**ECE254: Operating Systems and Systems Programming** Spring 2015

# Lecture 24 — Virtual Memory II

*Jeff Zarnett*

## Virtual Memory, Continued

We will continue with our discussion of virtual memory by looking at a few advanced considerations in virtual memory.

### Allocation of Frames

In a simple system where we do not do anything advanced, if there are $n$ frames free in the system, we will demand page all of them. So the initial state is that all frames are empty, and as needed, pages are read into those frames. Once all $n$ frames are filled with pages, page $n + 1$ must replace a page already in a frame (because there is no more space). When a process terminates, all its frames are marked as free. In theory, one process could fill all the frames in the system. This is as simple as it can be; from there we can build on it.

We might reserve a few pages to be free at all times for performance considerations. When we want to move a page into a frame, if all of the frames are full, we select a victim and write that victim out to disk if necessary. If we keep a few frames free, the newly-read page can be read into one of the free frames, and we can write the old page out to disk at a convenient time. The read does not need to wait for the write (flush of the old page) and therefore the user process gets to continue a bit sooner than it otherwise would.

Assuming, as we did with cache, that we might want to allocate different numbers of frames to different processes (and not necessarily let one run wild), we are constrained in the number of pages we can allocate. The maximum number of frames a process could have is the maximum number of frames in the system (obviously), but the minimum is more interesting.

The motivation behind allocating at least a minimum number of frames is ostensibly performance. As long as our page replacement algorithm is sane (i.e., not FIFO), adding more frames reduces the page fault rate. As we demonstrated earlier, a page fault is a huge performance decrease.

The minimum number of frames is determined by the architecture of the system. Imagine a machine where a memory reference instruction may contain one memory address. In the worst case, the instruction and the address are in different pages, so we will need two frames to be able to complete this instruction. If the max frames for this process were 1, the instruction could never be completed. The IBM 370 `MVC` instruction is an extreme example: it moves data from one storage location to another. It takes six bytes, and the instruction itself can straddle two pages. That requires six frames. The worst case scenario is when the `MVC` instruction is the operand of an `EXECUTE` instruction that also straddles a page boundary. So eight frames are needed [**?**].

In theory, the problem could be infinitely bad if the computer architecture allows referencing of an indirect address. The address being referenced could be on another page. That instruction could then reference another indirect address, and it's turtles all the way down. If that is the case, the entire virtual memory must be in main memory (so, not possible). The standard solution is to limit the levels of indirection to some manageable value, like 16. Rather like recursion leading to a stack overflow, it is possible that a stack overflow prevents a program that is correct from running, but more likely the program is in an infinite loop and it is better off terminated. If we limited the levels of indirection to 16, then the worst case scenario is a requirement of 17 pages [**?**].

Assuming we do not allocate every process the minimum or maximum, there are a few allocation algorithms we might follow. We already got a glimpse at this when we talked about local vs. global cache replacement.

**Equal Allocation.** If there are $m$ frames in the system and the operating system reserves $k$ of them for its own use, there are $m - k$ frames available for processes. If there are $n$ processes in the system, if we allocate them equally, each process gets $(m - k)/n$ frames. If this division produces a remainder, the leftover frames can be kept as a pool of free frames for performance purposes as above. There is an obvious flaw in this plan: why does a text editing program get the same amount of frames as a web browser and the same amount of frames as a game (which tends to be VERY demanding on memory).

**Proportional Allocation.** So how about proportional allocation: each process should get a share of the frames based on its needs. The strategy suggested in [**?**] to do frame allocation runs something like this. Let the virtual memory size of a process $p_i$ be defined as $s_i$. Thus $S = \Sigma s_i$. If the total number of frames in the system is, once again, $m$, and the operating system reserves $k$ for itself, then we allocate $a_i$ frames to a process $p_i$ according to the following formula: $a_i = s_i/S \times (m - k)$.

This value of $a_i$ is only an estimate. It may not divide evenly, and we can only allocate an integer number of frames to each process. So we will have to raise or lower $a_i$ a bit to make it an integer. If $a_i$ is below the minimum number of frames, then it needs to get bumped up to that minimum. We also must respect the limits of the system: the sum of all $a_i$ values may not exceed the total frames of memory (minus the OS reserve), so a few of the larger processes may need to have their allocations nudged down.

Note that with proportional allocation, as with equal allocation, there is no regard given to the priorities of the processes. Normally, we would want to give more frames to higher priority processes so their page fault rates are lower and they can therefore execute more quickly. So we could modify the $a_i$ values according to process priority, as well [**?**]. The subject of priority is something we have not yet examined much yet, but will do so soon when we get to talking about scheduling; the next major topic. But first, a return to the subject of thrashing.

## Thrashing

Thrashing received a little bit of mention in the introduction to virtual memory, but it deserves some further consideration. The quick definition of thrashing still applies: the operating system is spending so much time moving pages into and out of memory that no useful work gets done. That still applies. Aside from intentionally depriving the system of RAM (as in the NeXTStep scenario), how can we get into this state, and how can we get out of it?