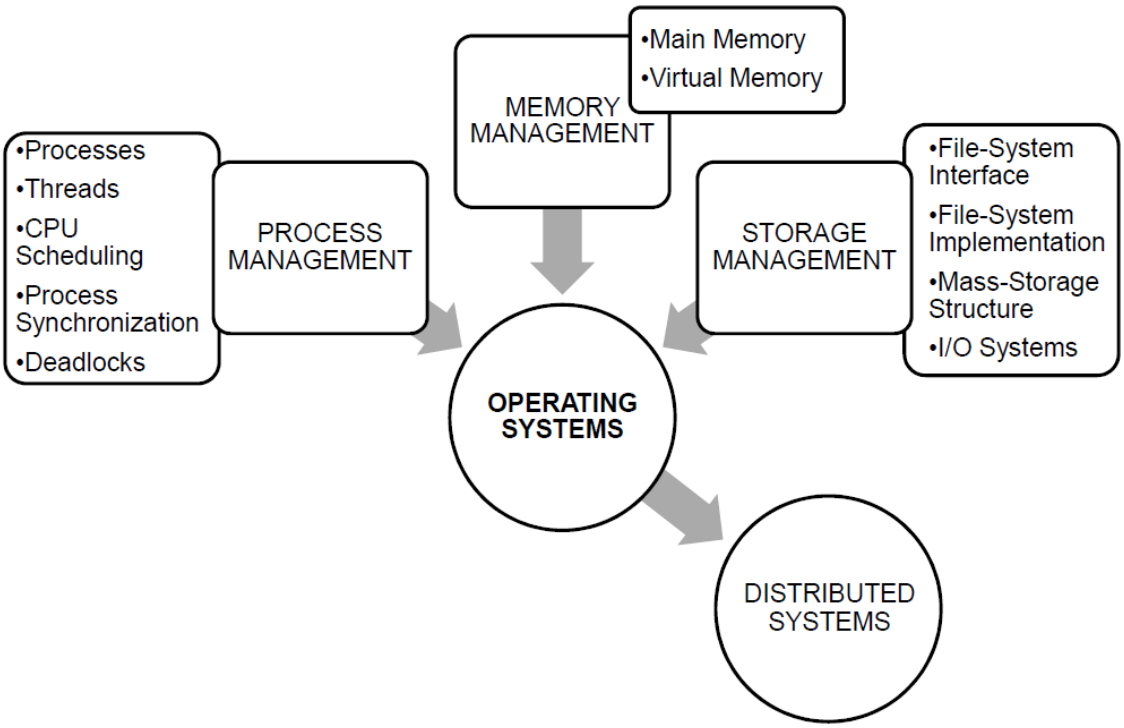
# Operating System

*All rights reserved by [CWHer](#)*

## Outline




## Chap0: Introduction

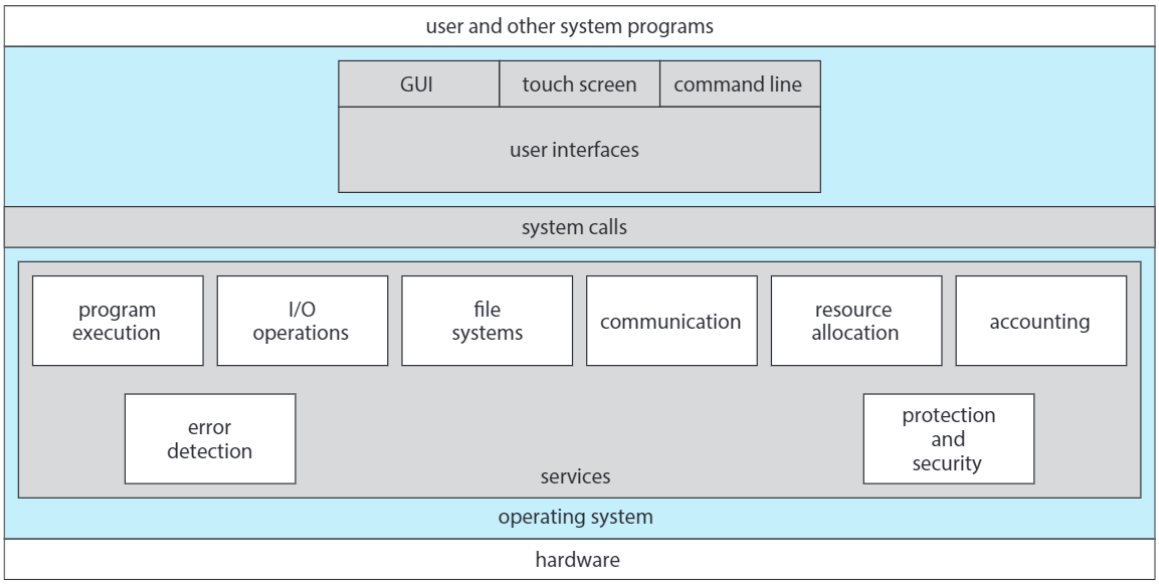
### OS

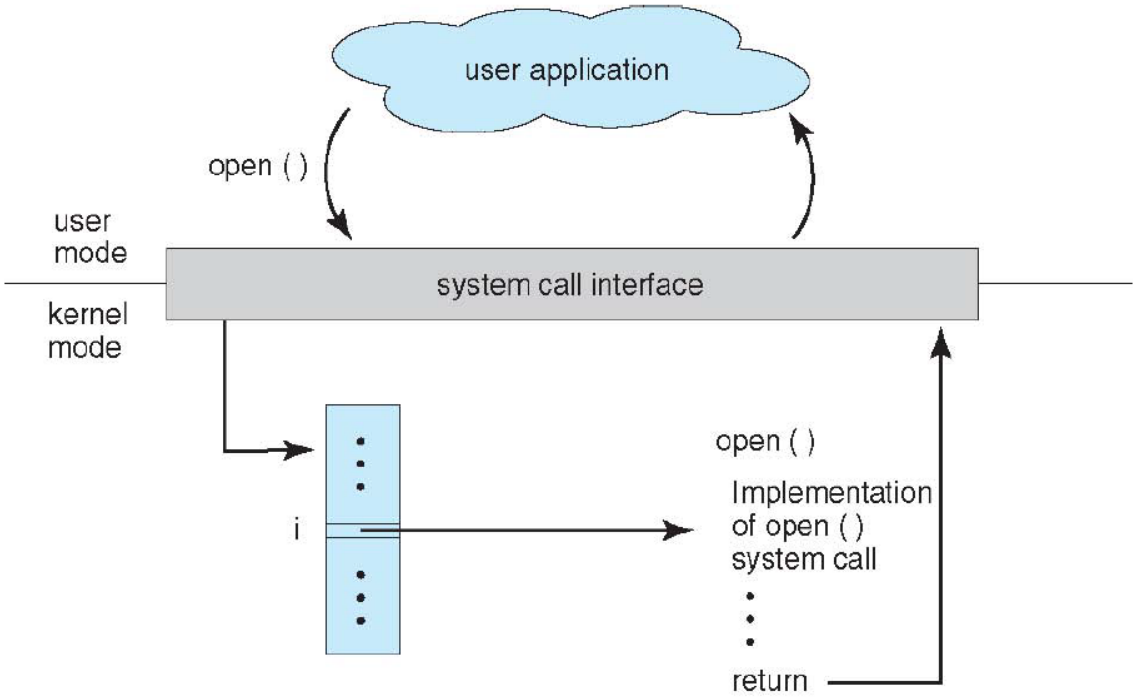
- intermediary between the computer user and the computer hardware
- resource allocator and control program

### kernel

The one program running at all times on the computer

## Chap1: OS Structures




**system call**: Programming interface to the services provided by the OS

**API**: a high-level Application Program Interface or maybe a combination of many system calls

Why use APIs rather than system calls?

- Program portability
- System calls are often more detailed and difficult to work with than the API

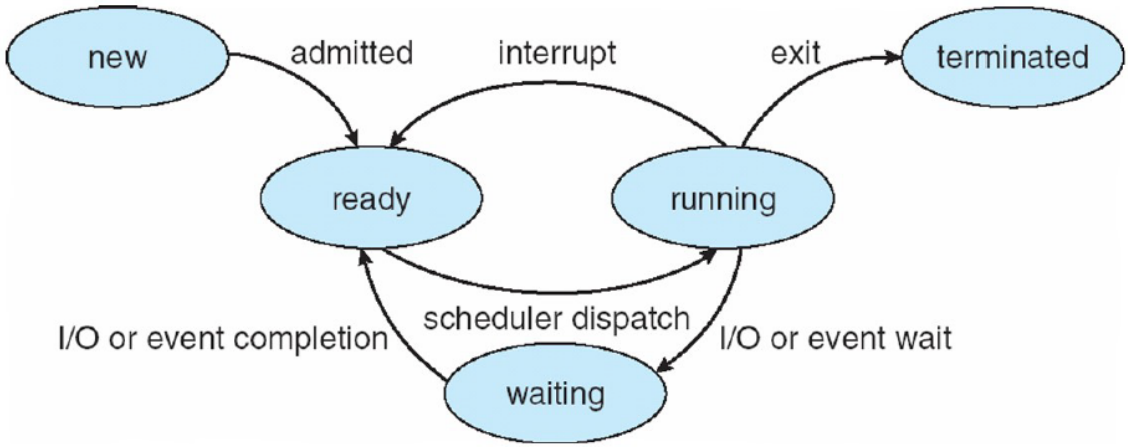
# Chap2: Processes

## process

program in execution (program+data+resource)

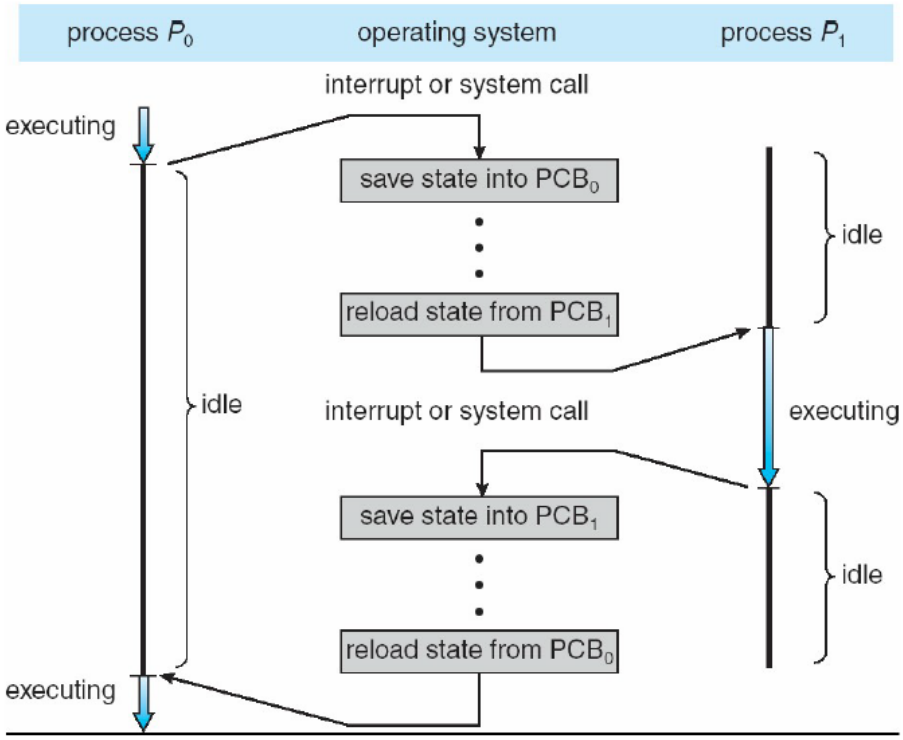
**PCB**: information associated with each process

## process state




## process switch

**context switch**: P1 preempt P0




## process scheduling

select among available processes for next execution on CPU

maintain **job** queue, **ready** queue and **device** queue

**long-term scheduler**: which to be brought into ready queue

**short-term scheduler**: which to be executed

**IO-bound process**: more time doing IO than CPU computation

**pid**: process identifier

### process creation

**challenges**: resource sharing, data initialization, execution order and address space

UNIX samples:

`fork()` : duplicate the environment of parent and use `wait()` to wait child exit

`exec()` : replace child's original environment and execute another program

**termination**: `exit (wait)` and `abort` (In some OS, parent's termination will cause cascading termination.)

### inter process communication (IPC)

**shared memory**:

producer-consumer problem (spin lock: busy waiting), pipe

**message passing**:

`send(P,message)` and `receive(Q,message)`

- direct communication: send directly to each other
- indirect communication: send to a mailbox and receive from it
    - sockets: an endpoint of communication (<IP , port>)

## Chap3: Threads

**concurrency**(并发): logical concept, a problem, single core with multiple threads

**parallelism**(并行): physical concept, a solution, multiple cores with multiple threads

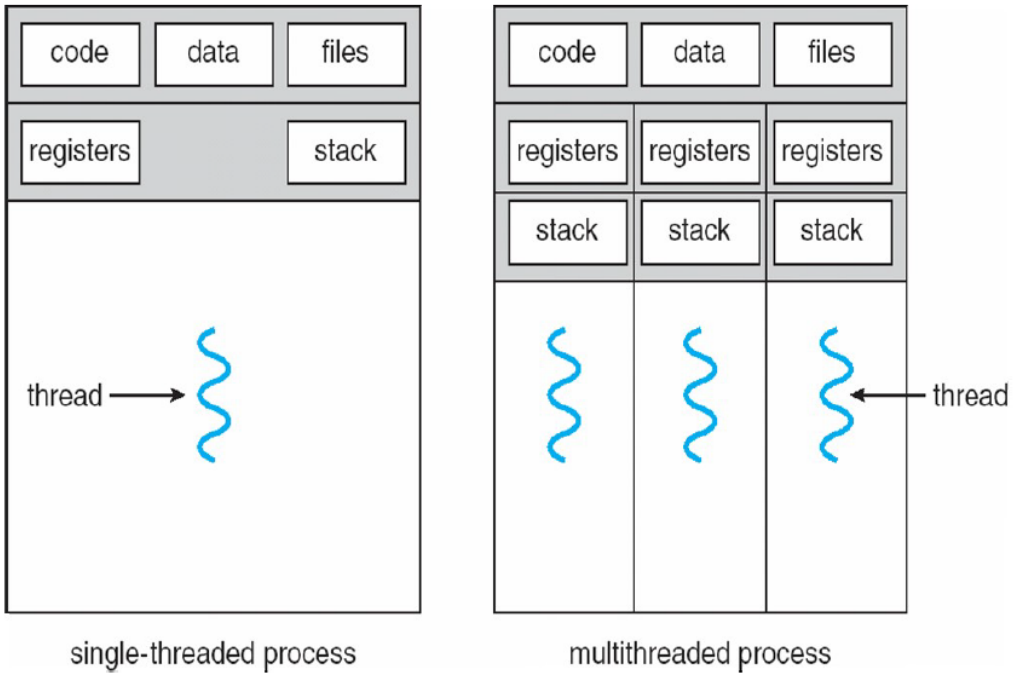
example: concurrency without parallelism

### thread

a basic CPU utilization unit

unique: thread ID, pc, registers, stack...

share: code section, data section, file handler.




**motivations**:

different jobs, speed up, multitasking within process.

**benefits**:

response even something stuck, resource sharing, fast that creating process, can be assigned on multiprocessor

**drawbacks**:

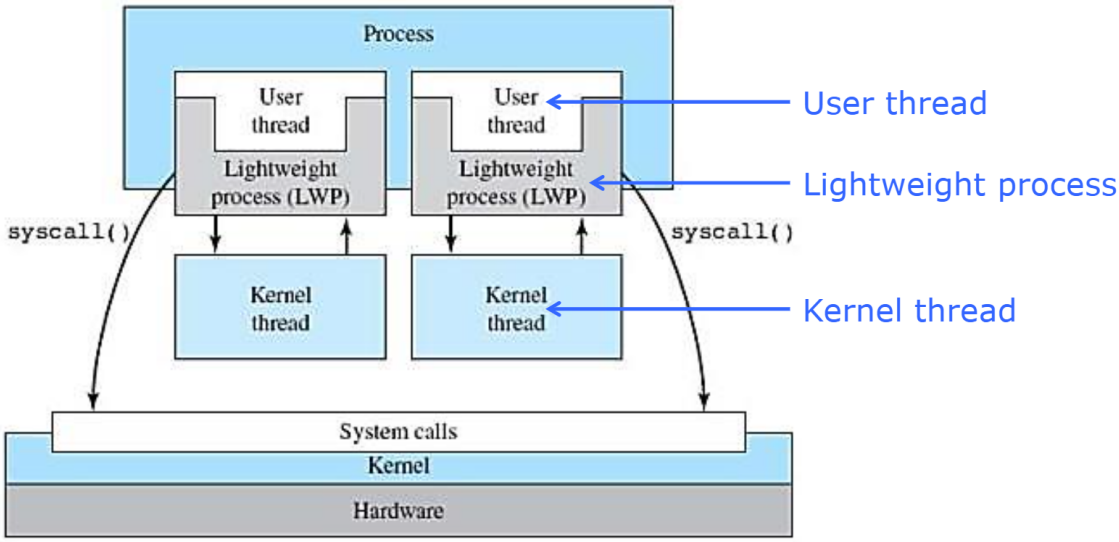
hard to program and debug, data race

## comparison with process

subset of process, share process state and memory, share address space, more ways to communicate, faster context switch

## supports for threads

- **kernel** threads: by OS kernel
- **user** threads: by user-level library (e.g. POSIX)




kernel thread (execution) - lightweight process - user thread (logical)

user-level library schedules user thread to lightweight process

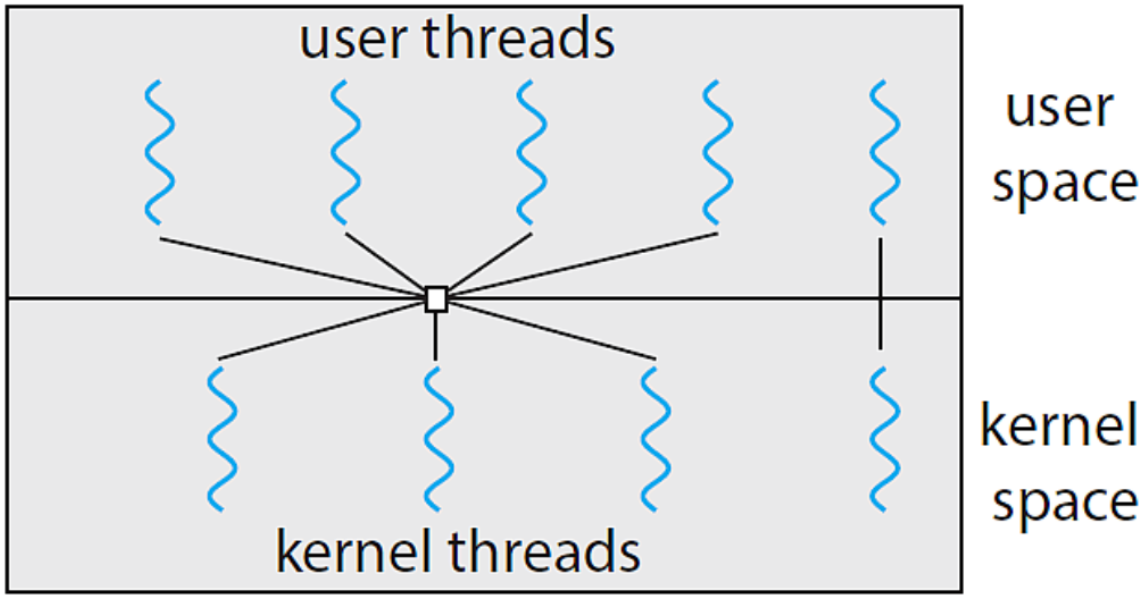
## multithreading model

many-to-one: no parallelism, might be blocked

one-to-one: more concurrency, more overhead (Linux)

many-to-many

two-level: combination of many-to-one and one-to-one




**thread libraries**:

user-level threads library: in user space, function call (Java threads)

kernel-level threads library: in kernel space, system call

## pthread

POSIX standard API, either user-level or kernel-level

```
pthread_t tid;
int param = 1;
pthread_create(&tid, NULL, runner, &param);
pthread_join(tid, NULL);
```

## thread issue

### some semantics

`fork()`: duplicate only caller or all threads

`exec()`: replace the entire process image

### thread cancellation

asynchronous cancellation: immediately cancel

deferred cancellation: target periodically check cancellation

### signal handling

deliver to where it applies, deliver to every thread, deliver to certain thread, assign a specific thread to receive

### thread pool

many threads await in a pool

slightly faster using existing threads, bound threads number

## examples

Linux:

`fork()` and `clone()` (can specify what resource to share)

# Chap4: CPU Scheduling

## CPU scheduler

select from ready queue

take place when

1. switch from running to waiting state (wait for IO)
2. switch from running to ready (time slice)
3. switch from waiting to ready (IO finish)
4. terminate

preemptive conditions: (2) and (3)

non-preemptive: CPU is allocated to a process until it terminates or turning to a waiting state

**dispatch latency**: time it takes to stop one and start another

## scheduling criteria

- CPU utilization
- throughput: # of processes complete per time unit
- turnaround time: interval from submission to completion
- waiting time: waiting in ready queue
- response time: interval from submission to first response

turnaround time = waiting time + request time

## scheduling algorithm

- FCFS

  convoy effect: shortest one behind long process
- SJF

  can cause starvation

  preemptive: interrupt current process and switch to the one with shorter burst time
- priority scheduling

  smaller number means higher priority

  use **aging** against starvation
- RR

  time quantum q too large => FIFO

  higher average turnaround time than SJF,  but better response

  80% CPU bursts should be shorter than q
- multilevel queue

  partition into separate queues

  each queue has own scheduling algorithm

  scheduling must be done between queues
- multilevel feedback queue
  - example

    Q1: RR with q=8

    Q2: RR with q=16

    Q3: FCFS

    can't finish within q: process will be degraded

    be preempted: process will be put into the end and still get same amount next time

  upgrade to avoid starvation

## thread scheduling

process-contention(进程竞争) scope: thread library schedules threads to run on kernel thread

system-contention(系统竞争) scope: kernel thread scheduled onto available CPU

# Chap5: Process synchronization

**hardware solutions**: atomic instruction or disable preempt (single core), spin lock (multicore)

**OS mechanisms**: semaphore, mutex, conditional variable

## background

concurrent access to shared data may result in data inconsistency

maintaining data consistency requires mechanisms

**race condition**: several processes access same data and outcome depends on execution order

**critical section**: at any time, critical section contains exactly one process

## requirements

- mutual exclusion

  no concurrency in critical section
- progress

  When no progress is in critical section, need to decide immediately which one next.
- bounded waiting
  - sequential access (stronger requirement): FCFS

## mechanisms

- synchronization hardware

  atomic instructions

  `TestAndSet()`: set target to true and return original value

  `swap()`: swap 2 variables

```
do {
    while (TestAndSet(&lock)) {}
    // while (key == true) swap(&lock,&key) {}
    lock = false;  // unbounded waiting
} while (true);
```

```
bool lock;  // lock==true if critical section is occupied
do {
    waiting[i] = true;  // I'm waiting
    key = true;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    waiting[i] = false;
    // critical section
    j = (i + 1) % n;    // bounded waiting
    increase j until waiting[j]==true
    if (j == i)
        lock = FALSE;       // waiting queue is empty
                            // no one is in critical section
    else
        waiting[j] = FALSE; // let j in
```

```
                                // also pass lock to j
} while (true);
```

- Peterson's solution

```
do {
    flag[i] = true; // I'm waiting
    turn = j;       // you first (token)
    while (flag[j] && turn == j);   // P[j] is in critical section
    // critical section
    flag[i] = false;
} while (TRUE);
```

- semaphores(信号量)

  with each semaphore there is an associated waiting queue

  may cause **deadlock** or **starvation**

  `wait()`

```
wait() {
    cnt--;
    if (cnt<0) {
        add to waiting queue
        sleep
    }
}
```

  `signal()`

```
signal() {
    cnt++;
    // initial value of cnt can be negative
    //  thus, we can sleep all jobs invoking wait()
    //  then let something else wakeup them
    if (cnt<=0&&!Q.empty()) {
        get a process from waiting queue
        wake up that process
    }
}
```

- monitors

## classical problems of synchronization

mainly use semaphores

- bounded-buffer problem

```
// producer
wait(position);
signal(item);
```

- dining-philosophers problem

naive solution: pick up left, pick up right and eat

improvement:

- get 2 chopsticks at the same time
- preemptive
- give up chopstick when stuck
- some grab left, some grab right

- readers and writers problem

  no starvation solution: let reader and writer wait in line

```
int read_cnt=0;
sem mutex=1,wrt=1,rd=1;
// writer
{
    wait(wrt);  // wait in line
    wait(rd);
    signal(rd);
    signal(wrt);
}
// reader
{
    wait(wrt);  // wait in line
    wait(mutex);
    if (read_cnt++==0) wait(rd);
    signal(mutex);
    signal(wrt);
    // reading
    wait(mutex);
    if (--read_cnt==0) signal(rd);
    signal(mutex);
}
```

- baboons crossing problem

```
{
    wait(wrt);          // wait in line
    wait(other_side);
    wait(not_full);     // whether full
    wait(cnt_mutex);
    if (cnt++==0) wait(this_side);
    if (cnt<K) signal(not_full);    // not full
    signal(cnt_mutex);
    signal(other_side); // this side can still get in
    signal(wrt);
    // cross
    wait(cnt_mutex);
    if (cnt==K) signal(not_full);   // someone left, not full again
    if (--cnt==0) signal(this_side);
    signal(cnt_mutex);
}
```

- barrier

```
{
    wait(mutex);
    cnt++;
    if (cnt<K) {
        signal(mutex);
        wait(barrier);
    }
    else {  // last arrive
        cnt--;
        while (cnt>0) signal(barrier);
        signal(mutex);
    }
}
```

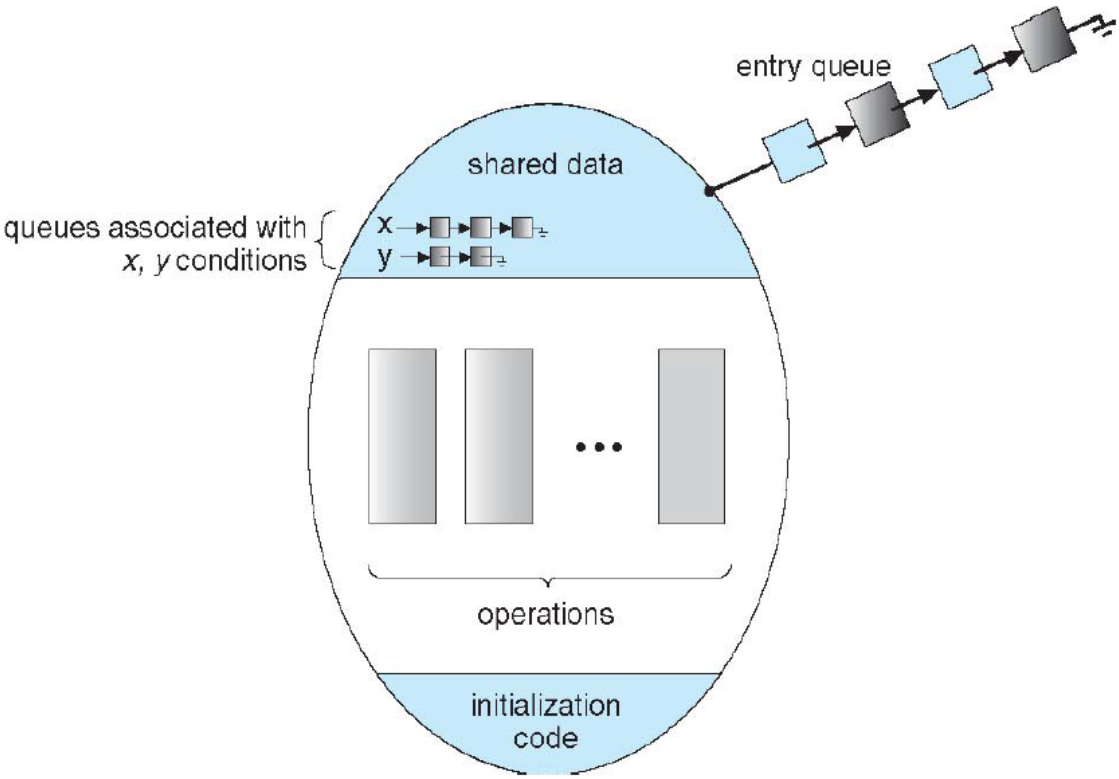
## monitor

**semaphore**: fine to first signal then wait

**monitor**: nothing happens when first signal then wait

abstract data type, internal variables only accessible by code within the procedure

only one process may be active within the monitor at a time




## condition variables

`x.wait()`: suspend current

`x.signal()`: resumes one of suspended processes

**options**: (when P signal Q)

- signal and wait: P wait

- signal and continue: Q wait

solution: P leaves the monitor immediately after signal Q

**dining philosophers**

Don't directly use mutex to represent chopstick. Instead, use mutex (or monitor) to protect chopstick status.

may cause starvation: neighbor is always eating

```
monitor
{
    void test(int i) {
        I'm hungry and neighbours are not eating
        {
            state[i] = EATING;  // I can eat
            self[i].signal();   // maybe waiting
        }
    }

    void pick(int i) {
        state[i] = HUNGRY;
        test(i);
        if (state[i]!=EATING) self[i].wait();   // can eat now
    }

    void putdown(int i) {
        state[i] = THINKING;
        test(neighbours);   // my neighbours maybe wait
    }
}
```

## example: using semaphore to implement monitor

monitor variables

```
sem mutex;  // init = 1
// let newly coming process wait
sem next;   // init = 0
// next only contains those invoking .signal
int next_cnt = 0;
```

procedure

```
wait(mutex);    // one process at a time
...
if (next_cnt>0)
    signal(next);   // let next get in, implicitly pass mutex
else
    signal(mutex);
```

conditional variable

```
sem x_sem;  // init = 0
int x_cnt = 0;
```

```
// wait
{
    x_cnt++;
    if (next_cnt>0) // let another procedure
        signal(next);
    else
        signal(mutex);
    wait(x_sem);    // waiting
    x_cnt--;
}

// signal
{
    if (x_cnt>0)    // someone is waiting
    {
        next_cnt++; // after wake up Q, P should wait
        signal(x_sem);
        wait(next); // put into next
        next_cnt--;
    }
}
```

# Chap6: Deadlocks

**starvation**: some one can't progress

**deadlock**: none can progress

## deadlock characteristics

- mutual exclusion
- hold and wait
- no preemption
- circular wait

## system model

- process $P_1 \dots P_n$
- resource types $R_1 \dots R_m$
- each resource $R_i$ has $W_i$ instances
- each process utilize a resource

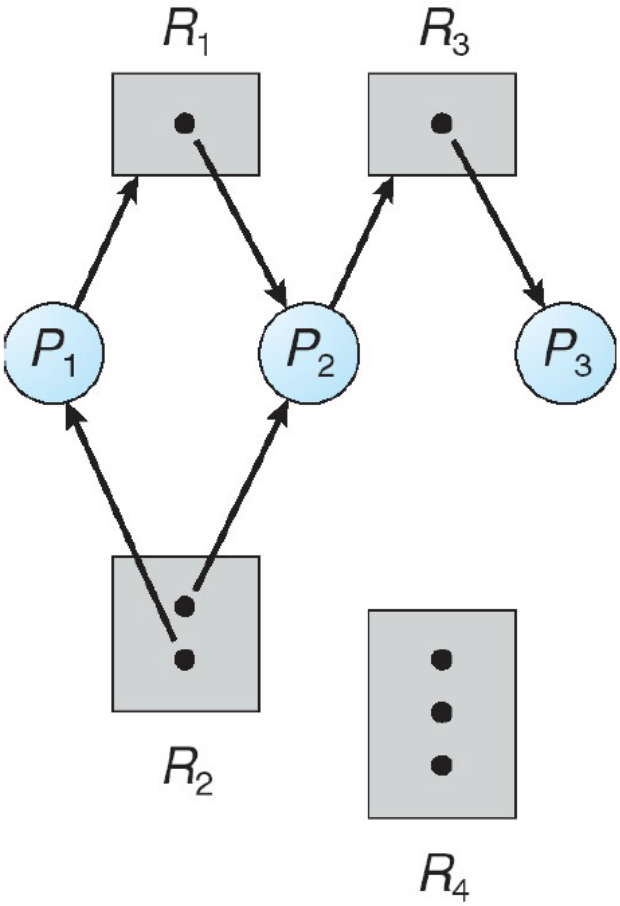
  request, use and release

## resource-allocation graph

request edge: $P_i$ to $R_i$

assignment edge: one of $R_i$ to $P_i$

**deadlock condition**: do topology sort, assign idle resources, repeat. (not correct)




## handle deadlock

ensure system never get into deadlock

- prevention(预防)
- avoidance(避免)

allow system to enter deadlock and recover

- detection
- recovery

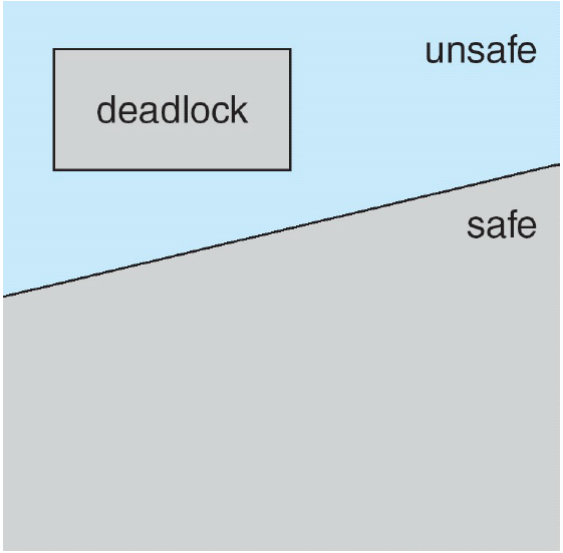
## deadlock prevention

break one of the deadlock characteristics

- not required for sharable resources
- hold and wait
  - require process to request and be allocated all its resources before it begins execution
  - allow process to request resources only when the process has none
- when requesting another resource that cannot be immediately allocated to it, then all resources currently being held are preempted
- impose a total ordering of all resource types and requests resources in an increasing order

## deadlock avoidance

need priori information




**safe state**: a safe sequence <P1, P2 ... Pn> of execution

```
Process A                   Process B
lock X                      lock Y           # state is "unsafe"
                            unlock Y
lock Y                                       # state is back to "safe" (no
deadlock this time.  We got lucky.)
```

**avoidance algorithm**

the request is granted only if the allocation leaves the system in a safe state.

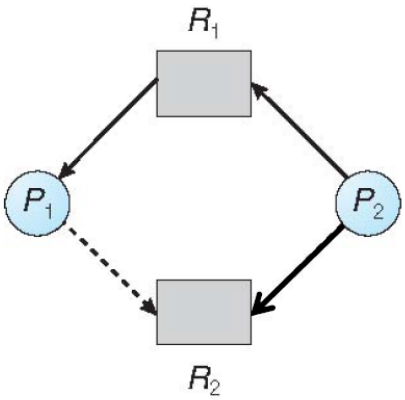
- resource-allocation graph (single instance)

  resources must be claimed a priori in the system

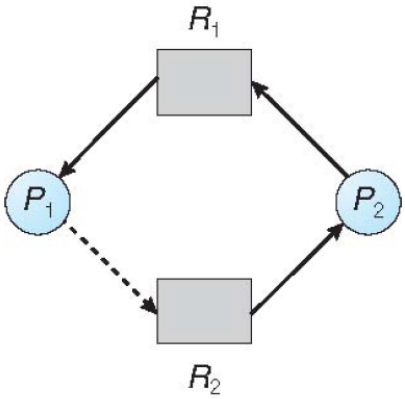
  claim edge (dashed): Pi will request Rj

  claim edge converts to request edge when a process requests a resource

  request edge converts to an assignment edge when the resource is allocated to the process




Can we grant $P_2$'s request for $R_2$?




Circle! Therefore, $P_2$'s request cannot be granted, and $P_2$ needs to wait.

- banker's algorithm

  Available[m]: Rj available

  Max[n, m]: Max[i, j] = k, Pi will request at most k instances of Rj

  Allocation[n, m]: Allocation[i, j] = k, Pi is allocated k instances of Rj

  Need[n, m]: Need[i, j] = k, Pi may need k more instances of Rj, = Max[i, j] - Allocation[i, j]

  - safety check

allocate resources to any Finish[i] = False, Need <= Work

Finish[i] = True, release all its resources

repeat this progress

- request check

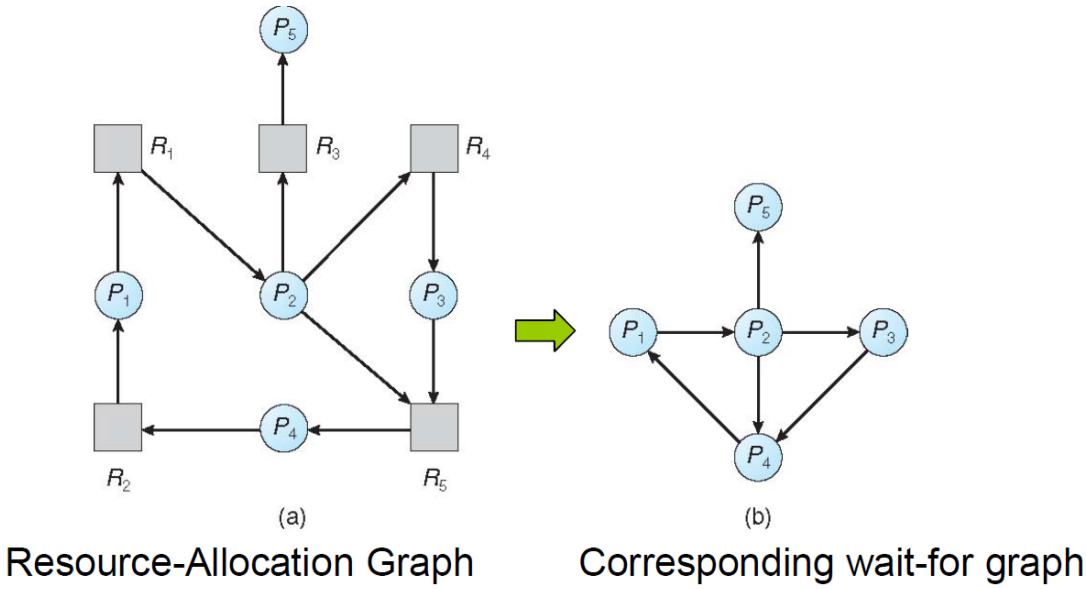
  for each request, if resource is sufficient, pretend to allocate and detect whether it leaves a safe state

## deadlock detection

- single instance

  maintain wait-for graph

  periodically check cycle




- several instances

  Available[m]

  Allocation[n, m]

  Request[n, m]

  find whether there's a safe sequence, just like banker's algorithm

### usage

- how often deadlock occurs?
  - periodically
  - when some requests can't be satisfied
- how many process will be involved?

## recovery from deadlock

- process termination
  - abort all
  - abort one at a time

    order: priority, time consumed, resources...
- resource preemption

  select a victim and roll back

# Chap7: Main Memory

8086: base + limit

**logical** address (virtual address): user program deals with

**physical** address: seen by memory unit

**MMU**: map logical address to physical address

## contiguous allocation

each process is contained in a single contiguous section of memory (base+limit)

strategies: first-fit (N allocated, 0.5N lost), best-fit and worst-fit

## fragmentation

- external fragmentation

  not contiguous holes
- internal fragmentation

  allocated memory slightly larger than request

solutions: compaction or paging + segmentation

## paging

divide physical memory into fix-sized blocks called **frames**

divide logical memory into blocks of same size called **pages**

set up a page table

**PTE** must contain PPN(physical page number) and optionally contains R/W/X, valid bit and dirty bit

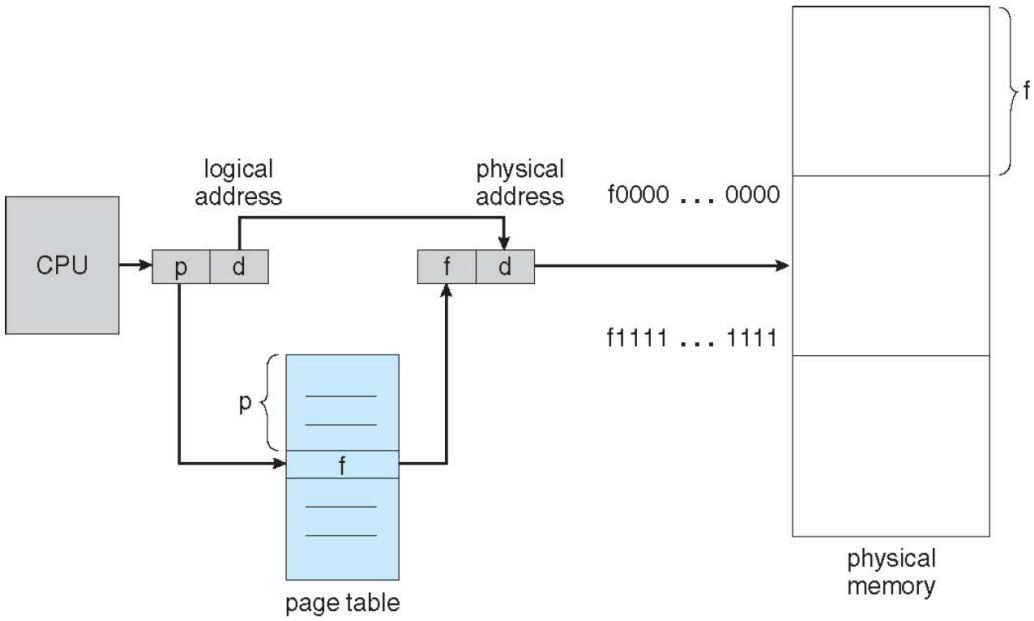
- address translation

  virtual address: $aaabbbccc|ddd$

  $aaabbbccc$ is page number, $ddd$ is offset

  if $aaabbbccc$ maps to $xxx$, then physical address is $xxxddd$

  $xxx$ doesn't contain zeros at end. (paging results in 0 at end)




internal fragmentation: frame size - 1 (worst)

- implementations

  page table stores in main memory

  need 2 memory accesses: one for page table, one for data

  may need more, when page table is multi-level

  - page-table base register (PTBR)
  - page-table length register (PTLR)

  **TLB** (translation lookaside buffer) need flush after context switch

  EAT(effective access time)=$(\varepsilon+1)\alpha+(\varepsilon+2)(1-\alpha)$

- structure of page table
  - hierarchical page tables

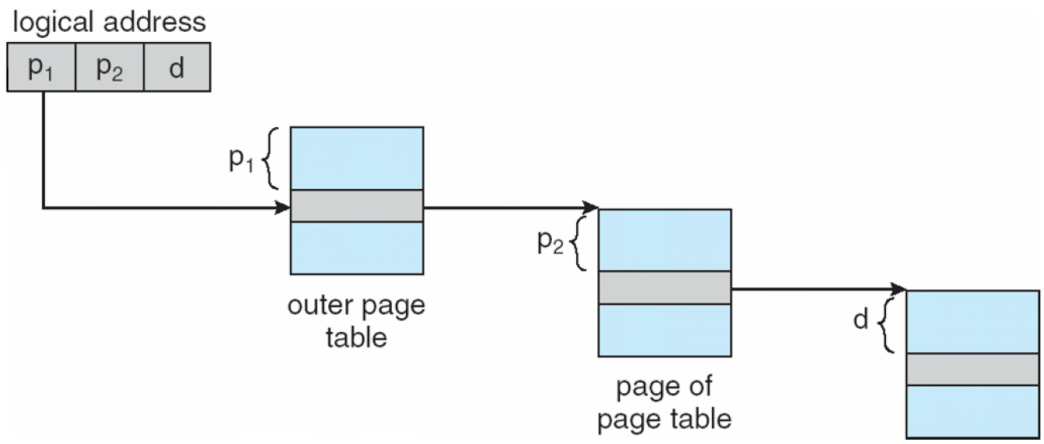
    example:

    32-bit machine => up to 4 bytes entry (no need to store the zeros at end)

    4K page size => 12 bits offset

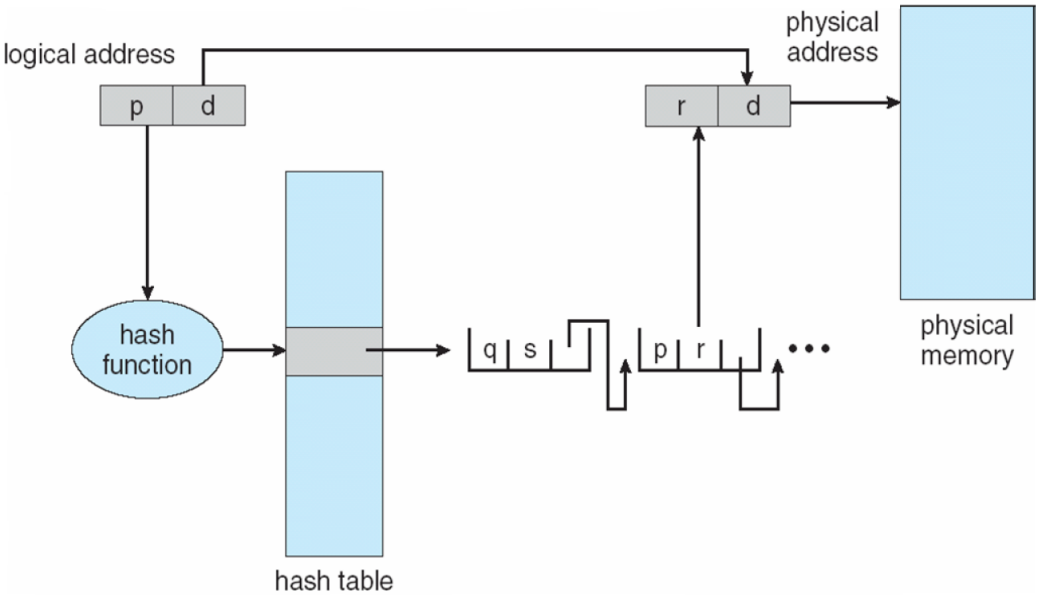
    one page table contains up to 4K/4=1K entries

    logical address: | p1 10bits | p2 10bits | offset 12bits |




  - hashed page tables

the virtual page number is hashed into a page table and the page table contains a chain

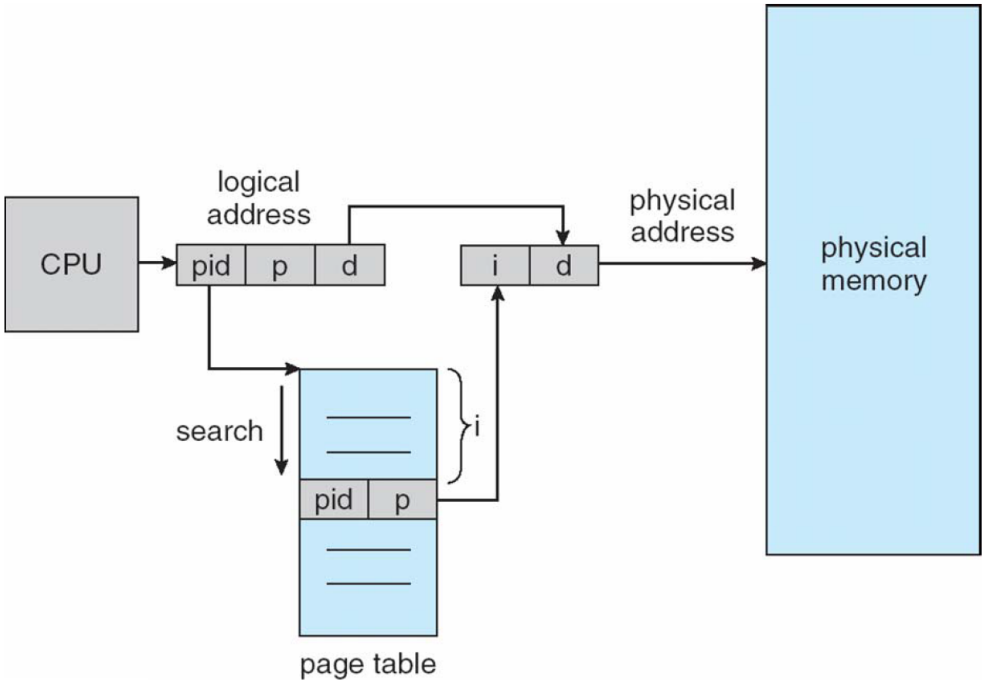



- inverted(倒置) page tables

  track all physical pages instead of logical pages

  entry consists of pid and virtual address

  decrease entry size but increase search time




## segmentation(分段)

a program is a collection of segments (e.g. stack, text, data)

logical address: <segment-number, offset>

segment table: <base, limit>

# Chap8: Virtual Memory

entire program doesn't need to be in memory at the same time

## virtual memory

separation of user logical memory from physical memory

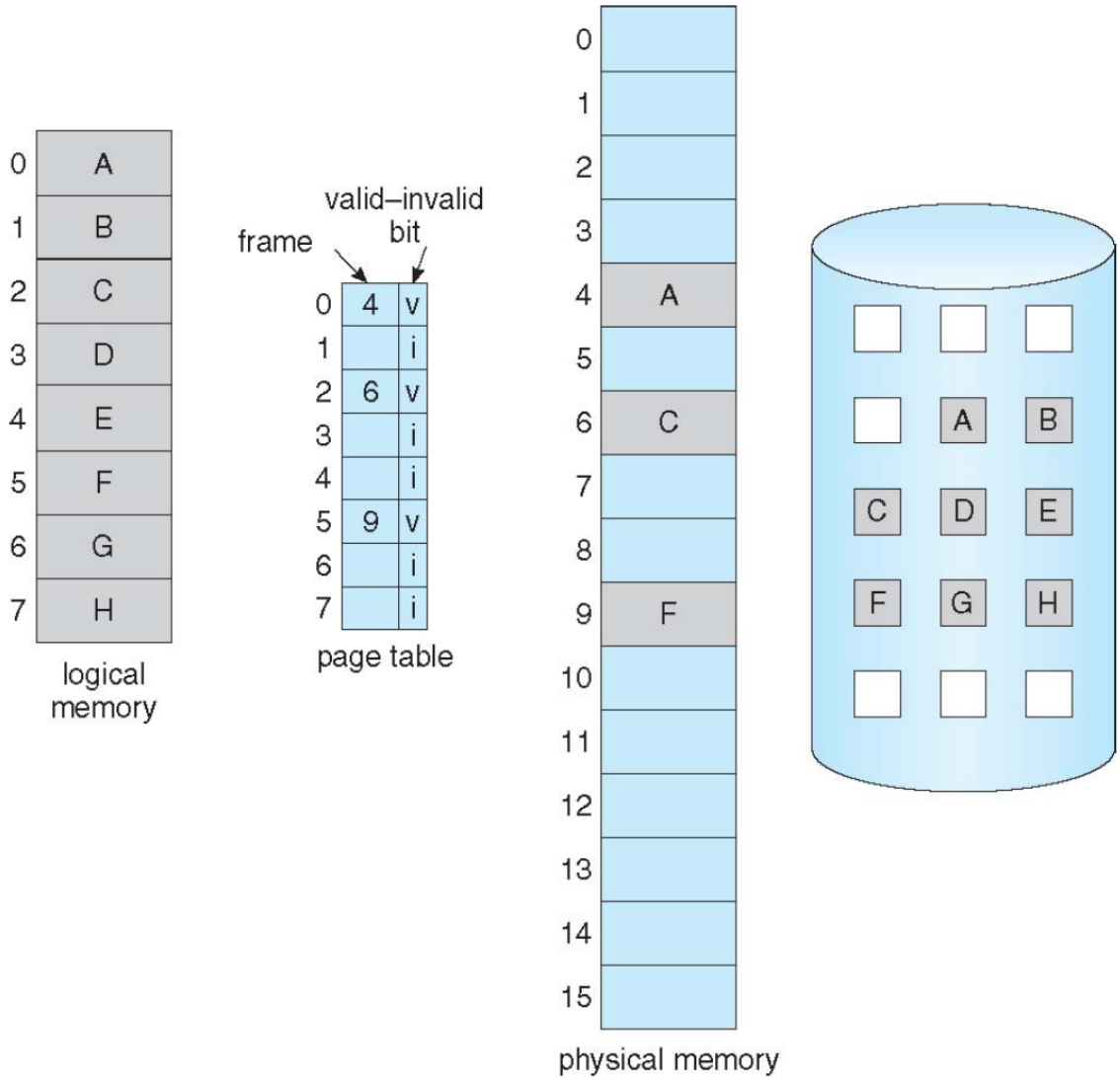
- only part of the program needs to be in memory
- larger logical address space
- allow shared memory
- more concurrent in memory

## demand paging

don’t bring page into memory until demand

page needed => refer to it

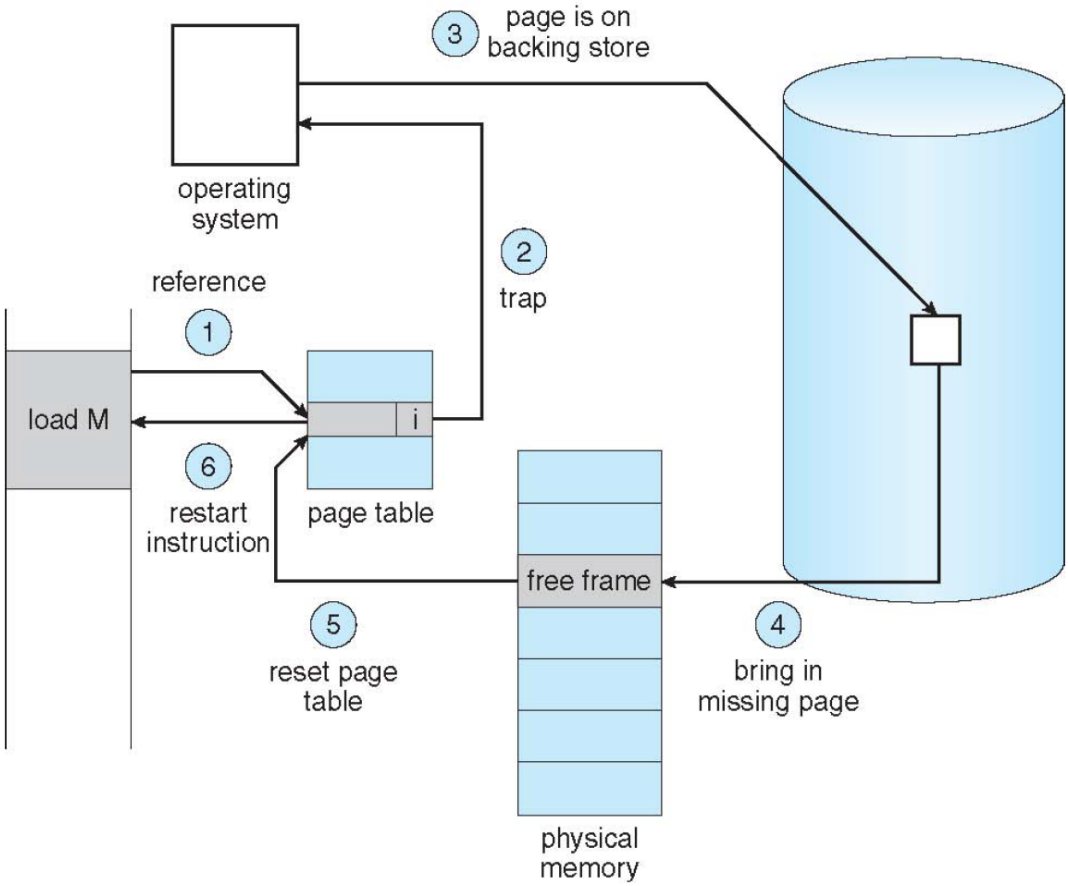
- invalid reference => abort
- not-in-memory => page fault



## page fault

page valid but not in memory

- get empty frame
- swap in page by disk and reset valid bit
- restart from last instruction




## page replacement

dirty bit: only modified pages are written to disk

**algorithms**

- FIFO
- optimal page replacement

  replace page that won't be used for longest time
- LRU (least recently used)
- LRU approximation
  - second-chance algorithm

    **reference** bit

    set to 1 when referenced

    which to replace
  - reference bit = 1 => set to 0, check next page
    - reference bit = 0 => replace it

    when all reference bits are 1, it becomes FIFO
- counting algorithms

keep a counter

least frequently used or most frequently used

### example of both chap7 and chap8

refer to slides

virtual address to MMU

- not in memory => page fault
- in memory => translate in to physical address

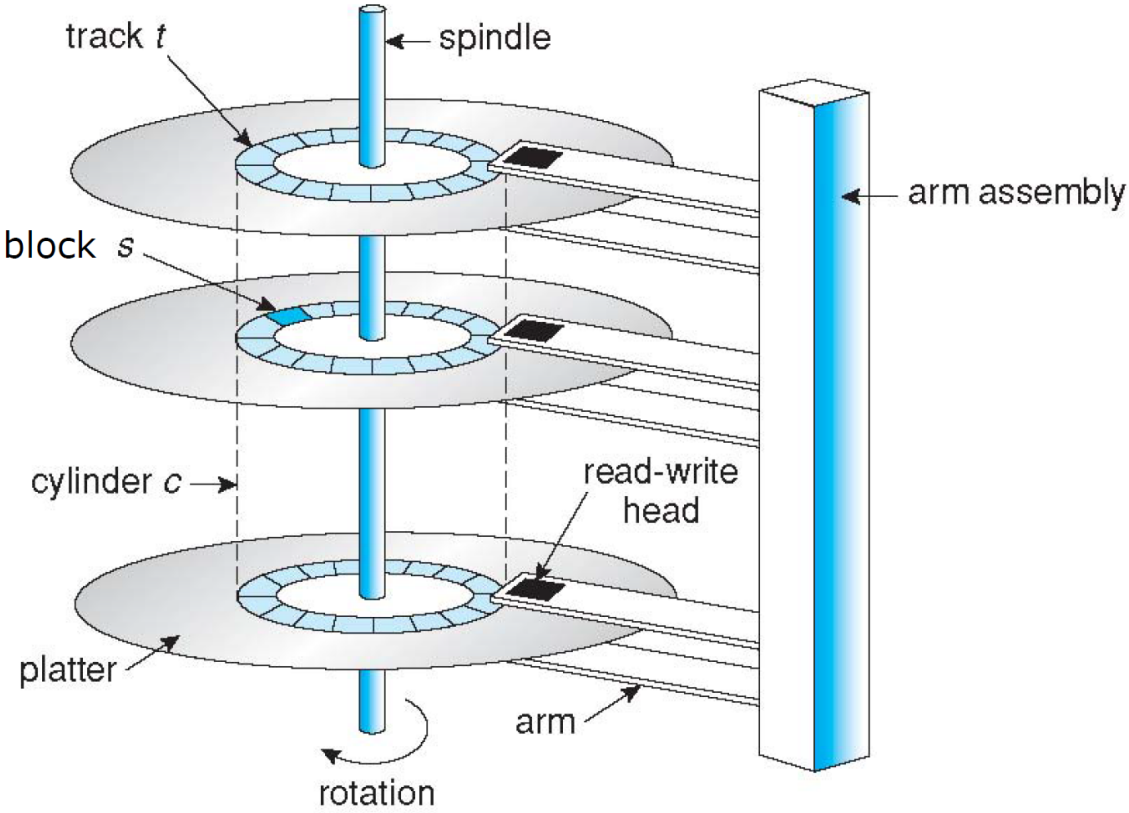
## Chap9: Mass-Storage System

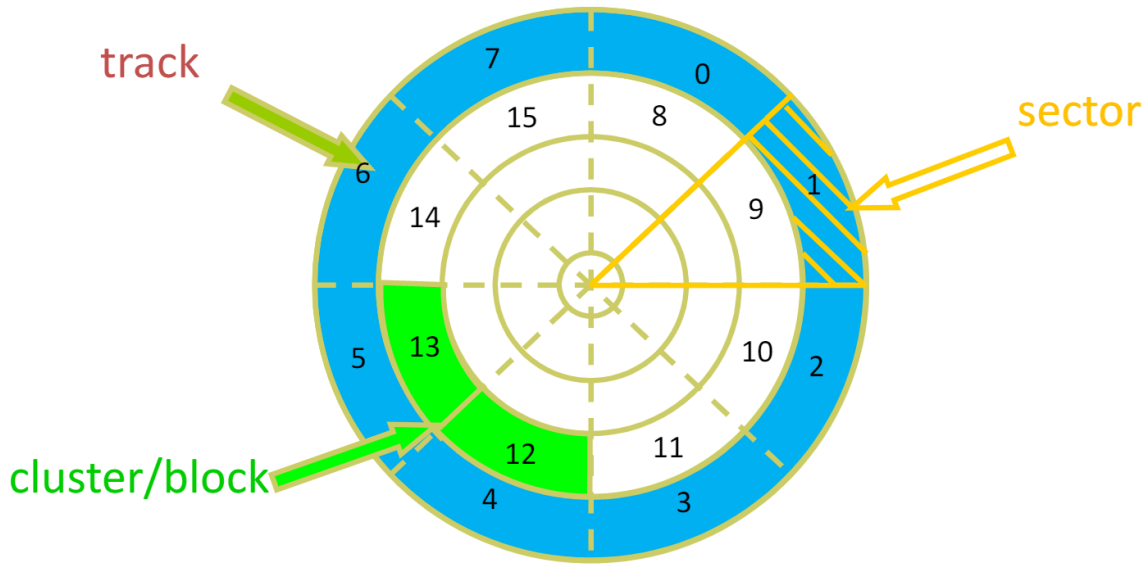
### structure

cluster(contain many blocks) => block(OS concept, logical) = sector(扇区) <= track(磁道) <= cylinder(柱面) <= surface <= platter(盘片)

each track contains same amount of sectors, because it takes certain time to rotate certain angle

positioning time = seek time (seek cylinder) + rotational latency (seek start sector)






## disk scheduling

only FCFS can permit no starvation

- FCFS
- SSTF

  may cause starvation
- SCAN

  from one end to another end and then reserve

  may cause more latency
- C-SCAN

  when reaching the other end, directly move back to start point

  can reduce latency for those requests at start point but not served during first scan

  besides, there are fewer requests at end point
- C-LOOK

  only go as far as the last request in each direction, then reverses direction immediately

## SSD

read and write in page (整块写入), can only erase page within a limited times

- modify

  copy and modify this block to an empty one

  set this block to be invalid
- garbage collection

# Chap10: File System

## access lists and groups

read, write, execute: 4+2+1

761: owner(RWX), group(RW), public(X)

## file system organization

partition: | directory | file |

partition can contain many disks and disk can also be divided into many partitions

volume: partition with file system

## directory

- single-level

  an array of directory
- two level

  first level is users
- tree structure

## file allocation

- contiguous allocation

  start + length, best performance, fragmentation
- linked allocation

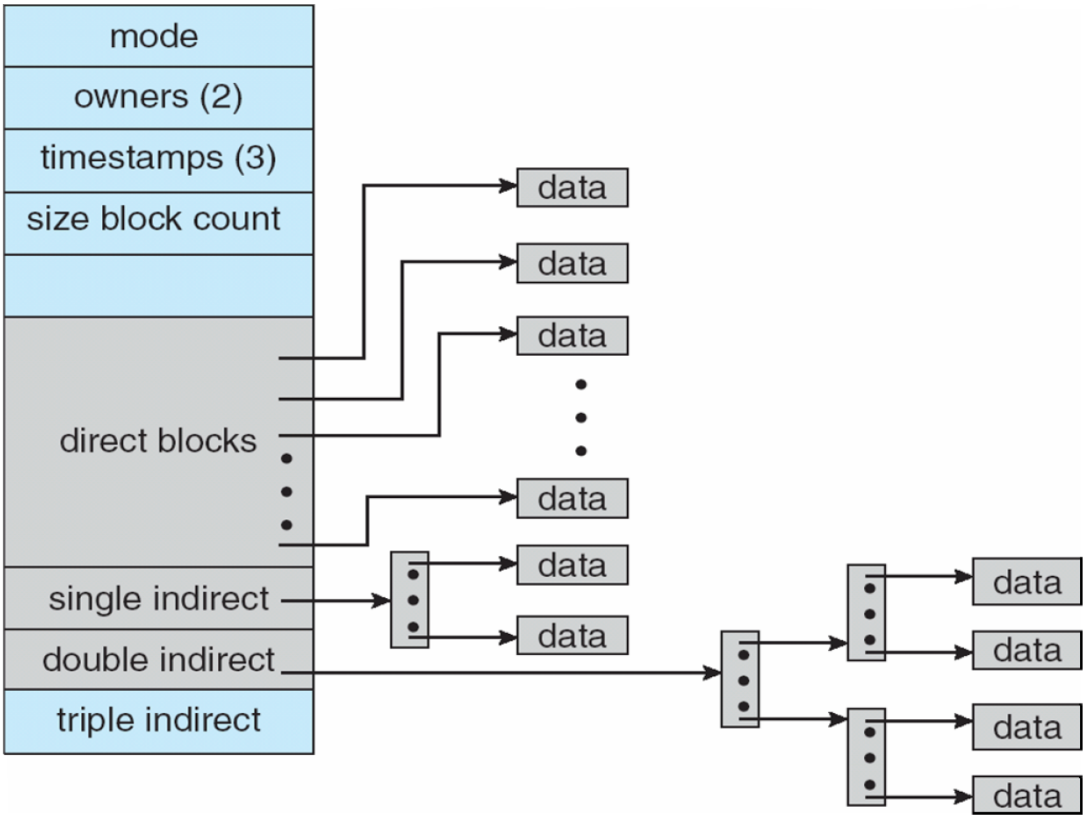
  each file is a linked list of blocks, each block contains pointer

  reliability problem (lose following blocks), more IO and seek
  - FAT (file-allocation table)

    beginning of volume has a table, indexed by block number

    more cacheable and faster
- indexed allocation

  each file has its own index blocks, which contains all blocks of this file




# Chap11: Distributed System Structure

## distributed system

distributed system is a collection of loosely coupled processors interconnected by a communication network

**site** is the location of processor (processors a.k.a. nodes, computers, machines, hosts)

reasons

- resource sharing (Google)
- computation speed-up (high performance computing)
- reliability
- communication

**tightly** coupled: two CPU shares same memory

**loosely** coupled: each CPU has own memory (easy to scale)

## basic performance metrics

workload (W) = number of ops required to complete the program

Tp = execution time using P processors

speed = W / Tp

speedup (S) = T1 / Tp

efficiency = S / P

general agreement: speed up = serial time / parallel time

## types of distributed systems

- network OS

  users are aware of multiple machines

  remote log, remote desktop, FTP

- distributed OS

  seems like a single machine to user

  - data migration: transfer necessary files
  - compute migration: compute on remote machine and return results
  - process migration: execute programs on different sites
    - load balance
    - speed up
    - hardware request (e.g. printer)
    - software request (e.g. slurm)
    - data access (use remote data)

## network structure

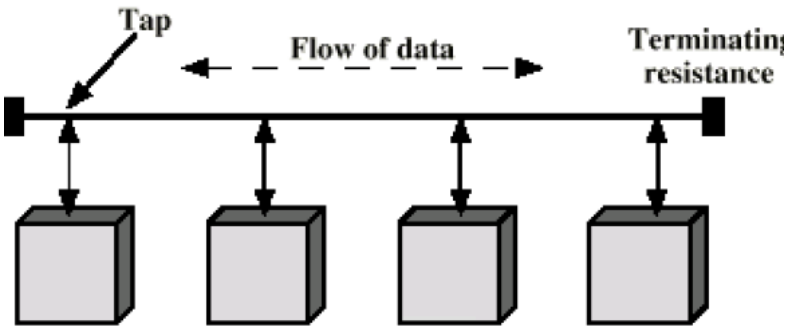
- LAN (local-area network)

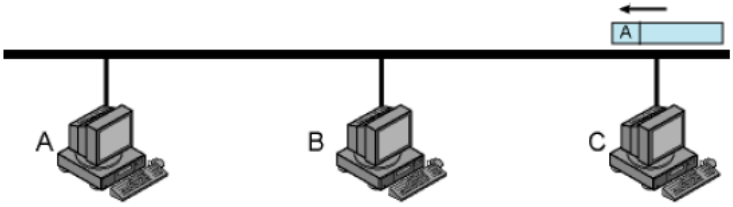
  faster, usually among personal computers or a few mainframes

  - bus topology

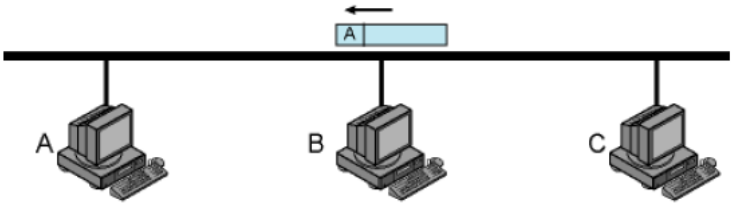
    stations attach to bus(linear medium)

    message receive by all other stations but only target will take

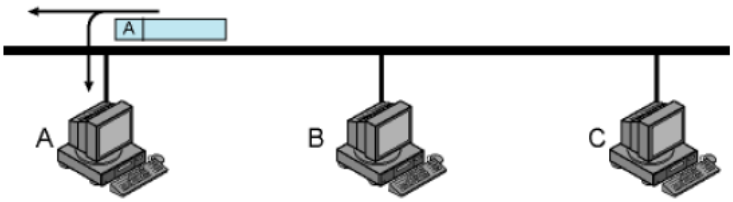



C transmits frame addressed to A




Frame is not addressed to B; B ignores it




A copies frame as it goes by

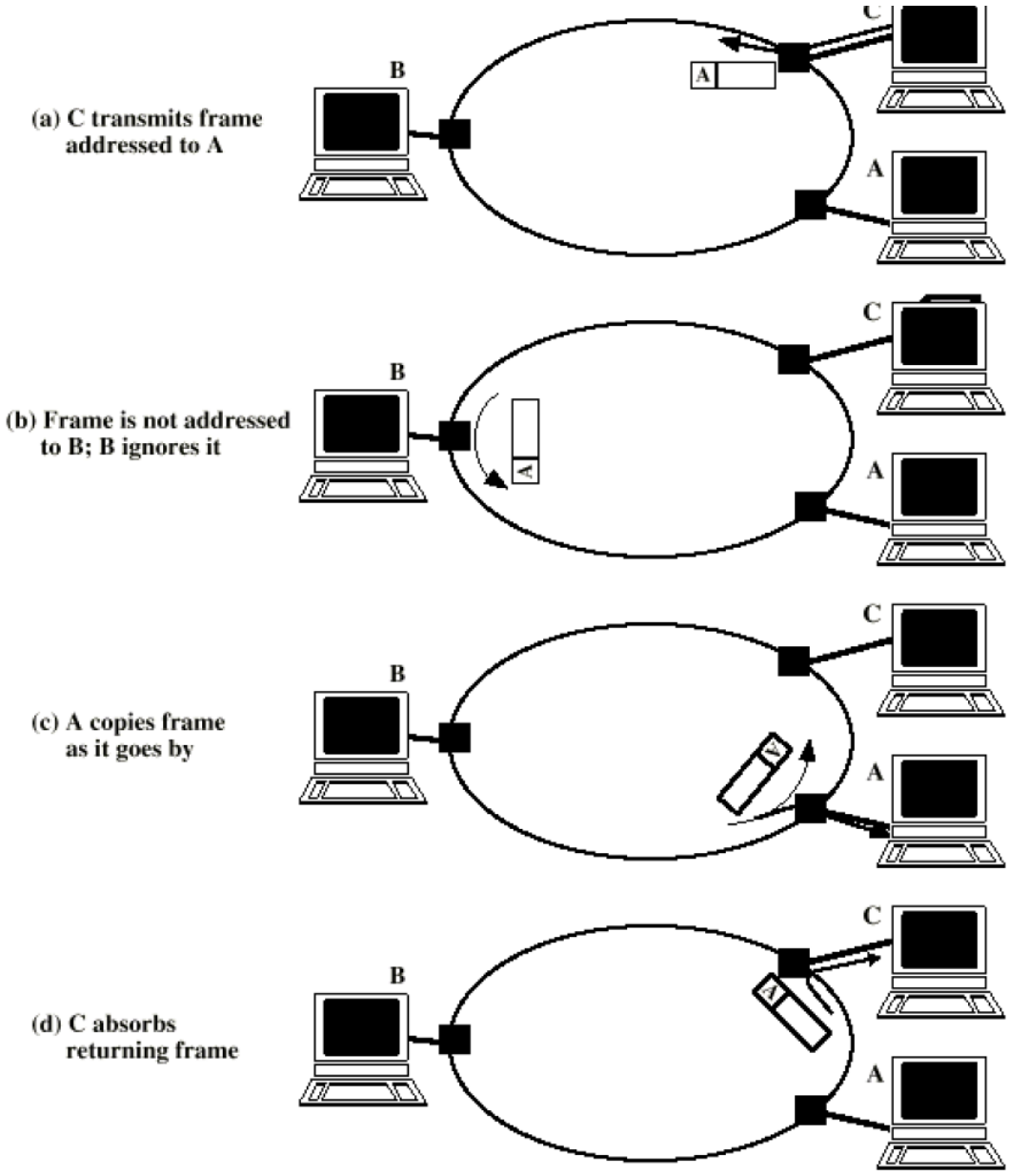
need to regulate transmission

- avoid collision: two stations' message overlap
- avoid starvation: someone continuously transmits

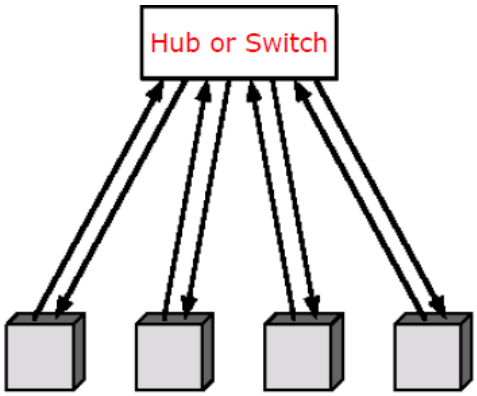
  solution: transmit in small blocks(frames)

- ring topology

  stations attach to repeaters



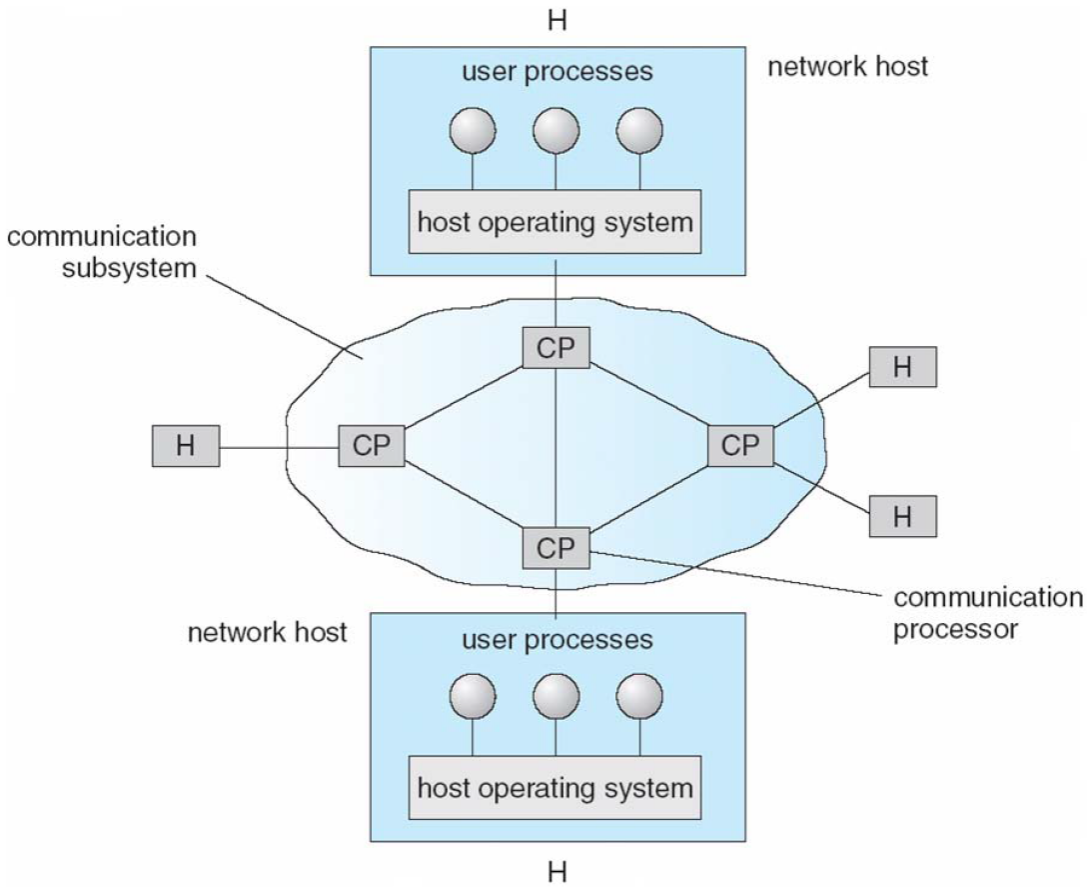

- star topology




only one node can transmit at a time

- WAN (wide-area network)

  p2p connection over long-haul(长途) lines




network topology

fully-connected, partially-connected, tree-structed, star, ring

- installation cost
- communication cost (latency)
- reliability (if a link is lost)

## communication structure

- naming and name resolution

  <host-name, id>

  DNS

  on OSI 7-layer model

  - application layer: [www.sjtu.edu.cn](http://www.sjtu.edu.cn)
  - transport layer: <IP, port>
  - network layer: IP
  - data layer: MAC address
- routing strategies (整个网络)

  how is message sent by the internet?

  - fixed routing
  - virtual routing

    fixed during one session
  - dynamic routing

    every message choose its own path

    messages may arrive out of order

- connection strategies (2个节点)

  how can 2 processes communicate mutually?

  - circuit switching

    establish a permanent physical link

  - message switching

    establish temporary link during one message transferring

  - packet(包) switching

    message is divided into fixed-length packets

- contention(竞争)

  - CSMA/CD

    A site determines whether another message is currently being transmitted over that link.

    If two or more sites begin transmitting at exactly the same time, then they will register a CD(collision detection) and restart after random interval.

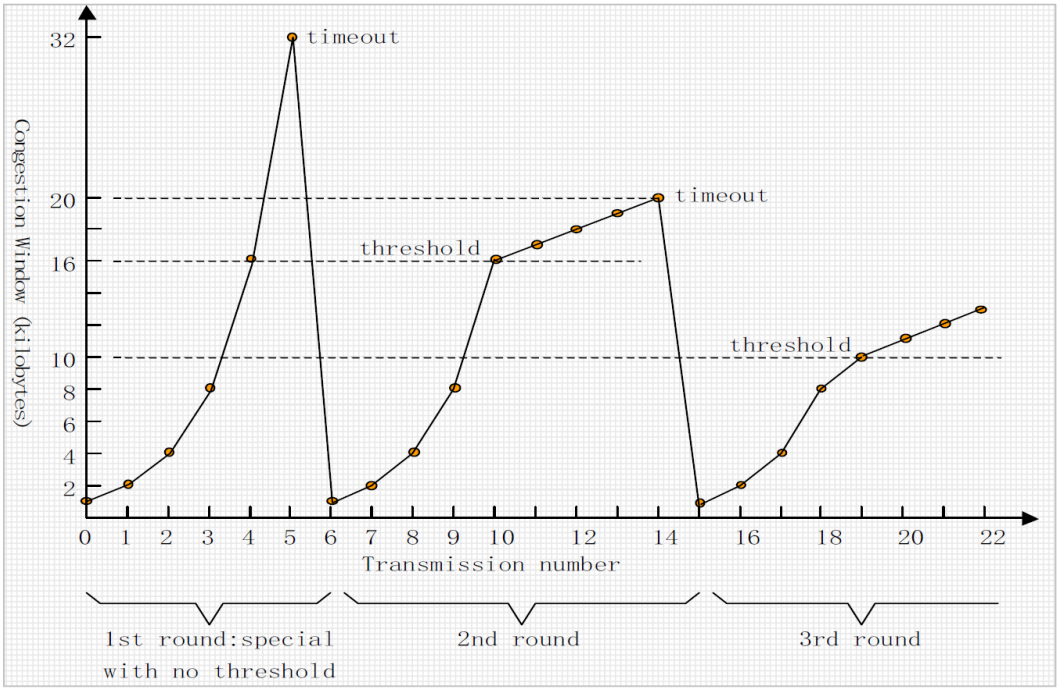
  - token passing

    need token to transmit information

  - message slots

    fixed-length message slots continuously circulate in the system

  - TCP congestion control(拥塞控制)




## communication protocol

- physical layer (electric details, bit stream)
- data-link layer (handle frames, e.g. PPPoE, Ethernet)
- network layer (packets, IP, IPsec, ICMP)
- transport layer (TCP, UDP)
- session layer
- presentation layer (SSL, TLS)
- application layer (HTTP, FTP)

### robustness

**failure detection**

**handshaking** protocol

exchange I-am-up from time to time, otherwise send are-you-up

site A cannot determine exactly why the failure has occurred

**reconfiguration**

failure occur

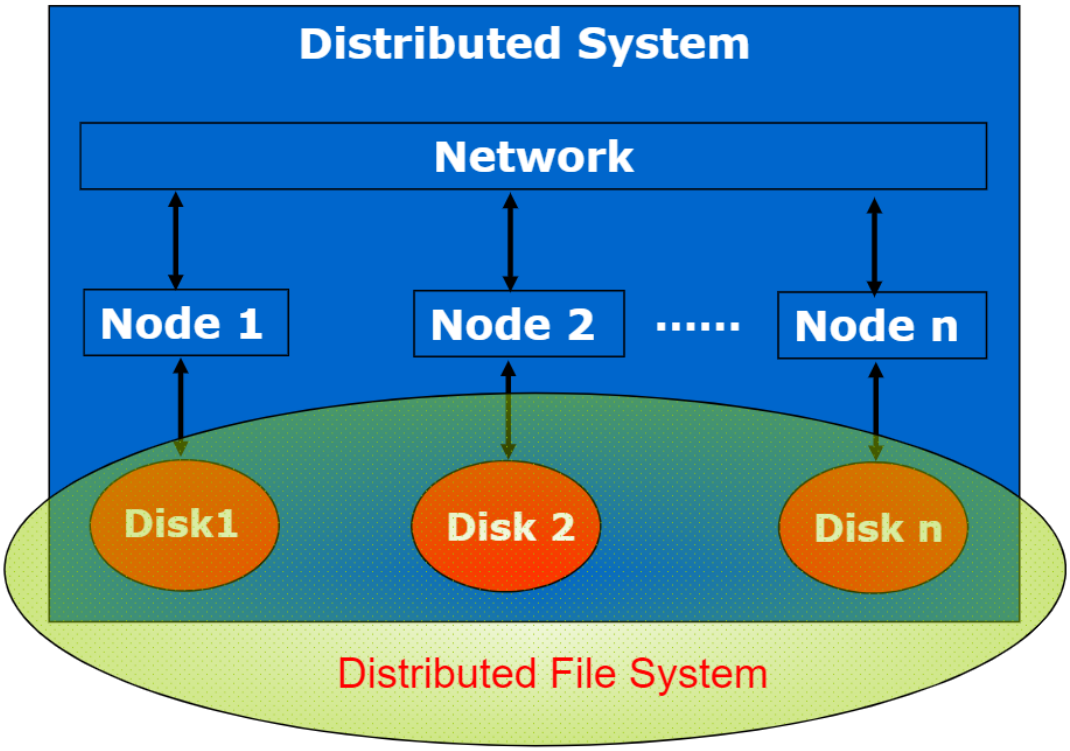
link failed / site failed, broadcast to others

### designing issues

- transparency: appear as a conventional, centralized system
- fault tolerance
- scalability
- clusters: a collection of semi-autonomous machines that acts as a single system

## Chap12: Distributed File System

DFS is composed of different, remotely located, smaller storage devices.




Disk(1~n)➔ many component units ➔ A file system

## DFS structure

- service: software providing file system
- server: where service runs
- client: invoke service

  client interface: primitive file operations, should be transparent(not distinguish remote and local)

## naming and transparent

naming: map logical filename to physical file

multi-level mapping: hides detail of how and where is stores

- first level: file name
- second level: block (inode)
- more levels if there are replicas

transparent DFS hides where file actually is stored

**naming structure**

- location transparency: file name not reveal physical storage
- location independence (stronger): file name is not bounded to physical location

**naming schemes**

- combination of host name and local name

  neither location transparency nor independence
- attach remote directories to local directory
- total integration of the component file systems

  global name spans all related files

## remote file access

remote-service mechanism: use RPC (remote process call)

reduce network traffic by cache

**cache location**

- disk

  more reliable
- main memory

  diskless, more quickly

**cache update policy**

NFS use different policies for directory(write-through) and data(write back)

- write-through
- delayed write (write back)

  poor reliability

  - scan cache at regular intervals
  - write-on-close

**consistency**

- client-initiated approach

  client initiates a validity check and server checks consistency with master copy

- server-initiated approach

  server records files that each client caches and notice client for possible collision

**comparing caching and remote service**

- many remote accesses handled by cache, only connect server occasionally in caching, but remote service handles every request across network
- network overhead in transmitting big chunks of data (caching) is lower than a series of responses to specific requests (remote-service)
- with frequent write, cache-consistency problem may cause overhead in network
- caching mainly used on those with local disks or large main memory, while remote service used on those with no disk or small memory

## stateful vs stateless

- stateful

  client use `open()`

  server gives client a unique id (until session ends) and opens file

  server must invoke garbage collection

- stateless

  avoid state information by making each request self-contained

  each request identifies the file and position in the file (read or write)

  no need to `open()`

**distinctions**

- stateful increase performance (read ahead for sequential access)
- failure recovery

  stateful server loses all information

  - need a recovery protocol or abort all operations
  - need to be aware of client failures to free allocated space
- stateless server
  - longer request message
  - slower request processing
- stateful required environments
  - server-initiated cache validation
  - UNIX: file descriptors and implicit offsets

    servers must maintain tables to map the file descriptors to inodes

## file replication

replicas of the same file reside on failure-independent machines

improves availability and can shorten service time

**naming** scheme maps a replicated file name to a particular replica

- existence of replicas should be invisible
- replicas must be distinguished from one another by different lower-level names

**operations**

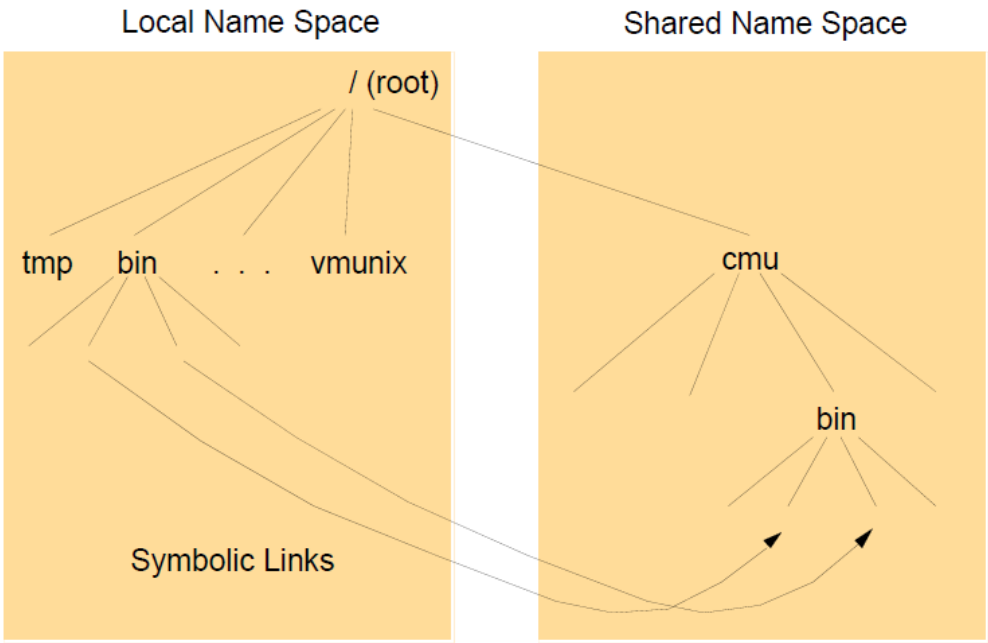
- update

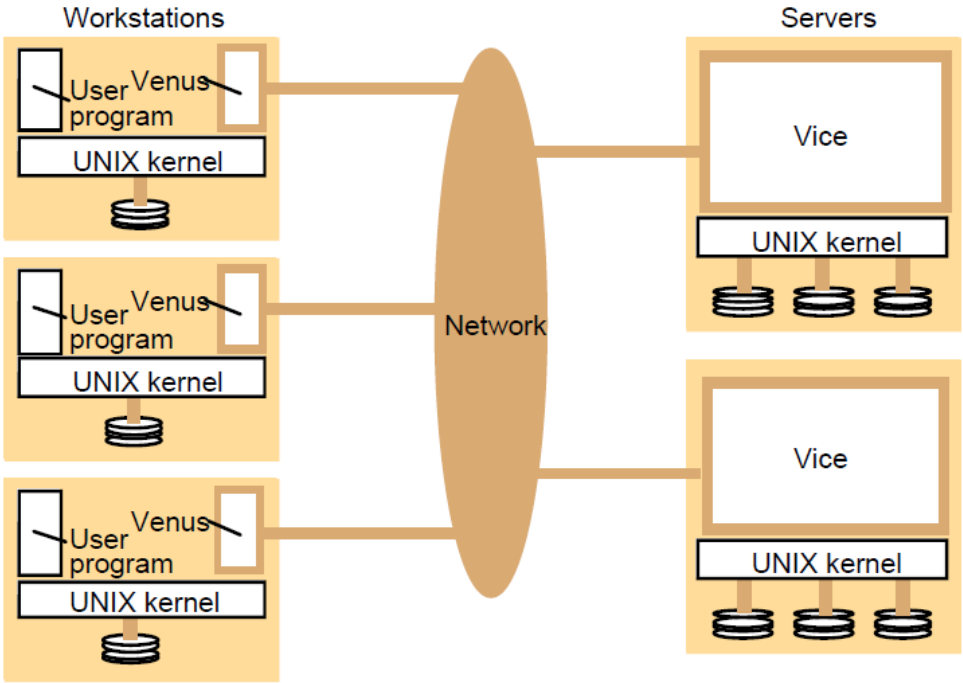
  an update to any replica must be reflected on all other replicas

- demand replication

  reading a nonlocal replica causes it to be cached locally

## AFS




vice is server while venus is client

servers manage storage and shared name space

cluster (in LAN): a cluster server + several workstations(clients) + a router

- naming

  local name space and shared name space (attach to local name space)

  volumes are small component units associated with the files of a single client

  fid = volume number + vnode number + uniquifier

  location independent (maybe move volume between servers doesn't change volume number)

  location information is kept on a volume basis, and the information is replicated on each server

- remote file access

  `open` and `close`

  caches files from Vice when they are opened, and stores modified copies of files back when they are closed

  however, modifications to directories that are made directly on the server responsibility for that directory

- caching
  - status cache: rapid finding file status and position
  - data cache: whole file caching
- consistency

  venus assume all cache are valid

  **callback**

  - when client open a file, server updates related information and this is a callback
  - before server allow another client to modify this file, it remove previous callbacks