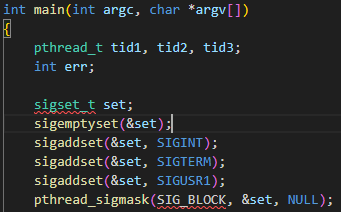
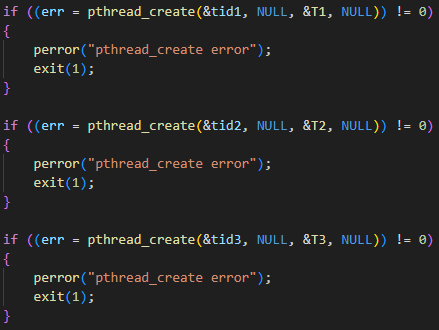
Final Exam

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



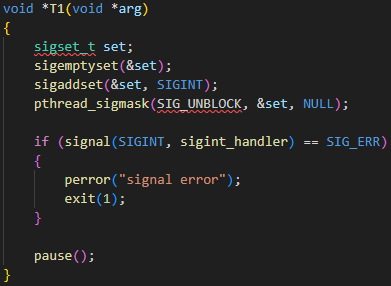
依序創建 thread T1、T2、T3



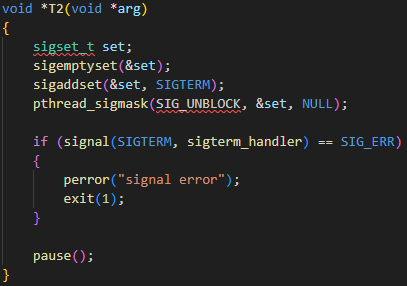
等待 T1、T2、T3 執行完成



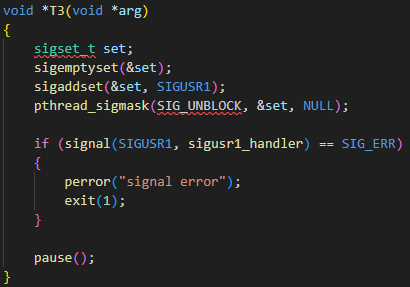
T1 thread：將 SIGINT 的屏蔽解除，設置 sigint\_handler 為 SIGINT 的 handler，並等待 signal 發生



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



T3 thread：將 SIGUSR1 的屏蔽解除，設置 sigusr1\_handler 為 SIGUSR1 的 handler，並等待 signal 發生

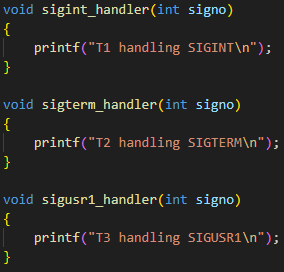


三個 dignal handler，分別會印出：

T1 handling SIGINT

T2 handling SIGTERM

T3 handling SIGUSR1



使用方法：

make 後，執行

./q1 &

程式會在後台運行

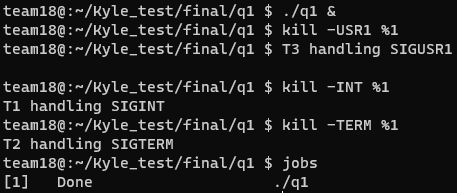
對程式發送 kill -USR1 %1，會 print 出 T3 handling SIGUSR1

對程式發送 kill -INT %1，會 print 出 T1 handling SIGINT

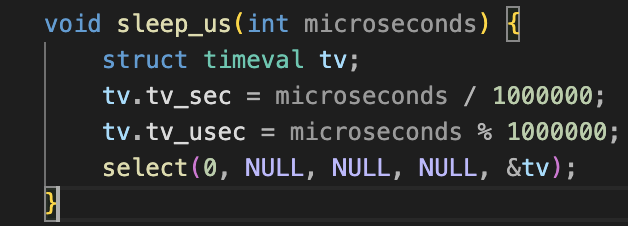
對程式發送 kill -TERM %1，會 print 出 T2 handling SIGTERM

三個 signal 都收到後，process 會終止

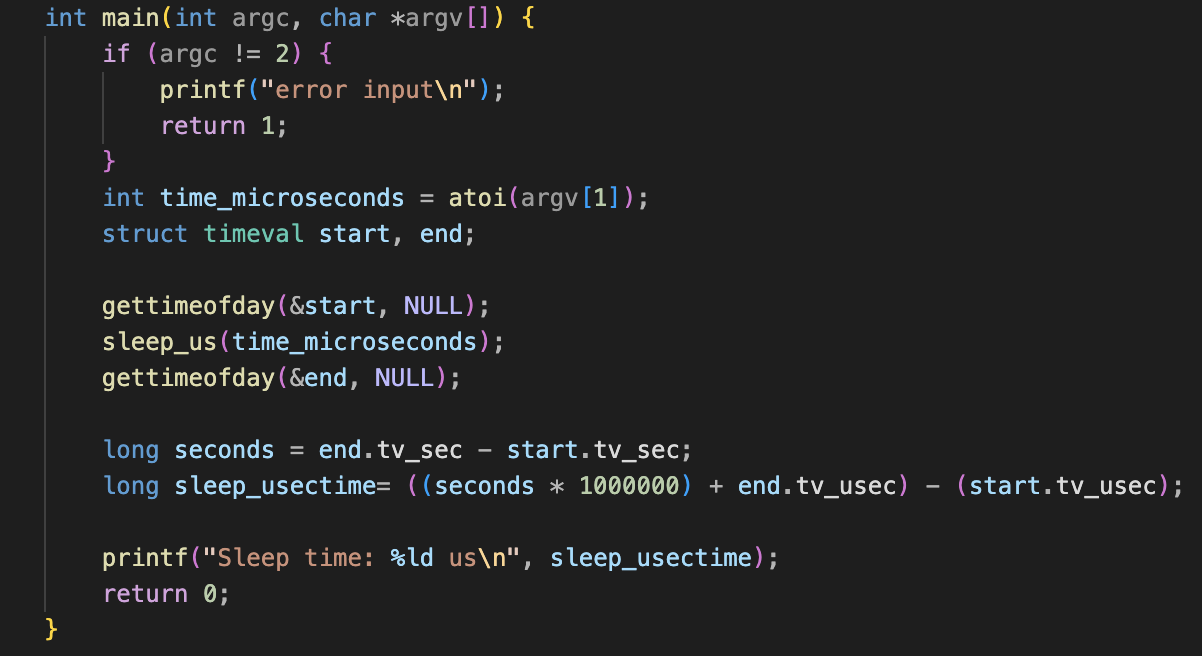
執行結果：



1. 在 sleep\_us 函數中，使用 select 來實現等待特定微秒秒數的睡眠：



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：



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：

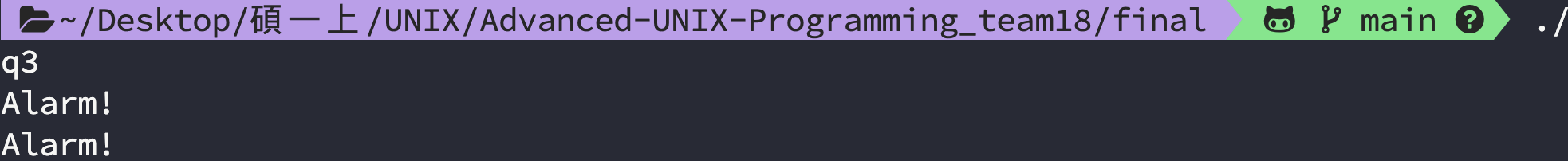


1. Use a **single timer** (either alarm or setitimer) to implement a set of functions that

enables a process to **set and clear any number of timers**. Please print “Alarm!” when the alarm signal trigger.

1. #include <signal.h>
2. #include <stdio.h>
3. #include <stdlib.h>
4. #include <sys/time.h>
5. #include <unistd.h>
6. // 定義一個結構體來表示定時器節點
7. typedef struct *TimerNode* {
8. struct *timeval* end\_time; // 定時器結束時間
9. struct *TimerNode* \*next; // 指向下一個定時器節點的指針
10. } *TimerNode*;
11. *TimerNode* \*head = NULL; // 定義一個指向定時器鏈表頭部的全局指針
12. // 釋放所有定時器節點的內存
13. void freeTimers() {
14. while (head) {
15. *TimerNode* \*temp = head;
16. head = head->next;
17. free(temp);
18. }
19. }
20. // 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
21. void insertTimer(struct *timeval* *end\_time*) {
22. *TimerNode* \*node = (*TimerNode* \*)malloc(sizeof(*TimerNode*));
23. node->end\_time = *end\_time*;
24. node->next = NULL;
25. // 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
26. if (!head || timercmp(&*end\_time*, &head->end\_time, <)) {
27. node->next = head;
28. head = node;
29. alarm(*end\_time*.tv\_sec - time(NULL)); // 設置新的 alarm
30. } else {
31. // 否則，遍歷鏈表，找到適合的插入位置
32. *TimerNode* \*current = head;
33. while (current->next && timercmp(&*end\_time*, &current->next->end\_time, >)) {
34. current = current->next;
35. }
36. node->next = current->next;
37. current->next = node;
38. }
39. }
40. // 當定時器信號被觸發時執行的處理函數
41. void alarmHandler(int *sig*) {
42. if (head) {
43. *TimerNode* \*temp = head;
44. head = head->next;
45. free(temp);
46. // 如果鏈表中還有其他定時器，則設置下一個定時器
47. if (head) {
48. struct *timeval* now;
49. gettimeofday(&now, NULL);
50. alarm(head->end\_time.tv\_sec - now.tv\_sec);
51. }
52. }
53. printf("Alarm!\n");
54. }
55. // 設置一個新的定時器
56. void setAlarm(int *sec*) {
57. struct *timeval* now, end\_time;
58. gettimeofday(&now, NULL);
59. end\_time.tv\_sec = now.tv\_sec + *sec*;
60. end\_time.tv\_usec = now.tv\_usec;
61. insertTimer(end\_time); // 將新定時器插入到鏈表中
62. }
63. // 清除所有定時器
64. void clearAlarm() {
65. freeTimers(); // 釋放所有定時器節點的內存
66. alarm(0); // 取消當前的 alarm
67. }
68. int main() {
69. signal(SIGALRM, alarmHandler); // 設置信號處理函數
70. // 測試設置和清除定時器的邏輯
71. setAlarm(2); // 在程式開始後2秒設置定時器
72. sleep(1);
73. setAlarm(6); // 在程式開始後1秒設置定時器，將在總共7秒後觸發
74. sleep(1);
75. setAlarm(3); // 在程式開始後2秒設置定時器，將在總共5秒後觸發
76. sleep(4);
77. clearAlarm(); // 在程式開始後6秒清除所有定時器
78. return 0;
79. }

輸出結果：



**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。