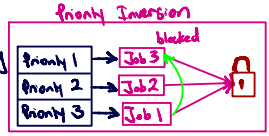
# CS 162 Midterm 2 Notes

Jeffrey Shen

## Scheduling

Scheduling decides which threads are given access to resources moment to moment
Goals: 1) ↓ Response Time 2) ↑ Throughput (operations/sec), ↓ overhead 3) ↑ Fairness
Waiting Time: time before it got scheduled
Completion Time: Waiting time + running time
Prioty handles differences in importance, watch Starving
Priority Inversion: high prioty task blocked waiting on low prioty thread




| Choose | For |
|---|---|
| FCFS / FIFO | CPU Throughput |
| SRTF Approx. | Avg Response Time |
| | I/O Throughput |
| Linux CFS | Fairness (CPU Time) |
| Round Robin | Fairness (CPU Wait Time) |
| EDF | Meeting Deadlines |
| Priority | Favor important |

### Policies

#### 1) First Come, First Served (FCFS/FIFO)

Idea: One program scheduled until done
Pro: Least overhead, simple
Con: convoy effect (short processes stuck behind large ones)

#### 2) Round Robin (RR)

Idea: Each process gets small unit of CPU time (quantum)
Pro: With n process, q time quanta, max waiting time $(n-1)q$
Con: Lots of convex switching, high completion time

#### 3) Shortest Job First (SJF), Shortest Remaining Time First (SRTF)

Idea: Run job with least amount of computation to do
Pro: Optimal!!
Con: Need to be able to see future, know process length

#### 4) Lottery Scheduling

Idea: Give job some # lottery tickets, randomly choose ticket
Pros: On avg, CPU time proportional to # of tickets
Cons: Could choose long jobs, low prioty, unfair for less jobs

#### 5) Multiple-Level Feedback Scheduling

Idea: Multiple queues, adjusts queue as process is run
have queues w/ Fixed prioty scheduling, time slice
Pro: Approximates SRTF
Con: Can counter by requiring IO and staying in highest

#### 6) Earliest Deadline First (EDF)

Real Time Scheduling

Idea: Tasks w/ deadlines, Computation Times, choose closest deadline
Feasible if $n$ tasks, Computation Time $C$, deadline $D$
Pro: for Real Time Scheduling

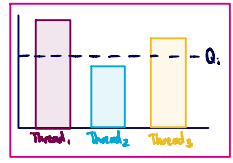
$$\sum_{i=1}^{n} \left(\frac{C_i}{D_i}\right) \leq 1$$

#### 7) Stride Scheduling, Linux Completely Fair Scheduler (CFS)

Idea: Track virtual CPU time per thread, gets equal share, choose thread w/ least CPU time

Basic equal share: $Q_i = \text{Target Latency} \cdot \frac{1}{N}$ ← threads
Weighted share: $Q_i = \left(\frac{w_i}{\sum_p w_p}\right) \text{Target Latency}$
↖ allow different rates of execution, ↓ weight ↑ Physical CPU time

Target Latency: period of time where every Service gets Service
Add min granularity to ensure each process gets to run, min time slice




## Deadlocks

Deadlock: cyclic waiting for resources, deadlock ⇒ starvation, starvation ⇏ deadlock

### Requirements for Deadlock

1) Mutual Exclusion and bounded resources
   - one thread at a time use resources
2) Hold and wait
   - thread holding resource waits to acquire more
3) No preemption
   - resource released voluntarily
4) Circular Wait
   - set of waiting threads waiting on each other

### Deadlock Prevention

1) Provide sufficient resources, VM unlimited
2) Abort requests, acquire atomically
3) Fail if waiting too long, force give up
4) Order resources usage in same order

### Deadlock Avoidance

Prevent System from entering unsafe State
↳ Use Banker's Algorithm
Safe space: can prevent by delaying acquisition
unsafe space: can unavoidably lead to deadlock, with certain acquisition
Deadlocked state: exists a deadlock

| Deadlock recovery | Deadlock denial |
|---|---|

### Bankers Algorithm

- Check if resource request leads to unsafe State
- State max resource needs in advance
  Allow thread to continue if
  available resources − # requested ≥ max

Idea:
Allocate resources dynamically
- Evaluate each request & grant if some ordering is deadlock free
- Pretend request granted, run deadlock detection algo

Deadlock Detection Algo [Banker]

add to unfinished
for each thread unfinished
if [Request [Max − Alloc] ≤ Avail]
remove from unfinished
Avail = Avail + Alloc

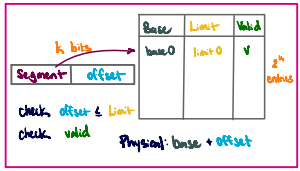
## Memory

Virtual Memory to multiplex memory, protection, controlled overlap

Pages: small fixed size physical memory chunks
Memory Management Unit (MMU): Translation box converts between virtual & physical addresses;
Page Table: one per process, has physical page & permission (R/W/Valid)
kernel handles evicting, invalidating, disk

## 1) Base & Bound / Segment Mapping

Idea: set registers w/ base and limit
Pro: Simple
Con: Internal and External Fragmentation, no Sparse address space support or interprocess sharing

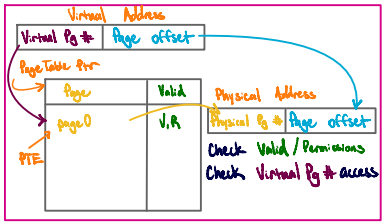



## 2) Inverted Page Table

Idea: use a hash table to map VPN to PPN
Pro: Efficient lookup
Con: complexity of hash chains, poor cache locality

## 3) Simple Paging

Idea: Translations in Page Table
Pro: Able to share memory: point to same physical page #, easy to (re)allocate memory
Cons: Page Table too big
Internal Fragmentation




## 4) Multilevel Page Table

Idea: Tree of Page Tables w/ fixed size, Save PageTablePtr (CR3)


(10b-10b-12b)

Pros: allocate just needed PTE, easy memory allocation, sharing
Cons: One pointer per page, ≥2 lookups per reference

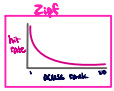
Page table Entry (PTE): pointer to next level page table or actual page, permission bits

## Caching

Translation Lookaside Buffer (TLB): Cache for translations
records just end result, recent VPN to PPN, include ProcessID, hardware
Temporal Locality: Time locality, recently accessed closer
Spatial Locality: Space locality, contiguous blocks

### Sources of Cache Misses

1) Compulsory: first access to block
   ↳ Clustering, Working set tracking
2) Capacity: cache cannot contain all blocks accessed
   ↳ increase cache size
3) Conflict: multiple mem location mapped to same cache location
   ↳ increase cache size, increase associativity
4) Coherence: other process updates memory




### Types of Caches

1) Direct Mapped Cache: single block per set, index


2) N-Way Set Associative: N-direct mapped caches


3) Fully Associative: Every block can hold any line, no index (TLB typically fully associative)


Cache typically physically indexed
Can lookup TLB and cache simultaneously


### Cache


Data = N * L * K
Write Policy / Eviction Policy

Block: minimum quantum of caching
Index: lookup candidates in cache, identify set
Tag: identify actual copy
Write Through: info written to both block and lower lvl memory
Write Back: info written only to blocks, write when evict
Zipf distribution: increasing size of cache has diminishing returns

### Average Memory Access Time (AMAT)

AMAT = Hit Rate × Hit Time + Miss Rate × Miss Time

## Demand Paging

Demand Paging: only keep active pages in memory, as cache: fully associative, LRU, write back, 1pg blocks

If invalid PTE:
1) MMU traps to OS w/ Page Fault
2) Find & replace page w/ page from disk
3) Reset Page Table & restart instruction

Freelist: keep set of free pages by Clock Algorithm
Working Set: group of pages accessed by process recently
Swapping: some or all of previous process moved to disk to make room
- Can share Code Segment, setting read-only

### Page Replacement Policies

1) FIFO: evict oldest page
   Con: evicts heavily used pages
2) RANDOM: choose random page for replacement
   Con: unpredictable
3) MIN: replaces page not used for longest time, optimal
   Con: don't know future
4) LRU: replace page not used for longest time
   Con: too much overhead

5) Second Chance
   Split into Active and Second Chance List
   Pro: few disk accesses
   Con: increased overhead trapping

Approx LRU:

6) Clock Algorithm Replace an old page, partition into old and young
7) Nth Chance: N chances to stay in memory

### Allocation of Memory for Processes:

Equal allocation, Proportional allocation, priority allocation
- Can set lower and upper bound for memory

Thrashing: busy swapping pages in and out w/ little progress b/c enough pages

# CS 162 Midterm 1 Notes — Jeffrey Shen

## Operating System Overview

Purpose: Special layer of software that provides application software access to hardware resources

1) Illusionist: Provide simple abstractions of physical resources (infinite memory, virtualization)
2) Referee: Manage protection, isolation, and sharing of resources (resource allocation, communication)
3) Glue: Common Services (Storage, Networking, Sharing, look and feel)

## Four Fundamental OS Concepts

### 1) Threads

- Single unique execution context
- has own Program Counter, Registers, Execution Flag, stack, Memory State
- When executing and resident on processor: running
- When not loaded in: suspended
- In order to execute multiple processes, multiplex in time, virtual cores (TCB)
  - Store other threads in Thread Control Block

### 2) Addresses

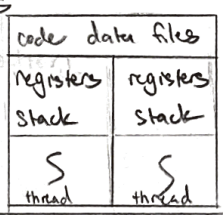
- Address Space: the set of accessible addresses + state associated w/ them

| Abstraction | |
|---|---|
| Processor | → Threads |
| Memory | → Address Space |
| Disks, SSD | → Files |
| Networks | → Sockets |
| Machines | → Processes |

- OS must protect user programs from one another & protect itself from other programs
  1) Base & Bound — base ≤ address < bound
     have base register and bound register to check address
  2) Address Space Translation (Page Table)
     Program operates in virtual (pages) translated to memory add

### 3) Process

- Execution environment with restricted rights
- Protected Address Space w/ 1+ threads
- Running program w/ memory protection
- Processes provide protection, isolation, thread provides concurrency




### 4) Dual Mode Operation

User Mode ← Transfers → Kernel Mode

certain operations prohibited

Interrupt handler to direct next steps

Transfers:
1) Syscalls — process requests system service
2) Interrupt — external async event
3) Trap / Exception — internal sync event

privileged mod

- Modes provide memory level isolation between processes

## Threads & Processes: Programmer POV

- Allow parallel programs to be run
- Multiprocessing: Multiple CPU (cores)
- Multiprogramming: multiple jobs / processes
- Multithreading: multiple threads / processes, same CPU
- Threads have non-determinism: can run in any order, leds to race cond.
- Process fork: copy current process: page table
  1) copy, new process has pid 0, parent pid > 0
- Each process/thread has kernel segment with PCB/TCB, kernel stack
  ksp stores kernel stack panter in order to reduce I/O blocking in kernel

Thread States
- Running - currently in CPU
- Ready - eligible, not running
- Blocked - ineligible to run

| | Process | Thread |
|---|---|---|
| Creation | fork() | pthread_create() |
| Page Table | Distinct | Same |
| Registers, ip | Distinct | Distinct |
| Stack | Separate & inaccessible | Separate but accessible |
| Heap, static var | Separate | Shared |
| File descriptors | Separate | Shared |
| Synchronization | wait(), waitpid() | pthread_join(), semaph locks |
| Overhead | Higher | Lower |
| Protection | Higher | Lower |

parallel ⇒ concurrent

Concurrency ⇏ parallel

# File I/O, Devices

Everything is a file: open, read, write, close

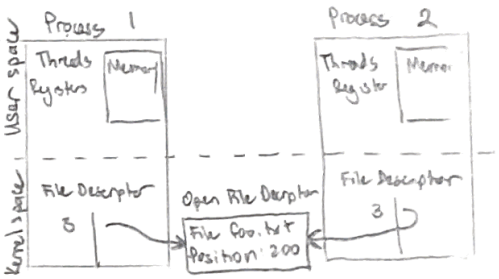
High level FILE: buffered, has fd, buffer, lock

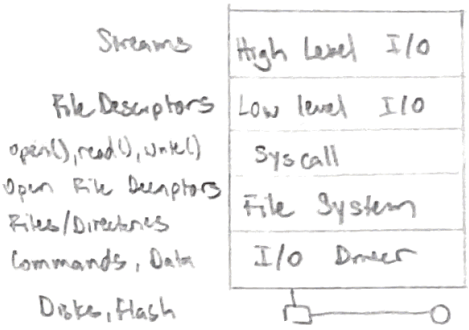
Low level file: returns fd (not buffered)

Drivers: device specific code in kernel that interacts directly w/ device hardware

Top half: accessed through syscalls, initiate I/O, put waiting thread to sleep

Bottom half: runs as interrupt routine, wakes up sleeping threads when I/O complete




Can have same file diff descriptors

# IPC, Pipes, Sockets

Interprocess Communication (IPC)

- communication between protected environments (processes)

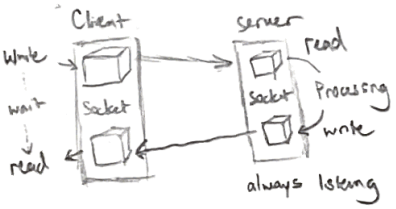
Pipes: act as single queue between processes

- write only on one side, read only on other

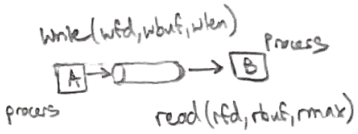
Sockets: allow two queues, communication between

- communication between multiple processes on different machines, socket / bind / connect / listen

Sockets




Pipes




# Synchronization

- Many different solutions to fixing synchronization issues, w/ at least busy waiting
- Atomic Operation: operation that always runs to completion or not at all
- Mutual Exclusion: ensuring only one thread does particular thing at a time, excludes the other
- Critical Section: piece of code only one thread can execute at once
- Locks: synchronization mechanism for enforcing mutual exclusion on critical sections to construct atomic operations
- Hardware atomicity primitives disabling interrupts test & set, swap, compare & swap, load-linked & store conditional
- Separate lock variable, use hardware mechanisms to protect modifications of that var

**Semaphores**: synchronization primitive

- Down() or P(): waits for semaphore to become positive then decrements by 1, like (wait)
- Up() or V(): atomic operation increments by 1, wake up waiting P (signal)

**Monitors**: a lock and zero or more condition variables managing concurrent access to shared data

- locks for mutual exclusion and condition var scheduling constraint
- Condition variable: queue of threads waiting for inside critical section var

Hoare Monitor

```
if (isEmpty(&queue))
    cond_wait(buf_CV, &buf_lock)
```

Mesa Monitor

```
while(isEmpty(&queue))
    cond_wait(&buf_CV, buflock)
```

- Wait(&lock): Atomically release lock and go to sleep
- Signal(): wake up one waiter
- Broadcast(): Wake up all waiters

Reader's / Writers Problems

```
while (test & set (guard))
```

**Lock**: prevent others from changing critical section

acquire(&lock): wait till lock is free, then grab, run critical section

release(&lock): unlock, wake up anyone waiting

Implementation:

```
Acquire():
  disable interrupts
  if (value == Busy)
    put thread wait queue
    go to sleep
  else: value = Busy
  enable interrupts
```

```
Release()
  disable interrupts
  if anyone on wait queue
    take thread off queue
    Place on ready queue
  else: value == 0
  enable interrupts
```

**futex**

Kernelspace wait queue attached to user space atomic integer

faster, no syscalls, FUTEX_WAIT, FUTEX_WAKE

```
write(ifd, char*, strlen()+1)
word_count* vc
ref_count?
```

```
char buffer[size]
intr_disable()
```

thread_current(): get current thread

strcpy(dest, src, len): copy from src to dest strlen()+1

cx) list_entry(e, word_count_t, elem) list_init(&cv->waiters)