
Examen de Système – Corrigé

Licence 3 Informatique

ENS Cachan, mardi 13 janvier 2009

Questions de cours : les threads

Voir le cours.

Exercice 1 : Points de rupture tas et pile

1. Indiquer une méthode pour trouver une adresse valide quelque part dans la pile (peu importe laquelle, il suffit d'en connaître une pour permettre de débiter le parcours). Donner le code de la fonction correspondante `adresse_valide_pile` qui calcule et renvoie une telle valeur.

On renvoie par exemple l'adresse d'une variable locale :

```
char *adresse_valide_pile(void) {
    char i=0;
    return &i;
}
```

2. Même question concernant le tas : indiquer une méthode pour y trouver une adresse valide et donner le code de la fonction `adresse_valide_tas`.

On renvoie l'adresse d'une variable globale, d'une fonction, ou simplement une adresse obtenue par malloc :

```
char *adresse_valide_tas(void) {
    return (char *) malloc(1);
}
```

3. Il s'agit maintenant de mettre en place le traitant de signal pour SIGSEGV, ainsi que le code qui permettra de revenir dans la fonction main une fois le signal délivré. Compléter le code du programme rupture de manière à récupérer dans la variable `rupture_pile`, avant l'appel à `printf`, l'adresse du point de rupture de la pile. On n'oubliera pas de donner le code de la fonction traitant de signal. On peut bien sûr ajouter des variables globales si nécessaire.

On fait une boucle où on décrémente le pointeur sur la pile et on tente d'y accéder jusqu'à déclencher l'erreur d'adressage. Le seul aspect important, c'est qu'il faut s'assurer que, lorsqu'on a fini de rattraper l'erreur, on pourra réexécuter l'instruction d'écriture fautive, sinon le signal sera de nouveau envoyé. Enfin, il faut sortir de la boucle. Cela pourrait donner quelque chose comme cela :

```
char *rupture_pile = NULL,
      *rupture = NULL;

int etape = 0;

void hand(int sig){
    rupture_pile = rupture;
    rupture = adresse_valide_tas(); // reparation
    etape++;
}

int main() {
    action.sa_handler = hand ;
    sigaction(SIGSEGV, &action, NULL) ;
    rupture_pile = adresse_valide_pile();
    while(etape==0) *rupture++ = 'a';
    printf("Zone interdite : entre %p et %p\n", rupture_tas, rupture_pile);
}
```

```
}
```

*Malheureusement, si vous testez le code, cela ne marche pas! C'était néanmoins la réponse attendue... Pour comprendre pourquoi cela ne marche pas, il faut considérer que le vrai code exécuté est de l'assembleur, et l'instruction fautive n'est pas une écriture à ***rupture**, mais sans doute une écriture à ***reg** où **reg** est un registre qui contient une copie locale de **rupture**... Bref, il n'est pas possible d'empêcher l'erreur de se renouveler en modifiant **rupture** comme suggéré.*

4. Compléter le programme de manière à récupérer également l'adresse du point de rupture du tas dans la variable `rupture_tas`, avant l'appel à `printf`. Est-il nécessaire d'utiliser un second traitant de signal? Expliquer.

*Il n'est pas nécessaire d'utiliser un autre handler de signal, il suffit d'utiliser la variable **etape** pour savoir où on en est du code. Cela donne (toujours du code qui ne marche pas, mais qui était l'idée attendue) :*

```
char *rupture_tas = NULL,
      *rupture_pile = NULL,
      *rupture = NULL;

void hand(int sig){
    if (etape==0) rupture_pile = rupture;
    else rupture_tas=rupture;
    rupture =adresse_valide_tas();
    etape++;
}

int main() {
    action.sa_handler = hand ;
    sigaction(SIGSEGV, &action, NULL) ;
    rupture_pile = adresse_valide_pile();
    while(etape==0) *rupture++ = 'a';
    while(etape==1) *rupture++ = 'a';
    printf("Zone interdite : entre %p et %p\n", rupture_tas, rupture_pile);
}
```

Bonus *Une solution qui marche à peu près (pas sur MAC, mais on ne sait pas pourquoi...). Merci à monsieur Raymond pour son aide!*

```
struct sigaction action;
jmp_buf buf;

char *rupture_tas = NULL,
     *rupture_pile = NULL;

void hand(int sig)
{
    printf("Signal %d caught\n", sig);
    siglongjmp(buf, 1);
}

int main()
{
    char x;

    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;
    action.sa_handler = hand ;

    sigaction(SIGSEGV, &action, NULL) ;

    if(sigsetjmp(buf, 1) == 0) {
        rupture_tas = malloc(1);
        while(1)
            x = *(rupture_tas++);
    }

    if(sigsetjmp(buf, 1) == 0) {
        rupture_pile = &x;
        while(1)
            x = *(rupture_pile--);
    }
}
```

```

    printf("Zone interdite : entre %p et %p\n", rupture_tas, rupture_pile);
}

```

Une autre solution à la réparation, qui ne marche que pour le point de rupture du tas, consiste à utiliser l'instruction `brk(rupture+1)` à ligne réparation du code vu avant, ce qui a pour effet de changer le point de rupture du tas. Mais pour la pile, je n'ai pas d'autre solution que les `sigsetjmp/siglongjmp`.

Exercice 2 : Un select ou des threads

1. Préciser le code de la boucle principale d'un tel serveur séquentiel (partie service). Pour quelles valeurs N et N' de n le serveur passe-t-il respectivement en régime saturé et en situation critique ?

Le code de la partie service du serveur séquentiel pourrait être le suivant :

```

while(1){
    for(i=0,i=n-1,i++){
        read(socket[service[i],requete,TailleRequete);
        reponse = traite(requete,i);
        write(socket[service[i],reponse,TailleReponse);
    }
}

```

Dans le cas où tout client a envoyé sa requête en avance sur la demande du serveur, le temps t entre le write et le read suivant sur la même socket d'écoute est donc $(n-1)q$. Pour que l'hypothèse qui justifie ce calcul soit correcte, il faut avoir $t \geq c$. En prenant $n = 1 + \frac{c}{q} \approx \frac{c}{q}$, on est dans cette hypothèse, donc à la limite du passage en régime saturé. Au-delà, chaque client attend un temps t supérieur à c , et instantanément il y a $\frac{t-c}{q} \approx n - \frac{c}{q}$ clients en attente. Conclusion : $N = \frac{c}{q}$, et $N' = \frac{2c}{q}$.

2. Rappeler le fonctionnement d'un serveur multithreadé. Pourquoi, même sur une architecture monoprocesseur peut-on obtenir un gain de temps par rapport au serveur séquentiel ? Pour cette question, on précise que

le temps c est un temps observé moyen, mais le temps d'attente exact peut fluctuer à plus ou moins quelques q près.

Dans un serveur multithreadé, chaque thread exécute en boucle l'une des séquences read-traite-write. Du fait des fluctuations, l'ordre d'arrivée des requêtes n'est pas un cycle constant. En régime non saturé, on peut obtenir un gain de temps par rapport au serveur séquentiel puisque une requête qui tarde à venir ne bloquera pas les suivantes qui seraient déjà arrivées.

3. Une commutation de contexte est estimée prendre un temps $S \ll c$, et au moins de l'ordre de q ($q \ll S$ ou $q \approx S$). Donner les valeurs N_t et N'_t de n pour lesquelles le serveur passe respectivement en régime saturé et en situation critique.

Le temps entre deux reads est minoré par c , et par le temps de calcul du traitement de l'ensemble des threads, soit $n(q + S)$ (chaque thread s'exécute dans un quantum de temps car $q \ll S$ ou $q \approx S$ et occasionne une seule commutation de contexte pour le read). A partir de là, le calcul est le même que pour le serveur séquentiel, avec un temps de calcul égal à $q + S$. Conclusion : $N_t = \frac{c}{q+S}$, et $N'_t = \frac{2c}{q+S}$.

4. Un appel système `select` permet de scruter plusieurs descripteurs de fichiers simultanément : dans un ensemble de descripteurs donné, `select` signale ceux pour lesquels des données sont actuellement disponibles. Expliquer comment construire un serveur monothreadé utilisant cet appel système.

La boucle while ne contient plus un for, mais un select sur toutes les sockets. Le select est suivi d'un read, d'un traitement et d'un write sur les sockets pour lesquelles les données sont disponibles.

5. Un appel à `sélect` est estimé à un temps du même ordre de grandeur qu'un changement de contexte, donc S . Justifier les valeurs $N_s = \frac{c}{q+S}$ et $N'_s = \frac{2(c-S)}{q}$ de n pour lesquelles le serveur passe respectivement en régime saturé et en situation critique.

En régime non saturé, le select signale un clients comme disponible, le traite, et on boucle, chaque itération prend donc un temps $q + S$. On entre donc en régime saturé lorsque $n(q + S) \geq c$, i.e $N_s = \frac{c}{q+S}$. Si k clients sont en attente, une itération de boucle coûte donc $S + kq$, si

la moitié des clients sont en attente, c'est que ce temps est égal à c , et $k = \frac{N'_s}{2}$, donc $N'_s = 2\frac{c-S}{q}$.

6. Selon ce modèle, quel est le serveur le plus efficace ? Dans quel cas le gain $\frac{N'_s}{N'_t}$ est-il significatif ?

Le nombre de clients pour lequel on entre en régime saturé est le même, par contre, le serveur avec select supporte mieux la charge puisque

$$\frac{N'_s}{N'_t} = \frac{2(c-S)(q+S)}{q \cdot 2c} = 1 + \frac{S}{q} \left(1 - \frac{q+S}{c}\right) \approx 1 + \frac{S}{q} \geq 1$$

Le gain devient vraiment significatif si $q \ll S$.

Exercice 3 : L'échec de Peterson

L'algorithme de Peterson est une implémentation d'un verrou partagé entre deux threads qui n'utilise aucune primitive de synchronisation ou instruction atomique autre que des opérations de lecture ou écriture en mémoire partagée. C'est donc en théorie un algorithme fiable pour réaliser l'exclusion mutuelle au plus bas niveau d'un système d'exploitation qui s'exécute sur un multiprocesseur à deux coeurs.

1. Rappeler pourquoi un système d'exploitation peut avoir besoin d'implémenter de l'exclusion mutuelle pour son propre compte. Sous quels choix de conception le système n'entre-t-il jamais en conflit avec lui-même ? Ce choix-là est-il envisageable sur un multicoeur à deux processeurs ?

Un système d'exploitation distribué peut traiter des appels systèmes simultanément, entrant en mode noyau sur plusieurs processeurs, et accédant concurrentiellement aux ressources systèmes (tables de fichiers, de pages, etc). Pour préserver la cohérence du système, l'accès à ces ressources au sein des traitant d'appel système doit être conçu dans un contexte de concurrence, soit en utilisant de l'exclusion mutuelle, soit des structures de données non bloquantes. Dans une conception centralisée, seul un processeur maître est autorisé à gérer les appels systèmes, ce qui constitue un goulot d'étranglement si le nombre de processeurs est élevé, mais reste acceptable pour un double coeur.

2. Rappeler l'implémentation de `lock` et `unlock` avec l'algorithme de Peterson (on utilisera la variable de pile `me` à valeur dans $\{0, 1\}$ pour identifier le thread appellant). Redémontrer pourquoi il est sans interblocage (deadlock).

Cf. cours :

```
bool interested[2];
int polite = 0;

void lock(){
    interested[me] = true;
    polite = me;
    while(interested[1-me] && polite == me) ;
}

void unlock(){interested[me] = false;}
```

3. Redémontrer pourquoi il implémente l'exclusion mutuelle. On raisonnera par l'absurde en listant les contraintes temporelles entre les différents événements qui amèneraient à une double acquisition du lock.

Supposons par l'absurde que 0 et 1 sont en section critique simultanément, et par symétrie on suppose que 0 s'est déclaré poli en dernier. Dans ce cas, lorsqu'il a passé le test, 0 a vu que 1 n'était pas intéressé. Comme 1 n'était pas intéressé, mais qu'il est intéressé à la fin (puisque en section critique), cela veut dire que 1 n'avait exécuté aucune instruction du lock, ce qui contredit que 0 était déclaré poli en dernier.

On implémente l'algorithme de Peterson, et on observe dans quelques cas des acquisitions simultanées de verrou. Ce phénomène s'explique par les optimisations de compilateurs et du processeur, qui peut réordonner ou paralléliser des instructions tant que l'effet global, dans une interprétation monothreadée, reste le même.

4. Expliquer pourquoi Peterson échoue. On donnera une exécution optimisée des instructions qui fait échouer l'exclusion mutuelle, et on précisera quelles contraintes temporelles utilisées dans la preuve de la question 3 ne peuvent plus être utilisées.

Il est possible, en préservant la sémantique séquentielle, de repousser la première instruction du lock (`interested[me]=true`) arbitrairement tard, car son effet n'est pas pris en compte par ce thread, mais par l'autre thread. Il est donc possible que 0 entre dans lock, se déclare poli, passe le test correctement pour sortir du while, soit préempté avant d'avoir fait le `interested[me]=true`, alors 1 entre en lock, ne voit pas l'autre comme intéressé donc retourne comme si de rien n'était et rentre en section critique, puis est préempté, et 0 termine son lock et retourne, rentrant lui aussi en section critique. L'hypothèse qui est remise en cause dans le raisonnement de la question précédente est que n'étant pas déclaré intéressé, 1 n'a exécuté aucune instruction du lock.

En pratique, certaines instructions forcent le processeur à s'acquitter des instructions lues n'ayant pas encore pris effet avant de poursuivre avec de nouvelles instructions, un peu comme « flush » force à vider un buffer d'écriture (on parle de barrière mémoire , memory fence en anglais).

5. Corriger l'algorithme de Peterson avec une telle instruction « flush ».

On rajoute un fence avant le `polite=me`. Le raisonnement de la question 3 reste alors justifié. Remarque : on peut mettre le fence avant la boucle while et cela marche aussi, mais le raisonnement qui justifie cela est légèrement différent de celui présenté. Exercice : voir pourquoi Peterson marche toujours si la déclaration d'intérêt et de politesse s'exécutent dans un ordre arbitraire.

Exercice 4 : Barrières de synchronisation

Dans un contexte multithread, une barrière de synchronisation est une instruction appelée par chaque thread pour attendre les autres (d'un groupe fixé à l'avance). Elle peut servir par exemple à synchroniser des threads se découpant le calcul et l'affichage d'une sortie écran réactualisée périodiquement.

1. Ecrire la spécification de la fonction `barrier` en explicitant le langage des exécutions valides sur l'alphabet d'événements

$$\Sigma = \text{IDThread} \times \{\text{barrier_call}, \text{barrier_return}\}.$$

Le langage des exécutions valides est le suivant :

- « l'exécution est séquentialisable » :
à chaque événement retour de fonction, on peut associer injectivement un événement d'appel du même thread qui le précède, et on n'a pas deux appels du même thread sans un retour de ce même thread entre les deux. Formellement, une exécution e est séquentialisable si pour tout thread t , sa projection $e \downarrow t$ sur le thread t appartient au langage $(\text{call.return})^* . (\text{call} + \epsilon)$. Cette propriété est générale, et ne se restreint pas aux barrières de synchronisation. La propriété intéressante est la suivante.
- « les threads s'attendent pour le retour de fonction » :
pour tous threads t, t' , le i ème retour de fonction du thread t est précédé d'exactly i appels de fonction du thread t' . Formellement, une exécution e vérifie cette condition si pour tout préfixe de e de la forme $e'(t, \text{return})$, le nombre d'événements (t', call) dans e' est le même pour tout t' .

2. Proposer l'implémentation d'une barrière à usage unique (on supposera si nécessaire que les threads possèdent un identifiant de 1 à n accessible par la variable de pile me). On utilisera uniquement des sémaphores. Justifier l'implémentation.

Une solution à n sémaphore consiste à donner un crédit de 1 à chacun des autres threads, puis attendre d'avoir soit-même n crédits pour se débloquer :

```
semaphore wait[n]; // supposées initialisées à 0

void barrier(){
    for(i=0, i<n, i++) V(wait[i]);
    for(i=0, i<n, i++) P(wait[me])}
}
```

On vérifie bien la spécification : il n'est pas possible au thread i de sortir de la fonction tant que le dernier thread du groupe à appeler `barrier` n'a pas fait le n ème $V(\text{wait}[i])$ attendu.

3. Proposer l'implémentation d'une barrière à usages multiples utilisant un nombre fixe de sémaphores. Justifier l'implémentation.

Le problème est plus compliqué pour plusieurs sessions : en effet dans l'état actuel, si le premier thread libéré rappelle tout de suite `barrier`, il

a des chances de pouvoir passer tout de suite. Par exemple, pour deux threads, une fois que chacun a accompli la phase V, wait vaut [2,2]. Si le premier thread sort sans que l'autre ait fait quoi que ce soit, wait vaut [0,2], puis si il réappelle barrier et finit sa phase V, wait vaut [1,3], et il peut à nouveau sortir, laissant wait dans l'état [0,2]. Non seulement il sort trop tôt, mais l'autre thread ne peut plus sortir!

Une solution simple (mais on peut faire plus économe en nombre de sémaphores) consiste à utiliser 2n sémaphores, et utiliser un jeu de n sémaphores pour les sessions paires, et les autres pour les sessions impaires :

```
semaphore wait[n][2]; // supposées initialisées à 0

int session[n]; // supposées initialisées à 0

void barrier(){
    for(i=0,i<n,i++ V(wait[i][session[me]]);
    for(i=0,i<n,i++) P(wait[me][session[me]])}
    session[me]=1-session[me];
}
```

On voudrait maintenant considérer une barrière pour laquelle le groupe de threads à synchroniser n'est pas défini à l'avance. On considère ainsi une opération supplémentaire appelée **register** qui permet à un thread d'annoncer son intention d'utiliser la barrière pour se synchroniser avec les autres threads qui se seront enregistrés au moment voulu.

4. Discuter la spécification et l'implémentation des deux fonctions **register** et **barrier** dans cette perspective.

Une spécification possible est la suivante : un groupe de thread est créé lors d'un register si il n'y aucun groupe ouvert disponible, et le thread appelant automatiquement inscrit à ce groupe. Si il y a un groupe ouvert, le thread appelant est ajouté à ce groupe. Lors d'un appel premier à barrier, par un thread de ce groupe, le groupe est fermé, et barrier se comporte comme auparavant avec les threads concernés. Une ébauche d'implémentation serait :

```
struct group {
    thread_list_t participants;
```

```

    lock_t group_lock
    bool registration_closed;
}

struct group *newgroup(){
    struct group *tmp = malloc(sizeof(struct group));
    init_lock(tmp->group_lock);
    init_list(tmp->participants);
    tmp->registration_closed = false;
}

struct group *current_group = newgroup();

struct group *register(){
    struct group *current;
    while (1) do {
        current = current_group;
        lock(current->group_lock);
        if !(current->registration_closed) break;
        unlock(current->group_lock);
    }
    add(me,current->participants);
    unlock(current->group_lock);
    return current;
}

void barrier(struct group *group){
    lock(group->group_lock);
    if (group==current_group)
        // registration closed
        group->registration_closed = true;
    current_group = newgroup();
    unlock(group->group_lock);
    barrier_wait(group);
}

```

Exercice 5 : Le problème « ABA »

On considère une pile concurrente « non bloquante », n'utilisant pas de verrous, mais seulement l'instruction CAS (compare and swap). Cette instruction compare ses deux premiers arguments et remplace la valeur du premier par celle du troisième en cas d'égalité, et renvoie le résultat du test, le test et l'affectation éventuelle prenant effet en même temps de façon atomique. Le code de la pile concurrente est décrit ci-dessous :

```
cell *top_ptr;

cell *pop() {
    while(1) {
        data *ret_ptr = top_ptr;
        if (!ret_ptr) return NULL;
        cell *next_ptr = ret_ptr->next;
        // If the top node is still ret,
        // then assume no one has changed the stack.
        if (CAS(top_ptr, ret_ptr, next_ptr)) {
            return ret_ptr;
        }
        // The stack has changed, start over.
    }
}

void push(cell *cell_ptr) {
    while(1) {
        cell *next_ptr = top_ptr;
        cell_ptr->next = next_ptr;
        // If the top node is still next,
        // then assume no one has changed the stack
        if (CAS(top_ptr, next_ptr, cell_ptr)) {
            return;
        }
        // The stack has changed, start over.
    }
}
```

Contrairement à un autre algorithme décrit plus bas, ce code souffre d'un bug connu sous le nom de « ABA problem », caractérisé par le scénario erroné suivant :

- 1) le thread 1 lit une valeur A en mémoire partagée ;
 - 2) il est préempté, et le thread 2 est ordonnancé ;
 - 3) 2 modifie la valeur de la cellule mémoire partagée en B, puis de nouveau en A avant d'être préempté à son tour.
 - 4) 1 s'exécute, voit la valeur A et continue.
1. Expliquer pourquoi le code ci-dessus souffre du problème ABA. Est-ce un bug seulement fonctionnel (la pile ne se comporte pas comme une pile) ou un bug de sûreté (il y a un risque d'erreur de déréférencement) ?

C'est un bug fonctionnel. Supposons que l'on a l'exécution suivante :

- *La pile contient initialement $12 \rightarrow 3 \rightarrow \text{null}$.*
- *Le thread 1 fait un pop, mais est préempté juste avant le CAS.*
- *Le thread 2 fait deux pop, puis un push de la valeur 2. La pile a alors le contenu $2 \rightarrow \text{null}$.*
- *le thread 1 reprend la main. Supposons que, par malheur, l'adresse de la cellule poussée par le thread 2 soit la même que celle qui contenait 3, alors le CAS du thread 1 réussit, et la valeur 12 est sortie, bien qu'elle l'ait déjà été par l'autre thread.*

Tout dépend bien sûr du code client. Pour le bug fonctionnel précédent, c'est le client que push une cellule qu'il a obtenue par un pop, ce qui semble la moindre des choses à autoriser. Mais si on n'interdit rien, on a aussi des bugs de sûreté, par exemple on peut désallouer une cellule renvoyée par un pop et du coup générer une violation mémoire de la part d'un thread qui serait encore en train de popper cette cellule, ou bien si on utilise des adresses de cellules dans la pile comme paramètre du push, on peut créer une liste circulaire, qui pourra créer une erreur de déréférencement si on la soumet à suffisamment de pop pour faire le tour de la liste...

Pour contrer ce problème, certains processeurs (DEC Alpha, Power PC, MIPS,...) disposent d'instructions LL/SC (« load linked, store conditional ») qui se comportent de la façon suivante (vues comme des fonctions C) :

- LL(addr) renvoie le mot (ou le double, l'octet,...) à l'adresse addr.

- `SC(addr, val)` écrit la valeur `val` à cette adresse, à condition que personne n'ait modifié ou LLé la valeur à l'adresse `addr` depuis le dernier appel du même thread à LL.

2. Proposer une version corrigée de la pile précédente utilisant LL/SC.

On remplace la sauvegarde du `top_ptr` dans une variable locale par un LL, et le CAS par un SC.

On considère un autre exemple de pile concurrente (dû à Treiber) :

```
void push(value v) {
    cell *t, *x;
    x = alloc();
    x->data = v;
    do {
        t = top_ptr;
        x->next = t;
    } while (!CAS(&top_ptr, t, x));
}

value pop() {
    cell *t, *x;
    do {
        t = top_ptr;
        if (t == NULL) return EMPTY;
        x = t->next;
    }
    while (!CAS(&top_ptr, t, x));
    return t->data;
}
```

3. Pourquoi la pile de Treiber ne souffre-t-elle pas du problème ABA ?

Chaque cellule ajoutée est allouée, son adresse ne peut pas correspondre à une ancienne valeur du `top_ptr` encore sauvée dans une variable locale par un thread. De même si on n'a pas une liste circulaire, en popant, on ne retrouvera pas une valeur de `top_ptr` sauvée dans une variable locale

4. On pourrait être tenté de désallouer `t` à la fin de `pop`. Pourquoi cela introduirait-il un bug? Serait-ce un bug fonctionnel ou de sûreté?

Le bug de la question 1 redevient possible si le `new` réalloue l'adresse d'une cellule désallouée par un `pop`. On a donc un bug de fonctionnement. On a aussi un bug de sûreté : si deux threads essaient de popper, qu'ils ont la même copie du pointeur de tête, et que le premier thread passe le CAS et désalloue avant que le deuxième n'ait fait quoi que ce soit, le deuxième risque de faire une violation mémoire lorsqu'il exécute `x = t->next`.