FACULTÉ
DES SCIENCES
Unité de formation
et de recherche

UNIVERSITE
BRETAGNE
LOIRE

Système

Fabien GARREAU
sur la base du cours d'André Rossi

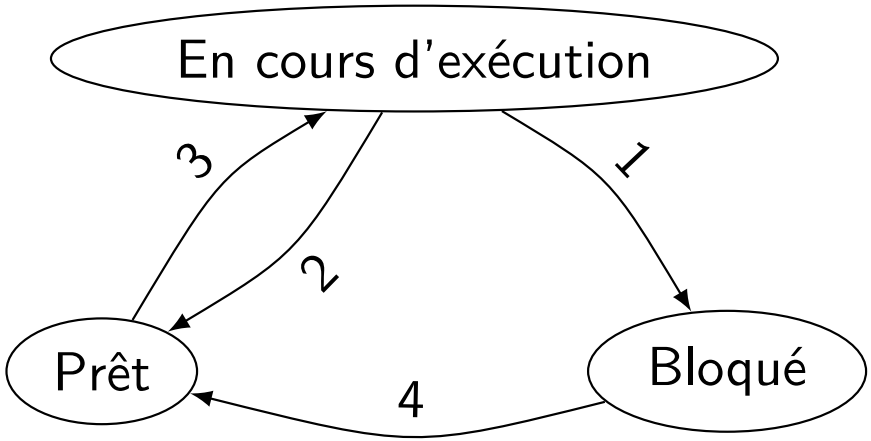
Université d'Angers

fabien.garreau@univ-angers.fr

L2 MPCIE, 2018–2019

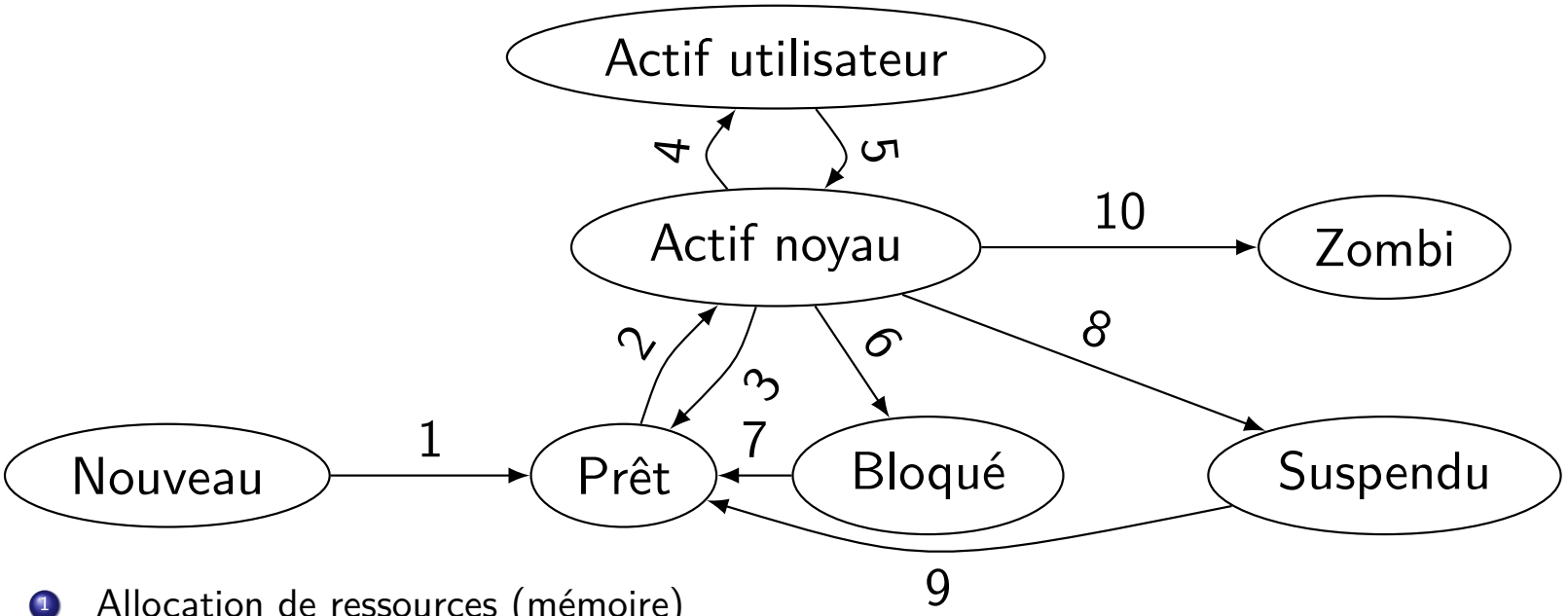
Les processus

Un processus est un programme en cours d'exécution, il a classiquement trois états :



1. Le processus en cours d'exécution est bloqué car une donnée (ou ressource) n'est pas disponible
2. Le processus en cours d'exécution est interrompu par l'ordonnanceur
3. Le processus prêt est choisi par l'ordonnanceur pour être exécuté
4. La donnée (ou ressource) manquante est disponible, le processus n'est plus bloqué, il attend d'être sélectionné par l'ordonnanceur.

Les processus d'UNIX



1. Allocation de ressources (mémoire)
2. Élu par l'ordonnanceur
3. Interrompu par l'ordonnanceur (round robin)
4. Passage en mode utilisateur, après un appel système ou une interruption
5. Passage en mode noyau (mode système) suite à une interruption ou pour faire un appel système
6. Le processus se met en attente d'un événement (E/S, `sleep`, `wait`)
7. L'événement attendu est arrivé
8. Processus suspendu par le signal `SIGSTOP`
9. Processus réveillé par le signal `SIGCONT`
10. Fin d'exécution

Les processus d'UNIX

Un processus est pourvu d'un espace d'adressage contenant :

- le code du programme, ses données et sa pile
- des registres pour le compteur ordinal, le pointeur de pile, etc

Un processus prend son sens pour un OS multiprogramme. Si on lance un traitement d'image très long, et qu'on surfe sur le WEB en relevant ses mails, on a trois processus, un par application. Périodiquement, l'OS interrompt le processus en cours pour en faire avancer un autre.

Un processus interrompu temporairement doit ensuite être redémarré exactement dans le même état : nécessité de sauvegarder le contexte du processus. Tout le contenu du processus, hormis l'espace d'adressage est sauvé dans la table des processus. Le PCB (*Process Control Block*) contient le PID, le PPID (ID du processus qui l'a créé) et le UID (User ID), l'état courant du processus (élu, prêt ou bloqué), le compteur ordinal du processus, le PSW, la liste des descripteurs de fichiers, le temps CPU accumulé par le processus, etc.).

Les processus d'UNIX

Les principaux appels systèmes pour la gestion des processus couvrent la création et la disparition des processus. Exemple : le terminal d'UNIX est un processus. L'utilisateur tape `g++ -o prog prog.cpp` : le terminal crée un nouveau processus. Quand ce processus a terminé la compilation, il lance un appel système pour se terminer.

Un processus peut créer des processus fils, ce qui conduit à une structure arborescente des processus (tout processus a un père, sauf la racine de l'arborescence qui est le processus `init`, dont le PID vaut 1 sous UNIX).

Un processus est dans l'état zombi lorsqu'il a terminé, mais que son père n'a pas encore pris en compte sa terminaison. Une fois la terminaison prise en compte, le processus est détruit.

Un processus non terminé dont le père est terminé est un orphelin. Sous UNIX, les orphelins sont adoptés par le processus `init`. Sous Linux, ils sont adoptés par un processus de l'utilisateur (par exemple 1700).

La communication inter-processus

Les processus peuvent communiquer. Soit un processus A qui envoie un message au processus B et demande un accusé de réception. S'il n'est pas arrivé au bout d'un temps t, l'OS doit le signaler à A afin qu'il tente, par exemple, de re-émettre le message. Une fois le timer lancé, le processus peut faire autre chose.

À l'expiration du délai, l'OS émet un **signal** (interruption logicielle) à A, qui démarre alors une procédure spécifique (réémission par exemple). Quand la procédure de gestion du signal est terminée, le processus reprend là où il a été interrompu. L'OS utilise les signaux en cas de tentative d'accès à une adresse hors de l'espace d'adressage, ou en cas d'instruction illégale.

Chaque utilisateur humain d'un système a un UID (user ID), les processus créés par cette personne héritent de l'UID, ainsi que les processus fils. Un processus a également un PID (Processus ID) qui le distingue des autres.

Visualisation des processus

On obtient la liste des processus en cours en tapant la commande UNIX `ps -e -o pid,ppid,uid,s,time,cmd`, et avec le gestionnaire des tâches de Microsoft Windows.

- `PID`
- `PPID`
- `UID`
- `S`
- `TIME`
- `CMD`

La colonne `S` indique : `R` en exécution, `S` prêt, `T` suspendu par `SIGSTOP`, `Z` zombi.
Pour voir les processus créés par le terminal courant :
`ps -o pid,ppid,uid,s,time,cmd`

Exemple d'envoi de signaux aux processus

Taper `gedit` : le terminal se bloque. `CTRL+Z`, il est suspendu (fenêtre minimale sous Ubuntu), on le voit avec `ps -e -o pid,ppid,uid,s,time,cmd`. On le réveille avec la commande `kill -SIGCONT 7220`, où 7220 est le PID du processus de `gedit`.

Pour lancer plusieurs processus sans bloquer le terminal, `gedit &`, on peut lire le PID du processus. Le terminer avec `kill -SIGTERM 6529`, sachant que `SIGTERM = 15`. Mais ce signal peut être masqué, et un processus suspendu ne prend connaissance des signaux qu'à son réveil.

On peut tuer un processus en adressant un signal au processus `init` avec `kill -SIGKILL 6529` sachant que `SIGKILL = 9`, le processus `init` reçoit l'ordre de tuer le processus 6529, ce dernier ne peut intercepter ce signal qui ne lui est pas adressé. On peut tuer un processus suspendu de cette manière.

Exemple sur les processus

Écrire un programme en langage C contenant un processus père qui crée 5 processus fils : le processus fils *i* affiche son `PID`, s'endort *i* secondes puis se termine. Le père stocke le `PID` de ces fils dans un tableau, et attend la fin des processus fils du dernier créé au premier créé.

- On utilisera les fonctions suivantes :
  - `setbuf(stdout, NULL);` pour débufferiser la sortie standard
  - `fork();` pour créer un processus fils
  - `getpid();` pour obtenir le PID du processus courant
  - `sleep(3);` pour endormir le processus courant pendant 3 secondes
  - `exit(0);` pour terminer le processus courant et renvoyer 0
  - `waitpid(A, &status, 0);` pour attendre la fin du processus `A`
- Utiliser ce programme et le terminal pour observer des processus dans l'état zombi.

Exercice de cours sur les processus – Solution

```
#include <stdio.h>  /* pour printf() */
#include <stdlib.h> /* pour exit() */
#include <unistd.h> /* pour fork(),
sleep(), getpid() */
#include <sys/wait.h> /* pour waitpid()
*/
#define N 5
int main(void)
{
int pid[N];
int pid_pere;
int i;
int status;
setbuf(stdout, NULL); /* Dé-bufferise
stdout (ou FILE* f) */

pid_pere = getpid();
printf("\nLe père a pour PID
%d",pid_pere);
for(i = 0; i < N; i++)
  {
  if((pid[i] = fork()) == 0)
    {
    /* Processus fils */
     printf("\nLe fils %d a pour pid %d."
     , i + 1, getpid());
    sleep(i + 1);
    exit(0);
     } /* end if */
  else
    {
    /* Processus père */
    printf("\nFils %d créé avec pid
     = %d.", i + 1, pid[i]);
    } /* end else */
  } /* next i */

/* Attente de la fin des fils */
for(i = N - 1; i >= 0; i--)
   {
  printf("\nAttente de la fin du fils
   %d...", i + 1);
  waitpid(pid[i], &status, 0);
  printf(" ok");
  } /* next i */
printf("\n");
return 0;
} /* end main */
```

Ordonnancement de processus

Quand plusieurs processus sont prêts, lequel élire ? On distingue deux types d'ordonnancements : les ordonnancements sans réquisition qui n'interrompent pas le processus en cours, et les ordonnancements avec réquisition. Les termes « préemption » et « réquisition » sont synonymes.

- FIFO : premier arrivé, premier sorti
  On choisit le premier processus prêt, et on ne l'interrompt pas (ordonnancement non préemptif). Simple, mais risqué : un processus peut monopoliser l'UC s'il ne se bloque jamais.
- Round robin (tourniquet)
  On choisit un quantum de temps constant T, et un ordre arbitraire des processus prêts. On laisse s'exécuter chaque processus pendant T unités de temps, après quoi on l'interrompt (ordonnancement préemptif). Si T est petit, la latence est faible, mais perte de temps à sauvegarder et restaurer le contexte des processus. Si T est grand, latence élevée, mais faible overhead (overhead = temps consacré par le système à sa propre gestion)

Ordonnancement de processus

- Ordonnancement par priorité

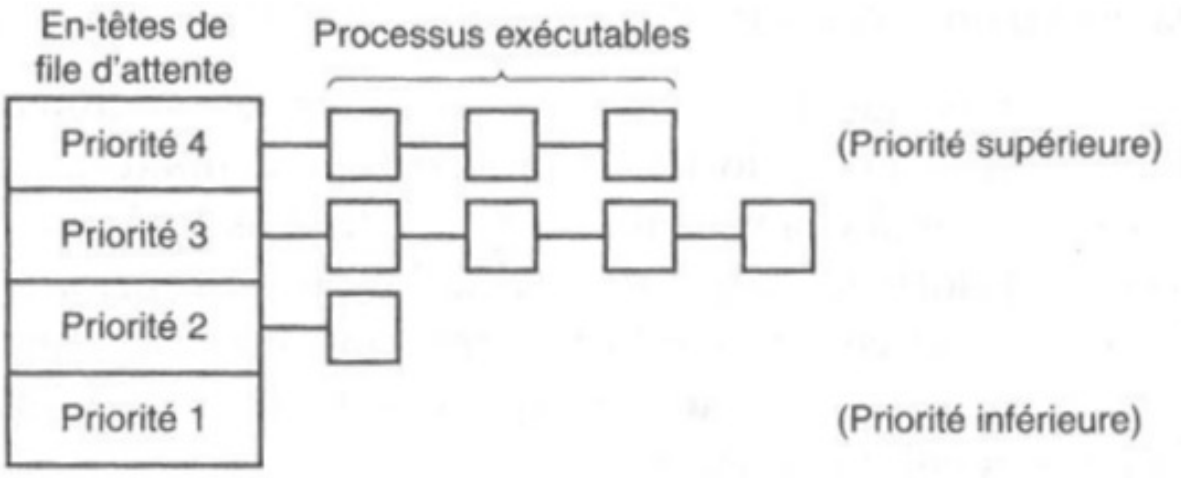
  Le round robin affecte le même quantum à tous les processus : ils sont implicitement d'égale importance. Or en réalité, une tâche de fond doit être moins prioritaire que l'affichage de l'écran par exemple.

  On affecte donc une priorité aux processus.

  Pour éviter que les processus prioritaires ne monopolisent l'UC, les priorités sont dynamiques. Sous UNIX, plus un processus a bénéficie de l'UC, moins il est prioritaire.

Ordonnancement de processus

En général, on utilise des classes de priorités :



Certains processus ne seraient jamais exécutés car pas suffisamment prioritaires si les priorités étaient statiques (**famine**) De manière générale, on cherche à ordonnancer les processus de manière **équitable**