Lecture Notes on Operating Systems

Problem Set: 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.

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: T

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: BC

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: F

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: T

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

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

    Ans: F

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

    Ans: F

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

    Ans: F

13. When a process makes a system call to transmit a TCP packet over the network, which of the following steps do NOT occur always?

    A. The process moves to kernel mode.

    B. The program counter of the CPU shifts to the kernel part of the address space.

    C. The process is context-switched out and a separate kernel process starts execution.

    D. The OS code that deals with handling TCP/IP packets is invoked.

    Ans: C

14. 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.

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 regular fork (not a copy-on-write fork).

Ans: 0

16. Repeat the previous question for a copy-on-write fork implementation in the OS. Recall that with copy-on-write fork, the parent and child use the same memory image, and a copy is made only when one of them wishes to modify any memory location.

    Ans: 0

17. 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

18. 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

19. Consider a process that has requested to read some data from the disk and blocks. Subsequently, the data from the disk arrives and the interrupt is serviced. However, the process doesn't start running immediately. What is the state of this process at this stage?

    Ans: Ready/runnable

20. 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

21. 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

22. 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.