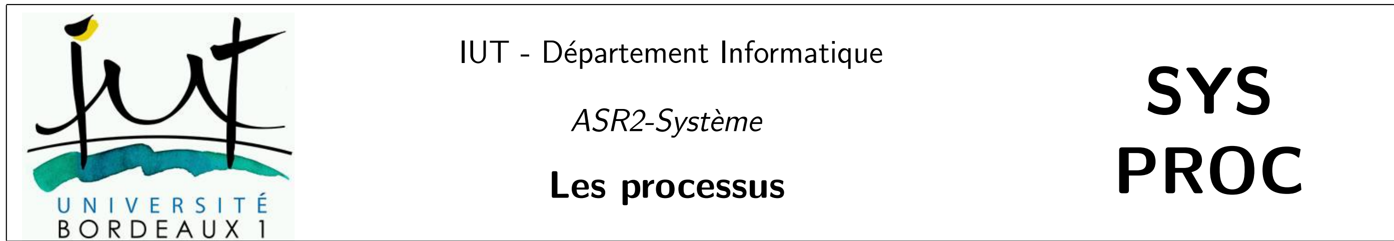


Table des matières

1 Introduction **1**

2 Histoire **1**
- 2.1 Multitâche 1
- 2.2 Exemple : intérêt du multi-tâches 2
- 2.3 Du *batch* au *time sharing* 3
- 2.4 Support matériel du multi-tâches : les interruptions 3
- 2.5 Support logiciel 3

3 Définitions **3**
- 3.1 Les processus 3
- 3.2 Les états des processus 3
- 3.3 Changements d'état 4
- 3.4 Multitâche coopératif 4
- 3.5 Scénario détaillé 5
- 3.6 Multitâche préemptif 5
- 3.7 Multitâche préemptif et utilisation interactive 6
- 3.8 Interruptions et multitâche, en résumé 7

4 Politiques d'ordonnancement **7**
- 4.1 Ordonnanceur 7
- 4.2 Critères d'évaluation 7
- 4.3 Tourniquet 7
- 4.4 Priorités 7
- 4.5 Priorités variables 8
- 4.6 Files multiples 8

Ce document est copiable et distribuable librement et gratuitement à la condition expresse que son contenu ne soit modifié en aucune façon, et en particulier que le nom de son auteur et de son institution d'origine continuent à y figurer.

1 Introduction

Les systèmes d'exploitation modernes sont tous capables de faire exécuter plusieurs tâches en même temps. Sous Unix, la commande `top` vous permet de voir les tâches en cours : sur un ordinateur personnel vous constaterez qu'il en a au moins une bonne centaine, plusieurs milliers sur un serveur.

```
Terminal
Fichier  Édition  Affichage  Terminal  Aide
top - 09:23:09 up  1:02,  3 users,  load average: 0.00, 0.01, 0.05
Tasks: 153 total,   2 running, 151 sleeping,   0 stopped,   0 zombie
Cpu(s): 15.6%us,  7.7%sy,  0.0%ni, 76.7%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   2058440k total,  1426396k used,   632044k free,    50544k buffers
Swap:  1044220k total,        0k used,  1044220k free,   877760k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2035 root      20   0  180m  99m 3032 R   17  4.9   0:24.31 puppet
 1377 avahi     20   0 37072 2624 1300 S    1  0.1   0:04.01 avahi-daemon
 1230 root      35  15 13708 3124  840 S    0  0.2   0:02.34 preload
14291 mibillau  20   0 21216 1340  948 R    0  0.1   0:00.04 top
    1 root      20   0  8404  788  660 S    0  0.0   0:01.87 init
    2 root      20   0     0    0    0 S    0  0.0   0:00.00 kthreadd
    3 root      20   0     0    0    0 S    0  0.0   0:00.14 ksoftirqd/0
    6 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/0
    7 root      RT   0     0    0    0 S    0  0.0   0:00.00 watchdog/0
    8 root      RT   0     0    0    0 S    0  0.0   0:00.00 migration/1
   10 root      20   0     0    0    0 S    0  0.0   0:00.06 ksoftirqd/1
   12 root      RT   0     0    0    0 S    0  0.0   0:00.00 watchdog/1
   13 root       0 -20     0    0    0 S    0  0.0   0:00.00 cpuset
   14 root       0 -20     0    0    0 S    0  0.0   0:00.00 khelper
   15 root      20   0     0    0    0 S    0  0.0   0:00.00 kdevtmpfs
   16 root       0 -20     0    0    0 S    0  0.0   0:00.00 netns
   17 root      20   0     0    0    0 S    0  0.0   0:00.00 sync_supers
```

Mais matériellement, un processeur ne peut exécuter qu'une instruction à la fois. Même si les ordinateurs possèdent plusieurs processeurs, on est loin du compte : il n'y a pas un processeur par programme. [1]

Le déroulement parallèle de ces tâches est donc une *illusion* : en réalité le processeur consacre un peu de temps "faire avancer" une tâche, puis passe à une autre etc. à tour de rôle. C'est la rapidité de cette alternance, quelques millisecondes par tâche, qui donne l'impression, à notre échelle, que tout avance en même temps.

Une des fonctions importantes d'un système multitâche est donc de gérer les *commutations de contexte* : il doit

- noter l'état de la tâche en cours (sauvegarde du contexte)
- choisir une des tâches (`ordonnancement`)
- la relancer dans l'état où elle était arrêtée (restauration du contexte)

2 Histoire

2.1 Multitâche

Les systèmes d'exploitation multi-tâches sont apparus très rapidement dans l'histoire de l'informatique.

- Ordinateur Gamma 60 de la société Bull, en 1958 `http://fr.wikipedia.org/wiki/Gamma_60`.
- Ordinateur LEO III de la société Lyons, 1961.

1. Dans la suite du cours, pour simplifier, on considére des machines à un seul processeur. Avec plusieurs processeurs, il y a des complications intéressantes, mais les principes de bas sont les mêmes.

La société Lyons regroupait une chaine de restaurants, des hotels et des activités dans l'alimentaire (biscuits).

Grande utilisatrice de machines à cartes perforées pour sa gestion, elle a compris avant beaucoup d'autres l'intérêt des recherches qui étaient menées sur les calculateurs électroniques (EDSAC, à Cambridge). Elle a donc embauché un technicien radar, loin de son coeur de métier, [2] pour développer ses propres ordinateurs pour ses applications de gestion. Le LEO I (Lyons Electronic Office) est sorti en 1951, et d'autres modèles ont suivi qui ont été commercialisés hors de la société Lyons. [3]

L'objectif du multitâche est évident : si plusieurs programmes s'exécutent en même temps, ils utilisent les divers périphériques en parallèle, ce qui rentabilise au mieux l'installation informatique : on augmente le nombre de programmes que l'on peut faire exécuter dans une journée d'utilisation.

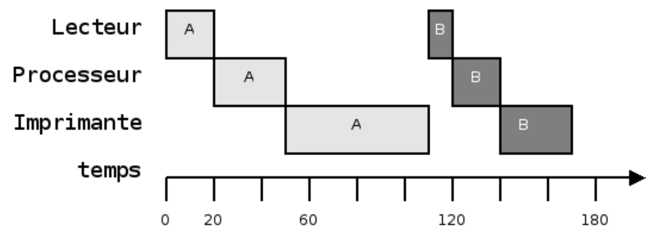
2.2 Exemple : intérêt du multi-tâches

Imaginons deux tâches A et B :

- le chargement de A depuis le lecteur de cartes dure 20 secondes, elle fait du calcul pendant 30 secondes, et l'impression des résultats prend 1 minute ;
- le chargement de la seconde B dure 10 secondes, son calcul 20 secondes et l'impression 30 secondes.

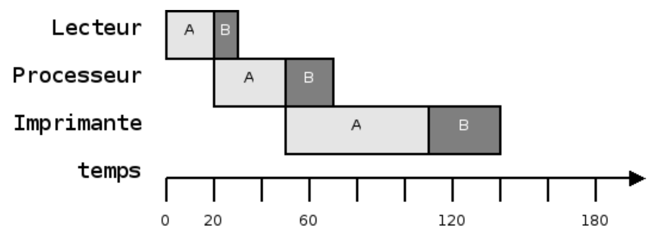
Le graphique ci-contre montre ce qui se passe sous le contrôle d'un "moniteur d'enchaînement de travaux". La tâche B n'est chargée en mémoire que quand A s'est terminée ($t = 110s$) et se termine à $t = 170s$.

Le processeur a travaillé $30 + 20 = 50s$, soit un taux d'occupation de $50/170 = 29,4\%$.



Exercice 1.
- calculez le taux d'occupation du lecteur de cartes
- calculez le taux d'occupation de l'imprimante.

Voici maintenant le déroulement dans un système multitâche ; la tâche B est chargée dès que le lecteur a été libéré, puis est exécutée quand le processeur est libre, etc.



2. Ce n'est pas un cas isolé : Honeywell (spécialiste de la régulation de chauffage) et Boeing (avions) ont aussi fabriqué des ordinateurs. Et plus tard le premier micro-ordinateur a été fabriqué en France pour l'INRA (Institut National de Recherche Agronomique), le Micral en 1972.

3. Sur le site `http://www.leo-computers.org.uk/` des anciens employés de LEO Computers, vous trouverez de nombreuses photos et descriptions (et même des enregistrements de l'ambiance des salles machine).

Exercice 2.
– Calculez les taux d'occupation, comparez avec les chiffres précédents.
– Même question si on commence par exécuter B au lieu de A.
– Imaginons qu'il s'y ajoute une troisième tâche C semblable à B. Représentez le déroulement dans les deux cas (enchaînement séquentiel et multitâche). Comparez les chiffres.

2.3 Du batch au time sharing

Traitement conversationnel. Un nouveau besoin apparait à la fin des années 50 : avec l'utilisation de terminaux interactifs. telex transformés, machines à écrire électriques, et écrans alphanumériques. Chaque utilisateur doit avoir l'impression d'utiliser une machine "réactive" : si un collègue lance un programme de calcul lourd (quelques milliers de décimales de π) [4], cela ne doit pas empêcher les autres de travailler en monopolisant complètement le temps du processeur.

C'est la prise en compte de cette contrainte qui conduit au *time sharing* : le temps du processeur doit être partagé "équitablement" entre les utilisateurs.

2.4 Support matériel du multi-tâches : les interruptions

Pour être réalisé, le multi-tâche nécessite l'ajout de quelques quelques fonctionnalités techniques sur le matériel.

En particulier, il ne serait pas raisonnable de devoir interroger constamment les périphériques pour savoir si les opérations qu'on leur a confiées (et qui sont attendues par certaines tâches) sont terminées ou non. Cette *boucle d'attente active* consommerait beaucoup de temps du processeur, temps que l'on souhaite consacrer à l'exécution de programmes "utiles".

L'ordinateur comporte donc des circuits spécialisés (contrôleurs de périphériques, en terminologie moderne) à qui le processeur confie l'exécution des entrées-sorties. Quand l'opération est terminée, le contrôleur envoie une `interruption`, signal électrique qui provoque le déroutement vers une "routine de traitement de l'interruption" située à une adresse convenue.

Dans un ordinateur multi-tâches, les interruptions "réveillent" le système d'exploitation, qui peut alors débloquer les tâches qui attendaient la fin de ces opérations.

D'autres mécanismes matériels sont également nécessaires pour la multiprogrammation, en particulier il faut *protéger l'espace mémoire de chaque tâche*, pour éviter qu'une autre tâche y accède indûment. La gestion de la mémoire fait l'objet d'un autre chapitre.

2.5 Support logiciel

Avant la multiprogrammation, les systèmes d'exploitation étaient des *moniteurs d'enchaînement de travaux* assez très rudimentaires chargés au début de la mémoire au démarrage de la machine, et dont le rôle était simplement :

1. de lire un programme exécutable (par exemple sur une bande magnétique) et de le copier un peu plus loin en mémoire,
2. de lancer son exécution,
3. et passer au suivant quand le programme s'est achevé.

Avec plusieurs tâches présentes simultanément, le système d'exploitation devient plus complexe : il doit traiter les interruptions provenant des périphériques, faire les commutations de contexte, gérer le partage de la mémoire, etc.

3 Définitions

3.1 Les processus

Le **processus** est une entité abstraite qui sert à représenter un **programme en cours d'exécution**.

Un processus regroupe :
– le **code** du programme : un espace mémoire contenant les instructions du programme ;
– un espace mémoire pour les **données** de travail (variables, pile, tas) ;
– d'**autres ressources** : descripteurs de fichiers ouverts, des ports réseau, etc.
– des **droits d'accès**

Le noyau du système d'exploitation détient une **table des processus** qui décrit l'état des processus présents dans la mémoire.

Pour chaque processus, il y a un **bloc de contrôle** (PCB, *process control block*) contient
– l'identifiant du processus
– son état : actif, prêt ou bloqué (voir plus loin)
– les valeurs des registres
– le compteur ordinal (numéro de la prochaine instruction à exécuter)
– ...

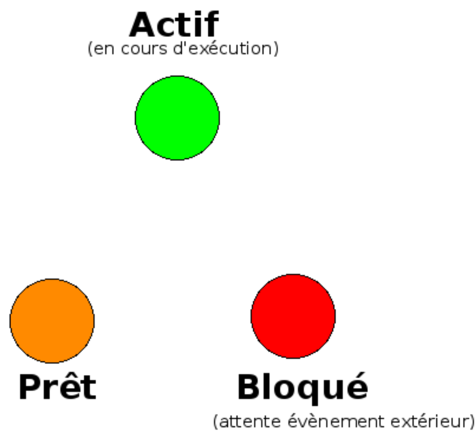
Ces informations donnent la "photographie" d'un programme au moment où il a été interrompu, et permettront de reprendre son exécution exactement là où il était arrêté, avec le même contenu dans chaque registre.

3.2 Les états des processus

Trois états sont possibles pour un processus :
– **actif** quand le processeur est en train d'exécuter une de ses instructions. Dans un système monoprocesseur,

4. ou un programme qui boucle, comme ça arrive parfois.

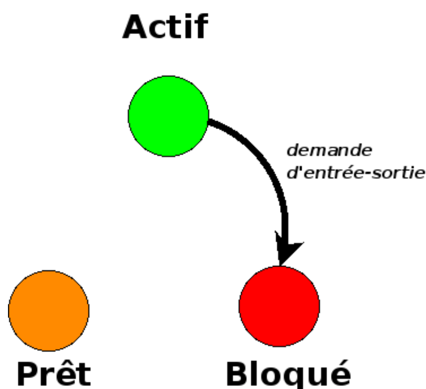
un seul processus peut être actif à la fois ;
- **bloqué** quand il est en attente d'un évènement, par exemple une lecture de données :
- **prêt** si il n'est ni actif ni bloqué.



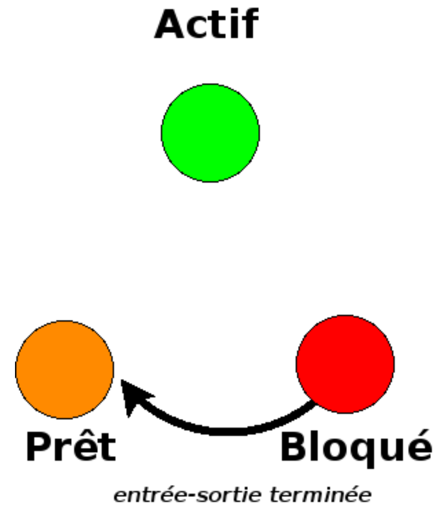
3.3 Changements d'état

1. Quand le processus actif a besoin de faire une opération d'entrée-sortie (E/S), il en fait la demande auprès du système d'exploitation par un "appel système" (sous Unix : `read`, `write`, etc.).

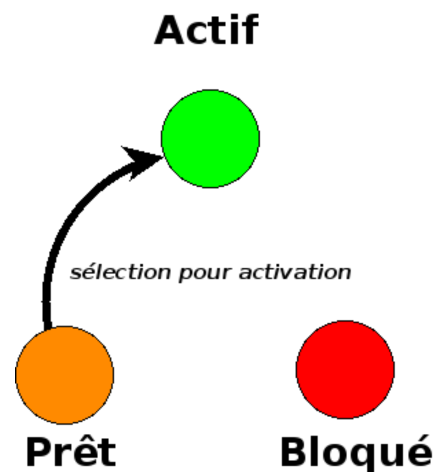
   Si il faut attendre la fin de cette opération pour continuer (par exemple pour une lecture), le système change l'état du processus, qui devient **bloqué**. On parle d'opération *bloquante*.



2. lorsqu'un périphérique signale au processeur qu'une opération d'E/S est achevée, le système d'exploitation "débloque" le processus qui attendait la fin de cette opération. Le processus est alors marqué comme **prêt**.



3. enfin, le système d'exploitation peut choisir un processus prêt pour le rendre **actif**.

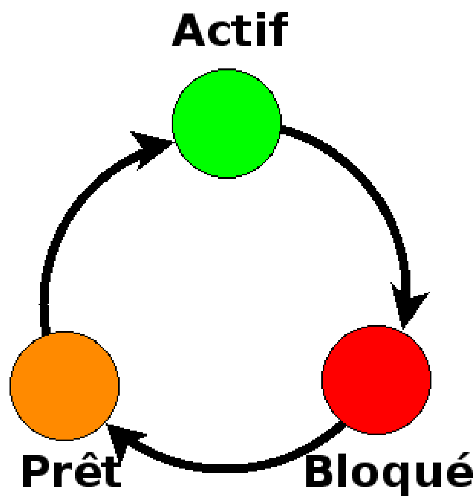


   Ce changement peut se faire quand le processus actif se termine ou se bloque, et "laisse sa place".

   À un moment donné il peut y avoir plusieurs processus prêts : le choix du processus à activer est fait par un module appelé **ordonnanceur** (**scheduler**). Diverses **politiques d'ordonnancement** sont envisageables, nous les verrons en détail plus loin.

3.4 Multitâche coopératif

Dans un système multitâche dit "coopératif", les changements d'états d'un processus se font donc selon le cycle suivant, qui résume les transitions vues plus haut :



On observe que le processus qui est actif le reste tant qu'il ne demande pas d'opérations d'entrée-sortie.

Dans un tel système, Il est donc nécessaire que les programmes soient bien écrits pour

- ne pas boucler indéfiniment
- faire des entrées-sorties de temps en temps pour "bien se comporter" envers les autres processus et leur laisser des occasions de s'activer.

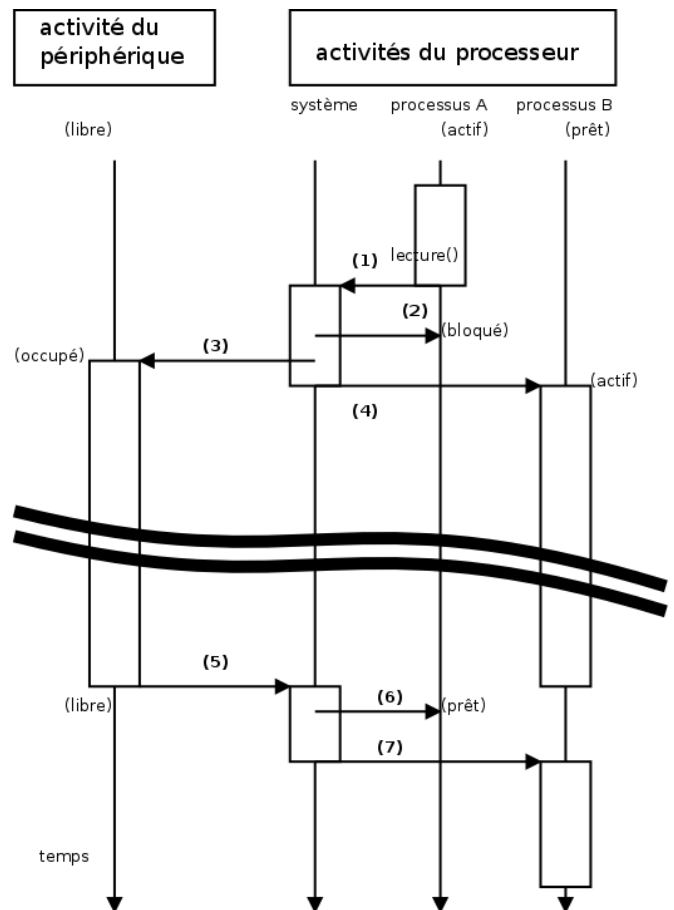
Évidemment ce genre de système marche assez mal en pratique. C'est ce qu'on trouvait dans Windows jusqu'à la version 3.11, et Mac OS jusqu'à MAC OS 9, plus de trente ans après l'invention du "vrai" multitâche !

3.5 Scénario détaillé

Le schéma ci-dessous montre le déroulement détaillé d'une opération d'entrée-sortie. On suppose qu'il y a au départ un processus A qui demande une lecture sur un périphérique inoccupé, et un processus B qui est prêt.

La scénario montre de haut en bas les "lignes de vie" des différentes activités : une pour le périphérique, et trois pour le processeur, pour distinguer ce qui relève du système, et des 2 processus.

Les flèches horizontales montrent les causalités.



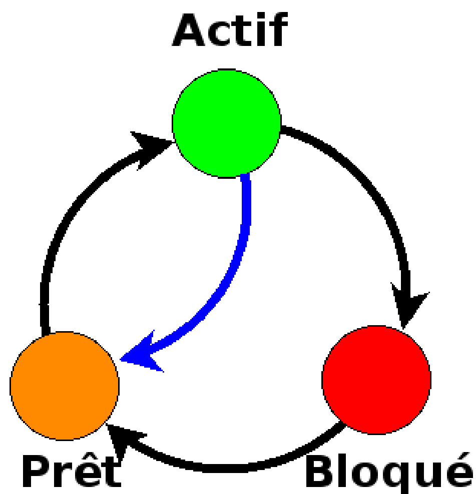
1. A fait un appel au système d'exploitation pour demander une lecture.
2. le système marque le processus A comme bloqué
3. il ordonne au périphérique de se mettre au travail
4. il active B
5. le périphérique signale la fin de l'opération
6. l'interruption rend la main au système d'exploitation, qui marque A comme prêt
7. le système rend la main au processus B

3.6 Multitâche préemptif

Le multitâche préemptif, qui traite correctement les problèmes de partage du temps, a été proposé très tôt par McCarthy et Teager. Dans un papier de 1959, McCarty écrit que "l'idée n'est pas vraiment nouvelle".

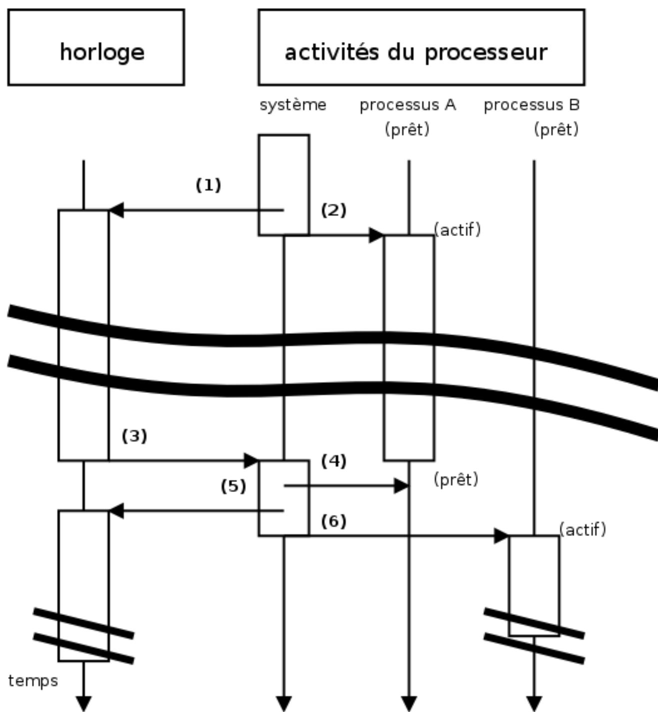
En fait, il suffit d'ajouter un circuit d'horloge qui envoie une interruption au bout d'un certain délai. Quand le système active un proessus, il programme cette horloge pour un certain **quantum de temps**. Il peut alors se passer deux choses

- soit le processus actif fait un appel système pour faire une E/S ou se terminer, et donc il "passe la main", au moins provisoirement, au système d'exploitation.
- soit il fait uniquement du calcul, et se trouve donc interrompu, automatiquement, par le signal d'horloge quand le quantum de temps est épuisé. Le processus est alors marqué comme `prêt`.



C'est ce qu'on appelle une **préemption**, capacité pour le système d'exploitation d'interrompre une tâche en cours pour activer une tâche plus prioritaire. Notez que ceci n'interdit pas que le processus ainsi préempté soit aussitôt ré-activé, si l'ordonnanceur détermine qu'il est le plus prioritaire.

Vu autrement, voici le déroulement d'une préemption, à partir du moment où l'ordonnnanceur a déterminé qu'il fallait activer le processus A :



1. le circuit d'horloge est armé pour un quantum de temps fixé
2. le processus A est activé, et se met à faire du calcul "indéfiniment"
3. le quantum est épuisé, l'horloge envoie une interruption
4. le système reprend la main
5. le processus A est marqué "prêt"
6. le processus B est choisi et activé, l'horloge est armée.

Dans ce système dit "multitâche préemptif" on a donc la garantie que le système d'exploitation reprend la main régulièrement, ce qui lui donnera l'occasion de distribuer le temps équitablement entre les processus non-bloqués, sans laisser un processus actif monopoliser le processeur.

Le multitâche préemptif est présent sur tous les ordinateurs multitâches depuis les années 60. On mesure donc à quel point, pendant la décennie 1980-1990, les principaux systèmes pour micro-ordinateurs (Windows et MacOS) étaient littéralement préhistoriques.

En effet, ces petits systèmes ont revécu, trente ans plus tard, l'histoire de l'informatique :

– au départ, petites machines à capacité très limitées
– utilisées par un seul programme à la fois, une seule personne à la fois. Dans le cas de la micro-informatique, l'objectif commercial était de vendre un ordinateur par personne, surtout pas de le partager à plusieurs !

L'évolution vers le multi-tâche a été forcée par l'utilisation d'interfaces graphiques (si on a un multi-fenêtrage, on veut fatalement y faire tourner plusieurs programmes). Le multi-tâche coopératif est assez facile à réaliser par modification d'un système mono-tâche. Par contre le passage au multitâche préemptif nécessitait une refonte complète du système, ainsi que du catalogue d'applications. [5]

3.7 Multitâche préemptif et utilisation interactive

Soient deux utilisateurs X et Y d'un ordinateur en temps-partagé. Ils lancent tous les deux, à peu près en même temps, un programme qui fait une minute de calcul.

– Dans le cas d'un système coopératif, le calcul de l'un sera effectué pendant une minute, et alors commencera le calcul du second.
– Dans un système préemptif, les deux calculs alterneront par petites tranches correspondant au quantum de temps (valeurs courantes entre 1/50 et 1/1000 de seconde). Ils finiront donc tous les deux au bout de deux minutes.

En résumé, dans le cas coopératif, l'attente moyenne des deux utilisateurs sera $\frac{60+120}{2} = 90s$, et avec le système coopératif $\frac{120+120}{2} = 120s$.

5. On le sait assez peu, mais la société Microsoft prévoyait, après le lancement de DOS 3.0, de faire évoluer son catalogue vers Xenix, un système UNIX dont elle avait acheté les droits à ATT à la fin des années 70 de façon à pouvoir enfin remplacer DOS par un vrai système multitâches. Ce plan a été abandonné vers 1985, quand Microsoft et IBM ont commencé à développer ensemble OS/2 (nom de code "CP/DOS") qui devait préserver la compatibilité avec les applications existantes. En effet, commercialement, il est mal avisé de forcer les clients à racheter tout leur parc logiciel pour bénéficier d'un nouveau système, fut-il techniquement bien meilleur. La plan OS/2 a lui même été abandonné pour la stratégie Windows NT, qui regroupe en fait les versions Windows 2000, XP, 2003, Vista, Home Server, Server 2008, et Windows 7.

Exercice 3. L'attente moyenne est-elle un bon critère pour mesurer la satisfaction des utilisateurs d'un système en temps partagé ? Pouvez-vous proposer mieux ?

3.8 Interruptions et multitâche, en résumé

Une interruption est un signal qui détourne le processeur de sa boucle d'exécution normale (lire une intruction, l'exécuter, passer à la suivante), pour effectuer un traitement particulier (routine de traitement d'interruption).

Les interruptions sont causées par
– les **périphériques** (fin d'exécution de requête)
– des **signaux d'horloge**
– des **évènements extérieurs**
– déroutements en cas d'**erreur** (accès illégal à la mémoire, division par zéro ...)
– **interruptions logicielles** provoquées par instruction spéciale

Dans un système d'exploitation multitâches, les interruptions sont traitées par le système, qui agit sur l'état des processus :
– venant d'un périphérique d'E/S : le système fait passer le processus demandeur à l'état prêt ;
– venant de l'horloge (épuisement du quantum de temps), le système fait passer le processus actif à l'état prêt ;
– interruption d'erreur (division par zero etc) : le processus actif est supprimé.

4 Politiques d'ordonnancement

4.1 Ordonnanceur

Dans un système d'exploitation multitâches, l'**ordonnanceur** (*scheduler*) est un composant du noyau, qui a pour fonction de choisir un des processus prêts pour l'activer.

L'ordonnanceur applique une **politique d'ordonnancement**, algorithme ou heuristique qui est censé donner "de bons résultats".

4.2 Critères d'évaluation

Une politique d'ordonnancement peut être évoluée selon plusieurs critères. On peut en effet souhaiter avoir
– **équité** : chaque processus dispose d'une part équitable du temps global,
– **aucun n'est empêché de tourner** (par exemple par une coalition de processus prioritaires)
– **minimiser le temps de réponse** : les utilisateurs interactifs souhaitent un système "réactif". Quand ils lancent des commandes courtes, la réponse doit être rapide.
– **minimiser le temps d'exécution** : les commandes longues ne s'éternisent pas.
– **maximiser le rendement** : on peut lancer davantage de travaux dans la journée.

Mais malheureusement
– ces objectifs sont contradictoires,
– le comportement des processus ne peut pas être prévu
.

Il n'y a donc **pas de politique optimale** valable dans tous les cas. On se contente d'**heuristiques**, dont l'expérience montre qu'elles fonctionnent bien (ou pas) dans des contextes voisins.

Qui plus est, pour chaque méthode il sera souvent possible de construire un "scénario pathologique" pour lequel les choses se passent mal, du point de vue d'un critère particulier. La difficulté est d'évaluer la probabilité avec laquelle de tels cas peuvent se produire, dans le contexte d'utilisation visé, ainsi que les conséquences.

Que des programmes d'affichage puissent "lagger" quelques secondes de temps en temps n'a pas la même importance pour une animation en flash dans un navigateur, et sur une console d'aiguilleur du ciel.

4.3 Tourniquet

L'algorithme du **tourniquet** (synonymes : *round robbin*, FIFO, premier arrivé-premier servi, ...)

Principe :
– le système détient une liste des processus prêts ;
– on choisit, pour l'activer, le premier processus de la liste ;
– à la fin de son quantum de temps, un processus actif est placé en fin de liste

Propriétés : cette heuristique garantit une **équité** entre les processus, qui ont tous une occasion de tourner.

Exercice 4. Soit trois processus A, B et C à comportements périodiques : ils font du calcul, une opération d'E/S et recommencent (un très grand nombre de fois).
– Pour A le calcul dure 10 ms, 10 ms pour B, et 45 ms pour C
– une opération d'E/S de 20 ms

Etudiez le déroulement pendant les 150 premières ms
– pour un ordonnancement circulaire (tourniquet) sans réquisition
– pour un ordonnancement avec réquisition (quantum 20 ms)

Comparez les taux d'utilisation des CPU dans les deux cas.

4.4 Priorités

Les priorités permettent de favoriser certains travaux.

On affecte un niveau de priorité (numérique) à chaque processus. Sous Unix, ce sont les numéros faibles qui sont les plus prioritaires.

Principe

- on choisit le processus de priorité la plus élevée
- si il y a plusieurs processus du même niveau, ils sont sélectionnés à tour de rôle (tourniquet)

Propriétés

- il y a un risque de **coalition** : si il y a beaucoup de processus prioritaires qui font du calcul, ils occupent le temps du processus à eux seuls, empêchant les autres processus de tourner.

Exercice 5. Soient deux processus A, B au fonctionnement cyclique : ils font du calcul (durées d_a, d_b), une entrée-sortie (durée d_{es}), et recommencent. Un processus C, moins prioritaire, ne fait que du calcul.
À quelle condition apparaît-il une coalition entre A et B qui empêche C de s'exécuter ?

Exercice 6. Soient trois processus A, B et C, qui ont un comportement répétitif : ils font un peu de calcul pour une durée t (respectivement 10ms, 15ms et 45ms), puis une opération d'entrée-sortie (qui dure 20ms), et recommencent. Le processeur d'entrées-sorties traite les requêtes séquentiellement, dans l'ordre où il les reçoit.

1. En supposant un ordonnancement par tourniquet sans réquisition, évaluez le taux d'occupation de la CPU, et du processeur d'entrées-sorties. Étudiez l'équité de l'ordonnancement.
2. Même question, avec un ordonnancement préemptif avec tourniquet, et un quantum fixé à 20 ms.
3. Même question, avec un ordonnancement préemptif avec priorités (dans l'ordre décroissant B, A, C), et un quantum fixé à 20 ms.

4.5 Priorités variables

En pratique on utilise des systèmes avec des priorités variables.

Exemple d'un tel système :

- chaque processus se voit accorder une priorité initiale (qui peut dépendre de l'utilisateur)
- la priorité baisse chaque fois que le processus termine son quantum de temps
- elle revient à son niveau initial après chaque entrée-sortie.

Ainsi

- les processus qui font beaucoup de calcul sont pénalisés, leur priorité baisse.
- les processus courts sont favorisés, ce qui donne aux utilisateurs interactifs une impression de réactivité.
- de même les processus qui font beaucoup d'E/S (et donc chargent peu le processeur), ont davantage d'occasions de tourner.

Exercice 7. Au département informatique nous avons eu un problème avec un système d'exploitation (SysVr4) qui remontait les priorités les processus qui faisaient des entrées-sorties.
Les étudiants devaient écrire du code pour afficher les requêtes d'une base de données, avec un algorithme du style

```
requete <=  "select * from voitures"
curseur <=  lancer-requete(req)

tant que code-erreur() == 100
  faire
        afficher valeur(curseur)
        avancer(curseur)
```

Dans ce code, les opérations sur la base de données communiquent par l'intermédiaire d'un "*pipe*" avec un processus qui accède effectivement aux fichiers de la base.
Certains étudiants ont inversé le test d'arrêt. Leur programme entrait donc dans une boucle de conversation avec l'autre processus, qui répondait immédiatement avec un code d'erreur différent de 100.
Conséquences pour les autres utilisateurs ?

4.6 Files multiples

On définit des **classes** de processus

- à chaque classe correspond une liste (FIFO) de processus
- chaque classe est sélectionnée régulièrement

On décidera par exemple d'avoir 3 classes : A (prioritaire), B (normal), C (non prioritaire), à qui on accordera respectivement 50 %, 30 % et 20 % du temps de calcul. Chaque classe sera gérée selon le principe du tourniquet,

Par exemple, pour les 5 premières activations l'ordonnanceur choisira un processus prêt de la classe A, pour les 3 suivantes dans la classe B, etc, et recommencera.

Ce système a l'avantage

- de respecter les priorités,
- d'interdire les coalitions.

Dans l'exploitation en traitement par lots, les classes permettaient une segmentation de la clientèle des centres de calculs : les tarifs étaient plus élevés pour les travaux dans la classe la plus prioritaires. Les clients "en classe économique" devaient attendre les résultats plus longtemps.