

ITEE



What's New:

Questions this time, in addition to building on lectures, are aimed at providing a start for Assignment 2.

Learning objectives for this week:

- Understand the major concepts we've covered, including:
 - why virtual memory is needed and why it works
 - differences between paged replacement algorithms
 - how paging policies draw in underlying concepts and theories
- 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

I have given reasonably complete answers in most cases except for the coding part, which you should be able to pick up from the given code in Assignment 2, or your own solution. If you didn't do the coding question, the chances are that you won't want to do any coding examples in the exam, but if you feel otherwise, let me know, and I will provide more detail on any points you can't do.

1. Concepts

- a. Why is it useful to be able to have more virtual address space across ready processes than the available physical address space? Consider the total virtual address space of a program with 32-bit addressing, and the possibility of 100 or more processes running on a Linux system with only one user, not running a very high number of programs, and how the number of processes is likely to scale as users are added.
The total virtual address space here would be hundreds of times the size of physical memory. Each process can use its virtual address space the way most convenient for that process, without concern for how any other process is using memory. Allowing each process to have its own address space allows code to be written in a very general way, without concern for how big the physical memory is, or for different configurations on which the process might run in future.
- b. A segmented system is not a popular option for implementing virtual memory. Explain why.
Segmentation leads to problems of external fragmentation: if several small segments not contiguously allocated become free, there may be sufficient total free memory to allocate a new large segment, but it may not be available in one place. An alternative is paged segmentation (segments are divided into fixed-sized pages) but the advantage of this model over pure paging is slight. Can you think of any advantages of this model over pure paging?
- c. Explain how having a swap system designed so that a given process's pages are contiguously allocated (i.e., so that if you know the location of the first page, page n is simply n page-lengths away on the disk) reduces the amount of information an operating system needs to store to manage page replacement.
You only need store the start address of the process. Page n can be found at start address + page size $\times n$ on the partition.

2. Memory-Mapped Files

- a. What is the difference between using a memory-mapped file, in the following sequence of steps (system calls are approximated from real calls for simplicity):


```
mmap(fileName, startAddress);
// file contents can now be accessed through the variable startAddress
// as if in memory starting from that variable's location
startAddress[42] = 'k'; // for example
munmap(startAddress);
// file contents no longer in address space -- file location 42 now contains letter 'k'
```

as opposed to the following sequence of steps:

```
startAddress = blockRead(fileName); // read all of file into memory
startAddress[42] = 'k';
blockWrite(fileName, startAddress); // write whole file back to disk
fclose(fileName);
```

In the first case, only the modified part of the file needs to be written back, and this is handled automatically by the OS. In the second case, the whole file is written back, which is rather wasteful by comparison.
- b. In more general terms, discuss advantages and uses of memory-mapped files.
Memory-mapped files make it easy to modify small parts of a file by treating the contents as part of the virtual address space. Parts of the file which are read or written can be efficiently transferred from or to disk as needed using the paging mechanism, which saves the programmer having to think through a specific strategy for deciding which parts to represent in internal data structures. The OS can use the same technique to page in code, which can be done simply by mapping the executable code file into the process's address space, eliminating the need to copy the code over to a paging partition or file.
- c. Look up how memory-mapped files are used on an operating system of your choice (hint: try looking for man pages on a UNIX-like system for the real versions of the system calls in the example in (a)).
A useful exercise to extend your knowledge: do this if you have time.

3. Replacement

- a. Construct a reference string to illustrate Belady's anomaly with the FIFO replacement strategy, when the number of physical frames is increased from 4 to 5. Can you see a general pattern? Is this problem likely to arise in a real-world scenario, where the number of page frames is usually increased by a (reasonably large) power of 2 when memory is upgraded?
1 2 3 4 5 1 2 6 1 2 3 4 5 6
to see the problem when the page size goes up to N , make a string of references $1 \dots N$, followed by $1 \ 2N+1$ then by $1 \dots N+1$. This specific pattern shouldn't cause the problem to arise in real examples. I'd be interested if someone can devise a more "real" example.

b. For the following reference string, with 3 physical frames:

4 3 3 2 2 2 1 2 3 4 1

i. Show the sequence of page faults in the notation used in lectures for each of the following algorithms:

- optimal

Page faults shown in red (total 5):

4 3 3 2 2 2 1 2 3 4 1

4	4	4	4	4	4	1	1	1	1	1
	3	3	3	3	3	3	3	3	3	4
			2	2	2	2	2	2	2	2

- FIFO

In this case, coincidentally the same as the optimal strategy.

- LFU

Slightly different from the previous cases – 1 more page fault.

4 3 3 2 2 2 1 2 3 4 1

4	4	4	4	4	4	1	1	1	4	1
	3	3	3	3	3	3	3	3	3	3
			2	2	2	2	2	2	2	2

- MFU

A few more misses here but mainly because we default to least recently used when most frequently used doesn't apply (all equally often used since last brought into memory). What other fallback strategies could be used as a tie-break?

4 3 3 2 2 2 1 2 3 4 1

4	4	4	4	4	4	4	4	3	3	3
	3	3	3	3	3	3	2	2	2	2
			2	2	2	1	1	1	4	1

- LRU

4 3 3 2 2 2 1 2 3 4 1

4	4	4	4	4	4	1	1	1	4	4
	3	3	3	3	3	3	3	3	3	3
			2	2	2	2	2	2	2	1

ii. What can you say about each algorithm's efficiency, based on this example? Are your observations likely to be true in general?

On the numbers here, optimal is best (as expected) and so is FIFO (not expected, but could happen). Least frequently used is only 1 behind, tying with least recently used. Most frequently used does the worst by a significant margin. FIFO is known to have strange behaviour (e.g., Belady's anomaly, so this "good" result is likely a coincidence. LRU is generally considered one of the better approximations to the optimal algorithm, so in a real situation, would like look better than here. The very bad showing of most frequently used could occur, but would depend on the memory reference pattern. Certainly, the principle of locality suggests that MFU is not a good policy.

c. Explain why the counting and stack approaches to implementing least recently used are impractical, as compared with the approximations.

Every time a page is referenced, there is a significant amount of work, which is not practical – it would impose a severe overhead on normal processing.

4. Working Sets

a. Explain why thrashing occurs in general; more specifically, in each of the following scenarios, will thrashing occur?

In general: the total working set size of all processes running on a system is greater than physical memory. This is an approximate statement, since the working set size is dynamically determined, and refers to a probability rather than a certainty that a given amount of memory is needed at a specific time.

i. 4 processes are ready to run, each of which has a working set of 25% of physical memory.

Assuming here that "physical memory" means that which is available to the processes (i.e., not including OS memory), then thrashing should not occur.

ii. In scenario (i), one process moves to a new locality 1 page larger in size than the previous locality, and the other processes continue unchanged.

In this situation, thrashing could occur though 1 page too big, if the replacement strategy is good, may not cause a serious slowdown.

iii. In scenario (i), a new process is started, while the others continue unchanged.

Assuming the new process is nontrivial in size, we are likely to have total working sets which significantly exceed the physical memory, so thrashing is very likely to occur.

b. Explain why approximating the working set by counting references over a specific window Δ , can sometimes be less accurate than you'd like.

You can fail to capture the situation of switching from one locality to another (if Δ is too big, you could over-estimate the working set size for this reason).

c. Explain why a purely local replacement strategy is unlikely to be useful in implementing a working set-based replacement policy.

You need to manage the total working-set size over all processes. If one process needs another page and can't get it without replacing a page in another process, you can't adjust for the case where a process is moving to a new larger locality while another is moving to a new smaller locality.

d. Explain how a proportional allocation strategy could be adapted to ensure that a process's working set was always in memory.

Set the working set size estimate as the minimum for each process. If any can't be allocated its minimum after adjusting the proportions, suspend processes until the number of available pages is big enough. Alternatively: allocate each process its working set, then divide up any remaining pages proportionally.

- e. How is the Windows NT approach of *clustering* pages related to working sets? Think of what happens when you move from one locality to another.
By bringing in pages surrounding that which caused the page fault, you are betting that a change in locality has occurred on a page fault, and the page fault is not an isolated event. If this bet is correct, pages near the faulting page will be needed soon. Since a disk is more efficient at transferring larger units, you can reduce the total amount of time waiting for the disk, if the "bet" is mostly accurate.

5. General

- a. Explain why TLB reach is a crude measure. Consider the case where a program which accesses 1024 objects each of size 4 bytes in a sequence such that each object is on a different page. If the TLB has 32 entries, and a page is 4KB, calculate the following, to provide data to answer the question:

- TLB reach
 $32 \times 4KB = 128KB$
- total memory needed for the objects
 $4KB$
- number of TLB misses, assuming no other memory references affect the TLB, and no page translations in the TLB to start with

Since each object is on a different page, each page will generate a TLB miss, total 1024 TLB misses.

Overall: TLB reach is much bigger than the total size of the objects (32 times), yet the TLB in this situation is (almost) useless (we don't know how many times each object is referenced so the TLB would have some value). TLB reach doesn't really tell us what fraction of the memory we access is covered by the TLB, because there is nothing to say that memory references will be close together.

- b. Explain why it's useful to keep a pool of free pages. Specifically, consider the time taken to handle a page fault when a writeback occurs, and the possibility that a page which is evicted may be needed again soon.
When a writeback occurs, it isn't really necessary for a process to wait for the write to complete so if a page which has to be written back is chosen for replacement, a 2-step process is useful. First, grab another free page, and let the process wanting a free page continue. Second, do the writeback in the background (after which the page becomes an ordinary free page). Secondly, if a page is evicted then needed again soon, if there is a pool of free pages, the page which is evicted is not immediately allocated to another use, and so can be retrieved.

6. Implementation Issues

Under the following assumptions:

- 4KB pages
- 32-bit addresses
- large stretches of the virtual address space are likely to be unused, but the used portions are likely to be reasonably big (not just 1 or 2 pages)
- 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
- in C, you can select specific bits by doing a bitwise **and** with a *mask* which has 1s in the required positions, for example, to select the high 10 bits of `vAddress`:

```
vAddress & 0x3FF
```

 if `vAddress` had the value `0xC148`, the result would be `0x148`
- in C, you can shift bits left or right respectively using operators `<<` and `>>`. So, for example, if you want to discard the *D* bits representing the page displacement, and use the rest of the virtual address as an array index (as you would do in a single-level page table):

```
pageTableIndex = vAddress >> D;
```
- in C, you can set a specific bit by doing a bitwise **or** of a number in which that specific bit is set with the number in which you wish to set the bit, e.g., if `validBit = 0x80000000`:

```
physPageAndStatus = validBit | physPage;
```
- in C, you can unset a specific bit by doing a bitwise **and** of the *complement* of a number in which the bit is set (i.e., invert the bits) and the number in which you want the bit unset, e.g.:

```
physPage = (~validBit) & physPageAndStatus;
```
- in C, any value which isn't 0 tests as *true* in a condition (**if**, **while**, etc.)

- a. Explain why each of the given C examples works.
This is of most interest if you are keen on coding and the idea was to have you work it out for the assignment. If you still need help on this, let me know.
- b. What is the best way to split the page number for a 2-level page table? Would you use the high 10 bits to select the 2nd-level (or L2) table, then the low 10 bits to select a page, or vice-versa? Explain your answer.
It makes most sense to use the 10 high bits for the L1 table, because if large stretches of the address space are empty, this will reflect in a large number of L1 entries being NULL, i.e., no corresponding L2 table. If the low 10 bits were used for the L1 table, a high fraction of entries would be used, reflecting the fact that the lower bits are not selecting as big a range of addresses.
- c. Give C code to do each of the following, assuming a type `Address`, in which you can store a 32-bit unsigned number:
- i. discard the bits representing the page displacement
This is in the given code – if you can't find it, let me know.
 - ii. select the low 10 bits of the resulting number
Quite straightforward assuming you have a mask as specified in the list of assumptions.
 - iii. select the high 10 bits of the number from part (i), and shift them right, so they now occupy the low 10 bits
This could be done quite simply by shifting first then masking the same way as in (ii).
 - iv. discard the bits representing the page displacement in a physical address, and set the valid bit
In the given code for other page table implementations. Again, let me know if you need help finding it.
 - v. given that you have a virtual address `vAddress` and have found a page table entry `physPageAndStatus` containing the physical address in the following format
 1 bit to indicate whether the entry is valid, 11 bits unused, 20 bits for page table
 check if the entry is valid, and if it is, move the frame number to the high end of the word and add in the displacement bits from the virtual address, with the result in `physAddress`
If you did the coding version, you should have something like this. If you still need help, please ask.
- d. The representation in (c)[v] leaves 11 bits unused. If you have a type `Byte` available which stores 8 bits, how would you store the physical page frame and status bit more compactly? Give C code to store and retrieve the page frame, corresponding to the steps in part (c).

Again there is code to do this in the given page table implementations.