Information about Labs 5 and 6
In lab 5 you use the Java monitor construct to begin writing a simulation of a computer system, and in lab 6 you finish coding the simulation. This large programming assignment is broken into two labs to make sure that no one ends up trying to write the entire program all at once at the last minute. Lab 5 is marked pass/fail or pass/marginal/fail, and the TA provides coding corrections and suggestions if needed. The TA marks Lab 6 (your complete simulation) in detail. If you have trouble with the coding of lab 5, please refer to “How do I get started on Lab 5?” on page 89 of the course reader: here you find detailed advice on how to break the design, coding, and debugging of lab 5 into small stages.

Readings for Assignment 4
In the course reader:
   Pages 49-59   Overview of Memory Management.

In the textbook:
   Chapter 7   Deadlock -- Banker’s algorithm in Section 7.5.3 (This reading was already given in assignment 3)
   Chapters 8-9   Memory management and virtual memory

Terminology for Memory Management
kilobyte  \(2^{10}\) bytes = 1024 bytes. Usually the prefix kilo means exactly 1000, as in kilometer or kilowatt. In a computer context, kilo means 1024 because computer memories come in power-of-two sizes.
megabyte  \(2^{20}\) bytes = 1024 kilobytes = approximately 1 million bytes (1,048,576 bytes).
gigabyte  \(2^{30}\) bytes = 1024 megabytes = approximately 1 billion bytes (1,073,741,824 bytes).
millisecond  \(10^{-3}\) seconds = 1/1000 of a second. The prefix milli means exactly 1/1000, as in millimeter or milliliter.
microsecond  \(10^{-6}\) seconds = 1/1000 of a millisecond.
nanosecond  \(10^{-9}\) seconds = 1/1000 of a microsecond.

The terms virtual address and logical address are synonymous. At runtime, address translation hardware converts a virtual address to a physical address. If paging is being used, address translation starts with a Page Table Base Register that contains the address of the page table for this process; the first bits of the virtual address are used as the array index in the page table.

Question on the Banker's Algorithm
1) This problem illustrates what happens when the operating system uses the Banker’s algorithm to prevent deadlock, and processes overestimate their need for resources. To keep the example simple, there is only one type of resource (tape drive). As you will see, the operating system sometimes makes processes wait for tape drives even though many unused tape drives are available. This underutilization of resources is justified in cases where it is very important to prevent deadlock.

Consider a system that has 10 tape drives. The system is executing a set of 20 processes, each of which overestimates its need for tape drives:
(a) Each of the 20 processes declares Max = 2 tape drives, but each process ends up using only one tape drive during execution. What is the maximum number of tape drives that can be in use at any one time, given this job set (and given that the Banker’s algorithm is being used)? Justify your answer. Hint: “5” is not the right answer.
(b) Each of the 20 processes declares Max = 10 tape drives, but each process ends up using only one tape drive. What is the maximum number of tape drives that can be in use at any one time, given this job set? Justify your answer.
(c) Generalize the answers from (a) and (b): Each of the 20 processes declares Max = N tape drives, where 1 ≤ N ≤ 10; all 20 processes use the same value of N. Each process ends up using only one tape drive during execution. What is the maximum number of tape drives that can be in use at any one time, given this job set? Give an expression involving N.
2) (a) Briefly describe the difference between absolute code and relocatable code (Section 8.1.2 of the text).
(b) Briefly describe dynamic loading and dynamically linked libraries (Sections 8.1.4 and 8.1.5).

3) This question is about the memory pyramid I discussed in lecture: a small amount of fast, expensive memory at the top of the pyramid, and large amounts of slow, cheap memory at the bottom of the pyramid. Data migrates up and down the pyramid, moving toward the top when it is accessed and shifting back down when it hasn’t been accessed for a while. Briefly describe how each of the following three types of data movement occur; indicate whether the compiler, the operating system and/or the hardware is involved.

- data moves between main memory and registers
- data moves between disk and main memory
- data from main memory is stored in a cache

Z) This question reviews an example I discussed in lecture. No need to hand anything in because I give the solution.

The task: A data structure of 10,000 items occupies 10 pages. Write an algorithm to search for 30 items in this data structure. Compare the runtime of (1) sequential search looking for all 30 items at once, (2) binary search (has to be done as 30 separate searches), and (3) hash table access (has to be done as 30 separate searches).

(a) assuming all 10 pages fit in main memory (b) assuming only 2 of the 10 pages fit in main memory.

Note that each page holds 1000 items, or 10% of the items. If an array is used, page 0 holds items 0 to 999, page 1 holds items 1000 to 1999, ..., page 9 holds items 9000 to 9999. If the algorithm accesses an item that is not in main memory, then there is a page fault and that 10% of the data structure gets moved into main memory (overwriting whatever 10% of the data structure was previously stored in that part of main memory).

Solution: In (a), sequential search takes about 1000 times as long as binary search and 10,000 times as long as hashing assuming an ideal hash function with no collisions. In (b), sequential search is fastest because it gets fewest page faults.

<table>
<thead>
<tr>
<th></th>
<th>(a) All 10 pages fit in main memory</th>
<th>(b) 2 of 10 pages fit in main memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Sequential search</td>
<td>Compare 30 search items to the first item in the array, to the second item etc. 150,000 comparisons on average 300,000 comparisons worst case</td>
<td>10 page faults. Approximating that 150,000 comparisons take about as much time as 2 page faults, total time is roughly 12 page faults</td>
</tr>
<tr>
<td>(2) Binary search</td>
<td>30*log(10,000) comparisons or about 400 comparisons</td>
<td>90 to 120 page faults (and negligible time for 400 comparisons)</td>
</tr>
<tr>
<td>(3) Hashing (requires preprocessing: make a hash table in which items are indexed according to the value of the hash function)</td>
<td>30 comparisons (in the case of an ideal hash function with no collisions)</td>
<td>80% chance of page fault each time, so calculate .8*30 = 24 page faults</td>
</tr>
</tbody>
</table>

Above I estimate that 80,000 comparisons equals one page fault, because a page fault typically takes 8ms to 16ms and main memory access typically takes 100ns to 200ns. (Disk transfer for a page fault is around 8ms, according to http://en.wikipedia.org/wiki/Page_fault. However, in many cases an old page has to be copied from main memory to disk to make room before the new page can be copied from disk to main memory; in that case the page fault time becomes 16ms.)

A typical binary search accesses 4 different pages of the array. This causes 3 or 4 page faults, as illustrated by the following examples. Each binary search begins by accessing the middle item of the array, that’s item 5000 at the start of page 5. Example 1, search for the first item in the array. Binary search accesses item 5000 (page 5), then item 2500 (page 2), then item 1250 (page 1), then item 625 (page 0), then item 313 (also on page 0) and other smaller items (all on page 0). In total we get four page faults (pages 5, 2, 1, 0); after we are done, pages 1 and 0 are left in main memory. Example 2, search for item 5003 in the array. Binary search accesses item 5000 (page 5), then item 7500 (page 7), then item 6250 (page 6), then item 5625 (page 5), then other items also on page 5. In total we get four page faults (pages 5, 7, 6, 5); after we are done, pages 6 and 5 are left in main memory, so the next binary search will not get a page fault when it begins by accessing page 5. In conclusion, each of the 30 binary searches gets 3 or 4 page faults.

B-Trees are another data structure well-suited for this type of task. From wikipedia: “In computer science, a B-tree is a tree data structure that keeps data sorted and allows searches, sequential access, insertions, and deletions in logarithmic time. The B-tree is a generalization of a binary search tree in that a node can have more than two children (Comer 1979, p. 123). Unlike self-balancing binary search trees, the B-tree is optimized for systems that read and write large blocks of data. It is commonly used in databases and filesystems.”
Questions on Segmentation and Paging

4) Refer to the introductory paging example on pages 50-52 of the course reader. Translate the following logical addresses to physical addresses, using the page table at the top of page 52. To get started: page 52 gives examples of how to translate from logical addresses to physical addresses.

(a) logical address is $2FFF_{16}$
(b) logical address is $30A0_{16}$

5) Consider a computer system in which the virtual memory consists of 8 pages of 1024 bytes each, mapped onto a physical memory of 32 page frames. How many bits are there in a virtual address? How many bits are there in a physical address? Assume that memory is byte-addressed: every byte of memory has its own address. (Alternatively, some computers use word-addressed memory, in which every word of memory has its own address.)

6) Three methods of dividing up logical addresses are shown as rows (i) to (iii) in the following table. For each row, calculate the maximum size of a segment, of a segment table, and of a segments’ page table. You may state your answers in terms of bytes, kilobytes, megabytes, gigabytes, or using a format like “$2^{15}$ bytes”. (Assume the memory is byte-addressed.)

<table>
<thead>
<tr>
<th>Bits in logical address</th>
<th>Page size</th>
<th>Size of entries in segment table and page table</th>
<th>Maximum number of segments</th>
<th>Maximum size of a segment</th>
<th>Maximum size of a segment table</th>
<th>Maximum size of a segments’ page table</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i) 16 bits 128 bytes 8 bytes</td>
<td>64 ($2^6$)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(ii) 32 bits 1024 bytes 8 bytes</td>
<td>4096 ($2^{12}$)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(iii) 32 bits 8192 bytes 8 bytes</td>
<td>65536 ($2^{16}$)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

7) Answer the segmentation-with-paging problem on the last page of this assignment: print that page and fill in the blanks.

Question on Effective memory access time

8) For the computer system described in the next paragraphs, find the maximum acceptable page-fault rate so that effective access time for main memory is ≤ 200 nanoseconds.

Memory management on this computer system uses paging. The page table for the current process is held in main memory. Memory access time is 100 nanoseconds. An address translation cache is used; this is called a translation lookaside buffer (TLB). The TLB has a 95% hit rate. Assume TLB access time is negligible. In other words:

- When address translation encounters a TLB hit, the effective memory access time is 100 nanoseconds. (Address translation time is negligible, so the effective memory access time is the same as memory access time.)
- When address translation encounters a TLB miss (but no page fault), the effective memory access time is 200 nanoseconds. (Address translation requires one access to memory, to read an entry in the page table. Thus, address translation adds 100 nanoseconds of overhead, bring the total effective memory access time up to 200 nanoseconds.)

The time needed for a page faults is 8 milliseconds when a clean page is replaced, and 16 milliseconds when a dirty page is replaced. The replaced page is dirty 65% of the time (so 35% of the time the replaced page is clean). Note: a dirty page is one that has been written to since it was loaded into main memory. It takes longer to replace a dirty page because the operating system has to write the contents of the dirty page back to disk.

Lengthy discussion to help you with problem 8

The page-fault rate is defined to be the fraction of accesses to virtual memory that result in a page fault. If 10,000 accesses to virtual memory result in 40 page faults, then in this case the page fault rate is 40/10000 or .4%. I will use the symbol $R$ to stand for page-fault rate, to avoid confusion with using “$P$” to stand for “probability”.
To find a bound on \( R \), write an equation for the effective memory access time; this is a weighted sum of slow and fast memory access times. Two examples of this type of equation are given in the textbook.

(a) Near the end of section 8.5.2 (8.4.2 in 8th edition) Eighty percent of the time the access is fast: 120 nanoseconds computed as TLB hit adding 20 nanoseconds to the raw memory access time of 100 nanoseconds. Twenty percent of the time the access is slow: 220 nanoseconds, because address translation overhead is 20ns for TLB miss plus 100ns to read the page table. The effective memory access time is 140 nanoseconds, a 40% slowdown compared to raw memory access. If the TLB hit rate goes up to 98%, the effective access time drops to 122 nanoseconds (only a 22% slowdown).

(b) Near the end of section 9.2.2 The fast access time is 200 nanoseconds (no page fault) and the slow access time is 8 milliseconds (with page fault). For performance degradation to be less than 10%, the page fault rate must be less than 0.0000025.

For this homework problem you do a similar calculation that combines four types of memory access. From fastest to slowest, the four types of memory access are (1) TLB hit, (2) TLB miss and no page fault, (3) TLB miss and a clean page fault, and (4) TLB miss and a dirty page fault. Compute the effective memory access time as the sum of four terms, where each term is weighted by the frequency of that type of memory access:

\[
A*\text{TLB hit time} + B*\text{TLB miss time} + C*\text{CleanPageFaultTime} + D*\text{DirtyPageFaultTime}
\]

You are told \( A = .95 \), and you know that the sum \( A+B+C+D \) must total to one. Your goal is to find a bound on the page fault rate, \( R \). To calculate this, figure out how to write \( B, C, \) and \( D \) in terms of \( R \). For example, \( B = .05 - R \).

After you have computed a value for \( R \), you can do a sanity check as follows. Calculate \( R*8\text{ms} \) and if that comes out to be bigger than 200ns then you know you made a mistake. The effective memory access time is certain to be larger than \( R*8\text{ms} \), since 8ms is the fastest time for processing a page fault, and you are saying that it is ok for page faults to occur at the rate \( R \). This reasoning demonstrates that \( R \) must be less than 0.000025. This is a rough upper bound on \( R \) – the correct answer for \( R \) is quite a bit smaller than 0.000025.

A page fault cannot occur when there is a TLB hit. Some students get confused about conditional probabilities. (Skip this paragraph if you don’t worry about conditional probabilities.) Let the event “TLB hit” be denoted by \( A \), “TLB miss” be denoted by \( B \), and “page fault” be denoted by \( R \). Events \( A \) and \( B \) partition the space: always exactly one of \( A \) or \( B \) occurs. Write this as \( P(A \cup B) = 1 \) and \( P(A \cap B) = \emptyset \). Because of this partitioning, the probability of any event \( X \) can be written as \( P(X) = P(X|A)P(A) + P(X|B)P(B) \). Thus, one way to calculate the probability of a page fault is \( P(R) = P(R|A)P(A) + P(R|B)P(B) \). Because \( P(R|A) = 0 \), this simplifies to \( P(R) = P(R|B)P(B) \). So if you choose to proceed by first determining the value of \( P(R|B) \), then remember to multiply by \( P(B) \) to get the final answer.
A computer uses segmentation with paging. A virtual address consists of six hexadecimal digits, SSPPPEE, where SS is a segment number and PPEE is the offset within the segment. The segment is paged, with PP as the page number and EE as the page offset. (The page size is $100_{16} = 256_{10}$.) Physical addresses are four hexadecimal digits.

The figure below shows selected parts of the current memory contents. All numbers are hexadecimal. Segment and page tables are indexed starting at 0. Two-digit frame numbers are stored in the segment tables, in the page tables, and in STBR (Segment Table Base Register). These frame numbers are converted to main memory addresses by appending $00_{16}$. For example, the entry 0F in a segment table means that the segment's page table is located at address 0F00.

A * means the page is on disk. The segment table and page table labels are given here as a hint; the operating system does not use such labels during address translation.

```
<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Main Memory Address</th>
<th>Is There a Page Fault?</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000B2</td>
<td></td>
<td>YES</td>
</tr>
<tr>
<td>020216</td>
<td></td>
<td>YES</td>
</tr>
<tr>
<td>02015E</td>
<td></td>
<td>YES</td>
</tr>
<tr>
<td>0202A7</td>
<td></td>
<td>YES</td>
</tr>
</tbody>
</table>
```

b) Following the four accesses performed in part (a), the system switches execution to another process. The value "02" is loaded into the STBR. What segments, if any, are shared between this new process and the previous process?

c) Describe what happens when the new process (STBR value 02) references address 030142. Keep using the list of free page frames from part (a). Write changes to the segment or page tables into the above figure. Can you determine what main-memory address results after address translation? If so, state the address. If not, state what additional information is needed.