## CS347: Operating Systems

# Problem Set 1

1. Suppose a machine (architecture: x86, single core) has two runnable processes P1 and P2. P1 executes a line of code to read 1KB of data from an open file on disk to a buffer in its memory. The content requested is not available in the disk buffer cache and must be fetched from disk. Describe what happens from the time the instruction to read data is started in P1, to the time it completes (causing the process to move on to the next instruction in the program), by answering the following questions.

   (a) The code to read data from disk will result in a system call, and will cause the x86 CPU to execute the `int` instruction. Briefly describe what the CPU's `int` instruction does.

   (b) The `int` instruction will then call the kernel's code to handle the system call. Briefly describe the actions executed by the OS interrupt/trap/system call handling code before the read system call causes P1 to block.

   (c) Now, because process P1 has made a blocking system call, the CPU scheduler context switches to some other process, say P2. Now, the data from the disk that unblocks P1 is ready, and the disk controller raises an interrupt while P2 is running. Describe the actions performed by P2 in kernel mode when servicing this disk interrupt.

   (d) Right after the disk interrupt handler has successfully serviced the interrupt above, and before any further calls to the scheduler to context switch from P2, what is the state of process P1?

2. Consider the following lines of code in a program running on xv6.

```
int ret = fork();
if(ret==0) { //do something in child}
else { //do something in parent}
```

   (a) When a new child process is created as part of handling fork, what does the kernel stack of the new child process contain, after fork finishes creating it, but just before the CPU switches away from the parent?

   (b) How is the kernel stack of the newly created child process different from that of the parent?

   (c) The EIP value that is present in the trap frames of the parent and child processes decides where both the processes resume execution in user mode. Do both the EIP pointers in the parent and child contain the same logical address? Do they point to the same physical address in memory (after address translation by page tables)? Explain.

   (d) How would your answer to (c) above change if xv6 implemented copy-on-write during fork?

   (e) When the child process is scheduled for the first time, where does it start execution in kernel mode? List the steps until it finally gets to executing the instruction after fork in the program above in user mode.

(f) How does xv6 ensure that the return values from fork are different in the parent and child?

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

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

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

4. 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.

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

   (a) Is it necessary for threads in a process to have separate stacks?

   (b) Is it necessary for threads in a process to have separate copies of the program executable?

6. Can one have concurrent execution of threads/processes without having parallelism? If yes, describe how. If not, explain why not.

7. Is it necessary that every running process will saturate (i.e., result in a utilization close to 100% of) some hardware resource (e.g., CPU / memory / network etc.) at *all* times? Answer yes/no, and explain.

8. Consider a multithreaded webserver running on a machine with $N$ parallel CPU cores. The server has $M$ worker threads. Every incoming request is put in a request queue, and served by one of the free worker threads. The server is fully saturated and has a certain throughput at saturation. Under which circumstances will increasing $M$ lead to an increase in the saturation throughput of the server?

9. 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?

10. 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. Also, the OS implements copy-on-write during fork. 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.

11. Consider a socket program that has exactly one socket open for communication. The logic of the application is as follows: it calls `read` once on the socket by providing a buffer of size 1KB, prints out whatever has been read (if any), and repeats the read again in an infinite loop. It is known that the other remote end point of the socket rarely sends data. The designer of the application has the option of using a blocking or non-blocking socket. While of two choices will lead to a lower usage of CPU cycles by the application, and why?

12. Consider two threads that concurrently execute a line of code `count = count + 1`. The variable `count` starts out with a value 1, and ends up with a value of 2 after both threads have incremented it once each. Explain the sequence of events that could lead to such a situation.

13. 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?

**Answer** _____

**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.