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

Midterm

Friday July 16, 1998

Reminder: OPEN Book and OPEN Notes

1. Anderson's Array Based Queueing Lock(*20 points*)

   Anderson's array based queueing lock is shown below, with Mellor-Crummey's correction, and Riley's correction to Mellor-Crummey's correction. This version assumes *numprocs* is 4, meaning we have exactly four CPU's which can be contending for the lock. The pseudo-code is given in a C-like format, rather than the Pascal-like format in the paper. Pay careful attention to the initialization of the fields in the Lock structure, which is slightly different than that given in the paper. *Think carefully about parts a and b.*

   (a) As coded below, what is the smallest and largest (ie. the range of values) that will ever be assigned to variable *Lock.nextslot*?

   (b) As coded below, and remembering that each processor has a private copy of variable *myplace*, what is the smallest and largest (ie. the range of values) that any instance of variable *myplace* can be assigned?

   (c) Variable *Lock.slots* is shown to be initialized to [*T,F,F,F*]. During normal execution of the algorithm, is it possible that *Lock.slots* will have the value [*F,F,F,F*]? If so, explain what this means, or if not explain why not.

   (d) Is it possible for variable *Lock.slots* to have the value [*T,T,T,T*]? Is so, explain what this means, or if not explain why not.

```
#define F 0
#define T 1
#define numprocs 4
typedef struct {
  int slots[numprocs] = {T, F, F, F};
  int nextslot = numprocs;
} Lock;
void acquire_lock( Lock* L, int* myplace);
{
  *myplace = fetch_and_increment(&L->nextslot);
  if ((*myplace % numprocs) == 0) /* % is the mod operator */
    atomic_add(&L->nextslot, -numprocs);
  *myplace = *myplace % numprocs;
  while(L->slots[*myplace] == F) spin;
  L->slots[*myplace] = F;
}
void release_lock(Lock* L, int* myplace)
{
 L->slots[(*myplace + 1) % numprocs] = T;
}
```

2. Barriers (*20 points*)

The code for *BuggyBarrier2* that we discussed in class, along with the *Sense Reversing Centralized Barrier* (Mellor-Crummey algorithm 7, slightly modified) is shown below.

(a) What is the problem with *BuggyBarrier2*? Explain in detail why it cannot work.

(b) The Sense Reversing barrier is quite similar to *BuggyBarrier2*, but it is in fact correct. Explain how the Sense Reversing barrier corrects the problem with *BuggyBarrier2*.

(c) Notice that the Sense Reversing barrier shown below has added a processor private variable *Barrier Invocation Number* (*BIN*), which simply counts by one each time the *Central_Barrier* routine is entered. Is it possible for some processor to be executing in routine *Central_Barrier* with $BIN = k$ $(k \geq 2)$ at the same time as some other processor in also executing in routine *Central_Barrier* with $BIN = (k-1)$? If so explain how this can happen, or explain why not.

(d) Is it possible for some processor to be executing in routine *Central_Barrier* with $BIN = k$ $(k \geq 2)$ at the same time as some other processor in also executing in routine *Central_Barrier* with $BIN = (k-2)$? If so explain how this can happen, or explain why not.

Algorithm BuggyBarrier2, by George Riley

```
shared int CountBarrier = 0;
Procedure BuggyBarrier2
   mycount = FetchAndIncrement(CountBarrier);
   if(mycount == (numprocs - 1)) {
     CountBarrier = 0; // All there, let others know and reset
   else
     while(CountBarrier != 0) spin // Wait for others
```

Sense-Reversing Centralized Barrier

```
shared int CountBarrier = P;
shared Boolean sense = TRUE;
processor private Boolean local_sense = TRUE;
processor private int BIN = 0;
Procedure Central_Barrier
   BIN = BIN + 1;
   local_sense = NOT  local_sense;
   mycount = FetchAndDecrement(CountBarrier);
   if(mycount == 1) {
     CountBarrier = P; // All there, reset count for next pass
     sense = local_sense; // All there, let others know
   else
     while(Sense != local_sense) spin // Wait for others
```

3. Filaments (*20 points*) For this question, assume we are running on a platform with 4 CPU's, and we are creating 4 servers in our filaments code (ie. we are calling `f_initialize(4)`.

   (a) The code for `sequential_code` on page 5 is reproduced below. Notice that on line 3 the variable k is incremented, but not atomically. Keeping in mind that we have defined 4 servers, it this an error? Should we have used an atomic increment here? Explain why or why not.

   (b) Refering to the main program at the bottom on page 5, give pseudo code or a verbal explanation of what the subroutine `f_iterative_thread` has to do to work properly.

```
1  sequential_code()
2    real** temp;
3    k++
4    if (k > MAXITERS or maxdiff < EPSILON) then return DONE
5    temp = old; old = new; new = temp;
6    maxdiff = 0.0
7    return NOTDONE
8  end
```

4. Remote Procedure Calls (*20 points*)

   (a) Why is it necessary for the RPC Server to buffer the *Reply* blocks for possible later reuse?

   (b) Of the RPC server application, RPC server stub, or RPC server runtime (see fig. 1 in the RPC paper), which of these does the buffering of the Replies?

   (c) When can these buffered reply blocks be discarded?

   (d) What is the purpose of the random value Y in the RFA message shown in fig 2 of the Secure RPC paper?

   (e) Assume that an RPC client is system A and the RPC server is system B. The protocol for the *Request for Authenticator RFA* message between B and A is somewhat complex. It would seem simpler just to have B ask the KDC for the conversation key (which would of course be given encrypted with B's private key). Give two reasons why Birrell did not design it this way.

5. Active Messages (*20 points*)

   (a) Give two reasons why an implementation of RPC's using active messages can perform so much better than traditional RPC's.

   (b) Explain two problems with the original design of active messages that the *Optimistic* active messages design is attempting to solve. How does it solve them?

   (c) Why does the performance of *Optimistic* active messages drop off so dramtically as the number of processes increases above a certain threshold (see figure 2 in the Wallach paper).