Final Exam

Team18

112062519 廖思愷 112062636 游竣量 111065547 游述宇

1. 將 SIGINT、SIGTERM、SIGUSR1 從執行 main function 的 thread 屏蔽掉

```c
int main(int argc, char *argv[])
{
    pthread_t tid1, tid2, tid3;
    int err;

    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    sigaddset(&set, SIGTERM);
    sigaddset(&set, SIGUSR1);
    pthread_sigmask(SIG_BLOCK, &set, NULL);
```

依序創建 thread T1、T2、T3

```c
if ((err = pthread_create(&tid1, NULL, &T1, NULL)) != 0)
{
    perror("pthread_create error");
    exit(1);
}

if ((err = pthread_create(&tid2, NULL, &T2, NULL)) != 0)
{
    perror("pthread_create error");
    exit(1);
}

if ((err = pthread_create(&tid3, NULL, &T3, NULL)) != 0)
{
    perror("pthread_create error");
    exit(1);
}
```

等待 T1、T2、T3 執行完成

```c
pthread_join(tid1, NULL);
pthread_join(tid2, NULL);
pthread_join(tid3, NULL);
```

T1 thread：將 SIGINT 的屏蔽解除，設置 sigint_handler 為 SIGINT 的 handler，並等待 signal 發生

```c
void *T1(void *arg)
{
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGINT);
    pthread_sigmask(SIG_UNBLOCK, &set, NULL);

    if (signal(SIGINT, sigint_handler) == SIG_ERR)
    {
        perror("signal error");
        exit(1);
    }

    pause();
}
```

T2 thread：將 SIGTERM 的屏蔽解除，設置 sigterm_handler 為 SIGTERM 的handler，並等待 signal 發生

```c
void *T2(void *arg)
{
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGTERM);
    pthread_sigmask(SIG_UNBLOCK, &set, NULL);

    if (signal(SIGTERM, sigterm_handler) == SIG_ERR)
    {
        perror("signal error");
        exit(1);
    }

    pause();
}
```

T3 thread：將 SIGUSR1 的屏蔽解除，設置 sigusr1_handler 為 SIGUSR1 的 handler，並等待 signal 發生

```c
void *T3(void *arg)
{
    sigset_t set;
    sigemptyset(&set);
    sigaddset(&set, SIGUSR1);
    pthread_sigmask(SIG_UNBLOCK, &set, NULL);

    if (signal(SIGUSR1, sigusr1_handler) == SIG_ERR)
    {
        perror("signal error");
        exit(1);
    }

    pause();
}
```

三個 dignal handler，分別會印出：
T1 handling SIGINT
T2 handling SIGTERM
T3 handling SIGUSR1

```c
void sigint_handler(int signo)
{
    printf("T1 handling SIGINT\n");
}

void sigterm_handler(int signo)
{
    printf("T2 handling SIGTERM\n");
}

void sigusr1_handler(int signo)
{
    printf("T3 handling SIGUSR1\n");
}
```

使用方法：
make 後，執行
./q1 &
程式會在後台運行
對程式發送 kill -USR1 %1，會 print 出 T3 handling SIGUSR1
對程式發送 kill -INT %1，會 print 出 T1 handling SIGINT
對程式發送 kill -TERM %1，會 print 出 T2 handling SIGTERM
三個 signal 都收到後，process 會終止

執行結果：

```
team18@:~/Kyle_test/final/q1 $ ./q1 &
team18@:~/Kyle_test/final/q1 $ kill -USR1 %1
team18@:~/Kyle_test/final/q1 $ T3 handling SIGUSR1

team18@:~/Kyle_test/final/q1 $ kill -INT %1
T1 handling SIGINT
team18@:~/Kyle_test/final/q1 $ kill -TERM %1
T2 handling SIGTERM
team18@:~/Kyle_test/final/q1 $ jobs
[1]   Done                    ./q1
```

2. 在 sleep_us 函數中，使用 select 來實現等待特定微秒秒數的睡眠：

```c
void sleep_us(int microseconds) {
    struct timeval tv;
    tv.tv_sec = microseconds / 1000000;
    tv.tv_usec = microseconds % 1000000;
    select(0, NULL, NULL, NULL, &tv);
}
```

在 main 中，紀錄睡眠前的時間以及睡眠後的時間，來確認 sleep_us 函數有 return 微秒時間，並輸出微秒睡眠時間：

```c
int main(int argc, char *argv[]) {
    if (argc != 2) {
        printf("error input\n");
        return 1;
    }
    int time_microseconds = atoi(argv[1]);
    struct timeval start, end;

    gettimeofday(&start, NULL);
    sleep_us(time_microseconds);
    gettimeofday(&end, NULL);

    long seconds = end.tv_sec - start.tv_sec;
    long sleep_usectime= ((seconds * 1000000) + end.tv_usec) - (start.tv_usec);

    printf("Sleep time: %ld us\n", sleep_usectime);
    return 0;
}
```

使用方法：
make後，執行 ./q2 <microsecond_time>，將 <microsecond_time> 替換為希望睡眠的微秒秒數，便會睡眠該微秒秒數後，輸出睡眠的時間。

輸出結果：

```
team18@:~/Sabrina_test/Final $ ./q2 1000000
Sleep time: 1005087 us
```

3. Use a **single timer** (either alarm or setitimer) to implement a set of functions that enables a process to **set and clear any number of timers**. Please print “Alarm!” when the alarm signal trigger.

```c
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <unistd.h>

// 定義一個結構體來表示定時器節點
typedef struct TimerNode {
 struct timeval end_time; // 定時器結束時間
 struct TimerNode *next;  // 指向下一個定時器節點的指針
} TimerNode;

TimerNode *head = NULL; // 定義一個指向定時器鏈表頭部的全局指針

// 釋放所有定時器節點的內存
void freeTimers() {
 while (head) {
   TimerNode *temp = head;
   head = head->next;
   free(temp);
 }
}

// 插入一個新的定時器到鏈表中，並保持鏈表按結束時間排序
void insertTimer(struct timeval end_time) {
 TimerNode *node = (TimerNode *)malloc(sizeof(TimerNode));
 node->end_time = end_time;
 node->next = NULL;

 // 如果鏈表為空或新定時器的結束時間最早，則將其插入到鏈表頭部
 if (!head || timercmp(&end_time, &head->end_time, <)) {
   node->next = head;
   head = node;
```

```
    alarm(end_time.tv_sec - time(NULL)); // 設置新的 alarm
  } else {
    // 否則，遍歷鏈表，找到適合的插入位置
    TimerNode *current = head;
    while (current->next && timercmp(&end_time, &current->next->end_time, >))
{
      current = current->next;
    }
    node->next = current->next;
    current->next = node;
  }
}

// 當定時器信號被觸發時執行的處理函數
void alarmHandler(int sig) {
  if (head) {
    TimerNode *temp = head;
    head = head->next;
    free(temp);

    // 如果鏈表中還有其他定時器，則設置下一個定時器
    if (head) {
      struct timeval now;
      gettimeofday(&now, NULL);
      alarm(head->end_time.tv_sec - now.tv_sec);
    }
  }
  printf("Alarm!\n");
}

// 設置一個新的定時器
void setAlarm(int sec) {
  struct timeval now, end_time;
  gettimeofday(&now, NULL);
  end_time.tv_sec = now.tv_sec + sec;
  end_time.tv_usec = now.tv_usec;
```

```
72.
73.  insertTimer(end_time); // 將新定時器插入到鏈表中
74. }
75.
76. // 清除所有定時器
77. void clearAlarm() {
78.  freeTimers(); // 釋放所有定時器節點的內存
79.  alarm(0);      // 取消當前的 alarm
80. }
81.
82. int main() {
83.  signal(SIGALRM, alarmHandler); // 設置信號處理函數
84.
85.  // 測試設置和清除定時器的邏輯
86.  setAlarm(2); // 在程式開始後2秒設置定時器
87.  sleep(1);
88.  setAlarm(6); // 在程式開始後1秒設置定時器，將在總共7秒後觸發
89.  sleep(1);
90.  setAlarm(3); // 在程式開始後2秒設置定時器，將在總共5秒後觸發
91.  sleep(4);
92.  clearAlarm(); // 在程式開始後6秒清除所有定時器
93.
94.  return 0;
95. }
96.
```

輸出結果：

```
~/Desktop/碩一上/UNIX/Advanced-UNIX-Programming_team18/final  main  ./
q3
Alarm!
Alarm!
```

使用一個排序鏈表來管理多個定時器。每次設置新定時器時，會計算其結束時間，並按順序插入鏈表中，如果其結束時間早於當前最早定時器結束時間則用新 alarm 取代，而原本定時器往後排隊。當 alarm 到期時（即當前最早的定時器觸發），alarmHandler 會被調用，刪除鏈表中的第一個節點（即已觸發的定時器），然後根據下一個節點重新設置 alarm。如果需要取消所有定時器，clearAlarm 會被調用，它將釋放鏈表中所有節點的內存並取消當前的 alarm。