Operating System Concepts

Syllabus

- 上課時間: Friday 19:35-22:00
- 教室:M 501
- 教科書：

  Silberschatz, Galvin, and Gagne, “Operating System Concept,” Seventh Edition, John Wiley & Sons, Inc., 2006.
- 成績評量：(subject to changes.):

  期中考(30%), 期末考(30%), 課堂參與(40%)

Contents

Chapter 1. Introduction

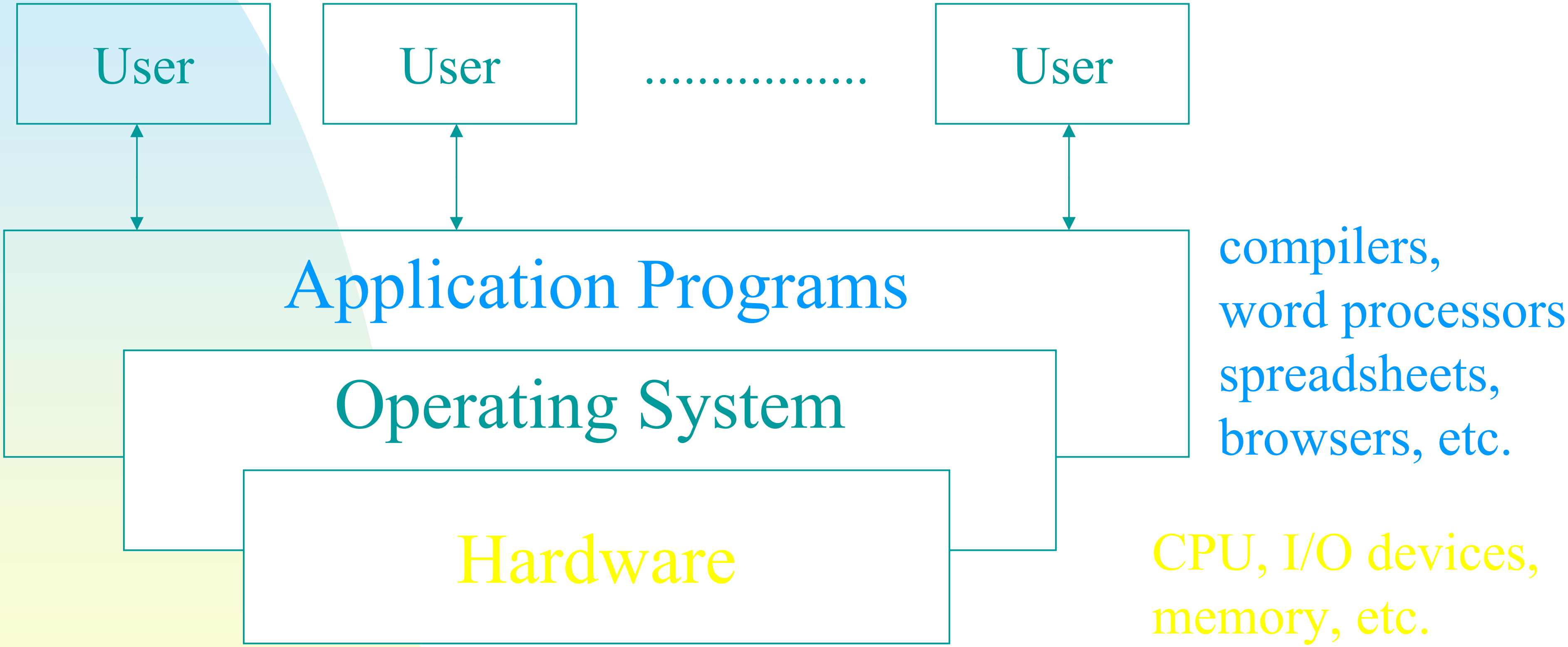
Introduction

- What is an Operating System?
  - A basis for application programs
  - An intermediary between users and hardware

- Amazing variety
  - Mainframe, personal computer (PC), handheld computer, embedded computer without any user view

Convenient vs Efficient

Computer System Components



- OS – a government/environment provider

User View

- The user view of the computer varies by the interface being used!
- Examples:
  - Personal Computer → Ease of use
  - Mainframe or minicomputer → maximization of resource utilization
    - Efficiency and fair share
  - Workstations → compromise between individual usability & resource utilization
  - Handheld computer → individual usability
  - Embedded computer without user view → run without user intervention

System View

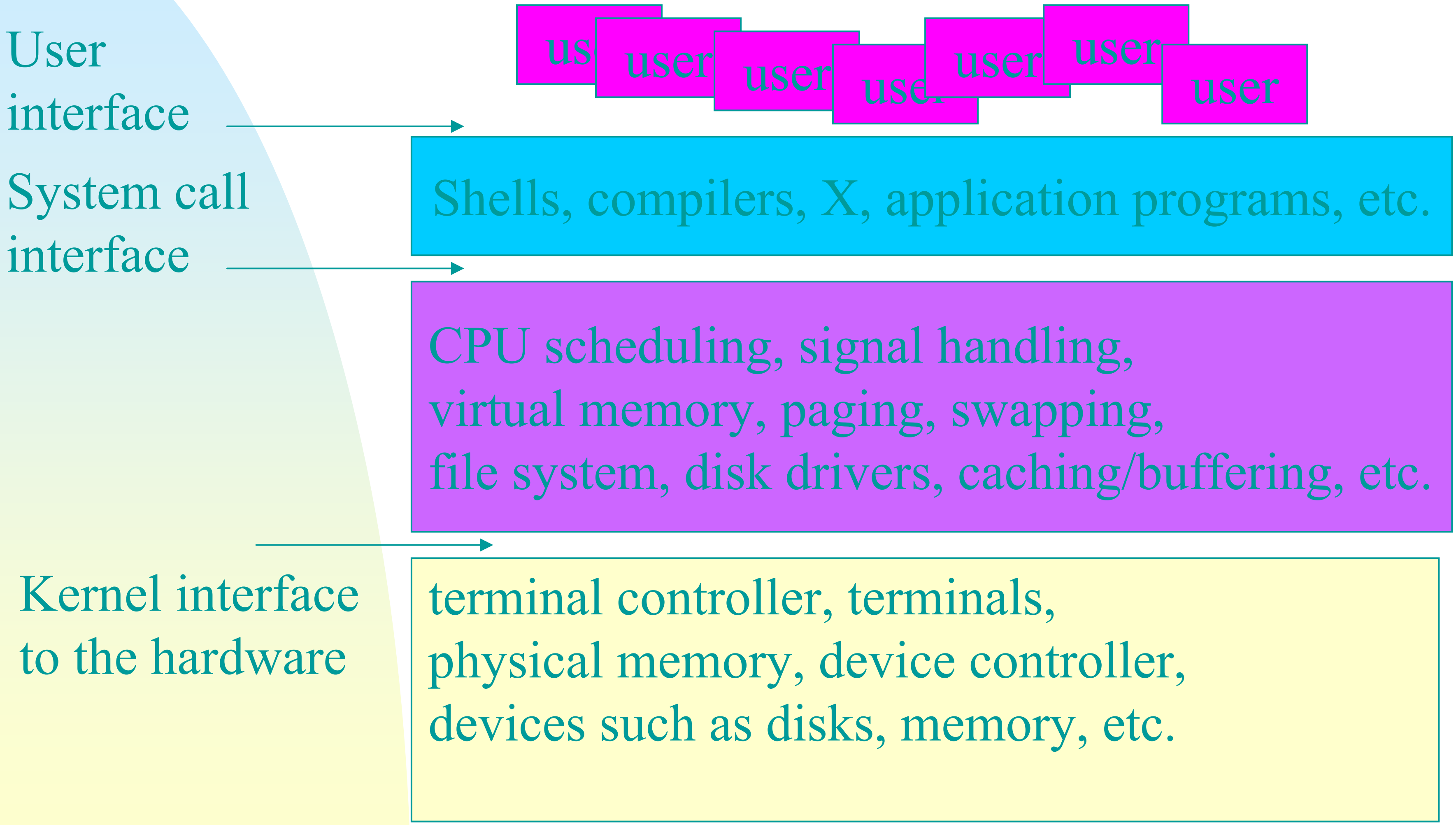
- A Resource Allocator
  - CPU time, Memory Space, File Storage, I/O Devices, Shared Code, Data Structures, and more
- A Control Program
  - Control execution of user programs
  - Prevent errors and misuse
- OS definitions – US Dept.of Justice against Microsoft in 1998
  - The stuff shipped by vendors as an OS
  - Run at all time

System Goals

- Two Conflicting Goals:
  - Convenient for the user!
  - Efficient operation of the computer system!

- We should
  - recognize the influences of operating systems and computer architecture on each other
  - and learn why and how OS's are by tracing their evolution and predicting what they will become!

UNIX Architecture



Mainframe Systems

- The first used to tackle many commercial and scientific applications!
  - 0th Generation – 1940?s
  - A significant amount of set-up time in the running of a job
  - Programmer = operator
  - Programmed in binary → assembler → (1950) high level languages

Mainframe – Batch Systems

- Batches sorted and submitted by the operator
- Simple batch systems
  - Off-line processing

    ~ Replace slow input devices with faster units → replace card readers with disks
  - Resident monitor

    ~ Automatically transfer control from one job to the next

• loader • job sequencing • control card interpreter	monitor
User Program Area	

Mainframe – Batch Systems

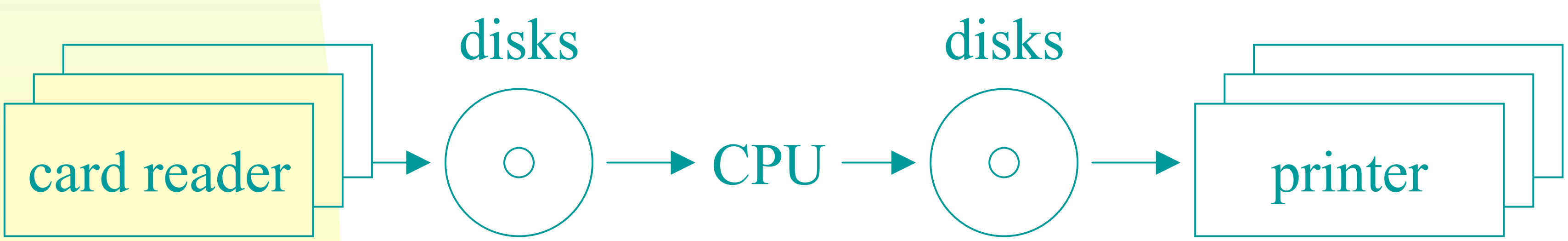
- Spooling (Simultaneous Peripheral Operation On-Line)

~ Replace sequential-access devices with random-access device

=> Overlap the I/O of one job with the computation of others

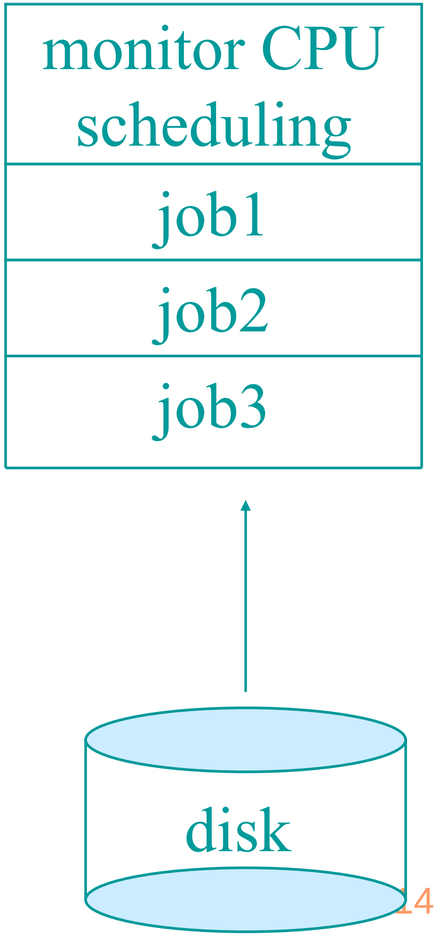
e.g. card → disk, CPU services, disk → printer

- Job Scheduling



Mainframe – Multiprogrammed Systems

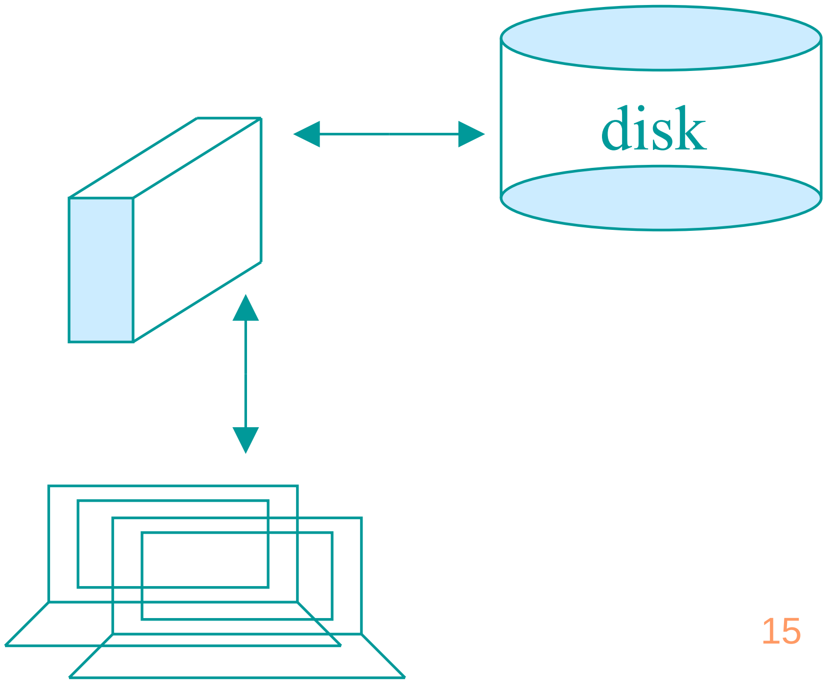
- Multiprogramming increases CPU utilization by organizing jobs so that the CPU always has one to execute – Early 1960
  - Multiporgrammed batched systems
  - Job scheduling and CPU scheduling
  - Goal : efficient use of scare resources



Mainframe – Time-Sharing Systems

- Time sharing (or multitasking) is a logical extension of multiprogramming!
  - Started in 1960s and become common in 1970s.
  - An interactive (or hand-on) computer system
  - Multics, IBM OS/360

on-line file system
virtual memory
sophisticated CPU scheduling
job synchronization
protection & security
......
and so on



Desktop Systems

- Personal Computers (PC's)
  - Appeared in the 1970s.
  - Goals of operating systems keep changing
    - Less-Powerful Hardware & Isolated Environment→ Poor Features
      - Benefited from the development of mainframe OS's and the dropping of hardware cost
      - Advanced protection features
    - User Convenience & Responsiveness

Parallel Systems

- Tightly coupled: have more than one processor in close communication sharing computer bus, clock, and sometimes memory and peripheral devices
- Loosely coupled: otherwise
- Advantages
  - Speedup – Throughput
  - Lower cost – Economy of Scale
  - More reliable – Graceful Degradation → Fail Soft (detection, diagnosis, correction)
    - A Tandem fault-tolerance solution

Parallel Systems

- Symmetric multiprocessing model: each processor runs an identical copy of the OS
- Asymmetric multiprocessing model: a master-slave relationship
  - ~ Dynamically allocate or pre-allocate tasks
  - ~ Commonly seen in extremely large systems
  - ~ Hardware and software make a difference?
- Trend: the dropping of microporcessor cost

  ➔ OS functions are offloaded to slave processors (back-ends)

Distributed Systems

- Definition: Loosely-Coupled Systems – processors do not share memory or a clock
  - Heterogeneous vs Homogeneous
- Advantages or Reasons
  - Resource sharing: computation power, peripheral devices, specialized hardware
  - Computation speedup: distribute the computation among various sites – load sharing
  - Reliability: redundancy → reliability
  - Communication: X-window, email

Distributed Systems

- Distributed systems depend on networking for their functionality.
  - Networks vary by the protocols used.
    - TCP/IP, ATM, etc.
  - Types – distance
    - Local-area network (LAN)
    - Wide-area network (WAN)
    - Metropolitan-area network (MAN)
    - Small-area network – distance of few feet
  - Media – copper wires, fiber strands, satellite wireless transmission, infrared communication,etc.

Distributed Systems

- Client-Server Systems
  - Compute-server systems
  - File-server systems
- Peer-to-Peer Systems
  - Network connectivity is an essential component.
- Network Operating Systems
  - Autonomous computers
  - A distributed operating system – a single OS controlling the network.

Clustered Systems

- Definition: Clustered computers which share storage and are closely linked via LAN networking.
- Advantages: high availability, performance improvement, etc.
- Types
  - Asymmetric/symmetric clustering
  - Parallel clustering – multiple hosts that access the same data on the shared storage.
  - Global clusters
- Distributed Lock Manager (DLM)

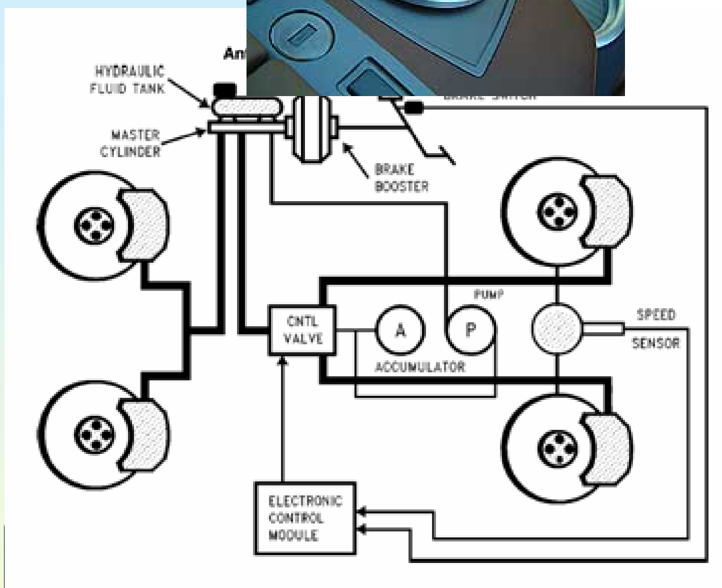
Real-Time Systems

- Definition: A real-time system is a computer system where a timely response by the computer to external stimuli is vital!
- Hard real-time system: The system has failed if a timing constraint, e.g. deadline, is not met.
  - All delays in the system must be bounded.
  - Many advanced features are absent.

Real-Time Systems

- Soft real-time system: Missing a timing constraint is serious, but does not necessarily result in a failure unless it is excessive
  - A critical task has a higher priority.
  - Supported in most commercial OS.
- Real-time means on-time instead of fast

Applications for Real-Time Systems!



Real-Time Systems

- Applications
  - Air traffic control
  - Space shuttle
  - Navigation
  - Multimedia systems
  - Industrial control systems
  - Home appliance controller
  - Nuclear power plant
  - Virtual reality
  - Games
  - User interface
  - Vision and speech recognition (approx. 100 ~ 200ms)
  - PDA, telephone system
  - And more

Handheld Systems

- Handheld Systems
  - E.g., Personal Digital Assistant (PDA)
- New Challenges – convenience vs portability
  - Limited Size and Weight
  - Small Memory Size
    - No Virtual Memory
  - Slow Processor
    - Battery Power
  - Small display screen
    - Web-clipping

Feature Migration

- MULTIplexed Information and Computing Services (MULTICS)
  - 1965-1970 at MIT as a utility
- UNIX
  - Since 1970 on PDP-11
- Recent OS's
  - MS Windows, IBM OS/2, MacOS X
- OS features being scaled down to fit PC's
  - Personal Workstations – large PC's

Computing Environments

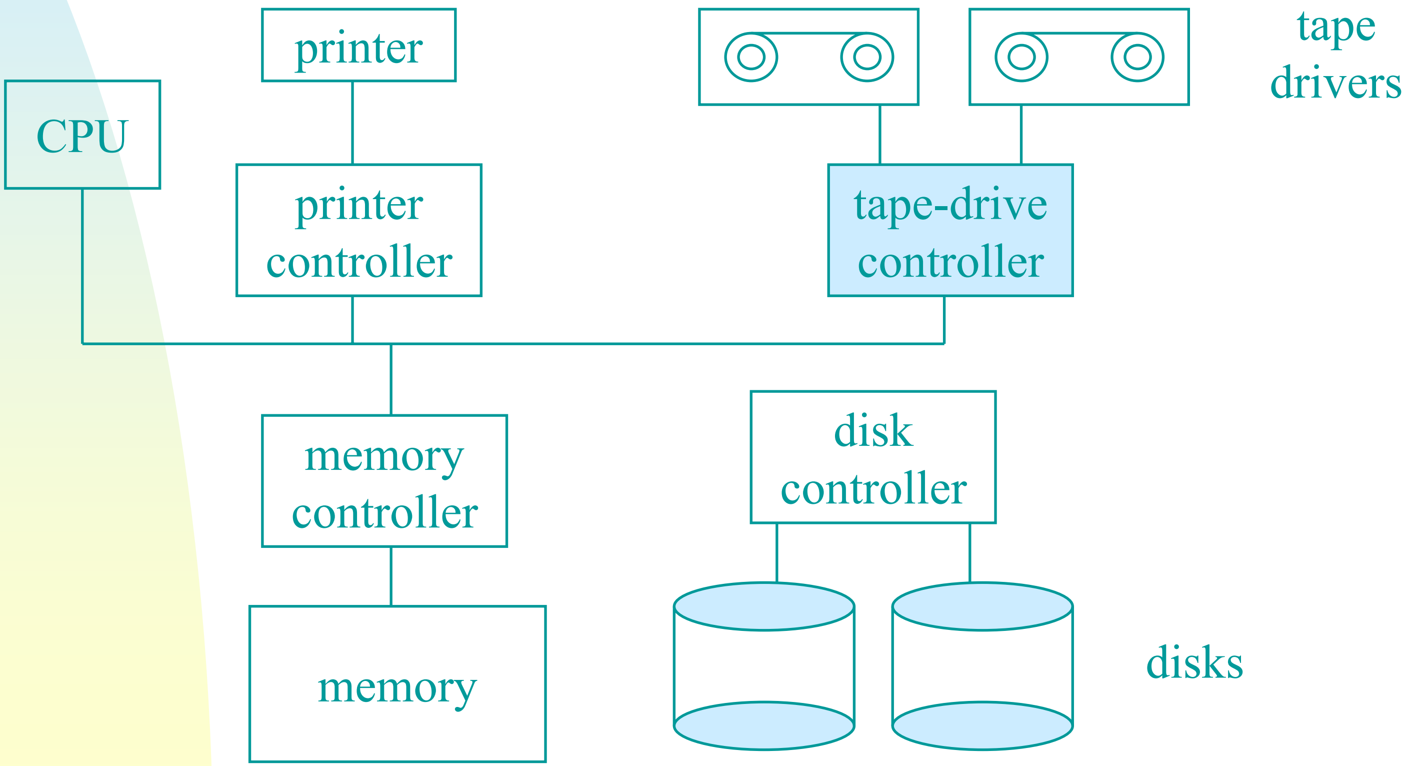
- Traditional Computing
  - E.g., typical office environment
- Web-Based Computing
  - Web Technology
    - Portals, network computers, etc.
  - Network connectivity
  - New categories of devices
    - Load balancers
- Embedded Computing
  - Car engines, robots, VCR's, home automation
  - Embedded OS's often have limited features.

Contents

Chapter 2
Computer-System Structure

Computer-System Structure

- Objective: General knowledge of the structure of a computer system.



- Device controllers: synchronize and manage access to devices.

Booting

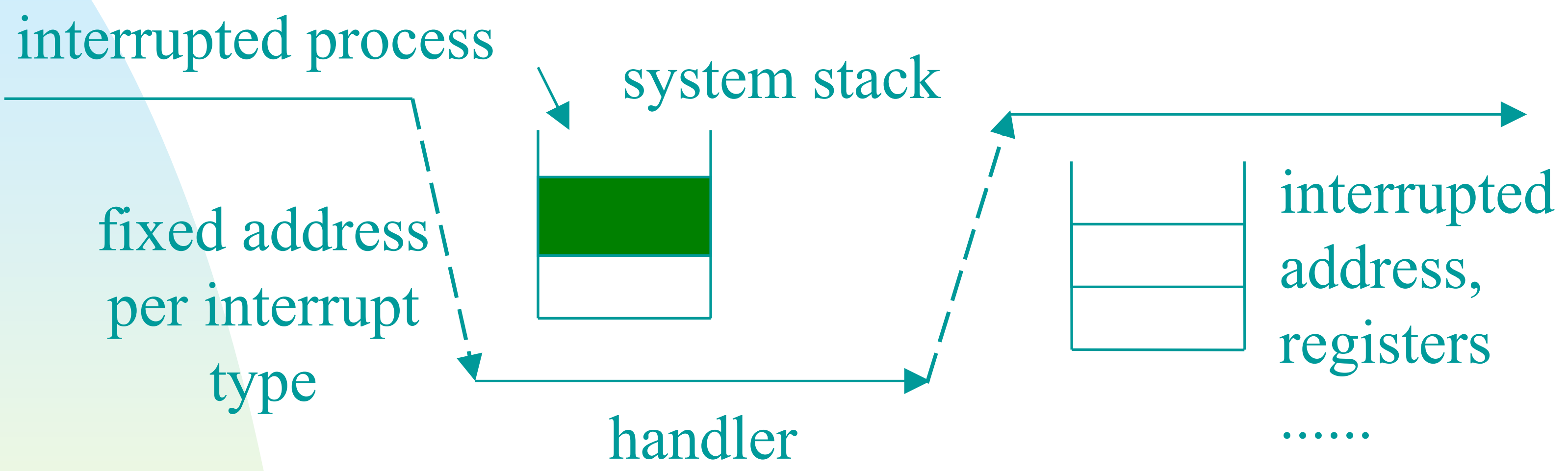
- Bootstrap program:
  - Initialize all aspects of the system, e.g., CPU registers, device controllers, memory, etc.
  - Load and run the OS
- Operating system: run *init* to initialize system processes, e.g., various daemons, login processes, after the kernel has been bootstrapped. (/etc/rc* & init or /sbin/rc* & init)

Interrupt

- Hardware interrupt, e.g. services requests of I/O devices
- Software interrupt, e.g. signals, invalid memory access, division by zero, system calls, etc – (trap)

- Procedures: generic handler or interrupt vector (MS-DOS,UNIX)

Interrupt Handling Procedure

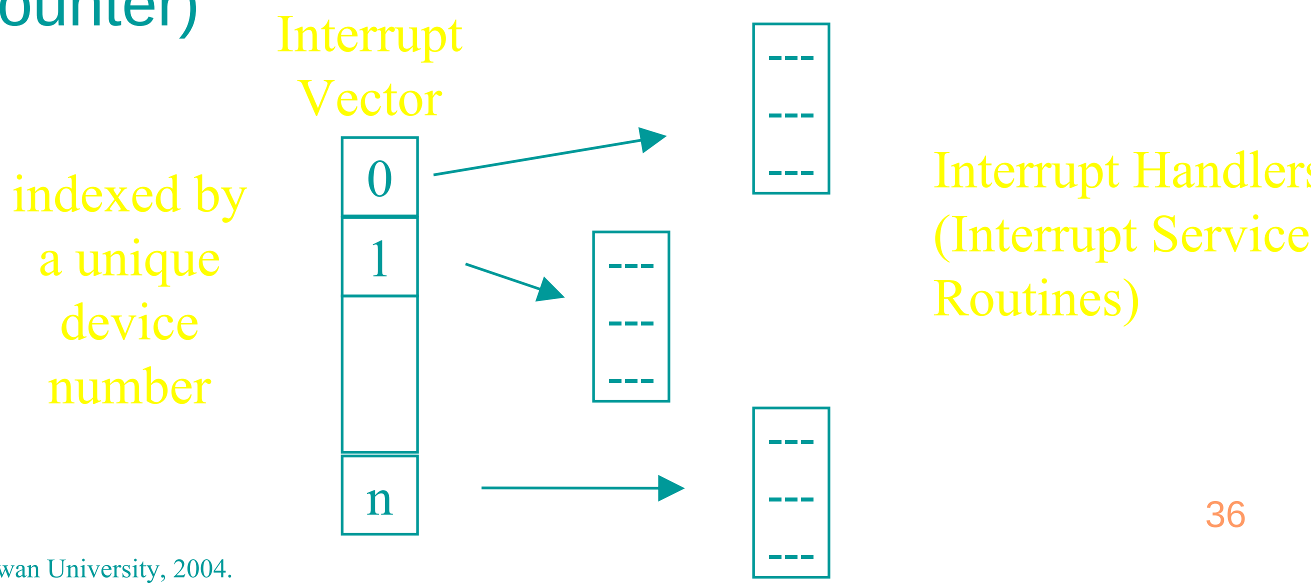


- Saving of the address of the interrupted instruction: fixed locations or stacks
- Interrupt disabling or enabling issues: lost interrupt?!

prioritized interrupts → masking

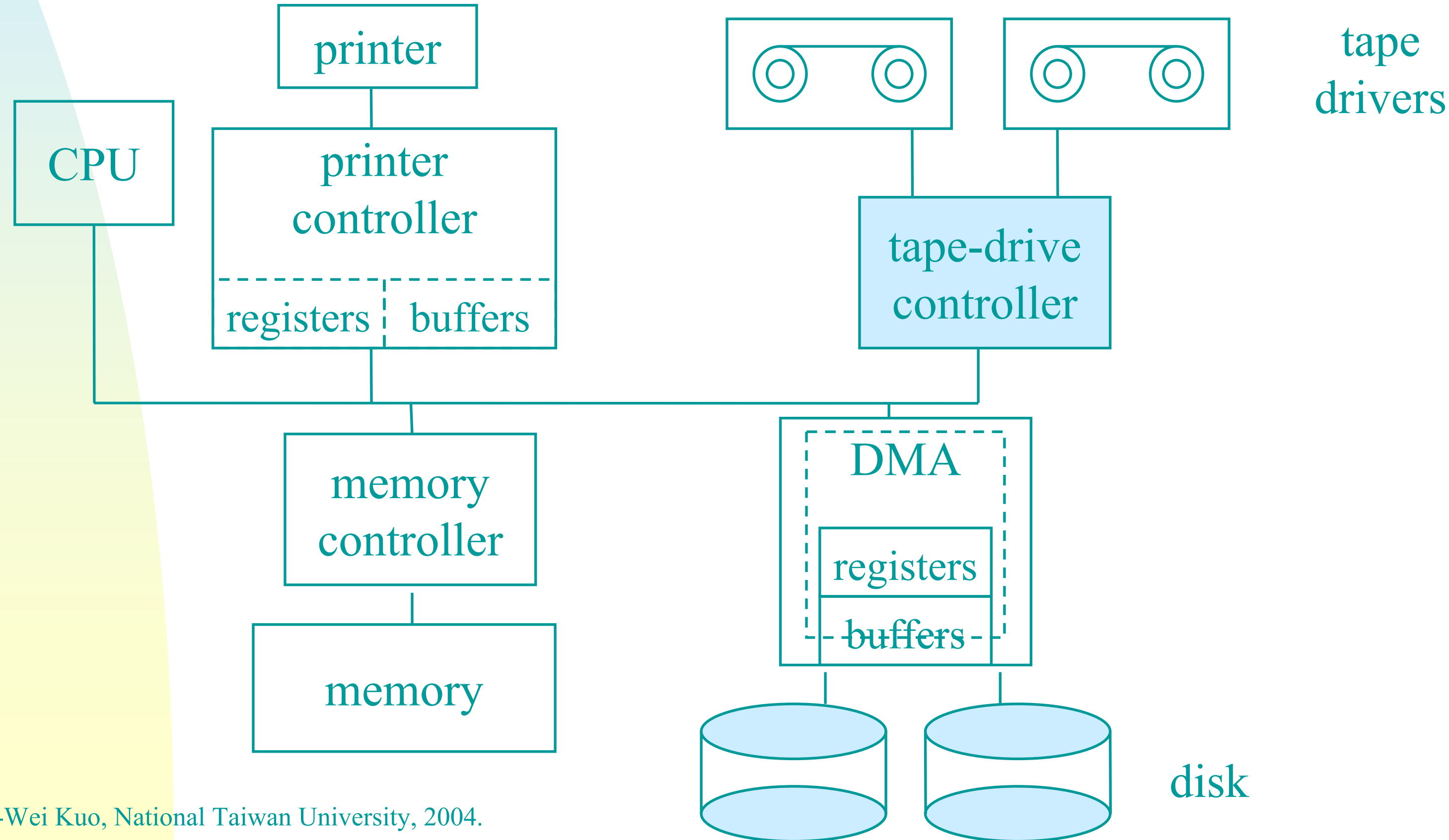
Interrupt Handling Procedure

- Interrupt Handling
  - → Save interrupt information
  - → OS determine the interrupt type (by polling)
  - → Call the corresponding handlers
  - → Return to the interrupted job by the restoring important information (e.g., saved return addr. → program counter)



I/O Structure

- Device controllers are responsible of moving data between the peripheral devices and their local buffer storages.



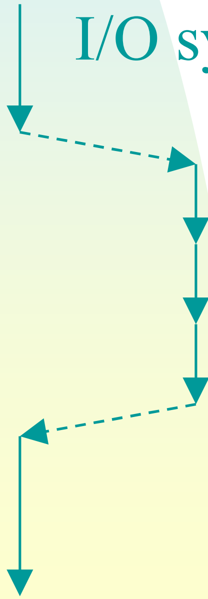
I/O Structure

- I/O operation
  a. CPU sets up specific controller registers within the controller.
  b. Read: devices → controller buffers → memory

     Write: memory → controller buffers → devices
  c. Notify the completion of the operation by triggering an interrupt

I/O Types

a. Synchronous I/O
   - Issues: overlapping of computations and IO activities, concurrent I/O activities, etc.



I/O system call

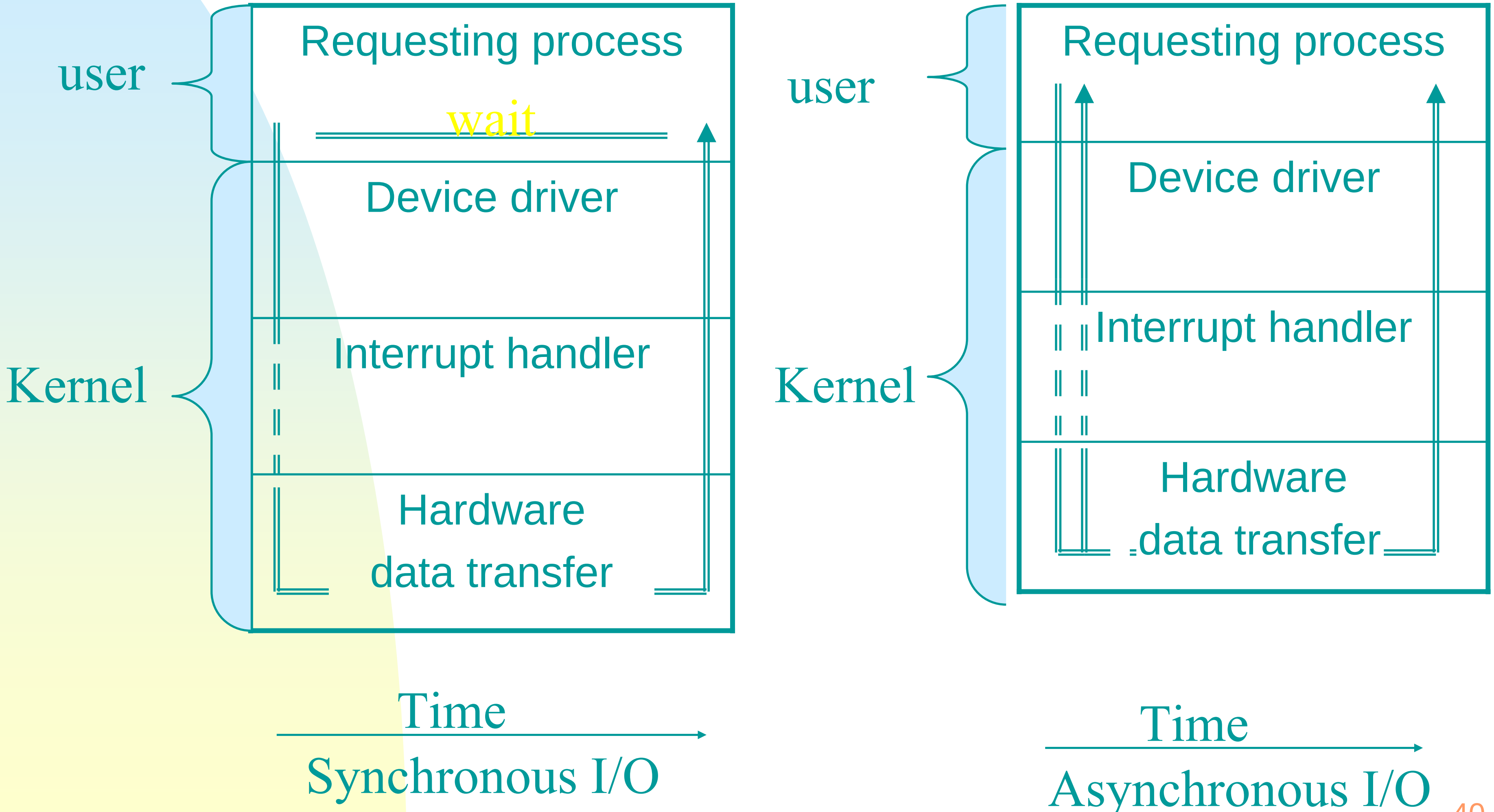
wait till the completion

or
- wait instruction (idle till interrupted)
- looping

or
- polling
- wait for an interrupt

Loop: jmp *Loop*

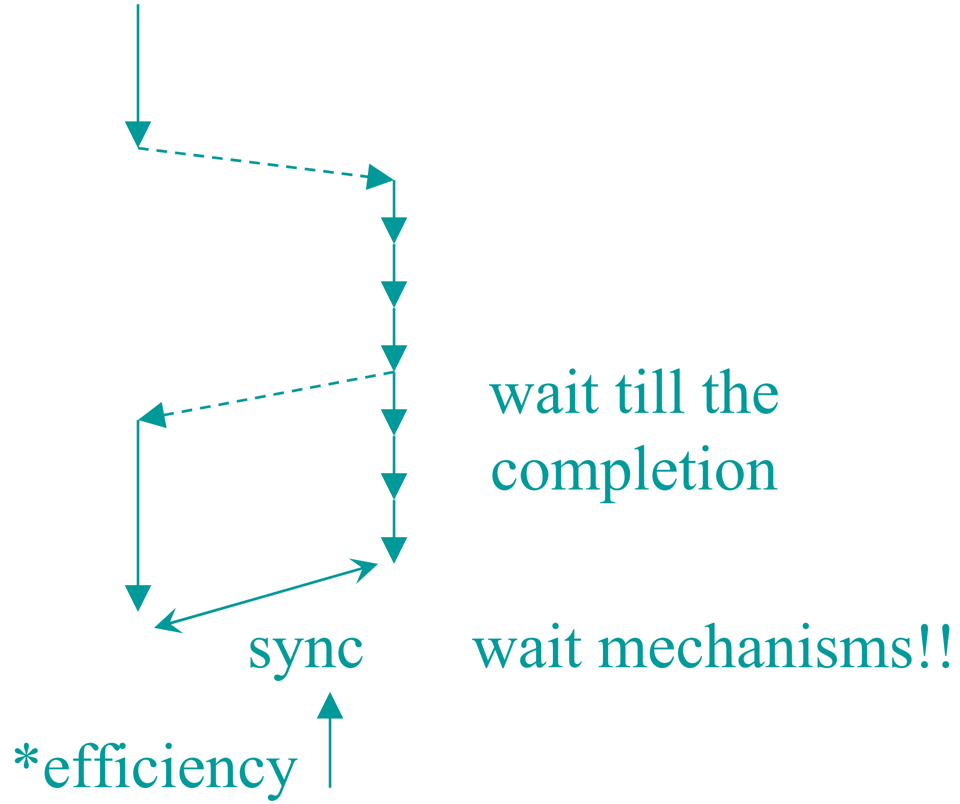
I/O Types



user
Kernel
Requesting process
wait
Device driver
Interrupt handler
Hardware
data transfer
Time
Synchronous I/O
user
Kernel
Requesting process
Device driver
Interrupt handler
Hardware
data transfer
Time
Asynchronous I/O

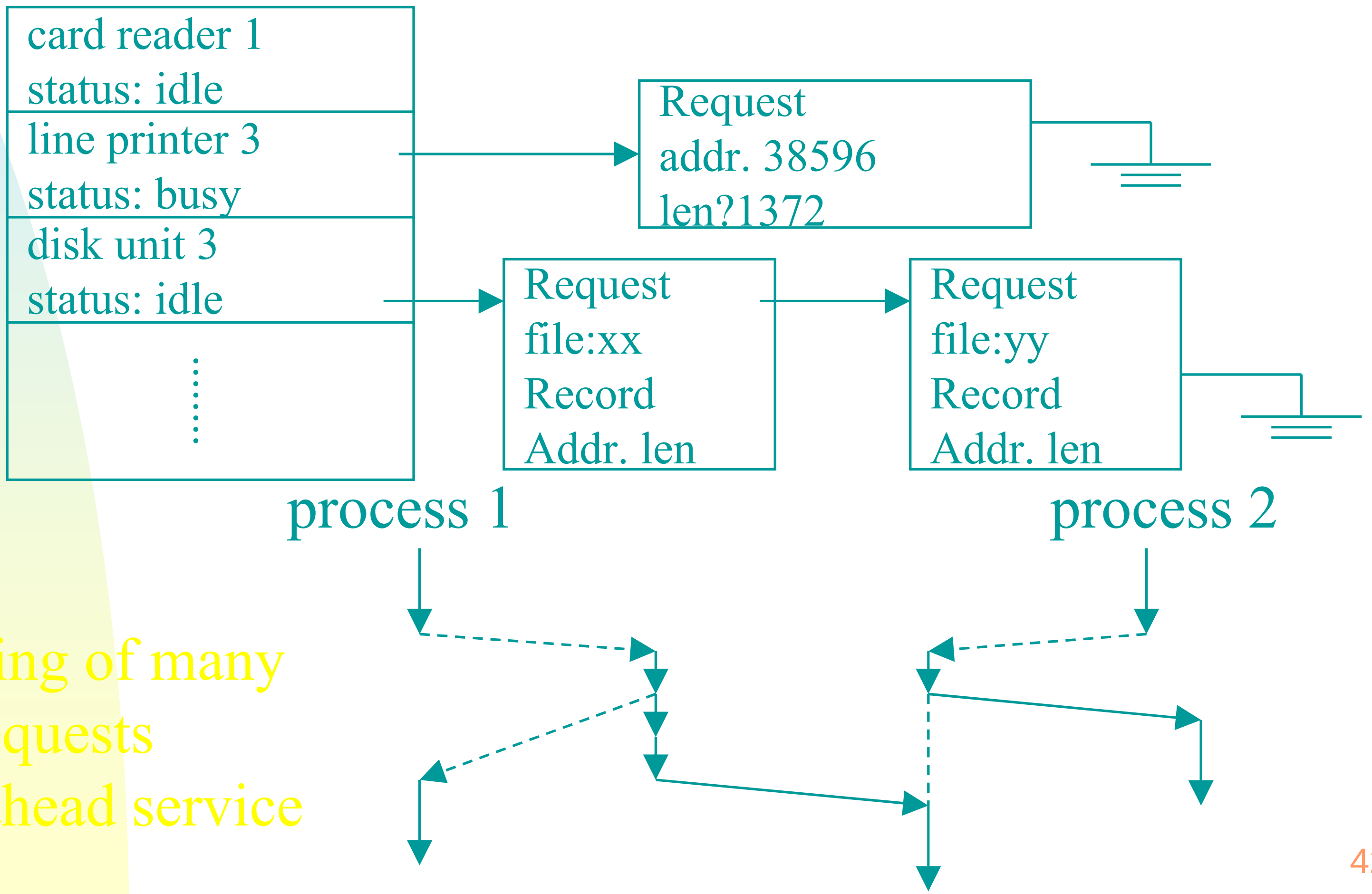
I/O types

b. Asynchronous I/O



I/O Types

- A Device-Status Table Approach



- Tracking of many I/O requests
- type-ahead service

DMA

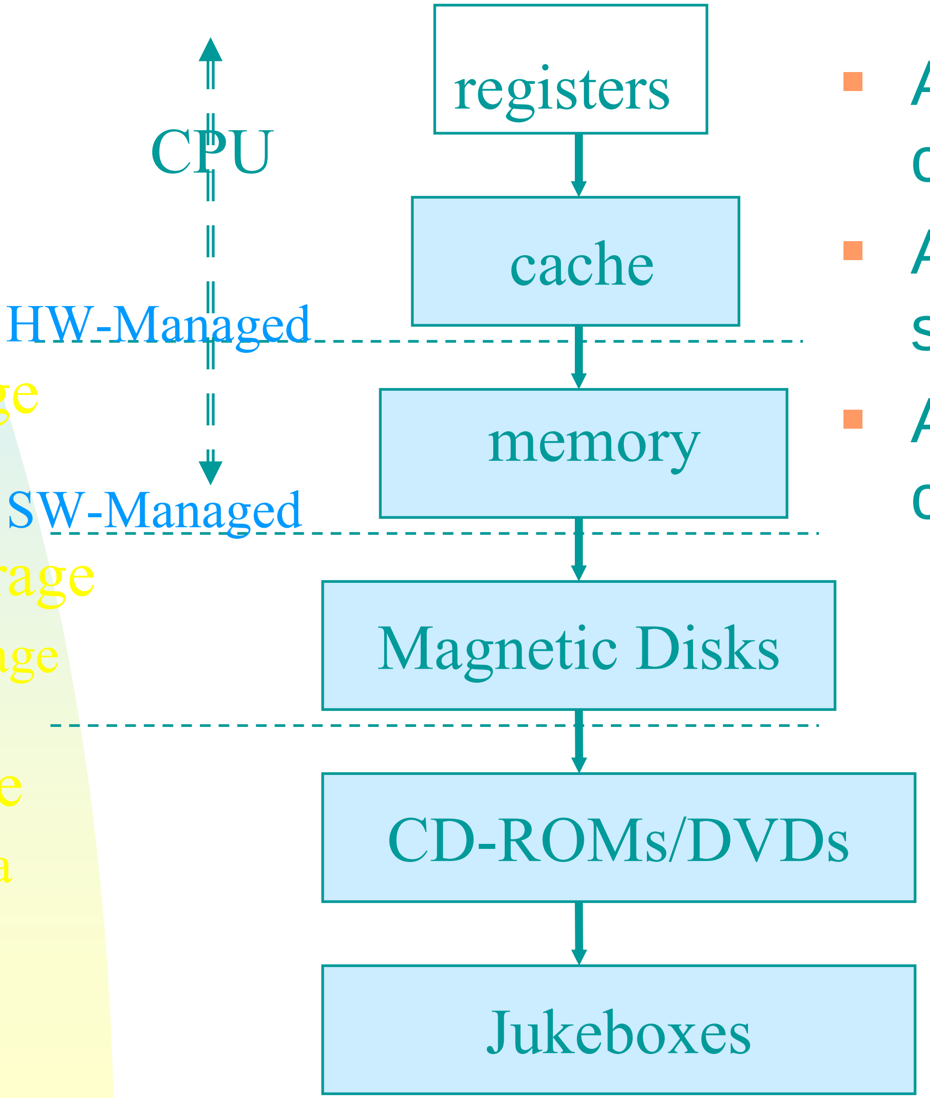
- Goal: Release CPU from handling excessive interrupts!
  - E.g. 9600-baud terminal

    2-microsecond service / 1000 microseconds

    High-speed device:

    2-microsecond service / 4 microseconds
- Procedure
  - Execute the device driver to set up the registers of the DMA controller.
  - DMA moves blocks of data between the memory and its own buffers.
  - Transfer from its buffers to its devices.
  - Interrupt the CPU when the job is done.

Storage Structure



Primary Storage
- volatile storage

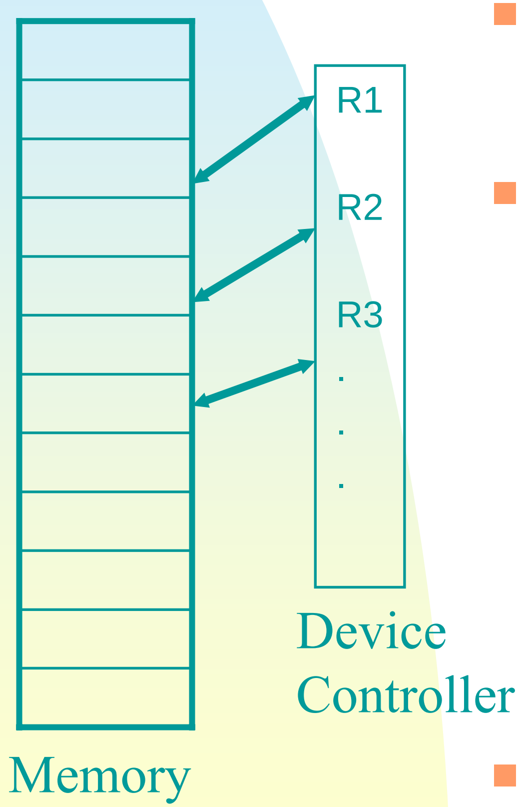
Secondary Storage
- nonvolatile storage

Tertiary Storage
- removable media

- Access time: a cycle
- Access time: several cycles
- Access time: many cycles

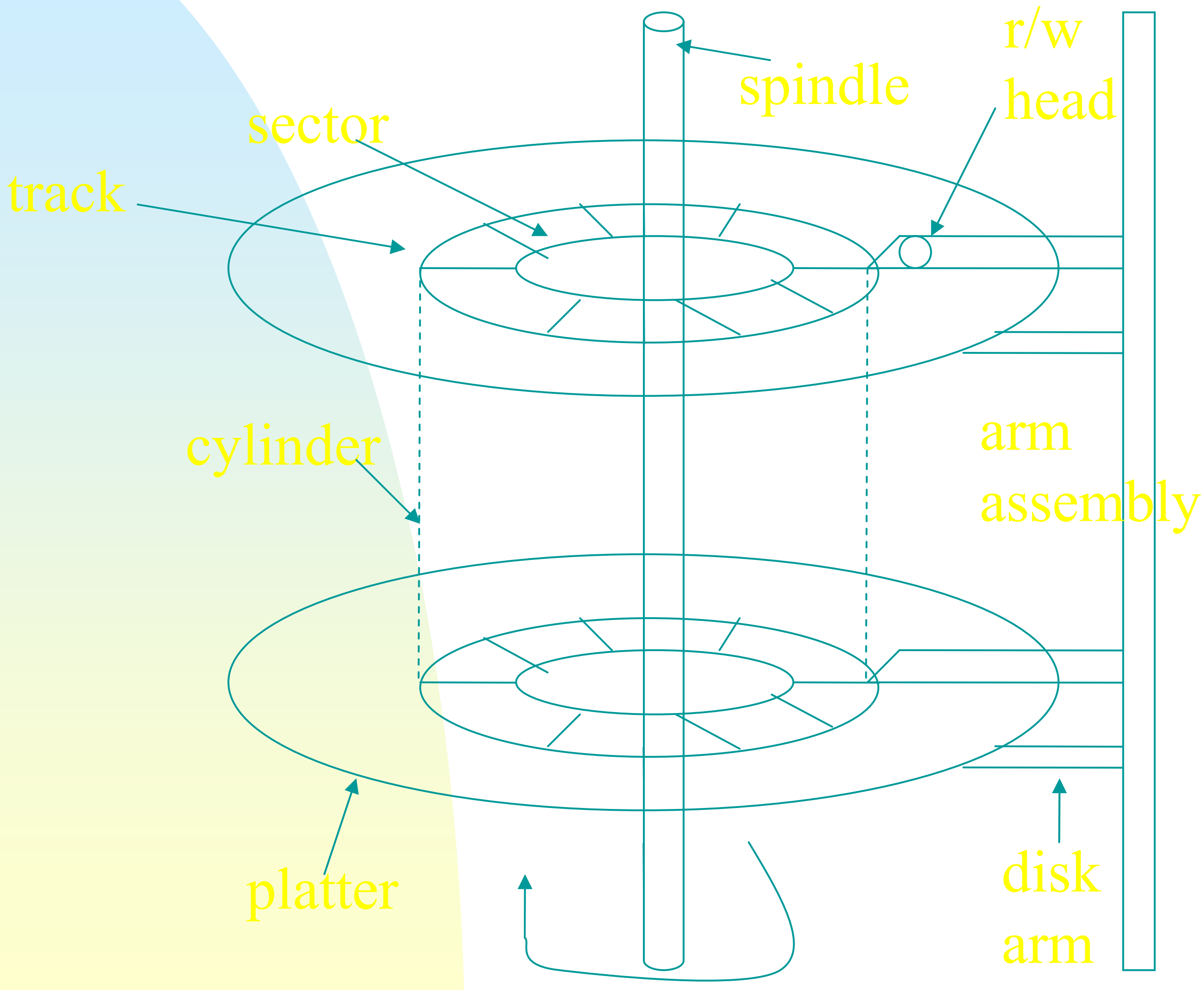
* Differences: Size, Cost, Speed, Volatility

Memory



- Processor can have direct access!
- Intermediate storage for data in the registers of device controllers
- Memory-Mapped I/O (PC & Mac)
  (1) Frequently used devices
  (2) Devices must be fast, such as video controller, or special I/O instructions is used to move data between memory & device controller registers
- Programmed I/O – polling
  - or interrupt-driven handling

Magnetic disks

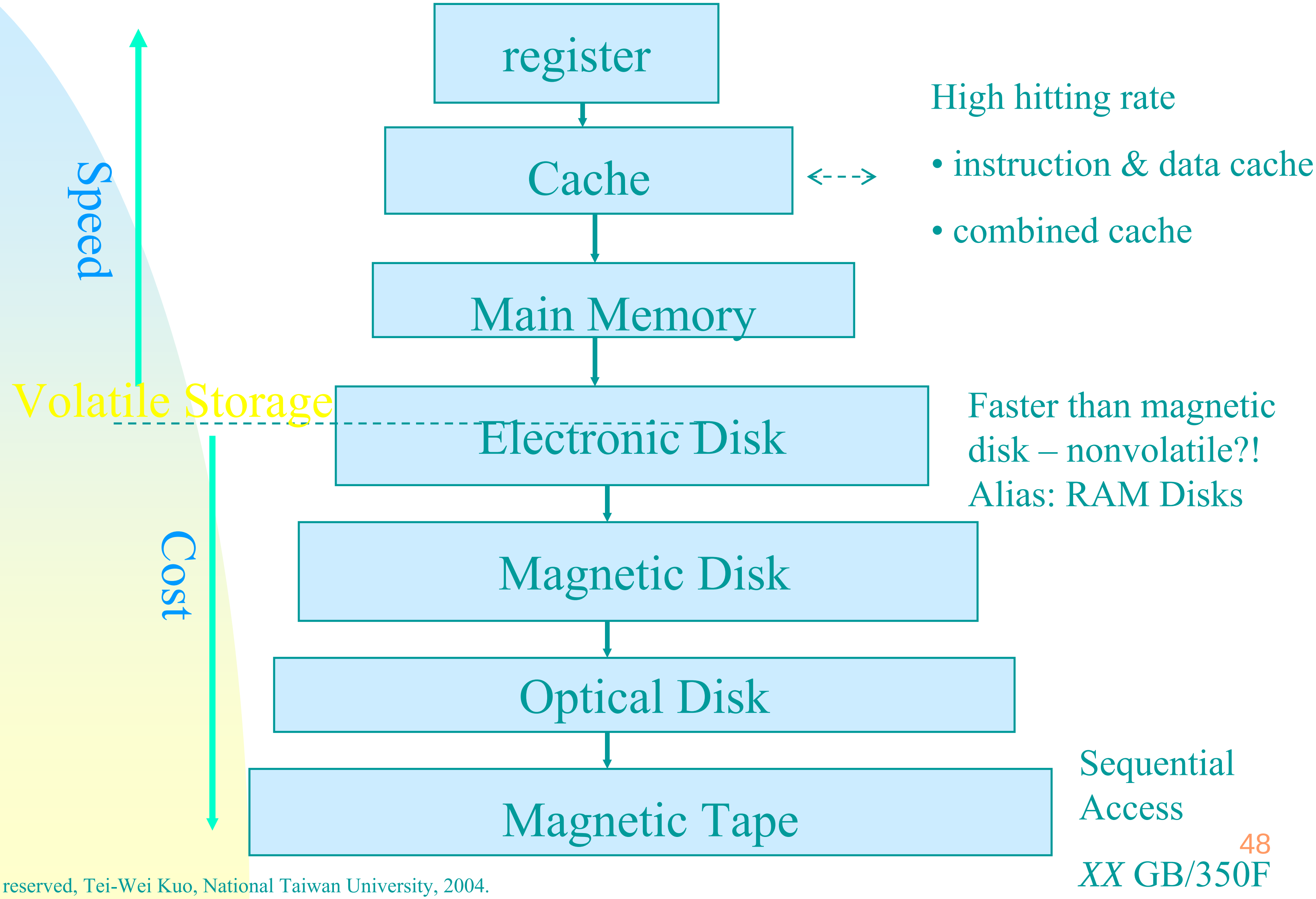


- Transfer Rate
- Random-Access Time
  - Seek time in *x* ms
  - Rotational latency in *y* ms
    - 60~200 times/sec

Magnetic Disks

- Disks
  - Fixed-head disks:
    - More r/w heads v.s. fast track switching
  - Moving-head disks (hard disk)
  - Primary concerns:
    - Cost, Size, Speed
  - Computer → host controller → disk controller → disk drives (cache ←→ disks)
- Floppy disk
  - slow rotation, low capacity, low density, but less expensive
- Tapes: backup or data transfer bet machines

Storage Hierarchy



Storage Hierarchy

- Caching
  - Information is copied to a faster storage system on a temporary basis
  - Assumption: Data will be used again soon.
    - Programmable registers, instr. cache, etc.
- Cache Management
  - Cache Size and the Replacement Policy
- Movement of Information Between Hierarchy
  - Hardware Design & Controlling Operating Systems

Storage Hierarchy

- Coherency and Consistency
  - Among several storage levels (vertical)
    - Multitasking vs unitasking
  - Among units of the same storage level , (horizontal), e.g. cache coherency
    - Multiprocessor or distributed systems

Hardware Protection

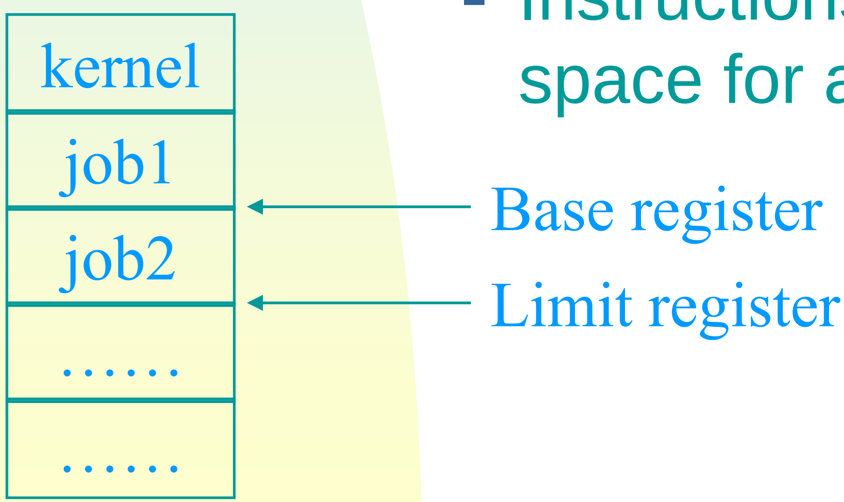
- Goal:
  - Prevent errors and misuse!
    - E.g., input errors of a program in a simple batch operating system
    - E.g., the modifications of data and code segments of another process or OS
- Dual-Mode Operations – a mode bit
  - User-mode executions except those after a trap or an interrupt occurs.
  - Monitor-mode (system mode, privileged mode, supervisor mode)
    - Privileged instruction:machine instructions that may cause harm

Hardware Protection

- System Calls – trap to OS for executing privileged instructions.
- Resources to protect
  - I/O devices, Memory, CPU
- I/O Protection (I/O devices are scare resources!)
  - I/O instructions are privileged.
    - User programs must issue I/O through OS
    - User programs can never gain control over the computer in the system mode.

Hardware Protection

- Memory Protection
  - Goal: Prevent a user program from modifying the code or data structures of either the OS or other users!
  - Instructions to modify the memory space for a process are privileged.



⇔ Check for every memory address by hardware

Hardware Protection

- CPU Protection
  - Goal
    - Prevent user programs from sucking up CPU power!
  - Use a timer to implement time-sharing or to compute the current time.
    - Instructions that modify timers are privileged.
  - Computer control is turned over to OS for every time-slice of time!
    - Terms: time-sharing, context switch

Network Structure

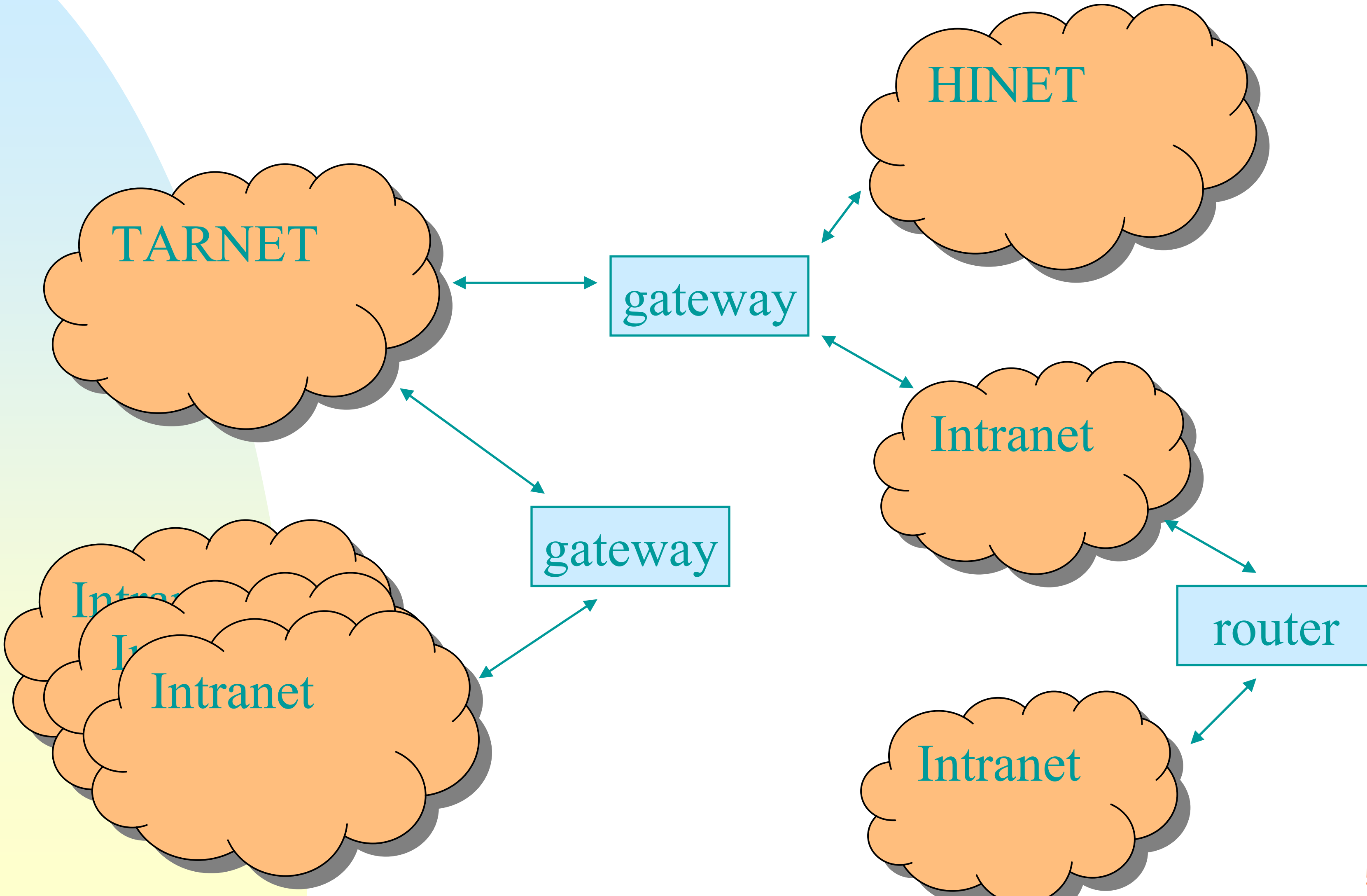
- Local-Area Network (LAN)
  - Characteristics:
    - Geographically distributed in a small area, e.g., an office with different computers and peripheral devices.
    - More reliable and better speed
      - High-quality cables, e.g., twisted pair cables for 10BaseT Ethernet or fiber optic cables for 100BaseT Ethernet
  - Started in 1970s
  - Configurations: multiaccess bus, ring, star networks (with gateways)

Network Structure

- Wide-Area Network (WAN)
  - Emerged in late 1960s (Arpanet in 1968)
- World Wide Web (WWW)
  - Utilize TCP/IP over ARPANET/Internet.

- Definition of "Intranet": roughly speaking for any network under one authorization, e.g., a company or a school.
  - Often in a Local Area Network (LAN), or connected LAN's.
  - Having one (or several) gateway with the outside world.
  - In general, it has a higher bandwidth because of a LAN.

Network Structure – WAN



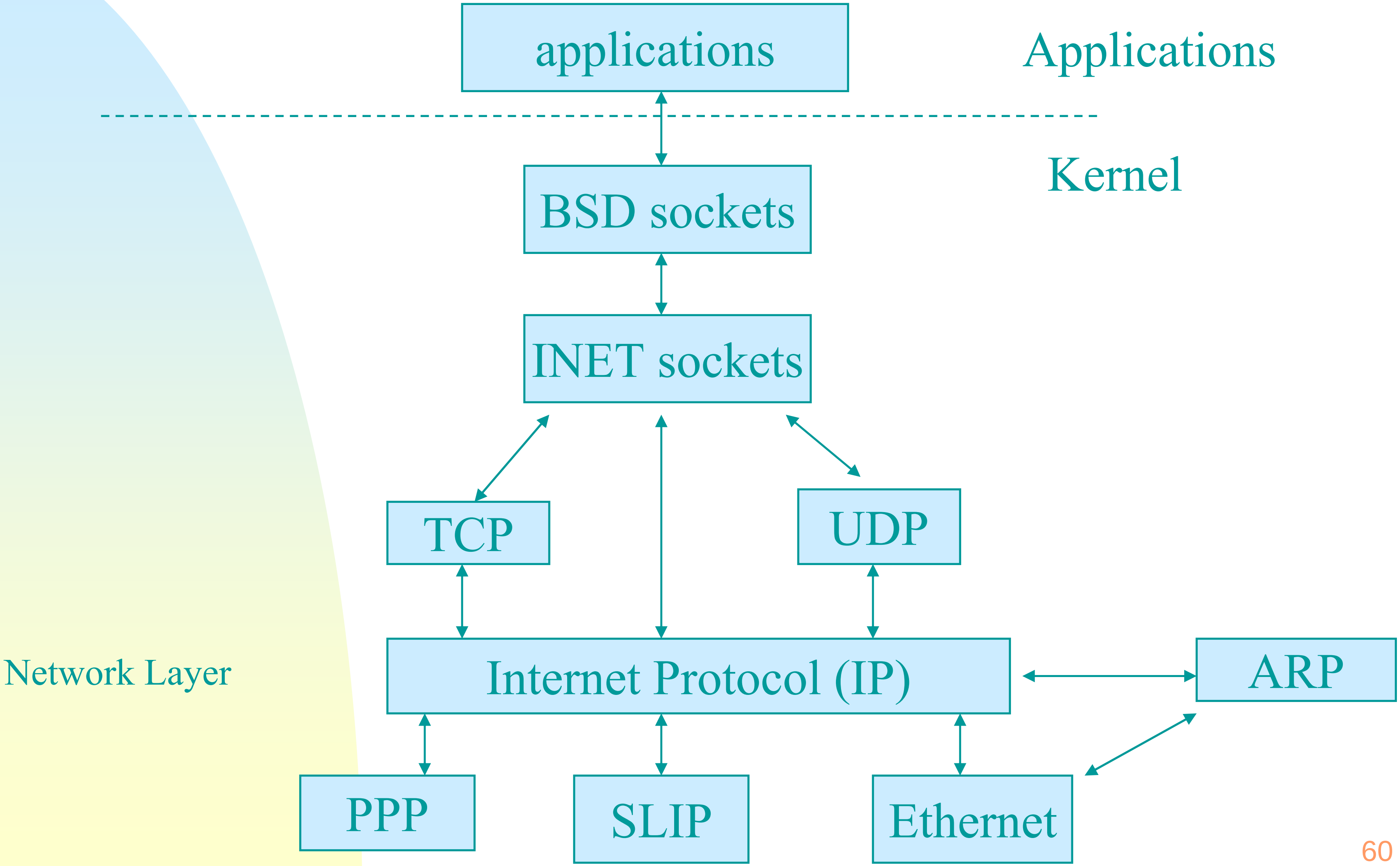
Network Structure – WAN

- Router
  - With a Routing table
    - Use some routing protocol, e.g., to maintain network topology by broadcasting.
  - Connecting several subnets (of the same IP-or-higher-layer protocols) for forwarding packets to proper subnets.
- Gateway
  - Functionality containing that of routers.
  - Connecting several subnets (of different or the same networks, e.g., Bitnet and Internet)for forwarding packets to proper subnets.

Network Structure – WAN

- Connections between networks
  - T1: 1.544 mbps, T3: 45mbps (28T1)
    - Telephone-system services over T1
- Modems
  - Conversion of the analog signal and digital signal

Network Layers in Linux



TCP/IP

- IP Address:
  - 140.123.101.1
    - 256*256*256*256 combinations
      - 140.123 -> Network Address
      - 101.1 -> Host Address
    - Subnet:
      - 140.123.101 and 140.123.102
  - Mapping of IP addresses and host names
    - Static assignments: /etc/hosts
    - Dynamic acquisition: DNS (Domain Name Server)
      - /etc/resolv.confg
    - If /etc/hosts is out-of-date, re-check it up with DNS!
  - Domain name: cs.ccu.edu.tw as a domain name for 140.123.100, 140.123. 101, and 140.123.103

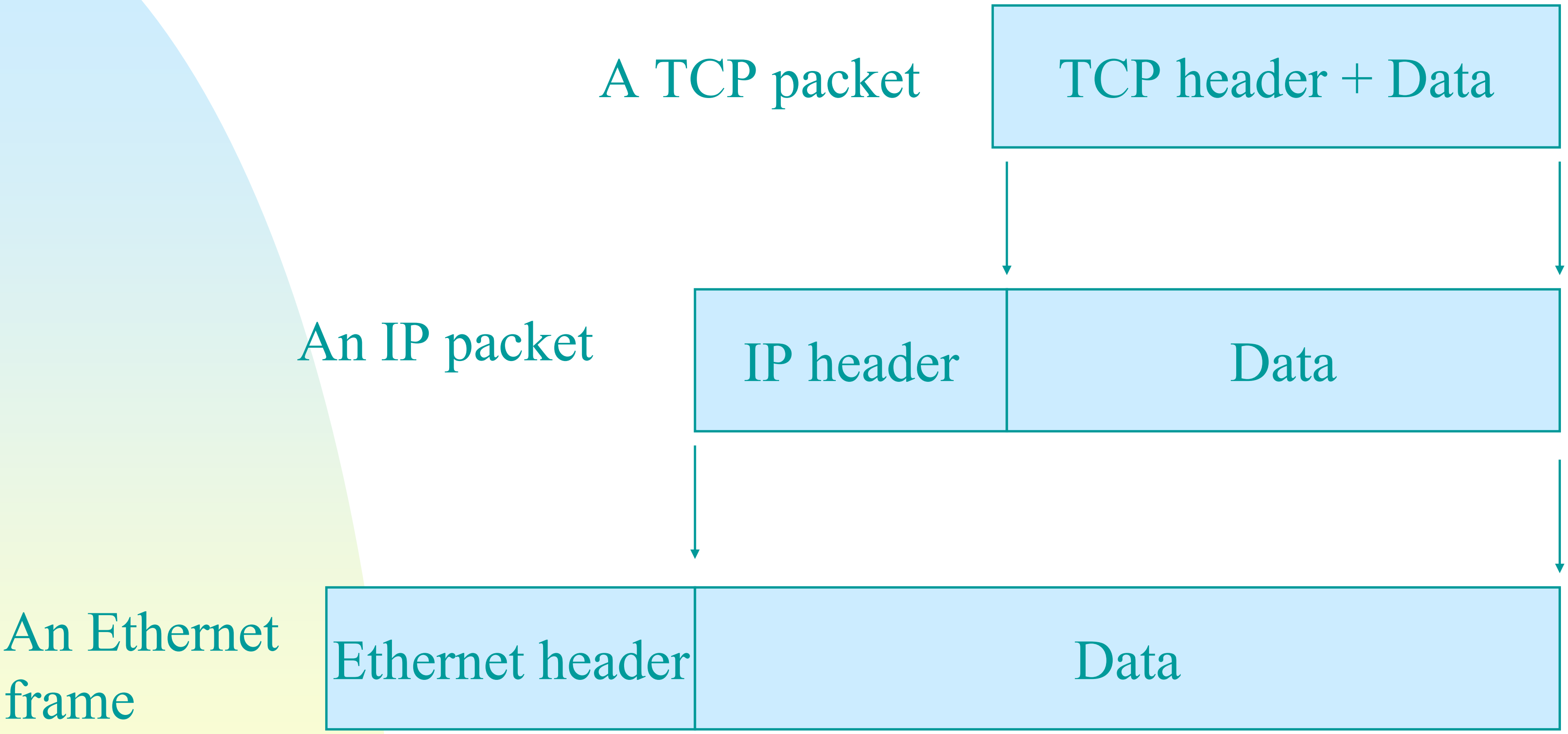
TCP/IP

- Transmission Control Protocol (TCP)
  - Reliable point-to-point packet transmissions.
  - Applications which communicate over TCP/IP with each another must provide IP addresses and port numbers.
    - /etc/services
    - Port# 80 for a web server.
- User Datagram Protocol (UDP)
  - Unreliable point-to-point services.
- Both are over IP.

TCP/IP

- Mapping of Ethernet physical addresses and IP addresses
  - Each Ethernet card has a built-in Ethernet physical address, e.g., 08-01-2b-00-50-A6.
  - Ethernet cards only recognize frames with their physical addresses.
  - Linux uses ARP (Address Resolution Protocol) to know and maintain the mapping.
    - Broadcast requests over Ethernet for IP address resolution over ARP.
    - Machines with the indicated IP addresses reply with their Ethernet physical addresses.

TCP/IP



- Each IP packet has an indicator of which protocol used, e.g., TCP or UDP

Contents

Chapter 3 Operating-System Structures

Operating-System Structures

- Goals: Provide a way to understand an operating systems
  - Services
  - Interface
  - System Components
- The type of system desired is the basis for choices among various algorithms and strategies!

System Components – Process Management

- Process Management
  - Process: An Active Entity
    - Physical and Logical Resources
      - Memory, I/O buffers, data, etc.
    - Data Structures Representing Current Activities:

Program (code) + Program Counter
Stack
Data Section
CPU Registers
….
And More

System Components – Process Management

- Services
  - Process creation and deletion
  - Process suspension and resumption
  - Process synchronization
  - Process communication
  - Deadlock handling

System Components – Main-Memory Management

- Memory: a large array of words or bytes, where each has its own address
- OS must keep several programs in memory to improve CPU utilization and user response time
- Management algorithms depend on the hardware support
- Services
  - Memory usage and availability
  - Decision of memory assignment
  - Memory allocation and deallocation

System Components – File Management

- Goal:
  - A uniform logical view of information storage
  - Each medium controlled by a device
    - Magnetic tapes, magnetic disks, optical disks, etc.
- OS provides a logical storage unit: File
  - Formats:
    - Free form or being formatted rigidly.
  - General Views:
    - A sequence of bits, bytes, lines, records

System Components – File Management

- Services
  - File creation and deletion
  - Directory creation and deletion
  - Primitives for file and directory manipulation
  - Mapping of files onto secondary storage
  - File Backup

* Privileges for file access control

System Components – I/O System Management

- Goal:
  - Hide the peculiarities of specific hardware devices from users
- Components of an I/O System
  - A buffering, caching, and spooling system
  - A general device-driver interface
  - Drivers

System Components – Secondary-Storage Management

- Goal:
  - On-line storage medium for programs & data
    - Backup of main memory
- Services for Disk Management
  - Free-space management
  - Storage allocation, e.g., continuous allocation
  - Disk scheduling, e.g., FCFS

System Components – Networking

- Issues
  - Resources sharing
  - Routing & connection strategies
  - Contention and security
- Network access is usually generalized as a form of file access
  - World-Wide-Web over file-transfer protocol (ftp), network file-system (NFS), and hypertext transfer protocol (http)

System Components – Protection System

- Goal
  - Resources are only allowed to accessed by authorized processes.
- Protected Resources
  - Files, CPU, memory space, etc.
- Services
  - Detection & controlling mechanisms
  - Specification mechanisms
- Remark: Reliability!

System Components – Command-Interpreter system

- Command Interpreter
  - Interface between the user and the operating system
  - Friendly interfaces
    - Command-line-based interfaces or mused-based window-and-menu interface
  - e.g., UNIX shell and command.com in MS-DOS

User-friendly?

Get the next command
Execute the command

Operation-System Services

- Program Execution
  - Loading, running, terminating, etc
- I/O Operations
  - General/special operations for devices:
    - Efficiency & protection
- File-System Manipulation
  - Read, write, create, delete, etc
- Communications
  - Intra-processor or inter-processor communication – shared memory or message passing

Operation-System Services

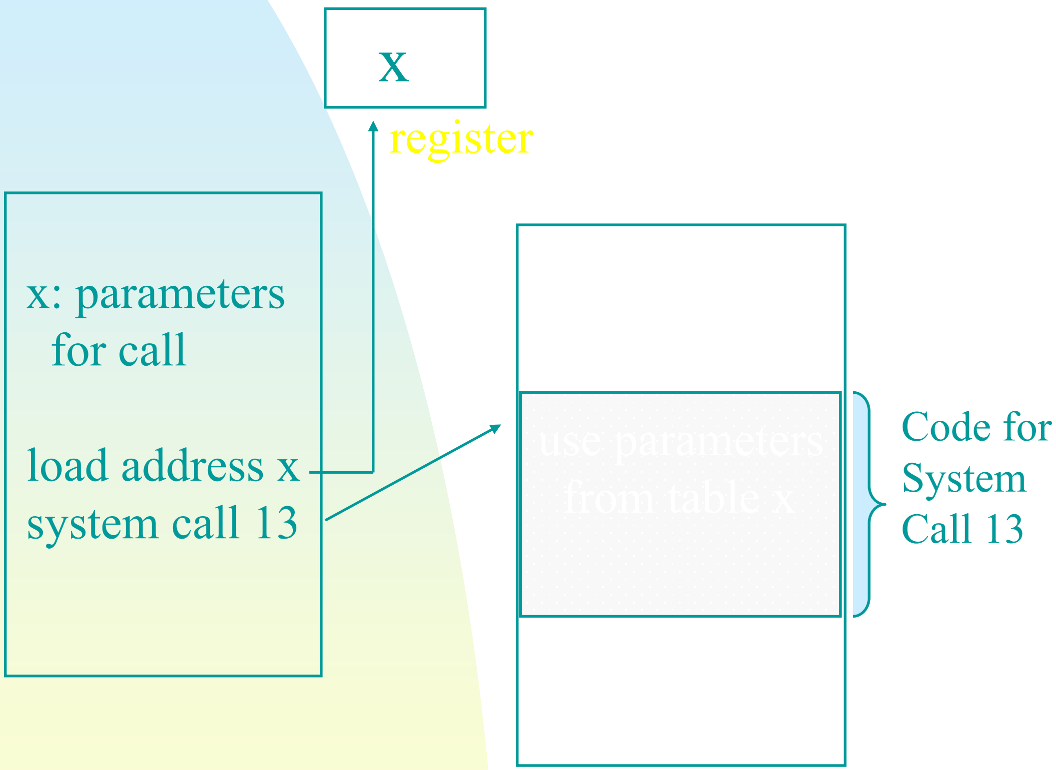
- Error Detection
  - Possible errors from CPU, memory, devices, user programs → Ensure correct & consistent computing
- *Resource Allocation*
  - *Utilization & efficiency*
- *Accounting*
- *Protection & Security*

• user convenience <u>or</u> *system efficiency!*

Operation-System Services

- System calls
  - Interface between processes & OS
- How to make system calls?
  - Assemble-language instructions or subroutine/functions calls in high-level language such as C or Perl?
    - Generation of in-line instructions or a call to a special run-time routine.
- Example: read and copy of a file!
  - Library Calls vs System Calls

Operation-System Services



- How a system call occurs?
  - Types and information
- Parameter Passing
  - Registers
  - Registers pointing to blocks
    - Linux
  - Stacks

Operation-System Services

- System Calls
  - Process Control
  - File Management
  - Device Management
  - Information Maintenance
  - Communications

Operation-System Services

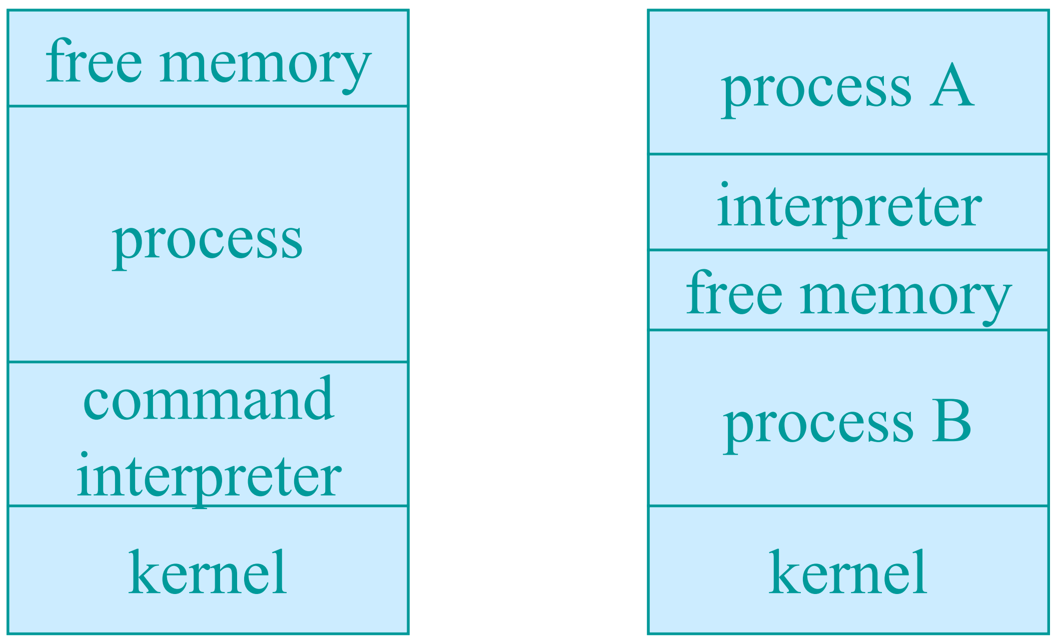
- Process & Job Control
  - End (normal exit) or abort (abnormal)
    - Error level or no
  - Load and execute
    - How to return control?
      - e.g., shell load & execute commands
  - Creation and/or termination of processes
    - Multiprogramming?

Operation-System Services

- Process & Job Control (continued)
  - Process Control
    - Get or set attributes of processes
  - Wait for a specified amount of time or an event
    - Signal event
  - Memory dumping, profiling, tracing, memory allocation & de-allocation

Operation-System Services

- Examples: MS-DOS & UNIX



Operation-System Services

- File Management
  - Create and delete
  - Open and close
  - Read, write, and reposition (e.g., rewinding)
    - lseek
  - Get or set attributes of files
  - Operations for directories

Operation-System Services

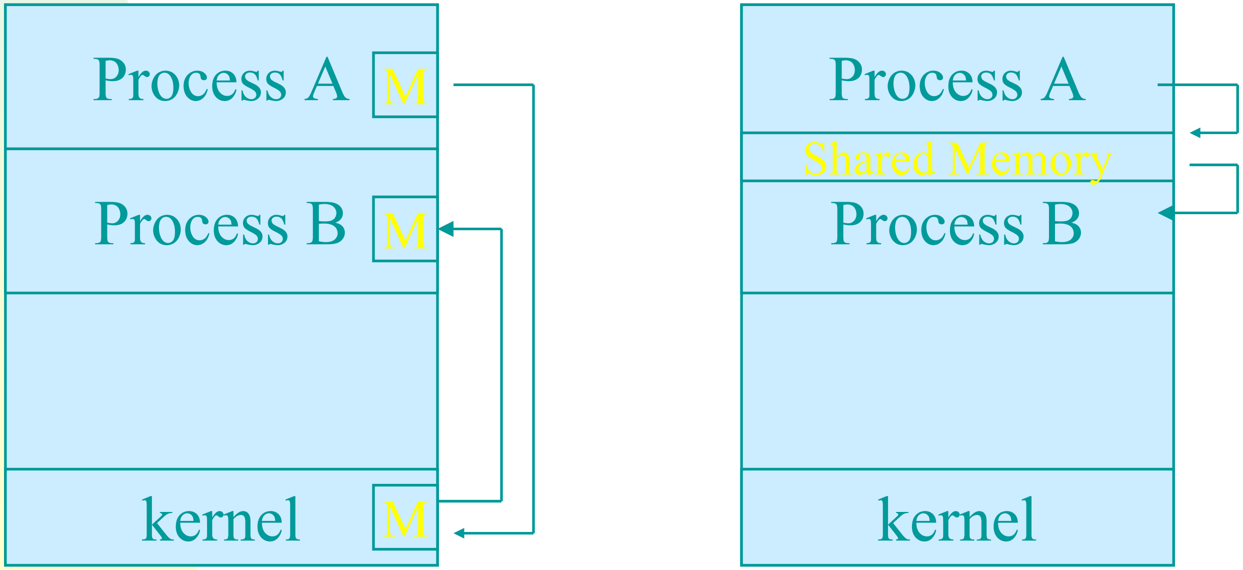
- Device management
  - Request or release
    - Open and close of special files
    - Files are abstract or virtual devices.
  - Read, write, and reposition (e.g., rewinding)
  - Get or set file attributes
  - Logically attach or detach devices

Operation-System Services

- Information maintenance
  - Get or set date or time
  - Get or set system data, such as the amount of free memory
- Communication
  - Message Passing
    - Open, close, accept connections
      - Host ID or process ID
    - Send and receive messages
    - Transfer status information
  - Shared Memory
    - Memory mapping & process synchronization

Operation-System Services

- Shared Memory
  - Max Speed & Comm Convenience
- Message Passing
  - No Access Conflict & Easy Implementation

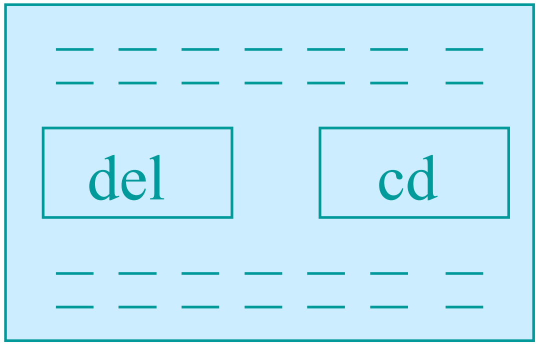


System Programs

- Goal:
  - Provide a convenient environment for program development and execution
- Types
  - File Management, e.g., rm.
  - Status information, e.g., date.
  - File Modifications, e.g., editors.
  - Program Loading and Executions, e.g., loader.
  - Programming Language Supports, e.g., compilers.
  - Communications, e.g., telnet.

System Programs – Command Interpreter

- Two approaches:
  - Contain codes to execute commands
    - Fast but the interpreter tends to be big!
    - Painful in revision!



System Programs – Command Interpreter

- Implement commands as system programs → Search exec files which corresponds to commands (UNIX)
  - Issues
    a. Parameter Passing
       - Potential Hazard: virtual memory
    b. Being Slow
    c. Inconsistent Interpretation of Parameters

System Structure – MS-DOS

- MS-DOS Layer Structure

Application program

↓

Resident system program

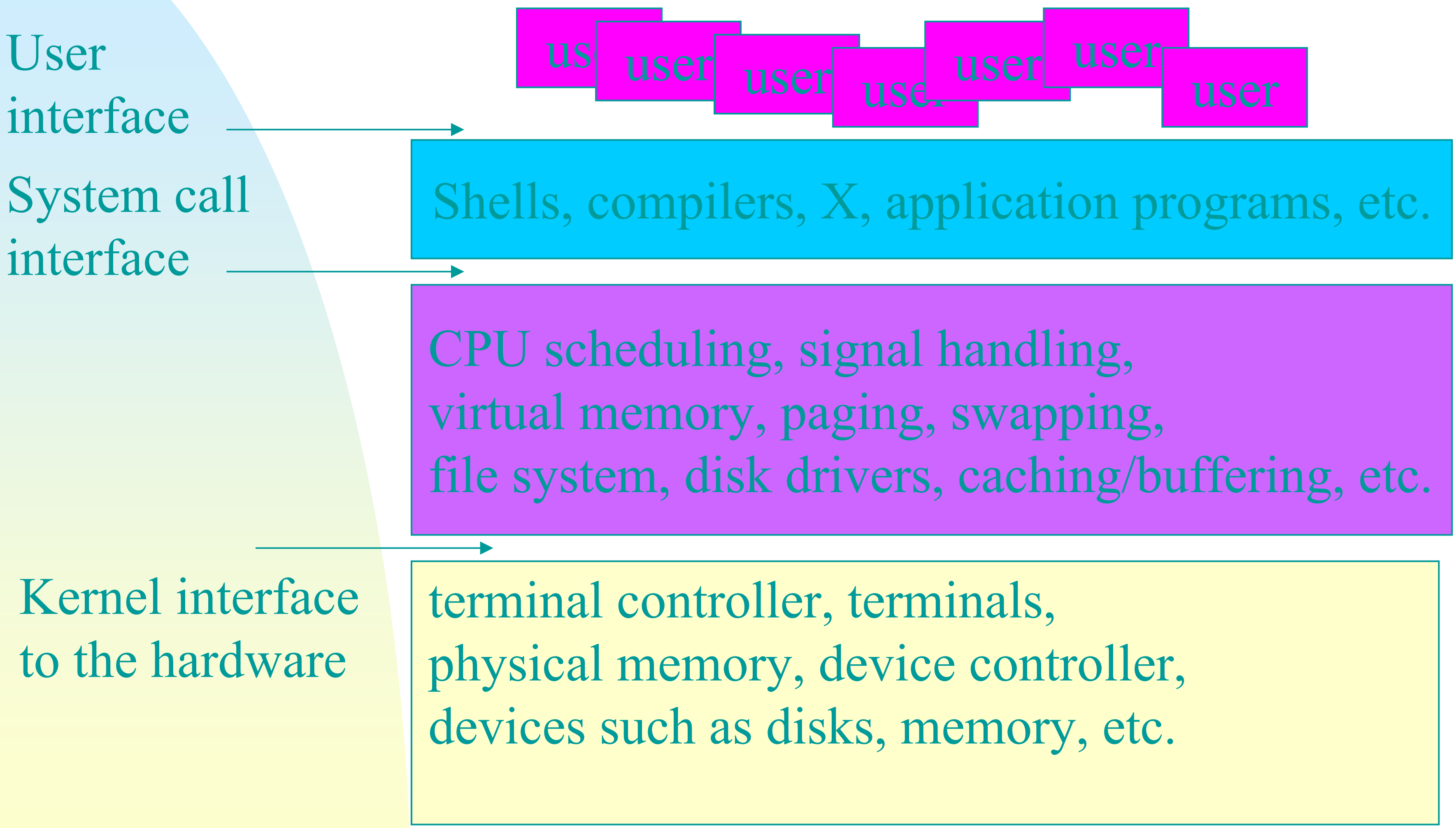
↓

MS-DOS device drivers

↓

ROM BIOS device drivers

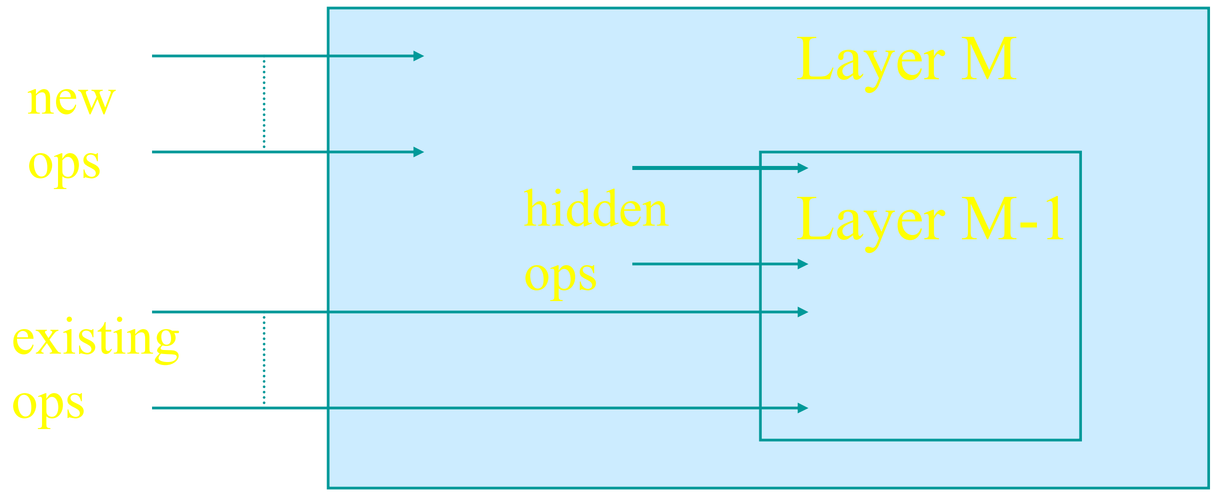
System Structure – UNIX



UNIX

System Structure

- A Layered Approach – A Myth



Advantage: Modularity ~ Debugging & Verification

Difficulty: Appropriate layer definitions, less efficiency due to overheads！

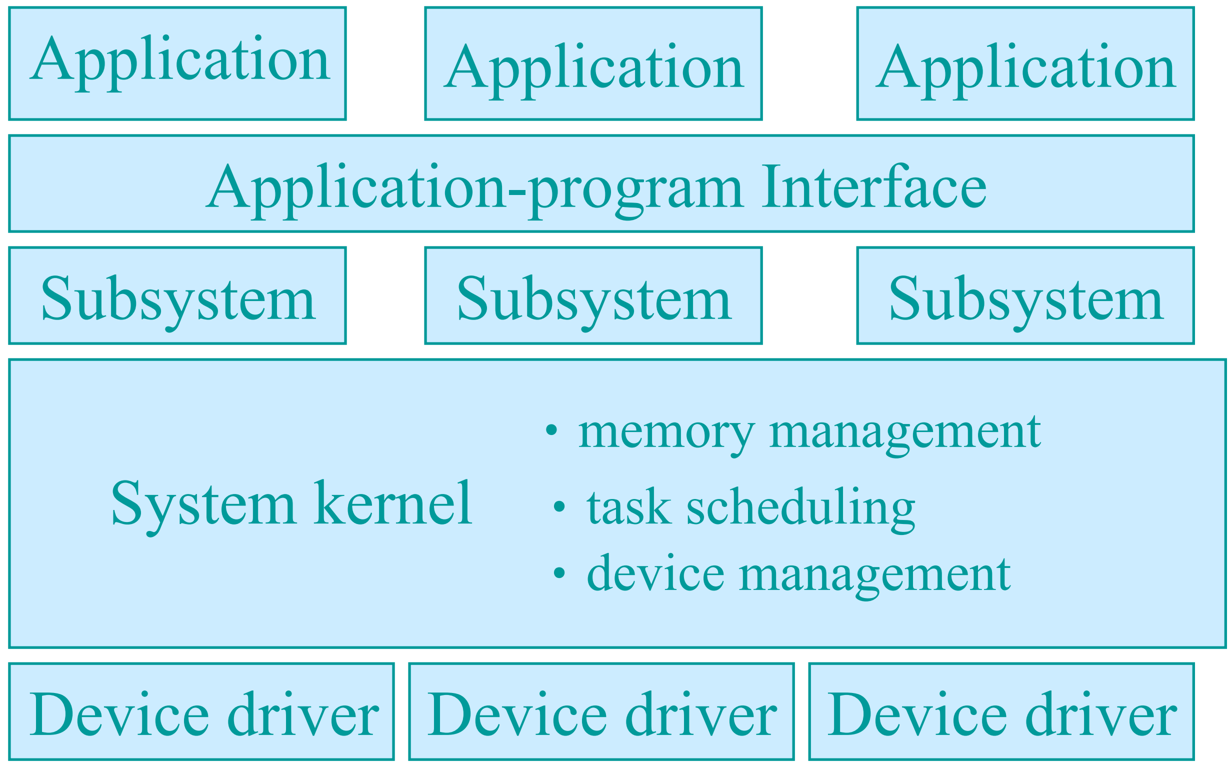
System Structure

- A Layer Definition Example:

L5	User programs
L4	I/O buffering
L3	Operator-console device driver
L2	Memory management
L1	CPU scheduling
L0	Hardware

System Structure – OS/2

- OS/2 Layer Structure



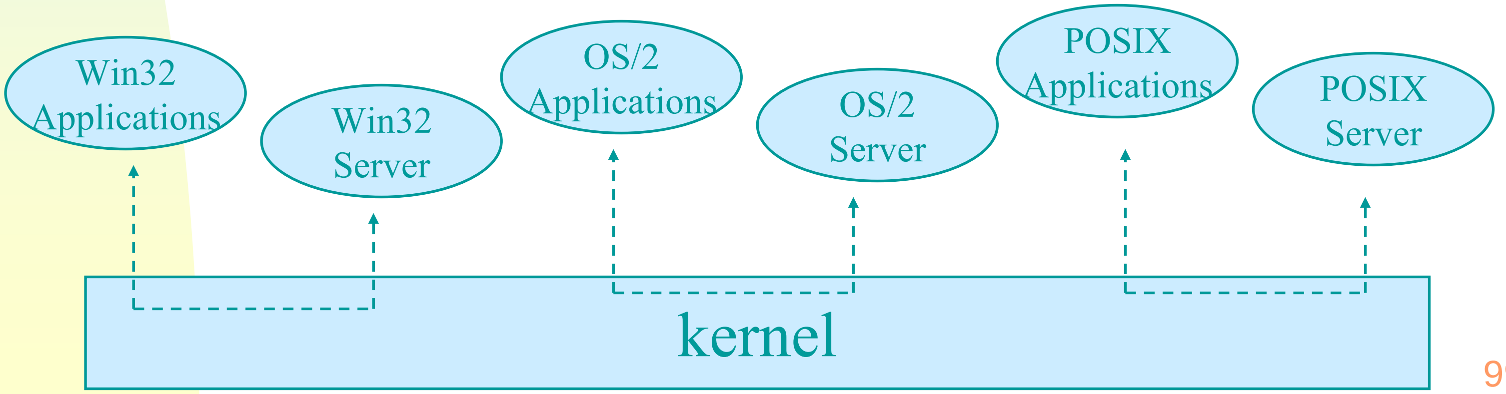
* Some layers of NT were from user space to kernel space in NT4.0

System Structure – Microkernels

- The concept of microkernels was proposed in CMU in mid 1980s (Mach).
  - Moving all nonessential components from the kernel to the user or system programs!
  - No consensus on services in kernel
    - Mostly on process and memory management and communication
- Benefits:
  - Ease of OS service extensions → portability, reliability, security

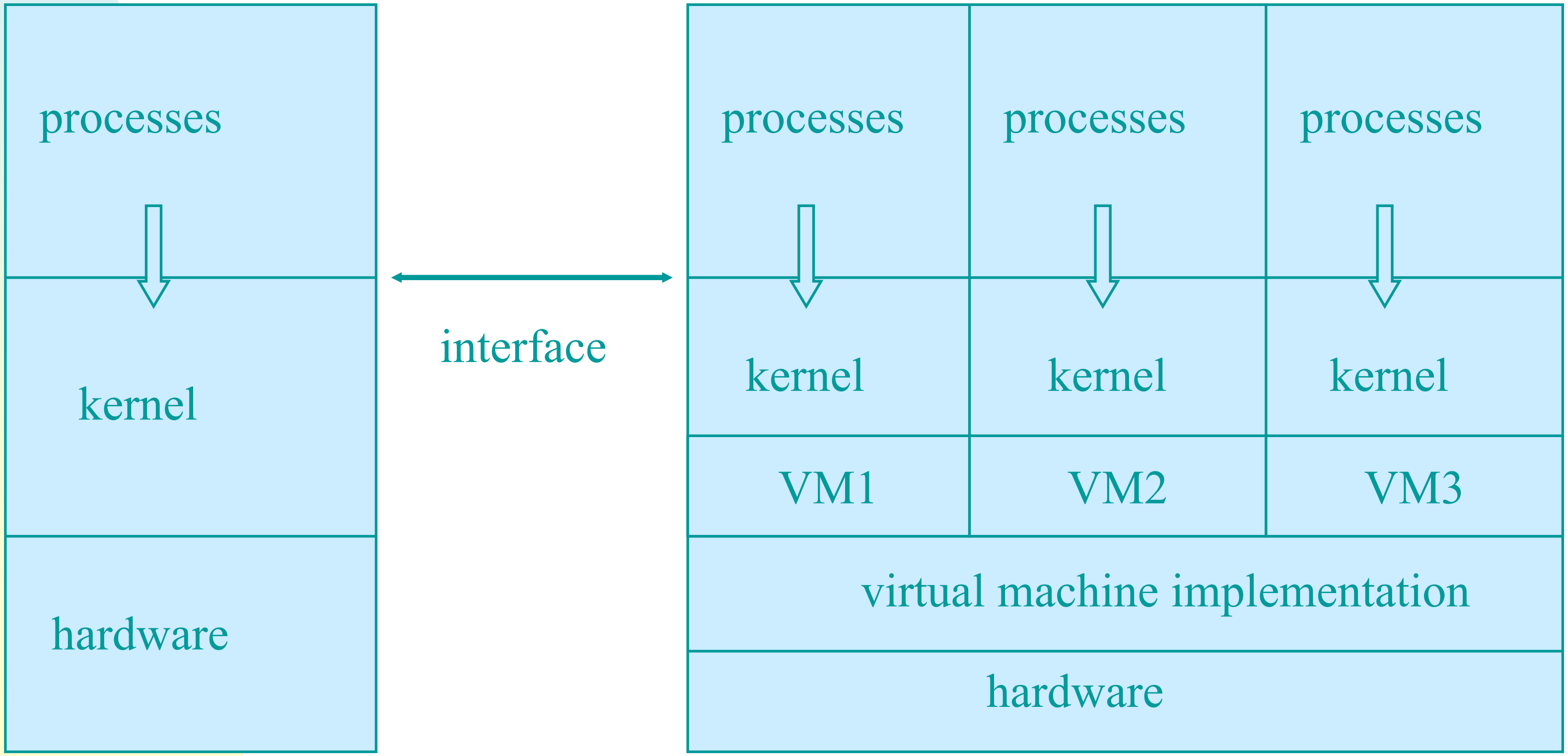
System Structure – Microkernels

- Examples
  - Microkernels: True64UNIX (Mach kernel), MacOS X (Mach kernel), QNX (msg passing, proc scheduling, HW interrupts, low-level networking)
  - Hybrid structures: Windows NT



Virtual Machine

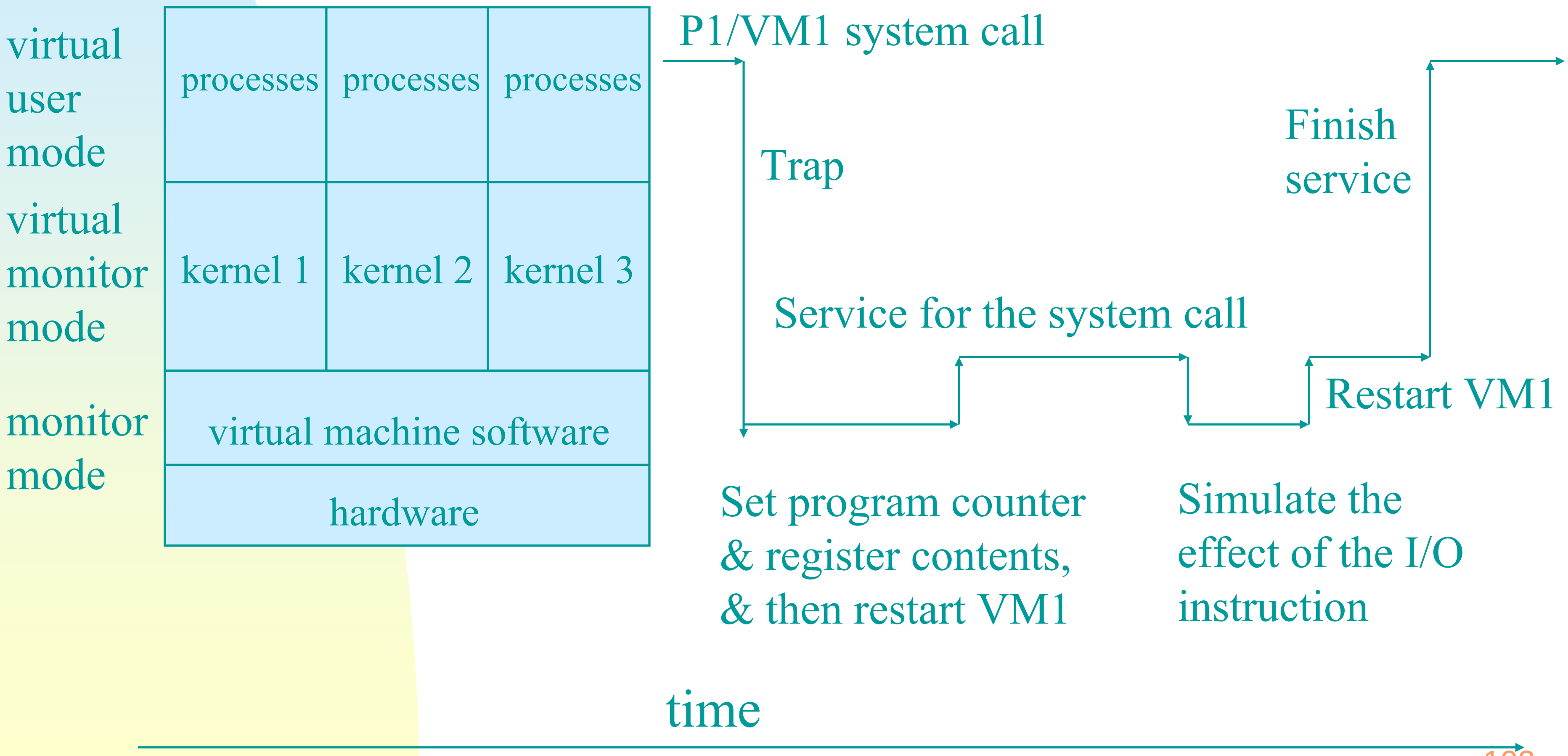
- Virtual Machines: provide an interface that is identical to the underlying bare hardware



Virtual Machine

- Implementation Issues:
  - Emulation of Physical Devices
    - E.g., Disk Systems
      - An IBM minidisk approach
  - User/Monitor Modes
    - (Physical) Monitor Mode
      - Virtual machine software
    - (Physical) User Mode
      - Virtual monitor mode & Virtual user mode

Virtual Machine



Virtual Machine

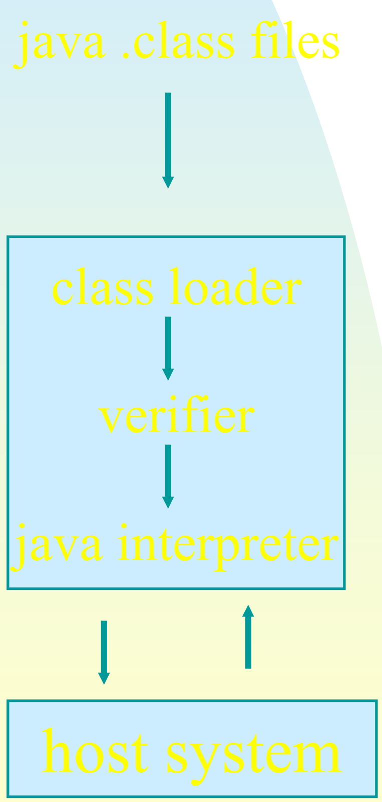
- Disadvantages:
  - Slow!
    - Execute most instructions directly on the hardware
  - No direct sharing of resources
    - Physical devices and communications

* I/O could be slow (interpreted) or fast (spooling)

Virtual Machine

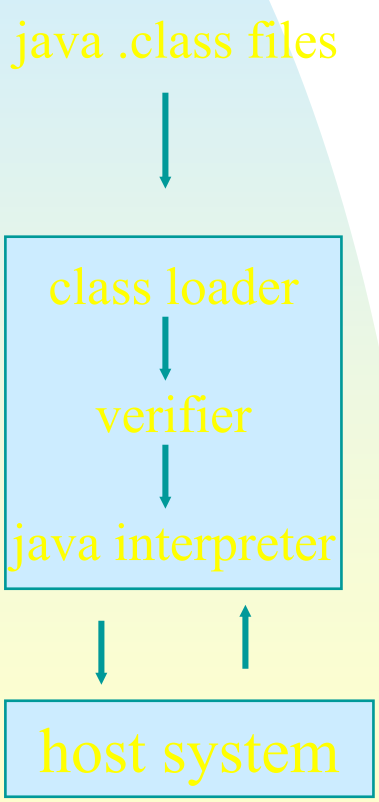
- Advantages:
  - Complete Protection – Complete Isolation！
  - OS Research & Development
    - System Development Time
  - Extensions to Multiple Personalities, such as Mach (software emulation)
    - Emulations of Machines and OS's, e.g., Windows over Linux

Virtual Machine – Java



- Sun Microsystems in late 1995
  - Java Language and API Library
  - Java Virtual Machine (JVM)
    - Class loader (for bytecode .class files)
    - Class verifier
    - Java interpreter
      - An interpreter, a just-in-time (JIT) compiler, hardware

Virtual Machine – Java



- JVM
  - Garbage collection
    - Reclaim unused objects
  - Implementation being specific for different systems
    - Programs are architecture neutral and portable

System Design & Implementation

- Design Goals & Specifications:
  - User Goals, e.g., ease of use
  - System Goals, e.g., reliable
- Rule 1: Separation of Policy & Mechanism
  - Policy：What will be done?
  - Mechanism：How to do things?
  - Example: timer construct and time slice
- Two extreme cases:

Microkernel-based OS ←·········→ Macintosh OS

System Design & Implementation

- OS Implementation in High-Level Languages
  - E.g., UNIX, OS/2, MS NT, etc.
  - Advantages:
    - Being easy to understand & debug
    - Being written fast, more compact, and portable
  - Disadvantages:
    - Less efficient but more storage for code

* Tracing for bottleneck identification, exploring of excellent algorithms, etc.

System Generation

- SYSGEN (System Generation)
  - Ask and probe for information concerning the specific configuration of a hardware system
    - CPU, memory, device, OS options, etc.

No recompilation & completely table-driven	←······→ Linking of modules for selected OS	Recompilation of a modified source code

- Issues
  - Size, Generality, Ease of modification

Contents

Chapter 4 Processes

Processes

- Objective:
  - Process Concept & Definitions
- Process Classification:
  - Operating system processes executing system code
  - User processes executing system code
  - User processes executing user code

Processes

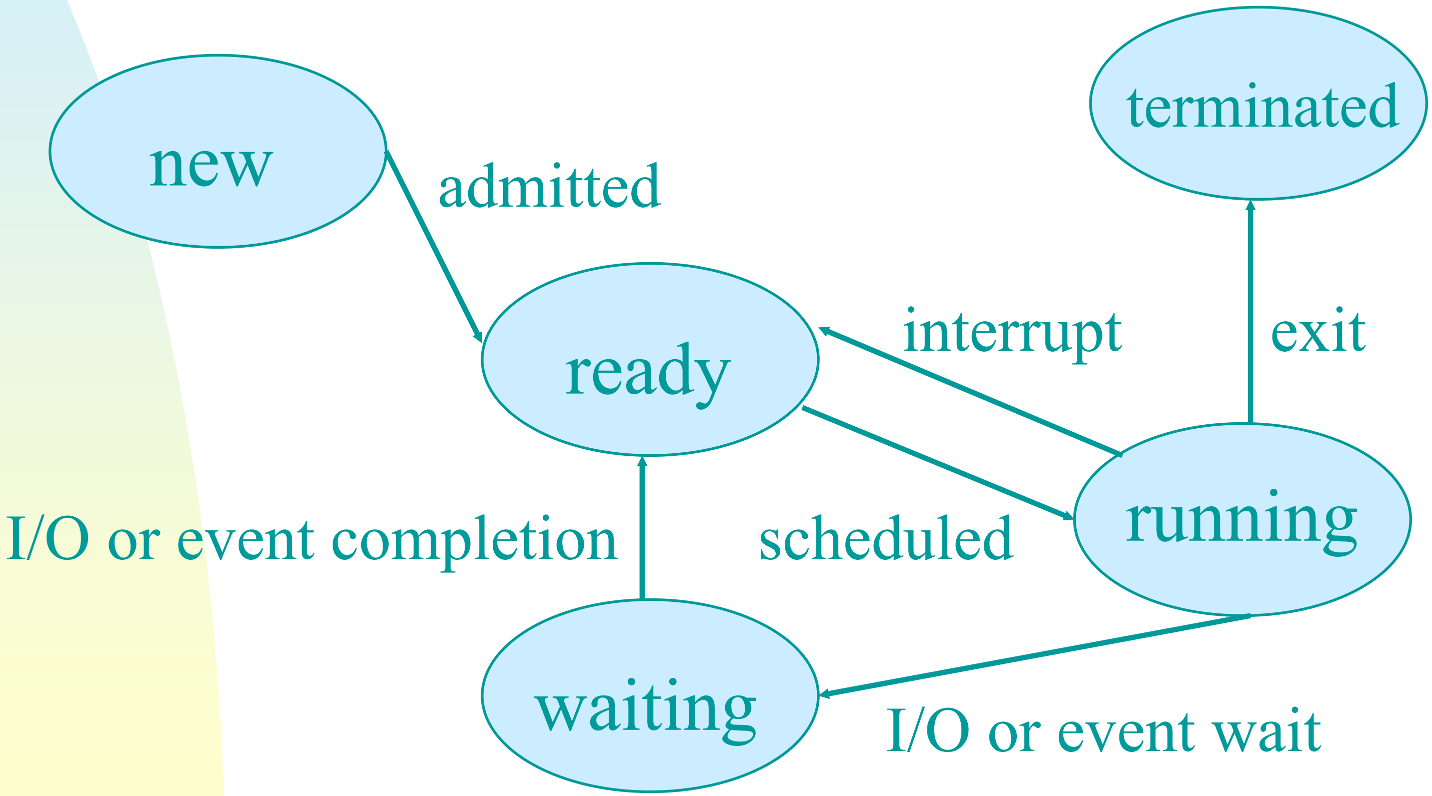
- Example: Special Processes in Unix
  - PID 0 – *Swapper* (i.e., the scheduler)
    - Kernel process
    - No program on disks correspond to this process
  - PID 1 – *init* responsible for bringing up a Unix system after the kernel has been bootstrapped. (/etc/rc* & init or /sbin/rc* & init)
    - User process with superuser privileges
  - PID 2 - pagedaemon responsible for paging
    - Kernel process

Processes

- Process
  - A Basic Unit of Work from the Viewpoint of OS
  - Types:
    - Sequential processes: an activity resulted from the execution of a program by a processor
    - Multi-thread processes
  - An Active Entity
    - Program Code – A Passive Entity
    - Stack and Data Segments
  - The Current Activity
    - PC, Registers , Contents in the Stack and Data Segments

Processes

- Process State

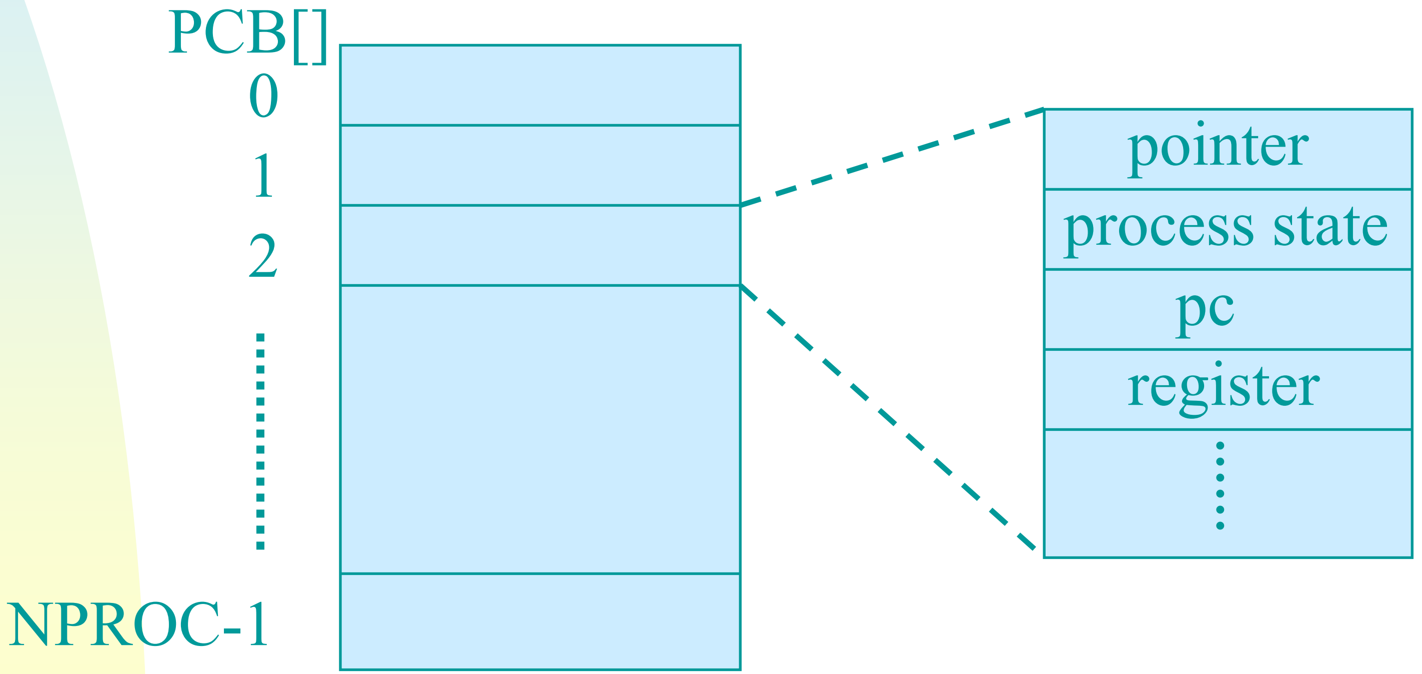


Processes

- Process Control Block (PCB)
  - Process State
  - Program Counter
  - CPU Registers
  - CPU Scheduling Information
  - Memory Management Information
  - Accounting Information
  - I/O Status Information

Processes

- PCB: The repository for any information that may vary from process to process

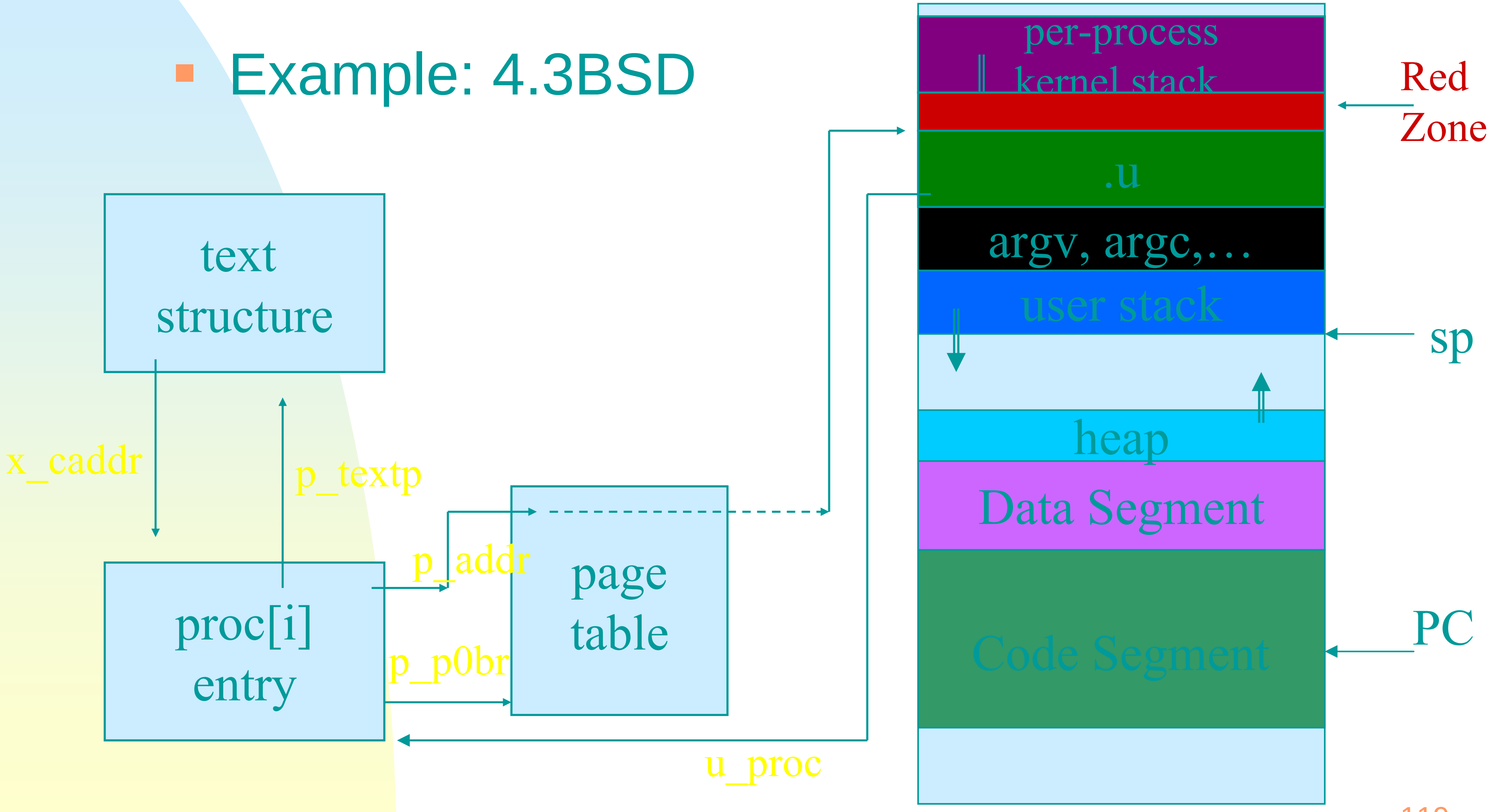


Processes

- Process Control Block (PCB) – An Unix Example
  - proc[i]
    - Everything the system must know when the process is swapped out.
      - pid, priority, state, timer counters, etc.
  - .u
    - Things the system should know when process is running
      - signal disposition, statistics accounting, files[], etc.

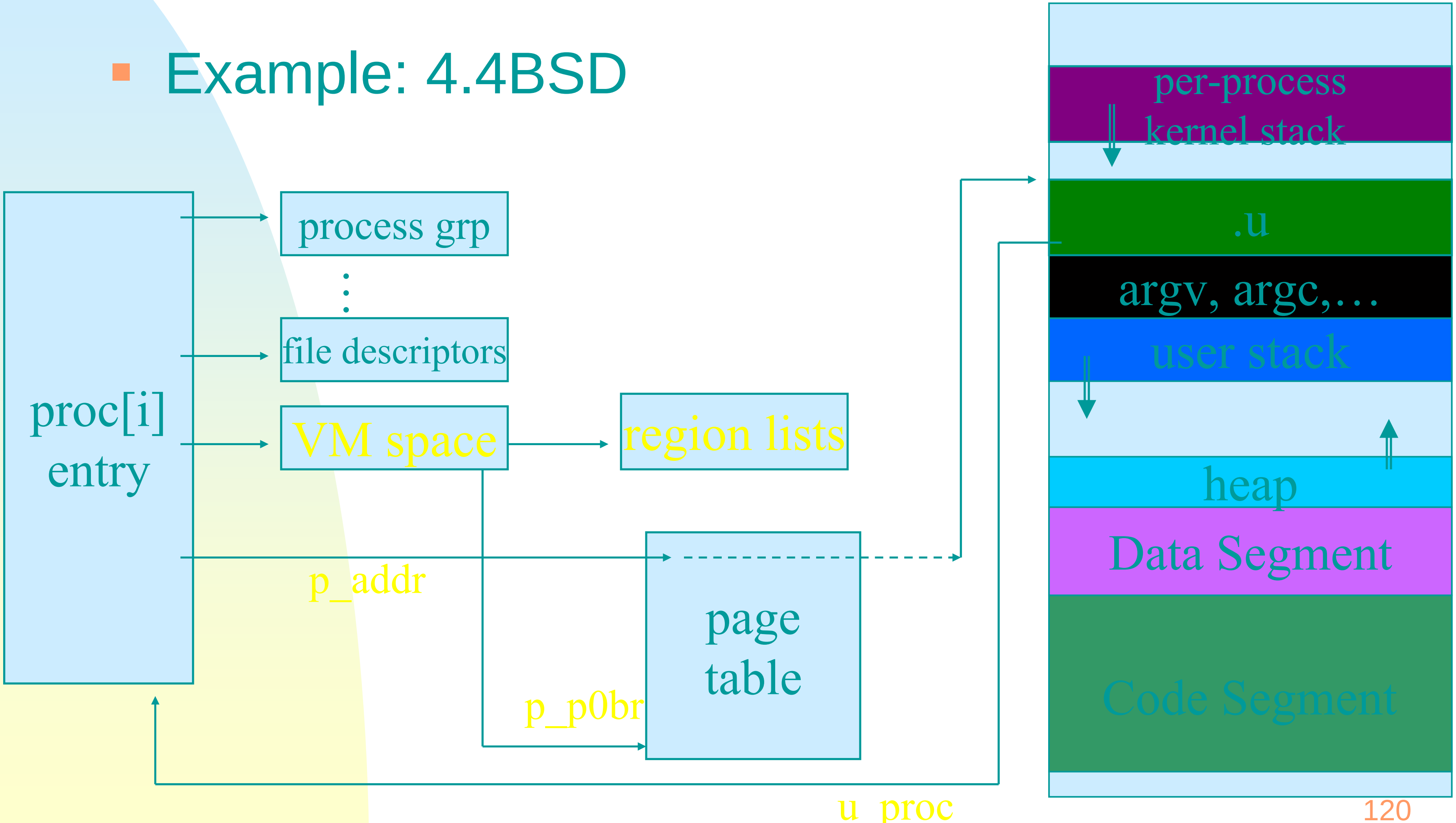
Processes

- Example: 4.3BSD



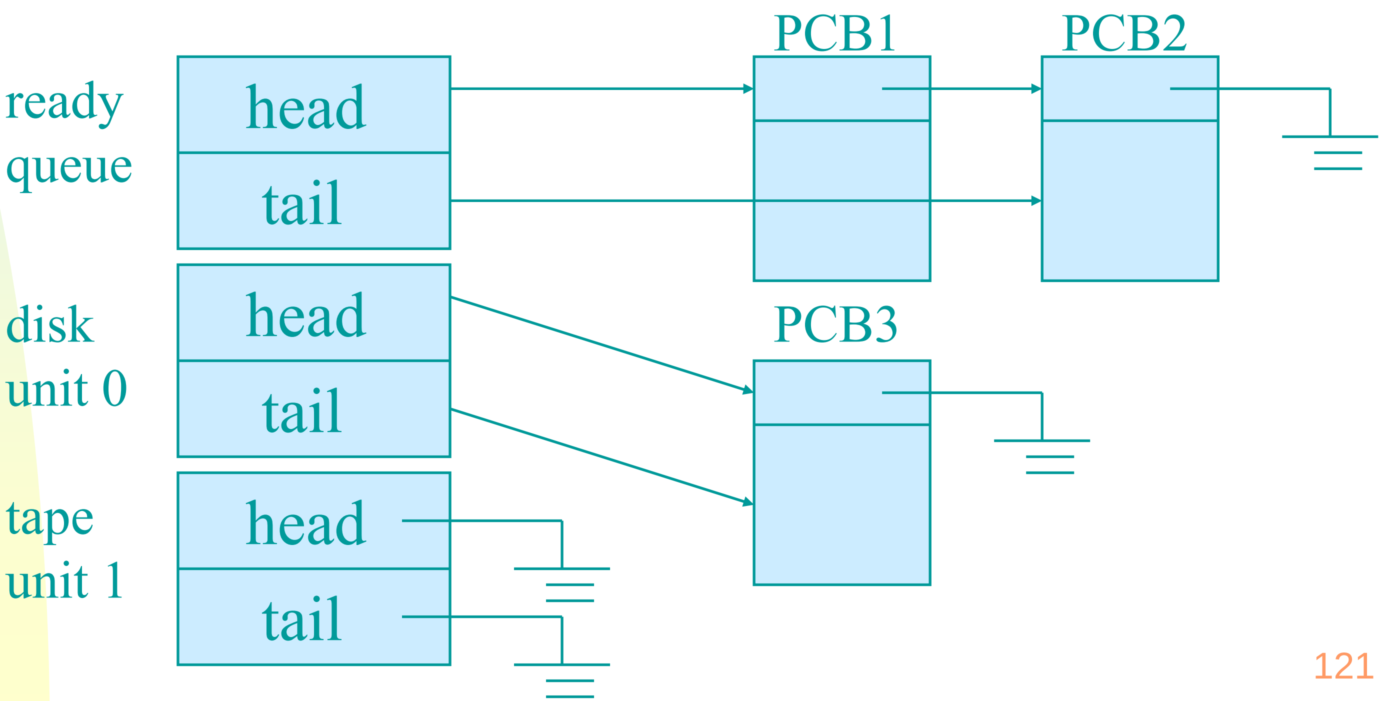
Processes

- Example: 4.4BSD

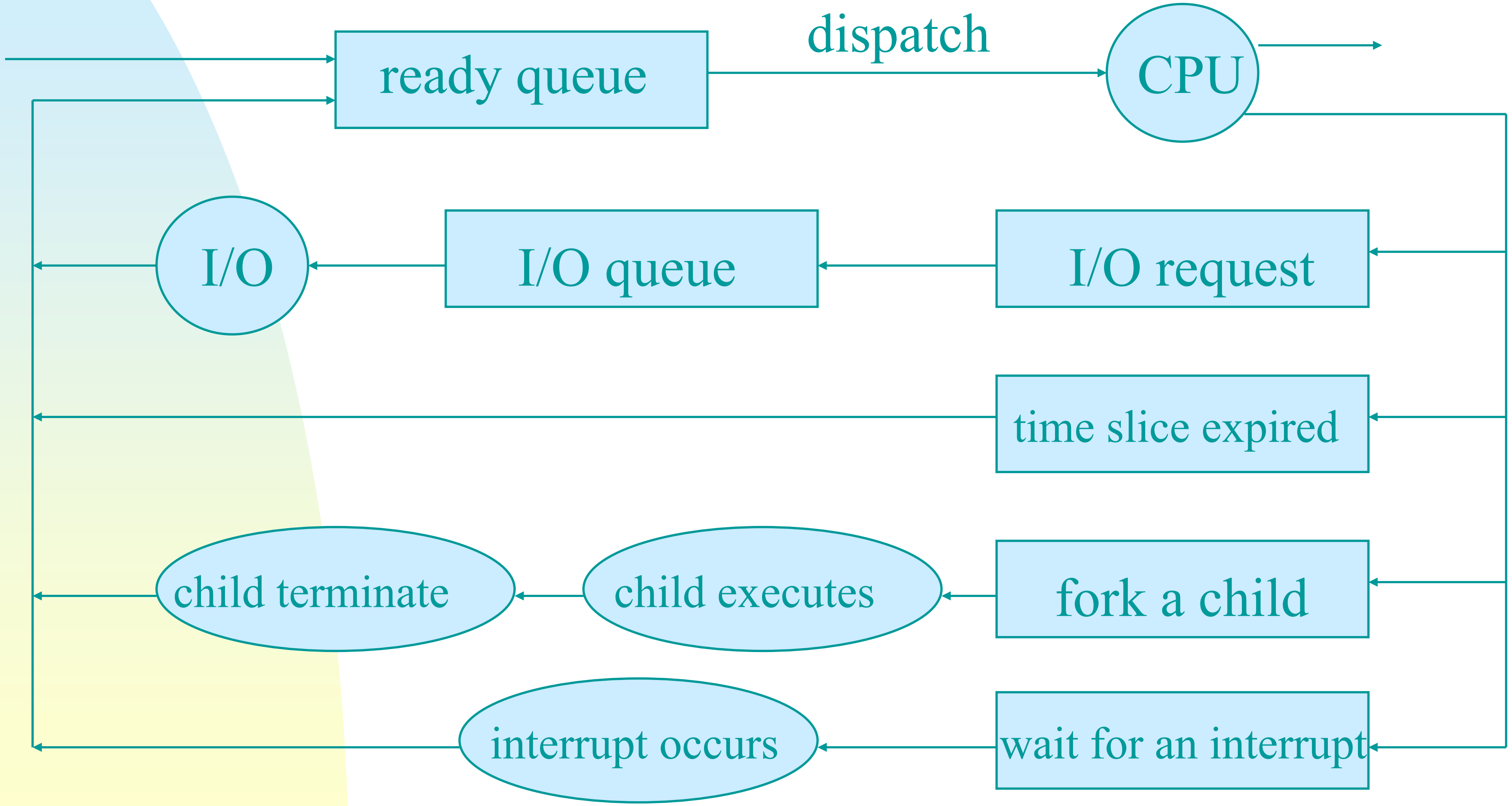


Process Scheduling

- The goal of multiprogramming
  - Maximize CPU/resource utilization!
- The goal of time sharing
  - Allow each user to interact with his/her program!

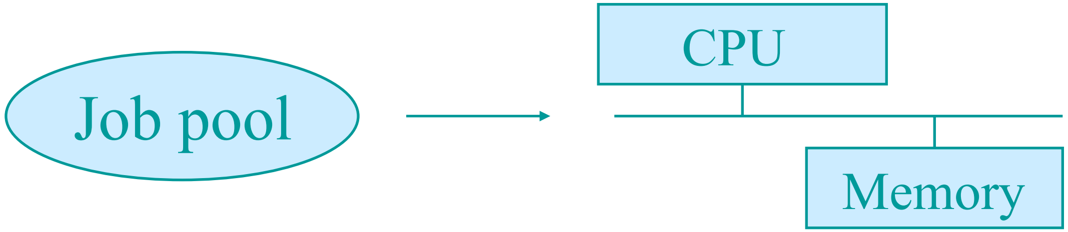


Process Scheduling – A Queueing Diagram



Process Scheduling – Schedulers

- Long-Term (/Job) Scheduler



  - Goal: Select a good mix of I/O-bound and CPU-bound process
  - Remarks：
    1. Control the degree of multiprogramming
    2. Can take more time in selecting processes because of a longer interval between executions
    3. May not exist physically

Process Scheduling – Schedulers

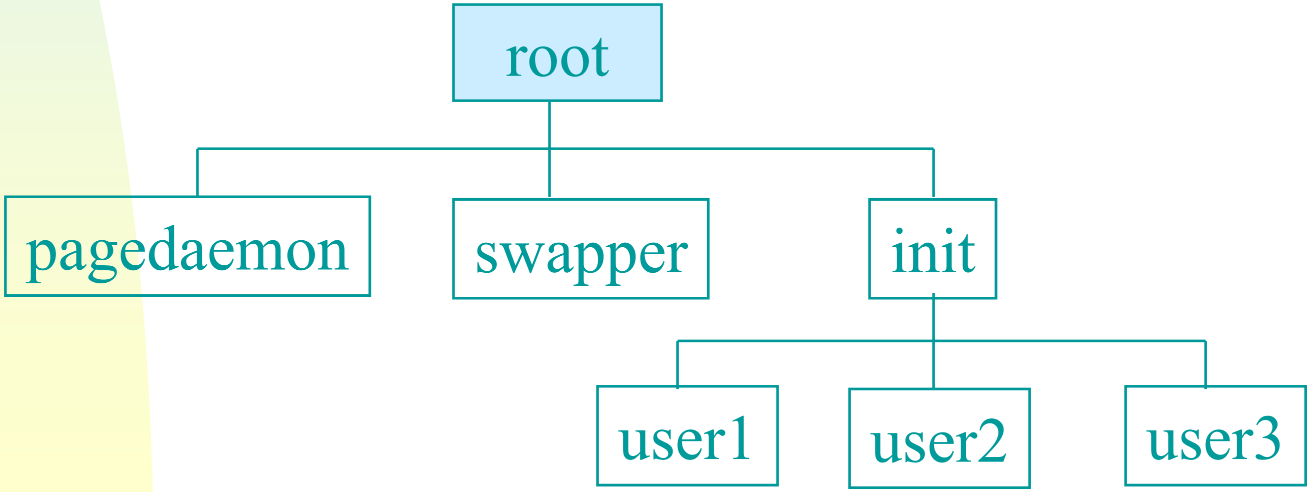
- Short-Term (/CPU) Scheduler
  - Goal：Efficiently allocate the CPU to one of the ready processes according to some criteria.
- Mid-Term Scheduler
  - Swap processes in and out memory to control the degree of multiprogramming

Process Scheduling – Context Switches

- Context Switch ~ Pure Overheads
  - Save the state of the old process and load the state of the newly scheduled process.
    - The context of a process is usually reflected in PCB and others, e.g., .u in Unix.
- Issues：
  - The cost depends on hardware support
    - e.g. processes with multiple register sets or computers with advanced memory management.
  - Threads, i.e., light-weight process (LWP), are introduced to break this bottleneck！

Operations on Processes

- Process Creation & Termination
  - Restrictions on resource usage
  - Passing of Information
  - Concurrent execution



Operations on Processes

- Process Duplication
  - A copy of parent address space + context is made for child, except the returned value from fork()：
    - Child returns with a value 0
    - Parent returns with process id of child
  - No shared data structures between parent and child – Communicate via shared files, pipes, etc.
  - Use execve() to load a new program
  - fork() vs vfork() (Unix)

Operations on Processes

- Example:

```
…
 if ( pid = fork() ) == 0) {
    /* child process */
     execlp("/bin/ls", "ls", NULL);
  } else if (pid < 0) {
    fprintf(stderr, "Fork Failed");
    exit(-1);
  } else {
    /* parent process */
    wait(NULL);
   }
```

Operations on Processes

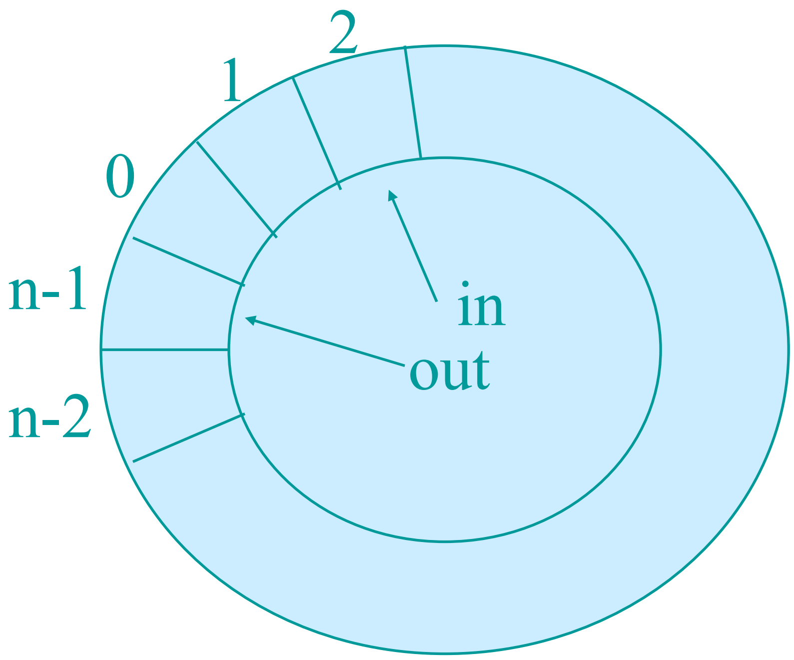
- Termination of Child Processes
  - Reasons:
    - Resource usages, needs, etc.
  - Kill, exit, wait, abort, signal, etc.
- Cascading Termination

Cooperating Processes

- Cooperating processes can affect or be affected by the other processes
  - Independent Processes
- Reasons:
  - Information Sharing, e.g., files
  - Computation Speedup, e.g., parallelism.
  - Modularity, e.g., functionality dividing
  - Convenience, e.g., multiple work

Cooperating Processes

- A Consumer-Producer Example:
  - Bounded buffer or unbounded buffer
    - Supported by inter-process communication (IPC) or by hand coding



buffer[0…n-1]

Initially,

in=out=0；

Cooperating Processes

Producer:

```
while (1) {
    /* produce an item nextp */
      while (((in+1) % BUFFER_SIZE) == out)
              ;  /* do nothing */
      buffer[ in ] = nextp;
      in = (in+1) % BUFFER_SIZE;
}
```

Cooperating Processes

Consumer:

```
while (1) {
    while (in == out)
        ; /* do nothing */
    nextc = buffer[ out ];
    out = (out+1) % BUFFER_SIZE ;
    /* consume the item in nextc */
}
```

Interprocess Communication

- Why Inter-Process Communication (IPC)?
  - Exchanging of Data and Control Information!
- Why Process Synchronization?
  - Protect critical sections!
  - Ensure the order of executions!

Interprocess Communication

- IPC
  - Shared Memory
  - Message Passing
- Logical Implementation of Message Passing
  - Fixed/variable msg size, symmetric/asymmetric communication, direct/indirect communication, automatic/explicit buffering, send by copy or reference, etc.

Interprocess Communication

- Classification of Communication by Naming
  - Processes must have a way to refer to each other!
  - Types
    - Direct Communication
    - Indirect Communication

Interprocess Communication – Direct Communication

- Process must explicitly name the recipient or sender of a communication
  - Send(P, msg), Receive(Q, msg)
- Properties of a Link:

  a. Communication links are established automatically.

  b. Two processes per a link

  c. One link per pair of processes

  d. Bidirectional or unidirectional

Interprocess Communication – Direct Communication

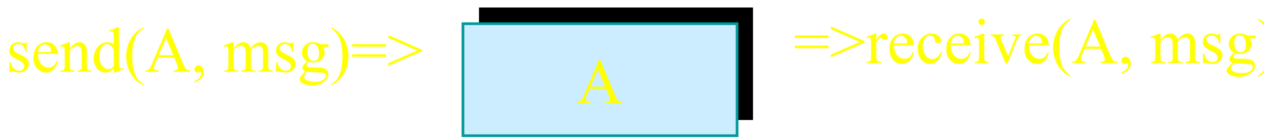
- Issue in Addressing:
  - Symmetric or asymmetric addressing

    Send(P, msg), Receive(id, msg)

- Difficulty:
  - Process naming vs modularity

Interprocess Communication – Indirect Communication

- Two processes can communicate only if the process share a mailbox (or ports)



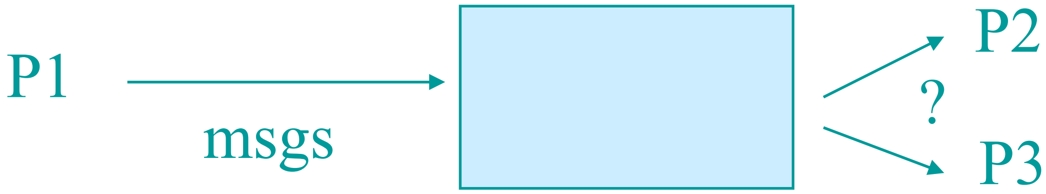
- Properties:

1. A link is established between a pair of processes only if they share a mailbox.
2. *n* processes per link for *n* >= 1.
3. *n* links can exist for a pair of processes for *n* >=1.
4. Bidirectional or unidirectional

Interprocess Communication – Indirect Communication

- Issues:

a. Who is the recipient of a message?



b. Owners vs Users

- Process → owner as the sole recipient?
- OS → Let the creator be the owner?
  Privileges can be passed?
  Garbage collection is needed?

Interprocess Communication – Synchronization

- Blocking or Nonblocking (Synchronous versus Asynchronous)
  - Blocking send
  - Nonblocking send
  - Blocking receive
  - Nonblocking receive
- Rendezvous – blocking send & receive

Interprocess Communication – Buffering

- The Capacity of a Link = the # of messages could be held in the link.
  - Zero capacity(no buffering)
    - Msg transfer must be synchronized – rendezvous!
  - Bounded capacity
    - Sender can continue execution without waiting till the link is full
  - Unbounded capacity
    - Sender is never delayed!
- The last two items are for asynchronous communication and may need acknowledgement

Interprocess Communication – Buffering

- Special cases:
  a. Msgs may be lost if the receiver can not catch up with msg sending
     → synchronization
  b. Senders are blocked until the receivers have received msgs and replied by reply msgs
     → A Remote Procedure Call (RPC) framework

Interprocess Communication – Exception Conditions

- Process termination
  a. Sender Termination→ Notify or terminate the receiver!
  b. Receiver Termination
     a. No capacity → sender is blocked.
     b. Buffering→ messages are accumulated.

Interprocess Communication – Exception Conditions

- Ways to Recover Lost Messages (due to hardware or network failure):
  - OS detects & resends messages.
  - Sender detects & resends messages.
  - OS detects & notify the sender to handle it.
- Issues:
  a. Detecting methods, such as timeout!
  b. Distinguish multiple copies if retransmitting is possible
- Scrambled Messages:
  - Usually OS adds checksums, such as CRC, inside messages & resend them as necessary!

Example - Mach

- Mach – A message-based OS from the Carnegie Mellon University
  - When a task is created, two special mailboxes, called ports, are also created.
    - The *Kernel* mailbox is used by the kernel to communication with the tasks
    - The *Notify* mailbox is used by the kernel sends notification of event occurrences.

Example - Mach

- Three system calls for message transfer:
  - msg_send:
    - Options when mailbox is full:

    a. Wait indefinitely
    b. Return immediately
    c. Wait at most for *n* ms
    d. Temporarily cache a message.
       a. A cached message per sending thread for a mailbox

* One task can either own or receive from a mailbox.

Example - Mach

- msg_receive
  - To receive from a mailbox or a set of mailboxes. Only one task can own & have a receiving privilege of it

    * options when mailbox is empty:

    a. Wait indefinitely
    b. Return immediately
    c. Wait at most for *n* ms
- msg_rpc
  - Remote Procedure Calls

Example - Mach

- port_allocate
  - create a mailbox (owner)
  - port_status ~ .e.g, # of msgs in a link
- All messages have the same priority and are served in a FIFO fashion.
- Message Size
  - A fixed-length head + a variable-length data + two mailbox names
- Message copying: message copying → remapping of addressing space
- System calls are carried out by messages.

Example – Windows 2000

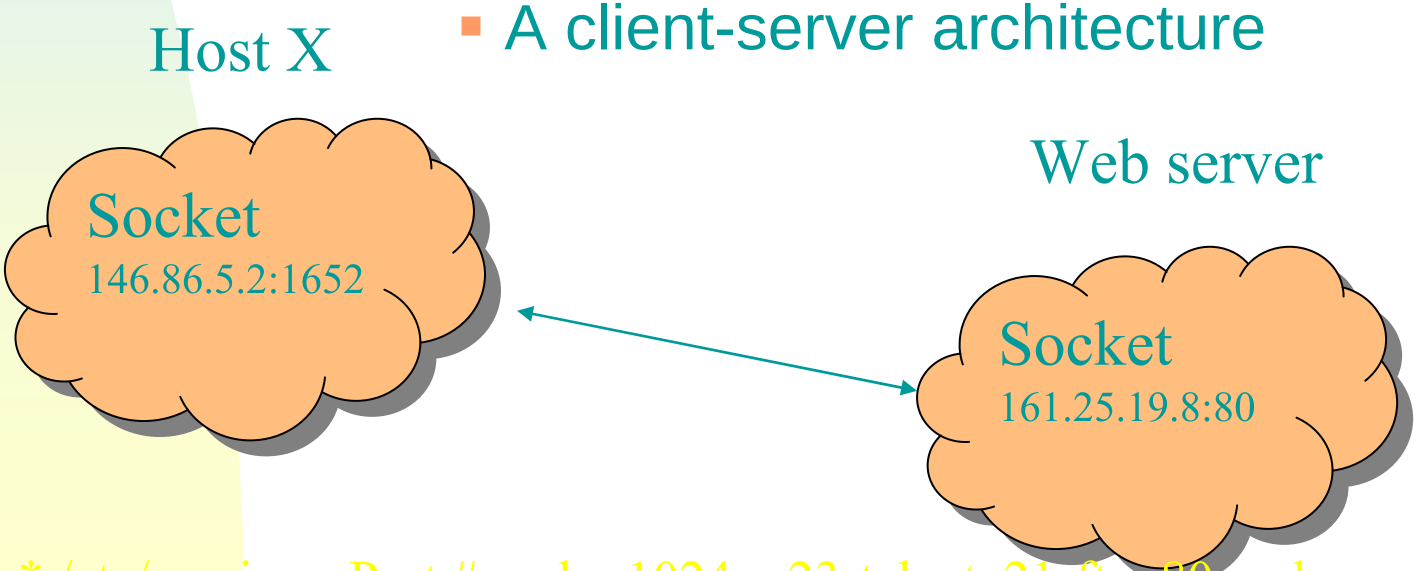
- Local Procedure Call (LPC) – Message Passing on the Same Processor
  1. The client opens a handle to a subsystem's *connection port* object.
  2. The client sends a connection request.
  3. The server creates two private *communication ports*, and returns the handle to one of them to the client.
  4. The client and server use the corresponding port handle to send messages or callbacks and to listen for replies.

Example – Windows 2000

- Three Types of Message Passing Techniques
  - Small messages
    - Message copying
  - Large messages – section object
    - To avoid memory copy
    - Sending and receiving of the pointer and size information of the object
  - A callback mechanism
    - When a response could not be made immediately.

Communication in Client-Server Systems

- Socket
  - An endpoint for communication identified by an IP address concatenated with a port number
    - A client-server architecture



* /etc/services: Port # under 1024 ~ 23-telnet, 21-ftp, 80-web server, etc.

Communication in Client-Server Systems

- Three types of sockets in Java
  - Connection-oriented (TCP) – Socket class
  - Connectionless (UDP) – DatagramSocket class
  - MulticastSocket class – DatagramSocket subclass

Server

```
sock = new ServerSocket(5155);
…
client = sock.accept();
pout = new PrintWriter(client.getOutputStream(),
        true);
…
Pout.println(new java.util.Date().toString());
pout.close();
client.close();
```

Client

```
sock = new Socket(“127.0.0.1”,5155);
…
in = sock.getInputStream();
bin = new BufferReader(new
        InputStreamReader(in));
…
sock.close();
```

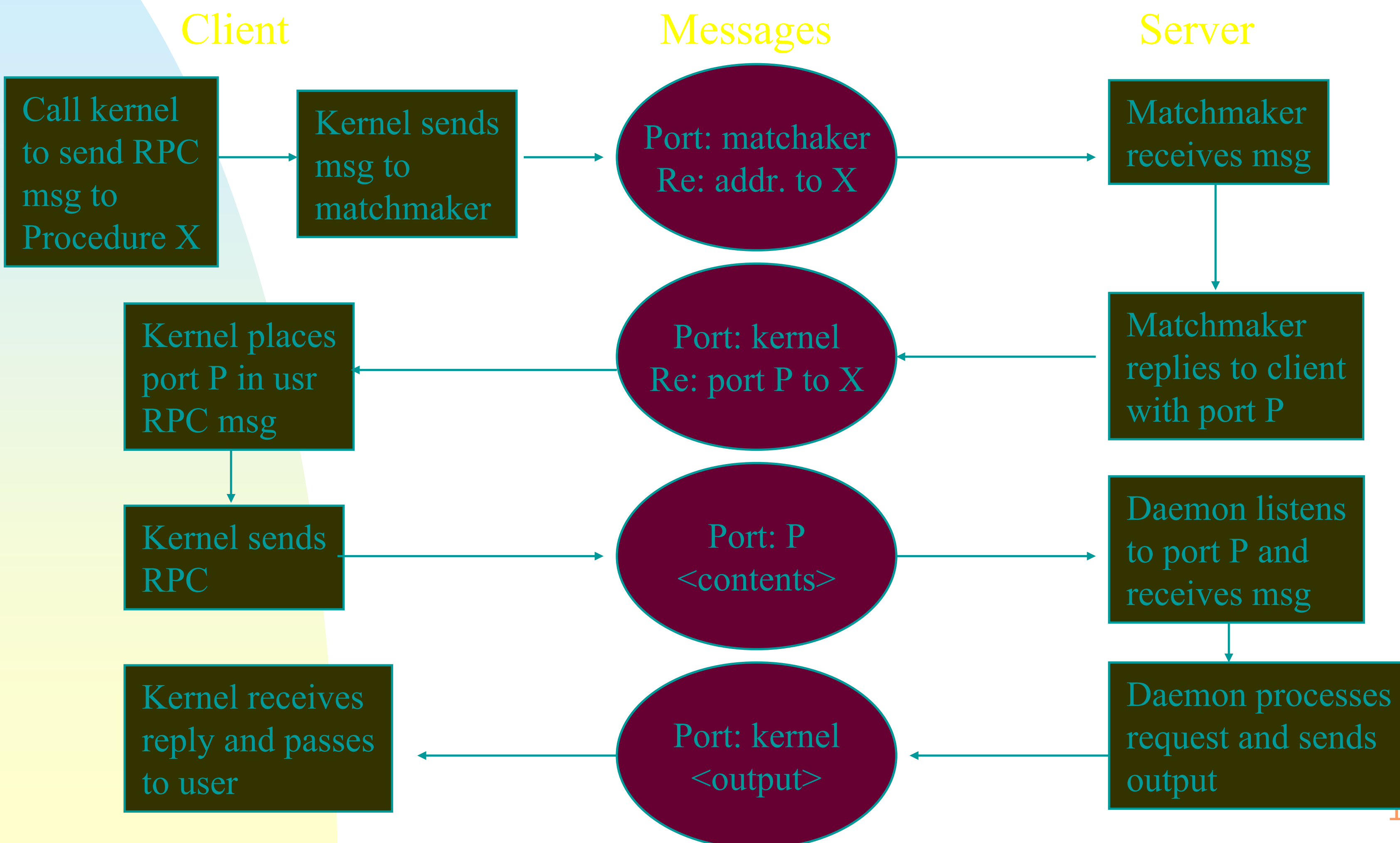
Communication in Client-Server Systems

- Remote Procedure Call (RPC)
  - A way to abstract the procedure-call mechanism for use between systems with network connection.
  - Needs:
    - Ports to listen from the RPC daemon site and to return results, identifiers of functions to call, parameters to pack, etc.
    - Stubs at the client site
      - One for each RPC
      - Locate the proper port and marshall parameters.

Communication in Client-Server Systems

- Needs (continued)
  - Stubs at the server site
    - Receive the message
    - Invoke the procedure and return the results.
- Issues for RPC
  - Data representation
    - External Data Representation (XDR)
      - Parameter marshalling
  - Semantics of a call
    - History of all messages processed
  - Binding of the client and server port
    - Matchmaker – a rendezvous mechanism

Communication in Client-Server Systems



Communication in Client-Server Systems

- An Example for RPC
  - A Distributed File System (DFS)
    - A set of RPC daemons and clients
    - DFS port on a server on which a file operation is to take place:
      - Disk operations: read, write, delete, status, etc – corresponding to usual system calls

Communication in Client-Server Systems

- Remote Method Invocation (RMI)
  - Allow a thread to invoke a method on a remote object.
    - boolean val = Server.someMethod(A,B)
- Implementation
  - Stub – a proxy for the remote object
    - Parcel – a method name and its marshalled parameters, etc.
  - Skeleton – for the unmarshalling of parameters and invocation of the method and the sending of a parcel back

Communication in Client-Server Systems

- Parameter Passing
  - Local (or Nonremote) Objects
    - Pass-by-copy – an object serialization
  - Remote Objects – Reside on a different Java virtual machine (JVM)
    - Pass-by-reference
  - Implementation of the interface – *java.io.Serializable*

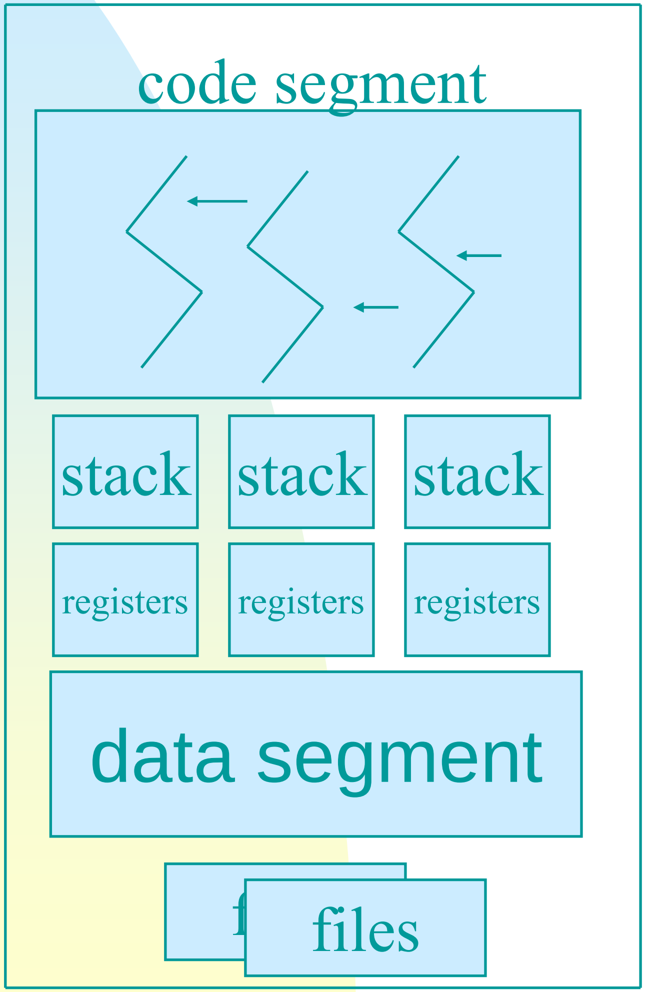
Contents

Chapter 5 Threads

Threads

- Objectives:
  - Concepts and issues associated with multithreaded computer systems.
- Thread – Lightweight process(LWP)
  - a basic unit of CPU utilization
    - A thread ID, program counter, a register set, and a stack space
  - Process – heavyweight process
    - A single thread of control

Threads

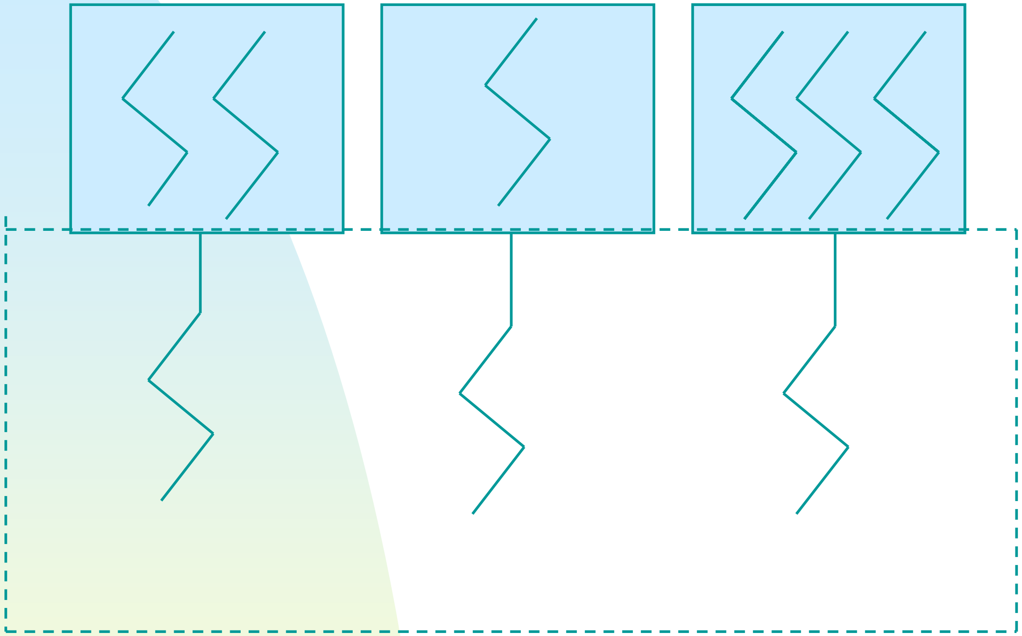


- Motivation
  - A web browser
    - Data retrieval
    - Text/image displaying
  - A word processor
    - Displaying
    - Keystroke reading
    - Spelling and grammar checking
  - A web server
    - Clients' services
    - Request listening

Threads

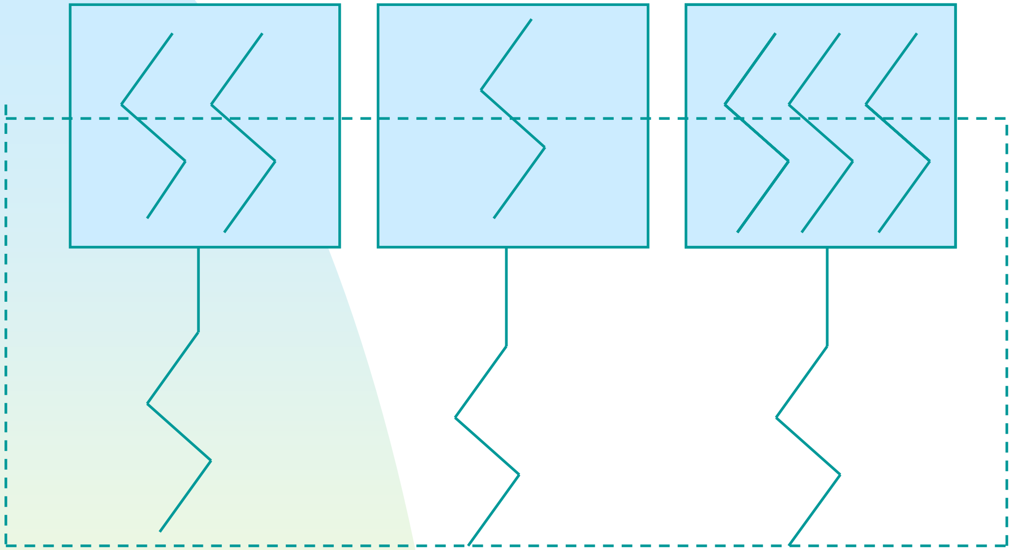
- Benefits
  - Responsiveness
  - Resource Sharing
  - Economy
    - Creation and context switching
      - 30 times slower in process creation in Solaris 2
      - 5 times slower in process context switching in Solaris 2
  - Utilization of Multiprocessor Architectures

User-Level Threads



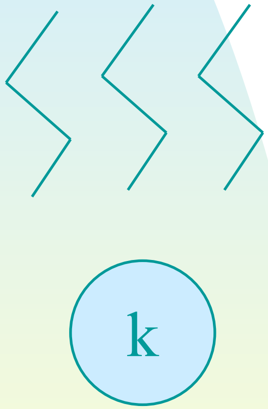
- User-level threads are implemented by a thread library at the user level.
- Examples:
  - POSIX Pthreads, Mach C-threads, Solaris 2 UI-threads
- Advantages
  - Context switching among them is extremely fast
- Disadvantages
  - Blocking of a thread in executing a system call can block the entire process.

Kernel-Level Threads



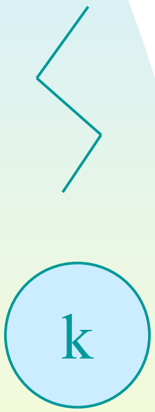
- Kernel-level threads are provided a set of system calls similar to those of processes
- Examples
  - Windows 2000, Solaris 2, True64UNIX
- Advantage
  - Blocking of a thread will not block its entire task.
- Disadvantage
  - Context switching cost is a little bit higher because the kernel must do the switching.

Multithreading Models



- Many-to-One Model
  - Many user-level threads to one kernel thread
  - Advantage:
    - Efficiency
  - Disadvantage:
    - One blocking system call blocks all.
    - No parallelism for multiple processors
  - Example: Green threads for Solaris 2

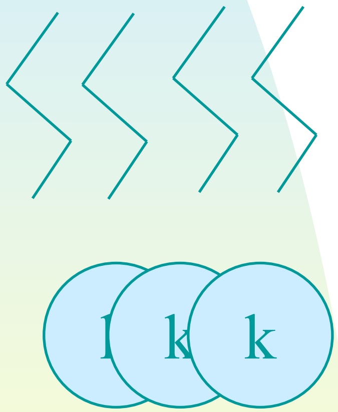
Multithreading Models



- One-to-One Model
  - One user-level thread to one kernel thread
  - Advantage: One system call blocks one thread.
  - Disadvantage: Overheads in creating a kernel thread.
  - Example: Windows NT, Windows 2000, OS/2

Multithreading Models

- Many-to-Many Model
  - Many user-level threads to many kernel threads
  - Advantage:
    - A combination of parallelism and efficiency
  - Example: Solaris 2, IRIX, HP-UX,Tru64 UNIX



Threading Issues

- Fork and Exec System Calls
  - Fork: Duplicate all threads or create a duplicate with one thread?
  - Exec: Replace the entire process, including all threads and LWPs.
  - Fork → exec?

Threading Issues

- Thread Cancellation
  - Target thread
  - Two scenarios:
    - Asynchronous cancellation
    - Deferred cancellation
      - *Cancellation points* in Pthread.
  - Difficulty
    - Resources have been allocated to a cancelled thread.
    - A thread is cancelled while it is updating data.

Threading Issues

- Signal Handling
  - Signal
    - Synchronous – delivered to the same process that performed the operation causing the signal,
      - e.g., illegal memory access or division by zero
    - Asynchronous
      - e.g., ^C or timer expiration
  - Default or user-defined signal handler
  - Signal masking

Threading Issues

- Delivery of a Signal
  - To the thread to which the signal applies
    - e.g., division-by-zero
  - To every thread in the process
    - e.g., ^C
  - To certain threads in the process
  - Assign a specific thread to receive all threads for the process
    - Solaris 2
- Asynchronous Procedure Calls (APCs)
  - To a particular thread rather than a process

Threading Issues

- Thread Pools
  - Motivations
    - Dynamic creation of threads
    - Limit on the number of active threads
  - Awake and pass a request to a thread in the pool
  - Benefits
    - Faster for service delivery and limit on the # of threads
  - Dynamic or static thread pools
- Thread-specific data – Win32 & Pthreads

Pthreads

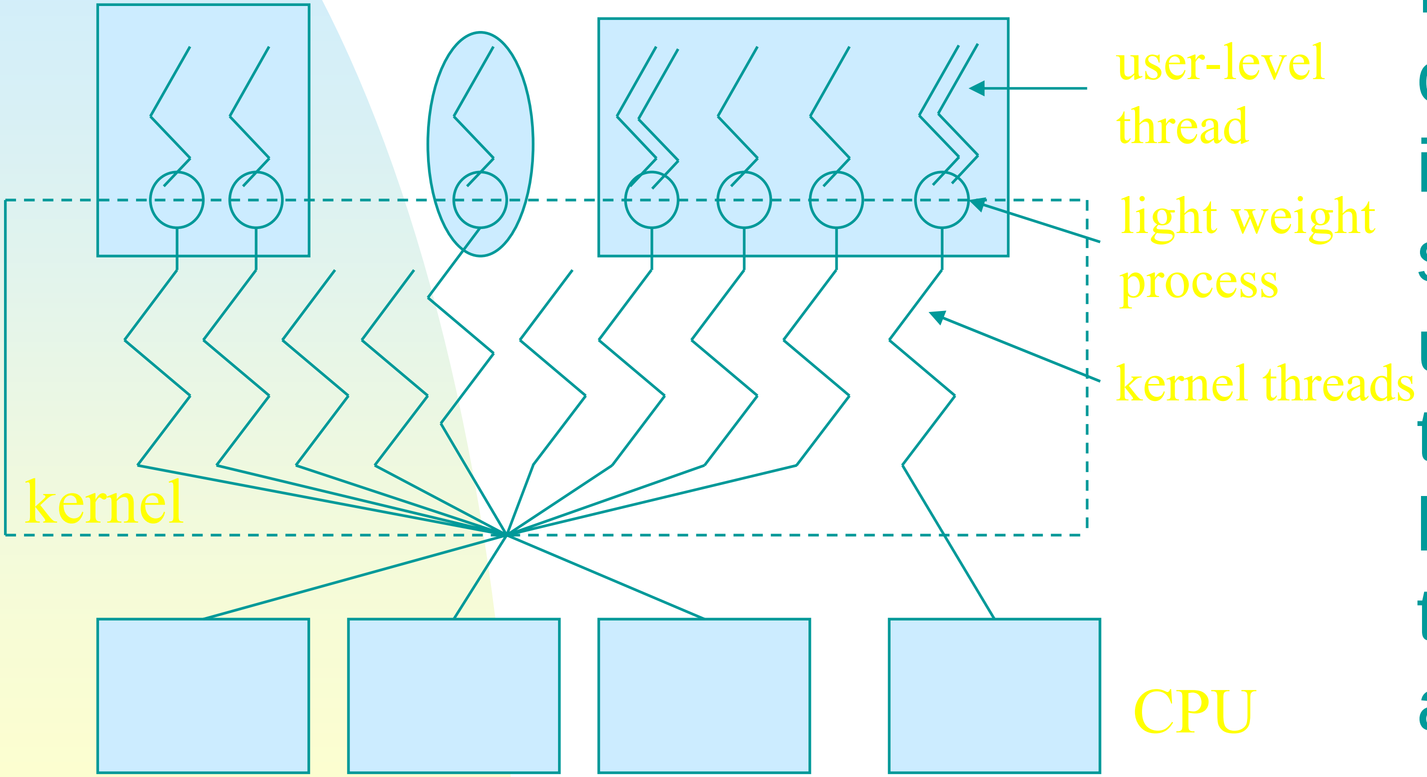
- Pthreads (IEEE 1003.1c)
  - API Specification for Thread Creation and Synchronization
  - UNIX-Based Systems, Such As Solaris 2.
- User-Level Library
- Header File: <pthread.h>
- pthread_attr_init(), pthread_create(), pthread_exit(), pthread_join(), etc.

Pthreads

```
#include <pthread.h>
main(int argc, char *argv[]) {
    …
    pthread_attr_init(&attr);
    pthread_create(&tid, &attr, runner, argv[1]);
    pthread_join(tid, NULL);
… }

void *runner(void *param) {
  int i, upper = atoi(param), sum = 0;
    if (upper > 0)
          for(i=1;i<=upper,i++)
                    sum+=i;
    pthread_exit(0);
}
```

Solaris 2



- Implementation of Pthread API in addition to supporting user-level threads with a library for thread creation and management.

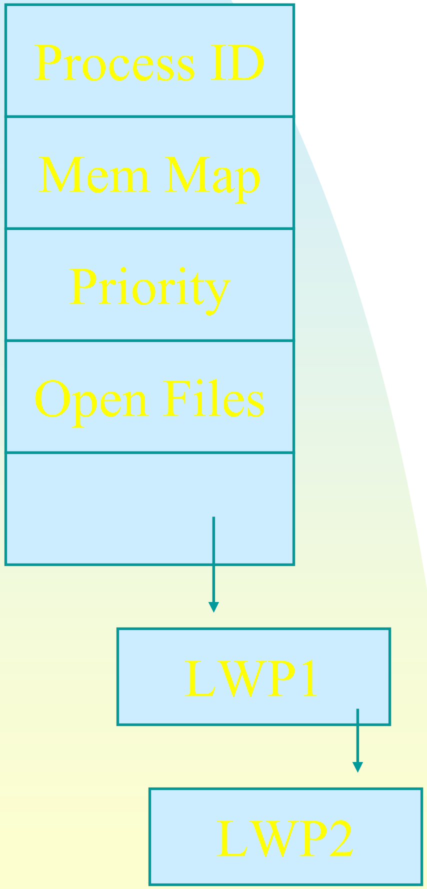
Solaris 2

- Many-to-Many Model
  - Each process has at least one LWP
    - Each LWP has a kernel-level thread
  - User-level threads must be connected to LWPs to accomplish work.
    - A bound user-level thread
    - An unbound thread
- Some kernel threads running on the kernel's behalf have no associated LWPs – system threads

Solaris 2

- Processor Allocation:
  - Multiprogramming or Pinned
- Switches of user-level threads among LWPs do not need kernel intervention.
- If the kernel thread is blocked, so does the LWP and its user-level threads.
  - Dynamic adjustment of the number of LWPs

Solaris 2



Solaris 2 Process

- Data Structures
  - A User-Level Thread
    - A Thread ID, a register set (including PC, SP), stack, and priority – in user space
  - A LWP
    - A Register set for the running user-level thread – in kernel space
  - A Kernel thread
    - A copy of the kernel registers, a pointer to its LWP, priority, scheduling information

Windows 2000

- Win32 API
  - One-to-One Model
  - Fiber Library for the M:M Model
- A Thread Contains
  - A Thread ID
  - Context: A Register Set, A User Stack, A Kernel Stack, and A Private Storage Space

Windows 2000

Kernel Space

User Space

- Data Structures
  - ETHREAD (executive thread block)
    - A ptr to the process,a ptr to KTHREAD, the address of the starting routine
  - KTHREAD (kernel thread block)
    - Scheduling and synchronization information, a kernel stack, a ptr to TEB
  - TEB (thread environment block)
    - A user stack, an array for thread-specific data.

Linux

- Threads introduced in Version 2.2
  - clone() versus fork()
    - Term task for process& thread
    - Several per-process data structures, e.g., pointers to the same data structures for open files, signal handling, virtual memory, etc.
    - Flag setting in clone() invocation.
- Pthread implementations

Java

- Thread Support at the Language Level
  - Mapping of Java Threads to Kernel Threads on the Underlying OS?
    - Windows 2000: 1:1 Model
- Thread Creation
  - Create a new class derived from the Thread class
  - Run its start method
    - Allocate memory and initialize a new thread in the JVM
    - start() calls the run method, making the thread eligible to be run by the JVM.

Java

```
class Summation extends Thread
  { public Summation(int n) {
          upper = n; }
     public void run() {
          int sum = 0;
          … }
    …}
public class ThreadTester
{ …
    Summation thrd = new
    Summation(Integer.ParseInt(args[0]));
    thrd.start();
…}
```

Contents

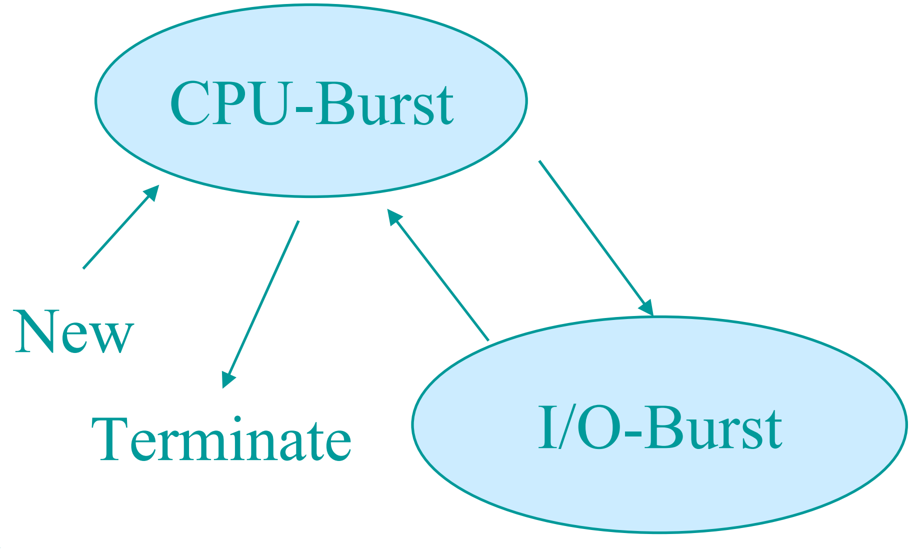
Chapter 6 CPU Scheduling

CPU Scheduling

- Objective:
  - Basic Scheduling Concepts
  - CPU Scheduling Algorithms
- Why Multiprogramming?
  - Maximize CPU/Resources Utilization (Based on Some Criteria)

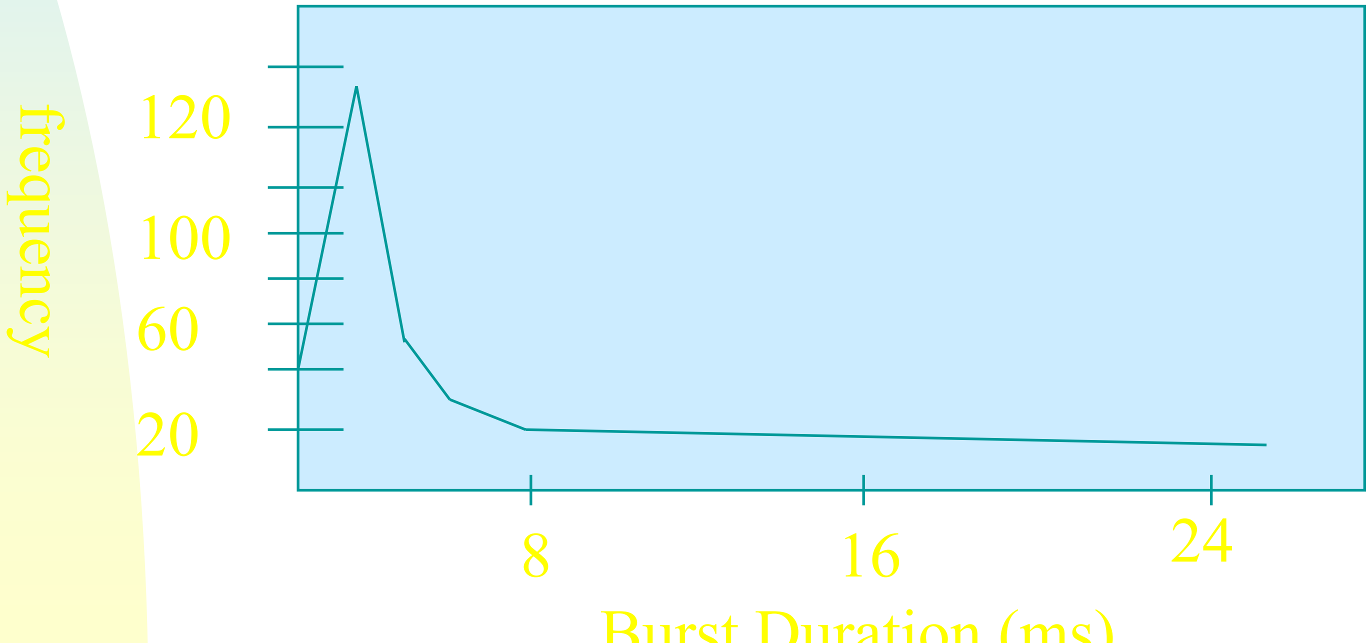
CPU Scheduling

- Process Execution
  - CPU-bound programs tend to have a few very long CPU bursts.
  - IO-bound programs tend to have many very short CPU bursts.



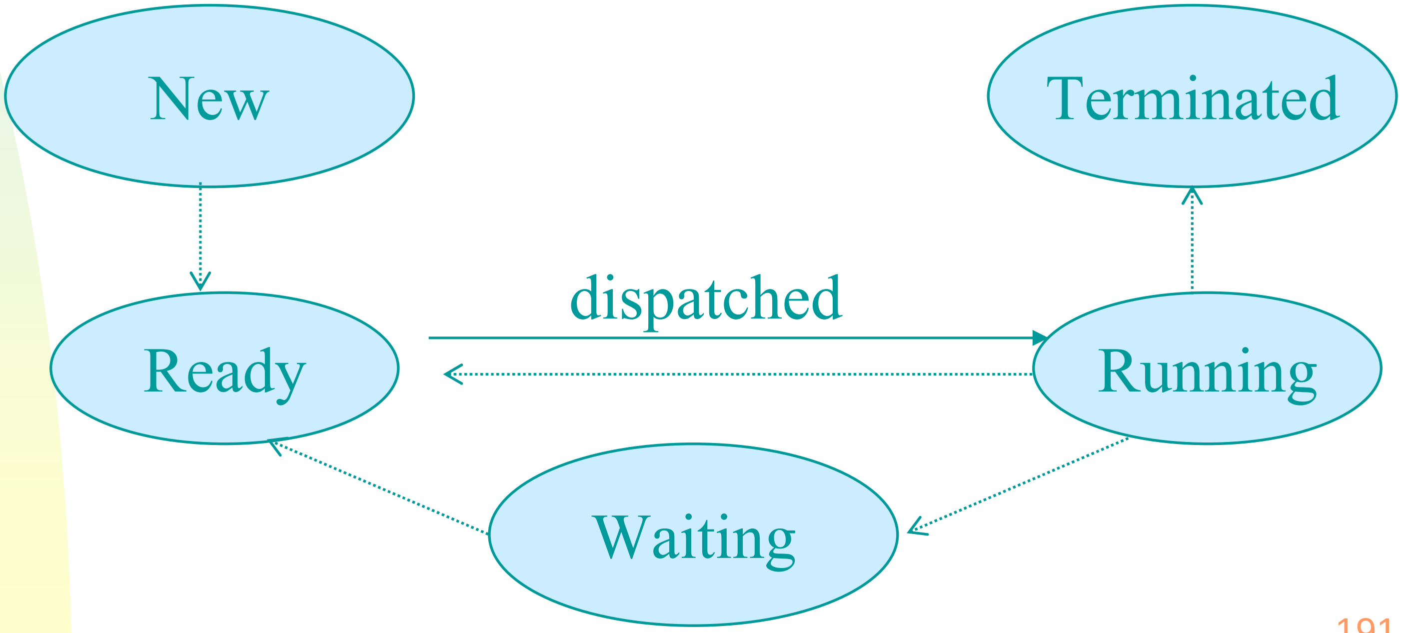
CPU Scheduling

- The distribution can help in selecting an appropriate CPU-scheduling algorithms



CPU Scheduling

- CPU Scheduler – The Selection of Process for Execution
  - A short-term scheduler



CPU Scheduling

- Nonpreemptive Scheduling
  - A running process keeps CPU until it volunteers to release CPU
    - E.g., I/O or termination
  - Advantage
    - Easy to implement (at the cost of service response to other processes)
  - E.g., Windows 3.1

CPU Scheduling

- Preemptive Scheduling
  - Beside the instances for non-preemptive scheduling, CPU scheduling occurs whenever some process becomes ready or the running process leaves the running state!
- Issues involved:
  - Protection of Resources, such as I/O queues or shared data, especially for multiprocessor or real-time systems.
  - Synchronization
    - E.g., Interrupts and System calls

CPU Scheduling

- Dispatcher
  - Functionality:
    - Switching context
    - Switching to user mode
    - Restarting a user program
  - Dispatch Latency:

Stop a process $\xleftrightarrow{\text{Must be fast}}$ Start a process

Scheduling Criteria

- Why?
  - Different scheduling algorithms may favor one class of processes over another!
- Criteria
  - CPU Utilization
  - Throughput
  - Turnaround Time: CompletionT-StartT
  - Waiting Time: Waiting in the ReadyQ
  - Response Time: FirstResponseTime

Scheduling Criteria

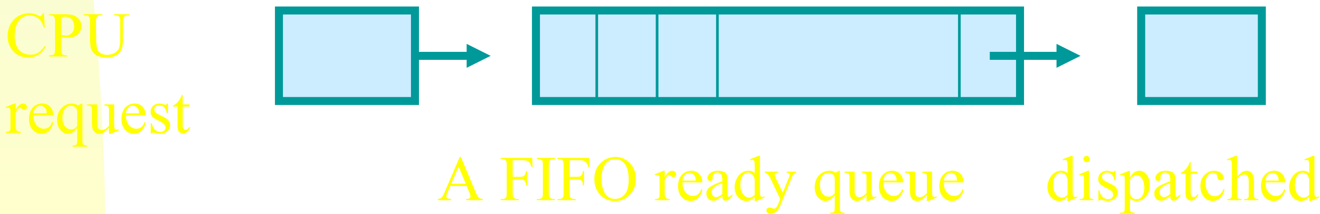
- How to Measure the Performance of CPU Scheduling Algorithms?
- Optimization of what?
  - General Consideration
    - Average Measure
    - Minimum or Maximum Values
  - Variance → Predictable Behavior

Scheduling Algorithms

- First-Come, First-Served Scheduling (FIFO)
- Shortest-Job-First Scheduling (SJF)
- Priority Scheduling
- Round-Robin Scheduling (RR)
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling
- Multiple-Processor Scheduling

First-Come, First-Served Scheduling (FCFS)

- The process which requests the CPU first is allocated the CPU
- Properties:
  - Non-preemptive scheduling
  - CPU might be hold for an extended period.



First-Come, First-Served Scheduling (FCFS)

- Example

Process	CPU Burst Time
P1	24
P2	3
P3	3

Gantt Chart

P1	P2	P3
0	24	27

30

Average waiting time = (0+24+27)/3 = 17

P2	P3	P1
0	3	6

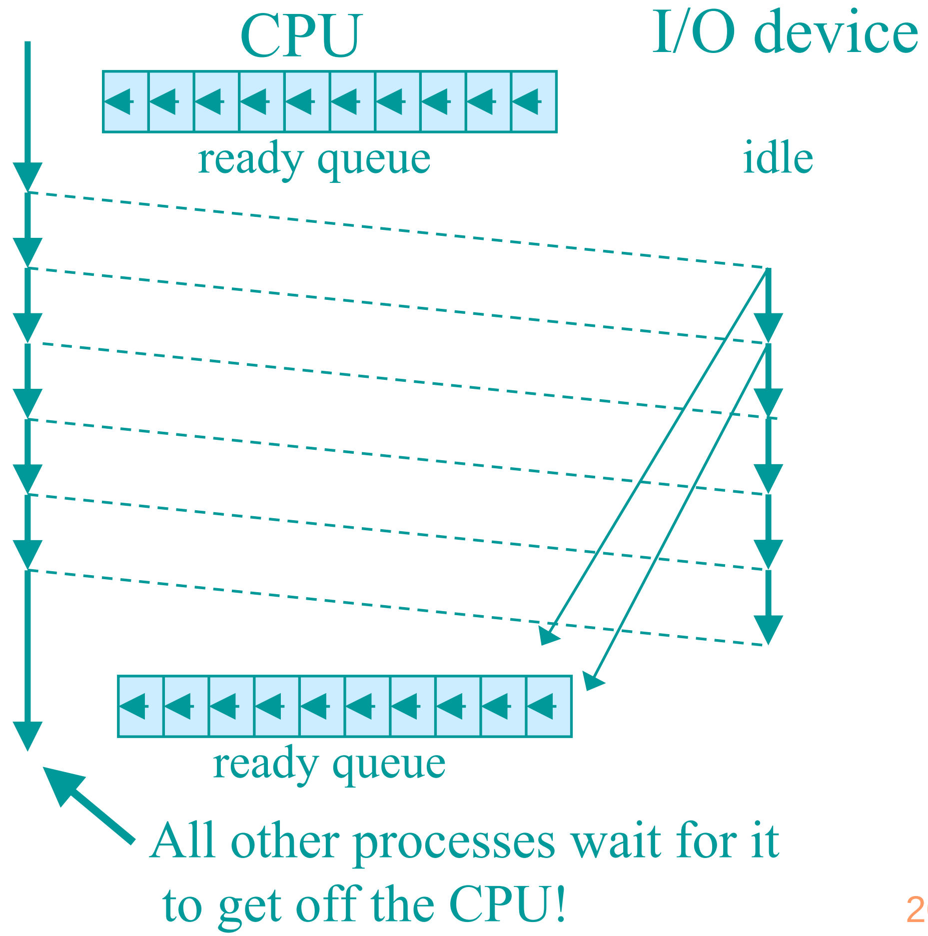
30

Average waiting time = (6+0+3)/3 = 3

*The average waiting time is highly affected by process CPU burst times !

First-Come, First-Served Scheduling (FCFS)

- Example: Convoy Effect
  - One CPU-bound process + many I/O-bound processes



Shortest-Job-First Scheduling (SJF)

- Non-Preemptive SJF
  - Shortest next CPU burst first

process	CPU burst time
P1	6
P2	8
P3	7
P4	3

Average waiting time
= (3+16+9+0)/4 = 7

Shortest-Job-First Scheduling (SJF)

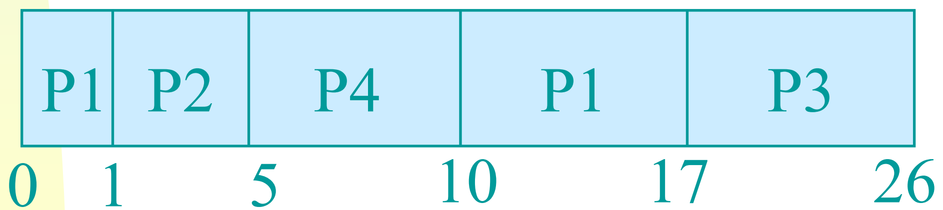
- Nonpreemptive SJF is optimal when processes are all ready at time 0
  - The minimum average waiting time!
- Prediction of the next CPU burst time?
  - Long-Term Scheduler
    - A specified amount at its submission time
  - Short-Term Scheduler
    - Exponential average ($0 <= \alpha <= 1$)

$$\tau_{n+1} = \alpha\, t_n + (1-\alpha)\, \tau_n$$

Shortest-Job-First Scheduling (SJF)

- Preemptive SJF
  - Shortest-remaining-time-first

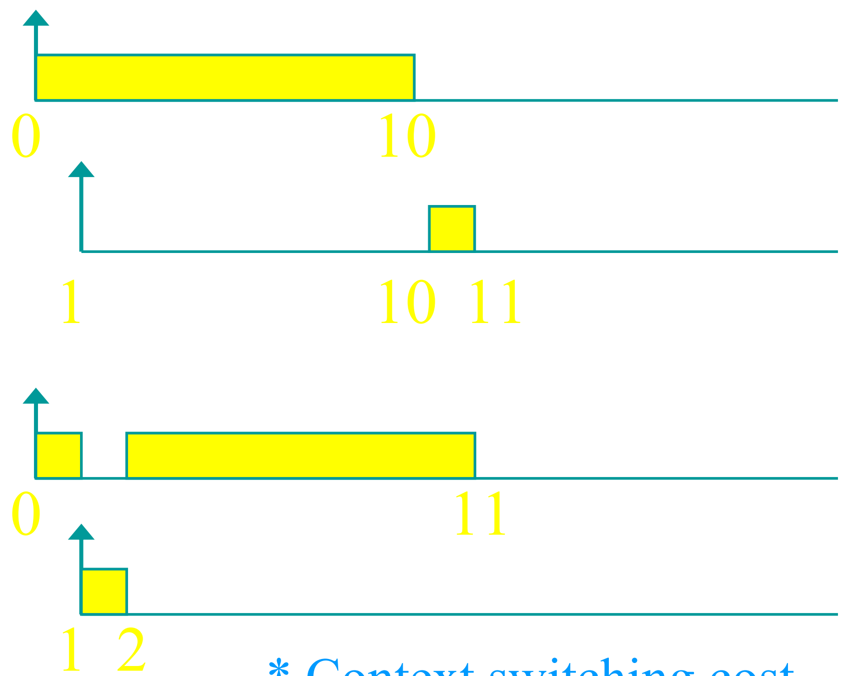
Process	CPU Burst Time	Arrival Time
P1	8	0
P2	4	1
P3	9	2
P4	5	3



Average Waiting Time = ((10-1) + (1-1) + (17-2) + (5-3))/4 = 26/4 = 6.5

Shortest-Job-First Scheduling (SJF)

- Preemptive or Non-preemptive?
  - Criteria such as AWT (Average Waiting Time)



Non-preemptive
AWT = (0+(10-1))/2
= 9/2 = 4.5

or

Preemptive AWT
= ((2-1)+0) = 0.5

* Context switching cost ~ modeling & analysis

Priority Scheduling

- CPU is assigned to the process with the highest priority – A framework for various scheduling algorithms:
  - FCFS: Equal-Priority with Tie-Breaking by FCFS
  - SFJ: Priority = 1 / next CPU burst length

Priority Scheduling

Process	CPU Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

Average waiting time
= (6+0+16+18+1)/5 = 8.2

Gantt Graph

P2	P5	P1	P3	P4

0 1 6 16 18 19

Priority Scheduling

- Priority Assignment
  - Internally defined – use some measurable quantity, such as the # of open files, $\frac{\text{Average CPU Burst}}{\text{Average I/O Burst}}$
  - Externally defined – set by criteria external to the OS, such as the criticality levels of jobs.

Priority Scheduling

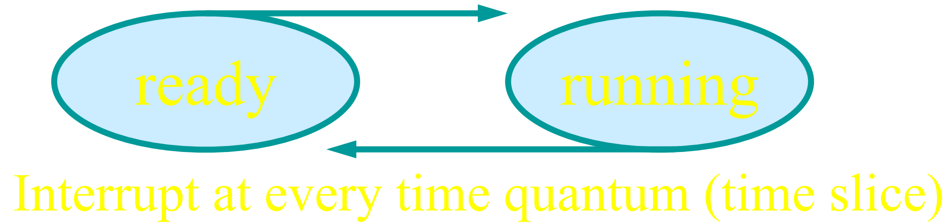
- Preemptive or Non-Preemptive?
  - Preemptive scheduling – CPU scheduling is invoked whenever a process arrives at the ready queue, or the running process relinquishes the CPU.
  - Non-preemptive scheduling – CPU scheduling is invoked only when the running process relinquishes the CPU.

Priority Scheduling

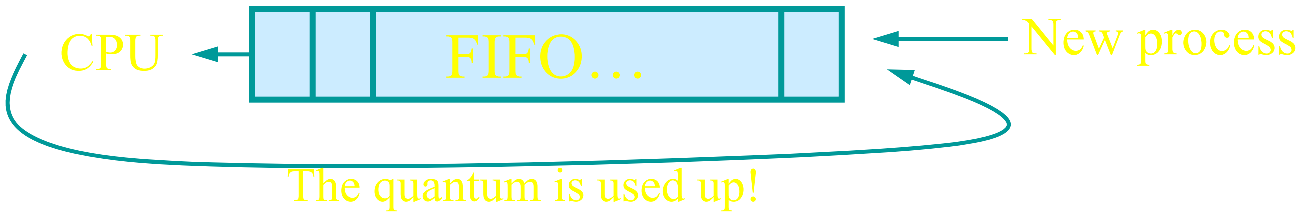
- Major Problem
  - Indefinite Blocking (/Starvation)
    - Low-priority processes could starve to death!
  - A Solution: Aging
    - A technique that increases the priority of processes waiting in the system for a long time.

Round-Robin Scheduling (RR)

- RR is similar to FCFS except that preemption is added to switch between processes.



- Goal: Fairness – Time Sharing



Round-Robin Scheduling (RR)

Process	CPU Burst Time
P1	24
P2	3
P3	3

Time slice = 4

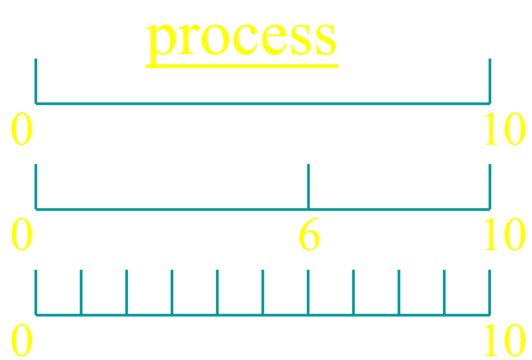
P1	P2	P3	P1	P1	P1	P1	P1

0 4 7 10 14 18 22 26 30

AWT = ((10-4) + (4-0) + (7-0))/3
= 17/3 = 5.66

Round-Robin Scheduling (RR)

- Service Size and Interval
  - Time quantum = q → Service interval <= (n-1)*q if n processes are ready.
  - IF q = ∞, then RR → FCFS.
  - IF q = ε, then RR → processor sharing. The # of context switchings increases!



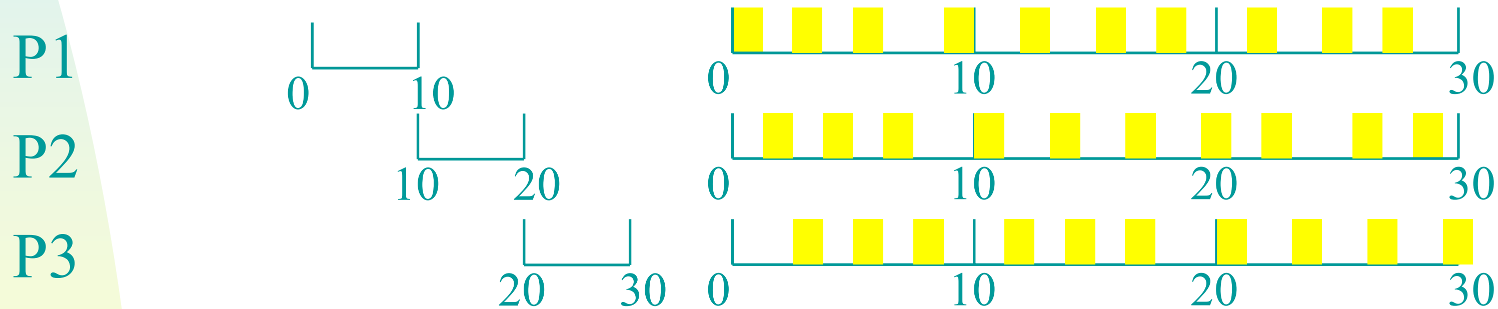
quantum	context switch #
12	0
6	1
1	9

$$\frac{\text{If context switch cost}}{\text{time quantum}} = 10\% \quad \Rightarrow \quad 1/11 \text{ of CPU is wasted!}$$

Round-Robin Scheduling (RR)

- Turnaround Time

process (10ms)   quantum = 10   quantum = 1



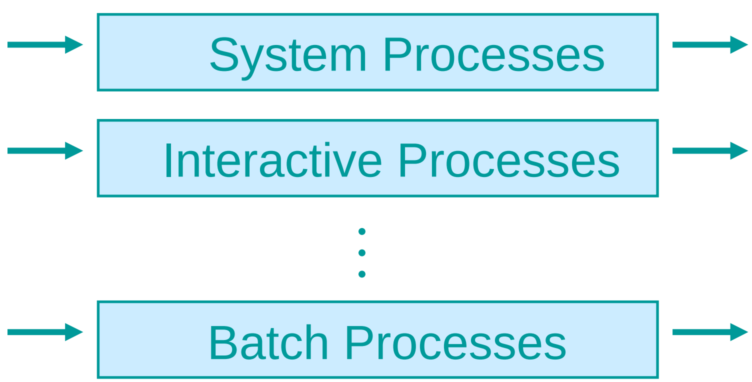
Average Turnaround Time
= (10+20+30)/3 = 20

ATT = (28+29+30)/3 = 29

=> 80% CPU Burst < time slice

Multilevel Queue Scheduling

- Partition the ready queue into several separate queues => Processes can be classified into different groups and permanently assigned to one queue.



Multilevel Queue Scheduling

- Intra-queue scheduling
  - Independent choice of scheduling algorithms.
- Inter-queue scheduling
  a. Fixed-priority preemptive scheduling
     a. e.g., foreground queues always have absolute priority over the background queues.
  b. Time slice between queues
     a. e.g., 80% CPU is given to foreground processes, and 20% CPU to background processes.
  c. More??

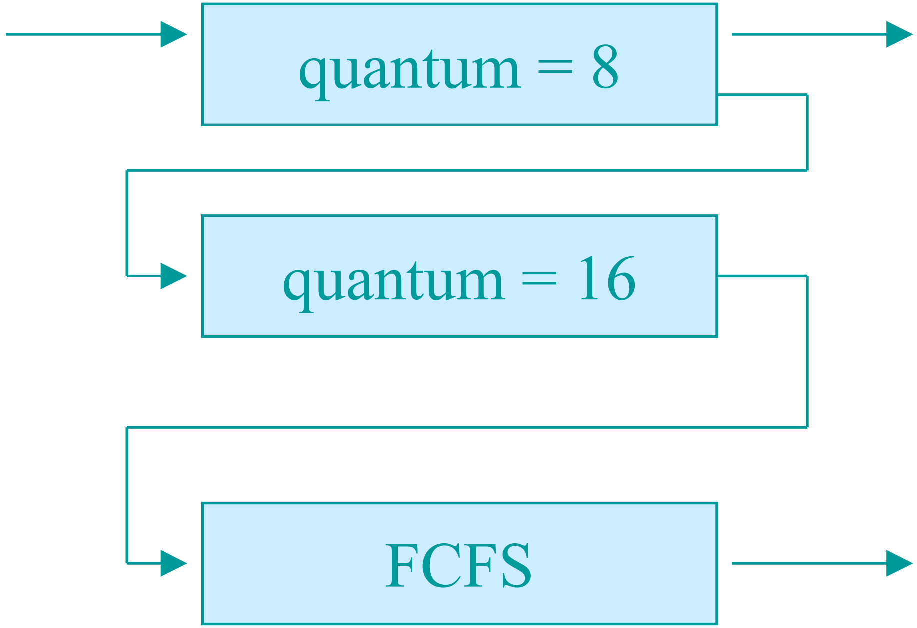
Multilevel Feedback Queue Scheduling

- Different from Multilevel Queue Scheduling by Allowing Processes to Migrate Among Queues.
  - Configurable Parameters:
    a. # of queues
    b. The scheduling algorithm for each queue
    c. The method to determine when to upgrade a process to a higher priority queue.
    d. The method to determine when to demote a process to a lower priority queue.
    e. The method to determine which queue a newly ready process will enter.

*Inter-queue scheduling: Fixed-priority preemptive?!

Multilevel Feedback Queue Scheduling

- Example



*Idea: Separate processes with different CPU-burst characteristics!

Multiple-Processor Scheduling

- CPU scheduling in a system with multiple CPUs
- A Homogeneous System
  - Processes are identical in terms of their functionality.
    - ➔ Can processes run on any processor?
- A Heterogeneous System
  - Programs must be compiled for instructions on proper processors.

Multiple-Processor Scheduling

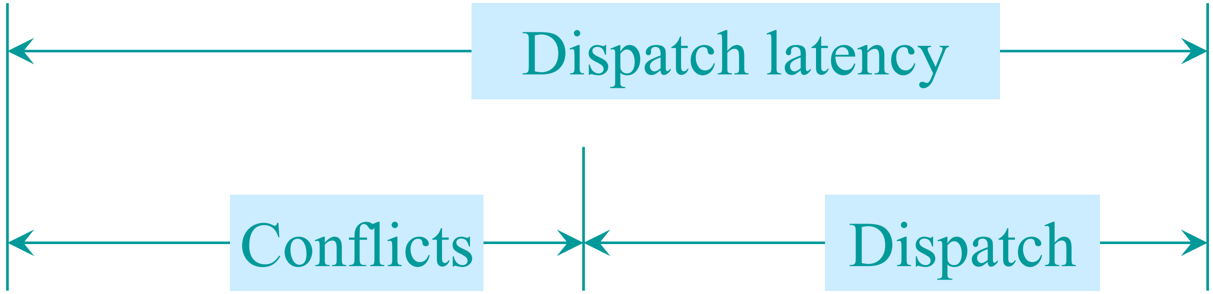
- Load Sharing – Load Balancing!!
  - A queue for each processor
    - Self-Scheduling – Symmetric Multiprocessing
  - A common ready queue for all processors.
    - Self-Scheduling
      - Need synchronization to access common data structure, e.g., queues.
    - Master-Slave – Asymmetric Multiprocessing
      - One processor accesses the system structures → no need for data sharing

Real-Time Scheduling

- Definition
  - Real-time means on-time, instead of fast!
  - Hard real-time systems:
    - Failure to meet the timing constraints (such as deadline) of processes may result in a catastrophe!
  - Soft real-time systems:
    - Failure to meet the timing constraints may still contribute value to the system.

Real-Time Scheduling

- Dispatch Latency



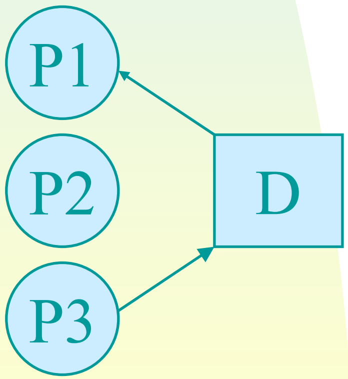
1. Preemption of the running process
2. Releasing resources needed by the higher priority process
3. Context switching to the higher priority process

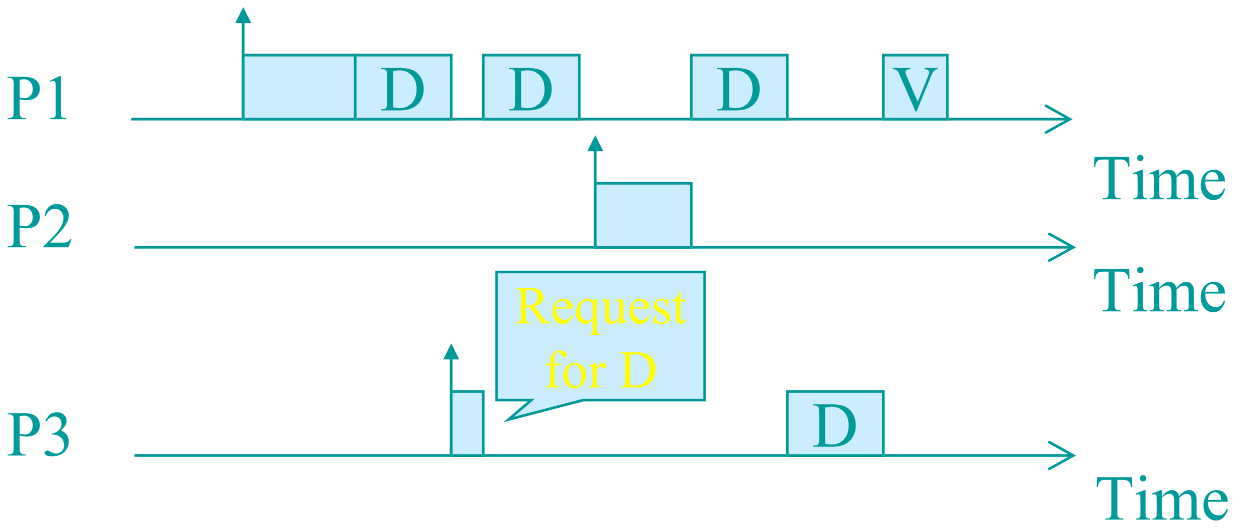
Real-Time Scheduling

- Minimization of Dispatch Latency?
  - Context switching in many OS, e.g., some UNIX versions, can only be done after a system call completes or an I/O blocking occurs
- Solutions:
  1. Insert safe preemption points in long-duration system calls.
  2. Protect kernel data by some synchronization mechanisms to make the entire kernel preemptible.

Real-Time Scheduling

- Priority Inversion:
  - A higher-priority processes must wait for the execution of a lower-priority processes.

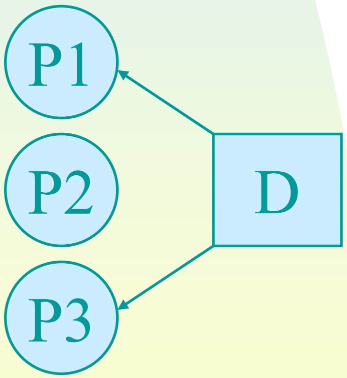


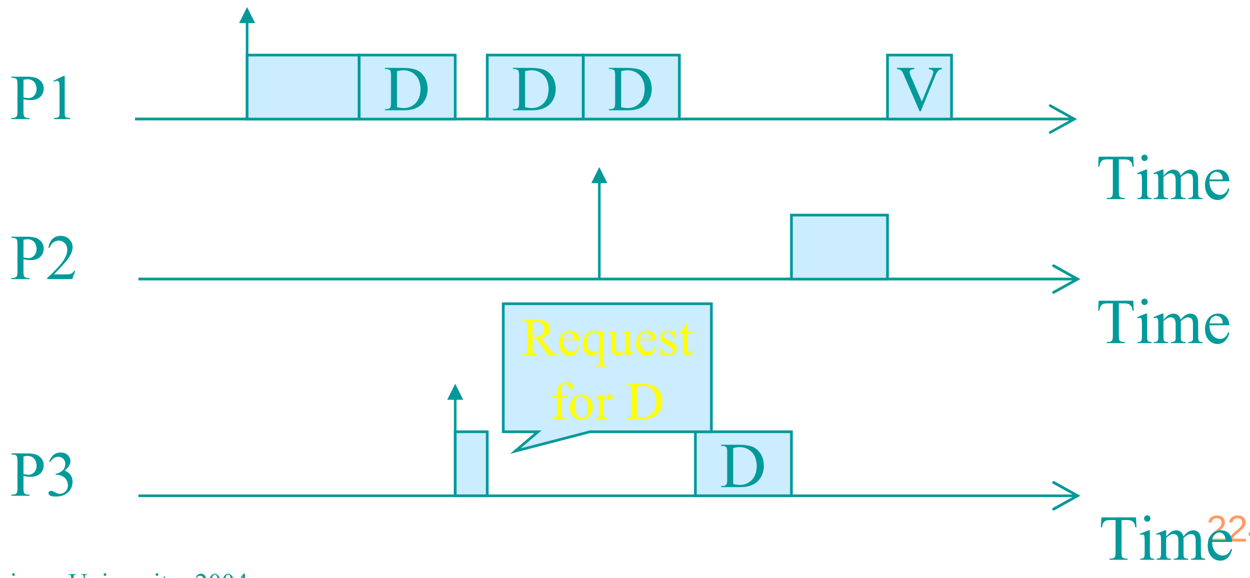


P3 also waits for P2 to complete?

Real-Time Scheduling

- Priority Inheritance
  - The blocked process inherits the priority of the process that causes the blocking.





Real-Time Scheduling

- Earliest Deadline First Scheduling (EDF)
  - Processes with closer deadlines have higher priorities.

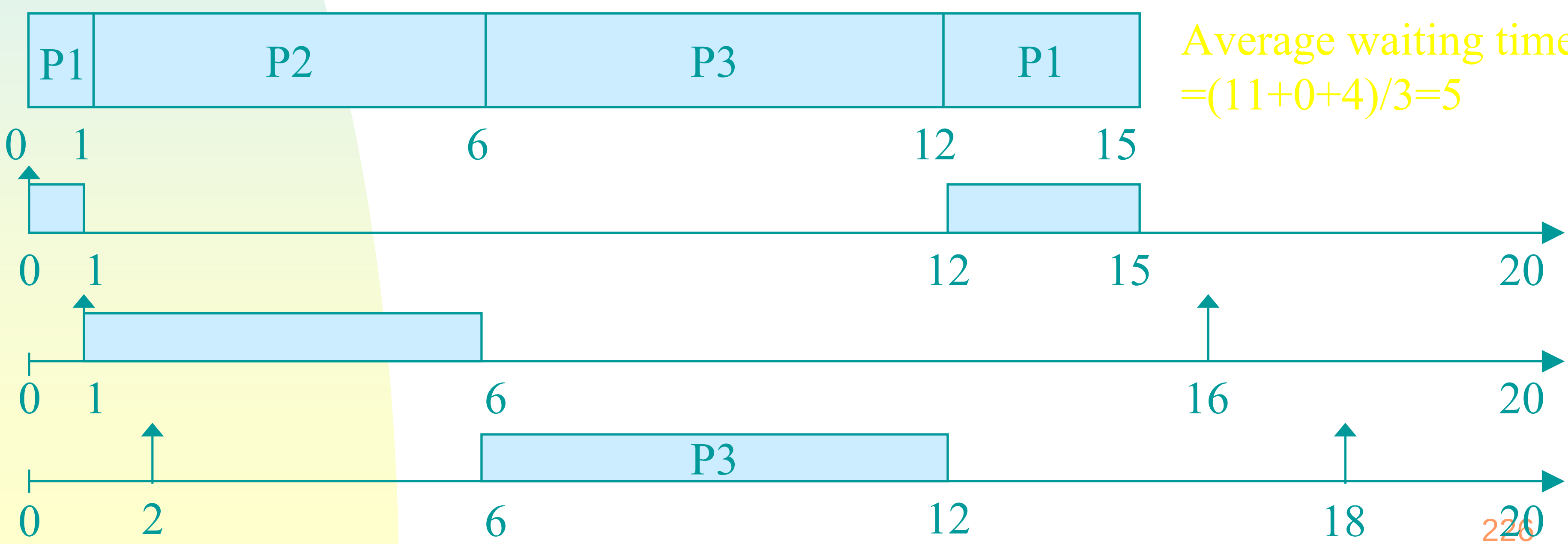
$$(\,priority\,(\tau_i) \propto (1/d_i))$$

  - An optimal dynamic-priority-driven scheduling algorithm for periodic and aperiodic processes!

Liu & Layland [JACM 73] showed that EDF is optimal in the sense that a process set is scheduled by EDF if its CPU utilization $\sum\left(C_i/P_i\right)$ is no larger than 100%.

Real-Time Scheduling – EDF

process	CPU Burst time	Deadline	Initial Arrival Time
P1	4	20	0
P2	5	15	1
P3	6	16	2

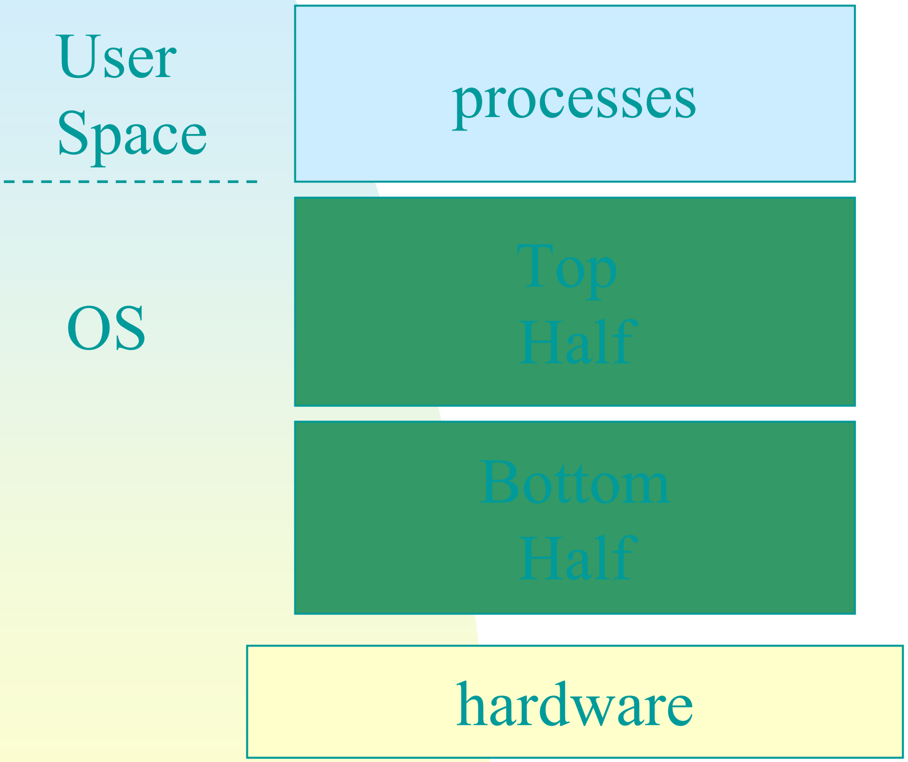


Average waiting time =(11+0+4)/3=5

A General Architecture of RTOS’s

- Objectives in the Design of Many RTOS’s
  - Efficient Scheduling Mechanisms
  - Good Resource Management Policies
  - Predictable Performance
- Common Functionality of Many RTOS’s
  - Task Management
  - Memory Management
  - Resource Control, including devices
  - Process Synchronization

A General Architecture



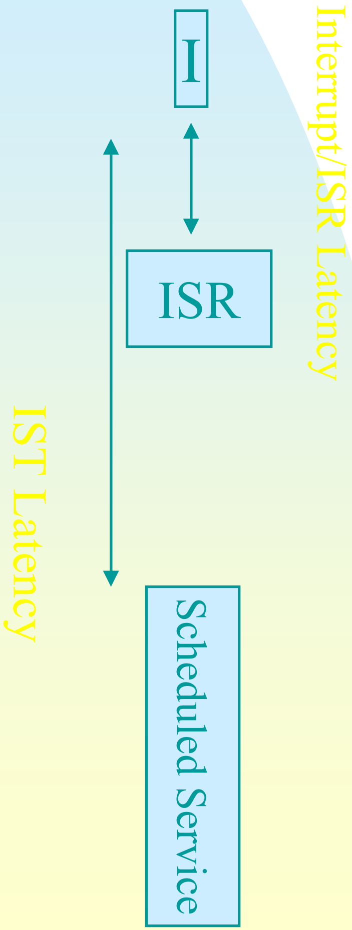
System calls such as I/O requests which may cause the releasing CPU of a process!

Timer expires to

- Expire the running process's time quota
- Keep the accounting info for each process

Interrupts for Services

A General Architecture



- 2-Step Interrupt Services
  - Immediate Interrupt Service
    - Interrupt priorities > process priorities
    - Time: Completion of higher priority ISR, context switch, disabling of certain interrupts, starting of the right ISR (urgent/low-level work, set events)
  - Scheduled Interrupt Service
    - Usually done by preemptible threads
  - Remark: Reducing of non-preemptible code, Priority Tracking/Inheritance (LynxOS), etc.

A General Architecture

- Scheduler
  - A central part in the kernel
  - The scheduler is usually driven by a clock interrupt periodically, except when voluntary context switches occur – thread quantum?
- Timer Resolution
  - Tick size vs Interrupt Frequency
    - 10ms? 1ms? 1us? 1ns?
  - Fine-Grained hardware clock

A General Architecture

- Memory Management
  - No protection for many embedded systems
  - Memory-locking to avoid paging
- Process Synchronization
  - Sources of Priority Inversion
    - Nonpreemptible code
      - Critical sections
    - A limited number of priority levels, etc.

Algorithm Evaluation

- A General Procedure
  - Select criteria that may include several measures, e.g., maximize CPU utilization while confining the maximum response time to 1 second
  - Evaluate various algorithms
- Evaluation Methods:
  - Deterministic modeling
  - Queuing models
  - Simulation
  - Implementation

Deterministic Modeling

- A Typical Type of Analytic Evaluation
  - Take a particular predetermined workload and defines the performance of each algorithm for that workload
- Properties
  - Simple and fast
  - Through excessive executions of a number of examples, treads might be identified
  - But it needs exact numbers for inputs, and its answers only apply to those cases
    - Being too specific and requires too exact knowledge to be useful!

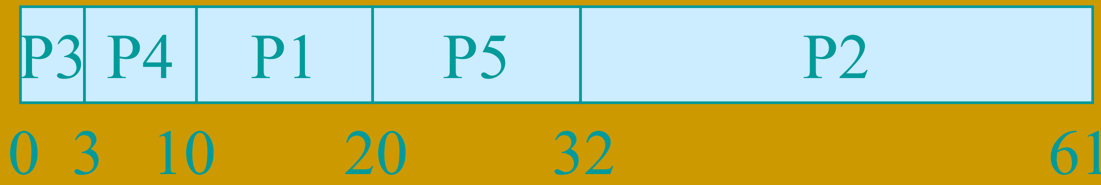
Deterministic Modeling

process	CPU Burst time
P1	10
P2	29
P3	3
P4	7
P5	12

FCFC

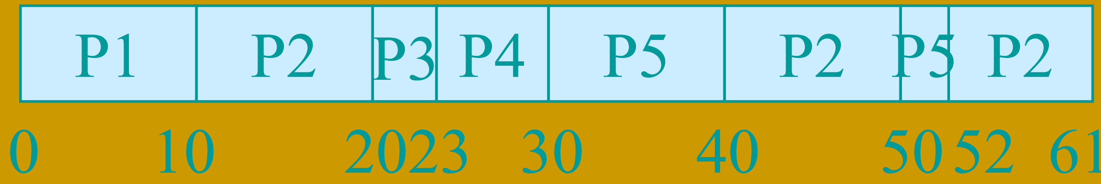
Average Waiting Time (AWT)=(0+10+39+42+49)/5=28

Nonpreemptive Shortest Job First



AWT=(10+32+0+3+20)/5=13

Round Robin (quantum =10)



AWT=(0+(10+20+2)+20+23+(30+10))/5=23

Queueing Models

- Motivation:
  - Workloads vary, and there is no static set of processes
- Models (~ Queueing-Network Analysis)
  - Workload:
    a. Arrival rate: the distribution of times when processes arrive.
    b. The distributions of CPU & I/O bursts
  - Service rate

Queueing Models

- Model a computer system as a network of servers. Each server has a queue of waiting processes
  - Compute average queue length, waiting time, and so on.
- Properties:
  - Generally useful but with limited application to the classes of algorithms & distributions
  - Assumptions are made to make problems solvable => inaccurate results

Queueing Models

- Example: Little's formula

$$n = \lambda * w$$

n = # of processes in the queue

λ = arrival rate

ω = average waiting time in the queue

- If n =14 & λ =7 processes/sec, then w = 2 seconds.

Simulation

- Motivation:
  - Get a more accurate evaluation.
- Procedures:
  - Program a model of the computer system
  - Drive the simulation with various data sets
    - Randomly generated according to some probability distributions

      => inaccuracy occurs because of only the occurrence frequency of events. Miss the order & the relationships of events.
    - Trace tapes: monitor the real system & record the sequence of actual events.

Simulation

- Properties:
  - Accurate results can be gotten, but it could be expensive in terms of computation time and storage space.
  - The coding, design, and debugging of a simulator can be a big job.

Implementation

- Motivation:
  - Get more accurate results than a simulation!
- Procedure:
  - Code scheduling algorithms
  - Put them in the OS
  - Evaluate the real behaviors

Implementation

- Difficulties:
  - Cost in coding algorithms and modifying the OS
  - Reaction of users to a constantly changing the OS
  - The environment in which algorithms are used will change
    - For example, users may adjust their behaviors according to the selected algorithms

  => Separation of the policy and mechanism!

Process Scheduling Model

- Process Local Scheduling
  - E.g., those for user-level threads
  - Thread scheduling is done locally to each application.
- System Global Scheduling
  - E.g., those for Kernel-level threads
  - The kernel decides which thread to run.

Process Scheduling Model – Solaris 2

- Priority-Based Process Scheduling
  - Real-Time
  - System
    - Kernel-service processes
  - Time-Sharing
    - A default class
  - Interactive
- Each LWP inherits its class from its parent process

low

Process Scheduling Model – Solaris 2

- Real-Time
  - A guaranteed response
- System
  - The priorities of system processes are fixed.
- Time-Sharing
  - Multilevel feedback queue scheduling – priorities inversely proportional to time slices
- Interactive
  - Prefer windowing process

Process Scheduling Model – Solaris 2

- The selected thread runs until one of the following occurs:
  - It blocks.
  - It uses its time slice (if it is not a system thread).
  - It is preempted by a higher-priority thread.
- RR is used when several threads have the same priority.

Process Scheduling Model – Windows 2000

- Priority-Based Preemptive Scheduling
  - Priority Class/Relationship: 0..31
  - Dispatcher: A process runs until
    - It is preempted by a higher-priority process.
    - It terminates
    - Its time quantum ends
    - It calls a blocking system call
  - Idle thread
- A queue per priority level

Process Scheduling Model – Windows 2000

- Each thread has a base priority that represents a value in the priority range of its class.
- A typical class – Normal_Priority_Class
- Time quantum – thread
  - Increased after some waiting
    - Different for I/O devices.
  - Decreased after some computation
    - The priority is never lowered below the base priority.
  - Favor foreground processes (more time quantum)

Process Scheduling Model – Windows 2000

A Typical Class

	Real-time	High	Above normal	Normal	Below normal	Idle priority
Time-critical	31	15	15	15	15	15
Highest	26	15	12	10	8	6
Above normal	25	14	11	9	7	5
Normal	24	13	10	8	6	4
Below normal	23	12	9	7	5	3
Lowest	22	11	8	6	4	2
Idle	16	1	1	1	1	1

Base Priority

Real-Time Class

Variable Class (1..15)

Process Scheduling Model – Linux

- Three Classes (POSIX.1b)
  - Time-Sharing
  - Soft Real-Time: FCFS, and RR
- Real-Time Scheduling Algorithms
  - FCFS & RR always run the highest priority process.
  - FCFS runs a process until it exits or blocks.
- No scheduling in the kernel space for conventional Linux

Process Scheduling Model – Linux

- A Time-Sharing Algorithm for Fairness
  - Credits = (credits / 2) + priority
    - Recrediting when no runnable process has any credits.
    - Mixture of a history and its priority
  - Favor interactive or I/O-bound processes
  - Background processes could be given lower priorities to receive less credits.
    - *nice* in UNIX

Contents

Chapter 7
Process Synchronization

Process Synchronization

- Why Synchronization?
  - To ensure data consistency for concurrent access to shared data!
- Contents:
  - Various mechanisms to ensure the orderly execution of cooperating processes

Process Synchronization

- A Consumer-Producer Example

- Producer

```
while (1) {
    while (counter == BUFFER_SIZE)
        ;
    produce an item in nextp;
    ….
    buffer[in] = nextp;
    in = (in+1) % BUFFER_SIZE;
    counter++;
}
```

- Consumer:

```
while (1) {
    while (counter == 0)
        ;
    nextc = buffer[out];
    out = (out +1) % BUFFER_SIZE;
    counter--;
    consume an item in nextc;
}
```

Process Synchronization

- counter++ vs counter—

  r1 = counter r2 = counter

  r1 = r1 + 1 r2 = r2 - 1

  counter = r1 counter = r2

- Initially, let counter = 5.

1. P: r1 = counter
2. P: r1 = r1 + 1
3. C: r2 = counter
4. C: r2 = r2 – 1
5. P: counter = r1
6. C: counter = r2

A Race Condition!

Process Synchronization

- A Race Condition:
  - A situation where the outcome of the execution depends on the particular order of process scheduling.
- The Critical-Section Problem:
  - Design a protocol that processes can use to cooperate.
    - Each process has a segment of code, called a <u>critical section</u>, whose execution must be <u>mutually exclusive</u>.

Process Synchronization

- A General Structure for the Critical-Section Problem

```
do {
        entry section;        ← permission request
            critical section;
        exit section;         ← exit notification
            remainder section;
} while (1);
```

The Critical-Section Problem

- Three Requirements

1. Mutual Exclusion
   a. Only one process can be in its critical section.
2. Progress
   a. Only processes not in their remainder section can decide which will enter its critical section.
   b. The selection cannot be postponed indefinitely.
3. Bounded Waiting
   a. A waiting process only waits for a bounded number of processes to enter their critical sections.

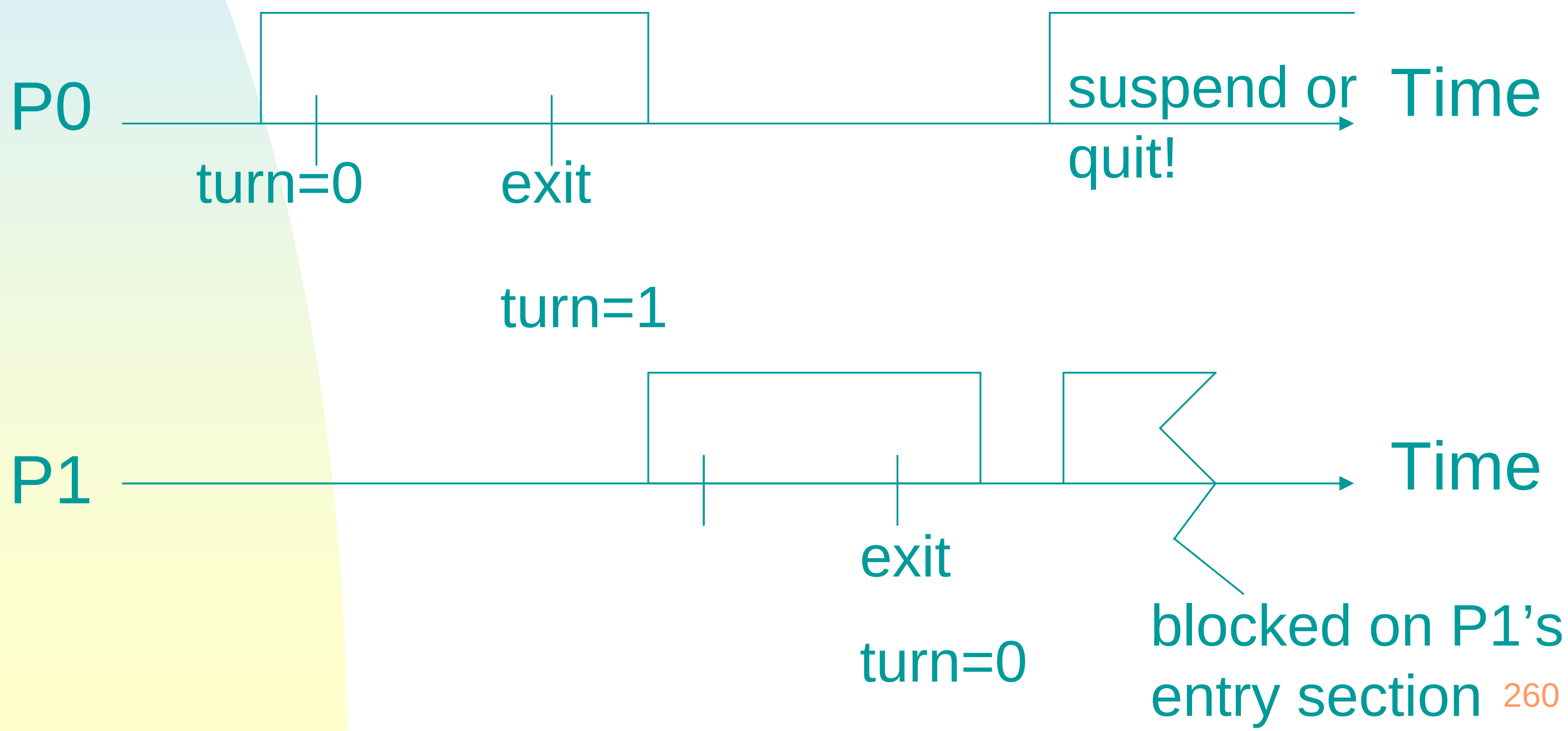
The Critical-Section Problem – A Two-Process Solution

- Notation
  - Processes Pi and Pj, where j=1-i;
- Assumption
  - Every basic machine-language instruction is atomic.
- Algorithm 1
  - Idea: Remember which process is allowed to enter its critical section, That is, process i can enter its critical section if turn = i.

```
do {
    while (turn != i) ;
    critical section
    turn=j;
    remainder section
} while (1);
```

The Critical-Section Problem – A Two-Process Solution

- Algorithm 1 fails the progress requirement:



The Critical-Section Problem – A Two-Process Solution

- Algorithm 2
  - Idea: Remember the state of each process.
  - flag[i]==true → Pi is ready to enter its critical section.
  - Algorithm 2 fails the progress requirement when flag[0]==flag[1]==true;
    - the exact timing of the two processes?

Initially, flag[0]=flag[1]=false

```
do {
    flag[i]=true;
    while (flag[j]) ;
    critical section
    flag[i]=false;
    remainder section
} while (1);
```


* The switching of "flag[i]=true" and "while (flag[j]);".

The Critical-Section Problem – A Two-Process Solution

- Algorithm 3
  - Idea: Combine the ideas of Algorithms 1 and 2
  - When (flag[i] && turn=i), Pj must wait.
  - Initially, flag[0]=flag[1]=false, and turn = 0 or 1

```
do {
    flag[i]=true;
    turn=j;
    while (flag[j] && turn==j) ;
    critical section
    flag[i]=false;
    remainder section
} while (1);
```

The Critical-Section Problem – A Two-Process Solution

- Properties of Algorithm 3
  - Mutual Exclusion
    - The eventual value of *turn* determines which process enters the critical section.
  - Progress
    - A process can only be stuck in the while loop, and the process which can keep it waiting must be in its critical sections.
  - Bounded Waiting
    - Each process wait at most one entry by the other process.

The Critical-Section Problem – A Multiple-Process Solution

- Bakery Algorithm
  - Originally designed for distributed systems
  - Processes which are ready to enter their critical section must take a number and wait till the number becomes the lowest.
  - int number[i]: Pi's number if it is nonzero.
  - boolean choosing[i]: Pi is taking a number.

The Critical-Section Problem – A Multiple-Process Solution

```
do {
    choosing[i]=true;
    number[i]=max(number[0], …number[n-1])+1;
    choosing[i]=false;
    for (j=0; j < n; j++)
        while choosing[j] ;
        while (number[j] != 0 && (number[j],j)<(number[i],i)) ;
    critical section
    number[i]=0;
    remainder section
} while (1);
```

- An observation: If Pi is in its critical section, and Pk (k != i) has already chosen its number[k], then (number[i],i) < (number[k],k).

Synchronization Hardware

- Motivation:
  - Hardware features make programming easier and improve system efficiency.
- Approach:
  - Disable Interrupt → No Preemption
    - Infeasible in multiprocessor environment where message passing is used.
    - Potential impacts on interrupt-driven system clocks.
  - Atomic Hardware Instructions
    - Test-and-set, Swap, etc.

Synchronization Hardware

```
boolean TestAndSet(boolean &target) {
    boolean rv = target;
    target=true;
    return rv;
}
```

```
do {
    while (TestAndSet(lock)) ;
      critical section
    lock=false;
      remainder section
} while (1);
```

Synchronization Hardware

```
void Swap(boolean &a, boolean &b) {
    boolean temp = a;
    a=b;
    b=temp;
}
```

```
do {
    key=true;
    while (key == true)
        Swap(lock, key);
      critical section
    lock=false;
      remainder section
} while (1);
```

Synchronization Hardware

```
do {
      waiting[i]=true;
      key=true;
      while (waiting[i] && key)
                key=TestAndSet(lock);
      waiting[i]=false;
      critical section;
      j= (i+1) % n;
      while(j != i) && (not waiting[j])
                j= (j+1) % n;
      If (j=i) lock=false;
      else waiting[j]=false;
      remainder section
  } while (1);
```

- Mutual Exclusion
  - Pass if key == F or waiting[i] == F
- Progress
  - Exit process sends a process in.
- Bounded Waiting
  - Wait at most n-1 times
- Atomic TestAndSet is hard to implement in a multiprocessor environment.

Semaphores

- Motivation:
  - A high-level solution for more complex problems.
- Semaphore
  - A variable S only accessible by two atomic operations:

```
wait(S) {        /* P */
    while (S <= 0) ;
    S—;
}
```

```
signal(S) {     /* V */
    S++;
}
```

•Indivisibility for “(S<=0)”, “S—”, and “S++”

Semaphores – Usages

- Critical Sections

```
do {
    wait(mutex);
    critical section
    signal(mutex);
    remainder section
} while (1);
```

- Precedence Enforcement

```
P1:
    S1;
    signal(synch);

P2:
    wait(synch);
    S2;
```

Semaphores

- Implementation
  - Spinlock – A Busy-Waiting Semaphore
    - “while (S <= 0)” causes the wasting of CPU cycles!
    - Advantage:
      - When locks are held for a short time, spinlocks are useful since no context switching is involved.
  - Semaphores with Block-Waiting
    - No busy waiting from the entry to the critical section!

Semaphores

- Semaphores with Block Waiting

```
typedef struct {
        int value;
        struct process *L;
} semaphore ;
```

```
void wait(semaphore S) {
        S.value--;
        if (S.value < 0) {
            add this process to S.L;
            block();
        }
}
```

```
void signal(semaphore S);
        S.value++;
        if (S.value <= 0) {
            remove a process P form S.L;
                wakeup(P);
        }
}
```

* |S.value| = the # of waiting processes if S.value < 0.

Semaphores

- The queueing strategy can be arbitrary, but there is a restriction for the bounded-waiting requirement.
- Mutual exclusion in wait() & signal()
  - Uniprocessor Environments
    - Interrupt Disabling
    - TestAndSet, Swap
    - Software Methods, e.g., the Bakery Algorithm, in Section 7.2
  - Multiprocessor Environments
- Remarks: Busy-waiting is limited to only the critical sections of the wait() & signal()!

Deadlocks and Starvation

- Deadlock
  - A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set.

```
P0:  wait(S);          P1: wait(Q);
     wait(Q);              wait(S);
       …                     …
     signal(S);            signal(Q);
     signal(Q);            signal(S);
```

- Starvation (or Indefinite Blocking)
  - E.g., a LIFO queue

Binary Semaphore

- Binary Semaphores versus Counting Semaphores
  - The value ranges from 0 to 1 → easy implementation!

<u>wait(S)</u>

```
wait(S1);    /* protect C */
C--;
if (C < 0) {
    signal(S1);
    wait(S2);
}
signal(S1);
```

<u>signal(S)</u>

```
wait(S1);
C++;
if (C <= 0)
    signal (S2); /* wakeup */
else
    signal (S1);
```

* S1 & S2: binary semaphores

Classical Synchronization Problems – The Bounded Buffer

Producer:

```
do {
        produce an item in nextp;
        …….
        wait(empty);  /* control buffer availability */
        wait(mutex);  /* mutual exclusion */
        ……
        add nextp to buffer;
        signal(mutex);
        signal(full);  /* increase item counts */
} while (1);
```

Initialized to *n* ⇨ wait(empty)

Initialized to *1* ⇨ wait(mutex)

Initialized to *0* ⇨ signal(full)

Classical Synchronization Problems – The Bounded Buffer

Consumer:

Initialized to *0* ⇨ Initialized to *1* ⇨ Initialized to *n* ⇨

```
do {
    wait(full);  /* control buffer availability */
    wait(mutex);  /* mutual exclusion */
    …….
    remove an item from buffer to nextp;
    ……
    signal(mutex);
    signal(empty);  /* increase item counts */
    consume nextp;
} while (1);
```

Classical Synchronization Problems – Readers and Writers

- The Basic Assumption:
  - Readers: shared locks
  - Writers: exclusive locks
- The first reader-writers problem
  - No readers will be kept waiting unless a writer has already obtained permission to use the shared object → potential hazard to writers!
- The second reader-writers problem:
  - Once a writer is ready, it performs its write asap! → potential hazard to readers!

Classical Synchronization Problems – Readers and Writers

First R/W Solution

semaphore wrt, mutex;

(initialized to 1);

int readcount=0;

Writer:

Queueing mechanism

```
wait(wrt);
……
writing is performed
……
signal(wrt)
```

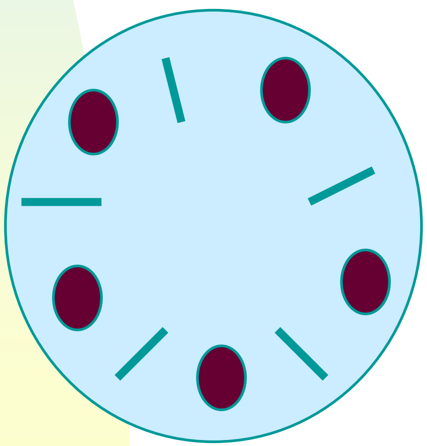
Reader:

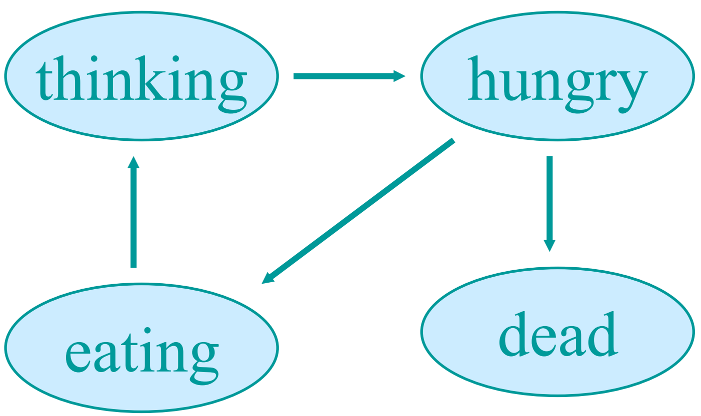
```
wait(mutex);
readcount++;
if (readcount == 1)
        wait(wrt);
signal(mutex);
…… reading ……
wait(mutex);
readcount--;
if (readcount== 0)
        signal(wrt);
signal(mutex);
```

Which is awaken?

Classical Synchronization Problems – Dining-Philosophers

- Each philosopher must pick up one chopstick beside him/her at a time
- When two chopsticks are picked up, the philosopher can eat.





Classical Synchronization Problems – Dining-Philosophers

```
semaphore chopstick[5];
do {
        wait(chopstick[i]);
        wait(chopstick[(i + 1) % 5 ]);
        … eat …
        signal(chopstick[i]);
        signal(chopstick[(i+1) % 5]);
        …think …
} while (1);
```

Classical Synchronization Problems – Dining-Philosophers

- Deadlock or Starvation?!
- Solutions to Deadlocks:
  - At most four philosophers appear.
  - Pick up two chopsticks "simultaneously".
  - Order their behaviors, e.g., odds pick up their right one first, and evens pick up their left one first.
- Solutions to Starvation:
  - No philosopher will starve to death.
    - A deadlock could happen??

Critical Regions

- Motivation:
  - Various programming errors in using low-level constructs,e.g., semaphores
    - Interchange the order of wait and signal operations
    - Miss some waits or signals
    - Replace waits with signals
    - etc
- The needs of high-level language constructs to reduce the possibility of errors!

Critical Regions

- Region v when B do S;
  - Variable v – shared among processes and only accessible in the region

```
struct buffer {
   item pool[n];
   int count, in, out;
};
```

  - B – condition
    - count < 0
  - S – statements

Example: Mutual Exclusion
region *v* when (true) *S1*;
region *v* when (true) *S2*;

Critical Regions – Consumer-Producer

```
struct buffer {
        item pool[n];
        int count, in, out;
};
```

Producer:

```
region buffer when
(count < n) {
        pool[in] = nextp;
        in = (in + 1) % n;
        count++;
}
```

Consumer:

```
region buffer when
(count > 0) {
        nextc = pool[out];
        out = (out + 1) % n;
        count--;
}
```

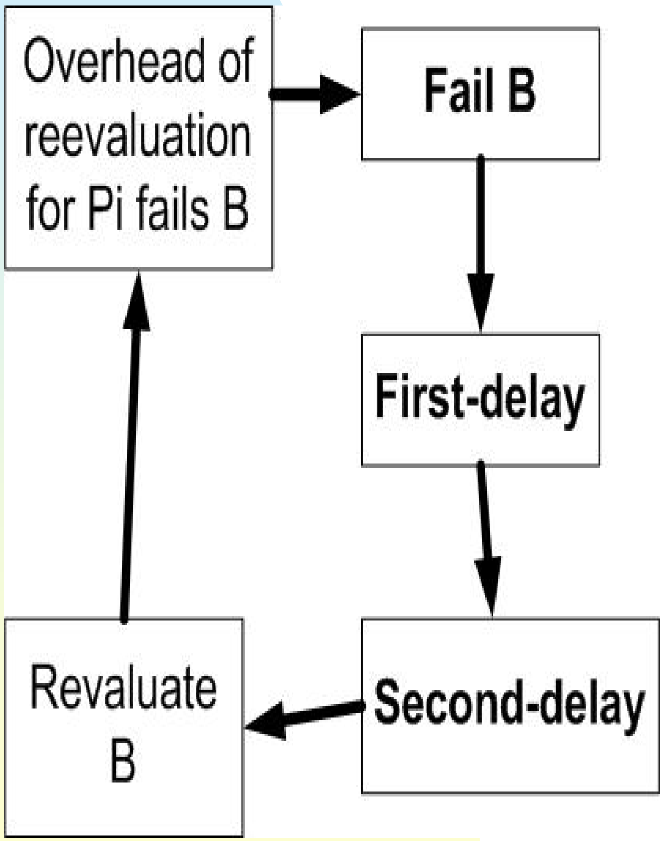
Critical Regions – Implementation by Semaphores

Region **x** when B do S;

```
/* to protect the region */
semaphore mutex;
/* to (re-)test B */
semaphore first-delay;
int first-count=0;
/* to retest B */
semaphore second-delay;
int second-count=0;
```

```
wait(mutex);
  while (!B) {
        /* fail B */
        first-count++;
        if (second-count > 0)
          /* try other processes waiting
              on second-delay */
          signal(second-delay);
          else signal(mutex);
        /* block itself on first-delay */
        wait(first-delay);
```

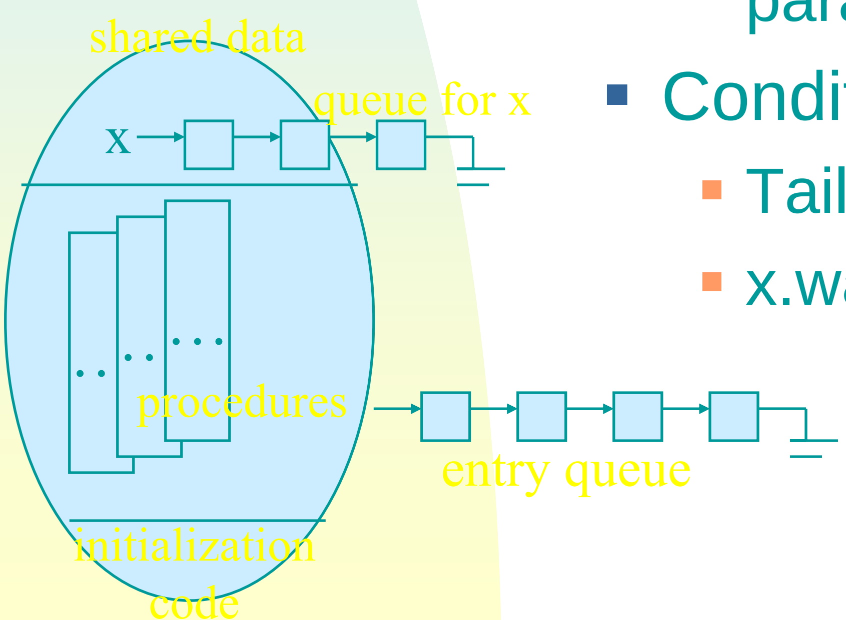
Critical Regions – Implementation by Semaphores




```
        first-count--;
        second-count++;
        if (first-count > 0)
           signal(first-delay);
           else signal(second-delay);
        /* block itself on first-delay */
        wait(second-delay);
        second-count--;
}
S;
if (first-count > 0)
        signal(first-delay);
else if (second-count > 0)
        signal(second-delay);
else signal(mutex);
```

Monitor

- Components
  - Variables – monitor state
  - Procedures
    - Only access local variables or formal parameters
  - Condition variables
    - Tailor-made sync
    - x.wait() or x.signal




```
monitor name {
  variable declaration
  void proc1(…) {
  }
  …
  void procn(…) {
  }
}
```

Monitor

- Semantics of signal & wait
  - x.signal() resumes one suspended process. If there is none, no effect is imposed.
  - *P* x.signal() a suspended process *Q*
    - *P* either waits until *Q* leaves the monitor or waits for another condition
    - *Q* either waits until *P* leaves the monitor, or waits for another condition.

Monitor – Dining-Philosophers

```
monitor dp {
    enum {thinking, hungry, eating} state[5];
    condition self[5];
    void pickup(int i) {
            stat[i]=hungry;
            test(i);
            if (stat[i] != eating)
                self[i].wait;
    }
    void putdown(int i) {
            stat[i] = thinking;
            test((i+4) % 5);
            test((i + 1) % 5);
    }
```

```
Pi:
   dp.pickup(i);
   … eat …
   dp.putdown(i);
```

Monitor – Dining-Philosophers

```
void test(int i) {
    if (stat[(i+4) % 5]) != eating &&
        stat[i] == hungry &&
        state[(i+1) % 5] != eating) {
            stat[i] = eating;
            self[i].signal();
    }
}
void init() {
    for (int i=0; i < 5; i++)
            state[i] = thinking;
}
```

No deadlock!
But starvation could occur!

Monitor – Implementation by Semaphores

- Semaphores
  - *mutex* – to protect the monitor
  - *next* – being initialized to zero, on which processes may suspend themselves
    - *nextcount*
- For each external function *F*

```
wait(mutex);
…
body of F;
…
if (next-count > 0)
   signal(next);
else signal(mutex);
```

Monitor – Implementation by Semaphores

- For every condition *x*
  - A semaphore *x-sem*
  - An integer variable *x-count*
  - Implementation of x.wait() and x.signal :

- x.wait()

```
x-count++;
if (next-count > 0)
     signal(next);
else signal(mutex);
wait(x-sem);
x-count--;
```

- x.signal

```
if (x-count > 0) {
          next-count++;
          signal(x-sem);
          wait(next);
          next-count--;
}
```


* x.wait() and x.signal() are invoked within a monitor.

Monitor

- Process-Resumption Order
  - Queuing mechanisms for a monitor and its condition variables.
  - A solution:

    x.wait(c);

    - where the expression *c* is evaluated to determine its process's resumption order.

```
R.acquire(t);
 …
access the resource;
R.release;
```

```
monitor ResAllc {
boolean busy;
condition x;
void acquire(int time) {
        if (busy)
                x.wait(time);
        busy=true;
}
…
}
```

Monitor

- Concerns:
  - Processes may access resources without consulting the monitor.
  - Processes may never release resources.
  - Processes may release resources which they never requested.
  - Process may even request resources twice.

Monitor

- Remark: Whether the monitor is correctly used?

  => Requirements for correct computations

  - Processes always make their calls on the monitor in correct order.
  - No uncooperative process can access resource directly without using the access protocols.
- Note: Scheduling behavior should consult the built-in monitor scheduling algorithm if resource access RPC are built inside the monitor.

OS Synchronization – Solaris 2

- Semaphores and Condition Variables
- Adaptive Mutex
  - Spin-locking if the lock-holding thread is running; otherwise, blocking is used.
- Readers-Writers Locks
  - Expensive in implementations.
- Turnstile
  - A queue structure containing threads blocked on a lock.
  - Priority inversion → priority inheritance protocol for kernel threads

OS Synchronization – Windows 2000

- General Mechanism
  - Spin-locking for short code segments in a multiprocessor platform.
  - Interrupt disabling when access to global variables is done in a uniprocessor platform.
- Dispatcher Object
  - State: signaled or non-signaled
  - *Mutex* – select one process from its waiting queue to the ready queue.
  - *Events* – select all processes waiting for the event.

Atomic Transactions

- Why Atomic Transactions?
  - Critical sections ensure mutual exclusion in data sharing, but the relationship between critical sections might also be meaningful!

  → Atomic Transactions

- Operating systems can be viewed as manipulators of data!

Atomic Transactions – System Model

- Transaction – a logical unit of computation
  - A sequence of read and write operations followed by a commit or an abort.
- Beyond "critical sections"
  1. Atomicity: All or Nothing
     - An aborted transaction must be *rolled back*.
     - The effect of a committed transaction must persist and be imposed as a logical unit of operations.

Atomic Transactions – System Model

2. Serializability:

- The order of transaction executions must be equivalent to a serial schedule.

T0	T1
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

Two operations O_i & O_j conflict if

1. Access the same object
2. One of them is write

Atomic Transactions – System Model

- Conflict Serializable:
  - *S* is conflict serializable if *S* can be transformed into a serial schedule by swapping nonconflicting operations.

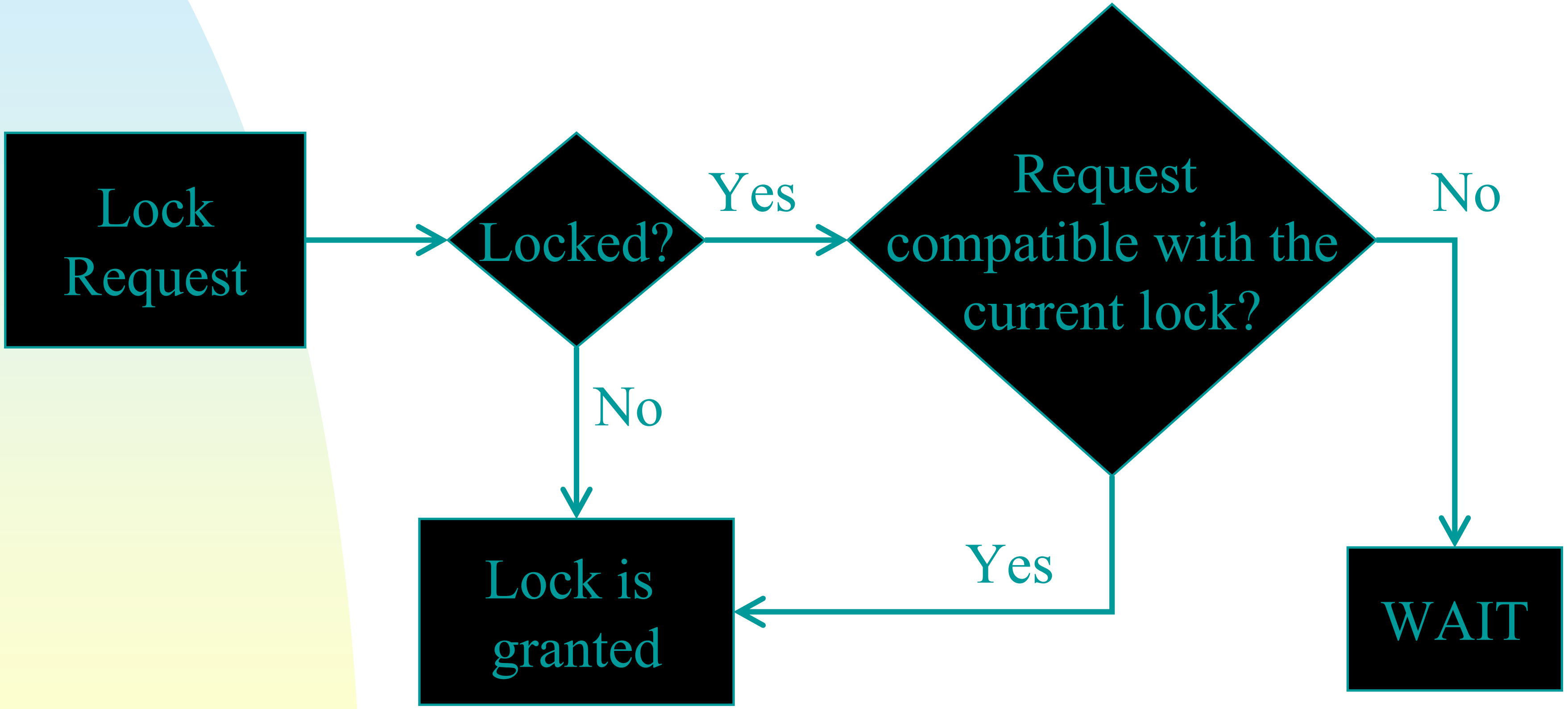
T0	T1
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

T0	T1
R(A)	
W(A)	
R(B)	
W(B)	
	R(A)
	W(A)
	R(B)
	W(B)

Atomic Transactions – Concurrency Control

- Locking Protocols
  - Lock modes (A general approach!)
    - 1. Shared-Mode: “Reads”.
    - 2. Exclusive-Mode: “Reads” & “Writes“
  - General Rule
    - A transaction must receive a lock of an appropriate mode of an object before it accesses the object. The lock may not be released until the last access of the object is done.

Atomic Transactions – Concurrency Control

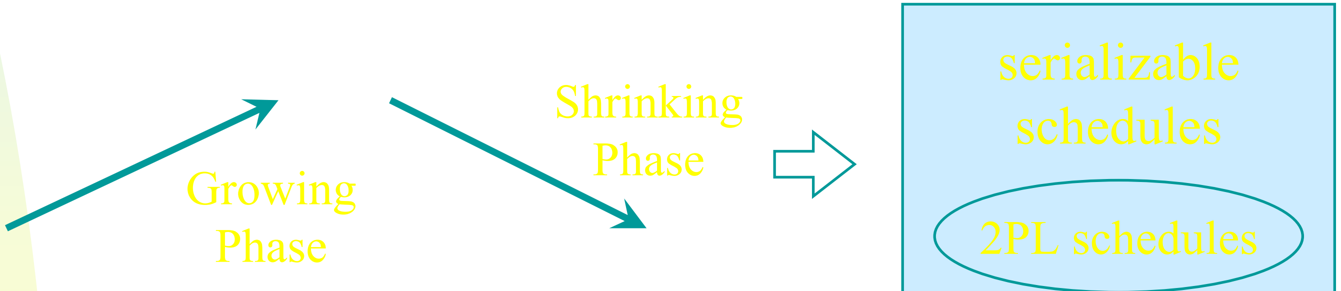


Atomic Transactions – Concurrency Control

- When to release locks w/o violating serializability

  R0(A) W0(A) R1(A) R1(B) R0(B) W0(B)

- Two-Phase Locking Protocol (2PL) – Not Deadlock-Free



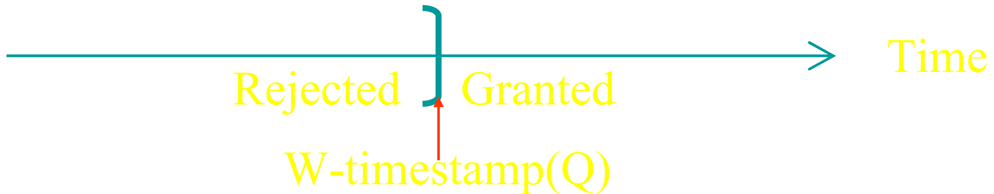
- How to improve 2PL?
  - Semantics, Order of Data, Access Pattern, etc.

Atomic Transactions – Concurrency Control

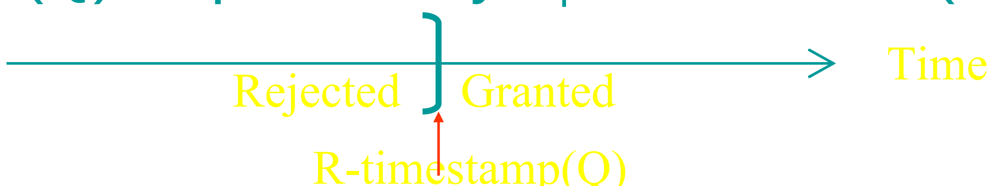
- Timestamp-Based Protocols
  - A time stamp for each transaction $TS(T_i)$
    - Determine transactions' order in a schedule in advance!
  - A General Approach:
    - $TS(T_i)$ – System Clock or Logical Counter
      - Unique?
    - Scheduling Scheme – deadlock-free & serializable
      - $W-timestamp(Q) = Max_{T_i - W(Q)}(TS(T_i))$
      - $R-timestamp(Q) = Max_{T_i - R(Q)}(TS(T_i))$

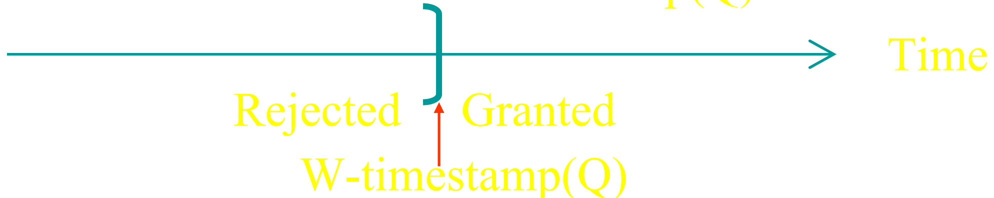
Atomic Transactions – Concurrency Control

- R(Q) requested by T_i → check $TS(T_i)$!



- W(Q) requested by T_i → check $TS(T_i)$!





- Rejected transactions are rolled back and restated with a new time stamp.

Failure Recovery – A Way to Achieve Atomicity

- Failures of Volatile and Nonvolatile Storages!
  - Volatile Storage: Memory and Cache
  - Nonvolatile Storage: Disks, Magnetic Tape, etc.
  - Stable Storage: Storage which never fail.
- Log-Based Recovery
  - Write-Ahead Logging
    - Log Records

< Ti starts >

< Ti commits >

< Ti aborts >

< Ti, Data-Item-Name, Old-Value, New-Value>

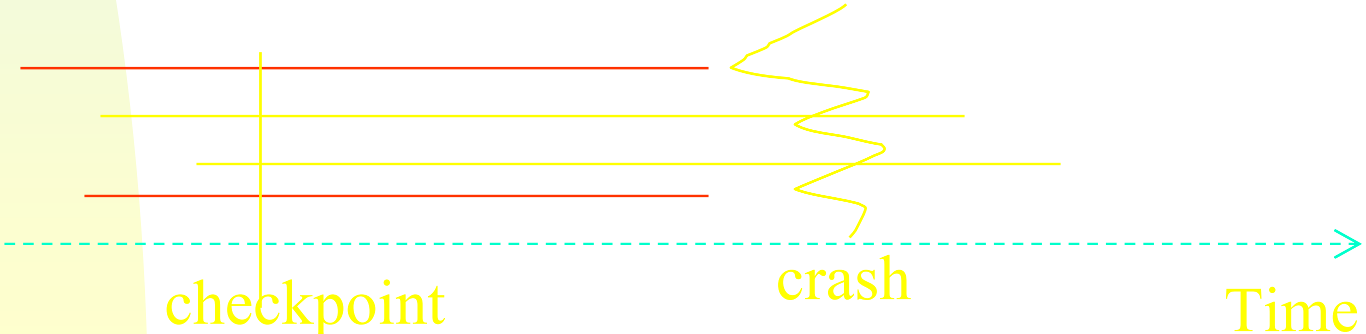
Failure Recovery

- Two Basic Recovery Procedures:

  - undo(Ti): restore data updated by Ti
  - redo(Ti): reset data updated by Ti
- Operations must be idempotent!
- Recover the system when a failure occurs:
  - "Redo" committed transactions, and "undo" aborted transactions.

Failure Recovery

- Why Checkpointing?
  - The needs to scan and rerun all log entries to redo committed transactions.
- CheckPoint
  - Output all log records, Output DB, and Write <check point> to stable storage!
  - Commit: A Force Write Procedure



Contents

Chapter 8 Deadlocks

Deadlocks

- A set of process is in a *deadlock* state when every process in the set is waiting for an event that can be caused by only another process in the set.
- A System Model
  - Competing processes – distributed?
  - Resources:
    - Physical Resources, e.g., CPU, printers, memory, etc.
    - Logical Resources, e.g., files, semaphores, etc.

Deadlocks

- A Normal Sequence
  1. Request: Granted or Rejected
  2. Use
  3. Release
- Remarks
  - No request should exceed the system capacity!
  - Deadlock can involve different resource types!
    - Several instances of the same type!

Deadlock Characterization

- Necessary Conditions

(deadlock → conditions or ¬ conditions → ¬ deadlock)

1. Mutual Exclusion – At least one resource must be held in a non-sharable mode!
2. Hold and Wait – Pi is holding at least one resource and waiting to acquire additional resources that are currently held by other processes!

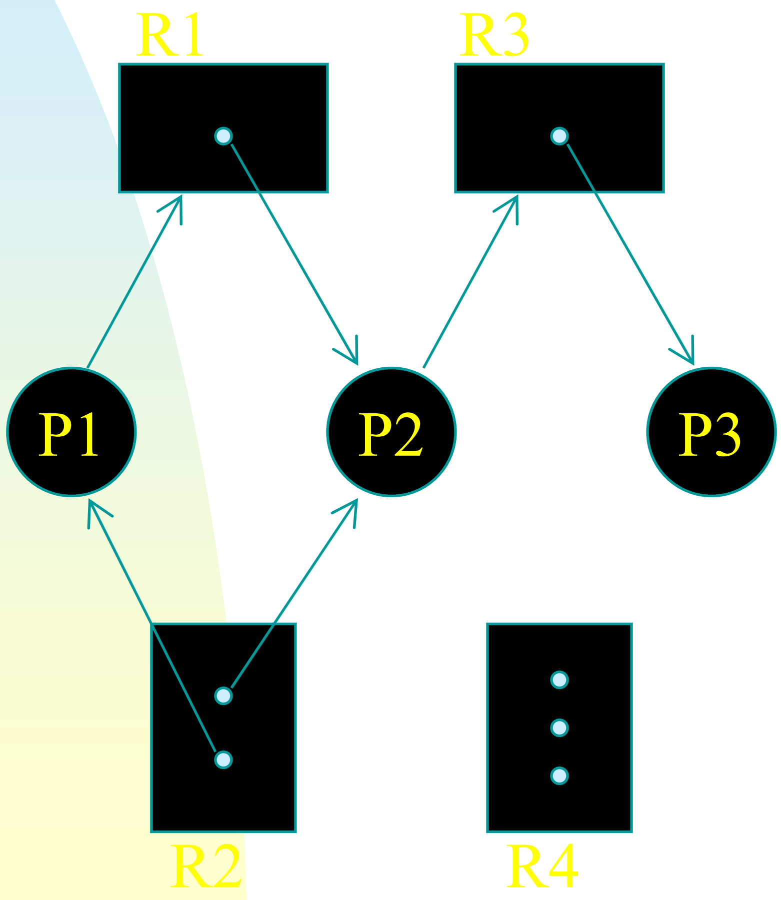
Deadlock Characterization

3. No Preemption – Resources are nonpreemptible!
4. Circular Wait – There exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting process such that $P_0 \xrightarrow[\text{wait}]{} P_1$, $P_1 \xrightarrow[\text{wait}]{} P_2$, …, $P_{n-1} \xrightarrow[\text{wait}]{} P_n$, and $P_n \xrightarrow[\text{wait}]{} P_0$.

- Remark:
  - Condition 4 implies Condition 2.
  - The four conditions are not completely independent!

Resource Allocation Graph

System Resource-Allocation Graph



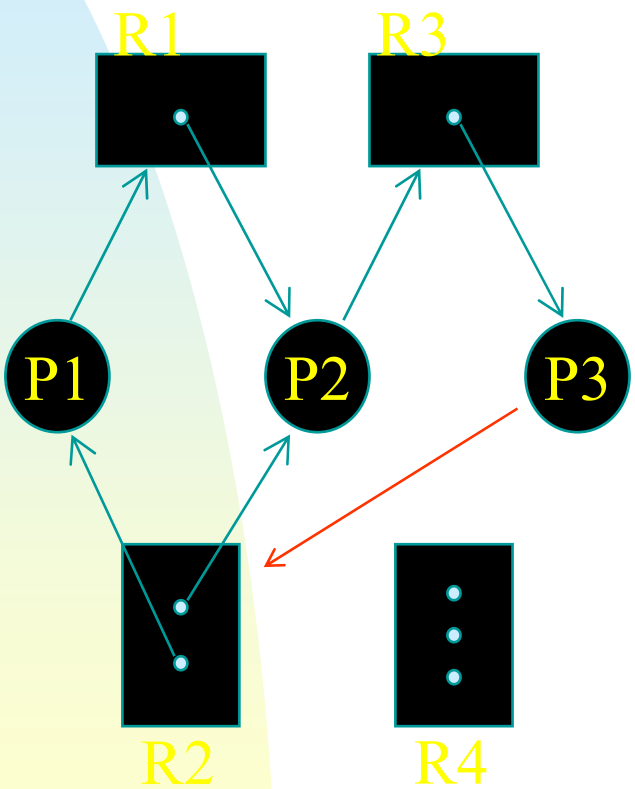
Vertices

- Processes:
  {P1,…, Pn}
- Resource Type :
  {R1,…, Rm}

Edges

- Request Edge:
  Pi → Rj
- Assignment Edge:
  Ri → Pj

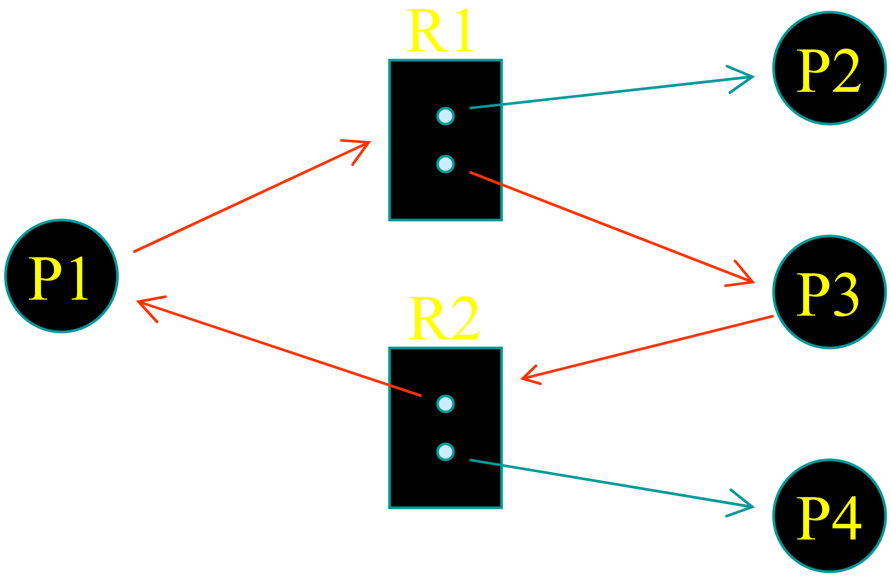
Resource Allocation Graph



- Example
  - No-Deadlock
    - Vertices
      - P = { P1, P2, P3 }
      - R = { R1, R2, R3, R4 }
    - Edges
      - E = { P1→R1, P2→R3, R1→P2, R2→P2, R2→P1, R3→P3 }
  - Resources
    - R1:1, R2:2, R3:1, R4:3
  - → results in a deadlock.

Resource Allocation Graph

- Observation
  - The existence of a cycle
    - One Instance per Resource Type → Yes!!
    - Otherwise → Only A Necessary Condition!!



Methods for Handling Deadlocks

- Solutions:

1. Make sure that the system never enters a deadlock state!
    - Deadlock Prevention: Fail at least one of the necessary conditions
    - Deadlock Avoidance: Processes provide information regarding their resource usage. Make sure that the system always stays at a “safe” state!

Methods for Handling Deadlocks

2. Do recovery if the system is deadlocked.
   - Deadlock Detection
   - Recovery
3. Ignore the possibility of deadlock occurrences!
   - Restart the system “manually” if the system “seems” to be deadlocked or stops functioning.
   - Note that the system may be “frozen” temporarily!

Deadlock Prevention

- Observation:
  - Try to fail anyone of the necessary condition!

    $\because \neg (\wedge \text{ i-th condition}) \rightarrow \neg \text{ deadlock}$

- Mutual Exclusion

  ?? Some resources, such as a printer, are intrinsically non-sharable??

Deadlock Prevention

- Hold and Wait
  - Acquire all needed resources before its execution.
  - Release allocated resources before request additional resources!

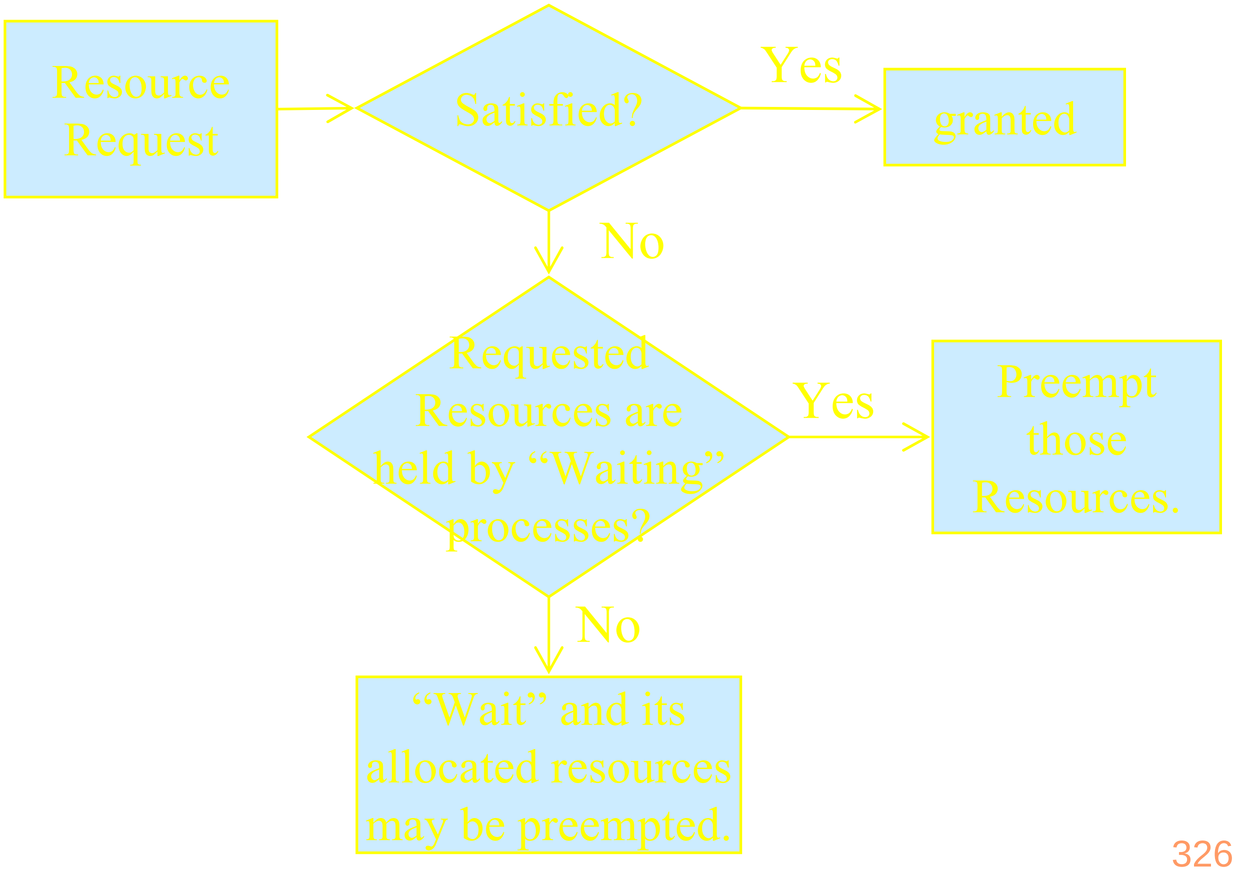
  - Disadvantage:
    - Low Resource Utilization
    - Starvation

Deadlock Prevention

- No Preemption
  - Resource preemption causes the release of resources.
  - Related protocols are only applied to resources whose states can be saved and restored, e.g., CPU register & memory space, instead of printers or tape drives.
  - Approach 1:

Deadlock Prevention

- Approach 2



Deadlock Prevention

- Circular Wait

  A resource-ordering approach:

  F : R → N
  Resource requests must be made in an increasing order of enumeration.

  - Type 1 – strictly increasing order of resource requests.
    - Initially, order any # of instances of Ri
    - Following requests of any # of instances of Rj must satisfy F(Rj) > F(Ri), and so on.

    * A single request must be issued for all needed instances of the same resources.

Deadlock Prevention

- Type 2
  - Processes must release all Ri's when they request any instance of Rj if $F(Ri) \geq F(Rj)$
- $F : R \rightarrow N$ must be defined according to the normal order of resource usages in a system, e.g.,

$$\left.\begin{array}{ll} F(\text{tape drive}) & = 1 \\ F(\text{disk drive}) & = 5 \\ F(\text{printer}) & = 12 \end{array}\right\} \text{?? feasible ??}$$

Deadlock Avoidance

- Motivation:
  - Deadlock-prevention algorithms can cause low device utilization and reduced system throughput!

➔ Acquire additional information about how resources are to be requested and have better resource allocation!

  - Processes declare their maximum number of resources of each type that it may need.

Deadlock Avoidance

- A Simple Model
  - A resource-allocation state

    <# of available resources,

    # of allocated resources,

    max demands of processes>
- A deadlock-avoidance algorithm dynamically examines the resource-allocation state and make sure that it is safe.
  - e.g., the system never satisfies the circular-wait condition.

Deadlock Avoidance

- Safe Sequence
  - A sequence of processes <P1, P2, …, Pn> is a safe sequence if

$$\forall Pi, need(Pi) \le Available + \sum_{j<i} allocated(Pj)$$

- Safe State
  - The existence of a safe sequence
  - Unsafe

Deadlocks are avoided if the system can allocate resources to each process up to its maximum request in some order. If so, the system is in a safe state!

Deadlock Avoidance

- Example:

	max needs	Allocated	Available
P0	10	5	3
P1	4	2	
P2	9	2	

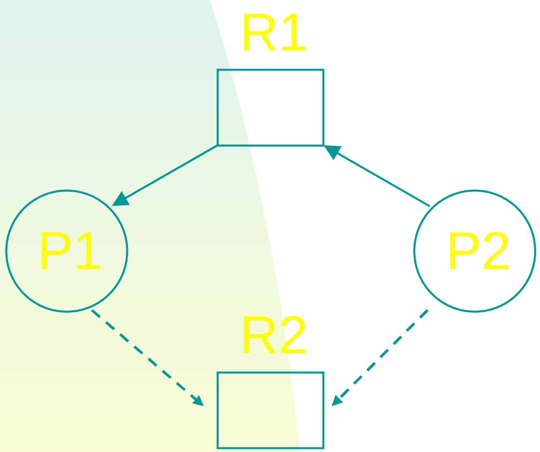
- The existence of a safe sequence <P1, P0, P2>.
- If P2 got one more, the system state is unsafe.

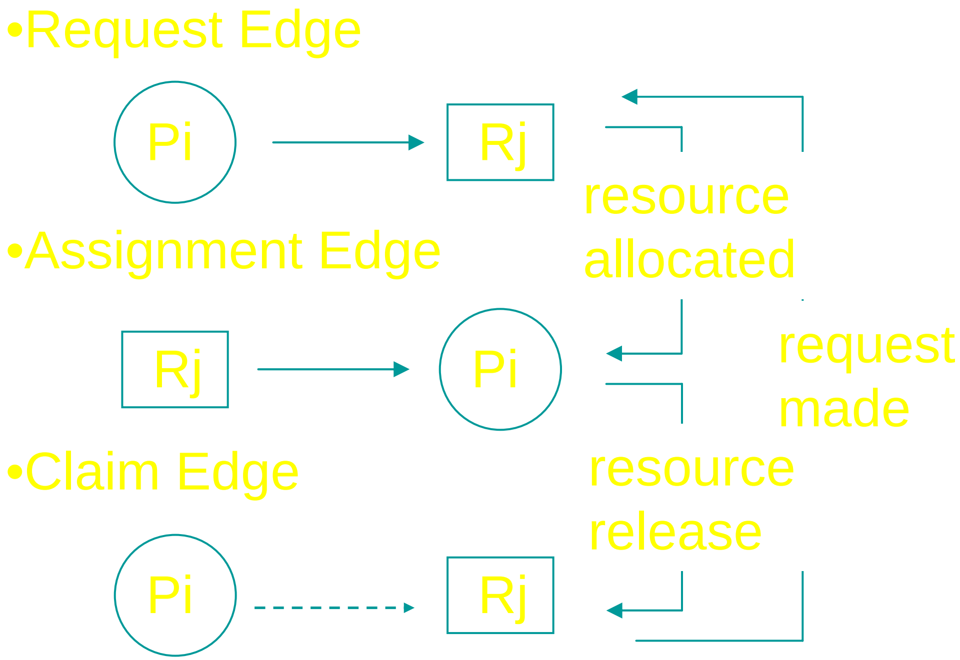
$$\because ((P0,5),(P1,2),(P2,3),(available,2))$$

How to ensure that the system will always remain in a safe state?

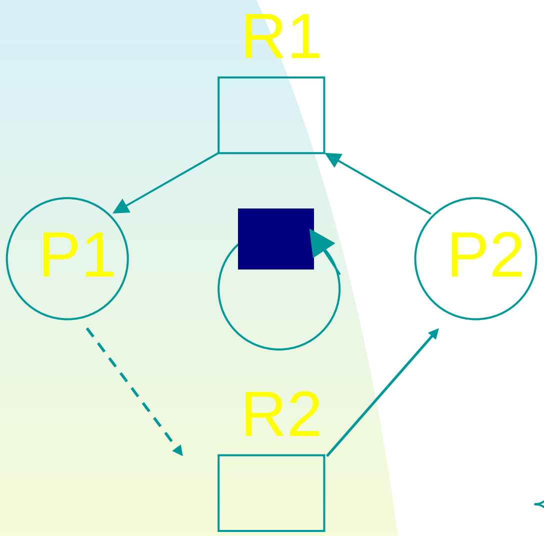
Deadlock Avoidance – Resource-Allocation Graph Algorithm

- One Instance per Resource Type





Deadlock Avoidance – Resource-Allocation Graph Algorithm



A cycle is detected!
→ The system state is unsafe!

- R2 was requested & granted!

$\left\{\begin{array}{l}\text{Safe state: no cycle} \\ \\ \text{Unsafe state: otherwise}\end{array}\right\}$ Cycle detection can be done in $O(n^2)$

Deadlock Avoidance – Banker's Algorithm

n: # of processes,
m: # of resource types

- Available [m]
  - If Available [i] = k, there are k instances of resource type Ri available.
- Max [n,m]
  - If Max [i,j] = k, process Pi may request at most k instances of resource type Rj.
- Allocation [n,m]
  - If Allocation [i,j] = k, process Pi is currently allocated k instances of resource type Rj.
- Need [n,m]
  - If Need [i,j] = k, process Pi may need k more instances of resource type Rj.

➢ Need [i,j] = Max [i,j] – Allocation [i,j]

Deadlock Avoidance – Banker's Algorithm

- Safety Algorithm – A state is safe??

n: # of processes,
m: # of resource types

1. Work := Available & Finish [i] := F, 1≦ i≦ n
2. Find an i such that both
   1. Finish [i] =F
   2. Need[i] ≦ Work

   If no such i exist, **then goto** Step4
3. Work := Work + Allocation[i]
   Finish [i] := T; **Goto** Step2
4. **If** Finish [i] = T for all *i*, **then** the system is in a safe state.

Where Allocation[i] and Need[i] are the *i*-th row of Allocation and Need, respectively, and
X≦ Y if X[*i*] ≦ Y[*i*] for all *i*,
X < Y if X ≦ Y and Y≠ X

Deadlock Avoidance – Banker's Algorithm

- Resource-Request Algorithm

Request$_i$ [j] =k: P$_i$ requests k instance of resource type Rj

1. If Request$_i$ $\leqq$ Need$_i$, then Goto Step2; otherwise, Trap
2. If Request$_i$ $\leqq$ Available, then Goto Step3; otherwise, Pi must wait.
3. Have the system pretend to have allocated resources to process P$_i$ by setting

   Available := Available – Request$_i$;

   Allocation$_i$:= Allocation$_i$ + Request$_i$;

   Need$_i$:= Need$_i$ – Request$_i$;

   Execute "Safety Algorithm". If the system state is safe, the request is granted; otherwise, Pi must wait, and the old resource-allocation state is restored!

Deadlock Avoidance

- An Example

	Allocation			Max			Need			Available		
	A	B	C	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	5	3	7	4	3	3	3	2
P1	2	0	0	3	2	2	1	2	2			
P2	3	0	2	9	0	2	6	0	0			
P3	2	1	1	2	2	2	0	1	1			
P4	0	0	2	4	3	3	4	3	1			

- A safe state
  - ∵ <P1,P3,P4,P2,P0> is a safe sequence.

Deadlock Avoidance

Let P1 make a request Request*i* = (1,0,2)

Request$_i$ ≦ Available ((1,0,2) ≦ (3,3,2))

	Allocation			Need			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	7	4	3	2	3	0
P1	3	0	2	0	2	0			
P2	3	0	2	6	0	0			
P3	2	1	1	0	1	1			
P4	0	0	2	4	3	1			

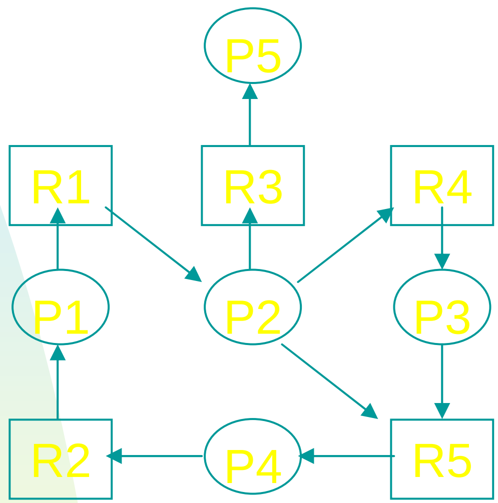
→ Safe ∵ <P1,P3,P4,P0,P2> is a safe sequence!

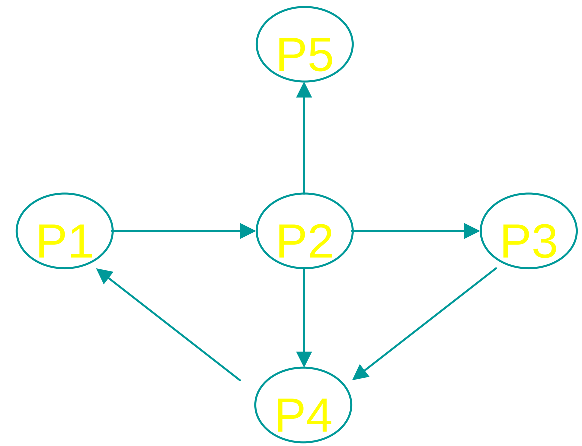
- If Request4 = (3,3,0) is asked later, it must be rejected.
- Request0 = (0,2,0) must be rejected because it results in an unsafe state.

Deadlock Detection

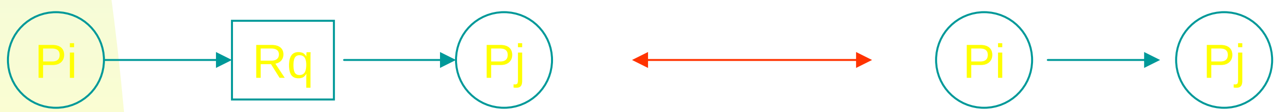
- Motivation:
  - Have high resource utilization and “maybe” a lower possibility of deadlock occurrence.
- Overheads:
  - Cost of information maintenance
  - Cost of executing a detection algorithm
  - Potential loss inherent from a deadlock recovery

Deadlock Detection – Single Instance per Resource Type





A Resource-Allocation Graph A Wait-For Graph



- Detect an cycle in $O(n^2)$.
- The system needs to maintain the wait-for graph

Deadlock Detection – Multiple Instance per Resource Type

- Data Structures
  - Available[1..m]: # of available resource instances
  - Allocation[1..n, 1..m]: current resource allocation to each process
  - Request[1..n, 1..m]: the current request of each process
    - If Request[i,j] = k, Pi requests k more instances of resource type Rj

n: # of processes, *m*: # of resource types

Deadlock Detection – Multiple Instance per Resource Type

Complexity = $O(m * n^2)$

1. Work := Available. For i = 1, 2, …, n, **if** Allocation[i] $\neq$ 0, **then** Finish[i] = F; **otherwise,** Finish[i] =T.
2. Find an i such that both
   a. Finish[i] = F
   b. Request[i] $\leq$ Work

   If no such i, **Goto** Step 4
3. Work := Work + Allocation[i]

   Finish[i] := T

   Goto Step 2
4. **If** Finish[i] = F for some i, **then** the system is in a deadlock state. **If** Finish[i] = F, then process Pi is deadlocked.

Deadlock Detection – Multiple Instance per Resource Type

- An Example

	Allocation			Request			Available		
	A	B	C	A	B	C	A	B	C
P0	0	1	0	0	0	0	0	2	0
P1	2	0	0	2	0	2			
P2	3	0	3	0	0	0			
P3	2	1	1	1	0	0			
P4	0	0	2	0	0	2			

➔ Find a sequence <P0, P2, P3, P1, P4> such that Finish[i] = T for all i.
If Request2 = (0,0,1) is issued, then P1, P2, P3, and P4 are deadlocked.

Deadlock Detection – Algorithm Usage

- When should we invoke the detection algorithm?
  - How often is a deadlock likely to occur?
  - How many processes will be affected by a deadlock?

Every rejected request + ← overheads → − ∞

− ← processes affected → +

- Time for Deadlock Detection?
  - CPU Threshold? Detection Frequency? …

Deadlock Recovery

- Whose responsibility to deal with deadlocks?
  - Operator deals with the deadlock manually.
  - The system recover from the deadlock automatically.
- Possible Solutions
  - Abort one or more processes to break the circular wait.
  - Preempt some resources from one or more deadlocked processes.

Deadlock Recovery – Process Termination

- Process Termination
  - Abort all deadlocked processes!
    - Simple but costly!
  - Abort one process at a time until the deadlock cycle is broken!
    - Overheads for running the detection again and again.
    - The difficulty in selecting a victim!

But, can we abort any process? Should we compensate any damage caused by aborting?

Deadlock Recovery – Process Termination

- What should be considered in choosing a victim?
  - Process priority
  - The CPU time consumed and to be consumed by a process.
  - The numbers and types of resources used and needed by a process
  - Process's characteristics such as "interactive or batch"
  - The number of processes needed to be aborted.

Deadlock Recovery – Resource Preemption

- Goal: Preempt some resources from processes from processes and give them to other processes until the deadlock cycle is broken!
- Issues
  - Selecting a victim:
    - It must be cost-effective!
  - Roll-Back
    - How far should we roll back a process whose resources were preempted?
  - Starvation
    - Will we keep picking up the same process as a victim?
    - How to control the # of rollbacks per process efficiently?

Deadlock Recovery – Combined Approaches

- Partition resources into classes that are hierarchically ordered.

  $\Rightarrow$ No deadlock involves more than one class

  - Handle deadlocks in each class independently

Deadlock Recovery – Combined Approaches

Examples:

- Internal Resources: Resources used by the system, e.g., PCB
  - $\rightarrow$ Prevention through resource ordering
- Central Memory: User Memory
  - $\rightarrow$ Prevention through resource preemption
- Job Resources: Assignable devices and files
  - $\rightarrow$ Avoidance $\because$ This info may be obtained!
- Swappable Space: Space for each user process on the backing store
  - $\rightarrow$ Pre-allocation $\because$ the maximum need is known!

Contents

Chapter 9
Memory Management

Memory Management

- Motivation
  - Keep several processes in memory to improve a system's performance
- Selection of different memory management methods
  - Application-dependent
  - Hardware-dependent
- Memory – A large array of words or bytes, each with its own address.
  - Memory is always too small!

Memory Management

- The Viewpoint of the Memory Unit
  - A stream of memory addresses!
- What should be done?
  - Which areas are free or used (by whom)
  - Decide which processes to get memory
  - Perform allocation and de-allocation
- Remark:
  - Interaction between CPU scheduling and memory allocation!

Background

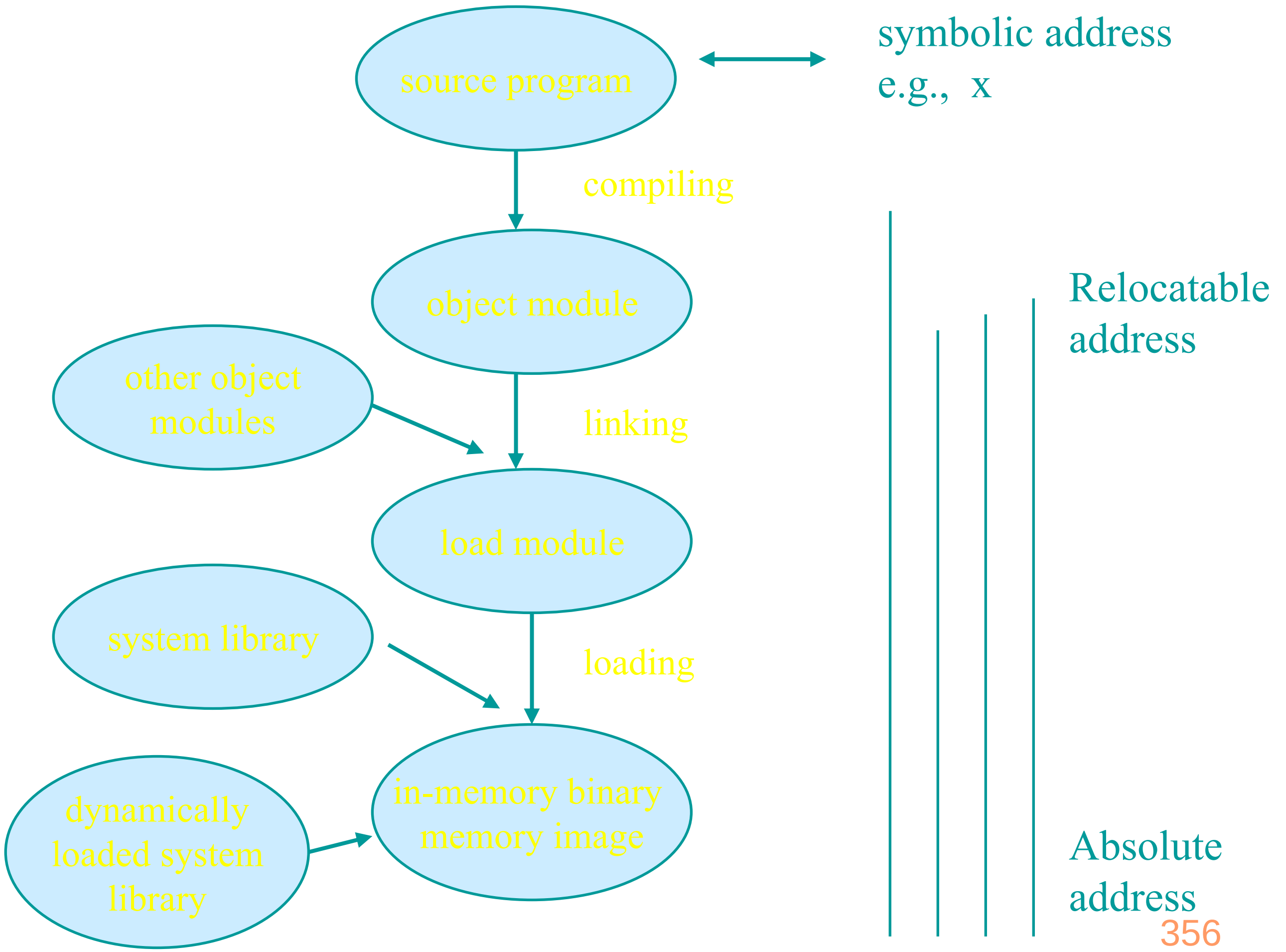
- Address Binding – binding of instructions and data to memory addresses

Binding Time

Known at compile time, where a program will be in memory - "absolute code" MS-DOS *.COM

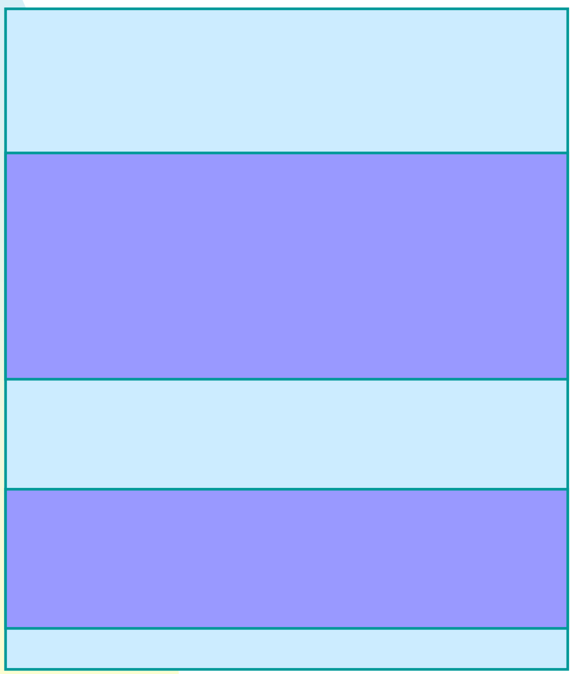
At load time:
- All memory reference by a program will be translated
- Code is relocatable
- Fixed while a program runs

At execution time
- binding may change as a program run



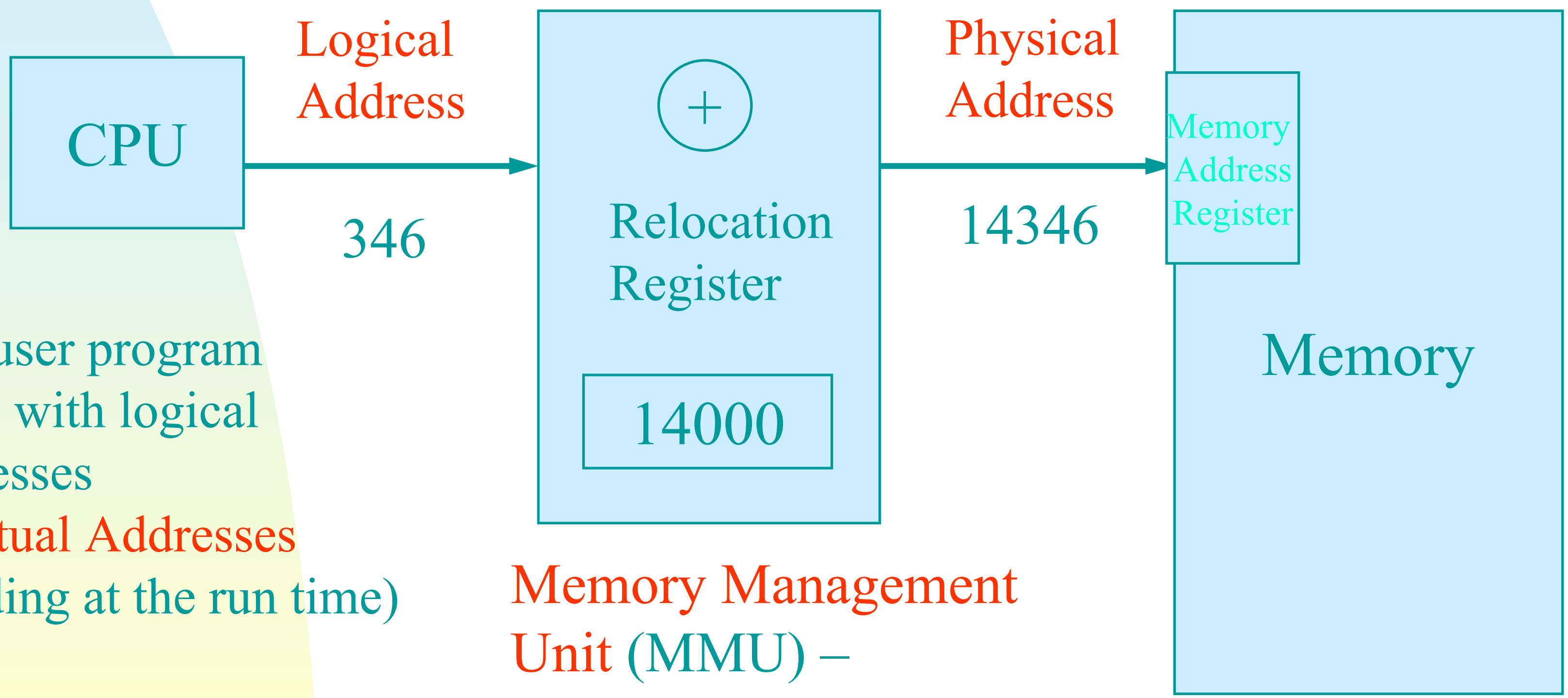
Background

Main Memory



- Binding at the Compiling Time
  - A process must execute at a specific memory space
- Binding at the Load Time
  - Relocatable Code
- Process may move from a memory segment to another → binding is delayed till run-time

Logical Versus Physical Address



The user program deals with logical addresses
- Virtual Addresses (binding at the run time)

Memory Management Unit (MMU) – "Hardware-Support"

Logical Versus Physical Address

- A logical (physical) address space is the set of logical (physical) addresses generated by a process. Physical addresses of a program is transparent to any process!
- MMU maps from virtual addresses to physical addresses. Different memory mapping schemes need different MMU's that are hardware devices. (slow down)
- Compile-time & load-time binding schemes results in the collapsing of logical and physical address spaces.

Dynamic Loading

- Dynamic Loading
  - A routine will not be loaded until it is called. A relocatable linking loader must be called to load the desired routine and change the program's address tables.
  - Advantage
    - Memory space is better utilized.
  - Users may use OS-provided libraries to achieve dynamic loading

Dynamic Linking

- Dynamic Linking ⟷ Static Linking

⇩

A small piece of code, called stub, is used to locate or load the appropriate routine

Advantage

Save memory space by sharing the library code among processes → Memory Protection & Library Update!

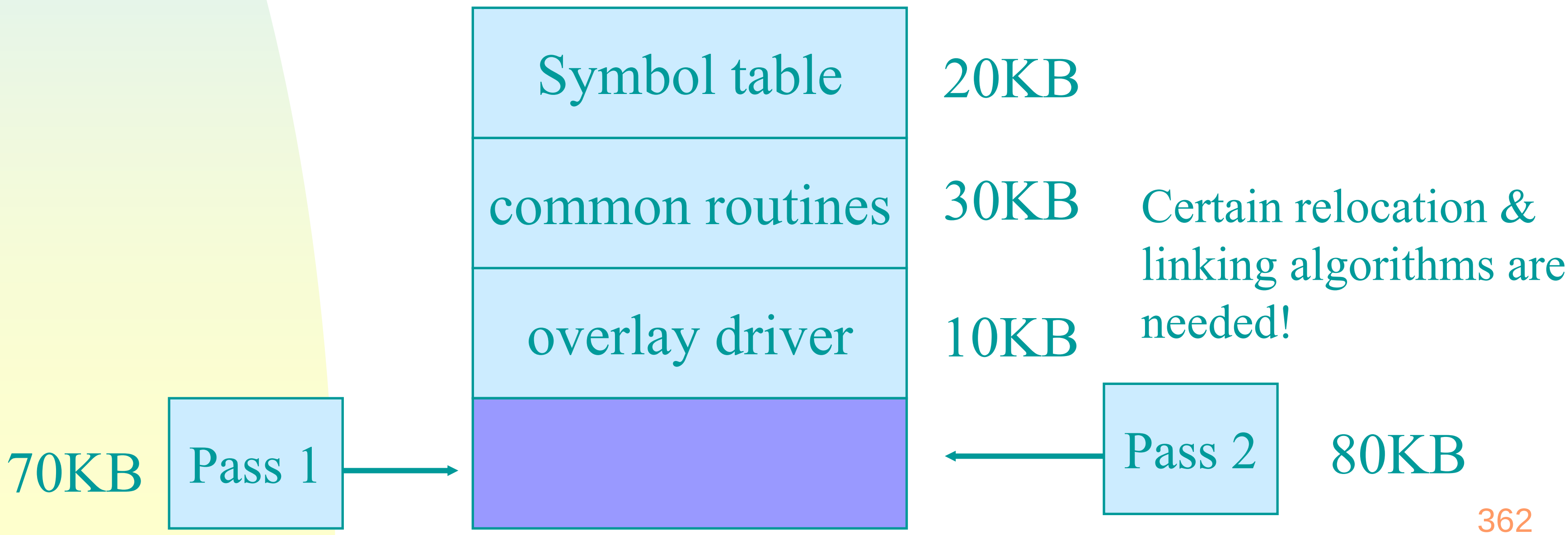
⇩

language library
+
program object module
↓
binary program image

Simple

Overlays

- Motivation
  - Keep in memory only those instructions and data needed at any given time.
  - Example: Two overlays of a two-pass assembler



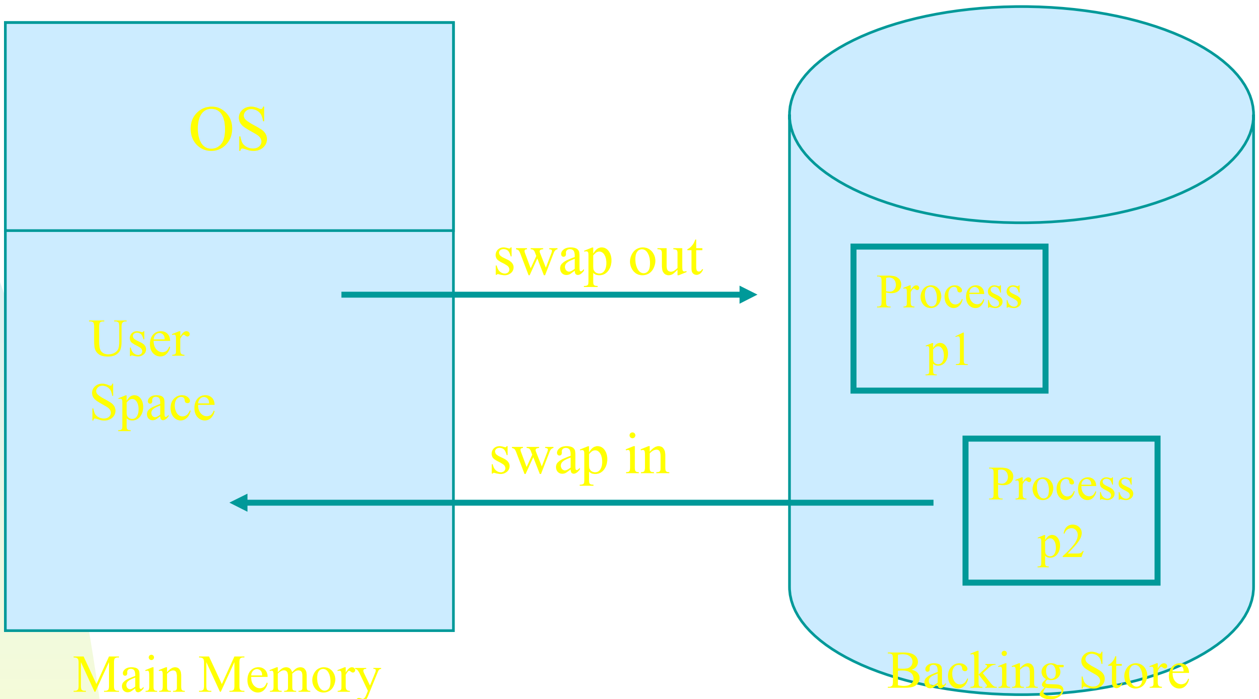
Overlays

- Memory space is saved at the cost of run-time I/O.
- Overlays can be achieved w/o OS support:

  $\Rightarrow$ “absolute-address” code
- However, it’s not easy to program a overlay structure properly!

  $\Rightarrow$ Need some sort of automatic techniques that run a large program in a limited physical memory!

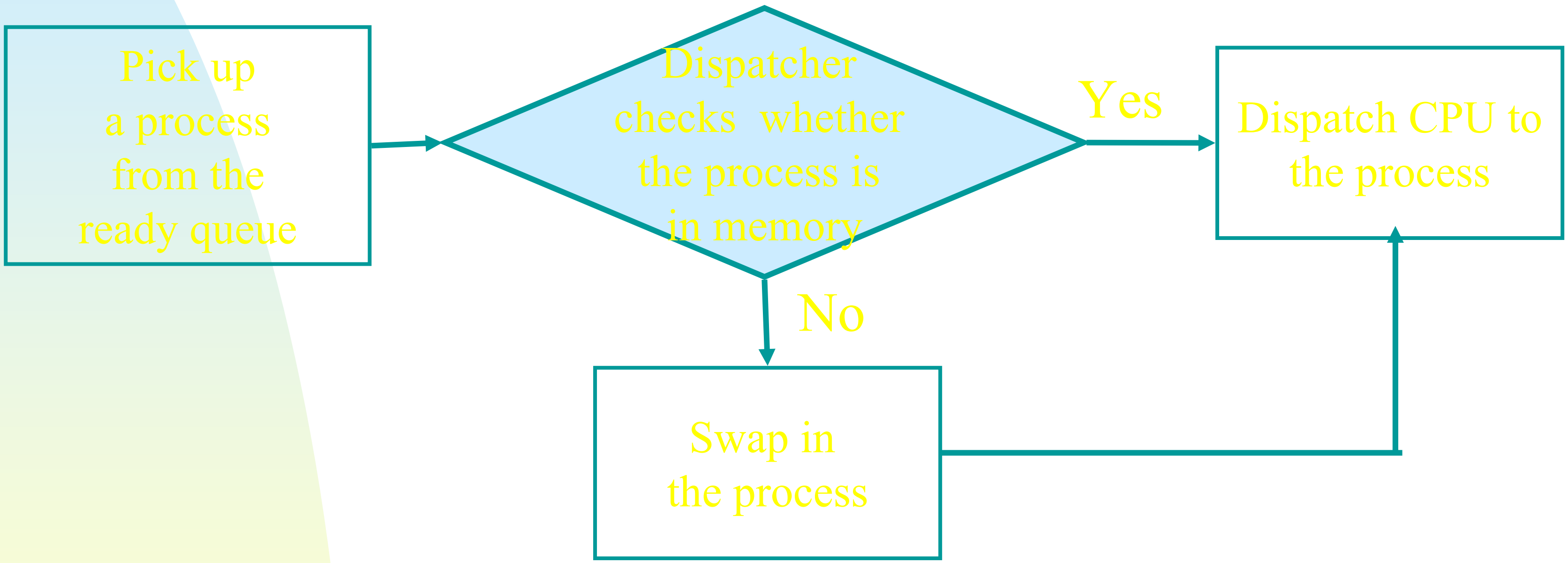
Swapping



Should a process be put back into the same memory space that it occupied previously?
↔ Binding Scheme?!

Swapping

- A Naive Way



Potentially High Context-Switch Cost:

2 * (1000KB/5000KBps + 8ms) = 416ms

Transfer Time Latency Delay

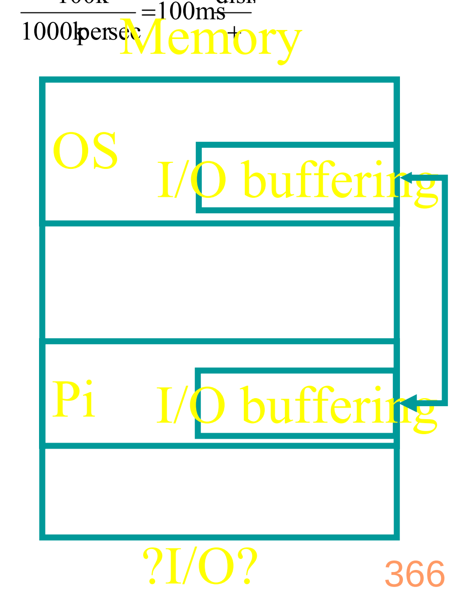
Swapping

- The execution time of each process should be long relative to the swapping time in this case (e.g., 416ms in the last example)!
- Only swap in what is actually used. ⇒ Users must keep the system informed of memory usage.

$$\frac{100k}{1000k \text{ per sec}} = 100ms \; \frac{disk}{+}$$

- Who should be swapped out?
  - “Lower Priority” Processes?
  - Any Constraint?

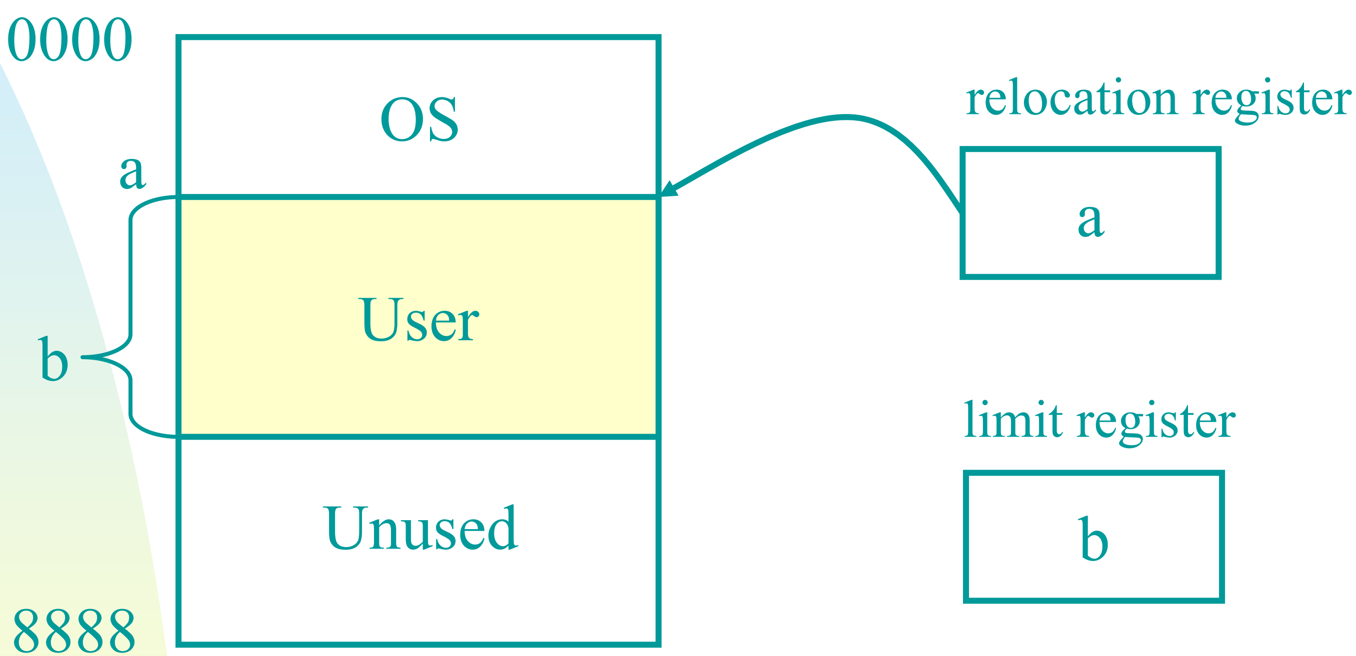
    ⇒ System Design



Swapping

- Separate swapping space from the file system for efficient usage
- Disable swapping whenever possible such as many versions of UNIX – Swapping is triggered only if the memory usage passes a threshold, and many processes are running!
- In Windows 3.1, a swapped-out process is not swapped in until the user selects the process to run.

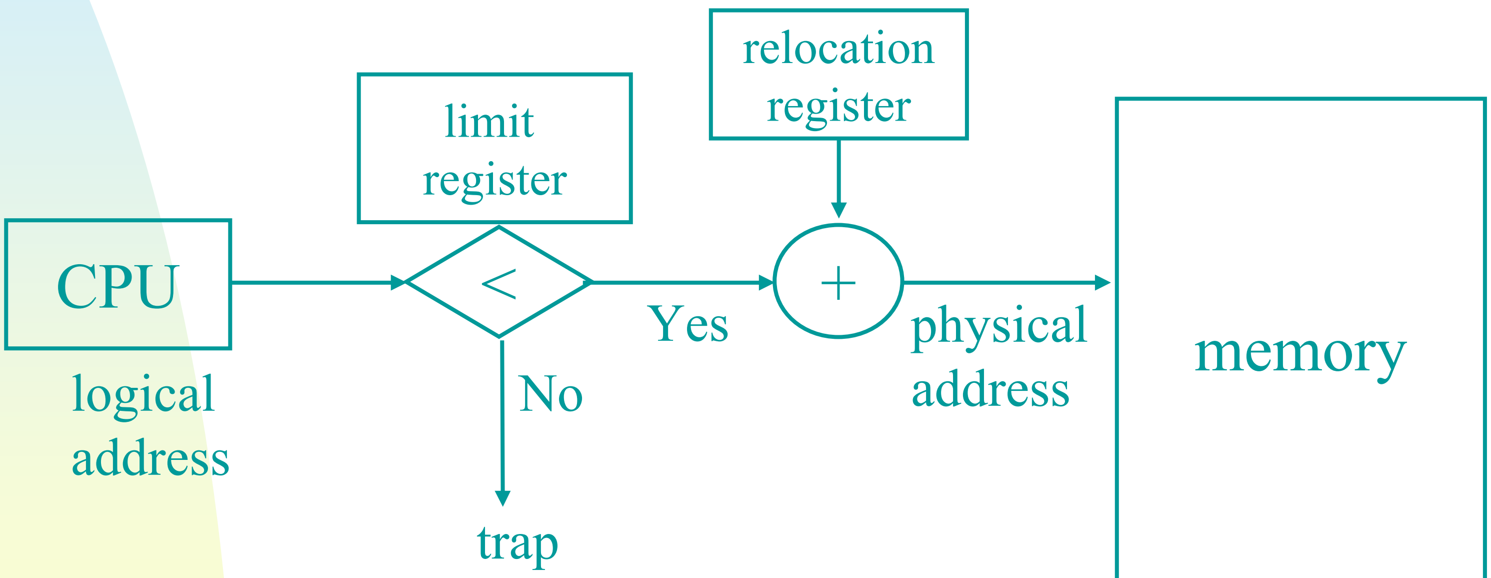
Contiguous Allocation – Single User



- A single user is allocated as much memory as needed
- Problem: Size Restriction → Overlays (MS/DOS)

Contiguous Allocation – Single User

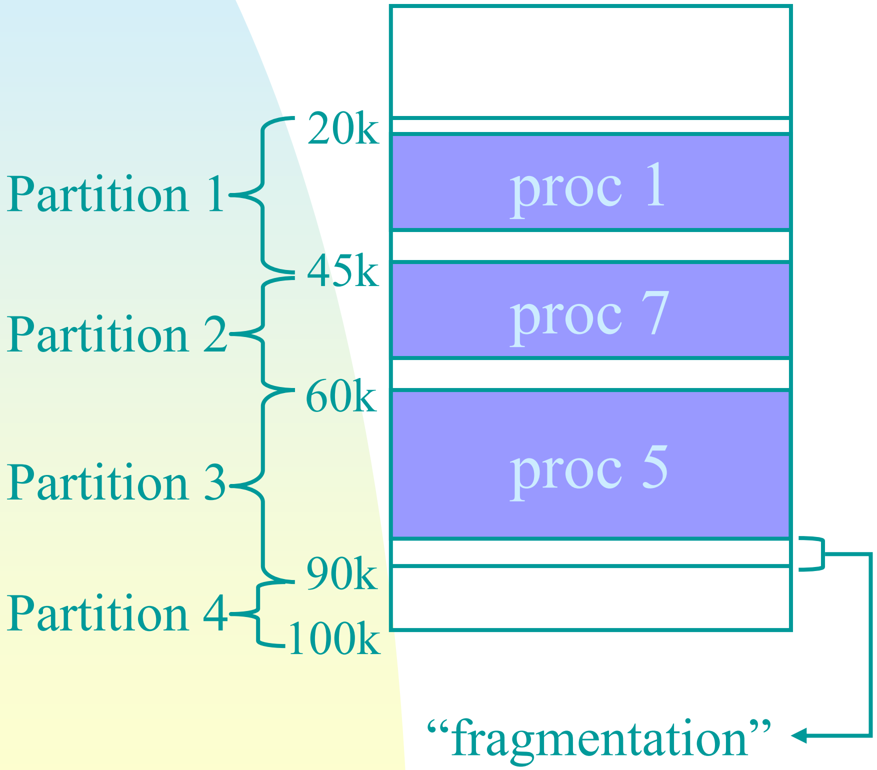
- Hardware Support for Memory Mapping and Protection



Disadvantage: Wasting of CPU and Resources
∵ No Multiprogramming Possible

Contiguous Allocation – Multiple Users

- Fixed Partitions



- Memory is divided into fixed partitions, e.g., OS/360 (or MFT)
- A process is allocated on an entire partition
- An OS Data Structure:

Partitions

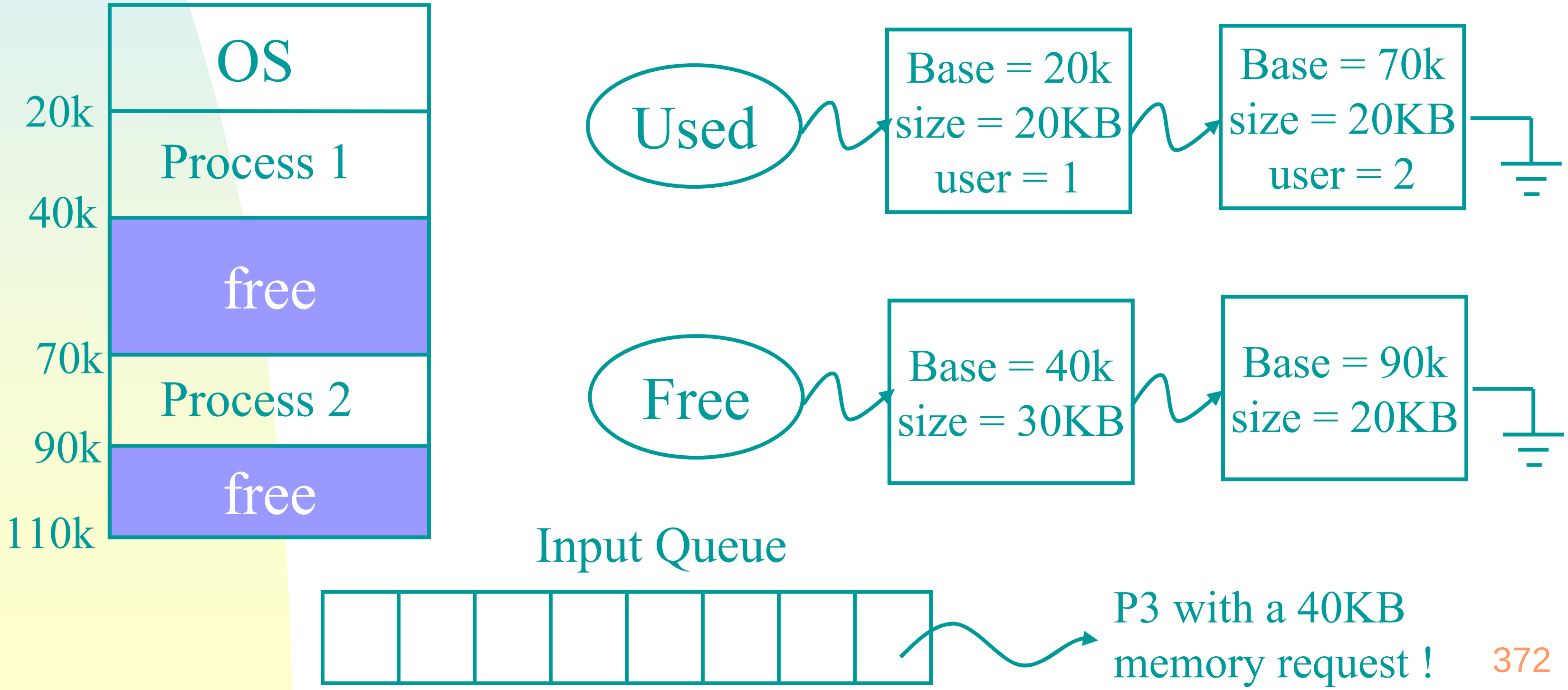
#	size	location	status
1	25KB	20k	Used
2	15KB	45k	Used
3	30KB	60k	Used
4	10KB	90k	Free

Contiguous Allocation – Multiple Users

- Hardware Supports
  - Bound registers
  - Each partition may have a protection key (corresponding to a key in the current PSW)
- Disadvantage:
  - Fragmentation gives poor memory utilization !

Contiguous Allocation – Multiple Users

- Dynamic Partitions
  - Partitions are dynamically created.
  - OS tables record free and used partitions



Contiguous Allocation – Multiple Users

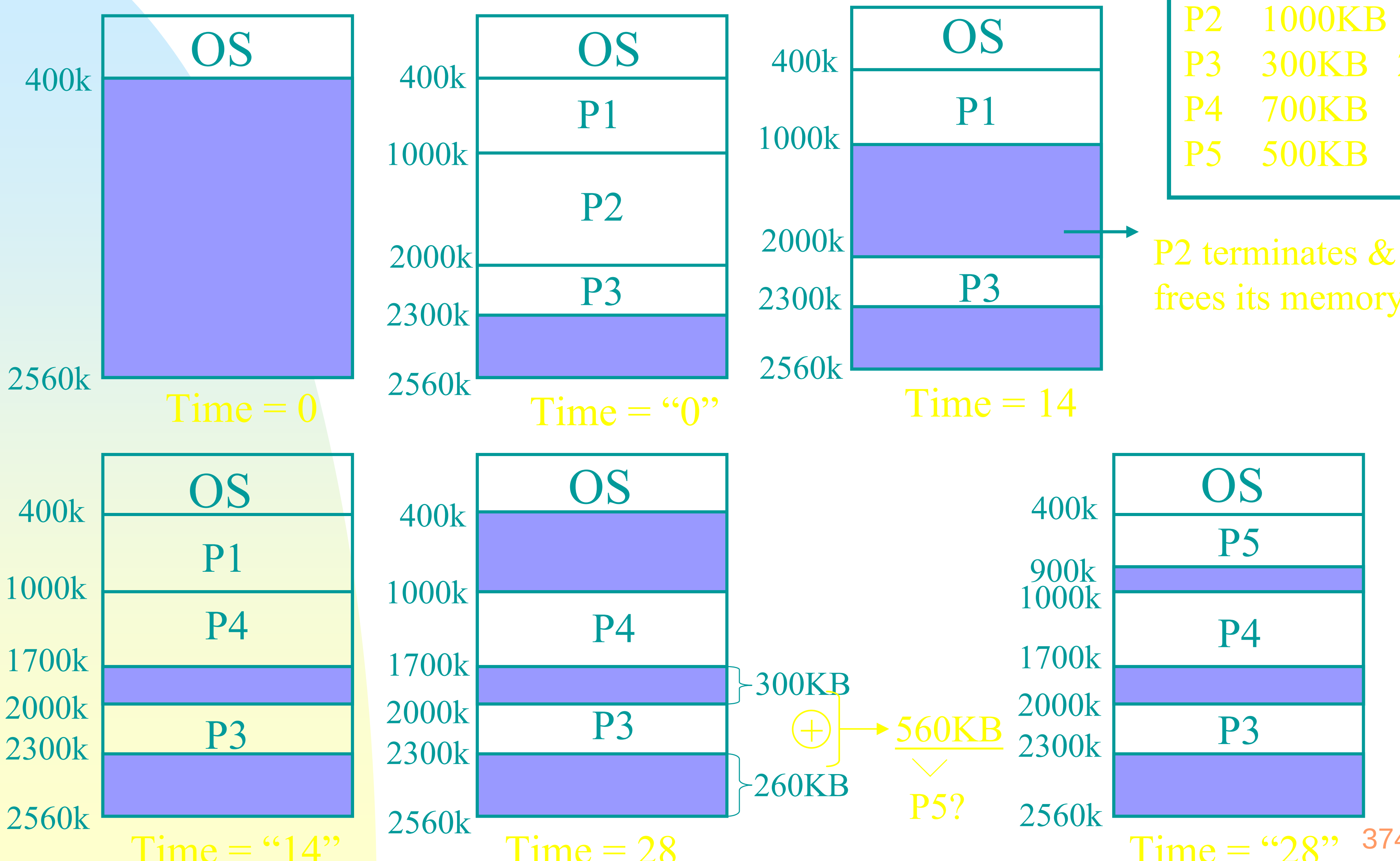
- Solutions for dynamic storage allocation :
  - First Fit – Find a hole which is big enough
    - Advantage: Fast and likely to have large chunks of memory in high memory locations
  - Best Fit – Find the smallest hole which is big enough. $\rightarrow$ It might need a lot of search time and create lots of small fragments !
    - Advantage: Large chunks of memory available
  - Worst Fit – Find the largest hole and create a new partition out of it!
    - Advantage: Having largest leftover holes with lots of search time!

Better in Time and Storage Usage

Contiguous Allocation Example – First Fit (RR Scheduler with Quantum = 1)

A job queue

Process	Memory	Time
P1	600KB	10
P2	1000KB	5
P3	300KB	20
P4	700KB	8
P5	500KB	15

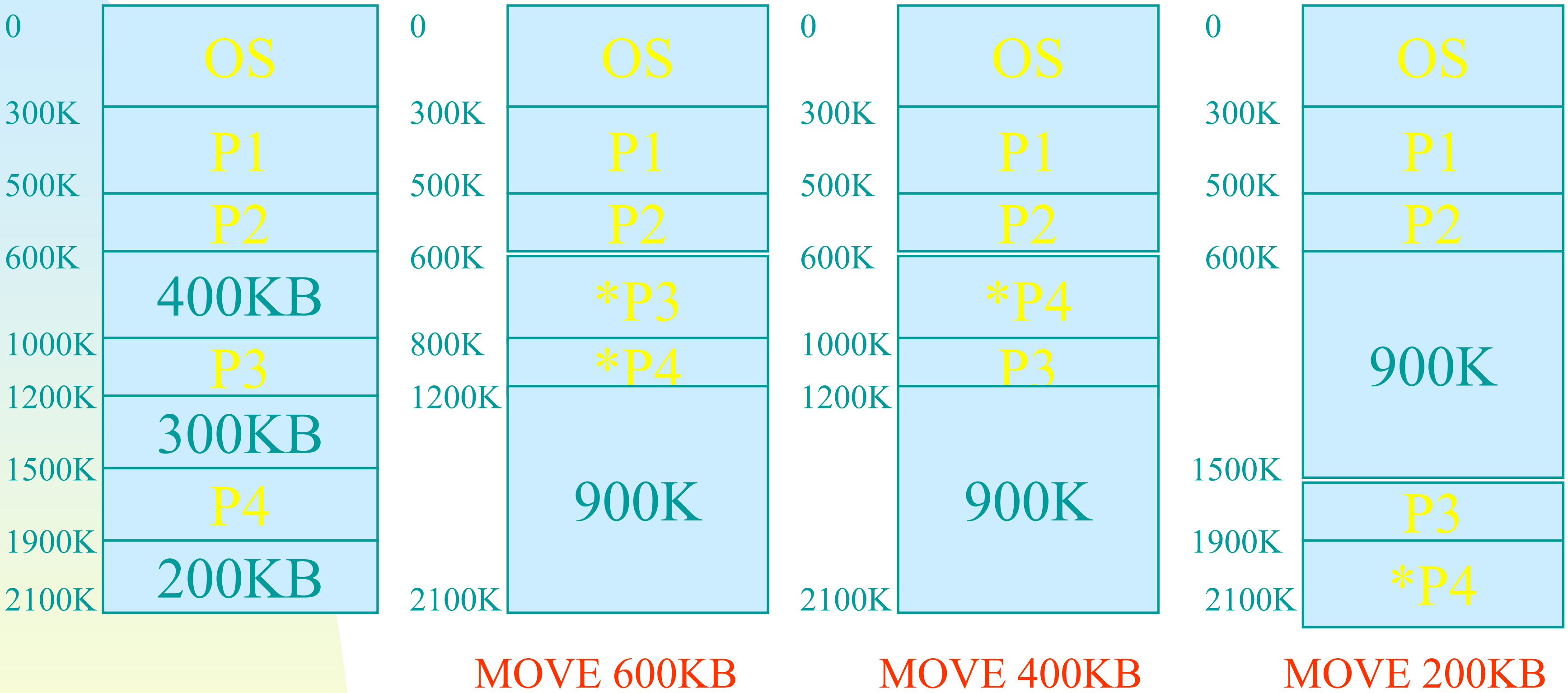


Fragmentation – Dynamic Partitions

- External fragmentation occurs as small chunks of memory accumulate as a by-product of partitioning due to imperfect fits.
  - Statistical Analysis For the First-Fit Algorithm:
    - 1/3 memory is unusable – 50-percent rule
  - Solutions:
    a. Merge adjacent free areas.
    b. Compaction
       - Compact all free areas into one contiguous region
       - Requires user processes to be relocatable

Any optimal compaction strategy???

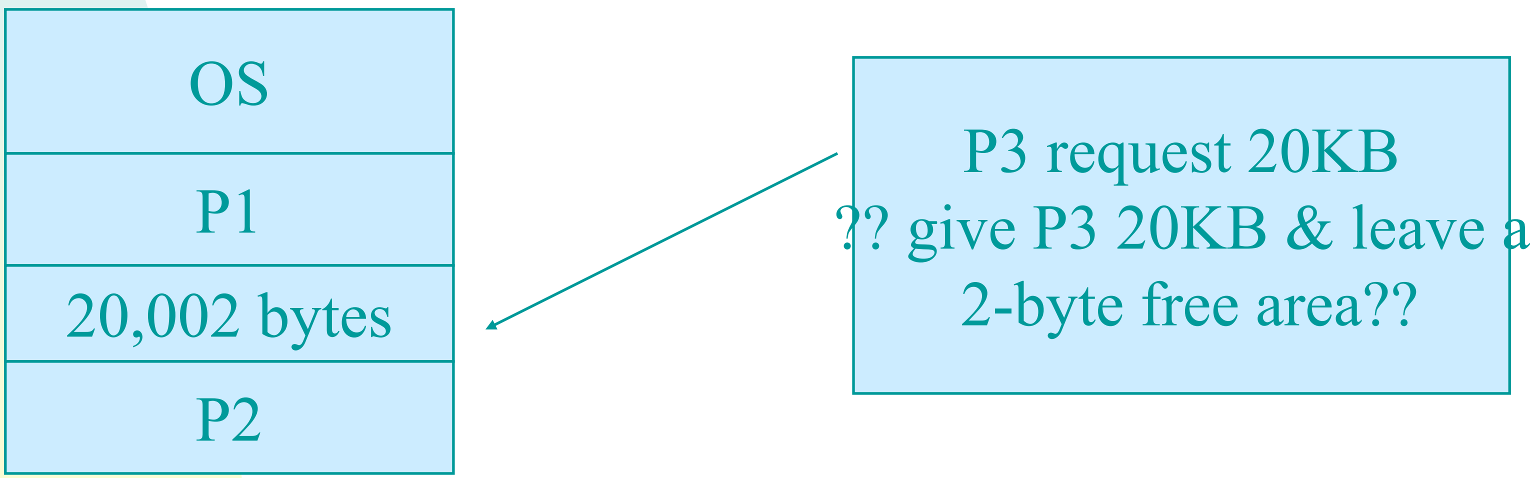
Fragmentation – Dynamic Partitions



- Cost: Time Complexity O(n!)?!!
- Combination of swapping and compaction
  - Dynamic/static relocation

Fragmentation – Dynamic Partitions

- Internal fragmentation:
  A small chunk of “unused” memory internal to a partition.



Reduce free-space maintenance cost

→ Give 20,002 bytes to P3 and have 2 bytes as an internal fragmentation!

Fragmentation – Dynamic Partitions

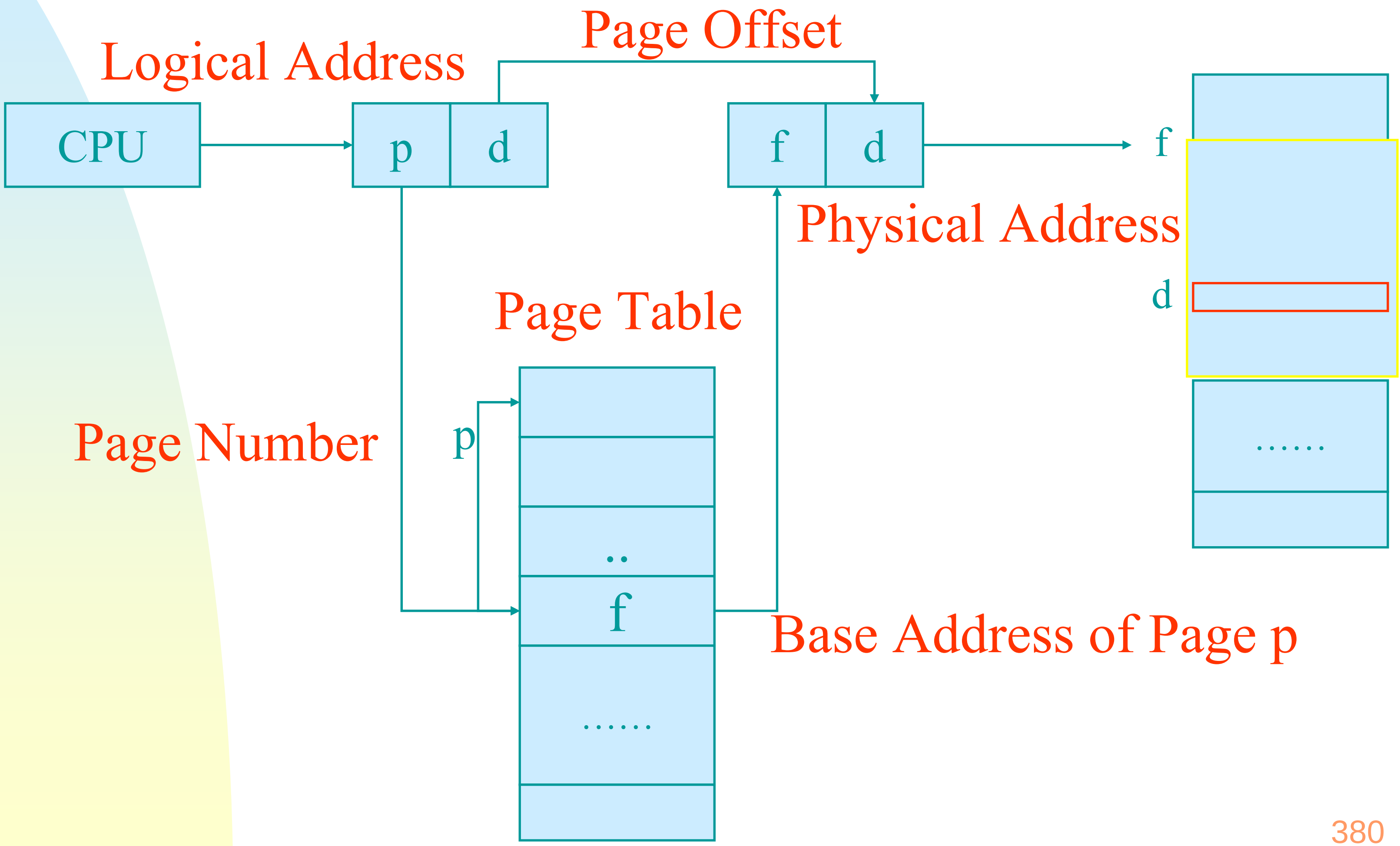
- Dynamic Partitioning:
  - Advantage:
    - ⇒ Eliminate fragmentation to some degree
    - ⇒ Can have more partitions and a higher degree of multiprogramming
  - Disadvantage:
    - Compaction vs Fragmentation
      - The amount of free memory may not be enough for a process! (contiguous allocation)
      - Memory locations may be allocated but never referenced.
    - Relocation Hardware Cost & Slow Down

⇒ Solution: Paged Memory!

Paging

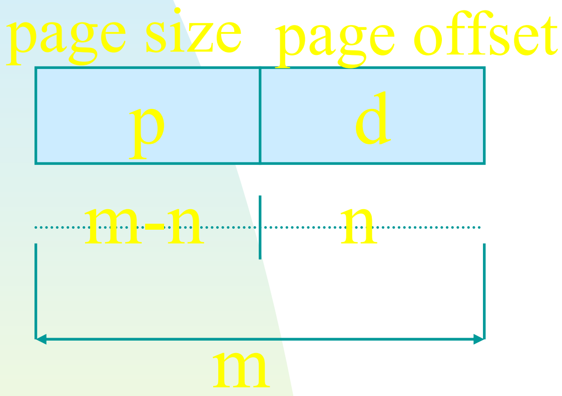
- Objective
  - Users see a logically contiguous address space although its physical addresses are throughout physical memory
- Units of Memory and Backing Store
  - Physical memory is divided into fixed-sized blocks called *frames.*
  - The logical memory space of each process is divided into blocks of the same size called *pages.*
  - The backing store is also divided into blocks of the same size if used.

Paging – Basic Method



Paging – Basic Method

- Address Translation



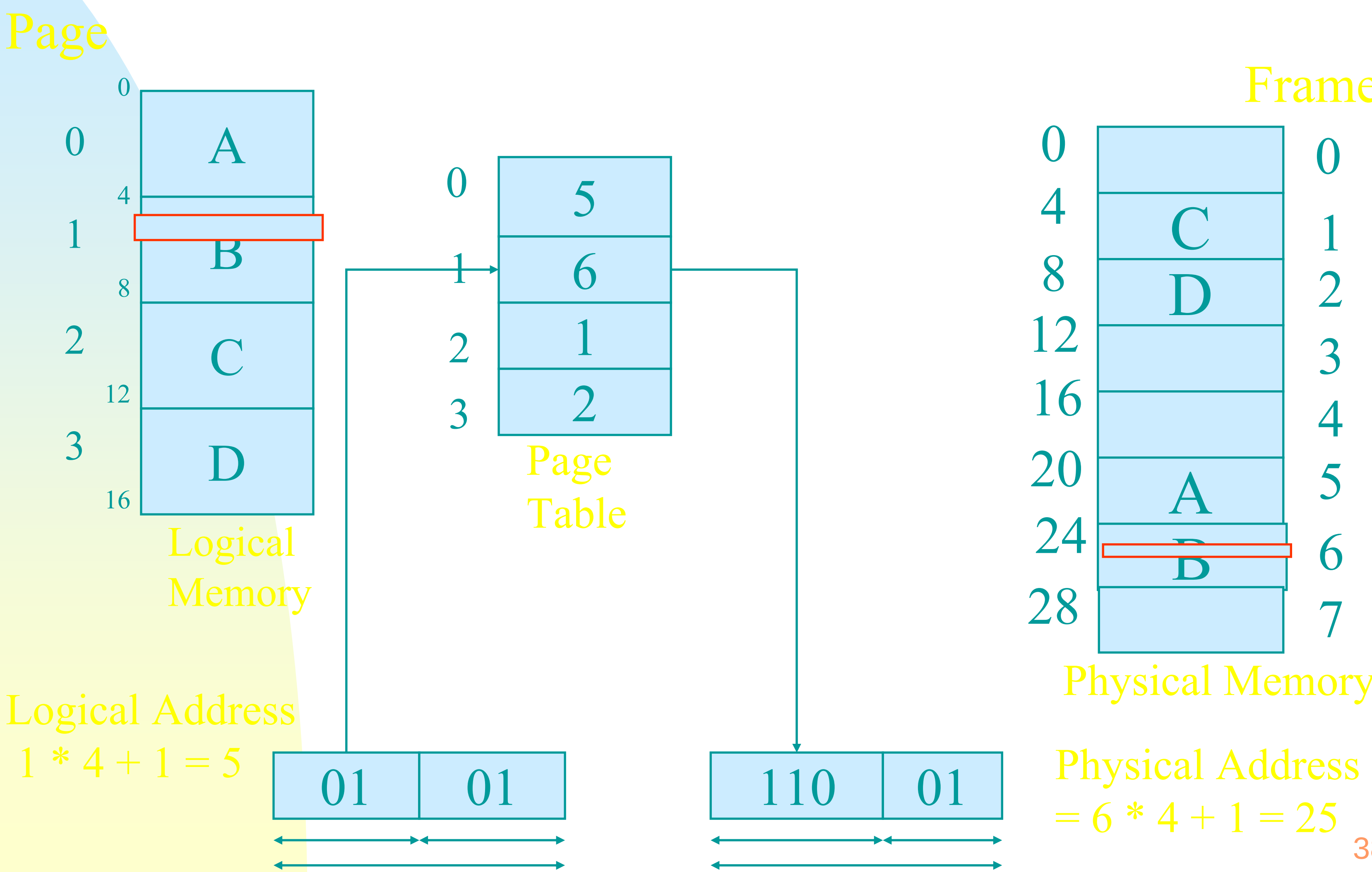
max number of pages: 2^{m-n}

Logical Address Space: 2^m

Physical Address Space: ???

- A page size tends to be a power of 2 for efficient address translation.
- The actual page size depends on the computer architecture. Today, it is from 512B or 16KB.

Paging – Basic Method



Paging – Basic Method

- No External Fragmentation
  - Paging is a form of dynamic relocation.
  - The average internal fragmentation is about one-half page per process
- The page size generally grows over time as processes, data sets, and memory have become larger.
  - 4-byte page table entry & 4KB per page → $2^{32} * 2^{12}B = 2^{44}B = 16TB$ of physical memory

Page Size ↑	Disk I/O Efficiency ↑	Page Table Maintenance ↓	Internal Fragmentation ↑

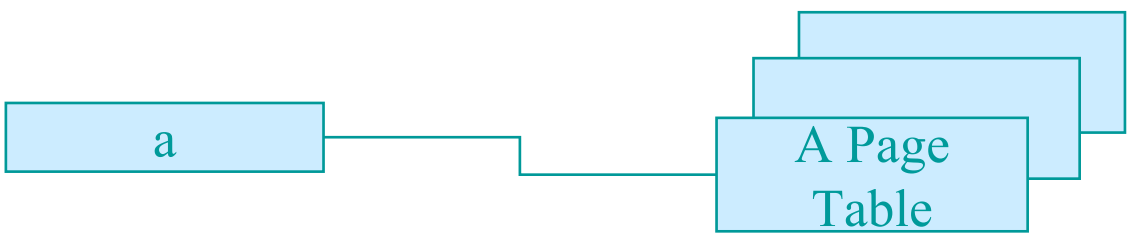
* Example: 8KB or 4KB for Solaris.

Paging – Basic Method

- Page Replacement:
  - An executing process has all of its pages in physical memory.
- Maintenance of the Frame Table
  - One entry for each physical frame
    - The status of each frame (free or allocated) and its owner
- The page table of each process must be saved when the process is preempted. → Paging increases context-switch time!

Paging – Hardware Support

- Page Tables
  - Where: Registers or Memory
    - Efficiency is the main consideration!
  - The use of registers for page tables
    - The page table must be small!
  - The use of memory for page tables
    - Page-Table Base Register (PTBR)

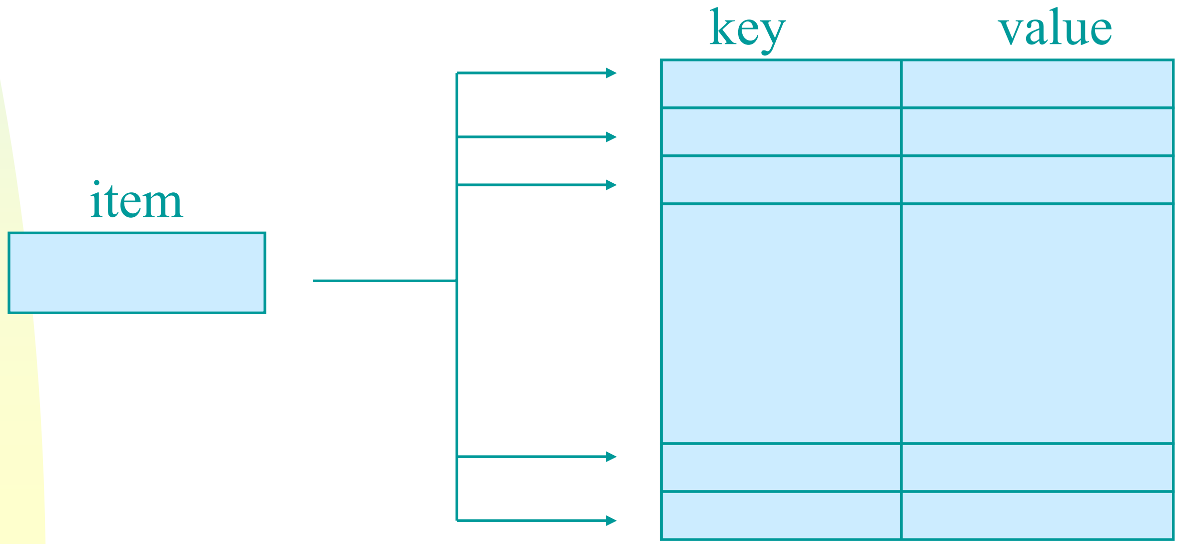


Paging – Hardware Support

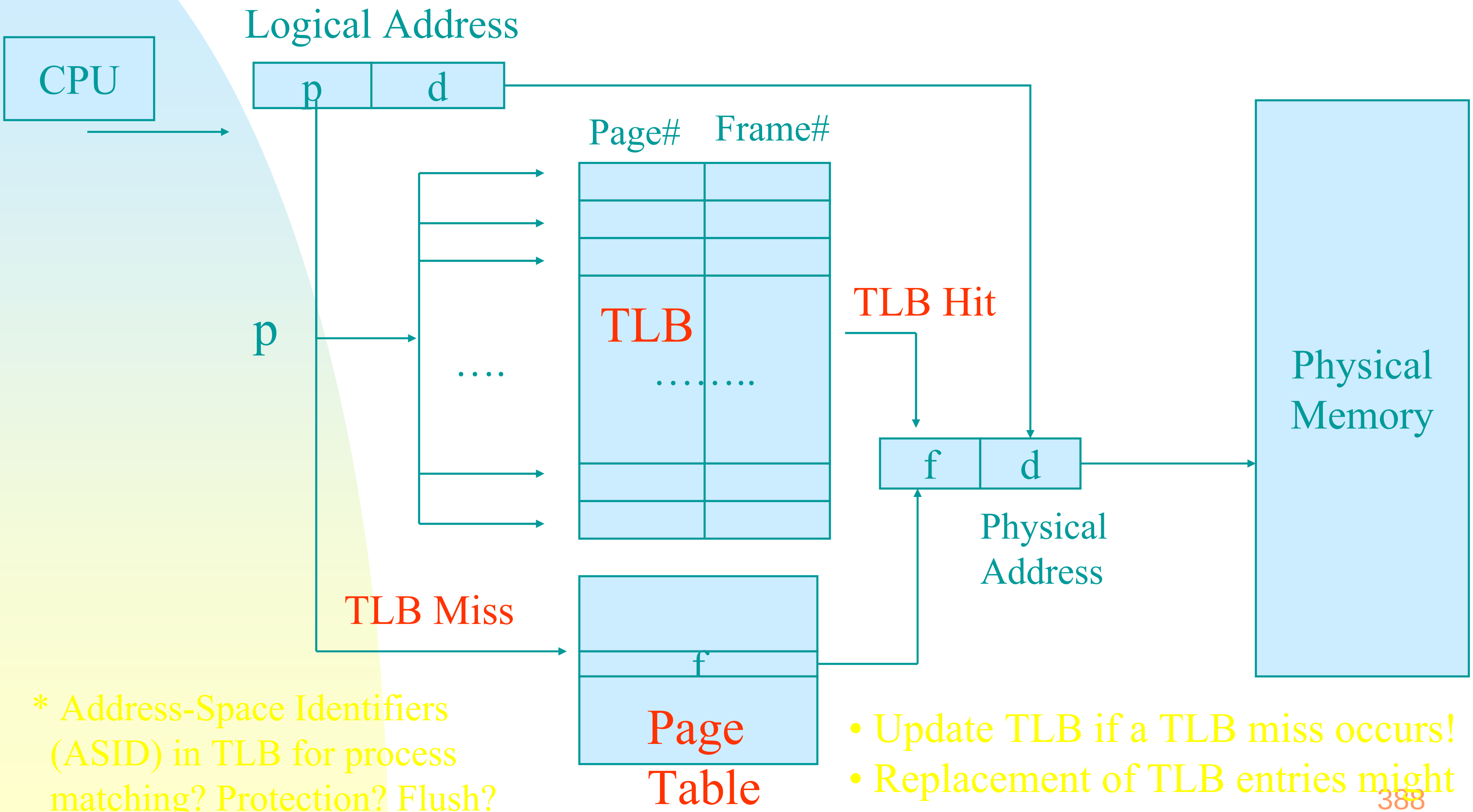
- Page Tables on Memory
  - Advantages:
    - The size of a page table is unlimited!
    - The context switch cost may be low if the CPU dispatcher merely changes PTBR, instead of reloading another page table.
  - Disadvantages:
    - Memory access is slowed by a factor of 2
      - Translation Look-aside buffers (TLB)
        - Associate, high-speed memory
        - (key/tag, value) – 16 ~ 1024 entries
        - Less than 10% memory access time

Paging – Hardware Support

- Translation Look-aside Buffers(TLB):
  - Disadvantages: Expensive Hardware and Flushing of Contents for Switching of Page Tables
  - Advantage: Fast – Constant-Search Time



Paging – Hardware Support



* Address-Space Identifiers (ASID) in TLB for process matching? Protection? Flush?

- Update TLB if a TLB miss occurs!
- Replacement of TLB entries might be needed.

Paging – Effective Memory Access Time

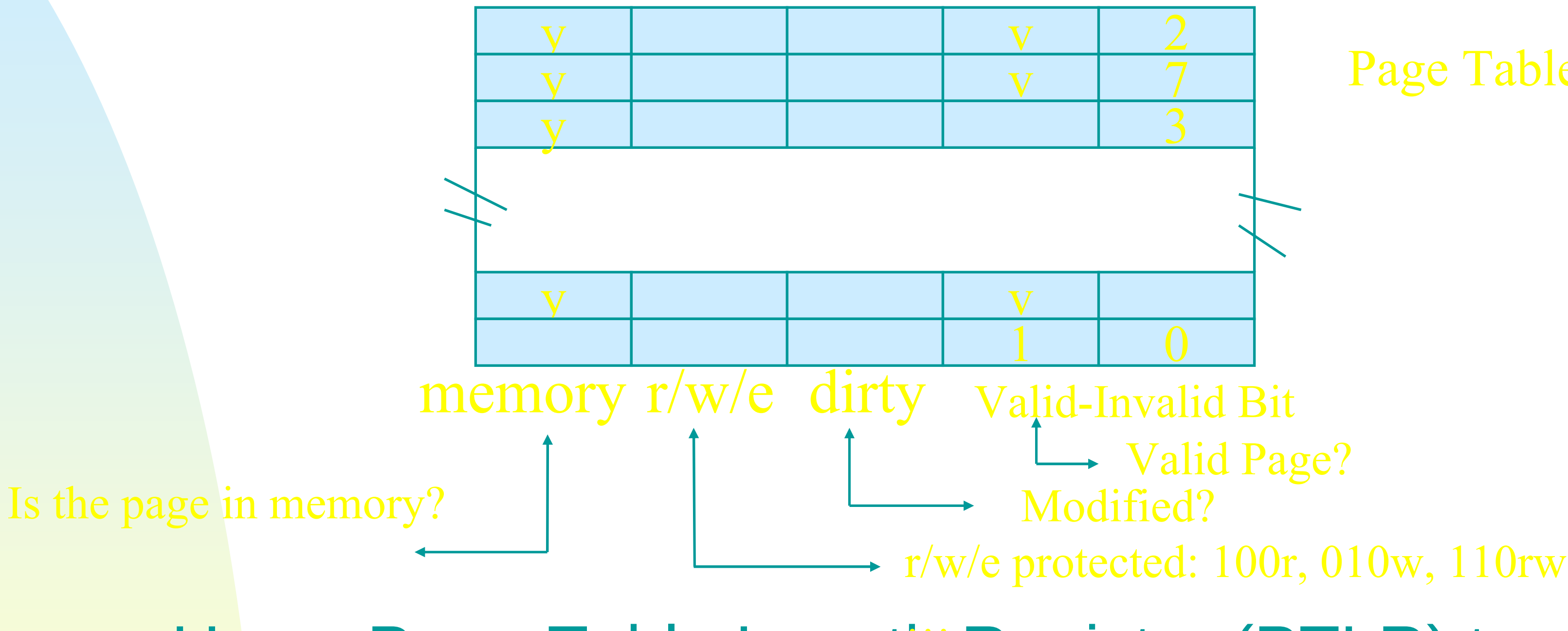
- Hit Ratio = the percentage of times that a page number is found in the TLB
  - The hit ratio of a TLB largely depends on the size and the replacement strategy of TLB entries!
- Effective Memory Access Time
  - Hit-Ratio * (TLB lookup + a mapped memory access) + (1 – Hit-Ratio) * (TLB lookup + a page table lookup + a mapped memory access)

Paging – Effective Memory Access Time

- An Example
  - 20ns per TLB lookup, 100ns per memory access
  - Effective Access Time = 0.8*120ns +0.2*220ns = 140 ns, when hit ratio = 80%
  - Effective access time = 0.98*120ns +0.02*220ns = 122 ns, when hit ratio = 98%
- Intel 486 has a 32-register TLB and claims a 98 percent hit ratio.

Paging – Protection & Sharing

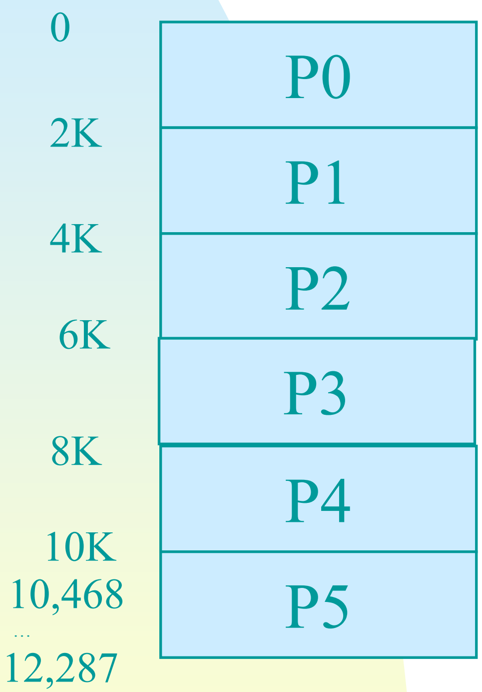
- Protection



  - Use a Page-Table Length Register (PTLR) to indicate the size of the page table.
  - Unused Paged table entries might be ignored during maintenance.

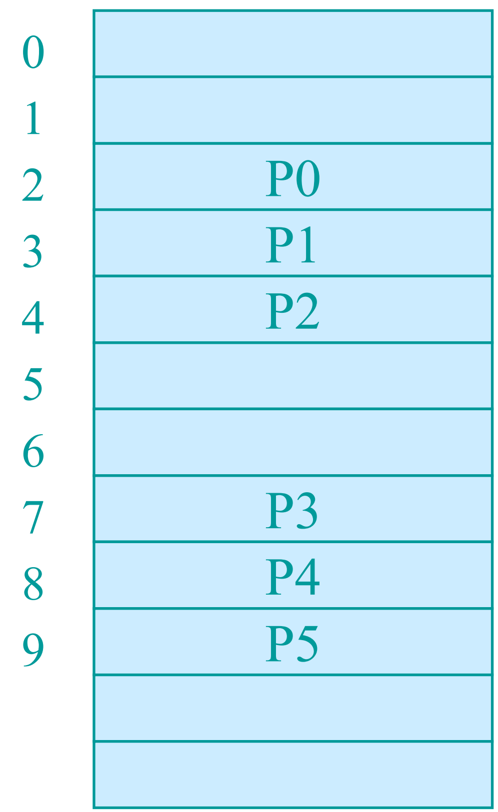
Paging – Protection & Sharing

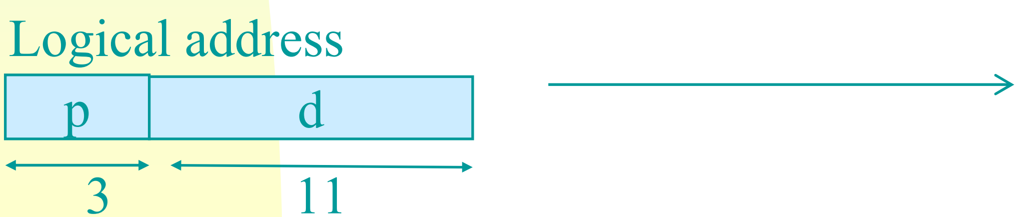
- Example: a 12287-byte Process ($16384=2^{14}$)



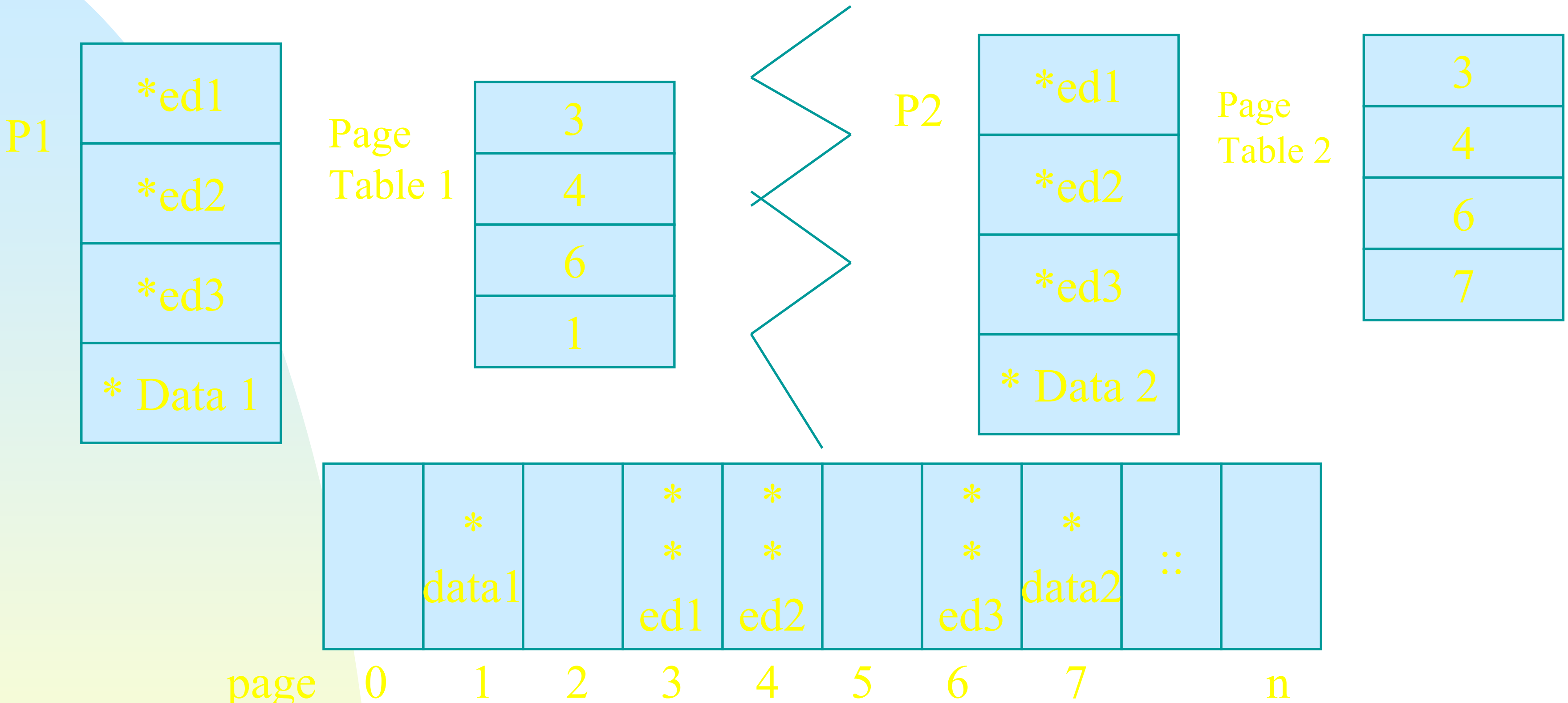
0		V	2
1		V	3
2		V	4
3		V	7
4		V	8
5		V	9
6		i	0
7		i	0

Page Table
(PTLR entries?)





Paging – Protection & Sharing



- Procedures which are executed often (e.g., editor) can be divided into procedure + date. Memory can be saved a lot.
- Reentrant procedures can be saved! The non-modified nature of saved code must be enforced
- Address referencing inside shared pages could be an issue.

Multilevel Paging

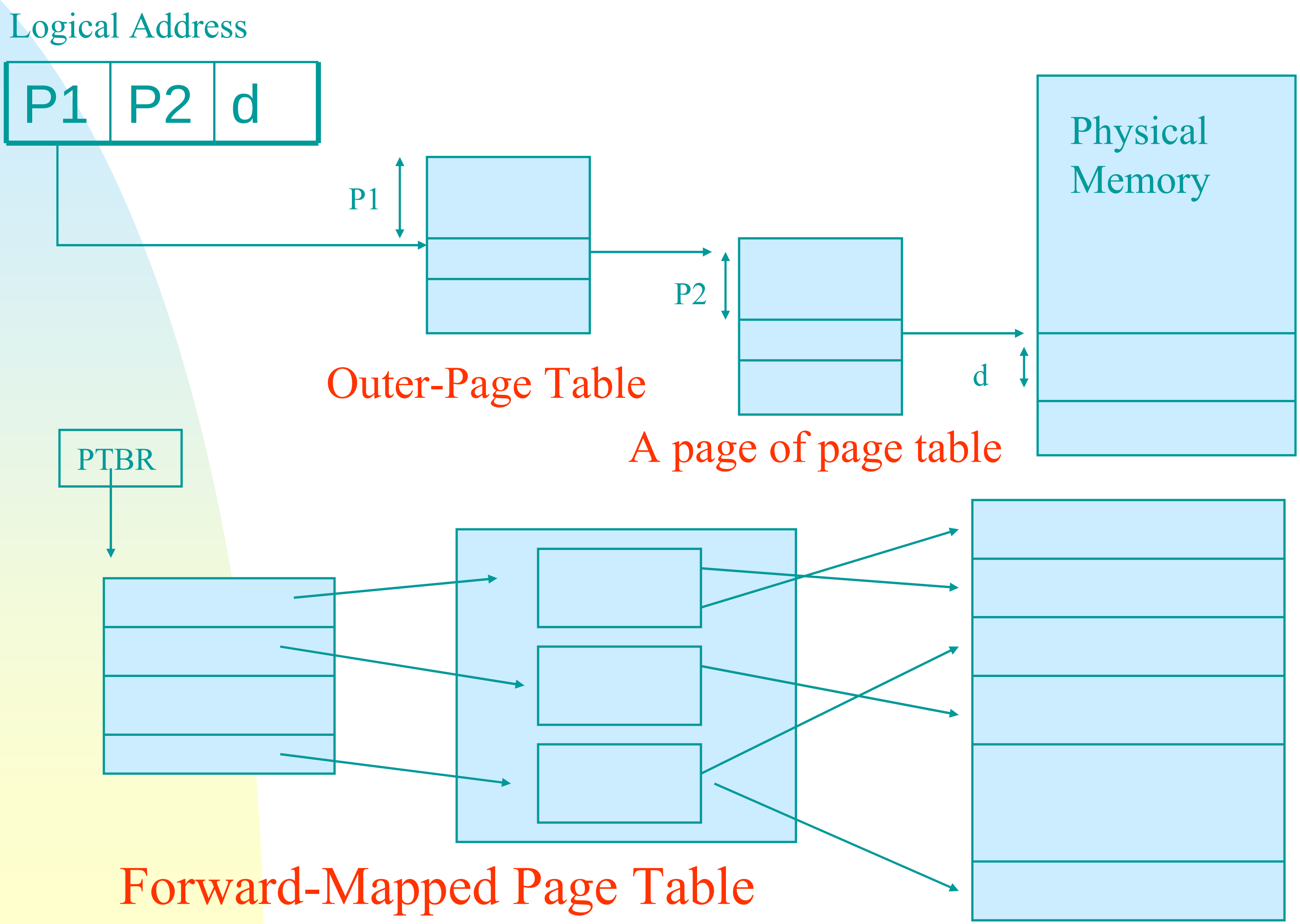
- Motivation
  - The logical address space of a process in many modern computer system is very large, e.g., 2^{32} to 2^{64} Bytes.

32-bit address → 2^{20} page entries → 4MB

4KB per page 4B per entries page table

→ Even the page table must be divided into pieces to fit in the memory!

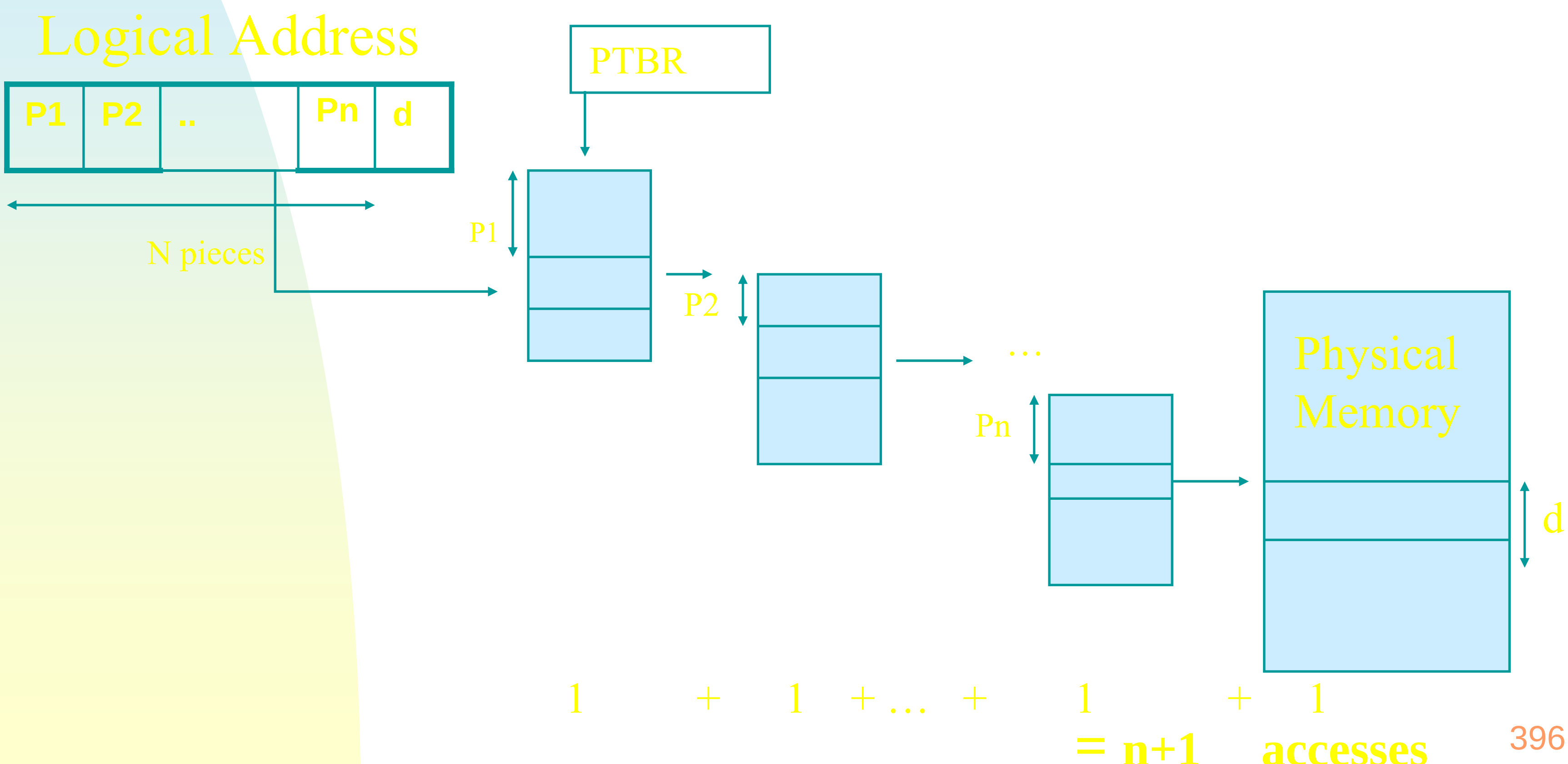
Multilevel Paging – Two-Level Paging



Forward-Mapped Page Table

Multilevel Paging – N-Level Paging

- Motivation: Two-level paging is not appropriate for a huge logical address space!



Multilevel Paging – N-Level Paging

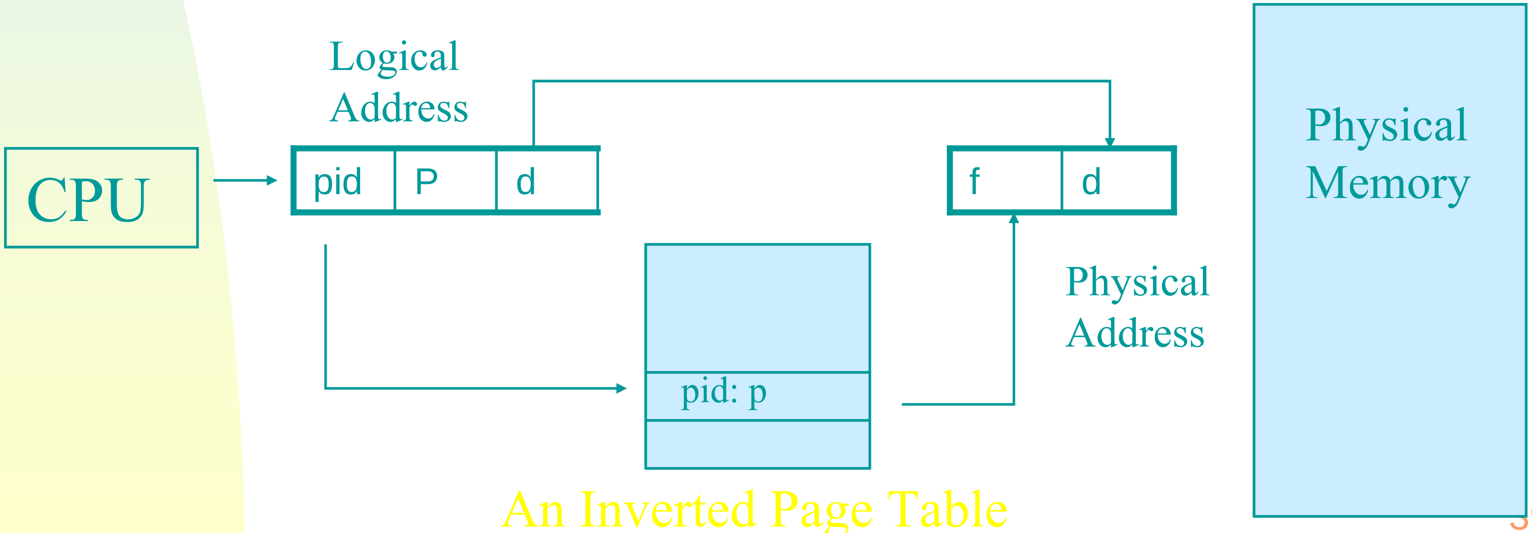
- Example
  - 98% hit ratio, 4-level paging, 20ns TLB access time, 100ns memory access time.
  - Effective access time = 0.98 X 120ns + 0.02 X 520ns = 128ns
- SUN SPARC (32-bit addressing) → 3-level paging
- Motorola 68030 (32-bit addressing) → 4-level paging
- VAX (32-bit addressing) → 2-level paging

Hashed Page Tables

- Objective:
  - To handle large address spaces
- Virtual address → hash function → a linked list of elements
  - (virtual page #, frame #, a pointer)
- Clustered Page Tables
  - Each entry contains the mappings for several physical-page frames, e.g., 16.

Inverted Page Table

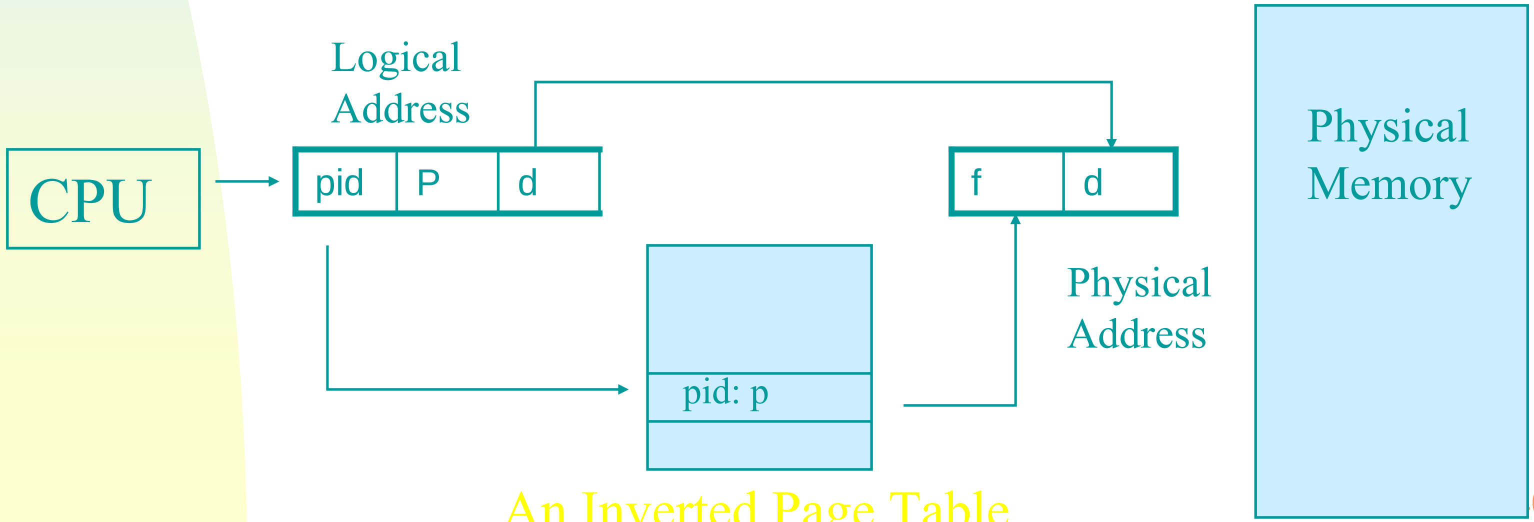
- Motivation
  - A page table tends to be big and does not correspond to the # of pages residing in the physical memory.
- Each entry corresponds to a physical frame.
  - Virtual Address: <Process ID, Page Number, Offset>



An Inverted Page Table

Inverted Page Table

- Each entry contains the virtual address of the frame.
  - Entries are sorted by physical addresses.
  - One table per system.
- When no match is found, the page table of the corresponding process must be referenced.
- Example Systems: HP Spectrum, IBM RT, PowerPC, SUN UltraSPARC



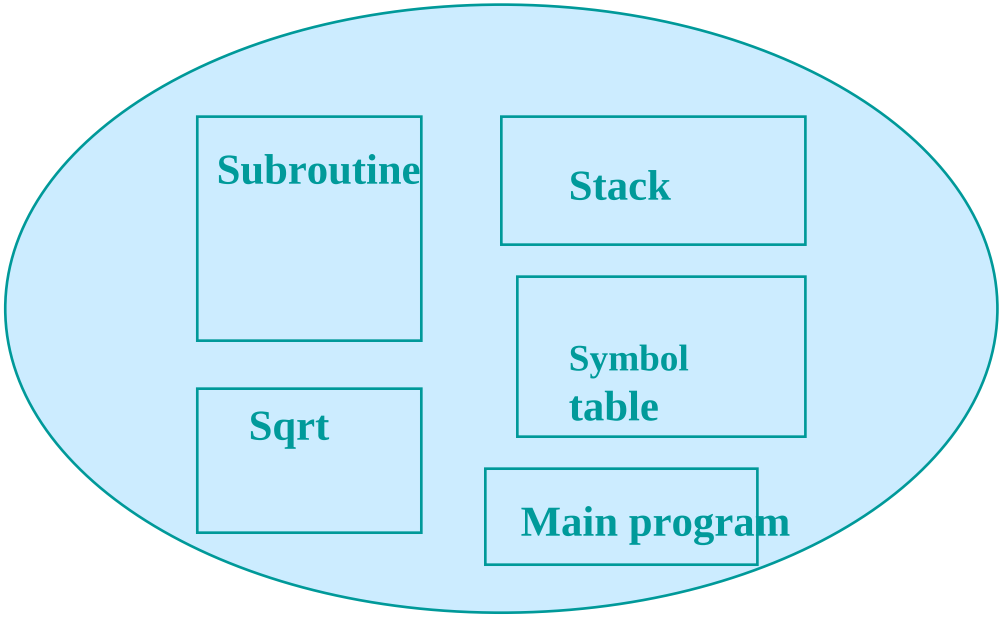
An Inverted Page Table

Inverted Page Table

- Advantage
  - Decrease the amount of memory needed to store each page table
- Disadvantage
  - The inverted page table is sorted by physical addresses, whereas a page reference is in logical address.
    - The use of Hash Table to eliminate lengthy table lookup time: 1HASH + 1IPT
    - The use of an associate memory to hold recently located entries.
  - Difficult to implement with shared memory

Segmentation

- Segmentation is a memory management scheme that support the user view of memory:
  - A logical address space is a collection of segments with variable lengths.



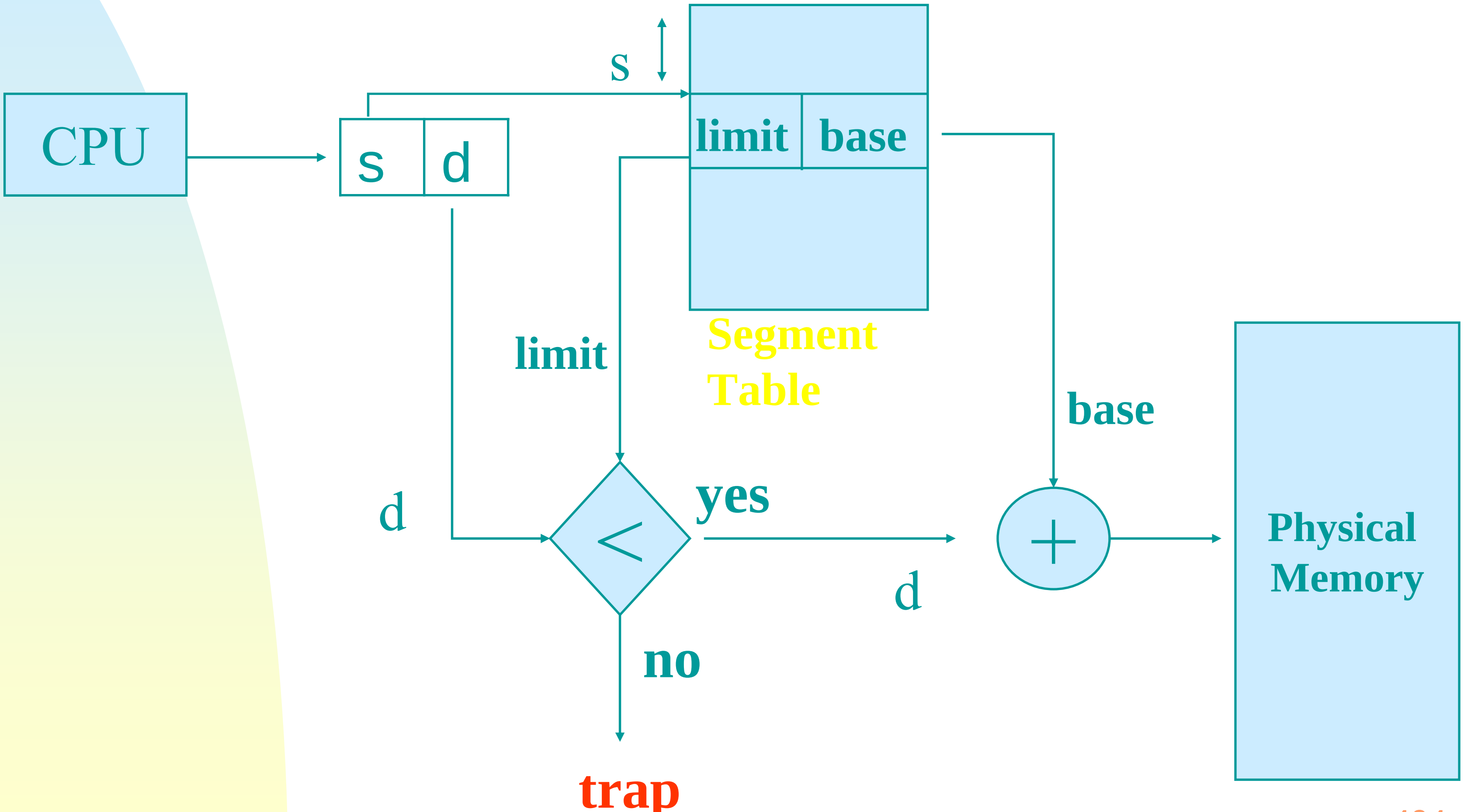
Segmentation

- Why Segmentation?
  - Paging separates the user's view of memory from the actual physical memory but does not reflect the logical units of a process!
  - Pages & frames are fixed-sized, but segments have variable sizes.
- For simplicity of representation,

  <segment name, offset> → <segment-number, offset>

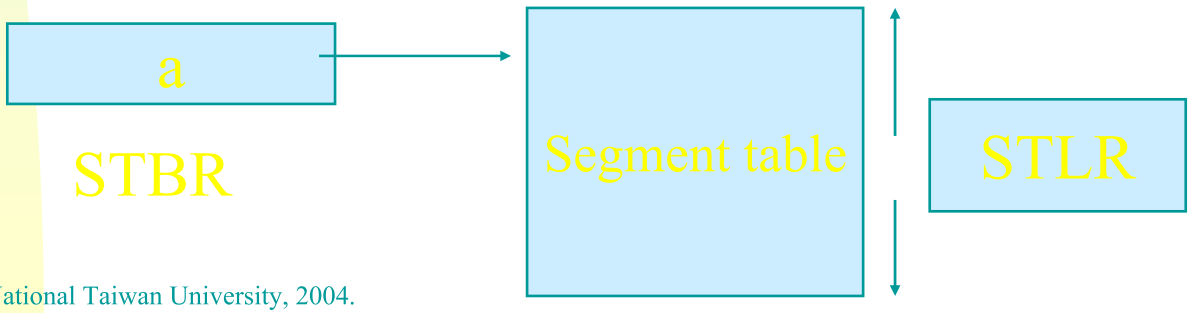
Segmentation – Hardware Support

- Address Mapping



Segmentation – Hardware Support

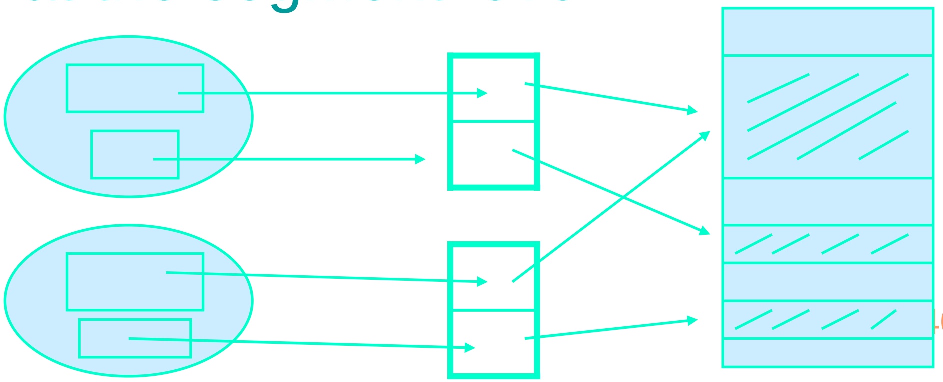
- Implementation in Registers – limited size!
- Implementation in Memory
  - Segment-table base register (STBR)
  - Segment-table length register (STLR)
  - Advantages & Disadvantages – Paging
    - Use an associate memory (TLB) to improve the effective memory access time !
    - TLB must be flushed whenever a new segment table is used !



Segmentation – Protection & Sharing

- Advantage:
  - Segments are a semantically defined portion of the program and likely to have all entries being “homogeneous”.
    - Example: Array, code, stack, data, etc.

      → Logical units for protection !
  - Sharing of code & data improves memory usage.
    - Sharing occurs at the segment level.



Segmentation – Protection & Sharing

- Potential Problems
  - External Fragmentation
  - Segments must occupy contiguous memory.
  - Address referencing inside shared segments can be a big issue:

Seg#	offset

Indirect addressing?!!!

Should all shared-code segments have the same segment number?

- How to find the right segment number if the number of users sharing the segments increase! → Avoid reference to segment #

Segmentation – Fragmentation

- Motivation:

  Segments are of variable lengths!

  → Memory allocation is a dynamic storage-allocation problem.

  - best-fit? first-fit? worst-ft?
- External fragmentation will occur!!
  - Factors, e.g., average segment sizes

Size

A byte

External Fragmentation

Overheads increases substantially!

(base+limit "registers")

Segmentation – Fragmentation

- Remark:
  - Its external fragmentation problem is better than that of the dynamic partition method because segments are likely to be smaller than the entire process.
- Internal Fragmentation??

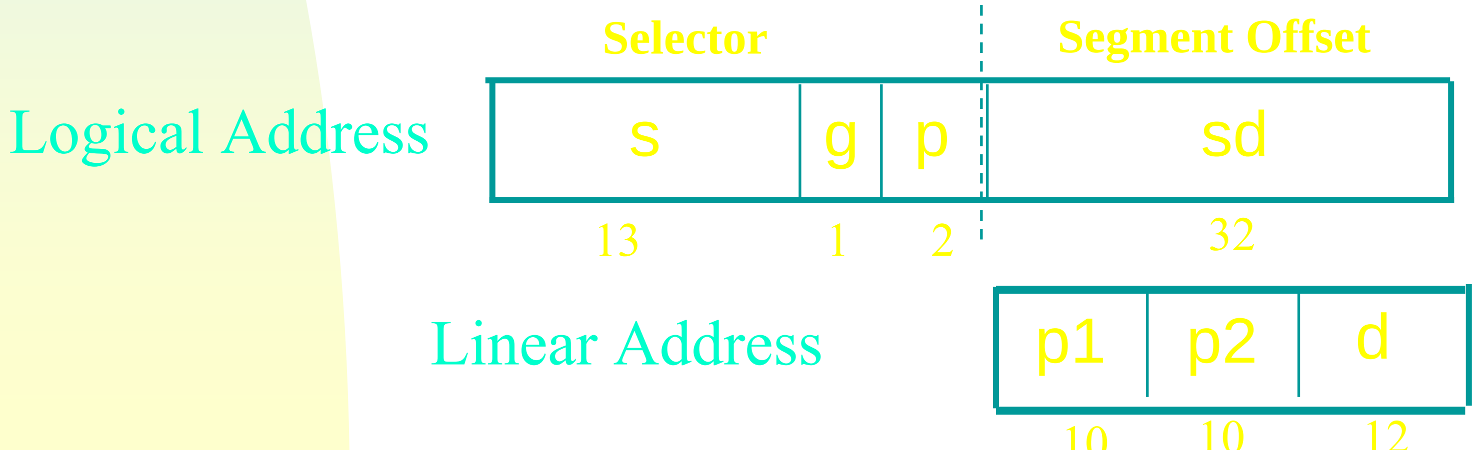
Segmentation with Paging

- Motivation :
  - Segmentation has external fragmentation.
  - Paging has internal fragmentation.
  - Segments are semantically defined portions of a program.

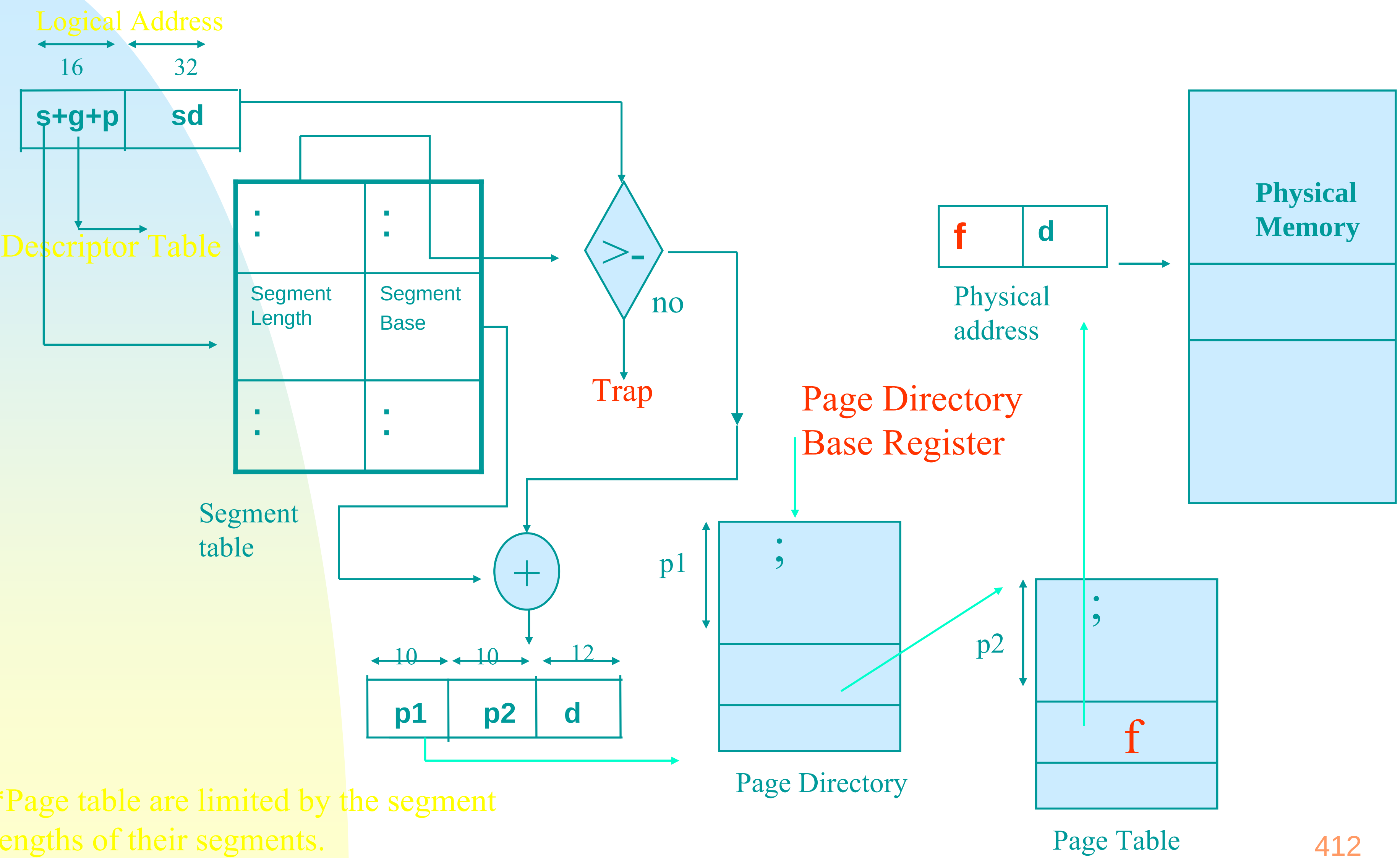
    → "Page" Segments !

Paged Segmentation – Intel 80386

- 8K Private Segments + 8K Public Segments
  - Page Size = 4KB, Max Segment Size = 4GB
  - Tables:
    - Local Descriptor Table (LDT)
    - Global Descriptor Table (GDT)
  - 6 microprogram segment registers for caching



Paged Segmentation – Intel 80386



*Page table are limited by the segment lengths of their segments.

Paging and Segmentation

- To overcome disadvantages of paging or segmentation alone:
  - Paged segments – divide segments further into pages.
    - Segment need not be in contiguous memory.
  - Segmented paging – segment the page table.
    - Variable size page tables.
- Address translation overheads increase!
- An entire process still needs to be in memory at once!

→ Virtual Memory!!

Paging and Segmentation

- Considerations in Memory Management
  - Hardware Support, e.g., STBR, TLB, etc.
  - Performance
  - Fragmentation
    - Multiprogramming Levels
  - Relocation Constraints?
  - Swapping: +
  - Sharing?!
  - Protection?!

Contents

Chapter 10
Virtual Memory

Virtual Memory

- Virtual Memory
  - A technique that allows the execution of a process that may not be completely in memory.
- Motivation:
  - An entire program in execution may not all be needed at the same time!
    - e.g. error handling routines, a large array, certain program features, etc

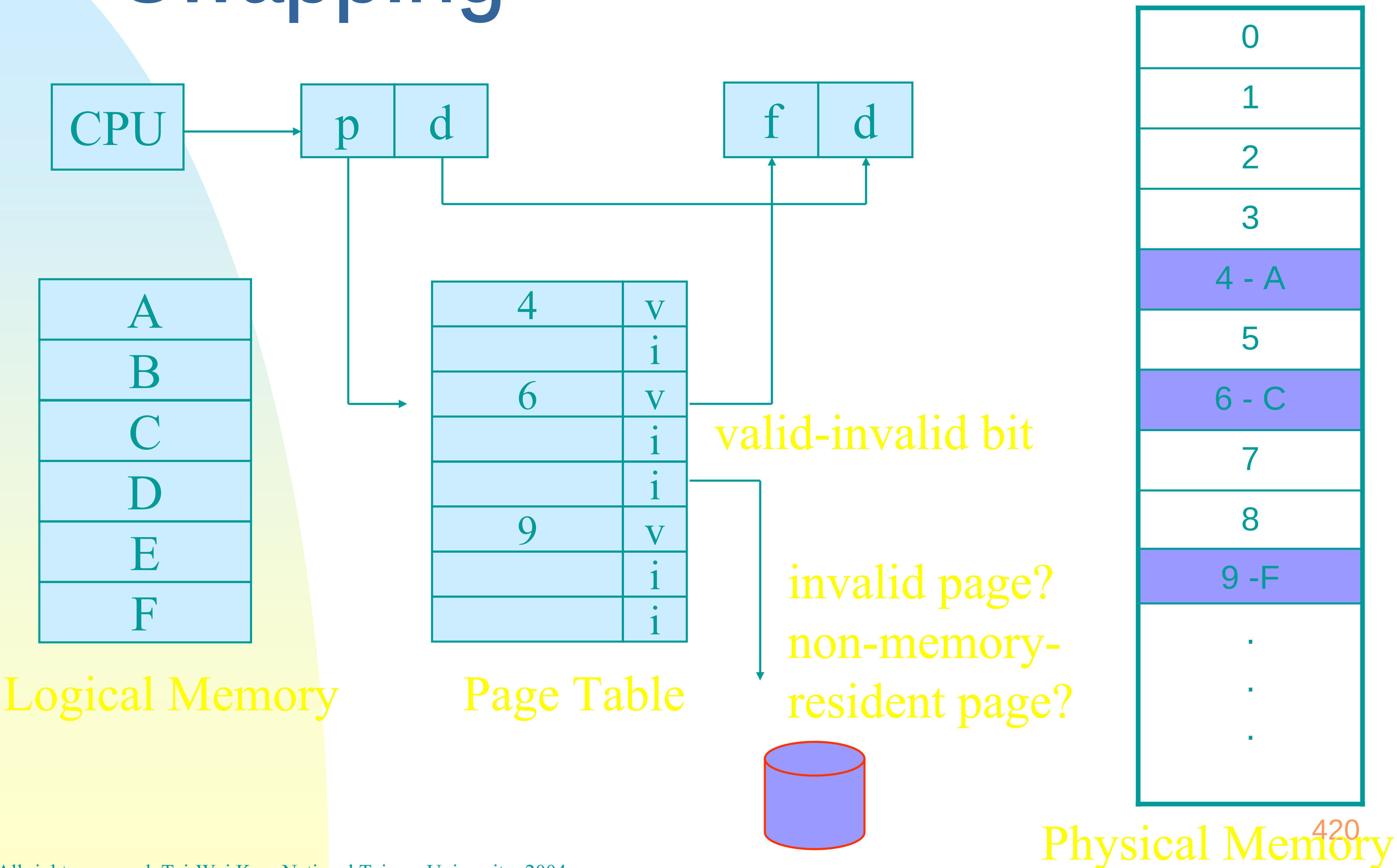
Virtual Memory

- Potential Benefits
  - Programs can be much larger than the amount of physical memory. Users can concentrate on their problem programming.
  - The level of multiprogramming increases because processes occupy less physical memory.
  - Each user program may run faster because less I/O is needed for loading or swapping user programs.
- Implementation: demand paging, demand segmentation (more difficult),etc.

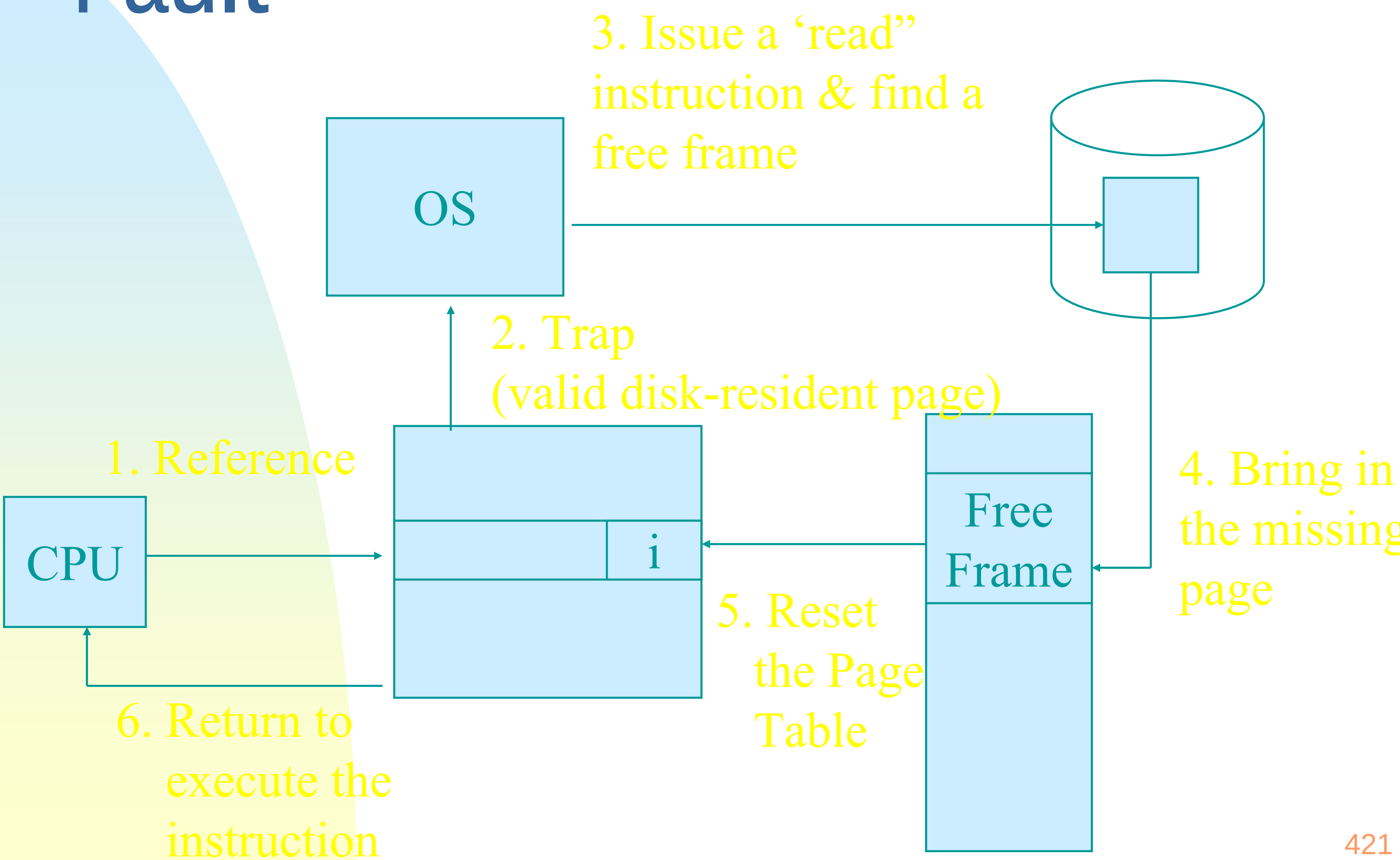
Demand Paging – Lazy Swapping

- Process image may reside on the backing store. Rather than swap in the entire process image into memory, Lazy Swapper only swap in a page when it is needed!
  - Pure Demand Paging – Pager vs Swapper
  - A Mechanism required to recover from the missing of non-resident referenced pages.
    - A *page fault* occurs when a process references a non-memory-resident page.

Demand Paging – Lazy Swapping



A Procedure to Handle a Page Fault



A Procedure to Handle A Page Fault

- Pure Demand Paging:
  - Never bring in a page into the memory until it is required!
- Pre-Paging
  - Bring into the memory all of the pages that “will” be needed at one time!
  - Locality of reference

Hardware Support for Demand Paging

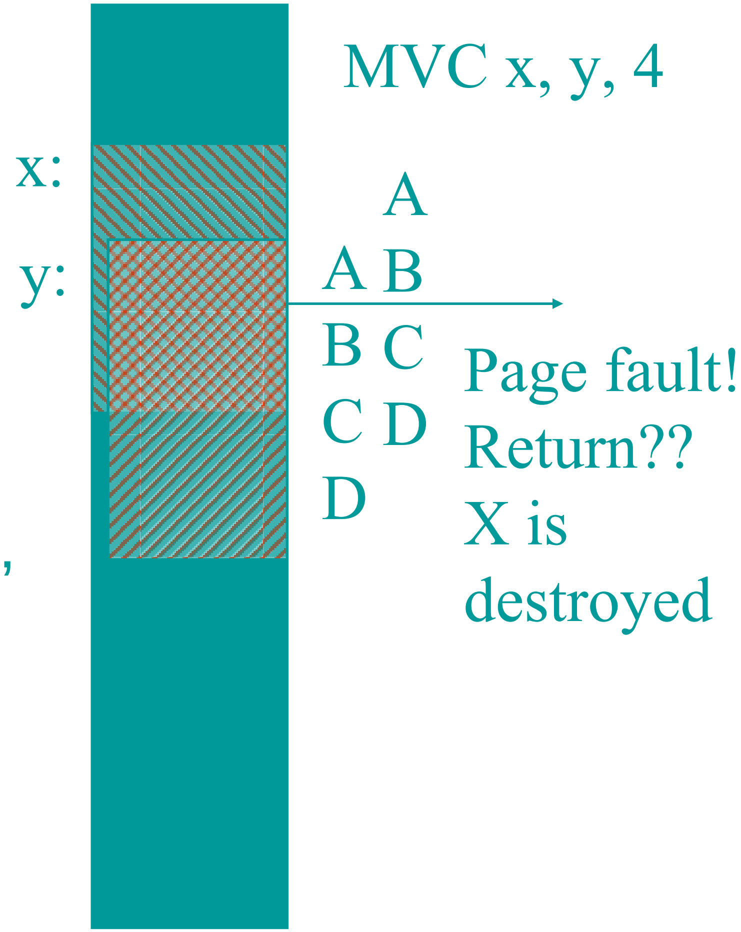
- New Bits in the Page Table
  - To indicate that a page is now in memory or not.
- Secondary Storage
  - Swap space in the backing store
    - A continuous section of space in the secondary storage for better performance.

Crucial issues

- Example 1 – Cost in restarting an instruction
  - Assembly Instruction: Add a, b, c
  - Only a short job!
    - Re-fetch the instruction, decode, fetch operands, execute, save, etc
  - Strategy:
    - Get all pages and restart the instruction from the beginning!

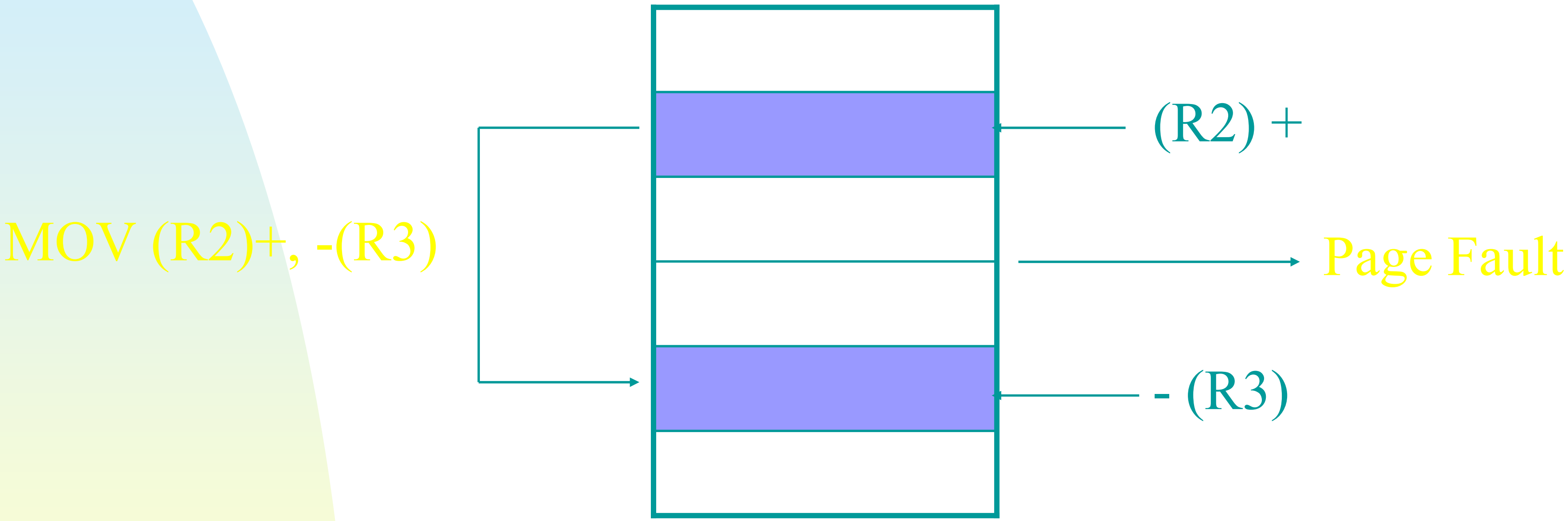
Crucial Issues

- Example 2 – Block-Moving Assembly Instruction
  - MVC x, y, 256
    - IBM System 360/ 370
  - Characteristics
    - More expensive
    - “self-modifying” “operands”
  - Solutions:
    - Pre-load pages
    - Pre-save & recover before page-fault services



Crucial Issues

- Example 3 – Addressing Mode



When the page fault is serviced,
R2, R3 are modified!
- Undo Effects!

Performance of Demand Paging

- Effective Access Time:
  - ma: memory access time for paging
  - p: probability of a page fault
  - pft: page fault time

$$(1 - p) * ma + p * pft$$

Performance of Demand Paging

- Page fault time - major components
  - Components 1&3 (about 10^3 ns ~ 10^5 ns)
    - Service the page-fault interrupt
    - Restart the process
  - Component 2 (about 25ms)
    - Read in the page (multiprogramming! However, let's get the taste!)
    - pft $\approx$ 25ms = 25,000,000 ns
- Effect Access Time (when ma = 100ns)
  - (1-p) * 100ns + p * 25,000,000 ns
  - 100ns + 24,999,900ns * p

Performance of Demand Paging

- Example (when ma = 100ns)
  - p = 1/1000
  - Effect Access Time ≈ 25,000 ns

    → Slowed down by 250 times
  - How to only 10% slow-down?

    $110 > 100 * (1-p) + 25{,}000{,}000 * p$

    $p < 0.0000004$

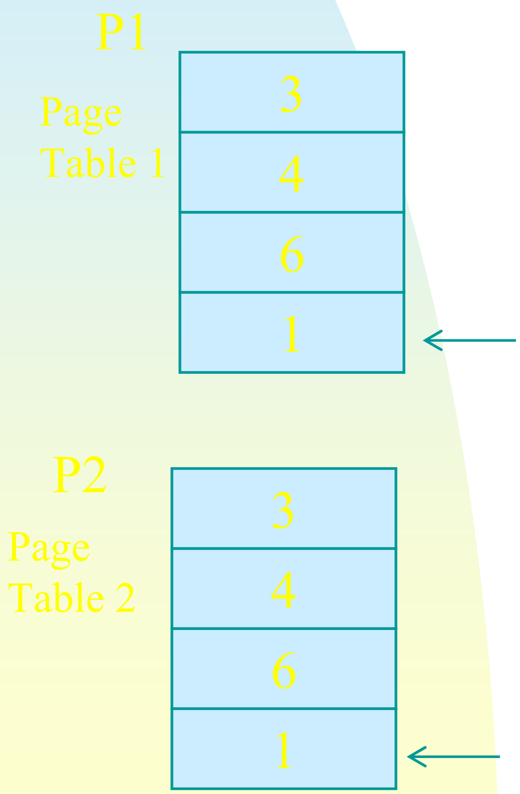
    $p < 1 / 2{,}500{,}000$

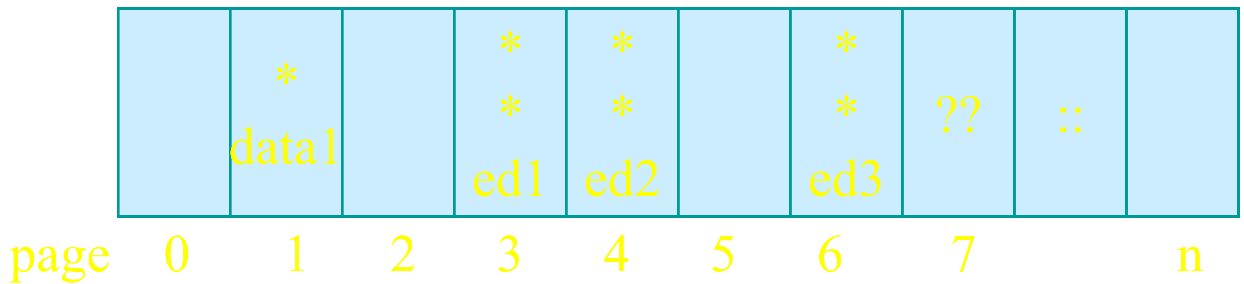
Performance of Demand Paging

- How to keep the page fault rate low?
  - Effective Access Time ≈ 100ns + 24,999,900ns * p
- Handling of Swap Space – A Way to Reduce Page Fault Time (pft)
  - Disk I/O to swap space is generally faster than that to the file system.
    - Preload processes into the swap space before they start up.
    - Demand paging from file system but do page replacement to the swap space. (BSD UNIX)

Process Creation

- Copy-on-Write
  - Rapid Process Creation and Reducing of New Pages for the New Process
  - fork(); execve()
    - Shared pages → copy-on-write pages
      - Only the pages that are modified are copied!





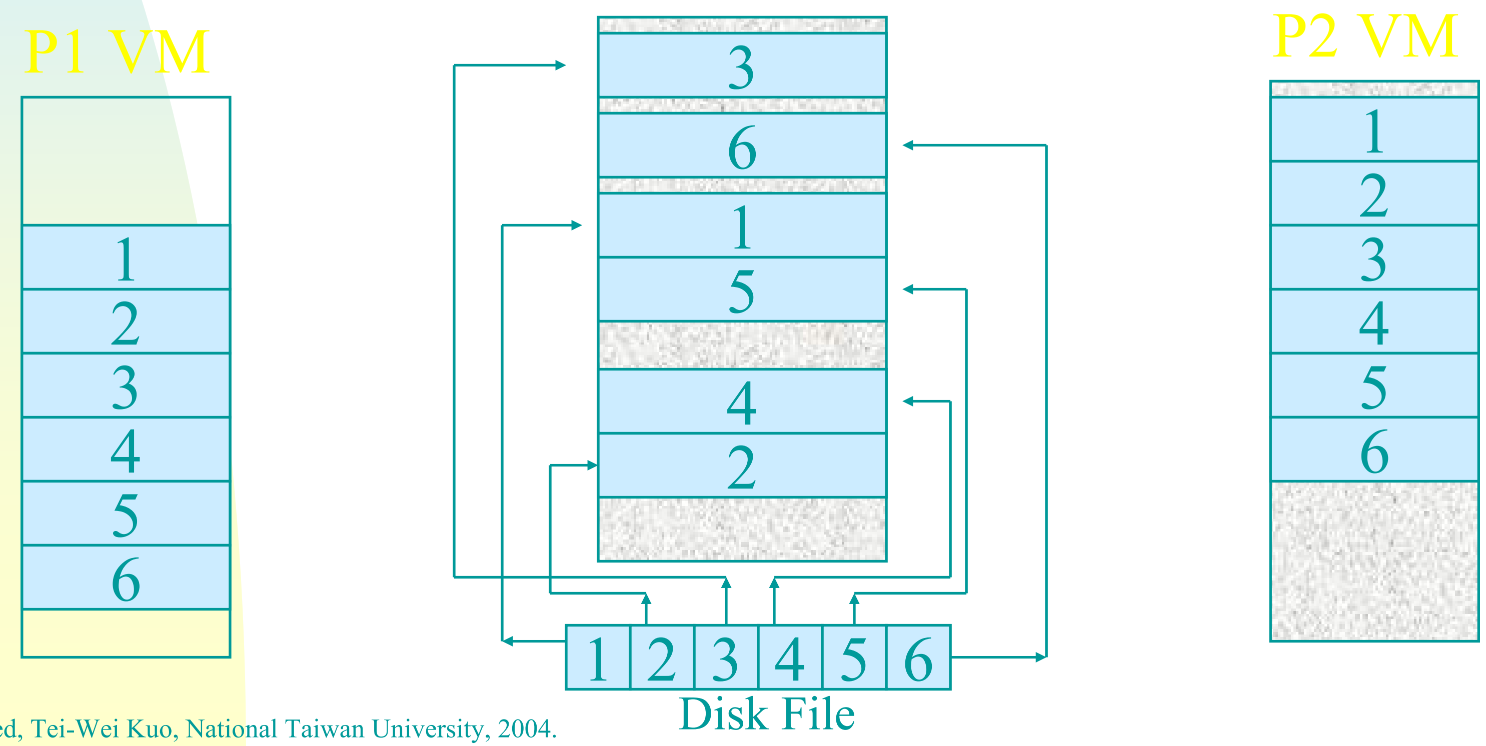
* Windows 2000, Linux, Solaris 2 support this feature!

Process Creation

- Copy-on-Write
  - zero-fill-on-demand
    - Zero-filled pages, e.g., those for the stack or bss.
  - vfork() vs fork() with copy-on-write
    - vfork() lets the sharing of the page table and pages between the parent and child processes.
    - Where to keep the needs of copy-on-write information for pages?

Memory-Mapped Files

- File writes might not cause any disk write!
- Solaris 2 uses memory-mapped files for open(), read(), write(), etc.



Page Replacement

- Demand paging increases the multiprogramming level of a system by "potentially" over-allocating memory.
  - Total physical memory = 40 frames
  - Run six processes of size equal to 10 frames but with only five frames. => 10 spare frames
- Most of the time, the average memory usage is close to the physical memory size if we increase a system's multiprogramming level!

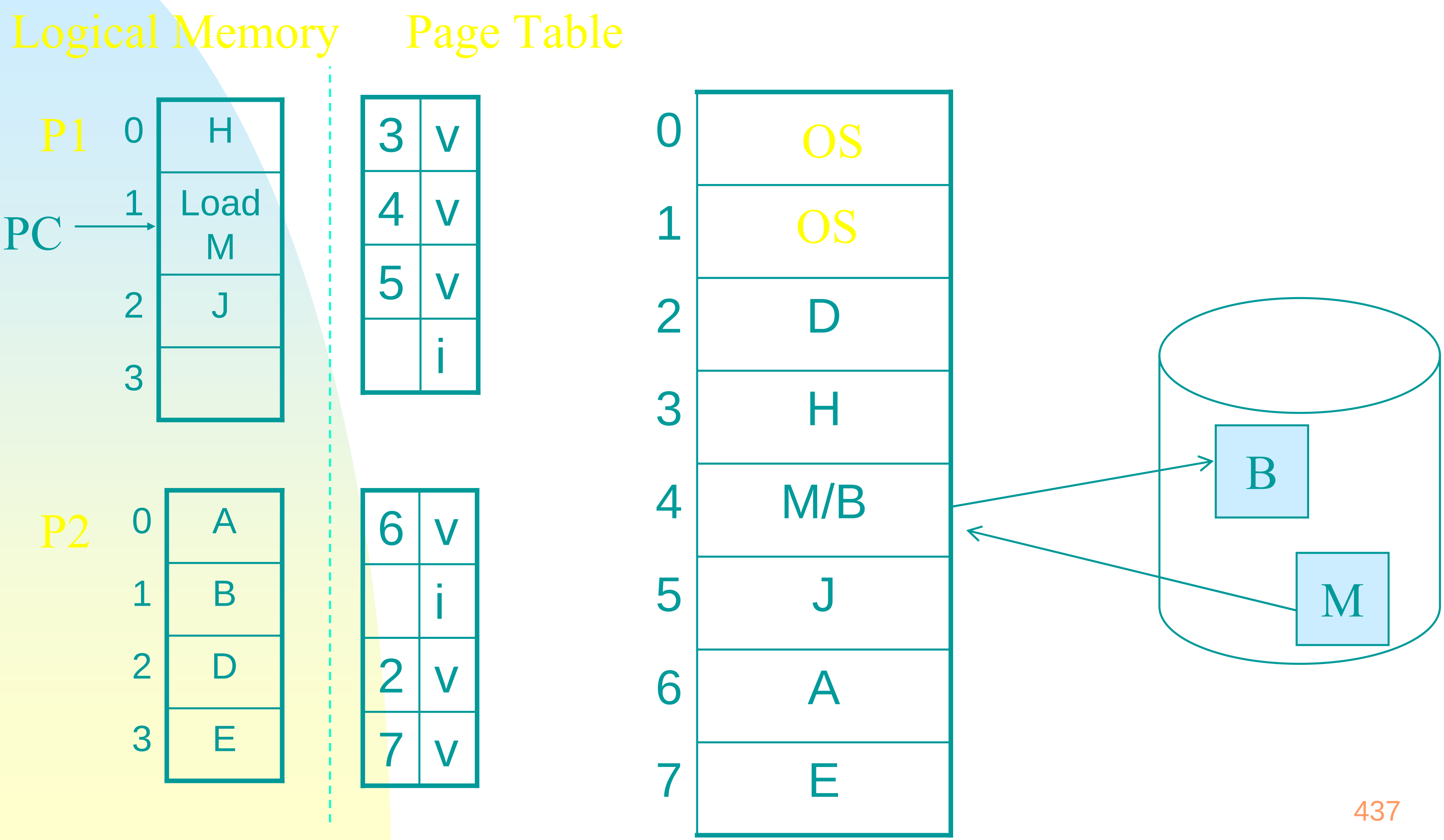
Page Replacement

- Q: Should we run the 7th processes?
  - How if the six processes start to ask their shares?
- What to do if all memory is in use, and more memory is needed?
- Answers
  - Kill a user process!
    - But, paging should be transparent to users?
  - Swap out a process!
  - Do page replacement!

Page Replacement

- A Page-Fault Service
  - Find the desired page on the disk!
  - Find a free frame
    - Select a victim and write the victim page out when there is no free frame!
  - Read the desired page into the selected frame.
  - Update the page and frame tables, and restart the user process.

Page Replacement



Page Replacement

- Two page transfers per page fault if no frame is available!

Page Table

6	V	N
4	V	N
3	V	Y
7	V	Y

Valid-Invalid Bit

Modify Bit is set by the hardware automatically!

Modify (/Dirty) Bit! To "eliminate" 'swap out" => Reduce I/O time by one-half

Page Replacement

- Two Major Pieces for Demand Paging
  - Frame Allocation Algorithms
    - How many frames are allocated to a process?
  - Page Replacement Algorithms
    - When page replacement is required, select the frame that is to be replaced!
  - Goal: A low page fault rate!
- Note that a bad replacement choice does not cause any incorrect execution!

Page Replacement Algorithms

- Evaluation of Algorithms
  - Calculate the number of page faults on strings of memory references, called reference strings, for a set of algorithms
- Sources of Reference Strings
  - Reference strings are generated artificially.
  - Reference strings are recorded as system traces:
    - How to reduce the number of data?

Page Replacement Algorithms

- Two Observations to Reduce the Number of Data:
  - Consider only the page numbers if the page size is fixed.
    - Reduce memory references into page references
  - If a page *p* is referenced, any immediately following references to page *p* will never cause a page fault.
    - Reduce consecutive page references of page *p* into one page reference.

Page Replacement Algorithms

- Example

$\underbrace{\boxed{\text{XX}}}_{\text{page\#}}\underbrace{\boxed{\text{XX}}}_{\text{offset}}$

0100, 0432, 0101, 0612, 0103, 0104, 0101, 0611

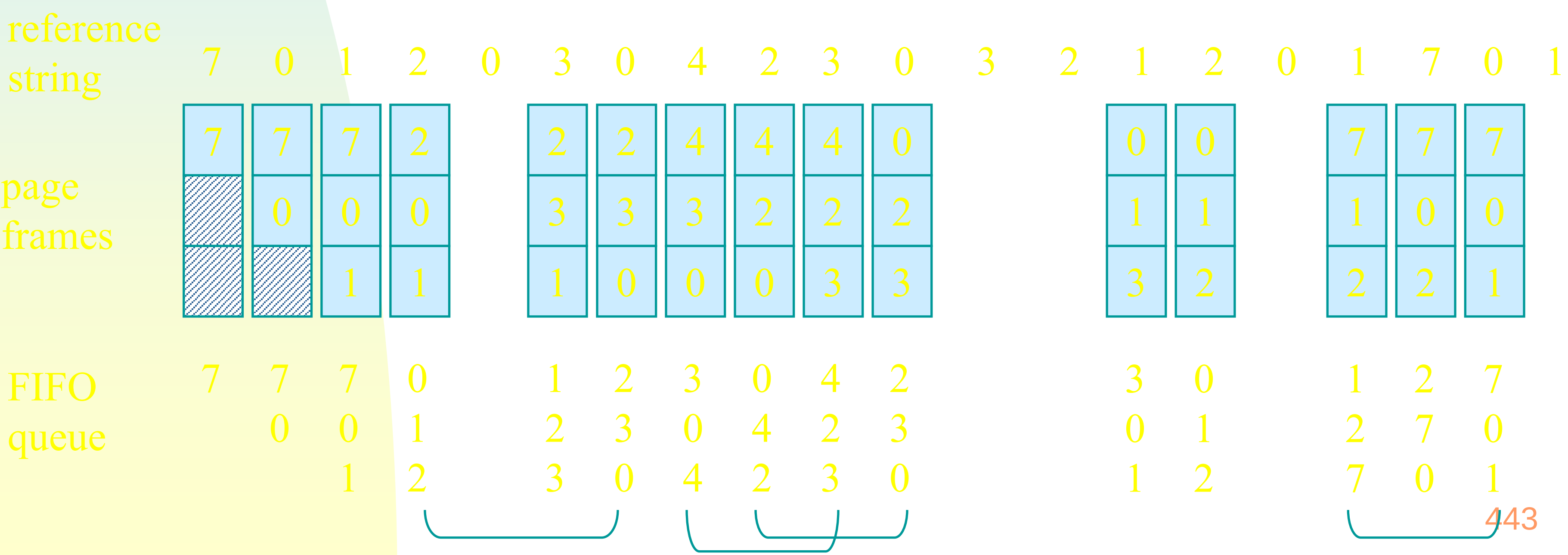
01, 04, 01, 06, <u>01, 01, 01</u>, 06

01, 04, 01, 06, 01, 06

Does the number of page faults decrease when the number of page frames available increases?

FIFO Algorithm

- A FIFO Implementation
  1. Each page is given a time stamp when it is brought into memory.
  2. Select the oldest page for replacement!



FIFO Algorithm

- The Idea behind FIFO
  - The oldest page is unlikely to be used again.

  ??Should we save the page which will be used in the near future??
- Belady’s anomaly
  - For some page-replacement algorithms, the page fault rate may increase as the number of allocated frames increases.

FIFO Algorithm

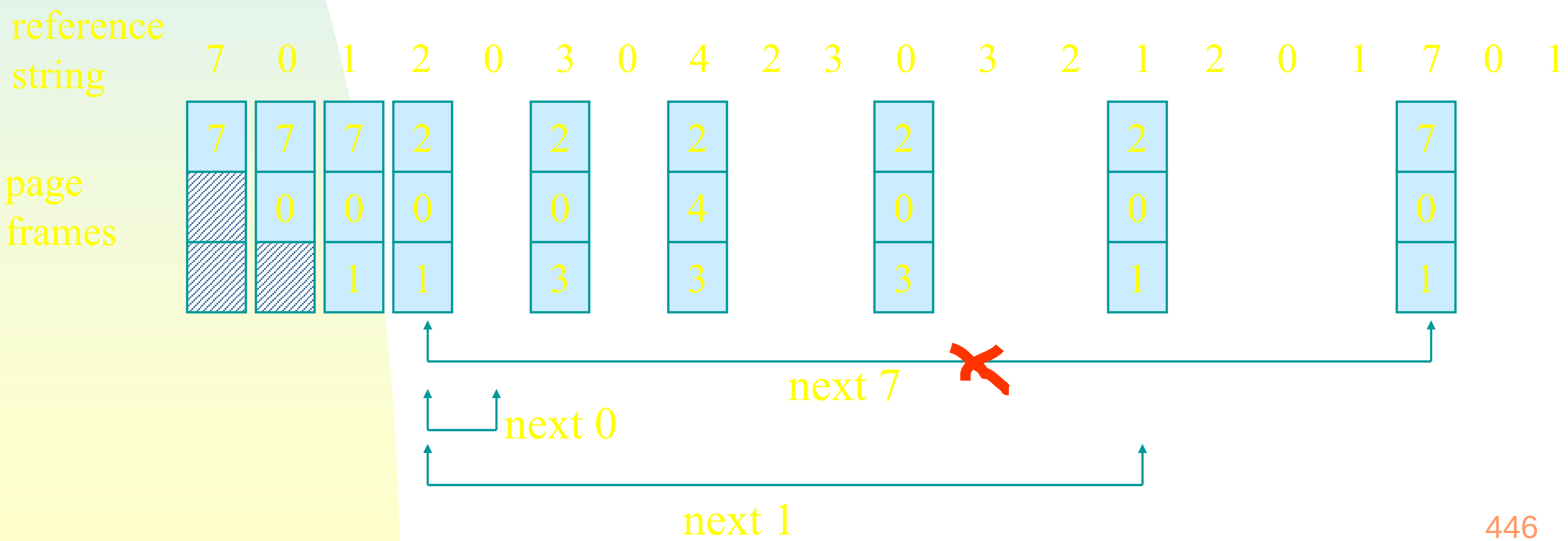
Run the FIFO algorithm on the following reference:

string:	1	2	3	4	1	2	5	1	2	3	4	5
	●	●	●	●	●	●	●			●	●	
3 frames	1	1	1	2	3	4	1	1	1	2	5	5
		2	2	3	4	1	2	2	2	5	3	3
			3	4	1	2	5	5	5	3	4	4
	●	●	●	●			●	●	●	●	●	●
4 frames	1	1	1	1	1	1	2	3	4	5	1	2
		2	2	2	2	2	3	4	5	1	2	3
			3	3	3	3	4	5	1	2	3	4
				4	4	4	5	1	2	3	4	5

→ Push out pages that will be used later!

Optimal Algorithm (OPT)

- Optimality
  - One with the lowest page fault rate.
- Replace the page that will not be used for the longest period of time. ←→ Future Prediction



Least-Recently-Used Algorithm (LRU)

- The Idea:
  - OPT concerns when a page is to be used!
  - “Don’t have knowledge about the future”?!

$$\Downarrow$$

- Use the history of page referencing in the past to predict the future!

$$S \stackrel{?}{=} S^R \text{ (} S^R \text{ is the reverse of } S \text{ !)}$$

LRU Algorithm

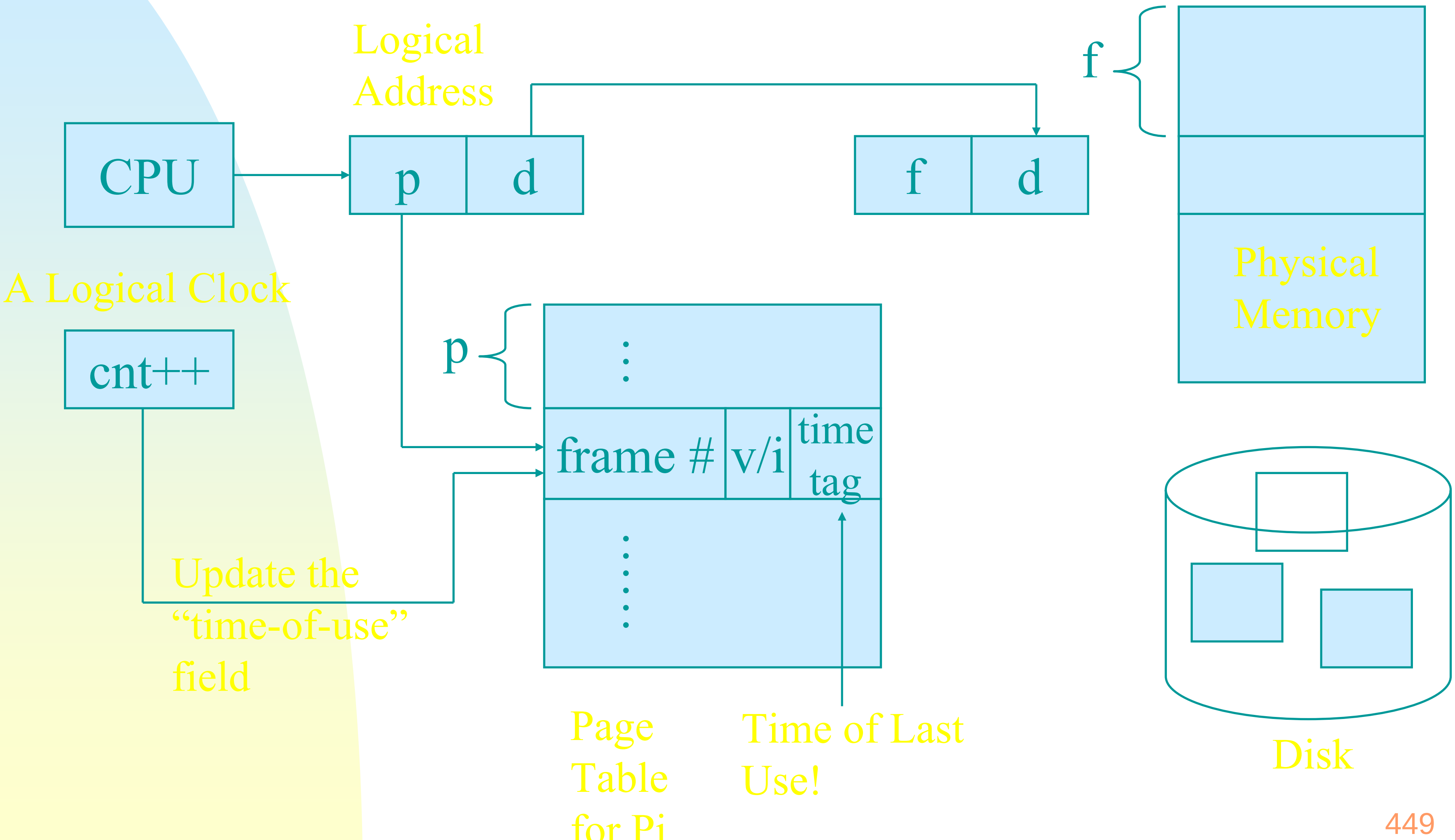
- Example

reference string	7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
page frames	7	7	7	2		2		4	4	4	0			1		1		7		
		0	0	0		0		0	0	3	3			3		0		0		
			1	1		3		3	2	2	2			2		2		7		
LRU queue	0	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
		7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0
			7	0	1	2	2	3	0	4	2	2	0	3	3	1	2	0	1	7

a wrong prediction!

Remark: LRU is like OPT which "looks backward" in time.

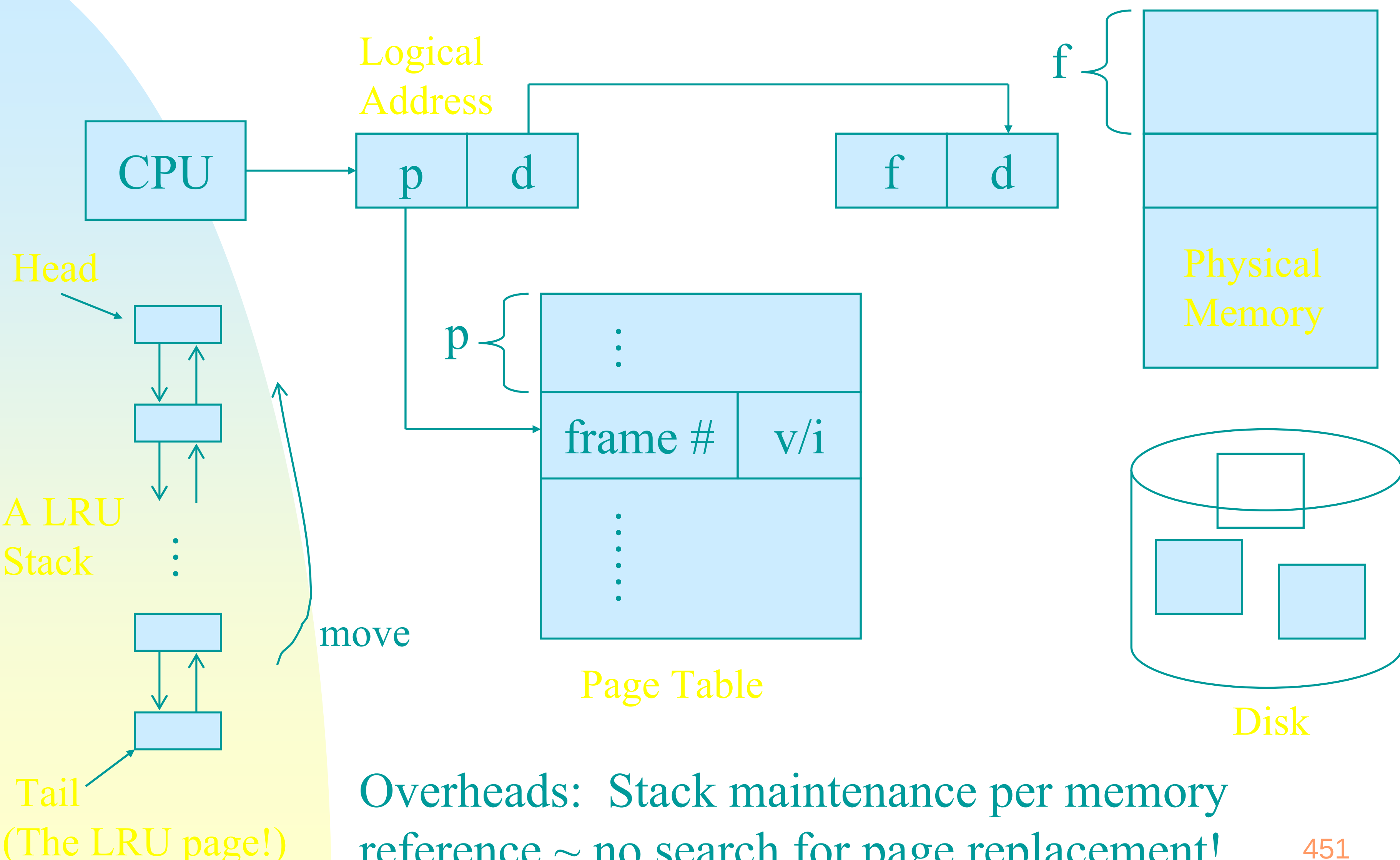
LRU Implementation – Counters



LRU Implementation – Counters

- Overheads
  - The logical clock is incremented for every memory reference.
  - Update the “time-of-use” field for each page reference.
  - Search the LRU page for replacement.
  - Overflow prevention of the clock & the maintenance of the “time-of-use” field of each page table.

LRU Implementation – Stack



Overheads: Stack maintenance per memory reference ~ no search for page replacement!

A Stack Algorithm

$$\left\{\begin{array}{l}\text{memory-}\\\text{resident}\\\text{pages}\end{array}\right\}_{\substack{\text{n frames}\\\text{available}}} \subseteq \left\{\begin{array}{l}\text{memory-}\\\text{resident}\\\text{pages}\end{array}\right\}_{\substack{\text{(n +1) frames}\\\text{available}}}$$

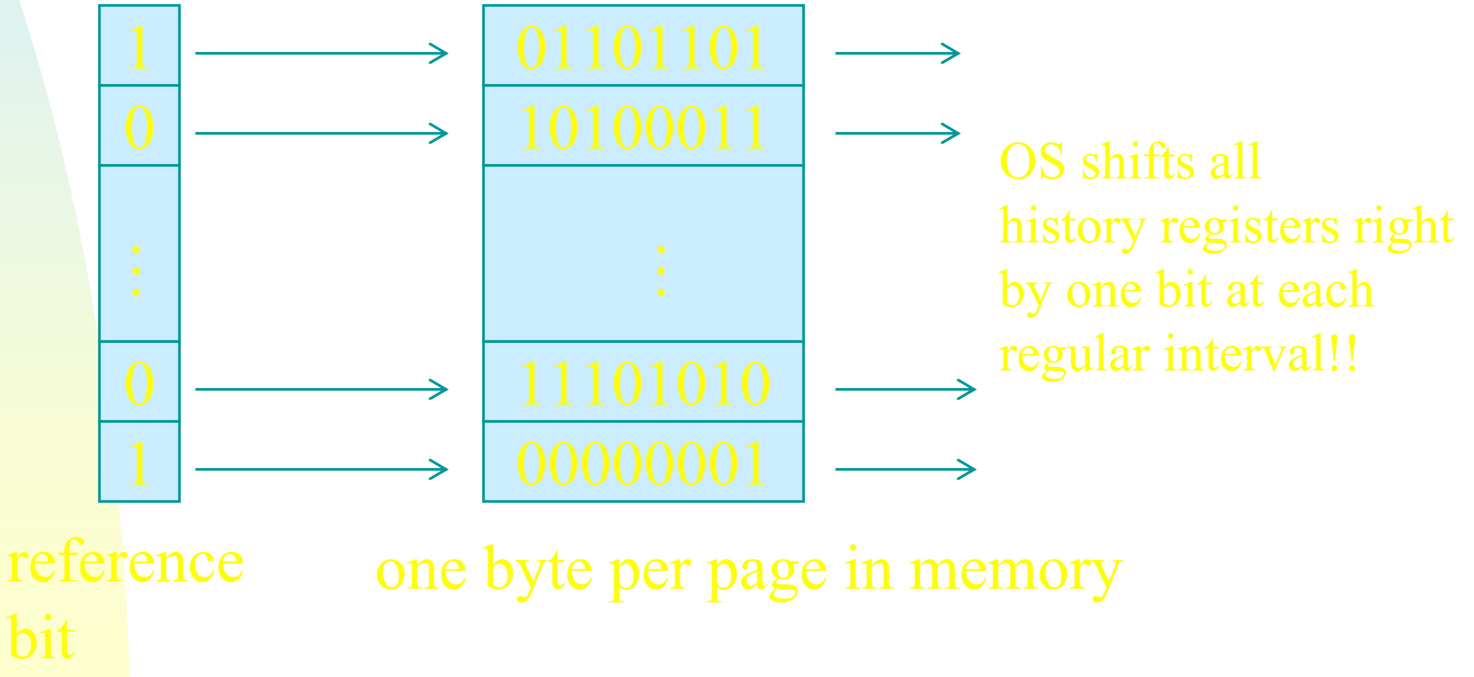
- Need hardware support for efficient implementations.
- Note that LRU maintenance needs to be done for every memory reference.

LRU Approximation Algorithms

- Motivation
  - No sufficient hardware support
  - Most systems provide only “reference bit” which only indicates whether a page is used or not, instead of their order.
- Additional-Reference-Bit Algorithm
- Second-Chance Algorithm
- Enhanced Second Chance Algorithm
- Counting-Based Page Replacement

Additional-Reference-Bits Algorithm

- Motivation
  - Keep a history of reference bits



Additional-Reference-Bits Algorithm

- History Registers

LRU
(smaller value!)

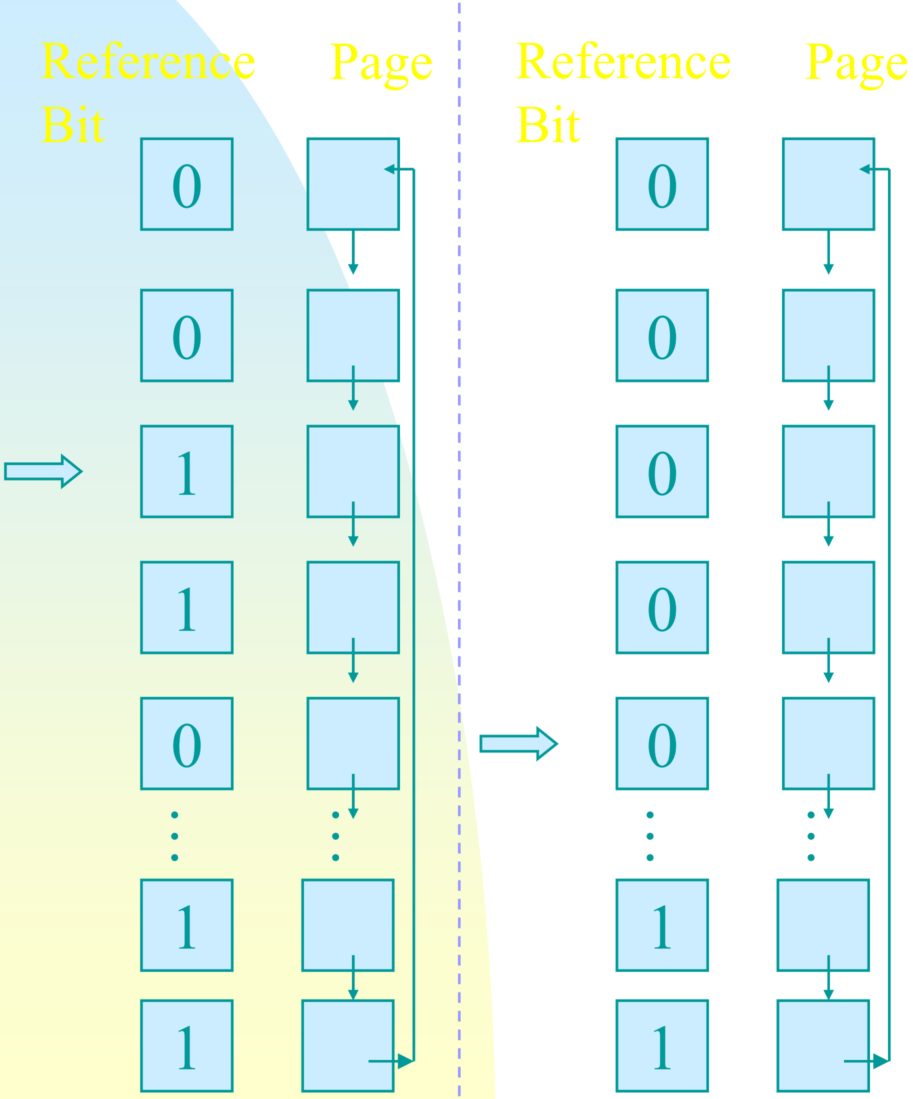
0 0 0 0 0 0 0 0 ← Not used for 8 times
0 0 0 0 0 0 0 1

1 1 1 1 1 1 1 0
1 1 1 1 1 1 1 1 ← Used at least once every time

MRU

- But, how many bits per history register should be used?
  - Fast but cost-effective!
  - The more bits, the better the approximation is.

Second-Chance (Clock) Algorithm



- Motivation
  - Use the reference bit only
- Basic Data Structure:
  - Circular FIFO Queue
- Basic Mechanism
  - When a page is selected
    - Take it as a victim if its reference bit = 0
    - Otherwise, clear the bit and advance to the next page

Enhanced Second-Chance Algorithm

- Motivation:
  - Consider the cost in swapping out” pages.
- 4 Classes (reference bit, modify bit)
  - (0,0) – not recently used and not “dirty”
  - (0,1) – not recently used but “dirty”
  - (1,0) – recently used but not “dirty”
  - (1,1) – recently used and “dirty”

low priority

high priority

Enhanced Second-Chance Algorithm

- Use the second-chance algorithm to replace the first page encountered in the lowest nonempty class.

  => May have to scan the circular queue several times before find the right page.

- Macintosh Virtual Memory Management

Counting-Based Algorithms

- Motivation:
  - Count the # of references made to each page, instead of their referencing times.
- Least Frequently Used Algorithm (LFU)
  - LFU pages are less actively used pages!
  - Potential Hazard: Some heavily used pages may no longer be used !
  - A Solution – Aging
    - Shift counters right by one bit at each regular interval.

Counting-Based Algorithms

- Most Frequently Used Algorithm (MFU)
  - Pages with the smallest number of references are probably just brought in and has yet to be used!
- LFU & MFU replacement schemes can be fairly expensive!
- They do not approximate OPT very well!

Page Buffering

- Basic Idea
  a. Systems keep a pool of free frames
  b. Desired pages are first "swapped in" some pages in the pool.
  c. When the selected page (victim) is later written out, its frame is returned to the pool.
- Variation 1
  a. Maintain a list of modified pages.
  b. Whenever the paging device is idle, a modified page is written out and reset its "modify bit".

Page Buffering

- Variation 2
  a. Remember which page was in each frame of the pool.
  b. When a page fault occurs, first check whether the desired page is there already.
     - Pages which were in frames of the pool must be “clean”.
     - “Swapping-in” time is saved!
- VAX/VMS with the FIFO replacement algorithm adopt it to improve the performance of the FIFO algorithm.

Frame Allocation – Single User

- Basic Strategy:
  - User process is allocated any free frame.
  - User process requests free frames from the free-frame list.
  - When the free-frame list is exhausted, page replacement takes place.
  - All allocated frames are released by the ending process.
- Variations
  - O.S. can share with users some free frames for special purposes.
  - Page Buffering - Frames to save “swapping” time

Frame Allocation – Multiple Users

- Fixed Allocation
  a. Equal Allocation

     m frames, n processes → m/n frames per process

  b. Proportional Allocation

     1. Ratios of Frames $\propto$ Size

        $S = \Sigma S_i$, $A_i \propto (S_i / S) \times m$, where (sum <= m) & ($A_i$ >= minimum # of frames required)

     2. Ratios of Frames $\propto$ Priority

        S_i : relative importance

     3. Combinations, or others.

Frame Allocation – Multiple Users

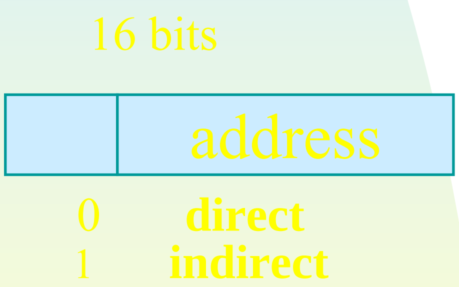
- Dynamic Allocation
  - a. Allocated frames $\propto$ the multiprogramming level
  - b. Allocated frames $\propto$ Others
- The minimum number of frames required for a process is determined by the instruction-set architecture.
  - ADD A,B,C → 4 frames needed
  - ADD (A), (B), (C) → 1+2+2+2 = 7 frames, where (A) is an indirect addressing.

Frame Allocation – Multiple Users

- Minimum Number of Frames (Continued)
  - How many levels of indirect addressing should be supported?
    - It may touch every page in the logical address space of a process

    => Virtual memory is collapsing!
  - A long instruction may cross a page boundary.

    MVC X, Y, 256 → 2 + 2 + 2 = 6 frames
    - The spanning of the instruction and the operands.



Frame Allocation – Multiple Users

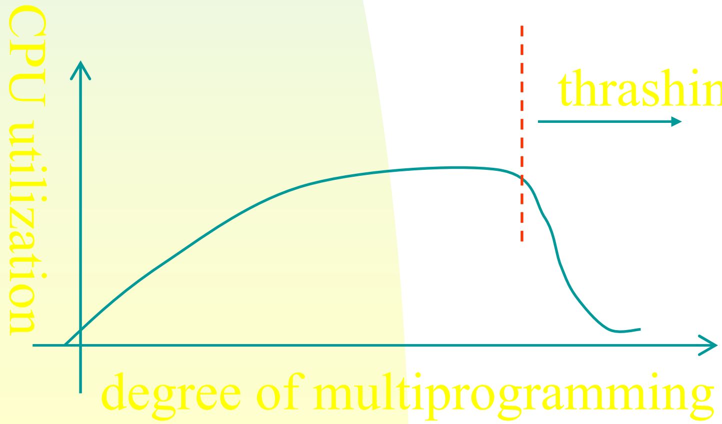
- Global Allocation
  - Processes can take frames from others. For example, high-priority processes can increase its frame allocation at the expense of the low-priority processes!
- Local Allocation
  - Processes can only select frames from their own allocated frames → Fixed Allocation
  - The set of pages in memory for a process is affected by the paging behavior of only that process.

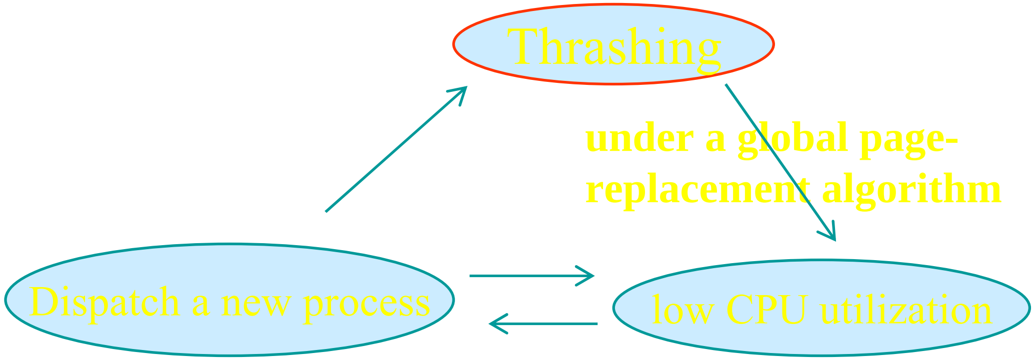
Frame Allocation – Multiple Users

- Remarks
  a. Global replacement generally results in a better system throughput
  b. Processes can not control their own page fault rates such that a process can affect each another easily.

Thrashing

- Thrashing – A High Paging Activity:
  - A process is thrashing if it is spending more time paging than executing.
- Why thrashing?
  - Too few frames allocated to a process!

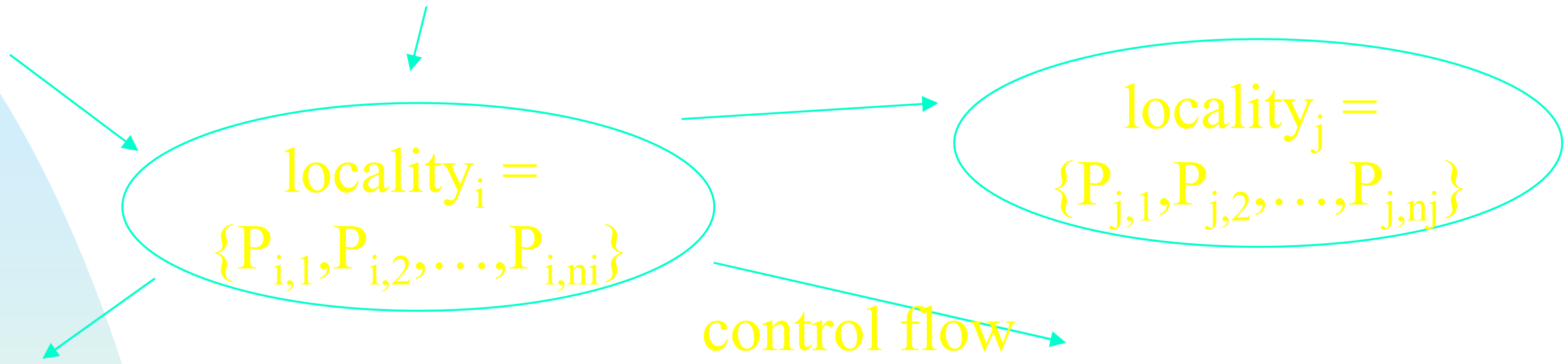




Thrashing

- Solutions:
  - Decrease the multiprogramming level → Swap out processes!
  - Use local page-replacement algorithms
    - Only limit thrashing effects "locally"
    - Page faults of other processes also slow down.
  - Give processes as many frames as they need!
    - But, how do you know the right number of frames for a process?

Locality Model



- A program is composed of several different (overlapped) localities.
  - Localities are defined by the program structures and data structures (e.g., an array, hash tables)
- How do we know that we allocate enough frames to a process to accommodate its current locality?

Working-Set Model

Page references

…2 6 1 5 7 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4

$\longleftarrow \Delta \longrightarrow$ working-set window t1

$\longleftarrow \Delta \longrightarrow$ working-set window t2

working-set(t1) = {1,2,5,6,7}

working-set(t2) = {3,4}

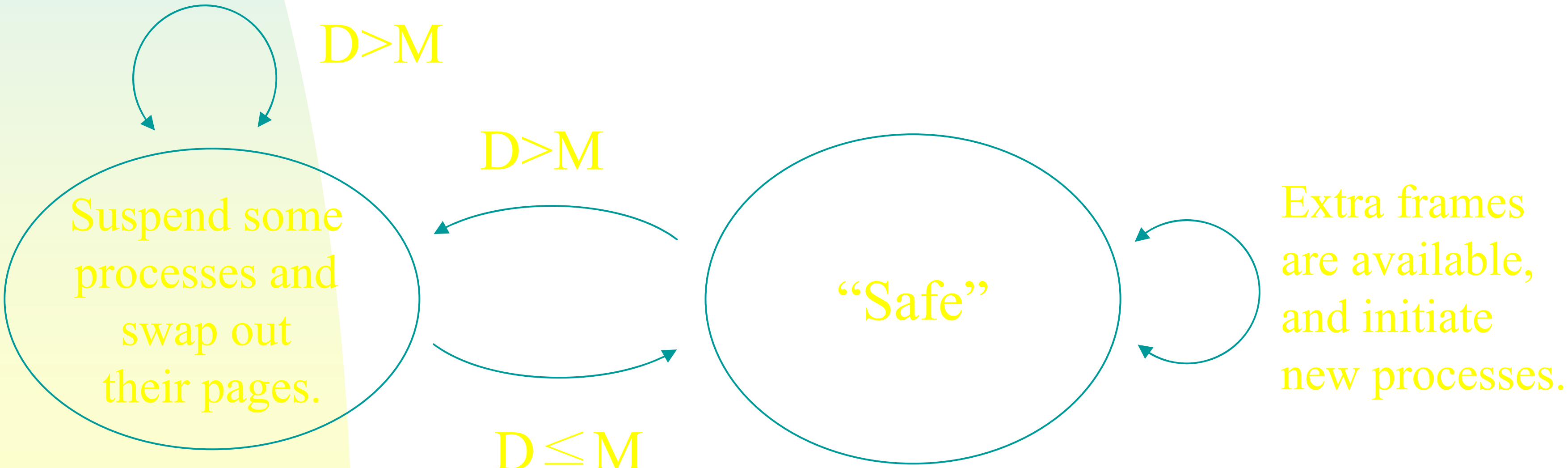
- The working set is an approximation of a program’s locality.

The minimum allocation $\Delta \longleftarrow\longrightarrow \infty$ All touched pages may cover several localities.

Working-Set Model

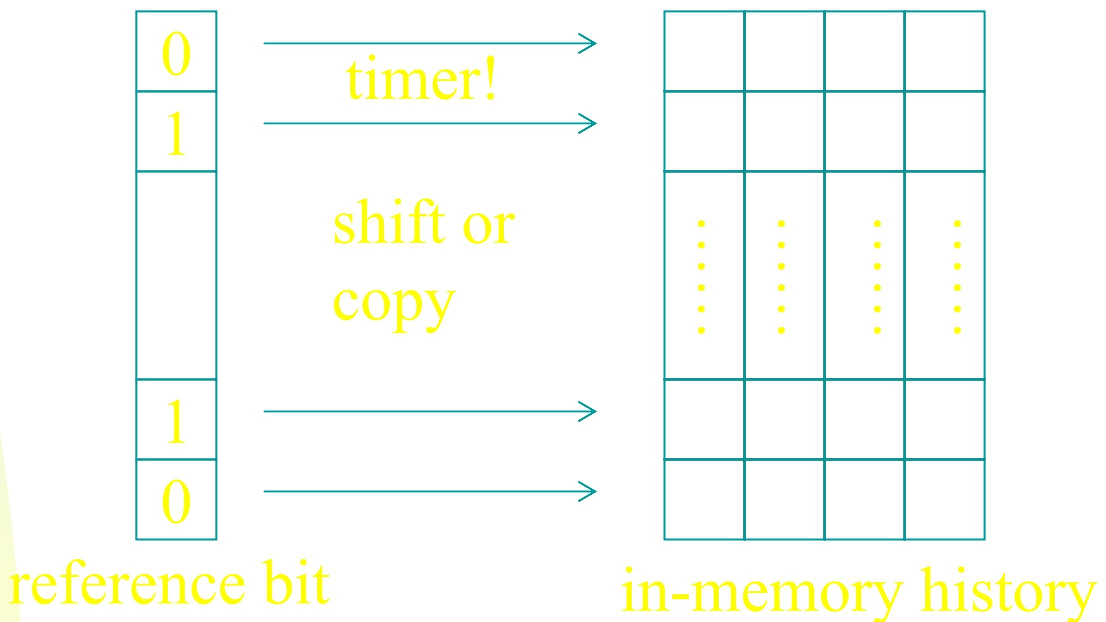
$$D = \sum working - set - size_i \leq M$$

where M is the total number of available frames.



Working-Set Model

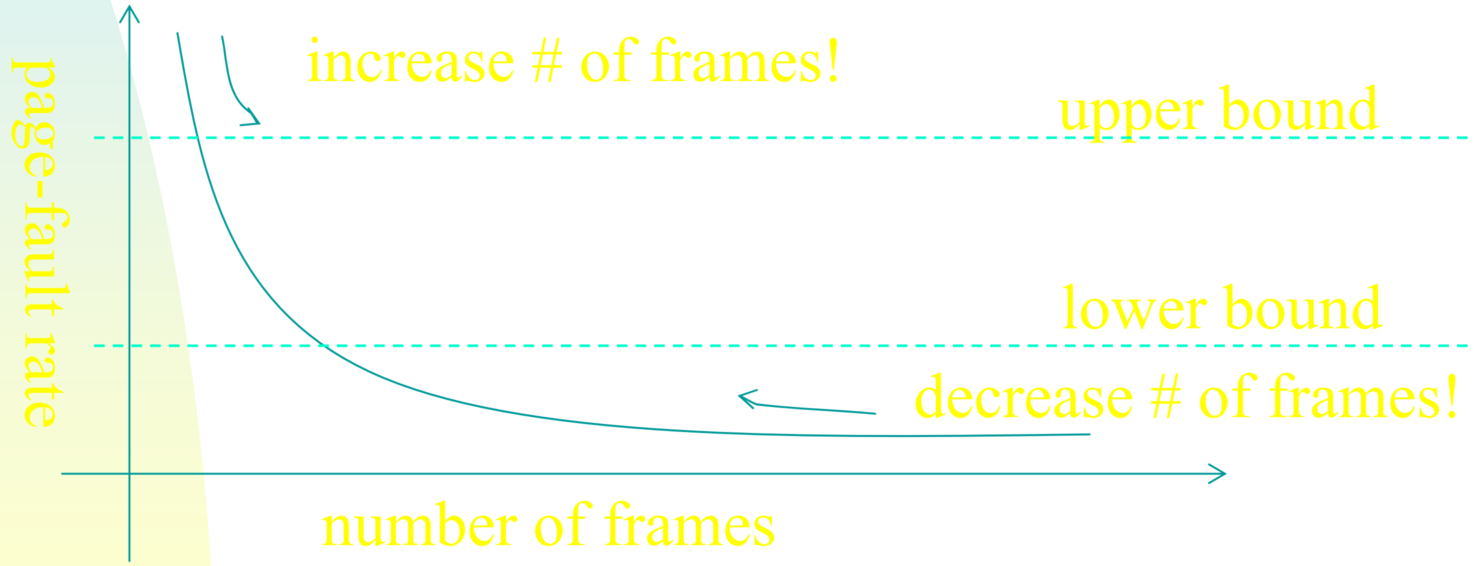
- The maintenance of working sets is expensive!
  - Approximation by a timer and the reference bit



  - Accuracy v.s. Timeout Interval!

Page-Fault Frequency

- Motivation
  - Control thrashing directly through the observation on the page-fault rate!

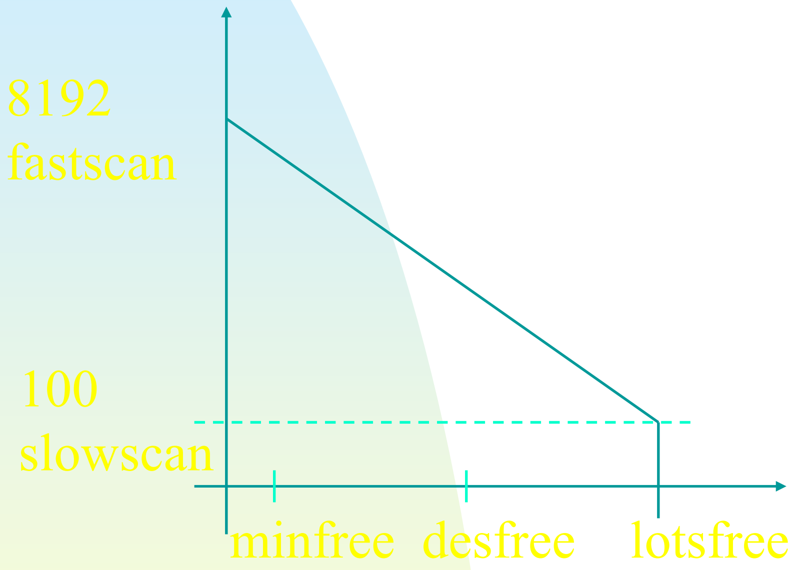


*Processes are suspended and swapped out if the number of available frames is reduced to that under the minimum needs.

OS Examples – NT

- Virtual Memory – Demand Paging with Clustering
  - Clustering brings in more pages surrounding the faulting page!
- Working Set
  - A Min and Max bounds for a process
    - Local page replacement when the max number of frames are allocated.
  - Automatic working-set trimming reduces allocated frames of a process to its min when the system threshold on the available frames is reached.

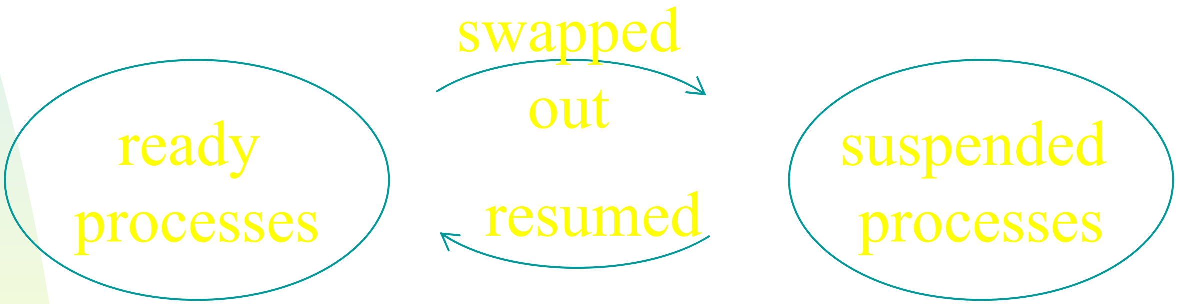
OS Examples – Solaris



- Process *pageout* first clears the reference bit of all pages to 0 and then later returns all pages with the reference bit = 0 to the system *(handspread).*
  - 4HZ → 100HZ when *desfree* is reached!
    - Swapping starts when *desfree* fails for 30s.
  - *pageout* runs for every request to a new page when *minfree* is reached.

Other Considerations

- Pre-Paging
  - Bring into memory at one time all the pages that will be needed!



Do pre-paging if the working set is known!

- Issue

Pre-Paging Cost $\longleftrightarrow$ Cost of Page Fault Services

Not every page in the working set will be used!

Other Considerations

- Page Size

Better Resolution for Locality & Internal Fragmentation

Page Size

small | p | d | large

$512B(2^9) \sim 16{,}384B(2^{12})$

Smaller Page Table Size & Better I/O Efficiency

- Trends - Large Page Size

∵ The CPU speed and the memory capacity grow much faster than the disk speed!

Other Considerations

- TLB Reach
  - TLB-Entry-Number * Page-Size
- Wish
  - The working set is stored in the TLB!
  - Solutions
    - Increase the page size
    - Have multiple page sizes – UltraSparc II (8KB - 4MB) + Solaris 2 (8KB or 4MB)

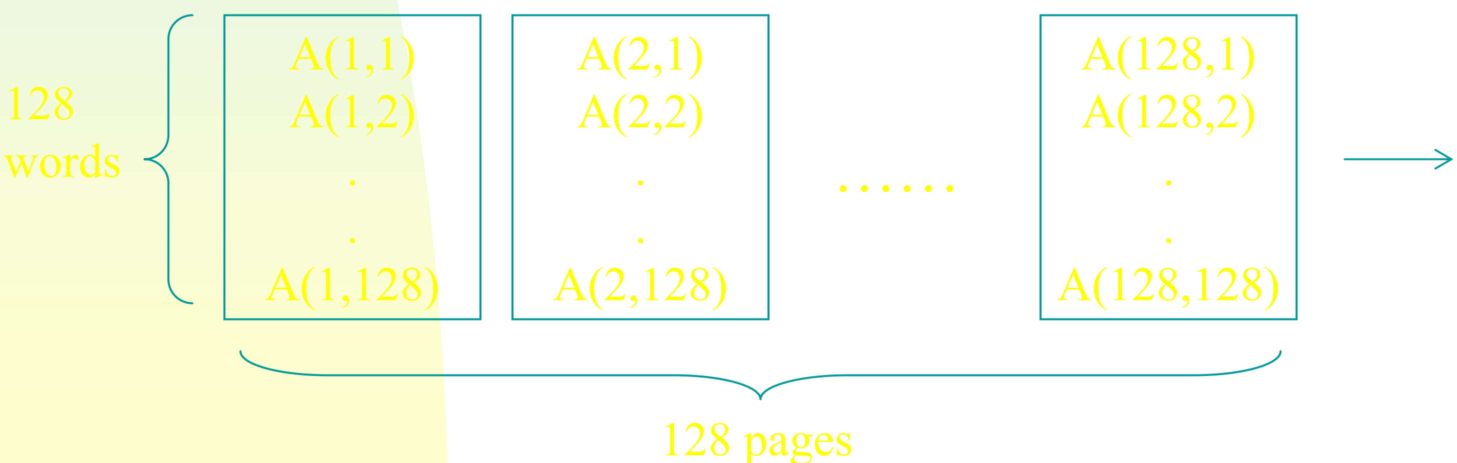
Other Considerations

- Inverted Page Table
  - The objective is to reduce the amount of physical memory for page tables, but they are needed when a page fault occurs!
  - More page faults for page tables will occur!!!

Other Considerations

- Program Structure
  - Motivation – Improve the system performance by an awareness of the underlying demand paging.

```
var A: array [1..128,1..128] of integer;
    for j:=1 to 128
        for i:=1 to 128
            A(i,j):=0
```

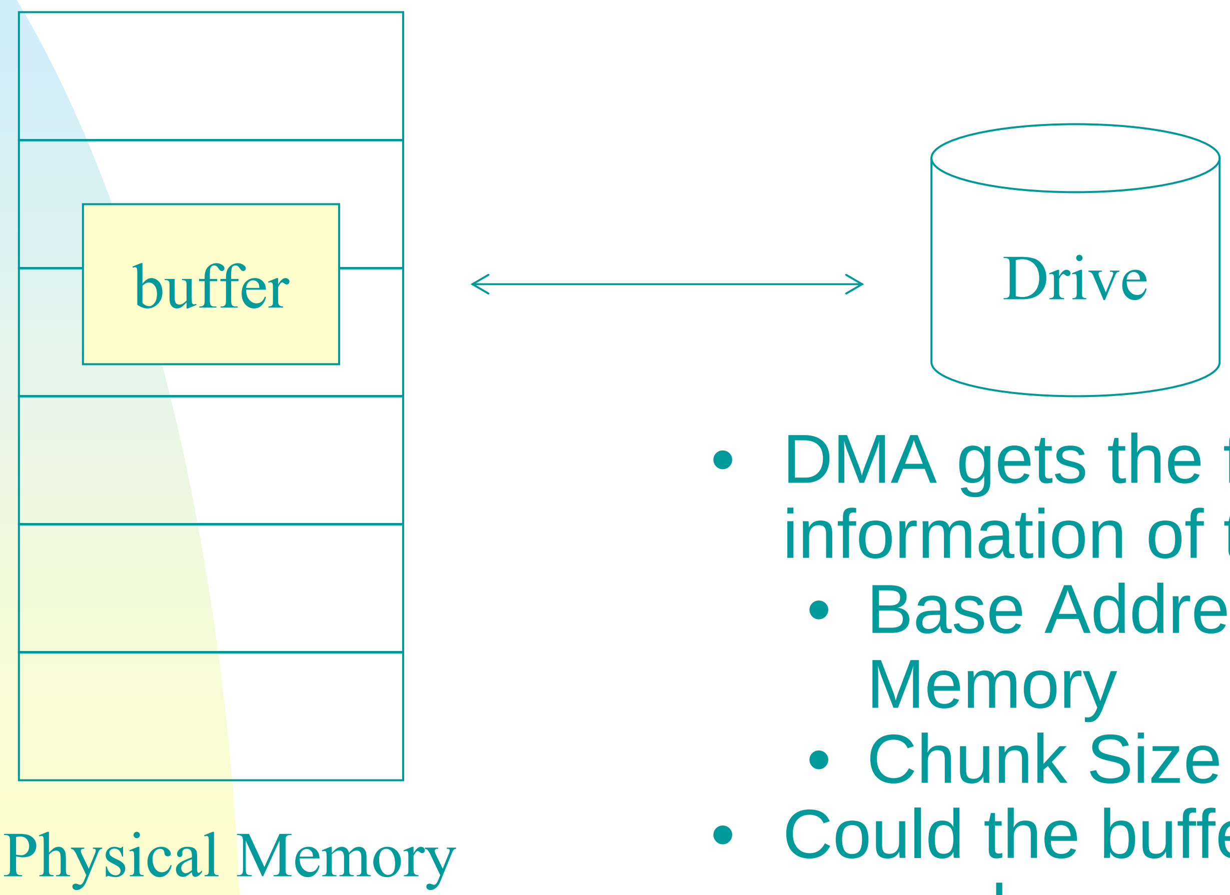


128x128 page faults if the process has less than 128 frames!!

Other Considerations

- Program Structures:
  - Data Structures
    - Locality: stack, hash table, etc.
    - Search speed, # of memory references, # of pages touched, etc.
  - Programming Language
    - Lisp, PASCAL, etc.
  - Compiler & Loader
    - Separate code and data
    - Pack inter-related routines into the same page
    - Routine placement (across page boundary?)

I/O Interlock



- DMA gets the following information of the buffer:
  - Base Address in Memory
  - Chunk Size
- Could the buffer-residing pages be swapped out?

I/O Interlock

- Solutions
  - I/O Device ←→ System Memory ←→ User Memory
    - Extra Data Copying!!
  - Lock pages into memory
    - The lock bit of a page-faulting page is set until the faulting process is dispatched!
    - Lock bits might never be turned off!
    - Multi-user systems usually take locks as “hints” only!

Real-Time Processing

Predictable Behavior ⟷ Virtual memory introduces unexpected, long-term delays in the execution of a program.

- Solution:
  - Go beyond locking hints ➔ Allow privileged users to require pages being locked into memory!

Demand Segmentation

- Motivation
  - Segmentation captures better the logical structure of a process!
  - Demand paging needs a significant amount of hardware!
- Mechanism
  - Like demand paging!
  - However, compaction may be needed!
    - Considerable overheads!