CS2106 CheatSheet

by Zachary Chua

Motivation for OSes

Abstraction

- Large variation in hardware configuration, but same functionality
- Hides low-level details, presents common high-level functionality to users
- Provides efficiency and portability

Resource Allocator

- Manages all resources eg. CPU, Memory, IO
- Ensure fair and efficient use of resources

Control Program

- Controls execution of programs
- Prevents errors and improper use of computer (accidentally or maliciously)
- Ensures separate user space if multiple users sharing
- Provides security and protection

Operating System Structures

Considerations: organisation, flexibility, robustness, maintainability

Kernel Mode: Have complete access to all hardware resources

- OS

User Mode: Limited (or controlled) access to hardware resources

- Other user software

Monolithic OS

Kernel is ONE big program

- services and components are within the OS

Advantages: Well understood, Good performance

Disadvantages: Highly coupled components, Very complicated internal structure

Microkernel OS

Kernel is small and clean, provides only essential and basic features like

- IPC, address space management, thread management, etc.

Higher Level services run on top of basic facilities

- runs as server process outside OS
- uses IPC to communicate

Advantages: Kernel more robust and extendible, better isolation and protection between kernel and high level services

Disadvantages: Lower performance

Process Abstraction

Process: abstraction over program information, allows multiple programs to share hardware

OS representation of process includes **all information required** to describe running program

1. Memory Context - code, data, stack, heap
2. Hardware Context - registers, PC...
3. OS Context - Process ID, Resources used...

Stack Memory: Used for function invocation

Solves control flow and data storage issues, grows upwards

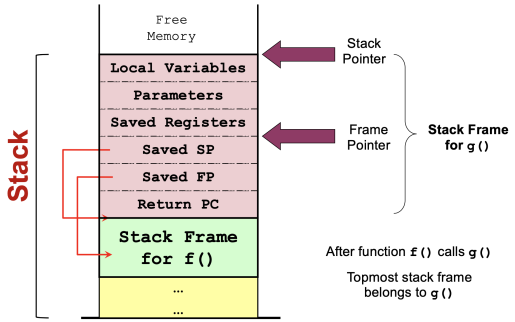
Consists of:

- return address of callee
- parameters of function
- storage of local variables
- Frame Pointer: points to fixed location in stack frame (platform dependent)
- saved registers

Stack Pointer: Points to top of stack (first unused location)

Saved Registers: limited registers in CPU, when exhausted

- use memory to temporarily hold the value
- Register can then be used by the funtion and restore after
- **register spilling**



Call Convention: No universal way - hardware and PL dependent

Setup:

Caller: Pass parameters with registers / stack

Caller: Save Return PC on stack

Callee: Save registers used by callee, old FP, SP

Callee: Allocate space for local variables of callee on stack

Callee: Adjust SP to point to new stack top

Teardown:

Callee: Place return result on stack (is applicable)

Callee: Restore saved registers, FP, SP

Caller: Utilise return result if applicable

Caller: Continues execution

Heap Memory: For dynamically allocated memory → acquire memory at runtime

Cannot use existing data / stack memory

- size not known during compilation time (cannot Data)
- no definite deallocation timing, accessible from anywhere (cannot Stack)

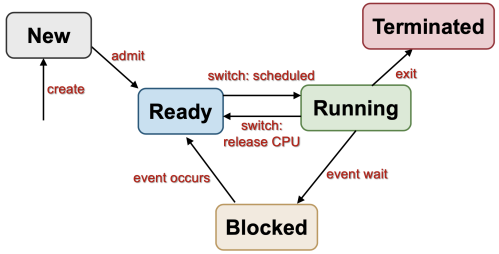
Harder to manage: variable size, (de)allocation timing, can result in "holes"

OS Context

PID - process ID, a number, unique among processes

Process State - Indication of the execution status

eg Generic 5-state process model



New - just created, may be under initialisation (not ready to run)

Blocked - waiting (sleeping) for event, cannot execute until event available

Terminated - Process has finished execution, may require OS cleanup

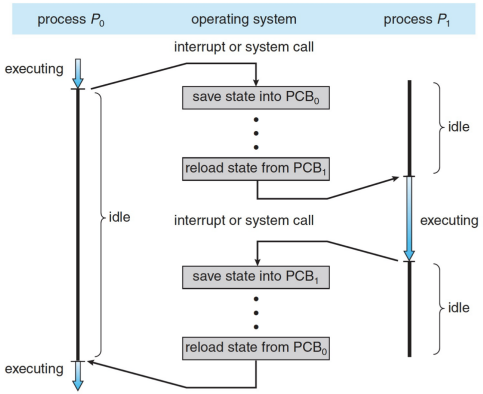
Process Control Block and Table

Process Control Block: execution context of process (aka Process Table Entry)

Kernel maintains PCB for all processes

- in one table conceptually aka Process Table

Context Switch

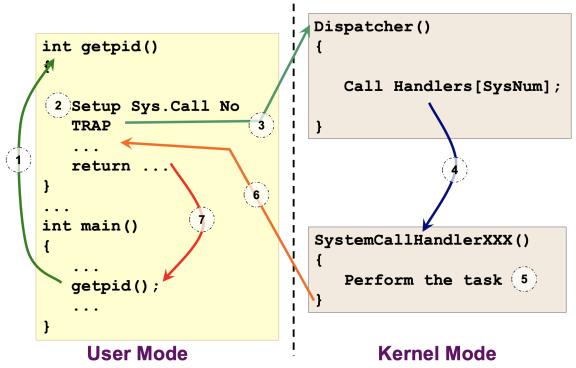


System Calls: API to OS, way to call services in kernel

NOT a normal function call, need to change from user to kernel mode

Call Mechanism

1. User Program invokes library call
2. Lib Call places **system call number** in a designated location
   - eg. register (usually in assembly)
3. Lib call does special instruction, switch from user to kernel mode
   - known as **TRAP**
4. In Kernel Mode, appropriate system call handler is determined
   - Using the syscall number as index, handled by **dispatcher**
5. Syscall handler is executed
6. Handler finishes and returns control to lib call, switch back to user mode
7. Lib call return to the user program, via normal function return mechanism



Exceptions: Happens when executing **machine level instruction**

eg. Overflow, underflow, divide by 0, illegal or misaligned mem access

Exception is **synchronous**

- occurs due to program execution

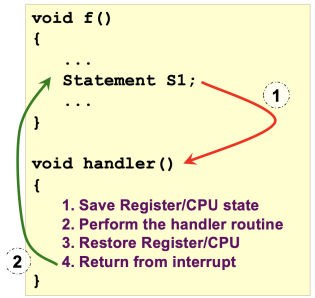
Effect: Have to execute **Exception Handler**, like forced function call

Interrupt: External events can interrupt execution of program

Usually hardware related: timer, keyboard, mouse

Interrupt is asynchronous, due to events independent of program execution

Effect: Program execution suspended, execute **interrupt handler**



1. Exception/Interrupt occurs:
   - Control transfer to a handler routine **automatically**
2. Return from handler routine:
   - Program execution resume
   - **May** behave as if nothing happened

Process Abstraction in Unix

same as above just add **zombie** process state

Process Creation — fork

syntax: `int fork()`

Returns: **PID** of new process if **parent** 0 if **child**

Behaviour: Creates child process that is a **duplicate** of current executable
- same code, same address space
- Data in child is a **COPY** of the parent (not shared)

- **Differs in**: Process Id (PID), parent PID (PPID), `fork()` return value
- Both parent and child continue executing after `fork()`
- Use return value of `fork()` to distinguish between parent and child

Implementation Optimisation: memory copy is very expensive
- Copy on Write: If child only reads, or overwritten with `execl`, don't need to copy
- duplicate a memory location when it is written to, o.w share memory

Execute new program — execl

To replace current executing process image with new one

Syntax: `int execl(const char *path, const char * arg0, ..., const char *argN, NULL)`
- path: Location of executable
- arg0 - argN: command line arguments
- NULl: indicate end of argument list

eg. `execl(''/bin/ls'', ''ls'', ''-l'', NULL)`

Note: **Does NOT** return to program after completion of new image!

Process Termination in Unix — exit

Ends execution of process

Syntax: `void exit(int status)`
- status: returned to parent process, Unix convention: 0 okay, !0 not okay

The function **does not return**

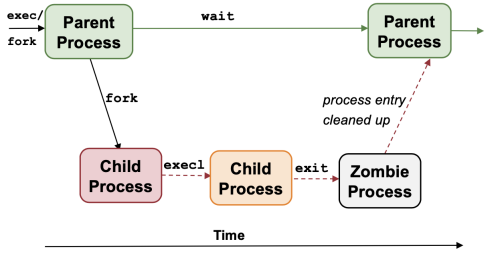
Behaviour:
- **Most** system resources used by process are released on exit, like FDs
- Some resources not releasable
  - PID and status (for parent-children sync), PCB **may** still be needed

Parent-Child Sync: Parent process can wait for child process to terminate

Syntax: `int wait(int *status)`

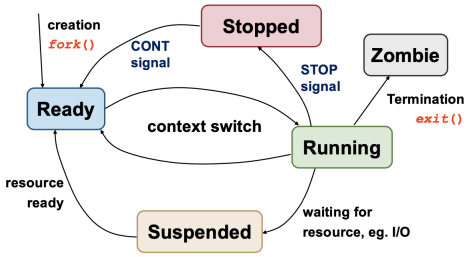
Returns: PID of terminated child
- status (address): stores the exit status of terminated child process
- can use NULL if not needed

Behaviour:
- Blocking: blocks until **at least one** child terminates
- Cleans up remainder of child system resources - kills zombie processes
- Useful variants:
  - waitpid(): waits for specific child process
  - waitid(): waits for any child process to change status



Zombie Processes

1. Parent terminates before child:
   - `init` becomes pseudo parent of child
   - child termination sends signal to `init`, which calls `wait` to cleanup
2. Child process terminates before parent but parent did not call wait()
   - Child process becomes zombie, can fill up process table



Process Scheduling

Concurrent execution: Both make progress within a short window of time

Processing Environment

1. Batch Processing
   - long-running without user intervention → **no need** to be responsive
2. Interactive — has active users interacting with it
   - **responsive, consistent response time**
3. Real time — have a strict deadline to meet

Criteria for all scheduling algos

Fairness: no starvation, **per user** or **per process**

Balanced Utilisation of System Resources
- all components of the system should be utilised without bottlenecks

Preemptive vs Non-Preemptive

Non-preemptive: Process stays running until blocks or yields CPU

Preemptive: CPU can be taken from running process at any time
- process usually given a time quantum to run
  - at end of time quantum process suspended (ready state)
  - process can block / yield CPU early

Batch Processing Algos: no user interaction
- non-preemptive scheduler predominant (no need to be responsive)
- Criteria:
  - Throughput: rate of task completion
  - Turnaround time: finish time - start time (time when process arrives)
    - related to waiting time: time spend waiting for CPU
  - CPU Utilisation: % of time when CPU is working

1. **First Come First Served (FCFS)**: FIFO queue (based on arrival time)
- blocked tasks are placed at back of queue
- **no starvation**: no. of tasks in front are always decreasing

Shortcomings:
- Convoy effect: short tasks stuck behind long task
- System resources sit idle

2. **Shortest Job First**: Select task that needs the shortest amount of CPU time
- Need to know total CPU time for a task in advance
- minimises **average waiting time**
- **Starvation possible**

Guessing CPU time: guess future cpu time from previous cpu bound phases
- exponential average: $Predicted_{n+1} = \alpha Actual_n + (1-\alpha) Predicted_n$

3. **Shortest Remaining Time**: **preemptive** version of SJF
- New jobs with shorter remaining time **can preempt** currently running job
- good service for short job even if arrive late

Interactive environment

Criteria
- Response time: time between request and response
- predictability, less variation in response time better

Timer
- Interval Timer Interrupt (ITI): OS scheduler is triggered every timer interrupt
- Time Quantum: Execution duration given to process, constant / variable, multiple of ITI

1. **Round Robin (RR)**: preemptive version of FCFS
- Response time guaranteed, Timer needed
- Choice of TQ:
  big: CPU utilisation better, but longer wait time
  small: more overhead → worse CPU utilisation, shorter waiting time

2. **Priority Scheduling**: Tasks have priority, select tasks with ↑ priority
- Preemptive: higher priority can preempt lower priority running process
- Nonpreemptive: wait for next round of scheduling
- can **starve lower priority**
  - can decrease priority of running process after every round
  - give each process minimum TQ

3. **Multi-Level Feedback Queue**: Adaptive — Learns process behaviour
- Tries to minimise response time for interactive and IO bound process and
- Turnaround time for CPU-bound processes

Rules
- if priority(A) ¿ priority(B), then B, if equal, RR
- New job: (highest priority)
- job fully utilises its TQ: priority reduced
- job blocks / yields CPU before TQ: priority retained

4. **Lottery Scheduling**: Scheduling done in rounds, with lottery tickets in each round:
   - give out lottery tickets for resources
   - When scheduling decision, choose random ticket w/o replacement
- In every round, process w X% of tickets gets to use resource X% of time
- Responsive: Every participating process gets to run every round, new processes can participate in next round
- Good control: important process given more tickets

Inter-Process Communication

Race Condition

If multiple accesses to a shared variable while modifying

Shared Memory: Communication through read/write to shared variables

1. P_1 creates shared memory region M
2. P_2 attaches M to its own memory space

3. P_1 and P_2 can communicate through M

Advantages:

- Efficient: OS only needed in setup, Easy to use: just like normal memory

Disadvantages:

- Limited to one machine, Requires synchronisation because of data races

Message Passing: Explicit communication through exchange of messages

- P_1 prepares message M and send to P_2
- P_2 receives message M
- sending and receiving provided as syscalls

Naming Scheme: how to identify other party (parties)

Synchronisation: blocking/ non-blocking

Naming Scheme: **Direct Communication**: Explicitly name other party

- One link per pair of communicating processes
- Need to know the identity of the other party

Naming Scheme: **Indirect Communication**: Messages sent to mailbox

- one mailbox shared among many processes

Blocking (**Synchronous**):

`send()`: sender is blocked until message received

- does not require buffer, sender can keep until receiver calls `receive`

`receive()`: blocked until message arrived

Non-blocking (**Asynchronous**):

`send()`: sender resumes immediately

- never blocked unless buffer full (can throw error also)
- finite buffer means not truly async

`receive()`: message hasn't arrived then just continue

Message Buffer

- OS control: no synchronisation necessary
- decouples sender and receiver, no need to unnecessarily wait

Advantages:

- applicable beyond one machine
- Portable: parties on different platforms can communicate
- Easier synchronisation

Disadvantages:

- Inefficient: requires OS intervention on every send and receive
- Harder to use: must use specific data format

Pipes

Process has 3 communication channels: `stdin, stdout, stderr`

Shell uses "—" symbol to link input output channels from one process to another

General Idea: Communication channel has 2 ends — read and write

- writers wait when buffer is full
- readers wait when buffer is empty

Syntax: `int pipe(int fd[])`

Returns: 0 for success, !0 for failure

fd[0] == reading end, fd[1] == writing end

Signal: Asynchronous notification

Recipient must handle by: default handlers or User supplied handlers

Threads - lightweight process

cheaper alternative to processes (fork is expensive)

- duplicate memory space, most of process context
- context switch requires saving/restoring process info

Communication difficult and inefficient (independent memory space)

- requires IPC

Multiple threads = multiple parts of progran executing concurrently

Threads

Threads share:

- **Memory context:** Text, Data, Heap **NOT stack**
- **OS Context:** PID, other resources like files

Unique info: thread ID, stack, registers (hw context)

Thread Switch vs Context Switch

Context switch involves: Memory, OS, Hardware context

Thread switch involves: Hardware Context, Stack (change FP and SP)

Benefits

- Economy: require much less resources to manage compared to processes
- Resource Sharing: share most resources of process, no need for IPC
- Responsiveness: can appear much more responsive
- Scalability: take advantage of multiple cores/CPU

Problems

- Synchronisation around shared memory harder
- System Call concurrency: OS must guarantee correctness and determine correct behaviour
- Process Behaviour: `fork()` duplicates threads? `exec()` changes all threads?

Thread Models

1. **User threads**: implemented as user lib

- runtime system handles thread-related operations
- Kernel is **not aware** of the threads within the process
  - thread level scheduling not possible

Advantages

- Can have multithreaded program on **any OS**
- Thread operations are just library calls
- More flexible and configurable (user-customised thread scheduling)

Disadvantages

- OS not aware of threads, scheduling performed at process level
  - One thread blocked → process blocked → all threads blocked
  - Cannot exploit multiple CPUs

2. **Kernel Threads**: implemented in OS, thread operations are syscalls

- Thread-level scheduling is possible, kernel schedules threads, not processes

Advantages: Kernel can schedule on thread levels

Disadvantages

- Thread operations are now system calls
  - slower and resource intensive
- less flexible
  - Used by all multithreaded programs
  - implemented with many features → overkill for simple program
  - implemented with few features → not flexible enough for some programs

3. **Hybrid Thread Model**: Have both Kernel and User threads

- OS schedules **kernel threads**
- User threads can bind to kernel threads

Synchronisation

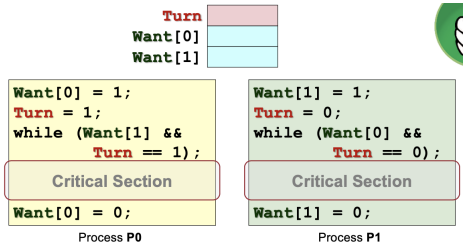
Properties of Correct CS Implementation

1. Mutual Exclusion: at most 1 process in Critical Section
2. Progress: if no process in CS, one of the waiting processes can enter
3. Bounded Wait: Upper bound on no. of times others can enter before P_i
4. Independence: Process **not** executing in CS should never block others

Symptoms of Incorrect Synchronisation

1. Incorrect output/behaviour: due to lack of mutual exclusion
2. Deadlock: all processes blocked: no progress
3. Livelock: processes keep changing state (avoid deadlock), but no progress
4. Starvation: some processes blocked forever

HLL Implementation: Peterson's Algorithm



Disadvantages:

1. Busy Waiting: keep testing while-loop (should block)
2. Low Level and Not general: general synchronisation mechanism desirable
   - not just mutual exclusion

Assembly Implementation: Test and Set: Used to implement a **lock**

syntax: `TestAndSet Register, MemoryLocation`

Behaviour (**SINGLE ATOMIC machine operation**):

1. Load current content at `MemoryLocation` into `Register`
2. Stores **1** into `MemoryLocation`

Note: Does not guarantee bounded wait out unless scheduling is fair

Usage:

```
void EnterCS(int *lock) {
while (TestAndSet(lock) == 1);
}
void ExitCS(int *lock) {
*lock = 0;
}
```

High Level Synchronisation Mechanism: Semaphore

- Only functional behaviour specified (can have different implementations)
- Provides means to
  - **block** a number of processes, known as sleeping processes (wait)
  - unblock wake one or more sleeping processes (signal)

Note: Both of these operations are **atomic**

Semaphore contains an integer value, initialised to any non-negative value

Invariant always holds: $S_{current} = S_{initial} + \#signal(S) - \#wait(S)$

General/Counting Sem: S $\geq$ 0, can be implemented with binary sempahore

Binary Semaphore: S = 0 or 1

`Wait(S)`: if $S \leq 0$: block, else: **Decrement** S

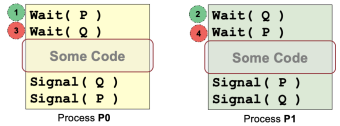
- aka `P(), Down()`

`Signal(S)`: **Increment** S, Wakes up sleeping process if any

- **never** blocks
- aka `V(), Up()`

Note: Deadlock can occur with incorrect use of semaphore

- Example: Semaphore **P = 1, Q = 1** initially



Memory Management

OS allocates, manages, protects mem space of process (from other processes)

OS provides memory-related syscalls to processes

Memory Abstraction

1. **No Memory Abstraction**

Cons: Processes cannot occupy same physical location (both start at 0)

2. **Address Relocation (No HW support)**

Cons: slow load time, hard to distinguish memory ref from normal int constant

3. **Base and Limit Register (HW support)**

Cons: every access incurs addition and comparison

Contiguous Memory Management

Assumptions: Process occupies contiguous mem region and physical mem is large enough to contain process

Fixed Partitioning: Process is given a fixed sized chunk of memory

Pros: Easy to manage, fast to allocate (no need choose)

Cons: Partition big enough for largest process, **internal fragmentation**

Dynamic Partitioning: Process given exact amount of memory required

Pros: Flexible and removes internal fragmentation

Cons: maintain more info, longer to allocate (must choose), external fragmentation

Dynamic Partitioning Allocation Algos

OS maintains a list of partitions and holes

1. **First-fit**: first hole that is large enough

2. **Best-fit**: find the smallest hole that is large enough

3. **Worst-fit**: find largest hole

When partition freed, **merge** with adjacent holes if can

Compaction: Move occupied partitions to create bigger holes, **costly**

Multiple Free Lists

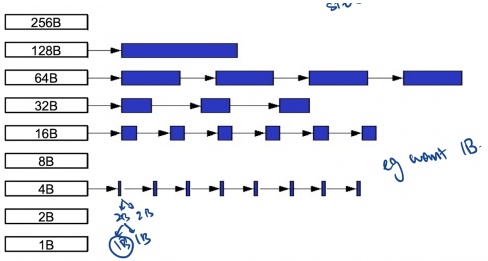
1. Separate list of free holes from list of occupied partitions

2. Keep multiple lists of diff hole sizes

- Take hole from list that most closely matches request size

- Partition size typically increases exponentially

- Faster allocation



Buddy System

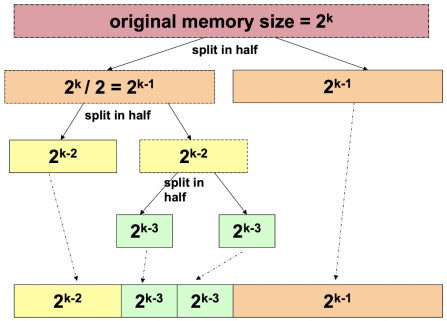
Provides efficient partition splitting, (de)allocation time and coalescing

1. Free block split into half until meet request size

   1.1 two halves form buddy blocks

2. When both buddies are free

   2.1 merge together to form larger block



Implementation

1. Keep array `A[0...k]` where 2^k is largest allocable block size
   1.1 each array element is LL keeping track of free blocks of that size
   1.2 Each free block is indicated just by starting address

May have smallest allocable block size as well

Allocate Algo: allocate block of size N

1. Find smallest S st $2^S \geq N$
2. Access `A[S]`
   2.1 if exists: remove block, allocate block
   2.2 else:
      2.2.1 Find smallest R from $S+1$ to K that has free block
      2.2.2 For ($R-1$ to S): repeatedly split till S, then go to 2

Deallocate Algo: free block B of size 2^S

1. Check `A[S]`, if buddy of B exists (free)
2. If so, remove B and buddy from list, merge to get larger B'
   2.2 goto step 1, with B' as B
3. else: insert B into `A[S]`

Using Binary: blocks B and C are buddies if

1. Lowest S bits (0 to S) are identical and Bit S ($S+1$ bit) is different

Disjoint Memory Schemes

Remove assumption that process occupies one contiguous memory block

Paging: Memory split into physical frames

Logical memory of process also split into logical pages, same size

At execution time, pages are loaded into **any avail** frame

Lookup Mechanism: Page Table

Physical Address = Frame Number × sizeof(frame) + offset

Frame Number = (Page_Table [page_no])

Offset: displacement from beginning of physical frame

Note: Keep frame size as power of 2, can use bit manipulation

Address Translation Formula: frame size 2^n, m bit logical address

1. Split LA into $p = (m-n)$ MSB of LA and o = remaining n bits
2. Use p to find frame number f (from Page Table)
3. $PA = f * 2^n + o$

Properties

1. No external fragmentation as every single frame can be used
2. Insignificant internal fragmentation - max 1 page per process not utilised
3. Clean separation of LA and PA

Implementation

Software solution: OS stores PT info in PCB (part of mem context), pointer cos PT big

Issues: 2 mem accesses per mem ref. 1st for PT, 2nd for memory item

Hardware Support: Translation Look-Aside Buffer (TLB)

- cache for PT entries, very small (10s of entries) fast ($\leq$ 1 clock)

Avg Access Time w TLB: TLB 1ns, Memory 10ns, avg hit rate 90%

90%(1ns + 50ns) + 10%(1ns + 50ns + 50ns) = 56ns

TLB and Context Switching: TLB flushed

Flushed for correctness, security, safety

TLB is per core, must be fast so can't be centralised, part of pipeline (checked before access memory)

Note: TLB is not part of hw context

Paging Protection

1. Access-Right Bit: Each PTE has (read, write execute) bits
2. Valid bits: prevents mem access to unused pages

Every mem access checked against these bits in hw

Page Sharing: Same frame number in PTEs of diff processes

1. Share code page, eg. library code, syscalls
2. Implement **Copy-on-Write**

Segmentation

Reasons:

1. Split memory space according to logical memory regions
   1.1 diff regions have diff usages, permissions, lifetimes, etc
2. Some regions may grow / shrink at execution time

Each memory segment has a name, limit

Memory References: Segment name + offset

Logical Address Translation

Each segment maps to contiguous memory region with base addr and limit

Logical Address <SegId, Offset>

1. SegId used to look up <Base, Limit> of segment in **segment table**
2. Physical Address $PA = Base + Offset$

Note: Offset < Limit for valid access

Pros

1. Each segment is indep contiguous memory space
   1.1 More efficient bookkeeping
   1.2 Segments can grow / shrink and be protected / shared independently

Cons

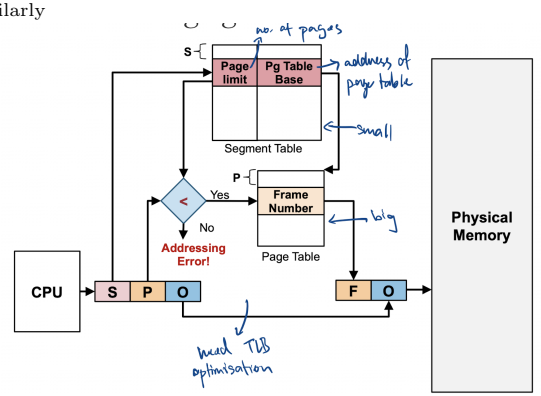
1. Requires variable-size contiguous memory region
   1.1 external fragmentation

Segmentation with Paging

Segment composed of several pages instead of contiguous mem region

- Each segment has a PT

Segment can allocate new page and add to its PT to grow, and shrinks similarly



Memory ref: SegId, Frame no, offset OR SegId, offset

- calculate frame number from offset in the second case

Virtual Memory Management

Remove second assumption that process can fit in memory

Idea: Split logical addr space into chunks, some in memory, others in secondary storage

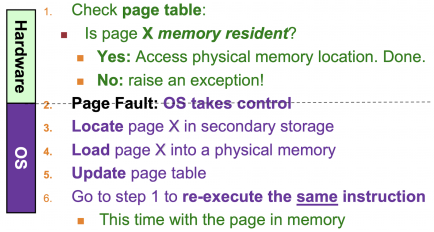
Extension of paging: some pages in memory, some in secondary storage

Extended Paging Scheme

Still uses PT for Logical to Physical address translation

Page Types: Mem resident vs non-mem resident (need new resident bit)

Page Fault: CPU tries to access non-mem resident, OS need to bring into memory



May have to kick resident page, process that page fault in blocked state

Cons: Secondary storage slow, ms compared to ns

- Thrashing: if page fault most of the time
- amortized by locality principles

Pros:

1. More efficient use of memory (unused pages kept on secondary storage)
2. More processes can reside in memory (improve CPU utilisation)

Demand Paging

Processes start with no memory resident pages. Allocate when page fault

Pros: fast startup time, small memory footprint

Cons: Processes slow at start due to page faults, may cause thrashing

Page Table Structure

Large page tables → high overhead, page table larger than a page

Page Tables must be contiguous in memory (even if larger than a page), to allow for efficient retrieval of entries

2-level Paging

Processes may not use entire virtual memory space, having full PT is a waste

1. Split PT into smaller PTs with PT number
2. Only a few regions used: new regions can be allocated if required
3. Need a "directory" to keep track of regions

- If original PT has 2^p entries:
  - with 2^m smaller PTs, m bits needed to identify PT
  - each smaller PT contains $2^{(p-m)}$ entries
  - **page directory** contains 2^m entries, null if empty

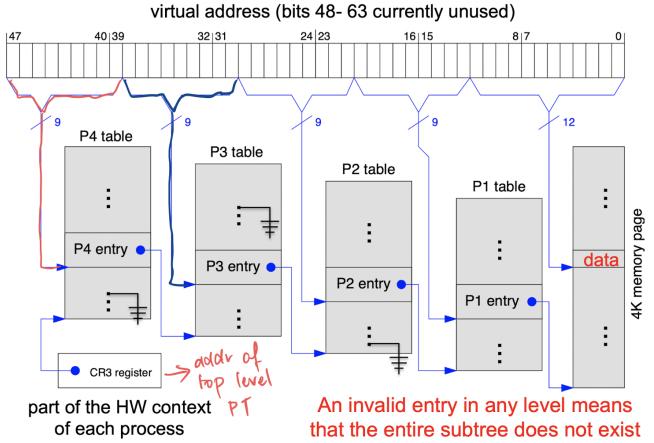
Pros:

1. PT structures can be larger than frame (does not need to be contiguous)
2. Can have empty entries in page directory (save space)

Cons:

1. 2 serialised memory accesses to get frame number (longer PT walk)

Solution: MMU cache (TLB for directory entries)



Inverted Page Table: usually used as auxiliary structure

Page Table is per-process: m processes, m indep PT

But, only N physical memory frames, out of M PT, only N entries valid

Idea: Keep a single mapping of physical frame to (pid, page#)

Note: page# is not unique

Lookup: search whole tree

Pros: Big space savings

Cons: Slow translation

Page Replacement Algos

Memory Access Time: $T_{access} = (1-p) * T_{mem} + p * T_{page\_fault}$

Note: $T_{page\_fault} >> T_{mem}$ as involves access to disk

OPT: Replace page that will not be needed again for longest period of time

Guarantees min number of page faults, but not possible

FIFO: Pages evicted based on loading time (oldest page)

Implementation

1. OS maintains queue of resident page num.
2. Remove first pae in queue if needed
3. Update queue during page fault trap

Suffers from Belady's Anomaly: ↑ frames, ↑ page faults

- does not exploit temporal locality

LRU: make use of temporal locality

Note: Good result in practice, does not suffer from Belady's Anomaly

Implementation: not easy, requires substantial HW support

1. Use a counter
   1.1 time counter, which is updated every mem ref
   1.2 each PTE has "time-of-use" field, storing time counter
   1.3 Replace page with smallest "time-of-use"

Problems: Need to search through all pages, time always ↑ (overflow)

2. Stack
   2.1 When referenced, remove and place on top of stack
   2.2 Replace page at the bottom of the stack

Problems: Not "stack" (can be removed from anywhere), hard to do in HW

Second Chance Page Replacement aka CLOCK

Modified FIFO, give 2nd chance to pages that are accessed, uses circular LL

Each PTE has reference bit: 1 = accessed since last reset, 0 ow

Algo:

1. oldest FIFO page is selected.
2. If ref bit = 0 → page is replaced. DONE
3. Else: Page is given 2nd chance
   3.1 ref bit cleared to 0
   3.2 Effectively resets arrival time (as if page is newly loaded)
   3.3 Next FIFO page is selected, go to Step 2

When all pages have ref bit 1, degenerates to FIFO

Note: ref bit set to 1 when page is accessed

Frame Allocation: how to allocate frames among processes

Equal Allocation: each process get N/M frames

Proportional Allocation: Each process gets $size_p/size_{total} * N$ frames

Page Replacement

Local Replacement: Victim page selected among pages of the process

Pros: no. of frames constant, stable performance

Cons: if not enough, can hinder progress of process

Global Replacement: Victim page can be chosen among all physical frames

Pros: Allow self-adjustment between processes (need more get more)

Cons: Bad behaved processes can affect others, unstable performance

How to determine right no of frames

If insufficient frames → thrashing. If global replacement → cascading thrashing If local replacement → thrashing limited to one process, but can use up IO bandwidth

Working Set: set of pages that process references in a time period

Set of pages referenced by process is relatively constant in a period of time

When working set is stable and well-defined

- Page faults rare

When transitioning to a new working set

- Many page faults for the new set of pages

Implementation:

1. Define Working Set Window Δ (interval of time)
2. $W(t, \Delta)$ = active pages in the interval at time t
3. Allocate enough frames for working set to reduce possibility of page fault

Δ too big, contain pages from diff working set, too small miss out pages

File Systems

File System

Provides

1. Abstraction (direct access not portable, depends on HW spec and organisation)
2. High level resource management scheme
3. Protection between processes and users
4. Sharing between processes and users

Criteria:

1. Self-contained: info on media is enough to describe entire organisation
2. Persistent: beyond lifetime of OS and processes
3. Efficient: good management of free, used space, min bookkeeping overhead

	Memory Management	File System Management
Underlying Storage	RAM	Disk
Access Speed	Constant	Variable disk I/O time
Unit of Addressing	Physical memory address	Disk sector
Usage	Address space for process **Implicit** when process runs	Non-volatile data **Explicit** access
Organization	**Paging/Segmentation:** determined by HW & OS	Many different FS: ext* (Linux), FAT* (Windows), HFS* (Mac OS)etc.

File System Abstraction: Colection of files and directories

Provides an abstraction for accessing and using the above
File: Logical unit of information created by process
An ADT that contains:
1. Data: information
2. Metadata: aka file attributes eg. Name, ID, type, size, protections, etc
File Type: regular files, directories, special files
- has associated set of operations, possibly a specific program
- 2 Major types: ASCII / Binary (executable)
- Distinguished by:
  - extension in windows
  - embedded file info in Unix, (magic number stored at start of file)

File Protection:
- Type of Accesses: read, write, execute, append, delete, list (read metadata)
- Most general scheme
  Access Control List - list of user identity and allowed operation
  Pros: customisable Cons: Additional info associated with file

Operations on File Metadata: Rename, change attributes (date, access permission), read attribute
File Data: Structure
- Array of bytes: UNIX view, each byte has unique offset from start
- Fixed Length Records: array of records, can grow/shrink, jump to record
  - Offset of Nth record = size of record * (N - 1)
- Variable Length Record: Flexible but hard to locate record

File Data: Access methods
- Sequential Access: read in order from start. Cannot skip but can rewind
- Random Access: Data read in any order
  - `read(offset)`: every read explicitly state the position to be accessed
  - `seek(offset)`: special operation provided to move to new location in file used in UNIX and windows
- Direct Access: Used for file containing fixed-length records
  - Allows random access to any record directly
  - random access where 1B = record

File Data Operations
- Create, Open, Read, Write, Reposition / seek, Truncate

File Operations as Syscalls
OS provides protection, concurrent and efficient access.
OS maintains info for opened file:
1. File pointer: location in file
2. Disk Location: file location on disk
3. Open Count: how many process has this file opened

Implementation:
1. System-wide Open-File Table: 1 entry per unique file
2. Per-process open-file table (aka FD table): 1 entry per file used in process
   - each entry points to System-wide table
   - Enables file sharing: entries in diff FD tables point to same entry
     - eg. after fork
   - entries in diff FD tables could point to diff entries
     - but those entries point to same file on disk

Directory: provide logical grouping of files, keep track of files
1. Single Level: all in one root dir
2. Tree: Allows sub-directories
3. DAG: file appears in multiple directories using links
4. Graph: link to directory
   - undesirable: need to prevent infinite loop
   - hard to determine when to remove file/dir

Hard Link: A and B has separate pointers to same file on disk
Pros: Low overhead, only pointers added in directory
Cons: Deletion problems
UNIX command: `ln`
Symbolic Link: G contains path name of F
When G is accessed: find out where F is, then access F
Pros: Simple deletion, file deleted → link remains but not working
Cons: Larger overhead - link file takes up actual disk space
Note: can be linked to directory → allows graph structure to be created
Unix Command: `ln -s`

File System Implementations
Disk Structure: 1D array of Logical blocks
Logical Blocks: smallest accessible unit (512B - 4KiB)
mapped to disk sector, layout of disk sector is HW dependent

Disk organisation
Master Boot Record at sector 0 with partition table. Followed by ≥ 1 partitions, each with indep FS
FS contains:
1. Os boot information
2. Partition details: number of blocks, number and location of free disk blocks
3. Directory Structure: root dir, subdirs are files
4. Files info and actual file data

File Implementation: collection of logical blocks
Good file implementation must keep track of blocks, allow efficient access, ensure disk space utilised effectively (minimal overhead)
1. **Contiguous**: allocate contiguous disk blocks to file
Good for Write-Once Read-Many
Pros: Simple to keep track (starting block num and length), fast access
Cons: External fragmentation, file size needs to be specified in advance
2. **LL**: Each disk block stores data and next block number
File info stores first and last block number
Pros: solves fragmentation
Cons: Random access slow, 20-30 bits reserved for ptr, less reliable (wrong ptr)
3. **LL 2.0 (FAT)**: Move all block ptrs into single table kept in memory
Pros: Faster Random Access (LL traversal done in memory)
Cons: FAT can be huge, wastes memory space
4. **Indexed Allocation**: Each file has Index Block
Index Block: array of disk block addresses, Index[N] is N + 1 block address
Pros:
1. Less memory overhead — only index block of opened files in memory
2. Fast direct access

Cons:
1. Limited max file size: Max number of blocks == No. of index block entries
2. Index block overhead

Variations: LL, Multilevel Indexing (allow for larger size), or combination

Free Space Management — Bitmap
Each diskblock represented by bit, free: 1, occuped: 0
Pros: Provide good set of manipulators
Cons: Needs to be kept in mem for efficiency

Free Space Management — LL
LL of disk blocks, each disk block contains:
1. number of free disk block numbers
2. Ptr to next free space disk block

Pros: Easy to locate free block, only first ptr is needed in memory
Cons: High overhead, mitigated by storing free block list in free block

Directory Structure
Main tasks: Keep track of files (with metadata), map file name to file info
Directory Structure — LinkedList
Each entry is a file, stores:
1. file name (minimally) and maybe metadata
2. file information or ptr to file information

Locating file using list
1. Requires Linear Search
   - Solution: cache latest few searches

Directory Structure — Hash Table: contains hash table of size N
Hashes file name, chaining collision resolution
Pros: Fast Lookup
Cons: HT limited size, depends on hash function
File Information: file name, metadata and disk block info
Disk block info: eg. first, last block no. (LL), INODE address (Index)
Approaches:
1. Store in directory entry — fixed size entry
2. Store only file name and point to data structure for other info

File Operation — Create: Create `/.../.../parent/F`
1. Locate parent directory using full path name
2. Search for `F`, if exists terminate with error
   2.1 Search could be done on cached directory structure
3. Use free space list to find free disk block(s) depending on allocation scheme
4. Add entry into `parent` — relevant info (name, disk block info)

File Operation — Open: Open `/.../.../F`
1. Search System Wide Table for existing entry `E`
2. If found, create entry in `P's` table to point to `E`, return ptr to this entry
3. Else, use full pathname to locate `F`
   3.1 if not found, terminate with error
4. Load `F's` info into new entry `E` in System Wide Table
5. Create entry in `P's` table to point to `E`
6. Return ptr to this entry

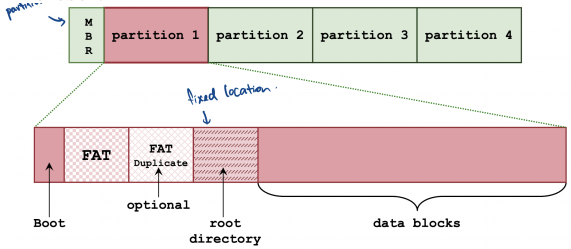
Disk I/O Scheduling
Seek — Change track (move head)
Rotation — Change Sector
3-stage read/write Process
Time taken: Seek time + Rotational Latency + Transfer time
1. Position disk head over proper track (seek time, avg 2-10ms)
   1.1 Avg Seek time = $\frac{1}{3}N$, N is time for max seek distance
2. 2. Wait for sector to rotate under disk head (Rotational Latency)
   2.1 Rotation speed: 4800-15000 rpm, 12.5ms-4ms per rotation respectively
   2.2 Avg: assume halfway ard track, 6.25ms and 2ms
3. Transfer sector(s) (Transfer Time)
   3.1 Transfer Size / Transfer Rate
   3.2 In the order of μs, $1 + 2 >> 3$

Disk Scheduling Algos
Aim to reduce seeking time
1. FCFS
2. SSF (shortest seek first)
3. SCAN family
   3.1 SCAN: Bi-Direction
   3.2 C-SCAN: 1-Direction (outer → inner)

File System Case Studies
Reasons for partitioning
1. File systems cannot support such large sizes

2. can use as swap
3. OS in 1 partition, files in another partition
4. Smaller FAT tables, Inode tables

Microsoft — FAT

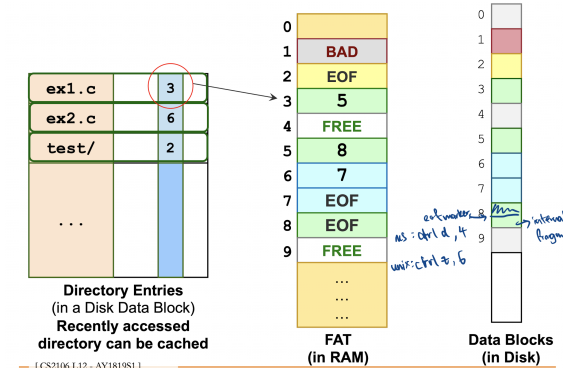


File data allocated to data blocks/ data block clusters
Allocation info is kept as a LL in a File Allocation Table

File Allocation Table:
1. 1 entry per data block/cluster
2. Store disk block information
   2.1 Free (unused), Block Num (next block), EOF (NULL), Bad (damaged)
   2.2 Damaged could be physical damage / checksum or crc doesnt tally
3. OS caches FAT in RAM to facilitate LL traversal

Directories: Special type of file
1. Root directory stored in special location (others stored in disk blocks)
2. Each file/ subdirectory within folder represented as directory entry

Directory Entry: Fixed 32 byte entry
1. Name + extension: 8 + 3, first byte may have special meaning
2. Attributes (read-only, Directory / File flag, hidden, etc)
3. Creation Date + Time
4. First disk block (12, 16, 32 bits for FAT12, FAT16, FAT32) + File size



File Operations:
1. Delete directory entry: set first letter in filename to 0xE5
2. Free data blocks: Set FAT entry to FREE
   Note: actual data blocks not touched (for undelete)

Free Space: does not keep track of free space, must be calculated by going through FAT
Variants due to:

Disk Cluster: number of contiguous disk blocks as smallest allocation unit instead of disk blocks

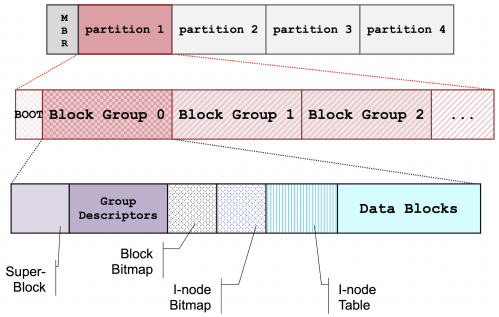
Fat Size: Larger FAT, more disk blocks/ clusters, more bits needed to represent

Largest Usable Partition: Cluster size * no of clusters (no of entries in FAT)

Pros: Larger Cluster → Larger usable partition
Cons: Larger Cluster → Larger internal fragmentation
Long File Names: use multiple dir entries, use previously unused file attribute to denote

Unix — Ext2

Disk space split into blocks (1 or more disk sectors, like FAT clusters)
Blocks are grouped into Block Groups
Each file/ directory descriped by Inode (index node) which contains:
1. File metadata (access rights, creation time, etc)
2. Data block addresses (table of contents)



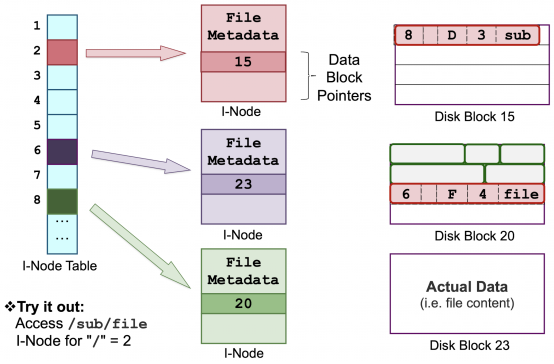
Partition Info
1. Superblock — Describes whole FS
   1.1 Total Inode number, Inodes per group
   1.2 total disk blocks, disk blocks per group
   1.3 Duplicated in each block group for redundancy
2. Group Descriptors — Describe each of the block group
   2.1 Number of free disk blocks, free I-nodes, location of bitmap
   2.2 Duplicated in each block group
3. Block Bitmap (data blocks) — Usage status of blocks in this block group (0 free)
4. I-Node Bitmap — Usage status of I-Nodes of this block group (0 free)
5. I-Node table — Array of I-Nodes, can be accessed by unique index
   5.1 Contains I-Nodes of this block group

I-Node Structure — 128 bytes
1. Mode (2) — File type (reg, dir, special) + permissions
2. Ownder info (4) — User ID (2), Group ID (2)
3. File size (4 / 8) — File size in bytes, larger for regular file (8B)
4. Timestamps (3 * 4) — Creation, Modification, Deletion
5. Data Block ptrs (15 * 4) — indices of data blocks
   5.1 12 direct, 1 indirect (disk block containing ptrs to disk blocks)
   5.1 1 double indirect, 1 triple indirect
6. Reference Count (2) — Number of times this I-Node is referenced by directory entry

Pros: Fast access to small files (1st 12 blocks direct), can handle big files

Directory Structure

LL of directory entries, which contains
1. I-Node number for that file/subdir
2. Size of this entry (for finding next one)
3. Length of name
4. Type: File / subdir / special (IO)
5. Name (up to 255 chars)



Operations
1. Delete: Remove entry from parent dir
   1.1 Point previous entry to next entry/ end, blank record if first entry
   1.2 Update I-node bitmap, Block Bitmap
2. Hard Link: Create dir entry in `B` point to same I-node, can have diff name
3. Sym Link: Create new file `Y` in `B`, `Y` contains path to `X`

FS consistency check: CHKDSK (win), fsck (UNIX)
eg. power loss / system crash may make file system inconsistent

Defragmntation:
Fragmentation: File data scattered across many disjoin blocks on storage (impacts IO)
Linux: Files allocated further apart, free blocks near to existing data block used if possible
- Fragmentation low when drive <90% full

Journaling: keep additional info to recover from system crash
1. Write info and or data into separate log file
2. Perform operation

Can recover to earlier stable state / re-perform interrupted file operation

Synchronisation Classics

Producer-Consumer

```
while (TRUE) {
        Produce Item;

        wait( notFull );
        wait( mutex );
        buffer[in] = item;
        in = (in+1) % K;
        //count++;
        signal( mutex );
        signal( notEmpty );
}
```
Producer Process

```
while (TRUE) {

        wait( notEmpty );
        wait( mutex );
        item = buffer[out];
        out = (out+1) % K;
        //count--;
        signal( mutex );
        signal( notFull );

        Consume Item;
}
```
Consumer Process

- Initial Values:
  - `in = out = 0`
  - `mutex = S(1), notFull = S(K), notEmpty = S(0)`

Message passing version

```
MessageQueue mQueue = new MessageQueue(K);
```

Producer

```
while (TRUE) {
        Produce Item;
        mQueue.send(Item);
}
```

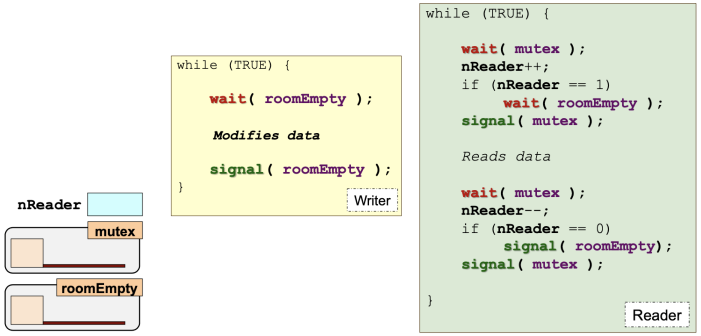
Consumer

```
while (TRUE) {
        mQueue.receive(Item)
        Consume Item;
}
```

Block consumer when buffer empty, block producer when buffer full

System handles mutual exclusion on buffer operations

Reader-Writer (Simple version)



Initial Values: roomEmpty = S(1), mutex = S(1), nReader = 0

Note: Writers can be starved

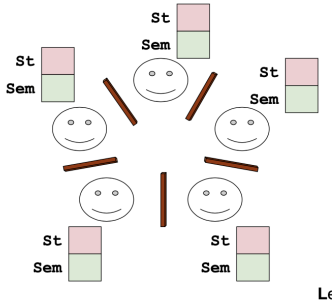
Dining Philosopher

1. Limited Seats

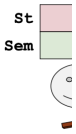
```
Semaphore seats = S(4);
//initialization

void philosopher( int i ){

    while (TRUE){
       Think( );
       wait( seats );
       takeChpStck( LEFT );
       takeChpStck( RIGHT );
       Eat( );
       putChpStck( LEFT );
       putChpStck( RIGHT );
       signal( seats );
    }
}
```

2. Tanenbaum

```
#define N 5
#define LEFT ((i+N-1)% N)
#define RIGHT ((i+1) % N)

#define THINKING 0
#define HUNGRY 1
#define EATING 2

int state[N];
Semaphore mutex = 1;
Semaphore s[N];

void philosopher( int i ){
    while (TRUE){
       Think( );
       takeChpStcks( i );
       Eat( );
       putChpStcks( i );
    }
}
```



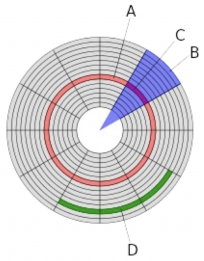

```
void takeChpStcks( i )
{
       wait( mutex );
       state[i] = HUNGRY;
       safeToEat( i );
       signal( mutex );
       wait( s[i] );
}
```




```
void safeToEat( i )
{
    if( (state[i] == HUNGRY) &&
        (state[LEFT] != EATING) &&
        (state[RIGHT] != EATING) ) {

         state[ i ] = EATING;
         signal( s[i] );
    }
}
```

```
void putChpStcks( i )
{
       wait( mutex );

       state[i] = THINKING;
       safeToEat( LEFT );
       safeToEat( RIGHT );

       signal( mutex );
}
```

Layout of Hard Disk



A = Track
B = Geometric Sector
C = Track Sector
D = Cluster

Pipes and Custom Signal Handler

```
#define READ_END 0
#define WRITE_END 1

int main()
{
   int pipeFd[2], pid, len;
   char buf[100], *str = "Hello There!";

   pipe( pipeFd );
   if ((pid = fork()) > 0) { /* parent */
     close( pipeFd[READ_END] );
     write( pipeFd[WRITE_END], str, strlen(str)+1 );
     close( pipeFd[WRITE_END] );
   } else {                  /* child */
     close( pipeFd[WRITE_END] );
     len = read( pipeFd[READ_END], buf, sizeof(buf) );
     printf("Proc %d read: %s\n", pid, buf);
     close( pipeFd[READ_END] );
    }
}
```

pipe example

pipeFd[1] pipeFd[0]

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

void myOwnHandler( int signo )
{
    if (signo == SIGSEGV){
        printf("Memory access blows up!\n");
        exit(1);
    }
}

int main(){
    int *ip = NULL;

    if (signal(SIGSEGV, myOwnHandler) == SIG_ERR)
        printf("Failed to register handler\n");

    *ip = 123;

    return 0;
}
```

User defined function to handle signal. In this example, we handle the "SIGSEGV" signal, i.e. the memory segmentation fault signal.

Register our own code to replace the default handler.

This statement will cause a segmentation fault.

General Semaphore with mutex

```
int count =   <any non-negative integer>;
Semaphore mutex = 1;   //binary semaphore
Semaphore queue = 0;    //binary semaphore, for blocking tasks
```

```
GeneralWait() {
  wait( mutex );
  count = count -1;
  if (count < 0) {
      signal(mutex);
      wait(queue)
  }  //else removed
  signal(mutex);
}
```

```
GeneralSignal() {
    wait(mutex);
    count = count + 1;
    if (count <= 0) {
       signal(queue);
    } else {    //else added
       signal(mutex);
    }
}
```