CS471: Operating System Concepts
Fall 2007
(Lecture: WF 9:15-10:30 AM)
Homework #1
Points: 20
Due: September 5, 2007
Solution

Question 1 [Points 5] 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, its child pid, and the command line argument passed to it. The child process prints its pid, its parent pid, its child pid, the original command line argument and the modified argument. The grandchild prints its pid, its parent id, the command line argument that was there when its parent called it, and the modified argument. (So if the program was called as "modfork1.out 10" the parent prints 10 for argument, child prints 10 and 20; and the grandchild prints 20 and 40 as the argument value.) 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) { /* This is the child */
pid_t pid1;
pid1 = fork(); /* Fork the grand child */
if (pid1 == 0) { /* This is the grand child */
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 the child \n");
printf("My pid is: %d\n", getpid());
printf("My child pid is: %d\n", pid1);
}
}
else {
printf("I am the parent\n");
printf("My 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: 29084
pid of grand child parent is: 29083
exiting ... 29084
I am the parent
My pid is: 29082
My child pid is: 29083
exiting ... 29082
I am the child
My pid is: 29083
grand child pid is: 29084
exiting ... 29083

Question 2 [Points 5] Modify and run **thread1.c** in the following way. There is an array of 18 elements defined in the program. The elements of the array are: [20 18 16 14 12 10 8 6 4 2 25 23 21 19 17 15 13 11]. Thread 1 adds the first three elements (i.e., 20, 18, 16), Thread 2 adds the next three elements (i.e., 16, 14, 12), …, Thread 6 adds the last three elements (15, 13, 11). Finally, the sum of all the 18 elements is printed by the program

SOLUTION:

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS 6
int counter=0;
int arrayElements[]={20 , 18 , 16 , 14 , 12 , 10 , 8 , 6 , 4 , 2 , 25 ,
23 , 21 , 19 , 17 , 15 , 13 , 11};
int sum=0;
int temp=0;
int iteration=0;
void *PrintHello(void *threadid)
{
sum=sum+arrayElements[counter]+arrayElements[counter+1]+
arrayElements[counter+2];
printf("\n Thread: %d\n",threadid);
printf("\n sum : %d,  %d and %d is %d\n",
arrayElements[counter],arrayElements[counter+1],arrayElements[counter+2
],arrayElements[counter]+arrayElements[counter+1]+arrayElements[counter
+2]);
counter++;
counter++;
counter++;
iteration++;
if(iteration==6)
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
Creating thread 5

Thread: 0

sum : 20, 18 and 16 is 54

Thread: 1

sum : 14, 12 and 10 is 36

Thread: 2

sum : 8, 6 and 4 is 18

Thread: 3

sum : 2, 25 and 23 is 50

Thread: 4

sum : 21, 19 and 17 is 57

Thread: 5

sum : 15, 13 and 11 is 39

Total sum 254

Question 3 [Point 5] Run **thread2.c** several times and list the output for each case.
SOLUTION:
First 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 0: English: Hello World!
Thread 1: French: Bonjour, le monde!
Thread 2: Spanish: Hola al mundo
Thread 3: Klingon: Nuq neH!
Thread 5: Russian: Zdravstvytye, mir!
Thread 6: Japan: Sekai e konnichiwa!
Thread 7: Latin: Orbis, te saluto!
Thread 4: German: Guten Tag, Welt!

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 6: Japan: Sekai e konnichiwa!
Thread 5: Russian: Zdravstvytye, mir!
Thread 4: German: Guten Tag, Welt!
Thread 3: Klingon: Nuq neH!
Thread 2: Spanish: Hola al mundo
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 6: Japan: Sekai e konnichiwa!

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

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

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

OUTPUT:

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 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 nondaemon 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.