8INF342
Systèmes d'exploitation
Séance 3 – Ordonnancement

Sara Séguin, ing., Ph.D.
Professeure au département d'informatique et de mathématique
sara.seguin@uqac.ca

UQAC

DERNIER COURS

libre de voir plus loin

- Processus
- Gestions des processus
- Modèles de processus
- Linux fork()

libre
de voir plus loin

UTILITÉ DE FORK

UTILITÉ DE FORK

libre de voir plus loin

La base de linux en termes de processus est le clonage par fork() et par fork()+exec():

- Exécuter une commande dans un terminal sans tuer le shell.
- Lancer un programme en parallèle.
- Pipelines.
- Isoler des processus (onglets navigateurs web, différents services).
- Gestion des clients sur un serveur.
- Dameons en arrière-plan.

UQAC

UTILITÉ DE FORK

libre de voir plus loin

- Si un programme doit lancer un autre programme, il est alors nécessaire d'utiliser fork.
- Sans fork, on est limité à un seul processus.
- L'enfant est une copie du parent. Identique mais nouveau PID.
- Labo 1. Si vous n'utilisez pas fork, alors comment conserver le terminal?
- Copy-on-write. L'enfant est copié. Dès qu'il tente de modifier la mémoire, alors il copie les pages en mémoire pour ne pas modifier les pages du parent.
  - Plus rapide (car ne copie pas la mémoire si pas nécessaire).
  - Chaque processus a son propre espace mémoire.
- Plusieurs mécanismes permettent au parent et à l'enfant de communiquer: tuyaux, messages, etc. Nous verrons cela lors des cours sur la synchronisation.

libre de voir plus loin

EXERCICES

Tirés du livre: Silberschatz, A., Galvin P.B., Gagne G., Operating systems concepts, ninth edition, Wiley, 2012,

EXERCICE #1

libre de voir plus loin

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>


int value = 5;

int main()
{
        pid_t pid;
        pid = fork();

        if(pid==0){
                value +=15;
                return 0;
        }
        else if(pid >0){
        wait(NULL);
        printf("PARENT: value = %d",value);
        return 0;
        }
 }
```

EXERCICE #2

libre de voir plus loin

```
#include <unistd.h>
#include <stdio.h>
#include <sys/types.h>


int main()
{
        int i;
        for(i=0;i<4;i++){
                fork();
        }
        return 0;
 }
```

EXERCICE #3

libre de voir plus loin

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid, pid1;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       pid1 = getpid();
       printf("child: pid = %d",pid); /* A */
       printf("child: pid1 = %d",pid1); /* B */
    }
    else { /* parent process */
       pid1 = getpid();
       printf("parent: pid = %d",pid); /* C */
       printf("parent: pid1 = %d",pid1); /* D */
       wait(NULL);
    }

    return 0;
}
```

Considérez que le PID du parent est 2600 et le PID de l'enfant 2603

EXERCICE #4

libre
de voir plus loin

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

#define SIZE 5

int nums[SIZE] = {0,1,2,3,4};

int main()
{
int i;
pid_t pid;

   pid = fork();

   if (pid == 0) {
      for (i = 0; i < SIZE; i++) {
         nums[i] *= -i;
         printf("CHILD: %d ",nums[i]); /* LINE X */
      }
   }
   else if (pid > 0) {
      wait(NULL);
      for (i = 0; i < SIZE; i++)
         printf("PARENT: %d ",nums[i]); /* LINE Y */
   }

   return 0;
}
```

CONTENU

libre de voir plus loin

- Ordonnancement
  - Long terme
  - Moyen terme
  - Court terme
- Principes de base
- Critères d'ordonnancement
- Politiques d'ordonnancement court terme
  - First Come First Serve
  - Round Robin
  - Shortest Process Next
  - Shortest Remaining Time
- UNIX/WINDOWS
- Ordonnancement multiprocesseur et multicoeur

libre de voir plus loin

ORDONNANCEMENT

CONTENU

libre de voir plus loin

ORDONNANCEMENT:

- Principes de base
- Critères d'ordonnancement
- Politiques d'ordonnancement
- UNIX et WINDOWS
- Ordonnancement multiprocesseur et multicoeur

libre
de voir plus loin

PRINCIPES DE BASE

libre
de voir plus loin

ORDONNANCEMENT

libre de voir plus loin

- Depuis le début de la session, nous avons vu que le CPU exécute des processus.
  - Les processus sont tous en concurrence pour obtenir du temps processeur.

- L'ordonnancement consiste à déterminer dans quel ordre sont assignés les processus au CPU.

- Dans plusieurs systèmes d'exploitation, l'ordonnancement est séparé en quatre niveaux:
  - Long-terme
  - Moyen-terme
  - Court-terme
  - Entrées/Sorties (plus tard dans la session, pas abordé dans cette présentation)

ORDONNANCEMENT

libre
de voir plus loin

ORDONNANCEMENT

libre de voir plus loin

À quoi sert chaque niveau?

- **Long-terme**: Ajouter un processus à l'ensemble des processus à être exécutés.
- **Moyen-terme**: Swapping (déplacement de processus ou de portions de processus du disque dur à la mémoire vive ou l'inverse).
- **Court-terme** : Choisir le processus à exécuter.

DISPATCHER

libre de voir plus loin

Le dispatcher spécifie au CPU quel processus exécuter. L'ordonnanceur court-terme détermine le processus à exécuter alors que le dispatcher est un module qui donne le contrôle de ce processus au CPU.

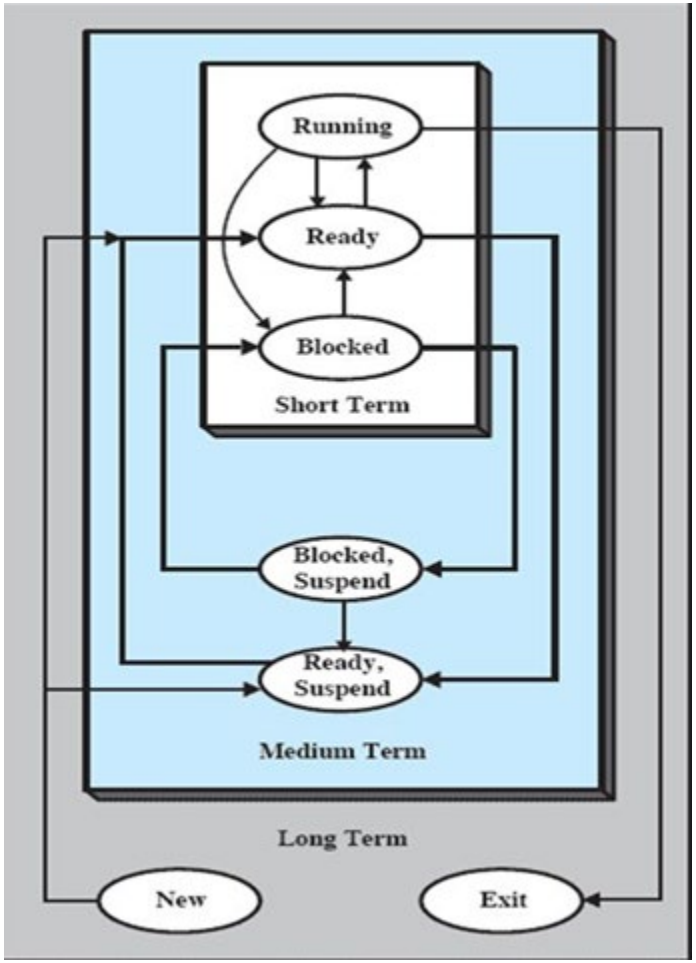
Le processeur doit être utilisé de façon efficace. À chaque fois qu'un nouveau processus est exécuté (process switch):
-> Changement de contexte.
-> Changement au mode utilisateur.
-> Retourner à l'endroit requis dans le programme.

Si l'ordonnanceur cause trop de changements de processus, ça devient coûteux en temps d'utilisation CPU.

Ordonnanceur VS dispatcher?

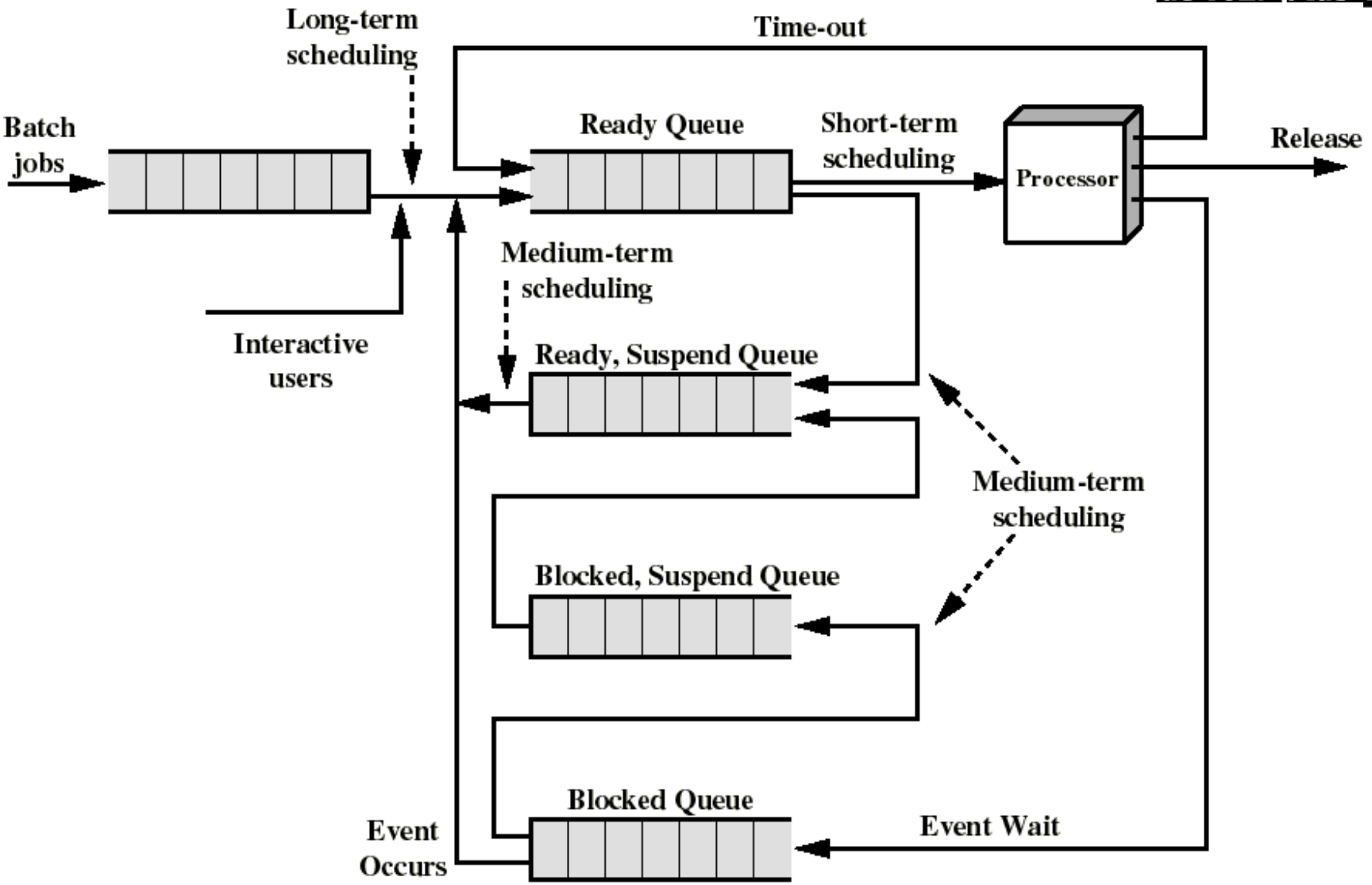
ORDONNANCEMENT

libre de voir plus loin



ORDONNANCEMENT

libre
de voir plus loin



LONG-TERME

libre de voir plus loin

Détermine les programmes qui sont admis dans le système pour une exécution ultérieure.

Contrôle le degré de multiprogrammation.

Lorsqu'un programme est admis, il est transformé en processus et ajouté à la file de processus pour l'ordonnanceur court-terme.

Deux décisions sont prises:

1. L'ordonnanceur détermine quand le SE peut gérer un job supplémentaire.
2. L'ordonnanceur doit choisir les jobs à ajouter et les transformer en processus.

LONG-TERME

libre de voir plus loin

1. L'ordonnanceur détermine quand le SE peut gérer un job supplémentaire.

Cette décision dépend du degré de multiprogrammation désiré.

Plus il y a de processus, plus le % de temps d'exécution est diminué. (Il ne faut pas oublier que les processus sont en compétition face au CPU)

L'ordonnanceur peut limiter le nombre de processus afin d'obtenir un % satisfaisant.

Autre situation: si le temps idle du processeur dépasse un certain treshold, l'ordonnanceur peut être invoqué.

LONG-TERME

libre de voir plus loin

2. L'ordonnanceur doit choisir les jobs à ajouter et les transformer en processus.

Soit géré par un outil pour maximiser la performance, ou être aussi simple que FCFS (first-come first-served).

Si un outil est utilisé, le critère de performance peut inclure la priorité, le temps d'exécution prévu et les requêtes E/S.

Moyen-terme

libre de voir plus loin

- Fait partie de la fonction swapping (changement de processus).
- Basé sur les besoins concernant la gestion de la mémoire principale.

COURT-TERME

libre de voir plus loin

- Cet ordonnanceur est celui qui s'exécute le plus souvent.
- Choisit le processus à exécuter.
- Invoqué quand un processus en cours d'exécution pourrait devenir bloqué.
  - Interruption d'horloge (time-out)
  - Interruption E/S
  - Appel système
  - Signaux

COURT-TERME

libre de voir plus loin

- L'objectif principal du répartiteur est de gérer le temps du CPU pour optimiser un ou plusieurs **aspects** du système.

- **Critères** de l'ordonnancement court-terme:

  1. Orientés vers l'utilisateur

  2. Orientés vers le système

libre de voir plus loin

CRITÈRES D'ORDONNANCEMENT

1. ORIENTÉS UTILISATEUR

libre de voir plus loin

- Turnaround time.
  Intervalle de temps entre le démarrage et la complétion d'un processus.

- Temps de réponse.
  Pour un processus interactif, temps entre une requête et le début de la réponse. Le répartiteur devrait tenter d'avoir un temps de réponse court et de maximiser le nombre d'utilisateurs qui obtiennent un temps de réponse acceptable.

- Dernière limite (deadline).
  Lorsqu'une limite peut être spécifiée, le répartiteur peut tenter de maximiser le nombre de deadlines respectées.

- Prévisibilité.
  Un processus devrait prendre environ le même temps, au même coût, peu importe la charge sur le système.

2. ORIENTÉS SYSTÈME

libre de voir plus loin

- Débit (throughput).
  Maximiser le nombre de processus complétés par unité de temps.

- Taux d'utilisation du processeur.
  % de temps que le CPU est occupé.

- Équité.
  Les processus devraient être traités équitablement, et ne devraient jamais souffrir de **famine**!

- Priorité.
  Le CPU devrait favoriser les processus avec une priorité plus haute.

- Balancement des ressources.
  Les ressources devraient être sollicitées. Les processus qui ne nécessitent pas de ressources devraient être favorisés.

PRIORITÉS

libre de voir plus loin

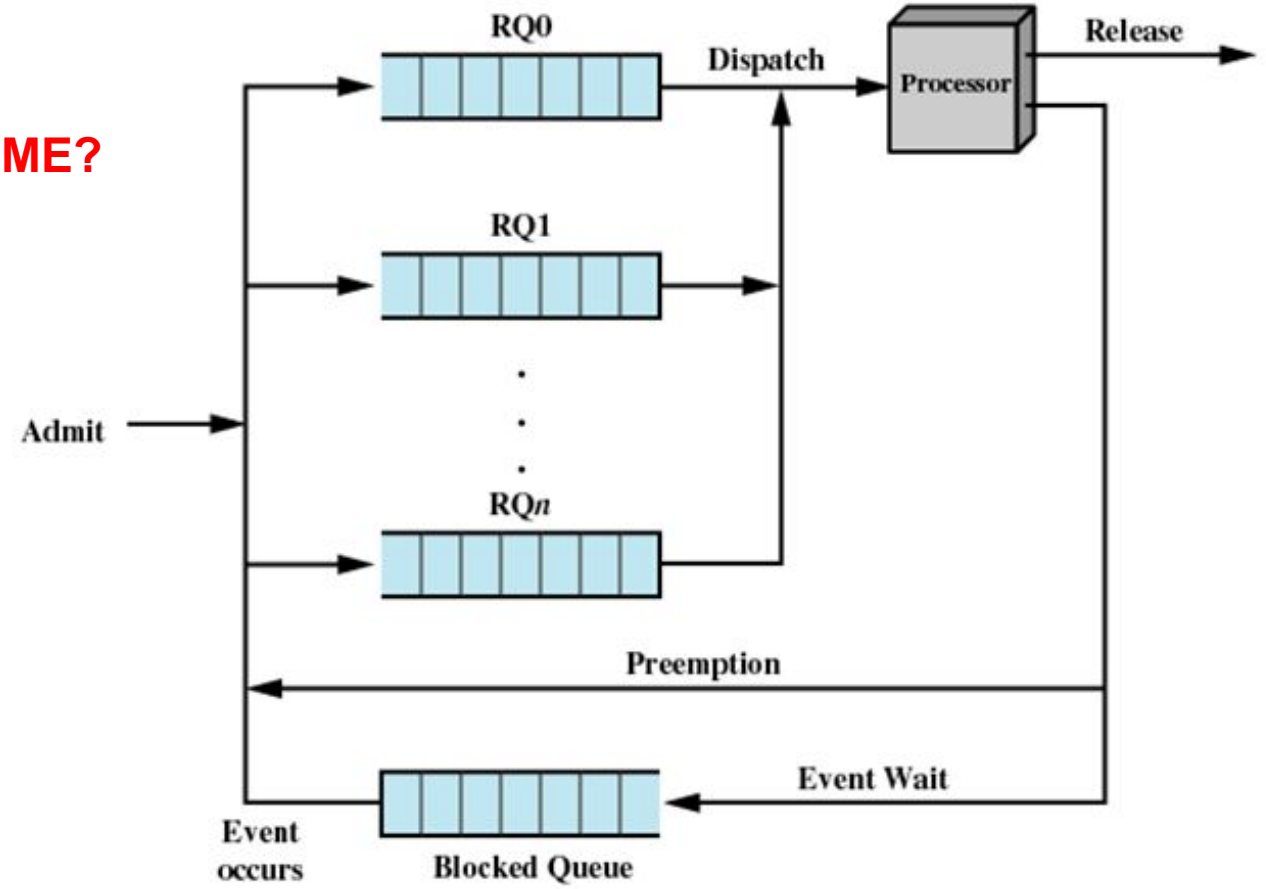
- Chaque processus se voit attribuer une priorité et le répartiteur devrait toujours choisir un processus ayant une priorité élevée.

- Des files de processus prêts sont créées par priorité. Lorsque le répartiteur choisit un processus dans une file, il utilise un algorithme d'ordonnancement. S'il n'y a aucun processus dans cette file, alors il passe à la prochaine file de priorités.

PRIORITÉS

libre de voir plus loin

PROBLÈME?



PRIORITÉS

libre de voir plus loin

- Problème?

Les processus de basse priorité pourraient souffrir de **famine**. S'il y a toujours des processus de plus haute priorité alors ils ne seraient jamais servis.

Une façon de contourner ce problème est de changer la priorité des processus selon son âge et son historique d'exécution.

libre de voir plus loin

POLITIQUES D'ORDONNACEMENT (COURT-TERME)

POLITIQUES D'ORDONNANCEMENT

libre de voir plus loin

Deux modes de décision permettent de déterminer le moment auquel la fonction de sélection (choix du processus à exécuter) sera exécutée:

1. **Non-préemptif.** Lorsqu'un processus est en cours d'exécution, il s'arrête soit:
   i) Lorsqu'il est terminé.
   ii) Lorsqu'il est bloqué par une E/S ou attend un service du SE.

2. **Préemptif**. Le SE peut interrompre le processus en cours d'exécution à l'état prêt. Peut survenir lorsqu'un nouveau processus est créé, lorsqu'une interruption survient et place un processus à l'état bloqué, prêt ou périodiquement.

   Ce mode empêche les processus de monopoliser le processeur.

POLITIQUES D'ORDONNANCEMENT

libre de voir plus loin

Plusieurs politiques d'ordonnancement sont expliquées ci-après. La fonction de sélection détermine le prochain processus à exécuter (à mettre à l'état prêt).

Politiques d'ordonnancement:

1. FCFS (first-come first-serve)
2. Round-robin
3. SPN (shortest process next)
4. SRT (shortest remaining time)
5. Feedback

EXEMPLE POUR L'ÉTUDE DES POLITIQUES

libre de voir plus loin

Processus	Temps arrivée	Temps service
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

Temps de service (T_s): temps d'exécution total.

Critères de comparaison:

- Temps de complétion
- Turnaround time (T_r.) Temps total dans le système (attente comprise).
- Ratio T_r/ T_s = Temps de délais relatif. Valeur min = 1.
  - Plus le ratio est élevée, moins la qualité de service est bonne.

1. FCFS

libre de voir plus loin

- First-come first-serve (FIFO)
- Lorsqu'un processus devient prêt, il est placé dans la file.
- Lorsque le processus en cours d'exécution arrête son exécution, le processus en file depuis le plus long moment est sélectionné.

Fonction de sélection = max(w),
où w est le temps total en attente dans le système.

- Non-préemptif.
- Famine impossible.

1. FCFS

libre de voir plus loin

On distingue plusieurs « catégories » d'exécution des processus:

1. **Processor bound**: Processus qui nécessite beaucoup de temps CPU. Par exemple, beaucoup de calculs. Limité par la vitesse du CPU.

2. **I/O bound**: Processus qui nécessite beaucoup d'entrées/sorties. Par exemple, un processus qui doit lire un fichier sur un disque. Limité par la vitesse des I/O.

1. FCFS

libre de voir plus loin

Supposons que plusieurs processus (type A) sont processor bound et que quelques-uns (type B) sont I/O bound.

Avec la politique FCFS, les processus A sont exécutés tant qu'ils ne sont pas terminés, donc les processus B sont en attente.

Un des processus B obtient son exécution, mais il devient bloqué par une demande d'E/S. Un des processus A est alors exécuté, mais devient bloqué par un événement. Que se produit-il?

Le CPU devient idle. FCFS est inefficace du point de vue du processeur unique et de l'utilisation des E/S.

2. ROUND ROBIN

libre de voir plus loin

- Mode de décision préemptif.
- Une interruption est générée à intervalles réguliers. Le processus en cours d'exécution est placé dans la file « prêt », puis le prochain processus est sélectionné comme FCFS.
- Aussi nommé « time slicing ». Le processus se voit allouer une tranche de temps (quantum temporel).

2. ROUND ROBIN (CHOIX DU QUANTUM)

libre de voir plus loin

Le quantum temporel :

• Ne doit pas être trop court. Sinon le processeur passera son temps à gérer la répartition plutôt qu'effectuer les tâches.

• Doit être légèrement supérieur au temps typique d'interaction avec le processus.

- Que se produit-il si le quantum est plus long que le processus avec le plus long temps de service?
  - Dégénère en FCFS.

2. ROUND ROBIN

libre de voir plus loin

Supposons que plusieurs processus (type A) sont processor bound et que quelques-uns (type B) sont I/O bound.

Un processus B est exécuté, puis bloqué pour une E/S. Lorsque l'événement survient, le processus B est déplacé dans la file prêt.

Un processus A est exécuté pour un quantum donné, puis replacé immédiatement dans la file prêt.

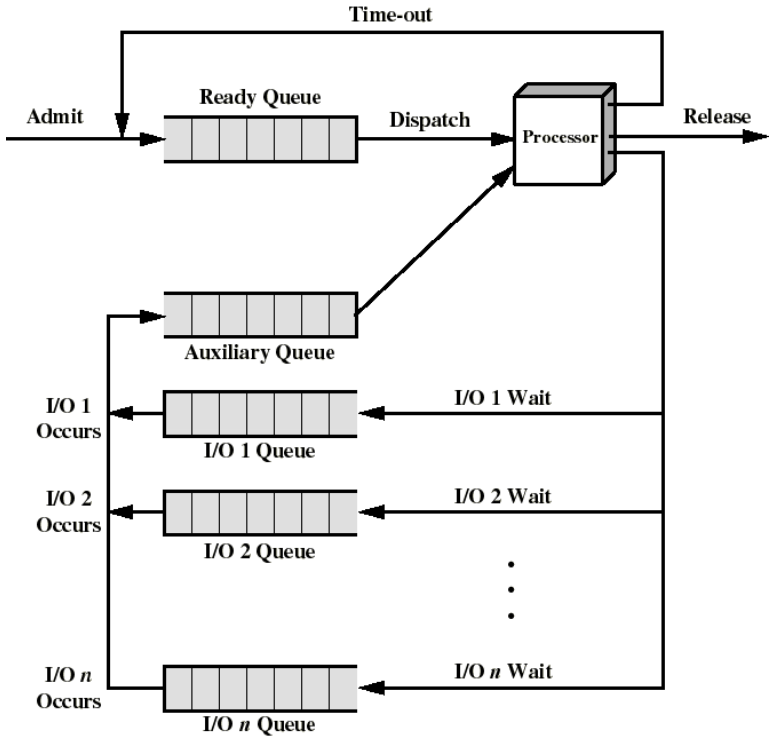
Les processus A ont donc plus d'accès au CPU, ce qui n'est pas équitable.

2. ROUND ROBIN VIRTUEL

Pour remédier à ce problème, l'algorithme round robin virtuel a été développé.

Une file auxiliaire contient les processus qui sont à nouveau prêt, suite à un blocage causé par une E/S.

Lorsque le répartiteur choisit le prochain processus, la file auxiliaire est prioritaire sur la file prêt.



3. SHORTEST PROCESS NEXT

libre de voir plus loin

- Mode de décision non-préemptif.
- Fonction de sélection: le processus avec le temps de traitement prévu le plus court. Min $T_{s.}$
- La performance est améliorée, mais la variabilité dans le temps de réponse augmente. Les processus orientés vers les E/S (processus B) sont choisis en premier.

3. SHORTEST PROCESS NEXT

libre
de voir plus loin

- Il est nécessaire d'évaluer le temps d'exécution de chaque processus.
- Calcul simple: moyenne des temps d'exécution des instances de processus.
- Soit s_n le temps prévu pour la n^e instance et T_n le temps processeur.
- $s_{n+1} = \frac{1}{n}\sum_{i=1}^{n} T_i$
- Récursivement:

$$s_{n+1} = \frac{1}{n} T_n + \frac{n-1}{n} S_n$$

3. SHORTEST PROCESS NEXT

libre
de voir plus loin

- Afin de donner plus de poids aux exécutions récentes, on utilise la moyenne exponentielle:

$$S_{n+1} = \alpha\, T_n + (1 - \alpha)\, S_n\,,$$

Où $0 < \alpha < 1$.

- La moyenne exponentielle permet de capturer les changements de comportement des processus.

- Risque avec SPN: FAMINE pour les processus longs, s'il y a un apport constant de processus courts.

4. SHORTEST REMAINING TIME

libre de voir plus loin

- Mode de décision préemptif.
- Même principe que pour SPN, sauf que le répartiteur choisit le processus qui a le temps espéré d'exécution restant le plus court.
- Cet algorithme ne favorise pas les longs processus comme FCFS. Le temps de turnaround est meilleur qu'avec SPN car les processus courts sont assignés avant les processus longs.

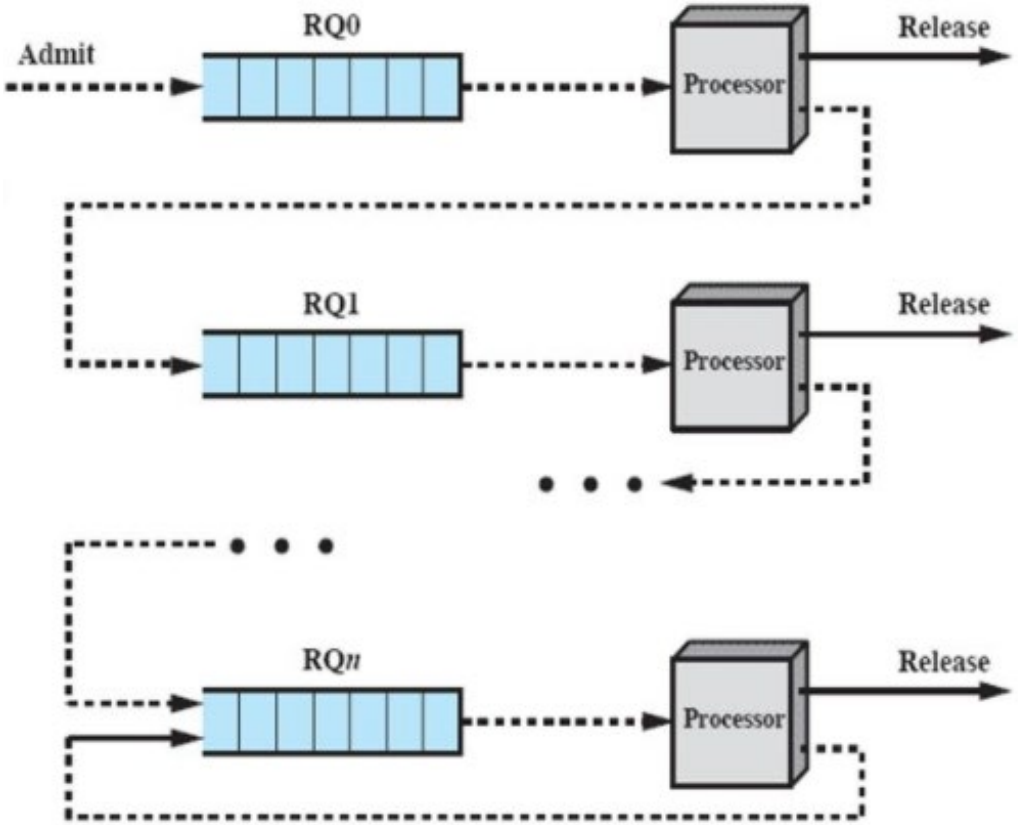
5. FEEDBACK

libre de voir plus loin

- Contrairement à SPN ou SRT, peut être utilisé si on ne connaît pas les temps d'exécution.
- Mode de décision préemptif basé sur les priorités dynamiques.
- Lorsqu'un processus arrive dans le système, il est placé dans une file.
- Suite à son exécution, il retourne à l'état prêt mais dans une file de priorité plus basse.
- À chaque exécution, le processus est déplacé vers une file moins prioritaire.
- Chaque file utilise un mécanisme FCFS pour choisir le prochain processus.
- Aussi connu sous le nom de « multilevel feedback ».

5. FEEDBACK

libre
de voir plus loin



5. FEEDBACK

libre de voir plus loin

- Risque de famine pour les processus longs.
- Une solution simple consiste à faire passer un processus dans une file à priorité plus élevée après un certain temps.

RÉSUMÉ DES POLITIQUES D'ORDONNANCEMENT

libre de voir plus loin

	FCFS	Round Robin	SPN	SRT	Feedback
Fonction de sélection	max(w)	constante	min(s)	min(s-e)	---
Mode de décision	non-préemptif	préemptif	non-préemptif	préemptif	préemptif
Effet sur processus	pénalise processus courts	égalitaire	pénalise processus longs	pénalise processus longs	favorise processus avec beaucoup E/S
Famine	non	non	possible	possible	possible

s = temps de service total (inclut e)
e = temps d'exécution jusqu'à présent
w = temps dans le système en attente

FAIR-SHARE SCHEDULING

libre de voir plus loin

Évidemment, il existe une panoplie d'algorithmes permettant l'ordonnancement selon différents critères.

Dans la plupart des systèmes d'exploitation, l'ordonnancement est basé sur un mélange de plusieurs algorithmes vus précédemment.

Le fair-share scheduling consiste à assigner un temps CPU équitable à tous les utilisateurs, dans un système multi-user. Globalement, les processus sont divisés par utilisateur, puis un ordonnancement est réalisé afin de donner accès au CPU équitablement.

libre de voir plus loin

UNIX ET WINDOWS

UNIX

libre de voir plus loin

- Utilise le multilevel feedback et un round robin dans chaque file de priorité.
- Un processus est préempté après 1 seconde.
- Le calcul de la priorité est basé sur le type de processus et son historique d'exécution:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2} \qquad P_j(i) = Base_j + \frac{CPU_j(i)}{2} + nice_j$$

Où

$CPU_j(i) =$ utilisation du processeur par le processus j dans l'intervalle i

$P_j(i) =$ priorité du processus j au début de l'intervalle i (priorité élevée pour valeurs faibles)

$Base_j =$ priorité de base du processus j

$nice_j =$ facteur d'ajustement utilisateur

La priorité est recalculée à chaque seconde.

Le calcul des priorités permet de diviser les processus par groupe priorité.

Les groupes de priorité sont (en ordre décroissant):

- Swapper
- Blocs E/S (disques)
- Gestionnaire de fichiers
- E/S caractères (ports)
- Processus utilisateur

Les groupes de priorité permettent d'optimiser l'accès aux E/S et de répondre rapidement aux appels système.

WINDOWS

libre de voir plus loin

- L'ordonnancement est basée sur un round-robin par priorités.
    - Pour chaque niveau, l'algorithme « À tour de rôle » est appliqué.
    - Un thread avec priorité basse peut être interrompu par un thread de priorité supérieure.
    - Certain niveaux sont dynamiques. La priorité d'un processus/thread peut changer durant son exécution.

WINDOWS

libre de voir plus loin

- Il y a deux classes de priorités: temps réel et dynamique.
  - Temps réel **(16-31)**
    - Les priorités temps réel correspondent à des processus prioritaires pour le système.
    - Les threads ayant la priorité temps réel ont préséance sur les autres.
    - La priorité d'un thread temps réel ne change jamais.

- Dynamique **(0-15)**

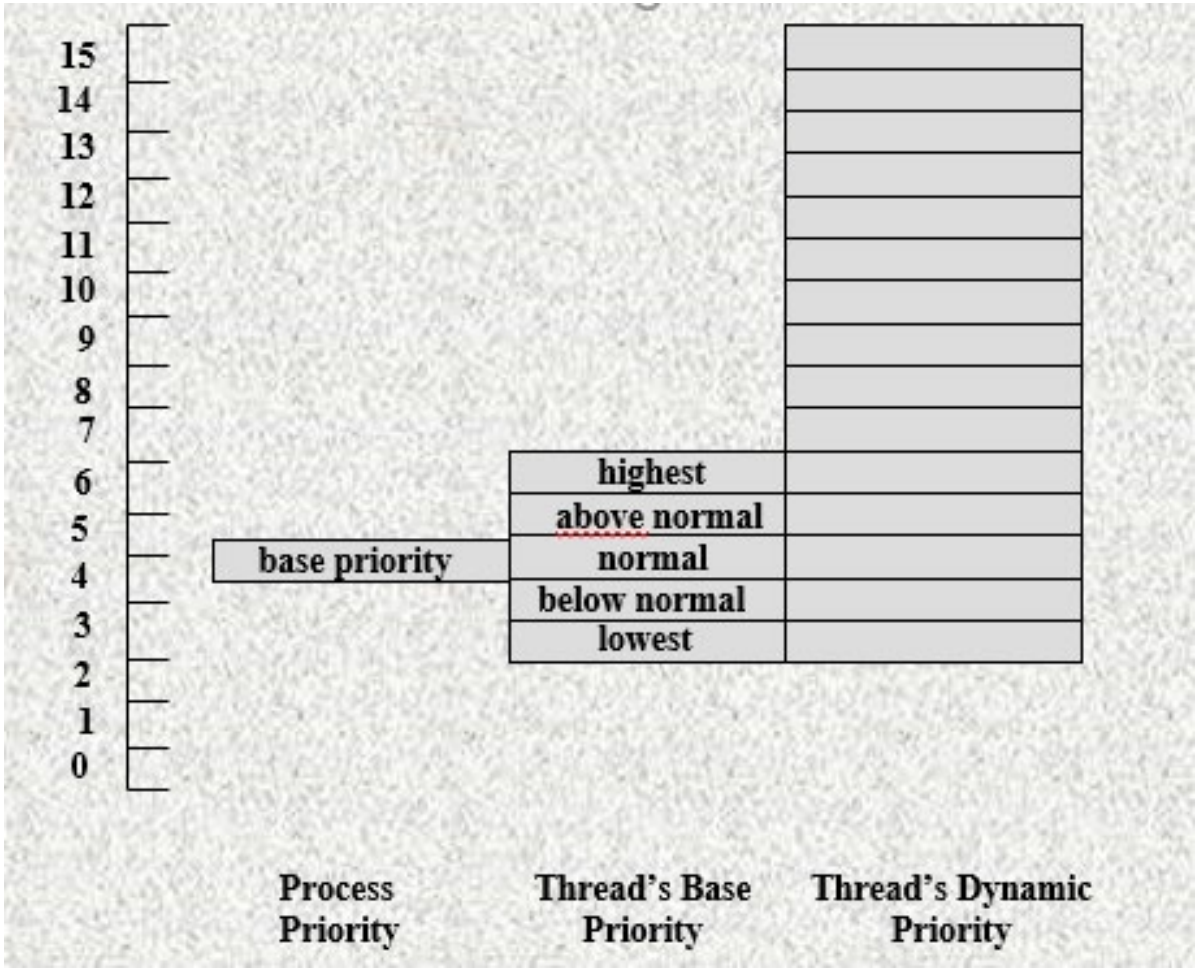
  • Un thread commence à un niveau de base et peut changer de priorité au cours de son exécution. Le SE change alors le thread de file.

  - La priorité du thread peut être ajustée selon divers critères:
    - Si un thread est interrompu parce que sont quantum de temps est terminé , sa priorité diminue.
    - Si un thread est interrompu pour des opérations d'E/S, sa priorité augmente.

  • Les threads orienté calcul ont tendance à avoir une priorité plus petite que ceux orienté vers les E/S.

WINDOWS

libre
de voir plus loin



WINDOWS

libre de voir plus loin

Je vous invite à lire cette page, qui contient plus de détails!

https://docs.microsoft.com/en-us/windows/win32/procthread/scheduling-priorities

libre de voir plus loin

ORDONNANCEMENT MULTIPROCESSEUR ET MULTICOEUR

ORDONNANCEMEMT MULTIPROCESSEUR

libre
de voir plus loin

Multiprocesseur symétrique:

ORDONNANCEMEMT MULTIPROCESSEUR

libre de voir plus loin

Trois considérations:

1. Assigner des processus aux processeurs
2. Multiprogrammation sur chaque processeur
3. Choix des processus à exécuter (ordonnancement)

1. ASSIGNATION PROCESSUS/PROCESSEURS

libre de voir plus loin

L'ordonnancement le plus simple consiste à considérer tous les CPU ensemble et simplement d'assigner des processus aux processeurs sur demande.

- Assignation statique: un processus est assigné au même processeur de sa création à sa terminaison (une file existe pour chaque CPU).
  - Avantage: répartition moins coûteuse car l'assignation est déterminée
  - Désavantage: un CPU peut être en attente alors qu'un autre CPU est débordé.
    - Solution?
      Une file pour tous les CPU. Le processus peut être exécuté par différents CPU pendant son existence.

1. ASSIGNATION PROCESSUS/PROCESSEURS

libre de voir plus loin

Deux méthodes peuvent être utilisées pour l'assignation:

1. Maître/Esclave

Le kernel est toujours exécuté sur le même CPU. Les autres CPU sont des esclaves et peuvent seulement exécuter des programmes utilisateurs.

Le maître est en charge de l'ordonnancement.

Si un esclave nécessite une E/S, il doit faire une requête au maître qui se chargera de l'ordonnancement.

Avantage: simple à implémenter.

Désavantage: Si le maître ne répond plus, alors tout le système est arrêté.

1. ASSIGNATION PROCESSUS/PROCESSEURS

libre de voir plus loin

2. Peer

Le kernel peut être exécuté sur tous les CPU et chaque processeur fait son propre ordonnancement.

Le SE doit s'assurer que 2 CPU ne choisissent pas le même processus et aussi veiller à ne pas perdre les processus (files).

Deux façons simples de s'y prendre:

1. Séparer les processus basé sur la priorité.
2. Séparer les processus basé sur l'historique d'exécution.

On cherche à maximiser la performance, en moyenne, pour toutes les applications. Selon l'application, il se pourrait que la performance moyenne ne nécessite plus l'utilisation maximale de tous les CPU, par exemple. Une application multithread peut nécessiter que tous ses threads soient exécutés en même temps pour bien performer…

3. Ordonnancement

Choix des processus à exécuter.

Avec un système à un CPU, l'utilisation des priorités et d'algorithmes d'ordonnancement est utilisé.

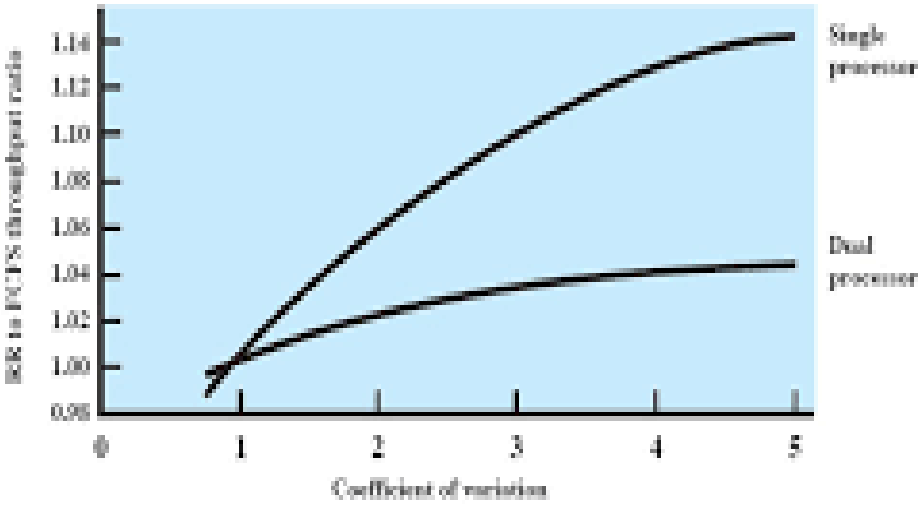
Avec plusieurs CPU, la répartition des processus et des threads est plus complexe.

3. Ordonnancement

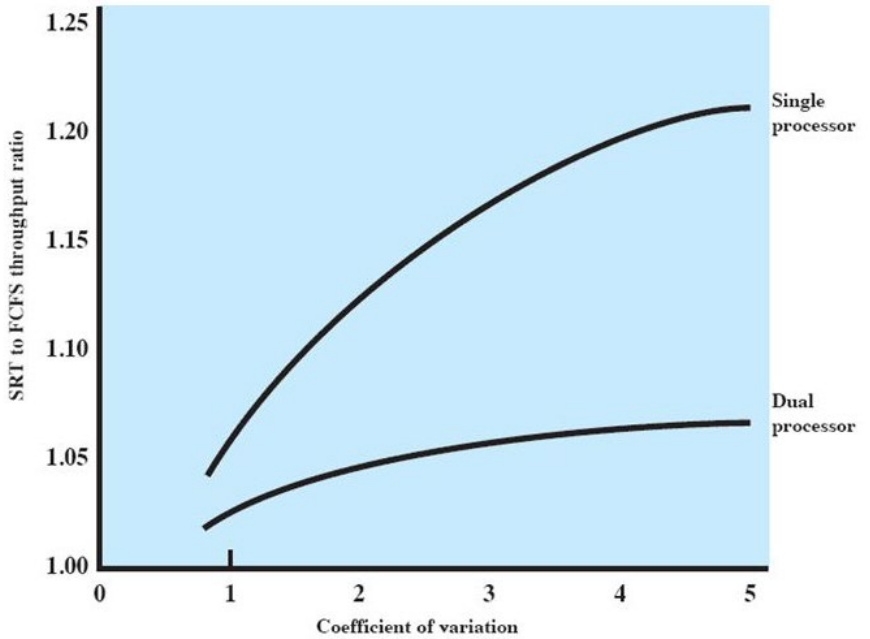
libre de voir plus loin

- Ordonnancement des processus

Le choix de l'algorithme d'ordonnancement court-terme influence peu la performance dans un système multiprocesseur.



(a) Comparison of RR and FCFS



(b) Comparison of SRT and FCFS

3. Ordonnancement

libre de voir plus loin

- Ordonnancement des threads
  1. Partage de la charge (load sharing)
  2. Ordonnancement de groupe (gang scheduling)
  3. Assignation dédiée (dedicated processor assignment)

3. Ordonnancement

libre de voir plus loin

1. Partage de la charge (load sharing)

Les processus ne sont pas assignés à un CPU en particulier. Une file globale existe pour tous les processus. Lorsqu'un CPU devient idle, il sélectionne un processus dans la file.

Avantages:

- La charge est distribuée équitablement et les CPU ne sont jamais idle s'il y a des processus dans la file.
- Pas nécessaire d'avoir un répartiteur central: chaque CPU est responsable de son ordonnancement.

Désavantages:

- L'exclusion mutuelle doit être instaurée pour la file…peut devenir problématique.
- Les threads sont exécutés sur plusieurs CPU, l'utilité des cache devient moins évidente.
- Si un programme nécessite que tous ses threads soient exécutés en même temps…peut aussi devenir problématique.

Cette méthode est la plus utilisée.

3. Ordonnancement

libre de voir plus loin

2. Ordonnancement de groupe (gang scheduling)

Consiste à assigner un ensemble de processus sur un ensemble de processeurs simultanément.

Avantages:

- Si un ensemble de processus sont reliés, permet d'améliorer la performance car on diminue la synchronisation nécessaire.
- Une seule décision de répartition affecte un ensemble de processus et de processeurs simultanément.

Utilisé lorsque des applications nécessitent que tous leurs threads soient exécutés en même temps.

Diminue le nombre de changement de processus.

3. Ordonnancement

libre de voir plus loin

3. Assignation dédiée

Assigner un groupe de processeurs à une application pour toute sa durée.

Le processeur est entièrement dédié à chaque thread de l'application tant qu'elle n'est pas complétée.

Problème?

Si un thread attend un événement, alors le CPU est idle, il n'y a plus de multiprogrammation.

Cependant:

- Si CPU par centaines, le % d'utilisation CPU n'est plus un indicateur si important.
- Réduction du nombre de changements de processus pour la durée de vie du programme…vitesse d'exécution plus rapide.

libre de voir plus loin

LABORATOIRE