TP système d'exploitation UNIX

1. Gestion des processus

Création

Permet la création dynamique d'un nouveau processus qui s'exécute de façon concurrente avec le processus qui l'a créé.
Un appel par un processus, que nous appellerons processus père, à la fonction fork entraîne, si cela est possible, la création d'un nouveau processus fils. La syntaxe est la suivante :

#include <sys/times.h>
pid_t fork(void);

Le programme exécuté par les deux processus étant le même et les données étant identiques, il est nécessaire, pour que le comportement des deux processus ne soit pas identique dans la suite de leur exécution, de pouvoir distinguer dans le code le retour dans le processus père de celui dans le processus fils.
La valeur de retour de la fonction est :

- ✓ 0, dans le processus fils;
- ✓ l'identité du processus fils créé, dans le processus père.
- ✓ -1, si la primitive échoue (et qu'il n'y a pas donc de création d'un nouveau processus, comme dans le cas où l'utilisateur créé trop de processus ou si le nombre total de processus dans le système est trop élevé).

getpid identité du processus en cours
getppid identité du processus parent de celui en cours
getuid propriétaire réel : en général celui du login
geteuid propriétaire effectif : propriétaire du processus pour lequel le bit set-uid a été modifié pour permettre à un exécutable d'être exécuter par un autre utilisateur.
set-uid est le bit à changer pour changer d'utilisateur en utilisant la fonction setuid
getgid groupe réel
getegid groupe propriétaire effectif
getpwd répertoire de travail

Pour calculer le CPU consommés dans les 2 modes on utilise la structure tms prédéfini dans le fichier sys/times et accessible au moyen de la fonction clock_t times(struct tms *buf);

Exemple :

```
#include <sys/times.h>
#include <stdio.h>
main(){int  pid;
        switch(pid=fork()){
        case    -1: perror("Création de processus");
                exit(2);
        case    0 :                     /* on est dans le processus fils*/
                printf("valeur de fork = %d", pid);
                printf("je suis le processus %d de père %d\n", getpid(), getppid());
                printf("fin de processus fils\n ");
                exit(0);
        default :                       /*on est dans le processus père*/
                printf("valeur de fork = %d", pid);
```

```
        printf("je suis le processus %d de père %d\n", getpid(), getppid());
        printf("fin de processus père\n ");}}
```

Héritage

Le fils hérite quelques attributs : les propriétaires réels et effectifs, le répertoire de travail et la valeur de nice. Par contre la priorité du fils est initialisée à une valeur standard et les verrous sur les fichiers détenus par le père ainsi que les signaux reçus pendant ne sont pas hérités par le fils :

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/times.h>
char buf[1024];              /* pour récupérer le répertoire de travail */
struct tms temps;            /* pour récupérer les nombres de clics */
main(){ int i;
    nice(10); augmentation de 10 de la valeur du nice avant le fork*/
    for (i=0;i<10000000;i++);  /* une boucle consommatrice de CPU */
    if (fork()==0) {printf("caractéristiques du fils \n ");
        printf("uid=%d  euid= %d  egid=%d\n ", getuid(),geteuid(),getegid());
        printf(" répertoire de travail : %s\n ",getcwd(buf,1024));
        printf("nice : %d \n",nice(0)+20);
        times(&temps);
        printf("clics en mode utilisateur : %d \n", temps.tms_utime);
        printf("clics en mode système : %d \n\n ", temps.tms_stime);}
    else{  sleep(5);                /* pour partir après la terminaison du fils */
        printf("caractéristiques du père \n ");
        printf("uid=%d  euid= %d  egid=%d\n ",getuid(),geteuid(),getegid());
        printf(" répertoire de travail : %s\n ",getcwd(buf,1024));
        printf("nice : %d \n",nice(0)+20);
        times(temps);
        printf("clics en mode utilisateur : %d \n", temps.tms_utime);
        printf("clics en mode système : %d \n\n ", temps.tms_stime);}}
```

Copie des données

La copie des données lors de la création d'un processus ne se fait plus juste après sa création, mais lors d'un accès à la donnée en écriture (ainsi les performances du système sont améliorées).

Exemple :

```
#include <sys/times.h>
int n=1000;
    main(){int m=1000, pid;
    printf("Adresse de n dans le père: %p\n ", &n);
    printf("Adresse de m dans le père: %p\n ", &m);
    printf("1 valeur de m et n dans le père : %d %d\n ", m, n);
    switch(pid=fork()){
      case   -1:   perror("fork");exit(2);
      case   0 :                  /* on est dans le processus fils*/
                   printf("Adresse de n dans le fils: %p\n ", &n);
                   printf("Adresse de m dans le fils: %p\n ", &m);
                   printf("2 valeur de m et n dans le fils : %d %d\n ", m, n);
```

```
                m*=2;n*=2;
                printf("3 valeur de m et n dans le fils : %d %d\n ", m, n);
                sleep(3);
                printf("6 valeur de m et n dans le fils : %d %d\n ", m, n);exit(0);
        default:                /*on est dans le processus père*/
                sleep(2);
                printf("4 valeur de m et n dans le père : %d %d\n ", m, n);
                m*=3;n=*3;
                printf("5 valeur de m et n dans le père : %d %d\n ", m, n);
                sleep(2);
                exit(0);}}
```

Les processus Zombies

Un processus Zombie est l'état d'un processus dont le père n'a pas encore pris connaissance de sa terminaison. Il se caractérise dans le résultat affiché par la commande ps par la valeur Z de l'état, 0 pour l'espace mémoire et sur de nombreuse versions 'defunct' dans le champ commande.

La raison du passage à cet état est que tout processus Unix se terminant possède une valeur (code de retour ou status d'exit) à laquelle son père peut accéder; un peu comme une fonction appelant une autre fonction récupère la valeur de cette fonction. Cependant à la différence du cas des fonctions dans lequel la fonction appelante est bloquée durant l'exécution de la fonction appelée, dans le cas des processus les processus père et fils peuvent se dérouler "en parallèle". Aussi le système doit-il fournir le moyen à un processus d'accéder à tout moment au code retour de ses fils terminés.

```
#include <stdio.h>
#include <sys/times.h>
main(){ if (fork()==0) {printf("fin du processus fils de numéro %d \n ", getpid());
                        exit(2); }
        sleep(30);}
```

Synchronisation

Les primitives wait et waitpid

Ce sont ces primitives et elles seules qui permettent l'élimination des processus Zombies et permettent la synchronisation d'un processus sur la terminaison de ses descendants avec récupération des informations relatives à cette terminaison.

L'appel système wait() permet à un processus appelant de suspendre son exécution en attente de recevoir un signal de fin de l'un de ses fils.

Syntaxe : int wait(status);
int *status;

Les principales actions de wait sont:

- ✓ Recherche de processus fils dans la table des processus : si le processus n'a pas de fils, le wait renvoie une erreur, sinon elle incrémente un compteur.
- ✓ S'il y a un zombie, il récupère les paramètres nécessaires (accounting) et libère l'entrée correspondante de la table des processus.
- ✓ S'il y a un fils mais pas de zombie, alors le processus se suspend (état endormi) en attente d'un signal.
- ✓ Lorsque le signal est reçu, la cause du décès est stockée dans la variable "status". 3 cas à distingués : processus stoppé, processus terminé volontairement par exit, processus terminé à la suite d'un signal.

✓ Enfin le wait permet de récupéré le PID du processus fils : PID=wait(status).

La primitive **waitpid** permet de sélectionner parmi les fils du processus appelant un processus particulier.

Processus Zombie

❖ Cas normal : Le père attend son fils

```
#include <sys/times.h>
#include <stdio.h>
main(){
        int PID, status;
        if (fork()==0)
                {printf("processus fils %d\n", getpid());
                exit(10);}
        PID=wait(&status);
        printf("processus père %d\n", getpid());
        printf("sortie du wait \n ");
        sleep(15);
                /* fils est bien terminé père toujours en place signal et infos reçus */
        printf("PID = %d status = %d\n", PID, status);
        exit(0);
}
```

❖ Cas d'un Zombie:le père n'attend pas son fils et est toujours en vie après la mort de son fils

```
#include <sys/times.h>
#include <stdio.h>
main(){int PID, status;
        if (fork()==0) {printf("processus fils %d\n", getpid());
                exit(10);}
        printf("processus père %d\n", getpid());
        for (;;);                       /* le processus père boucle */
}
```

La commande ps -ef affiche "defunct" pour le processus fils.

❖ Cas où le père reçoit le signal de terminaison de son fils et n'exécute le wait qu'après :
Le processus fils reste zombie momentanément, le temps pour le père de recevoir les différentes informations.

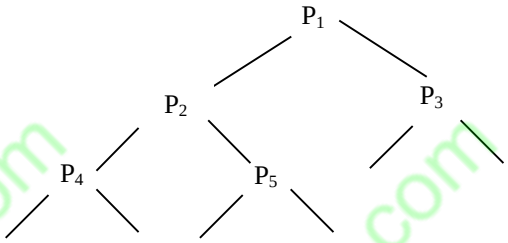
```
#include <sys/times.h>
#include <stdio.h>
main(){int PID, status;
        if (fork()==0)
                {printf("processus fils %d\n", getpid());
                exit(10);}
        printf("processus père %d\n", getpid());
        sleep(15);                      /* le père endormi n'attend pas ... */
        printf("sortie du 1 er sleep \n ");
        PID=wait(&status);
        printf("sortie du wait d\n ");          /* il n'y a plus de Zombie */
        sleep(15);  /* tout a été pris en compte et traité */
        printf("PID = %d status = %d\n", PID, status);
        exit(0);}
```

2. Communication inter-processus

5-1 Les pipes

Le pipe est le moyen de communication le plus utilisé par les applications dont les principales caractéristiques sont :

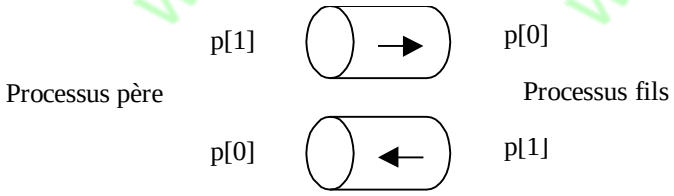
- La nature d'un pipe est de type FIFO.
- Un pipe est unidirectionnel: un processus peut soit lire soit écrire dans un pipe
- L'information disparaît après lecture et le pipe après le dernier processus
- La taille du pipe est limitée de 4k à 20k dépendant du matériel.
- Le pipe est un objet du type fichier associé à 2 descripteurs de fichier et à 2 entrées dans la table des fichiers ouverts
- Le pipe n'a pas de noms dans l'arborescence de Unix et donc son ouverture ne peut se faire à l'aide de open mais plutôt de pipe
- La communication n'a lieu qu'entre processus de la même famille par héritage des descripteurs.



Les pipes créés par le processus P_2 ne permettent la communication qu'entre les processus P_2 P_4, P_5 et leurs descendances. Les processus P_1 et P_3 ne peuvent accéder à ces pipes.

- Création

int pipe(p)
int p[0];
L'appel système pipe retourne 2 descripteurs de fichier, p[0] ouvert pour la lecture et p[1] ouvert pour l'écriture.



- Lecture

Effectuée par un read standard
read(p[0],buf,5); lecture de 5 caractères du pipe dans un buffer.

- Ecriture

Effectuée par un write standard
L'écriture sur un pipe fermé en lecture renvoie un signal SIGPIPE d'où l'interruption du processus si le signal n'est pas ignoré.
Un processus qui tente d'écrire sur un pipe plein reste bloqué (état endormi).
write(p[1],"ABCD",4); écriture de 4 caractères dans le pipe.
La fermeture est effectuée par le close.
Un processus qui ferme les descripteurs p[0] et p[1], renonce définitivement à l'utilisation du pipe et ne pourra plus l'ouvrir ne disposant d'aucun moyen pour le faire.

❖ Communication entre deux processus

```
#include <stdio.h>
#include <sys/signal.h>
main(){int fils1, fils2, n, m, p[2]; char buf[5];
        pipe(p);                                /* création de pipe */
        if (fils1=fork()==0)                    /* création du premier fils */
        {       printf("je suis le fils producteur \n");
                printf("j'écris 5 caractères dans le pipe \n");
                write(p[1],"ABCDE",5);
                printf("fin d'écriture dans le pipe \n");
                exit(3);}
        else                                    /* le père crée le fils consommateur */
        {       if (fils2=fork()==0)            /* création du deuxième fils */
                {       printf("je suis le fils consommateur \n");
                        read(p[0],buf,5);       /* lecture du pipe */
                        printf("voici les caractères lus \n");
                        write(1,buf,5);/*affichage des caractères sur output standard*/
                        printf("\n");
                        exit(3);}
                else{   printf("processus père c'est fini .... \n");
                        wait(&n);
                        wait(&m);}
        }
}
```

Pour ce programme exécutez les tests suivants et observez ce qui se passe :

- Remplir le pipe jusqu'à dépasser sa taille.
- Fermer le pipe avant de quitter le programme et essayer de le rouvrir.
- Lire d'un pipe vide.
- Lire deux fois les même données du pipe.

❖ Les pipes nommés

Un pipe nommé a les mêmes caractéristiques qu'un pipe ordinaire plus

- Il a un nom sous Unix comme un fichier ordinaire
- Il est créé de la même manière qu'un fichier spécial avec mknod
- Il est accessible par les processus n'ayant pas de lien de parenté
- Il subit les mêmes règles qu'un fichier ordinaire

Création

```
#include <sys/types.h>
#include <sys/stat.h>
int mknod (path, mode)
char  *path; /* chemin */
int mode; /* type + permissions */
```

le mode est indiqué par trois sortes de flags : type de fichier (S_FIFO=0010000 pour un FIFO), permission, exécution particulière.

Ouverture : open (/uo/tcom/mypipe, O_RDWR, 2) ouverture bloquante pour la L/E
open (/uo/tcom/mypipe, O_RDWR | O_NDELAY, 2) ouverture non bloquante la L/E

Lecture : read(d, buf, 2000) lecture d'un FIFO

Ecriture : pour écrire dans le pipe il faut l'ouvrir avec open.
write(d,"ABCDE",5) écriture de 5 caractères dans un FIFO de descripteur d.

Si le pipe est plein et que l'ouverture soit non bloquante (le flag O_NDELAY positionné) l'écriture retourne un 0.
Si l'ouverture est bloquante (le flag O_NDELAY non positionné) le processus reste bloqué jusqu'à ce qu'il y ait de la place (par suite d'une lecture).

Suppression : rm or unlink.
TP : Créer le pipe "mypipe" à partir du shell dans votre répertoire avec mknod mypipe

- Ecriture non bloquante : programme P1

```
#include <stdio.h>
#include <sys/signal.h>
#include <fcntl.h>
main(){int d, i, n;
        char buf[2000];
        d=open("/home/étudiant1/mypipe", O_RDWR | O_NDELAY, 2);
        if (d<0) {      printf("problème d'ouverture du pipe \n");
                        exit(0);}
                                        /* remplissage du buffer avec des A */
        for (i=0;i<2000; i++) buf[i]="A";
        if ((n=write(d, buf, sizeof(buf)))>0) {printf("écriture effective  %d\n", n);
                                                exit(0);}
        }
```

- Lecture bloquante : programme P2

```
#include <stdio.h>
#include <sys/signal.h>
#include <fcntl.h>
main(){int d, i;
        char buf[2000];
        d=open("/home/étudiant1/mypipe", O_RDWR, 2);
        if (d<0) {printf("problème de lecture du pipe \n");
                        exit(0);}
        n=read(d, buf, 2000);           /* lecture du pipe dans buf */
                                        /* affichage du buffer sur l'écran */
        printf("affichage du contenu du pipe \n");
        write(1, buf, n);
        printf("\n");}
```

Pour ce programme exécutez les tests suivants et observez ce qui se passe :

- Lancez en premier P1 et ensuite P2, est ce que la lecture et réalisée.
- Lancez en premier P2 et ensuite P1 que se passe-t-il pour la lecture.
- Lancez P1 ensuite P2 deux fois en lisant 2000.
- Lancez P2 avec lecture de 4000 et P1 avec écriture de 2000.

Remarque :

- Pour la lecture bloquante le processus se bloque jusqu'à réception d'un message du processus envoyeur, si ce dernier l'envoie avant que le récepteur se mette en attente, le récepteur continu à attendre jusqu'à réception d'un autre message.
- L'avantage majeur d'un pipe nommé par rapport à un pipe simple est lié à sa désignation externe. Ainsi n'importe quel processus pourvu qu'il en ait les droits peut ouvrir le pipe en lecture et en écriture et communiquer avec d'autre processus ayant ouvert le même pipe.

- Quant aux autres limites déjà soulignées, elles sont les mêmes pour les deux types de pipe: Petite taille, caractère temporaire de l'information, possibilité de blocage comme dans l'exemple ci-dessous ou la capacité du pipe est de 5120 et donc au-dessus de cette taille rien ne peut y être écrit :

- **Ecriture dans un pipe nommé plein**

```
#include <stdio.h>
#include <fcntl.h>
main(){int d, i;
        char buf[2000]; /*cette ouverture est bloquante, "blocage si mypipe est plein*/
        d=open("/home/étudiant1/mypipe", O_RDWR, 2);
        for (i=0;i<5121; i++) write(d, "P", 1) }
```

5-2 Les signaux

La signalisation constitue le mécanisme fondamental de communication non seulement entre processus mais également entre le noyau et l'ensemble des processus du système.
Un signal est défini par un code numérique (dans le fichier /usr/include/sys/signal.h) et se manifeste de manière asynchrone par notification d'un événement au processus concerné; cet événement est mémorisé dans la table des processus.
Contrairement aux autres mécanismes de communication interprocessus, les signaux ne transportent pas de données, mais indiquent des actions à effectuer dans certaines circonstances.

Types de signaux :

NOM	N°	FONCTION
SIGHUP	1	Signal hangup envoyé à tous les processus liés à un terminal, lors de la déconnexion de celui-ci (CTRL-D)
SIGINT	2	Interruption des processus d'un terminal (CTRL-C, BREAK, DEL)
SIGQUIT	3*	Interruption avec image mémoire : core (CTRL \)
SIGILL	4*	Instruction illégale
SIGTRAP	5*	Trappe envoyée" à un processus en mode trace après exécution de chaque instruction
SIGIOT	6*	Instruction IOT : problème matériel
SIGEMT	7*	Instruction EMT : problème matériel
SIGFPE	8*	Exception de virgule flottante
SIGKILL	9	Permet de tuer un processus de façon brutale; ne peut être capté ni ignoré pour des priorités > PZERO=25
SIGBUS	10*	Erreur BUS
SIGSEGV	11*	Violation de segment
SIGSYS	12*	Mauvais argument dans un appel système
SIGPIPE	13	Ecriture dans un pipe sans lecteur
SIGALARM	14	Alarme horloge
SIGTERM	15	Fin normale d'un processus (kill)
SIGUSR1	16	Signal à la disposition de l'utilisateur (signal 1)
SIGUSR2	17	Signal à la disposition de l'utilisateur (signal 2)
SIGCLD	18	Mort d'un fils signalée au processus père
SIGPWR	19	Redémarrage après coupure de courant
SIGPOLL	22	Sélection d'événements dans les streams (événements multiples asynchrones)

* production de l'image mémoire sur disque (core dump)

❖ Emission d'un signal kill()

Cette primitive permet d'envoyer un signal à un processus ou à un groupe de processus.

```
#include <sys/signal.h>
int kill(pid, sig)
int pid, sig;
```

les entiers pid et sig désignent respectivement l'identificateur du processus destinataire ou groupe de processus, et le numéro du signal émis. Les processus émetteurs et destinataire doivent avoir le même identificateur (réel ou effectif) d'utilisateur.

- pid > 0 : le signal est destiné au processus dont le PID = pid.
- pid = 0 : le signal est destiné à tous les processus de même groupe que le processus émetteur à l'exception des processus spéciaux : init, swapper, sched …
- pid = 1 : et l'euid n'est pas celui de su le signal est envoyé à tous les processus (hormis les spéciaux) dont l'uid = euid émetteur.
- pid = -1 : et euid = root le signal est envoyé à tous les processus sauf les processus spéciaux.
- pid < -1 : le signal est envoyé à tous les processus dont l'identificateur de groupe de processus est égal à la valeur absolue de pid. Kill retourne un 0 s'il n'y a pas d'erreur et -1 dans le cas d'erreur.

❖ Réception d'un signal par un processus

Le type de signal reçu par un processus indique la nature de l'événement qui a nécessité l'envoi de ce signal par un autre processus.

```
#include <sys/signal.h>
    int (*signal(sig, func))()
    int sig, (*func)();
```

La valeur de sig désigne le numéro du signal à recevoir. A la réception d'un signal par le processus, trois actions sont possibles selon la valeur de l'argument func (flags SIG-DFL et SIG-IGN ou une routine) :

1. **Action par défaut** : SIG-DFL; c'est la mort du processus
2. **Signal ignoré** : SIG-IGN; le signal reçu est ignoré par le processus, sauf le cas SIGKILL. Par exemple un processus peut ignorer le signal SIGCLD émis à la mort d'un fils: signal(SIGCLD, SIG_IGN)
3. **Une routine est exécutée par le processus**; cette routine est pointée par func : c'est par exemple une routine traitant le cas d'une écriture dans un pipe sans lecteur : signal(SIGPIPE, e_pipe).
   - ❖ Le numéro du signal sig est le seul argument à passer à la fonction de traitement du signal.
   - ❖ Au retour de la routine de traitement le processus reprend son exécution au point d'interruption.
   - ❖ Lorsque le signal arrive pendant un appel système (read, write, open, ioctl) sur un device lent (un terminal par exemple) ou pendant un appel pause ou wait, la routine de traitement du signal sera exécutée, et l'appel système sera interrompu avec un code de retour -1. Le seul signal ne pouvant pas être capté est SIGKILL.

   Exemples exécution d'une fonction lors de la réception du signal d'écriture dans un pipe 'SIGPIPE' :

1. Ecriture dans un pipe sans lecteur

```
#include <sys/signal.h>
main(){int p[2], e-pipe();
        signal(SIGPIPE, e-pipe);     /* appel signal (ne pas mettre e-pipe()*/
        pipe(p);                     /* création du pipe */
        close (p[0]);                /* aucun lecteur */
        write (p[1], "ABCDE", 5);    /* écriture */
}
e-pipe()                                        /* traitement de SIGPIPE */
{       printf(" réception du SIGPIPE \n");
        printf(" écriture dans un pipe sans lecteur \n");
        exit(0);}
```

2. Un processus n'attendant pas la mort du fils et ignorant le signal SIGCLD, donc pas de zombie

```
#include <sys/signal.h>
main(){signal(SIGCLD, SIG-IGN); /* SIGCLD ignoré */
        if (fork()==0)
        {       printf("processus fils PID= %d\n", getpid());
                exit(1);}
        printf("processus père PID=%d\n", getpid());
        sleep(30);
        printf("signal mort fils ignoré \n ");
        for(;;);
}
```

3. Un processus ignore le signal SIGHUP

Il arrive bien souvent que certains processus lancés en backround sur un terminal ne soient pas interrompus à la suite d'une déconnexion de celui-ci.

```
#include <sys/signal.h>
main(){signal(SIGHUP, SIG-IGN); /* hungup ignoré */
        for(;;);
}
```

Le programme lancé en backround ne s'arrêtera pas à la déconnexion du terminal(CTRL-D)

4. Un processus envoie le signal SIGUSR1 à définir par l'utilisateur à son fils

```
#include <sys/signal.h>
#include <stdio.h>
int sig_user(), PID=0;
main(){signal(SIGCLD, SIG-IGN); /* SIGCLD ignoré */
        if ((fork())>0)
        {       printf("processus père PID= %d\n", getpid());
                sleep(30); /* laisse au fils le temps nécessaire pour traiter le signal */
                kill(PID, SIGUSR1);
                printf("envoie du signal au fils \n");}
        else{   if ((fork())<0) exit(1);
                printf("le fils arme le signal \n");
                signal(SIGUSR1, sig_user);/* ne pas mettre les parenthèses sig_usre()*/
                pause(); /* attente du signal */}
        exit(0);
```

```
}
/* routine de traitement du signal */
sig_user(){     printf("signal reçu par le fils \n");
                exit(0);
          }
```

Nous pouvons remarquer que PID est initialisé à la valeur 0. Et donc le signal est envoyé au groupe de processus, c'est à dire aux deux fils. On peut faire l'expérience de remplacer les 2 appels fork par un seul au début x=fork et ne créer qu'un seul fils ou deux et d'envoyer le signal à un fils et aux deux et de comparer les résultats avec kill(x, SIGUSR1).

❖ Attente d'un signal : **pause**()

La primitive pause permet à un processus de se mettre en attente d'un signal qui ne doit pas être ignoré. On ne reviendra de l'appel pause que si le signal reçu entraîne la mort du processus appelant. Si le signal attendu doit être traité par une fonction et arrive pendant l'exécution de pause, on a un retour de -1 (erreur EINTR).

Exemple : signal attendu et ignoré

```
#include <sys/signal.h>
#include <stdio.h>
int sig, PID;
main(){int sig, PID;
        if ((PID=fork())<0)  exit(1);
        if (PID)
        {                                   /* processus père */
            sleep(10);
            kill (PID, SIGTERM);            /* signal au fils */}
        else{                               /* processus fils */
            sig=signal(SIGTERM, SIG_IGN);
            printf("je suis toujours là \n");
            pause();                        /* attente */
          }
}
```

TP : essayez d'exécuter le programme en commentant le kill pour voir si le signal est envoyé ou non ensuite de commenter le signal pour voir si le signal est ignorer si le processus reste bloqué sur le pause ou non.

❖ Activation du signal SIGALRM: alarm()

nsigned alarm(sec)
unsigned sec;

Elle permet l'envoi du signal SIGALRM au processus au bout de sec secondes. Si sec=0 les requêtes d'alarmes précédentes sont annulées. Les demandes d'alarmes ne sont pas empilables; aussi les demandes successives non satisfaites ne font que réinitialiser l'horloge d'alarme.

Exemple : Exécution d'une routine dès réception de l'alarme

```
#include <sys/signal.h>
#include <stdio.h>
routal()
{                           /* routine après réception de l'alarme */
```

```
        for(;;)
        {       system("/usr/games/banner -w40 'bonjour' ");
                sleep(2);
                system("/usr/bin/tput clear ");
        }
}
main(){         printf("j'attend une alarme \n");
                alarm(10);
                signal(SIGALRM, routal);
                pause();
        }
```

TP : enlevez la commande pause et remarquez le comportement du programme.

❖ Déroutements : setjmp() et longjmp()

Il est possible à la manière d'un goto de se dérouter vers un point déterminé d'un programme à l'aide de ces 2 primitives. Elles sont utiles par exemple dans le traitement d'erreurs et dans les interruptions de sous programmes.

```
#include <setjmp.h>
int setjmp(env)
jmp_buf(buf);
void longjump(env,val)
jmp_buf env;
int val;
```

La primitive setjmp() sauvegarde l'état dans env et longjmp() le restaure; l'exécution reprend à l'instruction où env a été sauvegardé avec comme valeur le retour val.

❖ Suspension d'un processus : sleep()

La primitive sleep permet de suspendre l'exécution d'un processus pendant une durée exprimée en secondes sans utiliser explicitement les primitives alarm() et pause().

```
Unsigned sleep(secs)
Unsigned secs;
```

5-3 IPC

L'IPC du système V apporte une solution aux différentes limitations des mécanismes précédents dont voici quelques rappels :

- Pour les pipes
  Taille faible, information non permanente, blocage …
- Pour les signaux
  Pas de transport de données, nécessité d'identifier le processus ou groupe de processus destinataire d'un signal.

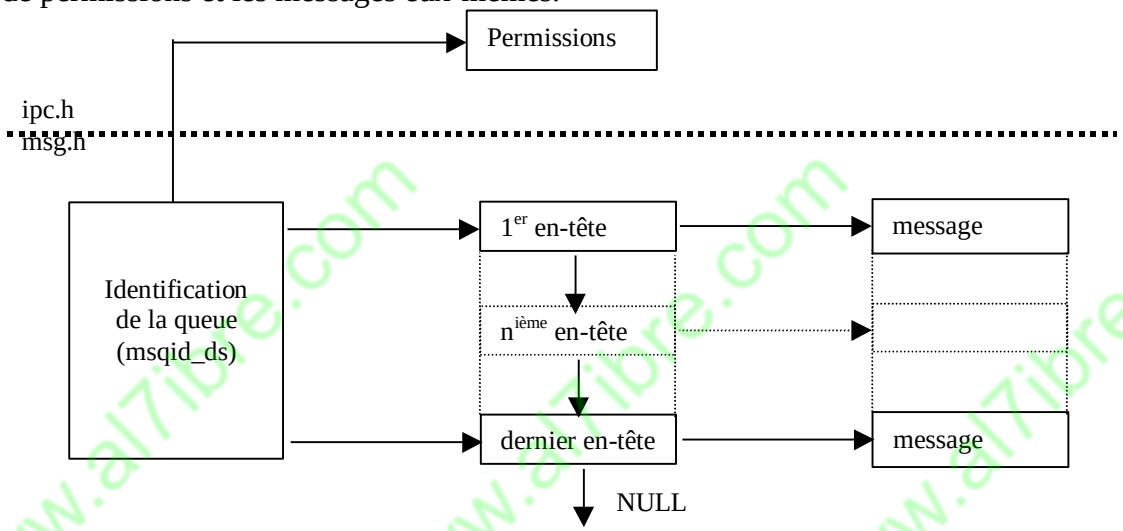
Avec l'IPC la coopération entre processus est beaucoup plus riche grâce à trois méthodes de communications :

- Les messages organisés en queues (message queue)
- Les segments de mémoires partagés (shared memory)
- Les sémaphores (semaphores)

5-3-1 Les queues ou files de messages

Les processus communiquent à travers des files de messages. Ces derniers sont créés par un processus donné appelé créateur et en est le propriétaire. Les messages écrits dans la file sont affectés de numéros conventionnels appelés types. Une queue est caractérisée par une clé définie par le créateur, un identifiant retourné à la création et récupéré avant toute opération d'écriture ou de lecture. Tout processus désirant effectuer une opération sur une file de messages doit posséder la clé et les permissions nécessaires.
Les opérations de lecture ou d'écriture peuvent être bloquantes ou non (flags IPC_NOWAIT).
Une file de messages est caractérisée par un ensemble de paramètres réglables du noyau : taille d'une file ou d'un message, nombre de messages dans une file, etc.

Les différentes structures des files de messages peuvent être schématiquement représentées en une structure identifiant chaque file de messages, plus un ensemble d'en-têtes, une structure de permissions et les messages eux-mêmes.



Chaque en-tête pointe sur l'en-tête suivant et sur le buffer contenant le message lui-même.

- Structure de données identifiant une queue

```
struct msqid-ds { struct ipc_perm msg_perm; /* permission */
                  struct msg *msg_first;     /* 1er en-tête */
                  struct msg *msg_last;      /*  dernier en-tête */
                  ushort msg_cbytes;         /* nombre d'octets dans la queue */
                  ushort msg_qnum;           /* nombre de messages dans la queue */
                  ushort msg_qbytes;         /* nombre max d'octets dans la queue */
                  ushort msg_lspid;          /* PID du dernier msgsnd */
                  ushort msg_lrpid;          /* PID du dernier msgrcv*/
                  time_t msg_stime;          /* heure du dernier msgsnd */
                  time_t msg_rtime;          /* heure du dernier msgrcv */
                  time_t msg_ctime;          /* heure du dernier changement */
                };
```

- Structure de données d'un en-tête de message

```
struct msg {  struct msg *msg_next;      /* ptr sur message suivant */
              long msg_type;             /* type de message */
              short msg_ts;              /* taille du texte du message */
              short msg_spot;            /* adresse du message lui-même */
              };
```

- Structure du message

```
struct msg {    long mtype;             /* type du message */
                char mtext[1|;          /* texte du message, le champ mtext doit
                                        être adapté à la taille souhaité */
        };
```

- Structure des permissions communes aux 3 IPC

```
struct ipc_perm {
        ushort uid;             /* uid du propriétaire */
        ushort gid;             /* gid du propriétaire */
        ushort cuid;            /* uid du créateur de la queue */
        ushort cgid;            /* gid du créateur de la queue */
        ushort mode;            /* mode d'accès */
        ushort seq;
        key_t key;               /* clé */
};
```

- Constante commune

Ce sont des flags utilisés pour la création et le contrôle de l'IPC considéré.
IPC_CREAT : création d'un nouvel élément (queue …)
IPC_EXCL : échec de la création si la clé existe
IPC_NOWAIT: opération non bloquante retour =-1
IPC_PRIVATE: clé privée
IPC_RMID : suppression d'un élément (queue, segment partagé, sémaphore)
IPC_SET : mise à jour des éléments de la structure d'identification (ex: msqid)
IPC_STAT : lecture du status de la structure, extraction des caractéristiques

- Constantes spécifiques aux files de messages

MSG_R=0400 : permission en lecture
MSG_R=0200 : permission en écriture
MSG_RWAIT=01000: un lecteur attend un message
MSG_WWAIT=02000: un écrivain attend d'envoyer un message

- Création d'une file de messages msgget()

Permet de créer une nouvelle file de message ou de récupérer l'identificateur d'une existante à condition d'en fournir la clé.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget(key, msgflg)
key-t key;              /* défini dans types.h */
int msgflg;             /* permission, mode d'exécution, commandes de contrôle */
```

Il y a création d'une nouvelle queue de messages dans les conditions suivantes:

- La clé est égale à IPC_PRIVATE (0)
- La clé n'existe pas déjà et le flag IPC_CREAT est positionné (msgflg et IPC_CREAT)
- Après la création la structure d'identification est initialisée (uid/gid du créateur, taille, date de création …) En cas de succès, msgget() retourne un nombre qui est l'identificateur de la file de messages. L'entier msgflg est une combinaison de différentes conditions.

Exemples
- msqid = msgget (key, (IPC_CREAT |0200)); création en écriture
- msqid = msgget (key, (IPC_CREAT | IPC_EXCL |0200)); création en écriture avec retour d'une erreur si la clé existe déjà.

- Opération de contrôle d'une file de messages msgctl()

Cette primitive permet d'effectuer les opérations de contrôles indiquées dans cmd dont les valeurs sont : IPC_STAT, IPC_SET, IPC_RMID. Elle retourne ensuite 0 en cas de succès et -1 en cas d'échec.

```
int msgctl(msqid,cmd, buf)
int msqid, cmd;
struct msqid_ds *buf;
```

Exemple : libération d'une queue msqid : msgctl(msqid, IPC_RMID, NULL)

- Emission d'un message msgsnd()

```
int msgsnd(msqid,msgp, msgsz, msgflg)
int msqid, msgsz, msgflg;
struct msgbuf *msgp;
```

la fonction msgsnd() permet d'envoyer un message dans une queue après avoir récupéré son identificateur; msgp pointe sur le buffer contenant le message dont la structure est la suivante:

```
struct msgbuf{
        long mtype;          /* type du message */
        char mtexte[ ];      /* texte du message */};
```

- La longueur d'un message msgsz va de 0 au maximum MSG_MAX
- L'écriture est bloquante ou non (msgflg et IPC_NOWAIT= faux ou vrai)
- Si msgsnd réussit : msg_qnum est incrémenté de 1, msg_lspid=PID de l'appelant, msg_stime=heure courante

- Réception d'un message msgrcv()

```
int msgrcv(msqid,msgp, msgsz, msgtyp, msgflg)
int msqid, msgsz, msgflg;
long msgtyp;
struct msgbuf *msgp;
```

La fonction msgrcv() permet le transfert d'un message d'une queue identifiée par msqid vers un buffer utilisateur pointé par msgp.

- msgsz désigne la longueur du message attendu. Si la longueur réelle est supérieure à msgsz, la taille lu est ramenée à msgsz si le flag MSG_NOERROR est positionné; sinon msgrcv() retourne une erreur.
- Msgtyp spécifie le type du message à lire; si msgtyp=0, le premier message de la queue sera lu; si msgtyp > 0 le premier message de ce type sera lu, si msgtyp < 0, le premier message de la queue dont le type est valeur absolue inférieur ou égal à msgtyp est reçu.
- Msgflg indique l'action à entreprendre selon que le message est présent ou non dans la file. Si IPC_NOWAIT est positionné, la lecture est non bloquante, sinon le processus se suspend jusqu'à l'arrivée d'un message, d'un signal ou d'une suppression de la queue de messages considérée. Et enfin dans le cas de succès de msgrcv() nous avons:
  - msg_qnum décrémenté de 1, diminution du nombre de messages de la queue.
  - msg_lrpid contient le PID du processus lecteur
  - msg_rtime = heure courante
  - valeur retournée = nombre d'octets lus dans le buffer.

Exemple :

1. Création d'une queue

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#define CLE 1024
main(){int msqid;                       /* identificateur de la queue */
                                        /* création de la queue de clé 1024 */
        if ((msqid=msgget((key_t) CLE, 0666+IPC_CREAT )) == 1)
        {       printf ("erreur d'allocation \n ");
                exit(1); }
                printf ("identifiant msqid= %d \n", msqid);
        exit(0);
        }
```

2. Suppression d'une queue

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
main(){int msqid, CLE;
        printf("suppression de la queue . . . CLE= ? \n");
        scanf ("%d", &CLE);
        if ((msqid=msgget((key_t) CLE, 0 )) < 0)
        {       perror ("échec récupération de l'identificateur");
                exit(1); }
                                /* suppression de la queue */
        if (msgctl(msqid, IPC_RMID, NULL)==-1)
        {       perror ("échec suppression");
                exit(2); }
        exit(0); }
```

3. Emission d'un message dans une queue

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <string.h>
#include <stdio.h>
struct msgbuf msgp;
struct msqid_ds buf;
char msg [20];                  /* message tapé au clavier */
main(){int msqid, CLE;
        printf("donnez la CLE= de la queue \n");
        scanf ("%d", &CLE);
        printf("tapez votre message de 20 caractères maximum \n");
        scanf ("%s", msg);/* utilisez read(1,msg,20) si vous voulez taper un message
```

```
                                contenant du blanc*/
        if ((msqid=msgget((key_t) CLE, 0 )) == 1) //récupérer l'identificateur d'abord
        {       printf ("problème de récupération identificateur \n ");
                exit(1);};
        printf ("voici le status de la queue \n ");
        msgctl(msqid, IPC_STAT,  &buf)
        printf ("cbytes     %d \n", buf.msg_cbytes);
        printf ("qbytes     %d \n", buf.msg_qbytes);
        printf ("lrpid       %d \n", buf.msg_lrpid);
                                        /* écriture du message . . . */
        msgp.mtype = 20; /* type associe au message */
        strcpy(msgp.mtext, msg); /* message à envoyer */
        printf("message copie dans le buffer    %s \n",msgp.mtext);
        if (msgsnd(msqid, &msgp,strlen (msg), IPC_NOWAIT==- 1)
        {       printf ("problème d'émission du message");
                exit(2); };
        printf ("le message est-il encore dans le buf %s\n", msgp.mtext);
        printf ("le type du message est %ld\n", msgp.mtype);
        exit(0); }
```

4. Lecture d'un message dans une queue connue

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/errno.h>
#include <sys/msg.h>
#include <stdio.h>
struct msgbuf msgp;
main(){
        int msqid, l,CLE;
        long TYP=20;
        printf("donnez la CLE= de la queue ...\n");
        scanf ("%d", &CLE);
        /* récupération de l'identificateur . . . . */
        if ((msqid=msgget((key_t) CLE, 0 )) < 0)
        {       perror ("problème de récupération de l'identificateur");
                exit(1);
        };
        printf ("msgid =  %d\n", msqid);
        printf ("type du message     %d \n", msg.mtype);
        printf ("texte du message     %d \n", msg.mtext);
        if (((l=msgrcv(msqid, &msgp, 18, TYP, IPC_NOWAIT))< 0)))
        {       printf (" type %d\n", msgp.mtype);
                printf (" message     %s\n", msgp_mtext);
                exit(2);
        };
        printf ("message reçu:  %d\n", msgp.mtext);
        printf ("le type du message est %ld\n", msgp.mtype);
        exit(0);
        }
```

TP : essayer la lecture bloquante (NULL à la place de IPC_NOWAIT) et non bloquante.