

Systemes informatiques et applications concurrentes

Partie 6 - Sémaphores et moniteurs

Ivan KURZWEG

31 mai 2010

Systemes informatiques et applications concurrentes

by Ivan KURZWEG

Copyright © 2010 Ivan KURZWEG, ivan.kurzweg@gmail.com, 2010

Permission to use, copy, modify, and distribute this documentation *for any purpose with or without fee* is here by granted, provided that *the above copyright notice and this permission notice appear in all copies*.

The documentation is provided "as is" and the author disclaims all warranties with regard to this documentation including all implied warranties of merchantability and fitness. In no event shall the author be liable for any special, direct, indirect, or consequential damages or any damages whatsoever resulting from loss of use, data or profits, whether in an action of contract, negligence or other tortious action, arising out of or in connection with the use or performance of this documentation.

Table des matières

1	Sémaphores	1
1.1	Principes	1
1.1.1	Primitives P et V	1
1.1.2	Différents types de sémaphores	1
1.1.3	Implémentation sous <i>FreeBSD</i>	2
1.2	Exclusion mutuelle	2
1.3	Lecteurs - rédacteurs	2
1.4	Producteurs consommateurs	4
1.4.1	Un seul producteur, un seul consommateur	4
1.4.2	P producteurs, N consommateurs	5
1.4.3	Exemple de programme en C	5
1.5	Sémaphores privés	7
1.6	Repas des philosophes avec les sémaphores privés	7
1.6.1	Principes	7
1.6.2	Exemple en langage C	8
2	Moniteurs	11
2.1	Principes	11
2.2	Producteurs consommateurs avec moniteurs	11
2.3	Implémentation sur Unix (Posix)	12
2.3.1	Principes	12
2.3.2	Exemple	13
3	Exercices	15
3.1	Ordonnancement	15
3.2	Barrière	15
3.3	Le coiffeur endormi	16
3.4	Les buveurs	16
4	Corrections des exercices de la partie 5	16
4.1	Exclusion mutuelle	16
4.2	Algorithme de Dekker	16
4.2.1	Question 1	16
4.2.2	Question 2	17
4.2.3	Question 3	17
4.2.4	Question 4	18
5	Références	18
5.1	Références	18

Résumé

Partie 6 du cours "Systèmes informatiques et applications concurrentes". Cette sixième partie du cours aborde le contrôle de la concurrence via les sémaphores et les moniteurs.

1 Sémaphores

Nous avons abordé le concept des sémaphores dans le chapitre précédent. Nous allons nous intéresser plus précisément à ce mécanisme, en rappelant dans un premier temps les principes d'utilisation, puis en proposant des solutions aux problèmes classiques de concurrence.

1.1 Principes

Les sémaphores sont la solution proposée par Dijkstra pour résoudre le problème de l'exclusion mutuelle de processus. Ils sont avant tout un mécanisme de synchronisation, parfois employés abusivement comme moyen de communication inter-processus.

Un sémaphore est une variable entière partagée, positive, et accessible au travers de deux opérations :

- $P(S)$: si $S=0$ alors bloquer le processus sinon $S=S-1$ ("puis-je")
- $V(S)$: $S=S+1$ et réveiller un processus en attente ("Vas-y")

Les sémaphores doivent bien sûr être implémentés de manière à garantir que :

- les opérations P et V précédentes sont atomiques, c'est à dire que la suite des opérations les réalisant est non interruptible
- il existe un mécanisme permettant de gérer la file d'attente des processus bloqués à la suite d'opérations P

1.1.1 Primitives P et V

Les primitives P et V peuvent donc se décrire de la manière suivante :

```
Primitive P(sémaphore S) :
début
  S=S-1; //retrait d'une autorisation
  si S<0 alors état du processus appelant bloqué;
  placer son id dans la file F;
fin
Primitive V(sémaphore S) :
début
  S=S+1; //ajout d'une autorisation
  si S<=0 alors //il y a au moins un processus
bloqué
  choix et retrait d'un processus de F;
  réveil du processus;finsi;
fin
Primitive E0(sémaphore S,entier I) :
début
  S=I; S.F=vide; //I0
fin
```

1.1.2 Différents types de sémaphores

Il existe plusieurs implémentations des sémaphores sur les systèmes de type UNIX : on parle de *sémaphores POSIX*, de *MUTEX* et de *sémaphores UNIX* :

- Les *MUTEX POSIX* sont des sémaphores binaires, ne pouvant avoir que deux valeurs, *LOCKED* et *UNLOCKED*. Ils permettent de synchroniser uniquement des threads appartenant à un même processus.
- Les *sémaphores POSIX* ont une valeur entière positive ou nulle. Ils permettent de synchroniser des processus différents.
- Les *sémaphores UNIX* sont regroupés dans des tableaux. Les opérations P et V peuvent alors s'exécuter de manière atomique sur un ou plusieurs sémaphores de l'ensemble, permettant ainsi des demandes d'allocations simultanées de plusieurs exemplaires de ressources différentes, opération possible avec les *MUTEX*, mais alors plus complexe.

Dans ce premier chapitre, nous nous intéresserons aux sémaphores POSIX, et nous verrons les *MUTEX* dans le deuxième chapitre.

1.1.3 Implémentation sous *FreeBSD*

La définition des sémaphores POSIX sur *FreeBSD* se trouve dans le fichier `src/sys/sys/semaphore.h`. Les primitives et définitions principales sont les suivantes :

- `sem_t` : type associé à un sémaphore (valeur quelconque, opérations P et V standard). Un tel objet peut-être déclaré directement ou alloué dynamiquement (il s'agit alors d'un sémaphore non nommé) ou un pointeur sur un tel objet peut être récupérée au retour d'un appel à la fonction `sem_init` (il s'agit alors d'un sémaphore nommé)
- `int sem_init(sem_t *sem, int partage, unsigned int valeur)` : initialise le sémaphore non nommé (pointé par `sem`) à la valeur spécifiée en paramètre. Le paramètre `partage` a le rôle suivant :
 - s'il est non nul, le sémaphore peut être utilisé pour synchroniser des threads de processus différents
 - s'il est nul, l'utilisation du sémaphore est limitée aux threads du processus appelant.
- `int sem_destroy(sem_t *sem)` : détruit le sémaphore pointé par `sem`
- `sem_t *sem_open(char *nom, sem_t *sem, int mode, ...)` récupère un pointeur sur un sémaphore à partir de son nom
- `int sem_close(sem_t *sem)` : indique la fin de l'utilisation par le processus appelant du sémaphore pointé par `sem`
- `int sem_unlink(const char *nom)` supprime le sémaphore nommé de nom spécifié.
- `int sem_wait(sem_t *sem)` : c'est l'opération P
- `int sem_trywait(sem_t *sem)` : c'est l'opération P en mode non bloquant
- `int sem_post(sem_t *sem)` : c'est l'opération V
- `int sem_getvalue(sem_t *sem, int *valeur)` : permet de récupérer la valeur (positive ou nulle) courante du sémaphore spécifié

Sous *FreeBSD* il est à noter que les *sémaphores SYSTEM V* (sémaphores UNIX ou IPC) ne sont disponibles que si le noyau est compilé avec l'option `SYVSEM`.

1.2 Exclusion mutuelle

L'exclusion mutuelle peut se faire très simplement avec les sémaphores, puisqu'ils sont au départ prévus pour ça ! L'accès exclusif à une section critique est ainsi géré simplement par les opérations P et V. L'exemple suivant garanti l'accès exclusif à la section critique pour N processus s'exécutant en concurrence :

Exemple 1.1 Exclusion mutuelle par sémaphore

```
//Contexte commun : ressource ou variables partagées
sémaphore sem1; E0(sem1,1);
Processus Pi
début
tant que vrai faire
  P(sem1); //entrée en section critique
  Section_critique;
  V(sem1); //sortie de section critique
Hors section critique;
fin;
```

Dans l'exemple précédent, il ne peut y avoir d'interblocage, puisque quand aucun processus n'est en section critique, la valeur du sémaphore est 1, et un seul processus peut être en section critique.

1.3 Lecteurs - rédacteurs

Le modèle des lecteurs - rédacteurs s'appliquant quand il y a comptétion d'accès à un ensemble de données par un ensemble de processus, certains ayant un accès en lecture seule (les lecteurs), d'autres

en lecture/écriture (les rédacteurs). Plusieurs lectures simultanées sont possibles, mais les écritures sont en exclusion mutuelle.

Nous allons utiliser un premier sémaphore `Mutex_A` qui permettra de garantir l'exclusion mutuelle des rédacteurs. Un rédacteur accèdera à sa section critique par `P(Mutex_A)`, et un lecteur donnera accès au groupe des lecteurs par un `P(Mutex_A)`. Un variable partagée `NL`, protégée par un autre sémaphore `Mutex_NL`, permettra de compter le nombre de lecteurs en section critique :

```
//Contexte commun : ressource ou variables partagées
Entier NL=0;
sémaphore mutex_A, mutex_NL;
E0(mutex_A,1); E0(mutex_NL,1);

Processus unLecteur;          Processus unRédacteur;
tant que vrai faire          tant que vrai faire
//début lecture              //début écriture
P(mutex_NL);                 P(mutex_A);
NL=NL+1;                     écritures;
si NL=1 alors                //fin écriture
    P(mutex_A);              V(mutex_A);
finsi;                        fait;
V(mutex_NL);
lectures;
//fin lecture
P(mutex_NL);
NL=NL-1;
si NL=0 alors
    V(mutex_A);
finsi
V(mutex_NL);
fait;
```

Le problème de cette solution est qu'elle peut engendrer une famine des rédacteurs : si un lecteur est en section critique, alors les lecteurs auront toujours priorité. Une solution possible est alors de rétablir cette égalité en plaçant les requêtes dans une file d'attente gérée à l'ancienneté :

```
//Contexte commun : ressource ou variables partagées
Entier NL=0;
sémaphore mutex_A, mutex_NL;
E0(mutex_A,1); E0(mutex_NL,1);
sémaphore FIFO; E0(FIFO,1);

Processus unLecteur;          Processus unRédacteur;
tant que vrai faire          tant que vrai faire
//début lecture              //début écriture
P(FIFO);                     P(FIFO);
P(mutex_NL);                 P(mutex_A);
NL=NL+1;                     écritures;
si NL=1 alors                //fin écriture
    P(mutex_A);              V(mutex_A);
finsi;                        V(FIFO);
V(mutex_NL);                  fait;
V(FIFO);
lectures;
//fin lecture
P(mutex_NL);
NL=NL-1;
si NL=0 alors
    V(mutex_A);
finsi
V(mutex_NL);
fait;
```

1.4 Producteurs consommateurs

Nous avons expliqué le cas des producteurs / consommateurs dans le chapitre précédent : il s'agit d'une classe de problèmes tels que :

- Des processus "producteurs" produisent des informations, stockées dans un tampon de taille finie, vide à l'initialisation, que des processus "consommateurs" enlèvent du tampon
- Les vitesses et l'ordonnement des processus sont inconnus
- Tout message est déposé et retiré une seule fois

Il faut donc trouver une solution qui permette l'accès exclusif aux messages, et la gestion des cas où le tampon est plein (blocage des producteurs) ou vide (blocage des consommateurs).

Le problème peut être décomposé en 2 cas :

- un tampon à N cases, avec un seul producteur et un seul consommateur
- un tampon à N cases, avec P producteurs et C consommateurs

1.4.1 Un seul producteur, un seul consommateur

Pour garantir le bon fonctionnement d'un programme utilisant ce schéma avec un tampon de N cases, nous allons utiliser deux sémaphores :

- un sémaphore *Vide* initialisé à N pour le contrôle du nombre de cases vides : le producteur exécute **P(Vide)** pour réserver une case le consommateur exécute **V(Vide)** pour libérer une case
- un sémaphore *Plein* initialisé à 0 pour le contrôle du nombre de messages (cases pleines) : le producteur exécute **V(Plein)** pour signaler au consommateur un nouveau message dans le tampon le consommateur exécute **P(Plein)** pour attendre un message

```
//Contexte commun
Sémaphore Plein,Vide;
E0(Plein,0);E0(Vide,N)
Processus producteur          Processus consommateur
message m;                    message m;
tant que vrai faire           tant que vrai faire
  produire un message(m);      P(Plein);
  P(Vide);                      retirer(m);
  déposer(m);                  V(Vide);
  V(Plein);                     consommer(m);
fait;                           fait;
```

Dans ce cas, les propriétés sont bien respectées : accès exclusif à la section critique, et pas d'interblocage possible. Il est néanmoins généralement nécessaire de gérer le tampon :

Considérons le tampon comme étant géré de manière circulaire, et implémenté par un tableau. Les messages doivent être consommés dans l'ordre de leur production. Pour réaliser cette implémentation, nous allons utiliser deux variables, *tete* et *queue*, qui indiqueront respectivement au producteur et au consommateur quelle case du tableau ils doivent manipuler :

```
//Contexte commun
sémaphore Plein,Vide;
E0(Plein,0);E0(Vide,N);
message tampon[N];

Processus producteur          Processus consommateur
message m;                    message m;
entier queue=0;               entier tete=0;
tant que vrai faire           tant que vrai faire
  produire un message(m);      P(Plein);
  P(Vide);                      m=tampon[tete]; //retirer(m);
  tampon[queue]=m; //déposer(m); V(Vide);
  V(Plein);                     tete=(tete+1)%N;
  queue=(queue+1)%N;           consommer le message(m)
fait;                           fait;
```

1.4.2 P producteurs, N consommateurs

Dans ce cas plus général, la synchronisation doit assurer une exclusion mutuelle entre producteurs et consommateurs, de façon à garantir :

- *l'exclusion mutuelle entre producteurs* : plusieurs producteurs ne doivent pas déposer dans la même case. On partagera donc l'indice de production `queue`.
- *l'exclusion mutuelle entre consommateurs* : plusieurs consommateurs ne doivent pas prélever le même message. On partagera donc l'indice de consommation `tete`.

```
//Contexte commun
sémaphore Plein,Vide;
E0 (Plein, 0); E0 (Vide, N);
sémaphore mutexProd,mutexCons;
E0 (mutexProd, 1); E0 (mutexCons, 1);
Entier tete,queue=0;
Message tampon[N];

Processus unProducteur
message m;
tant que vrai faire
    produire un message (m);
    P (Vide);
    P (mutexProd);
    tampon[queue]=m; //déposer (m);
    queue=(queue+1)%N;
    V (mutexProd);
    V (Plein);
fait;

Processus unConsommateur
message m;
tant que vrai faire
    P (Plein);
    P (mutexCons);
    m=tampon[tete]; //retirer (m);
    tete=(tete+1)%N;
    V (mutexCons);
    V (Vide);
    consommer le message (m)
fait;
```

1.4.3 Exemple de programme en C

L'exemple ci-dessous est librement inspiré d'une solution proposée par Jean-Marie Rifflet, et présente une implémentation du problème en C. Pour compiler sous *FreeBSD* utiliser la commande `gcc -lpthread -lrt`. Il s'agit d'une implémentation avec 5 producteurs et 2 consommateurs, utilisant un tampon de 5 cases.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <fcntl.h>

#define TAILLE 5 /* taille du buffer */
#define PROD 2 /* nombre de producteurs */
#define CONS 2 /* nombre de consommateurs */

void *produire(void *);
void *consommer(void *);

typedef
    struct {
        pthread_t prodLid; /* identite de la thread productrice */
        int num; /* numero de l'objet produit par cette thread */
    } ELEM;
sem_t *occupe, /* semaphore <-> nombre d'emplacements occupes */
*libre; /* semaphore <-> nombre d'emplacements libres */
pthread_mutex_t mutex; /* mutex pour acces au buffer et aux variables */

int ecrire = 0, /* premier emplacement libre */
lire = 0; /* prochain emplacement a lire */

ELEM buffer[TAILLE]; /* le buffer */
```

```

pthread_t prod[PROD],      /* identite des threads productrices */
      cons[CONS];        /* identite des threads consommatrices */

main(){
  int loop;
  /* Creation et initialisation des deux semaphores occupe et libre */
  sem_unlink("/sem_occupe");
  sem_unlink("/sem_libre");
  occupe = sem_open("/sem_occupe", O_CREAT, 0666, 0);
  if (occupe == (sem_t *) -1){
    printf("erreur occupe\n");
    return 0;
  }
  libre = sem_open("/sem_libre", O_CREAT, 0666, TAILLE);
  if (libre == (sem_t *) -1){
    printf("erreur libre\n");
    return 0;
  }

  /* initialisation du generateur de nombres */
  srand(getpid());
  /* initialisation du mutex */
  pthread_mutex_init(&mutex, NULL);
  /* creation des threads productrices */
  for(loop = 0; loop < PROD; loop++)
    pthread_create(prod + loop, NULL, produire, NULL);
  /* creation des threads consommatrices */
  for(loop = 0; loop < PROD; loop++)
    pthread_create(cons + loop, NULL, consommer, NULL);
  /* mettre toutes les threads en concurrences */
  pthread_setconcurrency(PROD+CONS);
  pause();
}

/* fonction executee par les threads productrices */
void *produire(void *arg){
  int valeur = 0; /* variable locale a la thread : numero d'objet produit */
  pthread_t monId = pthread_self(); /* identite de la thread courante */
  ELEM objet;
  objet.prodLid = monId;

  while(1){
    sleep(1 + rand() % 3); /* temps de production */
    objet.num = valeur++;
    sem_wait(libre); /* y-a-t-il un emplacement libre ? */
    pthread_mutex_lock(&mutex); /* prendre le mutex */
    printf("***** producteur %d depose %d\n", objet.prodLid, objet.num);
    buffer[ecrire] = objet; /* déposer l'objet */
    ecrire = (ecrire + 1) % TAILLE; /* prochain emplacement de depot */
    pthread_mutex_unlock(&mutex); /* liberer le mutex */
    sem_post(occupe); /* signaler un emplacement occupe de plus */
  }
}

/* fonction executee par les threads productrices */
void *consommer(void *arg){
  pthread_t monId = pthread_self(); /* identite de la thread courante */

  while(1){
    sleep(rand() % 6);
    sem_wait(occupe); /* y-a-t-il un objet a extraire ? */

```

```

pthread_mutex_lock(&mutex); /* acquerir mutex */
printf("      ***** consommateur %d lit %d produit par %d\n",
      monId, buffer[lire].num, buffer[lire].prodLid);
lire = (lire + 1) % TAILLE; /* prochain emplacement a lire */
pthread_mutex_unlock(&mutex); /* liberer mutex */
sem_post(libre); /* signaler un emplacement libre de plus */
}
}

```

1.5 Sémaphores privés

Un sémaphore privé est un sémaphore dont l'opération P n'est accessible que par un seul procesus, les autres pouvant exécuter uniquement la primitive V. Les sémaphores privés sont utilisés pour qu'un processus puisse se bloquer lui-même, généralement en utilisant des données de contrôle (protégées en section critique).

1.6 Repas des philosophes avec les sémaphores privés

1.6.1 Principes

Les baguettes des philosophes sont allouées globalement. Les variables de contrôle reflètent l'état des philosophes. Un philosophe peut être à un instant donné dans l'un des trois états suivants :

- penseur [0],
- souhaitant manger [1],
- en train de manger [2].

Un philosophe ne pourra passer de l'état 0 à l'état 2 que si les deux philosophes voisins ne sont pas actuellement en train de manger (c'est-à-dire si leur état n'est pas 2). Si au moins l'un de ses voisins est dans l'état 2, le philosophe demandeur passera à l'état 1. Le test nécessaire sur l'état des deux voisins et la modification de la valeur de l'état du philosophe constituent une section critique qui va être protégée par un sémaphore binaire. Par ailleurs, afin d'éviter une attente active, il est souhaitable qu'un philosophe dans l'état 1 soit effectivement endormi : pour cela, on associe à chaque philosophe un sémaphore privé (seul lui peut réaliser une opération P sur ce sémaphore).

La solution peut donc se modéliser de la manière suivante :

```

//Contexte commun
type statut=(pense, demande, mange);
statut état[5];
//initialement tous les philosophes pensent
Pour i de 0 à 4 faire
    état[i]=pense;
fait;
sémaphore mutex; E0(mutex,1); //accès aux variables
sémaphore sempriv[5];
Pour i de 0 à 4 faire
    E0(sempriv[i],0);
f
ait;
Processus philosophe i
booleen OK;
    tant que vrai faire
        penser;
        // demande à manger
        P(mutex); //accès aux variables de contrôle
        état[i]=demande;
        si (état[(i+1)%5]mange et état[(i-1)%5]mange) alors
            état[i]=mange;OK=vrai;
        sinon
            OK=faux;
        fin;
    V(mutex);

```

```

    si non OK alors
        P(sempriv[i]);
    finsi;

    manger;

    // finit de manger
    P(mutex); //accès aux variables de contrôle
    état[i]=pense;
    //le philosophe i essaie de réveiller ses voisins
    si (état[(i+1)%5]=demande et état[(i+2)%5]m ange) alors
        état[(i+1)%5]=mange;V(sempriv[(i+1)%5]);
    finsi;
    si (état[(i-1)%5]=demande et état[(i-2)%5]m ange) alors
        état[(i-1)%5]=mange;V(sempriv[(i-1)%5]);
    finsi;
    V(mutex);

fait;

```

1.6.2 Exemple en langage C

Cette solution est proposée par J.M. Rifflet, et utilise de la mémoire partagée pour mémoriser les états des philosophes :

```

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/shm.h>
#include <signal.h>

#define PHILO 5 /* Nombre de philosophes */
#define PENSEUR 0
#define AFFAME 1
#define MANGEUR 2

pid_t mainPid; /* pid du processus principal */
int semid; /* l'ensemble de tous les sémaphores utilisés */
int shmId; /* la mémoire partagée par les processus philosophes */
/* 2 tableaux de 2 opérations sur sémaphores */
struct sembuf op_debut[2], /* avant de manger */
              op_fin[2]; /* après manger */
int *etat; /* les états des processus (attachement à shmId) */

void penser(),
     manger(int),
     philosophe(int);

/* opération P sur un sémaphore */
void P(int sem){
    static struct sembuf op;
    op.sem_op = -1;
    op.sem_num = sem;
    semop(semid, &op, 1);
}

/* opération V sur un sémaphore */
void V(int sem){
    static struct sembuf op;
    op.sem_op = 1;

```

```

op.sem_num = sem;
semop(semid, &op, 1);
}

/*****
Handler de teminaison sur reception du signal SIGINT:
- les processus philosophes sont terminees et
- l'ensemble de semaphores et le segment de memoire sont
  supprime par le processus pere avant qu'il ne se termine
*****/
void handlerFin(int sig){
    if(getpid() == mainPid){
        semctl(semid, IPC_RMID, 0); /* supprimer l'ensemble de semaphores */
        semctl(shmid, IPC_RMID, 0); /* supprimer le segment de memoire */
    }
    exit(0);
}

main(){
int i;
int val;
mainPid = getpid(); /* memoriser le pid du processus pere */
srand(getpid()); /* initialise le generateur de nombre */
/* Creation de l'ensemble des semaphores:
  les PHILO premiers sont les semaphores privees des philosophes
  le dernier est utilise comme semaphore d'exclusion mutuelle */
if((semid = semget(IPC_PRIVATE, PHILO + 1, 0666)) == -1){
    perror("semget");
    exit(-1);
}
/* Initialisation des semaphores :
  Les PHILO premiers sont a 0 et le dernier a 1 */
for(i=0; i<PHILO; i++)
    semctl(semid, i, SETVAL, 0);
semctl(semid, PHILO, SETVAL, 1);
/* creation du segment de memoire */
shmid = shmget(IPC_PRIVATE, 1024, 0666);
/* Attachement du segment en memoire comme un pointeur sur entier */
etat = (int *)shmat(shmid, NULL, 0);
/* Initialisation des etats des processus */
for(i=0; i<PHILO; i++)
    etat[i] = PENSEUR;
/* Installation du handler de signal */
signal(SIGINT, handlerFin);
/* Creation des PHILO processus philosophes */
for(i=0; i<PHILO; i++)
    if(fork() == 0)
        philosophe(i);
pause();
}

/*****
Code execute par un philosophe :
  la fonction recoit en parametre le numero du philosophe
*****/
void philosophe(int numero){
    /* initialisation du tableau de semaphores pour pouvoir manger:
      pour pouvoir manger, il faut prendre simultanement
      les fourchettes numero et (numero + 1) mod PHILO */
    penser();
    manger(numero);
}

```

```

void penser(){
    sleep(1 + rand() %10);
}

void manger(int num){
    P(PHILO); /* acquerir le mutex */
    printf("%d veut manger\n", num);
    etat[num] = AFFAME; /* je veux manger */
    /* tester si aucun de mes voisins ne mange */
    if ((etat[(num+PHILO-1)%PHILO] != MANGEUR) && (etat[(num+1)%PHILO] != MANGEUR ←
        )){
        etat[num] = MANGEUR; /* je vais pouvoir manger */
        V(num); /* je prepare mon semaphore prive */
    }
    V(PHILO); /* je rends le mutex */
    P(num); /* j'essaie de prendre mon semaphore */
    printf("    ***** %d commence a manger\n", num);
    sleep(1 + rand() % 4);
    printf("    ***** %d a fini de manger\n", num);
    P(PHILO); /* acquerir le mutex */
    etat[0] = PENSEUR; /* je me remets a penser */
    /* je regarde si mes voisins sont affames.
       Si un voisin est affame et que son autre voisin n'est pas en train
       de manger, je le reveille en incrementant son semaphore prive */
    if ((etat[(num+1)%PHILO] == AFFAME) && (etat[(num+2)%PHILO] !=2)){
        etat[(num+1)%PHILO] = MANGEUR;
        V((num+1)%PHILO);
    }
    if ((etat[(num+PHILO-1)%PHILO] == AFFAME) && (etat[(num+PHILO-2)%PHILO] !=2)) ←
        {
            etat[(num+PHILO-1)%PHILO] = MANGEUR;
            V((num+PHILO-1)%PHILO);
        }
    V(PHILO); /* je rends le mutex */
}

```

Le programme génère par exemple la trace suivante :

```

0 veut manger
    ***** 0 commence a manger
2 veut manger
    ***** 2 commence a manger
1 veut manger
3 veut manger
4 veut manger
    ***** 2 a fini de manger
    ***** 0 a fini de manger
    ***** 1 commence a manger
    ***** 3 commence a manger
    ***** 1 a fini de manger
    ***** 3 a fini de manger
    ***** 4 commence a manger
4 veut manger
3 veut manger
3 veut manger
1 veut manger
2 veut manger
    ***** 2 commence a manger
4 veut manger
    ***** 4 commence a manger
    ***** 4 a fini de manger
    ***** 2 a fini de manger
    ***** 1 commence a manger
    ***** 4 a fini de manger
2 veut manger

```

```
3 veut manger
4 veut manger
  ***** 4 commence a manger
4 veut manger
  ***** 4 commence a manger
  ***** 1 a fini de manger
  ***** 2 commence a manger
  ***** 4 a fini de manger
  ***** 4 a fini de manger
3 veut manger
4 veut manger
  ***** 4 commence a manger
  ***** 2 a fini de manger
  ***** 4 a fini de manger
4 veut manger
  ***** 4 commence a manger
3 veut manger
2 veut manger
  ***** 4 a fini de manger
4 veut manger
  ***** 3 commence a manger
3 veut manger
  ***** 3 commence a manger
  ***** 3 a fini de manger
  ***** 3 a fini de manger
  ***** 4 commence a manger
4 veut manger
4 veut manger
  ***** 4 a fini de manger
....
```

2 Moniteurs

2.1 Principes

Tous les problèmes de concurrence et de synchronisation peuvent être résolus par les sémaphores, mécanisme de bas niveau. Mais leur utilisation est délicate, et les algorithmes peuvent devenir complexes rapidement. Hoare et Brinch Hansen ont donc proposé en 1973 une implémentation de plus haut niveau permettant d'offrir de manière transparente des mécanisme d'exclusion mutuelle et de synchronisation : les moniteurs.

Un moniteur consiste tout d'abord en une structure de données (définissant donc un ensemble de données) et d'autre part en un ensemble de points d'entrée (un ensemble de procédures externes) avec la propriété qu'à un instant donné un seul processus (ou thread) peut être actif à l'intérieur du moniteur. La demande d'entrée dans un moniteur est donc synchrone : si un autre processus est dans le moniteur, l'appelant est bloqué tant que le processus qui y est actif n'en est pas sorti. L'exclusion mutuelle est donc supportée de manière implicite.

Plus précisément, il existe deux types de primitives :

- *wait* : qui libère l'accès au moniteur, puis bloque le processus appelant sur une condition
- *signal* : qui réveille sur une condition un des processus en attente à l'intérieur du moniteur (un processus qui a exécuté précédemment un *wait* sur la même condition)

Les données d'un moniteur contiennent des *variables conditions*, variable qui n'ont pas de valeur, ne peuvent être manipulables que par les primitives *wait* et *signal*, et qui représente la file d'attente des processus bloqués sur la même cause

2.2 Producteurs consommateurs avec moniteurs

L'exemple suivant propose la résolution du problème des producteurs consommateurs avec les moniteurs :

```

Type ProducteurConsommateur = moniteur
  {variables locales }
  Var Compte : entier ; Plein, Vide : condition ;
  {procédures accessibles aux programmes utilisateurs }
  Procedure Entry Déposer(message M) ;
  Début
    si Compte=N alors Plein.Wait ;
    dépôt (M) ;
    Compte=Compte+1;
    si Compte==1 alors Vide.Signal;
  Fin

Procedure Entry Retirer(message M) ;
Début
  si Compte==0 alors Vide.Wait ;
  retrait (M) ;
  Compte=Compte-1;
  si Compte==N-1 alors Plein.Signal;
Fin
Début {Initialisations }Compte= 0; Fin.

Processus Producteur
message M;
  Début
    tant que vrai faire
      Produire (M) ;
      ProducteurConsommateur.déposer (M)
  Fin

Processus Consommateur
message M;
  Début
    tant que vrai faire
      ProducteurConsommateur.retirer (M) ;
      Consommer (M) ;
  Fin

```

2.3 Implémentation sur Unix (Posix)

2.3.1 Principes

Un moniteur POSIX est l'association d'un mutex (sémaphore binaire accessible uniquement par les threads d'un même processus), et d'une variable condition qui sert de point de signalisation. Les structures C sont décrites dans les sources du système (`/usr/src/sys/sys/_semaphore.h` et `/usr/src/sys/sys/_pthreadtypes.h` pour *FreeBSD*):

- `pthread_cond_t` : il s'agit du type de base correspondant à une variable conditionnelle
- `pthread_cond_attr_t` : il s'agit du type de base correspondant aux attributs d'une variable conditionnelle
- `int pthread_condattr_init(pthread_condattr_t *attr)` : initialise la structure correspondant aux attributs d'une variable conditionnelle pointé par `attr` (la zone correspondante est supposée allouée) à un ensemble de valeurs par défaut
- `int pthread_condattr_destroy(pthread_condattr_t *attr)` : rend la structure pointée par `attr` inutilisable comme attributs de variable conditionnelle (jusqu'à réinitialisation par `pthread_condattr_init`)
- `int pthread_cond_init(pthread_cond_t *cond, pthread_cond_attr_t *attr)` : initialise la variable conditionnelle référencé par `cond` : la zone correspondante doit avoir été allouée au préalable. La zone pointée par `attr` doit par ailleurs avoir été initialisée ou `attr` peut être NULL, auquel cas les attributs par défaut sont appliqués à la variable conditionnelle.
- `int pthread_cond_destroy(pthread_cond_t *cond)` : rend la variable conditionnelle pointée par `cond` inutilisable (la zone pointée doit être réinitialisée)

- int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex) : primitive wait
- int pthread_cond_signal(pthread_cond_t *cond) : primitive signal

2.3.2 Exemple

Le programme suivant propose la résolution d'un problème de producteur-consommateur avec les moniteurs POSIX :

```
#include <stdio.h>
#include <stdlib.h>
#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()%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,0);
    pthread_cond_init(&vide,0);
    pthread_cond_init(&plein,0);
    /* creation des threads */
    pthread_create(tid, 0, (void * (*)()) Prod, NULL);
    pthread_create(tid+1, 0, (void * (*)()) 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);
}

```

La trace générée est la suivante :

```

Mess depose: 189
    Mess preleve: 189
Mess depose: 884
Mess depose: 523
Mess depose: 775
Mess depose: 570
Mess depose: 488
Mess depose: 465
Mess depose: 66
Mess depose: 943
Mess depose: 194
Mess depose: 669
    Mess preleve: 884
Mess depose: 820
    Mess preleve: 523
Mess depose: 28
    Mess preleve: 775
Mess depose: 563
    Mess preleve: 570
Mess depose: 38
    Mess preleve: 488
Mess depose: 195
    Mess preleve: 465
Mess depose: 33
    Mess preleve: 66
Mess depose: 70
    Mess preleve: 943
Mess depose: 118
    Mess preleve: 194
Mess depose: 84
    Mess preleve: 669
Mess depose: 570
    Mess preleve: 820
    Mess preleve: 28
    Mess preleve: 563
    Mess preleve: 38
    Mess preleve: 195
    Mess preleve: 33
    Mess preleve: 70

```

...

3 Exercices

3.1 Ordonnement

On suppose que sur Unix on peut définir des variables x et y communes à deux processus comme suit :

```
shared long x = 0 ;
shared long y = 0 ;
```

Deux processus exécutent les codes suivants :

Processus P1	Processus P2
***	***
$x = x + 1;$	$x = x * 2;$
$y = y + 1;$	$y = y * 2;$
<code>printf("x=%d,y=%d\n", x, y);</code>	<code>printf("x=%d,y=%d\n", x, y);</code>
***	***

- Ces processus s'exécutent sur un système UNIX dont la politique d'ordonnement est du type *round robin*. Quelles peuvent être les couples de valeurs affichées par chacun des deux processus ?
- En utilisant un sémaphore, modifier le code pour assurer que les `printf` affichent toujours des valeurs identiques pour x et y .

3.2 Barrière

Ecrire les procédures d'accès à un objet de type barrière, c'est à dire dont le fonctionnement est le suivant :

- la barrière est fermée à son initialisation,
- elle s'ouvre lorsque N processus sont bloqués sur elle.

Définition de la barrière :

```
struct
{
  Sema Sema1;      /* Sémaphore de section critique */
  Sema Sema2;      /* Sémaphore de blocage sur la barriere */
  int Count;       /* Compteur */
  int Maximum;     /* Valeur déclenchant l'ouverture */
} Barriere;
```

Question 1 : Complétez le code de la procédure d'initialisation

```
void Init (Barriere *B; int Maximum)
{
  Init (B->Sema1, );
  Init (B->Sema2, );
  B->Count = ;
  B->Maximum = ;
}
```

Question 2 : Complétez le code de la procédure d'attente donné ci-dessous :

```
void Wait (Barriere *B)
{
  Boolean Do_Wait = True;
  int I;
  ***;
  B->Count++;
  if (B->Count == B->Maximum )
  {
```

```

for (I=0 ; I < (B->Maximum-1) ; I++ ) ***;
B->Count = 0;
Do_Wait = ***;
}
***;
if (Do_Wait) ***;
}

```

3.3 Le coiffeur endormi

Soit un salon de coiffure comportant n fauteuils pour les clients qui attendent, un fauteuil pour la personne coiffée, et un coiffeur. Si un client entre dans le salon et qu'il n'y a aucun fauteuil de libre, il s'en va. Si le client entre dans le salon, et que le coiffeur dort, il le réveille et se fait coiffer, si le coiffeur coiffe, il s'assoit et attend. Enfin, le coiffeur coupe des cheveux si il y a des clients, et s'endort sinon.

A l'aide de sémaphores, écrivez en pseudo code les programmes des processus `client()` et `coiffeur()`. On considère que les fonctions `coiffe()` et `se_fait_coiffer()` existent.

3.4 Les buveurs

Pour boire, une personne a besoin de 3 choses : eau, glaçons, et verre. On compte 3 personnes qui ont soif, chacun disposant de l'un des éléments nécessaires. Un quatrième personne, un serveur, possède une quantité illimitée de tous les ingrédients. Si personne ne boit, le serveur place deux des trois éléments (choisis de manière aléatoire) sur une table. La personne qui a soif les prend et boit un verre d'eau. Une fois qu'elle a terminé, elle avertit le serveur et le processus se répète.

Ecrivez un moniteur pour contrôler les personnes qui ont soif ainsi que le serveur dans le programme suivant :

```

void Serveur () {
    while (true)
    {
        buveur.servir();
    }
}

void buveur() {
    while (true)
    {
        buveur.prendre_ingredient(type);
        boire();
        buveur.avertir_Serveur(type);
    }
}

```

4 Corrections des exercices de la partie 5

4.1 Exclusion mutuelle

4.2 Algorithme de Dekker

4.2.1 Question 1

On utilise une seule variable booléenne M telle que $M = \text{vrai}$ si un des processus se trouve dans sa section critique, faux sinon. Ecrire le programme Vérifier que l'exclusion mutuelle ne peut être ainsi programmée. On suppose qu'on attribue à P_0 et P_1 , les identifiants 0 et 1

```

/*déclarations et initialisation des variables globales*/
int M = 0 ;//M est a faux
/* etat d'occupation de la ressource*/
Tâche P0
While (1) {

```

```

    /*entree_SC */
    while (M );/* attente active*/
    M=1;//M est a vrai
    Section_critique();
    /*sortie_SC*/
    M=0;
    Section_non_critique() ;
}
Tâche P1
While (1) {
    /*entree_SC */
    while (M );/* attente active*/
    M=1;
    Section_critique();
    /*sortie_SC*/
    M=0;
    Section_non_critique() ;
}

```

Une exécution « entrelacée » de l'entrée en Section critique, par les deux tâches entraîne le non-respect de l'exclusion mutuelle, `entree_SC` n'est pas atomique

4.2.2 Question 2

```

/*déclarations et initialisation des variables globales*/
int T = 0 ;
/* T vaut 0 ou 1 tâche autorisée à entrer en section critique*/
Tâche P0
While (1) {
    /*entree_SC */
    while (T==1 );/* attente active*/
    Section_critique();
    /*sortie_SC*/
    T=1;
    Section_non_critique() ;
}
Tâche P1
While (1) {
    /*entree_SC */
    while (T==0 );/* attente active*/
    Section_critique();
    /*sortie_SC*/
    T=0 ;
    Section_non_critique() ;
}

```

Cette solution règle le problème de l'exclusion mutuelle, une seule tâche peut entrer en SC (valeur de T). C'est un fonctionnement à l'alternat donc si un processus tombe en panne hors section critique, l'autre processus sera bloqué. La condition c) n'est pas respectée.

4.2.3 Question 3

```

/*déclarations et initialisation des variables globales*/
int C[2]={0,0};
Tâche P0
While (1) {
    /*entree_SC */
    while (C[1]);/* attente active*/
    C[0]=1 ;
    Section_critique();
    /*sortie_SC*/
    C[0]=0;
    Section_non_critique() ;
}

```

```

}
Tâche P1
While (1) {
  /*entree_SC */
  while (C[0]);/* attente active*/
  C[1]=1 ;
  Section_critique();
  /*sortie_SC*/
  C[1]=0;
  Section_non_critique() ;
}

```

L'exclusion mutuelle n'est pas garantie, chaque tâche peut trouver la variable C de l'autre tâche à faux et décider son entrée en SC Une autre solution pourrait être que chaque processus positionne sa variable à true, puis teste la variable de l'autre processus, dans ce cas on risque un blocage mutuel indéfini.

4.2.4 Question 4

```

/*déclarations et initialisation des variables globales*/
int C[2]={0,0} ;
int T=0
Tâche P0
while (1) {
  /*entree_SC */
  C[0]=1 ;
  while (C[1]) { /* P1 est en SC ou a demande a entrer en SC*/
    while (T==1) C[0]=0 ;/*P1 est prioritaire, retrait de P0*/
    C[0]=1 ;/*T=0 P0 recandidate*/
  }
  Section_critique();
  /*sortie_SC*/
  C[0]=0;
  T=1 ;
  Section_non_critique() ;
}
Tâche P1
While (1) {
  /*entree_SC */
  C[1]=1 ;
  while (C[0]) { /* P0 est en SC ou a demande a entrer en SC*/
    while (T==0) C[1]=0 ;/*P0 est prioritaire, retrait de P1*/
    C[1]=1 ;/*T=1 P1 recandidate*/
  }
  Section_critique();
  /*sortie_SC*/
  C[1]=0;
  T=0 ;
  Section_non_critique() ;
}

```

5 Références

5.1 Références

- [1] Paolo ZanellaYves Ligier, *Architecture et technologie des ordinateurs* , Dunod , isdn : ISBN 2 10 003801 X, 199,2002.
- [2] Matt WelshKalle DalheimerLar Kaufman, *Le systeme Linux* , O'Reilly , 1995, 1997, 1999, 2000.
- [3] Joelle Delacroix, *Linux* , Dunod , 2003.

-
- [4] Jean-Marie Rifflet, *La programmation sous Unix et les cours en ligne*, EdiScience , 1986, 1989, 1993, 2002.
 - [5] Michael W. Lucas, *Absolute Freebsd: The Complete Guide to Freebsd*, No Starch Press , 2007.
 - [6] Michael W. Lucas, *FreeBSD 7.0*, No Starch Press , 2007.
 - [7] Nicholas P. Carter, *Architecture de l'ordinateur*, EdiScience - Schaum's , 2002.
 - [8] J. Archer Harris, *Systèmes d'exploitation*, EdiScience - Schaum's , 2002.
 - [9] Eric Steven Raymond, *The art of Unix Programming*, Thyrsus Enterprises , 2003.