Examination 2 Operating Systems, Computer Science 500
Spring 2001 (May 8, 2001)

1 Rules and Hints for the Exam

This examination is open book and open note (your notes only please). Calculators are permitted. Networked devices are strictly prohibited. The questions are marked as to their relative value, the exam will be scored out of 100% but is worth 25 points towards your course grade. Relax and try to do what you can.

Here are some hints for doing well:

- You have about 100 minutes, so using one minute per percentage point of work is a good rate.
- You can do the problems out of order, just be sure that the grader can find the order.
- Please be neat (illegible answers cannot get credit).
- Please put in the upper right hand corner of each page your name, the page number and the total number of pages your grader should expect (in case the pages become detached). So for example if your name was James Taylor, then you would write: James Taylor 1 of 5.
- Don’t forget to staple your pages together at the end.

2 The Problem Set

1. Deadlock (10%) — Consider a system with 2 processes, $P_1$ and $P_2$, and 2 units of a particular resource, $R_1$ and $R_2$. Requests and grants are for a single unit of resource, although each process may require 2 units of resource during its execution. Draw a state diagram where each state describes the amount of resource granted and the amount of resource requested by each process. Mark deadlocked states by drawing them using 2 circles (much like accepting states in finite automata). (Hint: draw the system for a single process first and then try it with 2).

As per the hint, we first diagram a single process system. For clarity, the notation $(x, y)$ is introduced, where $x$ represents the amount of resource granted to the process, and $y$ represents the number of outstanding requests for resource. Since resource is requested and granted in single units, we can see that $0 \leq y \leq 1$ and that $0 \leq x \leq 2$ and $x + y \leq 2$. Furthermore, if we make the (reasonable) assumption of holding and waiting for resources, when the process is in a state where $y > 0$, it will not release a resource. The diagram in Figure 1 shows the single process solution.

Now consider the case where there are 2 processes, we can treat the two processes as symmetric and draw them in orthogonal dimensions. We introduce the notation $(x_1, y_1), (x_2, y_2)$, where $x_i$ shows the number of resources granted to process to $i \in \{1, 2\}$ and $y_i$ shows the
request requirements of process $i \in \{1, 2\}$. Again since each process only requests one unit of resource at a time $0 \leq y_1, y_2 \leq 1$ and since the amount of resource granted cannot exceed the amount available, $0 \leq x_1 + x_2 \leq 2$. If we make the reasonable assumption of processes holding resources and waiting then process $P_1$ can only release resources when $y_1$. We assume single units of resource are released at a time (although we could show double releases, they can be obtained via transitive closure). The state diagram that results can be clearly shown in Figure 2. Notice that the deadlocked state corresponds to the state where both processes

Figure 2: Solution State Diagram (2 processes) for Problem 1

have one unit of resource and want one additional unit.

2. Networking Systems Programming (15 %) —

   (a) Network Systems Programming (5 %) — In BSD sockets, what does the listen system call do and what is the significance of the second parameter.

   The listen system call is invoked by the process acting as the server in a client server program. The listen system call is applied to a socket that has been created and bound to a particular port, causing the socket to be readied for incoming connections. The second
parameter is the number of pending connections that can queue pending an accept system call.

(b) Data Communication Technology and Performance (10 %) — Traditional kernel technology for input and output (I/O) and data communication relied on buffering of data in the kernel prior to transferring the data into the address space of the user process. In contrast, zero copy systems calls and libraries are frequently used for I/O devices and in data communications devices, and transfer data directly from the device into the user's data space. Compare and contrast the two approaches, under which circumstances is one better or worse than the other.

Buffered I/O (using kernel buffers that is) allows for more straight forward systems programming, since the kernel copies the data between the user space and the kernel space for data transfers. This allows the kernel to coalesce data into larger blocks which assists in doing block oriented I/O. However, zero copy I/O allows users to get higher throughput since the kernel does not copy data into its internal structures. The major problem with zero copy I/O is that the programmer must be careful not to corrupt the data in their buffers during the transfer.

3. Threads and Synchronization (30 %) —
(a) Recall that Java uses the key word synchronized to guard a region of code (possibly a whole method or a smaller subregion). Lazy creation of objects is a way of deferring allocation and initialization until an object is actually needed (and can help avoid wasting systems resources). In a single threaded program, the logic might look something like this:

```java
class SomeClass {
    private Resource resource = null;

    public Resource getResource() {
        if (resource == null) {
            // Already allocated?
            resource = new Resource(); // does expensive allocation here
        }
        return resource;  // Now we have the resource
    }
}
```

However, suppose that you have a multi-threaded program AND you can turn off optimizations that reorder the generated instructions. If you looked in a text book and saw code that looked like:

```java
class SomeClass {
    private Resource resource = null; // reference to shared resource

    public Resource getResource() {
        if (resource == null) { // already allocated?
            synchronized {
                resource = new Resource();
            }
        }
        return resource;
    }
}
```

would this work? Why or why not? If not, can you provide a fix that does not involve making the entire `getResource` method synchronous (15 %)?

Actually this trick would not work. The obvious reason is that the return statement should not be in the body of the if, but that was not what I was trying to get at. Suppose the code had instead looked like:

3
class SomeClass {
    private Resource resource = null; // reference to shared resource

    public Resource getResource() {
        if (resource == null) { // already allocated?
            synchronized {
                if (resource == null) { // check it again
                    resource = new Resource(); // NOW allocate and initialize
                }
            }
        }
        return resource;
    }
}

The code would still be broken. Consider the case where there are two threads, both
of which invoke the getResource method at nearly the same time. Both threads would
evaluate the predicate in the if and find that resource == null returns true. The
synchronized portion would thus be invoked 2 times (albeit serially). The fix is to use
what is called double checked locking, which looks like:

class SomeClass {
    private Resource resource = null; // reference to shared resource

    public Resource getResource() {
        if (resource == null) { // already allocated?
            synchronized {
                if (resource == null) { // check it again
                    resource = new Resource(); // NOW allocate and initialize
                }
            }
        }
        return resource;
    }
}

which should work under our assumptions that the statements would not be reordered. It
is interesting to note that the Java memory model was recently discovered to be broken,
(at the time of writing that is). So that reordering of statements and memory consistency
is in fact NOT guaranteed (you cannot turn off reordering by the optimizer!). For more
details see:

(b) Using C like pseudo-code, and an atomic hardware test-and-set operator design a semaphore based
synchronization routines for both sem_wait and sem_signal (15 %).

Typically a semaphore has a queue of suspended processes and a counter, so we could
describe the semaphore operations of sem_wait and sem_signal as follows.

```c
typedef struct{
    int counter;
    ProcessQueuetype q;
} SemaphoreType;
```

shared int lock = 0; /* the structure is not locked */

```c
void acquire_lock(){
    while (!test_and_set(lock)){
        /* do nothing */
    }
}
```
void release_lock(){
    lock = 0;
}

void sem_wait(SemaphoreType *s){
    int pid;

    get_lock();
    s->count--;
    if (s->count < 0){
        pid = getpid();
        enqueue(&(s->q), pid);
        release_lock();
        suspend(pid);
    }
    release_lock();
}

void sem_signal(SemaphoreType *s){
    int pid;

    get_lock();
    s->count++;
    if (s->count <= 0){
        pid = dequeue(&(s->q));
        wakeup(pid); /* put it on the ready queue */
    }
    release_lock();
}

4. Networking Systems Performance (30 %) — Consider a $1000 \times 10^6 \frac{\text{bits}}{\text{sec}}$ Ethernet (Gigabit). Suppose that data travels at about $2 \times 10^5$ kilometers per second over a copper wire.

(a) Suppose that you have a 100 meter cable, how many bits can be in transit on that cable simultaneously? (10 %).

From our givens we know:

\[
\begin{align*}
\text{Bandwidth} & = 1000 \times 10^6 \frac{\text{bits}}{\text{sec}} \\
\text{Velocity} & = 2 \times 10^5 \frac{\text{km}}{\text{sec}} \\
\text{Distance} & = 100 \text{m}
\end{align*}
\]

If we can estimate the time of flight, we can tell how many bits must be in transit to sustain the rate of transfer. We do this by computing:

\[
\text{time of flight} = \frac{\text{Distance}}{\text{Velocity}} = \frac{100\text{m}}{2 \times 10^5 \frac{\text{km}}{\text{sec}} \times 10^3 \frac{\text{m}}{\text{km}}} = 5 \times 10^{-7} \text{sec}
\]

\[
\text{bits in transit} = \text{Bandwidth} \times \text{time of flight} = \left(10^9 \frac{\text{bits}}{\text{sec}}\right) \times \left(10^{-7} \text{sec}\right) = 500 \text{bits}
\]

(b) Suppose that the network requires a 100 byte gap between packets to support a distributed file system.
i. If the file system used traditional IP packets with 18 byte headers and up to 1500 bytes of payload data, which contain UDP packets with 8 byte headers. What is the maximum data transfer rate over the wire (assuming that the probability of packet loss, corruption and reordering is negligible). (10 %)
We can solve this as follows:

\[
\text{Effective Bandwidth} = \text{Max Bandwidth} \times \frac{\text{User Data}}{\text{Overhead} + \text{User Data}}
\]

\[
\text{Overhead} = \text{IP Header} + \text{UDP Header} + \text{Inter Packet Gap} = 18B + 8B + 100B = 126B
\]

\[
\text{User Data} = \text{IP Payload} - \text{UDP Header} = 1492B
\]

\[
\text{Effective Bandwidth} = 10^9 \times \frac{\text{bits}}{\text{sec}} \times \frac{1492B}{1492 + 126B} = 9.176 \times 10^8 \frac{\text{sec}}{\text{sec}}
\]

ii. Suppose that IP’s Jumbo Frames option is used, allowing for 9000 bytes of payload in an IP packet. What is the maximum rate of data transfer over the cable. (10 %)

The major difference between this case and the previous case is the size of the IP packet’s payload.

\[
\text{Effective Bandwidth} = \text{Max Bandwidth} \times \frac{\text{User Data}}{\text{Overhead} + \text{User Data}}
\]

\[
\text{Overhead} = \text{IP Header} + \text{UDP Header} + \text{Inter Packet Gap} = 18B + 8B + 100B = 126B
\]

\[
\text{User Data} = \text{IP Payload} - \text{UDP Header} = 8992B
\]

\[
\text{Effective Bandwidth} = 10^9 \times \frac{\text{bits}}{\text{sec}} \times \frac{8992B}{8992 + 126B} = 9.862 \times 10^8 \frac{\text{sec}}{\text{sec}}
\]

5. Naming and Persistence and Systems programming (15 %): A Bloom Filter is a heuristic algorithm for detecting membership in a set that may (with low probability) give false positives (that is a membership test falsely indicates that a particular element is a member of the set when in fact it is not actually a member of the set). Bloom filters are frequently used in information retrieval and in distributed file systems (to identify which machine in a distributed system a file resides on). A Bloom Filter models a set using an array of \( N \) bit-mapped sets with \( K \) bits (we will call this the filter). The insertion and membership test operators employ a unique hash function for each bitmap. To insert an element into the set represented by the filter, for each bitmap, the bitmap’s unique hash function is applied to the element and the bit selected is turned on. To test for membership, for each bitmap, the same unique hash function is evaluated and the corresponding bit is checked in the bitmap, with a report of the item being in the filter if the corresponding bit is on in each bitmap. Write efficient C like pseudo-code to implement a Bloom filter:

```c
#define K .... /* Number of bits in a bitmap */
#define N .... /* Number of bitmaps in the Bloom Filter */
/* assume K is a multiple of the number of bits in a long integer */
typedef unsigned long BitMap[K/(8 * sizeof(unsigned long))];
typedef BitMap Filter[N]; /* the Filter itself */

/* inserts the Filename into the set of files represented by the Filter */
void BF_insert(const char *FileName, Filter *bf_ptr);

/* tests to see if the Filename is represented by the set of files in the Filter */
bool BF_ismember(const char *FileName, const Filter *bf_ptr);
```

6
You can assume that the following functions are already written:

```c
/* Computes the hash function selected by WhichFunction on a given
   FileName returning a value in that is between 0 and NumValues - 1 */
int BF_hash(const char *FileName, int WhichFunction, int NumValues);
```

Give the computational complexity of your BF_insert and your BF_ismember operations.

Following our definition and our givens we get:

```c
const int bits_per_long = sizeof(unsigned long) * 8;

void BF_insert(const char *FileName, Filter *bf_ptr){
    int i, bit_to_set, idx, bit_pos;
    for (i = 0; i < N; ++i){
        bit_to_set = BF_hash(FileName, i, K);
        idx = bit_to_set / bits_per_long;
        bit_pos = bit_to_set % bits_per_long;
        bf_ptr[i][idx] |= (0x1L << bit_pos);
    }
}

bool BF_ismember(const char *FileName, const Filter *bf_ptr){
    int i, bit_to_check, idx, bit_pos;
    bool result = true;
    i = 0;
    while( result && (i < N) ){
        bit_to_check = BF_hash(FileName, i, K);
        idx = bit_to_check / bits_per_long;
        bit_pos = bit_to_check % bits_per_long;
        result = result || ((bf_ptr[i][idx] & (0x1 << bit_pos)) != 0);
        ++i;
    }
}
```

Notice that each function has a similar loop structure, if we denote the cost of the hash function on the file name as \(O(H(FileName))\), then the cost of these functions is both \(O(NH(FileName))\).