

# Systemes d'exploitation pour l'embarqué–communication et synchronisation inter-processus

Cours 4 - Master 2 LSE  
Jalil Boukhobza  
Université de Bretagne Occidentale – Lab-STICC

## Plan

1. Introduction
2. Les IPCs
3. Les IPCs sous VxWorks
4. La norme POSIX

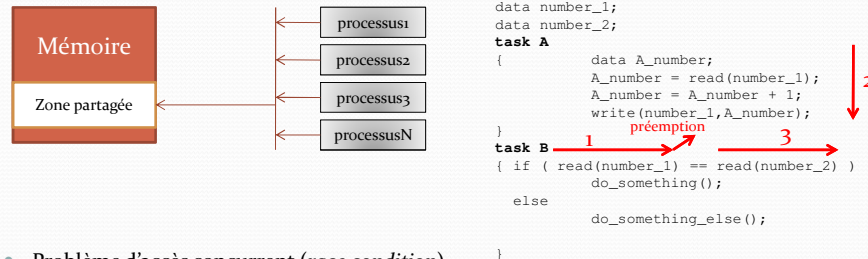
## Communication et synchronisation interprocessus

- Les tâches (dans un système multi tâches) dans un OS embarqués ont AUSSI besoin de communiquer et se synchroniser.
- **Communication**: implique au moins 2 tâches (receveur et émetteur, lecteur et rédacteur ou producteur et consommateur)
  - Communiquer efficacement sans avoir à connaître grand chose de l'autre.
- **Synchronisation**: pas d'ordre ou de direction
  - Être sûr que les tâches se trouvent (ou pas) à des endroits spécifiques dans leur code respectif en même temps.
- Une communication utile n'a lieu que si la synchronisation est efficace
- Communication/synchronisation:
  - Bloquante → communication synchrone
  - Non bloquante → communication asynchrone
  - Variantes:
    - Blocage pendant une durée max (timeout)
    - Blocage conditionnel

## Inter Process Communication (IPC); propriétés

- **Couplage**
  - **Connexions nommées**: Les tâches émettrices et celles réceptrices doivent se connaître et s'adresser par leurs identifiants respectifs.
    - Un à un, un à plusieurs, plusieurs à un ou plusieurs à plusieurs.
  - **Diffusion (broadcast)**: l'émetteur envoie son message à tout ceux qui « écoutent » et il ne sait pas quelle tâche l'a écouté. Ecoute qui le veut. Les messages sont perdus s'ils ne sont pas écoutés.
  - **Tableau noir (blackboard)**: similaire à la diffusion sauf que les messages sont sauvegardés quelque part.
  - **ORB (Object Request Broker)**: l'émetteur enregistre son interface avec l'ORB et les receveurs intéressés peuvent demander au « broker » de transférer leur requêtes au serveur appropriés sans avoir à connaître son identité.
- **Mise en tampon**
  - **Avec**: mise en tampon des messages
  - **Sans**: si message non reçu à l'envoi, il est perdu

## Accès concurrent à une zone partagée



- Problème d'accès concurrent (*race condition*)
- Processus préempté avant d'avoir fini un accès en écriture
  - Intégrité du système n'est plus garantie
- L'accès est alors une **section critique**
- On doit accéder à la zone en **exclusion mutuelle**
  - L'exclusion mutuelle permet à un processus de verrouiller l'accès à une ressource pendant que lui y accède.
  - Technique de synchronisation.

## Sections critiques

- **3 types** de sections critiques :
  - Accès aux mêmes **données** par des tâches différentes
  - Accès à un **service**: allocation d'une ressource, transaction BD → atomique
  - Accès à un code de **procédure**: quelques portions de code ne doivent pas être **réentrantes**.
- Les **problèmes d'accès concurrent**:
  - **Interblocage (deadlock)** toutes les ressources sont prises et toutes les tâches ont besoin de plus de ressources pour poursuivre leur exécution.
  - **Livelock** : les tâches ne sont pas bloquées mais demandent en continu des ressources qu'elles n'obtiennent pas.
  - **Famine (starvation)** : une tâche ne réussit jamais à obtenir la ressource qu'elle demande.

## Conditions pour un inter blocage

1. L'accès aux ressources se fait en exclusion mutuelle
2. Une tâche qui a une ressource peut en demander une nouvelle
3. Une tâche qui a une ressource est la seule à pouvoir la libérer
4. Réalisation d'une boucle avec les demandes et les obtention de ressource.

## Techniques de l'exclusion mutuelle pour la synchronisation

- Verrouillage assisté par le processeur
  - Les tâches qui accèdent à des données partagées sont ordonnancées de telle sorte à ne pas être interrompues
  - ... sauf par des interruptions
  - On peut aussi désactiver les interruptions pour éviter les problèmes d'accès concurrent entre la fonction/*handler* d'interruption et la tâche

- Sous **VxWorks**:

```
FuncA ()
{
  int lock = intLock ();
  .
  .section critique ne pouvant être interrompue
  .
  intUnlock (lock);
}
```

## Autre technique de verrouillage

- **Test and set lock**: appelé aussi variable de condition (matérielle)
  - Le drapeau d'un registre est **testé et modifié** en une seule opération atomique → événement non interruptible.
  - Ce drapeau est testé par n'importe quel tâche voulant accéder à une section critique.

```
int test_and_set(int *lock){
    int temp = *lock;
    *lock = 1; // je modifie la valeur du verrou (ou non)
    return temp; // je retourne l'ancienne valeur
}
```

- **Test and set lock** et variable de verrouillage:
  - Le système ne peut répondre à aucun événement pendant la durée de l'accès à la section critique (monopole du système)

## Les sémaphores

- Introduit par Dijkstra 1965
- Variables entières
  - Sémaphores **binaires**: 0 ou 1
  - Sémaphores **généraux**: 0..nbRessources
- **Verrouillage** de l'accès à une ressource partagée
- **Coordination** de processus avec des événements extérieurs ou entre eux.
- Operations sur les sémaphores: **atomiques**
- Invoqués via des appels système.

## Sémaphores sous VxWorks

1. **Sémaphores binaire**: valeur de 0 ou 1 et seule la ressource concernée peut être verrouillée.
  - Lorsque c'est une **ressource** → sémaphore généralement initialisé à 1
  - Pour de la **synchronisation** (exemple: RDV): initialisé à zéro.

```
#include "vxWorks.h"
#include "semLib.h"
#include "arch/arch/ivarch.h" /* replace arch with architecture type */
SEM_ID syncSem; /* ID of sync semaphore */
init (int someIntNum)
{
    /* connect interrupt service routine */
    intConnect (INUM_TO_IVEC (someIntNum),
               eventInterruptSvcRout, 0);
    /* create semaphore */
    syncSem = semBCreate (SEM_Q_FIFO, SEM_EMPTY);
    /* spawn task used for synchronization. */
    taskSpawn ("sample", 100, 0, 20000, task, 0, 0, 0, 0, 0, 0, 0);
}
task1 (void)
{
    ...
    semTake (syncSem, WAIT_FOREVER); /* wait for event to occur */
    printf ("task 1 got the semaphore\n");
    ... /* process event */
}
eventInterruptSvcRout (void)
{
    ...
    semGive (syncSem); /* let task 1 process event */
}
}
```

## Sémaphores sous VxWorks (2)

2. **Exclusion mutuelle**: sont des sémaphores binaires utilisés seulement à des buts de résolution de problèmes d'exclusion mutuelle relatifs au modèle d'ordonnement de VxWorks:
  - Inversion de priorités
  - Sécurité de suppression (une tâche qui bloque du monde doit déverrouiller l'accès avant d'être supprimée)
  - Accès récursif aux ressources.

```
/* Function A requires access to a resource which it acquires by taking mySem;
 * Function A may also need to call function B, which also requires mySem.*/
#include "vxWorks.h"
#include "semLib.h"
SEM_ID mySem;
/* Create a mutual-exclusion semaphore. */
init ()
{
    mySem = semMCreate (SEM_Q_PRIORITY);
}
funcA ()
{
    semTake (mySem, WAIT_FOREVER);
    printf ("funcA: Got mutual-exclusion semaphore\n");
    ...
    funcB ();
    ...
    semGive (mySem);
    printf ("funcA: Released mutual-exclusion semaphore\n");
}
funcB ()
{
    semTake (mySem, WAIT_FOREVER);
    printf ("funcB: Got mutual-exclusion semaphore\n");
    ...
    semGive (mySem);
    printf ("funcB: Releases mutual-exclusion semaphore\n");
}
}
```

## Sémaphores sous VxWorks (3)

3. Sémaphores de « **comptage** »:
- Gestion de plusieurs copies d'une ressource.
  - Incrémentation et décrémentation compteurs relatifs aux ressources (P et V).
- ```

/* includes */
#include "vxWorks.h"
#include "semLib.h"
SEM_ID mySem;
/* Create a counting semaphore. */
init ()
{
mySem = semCCreate (SEM_Q_FIFO,0);
}
//semTake() & semGive()

```

## Autre type de verrou

- Les **spinlocks**: verrou approprié pour les systèmes multiprocesseurs.
  - Peuvent être utilisés dans tous les contextes (appel système, routine d'interruption, etc.)
  - Attente active
- **Verrou en lecture/écriture**: ce qui est critique, c'est surtout l'écriture concurrente.
  - Plusieurs verrous peuvent être posés en lecture pour une ressource donnée
  - Mais un seul en écriture.
- **Barrière**: synchro d'un ensemble de tâches/processus/threads
  - RDV entre plusieurs threads
  - Barrière initialisée avec le nombre de threads à attendre, à chaque arrivée, le compteur est décrétementé.

## Variable de condition

- Les **variables de conditions** permettent:
  - À une tâche de se mettre en sommeil jusqu'à ce qu'un certain **critère logique défini par une application devienne vrai**.
  - Permet à une tâche de se mettre en sommeil dans une section critique (différent des sémaphores)
- Solution consiste à combiner:
  1. Un sémaphore binaire mutex
  2. Une expression booléenne qui représente la condition logique.
  3. Un signal que les autres tâches utilisent pour réveiller la tâche bloquée pour qu'elle puisse révérifier la variable de condition.
- Détaillé plus tard ...

## Les IPCs... quelques politiques

- Avec **perte de données** ou pas: est ce que tout ce qu'envoie une tâche va être reçu (par la tâche ou le médiateur)?
- **Bloquant** ou pas.
- **Couplage**: un à un, un à plusieurs, plusieurs à un, plusieurs à plusieurs.
- **Nommés/anonymes**: l'émetteur doit-il donner l'identifiant des receveurs?
- **Bufferisé** ou pas: envoi direct ou indirect ?
- Avec ou sans **priorité**



## Les différents IPCs

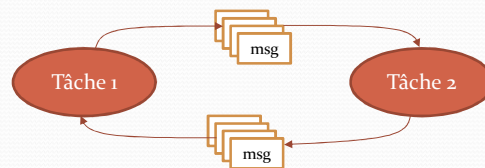
- **Mémoires partagées:**
  - pas de copie « en trop » (pas de médiateur, non bufferisé)
  - Synchronisation doit être explicite
  - Limité par la taille de la RAM
- **FIFO:**
  - Bufferisé/un à un (généralement)
  - Pas de synchronisation à faire.
- **Messages et boîtes aux lettres**
  - Communication de messages de tailles arbitraires
  - En plus des données effectives, un message contient des « méta données » concernant la structure du message.
  - **Messages:** création du message dans l'espace de la tâche, envoi de l'adresse et de l'identifiant du receveur à l'OS. Le receveur demande le message à l'OS.
  - **Boîte aux lettres:** l'émetteur notifie l'OS qu'il a un message, ce dernier copie le message dans la boîte aux lettres de la tâche réceptrice.

## Les différents IPCs (2)

- **Tampon/Buffer circulaire:** même chose que la mémoire partagée mais avec un tableau d'emplacements circulaire:
  - Lecture plus rapide que l'écriture → lecture de la même donnée
  - Écriture plus rapide → écrasement de données non lues
- **Swinging buffer:**
  - Un tampon circulaire avancé: utilisation de plusieurs tableaux circulaires
  - Verrou sans inter blocage possible: lecture et écriture sur des tableaux différents.
- **RPC (Remote Procedure calls):**
  - Invoquer des exécutions de tâches distantes: basé sur l'envoi/réception de messages.
  - plus haut niveau que les autres IPC.

## Communication inter processus via les files de messages

- Messages envoyés via des files de messages.
- L'OS définit:
  - Le **protocole** de communication
  - La méthode **d'authentification**
  - **Tailles** (de la file et du message)
- Les micronoyaux font un usage intensif de ce mécanisme pour la synchronisation.



J.Boukhobza-Systèmes d'exploitation embarqués

19

## Files de messages sous VxWorks

- Description des appels:
  - **msgQCreate ()** alloue et initialise la file de messages
  - **msgQDelete ()** termine et supprime la file de messages
  - **msgQSend ()** envoie un message dans une file de messages
  - **msgQReceive ()** reçoit un message à partir d'une file de messages.

```

/* envoi de msg par la tache 1 et reception +affichage par la tache 2.*/
#include "vxWorks.h"
#include "msgQLib.h"
#define MAX_MSGS 10
#define MAX_MSG_LEN 100
MSG_Q_ID myMsgQId;
task2 (void)
{
    char msgBuf[MAX_MSG_LEN];
    /* reception bloquante sur le msg*/
    if (msgQReceive(myMsgQId, msgBuf, MAX_MSG_LEN,
        WAIT_FOREVER) == ERROR) return (ERROR);
    /* affichage du message*/
    printf ("Message from task 1:\n%s\n", msgBuf);
}

#define MESSAGE "petit coucou de la part de la tache1"
task1 (void)
{
    /* creation de la file de messages*/
    if ((myMsgQId = msgQCreate (MAX_MSGS, MAX_MSG_LEN,
        MSG_Q_PRIORITY)) == NULL) return (ERROR);
    /* envoi de message bloquant (au cas ou la file est remplie)*/
    if (msgQSend (myMsgQId, MESSAGE, sizeof (MESSAGE),
        WAIT_FOREVER, MSG_PRI_NORMAL) == ERROR) return
        (ERROR);
}

```

J.Boukhobza-Systèmes d'exploitation embarqués

20

## Les signaux

- **Asynchrone**: la tâche qui envoie et celle qui reçoit le signal sont dans des états indépendants et ne partagent pas de mémoire → mécanisme de synchronisation
- La tâche envoie un signal et poursuit son exécution, c'est l'OS qui délivre le signal et si personne n'en veut ... tant pis !
- **Généralement**, les signaux:
  - ne sont pas mis en file d'attente
  - ne transportent pas de données
  - temps de livraison est non déterministe (tâche réceptrice non ordonnancée de suite)
  - ordre de livraison non déterministe (plusieurs signaux du même type reçus)
  - sauf les signaux temps réel .... on verra cela plus tard

## Signaux et gestion des interruptions

- A la réception **d'un signal**: suspension de l'exécution actuelle et exécution d'une autre ... fonction (**handler**)
  - Dans le contexte de la tâche actuelle
- Signaux généralement utilisés pour la gestion des interruptions dans un OS (nature asynchrone) ... implémentés de telle sorte qu'ils ne peuvent causer un inter blocage ou un blocage du **handler**.
- **Exceptions**: ... signal synchrone, levé par une tâche pour son propre compte (exécution anormale, ex: division par zéro) → exécution d'un **handler** (ex: message d'erreur + terminaison).

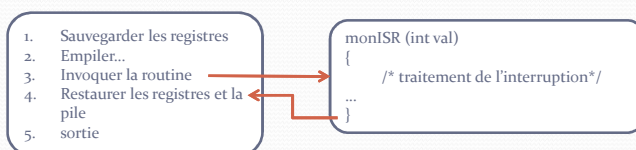
## Signaux sous VxWorks

- Le noyau supporte 2 types **d'interfaces** de signaux:

| BSD 4.3      | POSIX 1003.1                                                         |
|--------------|----------------------------------------------------------------------|
| sigmask()    | sigemptyset(), sigfillset(), sigaddset(), sigdelset(), sigismember() |
| sigblock()   | sigprocmask()                                                        |
| sigsetmask() | sigprocmask()                                                        |
| pause()      | sigsuspend()                                                         |
| sigvec()     | sigaction()                                                          |
| aucun        | sigpending()                                                         |
| signal()     | signaction()                                                         |
| kill()       | kill()                                                               |

## Signaux et interruptions

- Les signaux peuvent être utilisés par l'OS comme une **abstraction** pour les interruptions (matérielles):
  - Les *handlers* deviennent analogues aux ISRs (*Interrupt Service Routines*).
  - L'OS protège l'intégrité de la table d'interruptions ainsi que les ISRs car ce code est exécuté en mode privilégié/superviseur.



## Signaux et interruptions (2)

- C'est l'architecture qui détermine le modèle d'interruption:
  - Le nombre d'interruptions
  - Les types d'interruptions
- Le *pilote d'interruption* initialise et procure l'accès aux interruptions aux couches logicielles supérieures.
- **Latence d'interruption:** temps entre l'initiation d'une interruption et l'exécution effective de l'ISR.
- **Retour d'interruption:** temps pris pour reprendre l'exécution de la tâche.

## Gestion des interruptions sous VxWorks

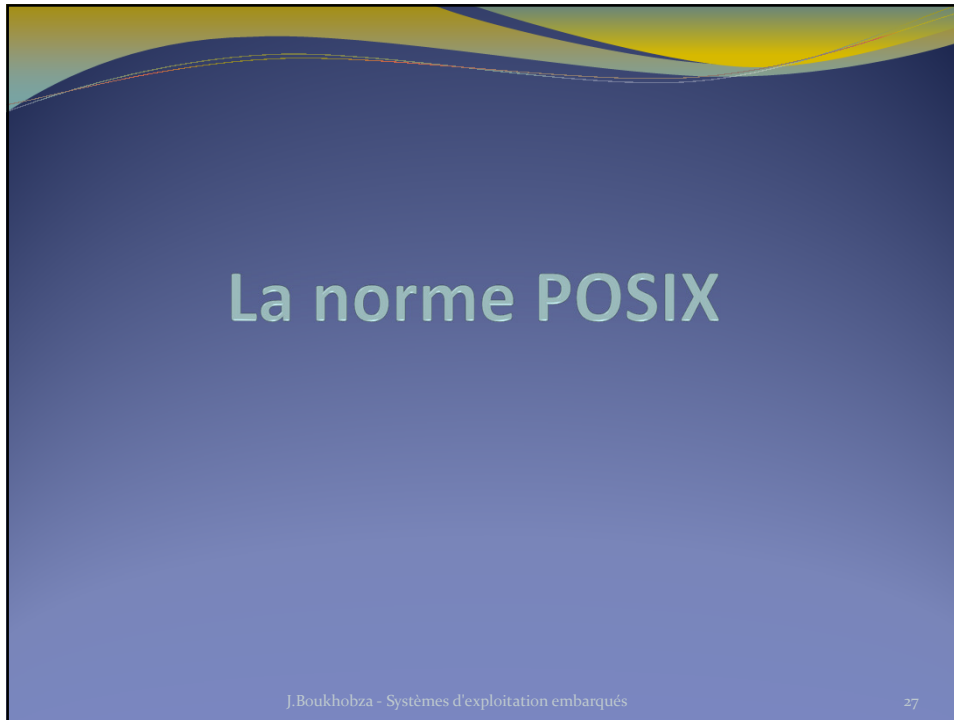
- Les différents ISRs utilisent généralement la **même pile** sauf dans le cas des architectures qui ne fournissent pas de pile pour les interruptions, celles-ci utilisent la pile de la tâche.

```

/* Cette routine initialise le pilote
série, et installe le vecteur
d'interruptions.*/
void InitSerialPort (void)
{
  initSerialPort(); //initialisation
  (void) intConnect (INUM_TO_IVEC
    (INT_NUM_SCC), serialInt, 0);
  ...
}

```

| Appel           | description                                                   |
|-----------------|---------------------------------------------------------------|
| intConnect()    | Connexion d'une routine C à une interruption                  |
| intContext()    | Retourne vrai si appelé à partir d'un niveau d'interruption   |
| intCount()      | Retourne la profondeur du niveau d'imbrication d'interruption |
| intLevelSet()   | Installe le niveau du masque d'interruption du processeur     |
| intLock()       | Désactive les interruptions                                   |
| intUnlock()     | Réactive les interruptions                                    |
| intVecBaseSet() | Installe l'adresse du vecteur de base                         |
| intVecBaseGet() | Demande l'adresse du vecteur de base                          |
| intVecSet()     | Installe un vecteur d'exception                               |
| intVecGet()     | Demande un vecteur d'exception                                |



## La norme POSIX

- 1<sup>ère</sup> 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.

J.Boukhobza - Systèmes d'exploitation embarqués 28

## La norme POSIX (2)

| Norme   | Description                                       |
|---------|---------------------------------------------------|
| 1003.1b | Extensions temps réel basiques (à partir de 1993) |
| 1003.1c | Extensions pour les threads (1995)                |
| 1003.1d | Extensions temps réel additionnelles (1999)       |
| 1003.1j | Extensions temps réel avancées (2000)             |

## La norme POSIX (3)

| Groupe fonctionnel                               | Quelques fonctions principales                                                                                                                              |
|--------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Multi-Threads</b>                             | pthread_create, pthread_exit, pthread_join, pthread_detach, pthread_equal, pthread_self                                                                     |
| <b>Ordonnancement de processus et de Threads</b> | sched_setscheduler, sched_getscheduler, sched_setparam, sched_getparam, pthread_getschedparam, pthread_setschedprio, ...                                    |
| <b>Signaux temps réel</b>                        | sigqueue, pthread_kill, sigaction, sigaltstack, sigemptyset, etc                                                                                            |
| <b>Synchro. et comm. inter processus</b>         | 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. |
| <b>Données spécifiques aux threads</b>           | pthread_key_create, pthread_get_specific, etc.                                                                                                              |
| <b>Gestion Mémoire</b>                           | mlock, mlockall, munlock, munlockall, mprotect                                                                                                              |
| <b>E/S Async</b>                                 | aio_read, aio_write, aio_error, aio_return, aio_fsync, etc.                                                                                                 |
| <b>Horloges et minuteriers (timers)</b>          | clock_gettime, clock_settime, clock_getres, timer_create, timer_gettime, etc.                                                                               |
| <b>Annulation</b>                                | 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 `brctl()`, `getgid()`, `getuid()`, `getpid()`, `getgrnam()`, `getlogin()`,  
11184 `getpdl()`, `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()`, `usalarm()`, 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.

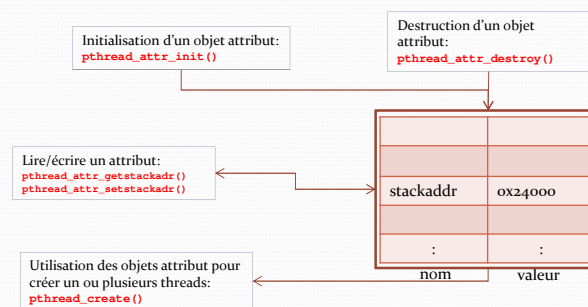
11199 Job control provides a means to group processes and control them as groups, and to control their  
11200 access to the function between the user and the system (the "controlling terminal"). It also  
11201 provides the means to suspend and resume processes.

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

Part D: Portability Considerations — Copyright © 2001-2004, IEEE and The Open Group. All rights reserved. 275

## 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**.





# 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
- Détachement d'un thread: `int pthread_detach(pthread_t th)`; récupération des ressources à la terminaison.
- `pthread_t pthread_self(void)`: retourne l'identifiant d'un thread

# Exemple

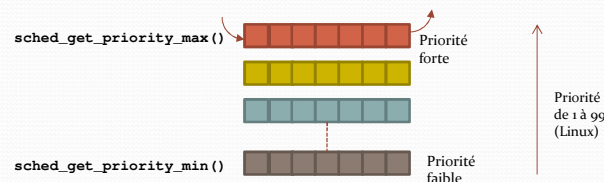
```
int sum; // donnée partagée par les threads

void *runner(void *param); // prototype du thread
main(int argc, char *argv[])
{
    char *val_ret;
    int arg_int=8;
    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, (void *)
    &arg_int);
    // attente que le thread se termine
    pthread_join(tid, (void *)&val_ret);
    printf("%s → sum = %d\n", (char *) &val_ret,
    sum);
}

void *runner(void *param) {
    int upper = *(int *) param;
    int i;
    sum = 0;
    char *c;
    if (upper > 0) {
        for (i = 1; i <= upper; i++)
            sum += i;
    }
    c = (char *) malloc(3*sizeof(char));
    c = strdup("ok");
    // terminaison du thread
    pthread_exit(c);
}
```

## 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).



J.Boukhobza - Systèmes d'exploitation embarqués

35

## Ordonnancement de threads et de processus (2)

- Politiques d'ordonnancement:
  - **SCHED\_FIFO**:
    - Priorités statiques > 0 (privileges 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*.

J.Boukhobza - Systèmes d'exploitation embarqués

36

## Ordonnancement de threads et de processus (3)

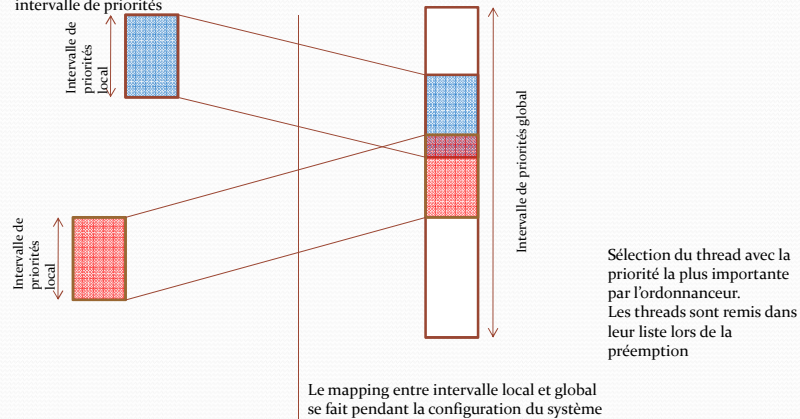
- Fonctions permettant de sélectionner la politique d'ordonnancement, configurer et lire les paramètres de cette dernière.
- Lire / fixer la **politique** d'ordonnancement des processus et ses paramètres:
  - `int sched_setscheduler(pid_t pid, int policy, const struct sched_param *p),`
  - `int sched_getscheduler(pid_t 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

Chaque politique d'ordonnancement contrôle le placement des threads dans son intervalle de priorités



J.Boukhobza - Systèmes d'exploitation embarqués

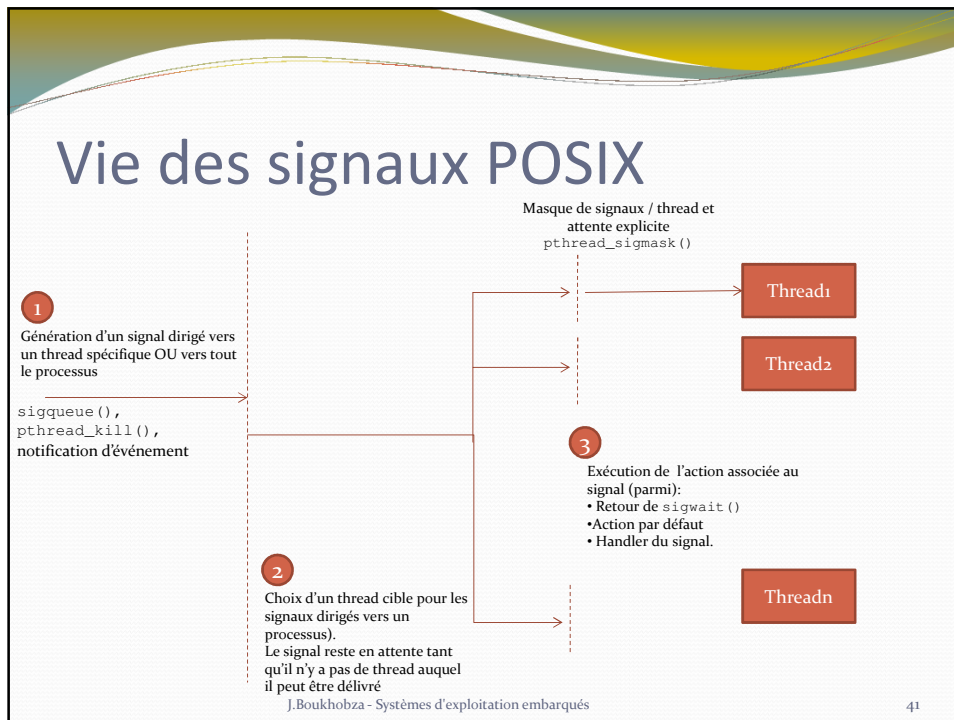
39

## Signaux temps réel et événements asynchrones.

- Utilisés dans les cas suivants:
  - Erreur pendant l'exécution d'un programme (référence mémoire erronée)
  - Panne système ou matériel (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.

J.Boukhobza - Systèmes d'exploitation embarqués

40



## 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)

42

## 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 (`sa_flags`) 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)` → retourne avec les infos 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)

## 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
  - ....

## 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.

J.Boukhobza - Systèmes d'exploitation embarqués

47

## Files de messages (3)

```
#include <mqueue.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>

#define MSG_SIZE 4096

// Cette fonction 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);

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

    // une file temporaire, dès qu'elle sera fermée, elle sera
    // supprimée
    mq_unlink ("autrefile");

    // 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);
    }

    // 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);

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

    // l'appel a échoué, s'assurer que errno est EAGAIN
    if (errno != EAGAIN) {
        perror ("mq_receive()");
        _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, "18-", 4, prio) == -1)
            perror ("mq_send()");
    }
}
```

48



## 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 fautive, le thread bloque sur la variable de condition et relâche atomiquement la mutex (implicite) 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.

```

Lock (mutex)
    wait (event)
    work
    signal (event)
Unlock (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
  - 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 débloquent (possibilité d'utiliser `int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime)`).
  - La fonction `int pthread_cond_signal(pthread_cond_t *cond)` peut être appelée pour débloquent au moins l'un des threads bloqués sur la variable de condition. `int pthread_cond_broadcast(pthread_cond_t *cond)` peut être utilisée pour débloquent 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;
}

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()%10000;
        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() /* MAIN */
{
    int i, num;
    pthread_mutex_init(&mutex, 0);
    pthread_cond_init(&vide, 0);
    pthread_cond_init(&plein, 0);
    /* creation des threads */
    pthread_create(&tid[0], 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);
}

```

Section critique

Section critique

Déclaration des variables de condition

Initialisation

J.Boukhobza - Systèmes d'exploitation embarqués 55

## 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 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 *addr, size_t length, int prot, int flags, int fd, off_t offset)` (l'inverse est effectué avec `int munmap(void *addr, 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 évite 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)`.
- **Protéger** des emplacements mémoire → `int mprotect(const void *addr, size_t len, int prot)`

## 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 aioctx`) 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**.

## E/S asynchrones (2)

- `int aio_read(struct aiocb *aiocbp)` et `int aio_write(struct aiocb *aiocbp)` : requête de lec/scr 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.

## 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 */
struct sigevent aio_sigevent; /* signal number and offset */
```