Final *Solutions*

CS 414 Operating Systems and Systems Competency Exam, Spring 2007
May 16th, 2007
Prof. Hakim Weatherspoon

Name (or Magic Number): _____NetId/Email:_____

Read all of the following information before starting the exam:
If you are a CS 414 student, write down your name and NetId/email NOW. Otherwise, if you are taking this as your Systems Competency Exam, write down your *Magic Number* (do ***NOT*** write your name, NetId, or Email).

This is a **closed book and notes** examination. You have 150 minutes (2 ½ hours) to answer as many questions as possible. The number in parentheses at the beginning of each question indicates the number of points given to the question; there are 100 points in all. You should read **all** of the questions before starting the exam, as some of the questions are substantially more time consuming.

Write all of your answers directly on this paper. ***Make your answers as concise as possible.*** If a question is unclear, please simply answer the question and state your assumptions clearly. If you believe a question is open to interpretation, then please ask us about it!

Good Luck!!

Problem	Possible	Score
1	20	
2	24	
3	15	
4	21	
5	20	
Total	**100**	

1. (20 points) Synchronization/Concurrency Control

For the following implementations of the "H20" problem, say whether it either (i) works, (ii) doesn't work, or (iii) is dangerous -- that is, sometimes works and sometimes doesn't. If the implementation does not work or is dangerous, explain why and show how to fix it so it does work.

Here is the original problem description: You have just been hired by Mother Nature to help her out with the chemical reaction to form water, which she does not seem to be able to get right due to synchronization problems. The trick is to get two H atoms and one O atom all together at the same time. The atoms are threads. Each H atom invokes a procedure *hReady* when it is ready to react, and each O atom invokes a procedure *oReady* when it is ready. For this problem, you are to write the code for *hReady* and *oReady*. The procedures must delay until there are at least two H atoms and one O atom present, and then one of the threads must call the procedure *makeWater* (which just prints out a debug message that water was made). After the *makeWater* call, two instances of *hReady* and one instance of *oReady* should return. Your solution should avoid starvation and busy-waiting.

You may assume that the semaphore implementation enforces FIFO order for wakeups—the thread waiting longest in P() is always the next thread woken up by a call to V().

(a) (10 points) Here is a proposed solution to the "H20" problem:

```
Semaphore h_wait = 0;
Semaphore o_wait = 0;
int count = 0;

hReady()
{
    count++;
    if(count %2 == 1) {
        P(h_wait);
    } else {
        V(o_wait);
        P(h_wait);
    }

    return;
}
```

```
oReady()
{
    P(o_wait);
    makeWater();
    V(h_wait);
    V(h_wait);

    return;
}
```

This solution is dangerous. Threads calling hReady() *access shared data without holding a lock! For example, you could have N threads increment* count*, and because they do so without a lock, the result could be 1 instead of N -- in other words, no water would be made regardless of how many* H*'s arrived.*

The solution is to put a lock acquire before the first line in hReady*, and release before the first* P(h_wait) *and after the second* P(h_wait)*. Some put the lock acquire after the increment, and that simply doesn't work!*

(b) (10 points) Another proposed solution to the "H20" problem:

```
Semaphore h_wait = 0;
Semaphore o_wait = 0;

hReady()
{
    V(o_wait)
    P(h_wait)

    return;
}
```

```
oReady()
{
   P(o_wait);
   P(o_wait);
   makeWater();
   V(h_wait);
   V(h_wait);

   return;
}
```

This is dangerous, since it may lead to starvation. If two H*'s arrive, then the value of the* o_wait *semaphore will be 2. If two* O*'s arrive, then they can each decrement* o_wait*, before either can decrement it twice. So no water is made, even though enough atoms have arrived.*

The fix is to put a lock acquire before the first line in oReady*, and a lock release after the two* V(h_wait)*'s. This way, only one oxygen looks for waiting* H*'s at a time -- if there aren't enough* H*'s for the first oxygen, there won't be enough for any of the later oxygens either.*

2. (24 points) Synchronization via Monitors

Some monkeys are trying to cross a ravine. A single rope traverses the ravine, and monkeys can cross hand-over-hand. Up to five monkeys can hang on the rope at any one time. If there are more than five, then the rope will break and they will all fall to their end. Also, if eastward-moving monkeys encounter westward-moving monkeys, all will fall to their end. (This is the same problem setup from an earlier assignment, but the synchronization mechanism differs).

Assume that monkeys are processes.
(a) (18 points) Write a *monitor* with two methods WaitUntilSafeToCross(Destionation dst) and DoneWithCrossing(Destination dst). Where Destination is an enumerator with value EAST=0 or WEST=1.
(b) (6 points) Does your solution suffer from starvation? If so, briefly explain (e.g. give a sequence). Otherwise, simply state *starvation-free*. In either, case state your assumptions, if any.

The solutions below are starvation-free.

```
int crossing[2] = {0, 0}, waiting[2]= {0, 0};
Condition wantToCross[2];

WaitUntilSafeToCross(Destination dst)
{
        if(crossing[!dst] > 0 || waiting[!dst] > 0 || crossing[dst] == 5)
        {
                ++waiting[dst];
                wantToCross[dst].wait();
                --waiting[dst];
        }
        crossing[dst]++;
}

DoneWithCrossing(Destination dest)
{
        --crossing[dst];
        if (crossing[dst] == 0)
                wantToCross[!dst].signal();
        else if (waiting[dst] > 0 && waiting[!dst] == 0)
                wantToCross[dst].signal();
}
```

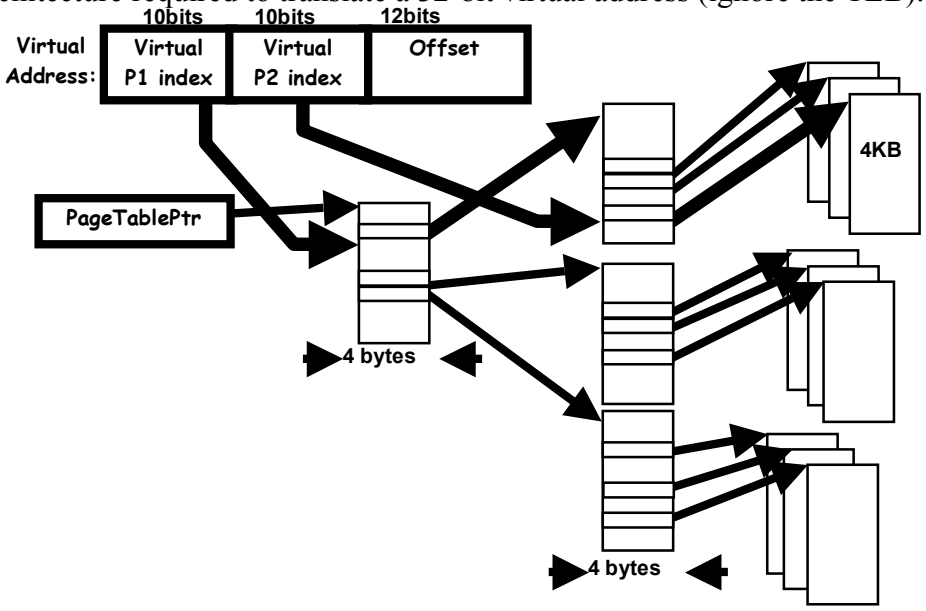
Alternative implementation

```
int wcrossing=0, ecrossing=0, wwaiting=0, ewaiting=0;
Condition wantToCrossWest, wantToCrossEast;

WaitUntilSafeToCross(Destination dest)
{
        if(dest == EAST)
                WantToGoEast();
        else
                WantToGoWest();
}

DoneWithCrossing(Destination dest)
{
        if(dest == EAST)
                DoneGoingEast();
        else
                DoneGoingWest();
}
WantToGoEast()
{
        if(wcrossing > 0 || wwaiting > 0 ||
          ecrossing == 5)
        {
                ++ewaiting;
                wantToCrossEast.wait();
                --ewaiting;
        }
        ecrossing++;
}

DoneGoingEast()
{
        ecrossing--;
        if (ecrossing == 0)
                wantToCrossWest.signal();
        else if (ewaiting>0 && wwaiting==0)
                wantToCrossEast.signal();
}
```

```
WantToGoWest()
{
        if(ecrossing > 0 || ewaiting > 0 ||
          wcrossing == 5)
        {
                ++wwaiting;
                wantToCrossWest.wait();
                --wwaiting;
        }
        wcrossing++;
}

DoneGoingWest()
{
        wcrossing--;
        if (wcrossing == 0)
                wantToCrossEast.signal();
        else if (wwaiting > 0 && ewaiting==0)
                wantToCrossWest.signal();
}
```

3. (15 points) Virtual Memory and Paging

a. (5 points) Suppose that we have a two-level page translation scheme with 4K-byte pages and 4-byte page table entries (includes a valid bit, a couple permission bits, and a pointer to another page/table entry). What is the format of a 32-bit virtual address? Sketch the paging architecture required to translate a 32-bit virtual address (ignore the TLB).



Grading Scheme: 2 point for the virtual address format (1/2 for [L1 page index, L2 page index, offset] and 1/2 point for # bits = 10,10,12) , 1 point for picture of single level page table,1 point for multiple page tables linked together correctly, and 1 point for the page table base register (PageTablePtr).

b. (10 points) Assume that a process has referenced a memory address not resident in physical memory (perhaps due to demand paging), walk us through the steps that the operating system will perform in order to handle the page fault.

Note: We are only interested in operations that the OS performs and data structures modified by the OS.

1. Trap to the OS
2. Save the process state (program counter, stack pointer, registers, etc)
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on disk
In particular, if the reference was invalid, terminate the process. If it was valid, but have not yet brought in that page, page it in.
5. Find a free frame (by taking one from the free-frame list, for example)
a. If there is a free frame, use it.
b. If there is no free frame, use a page-replacement algorithm to select a victim *frame.*
c. Write the victim frame to disk (if it was dirty)
i. Change the page table to mark the frame invalid (i.e. not present in memory).

6. Issue a read from the disk to the free frame:
a. Wait in a queue for this device until the read request is serviced
b. Wait for the device seek and/or latency time
c. Begin the transfer of the page to the free frame
7. While waiting, allocate the CPU to some other process
8. Receive an interrupt from the disk I/O subsystem (I/O completed)
9. Save the registers and process state for the other process (if step 7 is executed)
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show that the desired page is now in memory
11. Add the process to the ready queue
12. When process selected to run, restore user registers, process tate, and new page table, and then resume the interrupted instruction.

4. (21 points) File Systems

Adding Links to a File System. This design question asks you to consider adding links to a file system that does not have any linking mechanism. ***This is a design question; you should not write code implementation unless stated explicitly.***

a. (12 points) The first set of questions is about adding **hard** links to the file system.

i. (3 points) What changes would you make to the internals (directory, fileheader, free map, etc.) of the filesystem to support hard links?

You would have to add a link count to the file header. No partial credit.

ii. (3 points) How do the semantics of the Remove system call change, and how do you implement that change to Remove and any other affected system calls?

Remove must decrement the link count (1 points) and check the on disk link count (2 points). No other system calls are affected and we deducted up to 2 points if you included other system calls.

iii. (3 points) What new system calls, if any, must be added to the system? Give the C language-style signature of any new calls, e.g., int Open(char *file).
List the signature and give a one sentence definition of each arg and return value.

*Add a int Link(char *src, char *dst) or int Link(int inodeNumber, char *dst) to create a new link to an existing file.*

iv. (3 points) Use the Remove system call and answer to question part iii above to show the implementation of the Rename system call.

```
int Rename(char *old, char *new) {
        Remove(new);
        Link(old, new);
        Remove(old);
}
```

b. (9 points) This set of questions is about adding **soft** links to the file system. Soft links are also called symbolic links.

i. (3 points) What changes would you make to the internals (directory, fileheader, free map, etc.) of the filesystem to support soft links?

Add a new type of file to the directory or the file header (2 points), and store the soft link's pathname in a file (1 points).

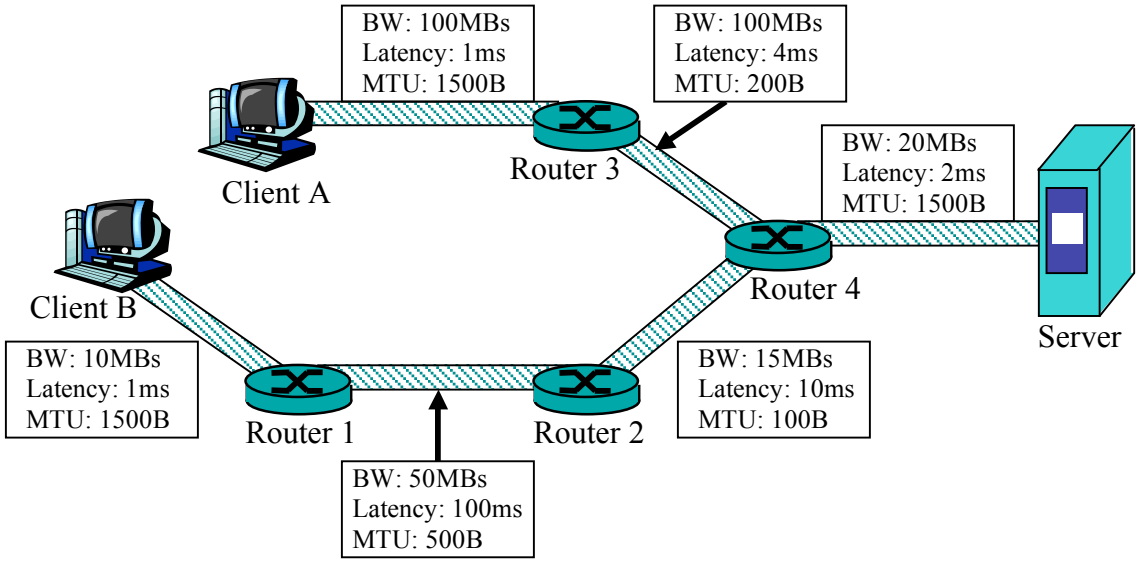
ii. (3 points) What existing system calls have to be modified to support soft links? How do the system calls need to be modified?

Open needs to be changed to implement the recursive lookup implied by soft links (1 point). If the file being opened resolves to a soft link, Open must open the file pointed to by the soft link (2 points). The dereferencing operation should count the number of dereference operations to prevent an error from loops (1 point). Remove must be modified in a minor way to remove soft links correctly. If you removed the target of a link (instead of the link), we deducted 2 points. If you said that system calls use file descriptors instead of names, we deducted 2 points.

iii. (3 points) What new system calls, if any, must be added to the system? Give the C language-style signature of any new calls, e.g., int Open(char *file).
List the signature and give a one sentence definition of each arg and return value.

*Add a int SymLink(char *src, char *dest) to create a new symbolic link to an existing file. No partial credit.*

5. (20 points) Networking



The above figure illustrates a network in which two clients (Client A and Client B) route packets through the network to the server. Each link is characterized by its Bandwidth (BW), one-way Latency, and Maximum Transfer Unit (MTU). All links are full-duplex (can handle traffic in both directions at full bandwidth).

a. (4 points) Under ideal circumstances, what is the maximum bandwidth that Client A can send data to the server without causing packets to be dropped (Assuming that the headers are of zero length)? How about Client B? Explain.

Here we are looking for the bottlenecks in the bandwidth and all we need to do is find the minimum bandwidth along the paths from the clients to the server. For A the minimum along its path to the server is 20MBps. And for B it is 10MBps

- We gave 2 points per correct answer (and took off 1 point if you swapped A & B) A lot of people tried to add in a lot of fancy stuff here, but there wasn't anything more complicated than finding bottlenecks.

b. (4 points) Keeping in mind that TCP/IP involves a total header size of 40 bytes (for TCP + IP), what is the maximum data bandwidth that Client A could send to the server through TCP/IP? Explain.

Here the interesting part was noticing that we had to take the minimum MTU to tell us what the largest packet we can create is. We see that A→S has a min MTU of 100 bytes of which 40bytes are used for header. This leaves us with 60 useable bytes of payload. Thus only (200-40)/200 = 80% of the bandwidth will be useful data bandwidth. Leaving us with 80% × 20MBps = 16MBps

- We gave 1 point for realizing that the min MTU was 200 bytes. 1 point for correctly seeing that only 80% of the bandwidth was usable and 2 points for multiplying the 80% with the correct bandwidth to give the maximum data bandwidth.

- Note: if you assume that the sender is not using an algorithm to avoid fragmentation, then they may try to send packets of size 1500. These will get fragmented into 15 IP packets when they cross the "MTU bottleneck". So, we will have 8×20 (IP header size) + 20 (TCP header size) overhead = 180/1500 ⇒ (1500-180) /1500 data = 22/25 data = 0.88%.

c. (4 points) Assume that Client A sends a continuous stream of packets to the server (and no other clients are talking to the server). How big should the send window be so that the TCP/IP algorithm will achieve maximum bandwidth without dropping packets? Explain. Hint: don't forget to account for the 40 bytes of header.

Remember that the optimal window size = roundtrip latency × effective bandwidth
The roundtrip latency along A's path to the server was (1+4+2)×2 = 14ms. The effective bandwidth from part b was 16MBps so the total was 16MBps × 0.014s = 0.224 MB
- We gave 2 points for telling us how to correctly calculate the correct window size, 1 point for giving the right latency term, and 1 point for the calculation itself.

d. (4 points) Assume that Clients A and B both send a continuous stream of packets to the server simultaneously. Assume that Bandwidth is shared equally on shared links. What must the sizes of their send windows be so that packets are not dropped? Explain.

Now the link between router 4 and the server gets shared equally between A and B and thus the bandwidth for each of them drops to 10MBps.
From part b we know that the MTU is still 200 bytes and therefore we can at most get 80% of the bandwidth so A's window size is 10 Mbytes/s × 0.014 s × 0.8 = 112 kB
- For B we need to calculate its effective bandwidth. We see that the min MTU is 100 bytes and therefore (100-40)/100 tells us that we can achieve 60% of the bandwidth. We also notice that B's roundtrip latency to the server is only 2×(1+100+10+2)=226ms. So running the same calculation as A we get B's window size is 10 Mbytes/s × 0.226s × 0.6 = 1.356 MB
- There were two points to each part and were distributed according to same ratios as part (c)

e. (4 points) Consider the situation of (5d). If the server is sending data back to Clients A and B at full rate, do the acknowledgements headed to the clients decrease the bandwidth in the forward direction (clients→server)? State your assumptions and explain your answer.

There were two answers here we accepted. The correct answer is that it doesn't affect the bandwidth because the acks are piggy backed on the reverse data packets and therefore the there is no loss of bandwidth for the acks.