adress;
{
        register int r;
        int value[2];
        double result;
        double atof();

        seek(mem,k_adress,0);

        if((r = read(mem,&value,4)) == -1)
                {
                printf("Error while reading counters\n");
                exit();
                }

        result = atof(locv(value[0],value[1]));
        return(result);

}

        /* Copies an array of n integers from address "k_address" in kernel
         * space into double array buf in user space.
         */

alkucp(pbuf,k_adress,n)
int k_adress, n;
double *pbuf;
{
        register int r, i;
        int value[2];
        double atof();

        seek(mem,k_adress,0);

        for(i=0;i<n;i++)
                {
                if((r = read(mem,&value,4)) == -1)
                        {
                        printf("Error while reading counters\n");
                        exit();
                        }
```

```
                pbuf[i] = atof(locv(value[0],value[1]));
                }

        return;
}

        /* Compute estimated server utilisations */

sutil()
{
        register int i;
        double normcons();

        buf.s_util[0] = normcons(buf.njobs-1,NSERVERS) / normcons(buf.njobs,NSE

        for(i=1;i<NSERVERS;i++)
                buf.s_util[i] = x[i] * buf.s_util[0];

        return;
}

        /* Compute mean number of jobs in server queues */

njserv()
{
        register int i,j,n;
        float expon();

        n = buf.njobs;

        for(i=1;i<=n;i++)
                buf.s_njobs[0] =+ normcons(n-i,NSERVERS);

        buf.s_njobs[0] =/ normcons(n,NSERVERS);

        for(i=1;i<NSERVERS;i++)
                {
                for(j=1;j<=n;j++)
                        buf.s_njobs[i] =+ (expon(x[i],j) * normcons(n-j,NSERVER

                buf.s_njobs[i] =/ normcons(n,NSERVERS);
                }
        return;
}


xcompute()
{
        register int i;
        double tmp;

        for(i=1;i < NSERVERS;i++)
                {
                if(sutrat[i] > 0.0)
                        {
                        tmp = tr_prob[i] * sutrat[0];
                        x[i] = tmp / sutrat[i];
                        }
                else
                        x[i] = 0.0;
```

```
                }

        return;
}

        /* Compute the normalization constant */

double normcons(n,m)
int n,m;
{
        register i, j;
        double c[MAXPROC];

        for(i=0;i<=n;i++)
                c[i] = 1.0;

        for(j=1;j<=n;j++)
                for(i=1;i<=n;i++)
                        c[i] =+ (x[m] * c[i-1]);

        return(c[n]);
}


s2server()
{
        return;
}


s1server()
{

        buf.s2utrat = (buf.s_util[0] * tr_prob[36]) * sutrat[0];

        buf.resp = buf.njobs / (buf.lambda * (50 - buf.njobs));

        return;
}

double olread()
{
        int rfd, isum, nj;
        double jsum;

        if((rfd = open("/mnt/mem/cornil/usr/adm/daylist",0)) <= 0)
                {
                printf("Can't open daylist\n");
                exit();
                }

        jsum = 0.0;
        isum = 0;

        while(read(rfd,&nj,2) > 0)
                {
                jsum =+ nj;
                isum++;
                }
```

```
        close(rfd);
        return(jsum / isum);
}


fcopy()
{
        register int n;
        int fdr, fdw;
        char fbuf[BUFSIZE];

        fdr = open("/mnt/mem/cornil/usr/adm/daylist", 0);
        fdw = open("/mnt/mem/cornil/usr/adm/oldlist", 1);

        while((n = read(fdr, &fbuf, BUFSIZE)) > 0)
                write(fdw, &fbuf, n);

        close(fdr);
        close(fdw);

        exit();
}

        /*
         * Computes mean reflexion time at ttye
         */

double reflx()
{
        register i, n;
        int fd;
        double mean, sum, lambda;
        double sqrt();

        struct
                {
                int flag;
                int time[SMAX];          /* Ttye reflexion times in (1/60) sec *
                } tbuf;

        if((fd = open("/mnt/mem/cornil/usr/adm/ttyacc", 0)) == -1)
                {
                printf("Can't open /mnt/mem/cornil/usr/adm/ttyacc\n");
                exit();
                }

        n = 0;
        sum = 0.0;

        /* Computes mean reflexion time at ttye */

        while(read(fd, &tbuf, sizeof(tbuf)) > 0)
                {
                for(i=0; i<SMAX; i++)
                        if(tbuf.time[i] != NULL)
                                {
                                n++;
                                sum =+ tbuf.time[i];
                                }
```

```
                }

        mean = sum / (n * 60);
        lambda = 1 / mean;

        close(fd);

        return(lambda);
}

        /*
         * Computes mean number of jobs in cpu queue.
         */

double cjcpu()
{
        register n;
        int fd, nj;
        double sum, mean;

        if((fd = open("/mnt/mem/cornil/usr/adm/cpujbs",0)) == -1)
                {
                printf("Can't open /mnt/mem/cornil/usr/adm/cpujbs\n");
                exit();
                }

        n = 0;
        sum = 0.0;

        while(read(fd,&nj,2) > 0)
                {
                sum =+ nj;
                n++;
                }

        close(fd);

        mean = sum / n;

        return(mean);
}
```

p_resp.c

```
#

        /*
         * Use cc -O p_resp.c lib.c
         *
         *    For a given reflexion time , computes response time for
         * a number of jobs in the system varying from the mean number
         * of jobs to the mean number of jobs (+/-) the standard deviation.
         *
         * Author : Cornil Dirk.
         *
         */


#define SMAX    256
#define NULL    0

main()
{
        register i, n;
        int fd1, fd2, inf, njobs;
        double sum, mean, sum2, tmp, lambda;
        double ec, var, sup;
        double sqrt();

        struct
                {
                int flag;
                int time[SMAX];          /* Ttye reflexion times in (1/60) sec *
                } buf;

        if((fd1 = open("/mnt/mem/cornil/usr/adm/daylist",0)) == -1)
                {
                printf("Can't open /mnt/mem/cornil/usr/adm/daylist\n");
                exit();
                }

        if((fd2 = open("/mnt/mem/cornil/usr/adm/ttyacc",0)) == -1)
                {
                printf("Can't open /mnt/mem/cornil/usr/adm/ttyacc\n");
                exit();
                }

        n = 0;
        sum = 0.0;
        sum2 = 0.0;

        /* Computes mean reflexion time at ttye */

        while(read(fd2,&buf,sizeof(buf)) > 0)
                {
                for(i=0;i<SMAX;i++)
                        if(buf.time[i] != NULL)
                                {
                                n++;
                                sum =+ buf.time[i];
                                }
                }

        mean = sum / (n * 60);
```

```
        lambda = 1 / mean;
        n = 0;
        sum = 0.0;

        /* Computes mean number of jobs in the system */

        while(read(fd1,&njobs,2) > 0)
                {
                sum =+ njobs;
                n++;
                }

        mean = sum / n;

        seek(fd1,0,0);

        while(read(fd1,&njobs,2) > 0)
                sum2 =+ (njobs - mean) * (njobs - mean);

        var = sum2 / n;
        ec = sqrt(var);

        sup = mean + ec;
        tmp = mean - ec;
        while(tmp > 0.0)
                {
                inf++;
                tmp =- 1.0;
                }

        printf("Mean number of jobs in the system:\t%.2f\n",mean);
        printf("Standard deviation:\t\t\t%.2f\n",ec);
        printf("\n");

        for(i=inf;i<sup;i++)
                printf("Response time for %d jobs:\t\t%.4f\tsec\n\n",i,(i / (la

        close(fd1);
        close(fd2);

        exit();
}
```

p_coll.c

```
#

        /*
         *      Collect observed and measured values and make
         * some computations in order to apply further multivariate
         * tests.
         *
         * Author : Cornil Dirk.
         */


#define MAX      12                 /* p = number of variables */
#define NSTAT    50                 /* maximum number of observations */
#define READ     0
#define CPU      0
#define PM00     4
#define PM02     6
#define PM06     10
#define PM08     12
#define PM10     14
#define PM11     15
#define PM14     18
#define PM15     19
#define PM17     21
#define PM26     30
#define NSERVERS 36

int n;                              /* Current number of observations */

struct  {
        double  obs[MAX];           /* Observed value */
        double  mes[MAX];           /* measured value */
        double  diff[MAX];          /* Difference between obs[] & mes[] */
        } stat[NSTAT];

main()
{
        vrange();

        printf("Number of observations : \t%d\n",n);

        dcompute();
        ocompute();
        mcompute();
        cmpec();

}


        /*
         *      In order to make further manipulations easier ,
         * put measures and observations in a file.
         */

vrange()
{
        register i;
        int fd;

        struct  {
```

```
        double m_util[NSERVERS];        /* Measured utilizations */
        double s_util[NSERVERS];        /* Estimated utilizations */
        double s_njobs[NSERVERS];       /*
                                         * Estimated number of jobs
                                         * in the server queues.
                                         */
        double m_cpuj;                  /*
                                         * Measured number of jobs
                                         * in the cpu queue.
                                         */
        int njobs;                      /* Number of jobs in the system
        double lambda;
        double s2utrat;
        double resp;
        } buf;

fd = open("/mnt/mem/cornil/usr/adm/cres1",0);

n = 0;

while(read(fd,&buf,sizeof(buf)) > 0 && n < NSTAT)
        {
        for(i=0;i<NSERVERS;i++)
                {
                if(i == CPU)
                        {
                        stat[n].obs[0] = buf.s_util[CPU];
                        stat[n].mes[0] = buf.m_util[CPU];
                        stat[n].diff[0] = stat[n].obs[0] - stat[n].mes[
                        }

                else if(i==PM00)
                        {
                        stat[n].obs[1] = buf.s_util[PM00];
                        stat[n].mes[1] = buf.m_util[PM00];
                        stat[n].diff[1] = stat[n].obs[1] - stat[n].mes[
                        }
                else if(i == PM02)
                        {
                        stat[n].obs[2] = buf.s_util[PM02];
                        stat[n].mes[2] = buf.m_util[PM02];
                        stat[n].diff[2] = stat[n].obs[2] - stat[n].mes[
                        }
                else if(i == PM06)
                        {
                        stat[n].obs[3] = buf.s_util[PM06];
                        stat[n].mes[3] = buf.m_util[PM06];
                        stat[n].diff[3] = stat[n].obs[3] - stat[n].mes[
                        }
                else if(i == PM08)
                        {
                        stat[n].obs[4] = buf.s_util[PM08];
                        stat[n].mes[4] = buf.m_util[PM08];
                        stat[n].diff[4] = stat[n].obs[4] - stat[n].mes[
                        }
                else if(i == PM10)
                        {
                        stat[n].obs[5] = buf.s_util[PM10];
                        stat[n].mes[5] = buf.m_util[PM10];
                        stat[n].diff[5] = stat[n].obs[5] - stat[n].mes[
```

```
                        }
                else if(i == PM11)
                        {
                        stat[n].obs[6] = buf.s_util[PM11];
                        stat[n].mes[6] = buf.m_util[PM11];
                        stat[n].diff[6] = stat[n].obs[6] - stat[n].mes[
                        }
                else if(i == PM14)
                        {
                        stat[n].obs[7] = buf.s_util[PM14];
                        stat[n].mes[7] = buf.m_util[PM14];
                        stat[n].diff[7] = stat[n].obs[7] - stat[n].mes[
                        }
                else if(i == PM15)
                        {
                        stat[n].obs[8] = buf.s_util[PM15];
                        stat[n].mes[8] = buf.m_util[PM15];
                        stat[n].diff[8] = stat[n].obs[8] - stat[n].mes[
                        }
                else if(i == PM17)
                        {
                        stat[n].obs[9] = buf.s_util[PM17];
                        stat[n].mes[9] = buf.m_util[PM17];
                        stat[n].diff[9] = stat[n].obs[9] - stat[n].mes[
                        }
                else if(i == PM26)
                        {
                        stat[n].obs[10] = buf.s_util[PM26];
                        stat[n].mes[10] = buf.m_util[PM26];
                        stat[n].diff[10] = stat[n].obs[10] - stat[n].me
                        }
                }

        stat[n].obs[11] = buf.s_njobs[0];
        stat[n].mes[11] = buf.m_cpuj;
        stat[n].diff[11] = stat[n].obs[11] - stat[n].mes[11];

        n++;
        }

    close(fd);
    return;
}

    /* Computes mean of differences and put them in a file */

dcompute()
{
    register i, j, k;
    int fd, fdc, fdl;
    double sum[MAX], mean[MAX];
    double a[MAX][MAX];

    fd = fcre("fmean");
    fdc = fcre("ca");
    fdl = fcre("la");

    for(i=0; i<MAX; i++)
        {
        sum[i] = 0.0;
```

```
                for(j=0; j<MAX; j++)
                        a[i][j] = 0.0;
                }

        for(i=0; i<n; i++)
                for(j=0; j<MAX; j++)
                        sum[j] =+ stat[i].diff[j];

        for(i=0; i<MAX; i++)
                mean[i] = sum[i] / n;

        write(fd, &mean, 96);
        close(fd);

        for(i=0; i<MAX; i++)
                {
                for(j=0; j<MAX; j++)
                        {
                        for(k=0; k<n; k++)
                                {
                                if(i==j)
                                        a[i][j] =+ (stat[k].diff[i] - mean[i])
                                                * (stat[k].diff[i] - mean[i]);
                                else
                                        a[i][j] =+ (stat[k].diff[i] - mean[i])
                                                * (stat[k].diff[j] - mean[j]);
                                }
                        }
                }

        fsave(fdc, fd1, a);

        return;
}

        /* Computes mean of observed values for the p variables
         * and computes also sum of squares and product of deviates
         * for them.
         */

mcompute()
{
        register i, j, k;
        int fd, fdc, fd1;
        double sum[MAX], mean[MAX];
        double a[MAX][MAX];

        fd = fcre("fmmean");
        fdc = fcre("ca2");
        fd1 = fcre("la2");

        for(i=0; i<MAX; i++)
                {
                sum[i] = 0.0;
                for(j=0; j<MAX; j++)
                        a[i][j] = 0.0;
                }

        for(i=0; i<n; i++)
                for(j=0; j<MAX; j++)
```

```
                        sum[j] =+ stat[i].mes[j];


        printf("Mean mesured utilization\n\n");

        for(i=0;i<MAX;i++)
                {
                mean[i] = sum[i] / n;
                printf("U%d :\t%.8f\n",i,mean[i]);
                }

        write(fd,&mean,96);
        close(fd);

        for(i=0;i<MAX;i++)
                {
                for(j=0;j<MAX;j++)
                        {
                        for(k=0;k<n;k++)
                                {
                                if(i==j)
                                        a[i][j] =+ (stat[k].mes[i] - mean[i])
                                                * (stat[k].mes[i] - mean[i]);
                                else
                                        a[i][j] =+ (stat[k].mes[i] - mean[i])
                                                * (stat[k].mes[j] - mean[j]);
                                }
                        }
                }

        fsave(fdc,fdl,a);

        return;
}

        /*
         *      Computes mean observed utilizations for the p variables
         * and the matrix of sum of squares and product of deviates.
         */


ocompute()
{
        register i,j,k;
        int fd, fdc, fdl;
        double sum[MAX], mean[MAX];
        double a[MAX][MAX];

        fd = fcre("fomean");
        fdc = fcre("ca1");
        fdl = fcre("la1");


        for(i=0;i<MAX;i++)
                {
                sum[i] = 0.0;
                for(j=0;j<MAX;j++)
                        a[i][j] = 0.0;
                }
```

```
        for(i=0; i<n; i++)
                for(j=0; j<MAX; j++)
                        sum[j] =+ stat[i].obs[j];

        printf("Mean observed utilizations\n");

        for(i=0; i<MAX; i++)
                {
                mean[i] = sum[i] / n;
                printf("U%d :\t%.8f\n", i, mean[i]);
                }

        write(fd, &mean, 96);
        close(fd);

        for(i=0; i<MAX; i++)
                {
                for(j=0; j<MAX; j++)
                        {
                        for(k=0; k<n; k++)
                                {
                                if(i==j)
                                        a[i][j] =+ (stat[k].obs[i] - mean[i])
                                                * (stat[k].obs[i] - mean[i]);
                                else
                                        a[i][j] =+ (stat[k].obs[i] - mean[i])
                                                * (stat[k].obs[j] - mean[j]);
                                }
                        }
                }

        fsave(fdc, fdl, a);

        return;
}


cmpec()
{
        register i, j;
        double mean[MAX], var[MAX], sum[MAX], ec[MAX];
        double sqrt();
        for(i=0; i<MAX; i++)
                {
                sum[i] = 0.0;
                for(j=0; j<n; j++)
                        sum[i] =+ stat[j].diff[i];
                mean[i] = sum[i] / n;
                printf("mdiff[%d]:\t%.8f\n", i, mean[i]);
                }

        for(i=0; i<MAX; i++)
                {
                var[i] = 0.0;
                for(j=0; j<n; j++)
                        var[i] =+ (stat[j].diff[i] - mean[i])
                                * (stat[j].diff[i] - mean[i]);
                var[i] =/ n;
                ec[i] = sqrt(var[i]);
                printf("ec[%d]:\t%.8f\n", i, ec[i]);
```

```
                }
        return;
}
```

p_comp.for

```
c
c       Use exec compute.for,sys:imsl/search
c
c       Program running on DEC 2060
c
c Input : file corn24.don (matrix 1 of sum of squares and
c                          product of deviates for population 1)
c         file corn25.don (matrix 2 of sum of squares and
c                          product of deviates for population 2)
c
c Remarks : Input and output on files in ASCII mode.
c
c Output : file corn26.don (inversion of matrix 1)
c          file corn27.don (inversion of matrix 2)
c          file corn29.don (inversion of matrix 1 + matrix 2)
c
c Author : Lambion Patrick & Cornil Dirk.
c


      integer itmp1, itmp2, itmp3, itmp4
      integer ier

      real d1, d2, de, da1, da2
      real a1(12,12), a2(12,12), sa1(12,12), sa2(12,12)
      real suma(12,12), e(12,12), sums(12,12), tmp(12,12)
      real ssmus(12,12), i1(12,12), i2(12,12)
      real wkarea(26), b(12)

10    format(f15.8)
11    format(e20.15)
20    format(10f10.5)
30    format(12f10.5)
40    format(i5)
50    format(' Erreur No ')

      open(unit=24,device='dsk',file='corn24.don',access='seqin')
      open(unit=25,device='dsk',file='corn25.don',access='seqin')

      open(unit=26,device='dsk',file='corn26.don',access='seqout')
      open(unit=27,device='dsk',file='corn27.don',access='seqout')
      open(unit=28,device='dsk',file='corn28.don',access='seqout')
      open(unit=29,device='dsk',file='corn29.don',access='seqout')
      open(unit=30,device='dsk',file='corn30.don',access='seqout')


      do 100 itmp1=1,12,1
      do 100 itmp2=1,12,1
      read(24, 10) a1(itmp1, itmp2)
100   continue

110   format(' ',12e10.3)
111   format(' origine 24')
      write(5, 111)
      write(5, 110) a1

      do 200 itmp1=1,12,1
      do 200 itmp2=1,12,1
      read(25, 10) a2(itmp1, itmp2)
200   continue
```

```
210     format(' ',12e10.3)
211     format(' origine 25')
        write(5, 211)
        write(5, 210) a2

       c: Substitute full symetric storage by symetric storage
       c: IMSL

        call VCVTFS(a1, 12, 12, sa1)
        call VCVTFS(a2, 12, 12, sa2)

       c: Add full symetric matrix a1 to symtric stored matrix sa2
       c: IMSL

        call VUAFS(a1, 12, 12, sa2, suma, 12)

        do 300 itmp1 = 1, 12, 1
        do 300 itmp2 = 1, 12, 1
        sums(itmp1, itmp2) = suma(itmp1, itmp2) / 11.
        e(itmp1, itmp2) = suma(itmp1, itmp2) / 2.
        suma(itmp1, itmp2) = suma(itmp1,itmp2) * 10.
300     continue

        do 310 itmp1=1,12,1
        do 310 itmp2=1,12,1

       c: Multilpy matrix by 10 ** 4 because of precision problems

        e(itmp1,itmp2)=e(itmp1,itmp2) * 10000.
        a1(itmp1,itmp2)=a1(itmp1,itmp2) * 10000.
        a2(itmp1,itmp2)=a2(itmp1,itmp2) * 10000.
        suma(itmp1,itmp2)=suma(itmp1,itmp2) * 10000.
        sums(itmp1,itmp2)=sums(itmp1,itmp2) * 10000.
310     continue

        d1 = 5

       c: computes inverse and determinant of matrix e
       c: IMSL

        call LINV3F(e,b,1,12,12,d1,d2,wkarea,ier)
        if (ier .NE. 0) write(3, 50)
        write(3, 11) d1
        write(3, 11) d2
        de = d1 * (2 ** d2)


        d1 = 5
        call LINV3F(a1,b,1,12,12,d1,d2,wkarea,ier)
        if (ier .NE. 0) write(3, 50)
        da1 = d1 * (2 ** d2)

        d1 = 4
        call LINV3F(a2,b,1,12,12,d1,d2,wkarea,ier)
        if (ier .NE. 0) write(3, 50)
        da2 = d1 * (2 ** d2)

        d1 = -1.0
        call LINV3F(suma,b,1,12,12,d1,d2,wkarea,ier)
```

```
      if (ier .NE. 0) write(3, 50)

      d1 = -1.0
      call LINV3F(sums,b,1,12,12,d1,d2,wkarea,ier)
      if (ier .NE. 0) write(3, 50)

      do 320 itmp1=1,12,1
      do 320 itmp2=1,12,1
      e(itmp1,itmp2)=e(itmp1,itmp2) * 10000.
      a1(itmp1,itmp2)=a1(itmp1,itmp2) * 10000.
      a2(itmp1,itmp2)=a2(itmp1,itmp2) * 10000.
      suma(itmp1,itmp2)=suma(itmp1,itmp2) * 10000.
      sums(itmp1,itmp2)=sums(itmp1,itmp2) * 10000.
320   continue

      call VCVTFS(sums, 12, 12, ssums)

     c: Multiply matrix sums by matrix sa1
     c: IMSL

      call VMULFS(sums, sa1, 12, 12, 12, tmp, 12)
      call VMULFS(tmp, ssums, 12, 12, 12, i1, 12)
      call VMULFS(sums, sa2, 12, 12, 12, tmp, 12)
      call vmulfs(tmp, ssums, 12, 12, 12, i2, 12)


1010  format(' DE = ', e10.3)
      write(5, 1010) de
1020  format(' DA1 = ', e10.3)
      write(5, 1020) da1
1030  format(' da2 = ', e10.3)
      write(5, 1030) da2

1040  format(' MATRICE A1 ')
      write(5, 1040)
      call affich(26, a1)
1050  format(' MATRICE A2 ')
      write(5, 1050)
      call affich(27, a2)
1060  format(' MATRICE E ')
      write(5, 1060)
      call affich(28, e)
1070  format(' MATRICE SUMA ')
      write(5, 1070)
      call affich(29, suma)
1080  format(' MATRICE SUMS ')
      write(5, 1080)
      call affich(30, sums)

      stop
      end

      subroutine affich(arg1, arg2)
      integer arg1
      real arg2(12,12)

      integer itmp1, itmp2

20    format(' ',f20.7)
30    format(' ',12e10.3)
```

p_comp.for

```
      write(5, 30) arg2
      do 100 itmp1 = 1, 12, 1
      do 100 itmp2 = 1, 12, 1
      write(arg1, 20) arg2(itmp1, itmp2)
100   continue
      return
      end
```