

# Homework #1

Due: 4 Feb 2011, 23:59 PST

Homework must be submitted electronically to <cs555@usc.edu>. It should have a subject of "Homework 1." You must submit plain text without embedded formatting commands or markup (ASCII or UTF-8 are acceptable). That means, among other things, no postscript, no Microsoft Word, no PDF, no FrameMaker, no TeX, no groff, no DocBook, no HTML, no XML, and no JavaDoc. Do not submit your homework as an attachment to your e-mail. Do not base 64 encode it. Do not Rot13 encode it. Plain text. You may PGP sign it, but are not required to do so. Do not PGP encrypt it. If you submit something that is not unmarked-up text, it is functionally the same as turning nothing in.

Homework turned in on the due date is not penalized, one day late is 25% off (that is the grade will be multiplied by 0.75), 2 days late is 50% off, and 3 days late is 75% off. No work will be accepted more than 3 days late. I will generally use the Date: line of the mail, but should the situation merit it, I am not above looking through mail system logs to confirm the submission time. I should not have to mention it, but forging a Date: line to avoid a late deduction is grounds for an F.

Do the work yourself. Computer science is a collaborative science, and I encourage you to talk over the ideas in the homework with other students. *However*, the final submission, that is, *the text of the homework*, must be composed individually by each student. If you hand in homework that is identical to another student, you risk failing the class. (In fact the only way that you would not fail the class in such a circumstance would be if one student had copied another student without the knowledge of the copied student; the copied student would not be penalized.) That is an awfully large risk for 10% of your total grade. Do the work yourself.

As with all work for cs555, this work is subject to the USC code of Student Conduct.<sup>1</sup> Read it, learn it, live it. Should you have any questions on how to apply the code, do not hesitate to contact me or the Office of Student Judicial Affairs and Community Standards.<sup>2</sup> Should it prove possible, do not plagiarize work from sources outside the class. Plagiarizing homework is grounds for *failing the class*. It is perfectly all right to properly cite external sources, should you find some that are useful.

Answers will not be graded on their beauty of expression. Answers will be graded on whether they show a logical approach and sensible explanation. Short, simple sentences are fine. What is important is that your ideas are clear to the reader, and that they answer the question. Of course, no answer will be penalized because it is beautifully expressed, either.

Each question has equal weight.

## Homework

1. Write a short evaluation of one of the papers that we read so far this semester. Your evaluation must include:
  - The most interesting or innovative aspect of the work
  - A major limitation of the work
  - An aspect of the related work that interests you

For each of these items you must justify your reasoning. Just saying that some aspect of the paper was interesting is insufficient. Provide some evidence for your opinion; unsupported

---

<sup>1</sup> <http://web-app.usc.edu/scampus/university-student-conduct-code/>

<sup>2</sup> <http://www.usc.edu/student-affairs/SJACS/>

statements are not worth credit. Support does not imply a study or some other work, and may include your personal experience. Of course if you cite specific work by others you should provide appropriate citations.

Answers that are simple restatements of the class slides are also worth no credit. The purpose of this question is to give you the opportunity to form, express, and support your opinions, not recall mine.

**Answer:** Each student will have their own answer here. The point of the question is to state and support opinions about one of the papers we've read in class.

**Grading:** Each the interesting aspect of this and related work is worth 3 points each. The limitation is worth 4.

2. In Section 1 of the Mesa paper[Lampson80] the authors discuss "pseudo-monitors" - cooperative multiprocessing where the language prevents `yield` calls inside procedures marked as monitors. (Cooperative multitasking means that a process that holds the processor must explicitly release it for another process to run, either by calling a `yield` system call or by exiting).

Objection (4) mentions that such cooperative multitasking does not work well in a virtual memory environment. Explain why a paged virtual memory system forces a process to yield the processor. (5 points)

From your understanding of the problem with virtual memory, it should be clear that there is another kind of system call that must be avoided in a pseudo-monitor. Explain what kind of call needs to be excluded and why. (5 points)

**Answer:** When a page fault occurs, a page may have to be brought in from disk, resulting in a long (to a processor) delay. That process is queued (put in a Waiting state) and another process run to keep the system from being idle. That's a forced yield of the processor.

These are unpredictable from the compiler's perspective. It cannot tell where page boundaries will be assigned, and even if it could on one machine, predicting it on many would be difficult.

Similar problems arise from any I/O system call that may block, which is certainly any that might lead to a disk I/O or network I/O, and these will have to be excluded. Fortunately, those are visible to the compiler.

Now, one can argue that the system can be made correct by simply idling the processor while disk I/O is happening (with the exception of the interrupt routines needed to run the disk, etc.) but this removes a significant incentive to multiprogramming in the first place.

**Grading:** 5 for explaining the page fault situation and 5 for generalizing to the I/O syscall.

3. Consider the implementation of multiple threads per address space in a desktop operating system. There are (at least) two ways to implement threads: in the kernel and in the user process.

Kernel level threads are scheduled and manipulated by the kernel. Each kernel thread can initiate I/O and be assigned different scheduling parameters.

User level threads are implemented entirely in the address space. Context switches between threads in the address space are implemented by user code. User level threads run only when the overall kernel process is scheduled, and only one I/O command can be issued from the kernel process at a time (because the kernel has to queue the process object).

Assuming that the cost of kernel implementation is not very high, kernel threads generally seem to be more flexible and powerful. What is a general rule for determining when user threads are

required for an application? (5 points) Suggest an example that would lead you to implement some or all of a threading package in user space, even when on a kernel that implements kernel threading. (5 points)

**Answer:** The principle is that if there is an aspect of thread semantics that can only be expressed in application terms, it must be implemented in the application. That's pretty much a restatement of the end-to-end principle[Saltzer81].

Coming up with such semantics is a little tricky. Consider a simulation where two threads need to be scheduled in such a way that events from one appear twice as often as events from another, but where the generation of an event is not directly related to CPU time. That would need a scheduler that was cognizant of application semantics, and that scheduler would have to be implemented in the application.

Other choices are possible, but because the main thing that the kernel does with respect to threads is schedule them, that is the easiest place to look for such application dependent semantics.

**Grading:** 5 points for realizing that this is an end-to-end question and citing that rule and 5 for arguing successfully that the student has identified such a feature.

4. Explain how you would implement a bounded buffer of integers that can be shared by many processes using Linda[Carriero85]. Describe the data structures in the tuple space and the implementation of `enqueue` and `dequeue`. subroutines. Calling `enqueue` on a full queue or `dequeue` on an empty one should block until they are meaningful. (You are not required to make any guarantees on the ordering in which blocked `enqueue` and `dequeue` operations occur.)

Describe any initialization of the tuple space. If you need additional processes, describe them as well. You can use either pseudocode or a description of the algorithms, but either should be clear enough that one can implement fairly directly.

**Answer:** The queue state is in two tuples one of the form (`"state"`, `size`, `head`, `tail`) where `head` and `tail` are integers representing the index of the head and tail elements in the queue respectively. The queue is empty if `head` is equal to `tail` and full when the difference between the two is at least `size`. When a process has removed the `"state"` tuple from the space, it holds a lock on the queue.

Another tuple (`"changed"`) is put into the tuple space when the queue state changes as a hint. The queue contents are a set of tuples of the form (`"queue"`, `value`, `index`) where `value` is the value stored at `index` in an infinite array. The `head` and `tail` values in the `"state"` tuple are index values into that array.

The tuple space is initialized with (`"state"`, `SIZE`, `0`, `0`) where `SIZE` is the size of the queue.

```
void enqueue(int val) {
    in("state", int s, int h, int t)
    while ( t - h >= s ) {
        out("state", s, h, t)
        in("changed")
        in("state", int s, int h, int t)
    }
    out("queue", val, t)
    out("state", s, h, t+1)
```

```

    out("changed")
}

int dequeue() {
    in("state", int s, int h, int t)
    while ( t == h) {
        out("state", s, h, t)
        in("changed")
        in("state", int s, int h, int t)
    }
    in("queue", int val, h)
    out("state", s, h+1, t)
    out("changed")
    return val
}

```

The ( "changed" ) tuple is just a hint, and the statements containing it can be removed. It does reduce contention in that the processes waiting for a queue change will not be busy waiting in the while loop.

This solution assumes only one queue, but adding a name to the queue tuples and the subroutines allows many such queues to coexist.

**Grading:** That answer, or one like it in spirit with a decentralized queue is worth full credit. A (correct) solution with a process implementing the queue and a conventional lock is only worth 7 points. There's some discretion in grading incorrect implementations, but nothing that fails should be worth more than 5, and decentralized failures are worth more than centralized ones.

## References

- [Saltzer81] J. H. Saltzer, D. P. Reed, and D. D. Clark, "End-to-end Arguments In System Design," *ACM Transactions on Computer Systems*, ACM Press, vol. 2, no. 4, 1984, 277-288,
- [Lampson80] B.W. Lampson and D.D. Ridell, "Experiences with Processors and Monitors in Mesa," *Communications of the ACM*, vol. 23, no. 2, February 1980, 105-117,
- [Carriero85] Nicholas Carriero and David Gelernter, "The S/Net's Linda Kernel," *ACM Transactions on Computer Systems*, ACM Press, vol. 4, no. 2, 1986, 110-129,