CS503 Spring 2019 – Midterm (A)

- Write your **name** in the first page and your **initials in each page**.
- The exam is closed notes, closed book and no collaboration is allowed.
- Please print or write legibly.
- Provide complete but concise answers.
- Answer the questions in the spaces provided. Your answer does not need to take up all the space provided. If you run out of room, continue on the back.
- Provide clear answers and justify statements where required.
- Unless otherwise noted, assume that questions refer to C code, AT&T assembly syntax and the x86 architecture.
- Please check you have all pages (10 pages).
- You have 120 minutes to complete the exam.

Question:	1	2	3	4	5	6	7	8	9	10	Total
Points:	20	10	10	10	5	10	15	10	10	0	100
Bonus Points:	0	0	0	0	0	0	0	0	0	10	10
Score:											

Name (please print): _____

1. State whether the following statements are true of false. **Explain the false statements.**

   (a) (2 points) In C, "`sizeof(int)`" dynamically evaluates to the size in bytes of an integer when the code executes.

   ◯ True
   ◯ False. Explanation:

   Solution sketch: False. It evaluates statically.

   (b) (2 points) The assembly statement "`pusha`" emits a single machine instruction that saves the values of several registers, including the program counter (EIP register), to the stack.

   ◯ True
   ◯ False. Explanation:

   Solution sketch: False. It does not save the EIP register.

   (c) (2 points) Processes view their address spaces as an array of bytes

   ◯ True
   ◯ False. Explanation:

   Solution sketch: True.

   (d) (2 points) The Galileo boards used in the labs use a RISC (Reduced Instruction Set Computer) architecture.

   ◯ True
   ◯ False. Explanation:

   Solution sketch: False. They use a CISC architecture (x86 architecture also accepted).

   (e) (2 points) The statements "`int i = 10; printf("%d", i++);`" print the value 11.

   ◯ True
   ◯ False. Explanation:

   Solution sketch: False. It produces the value 10.

   (f) (2 points) The code of the shared libraries is not added to the program executable binary.

   ◯ True
   ◯ False. Explanation:

   Solution sketch: True.

   (g) (2 points) It is not possible to see the output of the preprocessor phase in gcc.

   ◯ True
   ◯ False. Explanation:

Solution sketch: False. It's possible.

(h) (2 points) The calling convention specifies how arguments are passed but not how return values are passed.

◯ True
◯ False. Explanation:

Solution sketch: False. It specifies both arguments and return values (and other aspects).

(i) (2 points) When the CPU runs in user mode, instructions can read all CPU registers but can only write to a subset of them.

◯ True
◯ False. Explanation:

Solution sketch: False. In this mode, instruction can only access (read and write) a subset of registers.

(j) (2 points) Software interrupts can be used by programs to request operating system services.

◯ True
◯ False. Explanation:

Solution sketch: True.

2. (10 points) What are the advantages of using multithreading? Give an example when multithreading is preferable to using single-threaded processes.

Solution sketch: Multiple execution units (or concurrency) lead to increased performance. Might be easier to program than multiple processing. Efficient use of SMP systems / better use of cores. Better resource sharing / low communication overhead. (Answer should mention at least 2)

3. (10 points) List the main criteria a (short-term) scheduling algorithm should satisfy in a timesharing system. Assume that a process had the following CPU bursts: 8, 32, and 16 microseconds. Compute the estimated length of the next CPU burst using: (a) simple and (b) exponential (e.g., weighted) averaging with parameter 0.5.

Solution sketch: Responsiveness. Efficiency and fairness get partial score. Simple: 19, Exp: 17. (Other values accepted due to rounding or other acceptable methods used.)

4. (10 points) Explain what is a deadlock. Explain which conditions are necessary to create a deadlock. Describe a general strategy to prevent deadlocks that could be implemented by application developers. Explain how this strategy provides that guarantee.

> **Solution sketch:** The system is not making any progress. Circular dependency. Involvement of 2 or more execution units (processes or threads). General Strategy: lock ordering or make sure circular dependency do not arise with DAG.

5. (5 points) Acquiring locks by spinning (i.e., busy waiting) might be more efficient than blocking in a multi-processor system. Explain why it may be more efficient in multi-processors. Why is this not the case in uni-processor systems? Explain.

> **Solution sketch:** Blocking causes a context switch which is inefficient. In SMP systems a context switch is not always necessary for progress (there are two or more CPUs) so busy waiting might be more efficient. In UP systems, context switch is always necessary for progress so busy waiting wastes resources.

6. Some operating systems provide pipes for IPC. A pipe is a channel for transmitting a stream of data, based on a fixed-size buffer (e.g., 4 kB). Reading an empty pipe blocks the process until data is written by another process. Writing to a pipe whose buffer is full blocks until data is read.

   (a) (5 points) Taking advantage of the blocking semantic of pipes, explain how to implement a semaphore using a pipe.

   Solution sketch: Implement wait() by reading one byte from the pipe. Implement signal() by writing one byte to the pipe. Initialize the pipe with the initial semaphore counter value. Reverse is also ok.

   (b) (5 points) A semaphore implemented as a pipe may cause deadlocks in situations where a conventional semaphore would not. Describe the situation.

   Solution sketch: When the semaphore counter exceeds the size of the pipe buffer (e.g., due to several signals()).

7. Ping and pong are two separate processes executing their respective functions. The code below is intended to cause them to repeatedly take turns, alternately printing "ping" and "pong" to the console.

```
1   void ping() {
2       while(true) {
3           suspend(ping_pid);
4           printf("ping\n");
5           resume(pong_pid);
6       }
7   }
8
9   void pong() {
10      while(true) {
11          printf("pong\n");
12          resume(ping_pid);
13          suspend(pong_pid);
14      }
15  }
```

(a) (5 points) The code shown above exhibits a synchronization flaw. Briefly outline a scenario in which this code would fail, and describe the outcome of that scenario.

Solution sketch: Pong executes resume(ping) and then suspends, then ping suspends: both threads hang. Only "pong" is printed.

(b) (5 points) Show how to fix the problem by replacing the suspend/resume calls and use instead the semaphore operations.

Solution sketch: Replace suspend() with wait() and resume() with signal(). Initialize correctly. Two semaphores. Solution needs to guarantee that "ping" and "pong" always alternate (e.g., cannot print two "pong" messages sequentially). [Common for synchronization problems: Correct solutions need to work in all possible schedules. See lecture slides for additional requirements for full grade.]

(c) (5 points) Implement ping and pong correctly using a mutex and condition variables.

Solution sketch: Only one lock, one conditional variable, and one variable with two states are required.

8. (10 points) Explain the importance of validating system call arguments. Describe three checks that the OS should conduct in the context of the `write()` system call when invoked on a file descriptor corresponding to a file.

Solution sketch: Validating system call arguments ensures correct OS semantics by preventing incorrect applications from causing the entire system from failing, for instance, by causing the kernel to crash due to a null pointer violation or illegal memory access. Checks include, for instance: fd is open, fd is writable, application buffer is valid memory.

9. (10 points) Explain how a buffer-overflow attack could allow an attacker to gain control of a computer in modern operating systems (e.g., Linux). List two system design characteristics that make such attacks possible.

Solution sketch: If the kernel does not check the boundaries of its own buffers, by executing a system call with specially crafted arguments an attacker may be able to trick the kernel into writing to memory outside of the kernel buffer thus controlling the values of other memory. This could enable an attacker to control important data structures (e.g., user id of process) or IP pointer of kernel, eventually causing the attacker to gain full control of the system. Design decision examples: use of low-level languages (e.g., C) that do not automatically check buffer boundaries, lack of formal verification of kernel properties, allowing iterations to depend on user supplied arguments, return address on stack, mixing data with code, etc.

10. (10 points (bonus)) Linux includes a system call *fork()* that duplicates a process, i.e., it creates a copy of the invoking process. Each process gets its copy of the data, bss, and stack. The *fork* system call is invoked once (by the initial process) but returns twice, once for the initial (parent) process and once for the new (child) process. Furthermore, it returns different values: (1) the PID of the child in the case of the parent process and (2) zero in the case of the child process. Explain if it is possible to implement the *fork()* function under userspace in Xinu, without changing the kernel. Otherwise explain what is the smallest change(s) you would need to make to the kernel. Either way, explain how you would implement fork in Xinu.

Solution sketch: Fork requires processes to have separate address spaces to ensure that both processes have all sections (with the same contents) *at the exact same addresses*. This ensures, for instance, that the previous contents in the stack, including pointers to other existing stack elements, remain valid in the child. This is the key design change to Xinu that would enable the implementation of fork semantics. [No partial points to answers that do not mention the key design change.] *Implementation details: To implement fork in userspace, the kernel should be modified to ensure that processes have separate address spaces and stacks of different processes start at the same address. With these kernel modifications, fork can be implemented in userspace by: (1) creating a new process, (2) copying the stack and globals (bss and data sections) to the new process using IPC, (3) resuming the new process, and (4) returning 0 to the child process and the child PID to the parent process.*