C'est Quoi un Processus?

Processus.

1. C'est un programme qui tourne en machine (le chargement d'un code en mémoire)
   - Ensemble d'instructions et de données: code + données statiques + données allouées dynamiquement
2. Et une structure allouée par le système pour le contrôler = Environnement
   - Une partie pour la gestion du processus (appartient au noyau)
   - Une partie constituant le paramétrage du processus: arguments, variables d'environnement, etc.
3. Arborescence : Chaque processus a un unique père (sauf le 1er init).

Blocs de Contrôle d'un Processus (POSIX)

- Pid
- État
- Compteur ordinal
- Allocation mémoire
- Fichier ouverts...
- Tout pour suspendre/reprendre le processus

Etats d'un Processus

On est dans un environnement multi-processus.

1. Le processus utilise un laps de temps très bref le(s) processeur(s).
2. Particulièrement vrai pour un processus en attente d'une ressource (éviter de surcharger le système)
3. Ces différentes étapes d'un processus sont appelés Etats.
4. On a les états: Prêt pour l'exécution, Actif (utilise le processeur), bloquer/endormi (attend une ressource), suspendu (un utilisateur le désactive), zombie (réside en mémoire, mais ne peut être réactivé: par exemple pas de contrôleur de tâches pour le supprimer de la liste des processus)
5. Un processus peut demander intentionnelement à passer à l'état suspendu avec la fonction système sleep(unsigned int)

Création d'un Processus

1. Allocation d'un nouveau processus par clonage (appel système fork())
   - le fils hérite l'environnement du père (environnement d'exécution, variables système)
   - C'est une copie et non un partage : faire attention aux effets de bords (partage de tampon par exemple), descripteurs de fichiers dupliqués (partagent le même décalage par exemple).
   - Le fils reçoit un identifiant (numéro): `pid` différent de celui de son père `ppid`
   - Si un processus devient orphelin, il est adopté par le processus initial (le 1)
2. Remplacement du code père par un autre si besoin (exec*(chemin,arg,...))
   - Par le passé recopier tout, d'où le remplacement
   - Aujourd'hui: environnement seulement est copié et on récupère au fur et à mesure ce qui est important
3. Exemple au tableau

Création d'un Processus

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

int main ( ){
int i;
pid_t pid ;
printf("pid : %d, ppid : %d.\n", getpid(), getppid());
pid= fork();
if(pid== -1) perror("Fork fail :");// echec duplication
else if ( pid == 0 ){ // code processus fils
printf("pid : %d, ppid : %d.\n", getpid(), getppid());
sleep(5) ;
printf("pid : %d, ppid : %d.\n", getpid(), getppid()); }
else return 0; // code processus pere
return 0 ; }
```


Création d'un Processus

1. Allocation d'un nouveau processus par clonage (appel système fork())
   - le fils hérite l'environnement du père (environnement d'exécution, variables système)
   - C'est une copie et non un partage : faire attention aux effets de bords (partage de tampon par exemple), descripteurs de fichiers dupliqués (partagent le même décalage par exemple).
   - Le fils reçoit un identifiant (numéro): `pid` différent de celui de son père `ppid`
   - Si un processus devient orphelin, il est adopté par le processus initial (le 1)
2. Remplacement du code père par un autre si besoin (exec*(chemin,arg,...))
   - Par le passé recopier tout, d'où le remplacement
   - Aujourd'hui: environnement seulement est copié et on récupère au fur et à mesure ce qui est important
3. Exemple au tableau
4. Synchronisation parfois nécessaire

Création d'un Processus

Les différences (après duplication) entre père et fils sous POSIX

1. `pid` différents de tous les autres `pids` et `pgid`
2. `ppid fils` = `pid père`
3. Mesures de temps consommés initialisé à 0 [normal car pas encore consommé du processus]
4. Les verrous posés par le parent ne sont pas hérités [un verrou est relatif à un `pid`]
   - Un fichier verrouillé ne sera pas dupliqué
5. Minuterie désactivée
6. Ensemble des signaux pendants du père au moment de la duplication est initialisé à $\emptyset$.

Chargement d'un exécutable

1. Construction d'un nouveau code à partir d'un exécutable (précédent supprimé)
2. Initialisation de la liste d'arguments à transmettre au main
3. Initialisation de la variable globale `environ` (peut être une copie du processus)
4. Placement dans le CO l'adresse du main (on appelle le main en gros)
5. Six fonctions dans POSIX: `execl, execvl, execle, execve, execlp, execvp`
   - Descripteurs de fichiers verrouillés fermés (au niveau descripteur)
   - répertoires ouverts fermés
   - On ignore les signaux ignorés et le reste on prend le comportement par défaut
   - Les éléments d'environnement restants sont inchangés (répertoire courant, répertoire racine, masque, signaux pendants, etc.)
   - Si bit de positionnement (le fameux s à la place du x), alors on change `uid` avec celui de l'exécutable (de même pour `gid`).

Exemples d'Utilisation de exec

```
char *arguments ={"ls","-l",NULL}
execl("/bin/ls","ls","-l",NULL)
execv("/bin/ls", arguments)
execlp("ls","ls","-l",NULL)
execvp("ls",arguments)
```

```
int main ()
{
  system("ls -l /");
  return 0; }
```

Ça consiste en un fork() + exec() + wait()
C'est simple mais couteux : ça appel un shell
Ça fait 2 fork in fact !

Terminaison (1)

1. `exit(int)` (vidage tampons, flots ouverts fermés, etc.) ou `_exit(int etat)` (plus bas niveau)
2. L'état d'un processus arrêté peut être retourné à son parent si gestionnaire de tâches existe.
3. Utiliser les fonctions `wait` (bloquant) et `waitpid` (bloquant ou non et on peut demander une notification). Dans ce dernier cas on peut demander des infos d'arrêt sur plusieurs processus.
4. Faire attention aux processus fils zombis (en particulier il faut vraiment gérer les processus qui ne sont pas vraiment morst, mais juste par exemple stoppés).

Terminaison (2)

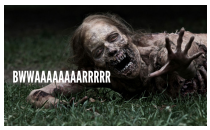
Le père est averti de la terminaison du fils (un signal `SIGCHLD`)

1. Cas 1: Le père récupère le code de retour du fils (donné à return ou exit) et appelle wait pour l'obtenir afin de libérer les ressources associées au fils

```
int status; pid_t pid_fils;
pid_fils = wait (&status);
if (pid_fils == -1) {
perror ("wait");
exit (EXIT_FAILURE);
}
if (WIFEXITED(status)) { /* terminaison normale */
printf ("%d = %d\n",pid_fils,WEXITSTATUS(status));}
```

2. Cas 2: le père ne récupère pas le code de retour, le fils devient un zombie: Lorsque le père meurt, les processus sont définitivement tués.
3. Cas 3: le père meurt avant le fils, le fils orphelin est adopté par `init`

Processus Zombie



Créer un zombie :

```
int main ( )
{
  pid_t pid ;
  pid= fork();
  if ( pid == 0 ) exit(0); //fils
  else sleep(10);//pere
  return 0; }
```

8874 pts/2 00:00:00 a.out
8875 pts/2 00:00:00 a.out <defunct>

Processus Zombie

Bref un père qui n'attends pas son fils
Un processus zombie et orphelin est adopté puis attendu !
Avec par exemple le code suivant :

```
while(waitpid(-1, NULL, WNOHANG))
```


Terminaison (3): Double fork

Que faire pour éviter les zombis ? =
Déléguer l'attente à un processus

1. On crée un processus fils
2. Le processus fils crée le processus que l'on voulait créer et meurt aussitôt
3. Celui que l'on voulait créer est adopté par `init`
4. Il ne peut pas être zombi car `init` nettoie (il est l'initial)
5. C'est la seule solution desfois car on n'a pas accés direct à `init`

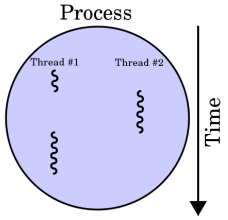
Les threads

Un thread est un chemin d'exécution d'un processus
Les threads sont inclus dans un processus

Les threads

Un thread est un chemin d'exécution d'un processus
Les threads sont inclus dans un processus

- Les threads d'un même processus partagent les ressources du processus sont partagés entre ces threads
  ⇒ Code exéc., mémoire, fichiers, périphériques...
- Chaque thread possède en plus : sa zone de données, sa pile d'exécutions, ses registres et son compteur ordinal



Pourquoi ?

- Pourquoi les threads ?
  - Rapide à créer : pas de contexte à créer
  - Communication inter-thread super simple et rapide !
  - Réactivité : un thread pour attendre l'user les autres bossent
  - Parallélisation d'une tâche !
- Plus difficile à manier : Synchronisation et Crash

Création de thread

Les fonctions de la bibliothèque pthread :

- int pthread_create(
  - pthread_t * thread,
  - pthread_attr_t * attr, (taille de la pile, priorité, ...)
  - void *nomfonction,
  - void *arg);

    ⇒ Créer un thread qui exécute nomfonction(arg)

Création de thread

Les fonctions de la bibliothèque pthread :

- int pthread_join(pthread_t *thid, void **valeur_de_retour);
- void pthread_exit(void *valeur_de_retour);
- int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);

  ⇒ sert à établir l'état de terminaison d'un processus léger :
  - PTHREAD_CREATE_DETACHED : le processus léger libérera ses ressources quand il terminera
  - PTHREAD_CREATE_JOINABLE : le processus léger ne libérera pas ses ressources

Création de thread

```
#include <stdio.h>
#include <pthread.h>

static void *task_a (void *p_data);
static void *task_b (void *p_data);

int main (void){
  pthread_t ta;
  pthread_t tb;
  pthread_create (&ta, NULL, task_a, NULL);
  pthread_create (&tb, NULL, task_b, NULL);
  pthread_join (ta, NULL);
  pthread_join (tb, NULL);
  puts ("main end");
  return 0; }
```