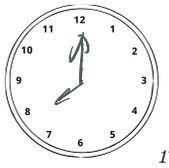


Processus

Thèmes abordés

— Notions de base sur la mise en œuvre et la gestion des processus
— Ordonnancement des processus
— Présentation de l'API processus Unix

1 Questions

1. Comment les interruptions rendent-elles possible la multiprogrammation ?
2. Sur un système biprocesseur, est-il concevable qu'un processus donné puisse s'exécuter sur l'un puis l'autre des deux processeurs alternativement ? Justifiez votre réponse en expliquant le mécanisme de commutation de processus.
3. Précisez grâce à quel mécanisme un processus peut appeler les primitives du noyau mais ne peut pas se brancher n'importe où dans le code du noyau ?

2 Ordonnancement des processus

2.1 Problématique (rappel)

Cette section est consacrée à la comparaison et à la réalisation de différentes politiques d'ordonnancement à **court terme**, c'est-à-dire aux politiques d'allocation du (ou des) processeur(s) entre processus prêts [1]. Les politiques d'ordonnancement se distinguent par les éléments qu'elles intègrent pour déterminer le (prochain) processus actif (*élu*) :

— elles peuvent ou non se baser sur une **priorité** associée à chacun des processus ;
— elles peuvent **préempter** (réquisitionner) le processeur pour attribuer à un nouveau processus élu, ou bien laisser au processus élu l'initiative de rendre le processeur ;
— elles peuvent ou non déterminer le processus élu en fonction de **contraintes de temps réel**.

Les politiques d'ordonnancement peuvent être évaluées selon des critères traduisant différents besoins des applications :

— temps de service : temps moyen d'attente ;
— temps de réponse : garantie d'un délai d'attente maximum.
— équité (absence de famine) : tout processus prêt finit-il par obtenir le processeur ?

2.2 Exemples

Indiquez les caractéristiques et le niveau de réalisation des différents critères pour les politiques suivantes :

— FIFO : le processeur est alloué aux différentes tâches suivant l'ordre chronologique des demandes d'allocation.
— SJF (*Shortest Job First*) : chaque processus annonce sa durée d'utilisation du processseur. Le processeur est alloué au processus ayant la plus courte durée d'utilisation.
— Tourniquet (*Round-Robin*) : le processeur est alloué par quantum, à tout de rôle à chacun des processus
— EDF : (*Earliest Deadline First*) : chaque processus annonce une date maximum de terminaison (échéance). Le processeur est alloué au processus ayant l'échéance la plus proche.

Question. *Les notions d'équité (absence de famine) et de priorité sont a priori antagonistes. Comment introduire cependant une forme d'équité dans un système avec priorités ?*

1. Un processus *prêt* est un processus qui n'est pas bloqué en attente d'une ressource ou d'un événement de synchronisation. Il peut donc progresser immédiatement, dès lors qu'il obtient le processeur.

2.3 Mise en œuvre du tourniquet

Programmer (en pseudo-code) l'algorithme de principe du tourniquet.

On supposera que les processus sont identifiés par un entier (qui correspond à l'indice de leur descripteur dans la table des processus). On supposera de plus que l'on dispose

- d'une implémentation du type file
- d'une opération `commuter(courant : id_proc, nouveau : id_proc)` qui sauvegarde le contexte d'exécution du processus actif en cours (d'identifiant `courant`) pour installer/restaurer le contexte du processus d'identifiant `nouveau`, qui devient le nouvel actif.
- d'une opération `cadencer_horloge(délai : entier)`, qui programme l'envoi d'un signal d'horloge (d'identifiant `IT_HORLOGE`) toutes les `délai` ms après l'exécution de cette opération.

Donner

- le code du traitant associé à `IT_HORLOGE`,
- ainsi que le code d'initialisation situé dans le programme principal de l'ordonnanceur. Pour ce dernier, et pour être complet, vous pouvez supposer que vous disposez d'une opération `associer(sgn : entier ; traitant : fonction())`, qui permet de demander l'exécution de la fonction d'identifiant `traitant`, chaque fois que le signal `sgn` est reçu.

Le code que vous venez d'écrire est un exemple caractéristique de *programmation événementielle* (ou *réactive*).

2.4 Ordonnancement par partage équitable

Les politiques précédentes ne conviennent pas nécessairement à toutes les situations :

1. chaque politique d'ordonnancement est adaptée pour certains critères (équité, ou efficacité...) mais aucune ne l'est pour tous les critères. Il semble donc inadéquat d'appliquer un traitement indifférencié et critères identiques pour tous les processus, alors que les comportements et les besoins peuvent varier selon le profil des applications.
2. La disponibilité du processeur pour un processus donné peut être difficile à prédire, car elle peut dépendre du comportement et des caractéristiques des autres processus. Cet inconvénient peut être critique dès lors que les applications ont des contraintes de réactivité, de temps-réel.
3. La gestion des ensembles de processus en attente (listes...) peut s'avérer gourmande en temps de calcul (même si des optimisations sont possibles, et utilisées).

Les stratégies d'allocation multiniveaux (ou *hiérarchiques*) résolvent la première de ces restrictions. Leur principe est de regrouper les processus par catégories. Les processus d'une catégorie donnée ont un profil similaire vis à vis de l'utilisation du processeur : tâches de fond, tâches interactives, etc... Chaque catégorie a une politique d'allocation spécifique, adaptée au profil de ses processus. Par ailleurs, l'ordonnanceur alloue le processeur aux différentes catégories, en suivant une politique d'allocation globale, entre catégories.

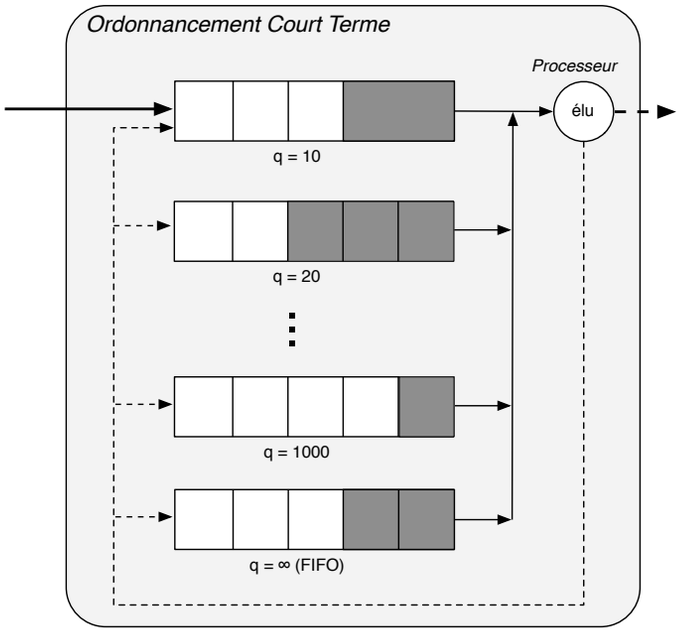
La file multiniveaux est un exemple classique de cette classe de politiques d'ordonnancement. Cette politique considère plusieurs files, rangées par ordre de priorité. Chaque file est gérée selon une politique de tourniquet. Le quantum est lié à la priorité de la file : la file la plus prioritaire a le plus petit quantum, tandis que la file la moins prioritaire est un simple FIFO. L'ordonnancement global entre files suit une simple priorité : pour qu'un processus au niveau i soit élu, il faut qu'il n'y ait aucun processus prêt dans les files plus prioritaires que i. Un processus de niveau i actif est préempté en cas d'arrivée ou de reprise d'un processus plus prioritaire.

Une dernière caractéristique intéressante de la file multiniveaux est qu'il s'agit d'une politique **adaptative** : un processus de niveau i peut passer dans une file moins prioritaire dans le cas où il épuise son quantum, ou au contraire passer dans une file plus prioritaire s'il ne l'épuise pas. L'idée est qu'un processus finira par se ranger dans la file dont le quantum correspond le mieux à la durée de sa période de calcul moyenne.

Pour une stratégie d'allocation multiniveaux, si l'on considère l'allocation du processeur entre les différentes catégories, il est fréquent que certaines catégories correspondent à des profils d'applications demandant la satisfaction de prévisibilité, ou de contraintes temporelles (applications interactives, temps-réel...).

```
Program Principale
{
cadencee_horloge(10);
associer(IT_HWR20GE, ordonnance());
file F;
```

```
ordonnance(),
    F.ajouter(id_courant)
id_prog id = F.defiler();
```



Les stratégies d'allocation par partage équitable visent à répondre à des besoins de prévisibilité en termes d'allocation du processeur. Nous allons étudier deux stratégies de base, appliquées aux processus, utilisées en particulier par Linux. Ces stratégies peuvent être étendues simplement pour la gestion de l'allocation entre catégories dans le cadre de l'allocation multiniveaux.

La question essentielle *(à laquelle vous pouvez prendre un peu de temps pour réfléchir...)* est donc :

Comment offrir à un processus donné une garantie d'accès au processeur, indépendamment de l'utilisation qu'en font les autres processus ?
Question subsidiaire : comment assurer cette garantie d'accès de manière efficace ?

L'algorithme de la loterie fournit une réponse originale et élégante à ces deux questions. Son principe est de mettre en circulation un nombre fixe de tickets, et de répartir ces tickets entre les différents processus. Le processeur est alloué par quantums de temps. A l'échéance du quantum, l'ordonnanceur tire un ticket aléatoirement. Le processus dont le ticket a été tiré est le nouveau processus élu.

Remarques :
- la répartition n'a pas à être nécessairement équilibrée, ce qui permet de réaliser une forme de priorité
- le nombre fixe de tickets permet de réaliser une forme de *contrôle d'admission*, ce qui est un outil de régulation important en cas de forte charge du système.

Le tirage aléatoire est généralement implanté de manière efficace. Le point important du point de vue des performances est donc d'assurer une distribution et une recherche efficace des tickets. Le principe est simple :
- les tickets sont numérotés de 0 à Max ;
- le descripteur de processus comporte le nombre de tickets alloués au processus ;
- les tickets attribués aux processus sont assimilés à des plages de valeurs successives : l'espace des valeurs [0 .. Max] est ainsi virtuellement partitionné en intervalles successifs, et un processus est (virtuellement) associé à chaque intervalle ;
  Exemple : 100 tickets sont émis. La liste des processus comporte (dans l'ordre) 3 processus : A avec 20 tickets, B avec 50 tickets, et C avec 30 tickets. On peut alors considérer que A détient les tickets 0 à 19, B détient les tickets 20 à 69, et C détient les tickets 70 à 99.
- lorsque le nombre N est tiré, la liste des processus est parcourue, en cumulant le nombre de tickets attribués. Le premier processus pour lequel le cumul est supérieur à N a le ticket gagnant.

Question 1. *Adapter l'algorithme pour améliorer l'efficacité de la boucle de recherche (autrement dit :*

minimiser le nombre d'itérations). Montrer que cette adaptation n'altère pas les chances d'accès au processeur de chacun des processus.

Le recours aux choix aléatoires permet d'allouer une fraction de temps processeur à chaque processus, indépendamment de l'utilisation qu'en font les autres processus, et ce de manière effective et efficace. Cependant, dans certaines situations, comme dans le cas de systèmes critiques, le caractère aléatoire de l'allocation peut s'avérer problématique. Des adaptations déterministes de l'algorithme de la loterie ont ainsi été élaborées, comme l'ordonnancement par pas (*stride scheduling*). L'ordonnanceur des versions récentes du système Linux fonctionne sur cette base.

L'ordonnancement par pas conserve le principe d'émission et d'attribution de tickets aux processus. Chaque processus est alors caractérisé par un **pas**, inversement proportionnel à son nombre de tickets. Par ailleurs, pour chaque processus, les pas sont cumulés chaque fois que le processus est élu. Le cumul des pas d'un processus représente donc la comptabilisation du temps processeur consommé par ce processus. Plus un processus a de tickets, plus la « facturation » d'un pas sera favorable.

L'algorithme d'ordonnancement proprement dit consiste simplement, à l'échéance du quantum, à parcourir la liste des processus pour trouver (et élire) le processus ayant le cumul de pas minimum (en cas d'égalité, l'identifiant de processus est utilisé pour départager et rester déterministe)

Question 2. *Est ce vraiment mieux que la loterie ? Autrement dit : quel est le coût de l'introduction du déterminisme ?*

3 API processus Unix

Résumé et objectifs de la présentation L'interface programmatique (API, Application Programming Interface) de gestion des processus proposée par UNIX repose sur un modèle et un patron de conception élégants et efficaces, bien que surprenants au premier abord. Les opérations de base de l'interface programmatique sont tout d'abord présentées de manière simplifiée. Elles sont ensuite justifiées et illustrées à travers la présentation du protocole d'usage qui leur est associé. Enfin, quelques éléments (rassurants) sont donnés quant à l'efficacité de la mise en œuvre de l'API.

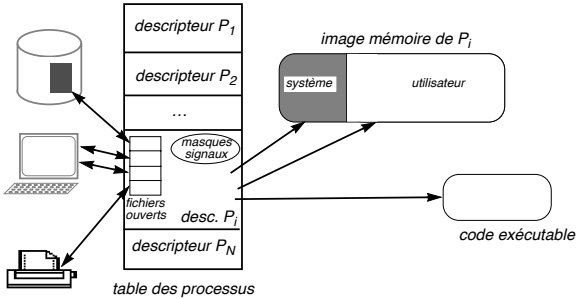
Déroulement

- Introduction : descripteur de processus UNIX (planche 3)
- Gestion des processus : `fork`, `exec`
  - fork : principe = clonage ; (planches 5,4)
  - exercice 2.2.5 (3 forks)
  - discrimination par la valeur de retour
  - premier schéma d'usage :
    `id_fils = fork() ; si (id_fils=0) alors code_fils() sinon code_père() ;`
    (planche 6)
  - recouvrement (exec...) : motivation = modularité + programmation du contexte d'exécution d'un programme avant de lancer ce programme.
- Synchronisation père/fils : `wait`, `exit`
  - planches 10-13
  - schéma du shell.
- Autres : `getpid`...
- *Elégance* ***et*** *efficacité* : copie sur écriture (*c.o.w : copy on write*). Le clonage de l'image mémoire réalisé par le fork() est en réalité virtuel : le père et le fils partagent en lecture la même image mémoire, et la duplication effective n'a lieu qu'en cas de modification (accès en écriture), et ne porte dans ce cas que sur le fragment de mémoire (la page) modifié. Ce mécanisme est une bonne illustration du principe d'évaluation paresseuse.

Interface programmatique de gestion des processus UNIX

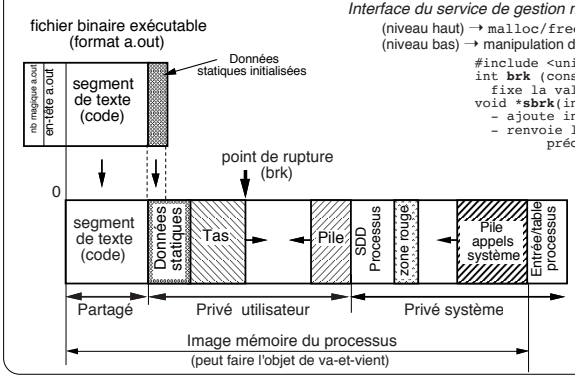
Plan

Descripteur de processus (vue système)



1 – Les processus dans le système UNIX

Image mémoire (vue programmeur)



Interface du service de gestion mémoire UNIX :

(niveau haut) → `malloc/free (stdlib.h)`

(niveau bas) → manipulation de `brk` :

```
#include <unistd.h>
int brk (const void *ptr)
  fixe la valeur de brk
void *sbrk(int increment)
  - ajoute increment à brk
  - renvoie la valeur
      précédente de brk
```


2 – Opérations sur les processus

Création

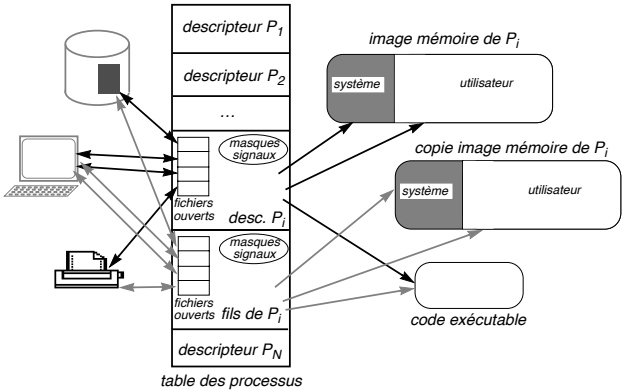
`pid_t fork();`

crée un (descripteur de) processus « clone » du processus appelant (processus *père*).

- Le processus créé (*fils*) hérite
  - ◊ du même code,
  - ◊ d'une copie de la zone de données du père,
    amélioration/optimisation : copie limitée aux pages modifiées après **`fork()`** (c.o.w))
  - ◊ de l'environnement,
  - ◊ de la priorité,
  - ◊ des descripteurs de fichiers ouverts,
  - ◊ du traitement des signaux.
- La valeur de retour de `fork()` est cependant différente :
  - ◊ `0` pour le fils
  - ◊ l'identifiant (pid) du fils pour le père

Intérêt

le père peut définir et transmettre simplement et précisément le contexte d'exécution du fils



L'ordonnancement des processus père et fils est indéterminé

Exemple

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void) {
    int pid = fork(); /* père et fils */
    printf("Valeur de fork = %d ",pid);
    printf("processus %d de pere %d\n",getpid(),getppid());
    if ( pid == 0 ) { /* processus fils */ printf("fin du fils\n"); }
    else { /* processus pere */ printf("fin du processus pere\n"); }
    return 0;
}
```

Résultats

```
prompt%creer_processus
Valeur de fork = 0 processus 434 de pere 433
fin du fils
Valeur de fork = 434 processus 433 de pere 364
fin du processus pere
prompt%

prompt%creer_processus
Valeur de fork = 0 processus 397 de pere 396
Valeur de fork = 397 processus 396 de pere 364
fin du processus pere
prompt%fin du fils

prompt%creer_processus
Valeur de fork = 440 processus 439 de pere 364
fin du processus pere
prompt%Valeur de fork = 0 processus 440 de pere 1
fin du fils
```


Destruction

```
void exit(int n);
```

termine le processus courant avec le code de retour `n` (en général, `0` si retour normal)

Chargement d'un code à exécuter : recouvrement

```
int execl(char *chemin, char *arg0, char *arg1, char *arg2,…, char *argn, NULL);
int execlp(char *chemin, char *arg0, char *arg1, char *arg2,…, char *argn, NULL);
int execle(char *chemin, char *arg0, char *arg1, …, char *argn, NULL, char *env[]);
int execv(char *chemin, char *argv[]);
int execvp(char *chemin, char *argv[]);
int execve(char *chemin, char *argv[],char *env[]);
```

- décodage
  - ◊ l/v : liste/tableau
  - ◊ p : utilisation de PATH
  - ◊ e : passage de l'environnement
- après recouvrement
  - ◊ une nouvelle image mémoire est allouée
  - ◊ les signaux non ignorés sont associés à leur traitant par défaut
  - ◊ les descripteurs restent ouverts, sauf ceux indiqués par fcntl (FD_CLOEXEC)
  - ◊ les autres attributs sont conservés

Identification

- **`pid_t getpid()`** : pid de l'appelant
- **`pid_t getppid()`** : pid du père de l'appelant
- **`uid_t getuid()`** : id de l'utilisateur ayant lancé le processus
- **`uid_t geteuid()`** : id de l'utilisateur effectif
- **`uid_t getgid()`** : id du groupe de l'utilisateur ayant lancé le processus
- **`uid_t getegid()`** : id du groupe effectif
- **`int setuid(uid_t u)`**
- **`int setgid(gid_t g)`**

Synchronisation

Attente d'un laps de temps

`int sleep(int n)`

suspend l'exécution du processus appelant pour n secondes (au moins)

Envoi/attente d'événement

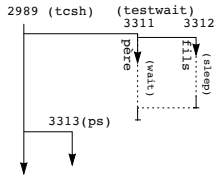
`kill/pause` : vu plus loin

Exemple : Programme testwait.c

```
#include <sys/types.h>
#include <sys/wait.h>
int main() {
  if (fork() != 0) { /* père */
    int statut; pid_t fils);
    printf("je suis le père %d, j'attends mon fils\n", getpid());
    fils = wait(&statut);
    if (WIFEXITED(statut)) {
        printf("%d : mon fils %d s'est terminé avec le code %d\n",
            getpid(), fils, WEXITSTATUS(statut); }
     exit(0);
   } else { /* fils */
    printf("je suis le fils, mon PID est %d\n", getpid());
    sleep(2)  /* blocage pendant 2 secondes */
    printf("fin du fils\n"); exit(1);
  }}
```

Exécution

```
<mozart> ./testwait
je suis le fils, mon PID est 3312
je suis le père 3311, j'attends mon fils
fin du fils
3311: mon fils 3312 s'est terminé avec le code 1
<mozart> ps
 PID TTY          TIME CMD
2989 pts/0    00:00:00 tcsh
3313 pts/0    00:00:00 ps
<mozart>
```



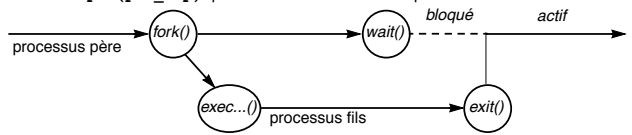
Coordination père/fils

`#include <sys/wait.h>`

`pid_t wait(int *n);`

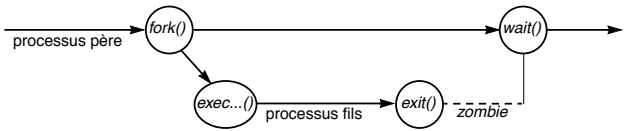
suspend l'appelant jusqu'à la terminaison d'un fils, et retourne le pid du fils terminé

- si le fils s'est terminé par `exit()`, le deuxième octet de `*n` est le code renvoyé par `exit()`
- si le fils s'est terminé suite à la réception d'un signal, le premier octet de `*n` est le numéro du signal (+128 si un fichier `core` a été engendré)
- des macros (`WIFSIGNALED, WIFEXITED, WTERMSIG, WEXITSTATUS`…) permettent de manipuler plus simplement `*n`
- comme la plupart des appels système, si le père reçoit un signal alors qu'il attend sur `wait`
  - ◊ si un traitant a été défini, le signal est traité et le père reste en attente
  - ◊ sinon, `wait` se termine et renvoie -1
- un appel à `wait` alors qu'aucun fils (zombie ou actif) n'existe, se termine aussitôt et renvoie -1
- **`int waitpid(pid_t p)`** permet d'attendre la fin d'un processus fils donné



Processus zombis

Tant que son père n'a pas pris connaissance de sa terminaison par `wait` ou `waitpid`, un processus terminé reste dans un état dit zombi.



- *Intérêt* : signification du wait : attente de la terminaison d'un fils, indépendamment de la durée d'exécution de ce fils
- *Inconvénient* : un processus zombi ne peut plus s'exécuter, mais consomme encore des ressources (tables)
  → éviter de conserver des processus dans cet état.
  → un traitement standard est d'exécuter **`wait`** dans un traitant associé au signal `SIGCHLD`

Processus orphelins

- lorsqu'un processus se termine, ses fils (dits orphelins) sont rattachés au processus 1 (`init`)
- le processus `init` élimine les zombies en appelant systématiquement `wait`

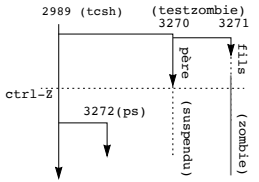
Observation des zombies : exemple

Programme testzombie.c

```
int main() {
  if (fork() != 0) {
    printf("je suis le père, mon PID est %d\n", getpid());
    while (1) ; /* boucle sans fin sans attendre le fils */
  } else {
    printf("je suis le fils, mon PID est %d\n", getpid());
    sleep(2)  /* blocage pendant 2 secondes */
    printf("fin du fils\n");
    exit(0);
  }
}
```

Exécution

```
<mozart> gcc -o testzombie testzombie.c
<mozart> ./testzombie
je suis le fils, mon PID est 3271
je suis le père, mon PID est 3270
fin du fils
==>frappe de <control-Z> (suspendre)
Suspended
<mozart> ps
 PID TTY          TIME CMD
2989 pts/0    00:00:00 tcsh
3270 pts/0    00:00:03 testzombie
3271 pts/0    00:00:00 testzombie <defunct>
3272 pts/0    00:00:00 ps
<mozart>
```



Signaux

Thèmes traités

- Notions de base et mise en œuvre des signaux UNIX.
- Opérations essentiellles de l'API signaux Unix, protocole d'usage.
- Programmation d'horloges
- Sauvegarde et restauration de points de reprise (API et protocole d'usage)

1 Questions

1. Qu'apporte le mécanisme d'interruptions à la gestion et la supervision des E/S par le système d'exploitation ? Est ce que l'utilisation de ce mécanisme est pertinente dans tous les cas ? Pourquoi ?
2. Du point de vue de l'application et du point de vue des mécanismes mis en jeu, quelles sont les différences entre la réception d'un signal et l'appel d'une procédure ?

2 Exercice

Écrire un code C qui imprime le numéro de tout signal reçu (signal émis depuis le terminal ou depuis un autre processus). Ce code indiquera toutes les 3 secondes qu'il est toujours actif et en attente de signaux. Au bout de 5 signaux reçus ou 27 secondes il devra s'arrêter. Le décompte du temps s'appuiera sur la programmation de l'envoi du signal `SIGALRM` par l'horloge de temps physique.

1. dans un premier temps, on utilisera la fonction `alarm` pour l'envoi de `SIGALRM`, et on ne distinguera pas la source des signaux `SIGALRM`.
2. traiter la question précédente en utilisant les timers.
3. les signaux `SIGALRM` peuvent résulter d'appels à `kill`, et non de la programmation de l'horloge. Compléter le code précédent pour réduire (ou éliminer) cette possibilité.

3 Déroulement

- présentation du mécanisme de signaux, mise en œuvre du masque, signaux masqués et pendants (planches 3-5 des transparents d'accompagnement)
- revue rapide des opérations de base (`signal/sigaction, kill, pause,alarm`) (planches 6,8 à 10)
  schéma de base de mise en place d'un traitant (planche 7)
  Exemple : retour sur l'ordonnanceur, avec le traitant de `SIGALRM`
- exercice 2.5.3 sans timers
- revue rapide des opérations de manipulation des masques de signaux (`sigsuspend/sigprocmask/sigpending`) (planche 10)
- présentation des timers (planche 14). Exercice 2.5.3 avec timers
- présentation de `setjmp/longjmp`.
  Illustration avec le principe de mise en œuvre des exceptions (planche 13)

2. Exercice :

<1>

```
int nb_sig = 0;
int nb_second = 0;
int main() {
    for (int i = 0; i < NSIG; i++) {
        signal(i, message);
    }
    signal(SIGALRM, sig_alarm);
    alarm(3);
    while (nb_second < 2 || nb_sig < 5) {
        pause();
    }
    return 0;
}
```

```
int message(int sig) {
    printf("nb recu %d.\n", sig);
    nb_sig++;
}

int sig_alarm(int sig) {
    nb_second += 3;
    alarm(3);
    printf("Actif.\n");
}
```

<2>

```
int nb_sig = 0;
int nb_second = 0;
struct timeval duree;
int main() {
    for (int i = 0; i < NSIG; i++) {
        signal(i, message);
    }
    signal(SIGALRM, sig_alarm);
    duree.it_interval.tv_sec = 3;
    duree.it_interval.tv_usec = 0;
    duree.it_value.tv_sec = 3;
    duree.it_value.tv_usec = 0;
    setitimer(ITIMER_REAL, duree, null);
    while (nb_second < 2 || nb_sig < 5) {
        pause();
    }
```

```
int message(int sig) {
    printf("nb recu %d.\n", sig);
    nb_sig++;
}

int sig_alarm(int sig) {
    getitimer(ITIMER_REAL,
        &duree);
    if (duree.it_interval.tv_sec == 2
        && duree.it_interval.tv_usec ==
        10^6 ± 50 ms) {
        nb_second += 3;
    }
    printf("Actif.\n");
}
```

Interface de communication asynchrone : signaux UNIX

Plan

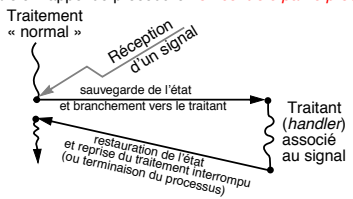
1 – La communication entre processus

UNIX fournit un ensemble de services permettant à un processus de communiquer avec son environnement ou avec d'autres processus, selon diverses formes et modalités :

- Communication asynchrone d'événements (signaux) :
  - ◊ **Schéma publier/s'abonner**
    - le récepteur manifeste son intérêt pour l'occurrence d'événements à venir/de données à produire en *s'abonnant*
    - chaque nouvelle occurrence est *transmise par l'émetteur* au récepteur
    - en réaction à cette transmission, le récepteur interrompt (provisoirement) son comportement courant pour traiter l'événement
  - ◊ Mécanisme analogue aux interruptions matérielles
  - ◊ Utilisation : communication d'événements asynchrones par le système (erreurs, interactions avec le matériel) ou l'utilisateur
- Communication synchrone
  - ◊ *le récepteur décide de l'instant de la réception*
  - ◊ communication explicite
    - flots d'E/S : fichiers, tubes
    - files de messages (IPC System V et POSIX)
    - sockets: canaux virtuels entre processus quelconques, éventuellement distants (vu plus tard)
  - ◊ communication implicite : mémoire (virtuelle) partagée et sémaphores (vus plus tard)

2 – Signaux : utilisation et réalisation

- un signal traduit l'occurrence d'un événement « observé » par l'environnement du processus qui reçoit le signal :
  - ◊ erreur liée à l'exécution du processus récepteur (accès errroné…)
  - ◊ certains événements matériels (frappe de caratères particuliers…) transmis par le système
  - ◊ événements applicatifs transmis par d'autres processus utilisateurs
- fonctionnement analogue au traitement des interruptions matérielles
  - ◊ différence : un signal ne correspond pas forcément à un événement matériel
  - ◊ transfert de contrôle ≈ appel de procédure *non contrôlé par le processus récepteur*



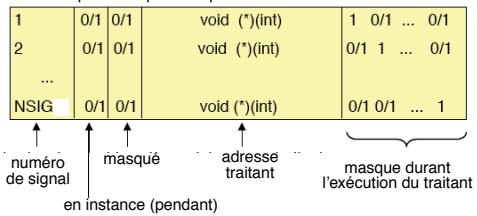
Quelques signaux (<signal.h>)

mnémonique	*événement correspondant*	*traitant par défaut*
SIGHUP	terminaison du leader	terminaison
SIGINT	control-C au clavier	terminaison
SIGQUIT	control-\ au clavier	terminaison+core
SIGTSTP	control-Z au clavier	suspension
SIGCONT	continuation d'un processus stoppé	reprise
SIGKILL	terminaison	terminaison
SIGPIPE	écriture dans un tube sans lecteur	terminaison
SIGFPE	erreur arithmétique (overflow…)	terminaison
SIGCHLD	terminaison d'un fils	vide (SIG_IGN)
SIGALRM	interruption horloge	terminaison
SIGTERM	terminaison normale	terminaison
SIGUSR1	laissé à l'utilisateur	terminaison
SIGUSR2	laissé à l'utilisateur	terminaison

- un traitant par défaut est associé à chaque signal
- le traitant par défaut peut être redéfini par l'utilisateur, sauf pour SIGKILL et SIGSTOP
- les mnémoniques sont communs à tous les UNIX
  (mais pas les numéros : SGCHLD vaut 17 pour Linux, 18 pour Solaris, 20 pour FreeBSD)
- un traitant vide permet d'ignorer un signal

Mise en œuvre

Dans chaque descripteur de processus :



- un signal reçu, mais non pris en compte est *en instance (ou : pendant)*
- lorsqu'un signal *masqué* est reçu, son traitement est mis en attente, jusqu'à ce que ce signal soit démasqué
- toute nouvelle occurrence d'un signal pendant est perdue
- lorsque le traitant associé au signal *S* est exécuté, *S* est masqué

Exemple

```
#include <signal.h>
void message(int sig) { /* traitant */
  printf("signal %d reçu\n",sig);
  exit(0);
}
int main() {
  signal(SIGINT, message); /* installe le traitant */
  signal(SIGQUIT, SIG_IGN); /* ignorer SIGQUIT*/
/* SIGINT et SIGQUIT sont interceptés */
  ...
/* on rétablit la terminaison par SIGINT et SIGQUIT */
  signal(SIGINT, SIG_DFL);
  signal(SIGQUIT, SIG_DFL);
  ...
}
```

Commentaires

- L'entier paramètre du traitant est le numéro du signal ayant provoqué l'exécution du traitant
  -> possibilité d'identifier l'évènement déclencheur dans le traitant
- 2 traitants sont définis par défaut
  - ◊ SIG_DFL : traitant par défaut associé au signal
  - ◊ SIG_IGN : traitant vide, permettant d'ignorer un signal
- En POSIX (et BSD), l'association définie par signal/sigaction est permanente

3 – Primitives de manipulation des signaux

Un des services où l'on observe le plus de divergences entre UNIX :

interfaces ≠
mécanismes ≠ } pour { BSD, POSIX, System V

Présentation centrée sur la définition POSIX

1) Association traitant/signal

Définition POSIX

```
int sigaction(int sig, const struct sigaction *newaction, struct sigaction *oldaction);
```

avec

```
struct sigaction {
  void (*sa_handler)(int);/* pointeur sur traitant */
  sigset_t sa_mask;       /* signaux à masquer durant l'exécution du traitant */
  int sa_flags;           /* options (souvent spécifiques aux implantations) */
}
```

Définition C standard

```
#include <signal.h>
typedef void (*handler_t)(int); /* procédure prenant un paramètre entier */
handler_t signal (int sig, handler_t traitant)
/* signal renvoie l'adresse du traitant précédent */
```


Héritage du traitement des signaux

- Après `fork()` : oui
- Après `exec()` :
  - ◊ le masque est conservé
  - ◊ les signaux ignorés (associés à `SIG_IGN`) le restent
  - ◊ les autres signaux reprennent leur traitant par défaut (`SIG_DFL`)

2)Emission

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

- désignation du destinataire
  - ◊ pid > 0 → signal envoyé au processus de numéro pid
  - ◊ pid = 0 → signal envoyé à tous les processus du même groupe que l'émetteur
  - ◊ pid = -1 → non défini
  - ◊ pid < -1 → signal envoyé à tous les processus du *groupe* |pid|
- le destinataire doit avoir le même propriétaire que l'émetteur

```
unsigned int alarm(unsigned int sec);
```

entraîne l'envoi du signal `SIGALRM` au processus appelant après un délai de `sec` secondes

3)Contrôle de la réception

Attente d'un signal

```
int pause();
```

Attente d'un signal quelconque

```
int sigsuspend(const sigset_t *masque);
```

positionne le masque courant à `masque` et attend un signal.
Le masque courant est restauré au retour de `sigsuspend`

Définition des masques

Opérations ensemblistes

```
int sigemptyset(sigset_t *set);       /*   *set = {}          */
int sigfillset(sigset_t *set);        /*   *set = {1..NSIG}   */
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
```

Affectation du masque courant :

```
int sigprocmask(int op, const sigset_t *set, sigset_t *oldSet);
```

- `op = SIG_BLOCK` → `set` est ajouté au masque courant ;
- `op = SIG_UNBLOCK` → `set` est retiré du masque courant ;
- `op = SIG_SETMASK` → `set` remplace le masque courant.

Ensemble des signaux masqués pendants

```
int sigpending(const sigset_t *set);
```


4 – Gestion des signaux pour les sessions interactives

Buts

- faciliter contrôle de sessions interactives
- factoriser la gestion de processus "liés" (démon + fils)

Organisation

- session = { groupes } = {{processus}}
- les groupes, ainsi que les sessions, sont disjoints
- groupes et sessions sont identifiés par leur créateur(*leader*)
- par défaut, groupes et sessions s'héritent
- un périphérique *peut* être associé à une session. Alors :
  - ◊ ce périphérique est le *terminal de contrôle* de la session ;
  - ◊ un unique groupe (groupe en *premier plan*) au plus *peut* interagir avec le terminal :
    - lire/écrire sur le terminal
    - capter (signaux) la frappe de : *intr, quit, susp (Ctrl-C,\,Z)*
  - ◊ les autres groupes (en arrière plan)
    - ignorent les signaux précédents, et
    - sont suspendus en cas de demande d'accès au terminal
  - ◊ lorsque le leader d'une session se termine, *SIGHUP* est diffusé aux membres de la session.
  - ◊ un périphérique peut être attaché à une session au plus
  - ◊ seul le leader peut définir le terminal de contrôle

Opérations

- gestion (création, test, affectation) des groupes et sessions : **`getpgrp, setpgid, getsid, setsid`**
- manipulation du groupe en avant plan : **`tcgetssid, tcgetpgrp, tcsetpgrp`**

5 – Transfert de contrôle asynchrone

But : offrir au programmeur un mécanisme (service) logiciel de commutation de contexte

Opérations de base

- sauvegarde du contexte du traitement courant
- remplacement du contexte courant par un contexte préalablement sauvegardé

Exemple (API système - UNIX -**) : bibliothèque *setjmp.h***

- *Sauvegarder* le contexte courant dans 1 zone mémoire (`sv_cntxt`) → `cr := setjmp(sv_cntxt)`
- Restaurer (et commuter avec) un contexte (sauvé dans `sv_cntxt`) → `longjmp(sv_cntxt, cr)`

Remarques

- Pour des raisons d'efficacité, la sauvegarde et la restauration ne portent que sur une partie du contexte des processus (pile d'appel, registres, et une partie du mot d'état programme). En particulier, les variables globales et le tas ne sont ni sauvegardés ni restaurés.
- En cas de restauration (appel à `longjmp`), le programme reprend son exécution comme s'il venait d'exécuter `setjmp`. La valeur (`cr`) renvoyée par `setjmp` permet de distinguer la sauvegarde (`0` pour l'appel à `setjmp`) de la restauration (valeur (≠`0`) renvoyée par `longjmp`).
- L'API POSIX (fonctions `sigsetjmp` et `siglongjmp`) est similaire (mais pas identique)

Exemple

```
#include <setjmp.h>
int val;
jmp_buf env;
...
/* sauvegarde d'un point de reprise */
val = setjmp(env);
if (val==0) {
   ...
   /* Cascade d'appels procéduraux  */
       ....
       /* Détection d'un problème et retour au point de reprise */
       longjmp(env, 1);
   ...
} else {
   /* traitement après longjmp  */
}
```


6 – Signaux et temps réel

1) Signaux et primitives de temporisation (`<sys/time.h>`)

Il est possible de programmer des temporisations avec un grain plus fin que ce que permet `alarm()`

Structures de données

```
struct timeval {
       long    tv_sec;         /* seconds */
       long    tv_usec;        /* microseconds */
};
```

```
struct  itimerval {
       struct  timeval it_interval; /* période */
       struct  timeval it_value; /* instant de départ (0 = jamais)*/
};
```

Horloges et signaux

- **`ITIMER_REAL`**temps physique (réel)émet **`SIGALRM`**
- **`ITIMER_VIRTUAL`**temps d'exécution en mode utilisateurémet **`SIGVTALRM`**
- **`ITIMER_PROF`**temps d'exécution totalémet **`SIGPROF`**

Primitives

```
int getitimer(int horloge, struct itimerval *val)
int setitimer(int horloge, struct itimerval *val, struct itimerval *oldval)
```


2)Signaux temps réel

- définis dans la norme POSIX 1.b
- les signaux temps réel
  - ◊ sont mémorisés (conservés dans des files)
  - ◊ ont une priorité, correspondant à leur numéro
  - ◊ peuvent être accompagnés de données spécifiques