Architecture et Système 2

Partie 2: Les threads POSIX

Jalil BOUKHOBZA

UBO / Lab-STICC

Email : boukhobza@univ-brest.fr

Partie 2 bis : Les threads

1. Multithreading
2. Ordonnancement
3. Sémaphores, Mutex et variables de condition
4. Files de messages
5. Mémoires partagées
6. Signaux
7. E/S asynchrones

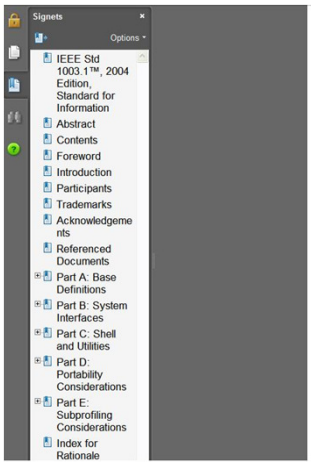
La norme POSIX

- 1ère version publiée entre 1988 et 1990
- **But:** Définition d'une méthode standard pour **interfacer** les applications avec les systèmes d'exploitation.
- L'ensemble inclut actuellement **plus de 30 standards** couvrant plusieurs sujets (gestion des processus, …, test de conformité de l'OS avec la norme).
- Standard concernant les interfaces du système: *(volume XSH) de l'IEEE Std 1003.1-2001* définissant un standard pour l'interface du système d'exploitation et l'environnement. Ce standard inclut une extension **temps réel**:
  - Fonctions et routines des services système
  - Services système spécifiques au langage C.
  - Notes sur la portabilité, gestion d'erreurs et récupération.

La norme POSIX (2)

Groupe fonctionnel	Quelques fonctions principales
Multi-Threads	pthread_create, pthread_exit, pthread_join, pthread_detach, pthread_equal, pthread_self
Ordonnancement de processus et de Threads	sched_setscheduler, sched_getscheduler, sched_setparam, sched_getparam, pthread_getschedparam, pthread_setschedprio, …
Signaux temps réel	sigqueue, pthread_kill, sigaction, sigaltstack, sigemptyset, etc
Synchro. et comm. inter processus	mq_open, mq_close, mq_receive, sem_init, sem_open, sem_wait, pthread_mutex_init, pthread_mutex_destroy, pthread_cond_init, shm_open, shm_unlink, mmap, etc.
Données spécifiques aux threads	pthread_key_create, pthread_get_specific, etc.
Gestion Mémoire	mlock, mlockall, munlock, munlockall, mprotect
E/S Async	aio_read, aio_write, aio_error, aio_return, aio_fsync, etc.
Horloges et minuteries (*timers*)	clock_gettime, clock_settime, clock_getres, times_create, timer_gettime, etc.
Annulation	pthread_cancel, pthread_setcancelstate, pthread_cleanup_push, pthread_cleanup_pop

Exemple



11175 None directly. However, as new areas are added, there will be a need for additional capability in
11176 this area.

11177 **D.2.2 Process Management**

11178 The *fork*(), *exec* family, *posix_spawn*(), and *posix_spawnp*() functions provide for the creation of
11179 new processes or the insertion of new applications into existing processes. The *_Exit*(), *_exit*(),
11180 *exit*(), and *abort*() functions allow for the termination of a process by itself. The *wait*() and
11181 *waitpid*() functions allow one process to deal with the termination of another.

11182 The *times*() function allows for basic measurement of times used by a process. Various
11183 functions, including *fstat*(), *getegid*(), *geteuid*(), *getgid*(), *getgrgid*(), *getgrnam*(), *getlogin*(),
11184 *getpid*(), *getppid*(), *getpwnam*(), *getpwuid*(), *getuid*(), *lstat*(), and *stat*(), provide for access to the
11185 identifiers of processes and the identifiers and names of owners of processes (and files).

11186 The various functions operating on environment variables provide for communication of
11187 information (primarily user-configurable defaults) from a parent to child processes.

11188 The operations on the current working directory control and interrogate the directory from
11189 which relative filename searches start. The *umask*() function controls the default protections
11190 applied to files created by the process.

11191 The *alarm*(), *pause*(), *sleep*(), *ualarm*(), and *usleep*() operations allow the process to suspend until
11192 a timer has expired or to be notified when a period of time has elapsed. The *time*() operation
11193 interrogates the current time and date.

11194 The signal mechanism provides for communication of events either from other processes or
11195 from the environment to the application, and the means for the application to control the effect
11196 of these events. The mechanism provides for external termination of a process and for a process
11197 to suspend until an event occurs. The mechanism also provides for a value to be associated with
11198 an event.

11199 Job control provides a means to group processes and control them as groups, and to control their
11200 access to the function between the user and the system (the "controlling terminal"). It also
11201 provides the means to suspend and resume processes.

11202 The Process Scheduling option provides control of the scheduling and priority of a process.

Généralités sur les threads

- Donner plusieurs activités à un même processus
  - Mener plusieurs traitements parallèles
  - Gagner du temps sur une machine multi-processeurs
- Plus efficace que plusieurs processus
  - Commutation plus efficace (processus légers)
  - Espace d'adressage commun (communication simplifiée)
- Informations propres à chaque thread
  - Pile d'exécution, valeurs des registres
- Informations partagées au sein du processus
  - Code, données, descripteurs de fichiers
- Partage dépendant de l'implémentation
  - Signaux, propriétés . .

1. Multithreading

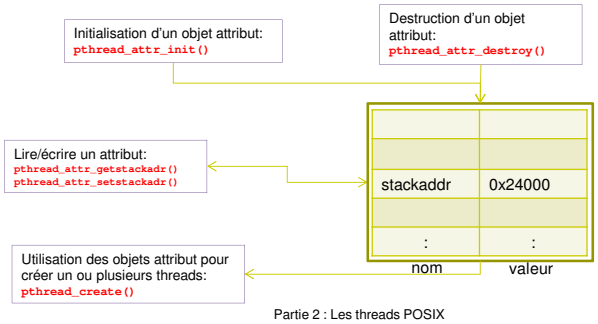
Création de thread pour **l'exécution d'une fonction** `pthread_create()`

```
int pthread_create(pthread_t * thread,
                   pthread_attr_t * attr,
                   void * (*start_routine)(void *),
                   void * arg);
```

- Possibilité de passer un argument à la fonction exécutée par le thread (*arg*).
- Possibilité (optionnel) de spécifier un objet attribut (taille et adresse du *stack*, politique d'ordonnancement) → **attr*
- Terminaison du thread:
  - Retour à/de la fonction principale
  - Appel explicite à `void pthread_exit(void *retval);`
  - Acceptation d'une requête d'annulation
- `int pthread_join(pthread_t thread, void **value_ptr)` : attente de la terminaison d'un autre thread

Les objets attributs

- Mécanismes conçus pour supporter les normalisations futures ainsi que des extensions portables de quelques entités spécifiées par le standard POSIX sans que les fonctions qui opèrent sur ces dernières ne changent; threads, dispositifs d'exclusion mutuelle, variables de conditions.
- Centralisation de certains attributs **par type d'objets** et non pas **par instance**.



Exemple

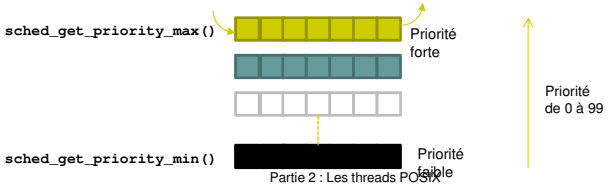
```
int sum;//donnée partagée par les threads

void *runner(void *param); //prototype du
   thread
main(int argc, char *argv[])
{
   void *val_ret;
   pthread_t tid; // l'identifiant du
   thread
   pthread_attr_t attr; //ensemble
   d'attributs du thread
   // Avoir les attributs par défaut
   (optionnel)
   pthread_attr_init(&attr);
   // création du thread
   pthread_create(&tid,&attr,runner,argv
   [1]);
   // attente que le thread se termine
   pthread_join(tid, &val_ret);
   printf("%s →sum = %d\n",(char *)
   &val_ret, sum);
}
```

```
void *runner(void *param) {
   int upper = atoi(param);
   int i;
   sum = 0;
   char *c;
   if (upper > 0) {
      for (i = 1; i <= upper;
   i++)
         sum += i;
   }
   c=strdup("ok");
  // terminaison du thread
   pthread_exit(c);
}
```

9

2. Ordonnancement de threads et de processus

- **L'interface POSIX** propose trois politiques différentes, **une** pour les processus classiques (conventionnels) temps partagé, et **deux** pour les applications à vocation temps-réel.
- Une valeur de priorité statique ***sched_priority*** est assignée à chaque processus (modif: appels sys)
- Conceptuellement, l'ordonnanceur dispose d'une liste de tous les processus prêts pour chaque valeur possible de *sched_priority* (intervalle 0 à 99).



10

Ordonnancement de threads et de processus (2)

- Politiques d'ordonnancement:
  - **SCHED_FIFO**:
    - Priorités statiques > 0 (privilèges super utilisateur)
    - Arrêt: bloqué par une opération d'entrée/sortie OU préempté par un processus de priorité supérieure OU appel ***sched_yield***.
  - **SCHED_RR** (Round Robin)
    - Priorités statiques > 0
    - Quantum de temps (lecture avec **`sched_rr_get_interval)`**
  - **SCHED_OTHER**
    - Priorités statiques =0
    - Utilisation d'une priorité dynamique basée sur la valeur de « *gentillesse* » du processus (**nice**, **setpriority**) → incrémentée à chaque *time quantum* où le processus est prêt mais non sélectionné par l'ordonnanceur. Garantit d'une progression équitable de tous les processus *SCHED_OTHER*.

Ordonnancement de threads et de processus (3)

- Fonctions permettant de sélectionner la politique d'ordonnancement, configurer et lire les paramètres de cette dernière.

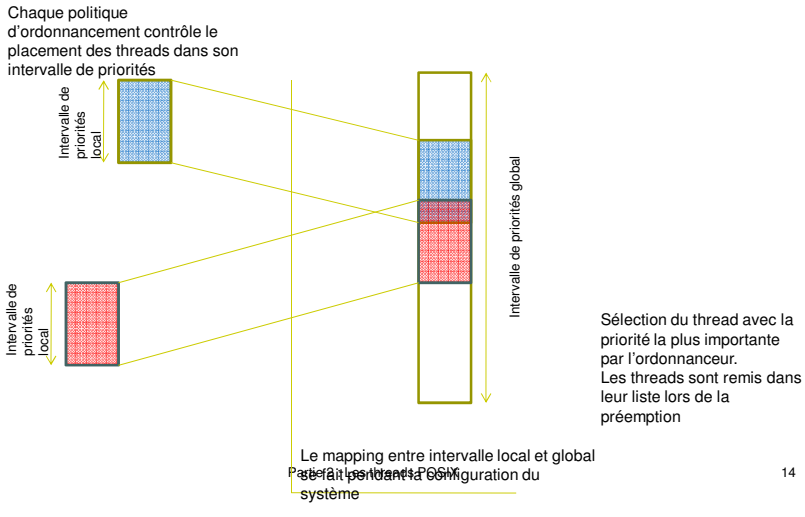
- Lire / fixer la ***politique*** d'ordonnancement des processus et ses paramètres:
  - `int sched_setscheduler(pid_t pid, int policy, const struct sched_param *p),`
  - `int sched_getscheduler(pid_t pid)`
  - `struct sched_param { ... int sched_priority; ... };`

- Lire / écrire les ***paramètres*** d'ordonnancement (la priorité)
  - `int sched_setparam(pid_t pid, const struct sched_param *p)`
  - `int sched_getparam(pid_t pid, struct sched_param *p).`

Ordonnancement de threads et de processus (4)

- **int pthread_setschedparam(pthread_t** *thread_cible*, **int** *politique*, **const struct sched_param *****param**),
- **int pthread_getschedparam(pthread_t** *thread_cible*, **int *****politique**, **struct sched_param *****param**);
- **int pthread_setschedprio(pthread_t** *thread_cible*, **int** *prio*) (accès dynamique; utilisé avec un thread existant)
- Modification dans l'objet attribut (et non pas pour un thread particulier) → utilisable pour la création d'autres threads:
  - **int pthread_attr_setschedpolicy(pthread_attr_t *****attr**, **int** *policy*);
  - **int pthread_attr_getschedpolicy(const pthread_attr_t *****attr**, **int *****policy**);
  - **int pthread_attr_setschedparam(pthread_attr_t *****attr**, **const struct sched_param *****param**);
  - **int pthread_attr_getschedparam(const pthread_attr_t *****attr**, **struct sched_param *****param**);
- Politiques d'ordonnancement: FIFO, Tourniquet, ou autre.

Ordonnancement sous POSIX



Synchronisation et communication inter processus

- Principaux mécanismes: sémaphores et files de messages
- Mécanismes bloquants et non bloquants
- Support multithreading:
  - Dispositifs d'exclusion mutuelle
  - Variables de condition
  - ….

3. Les sémaphores

- Non nommés
  - `int sem_init(sem_t *sem, int pshared, unsigned int valeur)` : création
  - `int sem_destroy(sem_t * sem)` : suppression de l'association entre descripteur et sémaphore + suppression de sémaphore.
- Nommés
  - `sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value)` : création
  - `int sem_close(sem_t *sem)` : suppression de l'association entre descripteur et sémaphore
  - `int sem_unlink(const char *name)` : suppression de sémaphore
- `int sem_wait(sem_t * sem)` → P(); `int sem_trywait(sem_t * sem)`, `int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout)`
- `int sem_post(sem_t * sem)` → V();
- `int sem_getvalue(sem_t * sem, int * sval)` : retourne la valeur du sémaphore.

Les sémaphores (2)

```
#include <semaphore.h>
#define S_MODE S_IRUSR | S_IWUSR      //permissions de lecture et d'écriture pour le
    propriétaire
…
void main();
{
     if ((my_lock = sem_open ("my.dat", O_CREAT|O_EXCL,S_MODE, 1) == -1) && errno== ENOENT)
     {
          perror("semaphore open failed"); exit(1);
     }
     …
     for (i = 1; i < n; ++i)
          if (childpid= fork()) break;

     …
     if (sem_wait (&my_lock) == -1) {
          perror("semaphore invalid); exit (1);
     }
     //Critical Section
     if (sem_post (&my_lock) == -1) {
          perror("semaphore done"); exit(1);
     }
     …
     if (sem_close (&my_lock) == -1) {
     perror("semaphore close failed"); exit(1);
     }
}
```


Mutexes

- Sémaphore binaire → assure une exclusion mutuelle entre **plusieurs threads**.
- `int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr)` : initialisation d'une mutex.
- `int pthread_mutex_destroy(pthread_mutex_t *mutex)` : suppression d'une mutex
- `int pthread_mutex_lock(pthread_mutex_t *mutex)` : équivalent d'un P() sur un sémaphore binaire.
  - `int pthread_mutex_trylock(pthread_mutex_t *mutex)`
- `int pthread_mutex_unlock(pthread_mutex_t *mutex)` : équivalent d'un V() sur un sémaphore binaire.

Mutex (2)

```
1   /* leger_mutex.c */
2   #include <pthread.h>
3   #include <stdio.h>
4   #include <unistd.h>
5   struct DonneesTache {
6      char *chaine;            /* chaine à écrire      */
7      int nombre;              /* nombre de répétitions */
8      int delai;               /* délai entre écritures */
9   };
10  int numero = 0;
11  pthread_mutex_t verrou;     /* protège l'accès à numero */
12  void *executer_tache(void *data)
13  {
14     int k;
15     struct Donnees *d = data;
16     for (k = 0; k < d->nombre; k++) {
17        pthread_mutex_lock(&verrou); /* DEBUT SECTION CRITIQUE */
18        numero++;
19        printf("[%d] %s\n", numero, d->chaine);
20        pthread_mutex_unlock(&verrou);    /* FIN SECTION CRITIQUE
    */
21        sleep(d->delai);
22     };
23     return NULL;
24  }
25  int main(void)
26  {
27     pthread_t t1, t2;
28     struct DonneesTache d1, d2;
29     d1.nombre = 3;
30     d1.chaine = "Hello";
31     d1.delai = 1;
32     d2.nombre = 2;
33     d2.chaine = "World";
34     d2.delai = 2;
35     pthread_mutex_init(&verrou, NULL);
36     pthread_create(&t1, NULL, executer_tache, (void *)
    &d1);
37     pthread_create(&t2, NULL, executer_tache, (void *)
    &d2);
38     pthread_join(t1, NULL);
39     pthread_join(t2, NULL);
40     pthread_mutex_destroy(&verrou);
41     printf(" %d lignes.\n", numero);
42     exit(0);
43  }
```


Variables de condition

- Les variables de condition peuvent être utilisées pour bloquer **atomiquement** les threads jusqu'à ce qu'une condition particulière se réalise.
- Les variables de condition sont toujours utilisées avec des mutex (variables d'exclusion mutuelle).
- La condition est testée sous la protection d'une mutex:
  - Lorsque la condition est fausse, le thread bloque sur la variable de condition et relâche atomiquement la mutex en attendant le changement de condition.
  - Lorsqu'un autre thread change de condition, il peut spécifier à la variable de condition s'il veut qu'un seul ou plusieurs threads en attente se réveille(nt), prenne(nt) la mutex et réévalue la condition.

```
P(mutex)
   si condition ne se réalise pas
   attente
V(mutex)
```

Variables de condition (2)

- `int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr)` : Initialisation des variables conditionnelles.
- Utilisation de *mutex* et de variable de condition
  - Chaque procédure appartenant à un moniteur doit être entourée par un verrouillage de la *mutex* au début et un déverrouillage à la fin.
  - Pour ce bloquer sur une variable de condition, un appel à la fonction `int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)` doit être effectué en prenant en paramètre la variable et la *mutex* utilisée. Cette fonction déverrouille automatiquement la *mutex* et se bloque sur la variable de condition; la *mutex* est reprise lorsque le thread est débloqué (possibilité d'utiliser `int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime)`).
  - Dans une procédure appartenant au moniteur, la fonction `int pthread_cond_signal(pthread_cond_t *cond)` peut être appelée pour débloquer au moins l'un des threads bloqués sur la variable de condition. `int pthread_cond_broadcast(pthread_cond_t *cond)` peut être utilisée pour débloquer tous les threads.
- Destruction en utilisant `int pthread_cond_destroy(pthread_cond_t *cond)`


```
#include <pthread.h>
/* définition du tampon */
#define N 10 /* Nb de cases du tampon */
#define NbMess 20 /* Nb de messages échangés */
int NbPleins=0;
int tete=0, queue=0;
int tampon[N];
/* définition des conditions et du mutex */
pthread_cond_t vide;
pthread_cond_t plein;
pthread_mutex_t mutex;
pthread_t tid[2];
void Deposer(int m){
    pthread_mutex_lock(&mutex);
    if(NbPleins == N) pthread_cond_wait(&plein, &mutex);
    tampon[queue]=m;
    queue=(queue+1)%N;
    NbPleins++;
    pthread_cond_signal(&vide);
    pthread_mutex_unlock(&mutex);
}
int Prelever(void){
    int m;
    pthread_mutex_lock(&mutex);
    if(NbPleins ==0) pthread_cond_wait(&vide, &mutex);
    m=tampon[tete];
    tete=(tete+1)%N;
    NbPleins--;
    pthread_cond_signal(&plein);
    pthread_mutex_unlock(&mutex);
    return m;
}
```

Déclaration des variables de condition

1ère procédure du moniteur

2ème procédure du moniteur

```
void * Prod(void * k) /********** PRODUCTEUR */
{
    int i; int mess;
    srand(pthread_self());
    for(i=0;i<=NbMess; i++){
    usleep(rand()%10000); /* fabrication du message */
    mess=rand()%1000;
    Deposer(mess);
    printf("Mess depose: %d\n",mess);
    }
}
void * Cons(void * k) /********** CONSOMMATEUR */
{
    int i; int mess;
    srand(pthread_self());
    for(i=0;i<=NbMess; i++){
        mess=Prelever();
        printf("\tMess preleve: %d\n",mess);
        usleep(rand()%1000000); /* traitement du message */}
}
void main() /* M A I N */
{
    int i, num;
    pthread_mutex_init(&mutex,NULL);
    pthread_cond_init(&vide,NULL);
    pthread_cond_init(&plein,NULL);
    /* creation des threads */
    pthread_create(tid, 0, Prod, NULL);
    pthread_create(tid+1, 0, Cons, NULL);
    // attente de la fin des threads
    pthread_join(tid[0],NULL);
    pthread_join(tid[1],NULL);
    // libération des ressources
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&vide);
    pthread_cond_destroy(&plein);
    exit(0);}
```

Entrée 1

Entrée 2

Initialisation

4. Files de messages

- Quelques caractéristiques:
  - **Priorité** à l'émission / réception
  - Fonctionnement bloquant ou non bloquant
  - Fonctionnement entre processus et threads.
- Externe au système de gestion des fichiers
- Gestion d'une table spécifique par type d'objet
- `mqd_t mq_open(const char *name, int oflag)`: création ou ouverture d'une file de message retournant un descripteur utilisé par toutes les autres fonctions.
- `mqd_t mq_close(mqd_t mqdes)`: ferme le descripteur de la file de message.

Files de messages (2)

- `mqd_t mq_unlink(const char *name)` : suppression d'une file de msg si aucun processus n'y accède, sinon → attente avant suppression.
- Le nombre de msg et leur taille sont des paramètres constants pour la durée de vie d'une file de msg et sont définis à la création.
- `mqd_t mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned msg_prio)` et `ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned *msg_prio)` : envoi et réception bloquants par défaut. Variante: `mq_timedsend()` & `mq_timedreceive()`.
- Possibilité de mettre des **priorités** sur les messages.
- `mqd_t mq_notify(mqd_t mqdes, const struct sigevent *notification)` : notifie lors de la transition file de msg vide → file de msg non vide.
- `mqd_t mq_getattr(mqd_t mqdes, struct mq_attr *attr), mqd_t mq_setattr(mqd_t mqdes, struct mq_attr *newattr, struct mq_attr *oldattr)` : lecture/écriture des attributs d'une file de msg.

Files de messages (3)

```
#include <mqueue.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#define MSG_SIZE      4096

// Cette fn est appelée lorsque la file devient non vide
void handler (int sig_num) {
   printf ("sig %d recu.\n", sig_num);
}

void main () {

  struct mq_attr attr, old_attr;   // sauvegarde des attr
        de la file
  struct sigevent sigevent;        // pour notification
  mqd_t mqdes, mqdes2;             // descripteur de file de
        msg
  char buf[MSG_SIZE];              // taille du buffer
  unsigned int prio;               // Priority

  // First we need to set up the attribute structure
  attr.mq_maxmsg = 300;
  attr.mq_msgsize = MSG_SIZE;
  attr.mq_flags = 0;

  // Open a queue with the attribute structure, les 2
        derniers paramètres ne sont pas obligatoires
  mqdes = mq_open ("/lafile", O_RDWR
        |O_CREAT,  0664, &attr);
  // ouverture d'une file avec les attr par défaut
  mqdes2 = mq_open (" /autrefile", O_RDWR |
        O_CREAT, 0664, 0);
     // une file temporaire, dès qu'elle sera fermée, elle
        sera supprimée
  mq_unlink (" /autrefile ");

  // demader les attr de la file de msg lafile
  mq_getattr (mqdes, &attr);
  printf ("%d messages sont actuellement dans la
        file.\n",
        attr.mq_curmsgs);
if (attr.mq_curmsgs != 0) {

  // Il existe des msg dans la file....à consommer

  // spécifier que la file ne se bloque à aucun appel
  attr.mq_flags = O_NONBLOCK;
  mq_setattr (mqdes, &attr, &old_attr);

  // consommer tous les messages
  while (mq_receive (mqdes, &buf[0], MSG_SIZE,
        &prio) != -1)
    printf ("Received a message with priority %d.\n",
        prio);

  // l'appel a échoué, s'assurer que errno est EAGAIN
  if (errno != EAGAIN) {
    perror ("mq_receive()");
    _exit (EXIT_FAILURE);   }

  // restaurer les attributs
    mq_setattr (mqdes, &old_attr, 0);
  }

  // notifier lorsque quelque chose est dans la
file
  signal (SIGUSR1, handler);
  sigevent.sigev_signo = SIGUSR1;

  if (mq_notify (mqdes, &sigevent) == -1) {
    if (errno == EBUSY)
      printf (
        « un autre processus veut recevoir cette
notif.\n");
    _exit (EXIT_FAILURE);
  }

  for (prio = 0; prio <= MQ_PRIO_MAX; prio
+= 8) {
    printf (« ecrire un message avec une
priorité %d.\n", prio);
    if (mq_send (mqdes, "I8-)", 4, prio) == -1)
      perror ("mq_send()");
  }

  // Fermeture de tous les descripteurs
  mq_close (mqdes);
  mq_close (mqdes2);
}
```


5. Mémoire partagée

- **`int shm_open(const char *`***`nom`***`, int `***`oflag`***`, mode_t `***`mode`***`)`**: crée et/ou ouvre une zone de mémoire partagée et l'associe à un descripteur de fichier.
- **`int ftruncate(int `***`fd`***`, off_t `***`taille`***`)`**: augmente la taille du segment partagé
- **`int close(int `***`fd`***`)`**: supprime l'association entre le descripteur et la mémoire partagée associée.
- **`int shm_unlink(const char *`***`nom`***`)`** : détruit l'objet de mémoire partagée à condition qu'il n'y ai aucun processus utilisant cette mémoire.
- Le mapping de la mémoire partagée dans l'espace d'adressage du processus n'est pas effectué avec un **`shm_open()`** → **`void *mmap(void *`***`start`***`, size_t `***`length`***`, int `***`prot`***`, int `***`flags`***`, int `***`fd`***`, off_t `***`offset`***`)`** (l'inverse est effectué avec **`int munmap(void *`***`start`***`, size_t `***`length`***`)`**)

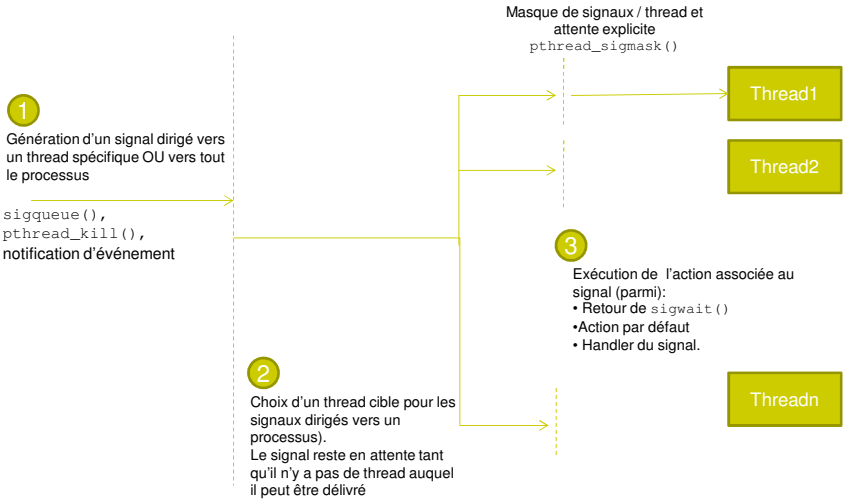
Gestion de la mémoire

- Possibilité de verrouiller des parties de l'espace d'adressage → `int mlock(const void *addr, size_t len)` : verrouillage à partir d'une adresse et d'un offset et `int mlockall(int flags)` : tout l'espace.
- Verrouillage: force la présence des données dans la mémoire centrale et leur éviter d'être « flushées » sur le stockage secondaire:
  - Essentiel pour le temps réel → prédictibilité + performance
- Déverrouillage de la mémoire avec `int munlock(const void *addr, size_t len)` et `int munlockall(void)`.
- Mapping d'un objet mémoire (shm) sur l'espace d'adressage → `void *mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset)`
- Protéger des emplacements mémoire → `int mprotect(const void *addr, size_t len, int prot)`

6. Signaux temps réel et événements asynchrones.

- Utilisés dans les cas suivants:
  - Erreur pendant l'exécution d'un programme (référence mémoire erronée)
  - Panne système ou hw (panne électrique)
  - Génération explicite d'un événement.
  - Achèvement d'une requête d'E/S
  - Disponibilité de données à lire dans une file de msg
  - …
- Par rapport aux signaux ISO C, la norme IEEE Std 1003.1-2001 permet de:
  - spécifier une hiérarchie de priorités pour les différents signaux
  - ne pas ignorer plusieurs signaux de même type qui arrivent tout en les dissociant → mise en file (politique de service des signaux: FIFO).
  - D'envoyer un ensemble (très restreint de données) avec le signal.

Vie des signaux POSIX



Génération du signal

- La plupart des signaux sont générés par le système
- Peuvent être adressés au thread **OU** au processus …
- Génération d'un signal:
  - Fonction **`int sigqueue(pid_t pid, int sig, const union sigval valeur);`**
    - Processus de destination
    - Numéro du signal
    - Une quantité limitée d'information `union sigval { int sival_int; void *sival_ptr; };`
  - **`int killpg(int pgrp, int sig);`** : envoi de signal vers un groupe de processus pgrp
    - Pas d'information supplémentaire (en plus du numéro de signal)
  - **`int pthread_kill(pthread_t thread, int signo);`** envoi d'un signal vers un thread du processus appelant (impossible vers un autre processus)

Action au niveau du processus

- Définition d'une action pour chaque type de signal et pour chaque processus → `int sigaction (int signum, const struct sigaction * act, struct sigaction *oldact)` :
  - Ignorer le signal
  - Action par défaut effectuée par l'OS (eg. terminaison du processus)
  - Exécution d'une fonction (*handler*) définie par le programmeur.

```
struct sigaction { void (*sa_handler)(int); // utilisé pour des raisons de compatibilité
    void (*sa_sigaction)(int siginfo_t *, void *); // siginfo contient la valeur envoyée par le
    processus, l'identifiant de ce dernier, la référence temporelle de l'envoi …
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void); // obsolète }
```

- Spécification de drapeaux associés au numéro de signal (en particulier pour le temps réel):
  - SA_SIGINFO: permet le rajout d'une information limitée lors de l'envoi de chaque signal.
  - SA_RESTART: permet le redémarrage automatique des appels système interruptible à la réception de ce signal.
  - SA_ONSTACK: permet de rediriger l'adresse de la pile du thread ou processus vers lequel le signal est envoyé (la fonction `int sigaltstack(const stack_t *ss, stack_t *oss)` permet de définir ce dernier).
- La définition du *handler* se fait au niveau du processus et non du thread.

Livraison et acceptation du signal

- Signal peut être **« délivré à »** (le cas d'un envoi de signal) ou **« accepté par »** (le cas d'une réception en utilisant `sigwait()`) un thread ou un processus.

- Mise en place d'un **masque** permettant à un thread de bloquer un signal:
  - `int pthread_sigmask(int how, const sigset_t *newmask, sigset_t *oldmask)` : lire/écrire le masque.
  - `int sigemptyset(sigset_t *set)` : vider l'ensemble des masques (aucun signal bloqué)
  - `int sigfillset (sigset_t *set)` : remplir l'ensemble des masques (tous les signaux sont bloqués)
  - `sigaddset (sigset_t *set, int signum)` : rajout d'un signal à l'ensemble des masques
  - `int sigdelset (sigset_t *set, int signum)` : suppression d'un signal à l'ensemble des masques
  - `sigismember (const sigset_t *set, int signum)` : teste si un signal appartient à un ensemble de masques
- `int sigwaitinfo(const sigset_t *set, siginfo_t *info)` → info sur le signal *accepté* & `sigtimedwait()` → attente à durée limitée
- **Un seul thread** reçoit le signal envoyé (qu'il soit envoyé au processus ou à un thread)

7. E/S asynchrones

- **E/S synchrones**: lecture d'un fichier → envoi de la requête et attente du résultat (une seule requête pour un processus à un moment donné):
  - Temps d'achèvement d'une E/S: imprévisible
  - Parallélisme au niveau des E/S non exploité.
  - → mauvais pour le temps réel
- **E/S asynchrones** → lancement de plusieurs E/S et réception asynchrone des acquittements. Bloc de contrôle des E/S asynchrones (`struct aiocb`) contenant toutes les informations permettant de décrire une E/S, on peut:
  - Spécifier le type d'opération(Lec/Ecr)
  - Identifier le fichier sur lequel l'opération doit se faire.
  - Déterminer la portion de fichier considérée
  - Localiser un tampon de données à utiliser
  - Donner des priorités aux opérations
  - Demander la notification de l'achèvement de la requête par un signal ou par l'exécution d'une fonction.

struct aiocb

- int aio_fildes; /* file descriptor */
- volatile void *aio_buf; /* buffer location */
- size_t aio_nbytes; /* length of transfer */
- off_t aio_offset; /* file offset */
- int aio_reqprio; /* request priority offset */
- struct sigevent aio_sigevent; /* signal number and offset */
- int aio_lio_opcode; /* listio operation */
- int aio_flags; /* flags */

E/S asynchrones (2)

- `int aio_read(struct aiocb *aiocbp)` et `int aio_write(struct aiocb *aiocbp)`: requête de lec/ecr en prenant en paramètre un bloc de contrôle d'E/S.
- `lio_listio()`: prépare une liste d'E/S chacune définie par un bloc de contrôle d'E/S.
- `int aio_error(const struct aiocb *aiocbp)` et `ssize_t aio_return(struct aiocb *aiocbp)` : permettent de récupérer les erreurs ou les informations relatives à un bloc de contrôle d'E/S après achèvement de ces dernières.
- `int aio_fsync(int op, struct aiocb *aiocbp)` : provoque une synchronisation de toutes les opérations d'E/S asynchrones en cours associées au bloc de contrôle.
- `int aio_suspend(const struct aiocb * const cblist[], int n, const struct timespec *timeout)` : bloque un thread jusqu'à ce qu'une des E/S spécifiées en argument s'achève (ou sinon jusqu'à un temps maximum).
- `int aio_cancel(int fd, struct aiocb *aiocbp)` : annule une opération d'E/S qui n'a pas été achevée.

Annulation

- Possibilité pour un thread d'annuler un autre thread avec la fonction `int pthread_cancel(pthread_t thread)`;.
- Chaque thread décide de la façon avec laquelle il réagit face à une requête d'annulation via `int pthread_setcancelstate(int state, int *oldstate)`, `int pthread_setcanceltype(int type, int *oldtype)` . Le thread peut:
  - Ignorer les requêtes d'annulation
  - les accepter immédiatement
  - les accepter au-delà d'un certain point dans le code (défini par void `pthread_setcancelstate()`)
- Éviter d'annuler un thread qui détient une ressource (sémaphore).