Les processus sont **isolés** par défaut

Mécanisme pour communiquer nécessaire

Exemple: processus parent et enfant TP1

- Il faut indiquer au parent l'échec de exec()
- Le code de retour est très limité

Signaux

Forme d'interruption logicielle

Peut survenir à n'importe quel moment

- S'imbrique par dessus le contexte d'exécution principal!

L'application peut définir une fonction de rappel par signal

L'application peut masquer (ignorer) certains signaux temporairement

Bloquer et attendre un signal

bloquer temporairement le signal SIGINT

sigprocmask() : empêcher certains signaux d survenir (sauf SIGKILL et SIGSTOP)

- Par exemple, si on ne veut pas être dérangé

Suspendre le processus jusqu'à réception d'un signal. Attente passive d'un signal

pause() : attend jusqu'au prochain signal

Ajouter une modification temporaire du masque des signaux

sigsuspend(mask) : combinaison de sigprocmask() suivi de pause()

Les tubes anonymes: création

Un tube de communication anonyme est créé par l'appel système:

```
int pipe(int p[2]).
```

Cet appel système crée deux descripteurs de fichiers. Il retourne, dans p, les descripteurs de fichiers créés : p[0] contient le descripteur réservé aux lectures à partir du tube p[1] contient le descripteur réservé aux écritures dans le tube. Les descripteurs créés sont ajoutés à la table des descripteurs de fichiers du processus appelant. Seul le processus créateur du tube et ses descendants (ses fils) peuvent accéder au tube (duplication de la table des descripteurs de fichiers).

Si le système ne peut pas créer de tube pour manque d'espace, l'appel système pipe() retourne la valeur -1, sinon il retourne la valeur 0.

L'accès au tube se fait via les descripteurs (comme pour les fichiers ordinaires).

Les tubes anonymes sont, en général, utilisés pour la communication entre un processus père et ses processus fils, avec un processus qui écrit sur le tube, appelé processus écrivain, et un autre qui lit à partir du tube, appelé processus lecteur.

La séquence d'événements pour une telle communication est comme suit :

1. Le processus père crée un tube de communication anonyme en utilisant l'appel système pipe( ) ;
2. Le processus père crée un ou plusieurs fils en utilisant l'appel système fork( ) ;
3. Le processus écrivain ferme le descripteur de fichier, non utilisé, de lecture du tube ;
4. De même, le processus lecteur ferme le descripteur de fichier, non utilisé, d'écriture du tube ;
5. Les processus communiquent en utilisant les appels système:

read(fd[0], buffer, n) et write(fd[1], buffer,n);

1. Chaque processus ferme son fichier lorsqu'il veut mettre fin à la communication via le tube.

Fonctionnement signaux

Le système d'exploitation sauvegarde l'état du programme (registres et instruction courante)

Ajoute sur la pile les informations suivantes:

- Les signaux masqués
- L'adresse de l'instruction courante
- Information à propos du signal

Le programme reprend l'exécution dans le gestionnaire de signal (fonction de rappel)

Au retour de cette fonction, le contexte d'origine est restauré.

Mémoire partagée

Principe : référencer les mêmes pages mémoires dans deux processus différents

shm_open() : ouvrir un descripteur de fichier sous /dev/shm

ftruncate() et mmap() requis ensuite pour accéder à l'espace

shm_unlink() : supprime l'espace partagé lorsque le dernier utilisateur a terminé

Commande ipcs : lister les ressources partagées

Attention : conditions critiques possibles sur des accès

Les tubes anonymes

Chaque tube a un nombre de lecteurs et un nombre d'écrivains

La fonction read() d'un tube retourne 0 (fin de fichier), si le tube est vide et le nombre d'écrivains est 0

L'oubli de la fermeture de descripteurs peut mener à des situations d'interblocage (attente indéfinie)

La fonction write() dans un tube génère le signal SIGPIPE, si le nombre de lecteurs est 0

Par défaut, les lectures et les écritures sont bloquantes

Redirection de stdin et stdout

La duplication de descripteur permet à un processus de créer un nouveau descripteur (dans sa table des descripteurs) synonyme d'un descripteur déjà existant.

dup() : duplique un descripteur de fichier

dup2() : duplique un descripteur de fichier avec

Ces fonctions peuvent être utilisées pour réaliser des redirections des fichiers d'entrées et sorties standards vers les tubes de communication.

```
// macros utiles pour se souvenir que read=0 et write=1
#define R 0 #define W 1
int main() {
 int fd[2]; // tableau de descripteurs de
 fichiers pipe(fd); // nouveau tube sans nom
 char message[100]; // tampon du message
 int nboctets; char *msg = "foo bar";
 if (fork() == 0) { /* * Producteur * * L'enfant
 ferme le descripteur de lecture non utilisé, * écrit le
 message dans le tube et ferme le descripteur *
 d'écriture. */
 printf("Écriture du message %s\n", msg);
 close(fd[R]); write(fd[W], msg, strlen(msg)
 1); close(fd[W]); } else {
 /* * Consommateur * * Le parent ferme le descripteur
 d'écriture non utilisé, * copie le message du tube vers
 le tampon, affiche le message * puis ferme le
 descripteur de lecture. */
 close(fd[W]);
 nboctets = read(fd[R], message, 100);
 printf("Lecture %d octets : %s\n",
 nboctets,message);
 close(fd[R]); }
 return 0; }
```

Intercepter un signal

L'appel système sigaction() permet de redéfinir le gestionnaire associé à un signal.

La structure sigaction :

- La fonction de rappel : void (*sa_handler)(int)
- L'ensemble des signaux à bloquer pendant l'exécution de la fonction de rappel: sigset_t sa_mask
- Options : int sa_flags

On peut associer un même gestionnaire à des signaux différents.

Envoyer un signal

```
int result = kill(pid, SIGUSR1);
```

Appel système kill() : peut envoyer n'importe quel signal (pas seulement SIGKILL)

Un processus peut envoyer un signal à un autre processus

Si pid>0 le signal sig est envoyé à ce processus

Si pid=0 le signal est envoyé à tous les processus du groupe de l'appelant

Les deux processus doivent appartenir au même utilisateur

Seul le super-utilisateur (root) peut envoyer un signal au proce d'un autre utilisateur

- Sinon, un utilisateur pourrait par exemple terminer les pro d'un autre utilisateur!

- SIGINT : interruption du clavier
- SIGALRM : expiration d'une minuterie
- SIGSEGV: erreur de segmentation
- SIGFPE: erreur de point flottant
- SIGCHLD: processus enfant a quitté
- SIGTRAP : point d'arrêt (débogueur)
- SIGTERM : demande de quitter
- SIGSTOP : mise en pause du processus
- SIGKILL : Sashay away

```
jmp_buf jmpbuffer;          SIGSEGV
void segfault_handler(int sig) {
printf("Signal SIGSEGV %d\n", sig);
longjmp(jmpbuffer, 1); }
int main() { printf("Démarrage...\n"); // Surcharger la fonction de rappel
pour SIGSEGV
signal(SIGSEGV, segfault_handler); // Initialiser jump buffer
if (setjmp(jmpbuffer) == 0) { // Exécution normale
  int *ptr = NULL; // Causer SIGSEGV en accédant à un pointeur NULL
  printf("KABOUM!\n");
  *ptr = 42;
 } else { // setjmp(jmpbuffer) retourne 1 à cause de longjmp
  printf("On a survécu à l'erreur de segmentation!\n");
} printf("Fin normale du programme\n"); return 0; }
```

Les tubes anonymes: remarques

Attention aux chaines de caractères

Deux protocoles habituels :

- Terminé par zéro '\0'
- Écrire la taille, puis les données (on peut omettre le caractère nul final)

La communication bidirectionnelle est possible en utilisant deux tubes (un pour chaque sens

Les tubes nommés

Les tubes de communication nommés fonctionnent aussi comme des files de discipline FIFO (first in first out)

Ils sont plus polyvalents que les tubes anonymes car ils offrent, en plus, les avantages suivants :

- Ils ont chacun un nom qui existe dans le système de fichiers
- Ils peuvent être utilisés par des processus indépendants (pas limité à la relation parent-enfant)
- Ils existeront jusqu'à ce qu'ils soient supprimés explicitement

Créés par la commande « mkfifo» ou « mknod » ou par l'appel système mknod( ) ou mkfifo()

Tubes anonymes

Liaisons unidirectionnelles de communication

Se comporte comme une file FIFO

Accédés par deux descripteurs de fichiers (lecture et écriture)

Taille limitée (mémoire en mode noyau)

La lecture dans un tube est destructrice : l'information ne peut être lue qu'une seule fois dans un tube

Le tube est détruit lorsque tous les descripteurs du tube seront fermés

Socket

Connexion locale ou réseau

Permet d'échanger des messages entre deux processus sur des ordinateurs différents (qui ne partage pas de mémoire)

Canal **bidirectionnel**

- Contrairement aux tubes qui sont unidirectionnels
- Permet read() et write() sur le même descripteur

Abstraction de la complexité de la communication

- Division en paquets, retransmission, etc.

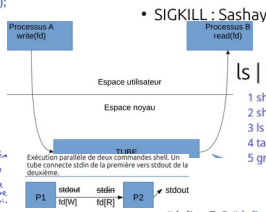
Serveur	Client
socket() : création du socket	• socket() : création du socket
bind() : utilisation du port	• connect() : établissement de la connexion
listen() : accepter des connexions	
accept() : attend une connexion entrante	

Sockets : lectures courtes

À cause d'un délai réseau, il se peut que la quantité à lire soit inférieure à celle attendue

Il faut s'attendre à recevoir l'information par fragment : c-à-d lire dans une boucle

Tant que la quantité demandée n'est pas atteinte, alors on fait une réception



ls | tail | grep

1 sh
2 sh
3 ls
4 tail
5 grep

```
#define R 0 #define W 1
int main(int argc, char* argv[]) {
int fd[2];                Redirection
pipe(fd); // création d'un tube sans nom
if (fork() > 0) { // parent
  close(fd[R]); // fermeture du descripteur de
                   lecture non utilisé
  dup2(fd[W], 1); // copie fd[W] en tant que
                   descripteur 1 (stdout)
  close(fd[W]); // fermeture du descripteur
                   d'écriture
  // exécute le producteur
  if (execvp(argv[1], argv + 1) == -1)
{ perror("execvp parent"); }
} else { // enfant
  close(fd[W]); // fermeture du descripteur d'écriture
                   non utilisé
  dup2(fd[R], 0); // copie fd[R] dans le descripteur
                   0 (stdin)
  close(fd[R]); // fermeture du descripteur de lecture
// exécute le consommateur
if (execvp(argv[2], argv + 2) == -1)
{ perror("execvp enfant"); } } return 0; }
```

Les 4 conditions pour l'exclusion mutuelle

Deux processus ne peuvent être en même temps dans leurs sections critiques.

Aucune hypothèse ne doit être faite sur les vitesses relatives des processus et sur le nombre de processeurs.

Aucun processus suspendu en dehors de sa section critique ne doit bloquer les autres processus.

Aucun processus ne doit attendre trop longtemps avant d'entrer en section critique (attente bornée).

Techniques pour l'exclusion mutuelle

Support matériel

- Désactivation des interruptions
- Instructions atomiques
- Verrou actif

Algorithmique

- Algorithme de Dekkers
- Algorithme de Peterson

Support du système d'exploitation

- Attente passive
- Communication inter-processus
- Barrière, Semaphore et Mutex

```
void entrer_region(int process)
{  int autre;
   autre = 1 – process;   //l'autre processus
   interesse[process] = TRUE;  //indiquer qu'on est intéressé
   tour = process; //la course pour entrer se gagne ici
   while (tour == process && interesse[autre] == TRUE);
}
```

attente active (busy waiting)

Si un autre processus appelle également entrer_region et modifie tour pour qu'il pointe vers lui-même (tour = autre), alors la condition tour == process deviendra false pour le processus actuel.

```
void quitter_region (int process )
{  interesse[process] = FALSE;
}
```

Problème : attente active = consommation du temps CPU

Défis de l'exclusion mutuelle

Interblocage (*deadlock*)

- Un cycle d'attente empêche le système d'évoluer.

Famine (*starvation*)

- Une tâche n'arrive jamais à s'exécuter parce qu'elle n'est jamais choisie par rapport à une autre tâche, malgré qu'elle soit prête.

Interblocage actif (*livelock*)

- Une situation dans laquelle au moins deux processus s'exécutent et dont l'état change sans cesse sans que le travail progresse.

Inversion des priorités

- Une tâche moins prioritaire détient un verrou qu'une tâche plus prioritaire a besoin pour s'exécuter.

Surcoût

- Quel est l'impact de performance du choix de synchronisation?

```
typedef struct semafun {
    int count;
    int lock;
    GQueue *queue;
} semafun_t;
```

Utilisation d'un verrou actif pour une courte durée

```
void mysemaphore_wait(sema_t *sem) {
    spin_lock(&sem->lock);
    sem->count--;
    if (sem->count < 0) {
        /* s'ajouter à la queue */
        spin_unlock(&sem->lock);
        /* se mettre en veille */
    } else {
        spin_unlock(&sem->lock);
    }
}
```

```
void mysemaphore_signal(sema_t *sem) {
    spin_lock(&sem->lock);
    sem->count++;
    if (sem->count <= 0) {
        /* pop queue */
        spin_unlock(&sem->lock);
        /* réveiller le processus */
    } else {
        spin_unlock(&sem->lock);
    }
}
```

Masquage des interruptions : Désactivation temporaire des interruptions pour éviter les accès simultanés dans les systèmes monoprocesseurs.

Attente active (Spinlocks) : Utilisation de boucles actives pour attendre qu'un verrou se libère, adapté aux situations avec contention faible et attentes courtes.

Verrous utilisant des instructions atomiques :

Sémaphores : Sémaphores binaires : Utilisés principalement pour l'exclusion mutuelle. Sémaphores comptés : Pour gérer les ressources multiples ou les conditions d'attente.

Mutex (Mutual Exclusion) : Un verrou binaire utilisé spécifiquement pour l'exclusion mutuelle entre processus ou threads.

Futex (Fast User-space Mutex) : Permet un verrouillage rapide sans appel système en cas d'absence de contention.

Sémaphores (1)

Pour contrôler les accès à un objet partagé, E. W. Dijkstra (1965) suggéra l'emploi d'un nouveau type de variables appelées sémaphores.

Un sémaphore est un compteur entier qui désigne le nombre d'autorisations d'accès disponibles.

Chaque sémaphore a un nom et une valeur initiale.

Les sémaphores sont manipulés au moyen des opérations :

P() désigné aussi par down ou wait (Proberen)

V() désigné aussi par up ou signal (Verhogen)

Utilisation de Test-and-Set-Lock (TSL) et thread_yield()

```
struct mutex_t {
int t; // t=0 si non verrouillé
}
mutex_lock(mutex_t *m) {
  while (TSL(m->t) != 0) {
  // verrou occupé
  // céder le processeur
  thread_yield();
  } }
mutex_unlock (mutex_t *m) {
m->t = 0;
}
```

Les opérations sur une donnée partagée par plusieurs processeurs peut conduire à une condition critique (résultat imprévisible, aléatoire)

L'état final dépend de l'ordre spécifique des opérations

Mécanisme de synchronisation permet d'empêcher les conditions critiques

But : assurer l'accès exclusif (exclusion mutuelle)

Problèmes liés au masquage des interruptions

La désactivation est généralement réservée au système d'exploitation. Un processus en espace utilisateur ne peut pas désactiver les interruptions, sinon il pourrait empêcher l'ordonnanceur de s'exécuter en bloquant les interruptions de l'horloge.

Si les interruptions ne sont pas réactivées, alors le système ne réagirait plus aux commandes et ne fonctionnerait plus normalement.

Elle n'assure pas l'exclusion mutuelle, si le système n'est pas monoprocesseur (le masquage des interruptions concerne uniquement le processeur qui a demandé l'interdiction). Les autres processus exécutés par un autre processeur pourront donc accéder aux objets partagés.

Instruction exécutant plusieurs opérations indivisible

- Fetch-And-Add
- Test-And-Set
- Compare-And-Swap

Cette instruction effectue deux opérations atomiques en une seule : 1)Lit la valeur d'une variable. 2)Ajoute une valeur à cette variable.

Lit la valeur d'une variable et la définit à une nouvelle valeur (souvent true), en 1 seule opération atomique. Utilisation classique : verrouillage binaire (spinlock).

Vérifie si une variable contient une valeur attendue, et si oui, remplace cette valeur par une nouvelle.

Permet d'implémenter des primitives de synchronisation efficacement

```
lock cmpxchg reg/mem, reg
```

Réservation explicite du bus	Nom de l'instruction	Opérande de destination	Opérande source

```
lock cmpxchg [rdi], rcx ; compare la valeur [rdi] avec rcx
                        ; si ([rdi] == rax)
                        ;   ZF = 1, [rdi] = rcx
                        ; sinon
                        ;   ZF = 0, rax = [rdi]
```

spinlock

- Rapide: une instruction atomique (ex: cmpxchg)
- Réservé pour une attente courte et probabilité de contention faible
- Fréquent dans le noyau (ex: interruption)

L'opération P(S) décrémente la valeur du sémaphore S si cette dernière est supérieure à 0. Sinon le processus appelant est mis en attente du sémaphore. L'opération V(S) incrémente la valeur du sémaphore S, si aucun processus n'est bloqué par l'opération P(S). Sinon, l'un d'entre-eux sera choisi et redeviendra prêt.

Chacune de ces deux opérations doit être implémentée comme une opération indivisible.

Chaque sémaphore a une file d'attente.

– File FIFO : sémaphore forte

– File LIFO : sémaphore faible (famine possible)

Les sémaphores POSIX sont implantés dans la librairie <semaphore.h>

Le type sémaphore est désigné par le mot : sem_t.

L'initialisation d'un sémaphore est réalisée par l'appel système :

Int sem_init(sem_t*sp, int pshared, unsigned int count);

où

– sp est un pointeur sur le sémaphore à initialiser

– count est la valeur initiale du sémaphore

– pshared indique si le sémaphore est local au processus ou non (0 pour local et non null pour partagé).

La suppression d'un sémaphore :

Int sem_destroy(sem_t * sem);

Acces exclusif → B.Mutex : principle:proteger l acces concurrent a une rsc partage

Contient un compteur → semaphore(pourrais aller plus10)

Scrutation → spinlock (soit 0 ou 1 → cmpxcchg) (attend actif, et les autres sont passive)

Point de rendez-vous → Barriere

Singnale un changement d etat → variable de condition

Chq obj au java est un verrou

Test vs verification: test est facile,mais pas l'exactitude, verification est diff mais exactitude

Quand un fil d exec tente d'obtenir un mutex alors qu'il est deja occupe? La fil bloque et est reveille lorsque verrou est libere

Il faut empêcher l'utilisation simultanée de la variable commune solde. Lorsqu'un thread (ou un processus) exécute la séquence d'instructions (1 et 2), l'autre doit attendre jusqu'à ce que le premier ait terminé cette séquence.

Section critique : suite d'instructions qui opèrent sur un ou plusieurs objets partagés et qui nécessitent une utilisation exclusive des objets partagés.

Chaque thread (ou processus) a ses propres sections critiques.

Les sections critiques des différents processus ou threads qui opèrent sur des objets communs doivent s'exécuter en exclusion mutuelle.

Avant d'entamer l'exécution d'une de ses sections critiques, un thread (ou un processus) doit s'assurer de [illegible] des objets partagés manipulés par la secti[illegible]

Assurer l'exclusion mutuelle?

- Encadrer chaque section critique par des opérations spéciales qui visent à assurer l'utilisation exclusive des objets partagés.
- Si P1 est dans sa section critique et P2 désire entrer dans sa section critique, alors P2 attend que P1 quitte sa section critique

P1

```
/* début du block protégé */
critical_section_enter(&lock);
tmp = x;
tmp++;
x = tmp;
/* fin du block protégé */
critical_section_leave(&lock);
```

P2

```
/* début du block protégé */
critical_section_enter(&lock);
tmp = x;
tmp--;
x = tmp;
/* fin du block protégé */
critical_section_leave(&lock);
```

```
/* début du block protégé */
spin_lock(&lock);
x++;
/* fin du block protégé */
spin_unlock(&lock);
```

La boucle se que le verrou n'e

```
spin_lock:
    mov rcx,1
spin_lock_retry:
    xor rax, rax
    lock mcpxchg [rdi], rcx

    jnz spin_lock_retry
    ret

spin_unlock:
    mov qword [rdi], 0
    ret
```

; adresse du verrou dans rdi
; valeur de comparaison
; remettre a zero rax
; compare la valeur [rdi] avec rcx
; si ([rdi] == rax)
;   ZF = 1, [rdi] = rcx
; sinon
;   ZF = 0, rax = [rdi]
; adresse du verrou dans rdi
; remettre le verrou à 0

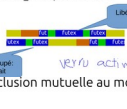
Exclusion mutuelle avec SÉ

- Remplacer l'attente active par l'attente passive
- SLEEP() est un appel système qui suspend l'appelant en attendant qu'un autre le réveille
- WAKEUP(process) : est un appel système qui réveille le processus process.

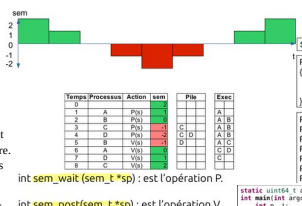
Synchronisation: attente passive

Mutex implémenté avec futex()

- Rapide si aucune contention: aucun appel système pour prendre le verrou (ex: cmpxchg)
- Si le verrou est déjà utilisé:
  - futex(FUTEX_WAIT): attente passive (schedule)
  - futex(FUTEX_WAKE): verrou libéré



Exemple d'exécution d'une sémaphore



Temps	Processus	Action	sem	Pile	Exec
0			2		
1	A	P(s)	1		A
2	B	P(s)	0		A B
3	C	P(s)	-1	C	A B
4	D	P(s)	-2	C D	A B
5	B	V(s)	-1	D	A C
6	A	V(s)	0		C D
7	D	V(s)	1		C
8	C	V(s)	2		

Exclusion mutuelle au moyen de sémaphores

Les sémaphores **binaires** permettent d'assurer l'exclusion mutuelle :

Semaphore sem = 1;

```
Processus P1 :
{   P(sem)
    Section_critique _de_P1() ;
    V(sem) ;
}
```

```
Processus P2 :
{  P(sem)
   Section_critique_de_P2();
   V(sem) ;
}
```

```
P1: P(mutex)  → mutex=0
P1 : entame Section_critique_de_P1();
P2 : P(mutex) → P2 est bloqué → file de mutex
P1: poursuit et termine Section_critique_de_P1();
P1 : V(mutex)→ débloque P2 → file des processus prêts
P2: Section_critique_de_P2();
P2: V(mutex) → mutex=1
```

```
int sem_wait (sem_t *sp) : est l'opération P.

int sem_post(sem_t *sp) : est l'opération V.

int sem_trywait( sem_t *sp) : décrémente la valeur du sémaphore sp, si sa valeur est supérieure à 0 ; sinon elle retourne une erreur (sem_wait non bloquant).

int sem_getvalue(sem_t * sem, int * sval) : retourne dans sval la valeur courante du sémaphore.
```

```
static uint64_t a = 0; // variable globale
int main(int argc, char **argv) {
  int p, i;
  pthread_t t1, t2;
  init_count_sem();
  pthread_create(&t1, NULL, count_passive, &a);
  pthread_create(&t2, NULL, count_passive, &a);
  pthread_join(t1, NULL);
  pthread_join(t2, NULL);
  printf("%lld^%d a=%" PRId64 "\n", getpid(), a);
  return EXIT_SUCCESS;
}
```

Deux fils accèdent en même temps à la variable globale « a »

Sémaphore	Mutex
Compteur	• Binaire
Opérations P(), V() par n'importe quel processus	• Spécifique à l'exclusion mutuelle
Synchronisation plus générale	

```
void *count_passive(void *arg) {
    volatile uint64_t *var = (uint64_t *) arg;
    volatile uint64_t i, j;
    sem_wait(&sem_lock);
    for (i = 0; i < MAX; i++) {
        *var = *var + 1;
    }
    sem_post(&sem_lock);
    return NULL;
}
```

```
static sem_t sem_lock;
void init_count_sem() {
    sem_init(&sem_lock, 0, 1);
}
```

La sémaphore empêche deux fils sérialise l'exécution des boucles

SIGTERM: tache termine proprement

SIGSTOP: a[res envoyer un signale→ proc bloque

shm_open()→ premet de creer un segment de mem partagee entre 2 proc

Tube anonyme: transfer unidirec entre 2 proc

Pour transmettre sur un socket, on doit utiliser read and write

Un fil d'esec demarre 4 fils d'exec avec pthread_creat(). On veut utiliser une barrier dans le fil principal pour s'assourer qu'ils sont tous demmarrer. Combien barrier? Fil +1

Solution prob au moyen des semaphore → besoin 2 semaphore: 1 occupe de 0 et 2e, libre et taille de N du tempon

Il sert a bloquer et debloquer le producteur

--

Philosophe interblocage: il prend pas meme forchette → il partagent (exclu mut) → semaphore mutex → prob de famine et equite (penser, manger et faim)

Il faut garantir que si un proc demande d entrer en sec critique au bout d un temps fini

--

- Situation qui survient quand un cycle d'attente se forme entre les tâches. Interblocage
- Format des exécutables sur Windows. PE- portable exec
- Appel système avec lequel sont implémentées les variables de condition sous Linux. futex
- Signal qu'une tâche ne peut pas masquer ou ignorer.
SIGSTOP - SIGKILL
- Nom de l'algorithme permettant de remplacer en priorité des ressources qui n'ont pas été utilisées depuis longtemps. LRU list recently
- Appel système qui permet d'attendre plusieurs descripteurs de fichiers simultanément. select / multiplexage (nom de catégorie)
- Technologie sous Linux permettant d'isoler un groupe de processus et utilisée pour réaliser des conteneurs. namespace / controlgroup
- Appel système qui permet d'obtenir des statistiques sur le temps processeur utilisé par la tâche courante. getrusage
- Méthode de communication bidirectionnelle entre deux processus sur des ordinateurs distincts. socket
- Appel système utilisé pour confirmer que les modifications en mémoire ont bien été écrites sur disque. sync

6.5 Un ordinateur utilise une table de pages avec 1024 entrées pour chaque processus. La lecture dans la table de pages prend 5ns. Une cache de pré-traduction d'adresse (TLB) contient 32 entrées et peut être accédée en 1ns (au lieu de 5ns). Quel est le taux de succès requis afin d'avoir un temps d'accès moyen de 2ns?

Tmps moy = delais x prop succes + delais x prop echec

Le temps moyen sera pour un taux de succès s :

s * 1ns + (1 − s) * 5ns = 2ns.

On a s + 5 − 5s = 2, en isolant s on obtient s = ¾ = 75%.

2 = 1x5 + 5 x(1-S) → -3 = -45 = 75%

```
void f1() {
  printf("A");   ← ABB
  int x = fork();
  wait(NULL);
  printf("B");
}
```

```
void f3() {
  int x = fork();
  wait(NULL);   ← A
  if (x > 0) {
    execlp("sleep", "sleep", "1",
    printf("B");
  }
  printf("A");
}
```

```
void f2() {     AB ou BA
  int x = fork();
  if (x == 0) {
    printf("A");
  } else {
    printf("B");
  }
  wait(NULL);
}
```

```
void f4() {     A B
  int a = 10;
  printf("A");
  if (fork() == 0) {
    close(1);   ← stdout = 1
  }
  wait(NULL);
  printf("B");  ← Ø
}
```

```
// sockfd : descripteur du socket de réception
// file_fd : descripteur pour enregistrer les don
// size : taille des données à recevoir, en octet
// block_size : taille des blocs transférés, en o
//
// read(int fd, void *buf, int len);
// write(int fd, void *buf, int len);
//                int size, int block_size) {
int reception(int sockfd, int file_fd, int size,
  int n = size / block_size;
  int r = size % block_size;
  int ret = -1;

  char *buf = malloc(block_size);

  // blocks entiers
  for (int i = 0; i < n; i++) {   block_size peut être pas la valeur de retour
    if (read(sockfd, buf, block_size) < 0) {
      goto err;
    }
    if (write(file_fd, buf, block_size) < 0) {
      goto err;
    }
  }

  // gestion du bloc incomplet
  if (r > 0) {
    if (read(sockfd, buf, r) < 0) {
      goto err;
    }
    if (write(file_fd, buf, r) < 0) {
      goto err;
    }
  }

  ret = 0;

err:
  free(buf);
  return ret;
}
```

solution: bouvle while de recep
while(recu<taille)
valread=read(sock,boffer+recu,taille-recu)
if (valread<0)
perror(); break;
recu=+

Vous êtes responsable de la mise en œuvre d'un nouveau site d'enchères sur Internet. Ce site permet à plusieurs utilisateurs de miser simultanément. Expliquez comment vous pourriez vous assurer que le logiciel ne présente aucune condition critique, ni de possibilité d'interblocage. Comparez les alternatives en terme d'avantages et d'inconvénients.

Réponse :
test de charge: simuler des utilisateurs, surcharger de requetes
(verification formulle) -> centrale nucleaire
Base de donnes -> valgrind/treadSanitizer

a) Si vous avez accès à la base de données locale d'authentification d'un ordinateur, comment faire pour retrouver un accès valide? (5 pts)
b) Sur le système Hopper, on exécute le code des étudiants. Quel risque cela peut-il représenter? Donner un exemple précis. (5 pts)
c) Comment faire pour mitiger ce risque? (5 pts)

-etudient qui fait fork bomb
-ce qui fait boucle infinie
-Inodes -> qte fixe pour un sys. fichier
-on peut mettre une limitation sur temps
-isolet les tache

Un processus a un espace mémoire virtuel de 16 pages (numérotées de 0 à 15), mais l'application est limitée à 4 pages seulement. Le processus accède aux pages dans la séquence indiquée dans le tableau ci-bas.

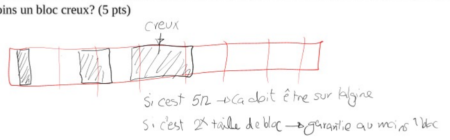
a) Complétez le tableau en suivant l'algorithme LRU. Marquez les fautes de pages par une étoile (*) et un accès sans faute par un crochet (√). (5 pts)
b) Combien de fautes de page majeure se produit-il au total? (5 pts)
c) L'espace mémoire est limité à 4 pages pour le processus, mais l'ordinateur peut contenir toutes les pages dans la cache. Les fautes de page majeures et mineures causent un délai de 100us et 10us respectivement. Assumez que le premier accès à une page cause une faute de page majeure, la pagination des pages présentes en cache causent une faute de page mineure et qu'aucune faute de page survient si la page est déjà dans l'espace mémoire du processus. Calculez le délai de pagination moyen. (5 pts)

temp	0	1	2	3	4	5	6	7	8	9	10
Accès	1	2	10	15	2	1	2	1	2	6	10
Page 1	1	1	1	1						1	
Page 2		2	2	2	(2)					2	
Page 3			10	10		1				6	
Page 4				15						15	
Faute	*	*	*	*	/	/	/	/	/	*	

[ ]
0 [1]
1 [2,1]
2 [10,2,1]
3 [15,10,2,1]
4 [2,15,10,1]
...
[2,1,15,10]
9 [6,2,1,15,10]

a) Considérez un fichier volumineux contenant d'importantes sections à zéro. Il est possible de réduire le nombre de blocs occupés sur disque dans cette situation grâce au support pour fichier creux du système de fichier. Décrivez une procédure permettant de transformer le fichier de départ en fichier creux. (5 pts). lab 7 ou 8
b) Si les blocs sont de 512 octets, quelle est la taille minimale d'une section de zéro pour qu'il y ait au moins un bloc creux? (5 pts)

b)



Si c'est 512 → ça doit être sur l'aligne
Si c'est 2x taille de bloc → garantie au moins 1 bloc

- Appel système qui lance un nouveau programme. exec → réinitialise
- Mécanisme qui permet de minimiser la copie de mémoire lors de la création d'un nouveau processus. COW → copy-on-write, copie paresseuse
- État d'une tâche quand elle peut s'exécuter, mais qu'elle n'a pas de processeur d'attribué. preempte
- Stratégie d'allocation de fichiers sur disque sujette à la fragmentation externe. Continue
- Nom d'une fonction de la librairie C qui utilise l'appel système clone(). pthread_create / fork
- Appel système permettant d'obtenir le nombre de faute de page faites jusqu'à maintenant par le programme courant. getrusage()
- Identifie un fil d'exécution de manière unique. TID
- Outil permettant d'obtenir la trace des appels système qu'un programme effectue. Strace
- Appel système permettant de déplacer le curseur d'un fichier ouvert. lseek
- Nom de l'appel système permettant d'accéder à un fichier en mémoire sans utiliser read() ou write(). mmap
-mkdir("",0755); → plus rapide,1, plus efficace
-execlp("mkdir", "mkdir", "test"); → lent 3
-shell → 2

```
int save_points(char *fname, struct points *pts) {
int fd = open(fname, O_CREAT | O_TRUNC | O_WRONLY, 0644);
write(fd, &pts->n, sizeof(pts->n));
int pos = 0; int bloc = 1024;
int taille = pts->n * sizeof(struct vec3);
while (pos < taille) {
  int reste = taille - pos;
  if (reste > bloc) { reste = bloc; }
  char *buf = pts->points;
  Int byte= write(fd, buf + pos, reste);
  pos += reste; }
close(fd); return 0; }
```

```
int file_exists(char *fname) {
  struct stat info;
  if (stat(fname, &info) < 0) {
     return 0; // File doesn't exist or error    }
  return S_ISREG(info.st_mode); // Returns true if it's a regular file }
```

1.11 Un système avec mémoire virtuelle contient de la mémoire cache, de 2ns de temps d'accès, de la mémoire vive, de 10ns de temps de pénalité d'accès (i.e.10ns qui s'ajoutent aux 2ns d'essai d'accès en cache), et un disque, de 10ms de temps de pénalité d'accès. En moyenne, 95% des accès peuvent se faire en mémoire cache et, pour les accès qui ne peuvent être servis en cache, 99% d'entre eux peuvent se faire en mémoire vive. Si l'accès n'est pas en mémoire, alors le système d'exploitation doit faire un accès disque. Quel est le temps d'accès moyen à la mémoire dans cette situation?

Le temps d'accès moyen est donc de 2ns + .05 x 10ns + .05 x .01 x 10,000,000ns = 5002.5ns. Ceci montre bien l'importance d'avoir un taux de succès extrêmement élevé au niveau de la mémoire vive car chaque accès disque prend une éternité et affecte énormément la performance.

4.1. Deux processus sont connectés par un canal bidirectionnel (un tube anonyme dans chaque direction) avec une taille de 4Ko. Le processus A demande au processus B la traduction d'un texte. Différentes stratégies sont possibles pour A.

Le processus B lit 128 caractères à la fois et une fois un paragraphe complet reçu, il le traduit et le renvoie à A avant de poursuivre sa lecture de A. Pour chacune des stratégies pour A, dites si elle peut fonctionner (jamais, parfois, toujours...) et pourquoi.

a) Le processus A envoie tout le contenu du texte vers B et ensuite lit de B le texte traduit.

b) Le processus A envoie 128 caractères du texte vers B et ensuite lit la réponse de B.

c) Le processus A utilise un fil d'exécution pour envoyer le texte à B et un autre pour lire la réponse de B.

Dans le premier cas, à mesure que A envoie les données, elles sont lues par B.

Toutefois, lorsque B aura retourné 4Ko de texte traduit sans que A ne le lise, B restera bloqué sur cette écriture. Ensuite, A pourrait remplir aussi jusqu'à 4Ko dans le tube vers B avant de bloquer. Ainsi, sauf pour de très petits textes, il y aura un interblocage. Dans le second cas, si les 128 caractères ne constituent pas un paragraphe, B sera toujours en attente alors que A se mettra aussi en attente, encore en interblocage. Le troisième cas fonctionne bien.

5.1Vous voulez réaliser un certain traitement d'image en 3 étapes sous forme de pipeline. La première étape est le chargement de l'image, ensuite l'application d'un effet, et la dernière étape est la sauvegarde. Chaque item est passé au suivant par un tampon circulaire. Comment réaliser cette architecture avec des sémaphores? Décrivez quels sont les rôles de chaque fils d'exécution (producteur et consommateur).

On doit démarrer 3 fils d'exécution, un pour chaque étape. Il faut 2 tampons et 2 sémaphore par tampon, un pour attendre si le tampon est plein et l'autre pour attendre si le tampon est vide. Il faut également un signal pour indiquer qu'il faut quitter. Pour cela, on peut utiliser une valeur spéciale sentinelle, comme null tout simplement. Le premier fil d'exécution n'est qu'un producteur. Le second fil d'exécution est un consommateur et un producteur, tandis que le dernier fil n'est qu'un consommateur.

5.3Plusieurs processus veulent ajouter de l'information à la fin d'un fichier avec mmap() simultanément. Décrivez comment vous pourriez faire pour éviter qu'un processus écrase l'information d'un autre processus en train d'écrire, tout en étant le plus efficace possible?

5.4Linux supporte uniquement le verrouillage de fichier coopératif (contrairement à un verrouillage obligatoire), c'est-à-dire que si on verrouille un fichier avec flock(), il faut que l'autre processus utilise flock() lui aussi. En d'autres termes, on ne peut pas s'assurer d'avoir l'accès exclusif à un fichier en forçant un autre fichier à attendre. Quels sont les avantages et les inconvénients de cette situation?

Il s'agit d'une vue optimiste à la concurrence, dans laquelle la plupart du temps, ce sera correct. Un avantage est que l'accès est toujours possible, par exemple pour des programmes de sauvegarde. Avec le verrouillage obligatoire, on pourrait oublier de fermer un fichier en cours d'édition, ce qui bloquerait la sauvegarde. L'inconvénient est que si un fichier est en cours de modification, il se peut qu'il ne soit pas cohérent. C'est la raison pour laquelle on ne peut pas tout simplement sauvegarder directement les fichiers d'une base de données pendant qu'elle fonctionne.

5.5Réalisez deux scripts pour expérimenter avec les verrous sur un fichier. Le premier script modifie un fichier périodiquement, tandis que l'autre doit le sauvegarder périodiquement. Utiliser la commande flock pour assurer l'exclusion mutuelle entre les deux script.

On utilise flock sur un fichier spécifique, dont le chemin est connu des deux scripts. La commande de modification du fichier utilise flock avec ce fichier pour lancer le programme qui modifier le fichier principal. Le script de sauvegarde doit utiliser également flock pour exécuter la copie du fichier. Le système d'exploitation empêchera les deux opérations de se faire en même temps.

6.11 Une instruction assembleur charge une valeur de 32 bits dans un registre sur une architecture n'ayant pas de contrainte d'alignement. Combien de défauts de pages pourraient subvenir à cause de cette instruction?

Cette architecture n'ayant pas de contrainte d'alignement, il est possible que l'instruction et la valeur à charger soit à la fin d'une page et s'étende sur la suivante. Si cette situation survient, il se produit un total de 4 fautes de pages pour cette seule instruction.

7.4Un même fichier est ouvert depuis deux processus. L'un d'eux ferme le fichier et le supprime. Que se passera-t-il dans l'autre processus? À quel moment l'espace occupé sera-t-il libéré?

Le fichier existe tant qu'il est référencé par au moins un processus. Ainsi, même si le fichier n'a plus de nom dans aucun répertoire après la suppression, son contenu n'est pas libéré et il continue d'être disponible pour les processus qui l'utilisent encore. Lorsque le fichier n'est plus ouvert par aucun processus, alors l'espace est libéré. Il faut faire attention à cet aspect, car il peut manquer d'espace sur un disque à cause de gros fichiers temporaires qui restent ouverts et qui ne sont pas visibles dans l'arborescence.

7.7Considérez la structure d'inode de la question précédente. Calculez le temps moyen d'accès à chacun des blocs de données pour un fichier de taille maximale et des accès aléatoires dans le cas où il n'existe aucune cache des blocs, et que la lecture d'un bloc nécessite un délai d = 6 ms.

Les pointeurs directs référencent 4 blocs. Le premier niveau indirect référence 64 blocs. Le second niveau indirect référence 64 * 64 = 4096 blocs. Ainsi, la taille maximale est de 4164 blocs. Le temps d'accès pour les blocs directs nécessite la lecture de l'inode puis le bloc de données (2*d), le premier niveau indirect consiste en l'inode, le bloc indirect puis le bloc de données (3*d), et le second niveau indirect consiste à un accès de plus que le précédent (4*d). Pour la taille maximale et un accès aléatoire, le temps moyen d'accès est donc : (2*d*4/4164) + (3*d*64/4164) + (4*d*4096/4164), ce qui donne environ 23,9 ms.

8.3Un ordinateur peut accéder un mot en mémoire en 10ns. Lors d'une interruption, il doit sauver 32 registres d'usage général, le compteur de programme et le registre de statut. Quel est le nombre maximal d'interruptions par seconde que ce système pourrait théoriquement soutenir?

Avec 34 registres à sauver puis à recharger à chaque interruption, il lui faut au moins 34 * 2 * 10ns = 680ns. Il est donc limité à un maximum de 1s / 680ns / Int. = 1.47MHz.

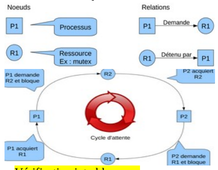
8.11 Le professeur louange la performance de l'algorithme de l'ascenseur utilisé dans le système d'exploitation Linux. Un étudiant fait un test avec un programme qui lit aléatoirement 10000 blocs bien répartis sur le disque. Il ne note aucune différence par rapport au même test fait avec un ordonnanceur de disque trivial, qui sert les requêtes dans leur ordre d'arrivée. Comment expliqueriez-vous cela?

Si le programme de test de l'étudiant utilise des lectures bloquantes, il n'est pas possible pour le système d'exploitation de voir plusieurs requêtes à la fois et d'en optimiser l'ordonnancement. Un bon test devrait soit utiliser plusieurs processus concurrents faisant des lectures ou un processus avec des lectures asynchrones.

Blocage dans une interface graphique:
La mise à jour de l'interface et la gestion des événement doit se faire continuellement.
Il ne faut JAMAIS bloquer le fil principal d'une interface graphique.
Sinon, dégrade l'interactivité et l'utilisateur peut penser que l'application est plantée.
Respecter l'échéance est primordiale pour la fluidité
–Sinon, le jeu est saccadé et cause frustration –Exemple de « soft realtime »

Utilisation d'un fil d'exécution:
-Bloquer dans un fil d'exécution n'empêche pas le fil principal de s'exécuter
-Technique classique qui fonctionne bien avec les fichiers ordinaires
-Nécessite de communiquer avec le fil principal (producteur/consommateur)
-L'idéal est de réutiliser les fils d'exécution pour réduire le surcoût
-Ne pas démarrer trop de fils d'exécution, car les fils vont concurrencer pour l'accès au processeur
-Suffisant pour éviter qu'une interface graphique ne bloque pas, mais les transferts se font séquentiellement

Interblocage: Conditions nécessaires:
1) Condition d'exclusion mutuelle
Une ressource est libre ou détenue par un processus.
2) Acquisition partielle suivie d'attente
- Un processus détenant des ressources peut en demander d'autres et bloquer parce qu'elles sont déjà occupées.
3) Les ressources détenues ne peuvent être volées
- Seul le processus qui les détient peut les libérer.
4) Il doit y avoir une attente circulaire
- Foo attend Bar qui attend Foo



Vérification interblocages:
Test: utiliser un test de charge en concurrence, laisser rouler longtemps, espérer que si une possibilité d'interblocage existe, elle se manifeste.
Si un interblocage survient, alors il existe une possibilité d'interblocage
Si aucun interblocage ne survient, alors il se peut qu'il y ait une possibilité d'interblocage, mais que le test n'ait pas exécuté une séquence menant à une interblocage.
Un test ne permet pas de prouver l'inexistence d'un interblocage, mais permet d'augmenter la confiance qu'il n'y en ait pas.
2:
Vérification formelle: explorer toutes les possibilités d'exécution et déterminer si un chemin conduit à un interblocage.
Si un chemin mène à l'interblocage, alors il est certain qu'un interblocage peut survenir (même s'il est peu probable).
Si aucun chemin ne mène à un interblocage, alors on est certain que le logiciel est valide.
La vérification formelle est une preuve.

Modéliser le programme sous forme d'automate
Explorer toutes les possibilités d'exécution
Vérifier si une exécution mène à un interblocage

L'explosion combinatoire rend difficile la vérification et applicable que pour de petits logiciels ou quelques routines (ex: implémentation d'une structure de donnée)
La vérification formelle est faite avec des algorithmes sophistiqués pour atténuer l'explosion combinatoire autant que possible.
Exemple de logiciel de vérification formelle: UPPAAL, Spin, Promela, JavaPF

Interblocage sous Linux:
CONFIG_PROVE_LOCKING
CONFIG_DEBUG_SPINLOCK_SLEEP
CONFIG_RCU_TORTURE_TEST

Read-Copy-Update:
Basé sur les instructions atomique et barrières mémoires
Cycle de vie :
–Publication d'une structure (i.e. météo)
–Copie de la structure pour modification
–Publier la nouvelle version
–Quand on est certain que l'ancienne version n'est plus utilisée, on la libère
Utilisé dans le noyau Linux pour éviter les verrous (Exemple : manipulation d'une liste partagée)

Source de blocages:
Faute de page majeur
-Délai imprévisible : dépend du périphérique!
-Se produit quand on accède à une adresse
Réception d'un paquet réseau
Le serveur ou la connexion est p-être lente
Par défaut read()/write() pourraient bloquer longtemps sur un socket
-Lecture d'un fichier sur disque avec read()

POSIX aiocb:
Basé sur des fils d'exécution sous Linux
- Pourrait changer dans le futur
Ajout des requêtes dans une file d'attente
- Structure aiocb(Async IO Control Block)
-aio_read(): démarrer une lecture en ar-plan
-aio_write(): démarrer une écrit. en arr-plan
-aio_error(): déterminer si une requête est terminée
-aio_return(): obtenir le statut d'une requête
-aio_suspend(): attendre qu'il y ait au moins une requête qui se termine

Multiplexage Linux aio:
API du noyau Linux
Permet de remplacer POSIX aio, sans utiliser de fil d'exécution
Le noyau gère la file d'attente
Disponible depuis Linux 2.5

Signal:
Définir l'attribut O_ASYNC sur le fichier avec fcntl()
Le signal SIGIO est émis quand le fichier est prêt pour la lecture/écriture
Exemple : compteurs de performance
fcntl(fd, F_SETOWN_EX, &fown)
flags = fcntl(ev->fd, F_GETFL);
fcntl(ev->fd, F_SETFL, flags | FASYNC | O_ASYNC) //Configuration asynchrone

Exemple : ressources limitées:
Soit un système avec 2 processus.
Le système permet au maximum 4 processus.
Chaque processus doit créer 2 enfants pour se terminer correctement. Si un fork() échoue, alors le processus parent réessaye.
Déterminer une s'équence d'événements (création, terminaison) qui produit un succès,et un autre qui produit une boucle infinie.
Algorithme du banquier $$$

Stratégies:
Ignorer le problème
– … et il surviendra pendant une démo
– Inacceptable pour des systèmes critiques : centrale nucléaire, navette spatiale, instrument médical, etc.
Détecter les interblocages et réagir
– Solution de dernier recours
Rendre les interblocages impossible
– Solution idéale

Prévention des interblocages:
Empêcher l'attente indéfinie
– Réserver une ressource à la fois, si une ressource n'est pas disponible, alors libérer les ressources et réessayer plus tard
– Sujet à la famine et à l'interblocage actif « livelock »
– Un processus peut se réessayer en boucle sans jamais réussir
– Solution: se mettre dans une file d'attente
Évite de créer un cycle
– Toujours demander les ressources dans le même ordre empêche un cycle de se former

Stratégies:
-Effectuer le travail qui risque de bloquer dans un fil d'exécution séparé
-Utilisation d'entrées-sorties non-bloquant
-Surveillance de plusieurs descripteurs de fichiers simultanément avec délai borné
-En utilisant des signaux

Option O_NONBLOCK:
-Modification d'un descripteur avec fcntl()
int flags = fcntl(fd, F_GETFL, 0);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
-Rend read() non-bloquant
-Non-prêt : read() ret == -1 && errno == EAGAIN
- On doit réessayer plus tard
Mais on peut faire autre chose en attendant!
-En pratique, fonctionne pour socket et pipe
- N'a pas d'effet avec fichiers réguliers

Multiplexage:
-Gérer la lecture et l'écriture dans plusieurs fichiers dans un seul fil d'exécution
-Appels systèmes : select(), poll() et epoll()
-Aucune attente ou attente bornée (timeout)
-Tous les descripteurs prêt sont retournés et les utiliser ne bloquera pas
Exemple : traitement des entrées du clavier et de la souris
Exemple : téléchargement simultané de plusieurs fichiers, peut se faire dans un seul fil d'exécution(Réduction potentielle des changements de contexte coûteux)

Python aiohttp:
Basé sur epoll comme
-select(), mais plus performant
-epoll_create() : créer un groupe d'attente
-epoll_ctl() : ajoute, modifie ou enlève des descripteurs à surveiller
-epoll_wait() : attend qu'un descripteur soit prêt, avec un délai maximum optionnel (timeout)
Démarrage explicite de la boucle événementielle → asyncio.run(main())
JS:struct async/await utilise epoll_wait()

Détection:
Créer le graphe de ressources.
Détecter un cycle dans le graphe.
– Parcourir le graphe et construire une liste des noeuds visités. Si en parcourant le graphe, on rencontre un noeud déjà dans la liste, alors il existe un cycle.
Le cycle indique quelles sont les ressources impliquées dans l'interblocage.
2:
Utilisation des interruptions non masquables périodiques
Survient même après avoir désactivé les interruptions
Dans le gestionnaire, on vérifie qu'il s'est produit un changement d'état (comme un événement d'ordonnancement)
Si rien ne s'est passé pendant plusieurs secondes, alors il est probable que le système soit en interblocage
Action : redémarrage forcé
La panne n'est pas évitée, mais réduit sa durée

Briser un interblocage:
Retour en arrière (rollback)
– Par ex avec pthread_mutex_trylock()
– Libérer tous les verrous acquis et réessayer depuis le début
Terminer un processus dans le cycle
– Construire le graph de ressources, parcourir le graph et détecter un cycle
– N'est pas utilisé en pratique

Conclusion: Conditions d'interblocage
– Règle simple : réserver les ressources toujours dans le même ordre!
Différence entre test et vérification
– Test : facile à faire, mais ne prouve pas l'exactitude
– Vérifi formelle : difficile à faire, mais prouve l'exactitude
Linux a plusieurs lignes de défense pour trouver les bogues reliés aux interblocages
Techniques sans verrous sont complexes, mais sont généralement performantes et sans interblocages

Question 1 Quelle fonction parmi les suivantes doit-on typiquement placer dans une boucle while pour un fonctionnement adéquat? pthread_cond_wait()

Question 4 Quel est un avantage d'un verrou actif (spinlock)?✅ Latence faible pour obtenir le verrou lorsqu'il devient disponible

Question 2 Combien de fois une donnée est copiée lorsqu'elle est transférée par un tube?✅ Deux copies, une écriture dans le tampon, et une lecture depuis le tampon

Question 3 Dans un serveur, on envoie une image de 1 mégaoctet sur un socket TCP. Dans le client, on reçoit les données avec un appel read() sur le socket. Quelle situation s'applique? ✅ read peut retourner avec des données partielles, auquel cas il faut faire d'autres appels à read pour compléter le transfert

Question 5 Dans quelle situation un sémaphore est typiquement utilisé?✅ architecture producteur-consommateur

Question 6 Dans l'extrait de code suivant, on souhaite synchroniser les itérations de la boucle entre les fils d'exécution, de manière à garantir que tous les fils aient terminé une itération avant de passer à la suivante. Quelle primitive de synchronisation faut-il utiliser? for (int i = 0; i < 10; i++) { work(i); } ✅ Barriere

Question 7 Quelle situation résume le mieux l'utilité d'un pipe? ✅ transfert unidirectionnel entre 2 processus qui accèdent au file descriptor

Question 9 Quelle signal bloque le processus temporairement? ✅ SIGSTOP

Question 10 Que se produit-il quand on tente d'obtenir un mutex déjà occupé? ✅ le fil d'exécution bloque

Question 12 Pour un sémaphore binaire, quel est l'équivalent de libérer un verrou? ✅ sem_post

Question 13 Quelle fonction permet d'initialiser un mutex? ✅ pthread_mutex_init

Question 15 Quelle méthode est efficace pour vérifier rapidement que les verrous sont valides dans un logiciel d'envergure? ✅ Recompiler avec ThreadSanitizer, puis executer une suite de testes pour reveler les problemes potentiel

Question 16 Dans le laboratoire météo, quel est l'avantage d'utiliser un verrou asymétrique pour une variable souvent lue, rarement modifiée, par rapport à un verrou symétrique? ✅ plus rapide, car permet aux lectures de lire en même temps

Question 17 Un socket est un mécanisme de communication absolument nécessaire dans quelle circonstance? ✅ Communiquer entre deux processus sur deux ordinateurs distincts

Question 18 Qu'allons-nous observer avec strace si un programme est en attente pour un pthread_mutex_t? ✅ un seul appel à futex()

Question 19 Quelle est la taille d'un tube sous Linux, tel que nous avons vu en classe? ✅ 65536

Question 20 À quelle valeur doit-on initialiser un sémaphore binaire de manière à modifier le nombre d'items libres dans un tampon circulaire de taille N? On suppose que le tampon est vide au départ. ✅ N-1