Précis de système d'exploitation
Les processus

1ième Année Informatique et Mathématiques Appliquées
Gérard Padiou

18 février 2004

Table des matières

1 Les processus **2**
1.1 Origine et motivation 2
1.2 Le concept de processus 3
1.3 Les opérations sur les processus 3
1.4 L'ordonnancement des processus 5

2 L'environnement d'exécution d'un processus **6**
2.1 La communication par événements asynchrones 6
2.2 La communication par flots de données 7

3 Conclusion **8**

1 Les processus

1.1 Origine et motivation

Une tâche fondamentale d'un système d'exploitation est d'assurer l'exécution de programmes divers et variés...

Ces programmes peuvent être de différentes natures :
- programme développé par un programmeur pour une application ;
- programme assurant une fonction dans la production même d'autres programmes : compilateurs, interpréteurs, éditeur de liens, metteur au point ;
- programme dit "système" assurant une tâche dans le fonctionnement même du système : interpréteur de commande par exemple.

Les concepteurs de systèmes d'exploitation se sont très vite aperçus que l'on pouvait envisager de gérer l'exécution de plusieurs programmes en parallèle En effet, une architecture classique de machine, même mono-processeur, autorise le parallélisme entre l'échange d'informations entre les périphériques (disques, bandes imprimantes, scanners, CD,...) et la mémoire centrale d'une part, et d'autre part, l'exécution d'un programme par le(s) processeur(s) avec cependant un problème de contention et de partage d'accès à la mémoire centrale qui apparaît alors comme une ressource commune partagée. La figure (1) illustre ce parallélisme des "entrées/sorties" avec l'exécution des programmes.

"

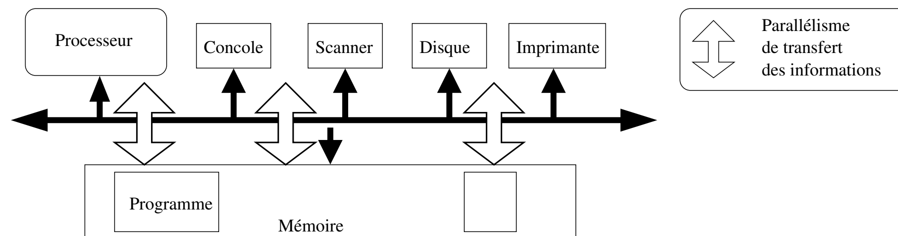

Fig. 1 – Parallélisme entre entrées/sorties et exécution d'un programme

Or, on a grand intérêt à exploiter ce parallélisme. Prenons l'exemple d'un interpréteur de commandes. Son schéma d'exécution comporte des lectures répétitives du clavier pour lire la commande de l'usager. Tant que la ligne de commande n'a pas été frappée, le programme interpréteur n'a rien d'autre à faire que de tester de façon continue si le "return" de fin de ligne a bien été lu et placé dans un octet de la mémoire centrale. Ce test incessant est une boucle qui va être exécutée durant plusieurs secondes. Autrement dit, le processeur exécute plusieurs millions de fois la même instruction de test. Le processeur pourrait donc être plus efficacement utilisé si l'on pouvait assurer l'exécution d'un autre programme pendant cette période où l'interpréteur est "logiquement" bloqué. Pour cela, plusieurs conditions sont néanmoins nécessaires :
- il faut savoir suspendre momentanément l'exécution d'un programme : le programme interpréteur doit être arrêté, son état courant d'exécution doit être sauvegardé en mémoire de telle sorte que l'on puisse continuer son exécution ultérieurement. Ceci implique de savoir sauvegarder/restaurer un contexte d'exécution de programme. C'est pourquoi, tout processeur possède un contexte minimal d'exécution rassemblant les informations caractéristiques et suffisantes pour pouvoir, si elles sont rechargées dans les registres du processeur, reprendre l'exécution en son point de sauvegarde. Cet ensemble d'informations se présente sous la forme d'un "mot d'état programme" (PSW : Program Status Word), car la sauvegarde de l'état en mémoire centrale (en général par empilement) occupe un (double)-mot ;
- l'interface de supervision des entrées/sorties doit pouvoir émettre une interruption vers le processeur lorsqu'un échange de données est terminé afin que le noyau système puisse enregistrer que le programme ayant lancé l'échange peut désormais continuer. Cette gestion des événements asynchrones de fin de lecture ou d'écriture met en jeu intensivement le mécanisme d'interruption. On peut remarquer que le traitement d'une interruption implique aussi la sauvegarde de l'état du programme qui est interrompu. Le processeur commute de l'exécution du programme (interrompu) à l'exécution de la routine de

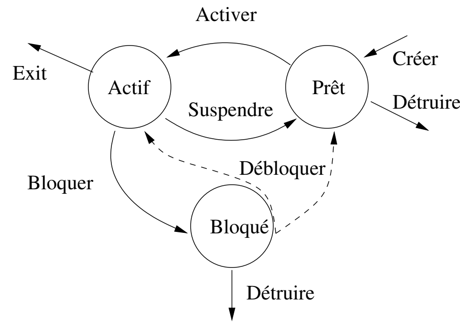

Fig. 2 – Diagramme de transitions d'état d'un processus

traitement de l'interruption. En fin de routine, le contrôle doit être rendu au programme interrompu et donc le processeur commute de l'exécution de la routine à l'exécution du programme interrompu.

Ainsi, d'un point de vue logique, le processeur(s) ne s'arrête(nt) jamais, mais il(s) commute(nt) fréquemment d'un programme à un autre, ce programme pouvant être un programme applicatif ou un programme "système" tel que l'interpréteur de commandes. Le noyau, quant à lui, sert d'intermédiaire et de superviseur de ces commutations (via l'exécution de primitives ou de routines d'exception). C'est pourquoi, les concepteurs de système ont eu l'idée d'abstraire, sous la forme d'un concept et aussi d'un objet, la gestion de l'exécution d'un programme sous le contrôle d'un noyau d'exploitation. Ce concept est celui de processus.

1.2 Le concept de processus

Une processus modélise l'exécution d'un programme. Il capte le caractère dynamique d'un traitement réalisé par un programme. Il ne doit donc pas être confondu avec la notion de programme qui capte l'aspect statique et descriptif d'un traitement. On peut par exemple comparer le programme à une bobine de film et le processeur à l'appareil de projection. Alors, le processus sera ... la séance de cinéma. S'il y a une interruption de la séance de cinéma, on sait qu'il faut reprendre la projection au point où elle avait été interrompue. C'est la même chose pour l'exécution d'un programme.

Un processus est donc une entité dynamique qui a une durée de vie limitée (la durée de l'exécution du programme) et dont on peut caractériser le comportement à un certain niveau d'abstraction par un diagramme de transitions d'état. La figure (2) illustre les règles comportementales d'un processus en distinguant 3 états :

- Dans l'état actif, le processus exécute des instructions d'un programme. Il monopolise un processeur.
- Dans l'état prêt, le processus est arrêté. Il ne lui manque que la ressource processeur pour devenir actif. Autrement dit, il suffira de lui allouer le (un) processeur et de charger le contexte d'exécution sauvegardé pour que celui-ci poursuive l'exécution du programme dont il a la charge.
- Dans l'état bloqué, le processus est aussi arrêté. Mais, pour se poursuivre, une condition logique devra de plus devenir vraie. Par exemple, le processus interpréteur attendant qu'une ligne de commande soit frappée est bloqué tant que cette ligne n'est pas disponible. La sortie de l'état bloqué pourra conduire le processus soit dans l'état actif, soit dans l'état prêt. Ce choix est fixé par la stratégie d'ordonnancement des processus vis-à-vis de la ressource processeur.

On désigne souvent l'ensemble composé des processus prêts et actifs comme l'ensemble des processus exécutables.

1.3 Les opérations sur les processus

Le diagramme de transitions fait apparaître les primitives applicables aux objets processus. Celles-ci définissent la classe ou le type des objets processus du point de vue du noyau de gestion des processus.

Créer

La première opération indispensable est l'opération de création d'un processus. Pour le noyau, un objet processus est un objet comme un autre. La création d'un processus apparaît donc comme la création d'un objet de la classe processus avec ses attributs et méthodes (opérations). Parmi les attributs d'un processus, on trouve par exemple, le nom du fichier d'où est issu le programme associé au processus, un nom interne de processus (numéro par exemple), l'état, le contexte initial qui sera chargé dans les registres du processeur pour débuter l'exécution, l'espace mémoire alloué pour le chargement et l'exécution du programme, certains paramètres d'exécution (priorité, délais de garde,...), le nom de l'usager pour lequel le processus "travaille", l'environnement fichier accessible (répertoire de travail), etc. Ainsi, un objet processus peut nécessiter un descripteur de plusieurs Koctets. Dans le système Unix, on distingue même une partie "noyau" du descripteur de processus (structure *sproc*) et une partie "usager" (structure *uproc*). La première est allouée dans l'espace mémoire du noyau, alors que la seconde est dans l'espace mémoire utilisateur.

Lorsqu'on crée un processus, deux approches sont possibles : soit le nouvel objet est créé de toute pièce, soit le nouvel objet est une copie d'un processus courant. Cette dernière approche est par exemple adopté par le système Unix. Elle présente l'avantage de pouvoir décomposer la création d'un processus en deux étapes : une première étape concerne l'aspect parallèle avec la création d'un processus sosie et une deuxième étape concerne l'aspect traitement (programme) avec la commutation du programme exécuté par le processus sosie. De plus, le processus fils créé hérite naturellement de tout l'environnement d'exécution du processus père (créateur). Ceci évite d'avoir à préciser explicitement un grand nombre de paramètres de création.

Enfin, la gestion globale des processus par le noyau comporte souvent la structuration de l'ensemble des processus existants selon une arborescence fondée sur la relation de création. Cette structure permettra en particulier à un processus père d'attendre la terminaison d'un ou plusieurs de ses fils assurant ainsi une synchronisation globale entre processus.

Activer

Un processus prêt peut devenir actif par l'opération *Activer*. Cette opération consiste essentiellement, à charger dans les registres du processeur le contexte sauvegardé en mémoire. La ressource processeur est ainsi allouée au processus pour exécuter réellement un programme. La période active du processus se termine soit par un blocage, soit par une préemption du processeur au bout d'un délai fixé maximal (voir opération *Suspendre*). Cette suspension forcée d'un processus permet de répartir plus équitablement la ressource processeur entre les différents processus prêts candidats.

Suspendre

Si la commutation d'un processus à un autre n'est causée que pas le blocage d'un processus actif, un processeur peut être monopolisé par un processus pendant une longue période. Par exemple, si le programme exécuté comporte de longues étapes de calcul sur des données en mémoire centrale, il faudra attendre une lecture ou écriture sur disque pour qu'une commutation se produise. Pour mieux répartir le temps processeur entre les processus, un processeur est en général alloué au processus pour une durée maximale fixée appelée quantum. Si le processus atteint la fin de quantum, alors une interruption est provoquée et le processus actif sur le processeur interrompu est suspendu. Libérant ainsi le processeur, un autre processus prêt peut être activé.

Bloquer

Cette opération comporte une première étape identique à l'opération *Suspendre*. Mais, le processus ne peut plus être candidat à la ressource processeur. Il devra être replacé dans l'état prêt, éventuellement directement dans l'état actif, par une opération explicite *Débloquer*. Les périodes pendant lesquelles un processus est bloqué correspondent aux périodes pendant lesquelles la poursuite de l'activité du processus est conditionnée par la disponibilité d'une ressource ou l'occurrence d'un événement (fin d'entrée/sortie par exemple).

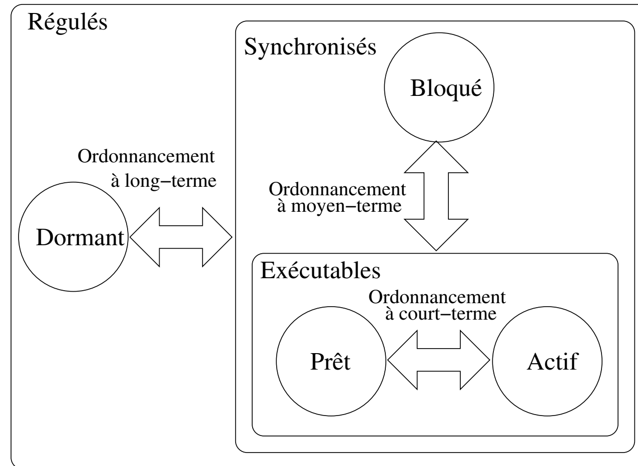

Fig. 3 – Niveaux d'ordonnancement

Cette opération a un effet de bord important : elle libère la ressource processeur pour un autre processus prêt. Par conséquent, elle conduit le noyau à enchaîner par une opération *Activer* (si possible).

Débloquer

Cette opération replace un processus parmi les processus exécutables. Elle est invoquée en général sur occurrence d'un événement asynchrone provoqué par un autre processus, une fin d'entrée/sortie ou une fin de quantum.

Exit

La terminaison d'un programme se traduit par la fin du processus mais pas l'arrêt du processeur. Par conséquent, la fin du programme doit se traduire par un appel explicite au noyau d'exécution via une primitive *Exit*. L'objet processus correspondant pourra alors être détruit sauf contraintes particulières. Une commutation de processus s'ensuivra automatiquement.

Détruire

Si un programme ne se termine pas normalement ou si une raison externe nécessite de détruire un processus (manque de ressource, surcharge du système), on pourra être amené à détruire un processus non actif.

1.4 L'ordonnancement des processus

Il faut bien distinguer trois niveaux d'ordonnancement des processus. La figure (3) illustre ces niveaux résumés dans le tableau ci-dessous :

Processus	Ordonnancement	Gestion des transitions
Exécutables	à court-terme	Prêt ⇔ Actif
Synchronisés	à moyen-terme	Exécutable ⇔ Bloqué
Régulés	à long-terme	Dormant ⇔ Synchronisé

Le niveau des processus exécutables gère l'ordonnancement à court terme qui consiste à contrôler l'allocation de la ressource processeur. Il s'agit de répartir selon une stratégie d'ordonnancement adéquate le temps processeur disponible entre les différents processus exécutables c'est-à-dire dans l'état prêt ou actif.

En général, le noyau gère donc une file d'attente des processus prêts (file simple chronologique *fifo* ou à priorité) et ceux-ci passent dans l'état actif dès qu'un processeur (et par conséquent du temps processeur) est disponible. Le choix d'un processus dans la file détermine ce qui se passe dans le système pour quelques centaines de millisecondes. On parle donc d'ordonnancement à court-terme.

Le niveau des processus synchronisés contrôle le bloquage/débloquage des processus intervenant lors de l'allocation/libération des autres ressources que processeur nécessaires à l'exécution du processus. À chaque demande de ressource physique ou logique peut être associée une file d'attente de processus bloqués et la stratégie d'ordonnancement consistera donc ici à choisir quel(s) processus débloquer lorsque la ressource attendue sera disponible. Ce choix a donc un impact sur ce qui s'exécute dans le système dans les quelques secondes qui suivent. On parle d'ordonnancement à moyen-terme.

Enfin, le niveau des processus régulés gère l'ordonnancement à long terme qui consiste à contrôler la création/destruction des processus. En effet, la décision de créer un nouveau processus peut être soumise à condition : par exemple, le nombre de processus existants est inférieur à une limite fixée. La création d'un processus peut avoir un impact sur ce qui se passe dans le système pendant une durée longue : si le processus doit exécuter un calcul de plusieurs heures ...

En résumé, l'ordonnancement des processus est une tâche complexe comportant différentes stratégies d'ordonnancement selon le niveau où l'on se place ou selon les ressources mises en jeu. Une difficulté de base réside dans la nécessité de partager équitablement les ressources entre les processus demandeurs. Un processus ne doit pas rester bloquer indéfiniment en attente d'une ressource. De nombreuses recherches ont été réalisées sur ce sujet pour optimiser le flot global de processus exécutés par le système tout en satisfaisant les contraintes temporelles d'exécution (temps de réponse) propres à chaque processus.

2 L'environnement d'exécution d'un processus

Le noyau d'exécution assure et contrôle le déroulement d'un processus de façon à garantir sa bonne exécution sans mettre en danger ni l'exécution des autres processus, ni le fonctionnement correct du noyau lui-même. Pour ce faire, il doit fournir une sorte de "machine virtuelle" si possible la plus portable possible. En effet, il est intéressant de pouvoir écrire des programmes qui s'appuient sur les services offerts par le système d'exploitation plutôt que sur les caractéristiques particulières de tel ou tel processeur ou périphérique. Entre autre chose, la machine "support" de l'exécution du processus doit être capable :

- de traiter les exceptions dues à des erreurs de programmation contenues dans un programme de façon à éviter l'arrêt du système global : seul le processus ayant provoqué l'erreur sera interrompu ;
- de communiquer des événements asynchrones externes vers le processus : par exemple, les interruptions d'entrée/sortie ;
- d'échanger des flots d'informations via les ressources périphériques, fichiers et/ou d'autres processus.

2.1 La communication par événements asynchrones

Un processus doit disposer d'un système de gestion d'exceptions lui permettant de réagir à des événements internes à l'exécution même du programme (déroutement) ou à des événements externes telles que les interruptions. Or, les concepteurs de processeurs dotent leurs architectures de systèmes d'exception très propriétaires. Bien que l'on retrouve de nombreux points communs, il existe des différences : nombre de niveaux d'interruptions, nombre exact d'interruptions, masquage ou invalidation des exceptions, etc. Face à cette hétérogénéité des processeurs, les concepteurs de système définissent un système standard d'exception.

La figure (4) illustre l'idée d'un système d'exception portable qui interagit avec le processus en cours d'exécution.

L'occurrence d'une exception provoquera une action par défaut : en général, le processus cible est arrêté et détruit. Un code de terminaison peut permettre à un autre processus de tester si le processus s'est terminé normalement ou sur exception.

Néanmoins, le noyau offre une interface de programmation sous la forme d'un ensemble de primitives qui permettent de programmer le système d'exception en offrant par exemple la possibilité :

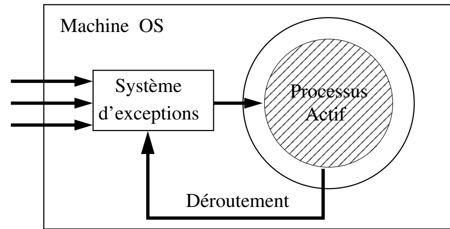

Fig. 4 – Système d'exception portable

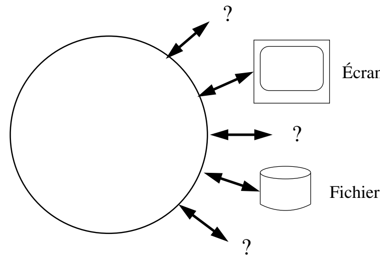

Fig. 5 – Environnement de communication d'un processus

- de masquer la plupart des exceptions ;
- de connecter une routine de traitement d'exception qui sera exécutée par le processus lui permettant ainsi de continuer son exécution après récupération de l'exception ;
- d'ignorer une exception.

2.2 La communication par flots de données

Pour exécuter un programme, un processus va consommer un ensemble de ressources du système : de la mémoire centrale, du temps processeur mais aussi des ressources d'un niveau plus abstrait tel que par exemple des fichiers et/ou les dispositifs d'entrées/sorties plus spécialisés tels que : imprimante, scanner, CD-Rom, etc. La connexion du processus à ces moyens de communication externe est dynamique. En effet, au cours de l'exécution du programme, celui-ci peut demander l'ouverture d'un flot de données avec un fichier ou une ressource périphérique précise. Le noyau d'exécution doit donc gérer, contrôler et permettre la communication du processus avec son environnement via un ensemble de canaux potentiels. Pour qu'une communication s'établisse réellement, le programme doit contenir des appels au noyau pour ouvrir et connecter ces canaux à une ressource effective qui peut être : un fichier, un (pseudo-)périphérique ou même un autre processus (via un tube ou pipe sous le système Unix par exemple).

Cette approche assure une indépendance (relative, il ne sera pas possible de lire sur une imprimante !) des programmes par rapport à l'environnement réel de leur exécution. En effet, les connexions des canaux peuvent être modifiées sans intervenir dans le code du programme. Cette possibilité est appelée redirection. à titre d'exemple, le système Unix fournit par défaut trois canaux connectés :

- *stdin* qualifiée d'entrée standard est connectée par défaut, pour un processus interactif, au clavier de l'usager connecté,
- *stdout* qualifiée de sortie standard est connectée à la fenêtre de lancement du processus,
- *stderr* deuxième sortie standard, dédiée par convention eux messages d'erreurs, connectée comme *stdout* à la fenêtre de lancement.

Ces trois connexions peuvent être modifiées : par exemple, l'entrée standard peut être redirigée sur un fichier de données et la sortie standard redirigée sur un autre fichier destiné à contenir les résultats.

Une possibilité offerte par ce mécanisme de redirection est de définir un schéma de communication entre processus via la notion de pipe ou tube. Un objet pipe est une ressource permettant de faire communiquer

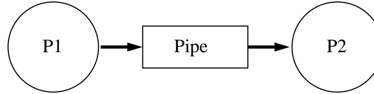

FIG. 6 – Communication par pipe

deux processus selon un schéma de producteur-consommateur. La figure (6) illustre cette connexion par pipe entre un processus producteur et un processus consommateur. Le processus producteur écrit via un canal connecté au pipe un flot d'octets. En parallèle, le processus producteur lit et consomme donc le flot d'octets produit. La consommation des données produites se fait dans l'ordre chronologique de production. Bien que le schéma de production-consommation soit asynchrone et parallèle, le producteur peut être bloqué sur la primitive d'écriture si la capacité de mémorisation du pipe est atteinte. En effet, la ressource pipe est essentiellement constituée par un tampon mémoire attribué au pipe par le noyau lors de la création de l'objet. Celui-ci n'est pas de capacité illimitée (quelques Koctets en général). De même, le processus consommateur sera bloqué en lecture si rien n'a été produit lors d'une demande de lecture.

Le mécanisme de pipe permet de réaliser très simplement un traitement parallèle de type "pipeline". Une suite de processus exécute des traitements complémentaires où chacun exploite un flot de données entrant et produit un flot résultat sortant consommé par le suivant. Les programmes ayant ce comportement générique de consommateur/producteur peuvent être considérés comme des filtres. Le système Unix offre ainsi en standard un ensemble de commandes de filtrage (*grep*, *tr*, *tail*, *head*, *cat*, *cut*, *join*,...) qui permettent de programmer " dans le large " très rapidement des applications complexes sur des flots de données.

3 Conclusion

La notion de processus intervient comme l'objet fondamental géré par un noyau d'exécution. Il permet d'assurer le suivi et le contrôle de l'exécution d'un programme de façon sûre vis-à-vis des autres programmes ou du noyau ainsi que d'exploiter le parallélisme possible entre les traitements en cours optimisant ainsi l'utilisation d'une architecture matérielle.

La définition d'un système d'exception et d'un environnement de communication par flots, indépendant des aspects matériels, assure par ailleurs une bonne portabilité des applications. Les programmes développés peuvent être écrits pour une machine virtuelle "OS" plutôt que pour la machine matérielle Sun, Intel ou Motorola.

La gestion des processus pose cependant des problèmes difficiles de programmation parallèle. En effet, ceux-ci entrent en concurrence pour accéder aux ressources du système dont ils ont besoin (mémoire, processeur, fichiers,...) et ils doivent se coordonner pour échanger des données (exemple du schéma producteur-consommateur). Les progrès technologiques aussi bien que ceux enregistrés dans le domaine de la programmation et de l'algorithmique, ont conduit à introduire un deuxième niveau de parallélisme : un processus n'exécute plus un seul programme séquentiellement, mais peut dérouler un calcul comportant plusieurs activités parallèles. On parle alors de processus légers (ou threads). Cependant, un processus (lourd par opposition) reste l'unité d'allocation de ressources pour l'exécution d'un programme désormais éventuellement parallèle.

Cet aspect a fait l'objet de longues recherches sur le thème de la synchronisation des processus (lourds ou légers). Ce sujet sera donc développé dans un module spécifique.