Processus

- Un **programme** c'est une description d'un calcul à faire (c'est-à-dire un algorithme)
- Un **processus** c'est un **programme en mémoire qui s'exécute.**
- Un **processeur** c'est une machine qui exécute un **processus**
- Par **multiplexage du CPU** il est donc possible d'avoir plus d'un processus qui évolue sur un même ordinateur (Multiprogrammation)
- **Chaque processus a l'illusion d'être sur un processeur qui lui est dédié**

Processus = Automate

- Pour représenter un processus, on peut se servir d'un **automate** (fini ou infini) qui contient tous les **états** possibles du processus
- Chaque état correspond à **une configuration des données** dans la mémoire du processus (précisément cela correspond aux variables globales, pile, et registres incluant le compteur ordinal)
- L'utilisation d'automates permet de mieux **comprendre** et **raisonner** sur le fonctionnement du processus
- À chaque processus est associé une structure dans laquelle est stocké l'état du processus :**Process Control Block** (PCB).

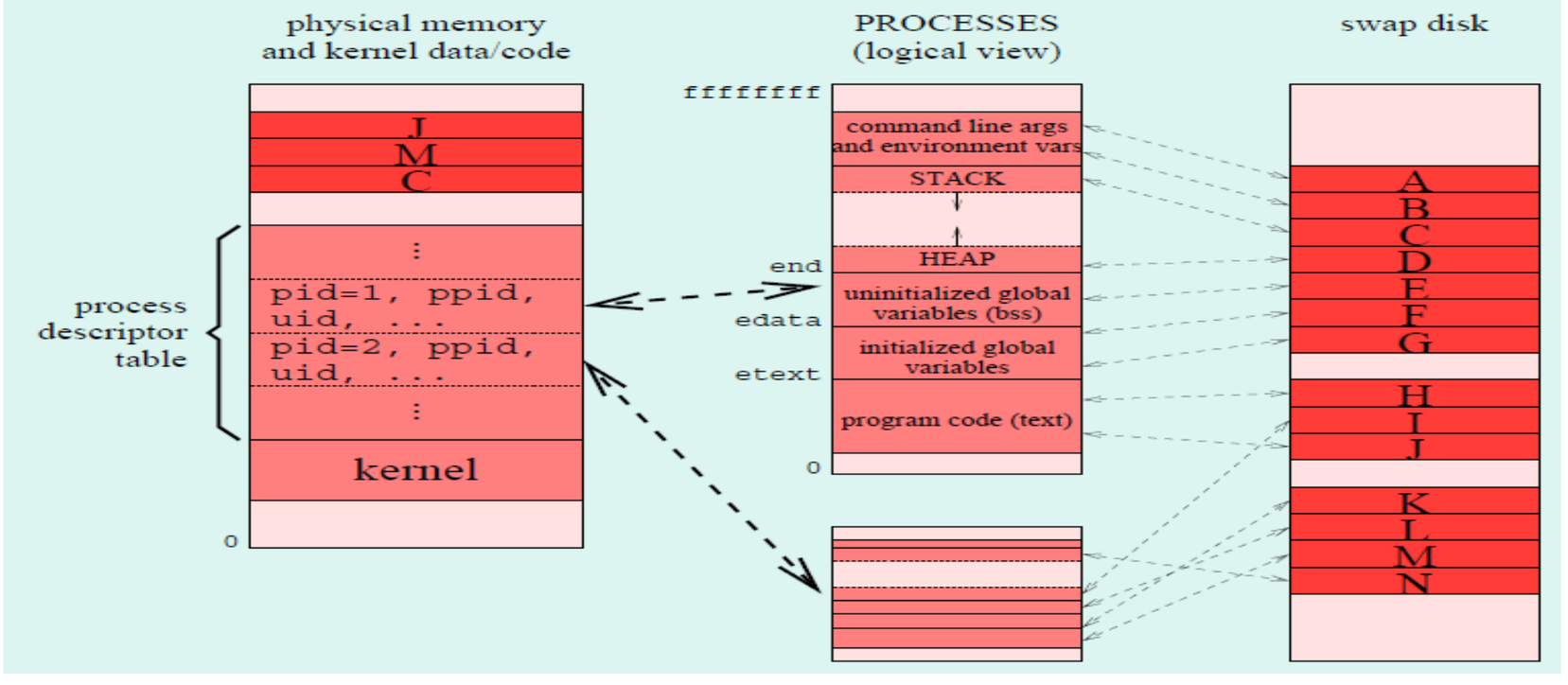
Le Modèle de Processus UNIX

- Les processus UNIX sont souvent appelés **processus lourds** (“**process**”) pour les distinguer des **processus légers** (“**threads**”) que nous verrons plus tard
- Les processus (lourds) sont **encapsulés**, c’est-à-dire que la mémoire principale accessible à un processus est **seulement accessible de ce processus**
  - Cela assure une certaine “**protection**” aux données du processus (il est impossible qu’un autre processus les modifie)
  - Grâce à la mémoire virtuelle, chaque processus a l’illusion d’avoir accès à un **espace d’adressage logique privé** complet (2^{32} octets); deux processus peuvent avoir des données à la même adresse logique sans que cela cause un conflit.

Organisation en Mémoire (1)

- Le kernel maintient une table (“**process descriptor table**”) qui contient pour chaque processus les attributs attachés à ce processus; en particulier le `PID` (“Process IDentifier”) qui est un entier qui identifie le processus (normalement entre 0 et 32767) et le `PPID` (“Parent PID”) qui est le PID du processus qui a créé ce processus
- L’espace mémoire accessible au processus contient
  - le code machine exécutable (“**text**”)
  - les variables globales initialisées (p.e. `int n = 5;`)
  - les variables globales non-initialisées (p.e. `int x;`) que le kernel initialise à zéro au chargement du programme (“**bss**”)
  - la pile et le tas (pour les allocations dynamiques)

Organisation en Mémoire (2)

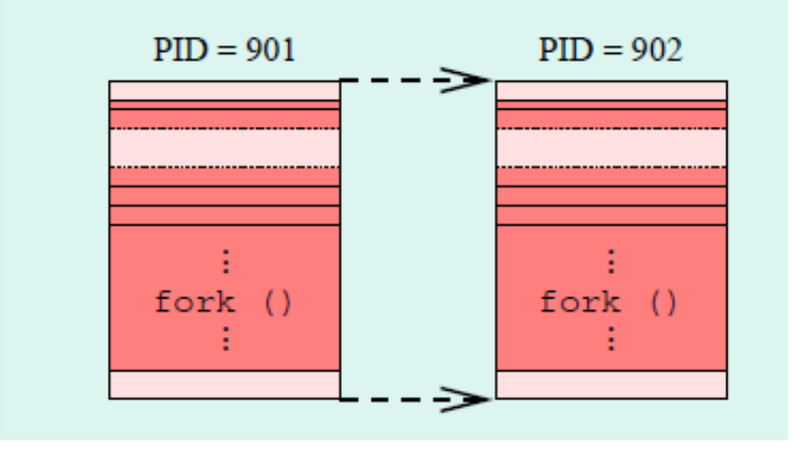


Création de Processus (1)

- L'unique façon de créer un nouveau processus c'est de faire appel à la fonction "`fork`" qui crée une **copie du processus courant** (le processus **enfant**)
- La copie est conforme, incluant les descripteurs de fichier et le contenu de la pile (donc l'enfant et le parent retournent tous deux de la fonction `fork`), sauf que
  - la fonction `fork` **retourne le PID de l'enfant** dans le parent et retourne 0 dans l'enfant
  - les PIDs et PPIDs sont différents (PPID(enfant) = PID(parent))

Création de Processus (2)

```
1. #include <iostream>
2. #include <sys/types.h> // pour pid_t
3. #include <unistd.h> // pour getpid, getppid, fork et sleep
4.
5. int main (int argc, char* argv[])
6. { cout << "PID=" << getpid () << " PPID=" << getppid () << "\n";
7.
8.   pid_t p = fork ();
9.
10.  cout << "PID=" << getpid () << " PPID=" << getppid ()
11.       << " p=" << p << "\n";
12.
13.  sleep (1);
14.  return 0;
15. }
```

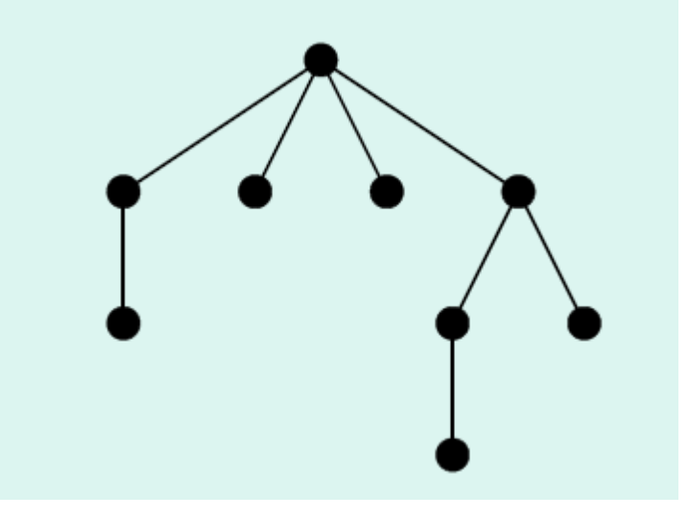



```
1. % ps
2.   PID TTY          TIME CMD
3.   817 pts/0    00:00:00 bash
4.   900 pts/0    00:00:00 ps
5. % echo $$
6. 817
7. % ./a.out
8. PID=901 PPID=817
9. PID=901 PPID=817 p=902
10.PID=902 PPID=901 p=0
```

- Pourquoi ‘sleep (1)’ ? Eviter que p soit orphelin..

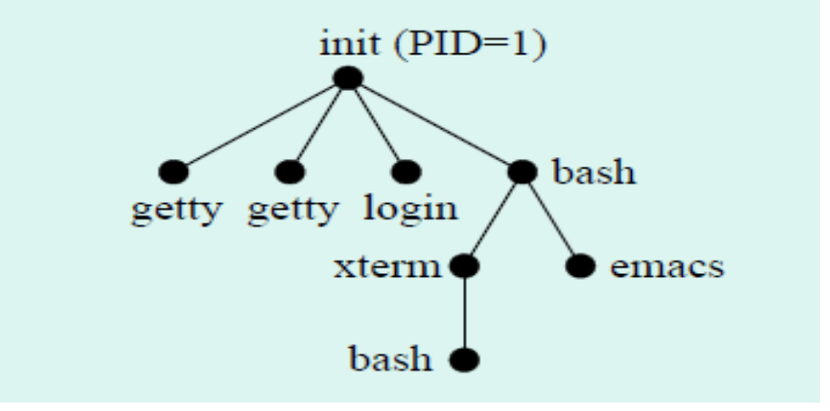
Exécution de Programmes

- La relation parent-enfant qui lie les processus donne une structure **hiérarchique** à l'ensemble des processus



- Tous les processus ont un parent, sauf celui à la racine
- Quel processus est à la racine?

- La racine est le processus “`init`” (PID=1) qui est créé spécialement par le kernel au moment du boot



- `init` se base sur un fichier de configuration pour démarrer les processus de service (`/etc/inittab`)
- En particulier, des processus exécutant le programme `getty` et `login` qui permet à un usager de se brancher
- Si le mot de passe fourni par l’usager est conforme à la base de donnée `/etc/passwd`, `login` change l’environnement, répertoire, UID, etc. du processus et fait un `exec` du shell de l’usager

Processus Orphelins

- Que se passe-t'il si le **parent d'un processus P meurt avant P**?
- Le processus P devient **orphelin**
- Le processus `init` (PID=1) adopte automatiquement les processus orphelins, de sorte que le parent de tout processus soit toujours un processus vivant
- Le processus `init` ne meurt jamais

Processus Zombie

- Que se passe-t'il si un processus P meurt mais qu'**aucun processus n'a encore obtenu son statut de terminaison** avec un appel à `wait` ou `waitpid`?
- Le processus P est un **zombie**
- Le problème c'est que le kernel ne peut savoir si plus tard un processus voudra obtenir le statut de P
- Le descripteur de processus de P **doit être préservé** par le kernel jusqu'à ce que son statut soit consulté avec un appel à `wait` ou `waitpid`
- Un zombie qui devient orphelin (i.e. son parent meurt) est éliminé de la table de descripteurs du kernel, car `init` exécute `wait` dans une boucle sans fin

Attente d'un Processus

- Les fonctions `wait` et `waitpid` permettent à un processus d'**attendre la terminaison** d'un processus et d'**obtenir le statut de terminaison**

```
pid_t wait (int* status)
pid_t waitpid (pid_t pid, int* status, int options)
```

- Cela est une forme simple de synchronisation
- Synchronisation de type "**fork-join**": on crée un sous-processus, on fait un travail concurrent, puis on attends que le sous-processus ait terminé

```
1. pid_t p = fork ();
2.
3. if (p == 0) // enfant?
4.   { f1 ();
5.     _exit (0);
6.   }
7.
8. f2 ();
9.
10. int statut;
11. waitpid (p, &statut, 0);
```

Processus legers :Threads

- L’ encapsulation offerte par les processus lourds est gênante lorsque les processus doivent **partager** et **échanger** des données
- Avec les processus légers (“threads”), **la mémoire est commune** à tous les processus légers
- Un processus peut écrire des données en mémoire et un autre processus peut immédiatement les lire
- Normalement, à chaque processus lourd est attaché un ensemble de processus légers; ceux-ci doivent respecter l’encapsulation du processus lourd
- À la création d’un processus lourd, il y a un processus léger qui est créé pour exécuter le code de ce processus lourd (thread “**primordial**”)

POSIX Threads (1)

- Les processus (légers) peuvent être **intégrés au langage de programmation** ou bien être **une librairie**
- Sous UNIX il y a plusieurs librairies de threads, "**POSIX Threads**" est une des librairies les plus portables
- Pour **créer et démarrer un nouveau thread** on utilise

```
int pthread_create(pthread_t* id,
                   pthread_attr_t*attr, void*
                   (*fn)(void*), void* arg)
```

- id = descripteur de thread
- attr = attributs de création, p.e. priorité (`NULL`= défauts)
- fn = fonction que le thread exécutera
- arg = paramètre qui sera passé à fn
- résultat = code d'erreur (0 = aucune erreur)

POSIX Threads (2)

- Pour **attendre la terminaison d'un thread** on utilise

  `int pthread_join (pthread_t` id, `void**` résultat)

- résultat = valeur retournée par la fonction d'exécution du thread (ou bien celle passée à `pthread exit`)

  `void pthread_exit (void*` résultat)

- Exemple

```
1.  #include <pthread.h>
2.
3.  void* processus (void* param)
4.  {
5.  int i = (int)param;
6.  for (int j = 0; j<10000000; j++) ;
7.  cout << i << "\n";
8.  return (void*)(i*i);
9.  }
11. int main ()
12. { pthread_t tid[5];
13. void* resultats[5];
14.
15. for (int i = 0; i<5; i++)
```

```
16.       pthread_create (&tid[i], NULL, processus, (void*)i);
17.
18.     for (int i = 0; i<5; i++)
19.         pthread_join (tid[i], &resultats[i]);
20.
21.     for (int i = 0; i<5; i++)
22.         cout << "resultat du processus " << i << " = "
23.                 << (int)resultats[i]                 << "\n";
24.
25.     return 0;
26. }
```