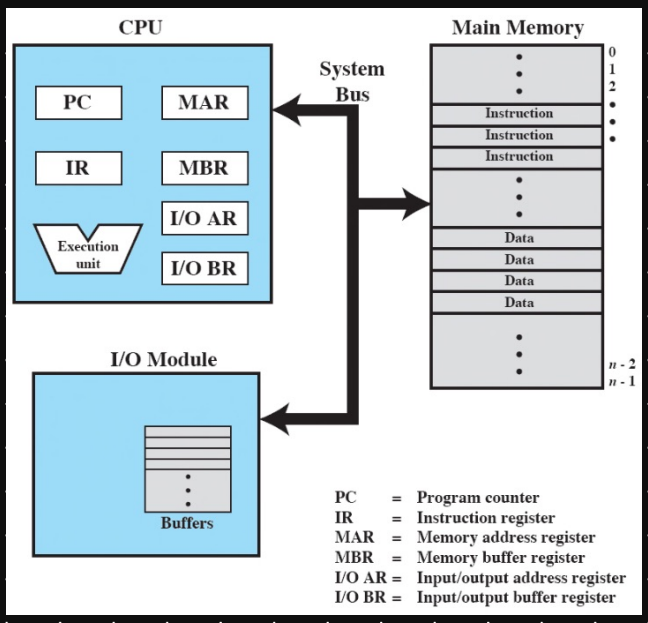
COMPUTER SYSTEM OVERVIEW

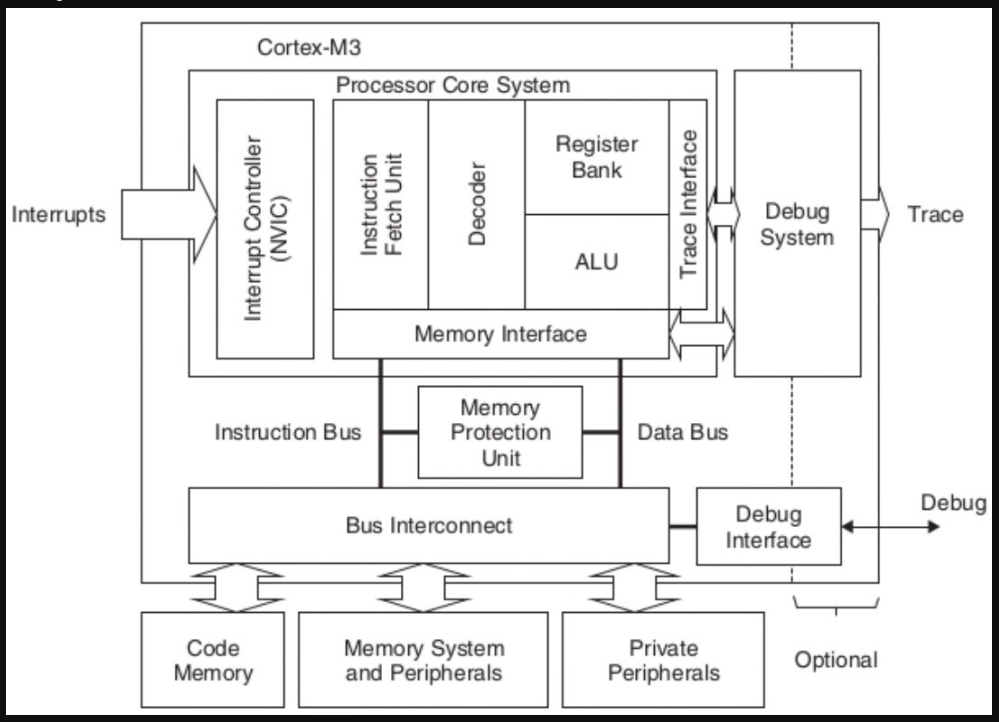
BASIC ELEMENTS

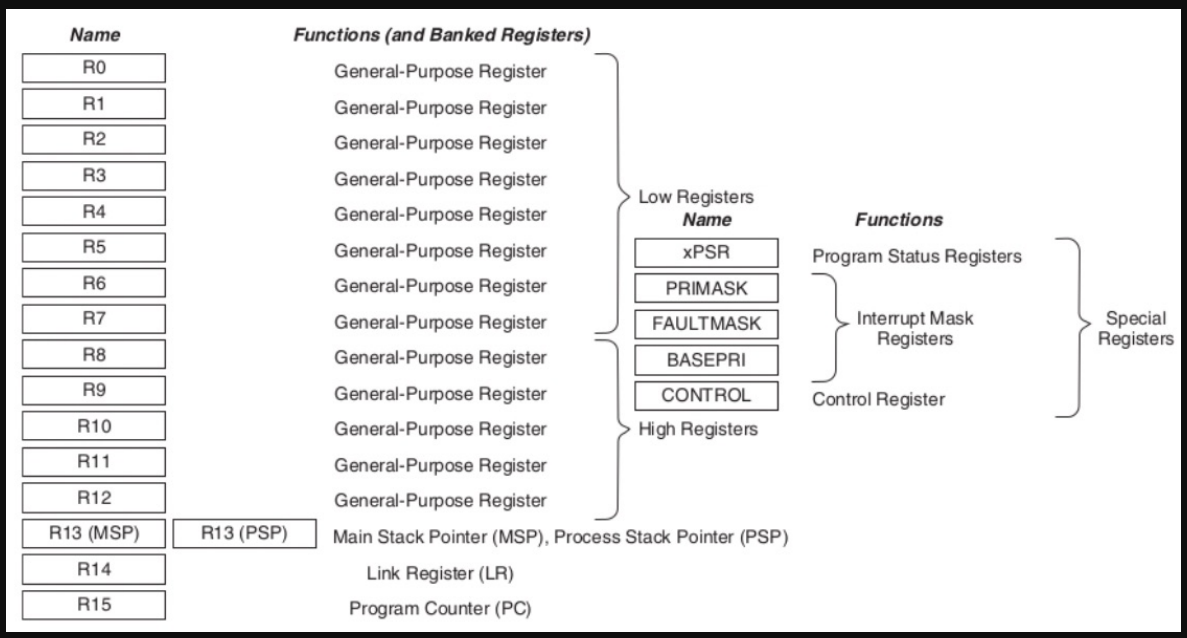
- an **operating system (OS)** is smth that helps users use available software
  - ↳ originally expensive hardware + cheap ppl
    - goal was efficiency (care abt **overhead**)
  - ↳ now cheap hardware + expensive ppl
    - goal is ease of use
- OS is standardized abstraction (**virtual machine (VM)**) implemented on underlying machine
  - ↳ manages resources
  - ↳ provides set of services to users
  - ↳ consumes resources
- a **computer** must contain:
  - ↳ 1+ processing units
  - ↳ memory
  - ↳ I/O modules
  - ↳ timers
  - ↳ interrupt controller
  - ↳ system bus to connect modules
- **processor** controls operations + performs data processing fcns
  - ↳ **CPU** is central processing unit
  - ↳ some registers / buffers:
    - **memory address register (MAR)**: specifies address for next read/write
    - **memory buffer register (MBR)**: contains data written into memory + receives data read from memory
    - I/O address register
    - I/O buffer register
- **main memory** stores data + programs
  - ↳ volatile (i.e. contents are lost when computer is shut down)
  - ↳ aka real / primary memory
- **I/O modules** move data btwn computer + external environment (e.g. secondary mem devices, communications equipment, terminals)
- **system bus** provides for communication among processors, main mem, + I/O modules
- top-lvl view:



- **user-visible registers** enable programmer to minimize main mem refs by optimizing register use
  - ↳ available to all programs (app + system)
  - ↳ depend on computer architecture
  - ↳ usually compiler-controlled but can be user-controlled
    - register + volatile keywords in C
    - clobber list in ASM
- **volatile** keyword indicates to compiler that var's val may be changed at any time externally
  - ↳ prevents compiler from making some optimizations
- **register** keyword tells compiler to store var in CPU register for faster access
- 2 categories of visible registers:
  - ↳ **data** : stores data
  - ↳ **address** : points to mem
    - indexed addressing
    - segment ptr
    - stack ptr
- **control + status registers** are invisible to user
  - ↳ used by processor to control operation of computer
  - ↳ used by privileged OS routines to control program executions
  - ↳ examples of invisible registers:
    - program counter (PC)
    - instruction register (IR)
    - program status word (PSW)
- **condition codes / flags** are part of PSW
  - ↳ bits set by processor hardware as result of ops
    - e.g. +ve, -ve, 0, or overflow
  - ↳ used for conditional branching

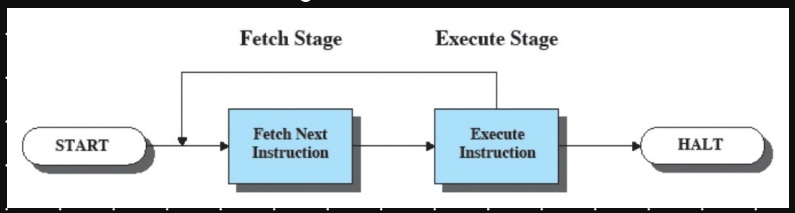
e.g. ARM Cortex-M3





INSTRUCTION EXECUTION

basic instruction cycle:

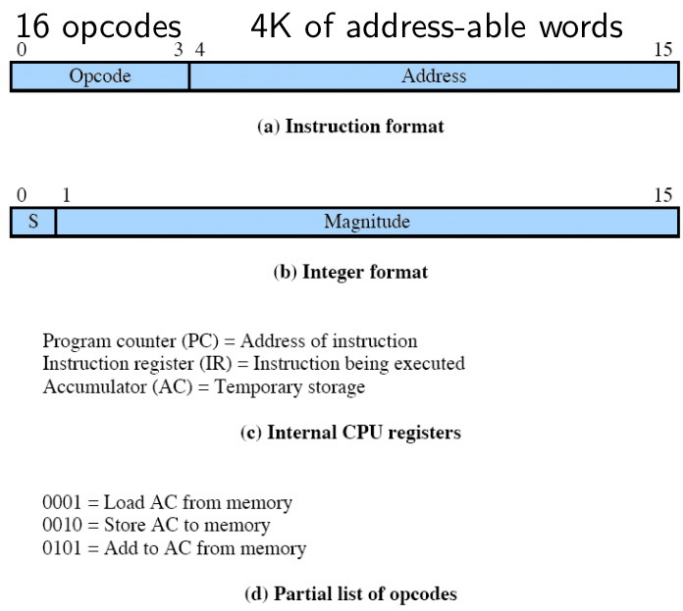


↳ processor fetches ins from mem

↳ PC holds address of ins to be fetched next
↳ PC inc after each fetch
↳ fetched ins loads into IR

categories of actions dictated by ins are processor - mem, processor - I/O, data processing, control
↳ no clear boundaries b/c ins may contain several of these actions

e.g.



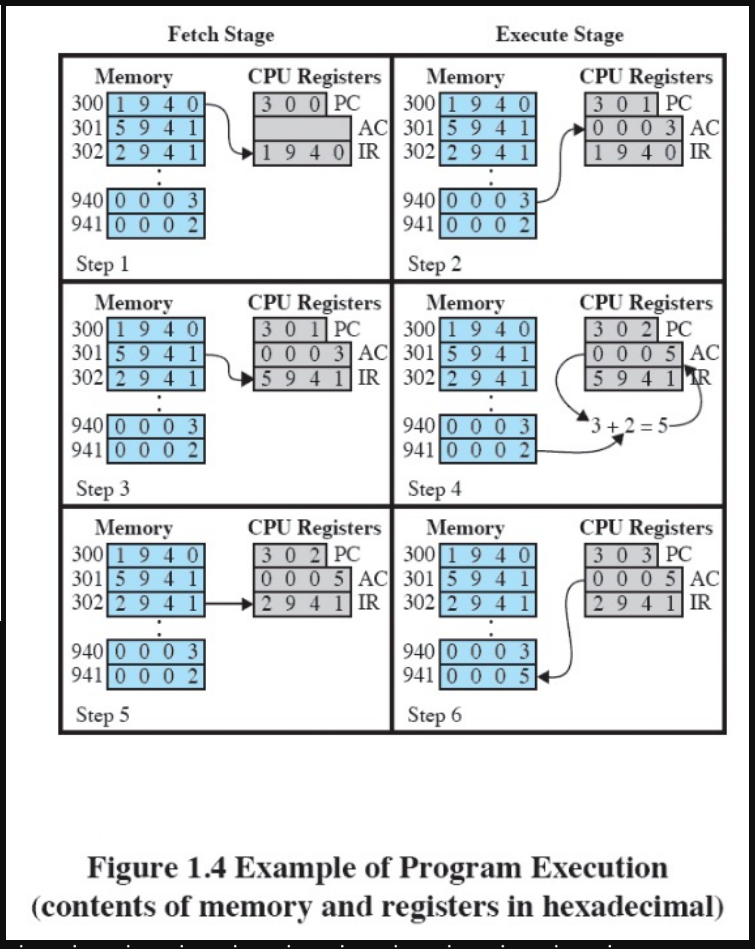


Figure 1.4 Example of Program Execution (contents of memory and registers in hexadecimal)

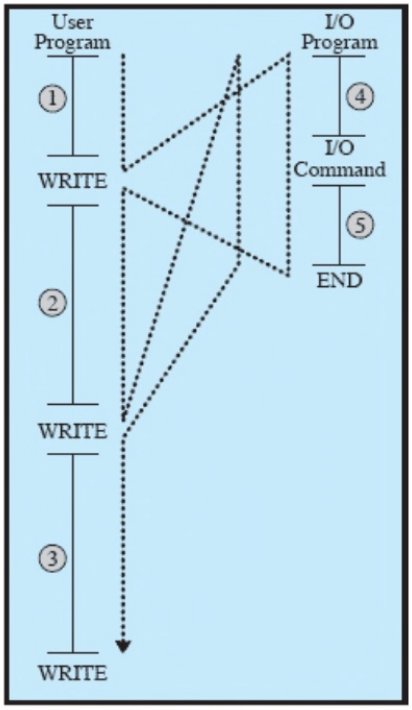
INTERRUPTS

most I/O devices are slower than processor so interrupts help improve processor utilization by changing normal sequencing

classes of interrupts:
↳ program: generated by some condition that occurs as result of ins execution (e.g. arithmetic overflow, ref outside user's allowed mem space)
↳ timer: allows OS to perform fcns on regular basis
↳ I/O: signals normal completion of op or variety of error conditions
↳ hardware failure: generated by failure (e.g. power, mem parity)

e.g. program flow of control
↳ no interrupts:

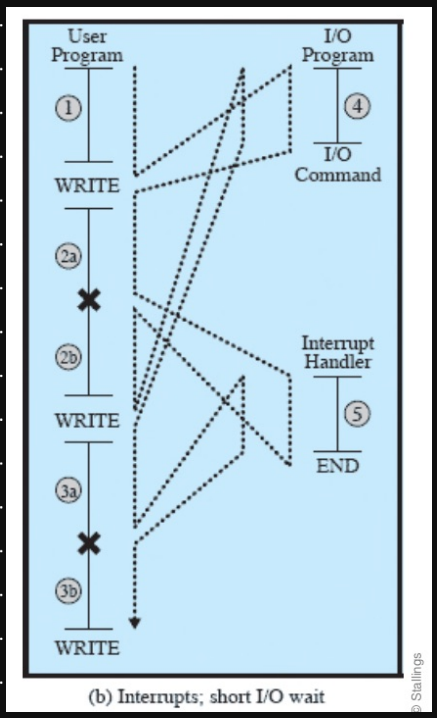
No interrupts.



(a) No interrupts

Wait until completion Here.

↳ interrupts w/ short I/O wait



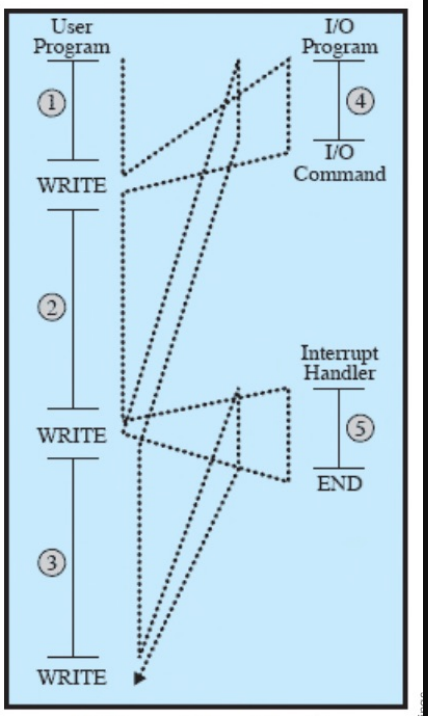
(b) Interrupts; short I/O wait

- issue is non-linear control flow so interrupts can occur at unwanted times

↳ interrupts w/ long I/O wait

Interrupt with long I/O wait.

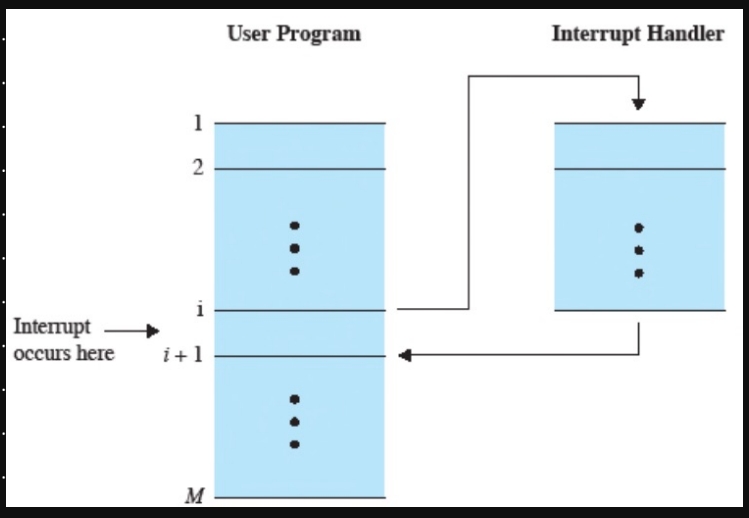
Need to wait for first I/O here.



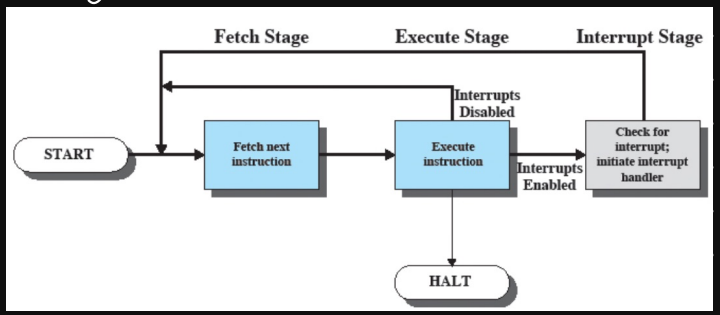
(c) Interrupts; long I/O wait

if interrupt detected by processor
- ↳ suspend execution of program + store snapshot
- ↳ execute interrupt-handler routine
- ↳ resume execution

transfer of control via interrupts:

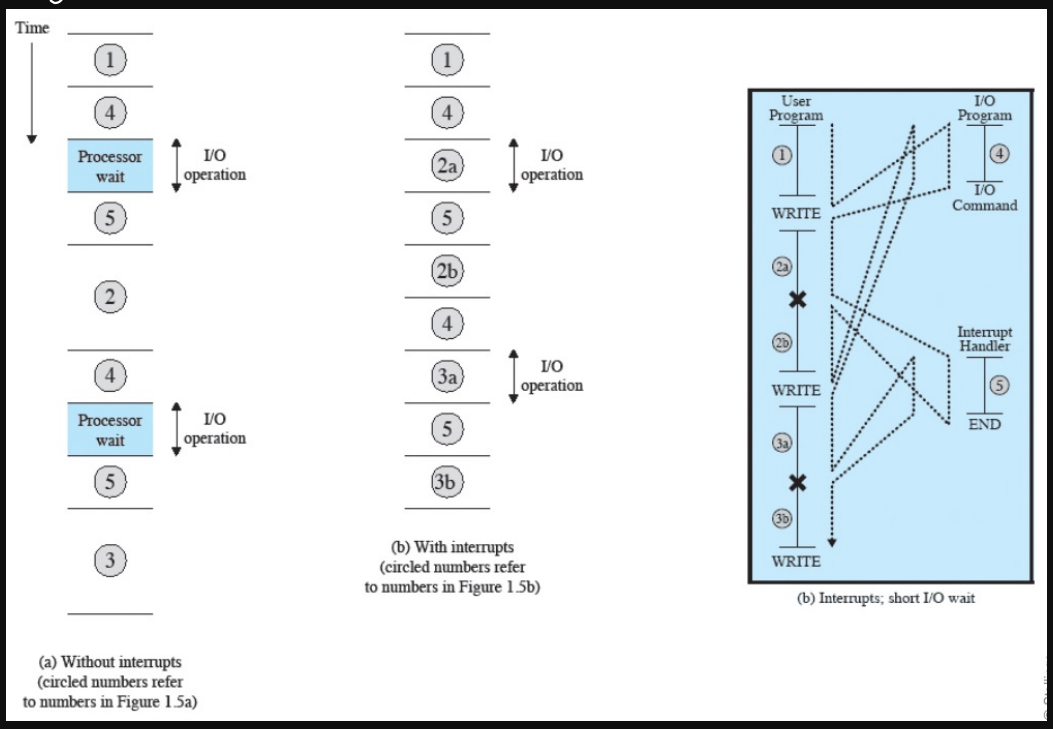


ins cycle w/ interrupts:

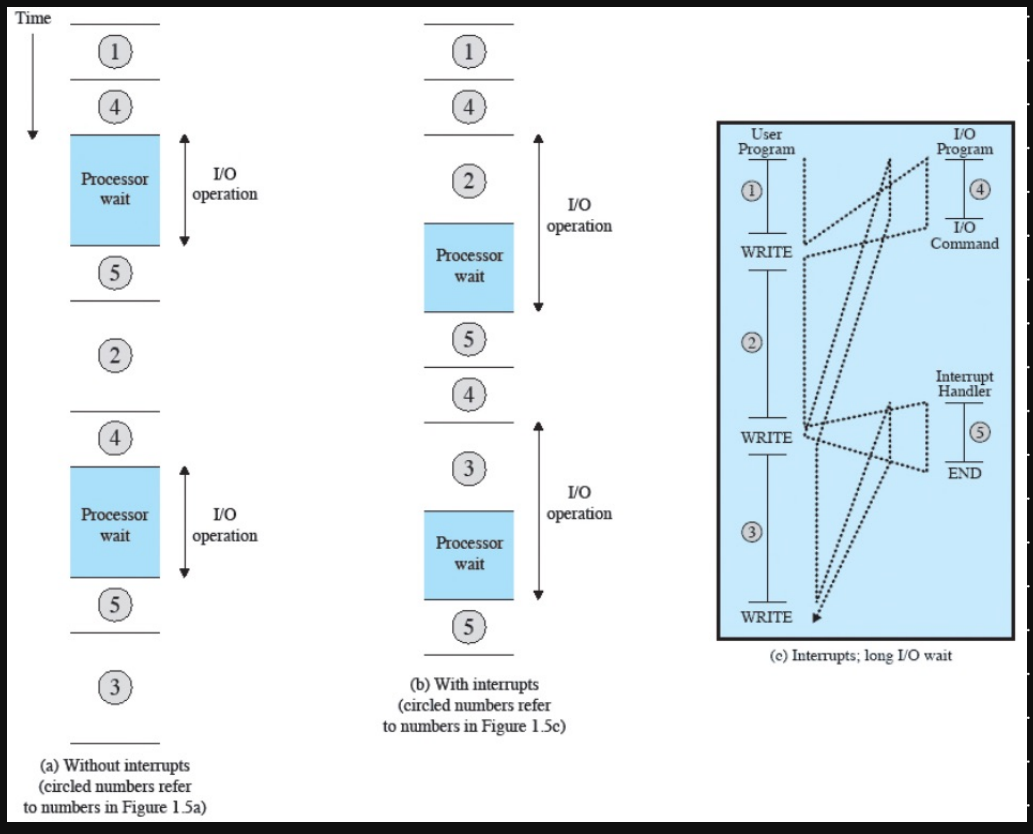


↳ no special code required b/c everything happens in hardware

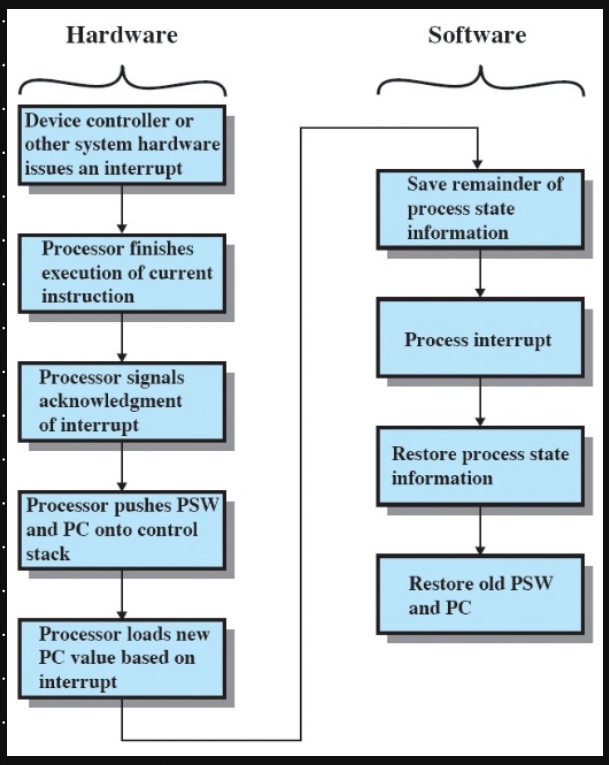
e.g. short I/O wait



e.g. long I/O wait



simple interrupt processing:

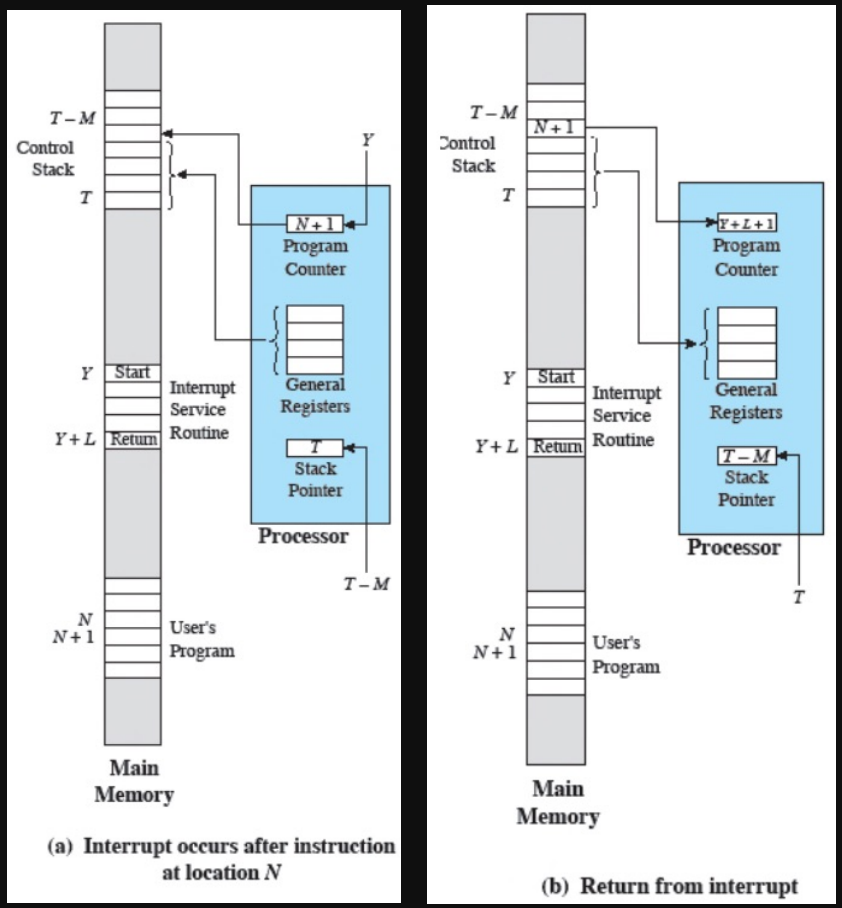


↳ lean ISR saves nothing + we must tell it what to save

↳ heavy ISR saves every register touchable by user by default

- std process

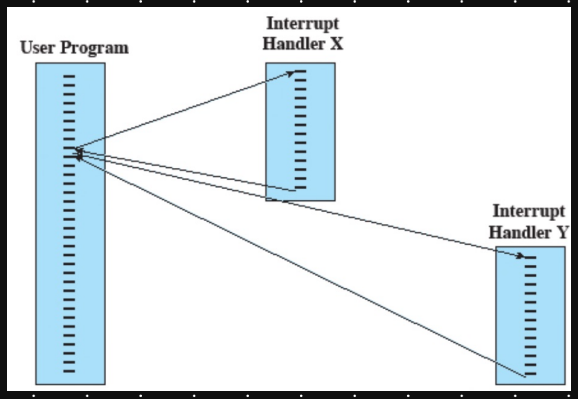
e.g. storing & restoring from snapshot



2 approaches to deal w/ multiple interrupts

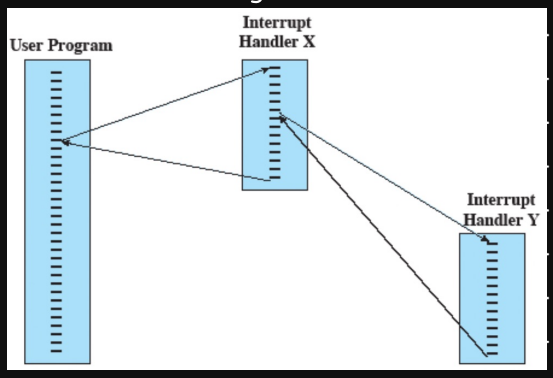
↳ sequential:

- disable interrupts in interrupt service routine (ISR)
- no priority info
- not useful for time critical elmts



↳ nested:

- priorities for ISR + how to assign them
- finite # priority lvls



long I/O requests leave processor idle so we enable multiprogramming

↳ processor has 1+ program to execute

↳ seq of programs executed depend on rltive priority + if they're waiting for I/O

↳ after interrupt handler completes, control may not return to program executing at time of interrupt

programmed I/O: I/O module performs action instead of processor

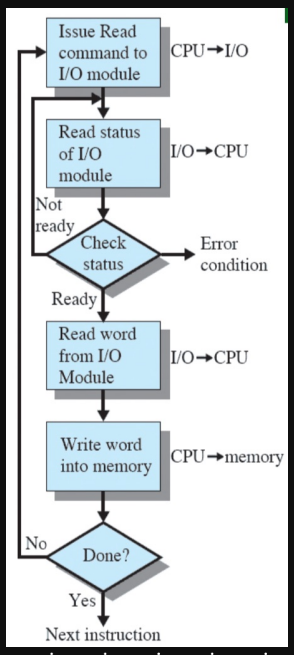
↳ no interrupts

↳ sets bits in I/O status register

↳ processor checks status until op is complete

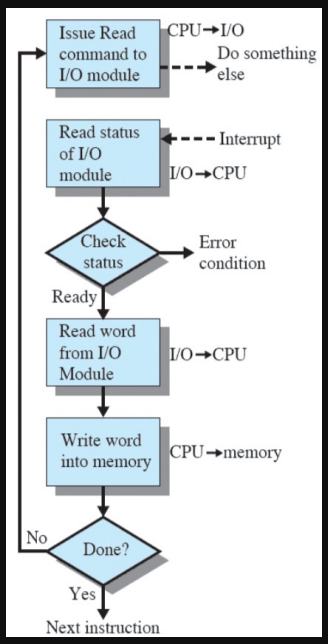
↳ aka busy waiting

↳



interrupt-driven I/O: processor is interrupted when I/O module is rdy to exchange data

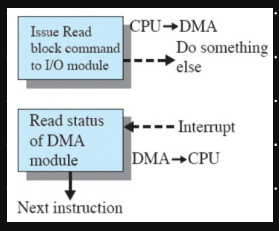
- ↳ processor saves context of program executing + begins interrupt handler
- ↳ consumes a lot of processor time b/c every word read/written passes thru processor
- ↳ no needless waiting
- ↳



direct mem access (DMA): transfers block of data directly to/from mem

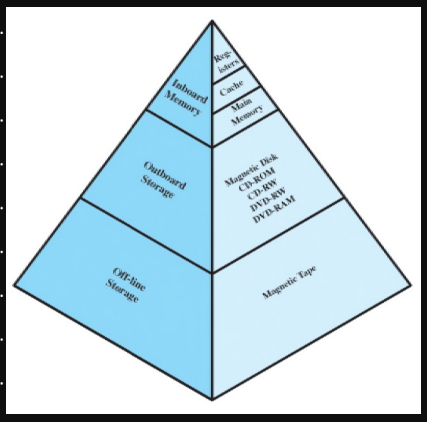
- ↳ send interrupt when transfer's complete
- ↳ more efficient
- ↳ not always avail (e.g. external peripherals)

↳

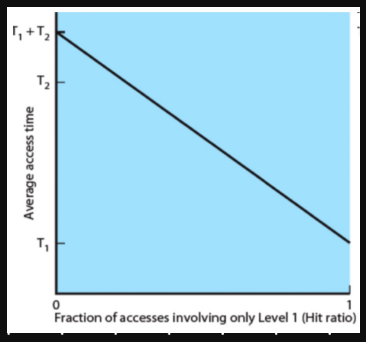


ORGANIZING STORAGE

- major constraints in mem are amount, speed, + expense
- going down **mem hierarchy**, following occur
  - ↳ dec cost per bit
  - ↳ inc capacity
  - ↳ inc access time
  - ↳ dec freq of access
- mem hierarchy:



- e.g. performance of 2-lvl mem

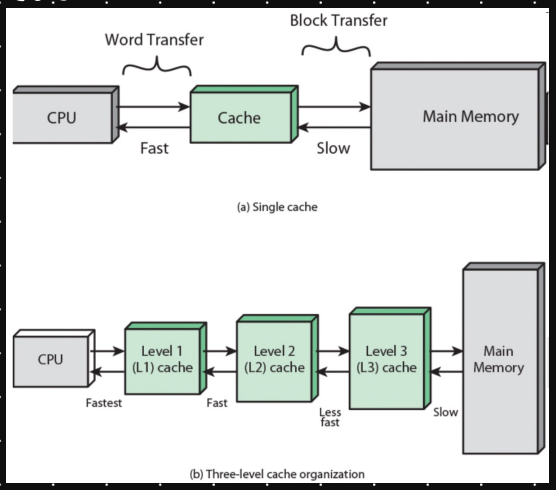


$$H\,(\text{hit ratio}) = \frac{\#\text{mem accesses in cache}}{\#\text{total mem accesses}}$$

  - ↳ for higher percentages of lvl 1 access, avg total access time is closer to T_1
- **principle of locality**: mem refs by program tend to cluster
  - ↳ organize data st accesses on each lvl are << next lvl of mem hierarchy

- processor accesses mem at least once per ins cycle so its execution is limited by **mem cycle time** (i.e. time to read/write 1 word from/to mem)
  - ↳ exploit principle of locality by using **cache**, which is small + fast mem that's invisible to OS
    - cache interacts w/ other mem management hardware

- cache contains copy of portion of main mem
  - ↳ processor checks cache + if not found, block is read into cache
  - ↳ b/c of principle of locality, likely that future refs will hit block
- cache + main mem:



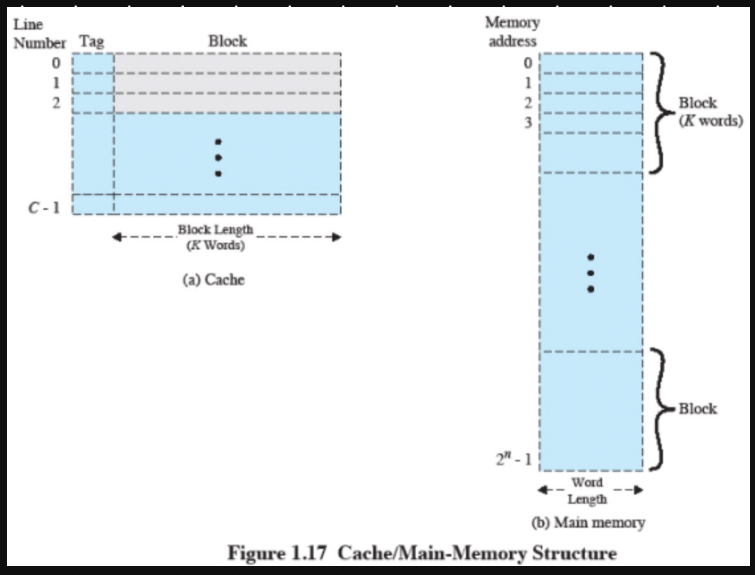


Figure 1.17 Cache/Main-Memory Structure

- cache read op:

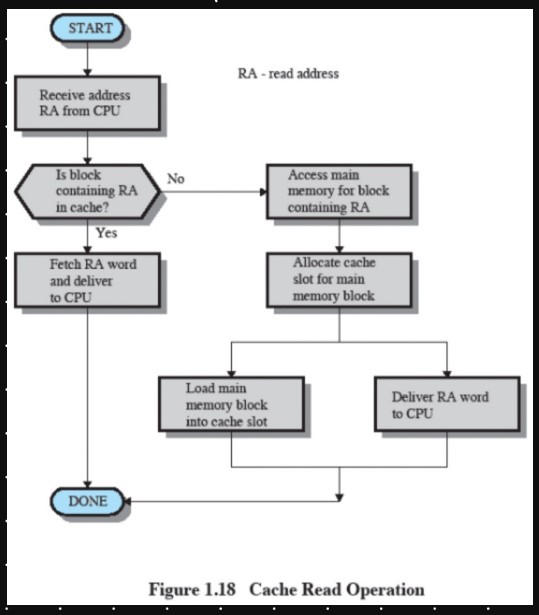
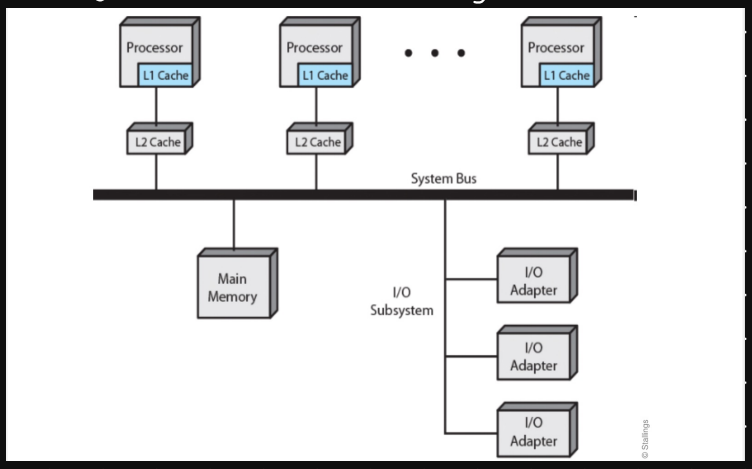


Figure 1.18 Cache Read Operation

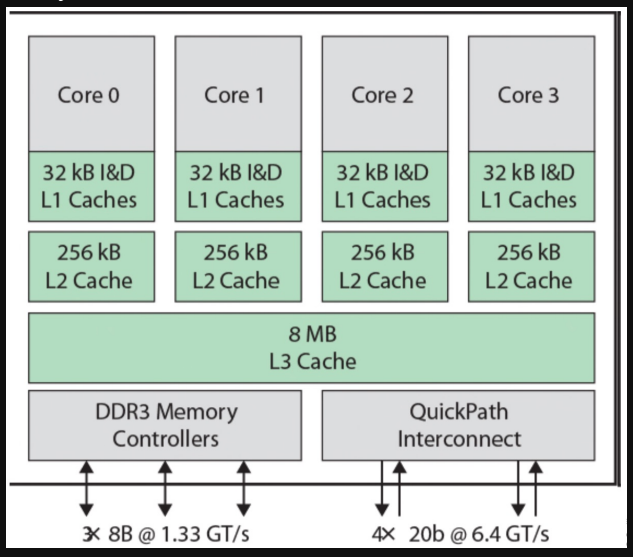
- mapping fcn determines where to store block in cache
- replacement fcn chooses which block to replace when new block needs to be loaded into full cache
  - ↳ e.g. least-recently used LRU, least freq used
- write policy dictates when mem write op happens b/c if block in cache is altered, main mem needs to reflect that change too
  - ↳ write back: writing occurs only when block is replaced
  - ↳ write thru: writing occurs every time block updates

MULTIPROCESSORS

symmetric multiprocessor (SMP) is stand-alone computer that has
- ↳ 2+ processors
- ↳ processors share same main mem + access to I/O devices
- ↳ processors interconnected by bus + perform same fcns
- ↳ integrated OS provides interaction btwn processors + programs
- ↳ **homogeneous** multiprocessing system
- ↳



multicore computer (aka chip multiprocessor) combines 2+ cores on 1 chip
- ↳ each core has components of indep processor
- ↳ cores may share common cache
- ↳ can be homogeneous (same cores) or heterogeneous (diff cores)
- ↳ e.g. Intel Core i7



OPERATING SYSTEMS

OBJECTIVES AND FUNCTIONS

- **operating system (OS)**: program that controls execution of apps + acts as standardized interface btwn apps + hardware
- OS objectives:
  - ↳ **convenience**: need no knowledge of underlying hardware + provides abstraction of std services
  - ↳ **efficiency**: moves burden of optimization from developers to tools
  - ↳ **ability to evolve**: can develop, test, + replace internals w/o changing interface

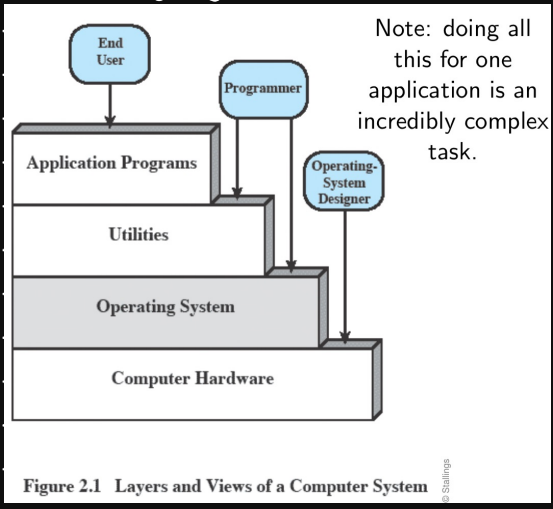


Figure 2.1 Layers and Views of a Computer System

- **OS services** include:
  - ↳ program development (e.g. editors + debuggers)
  - ↳ program execution
    - in multiprogramming OS + microcontrollers
  - ↳ access to I/O devices
    - reduces knowledge required to read / write
  - ↳ controlled access to files
  - ↳ system access control
    - protection from unauthorized access
  - ↳ error detection + response
  - ↳ accounting (i.e. collecting usage stats, monitoring performance)
- OS is responsible for **managing resources** avail in computer
  - ↳ fcns same way as ordinary software, meaning it's a program that's executed + frequently relinquishes control of processor

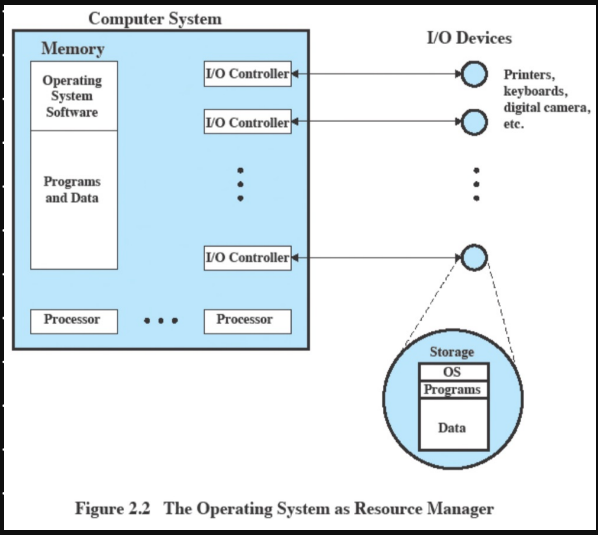


Figure 2.2 The Operating System as Resource Manager

- kernel / nucleus is portion of OS that contains most frequently used fcns
  - ↳ in main mem
  - ↳ embedded OSs are often only a nucleus

EVOLUTION IN OSS

- reasons for evolution:
  - ↳ hardware upgrades + new types of software
    - e.g. paging, GUI
  - ↳ new services
  - ↳ fixes
- serial processing: users access computer in series
  - ↳ no OS
  - ↳ machines run from console w/ display lights, toggle switches, input device, + printer
  - ↳ shortcomings:
    - schedule time: time reservations on hard copy signup sheets + non-optimal work env (e.g. forced interruptions b/c went over reserved time)
    - long setup times for program: load compiler, source program, save compiled program, loading + linking
- simple batch system: use monitor, which is software that controls sequence of events, to improve utilization.
  - ↳ user doesn't have direct access to processor
  - ↳ monitor batches jobs tgt
  - ↳ program returns control to monitor once finished
- job control language (JCL) provides ins to monitor on what compiler to use + what data to use
- hardware features that came w/ monitor (aka batch OS):
  - ↳ mem protection: doesn't allow mem area containing monitor to be altered
  - ↳ timer: prevents job from monopolizing system

↳ privileged ins: certain ML ins can only be executed by monitor
↳ interrupts

modes of op were used to protect users from each other + kernel from users
↳ 2 modes
- user: certain ins can't be executed
- kernel: privileged ins + access to protected mem areas

auto job sequencing improves thruput but IO still slow
↳ e.g.

Read one record from file	15 μs
Execute 100 instructions	1 μs
Write one record to file	15 μs
TOTAL	31 μs

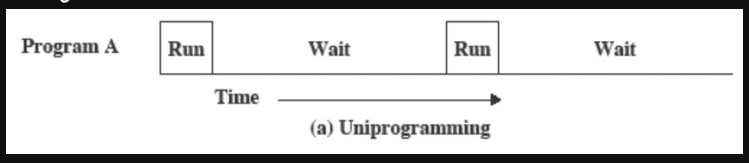
Percent CPU Utilization $= \frac{1}{31} = 0.032 = 3.2\%$

96.8% waiting!

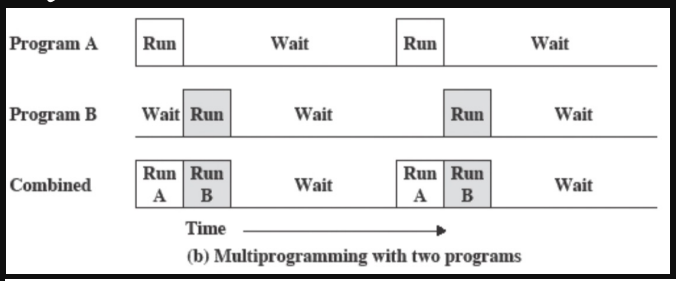
Figure 2.4 System Utilization Example

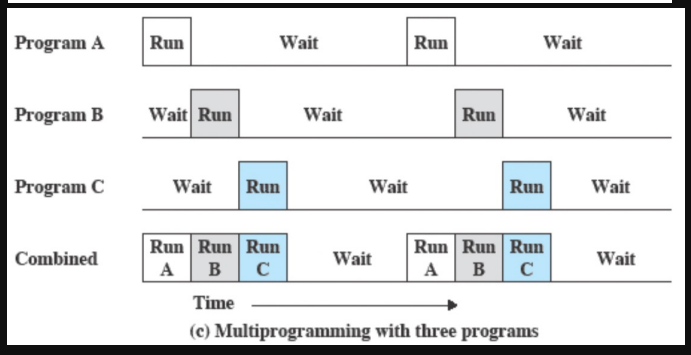
↳ soln is multiprogramming

uniprogramming: processor must wait for I/O ins to complete before proceeding
↳ e.g.



multiprogramming: when 1 job needs to wait for I/O, processor switches to another job
↳ e.g.





e.g.

Table 2.1 Sample Program Execution Attributes

	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration	5 min	15 min	10 min
Memory required	50 M	100 M	75 M
Need disk?	No	No	Yes
Need terminal?	No	Yes	No
Need printer?	No	No	Yes

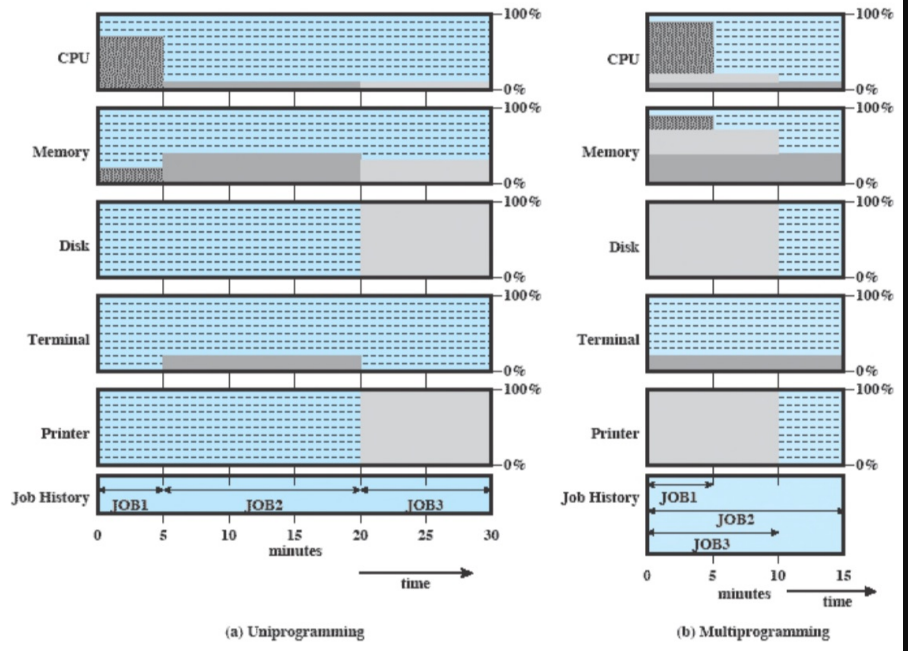


Figure 2.6 Utilization Histograms

	Uniprogramming	Multiprogramming
Processor use	20%	40%
Memory use	33%	67%
Disk use	33%	67%
Printer use	33%	67%
Elapsed time	30 min	15 min
Throughput	6 jobs/hr	12 jobs/hr
Mean response time	18 min	10 min

time sharing: multiple users simultaneously access system thru terminals
- ↳ share time on CPU
  - ° if there's n users requesting service at same time, each will see $\frac{1}{n}$ of effective computer capacity
- ↳ use multiprogramming to handle multiple interactive jobs
- ↳ human rxn time is rltively slow so response time should be similar to dedicated computer

	Batch multiprogramming	Time sharing
Principle objective	Maximize processor use	Minimize response time
Source of directives to OS	JCL cmds provided w/job	Cmds entered at terminal

MAJOR ACHIEVEMENTS

major achievements in OSs:
- ↳ processes
- ↳ mem management
- ↳ info protection + security
- ↳ scheduling + resource management
- ↳ system structure
- ↳ virtualization

process consists of 3 components
- ↳ executable program
- ↳ associated data needed by program
- ↳ execution context of program
  - ∘ all info OS needs to manage process

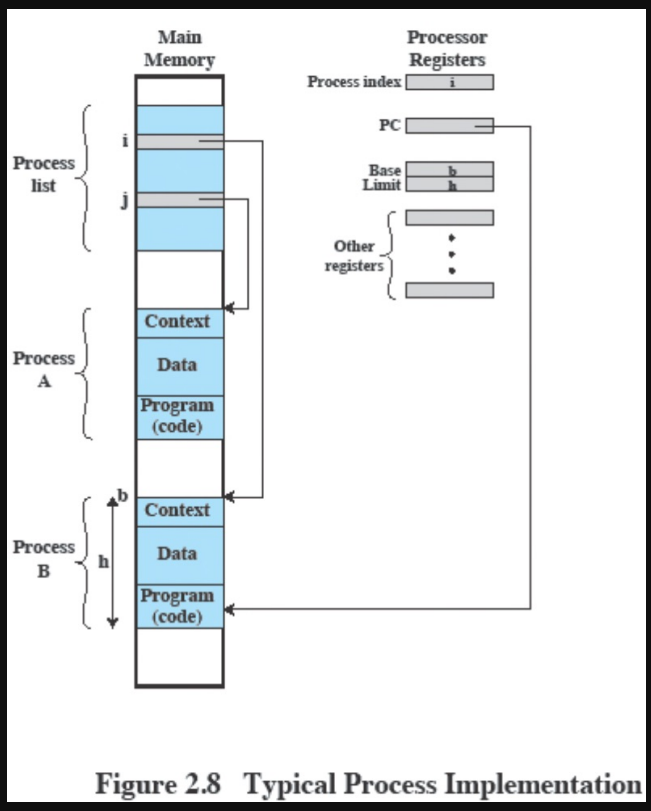


Figure 2.8 Typical Process Implementation

difficulties w/ designing system software:
- ↳ improper synchronization: lost / duplicate signals can lead to lost data or empty read)
- ↳ failed mutual exclusion: state corruption on shared mem
  - ∘ e.g. 2 users editing same file at same time
- ↳ nondeterminate program op: interference among programs due to shared mem

allocation, IO access, etc.
- schedule order of programs may affect outcomes.
  - ↳ deadlocks: 2+ programs forever waiting for each other

OS has 5 principle **mem storage management** responsibilities:
- ↳ process isolation
- ↳ auto allocation + management
  - e.g. don't care how 1kb is acc allocated
- ↳ support of modular programming
- ↳ protection + access control: appropriate sharing of mem
- ↳ long-term storage

virtual mem allows us to address mem from logical POV w/o regard to how much mem is avail
- ↳ virtual address: pg # + offset in pg
- ↳ real address: physical address

paging allows process to be comprised of fixed-size blocks (i.e. pgs)
- ↳ each pg may be located anywhere in main mem

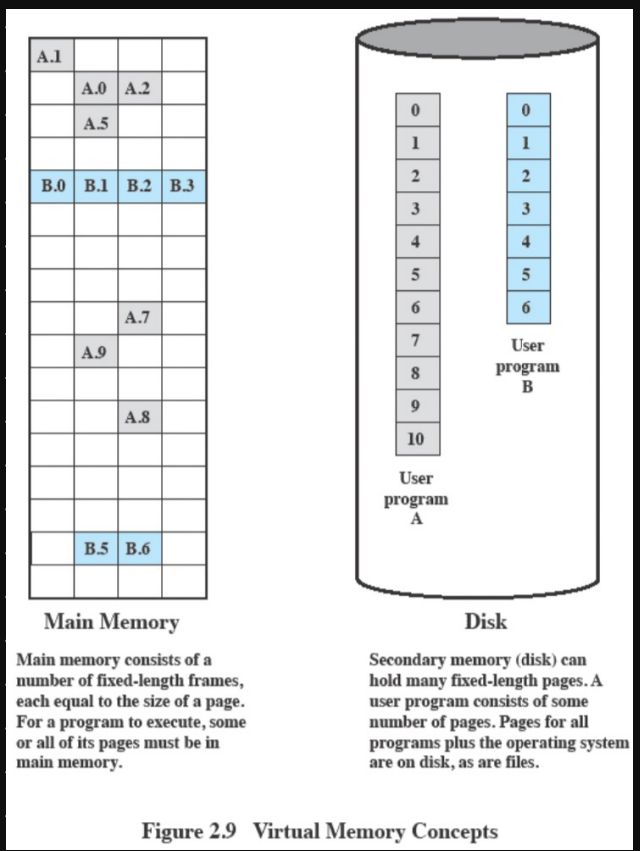


Figure 2.9 Virtual Memory Concepts

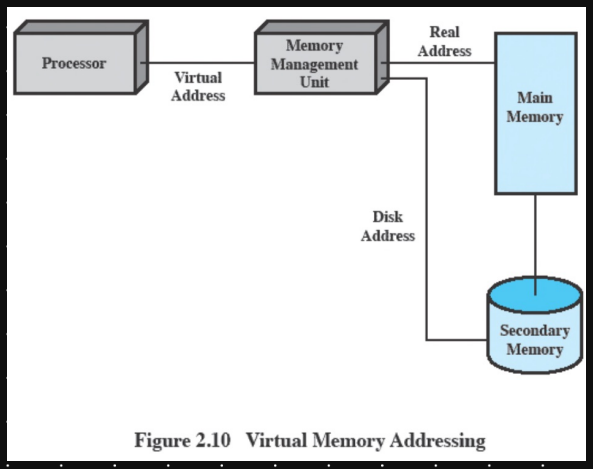


Figure 2.10 Virtual Memory Addressing

info protection + security in OSs can be grouped into 4 categories:
- ↳ availability: protect system against interruption
- ↳ confidentiality: assuring users w/unauthorized access can't read data
  - e.g. chmod to change permissions
- ↳ data integrity: protection of data from unauthorized modification

↳ authenticity: proper verification of users' identities + validity of msgs/data
  - e.g. phishing attacks, cross site scripting

resource allocation + scheduling policy of OS must consider:
↳ fairness: equal + fair access to resources
↳ differential responsiveness: discriminate among diff classes of jobs
  - e.g. if process is waiting for I/O device, OS should schedule that process ASAP so it's free later
  - aka quality of service (QoS)
  - necessity for real-time systems
↳ efficiency: maximize thruput, minimize response time, + accomodate as many uses as possible

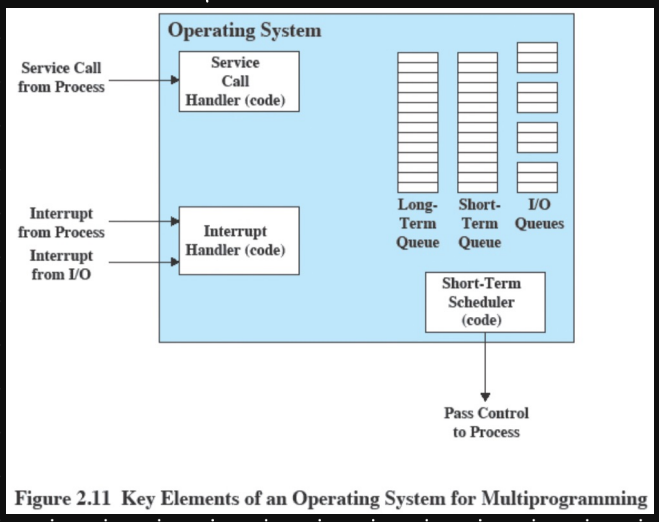


Figure 2.11 Key Elements of an Operating System for Multiprogramming

problems w/ OSs:
↳ usually late on delivery
↳ latent bugs
↳ slow performance
↳ security vulnerabilities

use hierarchic system structure to improve OS
↳ system is series of lvls
↳ each lvl performs subset of fcns + relies on lower lvl to perform more primitive fcns
↳ decomposes problem into set of more manageable subproblems
↳

Level	Name	Objects	Example Operations
13	Shell	User programming environment	Statements in shell language
12	User processes	User processes	Quit, kill, suspend, resume
11	Directories	Directories	Create, destroy, attach, detach, search, list
10	Devices	External devices, such as printers, displays, and keyboards	Open, close, read, write
9	File system	Files	Create, destroy, open, close, read, write
8	Communications	Pipes	Create, destroy, open, close, read, write

7	Virtual memory	Segments, pages	Read, write, fetch
6	Local secondary store	Blocks of data, device channels	Read, write, allocate, free
5	Primitive processes	Primitive processes, semaphores, ready list	Suspend, resume, wait, signal
4	Interrupts	Interrupt-handling programs	Invoke, mask, unmask, retry
3	Procedures	Procedures, call stack, display	Mark stack, call, return
2	Instruction set	Evaluation stack, microprogram interpreter, scalar and array data	Load, store, add, subtract, branch
1	Electronic circuits	Registers, gates, buses, etc.	Clear, transfer, activate, complement

MODERN OPERATING SYSTEMS

- **microkernel** architecture assigns only a few essential fcns to kernel
  - ↳ address spaces
  - ↳ interprocess communication (IPC)
  - ↳ basic scheduling

  as opposed to **monolithic kernel** (lots of OS fcnalities in 1 large kernel), microkernel treats other OS services as processes run in user mode
  - ↳ decouples kernel + server dev
- **multithreading** is when a process is divided into threads that can run concurrently
  - ↳ **thread**: dispatchable unit of work, executes sequentially, + is interruptable
    - includes processor context (e.g. PC + stack ptr) + data area for stack
  - ↳ process: collection of threads
- **symmetric multiprocessing (SMP)** refers to both hardware architecture + OS behaviour that exploits it

  multithreading + SMP complement each other + can be used effectively tgt

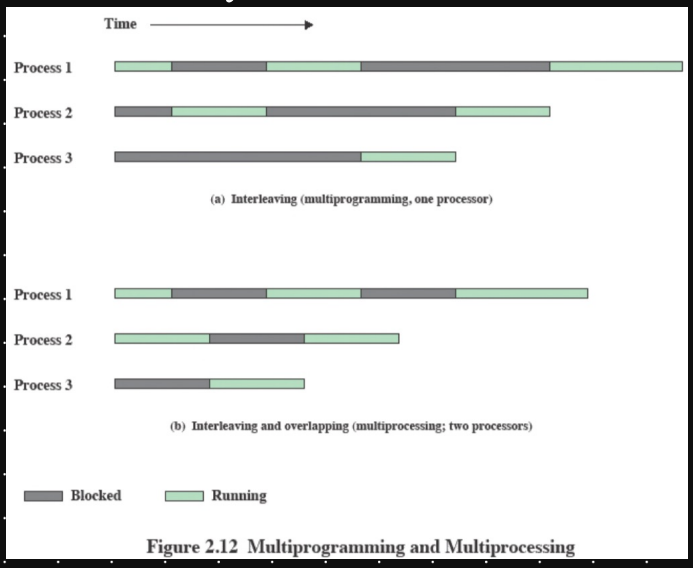


Figure 2.12 Multiprogramming and Multiprocessing

- **distributed OS** provides illusion of single main + secondary mem space
- **obj.-oriented design** allows programmer to customize OS by adding modular extensions to kernel w/o disrupting system integrity

- **fault** is erroneous hardware / software state from component failure,

operator error, physical interference from env, design, program error, or data struct error
  ↳ permanent
  ↳ temporary
    - transient: occurs only once
    - intermittent: occurs at multiple, unpredictable times
      → e.g. loose connection
- fault tolerance built by adding spatial/physical, temporal, + info redundancy
- OS mechanisms that support fault tolerance:
  ↳ process isolation
  ↳ concurrency controls
  ↳ VMs
  ↳ checkpoints + rollbacks

- key multicore/processor design considerations:
  ↳ concurrency: reentrant routines (i.e. can be called again before it finished prev execution), proper management of kernel structs
  ↳ scheduling: user vs kernel lvl
  ↳ synchronization: I/O management from single/multiple cores
  ↳ mem management: spatial separation of mem
  ↳ reliability + fault tolerance
- possible paradigm is to assign 1 app per core w/no switching

PROCESS DESCRIPTION AND CONTROL

MANAGING PROCESSES

- possible defns of **process**:
  - ↳ program execution
  - ↳ instance of program running on computer
  - ↳ entity that can be assigned to + executed on processor
  - ↳ unit of activity characterized by execution of seq of ins, current state, + associated set of system ins
- 2 essential elmts of process are **program code** + **set of data**
- while program is executing, process can be **uniquely characterized** by
  - ↳ identifier: unique ID
    - usually abbr as **process ID (PID)**
  - ↳ state
    - if program is currently executing, in **running state**
  - ↳ priority
    - rltive to other processes
  - ↳ mem ptrs
  - ↳ context data
    - e.g. registers, PSW, PC
  - ↳ I/O status info
    - e.g. outstanding I/O requests, used I/O devices
  - ↳ accounting info
    - e.g. amount of processor time, time limits, threads
- **process control block (PCB)**: data structure that contains process elmts
  - ↳ created + managed by OS
  - ↳ allows support for multiple processes
  - ↳ users aren't allowed to manipulate PCB b/c it contains sensitive info
  - ↳

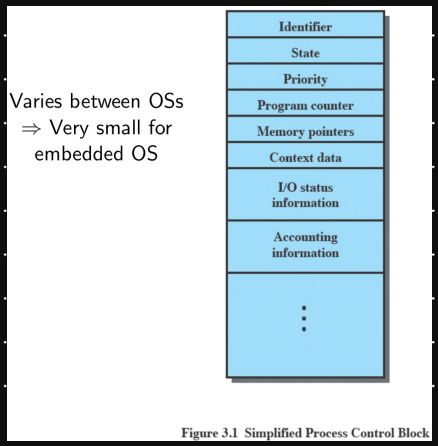


Figure 3.1 Simplified Process Control Block

↳ e.g. PICOS18

```
typedef rom const struct _rom_desc_tsk
{
    unsigned char prioinit;
    unsigned char *stackAddr;
    void (*addr_ROM)(void);
    unsigned char tskstate;
    unsigned char tskid;
    unsigned int  stksize;
} rom_desc_tsk;
```

Instantiation:

```
/***************************************
 * --------------- task VM ---------------
 ***************************************/
rom_desc_tsk rom_desc_task_vm = {
    TASK_VM_PRIO,      /* prioinit from 0 to 15       */
    _stack_vm,         /* stack address (16 bits)     */
    TASK_VM,           /* start function      */
    READY,             /* state at init phase         */
    TASK_VM_ID,        /* id_tsk from 0 to 15         */
    sizeof(_stack_vm) /* stack size     (16 bits)     */
};
```

trace of process is seq of ins that executes for it.

dispatcher switches processor from 1 process to another

e.g.

(a) Trace of Process A	(b) Trace of Process B	(c) Trace of Process C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

Figure 3.3 Traces of Processes of Figure 3.2

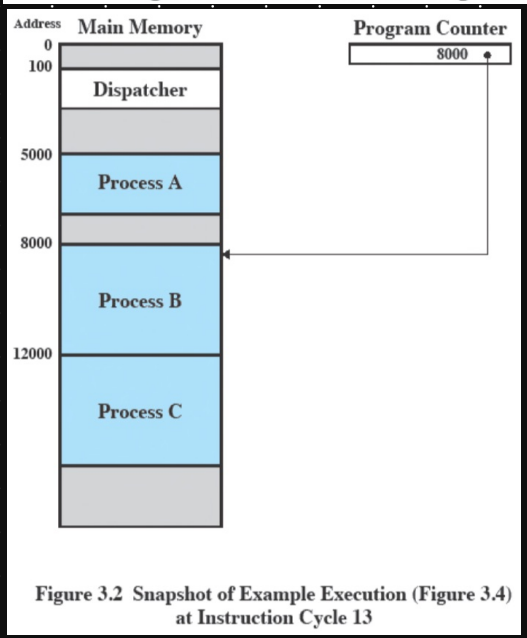


Figure 3.2 Snapshot of Example Execution (Figure 3.4) at Instruction Cycle 13

1	5000		27	12004	
2	5001		28	12005	
3	5002		---------- Timeout		
4	5003	A	29	100	
5	5004		30	101	
6	5005		31	102	D
---------- Timeout			32	103	
7	100		33	104	
8	101		34	105	
9	102	dispatcher (D)	35	5006	
10	103		36	5007	
11	104		37	5008	
12	105		38	5009	A
13	8000		39	5010	
14	8001		40	5011	
15	8002	B	---------- Timeout		
16	8003		41	100	
---------- I/O Request			42	101	
17	100		43	102	D
18	101		44	103	
19	102		45	104	
20	103	D	46	105	
21	104		47	12006	
22	105		48	12007	
23	12000		49	12008	C
24	12001		50	12009	
25	12002	C	51	12010	
26	12003		52	12011	
			---------- Timeout		

100 = Starting address of dispatcher program

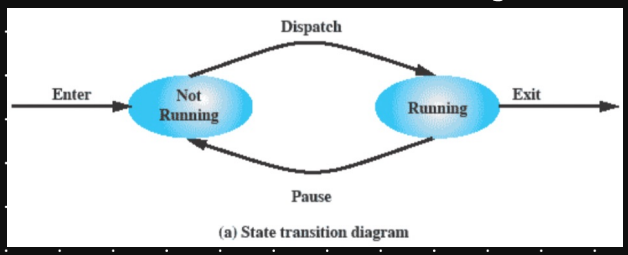
Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

Figure 3.4 Combined Trace of Processes of Figure 3.2

↳ OS only allows process to execute for max of 6 ins cycles
↳ timeout won't happen during dispatch b/c it doesn't count twd the 6 ins cycles

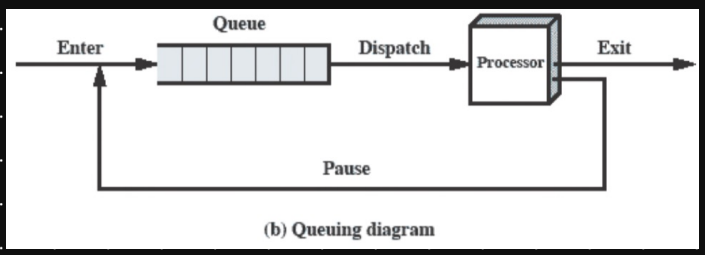
2-state process model: process may be in 1 of 2 states: running + non-running

↳



(a) State transition diagram

OS structure:
↳ stores data abt process (PCB)
↳ processes must wait in queue until it's their turn
↳



(b) Queuing diagram

reasons for process creation:
↳ new batch job
↳ interactive logon
↳ created by OS to provide service
  • e.g. OS creates printing process on behalf of user program
↳ spawned by existing process
  • i.e. parent process explicitly creates child process

reasons for process termination:
↳ normal completion
↳ time limit exceeded
  • ulimit cmd to set / display user-lvl resource limits
↳ mem unavail
↳ mem loc bounds violation
↳ protection error
  • e.g. unauthorized file access
↳ arithmetic error
↳ time overrun
  • i.e. process has waited longer than specified max for event to occur
↳ I/O failure
↳ invalid ins
↳ privileged ins
  • e.g. process tries to use ins only for OS
↳ data misuse

↳ operator / OS intervention
↳ parent termination
↳ parent request

core dump is file that captures mem contents of running program at time when it encounters a severe error or crashes

simple queuing mechanism (i.e. **round-robin** technique w/ FIFO queue) is inefficient b/c some not-running processes may be rdy to execute while others are blocked
↳ dispatcher would have to scan list for not-running, rdy process that's been in queue for longest
↳ use **multiple queues** so dispatcher can immediately pick right queue + go round-robin

5-state model:
↳ running: process is currently executing
↳ ready: process can be executed
↳ blocked / waiting: process that can't execute b/c it's waiting for smth
↳ new: new process that's been created but hasn't entered system yet
↳ exit: halted / aborted process

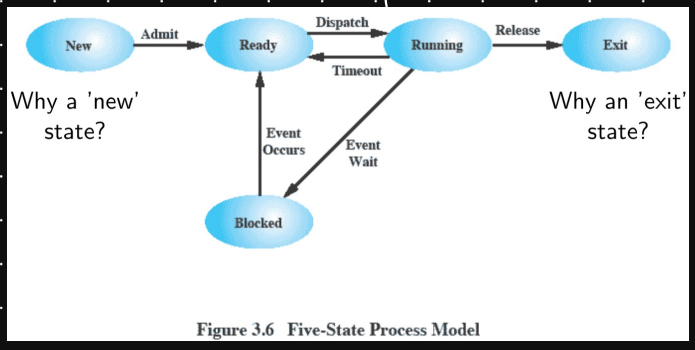


Figure 3.6 Five-State Process Model

↳ for **new state**, info concerning process that's needed by OS is in main mem but process itself (i.e. program code + space allocation for data) remains in secondary storage
  ◦ OS limits #processes in system b/c of performance + main mem limits
↳ for **exit state**, tables + other info from process are temporarily preserved so support programs can extract any needed info.
↳ **preemption** means a currently running process can be stopped by OS at any time

e.g.

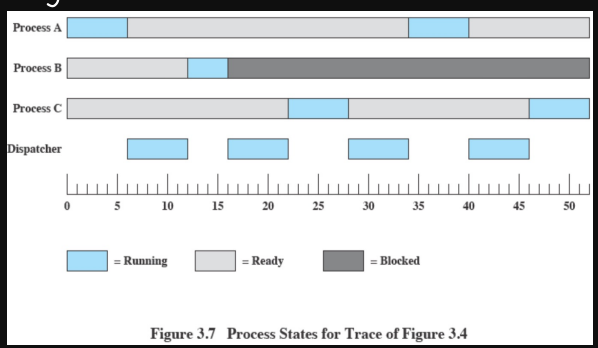
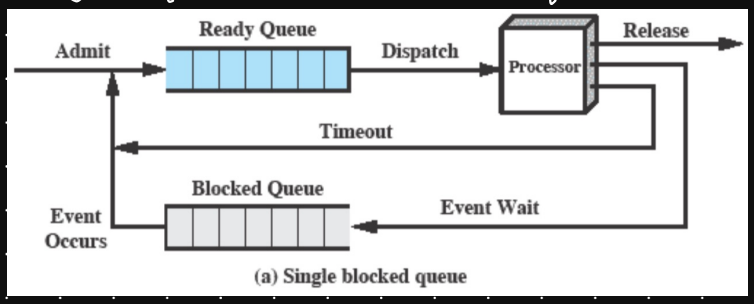


Figure 3.7 Process States for Trace of Figure 3.4

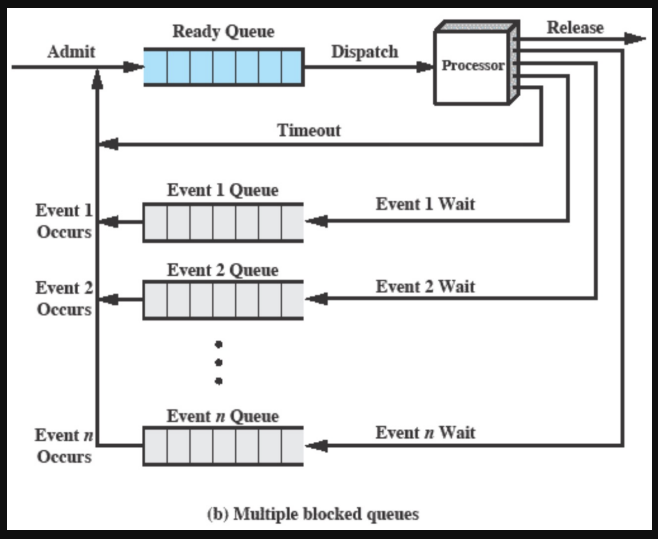
using 2 queues (single blocked queue):



(a) Single blocked queue

↳ problem is when event occurs, OS must scan entire blocked queue + search for processes waiting on that event

↳ soln is to have multiple blocked queues, 1 for each event:



(b) Multiple blocked queues

processor is faster than I/O so all processes could be waiting for I/O

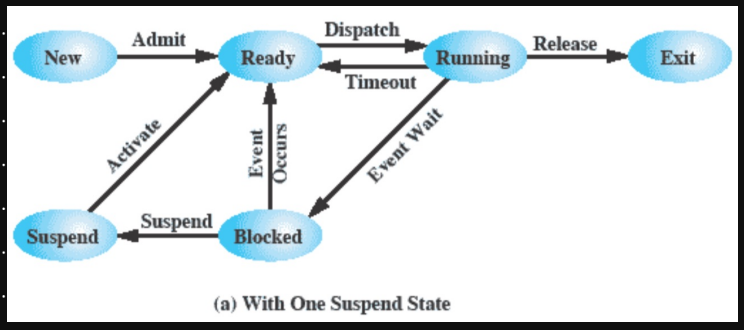
↳ soln is to admit more processes

↳ swap processes (i.e. moving part/all of process) from main mem to disk so we can free up space

↳ add 2 new states to show process is on disk:

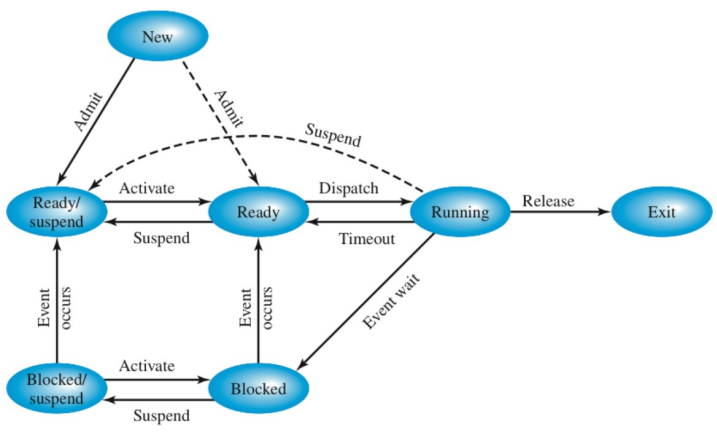
- blocked/suspend
- ready/suspend

1 suspend state:



(a) With One Suspend State

2 suspend state (better):



reasons for process suspension:
- ↳ swapping: OS needs to release sufficient main mem so process that's rdy to execute can be brought in
- ↳ other OS reason
  - bg/utility process
  - process that may cause problem
- ↳ interactive user request: user may want to debug or connect w/use of resource
- ↳ timing: periodic process may be suspended while waiting for next time interval
- ↳ parent process request: parent may want to examine/modify child process or coordinate activity of multiple children

MANAGING RESOURCES

OS is entity that manages use of system resources by processes
- ↳ e.g.

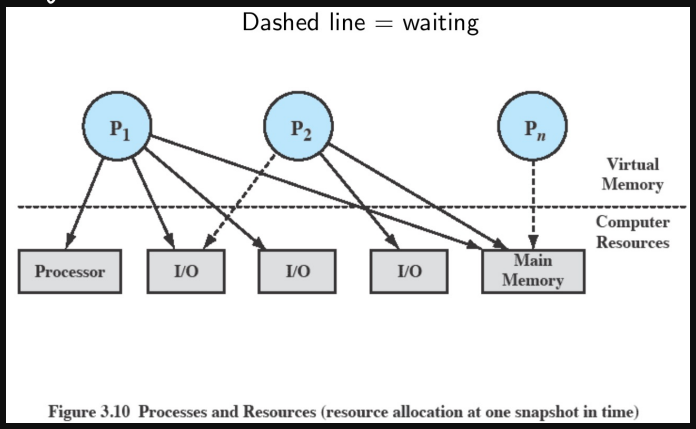


Figure 3.10 Processes and Resources (resource allocation at one snapshot in time)

OS must have info abt curr status of each process & resource
- ↳ tables are constructed for each entity that OS manages

mem tables are used to keep track of main (real) & secondary (virtual) mem
- ↳ must include info abt:

- allocation of main & secondary mem to processes
- protection attributes for access to shared mem regions
- how to manage virtual mem

- **I/O tables** used by OS to manage I/O devices
  - ↳ if I/O device is avail/assigned
  - ↳ status of I/O op
  - ↳ loc in main mem being used as source/destination of I/O transfer

e.g. /proc dir is virtual file system that allows access to kernel & process info as if they were regular files

- **file tables** provide info abt existence of files, loc on secondary mem, their curr status, & other attributes (e.g. rwxr-r-)
  - ↳ info sometimes maintained by file management system
- **process tables** contains info on:
  - ↳ where process is located in main mem
  - ↳ attributes in process img
  - ↳ program
  - ↳ data
  - ↳ stack
  - ↳ process control block

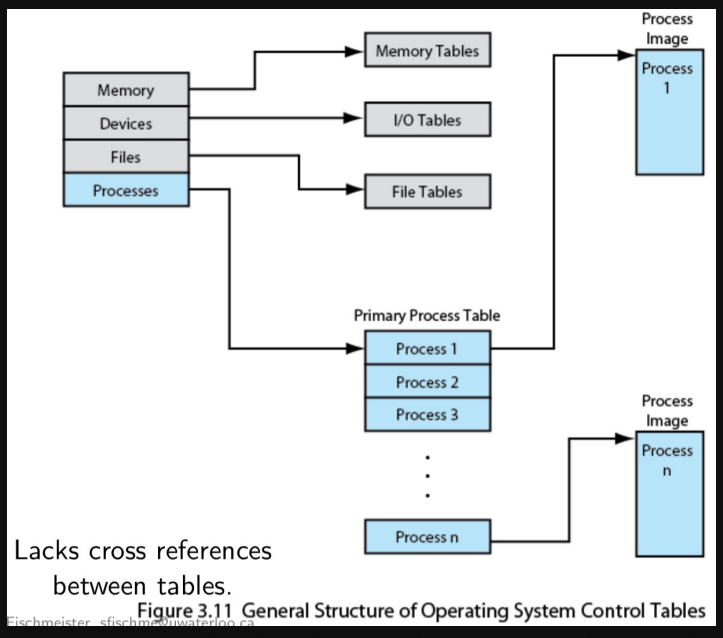


Figure 3.11 General Structure of Operating System Control Tables

- OS knows to create all these tables b/c when it's initialized, it has access to config data that define basic env (created outside OS)
- typical elmts of **process img**:
  - ↳ user data
    - e.g. program data, user stack area, programs that can't be modified
  - ↳ user program
  - ↳ system stack

- each process has 1+ LIFO system stacks for storing params + calling addresses for procedure + system calls

↳ process control block

process control block (PCB) contains:

↳ process identification: numeric ids that are unique to process
- process id
- parent process id
- user id

↳ processor state info
- user-visible registers can be referenced by ML executed by processor in user mode
  → typically 8-32, but might be low as 1
- control + status registers control op of processor
  → PC: contains address of next ins to be fetched
  → condition codes: result of most recent arithmetic / logical op (e.g. sign, 0, carry, equal, overflow)
  → status info: includes interrupt enabled / disabled flags, execution mode
  → program status word (PSW): contains status info, such as cond codes

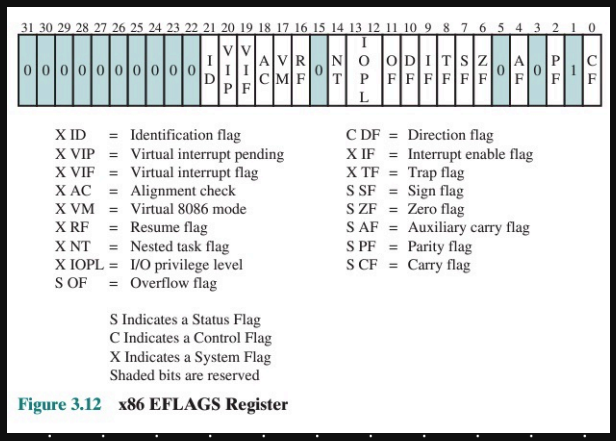


Figure 3.12 x86 EFLAGS Register

- stack ptrs

↳ process control info: meta info for handling processes
- scheduling + state info
  → process state (e.g. running, rdy, waiting)
  → priority (e.g. 0-255)
  → scheduling-related info (e.g. cpu time, time process has spent waiting)
  → event process is awaiting
- data structuring
  → linked lists for child processes, same priority processes, etc.
- interprocess communication
  → various flags, signals, + msgs

- process privileges
  - → mem access
  - → types of ins that can be executed
  - → use of system utilities + services
- mem management
  - → ptrs to segment + pg tables for virtual mem assigned to process
- resource ownership + utilization

2 modes of execution:

↳ **user mode** is less privileged
- user programs executed

↳ **kernel / system / control mode** is more privileged
- software has complete control of processor + all of its ins., registers, + mem

might want more modes so we create **protection / privilege rings** that allow for select privileged access to some things

SWITCHING PROCESSES

process creation steps:
1) assign unique process id
2) allocate space for process
3) init PCB
4) set up linkages
   - e.g. add new process to linked list for scheduling queue
5) create / expand other data structs
   - e.g. OS maintains accounting file

when to **switch processes**:

↳ clock interrupt (i.e. process has executed for max allowable time slice)

↳ I/O interrupt

↳ mem fault
- mem address is in virtual mem so it must be brought into main mem

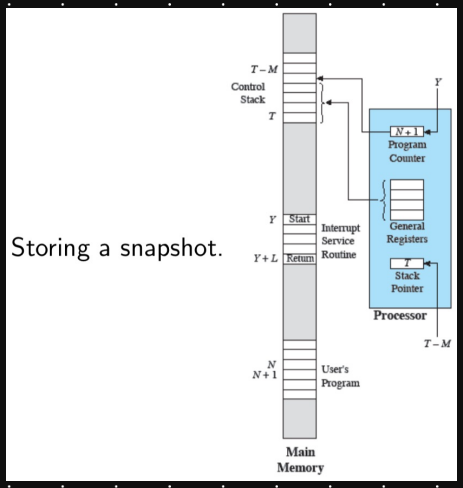
↳ **trap** (software-generated interrupt)
- error / exception occurred
- may cause process to be moved to Exit state
- used for debugging

↳ supervisor call
- switch to kernel process

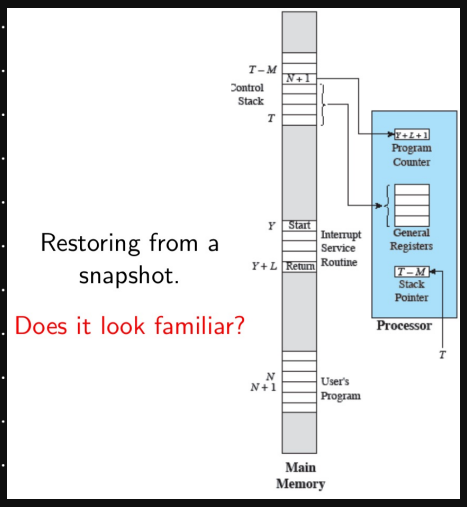
process switching steps:
1) save context of processor including PC + other registers
2) update PCB of process that's currently in Running state
3) move PCB to appropriate queue
4) select another process for execution

5) update PCB of selected process
6) update mem management data structs
7) restore context of selected process

switching processes is same as handling interrupts
↳ context switch:

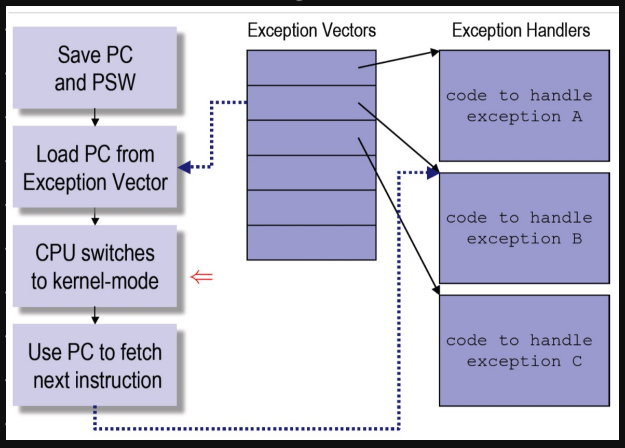


↳ resuming another process:

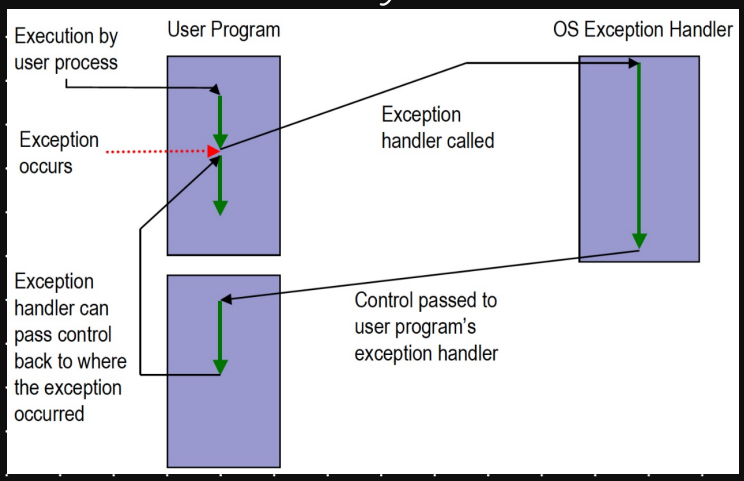


EXECUTION OF OPERATING SYSTEMS

exception handling:

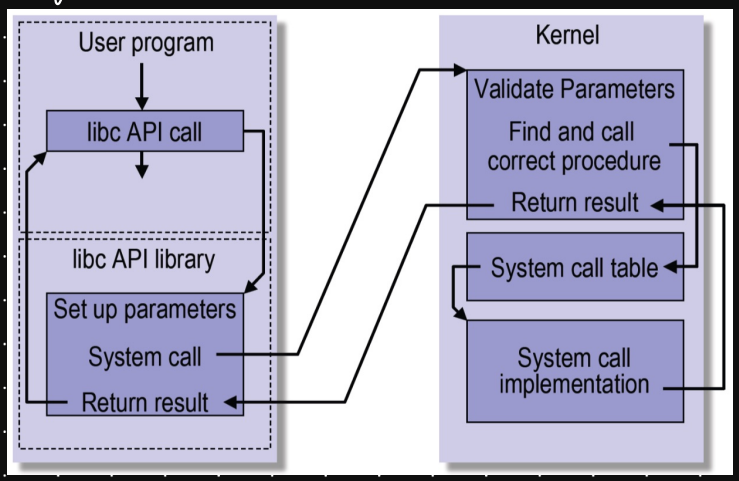


user exception handling



↳ similar to interrupts, except exceptions aren't necessarily made by I/O devices

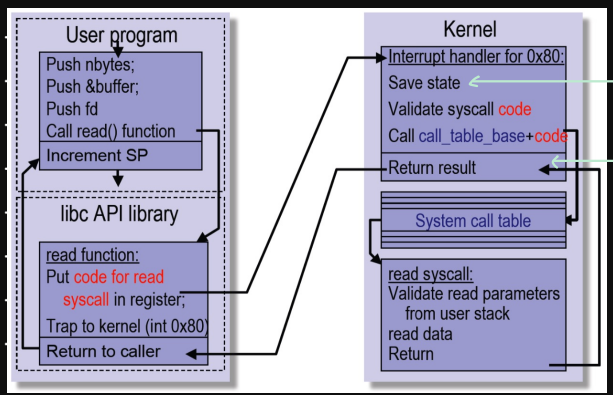
processing system call, which allow user programs to interact w/OS + request services / resources

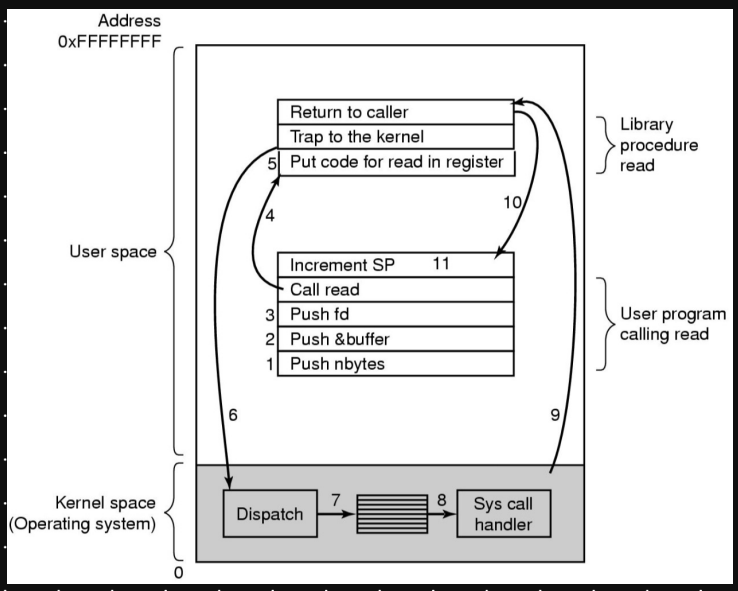


↳ e.g.

Call from C program:
count = read(fd, &buffer, length);

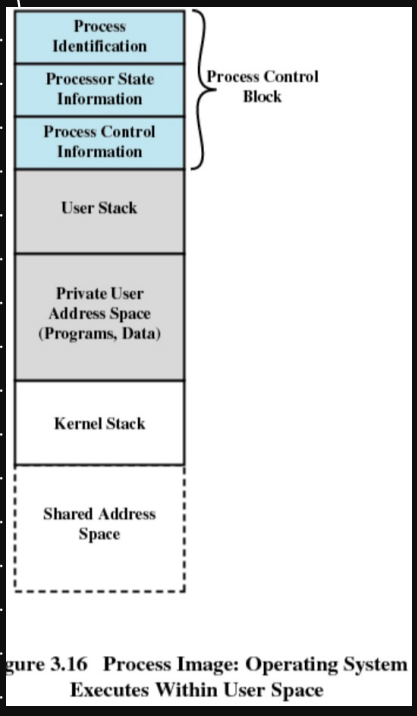
- read up to length bytes of data from from file
- descriptor fd into buffer, and returns the number of
- bytes actually read, or -1 if an error occurred.
- C library call for read() calls the read system call.





non-process kernel: execute kernel outside of all processes
- ↳ OS code executed as separate entity operating in privileged mode
- ↳ own mem + call stack
- ↳ processes only for user programs
- ↳ all OS calls are blocking

execution of kernel within user processes: OS is collection of routines + OS software executed in context of user process
- ↳ process executes in privileged mode
- ↳ context / mode switch when entering system call, but continue w/same process
- ↳



gure 3.16 Process Image: Operating System Executes Within User Space

THREADS, SMP, AND MICROKERNELS

THREADING

- process embodies 2 characteristics, which are treated independently by OS:
  - ↳ **resource ownership**: process includes virtual addr space to hold process img
  - ↳ **scheduling / execution**: follows execution path (i.e. **trace**) thru 1+ programs
    - trace may be interleaved w/ other processes
- 2 characteristics treated independently by OS
  - ↳ **thread / lightweight process**: unit of dispatching
  - ↳ **process / task**: unit of resource ownership
- **multithreading**: OS supports multiple threads of execution within single process
  - ↳ MS-DOS supports single thread
  - ↳ UNIX supports single thread but multiple user processes
  - ↳ Windows, Solaris, Linux, Mach, + OS/2 support multithreading
  - ↳

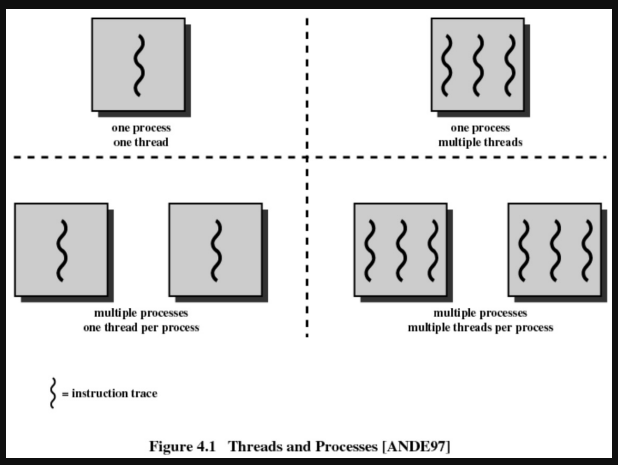


Figure 4.1 Threads and Processes [ANDE97]

- **processes** have virtual addr space which holds process img
  - ↳ protected access to processors, other processes, files, + I/O resources
- each **thread** within a process has:
  - ↳ execution state
  - ↳ saved thread context when not running
  - ↳ execution stack
  - ↳ per-thread static storage for local vars
  - ↳ access to mem + resources of process
    - shared w/ all other threads in process

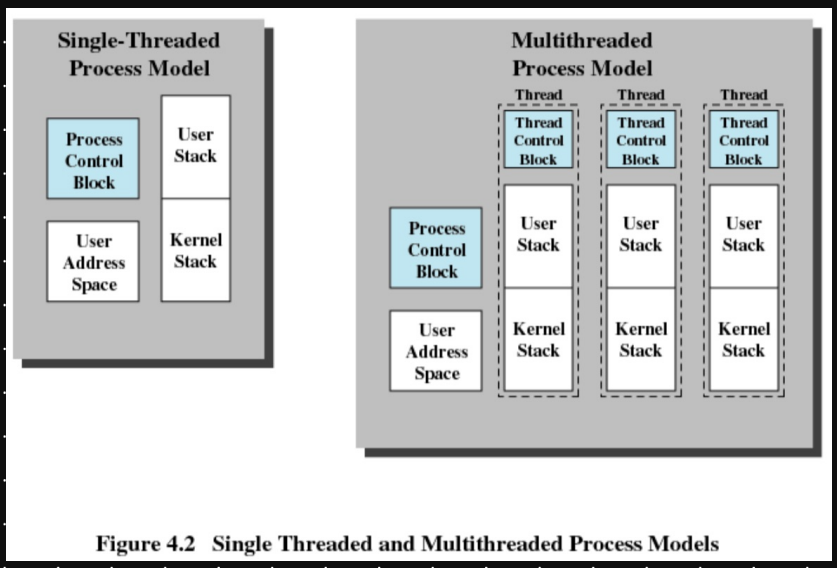


Figure 4.2 Single Threaded and Multithreaded Process Models

↳ there's multiple kernel stacks b/c each thread may want to switch to kernel mode + do diff kernel processes at diff times
- possible when we execute kernel within user process

benefits of threads:
↳ takes less time to create than process
- skip resource allocation thru kernel
- 10x faster

↳ less time to terminate than process b/c don't have to release resources thru kernel
↳ less time to switch btwn 2 threads
↳ more efficient communication btwn multiple execution entities b/c no kernel intervention needed
↳ threads within same process share mem

uses of threads in single-user multiprocessing system:
↳ foreground to bg work
- e.g. T1 handles sampling, T2 bg checks, T3 data processing

↳ asynch processing
- e.g. T1 computes everything, T2 gets/pushes data to resource

↳ speed of execution
- e.g. w/ multiple threads on multiple CPUs/cores, I/O blocking doesn't stop process + only 1 thread

↳ modular program struct
- programs that involve variety of activities + sources/destinations of I/O

suspending/terminating process → suspending/terminating all threads in it

4 basic ops associated w/ thread states:
↳ spawn
- thread can be spawned by new process or another thread in same process

↳ block

↳ unblock
↳ finish
- dealloc register context + stacks

e.g. remote procedure call (RPC) using threads:

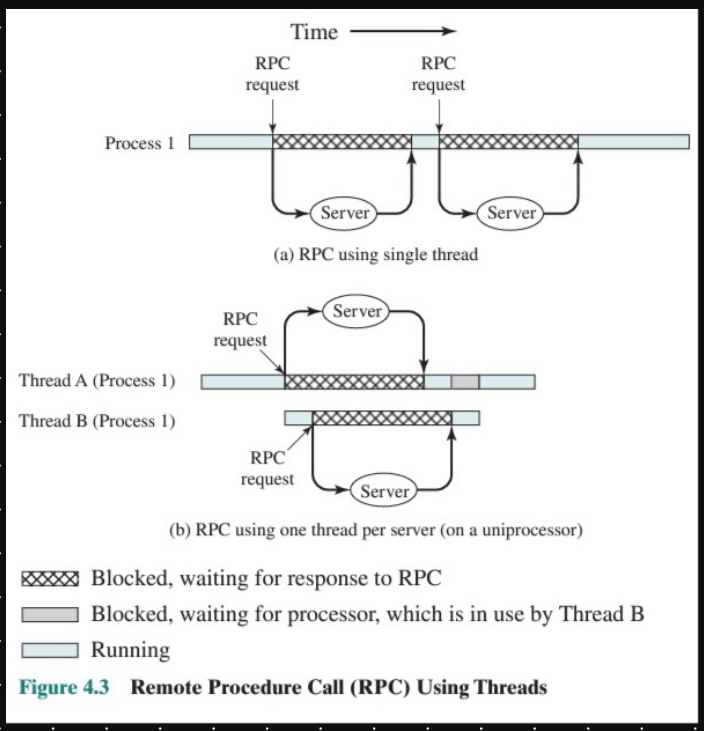


Figure 4.3 Remote Procedure Call (RPC) Using Threads

↳ assume independent fcn calls

e.g.

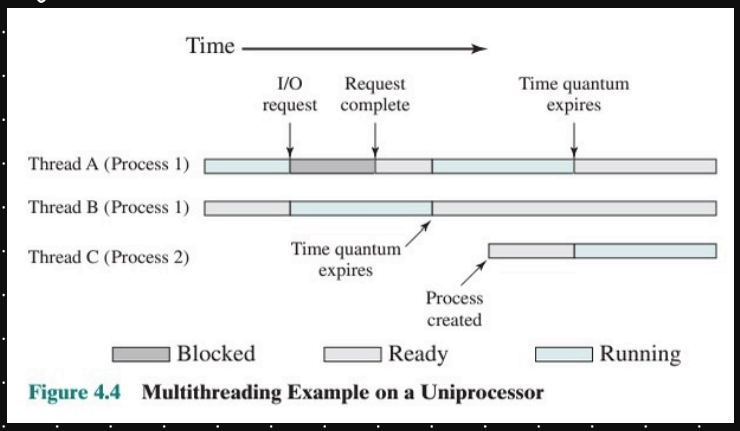
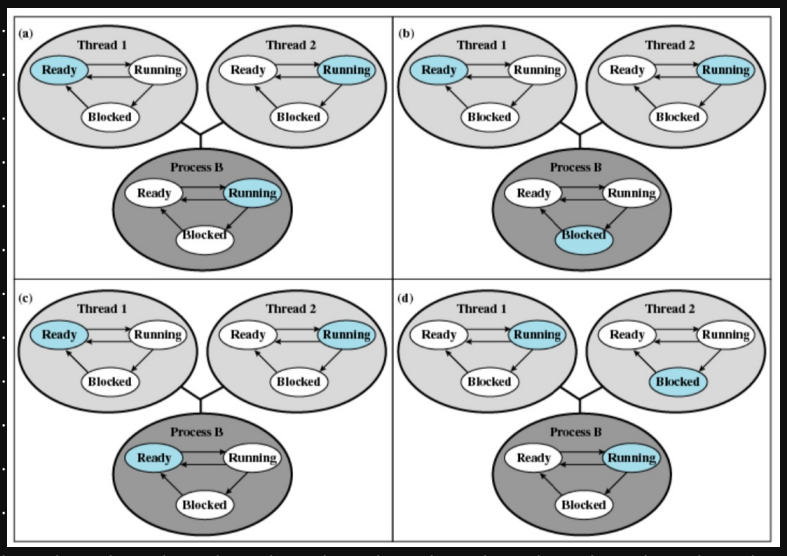


Figure 4.4 Multithreading Example on a Uniprocessor

user-lvl threads (ULTs): all thread management done by app + kernel unaware of threads' existences
↳ many threads in app
- app schedules execution

↳ 1 process in kernel
- kernel schedules processes

e.g.



↳ a) T2 is currently running b/c B is running.

↳ b) T2 made system-lvl blocking call

- control transfers to kernel
- OS only knows B so it blocks entire process B

↳ c) OS moves B to Ready state b/c B has finished its time slice so there's clock interrupt

↳ d) T2 needs T1 to perform action so T2 → Blocked & T1 → Running

- B remains Running thru out

kernel-lvl threads (KLTs): kernel maintains context info for process & its threads

↳ scheduling done on thread basis

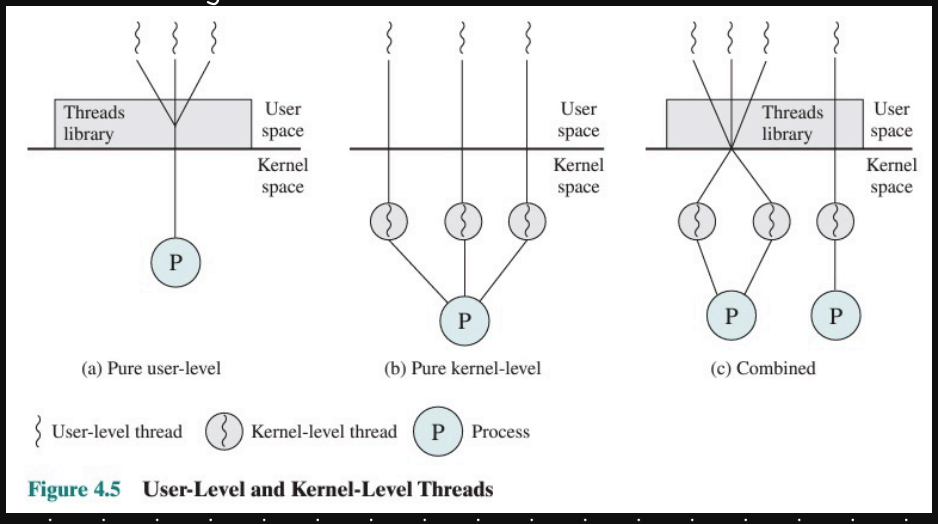


Figure 4.5 User-Level and Kernel-Level Threads

ULT pros:

↳ less switching overhead b/c we save 2 mode switches

↳ scheduling is app-specific

↳ ULT can run on any OS

KLT pros:

↳ OS calls block only thread + not entire process
  ◦ e.g. KLT is good approach for lots of I/O processes
↳ can schedule threads simultaneously on multiple processors

combining ULT + KLT:
↳ apps consist of multiple threads
↳ threads can be grouped to kernel threads
↳ kernel knows + schedules processes + threads

microkernel: type of OS kernel architecture where it's a small OS core that contains only essential core OS fcns
↳ many services in OS before become external subsystems
  ◦ device drivers
  ◦ file systems
  ◦ virtual mem manager
  ◦ windowing system
  ◦ security system
↳ e.g. L4 has only 7 system calls

monolithic kernel vs. microkernel

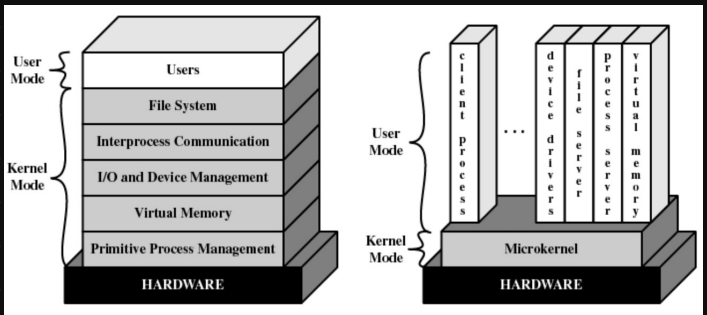


Figure 4.10 Kernel Architecture

benefits of microkernel:
↳ uniform interface on process' request
  ◦ no distinguishing btwn kernel + user-lvl services
  ◦ all services provided thru msg passing
↳ extensibility
  ◦ allows addition of new services
↳ flexibility
  ◦ new features added
  ◦ existing features removed
↳ portability
  ◦ changes needed to port system to new processor is changed only in microkernel + not other services
↳ reliability
  ◦ modular design

- microkernel allows for rigorous testing
- crashing module doesn't crash kernel

↳ distributed system support
- msgs sent w/o knowing target machine type
- opens up new app types

↳ obj-oriented OS
- components are objs w/ clearly defined interfaces that can be interconnected to form software

in **microkernel design**, mem management is low-lvl
↳ map each virtual pg to physical pg frame
↳ handles **pg faults** (i.e. virtual pg not currently in physical mem)
↳ grant/map/flush calls
↳ e.g.

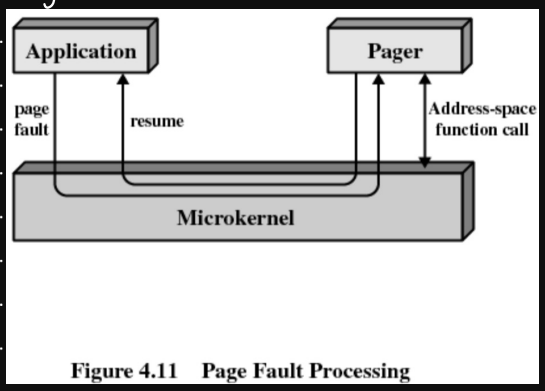


Figure 4.11 Page Fault Processing

for interprocess communication in microkernel, use msgs + ports
↳ msgs need to be copied
↳ remapping pgs may be faster

interrupts are msgs sent to processes
↳ microkernel doesn't know anything abt IRQ handling fcn

if we have N processors, our program is not N times faster
↳ **Amdahl's Law** is formula that calcs max speedup achievable by parallelizing computational task
- speedup = $\frac{\text{time to execute program on single processor}}{\text{time to execute program on N parallel processors}}$

$$\text{speedup} = \frac{1}{(1-f) + \frac{f}{N}}$$

→ $(1-f)$ is fraction of code that's inherently serial
→ f is fraction of code that infinitely parallelizable w/no scheduling overhead

↳ sources of serialization:
- critical sections in concurrent programming
- task dependencies
- sequential algos

apps that **benefit from multiprocessors**:
↳ multithreaded native apps

- small # highly threaded processes
  - ↳ multiprocess apps
    - many single-threaded processes
  - ↳ virtualized langs
    - e.g. Java lang supports multithreaded apps + JVM is multithreaded process that provides scheduling + mem management for Java apps
  - ↳ multi-instance apps
    - if multiple instances of app require some deg of isolation, virtualization tech provides each w/ own env

CONCURRENCY, MUTUAL EXCLUSION, + SYNCHRONIZATION

CONCURRENCY

- **requirements** for concurrency:
  - ↳ multiple apps so we can use multiprogramming
  - ↳ structured app (i.e. it can be set of concurrent programs)
  - ↳ OS struct is set of processes/threads
- **critical section**: portion of code in process that needs access to nonshareable resource
  - ↳ only 1 process may be in that critical section at a time
- **deadlock**: 2+ processes are unable to proceed b/c both waiting for each other to do smth
- **livelock**: 2+ processes continuously change state in response to changes in other process(es) w/o doing any useful work
  - ↳ e.g. 2 processes trying to access a shared resource at same time + they keep releasing + retrying lock so resource remains unavail.
- **mutual exclusion**: when 1 process is in critical section that accesses shared resources, no other processes may be in critical section accessing same resources
- **race condition**: multiple threads/processes read + write shared data item + final result depends on rltive timing of execution
- **starvation**: runnable process is overlooked indefinitely by scheduler so although it's able to proceed, it's nvr chosen
- **difficulties** of concurrency:
  - ↳ sharing global resources
  - ↳ OS managing allocation of resources optimally
  - ↳ difficult to locate programming errors

e.g.

Setup: two processes/two threads

```
void echo()
{
   chin = getchar();
   chout = chin;
   putchar(chout);
}
```

Thread P1	Thread P2
.	.
chin = getchar();	.
.	chin = getchar();
chout = chin;	chout = chin;
putchar(chout);	.
.	putchar(chout);
.	.

↳ P1 : enter "1" + P2 : enter "2"
- output : "2 2"

↳ "1 2" + "2 1" happen w/ processes
↳ "1 1" + "2 2" happen w/ threads b/c they share mem

OS **concerns** :
↳ keep track of various resources
↳ alloc / dealloc resources such as:
- processor time
- mem
- files
- I/O devices

↳ protect data + resources
↳ fcns of process must be independent of speed of execution of other concurrent processes

3 types of **process interactions**:
↳ unaware of each other
↳ indirectly aware of each other
- e.g. shared read / write access to doc

↳ directly aware of each other

Deg of Awareness	Rltnship	Influence of 1 Process on Other	Potential Control Problems
↳ unaware	↳ competition	↳ results of 1 process are indep of others' actions ↳ process timing may be affected	↳ mutual exclusion ↳ deadlock (renewable resource) ↳ starvation
↳ indirectly aware	↳ cooperation by sharing	↳ results of 1 process may depend on info from others ↳ process timing may be affected	↳ mutual exclusion ↳ deadlock (renewable resource) ↳ starvation ↳ data coherence
↳ directly aware	↳ coop by communication	↳ results of 1 process may depend on info from others ↳ process timing may be affected	↳ deadlock (consumable resource) ↳ starvation

3 **control problems** when dealing w/ competing processes
↳ mutual exclusion
- only 1 program is allowed at a time in its critical section
- e.g. 2 processes can't interact w/ printer at same time

↳ deadlock
↳ starvation

- facility providing support for **mutual exclusion** must meet requirements:
  - ↳ enforce mutual exclusion
    - i.e. only 1 process is allowed into its critical section for shared resource at a time
  - ↳ process that stops in non-critical section can't interfere w/ other processes while doing so
  - ↳ no deadlock / starvation for process requiring access to critical section
  - ↳ when critical section is unoccupied, process that requests entry must be permitted w/o delay
  - ↳ no assumptions abt rltive process speeds / # processors
  - ↳ process is in critical section for finite time
- **software approach** to mutual exclusion means processes are responsible for coordinating themselves w/o programming lang + OS support
  - ↳ prone to high processing overhead + bugs
  - ↳ e.g Dekker's algo

- **hardware approach** to mutual exclusion is to special machine ins in ISA
- can use **interrupt disabling** in uniprocessor system
  - ↳ concurrent processes can't have overlapping execution
  - ↳ process runs until it invokes OS service or it's interrupted
  - ↳ guarantee mutual exclusion by disabling interrupts
  - ↳ processor is limited in ability to interleave processes
  - ↳ doesn't work on multiprocessor systems b/c 1+ processes running at same time
- can use **special machine ins** performed **atomically** (in single ins cycle)
  - ↳ access to mem loc is blocked for any other ins
- e.g. **test + set**

```
boolean testset(int i) {
  if (i == 0) {
    i = 1;
    return true;
  }
  else {
    return false;
  }
}
```

  - ↳ i passed by ref
  - ↳ will not work in software b/c there might be race conds
    - e.g. 2 processes check that i==0 is T at same time + both enter critical section at same time
- e.g. **exchange**

```
void exchange(int register, int memory) {
  int temp;
  temp = memory;
  memory = register;
  register = temp;
}
```

↳ params passed by ref
↳ will not work in software either b/c race conds

e.g.

```
/* program mutualexclusion */
const int n = /* number of processes */;
int bolt;
void P(int i)
{
   while (true)
   {
      while (!testset (bolt))
          /* do nothing */;
      /* critical section */;
      bolt = 0;
      /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . . ,P(n));

}
```

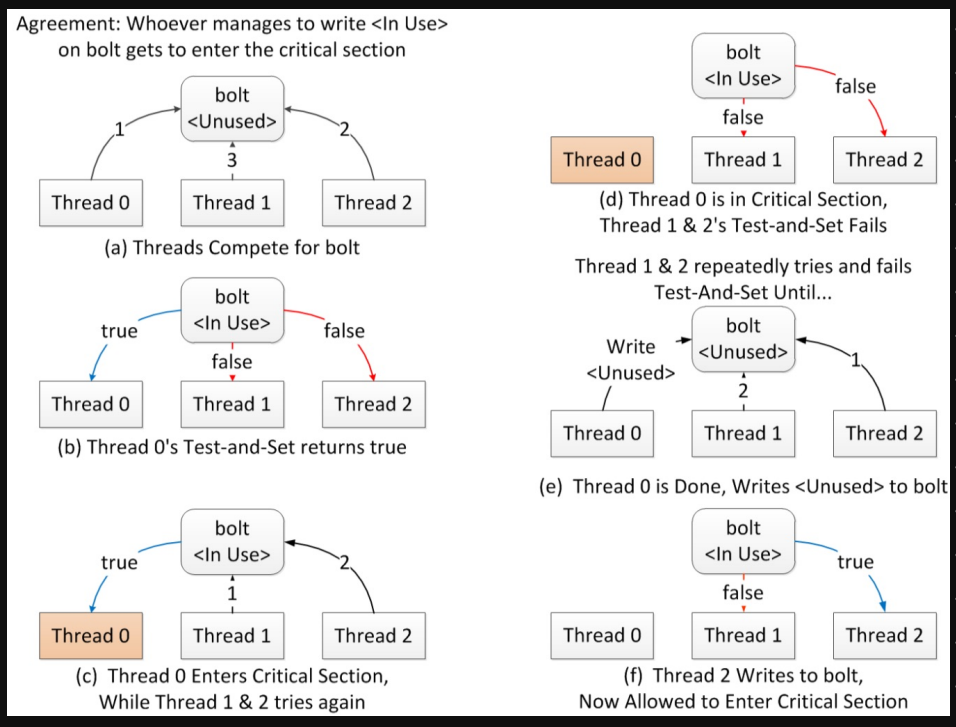
(a) Test and set instruction

```
/* program mutualexclusion */
int const n = /* number of processes**/;
int bolt;
void P(int i)
{
   int keyi;
   while (true)
   {
      keyi = 1;
      while (keyi != 0)
            exchange (keyi, bolt);
      /* critical section */;
      exchange (keyi, bolt);
      /* remainder */
   }
}
void main()
{
   bolt = 0;
   parbegin (P(1), P(2), . . ., P(n));
}
```

(b) Exchange instruction

Figure 5.2 Hardware Support for Mutual Exclusion

e.g.



advantages of hardware implementation:
↳ applicable to any #processes on single processor or multiple sharing main mem
↳ simple + easy to verify
↳ can support multiple critical sections
  • each is defined by own var

disadvantages of hardware implementation:
↳ busy waiting: while process waits for access, it consumes processor time

↳ possible starvation: selection of waiting process after critical section is freed is arbitrary
↳ possible deadlock
- e.g. P1 in critical section but interrupted to give processor to P2 (w/ higher priority) + P2 tries to use same resource P1 is using
  → infinite busy waiting loop b/c P1 is nvr dispatched since it's lower priority than P2

ADVANCED CONCURRENCY CONTROL MECHANISMS

OS + programming lang mechanisms can provide concurrency

semaphore: int val used to signal among processes
↳ if process is waiting for signal, it's suspended until that signal's sent
↳ may be init to non -ve #
↳ wait op dec semaphore val
↳ signal op inc semaphore val

semaphore ops:
↳ init w/ non -ve val
↳ semWait() dec semaphore val
↳ semSignal() inc semaphore val
↳ no other way to access semaphore
↳ any process can call semWait() + semSignal()
↳ for mutex, only process w/ lock can release it

properties of semaphore use:
↳ no way to know whether semWait() will block or not
↳ semSignal() may wake up a process
- OS doesn't have to make scheduling decision right after
↳ don't know whether semSignal() wakes up process
- don't know # processes waiting for signal

e.g. semaphore primitives

```
struct semaphore {
      int count;
      queueType queue;
}

void semWait(semaphore s)
{
      s.count--;
      if (s.count < 0)
      {
            place this process in s.queue;
            block this process
      }
}
void semSignal(semaphore s)
{
      s.count++;
      if (s.count <= 0)
      {
            remove a process P from s.queue;
            place process P on ready list;
      }
}
```

e.g. binary semaphore primitives

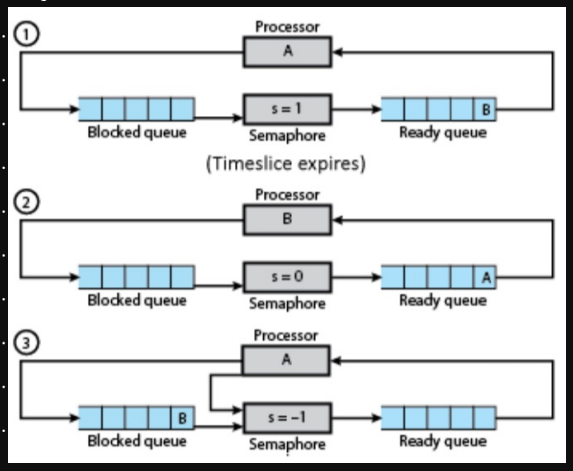
```
struct binary_semaphore {
     enum {zero, one} value;
     queueType queue;
};

void semWaitB(binary_semaphore s)
{
     if (s.value ==  1)
          s.value = 0;
     else
          {
               place this process in s.queue;
               block this process;
          }
}
void semSignalB(semaphore s)
{
     if (s.queue.is_empty())
          s.value = 1;
     else
     {
          remove a process P from s.queue;
          place process P on ready list;
     }
}
```

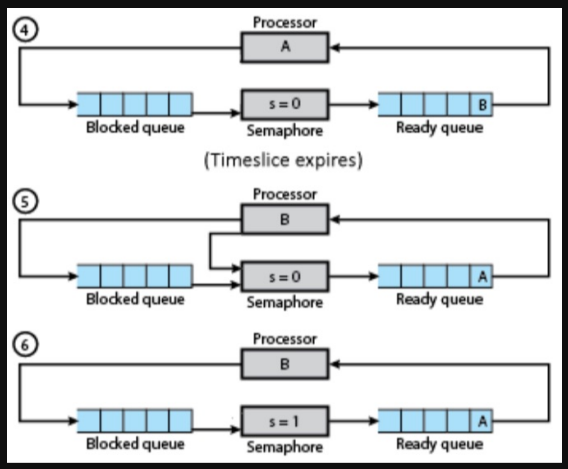
always protect inc/dec of semaphore

e.g. mutual exclusion using semaphores

```
/* program mutualexclusion */
const int n = /* number of processes  */;
semaphore s = 1;
void P(int i)
{
     while (true)
     {
          semWait(s);
          /* critical section   */;
          semSignal(s);
          /* remainder   */;
     }
}
void main()
{
     parbegin (P(1), P(2), . . ., P(n));
}
```

e.g.





↳ 1 → 2 : A calls semWait() + goes into critical section, then its timeslice expires so B gets processor

↳ 2 → 3 : B calls semWait() + gets blocked so A gets processor again

↳ 3→4: A finishes critical section + calls semSignal() so B gets moved to Ready queue
↳ 4→5: A's timeslice expires so B gets processor + enters critical section
↳ 5→6: B finishes critical section + calls semSignal() so s=1 again

to implement semaphores, need int val, queue, semWait fcn, + semSignal fcn
semaphores block when its val < 0
↳ semaphores can be -ve
init semaphore # indicates how many processes can be in critical section at same time
curr semaphore val:
↳ > 0: can allow processes in (max #processes currently allowed)
↳ = 0: critical section full
↳ < 0: processes are waiting to enter (# processes currently blocked)

producer / consumer problem is common concurrency issue
↳ 1+ producers are generating data + placing into buffer
↳ 1 consumer taking items out of buffer 1 at a time
↳ only 1 producer / consumer may access buffer at any time

e.g. infinite buffer
↳ producer:

```
while (true) {
   /* Produce item v */
   b[in] = v;
   in++;
}
```

↳ consumer:

```
while (true) {
   while (in < out) /* Do nothing */;
   w = b[out];
   out++;
   /* Consume item w */
}
```

↳

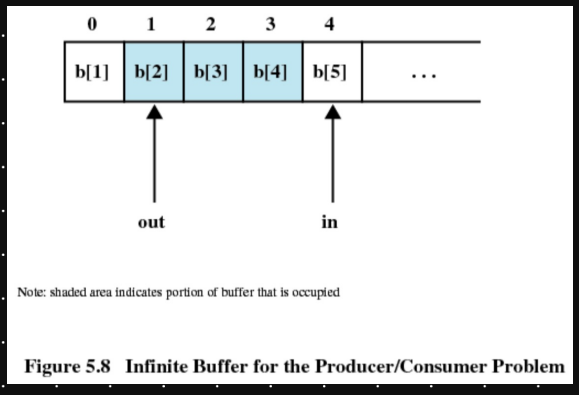


Figure 5.8 Infinite Buffer for the Producer/Consumer Problem

e.g. circular buffer

↳ producer:

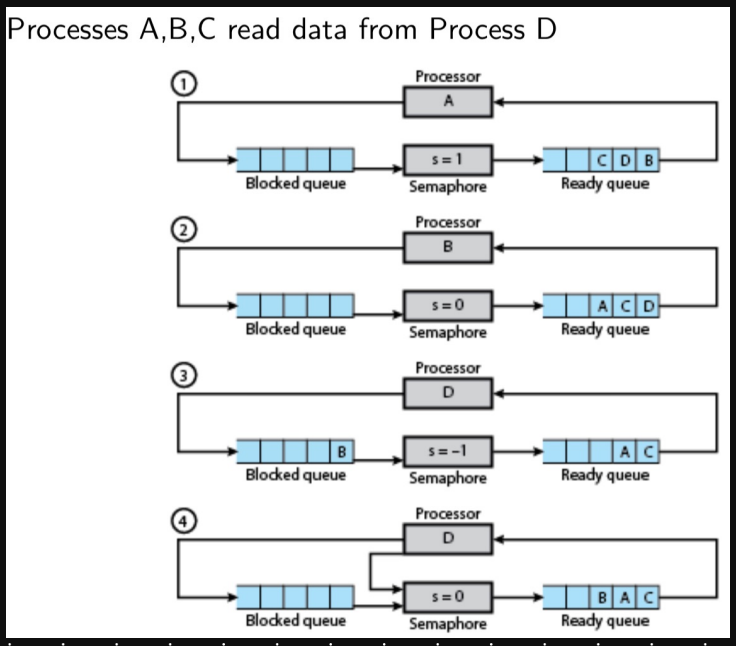
```
while (true) {
  /* produce item v */
  while ((in + 1) % n == out) /* do nothing */;
  b[in] = v;
  in = (in + 1) % n
}
```

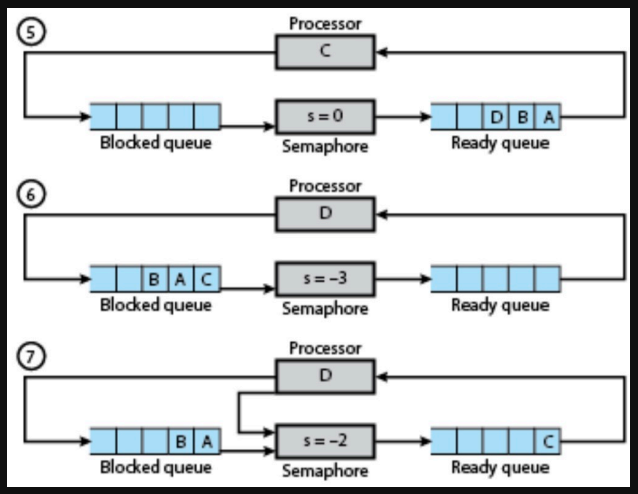
↳ consumer:

```
while (true) {
  while (in == out) /* do nothing */;
  w = b[out];
  out = (out + 1) % n;
  /* consume item w */
}
```

e.g.

Processes A,B,C read data from Process D





Producer: D
Consumers: A, B, C

↳ 1→2: A sends semWait() + enters critical section, then moved to Ready queue
↳ 2→3: B sends semWait() + gets blocked b/c $s < 0$
↳ 3→4: D sends semSignal() + s inc 1
↳ 4→5: D goes to Ready queue + C gets processor
↳ 5→6: A, B, C all send semWait() + get blocked
↳ 6→7: D sends semSignal() + s inc 1

e.g.

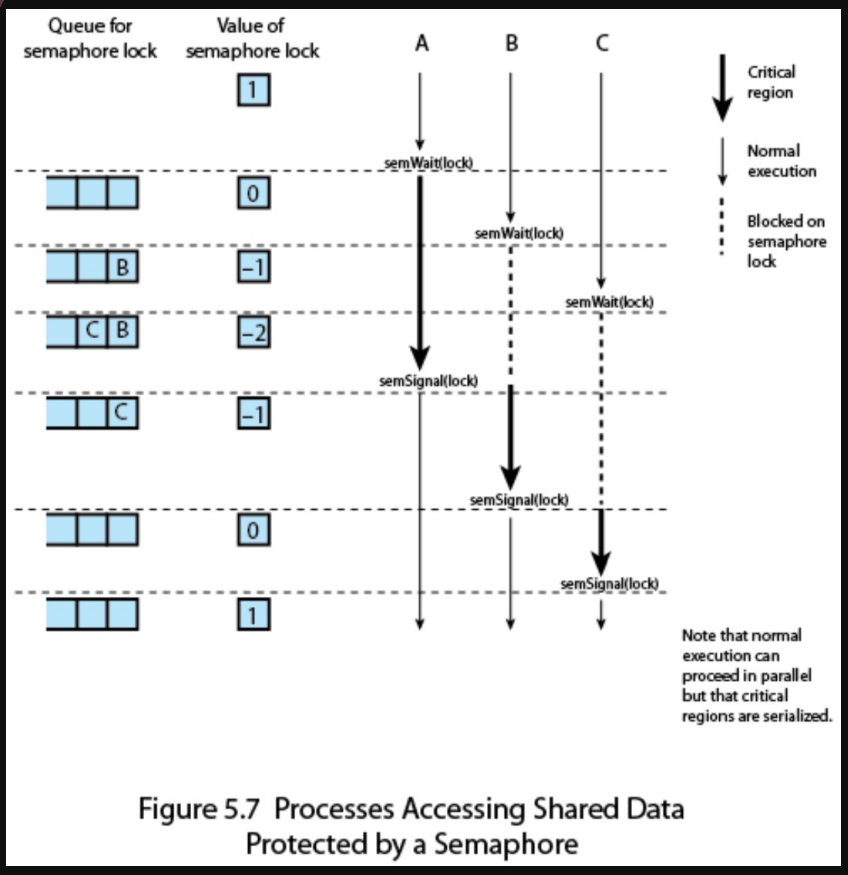


Figure 5.7 Processes Accessing Shared Data Protected by a Semaphore

e.g. producer/consumer problem w/circular buffer

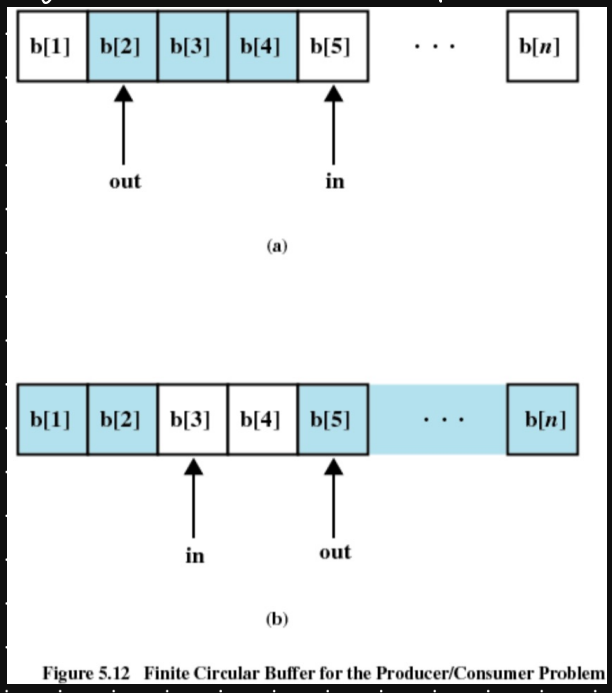


Figure 5.12 Finite Circular Buffer for the Producer/Consumer Problem

e g incorrect soln to infinite buffer producer/consumer

```
/* program producerconsumer */
int n;
binary_semaphore s = 1;
binary_semaphore delay = 0;
```

```
void producer()
{
      while (true)
      {
            produce();
            semWaitB(s);
            append();
            n++;
            if (n==1)
                  semSignalB(delay);
            semSignalB(s);
      }
}
void consumer()
{
      semWaitB(delay);
      while (true)
      {
            semWaitB(s);
            take();
            n--;
            semSignalB(s);
            consume();
            if (n==0)
                  semWaitB(delay);
      }
}
void main()
{
      n = 0;
      parbegin (producer, consumer);
}
```

↳ this could happen:

	Producer	Consumer	s	n	Delay
1			1	0	0
2	semWaitB(s)		0	0	0
3	n++		0	1	0
4	**if** (n==1) (semSignalB(delay))		0	1	1
5	semSignalB(s)		1	1	1
6		semWaitB(delay)	1	1	0
7		semWaitB(s)	0	1	0
8		n--	0	0	0
9		semSignalB(s)	1	0	0
10	semWaitB(s)		0	0	0
11	n++		0	1	0
12	**if** (n==1) (semSignalB(delay))		0	1	1
13	semSignalB(s)		1	1	1
14		**if** (n==0) (semWaitB(delay))	1	1	1
15		semWaitB(s)	0	1	1
16		n--	0	0	1
17		semSignalB(s)	1	0	1
18		**if** (n==0) (semWaitB(delay))	1	0	0
19		semWaitB(s)	0	0	0
20		n--	0	–1	0
21		semiSignlaB(s)	1	–1	0

NOTE: White areas represent the critical section controlled by semaphore s.

↳ correct soln: use local var to track # items in buffer

```
/* program producerconsumer */
int n;
binary_semaphore s = 1;
binary_semaphore delay = 0;
```

```
void producer()
{
     while (true)
     {
          produce();
          semWaitB(s);
          append();
          n++;
          if (n==1) semSignalB(delay);
          semSignalB(s);
     }
}
void consumer()
{
     int m; /* a local variable */
     semWaitB(delay);
     while (true)
     {
          semWaitB(s);
          take();
          n--;
          m = n;
          semSignalB(s);
          consume();
          if (m==0) semWaitB(delay);
     }
}
void main()
{
     n = 0;
     parbegin (producer, consumer);
}
```

Figure 5.10 A Correct Solution to the Infinite-Buffer Producer/Consumer Problem Using Binary Semaphores

e.g. soln to infinite buffer producer/consumer problem using semaphores

```
/* program  producerconsumer */
semaphore n = 0;
semaphore s = 1;
void producer()
{
     while (true)
     {
          produce();
          semWait(s);
          append();
          semSignal(s);
          semSignal(n);
     }
}
void consumer()
{
     while (true)
     {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

↳ n : # items in buffer

↳ s : controls access to buffer

e.g. soln to finite buffer producer/consumer problem using semaphores

↳ e : # empty spots in buffer

```
/* program boundedbuffer */
const int sizeofbuffer = /* buffer size */;
semaphore s = 1;
semaphore n= 0;
semaphore e= sizeofbuffer;
void producer()
{
     while (true)
     {
          produce();
          semWait(e);
          semWait(s);
          append();
          semSignal(s);
          semSignal(n)
     }
}
void consumer()
{
     while (true)
     {
          semWait(n);
          semWait(s);
          take();
          semSignal(s);
          semSignal(e);
          consume();
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Figure 5.13 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Semaphores

↳ deadlocks are prevented b/c n + e are nvr 0 at the same time

MONITORS

monitor: software module consisting of 1+ procedures, init sequence, + local data

chief characteristics:

↳ local data vars accessible only by monitor
- shared data is safe

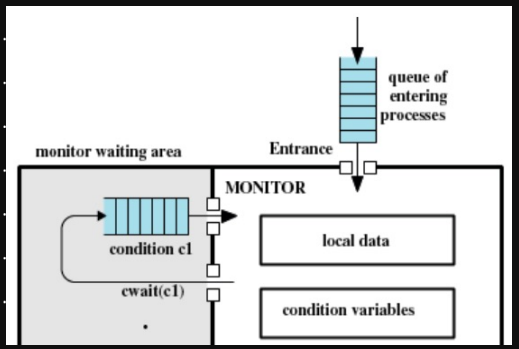
↳ process can only enter monitor by invoking 1 of its procedures
- controlled entry

↳ only 1 process may be executing in monitor at a time
- mutex

uses condition vars for signaling

unused signals are lost, unlike semaphores

struct of monitor:



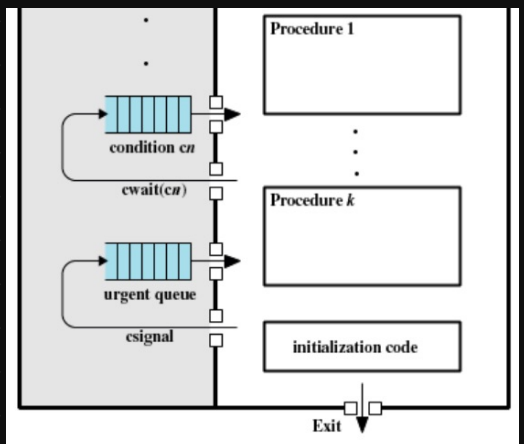


Figure 5.15 Structure of a Monitor

e.g. finite buffer producer/consumer soln w/ monitor

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];                                    /* space for N items */
int nextin, nextout;                                  /* buffer pointers */
int count;                                        /* number of items in buffer */
cond notfull, notempty;          /* condition variables for synchronization */

void append (char x)
{
     if (count == N)
        cwait(notfull);                     /* buffer is full; avoid overflow */
     buffer[nextin] = x;
     nextin = (nextin + 1) % N;
     count++;
     /* one more item in buffer */
     csignal(notempty);                        /* resume any waiting consumer */
}
void take (char x)
{
     if (count == 0)
        cwait(notempty);                  /* buffer is empty; avoid underflow */
     x = buffer[nextout];
     nextout = (nextout + 1) % N;
     count--;                                    /* one fewer item in buffer */
     csignal(notfull);                         /* resume any waiting producer */
}
{                                                              /* monitor body */
     nextin = 0; nextout = 0; count = 0;          /* buffer initially empty */
}
```

```
void producer()
char x;
{
     while (true)
     {
     produce(x);
     append(x);
     }
}
void consumer()
{
     char x;
     while (true)
     {
       take(x);
       consume(x);
     }
}
void main()
{
     parbegin (producer, consumer);
}
```

Figure 5.16 A Solution to the Bounded-Buffer Producer/Consumer Problem Using a Monitor

↳ process exits monitor immediately after executing csignal fcn

↳ if csignal doesn't occur at end of process, it must be blocked + placed in queue

- set in urgent queue to give it precedence over newly entering processes

e.g.

```
void append (char x)
{
    while(count == N)
        cwait(notfull);              /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;                                /* one more item in buffer */
    cnotify(notempty);                   /* notify any waiting consumer */
}

void take (char x)
{
    while(count == 0)
    cwait(notempty);                  /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;                                /* one fewer item in buffer */
    cnotify(notfull);                    /* notify any waiting producer */
}
```

Figure 5.17 Bounded Buffer Monitor Code for Mesa Monitor

↳ cnotify(x) causes x cond queue to be notified, but process that sent signal still runs

↳ process at start of x cond queue will resume when monitor is avail in future
- use while loop to recheck cond b/c we can't guarantee another process won't enter monitor before waiting process

MESSAGE PASSING

msg passing enforces mutual exclusion by exchanging info

↳ 2 fundamental requirements are synchronization + communication

```
send(destination, message)
receive(source, message)
```

synchronization combos (i.e. sender + receiver may/may not be blocking + waiting for msg)

↳ blocking send, blocking receive: both sender + receiver are blocked until msg is fully delivered
- aka rendezvous

↳ nonblocking send, blocking receive: sender continues on + receiver is blocked until requested msg arrives

↳ nonblocking send, nonblocking receive: neither party required to wait

addressing specifies processes in send + receive primitives (i.e source/destination)

↳ direct:
- send primitive has specific id of destination process
- receive primitive knows ahead of time from which process a msg is expected or use source param to return val to confirm when receive op has been performed

↳ indirect:
- msgs sent to shared data struct consisting of mailboxes (i.e. queues)
- 1 process sends msg to mailbox + other process picks it up from there

e.g.

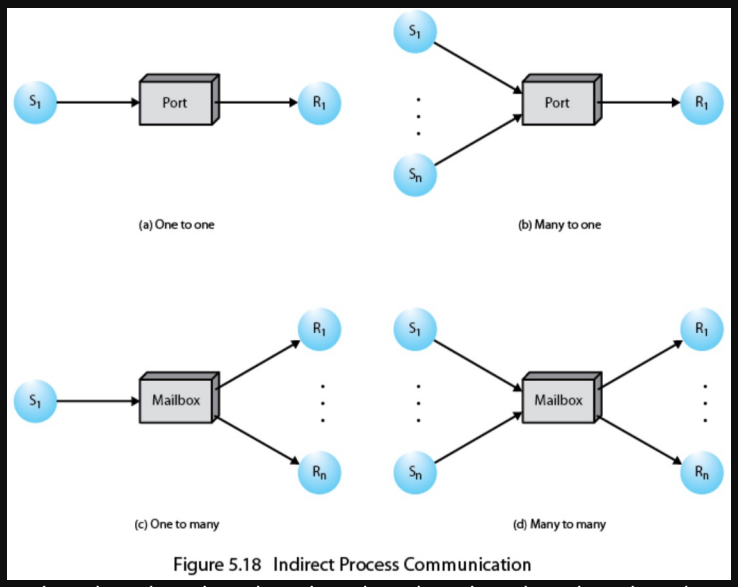


Figure 5.18 Indirect Process Communication

↳ port is when mailbox is specifically assigned to 1 receiver

general msg format:

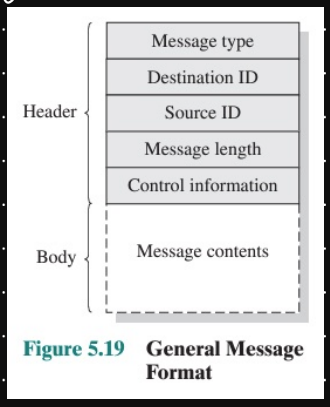


Figure 5.19 General Message Format

e.g. mutual exclusion w/ msgs

```
/* program mutualexclusion */
const int n = /* number of processes  */;
void P(int i)
{
    message msg;
    while (true)
    {
     receive (mutex, msg);
     /* critical section   */;
     send (mutex, msg);
     /* remainder   */;
    }
}
void main()
{
    create_mailbox (mutex);
    send (mutex, null);
    parbegin (P(1), P(2), . . ., P(n));
}
```

Figure 5.20 Mutual Exclusion Using Messages

↳ must init mailbox w/ token(s) or else all receivers will be forever blocked

e.g. finite buffer producer/consumer soln w/ msgs

```
const int
    capacity = /* buffering capacity */ ;
    null =/* empty message */ ;
int i;
void producer()
{   message pmsg;
    while (true)
    {
     receive (mayproduce, pmsg);
     pmsg = produce();
     send (mayconsume, pmsg);
    }
}
void consumer()
{   message cmsg;
    while (true)
    {
     receive (mayconsume, cmsg);
     consume (cmsg);
     send (mayproduce, null);
    }
}

void main()
{
    create_mailbox (mayproduce);
    create_mailbox (mayconsume);
    for (int i = 1; i <= capacity; i++)
        send (mayproduce, null);
    parbegin (producer, consumer);
}
```

Figure 5.21 A Solution to the Bounded-Buffer Producer/Consumer Problem Using Messages

↳ must have separate mailboxes for producers + consumers

READER/WRITER PROBLEM

reader/writer problem is when:

↳ any # readers may simultaneously read file

↳ only 1 writer may write to it at a time

↳ while writer is writing to file, no reader may read it

e.g. soln w/ semaphores + readers have priority

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
    while (true)
    {
     semWait (x);
     readcount++;
     if (readcount == 1)
          semWait (wsem);
     semSignal (x);
     READUNIT();
     semWait (x);
     readcount--;
     if (readcount == 0)
          semSignal (wsem);
     semSignal (x);
    }
 }
void writer()
{
    while (true)
    {
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
    }
}

void main()
{
    readcount = 0;
    parbegin (reader, writer);
    }
```

Figure 5.22 A Solution to the Readers/Writers Problem Using Semaphores: Readers Have Priority

↳ x protects readcount var

e.g. soln w/ semaphores r writers have priority

```
/*program readersandwriters*/
int  readcount, writecount;
semaphore x = 1, y = 1, z = 1, wsem = 1, rsem = 1;
void reader()
{
    while (true)
    {
     semWait (z);
          semWait (rsem);
               semWait (x);
                    readcount++;
                    if (readcount == 1)
                         semWait (wsem);
               semSignal (x);
          semSignal (rsem);
     semSignal (z);
     READUNIT();
     semWait (x);
          readcount--;
          if (readcount == 0)
               semSignal (wsem);
     semSignal (x);
    }
}
void writer ()
{
    while (true)
    {
     semWait (y);
          writecount++;
          if (writecount == 1)
               semWait (rsem);
     semSignal (y);
     semWait (wsem);
     WRITEUNIT();
     semSignal (wsem);
     semWait (y);
          writecount--;
          if (writecount == 0)
               semSignal (rsem);
     semSignal (y);
    }
}
void main()
{
    readcount = writecount = 0;
    parbegin (reader, writer);
}
```

↳ x protects readcount
↳ y protects writecount
↳ rsem stops all readers while there's at least 1 writer desiring access
↳ writecount var controls setting of rsem
↳ only 1 reader can queue up on rsem so rest of them queue up on z
↳

Table 5.6 State of the Process Queues for Program of Figure 5.23

Readers only in the system	• *wsem* set • no queues
Writers only in the system	• *wsem* and *rsem* set • writers queue on *wsem*
Both readers and writers with read first	• *wsem* set by reader • *rsem* set by writer • all writers queue on *wsem* • one reader queues on *rsem* • other readers queue on *z*
Both readers and writers with write first	• *wsem* set by writer • *rsem* set by writer • writers queue on *wsem* • one reader queues on *rsem* • other readers queue on *z*

tip: semaphores for capacity management should always be outside atomic buffer access
↳ avoid deadlock

CONCURRENCY, DEADLOCK, AND STARVATION

DEADLOCKS

- a **deadlock** is a permanent blocking of a set of processes that either compete for system resources or communicate w/ each other
  - ↳ no efficient soln in general case
  - ↳ involves conflicting needs for resources by 2+ processes
- e.g. **joint progress diagram** that shows deadlock

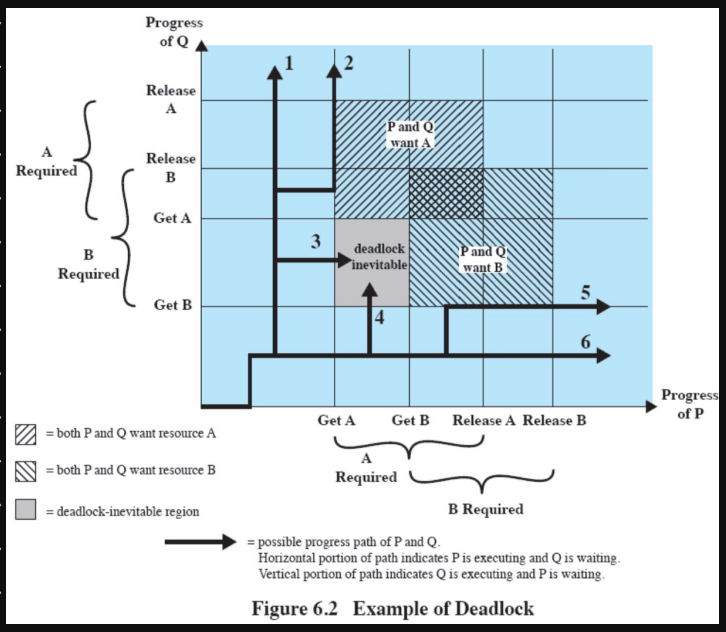


Figure 6.2 Example of Deadlock

Process P	Process Q
• • •	• • •
Get A	Get B
• • •	• • •
Get B	Get A
• • •	• • •
Release A	Release B
• • •	• • •
Release B	Release A
• • •	• • •

1) Q gets B then A, then releases B then A. When P resumes, it'll get A + B.
2) Q gets B then A. P is blocked on request for A. Q releases B then A. P resumes execution + it'll get A + B.
3) Q gets B, then P gets A. Deadlock is inevitable b/c Q will block on A + P will block on B.
4) P gets A, then Q gets B. Deadlock is inevitable b/c Q will block on A + P will block on B.
5) P gets A then B. Q is blocked on B. P releases A then B. Q resumes execution + it'll get A + B.
6) P gets A then B, then releases A then B. When Q resumes, it'll get A + B.

- ↳ gray area is **fatal region** b/c if execution path enters it, deadlock is inevitable
- e.g. deadlock avoidance by having P not need both resources at same time

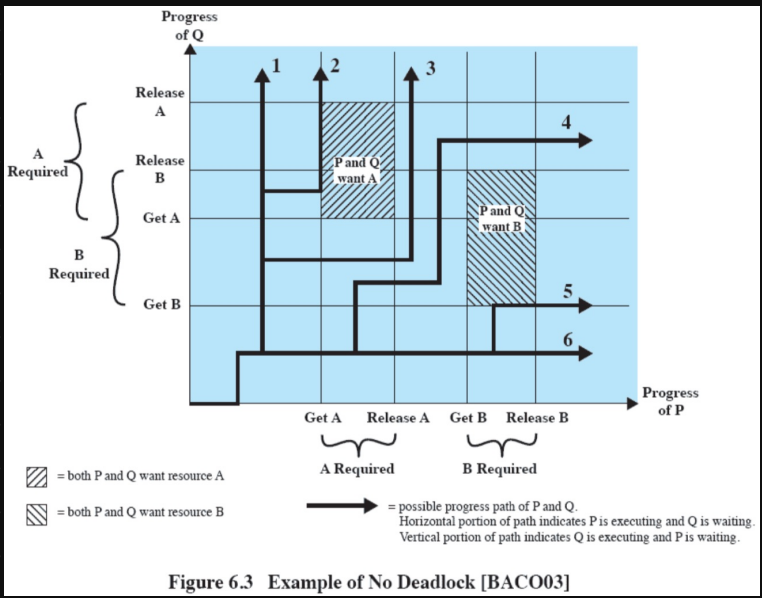


Figure 6.3 Example of No Deadlock [BACO03]

Process P	Process Q
• • •	• • •
Get A	Get B
• • •	• • •
Release A	Get A
• • •	• • •
Get B	Release B
• • •	• • •
Release B	Release A
• • •	• • •

reusable resources can be safely used by only 1 process at a time & aren't depleted after use

↳ e.g. processors, I/O channels, main & secondary mem, devices, & data structs (e.g. files, dbs, & semaphores)

↳ deadlock occurs if each process holds 1 resource & requests the other (assuming only 1 resource elmt exists)

- e.g.

Process P		Process Q	
Step	Action	Step	Action
p_0	Request (D)	q_0	Request (T)
p_1	Lock (D)	q_1	Lock (T)
p_2	Request (T)	q_2	Request (D)
p_3	Lock (T)	q_3	Lock (D)
p_4	Perform function	q_4	Perform function
p_5	Unlock (D)	q_5	Unlock (T)
p_6	Unlock (T)	q_6	Unlock (D)

Figure 6.4 Example of Two Processes Competing for Reusable Resources

→ deadlock can occur if processes are interleaved like $p_0 p_1 q_0 q_1 p_2 q_2$

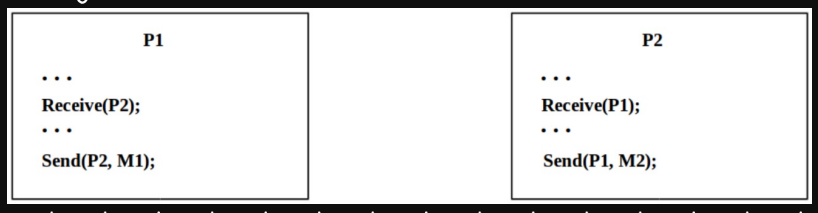
↳ deadlock can also occur if processes request large amount of resource, but in incremental small chunks

- e.g. space is avail for allocation of 200 KB total

P1	P2
. . .	. . .
Request 80 Kbytes;	Request 70 Kbytes;
. . .	. . .
Request 60 Kbytes;	Request 80 Kbytes;

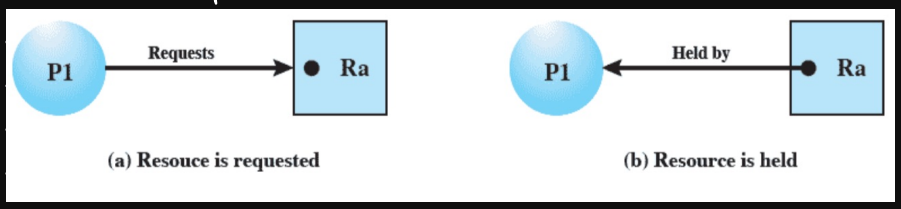
→ deadlock occurs if P1 & P2 progress to 2nd request (only 50 KB left)

consumable resource can be created / produced & destroyed / consumed
↳ e.g. interrupts, signals, msgs, & info in I/O buffers
↳ deadlock may occur if receive is blocking
- e.g.

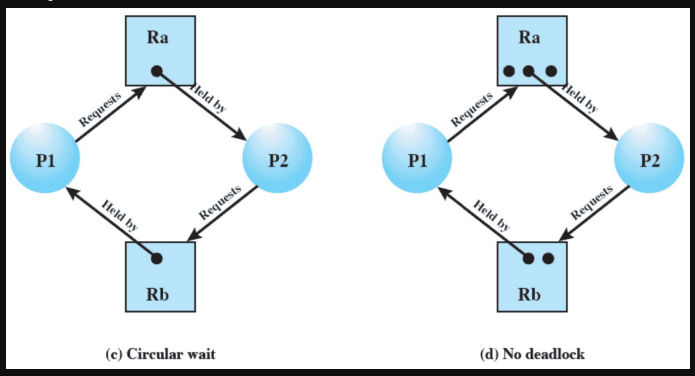


→ deadlock occurs if Receive fcn is blocking

resource allocation graph is directed graph that depicts state of system of resources & processes



↳ e.g.



conditions for deadlock:
1) mutual exclusion
→ only 1 process may use resource at a time
2) hold-and-wait
→ process holds alloced resources while awaiting asgnt of others
3) no preemption (wrt resources)
→ no resource can be forcibly removed from process holding it
4) circular wait
→ closed chain of processes st each process holds at least 1 resource needed by next process in chain
↳ 1-3 are necessary conditions, but 4 is sufficient for deadlock to acc take place

deadlock prevention is to design system in a way st no deadlock can occur

- ↳ indirect method is to prevent 1 of 3 necessary conditions
- ↳ direct method is to prevent circular wait

- mutual exclusion must be supported by OS
  - ↳ can't remove this cond
- prevent hold-and-wait by requiring process to request all of its required resources at 1 time & blocking it until all requests can be granted
  - ↳ process may be held up for long time when it could've proceeded w/ some of its resources
  - ↳ resources may be unused for some time, but still denied to other processes
  - ↳ process may not know all resources it needs in advance
- prevent no preemption:
  - ↳ if process holding resources is denied a further request, it must release all of its og ones
  - ↳ if process requests curr held resource, OS may preempt 2nd process & make it release resources
    - only works if no 2 processes have same priority
- prevent circular wait by defining linear ordering of resource types

- deadlock avoidance is when decision is made dynamically on whether curr resource alloc request will lead to potential deadlock if granted
  - ↳ allows 3 necessary conds so it allows more concurrency than prevention
  - ↳ requires knowledge of future process requests
  - ↳ at execution time
- consider system w/ n processes & m resource types, then define

Resource $= \mathbf{R} = (R_1, R_2, \ldots, R_m)$	Total amount of each resource in the system
Available $= \mathbf{V} = (V_1, V_2, \ldots, V_m)$	Total amount of each resource not allocated to any process
Claim $= \mathbf{C} = \begin{pmatrix} C_{11} & C_{12} & \ldots & C_{1m} \\ C_{21} & C_{22} & \ldots & C_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ C_{n1} & C_{n2} & \ldots & C_{nm} \end{pmatrix}$	C_{ij} = requirement of process i for resource j
Allocation $= \mathbf{A} = \begin{pmatrix} A_{11} & A_{12} & \ldots & A_{1m} \\ A_{21} & A_{22} & \ldots & A_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ A_{n1} & A_{n2} & \ldots & A_{nm} \end{pmatrix}$	A_{ij} = current allocation to process i of resource j

  - ↳ C is max requirements of each process
  - ↳ A is curr alloc of resources
  - ↳ all resources are either avail or alloced
    - $\forall j,\ R_j = V_j + \sum_{i=1}^{n} A_{ij}$
  - ↳ no process can claim more than total amt of resources in system
    - $\forall j,\ C_{ij} \leq R_j$

↳ no process is alloced more than its max
- $\forall j,\ A_{ij} \leq C_{ij}$

1st approach is don't start process if its demands might lead to deadlock
↳ if sum of all requested resources exceeds resource budget, don't admit process
- $\forall j,\ R_j \geq C_{(n+1)j} + \sum_{i=1}^{n} C_{ij}$

↳ aka process initiation denial
↳ not optimal b/c it assumes worst case where all processes will make max claims tgt

2nd approach is resource allocation denial
↳ aka banker's algo
↳ state of system is curr allocation of resources to processes
↳ goal is to always have safe state, which means there's at least 1 seq of resource allocs to processes that doesn't result in deadlock
- i.e. all processes can be completed

↳ e.g. safe state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	6	1	2
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	1
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(a) Initial state

Assign 1 R3 to P2 so it can run to completion.

	R1	R2	R3
P1	3	2	2
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
6	2	3

Available vector **V**

(b) P2 runs to completion

Choose to alloc 2 R1, 2 R2, + 2 R3 to P1 so it can run to completion.

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
7	2	3

Available vector **V**

(c) P1 runs to completion

Choose to alloc all resources P3 needs + it runs to completion. Finally, alloc resources to P4 + let it run to completion.

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	0	0	0
P2	0	0	0
P3	0	0	0
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
9	3	4

Available vector **V**

(d) P3 runs to completion

↳ e.g. unsafe state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	1	0	0
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	2	2	2
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
1	1	2

Available vector **V**

(a) Initial state

	R1	R2	R3
P1	3	2	2
P2	6	1	3
P3	3	1	4
P4	4	2	2

Claim matrix **C**

	R1	R2	R3
P1	2	0	1
P2	5	1	1
P3	2	1	1
P4	0	0	2

Allocation matrix **A**

	R1	R2	R3
P1	1	2	1
P2	1	0	2
P3	1	0	3
P4	4	2	0

C – A

R1	R2	R3
9	3	6

Resource vector **R**

R1	R2	R3
0	1	1

Available vector **V**

(b) P1 requests one unit each of R1 and R3

- each process needs at least 1 R1, which isn't avail
- b) isn't deadlocked state but has potential for it

e.g. deadlock avoidance logic

```
struct state {
       int resource[m];
       int available[m];
       int claim[n][m];
       int alloc[n][m];
}
```

(a) global data structures

```
if (alloc [i,*] + request [*] > claim [i,*])
     < error >;                              /* total request > claim*/
else if (request [*] > available [*])
     < suspend process >;
else {                                       /* simulate alloc */
     < define newstate by:
     alloc [i,*] = alloc [i,*] + request [*];
     available [*] = available [*] - request [*] >;
}
if (safe (newstate))
     < carry out allocation >;
else {
     < restore original state >;
     < suspend process >;
}
```

(b) resource alloc algorithm

```
boolean safe (state S) {
    int currentavail[m];
    process rest[<number of processes>];
    currentavail = available;
    rest = {all processes};
    possible = true;
    while (possible) {
        <find a process Pk in rest such that
            claim [k,*] - alloc [k,*] <= currentavail;>
        if (found) {                        /* simulate execution of Pk */
            currentavail = currentavail + alloc [k,*];
            rest = rest - {Pk};
        }
        else possible = false;
    }
    return (rest == null);
}
```

(c) test for safety algorithm (banker's algorithm)

- deadlock avoidance requirements:
  - ↳ max resource requirement must be stated in advance
  - ↳ processes must be indep (i.e. no synchro requirements)
  - ↳ must have fixed # resources to alloc
  - ↳ no process may exit while holding resources

- deadlock detection granted requested resources to processes whenever possible + OS periodically checks for circular wait cond
- deadlock detection is done after resource alloc but avoidance is done before to see if potential deadlock will occur
- algo for detecting deadlocks uses Allocation matrix, Avail vector, + request matrix Q (where Q_{ij} is # resources of type j requested by process i)
  1) start w/ unmarked Alloc matrix
  2) mark each row w/ all 0s in Alloc
     - process w/ no resources can't be in deadlock
  3) init temp vector W = Avail vector
  4) find unmarked row i st i^{th} row of Q ≤ W
     - if not found, terminate
  5) mark i^{th} row in Alloc + add that row to W, then go back to step 4
  - ↳ deadlock exists iff there's unmarked processes at end of algo
  - ↳ e.g.

	R1	R2	R3	R4	R5
P1	0	1	0	0	1
P2	0	0	1	0	1
P3	0	0	0	0	1
P4	1	0	1	0	1

Request matrix Q

	R1	R2	R3	R4	R5
P1	1	0	1	1	0
P2	1	1	0	0	0
P3	0	0	0	1	0
P4	0	0	0	0	0

Allocation matrix A

R1	R2	R3	R4	R5
2	1	1	2	1

Resource vector

R1	R2	R3	R4	R5
0	0	0	0	1

Available vector

Figure 6.10 Example for Deadlock Detection

  - mark P4

- W = (0 0 0 0 1)
- Q_{P3} = (0 0 0 0 1) ≤ W = (0 0 0 0 1) so i = 3
- mark P3
- W = (0 0 0 0 1) + (0 0 0 0 1)
  = (0 0 0 0 2)
- terminate algo b/c no unmarked row in Q ≤ W
- P1 + P2 are deadlocked

strategies once deadlock is detected:

↳ abort all deadlocked processes

↳ back up each deadlocked process to some prev defined checkpoint + restart all processes
- og deadlock may occur

↳ successively abort deadlocked processes until deadlock no longer exists

↳ successively preempt resources until deadlock no longer exists

when aborting / preempting, can choose process w/:

↳ least amt of processor time consumed

↳ least amt of output prod

↳ most estimated time remaining

↳ least total resources alloced so far

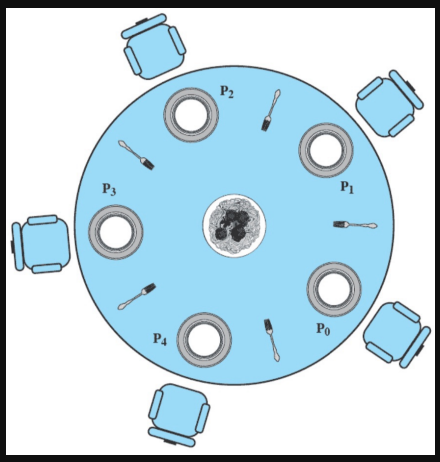
↳ lowest priority

↳ lowest importance

Table 6.1 Summary of Deadlock Detection, Prevention, and Avoidance Approaches for Operating Systems [ISLO80]

Approach	Resource Allocation Policy	Different Schemes	Major Advantages	Major Disadvantages
Prevention	Conservative; undercommits resources	Requesting all resources at once	• Works well for processes that perform a single burst of activity • No preemption necessary	• Inefficient • Delays process initiation • Future resource requirements must be known by processes
		Preemption	• Convenient when applied to resources whose state can be saved and restored easily	• Preempts more often than necessary
		Resource ordering	• Feasible to enforce via compile-time checks • Needs no run-time computation since problem is solved in system design	• Disallows incremental resource requests
Avoidance	Midway between that of detection and prevention	Manipulate to find at least one safe path	• No preemption necessary	• Future resource requirements must be known by OS • Processes can be blocked for long periods
Detection	Very liberal; requested resources are granted where possible	Invoke periodically to test for deadlock	• Never delays process initiation • Facilitates online handling	• Inherent preemption losses

dining philosophers problem : each requires 2 forks to eat so devise algo that'll satisfy mutual exclusion while avoiding deadlock + starvation



e.g. soln w/ semaphores results in deadlock

```
/* program      diningphilosophers */
semaphore fork [5] = {1};
int i;
void philosopher (int i)
{
     while (true) {
          think();
          wait (fork[i]);
          wait (fork [(i+1) mod 5]);
          eat();
          signal(fork [(i+1) mod 5]);
          signal(fork[i]);
     }
}
void main()
{
     parbegin (philosopher (0), philosopher (1), philosopher
(2),
          philosopher (3), philosopher (4));
     }
```

Figure 6.12 A First Solution to the Dining Philosophers Problem

↳ all philosophers pick up left fork at same time so it's a deadlock

e.g. correct soln w/ semaphores

```
/* program diningphilosophers */
semaphore fork[5] = {1};
semaphore room = {4};
int i;
void philosopher (int i)
{
    while (true) {
      think();
      wait (room);
      wait (fork[i]);
      wait (fork [(i+1) mod 5]);
      eat();
      signal (fork [(i+1) mod 5]);
      signal (fork[i]);
      signal (room);
    }

}
void main()
{
    parbegin (philosopher (0), philosopher (1), philosopher (2),
          philosopher (3), philosopher (4));
}
```

Figure 6.13 A Second Solution to the Dining Philosophers Problem

↳ w/at most 4 seated philosophers, at least 1 will have 2 forks

e.g. soln w/ monitor

```
monitor dining_controller;
cond ForkReady[5];          /* condition variable for synchronization */
boolean fork[5] = {true};       /* availability status of each fork */

void get_forks(int pid)         /* pid is the philosopher id number */
{
   int left = pid;
   int right = (++pid) % 5;
   /*grant the left fork*/
   if (!fork(left)
      cwait(ForkReady[left]);         /* queue on condition variable */
   fork(left) = false;
   /*grant the right fork*/
   if (!fork(right)
      cwait(ForkReady(right);         /* queue on condition variable */
   fork(right) = false:
}
void release_forks(int pid)
{
   int left = pid;
   int right = (++pid) % 5;
   /*release the left fork*/
   if (empty(ForkReady[left])      /*no one is waiting for this fork */
      fork(left) = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[left]);
   /*release the right fork*/
   if (empty(ForkReady[right])     /*no one is waiting for this fork */
      fork(right) = true;
   else                        /* awaken a process waiting on this fork */
      csignal(ForkReady[right]);
}
```

```
void philosopher[k=0 to 4]          /* the five philosopher clients */
{
   while (true) {
      <think>;
      get forks(k);          /* client requests two forks via monitor */
      <eat spaghetti>;
      release forks(k);      /* client releases forks via the monitor */
   }
}
```

Figure 6.14 A Solution to the Dining Philosophers Problem Using a Monitor

MEMORY MANAGEMENT

mem management is subdividing mem to accommodate multiple processes
- ↳ mem needs to be allocated to ensure reasonable supply of ready processes to consume avail processor time

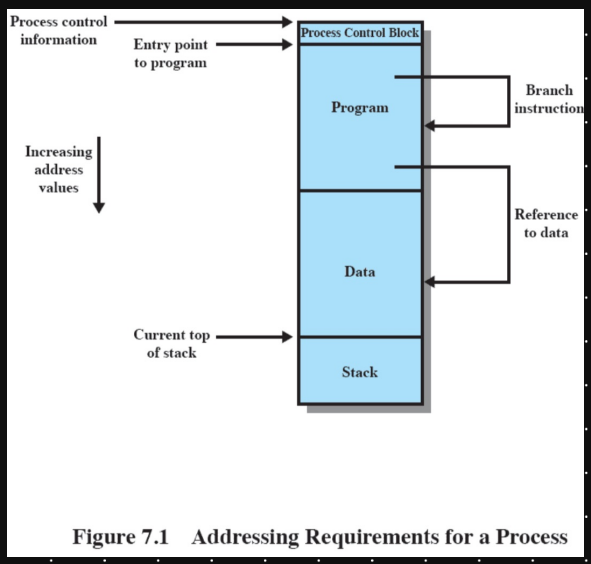
mem management requirements:
- ↳ **relocation**
  - programmer doesn't know where program will be placed in mem when executed
  - while executing program, may be swapped to disk + returned to main mem at diff loc
  - mem refs must be translated in code to acc physical mem address
    - → e.g. branch ins, data ref ins
  - 

Figure 7.1 Addressing Requirements for a Process
- ↳ **protection**
  - processes shouldn't be able to ref mem locs in another process w/o permission
  - impossible to check absolute addresses at compile time so must be checked at execution time
    - → e.g. seg fault bugs
  - must be satisfied by processor (hardware) instead of OS (software)
- ↳ **sharing**
  - allow several processes to access same portion of mem
  - better + more efficient to allow each process access to same copy of program rather than have separate copy
- ↳ **logical organization**
  - programs are written in modules, which can be written + compiled independently

- diff degs of protection for modules
  - → e.g. read-only, execute-only
- share modules among processes

↳ physical organization

- mem is organized into at least 2 lvls: main + secondary
- impractical to let programmer decide flow of info btwn 2 lvls of mem b/c:
  - → mem avail for program + its data may be insufficient so programmer must use overlaying, where various modules are assigned to same region of mem + programmer switches them in/out
  - → in multiprogramming env, programmer doesn't know how much space avail
- moving info btwn 2 lvls should be system responsibility

MEMORY PARTITIONING APPROACHES

fixed partitioning is to split avail mem into regions w/ fixed boundaries

↳ equal-size partitions

- any process whose size ≤ partition size can be loaded into an avail partition
- if all partitions full, OS can swap out process in a partition
- problems:
  - → program may be too big to fit so programmer must design program w/ overlays (i.e. only portion of program needs to be in main mem at 1 time + needed modules can be loaded/swapped in whenever)
  - → main mem utilization is inefficient, which leads to internal fragmentation (i.e. wasted space internal to partition)

↳ unequal-size partitions can lessen problems of equal-size ones

↳ e.g.

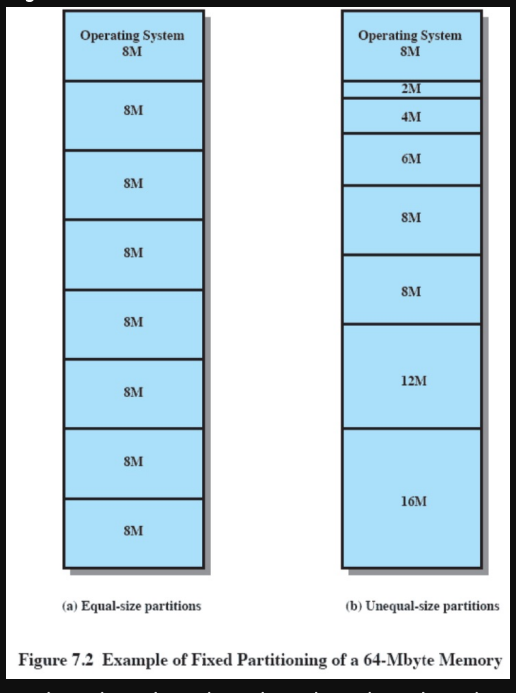


Figure 7.2 Example of Fixed Partitioning of a 64-Mbyte Memory

↳ placement algo:

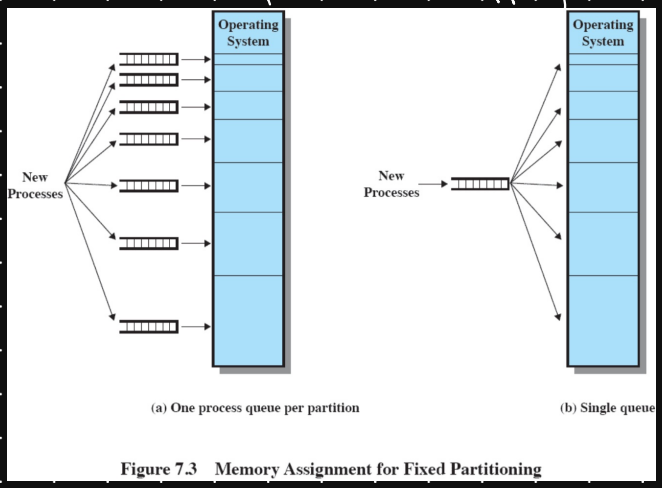
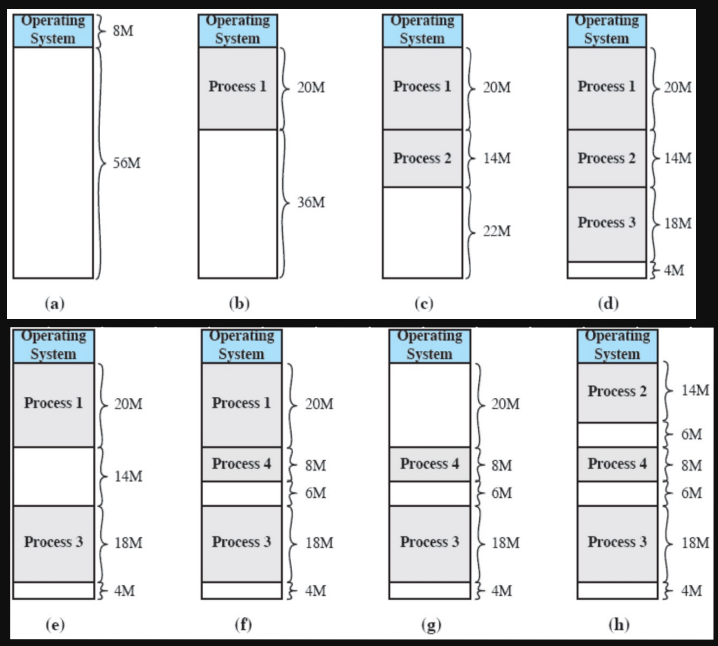
- trivial for equal-size
- for unequal-size, 2 possible ways:
  - → assign each process to smallest partition within which it'll fit. so we need scheduling queue for each partition
    - processes are always assigned st wasted mem in partition is minimized
    - bigger partitions may remain unused even though there's processes waiting in smaller partitions' queues
  - → employ single queue + select smallest avail partition to hold process
    - if all occupied, make swapping decision
- 

Figure 7.3 Memory Assignment for Fixed Partitioning

dynamic partitioning: process is alloced exactly as much mem as required.
↳ partitions are of var length + #
↳ leads to external fragmentation, which are holes in mem outside of partitions
↳ use compaction to shift processes so they're contiguous + all free mem is in 1 block.
↳ e.g.



↳ placement algo:

- best-fit chooses block that's closest in size to request
  - → worst performer b/c it guarantees fragment behind is as small as possible so main mem. ends up w/many blocks too small to satisfy requests
  - → compaction must be done more frequently
- first-fit scans mem from start & chooses 1st avail block that's big enough
  - → fastest & simplest
  - → may litter front end w/small free partitions
- next-fit scans mem from loc of last placement & chooses next avail block that's big enough
  - → more often allocs block of mem at end where largest block is & it gets broken up into smaller blocks
  - → compaction may be required more to keep large block at end
- e.g.

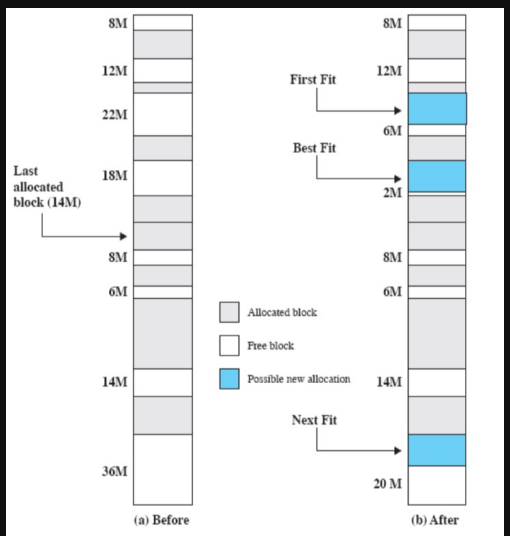


Figure 7.5 Example Memory Configuration before and after Allocation of 16-Mbyte Block

buddy system is compromise btwn fixed & dynamic partitioning

↳ mem blocks are avail of size 2^k words, $L \leq K \leq U$

- 2^L = smallest size
- 2^U = largest size

↳ entire space avail is treated as single block of 2^U

↳ if request of size s st $2^{U-1} < s \leq 2^U$, then entire block is alloced (i.e. find smallest possible block that'll fit s)

- otherwise, block is split into 2 equal sized buddies
- process continues until smallest block $\geq s$ is generated

↳ e.g.

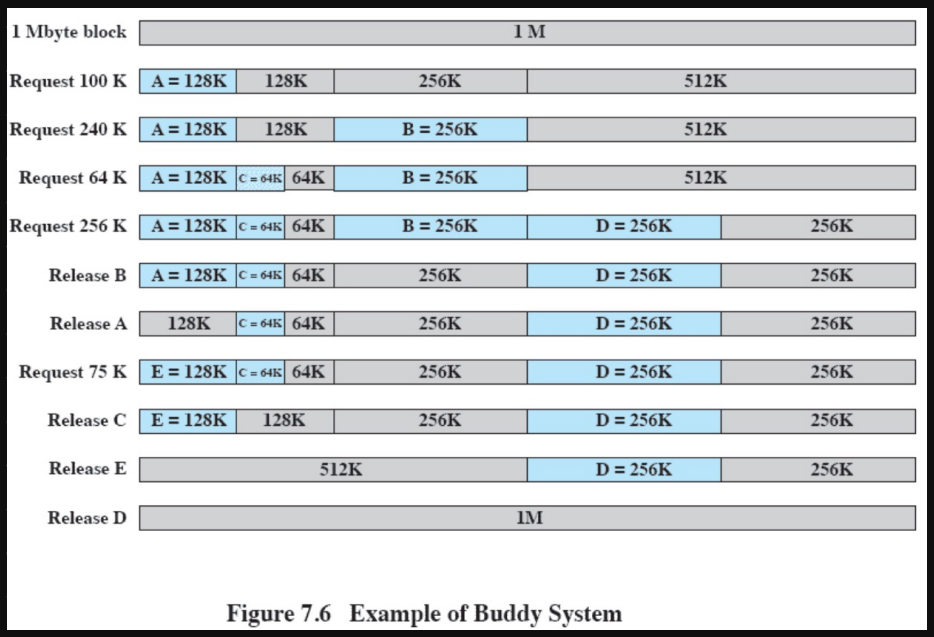


Figure 7.6 Example of Buddy System

↳ e.g. tree rep of buddy system

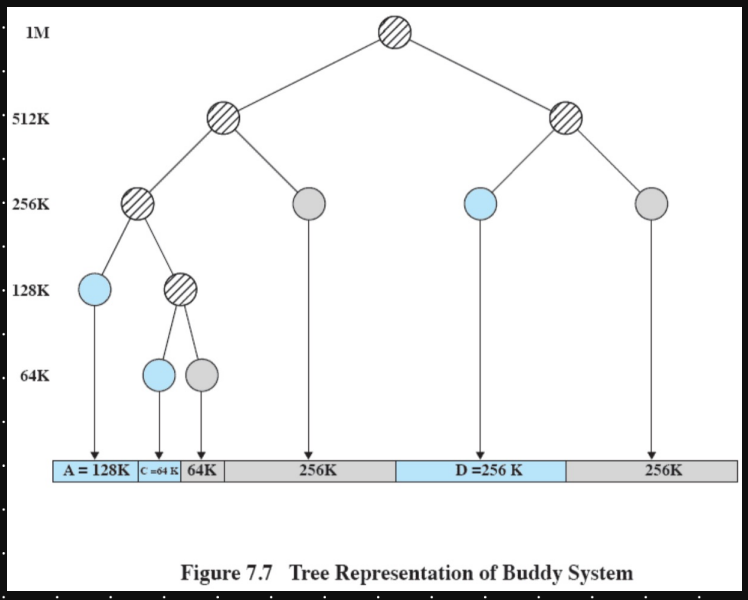


Figure 7.7 Tree Representation of Buddy System

- search in tree for avail. leaf nodes

- when process is loaded into mem, all rltive mem refs are replaced by absolute main mem addresses
  - ↳ in fixed-sized partitioning, process may occupy diff partitions during course of execution due to swapping
    - i.e. diff absolute addresses
  - ↳ in dynamic partitioning, compaction will also cause process to occupy diff partitions + swapping them in/out as well
- types of addresses:
  - ↳ logical: ref to mem indep of curr asgnt of data to mem
    - must be translated to physical addr

↳ rltive: expressed as loc rltive to some known pt
↳ physical: absolute addr / loc in main mem

hardware support for relocation (assuming process is in 1 partition)

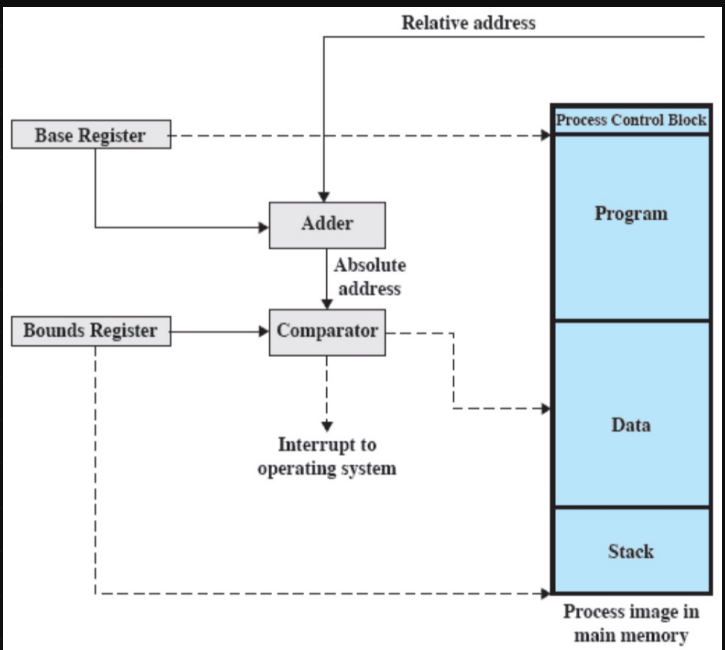


Figure 7.8 Hardware Support for Relocation

↳ base register: starting addr for process
↳ bounds register: ending loc of process
} set when process is loaded/swapped in

↳ process:
1) absolute addr = base register val + rltive addr
2) compare absolute addr w/ bounds register val
3) if not within bounds, gen interrupt to OS

↳ inefficient b/c putting process all into 1 partition leads to fragmentation

- fixed-size partitioning = internal fragmentation
- dynamic partitioning = external fragmentation

PAGING

partition mem into small, equal, fixed-size chunks + divide process into same sized chunks

↳ pages are chunks of process
↳ frames are chunks of mem

base addr ptr no longer sufficient b/c process doesn't have to be put in contiguous frames

↳ OS must have pg table for each process
- frame loc of each pg
- mem addr consists of pg # + offset (i.e. dist from start of pg where acc data is located)

↳ logical-to-physical addr translation still done by hardware
- processor translates logical (pg # + offset) to physical (frame # + offset) using pg table

w/ paging, there's only internal fragmentation on last pg + no external

e.g.

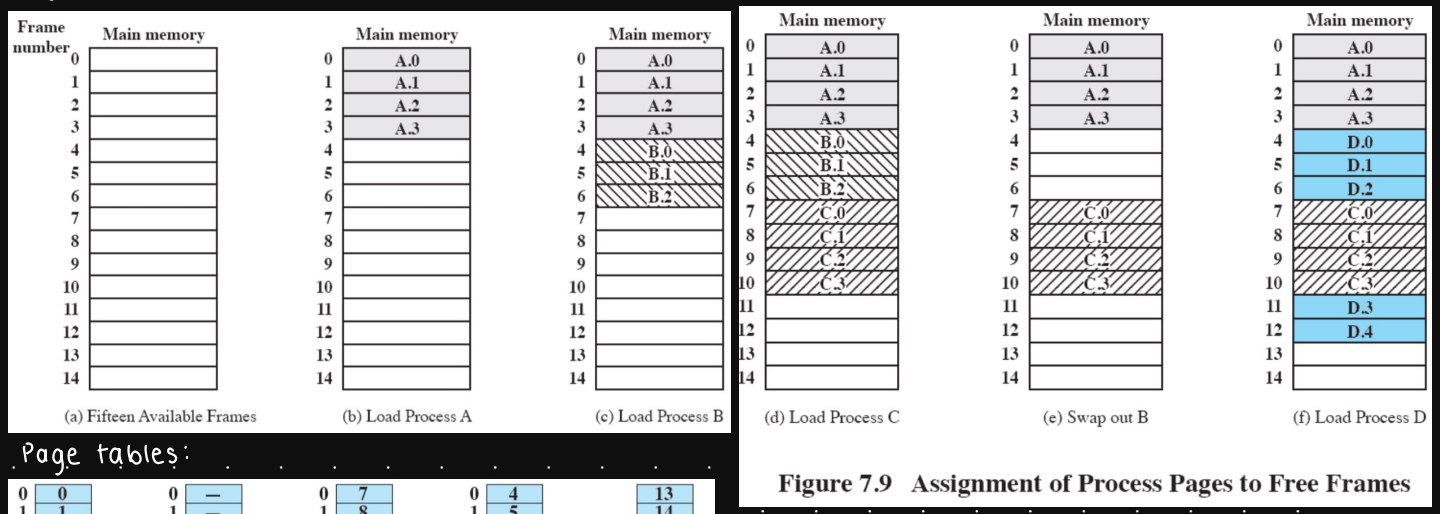
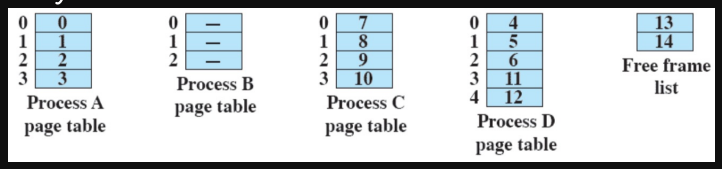
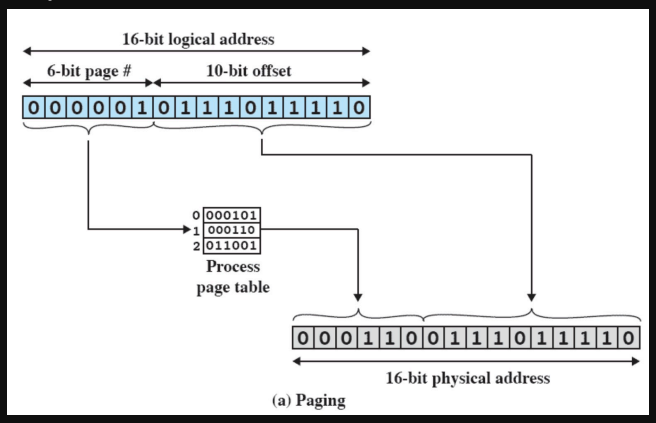


Figure 7.9 Assignment of Process Pages to Free Frames

Page tables:



e.g. address translation for paging



NOTE
Pg / frame size is to power of 2

↳ addr has n+m bits
- leftmost n bits are pg #
- rightmost m bits are offset within pg

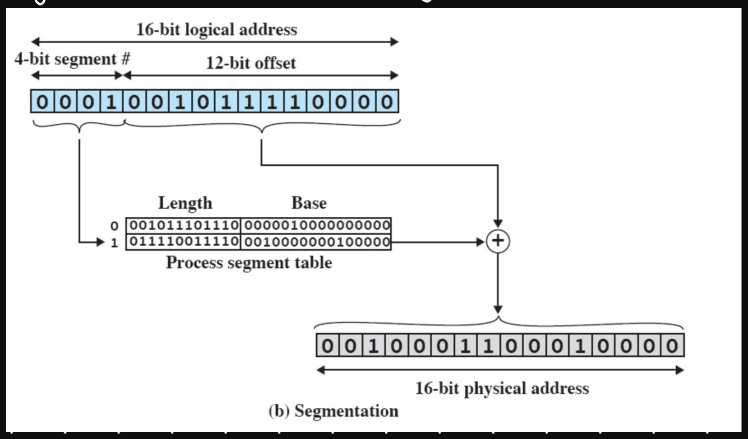
↳ extract pg # as leftmost n bits of logical addr
↳ use pg # as index into process pg table to find frame # k
↳ starting physical addr of frame is $k(2^m)$
- physical addr is $k(2^m)$ + offset

consequences of using pg size that's power of 2
↳ logical addressing scheme is transparent to programmer, assembler, + linker
- logical addr = rltive addr

↳ can implement fcn in hardware to perform dynamic addr translation at run time

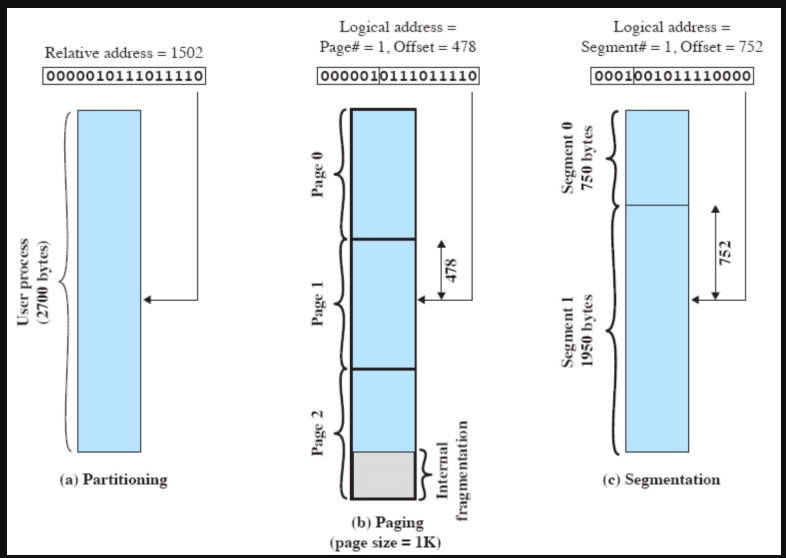
segmentation: program divided into segments, which don't have to be same length
- ↳ max segment length
- ↳ similar to dynamic partitioning, but program can have 1+ segments as opposed to only 1 partition
- ↳ logical addr has **segment # + offset**
- ↳ while paging is invisible, segmentation is visible + provided as convenience for organizing programs
- ↳ in **segment table**, each entry has starting addr in main mem + length of segment

e.g. addr translation for segmentation



- ↳ addr has n+m bits
  - ◦ leftmost n bits are segment #
  - ◦ rightmost m bits are offset
- ↳ extract segment # as leftmost n bits of logical addr
- ↳ use segment # as index into process segment table to get physical starting addr
- ↳ if offset ≥ segment length, addr invalid
- ↳ physical addr = starting addr + offset

e.g. **logical addrs**



all mem management methods can address same amt of total mem
↳ no method can address more mem

comparison btwn mem management techniques

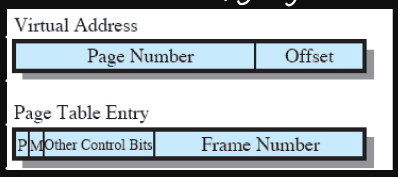
Technique	Description	Strengths	Weaknesses
Fixed Partitioning	Main memory is divided into a number of static partitions at system generation time. A process may be loaded into a partition of equal or greater size.	Simple to implement; little operating system overhead.	Inefficient use of memory due to internal fragmentation; maximum number of active processes is fixed.
Dynamic Partitioning	Partitions are created dynamically, so that each process is loaded into a partition of exactly the same size as that process.	No internal fragmentation; more efficient use of main memory.	Inefficient use of processor due to the need for compaction to counter external fragmentation.
Simple Paging	Main memory is divided into a number of equal-size frames. Each process is divided into a number of equal-size pages of the same length as frames. A process is loaded by loading all of its pages into available, not necessarily contiguous, frames.	No external fragmentation.	A small amount of internal fragmentation.
Simple Segmentation	Each process is divided into a number of segments. A process is loaded by loading all of its segments into dynamic partitions that need not be contiguous.	No internal fragmentation; improved memory utilization and reduced overhead compared to dynamic partitioning.	External fragmentation.
Virtual Memory Paging	As with simple paging, except that it is not necessary to load all of the pages of a process. Nonresident pages that are needed are brought in later automatically.	No external fragmentation; higher degree of multiprogramming; large virtual address space.	Overhead of complex memory management.
Virtual Memory Segmentation	As with simple segmentation, except that it is not necessary to load all of the segments of a process. Nonresident segments that are needed are brought in later automatically.	No internal fragmentation, higher degree of multiprogramming; large virtual address space; protection and sharing support.	Overhead of complex memory management.

VIRTUAL MEMORY

HARDWARE AND CONTROL STRUCTURES

- **virtual mem (VM)** allows for loading program's code + data on demand
  - ↳ upon reads, load mem from disk
  - ↳ when mem is unused, write it to disk
- 2 **enablers** of VM that come from pging + segmentation:
  - ↳ all mem refs in program are logical addrs that are dynamically translated to physical addrs at runtime
  - ↳ noncontiguous mem layouts
- implications of VM:
  - ↳ more processes maintained in main mem
  - ↳ process may be larger than all of main mem
- **resident set**: portion of process that's acc in main mem
- **real mem**: actual main mem
  - ↳ RAM chips
- **virtual mem**: user-perceived mem, which is larger + alloced on secondary mem (i.e. disk)
  - ↳ RAM + swap space
- only need 2 things in main mem: next ins to execute + next data to be accessed
- to use VM in executing process:
  - ↳ load some part of program
    - for new process, load in initial program pg/segment + data that the piece refers to
  - ↳ start executing until program tries to read / execute smth not in RAM (i.e. **pg fault**)
  - ↳ upon pg fault, block process, read in data, then resume process
- VM allows for **lazy loading**, meaning only part of process is loaded into main mem at a time
  - ↳ CPU is kept busy so less wasted time
  - ↳ hopefully, only small part of each process active at a time
- when main mem is full, OS has to decide what to remove when putting smth in
  - ↳ want to minimize **thrashing**, where OS spends most of its time swapping pieces rather than executing ins
- VM scheme works b/c of **principle of locality** (i.e. program + data refs within process tend to cluster)
  - ↳ OS assumes that stuff needed in future is close to stuff needed in past
- **requirements** of VM:
  - ↳ hardware must support pging + segmentation
  - ↳ OS system support for putting pgs on disk + reloading them from disk

- VM based on **paging**:



  - ↳ **P (present)**: bit that indicates whether curr pg is in main mem or not
  - ↳ **M (modify)**: bit that indicates whether pg contents have been altered since last loaded into main mem
    - if modified, must write pg out to disk before replacing
- each process must have its own pg table
- **addr translation in paging** system:

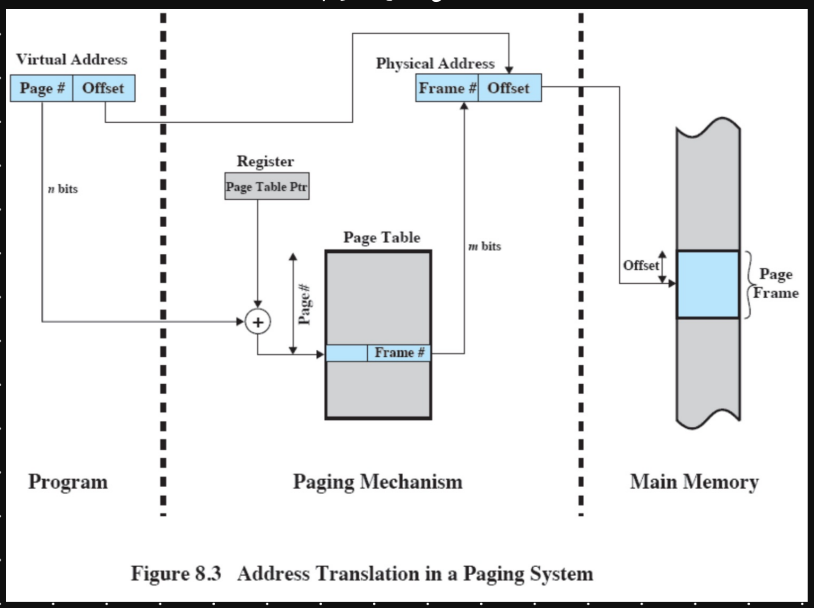
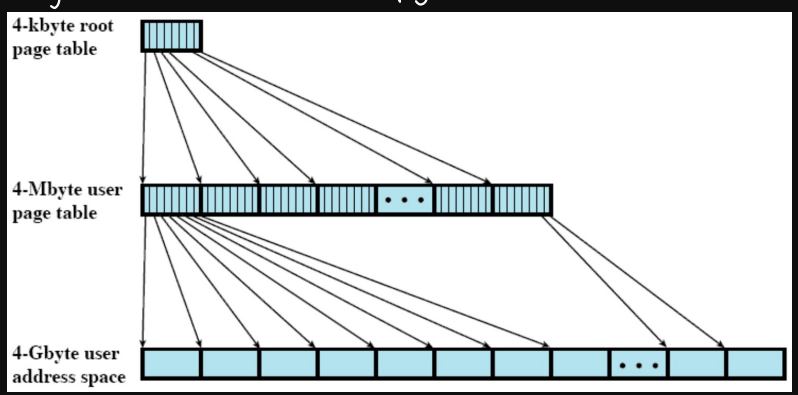


Figure 8.3 Address Translation in a Paging System

  - ↳ typically n>m
- pg tables for large processes will use a lot RAM themselves
  - ↳ e.g. 4MB pg tables per process for 4GB RAM
  - ↳ soln is to **store pg tables in VM**
    - when process is running, store part of its pg table (including **pg table entry (PTE)** of curr pg) in main mem
    - have **root pg table** fully in main mem
    - e.g. 2-lvl hierachical pg table



→ addr translation:

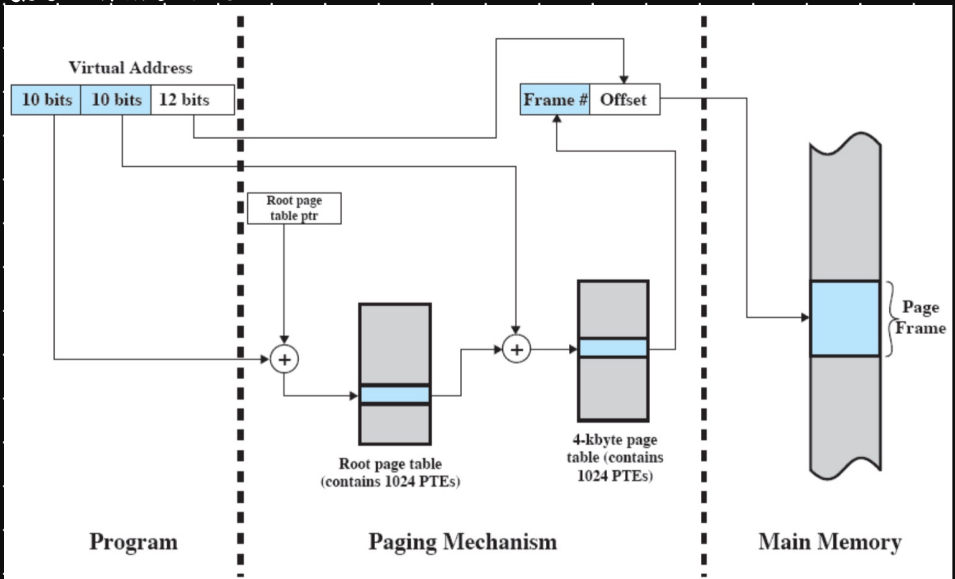


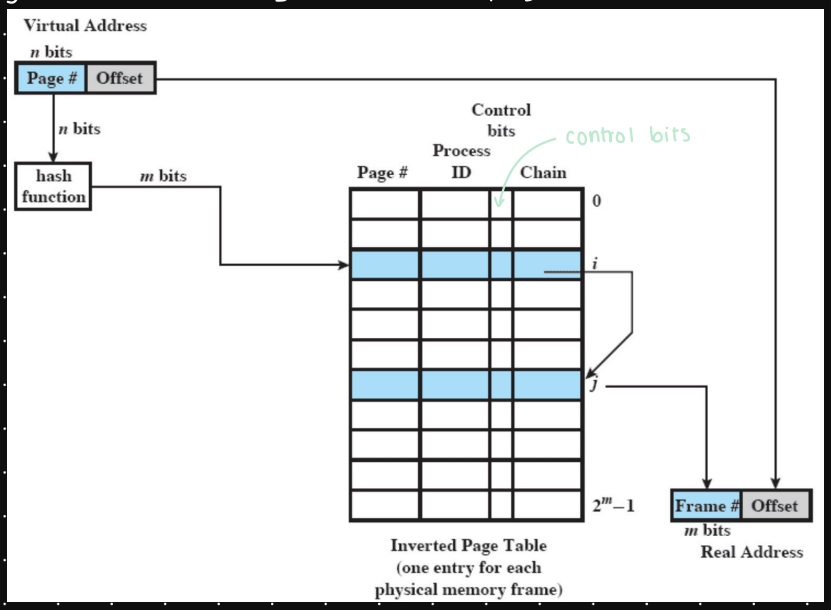
Figure 8.5 Address Translation in a Two-Level Paging System

- possible to have double pg fault

drawback of pg tables is that their pg table size is proportional to virtual addr space

↳ alt is to use hash table to make inverted pg table

- inverted b/c we index on frame #, not virtual addr
- grow proportionally to size of physical mem
-



→ process ID: process that owns that pg

→ control bits: flags for valid, referenced, modified, etc. + protection + locking info

→ chain ptr: null if no chained entries or index val (btwn 0 + $2^m - 1$) of next entry in chain

each VM ref causes 2 physical mem accesses:
- ↳ fetch PTE
- ↳ fetch data

translation lookaside buffer (TLB): high-speed cache for PTEs
- ↳ TLB inc efficiency due to principle of locality
- ↳

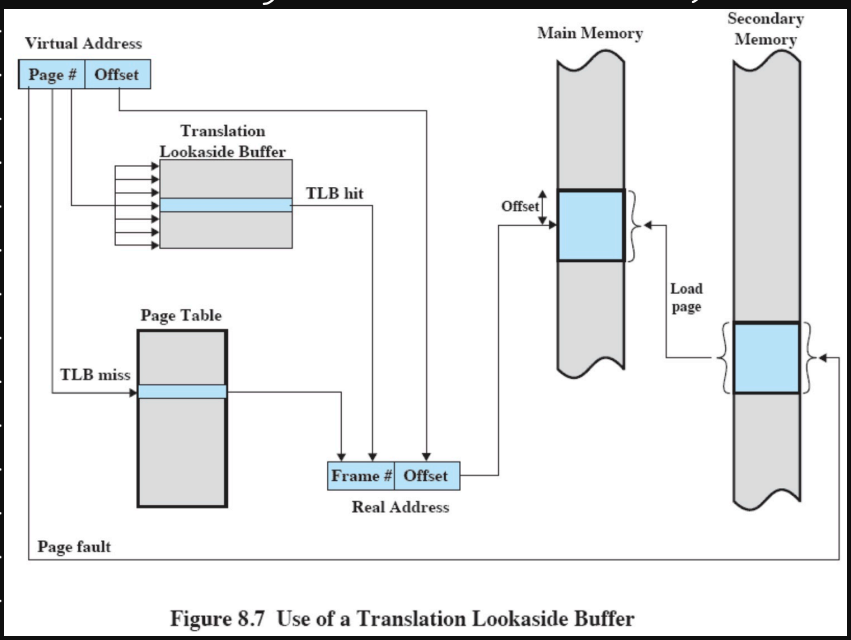
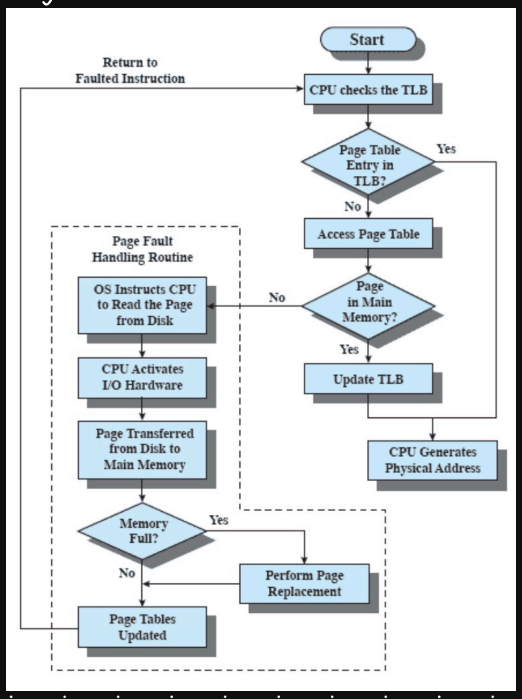
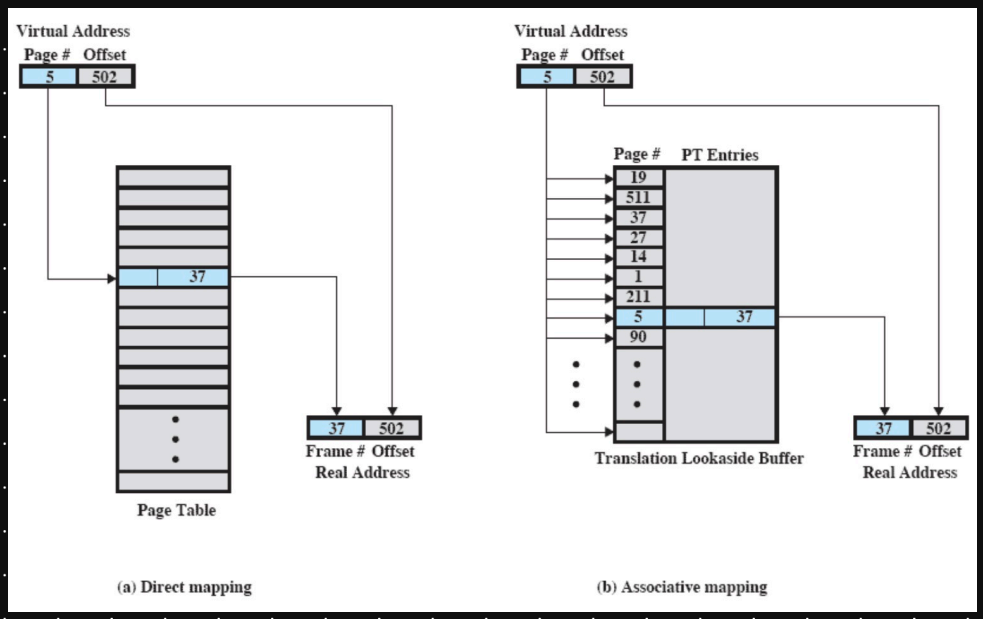


Figure 8.7 Use of a Translation Lookaside Buffer

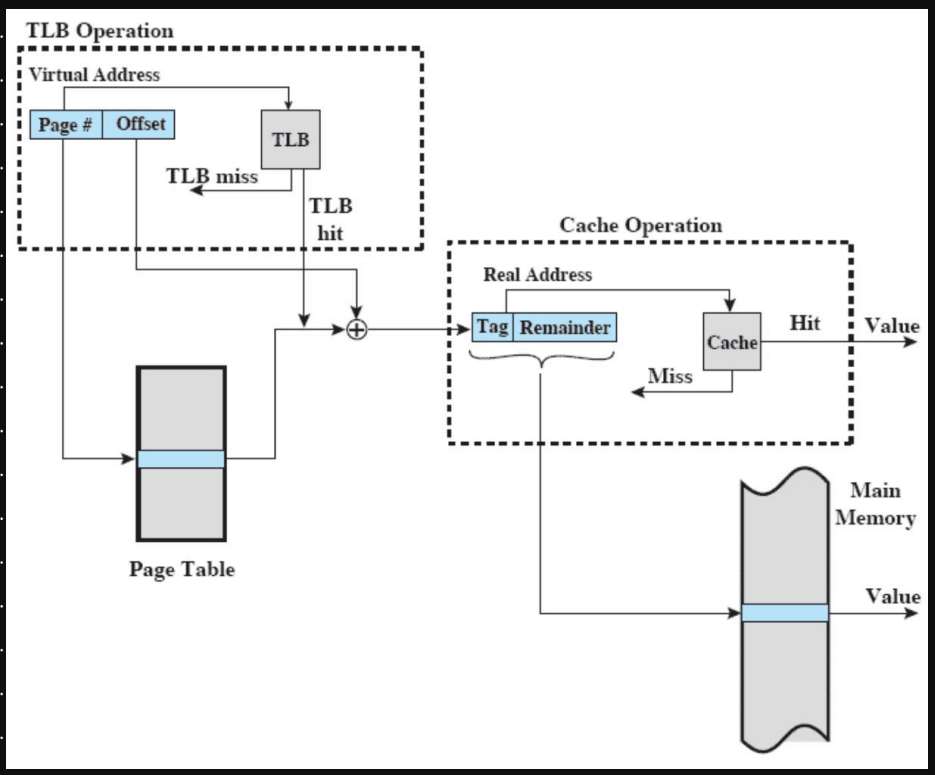
- ↳ logical flowchart:



- ↳ processor can simultaneously check multiple TLB entries to see if there's match on pg # (aka associative mapping)
  - faster than direct mapping needed for pg tables
  - comparison:



how VM mechanism interacts w/ main mem cache:



↳ virtual addr in form of pg # + offset
↳ TLB is consulted to see if PTE is present
- if so, physical addr = frame # + offset
- if not, pg table is consulted

↳ physical addr in form of tag (i.e. leftmost bits) + remainder
↳ cache is consulted to see if block containing word is present
- if so, it's returned to CPU
- if not, word retrieved from main mem

secondary mem is designed to efficiently transfer large blocks of data so

large pg size is better in this case
↳ disk prods streams

smaller pg sizes result in less internal fragmentation, but:
↳ more pgs required per process
- less TLB entries + more TLB misses

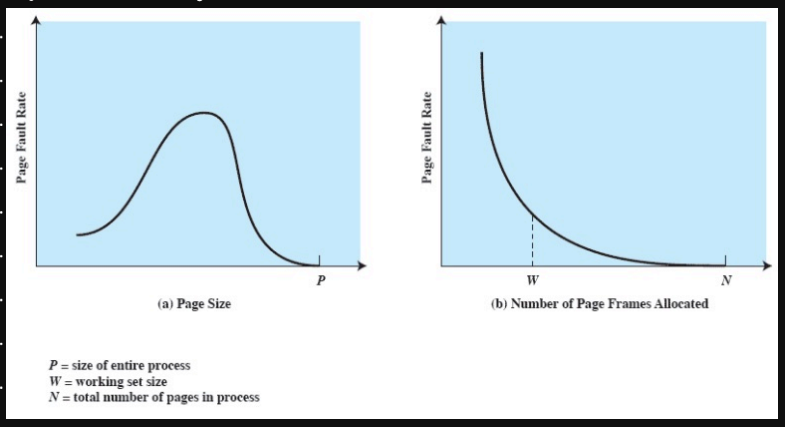
↳ larger pg tables b/c more pgs per process
- large portion of pg tables taking up virtual mem
- possibility of double pg fault

↳ more disk access

small pg size means larger # pgs found in main mem
↳ as execution of process progresses, more pgs in mem will all contain portions of process near recent refs
- low pg faults

↳ inc pg size causes pgs to contain locs further from recent refs
- inc pg faults

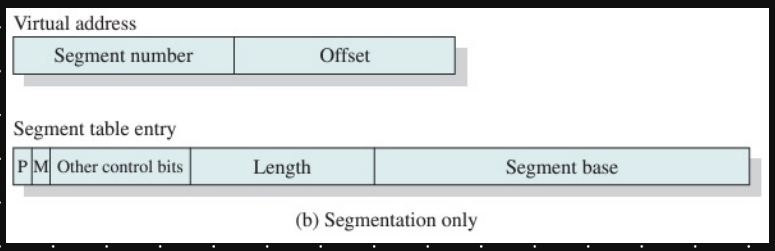
↳ pg size vs pg fault rate:



- pg fault rate eventually dec b/c size of pg approaches size of process
  → P is when they're equal

segmentation allows programmer to view mem consisting of several addr spaces or dynamic segments
↳ advantages:
- simplifies how to handle growing data struct
  → put entire struct into 1 segment
- allows programs to be altered & recompiled independently
  → code & data have own segments
- can share data among processes
  → share segment
- protects segments

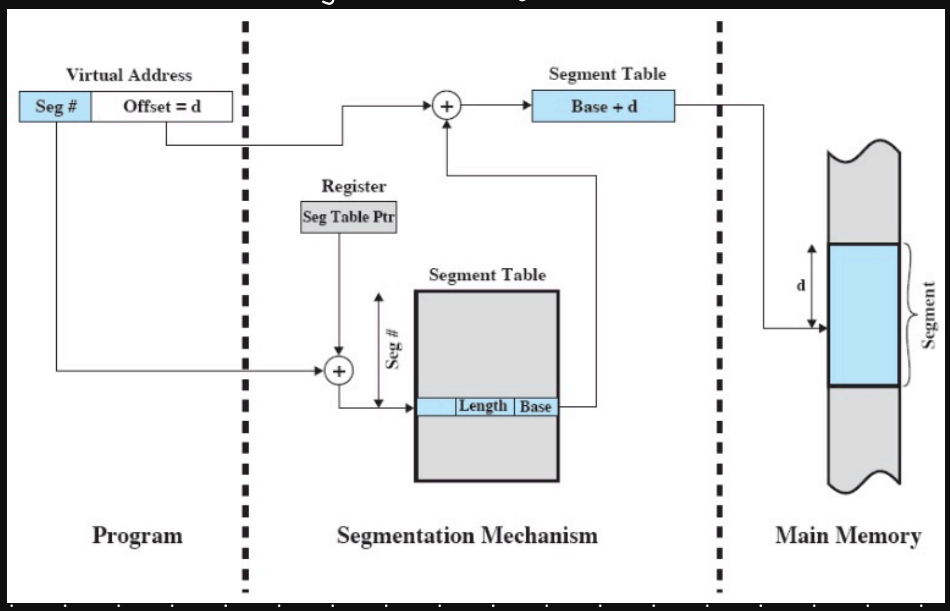
each entry in segment table looks like:



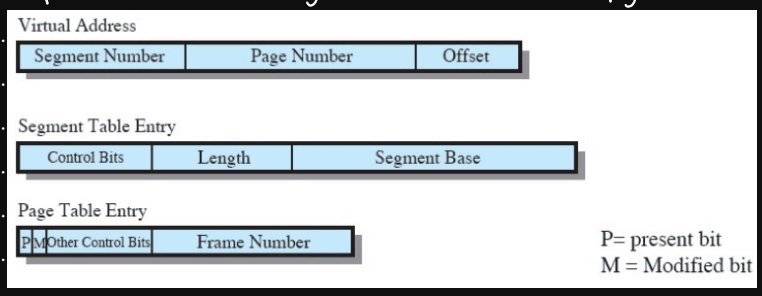
(b) Segmentation only

↳ segment base is starting addr of segment in main mem
↳ len of segment
↳ P (present) bit that indicates whether curr segment is in main mem or not
↳ M (modify) bit that indicates whether contents have been altered since last loaded into main mem
↳ other control bits (e.g. protection)

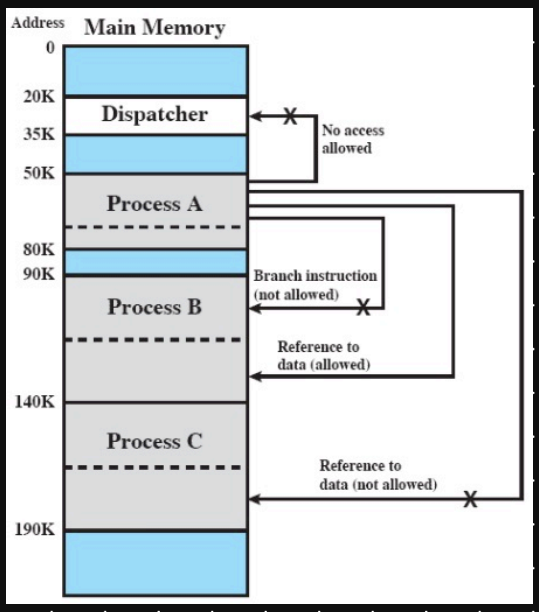
addr translation in segmentation system:



combined pging + segmentation system uses fact that pging is invisible & segmentation is visible to user
each process has 1 segment table & 1 pg table for each segment
↳

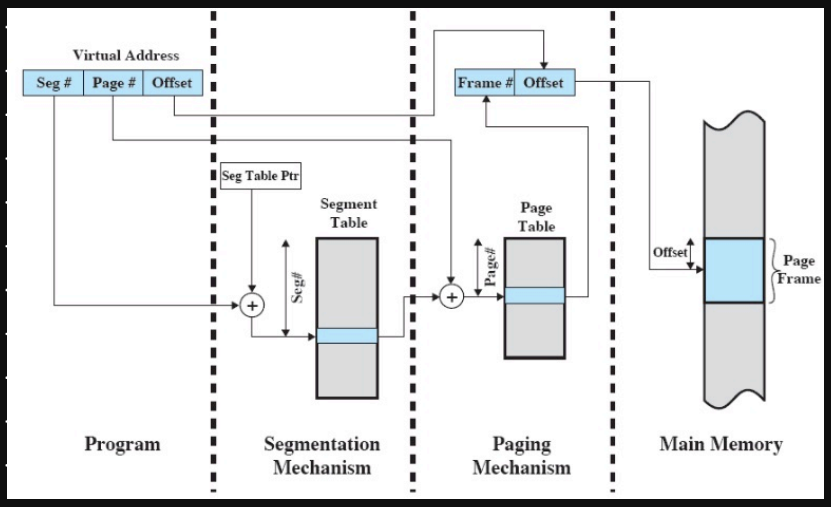


segmentation lends itself to use of protection & sharing policies
↳ same mechanisms avail in pging system, but more difficult b/c program's pg struct invisible to programmer
↳ e.g. protection rltnships btwn segments



↳ more sophisticated mechanisms, like ring-protection struct, can be used
- program may access only data on same ring or less-privileged ones
- program may call services on same ring or more-privileged ones

addr translation in combined pging + segmentation system:



OS SOFTWARE

design of mem management portion of OS depends on:
↳ if VM techniques used } depend on hardware
↳ if pging, segmentation, or both used } depend on hardware
↳ algos for mem management
- fetch policy
- placement policy
- replacement policy
- cleaning policy
- load control

fetch policy: determines when pg should be brought into mem
↳ demand pging only brings pgs into main mem when a ref is made to a loc on

pg
- many pg faults when process starts

↳ **prepging** brings in more pgs than needed
- more efficient to bring in pgs that reside contiguously on disk

placement policy: determines where in real mem a process piece resides
↳ in segmentation system, important to minimize external fragmentation
- e.g. first-fit, last-fit, best-fit

↳ in pging or combo system, irrelevant b/c addr translation + main mem access hardware equally efficient for any pg-frame set

replacement policy: determines which pg is replaced
↳ pg removed should be pg least likely to be ref in near future
↳ most policies predict future behaviour based on past b/c of principle of locality
↳ basic concepts to consider:
- # pg frames allocated to active process
  → fixed or var size
- local vs global scope when replacing
- which pg to choose

frame locking: restricts replacement policy b/c when frame is locked, it may not be replaced
↳ e.g. useful for kernel, key control structs, I/O buffers, + time critical elmts
↳ each frame has lock bit

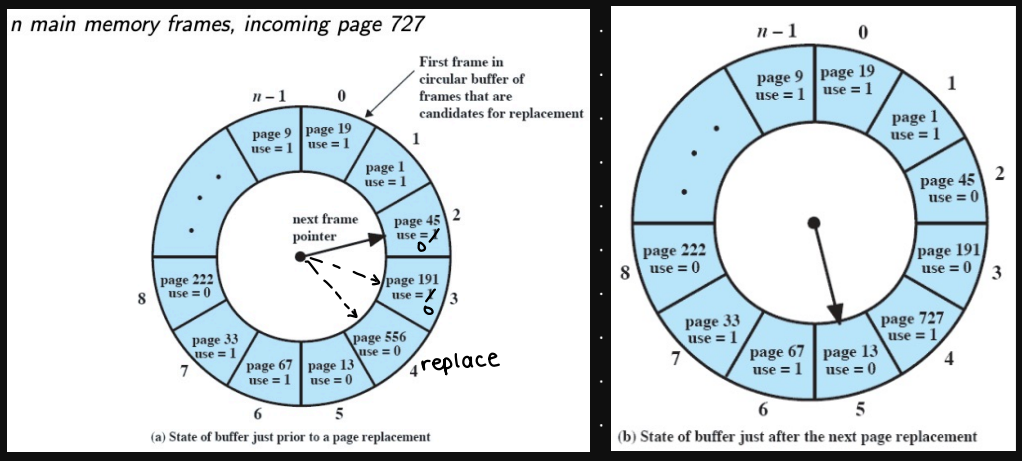
replacement algos:
↳ **optimal policy**: selects replacement st time to next ref of that pg is longest
- impossible

↳ **least recently used (LRU)**: replaces pg that has not been ref for longest time
- by principle of locality, pg should be least likely ref in near future
- nearly as good as optimal
- each pg needs to be tagged w/ time of last ref so it requires a lot of overhead

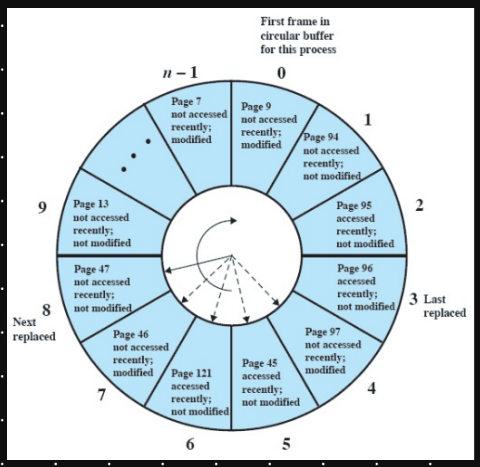
↳ **first-in, first-out (FIFO)**: pg that's been in mem longest is replaced
- assumes old pg won't be ref soon
- puts pgs alloced to process in circular buffer + removes them round-robin style
- simplest to implement
- worst performer

↳ **clock policy**: apx LRU using circular buffer + has bit called **use bit**
- when pg 1st loaded into main mem, use = 1
- when pg ref, use = 1
- when time to replace, 1st frame encountered w/ use = 0 is replaced
- during search for replacement, every use = 1 encountered is set to use = 0

- e.g.



- can employ another modify bit to improve
  1) make 1st scan around buffer searching for u=0, m=0 & make no changes to use bit
  2) if 1) fails, make 2nd scan for u=0, m=1 & set all passed u=1 to u=0
  3) if 2) fails, ptr should be in og pos & all frames have u=0 so we repeat 1) & 2) to find replacement frame
  → e.g.



comparison of pg replacement algos:

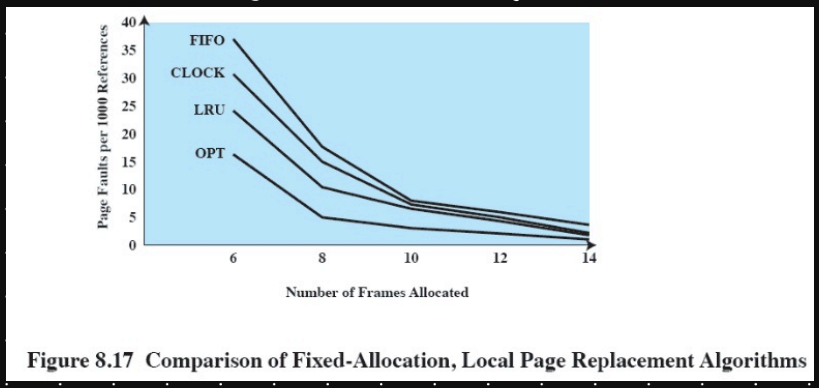
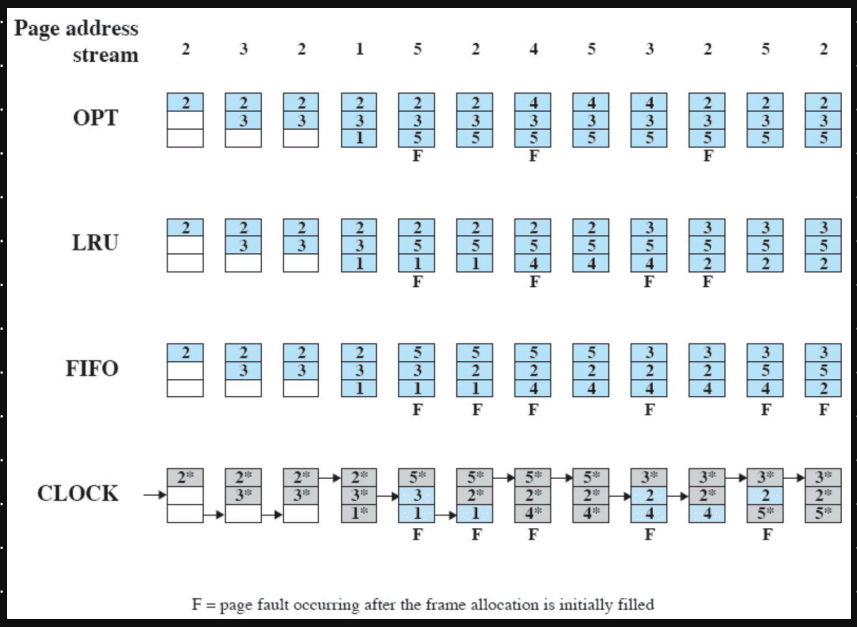


Figure 8.17 Comparison of Fixed-Allocation, Local Page Replacement Algorithms

↳ ideal mode of op is at knee of curve to get low pg fault & low allocation

e.g. behaviour of pg replacement algos



pg buffering: implements cache for mem pages
- ↳ replaced pg is added to 1 of 2 lists:
  - free pg list if pg hasn't been modified
  - modified pg list
    - modified pgs are written out in clusters instead of 1 at a time
- ↳ reduces swapping overhead + improves performance
- ↳ OS can revive pgs from list if they're needed again or if space becomes avail
- ↳ supports bursty block writes of I/O

important factors in deciding resident set size:
- ↳ smaller = more processes in main mem
- ↳ too small leads to high pg fault rate
- ↳ too big leads to no noticeable effect on pg fault rate due to principle of locality

2 policies to determine resident set size:
- ↳ fixed-allocation: gives process a fixed # pgs within which to execute
  - when pg fault occurs, 1 of pgs of that process must be replaced
- ↳ var-allocation: #pgs alloced to process varies over lifetime of process

replacement scope can be local (only resident pgs of process can be replaced) or global (all unlocked pgs in main mem can be replaced)

fixed alloc + local scope:
- ↳ decide ahead of time allocation amt to give process, which is difficult
- ↳ if alloc too small, high pg fault rate
- ↳ if alloc too big, too few programs in main mem
  - a lot of processor idle time or swapping

fixed alloc + global scope is not possible b/c that would change resident set

sizes of other processes

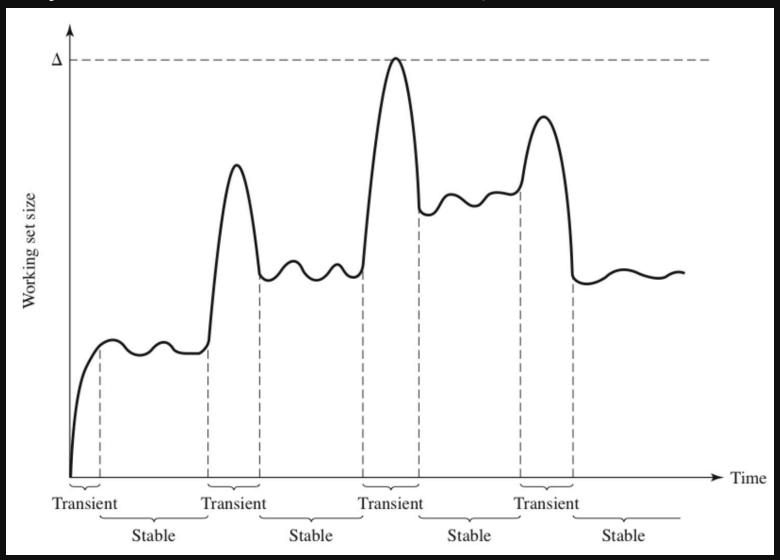
- **var alloc + global scope**:
  - ↳ OS keeps list of free frames
  - ↳ free frame is added to resident set of process when pg fault occurs
  - ↳ if no free frames avail, replace any unlocked one
  - ↳ easiest to implement
  - ↳ risk of process suffering reduction in resident set size
- **var alloc + local scope**:
  - ↳ when new process added, alloc # pg frames based on app type, program request, etc.
  - ↳ select pg from local resident set to replace when pg fault occurs
  - ↳ re-eval allocation from time to time
- **working set**: set of pgs of process that have been ref in last t logical time units
  - ↳ **window size Δ** is max # pgs in working set
  - ↳ e.g.

Sequence of Page References	Window Size, Δ = 2	Δ = 3	Δ = 4	Δ = 5
24	24	24	24	24
15	24 15	24 15	24 15	24 15
18	15 18	24 15 18	24 15 18	24 15 18
23	18 23	15 18 23	24 15 18 23	24 15 18 23
24	23 24	18 23 24	•	•
17	24 17	23 24 17	18 23 24 17	15 18 23 24 17
18	17 18	24 17 18	•	18 23 24 17
24	18 24	•	24 17 18	•
18	•	18 24	•	24 17 18
17	18 17	24 18 17	•	•
17	17	18 17	•	•
15	17 15	17 15	18 17 15	24 18 17 15
24	15 24	17 15 24	17 15 24	•
17	24 17	•	•	17 15 24
24	•	24 17	•	•
18	24 18	17 24 18	17 24 18	15 17 24 18

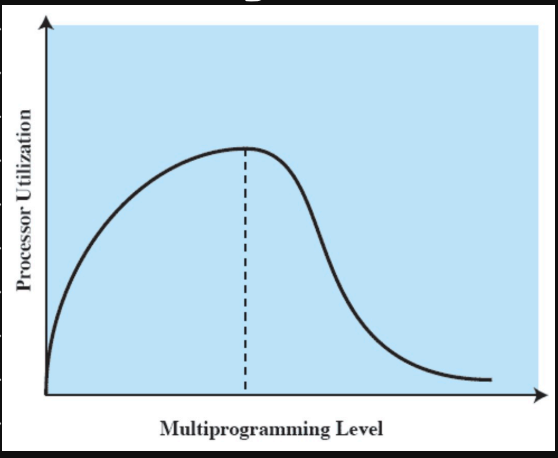
Figure 8.19 Working Set of Process as Defined by Window Size

- ° • means working set doesn't change (i.e. hit)
- ° get working set by counting last Δ pg refs

↳ e.g. typical graph of working set size



- to use working set as guide for resident set size:
  - ↳ monitor working set of process
  - ↳ periodically remove pgs only in resident but not working set
    - i.e. LRU policy
  - ↳ process may execute only if working set in main mem
    - i.e. resident set includes entire working set
  - ↳ problems:
    - past doesn't predict future
    - impractical to implement
    - optimal Δ unknown + varies
- **cleaning policy**: determines when modified pg should be written out to secondary mem
  - ↳ opp. of fetch policy
  - ↳ **demand cleaning**: pg written out only when selected for replacement
    - may require 2 pg transfers per pg fault
  - ↳ **pre-cleaning**: pgs written out in batches
    - allows bursty long I/O
    - write may be unnecessary
  - ↳ best approach is **pg buffering**
    - replaced pgs are either in modified or unmodified list
    - pgs in modified list periodically written out in batches
    - pgs in unmodified list either reclaimed if ref again or lost when its frame assigned to another pg
- **load control**: determines #processes that'll be resident in main mem
  - ↳ too few processes → all processes may be blocked + a lot of time spent swapping
  - ↳ too many processes → thrashing
- **multiprogramming lvl** effects:



- several strats to get optimal multiprogramming lvl:
  - ↳ working set or **pg fault freq (PFF)** algo implicitly incorporates load control

- only processes w/ large enough resident set can execute
- auto determines # active processes

↳ **L = S criterion**
- mean time btwn faults = time to process fault

↳ **50% criterion**
- keep use of pging device at 50%

if multiprogramming lvl reduced, options for process to **suspended** are:

↳ **lowest priority**

↳ **faulting**
- process doesn't have working set in main mem so it's blocked anyway

↳ **last activated**
- least likely to have working set

↳ **smallest resident set**
- least future effort to reload
- penalizes programs w/ strong locality

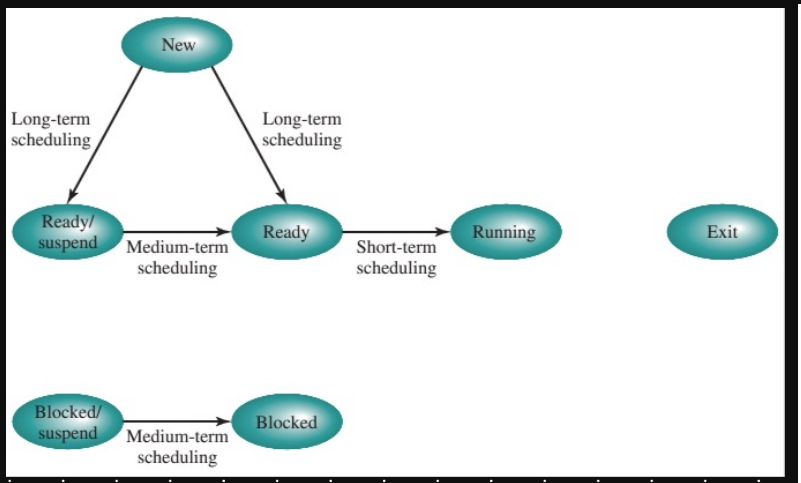
↳ **largest resident set**
- obtains most free frames

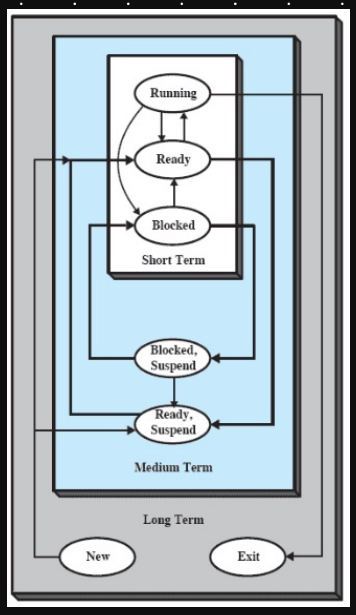
↳ **largest remaining execution window**
- apx shortest-processing-time-1st scheduling

UNIPROCESSOR SCHEDULING

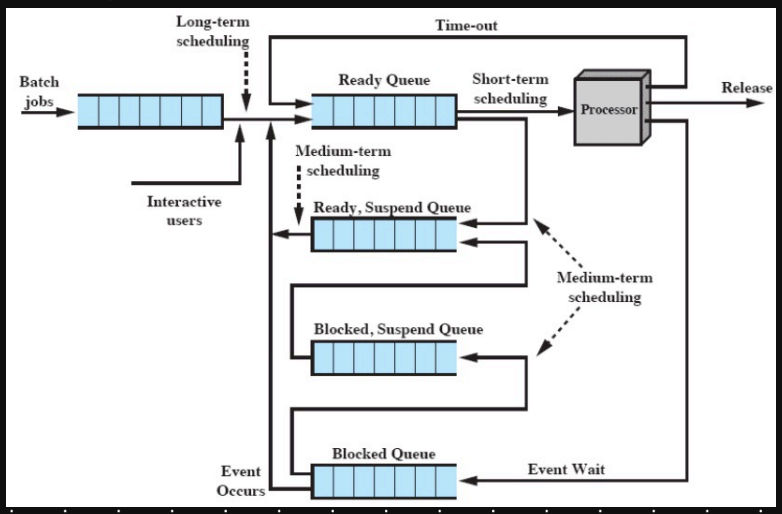
TYPES OF PROCESSOR SCHEDULING

- aim of **processor scheduling** is to assign processes to be executed
  - ↳ affects performance by determining which progress waits + which progresses
  - ↳ scheduler must meet many objectives (e.g. response time, thrupur, processor efficiency, temp, power)
- types of scheduling:
  - ↳ **long-term**: add to pool of processes to be executed
  - ↳ **medium-term**: add to #processes that are partially / fully in main mem
  - ↳ **short-term**: which avail process will be executed
  - ↳ **I/O**: which process' pending I/O request is handled by avail I/O device
- state transitions:





queuing diagram:

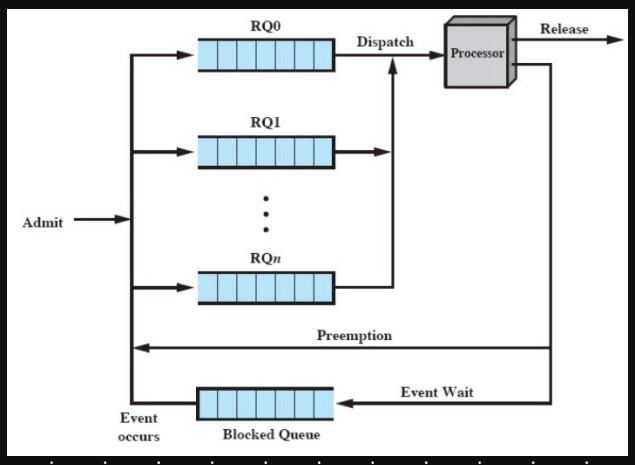


- **long-term scheduling** controls program admission + multiprogramming lvl
  - ↳ scheduler must decide if it can take on another process + which to choose (e.g. FIFO, priority, execution time)
  - ↳ more processes = smaller % time each process is executed
- **medium-term scheduling** is part of swapping fcn
  - ↳ depends on how much VM avail
  - ↳ based on need to manage deg of multiprogramming

SHORT-TERM SCHEDULING

- **short-term scheduling** executes most frequently + known as dispatcher
  - ↳ invoked when event occurs
    - clock interrupts
    - I/O interrupts
    - OS calls
    - signals
- criteria can be separated into 2 categories:
  - ↳ **user-oriented**
    - P • **turnaround time**: time btwn submission of process + completion
      - → execution time + waiting time
    - P • **response time**: time from submission of request until response begins to be received
    - P • **deadlines**: maximize % deadlines met
    - O • **predictability**: given job should take abt same time to run regardless of load on system
  - ↳ **system-oriented**
    - P • **thruput**: maximize #processes completed per time unit
    - P • **processor utilization**: % time processor's busy
      - → important in expensive shared system, but not so much in single-user
    - O • **fairness**: if there's no guidance, all processes should be treated same + none should starve
    - O • **enforcing priorities**
    - O • **balancing resources**: favour processes that underutilize stressed resources
- scheduler will always process of higher **priority**
  - ↳ use multiple rdy queues for each lvl
  - ↳ lower priority may suffer starvation
    - change priority based on age or execution history
  - ↳ priority queuing:

> NOTE
> P: performance-related
> ↳ quantitative + easy to measure
> O: other

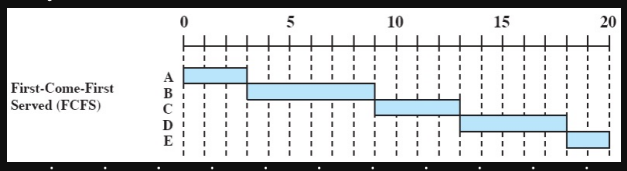


decision mode of scheduler specifies instants in time to do selection fcn
- ↳ nonpreemptive: once process is running, it'll continue until it terminates or blocks itself for I/O
- ↳ preemptive: OS can interrupt curr running process + move it to rdy state
  - better service b/c process can't monopolize processor
- ↳ cooperative

for following scheduling algos, use below set of processes as example:

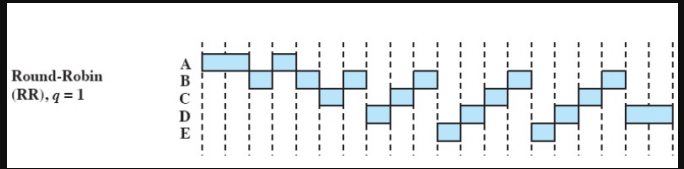
Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2

first-come-first-served (FCFS)
- ↳ each process joins Rdy queue
- ↳ when curr process stops executing, oldest process in queue selected
- ↳ nonpreemptive
- ↳ e.g.



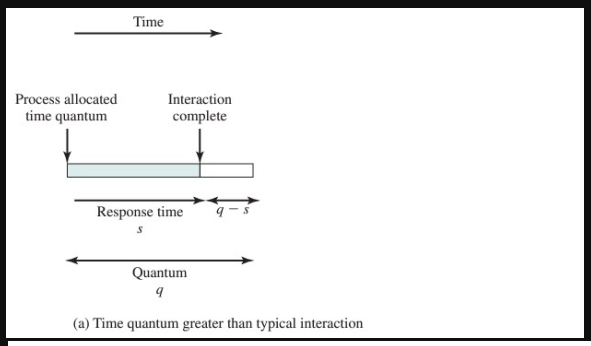
- ↳ short process may have to wait long time before executing
- ↳ good for computation-intensive processes b/c CPU-bound ones favoured
  - I/O processes block soon, then wait until all CPU-bound processes in front complete

round robin (RR)
- ↳ uses preemption based on clock interrupts gen at periodic intervals
- ↳ aka time slicing
- ↳ e.g.

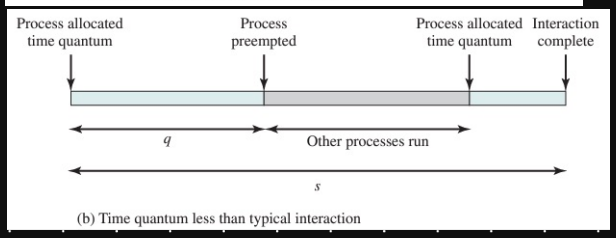


↳ good overall strat

↳ design issue is **time quantum** / slice len
- if too short, short processes will move thru quickly, but lots of overhead in dispatching
- effect of len:

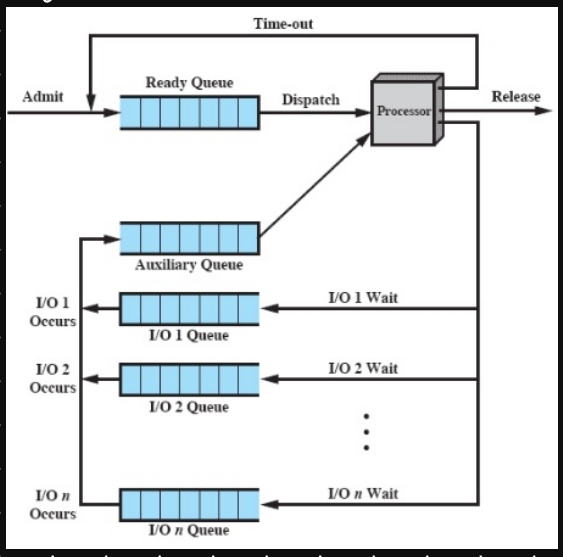


(a) Time quantum greater than typical interaction



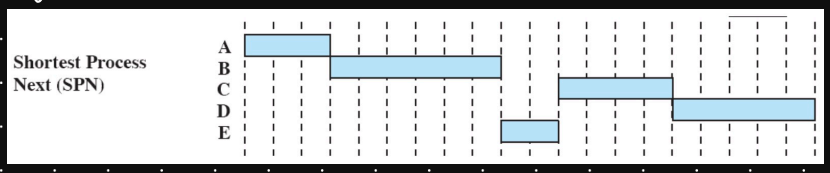
(b) Time quantum less than typical interaction

↳ CPU-bound processes receive unfair share b/c I/O processes probably won't use entire time slice
- use **virtual RR (VRR)**, where processes unblocked by I/O event are moved to **auxiliary queue** (raises priority of I/O-heavy processes)
  - → when process dispatched from aux queue, max time it runs is time quantum − total time running since it was last selected from main queue
  - → e.g.



shortest process next (SPN): process w/ shortest expected processing time selected next
↳ nonpreemptive policy
↳ short process jumps ahead of longer processes

↳ e.g.



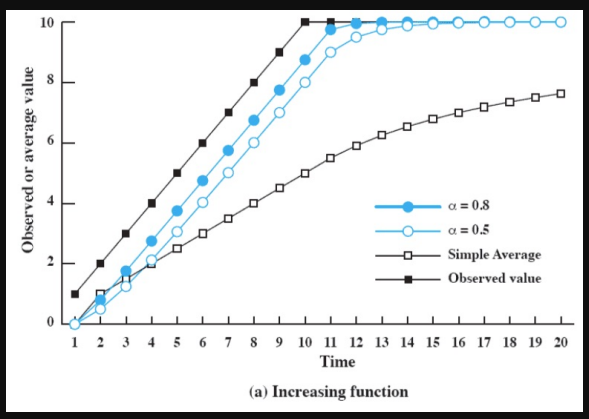
↳ high thruput

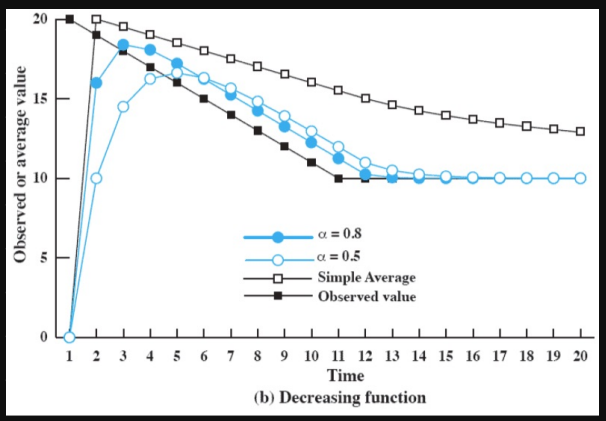
↳ possibility of starvation + reduced predictability for longer processes

↳ if estimated time for process is incorrect, OS may abort it

↳ limitation is that we must know service time beforehand

- can estimate w/ **linear avg**: $S_{n+1} = \frac{1}{n} \sum_{i=1}^{n} T_i = \frac{1}{n} T_n + \frac{n-1}{n} S_n$
  - → T_i = processor execution time for i^{th} instance of process
    - total time for batch job
    - burst time for interactive job
  - → S_i = predicted val for i^{th} instance
    - S_1 not calc
  - → each term is given equal weight
- better estimate w/ **exponential avg**: $S_{n+1} = \alpha T_n + (1-\alpha) S_n$
  - → α is constant weight factor st $0 < \alpha < 1$
  - → greater weight to more recent instances
- e.g. use of exponential avg



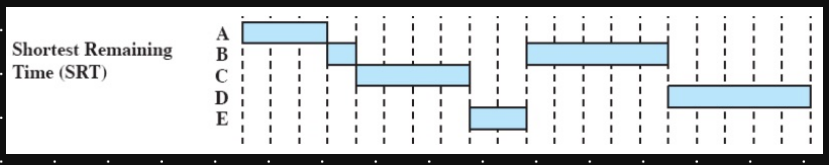


shortest remaining time (SRT): preemptive version of SPN

↳ must estimate process time

↳ starvation of long processes

↳ e.g.



↳ advantages:

- better turnaround time than SPN b/c short processes given immediate preference

- no bias for long processes like FCFS

↳ disadvantage:

- elapsed service time must be recorded, which creates more overhead

highest response ratio next (HRRN): choose next process w/ greatest ratio to minimize turnaround time

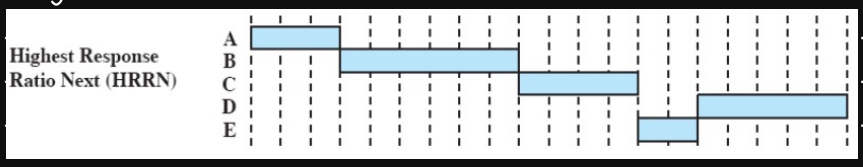
↳ $R = \frac{w + s}{s}$

- R = response ratio
- w = time spent waiting for processor
- s = expected service time

↳ accounts for age + short processes (i.e. both will have greater ratios)

↳ still need to know service time

↳ e.g.



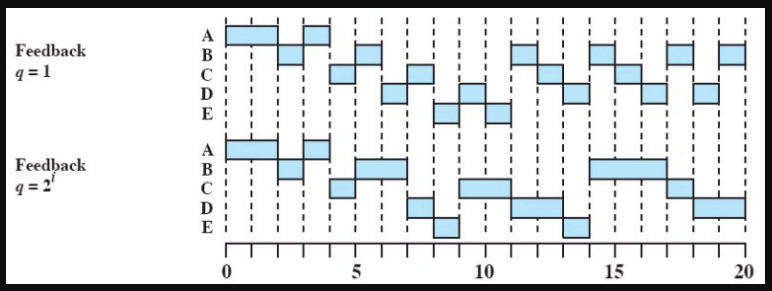
feedback-based scheduling

↳ every time process is preempted, it's demoted to next lower-priority queue

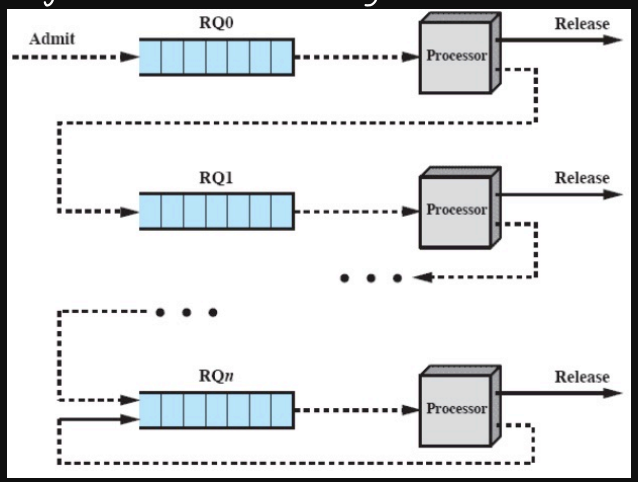
- each queue (except lowest) is FCFS
- lowest is RR
- penalizes jobs that have been running longer

↳ don't know remaining time process needs to execute

↳ e.g.



↳ e.g. feedback scheduling



summary of scheduling policies:

Table 9.3 Characteristics of Various Scheduling Policies

	FCFS	Round robin	SPN	SRT	HRRN	Feedback
Selection function	max[w]	constant	min[s]	min[s – e]	$\max\left(\frac{w+s}{s}\right)$	(see text)
Decision mode	Non-preemptive	Preemptive (at time quantum)	Non-preemptive	Preemptive (at arrival)	Non-preemptive	Preemptive (at time quantum)
Through-Put	Not emphasized	May be low if quantum is too small	High	High	High	Not emphasized
Response time	May be high, especially if there is a large variance in process execution times	Provides good response time for short processes	Provides good response time for short processes	Provides good response time	Provides good response time	Not emphasized
Overhead	Minimum	Minimum	Can be high	Can be high	Can be high	Can be high
Effect on processes	Penalizes short processes; penalizes I/O bound processes	Fair treatment	Penalizes long processes	Penalizes long processes	Good balance	May favor I/O bound processes
Starvation	No	No	Possible	Possible	No	Possible

↳ all treat collection of processes as single, homogeneous pool (other than priority) in multiuser system, a user's apps run as collection of processes / threads

↳ user is only concerned abt their apps

↳ fair-share scheduling: make scheduling decisions based on process sets

↳ e.g.

Time	Process A Priority	Process A Process CPU count	Process A Group CPU count	Process B Priority	Process B Process CPU count	Process B Group CPU count	Process C Priority	Process C Process CPU count	Process C Group CPU count
0	60	0 1 2 • • 60	0 1 2 • • 60	60	0	0	60	0	0
1	90	30	30	60	0 1 2 • • 60	0 1 2 • • 60	60	0	0 1 2 • • 60
2	74	15 16 17 • • 75	15 16 17 • • 75	90	30	30	75	0	30
3	96	37	37	74	15	15 16 17 • • 75	67	0 1 2 • • 60	15 16 17 • • 75
4	78	18 19 20 • • 78	18 19 20 • • 78	81	7	37	93	30	37
5	98	39	39	70	3	18	76	15	18
	Group 1			Group 2					

Colored rectangle represents executing process

- each group has weight 0.5
- B + C in same group

↳ formulas for process j in group k:

$$CPU_j(i) = \frac{CPU_j(i-1)}{2}$$

$$GCPU_k(i) = \frac{GCPU_k(i-1)}{2}$$

$$P_j(i) = Base_j + \frac{CPU_j(i)}{2} + \frac{GCPU_k(i)}{4W_k}$$

NOTE
All results from eqns are floored

- $CPU_j(i)$ = processor utilization by process j thru interval i
- $GCPU_k(i)$ = processor utilization by group k thru interval i
- $P_j(i)$ = priority of process j at start of interval i
- $Base_j$ = base priority of process j
- W_k = weighting assigned to group k
  - → $0 < W \leq 1$
  - → $\sum_k W_k = 1$

traditional UNIX scheduling uses multilvl feedback using round robin within each priority queue.

↳ if running process doesn't block/complete within 1s, it's preempted

↳ priorities are recomputed once per s

↳ base priority divides all processes into fixed bands of priority lvls
- in dec order of priority (i.e. high → low), bands are:
  - → swapper
  - → block I/O device control
  - → file manipulation
  - → char I/O device control
  - → user processes

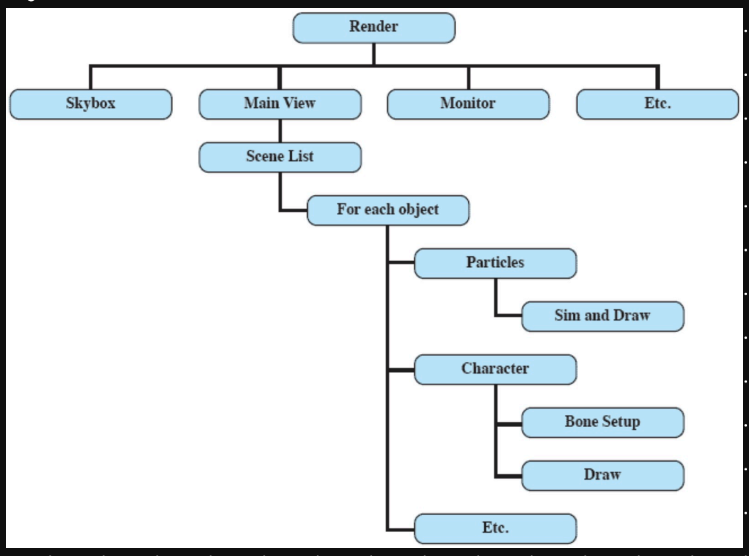
MULTIPROCESSOR AND REAL-TIME SCHEDULING

MULTIPROCESSOR SCHEDULING

- classifications of multiprocessor systems:
  - ↳ loosely coupled / distributed multiprocessor
    - aka cluster
    - each processor has own mem + I/O channels
    - classical distributed system
  - ↳ fcnally specified processors
    - controlled by master, general-purpose processor
    - e.g. I/O processor, FPGA (Field Programmable Gate Arr) cards
  - ↳ tightly coupled multiprocessors
    - processors share main mem
    - controlled by OS
    - e.g. multi-core, SMP, cell processor, FPGA in processor socket
- types of parallelism can be separated into categories based on granularity of synchronization:
  - ↳ independent: no sync among processes
    - e.g. time-sharing system
    - similar to uniprocessor system but avg response time is less
  - ↳ coarse + very coarse-grained
    - coarse: multiprocessing of concurrent processes in multiprogramming env
    - very coarse: distributed processing across network nodes to form single computing env
    - little sync among processes
    - handled as concurrent processes on multiprogrammed uniprocessor + translated/supported on multiprocessor w/ little change
  - ↳ medium-grained: parallel processing / multitasking within single app
    - threads usually interact frequently + share data so they require sync
    - scheduling decisions regarding 1 thread may affect performance of entire app
  - ↳ fine-grained: parallelism inherent in single ins stream
    - highly parallel apps
  - ↳

Grain Size	Description	Synchronization Interval (Instructions)
Fine	Parallelism inherent in a single instruction stream.	<20
Medium	Parallel processing or multitasking within a single application	20-200
Coarse	Multiprocessing of concurrent processes in a multiprogramming environment	200-2000
Very Coarse	Distributed processing across network nodes to form a single computing environment	2000-1M
Independent	Multiple unrelated processes	not applicable

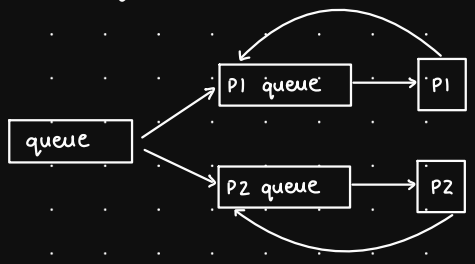
e.g. thread struct for rendering module:



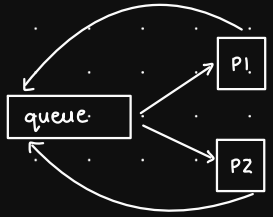
multiprocessor scheduler needs to consider:
- ↳ asgnt of processes to processors
- ↳ use of multiprogramming on individual processors
- ↳ acc dispatching of process

in assigning processes to processors, must consider 2 architectural styles of multiprocessors:
- ↳ uniform
  - assign processes to dedicated processor
  - migrate processes btwn processors
- ↳ heterogeneous
  - requires special software

assuming uniform multiprocessors, options for asgnt of processes to processors:
- ↳ static asgnt

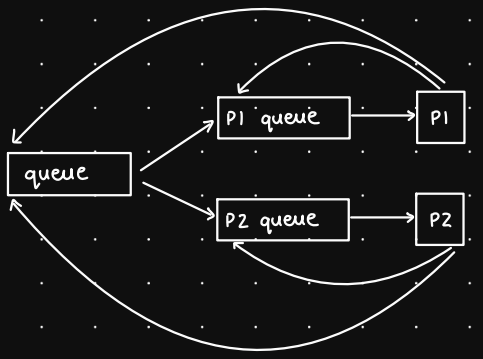


  - low system overhead b/c decision made once
  - permits group/gang scheduling
  - 1 processor can be idle while another has backlog
- ↳ process migration



- queue is shared resource so other processors can't access it while I is
- potential high overhead due to migration
  - → process A was in P1 before, then gets assigned to P2 + has a **cold cache** (i.e. P2 contains no useful data for A + results in many cache misses)

↳ **dynamic load balancing**



- have static asgnt but migrate processes sometimes

approaches for assigning processes to processors:

↳ **master/slave**: key kernel fcns always run on particular processor (i.e. master)
- master does scheduling
- slave sends service request to master + waits for result
- advantages:
  - → simple design
  - → similar to uniprocessor system
- disadvantages:
  - → failure of master brings down whole system
  - → master becomes performance bottleneck (i.e. restricts entire system)

↳ **peer**: kernel can execute on any processor
- each processor does self-scheduling
- complicates OS
  - → e.g. make sure 2 processors don't choose same process

on deciding if **multiprogramming** is needed for each processor:

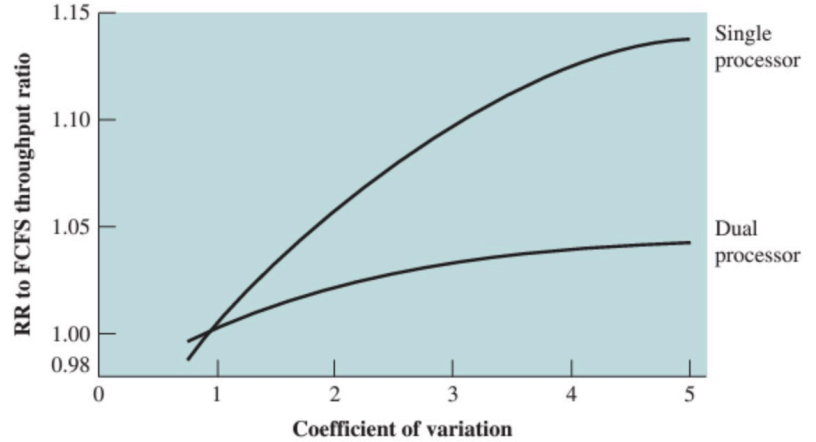
↳ when dealing w/ indep or coarse-grained sync granularity, still needed

↳ when dealing w/ medium-grained sync granularity, multiprogramming may not be needed on each processor
- low system overhead + less complexity
- not as important that every processor is as busy as possible

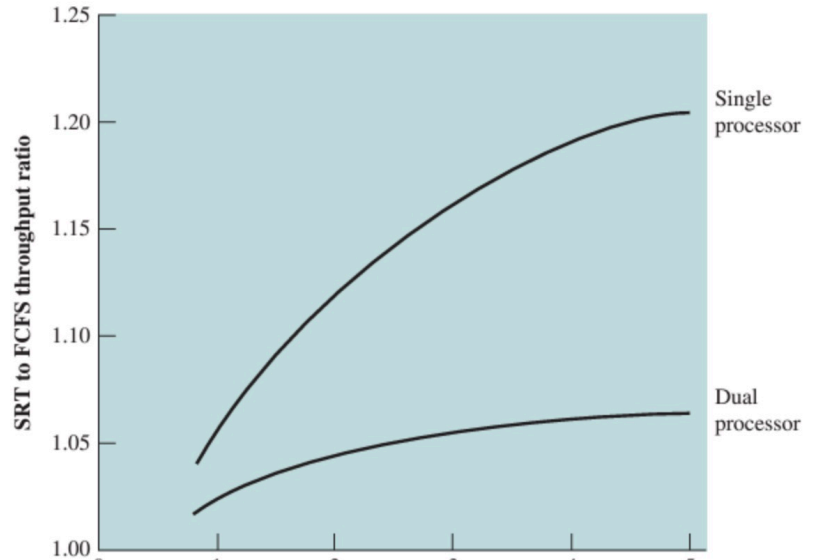
↳ in-app concurrency is still necessary

for **process dispatching**, some sophisticated concepts to guide scheduling decisions are unnecessary + counter productive

↳ specific scheduling discipline is less important w/ multiple processors
- e.g.



(a) Comparison of RR and FCFS



(b) Comparison of SRT and FCFS

Figure 10.1 **Comparison of Scheduling Performance for One and Two Processors**

→ C_s (coeff. of variation) measures variability in service times

THREAD SCHEDULING

- using threads, concept of execution is separated from resource ownership
- app consists of multiple cooperating, concurrently-executing threads
  - ↳ in uniprocessor, threads:
    - are program struct aid
    - overlap I/O w/ processing
    - have low management overhead (compared to multiprogramming)
  - ↳ in multiprocessor, threads provide true parallelism
- approaches for multiprocessor thread scheduling:
  - ↳ load sharing: processes aren't assigned to particular processor
    - diff from load balancing, where work is alloced on more permanent basis

- global queue + processor picks process
- load is distributed evenly across processors
- common scheduling methods
  - → FCFS
  - → smallest #threads 1st
  - → preemptive smallest #threads 1st
- advantages:
  - → no processor is idle when there's avail processes in queue
  - → no centralized scheduler needed
  - → transparent for developer
- disadvantages:
  - → central queue requires mutual exclusion
    - may become bottleneck
  - → preemptive threads unlikely to resume execution on same processor
    - cache misses
  - → unlikely threads of same program will get processors at same time so if they require high deg of coordination, they'll need many process switches

↳ **gang scheduling**: set of threads for 1 process is scheduled to run on set of processors at same time

- threads often need to sync w/each other
- advantages:
  - → useful for apps where performance severely degrades when any part of app isn't running
  - → minimizes process switches
- disadvantages:
  - → programmer must decide which threads are in a gang (i.e. management overhead)
  - → lead to inefficient use of multiprocessor
    - soln is to use weights in scheduling algo
    - e g

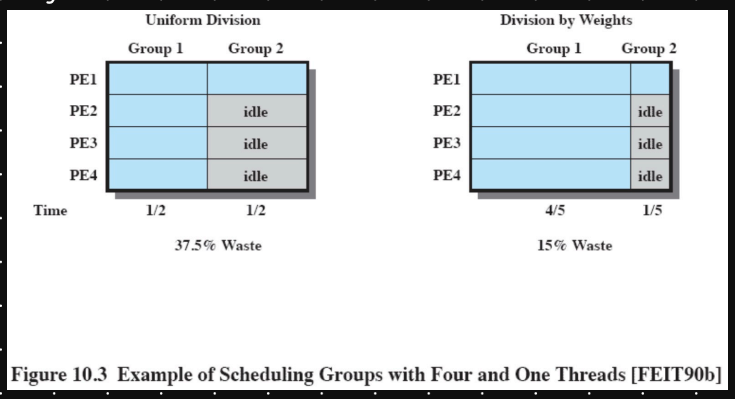


Figure 10.3 Example of Scheduling Groups with Four and One Threads [FEIT90b]

↳ **dedicated processor asgnt**: dedicate group of processors to app for its duration

- i.e. each thread of app gets processor until app runs to completion

- no multiprogramming & extremely wasteful
- advantages:
  - → in highly parallel system w/ large # processors (e.g. 1000), processor utilization isn't as important
  - → no process switching overhead
- e.g. running matrix multiplication & FFT on system w/ 16 processors

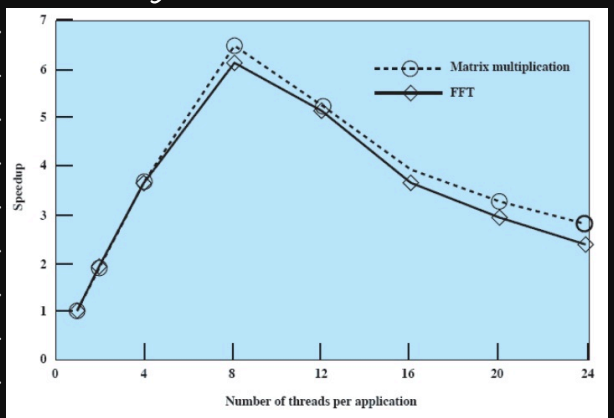


Figure 10.4 Application Speedup as a Function of Number of Threads

  - → performance worsens when # threads > 8 for both apps b/c that exceeds total # processors
    - more thread preemption & rescheduling
  - → effective strat is to limit # active threads to # processors

↳ **dynamic scheduling**: # threads in process are altered dynamically by app
- requires layer of indirection which maps computation tasks to threads
- when job requests & processors:
  - → if there's idle processors, use them to satisfy request
  - → otherwise, if job is new, take 1 processor away from any job w/ > 1 processors & give it to new job
  - → if request can't be satisfied, it's outstanding until a processor is avail or job rescinds request
- when processors get released:
  - → scan queue of unsatisfied requests
  - → assign 1 to each job w/ no processors
  - → scan list again & assign on FCFS basis

REAL-TIME SCHEDULING

real-time (RT) computing: correctness of system depends on logic behaviour & timing
↳ tasks/processes try to control/react to events in outside world

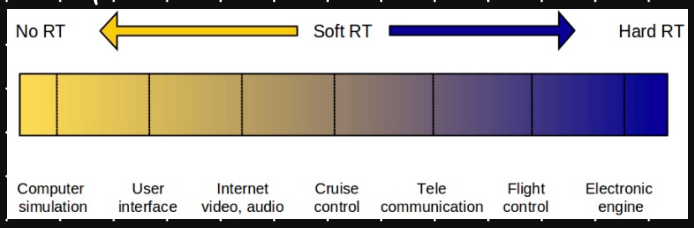
correct val at wrong time is **fault**

↳

Value	Timing	Result
wrong	late	fail
wrong	on time	fail
correct	late	fail
correct	on time	ok

↳ temporal constraints are a way to specify if val is on time

- **soft RT system**: response time usually specified as avg val
  - ↳ time is normally dictated by business/market
  - ↳ single late computation isn't significant to op of system, but many late arrivals will be
  - ↳ e.g. airline reservation system
- **hard RT system**: response time is specified as absolute val
  - ↳ time is normally dictated by env
  - ↳ tasks must always finish execution before their deadlines or msg can always be delivered within specified time interval
  - ↳ often are safety-critical apps
    - failure (i.e. missed deadline) can lead to loss of human life or severe economical dmg
- **firm RT system**: combo of both hard + soft ones
  - ↳ computation will have shorter soft requirement + longer hard requirement
  - ↳ e.g. ventilator where it must ventilate patient so many times within given time period
    - few sec delay is allowed, but no more
- RT spectrum:

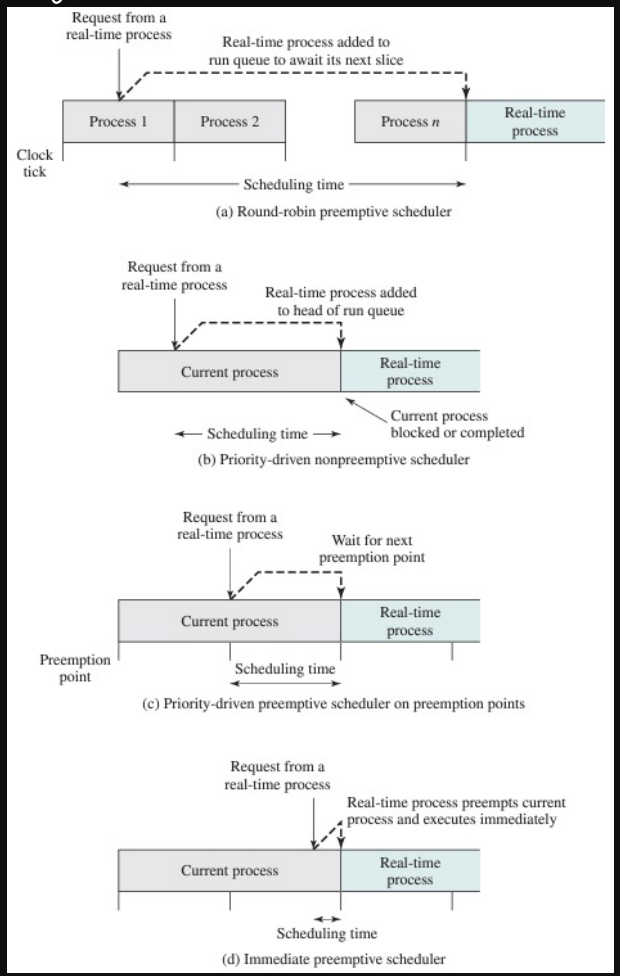


- **characteristics** of RT systems:
  - ↳ **deterministic**: ops are performed at fixed, predetermined times or within predetermined time intervals
    - i.e. how long OS delays before acknowledging interrupt
    - must have sufficient capacity to handle all requests within required time
  - ↳ **responsive**: how long OS takes to service interrupt after acknowledgment
    - includes:
      - → amt of time to begin execution of interrupt
      - → amt of time to perform ISR
      - → interference from interrupt nesting
  - ↳ **user control**: user specifies priority, importance, + timing
    - usually user has little control in non-RTOS
    - manually control mem management
      - → e.g. locking pgs, specifying resource demands
    - manually control I/O algos
      - → e.g. disk transfer algos
  - ↳ **reliable**: degradation of performance may have catastrophic consequences

- e.g. safety-critical systems
  - ↳ **fail-soft op**: ability of system to preserve as much capability + data as possible after fail
    - graceful degradation
    - e.g. if it's impossible to meet all deadlines, meet those of most critical + highest-priority tasks

features of RTOS:
- ↳ fast process/thread switch
- ↳ small size
  - can run on cheaper processor b/c it uses less resources
- ↳ ability to respond external interrupts quickly
- ↳ multitasking w/interprocess communication tools (e.g. semaphores, signals, events)
- ↳ use of special sequential files that can accumulate data at fast rate
- ↳ preemptive scheduling based on priority
- ↳ minimization of intervals where interrupts are disabled
- ↳ delay tasks for fixed amt of time (i.e. no starvation)
- ↳ special alarms + timeouts

e.g. ways to schedule RT process



unacceptable

still too slow for many RT apps.

↳ **immediate preemptive scheduler** is only acceptable method

types of RT scheduling algos.

↳ **static table-driven**: determines before run time when task begins execution
- applicable to periodic tasks
- e.g. EDF

↳ **static priority-driven preemptive**: static analysis used to assign priorities to tasks
↑ traditional priority-driven scheduler is used
- e.g. RM

↳ **dynamic planning-based**: feasibility determined at run time

↳ **dynamic best effort**: no feasibility analysis ↑ systems tries to meet all deadlines
- aborts any started process whose deadline is missed
- used by many commercially avail RT systems

RT apps aren't concerned w/speed but w/completing tasks.

task info that's used by RT system:

↳ rdy time
↳ starting deadline
↳ completion deadline
↳ processing time
↳ resource requirements
↳ priority
↳ subtask scheduler

e.g. execution of 2 **periodic tasks**

Table 10.2 Execution Profile of Two Periodic Tasks

Process	Arrival Time	Execution Time	Ending Deadline
A(1)	0	10	20
A(2)	20	10	40
A(3)	40	10	60
A(4)	60	10	80
A(5)	80	10	100
•	•	•	•
•	•	•	•
•	•	•	•
B(1)	0	25	50
B(2)	50	25	100
•	•	•	•
•	•	•	•
•	•	•	•

T=(e,p), A=(10,20), B=(25,50)

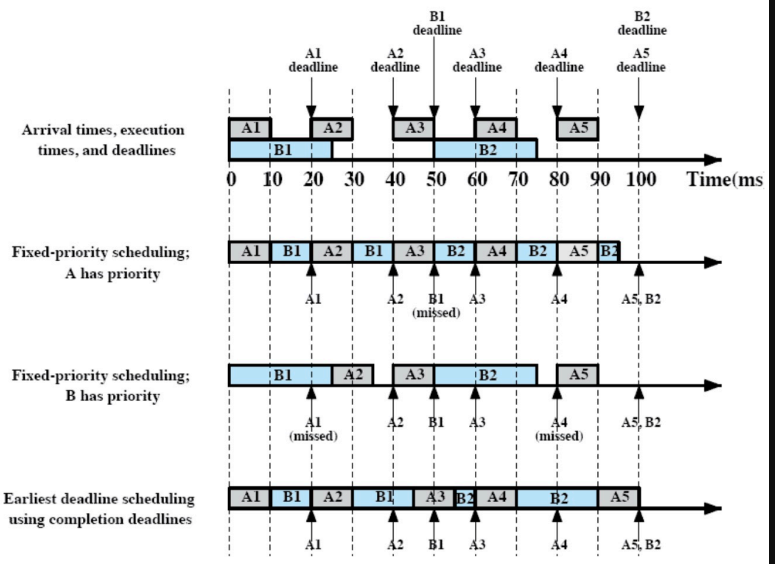


Figure 10.6 Scheduling of Periodic Real-time Tasks with Completion Deadlines (based on Table 10.2)

↳ earliest deadline first (EDF) scheduling is successful

- always execute task w/earliest deadline
- optimal if system supports preemption (i.e. can schedule CPU utilization of 1)

e.g. execution of 5 aperiodic tasks

Table 10.3 Execution Profile of Five Aperiodic Tasks

Process	Arrival Time	Execution Time	Starting Deadline
A	10	20	110
B	20	20	20
C	40	20	50
D	50	20	90
E	60	20	70

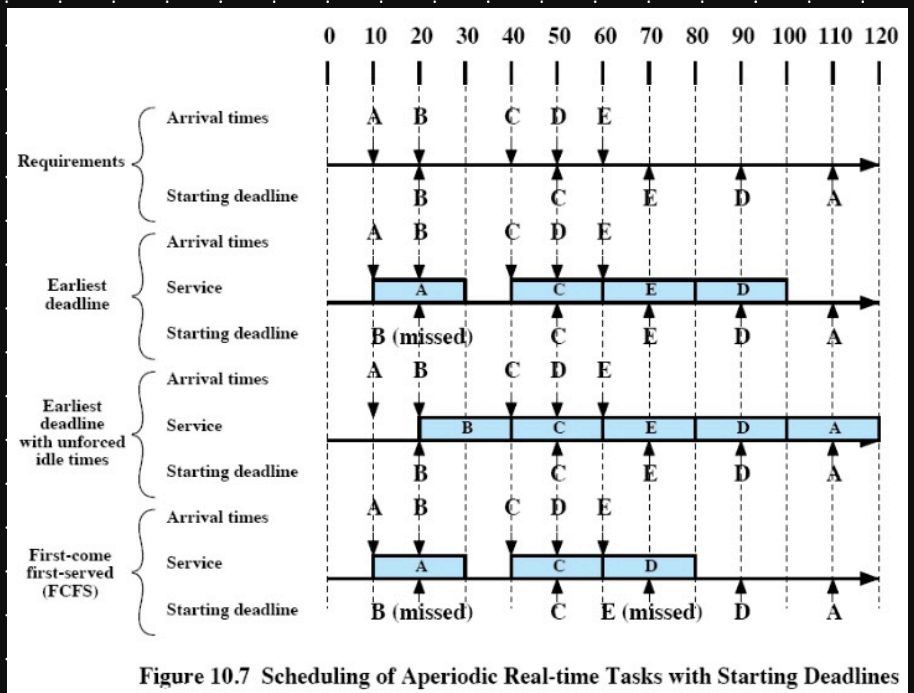
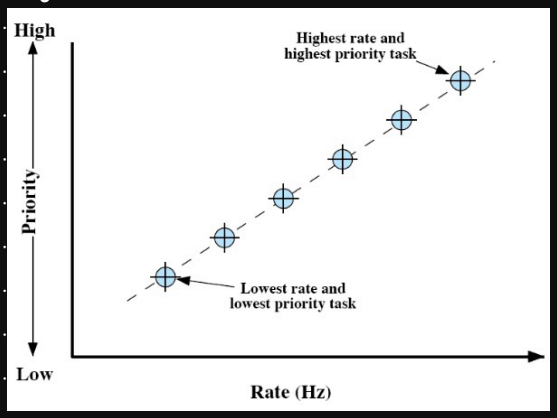


Figure 10.7 Scheduling of Aperiodic Real-time Tasks with Starting Deadlines

↳ EDF w/unforced idle times is successful
- if deadlines known ahead of time, always schedule task w/earliest deadline 1st + let it run to completion
  → might result in idle time if task isn't rdy

rate monotonic (RM) scheduling: tasks w/ shorter periods have higher priority
↳ plot of priorities as fcn of their rate + we get monotonically inc fcn
- e.g.



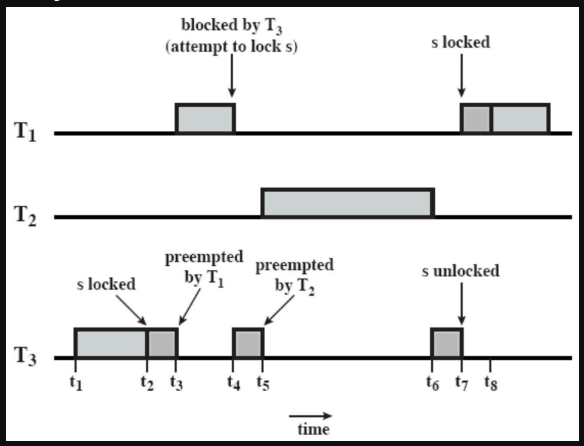
↳ max processor utilization is 69.3%

industry prefers RM over EDF b/c:
↳ predictions for deadlines may be inaccurate
↳ performance diff is small in practice
- 30% gain doesn't make much diff

↳ only some parts of system are hard RT critical
- use portion for soft RT tasks

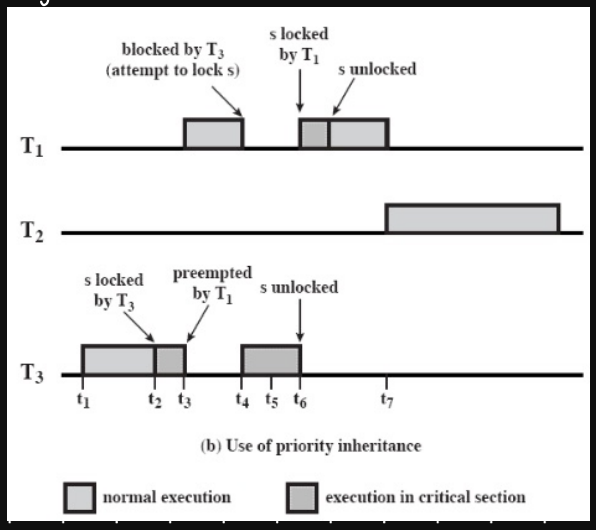
↳ easier to achieve stability b/c RM is more predictable in overload situations
↳ might not know deadlines, but always know periods

priority inversion: circumstances within system force higher priority task to wait for lower priority task
↳ can occur in any priority-based preemptive scheduling scheme
↳ unbounded priority inversion: duration of priority inversion depends on unpredictable actions of unrelated tasks
- e.g.



→ t_1: T_3 begins executing
→ t_2: T_3 locks semaphore s + enters critical section
→ t_3: T_1 (higher priority than T_3) preempts T_3 + begins executing
→ t_4: T_1 attempts to enter critical, gets blocked by s, then T_3 resumes
→ t_5: T_2 (higher priority than T_3) preempts T_3 + begins executing
→ t_6: T_2 is suspended for reason unrelated to T_1 + T_3, then T_3 resumes
→ t_7: T_3 leaves critical section + unlocks s, then T_1 preempts T_3 + enters critical section

- 2 solns to avoid unbounded priority inversion:
  - ↳ **priority inheritance**: lower-priority task inherits priority of any higher-priority task pending on resource they share
    - e.g.



  - ↳ **priority ceiling**: give priority to each resource st its priority is 1 lvl higher than priority of its highest-priority user
    - resource's priority is dynamically assigned to task using it

I/O MANAGEMENT AND DISK SCHEDULING

I/O DEVICES

- categories of I/O devices:
  - ↳ human-readable
    - communicates w/ user
    - e.g. printers, terminals
  - ↳ machine-readable
    - communicates w/ electronic equipment
    - e.g. disk drives, USB keys, sensors, controllers, actuators
  - ↳ communication
    - communicates w/ remote devices
    - e.g. digital line drivers, modems
- key diffs btwn I/O devices:
  - ↳ data rate: may have several orders of magnitude in diff btwn data transfer rates
    - e.g.

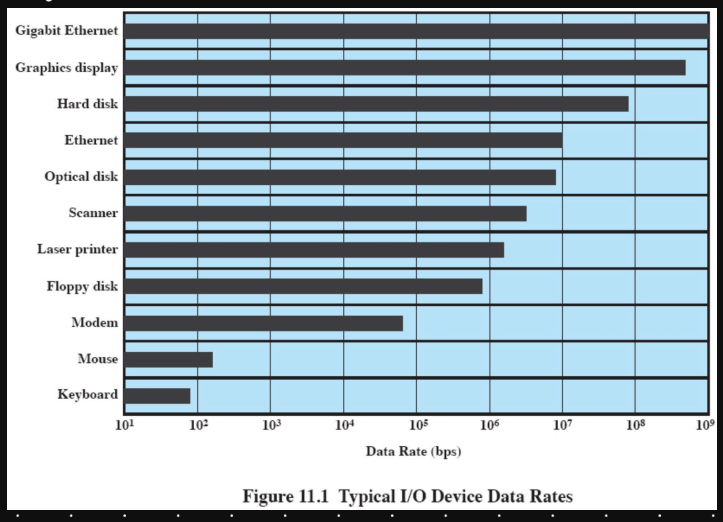


Figure 11.1 Typical I/O Device Data Rates

  - ↳ application: use of device affects management software
    - e.g. disk used to store files needs file management software
    - e.g. disk used to store VM pgs needs special hardware + software
    - e.g. terminal used by system admin may have higher priority than regular user's
  - ↳ complexity of control
    - simple 1 line on/off
    - time dependent interaction
      - → e.g. duty cycle of stepper motors

- complex, protocol-based interaction (most communication devices)

↳ **unit of transfer**: may be stream of bytes for terminal or larger blocks for disk

↳ **data rep**
- encoding schemes
- diffs in parity

↳ **error conds**: devices respond to errors differently

I/O FUNCTION

techniques for performing I/O:

↳ **programmed I/O**: process busy-waits for op to complete

↳ **interrupt-driven I/O**: processor issues I/O cmd + continues executings ins

↳ **direct mem access (DMA)**: DMA module controls exchange of data btwn main mem + I/O device
- processor only interrupted after entire block is transferred

evolution of I/O fcn:

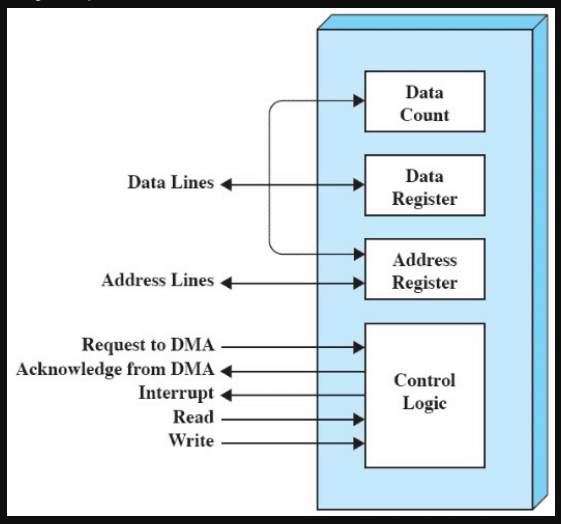
1) processor directly controls peripheral
2) controller / I/O module added
   - programmed I/O
   - processor doesn't handle details of external devices
3) controller / I/O module w/interrupts
4) DMA
   - blocks of data moved into main mem w/o involving processor
   - processor involved only at start + end
5) I/O module is separate processor (i.e. channel)
6) I/O processor w/ own local mem
   - i.e. it's a computer in its own right

} I/O channel

DMA module mimics processor + takes control of system bus to directly transfer data to/from mem

↳ when transfer complete, DMA module sends interrupt to processor

↳ e.g. typical DMA block diagram



- addr reg stores starting loc in mem to read/write
- data cnt reg stores # words to be read/written

types of DMA configs:

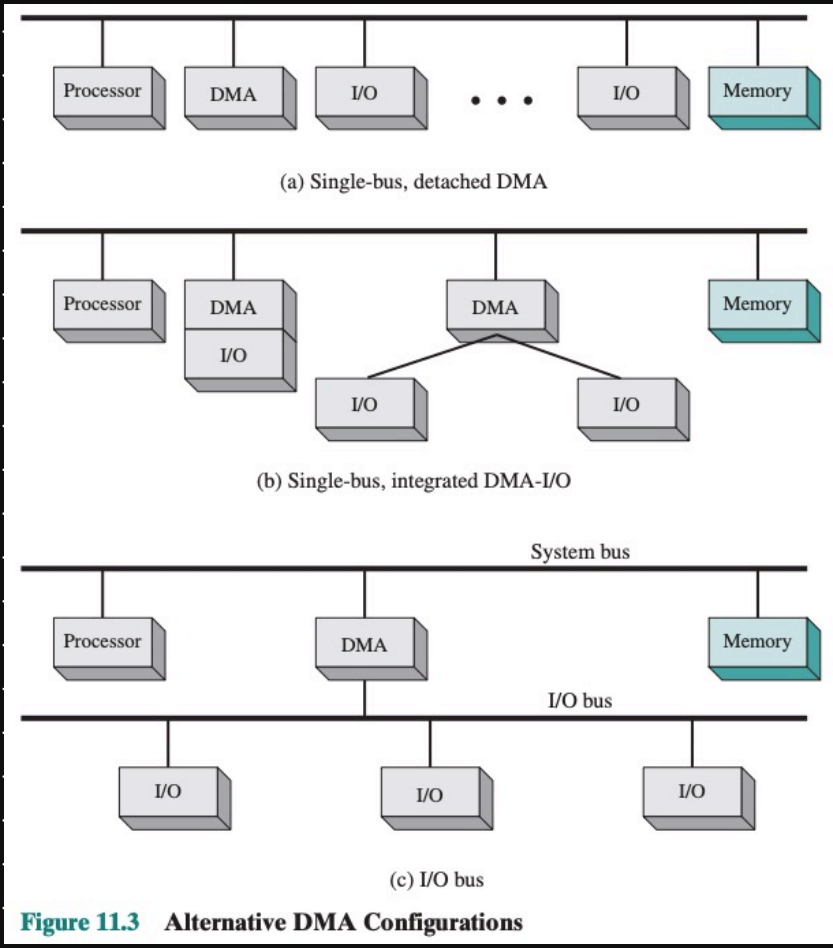


Figure 11.3 Alternative DMA Configurations

↳ a) is inefficient b/c each word transfer takes 2 bus cycles (transfer request + transfer)

↳ b) integrates DMA + I/O fcns to reduce # bus cycles

↳ c) reduces # I/O interfaces in DMA module + provides easier expansion

OS DESIGN ISSUES

2 objectives in designing I/O ops:

↳ efficiency
- most I/O devices are very slow compared to main mem + I/O can't keep up w/ processor thruput
- multiprogramming allows interleaving of I/O + processing
  - → swapping is I/O op itself

↳ generality
- desirable to handle all I/O devices in uniform manner
- hide most details of device I/O in lower-lvl routines

logical struct of I/O:

↳ logical I/O: treats device as logical resource w/no concerns of details of control

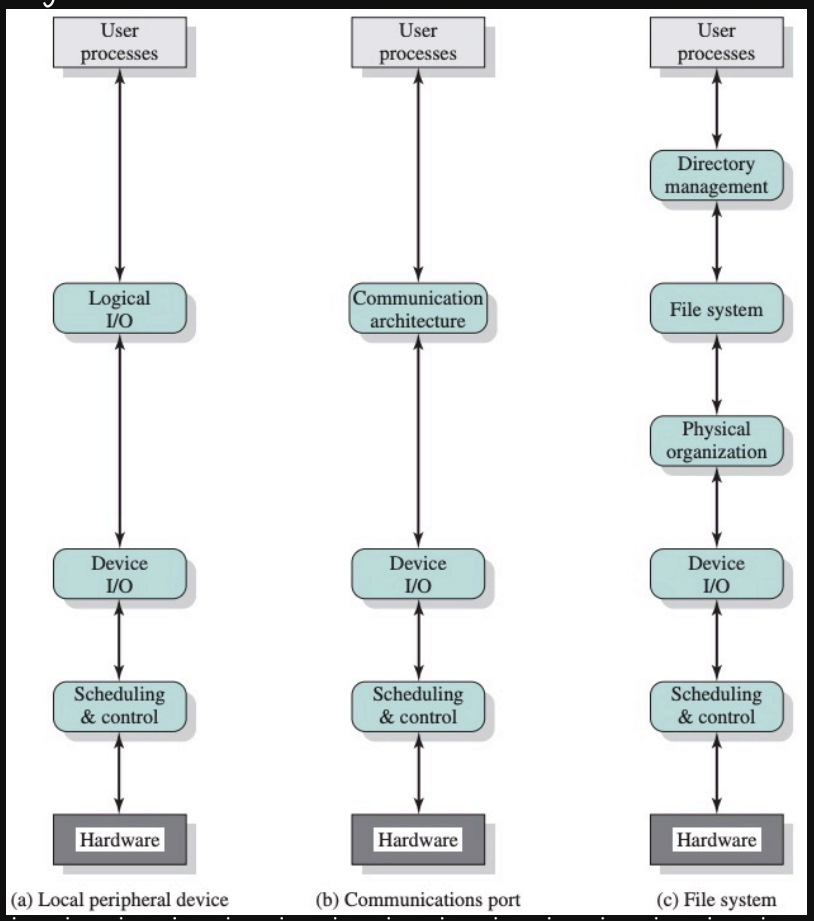
- manages I/O fcns on behalf of user process
- provides open, close, read, write, etc.

↳ **device I/O**: converts ops + data into I/O ins, channel cmds, + controller orders

↳ **scheduling + control**: queuing + scheduling of ops
- handles interrupts
- checks I/O status

↳ for secondary storage devices w/ file management system:
- **directory management**: convert symbolic file names to ids that ref file
  → manage add, delete, + organize ops
- **file system**: logical struct of files + user ops
  → provides open, close, read, write, etc.
- **physical organization**: convert file addrs into physical locs

↳ e.g.



(a) Local peripheral device (b) Communications port (c) File system

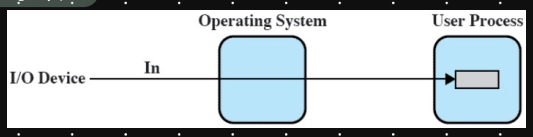
- in communications port, communications architecture may have multiple layers
  → e.g. TCP/IP

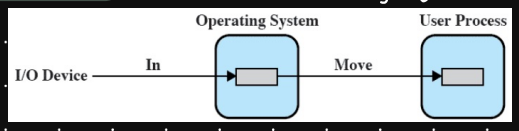
I/O BUFFERING

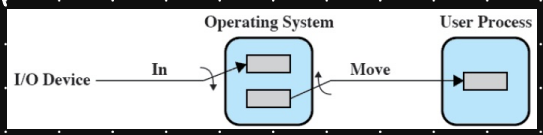
reasons for **buffering**:

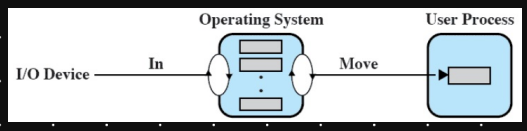
↳ process must wait for I/O to complete before proceeding

↳ interferes w/ swapping decisions b/c virtual locs assigned for read/write must remain in main mem during block transfer

- impossible to swap out process completely
  - ↳ risk of **single-process deadlock**:
    - process issues I/O cmd + gets suspended
    - process swapped out before I/O op
    - process blocked on I/O event
    - I/O blocked on waiting for process to be swapped in
    - prevent by locking user mem involved in I/O op before I/O request is issued
- OS assigns buffer in main mem for I/O request
- 2 types of I/O devices:
  - ↳ **block-oriented**: input transfers made to buffer 1 block at a time
    - block moved to user space when needed
    - **read ahead** b/c user process can process 1 block while next is read in
    - swapping can occur b/c input is happening in system mem (not user mem)
    - OS keeps track of which system buffers are assigned to user processes
  - ↳ **stream-oriented**: transfers data 1 line at a time
    - user input from terminal is 1 line w/ carriage return signaling end of line
    - output to terminal is 1 line at a time
- types of I/O buffers:
  - ↳ **no buffer**
    - 

  - ↳ **single buffer**: simplest buffering system
    - 

  - ↳ **double buffer**: use 2 system buffers where process transfers data to/from 1 + OS empties/fills other 1
    - 

    - for stream-oriented input, producer/consumer model followed
  - ↳ **circular buffer**: ≥ 2 buffers used
    - 

    - good for rapid bursts of I/O
    - bounded buffer producer/consumer model
- block-oriented device can **seek** (i.e start reading/writing from certain pt in block) while stream-oriented device can't (e.g see what's been written in 5 secs into future)

↳ both can read, write, open, & close

DISK SCHEDULING

- to read/write, disk head must be at desired track & beginning of desired sector
- **access time**: sum of seek time & rotational delay
  - ↳ i.e. time it takes to get into pos to read/write
  - ↳ **seek time**: time to pos head at desired track
  - ↳ **rotational delay / latency**: time for start of sector to reach head
  - ↳ can be huge diff btwn sequential vs. random read access
    - elim seek time for all sequential reads after finding pos of 1st read
- **timing** of disk I/O transfer:

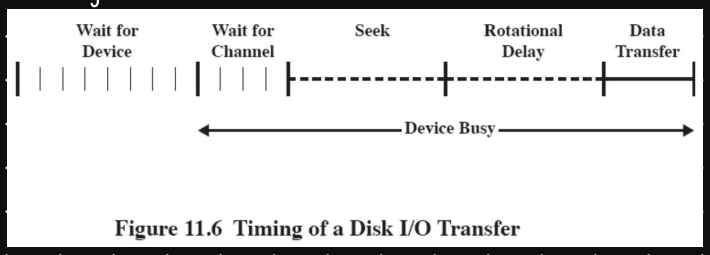


Figure 11.6 Timing of a Disk I/O Transfer

  - ↳ seek time is only thing OS can improve
- for single disk, there's several I/O requests
  - ↳ if selected randomly, seek time will degrade performance
- disk scheduling policies:
  - ↳ **first-in, first-out (FIFO)**
    - process requests sequentially
    - fair to all processes
    - if too many, approaches random scheduling in performance
  - ↳ **priority-based**
    - goal isn't to optimize disk use, but meet other objectives
    - e.g. short batch jobs have higher priority so there's good interactive response time
    - e.g. EDF
    - can lead to user countermeasures
      - → e.g. split job into smaller pieces
    - can lead to starvation for longer jobs
  - ↳ **last-in, first-out (LIFO)**
    - good for transaction processing systems
      - → giving device to most recent users leads to little/no arm movement going thru sequential file
    - can lead to starvation b/c job may nvr be at head of queue
  - ↳ **shortest service time first (SSTF)**
    - select request that takes least arm movement from arm's curr pos
    - always choose min seek time
    - tiebreaker needed for equal dists in opp dirs
    - can lead to starvation of requests far away if new closer requests keep coming

↳ **SCAN**

- arm moves in 1 dir only (inc / dec track #) satisfying all outstanding requests until it reaches last track in that dir, then reverse
- prevents starvation of older requests
- similar performance as SSTF
- biased against most recently traversed area b/c it doesn't change dir
  → doesn't exploit principle of locality as well as SSTF
- favours requests for tracks nearest to innermost & outermost tracks & latest-arriving jobs

↳ **C-SCAN**

- restricts scanning to 1 dir only
- when last track of that dir is visited, arms returns to opp end of disk & restarts
- deterministic
- reduces delay for new requests

↳ **N-step-SCAN**

- segments disk request queue into subqueues of length N
- subqueues processed 1 at a time using SCAN
- new requests added to a queue that's not being currently processed

e.g.:

Assume track requests are 55, 58, 39, 18, 90, 160, 150, 38, 184 and the current track is 100:

FIFO		SSTF		SCAN		C-SCAN	
Nxt Trk	# Trks	Nxt Trk	# Trks	Nxt Trk	# Trks	Nxt Trk	# Trks
55	45	90	10	150	50	150	50
58	3	58	32	160	10	160	10
39	19	55	3	184	24	184	24
18	21	39	16	90	94	18	166
90	72	38	1	58	32	38	20
160	70	18	20	55	3	39	1
150	10	150	132	39	16	55	16
38	112	160	10	38	1	58	3
184	146	184	24	18	20	90	32
Avg:	55.3	Avg:	27.5	Avg:	27.8	Avg:	35.8

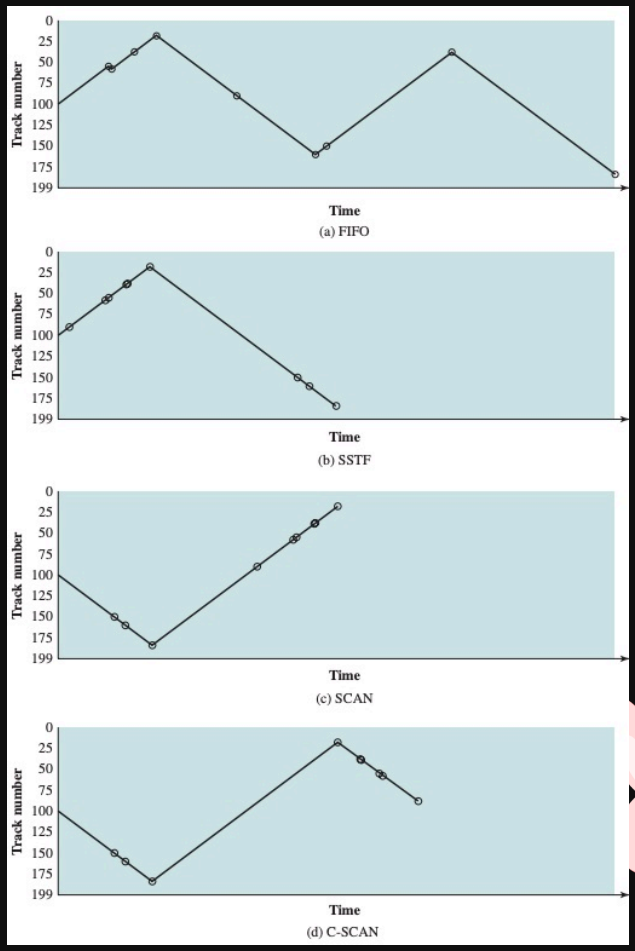


table summary of disk scheduling algos:

Name	Description	Remarks
Selection according to requestor		
Random	Random scheduling	For analysis and simulation
FIFO	First in first out	Fairest of them all
PRI	Priority by process	Control outside of disk queue management
LIFO	Last in first out	Maximize locality and resource utilization
Selection according to requested item		
SSTF	Shortest-service-time first	High utilization, small queues
SCAN	Back and forth over disk	Better service distribution
C-SCAN	One way with fast return	Lower service variability
N-step-SCAN	SCAN of *N* records at a time	Service guarantee
FSCAN	*N*-step-SCAN with *N* = queue size at beginning of SCAN cycle	Load sensitive

RAID

RAID (redundant arr of indep disks) is std scheme for multiple-disk db design
- ↳ organize data st redundancy can be added to improve reliability
- ↳ 3 characteristics:
  1) RAID is set of physical disk drives viewed by OS as single logical drive
  2) data distributed across physical drives of arr w/ striping
     - → logical disk divided into strips that are mapped round robin to consecutive physical disks in RAID arr
     - → stripe: set of logical strips that map 1 strip to each arr member (i.e. 1 row)
  3) redundant disk capacity used to store parity info so recovery's possible
     - → not supported by RAID 0 + 1

e.g.

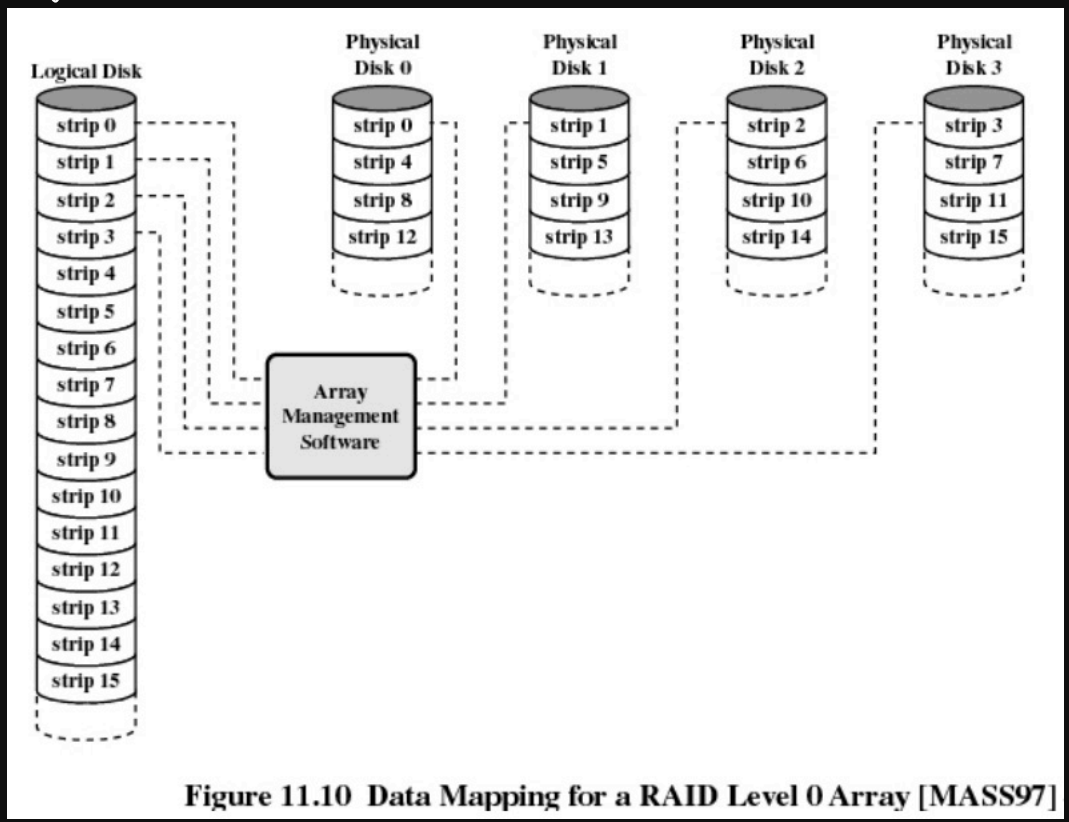
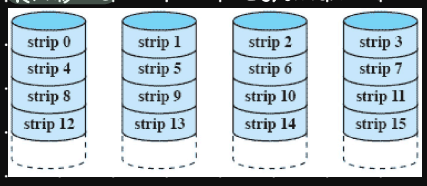


Figure 11.10 Data Mapping for a RAID Level 0 Array [MASS97]

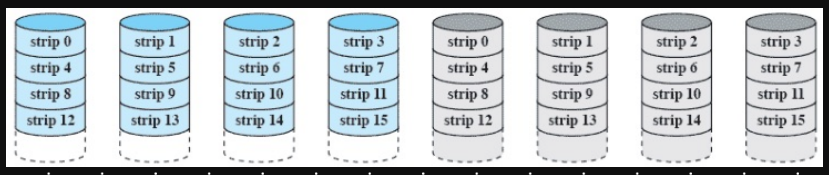
7 RAID categories:

↳ RAID 0 : non-redundant



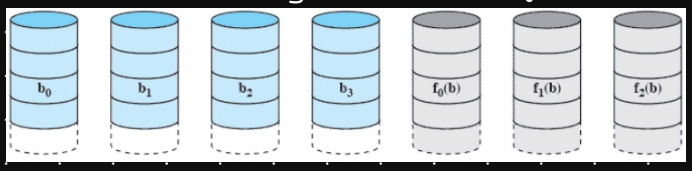
- high data transfer capacity
- lower reliability
  - → loss of 1 drive results in loss of data
- not true RAID

↳ RAID 1 : mirrored



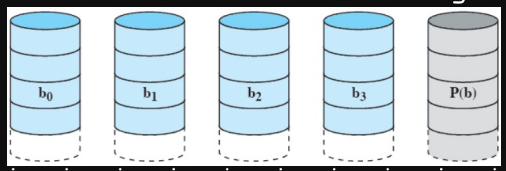
- redundancy thru simple duplication of data
- advantages:
  - → read requests serviced by either disk
  - → write requires both strips to be updated, but can be done in parallel
  - → recovery from single drive failure means using other drive
- disadvantage:
  - → cost of 2x disk space

↳ RAID 2: redundancy thru Hamming code



- all disks participate in disk transfers & spindles are synced so each head is in same pos on every disk
- error correcting code calced across corresponding bits on each disk
- single-bit errors recognized & corrected immediately
  - → no impact on access time
- on single write, all disks & parity disks accessed
- generally overkill soln
  - → not used b/c modern disks have high reliability

↳ RAID 3: bit-interleaved parity



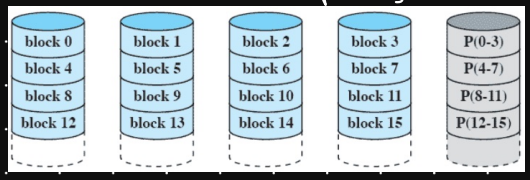
- 1 redundant disk

- parallel access w/ data distributed in **small strips** (bit / byte lvl)
- computes parity bit for set of individual bits in same pos on all disks
- e.g.

example: disks X0, X1, X2, X3 store data and X4 stores parity

- parity for i-th bit calculated as: $X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i)$
- let X1 fail, then add $x4(i) \oplus X1(i)$ to both sides as: $X1(i) = X4(i) \oplus X3(i) \oplus X2(i) \oplus X0(i)$

↳ **RAID 4**: block-lvl parity



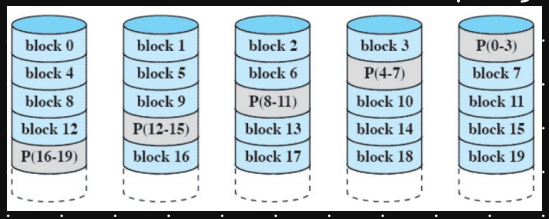
- uses **indep access arr** so each disk operates indep
  → separate I/O requests satisfied in parallel
- bit-by-bit parity strip calced across corresponding strips on each data disk
- penalty on small size writes
  → each strip write involves 2 reads of old user + parity strips + 2 writes of new user + parity strips
- e.g.

$$X4(i) = X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i) \quad (11.1)$$

After the update, with potentially altered bits indicated by a prime symbol:

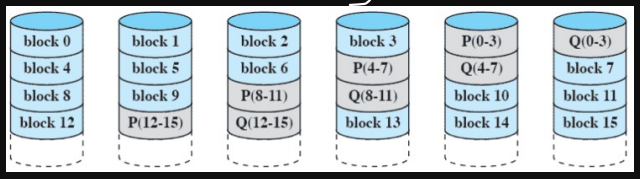
$$\begin{aligned} X4'(i) &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \\ &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \oplus X1(i) \oplus X1(i) \\ &= X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i) \oplus X1(i) \oplus X1'(i) \\ &= X4(i) \oplus X1(i) \oplus X1'(i) \end{aligned}$$

↳ **RAID 5**: block-lvl distributed parity



- similar to RAID 4 except parity strips distributed across all disks
- avoids potential I/O bottleneck of single parity disk of RAID 4
- bad when many incoming write requests hit same parity strip

↳ **RAID 6**: dual redundancy



- 2 diff parities carried + stored in separate blocks on diff disks
- N disk requirement needs N+2 disk config

- possible to regen data even if 2 disks fail
- extremely high availability
- substantial write penalty b/c 1 write affects 2 parity blocks

RAID summary table:

Table 11.4 RAID Levels

Category	Level	Description	Disks Required	Data Availability	Large I/O Data Transfer Capacity	Small I/O Request Rate
Striping	0	Nonredundant	N	Lower than single disk	Very high	Very high for both read and write
Mirroring	1	Mirrored	$2N$	Higher than RAID 2, 3, 4, or 5; lower than RAID 6	Higher than single disk for read; similar to single disk for write	Up to twice that of a single disk for read; similar to single disk for write
Parallel access	2	Redundant via Hamming code	$N + m$	Much higher than single disk; comparable to RAID 3, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
	3	Bit-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
Independent access	4	Block-interleaved parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 5	Similar to RAID 0 for read; significantly lower than single disk for write	Similar to RAID 0 for read; significantly lower than single disk for write
	5	Block-interleaved distributed parity	$N + 1$	Much higher than single disk; comparable to RAID 2, 3, or 4	Similar to RAID 0 for read; lower than single disk for write	Similar to RAID 0 for read; generally lower than single disk for write
	6	Block-interleaved dual distributed parity	$N + 2$	Highest of all listed alternatives	Similar to RAID 0 for read; lower than RAID 5 for write	Similar to RAID 0 for read; significantly lower than RAID 5 for write

Note: N, number of data disks; m, proportional to log N.

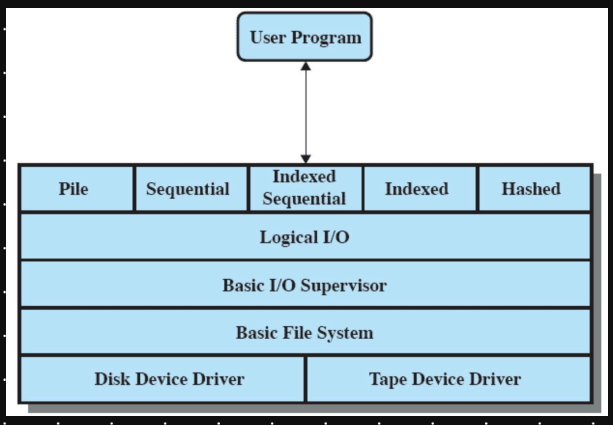
FILE MANAGEMENT

- reasons to have file system:
  - ↳ long-term storage
  - ↳ shareable data btwn processes
  - ↳ structured, hierarchic rltnships among files
- typical **file ops**:
  - ↳ create
  - ↳ delete
  - ↳ open
  - ↳ close
  - ↳ read
  - ↳ write
  - ↳ seek
- **field**: basic elmt of data
  - ↳ contains single val
    - e.g. student name
  - ↳ characterized by length + data type
- **record**: collection of related fields
  - ↳ treated as unit
- **file**: collection of similar records
  - ↳ treated as single entity
  - ↳ may restrict access
  - ↳ some are structured only in terms of fields, not records
- **database**: collection of related data
  - ↳ rltnships btwn elmts are explicit
- typical **record ops**:
  - ↳ Retrieve_All
  - ↳ Retrieve_One
  - ↳ Retrieve_Next
  - ↳ Retrieve_Previous
  - ↳ Insert_One
  - ↳ Delete_One
  - ↳ Update_One
  - ↳ Retrieve_Few
- **file management system (FMS)** is set of system software that provides services to users + apps in use of files
  - ↳ sole interface for users + apps to access files
  - ↳ management provided by OS
    - i.e. programmer doesn't need to develop FMS

objectives of FMS:
- ↳ meet data management needs + requirements of user
- ↳ guarantee data in file is valid
- ↳ optimize performance for user + system side
- ↳ provide I/O support for variety of storage device types
- ↳ minimize/elim potential for lost/destroyed data
- ↳ provide stdized set of I/O interface routines
- ↳ provide I/O support for multiple users

min requirements for FMS are that authorized users
- ↳ can create, delete, read, write, + modify files
- ↳ have controlled access to other users' files
- ↳ can control access permissions for own files
- ↳ can restructure their files
- ↳ can move data btwn files
- ↳ can back up + recover files in case of dmg
- ↳ can access files using symbolic names instead of # id

file system software architecture:

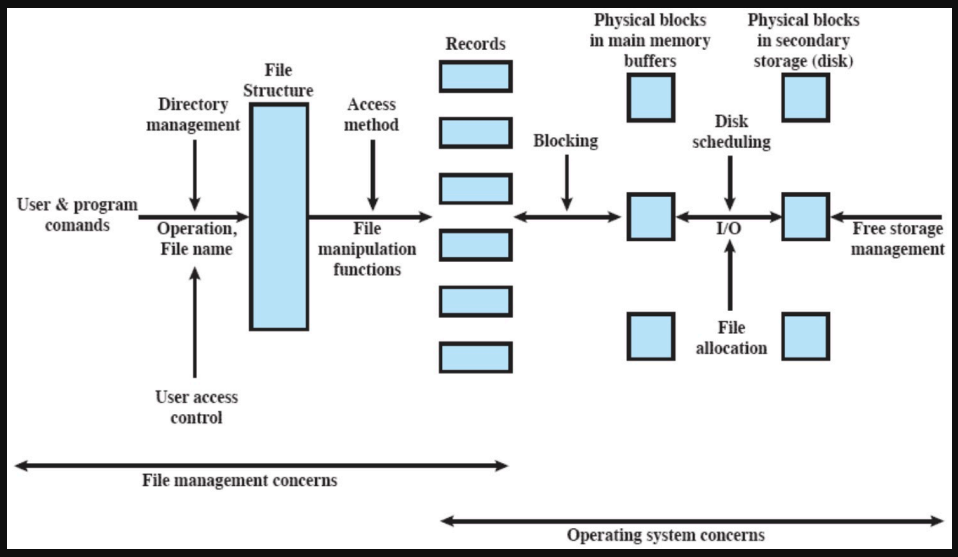


- ↳ **device drivers**
  - lowest lvl
  - communicates directly w/ peripheral devices
  - responsible for starting I/O ops on device
  - processes completion of I/O request
- ↳ **basic file system** (aka physical I/O)
  - doesn't understand + only handles it
  - deals w/ exchanging blocks of data
  - placement of blocks
  - buffering blocks in main mem
- ↳ **basic I/O supervisor**
  - maintains control struct for device I/O, scheduling, + file status
  - optimizes access to device I/O
  - manages I/O buffers

↳ logical I/O
- enables users + apps to access records
- provides general-purpose record I/O capability
- maintains basic data abt file

↳ access method
- lvl of file system closest to user
- diff methods reflect diff file structs + have diff ways to access/process data
- optimize access depending on app needs
  - → sequential
  - → direct (i.e. direct addr)
  - → indexed (i.e. id)

elmts of file management:



↳ fcns include:
- identify + locate selected file
- use directory to describe loc of all files + their attrs
- on shared system, describe user access control

FILE ORGANIZATION AND ACCESS

criteria for choosing file organization:

↳ short access time
- needed when accessing single record

↳ ease of update
- not concern for file on CD-ROM b/c it won't be updated

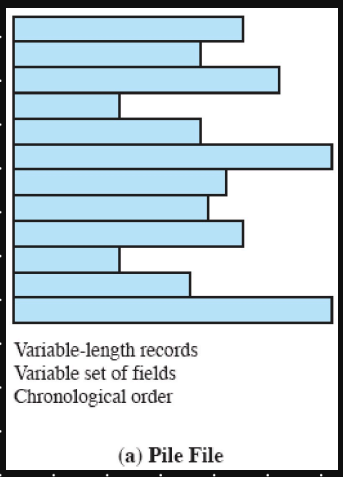
↳ economy of storage
- min. redundancy in data
- redundancy can speed up access (e.g. access)

↳ simple maintenance

↳ reliability

pile

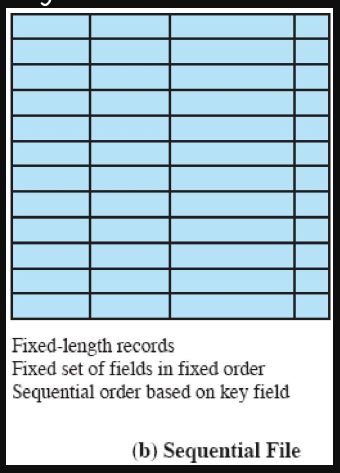
↳ e.g.



↳ data collected in order they arrive
↳ purpose is to accumulate mass of data & save it
↳ records may have diff fields
↳ no struct
↳ record access by exhaustive search
↳ useful when data is only collected & stored prior to processing or not easy to organize

sequential file

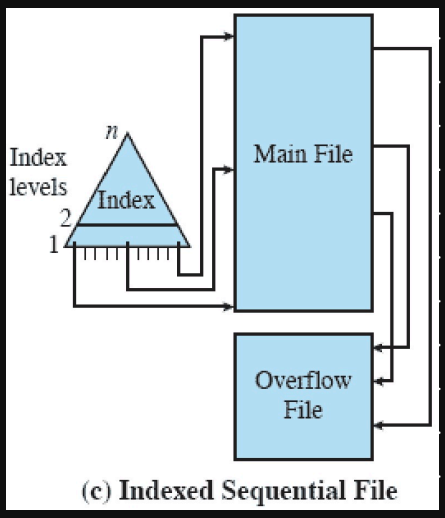
↳ e.g.



↳ fixed format used for records
↳ records are same len
↳ fields are same order & len
↳ field names & lens are attrs of file
↳ **key field** uniquely ids record
  - records stored in key seq
↳ search is inefficient
↳ updates are difficult b/c physical & logical organization must match

indexed sequential file

↳ e.g.



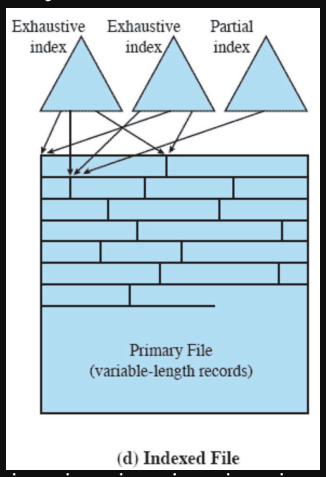
(c) Indexed Sequential File

↳ index provides lookup capability to quickly reach vicinity of desired record
↳ use index file for search:
- each record contains key field + ptr to main file
- search indexes to find highest key val that's equal to / precedes desired key val
- search cont in main file at loc given by index ptr

↳ new records are added to overflow file
- preceding record in main file updated to contain ptr to new record
- overflow file merged w/ main file during batch update
- when searching, go back to main file after coming across NULL ptr

↳ multiple index lvls for same key field can inc efficiency

indexed file
↳ e.g.



(d) Indexed File

↳ uses multiple indexes for diff key fields
↳ exhaustive index: contains 1 entry for every record in main file
↳ partial index: contains entries to records where field of interest exists
- w/ var-len records, some won't contain all fields

direct / hashed file
↳ directly access block at known addr
↳ key field required for each record

- performance for 5 basic file organizations:

File Method	Space: Attributes: Variable	Space: Attributes: Fixed	Update: Record Size: Equal	Update: Record Size: Greater	Retrieval: Single record	Retrieval: Subset	Retrieval: Exhaustive
Pile	A	B	A	E	E	D	B
Sequential	F	A	D	F	F	D	A
Indexed sequential	F	B	B	D	B	D	B
Indexed	B	C	C	C	A	B	D
Hashed	F	B	B	F	B	F	E

A	=	Excellent, well suited to this purpose	$\approx O(r)$
B	=	Good	$\approx O(o \times r)$
C	=	Adequate	$\approx O(r \log n)$
D	=	Requires some extra effort	$\approx O(n)$
E	=	Possible with extreme effort	$\approx O(r \times n)$
F	=	Not reasonable for this purpose	$\approx O(n^{>1})$

where
- r = size of the result
- o = number of records that overflow
- n = number of records in file

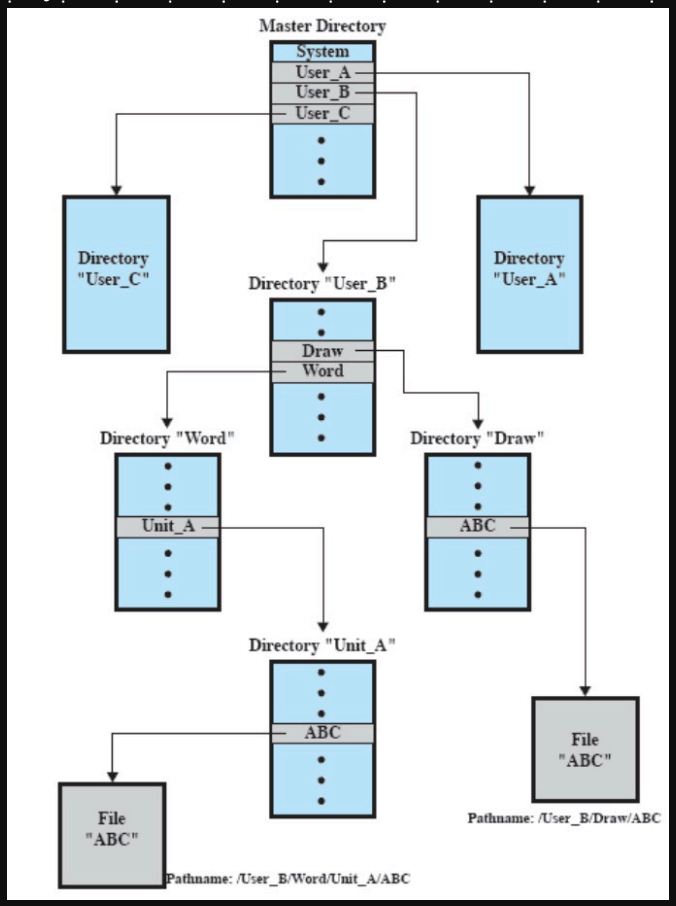
FILE DIRECTORIES

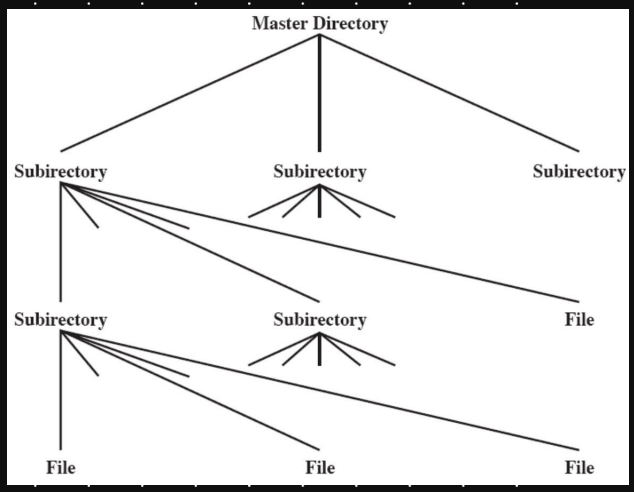
- file directories contain info abt files
  - ↳ attrs
  - ↳ loc
  - ↳ ownership
- directory is file itself usually accessed indirectly by user
  - ↳ provides mapping btwn file names & files themselves
- info. elmts of file dir:

Basic Information	
File Name	Name as chosen by creator (user or program). Must be unique within a specific directory
File Type	For example: text, binary, load module, etc.
File Organization	For systems that support different organizations
Address Information	
Volume	Indicates device on which file is stored
Starting Address	Starting physical address on secondary storage (e.g., cylinder, track, and block number on disk)
Size Used	Current size of the file in bytes, words, or blocks
Size Allocated	The maximum size of the file
Access Control Information	
Owner	User who is assigned control of this file. The owner may be able to grant/deny access to other users and to change these privileges.
Access Information	A simple version of this element would include the user's name and password for each authorized user.
Permitted Actions	Controls reading, writing, executing, and transmitting over a network
Usage Information	
Date Created	When file was first placed in directory
Identity of Creator	Usually but not necessarily the current owner
Date Last Read Access	Date of the last time a record was read
Identity of Last Reader	User who did the reading
Date Last Modified	Date of the last update, insertion, or deletion
Identity of Last Modifier	User who did the modifying
Date of Last Backup	Date of the last time the file was backed up on another storage medium
Current Usage	Information about current activity on the file, such as process or processes that have the file open, whether it is locked by a process, and whether the file has been updated in main memory but not yet on disk

types of ops on directory:
- ↳ search
- ↳ create file
- ↳ delete file
- ↳ list directory
- ↳ update directory

approaches to implement directory:
- ↳ single list
  - list of entries (1 for each file)
  - sequential file w/ file name as key
  - no help in organizing files
  - forces user to not use same name for diff files
- ↳ 2-lvl: 1 list per user
  - 1 directory for each user + master directory
    - → master contains entry for each user + provides addr + access control info
    - → user is simple list of files for that user
  - no help in structuring collection of files
- ↳ hierarchical / tree-structured
  - master directory w/ user directories under it
  - each user directory may have subdirectories + files as entries
  - e.g.





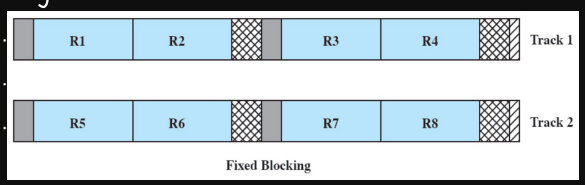
- files can be located by following **path from root / master** dir down branches
  - → i.e. **pathname**
- can have several files w/ same file name as long as they have unique pathnames
- **working dir**: curr dir
  - → files refed rltive to working dir

FILE SHARING

- in multiuser system, allow files to be shared among users
- **2 issues**:
  - ↳ access rights
  - ↳ management of simultaneous access
- access rights for user for particular file (each right implies those that precedes it):
  - ↳ **none**: user may not know existence of file + not allowed to read user dir that includes that file
  - ↳ **knowledge**: user can only know file exists + its owner
  - ↳ **execution**: user can load + execute program but not copy it
  - ↳ **reading**: user can read file, including copy + execute
  - ↳ **appending**: user can add data to file, but can't modify / delete any of file contents
  - ↳ **updating**: user can modify, delete, + add to file's data
    - includes creating file, rewriting it, + removing all / part of data
  - ↳ **changing protection**: user can change access rights granted to other users
  - ↳ **deletion**: user can delete file
  - ↳ **owner**: may grant rights to others based on diff classes of users:
    - specific user
    - user groups
    - all (i.e. public file)
- when dealing w/ **simultaneous access**, user may lock entire file or individual records during update
  - ↳ issues of mutual exclusion + deadlock

RECORD BLOCKING

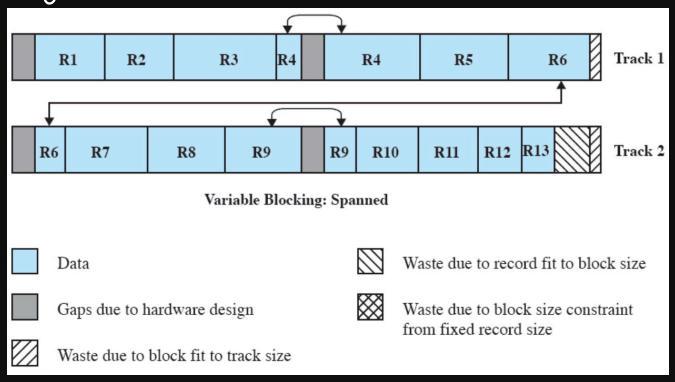
- records are logical unit of access of structured file
- blocks are unit of I/O w/ secondary storage
- for I/O to be performed, organize records in blocks on I/O device
- blocking methods:
  - ↳ **fixed blocking**: stores integral # fixed-len records in block
    - internal fragmentation
    - e.g.



Fixed Blocking

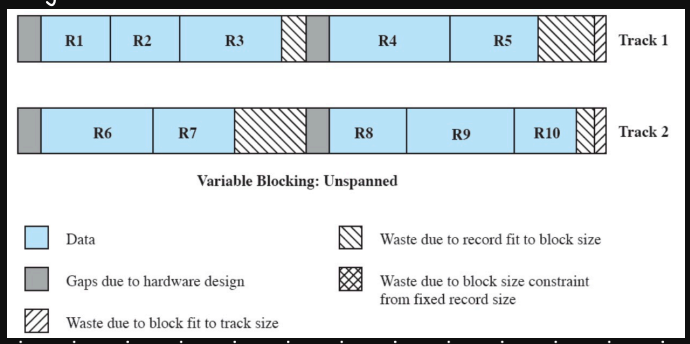
↳ **var-len spanned blocking**: var-len records + some span 2 blocks w/ continuation ptr.

- e.g.



↳ **var-len unspanned blocking**: var-len records w/o spanning

- e.g.



SECONDARY STORAGE MANAGEMENT

- OS / file management system allocs blocks to files
- 2 management issues in secondary storage:
  - ↳ space must be alloced to files
  - ↳ must keep track of space avail for allocation
- **portion**: contiguous set of alloced blocks to a file
  - ↳ 2 ways to determine portion size:
    - var, large contiguous portions
      - → provide better performance
      - → avoid waste
      - → small FATs
      - → hard to reuse space
    - blocks (i.e. small, fixed portions)
      - → greater flexibility
      - → may require complex structs + large FATs
      - → contiguity isn't primary goal

file allocation table (FAT): keeps track of portions assigned to file.

preallocation policy needs max size for file at time of creation.

↳ difficult to reliably estimate max potential size of file

↳ users tend to overestimate so they don't run out of space

contiguous allocation policy: single set of blocks alloced to file at time of creation

↳ single entry in FAT

- starting block + len of file

↳ external fragmentation will occur

- perform compaction

↳ e.g.

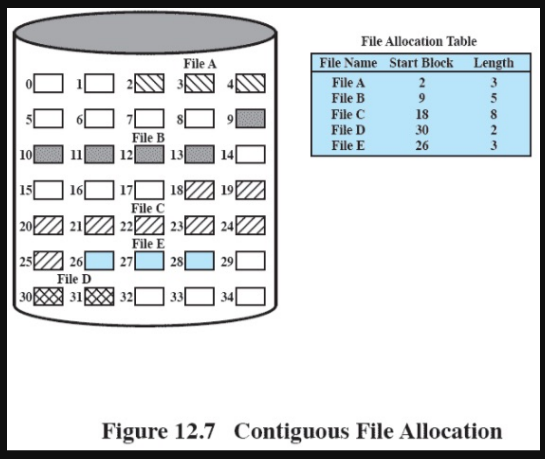


Figure 12.7 Contiguous File Allocation

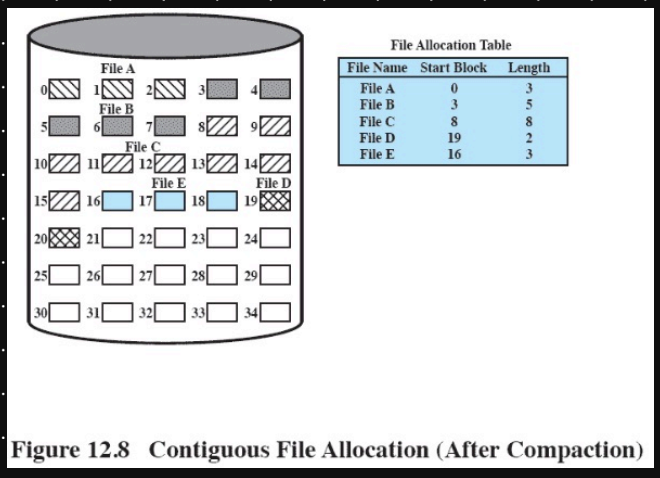


Figure 12.8 Contiguous File Allocation (After Compaction)

chained allocation policy: allocation on basis of individual block

↳ each block contains ptr to next block in chain

↳ single entry in FAT

- starting block + len of file

↳ no external fragmentation

↳ no accomadation of principle of locality

↳ best for sequential files

↳ if 1 block breaks, entire file is broken

↳ e.g.

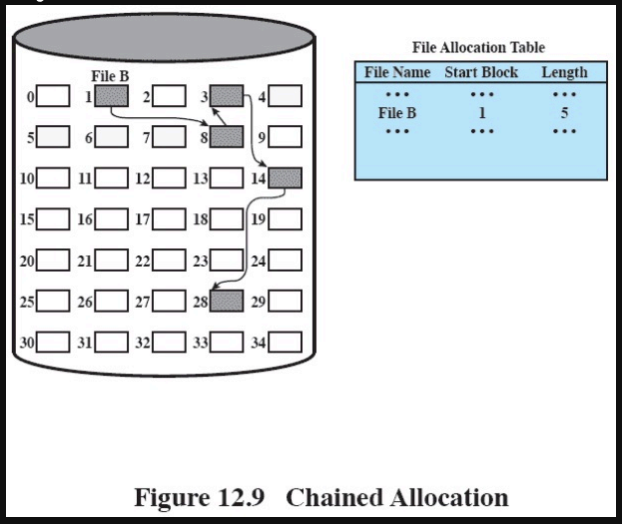


Figure 12.9 Chained Allocation

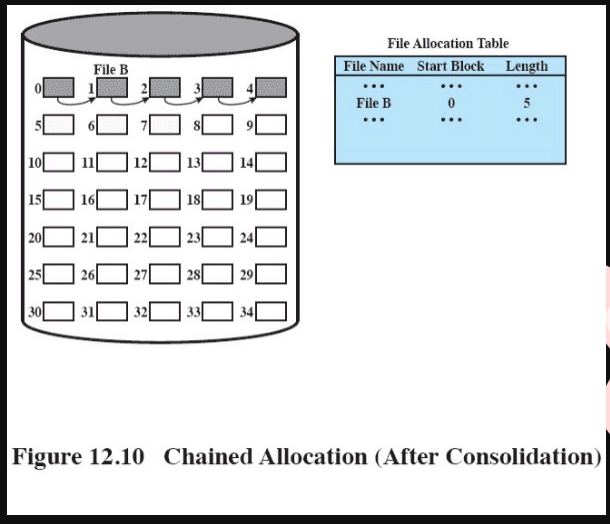


Figure 12.10 Chained Allocation (After Consolidation)

- **indexed allocation** policy: FAT contains separate 1-lvl index for each file (i.e. block # for index)
  - ↳ index has 1 entry for each portion alloced to file
  - ↳ if index block is corrupted, entire file is corrupted
  - ↳ e.g.

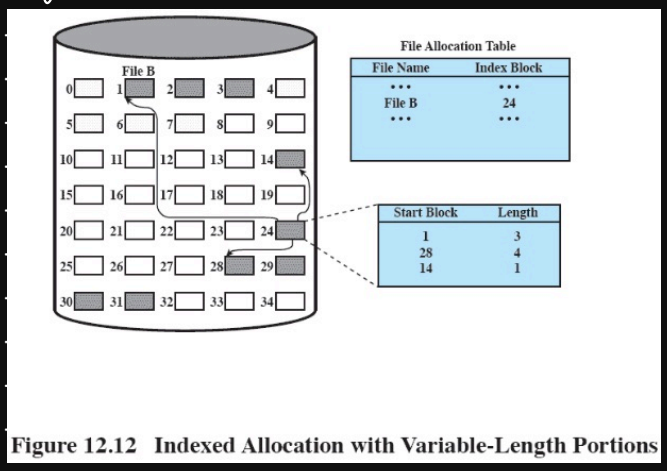


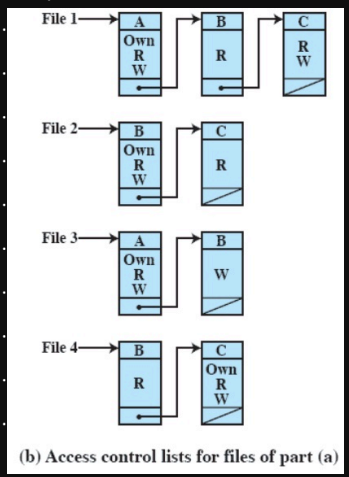
Figure 12.12 Indexed Allocation with Variable-Length Portions

- **access matrix**: general model of access control exercised by file management system
  - ↳ basic elmts:
    - subject: entity accessing obj
    - object: anything to which access is controlled
    - access right: way that obj is accessed by subject
  - ↳ e.g

	File 1	File 2	File 3	File 4	Account 1	Account 2
User A	Own R W		Own R W		Inquiry Credit	
User B	R	Own R W	W	R	Inquiry Debit	Inquiry Credit
User C	R W	R		Own R W		Inquiry Debit

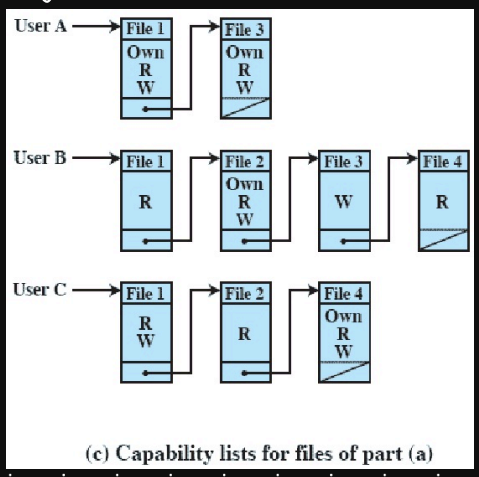
(a) Access matrix

  - ↳ usually sparse so can be decomposed in 2 ways:
    - by cols to create **access control list**
      - → e.g.



(b) Access control lists for files of part (a)

- by rows to create capability tickets / lists
  - → e.g.



(c) Capability lists for files of part (a)