MASTER EN SCIENCES INFORMATIQUES

Simulation d'un réseau local Ethernet utilisant un système de fichiers distribués

Derzelle, Gérald; Despriet, Vincent

Award date:
1992

Awarding institution:
Universite de Namur

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique

Rue Grandgagnage, 21
B-5000 NAMUR (Belgium)

Simulation d'un réseau local Ethernet utilisant un système de fichiers distribués

Mémoire présenté en vue de l'obtention
du diplôme de Maître en Informatique

Gérald Derzelle
Vincent Despriet

Promoteur : M. Noirhomme-Fraiture

Année académique 1991-1992

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique
rue de Bruxelles 61, B-5000 NAMUR
Tél. 081-72.41.11
Télex 59222 facnam-b
Téléfax 081-23.03.91

Simulation d'un réseau local Ethernet utilisant un système de fichiers distribués

Gérald Derzelle
Vincent Despriet

Résumé

Pour le gestionnaire d'un système informatique, il est toujours intéressant de disposer d'un outil permettant de mesurer les performances d'un réseau. Le but de ce travail est de développer un programme de simulation qui, à partir de la configuration d'un réseau fournit tous les résultats statistiques désirés. La modélisation d'un tel système a exigé une étude approfondie du système d'exploitation UNIX et du système de fichiers distribués (NFS). Des tests statistiques sont utilisés pour le traitement des résultats. Lors de la modélisation, il faut effectuer des hypothèses simplificatrices qui ne peuvent pas être trop éloignées de la réalité; la qualité du simulateur en dépend.

Abstract

System managers are often interested in tools wich are able to mesure network performances. The aim of this work is to develop a simulation program wich is able to produce all statistical results from a given network configuration. The modelisation of such a system has asked an elaborate study of the UNIX operating system and the Network File System (NFS) developped by SUN. The treatment of the results uses statistical tests. During the modelisation, it is necessary to make simplifying hypothesis wich can't be too far from reality; this stage have an influence on the quality of the final program.

Mémoire de maîtrise en Informatique
Septembre 1992
Promoteur : M. Noirhomme-Fraiture

Table des matières

3. Supports statistiques de la simulation

4. Conception du simulateur

5. Prise des mesures

6. Validation du simulateur

7. Conclusions

Remerciements

Nous tenons à remercier les différentes personnes qui nous ont aidés lors de l'élaboration de ce travail.
Nous avons notamment pu bénéficier des conseils avisés de notre promoteur madame M. Noirhomme.
Soulignons également les nombreuses interventions pertinentes de notre maître de stage monsieur de Vaney qui n'a jamais hésité à nous consacrer de larges moments de son temps précieux.
Nous pensons aussi à Anne Putmans et Christian Linck qui, par leurs précieux commentaires, ont permis d'améliorer la rédaction de la version définitive du mémoire.
Enfin, nous voulons remercier toutes les personnes qui, de près ou de loin, ont contribué à la réalisation de ce travail.

Chapitre 1: Introduction

1.1) Introduction du mémoire

L'évolution des réseaux locaux, avec notamment l'apparition des systèmes de fichiers distribués, a considérablement compliqué la modélisation de tels systèmes. De plus, la manière dont le système d'exploitation d'un réseau (en général UNIX) gère les files d'attente est très complexe. Par conséquent, la création d'un modèle analytique est souvent impossible. La seule alternative consiste à effectuer une simulation du système.

Le but de ce travail est de développer un programme qui permet d'analyser le comportement d'un réseau afin d'en améliorer les performances. De plus, une entreprise désireuse d'installer un nouveau réseau peut employer ce programme de manière à adapter la configuration du système à ses besoins.

Ce mémoire est le prolongement d'un stage que nous avons effectué au Luxembourg pour la société World Systems. Nous présentons dans ce chapitre la société World Systems et détaillons l'évolution de notre travail.

La création d'un simulateur exige une bonne connaissance du système existant. Le second chapitre décrit le système d'exploitation UNIX et le système de fichiers distribués NFS. Cette partie a une grande importance pour l'élaboration du modèle.

Les résultats fournis par le simulateur doivent subir un traitement statistique. De plus, les informations recueillies doivent respecter certaines conditions. Les éléments statistiques que nous intégrons dans notre programme sont expliqués au chapitre trois.

Lors de la conception du modèle, il faut effectuer des hypothèses simplificatrices. La qualité du simulateur en dépend directement. Au plus elles sont fortes, au plus les résultats de la simulation s'éloignent de la réalité. Ces hypothèses sont décrites au chapitre quatre. Nous y détaillons également la conception logique et physique du programme et de la base de données.

World Systems dispose au Luxembourg d'un réseau Ethernet. Lors de notre stage, nous avons effectué des mesures sur le réseau. Les méthodes que nous avons employées pour prendre ces mesures sont décrites au cinquième chapitre.

Tout modèle de simulation doit être validé. Au chapitre six, nous présentons différentes méthodes de validation accompagnées chacune des problèmes liés à leur application.

Le chapitre sept reprend les extensions possibles. Nous clôturons par une analyse des résultats.

1.2) Critères du choix de notre stage

Ce stage nous a permis de nous familiariser avec l'utilisation des réseaux locaux basés sur une architecture Unix.

Nous y avons vu l'opportunité d'approfondir nos connaissances sur ces deux points dont l'importance dans le milieu informatique est considérable.

Il nous a également semblé fort intéressant d'étudier la possibilité d'accéder à un réseau basé sur une architecture Unix à partir d'un PC.

Enfin, nous avons été attirés par la possibilité de mettre en pratique nos connaissances acquises au cours concernant les mesures de performances des réseaux, et de voir les différentes difficultés rencontrées lors du passage de la théorie à la pratique.

trimestre de l'année 91. Le projet ayant pris du retard, il nous était impossible de réaliser ce travail.

1.4.2) Recherche d'un modèle analytique

Ne disposant que d'un réseau NFS, nous avons étudié son comportement et essayé de trouver le modèle analytique correspondant. Il s'est avéré trop complexe voire impossible à définir pour les raisons suivantes:

-gestion complexe des processus sous Unix
-présence de multiples files d'attente au processeur
-partage du processeur
-discipline de file d'attente complexe basée entre autres sur la priorité des processus et sur le temps passé au serveur CPU déjà octroyé

Il est néanmoins possible de trouver un modèle analytique mais uniquement au prix de fortes hypothèses restrictives qui nous éloigneraient trop de la réalité.

1.4.3) Simulation

Après avoir exposé le problème à Mr de Vaney, il nous a fait savoir qu'il serait intéressant pour la société de disposer d'un logiciel d'aide à la décision pour le choix d'un réseau. Ce logiciel devrait permettre de tester les performances des différentes configurations possibles lors de la construction d'un nouveau réseau. Le client pourrait dès lors, en fonction de ses critères, choisir la configuration qui lui semble la plus appropriée.

Chapitre 2 : Description du système informatique de Luxembourg

2.1) Introduction

Pour concevoir le simulateur, il est nécessaire que nous disposions d'un modèle de référence. World Systems nous a fourni un espace de travail sur le réseau de Hamm. Seul l'accès aux données confidentielles de la "Commission Européenne" nous a été interdit. Lorsque nos travaux demandaient le privilège 'Super User', nous exposions nos souhaits au gestionnaire du système. Celui-ci acceptait d'exécuter les tâches si elles n'entravaient pas le bon déroulement du système.

Une étude approfondie du système nous a permis de définir l'architecture du réseau et de comprendre son fonctionnement. Il nous a dès lors été possible de concevoir le simulateur.

2.2) Configuration du réseau

Le réseau de Hamm est de type Ethernet. Un câble coaxial souple sert de support de communication sur lequel sont connectés le serveur, les trente PC et l'imprimante laser. Le serveur est une station SUN 3/80. Il possède un disque dur de 600 Mégabytes et est basé sur une architecture UNIX. Chaque PC peut accéder au système UNIX par TELNET et un émulateur de terminal (VT100).

2.3) Utilisation du réseau

Le réseau est principalement destiné à la centralisation des fichiers DOS et à l'impression de rapports à partir des PC. Une base de données regroupe des informations statistiques pour la 'Commission Européenne' (EUROSTAT); leurs traitements nécessitent beaucoup d'opérations de calcul, d'où une grande occupation du processeur. Les mises à jour de la base de données se font durant la nuit. Les informations sont transmises depuis les ordinateurs de la CCE vers le serveur de Hamm via une ligne X25.

2.4) Description des moyens logiciels

Les utilisateurs PC (la majorité) travaillent principalement avec WORD 5.0 et EXCELL. PC-NFS est une interface entre les systèmes d'exploitation UNIX et DOS. Il permet d'accéder au disque dur Unix et à l'imprimante du réseau à partir du DOS. Ceci est totalement transparent pour l'utilisateur. En effet, il lui suffit de choisir J: comme unité logique, et chaque accès à un fichier se trouvant sur J: est transformé en une requête PC-NFS (rpc.pcnf). De même, pour accéder à une imprimante du réseau, il suffit de faire son choix et automatiquement la demande d'impression est changée en une requête PC-NFS vers l'imprimante désirée.

La base de données est accessible via le langage interprété APL. Ce langage peut être employé du côté PC ainsi que du côté UNIX. Au cours de notre séjour à Hamm, il s'est avéré qu'il était peu utilisé; les utilisateurs de la base de données s'en servent en général uniquement pour transférer les informations vers un tableur (EXCELL), celui-ci étant beaucoup plus ergonomique.

Quant au réseau UNIX proprement dit, il n'est en fait utilisé que par le gestionnaire du système et deux chercheurs.

2.5) Présentation des supports software du réseau

2.5.1) UNIX[1]

2.5.1.1) Introduction

Notre programme de simulation a pour but de simuler le processeur, le disque et l'imprimante. Le comportement de ces serveurs est déterminé par le système d'exploitation.

Le réseau de Hamm est basé sur le système d'exploitation Unix. Celui-ci comprend deux parties. La première est constituée des programmes (shell, mail, programmes de traitement de texte, ...) tandis que la seconde concerne la prise en charge de ces programmes et de leurs services.

Il tourne sur un éventail de machines de puissance de calcul assez large allant des microprocesseurs aux gros ordinateurs.

Voici quelques raisons qui expliquent en partie son succès :

-il a été écrit dans un langage de haut niveau (C), ce qui le rend facile à lire, à comprendre, à charger et à porter sur d'autres machines;

- le code est disponible;

-il utilise un système de fichiers hiérarchique qui permet une maintenance facile et une implémentation efficace;

-c'est un système multi-utilisateur et multitâche : chaque utilisateur peut exécuter simultanément plusieurs processus;

-il cache l'architecture de la machine à l'utilisateur, rendant plus facile l'écriture de programmes qui s'exécutent sur des implémentations matérielles différentes.

Les grands traits de l'architecture du système UNIX peuvent être représentés par la figure 2.1.

---

[1]Nous nous sommes inspirés de [BACH86] pour rédiger la description du système d'exploitation UNIX

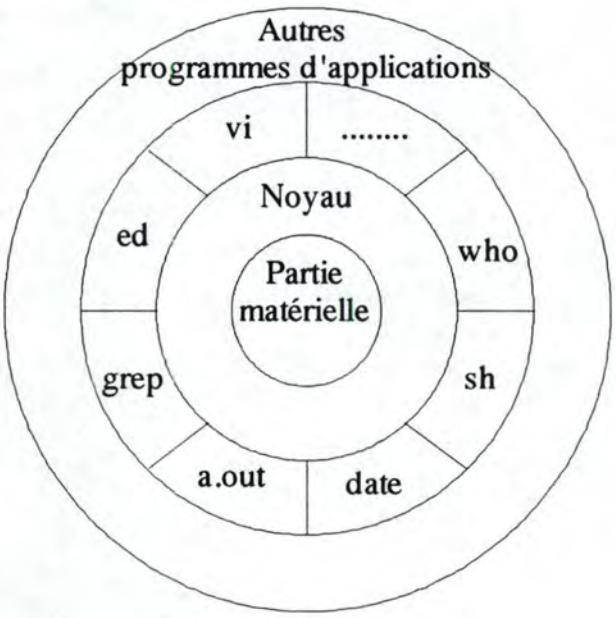


fig 2.1 : structure du système Unix

La partie matérielle, au centre du diagramme, fournit des services de base au système d'exploitation. Celui-ci, communément appelé noyau, procure des services communs aux programmes. Il les isole des particularités de la partie matérielle ce qui les rend plus faciles à utiliser sur des systèmes UNIX tournant sur des matériels différents (si ces programmes ne considèrent pas la partie matérielle sous-jacente). La couche supérieure au noyau est composée non seulement des programmes tels le shell et les éditeurs (vi et ed) mais également d'un ensemble de programmes appartenant aux configurations standards du système (connus sous le nom de commande) et de programmes utilisateurs privés (ex: a.out). Ces programmes interagissent avec le noyau en invoquant un ensemble bien défini d'appels systèmes qui lui demandent de réaliser différentes opérations pour le programme appelant et d'échanger des données avec celui-ci. L'ensemble des appels systèmes et les algorithmes qui les implémentent forment le corps du noyau. D'autres programmes d'applications peuvent être élaborés à partir de programmes de niveau inférieur, d'où l'existence de la couche extérieure de la figure. Bien que

la figure représente une hiérarchie à deux niveaux de programmes d'applications, les utilisateurs peuvent élargir la hiérarchie à plusieurs niveaux.

Services du système d'exploitation

Le noyau fournit un ensemble de services tels :

- le contrôle de l'exécution des processus en permettant leur création, leur fin ou leur suspension, et leur communication;
- l'élection des processus d'une manière équitable en vue de leur exécution dans l'unité centrale. Les processus partagent l'unité centrale d'une manière "temps partagé" : le noyau suspend le processus en cours d'exécution à l'unité centrale dès que sa tranche de temps s'est écoulée et élit un autre processus en vue de son exécution. Il reprend plus tard le processus suspendu afin de poursuivre son exécution.
- l'attribution de la mémoire principale à un processus en exécution. Le noyau permet aux processus de partager des parties de leur espace d'adressage sous certaines conditions mais il protège de l'extérieur l'espace d'adressage privé d'un processus. Si le système épuise la mémoire libre, le noyau libère de la mémoire en écrivant un processus temporairement en mémoire secondaire, appelée périphérique de swap. Si le noyau écrit des processus entiers sur un périphérique de swap, l'implémentation du système UNIX est appelée un système à swapping, par contre, s'il écrit des pages de mémoire sur un périphérique de swap, elle est appelée système à pagination.
- l'allocation de la mémoire secondaire pour rendre efficace le stockage et l'extraction des données de l'utilisateur. Ce service constitue le système de fichiers. Le noyau attribue un emplacement secondaire aux fichiers utilisateurs, récupère les emplacements inutilisés, structure le système de fichiers d'une manière bien compréhensible, et protège les fichiers utilisateurs d'accès illégaux.
- la possibilité aux processus de contrôler l'accès aux périphériques tels que les terminaux, les contrôleurs de bandes, les contrôleurs de disques et les réseaux.

Le noyau fournit tous ses services de manière transparente.

Environnement de traitement

Un programme est un fichier exécutable tandis qu'un processus est une occurrence d'un programme en exécution. Plusieurs processus peuvent s'exécuter simultanément sur les systèmes UNIX (multiprogrammation ou multitâche) sans limitation logique, et plusieurs occurrences d'un programme peuvent exister simultanément dans le système.

Divers appels systèmes permettent aux processus de créer de nouveaux processus, de terminer un processus, de synchroniser des étapes d'exécutions de processus et de contrôler la réaction à des événements divers. Les processus s'exécutent indépendamment les uns des autres en dehors du fait qu'ils doivent utiliser les mêmes appels systèmes.

Les interruptions et les exceptions

Le système UNIX permet aux périphériques tels que les périphériques d'entrées-sorties ou l'horloge du système d'interrompre l'unité centrale d'une manière asynchrone. Sur réception d'une interruption, le noyau sauvegarde son contexte courant (une image de ce que réalisait le processus), détermine la cause de l'interruption, et traite l'interruption. Après le traitement de celle-ci, le noyau restaure le contexte interrompu et poursuit comme si rien ne s'était passé. Une priorité est donnée aux périphériques selon l'ordre dans lequel les interruptions doivent être traitées.

Une situation d'exception se réfère aux événements non attendus provoqués par un processus tels un adressage illégal de la mémoire, l'exécution d'instructions privilégiées, la division par zéro,... Les exceptions sont distinctes des interruptions qui sont provoquées par des événements externes aux processus.

Elles surviennent au milieu d'une instruction tandis qu'une interruption est prise en compte entre l'exécution de deux instructions.

Niveaux d'exécution du processus

Le noyau doit parfois empêcher l'occurrence d'interruptions pendant des activités critiques qui pourraient provoquer la corruption des données si elles étaient permises.

Les ordinateurs ont généralement un ensemble d'instructions privilégiées qui positionnent le niveau d'exécution du processus dans le mot d'état du processeur. En positionnant le niveau d'exécution du processus à certaines valeurs, le noyau masque les interruptions de ce niveau et des niveaux plus bas, permettant seulement des interruptions de plus haut niveau (fig. 2.2).

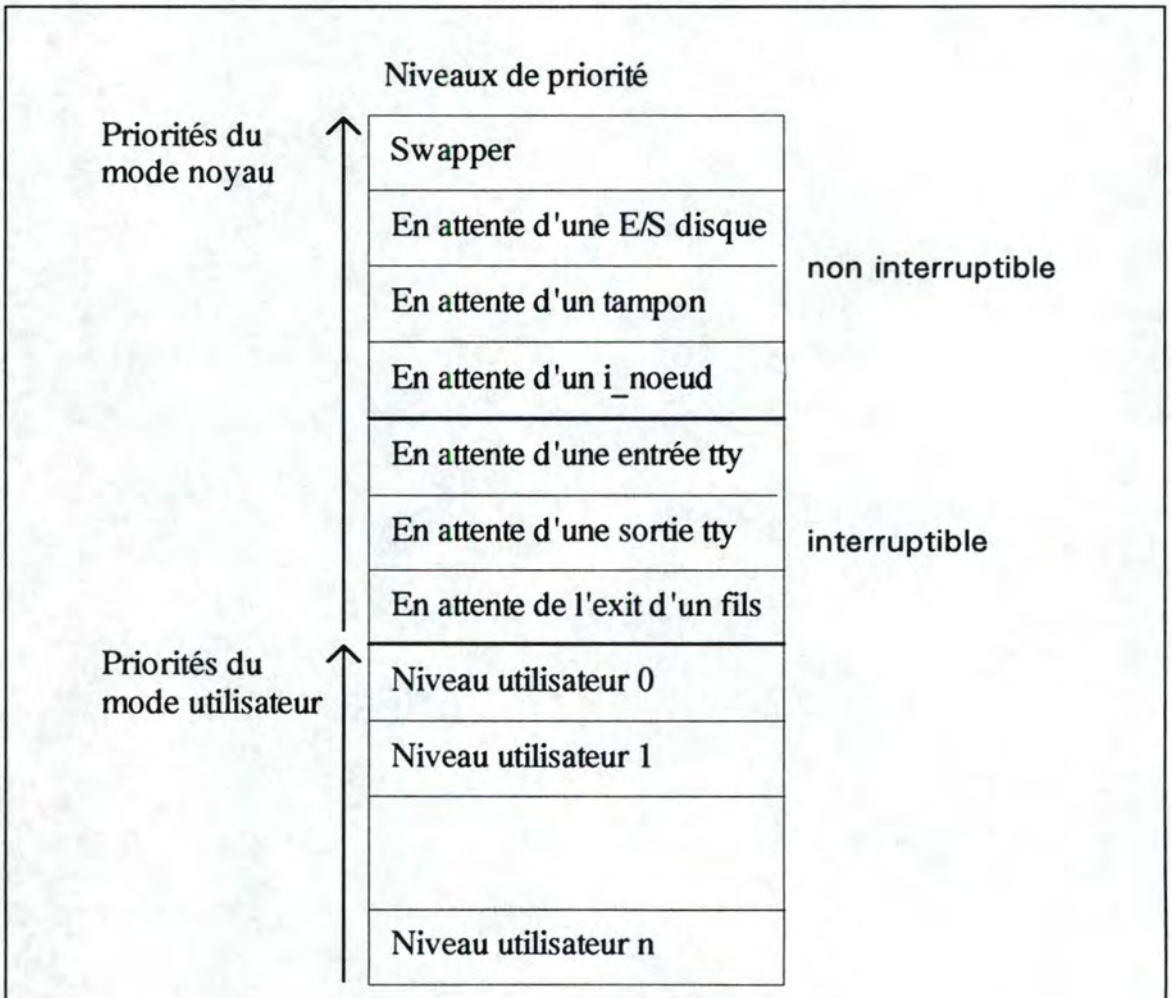


fig. 2.2 : Niveaux d'exécution d'un processus

Deux concepts centraux

Les deux entités, fichiers et processus, sont les deux concepts centraux dans le modèle du système UNIX. En effet, le sous-système de fichiers et le sous-système de contrôle de processus sont deux composantes majeures du noyau.

Le sous-système de fichiers administre les fichiers, allouant de l'espace pour un fichier, administrant l'espace libre , contrôlant l'accès aux fichiers et extrayant les données pour les utilisateurs.

Les processus interagissent avec le sous-système de fichiers via un ensemble spécifique d'appels systèmes (open, close, read, write,...). Le sous-système de contrôle de processus est responsable de la synchronisation des processus, de la communication entre les processus, de la gestion de la mémoire (contrôle de l'allocation de la mémoire) et de l'ordonnancement des processus.

Le sous-système de fichiers et le sous-système de contrôle des processus interagissent lors du chargement d'un fichier en mémoire en vue de son exécution. Le sous-système des processus lit des fichiers exécutables en mémoire avant de les exécuter.

Dans le cadre de notre simulation, nous allons approfondir la notion de processus et celle de sous-système de contrôle de processus. Ceci est indispensable pour comprendre le fonctionnement du processus et ainsi pouvoir le simuler correctement. En ce qui concerne les notions de fichier et de sous-système de fichiers, nous n'allons pas entrer dans de trop amples détails car la technique utilisée pour gérer les fichiers sur le disque n'a pas d'influence sur les résultats de la simulation.

2.5.1.2) Le concept de processus

Processus

Un processus est l'exécution d'un programme. Plusieurs processus peuvent être des exemplaires d'un même programme.
Ils communiquent avec d'autres processus et avec le reste du monde via les appels systèmes.

Les mécanismes de communication interprocessus permettent à des processus quelconques d'échanger des données et de synchroniser leur exécution. Les différentes formes sont : les tubes, les tubes nommés, les signaux, le réseau et les sockets.

Deux modes d'exécution

L'exécution de processus utilisateurs sur des systèmes UNIX comporte deux niveaux: utilisateur et noyau. Quand un processus exécute un appel système, le mode d'exécution du processus passe du mode utilisateur au mode noyau. Le système d'exploitation exécute et tente de servir la requête utilisateur, en retournant un code d'erreur s'il échoue. Même si l'utilisateur ne fait pas de requêtes explicites aux services du système d'exploitation, celui-ci fait des opérations de gestion qui se rapportent aux processus utilisateurs, en traitant les interruptions, en ordonnançant les processus, en gérant la mémoire,...

Sur des systèmes UNIX, les deux modes, noyau et utilisateur, suffisent. Ils se différencient comme suit:

- les processus en mode utilisateur peuvent accéder à leurs instructions et à leurs données propres mais pas aux instructions ni aux données du noyau (ou celles des autres processus). Les processus en mode noyau, cependant, peuvent accéder aux adresses noyau et utilisateur.
- certaines instructions machine sont privilégiées et provoquent une erreur quand elles sont exécutées en mode utilisateur.

Bien que le système s'exécute dans l'un des deux modes, le noyau répond au nom d'un processus utilisateur. Il n'est pas constitué d'un ensemble particulier de processus qui tournent en parallèle avec les processus utilisateurs, mais il est une partie de chaque processus utilisateur.

L'attribution des ressources à un processus et la réalisation de diverses opérations ne sont pas effectuées par le noyau mais par le processus lui-même s'exécutant en mode noyau.

Etat d'un processus

La durée de vie d'un processus peut être divisée en un ensemble d'états, chacun avec certaines caractéristiques qui décrivent le processus.

Comme le système d'exploitation permet à plusieurs processus de s'exécuter simultanément et que les processeurs ne peuvent traiter les instructions que séquentiellement, les ordinateurs monoprocesseur doivent faire appel à une technique appelée "partage de processeur". Ceci consiste à attribuer séquentiellement une petite tranche de temps à chaque processus nécessitant du temps processeur. Les processus étant servis à tour de rôle, ils doivent passer de l'état d'attente à l'état de service et vice-versa. En UNIX, il existe plusieurs états différents et la vie d'un processus peut se résumer à un ensemble de transitions entre ces différents états (fig 2.3).

L'existence de ces différents états impose au système d'exploitation de connaître à tout moment l'état dans lequel chacun des processus actifs se trouve. Celui-ci est décrit par les deux structures de données suivantes:

* une entrée dans la table des processus
* une zone u (u pour utilisateur)

La table des processus permet d'accéder au code, aux données et aux piles d'un processus. Chacun de ses éléments décrit l'état d'un processus actif dans le système.

La zone u contient les données privées manipulées uniquement par le noyau :

- le pointeur sur l'élément de la table des processus du processus courant en exécution
- les paramètres de l'appel système courant, la valeur de retour et les codes d'erreur
- les descripteurs de fichier de tous les fichiers ouverts
- les paramètres internes d'entrées-sorties
- le répertoire courant et la racine courante
- les limites quant à la taille des fichiers et des processus

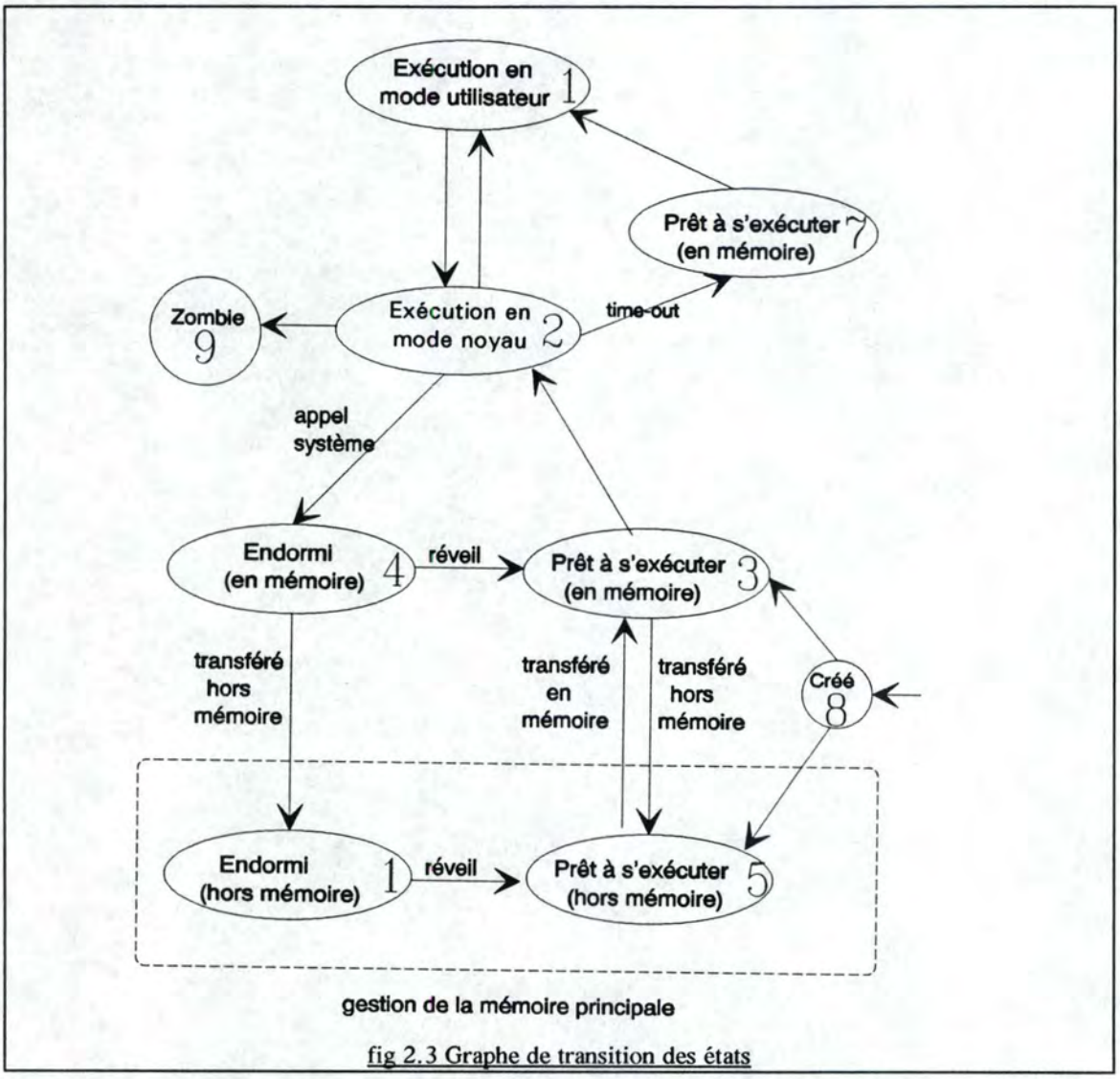


fig 2.3 Graphe de transition des états

Liste des états possibles d'un processus

Comme il a été dit précédemment, la durée de vie d'un processus peut être conceptuellement divisée en un ensemble d'états qui décrivent ce même processus; la liste suivante contient l'ensemble complet des états possibles d'un processus.

1. Le processus s'exécute en mode utilisateur.
2. Le processus s'exécute en mode noyau.
3. Le processus est prêt à s'exécuter dès que le noyau l'élira.
4. Le processus est endormi et réside en mémoire principale.

5. Le processus est prêt à s'exécuter mais il doit être transféré en mémoire principale avant que le noyau ne puisse l'élire en vue de son exécution.
6. Le processus est endormi et a été transféré en mémoire secondaire pour obtenir de la place en mémoire principale pour les autres processus.
7. Le processus est passé du mode noyau au mode utilisateur mais le noyau l'a interrompu et a effectué un changement de contexte pour élire un autre processus.
8. Le processus est nouvellement créé et se trouve dans un état de transition: le processus existe, mais n'est pas prêt en vue de son exécution et n'est pas endormi.
9. Le processus a exécuté l'appel système exit (fin de son exécution) et est dans l'état zombie. Le processus n'existe plus mais il laisse un enregistrement contenant un code de retour et quelques statistiques de temps.

Les éléments représentés sont artificiels dans le fait que les processus ne les éprouvent pas toujours mais ils illustrent diverses transitions d'état.

Transitions d'état

--> 8 création d'un processus
1 --> 2 appel système ou interruption
2 --> 1 fin d'un appel système ou d'une interruption
2 --> 4 appel système pour demande d'une ressource non disponible
2 --> 7 fin de l'interruption time out (fin de sa tranche de temps)
2 --> 9 fin de l'exécution d'un processus
3 --> 2 processus élu (pour exécution en mode noyau)
3 --> 5 processus transféré hors mémoire car mémoire insuffisante
4 --> 3 ressource disponible
4 --> 6 processus transféré hors mémoire car mémoire insuffisante
5 --> 3 processus transféré en mémoire car mémoire suffisante
6 --> 5 ressource disponible
7 --> 1 processus élu (pour exécution en mode utilisateur)
8 --> 3 mémoire suffisante
8 -> 5 mémoire insuffisante

Contexte d'un processus

A chaque processus est associé un contexte. Le contexte d'un processus est son état; il est défini par :

* son code
* les valeurs de ses variables globales utilisateur et les structures de données
* les valeurs des registres machine qu'il utilise
* les valeurs rangées dans son élément de la table des processus et dans sa zone u
* le contenu de ses piles utilisateur et noyau.

En exécutant un processus, le système est dit "s'exécutant dans le contexte du processus". Quand le noyau décide qu'il doit exécuter un autre processus, il fait un changement de contexte de telle sorte que le système s'exécute dans le contexte de l'autre processus.

En faisant un changement de contexte, le noyau sauvegarde assez d'informations pour qu'il puisse plus tard revenir sur le premier processus et reprendre son exécution.

Le noyau permet un changement de contexte dans quatre circonstances :

* quand un processus se met lui-même en sommeil
* quand il se termine
* quand il revient d'un appel système (mode noyau) au mode utilisateur alors qu'il n'est pas le plus éligible en vue d'une exécution
* quand il revient au mode utilisateur après que le noyau ait fini de traiter une interruption alors qu'il n'est pas le processus le plus éligible en vue d'une exécution

Le noyau s'assure de l'intégrité et de la cohérence des structures de données internes en prohibant des changements de contexte arbitraires. Le contexte d'un processus est contrôlé via l'implémentation des appels systèmes :

fork : création d'un nouveau processus
exit : fin de l'exécution d'un processus
wait : synchronisation entre deux processus
.....

Les files d'attente

La possibilité d'avoir plusieurs processus actifs et des changements fréquents d'état nécessite l'utilisation de files d'attente. Il en existe de multiples sous Unix. Leur concept est directement associé à celui des priorités. En général, on distingue deux classes de priorité : les priorités utilisateur (en mode utilisateur) et les priorités noyau (en mode noyau).

Chaque classe contient plusieurs valeurs (niveaux) de priorité à chacune desquelles une file de processus est logiquement associée.

Ordonnancement des processus

L'ordonnancement des processus joue incontestablement un rôle très important dans tout système d'exploitation. En effet, les performances de celui-ci sont en grande partie déterminées par la manière d'ordonnancer les processus. L'ordonnanceur dans le système Unix est du type " round robin à plusieurs niveaux de réaction".

Le noyau alloue l'unité centrale à un processus pour une période de temps appelée "tranche de temps" ou "quantum de temps". Dès que le processus dépasse sa tranche de temps, il est interrompu et réintroduit dans l'une des multiples files de priorité. Le processus en attente (dans l'état 'prêt à s'exécuter, chargé en mémoire') avec la plus haute priorité est alors élu. Si un processus est bloqué ou a terminé son exécution avant la fin de son quantum, on libère immédiatement le processeur.

Gestion du temps

Le système gère le temps à l'aide d'une horloge matérielle qui interrompt l'unité centrale à une valeur fixée, une vitesse qui dépend de la partie matérielle, comprise habituellement entre 50 et 100 fois par seconde. Lors de chaque occurrence d'une interruption horloge, appelée "top d'horloge", le système d'exploitation vérifie si le processus en cours d'exécution a utilisé tout son temps CPU (tranche de temps) de manière à l'interrompre si c'est le cas.

Le choix de la tranche de temps est très important. Trop courte, elle provoquerait de nombreux changements de processus et diminuerait l'efficacité du CPU tandis que trop longue, elle causerait un mauvais

temps de réponse aux petites requêtes interactives. Une tranche de temps aux alentours de 100 msec est souvent un compromis raisonnable [TAN87].

Les priorités

L'ordonnanceur doit aussi tenir compte des priorités. En effet, chaque processus possède une priorité dans chacun des deux modes. Celle du niveau noyau est calculée en mettant en corrélation une valeur de priorité fixée avec la raison du sommeil. Plus le processus peut libérer de ressources à la fin de son exécution, plus il a une haute priorité. Celle du niveau utilisateur est fonction de la récente utilisation de l'unité centrale (CPU) par le processus : s'il l'a utilisée récemment, il a une priorité plus basse.

Chaque fois qu'un processus en cours d'exécution est sur le point de s'endormir, le noyau recalcule sa priorité. De plus, il ajuste la priorité de tout processus qui passe du mode noyau au mode utilisateur de manière à être certain qu'il ait une priorité au niveau utilisateur.
Toutes les secondes, le noyau réajuste les priorités de tous les processus en mode utilisateur.

A chaque top d'horloge (interruption de l'horloge), le champ (*UC*) dans la table des processus qui contient l'utilisation récente de l'unité centrale par le processus en cours d'exécution est incrémenté.
Une fois par seconde, ce champ est recalculé pour chaque processus selon une fonction d'extinction :

$$\text{extinction (UC)} = UC/2$$

Au même moment, la priorité des processus dans l'état "interrompu mais prêt à s'exécuter" (mode utilisateur) est également recalculée selon la formule :

$$\text{priorité} = (UC / 2) + \text{priorité initiale}$$

De cette manière, les processus en mode utilisateur se déplacent dans les files d'attente de façon à avoir accès au processeur chacun à leur tour. Le noyau ne change pas la priorité des processus du mode noyau.

Gestion de la mémoire

Le noyau réside en permanence en mémoire principale comme tout processus (ou au moins une partie de celui-ci) en exécution. Quand le programme est en exécution sur la machine, le noyau (le programme lui-même) lui attribue de l'espace en mémoire principale.

Dans tout système distribué, une partie seulement de chaque programme en cours d'exécution se trouve en mémoire centrale. Le reste du programme se trouve dans une mémoire secondaire (en général le disque). Ceci nécessite une gestion complexe de la mémoire. A ce niveau on peut rencontrer trois politiques différentes: le swapping, la pagination et un système hybride avec swapping et pagination.

2.5.1.3) Le concept de fichier

Le système de fichiers

Le système de fichiers UNIX est caractérisé par :

- une structure hiérarchique
- un traitement cohérent des données des fichiers
- une croissance dynamique des fichiers
- une protection des fichiers de données
- un traitement des périphériques en tant que fichiers

Les fichiers

Une des particularités du système UNIX réside dans l'utilisation des fichiers. En effet, toutes les informations nécessaires au système sont sauvegardées dans des fichiers (périphériques, mots de passe, etc...). Tout fichier d'un système Unix a un unique i_noeud. Celui-ci contient les informations nécessaires à un processus pour accéder au fichier, telles que le propriétaire du fichier, les droits d'accès, la taille du fichier, et la localisation des données du fichier dans le système de fichiers. Les processus accèdent aux fichiers par un ensemble bien défini d'appels systèmes. Ils doivent spécifier un fichier par une chaîne de caractères qui constitue le chemin d'accès. Chaque chemin d'accès détermine de manière unique un fichier, et est converti en l'i_noeud du fichier par le noyau.

Les fichiers sous Unix se trouvent dans des périphériques de stockage de masse tels que les disques. Quand un processus cherche à accéder aux données d'un fichier, le noyau ramène les données en mémoire principale où le processus peut les traiter. Il tente de minimiser la fréquence d'accès au disque en gérant une réserve de tampons en données internes, appelée buffer cache, qui contient les données des blocs disque les plus récemment utilisés.

2.5.2) NFS[1]

2.5.2.1) Introduction

Les procédures RPC (remote procedure call) permettent à un processus (appelé processus appelant) de lancer d'autres processus se trouvant sur des machines éloignées. NFS est un service RPC qui permet de partager des fichiers dans un environnement hétérogène de processus, de systèmes d'exploitation et de réseaux.

Il permet à un utilisateur d'accéder à des fichiers et des hiérarchies de fichiers éloignés (sur d'autres machines) de manière transparente comme s'ils étaient locaux à sa machine. Pour celui-ci, tous les fichiers accédés à travers NFS apparaissent comme des fichiers locaux.

Il n'y a pas de différence apparente entre lire et écrire un fichier sur un disque local et un fichier sur un disque d'une autre machine.

2.5.2.2) Avantages

NFS permet à des machines multiples d'utiliser les mêmes fichiers. Les données peuvent être accessibles par n'importe quelle machine sur le réseau. De plus, les coûts de stockage peuvent être réduits lorsque plusieurs machines se partagent de grands programmes. Il diminue

[1] Nous nous sommes inspirés de [SUN91] pour rédiger la description du système de fichiers distribués NFS.

également le risque d'avoir plusieurs copies d'un même fichier réparties sur le réseau.

2.5.2.3) Description du système NFS

Relation client-serveur

Un serveur est une machine qui partage au moins une de ses hiérarchies de fichiers avec d'autres machines. Un client est une machine qui accède au moins à une hiérarchie de fichiers d'un serveur.
On dit d'un serveur qu'il "exporte" un système de fichiers et des répertoires, d'un client, qu'il "monte" un système de fichiers et des répertoires.

"Exporter" un système de fichiers ou un répertoire consiste à autoriser des clients NFS à pouvoir les "monter" sur leur système local. On peut parler d'architecture Client-Serveur puisque les systèmes clients requièrent des ressources fournies par d'autres systèmes, les serveurs.
Il n'y a pas une relation 1-1 entre les serveurs et les clients. En effet, un système peut à la fois être un serveur et un client. Un serveur qui "exporte" un système de fichiers et des répertoires peut également monter des systèmes de fichiers et des répertoires exportés par d'autres systèmes et ainsi devenir client.

Fonctionnement

Le système doit être mis en mode multi-utilisateurs. NFS est basé sur trois fichiers :

1. /ETC/EXPORTS
2. /ETC/FSTAB
3. /ETC/RC.LOCAL

1./ETC/EXPORTS contient la liste des systèmes de fichiers et des répertoires à "exporter".
2./ETC/FSTAB contient la liste des systèmes de fichiers et des répertoires à "monter" lors du lancement du système ou lors du passage en mode multi-utilisateurs.

3./ETC/RC.LOCAL contient, entre autres, la liste des démons que l'on veut lancer automatiquement chaque fois que le système est placé en mode multi-utilisateurs.

C'est la tâche de l'administrateur du réseau de gérer ces fichiers qui sont primordiaux pour le bon fonctionnement du système. Le serveur NFS est contrôlé par le programme système **EXPORTFS** et les démons **MOUNTD** et **NFSD**.
Le programme **EXPORTS** consulte le fichier /ETC/EXPORTS et informe le noyau du serveur des permissions applicables sur chaque hiérarchie de fichiers exportée.
Le processus de montage utilise une série de **RPC** pour permettre au client d'accéder aux répertoires du serveur de manière transparente. A chaque requête d'un client, le démon **MOUNTD** du serveur reçoit des informations du client qu'il traite. En fonction de l'analyse des données reçues et de celles consultées dans le fichier \ETC\XTAB, il donne ou non la permission au client de monter la hiérarchie. Si la réponse est positive, il renvoie un pointeur au client. L'accès au point de montage ou à un répertoire plus bas se fait avec le pointeur envoyé au démon **NFSD** du serveur.

Installation et maintenance

C'est l'administrateur du serveur qui doit s'occuper de l'installation de NFS. Il doit connaître un certain nombre de paramètres :

-si le système va agir comme un serveur : les permissions d'accès, les chemins (pathnames) des systèmes de fichiers et des répertoires,...
-si le système va agir comme un client : les noms des serveurs qui contiennent les systèmes de fichiers et les répertoires qu'on veut importer, les chemins (pathnames),...

Exportation

Un serveur ne peut exporter que ses systèmes de fichiers locaux et ses répertoires locaux.

- Il est possible pour un serveur d'exporter une série de hiérarchies de fichiers de manière automatique lors du lancement du système. Ceux-ci seront alors exportés en permanence.
Il faut utiliser le fichier /ETC/EXPORTS qui contient une entrée par hiérarchie à exporter.

Chaque entrée a la forme suivante :

Répertoire (chemin) Options : ro/rw (permissions d'accès) root = hostnames (clients qui peuvent travailler en mode superuser) access = client1:client2... client (liste des clients qui peuvent monter) secure (indique un protocole plus sûr)

Par l'intermédiaire de celui-ci, le serveur contrôle la liste des clients qui peuvent monter les hiérarchies (exportées par ce serveur) ainsi que leurs permissions respectives.

Quand on modifie le contenu de ce fichier, if faut relancer le système ou utiliser la commande EXPORTFS pour mettre à jour les informations dans le noyau du serveur.
- Si le serveur désire exporter des hiérarchies un nombre fini de fois puis supprimer cette exportation, il doit utiliser la commande EXPORTFS. La syntaxe est la suivante :

/USR/ETC/EXPORTFS	-a (exporte tous les répertoires listés dans le fichier /ETC/EXPORTS) -u répertoire (suppression) -o options (idem options du fichier /ETC/EXPORTS)

Cette commande permet également de modifier les caractéristiques de répertoires déjà exportés. Utilisée sans paramètre, elle donne la liste des hiérarchies exportées à tout moment par le serveur.

Il faut avoir les privilèges du superuser pour pouvoir exécuter cette commande.

- Chaque fois que la commande **EXPORTFS** est lancée, le fichier /ETC/XTAB est modifié mais pas le fichier /ETC/EXPORTS. Donc, seul le fichier /ETC/XTAB contient la liste complète des hiérarchies exportées à tout moment. Son format est le même que celui du fichier /ETC/EXPORTS.

Rem : on ne peut pas exporter un répertoire qui se trouve dans une hiérarchie déjà exportée. (fin de remarque)

Montage

La commande **MOUNT** permet de monter une hiérarchie éloignée. Elle exige que le client puisse atteindre le serveur de la hiérarchie à travers le réseau et que le serveur exporte la hiérarchie vers le client en question.

Il faut créer un point de montage sur le système client pour chaque hiérarchie de fichiers à monter avec la commande :

MKDIR répertoire.

Tout répertoire peut servir de point de montage.

Une fois le montage effectué, le client peut accéder à la hiérarchie comme à un répertoire local avec la commande :

CD répertoire.

Il existe deux options possibles :

. option soft : après un nombre fini d'essais négatifs, l'opération échoue
. option hard : les essais continuent jusqu'au succès de l'opération ou à l'arrêt du job

Rem : le montage lui-même est provoqué par le client. (fin de remarque)

- Il est possible pour un client de monter une série de hiérarchies de fichiers de manière automatique lors du lancement du système. Ceux-ci seront alors montés en permanence.

Il faut utiliser le fichier /ETC/FSTAB qui contient une entrée par hiérarchie à monter. Chaque entrée a la forme suivante :

Répertoire (hiérarchie à monter = serveur + chemin) Répertoire (point de montage) Type (4.2 ou NFS) Options (ro/rw) freq (fréquence des dump)

Quand on modifie le contenu de ce fichier, il faut relancer le système ou utiliser la commande **MOUNT**.

- Si le client désire monter des hiérarchies un nombre fini de fois puis les démonter, il doit utiliser la commande **MOUNT** et **UMOUNT**. La syntaxe est la suivante :

MOUNT -a (monte tous les répertoires listés dans le fichier /ETC/FSTAB) -t type -o options serveur:chemin point de montage

La commande **UMOUNT** permet de démonter une hiérarchie de fichiers. La syntaxe est la suivante :

UMOUNT point de montage ou serveur:chemin

Utilisée sans paramètre, elle donne la liste des hiérarchies montées par le client à tout moment.

Il faut avoir les privilèges superuser pour pouvoir utiliser les commandes **MOUNT** et **UMOUNT**.

Chaque fois que la commande **MOUNT** ou **UMOUNT** est lancée, le fichier /ETC MTAB est modifié mais pas le fichier /ETC/FSTAB.

Donc, seul le fichier /ETC/MTAB contient la liste complète des hiérarchies montées à tout moment. Son format est le même que celui du fichier /ETC/FSTAB.

Initialisation d'un serveur

* identifier la machine comme un serveur de fichiers NFS
* définir le ou les disques du serveur indiquant les partitions à exporter
* définir les paramètres pour chaque client du serveur
* créer le fichier /ETC/EXPORTS

Initialisation d'un client

* déclarer la machine client
* modifier le fichier /ETC/FSTAB
* créer les points de montage dans l'arbre des répertoires pour monter les hiérarchies listées dans le fichier FSTAB.

Accès "Super User" dans le réseau

Les gestionnaires des systèmes clients font partie de la catégorie "other" sur le serveur. Ils n'ont pas de permissions sur les systèmes de fichiers montés sauf si ceux-ci ont des permissions de lecture et d'écriture pour le groupe d'utilisateurs "other".
Il est possible de permettre sur un serveur des accès à des clients par l'intermédiaire du fichier /ETC/EXPORTS.

Chapitre 3 :
Supports statistiques de la simulation

3.1) Introduction

Dans toute simulation, le nombre aléatoire est une composante essentielle. Pour notre programme, nous employons le générateur qui est fourni avec le Borland C++. Avant de l'utiliser, nous avons effectué des tests afin de nous assurer du caractère aléatoire de la suite de nombres générés. Ces tests sont développés dans le paragraphe suivant.

La simulation doit nous donner des renseignements tels le taux d'utilisation des différents serveurs, le temps d'attente moyen dans les files, le nombre moyen de processus en attente ou encore le temps de réponse moyen. Ces valeurs correspondent à une estimation de la réalité. En effet, si l'on effectue plusieurs simulations, il est très peu probable d'obtenir deux fois un même résultat. Le calcul de ces estimateurs est différent selon que le système se trouve en régime transitoire ou stationnaire.

En régime transitoire, les variables dépendent du temps. Il est donc nécessaire d'effectuer plusieurs simulations. Pour chacune d'entre elles, il convient d'observer les variables à un moment bien précis et de calculer la moyenne ainsi que la variance de ces observations. Les analyses statistiques à effectuer lors d'une simulation en régime transitoire sont expliquées au point 3.3.

Lorsque le système se trouve en régime stationnaire, le théorème de Doob-Birkhov[1] nous permet d'alléger fortement la procédure d'estimation. En effet, une seule simulation suffit pour effectuer les calculs. Il faut cependant être certain que le système se trouve bien en

[1] Ce théorème est énoncé en Annexe A.

régime stationnaire. Pour ce faire, il existe plusieurs méthodes. Ce problème est abordé au point 3.4.1. La procédure d'estimation, quant à elle, est développée au point 3.4.2.

3.2 Le générateur pseudo-aléatoire

Certains programmes ont besoin d'une grande quantité de nombres aléatoires. C'est pourquoi la plupart des langages actuels fournissent un générateur de nombres pseudo-aléatoires. Afin de nous assurer du caractère aléatoire de la séquence, nous devons vérifier que les nombres générés respectent certaines propriétés. Un générateur de nombres au hasard doit, pour être intéressant, présenter les caractéristiques suivantes [NOI91] :

1. la procédure doit être simple et de courte durée;
2. la suite de nombres générés doit présenter une période très longue;
3. elle doit être reproduite plusieurs fois, afin par exemple, de comparer plusieurs politiques ou d'exercer un contrôle sur le calcul;
4. elle doit être conforme du point de vue statistique, c'est-à-dire que non seulement la distribution des nombres générés doit respecter la distribution désirée mais encore faut-il que ces nombres présentent toutes les caractéristiques des variables aléatoires indépendantes.

Avant d'utiliser le générateur proposé par Borland, il est prudent de vérifier ses propriétés statistiques [P&M88]. C'est dans ce but que nous avons effectué deux tests sur la suite des nombres issus du générateur du langage C. Le premier contrôle l'uniformité des nombres; il utilise le test classique du Khi Carré. Le second teste l'existence d'une corrélation sérielle. Cela consiste à vérifier s'il y a une corrélation entre les x_i et x_{i+k} en faisant varier k.

Le test d'uniformité

Soit n le nombre d'observations. Considérons s classes équidistantes. Le test d'uniformité est réussi si la variable :

$$D^2 = \frac{\sum_{i=1}^{N} (R_{i_{obs}} - R_{i_{th}})^2}{R_{i_{th}}}$$

$$\text{avec } R_{i_{th}} = \frac{n}{s}$$

respecte la distribution de χ^2 avec s-1 degrés de liberté. La zone de rejet est obtenue par :

$$\{ \omega : D^2 > Q_{\chi^2_{s-1}} (1 - \alpha) \}$$

au niveau d'incertitude α.

Le test de corrélation

Le test de corrélation consiste à vérifier qu'il n'y a pas de corrélation sérielle [NOI91] c'est-à-dire à tester l'hypothèse nulle

$$\rho_k = \frac{\text{cov}(x_n, x_{n+k})}{\text{var } x_n} = 0$$

On considère généralement l'estimateur

$$r_k = \frac{\sum_{i=1}^{N-k} (x_i - \bar{x})(x_{i+k} - \bar{x})}{\sum_{i=1}^{N-k} (x_i - \bar{x})^2}$$

où $\bar{x}$ est la moyenne des N premières observations.

Si l'hypothèse nulle est vraie, la variable

$$t_k = \sqrt{\frac{r_k^2 (N-2-k)}{1 - r_k^2}} = r_k \sqrt{\frac{N-2-k}{1 - r_k^2}}$$

a une distribution t de Student à N-2-K degrés de liberté. La zone de rejet (région critique) de l'hypothèse nulle peut s'écrire :

$$\{ \omega : \left| t_k \right| > Q_{t(N-2-K)} (1 - \frac{\alpha}{2}) \}$$ au niveau d'incertitude α.

Le générateur du Borland C++

Borland propose un générateur congruentiel du type :

$$X_{n+1} = A \quad X_n \quad + \quad C \quad (\text{mod } M)$$

Les nombres produits par cette formule sont compris entre 0 et M. Le choix de la racine x_0 détermine la suite de nombres que l'on va obtenir. La qualité du générateur dépend de la valeur affectée aux paramètres A,C et M. Nous avons recherché les valeurs que Borland a attribuées à ces paramètres. Pour cela, nous avons dû employer le 'debugger' fourni avec le Borland C++. Le Turbo Debugger permet en

effet de consulter le code assembleur du programme ainsi que tous les registres du système d'exploitation. Les variables A et C étant des constantes de la fonction RAND(), il nous a suffit de découvrir dans quels registres elles étaient stockées. Nous avons pu en déduire la formule suivante :

$$X_{n+1} = 22695477 * X_n + 1 \mod 2^{31} - 1$$

La méthode utilisée pour ramener le nombre dans l'intervalle demandé par l'utilisateur (paramètre de la fonction RAND()) consiste à diviser le nombre obtenu par M et à le multiplier par le paramètre.

La formule finale devient donc :

$$Xn+1 = (((A * Xn + C) \mod M) / M) * RANGE$$

avec

$M = 2^{31} - 1$

$A = 22695477$

$C = 1$

RANGE compris entre 0 et $2^{15} - 1$

Résultats des tests

Nous avons testé le générateur pour les cas N = 100, NB_CL = 10 et N = 1000, NB_CL = 50 où N est le nombre d'observations et NB_CL le nombre de classes. Pour chacun des deux cas, nous faisons les tests avec dix racines différentes prises au hasard. Nous disposons donc de dix mesures indépendantes (une mesure par racine).

Les résultats pertinents sont le minimum, le maximum, la moyenne et le pourcentage de refus. Pour chacun de ces résultats, nous calculons l'écart avec les deux valeurs théoriques (α = 0,01 et α = 0,05). Un écart négatif signifie que la valeur observée est inférieure à la valeur théorique (test accepté).

Test du Khi carré

* N = 100 et NB_CL = 10

Khi carré théorique (α=0,01) = 21,66
Khi carré théorique (α=0,05) = 16,92

		écart entre χ^2 obs. et χ^2 théor. α=0,01	α=0,05
Khi carré minimum	4,80	-16,86	-12,12
Khi carré maximum	19,40	-2,26	2,48
Khi carré moyen	10,98	-10,68	-5,94
Pourcentage de refus (α = 0.01)	0%		
Pourcentage de refus (α = 0.05)	10%		

* N = 1000 et NB_CL = 50

Khi carré théorique (α=0,01) = 74,13
Khi carré théorique (α=0,05) = 66,05

		écart entre χ^2 obs. et χ^2 théor. α=0,01	α=0,05
Khi carré minimum	29,6	-44,53	-36,45
Khi carré maximum	72,6	-1,53	6,55
Khi carré moyen	46,9	-27,23	-19,15
Pourcentage de refus (α = 0.01)	0%		
Pourcentage de refus (α = 0.05)	10%		

Test de corrélation

Les tableaux ci-dessous résument les résultats observés :

* Pour N = 100

	MIN	MAX	MOY
T(1)	0,09	1,85	0,807
T(2)	0,05	1,56	0,863
T(3)	0,24	1,56	0,651
T(4)	0,03	1,82	0,733
T(5)	0,12	2,38	1,195

* Pour N = 1000

	MIN	MAX	MOY
T(1)	0,07	2,59	1,291
T(2)	0,13	1,38	0,677
T(3)	0,02	1,47	0,739
T(4)	0,03	0,94	0,443
T(5)	0,02	1,72	0,491

Conclusions des tests

Pour les deux cas analysés, le test du khi carré est accepté pour neuf des dix racines. On peut donc admettre que le générateur du Borland C++ passe le premier test avec succès. Il en va de même pour le test de corrélation. En effet, le taux de refus est très faible. Aucune moyenne n'est refusée. Il s'avère donc que nous pouvons utiliser le générateur du Borland C++ pour notre programme de simulation.

3.3 Le régime transitoire

Lorsqu'un système se trouve en régime transitoire, il faut effectuer N simulations indépendantes et prendre les mesures à un moment bien précis. Ceci correspond à un échantillonnage statistique. Nous disposons donc d'un ensemble de variables observées à un temps t de chacune des N simulations ($n_1(t)$, $n_2(t)$,..., $n_N(t)$). Ces valeurs correspondent à N réalisations d'une variable aléatoire. L'estimateur de cette variable est obtenu par :

$$\bar{n}(t) = \frac{\sum_{i=1}^{N} n_i(t)}{N}$$

L'intervalle de confiance pour cette moyenne est calculé en utilisant la méthode habituelle des échantillons nombreux :

$$\bar{n}(t) - \frac{s}{\sqrt{N}} Q_G(1 - \frac{\alpha}{2}) \leq \xi \underline{n}(t) \leq \bar{n}(t) + \frac{s}{\sqrt{N}} Q_G(1 - \frac{\alpha}{2}) \qquad \text{spr}\alpha$$

avec α le niveau d'incertitude
s l'écart-type de l'échantillon

$Q_G(1 - \frac{\alpha}{2})$ le quantile gaussien en $1 - \frac{\alpha}{2}$

$$s^2 = \frac{1}{N} \sum_{i=1}^{N} n_i^2(t) - \bar{n}(t)^2$$

3.4) Le régime stationnaire

Lorsque son état initial est vide (variables initialisées à zéro), tout système de file d'attente non-déterministe avec un flux d'arrivées aléatoire et/ou des temps de service aléatoires est dans une phase transitoire. Ce n'est qu'après une certaine période que le système atteint son régime

stationnaire (fig 3.1). A partir de ce moment seulement le système se trouve dans les conditions d'applicabilité du théorème de Doob-Birkhov. Les observations recueillies lors de la période initiale ne peuvent être prises en considération. Notons également que le régime stationnaire peut ne jamais être atteint lorsque L > c μs où L est le taux d'arrivée et $1/\mu s$ le taux de service par serveur; c étant le nombre de serveurs.

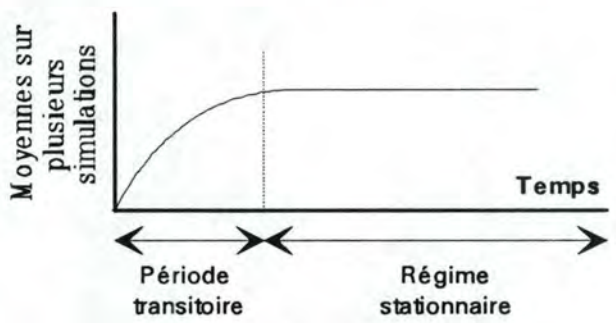


Fig. 3.1 Période transitoire et régime stationnaire d'une simulation

3.4.1) Détection de la période transitoire

Des recherches ont été faites afin de réduire la période initiale. La solution proposée par Conway [CON63] consiste à fabriquer un état initial qui se rapproche au maximum d'un état en régime stationnaire. Depuis lors, il y a eu beaucoup de tentatives pour déterminer les conditions initiales optimales qui permettent de limiter le plus possible l'influence de la période transitoire sur les résultats en régime stationnaire. Les conclusions sont malheureusement ambiguës.

Une autre solution consiste à déterminer la longueur de la période transitoire et ainsi découvrir à partir de quel moment les données observées appartiennent au régime stationnaire. Beaucoup d'études ont été réalisées dans cette direction et plusieurs règles empiriques ont été énoncées. Nous appliquons une de ces règles dans notre programme. Elle est proposée par Fishman [FIS73] et affirme que :

La période initiale transitoire est terminée après n_0 observations si la série d'observations séquentielles dans le temps $x_1, x_2, \ldots x_n$ traverse la moyenne $\overline{X}(n_0)$ k fois.

La règle (fig. 3.2) est tributaire de la valeur affectée à k. A une valeur trop grande correspond une surestimation de la période transitoire, par contre, pour une valeur trop petite, la période initiale est sous-estimée (principalement pour les systèmes chargés). Des recherches ont été réalisées et il semble que 25 soit une valeur satisfaisante pour k.

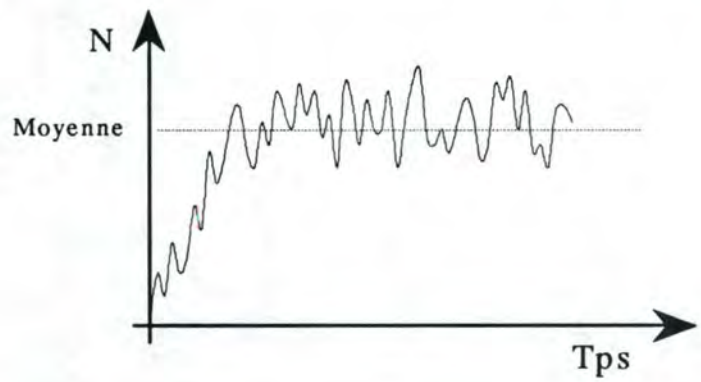


fig. 3.2 : Règle permettant de déterminer la période transitoire

Lorsque la série d'observations a traversé 25 fois la moyenne, nous pouvons *supposer* que nous avons atteint la période stationnaire. Il faut cependant *s'en assurer.* C'est pourquoi la détection de la période transitoire se fait en deux étapes. La première consiste à effectuer des mesures jusqu'à ce que la série d'observations traverse 25 fois la moyenne. La seconde vérifie si le système se trouve bien en régime stationnaire. Le test de stationnarité pose cependant un problème. En effet, pour effectuer ce test sur un ensemble de valeurs (considérons les n_t dernières observations), il faut connaître l'estimateur de la variance $\sigma^2(\overline{X}(n_t))$ et le nombre de degrés de liberté K pour la distribution de Khi Carré avant de pouvoir déterminer si le système se trouve bien en régime

stationnaire. Pour obtenir un bon estimateur de la variance, l'estimation doit être faite sur un sous-ensemble (n_v) des n_t dernières observations. Il est souhaitable que

$$n_t \geq \gamma_v * n_v$$

où γ_v ($\gamma_v \geq 2$) est le coefficient de "sécurité" pour que l'estimateur de la variance représente l'état stationnaire. Heildeberger et Welch [H&W83] ont proposé $n_v \geq 100$ et $\gamma_v = 2$.

Il existe plusieurs méthodes pour estimer la variance. Nous utilisons celle des blocs indépendants décrite par Pawlikowski dans [PAW90]. Cette technique consiste à décomposer la suite des n_t observations (x_i, $x_{i+1},\ldots,x_{i+n_t}$) en une série de k_b blocs disjoints (x_{11}, $x_{12},\ldots,x_{1m}$), (x_{21}, $x_{22},\ldots,x_{2m}$),..... de taille m ($k_b = n_t/m$). Nous obtenons la suite $\overline{X}_1(m)$, $\overline{X}_2(m)$,...., $\overline{X}_{K_b}(m)$ des moyennes des blocs par :

$$\bar{x}_i(m) = \frac{1}{m}\sum_{j=1}^{m} x_{ij}$$

Nous estimons la moyenne par :

$$\overline{\overline{X}}(k_b, m) = \frac{1}{k_b}\sum_{i=1}^{k_b} \overline{X}_i(m)$$

L'estimateur de la variance s'obtient par :

$$\hat{\sigma}^2\ (\overline{X}(k_b, m)) = \sum_{i=1}^{k_b} \frac{(\overline{X}_i(m) - \overline{X}(k_b,m))^2}{k_b(k_b-1)}$$

Nous pouvons admettre que

$$T = \frac{\sqrt{45}}{n_t^{1,5}\ n_v^{0,5}\ \hat{\sigma}(\bar{x}(n_v))} \sum_{k=1}^{n_t} k\ (1-\frac{k}{n_t})\ (\bar{x}(n_t) - \bar{x}(k))$$

où

$$\bar{x}(i) = \sum_{j=n_0+1}^{n_0+i} \frac{x_j}{i}$$

est approximativement une variable aléatoire t de Student à (n_t/m)-1 degrés de liberté [SCH83].

Le régime stationnaire est atteint à $(1 - \alpha) * 100$ % de certitude lorsque :

$$T \leq t_{k,1-\alpha}$$

où $t_{k,1-\alpha}$ est le quantile de la distribution de Student
avec k degrés de liberté et un degré de certitude $1-\alpha$

La figure 3.3 représente la démarche à effectuer pour déterminer la période transitoire. Ce n'est que lorsque cette étape est franchie que les observations peuvent être prises en considération pour réaliser le traitement statistique en régime stationnaire.

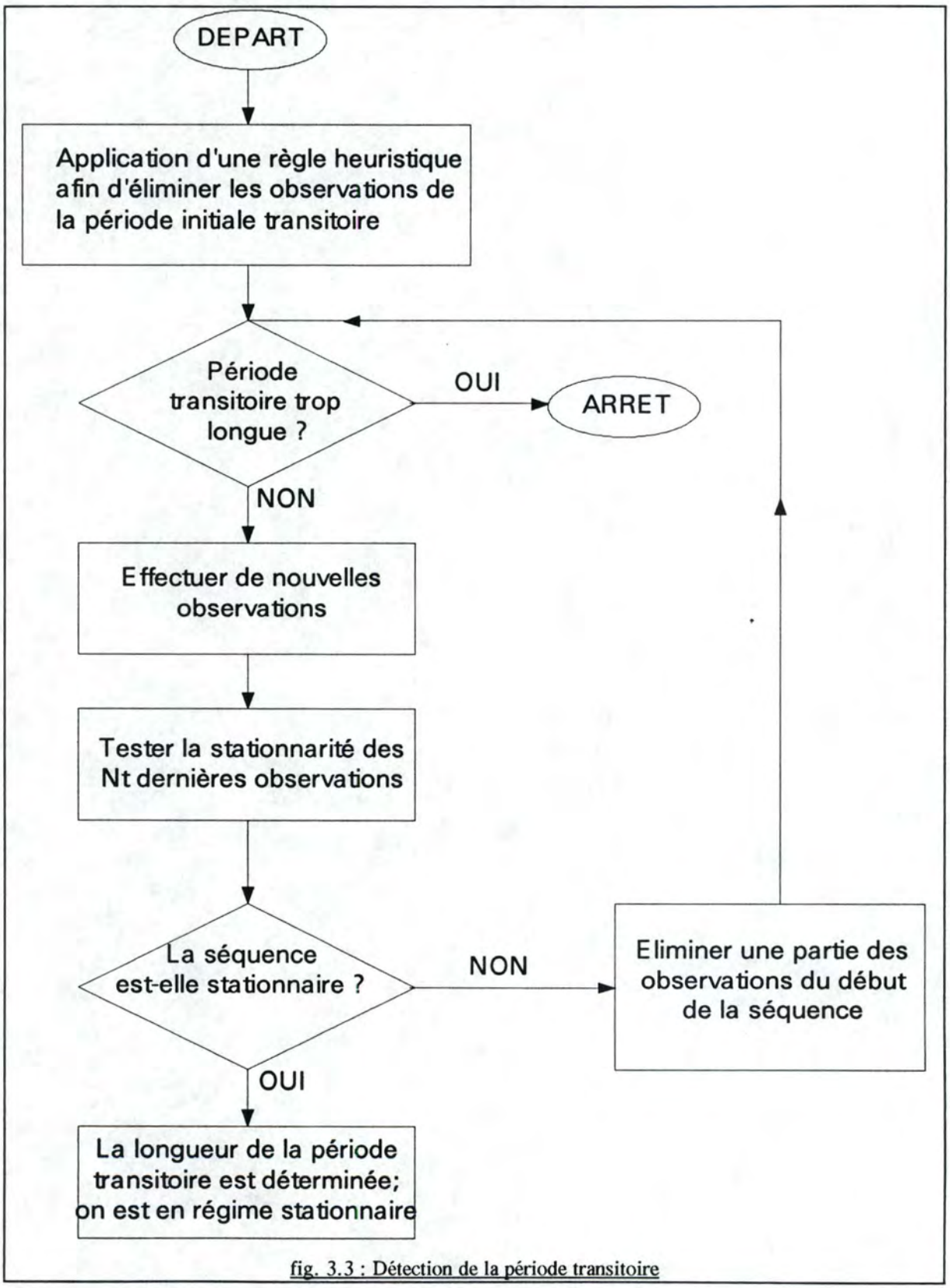


fig. 3.3 : Détection de la période transitoire

3.4.2) Analyses en régime stationnaire

Lorsque nous sommes "certains" que le système se trouve en régime stationnaire, nous pouvons commencer les prises de mesures. Il faut cependant que les valeurs recueillies soient indépendantes. Plusieurs techniques permettent de s'assurer de l'indépendance des observations. Nous utilisons dans notre programme la méthode "par blocs" proposée par Fisher [NOI91]. Elle consiste à décomposer la suite des n observations en une série de K blocs disjoints de taille m. Afin de s'assurer de l'indépendance des données, après chaque bloc, p observations sont ignorées (fig 3.4). Cette approche est basée sur l'hypothèse que : " au plus les observations sont séparées dans le temps, au moins elles sont corrélées".

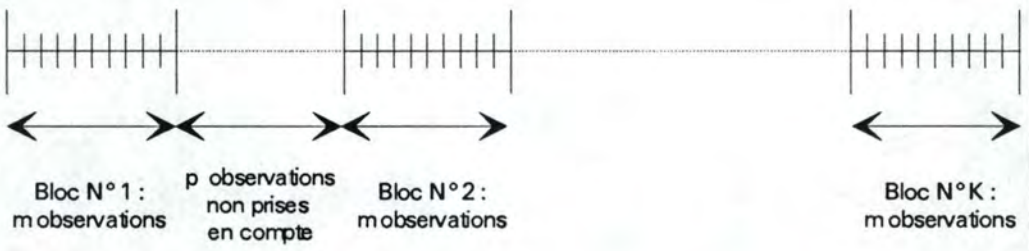


fig. 3.4 : Prise de mesures selon la méthode des blocs indépendants.

Nous pouvons ainsi calculer la moyenne de chaque bloc i par l'équation :

$$\bar{x}_i(m) = \frac{1}{m}\sum_{j=1}^{m} x_{ij}$$

L'estimateur de la moyenne est obtenu par l'équation :

$$\bar{X} = \frac{1}{K}\sum_{i=1}^{K} \bar{X}_i(m)$$

Soit :

$$V^2 = \frac{1}{K-1}\left(\sum_j \bar{x}^2(j) - k\,\bar{\bar{x}}^2\right)$$

On suppose en général que

$$\frac{\bar{\bar{x}} - \xi x}{\frac{v}{\sqrt{K}}}$$

est approximativement une variable aléatoire t de Student à K-1 degrés de libertés.

Si bien que l'intervalle de confiance est fourni par :

$$\bar{\bar{x}} - \frac{V}{\sqrt{K}} Q_{t_{k-1}}\left(1 - \frac{\alpha}{2}\right) \leq \xi x \leq \bar{\bar{x}} + \frac{V}{\sqrt{K}} Q_{t_{k-1}}\left(1 - \frac{\alpha}{2}\right) \qquad \text{spr}\alpha$$

Chapitre 4 :
Conception du simulateur

4.1) Introduction

Les méthodes de simulation, bien qu'anciennes, se sont fortement développées avec l'apparition de calculateurs puissants. Généralement, la technique de simulation est employée lorsque l'on se trouve devant un problème stochastique auquel il est impossible d'associer un modèle analytique. Bien que l'on puisse tenir compte d'hypothèses très complexes, il faut savoir que le programme ne simule qu'un modèle du système étudié et non le système réel. Les résultats de la simulation ne vaudront donc que par la qualité des hypothèses choisies. En effet, si elles sont trop fortes, elles peuvent être le défaut d'une analyse théorique d'un système. Par contre, lorsqu'elles sont insuffisantes, elles peuvent conduire à une simulation tellement complexe qu'il devient aussi difficile d'analyser le modèle que le système lui-même.

Les résultats fournis par la simulation peuvent être utilisés dans deux buts bien distincts. Lorsque le programme est employé pour simuler un réseau non existant, l'utilisateur doit faire varier les paramètres en entrée. La lecture des résultats doit lui permettre de choisir la configuration la plus adaptée à ses besoins. Par contre, lorsqu'il s'agit d'une simulation d'un réseau déjà installé, l'exécution du programme doit aider le gestionnaire du système à détecter les origines d'un affaiblissement des performances. Celles-ci peuvent provenir de deux sources bien distinctes: les supports de transmission et/ou les serveurs. Elles font chacune appel à des notions différentes. En effet, d'une part, les observations se font sur des paquets et de l'autre sur des processus. Vu qu'il est très complexe de faire un lien entre ces deux concepts, nous effectuons dans un premier

temps la simulation des lignes de communication. Elle est décrite au point 4.2. Ensuite, nous développons un second modèle qui traite les serveurs. Il est présenté au point 4.3.

4.2) Simulation des lignes du réseau

4.2.1) Présentation du réseau Bus Ethernet

Il existe plusieurs types de réseaux locaux. Les différences résident principalement dans la manière dont les paquets transitent. Le réseau de Hamm est du type Bus Ethernet. Les messages émis sont envoyés sur toute la ligne. Chaque poste "regarde" les messages qui passent sur le réseau. Si l'adresse de destination du paquet correspond à celle du poste, celui-ci lit les informations contenues dans le paquet.

Le temps de diffusion sur la ligne dépend de plusieurs critères :

- la longueur de la ligne
- la vitesse de propagation d'un message
- la vitesse de la ligne
- la longueur du paquet

Lorsque nous possédons toutes ces informations, nous pouvons calculer le temps de transfert d'un paquet par la formule :

$$\text{tps\_service (sec)} = \frac{\text{longueur\_ligne (m)}}{\text{vitesse\_propagation (m/sec)}} + \frac{\text{longueur\_paquet (bits)}}{\text{vitesse\_ligne (bits/sec)}}$$

1) Il existe une longueur maximale pour la ligne. Elle dépend du support de transmission choisi. On en distingue deux types :

- le Thick-Ethernet; câble multi-brins capable de transmission sur longue distance (±1600m) mais celui-ci est rigide et donc difficile à placer.

- le Thin-Ethernet; câble coaxial souple mais dont la longueur est limitée à 180m.

2) La vitesse de propagation correspond à la vitesse de déplacement d'un bit sur la ligne.
3) La capacité de la ligne est représentée par sa vitesse. Pour un réseau Ethernet, la vitesse standard est de 10Mbits/sec.
4) La longueur du paquet est le seul élément variable. En effet, sur les réseaux Ethernet, la taille des paquets n'est pas constante. Elle varie de 64 à 1518 bytes.

Le temps de propagation sur la ligne est obtenu à partir des deux premiers paramètres. Sachant que la vitesse de propagation est de plus ou moins 200000 Km/sec [VAN90], nous pouvons considérer que le temps nécessaire pour qu'un bit traverse un câble d'une centaine de mètres est négligeable. C'est pourquoi nous admettons que le temps de service de la ligne ne dépend que de la taille du paquet à transmettre et de la capacité de la ligne. Il est donc obtenu par la formule :

$$\text{tps\_service (sec)} = \frac{\text{longueur\_paquet (bits)}}{\text{vitesse\_ligne (bits/sec)}}$$

4.2.2) Elaboration du modèle

Le modèle employé pour la simulation des lignes de communication est assez simple (fig. 4.1). La ligne constitue le serveur dont le temps de service est déterminé par la taille du paquet. En ce qui concerne le taux d'arrivée au serveur, il est inutile de connaître le taux d'émission de chaque poste. En effet, comme il s'agit d'un réseau Bus Ethernet, quels que soient les postes émetteurs et récepteurs, le temps de service est toujours identique. Il nous est permis de faire cette hypothèse car le but recherché est d'établir si la ligne constitue un frein aux performances du réseau et non d'analyser l'influence d'un poste particulier sur le comportement de la ligne.

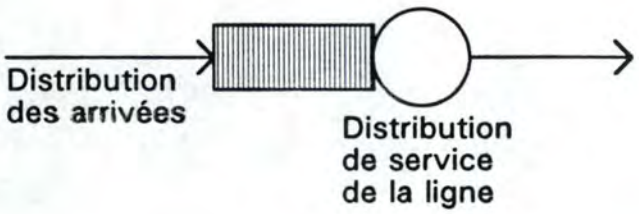


fig 4.1 : Modèle pour la simulation de la ligne d'un réseau Ethernet

4.2.3) Le programme

Notre programme est basé sur la notion d'événement. Nous en distinguons deux types :

- la fin de traitement d'un paquet par la ligne (fin de service)
- l'émission d'un paquet sur la ligne par un poste (arrivée)

Les mesures se font en régime stationnaire. La manière employée pour déterminer la période transitoire est décrite au paragraphe 3.4. Le programme nécessite deux paramètres en entrée : la distribution de la taille des paquets et la distribution globale des arrivées au serveur.

4.2.4) Détermination des distributions

Nous disposons à l'Institut d'Informatique de Namur d'un logiciel qui possède de multiples possibilités d'analyse statistique. Il s'agit du Lan Probe développé par Hewlett Packard. Il est constitué d'une partie Hardware -une boîte branchée sur le câble qui "regarde" tout ce qui passe sur la ligne- et d'une partie Software -logiciel statistique s'exécutant sous Windows- qui permet notamment de consulter les informations présentes au niveau 2 du modèle OSI quel que soit le protocole employé (intéressant pour les réseaux hétérogènes). Cet outil nous permet de déterminer la distribution de la taille des paquets ainsi que celle du taux global des arrivées sur la ligne.

Nous ne disposons pas du Lan Probe au Luxembourg. Cependant, le réseau de l'Institut est aussi de type Bus Ethernet. Comme nous l'avons vu, la configuration du réseau n'a pas d'effet sur le comportement de la

ligne. Il suffit de connaître le taux d'arrivée global des paquets sur la ligne. Nous avons constaté que le réseau de Namur est plus chargé que celui de Hamm[1]. Nous avons donc effectué une simulation du réseau de l'Institut. Si les résultats obtenus nous permettent d'affirmer que la ligne ne constitue pas un frein aux performances du réseau et n'influence pas le temps de réponse, nous pouvons donc supposer qu'il en va de même pour un réseau moins chargé.

4.2.5) Résultats de la simulation

Le tableau 4.1 reprend les résultats de la simulation du réseau de l'Institut d'Informatique de Namur. Nous avons effectué plusieurs simulations. Nous avons utilisé la distribution de la taille des paquets obtenue à partir du Lan Probe et fait varier le taux d'arrivée global.

nb. paquets / seconde	100	200	500	700
taux d'utilisation (%)	12,12	21,22	56,21	77,73
tps attente moyen (ms)	0,06	0,12	0,62	1,32
nb. moyen dans la file	0,02	0,03	0,28	0,91

tableau 4.1 : Résultats de quatre simulations

Nous remarquons que pour atteindre une saturation de la ligne (taux d'utilisation supérieur à 70%), il faut un taux moyen d'arrivée supérieur à 700 paquets par seconde. Les mesures que nous avons effectuées à Namur, nous ont permi d'établir que le taux d'utilisation moyen est de 62.4 % et que le taux d'arrivée moyen est de 576 paquets par seconde. Le taux d'utilisation du réseau de Hamm étant largement inférieur à celui de l'Institut, nous pouvons considérer que ses performances ne sont pas influencées par la capacité des supports de communication mais bien par celle des serveurs.

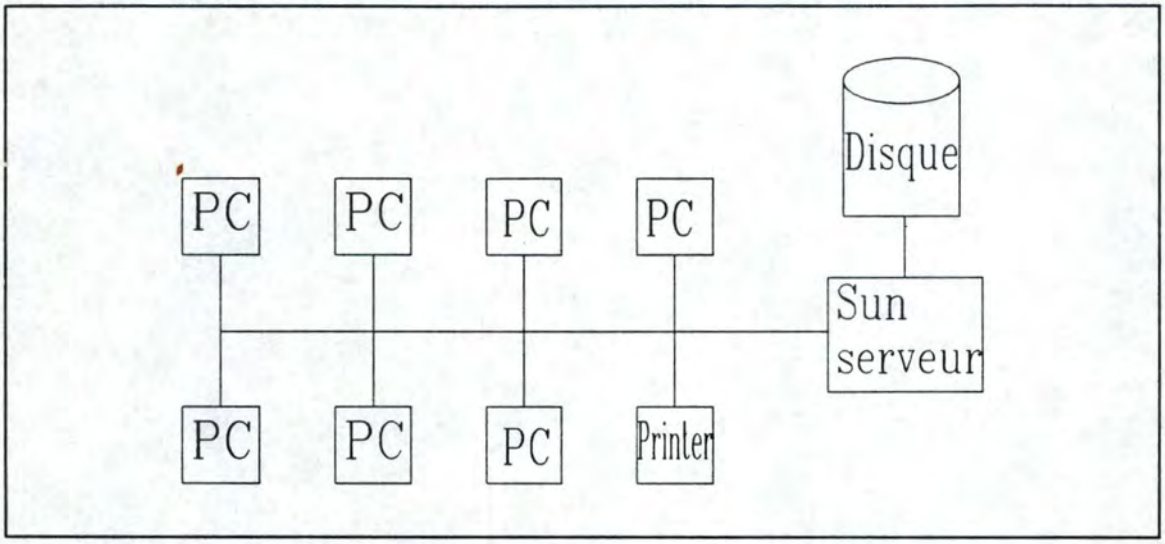
[1] Nous avons consulté et comparé les fichiers de comptabilité du système de Namur avec ceux de Hamm.

4.3) Simulation des serveurs

4.3.1) Définition du modèle

4.3.1.1) Présentation du modèle du réseau de Hamm

Le réseau de Hamm a la configuration suivante :



La notion de serveur a une place importante dans toute simulation de réseau. Un serveur est un gestionnaire d'une ou plusieurs ressources partagée(s) entre les différents utilisateurs branchés sur le réseau. Sur le réseau de Hamm, nous en distinguons trois : le serveur CPU qui gère le partage du processeur central (du serveur Sun), le serveur DISQUE qui gère la mémoire de masse (du serveur Sun) et le serveur IMPRIMANTE qui gère les impressions.

Il faut déterminer les relations entre ces différents serveurs. Pour cela, nous devons analyser la manière dont le système d'exploitation Unix gère les accès au disque et à l'imprimante (fig 4.2).

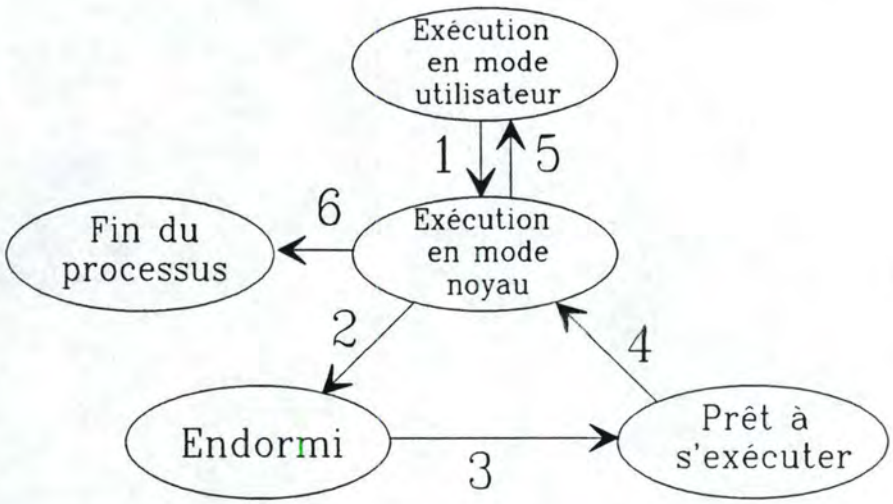


fig. 4.2 : Ordonnancement des processus lors d'une demande d'une ressource

Lorsqu'un processus veut accéder à une ressource (disque ou imprimante), il doit le faire via des appels systèmes. Il passe alors d'une "exécution en mode utilisateur" à une "exécution en mode noyau" (1). Si la ressource n'est pas disponible, il est "endormi" et placé dans un état d'attente (2). Dès que la ressource lui est accordée, il est réveillé et placé dans l'état "prêt à s'exécuter" (3). Quand il est élu par l'ordonnanceur pour occuper le CPU, il termine d'abord l'appel système en mode noyau (4) et peut alors repasser en mode utilisateur (5). Un processus termine donc toujours la demande d'une ressource par un ou plusieurs passages au CPU. C'est pourquoi, dans notre modèle (fig. 4.4), il n'est pas possible d'aller du serveur DISQUE au serveur IMPRIMANTE et inversément sans passer préalablement par le serveur CPU. De même, un processus termine toujours son exécution par un traitement au serveur CPU (6).

Rem : quand un processus demande une ressource, il est endormi et une requête est envoyée au serveur de la ressource sollicitée (DISQUE ou IMPRIMANTE). Le processus reste donc au serveur CPU. Dans notre modèle, nous faisons l'hypothèse que le processus passe d'un serveur à

l'autre, qu'il est lui-même porteur de la requête. Les résultats n'en sont pas modifiés. En effet, étant donné que le processus n'est réveillé que lorsque la ressource est accordée, la durée de son sommeil correspond au temps de traitement de la requête par le serveur concerné (DISQUE ou IMPRIMANTE). Nous ne faisons donc que changer son état : nous remplaçons l'état 'endormi' par les deux états 'servi par le disque' et 'servi par l'imprimante' (fig. 4.3). (fin de remarque)

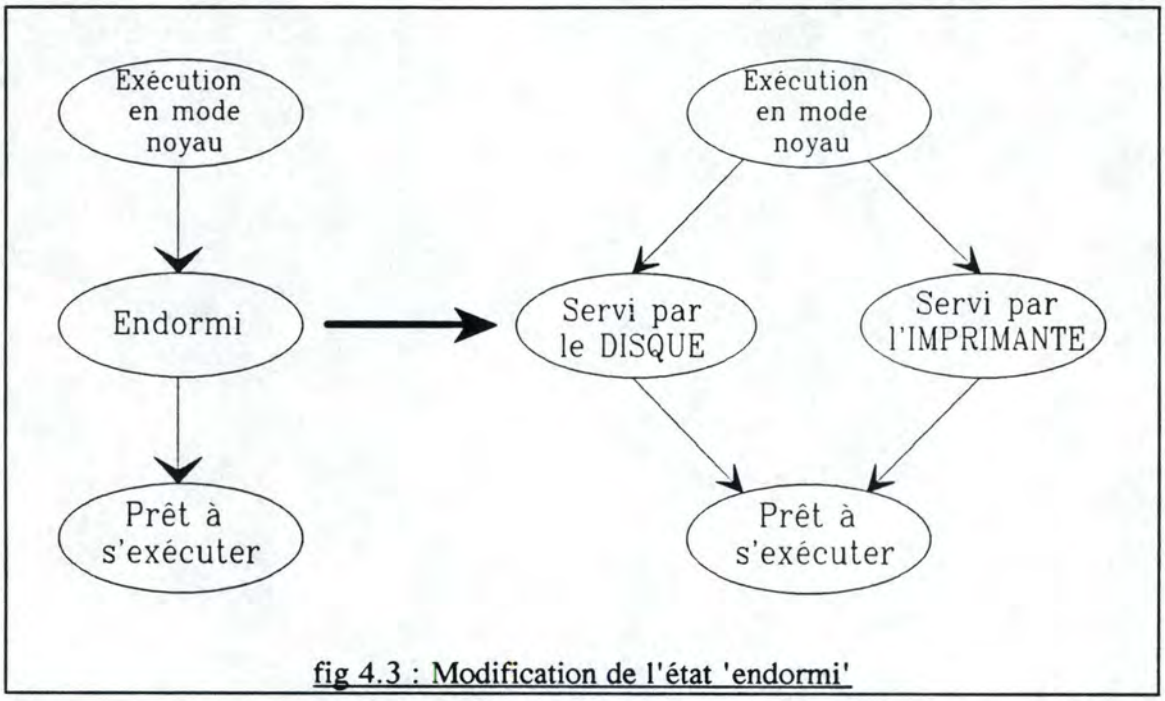


fig 4.3 : Modification de l'état 'endormi'

Le modèle du réseau est ouvert. Pour rappel, dans un réseau fermé, le nombre de processus dans le système reste constant tandis que, dans un réseau ouvert, il varie en fonction du temps. Dans notre cas, un terminal peut créer plusieurs processus qui tournent simultanément. Par exemple, lors d'un processus d'impression, la main est directement rendue au terminal qui peut immédiatement lancer un nouveau processus sans devoir attendre la fin de l'impression. Nous sommes donc bien en présence d'un réseau ouvert.

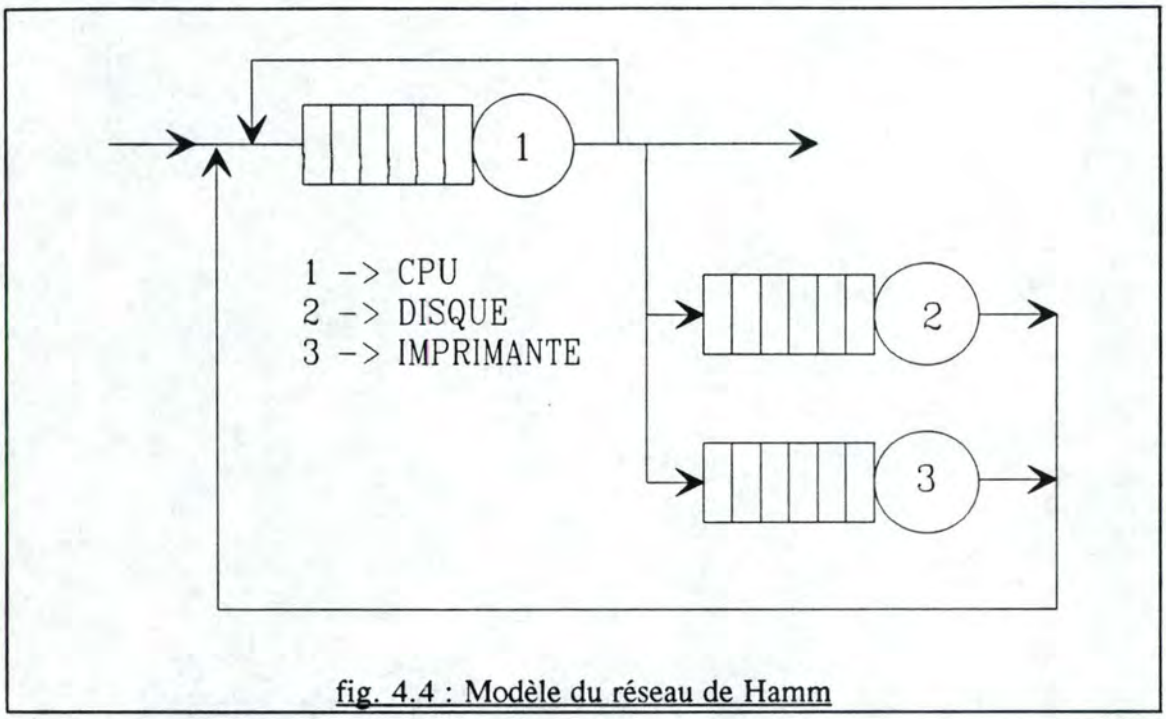


fig. 4.4 : Modèle du réseau de Hamm

4.3.1.2) Résultats désirés

Les informations fournies par le programme de simulation doivent être pertinentes. En effet, un gestionnaire de réseau doit pouvoir constater, à la lecture des résultats, la présence d'une anomalie (si elle existe).

Un taux d'utilisation trop élevé d'un serveur signale, par exemple, une saturation. Il faut cependant déterminer le taux d'occupation maximum admissible pour un serveur. En général, cette borne est fonction du temps de réponse des processus. En effet, par souci de rentabilité, il est préférable d'avoir le taux d'utilisation des serveurs le plus grand possible. La limite est toutefois atteinte lorsque les temps de réponse des processus deviennent trop longs. C'est pourquoi les résultats doivent fournir le temps moyen passé par les processus dans le système.

Si l'on remarque un temps de réponse trop important, il faut localiser le lieu de congestion. Le temps moyen d'attente et le nombre moyen de processus dans chaque file permettent d'établir, par serveur, s'il y a un engorgement.

Tout réseau traite différents types de processus. Il peut être intéressant pour le gestionnaire du système de diposer du temps de réponse moyen pour chaque type de processus. De cette manière, s'il doit ajouter de nouveaux utilisateurs au système, il sait se faire une idée de leurs impacts sur les performances du réseau en analysant les types de processus qu'ils vont émettre.
Voici la liste complète des résultats désirés :

- le temps de réponse (ou temps passé dans le système) moyen des processus
- le temps de réponse moyen par type de processus
- le taux d'utilisation des serveurs
- le nombre moyen de processus dans la file d'attente de chaque serveur
- le temps d'attente moyen des processus à chaque serveur

4.3.1.3) Données nécessaires en entrée

Afin d'avoir un programme de simulation complet et précis, il est nécessaire de disposer du plus grand nombre d'informations possible sur le matériel composant le réseau ainsi que sur l'utilisation de celui-ci. Nous avons distingué trois classes de données.

Nous avons tout d'abord les renseignements sur les arrivées des processus. Nous avons besoin de :

- la distribution des arrivées des processus au processeur
- la répartition des types de processus
- la densité des arrivées en fonction du temps
  ex : plus dense à 10h qu'à 13h.

La seconde catégorie concerne le taux d'utilisation des différents serveurs. Nous devons donc disposer des distributions de service global du :

- processeur (en msec)
- disque (en nombre d'entrées et sorties disque (E/S))
- imprimante (en nombre de pages)

La dernière catégorie reprend les caractéristiques techniques du matériel. Elles peuvent être divisées en deux parties : d'un côté, les données relatives au hardware et de l'autre celles relatives au software.

Pour ce qui est du hardware, nous avons besoin des informations suivantes :

- disque : vitesse de rotation, de déplacement de la tête de lecture, de transfert des données,...
- imprimante : vitesse d'impression, taille du buffer,...
- réseau : vitesse de transmission,...
- processeur : vitesse du CPU,...

Tandis qu'au niveau du software, nous devons disposer des données suivantes :

- PC-NFS : gestion des requêtes d' accès au disque et à l'imprimante
- UNIX : ordonnancement des processus, gestion des files d'attente

4.3.1.4) Hypothèses sur le système UNIX

Nous avons constaté qu'il est indispensable, lorsque l'on fait une simulation, d'émettre des hypothèses lors de la modélisation du système étudié. Comme vu précédemment, le programme de simulation a pour but de simuler le comportement des serveurs (du réseau de Hamm) qui est déterminé par le système d'exploitation utilisé. Nous présentons ici les hypothèses liées au fonctionnement du système d'exploitation Unix présenté au point 2.5.1.

Au sujet de la gestion des fichiers, nous disposons du nombre de blocs lus ou écrits (E/S) pour chaque processus. Nous devons donc utiliser une formule qui permette d'estimer le temps de service d'un processus au serveur DISQUE à partir du nombre de blocs lus ou écrits. Nous ne simulons pas la gestion des fichiers mais seulement les accès disques.

Il nous est impossible de traiter les exceptions qui sont des événements non attendus provoqués par le processus tels une division par zéro, un adressage illégal de la mémoire,...

Pour rappel, l'exécution d'un processus sur les systèmes Unix comporte deux niveaux (ou modes) : utilisateur et noyau. Pour pouvoir les simuler, il est indispensable de connaître avec précision les appels systèmes invoqués par chaque processus et les moments de ces appels. Nous ne disposons pas de ces données et ne faisons donc pas de distinction entre ces deux modes. Cette hypothèse n'a aucune influence sur les résultats car nous analysons le temps total passé par chaque processus au serveur CPU quel que soit le mode. Elle simplifie cependant fortement notre modélisation du système d'exploitation Unix.

Voici la liste de ses répercussions :

- au lieu d'avoir une file d'attente associée à chaque valeur de priorité, et ce pour chacun des deux modes, nous avons une seule file d'attente au serveur CPU qui sera gérée selon la stratégie Round Robin à une petite modification près. En effet, la stratégie Round Robin fait l'hypothèse que tous les processus ont la même importance. Dans notre cas, chaque processus a sa propre priorité. Nous avons également une file d'attente au disque et à l'imprimante qui sont gérées toutes les deux selon la stratégie FIFO.
- à chaque processus on attribue une priorité qui est régulièrement recalculée. Le choix du processus à élire se fait en fonction de celle-ci. Nous considérons que tous les processus se trouvent en mode utilisateur. Chaque seconde, la priorité de tous les processus en attente au serveur CPU est recalculée. Il en est de même pour le processus qui s'endort (fin de l'utilisation du serveur) au moment où il est interrompu.
- le diagramme des états se simplifie très fortement (fig. 4.5) d'autant plus qu'on ne tient pas compte de la gestion de la mémoire principale.

Quatre états subsistent :

1. le processus s'exécute
2. le processus est prêt à s'exécuter dès que le noyau l'élira
3. le processus est endormi
4. le processus a exécuté l'appel exit (fin de son exécution)

Nous avons donc cinq transitions :

--> 1 création d'un processus
1 --> 2 processus élu par le noyau
2 --> 1 fin de la tranche de temps
2 --> 3 demande d'une ressource non disponible
2 --> 4 le processus fait l'appel "EXIT"
3 --> 1 ressource demandée disponible

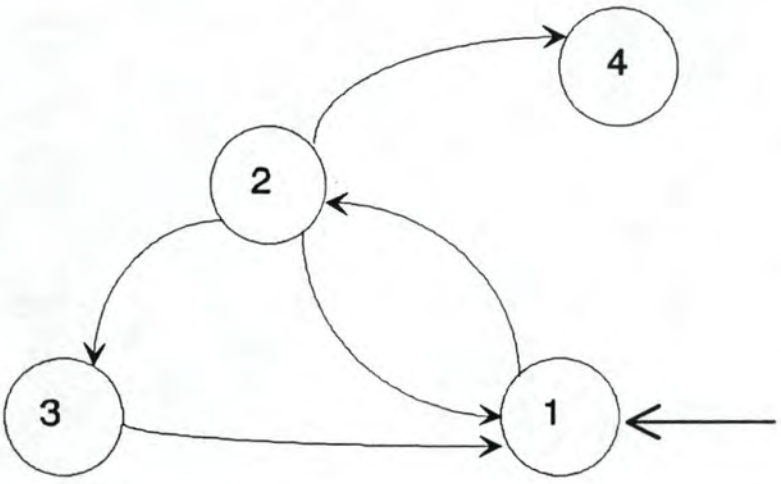


fig. 4.5 : Diagramme simplifié des transitions d'état

Soient les quatre états : "En exécution" (1), "Prêt" (2), "Endormi" (3) et "Zombie" (4). L'état "Prêt" regroupe l'ensemble des processus qui sont prêts à s'exécuter mais qui attendent la libération du processeur. Le processus qui occupe ce dernier se trouve dans l'état "En exécution". Dès que sa tranche de temps est écoulée, il est interrompu et repasse à l'état "Prêt" (2 --> 1). Un processus de ce même état est sélectionné selon des critères bien précis (principe du partage de temps) afin d'utiliser à son tour le serveur (1--> 2). Si le processus en cours d'exécution a besoin d'une ressource (qui n'est pas immédiatement disponible) pour pouvoir poursuivre son traitement, il est placé dans l'état "Endormi"(2 --> 3). De cette manière, le serveur est disponible pour un autre processus. Dès que l'appel système est terminé et la

ressource accordée, il repasse à l'état "Prêt" CPU (3 --> 1). Un processus qui termine son exécution est placé dans l'état "Zombie" (2 --> 4).

A chaque processus est associé un contexte qui est défini, entre autres, par son code, les valeurs de ses variables, les valeurs des registres machines, le contenu de ses piles, ...

Il est évident que, dans le cadre d'une simulation, nous ne disposons pas de toutes ces informations.

Le contexte d'un processus, dans notre cas, se limite aux données suivantes :

- le type de processus
- le temps de service restant à effectuer au serveur CPU
- le temps de service restant à effectuer au serveur DISQUE
- le temps de service restant à effectuer au serveur IMPRIMANTE
- la priorité

Lors de chaque changement de contexte, le noyau sauvegarde ces informations afin de pouvoir reprendre l'exécution de ce processus plus tard. Nous considérons qu'il faut cinq msec au noyau pour effectuer cette opération [TAN87].

Le noyau permet un changement de contexte dans trois circonstances :

- le processus se termine
- le processus s'endort (demande d'une ressource non disponible)
- le processus a terminé sa tranche de temps

Les programmes (processus) interagissent avec le noyau en invoquant un ensemble d'appels systèmes. Ceux-ci permettent aux processus de communiquer entre eux et avec le reste du monde, d'accéder aux fichiers, de créer de nouveaux processus, de terminer un processus, de synchroniser des étapes d'exécutions de processus,...

Ils peuvent être invoqués à tout moment par un processus. La méthode utilisée est expliquée au point suivant.

4.3.1.5) Ordonnancement des serveurs

Hypothèse d'alternance

Pour notre simulation, nous ne savons pas à quels moments un processus demande des ressources car cela dépend de son code d'exécution (appels systèmes) et varie donc d'un processus à l'autre.
Nous sommes obligés de généraliser le phénomène.
L'idée principale est d'alterner les passages au processeur et au disque en faisant l'hypothèse que tout processus commence par un traitement d'initialisation, répète le cycle "accès disque - traitement des données de l'accès disque" et termine par un traitement de clôture et par un accès à l'imprimante si nécessaire.

Détermination de p

Lorsqu'un processus entre au serveur CPU ou au serveur DISQUE, nous calculons sa probabilité p d'y rester à la fin de son passage et $1-p$ de changer de serveur. Elle est calculée en fonction des temps de service globaux restant à chacun des deux serveurs (CPU et DISQUE). Elle tient donc compte du nombre de passages déjà effectués.

Lorsqu'un processus est élu pour occuper le serveur CPU, nous calculons la probabilité p qu'il reste à ce serveur à la fin du service (fig. 4.6). La formule utilisée pour calculer la probabilité p que le processus en cours de service au disque reste à ce serveur après n accès disque est présentée à la figure 4.7. Nous utilisons un coefficient dans les deux formules. Ce choix est expliqué au point suivant.

$$p = \frac{(1) - (2)}{(1) - (2) + (3)}$$

avec (1) : tps_cpu_global_restant
(2) : min (durée_tranche, tps_cpu_global_restant)
(3) : tps_disque_global_restant * coeff

Rem : (1) - (2) correspond au temps global que le processus (qui va entrer au serveur CPU) devra encore passer au serveur CPU à la fin d'une tranche de temps complète

fig. 4.6 : formule du calcul de *p* au serveur CPU

$$p = \frac{((1) - (2)) * coeff}{((1) - (2)) * coeff) + (3)}$$

avec (1) : tps_disque_global_restant
(2) : N * tps_accès
(3) : tps_cpu_global_restant

Rem : (1) - (2) correspond au temps global que le processus (qui va entrer au serveur DISQUE) devra encore passer au serveur DISQUE après N accès disque

fig. 4.7 : formule du calcul de *p* au serveur DISQUE

Utilisation d'un coefficient

Au départ, nous voulons donner théoriquement la même chance à un processus d'aller au serveur CPU et au serveur DISQUE. Vu que les deux temps de service globaux (au CPU et au DISQUE) ne sont jamais identiques, il est nécessaire d'utiliser un coefficient pour donner la même importance aux deux serveurs. Celui-ci est obtenu par la formule :

coeff = tps_cpu_global / tps_disque_global

En effet, sans ce coefficient (fig. 4.8), si un processus a un beaucoup plus grand temps de service global à un serveur qu'à l'autre, lors de sa création, il a une grande probabilité d'aller d'abord au premier serveur et d'y rester un certain temps avant de passer au second. Ceci va à l'encontre de notre hypothèse d'alternance.

a) calcul de p sans utiliser de coefficient (lors de la création d'un processus)

tps_cpu_global = 100
tps_disque_global = 20
p = tps_cpu_global / (tps_cpu_global + tps_disque_global)
= 100 / (100 + 20) = 5 / 6 = 0,83

= = > le processus a une probabilité 0,83 d'aller au serveur CPU et une probabilité 0,17 d'aller au serveur DISQUE

b) calcul de p en utilisant le coefficient (lors de la création d'un processus)

tps_cpu_global = 100
tps_disque_global = 20
coeff = 100 / 20 = 5
p = tps_cpu_global / (tps_cpu_global + (tps_disque_global * coeff))
= 100 / (100 + (20 * 5)) = 100 / (100 + 100) = 0,5

= = > le processus a la même probabilité d'aller au serveur CPU et d'aller au serveur DISQUE

fig. 4.8 : Utilité du coefficient pour le calcul de p

Rem : les deux exemples ci-dessus sont théoriques car, en réalité, nous forçons les processus à commencer au serveur CPU pour faire le traitement d'initialisation. Ils permettent néanmoins de se rendre compte de l'utilité du coefficient. (fin de remarque)

Détermination des temps de service

Lorsqu'un processus est élu pour occuper le processeur, une tranche de temps lui est assignée. Néanmoins, pendant son exécution, il peut faire un accès disque ou effectuer une impression à tout moment via un appel système. Il est alors immédiatement interrompu. On peut déjà en conclure que la durée du temps de service d'un processus au serveur CPU varie en fonction des appels systèmes.

C'est pourquoi nous devons déterminer à l'avance si le processus va faire un appel système pour demander une ressource pendant sa tranche. Pour cela, nous calculons la probabilité p (fig. 4.6) que le processus a de rester au serveur CPU à la fin de la tranche de temps qui lui est accordée. Si on en conclut (à partir d'un nombre aléatoire) qu'il reste au processeur, nous savons qu'il doit faire une tranche complète. Son temps de service équivaut donc à la durée de la tranche.

Dans le cas contraire, il fait un appel système à un certain moment dans la tranche. Nous lui assignons donc un temps de service compris dans l'intervalle]0,tranche].

En résumé, pour établir la durée du service au serveur CPU, deux situations sont possibles :

- soit le processus utilise toute la tranche de temps qui lui est accordée et reste au serveur après son service :

$$tps\_service\_cpu = durée\_tranche \quad \text{(cas 1)}$$

- soit le processus fait une demande d'accès disque pendant son service. Cette demande peut avoir lieu à n'importe quel moment dans la tranche :

$$0 < tps\_service\_cpu < durée\_tranche \quad \text{(cas 2)}$$

La détermination de la durée de service au serveur DISQUE suit le même principe. Nous devons fixer le nombre de blocs à lire ou à écrire pour le passage. Nous calculons la probabilité *p* que le processus a de rester au serveur DISQUE après *N* accès disque selon la figure 4.7. *N* est initialisé à un. Nous l'incrémentons d'une unité jusqu'à ce qu'on détermine (à partir du *p* calculé) que le processus quitte le serveur DISQUE. Dans ce cas, *N* représente le nombre d'accès disque à réaliser par le processus pour le passage :

tps_service_disque = N * tps_acces

4.3.1.6) Principes de base

Notre simulation est basée sur la notion d'événement. Le principe est de traiter les événements chronologiquement dans le temps (fig. 4.9). A chaque type d'événement va être associée une variable *délai* qui indique le temps qui doit encore s'écouler avant le déclenchement de l'événement. Il s'agit alors de déterminer celui qui est le plus proche dans le temps en consultant les différentes variables *délai*, de progresser sur la ligne du temps de manière à se positionner sur le moment du déclenchement de l'événement et de traiter l'événement. Plusieurs événements peuvent se déclencher simultanément.

Une première étape consiste à détecter les différents types d'événement. Nous distinguons les événements fictifs, qui ne changent pas l'état du réseau (c'est-à-dire le nombre de processus présents à chaque serveur), et les événements réels, qui correspondent à des mouvements de processus (création d'un processus, terminaison d'un processus, changement de serveur pour un processus).

L'état d'un serveur peut être modifié soit par le départ d'un processus (fin de service), soit par l'arrivée d'un nouveau processus au processeur. On peut en déduire les quatre événements réels qui doivent correspondre à la génération d'une arrivée et aux traitements de la fin de service pour chaque serveur.

Le système d'exploitation, pour gérer le partage du processeur (serveur CPU), associe à chaque processus une priorité qui est réajustée

régulièrement. Il est donc nécessaire d'avoir un événement fictif 'calcul des priorités' qui déclenche le traitement de ces priorités.
Nous avons également un deuxième événement fictif 'génération des arrivées' qui résulte d'une limite lors de nos prises de mesure.

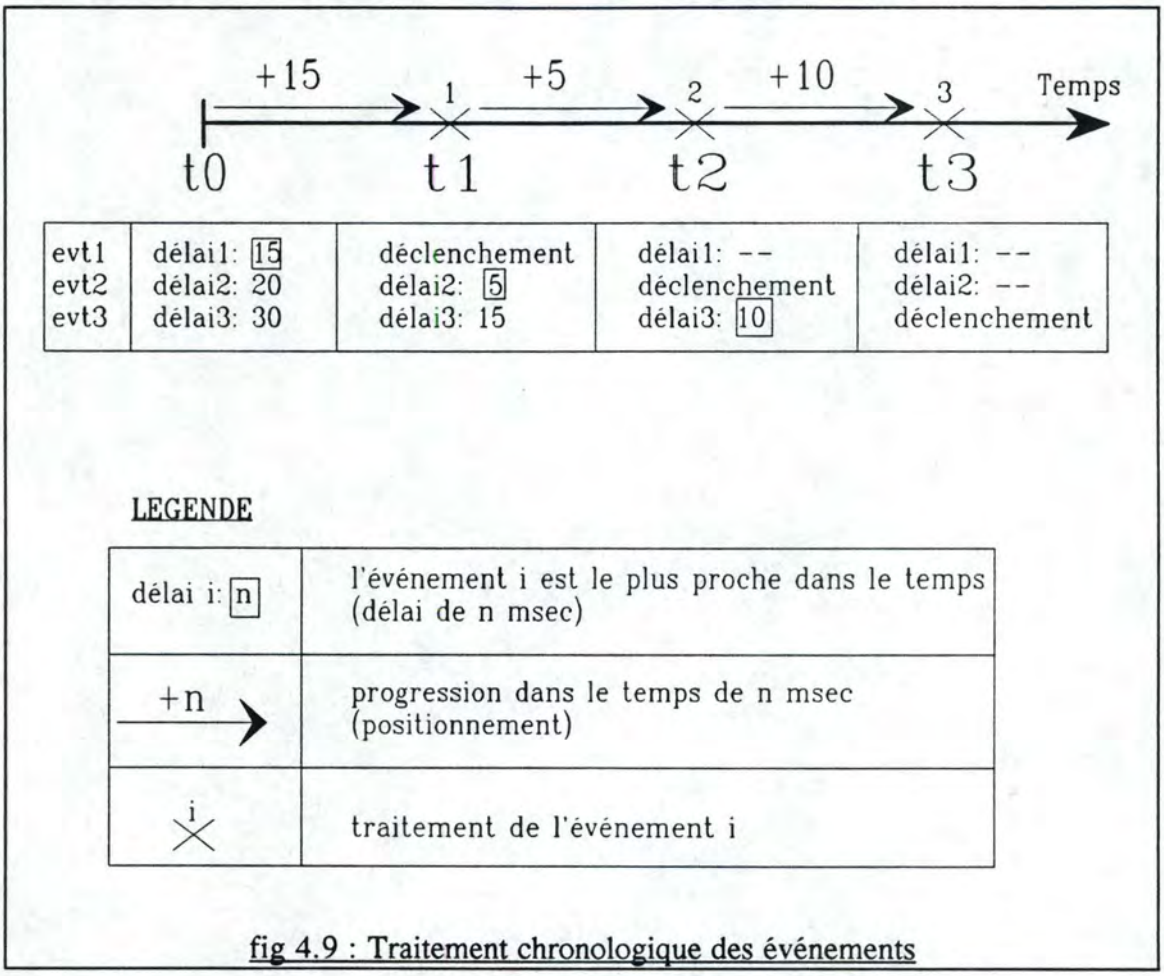


fig 4.9 : Traitement chronologique des événements

Les événements fictifs sont :
- 'génération des arrivées'
- 'calcul des priorités'

Les événements réels sont les suivants :
- 'génération d'une arrivée'
- 'fin de service CPU'
- 'fin de service DISQUE'
- 'fin de service IMPRIMANTE'

4.3.1.7) Présentation des événements

L'événement fictif 'génération des arrivées'

Nos prises de mesure nous ont permis de connaître la distribution des arrivées des processus dans le réseau. Pour rappel, nous disposons de l'instant de création de chaque processus. L'unité est la seconde. Nous avons donc pu calculer le nombre d'arrivées de processus par seconde. Il est cependant irréaliste de générer un flot d'arrivées chaque seconde. En effet, les divers temps de service aux différents serveurs sont exprimés en millisecondes. Des arrivées massives à intervalle régulier (chaque seconde) vont fausser les temps d'attente qui seront supérieurs à la réalité. Il est donc préférable de répartir les arrivées dans la seconde. Pour cela, il est nécessaire d'avoir un événement fictif *'génération des arrivées'* qui est déclenché chaque seconde et détermine le nombre de processus à générer dans la seconde.

L'événement fictif 'calcul des priorités'

Comme vu précédemment, dans le système Unix, chaque processus possède une priorité en mode noyau et en mode utilisateur. Pour rappel, nous ne considérons que le mode utilisateur. Lorsque l'ordonnanceur doit élire un processus dans la file d'attente du serveur CPU, il choisit en fonction de la priorité des différents processus ainsi que l'utilisation récente de l'unité centrale.

Toutes les secondes, le noyau recalcule la priorité des processus en mode utilisateur et ajuste l'utilisation récente de l'unité centrale 'UC' de tous les processus. Nous avons donc décidé de créer un second événement fictif *'calcul des priorités'* qui est déclenché chaque seconde et qui recalcule les priorités. Cet événement est nécessaire pour pouvoir simuler le principe du partage du processeur.

L'événement réel 'génération d'une arrivée'

Cet événement est indispensable pour implémenter les arrivées dans le réseau. Les différents moments de son déclenchement dépendent du nombre généré par l'événement fictif *'génération des arrivées'*. Nous avons décidé de les répartir selon des intervalles de temps constants.

L'événement 'fin de service CPU'

Il indique que le processus en cours d'exécution au serveur CPU est interrompu. La réaction va être fonction de l'origine de l'interruption. Il peut s'agir d'une demande d'une ressource par le processus via un appel système ou de la fin de la tranche de temps qui est accordée à ce même processus (gestion du partage de processeur).

L'événement réel 'fin de service DISQUE'

Il indique la fin de service du processus au serveur DISQUE.

L'événement réel 'fin de service IMPRIMANTE'

Il indique la fin de service du processus au serveur IMPRIMANTE.

4.3.1.8) Algorithme général

Notre programme effectue ses mesures statistiques en régime stationnaire de manière à pouvoir appliquer le théorème de Doob-Birkhov (voir chapitre trois). Il est composé de deux parties. La première a pour but de mettre le système en régime stationnaire si celui-ci existe. En cas de succès, la seconde enregistre toutes les informations nécessaires au traitement statistique des données du système.

Détection de la période transitoire

La détection de la période transitoire s'effectue en deux étapes (fig. 4.10). La première consiste à appliquer la règle heuristique (cfr. §3.4) qui permet d'estimer le moment où le système se trouve en régime stationnaire. Cela revient à déterminer n_0 qui correspond à la longueur de la période transitoire.

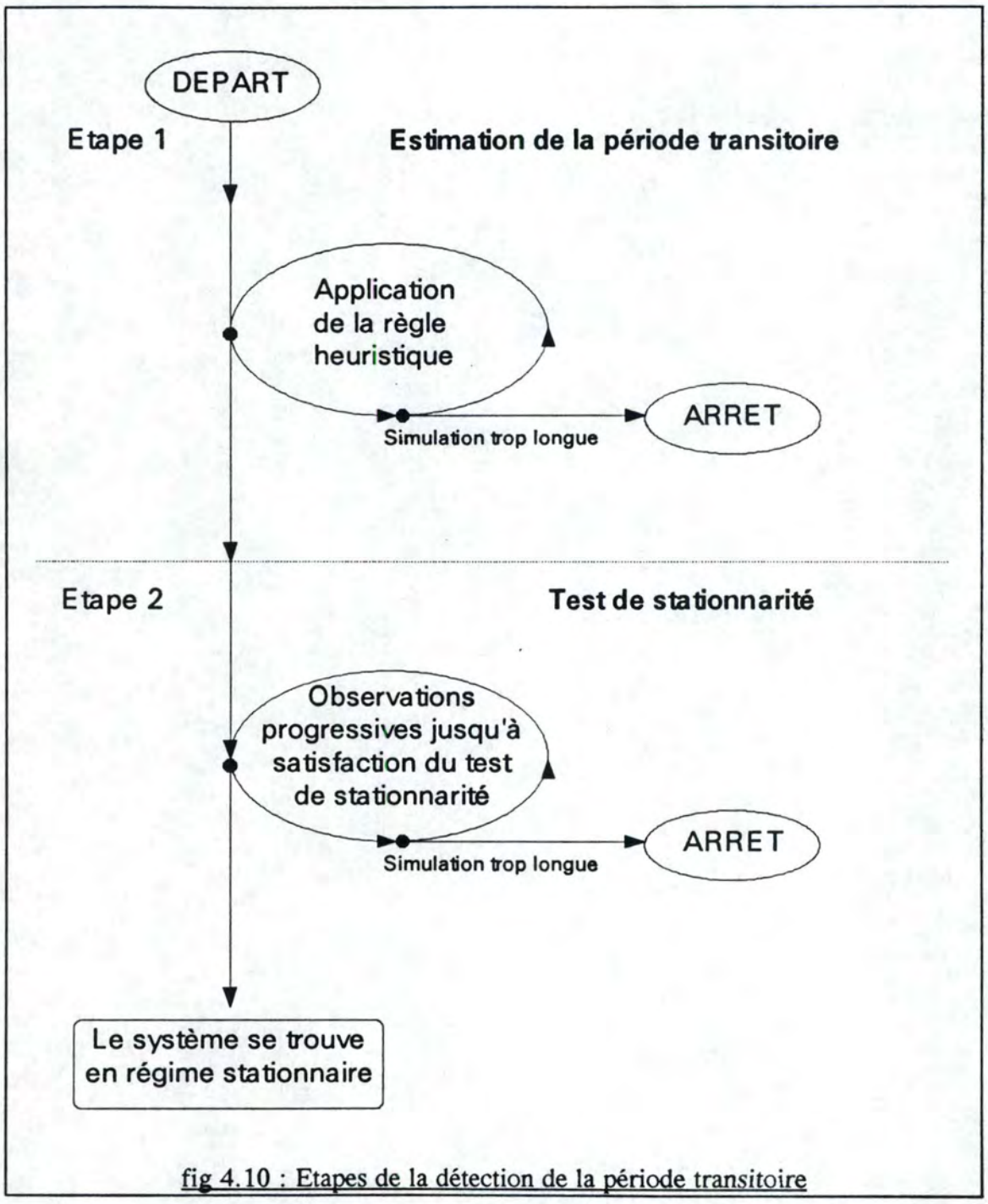


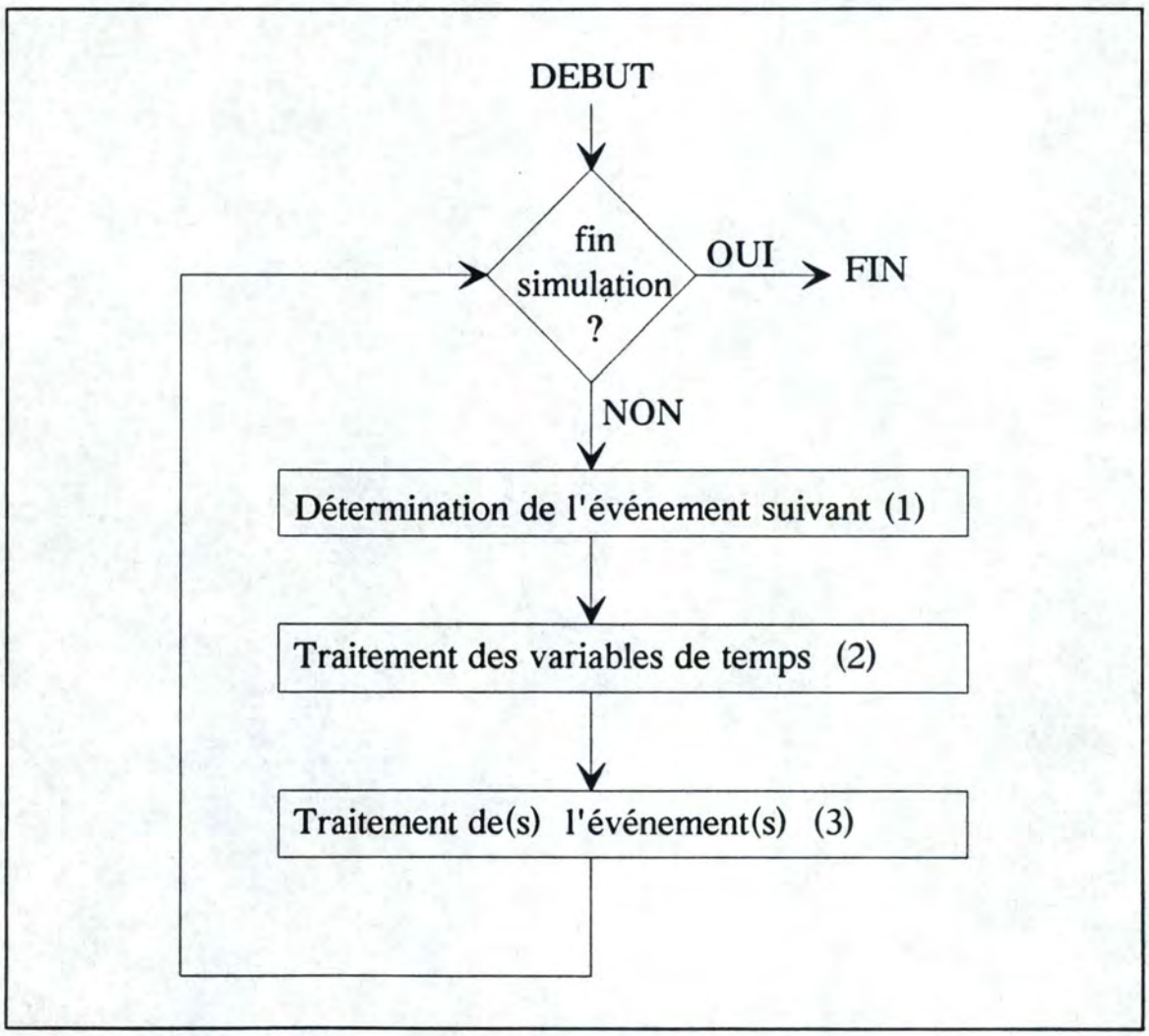
fig 4.10 : Etapes de la détection de la période transitoire

Lors de la seconde étape, nous effectuons le test de stationnarité (cfr §3.5). Il vérifie si l'estimation de n_0 est correcte. Dans l'affirmative, le système se trouve bien en régime stationnaire et les prises de mesures peuvent commencer. Dans le cas contraire, il faut effectuer des

observations supplémentaires jusqu'à ce que le test de stationnarité soit satisfait. Il convient cependant de donner une borne (n_O_max) à n_O. Si le régime stationnaire n'est toujours pas atteint après n_O_max observations, cela signifie que le système n'est pas stable. Les analyses statistiques doivent alors être effectuées en régime transitoire.

Analyses en régime stationnaire

L'algorithme général pour la prise des mesures a la forme suivante :



Nous avons donc trois étapes :

- (1) La notion de temps est présente dans toute simulation. Nous devons donc, avant de pouvoir traiter un événement, déterminer le type d'événement qui va se produire ainsi que le temps qui doit encore s'écouler avant son déclenchement.
Pour cela, nous disposons d'une variable *délai* par type d'événement qui contient cet intervalle de temps. Il nous suffit donc de calculer le minimum de ces variables (soit v_0) pour connaître l'événement le plus proche.

- (2) Avant de pouvoir traiter l'événement le plus proche, il faut se positionner sur son déclenchement dans le temps. Nous utilisons un compteur de temps qui représente le temps écoulé de la simulation. Ce compteur doit être incrémenté de l'intervalle de temps v_0 calculé au point (1). Nous devons évidemment mettre à jour toutes les autres variables de temps.

Voici l'ensemble des variables qui doivent être incrémentées :
- le compteur du temps écoulé de la simulation
- pour chaque serveur occupé :
    * le temps total octroyé par le serveur au processus en cours de service
    * le temps d'attente des éventuels processus dans la file du serveur
    * le temps d'occupation du serveur
- pour le serveur CPU s'il est occupé :
    * le temps d'utilisation récente de l'unité centrale (*UC*) par le processus en cours de service. Ceci est nécessaire pour pouvoir calculer sa priorité

Voici l'ensemble des variables qui doivent être décrémentées :
- la variable *délai* de chaque type d'événement dont le moment de déclenchement est connu (par exemple, il est évident que si le serveur CPU est vide, l'événement 'fin de service CPU' ne saurait pas se

déclencher. On ne doit donc pas traiter la variable *délai* associée à cet événement).

Rem : Nous avons deux types d'événement. Les événements fictifs sont utilisés pour déclencher une certaine action à intervalle de temps régulier t_0. La gestion des variables de temps *délai* associées à ces événements est assez simple. Chaque fois que l'événement se produit, nous savons que le prochain déclenchement aura lieu dans le laps de temps t_0. Il suffit donc de réinitialiser la variable *délai* avec cette valeur t_0.

Ce n'est pas le cas pour les événements réels qui ne sont pas réguliers et ne sont pas toujours prévisibles. La gestion des variables *délai* est plus complexe. Le moment de déclenchement d'un événement réel est imprévisible dans les situations suivantes :

- pour chaque serveur, s'il est libre, il est impossible de déterminer quand aura lieu la prochaine fin de service;
- s'il n'y a plus d'arrivées à générer dans la seconde en cours, le moment de la prochaine arrivée dépend du nombre généré 'nb_arr_gen' par l'événement 'génération des arrivées'. Il faut donc attendre son déclenchement.

Dans les cas cités ci-dessus, nous avons décidé d'attribuer une très grande valeur à la variable *délai* de l'événement en question. De cette manière, elle n'entre pas en ligne de compte lors de la détermination de l'événement le plus proche (étape 1). Lors de l'étape 2, nous ne modifions pas la variable *délai* d'un événement dont le moment de déclenchement n'est pas encore connu vu que sa valeur (9999999) a une signification bien précise. (fin de remarque)

-(3) Nous savons qu'un événement doit se produire. Son traitement va être fonction de son type.

Si l'événement est du type 'génération des arrivées' :

on détermine le nombre de processus à générer, 'nb_proc_a_gen', et ce , à partir de la distribution des arrivées. Pour cela, nous générons un nombre aléatoire *nb_al* et nous regardons dans la distribution des arrivées (fig 4.11) la valeur correspondante.

Comme vu précédemment, on répartit ces arrivées dans la seconde selon des intervalles constants. On calcule dès lors l'intervalle de temps (en milliseconde) qui va séparer chaque arrivée en appliquant la formule suivante:

intervalle = 1000/nb_proc_a_gen

Distribution

Valeurs	x0	x1	x2	.....	xi	.....	xn
Observations	r0	r1	r2	.....	ri	.....	rn

si ('nb_al' < (r_0 + r_0 + ... + r_i)) AND
('nb_al' > (r_0 + r_1 + ... + r_{i-1}))
alors 'nb_arr_gen' = x_i

fig 4.11 : Détermination de 'nb_arr_gen' à partir d'un nombre aléatoire 'nb_al'

La première arrivée est générée au temps 0 (début de la seconde). Admettons que *N* processus doivent être générés dans la seconde, les arrivées se passeront comme suit :

intervalle = 1000/N
arrivée n°1 t_1 = 0 msec
arrivée n°2 $t_2 = t_1$ + intervalle = 1000/N msec
arrivée n°3 $t_3 = t_2$ + intervalle = 2000/N msec
...
arrivée n°N-1 $t_{N-1} = t_{N-2}$ + intervalle = (N-2)*1000/N msec
arrivée n°N $t_N = t_{N-1}$ + intervalle = (N-1)*1000/N msec

La variable *délai* associée à l'événement est chaque fois réinitialisée à 1000 msec (1sec) car l'événement doit se déclencher chaque seconde.

Si l'événement est du type 'calcul des priorités' :
on recalcule la priorité de tous les processus en attente au serveur CPU (dans l'état 'endormi et prêt à s'exécuter') selon la formule suivante :

priorité = priorité_initiale + (UC/ 2)

L'utilisation récente de l'unité centrale (*UC*) de tous les processus (sauf celui en cours de service au serveur CPU) est réajustée selon la fonction d'extinction :

extinction(UC) = UC/2

La priorité du processus en cours d'exécution au serveur CPU ainsi que son utilisation récente de l'unité centrale sont recalculées lorsque le processus termine son service.

Si l'événement est du type 'génération d'une arrivée' :
le traitement consiste à:
- créer un processus
- lui affecter un type (selon la distribution des types)
- en fonction de ce type, déterminer le temps de service global au processeur (*tps_cpu_global*) et le nombre d'E/S
- à partir de ce nombre d'E/S, calculer le temps de service global au disque (*tps_disque_global*)
- si le type du processus est une impression, déterminer le temps de service global à l'imprimante
- calculer le coefficient *tps_cpu_global / tps_disque_global*
- ajouter le processus dans la file du serveur CPU
- incrémenter d'une unité le nombre de processus générés pour la seconde en cours. S'il est inférieur au nombre total de processus à créer (*nb_proc_a_gen*), la variable 'délai' associée à l'événement 'génération d'une arrivée' est réinitialisée.

Rem : les affectations du type, du temps de service global au processeur et à l'imprimante, et du nombre d'E/S se font selon le même principe que

celui expliqué lors de la détermination du nombre de processus à générer pour la seconde (fig 4.11). Cependant, pour les distributions des temps de service, les x_i ne représentent pas des valeurs mais des intervalles. Nous avons décidé de renvoyer la valeur qui correspond au milieu de l'intervalle x_i. (fin de remarque)

Le temps de service global du disque est calculé à partir du nombre d'E/S. Le temps nécessaire pour faire une E/S (tps_accès) est fonction du temps de déplacement du bras, du positionnement de la tête de lecture/écriture et du temps de lecture (et d'écriture) d'un bloc de données (une E/S).

La formule d'estimation du temps de lecture et d'écriture d'un bloc de données (d'un E/S) est la suivante :

tps_accès = tps_depl_bras + tps_1/2_rotation + tps_lecture_écriture

(tps_accès ≈ 20 msec [CAR92])

Le temps d'impression est obtenu à partir de la vitesse de l'imprimante et du nombre de pages à imprimer. Le temps de service global à l'imprimante est calculé par la formule :

$$\text{tps\_prn\_global (sec)} = \frac{\text{nb\_pages (pg)}}{\text{vitesse\_impr (pg/m)}} * 60$$

Si l'événement est du type 'fin de service CPU':

- Soit le processus a terminé sa tranche de temps; il est interrompu et placé dans la file d'attente du serveur CPU afin de permettre à un autre processus de s'exécuter. Il s'agit en fait du partage de processeur .
- Soit le processus a fait une demande d'accès disque, il est alors placé dans la file d'attente du serveur DISQUE.
- Soit le processus a fait une demande d'impression, il est dès lors placé dans la file d'attente du serveur IMPRIMANTE.

- Soit le processus a terminé son exécution; on place les différentes données nécessaires au calcul des résultats dans les blocs adéquats. Le processus est alors tué.

Le serveur CPU est donc libre. Dans ce cas, si sa file d'attente n'est pas vide, un nouveau processus est sélectionné en fonction des priorités des processus présents dans cette file afin d'occuper le serveur CPU. Si deux processus sont éligibles (s'ils ont la même priorité et que cette priorité est la plus élevée), celui qui attend depuis le plus longtemps sera choisi. La durée de la tranche de service est calculée de manière à réinitialiser la variable *délai* associée à l'événement.

Si la file d'attente du serveur CPU est vide, on affecte une valeur très grande (9999999) à la variable *délai* de manière à ce qu'elle n'entre pas en ligne de compte lors du choix de l'événement suivant.

Si l'événement est du type 'fin de service DISQUE' :

- le processus est placé dans la file d'attente du serveur CPU pour terminer l'appel système invoqué et continuer son exécution si nécessaire.

Le serveur DISQUE est libre. Si la file d'attente n'est pas vide, le processus suivant est sélectionné (selon la stratégie FIFO) afin de faire ses accès disque. Le temps de service est calculé en fonction du nombre d'accès à réaliser afin de réinitialiser la variable *délai* associée à l'événement.

Si la file d'attente du serveur DISQUE est vide, on affecte une valeur très grande (9999999) à la variable 'délai' de manière à ce qu'elle n'entre pas en ligne de compte lors du choix de l'événement suivant.

Si l'événement est du type 'fin de service IMPRIMANTE' :

- le processus est placé dans la file d'attente du serveur CPU pour terminer l'appel système invoqué et continuer son exécution si nécessaire.

Le serveur IMPRIMANTE est libre. Si la file d'attente n'est pas vide, le processus suivant est sélectionné (selon la stratégie FIFO) afin de faire son impression. Vu que nous faisons l'hypothèse qu'un processus fait toutes ses impressions en une fois, on réinitialise la variable *délai*

associée à l'événement avec le temps de service global du processus d'impression.
Si la file d'attente du serveur IMPRIMANTE est vide, on affecte une valeur très grande (9999999) à la variable *délai* de manière à ce qu'elle n'entre pas en ligne de compte lors du choix de l'événement suivant.

4.3.1.9) Traitement statistique des observations

Le régime transitoire

Si le système n'a pas encore passé le test de stationnarité alors que $n_0 > n_{0\_max}$, nous devons effectuer nos mesures en appliquant les formules en régime transitoire. Pour cela, nous réalisons N simulations ($N > 100$) et prenons les mesures à un moment bien précis T (cfr. §3.3).

Chaque simulation se termine donc au temps T. Il faut alors réinitialiser toutes les variables du système pour pouvoir démarrer la suivante. On répète cette opération jusqu'à ce qu'on ait suffisamment d'observations (N).

Calcul des résultats

Pour le calcul des taux d'utilisation de chaque serveur, il faut deux compteurs :

- le nombre de simulations (N)
- le nombre de succès (s) (succès ≡ serveur occupé)

La fréquence est obtenue par :

$$f = \frac{s}{N}$$

L'intervalle de confiance est calculé selon la formule suivante :

$$f - \sqrt{\frac{\rho\,(1-\rho)}{n}}\, Q_G\,(1 - \frac{\alpha}{2}) \le \rho(t) \le f + \sqrt{\frac{\rho\,(1-\rho)}{n}}\, Q_G\,(1 - \frac{\alpha}{2})$$

Pour les autres résultats, nous appliquons la méthode expliquée à la section 3.3.

Le régime stationnaire

Dès que nous sommes en régime stationnaire, nous pouvons commencer à prendre les mesures nécessaires pour calculer les résultats désirés à savoir le temps de réponse moyen des processus et par type de processus, le nombre moyen de processus dans le réseau et, pour chaque serveur, le temps d'attente moyen, le nombre moyen de processus dans la file d'attente et le taux d'utilisation.
Nous allons analyser chaque résultat afin de déterminer les mesures nécessaires (observations) ainsi que les moments adéquats pour les prendre.

Calcul des résultats.

Pour le calcul du taux d'utilisation de chaque serveur, deux compteurs sont suffisants :

- un compteur pour le temps d'utilisation total du serveur
- un compteur pour le temps de simulation total

Ces compteurs sont incrémentés au cours de la simulation.
Il suffit alors, à la fin de la simulation, de faire le calcul suivant:

$$\text{taux\_utilisation\_serveur} = \frac{\text{temps\_utilisation\_serveur}}{\text{temps\_total\_simulation}} * 100$$

La philosophie est différente pour les autres résultats. Nous devons calculer des intervalles de confiance. Pour cela, nous utilisons la méthode des blocs d'observations indépendants. Elle comprend trois paramètres : le nombre de blocs, la taille de chaque bloc et le nombre d'observations séparant chaque bloc. Ces paramètres sont facilement modifiables. Nous avons choisi de travailler avec 50 blocs de 50 observations; chaque bloc étant séparé de 50 observations (que nous ne traitons pas), de manière à avoir une indépendance entre les blocs (cfr. §3.4.2).

Pour le calcul du nombre moyen de processus dans la file d'attente de chaque serveur, nous apportons une modification à la méthode précédente. En effet, nous devons tenir compte de l'intervalle de temps entre chaque observation. Chaque mesure est multipliée par la durée de l'intervalle de temps pendant lequel on l'a observée. Pour le calcul de la moyenne des 50 observations d'un bloc, nous devons diviser la somme de ces observations par la somme des intervalles de temps. L'exemple présenté ci-dessous démontre l'utilité de la prise en compte des intervalles de temps.

Admettons l'énoncé suivant :
nous avons observé respectivement 4 et 2 processus dans la file d'attente du serveur pendant 20 et 10 msec. Si nous représentons cela de manière plus mathématique, nous obtenons :

Observations	Intervalles de temps
x1 = 4	[t0,t1] = [0 , 20]
x2 = 2	[t1,t2] = [20 , 30]

Le calcul de la moyenne sans tenir compte des intervalles de temps donne :

$$\frac{x1 + x2}{2} = \frac{4 + 2}{2} = 3$$

Par contre, le calcul de la moyenne en tenant compte des intervalles de temps donne :

$$\frac{x1\,(t1 - t0) + x2\,(t2 - t1)}{(t1 - t0) + (t2 - t1)} = \frac{100}{30} = 3{,}3$$

4.3.2) Conception logique

4.3.2.1) La base de données

Le modèle conceptuel

Rappelons que World Systems désire, par le biais de notre programme, offrir un service supplémentaire à ses clients. En effet, le logiciel permet de simuler des réseaux existants afin de déceler des problèmes éventuels tels une saturation d'un serveur, une mauvaise configuration du système,... Il est également destiné à étudier le comportement de réseaux avant leur installation afin d'adapter au maximum leur architecture aux besoins du client. Il est donc intéressant que World Systems possède une base de données contenant les réseaux de ses clients ainsi que des réseaux types pouvant servir de références. Elle doit naturellement contenir toutes les informations nécessaires à la simulation. Afin d'en donner une définition structurée et rigoureuse, nous nous inspirons du modèle Entité-Association. Ce modèle permet d'exprimer la sémantique des données mémorisables à l'aide de concepts d'entité, d'association, d'attribut et du mécanisme des contraintes d'intégrité. Nous nous limitons à la description du schéma final (fig. 4.12) et expliquons les motivations qui ont guidé nos choix. Le lecteur intéressé trouvera dans [BOD90] une analyse complète et détaillée du modèle.

Définition des types d'entité

RESEAU

Notre base de données contient uniquement les réseaux qui peuvent être simulés par notre programme. Ils doivent donc avoir une configuration similaire au réseau de Hamm. C'est pourquoi la définition énoncée ci-dessous ne reprend que les réseaux basés sur une architecture client-serveur. Le réseau doit être géré par un système d'exploitation Unix.

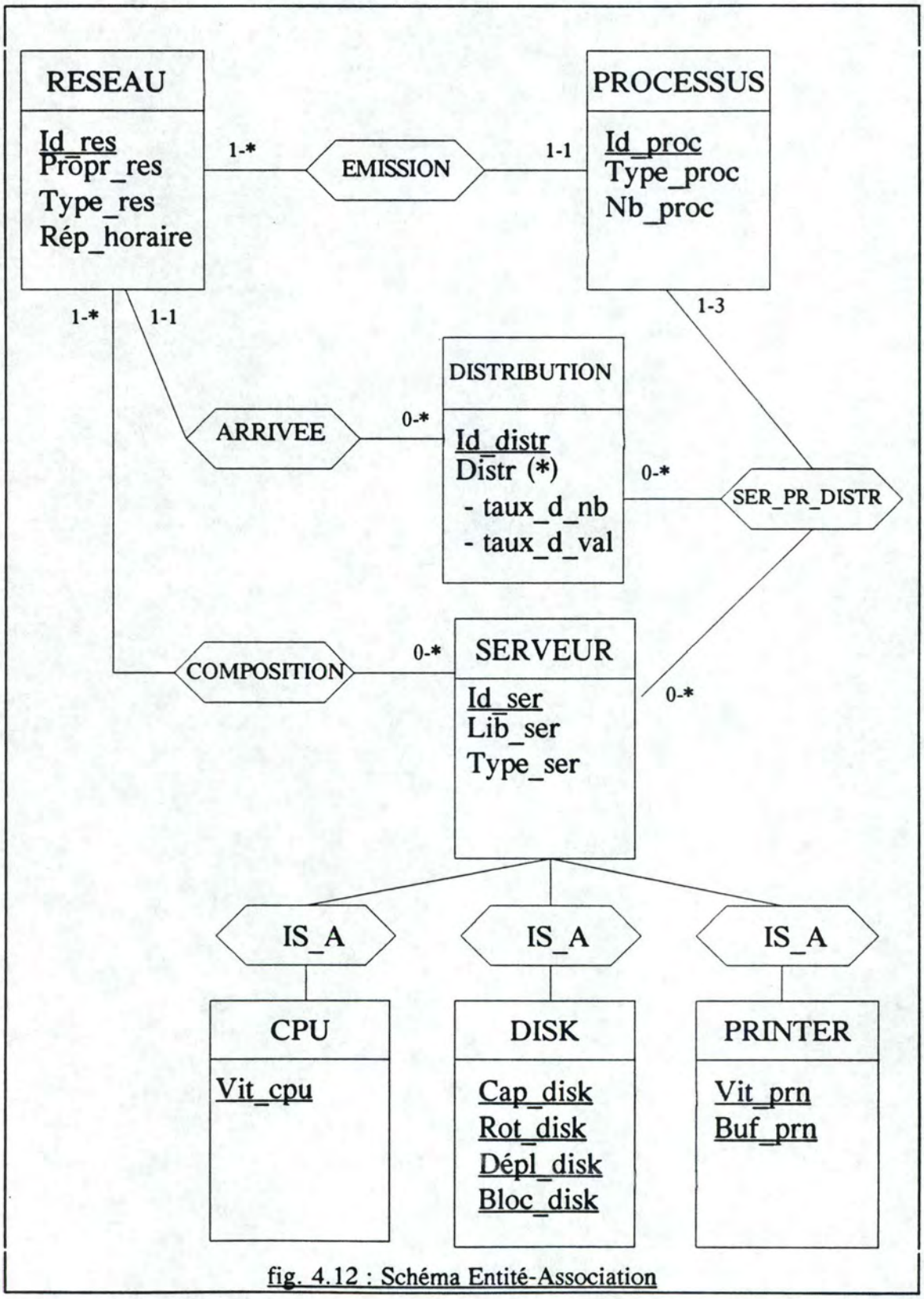


fig. 4.12 : Schéma Entité-Association

Définition constitutive

Tout système qui se compose d'un câble de communication reliant un serveur et un poste au minimum est considéré comme réseau. Le disque dur est géré par un système de fichiers distribués. Une ou plusieurs imprimantes peuvent être reliées au réseau. Le réseau doit être géré par un système d'exploitation UNIX.

Attributs

ID_RES : Identifiant du réseau

Chaque réseau possède un numéro qui l'identifie. A partir de ce numéro, il doit être possible d'obtenir toutes les informations nécessaires à la simulation. Nous avons choisi un entier plutôt qu'une chaîne de caractères, car ceci élimine le problème lié à l'orthographe et au choix des majuscules ou minuscules. Ce critère est primordial lorsqu'il s'agit d'un attribut identifiant.

PROP_RES : Propriétaire du réseau

Cet attribut permet d'ajouter de l'information au réseau simulé. Il correspond en général au nom du client. Lorsqu'il s'agit d'un réseau référentiel (sans propriétaire) on lui assigne une valeur qui permet de caractériser le réseau.

TYPE_RES : Type du réseau

Bien que cette information ne soit pas indispensable pour la simulation, il peut être intéressant de connaître l'architecture du réseau simulé. Le domaine de cet attribut est limité, il faut en effet choisir une architecture existante comme par exemple : token bus, bus Ethernet,...

REP_HORAIRE : Répartition horaire

Un poste de travail n'est pas utilisé de manière constante durant toute la journée. Dans toute entreprise, il existe des pauses café, un arrêt à midi,... De plus, la productivité est souvent meilleure le matin. Lors de nos prises de mesures à Hamm, nous avons en effet constaté une plus grande utilisation du réseau dans la matinée. C'est pourquoi nous avons

rendu possible de répartir la charge de travail de manière non constante sur la journée (entre 9h et 17h).

PROCESSUS

Lors de l'étude du comportement du réseau de Hamm, nous avons constaté que les distributions du temps de service des serveurs variaient en fonction du type de processus. C'est pourquoi nous avons créé une entité PROCESSUS qui contient tous les types de processus qui peuvent être émis par les postes de travail. Le type est caractérisé par les ressources utilisées par le processus et les distributions du temps de service de celui-ci aux différents serveurs. Remarquons que le choix des types de processus influencera directement les résultats de la simulation. Une étude approfondie devra être effectuée afin de détecter les différents types de processus émis par le réseau étudié et ainsi accroître la validité des résultats. Cette démarche exige beaucoup de temps. A Hamm, nos mesures ont conduit à distinguer quatre types de processus.

Définition constitutive

Un processus est une occurrence d'un programme en exécution. Il est caractérisé par les ressources nécessaires à son exécution.

Attributs

ID_PROC : Identifiant du processus

Nous assignons un numéro par type de processus. Ceci élimine tout risque de confusion présent lorsque l'identifiant est une chaîne de caractères.

TYPE_PROC : Type du processus

Cet attribut permet de connaître le type de processus correspondant au numéro. Le type détermine le temps de service du processus aux différents serveurs.

NB_PROC : Nombre de processus

Afin de connaître le taux d'arrivée des processus d'un certain type par serveur, il faut connaître le nombre de processus de ce type émis par jour. Dans le cadre de notre simulation, il n'est en effet pas nécessaire de connaître la distribution d'émission des processus par poste car nous ne simulons que les serveurs et non les lignes.

SERVEUR

Le programme doit donner des résultats quant aux performances des différents serveurs. Bien que le processeur (CPU), le disque dur et l'imprimante soient tous trois des serveurs, chacun d'eux possède des caractéristiques totalement différentes. C'est pourquoi nous allons appliquer la technique de la généralisation [BOD90]. Ceci est en fait un mécanisme d'abstraction en fonction duquel une relation entre des classes d'objets est considérée comme une classe d'objets génériques de plus haut niveau. Les objets spécifiques héritent des propriétés de l'objet générique. Nous considérons que le T.E. SERVEUR est le type générique et que les T.E. CPU, DISQUE et IMPRIMANTE sont les types spécifiques.

Définition constitutive

Toute machine qui permet l'exécution d'un processus.

Attributs

T.E. Générique : SERVEUR

ID_SER : Identifiant du serveur

Comme pour les autres T.E., les serveurs sont identifiés par un numéro (entier).

LIB_SER : Libellé du serveur

Le libellé correspond au modèle du serveur.

T.E. Spécifique : CPU

VIT_CPU : Vitesse du processeur

La vitesse du processeur est exprimée en millier d'instructions par seconde (Mips).

T.E. Spécifique : DISQUE

CAP_DISQUE : Capacité du disque dur

La capacité est exprimée en Mégabytes. Cet attribut n'est pas nécessaire pour la simulation.

ROT_DISQUE : Vitesse de rotation du disque dur

La vitesse de rotation du disque dur est exprimée en nombre de tours par seconde.

DEPL_DISQUE : Temps de déplacement de la tête de lecture

L'unité du temps de déplacement est la milliseconde.

BLOC_DISQUE : Taille d'un bloc

La taille d'un bloc est donnée en kilo-byte.

TRANSFERT_DISQUE : Temps de transfert des données

Le temps de transfert correspond au temps nécessaire à un bloc pour passer du disque dur à la mémoire centrale. Ce temps se mesure en millisecondes.

T.E. Spécifique : IMPRIMANTE

VIT_PRN : vitesse d'impression

Le choix de l'unité pour la vitesse d'impression n'est pas chose simple. En effet, l'unité utilisée pour les imprimantes matricielles est le nombre de caractères par seconde tandis que pour les imprimantes laser, on parle en nombre de pages à la minute. La majorité des réseaux actuels sont équipés d'imprimantes laser et tout porte à croire que dans un futur proche, les imprimantes matricielles n'auront plus leur place dans de tels systèmes. C'est pourquoi nous utilisons le nombre de pages par minute comme unité de mesure.

BUF_PRN : Taille du buffer de l'imprimante

La taille du buffer de l'imprimante influence les performances du spooler. L'unité de mesure est le Méga-byte.

DISTRIBUTION

Afin d'effectuer la simulation, il faut disposer d'un ensemble de distributions de service ainsi que d'une distribution des arrivées. Les mesures que nous avons prises sur le réseau de Hamm, nous apprennent qu'il n'est pas possible de se baser sur une distribution théorique. Nous devons donc pouvoir enregistrer dans la base de données les informations recueillies lors de nos prises de mesures.

Définition constitutive

Elément qui permet de générer des valeurs en fonctions des probabilités fixées.

Attributs

ID_DISTR : Identifiant de la distribution

Chaque distribution est identifiée par un numéro.

PROB_DISTR (i) : Probabilités cumulées de la distribution

VAL_DISTR (i) : Valeurs de la distribution

Définition des types d'association

EMISSION

Le comportement du réseau est fonction des types de processus exécutés. C'est pourquoi nous avons relié les T.E. RESEAU et PROCESSUS par le T.A. EMISSION. Il est dès lors possible de connaître tous les types de processus émis par le réseau étudié et ainsi accéder aux distributions qui dépendent à la fois du type de processus et du serveur. Un réseau peut naturellement émettre plusieurs types de processus (connectivité 1-N) et un type de processus ne peut être repris que par un seul réseau (connectivité 1-1). Ce T.A. ne possède pas d'attribut.

SER_PR_DISTR

A un type de processus et à un serveur peut correspondre une distribution. Un processus est au moins associé à un serveur et maximum trois (connectivité 1-3). Ceci se traduit dans le modèle Entité-Association par une association ternaire. Cette association va relier les trois T.E. PROCESSUS, SERVEUR et DISTRIBUTION. Elle ne possède pas d'attribut. Un serveur et une distribution peuvent jouer plusieurs fois le rôle (connectivité 0-N).

COMPOSITION

Afin de représenter la connexion des serveurs au réseau simulé, il est nécessaire d'effectuer un lien entre les T.E. SERVEUR et RESEAU. Tous les serveurs qui composent le réseau doivent participer à cette association. Un réseau peut contenir plusieurs serveurs et doit en posséder au moins un (connectivité 1-N). Un serveur d'un certain modèle peut être présent sur plusieurs réseaux différents (connectivité 0-N). Cette association ne possède pas d'attribut.

Le modèle de spécification de structures de données

"De l'expression conceptuelle d'une solution à sa forme exécutable en machine, il existe un certain nombre d'expressions qui, toutes, décrivent implicitement les mêmes concepts mais dont la forme est liée à l'intégration de certaines contraintes logiques ou techniques" [HAI86]. Le schéma conceptuel Entité-Association ne peut pas être utilisé tel quel lors de l'implémentation. Il est nécessaire de le transformer afin d'en obtenir une expression qui respecte les contraintes liées aux techniques employées. Le modèle d'accès généralisé (MAG) permet de représenter les concepts essentiels liés à tout système de gestion de base de données. A partir de ce schéma MAG, il nous est possible de définir entièrement la base de données. Cependant, s'il est obtenu par simple transformation du schéma conceptuel (fig 4.13), il ne respecte généralement pas les spécifications du système de gestion de base de données (SGBD) utilisé. Il convient d'y effectuer des transformations. Chaque modification a pour but d'éliminer un concept qui n'est pas conforme aux spécifications du

SGBD. Pour ce faire, nous disposons d'un ensemble de règles de transformation qui garantissent l'équivalence des deux schémas aussi bien du point de vue de la sémantique que de celui des accès. Le lecteur intéressé peut trouver une description détaillée de ces règles dans [HAI86]. L'expression finale doit respecter toutes les contraintes du SGBD; on dit alors du schéma qu'il est conforme au modèle du SGBD.

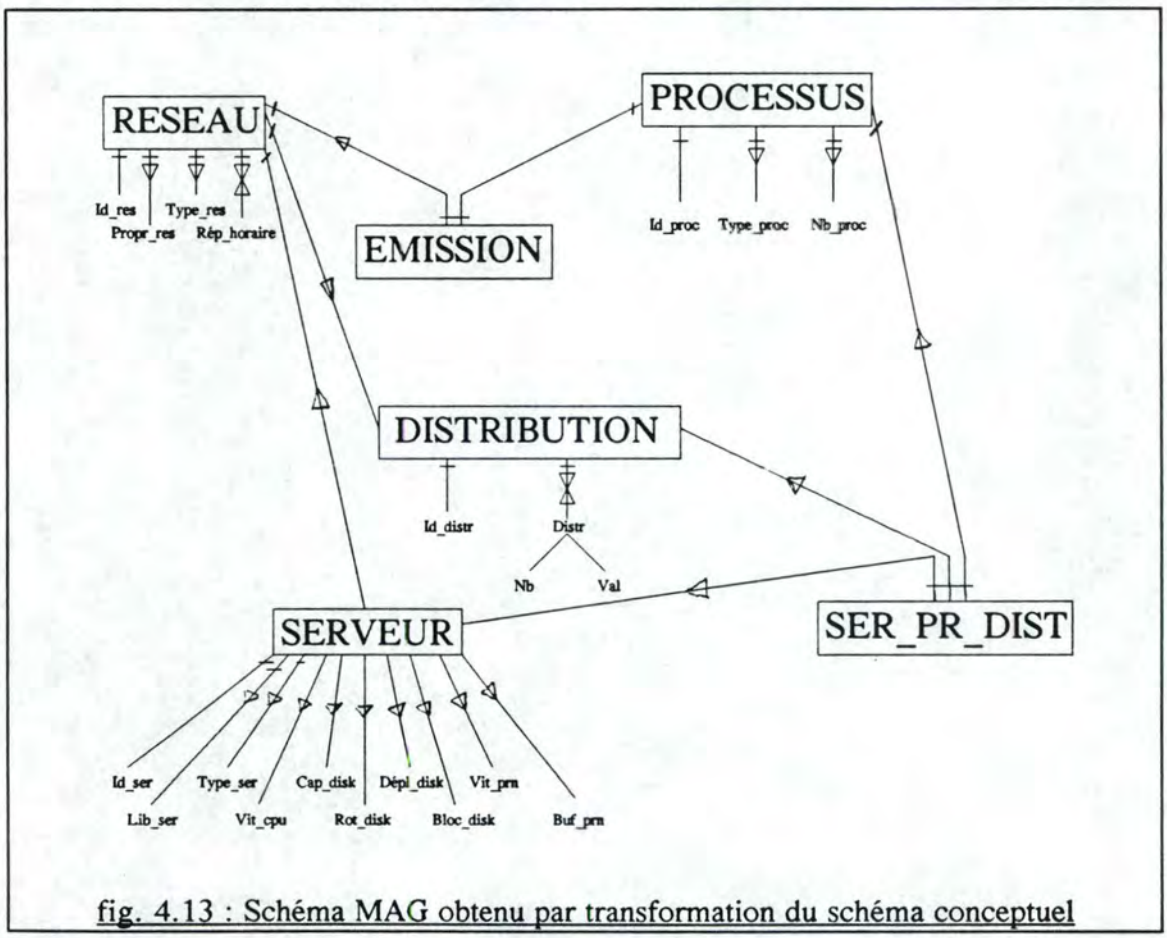


fig. 4.13 : Schéma MAG obtenu par transformation du schéma conceptuel

4.3.2.2) Le programme

Lorsque le modèle est constitué, il convient d'écrire le programme en respectant une méthodologie de travail. Pour cela, nous nous sommes inspirés de la méthode enseignée par Eric Dubois [DUB90]. Le développement du programme s'effectue en plusieurs phases :

a) établir l'architecture du programme

b) définir formellement les fonctions et primitives

c) écrire l'algorithme de chaque fonction et primitive

d) encoder et "debugger"

Les deux premières étapes font partie de la conception logique. Quant aux deux dernières, elles constituent la conception physique du programme.

a) Elaboration de l'architecture

Nous avons adopté une architecture hiérarchique (fig. 4.14). Chaque message doit respecter des conditions pour pouvoir descendre de niveau (préconditions) et possède des caractéristiques lorsqu'il sort de la fonction (postconditions). Nous évitons ainsi la propagation de messages erronés dans plusieurs fonctions.

L'ordre d'exécution des fonctions respecte le séquencement temporel. Ceci est lié à notre stratégie "event to event". En effet, le déroulement de notre programme suit le parcours de la ligne du temps. A chaque fois qu'un événement se produit, il effectue une série d'opérations liées à l'événement déclencheur.

Notre architecture comporte plusieurs niveaux. Les deux niveaux supérieurs reprennent toutes les fonctions du programme. Celles qui effectuent les calculs statistiques se trouvent au premier niveau; les fonctions de traitement des processus sont au second. En-dessous de celles-ci, nous avons placé les primitives. Une primitive correspond à une opération (consultation ou mise à jour) effectuée sur une donnée du système. Le niveau inférieur comprend le module d'accès à la base de données et le module de gestion de l'interface.

b) Définition formelle des fonctions et primitives

Afin de donner un définition précise et rigoureuse de chaque fonction et primitive, il convient de les définir de façon formelle. Il s'agit de donner les préconditions, postconditions, paramètres et variables intermédiaires pour chaque fonction et primitive. L'ensemble des définitions formelles se trouve en Annexe B (volume II).

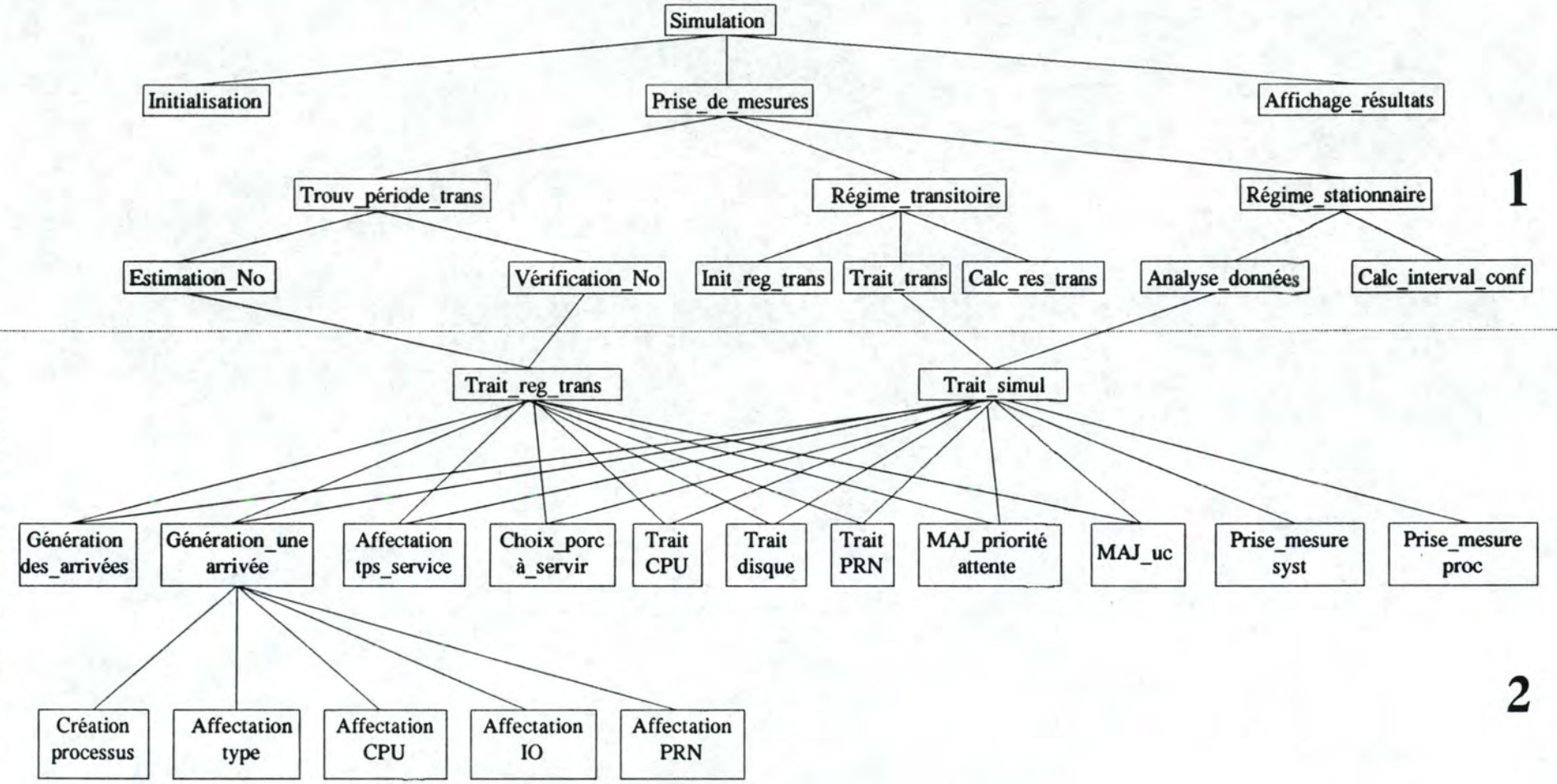


fig. 4.14 Architecture du programme

c) Ecriture des algorithmes

Cette phase consiste à traduire les définitions formelles en algorithmes.

d) Encodage et "debuggage"

Après avoir encodé les algorithmes, il convient de détecter toutes les erreurs. Si les erreurs de syntaxe sont décelées instantanément par le compilateur, il en va tout autrement pour les erreurs de programmation. Il nous faut donc adopter une méthode pour le debuggage des fonctions et des primitives. Nous avons commencé le travail de vérification et de correction par le bas de la hiérarchie. Après avoir vérifié l'exécution de toutes les primitives, nous avons remonté la hiérarchie niveau par niveau, fonction par fonction. Etant assurés de la validité des niveaux inférieurs, il nous était plus facile de vérifier les fonctions de plus haut niveau.

4.3.3) Conception physique

4.3.3.1) La Base de données

La conception physique consiste à créer, à partir du schéma MAJ initial, un autre schéma MAJ conforme aux spécifications du SGBD utilisé.

Spécifications d'un SGBD relationnel

Le système de gestion de base de données qui nous est proposé est le système développé par OCELOT au Canada. Il s'agit d'un SGBD relationnel compatible SQL pour PC. Les contraintes d'un tel système sont les suivantes [HAI86]:

1) Il n'y a pas de types de chemins;
2) Les items sont simples et élémentaires;
3) Les items sont obligatoires (possibilité de valeur NULL);
4) Il y a au moins un item par type d'articles;
5) Tout identifiant est une clé d'accès et une clé de tri;

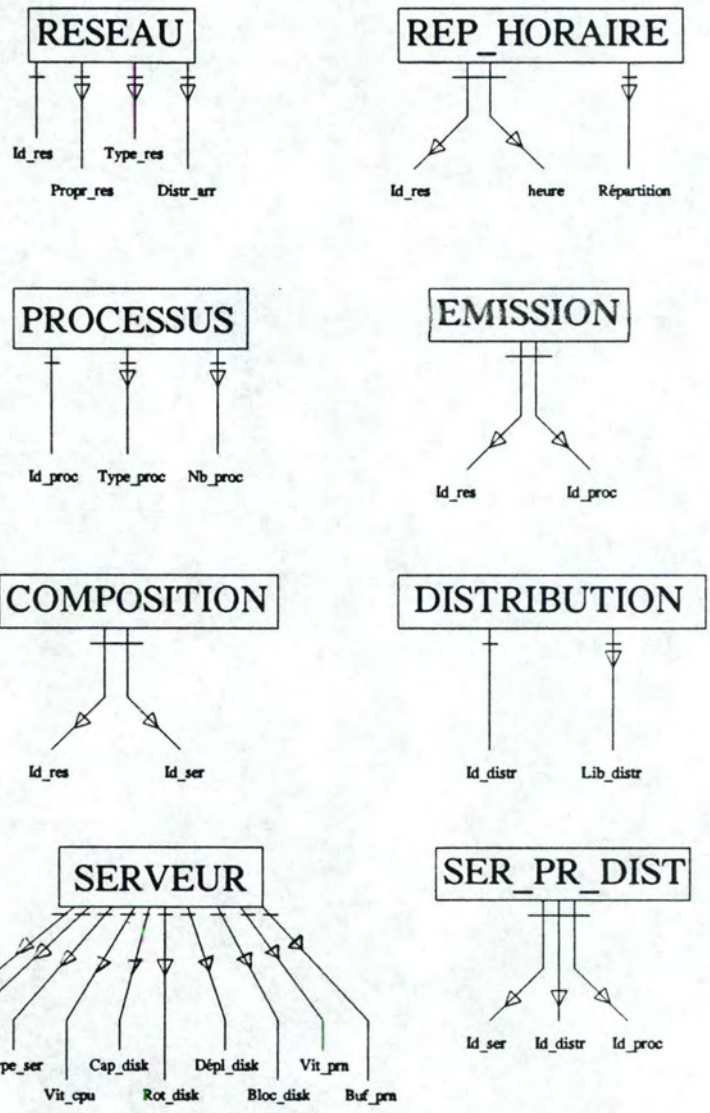
6) Il n'y a pas d'article système, ni de notion de fichier ni de base de données;
7) Il n'y a pas de contraintes d'intégrité autres que sur le type de valeurs d'un item; en particulier pas de contraintes d'inclusion.

Production d'un schéma MAG conforme

Nous n'expliquons pas les différentes étapes qui nous ont permis d'établir le schéma conforme au modèle relationnel (fig 4.16). Celui-ci est obtenu en appliquant les règles de transformation décrites dans [HAI86]. Il est cependant nécessaire d'analyser le cas de l'entité DISTRIBUTION qui a fait l'objet d'une transformation particulière. En effet, l'application de la règle permettant d'éliminer l'item répétitif VAL_DISTR a pour effet de créer un type d'article auquel il est difficile de donner une signification. De plus, comme une distribution possède trente éléments, ce nouveau type d'article possède trente occurrences par distribution. Cette solution n'est donc pas "saine". C'est pourquoi nous avons décidé de placer les distributions dans un fichier indépendant de la base de données. Il convient au programme de vérifier l'intégrité des données c'est-à-dire de s'assurer que pour tout identifiant d'une distribution présente dans la base de données, cette distribution est bien présente dans le fichier. La figure 4.15 donne la structure du fichier C contenant les distributions.

```
struct distribution
{
  int        id_distr;
  char       lib_distr[31];
  unsigned   val[31];
  float      rep[31];
};
```

fig. 4.15 : Structure du fichier C de l'entité distribution



rem : Les contraintes supplémentaires sont présentées à la page suivante

Fig 4.16(a) : Schéma MAG conforme au modèle relationnel

1) DISTR_ARR (:RESEAU) in ID_DISTR (:DISTRIBUTION)
2) ID_RES (:REP_HORAIRE) in ID_RES (:RESEAU)
3) ID_RES (:EMISSION) in ID-RES (:RESEAU)
4) ID_PROC (:EMISSION) in ID_PROC (:PROCESSUS)
5) ID_RES (:COMPOSITION) in ID_RES (:RESEAU)
6) ID_SER (:COMPOSITION) in ID_SER (:SERVEUR)
7) ID_SER (:SER_PR_DIST) in ID_SER (:SERVEUR)
8) ID_DISTR (:SER_PR_DIST) in ID_DISTR (:DISTRIBUTION)
9) ID_PROC (:SER_PR_DIST) in ID_PROC (:PROCESSUS)
10) ID_SER (:SER_PR_DIST) et ID_PROC (:SER_PR_DIST) doivent appartenir au même réseau

fig. 4.16(b) : contraintes liées au schéma MAG conforme

4.3.3.2) Le programme

Lorsque l'analyse conceptuelle est terminée, la traduction des définitions formelles en un langage de programmation est aisée. Nous devons toutefois écrire le code sans oublier que le programme va certainement subir des corrections et qu'il sera de toute manière nécessaire d'effectuer une maintenance du programme. Afin de permettre une maintenance rapide et efficace, nous avons procédé par regroupement. Toutes les primitives qui effectuent des opérations (mise à jour, consultation) sur une donnée de même type (processus, file d'attente, distribution,...) sont regroupées au sein d'un même fichier. Il en va de même pour toutes les primitives d'accès à la base de données (module d'accès) et pour toutes les fonctions d'affichage à l'écran (module de gestion de l'interface). A chaque fonction correspond un fichier. Ceci permet, à partir des définitions formelles, de vérifier, fonction par fonction, si les spécifications sont bien respectées. La phase de "debuggage" est ainsi facilitée.

Lors de la conception physique, il est nécessaire d'effectuer des choix d'implémentation. Il s'agit principalement de choisir la façon de

stocker les données en mémoire. Notre programme traite deux types de données. Il y a d'une part celles qui subissent des traitements durant tout le programme. Il s'agit ici des informations attachées aux processus et aux serveurs. D'autre part, il y a les observations effectuées au cours de la simulation. Elles constituent les données statistiques qui doivent être mémorisées pour subir un traitement en fin de programme.

Chaque processus, durant la simulation, voyage de file d'attente en file d'attente. Notre choix d'implémentation doit donc permettre de retirer ou d'insérer rapidement un processus dans une file. De plus, le caractère aléatoire de la simulation ne permet pas d'établir avec certitude une taille maximum pour la file d'attente. C'est pourquoi, nous avons choisi de représenter la file d'attente sous forme de liste chaînée (fig. 4.17). Un pointeur est dirigé vers le premier élément de la chaîne. Un second est pointé vers le dernier élément de la file; ceci permet une insertion rapide d'un élément. En effet, vu que la file d'attente du disque et celle de l'imprimante sont gérées selon la discipline FIFO (First In First Out), tout processus qui arrive doit se positionner à la fin de la file. Nous considérons que le premier élément de la file d'un serveur contient le processus en cours de service.

Lors de la simulation, il faut souvent accéder au nombre d'éléments des files d'attente. Afin de ne pas devoir chaque fois parcourir toute la liste chaînée, nous avons ajouté l'attribut "NB_ELT" au type FILE D'ATTENTE. Cet attribut doit être mis à jour à chaque insertion ou supression d'un processus dans la liste chaînée.

```
struct file
{
        struct    elt_file  *ptr_first;
        struct    elt_file  *ptr_last;
        int       nb_elt;
} ;
```

fig. 4.17 : Structure d'une file d'attente

```
struct elt_file
{
    struct    processus    *proc;
    struct    elt_file     *ptr_svt;
} ;
```

fig. 4.18 Structure d'un élément d'une file d'attente

Une file d'attente est composée d'une séquence (l'ordre a de l'importance) de processus (fig. 4.18).

Chaque processus est représenté par les variables de la structure de la figure 4.19. Le "TYPE" permet de déterminer les distributions à consulter pour lui affecter les temps de service. Trois tableaux de trois éléments sont utilisés pour mémoriser son temps de service global, son temps de service déjà obtenu et son temps d'attente dans la file pour chaque serveur du système. L'indice '0' correspond au serveur CPU, le '1' au serveur DISQUE et le '2' au serveur IMPRIMANTE. Les quatre attributs suivants sont destinés au calcul des priorités et les deux derniers sont nécessaires pour établir la probabilité que le processus a de rester au processeur (ou au disque) après avoir effectué sa tranche.

```
struct processus
{
    unsigned type;
    unsigned tps[3];
    unsigned tps_octroyé[3];
    unsigned tps_file[3];
    int      priorité_init;
    int      priorité;
    int      uc;
    int      att_tranche;
    int      pass_svt;
    int      coeff;
} ;
```

fig. 4.19 : Structure d'un processus

Quant aux serveurs, nous leur associons une variable qui permet d'établir le temps d'utilisation.

Les valeurs des observations statistiques sont stockées en mémoire centrale pendant la simulation et sont consultées en fin de programme. La méthode des blocs indépendants (que nous avons adoptée pour effectuer les calculs statistiques) nous permet d'établir au début du programme le nombre d'observations à effectuer. Nous avons donc naturellement décidé de placer, pour chaque résultat désiré, les valeurs observées dans un tableau dont la taille dépend du nombre et de la taille des blocs (cfr §3.4).

Les choix d'implémentation doivent naturellement tenir compte des performances du programme. Ceci est d'autant plus nécessaire pour un programme de simulation que chaque opération est répétée un très grand nombre de fois. Lors de la simulation, des arrivées sont régulièrement générées. Pour chaque processus créé, il faut associer un type et un temps de service global au processeur, au disque et à l'imprimante. Pour cela, il est nécessaire de consulter la base de données pour accéder aux distributions adéquates. Ces accès ralentissent fortement le programme. Il est préférable de stocker les différentes distributions en mémoire centrale au début de la simulation.

La quantité de distributions nécessaires varie en fonction du nombre de types de processus différents. Trois distributions sont associées à chaque type de processus. Nous les avons placées en mémoire sous forme d'un tableau à trois dimensions. Les deux premiers indices permettent d'identifier la distribution désirée. Nos conventions sont les suivantes :

tab[i][j] désigne la distribution de service au serveur i pour le processus de type j.

i = 0 => CPU

i = 1 => DISQUE

i = 2 => IMPRIMANTE

On accède à la probabilité cumulée par le champ 'tab[i][j].repartition[k]' et à la valeur par 'tab[i][j].valeur[k]'.

rem : pour les distributions des arrivées et des types des processus, nous utilisons respectivement les variables distr_arr.champ[k] et distr_type.champ[k].

Conception de l'interface

Pour l'élaboration de l'interface, nous avons distingué trois parties. La première consiste à créer un environment qui permet de consulter et mettre à jour la base de données. Ensuite, il faut que l'utilisateur puisse suivre l'évolution de la simulation. Enfin, nous avons conçu les écrans qui affichent les résultats.

En ce qui concerne l'interface de gestion de la base de données, nous avons regroupé toutes les informations concernant le réseau sur un même écran. Ceci permet à l'utilisateur d'en avoir une vision globale sans être obligé pour cela de devoir parcourir plusieurs écrans. De plus, cela diminue le risque d'erreur à l'encodage. En effet, avant d'enregistrer les informations dans la base de données, nous proposons à l'utilisateur de confirmer ses choix. Il peut ainsi vérifier toutes les informations en un coup d'oeil. Nous restons cependant convaincu qu'il n'existe pas de méthode radicale qui supprime les erreurs d'encodage. C'est pourquoi les valeurs subissent un contrôle et ne sont stockées dans la base de données que si elles respectent les contraintes qui y sont liées. De ce fait, nous pouvons affirmer que la base de données reste toujours dans un état cohérent.

En ce qui concerne le suivi de la simulation, nous avons dû tenir compte de la lenteur des fonctions d'affichage. Nous avons de ce fait choisi une interface simple qui permet toutefois à l'utilisateur de suivre l'évolution du comportement du réseau. Au cours de la simulation, nous affichons quatre éléments :

- Le type de traitement effectué (estimation de la période transitoire, vérification de la stationnarité, analyses en régime stationnaire);
- A chaque événement, nous affichons :
  - La taille de chaque file d'attente
  - Le nombre de processus émis par type
  - Le nombre d'observations effectuées

Enfin, nous avons conçu une interface permettant à l'utilisateur de consulter les résultats. L'écran étant trop petit pour y afficher toutes les données, nous avons adopté la technique des menus. Nous proposons à l'utilisateur de choisir le type de résultats qu'il désire consulter (tps de réponse, taux d'utilisation, nombre de processus dans les files,...) et nous avons créé un écran par type de paramètre.

Chapitre 5 :
Prise des mesures

5.1) Introduction

Le système UNIX possède un système de comptabilité qui nous a permis de faire des mesures précises sur l' utilisation du réseau. Nous avons développé un programme qui, à partir des informations recueillies, fournit les données nécessaires à l'exécution du simulateur.

5.2) Description du système de comptabilité sous UNIX[1]

Pour faire fonctionner le système de comptabilité sur les SUN, le noyau du système doit contenir les lignes d'option **SYSACCT** et **PSEUDI-DEVICE SYSACCT.**

SUNOS enregistre deux types d'information de comptabilité : les temps de connexion des utilisateurs et les ressources des processus. L'information de comptabilisation des temps de connexion est stockée dans le fichier /VAR/ADM/WTMP. Le programme **/USR/ETC/AC** fournit un résumé de son contenu. L'information de comptabilisation des ressources des processus est stockée dans le fichier /VAR/ADM/PACCT qui est analysé et résumé par le programme **/USR/ETC/SA.**

Il est possible d'implémenter des procédures basées sur l'information fournie par ces commandes. Un moyen efficace est

[1] Nous nous sommes inspirés de [SUN91] pour rédiger le texte sur la description du système de comptabilité.

d'utiliser le démon de l'horloge **/VAR/ETC/CRON** et de lui demander d'exécuter certaines commandes chaque jour à un moment déterminé. Ceci est réalisé en ajoutant des lignes dans le fichier CRONTAB. Chaque ligne du fichier est composée de 6 champs, séparés par des espaces ou des tabulations.

1. le champ des minutes, dont les valeurs doivent être comprises entre 0 et 59;
2. le champ des heures, dont les valeurs doivent être comprises entre 0 et 23;
3. le jour du mois, dans l'intervalle [1,31];
4. le mois de l'année, dans l'intervalle [1,12];
5. le jour de la semaine, dans l'intervalle [0,6]. Dimanche est le jour 0;
6. le reste de la ligne est la commande à exécuter.

Chaque champ (de 1 à 5) peut être une liste de valeurs séparées par des virgules. Une valeur peut être un nombre ou une paire de nombres séparés par un trait d'union indiquant que la commande doit être exécutée pour tous les moments compris dans l'intervalle spécifié. Si un champ est un '*', cela signifie que la commande doit être exécutée pour toutes les valeurs possibles du champ. La ligne

0 0 1,15 * 1 **SA**

dans le fichier CRONTAB a pour effet de lancer la commande **SA** le premier et le quinze de chaque mois à 00h00 ainsi que chaque lundi à 00h00.

La commande **ACCT** est utilisée pour démarrer ou arrêter le processus de comptabilité. Si celui-ci est mis en marche, un enregistrement de comptabilité est écrit sur le fichier /VAR/ADM/PACCT (fig. 5.1) pour chaque processus qui se termine. Si le processus de comptabilité est déjà mis en marche et qu'un appel **ACCT** est fait avec succès, tous les enregistrements suivants seront écrits dans un nouveau fichier.

Lorsqu'un crash a lieu, aucune comptabilité n'est enregistrée pour les programmes en exécution. De plus, les programmes qui bouclent ne sont jamais pris en compte.
La comptabilité est automatiquement arrêtée s'il n'y a plus de place sur le système de fichiers; elle est redémarrée dès que de la place est à nouveau disponible.

```
char    flag;
char    stat;
short   uid;            /* identificateur de l'utilisateur */
short   gid;            /* identificateur du groupe */
short   tty;
long    b_time;         /* temps début du processus */
short   u_time;         /* temps utilisateur du processus */
short   s_time;         /* temps noyau du processus */
short   e_time;         /* temps total du processus */
short   mem;            /* utilisation moyenne de la mémoire */
short   io;             /* nombre de caractères transférés */
short   rw;             /* nombre de blocs lus/écrits */
char    comm[8];        /* nom du processus */
```

fig. 5.1 : format d'enregistrement du fichier /VAR/ADM/PACCT

Il existe un ensemble de commandes qui permettent une meilleure gestion du système de comptabilité. Nous décrivons celles que nous avons utilisées.

CKPACCT : cette commande devrait être exécutée chaque heure via le **CRON**. Elle vérifie périodiquement la taille du fichier /VAR/ADM/PACCT. Si elle dépasse 1000 blocs (valeur par défaut), la commande **TURNACCT** est exécutée avec l'argument *switch* (cette commande est expliquée plus loin). Si le nombre de blocs disques libres dans le système de fichiers /USR est inférieur à 500, **CKPACCT** arrête automatiquement la collection des enregistrements de

comptabilité des processus. Dès que 500 blocs sont à nouveau disponibles, la comptabilité est réactivée.

TURNACCT : c'est une interface à **ACCTON** qui permet d'activer ou de désactiver le processus de comptabilité. L'argument *switch* désactive la comptabilité, transfère le fichier /VAR/ADM/PACCT vers le nom libre suivant dans /VAR/ADM/PACCT*incr* (où *incr* est un numéro qui commence à 1 et est incrémenté par pas de 1 pour chaque fichier PACCT supplémentaire), et réactive à nouveau la comptabilité.

La procédure **TURNACCT** est appelée par **CKPACCT**. Elle peut être utilisée pour garder le pacct à une taille raisonnable.

La commande **SA** crée des rapports, efface et généralement fait la maintenance du fichier /VAR/ADM/PACCT. Elle est capable de condenser l'information du fichier /VAR/ADM/PACCT dans un fichier résumé /VAR/ADM/SAVACCT qui contient la fréquence d'appel de chaque commande ainsi que les temps d'utilisation des ressources. Cette condensation est préférable car, sur un gros système, le fichier /VAR/ADM/PACCT peut croître de 500 kbytes par jour. La commande **SA** comprend de nombreuses options.

5.3) Elaboration du programme de prise des mesures

5.3.1) Traitement du fichier /VAR/ADM/PACCT

Après avoir analysé les données nécessaires à la simulation, seule la comptabilité des ressources des processus s'est avérée utile. En effet, la comptabilité des temps de connexion concerne les utilisateurs. Or, nous devons simuler les processus créés par les utilisateurs et non le comportement de ces utilisateurs.

Afin de pouvoir utiliser le système de comptabilité, nous avons demandé à l'ingénieur système d'installer le démon **sysacct** dans le noyau qui a dû être recompilé.

Comme vu précédemment, il est possible d'analyser et de résumer le contenu du fichier /VAR/ADM/PACCT en utilisant la commande **/USR/ETC/SA**. Cependant, nous avons besoin des informations les plus précises possible sur le comportement de chaque processus. Nous avons donc décidé de concevoir un programme qui travaille directement sur le contenu du fichier. Les enregistrements sont donc rangés par ordre de terminaison des processus. Afin de pouvoir déterminer plus facilement la distribution des arrivées (nombre de processus créés par unité de temps), nous devons disposer des enregistrements rangés par ordre de commencement des processus. Nous avons donc décidé de trier le fichier /VAR/ADM/PACCT en ordre croissant sur la variable *b_time* (temps de début) de chaque processus.

Pour cela, nous avons utilisé la commande **SORT** du système d'exploitation Unix. Vu qu'elle ne travaille que sur des fichiers textes, nous avons tout d'abord converti le fichier binaire /VAR/ADM/PACCT en un fichier texte. Ce tri nous a permis de simplifier considérablement l'algorithme du programme des prises de mesures.

b time	u time	s time	e time	rw	commande
699181383	3	8	28	6	cp
699181384	0	4	9	4	ln
699181384	4	3	11	2	chmod
699181384	2	10	20	17	sync
699181384	2	10	27	8	sync
699181385	2	10	64	2	awk
699181385	4	16	65	11	df
699181385	2	13	70	2	egrep
699181386	2	8	23	1	sed
699181381	5	14	382	9	sh

Fichier /VAR/ADM/PACCT non trié

b time	u time	s time	e time	rw	commande
699181381	5	14	382	9	sh
699181383	3	8	28	6	cp
699181384	0	4	9	4	ln
699181384	2	10	20	17	sync
699181384	2	10	27	8	sync
699181384	4	3	11	2	chmod
699181385	2	10	64	2	awk
699181385	2	13	70	2	egrep
699181385	4	16	65	11	df
699181386	2	8	23	1	sed

Fichier /VAR/ADM/PACCT trié

5.3.2) Choix réalisés

Nous devons déterminer la distribution des temps de service aux différents serveurs (CPU, DISQUE et IMPRIMANTE) par type de processus.

En effet, nous avons remarqué la présence de deux types différents de requête : PC et Unix. Cette distinction est nécessaire car la quantité de ressources utilisées varie en fonction du type. Les requêtes PC ont, en général, un petit temps de service global au serveur CPU et un petit nombre d'E/S tandis que les requêtes Unix peuvent avoir un grand temps de service global au serveur CPU et un grand nombre d'E/S.
Ceci implique le calcul du pourcentage des requêtes PC et Unix ainsi que le calcul de distributions séparées.

Afin de pouvoir tenir compte de l'imprimante, nous devons encore prendre en considération un élément supplémentaire. Il serait en effet utile de différencier les requêtes d'impression et les requêtes normales

(pas d'impression) aussi bien au niveau PC qu'au niveau Unix. Il faut savoir que chaque requête d'impression provoque la création de trois processus (Rpc.pcnf, Lpr ,Lpd) du côté PC et de deux processus (Lpr, Lpd) du côté Unix.
Rpc.pcnf résulte de la demande d'impression du côté PC. Le démon **nfsd** traduit alors cette demande en un processus Unix Lpr. Celui-ci envoie une copie du fichier à imprimer dans une 'spool area' qui est gérée par le démon **lpd**. Ce dernier, à chaque requête d'impression, associe un processus Lpd qui va envoyer les données du fichier à l'imprimante (fig 5.2).

Pour les serveurs CPU et DISQUE, nous allons considérer, comme temps de service global d'une requête d'impression, le temps nécessaire aux données pour arriver au buffer de l'imprimante ((Rpc.pcnf +) Lpd + Lpr) (fig 5.3 et fig 5.4). Pour rappel, les informations qui permettent d'établir le temps d'impression ne sont pas fournies par le système de comptabilité de Unix.

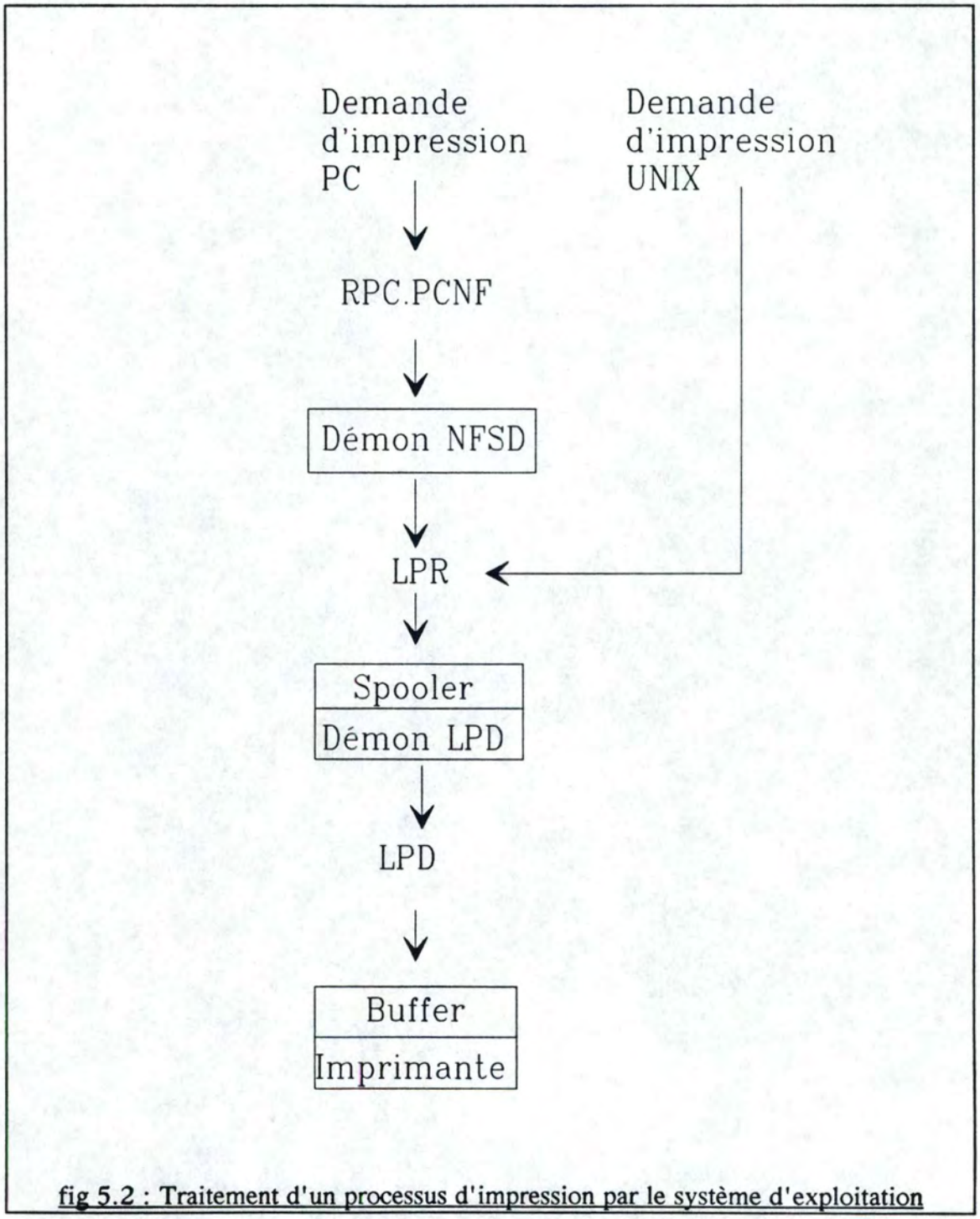


fig 5.2 : Traitement d'un processus d'impression par le système d'exploitation

699091904	1	16	22	4	rpc.pcnf
699091904	4	11	88	17	lpr
699091906	1	11	1871	18	lpd
DEVIENT					
699091904	6	38	1981	39	Impression PC

fig 5.3 : Transformation d'un processus d'impression PC

699092868	4	20	103	17	lpr
699092869	2	25	3071	26	lpd
DEVIENT					
699092868	6	45	3174	43	Impression Unix

fig 5.4 : Transformation d'un processus d'impression Unix

En conclusion, nous avons quatre distinctions :

* les demandes normales PC (rpc.pcnf)
* les demandes normales Unix (sh,rm,vi,...)
* les demandes d'impression PC (rpc.pcnf,lpr,lpd)
* les demandes d'impression Unix (lpr,lpd)

Nous devons créer une distribution des temps de service globaux au serveur CPU et du nombre d'E/S pour chacun des quatre cas.

La distribution des arrivées doit permettre de déterminer le nombre de processus à générer par intervalle de temps. Nous avons d'abord opté pour des intervalles de trente secondes. Ce choix ne s'est pas avéré judicieux. En effet, l'unité des temps de service étant de un soixantième de seconde, il est peu réaliste de générer des flots d'arrivées toutes les trente secondes. Il est en effet préférable de pouvoir répartir les arrivées de processus de façon plus homogène. Nous avons pris des intervalles d'une seconde (la plus petite unité possible) et, au vu des résultats, cela se révèle être un bon choix.

Nous avons décidé de créer les distributions pour chaque heure de manière à pouvoir découvrir la répartition des processus dans la journée

(la matinée est généralement plus dense et on remarque la présence d'un temps mort lors du dîner).

5.3.3) Représentation des résultats

Un premier fichier (TABJOUR) (fig 5.5) contient les informations nécessaires à l'élaboration des différentes distributions (des arrivées, des temps de service globaux au serveur au CPU, du nombre de blocs lus ou écrits sur le serveur DISQUE).

```
int   nb_ar_ut[MAX_AR];          /*  Distr. des arrivées  */
int   tps_cpu_pc[MAX_SER];       /*  Distribution         */
int   tps_cpu_un[MAX_SER];       /*    des services       */
int   tps_cpu_pc_imp[MAX_SER];   /*                       */
int   tps_cpu_un_imp[MAX_SER];   /*                       */
int   nb_io_pc[MAX_SER];         /*  Distributions        */
int   nb_io_un[MAX_SER];         /*     des E/S           */
int   nb_io_pc_impMAX_SER];      /*                       */
int   nb_io_un_imp[MAX_SER];     /*                       */
int   tps_cpu_150_pc;            /*  cumul des            */
int   nb_io_30_pc;               /*   tranches            */
int   tps_cpu_150_un;            /*  [150,infini]         */
int   nb_io_30_un;               /*       et              */
int   tps_cpu_150_pc_imp;        /*  [30,infini]          */
int   nb_io_30_pc_imp;           /*                       */
int   tps_cpu_150_un_imp;        /*                       */
int   nb_io_30_un_imp;           /*                       */
```

Rem : MAX_AR = 11 (pas = 1 arrivée)
MAX_SER = 31 (pas = 5/60 seconde)

fig 5.5 : Structure d'un enregistrement du fichier TABJOUR

Chaque distribution a la forme suivante :

Valeurs	x0	x1	x2	.....	xi	.....	xn
Observations	r0	r1	r2	.....	ri	.....	rn

Elle se lit :

- on a observé r_0 fois la valeur x_0
- on a observé r_1 fois la valeur x_1
- .....
- on a observé r_i fois la valeur x_i
- .....
- on a observé r_n fois la valeur x_n

Elle est représentée par un tableau de $n+1$ éléments. Le nombre total d'observations n équivaut à la somme des r_i pour i allant de 0 à n.

Avant de commencer les prises des mesures, il faut d'abord déterminer les valeurs des x_i (la valeur de x_0 et le pas entre x_i et x_{i+1}) et de n pour chaque distribution.

La distribution des arrivées est représentée par le tableau *nb_ar_ut* (**nombre d'arrivées par unité de temps**). Comme vu précédemment, l'intervalle de temps (T) est de 1 seconde. Le choix de n est lié à celui de l'intervalle de temps T. Il dépend également de la charge du réseau. En effet, plus le réseau est chargé, plus le nombre d'arrivées par intervalle de temps est grand. Par contre, si on diminue la taille de l'intervalle, cela réduit le nombre d'arrivées (30 arrivées toutes les 10 secondes devient 3 arrivées toutes les secondes).
Nous avons choisi un pas de 1 entre x_i et x_{i+1} de manière à obtenir la meilleure précision possible. L'élément i du tableau indique donc le nombre de fois qu'on a observé i arrivées par intervalle de temps.

$x_0 = 0$
$x_1 = 1$
.....
$x_n = n$

Par mesure de sécurité, nous avons fixé *n* à 31. Après analyse d'un fichier /VAR/ADM/PACCT, le nombre maximum d'arrivées observées sur une seconde est 6. Nous pouvons donc supposer qu'il ne se produit jamais 30 arrivées sur une seconde.

Les valeurs choisies pour la distribution des arrivées sont donc :

$T = 1$ seconde

$n = 31$

$x_0 = 0$

$x_i = x_{i-1} + 1$

Les quatre distributions des temps de service au serveur CPU sont reprises dans les tableaux *tps_cpu_pc* (**temps** de service **CPU** pour les processus **PC**), *tps_cpu_un* (**temps** de service **CPU** pour les processus **Unix**), *tps_cpu_pc_imp* (**temps** de service **CPU** pour les processus **PC** d'**impression**), *tps_cpu_un_imp* (**temps** de service **CPU** pour les processus **Unix** d'**impression**). Elles ont toutes les mêmes paramètres. Chaque x_i représente un intervalle de temps. Le nombre d'éléments *n* dépend de la précision désirée et des caractéristiques des processus (utilisation du processeur).

Après avoir analysé un fichier /VAR/ADM/PACCT, nous avons remarqué que les temps de service globaux au serveur CPU sont bornés. Ils dépassent rarement 100/60 secondes. Nous avons donc décidé de travailler avec un petit nombre d'intervalles de temps (n=31). Des intervalles de temps de 5/60 secondes procurent une précision suffisante. L'élément *i* indique le nombre de processus qui ont nécessité un temps de service global au serveur CPU compris dans l'intervalle de temps [5*i , 5*(i+1)].

Les valeurs choisies pour les distributions des temps de service CPU sont donc :

$n = 31$

$x_0 = [0 , 5]$

$x_i = [5*i , 5*(i+1)]$

Les quatre distributions du nombre de blocs lus ou écrits (E/S=i/o) sont représentées par les tableaux *nb_io_pc* (**nombre d'i/o** pour les processus **PC**), *nb_io_un* (**nombre d'i/o** pour les processus **Unix**), *nb_io_pc_imp* (**nombre d'i/o** pour les processus **PC** d'**impression**), *nb_io_un_imp* (**nombre d'i/o** pour les processus **Unix** d'**impression**). Elles ont toutes les mêmes paramètres. Le nombre d'éléments *n* dépend de la précision désirée et des caractéristiques des processus (utilisation du disque). Après avoir analysé un fichier /VAR/ADM/PACCT, nous avons remarqué que le nombre d'E/S réalisées par les processus est souvent petit (inférieur à 20). Nous n'avons donc pas besoin d'un grand *n* (n=31). L'élément *i* indique le nombre de processus qui ont réalisé *i* E/S.

Les valeurs choisies pour les distributions du nombre de blocs lus ou écrits sont donc :

$n = 31$

$x_0 = 0$

$x_i = i$

Rem : pour rappel, les distributions des temps de service globaux au serveur CPU et celles du nombre de blocs lus ou écrits portent sur une durée d'une heure de manière à obtenir la répartition horaire.

Certains processus ont un grand temps de service global au serveur CPU et/ou un grand nombre d'E/S. Nous avons décidé que le dernier élément des distributions des temps de service au serveur CPU représente la tranche [150,infini] et le dernier élément des distributions du nombre de blocs lus ou écrits, la tranche [30,infini]. Les variables associées aux distributions de service du serveur (*tps_cpu_150_pc*, *tps_cpu_150_un*, *tps_cpu_150_pc_imp*, *tps_cpu_150_ un_imp*) contiennent le cumul des temps de service globaux de la tranche [150,infini] tandis que les variables associées aux distributions de service global du serveur DISQUE (*nb_io_30_pc*, *nb_io_30_un*, *nb_io_30_pc_imp*, *nb_io_30_un_imp*) contiennent le cumul du nombre de blocs lus ou écrits de la tranche [30,infini]. Il est alors possible d'obtenir un temps moyen de

service global au serveur CPU (tps_cpu_cumule/nb_processus) ainsi qu'un nombre moyen d'E/S (i/o) (nb_io_cumule/nb_processus) pour cette tranche.

Un second fichier (TABPCJOUR) (fig 5.6) permet de déterminer la distribution des types des processus. Le nombre total de processus (*nb_tot_p*) par heure donne la répartition horaire des arrivées. Nous avons besoin du nombre de processus de type PC normal (*nb_proc_pc*), de processus de type Unix normal (*nb_proc_un*), de processus de type d'impression PC (*nb_proc_pc_imp*) et de processus de type d'impression Unix (*nb_proc_un_imp*). Pour cela, le nombre total de processus (*nb_tot_p*), le nombre de requêtes PC (*nb_rpc_p*), le nombre de requêtes d'impression PC (*nb_lpr_pc*) et le nombre de requêtes d'impression Unix (*nb_lpr_un*) suffisent. Il convient d'effectuer les calculs suivants :

```
nb_proc_pc = nb_rpc_p - nb_lpr_pc
nb_proc_un = nb_tot_p - nb_rpc_p - nb_lpr_un
nb_proc_pc_imp = nb_lpr_pc
nb_proc_un_imp = nb_lpr_un
```

```
int  nb_tot_p;  /* nombre total de processus */
int  nb_rpc_p;  /* nombre de requêtes PC-NFS */
int  nb_lpr_pc; /* nombre de processus d'impression PC */
int  nb_lpr_un; /* nombre de processus d'impression Unix */
int  nb_lpd_tot; /* nombre de processus Lpd */
```

fig 5.6 : Structure d'un enregistrement du fichier TABPCJOUR

A partir du fichier /VAR/ADM/PACCT (fig 5.7), nous créons donc les fichiers TABJOUR et TABPCJOUR.

Champs	Unité	Type	Utilisation
b_time	seconde	LONG	distr. des arrivées
u_time + s_time	1/60 seconde	INT	distr. de service CPU
rw	block	INT	distr. de service DISQUE
comm	caractères	CHAR[8]	détermination type processus

fig 5.7 : données utilisées du fichier /VAR/ADM/PACCT

5.3.4) Programme des prise des mesures

Après avoir observé l'utilisation du réseau pendant quelques jours, il s'est avéré pertinent de récolter les données entre 9 h. et 17 h. Il faut donc détecter dans le fichier /VAR/ADM/PACCT à partir de quel processus la prise des mesures doit commencer.
La commande **SA**, utilisée avec le paramètre *-s*, permet de réinitialiser le fichier /VAR/ADM/PACCT à zéro. Nous avons donc demandé au gestionnaire du système de lancer cette commande à 9h00 précise via le **CRON** (fig 5.8). De cette manière, le temps de début (*b_time*) du **SA** est le temps de référence (t0) qui correspond à 9h00. Vu que nous travaillons sur le fichier /VAR/ADM/PACCT trié sur le temps de début des processus, tous ceux qui précèdent le processus **SA** ne sont pas pris en compte (fig 5.9).

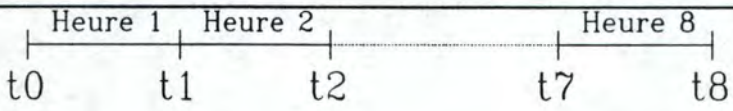
```
0  9  *  *  *  /usr/etc/sa -s
```

fig 5.8 : entrée du **CRON**

b time	u time	s time	e time	rw	comm	décision
698687511	124	110	37556	29	lockscreen	abandonné
698751033	20483	9535	36626	5	lockscreen	abandonné
699004800	5	13	2654	3	sh	abandonné
699004801	**1813**	**749**	**2619**	**22**	**sa**	**9h00**
699004861	15	9	26	0	rpc.pcnf	compté
699004864	3	10	19	2	rpc.pcnf	compté
699004865	3	10	17	2	rpc.pcnf	compté
699004980	0	9	11	3	cp	compté
699004980	10	17	166	0	sh	compté
699004980	4	15	200	3	sh	compté
699004981	0	3	5	2	ln	compté

fig 5.9 : Détermination des processus à prendre en compte

Comme vu précédemment, nous créons un enregistrement dans le fichier TABJOUR et dans le fichier TABPCJOUR par heure traitée (8 enregistrements par jour).
Tous les processus dont le temps de début est compris entre t_0 et $t_0 + 3600$ sec (t_1) sont pris en compte pour les distributions de la première heure, ceux dont le temps de début est compris entre t_1 et $t_1 + 3600$ sec (t_2) pour la deuxième heure,... (fig 5.10)



t0 = 'b_time' du SA
t1 = t0 + 3600
t2 = t1 + 3600
.....
t7 = t6 + 3600
t8 = fin des mesures de la journée

fig 5.10 : Temps de début des heures

Pour rappel, à chaque fois qu'un processus se termine, le démon sauve des informations dans le fichier /VAR/ADM/PACCT. Sa croissance est de l'ordre de plus ou moins 500 K par jour. Nous le traitons donc tous les jours afin d'éviter une occupation trop importante de l'espace disque (nous disposons d'un espace disque limité; la société a commandé un second disque qui n'arrivera qu'en début 1992).

Algorithme général

```
début du traitement
  {
  positionnement sur le processus SA
  tps deb_jour = b_time du SA ( = 9h00)
  tps fin_jour = tps deb_jour + 8 * 3600sec  ( = 17h00)
  tps deb_heure = tps deb_jour
  tps fin_heure = tps deb_heure + 3600sec
  lecture du premier processus
  tant que ( tps deb_heure < tps fin_jour )
    {
    /* Traitement d'une heure */
    tant que ( b_time < tps fin_heure ) AND ( NOT fin fichier )
      {
      traitement du processus
      lecture du processus suivant
      }
    écriture des deux enregistrements
    /* Passage à l'heure suivante */
    tps deb_heure = tps deb_heure + 3600sec
    tps fin_heure = tps fin_heure + 3600sec
    }
  }
fin du traitement
```

Glossaire des variables

tps deb_heure : indique l'heure de début de la période (heure) traitée
tps deb_jour : indique l'heure de début de la journée traitée
tps fin_heure : indique l'heure de la fin de la période (heure) traitée
tps fin_jour : indique l'heure de la fin de la journée traitée

Comme le montre l'algorithme général, nous traitons les processus du fichier /VAR/ADM/PACCT de manière chronologique. Nous vérifions que le processus lu appartient à l'heure en cours (*b_time* < *tpsfin_heure*). Si c'est le cas, nous pouvons commencer son traitement. Sinon, nous clôturons l'heure en cours (écriture des enregistrements), nous initialisons l'heure suivante et nous recommençons la procédure (vérification,...).

Le traitement d'un processus consiste tout d'abord à déterminer son type à partir du nom de la commande (*comm[8]*).
Si le processus lu est du type PC (*comm* = "rpc.pcnf"), nous regardons si le processus suivant est un Lpr (ou un Lpd).
Si ce n'est pas le cas, nous considérons le Rpc.pcnf comme une demande normale d'accès mémoire (lecture ou écriture). Nous analysons son temps de service global au serveur CPU (*u_time*+*s_time*) et le nombre de blocs lus ou écrits (*rw*) de manière à incrémenter d'une unité le bon élément des deux tableaux associés au type : *tps_cpu_pc* et *nb_io_pc*. Nous augmentons également les variables *nb_tot_p* (nombre total de processus) et *nb_rpc_p* (nombre de requêtes PC-NFS) d'une unité.
Sinon, nous cherchons le Lpd (ou Lpr) associé au Lpr (ou Lpd) précédent. Tous les processus lus ne répondant pas à notre critère de recherche sont traités au fur et à mesure de leur apparition. Après cinq échecs (échec = processus différent du Lpd (ou Lpr) cherché), nous abandonnons la recherche et traitons la requête d'impression comme une requête normale. Par contre, si on trouve le Lpd (ou Lpr) associé au Lpr (ou Lpd), nous créons une requête d'impression en cumulant les données des trois processus. Nous analysons le temps de service global cumulé au serveur CPU et le nombre de blocs lus ou écrits cumulés de manière à incrémenter le bon élément des deux tableaux associés au type :

tps_cpu_pc_imp et *nb_io_pc_imp*. Nous augmentons également les variables *nb_tot_p* (nombre total de processus), *nb_rpc_p* (nombre de requêtes PC-NFS), *nb_lpr_pc* (nombre d'impressions PC) et *nb_lpd_tot* (nombre de processus Lpd) d'une unité.

Si le processus lu est du type Unix (*comm* NOT = "rpc.pcnf"), nous regardons si c'est un Lpr. Si ce n'est pas le cas, le processus est du type Unix (normal). Nous analysons son temps de service global au serveur CPU (*u_time+s_time*) et le nombre de blocs lus ou écrits (*rw*) de manière à incrémenter le bon élément des deux tableaux associés au type : *tps_cpu_un* et *nb_io_un*. Nous augmentons également la variable *nb_tot_p* (nombre total de processus) d'une unité.
Sinon, nous cherchons le Lpd associé au Lpr. Tous les processus lus ne répondant pas à notre critère de recherche sont traités au fur et à mesure de leur apparition. Après cinq échecs (échec = processus différent du Lpd cherché), nous abandonnons la recherche et traitons la requête d'impression comme une requête normale. Par contre, si on trouve le Lpd associé au Lpr, nous créons une requête d'impression en cumulant les données des deux processus. Nous analysons le temps de service global cumulé au serveur CPU et le nombre de blocs lus ou écrits cumulés de manière à incrémenter le bon élément des deux tableaux associés au type : *tps_cpu_un_imp* et *nb_io_un_imp*. Nous augmentons également les variables *nb_tot_p* (nombre total de processus), *nb_lpr_un* (nombre d'impressions Unix) et *nb_lpd_tot* (nombre de processus Lpd) d'une unité.

Au niveau des arrivées, pour chaque processus lu, nous comparons son temps de début à celui du processus précédent. S'il est identique, cela signifie qu'on est toujours dans la même seconde. Nous incrémentons donc le compteur du nombre d'arrivées pour la seconde en cours et nous continuons le traitement.

Par contre, s'il est différent, cela veut dire que nous ne sommes plus dans la même seconde. Nous sauvons le compteur courant du nombre d'arrivées dans le tableau *nb_ar_ut* et le remettons à zéro. Nous déterminons le nombre de secondes qui se sont écoulées entre les deux

processus et pendant lesquelles aucune arrivée n'a eu lieu. Nous ajoutons ce nombre au contenu du premier élément du tableau *nb_ar_ut* et nous initialisons le compteur de la nouvelle seconde courante à 1.

Chapitre 6 :
Validation du simulateur

6.1) Introduction

Le modèle de simulation est une abstraction du système réel que nous avons étudié. Nous avons dû faire une série d'approximations de la réalité lors de la création du modèle. Nous devons donc faire une validation des résultats obtenus c'est-à-dire déterminer s'ils sont représentatifs du système réel.
Il n'existe pas de technique idéale de validation. Parmi les approches fréquemment utilisées, nous en citons quatre [H&P89] :

- la comparaison des résultats du modèle avec ceux du système réel
- la méthode Delphi
- le Test de Turing
- le comportement aux extrêmes

La première méthode est la plus souvent suggérée. Il s'agit d'une comparaison statistique. Les différences dans les mesures de performances doivent être testées statistiquement. Il faut néanmoins disposer des données suffisantes sur les performances du système réel.

Les méthodes telles que la méthode Delphi, le Test de Turing et le comportement aux extrêmes demandent la participation d'experts tels des analystes, des gestionnaires de système, ...

Dans la méthode Delphi, un groupe d'experts est choisi. Il est composé de gestionnaires et d'utilisateurs du système modélisé. Une série de questions est posée à chacun d'entre eux concernant les comportements ou les performances du système sous des conditions spécifiques d'exécution. Sur base des réponses, de nouvelles questions sont formulées de manière à obtenir des réponses plus spécifiques. On répète la

procédure jusqu'à ce que l'analyste possède une prédiction du temps de réponse selon les entrées considérées.

En ce qui concerne le test de Turing, des descriptions sommaires basées sur le système réel et sur le modèle de simulation sont présentées à un expert ou un groupe d'experts. S'ils ne savent pas identifier les rapports basés sur les résultats du modèle de simulation, la crédibilité du modèle augmente.

Pour réaliser une validation par le comportement aux extrêmes, le système réel est observé sous des conditions extrêmes. Cela peut être une situation idéale pour rassembler des données sur les mesures de performances du système réel. Il est parfois plus facile pour les gestionnaires du système de prédire le comportement du système réel sous des conditions extrêmes. En comparant les prédictions du comportement du système sous des conditions extrêmes aux performances du modèle sous les mêmes conditions, on peut augmenter la crédibilité du modèle.

6.2) Applicabilité des méthodes à notre simulateur

Vu que nous avons testé le générateur aléatoire du Borland C++ et que nous utilisons les distributions des temps de service globaux et des arrivées que nous avons mesurées à Hamm, nous pouvons affirmer que ces distributions correspondent à la réalité. Nous sommes donc déjà certains que les taux d'occupation des serveurs sont corrects. Les temps d'attente et de réponse moyens des processus ainsi que le nombre moyen de processus à la file d'attente de chaque serveur dépendent des hypothèses que nous avons prises sur le fonctionnement du système d'exploitation qui gère le réseau. Il faudrait donc pouvoir valider ces choix.

Pour cela, il faut que nous disposions des résultats réels qui nous permettent d'appliquer une des quatre méthodes ou que un ou plusieurs experts valident nos choix quant à la simulation du système d'exploitation (ordonnancement des processus, calcul des priorités, ...).

Vu que nous n'avons pas d'expert à notre disposition, la seule solution possible pour valider le simulateur consiste à mesurer les performances du système réel. Cela doit se faire via le système d'exploitation UNIX ou via un logiciel spécialisé du style du Lan Probe.

Comme il a été signalé au chapitre cinq, le système de comptabilité sous Unix permet d'enregistrer des informations sur les processus dans le fichier /VAR/ADM/PACCT (fig5.1). Ce sont les seules données disponibles sur la vie des processus. Après une analyse du contenu de ce fichier, il semble que la variable *e_time* (temps total ou temps de réponse du processus) devrait nous permettre de valider le simulateur.

Un problème apparaît cependant pour certains processus. La formule que nous utilisons pour calculer le temps de réponse d'un processus consiste à additionner son temps de service global et son temps d'attente au serveur CPU et au serveur DISQUE. On ne tient pas compte des attentes provoquées par des événements extérieurs tels :

- plus de page à l'imprimante;
- synchronisation avec un processus;
- attente d'une réponse d'un utilisateur (éditeurs, programmes interactifs,...);
- .....

La variable *e_time* de chaque processus touché par ces événements extérieurs ne correspond pas au temps de réponse que nous calculons. On ne peut donc pas l'utiliser pour faire notre validation.

Une solution possible consisterait à dresser la liste complète des processus touchés par les événements extérieurs. Pour cela, il serait nécessaire de faire appel à un expert. Il faudrait également disposer d'un réseau dont nous serions l'unique utilisateur. Il conviendrait alors de créer uniquement des processus qui ne sont pas repris sur la liste établie par l'expert et de prendre les mesures sur le système. De cette manière, nous disposerions du temps de réponse recherché pour chaque processus via la variable 'e_time'. Il serait alors possible de valider le simulateur. Cependant, nous ne disposons pas d'un expert et nous n'avons pas accès à un réseau inoccupé.

Pour rappel, le Lan Probe mis à notre disposition à l'Institut d'Informatique permet de "regarder" tout ce qui passe sur la ligne et de consulter les informations présentes au niveau deux du modèle OSI. Il existe des options qui permettent de monter plus haut dans les couches du modèle OSI. Il serait alors peut-être possible de réaliser des mesures sur les temps de réponse des paquets (temps passé dans le réseau) car ceux-ci seraient plus facilement identifiables (ce qui n'est pas le cas au niveau deux). Malheureusement, ces options ne sont pas disponibles sur la version du Lan Probe que nous avons utilisée.

Dans le cas présent, nous sommes donc dans l'impossibilité de valider nos résultats. Nous savons seulement regarder s'ils sont plausibles.

Chapitre 7 : Conclusions

7.1) Résultats de la simulation

Comme il a été démontré au chapitre précédent, nous ne sommes pas en mesure de valider les résultats obtenus. Nous pouvons seulement vérifier leur cohérence. Nous avons effectué dix simulations avec les mêmes paramètres en entrée à savoir les distributions observées sur le réseau de Hamm. Les dix intervalles de confiance obtenus pour chaque résultat désiré sont tous dans le même ordre de grandeur. Nous pouvons donc en conclure que le simulateur n'a pas de comportement imprévisible.

Voici la liste des résultats obtenus lors d'une simulation sous forme d'intervalles de confiance avec un niveau d'incertitude de 0.05 (l'unité des temps est la milliseconde) :

Type du résultat	Moyenne	Ecart-type	Borne inf.	Borne sup.	Nb obs.
Temps de réponse moyen des processus	2008	1063	1756	2260	2500
Temps d'attente moyen au processeur	158	2711	103	212	2500
Temps d'attente moyen au disque	88	2290	46	130	2500
Temps d'attente moyen à l'imprimante	519	723	269	769	2500
Nombre moyen de processus dans la file d'attente du processeur	0.012	0.0458	0.04	0.019	2500
Nombre moyen de processus dans la file d'attente du disque	0.047	0.0047	0	0.101	2500
Nombre moyen de processus dans la file d'attente de l'imprimante	0.051	0.1586	0	0.101	2500

Temps de réponse moyen des processus de type PC	555	173	498	612	1250
Temps de réponse moyen des processus de type Unix	2285	1344	1854	2716	2500
Temps de réponse moyen des processus de type impression PC	2689	668	2453	2925	750
Temps de réponse moyen des processus de type impression Unix	2229	361	2177	2280	250

Taux d'utilisation du processeur	9.31%
Taux d'utilisation du disque	2.53%
Taux d'utilisation de l'imprimante	18.81%

Le taux d'utilisation des différents serveurs est assez faible. Ceci s'explique par le taux d'arrivée au réseau. En effet, en moyenne, une arrivée est générée toutes les 20 secondes. Il est donc logique que les serveurs soient peu utilisés quand on sait que le temps de service global moyen au processeur est de 1,5 secondes et au disque de 0,3 seconde.
En ce qui concerne l'imprimante, un processus d'impression arrive en moyenne toutes les 7,5 minutes avec un temps de service global moyen de 2 minutes. (Ces données proviennent des mesures effectuées à Hamm)
Il en résulte que le nombre moyen de processus dans les différentes files d'attente est négligeable.

Toutes ces données confirment que le réseau simulé est très peu chargé. Des stations peuvent être ajoutées sans que les utilisateurs habituels ne soient incommodés.

Il est bon de souligner l'importance de la distinction des types de processus. En fonction de l'utilisation de la station, les types de processus émis sont différents. Les résultats obtenus montrent que les temps de réponse varient fortement en fonction du type de processus.
Lorsque l'on installe un réseau, il est donc préférable de savoir quels types d'utilisateur seront connectés.

Le temps de réponse moyen des processus paraît assez élevé (2sec). Ceci est dû au mélange des différents types. En effet, on peut constater que le temps de réponse moyen d'une requête de type PC est très bon et que les requêtes de type Unix influencent fortement la moyenne générale.

7.2) Extensions possibles

Comme il a été expliqué au chapitre précédent, le modèle de simulation doit encore être validé. Nous pouvons seulement affirmer que les résultats obtenus sont plausibles.

Notre programme simule un réseau Ethernet composé d'un processeur central, un disque partagé, une imprimante et plusieurs stations. Notre modèle ne prévoit pas la présence de plusieurs serveurs de même type (CPU, DISQUE ou IMPRIMANTE). Il peut être intéressant de supprimer cette limite. L'approche suivie lors de la conception du simulateur devrait faciliter l'exécution de ce travail. En effet, toutes les primitives nécessaires existent déjà. Il faut cependant étudier en profondeur la gestion en parallèle des serveurs de même type.
Au niveau des processeurs, les files d'attente sont gérées de manière dynamique selon la technique du "Load Balancing" [ZHO88].

7.3) Conclusions

Nous avons, lors de notre travail, effectué une étude approfondie du système d'exploitation UNIX et du système de fichiers distribués NFS. Nous avons pu, à cet effet, bénéficier d'un espace de travail sur le réseau Ethernet de Hamm. Une fois familiarisés avec le réseau, nous l'avons modélisé. Notre stage se terminant au mois de janvier 1992, nous avons dû effectuer les mesures avant la conception du programme de simulation. Nous utilisons les résultats de nos mesures comme paramètres du simulateur. Nous avons également développé une base de données qui contient toutes les informations nécessaires à la simulation. Une fois le

programme terminé, nous avons effectué plusieurs simulations du réseau de Hamm afin de vérifier la cohérence des résultats.

Les domaines de l'informatique et des télécommunications progressent très rapidement. Il est prévisible que le programme que nous avons conçu ne soit plus applicable à la prochaine génération de réseaux locaux. Cependant, lors de la conception du programme, nous avons essayé d'isoler les éléments et les traitements effectués en vue de faciliter les éventuelles modifications à apporter au logiciel.

Dans sa configuration actuelle, notre programme permet de simuler les réseaux composés d'un serveur central qui gère le partage d'un disque et d'une imprimante quel que soit le système de fichiers distribués utilisé. Cela est différent lorsque le réseau est constitué de plusieurs stations de travail. Il est dès lors nécessaire de réétudier le comportement du réseau pour modifier le simulateur en fonction des résultats désirés et des informations disponibles (prise de mesures).

Annexe A

Le théorème de Doob-Birkhov énoncé ci-dessous est issu de [NOI91].

Si $\{\underline{X}(t)\}$ est un processus strictement stationnaire et si $\xi\underline{X}(t)$ est finie, alors la moyenne ergodique

$$\underline{Z_T} = \frac{1}{T} \int_0^T \underline{X}(t)\ dt$$

converge presque sûrement pour $T \to \infty$

Si de plus la variance σ^2 est finie et si

$$\lim_{h \to \infty} \gamma(h) = 0$$

$$\text{avec } \gamma(h) = \text{cov}(\underline{X}(t), \underline{X}(t+h))$$

alors la moyenne ergodique converge vers la moyenne stochastique :

$$\Pr\left[\lim_{T \to \infty} Z_T = \xi\underline{X}\right] = 1$$

Ce théorème nous permet d'estimer $\xi\underline{X}$ à partir d'une seule simulation, en observant la variable $\underline{X}$ au cours de cette simulation.

Bibliographie

[BAC86] Maurice J. Bach, "The design of the UNIX operating system, Prentice-Hall, 1986.

[BFS83] Paul Bratley, Bennet L. Fox, Linus E. Schrage, "A guide to simulation", Springer-Verlag, 1983.

[BOD89] F. Bodard, Y. Pigneur, "Conception assistée des systèmes d'information", MASSON, Presses universitaires de Namur, 1989.

[CAR91] G. Cardinael, Cours de gestion des ressources informatiques, Institut d'informatique FUNDP, Namur, 1991-1992.

[CON63] R. Conway, "Some tactical problems in digital simulation", in : *Management Sciences*, vol. 6, p. 92-110, 1963.

[DUB90] E. Dubois, Cours de méthodologie de développement logiciel, Institut d'informatique FUNDP, Namur, 1990-1991.

[FIS73] G. Fishman, "Statistical analysis for queuing simulation", in : *Management Sciences*, vol. 20, p. 275, 1973.

[HAI86] J-L. Hainaut, "Conception assistée des applications informatiques : conception de la base de données", Masson, Presses universitaires de Namur, 1986.

[HOW89] Howard W. Sams & company, "Microsoft C : programming for the PC", Robert Lafore, 1989.

[H&P89] Stewart V.Hoover, Ronald F.Perry, "Simulation : a problem solving approach", Addison Wesley,1989.

[H&W83] P. Heidelberger, P. Welch, "Simulation run length control in the presence of an initial transient", in : *Operation Research*, vol. 31, p. 1109-1144, 1983.

[NOI91] M. Noirhomme-Fraiture, Notes du cours de simulation, Institut d'informatique FUNDP, Namur, 1991-1992.

[PAW90] K. Pawlikowski, "Steady-state simulation of queuing processes : a survey of problems and solutions", in : *ACM Computing Surveys*, vol 22, No2, p.123-170, Juin 1990.

[P&M88] Stephen K. Park, Keith W. Miller, "Random number generators : good ones are hard to find", in : *Communications of the ACM*, vol 31, No 10, Octobre 1988.

[SHR83] Schruben, L. W., "Confidence interval estimation using standardized time series", in : *Operation research*, vol 30, 1983.

[SUN91] Sun OS User Manual, Sun Corporation, 1991.

[TAN87] Andrew S. Tanenbaum, "Operating systems : design and implementation", Prentice-Hall, 1987.

[VAN90] Ph. Van Bastelaer, Cours de téléinformatique et réseaux : fonctions et concepts, Institut d'informatique FUNDP, Namur, 1990-1991.

[ZHO88] S. Zhou, "A trace-Driven Simulation Study of Dynamic Load Balancing" in : *IEEE Transactions on Software Engineering*, vol.14, n°9, Septembre 1988.

Facultés Universitaires Notre-Dame de la Paix
Institut d'Informatique

Rue Grandgagnage, 21
B-5000 NAMUR (Belgium)

Simulation d'un réseau local Ethernet utilisant un système de fichiers distribués

Annexes

Mémoire présenté en vue de l'obtention
du diplôme de Maître en Informatique

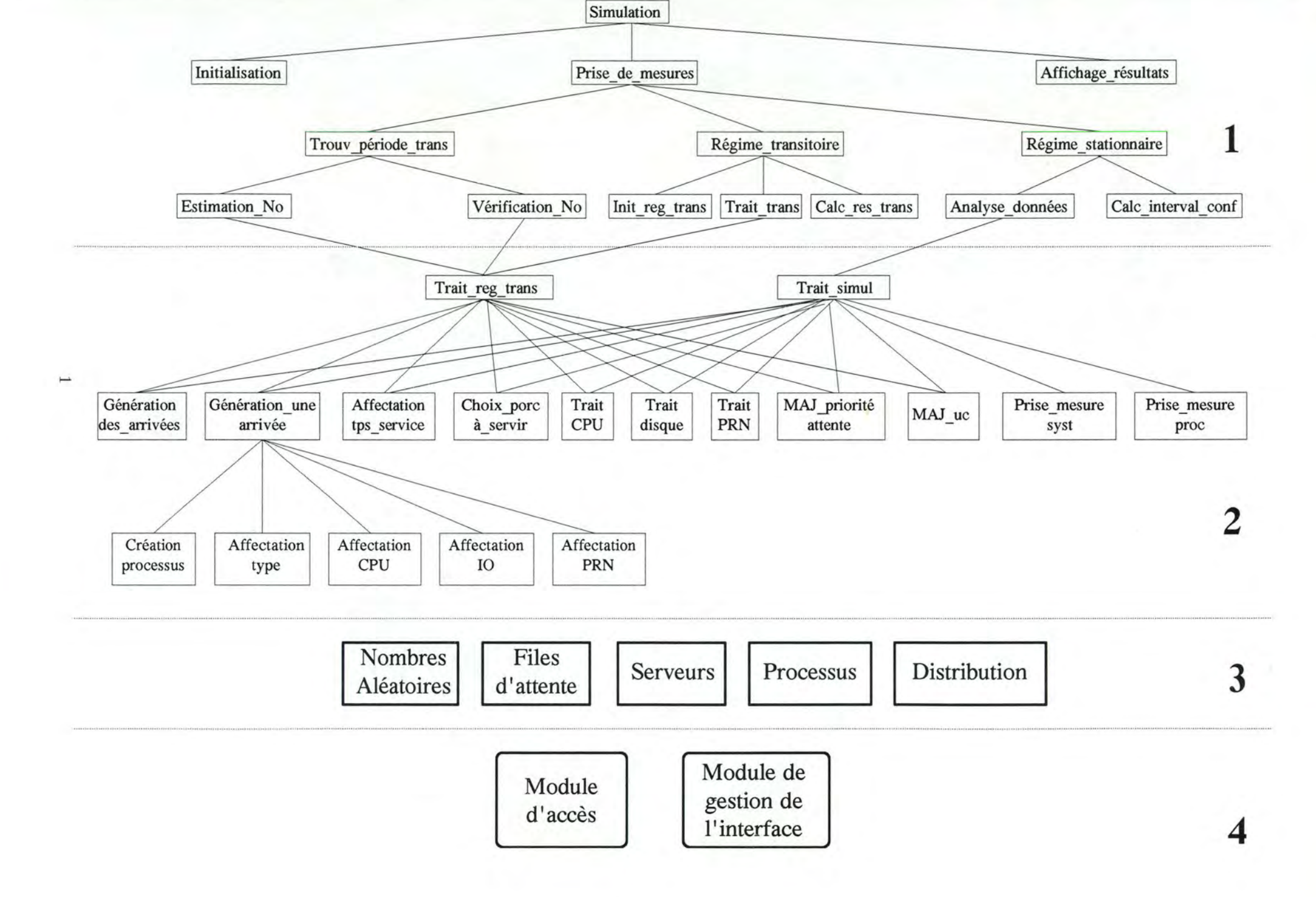
Gérald Derzelle
Vincent Despriet

Promoteur : M. Noirhomme-Fraiture

Année académique 1991-1992

Table des matières

1. Architecture


Simulation
Initialisation
Prise_de_mesures
Affichage_résultats
Trouv_période_trans
Régime_transitoire
Régime_stationnaire
1
Estimation_No
Vérification_No
Init_reg_trans
Trait_trans
Calc_res_trans
Analyse_données
Calc_interval_conf
Trait_reg_trans
Trait_simul
Génération des_arrivées
Génération_une arrivée
Affectation tps_service
Choix_porc à_servir
Trait CPU
Trait disque
Trait PRN
MAJ_priorité attente
MAJ_uc
Prise_mesure syst
Prise_mesure proc
Création processus
Affectation type
Affectation CPU
Affectation IO
Affectation PRN
2
Nombres Aléatoires
Files d'attente
Serveurs
Processus
Distribution
3
Module d'accès
Module de gestion de l'interface
4

2. Glossaire des variables

alpha : degré d'incertitude lors du calcul des intervalles de confiance.

borne_min_type_proc : nombre d'observations minimum à réaliser pour chaque type de processus de manière à obtenir des résultats valables (en régime stationnaire).

borne_obs : nombre d'observations à réaliser pour remplir tous les blocs de données nécessaires pour calculer les intervalles de confiance (en régime stationnaire).

DELAI

fin[0] : temps qui doit encore s'écouler avant le déclenchement de l'événement 'fin de service CPU'.

fin[1] : temps qui doit encore s'écouler avant le déclenchement de l'événement 'fin de service DISQUE'.

fin[2] : temps qui doit encore s'écouler avant le déclenchement de l'événement 'fin de service IMPRIMANTE'.

arr_svte : temps qui doit encore s'écouler avant le déclenchement de l'événement 'génération d'une arrivée'.

gen_nb_arr : temps qui doit encore s'écouler avant le déclenchement de l'événement génération des arrivées'.

calc_prior : temps qui doit encore s'écouler avant le déclenchement de l'événement 'calcul des priorités'.

DOS_RES_SERV

tps_util[i] : temps d'utilisation du serveur i (il permet de calculer le taux d'occupation du serveur i).

evt : événement.

FILE

proc : processus de la file

file_cpu : file d'attente au serveur CPU.
file_io : file d'attente au serveur DISQUE.
file_prn : file d'attente au serveur IMPRIMANTE.

INTERVAL

borne_inf : borne inférieure de l'intervalle de confiance
val : moyenne estimée
borne_sup: borne supérieure de l'intervalle de confiance

gamma : Variable servant à déterminer le nombre d'observations à prendre en compte lors du premier test de stationnarité.
gammav : Coéfficient de sécurité qui détermine la qualité de l'estimateur de la variance lors de la détection de la période transitoire.
interval : intervalle de temps qui sépare deux arrivées dans une seconde. Il permet de répartir les arrivées de manière constante.
nb : nombre d'arrivées à générer dans la seconde.
nb_arr_gen : nombre d'arrivées déjà générées dans la seconde.
nb_obs_proc : nombre d'observations déjà effectuées concernant les processus.
nb_obs_syst : nombre d'observations déjà effectuées concernant l'état du système (files d'attente).
nb_obs_type[i] : nombre d'observations déjà effectuées pour les processus de type i.
nb_simul : nombre de simulations déjà effectuées en régime transitoire.
no : estimation du nombre d'observations effectuées en régime transitoire.

no_batch_proc : numéro du bloc courant pour les observations concernant les processus.

no_batch_syst : numéro du bloc courant pour les observations concernant le système (files d'attente).

no_max : nombre maximum d'observations à effectuer pour déterminer la période transitoire.

no_res : numéro du réseau simulé.

nt : nombre d'observations sur lesquelles le test de stationnarité est effectué.

nv : nombre d'observations qui sont utilisées pour estimer la variance (sous ensemble de nt; on prend nt = gammav * nv)

obs : une observation qui permet de déterminer si le système se trouve en régime stationnaire (une observation correspond au nombre de processus dans le système.

PROCESSUS

type : type de processus.

tps[i] : temps de service global du processus au serveur i

tps_octroye[i] : partie du temps de service global du processus déjà octroyé au serveur i.

tps_file[i] : temps passé par le processus dans la file d'attente du serveur i.

priorite_init : priorité initiale attribuée au processus lors de sa création (elle est fonction du type du processus).

priorite : priorité attribuée au processus (elle est réajustée régulièrement).

uc : utilisation récente de l'unité centrale (processus = serveur 0). Cette variable est utilisée pour calculer la priorité d'un processus.

att_tranche : temps passé par le processus dans la file d'attente du processeur (serveur 0) depuis son dernier service à ce même serveur. Cette variable est utilisée pour déterminer le prochain processus qui va être servi par le processus.

pass_svt : variable positionnée lors de chaque passage au processeur ou au disque qui indique si le processus va rester au même serveur ou s'il va changer de serveur après son service.

coeff : coefficient calculé lors de la création du processus et utilisé pour positionner la variable 'pass_svt'.

RES_SIMUL

tps_file[i] : intervalle de confiance trouvé pour le temps d'attente moyen des processus au serveur i.

tps_rep : intervalle de confiance trouvé pour le temps de réponse moyen des processus.

taux_occ[i] : taux d'occupation du serveur i.

nb_file[i] : taux d'occupation du serveur i pour le nombre moyen de processus dans la file d'attente au serveur i.

tps_rep_type[i] : taux d'occupation du serveur i pour le temps de réponse moyen des processus de type i.

RES_SERV :

tps_util[i] : temps d'utilisation du serveur i (il permet de calculer le taux d'occupation du serveur i).

RES_TRANS

type : type du dernier processus qui s'est terminé.

tps_file[i] : temps d'attente au serveur i du dernier processus qui s'est terminé.

tps[i] : temps de service global au serveur i du dernier processus qui s'est terminé.

TAB_MOY : (en régime stationnaire)

syst[i][j] : moyenne des observations du bloc j concernant le nombre de processus dans la file du serveur i.

att[i][j] : moyenne des observations du bloc j concernant le temps d'attente des processus du serveur i.

rep[i] : moyenne des observations du bloc i concernant le temps de réponse des processus.

rep_type[i][j] : moyenne des observations du bloc j concernant le temps de réponse des processu de type i.

nb_bloc[i] : nombre de blocs d'observations concernant le temps de réponse des processus de type i.

TAB_OBS : (en régime transitoire)

att[i][j] : jème observation du temps d'attente d'un processus au serveur i.

file[i][j] : jème observation du nombre de processus dans la file du serveur i.

rep[i] : ième observation du tps de réponse d'un processus.

rep_type[i][j] : jème observation du tps de réponse d'un processus de type i.

taux_occ[i] : nombre de fois que l'on a observé que le serveur i était occupé.

nb_obs_type[i] : nombre d'observations réalisées d'un processus de type i.

tab_som_att[i][j] : somme des observations du bloc j du temps passé par les processus dans la file d'attente du serveur i.

tab_som_rep[j] : somme des observations du bloc i du temps de réponse des processus.

tab_som_rep_type[i][j] : somme des observations du bloc j du temps de réponse des processus de type i.

tab_som_syst[i][j] : somme des observations du bloc j du nombre de processus dans la file d'attente du serveur i.

tab_tps_bloc[j] : temps écoulé pour obtenir le bloc j de tab_som_syst.

tps_ser : intervalle de temps qui doit encore s'écouler avant le déclenchement de l'événement suivant.

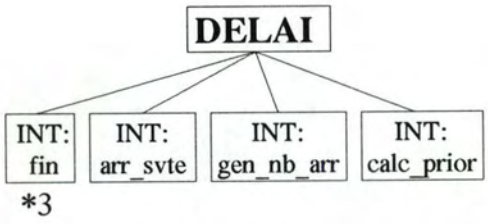
tps_simul : compteur qui indique le temps qui s'est déjà écoulé depuis le début de la simulation.

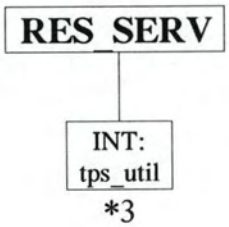
trouve : variable booléenne qui est positionnée à TRUE si le régime stationnaire est trouvé et à FALSE sinon.

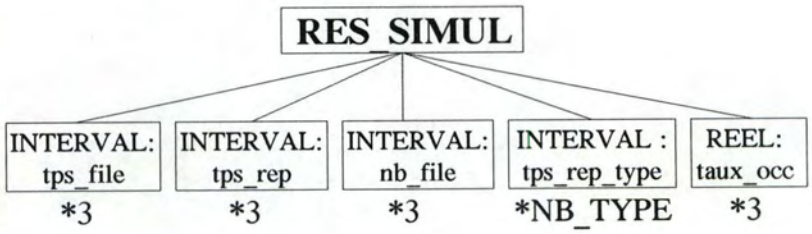
verifie : variable booléenne qui est positionnée à TRUE si le régime stationnaire à été vérifié et à FALSE sinon.

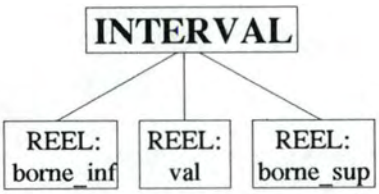
3. Structures de données

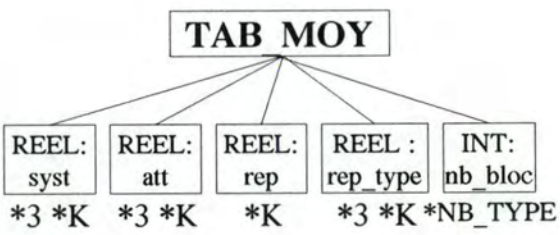

FILE
*
PROCESSUS:
proc

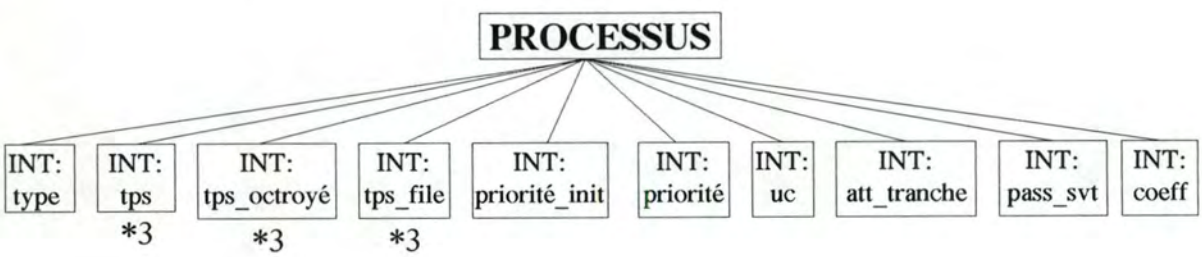

DELAI
INT:
fin
INT:
arr_svte
INT:
gen_nb_arr
INT:
calc_prior
*3

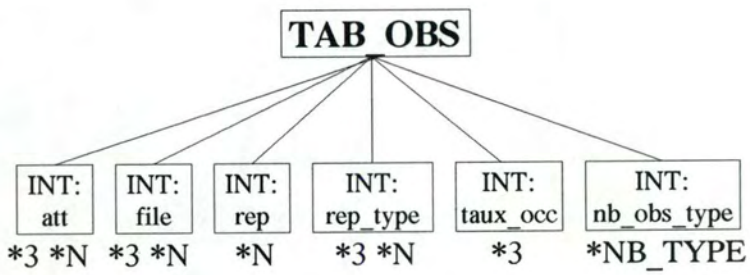

RES_SERV
INT:
tps_util
*3

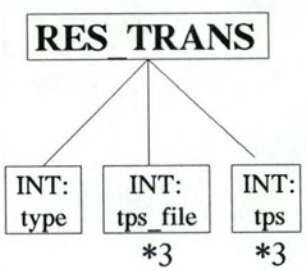

RES_SIMUL
INTERVAL:
tps_file
*3
INTERVAL:
tps_rep
*3
INTERVAL:
nb_file
*3
INTERVAL :
tps_rep_type
*NB_TYPE
REEL:
taux_occ
*3


INTERVAL
REEL:
borne_inf
REEL:
val
REEL:
borne_sup


TAB_MOY
REEL:
syst
REEL:
att
REEL:
rep
REEL :
rep_type
INT:
nb_bloc
*3 *K
*3 *K
*K
*3 *K *NB_TYPE


PROCESSUS
INT:
type
INT:
tps
*3
INT:
tps_octroyé
*3
INT:
tps_file
*3
INT:
priorité_init
INT:
priorité
INT:
uc
INT:
att_tranche
INT:
pass_svt
INT:
coeff





EVENEMENT = { 'générer_des_arrivées', 'arrivee', 'calc_prior', 'fin_cpu', 'fin_io', 'fin_prn' }

4. Glossaire des fonctions

Glossaire des Fonctionnalités.

(par ordre numérique)

1. Simulation
2. Initialisation
3. Prise_de_mesures
4. Trouv_période_trans
5. Estimation_no
6. Vérification_no
7. Régime_stationnaire
8. Analyse_données
9. Calc_interval_conf
10. Affich_résultats
11. Trait_reg_trans
12. Trait_simul
13. Génération_des_arrivées
14. Génération_une_arrivée
15. Création_processus
16. Affectation_type
17. Affectation_cpu
18. Affectation_io
19. Affectation_prn
20. Affectation_tps_service
21. Choix_proc_à_servir
22. Trait_cpu
23. Trait_disque
24. Trait_prn
25. Maj_priorité_attente
26. Maj_uc
27. Prise_mesure_syst
28. Prise_mesure_proc
29. Régime_transitoire
30. Init_reg_trans
31. Trait_trans
32. Calc_res_trans

Glossaire des Fonctionnalités.

(par ordre alphabétique)

17. Affectation_cpu
18. Affectation_io
19. Affectation_prn
20. Affectation_tps_service
16. Affectation_type
10. Affich_résultats
8. Analyse_données
9. Calc_interval_conf
32. Calc_res_trans
21. Choix_proc_à_servir
15. Création_processus
5. Estimation_no
13. Génération_des_arrivées
14. Génération_une_arrivée
2. Initialisation
30. Init_reg_trans
25. Maj_priorité_attente
26. Maj_uc
3. Prise_de_mesures
27. Prise_mesure_syst
28. Prise_mesure_proc
7. Régime_stationnaire
30. Régime_transitoire
1. Simulation
22. Trait_cpu
23. Trait_disque
24. Trait_prn
11. Trait_rég_trans
12. Trait_simul
31. Trait_trans
4. Trouv_période_trans
6. Vérification_no

5. Définitions formelles des fonctions

Fonctionnalité : Simulation **Numéro :** 1

Arguments : -----

Résultats : -----

Précondition : -----

Postcondition : (tps_simul, nb, nb_arr_gen, no_max, nv, nt, alpha, gammav, gamma, file_cpu, file_io, file_prn, dos_res_serv, dos_delai, no_res) = initialisation ()

& res_simul = prise_de_mesures (tps_simul, nb, nb_arr_gen, no_max, nv, nt, alpha, gammav, gamma, file_cpu, file_io, file_prn, dos_res_serv, dos_delai, no_res)

& affich_résultats (res_simul)

Structures de données : nb, nb_arr_gen, tps_simul, no_max, nv, nt : INT
gammav, no_res : INT
file_cpu, file_io, file_prn : FILE
dos_res_serv : RES_SERV
dos_delai : DELAI
res_simul : RES_SIMUL
gamma, alpha : REEL

Commentaires : Cette fonction effectue la simulation d'un réseau local de type Ethernet avec un serveur SUN, un disque dur et une imprimante.

Fonctionnalité : Initialisation **Numéro :** 2

Arguments : -----

Résultats : tps_simul, nb, nb_arr_gen, no_max, nv, nt, alpha, gammav, gamma, file_cpu, file_io, file_prn, dos_res_serv, dos_delai

Précondition : -----

Postcondition : $\forall$ i : $0 \leq i \leq 2$: fin[i](dos_delai) = HIGH_VALUE
& arr_svte(dos_delai) = HIGH_VALUE
& calc_prior(dos_delai) = 1000
& gen_nb_arr(dos_delai) = 0
& init_file (file_cpu)
& init_file (file_io)
& init_file (file_prn)
& init_dos_res_serv (dos_res_serv)
& nb = 0
& nb_arr_gen = 0
& tps_simul = 0
& no_max = 1000
& nv = 100
& alpha = 0.05
& gammav = 2
& gamma = 0.5
& no_res = lec_num_res()

Structures de données : nb, nb_arr_gen, tps_simul, no_max : INT
nv, nt, gammav, no_res : INT
file_cpu, file_io, file_prn : FILE
dos_delai : DELAI
dos_res_serv : RES_SERV
gamma, alpha : REEL

Commentaires : Cette fonction initialise toutes les variables qui seront utilisées dans le programme de simulation.

Fonctionnalité : Prise_de_mesures **Numéro :** 3

Arguments : tps_simul, nb, nb_arr_gen, no_max, nv, nt, alpha, gammav, gamma, file_cpu, file_io, file_prn, dos_res_serv, dos_delai, no_res

Résultats : res_simul

Précondition : -----

Postcondition : trouvé = trouv_période_trans (tps_simul, nb, nb_arr_gen, no_max, nv, nt, alpha, gammav, gamma, file_cpu, file_io, file_prn, dos_res_serv, dos_delai, no_res)

& trouvé = > res_simul = régime_stationnaire (tps_simul, nb, nb_arr_gen, alpha, file_cpu, file_io, file_prn, dos_res_serv, dos_delai, no_res)

& NOT trouvé = > res_simul = régime_transitoire (nb,nb_arr_gen, alpha, file_cpu, file_io, file_prn, dos_res_serv, dos_delai, no_res)

Structures de données : nb, nb_arr_gen, tps_simul, no_max : INT
nv, nt, gammav, no_res : INT
file_cpu, file_io, file_prn : FILE
dos_delai : DELAI
dos_res_serv : RES_SERV
trouvé : BOOLEEN
gamma, alpha : REEL
res_simul : RES_SIMUL

Commentaires : Cette fonction effectue le traitement de la simulation. Il se fait en deux étapes. En premier lieu, il convient de trouver la période transitoire. Ensuite, si le régime stationnaire est atteint, on effectue une simulation en régime stationnaire (théorème de Doob_Birkhov), sinon, on effectue plusieurs simulations en régime transitoire.

Fonctionnalité : Trouv_période_trans **Numéro :** 4

Arguments : tps_simul, nb, nb_arr_gen, no_max, nv, nt, alpha, gammav, gamma, file_cpu, file_io, file_prn, dos_res_serv, dos_delai, no_res

Résultats : trouvé

Précondition : -----

Postcondition : (no, {obs}) = estimation_no (tps_simul, nb, nb_arr_gen, no_max, file_cpu, file_io, file_prn, dos_res_serv, dos_delai, no_res)

& (no ≤ no_max) = > trouvé = vérification_no (no, {obs}, tps_simul, nb, nb_arr_gen, no_max, nv, nt, alpha, gammav, gamma, file_cpu, file_io, file_prn, dos_res_serv, dos_delai, no_res)

& (no > no_max) = > trouvé = FALSE

Structures de données : nb, nb_arr_gen, tps_simul, no_max : INT
nv, nt, gammav, no, obs_i, no_res : INT
file_cpu, file_io, file_prn : FILE
dos_res_serv : RES_SERV
dos_delai : DELAI
trouvé : BOOLEEN
gamma, alpha : REEL

Commentaires : Cette fonction effectue des observations jusqu'à ce que le système se trouve en régime stationnaire. Si celui-ci n'est pas atteint après no_max observations, la fonction renvoie la valeur FALSE. Les observations correspondent au nombre de processus dans le système.

Fonctionnalité : Estimation_no **Numéro :** 5

Arguments : tps_simul, nb, nb_arr_gen, no_max, file_cpu, file_io, file_prn, dos_res_serv, dos_delai, no_res

Résultats : no, {obs}

Précondition : i = 0 AND #{obs} = 0 AND no = 0

Postcondition : (NOT test_empirique AND no ≤ no_max)
=> trait_reg_trans (tps_simul, nb, nb_arr_gen, file_cpu, file_io, file_prn, dos_res_serv, dos_delai, no_res)
$i_{POST} = i_{PRE} + 1$
obs_i = nb_syst (file_cpu, file_io, file_prn)
(no, {obs}) = Estimation_no (tps_simul, nb, nb_arr_gen, no_max, file_cpu, file_io, file_prn, dos_res_serv, dos_delai, no_res) + 1

Structures de données : nb, nb_arr_gen, tps_simul, no, no_max, i, obs_i, no_res : INT
file_cpu, file_io, file_prn : FILE
dos_res_serv : RES_SERV
dos_delai : DELAI

Commentaires : Cette fonction détermine n_0 en employant la règle empirique énoncée par Fishman.

Test_empirique = res

```
k = 0
som = Σ obs_i
moyenne = som / # {obs}
si obs_1 > moyenne
    SW = 1
sinon
    SW = 0
pour  i = 2  à  # {obs}
    si  SW = 1 et obs_i < moyenne
            k = k + 1
            SW = 0
    sinon
            si SW = 0  et  obs_i > moyenne
             k = k + 1
             SW = 1

si  k ≥ 25
    res = TRUE
sinon
    res = FALSE
```

Fonctionnalité : Vérification_no **Numéro :** 6

Arguments : no, {obs}, tps_simul, nb, nb_arr_gen, no_max, nv, nt, alpha, gammav, gamma, file_cpu, file_io, file_prn, dos_res_serv, dos_delai, no_res

Résultats : vérifié

Précondition : -----

Postcondition : test_stationnarité = > vérifié = TRUE

& (NOT test_stationnarité & (no ≥ no_max)
= > vérifié = FALSE

& (NOT test_stationnarité & (no ≤ no_max))
= > trait_reg_trans (tps_simul, nb, nb_arr_gen,
file_cpu, file_io, file_prn, dos_res_serv,
dos_delai, no_res)
$i_{POST} = i_{PRE} + 1$
obs_i = nb_syst (file_cpu, file_io, file_prn)
$no_{POST} = no_{PRE} + 1$
vérifié = vérification_no (no, {obs}, tps_simul,
nb, nb_arr_gen, no_max, nv, nt, alpha, gammav,
gamma, file_cpu, file_io, file_prn, dos_res_serv,
dos_delai, no_res)

Structures de données : nb, nb_arr_gen, tps_simul, no_res : INT
no, no_max, nv, nt, gammav, obs_i : INT
file_cpu, file_io, file_prn : FILE
dos_res_serv : RES_SERV
dos_delai : DELAI
vérifié : BOOLEEN
gamma, alpha : REEL

Commentaires : Cette fonction vérifie que le système se trouve bien en régime stationnaire selon la méthode de Pawlikowski.

- **Test_de_stationnarité = res**

n_t = gammav * n_v
Δn = gammav * nv
ind = n_0

si (n_0 + n_t) ≤ n0_max

pour i = 0 à Δn - 1
Trait_reg_trans (file_cpu, file_io, file_prn, dos_res_serv, dos_delai, nb, nb_arr_gen, tps_simul)
obs_{ind} = Nb_syst (file_cpu, file_io, file_prn)
ind = ind + 1

moy_seq = $\sum_{i=n_0+n_t-n_v+1}^{n_0+n_t} obs_i$

(variance, k) = Calc_variance ({obs}, n_v, moy_seq_nv)
T = Calc_t (table, n_t, n_v, variance)
si T > Student (k, alpha)
Δn = gamma * n_0
n_0 = n_0 + Δn

sinon
'n_0 > n0_max = > arrêt de la simulation'
res = FALSE

jusqu'à ce que T < Student (k, alpha)

res = TRUE

- **Calc_variance**

Voir Volume I page 38.

- **Calc_t**

Voir Volume I page 39.

Fonctionnalité : Régime_stationnaire **Numéro :** 7

Arguments : tps_simul, nb, nb_arr_gen, alpha, file_cpu, file_io, file_prn, dos_res_serv, dos_delai, no_res

Résultats : res_simul

Précondition : -----

Postcondition : tab_moy = analyse_données (tps_simul, nb, nb_arr_gen, file_cpu, file_io, file_prn, dos_res_serv, dos_delai, no_res)

& res_simul = calc_interval_conf (tab_moy, alpha)

& $\forall$ i : $0 \leq i \leq 2$: taux_occ[i] (res_simul) = (lec_tps_util(i, dos_res_serv) / tps_simul) * 100

Structures de données : tps_simul, nb, nb_arr_gen : INT
file_cpu, file_io, file_prn : FILE
dos_res_serv : RES_SERV
dos_delai : DELAI
res_simul : RES_SIMUL
tab_moy : TAB_MOY
alpha : REEL

Commentaires : Cette fonction effectue le traitement de la simulation lorsque le système se trouve en régime stationnaire.

Fonctionnalité : Analyse_données **Numéro :** 8

Arguments : tps_simul, nb, nb_arr_gen, file_cpu, file_io, file_prn, dos_res_serv, dos_delai, no_res

Résultats : tab_moy

Constantes : K -> nombre de blocs
M -> taille d'un bloc
S -> nombre de blocs à ignorer entre deux blocs pris en compte
NB_TYPE -> nombre de types de processus émis sur le réseau

Précondition : borne_obs = (K + (K-1) * S) * M
& borne_min_type_proc = (5 + (4 * S)) * M

Postcondition : ((nb_obs_syst < borne_obs OR nb_obs_proc < borne_obs
OR $nb\_obs\_type_0$ < borne_min_type_proc
OR $nb\_obs\_type_1$ < borne_min_type_proc
.....
OR $nb\_obs\_type_{NB\_TYPE}$ < borne_min_type_proc)
=> (tab_som_att, tab_som_rep, tab_som_rep_type,
tab_tps_bloc, tab_som_syst)
= trait_simul (tps_simul, nb, nb_arr_gen, file_cpu, file_io,
file_prn, dos_res_serv, dos_delai, no_res))

& tab_moy = calc_moyennes (tab_som_att, tab_som_rep,
tab_som_rep_type, tab_tps_bloc, tab_som_syst)

Structures de données : tps_simul, nb, nb_arr_gen, borne_obs, borne_min_type : INT
file_cpu, file_io, file_prn : FILE
dos_res_serv : RES_SERV
dos_delai : DOS_DELAI
tab_moy : TAB_MOY
tab_som_att, tab_som_rep, tab_som_rep_type : SEQ[INT]
tab_tps_bloc, tab_som_syst : SEQ[INT]

Commentaires : Cette fonction effectue les mesures en régime stationnaire. Elle mémorise toutes les valeurs observées dans des tableaux. Elle renvoie les moyennes qui sont utilisées pour calculer les intervalles de confiance.

Calc_moyennes

```
pour i=0 à 2
    pour j = 0 à K-1
        syst[i][j] (tab_moy) = tab_moy_syst[i][j] / tab_tps_bloc[j]
        att[i][j] (tab_moy) = tab_som_att[i][j] / M
        si  i = 0
            rep[j] (tab_moy) = tab_som_rep[j] / M

pour  i=0 à NB_TYPE - 1
    nb_bloc = nb_obs_type[i] / (2*M)
    nb_obs_dern_bloc = nb_obs_type[i] - (nb_bloc*K)
    pour  j=0 à nb_bloc - 1
        rep_type[i][j] (tab_moy) = tab_som_rep[i][j] / M

    si  nb_obs_dern_bloc > 0
        rep_type[i][j] (tab_moy) = tab_som_rep_type[i][j] / nb_obs_dern_bloc
        nb_bloc = nb_bloc + 1

    nb_bloc[i] (tab_moy) = nb_bloc
```

Fonctionnalité : Calc_interval_conf **Numéro :** 9

Arguments : tab_moy, alpha

Résultats : res_simul

Constante : K -> nombre de blocs
NB_TYPE -> nombre de type de processus du réseau

Précondition : -----

Postcondition : (formule volume I p.42)

où X = syst[i][j] (tab_moy)	$0 \leq i \leq 2$	$0 \leq j \leq K-1$
X = att[i][j] (tab_moy)	$0 \leq i \leq 2$	$0 \leq j \leq K-1$
X = rep[j] (tab_moy)		$0 \leq j \leq K-1$
X = rep_type[i][j] (tab_moy)	$0 \leq i \leq NB\_TYPE - 1$	$0 \leq j \leq K-1$

Structures de données : res_simul : RES_SIMUL
tab_moy : TAB_MOY
alpna : REEL

Commentaires : Cette fonction calcule les intervalles de confiance pour chaque paramètre.

Fonctionnalité : Affich_résultats **Numéro :** 10

Arguments : res_simul

Résultats : -----

Précondition : -----

Postcondition :

Structures de données : res_simul : RES_SIMUL

Commentaires : Cette fonction a pour but d'afficher tous les résultats de la simulation à l'écran.

Fonctionnalité : Trait_reg_trans **Numéro :** 11

Arguments : tps_simul, nb, nb_arr_gen, file_cpu, file_io, file_prn, dos_res_serv, dos_delai, no_res

Résultats : -----

Précondition : -----

Postcondition : ({evt}, tps_ser) = affectation_tps_service (dos_délai)

& traitement_serveurs

& traitement_événement

Structures de données : tps_ser, tps_simul, nb, nb_arr_gen, obs, no_res : INT
file_cpu, file_io, file_prn : FILE
dos_res_serv : RES_SERV
dos_delai : DELAI
evt_i : EVENEMENT

Commentaires : Cette fonction effectue le traitement d'une simulation en régime transitoire (période initiale).

Traitement_des_serveurs

```
calc_prior (dos_delai) = calc_prior (dos_delai) - tps_ser
gen_nb_arr (dos_delai) = gen_nb_arr (dos_delai) - tps_ser

si arr_svte (dos_delai) ≠ HIGH_VALUE
      arr_svte (dos_delai) = arr_svte (dos_delai) - tps_ser

si  File_non_vide (file_cpu)
      Trait_cpu (tps_ser, file_cpu, dos_res_serv, dos_delai)

si  File_non_vide (file_io)
      Trait_disque (tps_ser, file_io, dos_res_serv, dos_delai)

si  File_non_vide (file_prn)
      Trait_prn (tps_ser, file_prn, dos_res_serv, dos_delai)

tps_simul = tps_simul + tps_ser
```

Traitement_événement

```
si  'générer_des_arrivées'  ∈ evt
      nb = génération_des_arrivées (no_res)
      gen_nb_arr (dos_delai) = 1000
      nb_arr_gen = 0

      si nb = 0
         arr_svte (dos_delai) = HIGH_VALUE
      si nb > 0
         arr_svte (dos_delai) = 0
         Append (evt, 'arrivée')
         interval = 1000 / nb
         Remove (evt, 'générer_des_arrivées')

si  'arrivée'  ∈ evt
      Génération_une_arrivée (dos_delai, nb, nb_arr_gen, interval, file_cpu, no_res)
      Remove (evt, 'arrivée')

si  'fin_cpu'  ∈ evt
      proc = Lec_prem_proc_file (file_cpu)
      si Proc_fini (proc, 0) ET Proc_fini (proc, 1)
         si Proc_fini (proc, 2)
              Suppression (file_cpu)
         sinon
              Ajout (proc file_prn)
              Suppression (file_cpu)
      sinon
         si  Lec_pass_svt_proc (proc) = 1
              Ajout (proc, file_cpu)
              Supression (file_cpu)
      Remove (evt, 'fin_cpu')

si  'fin_io'  ∈ evt
      proc = Lec_prem_proc_file (file_io)
      Ajout (proc, file_cpu)
      Supression (file_io)

      Remove (evt, 'fin_io')
```

```
si 'fin_prn' ∈ evt
    proc = Lec_prem_proc_file (file_prn)
    Ajout (proc, file_cpu)
    Supression (file_prn)
    Remove (evt, 'fin_prn')

si 'calc_prior' ∈ evt
    si File_non_vide (file_cpu)
        Maj_priorité_attente (file_cpu)
    si File_non_vide (file_io)
        Maj_uc (file_io)
    si File_non_vide (file_prn)
        Maj_uc (file_prn)
    Calc_prior (dos_delai) = 1000
    Remove (evt, 'calc_prior')

si File_non_vide (file_cpu)
    si fin[0](dos_delai) = 0 OU fin[0](dos_delai) = HIGH_VALUE
        Choix_proc_a_servir (file_cpu)
        fin[0](dos_delai) = Calc_delai_fin_cpu
sinon
    fin[0](dos_delai) = HIGH_VALUE

si File_non_vide (file_io)
    si fin[1](dos_delai) = 0 OU fin[1](dos_delai) = HIGH_VALUE
        fin[1](dos_delai) = Calc_delai_fin_io
sinon
    fin[1](dos_delai) = HIGH_VALUE

si File_non_vide (file_prn)
    si fin[2](dos_delai) = 0 OU fin[2](dos_delai) = HIGH_VALUE
        fin[2](dos_delai) = Calc_delai_fin_prn
sinon
    fin[2](dos_delai) = HIGH_VALUE
```

```
Calc_delai_fin_cpu = delai_fin

    proc = lec_prem_proc_file(file_cpu)
    tps_cpu = lec_tps_proc(0,proc)
    tps_cpu_octroye = lec_tps_proc_octroye(0,proc)
    tps_io = lec_tps_proc(1,proc)
    tps_io_octroye = lec_tps_proc_octroye(1,proc)
    coeff = lec_coeff(proc)

    si (proc_fini(proc,1))
        p = 1
    sinon
        p = (tps_cpu - tps_cpu_octroye) - min(100,tps_cpu - tps_cpu_octroye)
        si (p!=0) p = p / (p + ((tps_io - tps_io_octroye) * coeff))

    nb_genere = gen_nb()
    si (nb_genere ≤ p)
        delai_fin = min(100,tps_cpu - tps_cpu_octroye)
        proc->pass_svt = 0
    sinon
        nb_genere = gen_nb()
        delai_fin = min((int) (nb_genere * 100) + 1,tps_cpu - tps_cpu_octroye)
        proc->pass_svt = 1
```

```
Calc_delai_fin_io = delai_fin

    proc = lec_prem_proc_file(file_io)
    tps_cpu = lec_tps_proc(0,proc)
    tps_cpu_octroye = lec_tps_proc_octroye(0,proc)
    tps_io = lec_tps_proc(1,proc)
    tps_io_octroye = lec_tps_proc_octroye(1,proc)
    coeff = lec_coeff(proc)
    tps_acces = acc_tps_accès (no_res)

    delai_fin = 0
    tant que ((proc->pass_svt == 1) ET (tps_ser ≤ tps_io - tps_io_octroye))
        delai_fin = delai_fin + tps_acces

    si (proc_fini(proc,0))
        p = 1
    sinon
        p = ((tps_io - tps_io_octroye) - tps_ser) * coeff
        si (p!=0) p = p / (p + (tps_cpu - tps_cpu_octroye))

    nb_genere = gen_nb()
    si (nb_genere ≤ p)
        proc->pass_svt = 1
    sinon
        proc->pass_svt = 0


Calc_delai_fin_prn = delai_fin

    proc = lec_prem_proc_file(file_prn)
    delai_fin = lec_tps_proc(2,proc)
```

Fonctionnalité : Trait_simul **Numéro :** 12

Arguments : tps_simul, nb, nb_arr_gen, file_cpu, file_io, file_prn, dos_res_serv, dos_delai, no_res

Résultats : tab_som_att, tab_som_rep, tab_som_rep_type, tab_tps_bloc, tab_som_syst

Précondition : -----

Postcondition : ({evt}, tps_ser) = affectation_tps_service (dos_délai)

& traitement_serveurs

& traitement_événement

Structures de données : tps_ser, tps_simul, nb, nb_arr_gen, no_res : INT
file_cpu, file_io, file_prn : FILE
dos_res_serv : RES_SERV
dos_delai : DELAI
evt_i : EVENEMENT
tab_som_att, tab_som_rep, tab_som_rep_type : SEQ[INT]
tab_tps_bloc, tab_som_syst : SEQ[INT]

Commentaires : Cette fonction effectue le traitement de la simulation et prend les mesures en régime stationnaire.

Traitement_des_serveurs

```
calc_prior (dos_delai) = calc_prior (dos_delai) - tps_ser
gen_nb_arr (dos_delai) = gen_nb_arr (dos_delai) - tps_ser

si arr_svte (dos_delai) ≠ HIGH_VALUE
      arr_svte (dos_delai) = arr_svte (dos_delai) - tps_ser

si  File_non_vide (file_cpu)
      Trait_cpu (tps_ser, file_cpu, dos_res_serv, dos_delai)

si  File_non_vide (file_io)
      Trait_disque (tps_ser, file_io, dos_res_serv, dos_delai)

si  File_non_vide (file_prn)
      Trait_prn (tps_ser, file_prn, dos_res_serv, dos_delai)

tps_simul = tps_simul + tps_ser
```

Traitement_événement

```
si  'arrivée' OU 'fin_cpu' OU 'fin_io' OU 'fin_prn'  ∈ evt
     prise_mesure_syst (tps_ser, nb_obs_syst, no_batch_syst, tab_tps_bloc,
                              tab_som_syst, file_cpu, file_io, file_prn)

si  'générer_des_arrivées'  ∈ evt
     nb = génération_des_arrivées (no_res)
     gen_nb_arr (dos_delai) = 1000
     nb_arr_gen = 0

     si nb = 0
        arr_svte (dos_delai) = HIGH_VALUE
     si nb > 0
        arr_svte (dos_delai) = 0
        Append (evt, 'arrivée')
        interval = 1000 / nb
        Remove (evt, 'générer_des_arrivées')

si  'arrivée'  ∈ evt
     Génération_une_arrivée (dos_delai, nb, nb_arr_gen, interval, file_cpu, no_res)
     Remove (evt, 'arrivée')

si  'fin_cpu'  ∈ evt
     proc = Lec_prem_proc_file (file_cpu)
     si Proc_fini (proc, 0) ET Proc_fini (proc, 1)
        si Proc_fini (proc, 2)
           Prise_mesure_proc (nb_obs_proc, no_batch_proc, tab_som_att,
                              tab_som_rep, tab_som_rep_type, nb_obs_type,
                                    no_batch_type)
           Suppression (file_cpu)
        sinon
           Ajout (proc file_prn)
           Suppression (file_cpu)
     sinon
        si  Lec_pass_svt_proc (proc) = 1
           Ajout (proc, file_cpu)
           Supression (file_cpu)

     Remove (evt, 'fin_cpu')
```

```
si 'fin_io' ∈ evt
    proc = Lec_prem_proc_file (file_io)
    Ajout (proc, file_cpu)
    Supression (file_io)
    Remove (evt, 'fin_io')

si 'fin_prn' ∈ evt
    proc = Lec_prem_proc_file (file_prn)
    Ajout (proc, file_cpu)
    Supression (file_prn)
    Remove (evt, 'fin_prn')

si 'calc_prior' ∈ evt
    si File_non_vide (file_cpu)
        Maj_priorité_attente (file_cpu)
    si File_non_vide (file_io)
        Maj_uc (file_io)
    si File_non_vide (file_prn)
        Maj_uc (file_prn)
    Calc_prior (dos_delai) = 1000
    Remove (evt, 'calc_prior')

si File_non_vide (file_cpu)
    si fin[0](dos_delai) = 0 OU fin[0](dos_delai) = HIGH_VALUE
        Choix_proc_a_servir (file_cpu)
        fin[0](dos_delai) = Calc_delai_fin_cpu
sinon
    fin[0](dos_delai) = HIGH_VALUE

si File_non_vide (file_io)
    si fin[1](dos_delai) = 0 OU fin[1](dos_delai) = HIGH_VALUE
        fin[1](dos_delai) = Calc_delai_fin_io
sinon
    fin[1](dos_delai) = HIGH_VALUE

si File_non_vide (file_prn)
    si fin[2](dos_delai) = 0 OU fin[2](dos_delai) = HIGH_VALUE
        fin[2](dos_delai) = Calc_delai_fin_prn
sinon
    fin[2](dos_delai) = HIGH_VALUE
```

```
Calc_delai_fin_cpu = delai_fin

    proc = lec_prem_proc_file(file_cpu)
    tps_cpu = lec_tps_proc(0,proc)
    tps_cpu_octroye = lec_tps_proc_octroye(0,proc)
    tps_io = lec_tps_proc(1,proc)
    tps_io_octroye = lec_tps_proc_octroye(1,proc)
    coeff = lec_coeff(proc)

    si (proc_fini(proc,1))
        p = 1
    sinon
        p = (tps_cpu - tps_cpu_octroye) - min(100,tps_cpu - tps_cpu_octroye)
        si (p!=0) p = p / (p + ((tps_io - tps_io_octroye) * coeff))

    nb_genere = gen_nb()
    si (nb_genere ≤ p)
        delai_fin = min(100,tps_cpu - tps_cpu_octroye)
        proc->pass_svt = 0
    sinon
        nb_genere = gen_nb()
        delai_fin = min((int) (nb_genere * 100) + 1,tps_cpu - tps_cpu_octroye)
        proc->pass_svt = 1
```

Calc_delai_fin_io = delai_fin

```
proc = lec_prem_proc_file(file_io)
tps_cpu = lec_tps_proc(0,proc)
tps_cpu_octroye = lec_tps_proc_octroye(0,proc)
tps_io = lec_tps_proc(1,proc)
tps_io_octroye = lec_tps_proc_octroye(1,proc)
coeff = lec_coeff(proc)
tps_acces = acc_tps_accès (no_res)

delai_fin = 0
tant que ((proc->pass_svt == 1) ET (tps_ser ≤ tps_io - tps_io_octroye))
    delai_fin = delai_fin + tps_acces

si (proc_fini(proc,0))
    p = 1
sinon
    p = ((tps_io - tps_io_octroye) - tps_ser) * coeff
    si (p!=0) p = p / (p + (tps_cpu - tps_cpu_octroye))

nb_genere = gen_nb()
si (nb_genere ≤ p)
    proc->pass_svt = 1
sinon
    proc->pass_svt = 0
```

Calc_delai_fin_prn = delai_fin

```
proc = lec_prem_proc_file(file_prn)
delai_fin = lec_tps_proc(2,proc)
```

Fonctionnalité : Génération_des_arrivées **Numéro :** 13

Arguments : no_res

Résultats : nb

Précondition : -----

Postcondition : nb = calc_nb_arr(no_res)

Structures de données : nb, no_res : INT

Commentaires : Cette fonction détermine 'nb' qui correspond au nombre de processus à générer dans la seconde (qui débute).

Fonctionnalité : Génération_une_arrivée **Numéro :** 14

Arguments : file_cpu, dos_delai, nb, nb_arr_gen, interval, no_res

Résultats : -----

Précondition : nb ≥ 0

Postcondition : (nb > 0) = >

(proc = creation_processus()
& affectation_type (proc, no_res)
& affectation_cpu (proc, no_res)
& affectation_io (proc, no_res)
& type_imp(proc, no_res) = > affectation_prn (proc, no_res)
& calc_coeff(proc)
& ajout(proc, file_cpu)
& $nb\_arr\_gen_{POST} = nb\_arr\_gen_{PRE} + 1$
& (nb_arr_gen = nb) = > arr_svte (dos_delai) = HIGH_VALUE
& (nb_arr_gen < nb) = > arr_svte (dos_delai) = interval

Structures de données : file_cpu : FILE
proc : PROCESSUS
nb, nb_arr_gen, interval, no_res : INT
dos_delai : DELAI

Commentaires : Cette fonction initialise un processus et lui attribue les valeurs en consultant les différentes distributions.

Fonctionnalité : Création_processus **Numéro :** 15

Arguments : -----

Résultats : proc

Précondition : -----

Postcondition : proc = init_nv_proc ()

Structures de données : proc : PROCESSUS

Commentaires : Cette fonction crée une occurence d'un processus.

Fonctionnalité : Affectation_type **Numéro :** 16

Arguments : proc, no_res

Résultats : -----

Précondition : -----

Postcondition : calc_type_proc (proc, no_res)

Structures de données : proc : PROCESSUS
no_res : INT

Commentaires : Cette fonction détermine le type d'un processus en respectant la distribution des types de processus du réseau.

Fonctionnalité : Affectation_cpu **Numéro :** 17

Arguments : proc, no_res

Résultats : -----

Précondition : -----

Postcondition : calc_tps_cpu (proc, no_res)

Structures de données : proc : PROCESSUS
no_res : INT

Commentaires : Cette fonction détermine le temps de service global du processus au serveur CPU. Le calcul se fait en fonction du type du processus (cfr. affectation_type (16)).

Fonctionnalité : Affectation_io **Numéro :** 18

Arguments : proc, no_res

Résultats : -----

Précondition : -----

Postcondition : calc_tps_io (proc, no_res)

Structures de données : proc : PROCESSUS
no_res : INT

Commentaires : Cette fonction détermine le temps de service global du processus au disque. Le calcul se fait en fonction du type du processus (cfr. affectation_type (16)).

Fonctionnalité : Affectation_prn **Numéro :** 19

Arguments : proc, no_res

Résultats : -----

Précondition : type_imp (proc, no_res)

Postcondition : calc_tps_prn (proc, no_res)

Structures de données : proc : PROCESSUS
no_res : INT

Commentaires: Cette fonction détermine le temps de service global du processus à l'imprimante. Le calcul se fait en fonction du type du processus (cfr. affectation_type (16)).

Fonctionnalité : Affectation_tps_service **Numéro :** 20

Arguments : dos_delai

Résultats : tps_ser, {evt}

Précondition : # {evt} = 0

Postcondition : tps_ser = min (gen_nb_arr(dos_delai),arr_svte(dos_delai),
calc_prior(dos_delai), fin[0](dos_delai),
fin[1](dos_delai),fin[2](dos_delai))
& (tps_ser = gen_nb_arr (dos_delai)) = > 'arr_svte' ∈ {evt}
& (tps_ser = arr_svte (dos_delai)) = > 'arr_svte' ∈ {evt}
& (tps_ser = calc_prior (dos_delai)) = > 'calc_prior' ∈ {evt}
& (tps_ser = fin[0] (dos_delai)) = > 'fin_cpu' ∈ {evt}
& (tps_ser = fin[1] (dos_delai)) = > 'fin_io' ∈ {evt}
& (tps_ser = fin[2] (dos_delai)) = > 'fin_prn' ∈ {evt}

Structures de données : tps_ser : INT
evt_i : EVENEMENT
dos_delai : DELAI

Commentaires : Cette fonction détermine le temps qui doit encore s'écouler avant le déclenchement de l'événement suivant. Le traitement à effectuer dépend du type d'événement.

Fonctionnalité : Choix_proc_à_servir **Numéro :** 21

Arguments : file_cpu

Résultats : -----

Précondition : -----

Postcondition :

(file_non_vide(file_cpu)) = > (∀ p ∈ file_cpu :
((lec_prior_proc(lec_prem_proc_file(file_cpu)) ≤ lec_prior_proc(p))
& (∀ p ∈ file_cpu tq
(lec_prior_proc(lec_prem_proc_file(file_cpu)) = lec_prior_proc(p)):
(lec_att_tr_proc(lec_prem_proc_file(file_cpu)) ≥ lec_att_tr_proc(p)))

Structures de données : file_cpu : FILE
p : PROCESSUS

Commentaires : Cette fonction sélectionne le processus de la file du serveur CPU qui a la plus forte priorité (ici la plus petite) et le place au début de celle-ci de manière à lui allouer le CPU.

Fonctionnalité : Trait_cpu **Numéro :** 22

Arguments : tps_ser, file_cpu, dos_res_serv, dos_delai

Résultats : -----

Constante : TRANCHE : temps d'une tranche CPU

Précondition : (tps_ser ∈ [1,TRANCHE])
& (file_non_vide(file_cpu))

Postcondition : ((maj_dos_proc(lec_prem_proc_file(file_cpu),tps_ser,0, dos_délai)
& (maj_dos_attente(file_cpu,tps_ser,0)
& (maj_dos_serveur(0,tps_ser,dos_res_serv))

Structures de données : file_cpu : FILE
tps_ser : INT
dos_res_serv : RES_SERV
dos_delai : DELAI

Commentaires : Cette fonction traite le premier processus de la file du serveur CPU. Le temps de service est passé comme paramètre (tps_ser). Durant ce service, tous les compteurs de la simulation sont mis à jour.

Fonctionnalité : Trait_disque **Numéro :** 23

Arguments : tps_ser, file_io, dos_res_serv, dos_delai

Résultats : -----

Précondition : (file_non_vide(file_io))

Postcondition : ((maj_dos_proc(lec_prem_proc_file(file_io),tps_ser,1, dos_délai)
& (maj_dos_attente(file_io,tps_ser,1)
& (maj_dos_serveur(1,tps_ser,dos_res_serv))

Structures de données : file_io : FILE
tps_ser : INT
dos_res_serv : RES_SERV
dos_delai : DELAI

Commentaires : Cette fonction traite le premier processus de la file du serveur DISQUE. Le temps de service est passé comme paramètre (tps_ser). Durant ce service, tous les compteurs de la simulation sont mis à jour.

Fonctionnalité : Trait_prn **Numéro :** 24

Arguments : tps_ser, file_prn, dos_res_serv, dos_delai

Résultats : -----

Précondition : file_non_vide(file_prn)

Postcondition : ((maj_dos_proc(lec_prem_proc(file_prn),tps_ser,2, dos_délai)
& (maj_dos_attente(file_prn,tps_ser,2)
& (maj_dos_serveur(2,tps_ser,dos_res_serv))

Structures de données : file_prn : FILE
tps_ser ,tranche, tps_util : INT
dos_res_serv : RES_SERV
dos_delai : DELAI

Commentaires : Cette fonction traite le premier processus de la file de l'imprimante. Le temps de service est passé comme paramètre (tps_ser). Durant ce service, tous les compteurs de la simulation sont mis à jour.

Fonctionnalité : Maj_priorite_attente **Numéro :** 25

Arguments : file_cpu

Résultats : -----

Précondition : (file_non_vide (file_cpu))

Postcondition : (∀ proc ∈ file_cpu : calc_priorite(proc))

Structures de données : file_cpu : FILE
proc : PROCESSUS

Commentaires : Cette fonction recalcule la priorité de tous les processus de la file d'attente du serveur CPU.

Fonctionnalité : Maj_uc **Numéro :** 26

Arguments : file_cpu

Résultats : -----

Précondition : -----

Postcondition : ($\forall$ proc $\in$ file_cpu : maj_uc_proc(proc))

Structures de données : file_cpu : FILE
proc : PROCESSUS

Commentaires : Cette fonction recalcule l'utilisation récente de l'unité centrale (UC) de tous les processus de la file d'attente du CPU.

Fonctionnalité : Prise_mesure_syst **Numéro :** 27

Arguments : tps_ser, nb_obs_syst, no_batch_syst, tab_tps_bloc, tab_som_syst, file_cpu, file_io, file_prn

Résultats : -----

Précondition : -----

Postcondition : mesure_syst

Structures de données : tps_ser, nb_obs_syst, no_batch_syst : INT
tab_tps_bloc, tab_som_syst : SEQ[INT]
file_cpu, file_io, file_prn : FILE

Commentaires : Cette fonction effectue toutes les mesures sur les données du système (les serveurs) intervenant dans les résultats de la simulation.

```
Mesure_syst

    si (*nb_obs_syst%((S+1)*M) = = 0)
       ind_obs_syst = 0
       *no_batch_syst = *no_batch_syst + 1

    si (ind_obs_syst < M)
       si (nbr_elts(file_cpu) > 1)
          tab_som_syst[0][*no_batch_syst] = tab_som_syst[0][*no_batch_syst] +
                                            ((nbr_elts(file_cpu)-1)*tps_ser)
       si (nbr_elts(file_io) > 1)
          tab_som_syst[1][*no_batch_syst] = tab_som_syst[1][*no_batch_syst] +
                                            ((nbr_elts(file_io)-1)*tps_ser)
       si (nbr_elts(file_prn) > 1)
          tab_som_syst[2][*no_batch_syst] = tab_som_syst[2][*no_batch_syst] +
                                                ((nbr_elts(file_prn)-1)*tps_ser)
       tab_tps_bloc[*no_batch_syst] = tab_tps_bloc[*no_batch_syst] + tps_ser
       ind_obs_syst + +
    *nb_obs_syst = *nb_obs_syst + 1
```

Fonctionnalité : Prise_mesure_proc **Numéro :** 28

Arguments : nb_obs_proc, no_batch_proc, tab_som_att, tab_som_rep, tab_som_rep_type, nb_obs_type, no_batch_type

Résultats : -----

Précondition : -----

Postcondition : mesure_proc

Structures de données : nb_obs_proc, no_batch_proc, nb_obs_type, no_batch_type : INT
tab_som_att, tab_som_rep, tab_som_rep_type : SEQ[INT]

Commentaires : Cette fonction effectue toutes les mesures qui concernent les processus et qui interviennent dans les résultats de la simulation.

```
Mesure_proc

    si (*nb_obs < ((K + (K-1)*S)*M))
      si (*nb_obs%((S + 1)*M) = 0)
         ind_obs_proc = 0
         *no_batch_proc = *no_batch_proc + 1
      si (ind_obs_proc < M)
         tab_som_att[0][*no_batch_proc] = tab_som_att[0][*no_batch_proc] +
                                          lec_tps_file_proc(0,ptr_proc)
         tab_som_att[1][*no_batch_proc] = tab_som_att[1][*no_batch_proc] +
                                          lec_tps_file_proc(1,ptr_proc)
         tab_som_att[2][*no_batch_proc] = tab_som_att[2][*no_batch_proc] +
                                          lec_tps_file_proc(2,ptr_proc)
         tab_som_rep[*no_batch_proc] = tab_som_rep[*no_batch_proc] +
                                          calc_tps_rep(ptr_proc)
         ind_obs_proc + +
      *nb_obs = *nb_obs + 1

    type = lec_type(ptr_proc)
    si (nb_obs_type[type] < ((K + (K-1)*S)*M))
      si (nb_obs_type[type]%((S + 1)*M) = = 0)
         ind_obs_type[type] = 0
         no_batch_type[type] = no_batch_type[type] + 1
      si (ind_obs_type[type] < M)
         tab_som_rep_type[type][no_batch_type[type]] =
           tab_som_rep_type[type][no_batch_type[type]] + calc_tps_rep(ptr_proc)
         ind_obs_type[type] + +
       nb_obs_type[type] = nb_obs_type[type] + 1
```

Fonctionnalité : Régime_transitoire **Numéro :** 29

Arguments : alpha, nb, nb_arr_gen, file_cpu, file_io, file_prn,dos_res_serv, dos_delai, no_res

Résultats : res_simul

Constante : N -> nombre de simulations à effectuer

Précondition : nb_simul = 0

Postcondition : (nb_simul < N)
=> (Init_reg_trans (file_cpu, file_io, file_prn, dos_res_serv,
dos_delai, nb, nb_arr_gen)
& res_trans = Trait_trans (0, nb, nb_arr_gen,file_cpu, file_io,
file_prn, dos_res_serv, dos_delai, no_res)
& tab_obs = Calc_res_trans (res_trans, file_cpu, file_io,
file_prn, nb_simul)
& $nb\_simul_{POST} = nb\_simul_{PRE} + 1$
& régime_transitoire (nb, nb_arr_gen, file_cpu,file_io, file_prn,
dos_res_serv, dos_delai, no_res)
(nb_simul = N) => res_simul = calc_interval

Structures de données : nb, nb_arr_gen, no_res : INT
file_cpu, file_io, file_prn : FILE
dos_res_serv : RES_SERV
dos_delai : DELAI
alpha : REEL
res_trans : RES_TRANS
tab_obs : TAB_OBS

Commentaires : Cette fonction effectue N simulations en régime transitoire et calcule les intervalles de confiance.

- Calc_interval

formule volume I page 35 et 73

où X = file[i][j] (tab_obs) $0 \leq i \leq 2$ $0 \leq j \leq N-1$

X = att[i][j] (tab_obs) $0 \leq i \leq 2$ $0 \leq j \leq N-1$

X = rep[j] (tab_obs) $0 \leq j \leq N-1$

Fonctionnalité : Init_reg_trans **Numéro :** 30

Arguments : file_cpu, file_io, file_prn, dos_res_serv, dos_delai, nb, nb_arr_gen

Résultats : -----

Constante : HIGH_VALUE -> 999999

Précondition : -----

Postcondition : $\forall$ i : $0 \leq i \leq 2$: fin[i](dos_delai) = HIGH_VALUE
& arr_svte(dos_delai) = HIGH_VALUE
& calc_prior(dos_delai) = 1000
& gen_nb_arr(dos_delai) = 0
& Vider_file (file_cpu)
& Vider_file (file_io)
& Vider_file (file_prn)
& Init_dos_res_serv (dos_res_serv)
& nb = 0
& nb_arr_gen = 0

Structures de données : nb, nb_arr_gen : INT
file_cpu, file_io, file_prn : FILE
dos_res_serv : RES_SERV
dos_delai : DELAI

Commentaires : Cette fonction initialise les variables utilisées pour la simulation en régime transitoire.

Fonctionnalité : Trait_trans **Numéro :** 31

Arguments : tps_simul, alpha, nb, nb_arr_gen, file_cpu, file_io, file_prn, dos_res_serv, dos_delai, no_res

Résultats : res_trans

Constante : BORNE_SIM -> durée d'une simulation en régime transitoire

Précondition : tps_simul = 0

Postcondition : (tps_simul < BORNE_SIM)
=> (Trait_reg_trans (tps_simul, alpha, nb, nb_arr_gen, file_cpu, file_io, file_prn, dos_res_serv, dos_delai, no_res)

Structures de données : tps_simul, nb, nb_arr_gen, no_res : INT
file_cpu, file_io, file_prn : FILE
alpha : REEL
dos_res_serv : RES_SERV
dos_delai : DELAI

Commentaires : Cette fonction effectue une simulation en régime transitoire.

Fonctionnalité : Calc_res_trans **Numéro :** 32

Arguments : res_trans, file_cpu, file_io, file_prn, nb_simul

Résultats : tab_obs

Précondition : ----------

Postcondition : calc_res

Structures de données : file_cpu, file_io, file_prn : FILE
res_trans : RES_TRANS
nb_simul : INT

Commentaires : Cette fonction calcule les résultats des N simulations.

Calc_res

```
si File_non_vide (file_cpu)
    file[0][nb_simul] (tab_obs) = file[0][nb_simul] + (Nbr_elts(file_cpu) - 1)
    taux_occ[0] (tab_obs) = taux_occ[0] (tab_obs) + 1

si File_non_vide (file_io)
    file[1][nb_simul] (tab_obs) = file[1][nb_simul] + (Nbr_elts(file_io) - 1)
    taux_occ[1] (tab_obs) = taux_occ[1] (tab_obs) + 1

si File_non_vide (file_prn)
    file[2][nb_simul] (tab_obs) = file[2][nb_simul] + (Nbr_elts(file_prn) - 1)
    taux_occ[2] (tab_obs) = taux_occ[2] (tab_obs) + 1

pour  i=0 à 2
    att[i][nb_simul] (tab_obs) = tps_file[i] (res_trans)
    rep[nb_simul] (tab_obs) =    rep[nb_simul] (tab_obs)
                               + tps[i] (res_trans)
                               + tps_file[i] (res_trans)

rep_type[type(res_trans)][nb_obs_type[type(res_trans)](tab_obs)] (tab_obs) =
                    tps[0] (res_trans)
                  + tps[1] (res_trans)
                  + tps[2] (res_trans)
                  + tps_file[0] (res_trans)
                  + tps_file[1] (res_trans)
                  + tps_file [2] (res_trans)

nb_obs_type[type(res_trans)] (tab_obs) =
                    nb_obs_type[type(res_trans)] (tab_obs) + 1
```

6. Glossaire des primitives

Glossaire des primitives

(par ordre numérique)

1. Init_file (file)
2. Init_dos_res_serv (dos_res_serv)
3. Nb_syst (file_1, file2, file3) = nb
4. Nbr_elts (file) = nb
5. Student (ddl, alpha) = nb
6. Lec_tps_util (serveur, dos_res_serv) = tps
7. File_non_vide (file) = b
8. Lec_prem_proc_file (file) = proc
9. Proc_fini (proc, serveur) = b
10. Suppression (file) = file'
11. Ajout (proc, file) = file'
12. Lec_pass_svt_proc (proc) = svt
13. Lec_type (proc) = type
14. Lec_tps_proc (proc, serveur) = tps
15. Lec_tps_proc_octroye (proc, serveur) = tps
16. Lec_coeff (proc) = coeff
17. Gen_nb () = nb
18. Acc_tps_acces (no_res) = tps
19. Calc_nb_arr () = nb
20. Type_imp (proc) = b
21. Pas_de_prn (proc) = proc'
22. Calc_coeff (proc) = proc'
23. Init_nv_proc () = proc
24. Calc_type_proc (proc) = proc'
25. Calc_type (nb) = type
26. Calc_tps_cpu (proc) = proc'
27. Val_distrib (serveur, proc, nb) = valeur
28. Calc_tps_io (proc) = proc'
29. Calc_tps_prn(proc) proc'
30. Ecr_coeff (proc) = proc'
31. Lec_prior_proc (proc) = priorité
32. Lec_att_tr_proc (proc) = att_tranche

33. Calc_tps_rep (proc) = tps
34. Maj_dos_proc (proc, tps,serveur, dos_delai) = proc'
35. Maj_priorité_proc_servi (proc, tps, dos_delai) = proc'
36. Calc_priorité (proc) = proc'
37. Maj_dos_attente (file, tps, serveur) = file'
38. Maj_att_proc (proc, tps, serveur) = proc'
39. Maj_dos_serveur (serveur, tps, dos_res_serv) = dos_res_serv'
40. Maj_uc_proc (proc) = proc'
41. Vider_file (file) = file'
42. Acc_distr_arr (no_res) = distr
43. Acc_distr_type (no_res) = distr
44. Acc_distr_serveur (no_res, serveur, type) = distr

Glossaire des primitives

(par ordre alphabétique)

42. Acc_distr_arr (no_res) = distr
44. Acc_distr_serveur (no_res, serveur, type) = distr
43. Acc_distr_type (no_res) = distr
18. Acc_tps_acces (no_res) = tps
11. Ajout (proc, file) = file'
22. Calc_coeff (proc) = proc'
19. Calc_nb_arr () = nb
36. Calc_priorité (proc) = proc'
26. Calc_tps_cpu (proc) = proc'
28. Calc_tps_io (proc) = proc'
29. Calc_tps_prn(proc) proc'
33. Calc_tps_rep (proc) = tps
24. Calc_type_proc (proc) = proc'
25. Calc_type (nb) = type
30. Ecr_coeff (proc) = proc'
7. File_non_vide (file) = b
17. Gen_nb () = nb
2. Init_dos_res_serv (dos_res_serv)
1. Init_file (file)
23. Init_nv_proc () = proc
32. Lec_att_tr_proc (proc) = att_tranche
16. Lec_coeff (proc) = coeff
12. Lec_pass_svt_proc (proc) = svt
8. Lec_prem_proc_file (file) = proc
31. Lec_prior_proc (proc) = priorité
14. Lec_tps_proc (proc, serveur) = tps
15. Lec_tps_proc_octroye (proc, serveur) = tps
6. Lec_tps_util (serveur, dos_res_serv) = tps
13. Lec_type (proc) = type
38. Maj_att_proc (proc, tps, serveur) = proc'
37. Maj_dos_attente (file, tps, serveur) = file'
34. Maj_dos_proc (proc, tps,serveur, dos_delai) = proc'

39. Maj_dos_serveur (serveur, tps, dos_res_serv) = dos_res_serv'
35. Maj_priorité_proc_servi (proc, tps, dos_delai) = proc'
40. Maj_uc_proc (proc) = proc'
3. Nb_syst (file_1, file2, file3) = nb
4. Nbr_elts (file) = nb
21. Pas_de_prn (proc) = proc'
9. Proc_fini (proc, serveur) = b
5. Student (ddl, alpha) = nb
10. Suppression (file) = file'
20. Type_imp (proc) = b
27. Val_distrib (serveur, proc, nb) = valeur
41. Vider_file (file) = file'

7. Définition formelle des primitives

Primitive : init_file (file)

Numéro : 1

Objet : file_d'attente

Opération : mise à jour

Précondition : -----

Postcondition : file = []

Structure de données : file : FILE

Commentaire : Cette primitive initialise la file d'attente.

Primitive : Init_dos_res_serv (dos_res_serv)

Numéro : 2

Objet : serveur

Opération : mise à jour

Précondition : -----

Postcondition : tps_util [0-->2] (dos_res_serv) = 0

Structure de données : dos_res_serv : RES_SERV

Commentaire : Cette primitive initialise les temps d'occupation des trois serveurs (temps d'occupation initial = 0).

Primitive : Nb_syst (file1, file2, file3) = nb

Numéro : 3

Objet : file_d'attente

Opération : calcul

Précondition : -----

Postcondition : nb = nbr_elts (file1) + nbr_elts (file2) + nbr_elts (file3)

Structure de données : file1, file2, file3 : FILE
nb : INT

Commentaire : Cette primitive renvoie le nombre de processus présents dans le système.

Primitive : Nbr_elts (file) = nb

Numéro : 4

Objet : file_d'attente

Opération : calcul

Précondition : -----

Postcondition : nb = Card (file)

Structure de données : file : FILE
nb : INT

Commentaire : Cette primitive renvoie le nombre d'éléments présents dans la file passée en paramètre.

Primitive : Student (ddl, alpha) = nb

Numéro : 5

Objet : nombre aléatoire

Opération : consultation

Précondition : ddl < 30 AND alpha {'0,01', '0,05'}

Postcondition : nb = $t_{ddl,alpha}$

Structure de données : ddl : INT
alpha, nb : REEL

Commentaire : Cette primitive renvoie la valeur de la table de Student avec 'ddl' degrés de liberté et 'alpha' degré d'incertitude.

Primitive : Lec_tps_util (serveur, dos_res_serv) = tps

Numéro : 6

Objet : serveur

Opération : consultation

Précondition : -----

Postcondition : tps = tps_util[serveur] (dos_res_serv)

Structure de données : dos_res_serv : RES_SERV
serveur, tps : INT

Commentaire : Cette primitive renvoie le temps d'utilisation du serveur passé en paramètre.

Primitive : File_non_vide (file) = b

Numéro : 7

Objet : file_d'attente

Opération : consultation

Précondition : -----

Postcondition : (nbr_elts (file) > 0) => b = TRUE
(nbr_elts (file) = 0) => b = FALSE

Structure de données : b : BOOLEEN
file : FILE

Commentaire : Cette primitive teste si la file passée en paramètre contient des éléments. Elle renvoie :
- VRAI si la file contient au moins un élément
- FAUX sinon

Primitive : Lec_prem_proc_file (file) = proc

Numéro : 8

Objet : file_d'attente

Opération : consultation

Précondition : -----

Postcondition : proc = First (file)

Structure de données : file : FILE
proc : PROCESSUS

*Commentaire :*Cette primitive renvoie le premier processus de la file.

Primitive : Proc_fini (proc, serveur) = bool

Numéro : 9

Objet : processus

Opération : consultation

Précondition : -----

Postcondition :

Lec_tps_octroye_proc(proc,serveur) = Lec_tps_octroye_proc(proc,serveur)

=> bool = TRUE

Lec_tps_octroye_proc(proc,serveur) ≠ Lec_tps_octroye_proc(proc,serveur)

=> bool = FALSE

Structure de données : proc : PROCESSUS
serveur : INT

Commentaire : Cette primitive permet de déterminer si le processus a reçu tout son temps de service pour le serveur passé en paramètre.

Primitive : Suppression (file) = file'

Numéro : 10

Objet : file_d'attente

Opération : mise-à-jour

Précondition : file_non_vide (file)

Postcondition : Rem_ith (file, 1)

Structure de données : file : FILE

Commentaire : Cette primitive retire le premier processus de la file passée en paramètre. Cela correspond à un processus qui a obtenu tout son temps de service (le premier processus de la file correspond au processus dans le serveur).

Primitive : Ajout (proc, file) = file '

Numéro : 11

Objet : file_d'attente

Opération : mise-à-jour

Précondition : proc file

Postcondition : Append (file, proc)

Structure de données : file : FILE
proc : PROCESSUS

Commentaire : Cette primitive ajoute le processus ' proc ' à la fin de la file.

Primitive : Lec_pass_svt_proc (proc) = svt

Numéro : 12

Objet : processus

Opération : consultation

Précondition : -----

Postcondition : svt = pass_svt (proc)

Structure de données : proc : PROCESSUS
svt : INT

Commentaire : Cette primitive renvoie la valeur de la variable qui permet de déterminer si le processus reste au même serveur ou s'il change après son temps de service.

Primitive : Lec_type (proc) = type

Numéro : 13

Objet : processus

Opération : consultation

Précondition : -----

Postcondition : type = type (proc)

Structure de données : proc : PROCESSUS
type : INT

Commentaire : Cette primitive renvoie le type du processus.

Primitive : Lec_tps_proc (proc, serveur) = tps

Numéro : 14

Objet : processus

Opération : consultation

Précondition : -----

Postcondition : tps = tps[serveur] (proc)

Structure de données : proc : PROCESSUS
tps : INT

Commentaire : Cette primitive renvoie le temps global de service du processus au serveur passé en paramètre.

Primitive : Lec_tps_proc_octroyé (proc, serveur) = tps

Numéro : 15

Objet : processus

Opération : consultation

Précondition : -----

Postcondition : tps = tps_octroye[serveur] (proc)

Structure de données : proc : PROCESSUS
tps : INT

Commentaire : Cette primitive renvoie le temps déjà passé par le processus dans le serveur passé en paramètre.

Primitive : Lec_coeff (proc) = coeff

Numéro : 16

Objet : processus

Opération : consultation

Précondition : -----

Postcondition : coeff = coeff (proc)

Structure de données : proc : PROCESSUS
coeff : REEL

Commentaire : Cette primitive renvoie le coéfficient utilisé pour le calcul de 'pass_svt'.

Primitive : gen_nb () = nb

Numéro : 17

Objet : nb_aléatoire

Opération : calcul

Précondition : -----

Postcondition : nb [0,1[&
nb est un nombre aléatoire

Structure de données : nb : REEL

Commentaire : Cette primitive renvoie un nombre aléatoire compris entre 0 et 1 à partir des paramètres (de la suite aléatoire) utilisés pour calculer la distribution des arrivées. Le nombre généré doit satisfaire les tests statistiques de fréquence et de corrélation sérielle.

Primitive : Acc_tps_acces (no_res) = tps

Numéro : 18

Objet : acc_bd

Opération : calcul

Précondition : -----

Postcondition :

tps_depl := depl_disk (:SERVEUR (COMPOSITION : Id_res = no_res))
& tps_rot := Rot_disk (:SERVEUR (COMPOSITION : Id_res = no_res))
& tps_lec := Trans_disk (:SERVEUR (COMPOSITION : Id_res = no_res))
& tps = tps_dep + (tps_rotation / 2) + tps_lecture

Structure de données : tps, tps_depl, tps_rot, tps_lec, no_res : INT

Commentaire : Cette primitive calcule le temps nécessaire au disque pour effectuer une opération d'opération Entrée-Sortie.

Primitive : Calc_nb_arr(no_res) = nb_arr

Numéro : 19

Objet : distributions

opération : consultation

Précondition : -----

Postcondition : distr = Acc_distr_arr (no_res)
& (nb_genere = gen_nb())
& (i tq (0 ≤ i ≤ Nb_elt(distr) - 1)
& (nb(distr) [i] ≤ nb_genere ≤ nb(distr) [i + 1]))
= > (nb_arr = val(distr) [i]))

Structure de données : nb_genere : REEL
i, nb_arr, no_res : INT
distr : DISTRIBUTION

Commentaire : Cette primitive calcule le nombre d'arrivée pour une tranche de temps.

Primitive : Type_imp (proc, no_res) = b

Numéro : 20

Objet : processus

opération : consultation

Précondition : -----

Postcondition : val_distrib (2, proc, 1, no_res) = 0 = > b = FALSE
val_distrib (2, proc, 1, no_res) < > 0 = > b = TRUE

Structure de données : b : BOOLEEN
no_res : INT

Commentaire : Cette primitive renvoie TRUE si le processus est de type 'impression'; FALSE sinon.

Primitive : Pas_de_prn (proc)

Numéro : 21

Objet : processus

opération : mise à jour

Précondition : -----

Postcondition : Lec_tps_proc (proc, 2) = 0

Structure de données : proc : PROCESSUS

Commentaire : Cette primitive met le temps de service d'impression du processus à zéro.

Primitive : Calc_coeff (proc)

Numéro : 22

Objet : processus

Opération : mise-a-jour

Précondition : -----

Postcondition :

Lec_tps_proc(proc, 0) > Lec_tps_proc(proc, 1)
AND Lec_tps_proc(proc, 1) NOT = 0
= > Ecr_coeff (proc, Lec_tps_proc(proc, 0)
/ Lec_tps_proc(proc, 1))
& Lec_tps_proc (proc, 1) = 0
OR Lec_tps_proc (proc, 0) ≤ Lec_tps_proc (proc, 1)
= > Ecr_coeff (proc, 1)

Structure de données : proc : PROCESSUS

Commentaire : Cette primitive calcule le coéfficient qui permet de déterminer dans quel serveur le processus va se diriger après sont temps de service.

Primitive : Init_nv_proc() = proc

Numéro : 23

Objet : processus

Opération : calcul

Précondition : -----

Postcondition : type(proc) = 0
& tps[0à2] (proc) = 0
& tps_octroye[0à2] (proc) = 0
& tps_file[0à2] (proc) = 0
& attr_priorité_init (proc)
& priorité (proc) = priorité_init (proc)
& uc (proc) = 0
& att_tranche (proc) = 0

Structure de données : proc : PROCESSUS

*Commentaire :*Cette primitive crée un nouveau processus, et l'initialise.

Primitive : Calc_type_proc (proc, no_res) = proc'

Numéro : 24

Objet : processus

Opération : mise-à-jour

Précondition : -----

Postcondition : (nb_genere = gen_nb())
& (type(proc) = calc_type(nb_genere, no_res))

Structure de données : nb_genere : REEL
proc : PROCESSUS
no_res : INT

Commentaire : Cette primitive calcule le type d'un processus. Pour ce, elle consulte la distribution des types correspondant au réseau simulé.

Primitive : Calc_type (nb_genere, no_res) = type

Numéro : 25

Objet : distributions

opération : consultation

Précondition : (nb_genere [0,1[)

Postcondition : distr = Acc_distr_type (no_res)
& (i tq (0 ≤ i ≤ Nb_elt(distr) - 1)
& (nb(distr) [i] ≤ nb_genere ≤ nb(distr) [i + 1]))
=> (type = val(distr) [i])
& (0 < type ≤ NB_TYPE - 1)

Structure de données : nb_genere : REEL
i, type, no_res : INT
distr : DISTRIBUTION

Commentaire : Cette primitive calcule le type d'un processus à partir du nombre 'nb_genere'.

Primitive : Calc_tps_cpu (proc, no_res) = proc'

Numéro : 26

Objet : processus

Opération : mise-à-jour

Précondition : type(proc) [1, NB_TYPE[

Postcondition : (nb_genere = gen_nb())
& (tps[0](proc) = val_distrib (0,proc,nb_genere, no_res))

Structure de données : nb_genere : REEL
proc : PROCESSUS
no_res : INT

Commentaire : Cette primitive calcule le temps de service CPU d'un processus en fonction de son type.

Primitive : Val_distrib (serveur, proc, nb, no_res) = valeur

Numéro : 27

Objet : distributions

Opération : consultation

Constante : NB_TYPE : nombre de types de processus du réseau

Précondition : -----

Postcondition :

distr = Acc_distr_serveur (no_res, serveur, Lec_type (proc))
& (i tq (0 ≤ i ≤= Nb_elt(distr) - 1)
& (nb(distr)[i] ≤ nb ≤ nb(distr) [i + 1]))
= > (valeur = val(distr) [i])
& (0 < type ≤ NB_TYPE - 1)

Structure de données : distr : DISTRIBUTION
proc : PROCESSUS
valeur : INT

Commentaire : Cette primitive renvoie la valeur de la distribution à partir de la probabilité 'nb'.

Primitive : Calc_tps_io (proc, no_res) = proc'

Numéro : 28

Objet : processus

Opération : mise-à-jour

Précondition : (type(proc) [1,4] & tps[0](proc) NOT = 0)

Postcondition :

nb_genere = gen_nb ( )
& tps_accès = calc_tps_accès()
& nb = val_distrib (1,proc,nb_genere, no_res)
& tps[1](proc) = nb * Acc_tps_accès (no_res)
& tps_io_tr(proc) = up(nb / (up(tps[0](proc) / tranche)))
* Acc_tps_accès (no_res)

Structure de données : nb_genere : REEL
nb, tranche, tps_accès, no_res : INT
proc : PROCESSUS

Commentaire : Cette primitive calcule le temps de service du serveur disque ainsi que le nombre d'i/o à faire par tranche de temps CPU.

Primitive : Calc_tps_prn (proc, no_res) = proc'

Numéro : 29

Objet : processus

Opération : mise-à-jour

Précondition : type(proc) [0, 3]

Postcondition : (nb_genere = gen_nb())
& (tps[2](proc) = val_distrib (2,proc,nb_genere, no_res))

Structure de données : nb_genere : REEL
proc : PROCESSUS

*Commentaire :*Cette primitive calcule le temps de service à l'imprimante d'un processus en fonction de son type.

Primitive : Lec_proc_suivant (file, proc) = p

Numéro : 30

Objet : file_d'attente

Opération : consultation

Précondition : -----

Postcondition : $\exists$ i tq : $file_i$ = proc => p = $file_{i+1}$

Structure de données : file : FILE
p, proc : PROCESSUS

Commentaire : Cette primitive renvoie l'élément de la file d'attente se situant à la suite de l'élément passé en paramètre.

Primitive : Lec_prior_proc (proc) = priorité

Numéro : 31

Objet : processus

Opération : consultation

Précondition : -----

Postcondition : priorité = priorité (proc)

Structure de données : proc : PROCESSUS
priorité : INT

Commentaire : Cette primitive renvoie la priorité du processus contenu dans l'élément de la file passé en paramètre.

Primitive : Lec_att_tr_proc (proc) = att_tranche

Numéro : 32

Objet : processus

Opération : consultation

Précondition : -----

Postcondition : att_tranche = att_tranche (proc)

Structure de données : att_tranche : INT
proc : PROCESSUS

Commentaire : Cette primitive renvoie le temps d'attente du processus dans la tranche.

Primitive : Lec_dern_proc_file (file) = proc

Numéro : 33

Objet : file_d'attente

Opération : consultation

Précondition : -----

Postcondition : proc = Last (file)

Structure de données : file : FILE
proc : PROCESSUS

*Commentaire :*Cette primitive renvoie le dernier élément de la file.

Primitive : Maj_dos_proc (proc,tps,serveur, dos_delai) = proc'

Numéro : 34

Objet : processus

Opération : mise-à-jour

Précondition : serveur [0, 2] & tps > 0

Postcondition :

fin_{POST}[serveur](dos_delai) = fin_{PRE}[serveur](dos_delai) - tps
& $tps\_octroye_{POST}$[serveur](proc) = $tps\_octroye_{PRE}$[serveur](proc) + tps
& (serveur = 0 = > maj_priorite_proc_servi (proc, tps, dos_delai))

Structure de données : proc : PROCESSUS
tps, serveur : INT
dos_delai : DELAI

Commentaire : Cette primitive met à jour les informations d'un processus ayant été servi; c-à-d
- incrémentation de son temps de service
- si le serveur est le CPU, calcul de la nouvelle priorité

Primitive : Maj_priorité_proc_servi (proc, tps, dos_delai) = proc'

Numéro : 35

Objet : processus

Opération : mise-à-jour

Précondition : -----

Postcondition : $UC(proc)_{POST} = (UC(proc)_{PRE} + tps)$
& fin[0] (dos_delai) = 0 => (calc_priorite (proc)
& (att_tranche (proc) = 0)

Structure de données : tps : INT
proc : PROCESSUS
dos_delai : DELAI

Commentaire : Cette primitive recalcule la priorité du processus qui vient d'être servi par le serveur CPU.

Primitive : Calc_priorite (proc) = proc'

Numéro : 36

Objet : processus

Opération : mise-à-jour

Précondition : -----

Postcondition : ($UC(proc)_{POST} = UC(proc)_{PRE} / 2$)
& (priorite(proc) = priorite_init(proc) + ($UC(proc)_{POST} /2$))

Structure de données : proc : PROCESSUS

Commentaire : Cette primitive recalcule la priorité d'un processus.

Primitive : Maj_dos_attente(file,tps,serveur) = file'

Numéro : 37

Objet : file_d'attente

Opération : mise-à-jour

Précondition : file_non_vide(file)

Postcondition : (proc Tail (file) : maj_att_proc(proc,tps,serveur))

Structure de données : file : FILE
tps, serveur : INT
proc, $file_1$: PROCESSUS

Commentaire : Cette primitive met à jour le temps d'attente de tous les processus de la file du serveur 'serveur' en ajoutant 'tps' au temps d'attente de la bonne file.

Primitive : Maj_att_proc(proc,tps,serveur) = proc'

Numéro : 38

Objet : processus

Opération : mise-à-jour

Précondition : (serveur [0, 2]) & (tps > 0)

Postcondition : (tps_file_{POST}[serveur] (proc) =

tps_file_{PRE}[serveur](proc) + tps)

& ((serveur = 0) => ($att\_tranche_{POST}$(proc) =

$att\_tranche_{PRE}$(proc) + tps))

Structure de données : tps, serveur : INT
proc : PROCESSUS

Commentaire : Cette primitive met à jour le temps d'attente du processus 'proc' dans la file du serveur 'serveur' en lui ajoutant 'tps'.

Primitive : Maj_dos_serveur(serveur,tps,dos_res_serv) = dos_res_serv'

Numéro : 39

Objet : serveur

Opération : mise-à-jour

Précondition : (serveur [0, 2]) & (tps > 0)

Postcondition : (tps_util_{POST}[serveur](dos_res_serv) =
tps_util_{PRE}[serveur](dos_res_serv) + tps)

Structure de données : tps, serveur : INT
dos_res_serv : RES_SERV

Commentaire : Cette primitive met à jour le temps d'utilisation du 'serveur' en lui ajoutant 'tps'.

Primitive : Maj_uc_proc (proc) = proc'

Numéro : 40

Objet : processus

Opération : mise-à-jour

Précondition : -----

Postcondition : uc_{POST} (proc) = uc_{PRE} (proc) / 2

Structure de données : proc : PROCESSUS

Commentaire : Cette primitive met à jour l'utilisation récente de l'unité centrale (UC).

Primitive : Vider_file (file) = file'

Numéro : 41

Objet : file_d'attente

Opération : mise à jour

Précondition : -----

Postcondition : File_non_vide (file) = > Suppression (file)

Structure de données : file : FILE

Commentaire : Cette primitive vide la file d'attente.

Primitive : Acc_distr_arr (no_res) = distr

Numéro : 42

Objet : acc_bd

Opération : consultation

Précondition : -----

Postcondition : distr := distr (:distribution (arrivee : Id_res = no_res))

Structure de données : distr : DISTRIBUTION
no_res : INT

Commentaire : Cette primitive renvoie la distribution des arrivées du réseau.

Primitive : Acc_distr_type (no_res) = distr

Numéro : 43

Objet : acc_bd

Opération : consultation

Précondition : -----

Postcondition :

som = $\sum$ nb(:processus (emission : id_res = no_res)
& i = 0
& prec = 0
& $\forall$ proc : Id_proc (:processus (emission : Id_res = no_res)
=> val[i] (distr) = nb (proc)
rep[i] (distr) = prec + nb (proc) / som
prec = rep[i] (distr)
i = i + 1

Structure de données : distr : DISTRIBUTION
i, prec, som, no_res : INT
proc : PROCESSUS

Commentaire : Cette primitive renvoie la distribution des types de processus du réseau.

Primitive : Acc_distr_serveur (no_res, serveur, type) = distr

Numéro : 44

Objet : acc_bd

Opération : consultation

Précondition : -----

Postcondition :

distr = distr (:distribution (:ser_pr_dist : type_proc = type(proc))
AND (:ser_pr_dist : Id_res = no_res)

Structure de données : distr : DISTRIBUTION
no_res, serveur : INT
proc : PROCESSUS

Commentaire : Cette primitive renvoie la distribution de service global du serveur passé en paramètre en fonction du type du processus.

8. Liens entre les fonctions

FONCTION : SIMULATION

APPELLE : INITIALISATION
PRISE_DE_MESURES
AFFICH_RESULTATS

FONCTION : INITIALISATION

APPELLE : INIT_ FILE
INIT_ DOS_ RES_ SERV

FONCTION : PRISE DE MESURES

APPELLE : TROUV_PERIODE_TRANS
REGIME_STATIONNAIRE
REGIME_TRANSITOIRE

FONCTION : TROUV_PERIODE_TRANS

APPELLE : ESTIMATION_NO
VERIFICATION_NO

FONCTION : ESTIMATION_NO

APPELLE : TRAIT_REG_TRANS
NB_SYST

FONCTION : VERIFICATION_NO

APPELLE : TRAIT_REG_TRANS
NB_SYST
STUDENT

FONCTION : REGIME_STATIONNAIRE

APPELLE : ANALYSE_DONNEES
CALC_INTERVAL_CONF
LEC_TPS_UTIL

FONCTION : ANALYSE_DONNEES

APPELLE : TRAIT_SIMUL

FONCTION : CALC_INTERVAL_CONF

APPELLE : STUDENT

FONCTION : AFFICH_RESULTATS

APPELLE : ----------

FONCTION : TRAIT_REG_TRANS

APPELLE : AFFECTATION_TPS_SERVICE
TRAIT_CPU
TRAIT_DISQUE
TRAIT_PRN
FILE_NON_VIDE
GENERATION_DES_ARRIVEES
LEC_PREM_PROC_FILE
PROC_FINI
GENERATION_UNE_ARRIVEE
SUPPRESSION
AJOUT
MAJ_PRIORITE_ATTENTE
MAJ_UC
CHOIX_PROC_A_SERVIR
LEC_TPS_PROC
LEC_TPS_FILE_PROC
LEC_TYPE
LEC_PASS_SVT_PROC
LEC_TPS_PROC_OCTROYE
LEC_COEFF
GEN_NB
ACC_TPS_ACCES

FONCTION : TRAIT_SIMUL

APPELLE : AFFECTATION_TPS_SERVICE
TRAIT_CPU
TRAIT_DISQUE
TRAIT_PRN
FILE_NON_VIDE
GENERATION_DES_ARRIVEES
LEC_PREM_PROC_FILE
PROC_FINI
PRISE_MESURE_SYST
GENERATION_UNE_ARRIVEE
PRISE_MESURE_PROC
SUPPRESSION
AJOUT
MAJ_PRIORITE_ATTENTE
MAJ_UC
CHOIX_PROC_A_SERVIR
LEC_TPS_PROC
LEC_TPS_FILE_PROC
LEC_TYPE
LEC_PASS_SVT_PROC
LEC_TPS_PROC_OCTROYE
LEC_COEFF
GEN_NB
ACC_TPS_ACCES

FONCTION : GENERATION_DES_ARRIVEES

APPELLE : CALC_NB_ARR

FONCTION : GENERATION_UNE_ARRIVEE

APPELLE : CREATION_PROCESSUS
AFFECTATION_TYPE
AFFECTATION_CPU
AFFECTATION _IO
AFFECTATION_PRN
TYPE_IMP
PAS_DE_PRN
CALC_COEFF
AJOUT

FONCTION : CREATION_PROCESSUS

APPELLE : INIT_NV_PROC

FONCTION : AFFECATION_TYPE

APPELLE : CALC_TYPE_PROC

FONCTION : AFFECTATION_CPU

APPELLE : CALC_TPS_CPU

FONCTION : AFFECTATION_IO

APPELLE : CALC_TPS_IO

FONCTION : AFFECTATION_PRN

APPELLE : CALC_TPS_PRN
TYPE_IMP

FONCTION : AFFECTATION_TPS_SERVICE

APPELLE : ----------

FONCTION : CHOIX_PROC_A_SERVIR

APPELLE : FILE_NON_VIDE
LEC_PREM_PROC_FILE
LEC_PRIOR_PROC
LEC_ATT_TR_PROC

FONCTION : TRAIT_CPU

APPELLE : LEC_PREM_PROC_FILE
MAJ_DOS_PROC
MAJ_DOS_ATTENTE
MAJ_DOS_SERVEUR

FONCTION : TRAIT_DISQUE

APPELLE : LEC_PREM_PROC_FILE
MAJ_DOS_PROC
MAJ_DOS_ATTENTE
MAJ_DOS_SERVEUR

FONCTION : TRAIT_PRN

APPELLE : LEC_PREM_PROC_FILE
MAJ_DOS_PROC
MAJ_DOS_ATTENTE
MAJ_DOS_SERVEUR

FONCTION : MAJ_PRIORITE_ATTENTE

APPELLE : CALC_PRIORITE

FONCTION : MAJ_UC

APPELLE : MAJ_UC_PROC

FONCTION : PRISE_MESURE_SYST

APPELLE : NBR_ELTS

FONCTION : PRISE_MESURE_PROC

APPELLE : LEC_TPS_FILE_PROC
CALC_TPS_REP
LEC_TYPE

FONCTION : REGIME_TRANSITOIRE

APPELLE : INIT_REG_TRANS
TRAIT_TRANS
CALC_RES_TRANS
STUDENT

FONCTION : INIT_REG_TRANS

APPELLE : VIDER_FILE
INIT_DOS_RES_SERV

FONCTION : TRAIT_TRANS

APPELLE : TRAIT_REG_TRANS

FONCTION : CALC_RES_TRANS

APPELLE : FILE_NON_VIDE
NBR_ELTS

9. Liens entre les primitives

PRIMITIVE : INIT_FILE	
Appelle :	Est appelé par :
	- Fonction 2

PRIMITIVE : INIT_DOS_RES_SERV	
Appelle :	Est appelé par :
	- Fonction 2 - Fonction 30

PRIMITIVE : NB_SYST	
Appelle :	Est appelé par :
- Nbr_elts	- Fonction 6 - Fonction 5

PRIMITIVE : NBR_ELTS	
Appelle :	Est appelé par :
	- Fonction 27 - Fonction 32 - Nb_syst - File_non_vide

PRIMITIVE : STUDENT	
Appelle :	Est appelé par :
	- Fonction 6 - Fonction 9 - Fonction 29

PRIMITIVE : LEC_TPS_UTIL	
Appelle :	Est appelé par :
	- Fonction 7

PRIMITIVE : FILE_NON_VIDE	
Appelle :	Est appelé par :
	- Fonction 11 - Fonction 12 - Fonction 21 - Fonction 32 - Nbr_elts - Vider_file

PRIMITIVE : LEC_PREM_PROC_FILE	
Appelle :	Est appelé par :
	- Fonction 11 - Fonction 12 - Fonction 22 - Fonction 23 - Fonction 24 - Fonction 21

PRIMITIVE : PROC_FINI	
Appelle :	Est appelé par :
	- Fonction 11 - Fonction 12

PRIMITIVE : SUPPRESSION	
Appelle :	Est appelé par :
	- Fonction 11 - Fonction 12 - Vider_file

PRIMITIVE : AJOUT	
Appelle :	Est appelé par :
	- Fonction 11 - Fonction 12 - Fonction 14

PRIMITIVE : LEC_PASS_SVT_PROC	
Appelle :	Est appelé par :
	- Fonction 11 - Fonction 12

PRIMITIVE : LEC_TYPE	
Appelle :	Est appelé par :
	- Fonction 11 - Fonction 12 - Fonction 28 - Val_distrib

PRIMITIVE : LEC-TPS_PROC	
Appelle :	Est appelé par :
	- Fonction 11 - Fonction 12 - Proc_fini - Pas_de_prn

PRIMITIVE : LEC_TPS_PROC_OCTROYE	
Appelle :	Est appelé par :
	- Fonction 11 - Fonction 12 - Proc_fini

PRIMITIVE : LEC_COEFF	
Appelle :	Est appelé par :
	- Fonction 11 - Fonction 12

PRIMITIVE : GEN_NB	
Appelle :	Est appelé par :
	- Fonction 11 - Fonction 12 - Calc_nb_arr - Calc_type_proc - Calc_tps_cpu - Calc_tps_io - Calc_tps_cpu

PRIMITIVE : ACC_TPS_ACCES	
Appelle :	Est appelé par :
	- Fonction 11 - Fonction 12 - Calc_tps_io

PRIMITIVE : CALC_NB_ARR	
Appelle :	Est appelé par :
- Gen_nb - Acc_distr_arr	- Fonction 13

PRIMITIVE : TYPE_IMP	
Appelle :	Est appelé par :
- Val_distrib	- Fonction 14 - Fonction 19

PRIMITIVE : PAS_DE_PRN	
Appelle :	Est appelé par :
- Lec_tps_proc	- Fonction 14

PRIMITIVE : CALC_COEFF	
Appelle :	Est appelé par :
- Lec_tps_proc - Ecr_coeff	- Fonction 14

PRIMITIVE : INIT_NV_PROC	
Appelle :	Est appelé par :
	- Fonction 15

PRIMITIVE : CALC_TYPE_PROC	
Appelle :	Est appelé par :
- Calc_type	- Fonction 16

PRIMITIVE : CALC_TYPE	
Appelle :	Est appelé par :
- Acc_distr_type	- Calc_type_proc

PRIMITIVE : CALC_TPS_CPU	
Appelle :	Est appelé par :
- Gen_nb - Val_distrib	- Fonction 17

PRIMITIVE : VAL_DISTRIB	
Appelle :	Est appelé par :
- Acc_distr_serveur	- Type_imp - Calc_tps_cpu - Calc_tps_io - Calc_tps_prn

PRIMITIVE : CALC_TPS_IO	
Appelle :	Est appelé par :
- Gen_nb - Acc_tps_accès - Val_distrib	- Fonction 18

PRIMITIVE : CALC_TPS_PRN	
Appelle :	Est appelé par :
- Gen_nb - Val_distrib	- Fonction 19

PRIMITIVE : ECR_COEFF	
Appelle :	Est appelé par :
	- Calc_coeff

PRIMITIVE : LEC_PRIOR_PROC	
Appelle :	Est appelé par :
	- Fonction 21

PRIMITIVE : LEC_ATT_TR_PROC	
Appelle :	Est appelé par :
	- Fonction 21

PRIMITIVE : CALC_TPS_REP	
Appelle :	Est appelé par :
	- Fonction 28

PRIMITIVE : MAJ_DOS_PROC	
Appelle :	Est appelé par :
- Maj_priorité_proc_servi	- Fonction 22 - Fonction 23 - Fonction 24

PRIMITIVE : MAJ_PRIORITE_PROC_SERVI	
Appelle :	Est appelé par :
- Calc_priorité	- Maj_dos_proc

PRIMITIVE : CALC_PRIORITE	
Appelle :	Est appelé par :
	- Fonction 25 - Maj_priorité_proc_servi

PRIMITIVE : MAJ_DOS_ATTENTE	
Appelle :	Est appelé par :
- Maj_att_proc	- Fonction 22 - Fonction 23 - Fonction 24

PRIMITIVE : MAJ_ATT_PROC	
Appelle :	Est appelé par :
	- Maj_dos_attente

PRIMITIVE : MAJ_DOS_SERVEUR	
Appelle :	Est appelé par :
	- Fonction 22 - Fonction 23 - Fonction 24

PRIMITIVE : MAJ_UC_PROC	
Appelle :	Est appelé par :
	- Fonction 26

PRIMITIVE : VIDER_FILE	
Appelle :	Est appelé par :
- Suppression - File_non_vide	- Fonction 30

PRIMITIVE : ACC_DISTR_ARR	
Appelle :	Est appelé par :
	- Calc_nb_arr

PRIMITIVE : ACC_DISTR_TYPE	
Appelle :	Est appelé par :
	- Calc_type

PRIMITIVE : ACC_DISTR_SERVEUR	
Appelle :	Est appelé par :
	- Val_distrib

10. Les primitives classées par objet

OBJET : FILE_D'ATTENTE

INTERFACE

INVARIANT : /

FONCTIONS : INIT_FILE
NB_SYST
NBR_ELTS
FILE_NON_VIDE
LEC_PREM_PROC_FILE
SUPPRESSION
AJOUT
VIDER_FILE

INIT_FLE : FILE --> FILE
NB_SYST : FILE X FILE X FILE --> INT
NBR_ELTS : FILE --> INT
FILE_NON_VIDE : FILE --> BOOLEEN
LEC_PREM_PROC_FILE : FILE --> PROC
SUPPRESSION : FILE --> FILE
AJOUT : PROC X FILE --> FILE
VIDER_FILE : FILE --> FILE

OBJET : PROCESSUS

INTERFACE

INVARIANT : /

FONCTIONS : PROC_FINI
LEC_PASS_SVT_PROC
LEC_TYPE
LEC_TPS_PROC
LEC_TPS_PROC_OCTROYE
LEC_COEFF
TYPE_IMP
PAS_DE_PRN
CALC_COEFF
INIT_NV_PROC
CALC_TYPE_PROC
CALC_TPS_CPU
CALC_TPS_IO
CALC_TPS_PRN
LEC_PRIOR_PROC
LEC_ATT_TR_PROC
MAJ_DOS_PROC
MAJ_PRIORITE_PROC_SERVI
CALC_PRIORITE
MAJ_DOS_ATTENTE
MAJ_ATT_PROC
MAJ_UC_PROC

PROC_FINI : PROCESSUS X INT --> BOOLEEN
LEC_PASS_SVT_PROC : PROCESSUS --> INT
LEC_TYPE : PROCESSUS --> INT
LEC_TPS_PROC : PROCESSUS X INT --> INT
LEC_TPS_PROC_OCTROYE : PROCESSUS X INT --> INT
LEC_COEFF : PROCESSUS --> INT
TYPE_IMP : PROCESSUS --> BOOLEEN
PAS_DE_PRN : PROCESSUS --> PROCESSUS
CALC_COEFF : PROCESSUS --> PROCESSUS
INIT_NV_PROC : NIL --> PROCESSUS

CALC_TYPE_PROC : PROCESSUS --> PROCESSUS
CALC_TPS_CPU : PROCESSUS --> PROCESSUS
CALC_TPS_IO : PROCESSUS --> PROCESSUS
CALC_TPS_PRN : PROCESSUS --> PROCESSUS
LEC_PRIOR_PROC : PROCESSUS --> INT
LEC_ATT_TR_PROC : PROCESSUS --> INT
MAJ_DOS_PROC : PROCESSUS --> PROCESSUS
MAJ_PRIORITE_PROC_SERVI : PROCESSUS --> PROCESSUS
CALC_PRIORITE : PROCESSUS --> PROCESSUS
MAJ_DOS_ATTENTE : PROCESSUS --> PROCESSUS
MAJ_ATT_PROC : PROCESSUS --> PROCESSUS
MAJ_UC_PROC : PROCESSUS --> PROCESSUS

OBJET : NOMBRE_ALEATOIRE

INTERFACE

INVARIANT : /

FONCTION : GEN_NB

GEN_NB : NIL --> REEL

OBJET : ACC_BD

INTERFACE

INVARIANT : /

FONCTIONS : ACC_TPS_ACCES
ACC_DISTR_ARR
ACC_DISTR-TYPE
ACC_DISTR_SERVEUR

CALC_TPS_ACCES : NIL --> INT
ACC_DISTR_ARR : INT --> DISTRIBUTION
ACC_DISTR-TYPE : INT --> DISTRIBUTION
ACC_DISTR_SERVEUR : INT X INT X INT --> DISTRIBUTION

OBJET : SERVEUR

INTERFACE

INVARIANT : /

FONCTIONS : INIT_DOS_RES_SERV
LEC_TPS_UTIL
MAJ_DOS_SERVEUR

INIT_DOS_RES_SERV : RES_SERV --> RES_SERV
LEC_TPS_UTIL : INT X RES_SERV --> INT
MAJ_DOS_SERVEUR : INT X INT X RES_SERV --> RES_SERV

OBJET : DISTRIBUTION

INTERFACE

INVARIANT : /

FONCTIONS : STUDENT
CALC_NB_ARR
CALC_TYPE
VAL_DISTRIB

STUDENT : INT X REEL --> REEL
CALC_NB_ARR : NIL --> INT
CALC_TYPE : REEL --> INT
VAL_DISTRIB : INT X PROCESSUS X REEL --> INT

11. Schéma de la base de données

```
OPEN DATABASE simulation;


CREATE TABLE distribution
   ( id_distr INT NOT NULL PRIMARY KEY,
     lib_distr CHAR(30) DEFAULT "Distribution inconnue" );




CREATE TABLE reseau
   ( id_res INT NOT NULL PRIMARY KEY,
     prop_res CHAR(30) DEFAULT "inconnu",
     type_res CHAR(15) DEFAULT NULL,
     distr_arr INT
       REFERENCES distribution
                  ON UPDATE RESTRICT
                  ON DELETE RESTRICT );




CREATE TABLE processus
   ( id_proc INT NOT NULL PRIMARY KEY,
     type_proc CHAR(30) DEFAULT "inconnu",
     nb_proc INT NOT NULL);




 CREATE TABLE emission
   ( id_res INT NOT NULL REFERENCES reseau
                         ON UPDATE RESTRICT
                         ON DELETE CASCADE,
     id_proc INT NOT NULL REFERENCES processus
                          ON UPDATE RESTRICT
                          ON DELETE CASCADE,
     PRIMARY KEY (id_res,id_proc) );




CREATE TABLE rep_horaire
    ( id_res INT NOT NULL REFERENCES reseau
                          ON UPDATE RESTRICT
                          ON DELETE CASCADE,
      heure INT NOT NULL,
      PRIMARY KEY (id_res,heure),
      repartition INT NOT NULL )
      CHECK (heure BETWEEN 9 AND 16);
```

```
CREATE TABLE serveur
   ( id_ser INT NOT NULL PRIMARY KEY,
     lib_ser CHAR(30) DEFAULT "modèle inconnu",
     type_ser INT NOT NULL,
     vit_cpu INT DEFAULT NULL,
     cap_disque INT DEFAULT NULL,
     rot_disque INT DEFAULT NULL,
     depl_disque INT DEFAULT NULL,
     bloc_disque INT DEFAULT NULL,
     transfert_disque INT DEFAULT NULL,
     vit_prn INT DEFAULT NULL,
     buf_prn INT DEFAULT NULL )
   CHECK (type_ser BETWEEN 1 AND 3);



CREATE TABLE composition
   ( id_res INT NOT NULL REFERENCES reseau
                ON UPDATE RESTRICT
                ON DELETE CASCADE,
     id_ser INT NOT NULL REFERENCES serveur
                ON UPDATE RESTRICT
                ON DELETE CASCADE,
     PRIMARY KEY (id_res,id_ser) );



CREATE TABLE ser_pr_dist
   ( id_ser INT NOT NULL REFERENCES serveur
                ON UPDATE RESTRICT
                ON DELETE CASCADE,
     id_dist INT NOT NULL REFERENCES distribution
                ON UPDATE RESTRICT
                ON DELETE CASCADE,
     id_proc INT NOT NULL REFERENCES processus
                ON UPDATE RESTRICT
                ON DELETE CASCADE,
     PRIMARY KEY (id_ser,id_dist,id_proc) );



DISCONNECT simulation;
```