Midterm Exam Study Guide
CS 415 Fall 2025

Note: The midterm is on November 4, 2025
In class, in person, on paper, closed book/notes

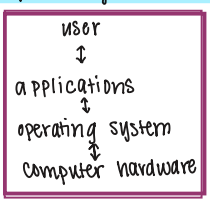
How to use this study guide? (We will also review these questions on Thursday.)

1. For **[Concept]** items, you should prepare for the exam by trying to find answers from the lecture slides and the textbook (Hint: AI is a good source too!). You might expect the following question types in the exam:
    a. Asking you to use a few sentences to explain/compare some concepts
    b. Asking you to draw a diagram to illustrate a concept using a simple example
    c. True/False questions
    d. Multiple-choice questions
2. For **[Algorithms]** items, you should get familiar with related algorithms. We will provide a brief explanation of the algorithms on the exam paper to help you, but it would be easier for you to become familiar with them in advance. The question type would be giving input and an algorithm, and we ask you to show the result.
3. For coding-related questions **[Reading/Writing Code]**, you should become familiar with the relevant functions. What do they do? We will provide a simplified version of the manual for those functions, but it would be easier for you to know them in advance. The code provided with the lectures is a good resource for learning about those functions. You will not be asked to fully master the functionality of those functions; we will only need you to know their basic usage and behavior.
4. For **[Reading Code]** items, you won't be asked to write any code for these types of questions, but you need to be able to run the provided code in your mind and write a possible output.
5. For **[Writing Code]** items, you will be asked to write code to complete given programs (e.g., filling in lines or blanks) to complete the program.

Lecture 1: Introduction to Operating Systems

1. **[Concept]** What is an Operating System?
2. **[Concept]** What is the role of an OS? What happens if we want to run a program but don't have an OS?
3. **[Concept]** Where does OS fit in a computer software/hardware system structure?
4. **[Concept]** What is the goal of an OS? What does OS do from the user and system perspectives?
5. **[Concept]** How does the OS interact with computer hardware? For example, how does the OS work with an I/O device such as a disk?
6. **[Concept]** What can cause an interrupt? How does the CPU/OS handle it?
7. **[Concept]** What is a trap? Who can cause a trap?
8. **[Concept]** What is the purpose of a timer interrupt?
9. **[Concept]** What are the differences between multiprogramming and timesharing?

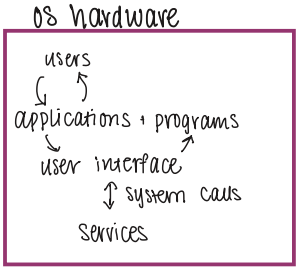
(1) OS → intermediary between user and hardware



Perspective:
- user: make hardware "easy to use"
  - convenience + ease
  - some performance + security
- system: using hardware efficiently
  - resource allocator
  - control program: manage execution of user programs

resource allocation
program execution: load program
user interface
logging | tracking
I/O operations
file system manipulation
communications
error detection

OS hardware

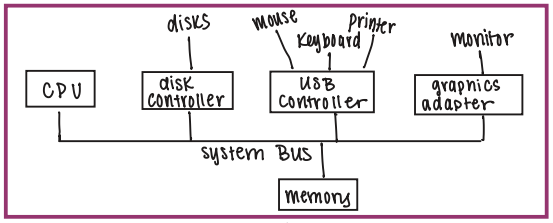


(2) without OS → need manual
- write program
- ownership of computer
- translate to machine code
- how to get program into memory
- start program
- turn off computer

with OS:
- put program into file
  - ↳ memory buffer
  - system call to send buffer out
- compile program
- tell OS to run executable file

Processor: CPU
handle all processing
- core
  - execute instructions

(3)



↳ system calls

← device driver: per device controller, understand DC and provide OS w/ uniform interface to device

← device controller: in charge of specific device type
local buffer storage

← System Bus: access between components + shared memory
↓
devices

(6) Traps + Interrupts

I/O request
read data from disk
I/O request to use device controller

→ CPU freed
pausing slow processes
context switch to new process

→ DMA transfer
transfer data to buffer in main memory
↳ using Direct Access Memory (DMA)
w/o CPU involvement

Direct Memory Access
allow I/O device controller to transfer data directly to main memory w/o CPU

Interrupt
tell CPU "done" loading data
via system bus

→ Interrupt Handling
pause process
save state + transfer control
↳ interrupt service routine
OS processes data, then switches: from A to B

Interrupt: signal from hardware to tell CPU to stop current work and move forward
timer interrupt: move from long processes

Trap: software-generated interrupt caused by error

Interrupt Architecture: save state info of what paused
restore info after fixing interrupt

Interrupt Vector: array of addresses w/ unique #
address of interrupt service routine

Interrupt Request Line: wire in CPU that CPU senses after executing after each instruction
→ read interrupt # and jump to IHR

Interrupt Handler Routine
interrupt # as index to interrupt vector
1) saves any state it will change
2) determine cause of interrupt
3) perform necessary process
4) perform state restore
5) executes return from interrupt

(9) Multiprogramming + Timesharing

Multiprogramming → Batch Systems
Maximize CPU utilization
OS organizes jobs
↳ CPU always has 1 to execute
when job must wait: OS switches jobs
Batch Jobs: when no user waiting

Timesharing → Multitasking
Maximize user interactivity + minimize response time
CPU switches jobs frequently
users can still interact w/ programs while running
illusion that many programs are running at same time

Lecture 2: Operating System Structure

- **[Concept]** What are the things typically managed by the OS? What services does OS provide for both the users and for efficient system operations?
- **[Concept]** Can OS work on its own in a purely software manner? If so, what would be the drawback of that?
- **[Concept]** What are the examples of hardware and OS working together to provide better service and management for the users?
- **[Concept]** What is the benefit of having a DMA in the system? What happens when a system does not have a DMA?
- **[Concept]** What is concurrency? What are the differences between logical and physical concurrency? Does having one type of concurrency in the system also indicate that the system has another type of concurrency?
- **[Concept]** What is the benefit of having logical or physical concurrency?

OS Services + Hardware Support

OS ⟷ Hardware
(work together)

Hardware: provides mechanisms for OS to enforce its policies

Key Hardware Features the OS relies on:

Protection (modes)

Kernel Mode: Priviledged
- OS runs in this mode
- access all hardware
- run protected instructions

user mode: non-priviledged
- user applications run in this mode
- trap if trying to run protected instruction
  - ↳ give control back to OS running instructions

Efficient I/O (interrupts + DMA)

DMA: efficiency boost
- w/o: read from disk manually and copy byte by byte
- with: DMA controller transfers a block on its own
  - CPU free to run another process
  - interrupt when entire block of data done

Scheduling (timer)
- periodic interrupts
- ensuring OS regains control
  - ↳ Make scheduling decisions (next or switch?)

Concurrency: processes

Logical: illusion of multiple processes running at once
- single core CPU → OS switch between processes quick (timesharing)

Physical: actually running multiple processes at once
- multi-core CPU

* **Seperation of Controls**
  - grouping | categorizing processes

Policy vs. Mechanism

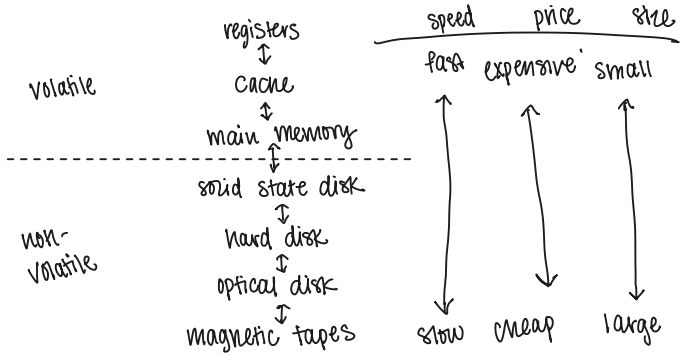
Mechanism: How to do something
- ex. hardware timer

Policy: What to do
- ex. scheduling policy

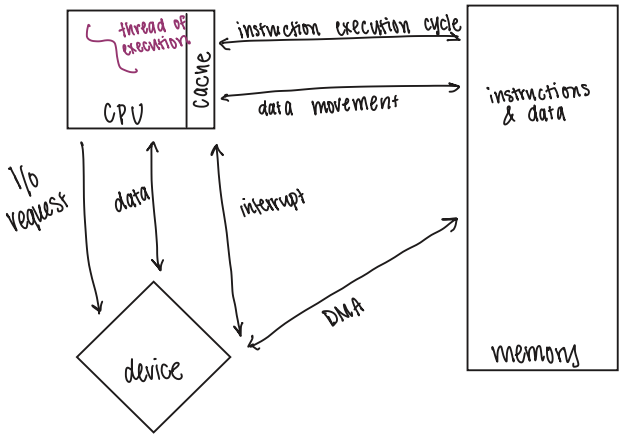
* OS uses mechanism to implement policy

mechanism doesnt change but policy can be updated

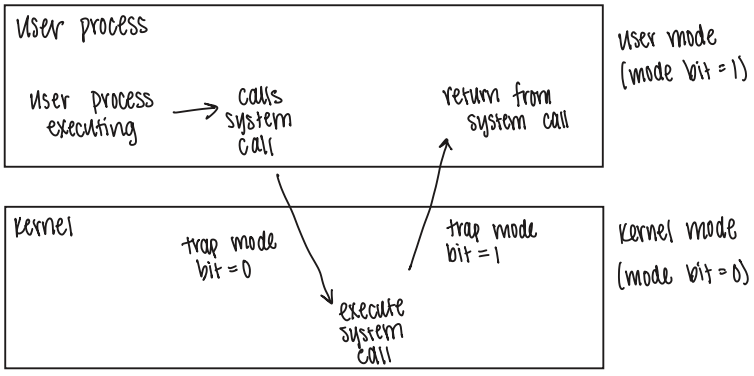
Storage + Memory Hierarchy



Modern Computer System



User to Kernel Mode

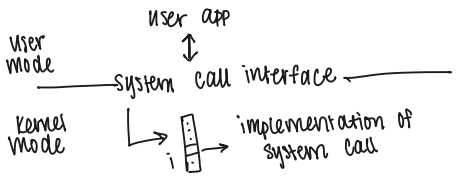


Lecture 3: System Calls

- **[Concept]** What are system calls? Where do they fit in the layers of OS designs?
- **[Concept]** Can you use an example to explain how a user application uses system calls to read files from a disk? Can you draw a diagram to show what happens?
- **[Concept]** What are the differences between system call (wrapper) and standard C library functions?
- **[Concept]** Can you list a few tasks that system calls can do?
- **[Concept]** What are system programs?
- **[Concept]** What is a file descriptor? In what scenarios are they used?
- **[Concept]** What are the differences between OS policy and mechanism? Examples?

What are System Calls?

programming interface to services provided by OS
way to switch to kernel mode

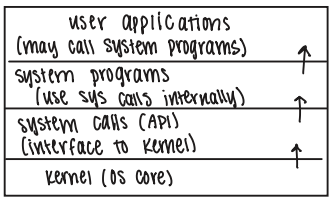


What can sys calls do?

Process control: fork(), exec(), wait(), exit()
File Management: open(), read(), write(), close()
Device Management: ioctl(), mmap()
Info. Maintanence: getpid(), gettimeofday()
Communications: pipe(), socket(), shm_open()

System Programs

programs that make it easier for user



ls, cp, basn

File Descriptors

integer to specify I/O stream to operate on
call func
↓
kernel creates entry in per-process file
↓
return integer as "handle"

0 → stdin
1 → stdout
2 → stderr

open() → file descriptor assoc. w/ file
close() → destroy file descriptor

System call process (ex. reading file from disk)

1) user application
program calls C library func.

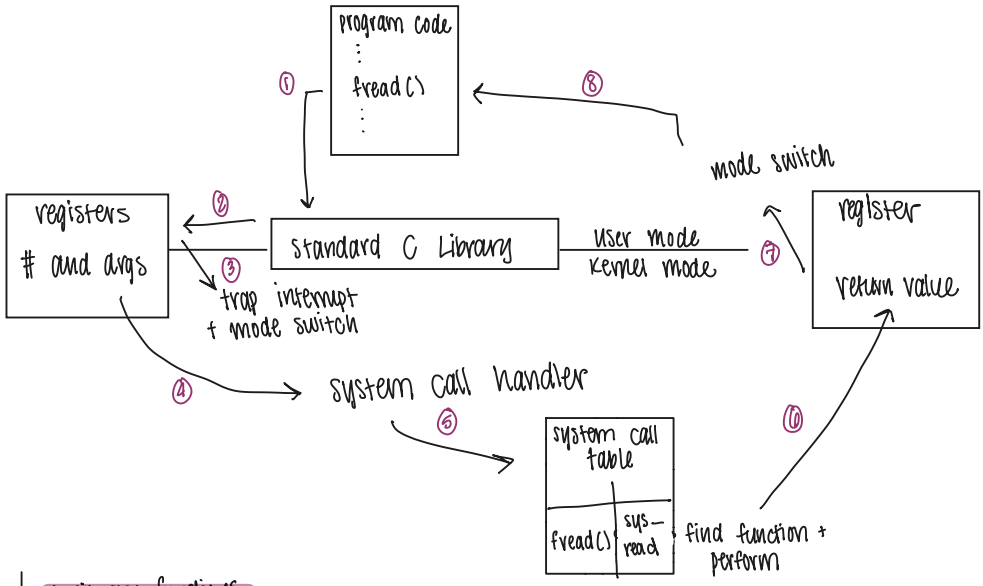
2) standard C library
lib func finds corresponding system call
puts # and arguments into CPU registers

3) Trap: software interrupt
lib func executes special trap instruction

4) Mode switch
from user to kernel mode
CPU jumps to spec. pre-defined location in kernel's code

5) Kernel Executes
system call handler looks at register info
find OS func to run using system call table
runs "real" function
performs protected operation

6) Return
place return value in a register
switch back to user mode
return control to program



System Call Wrapper vs. C Library Functions

Both: in C standard library (glibc)
can both be called from C code

System call wrappers

immediate response

- ex. read(2), fork(2), open(2)
- low-level
  1. load registers
  2. execute trap instruction to enter kernel
- almost 1-1 w/ actual system calls in kernel
- not portable: dif. on dif. machines
- unbuffered: perform action rn

Program
↓
System call wrappers
↓
Kernel (actual sys calls)

C library functions

- ex. printf(3), fopen(3), fread(3)
- high-level
- may call system call wrappers internally
- buffering: only performs expensive system call when buffer is full
- portable: C library handles correct non-portable system call by OS type

efficient + convenient + portable

small, frequent I/O operations

Program
↓
libc high-level functions
↓
sys call wrappers
↓
Kernel (actual sys calls)

Lecture 4: Processes

- Process concept and management
  - **[Concept]** What is the difference between a program vs. a process?
  - **[Concept]** What makes up a process? Explain what CPU state, process address space, and PCB are? What is the purpose of each of them?
  - **[Concept/Reading Code]** If a program is provided, can you draw a diagram showing what the process address space might be like at a given moment of execution? What roughly might be in the stack and the heap?
  - **[Concept/Reading Code]** If a program in assembly code is provided, can you show how much CPU state needs to be saved to the PCB when the OS decides to perform a context switch after executing a specific instruction?
  - **[Concept]** What might a process state be? How/Why do they convert between?
  - **[Concept]** How does context switching work?

Program vs. Process

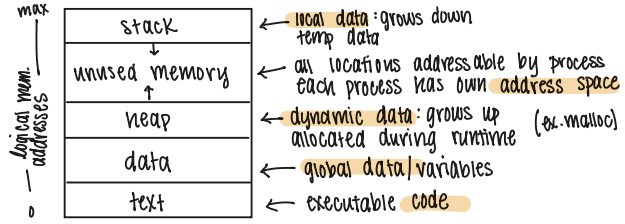
Program: Passive entity
Collection of instructions and data

Process: active entity
program in execution

single program can run many dif. processes

Process: make-up

① **Process address space**: logical layout for every process



② **CPU State**: what is required to execute each instruction on CPU
"snapshot" of what CPU doing at given moment
includes PC (points to next instruct) and all CPU registers

③ **Process Control Block**: kernels private data structure to manage process

process state ← "running", "waiting"
process number ← PID: process ID
program counter ← location of next instructions
registers ← process thread (data) CPU scheduling info } save CPU state
memory limits ← allocated, pointers
list of open files ← file descriptors
• • •

Process State:
new → creation
ready
running
waiting → blocked
terminated

Ex.
run → wait: sys call to read file
wait → ready: finish reading + send interrupt
run → ready: timer interrupt
ready → run: choose next process

Process Addressable Space Ex.

```
int = ... ← outside func = global → data
int = ... ← inside func = local → stack
malloc → allocating onto the heap
code + func → text
```

CPU State (Assembly Code)

```
Addr 100: MOV R1, 5     ; load a=5
Addr 101: MOV R2, 3     ; load b=3
Addr 102: ADD R3,R1,R2  ; c = a+b
Addr 103: MOV R0, R3    ; return value in R0
Addr 104: HALT
```

PC: 100	PC: 101	PC: 102	PC: 103	PC: 104
R0:	R0:	R0:	R0:	R0: 8
R1:	R1: 5	R1: 5	R1: 5	R1: 5
R2:	R2:	R2: 3	R2: 3	R2: 3
R3:	R3:	R3:	R3: 8	R3: 8

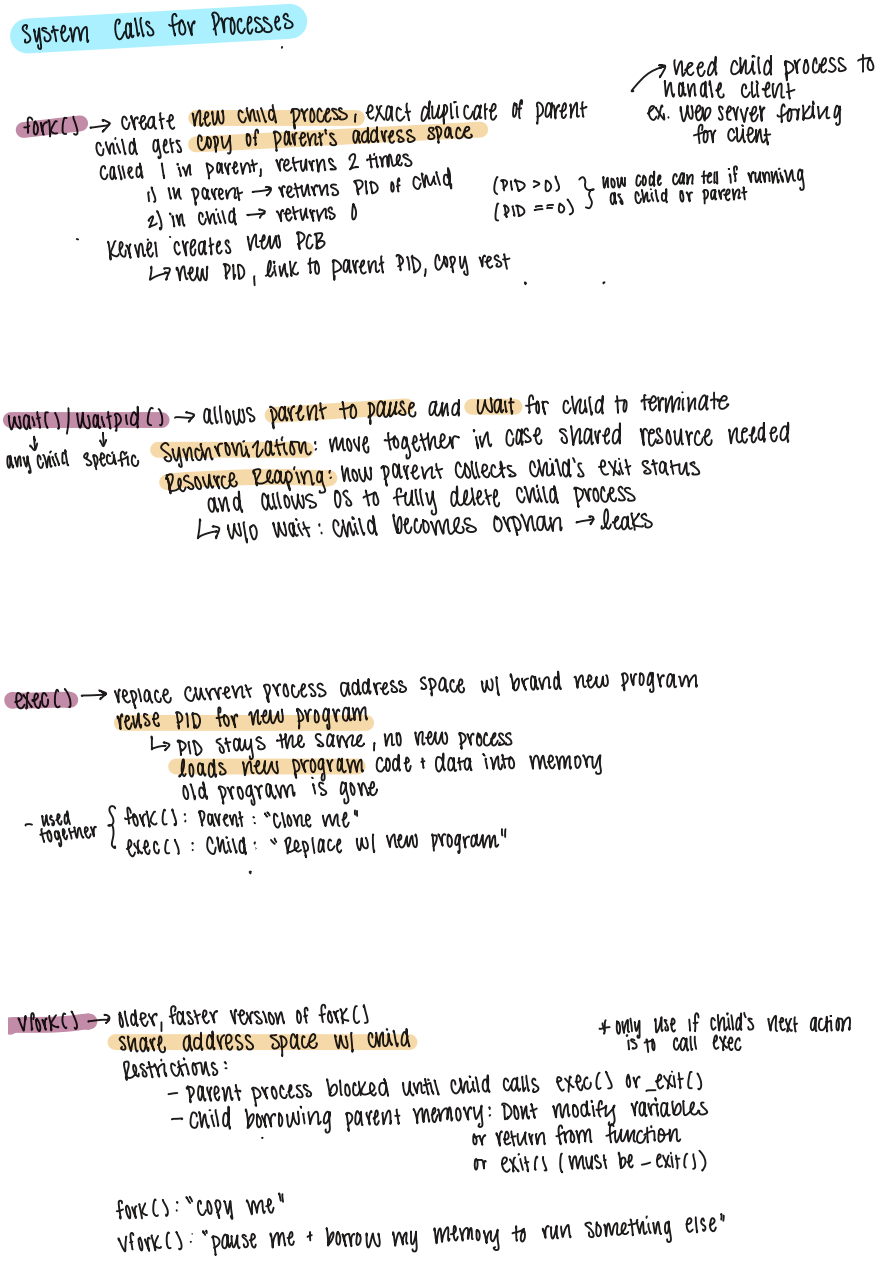
Context Switching

mechanism to switch CPU from process to process
when interrupt or trap occurs
1) Save CPU state of A into A's PCB
2) load saved CPU state of B into CPU
3) jump to PC just loaded, B resumes

} pure overhead
all handled by OS
↳no user work

Aspect	Process Address Space	Process Control Block
what	memory view of process	kernel data structure describing process
where	user space	kernel space
purpose	holds content of the program	holds metadata needed by os to manage process
Access	process itself	OS scheduler + kernel subsystems
Changes when	program loads new code, allocates memory, maps files	scheduler switches processes, updates state, tracks resources

- System calls/POSIX APIs for processes
  - **[Concept/Reading Code]** What happens after a process calls fork()?
  - **[Concept/Reading Code]** What is the purpose of wait/waitpid? Example scenarios where calling them is necessary? What happens if a parent does not call wait?
  - **[Concept/Reading Code]** What happens to a process after calling a function in the exec() family?
  - **[Concept/Reading Code]** What does vfork() do? How is it different than fork()? What should you do/not do after calling vfork()
  - **[Reading Code]** Given a program that calls fork/vfork/wait(pid)/exec(family), can you show the possible output of the program?
  - **[Writing Code]** If a requirement is provided, can you use the right fork/vfork/wait(pid)/exec(family) function to complete the program?



Lecture 5: Inter-process Communication (IPC)

- IPC in general
  - **[Concept]** What is IPC? Why do processes need them? Benefit? What would be the drawback of a system if it does not provide any IPC mechanisms?
  - **[Concept]** What are the two fundamental methods of IPC? How do they work? How is the kernel involved?
  - **[Concept]** What are the main properties (three dimensions) of an IPC implementation? What should we think about? Can you show examples of when we should use what?
- Shared Memory
  - **[Concept]** How does shared memory work? When is it better to use shared memory as IPC? Can you compare named and anonymous shared memory?
  - **[Concept]** What are the differences between POSIX shared memory and pipes?
  - **[Concept]** How does message passing work? How is it different than shared memory? When is it better to use message passing over shared memory?

Logical vs. Physical Memory

Logical: Process Address Space
memory as process sees → abstraction by OS
Physical: Physical hardware
Address Translation: CPU + OS together to take logic → physic

What is IPC? Inter-Process Communication

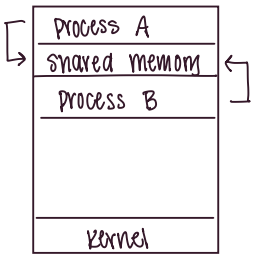
IPC: set of mechanisms allowing processes to exchange data and synchronize actions
Information Sharing : ex. web server + database
Computation Speed-Up : break up task, run parallel
Modularity: smaller pieces → big whole system

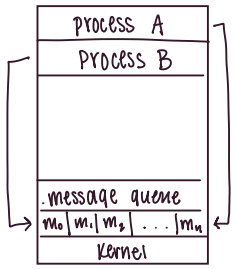
IPC Mechanisms

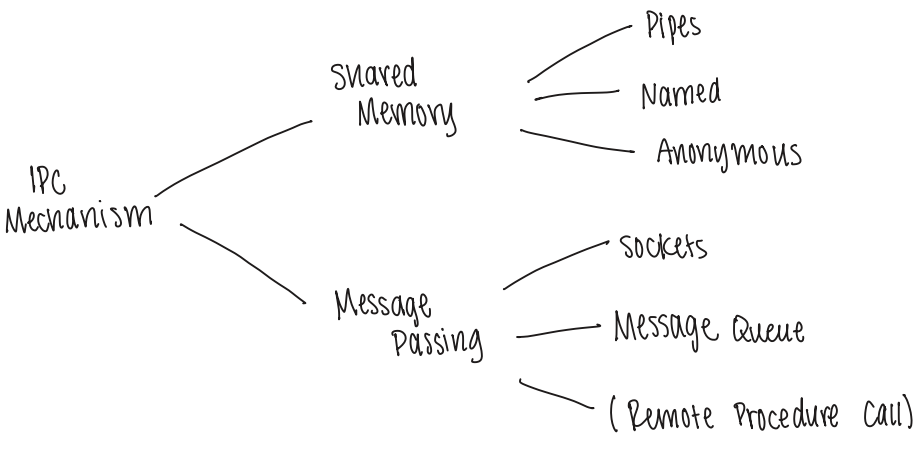
① Shared Memory : fast
- Kernel creates shared region in RAM
- each process maps shared region to own private address space

Kernel: setup + takedown, not in data transfer
Communication: reading + writing to shared memory
harder to program

② Message Passing : slower
- Kernel creates "mailbox" / "queue"
- Kernel involved in every operation
  - copy message from sender's address space to kernel's queue
  - copy from kernel queue to reciever's address space
- easier to program







Communication	Blocking	Buffering
ID	B	A
D	NB	E
D	NB	E
ID	B	A
ID	B	A

- Message Passing
  - **[Concept]** How does POSIX message queue work?
  - **[Concept]** How are file descriptors used in IPC such as shared memory, message queue, or socket?
  - **[Concept]** How do remote procedure calls work?
- Code with POSIX IPC
  - **[Reading Code]** Given a program that uses POSIX shared memory, pipes, or POSIX message queue, can you show the possible output of the program?
  - **[Writing Code]** If a requirement is provided, can you use the right IPC function (POSIX shared memory, pipes, or POSIX message queue) to complete the program?

Key Properties of IPC Implementation

① Direct vs. Indirect Communication
- **Direct**: Process must explicitly name each other
- **Indirect**: Processes send/recieve from intermediary
  - shared mailbox
  - flexible, don't need to know reciever

② Blocking vs. Non-Blocking : when buffer full & want to send
- **Blocking**: Synchronous : wait
  - send() waits for reciever to recieve
  - recieve() waits until message available
- **Non-Blocking**: Asynchronous : no wait
  - send() puts message in queue + returns immediately
  - recieve() grabs message if there
    - returns null/error if no

③ Automatic vs. Explicit Buffering
- **Automatic**: kernel manages buffer
- **Explicit**: programmers manage buffer

IPC Specific Mechanisms

Shared Memory (POSIX)
Direct Communication
Non-Blocking
Explicit Buffering

mmap() or shm_open() → file descriptor
→ mmap to map into process address space

- **Named**: can be used by unrelated processes
  - shm_open()
- **Anonymous**: only between parent + child
  - b/c child inherits mapping from parent
  - mmap w/ MAP_ANONYMOUS

PROS	CONS
speed large, complex data structures no system calls	responsible for all synchronization ↳ two processes writing @ same time = data corruption

Pipes: kernel managed circular buffer (shared memory)

Indirect communication
Blocking (default)
Automatic Buffering

pipe() → kernel-managed 1-way comm channel → 2 file descript.
fd[0] → read
fd[1] → write

Pipes	Shared Memory
implicit sync - kernel makes write() if pipe is full + read() if empty	explicit sync
byte stream - FIFO	random access - like array

PROS
simple, stream based data transfer
btwn related processes

Sockets: local + network communication

socket() → create socket → return file descript. → client calls connect() or accept()
→ use file des. to manage connection → send/recieve through read()/write() on socket file descript.

Indirect Communication
Blocking | Non
Automatic Buffering

POSIX Message Queue: kernel supported named queue

Indirect Communication
Blocking | Non (Config)
Automatic Buffering

mq_open() → named queue
→ mq_send() / mq_recieve() } discrete messages → close + unlink

Mechanism	Description	functions
Pipes	kernel managed, 1 way, byte stream FIFO buffer, file descriptors	pipe(), read() write()
Named Shared Memory	unrelated processes, map named mem back object to private adress space	shm_open() mmap()
Anonymous Shared Mem	flag to create memory region, not backed by a file, only parent → child	mmap() MAP_ANONYMOUS
Message Queues	kernel supported named queue, unrelated processes, discrete + prioritized messages	mq_send() mq_recieve()
Sockets	communication endpoints, send + recieve data locally or across network via FD	
Remote Procedure Call	abstraction, make procedure call to another process look like normal function call, stubs to marshall args	

one way
no overhead
RPC two way

PROS
distinct packets of info
more structured than pipes

Remote Procedure Calls (RPC): abstraction built on top of message passing mechanisms
↳ Sockets

make communication between processes a normal function call

1. client calls normal-looking func
2. client stub packs arguments into packs + sends over network
3. server stub recieves message, unpacks args, call real func
4. pack return value and send back the same

Lecture 6: Threads

- From processes to threads
  - **[Concept]** Why are processes considered heavyweight in terms of creation and context switch?
  - **[Concept]** How is execution state separated from resources? How are threads live inside a process?
  - **[Concept]** What is generally shared between threads in a process? What is private to each thread?
  - **[Concept]** Compare processes with threads. When is a multithreaded approach a better choice than multiprocessing? When is it not?
  - **[Concept]** If given requirements, can you choose the most suitable concurrency model (threads vs. processes)?
  - **[Concept]** Can different threads run different programs like child processes? What happens when a thread calls exec() to load a new program?
- Kernel threads
  - **[Concept]** What are kernel threads? How does it relate to user threads? Why are kernel threads needed?
  - **[Concept]** Does the OS scheduler directly manage user threads or kernel threads? Which type of thread is using OS resources, such as needing the OS to do bookkeeping?

Heavyweight Processes:

1) Creation: fork() → expensive
   - new PCB, copy PAS, cp FD, ...
2) Context Switch: slow
   - save entire CPU state
   - load new process from PCB
   - flush memory caches

solution to Heavyweight:

Seperating resources from execution

Threads: 1 process → multiple execution

Resources: address space

Execution: CPU state, stack

Shared vs. Private

in a single process, threads ...

share	own private
Address Space	CPU State ← threads exec dif. parts of code
Global Variables	Stack ← own local vars and func. calls
Open File Descripts.	

Processes vs. Threads:

Multiprocessing	Multithreading
+ isolation (one process crash is alone)	+ lightweight
+ protection	+ resource sharing (same mem)
+ security	+ multi-core processor utilization
- heavyweight (creating/switching)	- no isolation (thread crash → entire process)
- complex communication	- manual synchronization (prevent corrupting shared data)

Exec() in Threads

- replaces entire process (including all threads) w/ new program
- destroys current address space and re-init for new program
- all other threads terminated immediately
- threads cannot run dif. programs (only exec())

Kernel Threads vs. User Threads

User threads: programmer created + managed — abstraction
- using thread library

Kernel threads: "real" threads that OS kernel knows about and schedules to run on CPU — OS can schedule on CPU
- execution container

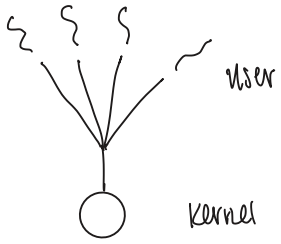
- Thread models
  - **[Concept]** What happens when there are more user threads than kernel threads?
  - **[Concept]** What ways can user threads be mapped to kernel threads?
  - **[Concept]** How does each thread model work? Scheduling?
  - **[Concept]** What are the major benefits/drawbacks of each thread model?
  - **[Concept]** Which thread model does Linux use?
- Code with POSIX thread (pthread)
  - **[Reading Code]** Given a program that uses POSIX threads, can you show the possible output of the program?
  - **[Writing Code]** If a requirement is provided, can you use the right (POSIX threads APIs to complete the program?

Thread Models: how user threads map to kernel threads

Many-to-One (M:1): many user threads → single kernel thread
- thread lib in user space manages

Cons: any thread making blocking system call blocks the single kernel thread.
entire process blocked, no other thread can run

(user thread library)

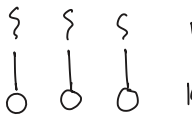


One-to-One (1:1): each user thread → own kernel thread

Pro: Blocking syscall only blocks one user thread.
other threads can continue to run.

(LINUX model)

(system call + kernel)



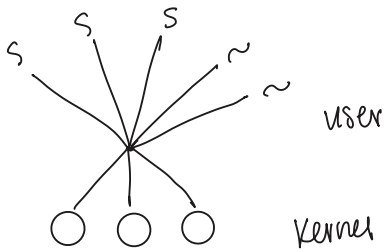
Many-to-Many (M:N): Pool of M user threads → Pool of N kernel threads

$n < m$ (usually)

Pro: Blocking syscall blocks one kernel thread
user thread lib can schedule another ready user thread onto available kernel thread

Con: complex

(user thread library)



POSIX Threads

API (function specifications) for creating + managing threads on UNIX-like systems

pthread_create(): Create new thread
pthread_join(): wait for specific thread to finish

(make user-level threads)

Lecture 7: Scheduling

- **[Concept]** What could be the goal of resource scheduling?
- **[Concept]** When does the OS need to make scheduling decisions?
- **[Concept]** What are the benefits/drawbacks of non-preemptive and preemptive algorithms?
- **[Algorithm]** If given processes with their arrival time, burst duration/priority, can you show the scheduling results using FCFS, SJF, SRTF, Priority Scheduling (non-preemptive or preemptive), and Round Robin?
- **[Concept]** Can you calculate the average waiting time for all processes?
- **[Concept]** Can you calculate the turnaround time and waiting time for the individual process?

Goals + Criteria of Scheduling

Maximize CPU utilization : keep CPU busy
Maximize Throughput : # of processes completed
Minimize Turnaround Time : submit → complete
Minimize wait time : time spent in ready queue before CPU
Minimize Response Time : user gives command → first response

Pre-emptive vs. Non-Preemptive Scheduler

Non-Preemptive: OS cannot reschedule actively running process
- process must give up CPU : terminate or blocking

Pros : low overhead (fewer context switches)
simple implementation
Cons : long job forces short jobs to wait long

Preemptive: OS can force process to stop running
move it to ready
timer interrupt

Pros : all processes have chance to run
better response time (for interactive users)
Cons : higher overhead (context switch)

Gantt Chart : Process tasks w/ timeline

Round Robin: Preemptive

FCFS w/ time quantum (q)
Ready queue is circular

picks first process → run for (at most) 1 q → timer interrupt

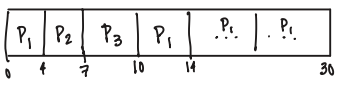
Still running
OS preempts → end of queue
move to next in line

finished before terminate/block
run next process in line

+ Response Time, Interactivity
- q is large → FCFS
- q is small → high overhead, waste CPU time

q = 4

Process	Burst
P_1	24
P_2	3
P_3	3



Thread Scheduling:

OS schedules Kernel threads (when multiple)

Process-Contention Scope (PCS): thread lib in user space decides mapping to kernel thread

System-Contention Scope (SCS): OS kernel decides which kernel thread (from any process on system) to schedule on CPU core

user thread → PCS → kernel thread → SCS → schedule on CPU

Scheduling Decisions : choose new process to run

1) Running → Waiting (I/O request)
2) Terminates
3) Running → Ready (timer interrupt)
4) Waiting → Ready (I/O complete)

Scheduling Algorithms

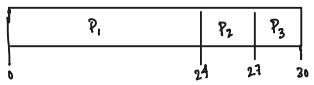
First-Come First Serve (FCFS) : Non-Preemptive

Processes → Queue (as they arrive) → Head of Queue 1st (until block/terminate)

- Convoy effect: long process causing shorter ones behind it to wait

Ex.

P_1	24
P_2	3
P_3	3

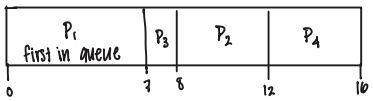


Shortest Job First (SJF): Non-Preemptive

CPU free → scheduler picks next shortest job → Run on CPU

+ provably optimal for minimizing average wait time

P_1	7
P_2	4
P_3	1
P_4	4

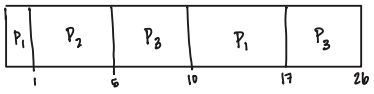


* scheduler only making decision when next job is to be scheduled

Shortest Remaining Time First : Preemptive (of SJF)

new process ready → burst time compare to remaining time on current process → if shorter by arrival → Preempt current process, run new process

Process	Arrival	Burst
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5



Priority Scheduling : NP or Preemptive

Process assigned priority # → runs process w/ highest priority

Process	Burst	Priority
⋮	⋮	⋮

- starvation: low priority may never run
+ Solution: Aging: Priority gradually increased as it waits in ready queue

Manual for Process and Thread Related Function Calls

Manual for POSIX Shared Memory

t shm_open(const char *name, int flag, mode_t mode) - *only needed for named shared memory*

- **name**: Name of the shared memory object.
- **flag**: Flags for accessing the file. (*you should set it to O_CREAT | O_RDWR*).
- **mode:** File permissions (*you should set it to 0666 in you answers*).
- Returns file descriptor on success or -1 on error (*you can skip checking error in your answers*).

t ftruncate(int fd, off_t length) - *only needed for named shared memory*

- **fd**: File descriptor (*from shm_open()*).
- **length**: Size of shared memory buffer in *bytes*.
- Returns 0 on success or -1 on error (*you can skip checking this in your answers*).

oid *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset) – *for named and anonymous*

- **addr**: Preferred address (*you should set it to 0 for "let kernel choose"*).
- **length**: Size of shared memory buffer in bytes.
- **prot**: Access permissions. Should set to PROT_READ | PROT_WRITE for both named and anonymous.
- **flags**: Named: set to MAP_SHARED. Anonymous: set to MAP_SHARED | MAP_ANONYMOUS.
- **fd**: Named: set to the file descriptor from shm_open(). Anonymous: set to -1.
- **offset**: Start offset in file (*you should set it to 0*)
- Returns address of shared memory or -1 if on error (*you can skip checking error in your answers*).

Manual for POSIX Message Queue

qd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr)

- **name**: Name of the message queue
- **flag**: Sender: *set to O_CREAT | O_WRONLY*. Receiver: set to O_RDONLY.
- **mode:** File permissions (*you should set it to 0666 in your answers. Only needed for sender*).
- **attr**: Pointer to mq_attr structure specifying queue limits.
- Returns message queue descriptor or -1 on error (*you can skip checking error in your answers*).

t mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned int msg_prio)

- **mqdes**: The message queue descriptor (*from mq_open()*).
- **msg_ptr**: Pointer to the message buffer to send (*remember: cast your pointer to char**).
- **msg_len**: Length of the message in *bytes*.
- **msg_prio**: Message priority (*you can always set it to 0 in your answers*).
- Returns 0 on success or -1 on error (*you can skip checking this in your answers*).

ize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio);

- **mqdes**: The message queue descriptor (*from mq_open()*).
- **msg_ptr**: Buffer to store the received message. (*remember: cast your pointer to char**)
- **msg_len**: Size of the buffer in *bytes*.
- **msg_prio**: Optional pointer to store message priority (*you can always set it to NULL in your answers*).