CS471: Operating System Concepts
Fall 2006
(Lecture: TRW 11:25-12:40 AM)

Homework #1
Points: 20
Due: September 7, 2006
Solution

Question 1 [Points 3] Problem 4.4, page 147 (7th edition)

Solution:

Shared variables: heap memory and global variables
Private variables: register values and stack memory

Question 2 [Points 4] Modify and run **fork1.c** program so that three generations of processes are created (instead of 2 in fork1.c). The original process prints its pid and its child pid. The child process prints its pid, its parent pid, and its child pid. The grandchild prints its pid and its parent id. Show the source listing and program output.

Solution:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int  main (int argvc, char **argv)
{
   pid_t pid;
   pid = fork();

   if (pid == 0) {

   pid_t pid1;
   pid1 = fork();

   if (pid1 == 0) {
      printf("I am a grand child\n");
      printf("grand child  pid is: %d\n", getpid());
      printf("pid of grand child parent is: %d\n", getppid());
   }
   else {
      printf("I am a parent of grand child \n");
      printf("pid of grand child parent is: %d\n", getpid());
      printf("grand child pid is: %d\n", pid1);
      /*printf("original parent pid is: %d \n", getppid());*/
       }
   }
   else {
        printf("I am a original parent\n");
        printf("original parent pid is: %d\n", getpid());
        printf("my child pid is: %d\n", pid);
    }

    /* This code executes in both processes */
    printf("exiting ... %d\n", getpid());
    exit(0);
}
```

Output:

```
I am a grand child
grand child  pid is: 12553
pid of grand child parent is: 12552
exiting ... 12553
I am a original parent
original parent pid is: 12551
my child pid is: 12552
exiting ... 12551
I am a parent of grand child
pid of grand child parent is: 12552
grand child pid is: 12553
exiting ... 12552
```

Question 3 [Points 5] Modify and run **thread1.c** in the following way. There is an array of 10 elements defined in the program. The elements of the array are:
[20 18 16 14 12 10 8 6 4 2]. Thread 1 adds the first two elements (i.e., 20, 18), Thread 2 adds the next two elements (i.e., 16, 14), …, Thread 5 adds the last two elements (4, 2). Finally, the sum of all the 10 elements is printed by the program.

Solution:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     5
int counter=0;
int arrayElements[]={20,18,16,14,12,10,8,6,4,2};
int sum=0;
int temp=0;
int iteration=0;
void *PrintHello(void *threadid)
{

   sum=sum+arrayElements[counter]+arrayElements[counter+1];
   printf("\n sum : %d  and :  %d\n", arrayElements[counter],arrayElements[counter+1]);
   printf("\n Thread Id: %d  Counter:  %d\n", threadid, counter);
   counter++;
   counter++;
   iteration++;
    if(iteration==5)
   printf(" \n Total sum %d\n", sum);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
pthread_t threads[NUM_THREADS];
int rc, t;
for(t=0;t<NUM_THREADS;t++){
  printf("Creating thread %d\n", t);
  rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
  if (rc){
    printf("ERROR; return code from pthread_create() is %d\n", rc);
    exit(-1);
    }
  }
pthread_exit(NULL);

}
```

Output:

```
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4

 sum : 20  and :  18

 Thread Id: 0  Counter:  0

 sum : 16  and :  14

 Thread Id: 1  Counter:  2

 sum : 12  and :  10

 Thread Id: 2  Counter:  4

 sum : 8  and :  6

 Thread Id: 3  Counter:  6
 sum : 4  and :  2

 Thread Id: 4  Counter:  8

 Total sum 110
```

Question 4 [Point 4] Run **threads2.c** several times and list the output for each case.

Solution:
First run:

```
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Thread 2: Spanish: Hola al mundo
Thread 1: French: Bonjour, le monde!
Thread 6: Japan: Sekai e konnichiwa!
Thread 4: German: Guten Tag, Welt!
Thread 7: Latin: Orbis, te saluto!
Thread 3: Klingon: Nuq neH!
Thread 5: Russian: Zdravstvytye, mir!
Thread 0: English: Hello World!
```

Second run:

```
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Thread 7: Latin: Orbis, te saluto!
Thread 5: Russian: Zdravstvytye, mir!
Thread 3: Klingon: Nuq neH!
Thread 6: Japan: Sekai e konnichiwa!
Thread 2: Spanish: Hola al mundo
Thread 4: German: Guten Tag, Welt!
Thread 1: French: Bonjour, le monde!
Thread 0: English: Hello World!
```

Third run:

```
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Thread 2: Spanish: Hola al mundo
Thread 1: French: Bonjour, le monde!
Thread 6: Japan: Sekai e konnichiwa!
Thread 7: Latin: Orbis, te saluto!
Thread 3: Klingon: Nuq neH!
Thread 0: English: Hello World!
Thread 4: German: Guten Tag, Welt!
Thread 5: Russian: Zdravstvytye, mir!
```

Fourth run:

```
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Thread 4: German: Guten Tag, Welt!
Thread 1: French: Bonjour, le monde!
Thread 5: Russian: Zdravstvytye, mir!
Thread 7: Latin: Orbis, te saluto!
Thread 6: Japan: Sekai e konnichiwa!
Thread 3: Klingon: Nuq neH!
Thread 0: English: Hello World!
Thread 2: Spanish: Hola al mundo
```

Question 5 [Point 4] Run **thread3.c**, show the output, and write a brief explanation of what the program does.

Solution:

```
Main thread = 1
Main Created threads A:3 B:2 C:4
Main Thread exiting...
A: In thread A...
A: Created thread D:5
```

```
C: In thread C...
C: Join main thread
C: Main thread (1) returned thread (1) w/status 1
D: In thread D...
D: Created thread E:7
D: Continue B thread = 2
D: Join E thread
F: In thread F...
E: In thread E...
E: Join A thread
B: In thread B...
A: Thread exiting...
E: A thread (3) returned thread (3) w/status 77
E: Join B thread
B: Thread exiting...
E: B thread (2) returned thread (2) w/status 66
E: Join C thread
C: Thread exiting...
E: C thread (4) returned thread (4) w/status 88
F: Thread F is still running...
E: Thread exiting...
D: E thread (7) returned thread (7) w/status 44
D: Thread exiting...
```

Explanation:
The main thread: In this example the main thread's sole purpose is to create new threads. Threads A, B, and C are created by the main thread. Notice that thread B is created suspended. After creating the new threads, the main thread exits. Also notice that the main thread exited by calling thr_exit(). If the main thread had used the exit() call, the whole process would have exited. The main thread's exit status and resources are held until it is joined by thread C.

Thread A: The first thing thread A does after it is created is to create thread D. Thread A then simulates some processing and then exits, using thr_exit(). Notice that thread A was created with the THR_DETACHED flag, so thread A's resources will be immediately reclaimed upon its exit. There is no way for thread A's exit status to be collected by a thr_join() call.

Thread B: Thread B was created in a suspended state, so it is not able to run until thread D continues it by making the thr_continue() call. After thread B is continued, it simulates some processing and then exits. Thread B's exit status and thread resources are held until joined by thread E.

Thread C: The first thing that thread C does is to create thread F. Thread C then joins the main thread. This action will collect the main thread's exit status and allow the main thread's resources to be reused by another thread. Thread C will block, waiting for the main thread to exit, if the main thread has not yet called thr_exit(). After joining the main thread, thread C will simulate some processing and then exit. Again, the exit status and thread resources are held until joined by thread E.

Thread D: Thread D immediately creates thread E. After creating thread E, thread D continues thread B by making the thr_continue() call. This call will allow thread B to start its execution. Thread D then tries to join thread E, blocking until thread E has exited. Thread D then simulates some processing and exits. If all went well, thread D should be the last nondaemon thread running. When thread D exits, it should do two things: stop the execution of any daemon threads and stop the execution of the process.

Thread E: Thread E starts by joining two threads, threads B and C. Thread E will block, waiting for each of these thread to exit. Thread E will then simulate some processing and will exit. Thread E's exit status and thread resources are held by the operating system until joined by thread D.

Thread F: Thread F was created as a bound, daemon thread by using the THR_BOUND and THR_DAEMON flags in the thr_create() call. This means that it will run on its own LWP until all the nondaemon threads have exited the process. This type of thread can be used when you want some type of "background" processing to always be running, except when all the "regular" threads have exited the process. If thread F was created as a non-daemon thread, then it would continue to run forever, because a process will continue while there is at least one thread still running. Thread F will exit when all the nondaemon threads have exited. In this case, thread D should be the last nondaemon thread running, so when thread D exits, it will also cause thread F to exit.