# CSE 380
# Computer Operating Systems

**Instructor: Insup Lee**

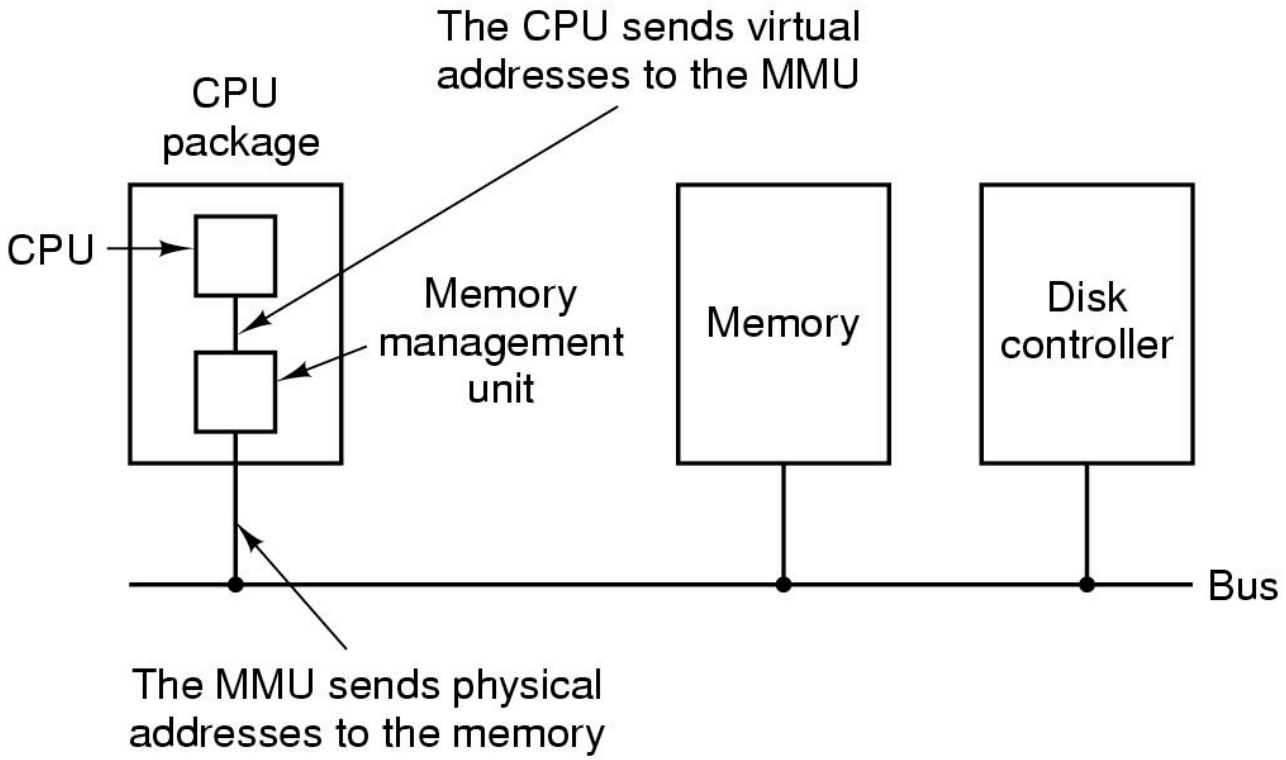
**University of Pennsylvania**
**Fall 2003**
**Lecture Note: Virtual Memory**
***(revised version)***

# Virtual Memory

- Recall: memory allocation with variable partitions requires mapping logical addresses to physical addresses
- Virtual memory achieves a complete separation of logical and physical address-spaces
- Today, typically a virtual address is 32 bits, this allows a process to have 4GB of virtual memory
  - Physical memory is much smaller than this, and varies from machine to machine
  - Virtual address spaces of different processes are distinct
- Structuring of virtual memory
  - Paging: Divide the address space into fixed-size pages
  - Segmentation: Divide the address space into variable-size segments (corresponding to logical units)

# Virtual Memory

## Paging (1)



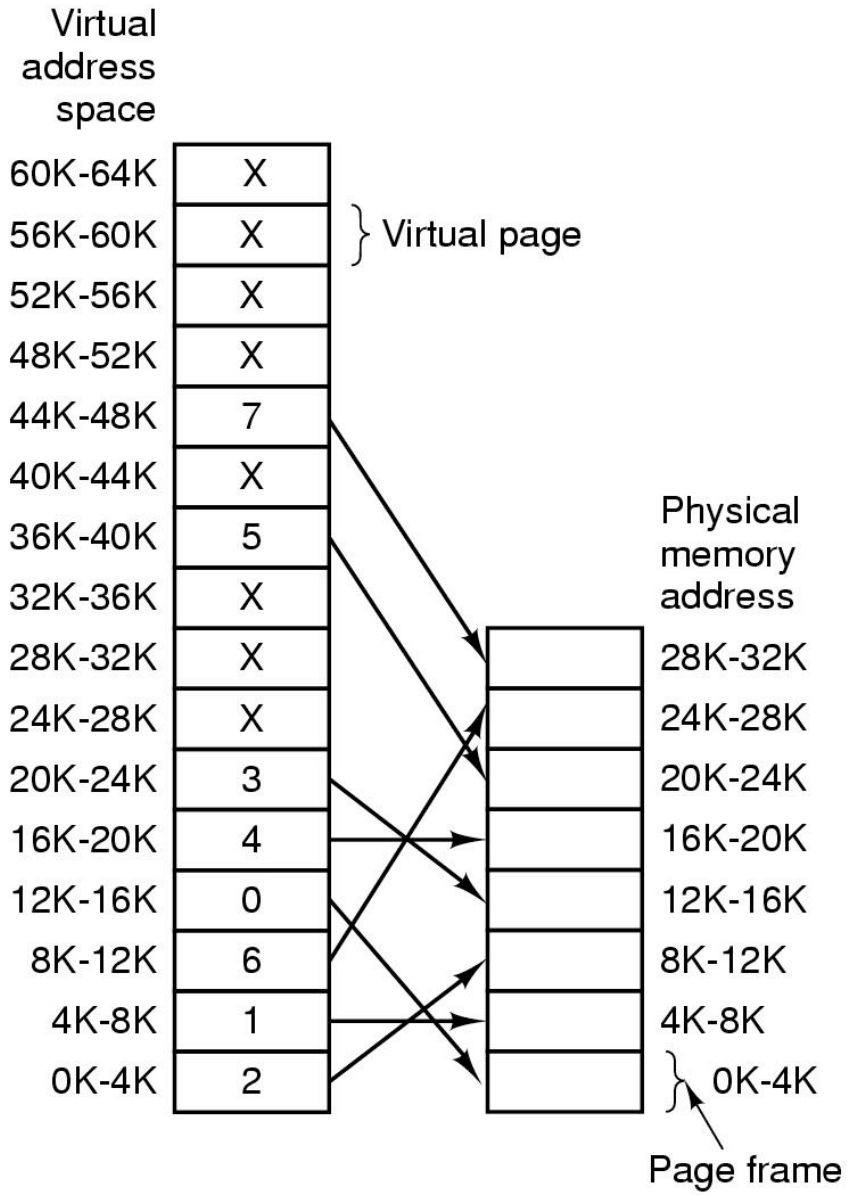

The position and function of the MMU

# Paging

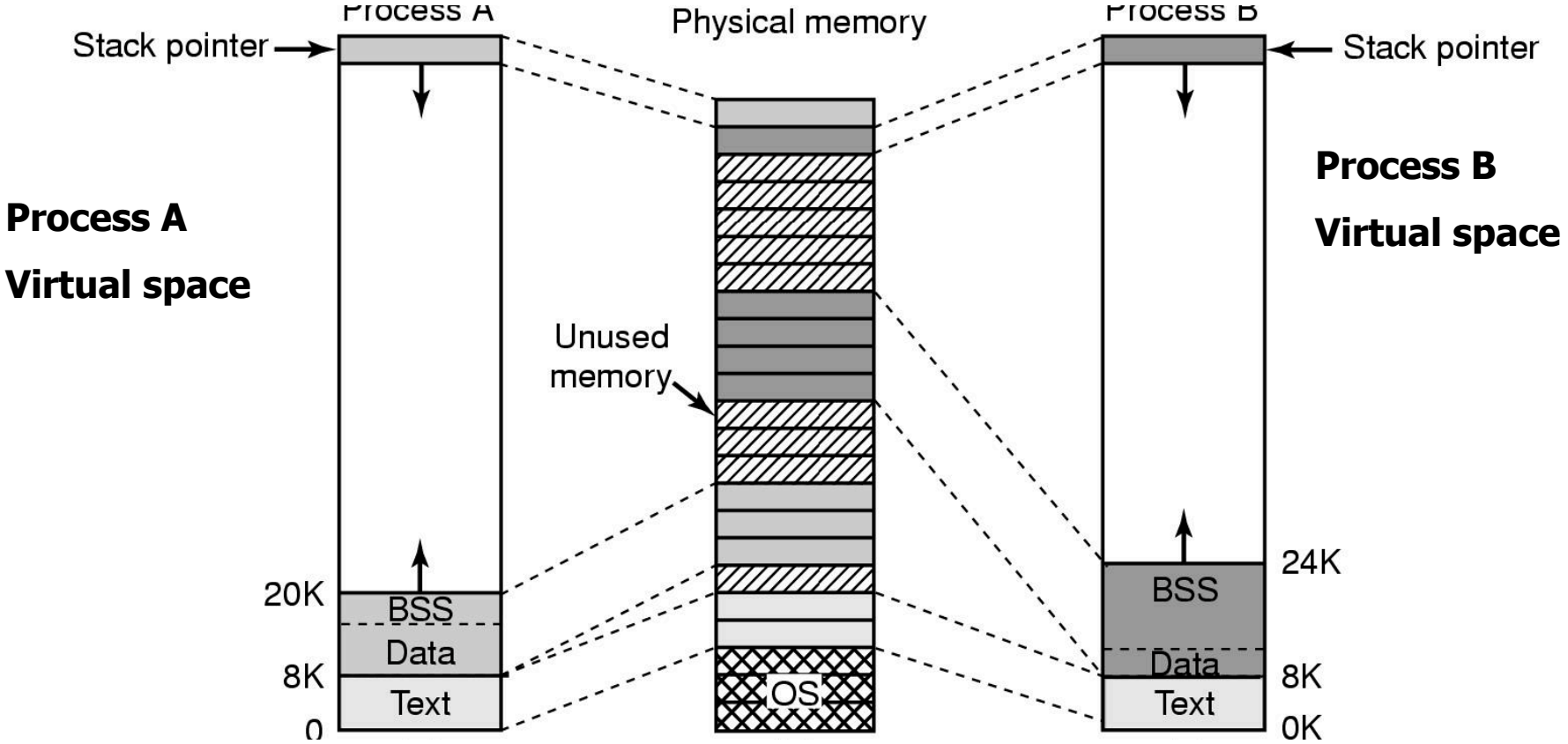
- Physical memory is divided into chunks called page-frames (on Pentium, each page-frame is 4KB)
- Virtual memory is divided into chunks called pages; size of a page is equal to size of a page frame
  - So typically, $2^{20}$ pages (a little over a million) in virtual memory
- OS keeps track of mapping of pages to page-frames
- Some calculations:
  - 10-bit address : 1KB of memory; 1024 addresses
  - 20-bit address : 1MB of memory; about a million addresses
  - 30-bit address : 1 GB of memory; about a billion addresses

# Paging (2)

The relation between virtual addresses and physical memory addresses given by page table



# Virtual Memory in Unix



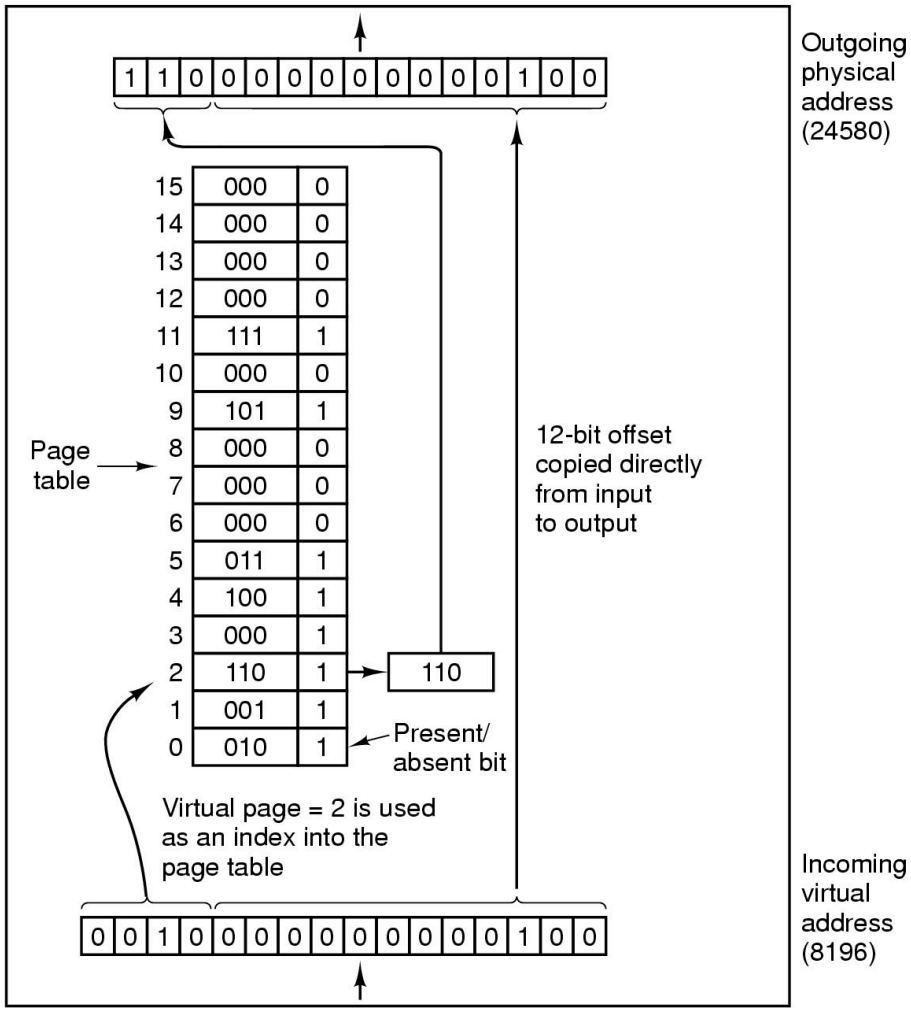
# Paging

- A virtual address is considered as a pair (p,o)
  - Low-order bits give an offset o within the page
  - High-order bits specify the page p
- E.g. If each page is 1KB and virtual address is 16 bits, then low-order 10 bits give the offset and high-order 6 bits give the page number
- The job of the Memory Management Unit (MMU) is to translate the page number p to a frame number f
  - The physical address is then (f,o), and this is what goes on the memory bus
- For every process, there is a page-table (basically, an array), and page-number p is used as an index into this array for the translation

# Page Table Entry

1. Validity bit: Set to 0 if the corresponding page is not in memory
2. Frame number
   - Number of bits required depends on size of physical memory
3. Protection bits:
   - Read, write, execute accesses
4. Referenced bit is set to 1 by hardware when the page is accessed: used by page replacement policy
5. Modified bit (dirty bit) set to 1 by hardware on write-access: used to avoid writing when swapped out

# Page Tables (1)




Internal operation of MMU with 16 4 KB pages

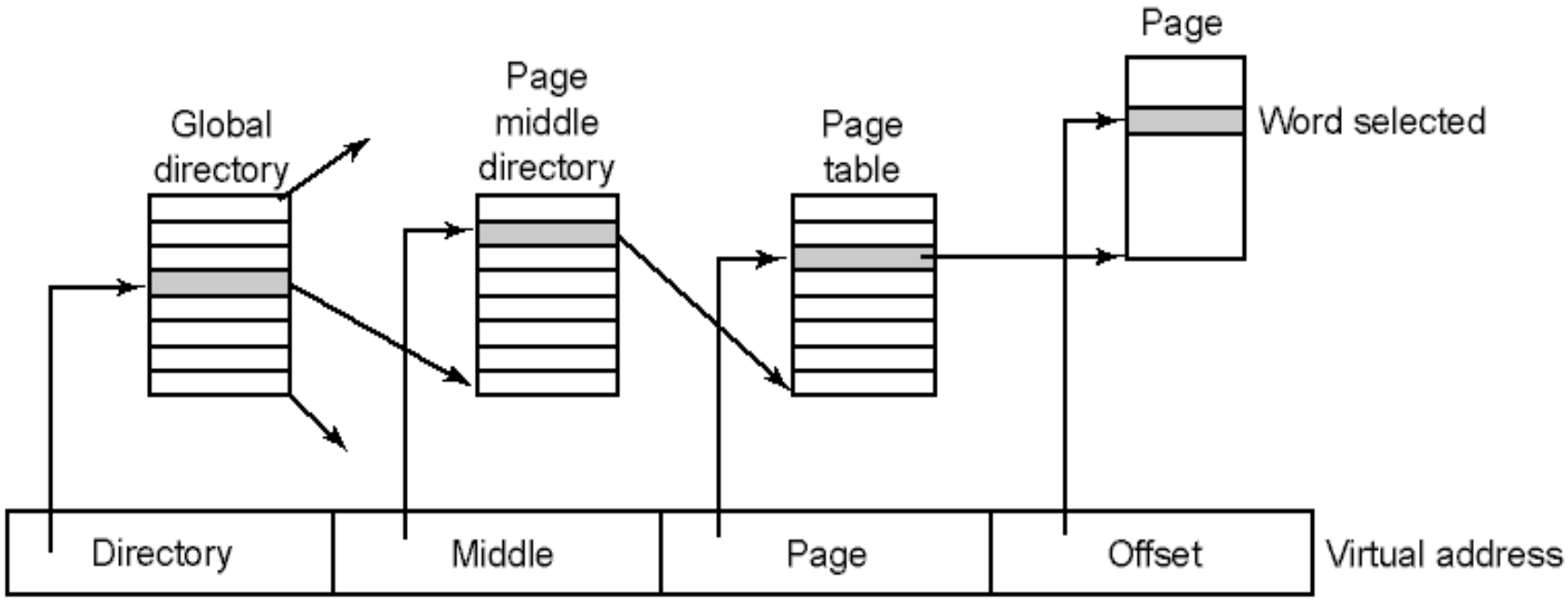
# Design Issues

- What is the "optimal" size of a page frame ?
  - Typically 1KB – 4KB, but more on this later
- How to save space required to store the page table
  - With 20-bit page address, there are over a million pages, so the page-table is an array with over million entries
  - Solns: Two-level page tables, TLBs (Translation Lookaside Beffers), Inverted page tables
- What if the desired page is not currently in memory?
  - This is called a page fault, and it traps to kernel
  - Page daemon runs periodically to ensure that there is enough free memory so that a page can be loaded from disk upon a page fault
- Page replacement policy: how to free memory?

# Multi-Level Paging

- Keeping a page-table with $2^{20}$ entries in memory is not viable
- Solution: Make the page table hierarchical
  - Pentium supports two-level paging
- Suppose first 10-bits index into a top-level page-entry table T1 (1024 or 1K entries)
- Each entry in T1 points to another, second-level, page table with 1K entries (4 MB of memory since each page is 4KB)
- Next 10-bits of physical address index into the second-level page-table selected by the first 10-bits
- Total of 1K potential second-level tables, but many are likely to be unused
- If a process uses 16 MB virtual memory then it will have only 4 entries in top-level table (rest will be marked unused) and only 4 second-level tables

# Paging in Linux

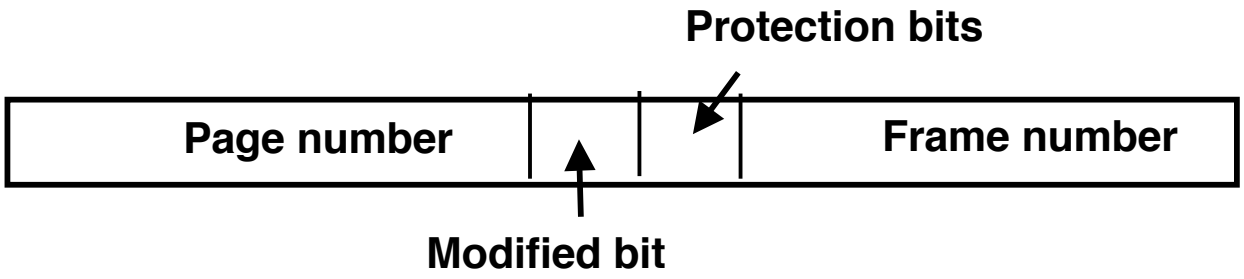



Linux uses three-level page tables

# Translation Lookaside Buffer (TLB)

- Page-tables are in main memory
- Access to main memory is slow compared to clock cycle on CPU (10ns vs 1 ns)
- An instruction such as **MOVE REG, ADDR** has to decode **ADDR** and thus go through page tables
- This is way too slow !!
- Standard practice: Use TLB stored on CPU to map pages to page-frames
- TLB stores small number (say, 64) of page-table entries to avoid the usual page-table lookup
- TLB is associative memory and contains, basically, pairs of the form (page-no, page-frame)
- Special hardware compares incoming page-no in parallel with all entries in TLB to retrieve page-frame
- If no match found in TLB, standard look-up invoked

# More on TLB

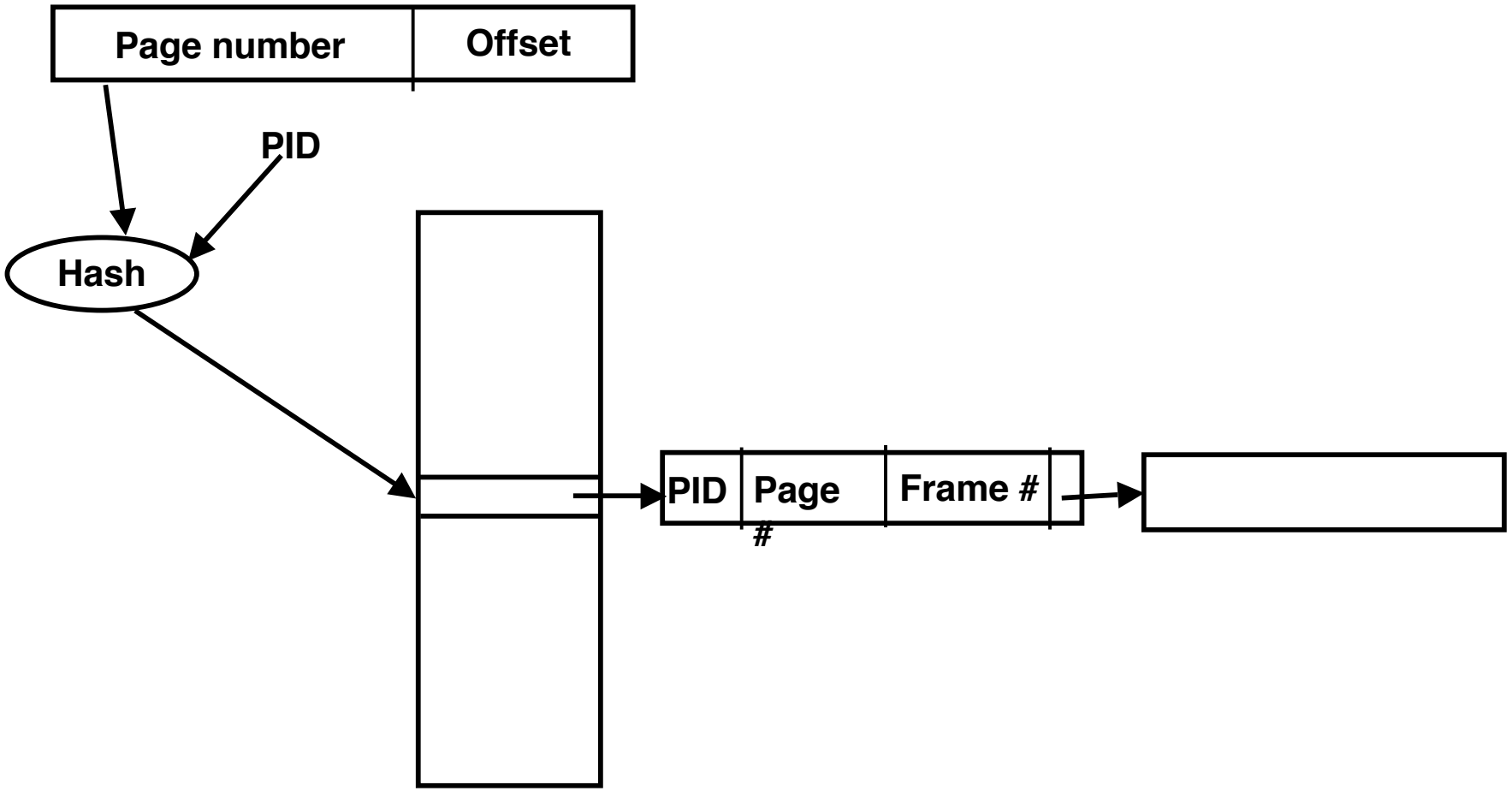



- Key design issue: how to improve hit rate for TLB?
  - Which pages should be in TLB: most recently accessed
- Who should update TLB?
  - Modern architectures provide sophisticated hardware support to do this
  - Alternative: TLB miss generates a fault and invokes OS, which then decides how to use the TLB entries effectively.

# Inverted Page Tables

- When virtual memory is much larger than physical memory, overhead of storing page-table is high
- For example, in 64-bit machine with 4KB per page and 256 MB memory, there are 64K page-frames but $2^{52}$ pages !
- Solution: Inverted page tables that store entries of the form (page-frame, process-id, page-no)
- At most 64K entries required!
- Given a page p of process x, how to find the corresponding page frame?
- Linear search is too slow, so use hashing
- Note: issues like hash-collisions must be handled
- Used in some IBM and HP workstations; will be used more with 64-bit machines

# Hashed Page Tables




**Hash table**
**Number of entries: Number of page frames**

# Steps in Paging

- Today's typical systems use TLBs and multi-level paging
- Paging requires special hardware support
- Overview of steps
  1. Input to MMU: virtual address = (page p, offset o)
  2. Check if there is a frame f with (p,f) in TLB
  3. If so, physical address is (f,o)
  4. If not, lookup page-table in main memory ( a couple of accesses due to multi-level paging)
  5. If page is present, compute physical address
  6. If not, trap to kernel to process page-fault
  7. Update TLB/page-table entries (e.g. Modified bit)

# Page Fault Handling

- Hardware traps to kernel on page fault
- CPU registers of current process are saved
- OS determines which virtual page needed
- OS checks validity of address, protection status
- Check if there is a free frame, else invoke page replacement policy to select a frame
- If selected frame is dirty, write it to disk
- When page frame is clean, schedule I/O to read in page
- Page table updated
- Process causing fault rescheduled
- Instruction causing fault reinstated (this may be tricky!)
- Registers restored, and program continues execution

# Paging Summary

- How long will access to a location in page p take?
  - If the address of the corresponding frame is found in TLB?
  - If the page-entry corresponding to the page is valid?
    - Using two-level page table
    - Using Inverted hashed page-table
  - If a page fault occurs?
- How to save space required to store a page table?
  - Two-level page-tables exploit the fact only a small and contiguous fraction of virtual space is used in practice
  - Inverted page-tables exploit the fact that the number of valid page-table entries is bounded by the available memory
- Note: Page-table for a process is stored in user space

# Page Replacement Algorithms

- When should a page be replaced
  - Upon a page fault if there are no page frames available
  - By pager daemon executed periodically
- Pager daemon needs to keep free page-frames
  - Executes periodically (e.g. every 250 msec in Unix)
  - If number of free page frames is below certain fraction (a settable parameter), then decides to free space
- Modified pages must first be saved
  - unmodified just overwritten
- Better not to choose an often used page
  - will probably need to be brought back in soon
- Well-understood, practical algorithms
- Useful in other contexts also (e.g. web caching)

# Reference String

**Def:** The virtual space of a process consists of $N = \{1,2,\ldots,n\}$ pages.

A process *reference string w* is the sequence of pages referenced by a process for a given input:

$$w = r_1\ r_2\ \ldots\ r_k\ \ldots\ r_T$$

where $r_k \in N$ is the page referenced on the $k^{th}$ memory reference.

E.g., N = {0,...,5}.

w = 0 0 3 4 5 5 5 2 2 2 1 2 2 2 1 1 0 0

Given *f* page frames,

- warm-start behavior of the replacement policy
- cold-start behavior of the replacement policy

# Forward and backward distances

- **Def:** The *forward distance* for page $X$ at time $t$, denoted by $d_t(X)$, is

- $d_t(X) = k$ if the first occurrence of $X$ in $r_{t+1}\ r_{t+2}$ …at $r_{t+k}$.
- $d_t(X) = \infty$ if $X$ does not appear in $r_{t+1}\ r_{t+2}$ ….

- **Def:** The *backward distance* for page $X$ at time $t$, denoted by $b_t(X)$, is
- $b_t(X) = k$ if $r_{t-k}$ was the last occurrence of $X$.
- $b_t(X) = \infty$ if $X$ does not appear in $r_1\ r_2 \ldots r_{t-1}$.

# Paging Replacement Algorithms

1 Random -- Worst implementable method, easy to implement.

2 FIFO - Replace the longest resident page. Easy to implement since control information is a FIFO list of pages.

Consider a program with 5 pages and reference string

w = 1 2 3 4 1 2 5 1 2 3 4 5

Suppose there are 3 page frames.

| | w = | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PF 1 | | 1 | 1 | 1 | 4 | 4 | 4 | 5 | 5 | 5 | 5 | 5 | 5 |
| PF 2 | | | 2 | 2 | 2 | 1 | 1 | 1 | 1 | 1 | 3 | 3 | 3 |
| PF 3 | | | | 3 | 3 | 3 | 2 | 2 | 2 | 2 | 2 | 4 | 4 |
| victim | | | | | 1 | 2 | 3 | 4 | | | 1 | 2 | |

# Optimal Page Replacement Algorithm

- If we knew the precise sequence of requests for pages, we can optimize for least number of faults
- Replace page needed at the farthest point in future
  - Optimal but unrealizable
- Off-line simulations can estimate the performance of this algorithm, and be used to measure how well the chosen scheme is doing
  - Competitive ratio of an algorithm = (page-faults generated by optimal policy)/(actual page faults)
- Consider reference string: 1 2 3 4 1 2 5 1 2 3 2 5

- Consider a program with 5 pages and reference string

  w = 1 2 3 4 1 2 5 1 2 3 4 5

  Suppose there are 3 page frames.

| | w = | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| PF 1 | | | | | | | | | | | | | |
| PF 2 | | | | | | | | | | | | | |
| PF 3 | | | | | | | | | | | | | |
| victim | | | | | | | | | | | | | |

# First Attempts

- Use reference bit and modified bit in page-table entry
  - Both bits are initially 0
  - Read sets reference to 1, write sets both bits to 1
  - Reference bit cleared on every clock interrupt (40ms)

- Prefer to replace pages unused in last clock cycle
  - First, prefer to keep pages with reference bit set to 1
  - Then, prefer to keep pages with modified bit set to 1

- Easy to implement, but needs additional strategy to resolve ties

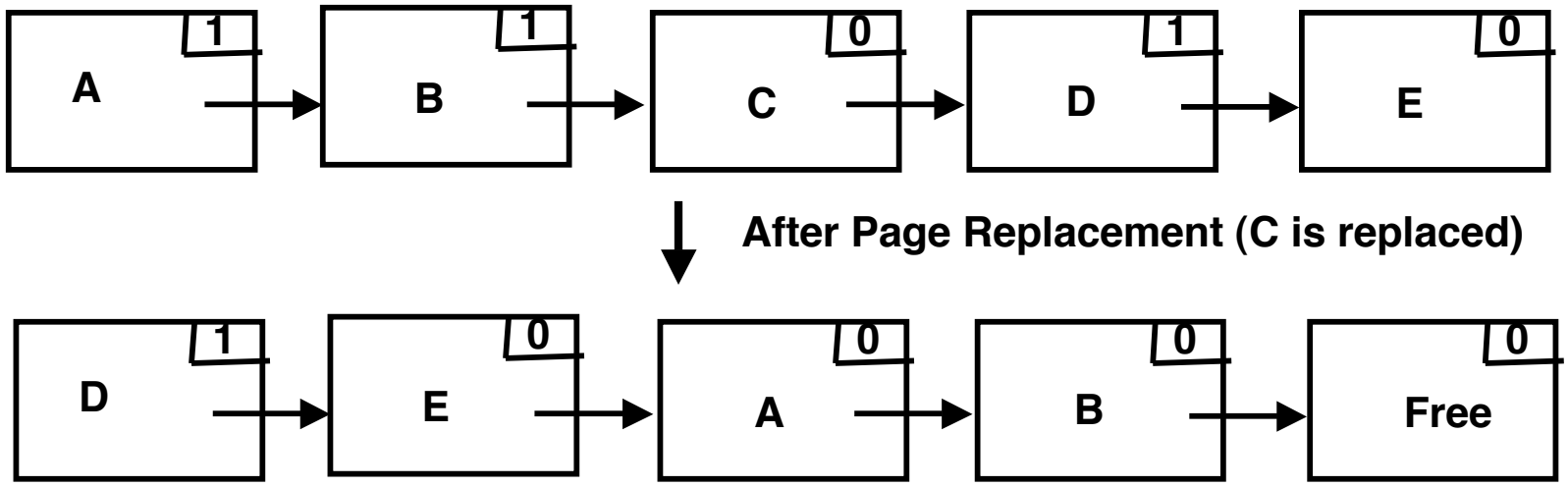
- Note: Upon a clock interrupt, OS updates CPU-usage counters for scheduling in PCB as well as reference bits in page tables

# Queue Based Algorithms

- FIFO
  - Maintain a linked list of pages in memory in order of arrival
  - Replace first page in queue
  - Easy to implement, but access info not used at all
- Modifications
  - Second-chance
  - Clock algorithm

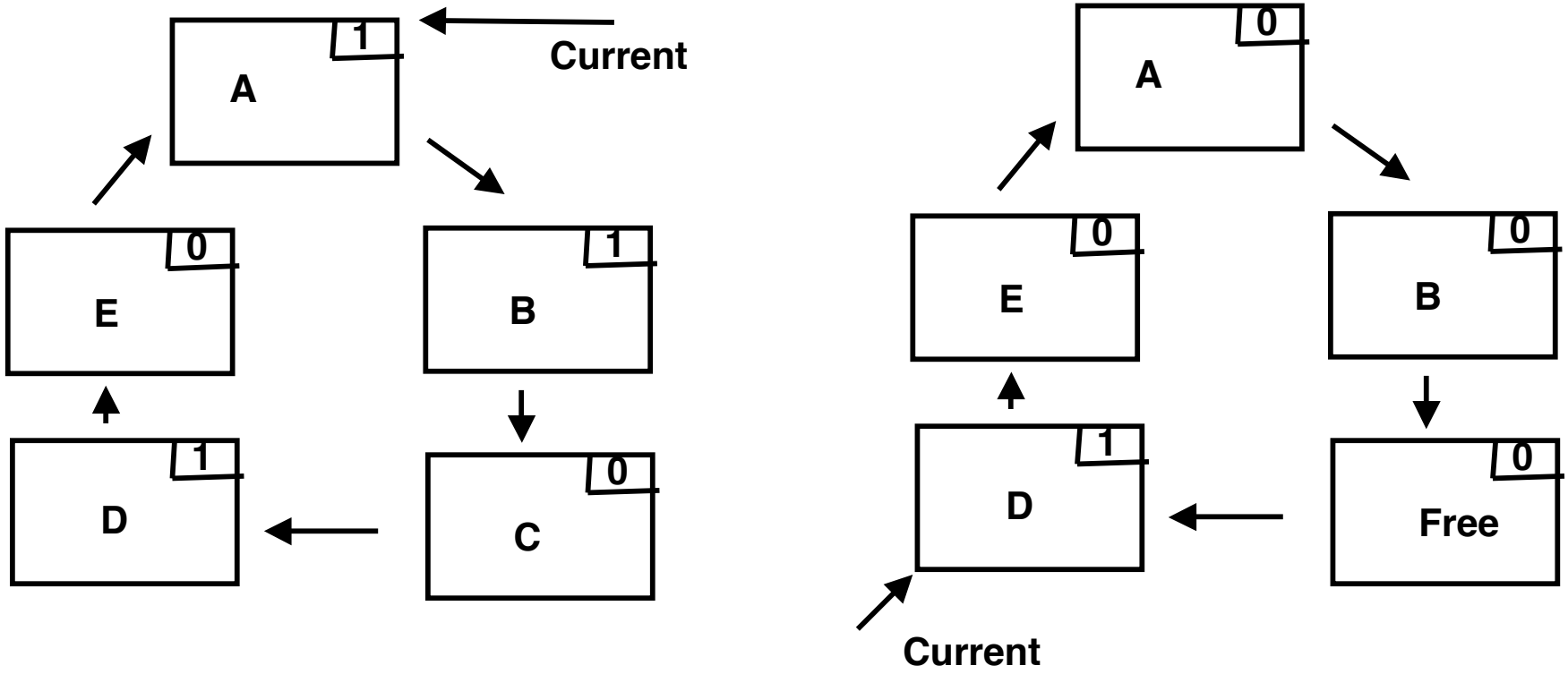
# Second Chance Page Replacement

- Pages ordered in a FIFO queue as before
- If the page at front of queue (i.e. oldest page) has Reference bit set, then just put it at end of the queue with R=0, and try again
- Effectively, finds the oldest page with R=0, (or the first one in the original queue if all have R=1)
- Easy to implement, but slow !!



# Clock Algorithm

- Optimization of Second chance
- Keep a circular list with current pointer
- If current page has R=0 then replace, else set R to 0 and move current pointer



# Least Recently Used (LRU)

- Assume pages used recently will be used again soon
  - throw out page that has been unused for longest time
- Consider the following references assuming 3 frames

  **1 2 3 4 1 2 5 1 2 3 2 5**
- This is the best method that is implementable since the past is usually a good indicator for the future.
- It requires enormous hardware assistance: either a fine-grain timestamp for each memory access placed in the page table, or a sorted list of pages in the order of references.

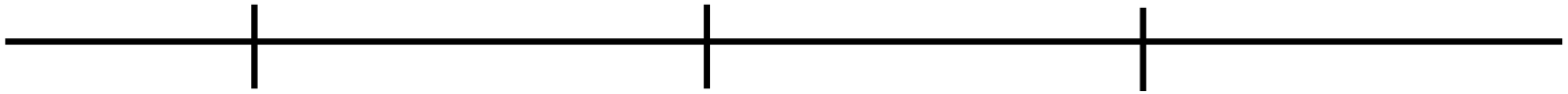
# How to implement LRU?

- Main challenge: How to implement this?
  - Reference bit not enough
- Highly specialized hardware required
- Counter-based solution
  - Maintain a counter that gets incremented with each memory access,
  - Copy the counter in appropriate page table entry
  - On page-fault pick the page with lowest counter
- List based solution
  - Maintain a linked list of pages in memory
  - On every memory access, move the accessed page to end
  - Pick the front page on page fault

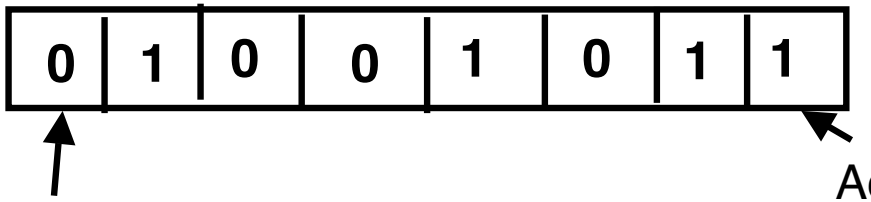
# Approximating LRU: Aging

- Bookkeeping on every memory access is expensive
- Software solution: OS does this on every clock interrupt
- Every page-entry has an additional 8-bit counter
- Every clock cycle, for every page in memory, shift the counter 1 bit to the right copying R bit into the high-order bit of the counter, and clear R bit
- On page-fault, or when pager daemon wants to free up space, pick the page with lowest counter value
- Intuition: High-order bits of recently accessed pages are set to 1 (i-th high-order bit tells us if page was accessed during i-th previous clock-cycle)
- Potential problem: Insufficient info to resolve ties
  - Only one bit info per clock cycle (typically 40ms)
  - Info about accesses more than 8 cycles ago lost

# Aging Illustration



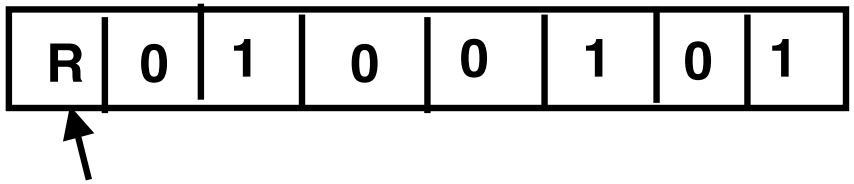
Clock tick




Accessed in last clock cycle?

Accessed in previous 8th clock cycle?

Update upon clock interrupt



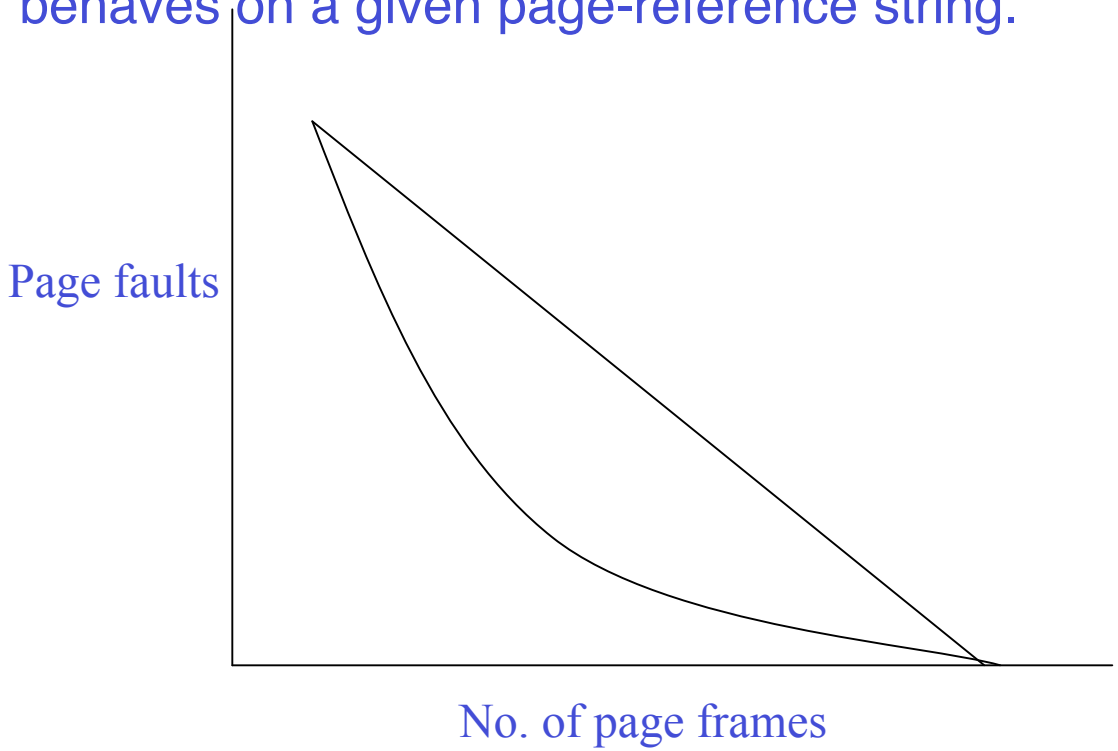

R bit of current cycle

# Analysis of Paging Algorithms

- Reference string *r* for a process is the sequence of pages referenced by the process
- Suppose there are *m* frames available for the process, and consider a page replacement algorithm *A*
  - We will assume demand paging, that is, a page is brought in only upon fault
- Let *F(r,m,A)* be the faults generated by *A*
- Belady's anomaly: allocating more frames may increase the faults: *F(r,m,A)* may be smaller than *F(r,m+1,A)*
- Worth noting that in spite of decades of research
  - Worst-case performance of all algorithms is pretty bad
  - Increase *m* is a better way to reduce faults than improving *A* (provided we are using a stack algorithm)

# Effect of replacement policy

- Evaluate a page replacement policy by observing how it behaves on a given page-reference string.



# Belady's Anomaly

For FIFO algorithm, as the following counter-example shows, increasing *m* from 3 to 4 increases faults

| w | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 1 | 2 | 5 | 5 | 5 | 3 | 4 | 4 | 9 page |
| m=3 | | 1 | 2 | 3 | 4 | 1 | 2 | 2 | 2 | 5 | 3 | 3 | faults |
| | | | 1 | 2 | 3 | 4 | 1 | 1 | 1 | 2 | 5 | 5 | |
| | 1 | 2 | 3 | 4 | 4 | 4 | 5 | 1 | 2 | 3 | 4 | 5 | 10 page |
| m=4 | | 1 | 2 | 3 | 3 | 3 | 4 | 5 | 1 | 2 | 3 | 4 | faults |
| | | | 1 | 2 | 2 | 2 | 3 | 4 | 5 | 1 | 2 | 3 | |
| | | | | 1 | 1 | 1 | 2 | 3 | 4 | 5 | 1 | 2 | |

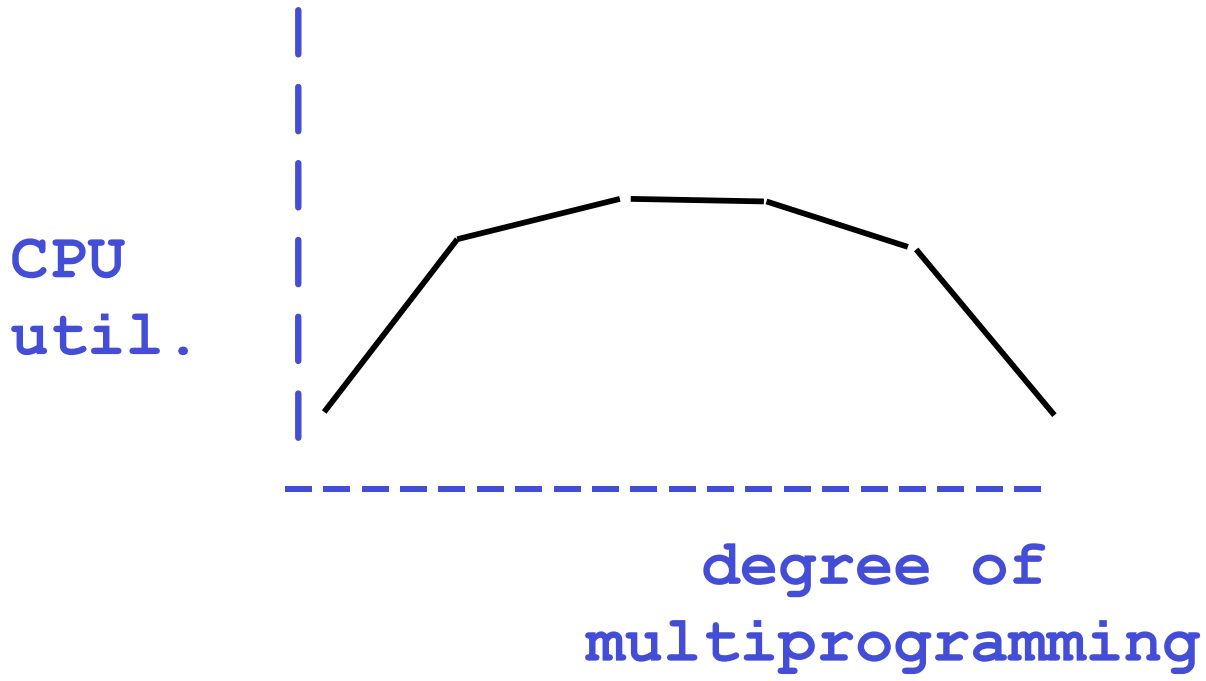
# Stack Algorithms

- For an algorithm *A*, reference string *r*, and page-frames *m*, let *P(r,m,A)* be the set of pages that will be in memory if we run *A* on references *r* using *m* frames
- An algorithm *A* is called a stack algorithm if for all *r* and for all m, *P(r,m,A)* is a subset of *P(r,m+1,A)*
  - Intuitively, this means the set of pages that *A* considers relevant grows monotonically as more memory becomes available
  - For stack algorithms, for all *r* and for all *m*, *F(r,m+1,A)* cannot be more than *F(r,m,A)* (so increasing memory can only reduce faults!)
- LRU is a stack algorithm: *P(r,m,LRU)* should be the last *m* pages in *r*, so *P(r,m,LRU)* is a subset of *P(r,m+1,LRU)*

# Thrashing

**Will the CPU Utilization increase monotonically as the degree Of multiprogramming (number of processes in memory) increases?**

**Not really! It increases for a while, and then starts dropping again.**
**Reason: With many processes around, each one has only a few pages in memory, so more frequent page faults, more I/O wait, less CPU utilization**

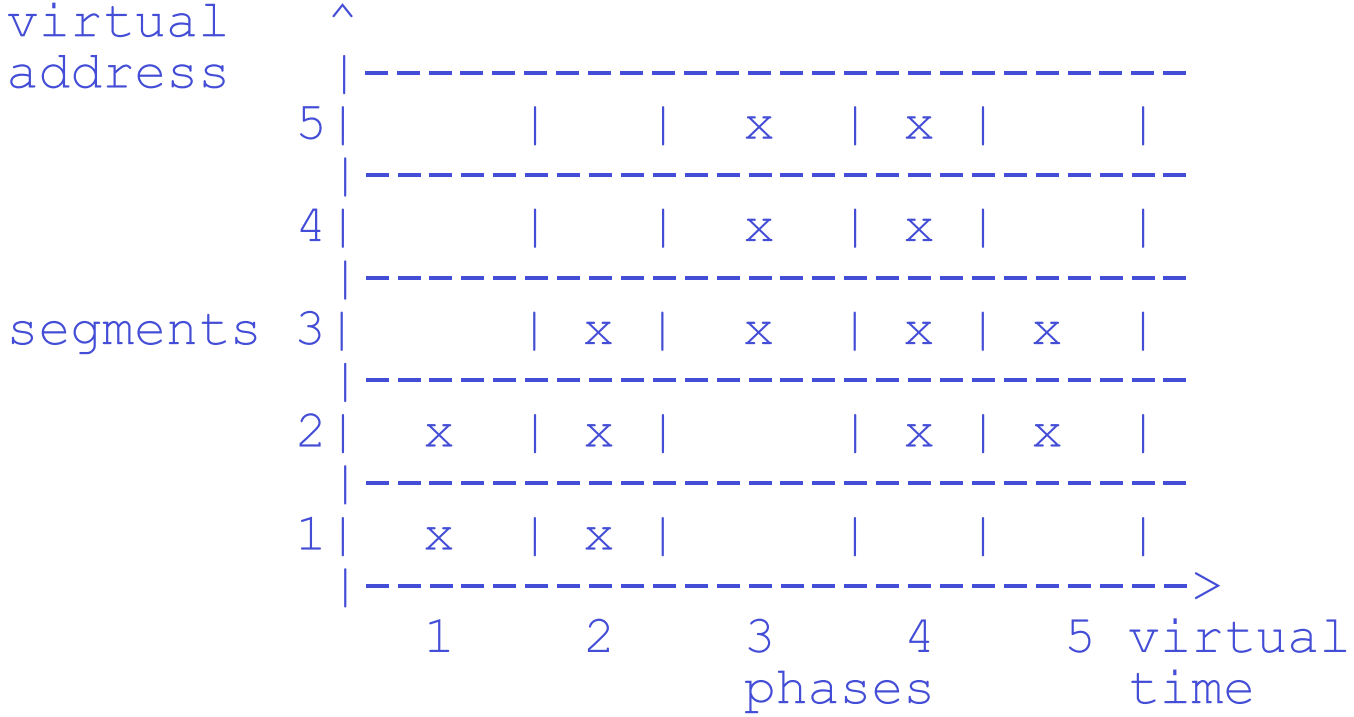



**Bottomline: Cause of low CPU utilization is either too few or too many processes!**

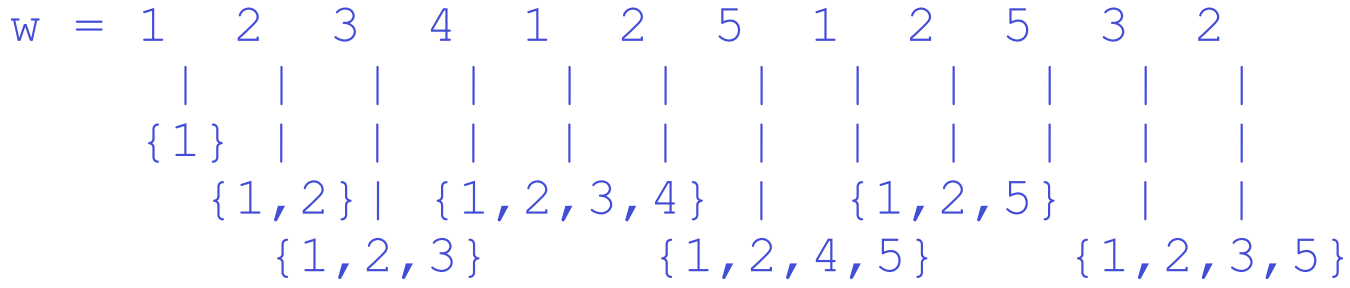
# Locality of Reference

- To avoid thrashing (i.e. too many page faults), a process needs "enough" pages in the memory
- Memory accesses by a program are not spread all over its virtual memory randomly, but show a pattern
  - E.g. while executing a procedure, a program is accessing the page that contains the code of the procedure, the local variables, and global vars
- This is called locality of reference
- How to exploit locality?
  - Prepaging: when a process is brought into memory by the swapper, a few pages are loaded in a priori (note: demand paging means that a page is brought in only when needed)
  - Working set: Try to keep currently used pages in memory

# Locality

- The phenomenon that programs actually use only a limited set of pages during any particular time period of execution.
- This set of pages is called the locality of the program during that time.
- **Ex.** Program phase diagram
- 

# Working Set

- The working set of a process is the set of all pages accessed by the process within some fixed time window.
  Locality of reference means that a process's working set is usually small compared to the total number of pages it possesses.
- A program's working set at the *k*-th reference with window size *h* is defined to be
  $W(k,h) = \{ i \in N \mid \text{page } i \text{ appears among } r_{k-h+1} \ldots r_k \}$
- The working set at time *t* is
  $W(t,h) = W(k,h)$ where $time(r_k) \le t < t(r_{k+1})$
- **Ex.** $h=4$
- 

# Working Set

- Working set of a process at time t is the set of pages referenced over last k accesses (here, k is a parameter)
- Goal of working set based algorithms: keep the working set in memory, and replace pages not in the working set
- Maintaining the precise working set not feasible (since we don't want to update data structures upon every memory access)
- Compromise: Redefine working set to be the set of pages referenced over last m clock cycles
  - Recall: clock interrupt happens every 40 ms and OS can check if the page has been referenced during the last cycle (R=1)
- Complication: what if a process hasn't been scheduled for a while? Shouldn't "over last m clock cycles" mean "over last m clock cycles allotted to this process"?

# Virtual Time and Working Set

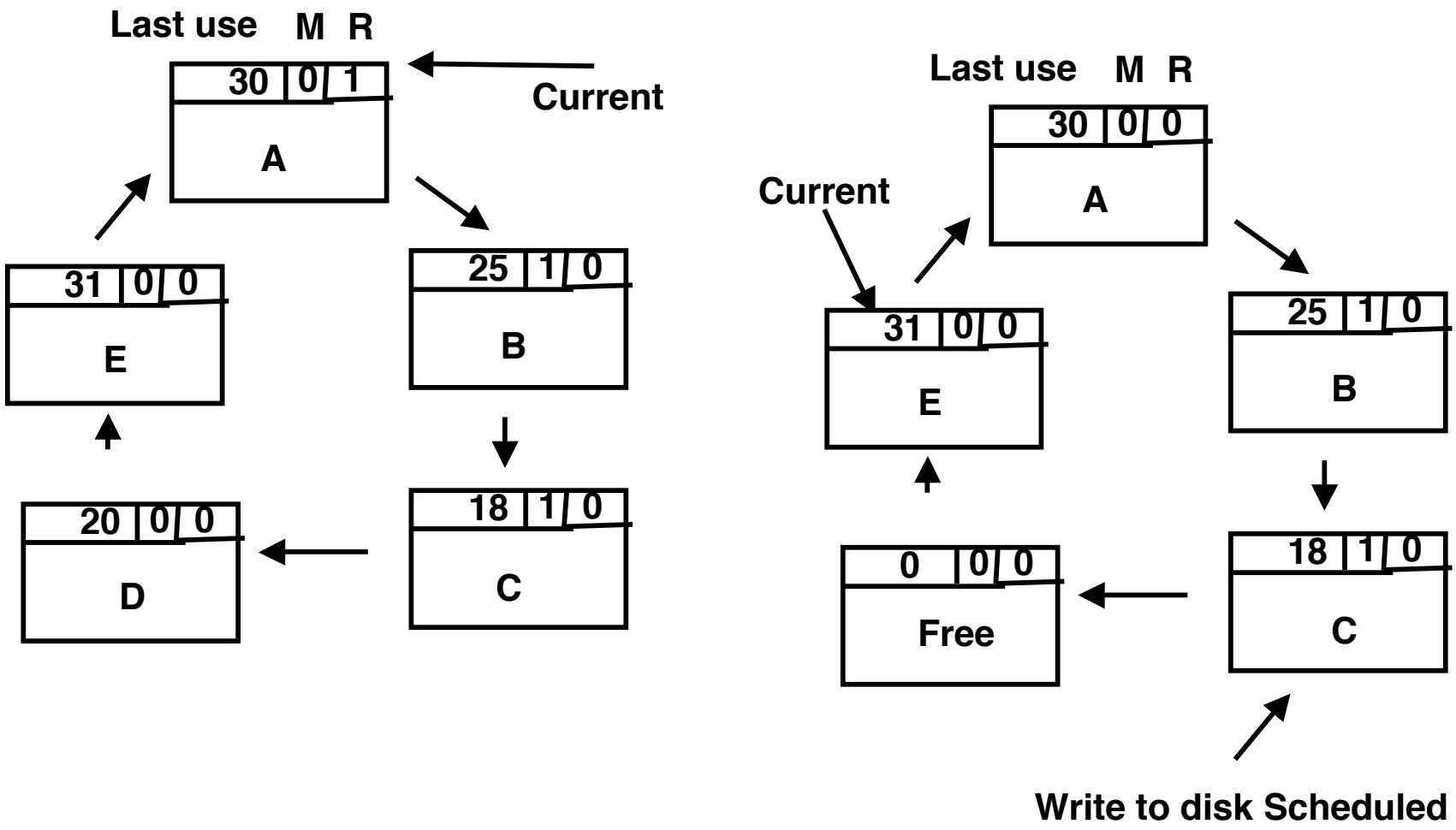
- Each process maintains a virtual time in its PCB entry
  - This counter should maintain the number of clock cycles that the process has been scheduled
- Each page table entry maintains time of last use (wrt to the process's virtual time)
- Upon every clock interrupt, if current process is P, then increment virtual time of P, and for all pages of P in memory, if R = 1, update "time of last use" field of the page to current virtual time of P
- Age of a page p of P = Current virtual time of P minus time of last use of p
- If age is larger than some threshold, then the page is not in the working set, and should be evicted

# WSClock Replacement Algorithm

- Combines working set with clock algorithm
- Each page table entry maintains modified bit M
- Each page table entry maintains reference bit R indicating whether used in the current clock cycle
- Each PCB entry maintains virtual time of the process
- Each page table entry maintains time of last use
- List of active pages of a process are maintained in a ring with a current pointer

# WSClock Algorithm

**Current Virtual Time = 32**
**Threshold for working set = 10**



# WSClock Algorithm

Maintain reference bit R and dirty bit M for each page

Maintain process virtual time in each PCB entry

Maintain Time of last use for each page (age=virtual time – this field)

To free up a page-frame, do:

- Examine page pointed by Current pointer
  - If R = 0 and Age > Working set window k and M = 0 then add this page to list of free frames
  - If R = 0 and M = 1 and Age > k then schedule a disk write, advance current, and repeat
  - If R = 1or Age <= k then clear R, advance current, and repeat
- If current makes a complete circle then
  - If some write has been scheduled then keep advancing current till some write is completed
  - If no write has been scheduled then all pages are in working set so pick a page at random (or apply alternative strategies)

# Page Replacement in Unix

- Unix uses a background process called paging daemon that tries to maintain a pool of free clean page-frames
- Every 250ms it checks if at least 25% (a adjustable parameter) frames are free
  - selects pages to evict using the replacement algorithm
  - Schedules disk writes for dirty pages
- Two-handed clock algorithm for page replacement
  - Front hand clears R bits and schedules disk writes (if needed)
  - Page pointed to by back hand replaced (if R=0 and M=0)

# UNIX and Swapping

- Under normal circumstances pager daemon keeps enough pages free to avoid thrashing. However, when the page daemon is not keeping up with the demand for free pages on the system, more drastic measures need be taken: swapper swaps out entire processes

- The swapper typically swaps out large, sleeping processes in order to free memory quickly. The choice of which process to swap out is a function of process priority and how long process has been in main memory. Sometimes ready processes are swapped out (but not until they've been in memory for at least 2 seconds).

- The swapper is also responsible for swapping in ready-to-run but swapped-out processes (checked every few seconds)

# Local Vs Global Policy

Paging algorithm can be applied either

1 locally: the memory is partitioned into "workspace", one for each process.

(a) equal allocation: if ***m*** frames and ***n*** processes then ***m/n*** frames.

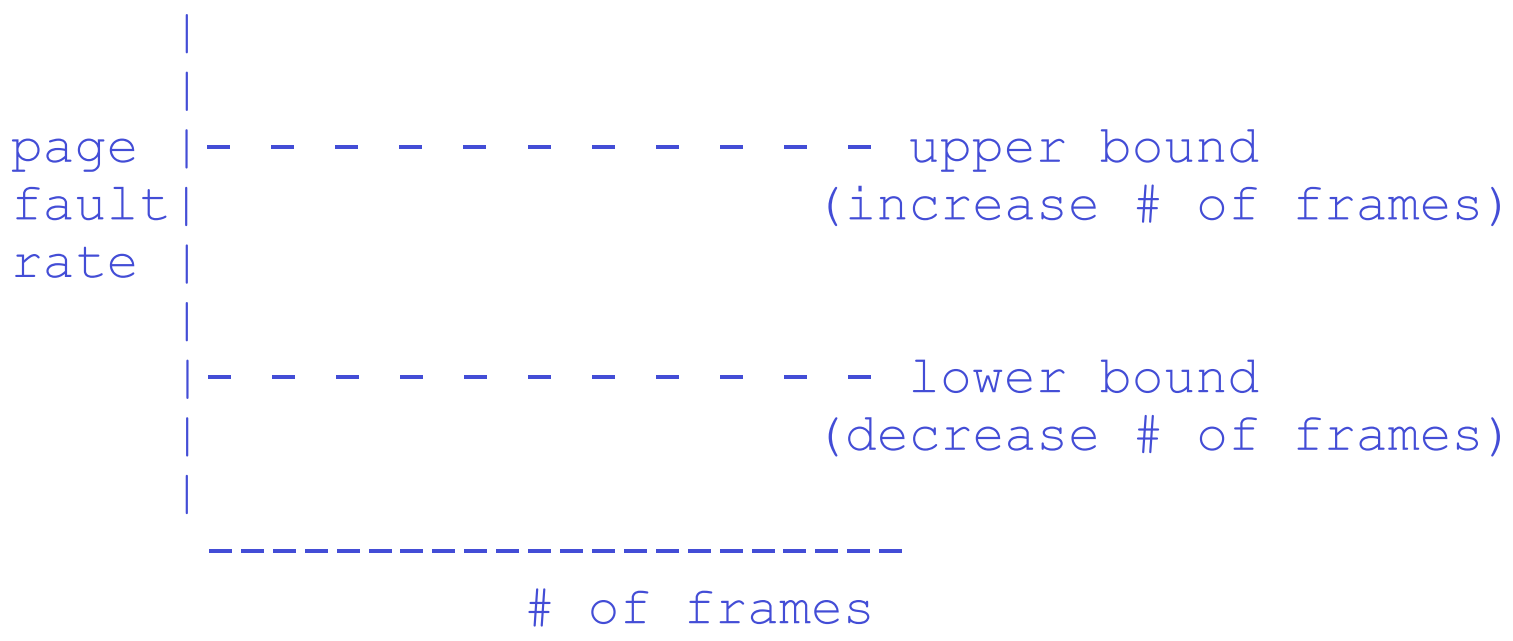
(b) proportional allocation: if ***m*** frames & ***n*** processes, let $s_i$ be the size of $P_i$.

$$S = \sum_{i=1}^{n} s_i$$

$$a_i = s_i / S \times m$$

2 globally: the algorithm is applied to the entire collection of running programs. Susceptible to thrashing (a collapse of performance due to excessive page faults).
Thrashing directly related to the degree of multiprogramming.

# PFF (page Fault Frequency)

- direct way to control page faults.
- 

- Program restructuring
  - to improve locality
  - at compile-time
  - at run-time
  - (using information saved during exec)

# Data structure on page faults

- ```
  int a[128][128]
  for (j=0, j<128, j++)
    for (i=0, i<128, i++)
      a[i,j]=0

  for (i=0, i<128, i++)
    for (j=0, j<128, j++)
      a[i,j]=0
  ```
- C row first
  FORTRAN column first
  ```

# What's a good page size ?

- OS has to determine the size of a page
  - Does it have to be same as size of a page frame (which is determined by hardware)? Not quite!
- Arguments for smaller page size:
  - Less internal fragmentation (unused space within pages)
  - Can match better with locality of reference
- Arguments for larger page size
  - Less number of pages, and hence, smaller page table
  - Less page faults
  - Less overhead in reading/writing of pages

# Page Size

1 (to reduce) table fragmentation ⇒ larger page

2 internal fragmentation ⇒ smaller page

3 read/write i/o overhead for pages ⇒ larger page

4 (to match) program locality (& therefore to reduce total i/o) ⇒ smaller page

5 number of page faults ⇒ larger page

# Thm. (Optimal Page Size)

- (wrt factors 1 & 2)
  Let $c_1$ = cost of losing a word to table fragmentation and $c_2$ = cost of losing a word to internal fragmentation.
  Assume that each program begins on a page boundary.
  If the avg program size $s_0$ is much larger than the page size $z$, then the optimal page size $z_0$ is approximately $\sqrt{2cs_0}$ where $c = c_1/c_2$.
- **Proof.**

$$\begin{aligned}
\text{int.}\mathit{frag}.\text{cost} &= c_2 z / 2 \\
\mathit{tablefrag}.\text{cost} &= c_1 s_0 / z \\
E[\text{cost}|z] &= c_1 s_0 / z + c_2 z / 2 \\
dE / ez &= -c_1 s_0 / z^2 + c_2 / 2 \\
0 &= -c_1 s_0 / z^2 + c_2 / 2 \\
c_1 s_0 &= c_2 z^2 / 2 \\
2c_1 / c_2 s_0 &= z^2 \\
z &= \sqrt{2c_1 / c_2 s_0}
\end{aligned}$$

# Page size examples

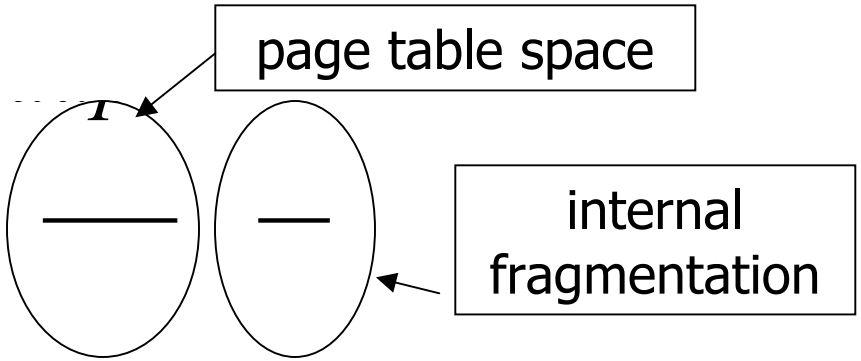
- $c_1=c_2=1$
- $z$ $s_0$ f=z/$s_0$× 100%

| $z$ | $s_0$ | f=z/$s_0$× 100% |
|---|---|---|
| 8 | 32 | 25 |
| 16 | 128 | 13 |
| 32 | 512 | 6 |
| 64 | 2K | 3 |
| 128 | 8K | 1.6 |
| 256 | 32K | .8 |
| 512 | 128K | .4 |
| 1024 | 512K | .2 |

- $c_1 > c_2 \Rightarrow$ larger page than above (need cache)
  $c_1 < c_2$ (unlikely) $\Rightarrow$smaller
- GE645 64 word & 1024 word pages
  IBM/370 2K & 4K
  VAX 512bytes
  Berkeley Unix 2 x 512 = 1024

# Page Size: Tradeoff

Overhead due to page table and internal fragmentation




where

- s = average process size in bytes
- p = page size in bytes
- e = page entry

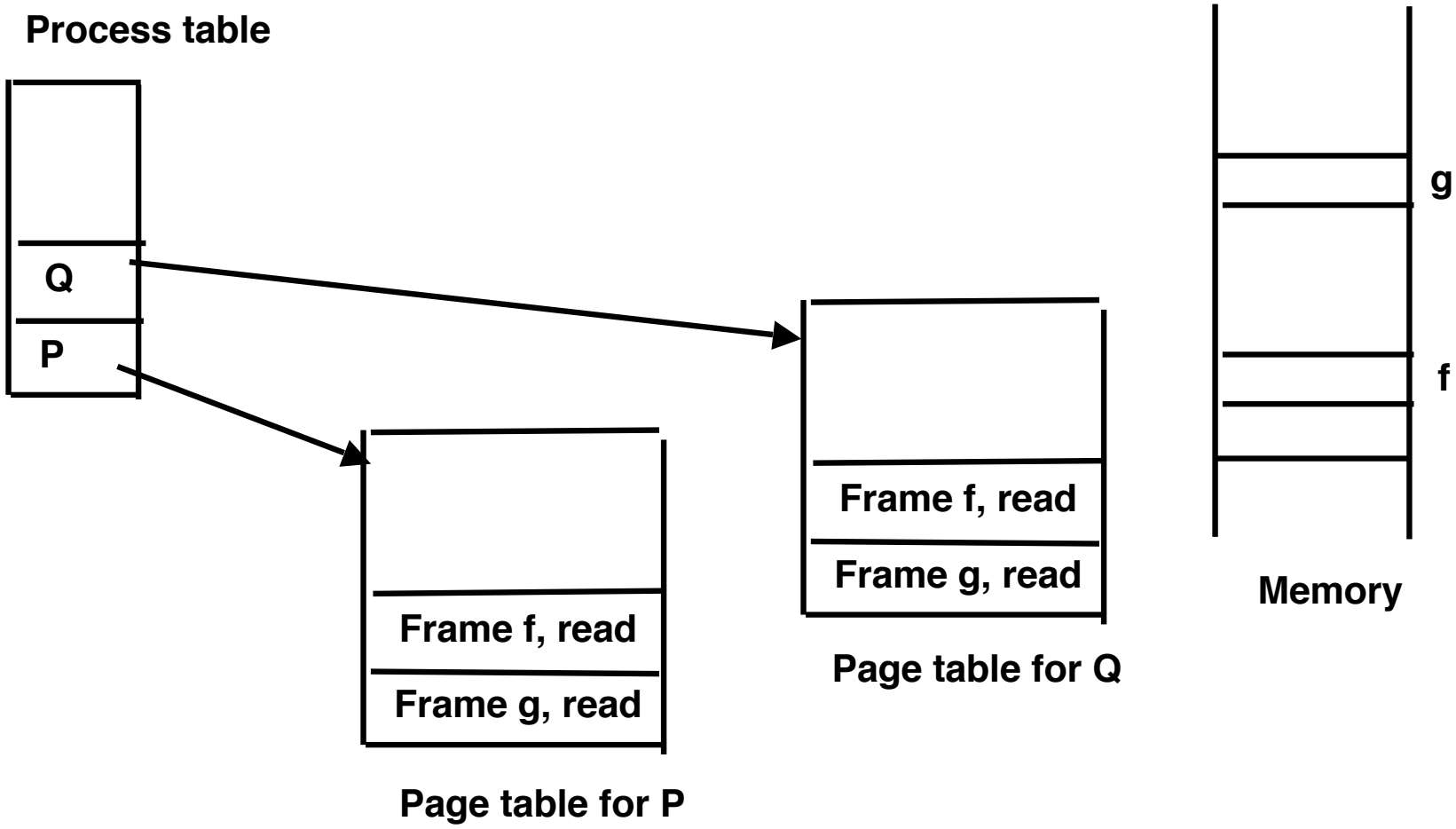
Optimized when

If s = 4 MB, and e = 8B, then p = 8 KB

# Sharing of Pages

- Can two processes share pages (e.g. for program text)
- Solution in PDP-11:
  - Use separate address space and separate page table for instructions (I space) and data (D space)
  - Two programs can share same page tables in I space
- Alternative: different entries can point to the same page
  - Careful management of access writes and page replacement needed
- In most versions of Unix, upon fork, parent and child use same pages but have different page table entries
  - Pages initially are read-only
  - When someone wants to write, traps to kernel, then OS copies the page and changes it to read-write (copy on write)

# Shared Pages with separate page tables

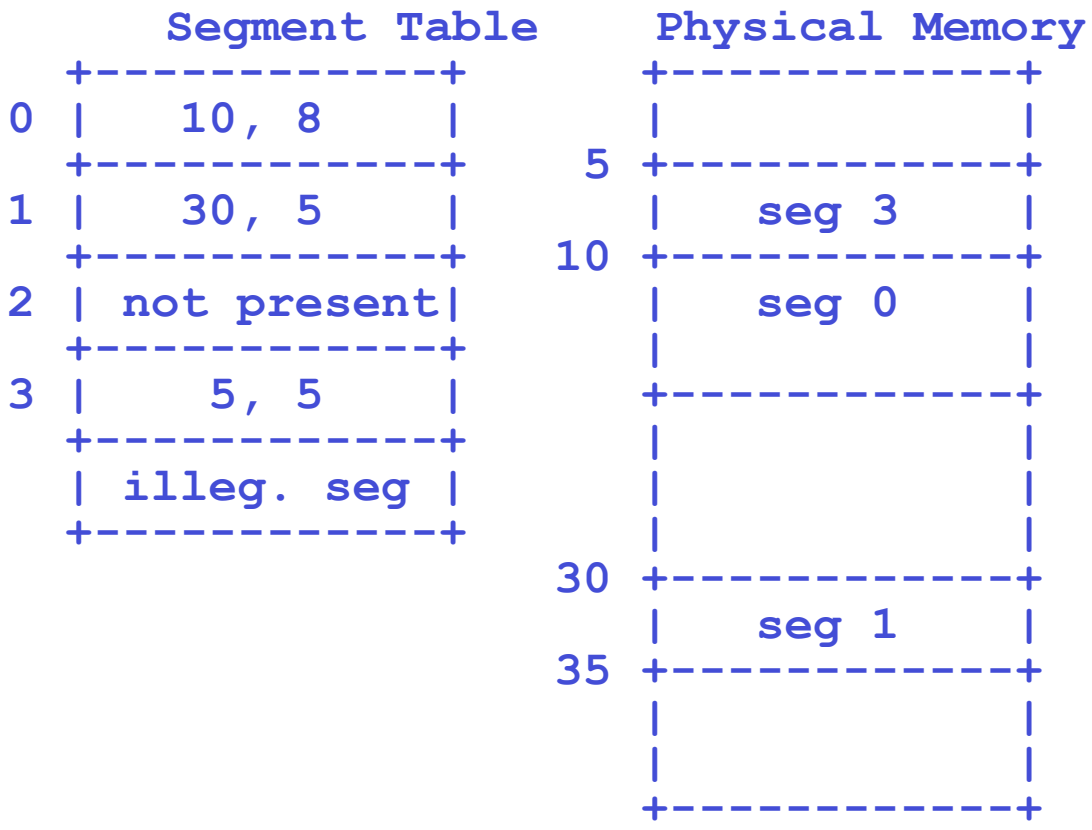



Two processes sharing same pages with “copy on write”

# Segmentation

- Recall: Paging allows mapping of virtual addresses to physical addresses and is transparent to user or to processes
- Orthogonal concept: Logical address space is partitioned into logically separate blocks (e.g. data vs code) by the process (or by the compiler) itself
- Logical memory divided into segments, each segment has a size (limit)
- Logical address is (segment number, offset within seg)
- Note: Segmentation can be with/without virtual memory and paging
- Conceptual similarity to threads: threads is a logical organization within a process for improving CPU usage, segments are for improving memory usage

# Implementation without Paging




First few bits of address give segment

Segment table keeps Base, Limit

| logical address | | physical address |
|---|---|---|
| 0,2 | ------> | 12 |
| 1,4 | ------> | 34 |
| 0,9 | ------> | illegal offset |
| 2,1 | ------> | absent seg. |

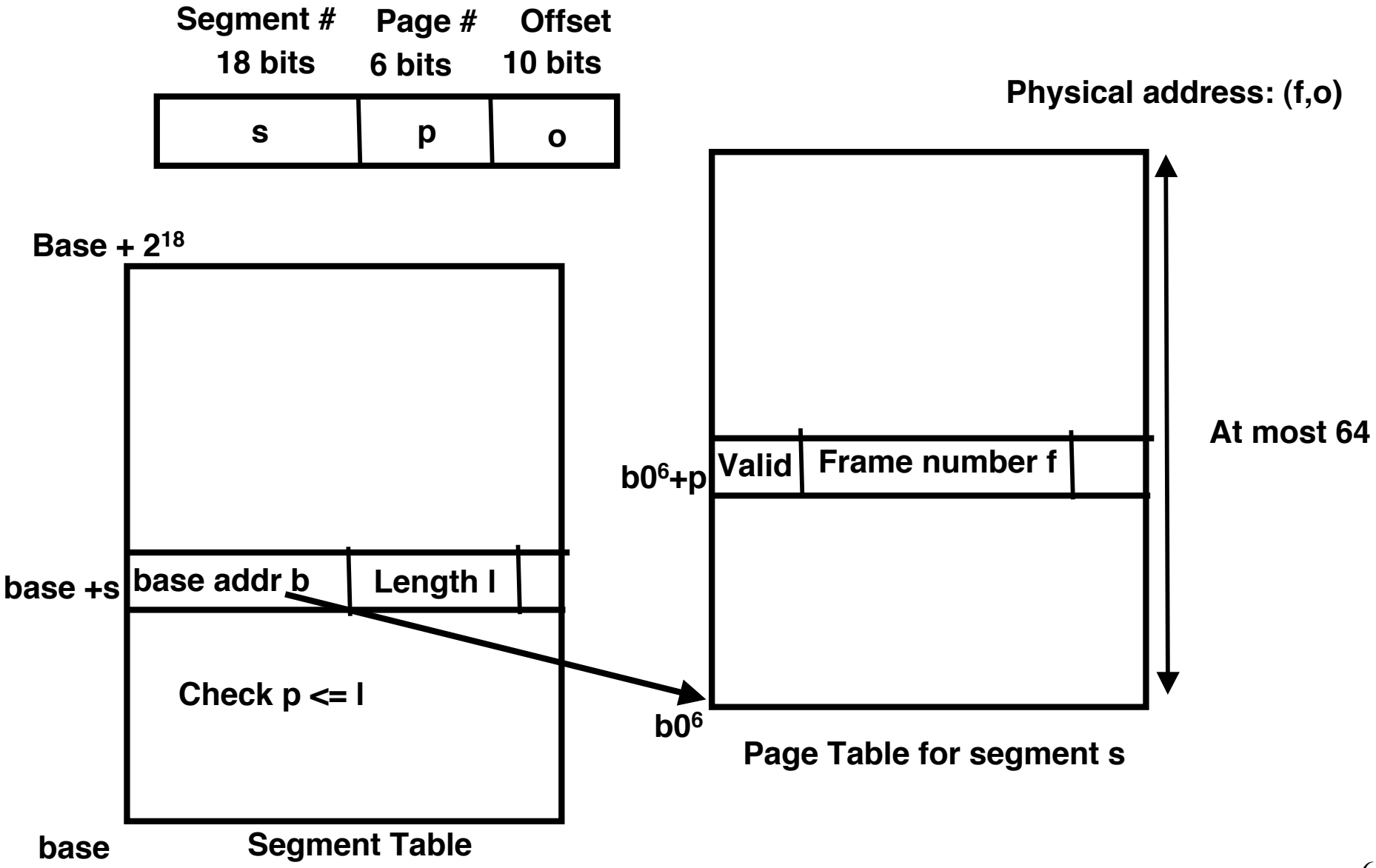
# Advantages

- Address allocation is easy for compiler
- Different segments can grow/shrink independently
- Natural for linking separately compiled code without worrying about relocation of virtual addresses
  - Just allocate different segments to different packages
- Application specific
  - Large arrays in scientific computing can be given their own segment (array bounds checking redundant)
- Natural for sharing libraries
- Different segments can have different access protections
  - Code segment can be read-only
- Different segments can be managed differently

# Segmentation with Paging

- Address space within a segment can be virtual, and managed using page tables
  - Same reasons as we saw for non-segmented virtual memory
- Two examples
  - Multics
  - Pentium
- Steps in address translation
  1. Check TLB for fast look-up
  2. Consult segment table to locate segment descriptor for s
  3. Page-table lookup to locate the page frame (or page fault)

# Multics: Segmentation+ Paging



# Multics Memory

- 34-bit address split into 18-bit segment no, and 16 bit (virtual) address within the segment
- Thus, each segment has 64K words of virtual memory
- Physical memory address is 24 bits, and each page frame is of size 1K
- Address within segment is divided into 6-bit page number and 10-bit offset
- Segment table has potentially 256K entries
- Each segment entry points to page table that contains upto 64 entries

# Multics details cont

- Segment table entry is 36 bits consisting of
  - main memory address of page table (but only 18 bits needed, last 6 bits assumed to be 0)
  - Length of segment (in terms of number of pages, this can be used for a limit check)
  - Protection bits
- More details
  - Different segments can have pages of different sizes
  - Segment table itself can be in a segment itself (and can be paged!)
- Memory access first has to deal with segment table and then with page table before getting the frame
- TLBs absolutely essential to make this work!

# Pentium

- 16K segments divided into LDT and GDT (Local/Global Descriptor Tables)
- Segment selector: 16 bits
  - 1 bit saying local or global
  - 2 bits giving protection level
  - 13 bits giving segment number
- Special registers on CPU to select code segment, data segment etc
- Incoming address: (selector,offset)
- Selector is added to base address of segment table to locate segment descriptor
- Phase 1: Use the descriptor to get a “linear” address
  - Limit check
  - Add Offset to base address of segment

# Paging in Pentium

- Paging can be disabled for a segment
- Linear virtual address is 32 bits, and each page is 4KB
- Offset within page is 12 bits, and page number is 20 bits. Thus, $2^{20}$ pages, So use 2-level paging
- Each process has page directory with 1K entries
- Each page directory entry points to a second-level page table, in turn with 1K entries (so one top-level entry can cover 4MB of memory)
- TLB used
- Many details are relevant to compatibility with earlier architectures