Computer Science Comprehensive Exam

Systems

May 22, 1997

This is a closed-book exam. You have 1.5 hours to complete it. Answer all of the questions. Be concise in your answers. You will receive partial credit for partially correct answers, but extraneous remarks may count against you.

1. Consider two processes $P1$ and $P2$, each having a critical section, with the requirement that access to their critical sections must be mutually exclusive. An adequate solution to this problem must have three properties:

   (a) **mutual exclusion:** at most one process at a time is executing its critical section.

   (b) **absence of deadlock:** if both processes are trying to enter their critical sections, at least one will succeed.

   (b) **absence of starvation:** whenever a process requires access to its critical section, it will eventually succeed in entering it.

   The following solution is proposed. Variables $need1$ and $need2$ are intended to declare when a process needs to enter its critical section. The variable $turn$ is intended to break "ties" in any race in which both processes try to simultaneously enter their critical sections.

```
bool need1 = false, need2 = false;
int turn = 2;

P1: while (1) {
        turn = 2;
        need1 = true;
        while (need2 == true && turn == 2);
        /* critical section */
        need1 = false;
        /* non-critical section */
    }

P2: while (1) {
        turn = 1;
        need2 = true;
        while (need1 == true && turn == 1);
        /* critical section */
        need2 = false;
        /* non-critical section */
    }
```

   You should assume that only individual load and store operations are guaranteed to be executed atomically.

   Question: Does the above code correctly solve the critical section problem? If it does, give a clear and convincing argument that properties (a), (b), and (c) hold. If it does not, give a counterexample, and then provide a correct version, along with a clear and convincing argument that properties (a), (b), and (c) hold.

2. In a *versioning* filesystem, each filename is augmented with a version number. For instance, `foo.txt;3` denotes the third version of the file `foo.txt`. Whenever a file is modified, its version number is automatically incremented. Previous versions of the file persist until they are explicitly deleted, and they can be accessed using their respective version number.

   (a) Discuss the advantages and disadvantages of a versioning filesystem. Which applications would likely benefit from such a filesystem? Would any applications suffer?

   (b) Describe how to *efficiently* implement a versioning filesystem. You may focus on discussing aspects that differ from the implementation of a standard UNIX filesystem. Be sure to consider if/how the on-disk data structures change, the procedure for reading and writing of a file, and any impact on the file cache. Does a versioning filesystem make it easier or harder to deal with cache and/or metadata consistency in the event of a system crash?

3. A *checkpointing* facility allows a process to take a "snapshot" of its present state on stable storage, and allows that process to later resume execution in the checkpointed state. A checkpointing facility is most frequently used to implement applications that can resume execution after a system crash or power failure. Specifically, the checkpointing facility adds two system calls to the operating system's API:

   (a) **checkpoint():** saves the current state of the invoking process to stable storage.

   (b) **resume():** resumes the invoking process's execution in the state that was most recently saved using the checkpoint operation.

   Sketch the design of a checkpointing facility for a UNIX-like operating system. Describe the actions taken during a `checkpoint` and `resume` system call, and all the involved data structures. You may assume that the system has a paged virtual memory implementation.