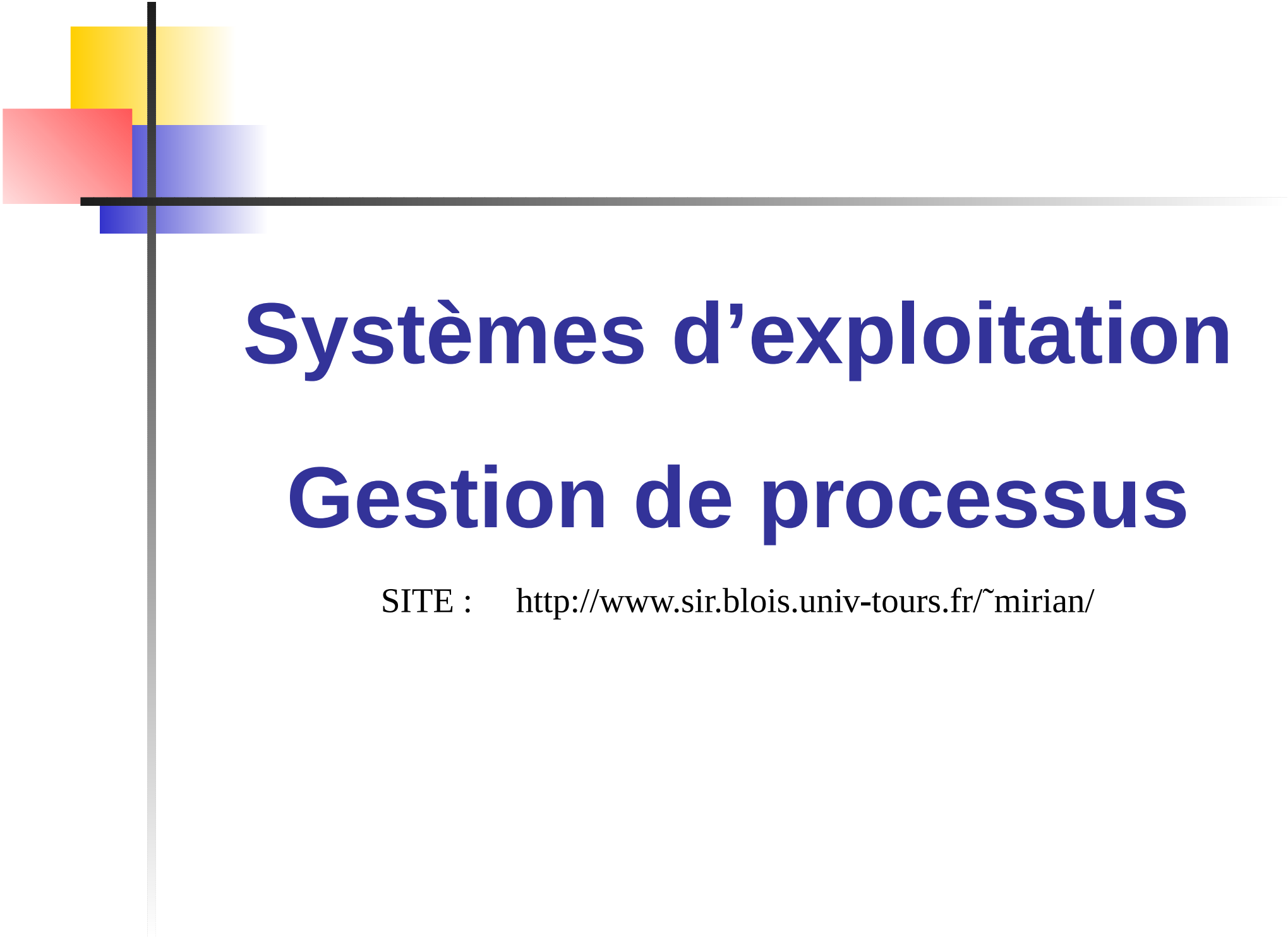


Systèmes d’exploitation

Gestion de processus

SITE : http://www.sir.blois.univ-tours.fr/˜mirian/

Les processus, à quoi ça sert?

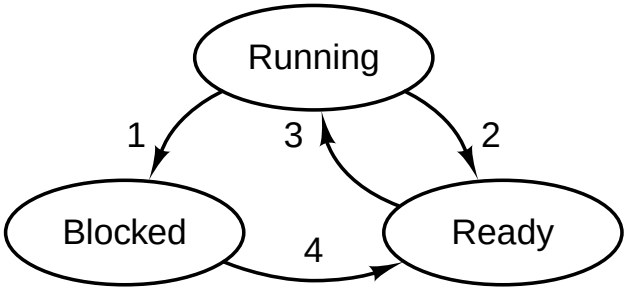
- À faire plusieurs activités ”en même temps”.
- Exemples
  - Faire travailler plusieurs utilisateurs sur la même machine. **Chaque utilisateur a l’impression d’avoir la machine à lui tout seul.**
  - Compiler tout en lisant son mail
- Problème: Un processeur ne peut exécuter **qu’une seule instruction à la fois**.
- BUT: **Partager** un (ou plusieurs) processeur entre différents processus.
- Attention!!! Ne pas confondre processus et processeur

Concept de processus

- Un processus est un programme en exécution
- L'exécution d'un processus doit progresser séquentiellement, cad, à n'importe quel moment *une seule* instruction au plus est exécutée au nom du processus
- Processus $\neq$ Programme

États des processus

- Quand un processus s'exécute, il change d'état.
- Chaque processus peut se trouver dans chacun des états suivants :
  - **En exécution:** Les instructions sont en cours d'exécution (en train d'utiliser la CPU).
  - **En attente:** Le processus attend qu'un événement se produise.
  - **Prêt:** Le processus attend d'être affecté à un processeur.
- Un **seul** processus peut être en exécution sur n'importe quel processeur à tout moment.
- Toutefois, plusieurs processus peuvent être prêts et en attente

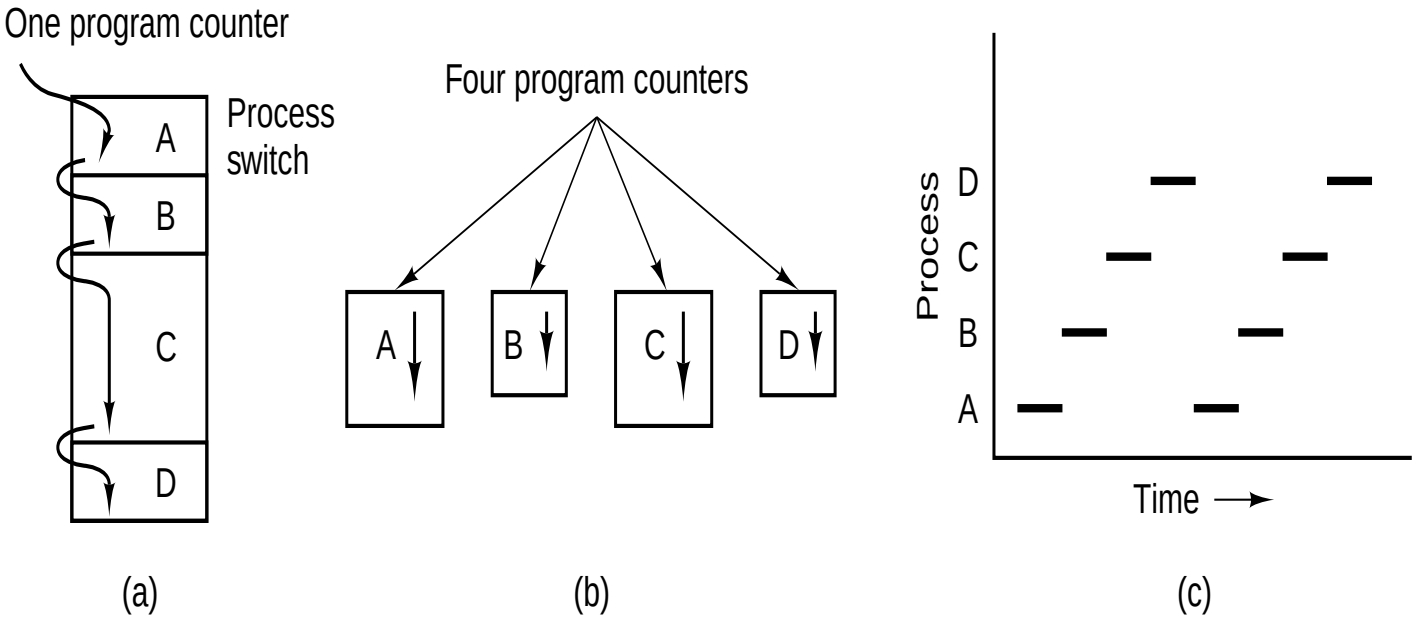


1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Processus

- Point de vue conceptuel: chaque processus possède son processeur virtuel.
- Réalité: **le processeur bascule constamment d'un processus à l'autre**.
- Ce basculement rapide est appelé **multiprogrammation**.
- Lorsque le processeur passe d'un processus à un autre, **la vitesse de traitement d'un processus** donné n'est pas uniforme et probablement **non reproductible** si le même processus s'exécute une nouvelle fois.

Processus


One program counter
A
B
C
D
Process switch
(a)
Four program counters
A
B
C
D
(b)
Process
D
C
B
A
Time
(c)

Bloc de contrôle de processus(1)

Chaque processus est représenté dans le SE par un PCB (*process control block*)

PCB	
Pointeur	État du processus
Numéro du processus	
Compteur d'instructions	
Registres	
Limite de la mémoire	
Liste des fichiers ouverts	
. . .	

Bloc de contrôle de processus(2)

- PCB: contient plusieurs informations concernant un processus spécifique, comme par exemple:
  - L'état du processus.
  - Compteur d'instructions: indique l'adresse de l'instruction suivante devant être exécutée par ce processus.
  - Informations sur le scheduling de la CPU: information concernant la priorité du processus.
  - Informations sur la gestion de la mémoire: valeurs des registres base et limite, des tables de pages ou des tables de segments.
  - Informations sur l'état des E/S: liste des périphériques E/S allouées à ce processus, une liste des fichiers ouverts, etc.

Linux/UNIX: Lister des informations sur les processus

Commande ps

```
[mirian@home2 Transp]$ ps -af
UID        PID  PPID  C STIME TTY          TIME CMD
mirian    4576     1  0 21:22 pts/1    00:00:00 wxd
mirian    4705  3949  0 21:28 pts/1    00:00:04 xpdf SE2007-Proces
mirian    5068  3949  0 21:40 pts/1    00:00:00 ps -af
```

- The process ID or PID: a unique identification number used to refer to the process.
- The parent process ID or PPID: the number of the process (PID) that started this process.
- Terminal or TTY terminal to which the process is connected.

Linux/UNIX: Lister des informations sur les processus

En supprimant qqs champs de l'exemple, voilà ps avec des autres options:

```
[mirian@home2 Transp]$ ps au
USER       PID ... TTY      STAT START   TIME COMMAND
root      3093 ...  tty1     Ss+  20:57   0:00 /sbin/mingetty tty1
...
mirian    3881 ...  pts/0    Ss+  21:08   0:00 bash
mirian    3949 ...  pts/1    Ss   21:09   0:00 bash
mirian    4576 ...  pts/1    S    21:22   0:00 wxd
mirian    4705 ...  pts/1    S    21:28   0:00 xpdf SE2007-Process
mirian    4841 ...  pts/1    R+   21:33   0:00 ps au
```

STAT: status du processus

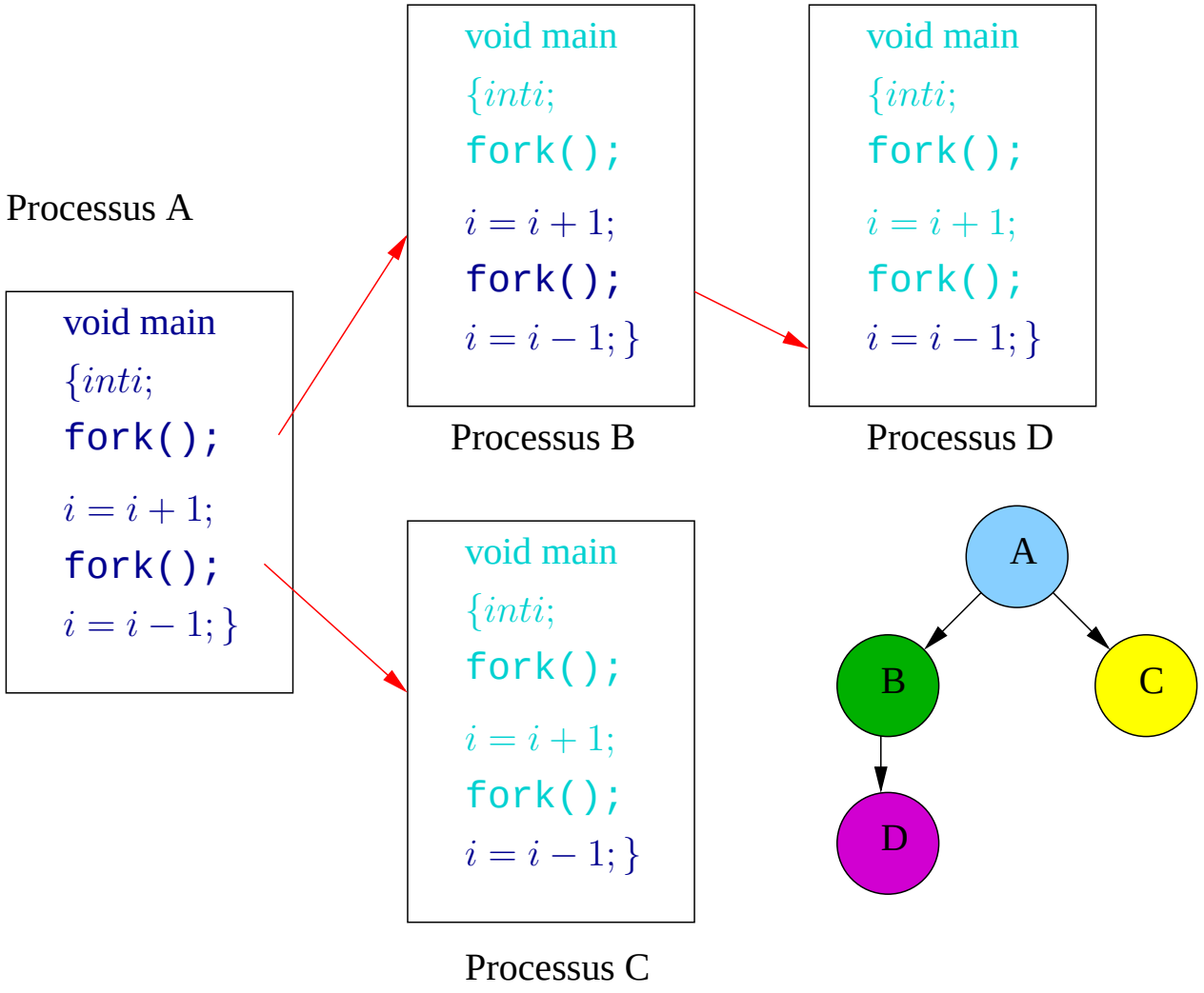
Certains états des processus

- R Running or runnable (on run queue)
- S Interruptible sleep (waiting for an event to complete)
- Z Defunct ("zombie") process, terminated but not reaped by its parent.

Hiérarchie de processus

- Dans certains SE, lorsqu'un processus crée un autre processus, les processus parent et enfant continuent d'être associés d'une certaine manière. Le processus enfant peut lui même créer plusieurs processus, formant un **hiérarchie de processus**.
- **Un processus a un seul parent et peut avoir** 0 **ou plusieurs fils**.
- Linux/UNIX:
  - Si le processus A crée le processus B, A est le parent de B, B est le fils de A (A par défaut, exécute le même code que B) B peut à son tour créer des processus. Un processus avec tous ses descendants forment un groupe de processus représenté par un arbre de processus.
  - `fork` est le seul appel système de création de processus.

Hiérarchie de processus



Linux/UNIX

- Démarrage Linux/UNIX: un processus spécial appelé `init` est présent dans l'image d'amorçage.
- Lorsqu'il s'exécute, il lit un fichier indiquant combien de terminaux sont présents; il génère un nouveau processus par terminal.
- Ce processus attendent une ouverture de session (`login`)
- Si l'une d'elles réussit, le processus de login exécute un **SHELL pour accepter des commandes**.
- Ces commandes peuvent lancer d'autres processus, et ainsi de suite.
- Tous les processus de l'ensemble du SE appartiennent à un arborescence unique, dont `init` est la racine.

Création d'un processus

- Événements provoquant la création d'un processus : initialisation système, exécution d'un appel système par un processus en cours, requête utilisateur sollicitant la création d'un processus.
- Linux/UNIX: L'appel système `fork` crée un clone du processus appelant.
- Après le `fork` les deux processus, père et fils, ont la même image mémoire et les mêmes fichiers ouverts.
- Généralement, le processus enfants exécute l'appel système `execve` pour modifier son image mémoire et exécuter un nouveau programme.
- Exemple:
  - Utilisateur adresse la commande `sort` au shell.
  - Le shell crée un processus fils qui exécute le programme `sort`
- Une fois qu'un processus est créé, le père et le fils disposent de leur propre espace d'adressage: s'il arrive que l'un des processus modifie un mot dans son espace d'adressage, le changement n'est pas visible par l'autre.
- Linux/UNIX: l'espace d'adressage initial du fils est une copie de celui du père : pas de mémoire partagée en écriture.

La fin d'un processus

Un processus peut se terminer suite à l'un des 4 événements :

- Sortie normale, lorsque le processus a terminé sa tâche (sous Unix par l'appel système `exit`)
- Sortie suite à une erreur (e.g. division par 0, inexistence d'un fichier passé en paramètre)
- Tué par un autre processus (sous Unix par l'appel système `kill`)

Toutes les ressources du processus sont libérées par le SE.

Exemple programme C avec `fork`

- Un nouveau processus est crée par l'appel système $fork$
- Le nouveau processus consiste en une copie de l'espace adresse du processus d'origine. Ce mécanisme permet au processus père de communiquer facilement avec son fils.
- Les 2 processus continuent l'exécution à partir de l'instruction située après le $fork$ mais avec une différence: le code de retour pour le $fork$ est zéro pour le nouveau processus (fils), tandis qu'on retourne au père l'identificateur du fils.
- L'appel système $execve$ est utilisé après un $fork$ par l'un des deux processus, afin de remplacer l'espace mémoire du processus par un nouveau programme.
- Si le père n'a aucune tâche à effectuer pendant que le fils s'exécute, il peut émettre un appel système $wait$ afin de sortir lui-même de la file d'attente des processus prêts jusqu'à la terminaison du fils.

Scheduling de processus

- Dans un système multitâche plusieurs processus sont en cours simultanément, mais le processeur ne peut, à un moment donné, exécuter qu'une instruction (d'un programme) à la fois. Le processeur travaille donc en **temps partagé**.
- L'ordonnanceur (scheduler) est le module du SE qui s'occupe de sélectionner le processus suivant à exécuter parmi ceux qui sont prêts.

Scheduling de processus

- Un processus passe entre les diverses files d'attente pendant sa durée de vie (file d'attente des processus prêts (attendant pour s'exécuter), fille d'attentes des périphériques, ...).
- Le SE doit sélectionner les processus à partir de ces files d'attente d'une manière quelconque.
- Le processus de sélection est mené à bien par le **scheduleur** approprié.
- Classification des processus
  - *Tributaire des E/S:* dépense la plupart du temps à effectuer des E/S plutôt que des calculs
  - *Tributaire de la CPU:* génère peut fréquemment des requêtes d'E/S, cad, passe plus de temps à effectuer des calculs

Scheduling des processus

- Le **scheduleur à long terme** (ou scheduleur de travaux)
  - Sélectionne le processus qui doit aller dans la fille de processus prêts.
  - S'exécute moins fréquemment : peut être lent (seconds, minutes).
  - Contrôle le degré de multiprogrammation (le nombre de processus dans la mémoire).
  - Il est important que le scheduleur à long terme réalise un bon mélange de processus tributaires de la CPU et tributaires des E/S
- Le **scheduleur à court terme** (ou scheduleur de la CPU):
  - Choisit parmi les processus prêts celui à être exécuté (alloue la CPU à lui).
  - Appelé assez fréquemment : doit être TRÈS rapide
- Possibilité d'introduire un **scheduleur à moyen terme**
  - *Idée clé:* Il peut être avantageux de supprimer des processus de la mémoire et réduire ainsi le degré de multiprogrammation.
  - Plus tard, un processus peut être réintroduit dans la mémoire et son exécution peut reprendre là ou elle s'était arrêtée
  - Ce schéma est appelé **swapping** (transfert des informations de la mémoire principale à la mémoire auxiliaire et vice-versa)

Scheduling des processus

- Les algorithmes d'ordonnancement (scheduleur) peuvent être classés en deux catégories:

1. **Non préemptif**
   - Sélectionne un processus, puis le laisse s'exécuter jusqu'à ce qu'il bloque (soit sur une E/S, soit en attente d'un autre processus) où qu'il libère volontairement le processeur.
   - **Même s'il s'exécute pendant des heures, il ne sera pas suspendu de force**.
   - Aucune décision d'ordonnancement n'intervient pendant les interruptions de l'horloge.
2. **Préemptif**:
   - Sélectionne un processus et le laisse s'exécuter pendant un délai déterminé.
   - Si le processus est toujours en cours à l'issue de ce délai, il est suspendu et le scheduleur sélectionne un autre processus à exécuter.

Scheduling des processus

- Les divers algorithmes de scheduling de la CPU possèdent des propriétés différentes et peuvent favoriser une classe de processus plutôt qu'une autre
- Critères utilisés:
  - *Utilisation de la CPU*: maintenir la CPU aussi occupée que possible.
  - *Capacité de traitement (Throughput)*: nombre de processus terminés par unité de temps.
  - *Temps de restitution (Turnaround time)*: temps nécessaire pour l'exécution d'un processus.
    Temps de restitution = temps passé à attendre d'entrer dans la mémoire + temps passé à la file d'attente des processus prêts + temps passé à exécuter sur la CPU + temps passé à effectuer des E/S
  - *Temps d'attente:* quantité de temps qu'un processus passe à attendre dans la file d'attente des processus prêts.
  - *Temps de réponse:* temps écoulé à partir du moment où on soumet une requête jusqu'à l'arrivée de la première réponse .

Algorithmes de Scheduling - FCFS

Scheduling du premier arrivé, premier servi
(FCFS: *First-come, first-served*)

- L'implantation de la politique FCFS est facilement gérée avec une file d'attente FIFO.
- Une fois que la CPU a été allouée à un processus, celui-ci garde la CPU jusqu'à ce qu'il la libère (fin ou E/S).
- FCFS est particulièrement incommode pour le temps partagé où il est important que chaque utilisateur obtienne la CPU à des intervalles réguliers.
- Temps moyen d'attente: généralement n'est pas minimal et peut varier substantiellement si les temps de cycles de la CPU du processus varient beaucoup
- Effet d'accumulation (*Convoy effect*): provoque une utilisation de la CPU et des périphériques plus lente que si on permettait au processus le plus court de passer le premier.

Algorithmes de Scheduling - SJF

Scheduling du travail plus court d'abord
(*SJF: Shortest job first*)

- Associe à chaque processus la longueur de son *prochain* cycle de CPU. Quand la CPU est disponible, elle est assignée au processus qui possède le prochain cycle le plus petit. Si deux processus possèdent la même longueur, le FCFS est utilisé.
- SJF est optimal: il obtient le temps moyen d'attente minimal pour un ensemble de processus donné.
- Difficulté: pouvoir connaître la longueur de la prochaine requête de la CPU
- Deux schémas: Non preemptif et preemptif.

Algorithmes de Scheduling - avec des priorités

- Une priorité est associée à chaque processus.
- CPU est allouée au processus de plus haute priorité (priorité $= 1 >$ priorité $= 2$
- Les processus ayant la même priorité sont schedulés dans un ordre FCFS.
- SJF: priorité déterminée par la durée du prochain cycle de CPU.
- Exemple

Processus	Temps de cycle	Priorité
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

- 2 schémas: Preemptif et non preemptif
- Problème: **Famine (*starvation*)** ou blocage indéfinie: processus avec des basses priorités peuvent attendre indéfiniment (ne jamais être exécutés).
- Solution: **Vieillissement (*aging*)**: technique qui consiste à augmenter

Algorithmes de Scheduling - RR

Scheduling du Tourniquet
(*RR: Round Rodin*)

- Chaque processus a une petite unité de temps appelée tranche de temps, ou quantum (en général de 10 à 100 ms).
- Le scheduleur de la CPU parcourt la file d'attente des processus prêts en allouant la CPU à chaque processus pendant un intervalle de temps allant jusqu'à un quantum.
- La file d'attente des processus prêts est implantée comme une file FIFO.
  - Cycle de CPU du processus < 1 quantum
    $\Rightarrow$ le processus lui-même libérera la CPU volontairement
  - Cycle de CPU du processus > 1 quantum
    $\Rightarrow$ l'horloge s'arrêtera et provoquera un interruption au SE
    $\Rightarrow$ Commutation de contexte
    $\Rightarrow$ Processus inséré en queue de la file d'attente

Algorithmes de Scheduling - RR

- Le temps moyen d'attente est souvent très long
- S'il existe n processus dans la file d'attente et quantum q alors:
  - Chaque processus obtient $1/n$ du temps processeur en morceaux d'au maximum q unités de temps.
  - Chaque processus ne doit pas attendre plus de $(n-1)q$ unités de temps jusqu'à sa tranche de temps suivante.
- Performance: dépend de la taille du quantum q
  q très grand $\Rightarrow$ FCFS
  q très petit $\Rightarrow$ partage du processeur
  MAIS... q doit être suffisamment grand par rapport au temps de commutation

Algorithmes de Scheduling - files d'attente à multiniveaux

Scheduling avec les files d'attente feedback à multiniveaux

- Séparer les processus ayant des caractéristiques différentes de cycle de CPU en différentes files. Permet aux processus de se déplacer entre les files.
  - Processus employant beaucoup de temps processeur: file de priorité inférieure
  - Processus tributaire E/S: file de priorité supérieure
  - Processus qui attend trop longtemps dans une file de priorité basse peut être déplacé à un file de priorité supérieure
- Exemple: 3 files d'attente
  - Q_0: quantum $8ns$; Q_1: quantum $16ns$; Q_2: FCFS
  - Un nouveau processus est inséré dans Q_0
  - Processus de Q_0: S'il ne finit pas en $8ns$, alors il est interrompu et placé en Q_1
  - Si Q_0 est vide, le processus en tête de Q_1 prend la CPU. S'il ne termine pas en $16ns$, il est interrompu et mis dans Q_2
  - Processus de Q_2: exécutés avec FCFS (seulement quand Q_0 et Q_1 sont vides)

Algorithmes de Scheduling - Évaluation

Comment choisir un algorithme de scheduling de CPU pour un système particulier?

Il existe différentes méthodes d'évaluation:

- Modélisation déterministe: Prend en compte une charge de travail prédéterminée et définit la performance de chaque algorithme pour cette charge de travail.
- Modèles de file d'attente
- Simulations
- Implantation

Commutation de contexte

- Le contexte d'un processus est l'ensemble des informations dynamiques qui représente l'état d'exécution d'un processus.
- La **commutation** de contexte est le mécanisme qui permet au SE de remplacer le processus élu par un autre processus éligible.
- Pour changer d'un processus à un autre, il faut
  - Sauvegarder l'état de l'ancien processus.
  - Charger l'état sauvegardé pour le nouveau processus.
- Le temps de la commutation de contexte est une surcharge car le système ne réalise pas de travail utile pendant qu'il commute.
- Le temps de la commutation de contexte dépend du support matériel.

Redirections - Linux/UNIX

Lorsqu'un processus démarre de façon standard, 3 fichiers spéciaux sont ouverts automatiquement :

- **Entrée standard** (stdin): descripteur 0, par défaut équivalent au clavier.
- **Sortie standard** (stdout): descripteur 1, par défaut équivalent à l'écran.
- **Sortie erreur standard** (stderr): descripteur 2, par défaut équivalent à l'écran

Ces 3 entrées/sorties peuvent être redirigées afin que la lecture/écriture se fasse à partir/vers d'autres fichiers.

- Pour rediriger la sortie d'un processus vers un fichier: $>$
- Pour rediriger la sortie d'un processus vers un fichier sans écraser son contenu: $>>$
- Pour lire l'entrée d'un processus à partir d'un fichier: $<$

Redirections - Linux/UNIX

`sort`	Trie les lignes tapées au clavier; écrit le résultat trié à l'écran
`sort < f1`	Trie les lignes du fichier $f1$; écrit le résultat trié à l'écran
`sort > f2`	Trie les lignes tapées au clavier; écrit le résultat trié dans le fichier $f2$ (si $f2$ existait avant il est efface et recréé)
`sort » f2`	Trie les lignes tapées au clavier; attache le résultat trié à la fin du fichier $f2$ (si $f2$ n'existait pas avant il créé)
`sort 2> f3`	Trie les lignes tapées au clavier; écrit le résultat trié à l'écran. Les messages d'erreur éventuels sont écrits dans le fi
`sort <f1 >f2 2>f3`	Trie le lignes du fichier $f1$; écrits le résultat trié dans le fichier $f2$ et les messages d'erreur éventuels sont écrits dans $f3$.
`sort < f1 2» f3`	Trie le lignes du fichier $f1$; attache les messages d'erreur éventuels à la fin de $f3$.
`<f1 >f2 2>f3 sort`	Trie le lignes du fichier $f1$; écrits le résultat trié dans le fichier $f2$ et les messages d'erreur éventuels sont écrits dans $f3$.

Les tubes

Nous pouvons aussi rediriger la sortie d'un processus vers la l'entrée d'un autre processus. Pour cela nous utilisons le pipe (**le tube**).

Exemple: : compter le nombre de mots uniques dans une liste .

Solution 1:

```
sort -u <f1 >f2 #Trie l'entrée standard en laissant
                #une occurrence unique de chaque ligne

wc -l <f2       #word count : affiche sur la sortie
                #standard le nombre de lignes
                #dans l'entrée standard
```

Solution 2: sans création de fichier intermédiaire.

```
sort -u <f1 | wc -l
```

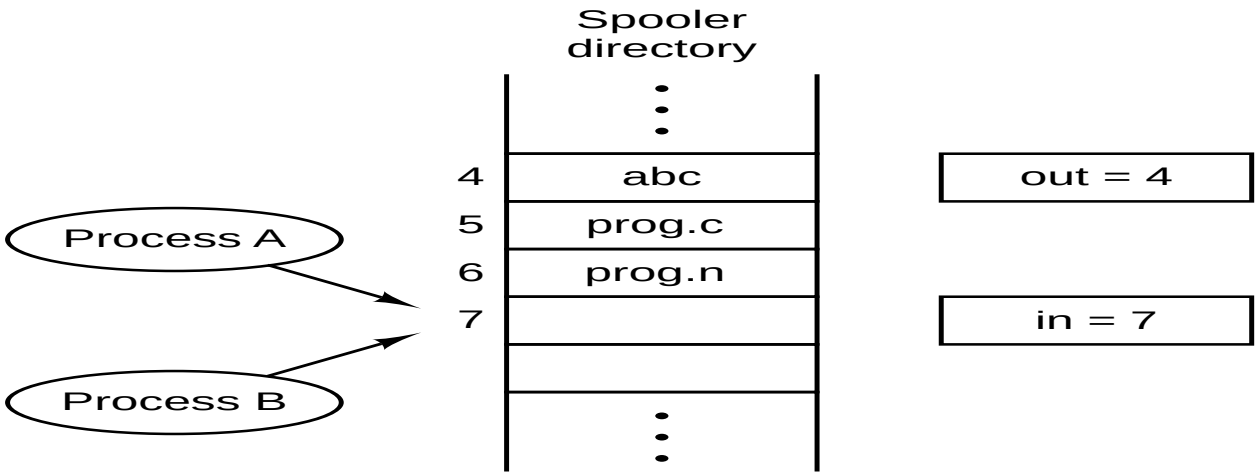
La communication inter-processus

- Les processus ont souvent besoin de communiquer entre eux (Exemple: pipeline - shell).
- Nécessité d'une communication entre processus de préférence de façon structurée, sans utiliser les interruptions.
- Le SE doit fournir aux processus coopératifs les moyens de communiquer entre eux par l'intermédiaire d'une fonction de communication inter processus.

IPC: *Interprocess communication*

Les problèmes de base de l'IPC

1. Comment un processus peut passer l'information à un autre?
2. Comment être sûr que les processus n'interviennent pas les uns avec les autres au moment d'exécuter des activités critiques?
   Exemple: 2 processus essaient de saisir les derniers $100K$ de mémoire
3. Comment assurer une séquence correcte quand il existe des dépendances ?
   Exemple: Si le processus A produit des données et le processus B les imprime, B doit attendre que A produise des données



Les conditions de concurrence

Race Conditions

Situation où plusieurs processus accèdent et manipulent en concurrence les mêmes données et où le résultat de l'exécution dépend de l'ordre dans lequel on accède aux instructions

- Comment éviter les **race conditions**?

  Interdire les processus de lire/écrire des données partagées au même moment.

- Nous avons besoin de **l'exclusion mutuelle (*mutual exclusion*)**

Problème de la section critique

- n processus $\{P_0, P_1, \ldots, P_{n-1}\}$ en concurrence pour utiliser des données partagées.
- **Section critique**: Segment de code dans lequel le processus peut accéder les données partagées, modifier des variables communes, mettre à jour une table, etc.
- Problème: Assurer que quand un processus exécute sa section critique, **aucun** autre processus n'est autorisé à exécuter la sienne.
- Structure du processus P_i:

repeat

Entry section

CRITICAL SECTION

Exit section

Reminder section

until false;

Conditions pour avoir une bonne solution

- **Exclusion mutuelle (*Mutual exclusion*):**
  Si le processus P_i exécute sa section critique, *aucun* autre processus peut exécuter la sienne.
- **Déroulement (*Progress*):**
  Un processus qui n'est pas dans la section critique ne peut pas bloquer un autre.
- **Attente limitée (*Bounded waiting*):**
  Aucun processus doit attendre indéfiniment.
- Aucune supposition concernant la vitesse relative de n processus.

Solutions

1. Via logiciel
2. Via matériel

Exclusion mutuelle avec attente active

Busy wait

Désactiver les interruptions

- Permettre à un processus de désactiver les interruptions pendant qu'une variable partagée est modifiée (cad, jusqu'après avoir entrer dans sa section critique).
- Avec la désactivation des interruptions, les interruptions de l'horloge ne sont pas possible et donc un processus peut garder le contrôle de la CPU.
- Mauvaise solution:
  - Pas très intelligent de donner au processus utilisateur le pouvoir de désactiver les interruptions.
  - Si le système est multiprocesseur, la désactivation des interruptions concerne seulement une CPU - les autres peuvent accéder la mémoire partagée.

Exclusion mutuelle avec attente active

Solution logicielle: Variables verrou (lock)

- Solutions pour 2 processus:
  Seulement 2 processus à la fois: P_0 et P_1
  Représentation: P_i
  Autre processus: P_j ($j = 1 - i$)
  Nous étudions 3 algorithmes
- Solution pour plusieurs processus:
  Algorithme du boulanger (*Bakery Algorithm*)

Algorithme 1

- Variables partagées (Shared variables):
  - `var turn: (0..1);` Initially `turn = 0`
  - `turn = i` P_i can enter its critical section .

Process P_i

repeat

while $turn \neq i$ do no-op;

critical section

turn := j;

remainder section

until false;

Satisfies mutual exclusion, but not progress.

Algorithme 2

- Variables partagées (Shared variables):
  - `var flag:  array [0..1] of boolean;`
    Initially $flag[0] = flag[1] = false$.
  - `flag[i] = true`
    P_i ready to enter its critical section

Process P_i

repeat

flag[i] := true;

while flag[j] do no-op;

critical section

flag[i] := false;

remainder section

until false;

Satisfies mutual exclusion, but not progress.

Algorithme 3

- Shared variables: Combined shared variables of algorithms 1 and 2.

Process P_i

repeat

$flag[i] := true$;

turn := j;

while ($flag[j]$ and $turn = j$) do no-op;

critical section

$flag[i] := false$;

remainder section

until false;

Meets all three requirements; solves the critical-section problem for two processes.

Algorithme du boulanger

Bakery Algorithm

- n processus
- Avant d'entrer dans la SC, un processus reçoit un numéro
- Celui qui a le plus petit numéro est celui qui va entrer dans sa SC

  Si P_i et P_j reçoivent le même numéro et si $i \leq j$
  alors P_i est servi d'abord
  sinon P_j est servi d'abord

  Noms des processus sont uniques et totalment ordonnés; l'algorithme est complètement déterministe.
- Le système de numérotation: ordre croissant

Bakery Algorithm

- Notation $<$: lexicographical order (ticket $\#$, process id $\#$).
- $(a, b) < (c, d)$ if $a < c$ or if $a = c$ and $b < d$.
- $max(a_0, ..., a_{n-1})$ is a number, k, such that $k \geq a_i$ for $i = 0, \ldots, n-1$
- Shared data

```
var choosing: array [0..n-1] of boolean;
    number: array [0..n-1] of integer;
```

- Data structures are initialized to false and 0, respectively

Bakery Algorithm

repeat

$choosing[i] := true;$

$number[i] := max(number[0], number[1], ..., number[n-1]) + 1;$

$choosing[i] := false;$

$for\ j := 0\ to\ n-1$

do begin

while choosing[j] do no-op;

while number[j] = 0

and (number[j],j) < (number[i], i) do no-op;

end;

critical section

number[i] := 0;

remainder section

until false;

Problème de l'attente active

- Gaspille le temps de CPU
- Peut avoir des résultats inattendus: inversion de la priorité
  Exemple:
  - 2 processus:
    H : priorité plus grande
    L: priorité basse
  - Scheduling: Si H est dans la file d'attente des processus prêts, alors H est choisi pour être exécuté
  - Situation:

Temps t_i: L est en exécution, L est dans la SC

Temps t_{i+j}: H arrive dans la fille des processus prêts

Donc... L est interrompu et H prend la CPU (H reste dans la boucle d'attente)

H ne sera jamais interrompu, L ne peut pas sortir de sa SC

Primitives IPC qui bloquent un processus

1. **Sleep**: appel système qui bloque un processus jusqu'à l'instant où il sera réveillé par un autre.
2. **Wakeup**: appel système pour réveiller un processus.

Exemple: Problème du producteur-consommateur

- Buffer plein et producteur veut ajouter un élément
  - Producteur va dormir
  - Reveillé quand le consommateur consomme
- Buffer vide et consommateur veut retirer un élément
  - Consommateur va dormir
  - Réveillé quand le producteur ajoute un élément
- Problème: *Race conditions*
  Producteur et consommateur peuvent dormir pour toujours

Producteur-consommateur

Producteur

```
repeat
        . . .
        if counter = n then Sleep();
        buffer[in] := nextp;
        in := (in + 1) mod n;
        counter := counter + 1
        if (counter = 1) then Wakeup(consumer);
until false
```

Consommateur

```
repeat
        if counter = 0 then Sleep();
        nextc := buffer[out];
        out := (out + 1) mod n;
        counter := counter - 1
        if (counter = n - 1) then Wakeup(producer);
        . . .
until false
```

Sémaphores

Un sémaphore S est une variable entière à laquelle (sauf pour l'initialisation) on accède seulement à travers deux opérations standards atomiques:

1. *Wait* (ou $DOWN$ ou P de *proberen*): correspond à $sleep$:
   - L'opération *DOWN* sur un sémaphore détermine si sa valeur est supérieure à 0.
   - Si c'est le cas, elle la décrémente et poursuit son activité.
   - Si la valeur est 0, le processus est placé en sommeil sans que le *DOWN* se termine.
2. *Signal* (ou UP ou V de *verhogen*): correspond à $wakeup$:
   - L'opération *UP* incrémente la valeur du sémaphore concerné.
   - Si un ou plusieurs processus dorment sur ce sémaphore, incapable de terminer une opération *DOWN* antérieure, l'un deux est choisi (aléatoirement) par le système et est autorisé à terminer son *DOWN*.
   - Une fois un *UP* accompli sur un sémaphore contenant des processus qui dorment, le sémaphore sera toujours à 0 mais il contiendra un processus en sommeil de moins.

Producteur-consommateur

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );                /* generate something to put in buffer */
          down(&empty);                         /* decrement empty count */
          down(&mutex);                         /* enter critical region */
          insert_item(item);                    /* put new item in buffer */
          up(&mutex);                           /* leave critical region */
          up(&full);                            /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                          /* decrement full count */
          down(&mutex);                         /* enter critical region */
          item = remove_item( );                /* take item from buffer */
          up(&mutex);                           /* leave critical region */
          up(&empty);                           /* increment count of empty slots */
          consume_item(item);                   /* do something with the item */
     }
}
```

L’échange de messages

- Méthode de IPC qui permet aux processus de communiquer sans avoir des variables partagées.
- Utilise 2 primitives: $send$ et $receive$
- Format:
  $send(destination, message)$
  $receive(source, message)$
- Un système de message doit traiter des problèmes de conception étrangers aux sémaphores et moniteurs - surtout si les processus sont dans des machine différentes. Exemple: message perdu

Producteur consommateur

producer

```
repeat
        produceItem (item);
        receive (consumer, message);
        buildMessage (message, item);
        send(consumer, message);
until false;
```

consummer

for $i = 1$ to N do send(producer, message)

```
repeat
        receive(producer, message);
        extractItem (message, item);
        send(producer, message);
        consumeItem(item);
until false;
```

Threads - Les processus légers

- Un thread contient:
  1. 1 compteur d'instructions
  2. 1 ensemble de registres
  3. 1 pile
- Un thread partage (avec des threads ressemblants)
  1. La section de code
  2. La section de données
  3. Les ressources du SE
- Processus traditionnels (lourds): *task* + *thread*
- *Threads:*
  1. Niveau système
  2. Niveau utilisateur

Threads - Les processus légers