Architecture et Système 2

Partie 1 :*Communication inter-processus*

Jalil BOUKHOBZA

UBO / Lab-STICC

Email : boukhobza@univ-brest.fr

(1ère version par P.Saliou)

PLAN DU COURS (partie système 2)

1. Introduction aux systèmes d'exploitation
2. Les processus
3. Concurrence et exclusion mutuelle

Prérequis

4. Communication inter processus — Cours 1
5. Threads — Cours 2
6. Ordonnancement — Cours 3
7. Gestion de la mémoire — Cours 4
8. Systèmes de fichiers — Cours 5
9. ...? — Cours 6

Evaluation de la partie système

- Contrôle continu:
  - Notes de TP
  - Micro projet
    - Mets en œuvre l'ensemble des notions de cours
    - Multi thread
    - Communication/synchronisation inter processus
    - Gestion de la mémoire
    - Ordonnancement
- Partielle de fin d'année

Partie 1 : Communication inter-processus

1. Présentation de la communication inter-processus
2. Les files de messages
3. Les segments de mémoire partagée
4. Les tubes de communication
5. Les signaux

1. Présentation de la communication inter-processus

- Des moyens de communication supplémentaires ou complémentaires aux sémaphores :
  - **Les fichiers normaux** .
  - **Les files de messages** qui permettent aux processus de s'échanger des informations à travers deux opérations :
    - *send (destinataire, message)*
    - *receive (source, message)*
  - **La mémoire partagée** qui permet à des processus de partager des mêmes pages physiques en mémoire.
  - **Les tubes** (ou pipe) *ordinaires* ou *nommés* qui sont des mécanismes de communication appartenant au système de gestion des fichiers.
  - **Les signaux** qui peuvent être assimilés à des interruptions logicielles.

2. Les files de messages

2.1. Principe des files de messages
2.2. Le producteur-consommateur avec échange de messages
2.3. Implémentation UNIX des files de messages

2.1 Principe des files de messages

- Les communications de messages se font à travers deux opérations fondamentales :
  - **send** (destinataire, message)
  - **receive** (source, message)
- Les messages sont de tailles variables ou fixes.
- Les opérations d'envoi et de réception peuvent être :
  - soit directes entre les processus,
  - soit indirectes par l'intermédiaire d'une « boîte aux lettres ».
- Les communications se font de manière synchrone ou asynchrone.

2.2. Le producteur et le consommateur (1)

- Deux processus se partagent un tampon de données de taille fixe (N) initialement vide :
  - Un premier processus (le **PRODUCTEUR**) produit des données et les écrit dans le tampon.
  - Un second processus (le **CONSOMMATEUR**) consomme les données du tampon en les lisant et en les détruisant au fur et à mesure de leur lecture.

2.2. Le producteur et le consommateur (2)

- Les problèmes surviennent lorsque :
  - Le producteur veut mettre des informations alors que la mémoire tampon est déjà pleine.
    - Il devra attendre (dormir) en attendant d'être réveillé par le consommateur.
    - Lorsque ce dernier aura retiré une ou plusieurs informations.
  - De la même manière si le consommateur tente de retirer une information alors que la mémoire tampon est vide.
    - Il devra dormir en attendant d'être réveillé par le producteur.
    - Lorsque ce dernier aura déposé une ou plusieurs informations.

2.2 Le producteur-consommateur avec échange de messages (3)

```
#include "prototypes h"
#define  N  100        /* Nb emplacements */
#define  TAILLE 4    /* taille du message */

typedef int message [TAILLE] ;

void producteur(void) {
     int objet;     /* tampon de messages */
     message m ;
     while (TRUE) {
          produire_objet(&objet);   /*produire l'objet suivant*/
          receive (consommateur, &m) ; /* attendre un message vide*/
          faire_message (&m, objet) ;  /*construire le message a envoyer*/
          send (consommateur, &m);  /* envoi de message au
                                       consommateur*/
     }
}
```

2.2 Le producteur-consommateur avec échange de messages (4)

```
#include "prototypes h"
#define  N   100        /* Nb emplacements */
#define  TAILLE 4    /* taille du message */
typedef int message [TAILLE] ;

void consommateur (void)   {
     int objet, i ;
     message m ;

     for (i=0 ;i <N ;i++)
          send (producteur,&m) ;
     while (TRUE) {
          receive (producteur, &m) ; /* attendre un message */
          retirer_objet(&m, &objet);      /*retirer l'objet du message*/
          send (producteur, &m); /*renvoyer un msg vide*/
          utiliser_objet (objet) ;
     }
}
```


2.2 Le producteur-consommateur avec échange de messages (5)

- Si le producteur travaille plus vite que le consommateur
  - Tous les messages se retrouveront pleins et attendront que le consommateur les consomme.
  - Le producteur se bloquera dans l'attente d'un message vide.
- Si le consommateur est le plus rapide :
  - Tous les messages seront vides et attendront que le producteur les remplisse.
  - Le consommateur sera bloqué tant qu'un message plein ne lui parviendra pas.

2.3 Implémentation UNIX des files de messages

2.3.1 La structure msqid_ds
2.3.2 La structure msgbuf
2.3.3 La primitive msgget
2.3.4 La primitive msgsnd
2.3.5 La primitive msgrcv
2.3.6 La primitive msgctl

2.3.1 La structure msqid_ds (1)

- Elle correspond à une entrée dans **la table des files de messages**.

```
struct msqid_ds {
        struct ipc_perm     msg_perm;   /*   droits d'acces
        struct __msg        *msg_first; /*   pointeur sur le 1er message
        struct __msg        *msg_last;  /*   pointeur sur le dernier message
        int                 msg_qnum;   /*   nombre de message dans la file
        …
        time-t              msg-stime;  /*   date de dernière émission (msgsnd)
        time-t              msg-rtime;  /*   date de dernière réception (msgrcv)
        time-t              msg-ctime;  /*   date de dernier changement par
msgctl
…

}
```

2.3.2 La structure msgbuf (1)

- La manipulation des files de messages suppose la définition d'une structure **spécifique à l'application** développée sur le modèle donné par ***msgbuf***.

```
struct msgbuf {
        long mtype ;        /* Type du message */
        char mtext [5] ;    /* Texte du message */
}
```

- Le type d'un message doit être strictement positif.
- La suite de la définition de la structure d'un message peut contenir des objets quelconques.

2.3.2 La structure msgbuf (2)

- Un exemple de structure **correcte**.

```
struct msgbuf_utilisateur_1 {
        long mtype ;
        float n1 ;
        int tab [4] ;
}
```

- Par opposition à une structure **incorrecte**.

```
struct msgbuf_utilisateur_2 {
        long mtype ;
        char *pt ;
}
```

2.3.3 La primitive msgget (1)

- Elle permet la création d'une nouvelle file ou la recherche de l'identification d'une file existante **à partir de sa *clé***.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgget (key_t cle,  int option) ;
```

- Le paramètre option est construit comme une combinaison des constantes : IPC_CREAT, IPC_EXCL et des droits d'accès.

Valeur de option	Explication
droits *(Clé=IPC_PRIVATE)*	Une nouvelle file privée est systématiquement créée.
IPC_CREAT | IPC_EXCL | droits	Création d'une nouvelle file associée à la **clé** passée en paramètre. Si la file existe déjà, une erreur est détectée.
IPC_CREAT | droits	Récupération de l'identification d'un file existante dont la **clé** est passée en paramètre. Si la file n'existe pas, elle est créée.

2.3.3 La primitive msgget (2)

- Exemple de création d'une file de message.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msqid;

msqid = msgget (ftok(« /opt/app/msg/essai», 1), IPC_CREAT | IPC_EXCL | 0666);
```

2.3.4 La primitive d'envoi msgsnd

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd ( int msqid, const void *p_msg, int size, int option) ;
```

- La primitive *msgsnd* envoi dans la file d'identification *msqid* le message *msgbuf* pointé par *p_msg.*
  - Le paramètre *size* correspond à la taille du message.
  - La primitive renvoie la valeur 0 en cas de succès et –1 sinon.
  - Le paramètre optionnel peut avoir la valeur *IPC_NOWAIT*. Dans ce cas si la file est pleine, l'appel à la primitive *msgsnd* n'est pas bloquant alors qu'il l'est par défaut.

2.3.5 La primitive d'envoi msgrcv

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv ( int msqid, void *p_msg, int size, long type,int option) ;
```

- La primitive *msgrcv* permet de lire un message dans la file identifiée par *msqid*.
  - Le pointeur *p_msg* pointe sur une zone mémoire susceptible de recevoir un message de longueur inférieure ou égale à *size*.
  - Le paramètre IPC_NOWAIT permet de ne pas bloquer si la file est vide.
  - Le type permet de spécifier le type du message à extraire :
    - > 0 le message le plus vieux de type égal à *type*.
    - 0 le message le plus vieux quel que soit son type.
    - < 0 le message le plus vieux du type le plus petit <= |*type*|
  - La valeur renvoyée en cas de réussite est la longueur du message (-1 sinon).

2.3.5 La primitive msgctl

- La primitive **msgctl** permet:
  - d'accéder aux informations contenues dans l'entrée de la table des files de messages,
  - et d'en modifier certains attributs.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgctl (int msqid, int op, /*struct msqid_ds p_msqid */) ;
```

- Quelques valeurs possibles du paramètre ***op*** sont :
  - IPC_RMID suppression de la file de messages.
  - IPC_STAT récupération des informations *msqid_ds* sur la file.
  - IPC_SET modification des informations de la file.

3. Les segments de mémoire partagée

3.1 Principe de la mémoire partagée
3.2 Le problème du producteur-consommateur avec la mémoire partagée
3.3 Implémentation UNIX des segments de mémoire partagée

3.1 Principe de la mémoire partagée (1)

- Avec les segments de mémoire partagée, les processus partagent des pages physiques (cadres) par l'intermédiaire de leur espace d'adressage.
  - Il n'y a donc plus de recopie des informations.
  - Les pages partagées par les processus sont des ressources critiques dont les accès doivent être protégés.
- Sous UNIX :
  - Il peut y avoir plusieurs segments partagés.
  - Un processus peut accéder à plusieurs segments de mémoire partagée.
  - Les segments de mémoire partagée ont une existence indépendante des processus.
  - Un processus doit attacher le segment à son espace d'adressage avant de pouvoir y accéder.

3.1 Principe de la mémoire partagée (2)

- L'utilisation d'un segment de mémoire partagée se fait habituellement de la manière suivante :

1. Un segment est d'abord créé à **l'extérieur** de l'espace d'adressage de tout processus par un premier processus.
2. Chaque processus voulant accéder au segment va exécuter une primitive pour l'attacher à son propre espace d'adressage.

3.2 Producteur-consomm. avec la mémoire partagée (1)

```
#define N 100

sema mutex = 1; /*sema d'exclusion mutuelle*/
sema production = N; /*sema places vides*/
sema consommation = 0; /*sema places pleines*/

typedef struct {
        int a_consommer ; /*prochain à consmmer*/
        int a_produire ;    /*prochain emplacement libre*/
        int tampon [N] ;    /*tampon de N entiers*/
} str_tampon ;
str_tampon *pt_shm ;

void consommateur() {
   pt_shm =PointeurSegmentMémoire ;
   while (1) {
       P(consommation);    /* une place pleine en moins*/
       P(mutex);          /*section critique*/
          vider_case (pt_shm->tampon [pt_shm->a_consommer]) ;
          pt_shm->a_consommer = (pt_shm->a_consommer +1)% N ;
       V(mutex);                /*fin de section critique* /
       V(production);           /* une place vide en plus*/
   }
 }
```


3.2 Producteur-consomm. avec la mémoire partagée (2)

```
#define N 100

sema mutex = 1;
sema production = N;
sema consommation = 0;

typedef struct {
        int a_consommer ;
        int a_produire ;
        int tampon [N] ;
} str_tampon ;
str_tampon *pt_shm ;

void producteur() {
   pt_shm =CréationSegmentMémoire ;
   while (1) {
       P(production);          /*une place vide en moins*/
       P(mutex);               /*section critique*/
          remplir_case (pt_shm->tampon [pt_shm->a_produire]) ;
          pt_shm->a_ produire = (pt_shm->a_produire +1 ) % N ;
       V(mutex);                /*fin de section critique* /
       V(consommation);     /* une place vide en plus*/
   }
}
```

3.3 Implémentation UNIX de la mémoire partagée

3.1 La structure shmqid_ds
3.2 La primitive shmget
3.3 La primitive shmat
3.4 La primitive shmdt
3.5 La primitive shmctl
3.6 Un exemple d'utilisation

3.3.1 La structure shmid_ds

- Elle correspond à une entrée dans la **table des segments de mémoire.**

```
struct shmid_ds {
        struct ipc-perm     shm-perm;   /*  droits d'acces */
        int                 shm-segsz;  /*  taille du segment en octets  */
        pid-t               shm-lpid;   /*  pid ayant fait la derniere opération*/
        pid-t               shm-cpid;   /*  pid createur */
        unsigned short int  shm-nattch; /*  nombre d'attachements  */
        time-t              shm-atime;  /*  date du dernier attachement  */
        time-t              shm-dtime;  /*  date du dernier détachement  */
        time-t              shm-ctime;  /*  date dernier changement par
shmctl*/
        }
```

3.3.2 La primitive shmget (1)

- Elle permet la création d'un nouveau segment ou la recherche de l'identification d'un segment existant.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int shmget (key_t cle, int taille, int option) ;
```

- Le paramètre *taille* spécifie la taille du segment.
- Le paramètre option est construit comme une combinaison des constantes : IPC_CREAT, IPC_EXCL et des droits d'accès.

Valeur de option	Explication
Droits (clé = IPC_PRIVATE)	Un nouveau segment de mémoire privé est systématiquement créé.
IPC_CREAT | IPC_EXCL | droits	Création d'un nouveau segment de mémoire associé à la **clé** passée en paramètre. Si le segment existe déjà, une erreur est détectée.
IPC_CREAT | droits	Récupération de l'identification d'un segment de mémoire existant dont la **clé** est passée en paramètre. Si le segment n'existe pas, il est créé.

3.3.2 La primitive shmget (2)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int idShm;
idShm = shmget(ftok("/opt/app/shm1", 1), sizeof(int), IPC_CREAT | IPC_EXCL | 0666);
```

3.3.3 La primitive d'attachement shmat

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
void *shmat(int shmid, const void *adr, int option);
```

- Un appel à la primitive *shmat* correspond à une demande d'attachement d'un segment identifié par *shmid* à l'adresse *adr* de l'espace d'adressage du processus.
  - La valeur de retour est l'adresse où l'attachement a été effectivement réalisé. Il s'agit d'un pointeur sur le premier octet du segment.
  - Si **adr = NULL** c'est le système d'exploitation qui choisit l'adresse.
  - Avec SHM_RDONLY dans *option* , le processus ne peut y accéder qu'en lecture.

3.3.4 La primitive de détachement shmdt

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int shmdt (const void *adr);
```

- La primitive *shmdt* correspond à une demande de détachement du segment de mémoire dont l'attachement été réalisé à l'adresse *adr*.
  - Si une demande de suppression du segment a été formulée et qu'après ce détachement le nombre d'attachements du segment devient nul, le segment est effectivement libéré.
  - La terminaison d'un processus entraîne le détachement de tous les segments qu'il a préalablement attachés.

3.3.5 La primitive shmctl

- La primitive *shmctl* permet
  - d'accéder aux informations contenues dans l'entrée de la table des segments de mémoire partagée,
  - et d'en modifier certains attributs.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int shmctl (int shmid, int op, struct shmid_ds
*p_shmid) ;
```

- Quelques valeurs possibles du paramètre *op* sont :
  - IPC_RMID suppression du segment de mémoire.
  - IPC_STAT récupération des informations dans shmid_ds.
  - IPC_SET modification des informations du segment.

3.3.7 Exemple d'utilisation d'un segment de mémoire partagée (1)

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

key_t cle;
int shmid, ind ;
int *p_shm_dwr, *p_shm_rdonly;

main(int argc, char *argv [ ]) {

        if ((cle=ftok(argv[1],0)) == -1) {
                fprintf (stderr,"erreur ftok(%s)\n" , argv[0]);
                exit(2);
        }

        if (shmid=shmget(cle, 256*sizeof(int), IPC_CREAT I IPC_EXCL | 0666)==-1) {
                perror("shmget");
                exit(3);
        }
        ...
```

3.3.7 Exemple d'utilisation d'un segment de mémoire partagée (2)

```
        ...

        If (p_shm_rdwr= (int *)shmat(shmid, NULL, 0) == -1) {
                perror("shmat_rdwr");
                exit(4);
        }

        If (p_shm_rdonly= (int *)shmat(shmid, NULL, SHM_RDONLY) == -1) {
                perror("shmat_rdonly");
                exit(5);
        }

        for(ind=0; ind<256; ind++)
                p_shm_rdwr[ind]=ind;

        for(ind=0; ind<256; ind++) {
                printf("%d ",p_shm_rdonly[indl]);
        putchar('\n ');

        exit (0) ;
}
```


4. Les tubes de communication avec Unix

4.1 Présentation des tubes de communication
4.2 Les tubes ordinaires
4.3 Les tubes nommés

4.1 Présentation des tubes de communication (1)

- Les tubes sont des mécanismes de communication appartenant au système de gestion de fichiers .
  - Ils sont désignés localement dans les processus par des descripteurs de fichiers.
  - Il sont manipulés au travers des primitives **read** et **write** **(comme les fichiers ordinaires)**.
- Quelques généralités sur les tubes :
  - Les tubes sont des mécanismes de communication **unidirectionnels**.
  - A un tube correspond **au plus 2 entrées** dans la table des fichiers.
  - L'opération de **lecture** dans un tube est **destructrice**.
  - Les tubes permettent la communication d'un flot continu de caractères.
  - La gestion des tubes est assurée en mode **FIFO**.

4.1 Présentation des tubes de communication (2)

- La différence entre un **tube ordinaire** ou un **tube nommé** est la suivante :
  - Un tube ordinaire est forcément créé depuis un processus et sa durée de vie ne peut excéder celle des processus l'utilisant.
  - Un tube nommé peut être accédé par n'importe quel processus connaissant sa référence (son nom) au travers de la primitive *open*.

4.2 Les tubes ordinaires (1)

- L'appel de la fonction du noyau **pipe()** crée un tampon de données, un tube, dans lequel deux processus pourront venir respectivement lire et écrire.
- La fonction fournit, d'autre part, **deux descripteurs** (df[2]) analogues aux descripteurs de fichiers.

```
#include <sys/unistd.h>
int pipe ( int df[2] ) ;
```

- Les deux processus communicants doivent partager les mêmes descripteurs obtenus par *pipe*. Il est donc nécessaire qu'ils aient un ancêtre commun car les descripteurs de fichiers sont hérités.

4.2 Les tubes ordinaires (2)

- Exemple1 : Le père écrit les lettres de l'alphabet que le fils lit

```
void main ()
{
        int df[2];                          /* descripteur du tube */
        char c, d;                          /* caractère de l'alphabet */
        int ret;
        pipe(df);                           /*  On crée le tube */
        if (fork() != 0) {                  /* On est dans le père */
                close (df[0]);                      /* Le père ne lit pas  */
                for (c = 'a'; c <= 'z'; c++) {
                        write(df[1], &c, 1);    /* Père écrit dans tube */
                }
                close(df[1]);               /* On ferme le tube en écriture */
                wait(&ret);                 /* Le père attend le fils */
        } else  {                             /* On est dans le fils */
                close(df[1]);               /* On ferme le tube en écriture */
                while (read(df[0], &d, 1) > 0) {    /*  Fils lit tube */
                        printf("%c\n", d);          /* Il écrit le résultat */
                }
                 close (df[0]);                     /* On ferme le tube en lecture */
        }
}
```

4.2 Les tubes ordinaires (3)

- Exemple2 : Utilisation du pipe par l'interpréteur de commande
  - Lorsque l'interprèteur de commande reconnaît une commande du type : " processus1 | processus2 ".

```
void synchro(char * processus1, char * processus2) {
        int df[2];                      /* descripteurs du tube */
        pipe(df);                       /* création du tube */
        if (fork() != 0) {
                close(df[0]);           /* fermeture lecture */
                close(1);               /* pas d'écriture sur l'écran (stdout) */
                dup(df[1]);             /* df[1] devient sortie standard */
                close(df[1]);           /* descripteur inutile après redirection */
                execl("processus1", "processus1", NULL);
        } else {
                close(df[1]);           /* fermeture écriture */
                close(0);               /* pas de lecture clavier */
                dup(df[0]);             /* df[0] devient entrée standard */
                close(df[0]);           /* descripteur inutile après redirection */
                execl("processus2", "processus2", NULL);
        }
}
```


4.3 Les tubes nommés

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo (const char *ref, mode_t mode) ;
```

- L'appel de la fonction du noyau ***mkfifo()*** crée un tube nommé au niveau du système.
  - Le paramètre *ref* définit le chemin d'accès au tube et le paramètre ***mode*** les droits d'accès.
  - L'ouverture d'un tube nommé par un processus se fera au moyen de la primitive *open ()*.

5. Les signaux

5.1 Présentation des signaux
5.2 L'envoi des signaux
5.3 Description de quelques signaux
5.4 La prise en compte des signaux
5.5 L'attente d'un signal

5.1. Présentation des signaux (1)

- Les signaux jouent pour les processus le rôle des sonneries susceptibles d'être entendues dans une maison (interruptions logicielle).
- Les signaux ne doivent être utilisés que pour des événements exceptionnels, et que très rarement pour des communications ordinaires.

5.1. Présentation des signaux (2)

- Les signaux diffèrent des messages :
  - Un signal peut être émis à tout moment, occasionnellement à partir d'un autre processus, mais plus souvent à partir du noyau comme conséquence d'un événement exceptionnel.
  - Un signal n'est pas nécessairement reçu ni pris en compte.
  - Par défaut, la plupart des signaux entraînent la terminaison du processus récepteur.
  - Un processus peut ignorer les signaux d'un type donné.
  - Les signaux n'ont pas de contenu informatif (vrai pour les dernières implémentations POSIX).
  - Le récepteur ne peut déterminer l'identité de l'émetteur.

5.2 L'envoi des signaux

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

- La primitive système ***kill()*** permet à un processus d'envoyer un *signal* à un autre processus dont le *pid* est passé en paramètre.
  - L'ensemble des processus destinataires du signal sera fonction du *pid* passé en argument :
    - >0 processus d'identité pid
    - 0 tous les processus appartenant au groupe du processus émetteur
    - <-1 tous les processus (sauf l'init)
  - Par défaut, la plupart des signaux provoquent la destruction du processus récepteur.

5.3 Description de quelques signaux

Signal	N°	Commentaires
SIGHUP	1	Signal émis lors d'une déconnexion
SIGINT	2	Il est émis à tout processus associé à un terminal de contrôle quand on appuie sur la touche d'interruption 'Ctrl-C'.
SIGQUIT	3	Semblable à SIGINT mais le signal est émis quand on appuie sur la touche d'abandon (normalement 'Ctl-\').
SIGILL	4	Instruction illégale.
SIGFPE	8	Erreur de calcul flottant.
SIGKILL	9	C'est la seule manière absolument sûre de détruire un processus, puisque ce signal est toujours fatal.
SIGALRM	14	Signal émis par alarm(int sec) au bout de sec secondes.
SIGTERM	15	Terminaison logicielle. Il s'agit du signal standard de terminaison.
SIGUSR1	16	Premier signal à la disposition de l'utilisateur.
SIGUSR2	17	Deuxième signal à la disposition de l'utilisateur.
SIGCLD	18	Ce signal est envoyé au père à la terminaison d'un processus fils.

5.4 La prise en compte des signaux (1)

- A chaque type de signal est associé, dans le système un ***handler*** par défaut désigné symboliquement par SIG_DFL.
- Ce *handler* définit le comportement par défaut pour chaque type de signal.
  - Terminaison du processus.
  - Terminaison du processus avec un image mémoire (core).
  - Signal ignoré.
  - Suspension du processus
  - Reprise d'un processus stoppé.
  - …
- Tout processus peut installer, pour chaque type de signal (excepté certains: **SIGKILL**, **SIGSTOP** et **SIGCONT**) un nouveau *handler*.

5.4 La prise en compte des signaux (2)

- Un processus peut modifier – " masquer " – son comportement aux signaux reçus par l'appel de la fonction signal ().

```
#include <signal.h>
void (* signal ( int sig, void (* p_handler)(int) ) )  ( int ) ;
```

- L'effet de la fonction est d'installer le handler spécifié par *p_handler* pour le signal *sig*.
- A l'arrivée du signal *sig*, le processus se comportera en fonction du *p_handler* fourni.

Nom	Action
SIG_IGN	Le processus ignorera l'interruption correspondante.
SIG_DFL	Le processus rétablira son comportement par défaut lors de l'arrivée de l'interruption.
void ***p_handler*** (int **sig**)	le processus exécutera la fonction ***p_handler***, définie par l'utilisateur, à l'arrivée de l'interruption **sig**. Il reprendra ensuite au point où il a été interrompu.

5.5 L'attente d'un signal (1)

```
#include <unistd.h>
int pause  (void) ;
```

- La primitive *pause()* permet de se mettre en attente de l'arrivée de signaux.

```
#include <unistd.h>
unsigned int alarm (unsigned int nb_sec) ;
```

- A l'aide de la fonction *alarm ()* un processus demande au système de lui envoyer le signal SIGALARM dans environ *nb_sec* secondes.
  - Le comportement par défaut d'un tel signal est la terminaison.
  - Un appel avec *nb_sec* = 0 **annule la demande antérieure**.

5.5 L'attente d'un signal (2)

- Terminaison d'un processus si l'opérateur n'a pas répondu dans un délai de 30 secondes.

```
void terminaison (int sig) {
        printf ("La réponse n'est pas arrivée à temps \n'') ;
        exit(0) ;
}

main () {
         int reponse ;
         signal (SIGALRM, terminaison) ;
         printf (''Veuillez répondre:\n'') ;
         alarm (30) ;
         scanf (''%d'', &reponse) ;
         alarm (0) ;
         printf (''Réponse lue \n'') ;
}
```