Systèmes et Réseaux – MIAGE 2
Systèmes d'Exploitation

Processus

Sara Bouchenak

Sara.Bouchenak@imag.fr
http://sardes.inrialpes.fr/~bouchena/teaching

Plan

1. *Introduction aux systèmes d'exploitation*
2. **Processus**
   - ***Processus***
   - ***Commutation de processus***
   - ***Communication entre processus***
3. Gestion des processus
4. Fichiers
5. Mémoire
6. Etude de cas

Processus (1 / 2)

- Définition
  - un processus est l'entité dynamique représentant l'exécution d'un programme sur un processeur
- Différence entre processus et programme :
  - le programme est une description statique
  - le processus est une activité dynamique
    - il a un début, un déroulement et une fin
    - il a un état qui évolue au cours du temps

Processus (2 / 2)

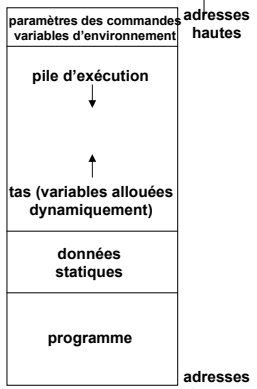
- Intérêt de la notion de processus
  - abstraction de la notion d'exécution séquentielle
  - ce qui la rend indépendante de la disponibilité effective d'un processeur physique
  - représentation des activités parallèles et de leurs interactions
- Exemple de processus
  - l'exécution d'un programme C
  - la copie d'un fichier sur disque
  - la transmission d'une séquence de données sur le réseau

Processus dans UNIX

- Un processus réalise l'exécution d'un programme
  - commande (du langage de commande, shell)
  - programme d'application

Contexte d'exécution d'un processus

- Le contexte d'un processus comprend
  - une mémoire qui lui est propre (mémoire virtuelle)
  - un contexte d'exécution (état instantané)
    - pile (en mémoire)
    - registres du processeur



mémoire d'un processus dans Unix

Identification d'un processus

- Un processus est identifié de manière unique par un numéro (*pid: Process IDentifier*)
  - La commande *ps* donne la liste des processus en cours d'exécution (voir *man ps*)
  - La fonction *getpid*() indique le numéro du processus qui l'exécute

```
#include <sys/types.h>      /* data types */
#include <unistd.h>         /* standard symbolic constants and types */

pid_t getpid(void);
```


Vie et mort d'un processus

- Un processus a généralement un début et une fin
  - Début : création par un autre processus
    - il existe un processus "primitif" créé à l'origine du système
  - Fin
    - auto-destruction (à la fin du programme)
    - destruction par un autre processus
    - certains processus ne se terminent pas ("démons", réalisant des fonctions du système)

Vie et mort de processus UNIX

- Dans le langage de commande
  - un processus est créé pour l'exécution de chaque commande
  - on peut créer des processus pour exécuter des commandes en (pseudo)-parallèle :
    - *prog1 & prog2 &*
      crée deux processus pour exécuter *prog1* et *prog2*
- Au niveau des appels système
  - un processus est créé par une instruction spéciale *fork*

Création des processus dans UNIX (1 / 2)

- L'appel système *fork* permet de créer un processus
- Le processus créé (le fils) est un clone (copie conforme) du processus créateur (le père)
  - Code identique
  - Contexte d'exécution identique

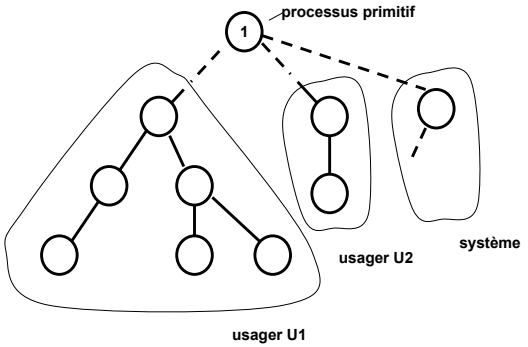
```
#include <sys/types.h>      /* data types */
#include <unistd.h>         /* standard symbolic constants and types */

pid_t fork(void);
```


Création des processus dans UNIX (2 / 2)

- Le père et le fils ne se distinguent que par le résultat rendu par *fork*
  - pour le père : le numéro du fils (ou -1 si création impossible)
  - pour le fils : 0

```
pid_t pid = fork();
if (pid != 0) {
        /* programme du père */
        ...
} else {
        /* programme du fils */
        /* en général exec (exécution d'un nouveau programme) */
}
```


Hiérarchie de processus dans UNIX



Hiérarchie de processus dans UNIX

- Fonctions utiles
  - *getppid()* : obtenir le numéro du père
  - *getuid()* : obtenir le numéro d'usager (auquel appartient le processus)

```
#include <sys/types.h>      /* data types */
#include <unistd.h>         /* standard symbolic constants and types */

pid_t getppid(void);
uid_t getuid(void);
```


Informations sur les processus UNIX

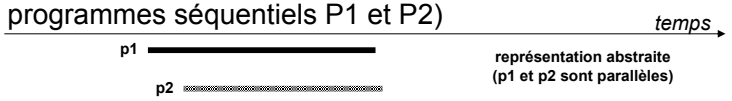
```
goedel> ps -l

F  S  UID   PID   PPID  C PRI  NI  ADDR   SZ   WCHAN     TTY      TIME     CMD
100 S 4518  969   968   0 75   0   -      811  rt_sig    pts/1    00:00:00 tcsh
000 R 4518  1029  969   0 74   0   -      790  -         pts/1    00:00:00 ps
```

- **S** : état du processus, S – Sleeping, R – Runnable
- **UID** : numéro de l'utilisateur
- **PID** : numéro du processus
- **PPID** : numéro du processus père
- **PRI** : priorité du processus
- **ADDR** : l'adresse mémoire du processus
- **SZ** : taille de l'image mémoire du processus
- **WCHAN** : l'adresse de l'évènement si processus en attente
- **TIME** : durée de l'exécution du processus
- **CMD** : nom de la commande à l'origine de la création du processus

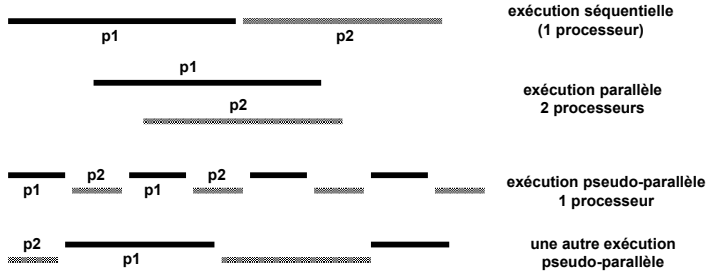
Plan

Parallélisme et pseudo-parallélisme

- Soient deux processus p1 et p2 (exécution de deux programmes séquentiels P1 et P2)



- Mise en œuvre concrète de l'exécution de p1 et p2

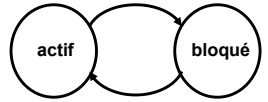


Alors …

- Questions
  - Comment s'exécutent plusieurs processus sur un processeur ?
  - Qui gère le (pseudo-)parallélisme des processus ?
- Réponse
  - Le système d'exploitation
  - ↳ C'est l'objet de ce cours …

États d'un processus

États logiques

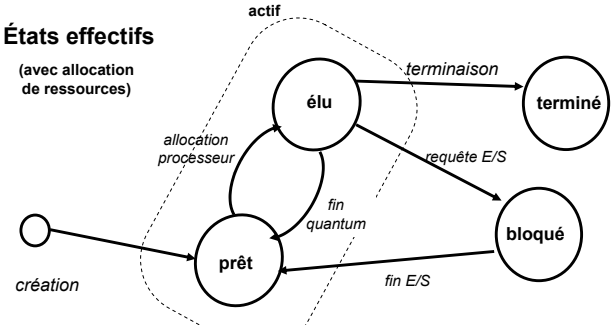


causes de blocage :

entrée-sortie
attente d'un signal
(entre autres *wait*, *pause*, *sleep*,...)

États effectifs

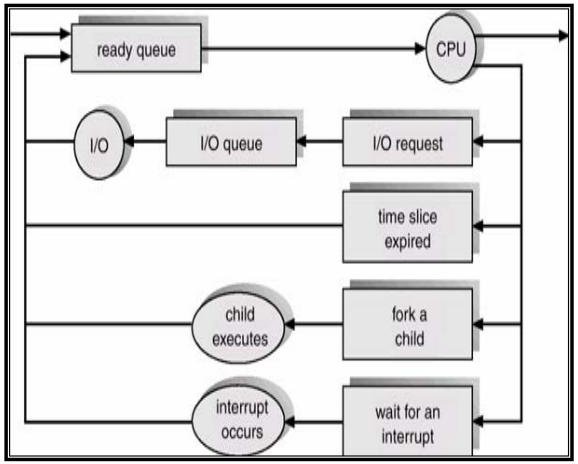
(avec allocation de ressources)



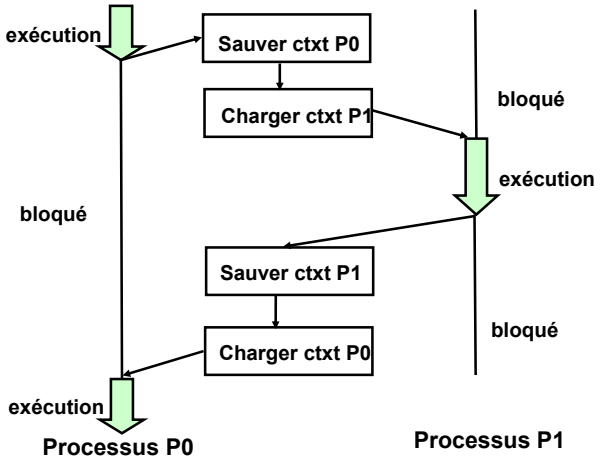
Files d'attente de processus (1/2)

- File des processus prêts
- File des processus bloqués
  - en attente d'une Entrée/Sortie
  - en attente de conditions de synchronisation

Files d'attente de processus (2/2)

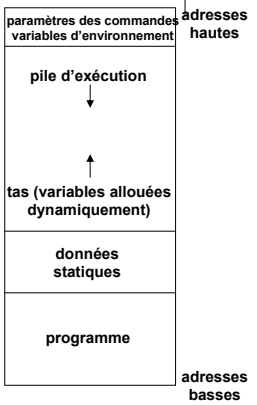


Commutation de processus



Rappel : Contexte d'exécution d'un processus

- Image mémoire (pile d'exécution, tas, …)
- Registres du processeur
  - Compteur ordinal (CO / PC)
  - Pointeur de sommet de pile (SP)
- Fichiers ouverts

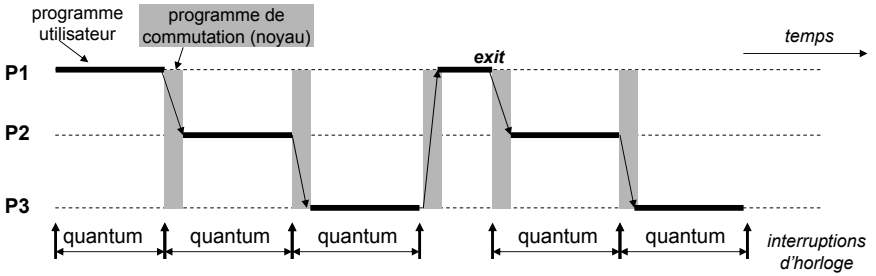


mémoire d'un processus dans Unix

Mécanisme d'allocation du processeur

- Que fait précisément le noyau du système lors de la commutation de processus ?
  1. Déterminer le prochain processus élu (en général, les processus sont alloués dans l'ordre d'arrivée, mais il peut y avoir des priorités). Le mécanisme utilise une file d'attente.
  2. Réaliser l'allocation proprement dite. Deux étapes :
     a. Sauvegarder le contexte du processus élu actuel (contenu des registres) pour pouvoir le retrouver ultérieurement, lors d'une prochaine allocation.
     b. Restaurer le contexte du nouveau processus élu
        - Restaurer les structures de données qui définissent la mémoire virtuelle.
        - Charger les registres, puis le "mot d'état" du processus, ce qui lance l'exécution du nouveau processus élu.

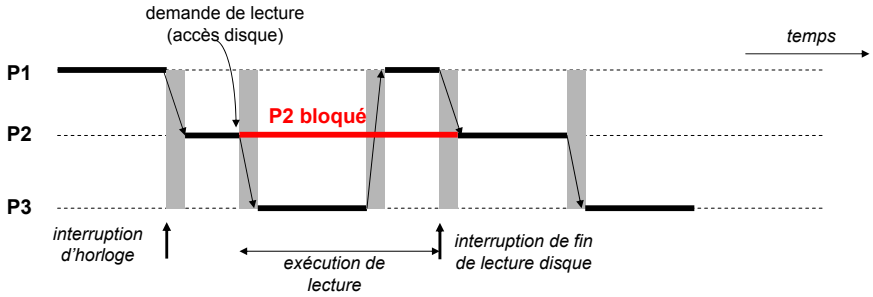
Allocation du processeur : Exemple détaillé



Allocation du processeur : Exemple détaillé (2)

- Le processeur est alloué par tranches de temps successives aux processus prêts à s'exécuter (non bloqués).
- Dans la pratique, la valeur du quantum est d'environ 10 ms (temps d'exécution de quelques millions d'instructions sur un processeur à 1 GHz).
- La commutation entre processus est déclenchée par une interruption d'horloge.
- La commutation entre processus prend un temps non nul (de l'ordre de 0,5 ms).
- La commutation entre processus est réalisée par un programme du noyau appelé ordonnanceur (*scheduler*).

Allocation du processeur : Autre exemple



Allocation du processeur : Autre exemple (2)

- Lorsqu'un processus est bloqué (par exemple, parce qu'il a demandé une entrée/sortie, ou a appelé *sleep*), il doit rendre le processeur puisqu'il ne peut plus l'utiliser.
- Le processeur pourra lui être réalloué à la fin de sa période de blocage (souvent indiquée par une interruption).
- Cette réallocation pourra se faire soit immédiatement (comme dans l'exemple donné), soit plus tard (après le quantum), selon la politique adoptée.

Plan

Communication entre processus

- La communication entre processus (IPC: *Inter-Process Communication*) est l'un des aspects les plus importants de la programmation des systèmes.
- Dans Unix, cette communication peut se faire de plusieurs manières :
  - Communication au moyen de signaux [suite de ce cours]
  - Communication par fichiers ou par tubes [cours 4]
  - Communication par files de messages [MIAGE 3]
  - Communication par mémoire partagée et par sémaphores [MIAGE 3]
  - Communication par sockets [cours réseaux]

Interruptions

- Définitions
  - Une interruption permet d'interrompre le processeur alors qu'il est en train d'effectuer une tâche quelconque
  - Le système est doté de 256 interruptions
  - Interruptions matérielles et logicielles

Interruptions matérielles

- **IRQ** (*Interruption Request*)
- 16 interruptions matérielles
- Les IRQ sont assignées au matériel : le processeur, la mémoire, les cartes d'extension, les périphériques, etc.
- IRQ donne la possibilité au matériel d'interrompre le processeur lorsque le matériel requiert un traitement immédiat
- Exemples d'interruptions
  - Le clavier informe le système qu'on appuie sur une touche
  - L'horloge système informe le processeur qu'il faut mettre à jour l'heure

Interruptions vs. Exceptions

- Les interruptions et les exceptions sont deux types de rupture "exceptionnelle" de flot
- Interruptions
  - Elles surviennent de façon asynchrones,
  - sont commandées par le matériel ou par le système,
  - n'ont aucune relation avec le programme en cours d'exécution
- Exceptions
  - Elles sont déclenchées par des "accidents" dans l'exécution du programme
  - Exemples
    - Division par zéro
    - Débordement arithmétique
    - Tentative d'utilisation d'instructions réservées
    - Erreur d'adressage

Signaux

- Définition
  - Les signaux sont pour les processus ce que les exceptions et les interruptions sont pour le processeur.
  - Certains signaux traduisent d'ailleurs la réception d'une interruption.
  - Le système fournit aux processus un moyen de s'envoyer des messages : des *signaux*.
  - Un **signal** est un évènement asynchrone destiné à un ou plusieurs processus.
  - Un signal peut être émis par un processus ou par le système d'exploitation.

Signaux : Principe de fonctionnement

- Le système fournit aux processus la liste des signaux disponibles
- Un processus, s'il le souhaite, définit quels sont les signaux auxquels il veut réagir
- Le processus associe une fonction de traitement à chaque signal
- À tout moment de sont exécution, le processus peut être interrompu par un signal
- Le processus interrompt son flot d'exécution classique et se déroute vers la fonction de traitement du signal
- A l'issue du traitement du signal, le processus reprend l'exécution là où il l'avait laissée

Caractéristiques des signaux

- Il existe différents signaux, chacun étant identifié par un **nom symbolique** (ce nom représente un entier).
- Chaque signal est associé à un **traitant par défaut**.
- Un signal peut être **ignoré** (le traitant associé est vide).
- Le traitant d'un signal peut être **changé** (sauf pour deux signaux particuliers).
- Un signal peut être **bloqué** (il n'aura d'effet que lorsqu'il sera débloqué).
- Les signaux ne sont **pas mémorisés**.

Exemples de signaux

Nom	Evènement associé	Traitement par défaut
SIGINT	Frappe <control-C>	terminaison
SIGTSTP	Frappe <control-Z>	suspension
SIGKILL	Terminaison du processus	terminaison
SIGSEGV	Violation de protection mémoire	terminaison + core dump
SIGALRM	Fin de temporisation (*alarm*)	terminaison
SIGCHLD	Terminaison d'un fils	ignoré
SIGUSR1	Signal émis par un processus utilisateur	terminaison
SIGUSR2	Signal émis par un processus utilisateur	terminaison
SIGCONT	Continuation d'un processus suspendu	reprise

Remarque : Les signaux SIGKILL et SIGTSTP ne peuvent être ni ignorés, ni bloqués et leur traitant ne peut être changé.

Etat d'un signal

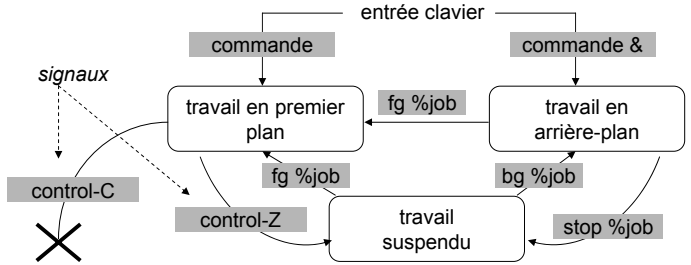
- Un signal est **envoyé** par un processus et **reçu** par un processus destinataire.
- Tant qu'il n'a pas été pris en compte par le destinataire, le signal est **pendant**.
- Lorsqu'un signal est pris en compte (exécution du traitant), il est dit **traité**.
- Qu'est-ce qui empêche qu'un signal soit directement traité dès qu'il est reçu ?
  - Le signal peut être **bloqué**, ou masqué (c.à.d. retardé) par le destinataire. Il est délivré dès qu'il est débloqué.
  - En particulier, un signal est **bloqué** pendant l'exécution du traitant d'un autre signal du même type ; il reste bloqué tant que ce traitant n'est pas terminé.
- Il ne peut exister qu'un signal pendant d'un type donné à la fois. S'il arrive un autre signal de même type, il est perdu

Structures internes associées aux signaux

No de signal	Pendant	Bloqué	Pointeur vers traitant	Masque temporaire (pendant traitant) 1 2 … NSIG
1	0/1	0/1	void (*)(int)	1 0/1 … 0/1
2	0/1	0/1	void (*)(int)	0/1 1 … 0/1
…				
NSIG	0/1	0/1	void (*)(int)	0/1 0/1 … 1

- Quand un signal d'un type i est reçu, pendant[i] = 1. Si alors bloqué[i]==0, le signal est traité puis pendant[i] est remis à 0
- Pendant l'exécution du traitant, un masque temporaire est placé pour bloquer certains signaux (signal i est automatiquement bloqué pendant l'exécution de son traitant, d'autres signaux peuvent être bloqués aussi)
- Si un signal i arrive alors que pendant[i]==1, alors il est perdu

Exemple : Shell



- Travail (*job*) = processus lancé par une commande au *shell*
- Seuls le travail de premier plan peut recevoir des signaux du clavier
- Les autres sont manipulés par des commandes au *shell*

Exemple (suite)

Programme loop.c :

```
int main() {
    printf("processus %d\n", getpid());
    while(1) ;
}
```

```
<unix> loop & loop & ps
processus 5123                    <---------- processus en arrière-plan
[1] 5123

processus 5124                    <---------- processus en arrière-plan
[2] 5124

PID   TTY      TIME      CMD
5100  pts/0    00:00:00  tcsh    <---------- processus en premier plan
5123  pts/0    00:00:00  loop
5124  pts/0    00:00:00  loop
5125  pts/0    00:00:00  ps
<unix>
```

Exemple (suite)

```
<unix> loop & loop & ps
processus 5123
[1] 5123
processus 5124
[2] 5124
PID   TTY     TIME      CMD
5100  pts/0   00:00:00  tcsh
5123  pts/0   00:00:00  loop
5124  pts/0   00:00:00  loop
5125  pts/0   00:00:00  ps

<unix> fg %1

loop

   => frappe de control-Z

Suspended

<unix>
```


Exemple (suite)

```
<unix> jobs
[1]  +  Suspended     loop
[2]  -  Running       loop

<unix> bg %1
[1]  loop &

<unix>

<unix> jobs

[1]  -  Running       loop
[2]  -  Running       loop
<unix>

<unix> fg %2

loop

   => frappe de control-C

<unix>
```


Exemple (suite)

```
<unix> ps
PID   TTY     TIME      CMD
5100  pts/0   00:00:00  tcsh
5123  pts/0   00:00:00  loop
5130  pts/0   00:00:00  ps
<unix>

<unix> => frappe de control-C>

<unix>

<unix> ps

PID   TTY     TIME      CMD
5100  pts/0   00:00:00  tcsh
5123  pts/0   00:00:00  loop
5130  pts/0   00:00:00  ps
<unix>
```


Définir un traitant associé à un signal

- Dans la suite, nous présentons comment définir le traitant d'un signal (en remplacement du traitant par défaut)
- Fonctions et structures standards

```
struct sigaction {
    void (*sa_handler)();    // pointeur sur traitant
    sigset_t sa_mask;        // autres signaux à bloquer
    int sa_flags;            // options
}
int sigaction(int sig, const struct sigaction* new sigaction, struct sigaction* oldaction);
```

- Primitive simple utilisée dans ce cours

```
#inclide <signal.h>
typedef void handler_t(int);   // un paramètre entier, ne renvoie rien
handler_t* Signal(int signum, handler_t* handler);  // associe un traitant à un signal
```

Exemple : Traitement d'une interruption au clavier

- Frappe control-C : interruption au clavier levant un signal de type SIGINT
- Traitant par défaut de SIGINT : tuer le processus concerné

test.c

```
int main() {
     pause();
     exit(0);
}
```

exécution

```
<unix> ./test

  => frappe de control-C
<unix>
```


Redéfinir le traitement d'une interruption

test.c

```
// Définir un nouveau traitant
void handler(int sig) {
     printf("Signal SIGINT reçu !\n");
     exit(0);
}

// Associer un traitant à un signal
int main() {
     Signal(SIGINT, handler);
     pause();
     exit(0);
}
```

Modifier le programme pour qu'il poursuive son exécution après la frappe control-C (au lieu de se terminer)

exécution

```
<unix> ./test

  => frappe de control-C

Signal SIGINT reçu !
<unix>
```


Plan

1. *Introduction aux systèmes d'exploitation*
2. **Processus**
   - *Processus*
   - *Commutation de processus*
   - *Communication entre processus*
3. Gestion des processus
4. Fichiers
5. Mémoire
6. Etude de cas

Références

- **Systèmes d'exploitation – 2ème édition**, A. Tanenbaum, Pearson Education, 2003.
- **Practical UNIX Programming**, K. A. Robbins, S. Robbins, Prentice Hall, 1996
- **Principe des systèmes d'exploitation des ordinateurs**, S. Krakowiak, Dunod, 1985.
- Ce cours a été conçu à partir d'autres supports :
  - Sacha Krakowiak, http://sardes.inrialpes.fr/people/krakowia/
  - Fabienne Boyer, http://sardes.inrialpes.fr/people/boyer/cours/SR/