UPMC
SORBONNE UNIVERSITÉS

UFR 919

19 janvier 2016

Licence d'Informatique 2015 -2016

L3 – 3I010

Module « Principes des Systèmes d'Exploitation »

Travaux sur Machines

Séances 1 à 11

TME 1 : PRISE EN MAIN DE L'ENVIRONNEMENT

L'objectif de ce TME est de vous familiariser avec la manipulation de programmes C sous Unix.

Fonctions utiles au TME :

La commande shell :

`man` (indispensable sous unix ! N'hésitez pas à taper `man man`… et utilisez l'option `-a`),

La fonction C :

`atoi`

1 . COMPILATION INDEPENDANTE – EDITION DE LIENS - MAKEFILE

On souhaite construire un exécutable à partir des 3 fichiers `liste.h`, `liste.c` et `testeliste.c` accessibles dans le répertoire

`/Infos/lmd/2004/licence/ue/li324-2005fev/TME1`

1.1

Recopiez ces fichiers dans votre répertoire local.

On veut que l'exécutable testeliste soit construit au moyen d'un `Makefile`.

1.2

Donnez les dépendances entre les différents fichiers.

L'option `-c` à la compilation permet de créer un fichier objet d'extension `.o` et de nom identique à celui du fichier source.

1.3

Quels sont les fichiers objet nécessaires à la création de `testeliste` ? Donnez les commandes permettant de créer ces fichiers.

L'option `-g` à la compilation est nécessaire si l'on veut par la suite utiliser un debugger symbolique sur le programme. On veut de plus que tous les programmes soient compilés avec l'option `-Wall`.

1.4

Donnez un `Makefile` permettant de construire l'exécutable `testeliste`.

2. DE BUGGER UN PROGRAMME : UTILISATION DE L'OUTIL DDD

On souhaite maintenant expérimenter le debugger `ddd` sur le programme `testeliste` créé à la question précédente.

2.1

Visualisez l'ensemble des chaînages produits par ce programme lorsqu'on exécute la fonction `afficher`.

On veut maintenant visualiser ce qu'il se passe lors de la désallocation des éléments de la liste.

2.2

Exécutez la fonction `detruire` pas à pas. Que constatez-vous ?

2.3

Proposez une modification du programme qui permette de résoudre ce problème.

3 . ECRIRE UN PROGRAMME C

3.1

Ecrivez un programme permettant de calculer le maximum d'une liste d'entiers passés par la ligne de commande. Générez l'exécutable, testez votre programme et débuggez-le si besoin à l'aide de `ddd`.

TME 2 - GESTION DU TEMPS

ECRITURE D'UNE COMMANDE MYTIMES

L'objectif de ce TME est d'écrire un programme C `mytimes` qui, comme la commande shell `time`, permet d'afficher les statistiques d'utilisation du processeur pour n'importe quelle commande shell. En particulier, `mytimes` devra afficher le temps passé en mode utilisateur et en mode système.

Fonctions utiles au TP :

Les commandes du shell :

`man` (indispensable sous unix… Pensez au `man -a`),
`time, nice, ps.`

Les fonctions en C (voir leurs descriptions détaillées en utilisant le manuel en ligne "`man`") :

`gettimeofday` permet de récupérer le temps écoulé depuis le 1er Janvier 70.

`times` permet de récupérer les statistiques d'utilisation de temps CPU d'un programme et de ses fils. Ces statistiques sont exprimées en nombre de "ticks" horloge. La durée d'un tick horloge est obtenue en appelant la fonction `sysconf` avec le paramètre `_SC_CLK_TCK`.

`system` permet de lancer une commande shell depuis un programme C.

1. STATISTIQUES D'EXECUTION D'UNE COMMANDE SHELL

1.1

Exécutez la commande shell `time` pour afficher les statistiques d'utilisation du processeur pour la commande « `sleep 5` ». Que constatez-vous ?

2. LANCEMENT D'UNE COMMANDE SHELL DEPUIS UN PROGRAMME C

2.1

Ecrivez une fonction C

`void lance_commande(char *` commande`)`

qui utilise la fonction `system` pour lancer l'exécution de la commande shell passée en paramètre. `lance_commande` doit renvoyer un message d'erreur si la commande n'a pu être exécutée correctement.

2.2

Ecrivez une fonction `main` qui appelle `lance_commande` pour chaque argument passé sur la ligne de commande. Générez un exécutable `mytimes`.

3. CALCUL DU TEMPS DE REPONSE EN UTILISANT GETTIMEOFDAY

3.1

Modifiez votre fonction `lance_commande` pour afficher le temps mis par l'exécution de la commande. La mesure du temps pourra être effectuée à l'aide de la fonction `gettimeofday` .

3.2

Testez votre programme avec la commande :

```
$ ./mytimes "sleep 5" "sleep 10"
```

4. CALCUL DES STATISTIQUES

La version précédente de permet de connaître le temps qui sépare le début d'exécution de la commande de sa terminaison. On veut maintenant affiner les résultats obtenus en mesurant le temps passé mode utilisateur et le temps passé en mode système.

4.1

Créez une nouvelle version de la fonction `lance_commande` qui affiche toutes les statistiques fournies par la fonction `times`. On fournit ci-dessous un exemple d'exécution de mytimes. Votre programme devra fournir un affichage similaire.

```
$ ./mytimes "ls -l" "cd /usr/include ; grep time *.h > /dev/null"
total 32
-rw-r--r--    1 sensl    lip6          145 oct 11 14:15 Makefile
-rwxr-xr-x    1 sensl    lip6        14959 oct 11 14:15 mytimes
-rw-r--r--    1 sensl    lip6         1283 oct 11 14:11 mytimes.c
-rw-r--r--    1 sensl    lip6         1281 oct 11 14:10 mytimes.c~
-rw-r--r--    1 sensl    lip6         2072 oct 11 14:14 mytimes.o
-rw-r--r--    1 sensl    lip6            0 oct 11 15:16 t.txt

Statistiques de "ls -l"
Temps total  :   0.0300
Temps utilisateur : 0.0000
Temps systeme : 0.0000
Temps util. fils : 0.0300
Temps sys. fils : 0.0000

Statistiques de "cd /usr/include ; grep time */*.h > /dev/null"
Temps total  :   5.6200
Temps utilisateur : 0.0000
Temps systeme : 0.0000
Temps util. fils : 0.1800
Temps sys. fils : 0.1600
```

4.2

Testez votre nouvelle version avec l'exemple suivant :

```
$ ./mytimes "sleep 5"
```

Comparez vos résultats avec ceux de la question 1.1.

5. CHANGEMENT DE PRIORITE

La commande `nice` permet de changer les priorités d'un programme. En tant qu'utilisateur non privilégié on ne peut que baisser la priorité de ses processus de façon à avantager les autres programmes (d'où le nom "`nice`" de la commande). Au maximum, on peut baisser la priorité d'un processus de 19.

5.1

Tapez la commande `ps -l`. Quelle est la priorité du processus `ps` ?

5.2

Tapez la commande `/bin/nice -19 ps -l`. Quelle est maintenant la priorité de la commande `ps` ?

5.3

Ecrivez un programme C dans lequel s'exécute une boucle effectuant 10^8 itérations.

5.4

Lancez deux exécutions en parallèle de votre programme de la question précédente, en mesurant leur temps d'exécution avec my`times`. L'une des exécutions sera lancée en priorité normale, l'autre en abaissant au maximum la priorité. Que constatez vous ?

TME 3 - ORDONNANCEMENT DE TACHES

Fonctions utiles au TME :
Celles définies dans la bibliothèque `libsched` présentée en annexe.

1 TESTER LA BIBLIOTHEQUE

1.1

Créez un répertoire TP3. Dans ce répertoire, copiez l'exemple fourni par la bibliothèque avec le `Makefile` :

```
$ mkdir TP3
$ cp /Infos/lmd/2005/licence/ue/li324-2006fev/libsched/demo/Makefile TP3
$ cp /Infos/lmd/2005/licence/ue/li324-2006fev/libsched/demo/main.c TP3
```

1.2

Compilez l'exemple et testez

```
$ cd TP3
$ make main
$ ./main
```

1.3

Modifiez les paramètres de l'ordonnanceur : changez le quantum de temps, testez l'élection aléatoire.

2 ECRITURE D'UN NOUVEL ALGORITHME D'ORDONNANCEMENT SJF

On considère une stratégie sans temps partagé. On souhaite implanter une stratégie SJF (Shortest Job First) donnant la priorité aux tâches courtes. Lors de la création des tâches, on précise dans le paramètre "duration" (le troisième paramètre) la durée estimée de la tâche.

Le scénario de test est fourni dans le fichier `scen.c` dans le répertoire
`/Infos/lmd/2005/licence/ue/li324-2006fev/libsched/demo/`

2.1

Copiez le fichier `scen.c` dans votre répertoire TP3.

Ecrivez la nouvelle fonction d'élection `SJFElect` (sur le modèle de l'exemple de l'élection aléatoire) implémentant l'ordonnancement SJF.

2.2

Compilez et testez votre programme:

```
$ make scen
$ ./scen
```

3 APPROXIMATION DE SJF EN TEMPS PARTAGE

L'algorithme SJF suppose de connaître à l'avance le temps d'exécution d'une tâche (ce qui est rarement le cas dans les systèmes). On veut privilégier les tâches courtes sans connaître le temps estimé des tâches.

3.1

Ecrivez la fonction `ApproxSJF` qui implante un algorithme privilégiant les tâches courtes en temps partagé. Indication : vous serez amenés à utiliser le champ `ncpu` qui indique le nombre de quantums de temps consommés par chaque tâche.

3.2

Testez votre programme avec un quantum d'une seconde. Comparez vos résultats avec l'algorithme aléatoire.

3.3

Votre algorithme peut-il provoquer une famine (c'est-à-dire une situation où une tâche prête n'est jamais élue) ? Si oui modifiez-le pour éviter ce problème.

LIBSCHED : MODE D'EMPLOI

http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2005/ue/sys-2006fev/libsched

La bibliothèque d'ordonnancement `libsched` permet de tester des algorithmes d'ordonnancement de fonctions utilisateur. L'utilisateur a l'illusion que ses fonctions s'exécutent en parallèle.
Grâce à `libsched`, on peut définir et paramétrer de nouveaux algorithmes d'ordonnancement.

Deux fichiers sont fournis par la bibliothèque : `libsched.a` et le fichier d'inclusion `sched.h`

1. FONCTIONS DE LA LIBRAIRIE

```
#include "sched.h"

int CreateProc(function_t func, void *arg, int duration);
```

Cette fonction permet de créer une nouvelle fonction (que l'on appelle processus léger) qui pourra s'exécuter en parallèle avec d'autres. Le paramètre `func` est le nom de la fonction, `arg` est un pointeur vers les arguments de la fonction et `duration` est la durée estimée de la fonction. Par défaut le paramètre duration n'est pas utilisé mais il peut être utile pour des algorithmes d'ordonnnancement du type SJF (Shortest Job First).

`CreateProc` retourne l'identifiant du processus léger créé (`pid`).

```
void SchedParam(int type, int quantum, int (*felect)(void));
```

Cette fonction permet de régler les paramètres de l'ordonnanceur. `type` indique le type d'ordonnancement. 3 types sont possibles (définis dans `sched.h`) :

- BATCH indique un ordonnancement sans temps partagé de type FIFO. Dans ce cas, les paramètres `quantum` et `felect` sont ignorés.
- PREMPT indique un ordonnancement préemptif de type "tourniquet". C'est l'ordonnancement par défaut. Dans ce cas, le paramètre `quantum` fixe la valeur du quantum de temps en secondes.
- NEW indique une nouvelle stratégie d'ordonnancement (définie par l'utilisateur). Dans ce cas, le paramètre `quantum` fixe la valeur du quantum de temps en secondes. Si `quantum` est égal à 0, l'ordonnancement devient non préemptif (sans temps partagé). Le paramètre `felect` est le nom de la fonction d'élection qui sera appelée automatiquement par la librairie avec une période de "`quantum` " secondes (si `quantum` est différent de 0).

La fonction d'élection `felect` doit avoir la forme suivante :

```
int Mon_election(void) {
/* Choix du nouveau processus élu */
    return elu;
}
```

La fonction d'élection choisit, parmi les processus à l'état `RUN`, le nouveau processus élu en fonction des informations regroupées dans la table `Tproc` définie dans `sched.h`:

```
struct proc {
    int flag;                          // Etat de la tâche : RUN, IDLE ou ZOMB
    int prio;                          // Priorité
    int pid;                           // Pid

...
    struct timeval end_time;           // date de fin
    struct timeval start_time;         // date de création
    struct timeval realstart_time;     // date de lancement
    double ncpu;                       // temps "cpu" consommé
    double duration;                   // temps estimé de la tâche
} Tproc[MAXPROC];
```

La priorité d'un processus varie entre les deux constantes `MINPRIO` et `MAXPRIO`. Plus la valeur de la priorité est élevée, plus le processus est prioritaire.

Une fois le processus choisi, `felect` doit retourner l'indice dans `Tproc` du processus élu.

```
int GetElecProc(void);
```

Fonction qui retourne l'indice dans `Tproc` du processus élu (celui qu'on va décharger). Elle retourne –1 si aucun processus n'est élu.

```
void sched(int printmode);
```

Cette fonction lance l'ordonnanceur. L'ordonnancement effectif des processus ne commence qu'à partir de l'appel à cette fonction. Par défaut l'ordonnanceur exécute un algorithme similaire à Unix à base de priorité dynamique. Le paramètre `printmode` permet de lancer l'ordonnanceur en mode "verbeux ". Si `printmode` est différent de 0, l'ordonnanceur affichera à chaque commutation la liste des tâches prêtes. Cette fonction se termine lorsqu'il n'existe plus de tâche à l'état prêt (`RUN`).

```
void PrintStat(void);
```

Cette fonction affiche les statistiques sur les tâches exécutées (temps réel d'exécution, temps processeur consommé, temps d'attente).

2. EXEMPLE

L'exemple suivant illustre l'utilisation des primitives.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <malloc.h>
#include <sched.h>

// Fonction utilisateur

void MonProc(int *pid) {
    long i;

    for (i=0;i<8E7;i++)
        if (i%(long)4E6 == 0)
            printf("%d - %ld\n",*pid, i);

    printf("############ FIN PROC %d\n\n", *pid );
}
```

```
// Exemple de primitive d'election definie par l'utilisateur
// Remarques : les primitives d'election sont appelées directement
// depuis la librairie. Elles ne sont appelées que si au
// moins un processus est à l'etat pret (RUN)
// Ces primitives manipulent la table globale des processus
// définie dans sched.h

// Election aléatoire

int RandomElect(void) {
    int i;

    printf("RANDOM Election !\n");

    do {
        i = (int) ((float)MAXPROC*rand()/(RAND_MAX+1.0));
    } while (Tproc[i].flag != RUN);
    return i;
}


int main (int argc, char *argv[]) {
    int i;
    int *j;

    // Créer 3 processus

    for (i = 0; i < 3; i++) {
        j = (int *) malloc(sizeof(int));
        *j= i;
        CreateProc((function_t)MonProc,(void *)j, 0);
    }

    // Exemples de changement de paramètres

    // Définir une nouvelle primitive d'election avec un quantum de 2 secondes
    SchedParam(NEW, 2, RandomElect);

    // Redéfinir le quantum par defaut
    SchedParam(PREMPT, 2, NULL);

    // Passer en mode batch
    SchedParam(BATCH, 0, NULL);

    // Lancer l'ordonnanceur en mode non "verbeux"
    sched(0);

    // Imprimer les statistiques
    PrintStat();

    return EXIT_SUCCESS;
}
```

3. UTILISATION

La `libsched` est disponible dans le répertoire
`/Infos/lmd/2005/licence/ue/li324-2006fev/libsched`.

Le sous-répertoire `demo` contient l'exemple précédent avec un `Makefile` permettant de générer directement l'exécutable.
Le sous-répertoire `src` contient le source de `libsched`. Il contient également la librairie " `libelf` " utilisée en interne par l'ordonnanceur.

Pour générer un exécutable, le plus simple est de copier le `Makefile` de l'exemple :

```
$ cp /Infos/lmd/2005/licence/ue/li324-2006fev/libsched/demo/Makefile .
```

Modifiez ensuite éventuellement le `Makefile`, en remplaçant le nom des fichiers exemples par celui de votre fichier, puis générez un exécutable :

```
$ make
```

4. POUR INSTALLER LA LIBRAIRIE CHEZ SOI

La librairie est disponible sous forme d'archive compressée sur la page web de la licence :
http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2005/ue/sys-2006fev/libsched

Pour la décompresser il suffit de taper sur votre machine :

```
$ tar xvfz libsched.tgz
```

Un répertoire libsched est créé, contenant les deux répertoires demo et src. Il suffit de compiler la librairie :

```
$ cd libsched/src
$ make
```

Vous pouvez alors compiler l'exemple et l'exécuter :

```
$ cd ../demo
$ make
$ ./main
```

Pour tout commentaire, ou rapport de " bug " : `Pierre.Sens@lip6.fr`

TME 4 - GESTION DE PROCESSUS

ÉCRITURE D'UNE VERSION PARALLELE DE LA COMMANDE GREP

Objectif : Manipulation des processus sous Unix.

Utilisation des appels systèmes : `fork`, `execl`, `wait` et `wait3`.

On souhaite implanter en C un "multi-grep" qui exécute la commande standard Unix `grep` en parallèle. `grep` permet de rechercher une chaîne de caractères dans un fichier et elle affiche les lignes où la chaîne apparaît.
L'exécution de votre programme devra être appelée de la manière suivante:

```
$ mgrep chaine liste-fichiers
```

Cette commande devra afficher, pour chaque fichier passé en paramètre, les lignes contenant la chaîne `chaine`. Elle créera autant de processus fils que de fichiers passés en paramètre.

1. MULTI-GREP SIMPLE

Ecrivez un programme C qui lance, pour chaque fichier passé en paramètre, un processus fils qui exécute le `grep` standard. Le programme (c'est-à-dire le père) ne doit se terminer que lorsque tous les fils ont terminé.

2. MULTI-GREP A PARALLELISME CONTRAINT

On souhaite désormais ne créer simultanément qu'un nombre maximum `MAXFILS` de processus fils. Si le nombre de fichiers est supérieur à `MAXFILS`, le processus père ne crée dans un premier temps que `MAXFILS` fils. Dès qu'un des fils se termine et s'il reste des fichiers à analyser, le père recrée un nouveau fils.

2.1

Modifiez votre programme en conséquence

3. MULTI-GREP AVEC STATISTIQUES

3.1

Modifiez votre programme de la partie 1 pour afficher les statistiques d'utilisation CPU système et utilisateur de chaque fils.

4. PROCESSUS "ZOMBIE"

4.1

Qu'est ce qu'un processus "zombie" ?

4.2

Ecrivez un programme qui crée pendant 10 secondes deux processus zombie.

TME 5 - 6 : SYNCHRONISATIONS PAR SEMAPHORES

REALISATION D'UN MECANISME DE DIFFUSION

On veut réaliser un mécanisme de diffusion. Il y a deux types de processus : les émetteurs (au nombre de NE) et les récepteurs (au nombre de NR). Les émetteurs déposent des messages dans un tampon initialement vide. Chaque message doit être lu par tous les récepteurs avant de pouvoir être effacé.

Les processus émetteurs et récepteurs sont cycliques. Autrement dit, chaque processus est une boucle infinie. Les émetteurs doivent se bloquer lorsqu'il n'y a pas de case libre, les récepteurs lorsqu'il n'y a pas de message à lire. Par contre, on autorise plusieurs récepteurs à lire simultanément un message.

Les synchronisations entre émetteurs et récepteurs sont réalisées au moyen de sémaphores et variables partagées.

LES SOLUTIONS QUE VOUS PROPOSEZ DOIVENT ETRE PROGRAMMEES A L'AIDE DE LA BIBLIOTHEQUE DE MANIPULATION DE SEMAPHORES `libIPC`, dont le descriptif est donné en annexe. Vous pouvez récupérer 2 squelettes de programme dans le répertoire

`/Infos/lmd/2005/licence/ue/li324-2006fev/TME6-7`

1. MISE EN OEUVRE AVEC UN TAMPON A UNE SEULE CASE

On considère d'abord le cas où le tampon ne contient qu'une case. Pour assurer la synchronisation, on définit les sémaphores et variables suivants :

- Le sémaphore `EMET` bloque les émetteurs tant qu'il n'y a pas de case dans laquelle ils puissent écrire.
- Le tableau de sémaphores `RECEP[1..NR]` bloque les récepteurs.
- Le compteur partagé `nb_recepteurs` indique le nombre de consommateurs ayant déjà lu le message produit par le producteur.

1.1.

Quel mécanisme de protection supplémentaire doit être introduit pour assurer la cohérence de `nb_recepteurs` ? Donnez les initialisations des sémaphores et variables partagées.

1.2.

Pourquoi utilise-t-on un tableau de sémaphores `RECEP[1..NR]`, dont chaque case est augmentée de 1 lors de l'émission d'un message, plutôt qu'un unique sémaphore `RECEP`, augmenté de `NR` lors de l'émission d'un message ? Qui prévient les émetteurs de la disponibilité de la case ?

1.3.

Programmez cette synchronisation en utilisant les primitives de la bibliothèque `libIPC`.

2. MISE EN OEUVRE AVEC UN TAMPON A NMAX CASES

On considère maintenant le cas d'un tampon à `nmax` cases, et on souhaite assurer un maximum de parallélisme au niveau de l'accès au tampon. En particulier :

- deux émetteurs doivent pouvoir déposer simultanément des messages s'ils écrivent dans des cases différentes ;
- un émetteur et un récepteur peuvent accéder simultanément au tampon s'ils n'accèdent pas à la même case.

A nouveau, chaque message déposé doit être lu par *tous* les récepteurs. On suppose que le tampon est utilisé de manière circulaire (avec une attribution ordonnée des cases et non avec une liste de cases vides et une liste de cases pleines).

2.1.

Définissez les sémaphores et variables nécessaires pour programmer cette synchronisation. Pour chaque sémaphore et variable vous préciserez son rôle et sa valeur initiale.

2.2.

Donnez les algorithmes des émetteurs et récepteurs. Décrivez brièvement un scénario de fonctionnement du système ainsi synchronisé, en insistant sur le parallélisme des émetteurs entre eux, des récepteurs entre eux, et des émetteurs vis à vis des récepteurs.

2.3.

Programmez cette synchronisation en utilisant les primitives de la bibliothèque libIPC.

Annexe : UTILISATION DE LA BIBLIOTHEQUE DE SEMAPHORES LIBIPC

La bibliothèque de sémaphores, appelée `libIPC`, est un outil facilitant l'utilisation des sémaphores au-dessus d'Unix. Elle reprend l'interface classique des sémaphores.

1. LES PRIMITIVES

libIPC contient les primitives suivantes :

```
#include <libipc.h>
```

```
int creer_sem(int nb);
```

Fonction qui crée `nb` sémaphores non initialisés. Cette fonction retourne -1 en cas d'erreur. Les sémaphores sont numérotés à partir de 0. Par exemple l'appel à `creer_sem(3)` va créer 3 sémaphores numérotés 0, 1 et 2.

```
int init_un_sem(int sem, int val);
```

Fonction qui initialise le sémaphore numéro `sem` à la valeur val. Cette fonction retourne -1 en cas d'erreur.

```
void P(int sem);
```

Réalisation de la primitive P sur le sémaphore numéro `sem`.

```
void V(int sem);
```

Réalisation de la primitive V sur le sémaphore numéro `sem`.

```
int det_sem(void);
```

Fonction qui détruit tous les sémaphores. Elle doit impérativement être appelée à la fin du programme. Cette fonction retourne -1 en cas d'erreur.

libIPC permet également de créer des segments de mémoire partagée :

```
char* init_shm(int taille);
```

Fonction qui crée un segment de mémoire partagé de taille `taille`. Elle retourne l'adresse du segment créé. Cette fonction retourne NULL en cas d'erreur.

```
int det_shm(char *seg);
```

Fonction qui détruit le segment désigné par `seg`. Elle doit impérativement être appelée à la fin du programme. Cette fonction retourne -1 en cas d'erreur.

2. COMMANDES SHELL

Il existe quelques commandes Unix permettant d'intervenir sur les sémaphores en cas de problème (par exemple la non destruction des sémaphores) :

`ipcs`	permet d'afficher la liste des sémaphores et segments de mémoire partagée définis sur la machine.
`ipcrm sem id`	permet de détruire le sémaphore identifié par `id`.
`ipcrm shm id`	permet de détruire le segment de mémoire partagée identifié par `id`.

3. UTILISATION DE LA BIBLIOTHEQUE

`libIPC` est définie dans le répertoire `/Infos/licence/2003/sys/libipc/lib`. Lorsqu'un programme utilise les primitives de libIPC, il faut lier son exécutable à libIPC.

Le répertoire `/Infos/licence/2003/sys/libipc/demo` contient un exemple de programme utilisant libIPC, ainsi que le Makefile permettant de lier son exécutable à libIPC.

Le plus simple est de copier le fichier `Makefile` contenu dans ce répertoire

```
$ cp /Infos/licence/2003/sys/libipc/demo/Makefile .
```

puis de le modifier éventuellement en remplaçant `demo_ipc` par le nom de votre fichier. Vous pouvez alors générer l'exécutable.

```
$ make
```

4. POUR INSTALLER LA LIBRAIRIE CHEZ SOI

La librairie est disponible sous forme d'archive compressée sur la page web de la licence :
`http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2006/ue/LI324-2007fev/public/libipc/libipc.tgz`

Pour la décompresser il suffit de taper sur votre machine :

```
$ tar xvfz libipc.tgz
```

Un répertoire libipc est créé contenant les deux répertoires `demo` et `src`. Il suffit de compiler la librairie :

```
$ cd libipc/src
$ make
```

Vous pouvez alors compiler l'exemple présenté en 5 et l'exécuter :

```
$ cd ../demo
$ make
$ ./demo-ipc
```

5. EXEMPLE D'UTILISATION

Le programme ci-dessous réalise une barrière à trois processus. Le processus père crée deux fils et les attend à la barrière.

```
#include <libipc.h>

/* Definition des semaphores */

#define SEM1    0
#define SEM2    1

/* Définition du format du segment de memoire partagée */

typedef struct {
     int a;
} t_segpart;

t_segpart *sp; /* Pointeur sur le segment */

/* fonction exécutée par le premier processus fils */
void fils1(void)
{
     sleep(10);
     sp->a++;
     V(SEM1);
     printf("fin1\n");
     exit(0);
}

/* fonction exécutée par le second processus fils */
void fils2(void)
{
     sleep(10);
     sp->a++;
     V(SEM2);
     printf("fin2\n");
     exit(0);
}

int main(int argc, char *argv[])
{
     int semid, pid;

     /* Creer les semaphores */
     if ((semid = creer_sem(2)) == -1) {
          perror("creer_sem");
          exit(1);
     }

     /* Initialiser les valeurs */
     init_un_sem(SEM1,0);
     init_un_sem(SEM2,0);

     /* Creer le segment de memoire partagee */
     if ( (sp = init_shm(sizeof(t_segpart))) == NULL) {
          perror("init_shm");
          exit(1);
     }
     sp->a = 0;

     /* Creer le premier processus fils */
     if ((pid = fork()) == -1) {
```

```
            perror("fork");
            exit(2);
        }
        if (pid == 0) {
            /* Premier processus fils */
            fils1();
        }

        /* Creer le second processus fils */
        if ((pid = fork()) == -1) {
            perror("fork");
            exit(2);
        }
        if (pid == 0) {
            /* Second processus fils */
            fils2();
        }

        /* Processus Pere */
        printf("le pere attend...\n");
        P(SEM1);
        printf("fin du fils 1\n");
        P(SEM2);
        printf("fin du fils 2\n");
        printf("valeur du compteur = %d\n", sp->a);

        /* Destruction des semaphores et du segment de mémoire */
        det_sem();
        det_shm(sp);

        return EXIT_SUCCESS;
}
```

Pour tout commentaire, ou rapport de « bug » : Pierre.Sens@lip6.fr

TME 7 -8 – GESTION MEMOIRE

IMPLANTATION D'UNE GESTION DE TAS

Le tas est une zone mémoire réservée par le système pour permettre à un programme de faire de l'allocation dynamique (malloc, free). L'objectif de ce TME est de simuler une gestion simplifiée du tas.

On suppose ici que le tas est une zone de taille fixe égale à 128 octets.

On se propose de programmer les primitives `tas_malloc()` et `tas_free()` qui permettent respectivement d'allouer et de libérer une zone dans le tas :

`char *tas_malloc(unsigned int taille);`

réserve dans le tas une zone de `taille` octets. Cette fonction retourne l'adresse du début de la zone allouée. En cas d'erreur (si l'allocation est impossible), la fonction retourne `NULL`.

`int tas_free(char *ptr);`

libère la zone dont le début est désigné par `ptr`.

Pour gérer les espaces occupés et les espaces libres dans le tas, on utilise les structures de données suivante :

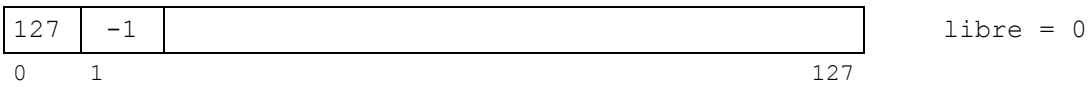
- une zone allouée contient 2 champs :
  - un octet donnant la taille `TD` de la donnée stockée,
  - la donnée elle-même.

  La taille de la zone est donc `TD+1`.
- une zone libre contient 2 champs :
  - un octet donnant la taille `TL` de la donnée pouvant être stockée dans la zone,
  - un octet donnant l'indice dans le tas du début de la zone libre suivante. Si la zone libre est la dernière, cet octet prend la valeur –1.

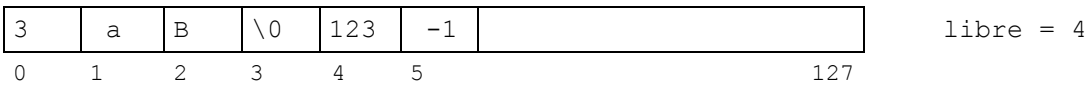
  La taille de la zone est donc `TL+1`.

On dispose en outre d'une variable `libre` contenant l'indice de début de la première zone libre du tas.

Initialement le tas est vide. On a donc l'image suivante :



Après exécution de l'opération
```
p1 = (char *) tas_malloc(3);
strcpy( p1, "ab" );
```
on obtiendrait l'image :



1. EXECUTION MANUELLE

1.1.

On suppose un tas initialement vide. Donnez l'apparence du tas après l'insertion d'une donnée de taille maximum.

1.2.

On suppose un tas initialement vide. Représentez l'apparence du tas après l'exécution des opérations suivantes :

```
char *p1, *p2, *p3, *p4, *p5;
p1 = (char *) tas_malloc(10);
p2 = (char *) tas_malloc(9);
p3 = (char *) tas_malloc(5);
strcpy( p1, "tp 1" );
strcpy( p2, "tp 2" );
strcpy( p3, "tp 3" );
tas_free( p2 );
p4 = (char *) tas_malloc(8);
strcpy( p4, "systeme" );
```

2. PROGRAMMATION

L'allocation d'une zone dans le tas s'effectue en deux étapes :

- recherche suivant une stratégie prédéfinie (best-fit, worst-fit, first-fit) d'un emplacement libre d'une taille suffisante ;
- réservation de cet emplacement pour stocker la donnée : mise à jour de la variable `libre` et du chaînage des blocs.

Quelle que soit la stratégie utilisée, le prototype de la fonction de recherche est le suivant :

```
int strategie(int taille, int *pred);
```

recherche dans le tas une zone libre de `taille` octets. Cette fonction retourne l'adresse du début de la zone, -1 si aucune zone de taille suffisante n'existe. `*pred` est l'adresse dans le tas du début de la zone précédant la zone retournée.

Les fichiers utiles au TME se trouvent dans le répertoire :

`/Infos/licence/2003/sys/TME10`

Vous y trouverez une bibliothèque avec une fonction d'initialisation du tas et une fonction affichant le contenu du tas, ainsi que le Makefile correspondant.

2.1.

Programmez la fonction `first_fit()`.

2.2.

Programmez la fonction `tas_malloc()` en implantant une stratégie first-fit.

2.3.

Facultatif : Programmez la fonction `tas_free()`.

2.4.

Programmez le jeu d'essai de la question 1.2 et affichez l'apparence du tas. Vous pourrez pour cela utiliser la fonction `afficher_tas()` définie dans `affiche_tas.h`.

TME 9 – REMPLACEMENT DE PAGES

LIBMEM

Pré requis : lire le mode d'emploi de la libmem

Copiez tous les fichiers du répertoire du TME dans votre « home »
$ cp /Infos/lmd/2005/licence/ue/li324-2006fev/libmem/TME/* .

Dans la suite vous devez compléter les fichiers Fifo.c et LRU.c en vous inspirant des exemples LFU et Random du mode d'emploi que vous trouverez dans le répertoire
/Infos/lmd/2005/licence/ue/li324-2006fev/libmem/algorithms

1. ECRITURE D'UNE STRATEGIE DE REMPLACEMENT FIFO

1. Complétez le fichier Fifo.c

2. Compilez et exécutez votre programme en lui donnant en entrée la suite de références contenu dans le fichier bench

$ make mainFifo
$ ./mainFifo < bench

2. ECRITURE D'UNE STRATEGIE DE REMPLACEMENT LRU

1. Complétez le fichier LRU.c

2. Compilez et exécutez votre programme en lui donnant en entrée la suite de références contenu dans le fichier bench

$ make mainLRU
$ ./mainLRU < bench

3. COMPARAISON LRU/FIFO

Comparez les résultats obtenus. Vérifiez en exécutant la suite de références « à la main » que vos deux algorithmes implémentent bien respectivement les stratégies LRU et FIFO.

LIBMEM : MODE D'EMPLOI

http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2005/ue/sys-2006fev/libmem

La bibliothèque `libmem` permet de tester des algorithmes de remplacement de pages.
L'arborescence est la suivante :

```
libmem
|-- Makefile
|-- README
|-- algorithms
|   |-- LFU.c
|   |-- Random.c
|   `-- main.c
|-- bench
|-- bin
|-- include
|   |-- LFU.h
|   |-- Random.h
|   |-- Swapper.h
|   `-- libmem.h
|-- lib
|-- obj
`-- src
    |-- Swapper.c
    `-- libmem.c
```

Le répertoire `src` contient les sources de la bibliothèque
Le répertoire `include` contient les fichiers d'en-tête de libmem (`libmem.h, Swapper.h`) et les fichiers propres aux exemples (`LFU.h`).

Le répertoire `algorithms` contient un exemple de main (`main.c`) ainsi que des exemples d'algorithmes de remplacement de pages :
`LFU.c` implémente l'algorithme Least Frequently Used
`Ramdon.c` implémente un algorithme qui choisit une page aléatoirement

1. FONCTIONS DE LA LIBRAIRIE

Remarque : dans le code de la libmem, la terminologie anglaise est utilisée : frame = case en mémoire physique, page = page en mémoire virtuelle.

Structure de données

La libmem maintient une structure de données « Swapper » définie dans Swapper.h :

```
typedef int Page;

typedef struct Swapper {
…
      unsigned int frame_nb;        /* Nombre de cases de la mémoire physique */
      Page *      frame;            /* Tableau des cases en mémoire */
      void *      private_data;   /* Donnée privée propre à chaque stratégie */
} Swapper;
```

Le champ `frame` indique pour chaque case la page correspondante :

Pour la case i, si i est libre frame[i] = -1, sinon frame[i] = numéro de la page.

Fonctions

libmem fournit deux fonctions :

```
#include "Swapper.h"

int initSwapper(
      Swapper* swap,
      unsigned int nc,                                  //Logical number of frames
      int (*Init)(Swapper*),                            //Init for private data
      void (*Reference)(Swapper*,unsigned int),         //Reference to keep stats
      unsigned int (*Choose)(Swapper*),                 //Choose function
      void (*Finalize)(Swapper*)                        //Finalize function
);
```

initSwapper permet d'initialiser la structure swap avec nc cases. 4 fonctions sont passées en paramètre. `Init` sera appelée à l'initialisation, `Reference` à chaque accès à une case, `Choose` à chaque défaut de page et `Finalize` à la terminaison.

Pour programmer une stratégie de remplacement de pages appelée par exemple « MaStrategie », il faut donc :

1) programmer les 4 fonctions suivantes :

```
int initMaStrategie(Swapper *swap);
```

Où est la variable swap est celle initialisée par la fontion initSwapper.
Cette fonction permet éventuellement d'allouer et d'initialiser le champ `private_data` de la struture swap.
La fonction doit retourner 0 en cas de succès

```
void referenceMaStrategie(Swapper *swap, unsigned int frame) ;
```

Fonction qui sera appelée lors d'un accès à la case mémoire numéro `frame`.

```
unsigned int chooseMaStrategie(Swapper *swap) ;
```

Fonction qui sera appelée lors d'un défaut de page. Elle doit retourner le numéro de la case contenant la page victime.

```
Void finalizeMaStrategie(Swapper *swap) ;
```

Fonction appelée à la fin du programme pour libérer de qui a été allouée par initMaStratégie.

2) Appeler initSwapper de la manière suivante :

```
Swapper *s,
initSwapper(&s, nbcases, initMaStrategie, refenceMaStrategie, chooseMaStrategie,
finalizeMaStrategie);
```

```
#include "libmem.h"

int   swapSimulation(Swapper *swap,FILE *f);
```

Lance la simulation des accès aux pages. `swap` doit être préalablement initialisé par initSwapper. Le fichier `f` contient un entier par ligne : la première ligne indique le nombre de cases de la mémoire physique et les autres lignes, la suite de référence mémoire.

2. EXEMPLE

L'exemple suivant illustre l'utilisation des primitives. On souhaite implémenter une stratégie de remplacement de page de type LFU (Least Frequently Used). Pour cela il faut avoir un tableau qui compte le nombre de la référence à une case. En cas de remplacement de page, il suffit de choisir la case ayant le compteur avec une valeur minimum.

include/LFU.h :

```
#include "Swapper.h"

int initMFUSwapper(Swapper*,unsigned int);
```

algorithms/LFU.c :

```
#include "LFU.h"
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int   initLFU(Swapper*);
void  referenceLFU(Swapper*,unsigned int frame);
unsigned int chooseLFU(Swapper*);
void  finalizeLFU(Swapper*);


int initLFUSwapper(Swapper*swap,unsigned int frames){
      initSwapper(swap,frames,initLFU,referenceLFU,chooseLFU,finalizeLFU);
}

int   initLFU(Swapper*swap){

      /* Allouer un tableau avec un compteur d'utilisation pour chaque case */

      swap->private_data = calloc(swap->frame_nb,sizeof(int));
      int * use = (int*)swap->private_data;
      int i;

      /* Initialisation du tableau */

      for( i=0 ; i<swap->frame_nb ; i++ )
            use[i] = 0;
      return 0;
}

void  referenceLFU(Swapper*swap,unsigned int frame){
      int i;
      int * use = (int*)swap->private_data;
```

```
      /* A chaque acces à la case frame augmente son compteur d'utilisation */
      use[frame]++;
}

unsigned int chooseLFU(Swapper*swap){
      int i, frame = 0;
      int * use = swap->private_data;

/* Choisir la case (contenant une page)  ayant le plus petit compteur */

      for ( i=0 ; i<swap->frame_nb ; i++ ){
            if( swap->frame[i] == -1 ){
                  frame = i;
                  break;
            }
            if( use[i] < use[frame] )
                  frame = i;
      }

      use[frame] = 0;

      return frame;
}

void  finalizeLFU(Swapper*swap){
      free(swap->private_data);
}
```

algorithms/main.c :

```
#include "libmem.h"
#include "LFU.h"

int main(int argc,char*argv[]){
      unsigned int frame_nb;
      Swapper     s;

      scanf("%i",&frame_nb);

      /* Initialisation du Swapper de la stratégie LFU */
      initLFUSwapper(&s,frame_nb);

      /* Lancer la simulation */
      if(swapSimulation(&s,stdin)<0){
            printf("Error during swap simulation !!!\n");
            return -1;
      }

      return 0;
}
```

3. UTILISATION

La `libsched` est disponible dans le répertoire
`/Infos/lmd/2005/licence/ue/li324-2006fev/libmem`

Tout d'abord, copiez la libmem dans un répertoire sur votre home

```
$ cp –r /Infos/lmd/2005/licence/ue/li324-2006fev/libmem .
```

Le sous-répertoire `libmem/algorithms` contient l'exemple précédent ainsi qu'une élection aléatoire (Random.c). Dans `libmem` un fichier makefile permet de générer directement l'exécutable et la bibliothèque en exécutant les commandes suivantes :

```
$ cd libmem
$ make
```

Pour créer une nouvelle stratégie (MaStrategie), il faut : créer un fichier MaStragie.h dans le répertoire include, un fichier MaStratgie.c dans algoritms et modifier le fichier main.c dans algorithms.

Le fichier `libmem/bench` contient un exemple de suite de référence mémoire pour 3 cases. Pour lancer la libmem sur cet exemple il faut entrer :

```
$ bin/main < bench
Page 1 referenced
LFU uses:

/!\ PAGE FAULT !!! /!\
Frame 0 has been choosen
(frame 0: 1) (frame 1: _) (frame 2: _)

Page 2 referenced
LFU uses: (page:1 time:1)

/!\ PAGE FAULT !!! /!\
Frame 1 has been choosen
(frame 0: 1) (frame 1: 2) (frame 2: _)

Page 3 referenced
LFU uses: (page:1 time:1) (page:2 time:1)

/!\ PAGE FAULT !!! /!\
Frame 2 has been choosen
(frame 0: 1) (frame 1: 2) (frame 2: 3)

Page 4 referenced
LFU uses: (page:1 time:0) (page:2 time:1) (page:3 time:1)

/!\ PAGE FAULT !!! /!\
Frame 0 has been choosen
(frame 0: 4) (frame 1: 2) (frame 2: 3)

Page 3 referenced
(frame 0: 4) (frame 1: 2) (frame 2: 3)

Page 4 referenced
(frame 0: 4) (frame 1: 2) (frame 2: 3)

Page 4 referenced
(frame 0: 4) (frame 1: 2) (frame 2: 3)

4/7 ~ 57.142857%
```

Par défaut la libmem est en mode « verbeux », pour desactiver le mode, il faut commenter dans le makefile la ligne :

```
CFLAGS=-D_DEBUG_
```

4. POUR INSTALLER LA LIBRAIRIE CHEZ SOI

La librairie est disponible sous forme d'archive compressée sur la page web de la licence :
`http://www-licence.ufr-info-p6.jussieu.fr/lmd/licence/2005/ue/sys-2006fev/libmem`

Pour la décompresser il suffit de taper sur votre machine :

```
$ tar xvfz libmem.tgz
```

Un répertoire libmem est créé. Il suffit de compiler la librairie et executer l'exemple :

```
$ cd libmem
$ make
$ bin/main < bench
```

Merci à Mathieu Valéro pour avoir développé la libmem
Pour tout commentaire, ou rapport de " bug " : `Pierre.Sens@lip6.fr`