Swapping, Paging, Virtual Memory

Swapping

- Move processes onto the disk when it yields or is blocked
- When its schedules, move it back to RAM

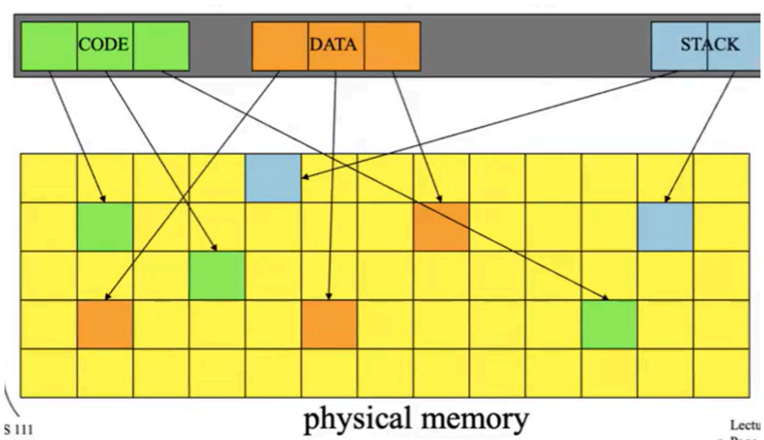
  Downsides to simple swapping
  - Moving the entire process out have high costs for context switches
  - Copy all RAM to disk and copy the other process's data back on to RAM
  - Processes are still limited to the amount of RAM they have

Segment Relocation

- Segment relocation uses base registers to compute a physical address from a virtual address
- Allowed us to move data sprung in physical memory
- Does not solve external fragmentation since segments are required to be contagious

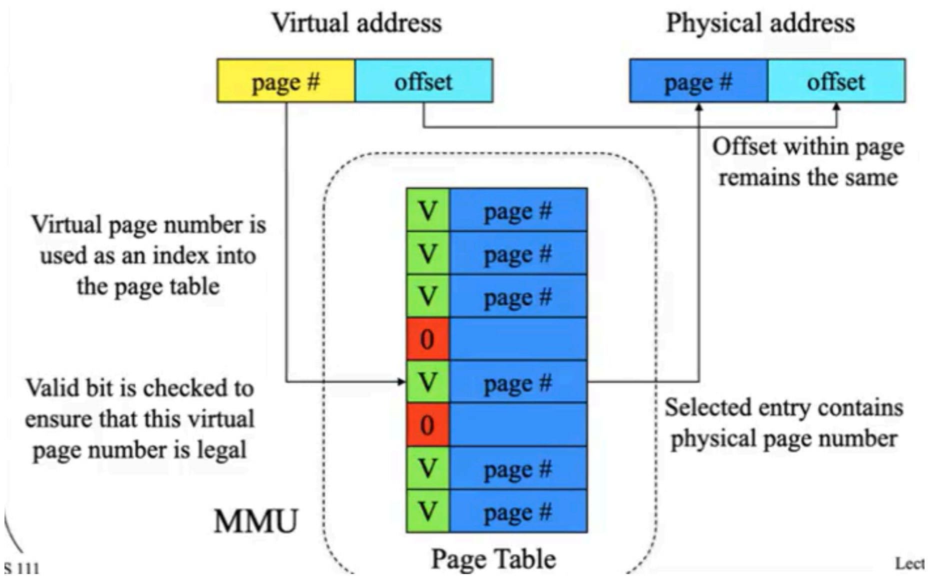
Paging

1. Divide physical memory into small units of a single fixed size (page frame)
2. Divided the virtual address space in the same way, where each virtual unit is a page
3. For each virtual address space page, store its data in one physical address page frame
4. Use a translation mechanism to convert virtual to physical pages



  Fragmentation
  - Segment is implemented as a set of virtual pages
  - External fragmentation doesn't exist (we never divide a page)
  - Small amounts of internal fragmentation

MMU Page Table Translation



1. Virtual address is made of a page number and offset (location within the page)
2. VPN used to index into the page table, valid bit is checked to ensure the page entry is valid
3. Page number stored in the page table entry is the first bits of the physical address
4. Offset is stored in the lower bits

Handling Large Page Tables

- MMU integrated into CPU chip to make it fast
- 16 M entries in a page table would mean we can't use registers. These would just be stored in normal memory then, making it slow
- We can't afford 2 bus cycles for each memory access
- MMU contains a cache of pages, which would result in many misses and cache invalidations

MMU and Multiple Processes

- If there are several running processes, each will need a set of pages
- Situations where there's more pages than page frames
- Some pages can be placed onto flash drives or persistent memory device
- Code can only access a page in RAM

Ongoing MMU Operation

- Add / Remove pages
    - Directly update the active page table in memory
    - Privileged instruction to flush stale entries in the table

- Switching from one process to another

- Maintains separate page tables for each process
- Privileged instructions loads pointer to new
- Reload instruction flushes previously cached entries

- How to share pages between multiple processes
    - Makes each page table point to same physical page
    - Can be read only or read/write sharing

Demand Paging

- Some programs are kept on disk and persistent storage
- Process spent need all its pages in memory to run

1. MMU must support page faults: page is on disk and not on RAM when it's needed
    - Generates a trap when its referenced
    - OS can bring in the requested page and retry the failed reference

2. Entire process doesn't need to be in memory to start running
    - Start each process with a subset of its pages
    - Load additional pages as program requests (could result in performance issues)

    Improving performance

    - Demand paging performs poorly if most memory references require disk access
    - Locality of Reference: place the next address you ask for is likely to close the last address you asked for
    - Programs that run consecutively likely have page tables located close to each other

    Locality of Reference

    - Code is a sequence of consecutive or nearby instructions
    - Typically access the current or previous stack frame
    - Many heap references to recently allocated structures

Page Fault

1. Initialize page table entries to not present
2. CPU faults if not present page is referenced
3. Fault enters kernel, just like any other exception
4. Forward to page fault handler
5. Determine which page is required and where its located
6. Schedule I/O to fetch it, then block the process
7. Ensure the page table entry points to newly read in page
8. Backup user mode PC to retry to failed instruction
9. Return to user-mode and continue running

- While this is happening, other processes can still run

Impact

- Don't effect correctness
- Only slow a process down
- After a fault is handled, the desired page is in RAM
- Process runs again and can use it
- Programs never crash because of page faults, although they can be slow if there are too many

Demand Paging Performance

- Page faults can block processes
- Overhead (current process is blocked when reading pages, resulting in delayed execution)
- Having the right page sin memory results in fewer faults
- Can't control which pages are read in, so improving which pages are removed from RAM is important

Virtual Memory

- Generalization of demand paging
- Abstraction of memory
  - Very large quantity of memory for each process
  - All directly accessible through normal addressing

Virtual Memory Concept

1. Give each process a large address space
2. Allow processes to request segments within that space
3. Use dynamic paging and swapping to support abstraction

Replacement Algorithms

- Goal is to have each page already in memory
- We can't predict which pages will be accessed next
- Use locality of access to determine which pages to move out of memory and onto the disk

Basics of Page Replacement

- Keep some set of all pages in memory
- In some circumstances, replace one of them with another page that's on disk
- Paging hardware and MMU translation allows us to choose any page for ejection to disk

Optimal Replacement Algorithm

- Replace the page that will be referenced furthest in the future
- Delays the page fault for a long time
- Works best on programs we have run before, so we know exactly what happens

- The more times we are correct, the fewer page faults

Approximating Optimality

- Rely on locality of reference
- Note which pages have recently been used
- Use this data to predict future behavior
- If it holds, recently accessed pages will be accessed again

Replacement algorithms

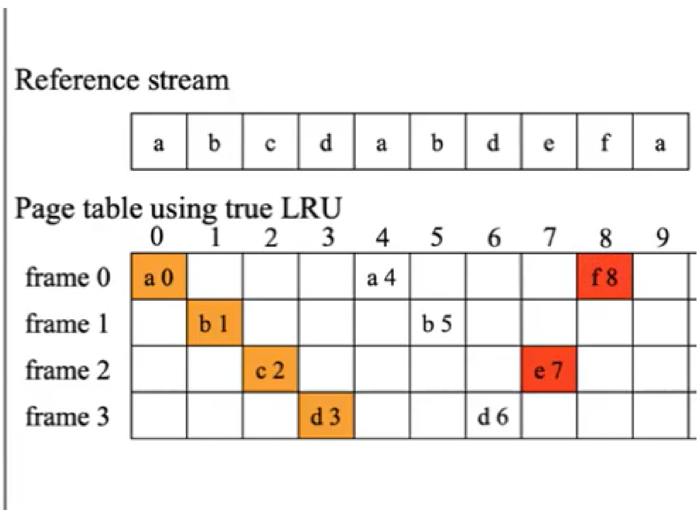
- Random, First In First Out, Least Frequently used don’t work
- Least Recently Used asserts the future is like the recent past
- If we haven't used a page recently, we probably won’t use it soon

Naive LRU

- Each time a page is accessed, record the time
- When a page needs to be ejected, look at all timestamps for pages in memory
- Choose the one with the oldest timestamp, which requires storing the timestamps
- Every timestamp needs to be searched

True LRU Page Replacement

- Maintain multiple frames and remove the earliest used one in the frame



Maintaining Information for LRU

- Keeping time in the MMu
  - MMU translations must be fast
  - No space on the MMU for time keeping registers

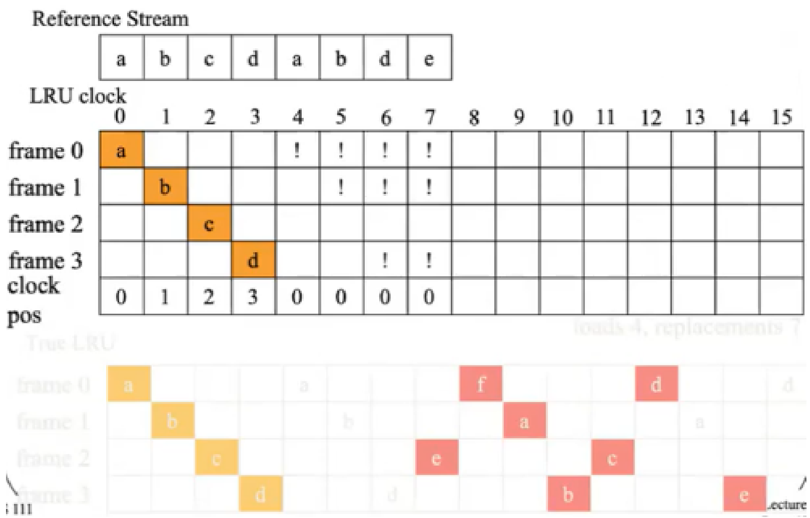
- Maintain info in software
  - Complicated and time consuming

- We need:
  - No extra instructions per memory reference
  - Can't have extra page faults

- Can scan the entire list on each replacement

Clock Algorithms

- All page frames are organized in a circular list
- MMU sets a reference bit for the frame on access (set it to 1)
- Scan whenever we need another page
    - For each frame, ask MMU if frame's reference bit is set (there is something stored there)
    - If so, clear the reference bit in the MMU and skip this frame (set it to 0)
    - When the reference bit is 0, then it means no one has touched that page for some time
    - The next search will start at this position

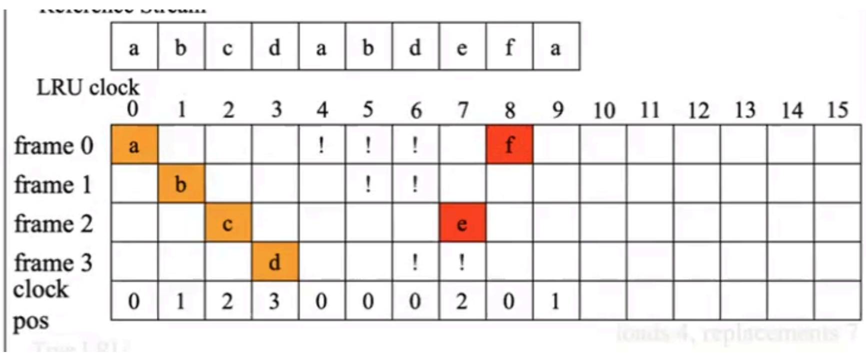


1. Fill up the available frames with pages, incrementing the clock each time. Clock time represent which page frame we will try to replace

2. Once we fill up all frames, the clock position returns to 0

3. Whenever we need a page that is already in memory, we mark it with a 1
    - This signals that the page was used recently
    - Searching a page marked with 1 doesn't increment the clock cycle

4. When we reach a new page, and there are no available spots:
    - Reset the reference bit to a 0 for each frame with a 1 until it reaches a frame with 0
    - Add the new page to the first frame which is 0
    - Set the clock to that frame



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Reference Stream	a	b	c	d	a	b	d	e	f	a						
frame 0	a				!	!	!		f							
frame 1		b				!	!									
frame 2			c					e								
frame 3				d			!	!								
clock pos	0	1	2	3	0	0	0	2	0	1						

Comparing True LRU To Clock Algorithm

- Same number of loads and replacements
  - But it didn't replace the same pages
- Both are approximations to the optimal (neither are completely)
- LRU clock decisions is very close to true LRU (pretty much the optimal)

Page Replacement and Multiprogramming

- Clearing page frames on each context switch is expensive
- Choices: single pool, fixed allocation of page frames per process, working set based page frame allocation

  Single Global Page Frame Pool
  - Treat the entire set of page frames as a shared resource
  - Whichever process page is least recently used is replaced
  - Bad interaction with round robin
  - Process last in the queue will have all pages swapped out if they aren't being used frequently

  Per Process Page Frame Pools
  - Set aside a number of page frames for each running process
  - Isolates the effects of changes within a process
  - Downsides:
    - Which pages are needed changes over time
    - Some processes need more or less page frames

Working Set

- Give each running process an allocation of page frames
- Working set: set of pages used by a process in a fixed length sampling window in the immediate past
- Allocate enough page frames to hold each process' working set
- Each process runs replacements within its own set

Optimal Working Sets

- Optimal working set for a process is the number of pages needed during the next time slice
- Using fewer frames results in pages being continuously replaced
- By observing the processes' behavior, the working set size is changed

Implementing Working Sets

- Manage the working set size
    - Assign page frames against themselves in working set
    - Observe paging behavior
    - Adjust number of assigned page frames

- Page stealing algorithm
    - Adjusts working sets
    - Track last time for each page, for owning process
    - Find page least recently used
    - Processes that need more pages are allocated more
    - Processes that don't use their pages tend to lose them

Thrashing

- Working set size characterizes each process, or the number of pages it needs to run for t milliseconds
- Processes won't have enough pages in memory, resulting in constant page faults
- Taking page frames from working sets results in contiguous cycle of processes that don't have enough page frames
- Results in slower processes
- Continuous relocation of pages due to page faults results in user code never running

Preventing Thrashing

- Cannot add more memory and cannot reduce working set sizes
- Reduce the number of competing processes by swapping some ready processes out
- Swapped out process won't run for a long time, but can be round robin in

Clean vs Dirty Pages

- If a page was recently passed in from disk then there are two copies: one on disk and one in memory
    - Results in two I/O for a page fault

- If the in-memory copy hasn't be modified, there is an identical valid copy on disk
    - In-memory copy is "clean"
    - Clean pages can be replaced without writing them back to the disk
    - (the one in the page frame can be discarded when ejecting)

- If the in-memory copy has been modified, the copy is no longer recent
    - In memory copy is “dirty”
    - If paged out of memory, it must be written to disk

Dirty Pages and Page Replacement

- Clean pages can be replaced at any time
    - Copy on the disk is already up to date

- Dirty pages must be written to disk before frame can be reused by another process
    - Slow operation
    - Removing only clean pages would limit the pages that can be removed from frames

- Need to avoid having too many dirty pages while maintaining speed

Pre-Emptive Page Laundering

- Clean pages give memory manager flexibility
    - Many pages can be replaced

- Increase flexibility by converting dirty pages to clean ones

- Ongoing background write-out of dirty pages
    - Find and write out all dirty non running pages
    - Don’t write out a page that is actively in use
    - We have the assumption that we will eventually page it out