Operating Systems

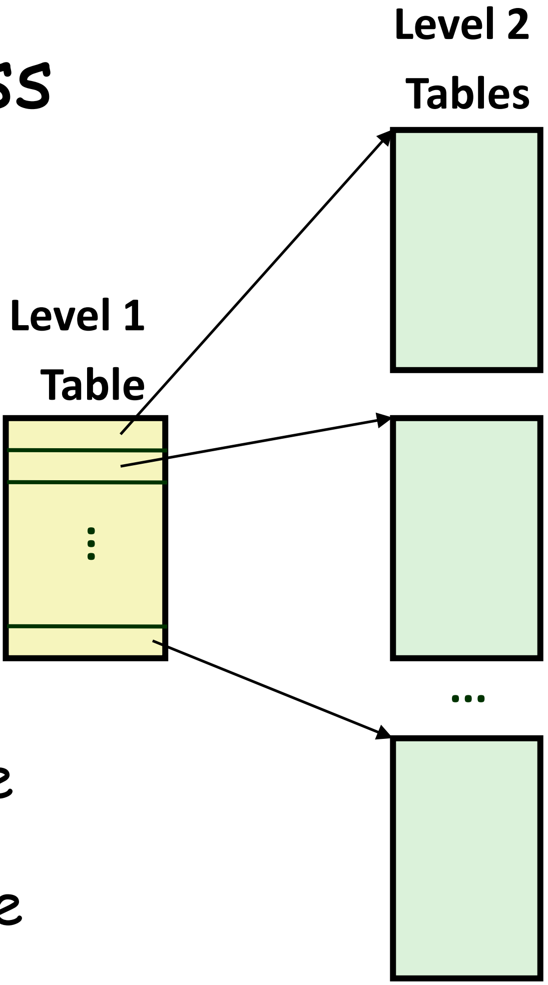
Memory Management III

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Tackling The Page Table Size Problem

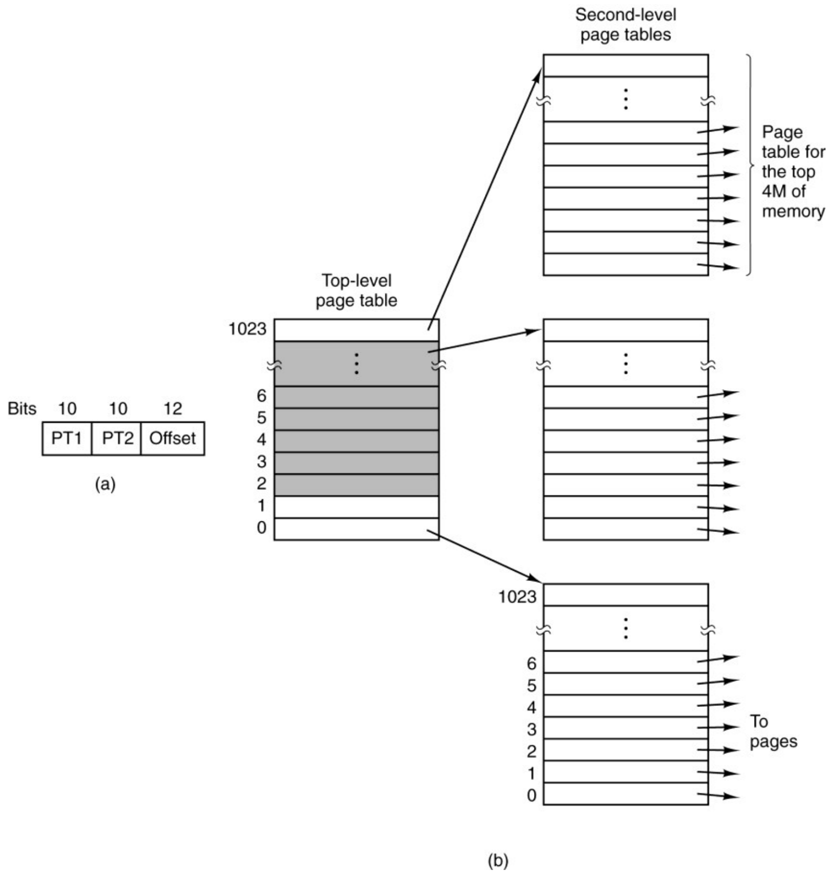
Reduce Page Table Size

- Assume: 4KB-page, 48-bit address space, and 8-byte PTE
- Size of page table needed?
  - $2^{48-12} * 2^3 = 2^{39} = 512$ GB
- Wasteful: most PTEs are invalid
  - i.e. not used
- Solution: multi-level page table
  - Example: 2-level page table
    - Level 1 table: each PTE points to a page table
    - Level 2 table: each PTE points to a page



Multi-Level Page Table

- To reduce storage overhead in case of large memories



Why Two-level Page Table Reduces Memory Requirement?

- If a PTE in the level 1 table is null, then the corresponding level 2 page table does not even have to exist.
- Only the level 1 table needs to be in main memory at all times.
- The level 2 page tables can be created and paged in and out by the VM system as they are needed.

Inverted Page Table

- Page table has one entry per page frame not one entry per page.
  - For each page slot (frame) it shows which virtual page it has.

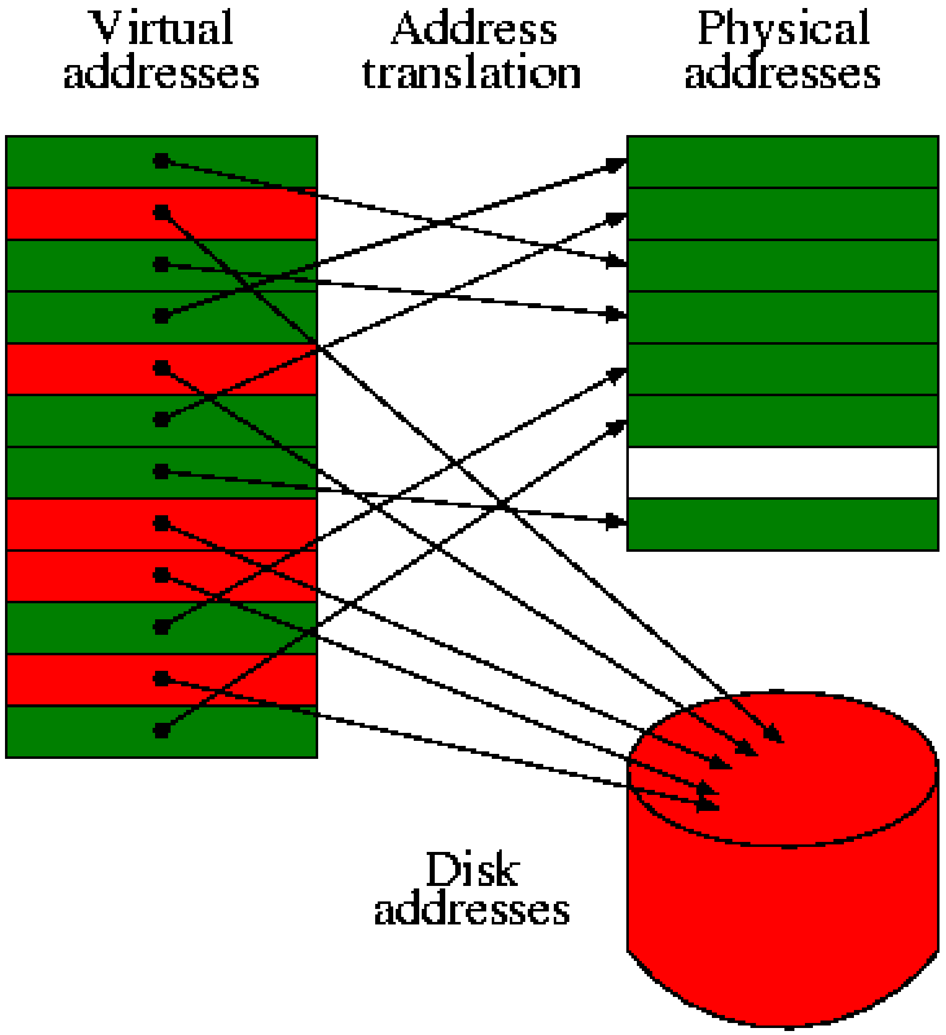
+ Save vast amount of storage

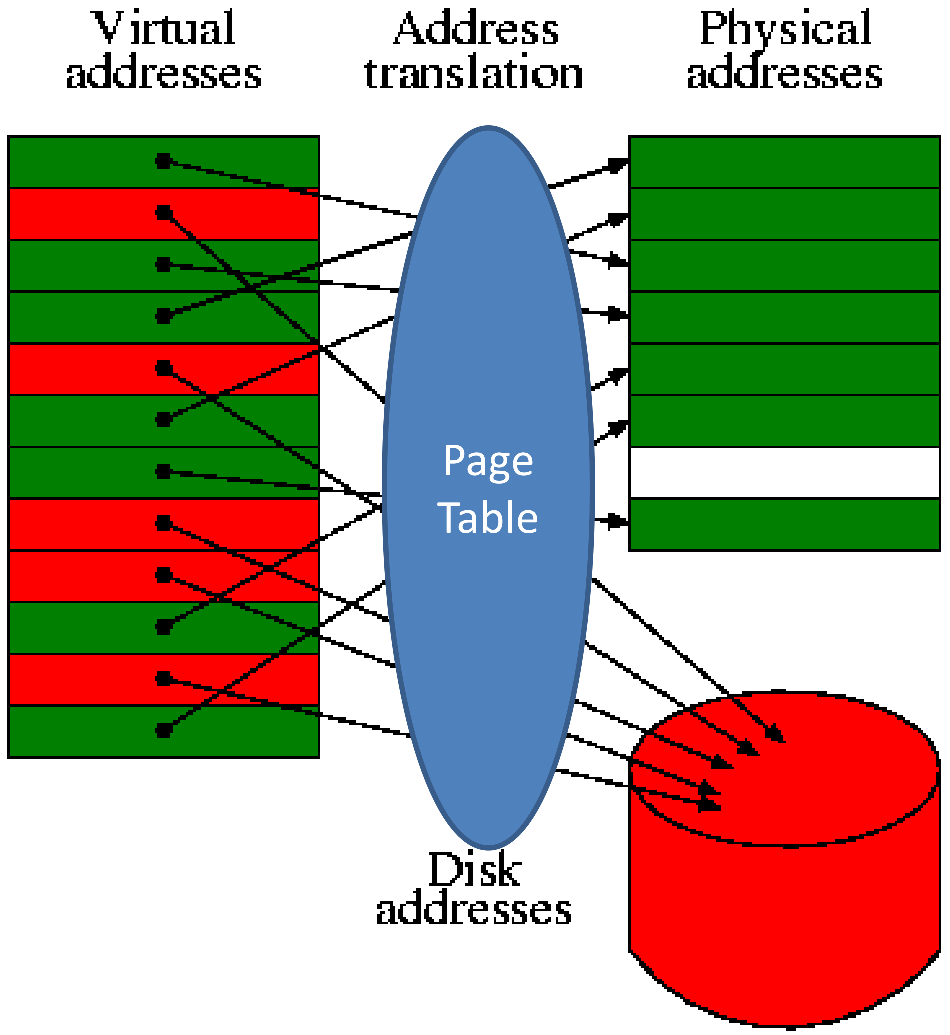
- virtual-to-physical translation much harder

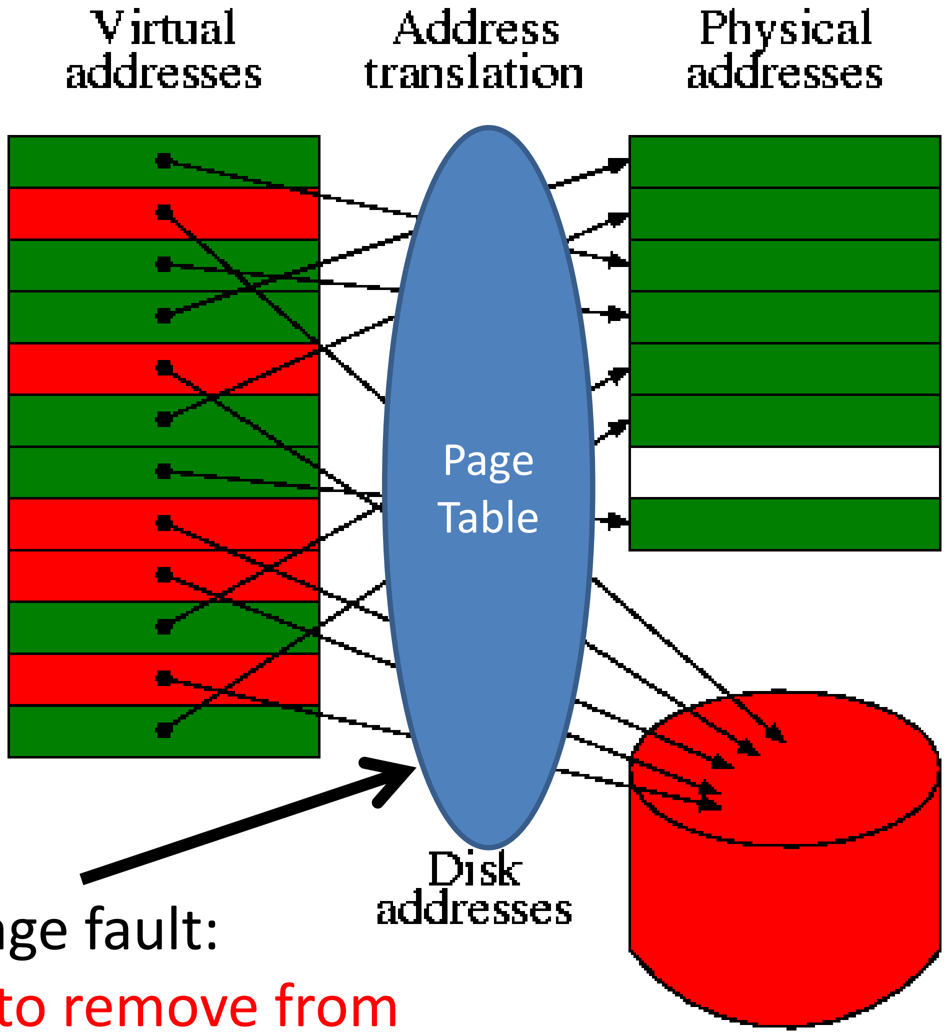
What About Page Faults?

Definition

- A Page fault is when the page table is consulted, and we found that the requested page is not in the memory but in the disk.
  - That is, the page has been swapped out because the memory was full.
- Note: In reality, the OS does not wait till memory is full to start making page replacement because the OS wants to have some spare memory free just in case.


Virtual addresses
Address translation
Physical addresses
Disk addresses


Virtual addresses
Address translation
Physical addresses
Page Table
Disk addresses



In case of page fault:

which page to remove from Memory if the memory is full?

Replacement Policies

- Used in many contexts when storage is not enough
  - caches
  - web servers
  - pages
- Things to consider when designing a replacement policy
  - measure of success
  - cost

Optimal Page Replacement Algorithm

- Each page labeled with the number of instructions that will be executed before this page is referenced.
- Page with the highest label should be removed.
- Impossible to implement because we do not know the future.

The Not Recently Used (NRU) Replacement Algorithm

- Two status bits with each page
  - R: Set whenever the page is referenced (used)
  - M: Set when the page is written
- R and M bits are available in most computers implementing virtual memory.
- Those bits are updated with each memory reference
  - Must be updated by hardware
  - Reset only by the OS
- Periodically the R bit is cleared
  - To distinguish pages that have been referenced recently

The Not Recently Used (NRU) Replacement Algorithm

	R	M
Class 0:	0	0
Class 1:	0	1
Class 2:	1	0
Class 3:	1	1

NRU algorithm removes a page at random from the lowest numbered nonempty class

The FIFO Replacement Algorithm

- OS maintains a list of the pages currently in memory.
- The most recent arrival at the tail.
- On a page fault, the page at the head is removed.

The Second-Chance Page Replacement Algorithm

- Modification to FIFO
- Inspect the R bit of the oldest page
  - If R=0 page is old and unused -> replace
  - If R=1 then
    - bit is cleared
    - page is put at the end of the list
    - the search continues
- If all pages have R=1, the algorithm degenerates to FIFO

The Least Recently Used (LRU) Page Replacement Algorithm

- Good approximation to optimal
- When page fault occurs, replace the page that has been unused for the longest time.
- Realizable but not cheap

LRU
Hardware Implementation 1

- 64-bit counter increment after each instruction
- Each page table entry has a field large enough to include the value of the counter
- After each memory reference to a page, the counter is incremented in the entry's counter field.
- At page fault, the page with lowest value is discarded

LRU
Hardware Implementation 1

- 64-bit counter increment after each instruction
- Each page table entry has a field large enou unter
- Afte age, the coun ter field.
- At page fault, the page with lowest value is discarded

Too expensive!

Too Slow!

LRU: Hardware Implementation 2

- Machine with n page frames
- Hardware maintains a matrix of nxn bits
- Matrix initialized to all 0s
- Whenever page frame k is referenced
  - Set all bits of row k to 1
  - Set all bits of column k to 0
- The row with lowest value is the LRU

LRU: Hardware Implementation 2

Page

$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 \\ 3 & 0 & 0 & 0 & 0 \end{array}$$

(a)

Page

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{array}$$

(b)

Page

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{array}$$

(c)

Page

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \end{array}$$

(d)

Page

$$\begin{array}{cccc} 0 & 1 & 2 & 3 \\ \hline 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 0 \end{array}$$

(e)

$$\begin{array}{|c|c|c|c|} \hline 0 & 0 & 0 & 0 \\ \hline 1 & 0 & 1 & 1 \\ \hline 1 & 0 & 0 & 1 \\ \hline 1 & 0 & 0 & 0 \\ \hline \end{array}$$

(f)

$$\begin{array}{|c|c|c|c|} \hline 0 & 1 & 1 & 1 \\ \hline 0 & 0 & 1 & 1 \\ \hline 0 & 0 & 0 & 1 \\ \hline 0 & 0 & 0 & 0 \\ \hline \end{array}$$

(g)

$$\begin{array}{|c|c|c|c|} \hline 0 & 1 & 1 & 0 \\ \hline 0 & 0 & 1 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 1 & 1 & 1 & 0 \\ \hline \end{array}$$

(h)

$$\begin{array}{|c|c|c|c|} \hline 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 1 & 1 & 0 & 1 \\ \hline 1 & 1 & 0 & 0 \\ \hline \end{array}$$

(i)

$$\begin{array}{|c|c|c|c|} \hline 0 & 1 & 0 & 0 \\ \hline 0 & 0 & 0 & 0 \\ \hline 1 & 1 & 0 & 0 \\ \hline 1 & 1 & 1 & 0 \\ \hline \end{array}$$

(j)

Pages referenced: 0 1 2 3 2 1 0 3 2 3

LRU Implementation

- Slow
- Few machines have required hardware

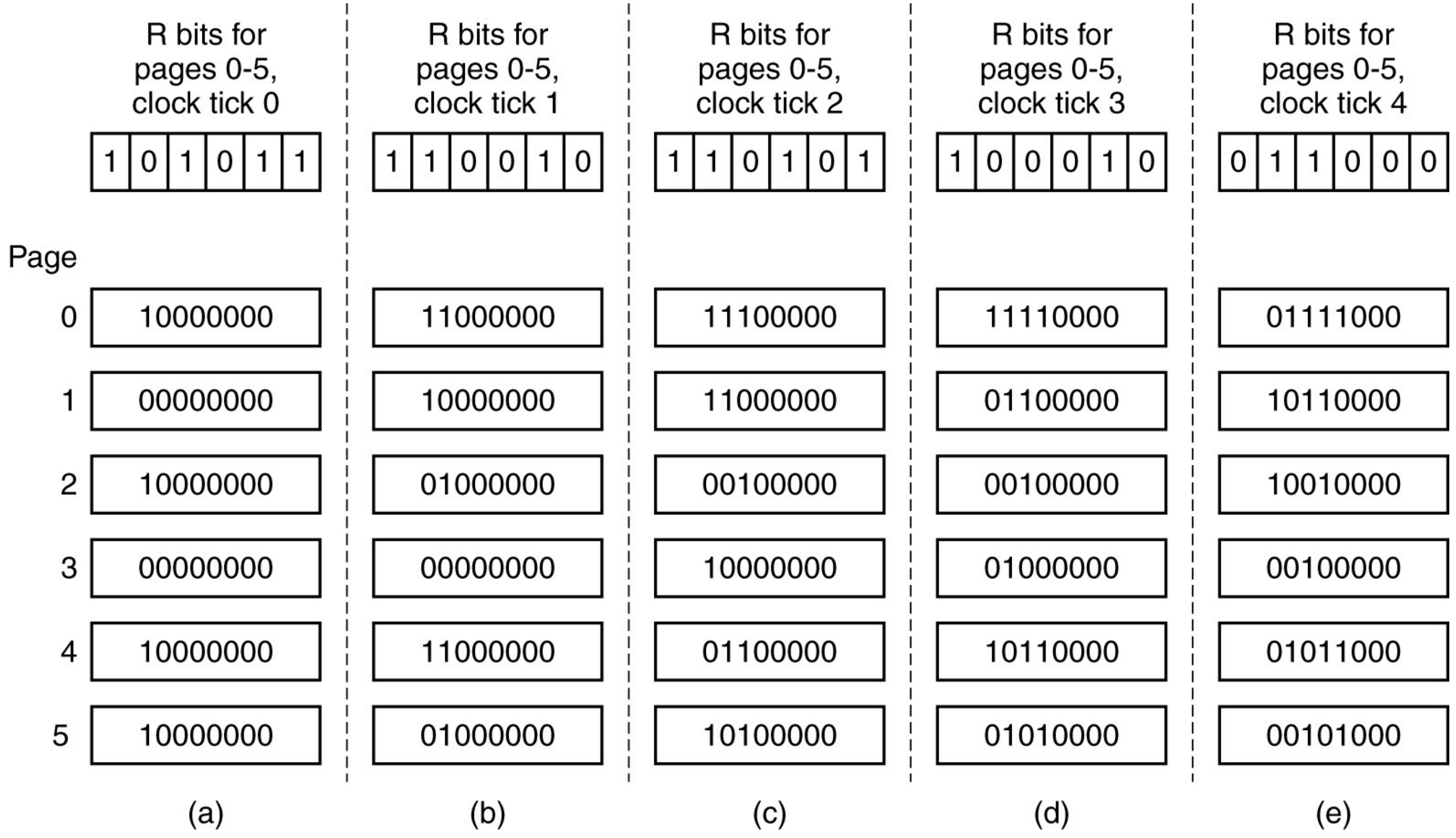
Simulating LRU in Software

- Not Frequently Used (NFU) algorithm
- Software counter associated with each page, initially zero
- At each clock interrupt, the OS scans all pages and add the R bit of each page to the counter of that page.
- At page fault: the page with lowest counter is replaced

Enhancing NRU

- NRU never forgets anything -> high inertia
- Modifications:
  - shift counter right 1 bit before adding R
  - R is added to the leftmost
- This modified algorithm is called aging
- The page whose counter is lowest is replaced at page fault

Aging Algorithm



	(a)	(b)	(c)	(d)	(e)
R bits for pages 0-5	clock tick 0	clock tick 1	clock tick 2	clock tick 3	clock tick 4
	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
Page 0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00100000	10010000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000

The Working Set Model

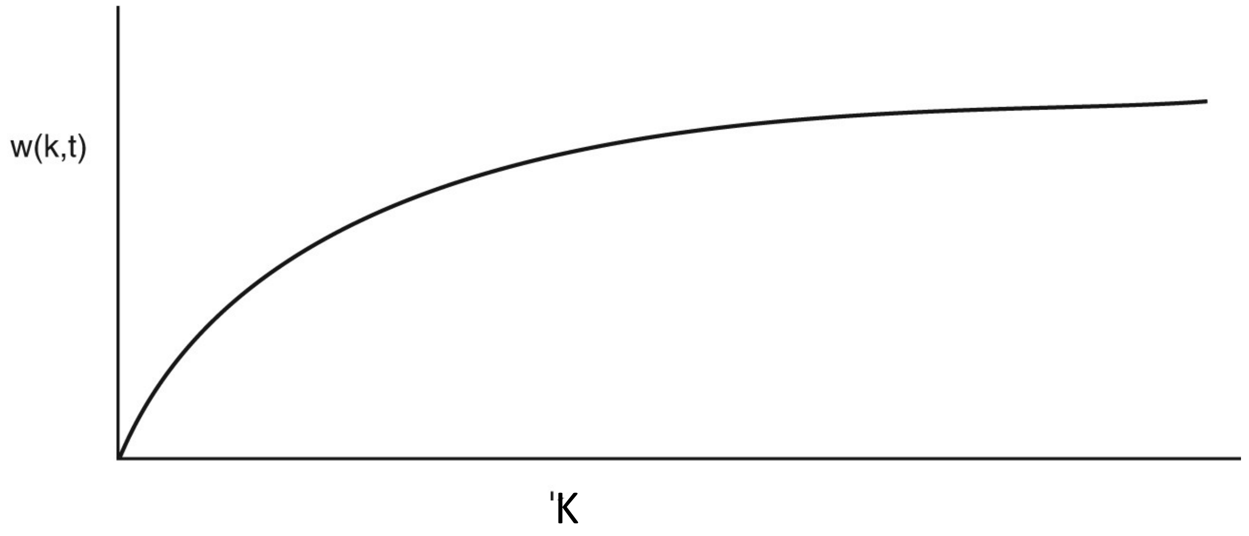
- Working set: the set of pages that a process is currently using
- Thrashing: a program causing page faults every few instructions

An important question:

In multiprogramming systems, processes are sometimes swapped to disk (i.e. all their pages are removed from memory). When they are brought back, which pages to bring?

The Working Set Model

- Try to keep track of each process' working set and make sure it is in memory before letting the process run.



w(k,t): the set of pages accessed in the last k references at instant t

The Working Set Model

- OS must keep track of which pages are in the working set
- Replacement algorithm: evict pages not in the working set
- Possible implementation (but expensive):
  - working set = set of pages accessed in the last k memory references
- Approximations
  - working set = pages used in the last 100 msec

Some Design Issues

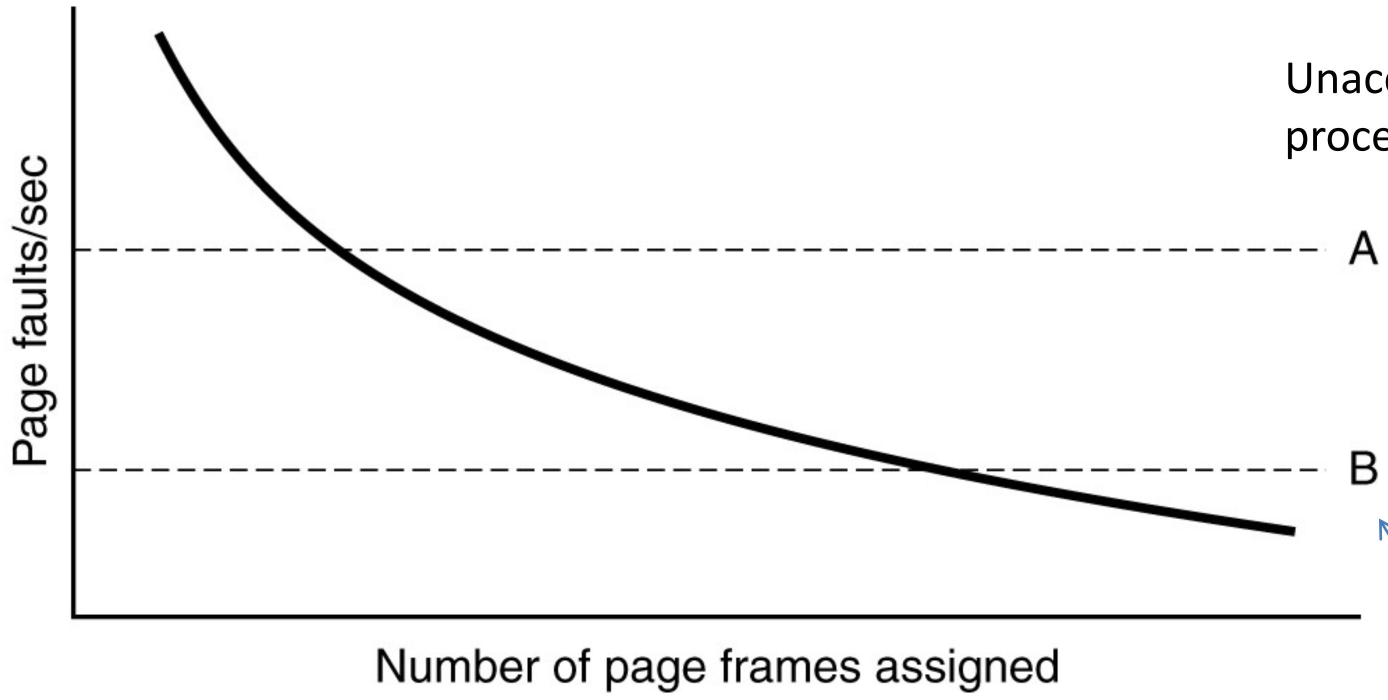
Design Issues for Paging: Local vs Global Allocation

- How memory should be allocated among the competing runnable processes?
- Local algorithms: allocating every process a fixed fraction of the memory
- Global algorithms: dynamically allocate page frames
- Global algorithms work better
  - If local algorithm used and working set grows → thrashing will result
  - If working set shrinks → local algorithms waste memory

Global Allocation

- **Method 1:** Periodically determine the number of running processes and allocate each process an equal share
- **Method 2** (better): Pages allocated in proportion to each process total size (in terms of number of pages).
- **Page Fault Frequency (PFF) algorithm**: tells when to increase/decrease page allocation for a process but says nothing about which page to replace.

Global Allocation: PFF



Unacceptable high
process given more pages

so low, process
has too much memory
page frames may be taken away

Design Issues: Load Control

- What if PFF indicates that some processes need more memory but none need less?
- Swap some processes to disk and free up all the pages they are holding.
- Which process(es) to swap?
  - Strive to make CPU busy (I/O bound vs CPU bound processes)
  - Process size

Design Issues: Page Size

- Large page size → internal fragmentation
- Small page size →
  - larger page table
  - More overhead transferring from disk

Design Issues: Page Size

- Assume:
  - s = process size (in bytes)
  - p = page size (in bytes)
  - e = size of each page table entry (in bytes)
- So:
  - number of pages needed = s/p
  - occupying: se/p bytes of page table space
  - wasted memory due to fragmentation: $p/2$
  - overhead = $se/p + p/2$
- We want to minimize the overhead:
  - Take derivative of overhead and equate to 0:
  - $-se/p^2 + \frac{1}{2} = 0 \rightarrow p = \sqrt{2se}$

Design Issues: Shared Libraries

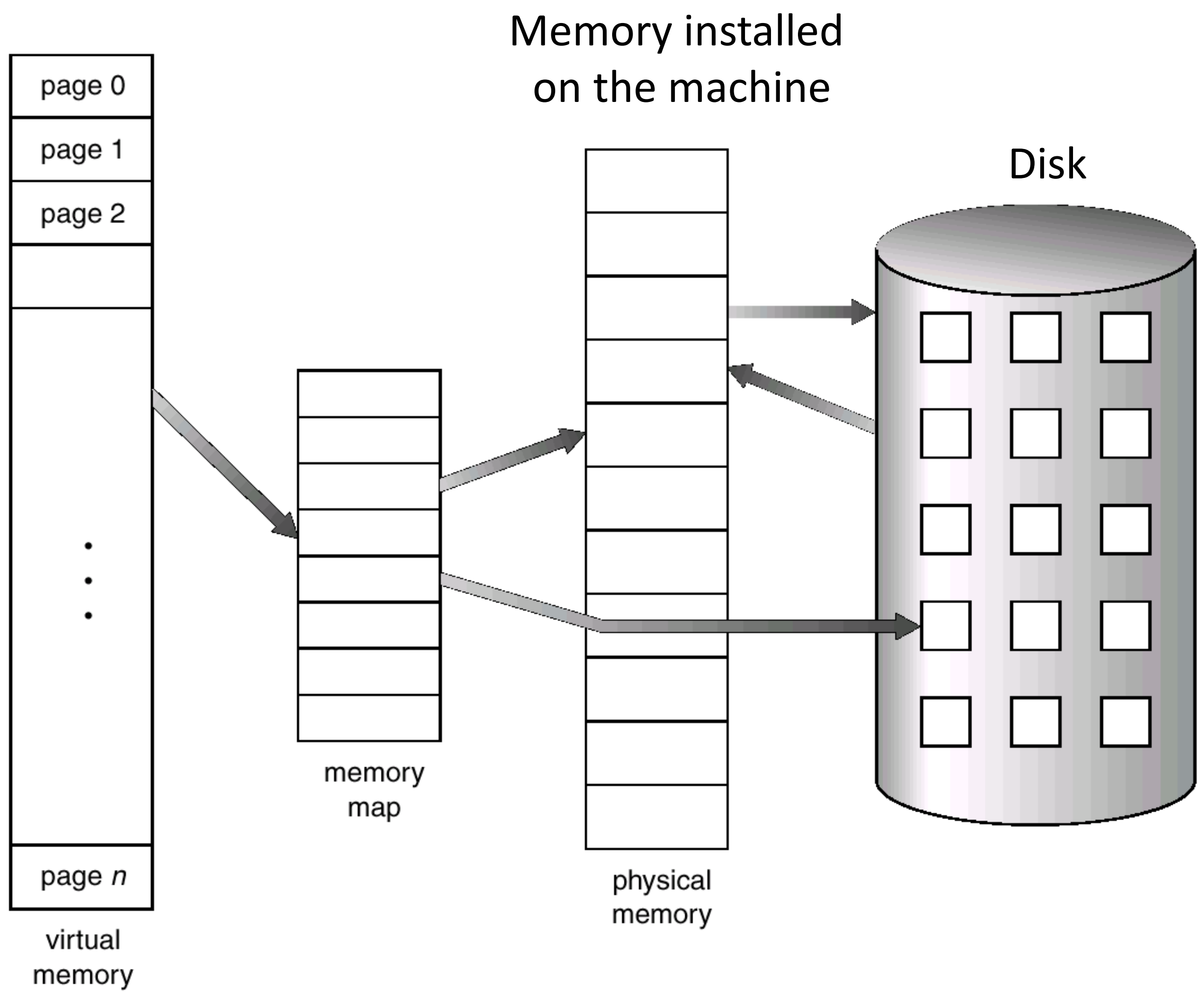
- Dynamically linked
- Loaded when program is loaded or when functions in them are called for the first time

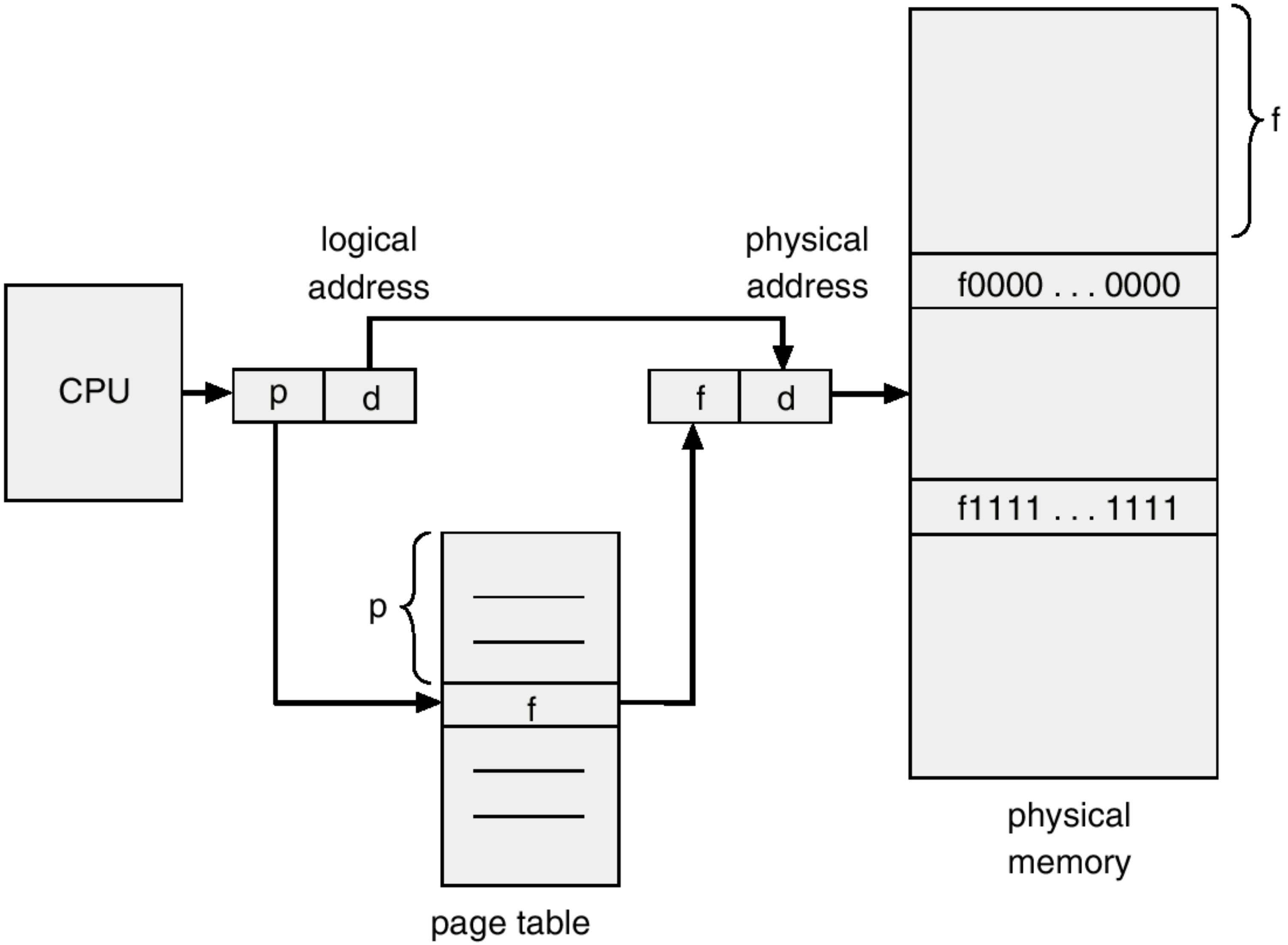
Design Issues: Cleaning Policy

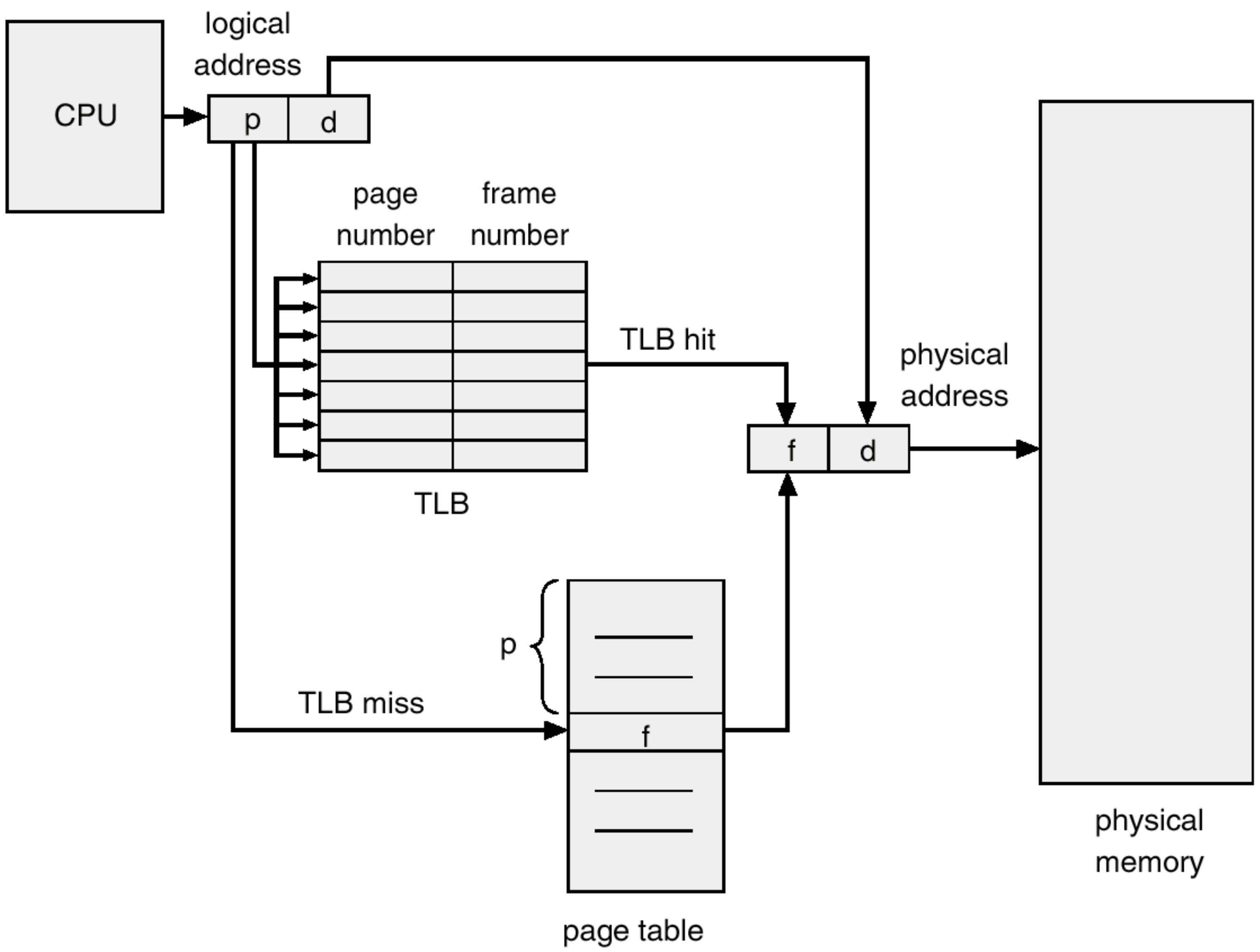
- Paging daemon
- Sleeps most of the time
- Awakened periodically to inspect state of the memory
- If too few page slots are free -> daemon begins selecting pages to evict

Summary of Paging

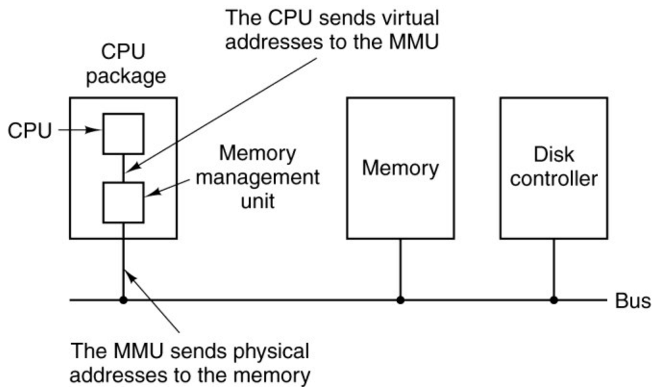
- Virtual address space bigger than physical memory
- Mapping virtual address to physical address
- Virtual address space divided into fixed-size units called pages
- Physical address space divided into fixed-size units called pages frames
- Virtual address space of a process can be non-contiguous in physical address space


Memory installed on the machine
Disk
page 0
page 1
page 2
page n
virtual memory
memory map
physical memory

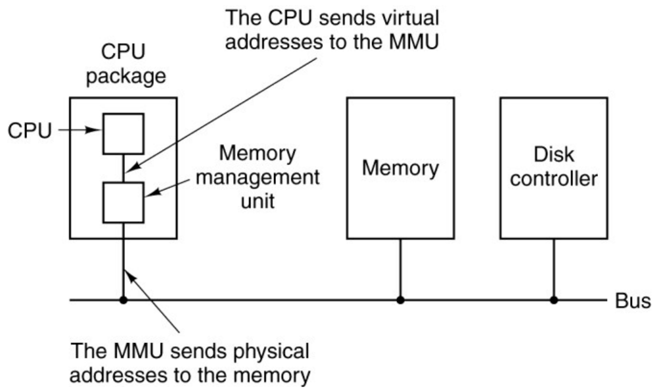

CPU
logical address
p
d
physical address
f
d
p
f
page table
f
f0000 . . . 0000
f1111 . . . 1111
physical memory


CPU
logical address
p
d
page number
frame number
TLB hit
TLB
physical address
f
d
p
TLB miss
f
page table
physical memory

$$\begin{tikzcd}
& \text{Paging} \arrow[dl] \arrow[dr] & \\
\text{MMU} & & \text{OS Involvement}
\end{tikzcd}$$


The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

$$\begin{tikzcd} & \text{Paging} \arrow[dl] \arrow[dr] & \\ \text{MMU} & & \text{OS Involvement} \end{tikzcd}$$


The CPU sends virtual addresses to the MMU
CPU package
CPU
Memory management unit
Memory
Disk controller
Bus
The MMU sends physical addresses to the memory

?

OS Involvement With Paging

- When a new process is created
- When a process is scheduled for execution
- When process exits
- When page fault occurs

OS Involvement With Paging

- When a new process is created
  - Determine how large the program and data will be (initially)
  - Create page table
  - Allocate space in memory for page table
  - Record info about page table and swap area in process table
- When a process is scheduled for execution
- When process exits
- When page fault occurs

OS Involvement With Paging

- When a new process is created
- When a process is scheduled for execution
- When process exits
- When page fault occurs

OS Involvement With Paging

- When a new process is created
- When a process is scheduled for execution
  - MMU initialized for the process
  - TLB flushed
  - Process table made current
- When process exits
- When page fault occurs

OS Involvement With Paging

- When a new process is created
- When a process is scheduled for execution
- When process exits
  - OS releases the process page table
  - Frees its pages and disk space
- When page fault occurs

OS Involvement With Paging

- When a new process is created
- When a process is scheduled for execution
- When process exits
- When page fault occurs

Page Fault Handling

1. The hardware:
   - Saves program counter
   - Switches to kernel mode
2. An assembly routine saves general registers and calls OS
3. OS tries to discover which virtual page is needed
4. OS checks address validation and protection and assign a page frame (page replacement may be needed)

Page Fault Handling

5. If page frame selected is dirty
   - Page scheduled to transfer to disk
   - Frame marked as busy
   - OS suspends the current process
   - Context switch takes place
6. Once the page frame is clean
   - OS looks up disk address where needed page is
   - OS schedules a disk operation
   - Faulting process still suspended
7. When disk interrupts indicates page has arrived
   - OS updates page table

Page Fault Handling

8. Process is scheduled for execution.
9. The OS reloads registers and other state information and returns to user space.

Interesting Scenario: Virtual Memory & I/O Interaction

- Process issues a syscall to read a file into a buffer
- Process suspended while waiting for I/O
- New process starts executing
- This other process gets a page fault
- If paging algorithm is global there is a chance the page containing the buffer of the first process be removed from memory.
- The I/O operation of the first process will write some data into the buffer and some other on the just-loaded page of the second process!

Interesting Scenario: Virtual Memory & I/O Interaction

- Process issues a syscall to read a file into a buffer
- Process suspended while waiting for I/O
- New process starts executing
- This other process gets a page fault
- If paging algorithm is global there is a chance the page containing the buffer of the first process be removed from memory.
- The I/O operation of the first process will write some data into the buffer and some other on the just-loaded page of the second process!

One solution: Locking (pinning) pages engaged in I/O so that they will not be removed.

Backing Store (i.e. the disk)

- Swap area: must not be accessed by user.
- Associated with each process the disk address of its swap area; stored in the process table
- Before process starts swap area must be initialized
  - One way: copy all process image into swap area [static swap area]
  - Another way: don't copy anything and let the process swap out [dynamic]
- Instead of disk partition, one or more preallocated files within the normal file system can be used [Windows uses this approach.]

Conclusions

- Memory management has two goals:
  - Protecting processes from each other.
  - Give each process the illusion that it has the whole memory as a huge continuous address space for itself.
- Technique used to do that: paging