CS2106 - Introduction to Operating Systems - AY 24/25 Sem 2 (Finals)

@darkrelicz · Zaidan @zaidansani

Introduction

Motivation: Abstraction, Resource alloc, Control prog
Structures: Monolithic (kernel is one big prg), Microkernel (miminal kernel, IPC comms), Layered (onion, monolithic), Client-server
Virtual Machines: Type 1 (bare metal) / Type 2 (need host OS)

Process Abstraction - Memory Context

Text - instructions / **Data** - global & static vars / **Heap** - dynamic allocation / **Stack** - function invocations
Memory context - Text, Data, Stack & Heap /
Hardware Context - GP registers, PC, SP, FP, TLB, ...

Stack frame - return addr (enables nested calls), args for func, space for local vars, other info (e.g. *saved registers*)
Stack pointer - points to top of stack / **Frame pointer** - fixed loc in stack frame (platform dependent)
Register spilling - registers needed > GPRs, store in stack

Heap challenges - variable size, (de)alloc creates holes

Stack - Function setup & teardown

1. Prepare to make a function call
   - Caller pass params with registers and/or stack
   - Caller save return PC on stack
2. Transfer control from caller to callee
   - Callee save regs used by callee & old FP, SP
   - Callee alloc space on stack for callee's local vars
   - Callee adjust SP to new stack top
3. On returning from function call
   - Callee put result on stack top (if have)
   - Callee restore saved SP, FP & saved regs
   - **Note:** SP is moved, stack data is not deleted
4. Transfer control to caller using saved PC
   - Caller read/use return result (if have)
   - Caller continue execution

Process Abstraction - OS Context

OS Context - PIDs, process states, ...
Processes are distinguished using PIDs
5 states - new, ready, running, blocked, terminated
Transitions

- **Create** - nil → new
- Admit - new → ready
- **Switch** - ready → running / running → ready
- **Event wait** - running → blocked
- **Event occurs** - blocked → ready

Process Control Blocks (Process Table Entries)

PCB stores entire execution context for process
Concerns: Scalability (no. of concurrent processes) & Efficiency (min. space wastage)

Process Interaction with OS

Unix system calls - function wrapper / function adapter
System call mechanism

1. User prog invokes library call
2. Library call places system call number in designated location (e.g. *register*)
3. Library call executes TRAP, switching from user mode to kernel mode
4. Determine system call handler through dispatcher using system call number
5. System call handler executed
6. System call handler ended
7. Library call returns to user program

Exception - synchronous, exception handler
Interrupt - asynchronous, interruption handler

Process Abstraction in Unix

`fork()` - returns child's PID (parent proc) / 0 (child proc)

* child is duplicate of parent, differs in PID, PPID, fork()
`exec*()` - replace current executing process
* only code is replaced, PID and other info still intact
`init()` - root process, PID 1
`exit()` - returns 0 (successful execution) / !0 (problematic execution) (implicitly called after proc ends)
`wait*()` - blocks until ⩾ 1 child terminates, child system resources cleaned up, returns terminated child's PID
* wait() not called → zombie processes (on exit())
Zombie process - terminated process with valid PCB, occupies resources
Orphan process - child with terminated parent, adopted by `init` proc, occupies resources

Process Scheduling

Criteria - fairness (fair share of CPU time / no starvation) & balance (all parts of computing system is utilized)
Preemmptive - fixed time quantum
Non-preemptive - runs till finish/blocked
Measures - turnaround time (finish - start time), waiting time (TT - burst), throughput (no. of tasks completed/time unit), response time (first execution - arrival time), CPU utilization (% CPU used/time unit)

Batch Processing

Criteria - turnaround time, throughput, CPU utilization
FCFS - FIFO format, may lead to convoy effect
SJF - run task with lowest burst time, minimizes avg. waiting time (must know burst time in advance)
* exponential average - $\text{predicted}_{n+1} = \alpha \times \text{actual}_n + (1-\alpha) \times \text{predicted}_n$
SRT - preemptively selects task with shortest remaining time, new jobs can preempt current jobs

Interactive Environment

Criteria - response time (using preemptive sched algos), predictability (less variation)
Timer interrupt - interrupts periodically
Interval of Timer Interrupt - OS scheduler triggered every timer interrupt
Time quantum - execution duration, multiples of ITI
Round Robin - preemptive FCFS
* big TQ ⇒ ↑ CPU utilization, ↑ waiting time
* small TQ ⇒ ↓ CPU utilization, ↓ waiting time
Priority Scheduling - highest priority runs first
* *issue* - can starve if higher priority processes hogs CPU
* *solutions* - assign TQ / decrease priority after every run
* priority inversion - lower priority, holding resource needed by higher priority, preempted by middle priority → higher priority can't run
* solution - lower priority gets highest priority temporarily to finish execution quickly
MLFQ - adaptive, minimizes RT for I/O bound processes, TT for CPU bound processes
$p(A) > p(B) \Rightarrow A$ runs, $p(A) == p(B) \Rightarrow$ RR
New job ⇒ highest priority, TQ fully utilized ⇒ priority reduced, gives up TQ / blocked ⇒ priority retained
Lottery Scheduling - process holding $X\%$ of tickets will win $X\%$ of lottery held and use resource $X\%$ of time
responsive / good level of control / no starvation

Threads

Unique to each thread - thread ID, registers, stack
Shares - memory & OS context
Thread switching - only hardware context modified
Benefits - multiple threads require less resources (economic), resource sharing, responsive, scalable w.r.t. CPUs
Problems - parallel system call possible, impacts process operations (`fork()` and `exec()` how?)

Thread Models

User thread - user library, kernel unaware
not OS-dependent, configurable & flexible / cannot exploit multiple CPUs, 1 thread blocked → process blocked → all threads blocked

Kernel thread - system calls, kernel aware
can exploit multiple CPUs / slower & more resource intensive, less flexible

Hybrid - OS schedules kernel threads, user binds to kernel threads (↑ flexibility)

POSIX Threads - `pthread`

`pthread_create(pthread_t *tid, const pthread_attr_t *tAttr, void* (*startRoutine)(void*), void *argForStartRoutine);`
`pthread_exit(void * exitVal);` - if not used, pthread terminates when end of `startRoutine` is reached
`pthread_join(pthread_t tidToWait, void ** exitStatus);`

Inter-Process Communication

Shared memory - P1 & P2 attach to common mem region, access shared region normally
efficient, easy to use / synchronization issues, harder to implement
`shmid=shmget(IPC_PRIVATE, size, IPC_CREAT|0600)`
`shm = (int*) shmat(shmid, NULL, 0)`
`shmdt((char*) shm)` / `shmctl(shmid, IPC_RMID, 0)`

Message passing - P1 sends message to P2, P2 receives message, sending & receiving are system calls
portable, easier synchronization / inefficient (syscall), harder to use (size limitation)
Schemes - direct (`send(P1, msg), rcv(P2, msg)`), indirect (`send(mailbox, msg), rcv(mailbox, msg)`)
Synchronization - Sync (blocking) - simplifies prgmming, ensures sync, may be inefficient & may block indefinitely / Async (non-blocking) - responsive, efficient, no deadlocks, complex if ops not completed immediately, may result in busy waiting

Unix Pipes

Semantic ver - buffer full → writers (`fd[1]`) wait, buffer empty → readers (`fd[0]`) wait — (fd - file descriptors)
`pipe(int fd[])` - return array of fd, ✓ - 0, × - !0 / `close(fd[i])` / `write(fd[1], item, len+1)` / `read(fd[0], buf, sizeof_buf)` / `dup(oldfd)` & `dup2(oldfd, newfd)` - create copies of given fd

Synchronization

Race condition - outcome depends on shared resource's order of modification / Issues - deadlock, livelock, starve
Critical section - only 1 process accesses shared resource

Properties of Critical Section

1. **Mutual exclusion** - only 1 executes in CS at any time
2. **Progress** - those waiting for CS get their turn
3. **Bounded wait** - none should wait forever to reach CS
4. **Independence** - processes not in CS cannot block

Critical Section Implementation

Assembly level - `TestAndSet(&lock)` (atomic)
High level language - **Peterson's Algo** - tracks turn & `want[]`, humble algo (i want it but let you go 1st)
Cons - busy waiting, low level, not general (only mutex)
High level abstraction - **Semaphores** - `wait(sem)` (sem ⩽ 0, blocks / sem--), `signal(sem)` (sem++ / unblocks)
$\text{Sem}_{\text{current}} = \text{Sem}_{\text{init}} + \#\text{signal(Sem)} - \#\text{wait(Sem)}$ /
Monitor - sync construct that handles locking automatically, only 1 thread can execute within monitor, conditional variable - built-in `signal()` & `wait()`

Synchronization Problems

Producer-Consumer - to enforce insert only when there's empty slot, remove only when there's filled slot
Solution - `mutex` (for CS), $\texttt{empty}_{\text{sem}}$ (no. of empty slots), $\texttt{full}_{\text{sem}}$ (no. of filled slots)

Readers-Writers - reader & writer access common struct, writers need exclusive access / issue - ⩾ 1 writer writing
Solution (*writers may starve*) - `readCount` (no. of process reading struct), `mutex` (for `readCount` var), $\texttt{wrt}_{\text{mutex}}$ (acquired when `readCount==1`, released when `readCount==0`)
Solution (*prevent writer starvation*) - existing soln. + $\texttt{turnstile}_{\text{mutex}}$ (block readers until all writers pass thru)

Dining Philosophers - X philosophers with X chopsticks,

philosopher thinks → takes chopsticks → eats → release chopsticks / issue - deadlock, livelock
Solution - *limited eaters* - allow max $X-1$ eaters / *leftie-rightie* - $\geqslant 1$ left-hander & $\geqslant 1$ right-hander / *Tanenbaum* - `state[N]` (records curr state of each philosopher), `sem[N]` (indicates if he can start eating), `mutex` (for CS)

Memory Management

Base (starting addr) + **Limit** (ending addr) registers
* check Actual = base + addr < Limit for validity

Contiguous Memory Allocation

Process occupies contiguous memory space
Fixed partitioning - mem space split into blocks
Easy to manage, fast allocation, partition must fit the largest avail process, internal fragmentation

Dynamic partitioning - alloc space == sizeof process
flexible, no internal fragmentation, larger overhead, time consuming, external fragmentation
Allocation algo - first fit / best fit / worst fit / buddy
Compaction - consolidate empty blocks (time consuming)
Buddy System - Alloc - find S where $2^S \geqslant N$, access $A[S]$ for free block, if no free block, split $A[S+1...]$ for free blocks at $A[S]$ / Dealloc - free occupied $A[S]$, check if buddy block is free, if free, merge and repeat
Buddy blocks - starting index differ by 1 bit (XOR size)

Disjoint Memory Allocation

Page size == Frame size (same offset)
Page table - page no. → frame no., stored in PCB
no external frag., flexible, simple, internal frag. on last page, not enough mem space to fit all procs' page table
TLB - caches some PTEs, Hit - 1 access to TLB, 1 access to mem, Miss - 1 access to TLB, 2 access to mem
Memory access time = % TLB-Hit + % TLB-Miss

Protection - access bits (`rwx`), valid bit
Copy on Write - duplicate memory only when written to

Segmentation

Solves - Diff parts of process used differently
Logical addr - SegID + Offset
Segmentation table - SegID → Base & Limit (sizeof seg)
each segment is independent contiguous mem space, can shrink/grow & protected/shared independently, variable-sized → external frag.
Seg with paging - $<s, p, d>$ ($\mathbf{s}$ - index segment table to locate page table, $\mathbf{p}$ - index page table for frame no., $\mathbf{d}$ - offset within page)

Virtual Memory Management

Extends paging - `isMemoryResident` bit in PTE
Page fault - CPU tries to access non-memory resident pages, handled by OS (TRAP)
Locality principles - Temporal - recently used likely to be used again / Spatial - close to used addr likely to be used

Demand Paging - load only when needed, fast startup time, small mem footprint, high starting page faults, thrashing effect

Page Table Structures

Direct paging - all PTE in same table
Multi-level paging - PTE points to a smaller page table, virtual addr - page dir #, page #, offset
dont need to load all PTEs
Inverted page table - frame no. → pid & page no.
1 table for all processes, slow translation

Page Replacement Algorithms

Evaluation - $T_{\text{access}} = (1-p) \times T_{\text{mem}} + p \times T_{\text{page\_fault}}$
p - prob. of page fault, goal - reduce p
OPT - replace page used latest down the road, require future knowledge → not realizable, good for comparison

FIFO - always evict oldest page loaded
simple implementation, doesn't exploit temporal locality
⇒ **Belady's Anomaly** (more frames → more page faults)

LRU - replace page not used for longest time
exploit temporal locality, doesn't suffer Belady's Anomaly
Implementation - time-of-use counter - replace page with smallest time-of-use, possible overflow / "stack" - referenced page put on stack top, replace page at stack bottom, entries can be removed from anywhere in stack → harder to implement in hardware

Second-chance - FIFO with accessed bit (0 = page replaced, 1 = `accessed--` & given 2nd chance)
suffers from Belady's Anomaly

Frame Allocation

Equal allocation - $\frac{\text{no. of frames}}{\text{no. of competing processes}}$ frames allocated
Proportional allocation - $\frac{\text{size}_p}{\text{size}_{total}} \times$ no. of frames
Local replacement - victim page selected within process causing page fault
frames allocated is constant ⇒ same performance with multiple runs, if not enough frames, can't help
Global replacement - victim selected among all frames
enables self-adjustment ⇒ not enough frames can get from other proc, affected by other proc, inconsistent perf

Thrashing - insufficient frames, I/O heavy to page faults, global replacement - "steals" pages from other proc ⇒ cascading thrashing, local replacement - hogs I/O ⇒ degrade perf of other proc
Working Set Model - contain most of the active pages within time interval (Δ) to reduce possibility of page fault (W(end time, Δ)), Δ too big ⇒ include pages from other locality, Δ too small ⇒ miss pages in curr locality

File Management

File system - self-contained, persistent, efficient
Access types - Read, write, execute, append, delete, list
Protection - Owner, group, users

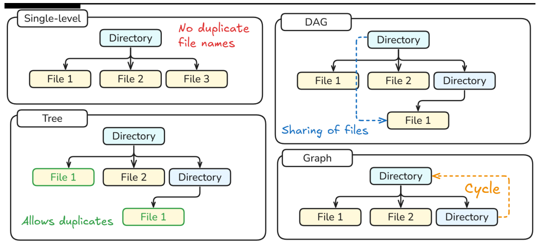
File data

Structures: array of bytes, fixed length, variable length
Access: Sequential, random, direct

File operations

Open-file table - system-wide & per-process

Directory

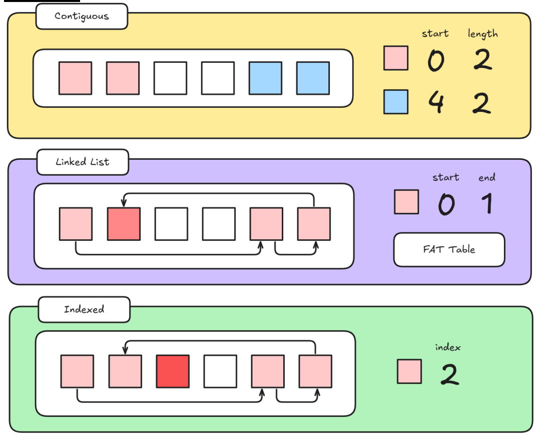


Filenames Absolute pathname, relative pathname

Hard Link Limited to file only `ln`
Low overhead, Deletion problem
Symbolic Link `ln -s`
Simple deletion, Larger overhead, can be broken

File System Implementation

File Block Allocations



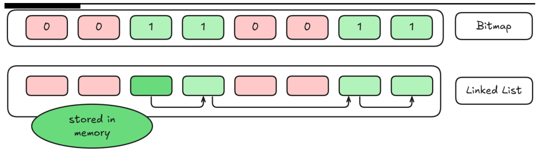
Contiguous:
Simple, fast access, External fragmentation, early specification of file size

Linked list:
No fragmentation, Slow random access, disk block used for pointer, less reliable

Linked list v2.0 (FAT) - last block contains terminator
Faster random access, Keep tracks of all disk blocks (large space)

Indexed:
Less memory overhead, fast direct access, Limited maximum file size, index block overhead

Free Space Management



Bitmap Easy to manipulate, Keep all in memory
Linked List Easy to locate free block, only first pointer needed in memory, High overhead

Directory Structure

Linear list Requires linear search
Hash table Fast lookup, Limited size, hash function dependent

File information (approaches)

1. Store everything in directory entry
2. Store only file name and pointers

File System Case Studies

FAT

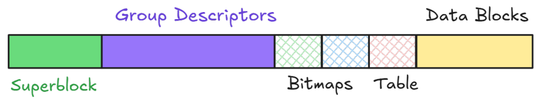
Entry codes - FREE, Block number, EOF, BAD

File names 8 + 3 characters
Extended by using multiple dir entries. (Virtual FAT)

Variants
Disk cluster - Number of contiguous disk blocks
Cluster size - Larger usable partition, larger internal fragmentation
FAT Size - Bigger FAT → More bits

Ext2

Blocks - Disk space split into blocks (grouped as blocked groups)
INode contains file metadata, data block addresses



Superblock Describes whole file system, duplicated for redundancy
Group descriptor Describes block group, duplicated
Bitmaps Keeps track of usage status of blocks & I-nodes
INode table Array of I-Nodes

I-Node structure

Pointers - 1-12 - direct / 13 - single indirect / 14 - double indirect / 15 - triple indirect
E.g. Block size = 1KB, block addr size = 4 bytes
1 block ⇒ $\frac{1024}{4}$ = 256 addr / Direct - 1 block ⇒ 1KB / Single indirect - 1×256 ptr $\times$ 1KB = 256KB / Double indirect - 1×256ptr $\times$ 256ptr$\times$ 1KB = 64MB
Fast access for small files & flexibility for large files

Hard link - I-Node ref count maintains no. of refs, can't link to directories
Symbolic link - only pathname, easily invalidated