Examination 1 Operating Systems, Computer Science 66421
Spring 1998 (March 2, 1998)

1 Rules of 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.

2 The Problem Set

1. Virtual Memory Management (total 10 %): Consider the reference string:

   \[ \omega = 1, 2, 3, 1, 1, 2, 4, 5, 3, 6 \]

   Please be sure that to count all pages loaded into memory as page faults (not just replacements). Circle page faults in your chart (much like in the lecture notes).

   (a) (5 %) Compute the set of resident pages for \( \omega \) given the (least recently used) LRU algorithm with 3 page frames. Give the total number of page faults.

   Applying LRU we get:

<table>
<thead>
<tr>
<th>Time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_t )</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Page 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Page 2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page 3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faults</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

   (b) (5 %) Compute the set of resident pages for \( \omega \) given the Least Frequently Used (LFU) algorithm with 3 page frames. LFU computes the frequency of references by counting the number of accesses since loading the page and dividing by the number of references made since it was last loaded.

   Applying LFU we get:

<table>
<thead>
<tr>
<th>Time</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_t )</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>Page 1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Page 2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>Page 3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Faults</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
2. Memory Management and Segmentation (total 20 %):

(a) What services does segmentation provide (5 %)? Provide a diagram depicting the architectural support needed for segmentation.

Segmentation provides support for relocation. It partitions the memory into dynamically sized regions (called segments). Segmentation can be used to support memory protection, sharing and virtual memory given combined hardware and software support (e.g. the Intel 8086 had segmentation but lacked hardware support for these features). Consider the diagram in Figure 1. Typically the segments are organized in a segment table, with the base address of the segment currently accessed being stored in a base register. A fence register records the maximum address for the segment. The unallocated memory regions are called holes, and are frequently stored in a list structure (not shown in the figure).

(b) Worst Fit Segmentation Placement (15 %):

i. System Software Design (10 %): Outline a computationally efficient placement algorithm for the worst fit placement algorithm. The call:

\[ \text{PhysicalAddressType PlaceSegment(SizeType SegmentSize, LogicalAddressType BaseAddress ProcessIdType Owner)} \]

attempts to allocate SegmentSize bytes of physical memory for use by a segment beginning at logical address BaseAddress owned by the process indicated by Owner. Upon success PlaceSegment returns the base address of the physical memory location allocated for the segment, otherwise it returns a reserved value: NullPtr indicating that it was unable to allocate physical memory for the segment. The parameter Owner indicates which process owns the segment if it is successfully placed.

Be sure to use the architectural features you described in your answer to Problem 2a. You can assume that basic data structures (such as queues, stacks, priority queues, array sorting routines, hash tables and binary trees) are implemented and that you do not to provide that level of detail.

To do this problem you first have to know what a worst fit placement strategy is. Worst fit placement satisfies a request for space by allocating the space from the largest free region. Applying case analysis gives the following solution:

A. Check for presence of the segment in the table, if there return its physical address if the segment size matches, otherwise NullPtr.
B. If \( k = 0 \) there are no segments, return NullPtr.
C. Get the largest hole, and call its size Hole.size.

---

1Placement strategies like worst fit, first fit, and best fit are not just limited to memory management applications, other uses include manufacturing applications like fabric cutting for example.
D. If \( \text{hole} \_\text{size} < \text{segmentsize} \) the hole is too small and by definition no other hole is large enough to contain the segment. Return NULLPtr.

E. If \( \text{hole} \_\text{size} = \text{segmentsize} \) the hole is an exact fit for the segment, return the hole’s base address and delete it from the hole list.

F. If \( \text{hole} \_\text{size} > \text{segmentsize} \) the hole is a “loose fit” for the segment. Allocate part of the hole to hold the segment and delete the old version of the hole and insert the unused portion of the hole back on the free space list.

Now the most common actions on the segment table are likely to be lookup, insertion and deletion. If we hash the segment table using the owner and logical base address as the key, we can look up a segment descriptor in the table in \( O(1) \) time using a hash table structure. The most common actions on the free space list are find the largest hole, insert a hole and delete a hole. We can allocate a priority queue for the holes such that the holes of largest size have the highest priority, using a binary heap is your most efficient bet for a priority queue, since accessing the head of the queue is an \( O(1) \) operation (find largest), and insertion and deletion of the head are \( O(\log_2 N) \) where \( N \) is the number of elements in the queue. So pseudocode might look like:

```c
PhysicalAddressType PlaceSegment(SizeType SegmentSize,
    LogicalAddressType BaseAddress ProcessIdType Owner)
{
    SegmentType segment;
    HoleType hole;

    int k = pqnelems(HoleList); /* the number of holes */
    
    segment.Owner = owner;
    segment.LogicalAddress = LogicalAddress;
    if (InSegmentTable(SegmentTable, segment)) {
        segment = SegmentTableFind(SegmentTable, segment);
        if (segment.size != SegmentSize) { /* Segment Size Mismatch */
            return NULLPtr;
        }
        return segment.PhysicalAddress;
    }
    if (k == 0) { /* No Holes remaining, all space is allocated */
        return NULLPtr;
    }
    hole = pqhead(HoleList); /* get the largest hole */
    if (hole.size < SegmentSize) { /* the hole is too small */
        return NULLPtr;
    }
    else if (hole.size == SegmentSize) { /* exact fit */
        hole = pqdelete(HoleList); /* remove the hole from the list */
        segment.PhysicalAddress = hole.PhysicalAddress;
        segment.LogicalAddress = LogicalAddress;
        segment.size = SegmentSize;
        segment.owner = Owner;
        SegmentTableInsert(SegmentTable, Segment);
    }
    else { /* lose fit, i.e. hole.size > SegmentSize */
        hole = pqdelete(HoleList); /* remove the hole from the list */
        segment.PhysicalAddress = hole.PhysicalAddress;
        segment.LogicalAddress = LogicalAddress;
        segment.size = SegmentSize;
        segment.owner = Owner;
        SegmentTableInsert(SegmentTable, Segment);
    }

    // Place the segment in the SegmentTable
    // Place the hole in the HoleList
    // Update the hole's size
    // Return the physical address
}
```
hole.PhysicalAddress += SegmentSize;
hole.size -= SegmentSize;
pqInsert(HoleList, Hole);
}

ii. Overhead Analysis (5 %) What is the computational complexity (using big O notation) of your version of PlaceSegment designed in question 2(b)ii if there are k holes and n segments in the system when it is called?

Since each case above has a constant number of operations and the most expensive operation (from a computational complexity standpoint) are the priority queue insertions and deletions with a cost of O(\log_2 k). If you used a (preferably balanced) binary search tree structure (i.e. AVL Trees, Red Black trees or Splay trees) the complexity is still the same. Furthermore we know that k ≤ n + 1 since at most one hole can separate a pair of segments and 2 more holes could be have a segment on one side and the max/min physical address at the other extreme.

3. Memory Performance (25 %): Consider a web browser which locally stores recently referenced web pages in local storage. The local storage is hierarchical with a fast memory cache, and a slower disk cache. On average, the server requires 1.5 seconds to retrieve a web page from a remote site and load it into main memory. Loading a web page from the disk into memory takes (on average) 85 milliseconds. Once a web page is resident in memory, it will take 0.95 seconds to display (on average).

(a) (10 %) Assume that 99.9% of references are locally cached pages. What is the minimum hit rate for a locally cached page to be resident in the main memory, \( p_r \), to ensure that pages display no slower than 1 second on average?

When choosing the values for this function, they were a bit skewed and the web browser turned out to be incredibly efficient. Let \( t_m = 0.95 \text{sec} \) denote the time it takes to display a memory resident page. Let \( t_d = 0.085 \text{sec} \) denote the time it takes to load a page from the disk into memory. Let \( t_N = 1.5 \text{sec} \) denote the time it takes to load a page from the network into memory. Let \( p_N = 0.001 = 1 - 0.999 \) denote the probability of a page not being locally cached (i.e. only on the network). Let

\[
    p_{r1} = \frac{p_r}{1 - p_N} \tag{1}
\]

be the conditional probability that a page is memory resident given that it is locally cached. Our constraint of one second lookup time leads to the following inequality, which we solve for a lower bound on \( p_{r1} \).

\[
    1 \geq t_m + (P_N t_n) + [(1 - P_N)(1 - p_{r1})t_d] \tag{2}
\]
\[
    1 - t_m - (P_N t_n) \geq [(1 - P_N)(1 - p_{r1})t_d] \tag{3}
\]
\[
    \frac{1 - t_m - (P_N t_n)}{(1 - P_N)t_d} \geq 1 - p_r \tag{4}
\]
\[
    \frac{1 - t_m - (P_N t_n) - 1}{(1 - P_N)t_d} \geq -p_r \tag{5}
\]
\[
    p_{r1} \geq 1 - \frac{1 - t_m - (P_N t_n)}{(1 - P_N)t_d} \approx 0.43 \tag{6}
\]

I accepted \( p_{r1} \) as a final result since the wording could be interpreted as a request for the conditional probability. The unconditional probability of a page being resident in memory can be solved for as: \( p_r = (1 - P_r)p_{r1} \approx 0.43 \) which is coincidentally very close in value (meaning that if \( P_N \) were not so close to 0, these values could be quite different).
<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>CPU Service Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_1$</td>
<td>0.1 sec.</td>
<td>6 sec.</td>
</tr>
<tr>
<td>$P_2$</td>
<td>0.3 sec.</td>
<td>8 sec.</td>
</tr>
<tr>
<td>$P_3$</td>
<td>0.8 sec.</td>
<td>3 sec.</td>
</tr>
<tr>
<td>$P_4$</td>
<td>1.5 sec.</td>
<td>9 sec.</td>
</tr>
</tbody>
</table>

Table 1: Job Arrivals for Problem 4a

<table>
<thead>
<tr>
<th>Part</th>
<th>Schedule</th>
<th>$t_q(P_1)$</th>
<th>$t_q(P_2)$</th>
<th>$t_q(P_3)$</th>
<th>$t_q(P_4)$</th>
<th>$t_q$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4(a)i</td>
<td>FCFS</td>
<td>6</td>
<td>13.7</td>
<td>16.2</td>
<td>24.5</td>
<td>15.1</td>
</tr>
<tr>
<td>4(a)ii</td>
<td>SJF</td>
<td>6</td>
<td>8.2</td>
<td>16.7</td>
<td>24.5</td>
<td>13.85</td>
</tr>
</tbody>
</table>

(b) (15 %) Assume that you want to obtain the main memory hit rate, $p_r$, computed in question 3a. Suppose you have $m$ kilobytes of main memory. A reference to a locally cached web page will be in the main memory with probability $p(m) = e^{-\frac{m}{m_{\text{hit}}}}$. Assume that main memory can be bought in kilobyte increments, with a cost of $C_m = \frac{30,000}{1000}$, How much will it cost to buy enough memory to ensure that the average time to display a web page will be no more than 1 second.

Using our results from last time and our givens we get:

$$p_r = p_m = e^{-\frac{m}{m_{\text{hit}}}}$$

(7)

$$\ln p_r = -\frac{1}{m}$$

(8)

$$m = -\frac{1}{\ln p_r} \approx 1.18 \text{KB}$$

(9)

We need to buy memory in units of KB so $m = 2KB = [1.18KB]$ at a cost of $mC_m = 80.0012$.

4. Scheduling (20 %):

(a) Computing Schedules and their performance (10 %) Suppose that you are given the following set of processes in Table 1 predict their required CPU service. You may assume that the CPU service required is known exactly by the operating system. Draw the scheduling diagram and compute the mean response time (turnaround time) for the following scheduling algorithms:

i. FCFS (First Come First Serve) (5 %)

ii. SJF (Shortest Job First) (5 %)

The schedules for the jobs are diagrammed in Figure 2, the solutions given in Table 2 (recall that the time of arrival is used in computing the response time for a job) .

(b) Scheduling Algorithm Analysis (total 10 %): Countermeasures are techniques employed by user programs to obtain resources in a fashion which contradicts the intent of the operating systems designer. Uniprocessor scheduling is susceptible to countermeasures employed by a user process to artificially increase that process’ priority for CPU access. Describe a counter measure for Shortest Job First (SJF) where the user processes provide accurate information for estimating their CPU requirement.

One solution would be to partition a long job into a sequence of shorter jobs where each job loads in any partially computed results at startup and preserves its partially computed results before terminating. Note that “lying to the scheduler” was not permitted by the problem statement, so a process could not merely claim a shorter expected run time.

5
Figure 2: Schedules for problems 4(a) and 4(a)ii

Figure 3: A Router described in Problem 5
5. Queuing Theory (15 %): Performance analysis (15 %): Consider the system shown in Figure 3. When a message is received, from one network (either side) the router processes the message and forwards the processed version on the other network. Suppose that the router receives a message (on average) every 5.9 milliseconds. (exponentially distributed) (exponentially distributed), and requires 5.7 milliseconds, to process the message and transfer it to the appropriate network. Assume that messages are processed in FIFO order, and that the router has storage space for 40 messages.

(a) (5%) What is the utilization of the router?

Since both arrival and service times are exponentially distributed we can use M/M/1 analysis techniques. Solving for the utilization, \( \rho \) using the mean service time \( s = 5.7\)msec, and the mean interarrival time of \( \tau = 5.9\)msec. gives:

\[
\rho = \frac{s}{\tau} = \frac{5.7\text{msec.}}{5.9\text{msec.}} \approx 0.966
\]  

(10)

(b) (5%) What is mean response time of the router?

Again using M/M/1 analysis we solve for the mean response time \( t_q \) as follows:

\[
t_q = \frac{s}{1 - \rho} \approx 167.64\text{msec.}
\]  

(11)

(c) (5%) What is mean message queue length of the router? Is buffer overflow likely?

Again using M/M/1 analysis we solve for the mean queue length, \( w \) as follows:

\[
w = \frac{\rho^2}{1 - \rho} \approx 27.4
\]  

(12)

Note that I also accepted the solution for computing the total number of jobs in the system \( q \) (if we assume that we want to know the number of buffers in use), which is calculated as follows:

\[
q = \frac{t_q}{\tau} = w + \rho \approx 28.4
\]  

(13)

6. I/O Management (10 %): DMA vs. Interrupt I/O — Briefly compare the relative efficiency of DMA vs. Interrupt driven I/O (qualitatively) and describe how they impact an operating systems performance.

Interrupt driven I/O uses the CPU to manage the data transfer between memory and the peripheral devices. This requires the following features/overhead:

(a) Preserving and restoring process state before and after service.
(b) Sending all data through the CPU to go to memory.
(c) Putting the instructions and data used to manage the (such as loop counters, etc...) on the bus adds to the volume of information which must be relayed over the bus.
(d) The CPU can only be interrupted before instruction execution during the fetch/execute cycle.

DMA uses a dedicated processor to do a high speed data transfer on to the bus. Typically this transfer has higher priority bus access than the CPU due to the real time requirements of I/O devices (to avoid loss of data). This means that the CPU has a hardware handshake with the DMA controller. DMA has the following features/overhead:

(a) DMA data transfers can occur at the start of any memory cycle, not just at the single point during the fetch execute cycle.
(b) DMA stalls the CPU (the CPU waits for memory access via the bus) until the DMA completes.
(c) DMA does not require preserving process state information to perform the data transfer.
(d) DMA might interrupt the CPU to indicate a data transfer.

DMA is a higher speed block transfer mechanism than interrupt driven I/O and is becoming more prevalent. Device driver technology is sensitive to whether the device uses interrupt driven data transfer or DMA.