L3 Informatique : Cours Systèmes et Réseaux
Synchronisation de processus et partage de ressources

Olivier Togni
Université de Bourgogne, IEM/LIB
Bureau G206
olivier.togni@u-bourgogne.fr

14 novembre 2022

Plan du cours

1. Exclusion mutuelle et sémaphores
   1. Ressources critiques
   2. Techniques d'exclusion mutuelle
   3. Applications : Blocage, Lecteurs/rédacteurs, Producteurs/consommateurs
   4. Aspects techniques : IPC Unix
2. Mémoire partagée et verrouillage de fichier
3. Processus légers
   1. Principes
   2. Implantation
   3. Threads POSIX

Ressource critique

Problèmes des ressources (variables) partagées: lorsque plusieurs processus s'exécutent en accédant simultanément à des ressources communes :

- volontairement s'ils coopèrent pour traiter un même problème,
- involontairement parce qu'ils sont obligés de se partager les ressources de l'ordinateur.

$\Rightarrow$ résultats imprévisibles si pas de précaution.

Exemples

Compte bancaire sauvé sur disque / mémoire commune / imprimante / ...

Ressource critique

- même pour machine mono-processeur/mono-coeur car
- entrelacement des exécutions (au niveau des instructions machine)
- noyau : exécution du processus / interruption / commutation de contexte

Exemple

A : ressource critique ; P1/P2 : incrémentation de A

Code de P1/P2 :
lire A depuis disque
A=A+1
ecrire A sur disque

Ressource critique

Exemple

A : ressource critique ; P1/P2 : incrémentation de A
Selon entrelacement, contenu de A différent

```
P1                              P2
lire  A depuis disque
---------------interruption/commutation---------------
                                lire A depuis disque
---------------interruption/commutation---------------
A=A+1
ecrire A sur disque
---------------interruption/commutation---------------
                                A=A+1
                                ecrire A sur disque
```

Section critique

- Section critique d'un programme = partie où se produit le conflit d'accès à l'objet partagé
  (l'exécution simultanée de deux sections critiques appartenant à des ensembles différents et ne partageant pas de variable ne pose pas de problème)
- Les sections critiques doivent être exécutées en exclusion mutuelle: avant d'exécuter une section critique, un processus doit s'assurer qu'aucun autre processus n'est en train d'exécuter une section critique du même ensemble.
- Dans le cas contraire, il ne devra pas progresser plus avant tant que l'autre processus n'aura pas terminé sa section critique.

Propriétés des sections critiques

- Exclusion mutuelle
- Progression : un processus demandant l'accès à une section critique doit obtenir satisfaction au bout d'un temps borné (attente bornée)
- Indépendance : les processus hors de la section critique ne bloquent pas l'accès à celle-ci
- Banalité : aucun processus ne joue de rôle particulier, et le choix d'entrée en section critique ne dépend que des processus en attente

Programmation des sections critiques

- Prélude avant la section critique
- Section critique
- postlude après la section critique

Exemple

Code de P1/P2 :

demander A

lire A depuis disque

A=A+1

ecrire A sur disque

restituer A

...

Programmation des sections critiques

Exemple : une exécution possible

```
P1                                P2
demander A
---------------interruption/commutation---------------
                                  demander A
---------------interruption/commutation---------------
lire A depuis disque
A=A+1
ecrire A sur disque
restituer A
---------------interruption/commutation---------------
                                  demander A
                                  lire A depuis disque
                                  A=A+1
                                  ecrire A sur disque
                                  restituer A
```

Programmation des sections critiques

Exemple : une autre exécution possible

```
P1                                P2
---------------interruption/commutation---------------
                                  demander A
                                  lire A depuis disque
                                  A=A+1
                                  ecrire A sur disque
                                  restituer A
---------------interruption/commutation---------------
demander A
lire A depuis disque
A=A+1
ecrire A sur disque
restituer A
```

Techniques d'exclusion mutuelle

- Masquage des interruptions : avant d'entrer en section critique, puis restauration en sortie. Utilisé par le système, mais interdit aux process utilisateur car dangereux.
- Instructions matérielles atomiques : Test and Set, Swap
- Instructions logisielles système :
  - Fichier verrou
  - sémaphores
  - ...

Teest and Set

C'est un verrou par rotation :
Boolean TS(boolean v) :

- renvoie la valeur de v et l'affecte à VRAI
- opération atomique

Utilisation :

variable commune cadenas
While (TS(cadenas)); // prélude, attente active
// section critique
cadenas = FAUX; // postlude

Premier arrivé met cadenas à VRAI, bloque les autres

Pas d'équité (progression)

Swap

Swap (boolean a, boolean b) :

- échange le contenu de a et b
- opération atomique

Utilisation :

variable commune verrou initialisé à FAUX
clé = VRAI // prélude
While(clé) Swap(clé,verrou);
// section critique
verrou = FAUX; // postlude

Premier arrivé met verrou à VRAI, bloquant les autres

Sur archi Intel, AMD : CMPXCHG

Exclusion logicielle

- Algotihme de Peterson
- pas besoin d'instruction machine atomique
- tableau drapeau[i] : le process i veut rentrer en s.c.
- tour : numéro du processus entrant en s.c.

Utilisation :

drapeau[i]=vrai; tour = j; //prélude
while(drapeau[j] && tour == j); // attente active
// section critique
drapeau[i] = FAUX; // postlude

Exclusion logicielle : sémaphores

- vairables entières gérées par le système
- opération Init(sem,val) : valeur de départ
- opération P(sem) : décrémentation bloquante : réservation de ressource (*prolaag*)
- opération V(sem) : incrémentation : libération de ressource (*verhogen*)
- E. W. Dijkstra (1965)

V et P sont des opérations atomiques indivisibles : on est sûr qu'aucun autre process ne peut accéder à un sémaphore tant qu'une opération sur ce sémaphore n'est pas terminée ou bloquée.

Sémaphores : exclusion mutuelle

Exclusion mutuelle = sémaphore binaire
Init(Mutex,1)

P1	P2
phase normale	phase normale
P(Mutex)	P(Mutex)
phase critique	phase critique
V(Mutex)	V(Mutex)
phase normale	phase normale

Schéma des lecteurs/rédacteurs

- fichier/zone de mémoire communs
- accès en lecture par les lecteurs
- accès en écriture par les rédacteurs
- objectif 1 : cohérence (pas d'écriture simultanées)
- objectif 2 : stabilité (pas de lecture/écriture simultanées)
- Conclusion, sur le fichier/la zone :
  - une seule écriture simultanée
  - une ou plusieurs lectures (pas de conflit)

Schéma des lecteurs/rédacteurs : rédacteur

- un rédacteur exclut tous les autres
- utilisation d'un schéma d'exclusion mutuelle simple

Utilisation :

Init(acces,1) //une seule ressource : mutex
P(acces) // zone libre ?
// section critique : accès en écriture
V(acces) // libération

Schéma des lecteurs/rédacteurs : lecteurs

- premier/dernier lecteur : bloque/libère les rédacteurs
- suis-je le premier ? compteur commun en exclusion mutuelle

Utilisation :

```
Init(mutex,1)   //une seule ressource : mutex
P(mutex)   // accès au compteur
NL=NL+1
if(NL==1)     // si je suis le 1er lecteur
    P(acces)  // bloque les rédacteurs
V(mutex)
// accès en lecture sur la zone
P(mutex)
NL=NL-1
if(NL==0)    // si je suis le dernier
    V(acces)
V(mutex)
```

Schéma des producteurs-consommateurs

- 2 processus partagent un même tampon de N places
- places numérotées de 0 à N-1, utilisées circulairement
- un processus P remplit le tampon
- un processus C vide le tampon
- Si tampon plein, P ne doit plus produire
- si tampon vide, C ne doit plus vider
- P et C ne doivent pas travailler sur la même place

1 producteur / 1 consommateur : 2 sémaphores

Init(Plein,0); Init(Vide,N)

```
Producteur

i=0
P(Vide)
tampon[i]=messages
i=i+1 mod N
V(Plein)
```

```
Consommateur

j=0
P(Plein)
message=tampon[j]
j=j+1 mod N
V(Vide)
```

Généralisation à n producteur et n consommateur

Indice d'écriture partagé i
Indice de lecture partagé j

Init(Mutexi,1); Init(Mutexj,1)
Init(Plein,0); Init(Vide,N)
i,j=0

```
Producteur

P(Vide)
P(Mutexi)
tampon[i]=messages
i=i+1 mod N
V(Mutexi)
V(Plein)
```

```
Consommateur

P(Plein)
P(Mutexj)
message=tampon[j]
j=j+1 mod N
V(Mutexj)
V(Vide)
```

Interblocage

Accès à deux ressources exclusives R1, R2
Init(R1,1); Init(R2,1)

```
P1                                P2

P(R1)                             P(R2)
P(R2)                             P(R1)
utilisation R1 et R2              utilisation R1 et R2
V(R2)                             V(R1)
V(R1)                             V(R2)
```

Conditions de l'interblocage

- Exclusion mutuelle
- occupation en attente : un processus occupant une ressource en attend une autre
- pas de réquisition : libération des ressources sur seule volonté des processus
- attente circulaire

Traitement de l'interblocage

- Guérison
  - laisser l'interblocage se produire
  - le détecter : cycle dans le graphe d'allocation
  - détruire les processus / réquisitionner les ressources
  - coûteux en temps ou nb de processus détruits
- Prévention
  - pas d'exécution avant d'avoir toutes ses ressources
  - ordre total sur les ressources (R1 toujours avant R2, etc)
- Evitement : à chaque allocation, voir si elle peut conduire à un interblocage
- Autruche : on suppose que c'est très rare et si cela arrive, tout relancer

Exclusion mutuelle par fichier

- Un fichier sert à indiquer la présence d'un processus en section critique.
- Avant d'entrer en section critique: tentative d'ouverture du fichier avec `open(fic, O_CREAT | O_EXCL)`
- Fichier effacé en fin de section critique par `unlink(fic)`.

Les IPC (Inter Process Communications) UNIX

- sémaphores,
- segments de mémoire partageables,
- file de messages.

Par nature un objet IPC est partageable par plusieurs processus et géré de façon globale par le système.
Principe identique pour les 3 ressources:

- clé IPC associée à la ressource (identifiant externe de type key_t) et identifiant interne (entier) dans le processus pour manipulation,
- droits d'accès définis à la création,
- commandes `ipcs` liste des ressources IPC allouées
  `ipcrm` libérer une ressource

primitive `ftok()` pour créer une clé publique

Sémaphores IPC

Gérés sous forme d'un tableau, on alloue un tableau de sémaphores et on effectue des opérations sur les différents éléments du tableau:

- création du tableau: `semget(...)`,
- manipulation: `semctl(...)`,
- accès: `semop(...)` permet de réserver/libérer N unités de ressource à la fois

Mémoire partagée

un segment de mémoire peut être simultanément attaché à l'espace d'adressage virtuel de plusieurs processus ou plusieurs fois à des adresses différentes d'un même processus.

- `shmget()` : création,
- `shmctl()` : modification des droits ou du propriétaire, destruction,
- `shmat()` : attachement à espace d'adressage virtuel, puis lecture/écriture si droits ok,
- `shmdt()` : détachement,

Verrouillage de fichier

Verrouillage de fichier par `flock()` (BSD): Permet de mettre des verrous partagés ou exclusifs sur les fichiers:

- verrou partagé autorise l'accès à plusieurs lecteurs simultanés,
- verrou exclusif limite l'accès à un seul écrivain (ni lecteur ni autre écrivain).

`flcok()` prend en paramètre le descripteur du fichier, mais le verrou porte sur le fichier, pas sur le descripteur donc si on duplique le descr (par dup()), le verrou reste actif sur le 2ième descr;

Verrouillage de régions par `fcntl` (POSIX): permet de traiter non pas un fichier mais une partie du fichier appelée région. Une région est définie par une taille en octets et une position de début dans le fichier. Utilise une structure `flock`.

Files de messages IPC

- Définies à l'origine sur Unix System V (utilisation différente des queues de messages Posix.4)
- Permet l'échange de message entre processus (plutôt que séquence d'octets comme avec les tubes)
- On peut associer un type (entier) à un message et ensuite demander la réception d'un type particulier
- Même fonctionnement que sémaphores et mémoire partagée IPC pour désignation externe (key_t) et interne (entier) mais trois espaces de désignation distincts
- Protection gérée par UID, GID

Files de messages IPC

- `msgget()` : création,
- `msgsnd(int msgid, void *pt_msg, int long, int option)` : envoi du message pointé par pt_msg qui est une structure qui contient le type et le message
- `msgrcv(int msgid, void *pt_msg, int longmax, int option, long type)` : réception du premier message dans la file de type donné
- si type = -k est négatif : permet de récupérer un message de type 1 si présent, sinon de type 2, sinon 3, etc, sinon k
- `msgctl()` : contrôle de la file (destruction : 2eme paramètre = IPC_RMID)

Threads

Thread = sous-processus

également appelé

- activité
- processus léger

Définition

Processus = ressources + exécution

Ressources:

- fichiers ouverts
- processus enfant
- alertes en attentes
- gestionnaire des signaux
- ...

Exécution:

- compteur ordinal
- registres pour variables en cours
- pile
- ...

Définition

Modèle des threads : ressources et exécution sont des entités séparées

Les processus regroupent les ressources

Les threads sont les entités planifiées pour leur exécution par le processeur

$\Rightarrow$ plusieurs threads par processus (multithreading) traités chacun à leur tour sur machine monoprocesseur ou en parralèle sur machine multiprocesseur

Définition

Element par processus	Element par thread
Espace d'adressage	Compteur ordinal
variables globales	Registres
Fichiers ouverts	Pile
Processus enfant	Etat
Alertes	

État = en cours d'exécution, bloqué (attente évennement externe ou autre thread), prêt, arrêté
Pile: contient les paramètres des procédures qui n'ont pas encore retourné de valeur

Avantages/inconvénients

Avantages:

- modèle de programmation plus simple: facilite la gestion de nombreuses activités en même temps
- plus faciles à créer et détruire que les processus (100 fois plus rapide)
- utile pour implanter des programmes parallèles

Inconvénients:

- compliquent le système d'exploitation
  Ex: que doit faire fork() si le père possède plusieurs threads?
- besoin de verrous/sémaphores pour les sections critiques

Implantation des threads

Plusieurs solutions:

- dans l'espace utilisateur
- dans le noyau
- hybride

Implantation dans l'espace utilisateur

- Pas d'existence au niveau du noyau
- fonctionne même sur les systèmes qui ne supportent pas les threads (rare à l'heure actuelle)
- chaque processus doit gérer sa propre table de threads
- basculement plus rapide
- chaque processuspeut avoir son propre algorithme d'ordonnancement
- problèmes avec les appels systèmes bloquants

Implantation dans le noyau

- table de threads globale gérée au niveau du noyau
- création/terminaison de thread passe par un appel système au noyau
- recyclage des threads sur certains systèmes
- simples à mettre en oeuvre

Implantation hybride

Une méthode pour combiner les avantages des threads utilisateur et noyau est d'employer des threads noyau et de multiplexer des threads utilisateur sur un ou plusieurs threads noyau

Le noyau ordonnance les threads noyau et le processus ordonnance les threads utilisateur

Le standard POSIX

Historiquement, chaque fabriquant avait une version propriétaire d'implantation des threads
$\Rightarrow$ complique la tâche des développeurs et non portable
Le standard IEEE POSIX 1003.1c à été proposé et adopté par la plupart (dans le domaine Unix)

On parle de threads POSIX ou Pthreads si leur implantation respecte ce standard

Création

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

Crée une nouvelle activité pour exécuter la routine indiquée, appelée avec l'argument arg
Les attributs sont utilisés pour définir la priorité et la politique d'ordonnancement
thr contient l'identificateur de l'activité créée

Terminaison et attente

```
void pthread_exit (void *status);
```

Termine l'activité appelante en fournissant un code de retour

`pthread_exit(NULL)` est implicitement exécuté en cas de terminaison du code de l'activité sans appel de `pthread_exit`

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

Attend la terminaison de l'activité indiquée et récupère le code retour

L'activité ne doit pas être détachée

Identification

```
pthread_t pthread_self (void);
```

Renvoie l'identificateur de l'activité appelante

```
int pthread_equal (pthread_t thread_1, pthread_t thread_2);
```

Renvoie vrai si les arguments désignent la même activité, faux sinon

Libération des ressources

```
int pthread_detach (pthread_t thread);
```

Détache l'activité thread

Normalement, les ressources allouées pour l'exécution d'une activité (pile...) ne sont libérées que lorsque l'activité s'est terminée et qu'un appel à join pour cette activité a été effectué

Pour éviter de devoir se synchroniser sur la terminaison d'une activité dont on compte ignorer le code retour, on peut détacher cette activité, auquel cas les ressources sont libérées dès la terminaison de l'activité

Il est interdit d'attendre la terminaison (join) d'une activité détachée

Exemple

```
#include <pthread.h>
#include <stdio.h>
void *routine (void *arg)
{
    int *status = malloc (sizeof(int)); /* pour renvoyer le code de retour */
    printf ("Arg = %d\n", *(int *)arg); /* casting vers (int *) necessaire */
    *status = *(int *)arg * 2;
    pthread_exit (status);
}

int main()
{
    pthread_t un_p;
    int erreur, argument = 3;
    int *resultat;
    erreur = pthread_create (&un_p, NULL, routine, &argument);
    if (erreur != 0) fprintf(stderr,"Echec creation de thread: %d\n",erreur);
    pthread_join (un_p, (void **)&resultat);
    printf ("Resultat: %d\n", *resultat);
    free (resultat);
    exit(0);
}
```

Création/destruction d'un verrou

```
int pthread_mutex_init (pthread_mutex_t *mutex,
                        const pthread_mutex_attr *attr);
```

ou `pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;`
Création d'un verrou

```
int pthread_mutex_destroy (pthread_mutex_t *mutex);
```

Destruction d'un verrou

Verrouillage/déverrouillage

```
int pthread_mutex_lock (pthread_mutex_t *mutex);
```
Verrouillage, avec blocage en attente si déjà verrouillé. Renvoie 0 si ok

```
int pthread_mutex_trylock (pthread_mutex_t *mutex);
```
Verrouillage si possible et renvoie 0, sinon renvoie EBUSY si le verrou est déjà verrouillé (EINVAL ou EFAULT en cas d'erreur)

```
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```
Déverrouillage. Seule l'activité qui a verrouillé mutex a le droit de le déverrouiller (en cas de tentative de déverrouiller un mutex verrouillé par une autre activité, le comportement est indéfini)

Création/destruction d'une variable condition

```
int pthread_cond_init (pthread_cond_t *cond,
                       const pthread_cond_attr *attr);
```

ou `pthread_cond_t cond = PTHREAD_COND_INITIALIZER;`
Crée une variable condition

```
int pthread_cond_destroy (pthread_cond_t *cond);
```

Détruit la variable condition

Attente

```
int pthread_cond_wait (pthread_cond_t *cond,
                       pthread_mutex_t *mutex);
```

L'activité appelante doit posséder le verrou mutex
L'activité est alors bloquée sur la variable condition après avoir libéré le verrou
L'activité reste bloquée jusqu'à ce que la variable condition soit signalée et que l'activité ait réussi à réacquérir le verrou

Signal

```
int pthread_cond_signal (pthread_cond_t *cond);
```

Signale la variable condition : une activité bloquée sur la variable condition est réveillée

Elle sera effectivement débloquée quand elle réussira à réacquérir ce verrou

Il n'y a aucun ordre garanti pour le choix de l'activité réveillée

L'opération signal n'a aucun effet s'il n'y a aucune activité bloquée sur la variable condition (pas de mémorisation).

```
int pthread_cond_broadcast (pthread_cond_t *cond);
```

Toutes les activités en attente sont réveillées, et tentent d'obtenir le verrou correspondant à leur appel de cond_wait

Pour aller plus loin

- Christophe Blaess, Développement système sous Linux, 5e edition, Eyrolles, 2019
- Joëlle Delacroix, LINUX, programmation système et réseau, 4e édition, Dunod, 2016