Cours 5: programmation parallèle
Communication inter-processus

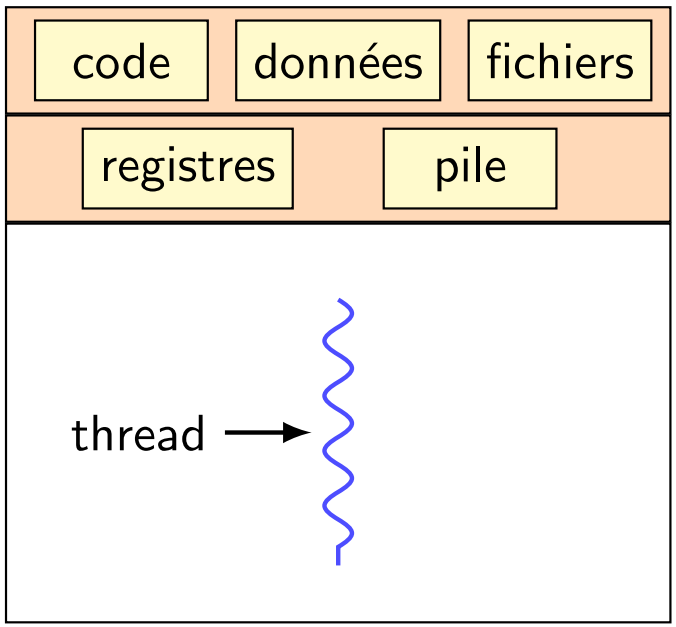
Christophe Morvan

Université de Marne-la-Vallée

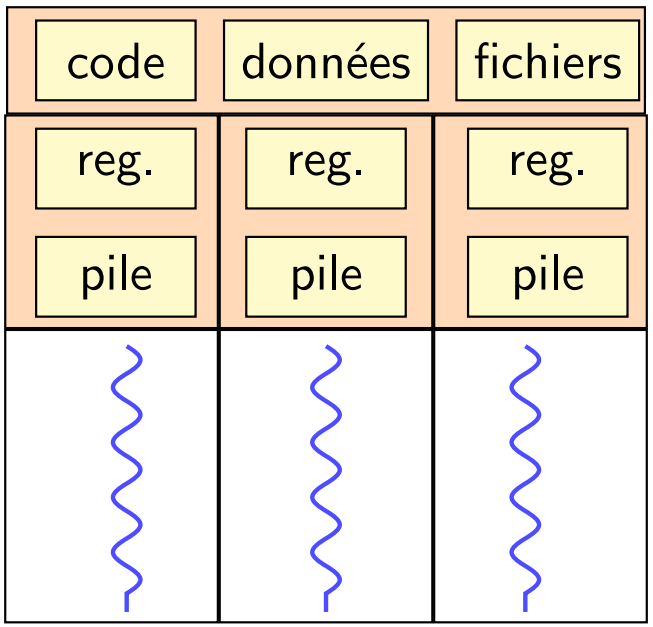
3 novembre 2015

Les threads

Application thread unique



Application threads multiples



Les threads / processus

Premier temps : problèmes du point de vue des threads
Second temps : on verra comment deux processus peuvent communiquer (et donc faire face à des problèmes similaires)

Plan

Première partie

Seconde partie

Progression

Première partie

Rappel : threads POSIX

```
#include <pthread.h>
```

Quelques appels

Création

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

Destruction

```
void pthread_exit(void *retval);
```

Attente d'un autre thread

```
int pthread_join(pthread_t thread, void **retval);
```

`pthread_attr_init` Permet de créer et initialiser la structure de type `pthread_attr_t`

L'appel `pthread_yield` permet à un thread de *rendre la main*

Que se passe-t-il ?

Programme 1

```
int flag1 = 0, flag2 = 0;
void * p1 (void *param) {
  flag1 = 1;
  if (!flag2) { section_critique_1 (); }
}
void * p2 (void *param) {
  flag2 = 1;
  if (!flag1) { section_critique_2 (); }
}
int main () {
  pthread_t * t2; void ** inutile=NULL;
  pthread_create (t2, NULL,&p1, NULL );
  p2 (NULL); pthread_join (*t2, inutile);
}
```

Les deux sections critiques peuvent-elles être exécutés simultanément ?

Que se passe-t-il ?

Programme 2

```
int data = 0, pret = 0;

void * p1 (void *param) {
  data = 2000;
  pret = 1;
}

void * p2 (void *param) {
  while (!pret)
    ;
  tester (data);
}

int main () { ... }
```

`tester` peut-elle être appelée avec 0 ?

Que se passe-t-il ?

Programme 3

```
int a = 0, b = 0;

void * p1 (void *param) { a = 1; }

void * p2 (void *param) {
  if (a == 1)
    b = 1;
}

void * p3 (void *param) {
  if (b == 1)
    tester (a);
}

int main () { ... }
```

`tester` peut-elle être appelée avec 0 ?

Réponses

Programme 1

Ça dépend !

Programme 2

Ça dépend !

Programme 3

Ça dépend !

Pourquoi ?

Dépend du processeur !

En cas de cohérence séquentielle toutes les réponses sont négatives

Ce n'est pas le cas de tous les processeurs

Cohérence séquentielle

Définition (Lamport 79)

On dit que l'exécution parallèle de deux programmes est séquentiellement cohérente s'il est possible de construire un entrelacement des deux programmes produisant le même résultat

En pratique

- Garantir l'ordre des programmes sur chaque processeur
- Assurer l'atomicité des écriture

Cohérence séquentielle

Définition (Lamport 79)

On dit que l'exécution parallèle de deux programmes est séquentiellement cohérente s'il est possible de construire un entrelacement des deux programmes produisant le même résultat

En pratique

- Garantir l'ordre des programmes sur chaque processeur
- Assurer l'atomicité des écriture

En l'absence de cohérence séquentielle (CS), les exécution de programmes sur plusieurs processeurs peuvent avoir des comportement **très** erratiques
Garantir la cohérence séquentielle est difficile

Difficultés matérielles et logicielles

La CS perturbe les optimisations processeur

- Réordonner les écritures (prog 1 et 2)
- Déterminer à l'avance la valeur d'une variable (prog 2)
- Utilisation de cache (prog 3)

Difficultés matérielles et logicielles

La CS perturbe les optimisations processeur

- Réordonner les écritures (prog 1 et 2)
- Déterminer à l'avance la valeur d'une variable (prog 2)
- Utilisation de cache (prog 3)

La CS perturbe les optimisations du compilateur

- Déplacement de code
- Élimination de sous-expressions communes ($\rightarrow$ lit moins souvent la mémoire)
- Réorganisation des blocs (découpage des boucles pour de meilleures performance de cache)

Difficultés matérielles et logicielles

La CS perturbe les optimisations processeur

- Réordonner les écritures (prog 1 et 2)
- Déterminer à l'avance la valeur d'une variable (prog 2)
- Utilisation de cache (prog 3)

La CS perturbe les optimisations du compilateur

- Déplacement de code
- Élimination de sous-expressions communes ($\rightarrow$ lit moins souvent la mémoire)
- Réorganisation des blocs (découpage des boucles pour de meilleures performance de cache)

Les processeurs de la famille x86 assure la CS (dans l'ensemble)

Il y a d'autres problèmes

Fondamental

En pratique on considère que la cohérence séquentielle est assurée par le matériel

Lorsque ce n'est pas toujours le cas, le système et le compilateur doivent faire en sorte qu'elle soit tout de même assurée

Il y a d'autres problèmes

Fondamental

En pratique on considère que la cohérence séquentielle est assurée par le matériel
Lorsque ce n'est pas toujours le cas, le système et le compilateur doivent faire en sorte qu'elle soit tout de même assurée

Autres problèmes

Une fois acquise la cohérence séquentielle, le programmeur est confronté à d'autres problèmes fondamentaux de la programmation parallèle (concurrente)

Progression

Première partie

Fonctions réentrantes – 1

Définition

Une fonction est réentrante, lorsqu'elle peut être interrompue, une autre invocation est faite dans l'intervalle, puis reprise, et les deux invocations fournissent un résultat correct.

Fonctions réentrantes – 1

Définition

Une fonction est réentrante, lorsqu'elle peut être interrompue, une autre invocation est faite dans l'intervalle, puis reprise, et les deux invocations fournissent un résultat correct.

Fonction 1

```
int temp;

void swap(int *a, int *b)
{
 temp = *a;
 *a = *b;
 // Changement de thread ???
 *b = temp;
}
```

Fonctions réentrantes –2

Fonction 2

```
int test (int val){
  static int mul = 1;
  mul++;
  return val * mul;
}
```

Fonctions réentrantes –2

Fonction 2

```
int test (int val){
  static int mul = 1;
  mul++;
  return val * mul;
}
```

Principes (pour être réentrante)

- Pas d'utilisation de variable globale ou statique
- Pas de retour de pointeur statique
- Pas d'appel de fonction non-réentrantes
  Ex. : `malloc`, `free` ou les fonctions d'entrée/sortie de la libc ne le sont pas

Progression

Première partie

Exclusion mutuelle

Définition

L'exclusion mutuelle est une méthode qui permet d'assurer que deux processus (threads) distincts n'utilisent pas simultanément une ressource partagée

Exclusion mutuelle

Définition

L'exclusion mutuelle est une méthode qui permet d'assurer que deux processus (threads) distincts n'utilisent pas simultanément une ressource partagée

Exemple

Deux processus souhaitent placer des fichiers dans une file d'impression

Il est souhaitable que les deux n'agissent pas simultanément sur les éléments de cette file

Section critique

Définition

On appelle section critique la portion de code où un programme souhaite un accès exclusif à une ressource donnée

Section critique

Définition

On appelle section critique la portion de code où un programme souhaite un accès exclusif à une ressource donnée

Principes

On veut réunir les 4 conditions suivantes :

1) 2 processus ne doivent pas se retrouver simultanément en section critique
2) Pas de supposition sur le nombre ou la vitesse des processus
3) Aucun processus hors de sa section critique ne doit bloquer d'autre processus
4) Aucun processus ne doit attendre indéfiniment pour pouvoir entrer en section critique

Cas simple

Programme exemple

```
int partage=0; /* variable partagée */

void * critiqueProt(){
  /* Teste avant passage critique */
  while (partage !=0) ;
  partage = 1; /* Positionne un verrou */
  section_critique();
  partage = 0; /* Libère le verrou */
}
```

Cas simple

Programme exemple

```
int partage=0; /* variable partagée */

void * critiqueProt(){
  /* Teste avant passage critique */
  while (partage !=0) ;
  partage = 1; /* Positionne un verrou */
  section_critique();
  partage = 0; /* Libère le verrou */
}
```

Trop simple

Deux appels concurrents à cette fonction peuvent constater que la variable vaut zéro
Positionner tous deux la variable à 1, puis exécuter leur section critique

Alternance stricte

Programme exemple

```
int tour=1; /* variable partagée */
void * thread1(void * param){
  while (1){
    while (tour !=1) ; /* Attention au ";" */
    section_critique();
    tour = 2; /* Libère le verrou */
    /* code non critique */ }}
void * thread2(void * param){
  while (1){
    while (tour !=2) ;
    section_critique();
    tour = 1; /* Libère le verrou */
    /* code non critique */ }}
```

Alternance stricte

Programme exemple

```
int tour=1; /* variable partagée */
void * thread1(void * param){
  while (1){
    while (tour !=1) ; /* Attention au ";" */
    section_critique();
    tour = 2; /* Libère le verrou */
    /* code non critique */ }}
void * thread2(void * param){
  while (1){
    while (tour !=2) ;
    section_critique();
    tour = 1; /* Libère le verrou */
    /* code non critique */ }}
```

Problèmes

Attente active (`while`) Seulement 2 processus

Solution de Peterson (1981) – 2 processus

Programme exemple

```
int tour, interet[2] ; /* Généralisable pour >2 threads */

void debut_section(int proc){
  int autre;
  autre = 1- proc; /* 0 -> 1 et 1 -> 0*/
  interet[proc]=1;
  tour = proc;
  while(tour == proc && interet[autre]==1) ;
}
void fin_section(int proc){
  interet[proc]=0;
}
```

Producteur/consomateur – 1

Principe

Il s'agit de coordonner deux processus/threads qui échangent des objets au travers d'un tampon de taille fixée

Producteur/consomateur – 1

Principe

Il s'agit de coordonner deux processus/threads qui échangent des objets au travers d'un tampon de taille fixée

Producteur/consommateur

```
void producteur (void *param) {
  /* Tableau *prods*  et int *count* partagés */
  while (1) {
    /* Production d'un item -> suivant */
    if (count == TAILLE_FILE)
      sleep() ; // attend d'être réveillé
    prods [nbProd] = suivant;
    nbProd = (nbProd + 1) % TAILLE_FILE;
    count++;
    if (count == 1) wakeup(consomateur);
  }
}
```

Producteur/consomateur – 2

Principe

Il s'agit de coordonner deux processus/threads qui échange des objets au travers d'un tampon de taille fixée

Producteur/consommateur

```
/* Tableau *prods*  et int *count* partagés */
void consomateur (void *param) {
  while (1) {
    if (count == 0)
      sleep() ; // attend d'être réveillé
    suivant = prods [nbCons];
    nbCons = (nbCons + 1) % TAILLE_FILE;
    count--;
    if (count == TAILLE_FILE -1) wakeup(producteur);
    /* Consommation de l'item suivant */
  }
}
```

Problème ?

Problème

Interruption du processus consommateur alors qu'il vient de tester le vide de la file mais n'est pas encore passé en `sleep`

Problème ?

Problème

Interruption du processus consommateur alors qu'il vient de tester le vide de la file mais n'est pas encore passé en `sleep`
Le producteur lance un `wakeup` qui n'est pas reçu
Le Consommateur s'endort ensuite, n'est plus jamais réveillé
Ne réveille jamais le producteur lorsque celui-ci s'endort à son tour

Problème ?

Problème

Interruption du processus consommateur alors qu'il vient de tester le vide de la file mais n'est pas encore passé en `sleep`
Le producteur lance un `wakeup` qui n'est pas reçu
Le Consommateur s'endort ensuite, n'est plus jamais réveillé
Ne réveille jamais le producteur lorsque celui-ci s'endort à son tour

Solution ?

Le problème est que les messages envoyés en avance ne sont jamais reçu
Il faut imaginer des verrous plus subtils

Mutex

Définition

Les mutex sont des verrous utilisateurs
Ils sont définis a l'aide de primitives materiels qui réalisent des opérations atomiques.
Ils reposent sur l'attente active
Ils perment de résoudre producteur/consommateur (à suivre)

Opérations

- Initialisation : création de la mutex
- Verrouillage : Demande du verrou, attente (active) jusqu'à obtention
- Déverrouillage : Libération du verrou

Producteur avec Mutex

Programme exemple

```
/* Tableau *prods*  et int *count* partagés */
mutex_t mutex = MUTEX_INITIALIZER;
void producteur (void *param) {
  int nbProd=0;
  while (1) {
    /* Production d'un item -> suivant */
    mutex_lock (&mutex);
    while (count == MAX_FILE) {
      mutex_unlock (&mutex);  // <--- Pourquoi ?
      thread_yield ();
      mutex_lock (&mutex);
    }
    prods [nbProd] = suivant;
    nbProd = (nbProd + 1) % TAILLE_FILE;
    count++;
    mutex_unlock (&mutex);
  }}
```

Consommateur avec Mutex

Programme exemple

```
/* Tableau *prods*  et int *count* partagés */
void consomateur (void *param) {
  while (1) {
    mutex_lock (&mutex);
    while (count == 0) {
      mutex_unlock (&mutex);
      thread_yield ();
      mutex_lock (&mutex);
    }
    suivant = prods [nbCons];
    nbCons = (nbCons + 1) % TAILLE_FILE;
    count--;
    mutex_unlock (&mutex);
    /* Consommation de l'item suivant */
  }}
```

Implémentation des Mutex

pthread implémente les Mutex

En général les bibliothèques de threads implémente des mutex.
Quelques méthodes :

```
void mutex_init (mutex_t *m, ...);
void mutex_lock (mutex_t *m);
int mutex_trylock (mutex_t *m);
void mutex_unlock (mutex_t *m);
```

Dans phtread on ajoute le préfixe pthread_ (aux méthodes *et* aux types)

Contrats des bibliothèques de threads

Assurer la cohérence séquentielle

La bibliothèque et le compilateur doivent assurer la cohérence séquentielle en cas de bon usage des threads

Contrats des bibliothèques de threads

Assurer la cohérence séquentielle

La bibliothèque et le compilateur doivent assurer la cohérence séquentielle en cas de bon usage des threads

Bon usage des threads

Les mutex doivent protéger les accès aux variables globales

Cela relève du programmeur de l'application

Contrats des bibliothèques de threads

Assurer la cohérence séquentielle

La bibliothèque et le compilateur doivent assurer la cohérence séquentielle en cas de bon usage des threads

Bon usage des threads

Les mutex doivent protéger les accès aux variables globales
Cela relève du programmeur de l'application

Observation

Le noyau d'un OS doit faire appel à des verrous (mutex)

Sémaphores

1965 : E.W. Dijkstra

Les sémaphore offrent une généralisation des mutex
Ceci autorise *plusieurs libération* de la ressource

Opérations historiques

- P : verrouille (`sleep, wait`) *proberen* (tester)
- V : déverrouille (`wakeup`) *verhogen* (incrémenter)

Sémaphores

1965 : E.W. Dijkstra

Les sémaphore offrent une généralisation des mutex
Ceci autorise *plusieurs libération* de la ressource

Opérations historiques

- *P* : verrouille (`sleep, wait`) *proberen* (tester)
- *V* : déverrouille (`wakeup`) *verhogen* (incrémenter)

En-tête : `semaphore.h`

Implémenté par plusieurs bibliothèques

- `int sem_wait(sem_t * sem);`
- `int sem_post(sem_t * sem);`

Voir : `man sem_overview`

Sémaphores : prod/cons – initialisation

Initialisation – reset

```
#include <semaphore.h>
#define MAX_FILE 7
void init_sem(sem_t ** nonVide, sem_t **nonPlein)
{
  *nonVide = sem_open("/semNonVide",O_CREAT,0666,0);
  if(*nonVide == SEM_FAILED) perror("NonVide");

  *nonPlein = sem_open("/semNonPlein",O_CREAT,0666,MAX_FILE);
  if(*nonPlein == SEM_FAILED) perror("NonPlein");
}
void reset(int * prods){
  /* Réciproque de la création != close */
  sem_unlink("/semNonVide");
  sem_unlink("/semNonPlein");
}
```

Sémaphores : prod/cons – producteur

Producteur

```
void producteur(int* prods){
  /* Tableau *prods* partagé */
  int suivant, nbProd=0;
  sem_t *nonVide, *nonPlein;
  init_sem(&nonVide,&nonPlein);
  while(1){
    /* Production de suivant */
    sem_wait(nonPlein);
    prods[nbProd]=suivant;
    nbProd=(nbProd+1)%MAX_FILE;
    sem_post(nonVide);
  }
}
```

Sémaphores : prod/cons – consommateur

Consommateur

```
void consommateur(int* prods){
  /* Tableau *prods* partagé */
  int nbCons=0;
  sem_t *nonVide, *nonPlein;
  init_sem(&nonVide,&nonPlein);
  while(1){
    sem_wait(nonVide);
    suivant=prods[nbCons];
    nbCons=(nbCons+1)%MAX_FILE;
    sem_post(nonPlein);
    /* Consommation effective de suivant */
  }
}
```

Étapes à venir

Système

Vendredi 9 janvier, 10h30-12h30 : Examen écrit
Mercredi 14 janvier : Début du projet

Algorithmique avancée

Jeudi 15 janvier, 8h15-10h15 / 10h30-12h30 : Examen écrit
Vendredi 9 janvier : Début du projet

Première partie : Synthèse

Cohérence séquentielle

La cohérence séquentielle est un aspect fondamental de la programmation distribuée

Première partie : Synthèse

Cohérence séquentielle

La cohérence séquentielle est un aspect fondamental de la programmation distribuée

Nombreux mécanisme pour assurer l'exclusion mutuelle

- Alternance stricte
- Peterson
- Mutex
- Sémaphores

Il y a d'autres mecanismes

Seconde partie : aperçu

Communication interprocessus

Plusieurs approches :

- Messages
- Mémoire partagée
- Tubes

Seconde partie : aperçu

Communication interprocessus

Plusieurs approches :

- Messages
- Mémoire partagée
- Tubes

Problèmes classiques

- Dîner des philosophes
- Lecteurs/rédacteurs

Progression

Seconde partie

Conditions de concurrence

Définition

On dit qu'il y a condition de concurrence (*data race*) lorsque deux processus s'exécutant en parallèle peuvent produire un résultat différent selon leur entrelassement

Conditions de concurrence

Définition

On dit qu'il y a condition de concurrence (*data race*) lorsque deux processus s'exécutant en parallèle peuvent produire un résultat différent selon leur entrelassement

Note

Condition de concurrence $\neq$ non (cohérence séquentielle)

Conditions de concurrence

Définition

On dit qu'il y a condition de concurrence (*data race*) lorsque deux processus s'exécutant en parallèle peuvent produire un résultat différent selon leur entrelassement

Note

Condition de concurrence $\neq$ non (cohérence séquentielle)

Exemple

```
                    valeur = 1;
```

`int val = getValeur();`	`int val = getValeur();`
`setValeur(++val);`	`setValeur(++val);`

Conditions de concurrence

Définition

On dit qu'il y a condition de concurrence (*data race*) lorsque deux processus s'exécutant en parallèle peuvent produire un résultat différent selon leur entrelassement

Note

Condition de concurrence $\neq$ non (cohérence séquentielle)

Exemple

```
                    valeur = 1;
```

```
int val = getValeur();      int val = getValeur();
setValeur(++val);           setValeur(++val);
```

Deux possibilités : `valeur=2` ou `valeur=3`

Transmission de messages

Principe général

Deux primitives :

`send(dest, &mess);`

Expédie le message `mess` à la destination `dest`

Cet appel peut être soit bloquant, soit renvoyer une erreur lorsque la file de messages est pleine

`receive(source, &mess);`

Reçoit le premier message de `source`

Cette appel peut être soit bloquant, soit renvoyer une erreur lorsque la file de messages est vide

Implémentation des messages

Difficulté

- Réseau ?
- Aquitement ?
  Si il y a un risque de perte de message, signaler la réception n'empêche pas la réémission du message original
- Authentification ?
- Stockage des messages ?

Implémentation des messages

Difficulté

- Réseau ?
- Aquitement ?
  Si il y a un risque de perte de message, signaler la réception n'empêche pas la réémission du message original
- Authentification ?
- Stockage des messages ?

Producteur/consommateur

A suivre un exemple d'utilisation des messages pour résoudre le problème du producteur et du consommateur
L'objet produit est passé dans les messages

Messages : prod/cons – consommateur

Consommateur

```
void consommateur(){
  int suivant, i;
  message m=NULL,;
  /* Initialisation de la "file de messages" */
  for(i=0; i< MAX_FILE ; i++) send(prod, &m);

  while(1){
    receive (prod, &m);
    suivant=extract_item(&m);
    send (prod, &m);
    /* Consommation effective de suivant */
  }
}
```

Messages : prod/cons – producteur

Producteur

```
void producteur(){
  int suivant, i;
  message m=NULL,;
  while(1){
    /* Production de suivant */
    receive (cons, &m);
    m=creer_message(&m, suivant);
    send (cons, &m);
  }
}
```

Messages dans Linux

Deux dispositifs (conformes POSIX)

- `mq` (depuis 2.6.6)
- `msg` (antérieure)

Messages dans Linux

Deux dispositifs (conformes POSIX)

- `mq` (depuis 2.6.6)
- `msg` (antérieure)

co-existent aujourd'hui

Messages dans Linux

Deux dispositifs (conformes POSIX)

- `mq` (depuis 2.6.6)
- `msg` (antérieure)

co-existent aujourd'hui

L'interface `mq`

Elle est construite en suivant le modèle classique des descripteurs de fichiers (appels `open` et `close`)
Il est nécessaire d'utiliser `-lrt` pour exploiter la plupart des fonctions de cette interface

Interface mq – 1

Information globale `man mq_overview`
En-tête `#include <mqueue.h>`

Création/destruction

- `mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);`
  Création/ouverture
- `mqd_t mq_close(mqd_t mqdes);`
  Fermeture
- `int mq_unlink(const char *name);`
  Destruction

Par défaut les files sont crées bloquante : en l'absence de message la réception bloque
Lorsque la file est pleine, l'émission est bloquée

Interface mq – 2

Messages

- `int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned msg_prio);`
- `ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned *msg_prio);`

Priorités : 0 (faible) $\rightarrow$ `sysconf(_SC_MQ_PRIO_MAX) - 1` (haute)

Envois/receptions temporisés (utilise `time.h`)

- `int mq_timedsend(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned msg_prio, const struct timespec *abs_timeout);`
- `ssize_t mq_timedreceive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned *msg_prio, const struct timespec *abs_timeout);`

Mémoire partagée

Note

Sous Linux, les sémaphores sont des *cas particuliers* de mémoire partagée

Fichiers d'en-tête

```
#include <sys/mman.h>
```

Commun avec les fichiers/sémaphores/messages

```
#include <sys/stat.h> /* Constantes de mode */
#include <fcntl.h> /* Constantes en O_* */
```

Nécessite également l'option `-lrt` de `gcc`

Création et destruction

Information globale `man shm_overview`

Création/destruction

- `int shm_open(const char *name, int oflag, mode_t mode);` Création
- `int shm_unlink(const char *name);` Destruction
- `int ftruncate(int fd, off_t length);` Taille

Utilisation

- `void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);`
  Obtention d'un pointeur
- `int munmap(void *addr, size_t length);`
  Libération

Mémoire partagée : Programme principal

main

```
int * init_mem();
void reset(int * prods);
void producteur(int* prods);
void consommateur(int* prods);

int main(int argc , char * argv[]){
  int * prods = init_mem();
  if (argc>1){
    switch (argv[1][0]){
    case '1' : producteur(prods);
      break;
    case '2': consommateur(prods);
      break;
    default: reset(prods);
    }}
  else printf("%s",messageUsage);
}
```

Mémoire partagée : producteur et consommateur

Ils sont identique à la version donnée en partie 1, pour les sémaphores

Producteur

```
void producteur(int* prods){
  /* Tableau *prods* en mémoire partagée */
  int suivant, nbProd=0;
  sem_t *nonVide, *nonPlein;
  init_sem(&nonVide,&nonPlein);
  while(1){
    /* Production de suivant */
    sem_wait(nonPlein);
    prods[nbProd]=suivant;
    nbProd=(nbProd+1)%MAX_FILE;
    sem_post(nonVide);
  }
}
```

Mémoire partagée : prod/cons – initialisation

Initialisation – reset

```
int * init_mem()
{
  int *prods, memDesc =
    shm_open("/memPart", O_CREAT|O_RDWR, S_IRUSR | S_IWUSR);
  if (memDesc == -1) perror("Open memDesc");
  if (ftruncate(memDesc, MAX_FILE*sizeof(int)) == -1)
    perror("Truncate memDesc");
  prods =(int*) mmap(NULL, MAX_FILE*sizeof(int), \
   PROT_READ|PROT_WRITE,MAP_SHARED,memDesc,0);
  if (prods == MAP_FAILED){
    perror("mmap prods");
    exit(0); }
  return prods;
}
void reset(int * prods){
  munmap(prods, MAX_FILE*sizeof(int));
  shm_unlink("/memPart");
  /* sem_unlink */
}
```

Tubes et tubes nommés

Principe

Il s'agit de files `FIFO` créées en mémoire
Elles sont utilisées comme des fichiers physiques
La lecture y est destructive

Les tubes anonymes

Ils ne peuvent être utilisé que par un processus (lui-même) ou un de ses fils (créé par `fork`)
Création :

```
int pipe(int pipefd[2]);
```

0 $\rightarrow$ lecture 1 $\rightarrow$ ecriture

Tubes nommés

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

Tubes : lecture

Les écritures et les lectures dans les tubes sont atomiques
La macro `PIPE_BUF` (définie dans `limits.h`) fixe la taille maximale (4096 octets dans linux) lue ou écrite

Lecture

```
int read (int fd, void *buf, int nbytes);
```

- S'il y a `nbytes` : lecture
- S'il est vide sans rédacteur : le résultat est nul
- S'il est vide avec rédacteur :
  - Par défaut lecture bloquante : attente (sommeil)
  - Si le tube a été ouvert avec `O_NBLOCK` : résultat : -1 et erreur (`errno == EAGAIN`)
  - Si le tube a été ouvert avec `O_NDELAY` : résultat : 0

Tubes : écriture

Écriture

```
int write (int fd, void *buf, int nbytes);
```

- S'il n'y a plus de lecteur signal `SIGPIPE`
- Si l'écriture est bloquante la fonction termine une fois les `nbytes` écrits
- Sinon le résultat est le nombre effectivement écrit (comparé à l'espace libre dans le tube)

Modifier les attributs

```
int fcntl(int fd, int cmd, ... /* arg */ );
```

La commande `F_SETFL` permet de positionner `O_NONBLOCK`

`lseek`

Cette fonction n'existe pas pour les tubes !

Comparaison – autres solution

Messages vs mémoire partagée

Messages	Mémoire partagée
• Simple	• Précis (types)
• Léger	• Transparent
• Multiples participants	• Polyvalent

Fichiers

Il est également possible pour des processus/threads de communiquer en utilisant des fichiers
C'est similaire à ce qui peut se faire par les tubes

Interblocage

Principe

Dès lors que des verrous sont définis, il est possible que deux processus se bloquent mutuellement
Ce phénomène est appelé interblocage (*deadlock*)

Exemple

```
mutex_lock(&m1);
mutex_lock(&m2);
section_critique1();
mutex_unlock(&m2);
mutex_unlock(&m1);
```

```
mutex_lock(&m2);
mutex_lock(&m1);
section_critique2();
mutex_unlock(&m1);
mutex_unlock(&m2);
```

Interblocage

Principe

Dès lors que des verrous sont définis, il est possible que deux processus se bloquent mutuellement
Ce phénomène est appelé interblocage (*deadlock*)

Exemple

```
mutex_lock(&m1);
mutex_lock(&m2);
section_critique1();
mutex_unlock(&m2);
mutex_unlock(&m1);
```

```
mutex_lock(&m2);
mutex_lock(&m1);
section_critique2();
mutex_unlock(&m1);
mutex_unlock(&m2);
```

Il est possible d'avoir des cycle de grande longueur

Interblocage – Résolution

Détection

Un interblocage est détecté par la présence d'un circuit dans le graphe de dépendance de ressources

Prévention

Le système peut refuser d'attribuer une ressource lorsqu'il détecte un interblocage en cas d'allocation

Libération forcée

Le système peut imposer la libération temporaire d'une ressource lorsque ça permet de résoudre l'interblocage
Risque ?

Interblocage – Résolution

Détection

Un interblocage est détecté par la présence d'un circuit dans le graphe de dépendance de ressources

Prévention

Le système peut refuser d'attribuer une ressource lorsqu'il détecte un interblocage en cas d'allocation

Libération forcée

Le système peut imposer la libération temporaire d'une ressource lorsque ça permet de résoudre l'interblocage
Risque ?
Appliqué à un sémaphore, on perd le bénéfice de ce sémaphore

Progression

Seconde partie

Problème des philosophes

Principe

Les philosophes ont 2 activités :

- Penser
- Manger

(Image B. D. Esham)

Problème des philosophes

Principe

Les philosophes ont 2 activités :

- Penser
- Manger

(Image B. D. Esham)

Problème

Ils ont besoin de 2 fourchettes pour manger
Objectif : définir un mode de fonctionnement pour les philosophes pour assurer qu'ils puissent tous manger régulièrement

Non-solution

Programme exemple

```
#define N 5
void philosophe (int moi){
  while(1){
    penser();
    prendre_fourchette(moi);
    prendre_fourchette((moi+1)%N);
    manger();
    poser_fourchette(moi);
    poser_fourchette((moi+1)%N);
  }
```

Non-solution

Programme exemple

```
#define N 5
void philosophe (int moi){
  while(1){
    penser();
    prendre_fourchette(moi);
    prendre_fourchette((moi+1)%N);
    manger();
    poser_fourchette(moi);
    poser_fourchette((moi+1)%N);
  }
```

Problème

Chaque philosophe s'empare de la fourchette à sa gauche

Ils sont tous bloqués

Non-solution

Programme exemple

```
#define N 5
void philosophe (int moi){
  while(1){
    penser();
    prendre_fourchette(moi);
    prendre_fourchette((moi+1)%N);
    manger();
    poser_fourchette(moi);
    poser_fourchette((moi+1)%N);
  }
```

Problème

Chaque philosophe s'empare de la fourchette à sa gauche
Ils sont tous bloqués
En utilisant une mutex on assure l'exlusivité du repas
Mais : un seul philosophe à la fois et éventuelle alternance entre un sous-groupe des 5

Une solution (Dijkstra 1965) – 1

Programme exemple

```
#define GAUCHE (moi+N-1)%N
#define DROITE (moi+1)%N
#define PENSE 0
#define FAIM 1
#define MANGE 2
sem_t * mutex = sem_open("/semMutex",O_CREAT,0666,1);
sem_t semTab[N]; /* Initialisé quelque part initialement 0*/
int etat [N];

void philosophe(int moi){
  while(1){
    penser(); /* positionne etat[moi] -> PENSER */
    prendre_fourchette(moi);
    manger();
    poser_fourchette(moi);
  }
```

Une solution (Dijkstra 1965) – 2

Programme exemple

```
void prendre_fourchette(int moi){
  sem_wait(mutex);
  etat[moi] = FAIM;
  test(moi);
  sem_post(mutex);
  sem_wait(semTab[moi]);
}
void test(int moi){
  if (etat[moi]==FAIM && etat[GAUCHE] != MANGE \
          && etat[DROITE] != MANGE)
    {
      etat[moi] = MANGE;
      sem_post(semTab[moi]);
    }
}
```

Une solution (Dijkstra 1965) – 3

Programme exemple

```
void poser_fourchette(int moi){
  sem_wait(mutex);
  etat[moi] = PENSE;
  test(GAUCHE);
  test(DROITE);
  sem_post(mutex);
}
```

Une solution (Dijkstra 1965) – 3

Programme exemple

```
void poser_fourchette(int moi){
  sem_wait(mutex);
  etat[moi] = PENSE;
  test(GAUCHE);
  test(DROITE);
  sem_post(mutex);
}
```

Synthèse

Cette solution assure l'absence d'interblocages
Elle assure également l'absence de famines

Lecteur/rédacteurs

Principe

Il existe une donnée qui est modifiée (écrite) par un rédacteur unique et lu par de multiples lecteurs
Tant qu'il y a des lecteurs actifs le rédacteur ne peut pas modifier la donnée
Lorsque le rédacteur modifie la donnée les lecteurs ne peuvent pas y acceder

Lecteur/rédacteurs : solution (Dijkstra)

Programme exemple

```
lect=0; sem_t *M_Lect, *M_Red, *Red;
void debut_lecture(){
  sem_wait(M_Lect);
  Lect++;
  if (Lect==1)  sem_wait(Red);
  sem_post(M_Lect); }
void fin_lecture(){
  sem_wait(M_Lect);
  Lect--;
  if (Lect==0) sem_post(Red);  /* Problème ? */
  sem_post(M_lect); }
void debut_ecriture(){
  sem_wait(M_Red);
  sem_wait(Red);  }
void fin_ecriture(){
  sem_post(Red);
  sem_post(M_Red); }
```