TP Temps Réel

Ludovic Saint-Bauzel <saintbauzel@isir.upmc.fr> and Jérôme Pouiller <jezz@sysmic.org>

Polytech Paris UPMC - 2020

Table des matières

1 Avant de commencer

Cette section est purement informative. Il n'y a pas de questions à l'intérieur.

1.1 Documentation

1.1.1 Références des pages de man

Comme le veut l'usage, les références des pages de man sont donnés avec le numéro de section entre parenthèses. Ainsi, *wait(2)* signifie que vous pouvez accéder à la documentation avec la commande `man 2 wait`. Si vous omettez le numéro de section, `man` recherchera la première page portant le nom `wait` (ce qui fonctionnera dans 95% des cas). Le numéro de section vous donne aussi une indication sur le type de documentation que vous aller trouver. D'après *man(1)*, voici les différentes sections :

1. Executable programs or shell commands
2. System calls (functions provided by the kernel)
3. Library calls (functions within program libraries)
4. Special files (usually found in /dev)
5. File formats and conventions (e.g. /etc/passwd)
6. Games
7. Miscellaneous (including macro packages and conventions), e.g. man(7), groff(7)
8. System administration commands (usually only for root)
9. Kernel routines [Non standard]

Vous pouvez trouver une version en ligne de la plupart des pages de man sur The Friendly Manual (https://www.gnu.org/manual/manual.fr.html).

Vous trouverez en particulier les pages relatives à l'API du noyau Linux qui ne sont pas présentes en local sur vos machine.

Prenez néanmoins garde aux éventuelles différences de versions d'API (très rare et quasiment exclusivement sur les pages relatives à l'ABI du noyau).

1.1.2 Standards

Vous subissez aujourd'hui tout le poids de l'histoire de la norme Posix (30 ans d'histoire de l'informatique). Ne soyez pas étonnés de trouver de multiples interfaces pour une même fonctionnalité. C'est particulièrement vrai sur les fonction relatives au temps.

Nous basons ces exercice sur la norme POSIX.1-2001 que vous pourrez trouver sur http://www.unix-systems.org/version3/online.html

Vous trouverez plus d'informations sur l'histoire des différentes normes existantes sur *standards(7)*.

1.2 Erreurs

Pour indiquer quelle erreur s'est produite, la plupart des fonctions Posix utilisent :
- soit leur valeur de retour
- soit une variable globale de type `int` nommée `errno`

Utilisez *strerror(3)* pour obtenir le message d'erreur équivalent au code de retour.

Exemple :

```
1 if (err = pthread_create(&tid, NULL, f, NULL))
    printf("Task create: %s", strerror(err));
  if (-1 == pause())
    printf("Pause: %m");
```

1.3 Compilation

Pour la plupart des exercices, vous aurez besoin de linker [1] avec `-lpthread` et `-lrt`.

Par ailleurs, nous vous conseillons fortement de compiler avec l'option `-Wall` de `gcc` et d'utiliser un système de Makefile.

1. C'est à dire la phase de compilation où vous liez tous vous fichiers `.o` ensemble

2 Tricky conditions

Nous avons un problème avec le code ci-dessous :

```c
/*
 * Creation date: 2010-01-03 11:22:08+01:00
 * Licence: GPL
 * Main authors:
 *   - éôJrme Pouiller <jerome@sysmic.org>
 *
 * Sample use of conditions.
 *
 */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>

static pthread_cond_t c = PTHREAD_COND_INITIALIZER;
static pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
static int count = 0;

void *sender(void *arg) {
    int i;

    for (i = 0; i < 10; i++) {
        count++;
        printf("%08lX: Increment count: count=%d\n", pthread_self(), count);
        if (count == 3) {
            pthread_cond_signal(&c);
            printf("%08lX: Sent signal. count=%d\n", pthread_self(), count);
        }
    }
    return NULL;
}

void *receiver(void *arg) {
    printf("%08lX: Begin waiting\n", pthread_self());
    pthread_cond_wait(&c, &m);
    printf("%08lX: Received signal. count=%d\n", pthread_self(), count);
    return NULL;
}

int main(int argc, char **argv) {
    pthread_t id[2];

    pthread_mutex_lock(&m);
    pthread_create(&id[0], NULL, sender, NULL);
    pthread_create(&id[1], NULL, receiver, NULL);

    pthread_join(id[0], NULL);
    pthread_join(id[1], NULL);
    pthread_mutex_unlock(&m);

    return 0;
}
```

De temps en temps, le signal envoyé par la thread *sender* n'est pas recu par la thread *receiver*.

Question 2.1. *Exacerber le problème en plaçant une temporisation dans le code.*

Remarque: La question peut se faire en 2 lignes.

Documentation utile: *sleep(3)*

Question 2.2. *Utilisez le mutex* m *de manière appropriée pour résoudre le problème.*

Remarque: La question peut se faire en 4 lignes.

Documentation utile: *pthread_cond_wait(3),pthread_mutex_lock(3), pthread_mutex_unlock(3)*

Question 2.3. *Modifiez votre programme de façon à lancer 3 threads* receiver *et une thread* sender. *Puis, modifiez votre programme de manière à ce que la thread* sender *débloque correctement les 3 threads* receiver.

Remarque: Nous attirons votre attention sur ce paragraphe de la norme Posix :

> The effect of using more than one mutex for concurrent `pthread_cond_wait()` or `pthread_cond_timedwait()` operations on the same condition variable is undefined; that is, a condition variable becomes bound to a unique mutex when a thread waits on the condition variable, and this (dynamic) binding ends when thewait returns.

Remarque: Pour résoudre élégamment ce problème, nous aurions aimé que `pthread_cond_wait` prenne en paramètre un sémaphore plutôt qu'un mutex. Nous allons donc devoir ajouter un semaphore à notre programme pour palier ce manque.

Remarque: La question peut se faire en environ 20 lignes.

Documentation utile: *pthread_cond_broadcast(3), pthread_cond_wait(3), pthread_mutex_lock(3), pthread_mutex_unlock(3), sem_init(3), sem_wait(3), sem_post(3)*

---

3 Protection de ressources

Nous avons besoin d'envoyer 4 messages vers un serveur distant. Nous utilisons le code ci-dessous :

```c
#define SZ 7

void *task(void *arg) {
    char *buf = (char *) arg;

    send(buf, SZ);
    return NULL;
}

int main(int argc, char **argv) {
    pthread_t id[4];
    int i;
    char msgs[4][SZ] = { "123456", "ABCDEF", "abcdef", "[{()}]" };

    for (i = 0; i < 4; i++)
        pthread_create(&id[i], NULL, task, msgs[i]);

    for (i = 0; i < 4; i++)
        pthread_join(id[i], NULL);

    printf("\n");
    return 0;
}
```

Dans le cadre de notre étude, nous n'avons pas d'accès réel au serveur, nous avons donc écrit une fonction de simulation de l'envoie des donnée :

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void send_char(char data) {
    double time = (double) random() / RAND_MAX * 8;
    struct timespec t;
    t.tv_sec = time / 1000;
    t.tv_nsec = (int) (time * 1000000) % 1000000000;
    nanosleep(&t, NULL);
    write(1, &data, 1);
}

void send(char *data, size_t len) {
    unsigned i;

    for (i = 0; i < len; i++)
        send_char(data[i]);
}
```

Malheureusement, il semblerait qu'il y ait un problème car les messages reçu par le serveurs ne sont pas cohérents :

```
Aa[1b{cB2(deC3)}DE45Ff]6
```

Or, pour que le serveur puisse interpréter les données, il ne faut pas que les messages des *tâches* concurrentes soit enchevêtrés.

On propose d'emballer `send` dans une nouvelle fonction alternative permettant de régler le problème :

```c
int safe_send(char *data, size_t len);
```

Question 3.1. *Implémentez* `safe_send`.

Remarque: La question peut se faire en moins de 10 lignes.

Documentation utile: *pthread_mutex_lock(3), pthread_mutex_unlock(3)*

La fonction `safe_send` résout le problème de cohérence, mais cause des problèmes de latences dans la fonction `task`. Dans certains cas, celle-ci s'exécute beaucoup plus lentement qu'avant.

Question 3.2. *Écrivez un scénario de test permettant de mettre en évidence et de mesurer le phénomène de latence.*

Remarque: La question peut se faire en moins de 20 lignes.

Documentation utile: *clock_gettime(3), memset(3)*

Nous allons essayer de résoudre le problème de latence. Pour cela, nous allons créer une nouvelle *tâche* permettant d'envoyer les données de manière asynchrone.

Afin de transmettre à la *tâche* les données à envoyer, nous devons mettre en place une structure de données adéquate. Nous proposons d'utiliser une file (FIFO) sous la forme d'un *buffer circulaire*.

Enfin, notre *buffer circulaire* nécessitera un mécanisme de réveil de la *tâche* d'envoi lorsque des données sont disponible dans le *buffer*.

Question 3.3. *Implémentez cette solution.*

Remarque: La question peut se faire en moins de 40 lignes.

Documentation utile: *pthread_create(3), pthread_mutex_lock(3), pthread_mutex_unlock(3), pthread_cond_wait(3), pthread_cond_sign*

4 Tâches et sémaphores Xenomai

Afin de faire ce TP, nous vous avons fourni un Linux avec un noyau patché avec Xenomai. Ce patch nous permet d'avoir un système temps réel et profiter d'une API spécialisée pour le temps réel. Notez que nous utiliserons exclusivement l'API Native durant ce TP.

Reportez-vous à l'annexe A pour les informations concernant l'usage de la bibliothèque Xenomai.

Question 4.1. *Écrivez un programme avec N taches, N étant une constante définie à la compilation. Chacune des taches contiendra une boucle incrémentant et affichant un compteur. Utilisez des sémaphores pour synchroniser les boucles. Vous devriez obtenir le comportement suivant ($N = 4$) :*

```
$ ./concurence
Task: 0, count: 0
Task: 1, count: 0
Task: 2, count: 0
Task: 3, count: 0
Task: 0, count: 1
Task: 1, count: 1
Task: 2, count: 1
Task: 3, count: 1
Task: 0, count: 2
Task: 1, count: 2
Task: 2, count: 2
[...]
```

Documentation utile: *rt_task_create, rt_task_start, rt_task_delete, rt_sem_create, rt_sem_delete, rt_sem_p, rt_sem_v, mlockall(2)*

Question 4.2. *Vérifiez le bon fonctionnement de votre programme avec $N = 1000$.*

Question 4.3. *Modifiez votre code pour créer un* dead lock *sur toutes les taches.*

Remarque: Une ligne à modifier suffit généralement.

Question 4.4. *A l'aide la documentation de* rt_mutex_acquire*, expliquez pourquoi cet exercice serait plus complexe à faire avec les mutex de la skin Native.*

Question 4.5. *Fixez N à 4. Ajoutez une cinquième tache. La cinquième tache devra monopoliser processeur (ie :* `for(;;);`*). En modifiant les priorités des taches, produisez une inversion de priorité.*

Documentation utile: *rt_task_create*

Question 4.6. *Modifiez votre programme pour en faire un module noyau. Limitez votre boucle à 100 itérations afin de ne pas mettre en péril la stabilité de votre machine.*

---

5 Libtimer

On se propose d'implémenter une bibliothèque `libtimer` permettant de lancer des *tâches* à une date fixe ou après une certaine période. Le fonctionnement de cette bibliothèque est décrit dans `timer.h` :

```c
/*
 * Author: éêJrme Pouiller <jerome@sysmic.fr>
 * Created: Fri Dec 11 22:57:13 CET 2009
 * Licence: GPL
 *
 * Define interface for timer services.
 */

#ifndef MYTIMER_H
#define MYTIMER_H
#include <pthread.h>
#include <time.h>

/* Helper to define a task */
typedef void (*task_t)();

/* INSERT YOUR mytimer_s DEFINITION HERE */

/* Contains all needed data */
typedef struct mytimer_s mytimer_t;

/*
 * All following functions return
 *  - 0 on success
 *  - another value on error
 */

/*
 * Initialize new timer instance.
 * Instance is returned using parameter out.
 * User have to call this function first
 */
int mytimer_run(mytimer_t *out);

/* Stop timer instance */
int mytimer_stop(mytimer_t *mt);

/* Add a task to timer. Task will be executed at time tp */
int mytimer_add_absolute(mytimer_t *mt, task_t t, const struct timespec *tp);

/* Add a task to timer. Task will be executed in ms milliseconds */
int mytimer_add_msecond(mytimer_t *mt, task_t t, unsigned long ms);

/* Add a task to timer. Task will be executed in s seconds */
int mytimer_add_second(mytimer_t *mt, task_t t, unsigned long sec);

/* Add a task to timer. Task will be executed in h hours */
int mytimer_add_hour(mytimer_t *mt, task_t t, unsigned long h);

/* Remove first occurence of task t from timer. */
int mytimer_remove(mytimer_t *mt, task_t t);

#endif /* MYTIMER_H */
```

— La structure `mytimer_s` (que vous définirez) contient tout le contexte d'exécution.
— Les *tâches* à lancer sont décrite par le type `task_t`.
— L'utilisateur de la bibliothèque commencera par un appel à `mytimer_run`.
— Il ajoutera les *tâches* avec les fonctions `mytimer_add`.

— `mytimer_add_absolute` exécute la *tâche* à une heure fixe donnée tandis que les autres fonctions `mytimer_add` exécutent la *tâche* à un temps relatif à l'instant présent. Par exemple, `mytimer_add_second(s, t, 3)` exécutera `t` 3 secondes après l'appel à cette fonction.
— Toutes les fonctions de `libtimer` retournent un entier. Il s'agit d'un code d'erreur. Vous êtes libre dans le choix des codes de retour.

On considère que les *tâches* sont exemptes de bugs et qu'elles ne tentent pas de mettre en péril la stabilité de votre bibliothèque et du système.

`libtimer` tentera d'être le plus précis possible dans l'heure de lancement des *tâches*. Néanmoins, on ignorera le biais introduit par le système d'exploitation et les autres processus.

Afin de vous aider, vous trouverez une implémentation de liste chainée dans `list.h` et `list.c` :

```c
/*
 * Author: éôJrme Pouiller <jerome@sysmic.fr>
 * Created: Fri Dec 11 22:57:54 CET 2009
 * Licence: GPL
 *
 */
#ifndef LIST_H
#define LIST_H
#include <stdlib.h>
#include <stdbool.h>

typedef struct list_item_s {
    /* ADD YOUR MEMBERS HERE */
    struct list_item_s *next;
} list_item_t;

#define EMPTY_LIST NULL
#define NEXT(s) (s)->next
#define FOREACH(I, LIST) for (I = LIST; I; I = NEXT(I))

bool list_isEmpty(list_item_t *l);

int list_size(list_item_t *list);

/* Add an element. Don't forget to allocate it before */
void list_add(list_item_t **list, list_item_t *item);

/* Remove an element. Return removed element. Don't forget to free it AFTER removing */
list_item_t *list_remove(list_item_t **list, list_item_t *item);

#endif /* LIST_H */
```

```c
/*
 * Author: éôJrme Pouiller <jerome@sysmic.fr>
 * Created: Fri Dec 11 22:57:54 CET 2009
 * Licence: GPL
 *
 */
#include "list.h"

bool list_isEmpty(list_item_t *l) {
    return (l == EMPTY_LIST);
}

int list_size(list_item_t *list) {
    int count = 0;
    list_item_t *i;
```

```
    FOREACH(i, list)
        count++;
    return count;
}

/* Add an element. Don't forget to allocate it before */
void list_add(list_item_t **list, list_item_t *item) {
    item->next = *list;
    *list = item;
}

/* Remove an element. Return removed element. Don't forget to free it AFTER removing */
list_item_t *list_remove(list_item_t **list, list_item_t *item) {
    list_item_t *i;

    if (*list == item) {
        *list = (*list)->next;
        return item;
    }
    FOREACH(i, *list)
        if (i->next == item) {
            i->next = item->next;
            return item;
        }
    // Nothing found
    return NULL;
}
```

5.1 Version synchrone

On considère pour l'instant que le temps d'exécution des *tâches* est nul.

Question 5.1. *Implémentez* `libtimer` *tel que décrit dans le fichier d'entête* `timer.h`.

Documentation utile: *pthread_cond_timedwait(3), pthread_cond_wait(3), pthread_cond_signal(3), clock_gettime(2), sleep(3), pthread_create(3), pthread_mutex_lock(3), pthread_mutex_unlock(3), pthread_mutex_init(3), pthread_cond_init(3), pthread_cancel(3), ctime_r(3)*

5.2 Version asynchrone

Nous avons supposé dans que le temps d'exécution des *tâches* étaient nul. Nous souhaitons maintenant écrire la version finale de la bibliothèque exécutant les *tâches* de manière asynchrone.

Question 5.2. *Modifiez votre bibliothèque de manière à exécuter les* tâches *de manière concurrente.*

Documentation utile: *pthread_attr_setdetachstate(3), pthread_create(3), pthread_attr_init(3)*

A Écriture des programmes Xenomai

A.1 Documentation de Xenomai

Vous pourrez trouvez la documentation de l'API de Xenomai dans `/usr/share/doc/xenomai-doc/html/api` ou en ligne sur `http://www.xenomai.org/documentation/xenomai-3/xeno3prm/index.html`.

Vous pouvez trouver les headers à inclure en explorant l'onglet *files*.

A.2 Compilation

Les commandes `xeno-config --xeno-cflags` et `xeno-config --xeno-ldflags` permettent d'obtenir respectivement les options de compilation et les options de link à utiliser pour que votre programme fonctionne avec Xenomai. Vous devrez de plus préciser que vous utilisez la skin Native avec l'option `-lnative`.

Par ailleurs, nous vous conseillons fortement de compiler avec l'option `-Wall` de `gcc` et d'utiliser un système de Makefile.

N'oubliez pas que dans un Makefile, il faut utiliser le mot-clef `shell` pour lancer une commande. Par exemple :

```
bin: file.o
	gcc -Wall $(shell xeno-config --xeno-ldflags) -lnative $< -o $@
file.o: file.c
	gcc -Wall $(shell xeno-config --xeno-cflags) -c $< -o $@
```

Nous vous fournissons un fichier `Makefile` reprennant les options à utiliser.

A.3 Vérouillage de la mémoire

En cas de nécessité, le gestionnaire de mémoire du noyau Linux peut-être amené à placer des pages de mémoire virtuelle sur le disque dur (processus de *swaping*). Lorsqu'une de ces pages de mémoire est nécessaire, elle doit être replacée en mémoire avant d'être utilisée. Cela crée une latence qui peut poser problème dans le cas d'application temps réelles.

Un appel à la fonction *mlockall(1)* permet de demander au noyau de vérouiller les pages de mémoire en mémoire physique :

```
mlockall(MCL_CURRENT | MCL_FUTURE);
```

La mémoire doit être vérouillée avant tout appel à l'API Xenomai.

A.4 A propos des noms des objets Xenomai

Il est possible dans Xenomai d'associer chaque objet à un nom unique sur le système. Cela permet de *binder* des objets entre des processus ou entre des domaines d'exécution.

L'utilisation des noms oblige à être rigoureux et à correctement détruire les objets avant de sortir des processus. Les instances des objets resteront en mémoire.

Par conséquent, nous vous conseillons de ne pas utiliser cette fonctionnalité des les exercices suivants et à passer un pointeur `NULL` comme nom.

A.5 Gestion des Erreurs

La plupart des fonctions Xenomai retournent une valeur négative pour indiquer qu'une erreur s'est produite. Vous pouvez utiliser *strerror(3)* pour obtenir le message d'erreur équivalent.

Exemple classique :

```
if (err = rt_task_start(t, f, NULL))
  printf("Task start: %s", strerror(-err));
```

A.6 Exécution

Vous aurez besoin d'exécuter les programmes que vous créerez avec les droits *root*. Nous vous déconseillons l'utilisation de *sudo(1)* pour lancer vos programmes car `Ctrl+C` sera moins réactif.

B Modalités de rendu

Votre rendu sera effectué sur la plateforme Moodle (https://moodle-sciences.upmc.fr/moodle-2019).

Vous devez rendre votre travail dans une archive gzippée. L'archive devra se nommer `TR_ROB5-nom.tar.gz` et se décompresser dans le répertoire du même nom.

Votre archives ne doit pas contenir de fichiers binaires. Il ne doit contenir que vos sources (fichiers sources, fichiers entêtes, Makefile, etc...).

Exemple de création de rendu :

```
make clean
cd ..
tar cvzf TR_ROB5-pouiller.tar.gz TR_ROB5-pouiller
```

Vous devrez envoyer déposer une version **à la fin de la séance** et vous pouvez faire évoluer ce document jusqu'au début du prochain TP.