Lesson Summary

- address space is a bunch of contiguous pages but virtualized as a big slab
- Process Address Space can only be partially in memory
- Main Issues
  - Page Replacement Policy
  - Fram Allocation Policy
- Thrashing is bad
- Memory Mapping is useful

Policies

after covering mechanism we define the policies

- Page Replacement Policy
  How to pick victims?
- Frame Allocation Policy
  How many frames to each process
- Main Goal Minimize page faults
- Contrast with CPU through
  - CPU Scheduling
  The CPU is so fast that the decisions have to be made very quickly
  Therefore, algorithms need to be simple
  - Memory Scheduling
  The disk is so slow that it is worth spending some time to make a decision
  Avoiding a few more page faults can have a large impact on performance
  More sophisticated algorithms may be worthwhile
  As usual the OS works with imperfect/partial information (e.g., no knowledge of the future, no knowledge of what jobs will do)

Page Replacement Policies: Algorithm Evaluation

- Page Replacement Problem
  Problem Input
  - A set of page references
  - A number of available frames allocated to the process
- Problem Objective
  - minimize the number of page faults
  - This is a computational difficult problem

Optimal Algorithm

- If we have perfect knowledge of the future, we can make optimal page replacement decisions
- Not feasible in practice, but useful to have an upper bound on how well we could do in an ideal scenario
- Optimal Algorithm
  evict the page that will not come in use for the longest time

references	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame #0	7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	7
frame #1		0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	0
frame #2			1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1	1	1
page faults	X	X	X	X		X		X			X			X				X		

- We have a total of 9 page faults – this is best we can do
- Let's now look at a simple algorithm that does not assume we know the future (because we don't!)

FIFO Page Replacement

- kick out the oldest page brought to memory

references	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame #0	7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	0	7	7	7
frame #1		0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	1	0	0
framek #2			1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	2	1
page faults	X	X	X	X		X	X	X	X	X	X			X	X			X	X	X

- have 15 page faults

- The problem with FIFO is that an old page may be used all the time
  → So it is likely better to keep track of when a page was last used

Least Recently Used (LRU) Page Replacement

- evict the LRU page

reference	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
frame #0	7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	1
frame #1		0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	0
frame #2			1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	7	7	7
page faults	X	X	X	X		X		X	X	X	X			X		X		X		

- have 12 page faults
- considered a "good" algorithm

Implementing LRU Page Replacement

- Use Counters
  - Augment each page table entry with a "time of use" of use field
  - Increment a "clock" counter each time a memory access is performed
  - Update the "time of use" field with the clock value
  - When eviction is necessary search for the minimum "time of use" field: it is the victim frame - High-overhead
- Use a Stack
  - A frame is moved to the top of the stack after
  Requires a bunch of pointers shuffling
  But the victim is always at the bottom of the stack

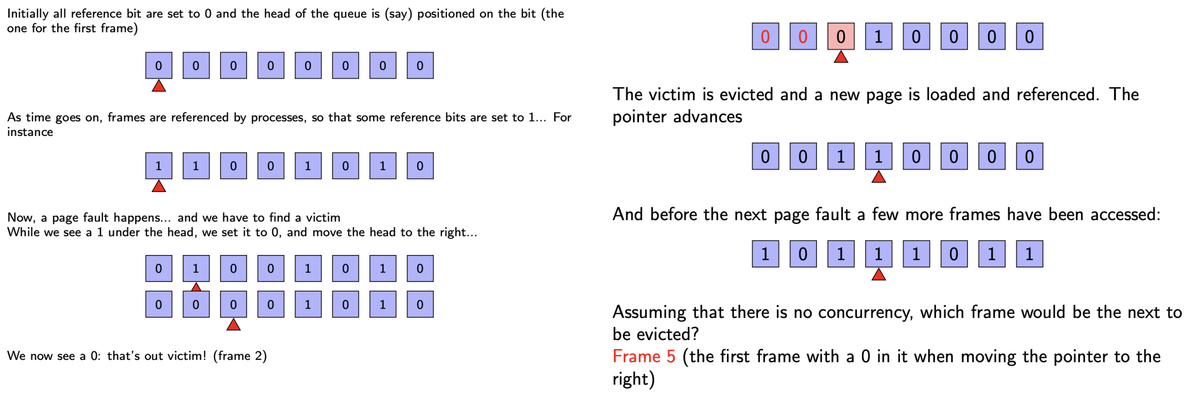
Help Form Hardware

- If the hardware does not provide any dedicated component, overhead to do anything other than FIFO is too expensive
- OSes do not implement LRU page replacement
- But the hardware usually provides a reference bit
  - Associated to each entry in each page table entry, and initially set to 0
  - Set to 1 by the hardware when the page is referenced
  - Settable to 0 by the OS
- One can do approximate LRU using the reference bit

Approximating LRU: The Clock Algorithm

- What OSes do: Clock Algorithm
  key idea: use on reference bit per frame
  Whenever a page is referenced by the program, set its entry's reference bit to 1
- When a page in a frame needs to be evicted:
  - If the reference bit is 1, set it to 0, and move the queue head to the next item in the queue
  - If the reference bit is 0, evict the page in that frame
- A page in a frame that keeps on being referenced is never evicted (its reference bit is always 1)

Clock Algorithm Page Replacement



Global/Local Page Replacement

- Local Replacement
  Victim among the process pages
  Limits the number of frames per process
- Global Replacement
  Any victim can be selected
  Good for high-priority processes
  Performance of one process depends on other processes
  Global is generally used: simple and increases system throughout
- your process could lose pages because my process is page-faulting

Frame Allocation Algorithms

- Frame Allocation Problem
  How many frames should be given to a process?
- Max number of frames: Physical Memory
  making one process happy is not going to please the other processes

Frame Allocation Policies

- Fair Allocation: m frames and n process $\rightarrow$ Give each process $\frac{m}{n}$ frames
- Proportional Allocation: if s_i is the size of process (and S $= \sum_i s_i$ is total size) give $\frac{s_i}{S} \times m$ frames
- Priority Allocation: tweak the above with priority

Thrashing

- Phenomenon observed on systems with a global page replacement policy and a high-level of multi-programming (many processes) using the whole memory
- process needs more frames → page-fault rate increases
- takes frames away from other processes → increasing their page-fault rates
- processes are moved from the ready queue to the waiting one (since they are waiting for the disk)
- CPU utilization decreases which is good for the CPU scheduler: It can start new processes which request memory frames and are sent to the waiting queue
  No work gets done: each process is waiting for pages
- the above is called Thrashing
  Paradox To increase the CPU utilization the multi-programming level must be reduced

Thrashing Prevention

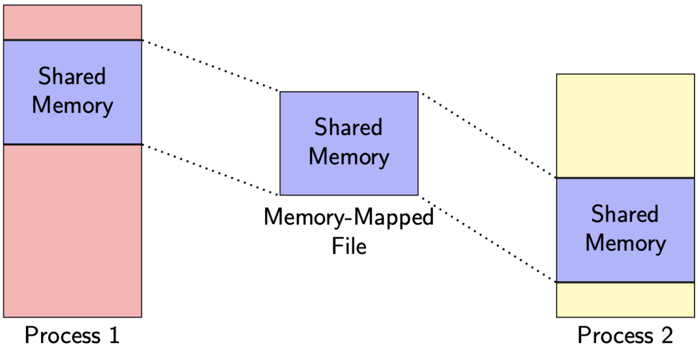
- Working Set Strategy
  Observe the pages referenced by each process (called the working set)
  When the sum of the sizes of all working sets gets greater than the number of memory frames, swap out an entire process and recover its frames
  Hence no thrashing (but one very unhappy process)
- Page-Fault Frequency Strategy
  - Monitor the page-fault rate for each process
  - If the rate is above some (fixed) upper bound, give the process another frame
  - If the rate is below some (fixed) lower bound, take a frame from the process
  - If a process requests a new frame but none is available: swap it out
- Thrashing and swapping are often use interchangeably. Formally though thrashing is the problem and swapping is the solution.

Aside: Memory-Mapped Files

- I/O is prohibitively expensive
  - Each access to a file requires a disk seek and a disk access
  - Out of question to read/write bytes one by one to a file
- On-disk address spaces are brought into RAM and virtualized
- Data files can be virtualized the same way, i.e., by mapping them to memory
- Memory Mapping
  - Map disk block(s) to a memory frame(s)
  - Initial access is expensive (and generates page faults)
  - Subsequent access is made in memory (and cheap-er)
  - The on-disk file may be updated at a convenient time, upon closing
  - Memory mapping is performed by dedicated system calls (mmap)
  - Potential concurrency issues: multiple processes can map the same file concurrently

Memory-Mapped Files and Shared Memory

- Memory mapping can be used to implement shared memory



- In Linux / FreeBSD, different mechanisms for POSIX shared memory (`shmget`) and memory mapping (`mmap`)
- In Windows shared memory is implemented only through memory mapping

- To access I/O devices, set aside ranges of memory addresses
- Loads/Stores to these addresses cause interaction with the device
- Convenient because all (memory-mapped I/O) devices look similar