

ITEE



What's New:

Some questions aren't answered in as much detail as the rest, to give you something to think about when working through the solutions later.

Learning objectives for this week:

- Understand the major concepts we've covered, including:
 - why memory management is needed
 - differences between segmented and paged models
 - how paging is implemented
- Apply these concepts to a specific situation, to understand how memory is being managed
- Apply these concepts to choosing an appropriate strategy for a given situation
- Understand where and how different strategies apply, and their significance in real systems

As before, you should aim to answer all of these questions by the end of the course. Being able to answer them by the end of week 8 is a useful goal. Be prepared: if you know what you can't do yourself, you will make most efficient use of the tutorial.

1. Concepts

- a. Why is non-contiguous memory allocation desirable? Consider what would happen otherwise if you had to load several programs into memory, and load new programs into memory after others finished executing.
External fragmentation can occur in the described scenario with contiguous memory allocation. This is bad because it may prevent you from executing programs even if you have enough memory available.
If you don't get it, draw a picture of a memory of 16 bytes in which the following sequence of operations occurs
 - P_0 allocates 6 bytes starting from location 0
 - P_1 allocates 4 bytes starting from location 6
 - P_2 allocates 4 bytes starting from location 10
 - P_0 terminates
 - P_3 needs to allocates 8 bytes
- b. A segmented system corresponds to a programmer's view of memory whereas a paged system does not. True or false? Explain your reasoning.
True. Paging does not care about where things are placed in memory, whereas segmentation groups like things together, e.g., a code block and a data block (similar to a programmer's view of memory). A segmentation system can treat a specific code-level entity (procedure/function.method, array, or object, to quote some examples) as a logical unit of allocation. Some segmentation schemes are this fine-grained; others may have 1 segment for all the code, 1 for the stack and 1 for other data.
- c. Page table design is a matter of trading memory for speed. Discuss this statement, using examples from the lectures
The fastest page table lookups require more memory, e.g., forward lookups are fast but take a large amount of memory. 2-level forward lookups use memory more effectively (allowing parts of the page table to be left out or even paged), but are slower. Inverted tables take less space if you have many processes, but looking up a page's physical frame can be slow. In general, designs that use more memory are faster. Digging deeper, while schemes which use less memory may appear slower for reasons like requiring more operations to get to the page translation, in practice, they may be faster. Anything which significantly increases total memory use will impact on the overall memory hierarchy (for example, the caches). Although the caches are not part of the page table implementation, accesses to page tables, like anything else in memory, will result in parts of the table ending up in the caches. If big parts of the table are empty, or entries are less compact than they need to be, the caches could have useful information pushed out by useless parts of the page table. A cache miss can add tens to hundreds of extra CPU cycles to an instruction execution, so doing 3 memory access which are all cached is a lot faster than doing 1 which results in a cache miss.

2. Relocation

- a. When a disk produces data as a consequence of a disk read, it needs to write the results to a location in memory where it knows the absolute address. Explain why this may cause problems for any scheme which allows relocation, and how these problems can be solved.
The write buffer may be relocated before the data is written, causing the wrong data to be written. If the disk controller goes through the memory scheme, these problems can be avoided (although at a performance cost). Further, if the disk controller uses the paging mechanism, it will inevitably interfere with CPU operation, since paging is tightly integrated with the CPU for performance reasons. As a result, it would be difficult to maintain the separation between I/O operations and the CPU aimed for in DMA. A disk operation can still interfere with CPU operation if the CPU needs to access memory (it may have to wait if the memory subsystem is busy), but the caches minimize the number of times the CPU accesses memory.
- b. The original Macintosh design did not include allowance for multitasking. Its memory management scheme implemented memory compaction in a way which required a programmer to be aware of when compaction may or may not occur. Why is this design poor in a multitasking environment?
It is hard for the programmer to be able to tell when relocations can occur. Consistency problems may occur if a context switch occurs during relocation. Programmers would need to be able to delay context switching, which does not suit pre-emptive systems. The original Mac OS, of course, did not implement preemptive multitasking. Any ideas on why that could have had downsides?
- c. Is relocation important in a page-based scheme (considering only the information in Chapter 9)? Explain your answer.
No. It doesn't matter which frames pages reside in. However, it may be beneficial to relocate empty frames together to reduce the overheads of maintaining a free frame list. In general, it's useful to keep related frames close together so that if for example a number of dirty pages needs to be written back to disk when a process moves to a new part of its working set, they can be written efficiently as a single block. It is also useful for free space management to keep free blocks contiguous because you can, e.g., implement a free space list node as a start address of the first free block, and a count of how many more there are in that sequence. Another reason to keep physical frames in order of virtual addresses is DRAM is more efficiently accessed using sequential operations, and

you may occasionally have code which does a long string of sequential operations. This code could be more efficient if it doesn't suddenly have to break sequence at a page boundary.

3. Address Binding

- a. In each situation, explain whether the time binding occurs is at compile, load or run time (as defined on p 275 of the textbook):
 - i. A page is not allocated in physical memory until it is first used.
runtime: can you explain why this is the correct answer?
 - ii. A page is allocated to a physical frame when the program is about to run, and once allocated stays in the same location until execution terminates.
load time: again, explain why this is a good answer.
 - iii. The operating system allows a user program to see the value of a global variable at a specific, invariable location, and the compiler is permitted to use the name of this variable, but the actual physical location is only inserted into the code when it is linked.
load time: in terms of Figure 9.1 (p 275) – link time is a more generally correct answer, but linking and loading are not differentiated as separate phases. The classification in Figure 9.1 is not an entirely natural, as static linking usually happens soon after compile time (unless you are compiling a library or a file not containing the main program). To make things more complicated, if you have a dynamic linker, linking does happen immediately before loading. So I consider it an error in the book not to split linking and loading as different times when binding could occur
- b. Early binding is easy, while late binding is flexible. Discuss this statement, in the light of any examples you know (including programming languages).
Early binding is easy as addresses can be directly included in the machine code generated at compile (or slightly later, at link) time. Late binding is more complicated as a stub is necessary to relocate code and addresses at runtime. Late binding is more flexible because it allows features such as polymorphism. Consider examples (e.g., operating systems with dynamically linked libraries, programming languages where binding of names to attributes including memory happen late ... list your own favourites) – do you associate these examples with efficiency or flexibility?

4. Page Table Implementation

Under the following assumptions (see also picture at end):

- 4KB pages
- 512MB of physical memory
- for 2-level paging, half the page number is used for each level
- page table entries are rounded up to a whole number of bytes
- a hashed page table is 80% full
- an inverted page table has one entry per physical address, and needs 25% more entries for handling hash collisions

- a. What fraction of real memory does the process need (excluding OS overheads like page tables)?

20 MB code + 512 KB constants + 16 MB dynamic allocation space + 1 MB stack = 37.5 MB

37.5 MB process memory / 512 MB physical memory = 7.3%

For anyone who doesn't get it, add up the totals in the picture.

- b. Work out the size of a forward page table (done in Week 7 classroom exercises).

4 KB pages => 12 bit offsets

32 bit address space - 12 bit offsets = 20 bit page numbers

512MB of physical memory => 17 bits needed for frame numbers

Rounding up to a whole number of bytes:

24 bit frame number x 2²⁰ = 3 MB page table

32 bit frame number x 2²⁰ = 4 MB page table

Details missing: *how do you get the number of page table entries? What has to be in each frame, to make it up to a given size?*

What do you need in addition to the physical frame number? Is it reasonable to make the physical frame number size specific to a given amount of memory, given it could be upgraded?

- c. Work out the size of a 2-level page table (done in Week 7 classroom exercises).

You need just enough level 2 (L2) tables for the actually-used address space.

Half the page number used for each level => 10 bit level 1 page numbers, 10 bit level 2 page numbers

Therefore there are 2¹⁰ level 2 page tables at most, but in this case, we need

- *20MB for code starting from address 0*
- *512KB for constants starting from address 128MB*
- *16MB for dynamic allocation starting from address 256MB and*
- *1MB for stack space growing down from the top of the address space*

Since each L2 table has 1024 entries, each covering 4KB of the address space, we need a new L2 table every 4MB. Since each of the areas in which we are starting new kinds of memory use is at an even multiple of 4MB, each one will start a new L2 table, so we can work out the total number of L2 tables simply by adding all the size requirements and dividing each by 4MB (rounded up since we have to allocate whole L2 tables):

- *20MB for code needs 5 L2 tables*
- *512KB for constants needs 1 L2 table*
- *16MB for dynamic allocation needs 4 L2 tables and*
- *1MB for stack space needs 1 L2 table*

So, in total, we need 11 L2 tables.

The L1 table just contains pointers to L2 tables, so each is a 32-bit entry, 4 bytes, for a total of 4KB. We can make various reasonable assumptions about the size of an L2 entry. The minimum is 21 bits: the 20-bit translation, plus a valid bit. If we round up to the nearest whole byte, we get 24 bits, or 3 bytes. This would make each L2 table 3KB in size. It may also be reasonable to round the table up to 4KB, so it's the same size as a page. Either way, the total memory for the page table is 11xL2 size + 4KB where the last term is the size of the L1 table.

- d. Work out the size of a hashed page table (done in Week 7 classroom exercises). Make sure you take into account extra information which would need to be stored in the page table, to identify whether a particular virtual page had actually been found.

The process uses 37.5 MB; since each "segment" is a whole multiple of the page size of 4KB, we can work out the total pages as 37.5 MB / 4 KB = 9 600 pages.

80% of the table size contains 9 600 entries, so the table has $9600/0.8 = 12\ 000$ entries in total.
 Each table entry contains a 32-bit pointer, so the table size is $12\ 000$ entries \times 32-bit pointers bytes.

Assume the page table is implemented according to the simplistic scheme of Figure 9.14, p 300, where each entry in the hash table is just a linked list. Each linked list node representing a page in the table would have a 20-bit page number, a 20-bit frame number (it could be as little as 17 bits, since physical memory is 512MB, but for efficiency, make the list node a whole number of bytes), and a 32-bit next pointer. Therefore each node requires 9 bytes. 9 600 \times 9 bytes are required for all of the linked list nodes. For efficiency, it would probably be better to increase the size of each page number and frame number stored to a whole number of bytes; it would be a useful exercise to adjust the calculation for this – in any case, we haven't taken account of the need to store status bits with each entry – valid, etc.).

$12\ 000$ entries \times 32-bit pointers + 9 600 nodes \times 9 bytes = 131.25 KB.

The 20-bit page number is required in each node so that it is possible to identify which virtual page has been found in the table.

- e. How would an inverted page table differ from a hashed page table in terms of
 - i. the situation where there is more than one process in memory?
Additional hashed page tables (one for each process) are required, compared to the single inverted page table.
 - ii. total memory usage? (Think of what else is needed to handle more than one process in using the same page table.)
Less memory would be required for the tables since only one inverted table is required per system. Collision handling (where more than one logical address maps to the same location in the table) is a bit of a challenge. Ideally, you would like to avoid storing the physical address, so it can be derived directly from the location in the table.
- f. IBM's RS/6000 RISC system used an inverted page table design, in which the low 13 bits of the page number uniquely determine the location in the page table. In other words, given that the hash function has taken you to a particular location, you know 13 bits of the virtual address. Why do you think they have implemented their inverted page tables this way?
To reduce the amount of memory required since the low 13-bits do not have to be stored in each node, i.e., if you know you are in location 42 in the table, you can work back from the value 42 to work out what the low 13 bits of the virtual page number are, instead of having to store this value in the table. As an extra exercise, it would be interesting to find out exactly how they implement the inverted page table.
- g. What would work well as a hash function for the hashed and inverted page tables?
You could use a combination of low and high bits from the address to attempt to scatter adjacent values. In general, finding a good hash functions is a hard problem, and depends on experimental determination of likely memory access patterns.
- h. Which strategy is best in terms of page table size, in this situation? Which would be best if you had many processes in memory at the same time?
The hashed page table will be smallest in this situation where there is only one process (the inverted table needs to store process IDs so it's not smaller if there is only one process). If you had multiple processes, the inverted page table would be the most compact.
- i. Now, redo the previous parts of the question assuming that memory is allocated in a large number of small, randomly scattered objects, each less than a page in size. Think through the differences, rather than redoing the numbers in detail. In this scenario, would your idea of the best strategy change?
When the allocated objects are small enough, the size of the hash table will approach the size of forward tables. In this case, it would be more efficient to just use forward tables. Anything which tends to increase the actual number of pages used will tend to make inverted or hashed page tables less of an option, because a larger fraction of the page table is actually used.

