COS3043
System Fundamentals

Lab 4

Task

Rewrite above program so that the processes instead of threads are created and the number of child processes created is fixed as two. The program should make use of kernel timer to measure and print the real time, processor time, user space time and kernel space time (a.k.a. system space time) for each process.

Basic Concepts

What is a Process?

Process is a program in execution. Process is a passive entity with a program counter specifying the next instruction to execute and a set of associated resources. Each process is represented in the operating system by a process control block (PCB) also called a task control block. It contains information such as process state, program counter, CPU registers.

Process VS Thread

COMPARISON BASIS	PROCESS	THREAD
Definition	A process is a program under execution i.e an active program.	A thread is a lightweight process that can be managed independently by a scheduler.
Context Switching Time	Processes require more time for context switching as they are heavier.	Threads require less time for context switching as they are lighter than processes.
Memory	Processes are totally independent and don’t share memory.	A thread may share some memory with its peer threads.
Communication	Communication between processes requires more time than between threads.	Communication between threads requires less time than between processes.
Blocked	If a process gets blocked, remaining processes can continue execution.	If a user level thread gets blocked, all of its peer threads also get blocked.

Process VS Thread (continue)

COMPARISON BASIS	PROCESS	THREAD
Resources	Processes require more resources than threads.	Threads generally need less resources than processes.
Dependency	Individual processes are independent of each other.	Threads are parts of a process and so are dependent.
Data and Code Sharing	Processes have independent data and code segments.	A thread shares the data segment, code segment, files etc. with its peer threads.
Treatment by OS	All the different processes are treated separately by the operating system.	All user level peer threads are treated as a single task by the operating system.
Time for creation	Processes require more time for creation.	Threads require less time for creation.
Time for termination	Processes require more time for termination.	Threads require less time for termination.

Kernel Timer

A Kernel timer is a data structure that instructs the kernel to execute a user-defined function with a user-defined argument at a user-defined time.

In computing, the kernel is the main component of most computer operating systems; it is a bridge between applications and the actual data processing done at the hardware level. The kernel's responsibilities include managing the system's resources (the communication between hardware and software components).

Processor Time

The amount of time a process takes to run, given that it has exclusive and uninterrupted use of the CPU. Note that in a modern computer, this would be very unusual, so the processor time calculation for most processes involves adding up all the small amounts of time the CPU actually spends on the process. Some systems break processor time down into user time and system time.

Real Time

Real time is the amount of time spent between process creation up to the end of process.

User Space Time

User space is that portion of system memory in which user processes run. The amount of time spent by a process in user mode from the time of creation to its process termination is called user space time.

Kernel Space Time (a.k.a. System Space Time)

The kernel is a program that constitutes the central core of a computer operating system. It is not a process, but rather a controller of processes, and it has complete control over everything that occurs on the system. This includes managing individual user processes within user space and preventing them from interfering with each other.

The amount of time spent by a process in kernel mode from the time of creation to its process termination is called kernel space time.

Keywords, Functions & Header Files

1) Times () - get process times

The times () function stores the current process times in the struct tms that buffer points to. The struct tms is as defined in <**sys/times.h**>:

```
struct tms{
        clock_ttms_utime; /* user time */
        clock_ttms_stime; /* system time */
        clock_ttms_cutime; /* user time of children */
        clock_ttms_cstime; /* system time of children */
};
```

1) Times () - get process times

- The tms_utime field contains the CPU time spent executing instructions of the calling process.
- The tms_stime field contains the CPU time spent in the system while executing tasks on behalf of the calling process.
- The tms_cutime field contains the sum of the tms_utime and tms_cutime values for all waited-for terminated children.
- The tms_cstime field contains the sum of the tms_stime and tms_cstime values for all waited-for terminated children.

2) Wait () - wait for process termination.

The wait function suspends execution of the current process until a child has exited, or until a signal is delivered whose action is to terminate the current process or to call a signal handling function. If a child has already exited by the time of the call (a so-called "zombie" process), the function returns immediately. Any system resources used by the child are freed.

3) Fork () - create a child process.

- Fork creates a child process that differs from the parent process only in its PID, and in the fact that resource utilizations are set to 0. File locks and pending signals are not inherited. Under Linux, fork is implemented using copy-on-write pages, so the only penalty incurred by fork is the time and memory required to duplicate the parent's page tables, and to create a unique task structure for the child.
- Return Value: On success, the PID of the child process is returned in the parent's thread of execution, and a 0 is returned in the child's thread of execution. On failure, a -1 will be returned in the parent's context, no child process will be created, and err no will be set appropriately.

4) Exit () - cause normal program termination.

- The exit () function causes normal program termination and the value of status & 0377 is returned to the parent. All functions registered with at exit () and on exit () are called in the reverse order of their registration, and all open streams are flushed and closed. Files created by tmpfile () are removed.
- The C standard specifies two defines EXIT_SUCCESS and EXIT_FAILURE that may be passed toexit () to indicate successful or unsuccessful termination, respectively.
- Return Value: The exit () function does not return.

5) Unistd.h

This header file is used for system call wrapper functions such asfork (), pipe () and I/O primitives such as read, write, close, etc.