Lecture 18 — Virtual Memory II

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

February 26, 2023

Allocation of Frames

In a simple system with *n* frames free in the system, demand page all of them.

Initially, all frames are empty, and as needed, pages are read into those frames.

Allocation of 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.

Make a Reservation

We might reserve a few pages to be free at all times for performance reasons

Make a Reservation

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.

How Many Frames?

We might want to allocate different numbers of frames to different processes.

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.

Minimum Frames

The motivation behind a minimum number of frames is ostensibly performance.

More frames is usually better!

But what determines the minimum?

Minimum Frames

The absolute minimum number of frames is determined by the architecture.

House owner: I'd like to break both of my legs.
Architect: Say no more.

Minimum Frames

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 complete.

Minimum Frames: IBM 370

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.

Minimum Fames: Infinite?

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.

Minimum Fames: Infinite?

The standard solution is to limit the levels of indirection to some value.

If we limited the levels of indirection to 16, then the worst case scenario is a requirement of 17 pages.

Allocation Strategies

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.

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.

Equal Allocation

If there are n processes, 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.

But: why does a text editor get the same amount of frames as a web browser?

And the same amount of frames as a game (VERY demanding on memory)?

Proportional Allocation

Each process should get a share of the frames based on its needs.

Let the virtual memory size of a process p_i be defined as s_i.

Thus $S = \Sigma s_i$.

Then we allocate a_i frames to a process p_i according to the formula:

$a_i = s_i/S \times (m - k)$.

Proportional Allocation

This value of a_i is only an estimate.

It may not divide evenly: rounding is required; minimums need to be enforced.

A few of the larger processes may need to have their allocations nudged down.

Proportional Allocation

Note that with proportional allocation, as with equal allocation, there is no regard given to the priorities of the processes.

We could modify the a_i values according to process priority.

Priority matters a lot in scheduling… soon…

Local and Global Replacement

Local and Global Replacement

We already saw the idea of local and global replacement in the cache.

If we have chosen the LRU algorithm, in a global replacement policy, the least recently used page anywhere in the cache (memory) will be replaced.

If we chose a local page replacement policy, it is the least recently used page that belongs to the current process.

Local and Global Replacement

Local is not affected much by our allocation concerns; leave that alone.

Suppose we choose global allocation.

We will see the number of pages allocated to each change over time.

Over time, a global replacement algorithm means processes that run more often will accumulate more pages in memory.

They will acquire more pages than get taken from them.

Watching Frame Counts

It might make sense to watch to see how many frames each process has.

Goal: prevent a process from falling below the minimum number of pages.

Or we swap completely to disk a process that no longer meets that threshold.

Thrashing

This topic was previously introduced, and now we'll come back to it.

The quick definition of thrashing still applies.

Aside from intentionally depriving the system of RAM, how can we get into this state, and how can we get out of it?

Thrashing

In simple operating systems, the logic that controlled how many processes to run at a time would rely just on the CPU utilization.

If CPU utilization is low, the CPU needs more work to do!

Thrashing

Assign it more work by starting or bringing more processes into memory.

The global page replacement policy is used here, so when a process gets a page fault, it takes a frame from another process.

Under most circumstances, this works just fine.

Downward Spiral

This situation can run until one process starts to have a lot of page faults.

This is not unreasonable; a compiler might be finished with reading and parsing the input files and moving to generation of binary code.

This requires a whole bunch of new instructions pages, plus pages for output.

When this process does so, it starts taking pages from other processes.

Downward Spiral

The victim processes need the pages they had, so when they get a turn to run, they too start generating page faults.

So more and more requests are queued up for memory writes and reads, so the CPU is not very busy.

Here's the fatal mistake: seeing that the CPU is not very busy, the OS schedules **more** programs to run.

The Doom in Thrashing

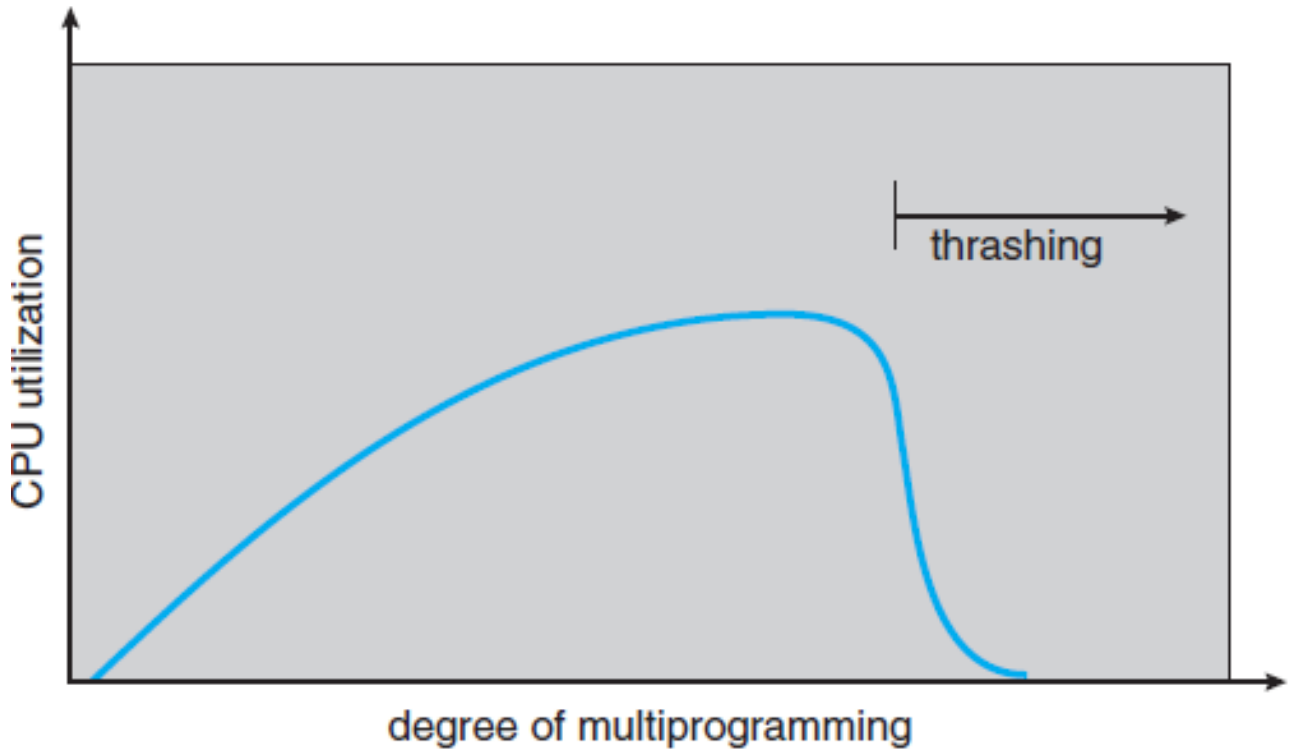
A new process getting started will need at least the minimum number of pages.

These have to come from somewhere, so they will necessarily come from the pages currently belonging to other processes.

This causes more page faults, more time spent paging, lower CPU utilization…
Prompting the OS to start more processes.

Whiplash!

No more work is getting done, because the system spends all its time moving pages into and out of memory, thrashing all around, acting like a maniac.



Breaking the Spiral

To increase CPU utilization we need to stop the thrashing…
which means we need **fewer** programs in memory at a time.

CPU usage alone is not a sufficient indicator of whether more or fewer processes need to be running right now.

It also matters *why* the CPU utilization is low. Another reason that might cause low CPU usage is, as you may recall, deadlock.

Local Replacement Instead

What if we stop using a global replacement policy and instead use local?

This limits some of the damage;

- If P_1 is thrashing, it cannot steal pages from P_2.
- We do not get a cascade.

But P_2 is still affected.

If P_1 is spending all its time paging to and from disk, any other process that wants to use the disk is going to have to spend more time waiting for the disk.

Starting and Suspending Processes

Decide whether to start or suspend programs not just based on CPU usage, but also on the number of page faults that occur in a given period of time.

If there are too many page faults, it indicates we have too much going on.

That is a reactive solution, however.

It would be better to be proactive and deal with this before thrashing has started.

More Clairvoyance

As is typical, a bit of clairvoyance would help here: how many pages is the process going to need and when will it need them?

Unfortunately, there is no good way to know or to figure this out. So we will have to rely on (educated) guessing.

Making Our Best Guess

What do we know about process memory accesses?

They tend to obey the principle of locality, both temporal and spatial.

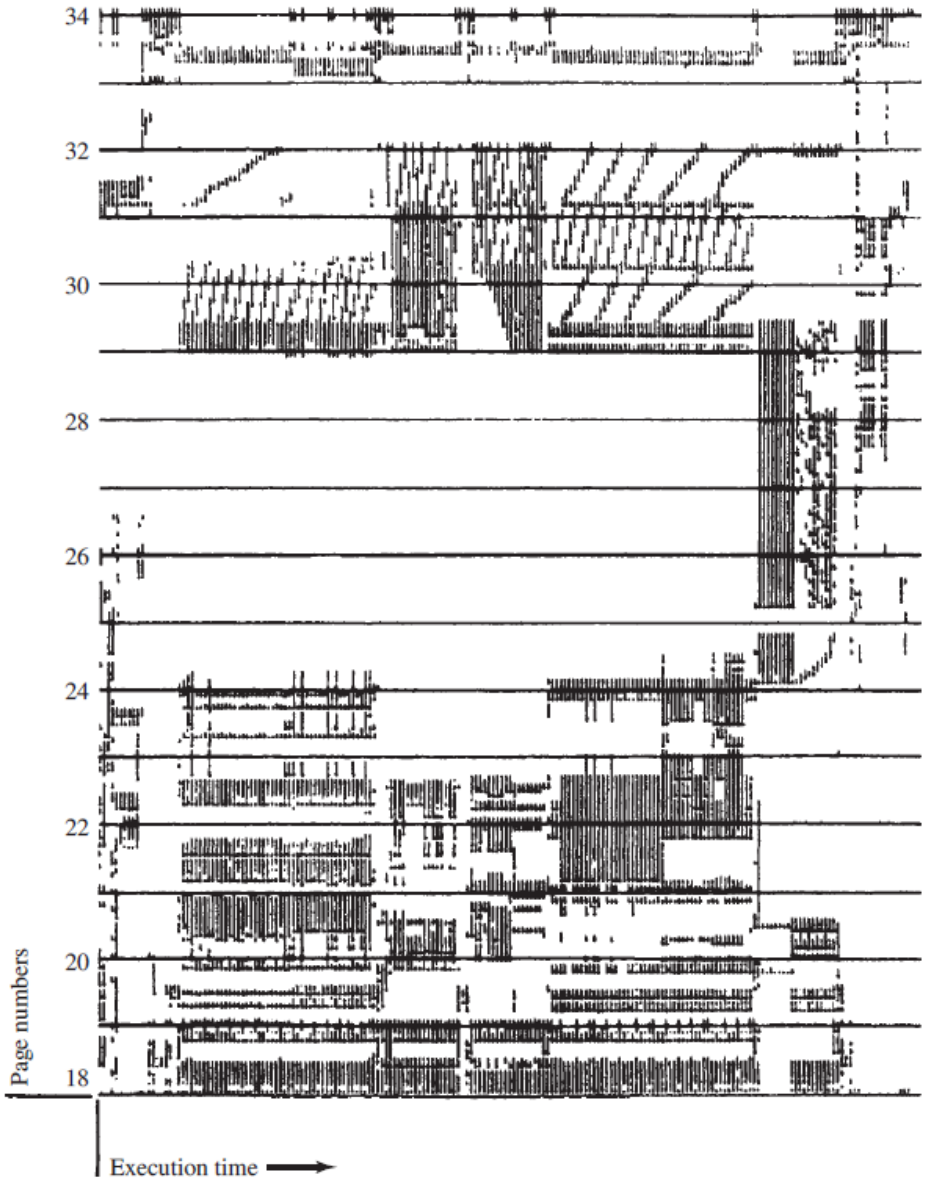
We could think of different parts of the program as different localities, as if they were areas on a map. And localities may overlap sometimes.

Making Our Best Guess

Give a process enough frames for its locality.

It can operate in this little area without encountering (too many) page faults.

Check Assumptions



The Working-Set Model

A potential solution is the working-set model.

Retain the last *n* pages in memory as they represent the locale of the program.

Assuming that most memory accesses are local, the most recent *n* pages will be the most frequently used.

The Working-Set Model

In the textbook descriptions, this n is usually called Δ, the working set window.

Pages that have been recently used are in the working set.

If a page has not been accessed recently, it will drop out of the working set after Δ time units since its last reference.

Working Set Example

Suppose the window is defined as being ten accesses.

Any page that was accessed in the last ten requests will then be considered part of the working set.

If the next ten memory accesses are all in page *k*, then after those further ten accesses, the working set will contain only *k*.

The size of the working set will change over time; can be from 1 to Δ pages.

Choosing Delta

If Δ is too small, the working set will not encompass the entire locale.

If it is too large, it will cover multiple locales.

Underestimating the size of the locale is bad; it will mean more page faults.

Overestimating is also bad; it means fewer processes will be allowed to run.

Sum of the Working Sets

If the working set of every process is summed up, we will get the total number of frames each process would “like” to have.

If this sum exceeds the $(m - k)$ available frames in the system, at least one process is going to be unhappy.

It does not have as many frames as it will need.

And like unhappy workers who go on strike, unhappy process start thrashing.

Working Sets

Once a value of Δ is determined, the OS will monitor the working set of each process and use that to figure out if the system is currently overloaded.

If it is, the OS will pick a victim and suspend it to prevent thrashing.

If the system is underloaded (frame supply exceeds demand), more processes can be started to run.

Working Set

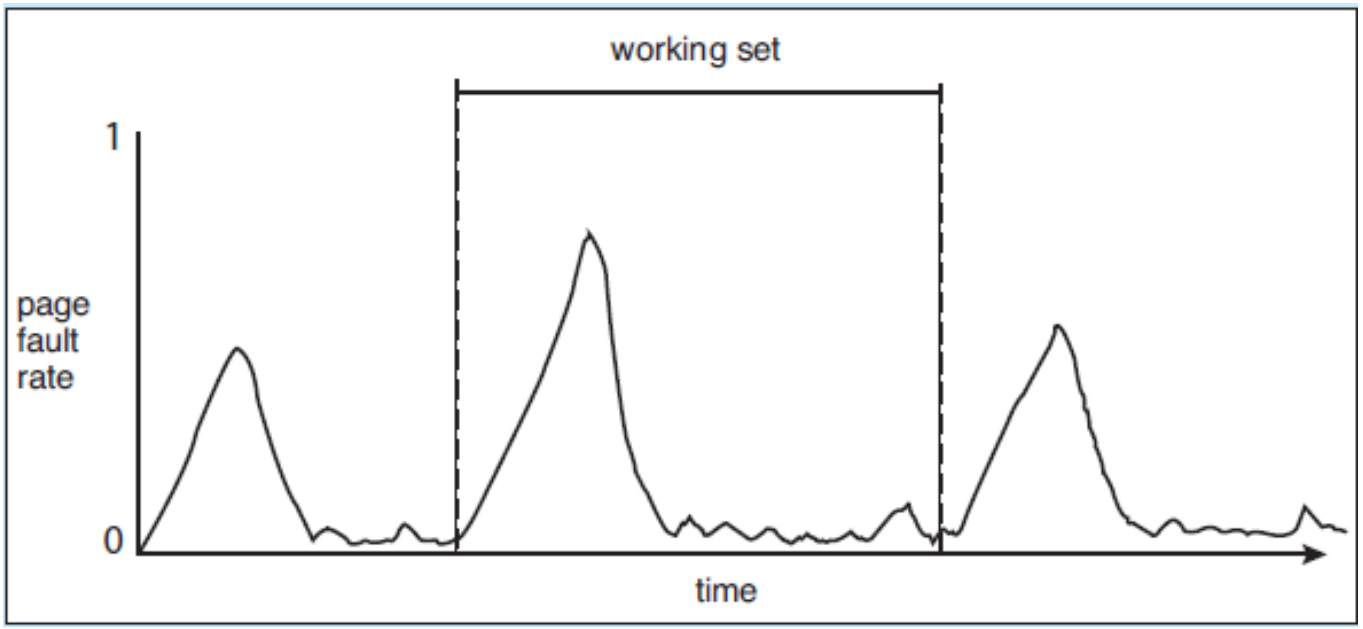
The page fault rate of a process tends to vary over time.

At the beginning, there will be a bunch of page faults as the program starts up.

Then once established in its first locale, the rate will drop.

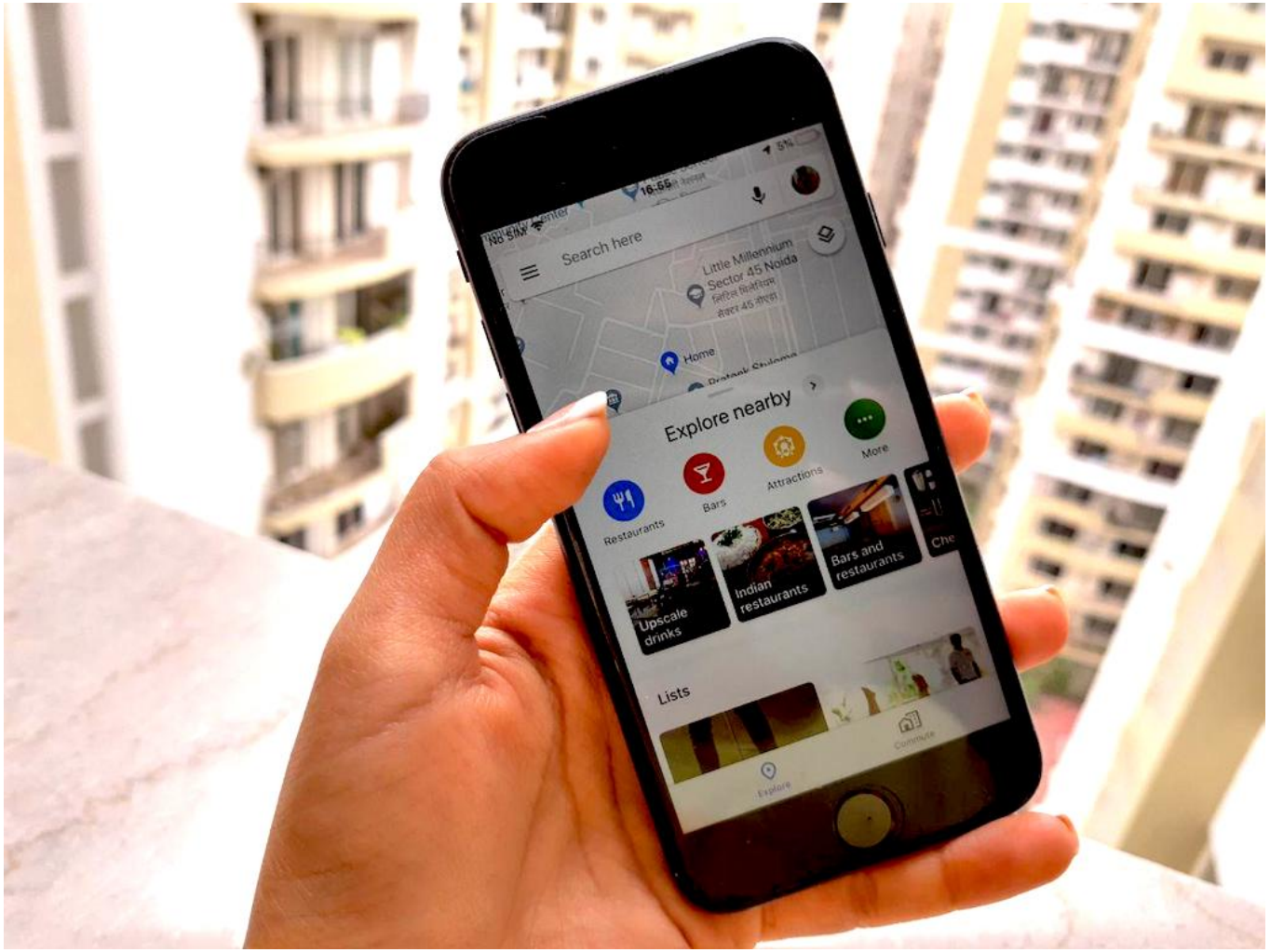
When it come time to move to a new locale, the page fault rate rises until the program is “settled” in that new locale.

Working Set



Working Set Analogy

Imagine that you have moved to a new city for a co-op term.



Working Set Analogy

When you have just moved, you will frequently rely on Google Maps to find what you are looking for and how to get there.

You need to buy groceries, so you ask Google for directions to the nearest grocery store of choice.

Once you've been there, you know the way so you don't have to ask again.

Working Set Analogy

Not knowing where the grocery store is equals a page fault, and asking Google is like asking the operating system to bring in that page from disk.

Once you know the way to the grocery store, it is part of your working set and you do not have to ask again.

… Until your next co-op term when you move somewhere else.