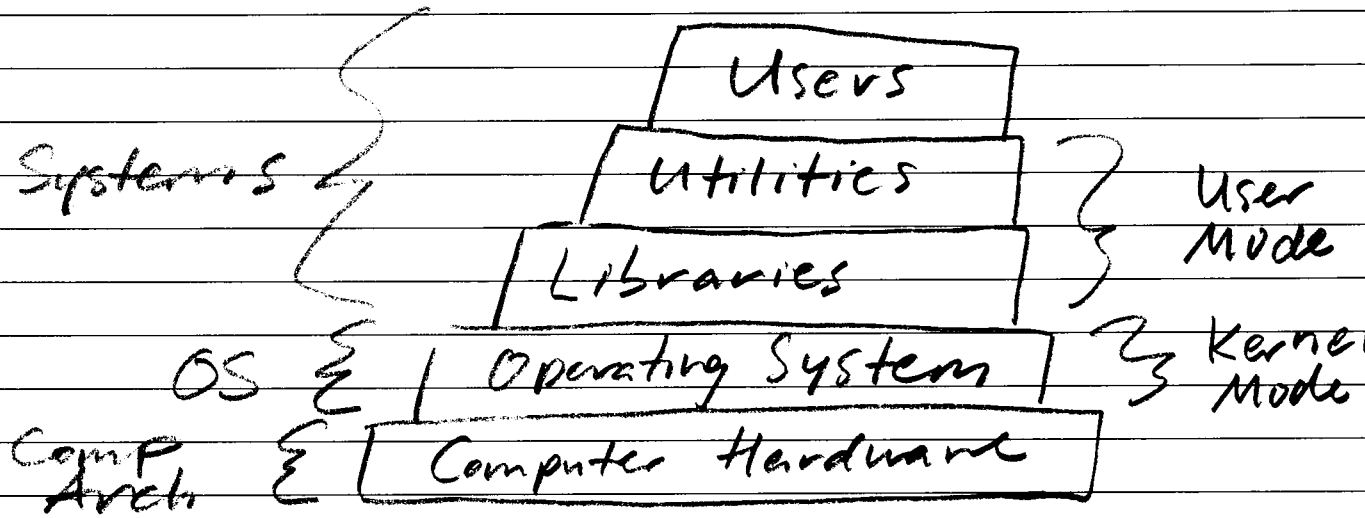
Operating System Principles

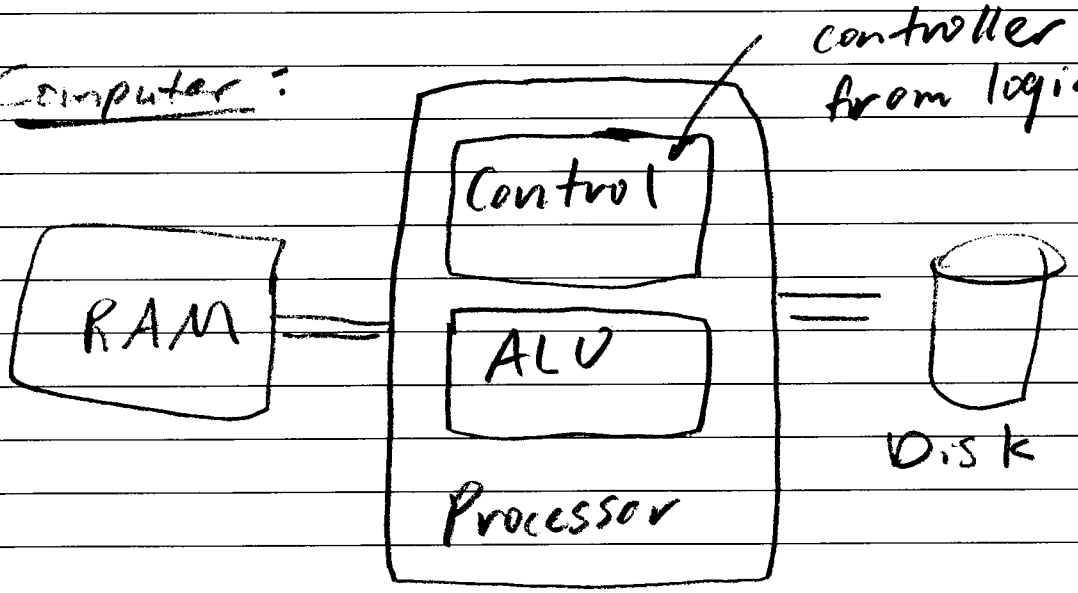
Date. 8.26.25
No. 1

House of Cards:

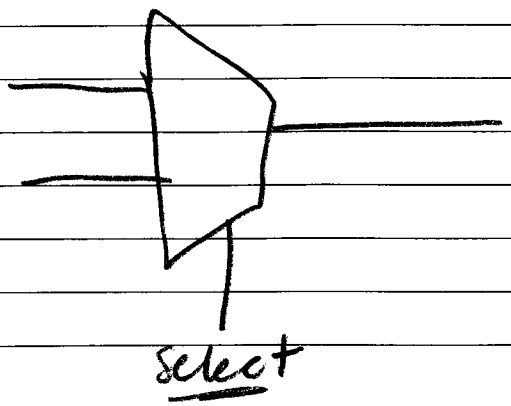


Kernal: another word for Operating System

Computer:

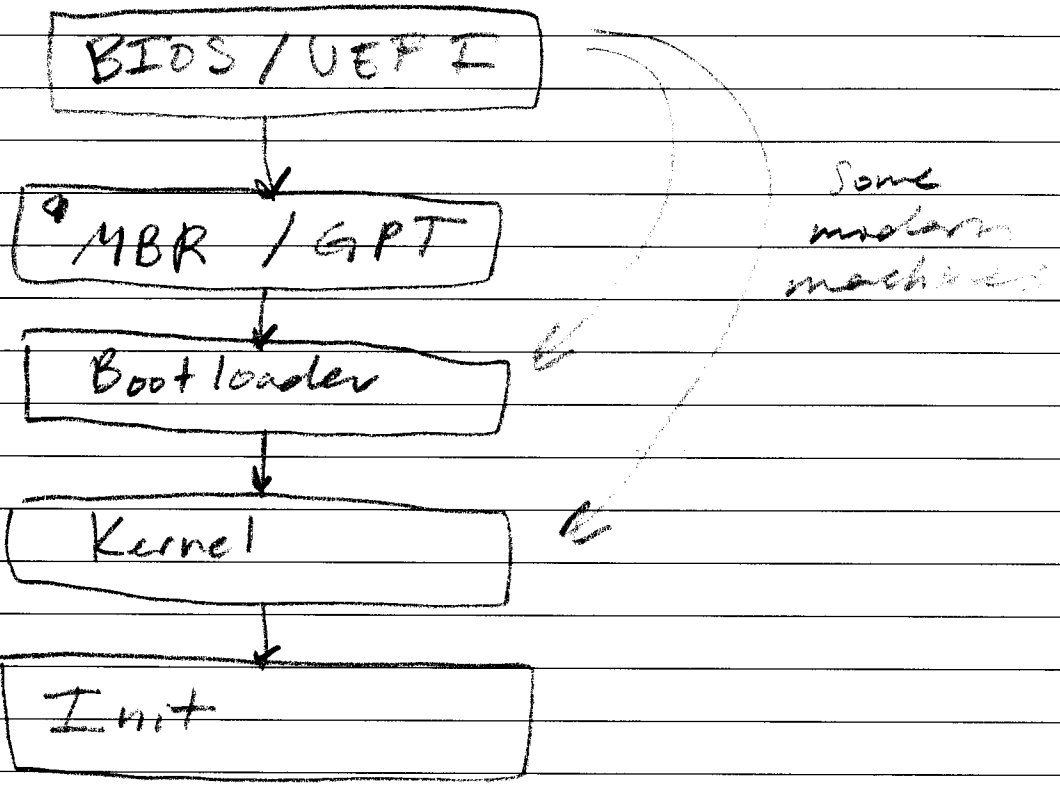


Binary: just controls the selects in the ALU
everything else is hardcoded



Date.
No.

Boot Sequence:



* master boot record / general partition table

Test Question: T/F: the OS is the first program to run on your computer after you boot

Answer: FALSE

Operating System: A set of abstractions

- Computer Architecture (constraints)
- Data Structures (Linked Lists, Trees)
- Algorithms (Policies, Trade-offs)

OS Taxonomy, History, Themes

Date. 8.28.25
No. 2

Question 4 in Project 01 quiz is on exam verbatim. Project quiz questions in exam in general too

Mainframe: process many jobs at once (timesharing, transaction processing, batch)

Server: provide services to multiple users (print, file, web, etc.)

Visa still uses mainframes for credit cards

Personal: support a single user (laptop/desktop)

Handheld: usually found in PDAs or mobile phones

Personal → optimize for latency
Handheld → optimize for energy/power
Mainframe → optimize for throughput
Server → optimize for throughput

Embedded: run on devices not traditional general purpose computers (TVs, cars, microwaves, etc.)

Real-time: provides absolute guarantees that a certain action will occur by a certain time

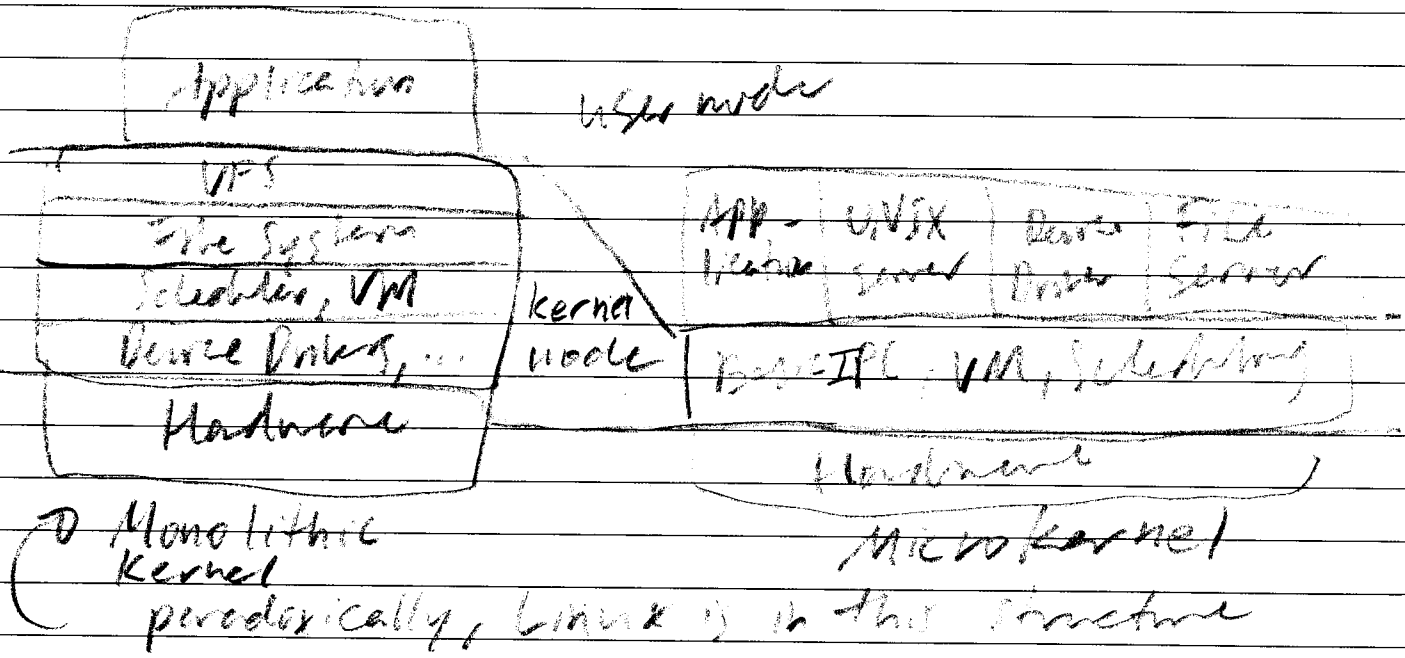
Embedded → optimized for reliability
Real-time → optimized for reliability

Date.
No.

OS Structure:

Monolithic Kernel: A single program (or collection of procedures linked together) run in kernel mode

Microkernel: Split the OS into small, well-defined modules, one of which runs in kernel mode and manages communications



Monolithic Kernel

periodically, Linux is in this structure

Context Switch: switching between user and kernel mode (costly operation)

* Microkernel better for security and reliability because of its modularity

* Windows was a microkernel at first and is now a hybrid kernel for better performance
→ same with macOS

Date.
No.

Multiprogramming: enable multiple jobs to run concurrently

Memory Protection: disallow one program from manipulating data of another program

Timesharing: Split processing time with multiple users

OS Themes: An OS is a body of software that enables other programs to interact with each other and the physical hardware resources in an efficient manner

To do this, it utilizes:
1. Virtualization
2. Concurrency
3. Persistence

Date. 9.2.25
No. 03

System Calls

What is a system call?

→ request a service from the OS kernel

I/O : open, close, read, write, lseek
File : stat, chmod
Directory: opendir, closedir, readdir

- strace [command] will show you the system calls heavy used by a program
  → ltrace does that for libraries

- man [command] → gives info on command
- man 2 [command] → gives info on system call

API or resource

- to see API definitions of system calls, look at:
  /usr/include/bits/syscall.h
  → useful stuff here in general

Ex. hello-asm.s

```
# write(1, "hello world\n", 13)
movl $4, %eax          # System call number 4
movl $1, %ebx          # standard fd = 1
movl $msg, %ecx        # msg to write
movl $len, %edx        # msg length
int $0x80              # Trap
```

↳ short for interrupt

```
# exit(0)
movl $1, %eax
movl $0, %ebx
int $0x80
```

note. 32-bit version

Date.
No.

ex. hello-asm-64.s

```
# write(1, "hello world\n", 13)
mov $1, %rax      # write is system call 1 now
mov $1, %rdi
mov $string, %rsi
mov $len, %rdx
syscall           # trap / interrupt

# exit(0)
mov $60, %rax     # system call 60 (exit)
mov $0, %rdi      # exit status
syscall           # trap
```

note - 64-bit version has different system call numbers!

How is memory organized?

Stack (local variables, parameters)

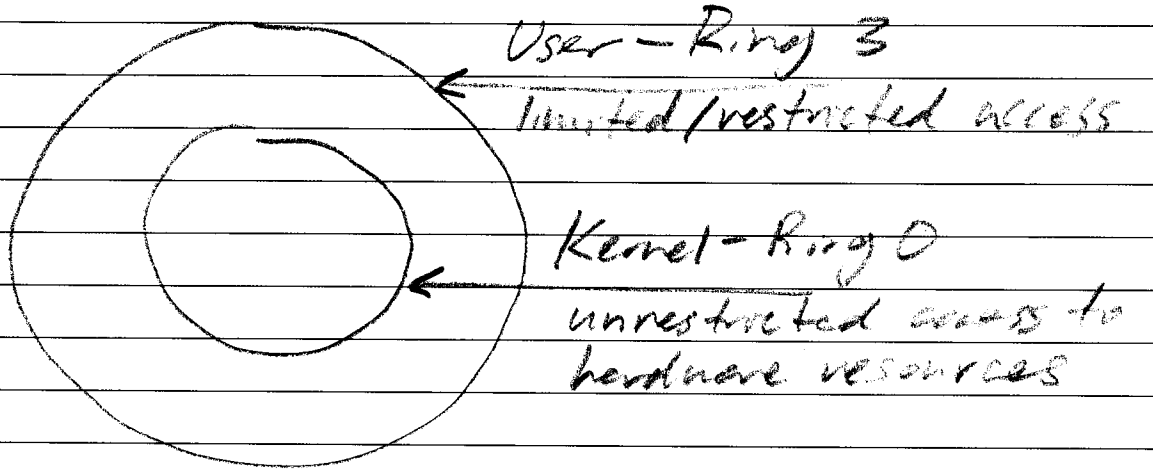
Heap (user managed memory)
Data (string literals, global, static)
Code (text, instructions)

* abstraction of the address space! *
it an "image" of memory

Date.
No.

What is a trap / interrupt / exception?

An event that forces the CPU to transition from user mode to kernel mode



What are the different types of events?

Class	Cause	Async / Sync	Return
Interrupts	Signal from I/O Device	Async	Always returns to next instruction
Trap	Intentional exception	Sync	
Fault	Potentially recoverable error	Sync	Might return
Abort	Nonrecoverable error	Sync	Never returns

When these events occur, the CPU looks at its **trap table** or **interrupt vector table** to determine what to do next

note, Tables are just arrays, and the system call #'s are just indices in the trap table

Date.
No.

How are events handled?

1. Register handlers with trap table

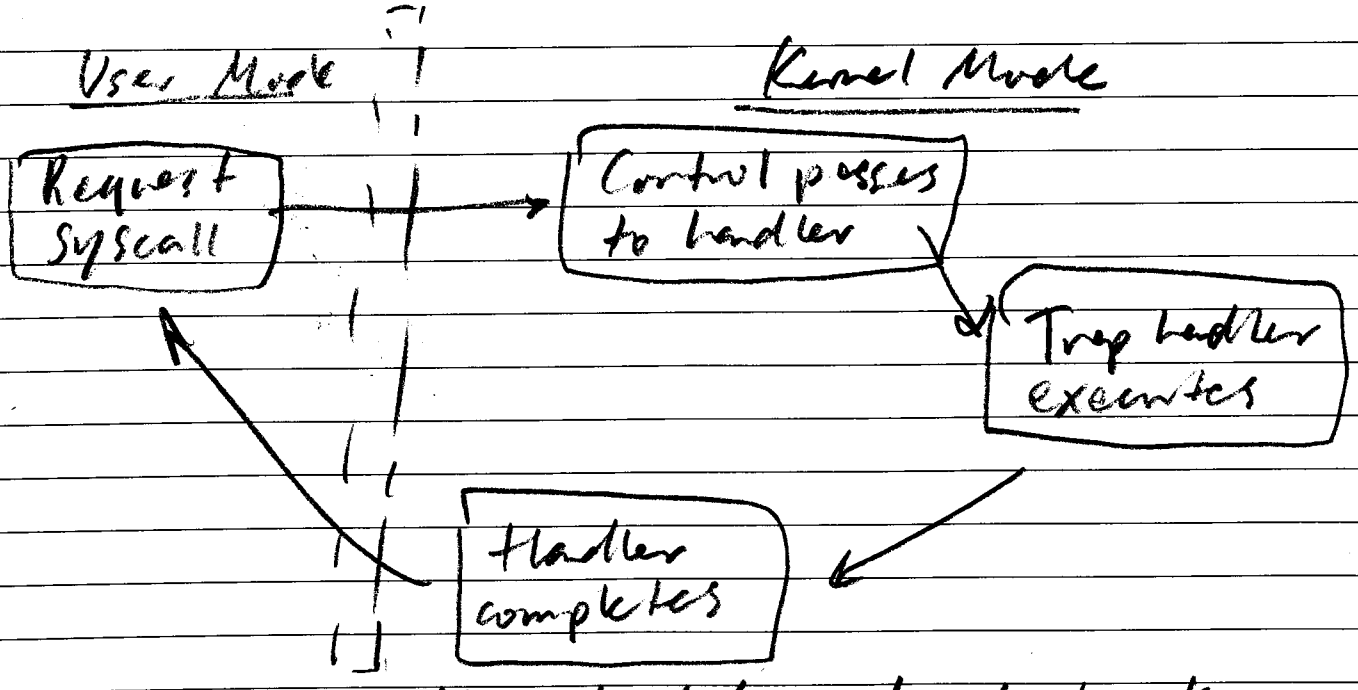
2. User process
a) Records syscall # (1)
b) Records syscall args (5)
c) Triggers trap
d) CPU uses trap table to select handler
e) Handler performs operation

Trap Table

#	syscall
1	read(int i) read message
2	write(char *s, int n) write message
3	kill(int pid, int sig) send signal to process
4	exit(int status) terminate process

How does a system call work?

To request a service, a user application performs a trap to interrupt the CPU, and forces it to look up a handler function in the trap table



note. The trap table is located at boot by the kernel, which registers which function to execute based on the syscall number

Date. 9/4/25
No. 04

Processes

* must error-check all system calls, they fail! *

note. when a higher ring tries to access memory in a lower ring, you get a seg-fault!

What is a Process?

A running (loaded) instance of a program, also a unit of allocation (resources)

- Address Space (memory)
  code, data, heap, stack
- Kernel State
  PID, owner, file descriptors
- Execution Context
  PC, Stack Pointer, Registers

} Machine State

note. a process is just a data structure!
in linux its just a tree represented in a struct!
ps tree shows this

thread: execution context of a process

note. /proc shows all PID's
echo $$ shows my terminal's PID
ps ux parses /proc/stat

Date.
No.

DOS: Disk Operating System

- older version of Windows

DOS + DOOM: people used it to play DOOM because DOS only runs one process at a time

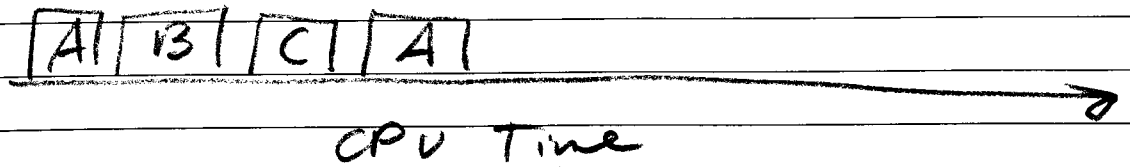
Advantages:
- uses all of your computer resources
- dedicate hardware access resources

Disadvantages:
- can't have background tasks
- if a program crashes, whole system crashes
- poor utalization

What is a process used for?

The OS will virtualize the CPU via the process abstraction

Each task is associated with a process which gets a certain slice or share of the CPU time



Date.
No.

How do we coordinate multiple processes?

Cooperative:

The OS trusts the processes of the system to behave reasonably

Pre-emptive:

The OS uses a timer interrupt to periodically suspend a running process and possibly switch to another process

How do we switch from one process to another?

When a timer interrupt goes off, we have the option of performing a context switch (i.e. switch from one process to another)

OS

Call Scheduler
Save Machine State (Current)
Load Machine State (next)

Hardware

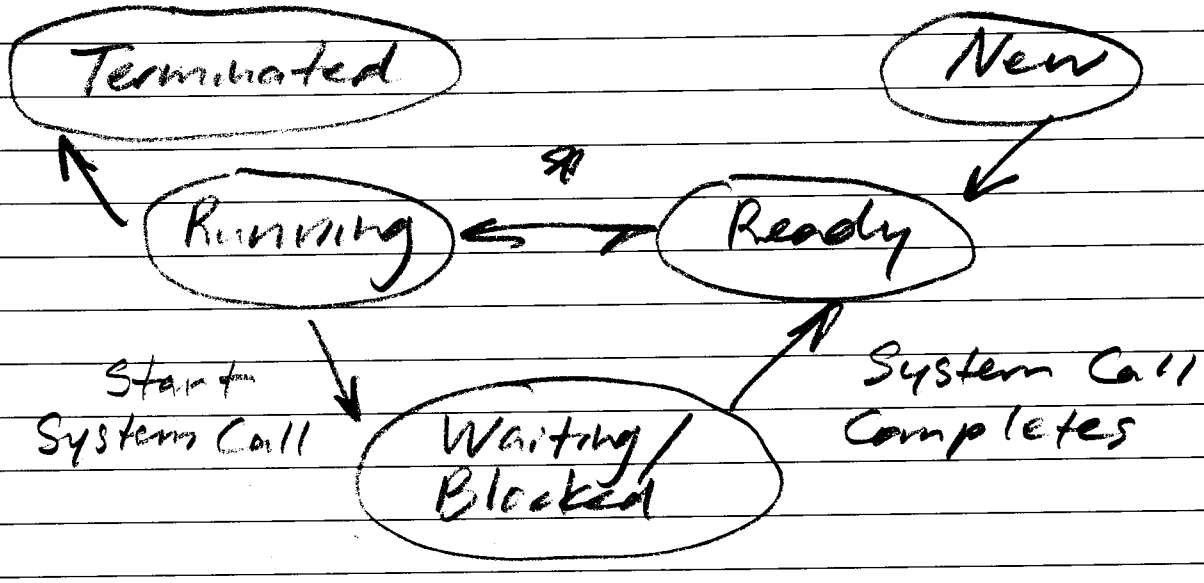
Timer triggers
Save execution context
Switch to kernel mode
Goto trap handler

Load execution context
Switch to user mode
Goto new program counter

note, clock/timer interrupt controlled by a quartz crystal in the CPU

Date.
No.

What States can a Process go through?



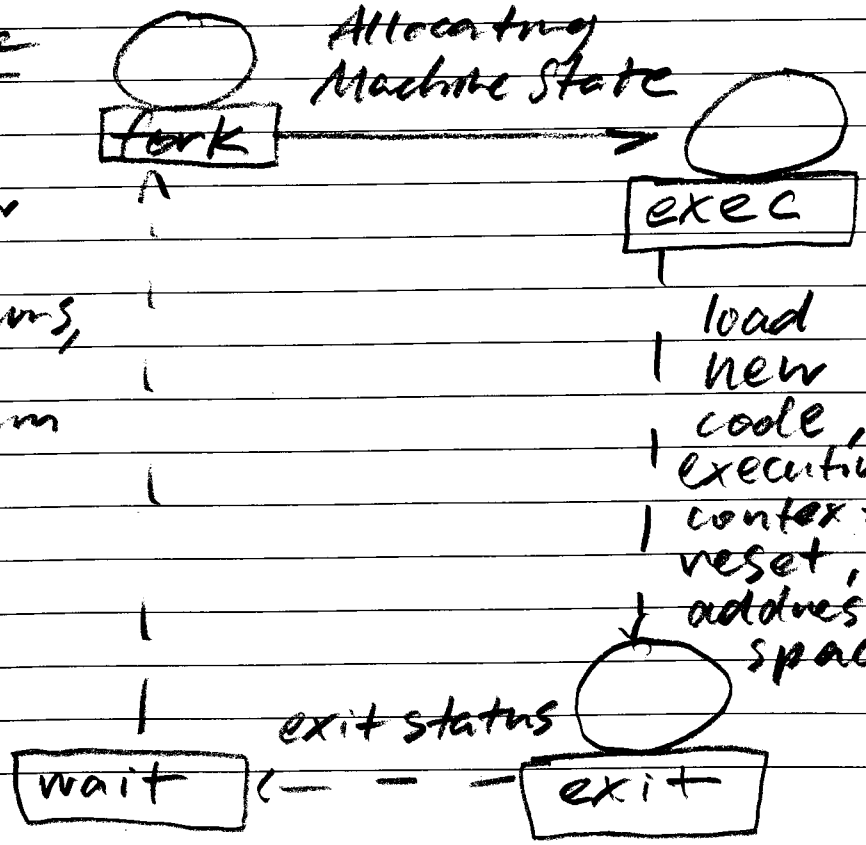
Ready -> Running (Scheduler picks you)
Running -> Ready (Timer interrupts you)

note. This diagram WILL be on the exam

note. You can go from any state to terminated

Process Life Cycle

1. Parent: fork to create new process
2. Child performs actions, possibly exec to run another program
3. Child exit to terminate
4. Parent wait for child process to get its state



Date. 9/19/25
No. 05

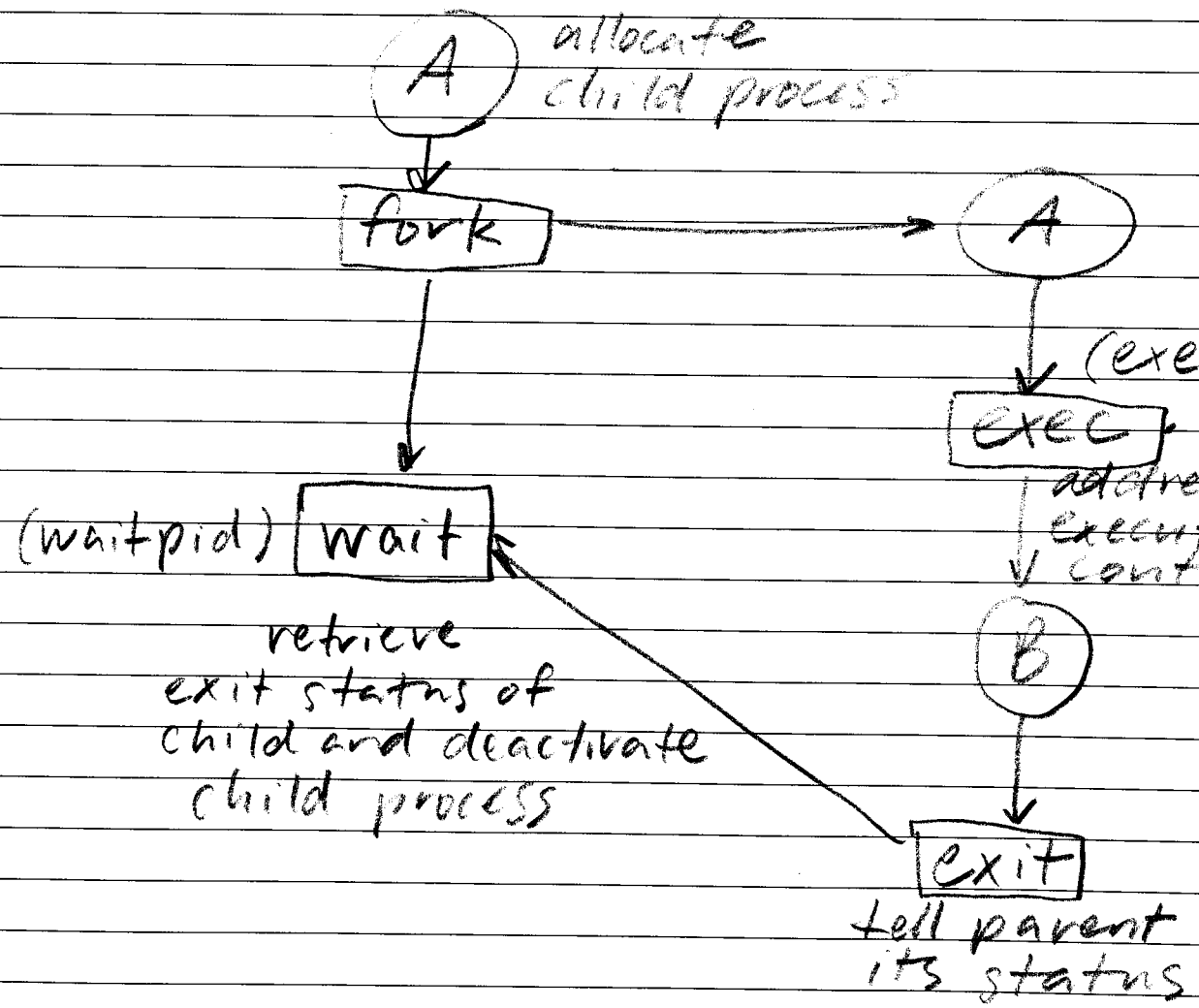
Scheduling (FIFO, RR)

Exam. Need to know!
- FIFO
- Round Robin
- MLFQ

What is the purpose of the scheduler?
→ to determine who runs next

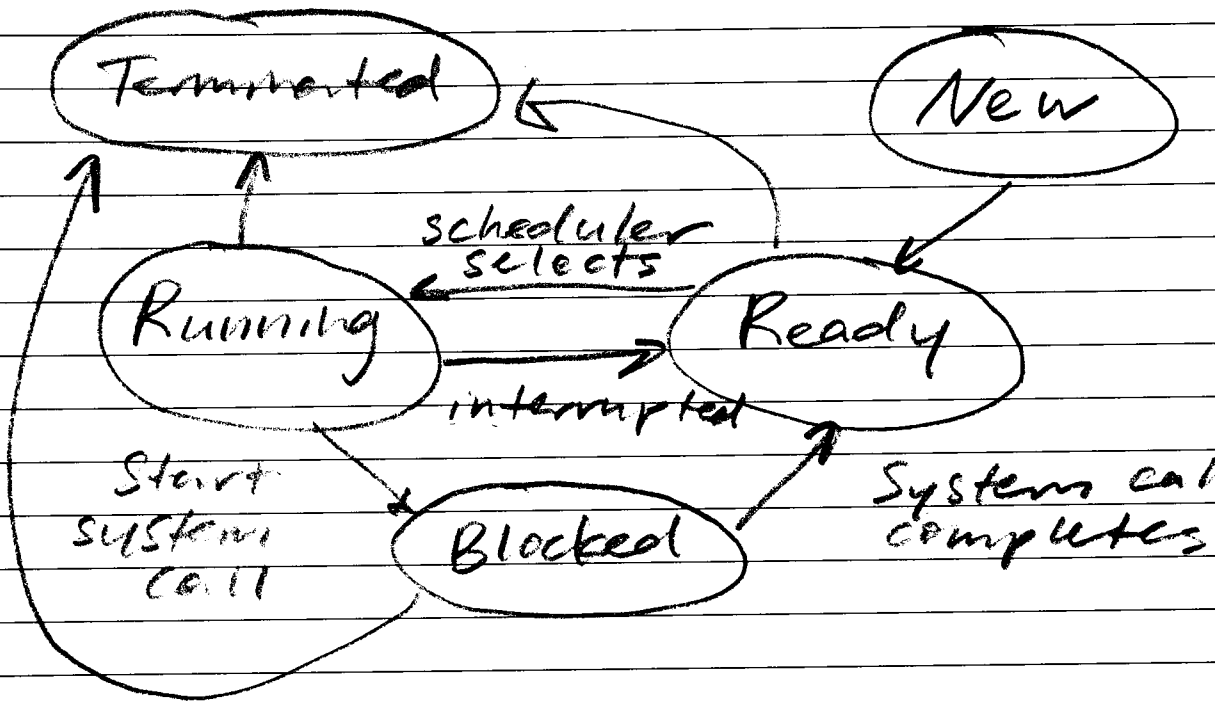
Scheduling Policy / Discipline:
→ decision making process (FIFO, RR ...)

Process Life Cycle:



Date.
No.

Process States:



Process Context Switch:

hardware

Timer Interrupt
Save Execution Context
Switch to kernel Mode
Goto trap handler

Operating System

Call Scheduler
Save Machine state (current)
Load Machine state (next)

Load Execution Context
Switch to user mode
Goto new program counter

Date.
No.

When does a scheduler execute?

1. Timer interrupt
2. Process dies
3. System call

Scheduling assumptions:

1. Each job runs for the same amount of time
2. All jobs arrive at the same time
3. Once started, each job runs to completion
4. All jobs only use CPU (no I/O)
5. Runtime of each job is known beforehand

to make scheduling policies, we need to consider our workload, or collection of processes running on our system

1. Start simple, and then slowly remove these assumptions

How do we evaluate a scheduling policy?

$$T_{turnaround} = T_{completion/finish} - T_{arrival}$$

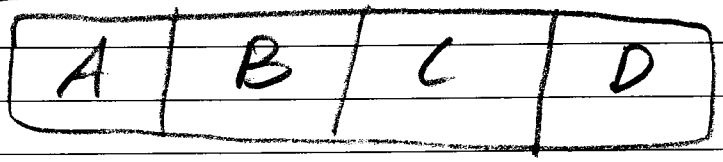
$$T_{response} = T_{firstrun/start} - T_{arrival}$$

need to consider these measurements

- latency/delay, ie how soon will program run?
- throughput, ie. how much work is done?

Date.
No.

FIFO:

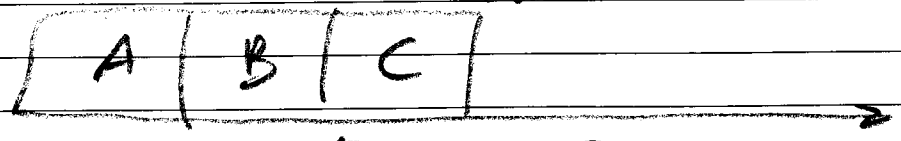


Execute jobs in the order they arrive

Python Pseudocode:

```
class Scheduler:
    running: Processes actively on CPU
    waiting: Processes ready to run

def schedule_fifo(s: Scheduler):

    while s.running.size() < s.cores and
          s.waiting.size():
        process = s.waiting.pop()
        StartProcess(process) #TODO: Error Check
        s.running.push(process)
```

Ex. Jobs A, B, C arrive at time 0 and run for 5 seconds each

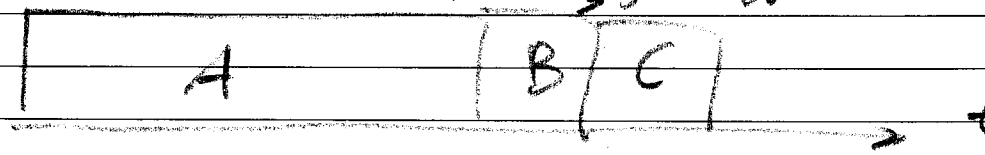


$$T_{turnaround} = \frac{5+10+15}{3} = 10 \text{ sec/job}$$

$$T_{response} = \frac{0+5+10}{3} = 5 \text{ sec/job}$$

Date.
No.

Ex. Job A arrives at t=0 and runs for 30s.
B and C arrive at t=0 and run for 5s

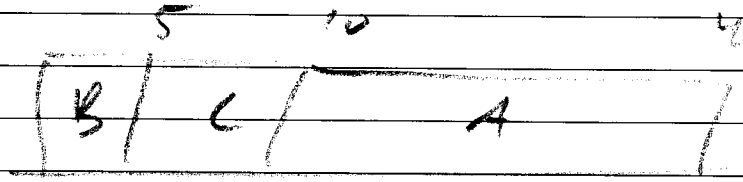


$$T_{turn} = \frac{30+35+40}{3} = 35 \text{ s/job}$$

$$T_{resp} = \frac{0+30+35}{3} = \sim 21 \text{ s/job}$$

Convoy Effect: a longer job blocks shorter jobs from being able to run

Ex. To combat the convoy effect, we can just run the shortest job first (SJF)



$$T_{turn} = \frac{5+10+40}{3} = 18 \text{ s/job}$$

$$T_{resp} = \frac{0+5+10}{3} = 5 \text{ s/job}$$

BUT:
- if A comes first still suffers from convoy effect
- if a bunch of small processes spam CPU, the larger processes starve

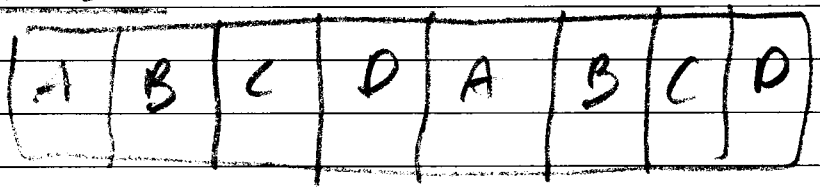
Starvation: processes don't get any CPU time

Date.
No.

FIFO Summary:

- Straightforward
- good turnaround time
- Poor response time
- Suffers from convoy effect and starvation

Round Robin:



instead of running jobs to completion, use a periodic timer interrupt to rotate through jobs

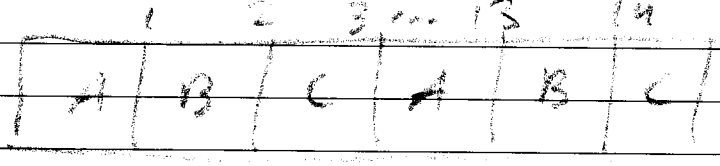
- this is a fair policy since we divide the CPU into time slices

Python Pseudocode:

```
def schedule_robin(s: Scheduler):

    if s.running.size() == s.cores:
        process = s.running.pop()
        PauseProcess(process)  #kill(process.pid,
        s.waiting.push(process)  SIGSTOP)

    while s.running.size() < s.cores and
        s.waiting.size():
        process = s.waiting.pop()
        if process.pid == -0:
            StartProcess(process) # TODO: Error check
        else:
            ResumeProcess(process) #kill(process.pid,
        s.running.push(process  SEGCONT)
```

Date.
No.

Ex. Jobs A, B, C arrive at time 0 and run for 5 seconds each



$T_{turnaround} = \frac{13 + 14 + 15}{3} = 14$ s/job

$T_{response} = \frac{1 + 2 + 3}{3} = 2$ s/job

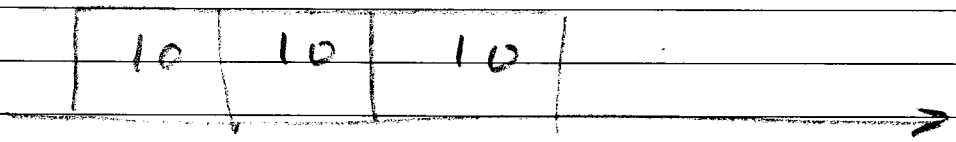
Round Robin: Summary

- Straightforward
- Poor turnaround time
- Good response time
- Preemption
- Overhead (context switching often)

Scheduling (MLFQ, Lottery)

Date. 9.11.25
No. 06

Ex. jobs A, B, C arrive at t=0 and run for 10 seconds each



	FIFO	RR
Turnaround	$\frac{10+20+30}{30} = 20$ s/j	$\frac{28+29+30}{30} = 29$ s/j
Response	$\frac{0+10+20}{3} = 10$ s/j	$\frac{0+1+2}{3} = 1$ s/j

Ex. Jobs A, B, C arrive at time 0. A runs for 10s and B, C for 1s

	FIFO	RR
Turnaround	$\frac{10+11+12}{3} = 11$ s/j	$\frac{12+2+3}{3} = 5.6$ s/j
Response	$\frac{0+10+11}{3} = 7$ s/j	$\frac{0+1+2}{3} = 1$ s/j

recall. convoy effect

Ex. same as above but SJF

$\frac{1+2+12}{3} = 5$ s/j $\frac{1+2+12}{3} = 5$ s/j

$\frac{0+1+2}{3} = 1$ s/j $\frac{0+1+2}{3} = 1$ s/j

recall. starvation

Date.
No.

Scheduling: FIFO vs RR:

FIFO	RR
• good turnaround time	• good response time
• sensitive to convoy effect	• requires preemption (context switching)
• try SJF (starvation)	• fair (shares CPU evenly)

Multi-Level Feedback Queue (MLFQ):

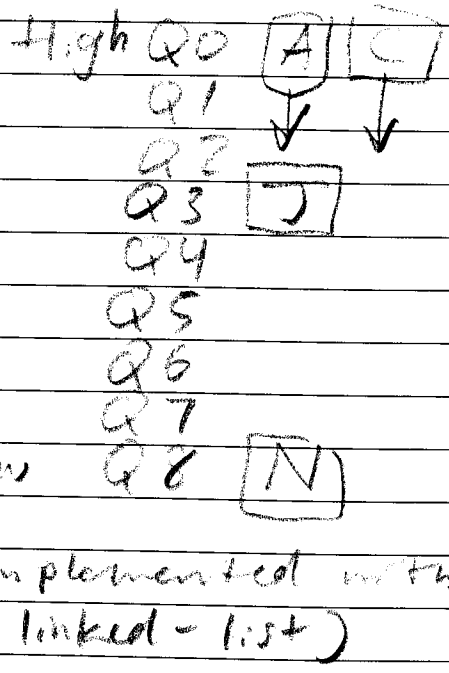
tries to optimize turnaround time and response time

- Like FIFO, it tries to complete SJF
- Like RR, it tries to be fair
- Unlike either, it will consider I/O and will adjust priority over time

MLFQ: Priority Levels

To accomplish this, MLFQ uses multiple queues, where each queue represents a priority level

- select jobs from the highest priority level
- within a level, we use round robin



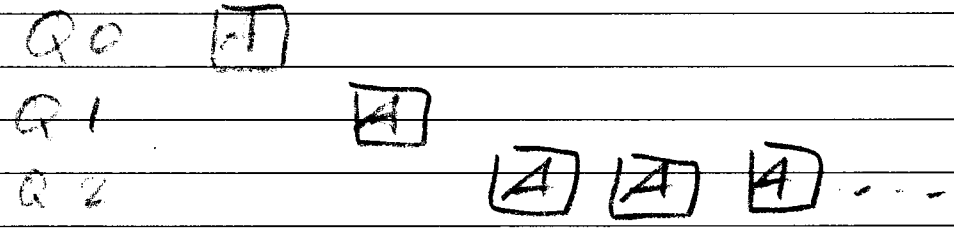
(implemented with a linked-list)

Date.
No.

MLFQ Rules:

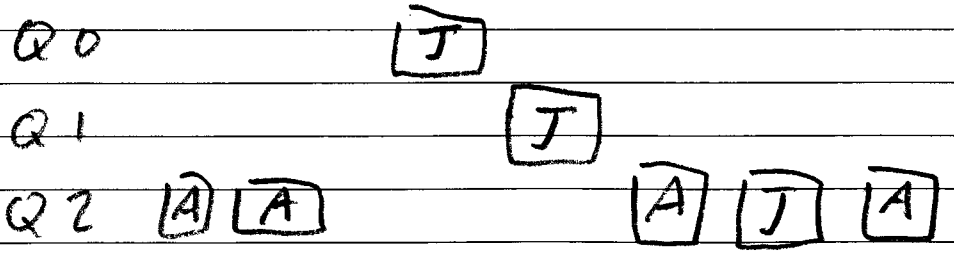
1. Priority(A) > Priority(B), then select A
2. Priority(A) = = Priority(B), use round robin
3. Once a job uses up time allotment, reduce priority
4. New jobs start in highest priority level
5. Periodically boost priority of all jobs

Ex. Single Long Job



- each job is broken up into discrete time slices
- over time, the job will reduce in priority to allow new jobs an opportunity to run

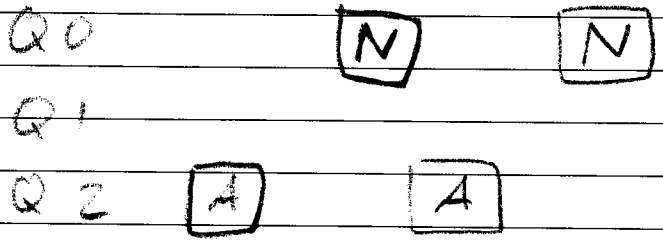
Ex. Long vs Short



- when a (short) job arrives, it will start in the highest priority level and be ran first
- this allows for (short) jobs have a fast response time and turnaround time

Date.
No.

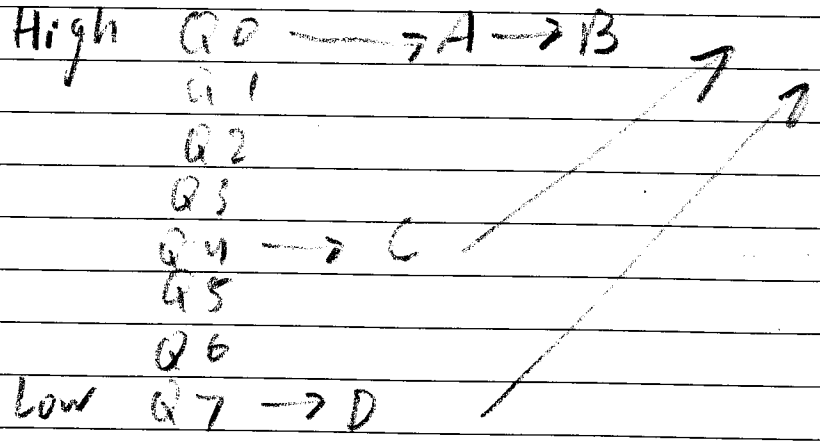
Ex. I/O vs CPU



- jobs that are mostly I/O will maintain a higher priority since they do not use up their time allotment as quickly
- this is good for interactive jobs which require good response time

MLFQ Priority Boost:

Problem: if a process is always in a lower priority level relative to other processes it will starve because it will have no opportunity to run



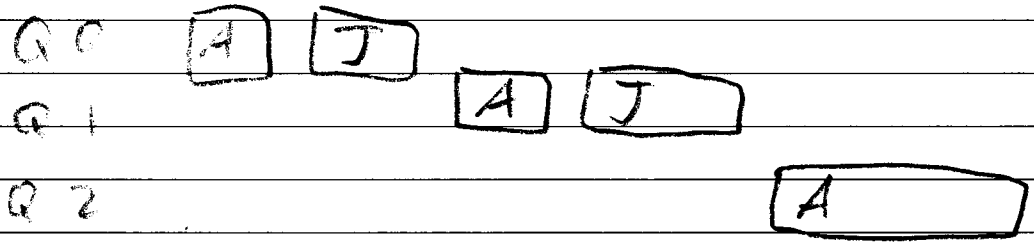
(these numbers are arbitrarily guessed for a system but can be changed to suit your needs)

Date.
No.

MLFQ Accounting:

to determine when a job should be moved down a queue (i.e. lower its priority), we keep track of how much of a time slice or allotment the job has used

- when a time allotment is used up, we lower its priority
- we allocate larger time slices for jobs in lower time slices



* this is stored in the kernel state of a process (in its struct) *

MLFQ Algorithm Pseudocode:

* in slides, too long to write down *

MLFQ Summary:

attempts to prioritize both turnaround time and response time

- prioritizes new, short, interactive over long CPU intensive
- devolves into round robin over time
- involves some accounting and magic numbers
- used in some older OS

* Linux uses CFS right now *

Date.
No.

Ex. Consider a single CPU, timeslice of 10ms, and three processes

Process	Arrival time	Run time
A	0 ms	30 ms
B	10 ms	20 ms
C	20 ms	10 ms

Waiting		B	B, C	C	C		
Running	A	A	A	B	B	C	
	0	10	20	30	40	50	60

* will be on the exam

Ex. Same scenario, but round robin

Waiting		A	C, B	B, A	A		
Running	A	B	A	C	B	A	
	0	10	20	30	40	50	60

* add to waiting, and THEN rotate (not shown)
→ the back of waiting

Ex. same situation, MLFQ

Q0	A	B	C				
Q1		A	A, B	A, B	A, B	B	
Running	A	B	C	A	A	B	
	0	10	20	30	40	50	60

Q0 has a timeslice of 10ms
Q1 has a timeslice of 20ms

Concurrency

Date. 9/16/25
No. 07

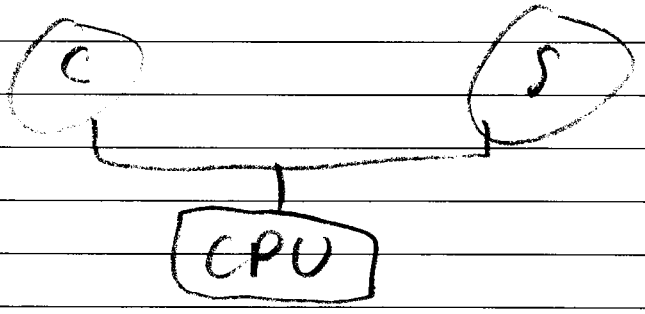
Motivation: Counter

we want to create an application, counter, that:
- periodically increments a counter
- provides a shell prompt to accept these commands

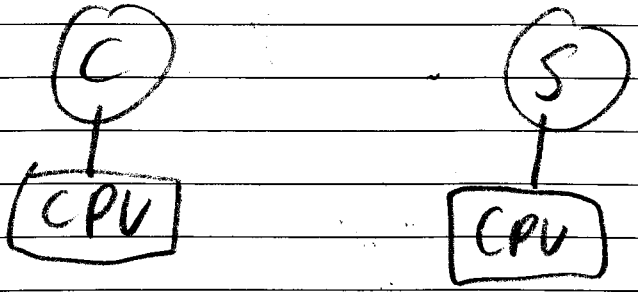
count -> returns current val of counter
reset -> resets counter
exit -> quits program

Concurrency: Overview

whenever we structure a problem such that we have multiple streams of execution, we have concurrency



whenever we have the hardware resources to simultaneously execute multiple streams of execution, we have parallelism



* if you structure your program concurrently, you get parallelism for free *

Date.
No.

Internal Concurrency:

Shell

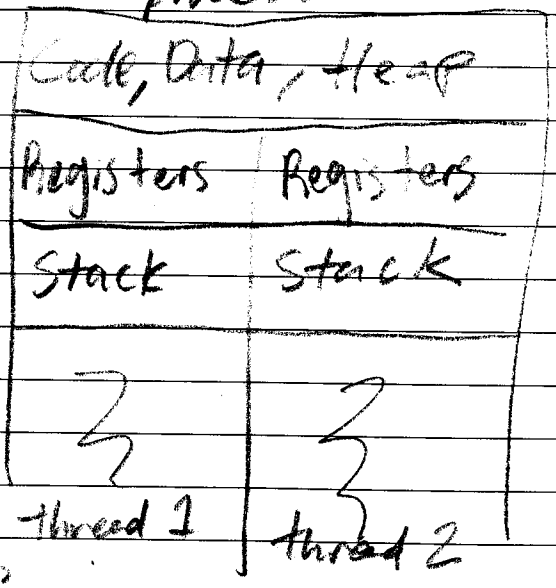
In addition to inter-process concurrency, we can also have concurrency within a single process:
1. Background vs Foreground
2. Mix I/O and Computation
3. Signals

Concurrent tasks:
1. Update counter
2. Handle shell input

Concurrency: Threads

divide a process into multiple streams of execution called threads

Recall. Machine State — shared?
1. Address Space ✓
2. Kernel State ✓
3. Execution Context ✗



Concurrency: Sharing Problem

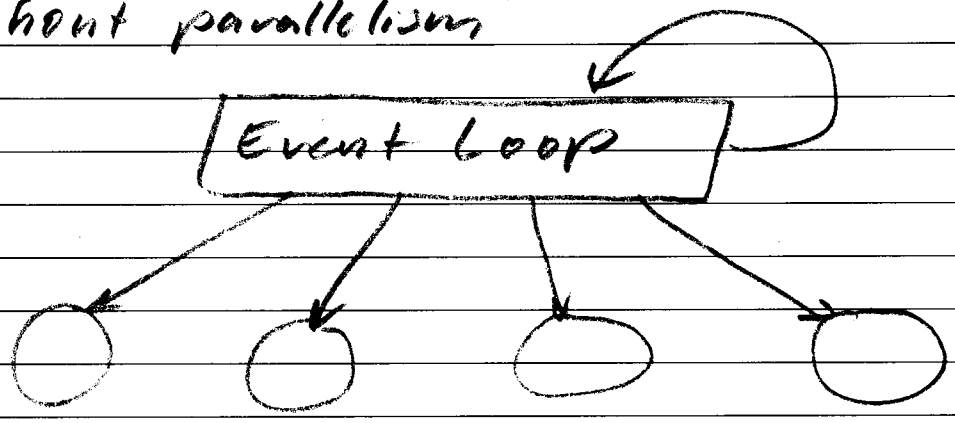
1. Shared Data → Critical Section
2. Uncontrolled Scheduling → Race Condition
3. Need for Atomicity → Mutual Exclusion

Events vs Threads

Date. 9.23.25
No. 09

Events: Overview

If we only need to overlap I/O and computation, we can use events to provide concurrency without parallelism



- register interest in events (callbacks)
- event loop waits for events, invokes callbacks (handlers)

* counter.c + DOS example in repo *

Questions: what are pros and cons of
- event-based concurrency with select/poll
- threads

	Pros	Cons
Events	• Concurrency with low overhead	• Denial of Service • Not structured to take advantage of multiple cores
Threads	• Can take advantage of extra hardware resources	• Complexity and overhead

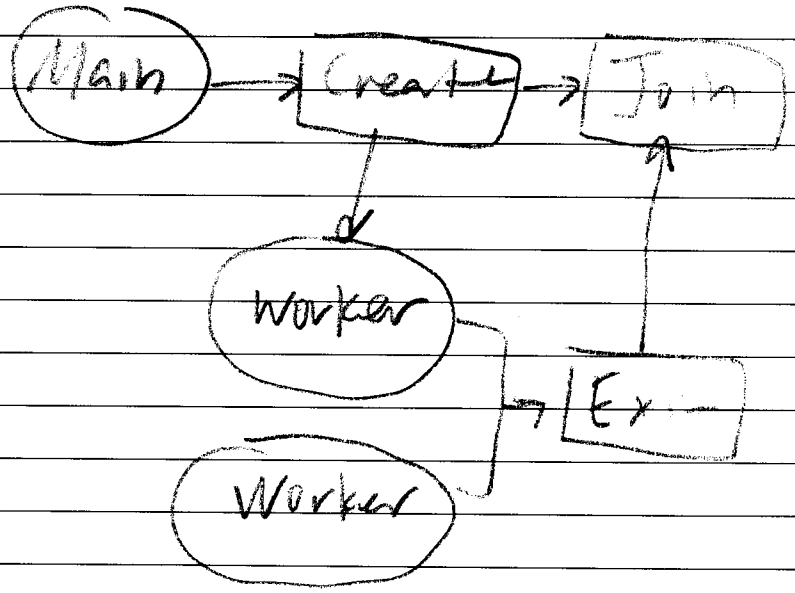
Date.
No.

POSIX Threads : P-threads

PThreads: Overview

on Unix systems, we can use the posix threads, which define functions for:

- creating a thread
- wait a thread
- notify a thread

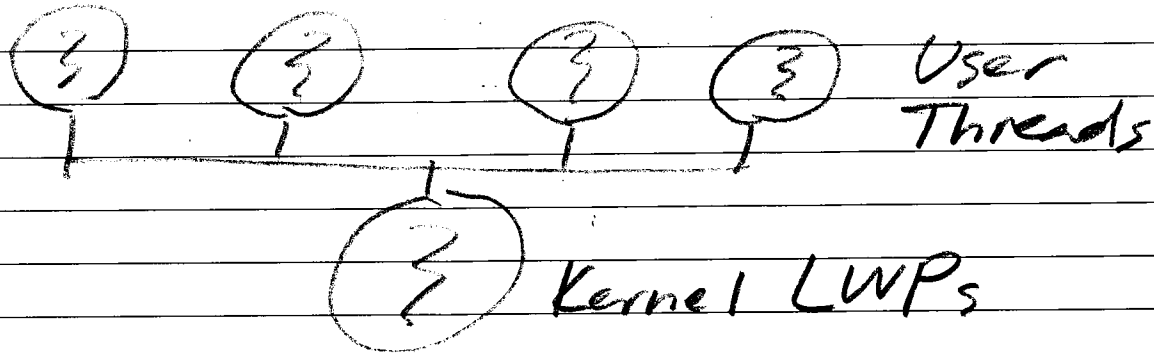


Pthread Analogies:

Action	Process	Thread
Create task	fork() + exec()	p-thread_create()
Wait for task	wait(), waitpid()	p-thread_join()
Lock critical section	sigprocmask(SIG_BLOCK...)	pthread_mutex_lock() pthread_mutex_unlock()
Notify another task	kill()	pthread_cond_wait() pthread_cond_signal()

Date.
No.

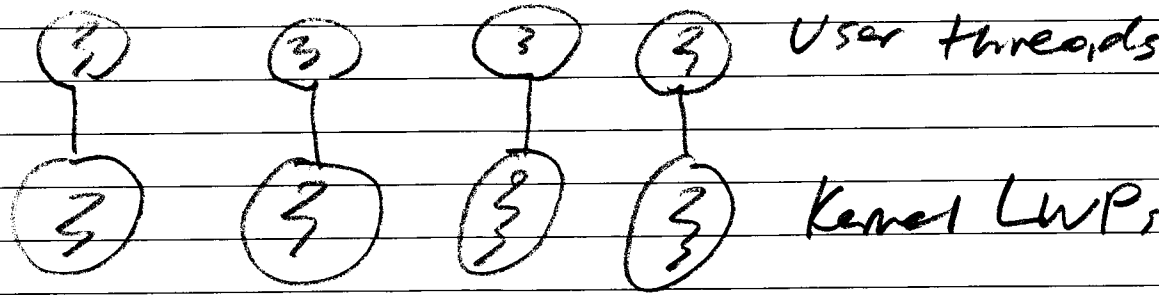
Pthreads: many to one



Pros:
+ portable, flexible
+ Straightforward to implement

Cons:
- No parallelism
- if one thread blocks, all threads block

P-threads: one to one

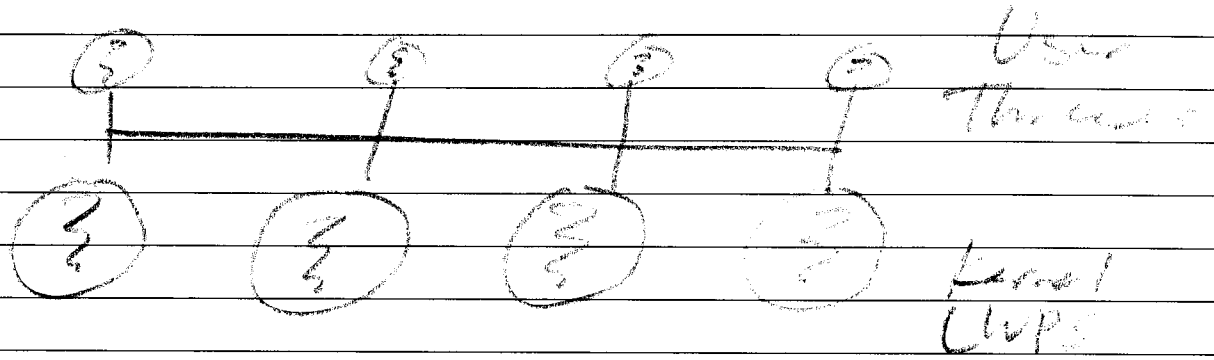


Pros:
+ takes advantage of hardware resources
+ One thread blocking does not prevent others from running

Cons:
- Limits to scalability (memory, kernel state)

Date.
No.

Pthreads: many to many



Pros:
+ scalability
+ custom scheduling

Cons:
- very complex

* Linux uses one-to-one at the kernel level, and M-to-M implemented at language level

Ex. counter.c with threads

* in class repo *

```
pthread_t background;
pthread_create(&background, NULL, callback, NULL);

void *callback(void *arg) {
...
return NULL;
```

Locks

Date. 9.25.25
No. 10

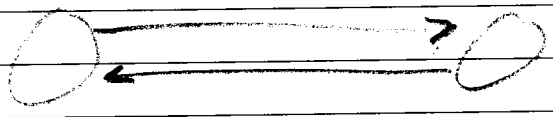
Overview:

Processes	Events	Threads
• abstraction (CPU)	• signals	• abstraction (execution context)
• address space X	• select/poll	• lightweight process (LWP)
kernel state X	• no parallelism	
execution context ✓		

* all three of these give concurrency
* only processes and threads give parallelism

Why use threads? because they share memory
-> processes do NOT

Processes have to communicate to share memory?



- pipes
- signals
- suffers from blocking
- lots of overhead

Test Q. By default every process has a t least one (main) thread

* Threads are just processes with multiple execution contexts

Date.
No.

Pthreads: Disassembly

Background Thread	Foreground Thread
Counter++	Counter = 0
lol r1, $Counter (1)	st i $counter, 0 (3)
addi r1, 1 (2)	
st $counter, r1 (4)	

* Context switching at the wrong time causes weird behavior — race conditions

Locks: Pthreads

use a lock or mutual exclusion to guard a critical section, which is a region of code that involves the shared resource

```
pthread_mutex_t lock;
pthread_mutex_init(&lock, NULL);

pthread_mutex_lock(&lock);
...
pthread_mutex_unlock(&lock);
```

* Counter and primes exs in repo w/ threads and locks

note. long-tail problem → solve with striping

Locks: Evaluation

to evaluate a lock implementation, we need to consider the following metrics

1.

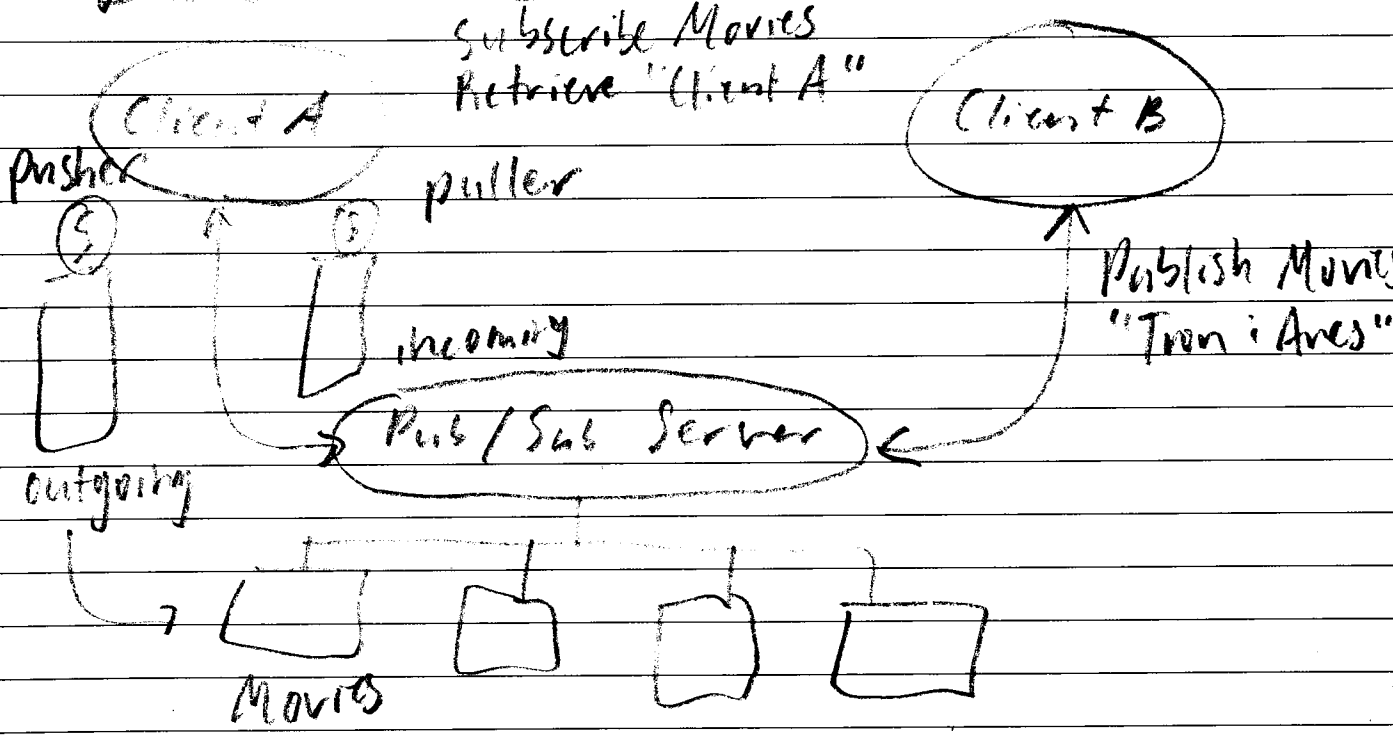
Condition Variables

Date. 9.30.25
No. 11

note. a process abstracts the CPU
a thread abstracts a specific part of the CPU (execution context)

locks solve race conditions but cause deadlock

Project 02: Pub/Sub



& sending and receiving from a server at the same time
→ 2 threads

Date.
No.

Concurrent DS: Producer/Consumer

Bounded Buffer

Producer

```
while True:
    data = produce()
    LOCK()
    while queue.full():
        pass

    queue.push(data)
    UNLOCK()
```

Consumer

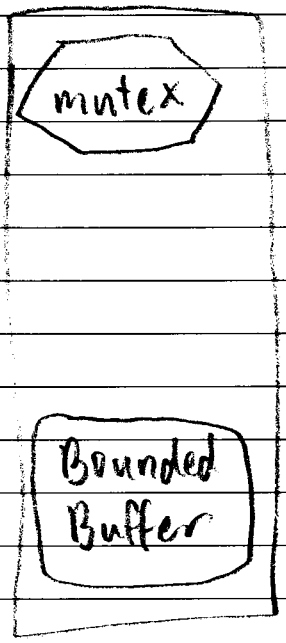
```
while True:
    LOCK()
    while queue.empty():
        pass

    data = queue.pop()
    UNLOCK()

    consume(data)
```

PROBLEM: this structure causes deadlock
ie. thread is waiting while holding onto the lock preventing other threads from satisfying the waiting condition

SOLUTION: Condition Variables

Date.
No.

Concurrent DS: Monitor




```
Push(data):
  Mutex.Lock()

  while queue.full():
    pass

  queue.push(data)
  Mutex.Unlock()
```

```
Pop():
  Mutex.Lock()

  while queue.empty():
    pass

  data = queue.pop()
  Mutex.Unlock()
  return data
```

* Allows us to abstract our Bounded Buffer to this *

Producer

```
while True:
  data = produce()
  queue.push(data)
```

Consumer

```
while True:
  data = queue.pop()
  consume(data)
```

* ex of this implemented in repo *

Date. 10.2.25
No. 12

Semaphores

Exam Q. To help avoid deadlock, we use condition variables

Pattern. when you abstract away locks/synchronization variables into class structures, this is called a monitor

Concurrency Patterns
- Producer/Consumer
- Monitor
- Thread Pool

Thread Pool: spawn N threads which get jobs from our monitor

Q. Which model would you use in a web server?
→ Thread Pool

Condition Variable: waiting

cond_wait (&cond, &lock);

1. Give up the lock
2. Go to sleep
...
3. Wake up
4. Take back lock

* while loop for cond_wait instead of if-statement because it's possible the condition the thread is waiting for has not changed after waking up

Date.
No.

Locks: Evaluation

to evaluate a lock implementation, we need to consider the following three metrics:

1. Correctness — does it actually provide mutual exclusion?
2. Fairness — does it give each thread a fair shot at acquiring the lock?
3. Performance — how much overhead is added by using the lock?

Locks: Disabling Interrupts

one way to implement locks is to simply disable interrupts

```
class Mutex:

    method Lock(self):
        DisableInterrupts()

    method Unlock(self):
        EnableInterrupts()
```

Problems:

1. Correctness
   Lose interrupts (IO)
2. Fairness:
   Need to trust processes
3. Performance
   Does not scale to multicore

Date.
No.

Locks: Implementation

A better way is to implement a spin lock:

```
class Mutex:
    method Init(self):
        self.flag = 0                    # 0 -> available, 1 -> taken

    method Lock(self):
        while self.flag == 1:            # Test Lock
            pass                         # Spin until available
        self.flag = 1                    # Set as Locked

    method Unlock(self):
        self.flag = 0
```

Problems:
1. Correctness (race conditions)
2. Performance (Spin / busy waiting)

Locks: Test and Set

To effectively implement a lock, we need special hardware instructions that provide atomic exchanges:

```
instruction TestAndSet(int *old_ptr, int new_val):
    old_val = *old_ptr          # fetch old_val at old_ptr
    *old_ptr = new_val          # Store new_val in old_ptr
    return old_value            #
```

All of the lines happen in 1 CPU instruction (atomic)

Date.
No.

With TestAndSet, we can implement a better spin lock:

```
class Mutex:
    :
    :
method Lock(self):
    while TestAndSet(self.flag, 1) == 1:
        pass
    :
    :
```

Test and Set Lock

Spin until we get a 0

Problems:
1. Performance (busy waiting)

Locks: Yielding

One way to reduce the cost of spin waiting is to simply yield as we spin:

```
class Mutex:
    :
    :
method Lock(self):
    while ...
        Yield()  # Give up CPU to the scheduler
    :
    :
```

Date.
No.

Ex. Stackbie (w/ stacker)

= Stackbie

= Global Vars

size_t CAPACITY = 3
size_t Slackers = 0
Mutex Mutex
Cond Cond
thread coach_person() -

= Threads

step_on()
go_across_stackbie()
step_off()

fnction step_on():

= Functions

Mutex.lock()

while Stackers >= CAPACITY:
Cond.wait()

Stackers++
Mutex.unlock()

fnction step_off():
Mutex.lock()
Stackers--
Cond.signal()
Mutex.unlock()

Date.
No.

Semaphors: Overview

A Semaphor is a synchronization primative that consists of an integer that we can manipulate with two operations:

```
sem_wait(S):
    S.value--
    if S.value < 0:
        thread_wait()

sem_post(S):
    S.value++
    if threads_waiting():
        thread_wakeup_one()
```

* Locks/condition variables work hand-in-hand, Semaphors are completely different and not related *

Ex. Starbucks (with Semaphors)

```
:
Semaphor S = CAPACITY

fuction step_on():
    sem_wait(S)

fuction step_off():
    sem_post(S)
```

Date. 10.7.25
No. 13

Semaphores (structures)

Semaphor : Lock

to use a semaphor as a lock :
- initialize the semaphor to 1
- perform a wait to aquire the lock
- perform a post to release the lock

Semaphor : Condition Variable

to use a semaphor as a condition variable :
- initialize the semaphor to 0
- perform a wait to wait on condition variable
- perform a post to signal condition variable

* A lot of using locks/condition variables will just easily semaphors as reading OS in repo *
- locks/CV → lock, and then CV
- Semaphores → CV, and then lock

note. Semaphor INT → number of things semaphor can do before sleeping
- sometimes needs to be set to something other than 0 or 1

* discussions for project in slides *

Test Q. project 2 is task parallel!
readings OS is data parallel!

Patterns

Date. 10.9.25
No. 14

Recall: Synchronization Primitives

	Problem	Approach
Lock	• Race Conditions	• Ownership
Cond Var	• Deadlock • Busy waiting	• Ownership
Semaphor	• All of the above	• Signaling

- think, like malloc()/free(), just tools!

Concurrency Patterns

- Scheduler: ordering
- Thread Pool: fixed # of threads
- Monitor: combine ds w/ synchronization primatives
- Rate Limiting
- Reader/Writer Lock

Exam Q. Given code, which concurrency pattern is this?

* Project02 overview in slides *

Semaphor: Producer/Consumer

- check condition semaphores **before** aquiring lock semaphor
- release lock semaphore **before** signaling conditions

Date.
No.

Semaphore : Reader_Writer Locks

with semaphores, we can create synchronization objects such as reader-writer locks

- once readers aquire the writelock, as many readers as possible can read the data
- once a writer aquires the writelock, no one else can access the data

Semaphore: Reader Preference

Cons : Starvation

```
size_t Readers = 0
sem_t WriterLock = 1
sem_t ReaderLock = 1

Writer()
    sem_wait(WriterLock)
    do_write()
    sem_post(WriterLock)

Reader()
    sem_wait(ReaderLock)
    if Readers == 0
        sem_wait(WriterLock)
    Readers++
    Sem_post(ReaderLock)

    do_read()

    Sem_wait(ReaderLock)
    Readers--
    if Readers == 0
        Sem_post(WriterLock)
    Sem_post(ReaderLock)
```

Date.
No.

Semaphore : Fair

```
size_t Readers = 0
sem_t WriterLock = 1
sem_t ReaderLock = 1
sem_t Queue = 1

Writer()
    sem_wait(&Queue)
    sem_wait(&WriterLock)
    sem_post(&Queue)

    do_write()
    sem_post(&WriterLock)

Reader()
    sem_wait(&Queue)
    sem_wait(&ReaderLock)
        ; same as in last ex
    sem_post(&ReaderLock)

    do_read()
        ; same as in last ex
```

Semaphore : Summary

- good when you need a barrier or want to have bounded access to a resource
- unlike locks and condition variables, there is no notion of ownership
- can avoid using locks and condition variables if desired

* Ex Exam Q's in slides *

Date. 10.14.25
No. 15

Concurrency Bugs

* Exam 2 [illegible] study guide [illegible]

Bug: Atomicity Violation: [illegible] [illegible]
[illegible]

Ex:

Thread 1	Thread 2
if buf_idx < BUF_SIZE memcpy(buf[buf_idx], ...)	buf_idx = [illegible]

OR

lock() [illegible] unlock()	lock() [illegible] unlock()

lock()
[illegible]
[illegible]

Bug: Order Violation

Ex:

Thread 1	Thread 2
mq.incoming = queue_create()	if mq.incoming [illegible] data = q_pop(mq.incoming)

OR

while True queue_pop(mq, req)	queue_delete(mq.incoming)

Date.
No.

Bug: Deadlock

Occurs when each thread is waiting for another thread to release a resource and no one can run. Usually happens because there is a cycle in the graph of dependencies

Ex.

Thread 1	Thread 2
Lock(Mask)	Lock(Sanitizer)
Lock(Sanitizer)	Lock(Mask)

Bug: Livelock

Occurs when multiple threads are actively attempting to acquire resources, but cannot continue progress

Thread 1	Thread 2

while True:
Lock(Mask)
if Locked(Sanitizer):
Unlock(Mask)
continue
Lock(Sanitizer)
	while True:
	Lock(Sanitizer)
	if Locked(Mask):
	Unlock(Sanitizer)
	continue
	Lock(Mask)

Date.
No.

Address Spaces

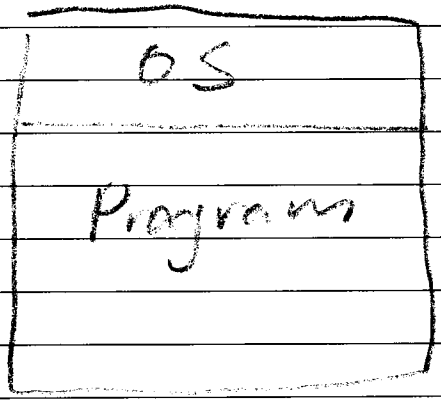
Date. 10.28.25
No. 17

Review : OS Themes

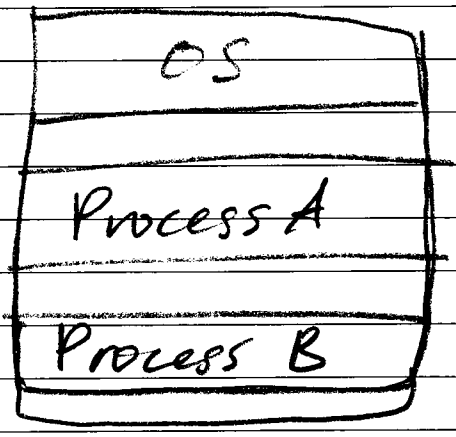
1. Virtualization : Processes, Threads (CPU)
2. Concurrency : Synchronization
   (Race condition, Deadlock, Busy waiting)
3. Persistence : File System

Virtual Memory : Motivation

Batch Computing



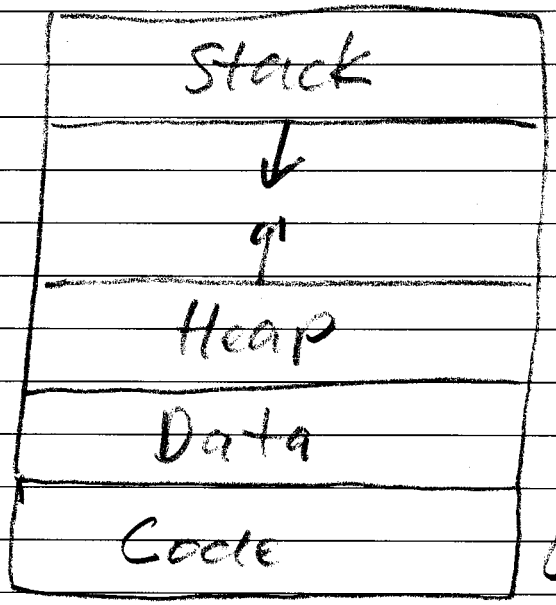
Interactive Computing



with the development of multiprogramming and timesharing, we need a way to protect programs from each other and to effectively Switch between them

Date.
No.

Virtual Memory: Address Space



Stack: local variables, function parameters
Heap: user managed memory
Data: static, global, string literals
Code: program instructions

note. stack/heap addresses change over executions
→ Security: ASLR
→ data/heap may also change organization

Virtual Memory: Challenges

1. Protection: prevent one process from interfering with another
2. Ownership: each process believes it has sole control of all of memory (infinite)

Date.
No.

Address Translation: Overview

each address in our program is actually a virtual address that must be translated into a physical address

This virtualization allows us to relocate a process to any location in memory

Goals:
1. Transparency: invisible to application
2. Efficiency: as little overhead as possible
3. Protection: protect process from each other

Address Translation: Relocation

Static: one way to achieve this relocation is to modify your application whenever you need to load it into memory

- Expensive (upfront)
- No protection
- Hard to relocate

Dynamic: A better approach is to use a special hardware (mmu) features to perform demand translation

- Pay cost as you go
- Real protection
- Update on the fly

Date.
No.

Address Translation: Base & Bounds

- hardware-based translation method

CPU has a **base register** which holds the starting point in physical memory, and a **bounds register** which holds the ending point in physical memory

A **physical** address can be generated with the following:

physical address = base + virtual address

TEST Q. There are multiple base & bounds in an OS ... T

Address Translation: Ex

(Bounds)

Suppose we had an address space of 6KB that has been located at address 32KB (Base)

Virtual Addr	Physical Addr
0KB	32K
1KB	33K
5000	~37K
9000	Out of bounds

note. $K = 2^{10} = 1024$
$M = 2^{20} \sim$ million
$G = 2^{30} \sim$ billion
$T = 2^{40} \sim$ trillion

Memory API

Date. 10.30.25
No. 18

Memory API : Allocation

How do we allocate an array of 1,000,000 ints?

```
int *a = malloc(1_000_000 * sizeof(int));
int *b = calloc(1_000_000, sizeof(int));
```

Memory API : Deallocation

How do we deallocate an array of ints?

```
free(a)
```

Memory API : Common Errors

What is the memory error?

```
char *buffer;  // char buffer[BUFSIZ]
fgets(buffer, BUFSIZ, stdin);
```

forgot to allocate memory → segfault

What is the memory error?

```
char *src = "hey";  // strdup(src)
char *dst = malloc(strlen(src));
strcpy(dst, src);
```

did not allocate enough memory → buffer overflow

Date.
No.

What is the memory error?

```
char *s = malloc(100 * sizeof(char));
puts(s);
```

use calloc

did not initialize memory → uninitialized read

What is the memory error?

```
for (int i=1; i<argc; i++) {
    char *s = strdup(argv[i]);
    process(s);
}
```

forgot to free memory → memory leak

What is the memory error?

```
for (...) {
    char *s = strdup(argv[i]);
    process(s); free(s); puts(s);
}
```

freed before we are done → dangling pointer
→ use after free

Ex F.

```
for (...) {
    char *s = strdup(...);
    if (checks) { free(s); s
    free(s);
}
```

try to free something more than once → double free

Date.
No.

Ex 6.

```
for (...) {
    char *s = argv[i];
    process(s);
    free(s);
}
```

Try to free something that isn't on the heap → invalid free

Note. When a process is terminated, its address space is deallocated

So in short-running programs (compilers), you can get away with not managing memory (not free-ing anything)

Two levels of memory management!

1. at the OS level
2. at the process level

Review. Virtual Memory

Address Space	Address Translation
1. Protection	• MMU
2. Exception / Fault	• Base and Bounds

Physical Address = Base + Virtual Address

Date.
No.

Review OS responsibilities

Memory accounting → track allocations (used / free list)

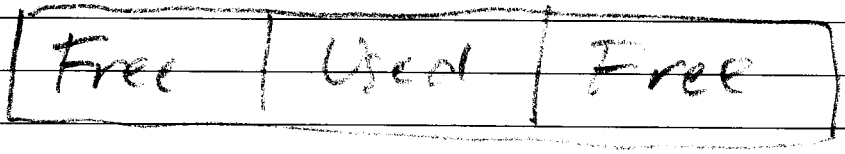
Hardware Management → update the MMU with base and bounds (context switch)

Exception Handling → register trap handlers and handle exceptions (not valid access)

Free-Space Management:

Challenge: External Fragmentation

Because memory is divided into blocks, we may have non-contiguous free space



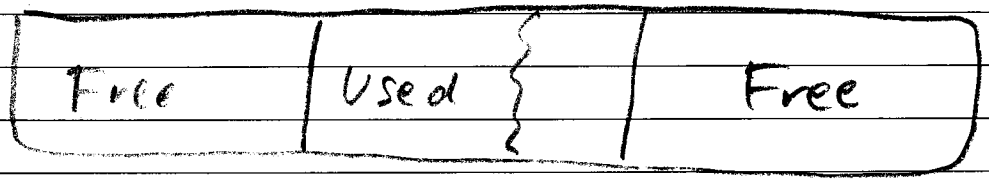
Not enough contiguous space in a single block although we have enough free space overall

$$\text{External Fragmentation} = 1 - \left(\frac{\text{largest free block}}{\text{all free space}}\right)$$

Date.
No.

Challenge: Internal Fragmentation

Suppose we over allocate a block

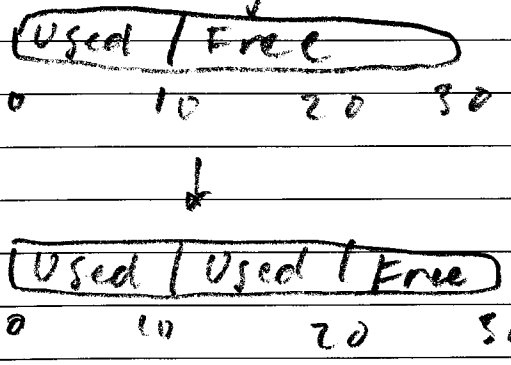


We waste space due to over allocation (word alignment)

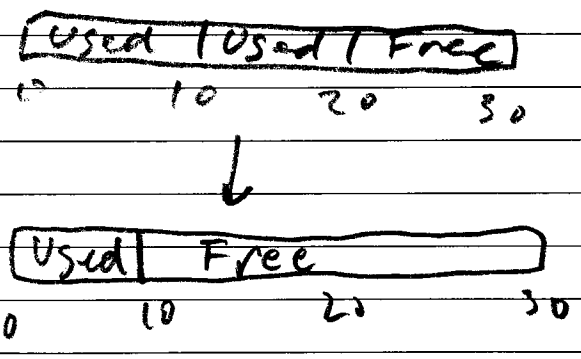
$$\text{Internal Fragmentation} = \frac{\text{All Used}}{\text{Heap Size}}$$

Mechanisms: Splitting & Coalescing

Splitting: malloc



Coalescing: free



Reduce internal fragmentation

Reduce external fragmentation

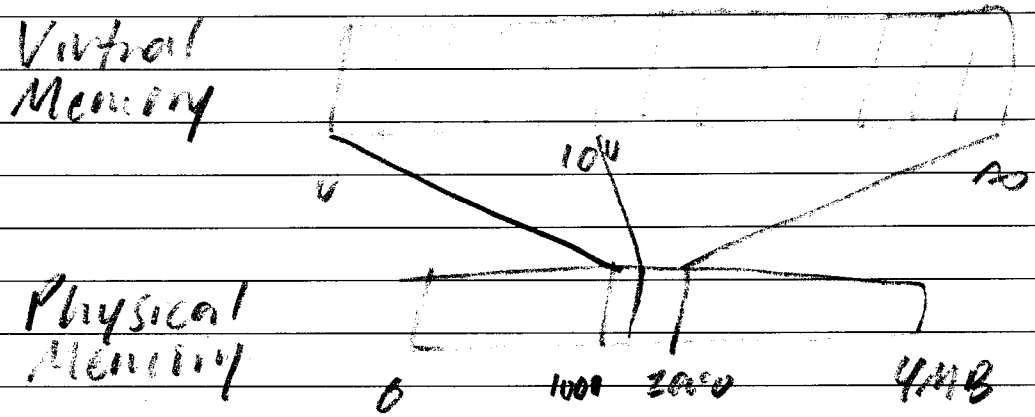
Free List: to track all of the block allocations, we use a linked list of the space reservations

- OS must track system memory allocations
- Individual processes must track heap allocations
- Must be wary of fragmentations

Date. 11/4/25
No. 19

Heap Allocation

Recall. Address Space : Abstraction of Memory

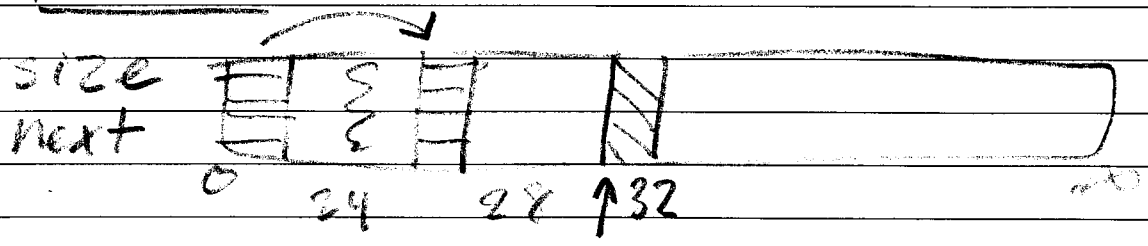


Address Translation (MMU)

1. Base and Bounds
2. Segmentation
3. Paging

CS → MMU → Base & Bounds

Free List : Linked List



capacity: how much memory allocated in block
size: amount of memory actually need

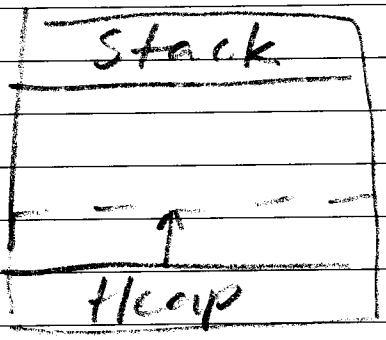
Test Q. Is malloc() a system call?

NO, we are writing it!

Date.
No.

Heap Allocation: System Call

to request memory from the kernel, we can use the sbrk system call



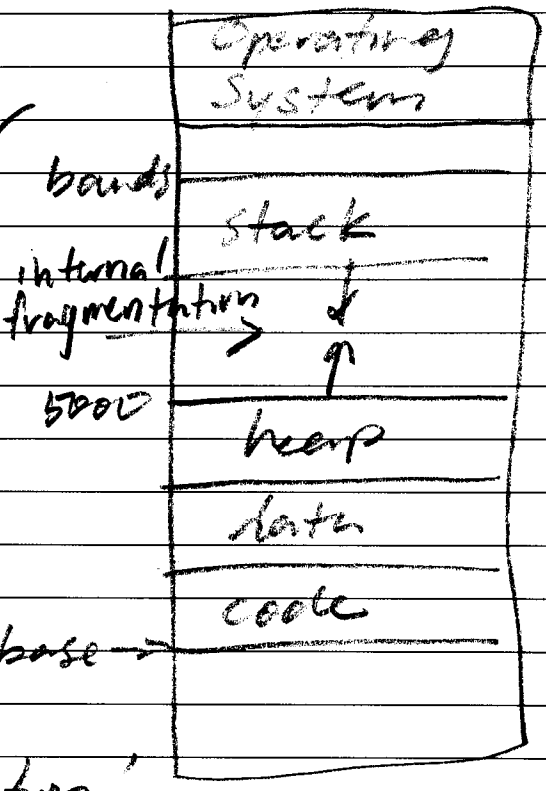
* walk-through of project and modified implementation in lecture recording *

Date. 11/6/25
No. 20

Segmentation

Virtualization: Lies

1. Infinite Memory ✗
2. Dedicated ownership ✓
3. Contiguous ✓



to implement these lies, we need to perform address translation

to perform translation we rely on MMU (hardware)

base and bounds: causes internal fragmentation.

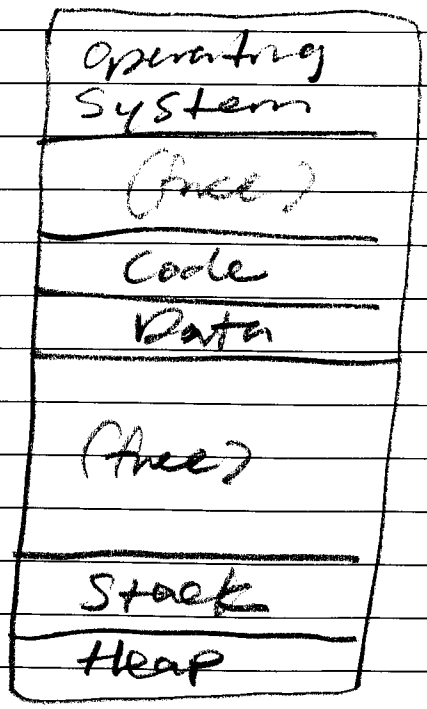
Segmentation: Overview

segmentation is a generalized form of base and bounds

each address component is considered a seperate segment

each segment has its own base and bounds

OS needs to tell MMU the base and bounds of each segment during a context switch



Date.
No.

Segmentation: Segments

because our address space is split into separate segments we have new issues and opportunities

Code Sharing: can share common code segments (multiple instances of the same program, shared libraries)

Segmentation: Hardware Information

to support segmentation, the MMU now needs to know the following info:

Segment	Base	Size	Grows Positive?	Protection
00 Code	32K	2K	1	rx
01 Data	34K	2K	1	rw
10 Heap	36K	2K	1	rw
11 Stack	28K	2K	0	rw

OS must track this for every process and save/restore on each context switch

Date.
No.

Segmentation: Addressing

Given this 16-bit virtual address:

1000 0010 0000 1010

segment | offset

512 + 8 + 2 = 522

1. What's the largest possible segment size?

$2^{14} = 2^4 \cdot 2^{10} = 16K$

2. What's the maximum possible offset value?

$2^{14} - 1 = 0011\ 1111\ 1111\ 1111$
$= 0x3FFF$

3. Which segment is this virtual address in?

heap

4. What is the physical address of this virtual address?

PA = 16K + 522

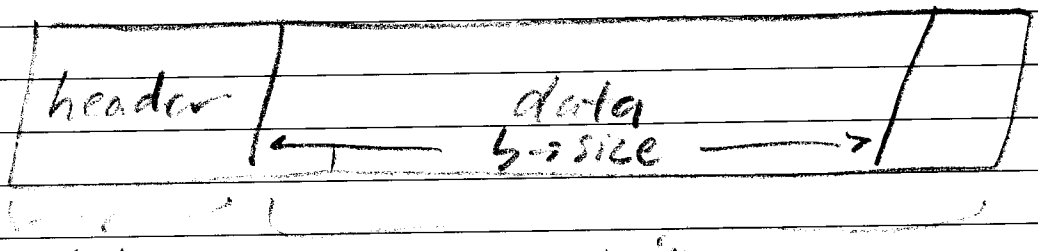
Paging

Date. 11.11.25
No. 21

Ex. void *p = malloc(100)

(104)

smallest multiple of 8 that fits 100



104

b->capacity

Block *b = header

b->data = data

intptr_t end = b->data + b->capacity

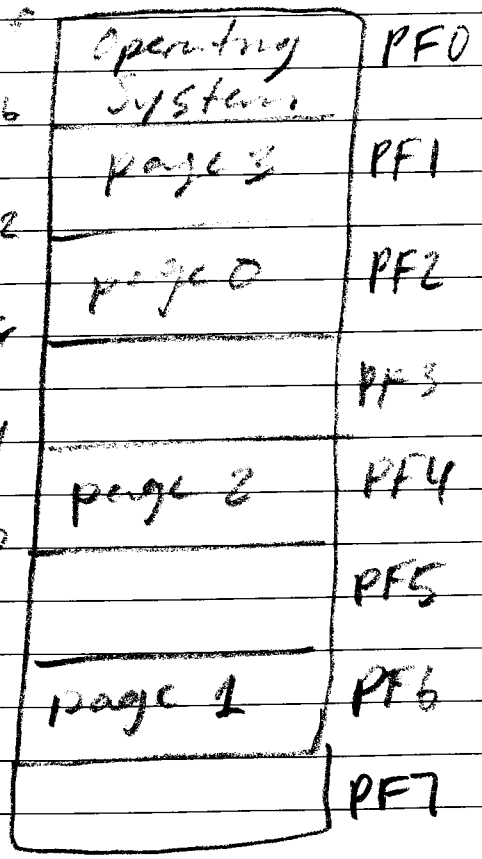
= (intptr_t)b + sizeof(Block) + b->capacity

Paging Overview

Rather than divide up memory into variable sized pieces, we divide it into fixed sized units called pages

Each process has an address space that consists of fixed size virtual pages

Each virtual page maps to a physical page frame



if array where

1 -> 6

2 -> 4

0 -> 2

3 -> 1

Date.
No.

Paging: Hardware information

Each process has a page table that stores the address translations for each virtual page to the corresponding physical page frame

Bit	Information
Valid	Is this translation valid?
Protection	Can we read/write/execute?
Present	Is it actually in RAM?
Reference	Has page been accessed recently?

In addition to the translation, each page table entry (PTE) may contain the bits of information above

Paging: Address Translation

Given the virtual address 56: 1 0 0 0 1 0 0

VPN | offset

1) Which page is this address located?

2

2. Which page frame is this address located?

5

3. What is the offset of this address?

4

4. What is the physical address?

PA = 64 + 4 = 68

Date.
No.

Paging: Memory Usage

Suppose we had a 32-bit machine with 4KB pages

1. How many pages can a process have?

$$\text{Number of Pages} = \text{Number of Virtual Addresses} / \text{Page Size}$$

$$= 2^{32} / 2^{12} = 2^{20} = 1\text{MB}$$

2. How many bits do we need for the VPN?

20

3. How many bits do we need for the offset?

12

4. How large is the page table (assuming each PTE is 4 bytes)?

$$\text{Page Table Size} = \text{Number of Pages} \cdot \text{PTE Size}$$

$$= 2^{20} \cdot 2^{2} = 2^{22} = 4\text{MB}$$

Paging: Problems

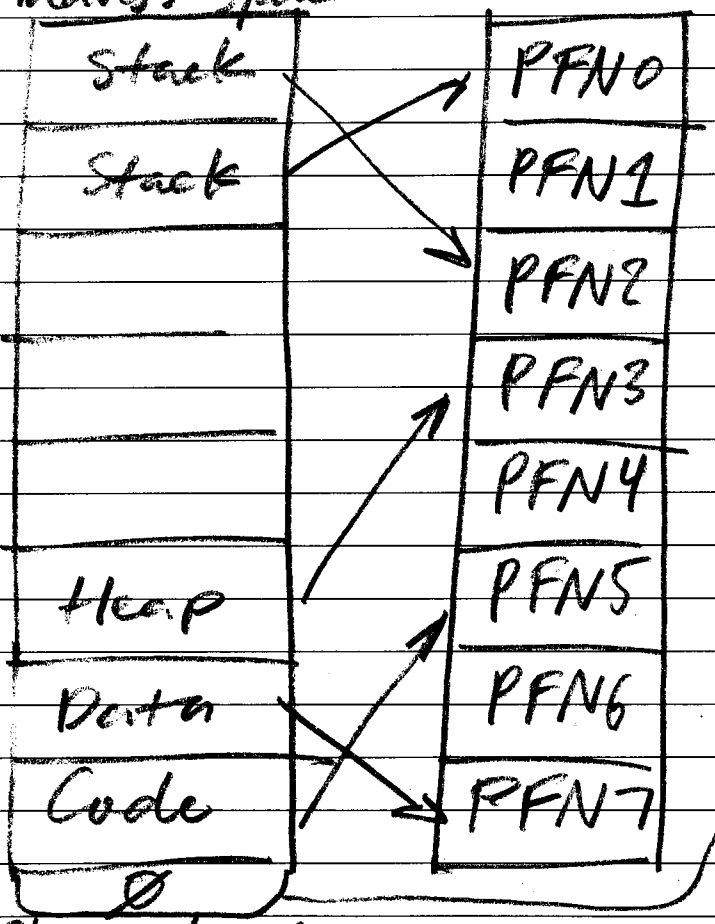
While better than segmentation when it comes to external fragmentation, paging suffers from the following problems:

1. Overhead: high memory usage per page table
2. Indirection: Each memory access requires us to lookup the PTE (in memory), we need 2 physical memory accesses per virtual address

Date. 11/13/25
No. 22

TLB / Multi-Level PT

Paging Architecture:



Page Table

VPN	PFN	Valid	Prot
0	2	1	rw
1	0	1	rw
2	—	0	—
3	—	0	—
4	—	0	—
5	3	1	rw
6	7	1	rw
7	5	1	rx
Ø	—	Ø	—

Observations:
- one page table per process
- page table stored in physical memory

Problems:
1. Each virtual address translation requires 2 physical memory accesses
2. page table requires a lot of physical memory

note. int *p = NULL
*p++ // SEGFAULTS

Date.
No.

TLBs: Faster Memory Access

Problem. each VM access requires two fetches from physical memory

- read PTE
- read memory

Solution. add a cache called the translation-lookaside buffer (TLB) to the MMU and use it to store PTEs:

TLBs: Hardware Information

The MMU uses a fully-associative cache and usually stores entries:

VPN → PFN | Other Bits

where other bits can be valid bits, protection bits, dirty bits, address space identifier, etc.

TLBs: Locality

spacial locality: if we access memory at address x, we will access memory near x in the future

temporal locality: if we access memory we will probably access it again soon

Date.
No.

TLB: Context Switch

once again, on each context switch, the OS must either:

1. Flush the TLB to mark all entries as invalid

OR

2. Set ASID register to the current PID

TLBs: Handling Misses

Hardware: on miss, hardware walks the page table, finds PTE, and updates the TLB with the translation
→ less flexible, hard to resume

Software: on miss, hardware raises an exception, and triggers a trap handler. OS then looks up PTE and updates TLB
→ infinite TLB misses

Date.
No.

Multi-Level Page Tables:

PFN	V	P
12	1	vx
13	1	vx
-	0	-
-	0	-
-	0	-
-	0	-
-	0	-
-	0	-
-	0	-
-	0	-
-	0	-
-	0	-
100	1	rw
86	1	rw
15	1	vx

page directory:

NULL
NULL

instead of an array to map virtual pages to physical page frames, we can use a tree instead

in this ex, save 1 page of memory, but now need 3 fetches to get memory

Ex. ON EXAM

14-bit VA — 64B page size

16-bit PA — 4B PTE / PDE

not needed

1. How many bytes can we address?

$2^{14}B = 16KB$

2. How many pages do we need?

$2^{14}B \times \frac{1 \text{ page}}{2^6 B} = 2^8 \text{ pages} = 256$

3. How many bits in VPN?

8

4. How many bits in offset?

6

Date.
No.

5. How much memory does page table require?

2^8 PTEs * $\frac{2^2 B}{1 PTE}$ = $2^{10} B$ = 1 KB

6. How many pages does page table require?

$2^{10} B$ * $\frac{1 page}{2^6 B}$ = 2^4 pages = 16 pages

7. How many PTE per page?

$2^6 B$ page size / $2^2 B$ / PTE = 2^4 = 16 PTE/page

8. How many PDEs in page directory?

16 (2^4) PDEs to keep track of the 16 pages the page table requires

9. How many bits in index of page directory?

4 bits

10. How many bits in index of page table?

4 bits (since VPN is 8-bits)

11. How much memory does page directory require?

2^4 PDEs * $2^2 B$ / PDE = $2^6 B$ = 64 B

12. How many pages does the page directory require?

1 (add more levels if this is not 1)

Multi-level: Summary

Pros: reduced memory usage

Cons: complex, more memory accesses

Swapping

Date. 11/18/25
No. 23

Memory Hierarchy

Fast	registers	10s	Expensive
	cache	MBs	(limited, small)
	RAM	GBs	
	hard drives	TBs	
Slow	cloud	∞	Cheap

Swapping: Overview

The swap space is a portion of disk reserved for moving pages back and forth:

- used to hold pages of currently running processes
- Present bit in PTE tracks whether page is in memory or not

Swapping: Page Replacement

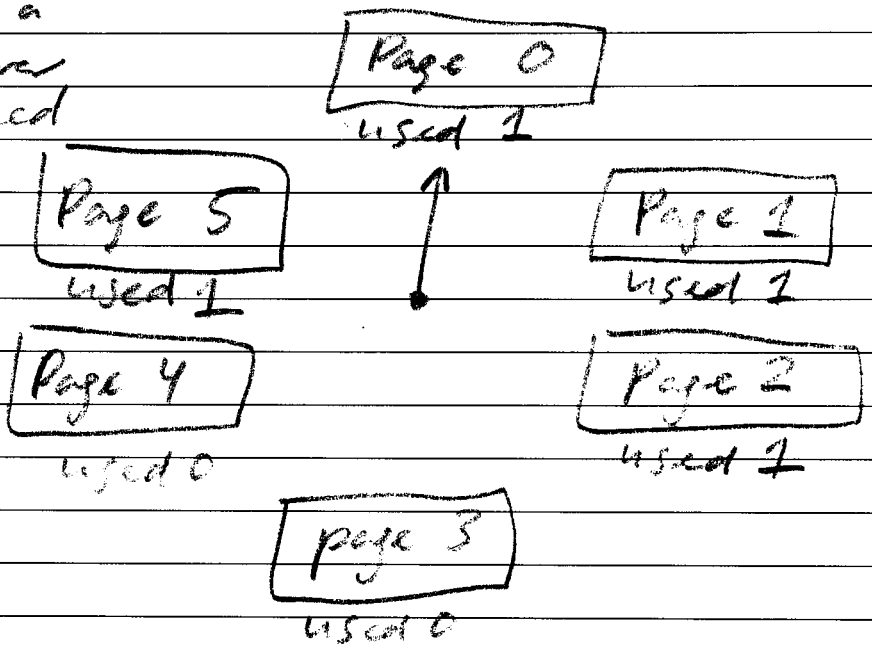
Ideally, we would minimize the amount of misses, thus want to evict pages that will be accessed farthest in the future:

- FIFO: poor performance since it doesn't consider access patterns
- Random: sometimes great, sometimes bad, mostly mid
- LRU: mostly optimal, but expensive

Date.
No.

Swapping: Clock Algorithm

- The hardware sets a used bit whenever a page is accessed
- OS arranges pages in a circular list
- A clock hand points to the current page



When replacement occurs, OS checks used bit of current page:

1. If 1, clear it and go to next
2. If 0, evict

Swapping: Page Selection

To decide when to bring a page into memory, the OS can deploy the strategies:

1. Demand Paging: bring pages into memory when they are accessed
2. Prefetching: bring pages into memory ahead of time

Date.
No.

Swapping: Danger

Suppose the amount of memory requested by running processes exceeds the available physical memory...

Evict page
Page in
Evict page
Page in
Evict page
...

} This condition of constantly swapping is called thrashing

Swapping: Laziness

Demand zeroing: rather than automatically allocating a page and zeroing it, the OS waits until the page is actually accessed

Copy-on-Write: rather than copying a page from one address space to another, share the page in read-only mode, only perform the copy if a write occurs

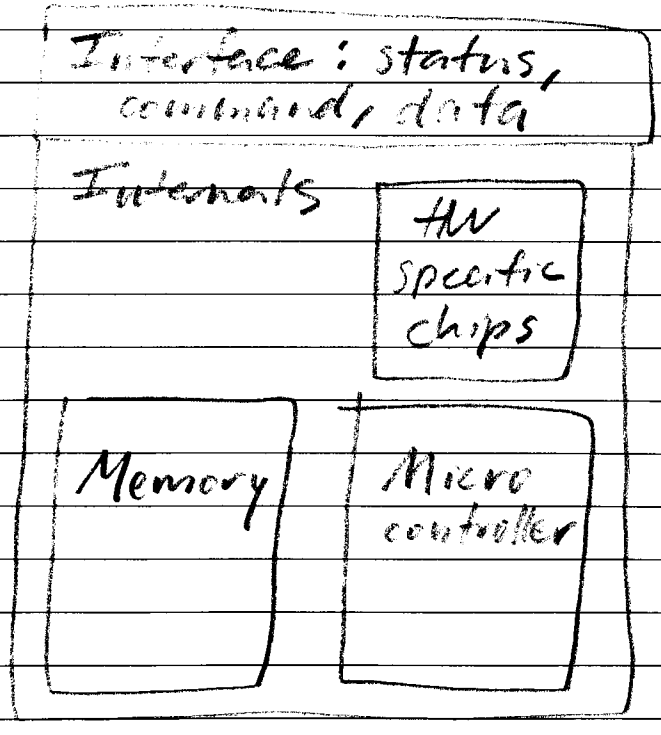
Date. 11.25.25
No. 25

I/O Devices, RAID

Recall. OS Themes

1. Virtualization: Abstract hardware resources
   - Process (threads) → CPU
   - Address Space → RAM
   - File System → Disk
2. Concurrency: Share resources safely
3. Persistence: Ensure data continues to exist

I/O Devices: Devices



while (STATUS == BUSY)

Write (DATA, data)
Write (COMMAND, command)

while (STATUS == BUSY)

Date.
No.

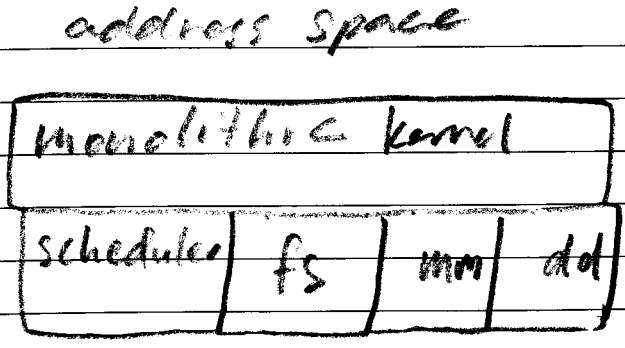
I/O Devices : Protocol

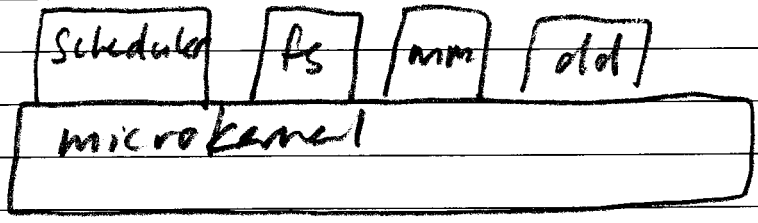
There are three methods to transfer data from memory to and from disk storage

Method	Descriptions	Pros	Cons
Programmed I/O	OS is directly involved in data movement	Simple and works	Wastes CPU time (busy waiting)
Interrupts	OS issues command and later handles interrupt	Allow for overlapping	can be wasteful due to context switch or interrupt storm
Direct Memory Access	OS programs DMA engine to handle data transfers	Allow for overlapping	Need more hardware and synchronization

I/O Devices : Kernel Space

Same address space





lots of context switches

note. fs → filesystem
mm → memory management
dd → device drivers

Date.
No.

I/O Devices : Ch. 11

We can create device drivers as a loadable Kernel Module... unmountable doesn't mean non-modular!

Hard Disks : Geometry

Platter: magnetic surface on which data is stored
Spindle: platters on Spindle together around motor component that has a motor
Tracks: data is encoded in these concentric circles
Disk head: where exactly data is read from
Disk Arm: holds the disk head

Hard Disks : Operations

Since platters are constantly spinning, to read/write data we need to move the disk arm so that the disk head is in the right location

- we must account for **rotational delay** (single-track latency)
- **seek time**: moving disk head from one track to another
- **settling time**: time required to guarantee correct location

Hard Drives : Seeking

acceleration: disk arm starts moving
coasting: disk arm moving full speed
deceleration: disk arm slows down
settling: disk arm positioned over track

Date.
No.

Hard Disks: Scheduling

As with processes, OS can schedule when to perform I/O operations:

- Shortest Seek Time First (FIFO)
  - → starvation
- Elevator (KDKN)
  - → improves rotation and access patterns
- Shortest Positioning Time First
  - → difficult to implement
  - → factors in both seek and rotational time

SSDs: Overview

neither flash nor slow spinning platters, instead use to it non-volatile NAND memory

+ Faster	- More expensive
+ Energy efficient	- Unknown reliability

SSDs: Garbage Collection

rather than overwrite data, we always write new pages of data and periodically collect any garbage

SSDs: Wear Leveling

since each individual memory cell has a limited number of write cycles, we must avoid repeated writing to the same area

- provides a logical block abstr to OS and associates these LBAs to physical blocks (virtual memory!)

Date.
No.

Hard Drive Reliability:

- can fail
- are slow
- have limited capacity

RAID: Overview

In a RAID system we combine multiple disks into a single array that appears as a virtual disk

1. Performance: faster
2. Capacity: bigger
3. Reliability: tolerate failure

RAID: Striping (0)

We can organize data blocks into larger called "chunks" and spread across multiple disks

	Disk 0	Disk 1	Disk 2	Disk 3
"chunk"	0	2	4	6
	1	3	5	7
	8	10	12	14
	9	11	13	15

Performance: I/O in parallel
Capacity: 100%
Reliability: none!

Date.
No.

RAID: Mirroring (1)

We can copy or mirror a block across multiple disks to add redundancy

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Reliability: lose up to half of disk
Performance: read in parallel, 2x write
Capacity: 50%.

Date. 12.02.25
No. 26

File Systems

RAID: Parity (5)

Rather than storing a copy of data, we can store a parity block which serves as an error detection code:

C0	C1	C2	C3	Parity
0	0	1	1	XOR(0,0,1,1) = 0
0	1	0	0	XOR(0,1,0,0) = 1

RAID 5: Parity Blocks

RAID 5 requires at least 3 disks, and rotates the parity across each disk in the array

RAID 5

Block	A_1	A_2	A_3	A_P
Block	B_1	B_2	B_P	B_3
Block	C_1	C_P	C_2	C_3
Block	D_P	D_1	D_2	D_3

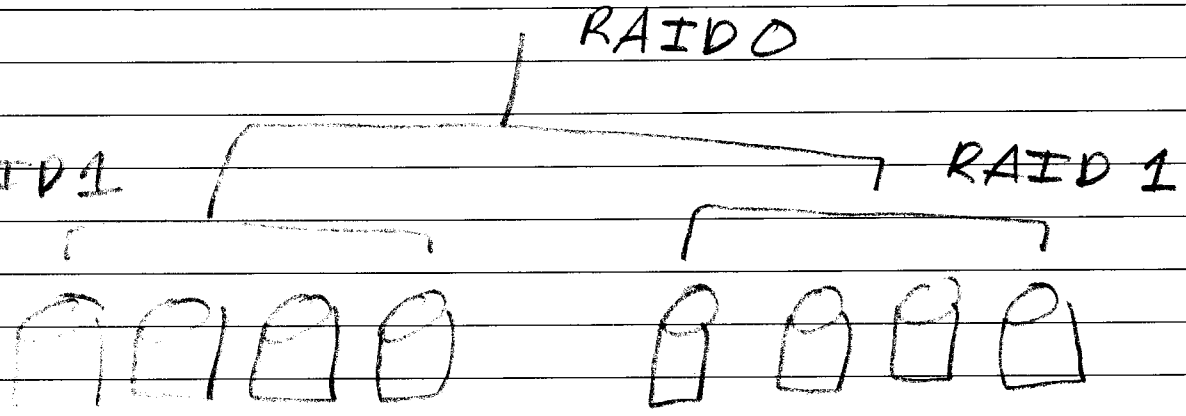
Capacity: $N - 1/N$

Reliability: Can tolerate losing one disk

Performance: read in parallel
writes multiple disks

Date.
No.

RAID 10 : (Mirroring + Striping)



note: with RAID 1 + RAID 0, we can only tolerate losing one disk

with RAID 0 + RAID 1, we can tolerate losing up to half the disks per array

RAID: Warnings

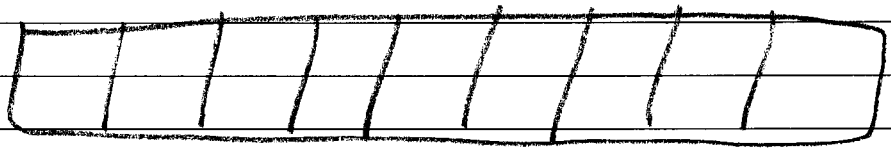
while RAID is a great way to build faster, bigger, and more reliable storage systems, it doesn't address every problem:

- RAID recovery time is atrocious
- RAID doesn't handle disk corruption (doesn't even detect it)
- RAID is not a backup system

Date.
No.

File System: Overview

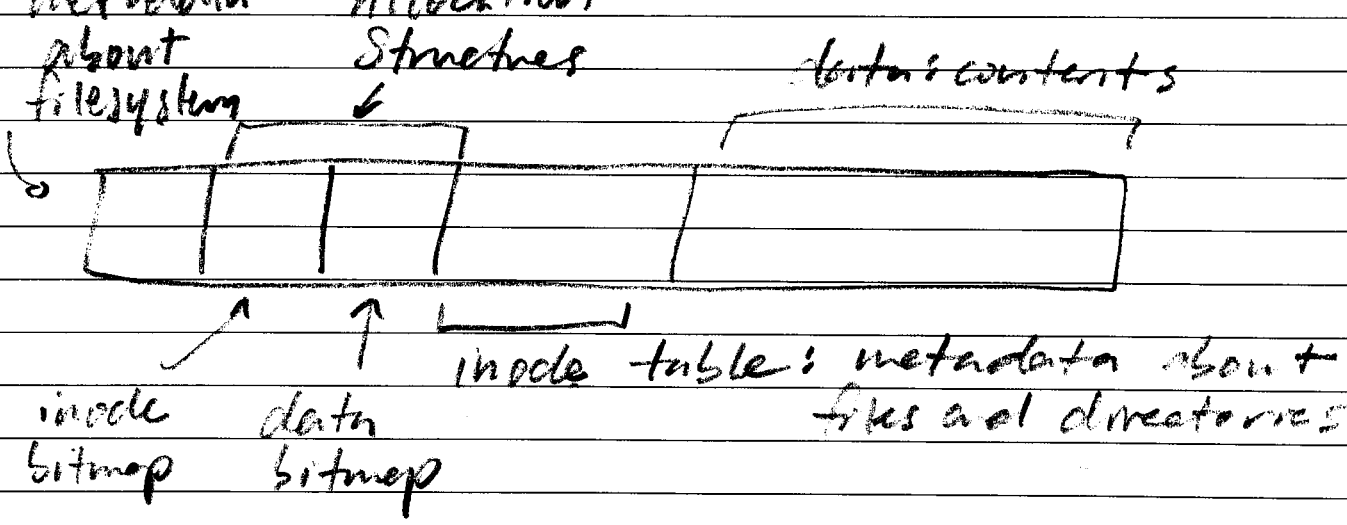
Use of data structures to organize data on disk



- Divide disk into fixed-size blocks (usually 4KB)
- Use portion of disk to store metadata
- Use another portion to store allocation structures
- Use initial portion to store filesystem information

File System: Organization

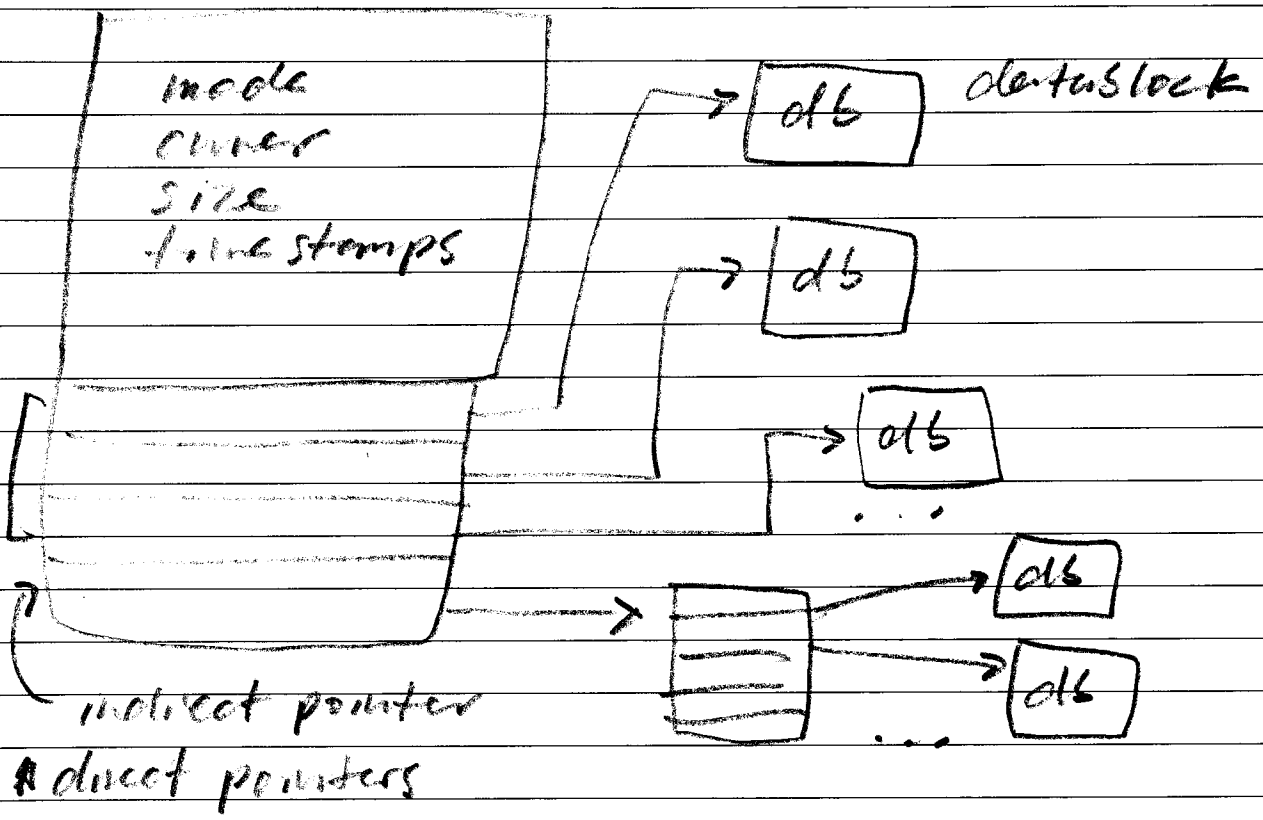
Superblock:



Date.
No.

File System: Inodes

Inodes store administrative data about files:



File System: Directories

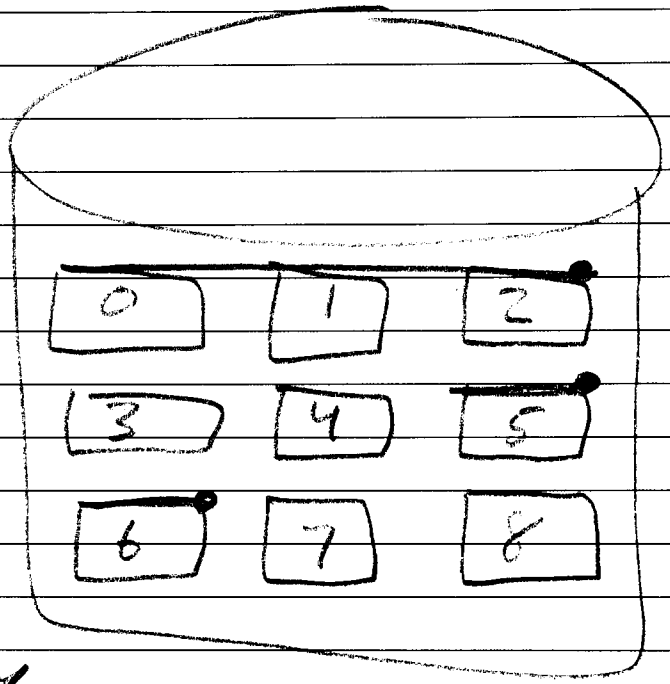
Directories typically contain data in a certain format to describe their directory entries (dentries):

inumber	reclen	strlen	name
5	4	2	.
2	4	3	..
12	4	4	foo
13	4	4	bar
24	8	7	foobar

Date.
No.

File System: Contiguous Allocation

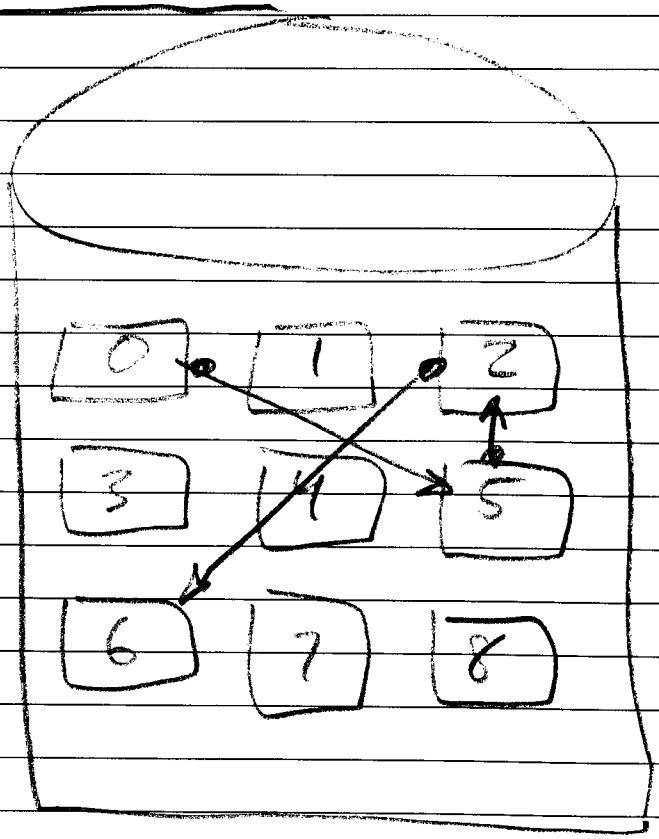
One way to organize data region is by using contiguous blocks:



- extents describe start and end
- easy and fast to access and lookup data
- prone to external fragmentation
- difficult to allocate efficiently

File System: Linked Allocation

Another way is to use a linked list:



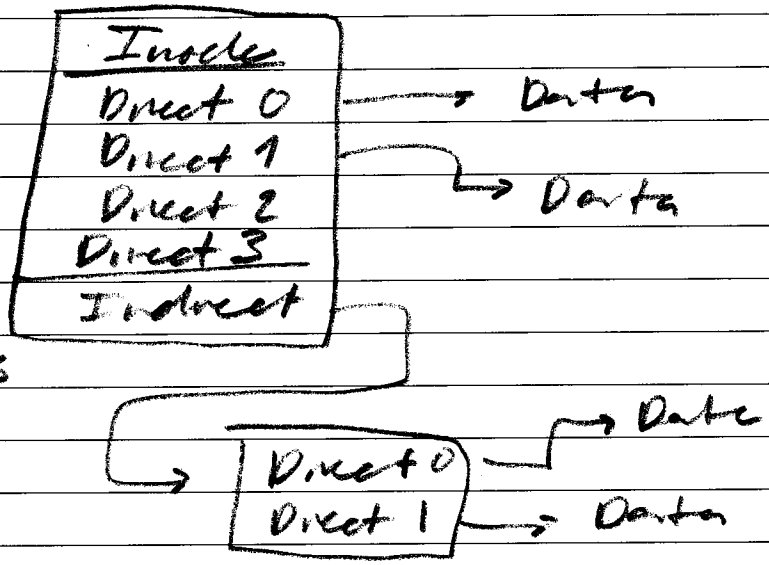
- each block points to the next block
- easy allocation
- poor random access, ignores spacerel locality

→ use a file allocation table to store links

Date.
No.

File System: Indexed Allocation

Another way is to use pointers to fixed size blocks:



- easy allocation
- relatively good random access
- overhead of pointers
- multiple levels of pointers

* walkthrough of Project 6.4 and pseudocode in lecture recording *

Date. 12/04/25
No. 27

FFS, LFS

Exam Q. Consider a multi-indexed file system w/ a 32-bit block address, 4KB block size, and an inode structure with 4 direct pointers and 1 indirect pointer:

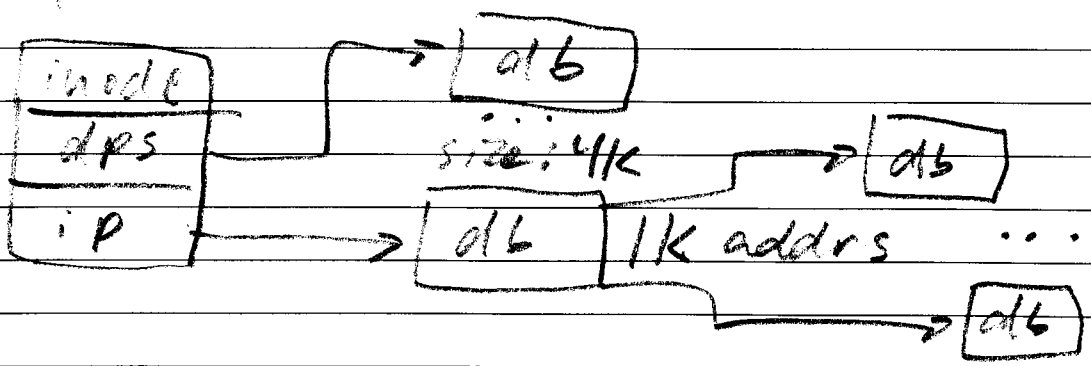
1. What is the largest disk this file system can use?

$$\text{Largest Disk} = \#\text{ of Blocks} \times \text{Size of block}$$

$$= 2^{32} \text{ blocks} \times 2^{12} B$$

$$= 2^{44} B = 16 \text{ TB}$$

2. What is the largest file this file system can hold?



$$\text{largest File} = \text{direct blocks} + \text{indirect blocks}$$

$4 \times 2^{12} B = 2^{14} B = 16 KB$ (direct)

$2^{10} \times 2^{12} B = 2^{22} B = 4 MB$ (indirect)

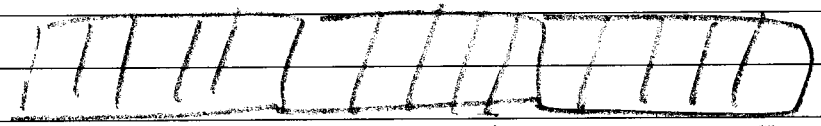
Date.
No.

FFS: Overview (Fast File System)

Problem. A hard drive is not like RAM; seeking is an expensive operation

- data blocks can be far from inodes
- data would be spread all over disk, leading to fragmentation

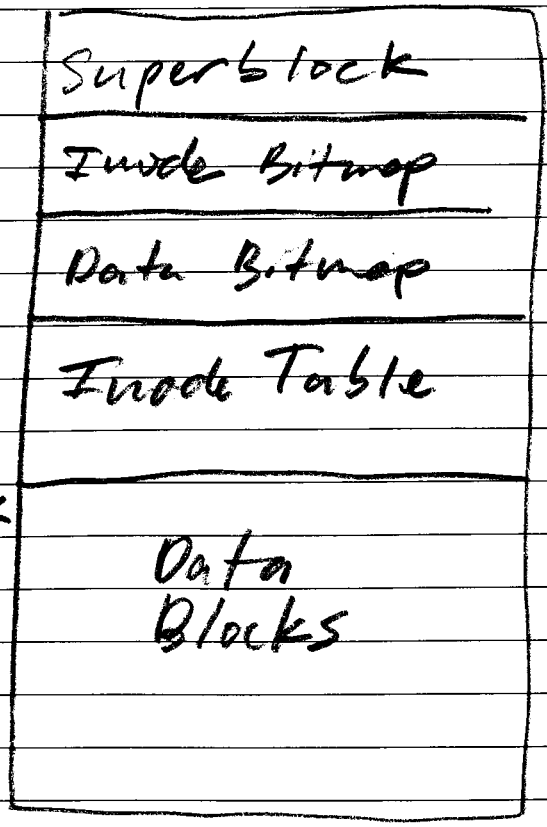
Solution. Make file system disk aware by organizing the disk into block groups



FFS: Block Groups

Each block group contains all necessary file system structures:

- per-group inode and data bitmaps which keep track of internal group inodes and data blocks
- multiple copies of superblock allow for reliability/redundancy
- most of the group contains data blocks



Date.
No.

FFS: Policies

To take advantage of this new organization, the FFS introduces the following heuristics:

- directories: Use group with low number directories and lots of free inodes
- files: Allocate data blocks in the same block group as inode and place files in the same directory in the same group (for large files, use up to a threshold with a group)

Log-structured File System (LFS): Overview

Problem: File systems do not take advantage of evolution in hardware resources nor do they consider modern workloads

- RAM is much bigger now
- Sequential I/O is really good

Solution: Buffer updates into a memory segment (buffer cache) and then eventually write the segment to disk when full:

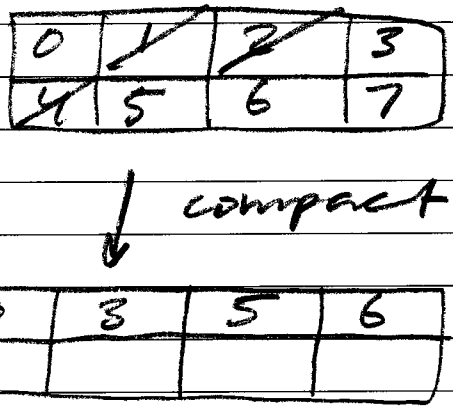
- Aggregates updates into fewer efficient writes
- Never overwrite data (always create new blocks)

Date.
No.

LFS: Garbage Collection

Never overwrite existing data. leads to the problem of dealing with garbage (old blocks)

1. Periodically read old segments
2. Determine which blocks are live
3. Write these live blocks to new segments



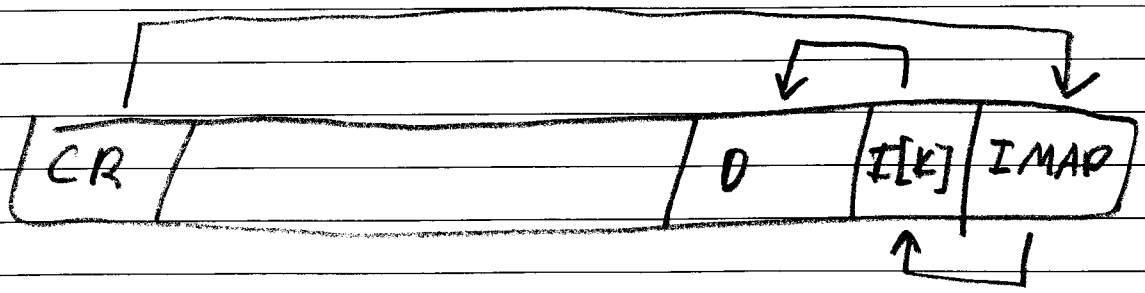
LFS: Imap and CR

Inode Map:

Because the data is now all over disk, we now need a data structure to map our inode numbers to the physical block

Checkpoint Region:

To keep track of these inode maps, we have fixed regions of memory that point to them



Date. 12.9.25
No. 28

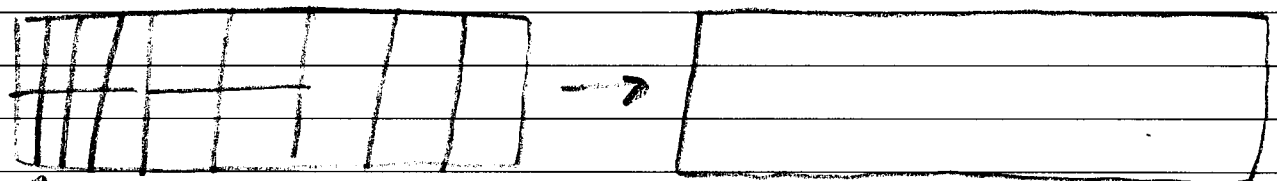
Consistency and Integrity

Consistency : Problem

Data written to the file system must persist, that is be accessible in the future. Unfortunately, some events can lead to inconsistent on-disk structures

- Forced removal
- System crash
- Power loss

Consistency : Example



Suppose we want to append to an existing file. We will need the following:

1. Allocate and copy to data block(s)
2. Update the inode
   a) size
   b) pointer to block(s)

Date.
No.

Consistency : Scenarios

What happens if only certain data is written?

Data Bitmap	Data Block	Inode Table	Consistent?	Outcome	Fixable?
X			No	Space Leak	Yes, clear bitmap *
	X		Yes	Data Loss	N/A *
		X	No	Garbage	Yes, update bitmap **
X	X		No	Space Leak	Yes, clear bitmap *
X		X	Yes	Garbage	N/A **
	X	X	No	Optimal	Yes, update bitmap

* but still data loss
** but still has garbage

note. Above on FINAL EXAM

note, Fixable means, structure self preservation...
values this over data integrity ... ironic!

Date.
No.

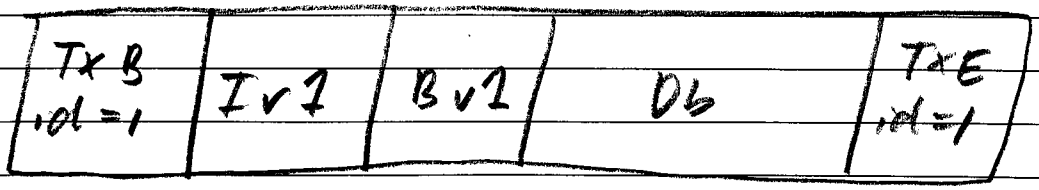
Consistency : FSCK

To recover from such scenarios, we can use the file system checker to repair the file system (usually runs at startup or before a device is mounted)

1. Superblock : Perform sanity checks
2. Free Blocks : Scan inodes and pointers to determine which blocks in use
3. Inode State : Check inode is structurally sound (otherwise clear it)
4. Inode Links : Check reference counts of links (dangling inodes move to lost+found)
5. Duplicates : Check if multiple inodes point to the same block (copy if necessary)
6. Bad Blocks : Check for any bad pointers (clear if bad)
7. Directory Checks : Checks structure and links

Consistency : Data Journaling

To speed up file system checking, we can use a write-ahead log to record our intentions and then perform those changes :



1. Journal Write : Write contents of transaction to log
2. Journal Commit : Write transaction commit block to log
3. Checkpoint : Write the contents of update to disk

Date.
No.

Consistency : Journaling Recovery

With this journal, we can recover from a system crash by scanning for any committed transactions and redo them in order:

Crash Before ...	Action
Journal write completed	Skip this transaction, data loss
Journal commit completed	Skip this transaction, data loss
Checkpoint completed	Replay transaction, duplicating work

note. The purpose of this is to maintain a consistent FS, not guarantee we maintain data

Consistency : Metadata

Unfortunately, data journaling is slow since we have to perform double the amount of writes. Instead, we can only journal the metadata:

TxB id=1	Iv1	Bv1	TxE id=1

1. Data Write : Write data to final location
2. Journal metadata write : Write begin transaction block
3. Journal checkpoint : Write transaction commit block
4. Checkpoint metadata : Write metadata to final location
5. Free : Mark transaction free

Date.
No.

Data Integrity: Problem

While fsck and journaling both keep the file system consistent, it doesn't protect the integrity of the data itself.

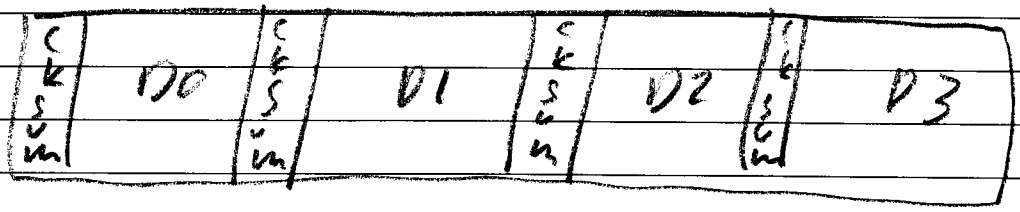
- Latent Sector Errors
- Block corruption

Although rare, these events do occur, so a storage system must account for these failures

Data Integrity: Checksum

Primary method for ensuring data integrity is computing the checksum of each block

- Every time we record a block to disk, we can generate and store a checksum
- Every time we are interested in the integrity of a block, we can compare the stored checksum to the computed checksum



Date.
No.

Integrity : Scrubbing

Once we have these checksums, we need to use them to detect corruption

1. We can check the checksum every time we read a block

2. We can also periodically scan the FS and check every block (we call this scrubbing)

Integrity : ZFS

- the Sun ZFS went for now and in the near future ... details in slides

Integrity : APFS

apple filesystem ... details in slides