Operating System Homework 4

Jin yan Xu, 3160101126, Information Security

1. Operating System Concept Chapter 4 Exercises: 4.8, 4.9, 4.10, 4.17, 4.19

Answer:

4.8) Provide two programming examples in which multithreading does not provide better performance than a single-threaded solution.

(1) Sequential programs, like calculate Fibonacci, can have a better performance in single-threaded solution, because this kind programs depends on the previous state and the same state will not be executed repeatedly.

(2) Programs have many I/O operations can have a better performance in single-threaded solution too, because our hard disk is in sequential structure. Frequent switching of orbit will cause waste.

4.9) Under what circumstances does a multithreaded solution using multiple kernel threads provide better performance than a single-threaded solution on a single-processor system?

When a program has to wait for system resource frequently, a multithreaded solution can have a better performance than single-threaded solution, because when a thread is blocking, another thread can continue to execute in multithreaded solution, while in single-threaded there is no ways but to wait for the blocking process.

4.10) Which of the following components of program state are shared across threads in a multithreaded process?

a. Register values **b. Heap memory**

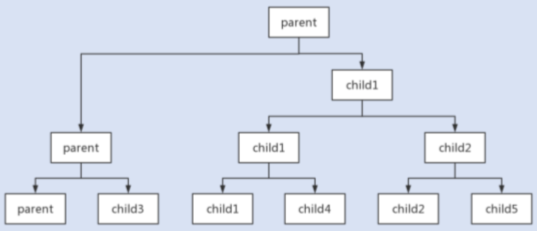
c. Global variables **d. Stack memory**

The threads in multithreaded processes share heap memory and global variables. But each thread has its own register values and stack memory. So, the answer is b & c.

4.17) Consider the following code segment:

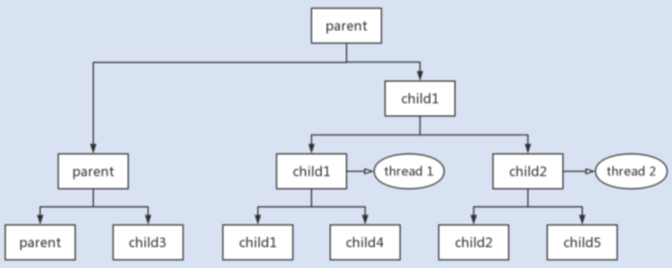
```
pid_t pid;

pid = fork();
if (pid == 0) { /* child process */
   fork();
   thread_create( . . .);
}
fork();
```

a. How many unique processes are created?

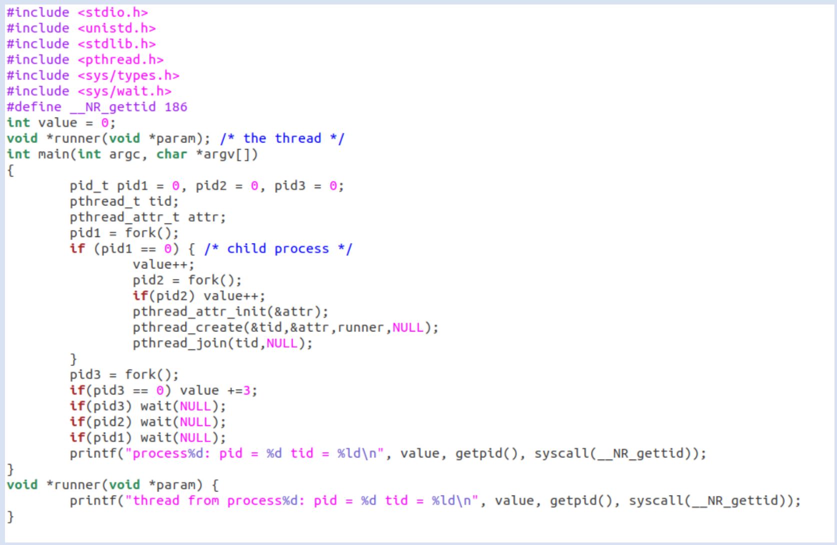


Left picture is the running flow of the program in my mind, there are **6 unique processes**, later, I will code a demo program to prove this.

b. How many unique threads are created?

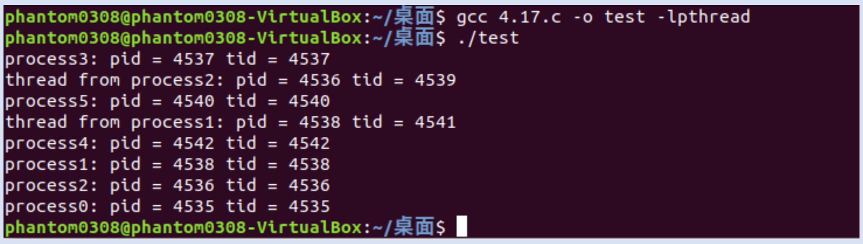


Left picture is the running flow of the program in my mind, there are **2 unique threads** (not count the main threads of all the processes), next, I will code a demo program to prove this. Follow is the code I use:



```c
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/types.h>
#include <sys/wait.h>
#define __NR_gettid 186
int value = 0;
void *runner(void *param); /* the thread */
int main(int argc, char *argv[])
{
        pid_t pid1 = 0, pid2 = 0, pid3 = 0;
        pthread_t tid;
        pthread_attr_t attr;
        pid1 = fork();
        if (pid1 == 0) { /* child process */
                value++;
                pid2 = fork();
                if(pid2) value++;
                pthread_attr_init(&attr);
                pthread_create(&tid,&attr,runner,NULL);
                pthread_join(tid,NULL);
        }
        pid3 = fork();
        if(pid3 == 0) value +=3;
        if(pid3) wait(NULL);
        if(pid2) wait(NULL);
        if(pid1) wait(NULL);
        printf("process%d: pid = %d tid = %ld\n", value, getpid(), syscall(__NR_gettid));
}
void *runner(void *param) {
        printf("thread from process%d: pid = %d tid = %ld\n", value, getpid(), syscall(__NR_gettid));
}
```

When a process or a thread terminates, it will print its information, containing pid and tid.

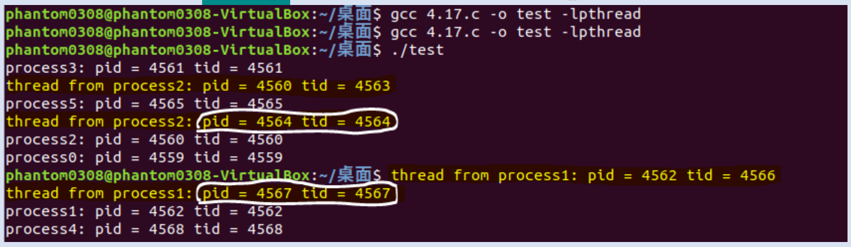


```
phantom0308@phantom0308-VirtualBox:~/桌面$ gcc 4.17.c -o test -lpthread
phantom0308@phantom0308-VirtualBox:~/桌面$ ./test
process3: pid = 4537 tid = 4537
thread from process2: pid = 4536 tid = 4539
process5: pid = 4540 tid = 4540
thread from process1: pid = 4538 tid = 4541
process4: pid = 4542 tid = 4542
process1: pid = 4538 tid = 4538
process2: pid = 4536 tid = 4536
process0: pid = 4535 tid = 4535
phantom0308@phantom0308-VirtualBox:~/桌面$ 
```

It matches the running flow, which means our conclusion is right.

Easter egg:

In this question, it doesn't contain the situation that the thread calls fork(), let's try this. Add a line: fork(); in function runner().

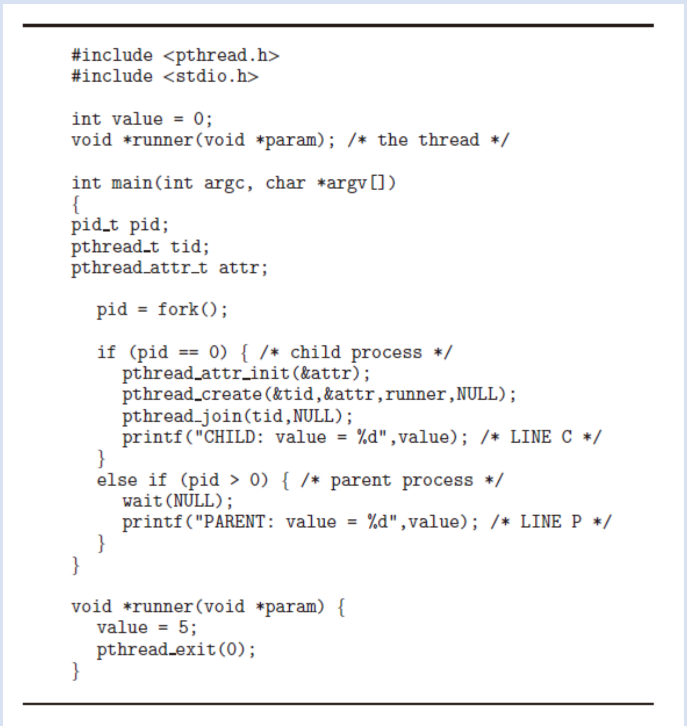


```
phantom0308@phantom0308-VirtualBox:~/桌面$ gcc 4.17.c -o test -lpthread
phantom0308@phantom0308-VirtualBox:~/桌面$ gcc 4.17.c -o test -lpthread
phantom0308@phantom0308-VirtualBox:~/桌面$ ./test
process3: pid = 4561 tid = 4561
thread from process2: pid = 4560 tid = 4563
process5: pid = 4565 tid = 4565
thread from process2: pid = 4564 tid = 4564
process2: pid = 4560 tid = 4560
process0: pid = 4559 tid = 4559
phantom0308@phantom0308-VirtualBox:~/桌面$ thread from process1: pid = 4562 tid = 4566
thread from process1: pid = 4567 tid = 4567
process1: pid = 4562 tid = 4562
process4: pid = 4568 tid = 4568
```

We can know that: In Ubuntu 18.04, the function fork() copies the thread who called fork() as a process instead of copies every threads.

So the result doesn't match teacher taught in class, thread calls function fork() performs differently in different systems.

4.19) The program shown in Figure 4.22 uses the Pthreads API. What would be the output from the program at LINE C and LINE P?

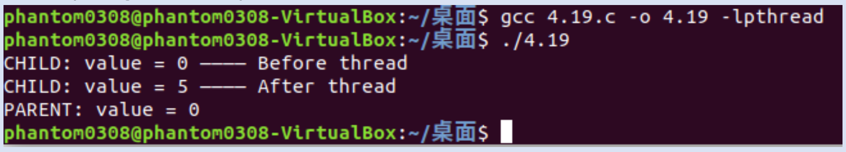


```
#include <pthread.h>
#include <stdio.h>

int value = 0;
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
pid_t pid;
pthread_t tid;
pthread_attr_t attr;

  pid = fork();

  if (pid == 0) { /* child process */
    pthread_attr_init(&attr);
    pthread_create(&tid,&attr,runner,NULL);
    pthread_join(tid,NULL);
    printf("CHILD: value = %d",value); /* LINE C */
  }
  else if (pid > 0) { /* parent process */
    wait(NULL);
    printf("PARENT: value = %d",value); /* LINE P */
  }
}

void *runner(void *param) {
  value = 5;
  pthread_exit(0);
}
```

Figure 4.22 C program for Exercise 4.19.

The output at LINE C is 5, at LINE P is 0. This is because child process copies the value from the parent process instead of sharing, but the thread does, so the value in child process has changed in thread function runner().

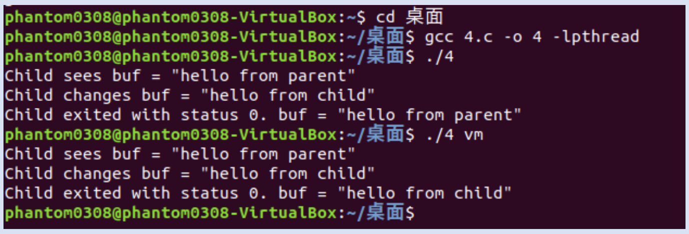
We can run the program to prove this:



```
phantom0308@phantom0308-VirtualBox:~/桌面$ gcc 4.19.c -o 4.19 -lpthread
phantom0308@phantom0308-VirtualBox:~/桌面$ ./4.19
CHILD: value = 0 ---- Before thread
CHILD: value = 5 ---- After thread
PARENT: value = 0
phantom0308@phantom0308-VirtualBox:~/桌面$
```

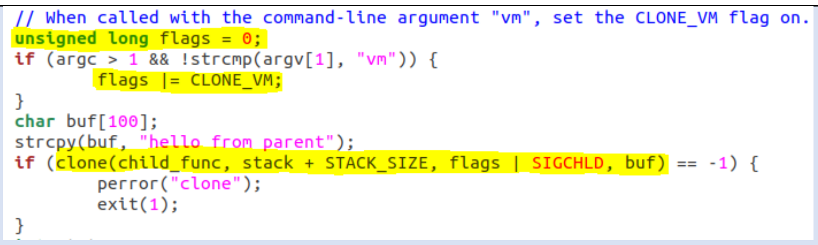
2. Compile and run the following program twice, with ‘vm’ and without ‘vm’ as the argument, respectively. Take the screenshots of the running results and explain why the output is different.

Answer:



```
phantom0308@phantom0308-VirtualBox:~$ cd 桌面
phantom0308@phantom0308-VirtualBox:~/桌面$ gcc 4.c -o 4 -lpthread
phantom0308@phantom0308-VirtualBox:~/桌面$ ./4
Child sees buf = "hello from parent"
Child changes buf = "hello from child"
Child exited with status 0. buf = "hello from parent"
phantom0308@phantom0308-VirtualBox:~/桌面$ ./4 vm
Child sees buf = "hello from parent"
Child changes buf = "hello from child"
Child exited with status 0. buf = "hello from child"
phantom0308@phantom0308-VirtualBox:~/桌面$
```

I have made a little change of the code, I add a line in child_func() to show buffer after changed. We can see that when running with “vm”, the changes in child can affect in parent. Without “vm” the result will not changed by child. The different between having “vm” and no “vm” is whether bring CLONE_VM parameters when calling function clone(), because this lines:



```c
// When called with the command-line argument "vm", set the CLONE_VM flag on.
unsigned long flags = 0;
if (argc > 1 && !strcmp(argv[1], "vm")) {
        flags |= CLONE_VM;
}
char buf[100];
strcpy(buf, "hello from parent");
if (clone(child_func, stack + STACK_SIZE, flags | SIGCHLD, buf) == -1) {
        perror("clone");
        exit(1);
}
```

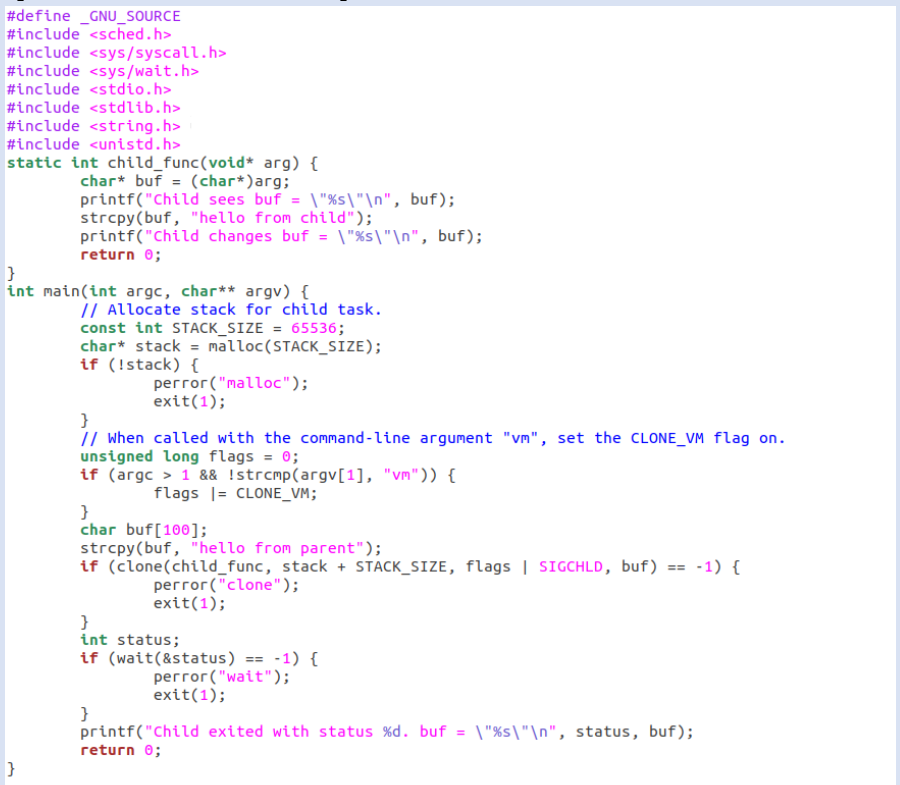
If CLONE_VM is set, the calling process and the child processes run in the same memory space. In particular, memory writes performed by the calling process or by the child process are also visible in the other process. So, ***hello from parent*** is changed to ***hello from child***.

If CLONE_VM isn't set, the child process runs in a separate copy of the memory space of the calling process at the time of clone(). Memory writes in one of the processes do not affect the other, just like fork(). So, the buffer in parent process won't change.

Notice:

When using function clone(), you need to notice that the stack address we pass is the end address of the stack, instead of the start address, so here is ***stack + STACK_SIZE*** instead of ***stack***. It depends on the hardware settings, when push, CPU performs subtraction operations on stack addresses, add when pop.

Following is the code after change:



```c
#define _GNU_SOURCE
#include <sched.h>
#include <sys/syscall.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
static int child_func(void* arg) {
        char* buf = (char*)arg;
        printf("Child sees buf = \"%s\"\n", buf);
        strcpy(buf, "hello from child");
        printf("Child changes buf = \"%s\"\n", buf);
        return 0;
}
int main(int argc, char** argv) {
        // Allocate stack for child task.
        const int STACK_SIZE = 65536;
        char* stack = malloc(STACK_SIZE);
        if (!stack) {
                perror("malloc");
                exit(1);
        }
        // When called with the command-line argument "vm", set the CLONE_VM flag on.
        unsigned long flags = 0;
        if (argc > 1 && !strcmp(argv[1], "vm")) {
                flags |= CLONE_VM;
        }
        char buf[100];
        strcpy(buf, "hello from parent");
        if (clone(child_func, stack + STACK_SIZE, flags | SIGCHLD, buf) == -1) {
                perror("clone");
                exit(1);
        }
        int status;
        if (wait(&status) == -1) {
                perror("wait");
                exit(1);
        }
        printf("Child exited with status %d. buf = \"%s\"\n", status, buf);
        return 0;
}
```

You can get the code used in this report from:
https://pan.baidu.com/s/1sV6uTLFzSawUG8AwnrJklw   password: gft9