CS 6420 Operating Systems **Summer Quarter, 1999**

Midterm Answer Key

Wednesday July 22, 1998

1. Anderson's Array Based Queueing Lock. (*20 points*)
   (a) The *nextslot* variable has a range of 1 to 8.
   (b) The *myplace* variable has a range of 0 to 7.
   (c) The *slots* variable can definitely go to [F,F,F,F]. This indicates that a processor has obtained the lock, has set it's place to F (inside `acquire_lock` at the last line), but net yet called `release_lock`.
   (d) The *slots* variable can NOT go to [T,T,T,T]. A true value in the slots array indicates that whatever process gets the corresponding place on the next entry to `acquire_lock` can have the lock without waiting. Clearly, this can only be true for one processor, otherwise all processors could think they have the lock at the same time. This of course assumes only processors holding the lock will call the `release_lock` procedure.

2. Barriers _______ (*out of 20 points*)
   (a) BuggyBarrier2 is erroneous due to the way it spins at line 7. The last processor to enter the barrier will clear the count at line 5, presuming that all other processors will notice this and stop spinning at line 7. The problem is that there is some chance that not all other processors will notice this right away. Then there some chance that some other processor will enter BuggyBarrier2 again, incrementing the count to a non-zero value before all others have noticed it went to zero.
   (b) Then sense-reversing barrier corrects the problem by spinning differently (at line 13). It spins on a variable (*sense*) which only changes when ALL processors have entered the lock, not when any single processor enters. Thus even if we have the situation above, processors entering the barrier before all others have exited does not create a problem.
   (c) We can in fact have one or more processors running in the barrier with $BIN = k$ at the same time other processors are in the barrier with $BIN = (k-1)$. This is precisely the situation described in (a) above, that all processors have arrived at the barrier with $BIN = (k-1)$, some have left and re-entered (with $BIN = k$) but others have not left yet.
   (d) No, this situation cannot happen. This would mean that some one (or more) processor is stuck in the barrier with $BIN = (k-2)$, while some one or more other processors have entered AND LEFT the barrier with $BIN = (k-1)$, and re-entered again with $BIN = k$. Then cannot have left the $k-1$ invocation while one or more processors at stuck at $k-2$.

3. Filaments _______ (*out of 20 points*)
   (a) The non-atomic increment of k at line 3 is NOT an error. The design of the filaments package is such that only server 0 will execute the sequential code. Since only one CPU can possibly be executing this procedure, an atomic increment is not necessary.

(b) The answer to this can be found almost verbatim in the paper. The paper gives code for `f_rtc_thread`, not `f_iterative_thread` but there is no difference. You could also say that the code is the same as `GThread_create` if you point out you don't need to allocate a private stack.

4. Remote Procedure Calls _______ (*out of 20 points*)

   (a) Reply blocks are buffered to handle the case of them being lost on the network and not arriving at the client. The client will eventually ask for the RPC again, but the key point is we DON'T to execute the remote call again. We just want to send the previous reply again.

   (b) They are buffered by the Server RPC Runtime.

   (c) They can be discarded either when a new RPC (with a different CallID) is received from the same client, or an acknowledgement is recieved from the client for that reply.

   (d) The nonce value Y allows the server to be sure that the reply to the RFA message in fact came from some system that already knows the conversation key C_k, which presumably can only be the client. When the server receives the RFA reply, one of the fields is the Y value encrypted with C_k. If the server decrypts it with this C_k it should be the same Y that was send in the original RFA message.

   (e) If my scheme were used, the KDC would be stuck with remembering the conversation keys that it handed out indefinitely, since it can't know how long will elapse before B would eventually ask for it. Secondly, Birrell's scheme puts less load on the KDC than my scheme, thus helping to reduce a potential bottleneck in the KDC.

5. Active Messages _______ (*out of 20 points*)

   (a) Active Messages don't have to wait for a server thread to schedule to respond to the request, in that they already have a CPU assigned (by virtue of the ISR). They don't have to do a second context switch to the server thread. They don't have to copy the data twice, (once to kernel memory and again to user memory), but instead can copy directly to the correct location in user memory.

   (b) Active Messages cannot block, since they are running in the context of a ISR. Also, they cannot run for a "long" time, for the same reason. Optimistic Active Messages solve these two by assuming that the code for the AM will NOT block and NOT run for a long time. Then runtime checks embedded in the AM code will check for these conditions, abort the Active Message, and demote it to a normal RPC call. We optimistically hope that the good case (not blocking and not taking too long) happens far more often than the bad case.

   (c) As the number of requests goes up, the output queues on the network interfaces will grow and eventually fill up. Then, the bad news is that we will have successfully processed the AM (without blocking or running too long), but then find that we have to block because the output queue is full on the network write. So we have to abort, demote to a normal RPC call after having already done all of the associated work.