CS 502 Operating Systems
Craig E. Wills
Given: Wednesday, October 17, 2001

WPI, Fall 2001
Midterm Exam (100 pts)

NAME:_____

This is a closed book (and notes) examination. Answer all questions on the exam itself. Take the number of points assigned to each problem and the amount of space provided for your answer as a measure of the length and difficulty of the expected solution. The exam totals 100 points.

SCORE:_____

1. (6 points) Rather than a monolithic organization, some operating systems use a client-server (also called a microkernel) organization. What is it and why is it used? Are there any disadvantages?

2. (8 points) Some operating system systems provide kernel-level threads, which can be mapped to threads within a process. Threads can also be implemented as a user-level library, independent of any operating system support. For each of the following issues, specify *one* of four answers: 1) issue is best handled by kernel-level threads, 2) issue is best handled by user-level treads, 3) issue is a problem for both approaches, or 4) issue is not a problem for either approach.

   (a) Shared access to a global variable amongst a set of threads within the same process using a mutex lock.

   (b) Creating a thread within the same process.

   (c) Executing a multi-threaded process on a multiprocessor.

   (d) Protecting other threads within a process against a fault of one thread (such as a divide-by-zero).

3. (8 points) Briefly explain when mutual exclusion is needed within the operating system kernel. How can mutual exclusion be implemented in the kernel? Be sure your answer considers an operating system for both a uniprocessor and multiprocessor machine.

4. (12 points) Indicate whether each of the following situations describe a short process, a long process or if the situation does not clearly describe a short or long process. Briefly explain as necessary.

   (a) an editor application

   (b) a prime factor computation

   (c) an application reading from a database stored in a file and processing the records

   (d) process type favored by a round robin process scheduling policy

   (e) process type favored by a preemptive shortest process next scheduling policy

   (f) process type favored by a non-preemptive first-come, first-served scheduling policy

5. (10 points) In Project 1, you used the system calls *fork()* and *execve()* (or its variant *execvp()*).

   (a) Is it possible that a call to *fork()* can fail? If it cannot fail explain why, if it can fail give an example of when it would fail.

   (b) Is it possible that a call to *execve()* (*execvp()*) can fail? If it cannot fail explain why, if it can fail give an example of when it would fail.

   (c) A programmer has written the following simplified version of the *doit* program you wrote for Project 1. Explain all possible outcome(s) of running this code with different command line arguments.

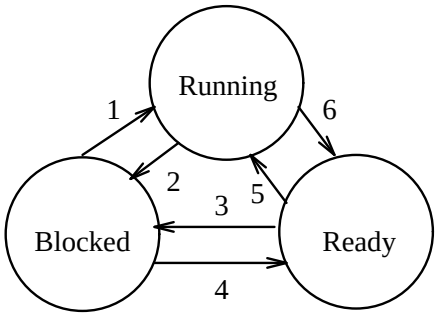
```
main(int argc, char **argv)
{
    fork();
    execvp(argv[1], &argv[1]);  /* shift arguments */
    printf("program finished\n");
}
```

6. (5 points) What is *thrashing* in the context of virtual memory management?

7. (10 points) Under each of the following process coordination scenarios, specify the number of semaphores needed and the initial counts of each semaphore. Explain your answers as necessary.

Scenario	Number of Semaphores	Initial Value of Each Semaphore
Provide mutual exclusion to a shared resource between two processes.		
A producer/consumer problem with two processes accessing a single shared variable.		
A producer/consumer problem with two producer processes producing values to be stored in a shared buffer of size *n* with a single consumer process consuming the values.		
A pipe-like mechanism between two processes where the pipe is a buffer of 4096 bytes.		
A readers/writers problem where any number of readers may read, but only one writer may write, at a time to a shared file. If a writer is writing to the file, no reader may read it.		

8. (10 points) As shown below, processes can be in one of three states: running, ready and blocked. There are six possible state transitions (labeled 1-6). For each label, indicate whether the transition is valid or not valid. If valid, indicate when the transition is used for a process. If the transition is not valid then indicate why.



State transitions:

(a) 1: Blocked to Running

(b) 2: Running to Blocked

(c) 3: Ready to Blocked

(d) 4: Blocked to Ready

(e) 5: Ready to Running

(f) 6: Running to Ready

9. (13 points) The page table for a process contains a number of fields: frame number, present bit, protection/permission, modified bit and referenced bit. Using these fields as needed, answer the following.

   (a) How is the physical address determined for a given virtual address?

   (b) When does a page fault occur?

   (c) What is a minor page fault?

   (d) What happens when a page fault occurs? Describe the steps from when the page fault occurs until the process is again ready for execution.

   (e) How is the modified bit used in paging?

   (f) How is the reference bit used in paging?

10. (8 points) Consider the two processes A and B in the following example with the global variable `x`, which is shared between the two processes. Assume that the routine `atomicprintf()` works just like `printf()` except it cannot be interrupted.

```
int x = 0;  /* global shared variable (not Unix!) */

ProcessA()
{
    int i;

    i = x;
    i = i + 2;
    x = i;
    atomicprintf("x = %d\n", x);
}

ProcessB()
{
    int i;

    i = x;
    i = i + 1;
    x = i;
    atomicprintf("x = %d\n", x);
}
```

(a) What is printed by the two processes in the example (give all possibilities)?

(b) What is the final value of `x` after both processes complete (give all possibilities)?

(c) What is a race condition?

(d) Is this situation an example of a race condition? If it is such an example, explain how to change the code to avoid a race condition. If it is not such an example, explain how to change the code to make it a race condition?

11. (10 points) One type of synchronization mechanism to coordinate the actions of a group of processes is a *barrier*. When a process within the group reaches a synchronization point (the barrier), it is blocked until all processes have reached the barrier. When all processes have reached the barrier then they are all allowed to proceed.

    Assume that *N* processes each call the routine *Barrier()* when they reach their synchronization point. This routine blocks the process until all *N* processes have reached the barrier, at which time the routine returns.

    Can this routine be implemented with semaphores? If so, show the code for the routine can be implemented with semaphores. If not, show how the code for the routine can be implemented with another synchronization primitive or explain why it cannot be implemented.