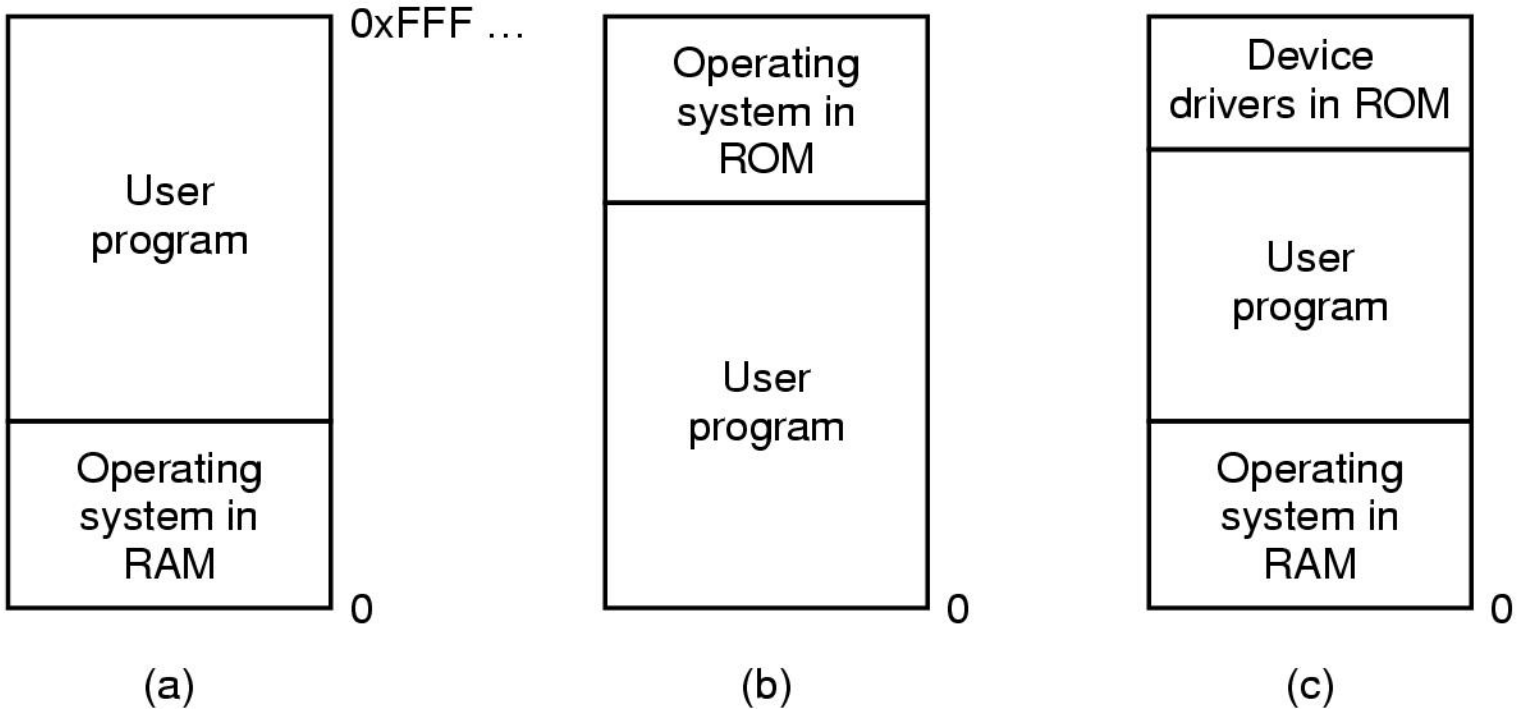
Memory Management

Instructor: Dr. Liting Hu

Memory Management Basics

- Don’t have infinite RAM
- Do have a memory hierarchy-
  - Cache (fast)
  - Main(medium)
  - Disk(slow)
- Memory manager has the job of using this hierarchy to create an **abstraction** (illusion) of easily accessible memory

One program at a time in memory



OS reads program in from disk and it is executed

One program at a time
Really want to run more than one program

- Can **only** have one program in memory at a time.
- Could swap new program into memory from disk and send old one out to disk
- **Not really concurrent**

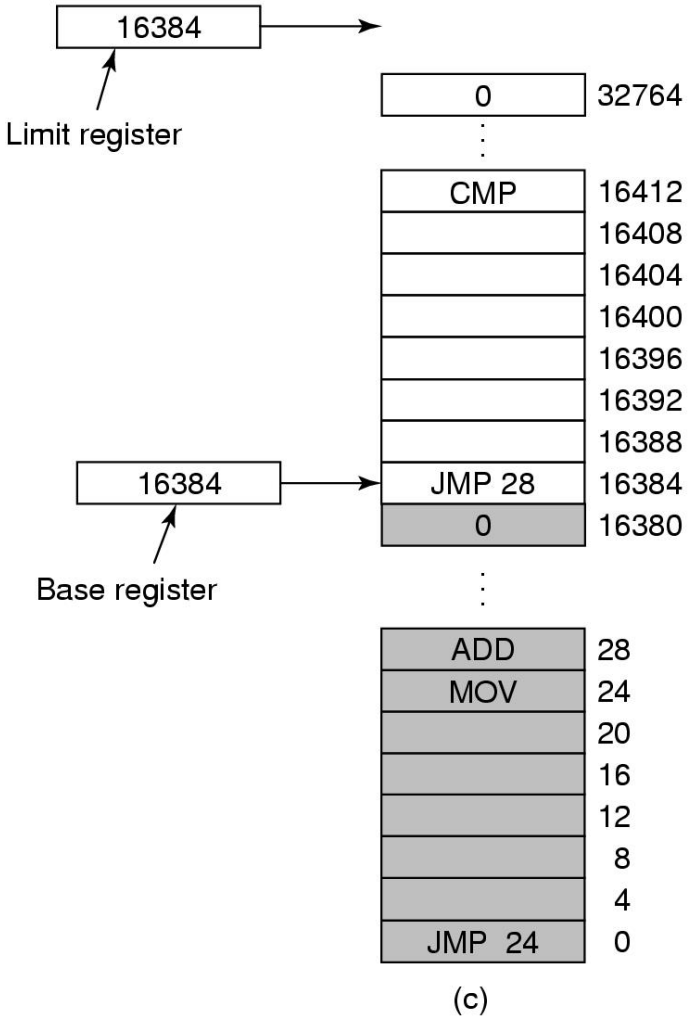
Address Space

- Create abstract memory space for program to exist in
  - Each program has its own set of addresses
  - The addresses are different for each program
  - Call it the address space of the program

Base and Limit Registers

- A form of dynamic relocation
  - Base contains beginning address of program
  - Limit contains length of program
  - Program references memory, adds base address to address generated by process. Checks to see if address is larger then limit. If so, generates fault
- Disadvantage - addition and comparison have to be done on every instruction

Base and Limit Registers



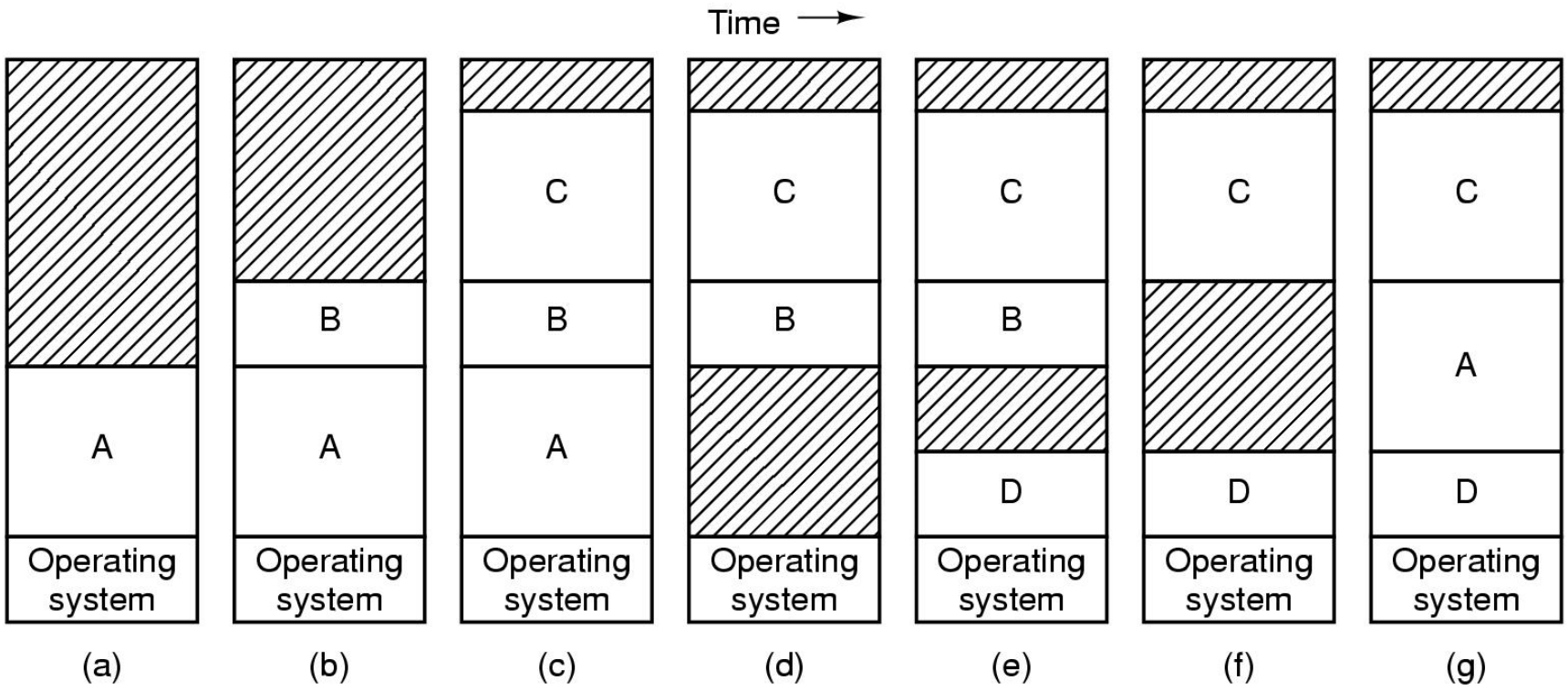
Add 16384 to JMP 28.

Hardware adds 16384 to 28 resulting in JMP 16412

How to run more programs then fit in main memory at once

- Can’t keep all processes in main memory
  - Too many (hundreds)
  - Too big (e.g. 200 MB program)
- Two approaches
  - Swap - bring program in and run it for awhile
  - Virtual memory - allow program to run even if only part of it is in main memory

Swapping, a picture



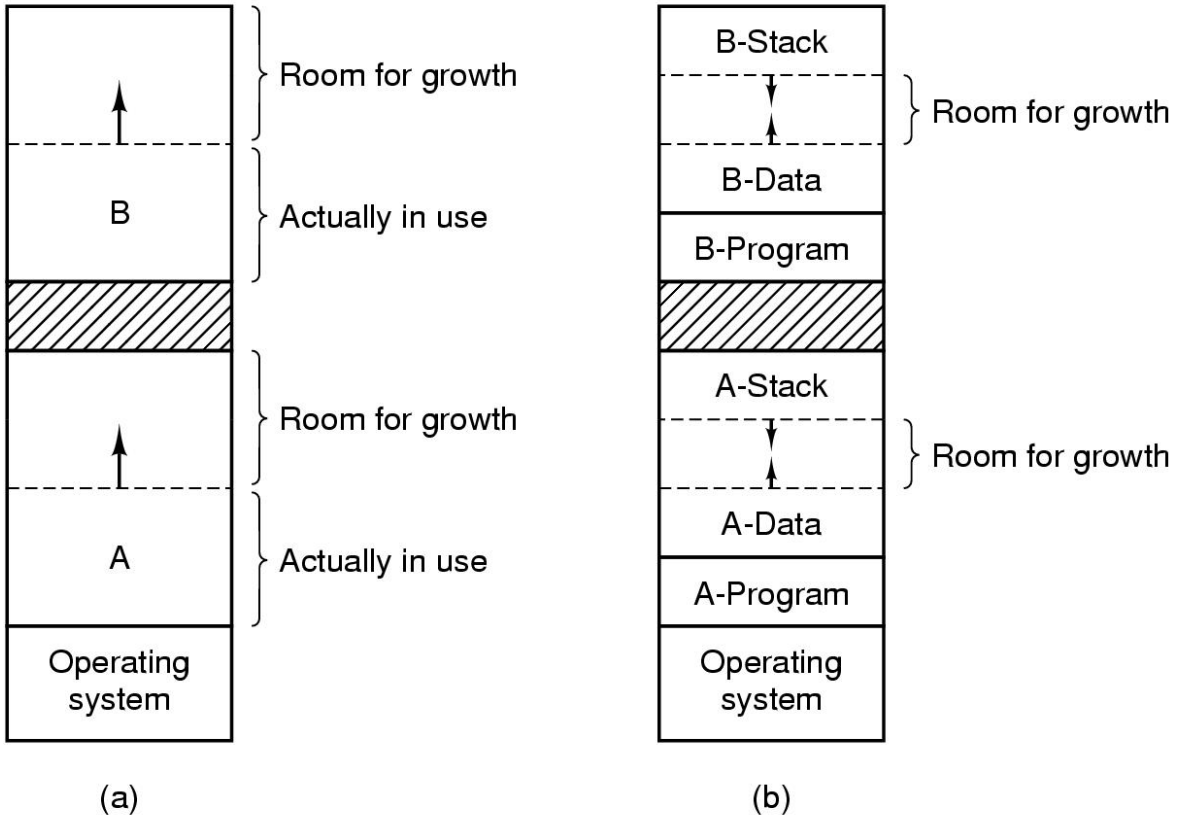
Can compact holes by copying programs into holes

This takes too much time

Programs grow as they execute

- Stack (return addresses and local variables)
- Data segment (heap for variables which are dynamically allocated and released)
- Good idea to allocate extra memory for both
- When program goes back to disk, don’t bring holes along with it!!!

2 ways to allocate space for growth



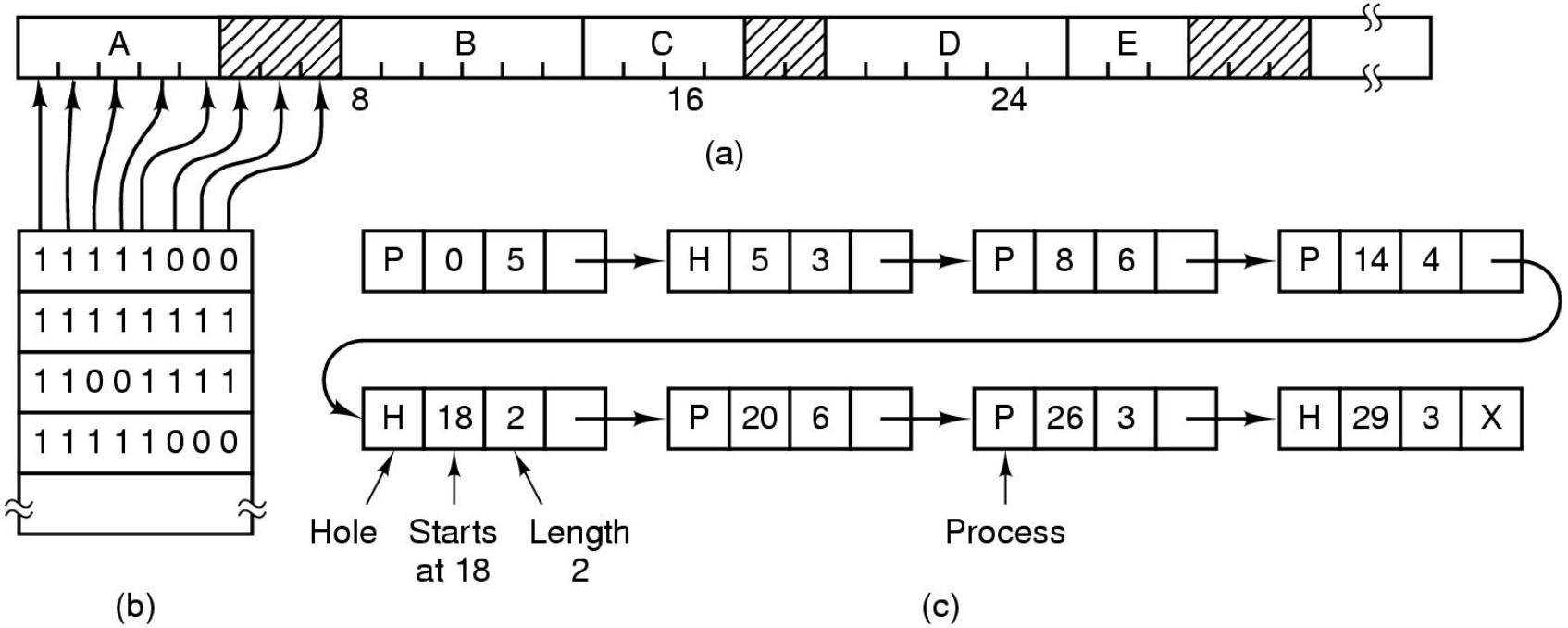
(a) Just add extra space

(b) Stack grows downwards, data grows upwards

Managing Free Memory

- Two techniques to keep track of free memory
  - Bitmaps
  - Linked lists

Bitmaps-the picture



(a)Picture of memory

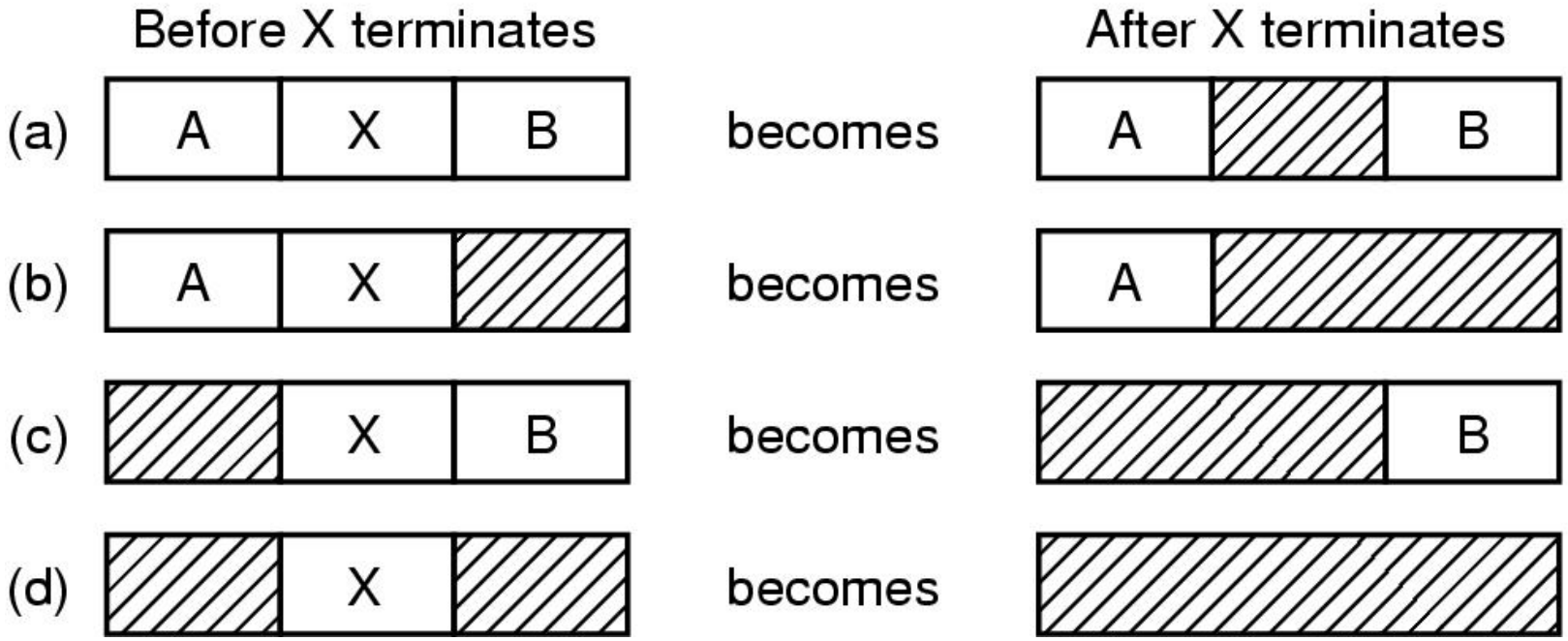
(b)Each bit in bitmap corresponds to a unit of storage (e.g. bytes) in memory

(c) Linked list P: process H: hole

Bitmaps

- The good - compact way to keep tract of memory
- The bad - need to search memory for k consecutive zeros to bring in a file k units long
  - Units can be bits or bytes or…….

Linked Lists-the picture



Four neighbor combinations for the terminating process, X.

Linked Lists

- Might want to use doubly linked lists to merge holes more easily
- Algorithms to fill in the holes in memory
  - Next fit
  - Best fit
  - Worst fit
  - Quick fit

The fits

- First fit - fast
- Next fit - starts search wherever it is
  - Slightly worse
- Best fit - smallest hole that fits
  - Slower, results in a bunch of wasted holes (i.e. worse algorithm)
- Worst fit - largest hole that fits
  - Not good (simulation results)
- Quick fit - keep list of common sizes
  - Quick, but can't find neighbors to merge with

Virtual Memory-the history

- Keep multiple parts of programs in memory
- Swapping is too slow (100 Mbytes/sec disk transfer rate=>10 sec to swap out a 1 Gbyte program)
- Overlays-programmer breaks program into pieces which are swapped in by overlay manager
  - Ancient idea - not really done
  - **Too hard to do - programmer has to break up program**

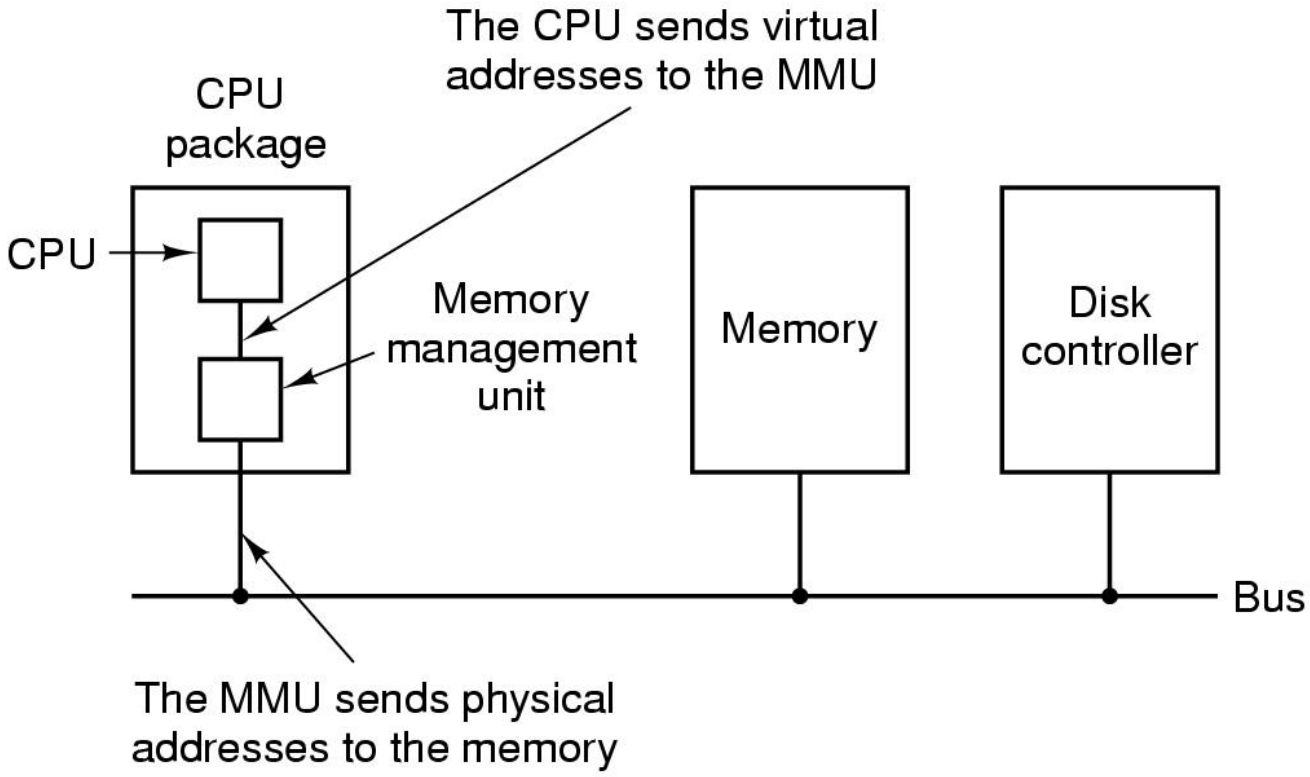
Virtual Memory

- Program's address space is broken up into fixed size pages
- Pages are mapped to physical memory
  - If instruction refers to a page in memory, fine
  - Otherwise OS gets the page, reads it in, and re-starts the instruction
  - While page is being read in, another process gets the CPU

Memory Management Unit

- Memory Management Unit generates physical address from virtual address provided by the program

Memory Management Unit



MMU maps virtual addresses to physical addresses and puts them on memory bus

Pages and Page Frames

- Virtual addresses divided into pages
  - 512 bytes-64 KB range
  - Transfer between RAM and disk is in whole pages
  - Example on next slide

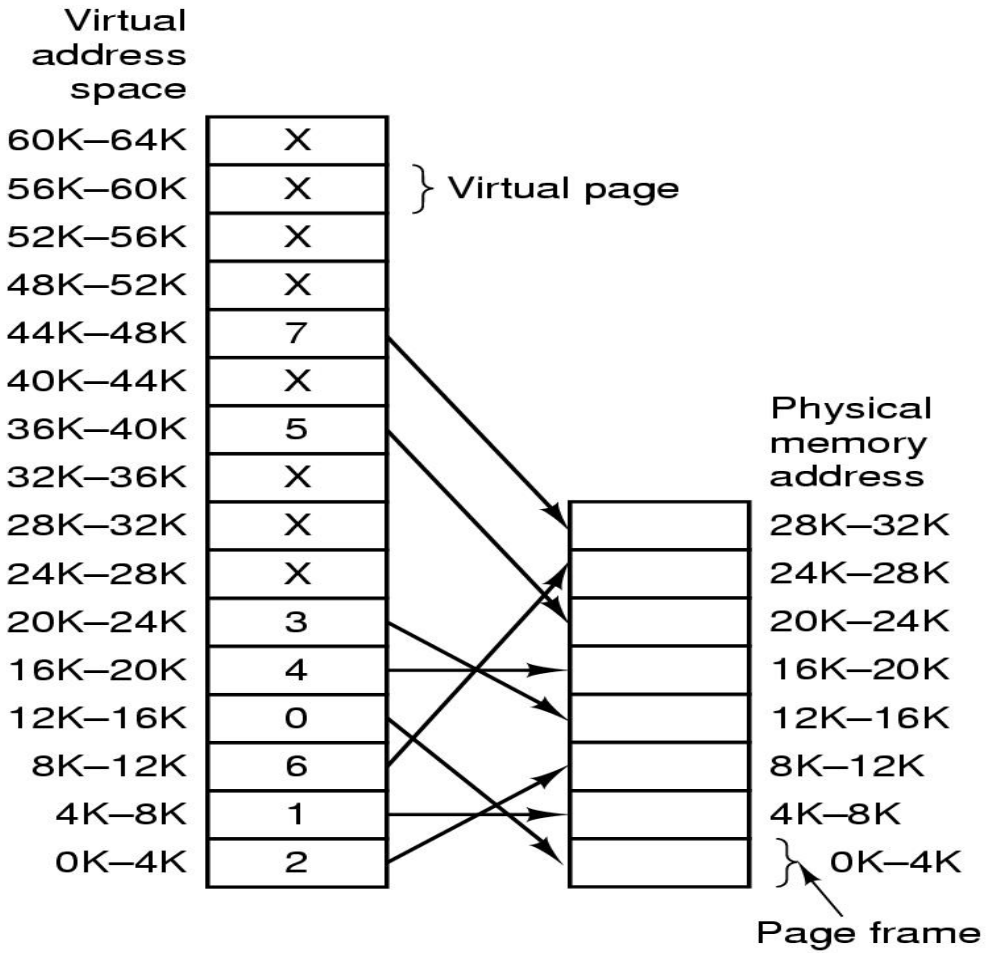
Mapping of pages to page frames

16 bit addresses,

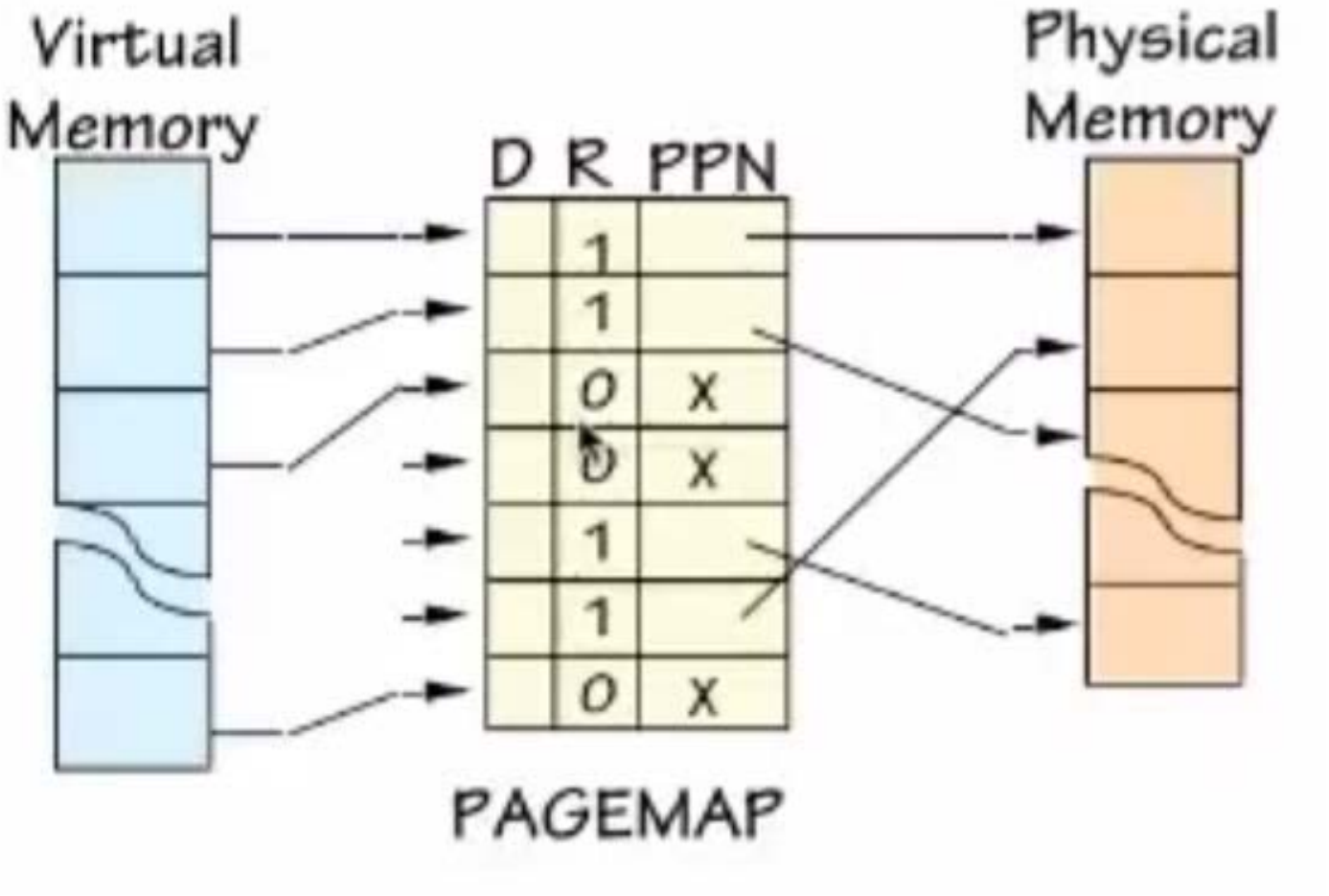
4 KB pages,

32 KB physical memory,

16 virtual pages and page frames



Mapping of pages to page frames



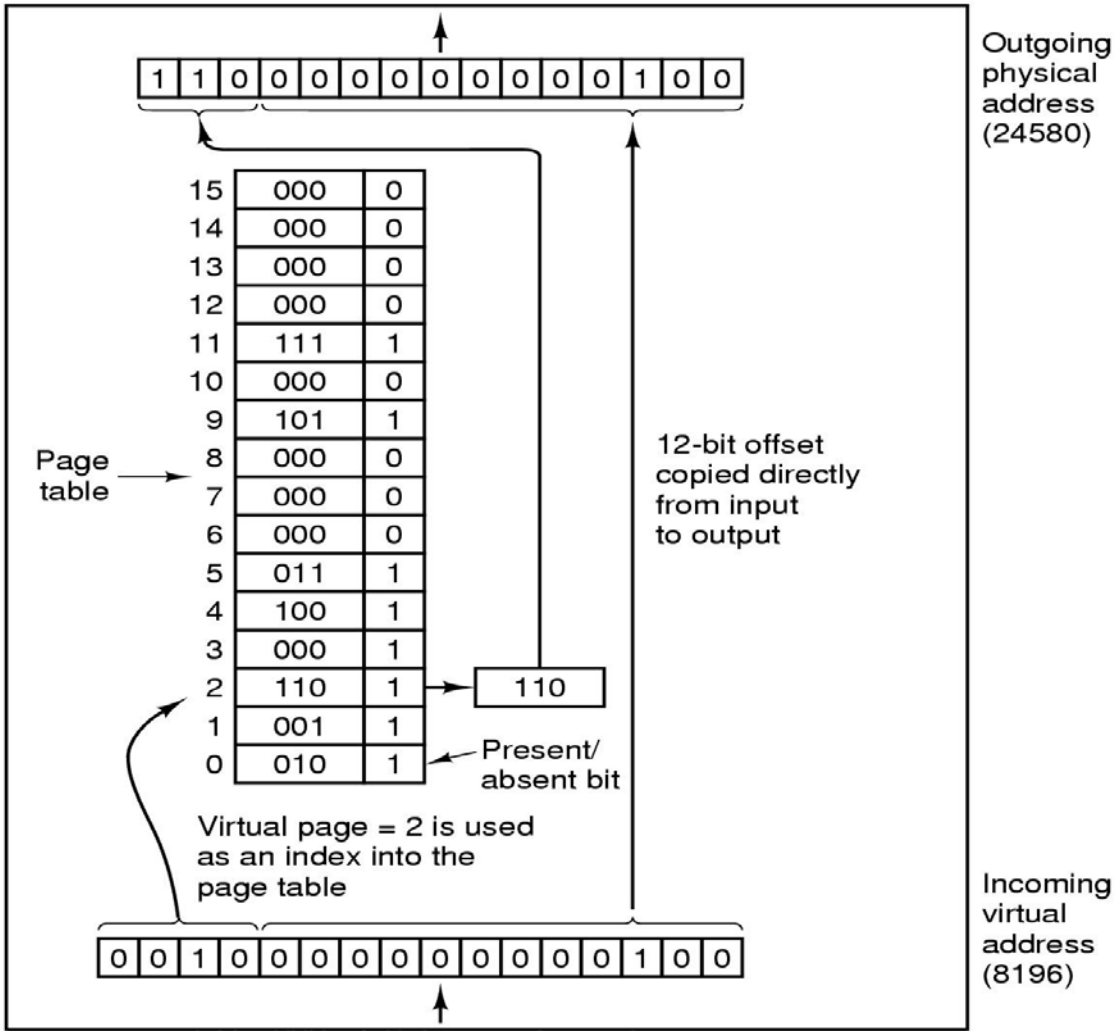
Page Fault Processing

- Present/absent bit tells whether page is in memory
- What happens If address is not in memory?
- Trap to the OS
  - OS picks page to write to disk
  - Brings page with (needed) address into memory
  - Re-starts instruction

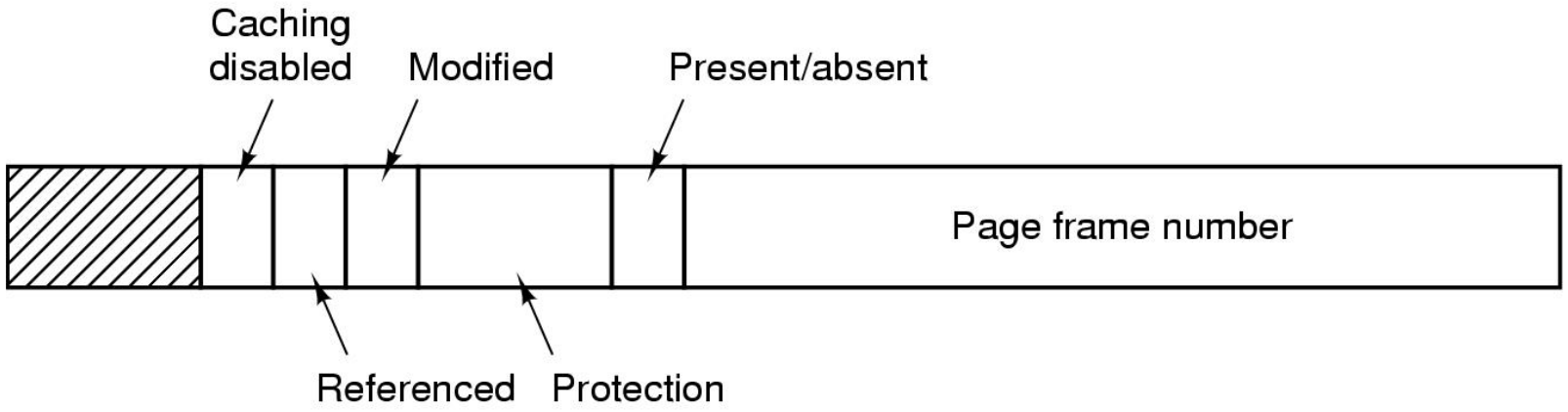
Page Table

- Virtual address={virtual page number, offset}
- Virtual page number used to index into page table to find page frame number
- If present/absent bit is set to 1, attach page frame number to the front of the offset, creating the physical address, which is sent on the memory bus

MMU operation



Structure of Page Table Entry



- Modified (dirty) bit: 1 means written to => have to write it to disk. 0 means don't have to write to disk.
- Referenced bit: 1 means it was either read or written. Used to pick page to evict. Don't want to get rid of page which is being used.
- Present (1) / Absent (0) bit
- Protection bits: r, w, r/w

Problems for paging

- Virtual to physical mapping is done on every memory reference => mapping must be fast
- If the virtual address space is large, the page table will be large. 32 bit addresses now and 64 bits becoming more common

Speed up Address Translation

- Most programs access a small number of pages a great deal
- Add Translation Lookaside Buffer (TLB) to MMU
  - Stores frequently accessed frames

Translation Lookaside Buffers

Valid	Virtual page	Modified	Protection	Page frame
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	R X	50
1	21	0	R X	45
1	860	1	RW	14
1	861	1	RW	75

Valid bit indicates whether page is in use or not

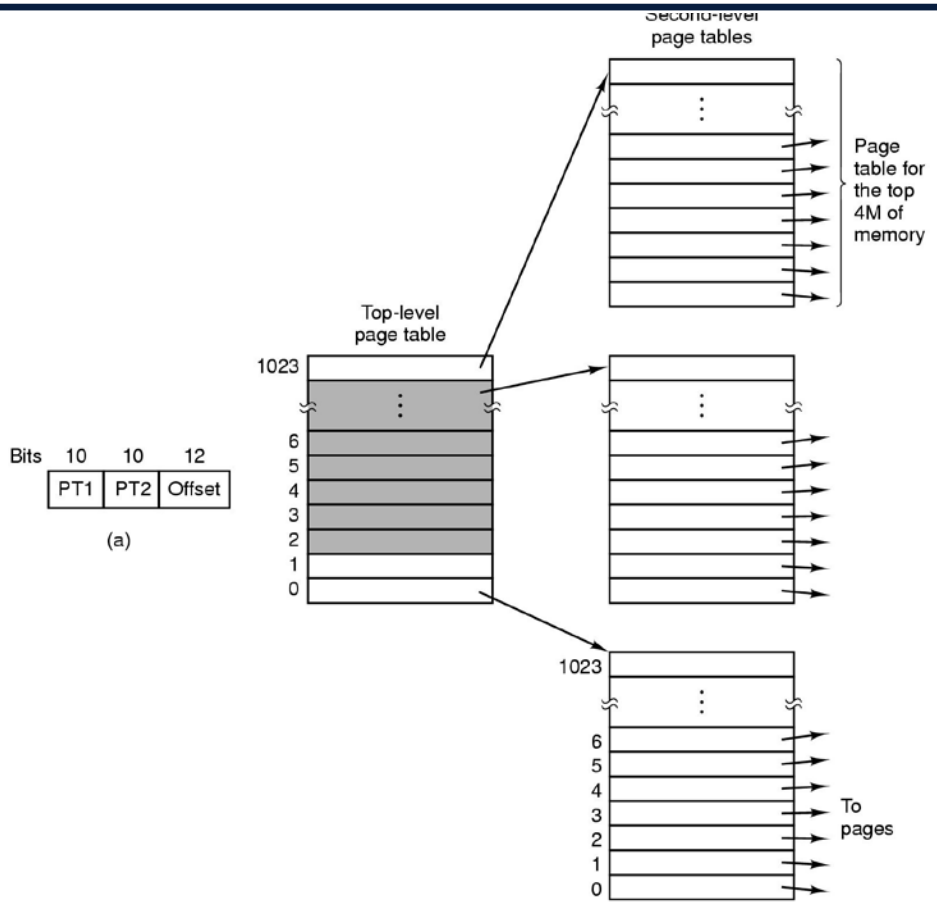
Translation Lookaside Buffer (TLB)

- If address is in MMU, avoid page table
- Uses parallel search to see if virtual page is in the TLB
- If not, does page table look up and evicts TLB entry, replacing it with page just looked up

Multi-level page tables

- Want to avoid keeping the entire page table in memory because it is too big
- Want to avoid keeping the entire page table in memory because it is too big
- The hierarchy is **a page table of page tables**

Multilevel Page Tables



(a) A 32-bit address with two page table fields.
(b) Two-level page tables.

Let’s See it as a Whole

- https://www.youtube.com/watch?v=knDdYTpp0h8
- https://www.youtube.com/watch?v=eyxL03AVLOY

Page Replacement Algorithms

- If new page is brought in, need to chose a page to evict
- Don’t want to evict heavily used pages
- If page has been written to, need to copy it to disk.
- Otherwise, a good copy is on the disk=>can write over it

Page Replacement Algorithms - the Laundry List

- Optimal page replacement algorithm
- Not recently used page replacement
- First-in, first-out page replacement
- Second chance page replacement
- Clock page replacement
- Least recently used page replacement
- Working set page replacement
- WSClock page replacement

Optimal Page Replacement

- Pick the one which will not used before the longest time
- Not possible unless know when pages will be referenced
- Used as ideal reference algorithm

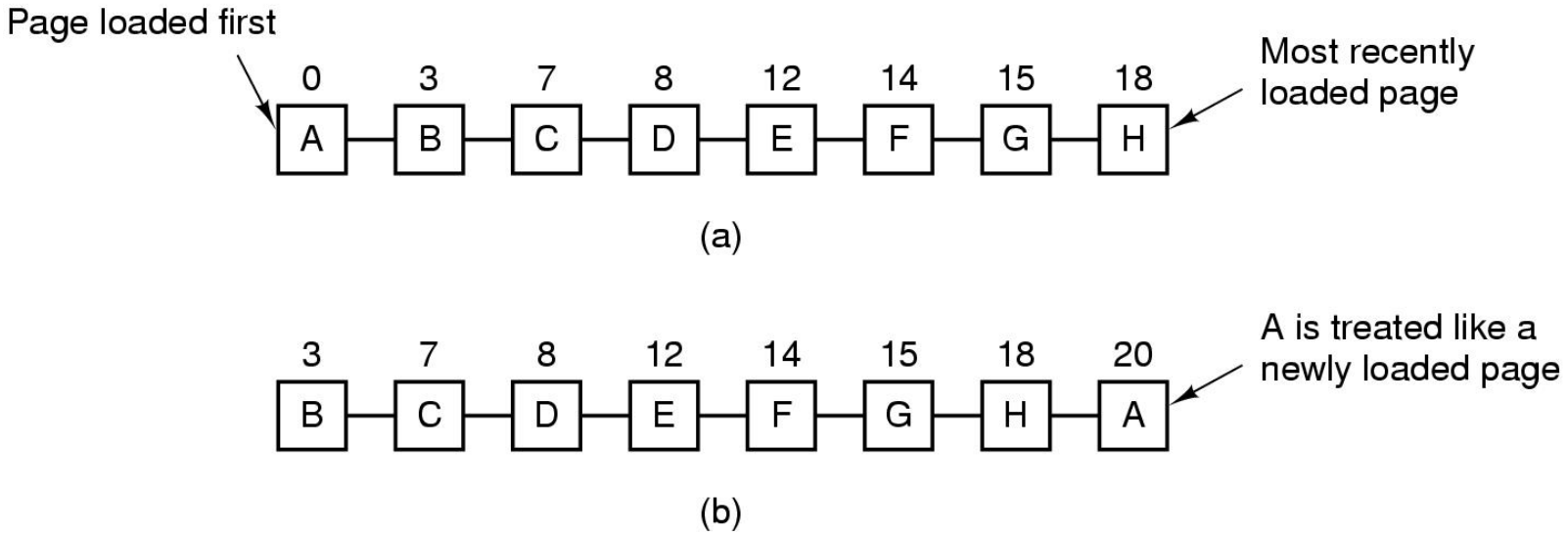
Not recently used

- Use R and M bits
- Periodically clear R bit
  - Class 0: not referenced, not modified
  - Class 1: not referenced, modified
  - Class 2: referenced, not modified
  - Class 3: referenced, modified
- Pick lowest priority page to evict

FIFO

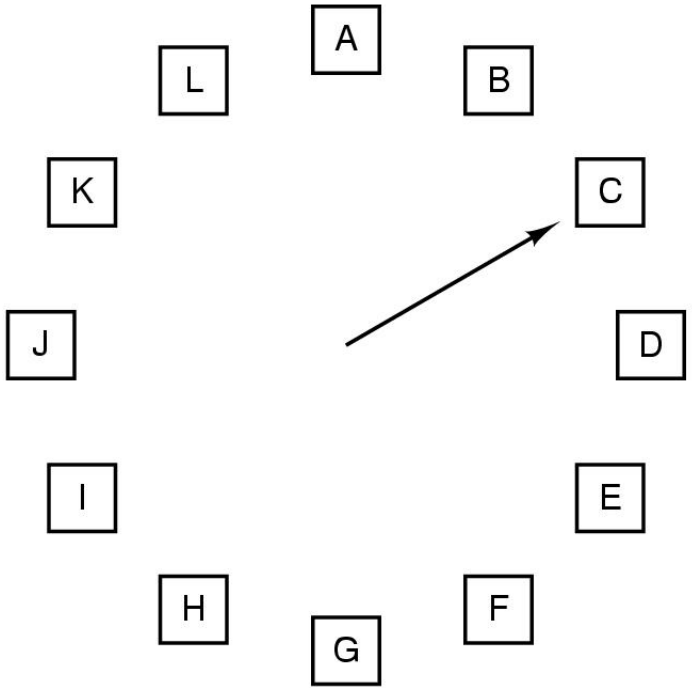
- Keep list ordered by time (latest to arrive at the end of the list)
- Evict the oldest, i.e. head of the line
- Easy to implement
- Oldest might be most heavily used! No knowledge of use is included in FIFO

Second Chance Algorithm



- Pages sorted in FIFO order by arrival time.
- Examine R bit. If zero, evict. If one, put page at end of list and R is set to zero.
- If change value of R bit frequently, might still evict a heavily used page

The Clock Page Replacement Algorithm



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

Clock

- Doesn’t use age as a reason to evict page
- Faster-doesn’t manipulate a list
- Doesn’t distinguish between how long pages have not been referenced

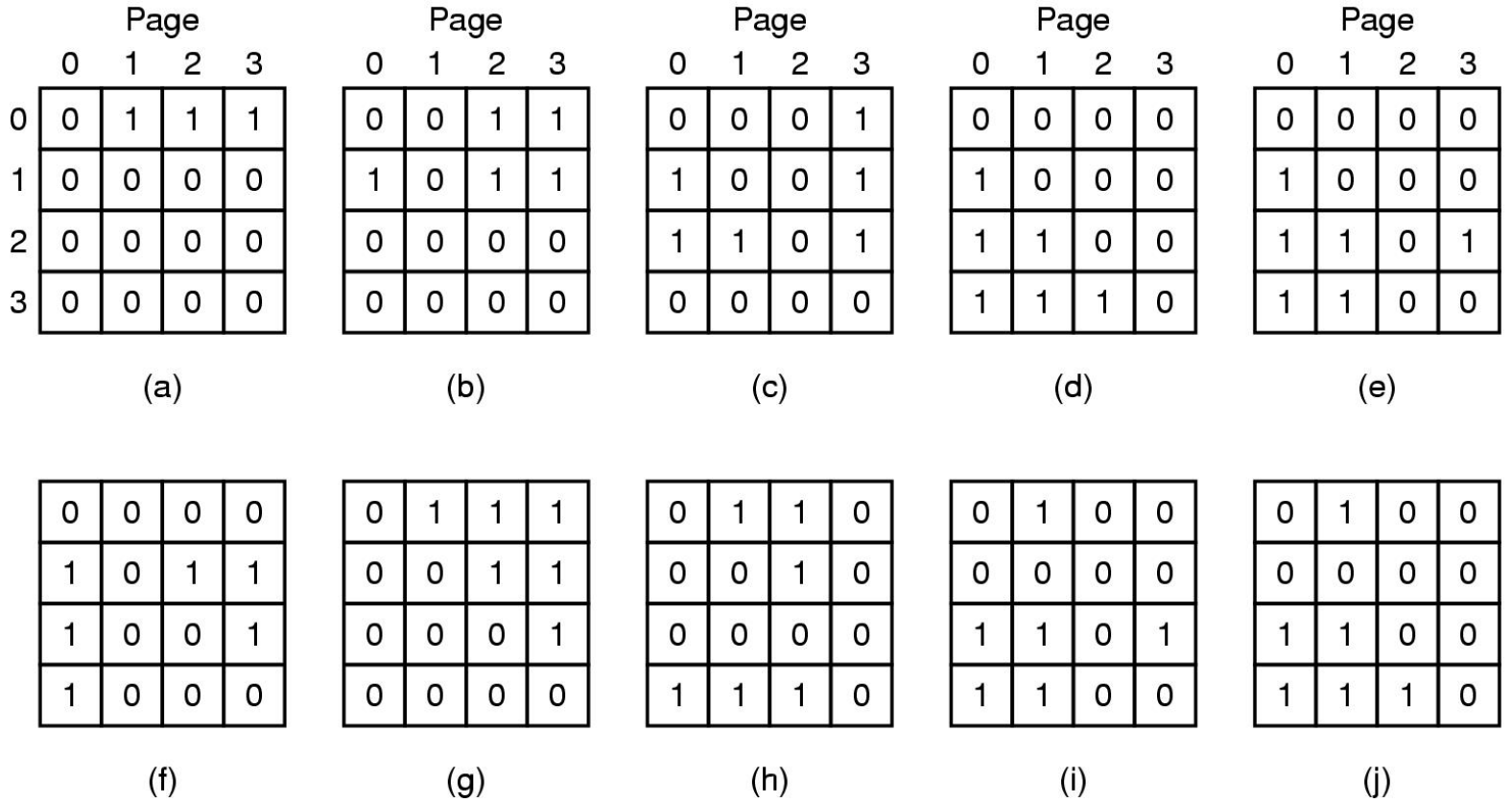
LRU

- Approximate LRU by assuming that recent page usage approximates long term page usage
- Could associate counters with each page and examine them but this is expensive

LRU-the hardware array

- Associate counter with each page.
  - At each reference increment counter.
  - Evict page with lowest counter
- Keep n x n array for n pages. Upon reference page k, put 1's in row k and 0's in column k.
  - Row with smallest binary value corresponds to LRU page. Evict k!
  - Easy hardware implementation

LRU-hardware



LRU using a matrix when pages are referenced in the order 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

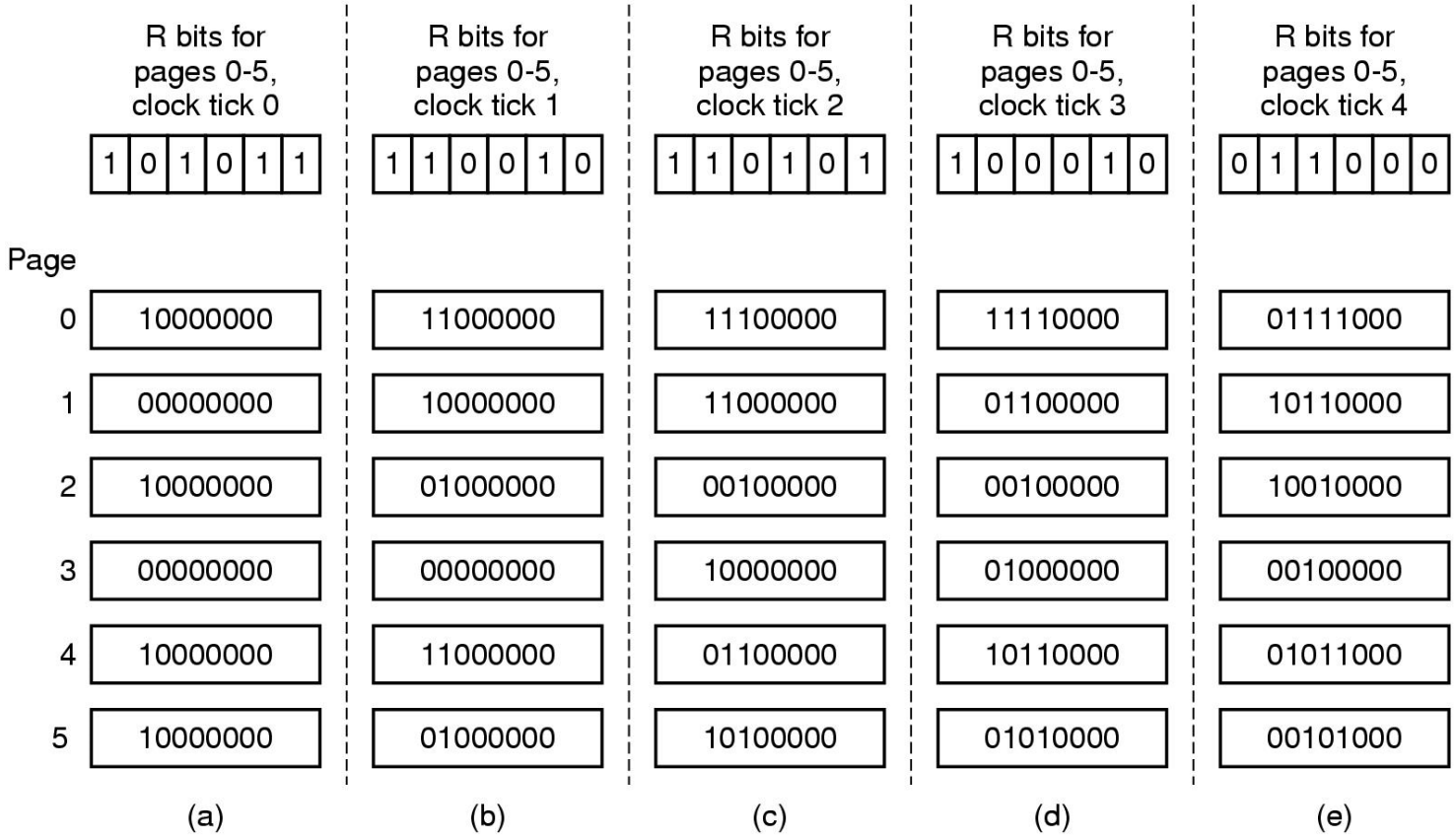
LRU-software

- Hardware uses space=> software implementation
- Make use of software counters

LRU-software

- “aging” algorithm
- Keep a string of values of the R bits for each clock tick (up to some limit)
- After tick, shift bits right and add new R values on the left
- On page fault, evict page with lowest counter
- Size of the counter determines the history

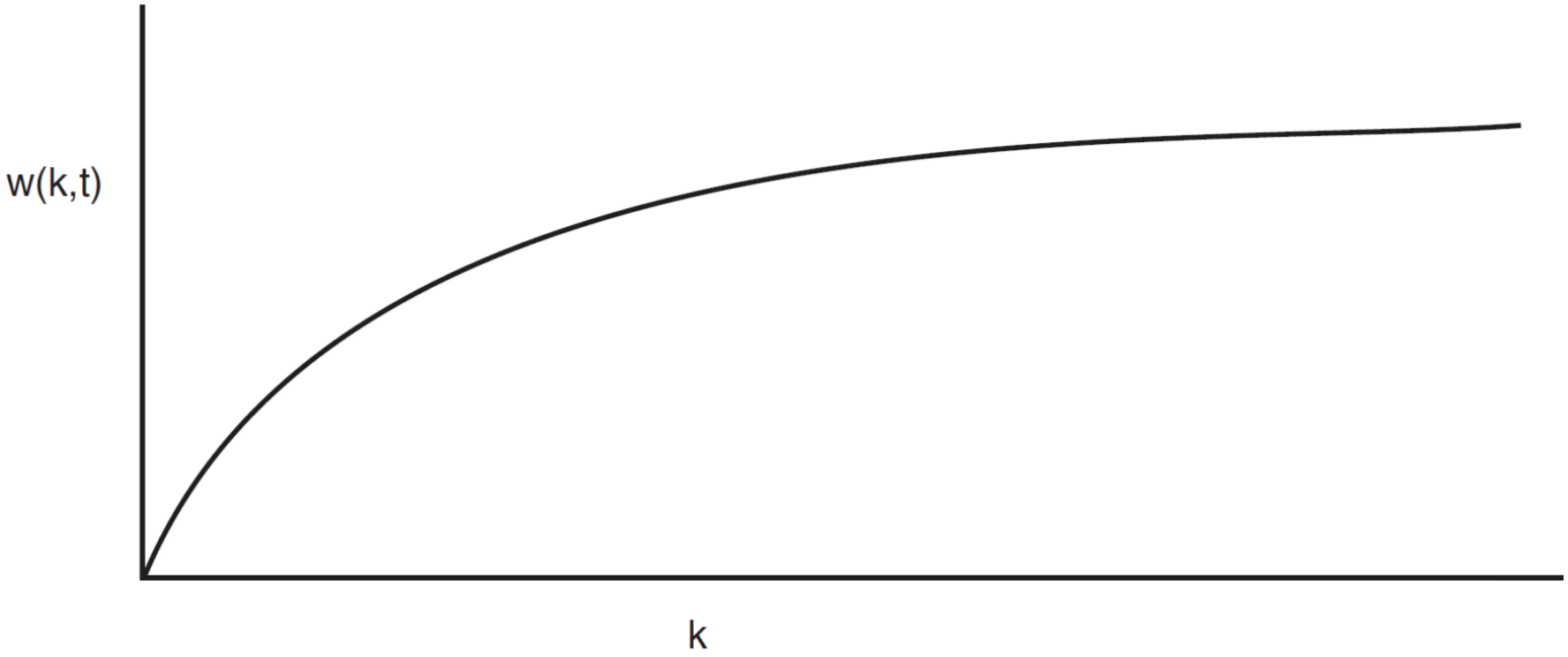
LRU in Software



Working Set Model

- Demand paging - bring a process into memory by trying to execute first instruction and getting page fault. Continue until all pages that process needs to run are in memory (the working set)
- Try to make sure that working set is in memory before letting process run (pre-paging)
- Thrashing - memory is too small to contain working set, so page fault all of the time

Behavior of working set as a function of k



W(k,t) is number of pages at time t used by k most recent memory references

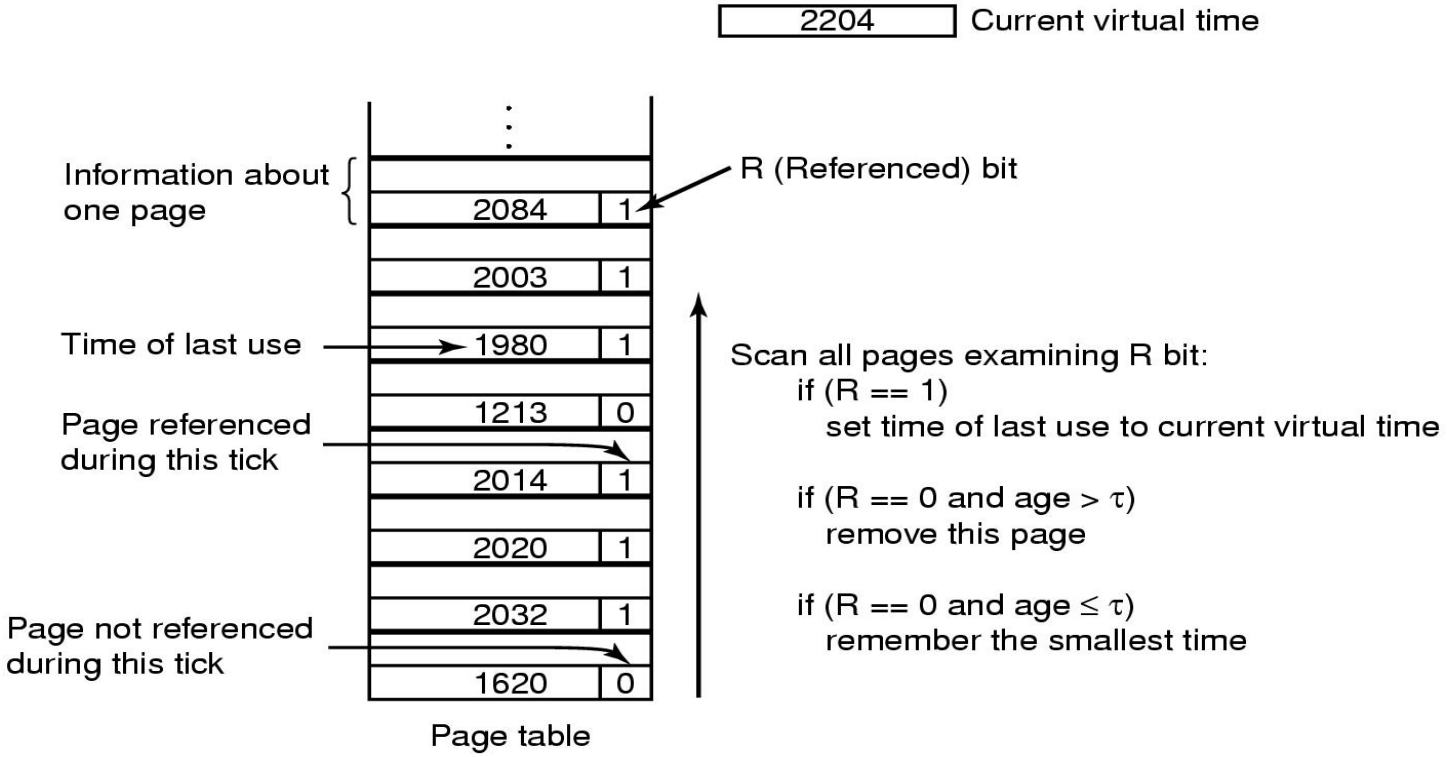
How to implement working set model

- When fault occurs can evict page not in working set (if there is such a page)
- Need to pick k
- Could keep track of pages in memory at every memory reference. Each k references results in a working set.
- Expensive

Use virtual time instead of number of references (k)

- Keep track of k last pages referenced during a period t of execution (CPU) time
- Virtual time for a process is the amount of CPU time used since it started
  - Measure of how much work a process has done

Working Set Page Replacement
(Check each clock tick)



Check each clock tick

Get rid of page with smallest time if all of the pages have R==0

Weakness with WS algorithm

- Need to scan entire page table at each page fault to find a victim
- Use clock idea with working set algorithm

Summary of Page Replacement Algorithms

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude approximation of LRU
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Need to take into account a number of design issues in order to get a working algorithm

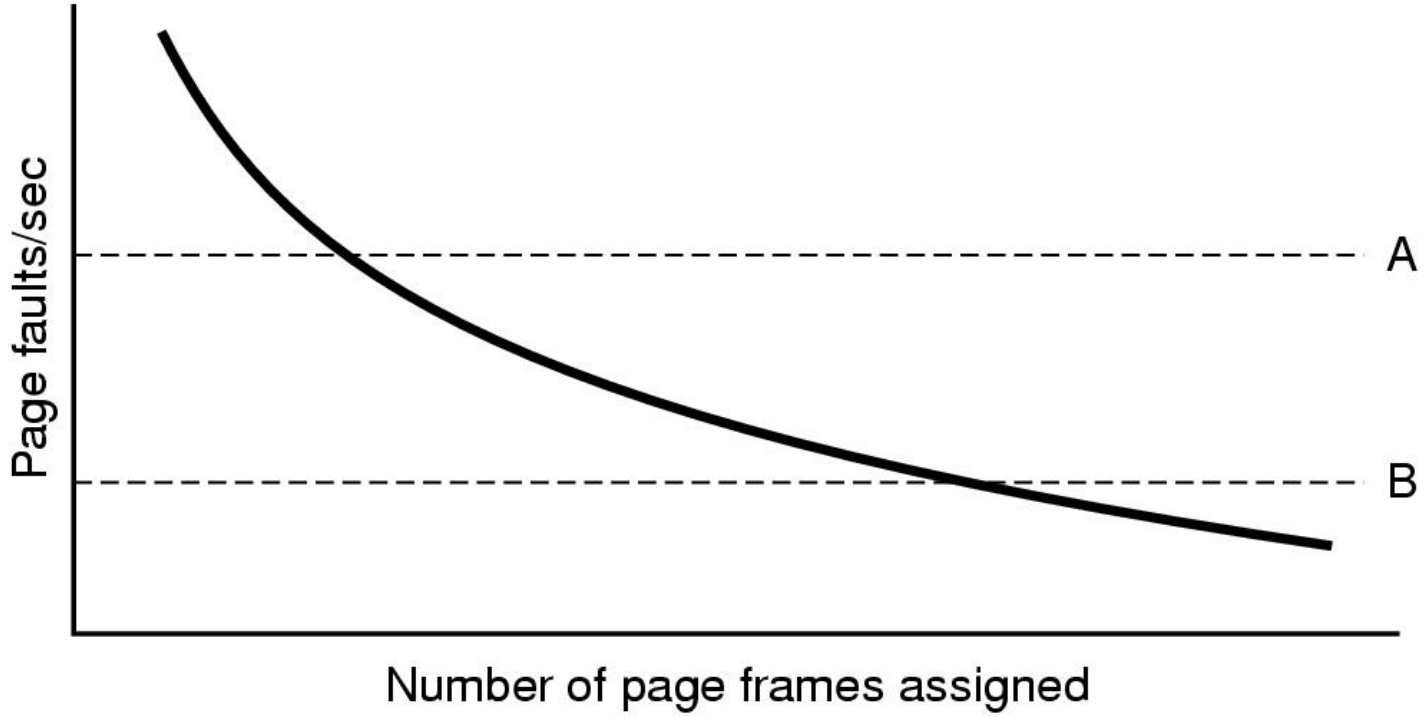
Global versus Local choice of page

- Global-take into account all of the processes
- Local-take into account just the process which faulted

Global is better for the memory

- Working sets grow and shrink over time
- Processes have different sizes
- Assign number of pages to each process proportional to its size
- Start with allocation based on size and use page fault frequency (pff) to modify allocation size for each process

PFF used to determine page allocation



Maintain upper (A) and lower (B) bounds for pff Try to keep process in between bounds

Local Versus Global

- Can use combination of algorithms
- PFF is global component - determines page allocation
- Replacement algorithm Is local component - determines which page to kick out

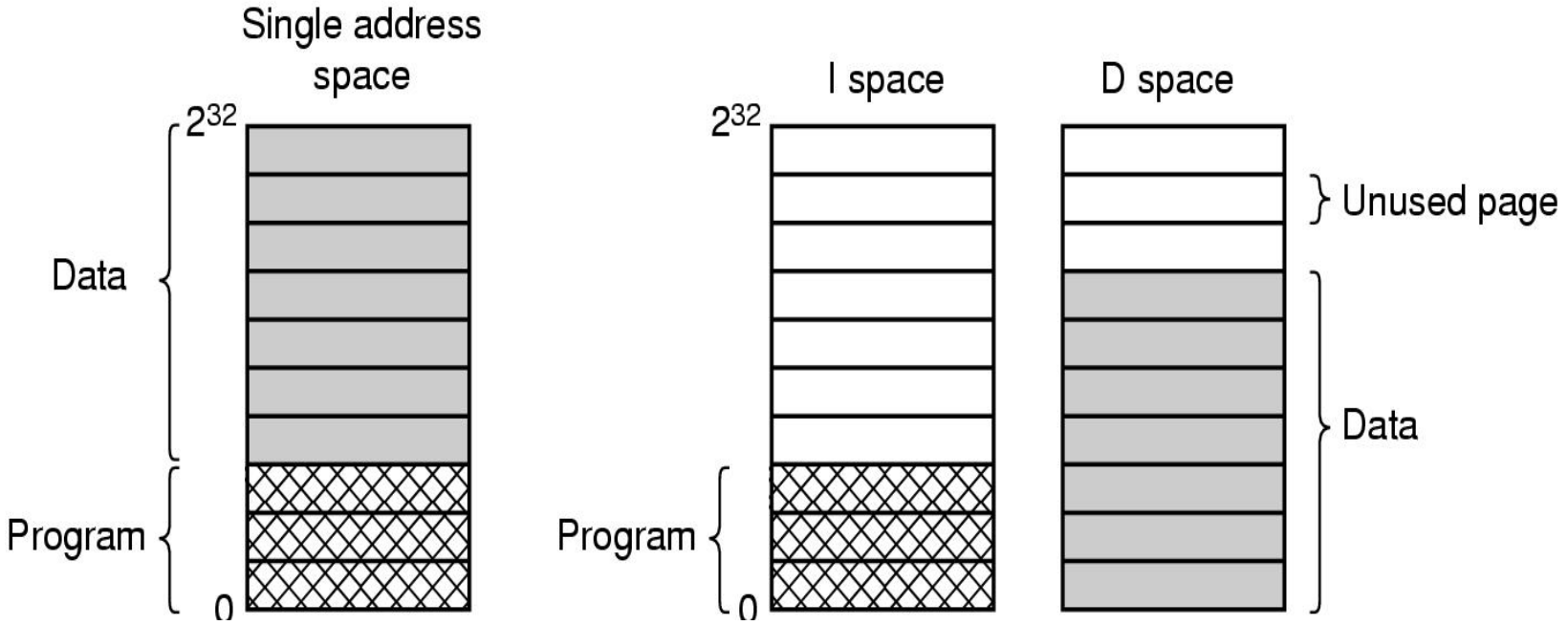
Load Control

- Why? Can still thrash because of too much demand for memory.
- Solution-swap process(es) out .
  - i.e., When desperate, get rid of a process

Page size

- Overhead= s*e/p + p/2 [size of page entries + frag]
  - p is page size
  - s is process size
  - e is size of the page entry (in page table)
- Differentiate, set to zero => p = √(2s*e)
  - s= 1 MB, e=8 bytes 4 KB is optimal
  - 1 KB is typical
  - 4-8 KB common
  - OK, this is a rough approach

Separate Instruction and Data Address Spaces



Process address space too small => difficult to fit program in space

Split space into instructions (I) and data (D)

Shared Pages

- Different users can run same program (with different data) at the same time.
- Not all pages can be shared (data can't be shared, text can be shared)
- Process can't drop pages when it exits w/o being certain that they not still in use
  - Use special data structure to track shared pages
- Data sharing is painful (e.g. *Unix fork, parent and child share text and data*) because of page writes
  - (Copy on write) solution is to map data to read only pages. If write occurs, each process gets its own page.

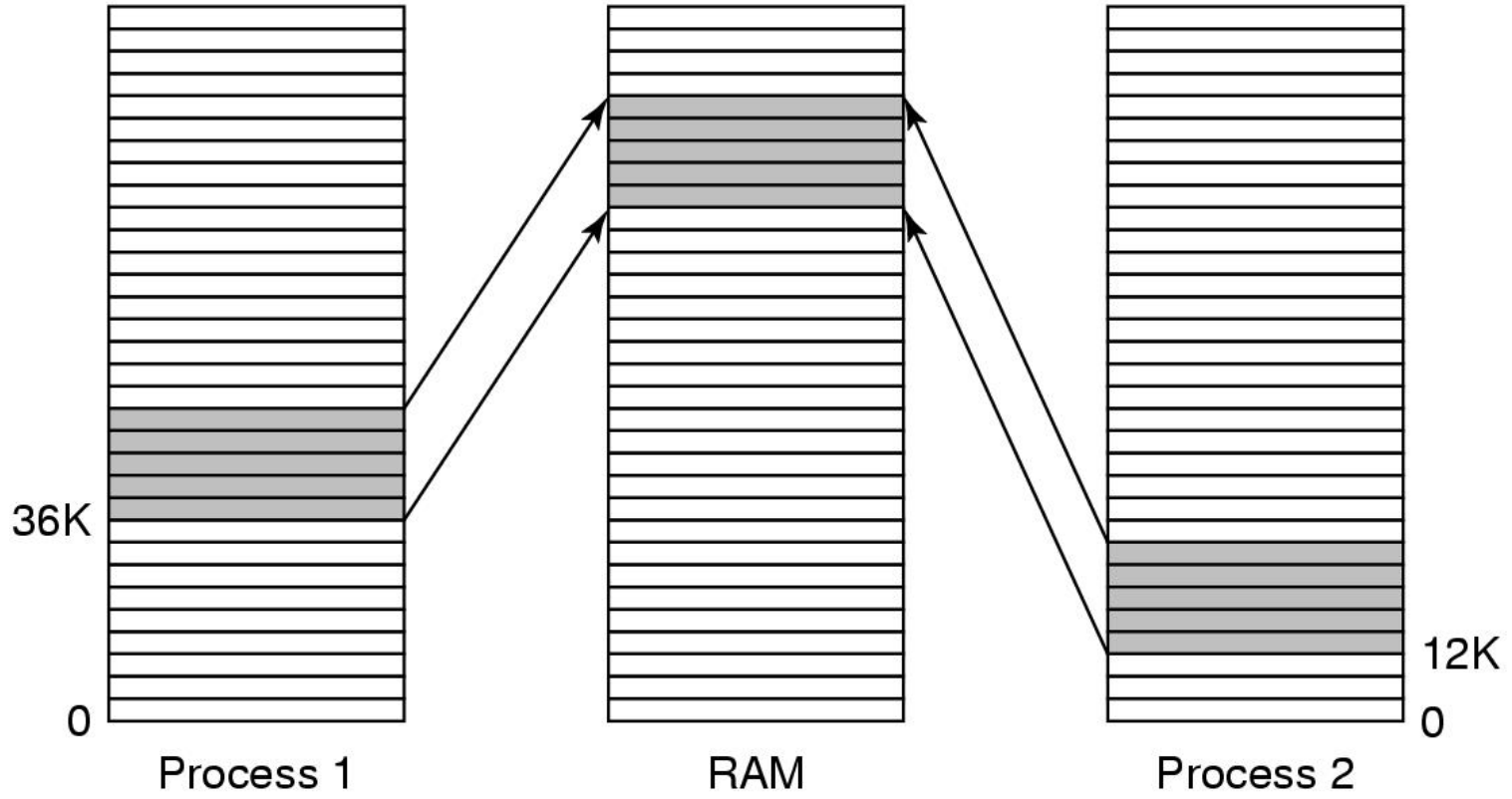
Shared Libraries

- Large libraries (e.g. graphics) used by many process. Too expensive to bind to each process which wants to use them. Use shared libraries instead.
- Unix linking: ld*.o –lc –lm . Files (and no others) not present in .o are located in m or c libraries and included in binaries.

Shared Libraries

- Linker uses a stub routine to call which binds to called function AT RUN TIME.
  - Shared library is only loaded once (first time that a function in it is referenced).
  - It is paged in
- Need to use position independent code to avoid going to the wrong address (next slide).
  - Idea: Compiler does not produce absolute addresses when using shared libraries; only relative addresses.

Shared Libraries



Memory Mapped Files

- Process issues system call to map a file onto a part of its virtual address space.
- Can be used to used to **communicate** via shared memory. Processes share same file. Use it to read and write.

Cleaning Policy

- Use a daemon to locate pages to evict before you need them instead of looking for victims when you need them
- Daemon sleeps most of the time, periodically awakens If there are “too few” frames, kicks out frames
- Make sure that they are clean before claiming them

Virtual Memory Interface

- Might want 2 programs to share physical memory
- Easy way to implement shared message passing
  - Avoids memory copies
- Distributed shared memory-page fault handler locates page in different machine, which sends page to machine that needs it

Implementation

- OS has lots of involvement in paging when process is: created, executes, page fault happens, terminates
- Look at several specific issues/problems
  - **Page fault handling**

Page Fault Handling (1)

- The hardware traps to the kernel, saving the program counter on the stack.
- An assembly code routine is started to save the general registers and other volatile information.
- The operating system discovers that a page fault has occurred, and tries to discover which virtual page is needed.
- Once the virtual address that caused the fault is known, the system checks to see if this address is valid and the protection consistent with the access

Page Fault Handling (2)

- If the page frame selected is dirty, the page is scheduled for transfer to the disk, and a context switch takes place.
- When page frame is clean, operating system looks up the disk address where the needed page is, schedules a disk operation to bring it in.
- When disk interrupt indicates page has arrived, page tables updated to reflect position, frame marked as being in normal state.

Page Fault Handling (3)

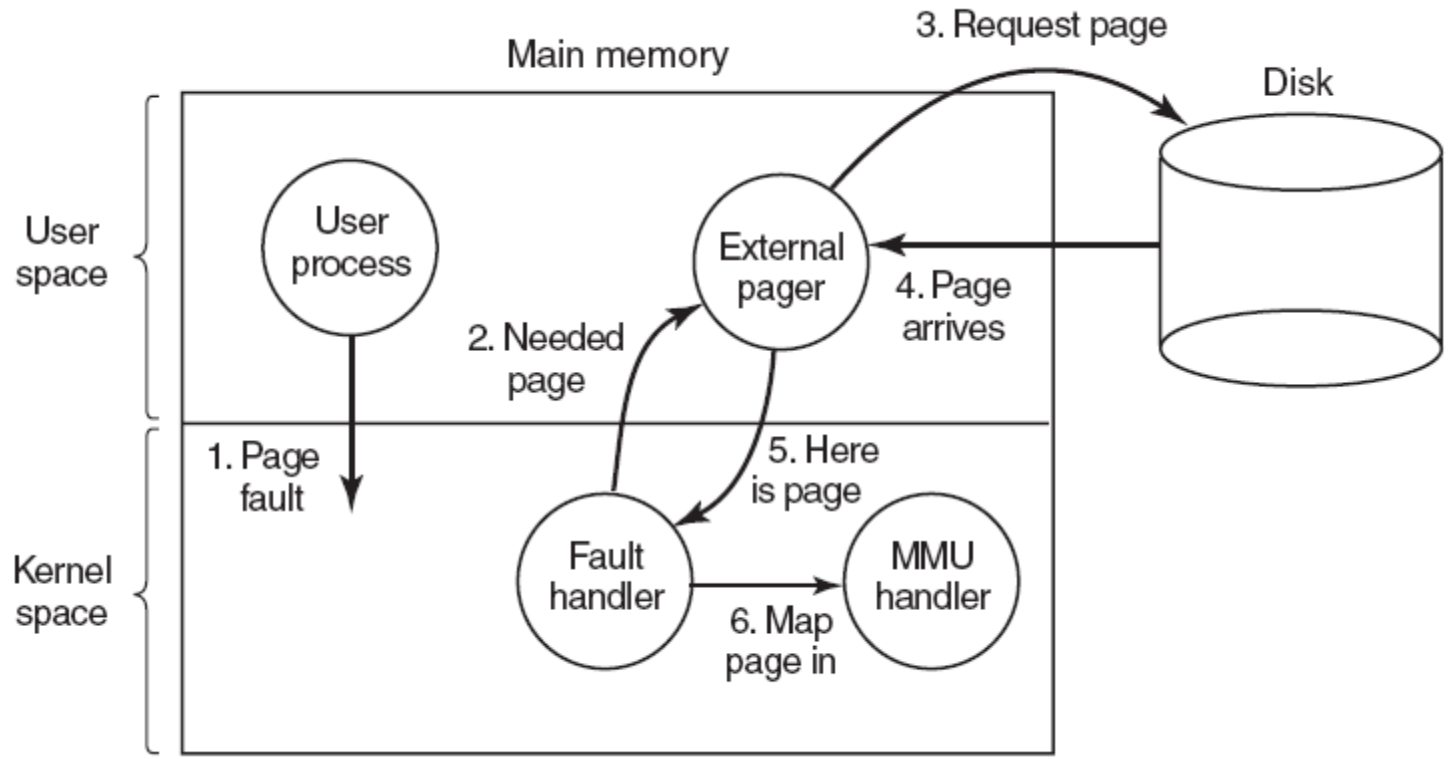
- Faulting instruction backed up to state it had when it began and program counter reset to point to that instruction.
- Faulting process scheduled, operating system returns to the (assembly language) routine that called it.
- This routine reloads registers and other state information and returns to user space to continue execution, as if no fault had occurred.

Separation of Policy and Mechanism (1)

Memory management system

- A low-level MMU handler (machine dependent)
- A page fault handler that is part of the kernel (machine independent) Asks MMU to assign space for incoming page in the process
- An external pager running in user space which contains the policy for page replacement and asks/receives pages from disk

Separation of Policy and Mechanism (2)



How page fault happens-who does what.

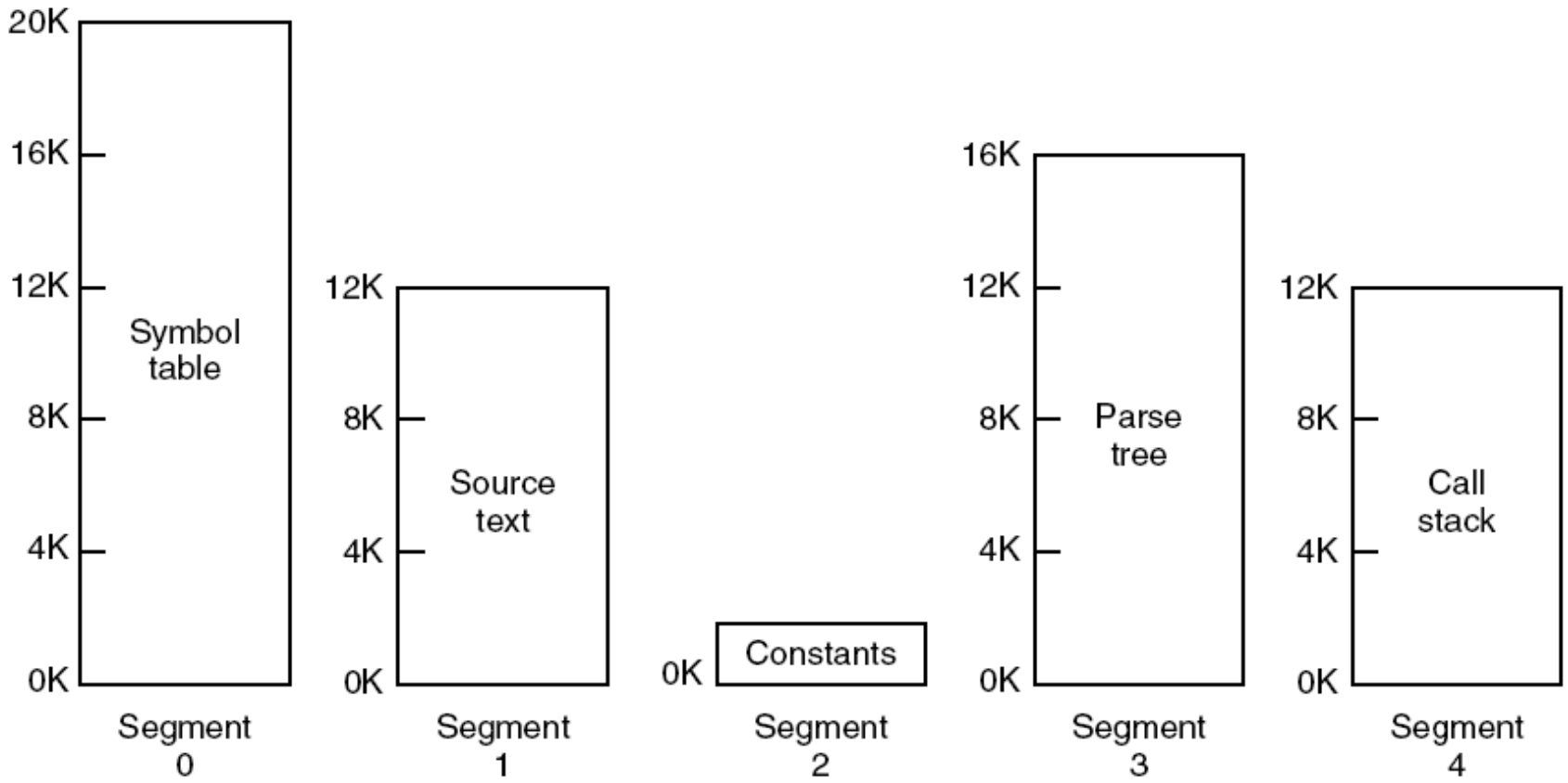
The good vs the bad

- The good: modular code => greater flexibility
- The bad: Cross the user/kernel interface several times in the course of a page fault

Segmentation

- A compiler has many tables that are built up as compilation proceeds, possibly including:
  - The source text being saved for the printed listing (on batch systems).
  - The symbol table – the names and attributes of variables.
  - The table containing integer, floating-point constants used.
  - The parse tree, the syntactic analysis of the program.
  - The stack used for procedure calls within the compiler.

Segmentation



A segmented memory allows each table to grow or shrink independently of the other tables.

Paging vs Segmentation

Consideration	Paging	Segmentation
Need the programmer be aware that this technique is being used?	No	Yes
How many linear address spaces are there?	1	Many
Can the total address space exceed the size of physical memory?	Yes	Yes
Can procedures and data be distinguished and separately protected?	No	Yes
Can tables whose size fluctuates be accommodated easily?	No	Yes
Is sharing of procedures between users facilitated?	No	Yes
Why was this technique invented?	To get a large linear address space without having to buy more physical memory	To allow programs and data to be broken up into logically independent address spaces and to aid sharing and protection

Takeaways