Lecture Overview

Primary Topic: Continuation of paging replacement algorithms. By topic:

1.) What is thrashing?
2.) Solution to thrashing:
    - Working sets
    - Page Fault Frequency
3.) Variable Page size
4.) Paging the operating system itself
5.) Problem: I/O accesses with virtual memory systems
6.) How do we study paging algorithms?
7.) Algorithm Performance

Lecture Content

Thrashing: A situation when the page fault rate in the system is so high that is spends most of its time processing page faults or waiting for a page to arrive. In other words, there is **too much page idle time. Conditions when this happen include:**

1.) Overly aggressive multiprogramming! Namely, there are too many tasks to switch between to make any real headway in any one task. Think of a student who is taking too many classes. For operating systems: consider a many process system, 40 page frames in physical memory, and each process demanding 50 pages.
2.) The system doesn't know when it has taken on more work than it can handle. Like in the above example, if any process needs more memory than there is space in physical memory, the OS is forced to thrash.

Some solutions are to:

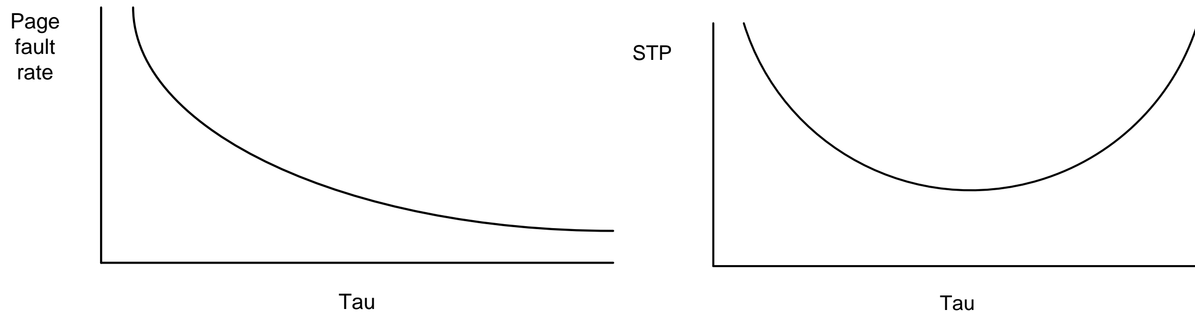
1.) Reduce the load on the OS (run less processes).
2.) Get more memory! Ultimately, thrashing is caused by a lack of physical memory. With infinite physical memory, there is less contention for resources between processes and even for a single process.
3.) Change the paging algorithm. If you are paging in and out the same page over and over again rather than using LRU, etc, then you are thrashing when you need not be!
4.) Working set (see below)

Working Set: Is a solution to thrashing proposed by Peter Denning. **Definitions:**

1.) A working set is the set of pages that a process is working with and which must be resident in order for the process to avoid thrashing.
2.) Summarized: The memory required for a process to work efficiently.

3.) Precise: "*Exactly that set of pages used in the preceding t (tau) time units*"

Tau is given in **virtual time units** which are the **number of memory accesses** instead of an absolute time (on any scale). At any given point in time, **all pages which have been accessed in the last tau "time units" comprise the processes' working set.** Typically, tau will be very large. But what if it is too large or too small? See below (STP = space time product):



The **working set paging algorithm** keeps in memory exactly those pages used in the preceding tau time units (typical values: 10,000-100,000). This has two implications to processes and memory:

1.) A process won't be run until its working set is in memory.
2.) Pages outside of the working set are up to be discarded.
3.) **There needs to be an effective way to keep track of what the working set actually is** (it can be thought of as a moving window). This is **working set paging.**

Working set paging is designed to eliminate thrashing through a clever paging scheme. It requires that **the sum of the working sets (**of all the jobs which can run – also known as the **balance set** or the jobs in the in-memory queue**) be able to fit into memory**. What if it didn't? Then the jobs eligible to run would need to thrash (or at least retrieve pages from memory) to get the pages that they needed to run. **Some issues are:**

1. There must be an algorithm present which updates what jobs are in the balance set. What happens when the balance set changes too frequently?
    1. You need to page in new memory to satisfy the new balance set.
    2. And you get thrashing as a result!
2. As the working set of each job changes, the balance set must also change.

These issues amount to a lot of **overhead**. Is it worth it? Keep in mind that Working set has an advantage over LRU in that it adapts based on a processes needs. LRU assumes static need.

Given the complexity, can we implement working set? Aside from what was mentioned already, some issues are that:

1. In order to be efficient, you need to know how much memory is needed in order to keep the CPU busy. If you don't have enough jobs on the queue, and just in general, implementing all of the overhead associated with working set is going to lead to the CPU being idle.

2. How do you compute tau if pages are shared? Do you add more bookkeeping information for the last-used time by each process?

Regardless of the issues, several attempts have been made at implementation. They are:

- **Idealized Solution (not implemented):** Capacitor / page: when capacitor discharges → page is taken out of the working set. When the page is accessed → the capacitor is charged. This is a true LRU scheme. Why won't it work?
    1. Tau changes based on **virtual time** not real time. (waiting processes need capacitors whose charge don't leak)
    2. What happens when there are shared pages for multiple processes? (sharing the "capacitor clocks" and making sure that they sometimes decrement and sometimes static make this scheme very complicated to implement physically.
- **Realistic Solution:** Use the reference/use bits associated with each page. At the end of tau time → check to see which bits have been flipped on. Why won't this work in a modern system?
    1. There are 1000s of pages associated with each process. This amounts to 100,000s of memory references made in a serial loop! What are the implications of this? WORKING SET IS NOT IMPLEMENTED THESE DAYS!

Back to issues with working set, we were discussing **overhead.** One strategy to minimize overhead is **Working Set Restoration.** The idea is that when a process is removed from the in-memory queue, its working set won't change (tau won't be changing as tau is calculated using virtual time). As such, we can **bring the working set back into memory along with the process immediately.** The advantages:

1. Reduce CPU overhead: increase performance of multilevel storage hierarchies.
2. A lot of overhead up front when process is started instead of waiting for each page fault to occur while the process is free running.
3. Because all of the page fetching happens at once, it can be organized based on size, latency, etc (instead of just done on the fly as faults occur).

Page Fault Frequency: How do you cut down on it? One solution is the **page fault frequency** algorithm (by Opderbeck and Chu):

On page fault, let X = virtual time (memory accesses) since the last page fault.

If X > tau:

a. Remove all pages for the process that have U (use) bit = off.
b. Get a page frame for the new page.
c. Turn off all R (reference) bits for the process.

This strategy aims to bring lots of pages into memory for a process that is page faulting too often, the assumption being that when it has enough pages in memory, it will stop page faulting as often. **One problem is large applications with scattered data accesses:** they will still fault a lot and not necessarily need more pages.

Page Size: Another consideration in paging is page size. Who chooses page size? Typically the hardware, but the OS has some flexibility. For example, the OS can string together smaller pages to form larger pages if it needs to simulate larger pages. What are the advantages of large pages?

1. Every fault requires the actual fetch time + memory access overhead. With larger pages, more data is transfer per unit of memory access overhead.
2. TLB reach covers more of the processes address space. The TLB is a fixed size where each entry maps to a page. The larger each page, the greater the reach of the TLB. Decreases TLB miss rate.
3. Smaller page tables. With fewer data pages covering the virtual address space, you need less page table entries and therefore smaller page tables.
4. Less time spent dealing with page replacement algorithms. There are less pages to look at!

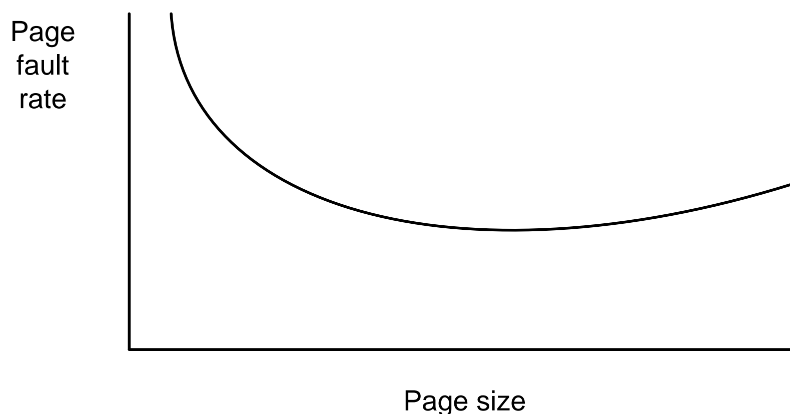
What are some of the disadvantages?

1. You need more total memory to cover the same active localities (data access wise) in a program).
2. Even though there is more data transferred per unit of memory access overhead, there **is** more data to transfer! From a strict latency standpoint, the page faults with larger pages take longer.

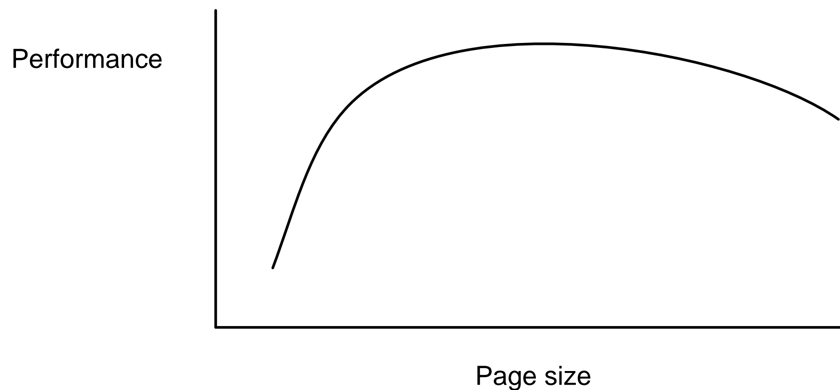
What are the typical page sizes these days?

1. **Most common:** 4KB (4096 bytes)
2. **Probably ideal these days:** 16KB
3. **Too small (still used sometimes but too small for modern apps):** 512 bytes

Graphically, performance and page fault rates are as follows:



Notice that there is an optimum in this graph (**the bottom of the dip**).



Paging the Operating System: This is possible, however with restrictions. You **cannot** page out pieces of the OS that are needed on-demand or are used to do demand paging. Why? If I/O code needs to be accessed **immediately**, it can't wait for a page fault (similar situation applies to trap handler code)! Also, how would we get demand paging code back into the system if we paged it out? Code that cannot be paged out is called "**wired down.**"

I/O with Virtual Memory: I/O naturally only deals with physical addresses. Thus, the OS must have a means of translating virtual buffer address → physical buffer address (and these utilities should **never** be paged out!). Why? I/O is on-demand (see "wired down" above). OS can implement "wired down" pages through **Lock** bits. The lock bit is just like the U/R/V/etc bits – it means that a page cannot be swapped out. Other issues:

1. I/O transfers that cross page boundaries: what happens if the pages are not contiguous? Some solutions are to:
    a. Break the transfer into several, such that each smaller transfer is contiguous.
    b. Make the transfer map to non-contiguous memory! (only very smart OSs can pull this off).
    c. Ignore virtual memory and place the I/O subsystem memory in a contiguous chunk of real memory, which is managed by the OS.

How do we study paging algorithms: Different techniques have been tried, and are as follows:

1. **Math model.** As in economics, it would be great if this actually worked. OSs are way to complex, however, so no such model yet conceived is an acceptable one.
2. **Random number driven simulation.** Same problem as the math model.
3. **Experiments on a real system.** This approach makes the most sense, however, requires actual machine time and makes results very difficult to reproduce typically.
4. **Trace driven simulation.** Given the shortcomings of 1-3, **this is what people actually do!** What happens is you get a **program address trace** (a sequence of virtual addresses that a

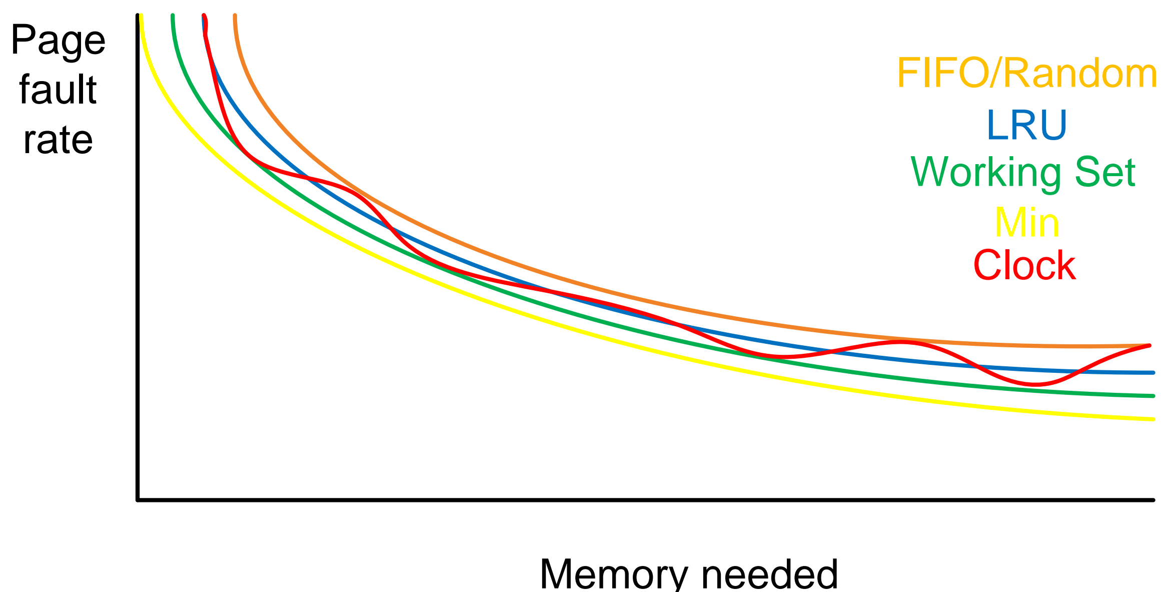
process has requested/used/accessed/etc) and use that trace to simulate different algorithms.

** **Important:** the program/virtual address trace is not dependant on the paging algorithm! The paging algorithm is used to improve efficiency in paging data in and out. It should **never** constitute the correctness of a program.

We can get the **program address trace** through hacking almost any layer of abstraction in the machine. See here for examples:

1. **Machine interpreter.** Simple idea: trace is generated as the machine executes.
   a. **"execute" instruction (IBM 370).** An instruction which tells you to execute another instruction! This makes trace generation very simple.
2. **Hardware monitor.** This is a logic analyzer! This is impossible these days ~ everything is embedded.
3. **Trace trap facility.** Trap on every instruction. The trap handler is responsible for recording bookkeeping information for the trace.
4. **Microcode modifications.** Similar to the above: the hacked microcode performs the bookkeeping.
5. **Instrument .o/.s code to write trace records.** The modified code is responsible for the bookkeeping (typically keeps track on every instruction or just on load/store/branches).
6. **Generate page fault on every reference.** Trace is generated when the faults are being handled. How is this typically implemented? **Turn off all of the valid bits!**

Algorithm Performance: Graphically speaking:



In general, you are **hosed** if you have significantly more data than physical memory! No matter what

algorithm you use, you will need to page in and out to run large applications. Some areas within your control, however, are:

1. Writing more efficient code / refactoring algorithms: Consider **matrix transpose.** Through naïve approaches and large matrices, this operation can lead to a page fault / operation. Solutions:
    a. Store the matrix as sub-matrices which actually fit into memory.
    b. Represent the matrix differently (row major order for example) so that sequential accesses are walking through a single page (this works better with matrix multiplication – transpose doesn't benefit much from it.