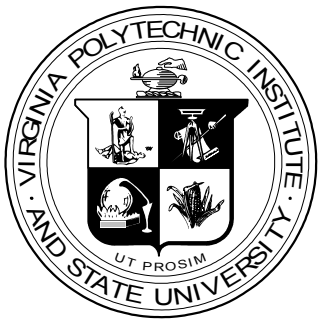


Instructions:

- Print your name in the space provided below.
- This examination is closed book and closed notes. No calculators or other computing devices may be used.
- Answer each question in the space provided. If you need to continue an answer onto the back of a page, clearly indicate that and label the continuation with the question number.
- If you want partial credit, justify your answers, even when justification is not explicitly required.
- There are xx questions, priced as marked. The maximum score is 100.
- When you have completed the test, sign the pledge at the bottom of this page and turn in the test.
- Note that either failing to return this test, or discussing its content with a student who has not taken it is a violation of the Honor Code.

Do not start the test until instructed to do so!

Name _____
printed

Pledge: On my honor, I have neither given nor received unauthorized aid on this examination.

_____
signed

1. [10 points] Consider two processes, P1 and P2, on a multi-tasking operating system with a single CPU. In each of the following situations would it be preferable for the OS to force a context switch and schedule another process? Explain why you reached your conclusion

   (a) P1 executes an input operation that requires reading data from a file on disk

   (b) P2 executes an arithmetic operation that involves a variable whose value is stored in primary RAM

   .

---

2. [10 points] Many modern multi-tasking operating systems use a quantum timer as part of the algorithm for scheduling processes. What is a quantum timer and how is it used?

3. [10 points] The scheduling algorithm in Windows 3.1 essentially waited for the running process to offer to be moved into the ready state. Describe at least two different (possibly related) problems with this approach?

4. Consider executing the two threads below, with the shared global variables that are shown.

```
int favoredThread =     1;  // shared globals
bool t1WantsIn    = false;
bool t2WantsIn    = false;
```

```
void* T1(void* Data) {
   . . .
   bool done = false;

   while ( !done ) {
      t1WantsIn = true;
      favoredThread = 2;
      while ( t2WantsIn &&
              favoredThread == 2 );

      // entering critical section now
         . . .
      // leaving critical section now
      // safe code follows
         . . .
   }
   pthread_exit(0);
}
```

```
void* T2(void* Data) {
   . . .
   bool done = false;

   while ( !done ) {
      t2WantsIn = true;
      favoredThread = 1;
      while ( t1WantsIn &&
              favoredThread == 1 );

      // entering critical section now
         . . .
      // leaving critical section now
      // safe code follows
         . . .
   }
   pthread_exit(0);
}
```

(a) [10 points] Does the design of the two threads guarantee that mutual exclusion will be achieved? If not, clearly describe an execution sequence that would lead to a violation.

(b) [10 points] Does the design of the two threads guarantee that deadlock will not occur? If not, clearly describe an execution sequence that would lead to a deadlock of the two threads.

5. [10 points] Aside from the technical distinction that `fork()` spawns a new process and `pthread_create()` spawns a thread, what is the most significant difference between them from a UNIX programmer's perspective? Assume that the `pthread` library will, in fact, be available on any UNIX platform.

---

6. [10 points] The simple program shown below creates a collection of threads that are intended to cooperate in the initialization of the elements of an array. Will the program always produce correct results, sometimes produce correct results, or never produce correct results? If not, explain how to fix it. If yes, explain why. Assume appropriate includes and declarations necessary for the code to compile and link; that is not the point.

```
struct Package {
      int* L;
      int  sIdx;
};
. . .
int main() {

   pthread_t hThread[MAX_THREADS];
   int* List = new int[OFFSET * MAX_THREADS];
   Package P[MAX_THREADS];

   for (int p = 0; p < MAX_THREADS; p++) {
      P[p].L    = List;
      P[p].sIdx = p;
      pthread_create(&hThread[p], NULL, F, (void*) &P[p]);
   }
   Print(cout, List); // print contents of array
   pthread_exit(0);
}

void* F(void* D) {

   Package* Data = (Package*) D;
   for (int Idx = Data->sIdx; Idx < OFFSET * MAX_THREADS; Idx += MAX_THREADS) {
      Data->L[Idx] = Data->sIdx;
   }
   pthread_exit(0);
}
```

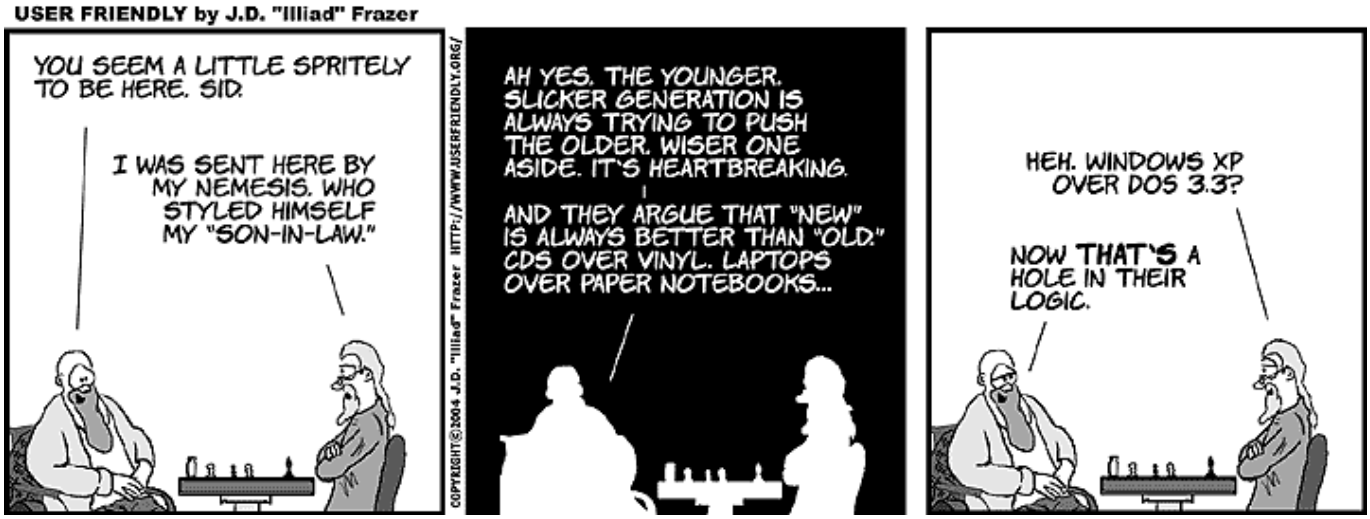
7. [10 points] Consider the simple run/ready/block state transition scheme for process scheduling. Draw the state diagram, showing all the possible transitions. Label each transition with a brief, precise description of an event that might cause a process to make that transition.

8. [10 points] The Linux task state-transition diagram includes a `zombie` state and a `dead` state. Evidently, some tasks might be placed in the `zombie` state before making a final transition to the `dead` state. Why is this a necessary feature (or at least a useful one)?

9. [10 points] If an interrupt occurs while a process is executing, it is possible the interrupt is either synchronous or asynchronous.

   Give an example of something that might cause a synchronous interrupt.

   Give an example of something that might cause an asynchronous interrupt.



Seemed appropriate somehow…. have a good break!