10 Virtual Memory-os

mostly based on chapter 10 of the book and
https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

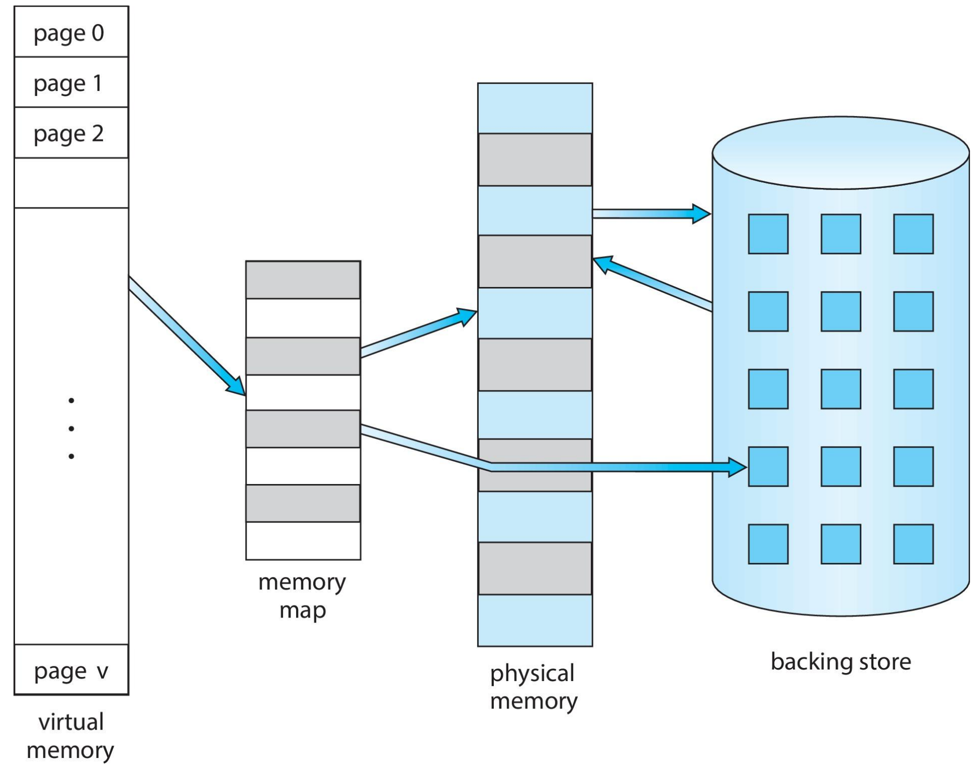
- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster

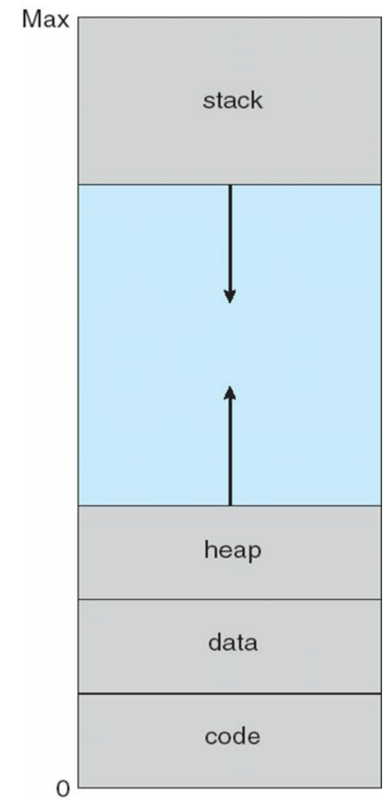
Virtual memory

- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes

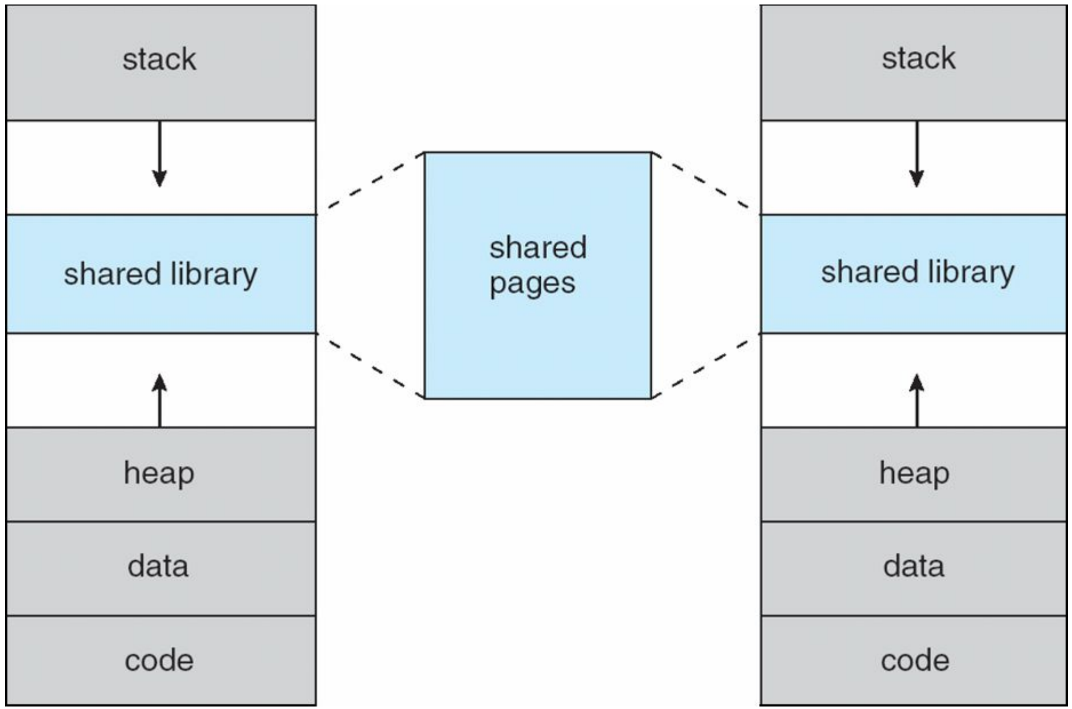


Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow "down" while heap grows "up"
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc.
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during `fork()`, speeding process creation



Shared Library Using Virtual Memory

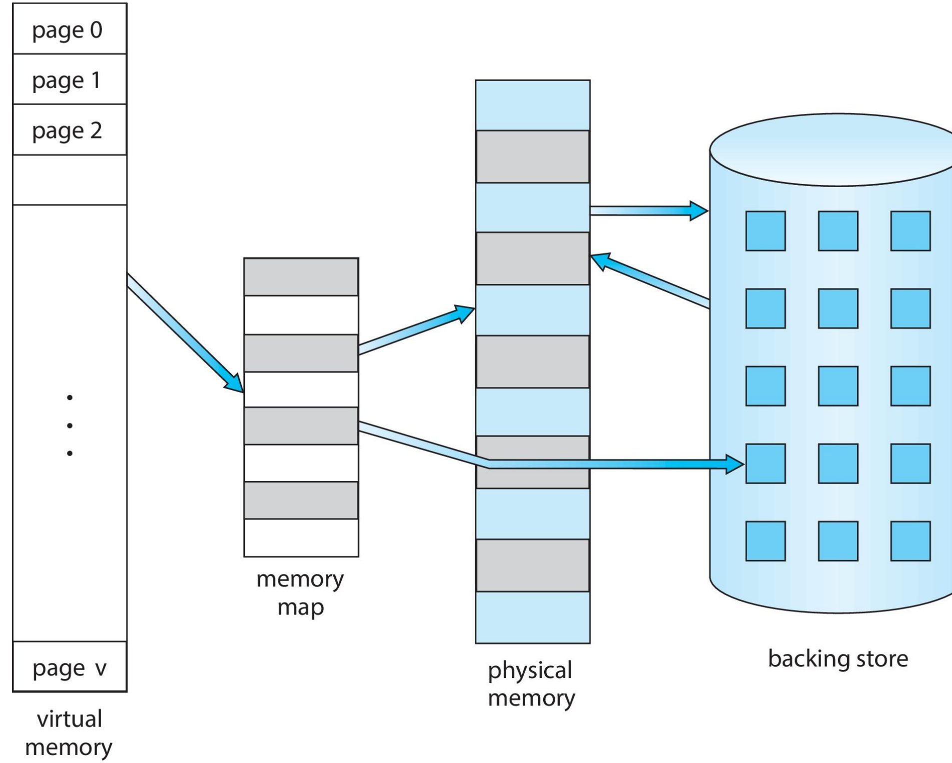


Virtual memory (Cont.)

- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Paging implementation

- Use disk to simulate **virtual memory > physical memory**
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

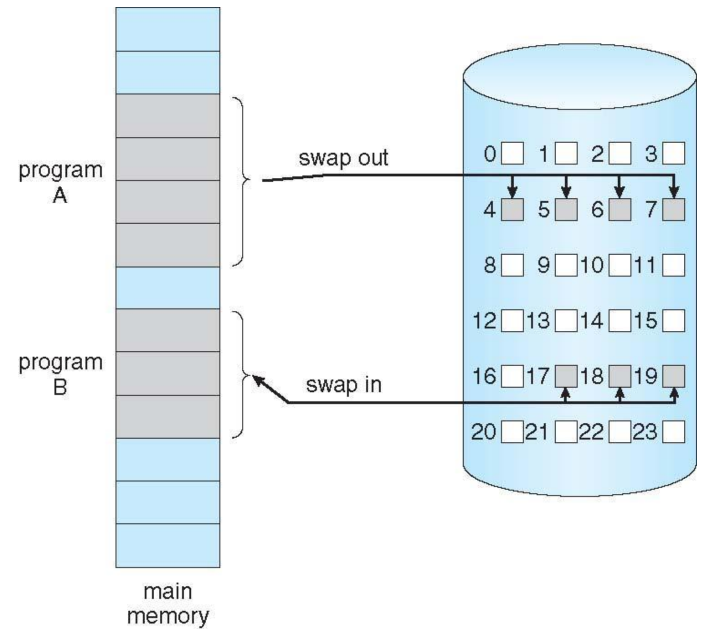


Disk much, much slower than memory

- **Goal:** run at memory speed, not disk speed

Demand paging choices

- Could bring entire process into memory at load time
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a pager



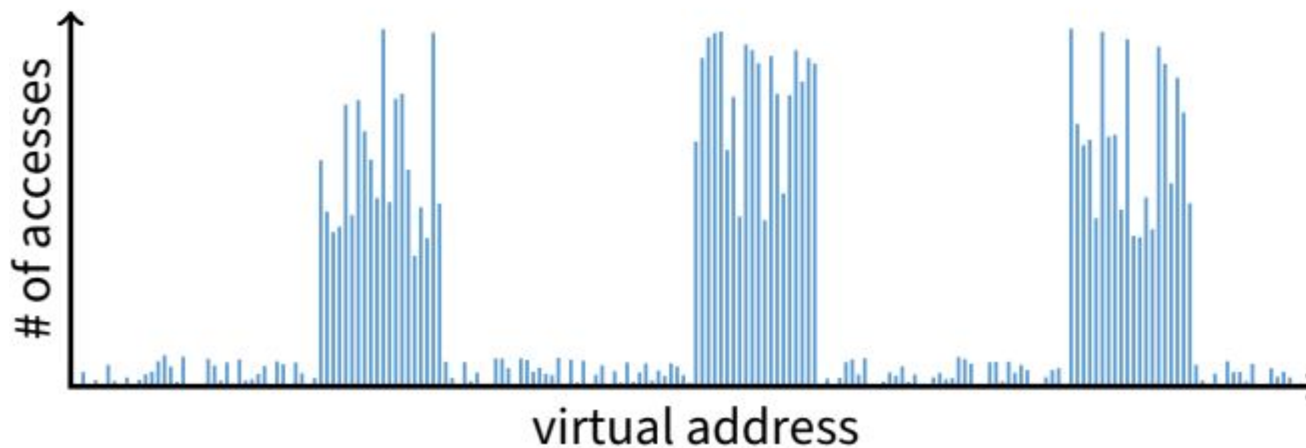
Paging challenges

Disk much, much slower than memory

➔ **Goal**: run at memory speed, not disk speed

80/20 rule:

20% of memory gets 80% of memory accesses

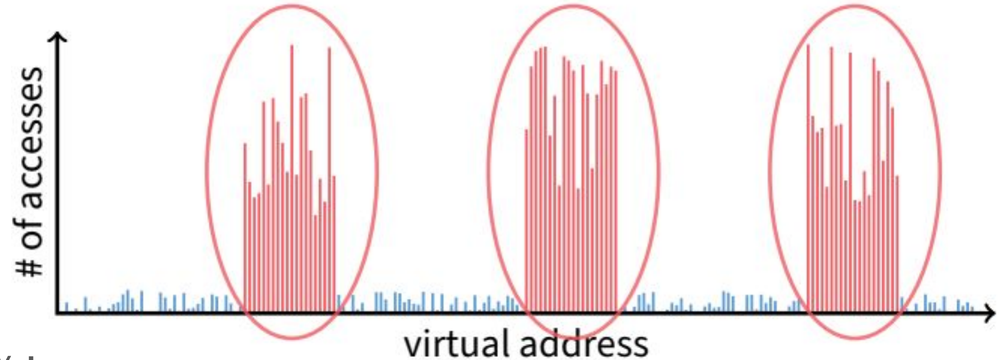


https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

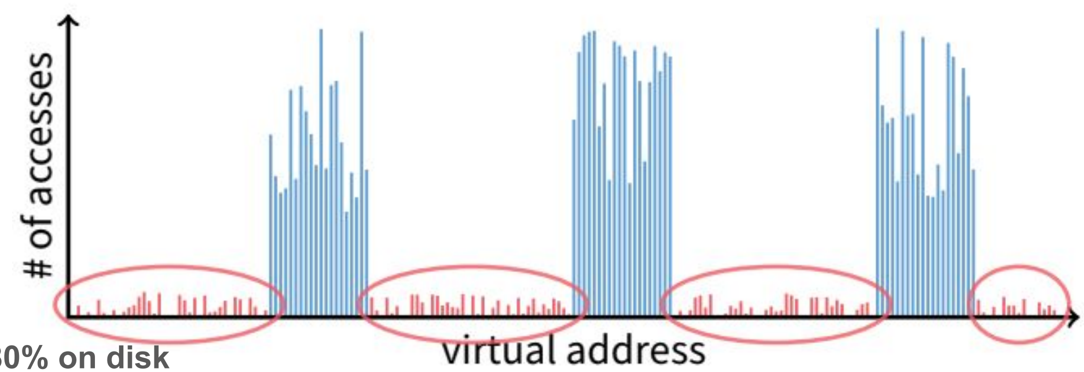
Working set model

80/20 rule:

20% of memory gets 80% of memory accesses



Keep the hot 20% in memory



Keep the cold 80% on disk

https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

Paging challenges

How to resume a process after a fault?

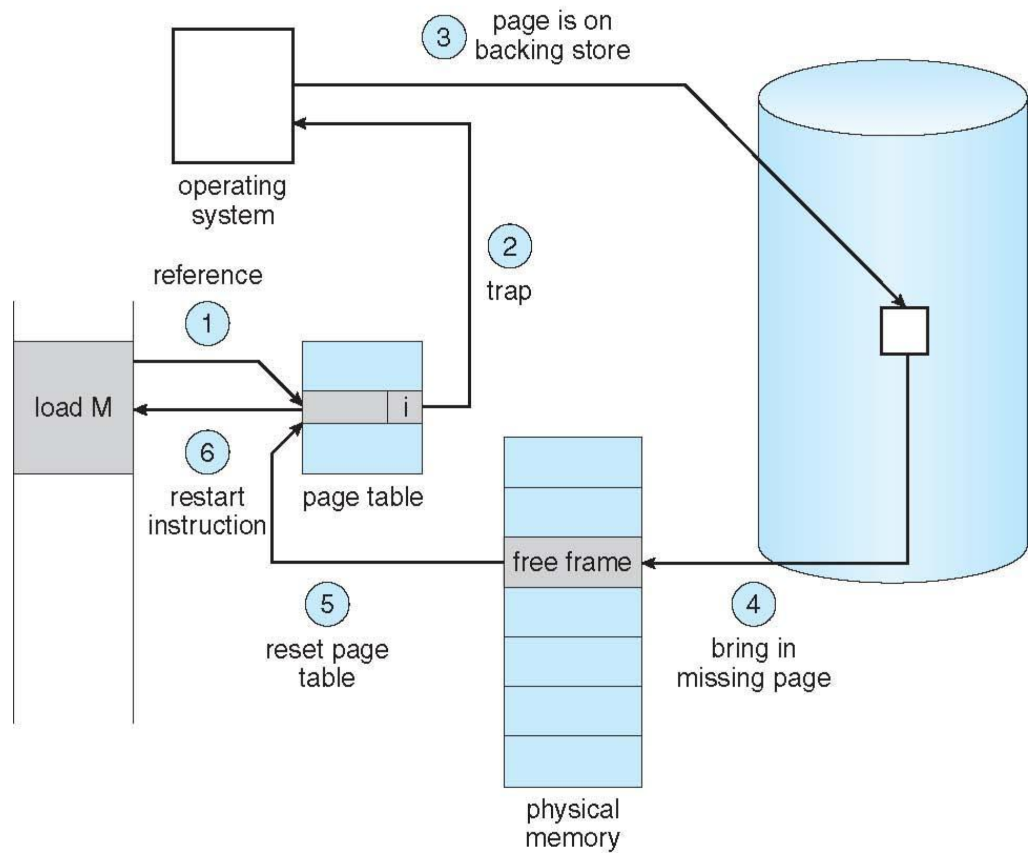
- Need to save state and resume
- Process may have been in the middle of an instruction!

What to fetch from disk?

- Just needed page or more?

What to eject?

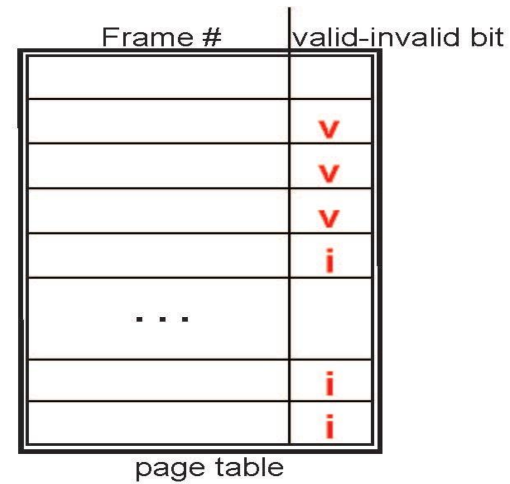
- How to allocate physical pages amongst processes?
- Which of a particular process's pages to keep in memory?



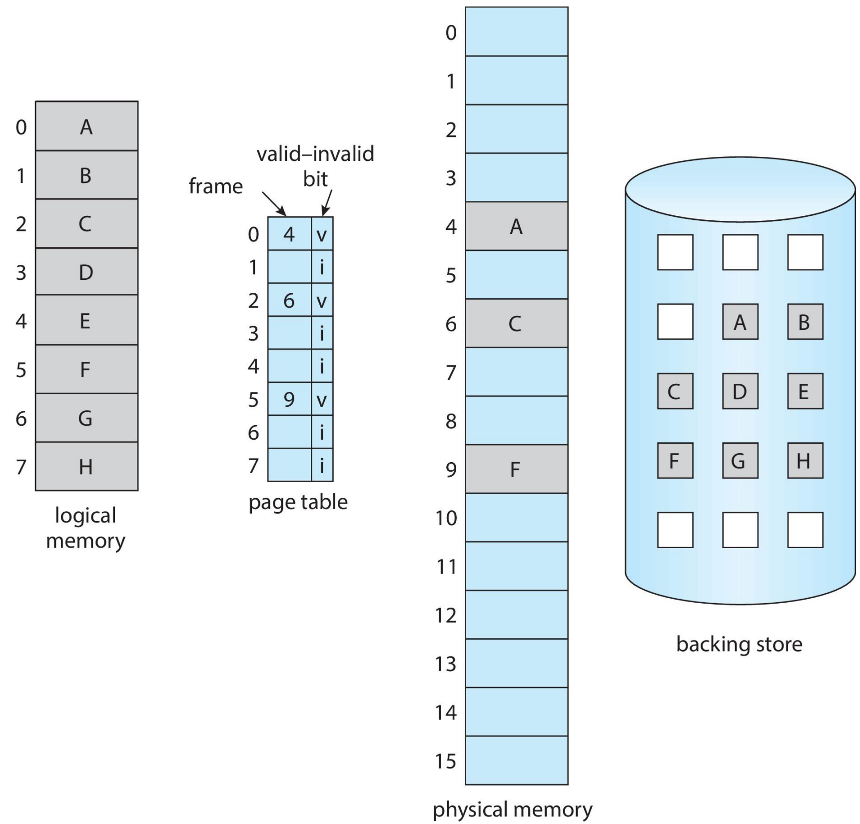
https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

Valid-Invalid Bit

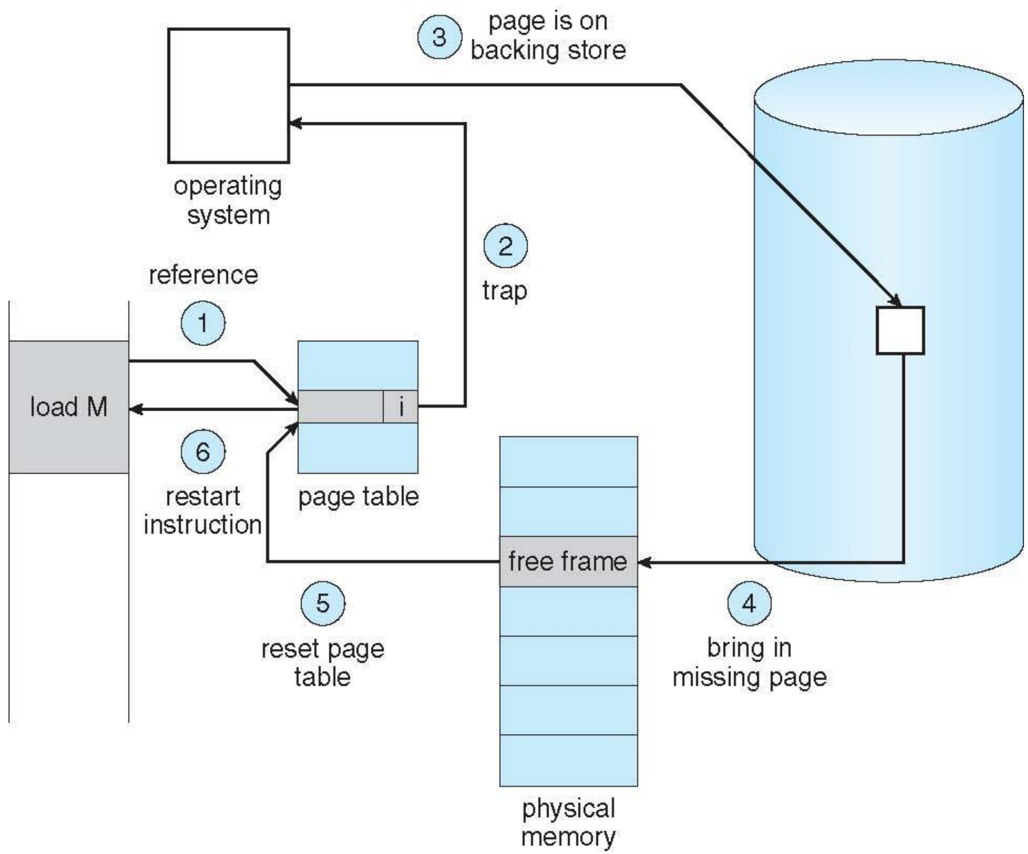
- With each page table entry a valid–invalid bit is associated
  (**v** ⇒ in-memory – **memory resident**, **i** ⇒ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault



Page Table When Some Pages Are Not in Main Memory



Steps in Handling a Page Fault

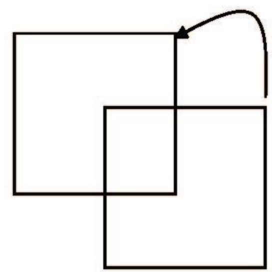


Steps in Handling Page Fault

1. If there is a reference to a page, first reference to that page will trap to operating system
   - Page fault
2. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
3. Find free frame
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory
   Set validation bit = **v**
6. Restart the instruction that caused the page fault

Instruction Restart

- Consider an instruction that could access several different locations
    - Block move



    - Auto increment/decrement location
    - Restart the whole operation?
        - What if source and destination overlap?

Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

Restarting instructions

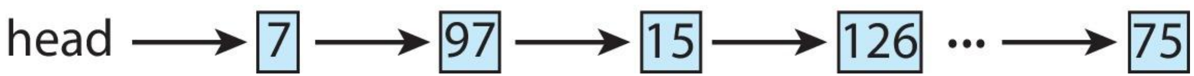
- Hardware must allow resuming after a fault
- Hardware provides kernel with information about page fault
    - Faulting virtual address
        - %cr2 reg on x86
    - Address of instruction that caused fault
    - Was the access a read or write?
    - Was it an instruction fetch?
    - Was it caused by user access to kernel-only memory?

- **Observation:** Idempotent instructions are easy to restart
    - E.g., simple load or store instruction can be restarted
    - Just re-execute any instruction that only accesses one address
- Complex instructions must be re-started, too
    - E.g., x86 move string instructions
    - Specify src, dst, count in %esi, %edi, %ecx registers
    - On fault, registers adjusted to resume where move left off

https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

Free-Frame List

- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
- Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests.



- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand**
    - the content of the frames zeroed-out before being allocated.
- When a system starts up, all available memory is placed on the free-frame list.

Stages in Demand Paging – Worse Case

1. Trap to the operating system
2. Save the user registers and process state
3. Determine that the interrupt was a page fault
4. Check that the page reference was legal and determine the location of the page on the disk
5. Issue a read from the disk to a free frame:
    a) Wait in a queue for this device until the read request is serviced
    b) Wait for the device seek and/or latency time
    c) Begin the transfer of the page to a free frame

Stages in Demand Paging (Cont.)

6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of Demand Paging

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- Page Fault Rate $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

EAT = (1 – *p*) x memory access
+ *p* (page fault overhead
+ swap page out
+ swap page in)

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p x 200 + p x 8,000,000

  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

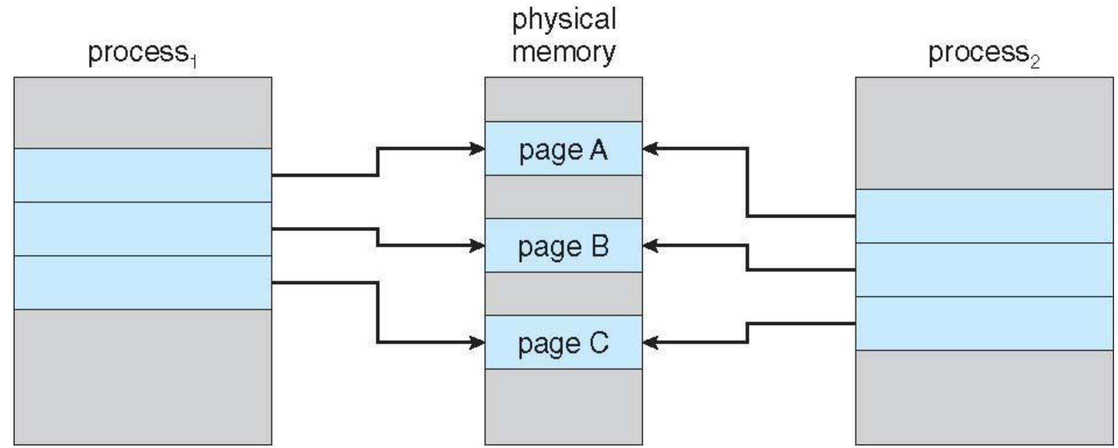
Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks, less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

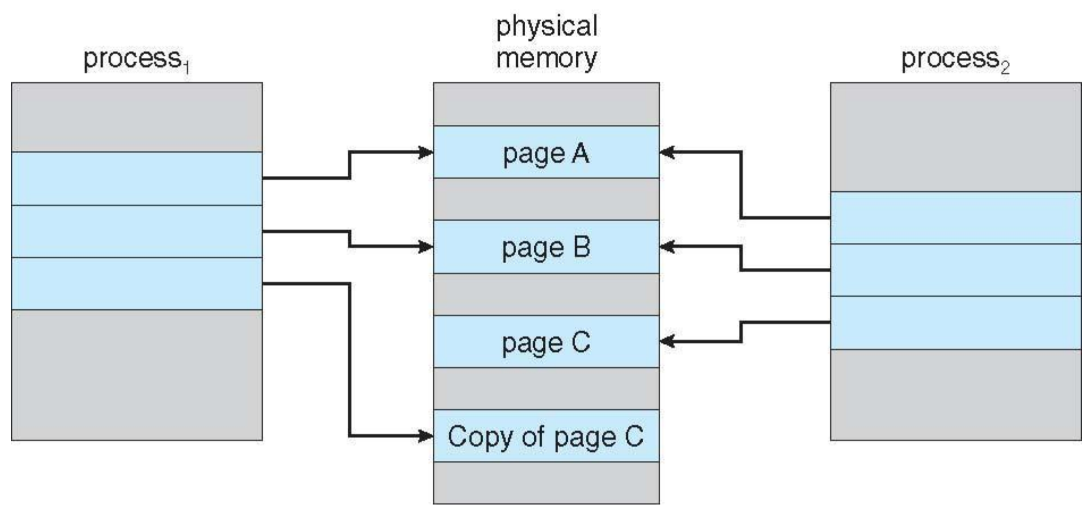
Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - Pool should always have free frames for fast demand page execution
    - Don't want to have to free a frame as well as other processing on page fault
  - Why zero-out a page before allocating it?
- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient

Before Process 1 Modifies Page C



Copy-on-Write: After Process 1 Modifies Page C



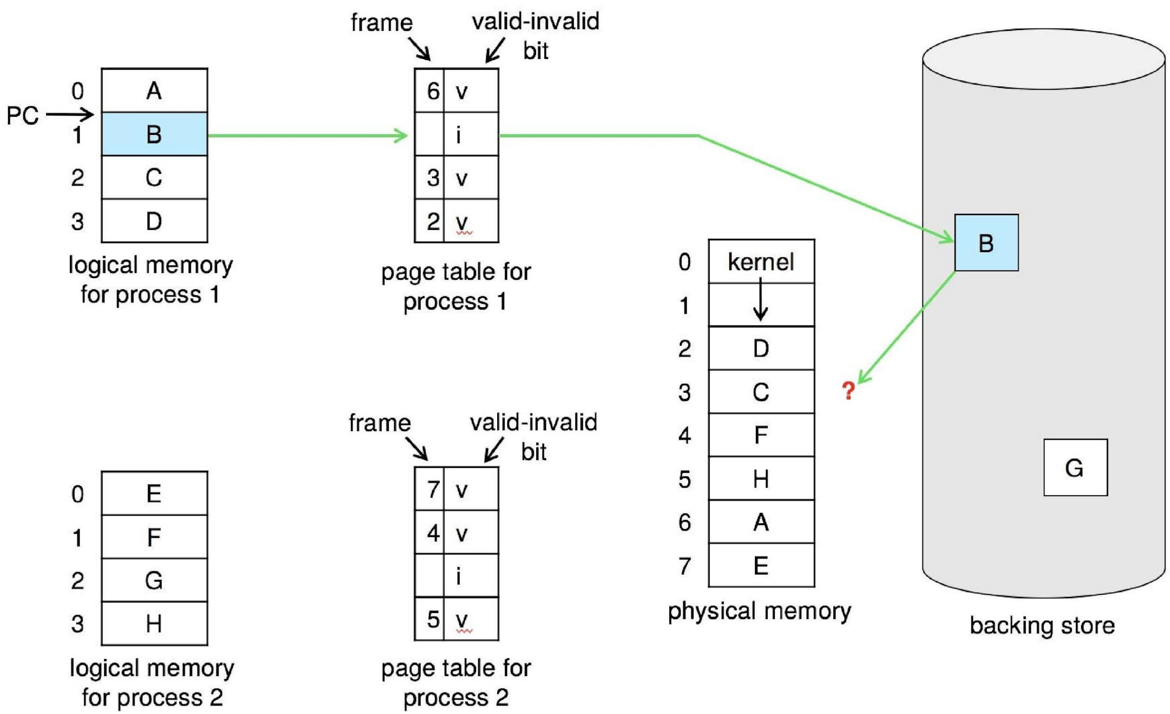
What Happens if There is no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory

Need For Page Replacement

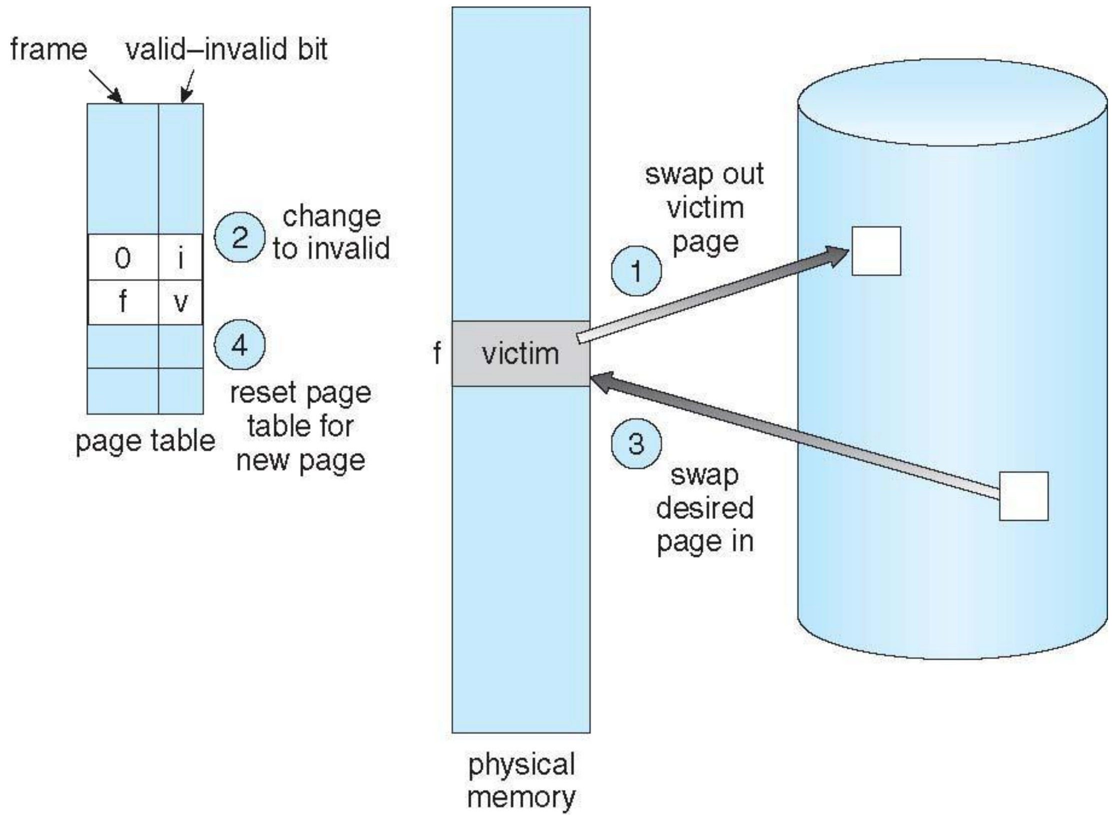


Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
   - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap

Note now potentially 2 page transfers for page fault – increasing EAT

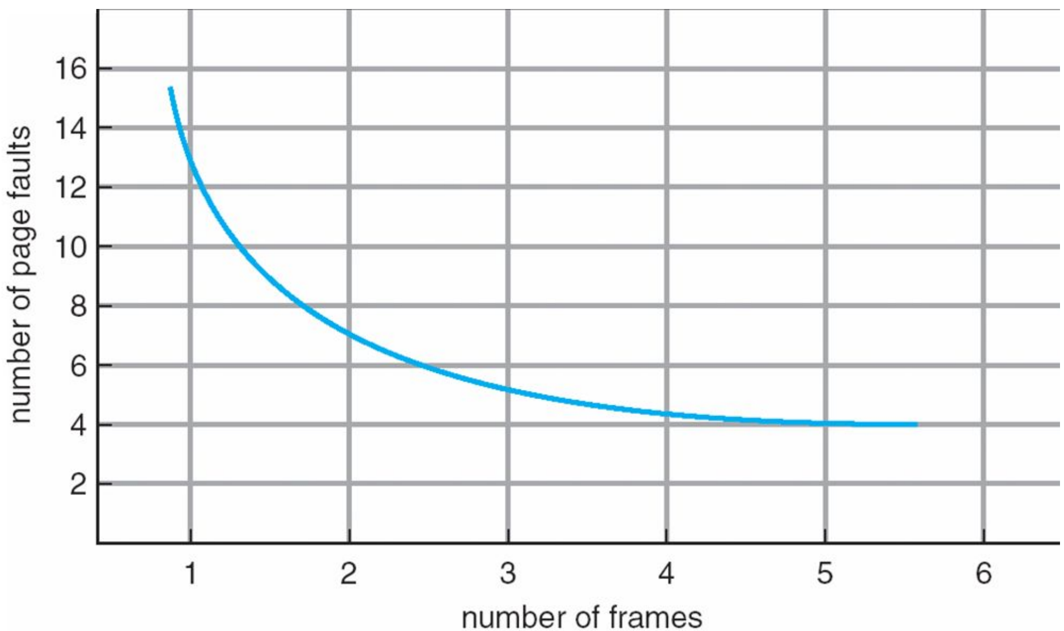
Page Replacement



Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm
  - by running it on a particular string of memory references (reference string)
  - and computing the number of page faults on that string

Graph of Page Faults Versus the Number of Frames



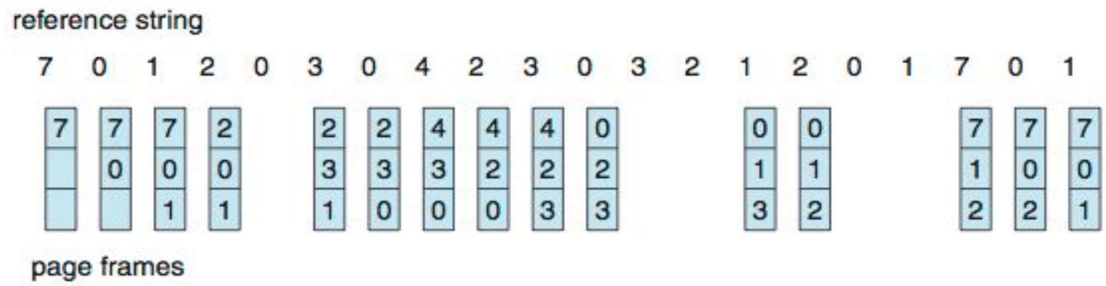
First-In-First-Out (FIFO) Algorithm

- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)

- How to track ages of pages?
  - Just use a FIFO queue

- **Number of faults?**

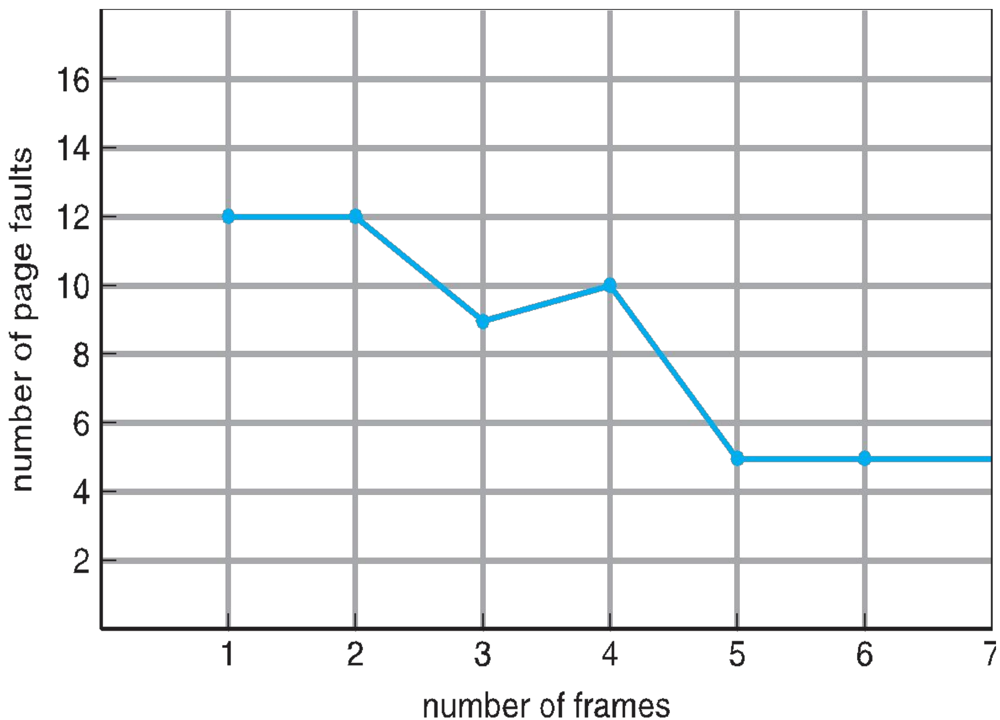
First-In-First-Out (FIFO) Algorithm



15 page faults

- Can vary by reference string: consider **1,2,3,4,1,2,5,1,2,3,4,5**
  - 3 physical pages: 9 page faults
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**
    - **4 physical pages: 10 page faults**

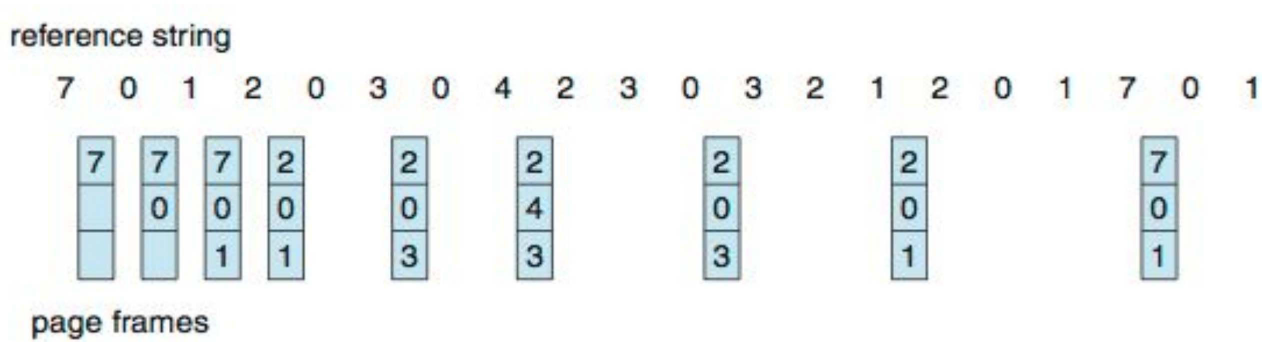
FIFO Illustrating Belady's Anomaly



More frames (physical memory) doesn't always mean fewer faults

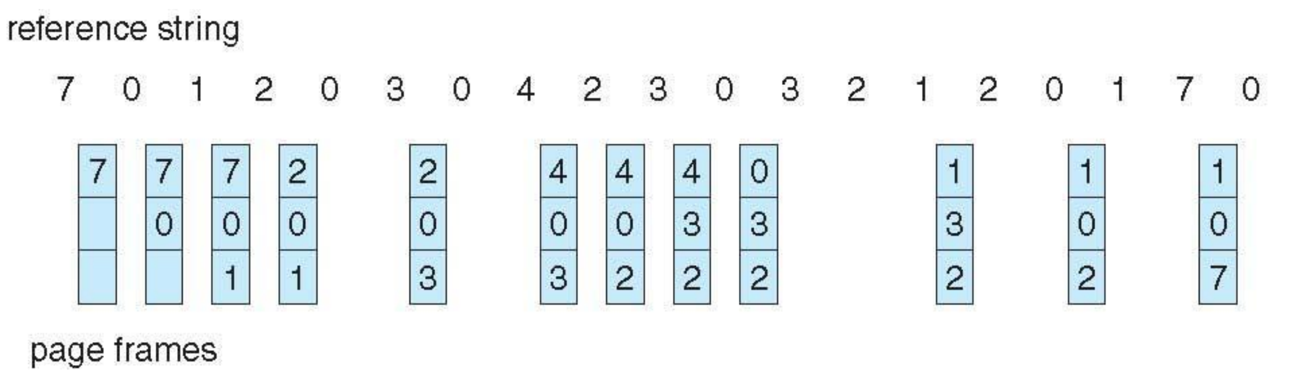
Optimal Page Replacement

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - **Can’t read the future**
- Used for measuring how well your algorithm performs



Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



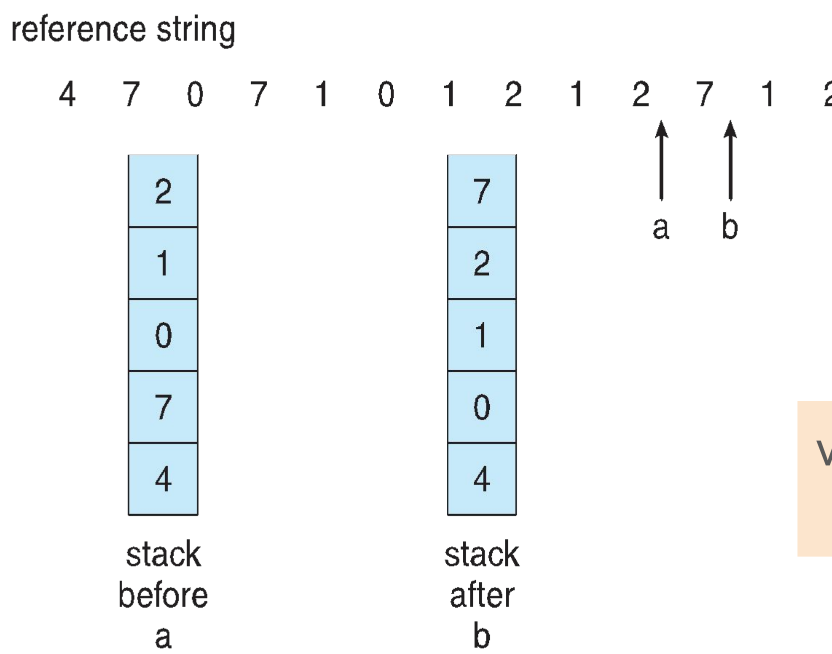
- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

LRU Algorithm (Cont.)

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a **double linked list** form:
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - **But each update more expensive**
  - No search for replacement

LRU Algorithm (Cont.)

- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly
- Use Of A Stack to Record Most Recent Page References



very expensive!

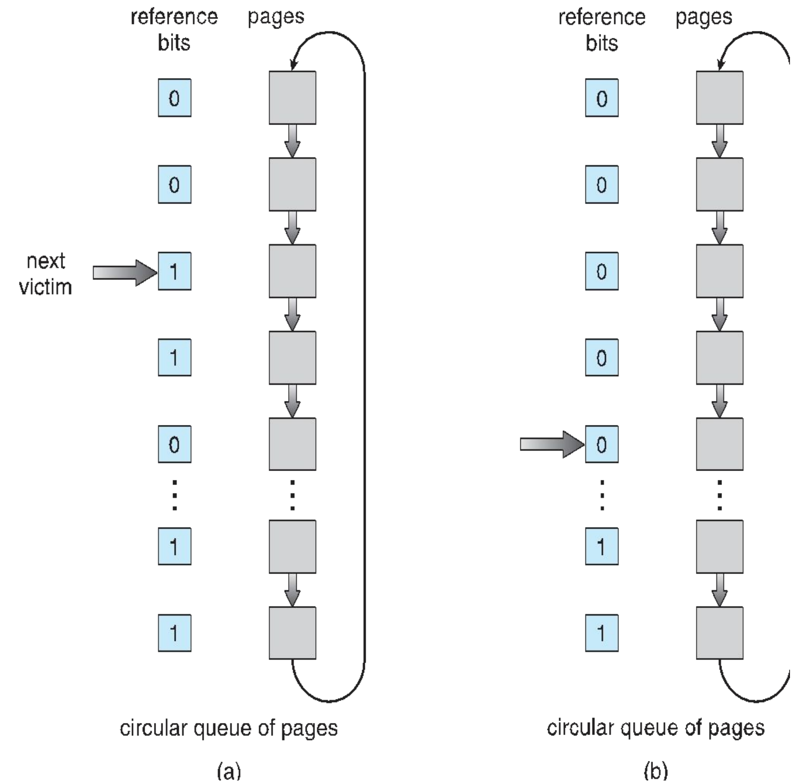
- approximate LRU

LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **Reference (accessed) bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however

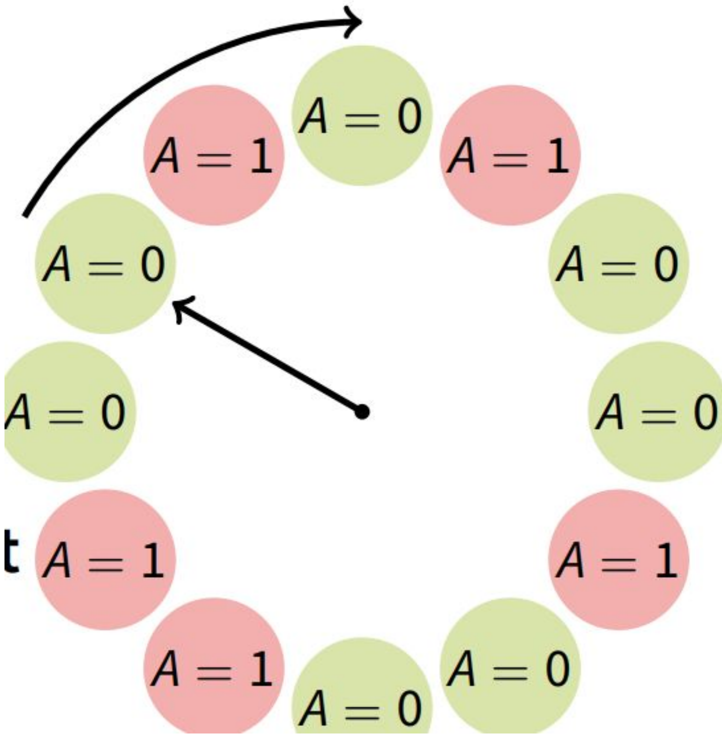
LRU Approximation: Clock algorithm

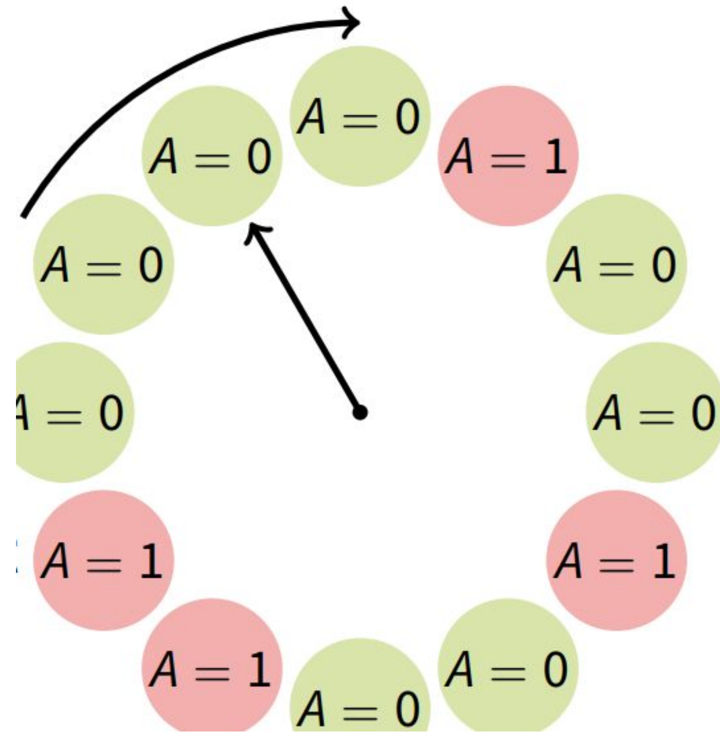
- **Second-chance replacement algorithm (clock algorithm)**
  - Generally FIFO, plus hardware-provided reference bit
  - **Clock** replacement
  - If page to be replaced has
    - **Reference bit = 0 -> replace it**
    - **reference bit = 1 then**:
      - **set reference bit 0,** leave page in memory
      - skip to next page, subject to same rules



for linux implementation see
https://www.kernel.org/doc/gorman/html/understand/understand013.html

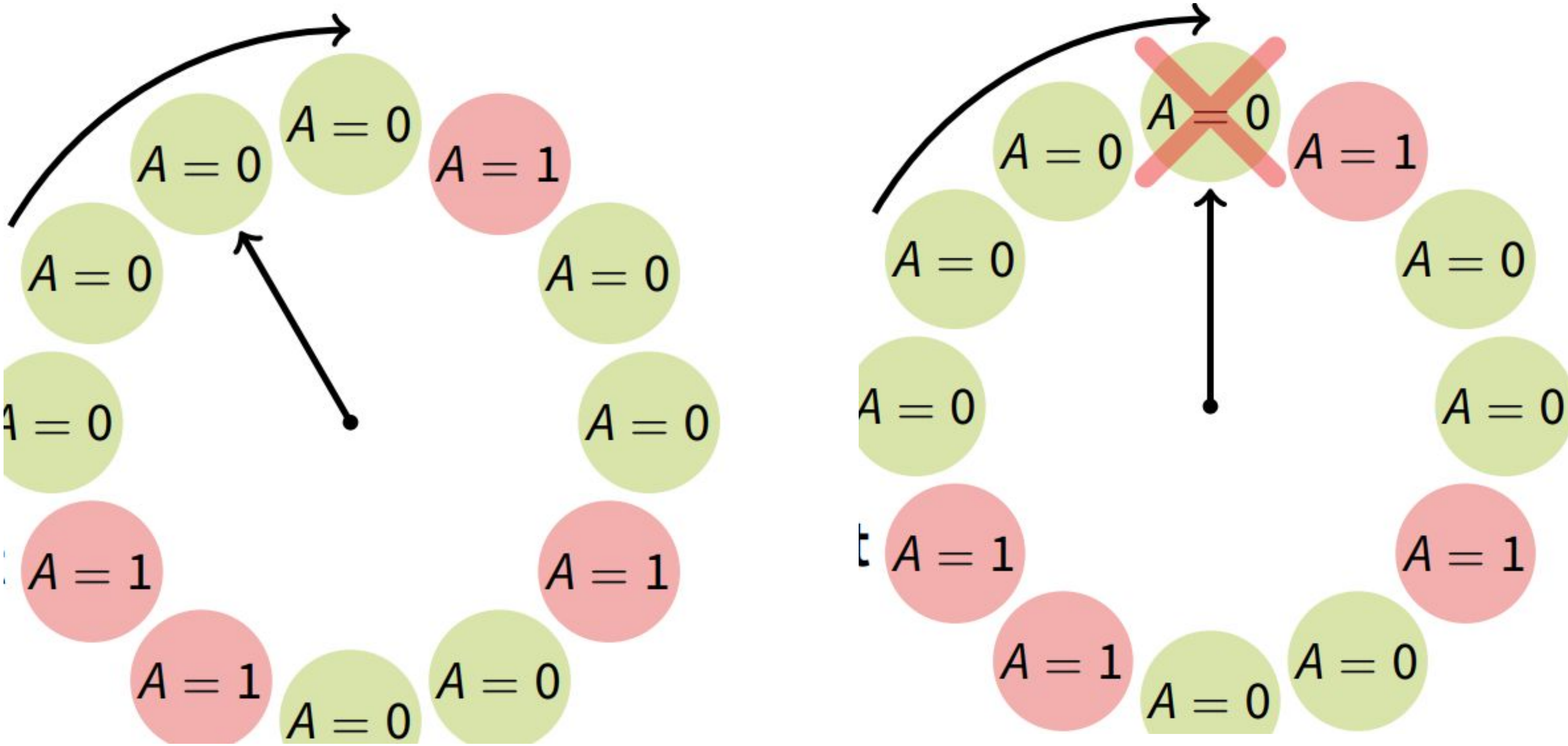
Clock algorithm: bit = 1, skip





https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

Clock algorithm: bit=0, eviction



https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

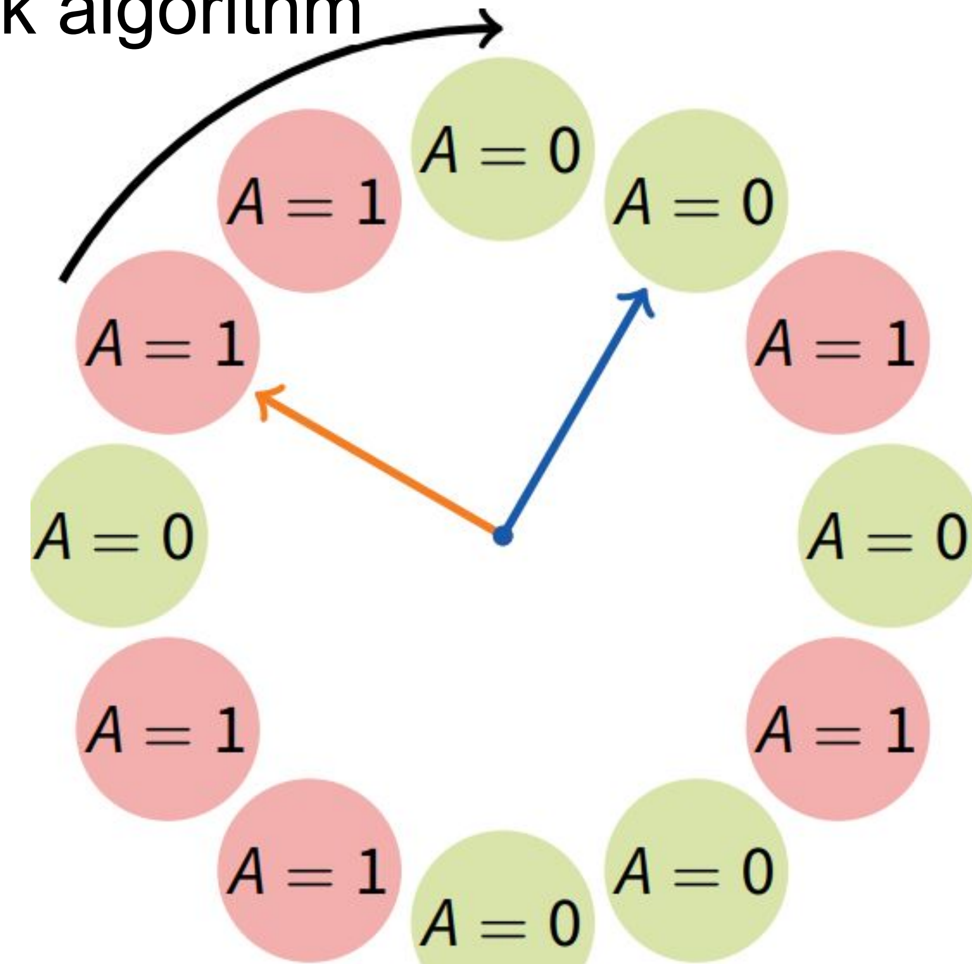
LRU Approximation: Clock algorithm

Large memory may be a problem

- Most pages referenced in long interval

Add a second clock hand

- Two hands move in lockstep
- **Leading hand clears A bits**
- **Trailing hand evicts pages with A=0**



https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

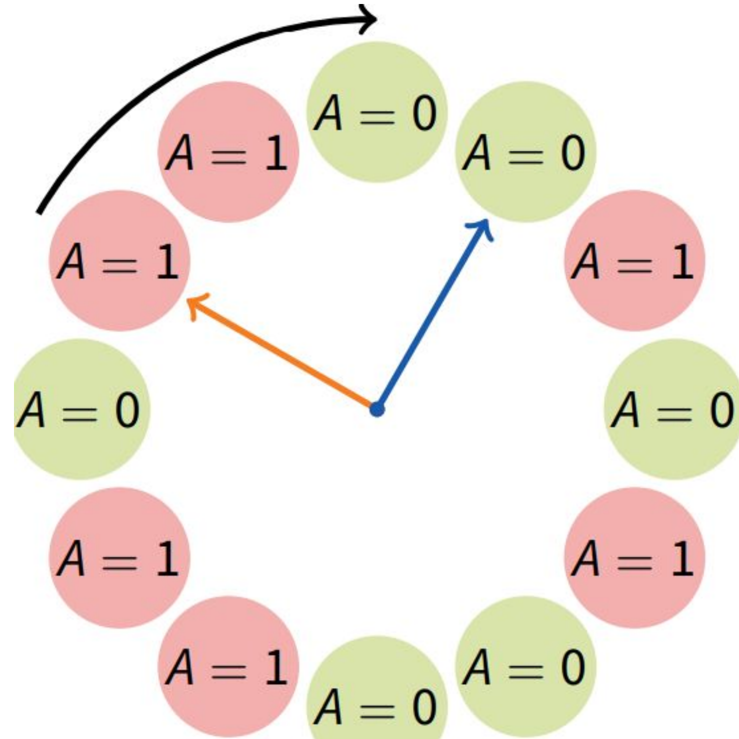
LRU Approximation: Clock algorithm

Can also take advantage of hardware **Dirty** bit

- (0, 0)Unaccessed, Clean
- (0, 1)Unaccessed, Dirty
- (1, 0)Accessed, Clean
- (1, 1)Accessed, Dirty
- Consider clean pages for eviction before dirty

Or use n-bit accessed count instead just A bit

- On sweep: count = (A << (n − 1)) | (count >> 1)
- Evict page with lowest count



https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

Other Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common
- Random eviction
  - Dirt simple to implement
  - Not overly horrible (avoids Belady & pathological cases)
- **Least Frequently Used (LFU) Algorithm**:
  - Replaces page with smallest count
- **Most Frequently Used (MFU) Algorithm**:
  - Based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Neither LFU nor MFU used very commonly

Page-Buffering Algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can given direct access to the disk, getting out of the way of the applications
  - **Raw disk** mode
- Bypasses buffering, locking, etc.

Allocation of Frames

- Each process needs ***minimum*** number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- ***Maximum*** of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some as free frame buffer pool
- Proportional allocation – Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

$$s_i = \text{size of process } p_i$$
$$S = \sum s_i$$
$$m = \text{total number of frames}$$
$$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$$

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 62 \approx 4$$
$$a_2 = \frac{127}{137} \times 62 \approx 57$$

Global vs. Local Allocation

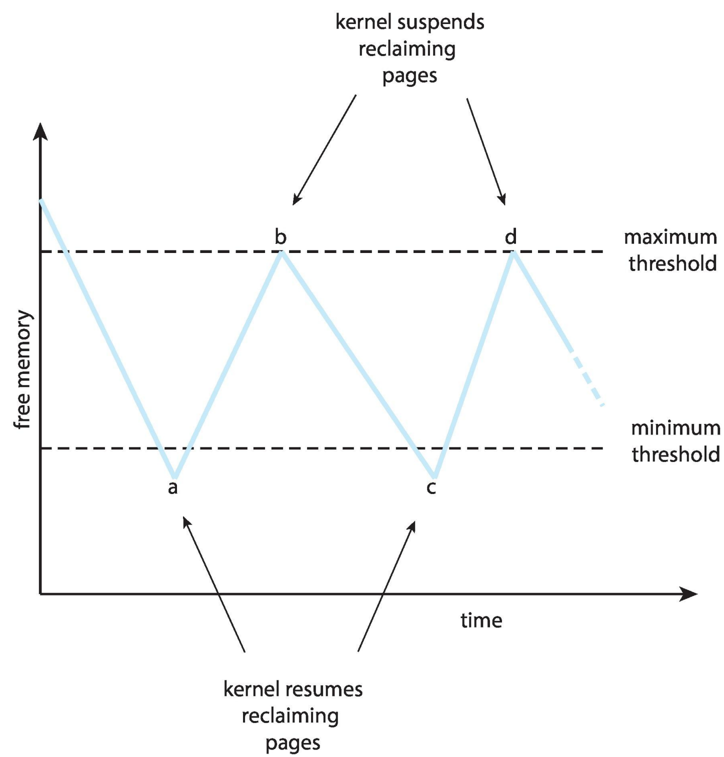
- **Global replacement** – Global allocation doesn't consider page ownership, process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common

- **Local replacement** –Local allocation isolates processes (or users), each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

Reclaiming Pages

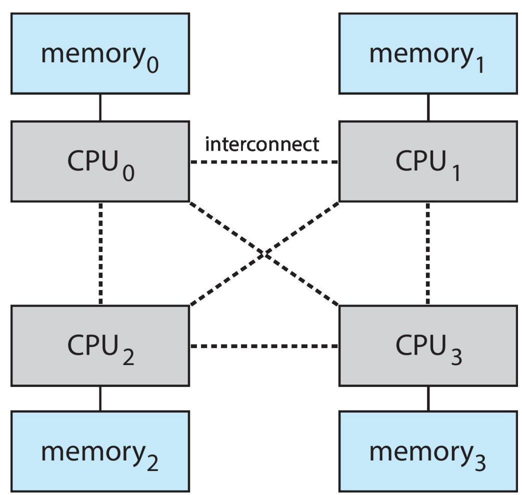
- A strategy to implement global page-replacement policy
- All memory requests are satisfied from the free-frame list, rather than waiting for the list to drop to zero before we begin selecting pages for replacement,
- Page replacement is triggered when the list falls below a certain threshold.
- This strategy attempts to ensure there is always sufficient free memory to satisfy new requests.

Reclaiming Pages Example



Non-Uniform Memory Access

- So far, we assumed that all memory accessed equally
- Many systems are **NUMA** – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- NUMA multiprocessing architecture



Non-Uniform Memory Access (Cont.)

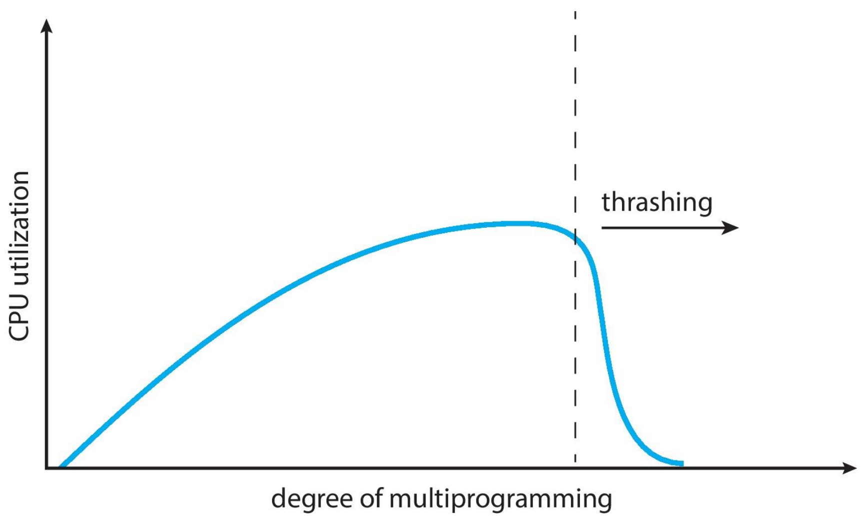
- Optimal performance comes from allocating memory “close to” the CPU on which the thread is scheduled
    - And modifying the scheduler to schedule the thread on the same system board when possible
    - Solved by Solaris by creating **lgroups**
        - Structure to track CPU / Memory low latency groups
        - Used my schedule and pager
        - When possible schedule all threads of a process and allocate all memory for that process within the lgroup

Thrashing

- If a process does not have "enough" pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system

Thrashing

- **Thrashing**. A process is busy swapping pages in and out



- Why does demand paging work?

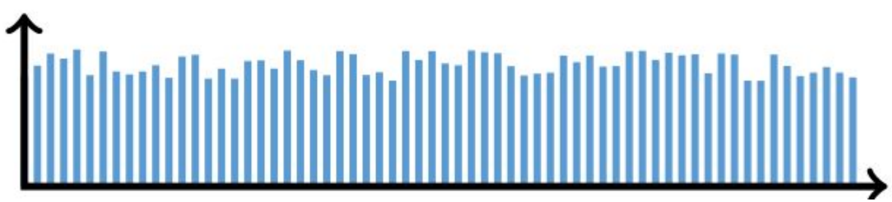
  Locality model

  - Process migrates from one locality to another
  - Localities may overlap
- Reasons for thrashing?

  Σ size of locality > total memory size
- Limit effects by using local or priority page replacement

Reasons for thrashing

Access pattern has no temporal **locality** (past ≠ future)

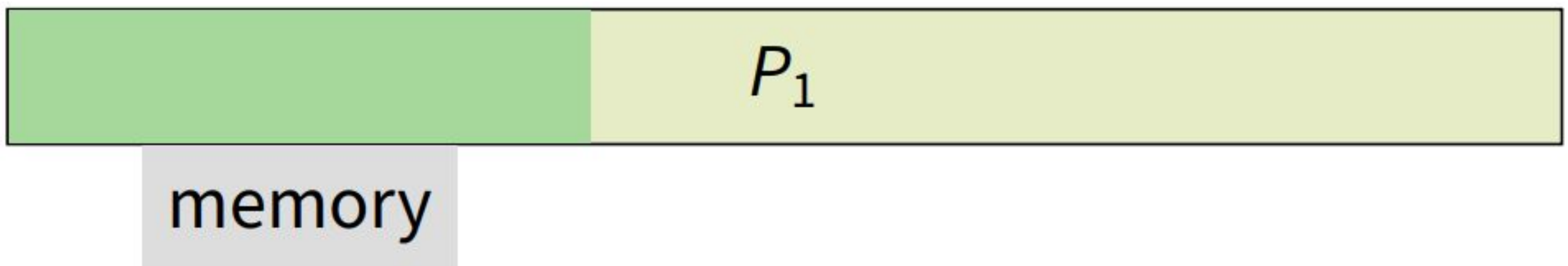


(80/20 rule has broken down)

https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

Reasons for thrashing

Hot memory does not fit in physical memory

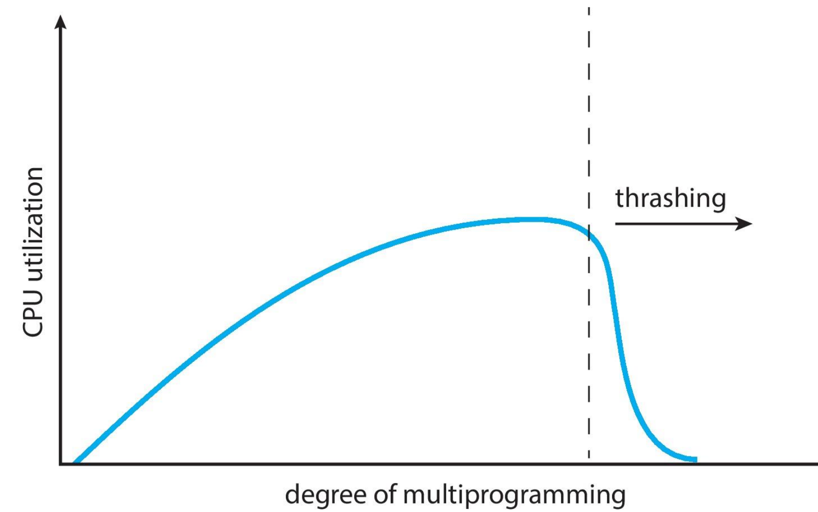


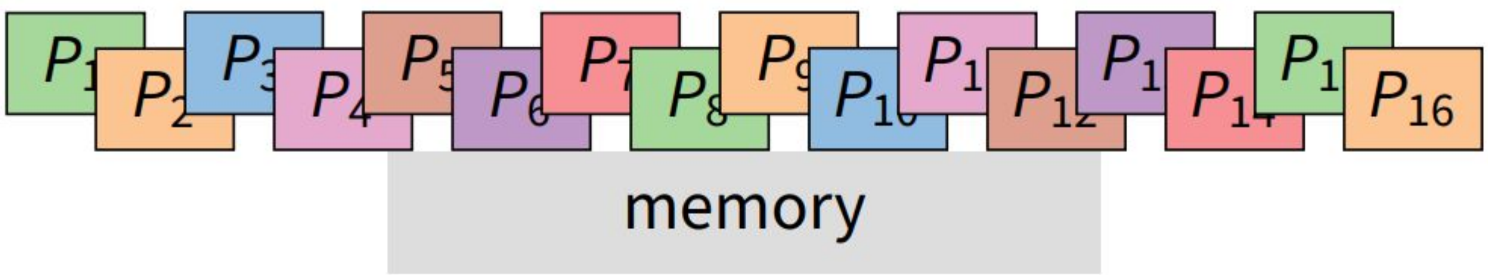
https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

Reasons for thrashing

Each process fits individually, but too many for system

- At least this case is possible to address





https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

Dealing with thrashing

Approach 1-working set model

Thrashing viewed from a caching perspective:

- given locality of reference,
- **how big a cache does the process need?**

Or: how much memory does the process need in order to make reasonable progress?

- **its working set?**

Only run processes whose memory requirements can be satisfied

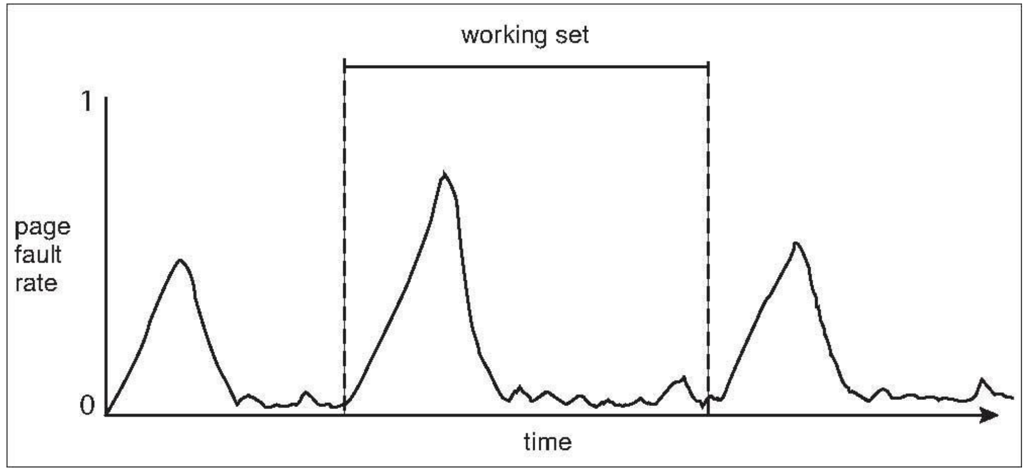
Approach 2: page fault frequency

Thrashing viewed as poor ratio of fetch to work

- PFF = page faults / instructions executed
- If PFF rises above threshold,
  - process needs more memory.
  - Not enough memory on the system? Swap out.
- If PFF sinks below threshold,
  - memory can be taken away

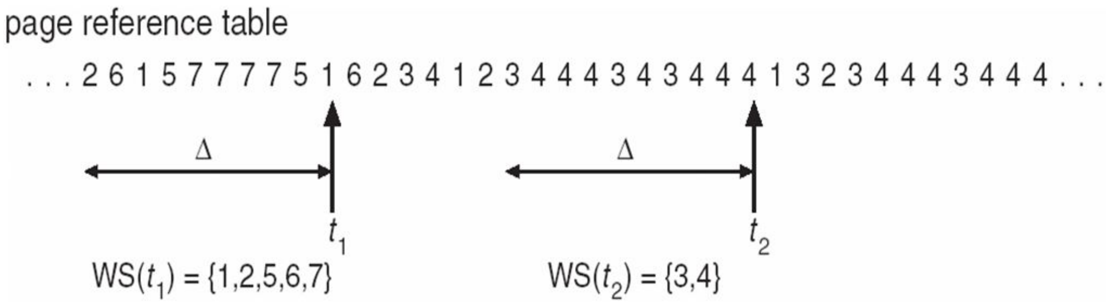
https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

Working Sets and Page Fault Rates



- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time

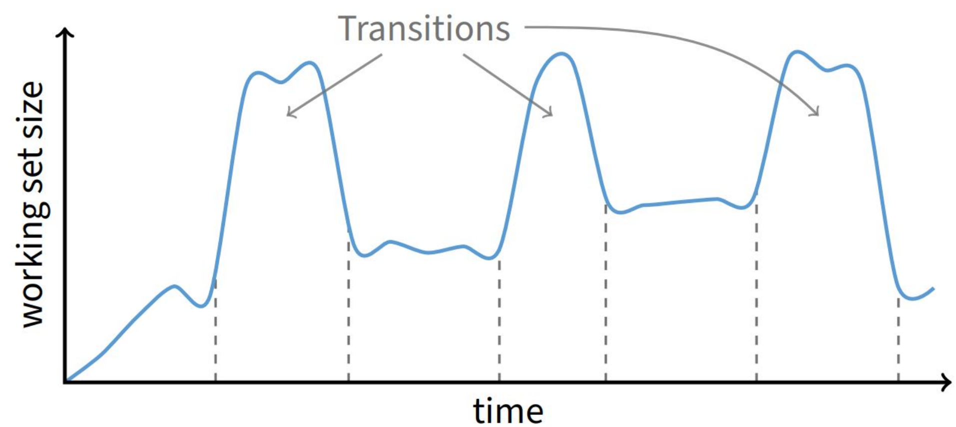
Working-Set Size



- D-demand
  - if $D > m \Rightarrow$ Thrashing
  - Policy if D > m,
  - then suspend or swap out one of the processes

working set size vs time

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  - Example: 10,000 instructions
- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma\, WSS_i \equiv$ total demand frames
  - Approximation of locality

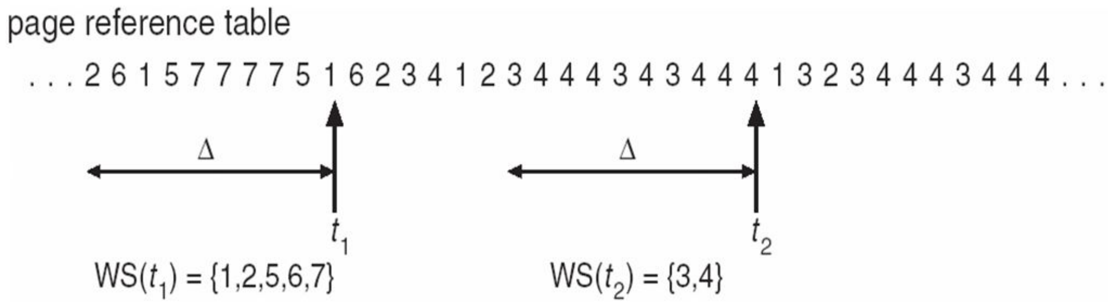


Working set changes across phases

- Baloons during phase transitions

https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

Calculating Working-Set Size



Working set: all pages that process will access in next T time

- **Can't calculate without predicting future**

Approximate by assuming past predicts future

- working set ≈ pages accessed in last T time
- Keep idle time for each page

Periodically scan all resident pages in system

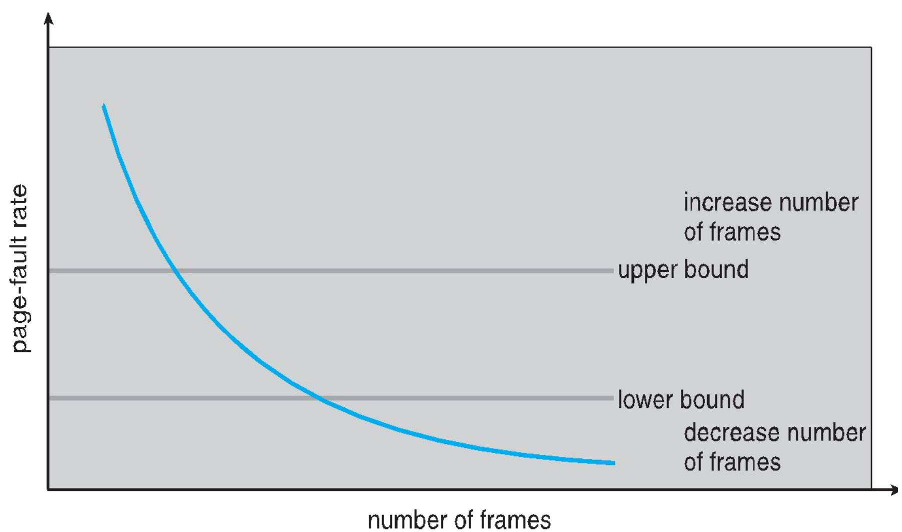
- **A** bit set? Clear it and clear the page's idle time
- **A** bit clear? Add CPU consumed since last scan to idle time
- Working set is pages with idle time < T

Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10{,}000$
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

Approach 2:Page-Fault Frequency

- More direct approach than WSS
- Establish “acceptable” **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame



Two-level scheduler

Divide processes into active & inactive

- Active – means working set resident in memory
  - **lower level scheduler:** choose among them to run
- Inactive – working set intentionally not loaded

Balance set: union of all active working sets

- Must keep balance set smaller than physical memory

Use long-term scheduler

- Moves procs active → inactive until balance set small enough
- Periodically allows inactive to become active
- As working set changes, must update balance set

Complications

- How to chose idle time threshold T?
- How to pick processes for active set
- How to count shared memory (e.g., libc.so)

https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

Some complications of paging

What happens to available memory?

- Some physical memory tied up by kernel VM structures

What happens to user/kernel crossings?

- More crossings into kernel
- Pointers in syscall arguments must be checked
    - can't just kill process if page not present—might need to page in

What happens to IPC?

- Must change hardware address space
- Increases TLB misses
- Context switch flushes TLB entirely on old x86 machines
    - Newer CPUs use more effective strategies marking which process an entry is for
    - Some CPUs have a process ID register
        - MIPS tags TLB entries with PID

What if you want a 64-bit virtual address space?

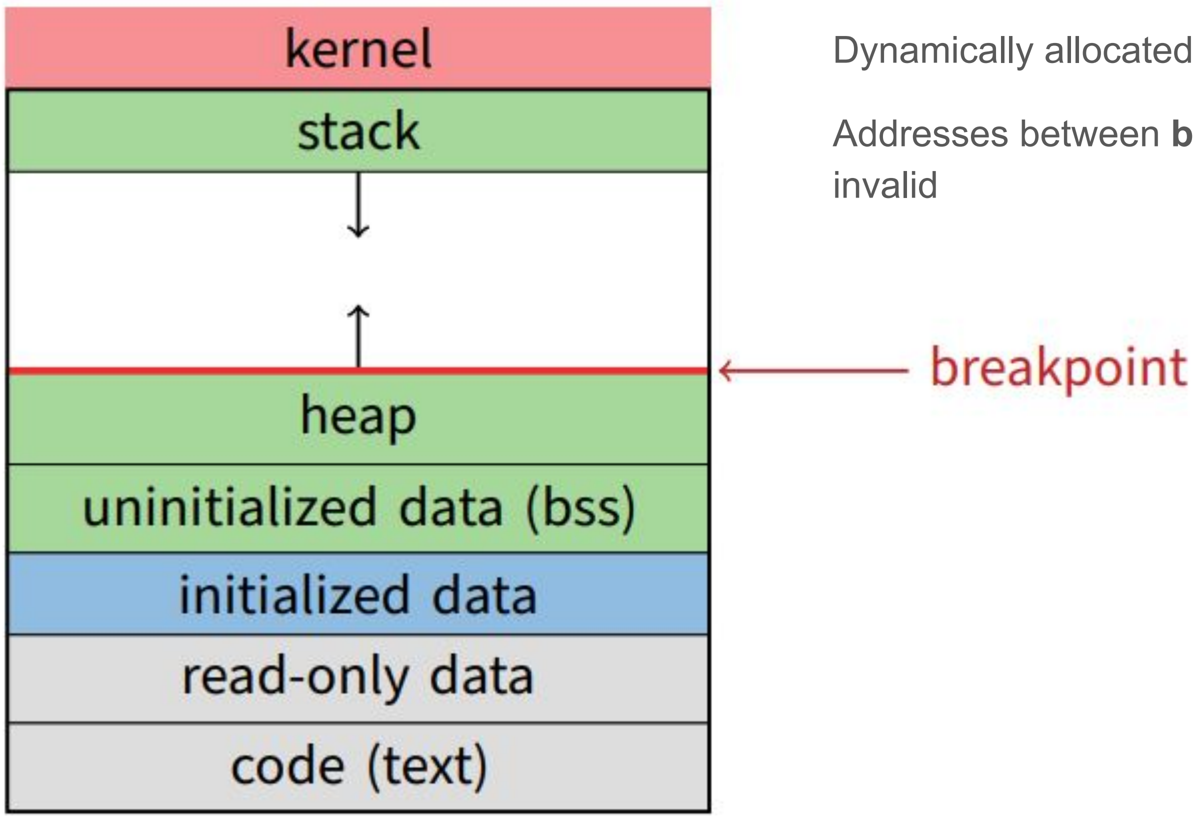
- Recall x86-64 only has 48-bit virtual address space
- Hashed page tables
- Guarded page tables
    - Omit intermediary tables with only one entry

https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

User-level perspective

From system programming

Recall virtual address space of a process

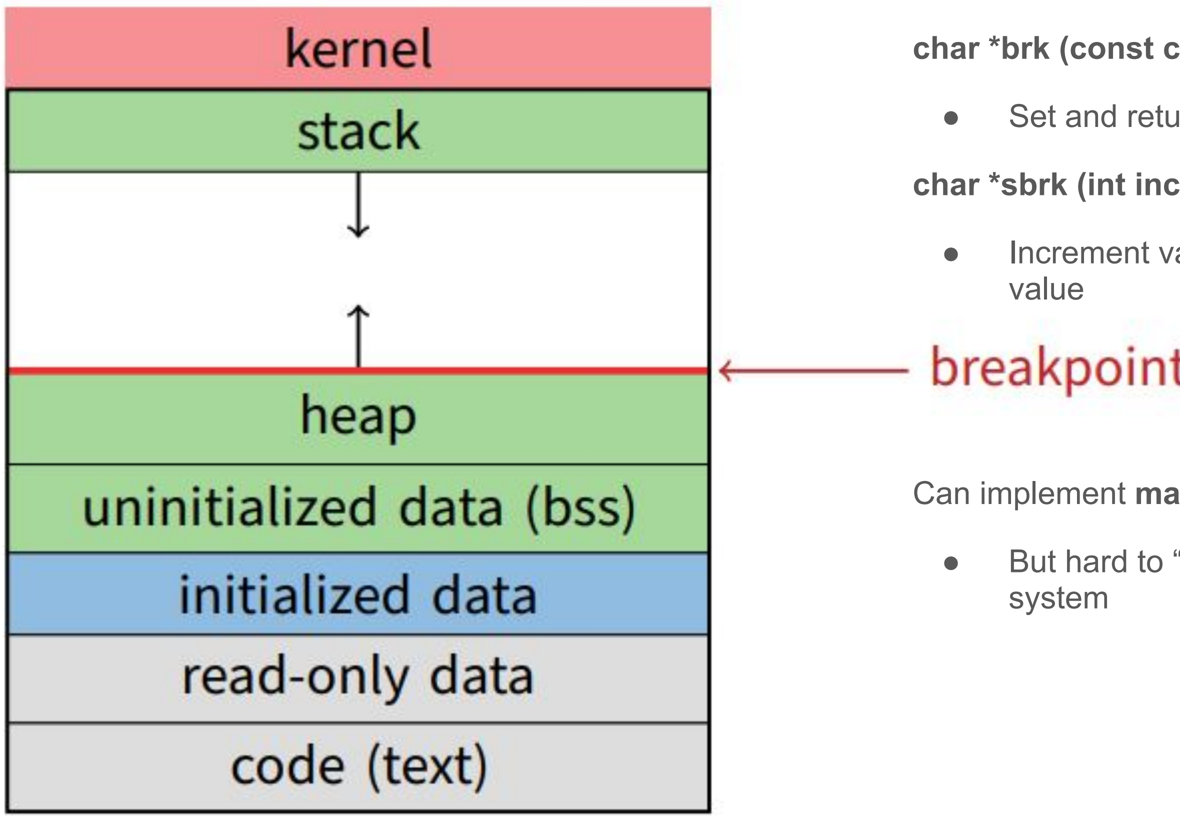


Dynamically allocated memory goes in heap

Addresses between **breakpoint** and stack all invalid

https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

Early system calls



OS keeps “Breakpoint” – top of heap

- Memory regions between breakpoint & stack fault on access

char *brk (const char *addr);

- Set and return new value of breakpoint

char *sbrk (int incr);

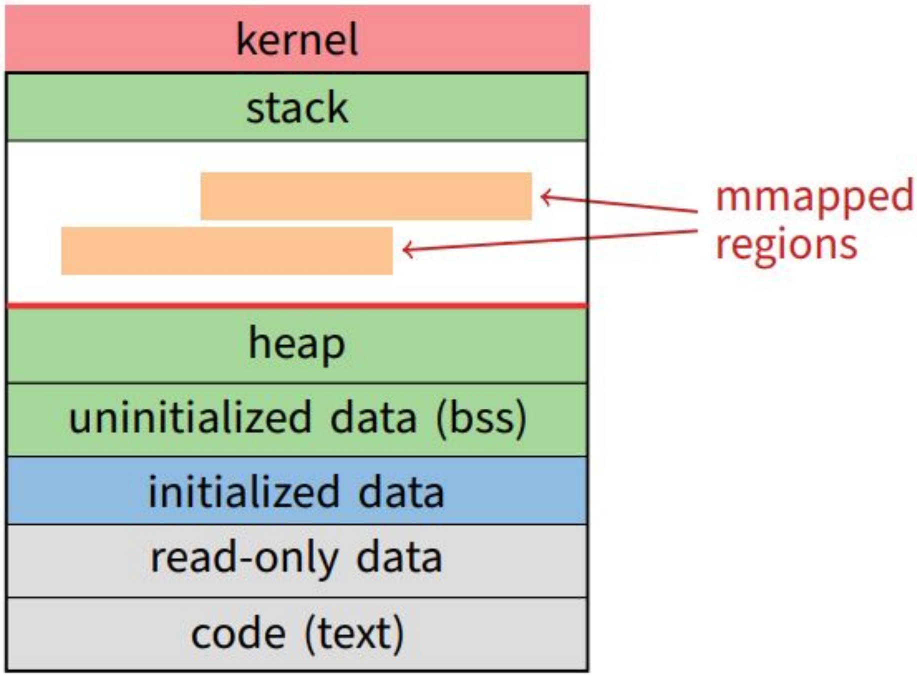
- Increment value of the breakpoint & return old value

Can implement **malloc** in terms of **sbrk**

- But hard to “give back” physical memory to system

https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

Memory mapped files and system calls



- Other memory objects between heap and stack

void *mmap (void *addr, size_t len, int prot, int flags, int fd, off_t offset)

- Map file specified by fd at virtual address addr
- If addr is NULL, let kernel choose the address

prot – protection of region

- OR of PROT_EXEC, PROT_READ, PROT_WRITE, PROT_NONE

flags

- MAP_ANON – anonymous memory (fd should be -1)
- MAP_PRIVATE – modifications are private
- MAP_SHARED – modifications seen by everyone

https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

int msync(void *addr, size_t len, int flags);

- Flush changes of mmapped file to backing store

int munmap(void *addr, size_t len);

- Removes memory-mapped object

int mprotect(void *addr, size_t len, int prot);

- Changes protection on pages to bitwise or of some PROT_. . . values

int mincore(void *addr, size_t len, char *vec)

- Returns in vec which pages present

https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

Exposing page faults

```
struct sigaction {
   union { /* signal handler */
       void (*sa_handler)(int);
       void (*sa_sigaction)
           (int, siginfo_t *, void *);
   };
   sigset_t sa_mask; /* signal mask to apply */
   int sa_flags;
};
int sigaction(int sig,
              const struct sigaction *act,
              struct sigaction *oact);
```

Can specify function to run on **SIGSEGV**

- Unix signal raised on invalid memory access
- this allows user-level context switching between multiple threads of control within a process.

https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

Example OpenBSD/i386 siginfo

```
struct sigcontext{
   int sc_gs;   int sc_fs;   int sc_es;   int sc_ds;   int sc_edi;
   int sc_esi;   int sc_ebp;   int sc_ebx;   int sc_edx;   int sc_ecx;
   int sc_eax;   int sc_eip;
   int sc_cs;     /* instruction pointer */
   int sc_eflags; /* condition codes, etc. */
   int sc_esp;
   int sc_ss;      /* stack pointer */
   int sc_onstack; /* sigstack state to restore */
   int sc_mask;    /* signal mask to restore */
   int sc_trapno;
   int sc_err;
};
```

Linux uses ucontext_t – same idea, just uses nested structures that won't all fit on one slide

https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

VM tricks at user level

Combination of mprotect/sigaction very powerful

- Can use OS VM tricks in user-level programs [Virtual memory primitives for user programs]

- E.g., fault, unprotect page, return from signal handler

Technique used in object-oriented databases

- Bring in objects on demand

- Keep track of which objects may be dirty

- Manage memory as a cache for much larger object DB

Other interesting applications

- Useful for some garbage collection algorithms

- Snapshot processes (copy on write)

https://www.scs.stanford.edu/24wi-cs212/notes/vm_os.pdf

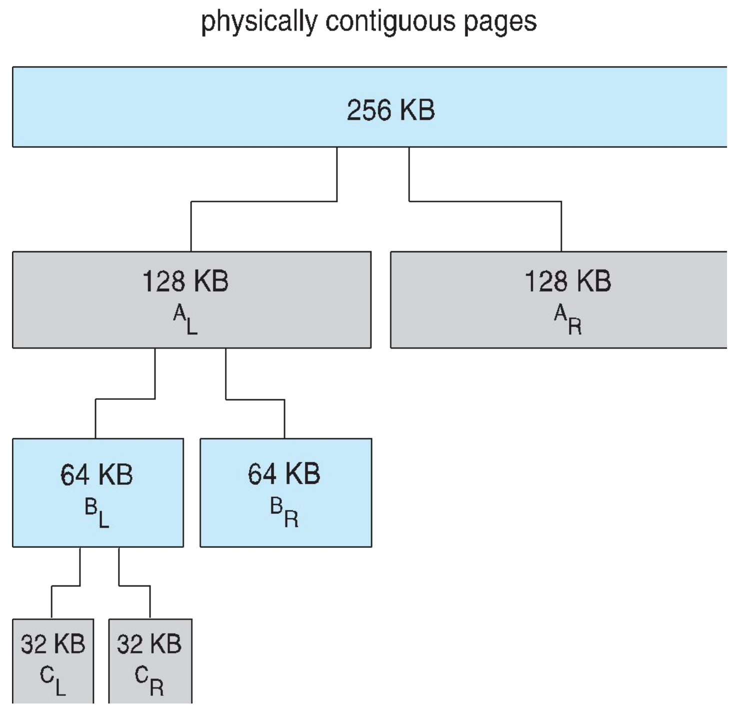
Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - i.e., for device I/O

Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available
- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_R of 128KB each
    - One further divided into B_L and B_R of 64KB
      - One further into C_L and C_R of 32KB each – one used to satisfy request
- Advantage – quickly **coalesce** unused chunks into larger chunk
- Disadvantage - fragmentation

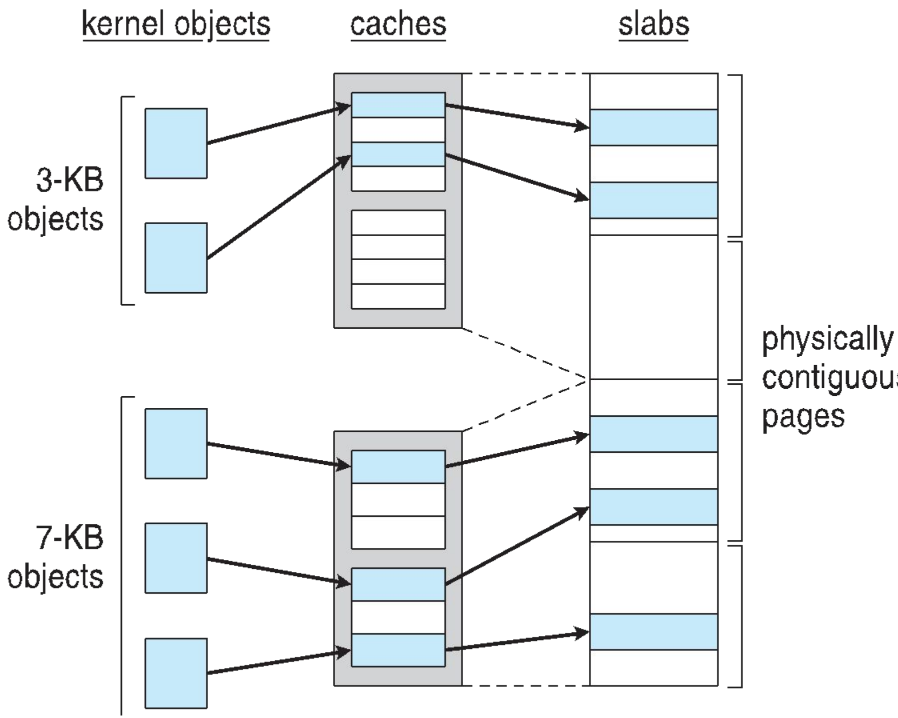
Buddy System Allocator



Slab Allocator

- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **`free`**
- When structures stored, objects marked as **`used`**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation



Slab Allocator in Linux

- For example process descriptor is of type `struct task_struct`
- Approx 1.7KB of memory
- New task -> allocate new struct from cache
  - Will use existing free `struct task_struct`
- Slab can be in three possible states
  1. Full – all used
  2. Empty – all free
  3. Partial – mix of free and used
- Upon request, slab allocator
  1. Uses free struct in partial slab
  2. If none, takes one from empty slab
  3. If no empty slab, create new empty

Slab Allocator in Linux (Cont.)

- Slab started in Solaris, now wide-spread for both kernel mode and user memory in various OSes
- Linux 2.2 had SLAB, now has both SLOB and SLUB allocators
  - SLOB for systems with limited memory
    - Simple List of Blocks – maintains 3 list objects for small, medium, large objects
  - SLUB is performance-optimized SLAB removes per-CPU queues, metadata stored in page structure

Other Considerations

- Prepaging
- Page size
- TLB reach
- Inverted page table
- Program structure
- I/O interlock and page locking

Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume s pages are prepaged and α of the pages is used
  - Is cost of $\boldsymbol{s * \alpha}$ save pages faults > or < than the cost of prepaging $\boldsymbol{s * (1- \alpha)}$ unnecessary pages?
  - $\boldsymbol{\alpha}$ near zero $\Rightarrow$ prepaging loses

Page Size

- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - **Resolution**
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time

TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults
- Increase the Page Size
  - This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation

Program Structure

- Program structure
  - `int[128,128] data;`
  - Each row is stored in one page
  - Program 1

```
for (j = 0; j <128; j++)
    for (i = 0; i < 128; i++)
        data[i,j] = 0;
```

128 x 128 = 16,384 page faults

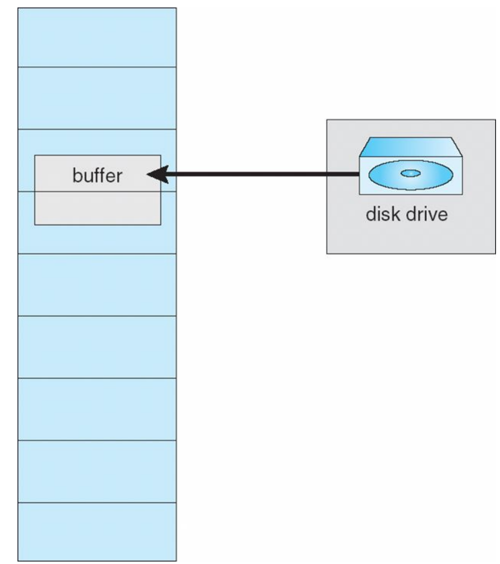
  - Program 2

```
for (i = 0; i < 128; i++)
    for (j = 0; j < 128; j++)
        data[i,j] = 0;
```

128 page faults

I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
- **Pinning** of pages to lock into memory



Operating System Implementations

- Solaris
- Linux

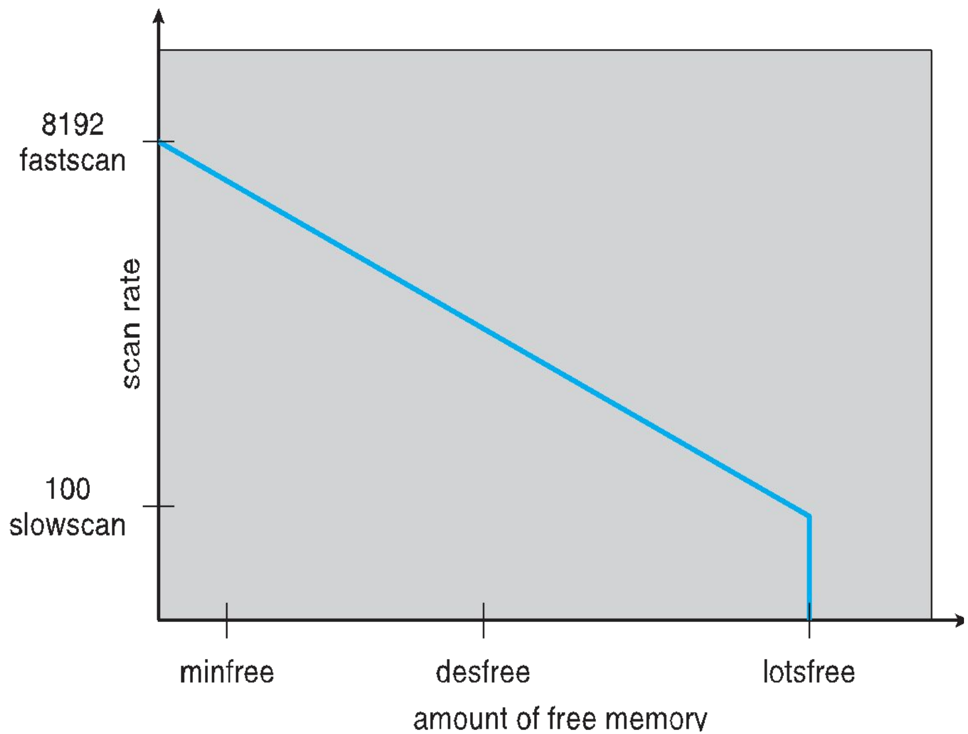
Solaris

- The page scanner thread runs and begins to walk through memory. A two-step algorithm is employed:
  - A page is marked as unused.
  - If still unused after a time interval, the page is viewed as a subject for reclaim

Paging-Related Parameters - Oracle Solaris

- **`Lotsfree`** – threshold parameter (amount of free memory) to begin paging
- **`Desfree`** – threshold parameter to increasing paging
- **`Minfree`** – threshold parameter to being swapping
- Paging is performed by **`pageout`** process
- **`Pageout`** scans pages using modified clock algorithm
- **`Scanrate`** is the rate at which pages are scanned. This ranges from **`slowscan`** to **`fastscan`**
- **`Pageout`** is called more frequently depending upon the amount of free memory available
- ***Priority paging*** gives priority to process code pages

Solaris 2 Page Scanner



Linux

```
+-----+
| PGD |
+-----+
   |
   |   +-----+
   +-->| P4D |
       +-----+
          |
          |   +-----+
          +-->| PUD |
              +-----+
                 |
                 |   +-----+
                 +-->| PMD |
                     +-----+
                        |
                        |   +-----+
                        +-->| PTE |
                            +-----+
```

[Page Tables — The Linux Kernel documentation](#)
[Memory mapping — The Linux Kernel documentation](#)

Each process(struct task_struct) has a pointer **(mm_struct→pgd)** to its own *Page Global Directory (PGD)* which is a physical page frame.

pgd, pgd_t, pgdval_t = **Page Global Directory**

p4d, p4d_t, p4dval_t = **Page Level 4 Directory**

pud, pud_t, pudval_t = **Page Upper Directory**

pmd, pmd_t, pmdval_t = **Page Middle Directory**

pte, pte_t, pteval_t = **Page Table Entry**

Each is array of pointers

```
     PMD
--> +-----+           PTE
    | ptr |-------> +-----+
    | ptr |-        | ptr |-------> PAGE
    | ptr | \       | ptr |
    | ptr |  \        ...
    | ... |   \
    | ptr |    \         PTE
    +-----+     +----> +-----+
                       | ptr |-------> PAGE
                       | ptr |
                         ...
```

Linux

`struct mm_struct` encompasses all memory areas associated with a process.

`struct page` is used to embed information about all physical pages in the system.

The kernel has a `struct page` structure for all pages in the system

```
struct page * alloc_page(unsigned int gfp_mask)
```

`struct vm_area_struct` holds information about a contiguous virtual memory area.

A `struct vm_area_struct` is created at each `mmap()` call issued from user space.

There are also functions for each struct

It implements LRU

There is also multigen_lru

Multi-Gen LRU — The Linux Kernel documentation

Concepts overview — The Linux Kernel documentation
Memory mapping — The Linux Kernel documentation
Memory Allocation Guide — The Linux Kernel documentation