

Lecture Notes
Operating Systems

CS 4560/6560

Instructor: E. Billard

Operating Systems: Outline 4560/6560* [2]

I. Introduction	Ch. 1,3
II. Queueing Models	
III. Process Management	Ch. 4,5,6
IV. Deadlock*	Ch. 7
V. Memory Management	Ch. 8, 9
VI. File Management	Ch. 10,11
VII. Distributed Systems	Ch. 15,16,17,18
VIII. Obsolete	
IX. Obsolete	
X. Design of Java*	David Flanagan
XI. Design of XINU*	Douglas Comer
XII. Design of UNIX*	Ch. 19
XIII. Design of Mach*	Ch. 20

Text: *Operating Systems Concepts* - Silberschatz, Galvin

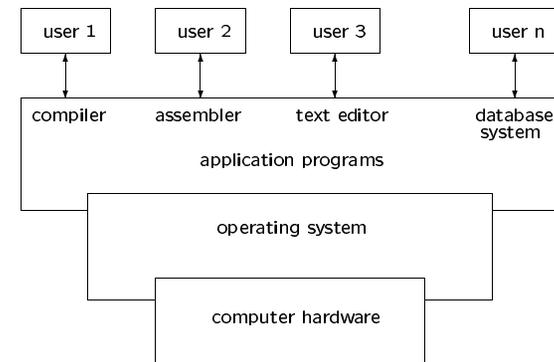
Text*: *Java in a Nutshell* - Flanagan

Text*: *Operating System Design: The XINU Approach* - Comer, Fossum

[1] Introduction to OS [3]

- **operating system** provides environment for program execution
- primary goal: make the computer **convenient** to use
- secondary goal: use the hardware in an **efficient** manner.
- OS is **intermediary** between user (and applications) and H/W
- OS is a **manager** and **allocator** of **resources**
 - must allocate resources **efficiently** and **fairly**
- OS is a **control program**
 - controls the execution of user programs
 - prevents errors and improper use of the computer
- OS is the **one program** running at all times on the computer

[1] Intro: Abstract View of OS as Intermediary [4]



[1] Intro: Early Systems

[5]

- **large** and **expensive** machines run from a console
- **programmer** was also the **operator**
- many separate steps:
 - loading/unloading of magnetic and paper tapes, punch cards
- while tapes were being mounted:
 - CPU sat idle - no work was being done

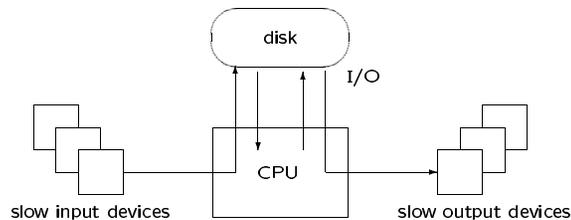
[1] Intro: Evolutionary Steps

[6]

- professional operator was hired
- jobs with similar needs were **batched** together and run as a group
- **resident monitors** transferred control from job to job
- overlapped CPU and I/O operations were introduced
- replaced card readers, line printers with magnetic-type units
- readers and printers were operated **off-line**
- replaced tapes (**sequential-access**) with disks (**random-access**)
- introduced simultaneous peripheral operation on-line (**spooling**)

[1] Intro: Spooling

[7]

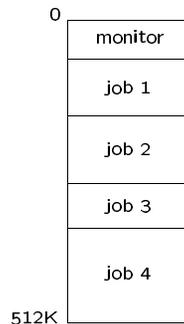


[1] Intro: Multiprogramming

[8]

- increases CPU **utilization**
- organizes jobs so that CPU always has something to execute
- OS keeps several jobs in memory
- OS picks and begins to execute one of the jobs
- OS switches to/executes another job when current job waits
 - when a program does I/O, the CPU works on another program
- all the jobs are kept in the job pool
 - on disk awaiting allocation of main memory
- if not enough memory, OS must choose (**job scheduling**)
- OS allocates memory (**memory management**)
- OS allocates CPU to one process (**CPU scheduling**)

[1] Intro: Memory Layout for Multiprogramming [9]



[1] Intro: Batch Systems [10]

- **multiprogrammed** but do **not** permit interaction with the user
 - appropriate for executing large jobs that need little interaction
- But**
- user provides commands to handle job steps (may be dependent)
 - programs must be debugged statically

[1] Intro: Time-Sharing Systems [10]

- **multiprogrammed** and do allow interaction with the user
- switch rapidly from one user to the next
 - each user has impression of their own computer
- programs can be larger than physical memory (**virtual memory**)
- programs interact with user (short time before it needs to do I/O)

[1] Intro: Process Management [11]

- OS manages processes, memory, files, networking
- a **process** is **program in execution**
- OS is responsible for process management:
 - creation/deletion of both user and system processes
 - suspension/resumption of processes
 - process synchronization, communication and deadlock handling

[1] Intro: Memory Management [11]

- CPU directly addresses main memory (and no other storage device)
- OS is responsible for memory management:
 - which parts of memory are being used and by whom
 - which processes are to be loaded into memory
 - allocate/deallocate memory space as needed

[1] Intro: Secondary-Storage Management [12]

- main memory is too small for all data and programs
- info lost when power is lost
- secondary storage (disk) is used to back up main memory
- OS is responsible for disk management:
 - item free-space management
 - storage allocation
 - disk scheduling

[I] Intro: File Management [13]

- info stored on several different types of physical media
- OS provides a uniform logical view of info (unit of storage: **file**)
- OS is responsible for file management:
 - creation/deletion of files and directories
 - support of primitives for manipulating files and directories
 - mapping of files onto secondary storage
 - backup of files on stable (nonvolatile) storage media

[I] Intro: Network Management [13]

- a **distributed** system collects
 - physically separate, possibly heterogeneous systems
 - into a single coherent system
- processors in system are connected by **communication network**
- LANs (local-area networks) and WANs (wide-area networks)
- OS generalizes network access as a form of file access

[I] Intro: OS Services [14]

- OS provides services to programs and users:
 - program execution and I/O operations
 - file-system manipulation
 - communications
 - error detection
 - resource allocation
 - accounting and protection

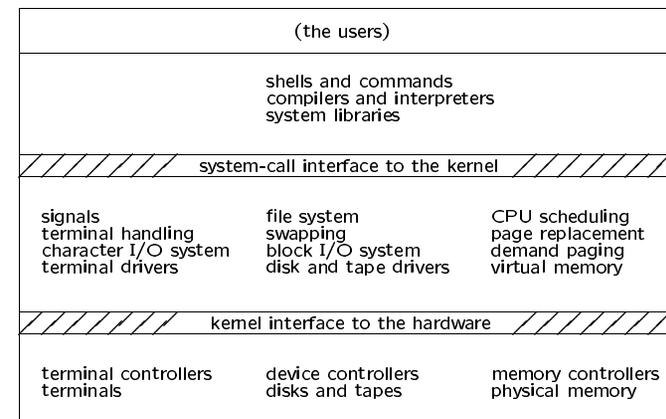
[I] Intro: System Calls [14]

- system calls: interface between running program and OS
 - process control
 - file manipulation
 - device manipulation
 - information maintenance
 - communications

[I] Intro: System Programs [15]

- convenient environment for program development/execution:
 - file manipulation and modification
 - status information
 - programming-language support
 - program loading and execution
 - communications
 - application programs

[I] Intro: UNIX Limited Structure [16]



[I] Intro: VENUS Layered Structure

[17]

layer 6:	user programs
layer 5:	device drivers and schedulers
layer 4:	virtual memory
layer 3:	I/O channel
layer 2:	CPU scheduling
layer 1:	instruction interpreter
layer 0:	hardware

- OS are complex and require a layered solution
- increases **modularity**
- a function only uses functions of lower-level layers

[II] Queues: Outline

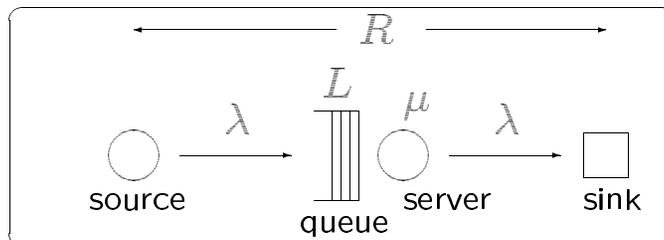
[18]

- Model of Single Queue [source (λ), queue (L), server (μ), sink]
- Performance Metrics
 - [utilization (ρ), response time (R), waiting time (W), throughput (λ), number (Q)]
- Queueing Networks [transition probability (p_{ij})]
- Operational Laws
- Stream Law: $\lambda_i = \sum p_{ji} \lambda_j$
- Utilization Law: $\rho_i = \lambda_i / \mu_i$
- Visit (Forced Flow) Law: $V_i = \lambda_i / \lambda$
- Bottleneck Law: $\lambda_{max} = \mu_b / V_b$
- Little's Law: $Q = \lambda R$
- M/M/1 Law: $R = 1 / (\mu - \lambda)$
- General Response Time Law: $R = \sum R_i V_i$
- Case Study: CPU vs. I/O

[II] Queues: Single Queue

[19]

model



source

- entry for new jobs at **arrival rate**: λ jobs/sec

queue

- waiting line of jobs with **queue length**: L jobs

server

- processor of jobs at **service rate**: μ jobs/sec

sink

- exit for finished jobs with **throughput**: λ jobs/sec

response

- time waiting in queue and in service: R sec

[II] Queues: Performance Metrics

[20]

ratio

- **utilization** of server: $\rho = \lambda / \mu$ % busy

ratio

- **traffic intensity**=utilization: $\rho = \lambda / \mu$

rate

- **throughput**: λ jobs/sec (**job flow balance**)

jobs

- **number** in system (queue and service): Q

time

- **waiting time** in queue: W sec

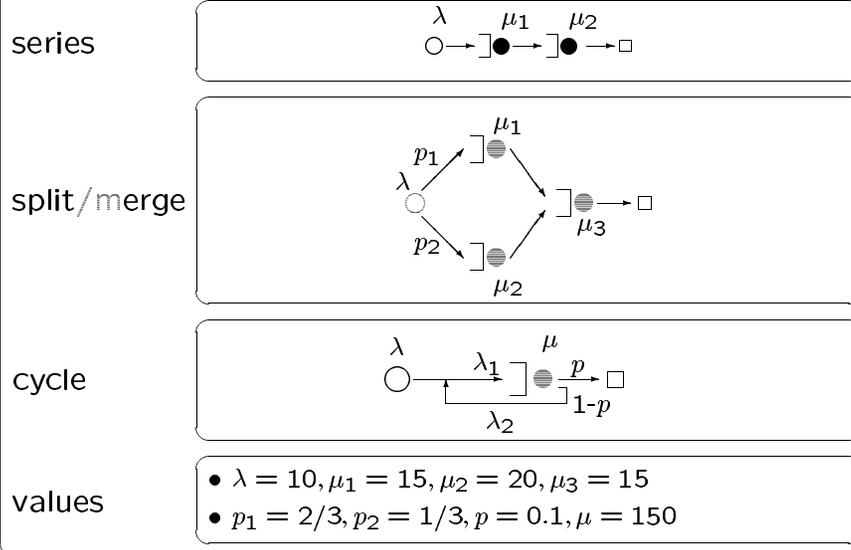
time

- **service time** per visit to server: $S = 1 / \mu$ sec/job

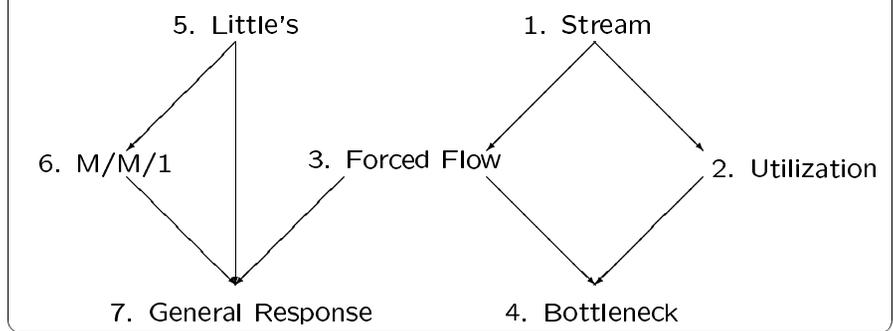
time

- **response time**: $R = W + S = W + 1 / \mu$ sec/job

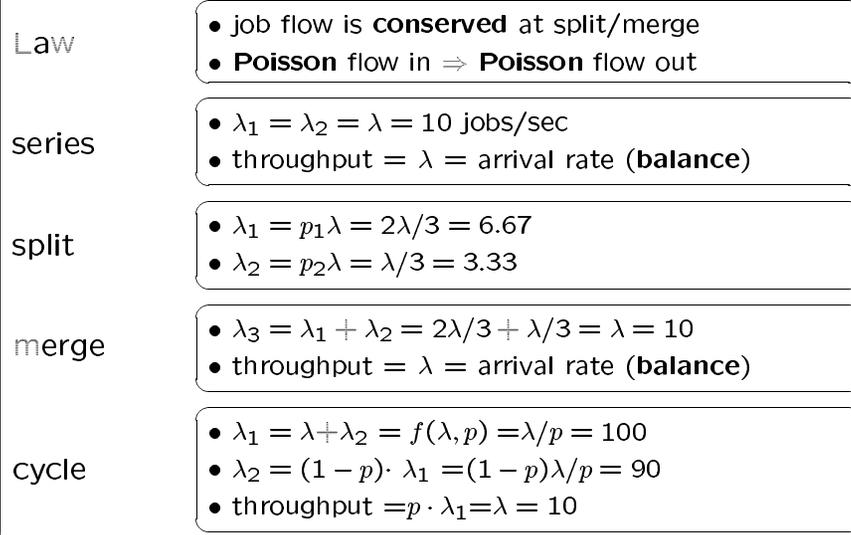
[II] Networks: probability p_{ij} from device i to j [21]



[II] Queues: Operational Laws [22]



[II] Queue Stream Law: $\lambda_i = \sum p_{ji} \lambda_j$ [23]



[II] VISUAL Queues: Sample Output [24]

SERIES:								
Time =	1000.00	(n)	(rho)	(Q)	(lambda)	(R)	(p _{ij})	
Type	Rate	# Jobs	% Busy	Avg jobs	Job rate	Job time	Prob.	
1	source	10.0	10030	83.21	3.031	10.030	0.302197	1.00
2	fifo	15.0	10030	67.31	2.070	10.030	0.206351	1.00
3	fifo	20.0	10030	49.19	0.961	10.028	0.095828	1.00
4	sink	0.0	10028	83.21	3.030	10.028	0.302197	
SPLIT/MERGE								
1	source	10.0	9994	82.69	2.902	9.994	0.290346	0.67 0.33
2	fifo	15.0	6717	44.16	0.783	6.717	0.116622	1.00
3	fifo	20.0	3277	16.42	0.198	3.277	0.060310	1.00
4	fifo	15.0	9994	66.07	1.920	9.992	0.192176	1.00
5	sink	0.0	9992	82.69	2.901	9.992	0.290346	
CYCLE:								
1	source	10.0	10112	58.51	2.170	10.112	0.214624	1.00
2	fifo	150.0	101138	67.59	2.170	101.136	0.021456	0.90 0.10
3	sink	0.0	10110	58.51	2.170	10.110	0.214624	

[II] Queue Utilization Law: $\rho_i = \lambda_i / \mu_i$ [25]

Law	<ul style="list-style-type: none"> utilization depends on flow and service rate
series	<ul style="list-style-type: none"> $\rho_1 = \lambda_1 / \mu_1 = \lambda / \mu_1 = 67\%$ busy $\rho_2 = \lambda_2 / \mu_2 = \lambda / \mu_2 = 50\%$
split/merge	<ul style="list-style-type: none"> $\rho_1 = \lambda_1 / \mu_1 = 2\lambda / 3\mu_1 = 44\%$ $\rho_2 = \lambda_2 / \mu_2 = \lambda / 3\mu_2 = 17\%$ $\rho_3 = \lambda_3 / \mu_3 = \lambda / \mu_3 = 67\%$
cycle	<ul style="list-style-type: none"> $\rho = \lambda_1 / \mu = \lambda / p\mu = 67\%$

[II] Queue Visit Law: $V_i = \lambda_i / \lambda$ [26]

Law	<ul style="list-style-type: none"> visit ratio to server is the same as the flow ratio measures fraction of jobs which visit server
series	<ul style="list-style-type: none"> $V_1 = \lambda_1 / \lambda = \lambda / \lambda = 1$ (all jobs visit once) $V_2 = \lambda_2 / \lambda = \lambda / \lambda = 1$
split/merge	<ul style="list-style-type: none"> $V_1 = \lambda_1 / \lambda = 2/3$ (some jobs visit) $V_2 = \lambda_2 / \lambda = 1/3$ $V_3 = \lambda_3 / \lambda = \lambda / \lambda = 1$
cycle	<ul style="list-style-type: none"> $V = \lambda_1 / \lambda = 1/p = 10$ (repeat visits)

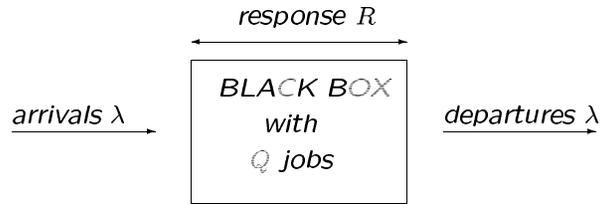
[II] Queue Bottleneck Law: $\lambda_{max} = \mu_b / V_b$ [27]

Law	<ul style="list-style-type: none"> bottleneck is server with highest utilization: ρ_{max} bottleneck limits maximum throughput: λ_{max}
step 1	<ul style="list-style-type: none"> Utilization Law: find bottleneck b with ρ_{max} bottleneck b service rate: μ_b
step 2	<ul style="list-style-type: none"> Visit Law: find $V_b = \lambda_b / \lambda$ maximum system throughput: $\lambda_{max} = \mu_b / V_b$
proof	<ul style="list-style-type: none"> stable system: $\lambda_i \leq \mu_i$ bottleneck: $\rho_{max} = \lambda_b / \mu_b \Rightarrow$ closest λ to μ Visit Law: $\lambda_i = \lambda V_i$ bottleneck: $\lambda_b = \lambda V_b \leq \mu_b$ maximum system throughput: $\lambda_{max} V_b \leq \mu_b$ $\lambda_{max} = \mu_b / V_b$

[II] Queue Bottleneck Law: $\lambda_{max} = \mu_b / V_b$ [28]

series	<ul style="list-style-type: none"> $\rho_{max} = 67\%$ at bottleneck $b = 1$ $\lambda_{max} = \mu_1 / V_1 = 15/1 = 15$ jobs/sec
split/merge	<ul style="list-style-type: none"> $\rho_{max} = 67\%$ at bottleneck $b = 3$ $\lambda_{max} = \mu_3 / V_3 = 15/1 = 15$
cycle	<ul style="list-style-type: none"> $\rho_{max} = 67\%$ $\lambda_{max} = \mu / V = p\mu = 0.1 \cdot 150 = 15$
goal	<ul style="list-style-type: none"> improve throughput by fixing the bottleneck b <i>minimize the maximum utilization</i> then fix next bottleneck b_1
split/merge	<ul style="list-style-type: none"> reduce bottleneck $b = 3$ to the same as $\rho_1 = 44\%$ $\rho_3 = \lambda / \mu_3 = 44\% \Rightarrow \mu_3 = \lambda / 44\% = 22.7$

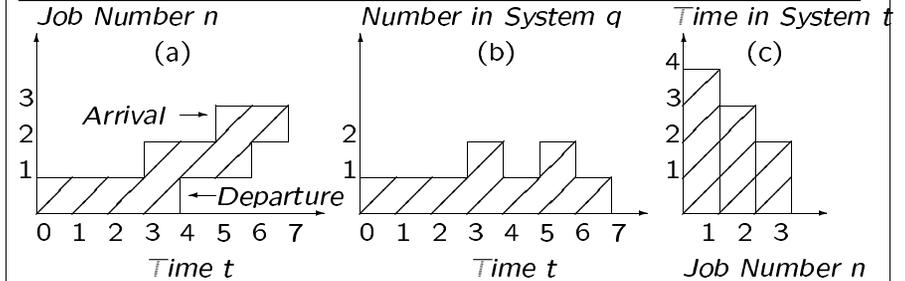
[II] Little's Law: $Q = \lambda R$ [29]



Little's Law

- mean number = arrival rate x mean response
- queue and server: $Q = \lambda R$
- subsystem in network: $Q_i = \lambda_i R_i$
- network system: $Q_{sys} = \lambda_{sys} R_{sys}$
- queue: queue length $L = \lambda W$ (waiting time)
- **assumption:** arrivals = departures
- **job flow balance**

[II] Little's Law: $Q = \lambda R$ [30]



rules

- (a) $y_{arrival} - y_{departure} = (b)$
- (a) $x_{departure} - x_{arrival} = (c)$
- (a) Area = (b) Area = (c) Area = $J = 9$

Little's Law

- (b) mean number in system $Q = \frac{J}{T} = \frac{N}{T} \times \frac{J}{N} = \lambda R$
- $Q =$ arrival rate x mean time in system
- $Q = \lambda R$: $3/7 \times 9/3 = 9/7$

[II] Little's Law: $Q = \lambda R$ [31]

series

- given: $R_1 = 0.2, R_2 = 0.1$ secs
- $Q_1 = \lambda R_1 = (10)(0.2) = 2$ jobs
- $Q_2 = (10)(0.1) = 1$
- $R_{sys} = 0.2 + 0.1 = 0.3$ secs
- $Q_{sys} = \lambda R_{sys} = (10)(0.3) = 3 = 2 + 1$ jobs

split/merge

- given waiting time in queue: $W_1 = 0.05$ secs
- $L_1 = \lambda_1 W_1 = (6.67)(0.05) = 0.33$ jobs

cycle

- given: $R = 0.02$
- $Q = \lambda_1 R = (10/0.1)(0.02) = 2$ jobs

[II] Queue M/M/1 Law: $R = 1/(\mu - \lambda)$ [32]

M/M/c

- **Markov=Poisson=memoryless**
- Markov arrivals at rate λ
- Markov service at rate μ
- c parallel servers connected to one queue
- high probability of Δ jobs = ± 1 in small time frame

M/M/1

- Markov arrivals, service to a queue with 1 server

memory-less (Markov)

- time until next event does not depend on the time since the last event
- if long time since job arrival, the expected time until the next job is still $1/\lambda$
- if 1 hr since last job, expected time is 0.1 secs

[II] Queue M/M/1 Law: $R = 1/(\mu - \lambda)$ [33]

response	<ul style="list-style-type: none"> • response time: $R = \frac{1}{\mu - \lambda}$ • Little's Law for system: $Q = \lambda R = \frac{\lambda}{\mu - \lambda}$
length	<ul style="list-style-type: none"> • expected queue length: L • $Q = L +$ probability server is busy (ρ) • $L = Q - \rho = \frac{\lambda^2}{\mu(\mu - \lambda)}$
waiting	<ul style="list-style-type: none"> • Little's Law for queue: $L = \lambda W$ • waiting time: $W = \frac{\lambda}{\mu(\mu - \lambda)}$

[II] M/M/1 Law: $R = 1/(\mu - \lambda)$ [34]

series	<ul style="list-style-type: none"> • response: $R_1 = 1/(\mu_1 - \lambda_1) = 1/(15 - 10) = 0.2$ • waiting: $W_1 = \frac{\lambda_1}{\mu_1(\mu_1 - \lambda_1)} = \frac{10}{15(15 - 10)} = 0.13$ secs • service: $S_1 = 1/\mu_1 = 1/15 = 0.07$ sec • response = waiting + service ($0.2 = 0.13 + 0.07$) • $R_2 = 1/(\mu_2 - \lambda_2) = 1/(20 - 10) = 0.1$
split/merge	<ul style="list-style-type: none"> • $R_1 = 1/(15 - 6.67) = 0.12$ • $R_2 = 1/(20 - 3.33) = 0.06$ • $R_3 = 1/(15 - 10) = 0.20$ • $W_1 = 6.67/15(15 - 6.67) = 0.05$ • $L_3 = 10 * 10 / 15(15 - 10) = 1.33$
cycle	<ul style="list-style-type: none"> • $R = 1/(150 - 10/0.1) = 0.02$ • $W = 100/150(150 - 100) = 0.013$

[II] General Response Time Law: $R = \sum R_i V_i$ [35]

Law	<ul style="list-style-type: none"> • system response time is mean time for jobs: R • $R =$ sum of server times weighted by visit ratio • Little's Law for system: $Q = \lambda R$ • Little's Law for server: $Q_i = \lambda_i R_i$ • number in system: $Q = Q_1 + Q_2 + \dots + Q_n$ • $Q = \lambda R = \lambda_1 R_1 + \lambda_2 R_2 + \dots + \lambda_n R_n$ • $R = R_1 \lambda_1 / \lambda + \lambda_2 / \lambda R_2 + \dots + \lambda_n / \lambda R_n$ • $R = \sum R_i V_i$
series	<ul style="list-style-type: none"> • $R = R_1 V_1 + R_2 V_2 = R_1 + R_2 = 0.2 + 0.1 = 0.3$ sec
split/merge	<ul style="list-style-type: none"> • $R = R_1 V_1 + R_2 V_2 + R_3 V_3$ • $R = (0.12)(2/3) + (0.06)(1/3) + (0.20)(1) = 0.30$
cycle	<ul style="list-style-type: none"> • $R = R_1 V_1 = (0.02)(10) = 0.2$

[II] Case Study: CPU vs. I/O [36]

$\lambda = 5, p = 0.1$
 $\mu_1 = 100, \mu_2 = 50$

1. Stream	<ul style="list-style-type: none"> • $\lambda_1 = \lambda + \lambda_2 = f(\lambda, p) = \lambda/p = 50$ jobs/sec • $\lambda_2 = (1 - p) \cdot \lambda_1 = (1 - p)\lambda/p = 45$ jobs/sec • $\lambda_3 = p \cdot \lambda_1 = \lambda = 5$ jobs/sec (job load balance) • $\lambda_4 = \lambda_2/3 = (1 - p)\lambda/3p = 15$ jobs/sec
2. Visits	<ul style="list-style-type: none"> • $V_{CPU} = \lambda_1/\lambda = 1/p = 10$ • $V_{I/O} = V_2 = V_3 = V_4 = \lambda_4/\lambda = (1 - p)/3p = 3$

[II] Case Study: CPU vs. I/O [37]

3. Utilization

- $\rho_{CPU} = \frac{\lambda_1}{\mu_1} = \frac{\lambda}{p\mu_1} = 0.5 = 50\%$
- $\rho_{I/O} = \rho_2 = \rho_3 = \rho_4 = \frac{\lambda_4}{\mu_2} = \frac{(1-p)\lambda}{3p\mu_2} = 0.3 = 30\%$

4. Bottleneck

- step 1: $\rho_{max} = 50\%, \mu_b = 100$ at $b = CPU$
- step 2: visit ratio $V_b = 1/p$
- $\lambda_{max} = \mu_b/V_b = p\mu_b = (0.1)(100) = 10$ jobs/sec
- usually I/O is bottleneck but 3 I/O's

[II] Case Study: CPU vs. I/O [38]

5. $M/M/1$

- $R_{CPU} = \frac{1}{\mu_1 - \lambda_1} = \frac{1}{\mu_1 - \lambda/p} = 0.020$ sec
- $R_{I/O} = \frac{1}{\mu_2 - \lambda_4} = \frac{1}{\mu_2 - (1-p)\lambda/3p} = 0.029$ sec

6. Little's

- $Q_{CPU} = \lambda_1 R_{CPU} = \frac{\lambda}{p\mu_1 - \lambda} = 1.0$ jobs
- $Q_{I/O} = \lambda_4 R_{I/O} = \frac{(1-p)\lambda}{3p\mu_2 - (1-p)\lambda} = 0.43$ jobs

7. General Response

- $R = \sum R_i V_i = R_{CPU} V_{CPU} + 3R_{I/O} V_{I/O}$
- $R = \frac{1}{p\mu_1 - \lambda} + \frac{3(1-p)}{3p\mu_2 - (1-p)\lambda} = 0.46$ sec

question

- what if there are n I/O processors? $3 \rightarrow n$

[III] Process Management Overview [39]

outline

- [A] Processes
- [B] CPU Scheduling
- [C] Process Synchronization

summary

- **process** is the execution of a **program**
- creation, execution, deletion of processes
- OS **schedules** which process gets the **CPU** next
- many processes appear to run **concurrently**
- orderly (**synchronized**) access to shared data
- Interprocess Communication (IPC)
 - semaphores and shared data
 - messages

[III.A] Process: Definitions [40]

program

- **program**: source code compiled to **executable**
- passive entity and resides on disk

process

- **process**: a program in execution
- active entity and the unit of work
- **data** for the variables used by the **instructions**
- also called a **job** or **task**
- long time to switch **heavyweight** processes

thread

- short time to switch **lightweight** processes
- subprocesses inside a process

CPU

- **Central Processing Unit** or **processor**
- executes processes, performing the instructions

[III.A] Process: Definitions [41]

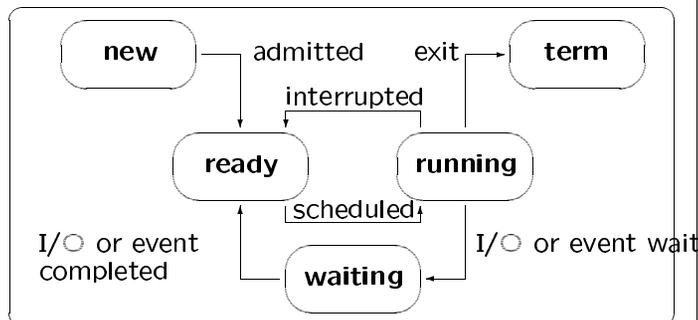
- running**
 - one process is **running**, or executing, on the CPU
- concurrent**
 - time-share CPU - appear to run simultaneously
- cooperating**
 - interact and affect each other
 - share data or resources
 - synchronous**
 - asynchronous**: not cooperating (no interaction)
- state**
 - current activity or condition (e.g. running)
- suspended**
 - waiting for memory - then joins the **ready queue**

[III.A] Process: Definitions [42]

- ready**
 - waiting for access to the CPU
 - not waiting for I/O or some other event
 - in the **ready queue** - list of process IDs (PIDs)
 - ready → running: **scheduled** for CPU
 - running → ready: **interrupted** by scheduler
 - running → **waiting**
- waiting**
 - not ready to use the CPU
 - waiting for I/O or memory
 - waiting for a **semaphore** to be signaled
 - waiting for a **message** to be received
- device queue**
 - waiting for **Input**
 - waiting for **Output**

[III.A] Process: State Changes [43]

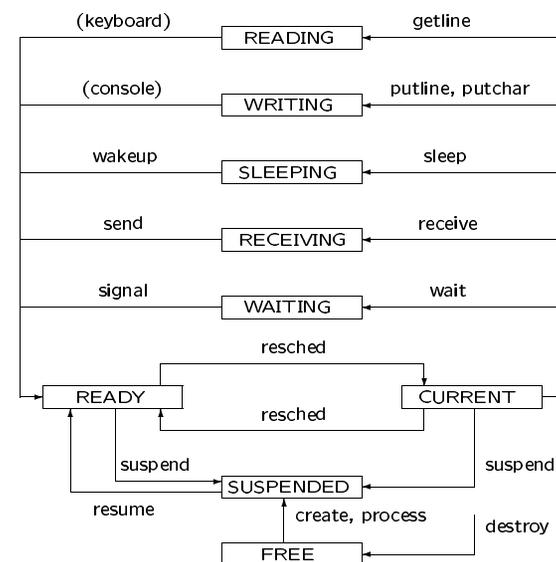
Figure 4.1



state changes

- new** processes **admitted** to **ready** queue
- ready process is **scheduled** for CPU
- current running process returns to ready or **terminates**
- or **waits** for I/O or other event

[III.A] Visual OS (VOS): A Process Manager [44]



[III.A] VOS: States

[45]

- **free:** not in the system - unused PID's
- **suspended:** waiting for admittance to ready queue
see long-term scheduler in Memory Management
- **ready:** waiting for the CPU
- **running:** executing on the CPU
- **waiting:** for a semaphore signal (synchronization)
- **receiving:** waiting for a message send
- **sleeping:** waiting for a wakeup timer
- **writing:** output to console
- **reading:** input from keyboard

[III.A] VOS System Calls

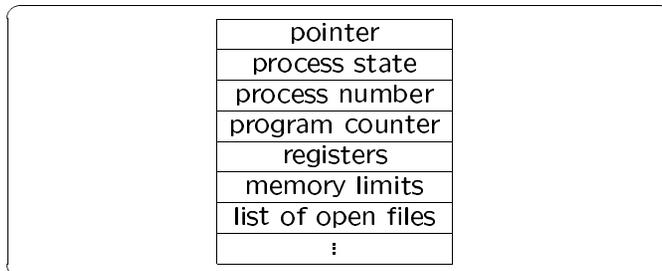
[46]

process	(pid,prio)	:	free	→	suspended
resume	(pid)	:	suspended	→	ready
suspend	(pid)	:	ready	→	suspended
			running	→	suspended
set_prio	(pid,prio)	:	ready	→	running
			running	→	ready
resched	()	:	ready	→	running
			running	→	ready
wait	(sem)	:	running	→	waiting
signal	(sem)	:	waiting	→	ready
receive	(&pid)	:	running	→	receiving
send	(pid,msg)	:	receiving	→	ready
sleep	(delay)	:	running	→	sleeping
wakeup	(pid)	:	sleeping	→	ready
putline	(str)	:	running	→	writing
(console)	()	:	writing	→	ready
getline	(str)	:	running	→	reading
(keyboard)	()	:	reading	→	ready
destroy	(pid)	:	<any>	→	free

[III.A] Process Control Block (PCB)

[47]

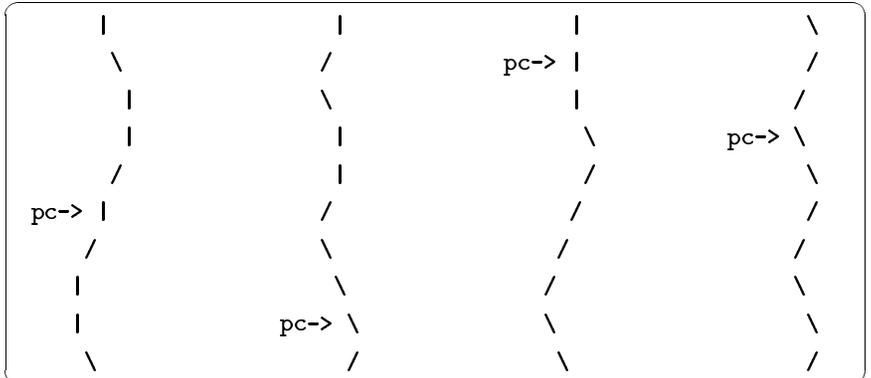
Figure 4.2



- each process has its own PCB
- program counter (**pc**) points to the next instruction to be executed
- information is sufficient to stop/restart process (**context switch**)
 - store all registers, etc. in PCB_1
 - put PID_1 back in ready queue (or other queue)
 - remove PID_2 from ready queue, load registers based on PCB_2
 - start executing PID_2 at **pc** register

[III.A] Program Counter (pc)

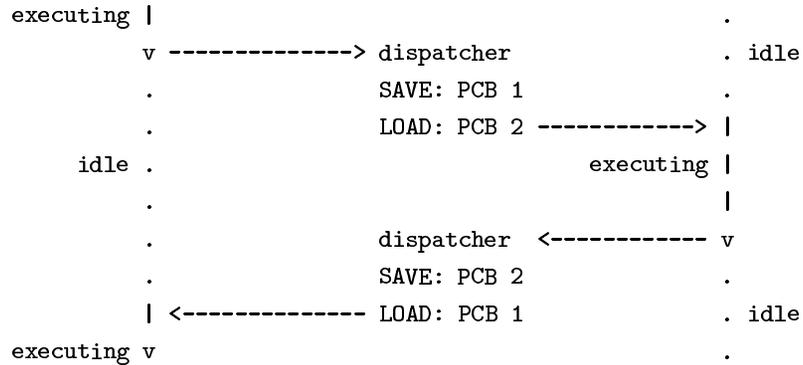
[48]



- each **process** (or **thread**) has a **pc**
- increments **sequentially** but branches for loops and conditionals

[III.A] Context Switch by Dispatcher

[49]



- CPU is switched to another process
- state (**PCB**) of the old process is saved
- saved state of the new process is loaded

[III.A] Interrupts

[50]

- *event that alters the sequence of instruction execution*
 - OS is interrupt driven:
 - sits quietly (no polling) until told there is something to do
 - interrupt is generated by hardware or software
- Steps:
- interrupts (usually) are **disabled** to prevent new ones
 - OS gains control of CPU
 - OS **saves** state of interrupted process (if user process: PCB)
 - OS analyzes interrupt, passes control to *interrupt handler* routine
 - predefined number of routines
 - index into table (*interrupt vector*) that points to routines
 - routine processes interrupt
 - **restore** state of interrupted process (or some "next" process)
 - interrupts (usually) are **enabled** to allow new ones
 - interrupted process (or "next") executes

[III.A] Interrupts

[51]

software interrupts:

- **program check** interrupt: division by zero, bad memory location
- **system call** to OS kernel (**trap**)
 - kernel is aware of process crossing its border
 - example: `read()`
 - kernel's device driver processes request
 - loads registers in device controller and starts controller
 - controller transfers data to buffer
 - when controller is finished: generates a hardware I/O interrupt

hardware interrupts:

- **I/O** interrupt
 - I/O completed, CPU can restart user process or "next" process
- **external** interrupt: expiration of quantum on clock
 - allows OS **dispatcher** to **context switch** to next process

[III.B] Process Scheduling

[52]

summary

- only one process at a time is **running** on the CPU
- process gives up CPU:
 - if it starts waiting for an **event**
- otherwise: other processes need **fair access**
- OS **schedules** which **ready** process to run next
- **time slice** or **quantum** for each process
- scheduling algorithms
 - different goals
 - affect performance

[III.B] Scheduling: Definitions

[53]

long-term scheduler

- **job scheduler**
- which process on disk should be given memory?
- result: new process in **ready queue**
- important in batch systems
- many processes in memory ⇒
high **degree of multiprogramming**

short-term scheduler

- **CPU scheduler**
- which process in ready queue should be given CPU
- result: new process on **CPU**

[III.B] Scheduling: Definitions

[54]

CPU-bound • most of its time doing computation - little I/O

I/O-bound • most of its time doing I/O - little computation

multilevel scheduling

- classified into different groups
- **foreground** (interactive) vs.
- **background** (batch)
- each group has its own ready queue

[III.B] Performance: Definitions

[55]

utilization

- **percentage** of time that the CPU is busy.
- if not busy, ready queue must be empty
 - CPU actually executes NULL process
- goal: keep the CPU busy

throughput

- if busy, then work is being done
- number of processes completed per second

turnaround

- total time to complete a process
- includes waiting in the **ready queue**
- executing on the **CPU**
- waiting for **I/O**
- goal: fast turnaround

[III.B] Performance: Definitions

[56]

response

- time waiting in the **ready queue** and
- executing on CPU until some output produced
- average is across all output events
- goal: fast response time

waiting

- sum of periods spent waiting in **ready queue**
- average is across all visits to ready queue
- goal: short waiting time

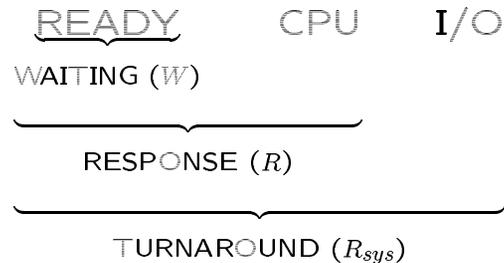
important

- **scheduler has a direct effect on waiting time**
- decides which process in queue gets to run next
- remaining processes must then wait longer
- OS cannot control code, amount of I/O, etc.

[III.B] Performance: Summary

[57]

- UTILIZATION: CPU %busy
- THROUGHPUT: jobs/sec
- WAITING: sec/job
- RESPONSE: sec/job (usually in time-share systems)
- TURNAROUND: sec/job (usually in batch systems)



[III.B] CPU Burst

[58]

CPU burst

- cycle of CPU burst, I/O wait, CPU burst, ...
- **program and data determine length of burst**
- scheduler may interrupt a burst
- but does not affect the full length

```
scanf n, a, b          /* I/O wait */
for (i=1; i<=n; i++)  /* CPU burst */
    x = x + a*b;
printf x              /* I/O wait */
for (i=1; i<=n; i++)  /* CPU burst */
    for (j=1; j<=n; j++)
        x = x + a*b;
printf x              /* I/O wait */
```

[III.B] Scheduling: FCFS

[59]

- First-Come, First-Served is simplest scheduling algorithm
- ready queue is a **FIFO** queue: First-In, First-Out
- longest waiting process at the front (**head**) of queue
- new ready processes join the rear (**tail**)
- **nonpreemptive**: executes until voluntarily gives up CPU
 - finished or waits for some event
- problem:
 - **CPU-bound** process may require a long **CPU burst**
 - other processes, with very short CPU bursts, wait in queue
 - reduces CPU and I/O device **utilization**
 - it would be better if the shorter processes went first

[III.B] Scheduling: FCFS

[60]

- assume processes arrive in this order: P_1, P_2, P_3
- **nonpreemptive** scheduling
- average waiting time: $(0+24+27)/3=17$ ms

PID	Burst	Gantt chart
P_1	24	
P_2	3	
P_3	3	

- assume processes arrive in this order: P_2, P_3, P_1
- average waiting time: $(6+0+3)/3=3$ ms

PID	Burst	Gantt chart
P_2	3	
P_3	3	
P_1	24	

- in general, **FCFS** average waiting time is not minimal
- in general, better to process shortest jobs first

[III.B] Scheduling: Round Robin (RR) [61]

- similar to **FCFS**, but **preemption** to switch between processes
- **time quantum (time slice)** is a small unit of time (10 to 100 ms)
- process is executed on the CPU for at most one time quantum
- implemented by using the **ready queue** as a **circular queue**
- **head** process gets the CPU
- uses less than a time quantum \Rightarrow gives up the CPU voluntarily
- uses full time quantum \Rightarrow **timer** will cause an **interrupt**
 - **context switch** will be executed
 - process will be put at the **tail** of queue

[III.B] Scheduling: RR [62]

- assume processes arrive in this order: P_1, P_2, P_3
- **preemptive** scheduling
- **time quantum**: 4 ms
- P_1 uses a full time quantum; P_2, P_3 use only a part of a quantum
- P_1 waits $0+6=6$; P_2 waits 4; P_3 waits 7
- average waiting time: $(6+4+7)/3=5.66$ ms

PID	Burst	Gantt chart																	
P_1	24	<table border="1"><tr><td>P_1</td><td>P_2</td><td>P_3</td><td>P_1</td><td>P_1</td><td>P_1</td><td>P_1</td><td>P_1</td></tr><tr><td>0</td><td>4</td><td>7</td><td>10</td><td>14</td><td>18</td><td>22</td><td>26</td><td>30</td></tr></table>	P_1	P_2	P_3	P_1	P_1	P_1	P_1	P_1	0	4	7	10	14	18	22	26	30
P_1	P_2		P_3	P_1	P_1	P_1	P_1	P_1											
0	4		7	10	14	18	22	26	30										
P_2	3																		
P_3	3																		

- very large time quantum \Rightarrow **RR = FCFS**
- very small time quantum \Rightarrow context switch is too much overhead
- quantum \approx CPU burst \Rightarrow better turnaround
 - rule of thumb: 80% should finish burst in 1 quantum

[III.B] Scheduling: Shortest-Job-First (SJF) [63]

- **assume** the next **burst time** of each process is known
- **SJF** selects process which has the shortest burst time
- **optimal** algorithm because it has the shortest average waiting time
- impossible to know **in advance**
- OS knows the **past** burst times - make a prediction using an average
- **nonpreemptive**
- or **preemptive**:
 - **shortest-remaining-time-first**
 - interrupts running process if a new process enters the queue
 - new process must have shorter burst than remaining time

[III.B] Scheduling: SJF [64]

- assume all processes arrive at the same time: P_1, P_2, P_3, P_4
- **nonpreemptive** scheduling
- average waiting time: $(3+16+9+0)/4=7$ ms

PID	Burst	Gantt chart									
P_1	6	<table border="1"><tr><td>P_4</td><td>P_1</td><td>P_3</td><td>P_2</td></tr><tr><td>0</td><td>3</td><td>9</td><td>16</td><td>24</td></tr></table>	P_4	P_1	P_3	P_2	0	3	9	16	24
P_4	P_1		P_3	P_2							
0	3		9	16	24						
P_2	8										
P_3	7										
P_4	3										

- **SJF is optimal**: shortest average waiting time
- but burst times are not known **in advance**
- next_predicted burst time by (weighted) average of **past** burst times
 - $\tau_{n+1} = \alpha \cdot t_n + (1 - \alpha) \cdot \tau_n$
 - next_predict = $\alpha \cdot$ last_observed + $(1 - \alpha) \cdot$ last_predict
 - $\alpha = 0 \Rightarrow$ next_predict = initialized value (usually 0)
 - $\alpha = 1 \Rightarrow$ next_predict = last_observed

[III.B] SJF: Weighted Average Burst [65]

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \dots + (1 - \alpha)^{n+1} \tau_0$$

$$\alpha = 1 : \tau_{n+1} = t_n$$

$$\alpha = 0 : \tau_{n+1} = \tau_0$$

$\alpha = 1/2$: recent and past history the same

time	0	1	2	3	4	5	6	7
Burst (t_i)		6	4	6	4	13	13	13
Guess (τ_i)	10	8	6	6	5	9	11	12

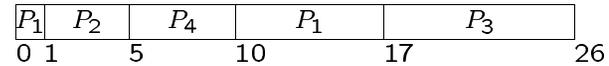
$$\tau_1 = \frac{1}{2}t_0 + \frac{1}{2}\tau_0 = \frac{1}{2}6 + \frac{1}{2}10 = 8$$

[III.B] Scheduling: SJF [66]

- assume processes arrive at 1 ms intervals: P_1, P_2, P_3, P_4
- **preemptive** scheduling: **shortest-remaining-time-first**
- P_1 waits $0+(10-1)=9$; P_2 waits $1-1=0$
- P_3 waits $17-2=15$; P_4 waits $5-3=2$
- average waiting time: $(9+0+15+2)/4=6.5$ ms

PID	Burst	Arrival
P_1	8	0
P_2	4	1
P_3	9	2
P_4	5	3

Gantt chart



- **nonpreemptive SJF**: 7.75 ms

[III.B] Scheduling: Priority (PRIO) [67]

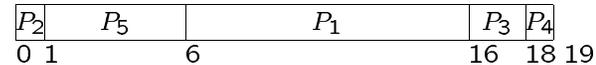
- assume a priority is associated with each process
- select highest priority process from the ready queue
- let τ be the (predicted) next CPU burst of a process
- **SJF** is a special case of priority scheduling
 - assume: high numbers \Rightarrow high priority
 - then priority is $1/\tau$
 - assume: low numbers \Rightarrow high priority
 - then priority is τ
- equal-priority processes are scheduled in **FCFS** order
- PRIO can be **preemptive** or **nonpreemptive**
- priorities can be defined **internally**
 - memory requirements, number of open files, burst times
- priorities can be defined **externally**
 - user, department, company

[III.B] Scheduling: PRIO [68]

- assume all processes arrive at the same time: P_1, P_2, P_3, P_4, P_5
- **nonpreemptive** scheduling
- **high** priority: **low** number
- **some OS use a high number!!! See VOS.**
- average waiting time is: $(6+0+16+18+1)/5=8.2$ ms

PID	Burst	Priority
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

Gantt chart



- **indefinite blocking (starvation)**: low priority process never runs
- **aging**: low priorities increase with waiting time, will eventually run

[III.B] VOS Scheduling: PRIO, FCFS, SJF [69]

```
for (i=1; i<=10; i++){ /* 10 CPU BURSTS */
  for (j=1;j<=HOWLONG;j++){ /* 1 CPU BURST */
    pm_busywait(); /* PID1:long PID2:medium PID 3:short*/
    pm_yield(); /* GO BACK TO READY QUEUE */
  }
}
```

PID	Burst	PRIO priority=fixed	FCFS priority=equal	SJF priority=1/burst
1	long	2	1	low
2	medium	3 high	1	medium
3	short	1 low	1	high

- schedulers **favor** different PIDs
- SUMMARY shows **CPU burst** (running) time for each PID
- SUMMARY shows waiting time for each PID in **ready queue**
- **Gantt chart** shows how long each PID is on the CPU
- schedulers have different performance

[III.B] VOS Scheduling: PRIO [70]

```
===== SUMMARY =====
```

PID	FREE	SUSPENDED	READY	RUNNING	WAITING	RECEIVING	SLEEPING	WRITING	READ						
time	cnt	time	cnt	time	cnt	time	cnt	time	cnt						
0	0	1	0	0	72	2	17	3	0	0	0	0	0	0	0
1	29	2	1	1	25	11	34	11	0	0	0	0	0	0	0
2	64	2	0	1	1	11	24	11	0	0	0	0	0	0	0
3	16	2	1	1	58	11	14	11	0	0	0	0	0	0	0
4	89	1	0	0	0	0	0	0	0	0	0	0	0	0	0
5	89	1	0	0	0	0	0	0	0	0	0	0	0	0	0
6	89	1	0	0	0	0	0	0	0	0	0	0	0	0	0
7	89	1	0	0	0	0	0	0	0	0	0	0	0	0	0
8	89	1	0	0	0	0	0	0	0	0	0	0	0	0	0
9	89	2	0	1	0	1	0	1	0	0	0	0	0	0	0
TOT	643	14	2	4	156	36	89	37	0	0	0	0	0	0	0

Utilization: 80.9 %Busy
 Throughput : 2.0 Jobs/Min
 Wait Time : 28.0 Sec/Job
 Burst Time : 24.0 Sec/Job

[III.B] VOS Scheduling: PRIO [71]

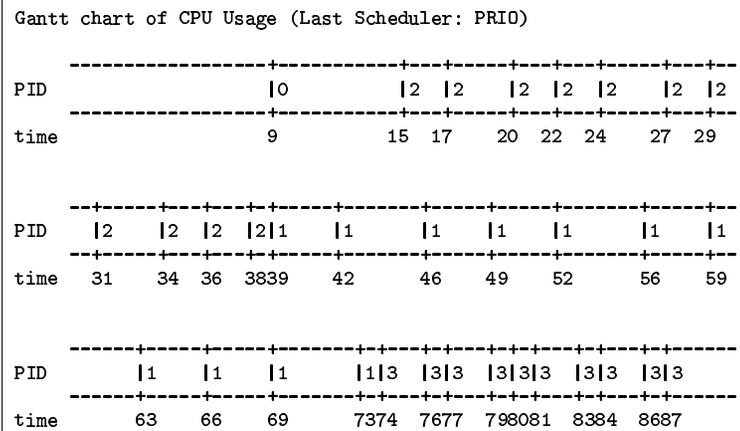
```
Scheduling Algorithm: PRIO
>>> SUMMARY (READY) <<< >>>>>>>>>> SUMMARY (RUNNING) <<<<<<<<<<<<
```

PID	TOT Wait Time	TOT Burst Time / Cnt = Single Burst
1	25	34 11 3.1
2	1	24 11 2.2
3	58	14 11 1.3

Sum Wait Time: 84 /3 Jobs Sum Burst Time: 72 /3 Jobs
 Avg Wait Time: 28 Sec/Job Avg Burst Time: 24 Sec/Job
 Longest Wait: 58(PID: 3) Longest Single Burst: 3.1(PID: 1)
 Shortest Wait: 1(PID: 2) Shortest Single Burst: 1.3(PID: 3)

- other algorithms will have different **average wait time**

[III.B] VOS Scheduling: PRIO [72]



- other algorithms will favor different PIDs

[III.B] Mechanism (how) vs. Policy (what) [73]

mecha- nism

- **how** to do something
- implementation or **function** with **parameters**
- used in many ways (by policies)
- OS may be **micro kernel** - only basic mechanisms
- policies are decided at the user level

policy

- **what** or **when** to do something
- set of **rules**
- use mechanisms by setting parameters
- important choices in the design of the OS
- **mechanisms should be separate from policies**

[III.B] Mechanism vs. Policy: Examples [74]

Timer(x sec)

- Policy 1: if LOW_PRIORITY Timer(0.1) else Timer(1.0)
- Policy 2: if LOW_PRIORITY Timer(0.1) else Timer(0.2)

Schedule(job)

- Policy 1: Schedule(I/O Job A); Schedule(CPU Job B)
- Policy 2: Schedule(CPU Job A); Schedule(I/O Job B)

Preempt(job)

- Policy 1: if A.running > 0.1 sec then Preempt(Job A)
- Policy 2: if A.running > 0.2 sec then Preempt(Job A)

Remove_From
Ready_Q(job)

- Policy 1: Remove_Ready_Q(oldest job): FCFS
- Policy 2: Remove_Ready_Q(highest priority job): PRI○
- Policy 3: Remove_Ready_Q(shortest job): SJF

[III.C] Process Synchronization [75]

- semaphores and classical problems
- interprocess communication (IPC)
- **cooperating** processes:
 - synchronized (orderly) access to shared globals
- **process control**:
 - mechanism to prevent execution until a certain **event** occurs
 - **send** of a **message**
 - **wakeup** of a **sleeping** process
 - **signal** of a **semaphore**
 - **wait** and **signal** guarantee only **one** process
at a time executes a **critical section** of code
 - protects the **shared access** to global variables

[III.C] Critical Section: Example Problem [76]

```
program code:      global variable x:
    x=0;            0
    x++;           1
    printf x;      1 <=== output
```

- **interleave** 3 processes executing the same program code:

```
PID 1      PID 2      PID 3      x:
x=0;       x=0;       x=0;       0
x++;       x++;       x++;       1
printf x;  printf x;  printf x;  0 <=== output
           x++;       x++;       1 <=== output
           x++;       x++;       2 <=== output
```

[III.C] Critical Section: Attempted Solution [77]

Shared Variable:

```
int v=0; /* v==0 => critical section OPEN; v==1 => critical section CLOSED */
```

Code: At $p_i \in \{1, \dots, n\}$:

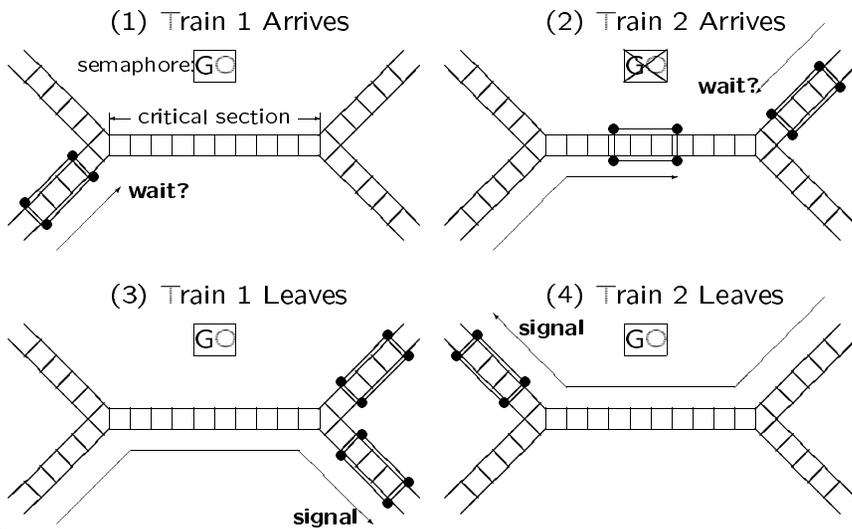
```
while (1) {  
    /* trying region */  
    while (v == 1) /* do nothing */ ;  
  
    v = 1;  
    /* critical section (region) */  
    x = 0;  
    x++;  
    printf x;  
  
    v = 0;  
    /* remainder region */  
}
```

- what is wrong? what if critical section is: dial_phone("555-1212")?
- how can it be fixed?
- is this **spinlock** good or bad?

[III.C] Critical Section: General Problem [78]

- n processes each with segment of code called **critical section**
- one process changes common (shared) variables, writes to a file.
- no other process is allowed to execute in its critical section
- execution of critical sections is **mutually exclusive** in time
- one solution: **semaphore**
 - lets **one** process into the critical section
 - puts other processes in a **semaphore queue** (no **busy waiting**)
 - process is finished: **head** of FIFO queue enters section

[III.C] Semaphore: wait and signal [79]



[III.C] Semaphore: wait and signal [80]

- critical section: general solution

```
while(1) {  
    pm_wait(sem);  
    /* critical section or region */  
    pm_signal(sem);  
}
```

- critical section: example solution

```
while(1) {  
    pm_wait(sem);  
    x=0;  
    x++;  
    printf x;  
    pm_signal(sem);  
}
```

[III.C] Semaphore: wait and signal [81]

- semaphore has **count** and FIFO **queue** of waiting processes

```
struct {
    int count=1;
    FIFO queue;
} semaphore;
```

- **count** $\geq 0 \Rightarrow$ queue is **empty**
- **count** of **negative** $n \Rightarrow$ queue has n **waiting** processes

```
wait(semaphore) : if (--semaphore.count<0){
    put_at_tail(pid, semaphore.queue);
    suspend(pid) }

signal(semaphore) : if (semaphore.count++<0){
    pid=get_at_head(semaphore.queue);
    ready(pid) }
```

[III.C] Semaphore: Initialization [82]

- usually first process to **wait** is allowed access to **critical section**
- next process to **wait** is placed on the **semaphore queue**
- until the first process is finished (**signal**)
- to guarantee this **first** access, initialize either:
- by setting **count=1**:

```
sem = pm_seminit(1);
```

- by setting **count=0** and then **signal**:

```
sem = pm_seminit(0);
pm_signal(sem);
```

[III.C] Semaphore: Count [83]

PID 1	PID 2	count
sem=pm_seminit(1);		1
pm_wait(sem);		0
(critical)	pm_wait(sem);	-1
v		
pm_signal(sem);		0
pm_wait(sem);	(critical)	-1
	v	
	pm_signal(sem);	0
(critical)	pm_wait(sem);	-1
v		
pm_signal(sem);		0

[III.C] Critical Section Solution [84]

- **mutual exclusion**:
 - guarantee only **one** process is executing its critical section
 - also called "safety"
- **progress**:
 - guarantee that all processes make **steps** through program code
 - also called "liveness"
 - FIFO queue \Rightarrow progress
 - LIFO queue \Rightarrow some processes may face **starvation**
- **fairness**:
 - guarantee that all processes will be able to access the resource
 - **eventuality**: by some time
 - **time-bounded**: by a specified time
 - **turn-bounded**: by some specified number of tries

[III.C] Critical Section: Deadlock [85]

- **deadlock:**
 - all processes are waiting **indefinitely** for some event to occur
 - that can only be caused by one of the waiting processes
 - “mutual waiting”
 - none of the processes make **progress**

Kansas legislature:

*When two trains approach each other at a crossing,
both shall come to a full stop
and neither shall start up again until the other has gone.*

[III.C] Semaphores: Deadlock [86]

- process 1 **waits** for semaphore S and Q:

```
pm_wait(S);  
pm_wait(Q);  
/* critical section */  
pm_signal(S);  
pm_signal(Q);
```

- process 2 **waits** for semaphore Q and S:

```
pm_wait(Q);  
pm_wait(S);  
/* critical section */  
pm_signal(Q);  
pm_signal(S);
```

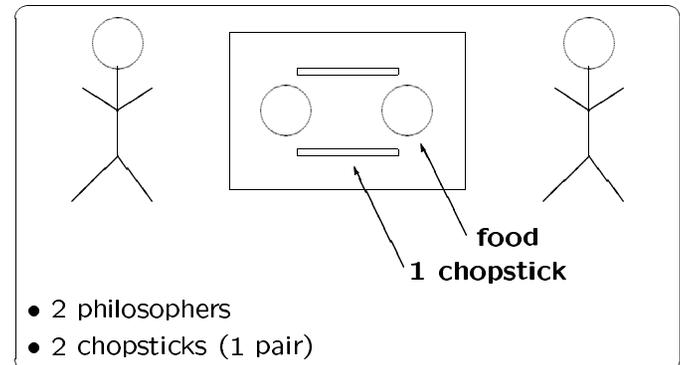
- both processes are **deadlocked** as each are **waiting** for the other

[III.C] Classical Problems: Introduction [87]

- **classical** problems are simple but represent complex, real problems
- show important features of real problem/solution
- examples from OS and real-time applications
- main points:
 - **mutual exclusion** to **critical section** code
 - **synchronized** access to **shared** resources and data

[III.C] Dining Philosophers [88]

problem



questions

- How can both philosophers eat?
- How to share resources?

answer

- TAKE TURNS USING RESOURCES

[III.C] Dining Philosophers

[89]

- Rule 1: a **philosopher** thinks.
- Rule 2: a philosopher gets 2 chopsticks (1 pair).
- Rule 3: a philosopher eats.
- Rule 4: there are only 2 chopsticks (1 pair).
- Rule 5: after eating, philosopher puts down 2 chopsticks (1 pair).

demo-OK

Exercise 7a: Philosopher 1 uses the 2 chopsticks (1 pair) to eat. Then Philosopher 2 uses the 2 chopsticks (1 pair) to eat. Both make **progress** and the access to the **resources** (chopsticks) is **synchronized**.

demo-BAD

Exercise 7a: Philosopher 1 gets one (1) chopstick. Philosopher 2 gets the other (1) chopstick. Neither can eat. Both are **deadlocked**.

[III.C] Dining Philosophers

[90]

- each chopstick is a **semaphore**
- a philosopher **waits** for access to 2 chopsticks
- a philosopher **signals** when finished eating

```
while(1) {
    /* philosopher thinks */
    pm_wait(chopstick1);
    pm_wait(chopstick2);
    /* philosopher eats */
    pm_signal(chopstick1);
    pm_signal(chopstick2);
}
```

[III.C] Burns' Dining Philosophers

[91]

Shared Variables:

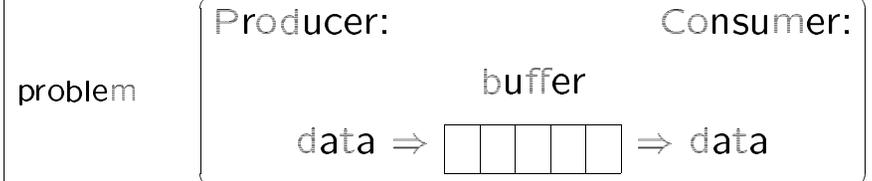
1. *FORK*: semaphore array $[0..n - 1]$, initially all available

Code: At $p_i \in \{0, \dots, n - 1\}$:

```
do forever
  if even(i) then {
    wait(FORKi+1) /* left fork */
    wait(FORKi) /* right fork */
  }
  if odd(i) then {
    wait(FORKi) /* right fork */
    wait(FORKi+1) /* left fork */
  }
  /* Critical region */
  signal(FORKi)
  signal(FORKi+1)
```

[III.C] Producer/Consumer

[92]



- questions
- What if producer is faster than consumer?
 - What if buffer overflows?
 - What if consumer is faster than producer?
 - What if buffer is empty?

- answer
- PRODUCER WAITS FOR NOT FULL
 - CONSUMER WAITS FOR NOT EMPTY

[III.C] Producer/Consumer

[93]

Process 1 is a **producer** of data. Process 2 is a **consumer** of data.

Rule 1: the producer puts data in the **buffer**.

Rule 2: the buffer holds only 5 numbers.

Rule 3: the consumer gets data from the buffer.

Rule 4: the producer waits for the buffer to be **not full**.

Rule 5: the consumer waits for the buffer to be **not empty**.

demo-OK

Exercise 8a: Producer puts data into buffer; consumer gets data and writes it.

demo-BAD

Exercise 8b: The producer fills the buffer with 5 numbers, then **waits** for the consumer to **signal** that the buffer is **not_full**. The consumer reads from the buffer, then **waits** for the producer to **signal** that the buffer is **not_empty**. Both are **dead-locked**.

[III.C] Producer/Consumer

[94]

- producer **waits** for the buffer to be **not full**
- producer **waits** for access to the buffer (**mutual exclusion**)
- producer **signals** after access to the buffer (**mutual exclusion**)
- producer **signals** that the buffer is **not empty**

```
while(1) {
    pm_wait(not_full);
    pm_wait(sem_buffer);
    /* put data into buffer */
    pm_signal(sem_buffer);
    pm_signal(not_empty);
}
```

[III.C] Readers/Writer

[95]

problem

many readers OK: FILE ⇒ data

one writer OK: FILE ← data

read/write NOT OK: data ⇒ FILE ⇒ data

questions

- What if readers are reading data from file?
- What if writer is writing data to file?

answer

- READERS WAIT FOR NO WRITERS
- WRITER WAITS FOR SOLE ACCESS

[III.C] Readers/Writer

[96]

Process 1 and 2 **read** data from a file.

Process 3 **writes** data to the file.

Rule 1: if a process reads data, another process **can** read data.

Rule 2: if a process writes data, another process **cannot** read data.

demo-OK

Exercise 9a: Readers 1 and 2 read from the file and Writer 3 **waits**. Then, Writer 3 writes to the file and Readers 1 and 2 **wait**. All make **progress** and the shared access to the file is **synchronized**. The readers see a **consistent** file.

demo-BAD

Exercise 9b: Writer 3 writes to the file **before** Readers 1 and 2 are finished reading from the file. The readers do not see a **consistent** file. Writer 3 needs to use a semaphore.

[III.C] Readers/Writer

[97]

- reader **waits** for writer to finish writing
- reader **signals** to writer when all readers are finished reading

```
while(1) {
    pm_wait(sem_readcount);
    readcount++;
    if (readcount==1) pm_wait(sem_wrt);
    pm_signal(sem_readcount);
    ...
    pm_wait(sem_readcount);
    readcount--;
    if (readcount==0) pm_signal(sem_wrt);
    pm_signal(sem_readcount);
}
```

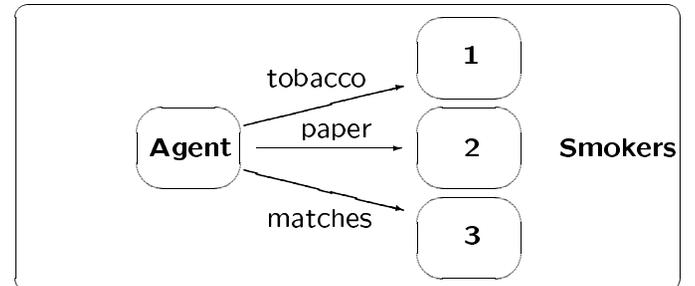
- writer **waits** for access to the file
- writer **signals** after access to the file

```
while(1) {
    pm_wait(sem_wrt);
    /* write to file */
    pm_signal(sem_wrt);
}
```

[III.C] Cigarette Smokers

[98]

problem



questions

- What if three products are needed to smoke?
- What if each smoker needs a different product?
- What if agent provides product only in turn?
- How to allocate resources?

answer

- AGENT: SIGNAL and WAIT FOR PRODUCT
- SMOKER: WAIT and SIGNAL FOR PRODUCT

[III.C] Cigarette Smokers

[99]

Process 1, 2, and 3 are smokers. Process 4 is an agent.
Rule 1: smokers need 3 products: tobacco, paper, matches.
Rule 2: Smoker 1 needs tobacco, 2 needs paper, 3 needs matches.
Rule 3: the agent puts tobacco on the table and Smoker 1 smokes.
Rule 4: the agent puts paper on the table and Smoker 2 smokes.
Rule 5: the agent puts matches on the table and Smoker 3 smokes.

demo-OK

Exercise 10a: Each smoker gets to smoke in turn. All make **progress** and the access to **resources** is **synchronized**.

demo-BAD

Exercise 10b: Smoker 1 gets to smoke while the other smokers **wait** for a **signal** that Smoker 1 is done. But Smoker 1 is **waiting** for its turn again. All are **deadlocked**.

[III.C] Cigarette Smokers

[100]

- Smoker 1 **waits** for tobacco to be supplied by the agent
- Smoker 1 **signals** to the agent when finished

```
while(1) {
    pm_wait(tobacco);
    /* smoke */
    pm_signal(tobacco);
}
```

- agent **signals** that tobacco is available
- agent **waits** for the smoker to finish

```
while(1) {
    pm_signal(tobacco);
    pm_wait(tobacco);
    ...
}
```

[III.C] IPC Message: send and receive [101]

- process **receives** messages from **pid**

```
while(1) {  
    msg = pm_receive(&pid);  
}
```

- process **sends** messages to **pid=7**

```
pid=7;  
while(1) {  
    pm_send(pid,msg++);  
}
```

- process **sends** and **receives** messages from **pid**

```
while(1) {  
    msg = pm_receive(&pid);  
    pm_send(pid,msg++);  
}
```

[III.C] Message Demo [102]

- Process 1 **waits** on a **semaphore**
- Process 1 **sends** a message to Process 2

```
pid = 2;  
while(1) {  
    pm_wait(sem);  
    pm_send(pid,msg);  
}
```

- Process 2 **signals** the semaphore
- Process 2 **receives** a message from Process 1

demo-OK Exercise 11a: Both processes make **progress**.

demo-BAD Exercise 11b: Both processes are **deadlocked**.

[III.C] Client/Server [103]

Process 1 is a **server**. Process 2 and 3 are **clients**.

Rule 1: a client **sends** a **message** for action by the server.

Rule 2: the server **receives** the message from a client.

Rule 3: the server sends an answer back to the client.

Rule 4: the client receives the message from the server.

demo-OK

Ex12a: Server waits in **receiving** for a message from a client. Each client sends a message and waits in receiving for an answer. The server receives each message, in turn, and sends back an answer.

demo-BAD

Ex12b: Server receives both messages but does not send back answers to either client. The clients are waiting for answers and the server is waiting for new messages. All processes are **deadlocked**.

[III.C] Client/Server [104]

- a client **sends** a **message** for action by the server
- the client receives a message from the server

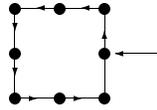
```
while(1) {  
    pm_send(server_pid,msg);  
    msg=pm_receive(&server_pid);  
}
```

- the server **receives** and **sends** messages

[III.C] Ring Network

[105]

ring



8 processes wait for messages on a **ring network**.

Rule 1: receive a message.

Rule 2: message++.

Rule 3: if the message has gone around the ring, "finished".

Rule 4: else, **send** the message to the next process on the ring.

[III.C] Ring Network

[106]

demo-OK

Exercise 13a: All processes are **deadlocked**, waiting for a message. **Send, Process 1**, message "0". The message goes to all processes on the ring.

demo-BAD

Exercise 13b: The message does not go to any other processes on the ring. All processes are **deadlocked** waiting for messages.

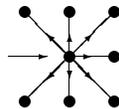
- process **receives** message and **sends** to next process on ring

```
while(1) {  
    msg=pm_receive(&pid);  
    ...  
    pm_send(nextpid,msg);  
}
```

[III.C] Star Network

[107]

star



8 processes wait for messages on a **star network**.

Process 1 is the **center**. The others are on the **rim**.

Rule 1: **receive** a message.

Rule 2: message++.

Rule 3: **rim** only: if message not from center, **send** message to center.

Rule 4: **rim** only: if message has gone to all processes, "finished".

Rule 5: **center** only: **send** message++ to the remaining processes.

[III.C] Star Network

[108]

demo-OK

Exercise 14a: All processes are **deadlocked**, waiting for a message. **Send, Process 1**, message "0". The message goes to all processes on the **rim** of the star.

demo-BAD

Exercise 14b: The message does not go to any processes on the **rim** of the star. All processes are **deadlocked** waiting for messages.

[III.C] Star Network

[109]

- **rim** process **receives** a message and **sends** to the **center** process

```
while(1) {  
    msg=pm_receive(&pid);  
    if (pid != center_pid)  
        pm_send(center_pid,msg++);  
}
```

- **center** process **receives** a message and **sends** to **rim** processes

```
while(1) {  
    msg=pm_receive(&pid);  
    msg++;  
    for (all rim processes which do not have the message)  
        pm_send(rim_pid,msg++);  
}
```

[IV] Deadlock: Introduction

[110]

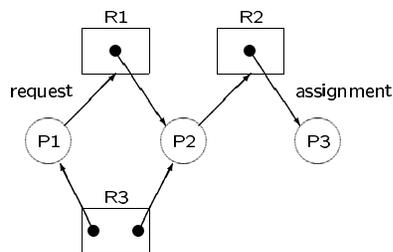
- finite number of **resources** distributed to competing processes
 - physical: memory space, CPU cycles, I/O devices
 - logical: files, semaphores
- multiple **instances** of a resource type
- process **requests**, **uses**, **releases** resource
- **deadlock**:
set of procs waiting for release from one (or more) members of set

[IV] Deadlock: Necessary Conditions

[110]

- mutual exclusion: at least one **nonshareable** resource
- hold and wait: at least one proc holds resource, waits for other
- no preemption: resources can only be voluntarily released
- circular wait: P_0 waiting for P_1 ... waiting for P_0
 - circular wait \Rightarrow hold and wait
- if you can break any one of these conditions \Rightarrow **no deadlock**

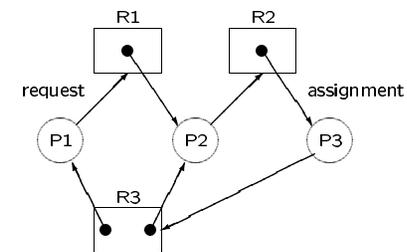
[IV] Deadlock: Resource-Allocation Graph [111]



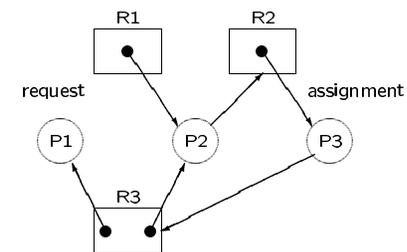
- no cycle \Rightarrow no deadlock
- deadlock \Rightarrow cycle (necessary condition)
- cycle \Rightarrow maybe deadlock (but not sufficient condition)
- single instance resource \wedge cycle \Rightarrow deadlock
- (necessary and sufficient)

[IV] Deadlock: Multiple Instance Resources [112]

DEADLOCK:



NO DEADLOCK:



[IV] Methods for Handling Deadlock [113]

- never let deadlock occur
 - **prevention**: break one of the 4 conditions
 - **avoidance**: resources give advance notice of maximum use
- let deadlock occur and do something about it
 - **detection**: search for cycles periodically
 - **recovery**: preempt processes or resources
- don't worry about it (UNIX and other OS)
 - cheap: just reboot (it happens rarely)

[IV] Deadlock: Prevention [114]

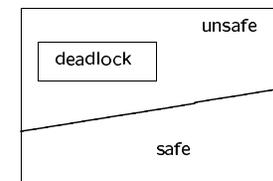
- break **mutual exclusion**:
 - read-only files are shareable
 - but some resources are intrinsically nonshareable (printers)
- break **hold and wait**:
 - request all resources in advance
request (tape, disk, printer)
 - release all resources before requesting new batch
request (tape,disk), release (tape,disk), request (disk,printer)
 - disadvantages: low resource utilization, **starvation**

[IV] Deadlock: Prevention [115]

- break **no preemption**:
 - process 1 requests resources already allocated to process 2:
 - process 1 forfeits its current resources
 - if process 2 is waiting for other resources: process 2 forfeits
 - used for resources whose state is easily saved/restored
 - CPU registers and memory space
 - but not printers or tape drives
- break **circular wait**:
 - order all resources by unique numbers (tape drives, etc.)
 - processes request resources in increasing order

[IV] Deadlock: Avoidance [116]

- processes give **advance** notice about **maximum** usage of resources
- processes make actual **requests** when they need a resource
- avoidance algorithm: **allocate** request only if it yields a **safe state**
a sequence of processes exists such that each process can still get their maximum in sequence
- conceptually the processes could be run in this order



[IV] Deadlock: Example of Avoidance [117]

- assume that system has 12 tape drives

	maximum needs	current needs
P_0	10	5
P_1	4	2
P_2	9	2
TOTAL	23	9

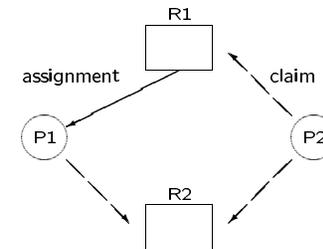
- sequence $\langle P_1, P_0, P_2 \rangle$ is a **safe sequence**
- P_2 requests one more tape drive:

	maximum needs	current needs
P_0	10	5
P_1	4	2
P_2	9	3
TOTAL	23	10

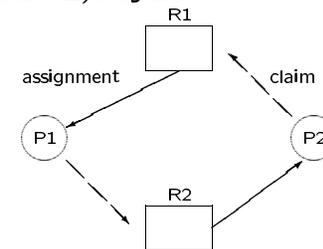
- no safe sequence $\langle P_1, \dots \rangle$ because P_2 and P_0 could deadlock
- for avoidance with multiple instances: use **Banker's Algorithm**

[IV] Avoidance with Single Instances [118]

SAFE:



UNSAFE (P_2 requests R_2): **cycle**



[IV] Deadlock: Detection [119]

- single instance resource types:
 - periodically make a **wait for graph** (just remove resources from the **allocation graph**)
 - check for cycles (n^2)
- multiple instance resource types:
 - use a time-varying **banker's algorithm**
- how often should detection algorithm be run?
 - how often is deadlock likely?
 - how many processes will be affected?

[IV] Deadlock: Recovery [120]

- **process termination**
 - abort all deadlocked processes
 - abort one process at a time until cycle eliminated
 - run detection algorithm each time
 - which process to abort next?
 - priority? CPU usage? how many resources?
- **resource preemption**
 - take resources from some processes and give them to others
 - **select victim**: cheapest cost?
 - **rollback**: to safe state and restart
 - **starvation**: deadlock could occur again, process restarts
 - include number of rollbacks in cost

[IV] Deadlock: Combined Approach [121]

- resources belong to **classes**
- order the classes hierarchically (each gets unique number)
- use appropriate technique within class

Example of 4 classes:

- **internal resources**: for PCBs, etc.
 order the resources within the class
- **central memory**: for a user job
 preemption because a job can always be swapped out
- **job resources**: for tape drives, etc.
 avoidance because of job-control cards
- **swappable space**: for user jobs on backing store
 preallocation because maximum requirements known

[IV] Deadlock: Banker's Algorithm [122]

- multiple instances of resource types \Rightarrow
 cannot use resource-allocation graph
- banks do not allocate cash unless they can satisfy customer needs
- when a new process enters the system
 declare in advance maximum need for each resource type
 cannot exceed the total resources of that type
- later, processes make actual request for some resources
- if the the allocation leaves system in safe state
 grant the resources
- otherwise
 suspend process until other processes release enough resources

[IV] Banker: Data Structures [123]

```
#define MAXN 10          /* maximum number of processes      */
#define MAXM 10          /* maximum number of resource types   */
int Available[MAXM];    /* Available[j] = current # of unused resource j */
int Max[MAXN][MAXM];    /* Max[i][j] = max demand of i for resource j */
int Allocation[MAXN][MAXM]; /* Allocation[i][j] = i's current allocation of j */
int Need[MAXN][MAXM];    /* Need[i][j] = i's potential for more j */
                        /* Need[i][j] = Max[i][j] - Allocation[i][j] */
```

Notation:

$X \leq Y$ iff $X[i] \leq Y[i]$ for all i

(0,3,2,1) is less than (1,7,3,2)

(1,7,3,2) is NOT less than (0,8,2,1)

Each row of *Allocation* and *Need* are vectors: $Allocation_i$ and $Need_i$

[IV] Banker: Example [124]

Initially:

Available

A B C

10 5 7

Later Snapshot:

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

[IV] Banker: Safety Algorithm [125]

- consider some sequence of processes
- if the first process has *Need* less than *Available*
it can run until done
then release all of its allocated resources
allocation is increased for next process
- if the second process has *Need* less than *Available*
- ...
- then all of the processes will be able to run eventually
- \Rightarrow system is in a **safe state**

[IV] Banker: Safety Algorithm [126]

```

STEP 1: initialize
    Work := Available;
    for i = 1,2,...,n
        Finish[i] = false
STEP 2: find i such that both
    a. Finish[i] is false
    b. Need_i <= Work
    if no such i, goto STEP 4
STEP 3:
    Work := Work + Allocation_i
    Finish[i] = true
    goto STEP 2
STEP 4:
    if Finish[i] = true for all i, system is in safe state
    
```

[IV] Banker: Safety Example [127]

Using the previous example, $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ satisfies criteria.

	Max			- Allocation			=	Need <= Work			Available					
	A	B	C	A	B	C		A	B	C	A	B	C			
P1	3	2	2	2	0	0		1	2	2	3	3	2	3	3	2
P3	2	2	2	2	1	1		0	1	1	5	3	2			
P4	4	3	3	0	0	2		4	3	1	7	4	3			
P2	9	0	2	3	0	2		6	0	0	7	4	5			
P0	7	5	3	0	1	0		7	4	3	10	4	7			
											10	5	7	<<< initial system		

[IV] Banker: Resource-Request Algorithm [128]

```

STEP 0: P_i makes Request_i for resources, say (1,0,2)
STEP 1: if Request_i <= Need_i
    goto STEP 2
    else ERROR
STEP 2: if Request_i <= Available
    goto STEP 3
    else suspend P_i
STEP 3: pretend to allocate requested resources
    Available := Available - Request_i
    Allocation_i := Allocation_i + Request_i;
    Need_i := Need_i - Request_i
STEP 4: if pretend state is SAFE
    then do a real allocation and P_i proceeds
    else
        restore the original state and suspend P_i
    
```

[IV] Banker: Resource-Request Algorithm [129]

Say P_1 requests (1,0,2)

Compare to $Need_1$: $(1,0,2) \leq (1,2,2)$

Compare to $Available$: $(1,0,2) \leq (3,3,2)$

Pretend to allocate resources:

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

Is this safe? Yes: $\langle P_1, P_3, P_4, P_0, P_2 \rangle$

Can P_4 get (3,3,0)? No, $(3,3,0) > (2,3,0)$ *Available*

Can P_0 get (0,2,0)? $(0,2,0) < (2,3,0)$ *Available*

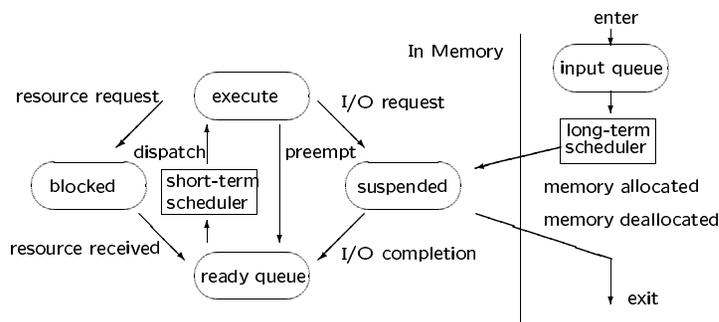
Pretend: *Available* goes to (2,1,0)

But ALL *Needs* are greater than *Available* \Rightarrow NOT SAFE

[V.A] Memory Management [130]

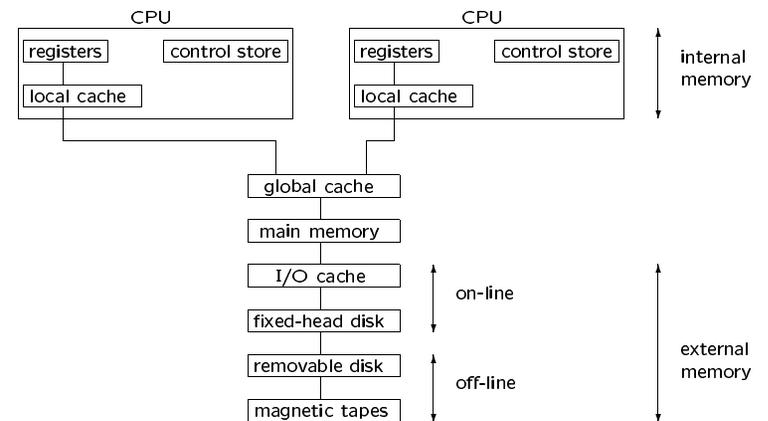
- CPU runs program instructions only when program is in memory
- programs do I/O sometimes \Rightarrow CPU wasted
- solution: **multiprogramming** (multitasking)
 - multiple programs (processes) share the memory
 - one program, at a time, gets CPU
 - simultaneous resource possession (CPU and memory)
 - better performance (response time, throughput)

[V.A] Long/Short Term Schedulers [131]



- long-term : job scheduler (memory management)
 - which jobs allocated memory and allowed into the system
- short-term: CPU scheduler (process management)
 - which job allocated the CPU

[V.A] Memory: Storage Hierarchy [132]



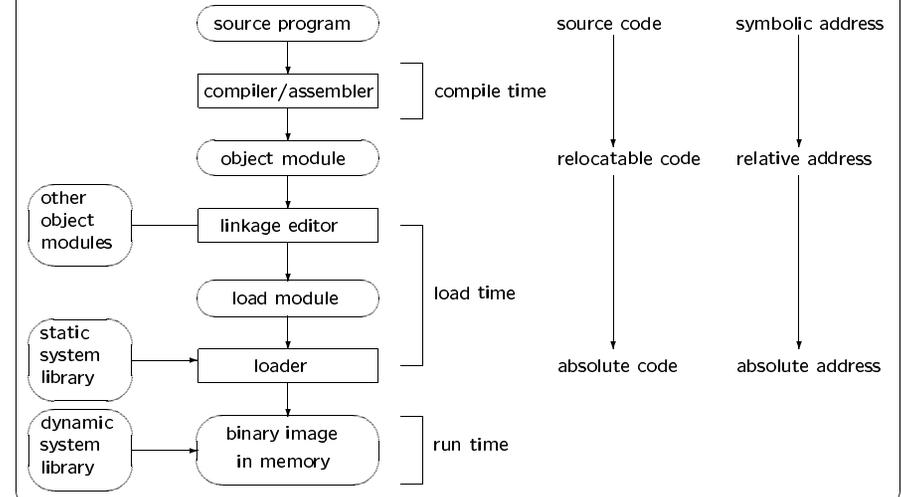
[V.A] Memory: Devices

[133]

memory type	capacity	access time	technology
control store	1K-32K words	3-100 ns	SRAM, ROM
registers	8-256 words	3-10 ns	SRAM
CPU cache	8 KB - 1 MB	3-100 ns	SRAM, DRAM
main memory	128 KB - 4 GB	20-200 ns	DRAM
I/O cache	32 KB - 1 MB	20-200 ns	DRAM
disk drive	1 MB -100 GB	10-65 ms	magnetic disk
tape drive	20 MG or more	seconds	magnetic tape

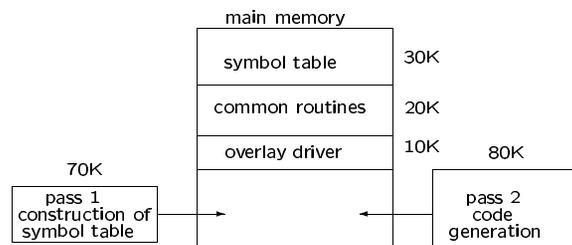
[V.A] Memory: Address Binding

[134]



[V.A] Memory: Overlays

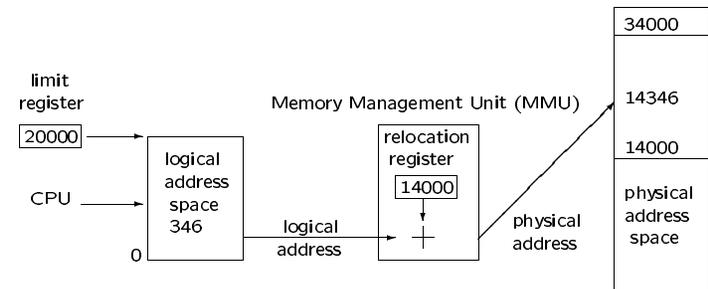
[135]



- program does pass 1; then pass 2
- programmer controls memory
- used when physical memory size was limited

[V.A] Memory: Dynamic Relocation

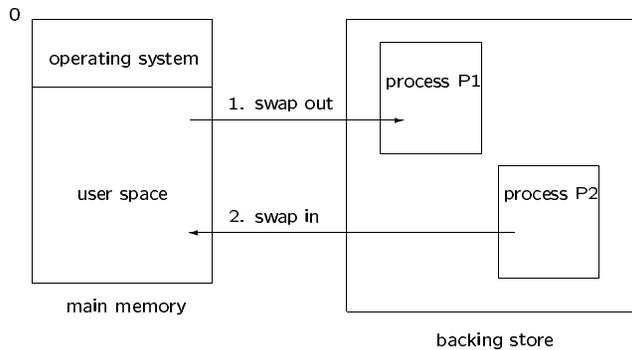
[136]



- logical address : as seen by CPU
- physical address: as seen by memory unit
- to relocate program/data: move and change register
 - save both registers during a context switch

[V.A] Memory: Swapping

[137]



- swap-out: executing job (round-robin)
- swap-in : old job (maybe dynamic relocation)
- roll-out,roll-in: low priority for high priority
- 2 jobs × (100K size × 1000K/sec + 8ms latency) = 216ms
- quantum: much larger than 216ms

[V.A] Memory: Contiguous Allocation

[138]

- program and data space are in sequential memory addresses
- single-partition: OS and one program
 - relocation register and limit register
 - protect OS and user program from each other
- multiple-partitions: many programs with their own partition
 - required for multi-programming
 - fixed-partition scheme (IBM OS/360 MFT)
 - variable-partition scheme (IBM OS/360 MVT)

[V.A] Memory: Scheduling Example

[139]

process	job queue	
	memory	time
P_1	600K	10
P_2	1000K	5
P_3	300K	20
P_4	700K	8
P_5	500K	15
TOTAL		58

Scheduling Discipline:

Job: FCFS

CPU: Round-Robin (Quantum=1)

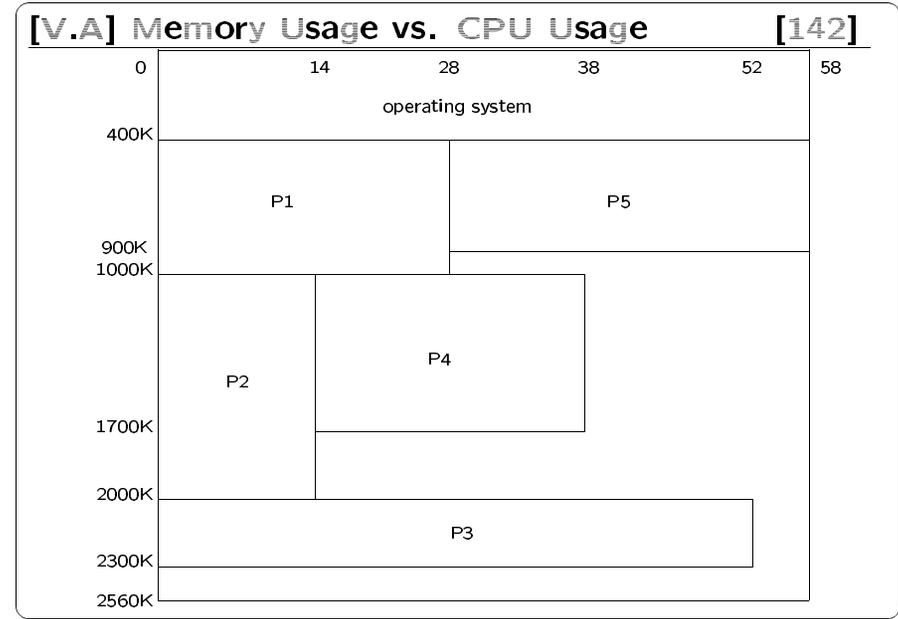
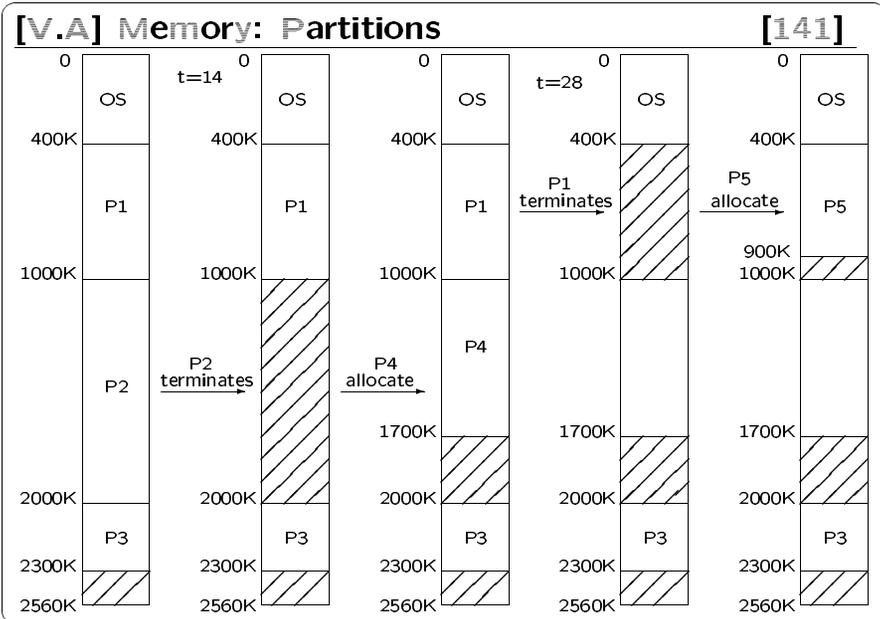
Memory: 2560 KB

Dynamic Storage Allocation Strategy: Best fit

[V.A] Memory: Scheduling Details

[140]

time	1	2	3	4	5	6	7	8	9	10	11	12	13	14
P_1	1	1	1	2	2	2	3	3	3	4	4	4	5	5
P_2	0	1	1	1	2	2	2	3	3	3	4	4	4	5
P_3	0	0	1	1	1	2	2	2	3	3	3	4	4	4
time	15	16	17	18	19	20	21	22	23	24	25	26	27	28
P_1	5	6	6	6	7	7	7	8	8	8	9	9	9	10
P_4	0	0	1	1	1	2	2	2	3	3	3	4	4	4
P_3	5	5	5	6	6	6	7	7	7	8	8	8	9	9
time	29	30	31	32	33	34	35	36	37	38				
P_5	0	0	1	1	1	2	2	2	3	3				
P_4	5	5	5	6	6	6	7	7	7	8				
P_3	9	10	10	10	11	11	11	12	12	12				
time	39	40	41	42	43	44	45	46	47	48	49	50	51	52
P_5	3	4	4	5	5	6	6	7	7	8	8	9	9	10
P_3	13	13	14	14	15	15	16	16	17	17	18	18	19	19
time	53	54	55	56	57	58								
P_5	10	11	12	13	14	15								
P_3	20													



[V.A] Memory: Search Strategies [143]

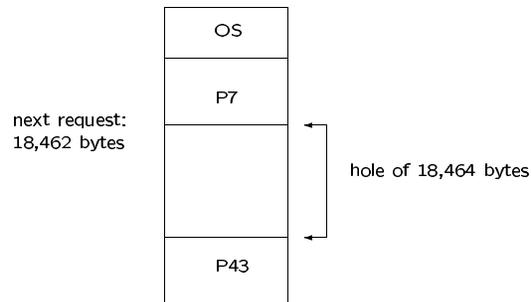
- list of memory holes: which one to use?
- first-fit : allocate the first hole which is big enough
- best-fit : allocate the smallest hole which is big enough
 - produces the smallest left-over hole
- worst-fit: allocate the largest hole
 - produces the largest left-over hole

strategy	search time	memory utilization
first-fit	fast	good
best-fit	slow	good
worst-fit	slow	bad

[V.A] Memory: External Fragmentation [144]

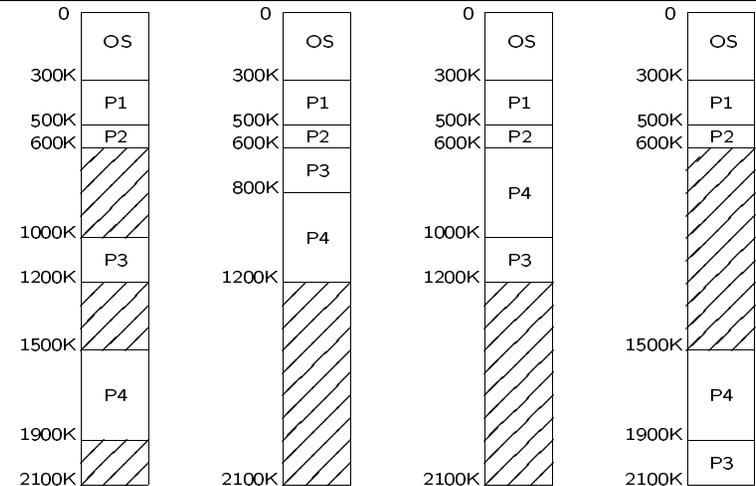
- request for memory cannot be satisfied
 - even though total free memory is sufficient
 - but not contiguous (broken into small holes)
- first-fit or best-fit may be better
- 50-percent rule:
 - given N allocated blocks $\Rightarrow 0.5N$ blocks unusable (1/3 wasted)
- solution: compaction or paging

[V.A] Memory: Internal Fragmentation [145]



- allocated memory slightly larger than requested memory
- overhead to keep track of small hole may be larger than hole itself

[V.A] Memory: Compaction Techniques [146]

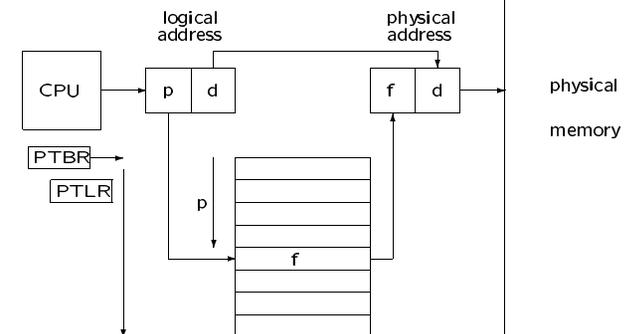


- move 600K, or 400K, or 200K
- dynamic relocation and swapping

[V.A] Memory: Paging [147]

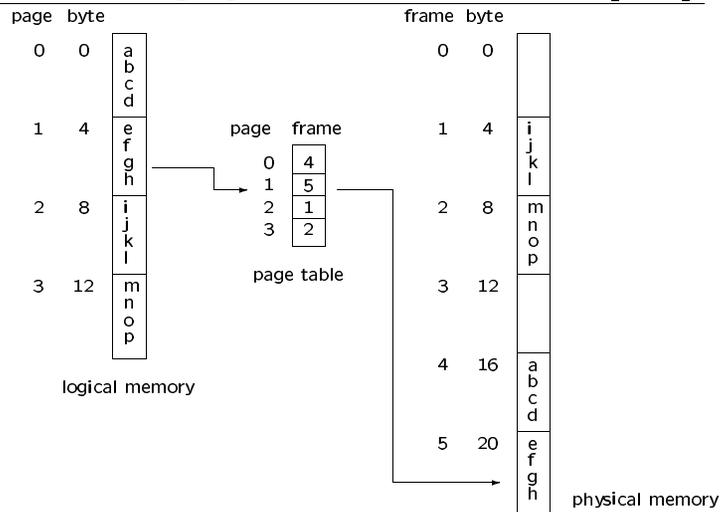
- goal: eliminate external fragmentation
- (allow noncontiguous processes)
- each process is a set of fixed-size pages
- pages are stored in same-size physical memory "frames"
- page table "connects" logical pages with physical frames
- may still have internal fragmentation

[V.A] Memory: Paging Hardware [148]



- logical address: p (page) and d (displacement/offset) in page
- physical address: f (frame) and d (displacement/offset) in frame
- PTBR: Page Table Base Register
- PTLR: Page Table Length Register

[V.A] Memory: Paging [149]



- byte 'g': logical address = 1 10 = 6
- byte 'g': physical address = 101 10 = 22

[V.A] Memory: Logical Address [150]

- page size: 2^n
- logical address space: 2^m
- page number: high-order $m - n$ bits
- page offset: low-order n bits

page number	page offset
p	d
$m - n$	n

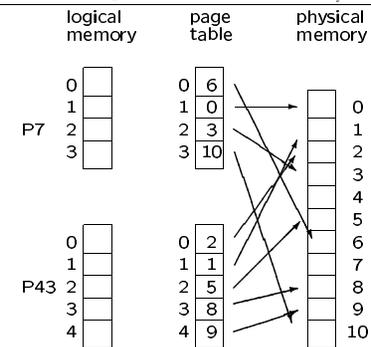
[V.A] Memory: Logical Addresses [151]

- $n = 10$ and $m = 32$

```
typedef union {
    struct {
        unsigned int page :22;
        unsigned int offset:10;
    } logical;
    int addr;
} laddr;
laddr l;
```

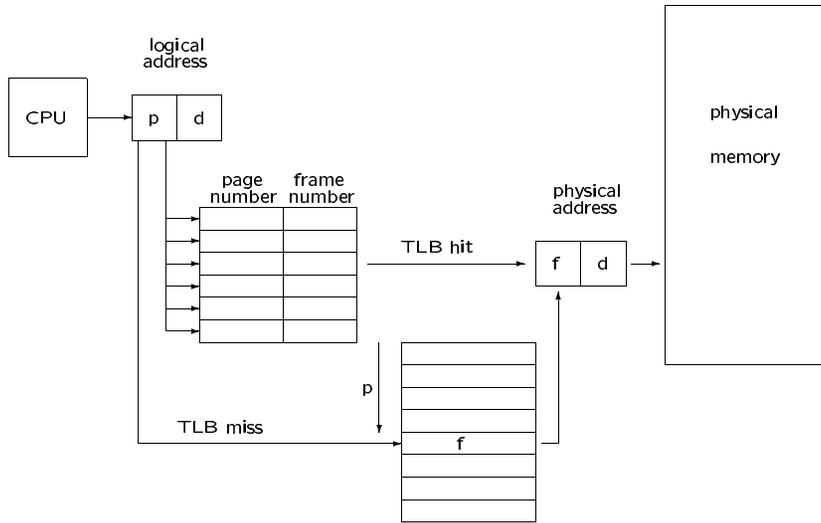
- page 0: 0..1023
- page 1: 1024..2047
- page $2^{m-n} = 4,194,304$
- `l = 1026; /* page=1, offset=2 */`

[V.A] Memory: Multiple Processes/Tables [152]



- each process has own page table
- small page tables: fast registers
- large page tables: main memory
 - context-switch: save/load PTBR
 - memory access: $2 \times 100 \text{ ns} = 200 \text{ ns}$

[V.A] Translation Look-Aside Buffers [153]



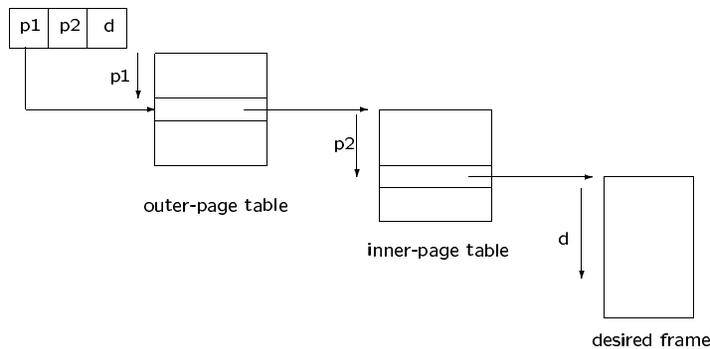
[V.A] Memory: TLB [154]

- small, fast, parallel lookup cache of “associative” registers
- if miss, then go to full table in memory
- hit ratio: 80% (depends on number of registers)
- effective memory access =
 - $0.80 \times (20 \text{ ns registers} + 100 \text{ ns memory}) +$
 - $0.20 \times (20 \text{ ns} + 100 \text{ ns} + 100 \text{ ns}) = 140 \text{ ns}$
- hit ratio: 98% \Rightarrow 122 ns

[V.A] Memory: Table Fragmentation [154]

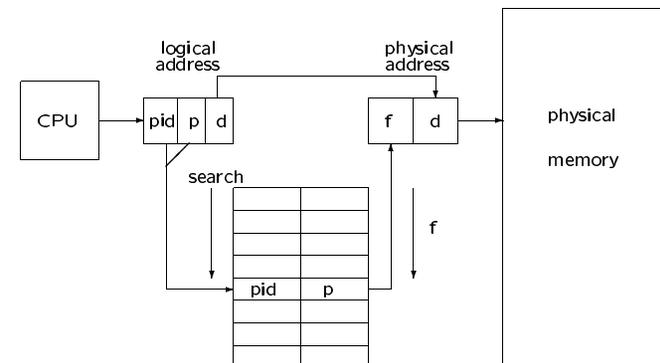
- page tables can be large (say 4MB for each process)
- creates its own form of memory fragmentation
- solution 1: pages of page tables (multilevel paging)
- solution 2: all processes share ONE page table (of “active” pages)
 - inverted page table (really used with “virtual” memory)

[V.A] Memory: Multilevel Paging [155]



- reduces the size of one monolithic page table/process
- 3-level paging: SPARC
- 4-level paging: Motorola 68030

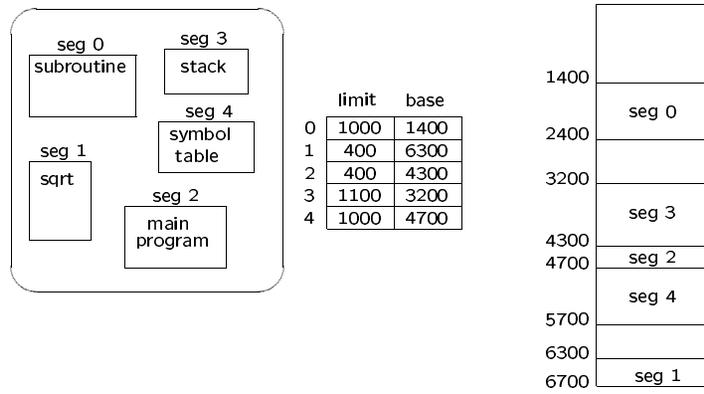
[V.A] Memory: Inverted Page Table [156]



- one entry for each real page (frame) of memory
- why is the offset f equal to the frame number?
- what happens if a fault occurs (page is not in memory)?
- where is the external page table?

[V.A] Memory: Segmentation [157]

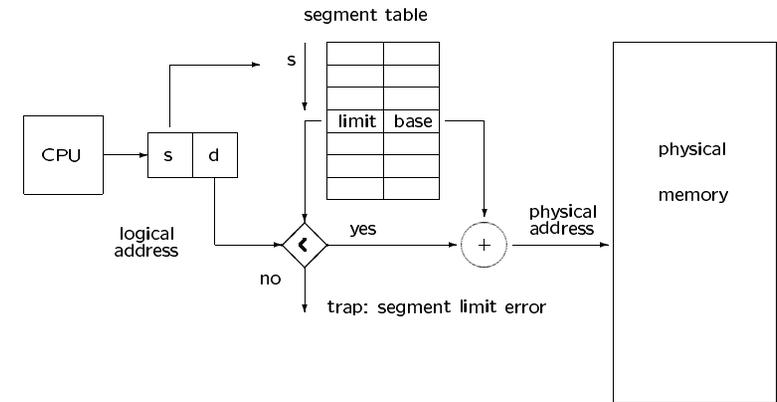
[157]



- supports user view of memory
- logical address space is a collection of segments (name and length)
- address = [segment name or number] [offset]
- closely related to partitions (but several per program)

[V.A] Memory: Segmentation Hardware [158]

[158]



- segment table in registers
- segment table in memory (with STBR and STLR)
 - may need associative registers for cache

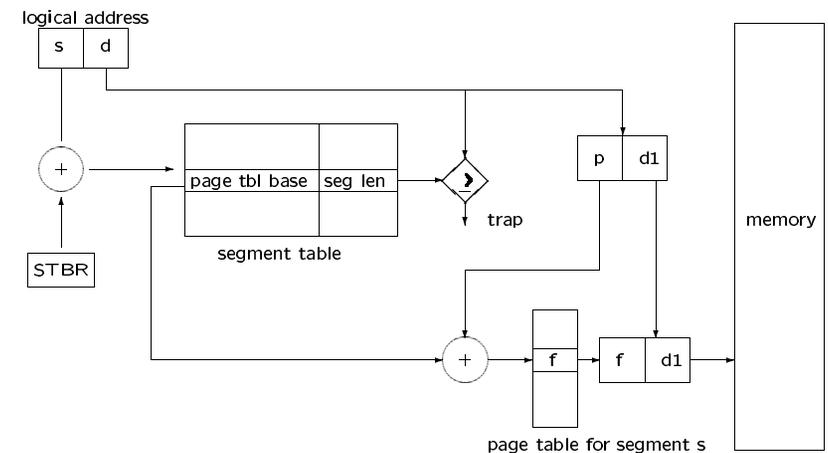
[V.A] Memory: Paging vs. Segmentation [159]

[159]

- paging
 - no external fragmentation
 - large tables
- segmentation
 - external fragmentation
 - small tables
 - easy protection at segment level
 - easy sharing at segment level
- solution: paged segmentation

[V.A] Paged Segmentation (MULTICS) [160]

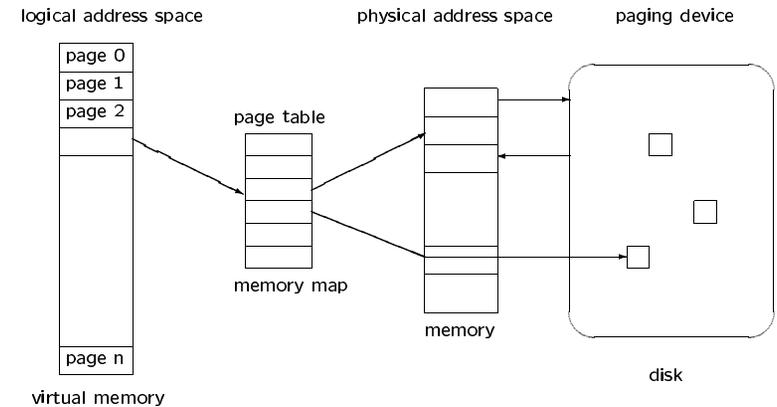
[160]



[V.B] Virtual Memory: Introduction [161]

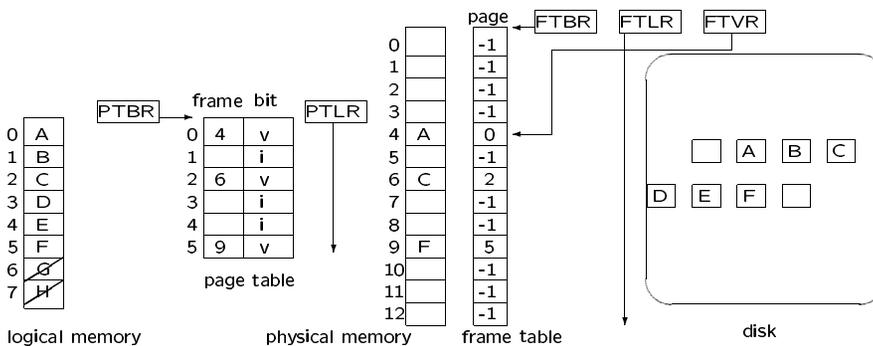
- up to now: all of a process in main memory (somewhere)
 - partitions, pages, segments
- now: virtual memory
 - allow execution of processes which may be partially in memory
- benefits:
 - programs can be large and memory can be small
 - increased multiprogramming ⇒ better performance
 - less I/O for loading/swapping programs
- why programs don't need to be entirely in memory:
 - code for unusual error conditions
 - more memory allocated than is needed
 - some features of program rarely used
- VM is the separation of user logical memory from physical memory

[V.B] VM: Page Table [162]



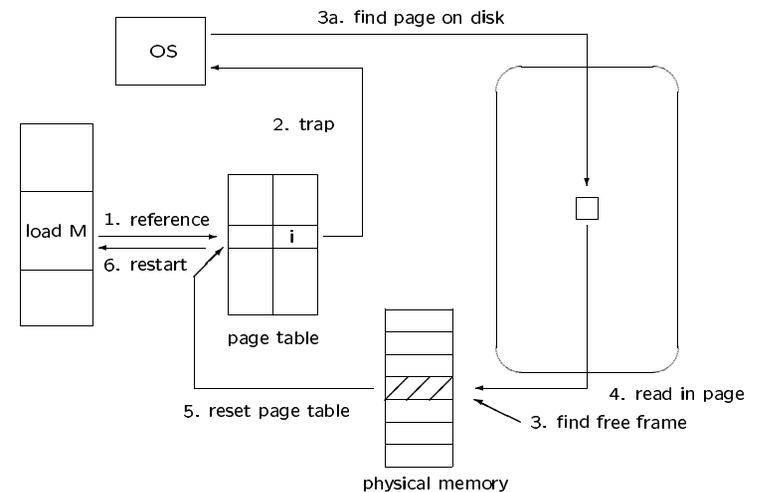
- logical address vs. physical address
- demand paging: only “necessary” pages are brought into memory

[V.B] VM: Valid-Invalid Bit and Frame Table [163]



- page table: points to frames
- frame table: points to pages
- (this implementation only works with one process)
- bit: is page loaded into a frame?

[V.B] VM: Page Fault [164]



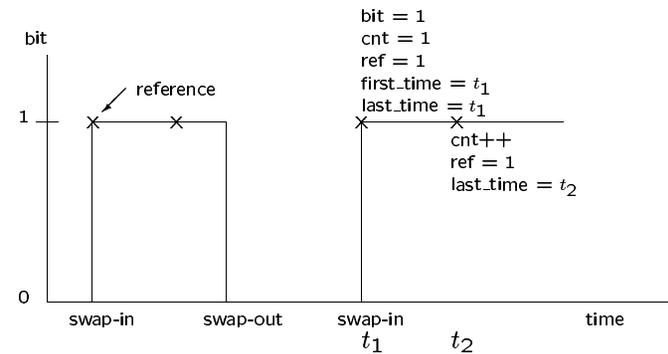
[V.B] VM: Page Fault Details

[165]

- trap to OS - save user registers and process state
- determine that interrupt was a page fault
- check page reference and location on disk
- issue read from the disk to a free frame (old page not DIRTY)
 - wait in queue for device
 - wait for seek and/or latency
 - begin transfer
- while waiting, allocate CPU to other user
- interrupt from the disk (I/O completed)
- save registers and process state of other user
- determine that interrupt was from disk
- correct page table (desired page is now in memory)
- wait for CPU to be allocated to this process again
- restore user registers, process state, new page table
- RESUME execution

[V.B] VM: History of Page A

[166]



[V.B] VM: Page and Frame Table in MEM [167]

```
typedef struct {
    unsigned int frame:22;
    unsigned int bit : 1; /* 1 => frame is valid */
    unsigned int dirty: 1; /* 1 => frame has been updated; needs writing */
    unsigned int ref : 1; /* 1 => the page has been referenced */
    int cnt; /* how many references */
    int first_time; /* swap-in time for page */
    int last_time; /* last time referenced */
} page_entry;

typedef struct {
    page_entry entries[MAXPAGES];
} page_table;

typedef struct {
    int page;
} frame_entry;

typedef struct {
    frame_entry entries[MAXFRAMES];
} frame_table;

page_table *PTBR;
frame_table *FTBR;
```

[V.B] VM: Page Reference in MEM [168]

[168]

```
void page_ref(page) {
    int frame, old_page;
    clock++;
    page_range(page);
    frame = PTBR->entries[page].frame;
    if (PTBR->entries[page].bit == INVALID) {
        frame = find_frame();
        old_page = FTBR->entries[frame].page;
        if (old_page != NIL) {
            if (PTBR->entries[old_page].dirty)
                write_block(old_page, frame);
            reset_page(old_page);
        }
        reset_page(page);
        read_block(page, frame);
        set(page, frame);
        PTBR->entries[page].first_time = clock;
    }
    PTBR->entries[page].cnt++;
    PTBR->entries[page].ref = REF;
    PTBR->entries[page].last_time = clock;
}
```

[V.B] VM: Sets and Resets in MEM

[169]

```
void reset_page(int page) {
    page_range(page);
    PTBR->entries[page].frame = 0;
    PTBR->entries[page].bit = INVALID;
    PTBR->entries[page].ref = NOREF;
    PTBR->entries[page].cnt = 0;
}

void reset_frame(int frame) {
    frame_range(frame);
    FTBR->entries[frame].page = NIL;
}

void set(int page,int frame) {
    page_range(page);
    frame_range(frame);
    PTBR->entries[page].bit = VALID;
    PTBR->entries[page].frame = frame;
    FTBR->entries[frame].page = page;
}
```

[V.B] Initialization and Deletion in MEM

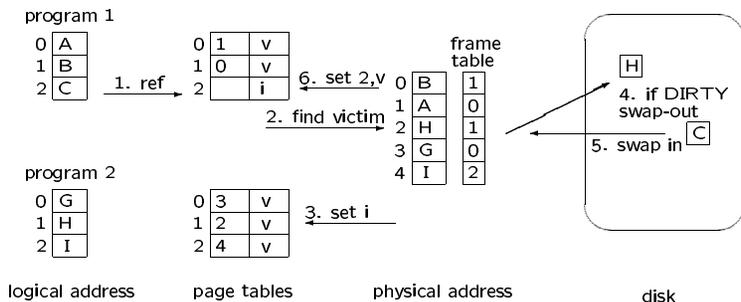
[170]

```
void init_tables(int tot_pages,int tot_frames) {
    int page,frame;
    PTLR = tot_pages;
    PTBR = (page_table *) malloc(sizeof(page_entry)*PTLR);
    FTLR = tot_frames;
    FTBR = (frame_table *) malloc(sizeof(frame_entry)*FTLR);
    FTVR = -1;
    for (page=0; page<PTLR; page++)
        reset_page(page);
    for (frame=0; frame<FTLR; frame++)
        reset_frame(frame);
}

void delete(int page) {
    page_range(page);
    if (PTBR->entries[page].bit == VALID) {
        frame = PTBR->entries[page].frame;
        if (PTBR->entries[page].dirty)
            write_block(page,frame);
        reset_frame(frame);
    }
    reset_page (page);
}
```

[V.B] VM: Page Replacement

[171]



- frame table's page numbers do not work for multiple programs
- frame table should be collection of pointers to page entries

```
typedef struct {
    page_entry *ptr;
} frame_entry;
```

[V.B] VM: Page Replacement in MEM

[172]

```
int find_frame() {
    int frame;
    if (!virtual) {
        frame = find_free_frame();
        if (frame == NIL) trap("NOT ENOUGH FRAMES FOR NON-VIRTUAL MEMORY");
    }
    else
        switch (alg) {
            case FIFO : frame = fifo_alg(); break;
            case OPT : frame = opt_alg(ref_ptr+1,tot_ref,reference); break;
            case LRU_TIME: frame = lru_time_alg(); break;
            case LRU_REF : frame = lru_ref_alg(ref_ptr+1,tot_ref,reference); break;
            case CLOCK : frame = clock_alg(); break;
            case LFU : frame = lfu_alg(); break;
            case MFU : frame = mfu_alg(); break;
            case ENHANCED: frame = enhanced_alg(); break;
            case LRU_STACK: frame = stack_alg(); break;
        }
    return(frame);
}
```

- choice of algorithm affects performance (number of page faults)

[V.B] VM: Find Free Frame in MEM [173]

```
int find_free_frame() {
int save_reg;
save_reg = FTVR;
do {
FTVR++;
if (FTVR >= FTLR) FTVR = 0;
if (FTBR->entries[FTVR].page == NIL) return(FTVR);
} while (FTVR != save_reg);
return(NIL);
}
```

[V.B] VM: FIFO in MEM [174]

- “replace the oldest page” (or use a free frame)
- usually implemented with a linked (FIFO) list

```
int fifo_alg() {
int save_reg,page,frame,earliest_time;
save_reg = FTVR;
earliest_time = INFINITY;
do {
FTVR++;
if (FTVR >= FTLR) FTVR = 0;
page = FTBR->entries[FTVR].page;
if (page == NIL) return(FTVR);
if (PTBR->entries[page].first_time < earliest_time) {
earliest_time = PTBR->entries[page].first_time;
frame = FTVR;
}
} while (FTVR != save_reg);
FTVR = frame;
return(FTVR);
}
```

[V.B] VM: MEM - A Simulator [175]

```
mem> h
i(nit pages frames [pages in 1..20, frames in 1..20]
r(ead page [page in 0..19]
w(rite page [page in 0..19]
d(elete page [page in 0..19]
l(ogical addr [page:22bits offset:10bits]
a(lgorithm 0:FIFO|1:OPT|2:LRU_TIME|3:LRU_REF|4:CLOCK|5:LFU|6:MFU|
7:ENHANCED|8:LRU_STACK
m(ode 0:DETAILED|1:FRAMES|2:SUMMARY
v(irtual 0:OFF|1:ON
n(ogo [save all reads for go]
g(o [do all reads; for SUMMARY try all frames]
q(uit
mem> quit
```

[V.B] VM: Page and Frame Tables in MEM [176]

```
mem> alg 0
mem> mode 0
mem> init 8 3
mem> logical 1026
Page Tbl Frame Tbl (Alg:0)
frame bit page
0 0 0 1 < Fault 1
> 1 0 1 -1
2 0 0 -1
3 0 0
4 0 0
5 0 0
6 0 0
7 0 0
Logical Address page : 1 offset: 2
Physical Address frame: 0 offset: 2
```

[V.B] VM: FIFO using MEM**[177]**

```

mem> alg 0
mem> mode 1
mem> init 8 3
mem> r 7 r 0 r 1 r 2 r 0 r 3 r 0 r 4 r 2 r 3 r 0 r 3 r 2 r 1 r 2 r 0 r 1 r 7
r 0 r 1
page: 7 faults: 1 frames: 7 -1 -1 time: 1
    0      2      7 0 -1      1 2
    1      3      7 0 1      1 2 3
    2      4      2 0 1      4 2 3
    0      4      2 0 1      4 2 3
    3      5      2 3 1      4 6 3
    0      6      2 3 0      4 6 7
    4      7      4 3 0      8 6 7
    2      8      4 2 0      8 9 7
    3      9      4 2 3      8 9 10
    0     10      0 2 3     11 9 10
    3     10      0 2 3     11 9 10
    2     10      0 2 3     11 9 10
    1     11      0 1 3     11 14 10
    2     12      0 1 2     11 14 15
    0     12      0 1 2     11 14 15
    1     12      0 1 2     11 14 15
    7     13      7 1 2     18 14 15
    0     14      7 0 2     18 19 15
    1     15      7 0 1     18 19 20

```

[V.B] VM: Optimal Algorithm**[178]**

- “replace the page that will not be used for the longest time”
- lowest page-fault rate of all algorithms
- requires advanced knowledge of page reference string
- useful for comparison studies

[V.B] VM: OPT using MEM**[179]**

```

mem> nogo
mem> r 7 r 0 r 1 r 2 r 0 r 3 r 0 r 4 r 2 r 3 r 0 r 3 r 2 r 1 r 2 r 0 r 1 r 7
r 0 r 1
mem> go
page: 7 faults: 1 frames: 7 -1 -1
    0      2      7 0 -1
    1      3      7 0 1
    2      4      2 0 1
    0      4      2 0 1
    3      5      2 0 3
    0      5      2 0 3
    4      6      2 4 3
    2      6      2 4 3
    3      6      2 4 3
    0      7      2 0 3
    3      7      2 0 3
    2      7      2 0 3
    1      8      2 0 1
    2      8      2 0 1
    0      8      2 0 1
    1      8      2 0 1
    7      9      7 0 1
    0      9      7 0 1
    1      9      7 0 1

```

[V.B] VM: Least Recently Used (LRU)**[180]**

- FIFO: when a page was brought into memory in the past
- OPT: when a page is used in the future
- LRU: when a page was used in the past
 - “replace the page that has not been used for the longest time”
 - requires a logical clock time for each page (LRU_TIME)
 - or a stack of recent page references (LRU_STACK)
 - or a list of all references (LRU_REF)
 - same as OPT but on reverse of page reference string
 - optimal algorithm looking backward in time

[V.B] VM: LRU_TIME using MEM [181]

- better than FIFO (15) but worst than OPT (9)

page:	7	faults:	1	frames:	7 -1 -1	time:	1
0	2	7	0	-1	1	2	
1	3	7	0	1	1	2	3
2	4	2	0	1	4	2	3
0	4	2	0	1	4	5	3
3	5	2	0	3	4	5	6
0	5	2	0	3	4	7	6
4	6	4	0	3	8	7	6
2	7	4	0	2	8	7	9
3	8	4	3	2	8	10	9
0	9	0	3	2	11	10	9
3	9	0	3	2	11	12	9
2	9	0	3	2	11	12	13
1	10	1	3	2	14	12	13
2	10	1	3	2	14	12	15
0	11	1	0	2	14	16	15
1	11	1	0	2	17	16	15
7	12	1	0	7	17	16	18
0	12	1	0	7	17	19	18
1	12	1	0	7	20	19	18

[V.B] VM: LRU_STACK [182]

- keep a stack of every page which owns a frame
- on each reference
 - find the page within the stack and remove it
 - push the page onto the top of the stack
- on page fault
 - if FREE frame, take it
 - otherwise, LRU page is at the bottom of the stack
 - return its frame

[V.B] VM: LRU_STACK using MEM [183]

page:	7	faults:	1	frames:	7 -1 -1	stack:	7
0	2	7	0	-1	7	0	
1	3	7	0	1	7	0	1
2	4	2	0	1	0	1	2
0	4	2	0	1	1	2	0
3	5	2	0	3	2	0	3
0	5	2	0	3	2	3	0
4	6	4	0	3	3	0	4
2	7	4	0	2	0	4	2
3	8	4	3	2	4	2	3
0	9	0	3	2	2	3	0
3	9	0	3	2	2	0	3
2	9	0	3	2	0	3	2
1	10	1	3	2	3	2	1
2	10	1	3	2	3	1	2
0	11	1	0	2	1	2	0
1	11	1	0	2	2	0	1
7	12	1	0	7	0	1	7
0	12	1	0	7	1	7	0
1	12	1	0	7	7	0	1

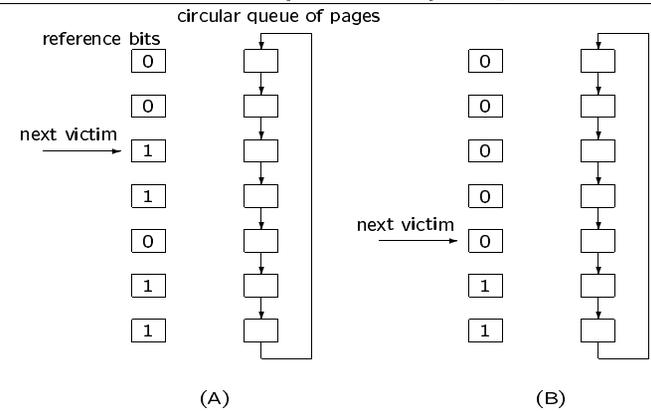
[V.B] VM: LRU_REF [184]

- new page reference : 4
 - current frame pages: 2 0 3
 - current ref string : 7 0 1 2 0 3 0 4
 - reverse ref string : 4 0 3 0 2 1 0 7
- ↑
 ↑
 ↑
- apply OPT algorithm: 2 0 3 → 4 0 3

[V.B] VM: LRU Approximations [185]

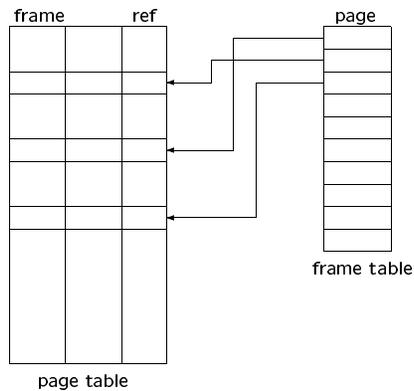
- hardware usually does not support LRU
- but does support REF bit
 - interrupt every 100 msec
 - move REF bit to 8-bit shift register (and clear)
 - 00000000 \Rightarrow no refs in last 8 periods
 - 11111111 \Rightarrow at least one ref in each period
 - 01111111 \Rightarrow no ref in the recent period
 - smallest integer $\Rightarrow \approx$ LRU
 - "additional-reference-bits algorithm"
- ONE bit of history
 - just use the REF bit itself
 - "second-chance" page-replacement algorithm

[V.B] Second-Chance (CLOCK) Algorithm [186]



- if REF then clear bit (NOREF) and move on
- next victim is a NOREF
- if all pages have been referenced then CLOCK = FIFO

[V.B] VM: CLOCK Circular Queue [187]



- frame table is circular queue of pages

[V.B] VM: CLOCK using MEM [188]

- better than FIFO (15) but worse than LRU (12)

page:	faults:	frames:	ref:
7	1	7 -1 -1	1
0	2	7 0 -1	1 1
1	3	7 0 1	1 1 1
2	4	2 0 1	1 0 0
0	4	2 0 1	1 1 0
3	5	2 0 3	1 0 1
0	5	2 0 3	1 1 1
4	6	4 0 3	1 0 0
2	7	4 2 3	1 1 0
3	7	4 2 3	1 1 1
0	8	4 2 0	0 0 1
3	9	3 2 0	1 0 1
2	9	3 2 0	1 1 1
1	10	3 1 0	0 1 0
2	11	3 1 2	0 1 1
0	12	0 1 2	1 1 1
1	12	0 1 2	1 1 1
7	13	0 7 2	0 1 0
0	13	0 7 2	1 1 0
1	14	0 7 1	1 1 1

[V.B] VM: ENHANCED Second Chance [189]

- if victim is dirty then it costs time to write it out
- better to choose a non-dirty victim (Macintosh VM)

```
class1: (ref=0,dirty=0) => good page to replace
class2: (0,1) => not as good because old page needs to be written
class3: (1,0) => not good because its recently referenced
class4: (1,1) => definitely not good because it also has to be written
```

```
PASS:  do
    a. if empty frame, take it
    b. if class1, take it
    c. if class2, then record first instance
    d. clear ref bit if class 2 has not been found yet
until complete pass
if class2 was found, take first instance
invariant1: there are no free frames
invariant2: there are only class1 and class2 because
            all bits were cleared.
```

if the first PASS does not succeed, try one more PASS

[V.B] VM: ENHANCED using MEM [190]

- if no writing, ENHANCED = CLOCK
- in the case below, WRITES alter the sequence of events

```

rpage: 7  faults: 1  frames: 7 -1 -1  ref,dirty: 1,0
        0          2          7 0 -1      1,0 1,0
wpage: 1          3          7 0 1      1,0 1,0 1,1
        2          4          2 0 1      1,0 0,0 0,1
        0          4          2 0 1      1,0 1,0 0,1
        3          5          2 0 3      1,0 0,0 1,0
wpage: 0          5          2 0 3      1,0 1,1 1,0
        4          6          4 0 3      1,0 0,1 0,0
        2          7          4 0 2      1,0 0,1 1,0
        3          8          4 3 2      0,0 1,0 1,0
        0          9          0 3 2      1,0 1,0 0,0
        3          9          0 3 2      1,0 1,0 0,0
        2          9          0 3 2      1,0 1,0 1,0
        1          10         0 1 2      0,0 1,0 0,0
        2          10         0 1 2      0,0 1,0 1,0
        0          10         0 1 2      1,0 1,0 1,0
        1          10         0 1 2      1,0 1,0 1,0
        7          11         0 1 7      0,0 0,0 1,0
        0          11         0 1 7      1,0 0,0 1,0
        1          11         0 1 7      1,0 1,0 1,0
    
```

[V.B] VM: Counting Algorithms [191]

- maintain a count of the number of references to a page
- Least Frequently Used (LFU): “replace page with smallest count”
 - active pages will have large counts
 - but initialization pages will have large counts
- Most Frequently Used (MFU): “replace page with largest count”
 - recent pages are active pages but will have small counts
- neither are common
- do not approximate OPT well

[V.B] VM: LFU using MEM [192]

```

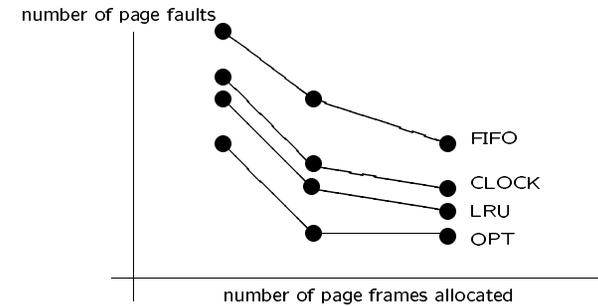
page: 7  faults: 1  frames: 7 -1 -1  count: 1
        0          2          7 0 -1      1 1
        1          3          7 0 1      1 1 1
        2          4          2 0 1      1 1 1
        0          4          2 0 1      1 2 1
        3          5          2 0 3      1 2 1
        0          5          2 0 3      1 3 1
        4          6          4 0 3      1 3 1
        2          7          4 0 2      1 3 1
        3          8          3 0 2      1 3 1
        0          8          3 0 2      1 4 1
        3          8          3 0 2      2 4 1
        2          8          3 0 2      2 4 2
        1          9          3 0 1      2 4 1
        2          10         3 0 2      2 4 1
        0          10         3 0 2      2 5 1
        1          11         3 0 1      2 5 1
        7          12         3 0 7      2 5 1
        0          12         3 0 7      2 6 1
        1          13         3 0 1      2 6 1
    
```

[V.B] VM: MFU using MEM

[193]

page:	faults:	frames:	count:
7	1	7 -1 -1	1
0	2	7 0 -1	1 1
1	3	7 0 1	1 1 1
2	4	2 0 1	1 1 1
0	4	2 0 1	1 2 1
3	5	2 3 1	1 1 1
0	6	2 3 0	1 1 1
4	7	4 3 0	1 1 1
2	8	4 2 0	1 1 1
3	9	4 2 3	1 1 1
0	10	0 2 3	1 1 1
3	10	0 2 3	1 1 2
2	10	0 2 3	1 2 2
1	11	0 1 3	1 1 2
2	12	0 1 2	1 1 1
0	12	0 1 2	2 1 1
1	12	0 1 2	2 2 1
7	13	7 1 2	1 2 1
0	14	7 0 2	1 1 1
1	15	7 0 1	1 1 1

[V.B] Replacement Algorithm Performance [194]

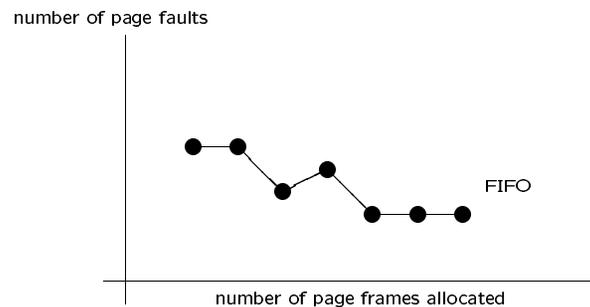


- more frames \Rightarrow less faults
- if [frames allocated] = 1?
- if [frames allocated] = [logical pages]?

```
mem> mode 2
mem> init 8 8
```

[V.B] VM: Belady's Anomaly

[195]



- more frames \Rightarrow more faults
- FIFO may replace pages that are just about to be used again

[V.B] VM: Belady's Anomaly using MEM [196]

page:	faults:	frames:	time:
1	1	1 -1 -1	1
2	2	1 2 -1	1 2
3	3	1 2 3	1 2 3
4	4	4 2 3	4 2 3
1	5	4 1 3	4 5 3
2	6	4 1 2	4 5 6
5	7	5 1 2	7 5 6
1	7	5 1 2	7 5 6
2	7	5 1 2	7 5 6
3	8	5 3 2	7 10 6
4	9	5 3 4	7 10 11
5	9	5 3 4	7 10 11
page:	faults:	frames:	time:
2	2	1 2 -1 -1	1 2
3	3	1 2 3 -1	1 2 3
4	4	1 2 3 4	1 2 3 4
1	4	1 2 3 4	1 2 3 4
2	4	1 2 3 4	1 2 3 4
5	5	5 2 3 4	7 2 3 4
1	6	5 1 3 4	7 8 3 4
2	7	5 1 2 4	7 8 9 4
3	8	5 1 2 3	7 8 9 10
4	9	4 1 2 3	11 8 9 10
5	10	4 5 2 3	11 12 9 10

[V.B] VM: Allocation of Frames

[197]

- up to now: just one process in memory and all frames are available
- m frames available for n processes
- equal allocation: m/n frames per process
 - but small processes may get too many frames
- proportional allocation based on size of process: $s_i/S \times m$
 - $S = \sum s_i$
 - may want to increase allocation for high-priority processes
- what happens if a fault occurs and no free frames?
 - local replacement: reuse a frame from the faulting process
 - can not make use of under-utilized frames
 - global replacement: take a frame from any process
 - high-priority takes from low-priority ("stealing")
 - "fate" of a process depends on the behavior of others
 - tends to increase system throughput
 - but may cause thrashing

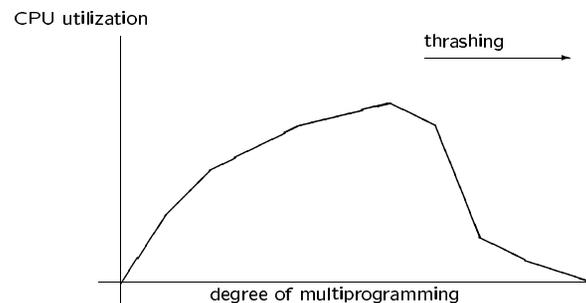
[V.B] VM: Thrashing

[198]

- scenario 1:
 - process has a small number of frames (allocation or stealing)
 - process has a large number of active pages
 - process spends more time paging than executing
- scenario 2:
 - OS monitors CPU utilization
 - if low utilization then increase degree of multiprogramming
 - new process takes frames from other processes
 - they start thrashing
 - utilization decreases and OS adds more processes
 - more thrashing

[V.B] VM: Thrashing

[199]



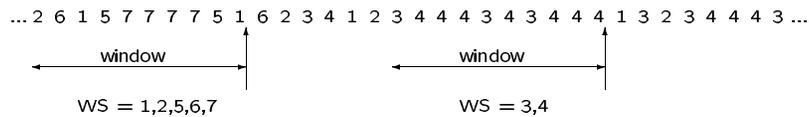
[V.B] VM: Thrashing Solutions

[200]

- use a local replacement algorithm
 - thrashing process cannot steal frames
 - but queue time (paging device) will increase for ALL processes
- provide a process as many frames as it "needs"
 - may suspend other processes (and free up their frames)
 - "need" based on *locality model*
 - set of pages that are actively used together
 - subroutines or data structures
 - if full set in memory then no more faults (until new locality)

[V.B] VM: Working-Set Model [201]

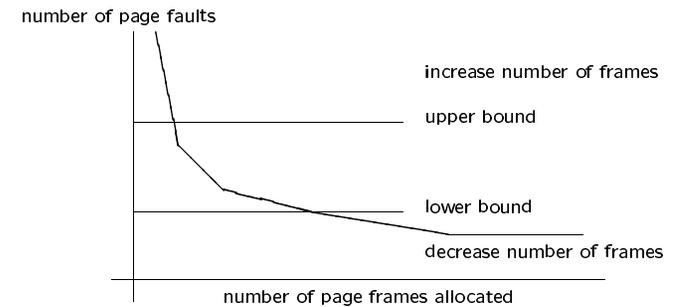
- approximation to the program's locality
- let Δ be the *working-set window*
- keep list of all pages used during the last Δ page references



- small $\Delta \Rightarrow$ does not encompass locality
- large $\Delta \Rightarrow$ given too many frames
- let WSS_i be the size of the working set for process i
- demand $D = \sum WSS_i$
- if not enough frames, then suspend processes (and free frames)
- prevents thrashing and keeps multiprogramming high as possible
- optimizes CPU utilization

[V.B] VM: Page-Fault Frequency Strategy [202]

- PFF Strategy
- direct approach to solve thrashing
- may need to suspend other processes to get more frames



[V.B] VM: Program Structure [203]

- assume OS allocates < 128 frames for this process:
- each row contains one page

```
int A[128][128]; /* ROW MAJOR: A[0][0], A[0][1], A[0][2] ...*/
for (i=0; i<= 127; i++)
    for (j=0; j<= 127; j++)
        A[i][j] = 0;
```

- if page is 128 words, then above has 128 page faults

```
for (j=0; j<= 127; j++)
    for (i=0; i<= 127; i++)
        A[i][j] = 0;
```

- now there will be $128 \times 128 = 16,384$ faults

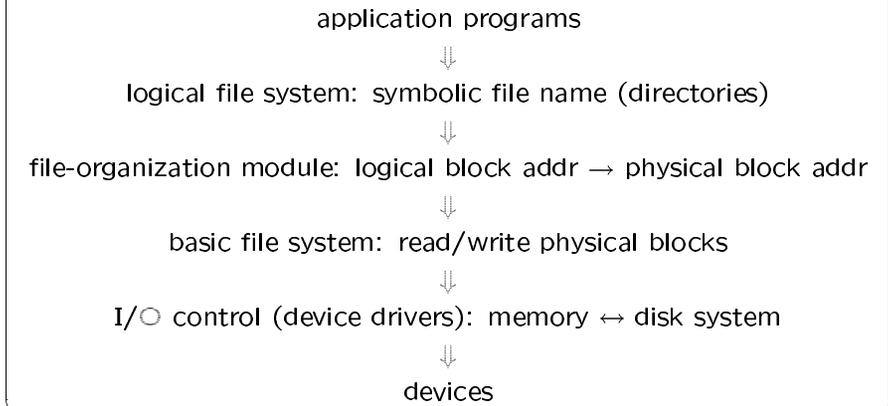
[VI] File Management: Introduction [204]

- files are **logical** units mapped onto **physical secondary** storage
 - file **name**: logical object
 - physical objects: **blocks** on disk, tape, optical disk
 - one or more **sectors**:
 - smallest unit to read from or write to disk
 - block: unit of I/O transfer from disk to memory
 - improves efficiency
 - secondary storage: **nonvolatile**
- file **attributes**: type, location, size, protection, time, date, user ID
 - **volatility**: frequency of additions and deletions
 - **activity**: percentage of records accessed during time frame
- **directories**: keep track of files (and are files themselves)
 - create the **illusion** of compartments
 - but are **indexes** to files which may be scattered
 - entry per file: name, attributes, disk address, etc.

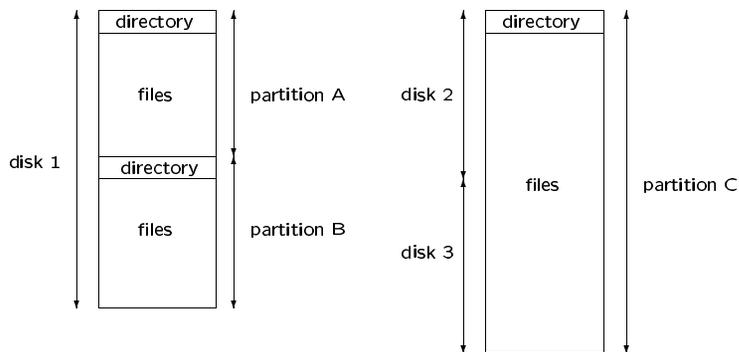
[VI] Files: Abstract Data Type [205]

- data (the file) and the **operations** on the file
- **create**: **allocate** storage, add to **directory**
- **open**: search directory
 - establish **logical** pointer *current-file-position*
- **write**: data at the pointer
- **read**: data at the pointer
- **reposition**: the pointer (*seek*)
- **delete**: file entry in directory
- **truncate**: update **length** attribute in directory
- **append**: new data and update length attribute in directory
- **rename**: change name in directory
- **close**: disconnect logical access to file
- access methods:
 - **sequential**: only increment pointer
 - **direct**: set pointer for seek
 - **indexed**: index of keys and associated direct pointers

[VI] Files: Layered Systems [206]

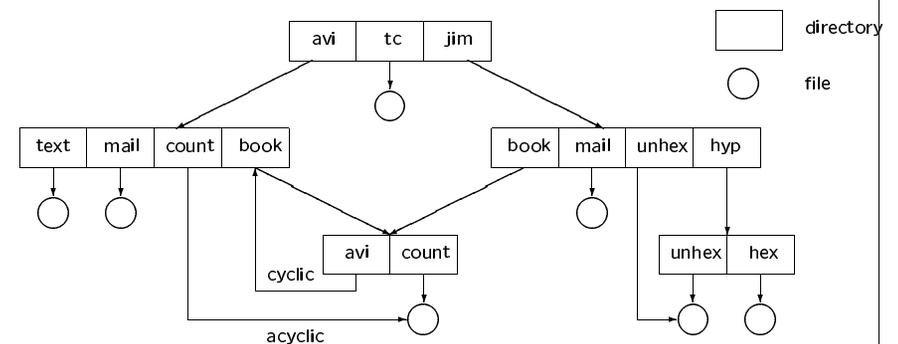


[VI] Files: Partitions - Directories on Disk [207]



- partitions = minidisks = volumes
- virtual disks

[VI] Files: General Graph Directory [208]

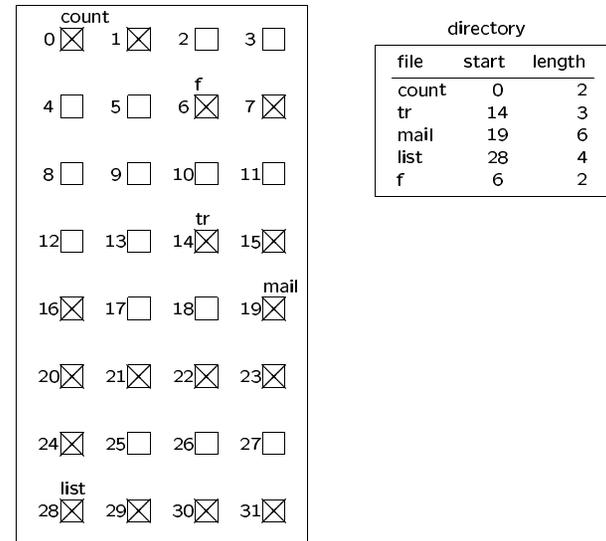


- single-level: one directory shared by all users
- two-level: one directory per user
- tree-structured: long **pathnames**
- graph-structured: sharing

[VI] Files: Contiguous Allocation [209]

- simplest method is a **contiguous** set of blocks
- disk address of start (**base**) block and length (in blocks)
- **sequential** access is easy - read next block
- **direct** access ("seek") is easy -
 - **logical offset = physical base + offset**
- *how to find space for new file?*
 - **first-fit**: first hole that is big enough
 - **best-fit**: smallest hole that is big enough
- **external fragmentation**
 - blocks are available but not sequentially
- **internal fragmentation**
 - preallocation of blocks is too large
 - left-over amount in last block

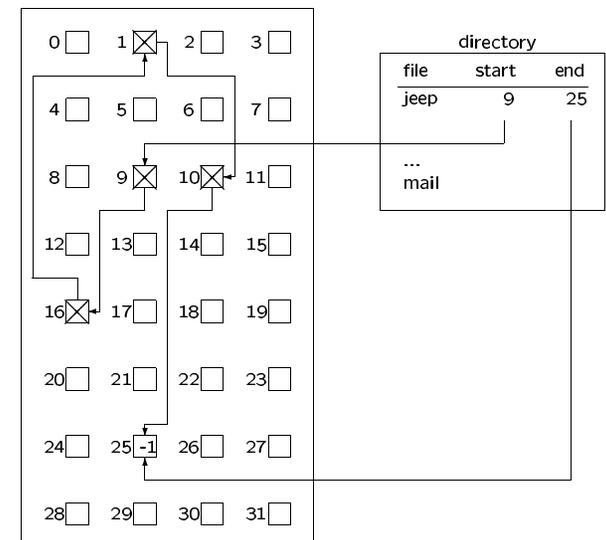
[VI] Files: Contiguous Allocation [210]



[VI] Files: Linked Allocation [211]

- solves external fragmentation: can use any block for any file
- solves "preallocation" internal fragmentation
- good for **sequential** access: chase the pointer
- not effective for **direct** access
 - *where is the nth block of the file?*
- requires space for pointers
- if a pointer is bad, file is lost

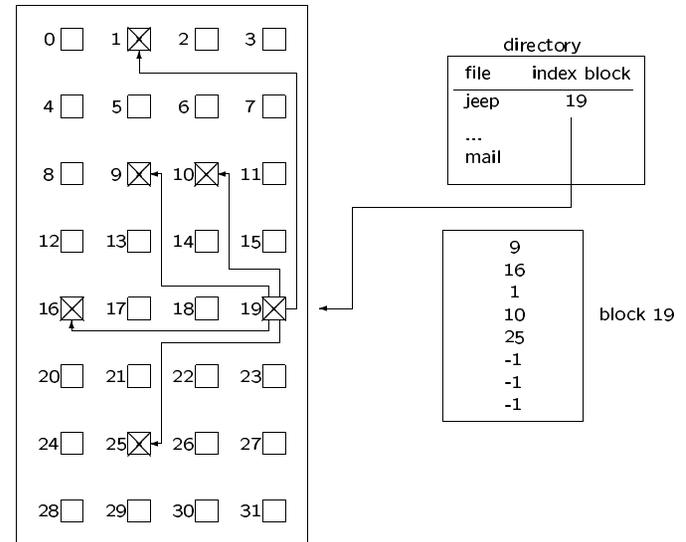
[VI] Files: Linked Allocation [212]



[VI] Files: Indexed Allocation [213]

- contiguous: easy sequential and direct access but fragmentation
- linked: no fragmentation but difficult direct access
- indexed: direct access and no fragmentation
 - bring all pointers into one location: the index block (IB)*
- each file has its own IB
- *i*-th entry in the IB points to the *i*-th block
- suffers from wasted space (IB may not be full)

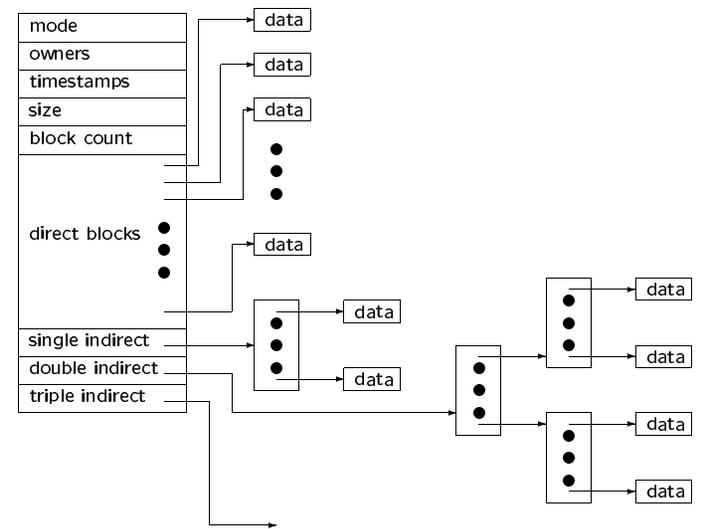
[VI] Files: Indexed Allocation [214]



[VI] Files: How Large an IB? [215]

- each file must have an IB - *IB should be small*
- some files are small - *IB should be small*
- some files are big - *IB should be large*
- solution: multiple levels of smaller IBs
- some files are small - *don't need multiple levels*
- solution: UNIX Index Block
 - 1 direct block of pointers to data (good for small files)
 - indirect blocks (**i-nodes**) for multiple levels

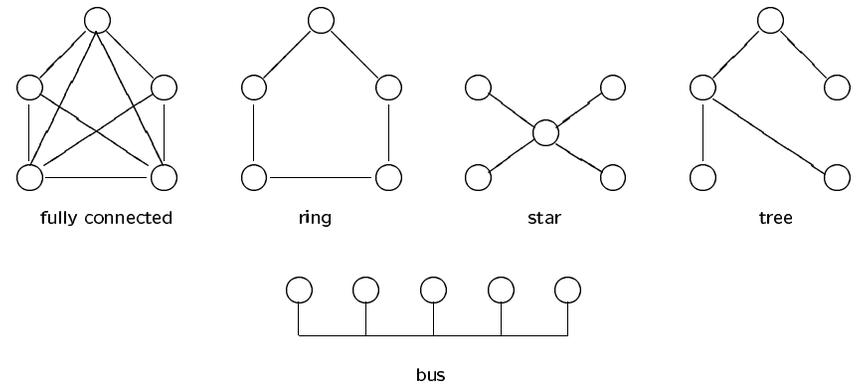
[VI] Files: UNIX inode [216]



[VII.A] DS: Advantages/Disadvantages [221]

- advantages over centralized systems
 - economics, speed, inherent distribution, reliability, incremental
- advantages over isolated (personal) computers
 - data sharing, device sharing, communication, flexibility
- disadvantages
 - little software, network saturation, security

[VII.A] DS: Topology [222]



- **basic cost:** how expensive to link various sites in system?
- **communication cost:** how long does it take to send a message?
- **reliability:** if site fails, can remaining sites still communicate?

[VII.A] DS: LANs vs. WANs [223]

- **Local-Area Networks (LANs):**
 - small geographical area
 - multiaccess bus (Ethernet), ring, star
 - cables: 1 megabyte/sec
 - optical networks: 1 gigabit/sec
 - one or more gateways to other networks
- **Wide-Area Networks (WANs):**
 - large geographical area (Internet)
 - relatively slow (1200 bits/sec to 1 megabyte/sec)
 - unreliable
 - telephone lines, microwave links, and satellite channels
 - communication processors
 - routers

[VII.A] DS: Naming [224]

- how does a process locate another host?
- *domain name service (DNS)*
- <host name, identifier>
- bob.cs.brown.edu
 - request to name server edu for address of server for brown.edu
edu must be known address
 - request to edu.brown for address for cs.brown.edu
 - request to cs.edu.brown for address for bob.cs.brown.edu
⇒ *Internet address* 128.148.31.100
- relies on caches for better performance
- name server is in "wait" state:

```
daemon name_server() {  
    while (1) {  
        receive(&name,&pid);  
        send(pid,lookup(name));  
    }  
}
```

[VII.A] DS: Circuit Switching [225]

- how do two processes communicate with each other?
- session s is allocated transmission rate r_s bits/sec
- fixed path established between two sites
 - each link guarantees portion r_s (say by time multiplexing)
 - if path cannot be found, then reject session
- like telephone switching
 - which has equal session transmission rates
- very inefficient for data networks
 - short bursts of high activity
 - lengthy inactive periods

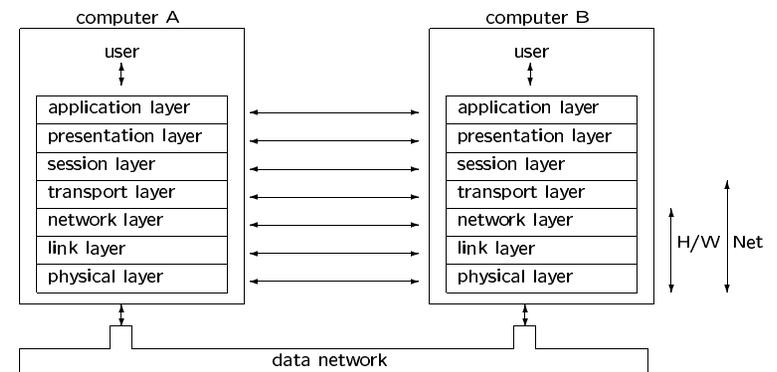
[VII.A] DS: Store-and-Forward Switching [226]

- each session is initiated without reserved allocation
- no multiplexing of links - use full transmission rate
 - links are fully utilized when there is traffic
- packets are queued - waiting for transmission
- decreases delay relative to circuit switching
 - needs (feedback) control mechanism to reduce queuing delays
- **message switching:**
 - store-and-forward with arbitrary message sizes
 - must be broken into packets
- **packet switching:** store-and-forward
- **virtual circuit routing:** store-and-forward but on a fixed path
 - uses full transmission rate of links
- **dynamic routing:** store-and-forward but packet finds its own path

[VII.A] DS: Contention [227]

- several sites want to transmit simultaneously (bus/ring)
- **CSMA:** carrier sense with multiple access (bus)
 - site must listen for a free link
 - if a collision occurs, try again after a random amount of time
 - good for lightly-loaded systems
- **token passing:** a token continually circulates around a ring
 - site can send messages only when it possesses the token
 - uniform performance
- **message slots:** fixed-length message slots circulate in the ring
 - each slot can hold a fixed-sized message and control
 - site must wait until an empty slot arrives
 - sites must check control info for possible messages

[VII.A] DS: ISO Network Model [228]



- use (7) layered approach to deal with complexity
- each layer "talks" with corresponding layer on another computer

[VII.A] DS: ISO Layers

[229]

- **physical**: mechanical and electrical transmission of a bit stream
- **data-link**: transmission of frames or packets
- **network**: routing of packets
- **transport**: transmission of messages as packets, maintaining order
- **session**: implement sessions and protocols (say rlogin)
- **presentation**: resolve differences in format
- **application**: interact directly with the user

[VII.A] DS: TCP/IP

[230]

- TCP/IP: Transmission Control Protocol/Internet Protocol
 - fewer levels, more difficult, more efficient
- IP: transmission of **datagrams**, basic unit of information
- TCP: uses IP to transport a reliable stream between two processes
 - establish and maintain a *connection*
- UDP: User Datagram Protocol
 - uses IP to transfer packets, but adds error correction
 - *connection-less*

[VII.A] DS: Distributed OS

[231]

- users are unaware of the underlying structure or operations
- access remote resources in the same manner as local resources
- **data migration**: data is transferred to sites which require access
 - transfer entire file or only those portions necessary
- **partial computation migration**:
 - synchronous: remote procedure call (RPC) using a datagram
 - asynchronous:
 - message to start new process for designated task
 - both return results
- **process migration** (full computation migration):
 - load balancing to even the workload
 - computation speedup via concurrency
 - hardware or software preference
 - data access

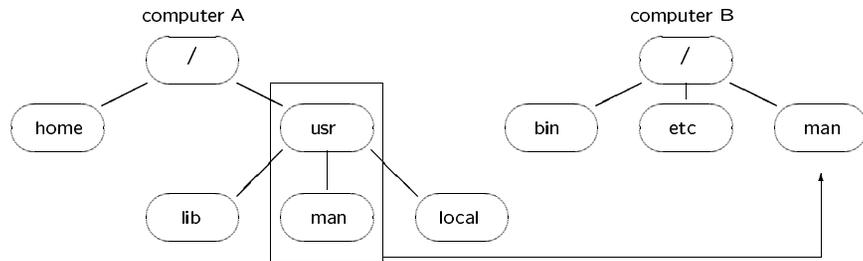
[VII.A] DS: Remote Procedure Call (RPC) [232]

```
result = multiply(7,2);                                     /*client*/
int multiply (x,y) {                                       /*stub */
    sprintf(str1,"%d %d",x,y);                             /*pack */
    general_transport(name_server("multiply"),str1,&str2);
    sscanf(str2,"%d",&result);                             /*unpack*/
    return(result);
}

void general_transport (server,str1,str2) {
    send(server,str1);
    receive(&str2,&server);                                /*block */
}
/***** NETWORK *****/
daemon multiply_transport () {
    while (1) {
        receive(&str1,&client);                            /*block */
        multiply_server_stub(str1,&str2);
        send(client,str2);
    }
}

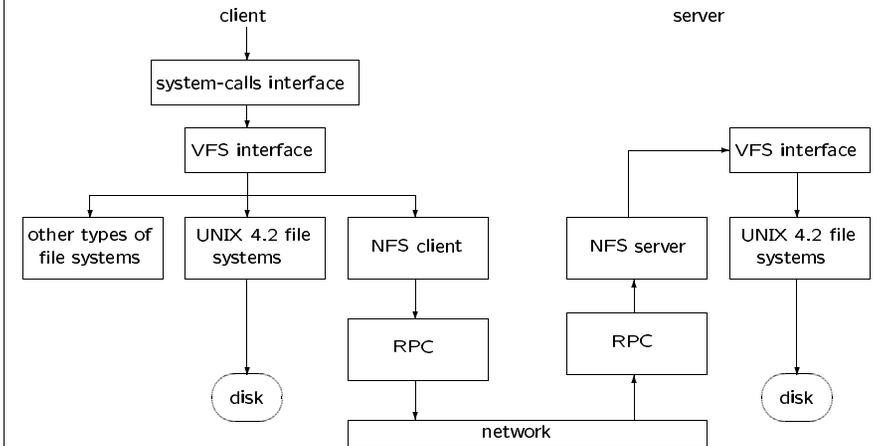
void multiply_server_stub (str1,str2) {                    /*stub */
    sscanf(str1,"%d %d",&x,&y);                             /*unpack*/
    result = multiply(x,y);
    sprintf(&str2,"%d",result);                            /*pack */
}
int multiply (x,y) { return(x*y); }                       /*server*/
```

[VII.A] DS: Network Files System (NFS) [233]



- mount a remote directory
- tables of all mounts `/etc/mntab` also kept in kernel memory

[VII.A] DS: NFS - (Client and Server) [234]



- inode \rightarrow vnode

[VII.B] Distributed Coordination: Introduction [235]

- Mutual Exclusion: *share a critical section*
remember semaphores!
- Resource Allocation: *share a resource*
remember Dining Philosophers!
- Reading/Writing Registers: *share a variable*
- Leader Election: *agree on a leader*
what if a token gets lost? need a root for a tree?
systems reboot?
- Common Knowledge: *everyone knows that everyone knows ...*
when can I be sure?
- Consensus: *everyone agrees on a value*
what if processors lie (faulty)?
- Distributed Minimum-Weight Spanning Tree:
find tree concurrently with distributed information
- see VISUAL OS (Distributed Algorithms)

[VII.B] LeLann's Mutual Exclusion [236]

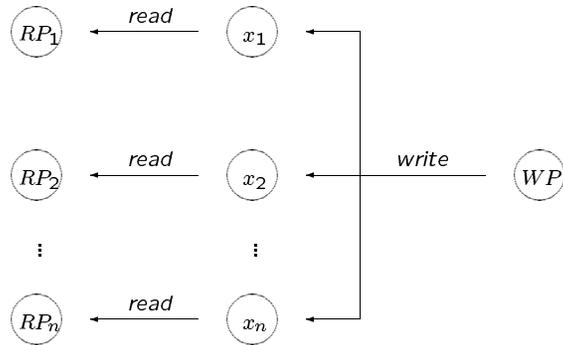
- use `TOKEN` to guarantee mutual exclusion on ring network
- to stop processes after *cycles* around the ring:

```

At all i:
do forever {
    Extra code at i = 1:
    send(neighbor, TOKEN)

    cycles ← cycles - 1
    if cycles = 0 then send(neighbor,
STOP)
    msg = receive()
    /* critical section */
    send(neighbor,msg)
    if msg = STOP then goto EXIT
}
EXIT: destroy(i)
    
```

[VII.B] Lamport's Logical Register [237]



- one logical "shared" register by distributed local registers
- *RP*: reading process
- *WP*: writing process
- logical register is only "regular" (not necessarily consistent)

[VII.B] Lamport's Logical Register [238]

At register process $x_i \in \{1, \dots, n\}$:

```

do forever
  message ← receive()
  if message = READ then send(reader, x)
  else { /* message is the value to be written */
    x ← message
    send(writer, DONE)
  }

```

At reading process $j \in \{n + 1, \dots, 2n\}$:

```

do forever
  send(register, READ)
  value ← receive()

```

At writing process $k = 2n + 1$:

```

do forever
  /* compute new value */
  for i = 1, ..., n do send(i, value)
  for i = 1, ..., n do message ← receive()

```

[VII.B] Coordination: Leader Election [239]

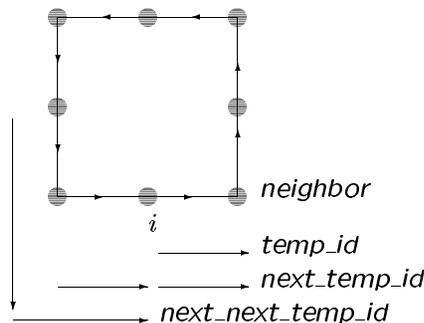
Given: n processes on a ring, each with unique identifier

Problem: elect a leader

say to regenerate a *token*

or to be a root in a spanning tree

all processes must agree and know who the leader is



[VII.B] LeLann's Leader Election [240]

• if received ID is larger, send it to the next node

• if received ID is smaller, ignore it

• if received ID is equal, process is the leader

because its ID traveled all the way around

Which process is elected leader? How should it be announced?

Local Variables:

1. $temp_id \leftarrow i$, the temporary ID of the process
2. $next_temp_id$, the temporary ID of the next clockwise process
3. $neighbor \leftarrow i + 1$, the next counterclockwise process on the ring

At $p_i \in \{1, \dots, n\}$:

send(*neighbor*, *temp_id*)

do forever

next_temp_id ← receive()

if *next_temp_id* > *temp_id* **then** send(*neighbor*, *next_temp_id*)

if *next_temp_id* = *temp_id* **then** announce(LEADER)

[VII.B] Peterson's Leader Election [241]

- process sends its own temp ID to next counterclockwise neighbor
- process receives, sends, temp ID of next clockwise neighbor
- process compares the three IDs
- if clockwise temp ID is largest, it replaces the temp ID
- if clockwise temp ID is not largest, process goes to relay mode
- if clockwise temp ID = own temp ID, receiver is leader

ACTIVE:

do forever

 send(*neighbor*, *temp_id*)

next_temp_id ← receive()

if *next_temp_id* = *temp_id* **then**

 announce(*LEADER*)

 send(*neighbor*, *next_temp_id*)

next_next_temp_id ← receive()

if *next_temp_id* > max(*temp_id*,
 next_next_temp_id)

then *temp_id* ← *next_temp_id*

else goto *RELAY*

RELAY:

do forever

temp_id ← receive()

 send(*neighbor*, *temp_id*)

[X] Java: Simple [243]

- designed so programmers can learn quickly
- especially if they know C++
- omits rarely used, poorly understood, confusing features of C++
- no operator overloading, multiple inheritance, or header files
- true OO means no struct or union
- **no pointers!!!**
- automatically handles referencing/dereferencing of objects
- automatic garbage collection
- arrays and strings are real objects
- small: basic interpreter = 40Kb
- libraries and threads (microkernel) = 175 Kb

[X] Java: System Design [242]

Java is a

- simple,
- object-oriented,
- distributed,
- interpreted,
- robust,
- secure,
- architecture neutral,
- portable,
- high-performance,
- multithreaded,
- and dynamic language.

[X] Java: Object Oriented [244]

- clean definition of interfaces
- good encapsulation/information hiding
- reusable software
- focus on data and methods that manipulate data
- rather than thinking strictly in terms of procedures
- **class** is a collection of data and methods
- data and methods describe state and behavior of **objects**
- **subclasses** inherit from parent class
- Java comes with extensive set of classes arranged as **packages**
- unlike C++, Java was designed to be OO from the ground up
- simple numeric, character, and boolean types are only exceptions

[X] Java: Distributed**[245]**

- extensive library for easy TCP/IP protocols like HTTP and FTP
- can access objects across the net via URL's
(Uniform Resource Locator)
- as easy to open remote file as it is a local file
- reliable stream **Socket** class for client-server model
- can also do unreliable **Datagrams** messages

[X] Java: Interpreted**[246]**

- compiler produces byte-codes rather than native machine code
- `javac myobj.java` produces `myobj.class`
- run the interpreter on any class with a "main": `java myobj`
- interpreter and run-time system: **Java Virtual Machine**
- run byte-codes on any machine that supports this virtual machine
- no "link" phase: classes are loaded dynamically (incrementally)
- compile-time info is stored in byte-codes for checks at load time

[X] Java: Robust**[247]**

- strongly typed
- emphasis on early checking for possible problems
- eliminate situations that are error prone
- C and C++ are slack about procedure declarations
- Java compiler can catch method invocation errors
- interpreter verifies array and string boundaries
- don't worry about corrupting memory
- automatic garbage collection prevents memory leaks
- exceptions make for easier error handling
- later dynamic (runtime) checking

[X] Java: Secure**[248]**

- Java is meant to be used in network environments
- protection against viruses
- authentication techniques are based on public-key encryption
- security related to robustness
- Java's memory allocation is main defense against malicious code
- all memory references are by symbolic handles
- delayed memory allocation/layout decisions ⇒
sources don't provide hackers with key information
- byte-code verification
- loaded classes in separate name space than local classes
- prevents malicious applet from "spoofing" a built-in class
- can disallow network access or access to specific hosts

[X] Java: Architecture Neutral [249]

- byte-codes can run on any machine that supports the Java runtime
- developers can create just one version of their software

[X] Java: Portable [249]

- byte-codes can run on any machine that supports the Java runtime
- no implementation-dependent aspects of language specification
- int, float are the same everywhere

[X] Java: High Performance [249]

- 20 times slower than C
- but fast for an interpreted language
- fine for interactive GUI applications
- byte-codes can be translated on the fly to native code
- byte-codes designed for "just in time" compilers
- runs nearly as fast as C or C++

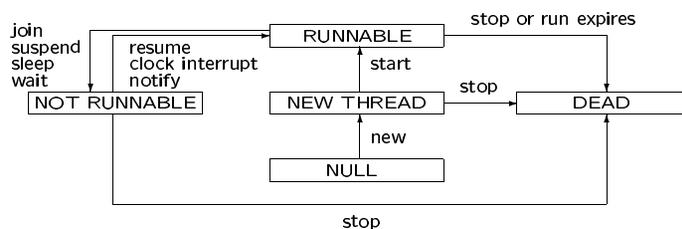
[X] Java: Multithreaded [250]

- multiple things going on in a GUI-based network application
- Thread class makes it easy to start/stop lightweight processes
- synchronized methods allow in only one thread at a time
- stand-alone Java runtime environment has good performance
- running on Unix, Windows, Mac is limited to underlying system

[X] Java: Dynamic [250]

- classes are loaded at runtime even from across the network
- consider a program that is handed an object
 - C or C++: cannot find out what class it belongs to
 - Java: run-time can provide this information
- can trust a "cast" in Java, not in C or C++

[X] Java: State Transitions [251]



- RUNNABLE ⇒ in ready queue OR on the CPU
- NOT RUNNABLE ⇒ suspended state (wait for notify)
- yield, higher priority, time slice ⇒ stay RUNNABLE

[X] Java: Inheritance Example [252]

```
import java.awt.Color; // package
public abstract class Primitive { // abstract=>cannot make an object instance

    public static final int X=100; // class constant
    public static final int Y=100;
    private int x; // instance variable
    private int y;
    private String name; // String is class
    protected Color color; // let subclasses have access

    public Primitive(String name, int x, int y, Color color) { // constructor
        this.name = name; // refer to self
        this.x = x;
        this.y = y;
        this.color = color;
    }

    public void setx(int x) {this.x = x; } // instance method
    public void sety(int y) { this.y = y; }
    public int getx() { return x; }
    public int gety() { return y; }
    public String getName() { return name; }
    public Color getColor() { return color; }
    public void chgAppearance() { color = color.brighter(); }
}
```

[X] Java: Inheritance Example**[253]**

```
import java.awt.Color;
public class Conic extends Primitive { // inheritance

    private int a; // ax^2 + bxy + cy^2 + dx + ey + f = 0
    private int b;
    private int c;
    private int d;
    private int e;
    private int f;

    public Conic(String name, int x, int y, Color color,
                 int a, int b, int c, int d, int e, int f) {
        super(name,x,y,color); // call constructor of parent
        this.a = a;
        this.b = b;
        this.c = c;
        this.d = d;
        this.e = e;
        this.f = f;
    }
}
```

[X] Java: Inheritance Example**[254]**

```
public class Circle extends Conic { // inheritance
    public static final double PI=3.14159; // class constant
    public static int num_circles=0; // class variable
    public int r; // instance variable
    public Circle(String name, int x, int y, Color color, int r) {
        super(name,x,y,color,1,0,1,0,0,-r*r); // call constructor of parent
        num_circles++; // count number of objects
        this.r = r;
    }
    public Circle(String name, int r) { // multiple constructors
        this(name,Primitive.X,Primitive.Y,Color.red,r);
    }
    public Circle(int r) { this("",r); }
    public double circumference() { return 2*PI*r; }
    public double area() { return PI*r*r; }
    public Circle bigger(Circle c) { // instance method
        if (c.r > this.r) return c; else return this; // return myself
    }
    public static Circle bigger(Circle a, Circle b) { // class method
        if (a.r > b.r) return a; else return b;
    }
    public void chgAppearance() { // overriding method
        color = color.darker(); // access protected variable
    }
}
```

[X] Java: Inheritance Example**[255]**

```
class Geo {
    public static void main(String args[]) { // entry point
        Circle a = new Circle("a",10); // object instance of class
        Circle b = new Circle("b",20); // another object
        Color color;
        // access class variable
        System.out.println("Number of circles: "+Circle.num_circles);
        // access instance method
        System.out.println("Circ of a: "+a.circumference()+" Area of b: "+b.area());
        // access public instance variable
        System.out.println("Radius of a: "+a.r);
        // access instance method of grandparent
        System.out.println("Location of a: "+a.getx()+" "+a.gety());
        // access instance method
        System.out.println("Biggest area: "+a.bigger(b).area());
        // access class method
        System.out.println("Biggest area: "+Circle.bigger(a,b).area());
        // access instance method of grandparent
        System.out.println("Circle with biggest area: "+a.bigger(b).getName());
        color = a.getColor();
        System.out.println("Color of circle a: "+color.toString());
        a.chgAppearance(); // make it darker
        color = a.getColor();
        System.out.println("Color of circle a: "+color.toString());
    } }
}
```

[X] Java: Philosopher Demo**[256]**

```
class PhilDemo {
    public static void main(String args[]) {
        String s = new String("A");
        Integer fork1 = new Integer(1); Integer fork2 = new Integer(2);
        if (args.length > 0) s = args[0];
        if (s.equals("A")) {
            new PhilA("Philosopher One",4).start();
            new PhilA("Philosopher Two",4).start();
        } else
        if (s.equals("B")) {
            new PhilB("Philosopher One",4,fork1,fork2).start();
            new PhilB("Philosopher Two",4,fork1,fork2).start();
        } else
        if (s.equals("C")) {
            PhilC phil1 = new PhilC("Philosopher One",4);
            PhilC phil2 = new PhilC("Philosopher Two",4);
            phil1.setOther(phil2); phil2.setOther(phil1);
            phil1.start(); phil2.start();
            Thread me = Thread.currentThread(); me.yield();
            synchronized(phil1)phil1.notify();
        } else
        if (s.equals("D")) {
            new PhilD("Philosopher One",4,6117,6118).start();
            new PhilD("Philosopher Two",4,6118,6117).start();
        } }
}
```

[X] Java: Philosopher Sync Method [257]

```

class PhilA extends Thread {           // inheritance
    private int steps;
    public PhilA (String name, int steps) { // constructor
        super(name);                   // let Thread class save the name
        this.steps = steps;
    }
    public void run() {                 // start executes the run method
        while (steps > 0) {
            eat();
            steps--;
        }
    }
    static synchronized void eat() {    // entire method is critical section
        print();
    }
    public static void print() {        // static=>can use without an object
        Thread me = Thread.currentThread();
        System.out.println(me.getName()+" start eating");
        me.yield();                    // try to give up the processor
        System.out.println(me.getName()+" stop eating");
        me.yield();
    }
}

```

[X] Java: Philosopher Sync Block [258]

```

class PhilB extends Thread {
    private int steps;
    private Integer fork1;              // any object will do
    private Integer fork2;
    public PhilB (String name, int steps, Integer fork1, Integer fork2) {
        super(name);
        this.steps = steps;
        this.fork1 = fork1;
        this.fork2 = fork2;
    }
    public void run() {
        while (steps > 0) {
            eat();
            steps--;
        }
    }
    void eat() {
        synchronized(fork1) {          // critical section based on object
            synchronized(fork2) {
                PhilA.print();
            }
        }
    }
}

```

[X] Java: Philosopher Wait/Notify [259]

```

class PhilC extends Thread {
    private int steps;
    Thread other;
    public PhilC (String name, int steps) {
        super(name);
        this.steps = steps;
    }
    public void setOther(Thread other) { // remember the other philosopher
        this.other = other;
    }
    public void run() {
        while (steps > 0) {
            if (getName().equals("Philosopher One"))
                eat1();
            else
                eat2();                // make 2 versions, both synchronized
            steps--;
        }
    }
    synchronized void eat1() { // must be synchronized to use wait
        try this.wait(); catch (InterruptedException e) System.err.println(e);
        PhilA.print();
        synchronized(other)other.notify();
    }
}

```

[X] Java: Philosopher Send/Receive [260]

```

import java.net.*;
class PhilD extends Thread {
    int steps;
    byte[] inbuf = new byte[1];        byte[] outbuf = new byte[1];
    DatagramPacket inpacket;           DatagramPacket outpacket;
    DatagramSocket mysocket;           DatagramSocket yoursocket;
    public PhilD (String name, int steps, int myport, int yourport) {
        super(name); this.steps = steps;
        try { mysocket=new DatagramSocket(myport); yoursocket=new DatagramSocket();
              inpacket=new DatagramPacket(inbuf,inbuf.length);
              outpacket=new DatagramPacket(outbuf,outbuf.length,
                                             InetAddress.getLocalHost(),yourport);
        } catch (Exception e) System.err.println(e);
    }
    public void run() {
        if (getName().equals("Philosopher Two"))
            try yoursocket.send(outpacket); catch (Exception e);
        while (steps > 0) { eat(); steps--; }
    }
    void eat() {
        try { mysocket.receive(inpacket);
              PhilA.print();
              yoursocket.send(outpacket);
        } catch (Exception e) System.err.println(e);
    }
}

```

[X] Java: Exceptions**[261]**

```

class MyException extends Exception {
    public MyException() { super(); }
    public MyException(String s) { super(s); }
}
class MyOtherException extends Exception {
    public MyOtherException() { super(); }
    public MyOtherException(String s) { super(s); }
}
class MySubException extends MyException {
    public MySubException() { super(); }
    public MySubException(String s) { super(s); }
}
public class throwtest {
    public static void main(String argv[]) { int i;
        try { i = Integer.parseInt(argv[0]); }
        catch (ArrayIndexOutOfBoundsException e) {
            System.out.println("Must specify an argument");
            return;
        }
        catch (NumberFormatException e) {
            System.out.println("Must specify an integer argument");
            return;
        }
        a(i);
    } }

```

[X] Java: Exceptions**[262]**

```

public static void a(int i){
    try b(i);
    catch (MyException e) {
        if (e instanceof MySubException)
            System.out.print("MySubException: ");
        else
            System.out.print("MyException: ");
        System.out.println(e.getMessage());
        System.out.println("Handled at point 1");
    }
}
public static void b(int i) throws MyException {
    int result;
    try {
        System.out.print("i="+i+" ");
        result = c(i);
        System.out.println("c(i)="+result);
    }
    catch (MyOtherException e) {
        System.out.println("MyOtherException: "+e.getMessage());
        System.out.println("Handled at point 2");
    }
}

```

[X] Java: Exceptions**[263]**

```

public static int c(int i) throws MyException, MyOtherException {
    switch (i) {
        case 0 : // processing resumes at point 1 above
            throw new MyException("input too low");
        case 1 : // processing resumes at point 1 above
            throw new MySubException("input still too low");
        case 99: // input resumes at point 2 above
            throw new MyOtherException("input too high");
        default: return i*i;
    }
}

```

[XI] XINU: System Design**[264]**

Layers:

- user programs
- file system
- intermachine communication
 - device manager and device drivers
 - real-time clock manager
 - interprocess communication
 - process coordination
 - process manager
 - memory manager
 - hardware

Written in C and runs on PC, Mac, Sun, LSI-11, etc.

[XI] XINU vs. VOS

[265]

- VOS based on XINU
- VOS has all the same states plus READING and WRITING
- VOS checks for sleeping processes with each reschedule
- VOS has almost all of the same system calls
- VOS has similar process table
- VOS has the same queues
- VOS has no memory management, clock, device drivers, file system
- VOS has no interrupts
- VOS has only one process eligible to run at a time
- VOS runs on top of UNIX
- XINU "is" the operating system
- XINU is PRIO scheduling but VOS also has SJF
- VOS has a GUI
 - display state transitions
 - provide complete environment for running demo code

[XI] XINU: Initialization

[266]

Initialization is the final step in design

- design the "steady" state first
- then design the "transient"
- typical of "function-oriented" systems
- but "object-oriented" systems have **constructors**
- initialization is easier because of hidden data

[XI] XINU: Processes

[267]

- Processes are referenced by their process id
- pid acts as an index of the saved state information in proctab
- highest priority process eligible for CPU service is executing
- among processes with equal priority, scheduling is round-robin
- current process does not appear on the ready list, but as **currp**id
- resched can only switch context from one process to another
- Null process just continues to call resched
- states: current, ready, receiving, sleeping, suspended, waiting

[XI] XINU: System Calls

[268]

- processes
create, getpid, kill, resume, sleep, suspend, chprio, getprio
- messages
receive, send
- ports
pcount, pcreate, pdelete, preceive, preset, psend
- semaphores
scount, screate, sdelete, signal, sreset, wait
- memory
getmem, getstk
- devices
close, control, getc, getdev, init, open, putc, read, seek, write

[XI] XINU: Memory

[269]

- **getmem** obtains memory from the heap
 - finds the first block large enough for request
 - create allocates a stack for a process
 - user programs can also use the heap
- **freemem** returns memory to the heap
 - blocks on the free list are ordered by increasing address
 - scan and find the proper location
 - adjacent free blocks are grouped into a larger block
 - user programs must return memory to the heap

[XI] XINU: Interrupts

[270]

- interrupts generated by clock, device controllers
- hardware calls interrupt handler when it finds an interrupt pending
- handler uses assembly language “dispatcher”
- saves/restores registers
- indexes into the interrupt vector to get specific high-level routine
- interrupts disabled when dispatcher calls high-level routine
- high-level routine must keep disabled until changes complete
- system calls, like resume, “disable” and then “restore” interrupts
- do not want other processes changing process table, etc.
- but disable time must be short so devices are OK

[XI] XINU: Interrupts

[271]

- process P is running when an interrupt occurs
- hardware uses P’s stack to save registers
- P continues to run the interrupt dispatcher
- interrupts disabled by dispatcher
- high-level routine may resched process Q
- Q might pick up from the end of a system call: enable
- new interrupt comes in but goes onto Q’s stack
- when P runs again, the context switch will turn off interrupts
- only one interrupt per process is stacked
- rescheduling during interrupt processing is safe provided
 - routines leave global data in valid state before rescheduling
 - no procedure enables interrupts unless it disabled them

[XI] XINU: Real-Time Clock

[272]

- time-of-day clock: pulses and counts the pulses
- real-time clock: pulses and generates interrupts
- CPU reads the time-of-day clock if it wants current date/time
- real-time clock forces CPU to process an interrupt with each pulse
- hardware gives highest priority to clock interrupts
- used for **preemption** to prevent infinite loops
- used for **round robin** scheduling among equal priority processes
 - resched sets “preempt” to QUANTUM
 - QUANTUM is the “granularity of preemption”
 - clock interrupt routine decrements “preempt”
 - if zero, call resched

[XI] XINU: Sleeping Processes [273]

- real-time clock used for **timed delay** for sleeping processes
 - cannot afford to search through long list of sleeping processes
 - all processes kept on a **delta list**
 - the first process is the one with the least delay
 - all other processes have deltas based on the preceding process
 - clock just decrements the first process until its zero
 - new sleepers inserted at proper place with proper delta

[XI] XINU: Device I/O [274]

- hide messy details in **device drivers**
- access must be fair and safe to shared devices
- provide uniform interface to all devices
- **asynchronous I/O** allows processes to continue
 - overlap computation and I/O
- **synchronous I/O** blocks processes until I/O completed
 - easier and works in most cases
- application calls **high-level routine** like **putc** with device descriptor

```
putc(int descrp, char ch)
```
- high-level routine indexes into **device switch table** using descriptor

```
devptr = &devtab[descrp];
```
- device table gives real device address and driver routine
- high-level routine calls the **upper-half device driver**

```
return( (*devptr->dvputc)(devptr, ch) );
```

[XI] Upper-Half TTY Output Device Driver [275]

- output function acts as a **producer** of chars

```
ttyputc(devptr, ch)
    struct tty *iptr = &tty[devptr->dvminor];
```
- wait for space in the buffer (waiting for consumer)

```
wait(iptr->osem);
```
- put the character into the buffer

```
iptr->obuf[iptr->ohed++] = ch;
++iptr->ocnt;
if (iptr->ohed >= OBUFLEN) iptr->ohed = 0;
```
- send a message to tty lower-half process (which may be blocked)

```
sendn(iptr->oprocnum, TMSGOK);
```

[XI] Lower-Half TTY Output Device Driver [276]

- output function acts as a **consumer** of chars

```
PROCESS ttyoproc()
```
- infinite loop with a receive

```
for (;;)
    receive();
```
- take the data out of the buffer

```
ch = iptr->obuf[iptr->otail++];
--iptr->ocnt;
if (iptr->otail >= OBUFLEN) iptr->otail = 0;
```
- signal the producer (upper-half) that space is available

```
signal(iptr->osem);
```

[XI] XINU: TTY Output Watermarks [277]

- very popular and fundamental technique
- producer runs faster and/or has higher priority than consumer
- application usually produces many chars at once \Rightarrow buffer full
- with each signal from consumer, producer puts one more char
- forces a reschedule with EVERY character: too slow
- if the buffer fills past the **high watermark**
 - consumer does not signal, just counts
- if the buffer empties below the **low watermark**
 - consumer makes-up for all of the signals
- allows the producer to run awhile before the buffer fills again

[XI] XINU: TTY Input Device Drivers [278]

- upper-half input function acts as a **consumer** of chars
`char ttygetc(devptry)`
- lower-half input function acts as a **producer** of chars
`PROCESS ttyiproc()`

[XI] Upper-Half Disk Output Device Driver [279]

- upper-half output function
`dswrite(devptry, buff, block)`
- enqueues the request and returns
`dskeng(drptry, devptry->dvioblk);`
- enqueue forces disk arm to sweep low to high and back again

When adding a request for block B to the existing list of requests, schedule it to be performed between requests for i and i+1 if the disk arm will pass over block B on its way from i to i+1. If no such pair i and i+1 exist, add the new request to the end of the list.

- FIFO order must be preserved for requests on the **same** block
- if enqueue on empty list, then send message to lower-half:

```
dskeng(drptry, dsptry) {
  if (dsptry->dreqlst == DRNULL) {
    dsptry->dreqlst = drptry;      /* enqueue */
    drptry->drnext = DRNULL;
    sendn(dsptry->dsprocnum);
  } else OPTIMIZE ALL READS, WRITES, SEEKS }
```

[XI] Lower-Half Disk Device Driver [280]

- lower-half process handles all requests
`PROCESS dsinter(dsptry,dsknum) {`
 `for (;;) {`
 `drptry = dsptry->dreqlst;`
 `if (drptry == DRNULL) receive(); /* if empty, receive */`
 `dsptry->dreqlst = drptry->drnext; /* dequeue sequentially */`
 `switch (drptry->drop) {`
 `case DREAD: dread(drptry->drbuff,dsknum,drptry->drdba);`
 `resume(drptry->drpid);`
 `break;`
 `case DWRITE:dwrite(drptry->drbuff,dsknum,drptry->drdba);`
 `case DSYNC, DSEEK, ...`
 `}`
 `}`
`}`

[XI] Disk Input Device Drivers

[281]

- upper-half input function

```
dsread(devptr, buff, block) {
    struct dreq *drptr;
    drptr->drbuff = buff;
    drptr->drdba = block;
    drptr->drop = DREAD;
    drptr->drpid = currpri;
    dskenq(drptr, devptr->dviobl);
    suspend(currpri);
}
```

- lower-half `disinter` function reads and resumes upper-half
`dread(drptr->drbuff, dsknum, drptr->drdba);`
`resume(drptr->drpid);`

[XII] UNIX: System Design

[282]

- no real design before implementation
- first development in 1969 at Bell Labs by Thompson and Ritchie
- Ritchie had worked on MULTICS
- UNIX is a pun on MULTICS
- file system and shell are similar
- implemented in C
- designed by programmers for programmers
- designed to be a time-sharing system
- not much layering (see earlier slide)
- shell programs combine ordinary programs
- pipes for redirection of input/output: `% myprog <indata >outdata`

[XII] UNIX: Processes

[283]

- **process** is a **program** in execution
- new process created by **fork**
- parent can create a child
- child runs the same program as parent
- typically, child will **execve** a new program
- parent **waits** for a child's **exit**
- if a parent exits, then the child is a **zombie**
- fork allows a **pipe** between parent and child (see IPC)
- read from empty pipe or write to full pipe: block
- **signal** handles exceptional conditions (keyboard interrupt)
- process can ignore a signal or have a **signal handler** routine
- signals can be used to start and stop subprocesses on demand

[XII] UNIX: Process Control Block

[284]

- called **process structure**
- process ID, priority, etc.
- array of PCBs defined at system linking time
- ready queue is doubly-linked list through the PCBs
- pointers to parent, youngest living child, other relatives
- normal execution is in **user mode**
- system calls switch to **system mode**
- system mode uses **kernel stack** for that process

[XII] UNIX: Scheduling

[285]

- designed to benefit interactive processes
- small CPU time slices using a priority algorithm
- larger numbers indicate lower priority
- processes doing disk I/O are less than "pzero"
- ordinary user processes have positive priorities
- less likely to run than any system process
- user processes can set precedence over one another using **nice**
- high CPU usage \Rightarrow lower priority (more positive)
- process **aging** prevents starvation
- time-slice every 0.1 sec and recompute priorities every 1 second
- round-robin uses **timeout** which tells clock to interrupt
- reschedule and then set another timeout
- priority recomputation also uses timeout

[XII] UNIX: Events

[286]

- relinquish CPU because of I/O or time slice expired
- or **sleep** waiting for some **event**
- argument is address of kernel data structure for that specific event
- system calls **wakeup** on all sleeping processes for that event
- wait for disk I/O to complete (sleep on address of buffer header)
- **race condition:**
 - process decides to sleep (based on, say, flag)
 - event occurs
 - process calls sleep (but event will never occur)
 - raise hardware processor priority for this critical section
 - no interrupts and process can run until sleeping

[XII] UNIX: Memory Management

[287]

- early system just swapped out processes if not enough memory
- PID0=scheduler process (**swapper**) wakes up every 4 secs
- swap out a process if:
 - idle
 - in main memory a long time
 - large
 - old
- swap in a process if:
 - swapped out a long time
 - small
- some UNIX systems still do this
- Berkeley UNIX uses demand paging and secondarily swapping

[XII] UNIX: Demand Paging

[288]

- **virtual memory**
- swapping kept to minimum because more jobs in memory
- only parts of each process in memory
- list of **free frames**
- modification of **second chance (clock)** algorithm
- memory is swept linearly and repeatedly by software **clock hand**
- if frame is already free or in use (say for I/O), skip
- otherwise, go to page-table entry for frame
- if **invalid**, put frame on free list
- otherwise, mark as invalid but **reclaimable**
- clock hand implemented by **pagedaemon**
- runs only if free frames falls below threshold
- if hardware supports ref bit, then use it
- one pass of the clock turns bits off
- second pass checks bit and puts onto free list

[XII] UNIX: File System

[289]

- **file** is a sequence of bytes
- files organized by **directories**
- file data linked by **inodes** (see earlier slide)
- system calls use a **file descriptor** as an argument
- kernel indexes into **table of open files** for current process
- each entry points to a **file structure**
- each file structure points to an **inode**

[XII] UNIX: I/O System

[290]

- disk files and I/O devices treated as similarly as possible
- general **device driver** code
- specific device driver code for each device
- **block devices**
 - disks and tapes
 - array of entry points for drivers
 - use a block buffer **cache** for block I/O
 - or use a queue of pending transfers for **raw device interfaces**
- **character devices**
 - terminals, line printers, etc.
 - array of entry points for drivers
 - **C-lists** are small blocks of characters
 - **write** enqueues onto the list for the device
 - interrupts cause dequeuing
 - input is also interrupt driven

[XII] UNIX: IPC

[291]

- not one of the strong points of UNIX
- pipe, shared files, messages, shared memory
- sockets
 - **stream**: reliable and sequenced stream
 - **datagram**: unreliable and unsequenced messages
- **socket** creates a socket and returns **descriptor**
- descriptor indexes into array of open "files"
- **bind** assigns a name to a socket
- **client-server model**
 - server creates **socket** and **binds** to well-known address
 - client uses **connect** on well-known address
 - server **listens** to say that it is ready for connections
 - server **accepts** individual connection
 - server usually **forks** a process to talk with client
 - server goes back to listening
 - server subprocess and client do **read** and **write**

[XIII] Mach: History

[292]

- CTSS - **multiprogramming** and **timesharing** (1962)
- MULTICS - **virtual memory** (1965)
 - a *process*
 - large and complex - *all things to all people*
 - command line interpreter
- UNIX - a pun on MULTICS (1969)
 - stripped-down version
 - "C"
- Rochester RIG - **message passing** over a network (1976)
- CMU Unix - **port capabilities** as object references (1979)
- Accent - integration of **memory and IPC** (1985)
- Mach - designed for **multiprocessors** (1989)
 - incorporated recent innovations
 - a few simple/powerful abstractions (and they interoperate)

[XIII] Mach: What about UNIX?

[293]

UNIX good points:

- **multiprogrammed**
- easy portability to wide class of **uniprocessors**
- simple programmer interface to system facilities
- extensive library
- pipes

UNIX bad points:

- not intended for **multiprocessors**
- kernel became repository for redundant/competing abstractions
- for example - IPC:
 - sockets, streams, pipe
 - shared files, system V messages, shared memory

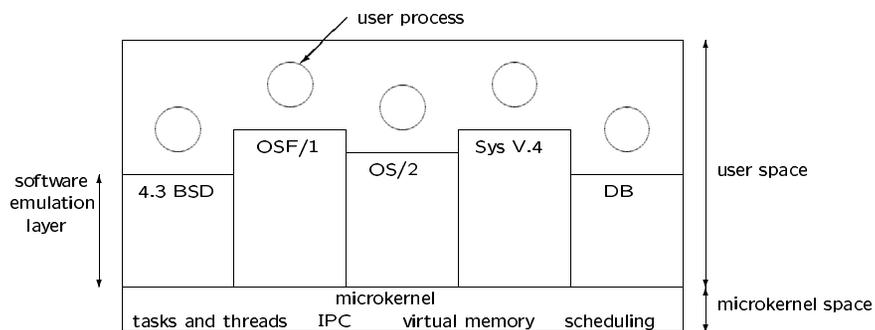
[XIII] Mach: Basic Goals

[294]

- simplicity: 5 powerful user abstractions which interoperate
integrated memory management and IPC
- extensibility:
 - kernel functions may be efficiently exported to user state
 - leaves just a **microkernel**
- compatibility: UNIX programs (binaries) fully supported
- **multiprocessor** systems - even heterogenous
- large and different types of memories
 - Uniform Memory Access (UMA)
 - Non-Uniform Memory Access (NUMA)
 - No Remote Memory Access (NORMA)
- transparent access to different types of networks
 - LANs and WANs
 - tightly coupled multiprocessors

[XIII] Mach: Structure and Emulation

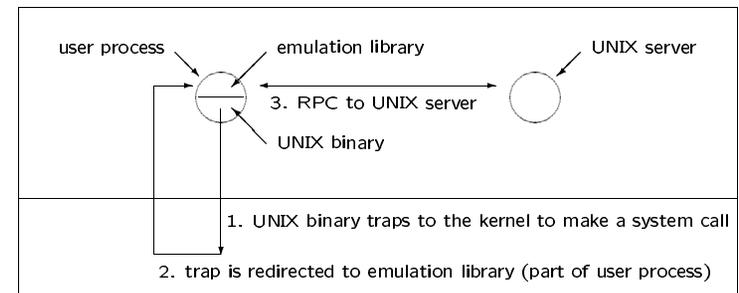
[295]



- many OS run on top of Mach
- BSD provides the user interface/programming environment

[XIII] Mach: UNIX Emulation in Mach

[296]



- Mach kernel acts as a "trampoline"

[XIII] Mach: Primitive Abstractions

[297]

- **task**: execution environment
 - provides virtual address space
 - provides protected access to system resources via **ports**
 - contains one or more **threads**
 - **it is not a process**: computationally passive
- **thread**: unit of computation (execution)
 - must run in the context of a task
 - task provides address space
 - all threads in a task share ports, memory, etc.
 - **process = task + thread**
 - minimal state information \Rightarrow **lightweight process**

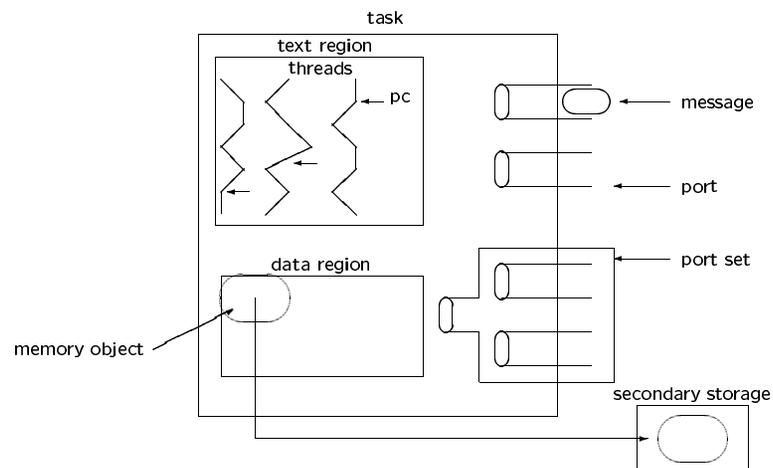
[XIII] Mach: Primitive Abstractions

[298]

- **ports**: communication channel
 - send/receive **messages** on ports
 - kernel maintains **capability** list of **rights** to send/receive
 - **port set** is a group of ports sharing a common message queue
 - thread can receive on a port set - service multiple ports
- **messages**: basic method of communication between threads
 - "typed" (self-describing) data - can be 4GB
 - **in-line** or **out-of-line** (pointer) data
 - port rights are passed in messages (the only way)
- **memory objects**: storage unit
 - tasks access objects by using ports!
 - **map** all or part of object into address space
 - object may be managed by **external memory manager**
 - examples: files, pipe

[XIII] Summary of Primitive Abstractions

[299]



[XIII] Mach: Blend Memory and IPC

[300]

- Memory Management:
 - memory object represented by port
 - IPC messages are sent to port (e.g. pagein, pageout)
 - memory objects may easily reside on remote systems
- IPC:
 - try to pass messages by moving pointers to shared memory
 - try to avoid, or at least delay, copying
 - let virtual memory management do the copying

[XIII] Mach: Process Management - Tasks [301]

- system calls to kernel: messages on **process port**
- **create**: parent task creates children tasks
 - children **inherit** all or selected regions of parent's memory
 - shared or copied
- **priority**: for current or future threads
- **assign**: (set of) processor for new threads
- **suspend**: all threads in task
- **resume**: all threads in task
- **terminate**: all threads in task

[XIII] Mach: Process Management - Threads [302]

- **create**: give function to execute and its parameters
- **suspend**: one thread but not the task
- **resume**: one thread but task may still be suspended
- all threads share the **process port** and other ports
- each thread has its own **thread port** say to **terminate**
- all threads share the address space of the task
 - ⇒ need **synchronization**

[XIII] Mach: Thread Synchronization [303]

- **mutex_lock**(mutex): a **wait** on mutex but with a **spinlock**
- **mutex_unlock**(mutex): a **signal**
- **condition variables**:
 - implement critical sections without busy waiting
- **condition_wait**(condition variable, mutex variable):
 - unlocks mutex variable
 - blocks for a **condition_signal**(condition variable)
- **condition_signal**(condition variable):
 - sets condition variable to true and unblocks (all) waiting threads
 - condition may not hold when wait returns
 - ⇒ need a loop for wait

[XIII] Producer-Consumer Synchronization [304]

```
INITIALIZATION:
int buffer[MAXBUF];    int buf_ptr = -1;
int nonempty = FALSE;  int nonfull = TRUE;
mutex_alloc(mutex,1);  condition_alloc(nonempty,nonfull);

void add_buffer(int item) {
    buf_ptr++;
    buffer[buf_ptr] = item;
    empty = FALSE;
    if (buf_ptr == MAXBUF-1) full = TRUE;
}

int rem_buffer() {
    int item = buffer[buf_ptr];
    buf_ptr--;
    full = FALSE;
    if (buf_ptr == -1) empty=TRUE;
    return(item);
}

PRODUCER:
while (1) {
    nextp = produce_item();
    mutex_lock(mutex);
    while (full)
        condition_wait(nonfull,mutex);
    add_buffer(nextp);
    condition_signal(nonempty);
    mutex_unlock(mutex);
}

CONSUMER:
while (1) {
    mutex_lock(mutex);
    while (empty)
        condition_wait(nonempty,mutex);
    nextc = rem_buffer();
    condition_signal(nonfull);
    mutex_unlock(mutex);
    consume_item(nextc);
}

TERMINATION: mutex_free(mutex); condition_free(nonempty,nonfull);
```

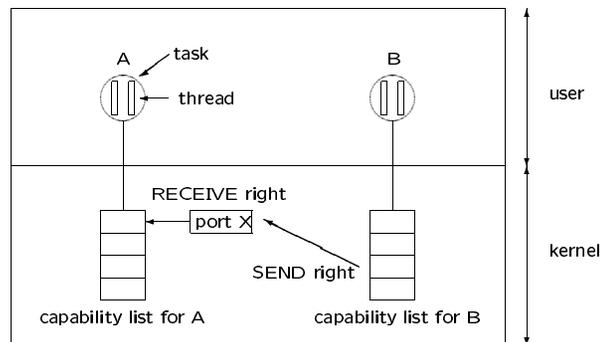
[XIII] Mach: CPU Scheduling [305]

- only threads are scheduled - no knowledge of task is needed
- CPUs and threads assigned to **processor sets** (independently)
 - ⇒ threads that need computing power and CPUs at disposal
- thread has **priority**:
 - **base** priority set by thread (within a **limit**)
 - **current** priority = base priority + $f(\text{recent CPU usage})$
- 32 global run queues for each processor **set**: one for each priority
 - lock the global run queues
 - find the highest priority thread (use hints)
- 1 "highest-priority" local run queue for "CPU" threads: I/O devices
- thread is given one **quantum** to run:
 - check queues again; if empty or low priorities, go again
 - with each tick: give thread a lower priority
- quantum is constant across a processor set
 - quantum increases as CPUs go up - or threads go down

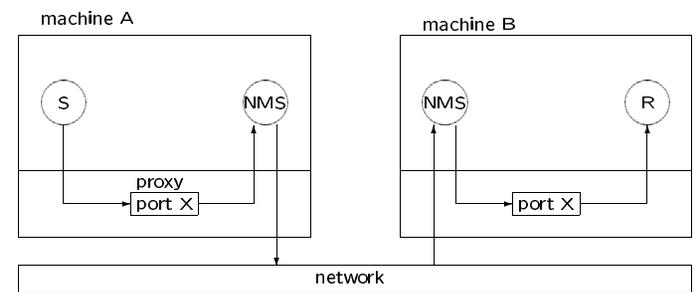
[XIII] Mach: Ports [306]

- bounded queue within the kernel
- **capability**: send or receive "right"
- only one receiver for each port (but must have right)
- multiple senders for each port (but must have right)
- **allocate**: new port (and get the rights)
- creator can give out rights in messages
- if receive right sent in a message, sender loses the right
- task allocates ports to the objects that it owns
- **deallocate**: revocation of all rights
- **port sets**: can only have receive rights

[XIII] Mach: Ports and Capabilities [307]



[XIII] Mach: Network Messages [308]



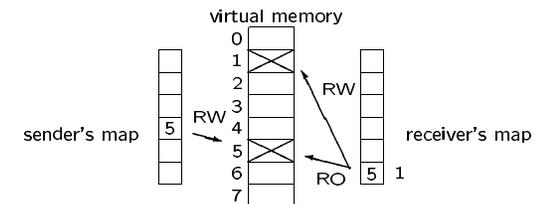
- NetMsgServer (NMS) is in user space
- R sends S a message with SEND right: NMS creates proxy X
- S sends R a message via proxy X: NMS delivers to port X
- networkwide name server: allows tasks to register ports for lookup

[XIII] Mach: Messages [309]

- fixed-length header and variable number of typed data objects
 - data
 - port rights
 - pointers to **out-of-line** data
- **send** message
 - *SEND_TIMEOUT*: sending data too fast
 - *SEND_NOTIFY*: if cannot be sent now, notify when OK
- **receive** message
 - *RCV_TIMEOUT*: block for only so long
 - *RCV_NO_SENDERS*: return if no senders

[XIII] Mach: Messages - Out-of-Line Data [310]

- pointer would be invalid in receiver's address space
- **copy-on-write**:
 - put the virtual memory **map** into the receiver's space
 - much faster than copying the data itself
 - if receiver only reads - OK
 - if receiver writes to a page - protection fault
 - distinction: read-only vs. copy-on-write
 - make copy of just the page, map it to receiver's space



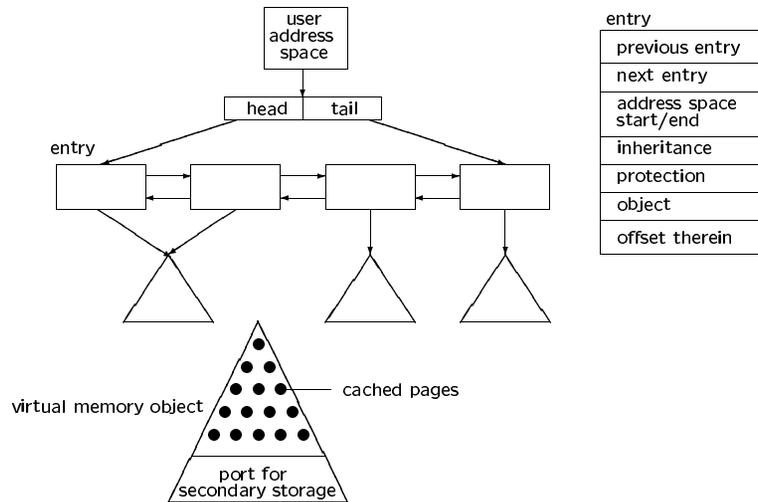
[XIII] Mach: Memory Management [311]

- object-oriented: message to port associated with memory object
- page fault: message to the object's port
- **user-level memory managers** instead of kernel
 - **external pager**
 - task may not have manager for a region
 - (not on secondary storage)
 - manager may fail to reduce resident pages when asked
 - use Mach's **default memory manager**
 - FIFO with "second chance"
- secondary storage: just like any other object
- physical memory: cache onto memory objects

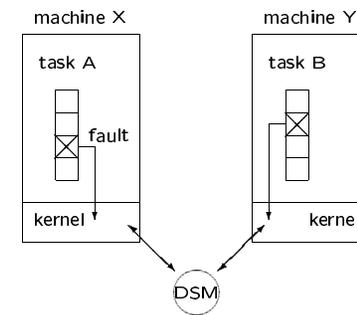
[XIII] Mach: Virtual Memory [312]

- 32-bits \Rightarrow 4GB
- 1K page size \Rightarrow 4 million page table entries
- Mach's virtual memory: **sparse**
 - **allocate region** of virtual memory
 - specify base VM address and size (say 50MB)
 - used for file objects, large messages
 - regions can be shared/inherited with other tasks
 - **deallocate region**
 - many holes of unallocated VM space
- page table: not the regular kind
 - entries for only currently allocated regions
 - cannot simply index into the table
 - check for page in valid region
 - **address map**

[XIII] Mach: Address Map for Sparse VM [313]



[XIII] Distributed Shared Memory Server [314]



- shared page is readable: may be replicated on multiple machines
- shared page is writable: only one copy
- DSM knows which machines have the page
- reader writes to the page: DSM sends messages to kernel
- upon acknowledgement: single writer is given permission

Index to Slides: Operating Systems [315]

I. Introduction	3	Deadlock	85	FIFO	174
II. Queuing Laws	18	Philosophers	88	OPT	178
Stream	23	Producer/Consumer	92	LRU	180
Utilization	25	Readers/Writer	95	CLOCK	186
Visit	26	Smokers	98	LFU/MFU	191
Bottleneck	27	Client/Server	103	Belady's Anomaly	195
Little's Law	29	Ring	105	Thrashing	198
M/M/1	32	Star	107	Working Set	201
General Response	35	IV. Deadlock	110	VI. File Management	204
III. Process Management	39	Allocation Graph	111	Partitions	207
A. Process	40	Prevention	114	Graph Directory	208
State Changes	43	Avoidance	116	Allocation Methods	209
VOS	44	Detection	119	UNIX Inode	216
PCB	47	Recovery	120	Free Space	217
Program Counter	48	Banker's Algorithm	122	VII. Distributed Systems	220
Context Switch	49	V. Memory Management	130	A. Systems/Networks	220
Interrupts	50	A. Main Memory	130	Circuit Switching	225
B. Scheduling	52	CPU/Job Schedulers	131	Contention	227
CPU Burst	58	Address Binding	134	ISO Model	228
FCFS	59	Dynamic Relocation	136	Distributed OS	231
RR	61	Partitions	141	RPC	232
SJF	63	Fragmentation	144	NSF	233
PRIO	67	Paging	147	B. Coordination	235
VOS	69	Logical Address	150	Mutual Exclusion	236
C. Synchronization	75	Segmentation	157	Logical Register	237
Critical Section	76	B. Virtual Memory	161	Leader Election	238
Semaphores	79	Page Fault	164		
Progress/Fairness	84	VM using MEM	167		

Index to Slides: Operating System Designs [316]

VIII. Obsolete	XII. UNIX Design	282
IX. Obsolete	Processes	283
X. Java Design	Scheduling	285
Goals	Memory	287
State Transitions	I/O System	290
Inheritance	IPC	291
Sync Method	XIII. Mach Design	292
Sync Block	Tasks/Threads	301
Wait/Notify	Synchronization	303
Sockets	Scheduling	305
Exceptions	Ports/Capabilities	306
XI. XINU Design	Messages	308
System Calls	Memory Objects	311
Memory		
Interrupts		
Real-Time Clock		
TTY I/O		
Disk I/O		