

Exam 1 REDO Solutions

CS 4210 Advanced Operating Systems

Fall 1999 • Georgia Tech/Computer Science • Hutto

DUE: 3:05 pm (classtime) Thursday 14 October 1999

To improve your Exam 1 score and to reinforce your understanding of the material you may submit a detailed Solution Key to the exam questions. You may consult any material but you must prepare this solution key on your own. You should also answer the additional questions below. Again, you may consult any printed material but do the work on your own. Correct submissions will earn up to 70% of the points marked WRONG on your original Exam 1. For example, if you got 20 points wrong on the original exam, you can earn up to 14 additional points by completing this assignment.

1. Pthreads supports a feature we didn't discuss called "thread specific data" (TSD). How do you think this is implemented?

Well, the data will live in the process address space along with all other thread-related data but it is part of a thread's "context". When we schedule a new thread we need to somehow make this data segment "available" to the current thread. The Lewis & Berg book has a short chapter dedicated to this topic (chapter 8, pages 129-136). Under Solaris the global register g7 is reserved in threaded programs. It contains the address of the current "thread control block". Scheduling a new thread basically involves changing g7 to point to the new thread and setting a few other registers like the program counter. Under Solaris, the thread control block contains a pointer to any thread specific data that has been allocated. The actual POSIX calls for manipulating TSD are kind of ugly and use a notion of keys (something like the System V IPC). TSD is implemented similarly under Win32 and OS2.

2. What capabilities does the Psyche system provide to allow different thread implementations to interact or cooperate?

Psyche maintains a table of thread manipulation functions for each process using a threads package. "Foreign" thread packages can interact using these "well-known" interface routines. This is basically how device drivers are handled. The OS maintains a table of pointers to "generic" io functions like read and write. Each device installs its own "handler" routines that encapsulate the device specific processing.

3. How do Pthread condition variables and mutex locks differ from Hoare's monitor concept? Are they the same? Is one more general than the other? If so, how?

Pthread cvs and mutexes are generalizations of Hoare's monitor concept. Once again, we have "ripped open" Hoare's monitor and exposed some of the internal components. This gives the programmer the ability to compose the primitives in new ways, yielding variations on the original concept. It also makes things a bit more complex and potentially, more efficient. We could use a term popular in the humanities and say we have "deconstructed" Hoare's monitor into more primitive components and exposed those to the programmer. One specific example: monitors provide a single mutex that is automatically associated with all the condition variables defined within that monitor. In Pthreads, the programmer gets to define the association between cvs and mutexes.

4. What's the point of light-weight processes (LWPs) in the Solaris operating system?

LWPs provide a level of indirection between user-level threads and processors. They are “virtual processors”. By mapping threads to LWPs and LWPs to processors, we can “tune” the amount of “processor resource” that a given process receives. Without LWPs we have essentially two choices: each process gets one processor or each process gets one processor per thread.

5. The UNIX malloc routine has been made thread-safe under Solaris with a single global mutex lock. This works but doesn't provide much parallelism. Sketch (outline) a design for making malloc more efficient by decreasing the lock granularity. (In Solaris lingo, they call this making the routine “mt-hot”.)

This is a fun question. I stole it from Lewis & Berg. See pages 210-213 (chapter 13). One simple approach is to reserve separate sections of the heap for each thread. In essence, each thread has its own little heap. Accesses to these heaps can be completely concurrent since they aren't shared. This isn't a great idea, however. What happens if one thread runs out of heap and the others never use theirs?

A second approach would be to associate a mutex lock with each free block on the free list. When a malloc request comes in, it traverses the free list, looking for a free block that is “big enough”. If a block is locked, simply skip to the next one. This is Ok but is probably too “fine-grained”.

An interesting hybrid solution is to allocate some fixed number of “malloc regions” (less than the number of threads). Associate a lock with each “region”. When a thread needs storage, try the first region. If it is locked, go on to the next. The number of held locks corresponds to the number of simultaneous malloc requests. The probability of a region being “busy” becomes very small as you go further down the list. If there is no contention, then you just need to touch one lock. Very nice. Basically threads are “sharing” multiple malloc regions instead of having one statically allocated for them.