

Système d'exploitation

SYSIR3 - SYSG4

M. Bastreggi (mba)

Haute École Bruxelles Brabant — École Supérieure d'Informatique

Année académique 2020 / 2021

D'après le cours de M.Jaumain

Section Process

Process

- init/systemd
- naissance et vie
- shell
- pipe
- signaux
- ordonnancement

Process

Qu'est un **process** ?

Process - 1, le premier

En Unix, les processus sont créés par 'clonage' de leur père.

Au lancement du système, le processus **init/systemd** est créé par le noyau.

- ▶ premier processus
- ▶ appartient à l'administrateur (au root)
- ▶ initialise le système et les services
- ▶ reste actif en permanence.
- ▶ init/systemd adopte les processus orphelins

Système d'exploitation

└ Process

└ init/systemd

└ Process - 1, le premier

En Unix, les processus sont créés par 'clonage' de leur père.

Au lancement du système, le processus **init/systemd** est créé par le noyau.

- premier processus
- appartient à l'administrateur (au root)
- initialise le système et les services
- reste actif en permanence.
- init/systemd adopte les processus orphelins

Systemd remplace init à partir de 2010 pour accélérer le démarrage des services d'un système unix. Son utilisation est généralisée à partir de 2015. Il met en oeuvre plus de parallélisme pour les démarrages de services.

man init

man systemd

les premiers process

Les principaux autres processus créés au lancement sont des démons.

- ▶ cron, cupsd, sshd ...

la connexion en mode console, crée un processus, il s'agit d'un shell (interpréteur de commandes)

Le fichier `/etc/passwd` spécifie de quel processus il s'agit :

```
user0:x:1000:100:user0:/home/user0:/bin/bash
```

Système d'exploitation

└ Process

└ init/systemd

└ les premiers process

les premiers process

Les principaux autres processus créés au lancement sont des démons.

• cron, cupsd, sshd ...

la connexion en mode console, crée un processus, il s'agit d'un shell (interpréteur de commandes)

Le fichier `/etc/passwd` spécifie de quel processus il s'agit :

```
user0 : 1000:100 user0 : /home/user0 /bin/bash
```

- cron, crond - planification de commandes
- cupsd - impressions
- sshd - communications sécurisées
- smartd - SMART Disk Monitoring Daemon

Process - p i d

Chaque processus possède un identifiant : pid.

- ▶ C'est un entier non signé.
- ▶ Le process init ou systemd a le n° 1.

L'appel système **getpid** retourne l'identifiant du process appelant.

```
pid_t getpid(void); // retourne l'id du process
```


ps ajf

Voir l'identifiant des process et de leur parent

```
>>ps ajf
```

PPID	PID	PGID	SID	TTY	TPGID	STAT	UID	TIME	COMMAND
4473	5849	5849	5849	pts/4	6779	Ss	1000	0:00	/bin/bash
5849	6778	6778	5849	pts/4	6779	R	1000	0:00	_ ps ajf

La variable d'environnement \$ du shell contient l'identifiant du shell

```
>>echo $$
```

```
>>5849
```

les fils du shell

Le shell crée un processus fils chaque fois que nous lui demandons d'exécuter une commande **externe** (ls, ps, ...) ou un programme utilisateur (./a.out).

Système d'exploitation

└ Process

└ init/systemd

└ les fils du shell

Le shell crée un processus fils chaque fois que nous lui demandons d'exécuter une commande **externe** (ls, ps, ...) ou un programme utilisateur (./a.out).

Au départ un fils du shell est une copie du shell.

Il se différencie de son père par son id unique.

commande externe

Une commande externe correspond à un fichier exécutable sur le disque (ls, ps,...)

La commande **which ls** localise l'exécutable de la commande externe ls en se servant de la variable d'environnement **PATH** et affiche son chemin.

```
>>which ls  
>>/usr/bin/l
```

- ▶ une commande externe a sa propre documentation
- ▶ **man ls** documente ls

Système d'exploitation

└ Process

└ init/systemd

└ commande externe

commande externe

Une commande externe correspond à un fichier exécutable sur le disque (ls, ps,...)

La commande **which ls** localise l'exécutable de la commande externe ls en se servant de la variable d'environnement **PATH** et affiche son chemin.

```
>>which ls  
>>/usr/bin/ls
```

- une commande externe a sa propre documentation
- **man ls** documente ls

La commande **whereis** `<cmd>` localise l'exécutable de la commande cmd, son source et sa page de manuel

commande interne

une commande **interne** (SHELL BUILTIN COMMAND) ne passe pas par la création d'un fils, elle est réalisée par le code même du shell (cd, alias, history, source)

- ▶ une commande interne peut modifier l'environnement du textbfshell
- ▶ **man bash** documente les commandes internes

Exemple - un fils du shell

Les processus exécutés à partir d'une console sont fils du shell. L'appel système **getppid** permet d'obtenir l'identifiant du processus parent.

```
int main(void)
{ printf ("Je_suis_le_process_%d\n",getpid());
  printf ("Mon_père_est_le_process_%d\n",getppid());
  exit (0);
}
```

```
>>./ filsdushell
Je suis le process 12180
Mon père est le process 4710
>>echo $$
4710
```

Avancement

- init/systemd
- **naissance et vie**
- shell
- pipe
- signaux
- ordonnancement

Process - table

dans la mémoire du noyau la table des process :

- ▶ pid - le processus
- ▶ ppid - son père
- ▶ état - élu, ...
- ▶ contexte (registres, registres sélecteurs de segment CS DS SS)
- ▶ table des handles/descripteurs
- ▶ tables des signaux (en attente, pointeurs de fonctions)
- ▶ statistiques (CPU, ...)
- ▶ pointeur vers table des pages
- ▶ ...

Process - naissance par fork

```
pid_t fork(void);
```

appel système qui dédouble/clone le process appelant :

- ① sauvegarde le contexte du process courant dans la table des process
- ② clone le process
 - dédoublement de l'entrée de la table des process
 - dédoublement de l'espace d'adressage
 - adaptation de certaines valeurs
- ③ appel à l'**ordonnanceur**

fork - clonage

fork crée un clone du processus.

- ▶ Le fils hérite d'une copie des variables de son père.
- ▶ Les deux process 'repartent' du même RIP
- ▶ Les process sont indépendants, l'un ne peut pas modifier l'environnement de l'autre.

On ne peut prédire lequel des deux aura la main en premier.

Système d'exploitation

└ Process

└ naissance et vie

└ fork - clonage

fork crée un clone du processus.

- Le fils hérite d'une copie des variables de son père.
- Les deux process 'repartent' du même RIP
- Les process sont indépendants, l'un ne peut pas modifier l'environnement de l'autre.

On ne peut prédire lequel des deux aura la main en premier.

La valeur de RIP est une valeur relative, il faut en réalité la lire CS :RIP

L'ordre dans lequel les processus s'exécutent après l'appel système fork, dépend du choix qu'aura fait l'ordonnanceur.

L'appel système se termine en donnant la main à l'ordonnanceur (interruption logicielle)

fork - clonage

Qualis pater, talis filius

Exemple - fork

Après fork nous sommes bien en présence de deux process qui s'exécutent.

```
int main(void)
{ printf ("hello\n");
  fork ();
  printf ("world\n");
  exit (0);
}
```

```
hello
world
world
```

fork - clonage

Les processus père et fils sont identiques. Or, il faut les distinguer. L'appel système fork renvoie un entier qui vaut :

- ▶ 0 chez le process fils
- ▶ le pid du fils chez le process père
- ▶ -1 si le fork() n'a pas pu créer de clone.

L'Appel Système fork a **deux valeurs de retour**

fork - qui suis-je ?

```
if ((pid=fork()) == 0) {  
    // fils  
} else {  
    // père  
}  
// père et fils !
```


fork - qui suis-je ?

en assembleur intel 64 bits cela donne quelque chose comme ...

```
MOV RAX,1079 ; fork
SYSCALL      ; appel système fork → ordonnancement
OR RAX,0     ; ← RIP identiques chez père et fils (CS)
JZ  fils
```

pere: ...

fils : ...

- ▶ RAX=0 - chez le fils
- ▶ RAX=pid du fils - chez le père

Système d'exploitation

└ Process

└ naissance et vie

└ fork - qui suis-je ?

fork - qui suis-je ?

en assembleur intel 64 bits cela donne quelque chose comme ...

```
MOV RAX,1079 : fork
SYSCALL      : appel système fork -> ordonnanceur
OR RAX,0     : <- RIP identiques chez père et fils (CS)
JZ fils
```

père: ...
fils: ...

- RAX=0 - chez le fils
- RAX=pid du fils - chez le père

Un processus peut avoir plusieurs fils et connaît le pid de chaque fils grâce au retour de l'Appel Système **fork**.

Un processus connaît le pid de son père (un seul) grâce au retour de l'Appel Système **getppid**

fork - clonage

Conséquence du simple clonage ?

- ▶ mémoire : mêmes adresses **mais chacun chez soi** (segments, table des pages).
- ▶ fichiers ouverts : copie d'une référence vers TDFO -> **partage des entrées de la TDFO**

fork - clonage

Adaptations par l'OS :

- ▶ pid
- ▶ ppid
- ▶ statistiques // temps CPU ..
- ▶ signaux en attente
- ▶ RAX // 0 chez le fils, pid chez le père
- ▶ mémoire // attribution de mémoire physique propre au processus créé (on adapte CD, DS, ... ou la table des pages)

Système d'exploitation

└ Process

└ naissance et vie

└ fork - clonage

Adaptations par l'OS :

- pid
- ppid
- statistiques // temps CPU ..
- signaux en attente
- RAX // 0 chez le fils, pid chez le père
- mémoire // attribution de mémoire physique propre au processus créé (on adapte CD, DS, ... ou la table des pages)

Pour des raisons de performance les deux process vont commencer par partager l'espace d'adressage, il n'est nécessaire de le différencier qu'à partir du moment où on le modifie.

fork - clonage

- ▶ La table des handle est identique dans les deux process
- ▶ Elle contient les références vers la TDFO

-> partage des fichiers ouverts et de leurs offsets de lecture/écriture !

Exemple - fork

Après fork nous sommes bien en présence de deux process qui s'exécutent.

```
>./script > out
```

Les commandes de ce script partagent via leur stdout le même descripteur du fichier out

Pour obtenir ce comportement les appels système **open** (et dup2) doivent précéder l'appel système **fork**

Exemple - fork et exit

```
int main(void)
{ int r;
  printf ("hello\n");
  if ((r=fork()) == 0) {
    printf ("world\n");
  } // le code du fils ne se termine pas ici !
  else
    printf ("je préfère courir\n");
  printf ("je ne suis pas digne de ce monde\n");
  exit (0);
}
```

```
hello
je préfère courir
je ne suis pas digne de ce monde
world
je ne suis pas digne de ce monde
```


Système d'exploitation

└ Process

└ naissance et vie

└ Exemple - fork et exit

Exemple - fork et exit

```
int main(void)
{
    int i;
    printf("hello\n");
    if ((i=fork()) == 0) {
        printf("world\n");
    } // le code du fils ne se termine pas ici !
    else
        printf ("je préfère courir\n");
    printf ("je ne suis pas digne de ce monde\n");
    exit (0);
}
```

```
hello
je préfère courir
je ne suis pas digne de ce monde
world
je ne suis pas digne de ce monde
```

Deux process nécessitent deux exit.

Le process fils ne se termine pas dans le else !

L'affichage obtenu n'est pas déterministe :
il dépend de l'ordonnancement après le fork

Exemple - fork dans une boucle

```
int main(void)
{ int i;
  for (i=0; i<2; i++)
  { printf ("hello\n");
    fork ();
    printf ("world\n");
  }
  exit (0);
}
```

output ?

Système d'exploitation

└ Process

└ naissance et vie

└ Exemple - fork dans une boucle

```
int main(void)
{
    int i;
    for (i=0; i<2; i++)
    {
        printf("hello\n");
        fork();
        printf("world\n");
    }
    exit (0);
}
```

output ?

hello

world

hello

world

hello

world

world

world

world

quid de while (1) fork();

la "bombe fork" est une attaque de type "dénî de service"

un système s'en protège en limitant le nombre de process pour
le shell (ulimit)

Système d'exploitation

└ Process

└ naissance et vie

└ Exemple - fork dans une boucle

```
int main(void)
{
    int i;
    for (i=0; i<2; i++)
    {
        printf("hello\n");
        fork();
        printf("world\n");
    }
    exit (0);
}
```

output ?

- **ulimit** est une commande interne du shell, elle affecte l'environnement de ce dernier
- Élargir les limites nécessite des droits d'administration
- `ulimit -u`
- `ulimit -u 1000 //limite à 1000 process`
- `ulimit -u 2000 //Erreur`

Exemple - fork et variables

```
int main(void)
{ int r, var1=1;
  printf ("père_%d,ma_variable=%d\n",getpid(),var1);
  if ((r=fork())==0)
  {
    printf (" fils _%d,ma_variable=%d\n",getpid(),var1);
    sleep (1); // donnons une chance au père de terminer
    printf (" fils _%d,ma_variable=%d\n",getpid(),var1);
    exit (0);
  }
  var1=1000;
  printf ("père_%d,ma_variable=%d\n",getpid(),var1);
  exit (0);
}
```

Exemple - fork et variables

```
> ./a.out  
père 13777, ma variable=1  
père 13777,ma variable=1000  
  fils 13778,ma variable=1  
> fils 13778,ma variable=1
```

Les variables se trouvent bien dans des espaces de mémoire distincts

Système d'exploitation

└ Process

└ naissance et vie

└ Exemple - fork et variables

```
> ./a.out  
père 13777, ma variable=1  
père 13777, ma variable=1000  
fil 13778, ma variable=1  
> fil 13778, ma variable=1
```

Les variables se trouvent bien dans des espaces de mémoire distincts

Ici nous faisons le test avec une variable automatique.

Nous pouvons montrer la même chose avec des variables de classe d'allocation static et dynamique

Au passage, remarquons que l'affichage de la dernière ligne est particulier : le prompt du shell précède l'affichage fait par le fils.

Le shell aura attendu la fin de son fils pour récupérer la main en affichant le prompt

Il n'attend pas la fin de ses petits fils

fork et fichiers ouverts

La table des handle est clonée et contient des pointeurs dans la mémoire du noyau. . .



L'offset d'avancement du fichier va être partagé par père et fils !

Système d'exploitation

└ Process

└ naissance et vie

└ fork et fichiers ouverts

La table des handle est clonée et contient des pointeurs dans la mémoire du noyau...



L'offset d'avancement du fichier va être partagé par père et fils !

Pour tout fichier ouvert avant l'appel système **fork**, le processus père et fils partagent l'entrée de la TDFO et donc le pointeur d'avancement dans le fichier.

Cela rend possible la programmation des pipes et la redirection d'un groupe de commandes :

(ls;ps) > f - où () est un sous-shell

./Demo > f - où Demo est un script contenant plusieurs commandes. Les deux exemples précédents, exécutent un shell.

C'est ce shell qui invoquera **open** suivi de **dup**, avant de réaliser les **fork** correspondant aux commandes.

Exemple - fork et fichiers

```
int main(void)
{ int r, f; char buf[3];
  f=open("alphabet.dat",O_WRONLY|O_CREAT|O_TRUNC,0666);
  write(f,"abcdefghijklmnopqrstuvwxy",26);
  close(f);
  f = open("alphabet.dat",O_RDONLY);
  if ((r=fork())==0) { // fils
    read(f,buf,3); printf ("%c%c%c\n",buf[0],buf[1],buf[2]);
    //lseek(f,10,SEEK_CUR);
    exit (0);
  }
  // père
  //sleep(1);
  read(f,buf,3); printf ("%c%c%c\n",buf[0],buf[1],buf[2]);
  //lseek(f,2,SEEK_CUR);
}
```

Comportements ? Avec lseek ? Avec sleep ?

Système d'exploitation

└ Process

└ naissance et vie

└ Exemple - fork et fichiers

Exemple - fork et fichiers

```
int main(void)
{
    int r, i;
    char buf[3];
    if (open("alphabet.dat", O_WRONLY | CREATION_TRUNC, 0666))
        return 1;
    write(r, "abcdefghijklmnopqrstuvwxyz", 26);
    close(r);
    if (open("alphabet.dat", O_RDONLY))
        return 1;
    if ((r=fork())==0) { // fils
        read(r, buf, 3);
        printf("%c%c%c\n", buf[0], buf[1], buf[2]);
        // lseek(r, 10, SEEK_CUR);
        exit(0);
    }
    // père
    // sleep(1);
    read(r, buf, 3);
    printf("%c%c%c\n", buf[0], buf[1], buf[2]);
    // lseek(r, 2, SEEK_CUR);
}
```

Comportements ? Avec lseek ? Avec sleep ?

- indépendant de la séquence d'ordonnancement :
 - abc
 - def
- pourrait varier selon la séquence d'ordonnancement :
 1. en décommentant les lseek
 - abc
 - fgh
 2. en décommentant le sleep
 - abc
 - nop

Questions ?



Process - mort

Un process peut mourir de mort naturelle ou mettre fin à ses jours prématurément,
il peut aussi mourir de mort accidentelle ...

- ▶ `exit (0)`
- ▶ `exit (n)`
- ▶ réception d'un signal non traité

Système d'exploitation

└ Process

└ naissance et vie

└ Process - mort

Un process peut mourir de mort naturelle ou mettre fin à ses jours prématurément, il peut aussi mourir de mort accidentelle ...

- exit (0)
- exit (n)
- réception d'un signal non traité

Un signal ne tue pas forcément un processus.

Certaines erreurs génèrent un signal destiné au process même
SIGSEGV, SIGILL, SIGFPE,...

Fonction exit - la fin

Mettre fin au processus

```
void exit (int status);  
void _exit (int status);
```

- ▶ ferme les descripteurs de fichiers
- ▶ modifie le pid du père de ses fils (-> 1 init - adoption)
- ▶ prévient le père du processus, de cette mort (SIGCHLD)
- ▶ la valeur **status** est à lire par le père (wait)

Une instruction **return** dans le main est transformée en exit.

Système d'exploitation

└ Process

└ naissance et vie

└ Fonction exit - la fin

Fonction exit - la fin

Mettre fin au processus

```
void exit (int status);  
void _exit (int status);
```

- ferme les descripteurs de fichiers
- modifie le pid du père de ses fils (-> 1 init - adoption)
- prévient le père du processus, de cette mort (SIGCHLD)
- la valeur **status** est à lire par le père (wait)

Une instruction **return** dans le main est transformée en **exit**.

La variable d'environnement **?**, contient le statut de fin de la dernière commande exécutée.

la commande **echo \$?** affiche ce statut Dans le shell la valeur

0 veut dire vrai

`mkdir dir && cd dir`

`gcc p.c || echo erreurs`

wait4 - waitpid

Famille d'appels système permettant à un père de lire les status de ses processus fils terminés

- ▶ Le système ne libérera pas l'entrée de la table des processus assignée au fils tant que cette lecture n'aura pas eu lieu, un **zombie** occupe une entrée de la table des process.
- ▶ Dans le cas de fils encore en vie, ces appels système sont bloquants. C'est leur comportement par défaut.

wait4 - waitpid

```
pid_t waitpid(pid_t pid, int *status, int options);  
pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);
```

- ▶ pid : le process fils concerné ou -1 pour un fils quelconque.
- ▶ status : (en sortie) zéro ou raison de la mort exit(n) ou kill -m.
- ▶ option : attendre ou pas WNOHANG
- ▶ rusage : (en sortie) utilisation des ressources par le fils
- ▶ retour : le pid du fils terminé ou 0 si tous les fils sont encore en vie et WNOHANG a été spécifié, -1 si erreur.

Système d'exploitation

└ Process

└ naissance et vie

└ wait4 - waitpid

wait4 - waitpid

```
pid_t waitpid(pid_t pid, int *status, int options);  
pid_t wait4(pid_t pid, int *status, int options, struct rusage *rusage);
```

- pid : le process fils concerné ou -1 pour un fils quelconque.
- status : (en sortie) zéro ou raison de la mort exit(n) ou kill -m.
- option : attendre ou pas WNOHANG
- rusage : (en sortie) utilisation des ressources par le fils
- retour : le pid du fils terminé ou 0 si tous les fils sont encore en vie et WNOHANG a été spécifié, -1 si erreur.

Cette famille d'appels système permet de surveiller les changements d'état d'un fils (interrompu , relancé par signal ...).

La mort en est un cas particulier. Si le processus n'a aucun fils, l'appel système retourne une erreur

wait

```
pid_t wait(int *status);
```

status = 0 -> pas d'information en sortie

wait(&status) équivaut à waitpid(-1, &status, 0);

- ▶ bloque le process jusqu'à la fin d'un fils quelconque.
- ▶ status : zéro ou comment le fils s'est terminé exit(n) ou kill -m.
- ▶ retour : le n° du process terminé ou -1 si il n'existe (plus) aucun fils.

Exemple - while (wait());

```
int main(int argc, char * argv [])
{ if (fork()==0){ printf("Hello1\n"); exit(0);}
  if (fork()==0){ printf("Hello2\n"); exit(0);}
  if (fork()==0){ printf("Hello3\n"); exit(0);}

  while( wait(0) > 0 ); // attendre tous les fils

  if (fork()==0){ printf("_World\n"); exit(0);}
  exit(0);
}
```

wait - status

status

wait a comme paramètre en sortie un entier (dont 16 bits sont utilisés)

Raison de la mort du fils.

Un process se termine par `exit(n)` ou par `kill -m`.

Interprétation du status :

- ▶ `exit` = bits 8 à 15 de l'entier (8 bits) = `n`
- ▶ `kill` = bits 0 à 7 de l'entier (8 bits) = `m`

Exemple - wait status

```
int main(void)
{ int i,j,r;
  printf ("processus_%d\n",getpid());
  if ((r=fork())==0)
    { printf (" fils _%d\n",getpid());
  //   for (;;)
    exit (7);
  }
  i = wait(&j);
  printf ("père_%d\n",getpid());
  printf ("j' ai attendu la mort de mon fils_%d\n",r);
  printf ("c' est bien_%d qui est mort\n",i);
  printf ("et il est mort avec le status_(%d)_(0x%x)\n",j, j);
  exit (0);
}
```

Exemple - wait status

Exécution

```
processus 24414  
  fils  24415  
  père  24414  
  j'ai attendu la mort de mon fils 24415  
  c'est bien 24415 qui est mort  
  et il est mort avec le status (1792) (0x700)
```


wait - status

Une série de macros POSIX permettent au père de savoir si le fils est mort par exit ou kill ainsi que la valeur de m ou n.

```
WIFEXITED(status)  
WEXITSTATUS(status)  
WIFSIGNALED(status)  
WTERMSIG(status)
```

```
...  
i = wait(&j); // i est le fils mort  
              // j est la raison de sa mort  
if (WIFEXITED(j))  
    if (WEXITSTATUS(j) == EXIT_SUCCESS)  
        printf ("est_mort_de_sa_belle_mort\n");
```

Exemple - wait status

Exécution en décommentant la boucle for

```
> ./a.out&
processus 24443
fils 24446

> kill 24446

père 24443
j'ai attendu la mort de mon fils 24446
c'est bien 24446 qui est mort
et il est mort avec le status (15) (0xf)
```

no wait ? zombie

Un process se termine par `exit` ou réception d'un signal.

Il reste présent dans la table des process jusqu'à ce que son père utilise un appel système `wait` pour prendre connaissance de son état.

Ce n'est qu'à ce moment que l'entrée de la table des process sera libérée.

Entre le moment où le process se termine et le moment où le père effectue le `wait`, le process est en l'état **zombie**.

Exemple - zombie

```
int main(void)
{ int r;
  if ((r=fork())==0) exit(0);
  sleep (1);
  system("ps"); // la fonction system exécute une commande
                // dans un sous-shell
  for (;;)
    exit (0);
}
```

Exécution :

23959 pts/5 00:00:00 a.out <defunct>

Système d'exploitation

└ Process

└ naissance et vie

└ Exemple - zombie

Exemple - zombie

```
int main(void)
{
    int r;
    if ((r=fork())==0) exit(0);
    sleep (1);
    system("ps"); // la fonction system execute une commande
                  // dans un sous-shell
    for (;;)
        exit (0);
}
```

Exécution :

23959 pts/5 00:00:00 a.out <defunct>

Comment afficher l'état zombie dans le programme ?

- system ("ps")

la commande system crée un nouveau shell ...

zombie

- ▶ Un zombie disparaît dès la lecture de son état par son père.
- ▶ Il existe des cas où le père ne fait jamais de wait de ses fils (pour éviter le blocage et l'attente active)
Serveurs réseau, shell pour commande en arrière plan, ...
- ▶ La gestion du signal SIGCHLD reçu par le père à la mort de son fils permettra de gérer ces cas.



La présence de zombies peut être gênante si elle perdure, car ceux-ci bloquent des entrées de la table des processus

adoption - déléguer init

Si le père se termine sans avoir fait de wait de son fils, ce dernier sera **adopté** par init qui devient responsable de l'élimination des zombies

- ▶ lors de la mort d'un père, les processus devenus orphelins sont adoptés par init (1)
- ▶ un processus est adopté par init qu'il soit terminé ou non
- ▶ init élimine par wait les fils adoptés lorsqu'ils se terminent.

Système d'exploitation

└ Process

└ naissance et vie

└ adoption - déléguer init

adoption - déléguer init

Si le père se termine sans avoir fait de wait de son fils, ce dernier sera **adopté** par init qui devient responsable de l'élimination des zombies

- lors de la mort d'un père, les processus devenus orphelins sont adoptés par init (1)
- un processus est adopté par init qu'il soit terminé ou non
- init élimine par wait les fils adoptés lorsqu'ils se terminent.

l'adoption permet d'éviter que les processus zombie dont le père serait terminé, encombrant la table des process.

la technique du "**double fork - exit**" consiste à créer d'emblée des orphelins.

Elle force une adoption.

C'est une manière de régler le problème des zombies, il est aussi possible d'éliminer un zombie en s'aidant du signal SIGCHLD (traiter - ignorer).

Exemple - adoption

```
int main(void)
{ int r;
  printf ("Je_suis_le_processus_père_%d\n",getpid());
  if ( (r=fork()) == 0)
  {
    while (getppid() != 1) ; // attend l'adoption par init
    printf ("Je_suis_le_fils_%d,_"
           "mon_père_est_le_%d\n", getpid(),getppid());
    exit (0);
  }
  printf ("Je_suis_le_père_%d,_"
         "j'ai_un_fils_%d\n", getpid(),r);
  exit (0);
}
```

```
Je suis le processus père 16744
Je suis le père 16744, j'ai un fils 16745
Je suis le fils 16745, mon père est le 1
```

Exemple - zombie et double fork

```
int main(int argc, char * argv [])
{ int r;
  printf ("Je_suis_le_processus_père_%d\n",getpid());
  if ( (r=fork()) == 0) // père temporaire
  {
    if ( (r=fork()) == 0){
      usleep (1); // pourquoi ?
      printf ("Je_suis_le_fils_non_attendu:(_%d,_\n"
              "mon_père_est_le_%d\n", getpid(),getppid());
      exit (0);
    }
    else exit (0); // mon fils sera adopté
  }
  wait(0); // wait père temporaire
  while(1);
  exit (0);
}
```

Exemple - wait4 & struct rusage

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<sys/wait.h>
#include<sys/resource.h>
int main(void)
{ pid_t pid; int status; struct rusage usage; int i,j;
  if ((pid = fork()) == 0)
  { printf ("pid_du_fils =%u\n", getpid());
    j = 0; for(i=0; i < 5000000; i++) {j+=i; write(2,"a\n",2);}
    exit (0); }
  if ((wait4(pid, &status, 0, &usage)) > 0 )
  { printf ("Temps_utilisateur_%ld_s,%ld_microsec\n",
    usage.ru_utime.tv_sec, usage.ru_utime.tv_usec);
    printf ("Temps_en_mode_noyau_%ld_s,%ld_microsec\n",
    usage.ru_stime.tv_sec, usage.ru_stime.tv_usec); }
  exit (0);
}
```

Questions ?



Process - chargement

L'appel système `execve` remplace le contexte d'exécution d'un process par un nouveau contexte décrit dans un fichier exécutable.

Tous les segments du programme (texte, données et pile) sont remplacés par ceux du nouveau qui démarre à sa fonction `main`. et se termine par un `exit`



`execve` est un 'chargeur'

execve

```
int execve (const char * fichier , char * const argv [],  
            char *const envp []);
```

- ▶ fichier : chemin de l'exécutable sur le disque.
- ▶ argv : tableau des arguments passés au main.
- ▶ envp : tableau des variables d'environnement passé au main.
- ▶ pas de retour : ne retourne qu'en cas d'erreur (renvoie -1 dans ce cas)

pas de retour en fonctionnement normal !

Système d'exploitation

└ Process

└ naissance et vie

└ execve

execve

```
int execve (const char * fichier, char * const argv [],  
            char * const envp []);
```

- fichier : chemin de l'exécutable sur le disque.
- argv : tableau des arguments passés au main.
- envp : tableau des variables d'environnement passé au main.
- pas de retour : ne retourne qu'en cas d'erreur (renvoie -1 dans ce cas)

pas de retour en fonctionnement normal !

voici un extrait de la page de manuel de execve :

argv est un tableau de chaînes d'arguments passées au nouveau programme. envp est un tableau de chaînes, ayant par convention la forme clé=valeur, qui sont passées au nouveau programme comme environnement. argv ainsi que envp doivent se terminer par un pointeur NULL.

En cas de réussite, execve() ne revient pas à l'appelant, et les segments de texte, de données (« data » et « bss »), ainsi que la pile du processus appelant sont remplacés par ceux du programme chargé.

execve

execve

- ▶ remplace les segments - text, data, stack
- ▶ efface les signaux en attente et restaure leur comportement par défaut
- ▶ conserve le pid du processus appelant
- ▶ conserve par défaut les descripteurs de fichiers (handle)

Fonctions exec

La famille de fonctions exec simplifie l'appel de execve.

```
int execl (const char *chemin, const char *arg, ...);
int execl (const char *chemin, const char *arg , ...,
           char * const envp []);
int execv (const char *chemin, char *const argv []);

int execlp (const char * fichier , const char *arg, ...);
int execvp (const char * fichier , char *const argv []);
```

les fonctions **exec ?p** utilisent la **variable d'environnement \$PATH** pour trouver le fichier à charger

Fonctions exec

- ▶ `execl` - arguments en liste de chaînes, `NULL` pour terminer.
- ▶ `execv` - arguments en tableau de chaînes, `NULL` pour terminer.
- ▶ `execle` - idem que `execl`, environnement en plus
- ▶ `execlp` - idem que `execl` mais utilise `PATH` pour trouver l'exécutable.
- ▶ `execvp` - idem que `execv` mais utilise `PATH` pour trouver l'exécutable.
- ▶ ces fonctions renvoient `-1` en cas d'erreur ou ne reviennent pas à l'appelant.

Exemple execve

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
extern char ** environ;
int main (void)
{
    char * argv[] = {"sh", "-c", "set", 0};
    execve("/bin/sh", argv, environ);
    perror("execve");
    exit(0);
}
```

Exemple execl

```
int main(void)
{ int i;
  i=execl("/bin/ls", "ls", " fichier ", "-ail", 0);
  perror("execl");
  exit (0);
}
```

Pourquoi ne pas tester que exec renvoie -1 pour afficher l'erreur ?

Pourquoi répéter ls ?

Exemple execlp

```
int main(void)  
{ execlp("ls","ls","execex2.c","-ail",0);  
  perror("execlp");  
  exit (0);  
}
```

La variable d'environnement PATH détermine quel ls à exécuter

Exemple execvp

```
int main(void)
{
  char * arg [4];
  arg[0]="ls "; arg[1]=" fichier "; arg[2]=" -ail"; arg[3]=0;
  execvp("ls",arg);
  perror("execvp");
  exit (0);
}
```

La variable d'environnement PATH détermine le ls à exécuter

faille execvp, execlp et SUID

Combinées avec des programmes SUID,
les fonctions `exec?p` introduisent une **faille de sécurité**.

La faille est due au fait que ces fonctions se basent sur la variable d'environnement `PATH` de l'utilisateur pour localiser l'exécutable, ici avec les droits d'un autre utilisateur.

La variable d'environnement `PATH` peut être personnalisée par l'utilisateur ...

Exemple faille execlp, exevp et SUID

On désire fournir l'autorisation de lister nos répertoires à d'autres utilisateurs.

Soit le programme `lister.c` :

```
int main(int argc, char *argv[]) {  
    execlp("ls", "ls", NULL);  
}
```

Il est bien SUID

```
>>ls -l lister  
>>-rwsr-xr-x 1 mba prof 12549 27 sept. 16:47 lister
```


Exemple faille execvp, exevp et SUID

- 1 copiez /bin/cat dans votre répertoire et renommez-le **ls**
- 2 ajoutez le chemin . en début du PATH
- 3 le programme cat renommé ls dans . sera choisi plutôt que le vrai ls

```
>>export PATH=.:$PATH  
>>./lister
```

lister, où qu'il se trouve, est un outil pour voir le contenu des fichiers de mba

Qui a fait une erreur ?

Évitez exec...p dans un exécutable S_UID!!

Applications de fork et exec

- ▶ Un process se décharge d'une tâche à l'aide d'un process qu'il crée.
 - Par exemple, un serveur se décharge de la gestion d'un client dès que celui-ci se présente.
 - Deux parties distinctes du programme sont gérées par deux process.
- ▶ Quand un process doit exécuter un autre programme.
 - Par exemple, il peut créer un process qui se remplacera par le programme à exécuter.
 - C'est ce que fait le shell.
- ▶ ...

Système d'exploitation

└ Process

└ naissance et vie

└ Applications de fork et exec

- Un process se décharge d'une tâche à l'aide d'un process qu'il crée.
 - Par exemple, un serveur se décharge de la gestion d'un client dès que celui-ci se présente.
 - Deux parties distinctes du programme sont gérées par deux process.
- Quand un process doit exécuter un autre programme.
 - Par exemple, il peut créer un process qui se remplacera par le programme à exécuter.
 - C'est ce que fait le shell.
- ...

- Le shell utilisera exec pour exécuter une commande externe
- L'écriture `execl("/bin/ls","ls","*",0)` provoquerait une erreur dans ls. Rappelez-vous que le shell interprète wildcards, redirections et pipes. Il le fait avant d'invoquer la fonction `execl` et donc ls.
- Pour ne pas succomber à l'exec, le shell créera un fils avant d'appeler exec.
- C'est le code de ce dernier qui sera finalement écrasé ..

Avancement

- init/systemd
- naissance et vie
- **shell**
- pipe
- signaux
- ordonnancement

Process - shell : un algorithme simple

- 1 Lire une ligne de commande
- 2 "Tokeniser" la ligne de commande
- 3 Si commande externe
- 4 Créer un fils pour cette commande (fork + exec) et Attendre sa fin (wait)
- 5 Retourner au point 1.

Vous disposez de tous les outils pour programmer ce shell.

Système d'exploitation

└ Process

└ shell

└ Process - shell : un algorithme simple

- 1 Lire une ligne de commande
- 2 "Tokeniser" la ligne de commande
- 3 Si commande externe
- 4 Créer un fils pour cette commande (fork + exec) et Attendre sa fin (wait)
- 5 Retourner au point 1.

Vous disposez de tous les outils pour programmer ce shell.

Chaque commande externe génère la création d'un fils
Celui-ci sera écrasé par l'exécutable de la commande

Ce shell ne dispose d'aucune commande interne
Il ne traite pas les wildcards, redirections, pipes

Nous devons encore les programmer. . .

shell simple

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
int main(void)
{
    char ligne [257];
    char *tokens [100];
    printf ("$_"); fgets ( ligne , 256, stdin );
    while (strcmp (ligne , "exit\n")) {
        tokeniser ( ligne , tokens);

        if ( fork () == 0) execvp(tokens[0], tokens);
        wait (0);
        printf ("$_"); fgets ( ligne , 256, stdin );
    }
    exit (0);
}
```

2020-10-05

Système d'exploitation

└ Process

└ shell

└ shell simple

shell simple

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
#include <string.h>
int main(void)
{
    char ligne[257];
    char tokens[100];
    printf("S,>"); fgets(ligne, 256, stdin);
    while (strcmp(ligne, "exit\n") != 0) {
        tokenizer(ligne, tokens);
        if (fork() == 0) execvp(tokens, tokens);
        wait(0);
        printf("S,>"); fgets(ligne, 256, stdin);
    }
    exit(0);
}
```

le shell utilise les fonctions **exec** avec la variable **PATH**

shell - "tokeniser"

Construit un tableau de sous-chaînes terminé par NULL

```
int tokeniser (char *ligne, char *tokens[]){  
    int i=0;  
    tokens[i] = strtok(ligne, " \n");  
    while (tokens[i] != NULL) tokens[++i] = strtok(NULL, " \n");  
    return i;  
}
```

commande invalide - bug ?

Une commande invalide (chemin non trouvé) est un cas d'erreur de l'appel système.



Si vous utilisez le shell ci-dessus avec une commande invalide, il faudra effectuer deux commandes `exit` pour sortir de ce shell

songez à tester une erreur de exec et quitter le fils !

shell - particularités, ...

En vous aidant de la page de manuel de **bash**, explorez la signification pour le shell de :

```
*      // remplacé par la liste des noms de fichiers et dossiers du répertoire  
;      // permet une liste de commandes les unes après les autres (on attend  
(      // lance un sous-shell (ls; ps)  
&      // commande en background idétachée du terminal,(pas d'attente)  
&&     // and : (a && b) si a OK alors b  
||     // or  
>      //  
>>     //  
'cmd' ou $(cmd) // substitution de commande  
...
```

Modifiez le shell simple afin qu'il interprète correctement chacun de ces symboles.

shell - &, batch ou background

Il est possible de placer des processus en "tâche de fond".
Le shell permet de placer un process en foreground (fg) et plusieurs en background (bg).

shell - jobs

Un jeu de commandes permet de gérer cette notion de process batch.

- ▶ commande '**normale**' -> fg.
- ▶ commande suivie de **&** -> bg, le shell n'attend pas sa fin.
- ▶ **Ctrl-Z** fg -> bg et commande suspendue.
- ▶ **jobs** donne l'état des process en bg en les numérotant : 1,2,3, ...
- ▶ **fg** n remplace en fg le job n°n.
- ▶ **bg** n réactive le job n°n qui était suspendu.

shell - opérateurs &&, ||, ...

```
mkdir rep && cd rep  
mkdir rep || echo erreur
```

la programmation de ces opérateurs nécessite d'examiner le status de fin de la commande mkdir

shell - *

wildcard

```
ls *
```

le wildcard * est converti par bash en une liste de fichiers et dossiers passée en paramètre à la commande.

```
for i in *;do if [ -d $i ] ; then rm -r $i; fi ; done
```

Boucle bash pour supprimer des sous-dossiers et leur contenu

shell - redirections

Remplacer stdin, stdout, stderr pour la commande à exécuter

```
...
int nb;
nb = tokeniser (ligne , tokens);

if ( fork()==0) {
    if (strncmp (">",tokens[nb-2],1) == 0){
        int fd = open(tokens[nb-1],O_RDWR|O_CREAT|O_TRUNC,0666);
        dup2 (fd,1);
        close (fd);
        tokens[nb-2] = NULL;
    }
    execvp(tokens [0], tokens);
}
wait (0);
...
```


Système d'exploitation

└ Process

└ shell

└ shell - redirections

Remplacer stdin, stdout, stderr pour la commande à exécuter

```

...
int nb;
nb = tokeniser ( ligne , tokens );

if ( fork() == 0 ) {
    if ( strcmp ( ">" , tokens[nb-2][1] ) == 0 ) {
        int fd = open( tokens[nb-1], O_RDWR | O_CREAT | O_TRUNC, 0666 );
        dup2 ( fd , 1 );
        close ( fd );
        tokens[nb-2] = NULL;
    }
    execvp( tokens[0], tokens );
}
wait ( 0 );
...

```

Avec les appels système open, dup() et dup2()

Nous pouvons programmer les redirections

- ">f - stdout est remplacé par f. Si f existe, il est écrasé.
- "2»f - stderr est remplacé par f. Si f existe, on ajoute à la fin sinon, il est créé.
- "<f - stdin est remplacé par f.

&1 symbolise le handle 1 et &2, le handle 2. 1>f 2>&1

signifie que stderr est le même handle que le handle 1.

cette écriture, à différence de 1>f 2>f, permet de partager l'entrée de la table TDFO.

Questions ?



shell & pipe non nommé

```
who | wc -l  
ls -l | cut -c 1-10 | sort | uniq  
cat /etc/rc.d/rc3.d | grep while
```

- ▶ **simultanéité** des commandes
- ▶ (stdout cmd1) -> | -> (stdin cmd2)
- ▶ communication **half duplex** entre processus parents (cmd1 et cmd2)

pipe - généralités

Un pipe est un tableau partagé appartenant au S.E. accédé via des descripteurs de fichier, un en lecture, l'autre en écriture.

- ▶ Deux **processus parents** se partagent le pipe
- ▶ L'un y écrit
- ▶ L'autre y lit



L'accès à la zone partagée par deux processus nécessite une synchronisation

pipe - synchronisation

Un pipe est un ensemble de deux descripteurs de fichiers et une zone de mémoire partagée.

- ▶ Le S.E. garantit que toute tentative d'écriture dans un pipe plein **bloque** l'écrivain.
- ▶ Le S.E. garantit que toute tentative de lecture d'un pipe vide **bloque** le lecteur.

Le pipe est une application du problème du **producteur-consommateur**.

Système d'exploitation

└ Process

└ pipe

└ pipe - synchronisation

Un pipe est un ensemble de deux descripteurs de fichiers et une zone de mémoire partagée.

- Le S.E. garantit que toute tentative d'écriture dans un pipe plein **bloque** l'écrivain.
- Le S.E. garantit que toute tentative de lecture d'un pipe vide **bloque** le lecteur.

Le pipe est une application du problème du **producteur-consommateur**.

C'est un exemple de synchronisation de processus utilisant plusieurs sémaphores. Nous étudions l'algorithme du producteur-consommateur dans le chapitre dédié aux IPC.

pipe - 2 descripteurs

```
int pipe(int pipefd [2]);
```

Utilisation :

```
int p [2];  
pipe(p);
```

- ▶ l'OS crée deux descripteurs de fichiers `p[0]` et `p[1]` pointant sur un inode de tube.
- ▶ `p[0]` : pour la lecture, `p[1]` : pour l'écriture

Les données écrites sur l'**extrémité écriture** d'un tube sont à lire sur l'**extrémité lecture** du tube.

pipe - 2 descripteurs

L'appel système **pipe** crée deux descripteurs de fichiers :

```
#include <unistd.h>
int main()
{ int p[2];

  pipe(p);

  printf (" descripteurs du pipe : %d, %d\n", p[0], p[1]);
  close (p [0]);
  close (p [1]);
}
```

descripteurs du pipe : 3, 4

Système d'exploitation

└ Process

└ pipe

└ pipe - 2 descripteurs

L'appel système **pipe** crée deux descripteurs de fichiers :

```
#include <unistd.h>
int main()
{ int p[2];

  pipe(p);

  printf ("descripteurs du pipe : %d, %d\n", p[0], p[1]);
  close(p[0]);
  close(p[1]);
}
```

descripteurs du pipe : 3, 4

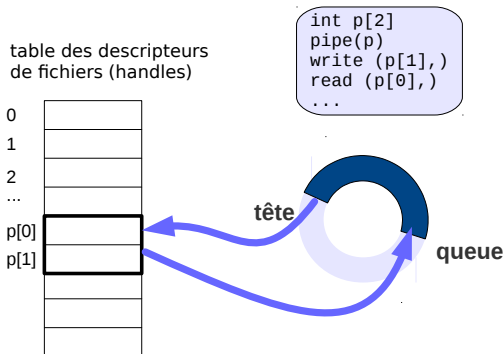
Ces descripteurs peuvent être assimilés à stdin ou stdout via l'appel système dup .

C'est la technique utilisée dans le shell pour programmer les pipes

pipe - pas à pas

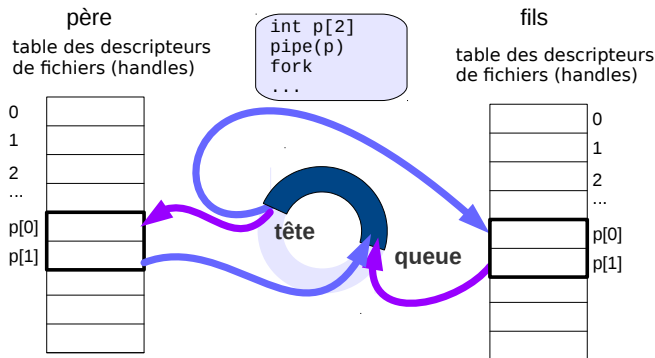
un pipe s'utilise comme une file FIFO :

- ▶ on lit dans `p[0]` // tête
- ▶ on écrit dans `p[1]` // queue



pipe & fork - pas à pas

le partage des descripteurs permet à père et fils de communiquer



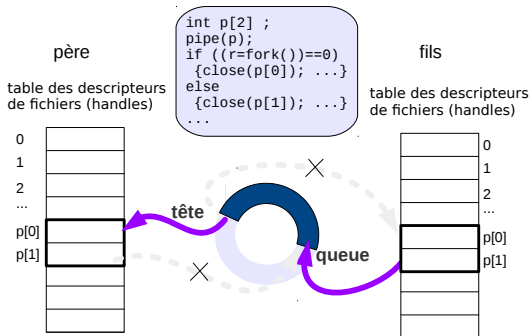
pipe & fork - pas à pas

le partage des descripteurs permet à père et fils de communiquer

```
int p[2];  
pipe(p);  
if (fork()==0)  
{ // fils écrivain , ferme p[0] et utilise p[1] ...  
  exit (0);  
}  
//père lecteur , ferme p[1] et utilise p[0] ...
```

pipe & fork - pas à pas

- ▶ `close(p[0])` chez l'écrivain (ici le fils)
- ▶ `close(p[1])` chez le lecteur (ici le père)



pipe - Exemple père & fils

```
int main()
{ int p[2]; pipe(p); char buf;
  if (fork()==0)
  {
    close(p[1]);           // fils lecteur
    while (read(p[0],&buf,1) >0)
      write (1,&buf,1);
    close(p[0]);
    exit (0);
  }
  close(p[0]);           //père écrivain
  while (read(0,&buf,1) >0)
    write(p[1],&buf,1);
  close(p[1]);
  wait (0);               // remarquez la position du wait
}
```

pipe - Exemple deux fils

```
int main()
{ int p[2]; pipe(p); char buf;
  if (fork()==0)
  { close(p[0]);           // fils1  écrivain
    while (read(0,&buf,1) >0) // clavier
      write(p[1],&buf,1);    // pipe
    close(p[1]); exit(0);
  }
  if (fork()==0)
  { close(p[1]);           // fils2  lecteur
    while (read(p[0],&buf,1) >0) // pipe
      write(1,&buf,1);        // écran
    close(p[0]); exit(0);
  }
  close(p[0]); close(p[1]); // père
  wait(0); wait(0);        // les wait à la fin
}
```

pipe - fin des données ?

Le S.E. garantit que toute tentative de lecture d'un pipe vide **bloque** le process tant qu'il n'y a rien à lire....

- ▶ attente de données du producteur
- ▶ copies de $p[1]$ non fermées

fin de fichier au dernier close de $p[1]$

pipe - blocages ?

pipe, fork, fork

n'oublions aucun close!!!

Système d'exploitation

└ Process

└ pipe

└ pipe - blocages ?

pipe, fork, fork

n'oublions aucun close!!!

- L'appel Système pipe ouvre deux handles.
- Chaque appel à fork dédouble les handles.
- Il y a autant de producteurs que de copies de `p[1]`.
- Un consommateur est bloqué tant qu'il n'y a rien à lire.
- Un consommateur est bloqué tant qu'il y a des producteurs qui ne produisent pas/plus.
- Le lecteur reste bloqué si on oublie un seul `close` de `p[1]`

pipe - blocage par oubli du close

```
int main()
{ int p[2]; pipe(p); char buf;
  if (fork()==0){
    close(p[0]);           // fils écrivain
    while (read(0,&buf,1) >0)
      write(p[1],&buf,1);
    //close(p[1]); // l'oubli de ce close bloque le pere jusqu'à
                    // l'exit du fils

    ...
    exit(0);
  }
  // close(p[1]); // pere lecteur sans ce close, auto blocage du pere
  while (read(p[0],&buf,1) >0)
    write(1,&buf,1);
  close(p[0]);
  wait(0);
}
```

pipe - close

```
ls | wc
```

ls ne sait pas qu'il écrit dans un pipe où sont les close ?

pipe - blocage "close après wait"

```
int main()
{ int p[2]; pipe(p); char buf;
  if (fork()==0)
  {
    close(p[1]);           // fils   lecteur
    while (read(p[0],&buf,1) >0) // read bloquant — jusqu'à quand ?
      write (1,&buf,1);
    close(p[0]); exit (0);
  }

  close(p[0]);           // pere  écrivain
  while (read(0,&buf,1) >0)
    write(p[1],&buf,1);
  wait(0);               // le wait précède le close — interblocage
  close(p[1]);
}
```

pipe - blocage "wait entre les fork"

```
int main()
{ int p[2]; pipe(p); char buf;
  if (fork()==0) // fils1
  { close(p[1]);
    while (read(p[0],&buf,1) >0)
      write (1,&buf,1);
    close(p[0]); exit (0);
  }
  wait (0);           // wait mal place !!
                      // empêche la création du producteur et les close
  if (fork()==0) // fils2
  { close(p[0]);
    while (read(0,&buf,1) >0)
      write(p[1],&buf,1);
    close(p[1]); exit (0);
  }
  close (p[0]); close (p[1]);
  wait(0);
}
```

pipe & shell - dup2

```
ps ux | grep a.out
```

deux process **ps** et **grep** communiquent via un pipe
les appels système pipe et dup2 établissent un canal de communication entre stdout et stdin des deux process créés :

- ▶ ps : stdin inchangé
- ▶ ps : dup2 (p[1],1) : stdout=pipe p[1]
- ▶ grep : dup2 (p[0],0) : stdin=pipe p[0]
- ▶ grep : stdout inchangé

Exemple - ps ux | grep a.out

```
#include <unistd.h>
int main()           // ceci n'est pas un shell
{ int p[2]; pipe(p);
  if (fork()==0) // fils A
  { dup2(p[1],1); close(p[0]); close(p[1]); // écrivain
    execl ("/bin/ps", "ps", "ux", 0);
    perror ("exec"); exit (0);
  }
  if (fork() == 0) // fils B
  { dup2(p[0],0); close(p[0]); close(p[1]); // lecteur
    execl ("/bin/grep", "grep", "a.out", 0);
    perror ("exec"); exit (0);
  }
  close(p[0]); close(p[1]);
  wait(0); wait(0);
}
```


pipes multiples

```
ls -al | tr ' ' '.' | sort | more
```

3 pipes p1, p2, p3

- ▶ ls : stdin inchangé = clavier
- ▶ ls : dup2(p1[1],1) : stdout = pipe p1[1]
- ▶ tr : dup2(p1[0],0) : stdin = pipe p1[0]
- ▶ tr : dup2(p2[1],1) : stdout = pipe p2[1]
- ▶ sort : dup2(p2[0],0) : stdin = pipe p2[0]
- ▶ sort : dup2(p3[1],1) : stdout = pipe p3[1]
- ▶ more : dup2(p3[0],0) : stdin = pipe p3[0]
- ▶ more : stdout inchangé = écran

Exemple - cat < f1 | wc -l > f2

```
#include <fcntl.h>
#include <unistd.h>
int main()           // ceci n'est pas un shell
{ int p[2]; pipe(p);
  if (fork()==0) // fils A
  { int in = open ("f1", O_RDONLY);
    dup2(in,0); close(in); // redirection de 0
    dup2(p[1],1); close(p[0]); close(p[1]);
    execl ("/bin/cat", "cat", 0); exit(1);
  }
  if (fork() == 0) // fils B
  { int out=open("f2",O_WRONLY|O_CREAT|O_TRUNC,0655);
    dup2(out,1); close(out); // redirection de 1
    dup2(p[0],0); close(p[0]); close(p[1]);
    execl ("/usr/bin/wc", "wc", "-l", 0); exit(5);
  }
  close(p[0]); close(p[1]);
  wait(0); wait(0);
}
```

Questions ?



défi ...

programmer ceci :

```
ls | head | wc -l
```

en s'aidant d'une fonction `closeall` :

```
void closeall (int p1 [], int p2 []) { close (p1 [0]); close (p1 [1]);  
                                     close (p2 [0]); close (p2 [1]); }
```

défi ...

et ceci ?

```
ls -al > f1 | wc -l
```

Avancement

- init/systemd
- naissance et vie
- shell
- pipe
- **signaux**
- ordonnancement

signaux

Un signal informe un processus d'un évènement

- ▶ Asynchrone (CTRL-C, SIG_KILL, temporisation, fin d'un fils, ...)
- ▶ Synchrone (erreur mémoire, erreur mathématique, écriture dans tube sans lecteur, ...)

signaux - comportement par défaut

Lorsque un processus reçoit un signal un comportement par défaut s'en suit :

T - Terminer le processus

C - Terminer le processus et créer un fichier core

I - Ignorer le signal

Un

Ct - Continuer le processus s'il est arrêté

S - Arrêter le processus

processus peut définir une fonction qui remplace ce comportement par défaut.

quelques signaux

man 7 signal ...

Nom	n°	Action	commentaire
SIGINT	2	T	Interruption depuis le clavier(Ctrl-C)
SIGFPE	8	C	Erreur arithmétique virgule flottante
SIGKILL	9	T	Signal Kill
SIGSEGV	11	C	Référence mémoire invalide
SIGPIPE	13	T	Écriture dans un tube sans lecteur
SIGALRM	14	T	Temporisation alarm(2) écoulée
SIGTERM	15	T	Signal de fin
SIGUSR1/2	10/12	T	Signal utilisateur 1 et 2
SIGCHLD	17	I	Fils arrêté ou terminé
SIGCONT	18	Ct	Continuer si arrêté
SIGSTOP	19	S	Arrêt du processus

2020-10-05

Système d'exploitation

└ Process

└ signaux

└ quelques signaux

quelques signaux

man 7 signal ...

Nom	n°	Action	commentaire
SIGINT	2	T	Interruption depuis le clavier(Ctrl-C)
SIGFPE	8	C	Erreur arithmétique virgule flottante
SIGKILL	9	T	Signal Kill
SIGSEGV	11	C	Référence mémoire invalide
SIGPIPE	13	T	Écriture dans un tube sans lecteur
SIGALRM	14	T	Temporisation alarm(2) écoulée
SIGTERM	15	T	Signal de fin
SIGUSR1/2	10/12	T	Signal utilisateur 1 et 2
SIGCHLD	17	I	Fils arrêté ou terminé
SIGCONT	18	Ct	Continuer si arrêté
SIGSTOP	19	S	Arrêt du processus

voir aussi

kill -l

signaux - structures

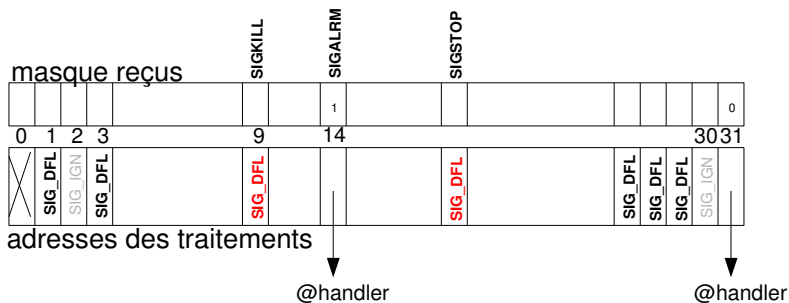
Les signaux sont numérotés 1-31, (32-63 temps réel)

Chaque numéro correspond à un nom : SIGALRM, SIGKILL, SIGINT, ...

A chaque process, on associe deux tableaux dans la table des process :

- ▶ 31 bits (0/1) - masque des signaux reçus non encore traités
- ▶ 31 pointeurs de fonctions (adresses de traitements associés aux signaux)

signaux - structures



signaux

- ▶ Les signaux ont un traitement par défaut (SIG_DFL).
- ▶ Un processus peut modifier le traitement associé à un signal.
- ▶ Font exception les signaux SIGKILL et SIGSTOP.



SIGKILL et SIGSTOP provoquent l'action qui leur est associée par défaut

signaux - envoi

Envoyer un signal au process P ?

kill est une commande et un appel système

kill positionne un bit à 1 dans la table des signaux de P

- Le signal ainsi posté sera traité au moment où P sera l'élus

signaux - rôle de l'OS

L'ordonnanceur

- ▶ Élit un process P, et exécute en priorité l'ensemble des fonctions associées aux signaux reçus par P.
- ▶ Remet à 0 les bits correspondant aux signaux traités.
- ▶ Le process reprend à l'endroit où il était resté.

Les signaux sont traités dans l'ordre des indices du tableau
En présence de plusieurs signaux, l'ordre d'arrivée n'est pas respecté.

sigaction - handlers

L'appel système **sigaction** remplace le traitement (handler) propre à un signal

- ▶ fonction personnalisée , deux signatures pour le traitement
 - **void sa_h (int s)**
 - **void sa_s (int sig, siginfo_t * t, void * old)**

Les traitements des signaux SIGKILL et SIGSTOP ne peuvent être modifiés !

sigaction - associer un traitement

```
int sigaction (int signum, const struct sigaction *act,
               struct sigaction *oldact);

struct sigaction {
    void (*sa_handler) (int);
    void (*sa_sigaction) (int, siginfo_t *, void *); // SA_SIGINFO
    sigset_t sa_mask;
    int sa_flags; // SA_SIGINFO | SA_NOCLDWAIT ...
    void (*sa_restorer) (void); // obsolete
};
```

sigaction - sa_handler

```
void traps(int sig)
{
    printf ("signal %d\n", sig);
}
static struct sigaction act;
...
act.sa_handler=traps;
sigaction (SIGINT,&act,NULL); // tout signal SIGINT sera géré.
...
```

sigaction - sa_sigaction

```
void traps(int sig, siginfo_t * pinfo, void * pucontext)
{
    printf ("signal %d\n", sig);
}
static struct sigaction act;
...
act.sa_sigaction=traps;
act.sa_flags=SA_SIGINFO;
sigaction (SIGINT,&act,NULL); // tout signal SIGINT sera géré.
...
```

sigaction - le traitement

- ▶ Les **paramètres** des traitements sont fournis par le S.E. (n° du signal, ... sur la pile) avant d'appeler la fonction.
- ▶ Un traitement peut faire appel à `exit` qui termine le process ou envoyer lui-même un signal.
- ▶ **sa_handler** peut prendre les valeurs particulières :
 - `SIG_DFL` le traitement par défaut.
 - `SIG_IGN` ignorer le signal.

sigaction - siginfo_t

```
int      si_signo;      /* Numéro de signal */
pid_t     si_pid;       /* PID de l'émetteur */
uid_t     si_uid;       /* UID réel de l'émetteur */
int      si_status;     /* Valeur de sortie */
clock_t   si_utime;     /* Temps utilisateur écoulé */
clock_t   si_stime;     /* Temps système écoulé */
```

Exemple - sigaction

Un process qui trappe tout signal

```
void trapall (int sig, siginfo_t * pinfo, void * pucontext){  
    printf ("reçu le signal=%d\n", pinfo->si_signo);  
}  
struct sigaction act;  
  
int main (...){  
    ...  
  
    act.sa_flags = SA_SIGINFO;  
    act.sa_sigaction = trapall;  
  
    for (noSig=1; noSig<32; noSig++)  
        if ( sigaction (noSig,&act,NULL) < 0)  
            perror ("signal");  
  
    while (1) pause(); // boucle pour signaux non mortels ...  
}
```

Exemple - SIG_DFL

Un process qui trappe SIGINT une seule fois

```
struct sigaction act;
void trap(int sig){
    printf ("reçu le signal=%d\n", sig);
    act.sa_handler= SIG_DFL; // traitement par défaut
    sigaction (SIGINT, &act, NULL);
}
int main (){
    act.sa_handler= trap;
    sigaction (SIGINT, &act, NULL);
    while(1) pause ();
}
```

kill

```
int kill (pid_t pid, int sig)
```

- ▶ **pid** - n° du processus destinataire (qui peut être celui du process même).
- ▶ **sig** - n° du signal (SIGCONT, SIGINT, ...).

sig = 0 => pas d'envoi de signal, mais un status d'erreur (ESRCH) dans **errno** indique si le process est **inexistant**.

kill - droits

- ▶ L'envoi de signaux est autorisé pour l'utilisateur **root**.
- ▶ L'envoi de signaux est autorisé entre processus appartenant au même utilisateur.
- ▶ Le processus **init** est protégé : il peut recevoir uniquement les signaux qu'il traite (SIG_DFL est remplacé par SIG_IGN pour init)

impossible d'envoyer à **init** un signal qu'il ne traite pas

pause

Attendre un signal ...

```
int pause (void);
```

- ▶ Force à s'endormir jusqu'à ce qu'un signal soit reçu
- ▶ Ne revient qu'à la réception d'un signal

Système d'exploitation

└ Process

└ signaux

└ pause

pause

Attendre un signal ...

```
int pause (void);
```

- Force à s'endormir jusqu'à ce qu'un signal soit reçu
- Ne revient qu'à la réception d'un signal

Evitez l'attente active de **while(1)** ; en utilisant **while(1)**
pause() ; à la place

signal & fork

Le traitement d'un signal est préservé par le fork

```
struct sigaction act;
void trap (int s){ printf ("TERMINER PROPREMENT !...\n",
getpid()); /* fflush (stdout);*/ exit (0);}
int main (int argc, char* argv[]){
    act.sa_handler= trap;
    sigaction (SIGTERM, &act, NULL);
    if (fork () == 0 ){
        printf ("fils = [%d]\n", getpid());
        while(1) pause(); exit (0); // suis le fils
    }
    printf ("père = [%d]\n", getpid());
    while (1) pause(); // suis le père
}
```

Le fils traite le signal comme le père.

Question

Quid des fonctions exec ?

Conservent-t-elles le traitement d'un signal à votre avis ?

Qu'en est-il de SIG_IGN ?

CTRL-C et SIGINT

CTRL-C : SIGINT au groupe en foreground.

```
/** dé-commenter signal – vérifier CTRL-C, ps, kill -2 : */
struct sigaction act;
void survie (int s){ printf ("pas mourir.. (%d)\n",getpid());}
int main (int argc, char* argv[]){
    act.sa_handler = survie;
    //sigaction (SIGINT, &act, NULL); // décommenter – hérité après fork
    if (fork () == 0 ){
        //sigaction (SIGINT, &act, NULL); // décommenter
        while(1){ write (1,"1",1); sleep (1); } exit (0);
    }
    if (fork () == 0 ){
        //sigaction (SIGINT, &act, NULL); // décommenter
        while(1){ write (1,"2",1); sleep (1); } exit (0);
    }
    //sigaction (SIGINT, &act, NULL); // décommenter pere -> 2 zombies
    while (1) pause(); wait(0);
}
```

alarm - un réveil

Programmons un réveil :

```
unsigned int alarm(unsigned int nb_sec)
```

Un signal SIGALRM sera envoyé nb_sec secondes plus tard au processus même

Que se passe-t-il si le processus ne traite pas ce signal ?

Exemple - SIGALRM

Recevoir SIGALRM périodiquement pour afficher l'heure :

```
struct sigaction affheure;  
void aff (int s){  
    time_t maintenant = time (&maintenant);  
    struct tm *nu = localtime (&maintenant);  
    printf ("%02d:%02d:%02d\n",nu->tm_hour,nu->tm_min,  
            nu->tm_sec);  
    alarm (2);  
}  
int main (int argc, char* argv[]){  
    affheure.sa_handler= aff;  
    sigaction (SIGALRM, &affheure, NULL);  
    alarm(2); while(1) pause ();  
    wait(0);  
}
```


Exemple - SIGALRM

Avec un process fils

```

struct sigaction actheure; struct sigaction actreveil ;
pid_t fils ; // globale utilisée par le pere
void reveil (int s){ kill ( fils ,SIGUSR1); alarm (2); }
void heure (int s){
    time_t nu = time (&nu);
    struct tm *nu = localtime (&nu);
    printf ("%02d:%02d:%02d\n",nu->tm_hour,nu->tm_min,nu->tm_sec);
}
int main (int argc, char* argv[]){
    actheure.sa_handler= heure; actreveil .sa_handler= reveil ;
    if (( fils =fork()) == 0){
        sigaction (SIGUSR1, &actheure, NULL);
        while (1) pause();    exit  (0);
    }
    sigaction (SIGALRM, &actreveil, NULL);
    alarm(2);
    while(1) pause ();
    wait(0);

```

SIGCHLD - zombies

```
int fin = 0;
void supprimerZombie (int s){ wait(0); fin=1; }
struct sigaction act;
int main (int argc, char* argv[]){
    act.sa_handler = supprimerZombie;
    sigaction (SIGCHLD, &act, NULL);
    if (fork() == 0)
        {exit(0);}

    while (!fin );
    system ("ps ux");
    exit (0);
}
```

traiter le signal SIGCHLD permet d'éliminer les fils zombies

signaux - faiblesses

Conséquences de la représentation par un tableau de bits :

- ▶ occurrences rapprochées du même signal non cumulables
 - nombre de traitements d'un signal \leq nombre de réceptions du signal
- ▶ ordre d'arrivée des signaux inconnu.
 - les traitements ne suivent pas l'ordre d'arrivée des signaux.

Système d'exploitation

└ Process

└ signaux

└ signaux - faiblesses

Conséquences de la représentation par un tableau de bits :

- occurrences rapprochées du même signal non cumulables
 - nombre de traitements d'un signal \ll nombre de réceptions du signal
- ordre d'arrivée des signaux inconnu.
 - les traitements ne suivent pas l'ordre d'arrivée des signaux.

pour pallier à cela, les signaux temps réel seront mémorisés dans une liste

SIGCHLD - zombies et perte de signal

```
struct sigaction suppr;  
void supprimerZombie (int s){ wait(0);}  
int main (int argc, char* argv[]){  
    int i;  
    suppr.sa_handler = supprimerZombie;  
    pid_t pere = getpid();  
    sigaction (SIGCHLD, &suppr, NULL);  
  
    for (i=0; i<20; i++)  
        if (getpid() == pere)  
            if (fork() == 0) {exit(0);}  
    while (1);  
}
```

SIGCHLD - zombies

```
struct sigaction suppr;  
void supprimerZombies (int s){ while (waitpid(-1,NULL, WNOHANG) > 0) ; }  
int main (int argc, char* argv[]){  
    suppr.sa_handler = supprimerZombies;  
    if (fork() == 0) exit(0);  
    if (fork() == 0) exit(0); //deux zombies  
    sleep (1); system ("ps_u_x_|_grep_a.out"); sleep (1);  
    sigaction (SIGCHLD, &suppr, NULL); sleep (4);  
    if (fork() == 0){ sleep (2); exit (0); } // nouveau fils + long  
    system ("ps_u_x_|_grep_a.out"); sleep (4); // qui est la ?  
    system ("ps_u_x_|_grep_a.out"); sleep (2);  
    printf ("tous_les_zombies_sont_éliminés_... _");  
}
```

traiter le signal SIGCHLD permet d'éliminer les fils zombies

Système d'exploitation

└ Process

└ signaux

└ SIGCHLD - zombies

```

struct sigaction sigr;
void supprimerZombies (int x){ while (waitpid(-1,NULL,WNOHANG) > 0) {} }
int main (int argc, char* argv[]){
    sigr.sa_handler = supprimerZombies;
    if (fork() == 0) exit(0);
    if (fork() == 0) //deux zombies
        sleep (1); system ("ps aux | grep a.out"); sleep (1);
    sigaction (SIGCHLD, &sigr, NULL); sleep (4);
    if (fork() == 0){ sleep (2); exit (0); // nouveau fils + long
        system ("ps aux | grep a.out"); sleep (4); // qui est le ?
        system ("ps aux | grep a.out"); sleep (2);
        printf ("tous les zombies sont éliminés...\n");
    }
}

```

traiter le signal SIGCHLD permet d'éliminer les fils zombies

Pourquoi pas **while (wait(0) > 0) ?**

SIGCHLD - zombies

ignorer explicitement le signal SIGCHLD inhibe la création de processus zombies

```
struct sigaction suppr;  
int main (int argc, char* argv[]){  
    // suppr.sa_handler = SIG_IGN;  
    // sigaction (SIGCHLD, &suppr, NULL);  
    if (fork () == 0) exit(0);  
    if (fork () == 0) exit(0);  
    sleep (1);  
    system (ps -o ppid,pid,status,command | grep a.out); // qui est la ?  
}
```


SIGCHLD - zombies

avec le commentaire

```
user0@linux-r91f:/mnt/data/part9/ccode> ./a.out
PPID  PID STATUS COMMAND
1655  1077    - ./a.out
1077  1078    - [a.out] <defunct>
1077  1079    - [a.out] <defunct>
1077  1080    - ps -o ppid,pid,status ,command
1618  1655    - /bin/bash
```

les deux fils sont des zombies

SIGCHLD - zombies

sans le commentaire

```
user0@linux-r91f:/mnt/data/part9/ccode> ./a.out
PPID  PID STATUS COMMAND
1655  1126    - ./a.out
1126  1129    - ps -o ppid,pid,status,command
1618  1655    - /bin/bash
```

pas de création de zombies dans ce cas

SIGUSR1 - exemple affcontinu

affichage du clavier en continu

```
static char ch[2]; static int p[2]; struct sigaction action;  
void lirePipe (int s) {  
    read (p[0], ch, 1); // NE PEUT BLOQUER  
}  
int main (void) {  
    pipe (p); action.sa_handler = lirePipe;  
    if (fork() == 0) {  
        close (p[0]);  
        while (1) {  
            read (0, ch, 2); // BLOQUANT  
            write (p[1], ch, 1);  
            kill (getppid(), SIGUSR1);  
        }  
        exit (0);  
    }  
    close(p[1]); sigaction (SIGUSR1, &action, NULL);  
    while (1) write (1, ch, 1);  
}
```

Questions ?



Avancement

- init/systemd
- naissance et vie
- shell
- pipe
- signaux
- **ordonnancement**

ordonnancement

L'ordonnanceur est la partie du noyau qui décide quel processus prêt va être élu.

Une bonne politique d'ordonnancement doit :

- ▶ tenir compte des temps de réponse (processus interactifs)
- ▶ assurer un débit aux tâches d'arrière-plan
- ▶ éviter la famine

ordonnancement

Les processus(threads) Linux connaissent cinq états de base

- ▶ R (élu ou éligible)
- ▶ S,D (attente d'évènement - interruptible ou non(accès disque))
- ▶ T (arrêté par un signal (SIGSTP,...))
- ▶ Z terminé et dont le père n'a fait ni wait ni exit
- ▶ X marqué pour suppression

politique d'ordonnancement

Le noyau Linux est préemptif (depuis 2.6).

Il utilise le multiplexage temporel (time slicing) et la notion de priorité.

on classe les processus par politique d'ordonnancement

- ▶ processus 'classiques' (CL)
- ▶ processus 'temps réel' (RT)

Il y a toujours un processus à exécuter, un par processeur

Système d'exploitation

└ Process

└ ordonnancement

└ politique d'ordonnancement

Le noyau Linux est préemptif (depuis 2.6).

Il utilise le multiplexage temporel (time slicing) et la notion de priorité.

on classe les processus par politique d'ordonnancement

- processus 'classiques' (CL)
- processus 'temps réel' (RT)

Il y a toujours un processus à exécuter, un par processeur

La commande `ps -clax` montre les processus et leur priorité

CL - Les processus 'classiques'

politique d'ordonnancement SCHED_OTHER

- ▶ Temps partagé.
- ▶ Priorité de base, statique dérivée de la valeur nice (-20 - 19)
- ▶ La priorité dynamique est recalculée en fonction de l'utilisation du temps CPU (Principe du Bonus +5,-5)
- ▶ Basée sur leur gentillesse (nice) et sur la durée moyenne de veille

CL - Les processus 'classiques'

Le processus classiques sont soumis à 'vieillessement' :

- ▶ Une plus longue durée moyenne de veille augmente leur priorité.
- ▶ Les processus interactifs sont ainsi avantagés par rapport aux processus batch

Ainsi on garantit que tous les processus utiliseront le processeur tout en conservant un meilleur temps de réponse pour les processus interactifs.

RT - les processus Temps Réel

Il y a deux 'sous-classes' des processus temps réel :

- ▶ ceux ayant une politique du tourniquet ou 'Round Robin' -> SCHED_RR
- ▶ ceux ayant une politique 'FIFO' -> SCHED_FIFO

RT - priorité

Une valeur de priorité statique `sched_priority` est assignée à chaque processus, et ne peut être modifiée que par l'intermédiaire d'appels systèmes. `sched_priority` est dans l'intervalle 1 à 99.

L'ordonnanceur choisit les processus dans l'ordre des priorités statiques décroissantes.

Système d'exploitation

└ Process

└ ordonnancement

└ RT - priorité

Une valeur de priorité statique `sched_priority` est assignée à chaque processus, et ne peut être modifiée que par l'intermédiaire d'appels systèmes. `sched_priority` est dans

l'intervalle 1 à 99.

L'ordonnanceur choisit les processus dans l'ordre des priorités statiques décroissantes.

la plus grande priorité est donc 1

RT - processus SCHED_FIFO

Dès qu'un processus SCHED_FIFO bascule dans l'état prêt, il aura l'avantage sur un processus RT de priorité inférieure.

Un processus SCHED_FIFO s'exécute jusqu'à ce qu'il ...

- ▶ soit bloqué par une opération d'entrée/sortie, ...
- ▶ soit préempté par un processus de priorité supérieure (appel système)
- ▶ appelle l'appel système sched_yield (ordonnancement coopératif)

RT - processus SCHED_RR

- ▶ ce qui est décrit pour SCHED_FIFO s'applique aussi à SCHED_RR
- ▶ tranche temporelle limitée pour son exécution (quantum)
- ▶ un SCHED_RR qui a utilisé tout son quantum sera placé à la fin de la liste de sa priorité.
- ▶ un SCHED_RR préempté par un processus de priorité supérieure, reste en tête de liste et terminera sa tranche de temps plus tard

Système d'exploitation

└ Process

└ ordonnancement

└ RT - processus SCHED_RR

- ce qui est décrit pour SCHED_FIFO s'applique aussi à SCHED_RR
- tranche temporelle limitée pour son exécution (quantum)
- un SCHED_RR qui a utilisé tout son quantum sera placé à la fin de la liste de sa priorité.
- un SCHED_RR préempté par un processus de priorité supérieure, reste en tête de liste et terminera sa tranche de temps plus tard

il y a une liste de processus prêts pour chaque valeur de priorité
la tranche de temps est liée à la valeur nice.

nice	durée
-20	800ms
0	100ms
19	5ms

après fork ?

Les processus fils héritent de la politique d'ordonnancement et des paramètres associés lors d'un fork.

type d'ordonnancement et priorité statique

lire - modifier, le type d'ordonnancement associé à un processus :

- ▶ `policy sched_getscheduler(pid)`
- ▶ `sched_setscheduler(pid, policy, param)`
 - `pid` : le numéro du processus visé
 - `policy` : la politique d'ordonnancement : `SCHED_RR`, `SCHED_FIFO`, `SCHED_OTHER`
 - `param` : les paramètres (actuellement, seulement la priorité statique)

autres paramètres

```
int sched_get_priority_max(int policy);  
int sched_rr_get_interval(pid_t pid, struct timespec * tp);
```

- ▶ priorité minimale et maximale
- ▶ quantum (en lecture seulement)
- ▶ **struct timespec** deux entiers de 32 bits
tv_sec secondes et tv_nsec nanosecondes -> plus de
100 ans à la nanoseconde près.

changer de politique

```
void f(void) { int  b,i;
    i=getpid(); b=sched_getscheduler(i);
    switch (b) { case SCHED_FIFO : printf("%d:_FIFO_\n",i); break ;
                case SCHED_RR  : printf ("%d:_RR_\n",i); break ;
                case SCHED_OTHER : printf("%d:_OTHER_\n",i); break ;
                } }

int main ()
{ struct sched_param param;
  f();
  param.sched_priority = sched_get_priority_min(SCHED_FIFO);
  sched_setscheduler (getpid(), SCHED_FIFO, &param) ;
  sched_getparam(0, &param);
  printf (" priorité _:_%d\n", param.sched_priority);
  f(); exit (0);
}
```

changer de politique

Cela donne

16331 : OTHER

priorité : 1

16331 : FIFO

Pour un processus RT, la priorité 1 est la priorité la plus haute, 99 la plus basse

durée du quantum

```
int main () {  
    struct sched_param sp;  
    struct timespec ts;  
  
    sp.sched_priority = 1;  
    sched_setscheduler (0, SCHED_RR, &sp);  
    sched_rr_get_interval (0,&ts);  
  
    printf ("Quantum: %ds, %dns\n", ts.tv_sec, ts.tv_nsec);  
    exit (0);  
}
```

durée du quantum

cela donne

Quantum : 0s, 100000000ns

RR - utilise un quantum de 100ms

OTHER

ordonnancement par tourniquet

utilise une priorité dynamique [-20, 19]

```
int main (int argc, char * argv [])
{
    struct sched_param param;
    struct timespec ts;
    if ( setpriority (PRIO_PROCESS, getpid(), -20) < 0 )
        perror (" setpriority ");
    sched_getparam(0, &param);
    switch (sched_getscheduler (getpid())) {
        case SCHED_OTHER : printf("OTHER_\n"); break ;
    }
    printf (" priorité _:%d_\n", param.sched_priority,
            getpriority (PRIO_PROCESS, getpid()));
    sched_rr_get_interval (0,&ts);
    printf ("Quantum_:%ds,_%dns\n", ts.tv_sec, ts.tv_nsec);
}
```

priorité d'ordonnancement

cela donne

OTHER : priorité : 0 -20

Quantum : 0s, 17000000ns

-20 est la plus grande priorité





le quantum est de 17 ms dans ce cas

ordonnancement

- ▶ L'appel système `sched_yield` est un appel à l'ordonnanceur.
- ▶ Il sert à laisser la chance à d'autres d'être élus
- ▶ Les problèmes de famine sont évités en doublant le tableau de liste de processus (actifs et expirés). Tous les processus auront leur chance.

ordonnancement

- ▶ La modification de politique d'ordonnancement est réservée à l'administrateur.
- ▶ "Diminuer" la valeur de la priorité nécessite des droits d'administration.

-  Modern Operating Systems Fourth edition - Andrew Tanenbaum, Herbert Bos - Pearson Education
-  Advanced Programming in the UNIX Environment Third Edition - W.Richard Stevens, Stephen A. Rago - Addison Wesley (2014)
-  Programmation Système en C sous Linux 2ième édition - Christophe Blaess - Eyrolles (2005)
-  Intel 64 and IA-32 Architectures Software Developer's Manual - December (2011) (pour toutes les images du chapitre mémoire)

remerciements

merci à P.Bettens et M.Codutti pour la mise en page

Crédits

Ces slides sont le support pour la présentation orale des activités d'apprentissage **SYSIR3** et **SYSG4** à HE2B-ÉSI

Crédits

La distribution opensuse
du système d'exploitation **GNU Linux**.

LaTeX/Beamer comme système d'édition.

GNU make, rubber, pdfnup, ... pour les petites tâches.

Images et icônes

deviantart, flickr, The Noun Project 