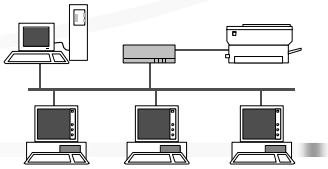


Operating Systems
600.418
Virtual Memory

Department of Computer Science
The Johns Hopkins University

Virtual Memory

Lecture 6

Reading:
- Silberschatz & Galvin
- chapter 9

Additional Reading:
- Stallings
- chapter 8

Observations

- In many cases, not all of the logical address space of a process is accessed during the life of a process:
  - **80-20 rule: only, say, 20% is used code while the rest is error handling etc, which is rarely used.**
  - **Arrays, lists, tables - many times are allocated to be able to contain the maximal need, which is rarely needed.**
- And, of course, not all of the logical memory space is needed at the same time.

Virtual Memory

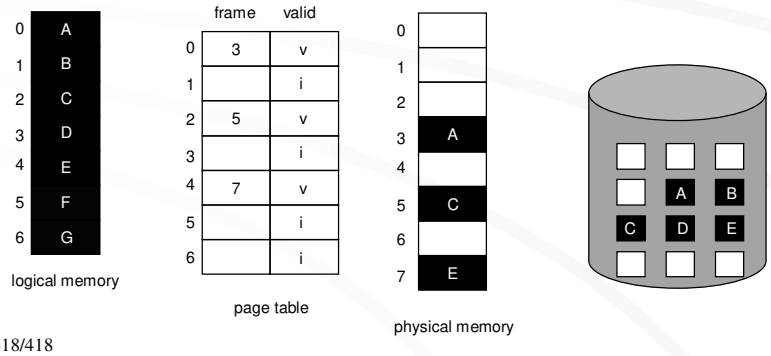
- Virtual memory is a scheme that allows a process to execute while it is only partially in memory:
  - **A process is no longer constrained by the amount of physical memory available.**
  - **More programs could be run at the same time, increasing the utilization and throughput of the machine.**
  - **Less I/O is required to load each program and start running it, so each program would start to run faster.**

Fetch Policy

- Determines when a page should be brought into main memory. Two common policies:
  - **Demand paging only brings pages into main memory when a reference is made to a location on the page (paging on demand only).**
    - many page faults when a process is started but should decrease as more pages are brought in.
  - **Prepaging brings in more pages than needed.**
    - It is more efficient to bring in pages that reside contiguously on the disk.
    - efficiency not definitely established: the extra pages brought in are "often" not referenced.

Demand Paging

- Initially, valid bit is set to 0 for all page table entries.
- During address translation, if valid bit in page table entry is 0, a page fault trap occurs.



Page Fault

- Trap to the operating system (save registers and state).
- Operating system looks at the process PCB to decide whether:
  - **Not in logical address space ⇒ abort the process.**
  - **Page is not in memory ⇒ continue:**
- Allocate a free frame in memory. ***
- Schedule a disk read of the page to the free frame.
- Allocate the CPU to some other process.

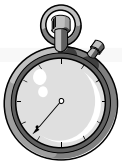
  ≈
- Interrupt from the disk (I/O completed).
- Save registers and state for currently running process.
- Update original process page table (the desired page is in memory).
- Move original process to ready queue - will wait for CPU allocation.

  ≈
- Restore registers and state then **restart** the trapped instruction.

Architecture Support

- Page table.
- High speed secondary memory.
- Ability to trap on an invalid bit.
- Ability to restart an instruction:
  - **Several page faults may be needed for one instruction. (why?)**
  - **Atomic operation guarantee.**
    Examples: PDP-11
    MOV (r2)+, -(r3) IBM 370
    Move block (up to 256 bytes, possibly overlapping).

Performance of Demand Paging



- Page fault rate $0 \le p \le 1$
  - **if p = 0, no page faults**
  - **if p = 1, every reference results in a page fault.**
- Effective Access Time:

  EAT =

  (1 - p) effective_memory_access +

  p (page fault overhead + [swap out] + swap in + restart overhead)

Restart overhead includes access to the data (which is now in memory) via the page table. Note that we pay the TLB penalty in this case but the TLB will definitely fail.

Performance Example

Suppose:

- Memory access time is 100 nano sec.
- Effective_memory_access is 107 nano sec (with TLB when TLB access is 5 nano sec).
- Page fault overhead is 100 micro sec.
- Page swap time is 10 millisecond.
- 50% of the time, the page that is being replaced has been modified and therefore needs to be swapped out.
- Restart overhead is 20 micro sec.
- p - page fault rate.

Effective Access Time = 107* (1-p) +(100,000+ 10,000,000+0.5*10,000,000+20,000+5+200)*p

EAT = 107 *(1-p) + 15,120,205*p.

Performance Example (cont).

So: What is a good page fault rate?

Effective Access Time = 107* (1-*p)* +(100,000+ 10,000,000+0.5*10,000,000+20,000+205)**p*

EAT = 107 *(1-*p*) + 15,120,205**p*.

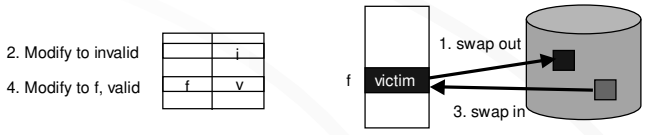
Suppose *p* is 1% $\Rightarrow$ EAT = 151,309

Talking about a slowdown of 1414!!!

- For the luxury of Virtual Memory to cost only 10% overhead, we need *p* to be around 0.00000066 !!!!.
- This means one page fault for every 1,500,000 memory accesses !!!

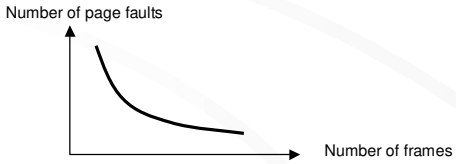
Page Replacement

- What happens if there is no free frame to allocate?
- Here comes page replacement - find some page in memory that is "not that used" and swap it out. A free frame was just created.
- Note that the same page may be brought into memory several times over the life of a process.
- Use a dirty bit to reduce overhead of page transfers - only modified pages need to be written to disk, unmodified pages can be discarded.
- Page replacement completes the separation between logical memory and physical memory. Large virtual memory can be provided on a smaller physical memory.



Page Replacement Algorithms

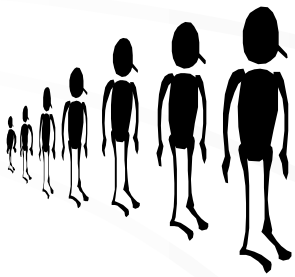
- Goal
  - **Lowest page fault rate.**
- Evaluation
  - **Usually by running it on reference stings and counting the number of page faults.**
- Adding more memory
  - **Should hopefully lead to lower page fault rate.**



First In First Out (FIFO)

- The victim is the oldest page in memory.
- Assume 4 frames and the following reference string:

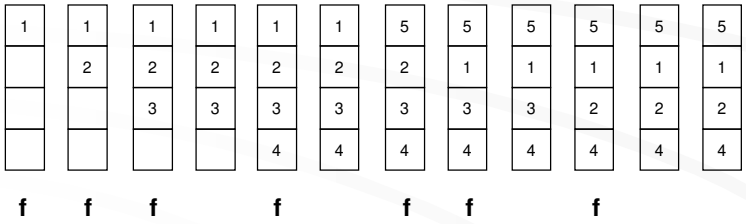
  1 2 3 2 4 3 5 1 5 2 4 5



First In First Out (FIFO)

- **Replace the oldest page in memory.**
- Assume 4 frames and the following reference string:

1 2 3 2 4 3 5 1 5 2 4 5

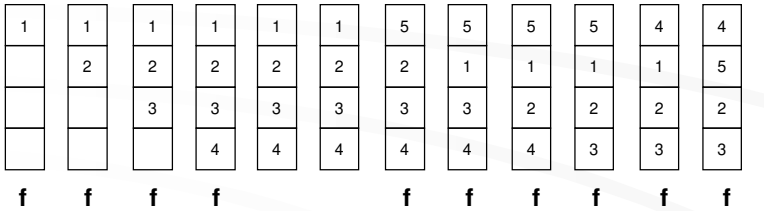


Easy to understand and program.
Will adding memory always improve things?

First In First Out (FIFO)

- FIFO suffers from the following anomaly which is called Belady's anomaly.

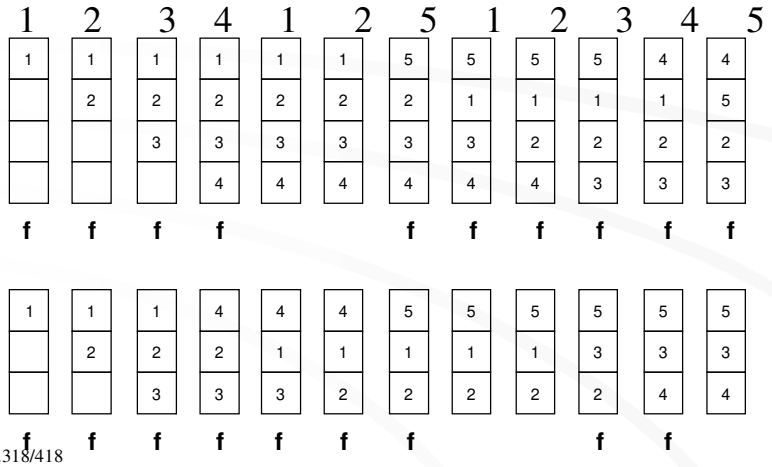
1 2 3 4 1 2 5 1 2 3 4 5



Try it with 3 frames...

First In First Out (FIFO)

- FIFO suffers from the following anomaly which is called Belady's anomaly.

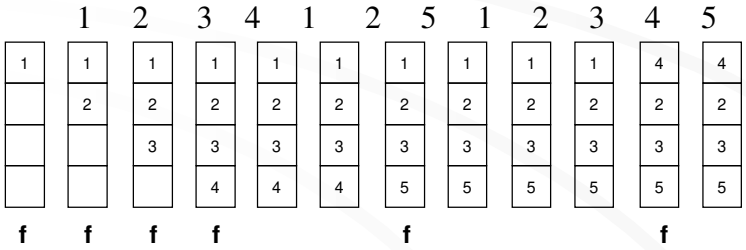


The Optimal Algorithm (OPT)

- Assumes knowing the future.
- Can you think of it?
- **Replace the page that will not be used for the longest period of time.**

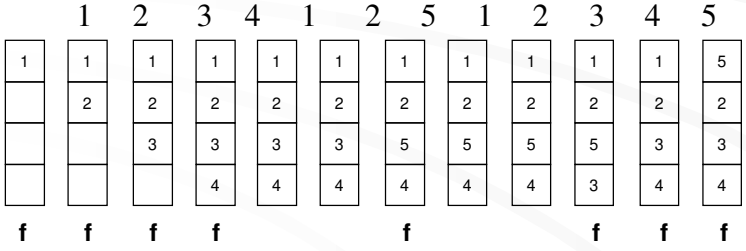
The Optimal Algorithm (OPT)

- Assumes knowing the future.
- Can you think of it?
- **Replace the page that will not be used for the longest period of time.**



Least Recently Used (LRU)

- Well, OPT is not possible...
- **Replace the page that has not been used for the longest period of time.**



LRU Implementations

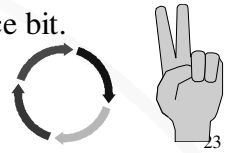
- Counter implementation:
  - **Every page entry has a counter.**
  - **Every time a page is referenced, increment a global counter and store it in the page counter.**
  - **For page replacement, select the page with the lowest counter (search for it).**
- Stack implementation:
  - **Maintain a stack of page number in a double link list.**
  - **Move a referenced page to the top (locate and change pointers).**
  - **For page replacement, take the bottom page.**
- In any case:
  - **LRU is hard to implement as is. It requires too much work per page access or too much hardware support.**

Stack Algorithms

- A class of page replacement algorithms for which the set of pages in memory for *n* frames is always a subset of the set of pages that would be in memory with *n*+1 frames.
- (Belady anomaly) **if and only if** (not a stack algorithm). WHY ?
- What about OPT ? LRU ?
- Both OPT and LRU are stack algorithms.

LRU Apprx.: Second-Chance Algorithm

- Second-Chance algorithm is actually a FIFO replacement algorithm with a small modification that causes it to approximate LRU.
- When a page is selected according to a FIFO order, we check its reference bit. If it is set (the page was referenced), we clear it and look for another page.
- **Also called the clock algorithm.**
- In the worst case, when all bits are set, the pointer cycles through all of the pages, giving each page a second chance and clearing its bit.
- Requires hardware support for reference bit.



Enhanced Second-Chance

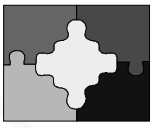
- Considers both reference bit and modify bit.
- We have the following four possible cases:
  - **(0, 0)** - neither recently used nor modified - best page to replace.
  - **(0, 1)** - not recently used but modified - not as good because we need to swap out a page, but still better than used pages.
  - **(1, 0)** - recently used but unmodified.
  - **(1, 1)** - recently used and modified - the worst page to replace.
- Requires hardware support for reference and modify bits.
- May require **two** cycles.

Page Buffering Algorithm

- As an add-on to any previous algorithm.
- A pool of free frames is maintained.
- When a page fault occurs, the desired page is read into a free frame from the pool. The victim frame is later swapped out if necessary and put into the free frames pool.
- Advantage / disadvantage ?
- **Plus - Process is put back to ready queue faster.**
- Minus - less pages are in use overall.
- VAX/VMS version - basic FIFO replacement with a free frame pool. A victim is put into the pool but the original virtual address is kept. When a page fault occurs, we first look in the pool. If we find the page there - no need for disk operation.

Frame Allocation

- Minimum number of frames in memory per process
  - **Architecture dependent: All of the pages needed for any possible instruction need to be in memory for this instruction to be executed.**
  - **Example: the IBM 370 move block operation will need up to 6 pages in memory at the same time in order to execute (two for the instruction, two for the source and two for the destination).**
- Equal allocation
  - **Each process gets the same number of frames. Divide the frames equally between the processes.**
- Proportional allocation
  - **Allocate frames according to size of process.**
- Priority allocation
  - **Higher priority processes get more frames.**



Global versus Local Frame Allocation

- Global replacement
  - **Allows a process to select a victim frame from the set of all frames.**
- Local replacement
  - **A victim frame is selected from the set of frames allocated to this process.**
- Tradeoffs?
- Global replacement is more efficient overall and therefore is commonly used. Also easier to implement.
- Local replacement prevents external influences on the page fault rate of this process as long as the same number of frames is allocated.

I/O Interlock

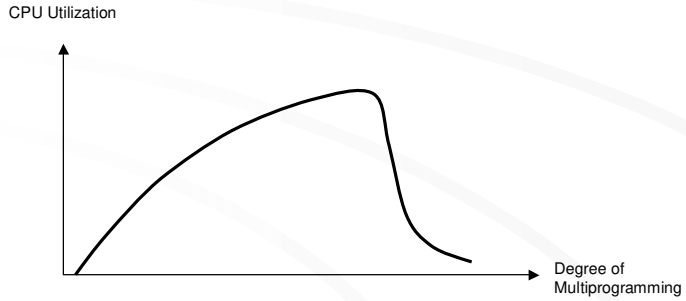
- Sometimes it is necessary to lock pages in memory so that they are not paged out.
- **Examples**:
  - I/O operation - the frame into which the I/O device was scheduled to write should not be replaced.
  - New page that was just brought - looks like the best candidate to be replaced because it was not accessed yet, nor was it modified.

Thrashing

- Why does paging work in practice ?
- Because of locality of reference in practical programs.
- If a process does not have "enough" pages to capture this locality of reference (varies from program to program and over time), the resulting page fault rate will be very high.
- A high page fault rate will lead to:
  - Low CPU Utilization.
  - The operating system will think that it needs to increase the degree of multiprogramming (bring back swapped processes into memory)
  - This will make the problem even worse!!!

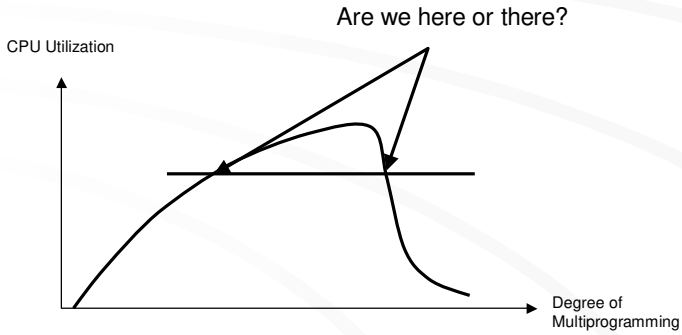
Thrashing (cont.)

- Thrashing - Operating System is busy swapping pages in and out:
  - Not enough memory to capture locality for all processes: **$\sum$size of locality for in-memory processes > total memory size.**



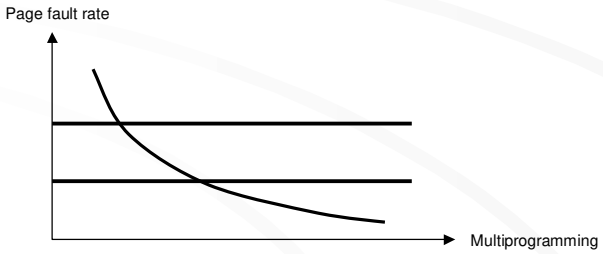
Thrashing (cont.)

- Thrashing - Operating System is busy swapping pages in and out:
  - Not enough memory to capture locality for all processes: **∑size of locality for in-memory processes > total memory size.**



Page Fault Frequency Scheme

- Establish acceptable page-fault rates.
  - **Upper watermark.**
  - **Lower watermark.**
- If Actual rate lower than lower watermark, multiprogramming can be increased (swap in additional process).
- If Actual rate higher than upper watermark, multiprogramming should be decreased (swap out additional process).



Page Size Considerations

- How to select a page size?
- Tradeoffs:
  - **Size of the page table - to keep it relatively small, we need bigger pages).**
  - **Memory utilization - better with smaller pages because of less internal fragmentation.**
  - **I/O time - latency / throughput characteristics call for bigger pages.**
  - **Capturing locality - better with smaller pages because of better resolution.**
- Historical trend is for bigger page sizes.
  - **Memory is cheap so we care less about utilization.**
  - **Disk access advances in slower pace compared with memory/CPU so I/O factor is most important.**

Program Structure

- Careful selection of data and programming structures can increase locality and lower page fault rate.
- **Example:**
  - Assume that each page is 4K bytes and we have about 64Mbytes of memory on the machine.
  - Program 1

```
A: Array[1..10000, 1..10000] of integer;
for i = 1 to 10000
    for j = 1 to 10000
            A[i,j] = i*j;
```

Program Structure

- Careful selection of data and programming structures can increase locality and lower page fault rate.
- **Example:**
  - Assume that each page is 4K bytes and we have about 64Mbytes of memory on the machine.
  - Program 1

```
A: Array[1..10000, 1..10000] of integer;
for i = 1 to 10000
   for j = 1 to 10000
         A[i,j] = i*j;
```

  - Program 2

```
A: Array[1..10000, 1..10000] of integer;
for j = 1 to 10000
   for i = 1 to 10000
         A[i,j] = i*j;
```