OS Structures

Types
- interface to services
- system calls

Device management
- read/write/reposition
- request/release devices
- get/set device attributes
- logically attach/detach device

File management
- create/delete files
- open/close files
- read/write/reposition
- get/set file attributes

Process control
- allocate/free memory
- wait/signal event
- get/set process attributes
- load, execute
- create/terminate process

Information maintenance
- get/set system data
- get/set time/date
- get/set process, file, device attributes

communications
- create/delete communication connection
- send/receive messages
- transfer status information
- attach/detach remote devices

protection
- get/set file permissions

Overview

Users
↓↑
User Interfaces
↓↑
System Calls
↓↑
System services
↑↓
Operating system
↓↑
Hardware

OS services
- protection and security
- logging
- resource allocation
- operation of system

Users
- I/O
- program execution
- communications (processes)
  - shared memory → conflicts
  - fast
  - message passing → for small data
  - no conflicts
- error detection
- file-system manipulation

User Interfaces
- Command-Line Interface / command Interpreter
  - get and execute user-specified command
  - approach
    - implements commands through system programs
      - small size
      - CLI doesn't change for new command
    - CLI contains code (size = code size)
      - large size
- Touch screen Interface
- Graphical User Interface GUI

host: creates illusion (process has own processor and virtual memory)
guest: provided virtual copy of underlying computer

- layered approach
- hardware + kernel = hardware

Virtual machine

- set of core components
- link services via modules

Loadable Kernel Modules LKMs

modules

Hybrid Systems

*modern

*for network + web app

advantages → changing 1 layer only affects itself

↓ simple to construct + debug

Layered Approach

- layer = data + operations

disadvantages → ↓ performance (overhead to traverse layers)

- hard to define

Structure

easy to port

advantages

↓ easy extended

↓ security + reliability

disadvantages

↓ slow communication ∴ user-level service

microkernels

- remove all non-essential components from kernel

↓ put them into user-level

↓ performance ↑

fast communications within kernel

monolithic

- all functions of kernel are in a file of single address space

↓ difficult to implement/extend

↓ tightly coupled

program loading and execution

Programming-language support

File modification

status information

communications

-provide environment for program development and execution

File management

Background services (daemons/subsystems)

system services/utilities

OS Structure 2

run-time environment

RTE

interpreter

provides

compiler

library loader

system-call interface

- link to system calls

user application

user ↓ ↑

system call interface

kernel ↓ ↑

system call

Application Programming Interface API

- specifies set of functions available

→ parameters passed to function

→ return values

→ benefits → program portability

→ system calls are difficult to deal with

Design

system goals → efficient

easy to use

user goals → safe → fast

mechanisms and policies

↓ - how

↓ - what - changeable

implementation

- use high-level language

↓ easy to port

↓ reduced speed storage ↑ *minor

Linkers and Loaders

→ steps

source files

↓ compile

relocatable object file

↓ linker combines

other object files/libraries

binary executable file

↓ loader

dynamically linked libraries → memory

method to pass parameters to OS

→ pass to registers

↓ small data

↓ address passed

registers

→ parameters stored in memory

↓ large data

stored in a stack

↓ do not limit no. of parameters

infrequently, slow

- select from job queue to ready queue
Long-term / job

- set of all processes
job queue

- waiting for I/O device
device queue

wait queue
- waiting for event

ready queue
- ready to execute

frequently invoked, fast

Short-term / CPU scheduler
- select from ready queue to execute

Schedulers

Scheduling Queues

Bounds → I/O bound process

CPU bound process

memory management information

process priority

pointers to scheduling queues

accounting information

CPU scheduling information

CPU registers

I/O status information

Process / Task Control Block PCB

program counter
- address of next instruction

process state

process number

process state
process number
program counter
Registers
memory limits
list of open files

PCB

Process Scheduling
- maximize CPU utilization

Context Switch
* time-dependent on hardware

Processes
- program in execution
- unit of work

* only 1 process can be running on core

Components

Process State

terminated
- finish execution

ready
- waiting to be assigned to processor

waiting
- waiting for event

- current activity of process

running
- instructions being executed

new
- being created

process P_0

executing → interrupt → save state into PCB_1 → idle

reload state from PCB_2 → process P_1 → executing

idle

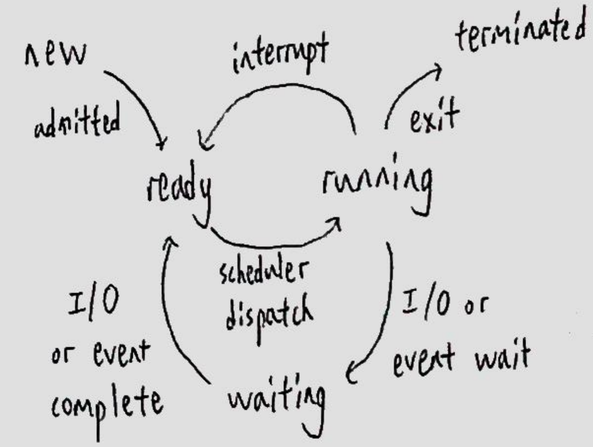
stack → temporary data storage when invoking functions

- dynamically changing
- no overlap

heap → dynamically allocated memory

data → global variables

text → executable code



Processes 2

- need both names symmetry
- need one name only link 22 processes asymmetry
- symmetricity
- link automatically with identity properties
- 1 link per pair
- only link 2 processes
- direct
  - explicitly name recipient/sender
  - change 1 name * limited modularity
  - change all code
- properties
  - each pair has different no. links
  - link only if have shared mailbox
- indirect
  - mailboxes, ports
- direct/indirect
- communication link
- can be used across network
- message passing
  - easy to implement
  - small data
  - slow
- mechanism
- fast
- shared memory
  - in creater process address space
  - attach to
  - other process address space
- A / B / message queue
- A / shared memory / B
- Buffering
  - 0 capacity - must block
  - bounded capacity - block if full
  - unbounded capacity - never block
- synchronization
- blocking/ synchronous
  - send - block till message received
  - receive - block till receive
- nonblocking/ asynchronous
  - receive - receive message/ null
  - send - immediately resume after send
- Interprocess Communication IPC
  - cooperating - affect / affected by other processes
    - why?
      - computation speedup - break task into subtasks
      - modularity - divide system functions
        - processes
        - threads
      - concurrently access information
  - independent - not sharing data
- Operations on Processes
- Process Creation
  - parent process creates children processes
    - Identification - pid (integer as unique process identifier)
    - resource sharing
      - no sharing
      - share all
      - children share subset of parent
    - execution
      - concurrently
      - parent wait for termination of child
    - address space
      - child is duplicate (same data and program) * easy to communicate
      - new program loaded in child
- Process Termination
  - child finish execution (call exit())
    - return status value
    - waiting parent
  - Types
    - zombie process: child terminate but parent not call wait() yet
    - Parent terminate child
      - why?
        - child exceeds usage of resources
        - task assigned not required now
        - parent exit - cascading termination
          - if children x terminate
            - orphans

Multithreading Model

- many-to-many model
  - user ≤ kernel (multiplex)
  - difficult to implement
  - combine advantages of 1-1 and many-1
- one-one model *most used
  - too many kernel threads, burden system
  - more concurrency
- two-level model
  - user ≤ kernel + user = kernel
- Many-to-one model
  - entire process blocked if a thread make blocking call
  - can't parallel
  - Thread management done by thread library, efficient
  - user threads
  - kernel thread

Threads

- basic unit of utilization
- components
  - register set
  - thread ID
  - Program counter
  - stack

code	data	files
register	r2	r3
stack	s2	s3
∫	∫	∫

Benefits

- resource sharing
- Responsiveness
  - response to user when doing other task
- economy
- increase concurrency scalability

Thread libraries

- API for creating / managing threads
  - in user space
  - in kernel-level
    - function call
    - system call

strategy of multithreading

- asynchronous threading
  - parent-child run concurrently + independently
  - little data sharing
- synchronous threading
  - parent waits for all children terminate
  - data sharing

motivation

- Client-Server efficiency↑
- multithreaded kernels
  - process creation heavy but thread creation light-weight
- application do multiple tasks

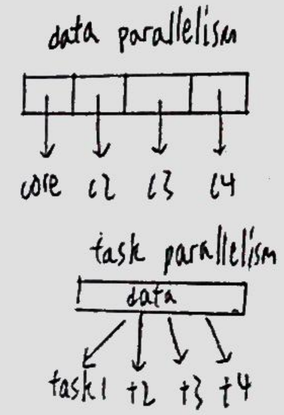
Programming Challenges

- Identifying tasks
  - find independent tasks
- Balance
  - value vs time
- data splitting
  - divide data
- data dependency
- Testing and Debugging

concurrency vs parallelism

- multi core
- concurrency and parallelism - different threads run on different cores
- single core concurrency
  - use one core and switch tasks (interleaving) rapidly to make progress on all tasks

Types of Parallelism



upcall handlers
↑ handled by
upcall
←
user thread — lightweight process LWP + applications — kernel thread — Scheduler Activation

signal
generate → handled
process (* same process)
synchronous

Signal Handling → where?
- certain threads
- every thread
- specific thread to handle all signals
- signal applied thread

Signal Handling → Steps
1) signal generated (event)
2) signal → process
3) handle signal
- default signal handler (every signal has)
- user defined signal handler (override default)

Signal Handling
- asynchronous
- external signal

Threads 2
- Thread pools ← advantages ← no. of threads bound to size of pool
  - create threads in a pool where they await work
  - faster to service
  - existing thread > create thread
- Threading Issues
  - Thread-Local Storage TLS
    - useful when no control over thread creation process
      - e.g. thread pools
    - create data unique to each thread
  - fork()/exec() system calls
    - 2 versions
      - duplicate the thread invoked fork()
        * used if call exec() after fork()
      - duplicate all threads
    - replace entire process with program specified

Thread Cancellation
- target thread: thread to be cancelled
  - deferred cancellation
    - periodically checks if need to terminate → default
      - can safely reclaim resource
  - asynchronous
    - immediately terminates
      - resources may not reclaimed
      - not recommended

Thread scheduling

user-level threads → thread library schedules → LWP → kernel threads → CPU scheduler → CPU

* by programmer

process contention scope PCS

SCS system contention scope

(since don't know length of next CPU burst) can't be implemented in real world

predict exponential average

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

predict value ← length of nth CPU burst

disadvantages
- all processes wait for 1 big process
- convoy effect

Shortest-Job-First SJF

min average waiting time

advantage

Algorithms

disadvantages
- average waiting time

First come first served FCFS

advantages
- simple to implement (FIFO queue)

Round-Robin RR
- similar to FCFS
- preemptive

Basics
- time quantum/slice
- 1) start run 1 quantum/interrupt
- 2) put to tail
- 3) do next
- wait $\leq (n-1) \times q$
- do work $\leq q$
- disadvantages
  - long average waiting time

circular ready queue

Priority Scheduling
- sort by priority

Problem
- indefinite blocking/starvation

solution → RR + PS
- gradually increase priority
- aging

Multilevel queue
- separate queues for distinct priority
- different algo for each queue

*most complex

Multilevel Feedback Queue
- same as multilevel queue
- but
  - lower p too long → higher p
  - higher p don't finish → lower p
  - allows process to move between queues
  - parameters
    - method to ↑↓ process
    - method to enter queue
    - no. of queues
    - scheduling algo for each queue

CPU Scheduling

Basic concepts
- CPU/I/O burst cycle
  - process consists of cycle of CPU execution and I/O wait
- CPU Scheduler
  - select from ready queue
- dispatcher
  - give control of core
  - process selected by CPU-scheduler
  - involves
    - switching context
    - switching to user mode
    - jumping to location in user program to resume
  - dispatch latency
    - time to stop and start another process

Criteria

$t_{complete} - t_{submit} =$ Turnaround time
- ↓ better

Throughput
- work per unit time

CPU utilization
- as busy as possible

response time
- $t_{response} = t_{submit}$

time in ready queue
waiting time

Decision
- preemptive
  - waiting → ready
  - running → ready
- non-preemptive/cooperative
  - terminates
  - running → waiting

Process Synchronization

- to maintain data consistency

Problem

- race condition: several processes access and manipulate same data concurrently
  - causing outcome depends on order of accesses (indeterministic)

Process structure

```
while(true){
    entry section (request permission to enter critical section)
        critical section (segment of code using shared data)
    exit section
        remainder section
}
```

solution requirement

- mutual exclusion
  - only 1 process in critical section
- progress
  - only processes not in remainder section can enter critical section, and selection can't postponed indefinitely
- bounded waiting
  - limit on time to wait for entering critical section after request

Peterson's solution

- for 2 processes only
  - * don't work nowadays
  - * software-based

shared variables

- int turn (whose turn to enter)
- bool flag[2] (ready or not)

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

Hardware support

- atomic variables
  - atomically update data

memory barriers / fences

- force any changes in memory to be propagated to other processors

Hardware instructions

- all atomic

test and set

```
def
bool test_and_set(bool *x)
    bool rv = *x;
    *x = true;
    return rv

do {
    while(test_and_set(&lock))
        wait
    critical section
    lock = false
    remainder section
} while (true)
```

compare and swap

```
def
swap(bool *a, bool *b)
    bool temp = *a
    *a = *b
    *b = temp

key = true
while (key == true) swap(&lock, &key)
critical section
lock = false
remainder section
```

Semaphores

- int variable

Types

- counting S
  - control access to given resource
- binary S
  - same as mutex locks

operations

- wait(S)

```
while(S<=0)
    wait
    add process to waiting queue
    sleep
S--
```

- signal()

```
S++
remove P from waiting queue
wakeup()
```

Mutex locks

- mutual exclusion

```
while (true)
    acquire lock
    critical section
    release lock
    remainder section
```

```
while (!available)
    wait
available = false

available = true
```

- disadvantages
  - busy waiting (looping to acquire lock)

contention

- low contention
- high contention
  - ↓ performance

uncontended

- available

contended

- thread blocks

operations

- acquire
- release

process accessing resources S needed by high priority process share priority until S released

↑

solution: priority-inheritance protocol

↑

Process Synchronization 2

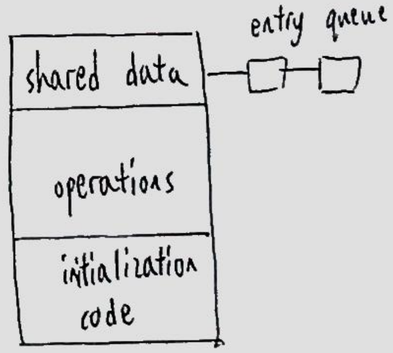
↑

- liveness
  - set of properties system must satisfy to ensure processes make progress
  - Priority inversion
    - L < M < H
    - L running, H wait for L, M interrupts, M indirectly interrupts H
  - Deadlock
    - every process in set is waiting for event which can only caused by another process in set

↓

- Monitors
  - resuming processes
    - priorities
    - FCFS
  - ensure correctness
    - uncooperative process does not ignore mutual exclusion
    - user processes calls monitor in correct sequence
  - Properties
    - monitor type - abstract data type that includes set of programmer defined operations provided with mutual exclusion
    - only 1 process is active in monitor
  - structure

```
monitor x
{
    shared variable declarations
    procedure ...
    Initialization code
}
```



- condition variables
  - operations
    - wait()
      - process invoked wait() suspended until signal()
    - signal()
      - resumes suspended processes
  - possibilities
    - signal and continue
      - P → signal, P → continue
    - signal and wait
      - P → signal, P → wa
    - compromise
      - P → signal, P → leave
      - ↓
      - Q → resume

Deadlock

- every process in a set of processes is waiting for an event that can be caused only by other process in the set

necessary condition

- mutual exclusion
- hold and wait
  - process holding resource and waiting other resource held by other friends
- no preemption
  - resource can only be released by process holding it
- circular wait
  - $P_0 \to P_1 \to \dots \to P_n \to P_0$

resource allocation graph

- vertices
  - resource R
  - process P
- edges
  - assignment edge: $R \to P$
    - *1 instance 1 edge
  - request edge: $P \to R$

cycle

- multiple instances per R
  - may have deadlocks
- 1 instance per R
  - deadlock

no deadlocks $\leftrightarrow$ no cycle

Prevention

- mutual exclusion
  - must hold
- hold and wait
  - ensure when process requests resource, it doesn't hold resources
  - requires each process to request and allocate all resources before execution
    - *impractical
- circular wait
  - assign id to R
  - only request resources in increasing order
- no preemption
  - request
    - allocate available
    - not available
      - other process holding it and waiting
        - preempt
      - not held by waiting process
        - wait (may get preempted)
          - restart when get old + new resources

disadvantages

- low resource utilization
- starvation

allow process to request resources only when process has none

Avoidance

- requires priori information
- ensure system don't enter unsafe state

unsafe state

- may have deadlocks

safe state

- system exists safe sequence
  - if for each P_i, request can be satisfied by currently available resources + resources held by all P_j, $j < i$
    - if P_i request not immediately available, wait P_j finish
      - P_i get resources, do work
        - P_{i+1} get resources

Algorithm (1 instance 1 resource)

- claim edge $P_i \to R_j$: P_i may request R_j in future
  - assignment $\to$ claim
    - release
  - request
    - claim
      - request
    - else wait
    - can be granted if
      - $P_i \to R_j \Rightarrow R_j \to P_i$ don't form cycle
        - *check cycle $O(n^2)$

resource allocation state

- no. of allocated resources
- maximum demands
- no. of available resources

Deadlock

Recovery
- preempt and give R to other P until deadlock cycle broken
resource preemption
- abort all deadlocked processes
- expense ↑
- abort 1 at a time until no deadlock cycle
overhead (check everytime remove 1 process)

factors of policy to choose
- how many P to terminate
- priority
- how long it ran + how much longer to finish
- how many and what type of R used
- how more R is needed

Rollback → to where
how to restore

issues
- select victim - which R/P
- starvation - not preempted from same process

cost
factors → amount of time
↳ no. of R

Detection
- several instances of R
1) Work = available
Finish[i] = false if allocation$_i$ ≠ 0
else true
2) find i such that
a) finish[i] == false if no i, go (4)
b) Request$_i$ ≤ work
3) Work = allocation$_i$ + Finish[i] = true
go (2)
4) If no any i
Finish[i] == false
deadlocked

- wait-for graph (single instance)
remove R nodes
$T_i \rightarrow R_q$, $R_q \rightarrow T_j$
$T_i \rightarrow T_j$
- maintain wait-for graph + periodically invoke algo to search cycle $O(n^2)$

Banker's Algorithm
(deadlock avoidance for multiple instances per resource)
- when new P enters, declare max. no. of instances of each R type (≤ total no. of R)
- system determine safe/unsafe
  - safe - allocate
  - unsafe - wait

Safety Algorithm
- to check state (safe/unsafe)
$O(m \times n^2)$
1) Work (m-length vector)
Finish (n-length vector)
Work = Available
Finish[i] = false
2) Find i such that
a) Finish[i] == false
b) Need$_i$ ≤ Work
if no i, go to (4)
3) Work = Work + Allocation$_i$
finish[i] = true
go to (2)
4) If finish[i] == true for all i, safe

Resource-Request Algorithm
request[i][j] = k
↕
P_i want k instances of R_j
1) if request$_i$ ≤ need$_i$ → go (2)
else error
2) If request$_i$ ≤ available → go (3)
else wait
3) system pretend to allocate
- change state to
available -= request$_i$
allocation$_i$ += request$_i$
need$_i$ -= request$_i$
- safety algorithm → safe, complete
- restore state + wait
unsafe

Data Structures
- Available
m-length vector
* no. of available instance for each R
available[j] = k
↕
k instances of R_j is available
- Max
n×m matrix
* max demand of each P
Max[i][j] = k
↕
P_i needs at most k instances of R_j
- Allocation
n×m matrix
* no. of resources of each R_j currently allocated
Allocation[i][j] = k
↕
P_i allocated k instances of R_j
- Need
n×m matrix
* remaining resource needed
Need[i][j] = k
↕
P_i needs k more instances of R_j

Main Memory

- contiguous memory allocation
  - each process is stored contiguously
  - 1 partition - 1 process memory allocation
    - 1) process arrives, system search a hole
      - first fit
        - choose first suitable hole
      - best fit
        - choose smallest available hole
      - worst fit
        - choose largest available hole
      - need to search whole list
      - dynamic storage allocation problem
        - request of size n from list of free holes
    - 2) if found, split the hole if the hole is too large
    - 3) allocate memory to hole
    - if not found, wait or reject
  - protection
    - relocation register
      - smallest physical address
    - limit register
      - range of logical address
      - logical address < limit + relocation → physical address
  - execution time
    - use relocation register (added to address at run time)
    - can be moved from segment to segment
    - need special hardware
      - memory-management unit MMU
        - user program
        - logical address
        - MMU
        - physical address
        - physical memory
  - address binding
    - compile time
      - if know location of process in memory
      - generate absolute code
      - if changes occur
      - recompile
    - load time
      - unsure location at compile time
      - generate relocatable code
      - if changes occur
      - reload user code
    - logical address = physical address

- Basic hardware
  - protection (make sure each process has separate memory space)
    - base register
      - smallest legal physical memory address
    - limit register
      - size of range
    - ≥ base and < base + limit
      - legal address

- address space
  - set of address
    - logical
      - generated by CPU
    - physical
      - loaded into memory-address register

- all routines kept on disk *relocatable load format
  - loaded into memory when called

- Dynamic loading
- Dynamic linking and shared libraries DLLs
  - system library linked to user programs at run time
    - advantages
      - can extended to library updates
      - share library
      - reduce size (only 1 copy of library stored)

add page no., frame no. to TLB → replacement policy (some entries are wired down in TLB) — full

hit, get frame no.

memory reference to page table ← miss ← TLB check ← steps

page-table length register PTLR
- size of page table

page table base register PTBR
- points to page table

stored in main memory

translation look-aside buffer TLB
- small, fast cache

compared with all keys simultaneously

Hardware

context-switch time ↑

stored in dedicated high-speed registers

for small size only

fast ↑

OS keeps frame table (1 entry ↔ 1 page frame)

available/not

no. of frames

divided to fixed size blocks (size = frame size)

4) frame no. → page table

3) pages loaded into frames from source (file system/backing store)

2) if frame available, allocated

process → page table

steps
* pointer to page table is stored in PCB

1) examine size
1 page = 1 frame

Page Table
- contains base address of frames

Main Memory 2

→ Paging
- break physical memory into frames
- break logical memory into pages
> same size blocks

still cause internal fragmentation

↓ page sizes
↕ → but ↑ overhead into page table

↓ internal frag

allocated memory > requested

hole size = allocated - requested

physical address

frame no.	offset
m-n	n

totally separated → logical address

page no.	offset
m-n	n

if logical address space = 2^m
page size = 2^n

CPU → LA → page no. / offset

page table
- extract frame no.

PA
frame no. + offset

* conversion of LA → PA

Fragmentation
- memory space broken into little pieces

internal

caused by allocation (unavoidable)

external

caused by first/best fit in contiguous allocation

have enough total memory space but not contiguous

solution

compaction
- shuffle memory to let free memory become 1 hole → * only possible if relocation is dynamic

use noncontiguous logical address space

* 50-percent rule: 1/3 of memory space may be unusable

Structure of Page table

forward-mapped/ hierarchical
- page the page table

Hashed
- hash value = page no.

page no.

hash table → value of mapped page frame

pointer to next element

clustered page tables
- several pages in 1 entry of hash table

variation

Inverted
- 1 entry ↔ 1 frame

only 1 page table in system

entry consists of VA stored in real memory location + info of process

no sharing

VA = <pid, page no., offset>

page table entry = (pid, p no.)
Page table index = frame no.

other info

valid bit

protection bit

- parallelly check when computing PA

TLB
solution
2 memory read

hash table
solution
↑ time to search

modify / dirty bit
- when victim frame selected

= 1
- modified
- need to write to storage

= 0
- clean
- don't write

Virtual Memory

Demand Paging
- load pages only as they are needed
- pure demand paging: only load page when it is required
- certain features are rarely used
- examples
  - needed programs are not needed at same time
  - data structures which allocated much more memory than needed
  - error handle code

access page, check valid bit
- valid, use it
- invalid
  - page fault or invalid reference
    1) check internal table
    2) if it's invalid reference, terminate else page in
    3) find free frame
    4) read from secondary memory
    5) modify page table and internal table
    6) restart instruction

free-frame list
- list of free frames maintaine by OS
- zero-fill-on-demand technique
  - erase previous content of frame before allocated

major tasks
- service page fault interrupt
- read in the page
- restart the process

- heavily depends on page fault rate performance

demand page from file system initially
* common
- only needed pages read from file system
- write the pages to swap space when replaced

handling swap space (* swap space generally faster than file system)
- copying entire file image into swap space at startup
  - heavy load at start up
- demand paging from swap space

Page replacement steps
1) find location of page on secondary storage
2) find a free frame
- not found
  - use page replacement algorithm to select victim frame
  - write victim frame to secondary storage
  - change page, frame tables
- found, use it
3) read page into freed frame
- change page, frame tables
4) continue process from page fault

Background
- not all program are needed in memory
- we put only part of program in memory

Benefits
- more programs can run at same time
- allow sharing memory space
  * mapping shared object into virtual address space
- less I/O used to swap
- not constrained by physical memory

copy-on-write
- allow parent and child processes initially share same pages
- copy-on-write pages
  - only create copy when written

speed of process creation ↑

Page Replacement Algorithm

LRU Approximation

- set when page is referenced
- reference bit

Additional-Reference-Bits

- under regular intervals
- shift reference bit into high order bit of 8-bit
- keep 8-bit for each page in memory
- 8-bit as unsigned → smallest
- LRU

Second-chance

- =0 replace
- =1 set reference bit = 0 → select next
- (0,0), best replace
- (0,1), need to write

enhanced second-chance

- consider reference bit and modify bit as ordered pair
- (1,0) - likely to be used again
- (1,1) - very likely to be used again

implement

- circular queue
- pointer point to next-to-be replaced page

LRU

Least Recently Used

- replace page has not been used for longest time

implement

- * need hardware support
- counters
  - record time-of-use
- stack of page no. (doubly linked list)

FIFO

First In First Out

- use a queue to store frames

* Belady's anomaly
  - ↑ frames ⟷ ↑ page-fault rate in some cases

Optimal Page Replacement / Farthest in Future

- lowest page-fault rate
- replace page that will not be used for the longest period of time

difficult to implement
- need future information

Stacking algorithm

- set of pages for n frames is always a subset of set of pages for n+1 frames

Counting-Based

LFU

Least Frequently Used

MFU

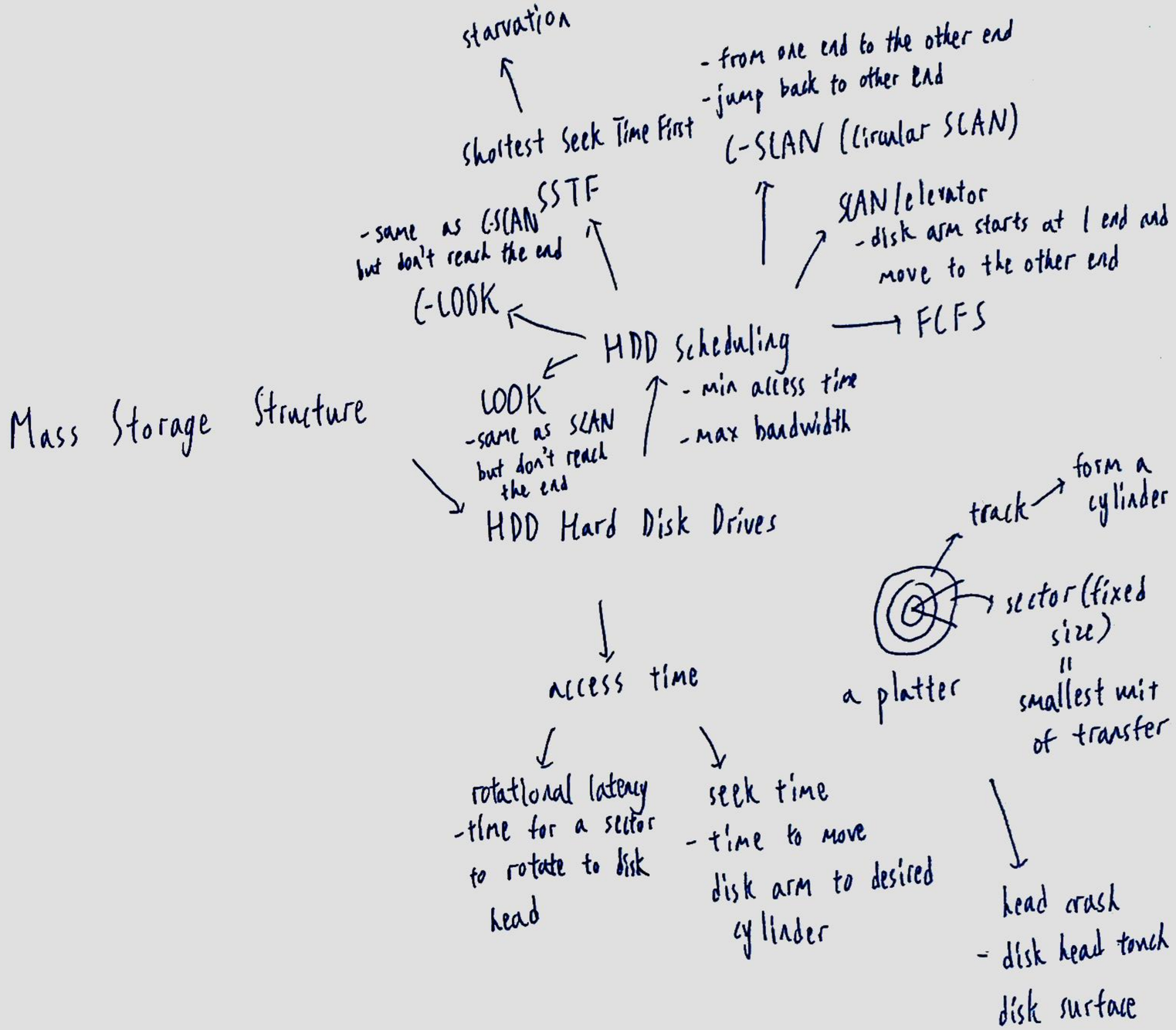
Most Frequently Used

expensive, useful under special scenarios

Page Buffering

- store pool of free frames
- read page into free frame ↕ parallely write victim out
- victim (clean) → free-frame
- maintain list of modified pages
- whenever paging device idle
- write modified page to secondary storage
- increase no. of clean page
- check free frames
- page fault
- remember which page was in each free frame

Mass Storage Structure



File System

- File: a named collection of related info recorded on secondary storage

File Attributes
- Name
  - human readable
- Identifier
  - tag
- Type
- Location
  - pointer
- Size
- Protection
- Time, date and user identification

File Operations
- create
- open
- read
- write
- reposition
- delete
- truncate

Access Lists and Groups
- Mode of access
  - read
  - write
  - execute
- classes of users
  - owner: RWX = 111 = 7
  - group: RWX = 110 = 6
  - public access: RWX = 001 = 1

File Allocation
- Contiguous Allocation
  - each file occupies set of contiguous blocks
  - adv
    - best performance
    - simple, only need
      - starting location
      - length
  - dis
    - fragmentation, needs compaction
- Linked Allocation
  - each file is a linked list of blocks
  - adv
    - efficiency ↑ by clustering
    - no external fragmentation
  - dis
    - internal fragmentation
    - locating block needs ↑ I/O + disk seeks
    - reliability ↓
      - if one middle block X, can't find remaining
- Indexed Allocation
  - each file has own index block of pointer → data blocks
  - adv
    - no external fragmentation
    - random access
  - dis
    - can't grouping
      - id. of block
      - no. of blocks
- File Allocation Table FAT
  - beginning of volume has a table, indexed by block no.
  - adv
    - simple access
    - faster than linked list (cacheable)

Directory
- single-level directory
  - adv
    - easy to implement
  - dis
    - must have unique names
    - efficiency ↓
    - grouping
- two-level directory
  - separate directory for each user
  - adv
    - diff user same file name ok
  - dis
    - isolates user
- Tree-structured directory
  - arbitrary height

* I-node (Indexed allocation)
- block types
  - direct blocks
    - directly point to data blocks
  - single-indirect
    - point to another block
  - double-indirect
    - point to single-indirect

load balancing
- spread out workload across network

computation speedup
- run concurrently

process migration

computation migration
- transfer computation only

data migration

hardware/software preference
- use remote specialized hard/software

data access
- run process remotely (don't transfer data)

Distributed - OS

user don't aware of multiplicity of machine

Types

Network OS
- access to resource done by remote access

Distributed System
- Collection of loosely coupled processors interconnected by a communication network

fast
cheap
nodes
- PC/1/2 mainframes

- cover small geographical area local-area network LAN

slower
long-haul lines
nodes: mainframes

- link geographically separated sites Wide-area network WAN

communication cost

network structure

criteria

installation cost

reliability

Topology

Bus Topology

broadcast to all, addressed receive

bidirectional bus

need regulation

avoid starvation
- small block frames

avoid collision

Star Topology
- all station directly linked to central node

central node as. frame switch

broadcast/ hub

1 station transmit at a time

Ring Topology
- repeaters links in closed loop

circular, unidirectional

remove frame when back to source.

S/P
=
efficiency

S = T1/Tp
speedup

Performance Metrics

speed
=
W/Tp

Tp
- execution time using p processors

Workload W
- no. of ops required to complete program

Motivation

communication

process into in distributed database

resource sharing

use remote specialized hardware

share and print files at remote sites

reliability
- transfer data if one site fails

speedup computation
- load sharing

hard to scale

easy to scale

tightly coupled vs loosely coupled

easy to program

hard to program

shared memory

distributed memory

Designing Issues
- transparency
- fault tolerance
- scalability

clusters
- collections of machines act as a single system

Communication Protocol
1) physical layer (bit stream)
2) data-link layer (frames/packets)
3) network layer (addresses/routing info)
4) transport layer (messages transfer between clients/generate physical address)
5) session layer
6) presentation layer
7) application layer (user interaction)

messages → (broke down) packets → (fix into) slots → transfer → target

reassemble → messages
↑ transfer through different path

fixed length packets
↑
- divide message
packet switching
- less setup time
- more overhead

Distributed System 2

Communication structure

Connection Strategies
- how 2 processes send sequence of messages

message switching
- temporary link

circuit switching
- permanent physical link
→ less overhead
→ need setup time

dynamic routing
- choose path when sending message
→ out-of-order delivery
- solution: append sequence to message
→ choose least used link

Routing Strategies
- how to send messages

virtual circuit
- fixed path in the same session
→ adaptable
→ in order delivery

fixed routing
- specify path in advance
→ minimum cost
→ in order delivery

Contention
- resolve conflicting demands
- a no. of fixed-length slots circulate in system (usually in ring)
Message slots

- register a CD and stop if 2↑ sites transmit at the same time
CSMA/CD carrier sense with multiple access / collision detection
* used in ethernet

Token passing
- circulates a token (usually in ring)
1) take token → transmit
2) complete → transmit token

Naming and name resolution
- how do processes locate each other
→ name systems in network
→ address messages with process id
→ domain-name system DNS for host-name resolution

Distributed File System

- distributed implementation of file system, it manages set of dispersed storage devices

Naming

- mapping between logical and physical objects

- single global name structure
  - total integration
- schemes
  - attach remote directory to local directory
  - host name + local name
    - x independent
    - x transparent

* multilevel mapping
- hide details of file (where and how)

first level: file name

second level: inode

Structure

- service: software providing function
- client interface
  - set of file operations
  - transparent (local files = remote files)
- client: process that invoke service
- server: services run on server

Transparency

- hide location of file

Independence
- file name don't change when change file location

vs

> stronger requirement

Andrew

- clients/servers structured in clusters by LAN
  - ↓ connected by router
  - 1 server + several workstations
- client: local namespace (local disk) + shared namespace (server)
- transparent location
  - locations info kept on each server
- file transfer between servers does not invalidate cached contents

remote file operation: whole file caching

write-back on close file

Cache

- location
  - main memory
    - ↓ quick
    - enable diskless workstation
  - Disk
    - -reliable
- update policy
  - write-back
    - write back when close file
    - or
    - scan cache regularly
  - write-through
- consistency
  - client-initiated
    - client check request
    - server check local ↔ master
  - server-initiated
    - server records files on client
    - broadcast when inconsistent
- adv
  - less contact to server
  - ↓ server load/traffic
  - better for access with infrequent write

Stateful File Service

- store file state in memory of server
- mechanism
  - 1) client open file
  - 2) server fetch into memory
  - 3) server → identifier client
  - 4) end, server reclaim memory
- dis
  - loses all volatile state in a crash
- penalties
  - ↑ request process time
  - ↑ request message
- additional constraint on DFS design

vs

Stateless File Service

- each request is new to server