CIS657 Principles of Operating Systems

Zhen Ren

1 Homework2

Ex. 2.1 — (15pts) Draw schedule diagrams of the following task set for Round-Robin, First-in-first-out (FIFO), and Shortest job first (SJF), respectively. The task set, (arrival time, execution time): T1(1, 10), T2(2, 4), T3(3, 1), T4(4, 6). (5pts for each diagram).?

Round-Robin	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17		
	T1	T2	T3	T4	T1	T2	T4	T1	T2	T4	T1	T2	T4	T1	T4	T1	T4		
	18	19	20	21	22	23													
	T1	T1	T1	T1															

First in First Out	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T2	T2	T2	T2	T3	T4	T4	
	18	19	20	21	22	23												
	T4	T4	T4	T4														

Shortest job First	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	
	T1	T1	T1	T1	T1	T1	T1	T1	T1	T1	T3	T2	T2	T2	T2	T4	T4	
	18	19	20	21	22	23												
	T4	T4	T4	T4														

Ex. 2.2 — (15pts) The following shows three different ways a process can be created.
1.A human user explicitly commands the OS to run a program by double-clicking the program icon or typing a command.
2.A currently running process creates a sub-process (also known as a child process) prompted by an external event (a human user's request, an event caused by another process).
3.A currently running process creates a child process explicitly executing an instruction such as fork().
The third case, in which a currently running process creates a new process, in general covers the first two cases. Explain why.

For the first case, we can use a daemon process in the OS will receive the double-clicking signal from the user, then the daemon process will create, using fork() to create, its child process and run the program that a user selected. Daemon received the signal, which means this is a running process, then call fork() to create a child process. From this example, the third case covers the first case.

For the second case, a running process creates a sub-process, which is the same as a child process, when receive the external devices' signal or interruptions. The method that running process creates a child process which is fork(). So the processing is covered by the third case.
So the all fundenmtally are the same

Ex. 2.3 — (15pts) Write a small program that uses fork() and exec() to run the ls command.

```
# include < stdio .h >
# include < stdlib .h >
# include < unistd .h >
# include < sys / types .h >
main () {
pid_t pid ;
int status ;
8. pid = fork ();
9. if ( pid != 0) { // parent code
10. waitpid ( pid , & status , 0);
11. printf ("******* I am the parent !!!\ n "); }
12. else {
13. static char * argv []={" ls " ," - a " , NULL };
14. execvp (" ls " , argv ); }
15.} // main
```

Ex. 2.4 — (15pts) The amount of time that takes to make a context-switch between threads versus that of between processes is much shorter. Explain why.

1. Thread Switching : Thread switching is a type of context switching from one thread to another thread in the same process. Thread switching is very efficient and much cheaper because it involves switching out only identities and resources such as the program counter, registers and stack pointers. The cost of thread-to-thread switching is about the same as the cost of entering and exiting the kernel.
2. Process Switching : Process switching is a type of context switching where we switch one process with another process. It involves switching of all the process resources with those needed by a new process. This means switching the memory address space. This includes memory addresses, page tables, and kernel resources, caches in the processor.

Ex. 2.5 — (25pts) Write a pseudo-code for exec(), fork(), waidpid(), and exit() system calls. Explain how these system calls are inter-related. Each pseudo-code is 5pts and the explanation is 5pts.

```
1. Algorithm Exec ()
2. Input : the name of a file that is to be executed.
3. Output : None
4. if an error has occurred :-1
5. replace the current contents of the process with a name of file to execute
6. check for available kernel resources ;
7. reload the address space ( text , data , stack , etc ) of the process 'name'
8. do other initializations
9. change state to ready to run.
10. put it to the ready - queue
11. executing the process from the entry point (the name of the file)
12. end Algorithm Exec ()
```

```
1. Algorithm Fork ()
```

```
2. Input : none
3. Output :
4. to parent process , child PID number ;
5. to child process , 0
6. if ( canBeParent ) {
// executing process is a parent process
7. check for available kernel resources ;
8. get free process table slot and a unique ID
9. make child state being created .
10. copy meta - data of parent process to the child
11. copy parent's address space ( text , data , stack , etc )to child's
12. do other initializations
13. change child state to ready to run.
14. put the child to the ready - queue
15. return ( child ID ); }
16. else {
17. set this processes canBeParent = True ;
18. and other necessary initializations
19. return (0); }
20. end Algorithm Fork ()
```

```
1. Algorithm Waitpid ()
2. Input : pid, options
3. Output :
4. successfully run: a value of the process (usually a child) whose status
information has been obtained;
5. options=WNOHANG And if there is at least one process (usually a child) whose
status information is not available, returns 0;
6. error: -1, and error will point out where the error is;
7. suspend or stall the current process to wait for process ID, which equals to
8. pid to finish.
9. if (process pid successfully finish){
10. return process ID;
11. }
12. if(option==WNOHANG && no terminated process available){ return 0;}
13. if(executing with errors) {return -1;}
14. end Algorithm Waitpid ()
```

```
1. Algorithm Exit ()
```

```
2. Input : none
3. Output : status // will be a number between 0 and 255
4. Terminate the current process
5. if(canSuccessfullyTerminate){ return 0;}// a zero exit status means
successfully terminate the process.
6. else{//a non-zero number means failure statues.
7. if(commandNotFound){return 127;}//return 127 means the command not found.
8. if(commandFoundNotExecutable){return 126;}//return 126 means command found
but not executable.
9  all the other cases means a different failure will return a non-zero number
10   }
11. end Algorithm Exit ()
```

exec() will replace the current executing process and run from the entry point. The parent process which runs exec() will suspend until the child complete it process
fork() will create a new process by copying the current executing process called parent process. Both parent and child can execute simultaneously
waitpid() blocks current process until the process is finished. Mainly used to release the resources of called process.
exit() terminates the process itself with the given value as return

Ex. 2.6 — (15pts) In Unix, there is a category of processes called zombie process. What are these?

A zombie process is a process whose execution is completed but it still has an entry in the process table in the OS. Zombie process mostly happened in the child process, while the parent processes still need resource from the child. Following is three ways caused zombie process.
1. Using wait() system call: When the parent process calls wait(), after the creation of a child, it indicates that, it will wait for the child to complete and it will reap the exit status of the child. The parent process is suspended(waits in a waiting queue) until the child is terminated. It must be understood that during this period, the parent process does nothing just waits.
2. By ignoring the SIGCHLD signal: When a child is terminated, a corresponding SIGCHLD signal is delivered to the parent, if we call the 'signal(SIGCHLD,SIGIGN)', then the SIGCHLD signal is ignored by the system, and the child process entry is deleted from the process table. Thus, no zombie is created. However, in this case, the parent cannot know about the exit status of the child.
3. By using a signal handler: The parent process installs a signal handler for the SIGCHLD signal. The signal handler calls wait() system call within it. In this scenario, when the child terminated, the SIGCHLD is delivered to the parent. On receipt of SIGCHLD, the corresponding handler is activated, which in turn calls the wait() system call. Hence, the parent collects the exit status almost

immediately and the child entry in the process table is cleared. Thus no zombie is created.
The Zombie process is not a huge danger for OS, due to the execution of the process is completed. But the zombie processes will take a process ID, and not released the ID number for OS. So the ID is not available for other process.