Lectures on Operating Systems (Mythili Vutukuru, IIT Bombay)

Practice Problems: Processes

1. Answer yes/no, and provide a brief explanation.

   (a) Can two processes be concurrently executing the same program executable?

   (b) Can two running processes share the complete process image in physical memory (not just parts of it)?

   Ans:

   (a) Yes, two processes can run the same program.

   (b) No, in general. (Only time this is possible is with copy-on-write during fork, and before any writes have been made.)

2. Consider a process executing on a CPU. Give an example scenario that can cause the process to undergo:

   (a) A voluntary context switch.

   (b) An involuntary context switch.

   Ans:

   (a) A blocking system call.

   (b) Timer interrupt that causes the process to be switched out.

3. Consider a parent process P that has forked a child process C. Now, P terminates while C is still running. Answer yes/no, and provide a brief explanation.

   (a) Will C immediately become a zombie?

   (b) Will P immediately become a zombie, until reaped by its parent?

   Ans:

   (a) No, it will be adopted by init.

   (b) Yes.

4. A process in user mode cannot execute certain privileged hardware instructions. [T/F]

   Ans: True, some instructions in every CPU's instruction set architecture can only be executed when the CPU is running in a privileged mode (e.g., ring 0 on Intel CPUs).

5. Which of the following C library functions do NOT directly correspond to (similarly named) system calls? That is, the implentations of which of these C library functions are NOT straightforward invocations of the underlying system call?

   A. `system`, which executes a bash shell command.

   B. `fork`, which creates a new child process.

   C. `exit`, which terminates the current process.

   D. `strlen`, which returns the length of a string.

   Ans: A,D

6. Which of the following actions by a running process will *always* result in a context switch of the running process, even in a non-preemptive kernel design?

   A. Servicing a disk interrupt, that results in another blocked process being marked as ready/runnable.

   B. A blocking system call.

   C. The system call exit, to terminate the current process.

   D. Servicing a timer interrupt.

   Ans: B,C

7. Consider two machines A and B of different architectures, running two different operating systems OS-A and OS-B. Both operating systems are POSIX compliant. The source code of an application that is written to run on machine A must always be rewritten to run on machine B. [T/F]

   Ans: False. If the code is written using the POSIX API, it need not be rewritten for another POSIX compliant system.

8. Consider the scenario of the previous question. An application binary that has been compiled for machine A may have to be recompiled to execute correctly on machine B. [T/F]

   Ans: True. Even if the code is POSIX compliant, the CPU instructions in the compiled executable are different across different CPU architectures.

9. A process makes a system call to read a packet from the network device, and blocks. The scheduler then context-switches this process out. Is this an example of a voluntary context switch or an involuntary context switch?

   Ans: Voluntary context switch.

10. A context switch can occur only after processing a timer interrupt, but not after any other system call or interrupt. [T/F]

    Ans: False, a context switch can also occur after a blocking system call for example.

11. A C program cannot directly invoke the OS system calls and must always use the C library for this purpose. [T/F]

    Ans: False, it is cumbersome but possible to directly invoke system calls from user code.

12. A process undergoes a context switch every time it enters kernel mode from user mode. [T/F]

    Ans: False, after finishing its job in kernel mode, the OS may sometimes decide to go back to the user mode of the same process, without switching to another process.

13. Consider a parent process P that has forked a child process C in the program below.

```
int a = 5;
int fd = open(...) //opening a file
int ret = fork();
if(ret >0) {
  close(fd);
  a = 6;
  ...
}
else if(ret==0) {
  printf("a=%d\n", a);
  read(fd, something);
}
```

After the new process is forked, suppose that the parent process is scheduled first, before the child process. Once the parent resumes after fork, it closes the file descriptor and changes the value of a variable as shown above. Assume that the child process is scheduled for the first time only after the parent completes these two changes.

(a) What is the value of the variable `a` as printed in the child process, when it is scheduled next? Explain.

(b) Will the attempt to read from the file descriptor succeed in the child? Explain.

Ans:

(a) 5. The value is only changed in the parent.

(b) Yes, the file is only closed in the parent.

14. Consider the following pseudocode. Assume all system calls succeed and there are no other errors in the code.

```
int ret1 = fork(); //fork1
int ret2 = fork(); //fork2
int ret3 = fork(); //fork3
wait();
wait();
wait();
```

Let us call the original parent process in this program as P. Draw/describe a family tree of P and all its descendents (children, grand children, and so on) that are spawned during the execution of this program. Your tree should be rooted at P. Show the spawned descendents as nodes in the tree, and connect processes related by the parent-child relationship with an arrow from parent to child. Give names of the form C¡number¿ for descendents, where child processes created by fork ”i” above should have numbers like ”i1”, ”i2”, and so on. For example, child processes created by fork3 above should have names C31, C32, and so on.

Ans: P has three children, one in each fork statement: C11, C21, C31. C11 has two children in the second and third fork statements: C22, C32. C21 and C22 also have a child each in the third fork statement: C33 and C34.

15. Consider a parent process that has forked a child in the code snippet below.

```
int count = 0;
ret = fork();
if(ret == 0)  {
printf("count in child=%d\n", count);
}
else  {
count = 1;
}
```

The parent executes the statement ”count = 1” before the child executes for the first time. Now, what is the value of count printed by the code above? Assume that the OS implements a simple fork (not a copy-on-write fork).

Ans: 0 (the child has its own copy of the variable)

16. Consider the wait family of system calls (wait, waitpid etc.) provided by Linux. A parent process uses some variant of the wait system call to wait for a child that it has forked. Which of the following statements is always true when the parent invokes the system call?

A. The parent will always block.

B. The parent will never block.

C. The parent will always block if the child is still running.

D. Whether the parent will block or not will depend on the system call variant and the options with which it is invoked.

Ans: D

17. Consider a simple linux shell implementing the command `sleep 100`. Which of the following is an accurate ordered list of system calls invoked by the shell from the time the user enters this command to the time the shell comes back and asks the user for the next input?

A. wait-exec-fork

B. exec-wait-fork

C. fork-exec-wait

D. wait-fork-exec

Ans: C

18. Consider a process P1 that forks P2, P2 forks P3, and P3 forks P4. P1 and P2 continue to execute while P3 terminates. Now, when P4 terminates, which process must wait for and reap P4?

    Ans: init (orphan processes are reaped by init)

19. Consider the following three processes that arrive in a system at the specified times, along with the duration of their CPU bursts. Process P1 arrives at time t=0, and has a CPU burst of 10 time units. P2 arrives at t=2, and has a CPU burst of 2 units. P3 arrives at t=3, and has a CPU burst of 3 units. Assume that the processes execute only once for the duration of their CPU burst, and terminate immediately. Calculate the time of completion of the three processes under each of the following scheduling policies. For each policy, you must state the completion time of all three processes, P1, P2, and P3. Assume there are no other processes in the scheduler's queue. For the preemptive policies, assume that a running process can be immediately preempted as soon as the new process arrives (if the policy should decide to premempt).

    (a) First Come First Serve
    (b) Shortest Job First (non-preemptive)
    (c) Shortest Remaining Time First (preemptive)
    (d) Round robin (preemptive) with a time slice of (atmost) 5 units per process

    Ans:

    (a) FCFS: P1 at 10, P2 at 12, P3 at 15
    (b) SJF: same as above
    (c) SRTF: P2 at 4, P3 at 7, P1 at 15
    (d) RR: P2 at 7, P3 at 10, P1 at 15

20. Consider an application that is composed of one master process and multiple worker processes that are forked off the master at the start of application execution. All processes have access to a pool of shared memory pages, and have permissions to read and write from it. This shared memory region (also called the request buffer) is used as follows: the master process receives incoming requests from clients over the network, and writes the requests into the shared request buffer. The worker processes must read the request from the request buffer, process it, and write the response back into the same region of the buffer. Once the response has been generated, the server must reply back to the client. The server and worker processes are single-threaded, and the server uses event-driven I/O to communicate over the network with the clients (you must not make these processes multi threaded). You may assume that the request and the response are of the same size, and multiple such requests or responses can be accommodated in the request buffer. You may also assume that processing every request takes similar amount of CPU time at the worker threads.

    Using this design idea as a starting point, describe the communication and synchronization mechanisms that must be used between the server and worker processes, in order to let the server correctly delegate requests and obtain responses from the worker processes. Your design must ensure that every request placed in the request buffer is processed by one and only one worker thread. You must also ensure that the system is efficient (e.g., no request should be kept waiting if some worker is free) and fair (e.g., all workers share the load almost equally). While you can use any IPC mechanism of your choice, ensure that your system design is practical enough to be implementable in a modern multicore system running an OS like Linux. You need not write any code, and a clear, concise and precise description in English should suffice.

Ans: Several possible solutions exist. The main thing to keep in mind is that the server should be able to assign a certain request in the buffer to a worker, and the worker must be able to notify completion. For example, the master can use pipes or sockets or message queues with each worker. When it places a request in the shared memory, it can send the position of the request to one of the workers. Workers listen for this signal from the master, process the request, write the response, and send a message back to the master that it is done. The master monitors the pipes/sockets of all workers, and assigns the next request once the previous one is done.

21. Consider the following events that happen during a context switch from (user mode of) process P to (user mode of) process Q, triggered by a timer interrupt that occurred when P was executing, in a Unix-like operating system design studied in class. Arrange the events in chronological order, starting from the earliest to the latest.

    (A) The CPU program counter moves from the kernel address space of P to the kernel address space of Q.

    (B) The CPU executing process P moves from user mode to kernel mode.

    (C) The CPU stack pointer moves from the kernel stack of P to the kernel stack of Q.

    (D) The CPU program counter moves from the kernel address space of Q to the user address space of Q.

    (E) The OS scheduler code is invoked.

    Ans:

    B E C A D

22. Consider a system with two processes P and Q, running a Unix-like operating system as studied in class. Consider the following events that may happen when the OS is concurrently executing P and Q, while also handling interrupts.

    (A) The CPU program counter moves from pointing to kernel code in the kernel mode of process P to kernel code in the kernel mode of process Q.

    (B) The CPU stack pointer moves from the kernel stack of P to the kernel stack of Q.

    (C) The CPU executing process P moves from user mode of P to kernel mode of P.

    (D) The CPU executing process P moves from kernel mode of P to user mode of P.

    (E) The CPU executing process Q moves from the kernel mode of Q to the user mode of Q.

    (F) The interrupt handling code of the OS is invoked.

    (G) The OS scheduler code is invoked.

    For each of the two scenarios below, list out the chronological order in which the events above occur. Note that all events need not occur in each question.

    (a) A timer interrupt occurs when P is executing. After processing the interrupt, the OS scheduler decides to return to process P.

(b) A timer interrupt occurs when P is executing. After processing the interrupt, the OS scheduler decides to context switch to process Q, and the system ends up in the user mode of Q.

Ans:

(a) C F G D

(b) C F G B A E

23. Which of the following pieces of information in the PCB of a process are changed when the process invokes the exec system call?

    (a) Process identifier (PID)

    (b) Page table entries

    (c) The value of the program counter stored within the user space context on the kernel stack

    Ans: (a) does not change. (b) and (c) change because the process gets a new memory image (and hence new page table entries pointing to the new image).

24. Which of the following pieces of information about the process are identical for a parent and the newly created child processes, immediately after the completion of the `fork` system call? Answer "identical" or "not identical".

    (a) The process identifier.

    (b) The contents of the file descriptor table.

    Ans: (a) is not identical, as every process has its own unique PID in the system. (b) is identical, as the child gets an exact copy of the parent's file descriptor table.

25. Consider the following sample code from a simple shell program.

```
command = read_from_user();
int rc = fork();
if(rc == 0) { //child
  exec(command);
}
else {//parent
  wait();
}
```

Now, suppose the shell wishes the redirect the output of the command not to STDOUT but to a file "foo.txt". Show how you would modify the above code to achieve this output redirection. You can indicate your changes next to the code above.

Ans:

Modify the child code as follows.

```
close(STDOUT_FILENO)
open("foo.txt")
exec(command)
```

26. Consider a simple program shown below. The OS uses a copy-on-write fork implementation. Indicate the line of code whose execution causes the OS to start making two separate copies of the memory image for the parent and child processes. Assume that the parent process is scheduled to run before the child after the fork system call.

    ```
    int a = 0;
    int rc = fork();
    if(rc == 0) { //child
      a = -1;
      exec(some_other_executable);
    }
    else {//parent
      a = 1;
      wait();
    }
    ```

    Ans: The line a = 1 in parent.

27. What is the output of the following code snippet? You are given that the `exec` system call in the child does not succeed.

    ```
    int ret = fork();
    if(ret==0) {
      exec(some_binary_that_does_not_exec);
      printf(``child\n'');
      }
    else {
      wait();
      printf(``parent\n'');
      }
    ```

    Ans:

    ```
    child
    parent
    ```

28. When a process makes a system call and runs kernel code:

    (a) How does the process obtain the address of the kernel instruction to jump to?

    (b) Where is the userspace context of the process (program counter and other registers) stored during the transition from user mode to kernel mode?

Ans:

(a) From IDT (interrupt descriptor table) (b) on kernel stack of process (which is linked from the PCB)

29. Consider a process P1 that is executing on a Linux-like OS on a single core system. When P1 is executing, a disk interrupt occurs, causing P1 to go to kernel mode to service that interrupt. The interrupt delivers all the disk blocks that unblock a process P2 (which blocked earlier on the disk read). The interrupt service routine has completed execution fully, and the OS is just about to return back to the user mode of P1. At this point in time, what are the states (ready/running/blocked) of processes P1 and P2?

    (a) State of P1

    (b) State of P2

    Ans:

    (a) P1 is running (b) P2 is ready

30. Consider the following code snippet, where a parent process forks a child process. The child performs one task during its lifetime, while the parent performs two different tasks.

```
int ret = fork();
if(ret == 0) { do_child_task(); }
else { do_parent_task1();
       do_parent_task2(); }
```

With the way the code is written right now, the user has no control over the order in which the parent and child tasks execute, because the scheduling of the processes is done by the OS. Below are given two possible orderings of the tasks that the user wishes to enforce. For each part, briefly describe how you will modify the code given above to ensure the required ordering of tasks. You may write your answer in English or using pseudocode.

Note that you cannot change the OS scheduling mechanism in any way to solve this question. If a process is scheduled by the OS before you want its task to execute, you must use mechanisms like system calls and IPC techniques available to you in userspace to delay the execution of the task till a suitable time.

(a) We want the parent to start execution of both its tasks only after the child process has finished its task and has terminated.

    Ans: Parent does wait() until child finishes, and then starts its tasks.

(b) We want the child process to execute its task after the parent process has finished its first task, but before it runs its second task. The parent must not execute its second task until the child has completed its task and has terminated.

    Ans: Many solutions are possible. Parent and child share two pipes (or a socket). Parent writes to one pipe after completing task 1 and child blocks on this pipe read before starting its task. Child writes to pipe 2 after finishing its task, and parent blocks on this pipe read before starting its second task. (Or parent can use wait to block for child termination, like in previous part.)

31. Consider a system with a single CPU core and three processes A, B, C. Process A arrives at $t = 0$, and runs on the CPU for 10 time units before it finishes. Process B arrives at $t = 6$, and requires an initial CPU time of 3 units, after which it blocks to perform I/O for 3 time units. After returning from I/O wait, it executes for a further 5 units before terminating. Process C arrives at $t = 8$, and runs for 2 units of time on the CPU before terminating. For each of the scheduling policies below, calculate the time of completion of each of the three processes. Recall that only the size of the current CPU burst (excluding the time spent for waiting on I/O) is considered as the "job size" in these schedulers.

   (a) First Come First Serve (non-preemptive).

   Ans: A=10, B=21, C=15. A finishes at 10 units. First run of B finishes at 13. C completes at 15. B restarts at 16 and finishes at 21.

   (b) Shortest Job First (non-preemptive)

   Ans: A=10, B=23, C=12. A finishes at 10 units. Note that the arrival of shorter jobs B and C does not preempt A. Next, C finishes at 12. First task of B finishes at 15, B blocks from 15 to 18, and finally completes at 23 units.

   (c) Shortest Remaining Time First (preemptive)

   Ans: A=15, B=20, C=11. A runs until 6 units. Then the first task of B runs until 9 units. Note that the arrival of C does not preempt B because it has a shorter remaining time. C completes at 11. B is not ready yet, so A runs for another 4 units and completes at 15. Note that the completion of B's I/O does not preempt A because A's remaining time is shorter. B finally restarts at 15 and completes at 20.

32. Consider the following code snippet running on a modern Linux operating systems (with a reasonable preemptive scheduling policy as studied in class). Assume that there are no other interfering processes in the system. Note that the executable "good_long_executable" runs for 100 seconds, prints the line "Hello from good executable" to screen, and terminates. On the other hand, the file "bad_executable" does not exist and will cause the `exec` system call to fail.

```
int ret1 = fork();
if(ret1 == 0) { //Child 1
  printf("Child 1 started\n");
  exec("good_long_executable");
  printf("Child 1 finished\n");
}
else { //Parent
  int ret2 == fork();
  if(ret2 == 0) { //Child 2
    sleep(10); //Sleeping allows child 1 to begin execution
    printf("Child 2 started\n");
    exec("bad_executable");
    printf("Child 2 finished\n");
  } //end of Child 2
  else { //Parent
    wait();
    printf("Child reaped\n");
```

```
        wait();
        printf("Parent finished\n");
    }
}
```

Write down the output of the above program.

Ans:

Child 1 started
Child 2 started
(Some error message from the wrong executable)
Child 2 finished
Child reaped
Hello from good executable
Parent finished

33. What are the possible outputs printed from this program shown below? You may assume that the program runs on a modern Linux-like OS. You may ignore any output generated from "some_executable". You must consider all possible scenarios of the system calls succeeding as well as failing. In your answer, clearly list down all the possible scenarios, and the output of the program in each of these scenarios.

```
int ret = fork();
if(ret == 0) {
  printf(``Hello1\n'');
  exec(``some_executable'');
  printf(``Hello2\n'');
} else if(ret > 0) {
  wait();
  printf(``Hello3\n'');
} else {
  printf(``Hello4\n'');
}
```

Ans: Case I: fork and exec succeed. Hello1, Hello3 are printed. Case II: fork succeeds but exec fails. Hello1, Hello2, Hello3 are printed. Case III: fork fails. Hello4 is printed.

34. Which of the following operations by a process will definitely cause the process to move from user mode to kernel mode? Answer yes (if a change in mode happens) or no.

    (a) A process invokes a function in a userspace library.
    Ans: no

    (b) A process invokes the `kill` system call to send a signal to another process.
    Ans: yes

35. Consider the following sample code from a simple shell program.

```
int rc1 = fork();
if(rc1 == 0) {
  exec(cmd1);
}
else {
  int rc2 = fork();
  if(rc2 == 0) {
  exec(cmd2);
  }
  else {
    wait();
    wait();
  }
}
```

(a) In the code shown above, do the two commands `cmd1` and `cmd2` execute serially (one after the other) or in parallel? **Ans:** parallel.

(b) Indicate how you would modify the code above to change the mode of execution from serial to parallel or vice versa. That is, if you answered "serial" in part (a), then you must change the code to execute the commands in parallel, and vice versa. Indicate your changes next to the code snippet above. **Ans:** move the first wait to before second fork.

36. What is the output printed by the following snippet of pseudocode? If you think there is more than one possible answer depending on the execution order of the processes, then you must list all possible outputs.

```
int fd[2];
pipe(fd);
int rc = fork();
if(rc == 0) { //child
  close(fd[1]);
  printf(``child1\n'');
  read(fd[0], bufc, bufc_size);
  printf(``child2\n'');
}
else {//parent
  close(fd[0]);
  printf(``parent1\n'');
  write(fd[1], bufp, bufp_size);
  wait();
  printf(``parent2\n'');
}
```

Ans: If child scheduled before parent: child1, parent1, child2, parent 2. If parent scheduled before child, parent1, child1, child2, parent2.