

ITEE



What's New:

Questions this time, in addition to building on lectures, are aimed at getting you thinking about implementation issues 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

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 9 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 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.
- b. A segmented system is not a popular option for implementing virtual memory. Explain why.
- 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.

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);
```

- b. In more general terms, discuss advantages and uses of memory-mapped files.
- 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)).

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 in a real-world case, where the number of page frames is usually increased by a (reasonably large) power of 2 when memory is upgraded?
- b. For the following reference string, with 3 physical frames:


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

 - optimal
 - FIFO
 - LFU
 - MFU
 - LRU
 - i. Show the sequence of page faults in the notation used in lectures for each of the following algorithms:
 - ii. What can you say about each algorithm's efficiency, based on *this* example? Are your observations likely to be true in general?
- c. Explain why the counting and stack approaches to implementing least recently used are impractical, as compared with the approximations.

4. Working Sets

- a. Explain why thrashing occurs in general; more specifically, in each of the following scenarios, will thrashing occur?
 - i. 4 processes are ready to run, each of which has a working set of 25% of physical memory.
 - 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.
 - iii. In scenario (i), a new process is started, while the others continue unchanged.
- b. Explain why approximating the working set by counting references over a specific window Δ , can sometimes be less accurate than you'd like.
- c. Explain why a purely local replacement strategy is unlikely to be useful in implementing a working set-based replacement policy.

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

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
 - total memory needed for the objects
 - number of TLB misses, assuming no other memory references affect the TLB, and no page translations in the TLB to start with
- 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.

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.
- 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.
- 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
 - ii. select the low 10 bits of the resulting number
 - iii. select the high 10 bits of the number from part (i), and shift them right, so they now occupy the low 10 bits
 - iv. discard the bits representing the page displacement in a physical address, and set the valid bit
 - 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`
- 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).