OS Background

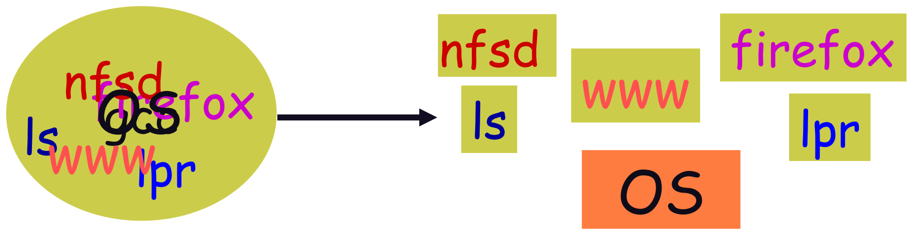
Dominic Duggan
CS526 Enterprise & Cloud Computing
Stevens Institute of Technology

MANAGING THE CPU

2

Processes in Operating Systems

- Hundreds of things going on in the system



- How to make things simple?
  - Separate each into an isolated process
  - Decomposition of large systems into smaller parts

3

What is a process?

- Process represents a running program
  - Program = recipe
  - Process = cake
- A process has two parts
  - Address space
  - Thread(s) of control

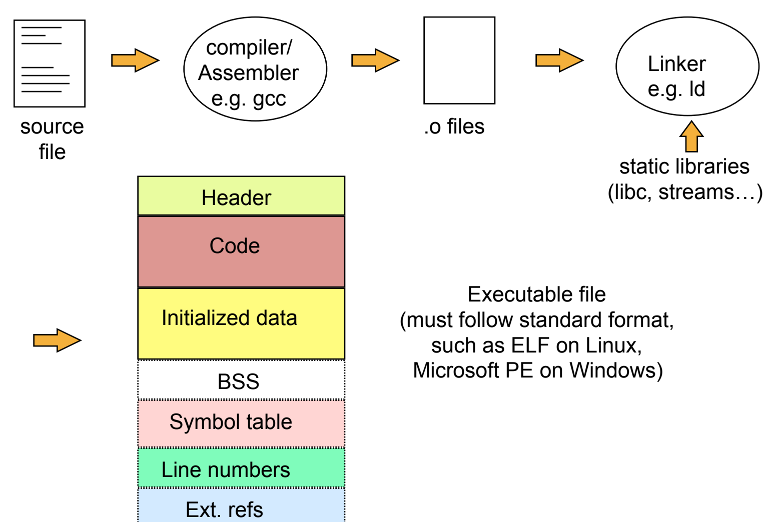
4

What is a program?

- Code: machine instructions
- Data: variables stored in memory
  - Initialized data: global variables
  - BSS (block started by symbol): global variables initialized with zeros when execution begins
  - At run-time, data will be allocated dynamically
    - On the run-time stack, for variables
    - On the heap, for storage allocated with new, etc

5

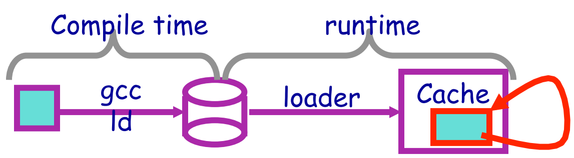
Preparing a Program on Unix



6

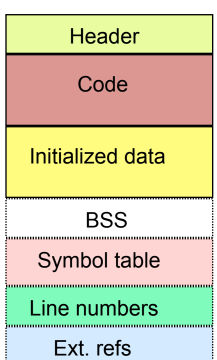
Running a program

- OS creates a "process" and allocates memory for it
- The loader:
  – Reads executable file
  – sets process' memory from exec
  – pushes arguments onto run-time stack
  – sets CPU registers & calls a start procedure
  – start procedure calls into main program



7

Process != Program

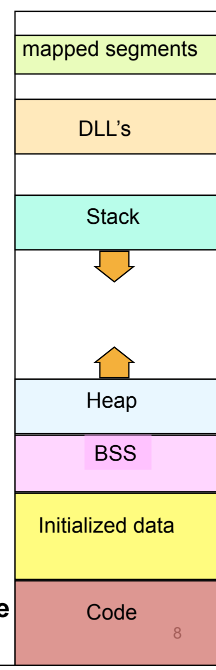


Executable program

Program is passive
• Code + data

Process is running program
• stack, regs, program counter

Example:
We both run Firefox:
- Same program
- Separate processes



Process address space

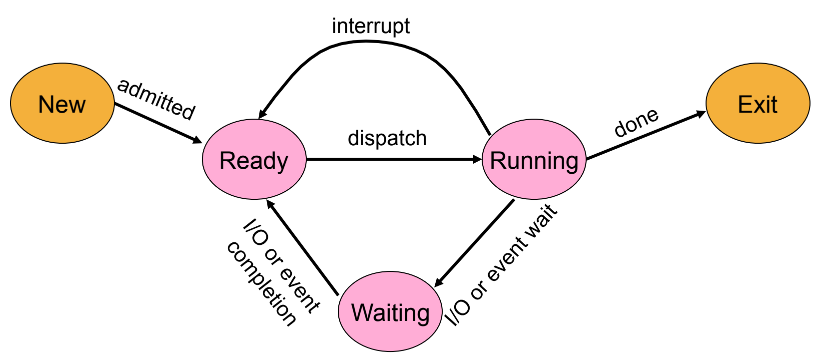
8

Process States

- Many processes, only one on the CPU
  - One thread on each core
- "Execution State" of a process:
  - Ready: waiting to be assigned to the CPU
  - Running: executing instructions on the CPU
  - Waiting: waiting for an event, e.g. I/O completion
- Process moves across different states

9

Process State Transitions



Processes hop across states as a result of:
- Actions they perform, e.g. system calls
- Actions performed by OS, e.g. rescheduling
- External actions, e.g. I/O

10

PROTECTION IN OPERATING SYSTEMS

Protection in Operating Systems

- Operating system protects:
  - Processes from other processes
    - Contain damage from errors
    - Isolate viruses
  - The machine from processes
    - Protect disk against unauthorized access
    - Protect network against password sniffing
    - Protect CPU from denial-of-service attack (loops)

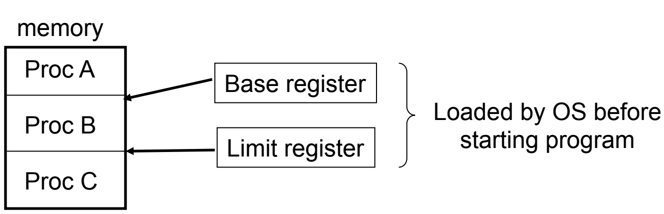
Prevent runaway processes

- Control use of CPU
  - Hardware **timer** device
  - OS sets the timer, runs process
  - Tmer interrupt gives control back to the OS
    - Execute the interrupt handler for timer interrupts

- Setting timer is a *privileged operation*

Protect a process' memory

- Base and limit registers:



- Setting these registers is a *privileged operation*

Privileged Instructions

- Also called protected instructions:
  - Direct user access to I/O devices like disks, etc.
  - Instructions that manipulate page table pointers, TLB load, etc.
  - Setting of special mode bits
  - Halt instruction
- How do we make sure only the kernel can execute privileged instructions?

15

Dual-Mode Operation

- OS runs in **kernel mode**, user programs in **user mode**
  - OS is god, the applications are peasants
  - Privileged instructions only executable in kernel mode
- Mode bit provided by hardware
  - System call changes mode to kernel
  - Return from system call resets it to user
- How do user programs do something privileged?

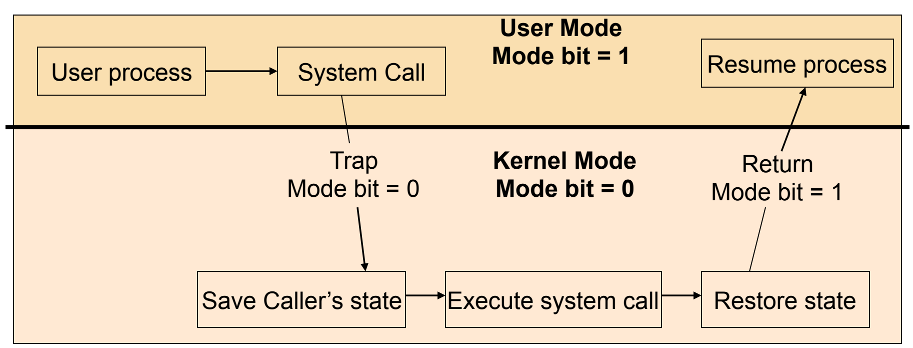
16

System Calls: Crossing Protection Boundaries

- User process uses **system calls** to call OS
- Requires *unprivileged* instruction to switch mode from user to kernel
- Safe switching to kernel mode: the **trap instruction**
  - Switch from user to kernel mode
  - Start executing at a specific location in memory,
  - OS will have installed a *trap handler*

17

System Calls: Crossing Protection Boundaries



18

System Call Overhead

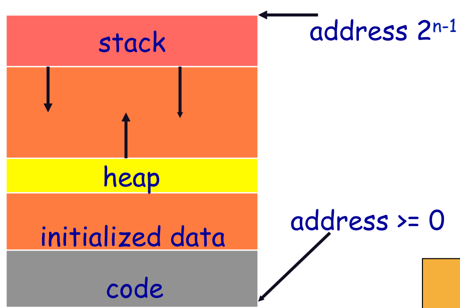
- Problem: The user-kernel mode distinction poses a performance barrier
  - System calls take 10x-1000x more time than a proc call
- Solution: Perform some system functionality in user mode
  - Libraries (DLLs)
  - Caching results (getpid)
  - Buffering operations (open/read/write Unix system calls vs. fopen/fread/fwrite C library calls).

19

MEMORY MANAGEMENT

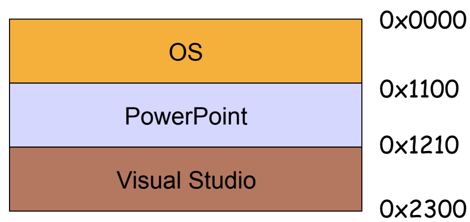
20

Memory Protection



Each process thinks it "owns" all of memory, from address 0 to 2^{n-1}...

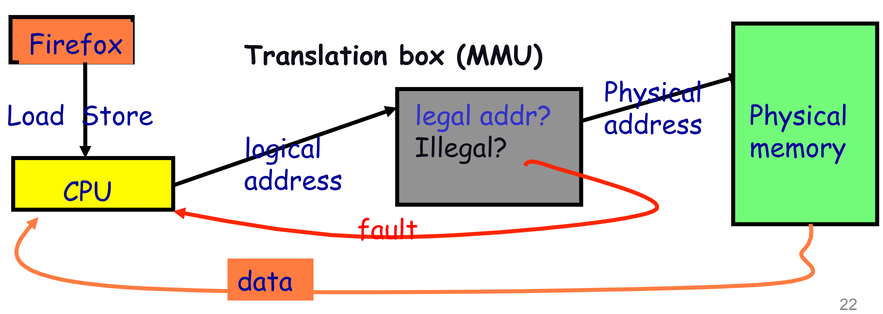
...but the reality is that physical memory is occupied by several processes.



21

Memory Management Unit (MMU)

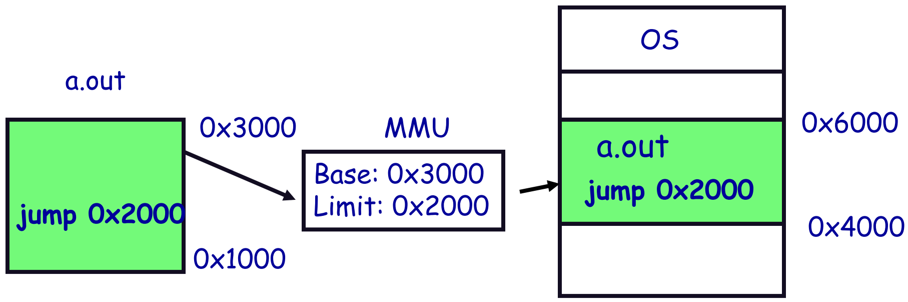
- Protection: Errors in process should not affect others
- *Logical address* assumes process starts at location 0
- MMU is hardware that translates to *physical address*



22

Base and Limit Registers

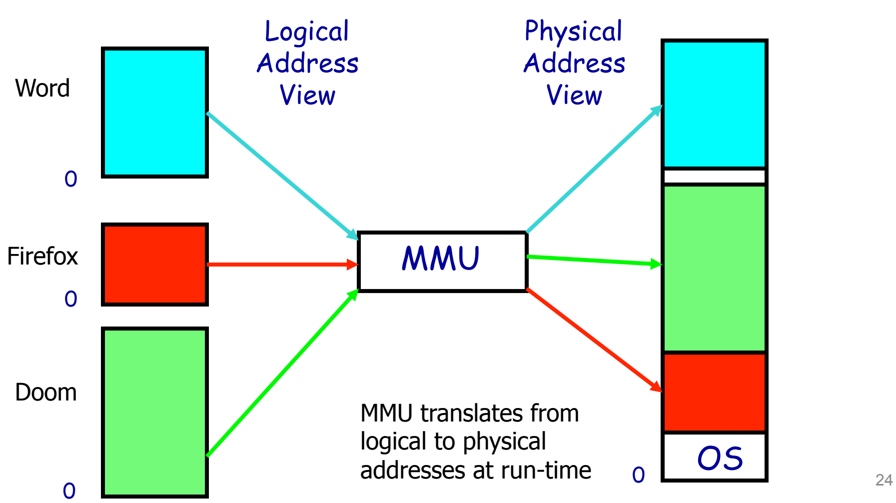
- Dynamic address translation for every memory access
- Relocation: physical address = logical address + base
- Protection: is virtual address < limit?



  - When process runs, base register = 0x3000, bounds register = 0x2000. Jump addr = 0x2000 + 0x3000 = 0x5000

23

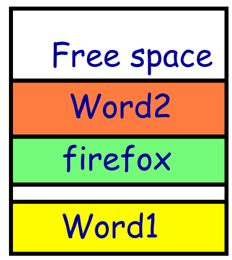
Logical and Physical Addresses



24

Problems with Base and Limit Regs

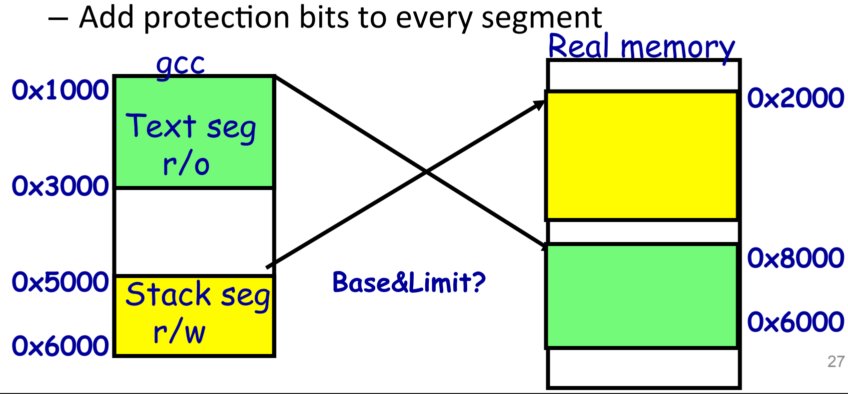
- Problem 1: growing processes
  - Multiple Word docs
- Problem 2: how to share code and data?
  - Multiple Word processes
- Problem 3: how to separate code and data?
  - Prevent viruses
  - Support shared libraries
- Idea: Split process memory into *segments*



MEMORY MANAGEMENT: SEGMENTATION

Segmentation

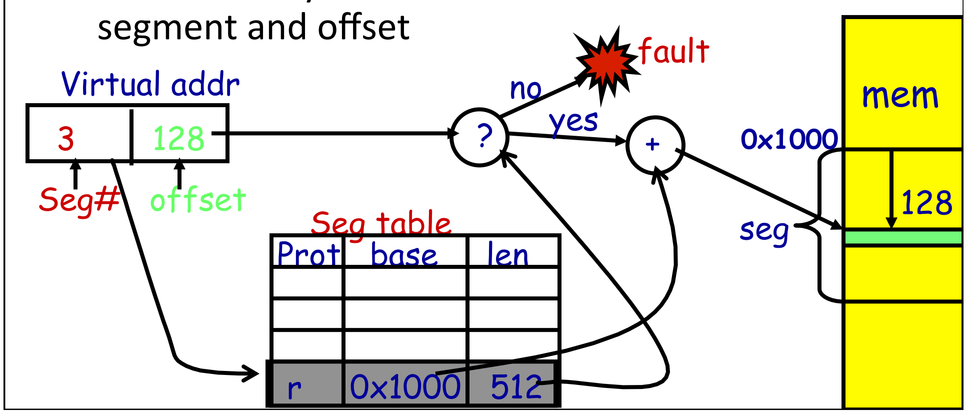
- Processes have *multiple base + limit registers*
- Processes address space has multiple segments
  - Each segment has its own base + limit registers
  - Add protection bits to every segment



27

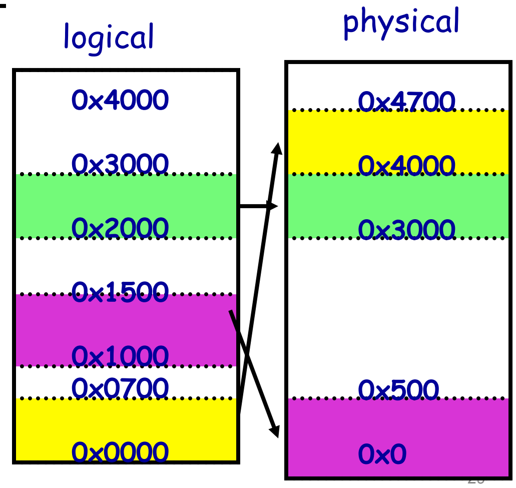
Mapping Segments

- Segment Table: entry for each segment
  - Each entry is a tuple <protection bits, base, limit>
- Each memory reference indicates segment and offset



Segmentation Example

Seg	base	bounds	rw
0	0x4000	0x6ff	10
1	0x0000	0x4ff	11
2	0x3000	0xfff	11
3			00



- where is 0x0240?
- 0x1108?
- 0x265c?
- 0x3002?
- 0x1700?

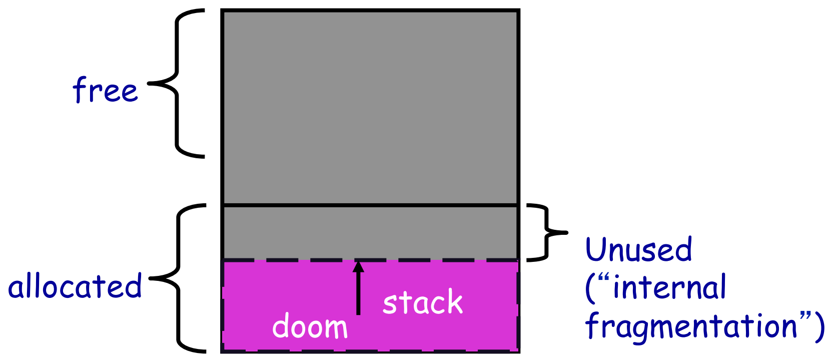
- First two bits for segments (4)
- Next 12 for offset (4KB)

Segment Problem: Fragmentation

- “The inability to use free memory”
- Internal Fragmentation:
  – *Fixed sized* pieces ⇒ internal waste if entire piece is not used
- External Fragmentation
  – *Variable sized* pieces ⇒ many small holes over time

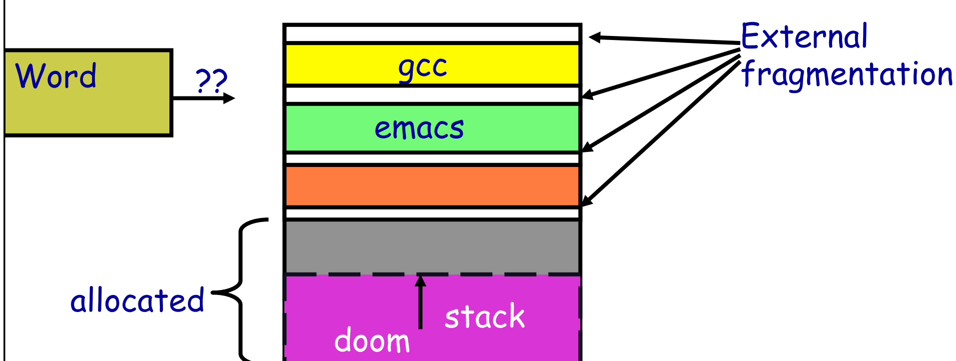
30

Internal Fragmentation



31

External Fragmentation

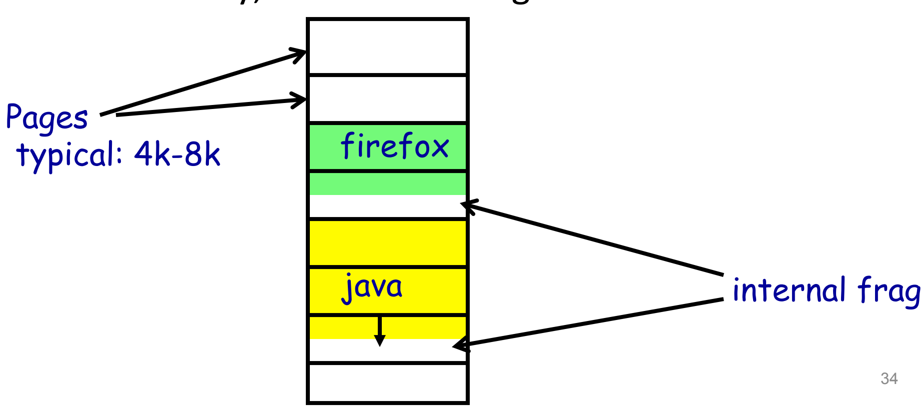


32

MEMORY MANAGEMENT: PAGING

Paging

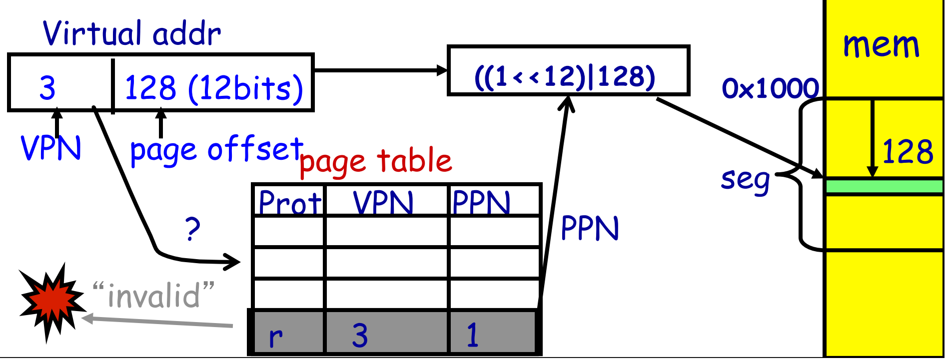
- Divide memory into fixed size pieces
  - Called “frames” or “pages”
- Pros: easy, no external fragmentation



34

Mapping Pages

- If 2^m virtual address space, 2^n page size
  - (m - n) bits to denote page number
  - n bits for offset within page
- Translation done using a *Page Table*

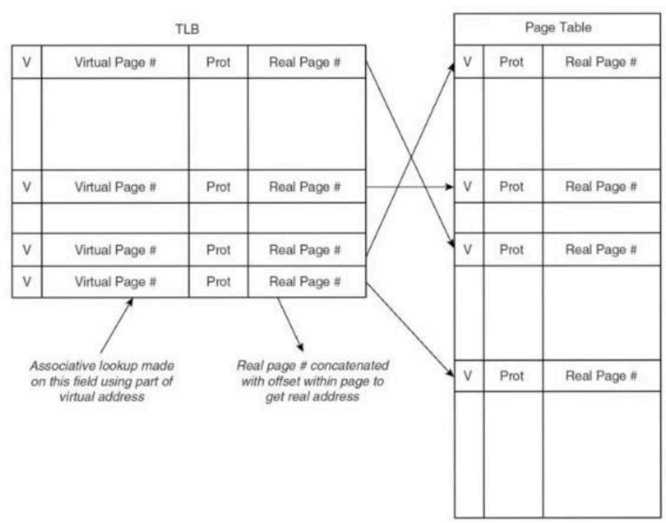


Paging: Hardware Support

- Entire page table (PT) in registers
  - PT can be huge ~ 1 million entries
- Store PT in main memory
  - Have **Page Table Base Register (PTBR)** point to start of PT
  - Con: 2 memory accesses to get to any physical address
- Use **Translation Lookaside Buffers (TLB)**:
  - High speed associative memory
  - Basically a cache for PT entries

36

Translation Lookaside Buffer (TLB)

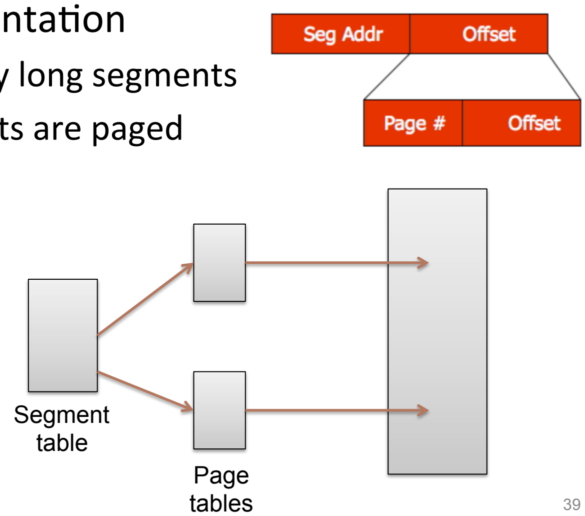


37

MEMORY MANAGEMENT: PAGING + SEGMENTATION

Paging + Segmentation

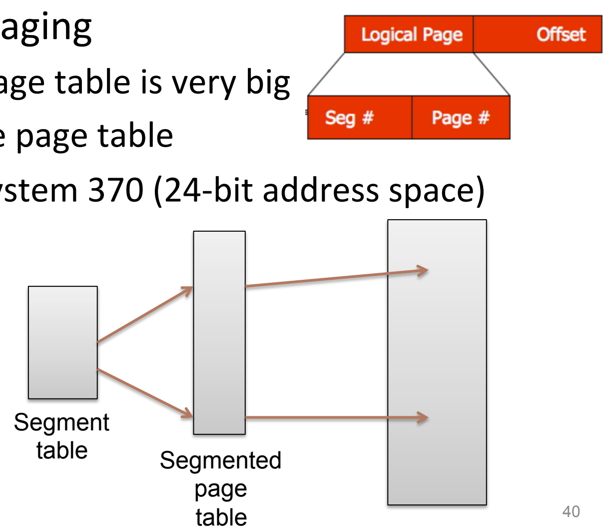
- Paged segmentation
  - Handles very long segments
  - The segments are paged



39

Paging + Segmentation

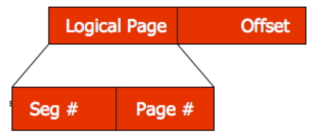
- Segmented Paging
  - When the page table is very big
  - Segment the page table
  - Example: System 370 (24-bit address space)



40

Paging + Segmentation

- Segmented Paging
  - When the page table is very big
  - Segment the page table
  - Example: System 370 (24-bit address space)



Seg # (4 bits)	page # (8 bits)	page offset (12 bits)

41

Segmented Paging

Table entry addresses

	Base	bound	prot
0	0x2000	0x14	R
1	0x0000	0x00	
2	0x1000	0x0d	RW

Segment table

Page table (2-byte entries)

Page table	Table entry addresses
0x001f	
0x0011	0x2020
…	
0x0003	
0x002a	
0x0013	0x2000
…	
0x000c	
0x0007	0x1020
…	
0x004	
0x00b	
0x006	0x1000

0x202016 read?
0x104c84 read?
0x011424 read?
0x210014 write?

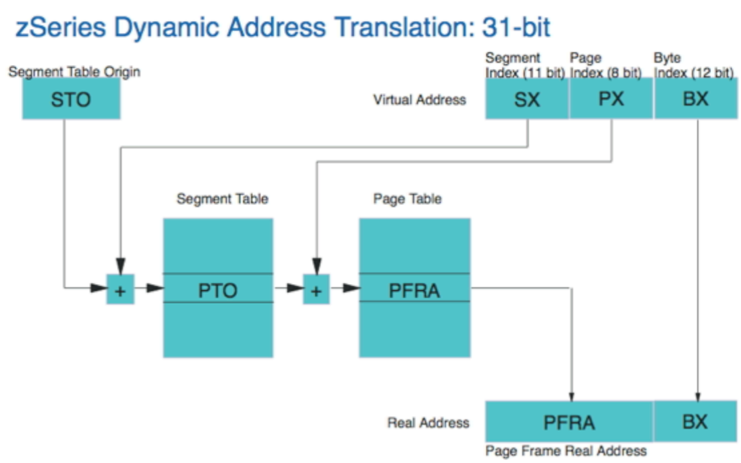
Seg 2: page 2, PTE @ 0x1004, addr = 0x4016
Seg 1: page 4, PTE @ protection error
Seg 0, page 0x11, PTE@0x2022, addr = 0x1f424
Seg 2, page 0x10, PTE@ bounds violation

42

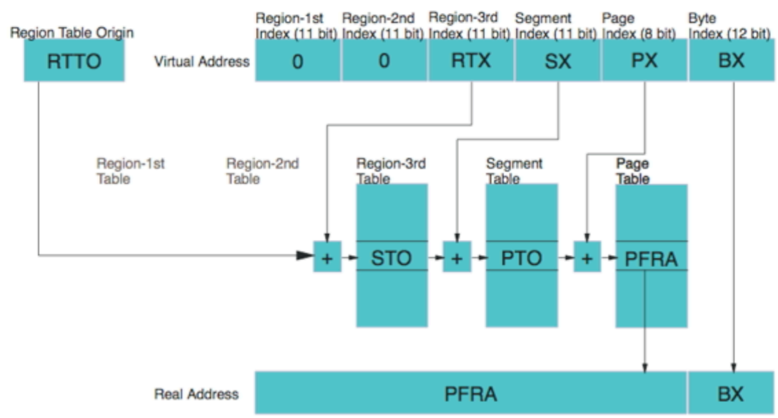
Handling PT size

- Segmented Paging: no need for contiguous alloc
- Other approaches:
  - Hierarchical Paging: Page the page table
  - Hashed Page Table: Each entry maps to linked list of pages
  - Inverted Page Table:
    - Map from Frame to (VA, process) instead of VA to frame per process

43



zSeries DAT: 64-bit Three-Level Translation



MEMORY MANAGEMENT: VIRTUAL MEMORY

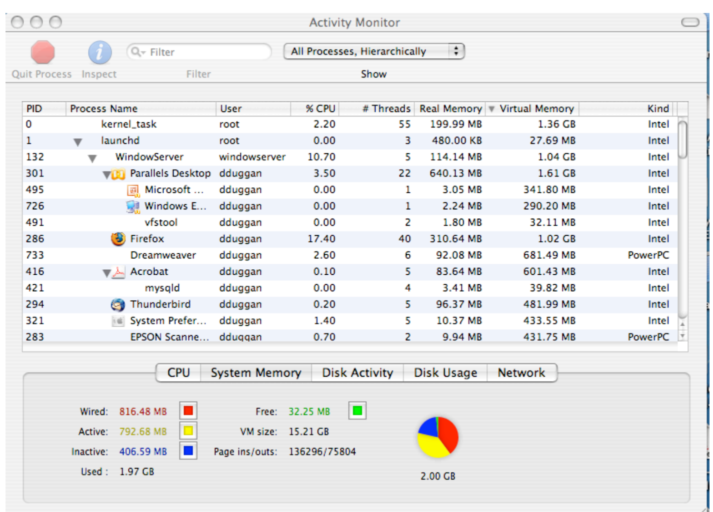
46

Virtual Memory

- Each process has illusion of large address space
  - 2^{64} for 64-bit addressing
  - i.e. 16 x 1018 = 16 million terabytes
- Reality:
  - Memory shared by other processes
  - Heavy memory requirements
  - Physical memory relatively small

47

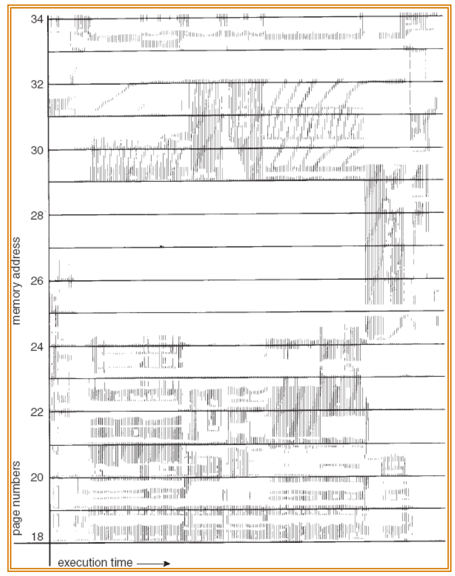
Example Memory Requirements



48

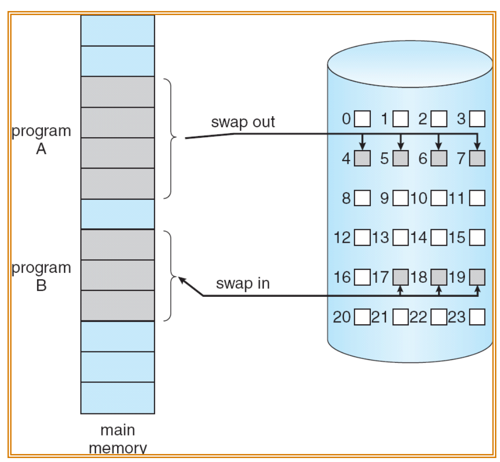
Locality of Reference

- Key insight: programs go through phases with different requirements
- Example: Compiler has separate parsing, analysis, optimization, code generation phases.
- Just keep used part of a process in memory
- Store the unused part on disk (in the *swap partition* or *swap file*).



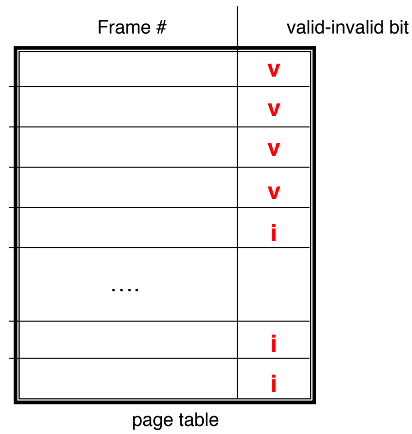
Paging In and Out Process State

- **Page in:** load part of process' logical memory from disk into physical memory.
- **Page out:** write part of process' logical memory to disk from physical memory
  – free memory
- **Swap in/out:** move entire process state from/to disk



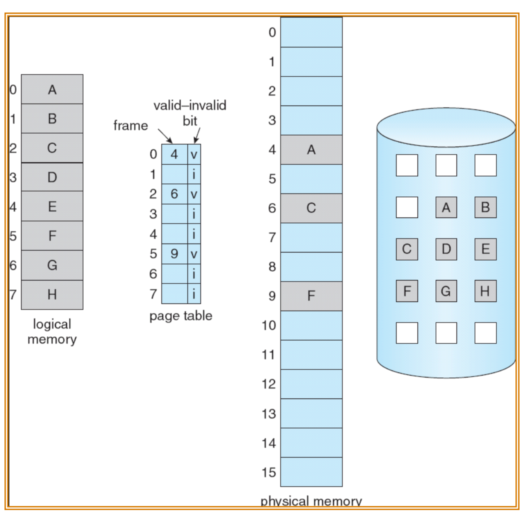
Demand Paging

- Bring a page into memory only when it is needed
- Page table: information about whether a page is in memory (valid) or stored in swap space (invalid)
- Reference to address in page with invalid bit causes page fault
  - Trap to kernel



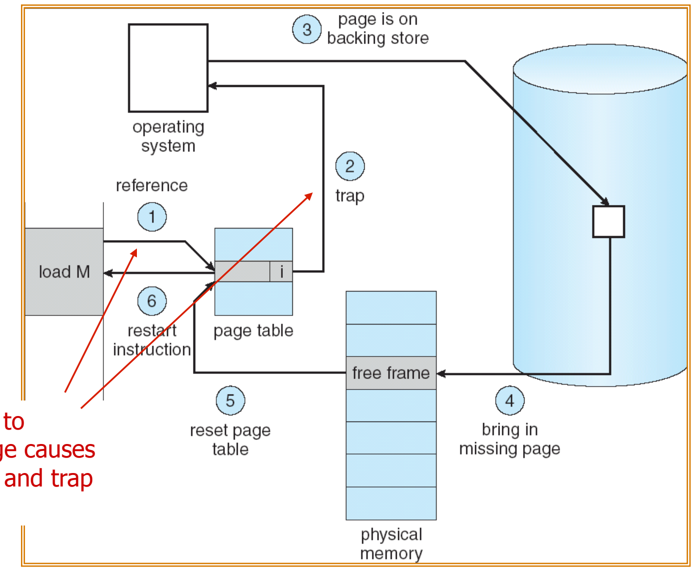
51

Page Table When Some Pages Are Not in Main Memory



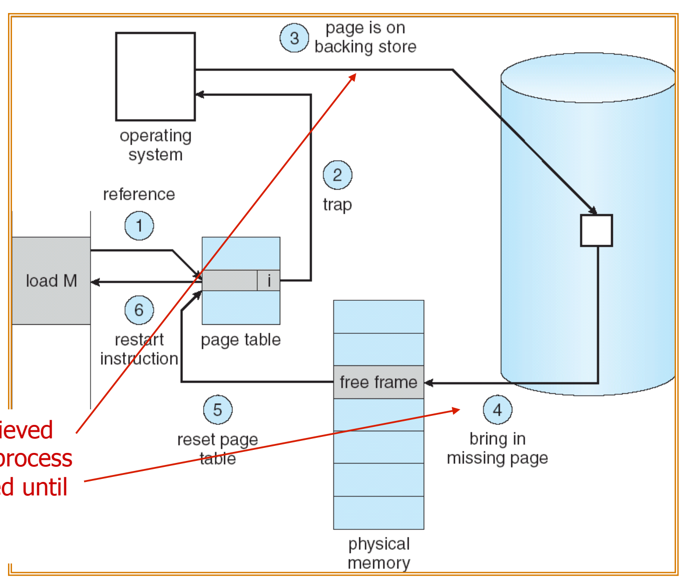
52

Steps in Handling a Page Fault



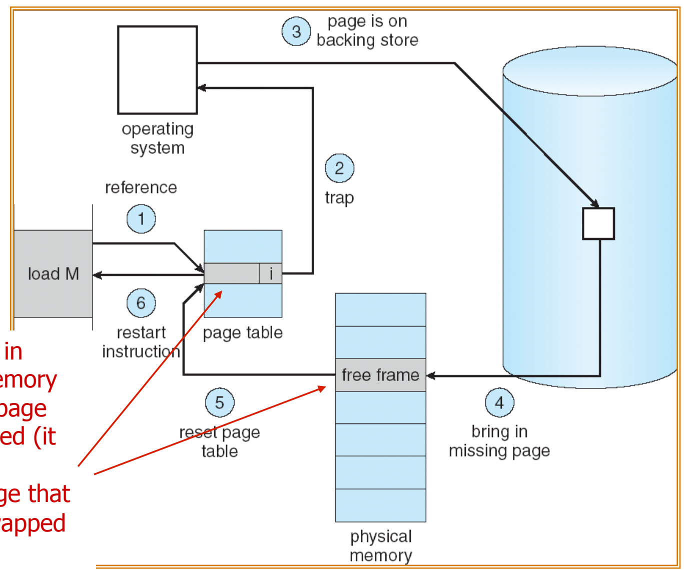
Reference to invalid page causes page fault and trap to kernel

Steps in Handling a Page Fault



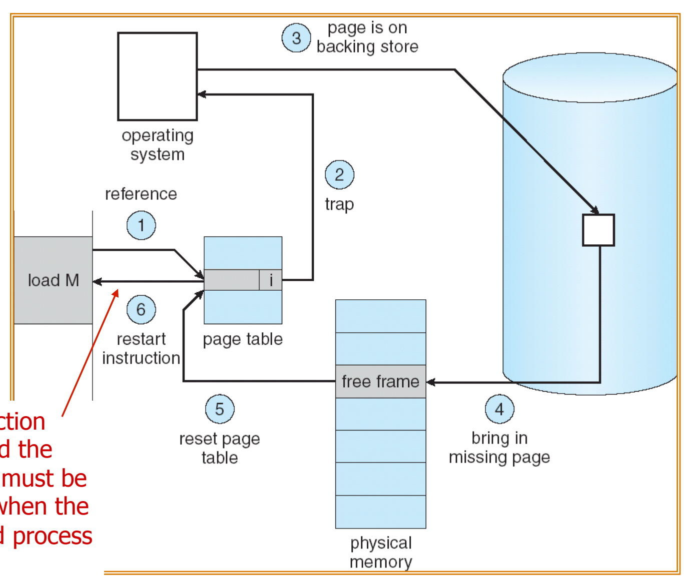
Page is retrieved from disk (process is suspended until disk I/O completes)

Steps in Handling a Page Fault



Page is put in physical memory frame and page table updated (it will replace another page that must be swapped out)

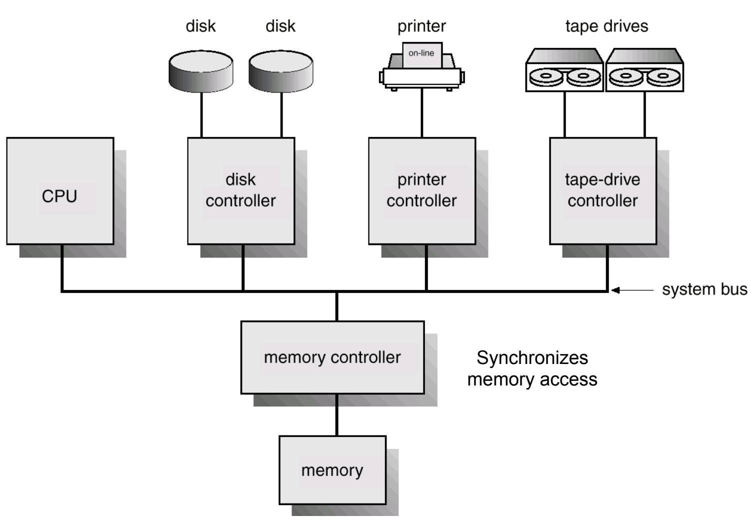
Steps in Handling a Page Fault



The instruction that caused the page fault must be restarted when the interrupted process resumes

INPUT OUTPUT

Computer System Architecture



58

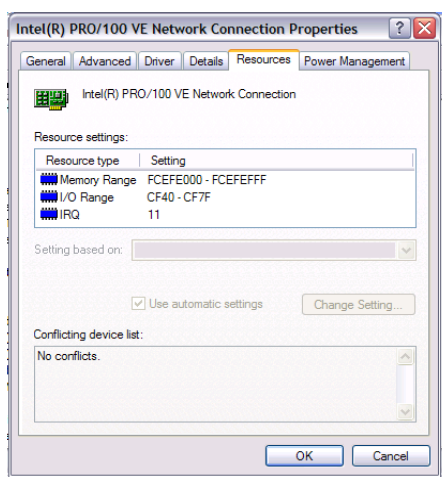
I/O operations

- I/O devices and the CPU can execute concurrently.
  - I/O is moving data between device & controller's buffer
  - CPU moves data between controller's buffer & main memory
- Each device controller is in charge of a device type.
  - May be more than one device per controller
    - SCSI can manage up to 7 devices
  - Each device controller has local buffer, special registers
- A device driver for every device controller
  - Knows details of the controller
  - Presents a uniform interface to the rest of OS

59

Accessing I/O Devices

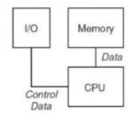
- Memory Mapped I/O
  - I/O devices appear as regular memory to CPU
  - Regular loads/stores used for accessing device
- Programmed I/O
  - Also called I/O mapped I/O
  - CPU has separate bus for I/O devices
  - Special instructions are required



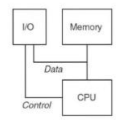
60

I/O Organization

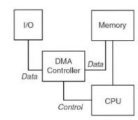
- Programmed I/O



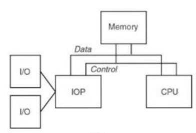
- Interrupt-driven I/O



- DMA-managed I/O



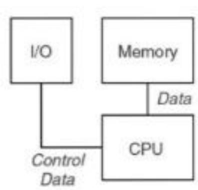
- IOP-driven I/O



61

Programmed I/O

- OS issues I/O request over I/O bus
- Polls device controller until the request is satisfied



62

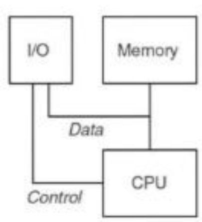
Interrupts

- Notification that device needs servicing
  - Hardware: sends trigger on bus
  - Software: uses a system call
- On receiving an interrupt:
  - Stop kernel execution
  - Save machine context at interrupted instruction
  - Commonly, incoming interrupts are disabled
  - Transfer execution to Interrupt Service Routine (ISR)
  - After ISR, restore kernel state and resume execution

63

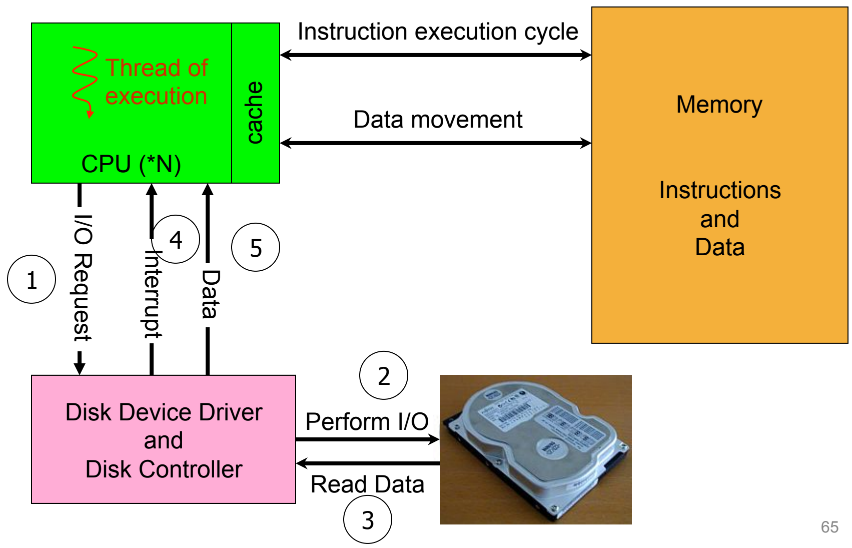
Interrupt-driven I/O

- OS issues I/O request, then performs other work
- Device controller notifies OS via interrupt when the request is satisfied

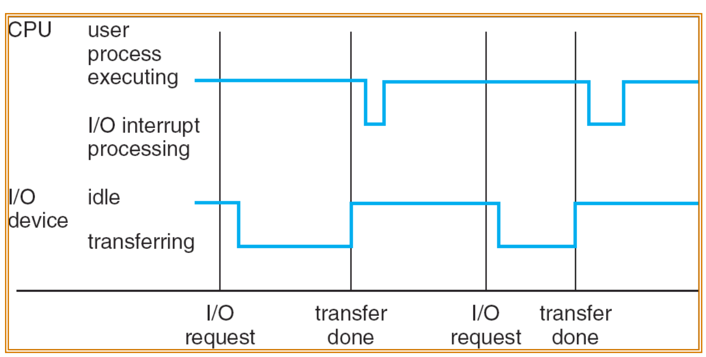


64

Interrupt-driven I/O



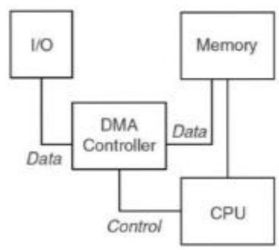
Interrupt Timeline



The cost is the interruption of the CPU to service interrupts. On some architectures e.g. PCI, devices may send interrupts directly to each other without involving the CPU.

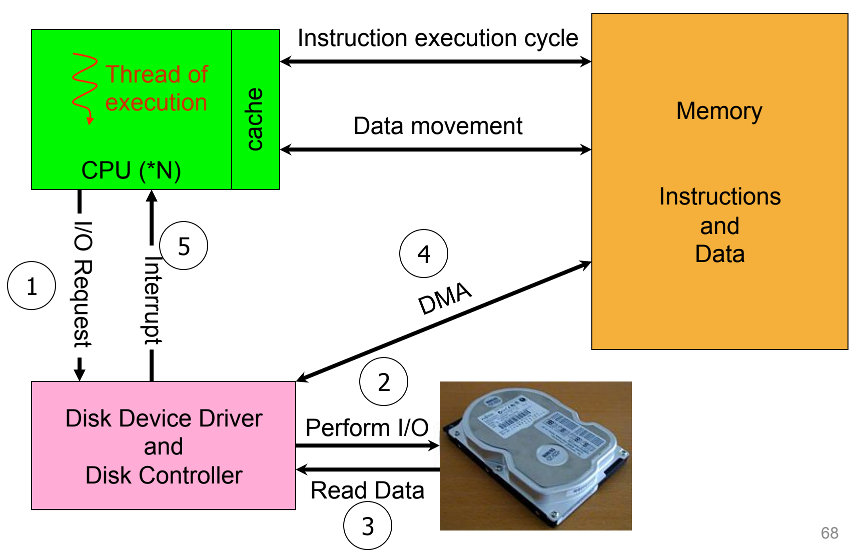
DMA-managed I/O

- CPU initiates block data transfer
- I/O Controller can access memory directly
  - Data transfer independent of CPU
- Interrupts to notify CPU when I/O completed



67

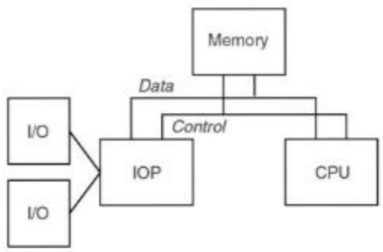
DMA-managed I/O



68

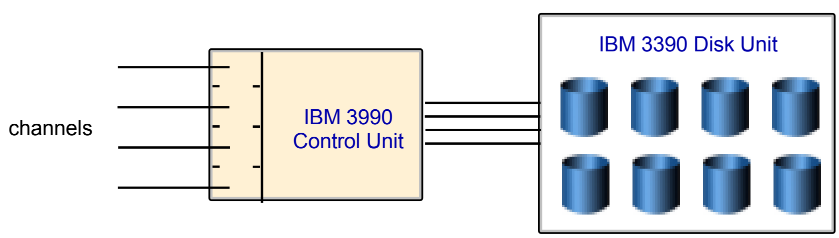
IOP-based I/O

- I/O processors (IOPs): special processors that can execute special I/O programs
- OS and IOP communicate through memory
  - OS sets up program in main memory
  - IOP executes the program



69

Example: Channel I/O



- Channel I/O can be viewed as a generalization of DMA
  - DMA transfers block of bytes from device to main memory
  - Channel program may e.g. format a disk track
  - Initiated by a single I/O instruction on a central processor

70