CS-537: Midterm Exam (Fall 2004)
Midterm Harder: The Solutions

Please Read All Questions Carefully!

There are eight (8) total numbered pages.

Please put your student ID (but NOT YOUR NAME) on every page.

Name and Student ID: _____

Part I: Short Questions

The following questions require short answers. **Each of 12 is worth 5 points (60 total)**.

1. In a traditional single-threaded process, we have both a **stack** and a **heap**.

   - What is the stack used for?

     Dynamically-allocated data that is allocated and deallocated in a predictable order, namely things like procedure call parameters and local variables. Hence, this memory tends to be compiler managed (the user does not have to be involved).
   - What is the heap used for?

     Dynamically-allocated data that is allocated and deallocated in an unpredictable manner, such as arbitrary data structures like linked lists, etc. Hence this memory tends to be user managed (requiring calls to allocate memory, and depending on the language, deallocate memory too).
   - If, for some reason, you could only have one of these two, which would you pick and why?

     The heap, because it is more general albeit slower.

2. Assume you have a free list that consists of the following free chunks, in this order from the head of the list: 10 bytes, 20 bytes, 40 bytes, and 10 bytes. Then assume you get the following allocation requests: allocate 10, allocate 15, allocate 10, allocate 35.

   - Using **first fit allocation**, will all requests succeed?

     First fit fails upon the fourth request, for 35 bytes. Why? Because the 10 byte chunk is used by request 1, the 20 byte chunk used by the 15-byte request 2 (leaving a 5-byte chunk), and then the 40-byte chunk gets used by request 3 (leaving a 30-byte chunk). Hence, when the 35-byte request rolls around, there is a 5-byte, 30-byte, and 10-byte chunk, leading to failure.
   - Using **best fit allocation**, will all requests succeed?

     Yes. Request 1 is served by the 10-byte chunk, request 2 by the 20-byte chunk (leaving 5), request 3 by the last 10-byte chunk, and request 4 by the 40-byte chunk (leaving 5).
   - In general, which is better, first fit or best fit?

     Hard to say. Could make the argument that you can't say because you don't know what the workload is like. Could also argue first-fit (because it's likely to be faster), or best-fit (because it avoids problems like the above).

3. With **dynamic relocation**, the hardware has a **base** register and a **bounds** register, which it uses to support multiprogramming.

   - Imagine if you just had a **base** register; what functionality do you lose with the loss of the bounds register?

     Bounds gives you protection, checking that references stay within the address space of a given process.
   - Imagine if you just had a **bounds** register; what functionality do you lose with the loss of the base register?

     Base gives you the ability to relocate through hardware.
   - In a multiprogrammed system, if you could only have one such register, which would you choose, **base** or **bounds**? Why?

     Simple answer: multiprogramming demands relocation (to enable multiple processes to run within the same memory). Hence, the base register.

     Other accepted answer: bounds, because we can use static relocation (done via the loader) to get relocation.

4. This question is about **external fragmentation**.

   - Define it.

     External fragmentation is the type of fragmentation that occurs outside of the unit of allocation. Hence, the free space is broken up into a number of holes, often of variable size and hence potentially problematic under an arbitrary request stream.

   - Give an example of where it occurs.

     It occurs in heap-based memory management. Others gave more specific examples, which was also OK.

5. Name and describe **two** advantages that **segmentation** has over simple **dynamic relocation**:

   - Advantage #1:

     Permits sharing (say of code pages).

   - Advantage #2:

     Supports sparse address spaces (all of those pages in-between the stack and the heap no longer need be allocated).

6. Envision a system that uses **pure paging** (i.e., no segmentation) and a hardware **TLB**. Also assume the the TLB is **software managed**, i.e., any updates to the TLB are handled by the operating system.

   - What happens on a **TLB hit**?

     A TLB hit is completely handled by hardware. TLB takes the VPN from the address, looks up the matching entry in the TLB to find the PPN, appends the offset from the original address, and presents the fully-qualified physical address to memory.

   - What happens on a **TLB miss**?

     A TLB miss follows the same path above until the TLB is consulted and it is found that the matching entry for this VPN is not in the TLB. At that point, a fault is generated, jumping into the OS to handle the problem (it is a software-managed TLB after all). The OS then consults the page table, and, assuming that the relevant page is in memory, takes the PPN, installs it in the TLB (removing another entry), and restarts the instruction.

   - What happens on a **page fault**?

     The same path as above is taken, except that when the page table is consulted, it tells you that the page is not resident in memory. Hence, one must find the page on disk and bring it into memory. If there was no free memory available, a page replacement algorithm must first be run, evicting an existing page, and if that page was dirty, it must first be written to disk. Once the new page is brought in, the page table and TLB must be updated, and the instruction restarted.

7. This question is about the contents of a typical TLB.

   - Sometimes a TLB will contain two entries that have the same **physical address** – when?

     I really should have said the same "physical page number" – the TLB of course does not hold the offset part of the addresses. However, any time two separate processes are *sharing* a page of memory, the TLB will have two entries that refer to the same physical page.

   - Sometimes a TLB will contain two entries that have the same **virtual address** – when?

     I again should have said "virtual page number". The TLB will hold entries with the same virtual page number when two different processes have entries in the TLB at the same time.

   - Do either or both of these cases require extra hardware support from the system to work properly?

     Really, both do. In the first case, we have sharing, and hence undoubtedly need protection bits. In the second, we need to distinguish entries with the same VPN, and hence need an address-space identifier of some kind.

8. In this question, we discuss the **clock** replacement strategy.

   - Describe how clock works. What hardware support is needed? What software structures must be kept?

     Clock uses reference bits (or use bits) to approximate LRU-based page replacement. Upon a memory reference to a page, clock assumes that the hardware will set a bit saying that the given page has been accessed. The OS, when it needs a page, will then sweep through the pages of memory in a circular (or clock-like) fashion, looking for a page that has not been accessed recently (i.e., its use bit has not been set). In doing the sweep, the OS also clears the bits as it examines them, and hence in the worst case will sweep all the way around before replacing a page (in the case that all pages had been accessed).

   - Can clock ever behave exactly like "perfect" LRU? (describe)

     I basically took answers that went either way here. If you said yes, you probably meant that it could if the reference stream were just right and it matched LRU's behavior. If you said no, you probably argued that clock just does "NRU" replacement (not recently used), and hence is only an approximation of LRU.

9. This question is about physical addressability in a system that uses **paging**. Let's say we have a **20-bit virtual address**, with a **4 KB page size**.

   - Let's assume that the system we're running upon has a maximum of 1 GB of physical memory. How big is each page table? (assume 2 extra bits of information are needed beyond the usual stuff).

     Size of a page table: number of entries times the size per entry. Number of entries here: determine by the number of pages in the *virtual* address space. With a 20-bit address space and a 4 KB page size, we have 8 bits of each address which determine the number of virtual pages in the address space, and hence 2^8 or 256 pages. Hence, 256 entries.

     Now we must calculate the size per entry. For this, we have to realize that each entry must hold the corresponding physical page number, plus 2 bits. In this part of the question, it is stated the we have 1 GB of addressable physical memory. 1 GB is 30 bits, and we have to subtract the size of the offset (12 bits again from the 4 KB page size) to get an 18-bit physical page number. We add the 2 bits and get 20 bits per entry.

     So the answer is 256 entries, each of size 20 bits. Because we probably have to use bytes and not bits, a very reasonable answer is 256 entries each of size 3 bytes, or 768 bytes.

   - Let's assume a different system we're running upon has a maximum of 64 KB of memory. How big is each page table? (again assume the 2 extra bits).

     Similar reasoning for the number of entries: 256 (we are using the same virtual address space after all).

     As for the number of bits needed for the PPN, this time we have 64 KB addressable physical memory. 64 KB implies only 4 bits for the PPN (with 12 for the 4-KB offset again), and we again add 2 bits to get 6 bits per entry.

     So the answer is 256 entries each of 6 bits. Again rounding to the neartest byte per entry, 256 bytes is your answer.

- Which of the preceding two cases is worse, having more physical memory than your process can address, or less? Why?

  Having more physical memory than your process can address is a bad thing – it implies the only way to use the entire memory is through multiprogramming. A single program with large memory requirements could not.

  Some people said the latter was worse because it leads to paging, but this confuses the issue of paging with the issue of addressability.

10. In this question, we explore page cache replacement strategies.

    Assume you have the following page reference stream: A, B, C, D, A, B, E, A, B, C, D, E.

    - Assuming a page cache of size **3 pages** and a **FIFO** replacement policy, how many misses will there be?

      9 misses. A - miss (afterwards, in cache: A), B - miss (cache: AB), C - miss (cache: ABC), D - miss (cache: BCD), A - miss (cache: CDA), B - miss (cache: DAB), E - miss (cache: ABE), A - hit (cache: ABE), B - hit (cache: ABE), C - miss (cache: BEC), D - miss (cache: ECD), E - hit.

    - Assuming a page cache of size **4 pages** and a **FIFO** replacement policy, how many misses will there be?

      10 misses. A - miss (cache: A), B - miss (cache: AB), C - miss (cache: ABC), D - miss (cache: ABCD), A - hit, B - hit, E - miss (cache: BCDE), A - miss (cache: CDEA), B - miss (cache: DEAB), C - miss (cache: EABC), D - miss (cache: ABCD), E - miss (cache: BCDE).

    - Does the comparison between the 3-page and 4-page caches surprise you in any way? Why?

      Most (who got the numbers right) said "yes", because bigger caches are supposed to be better. Some said "no", because FIFO is a silly policy and can lead to funny anomalies like this (in fact, this one is called "Belady's anomaly"). If you are interested, you can also prove to yourself that this *cannot* happen for an LRU-managed cache.

11. **Thrashing** occurs when more memory is being actively utilized than the system contains. When talking about thrashing, one often refers the **working set** of a process.

    - Define the "working set" of a process.

      Simply, the number of unique pages accessed by the process within the last T seconds. Some also said the number of pages a process is using at a given time or in a given time frame.

    - If a system is thrashing, how can we try to reduce thrashing **within the OS**? (i.e., how would we change the OS?)

      Thrashing implies that we are repeatedly evicting pages from memory and fetching new ones, hence making slow progress. The OS could handle this in a number of ways. However, one really good way is to use admission control: by descheduling some of the jobs causing the thrashing, we can reduce the load on memory and make better progress. We could also perhaps change the replacement policy, but that might be harder to get right.

    - If a system is thrashing, how can we try to reduce thrashing **with hardware (of some kind)**? (i.e., how would we change the hardware?)

      Simplest answer: buy more memory. Other more esoteric suggestions were also made, and yet often rejected by this exacting grader.

12. Assume you have a **physical address** P. Let's say this is in a system that has a typical **linear page table** structure.

    - How would you find out which virtual address(es) are mapped to P?

      Given a linear page table and a physical address P, one must extract the PPN from the address and scan the table looking for any and all addresses that map to this physical page.

    - What kind of data structures might you add to speed up this process?

      Something like an inverted page table, i.e., something that hashes physical page numbers to the virtual pages that are mapped to it.

Part II: Longer Questions

The second half of the exam consists of two longer questions, **each worth 20 points (total 40)**.

1. **Improving your memory.**

   In this question, we consider a new hardware system that has two types of memory. **Primary memory** is the type of memory attached to the system in the typical manner (e.g, on a memory bus). **Secondary memory** is additional memory that you add for more capacity, but it's added onto the I/O bus. The implication of this structure is that **secondary memory accesses are slower than primary memory accesses** (but still of course are much faster than disk accesses).

   Note that secondary memory is addressed just like primary memory. Specifically, the hardware looks like it just has one big physical memory. However, the lower part of this "physical" memory is the faster primary memory, and the upper part is slower secondary memory. The OS can also easily find out where the boundary is, and thus can be intelligent in how it uses primary versus secondary memory.

   We now discuss how OS design must change to accommodate the new hardware.

   (a) Let's say you decide to use a combination of segmentation and paging in the OS for this system. Describe how a combined segmentation and paging approach works, and how this hybrid approach improves on strict paging and strict segmentation.

   Paging is great because it removes the problems of external fragmentation – all requests are for page-sized entities and hence can be serviced by any free page. However, pure paging requires really large page tables, with one entry per virtual page. Adding segmentation solves this problem, by requiring only one entry per virtual page in each segment (and removing the need for all of the entries in the middle part of the address space between the stack and the heap). Hence, paging plus segmentation avoids the problems of external fragmentation while supporting large, sparse address spaces effectively.

   (b) In this system, you of course will have page tables. Should you change how the page tables are structured in this new system? (if so, how, if not, why not)

   Some said yes, in that minimally you might want to track some info about which type of memory the page is residing in (primary or secondary). I accepted this, although it is redundant: simply knowing the physical address tells you which type of memory it is in.

   Some said yes, and suggested putting the page tables in primary memory to speed up access. I liked this answer.

   Some said no: although we have primary and slower secondary memory, the page tables could remain intact. This was acceptable with good justification.

   (c) You also decide to use LRU replacement for the page cache. Describe how LRU replacement works.

   Standard "perfect" LRU replacement works by evicting the least-recently used page when a free page frame is required. The newly brought in page becomes, of course, the most-recently used page.

   (d) Should you change how your basic LRU algorithm works in this new system? (if so, how, if not, why not)

   The new system might want to try to keep pages that are being heavily used by the system in primary memory, to speed up accesses. One excellent way to do this is to treat secondary memory like another cache in the hierarchy, some people suggested, and I liked that suggestion. Others suggested more complicated schemes which moved pages back and forth between primary and secondary depending on their LRU order. This was also quite acceptable. Others said that they would prefer to evict pages from primary memory, because the eviction was faster(!), which I did not like at all but gave some credit for – our goal is not fast eviction but rather to make sure the program runs fast when it is accessing memory!

2. **Size does matter.**

   In this question, we try to understand the issues that surround the choice of **page size** in a system. In particular, we will discuss some new systems that have support for **two** pages sizes, one **"small"** sized (say 4 KB), and one that is **"big"** (say 1 MB).

   (a) If we just have a single page size, and it is quite **big**, what are the possible negative consequences? In contrast, what negative consequences arise if our page size is **too small**?

   Bigger pages have the problem of increased waste due to internal fragmentation.

   Smaller pages have the problem of increasingly large page tables.

   (b) Using **big pages** can improve performance. Which **hardware resources** of the system are better utilized with big pages?

   Big pages are primarily used to reduce pressure on the TLB – with big pages, a much bigger portion of your address space can be actively mapped by the TLB, and hence your TLB hit rates improve. If you didn't say "TLB" in your answer, it was hard for you to get full credit.

   Big pages may also reduce memory pressure because you have smaller page tables and hence more other stuff can fit into memory.

   Finally, big pages may improve disk access times, as it is generally more efficient to move large pages in and out of disk.

   (c) To support multiple page sizes, some aspects of the **page table** must **change**. Describe the most important changes needed, and how you would implement them. Assume a simple **linear page table**.

   I accepted answers that minimally realized that different page sizes implied a differing number of bits needed in the PPN in each page table. However, it's a little more tricky than that, because you also have to index the page table with the VPN of the address, so a more complete answer would have to deal with that issue as well.

   (d) Now imagine a scheme that tries to make use of big pages where possible. Specifically, the OS first hands out small pages when a process asks for more memory. Then, the OS periodically tries to **convert** batches of these small pages into big pages. Describe how the OS would do this – what must be true to convert small pages into a big page? What (software/hardware) structures must be updated?

   To convert a set of small pages into a single big one, the OS must look for a set of small pages that are aligned with the big page, and are contiguous in both the virtual address of the process as well as in physical memory. These pages also should have the same characteristics (e.g., they should have the same protection). Note that the if the physical contiguity requirement is not met, the OS could reshuffle pages (at some cost) to meet the requirement. Once such a set of small pages is found, they can be promoted to a large page, which would involve changing the page table to reflect such a change, as well as the TLB.