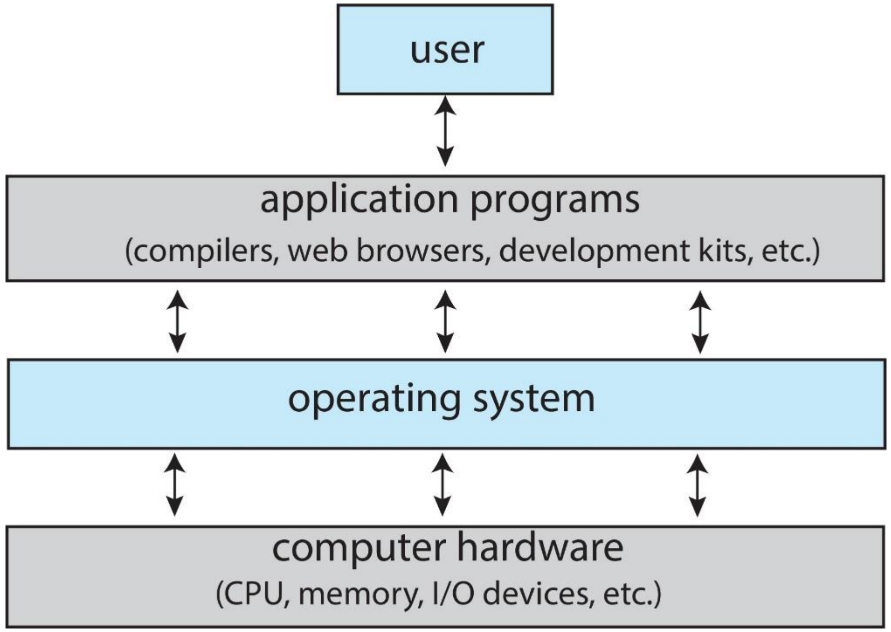
Lecture 3: Introduction to Concurrency

Xin Liu
xl24j@fsu.edu
COP 4610 Operating Systems

Outline

- OS as a State Machine
- Processes & Threads
- Dispatcher
- Amdahl’s Law

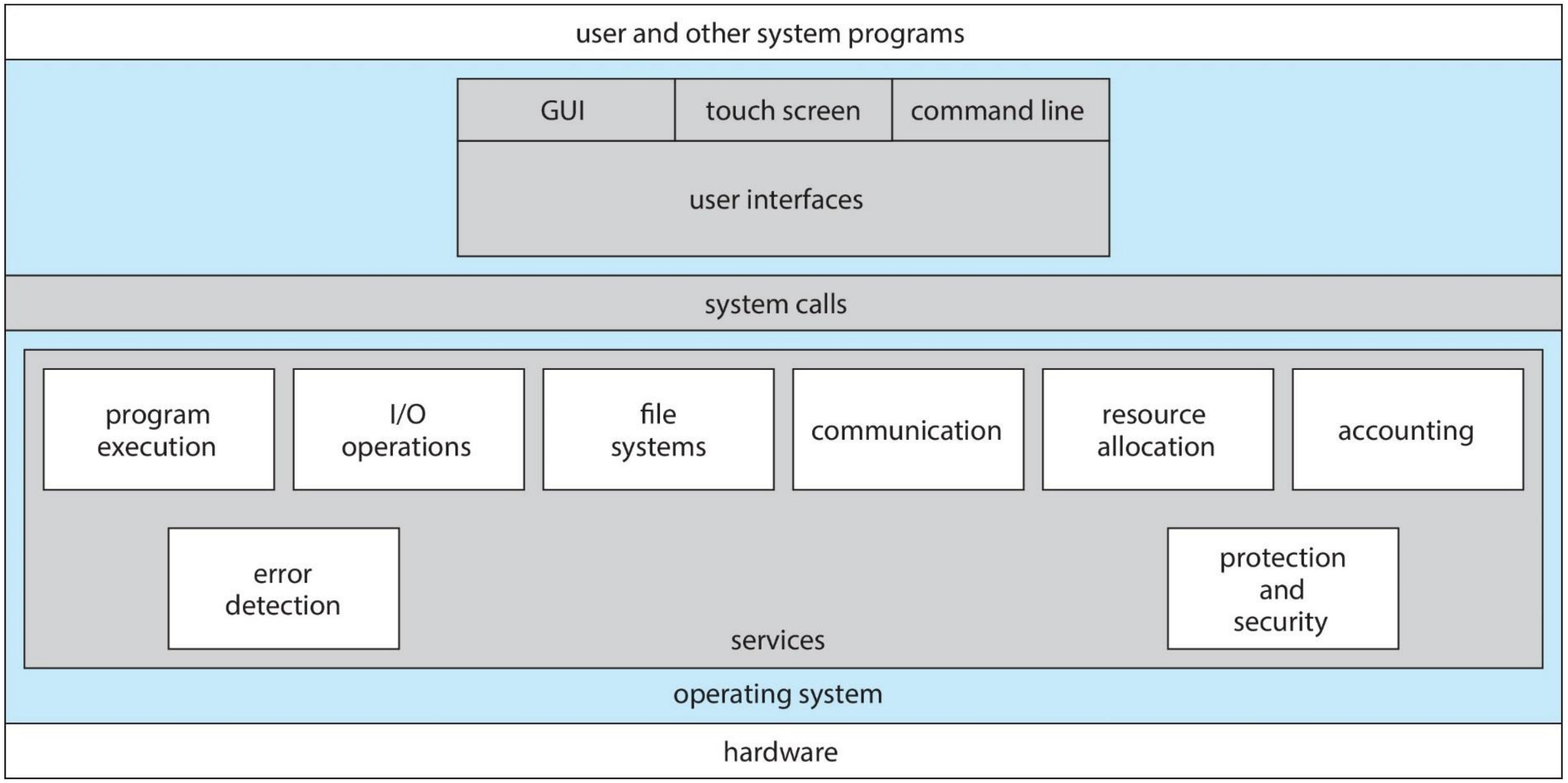
Recap: Abstract View of Components of Computer


user
application programs
(compilers, web browsers, development kits, etc.)
operating system
computer hardware
(CPU, memory, I/O devices, etc.)

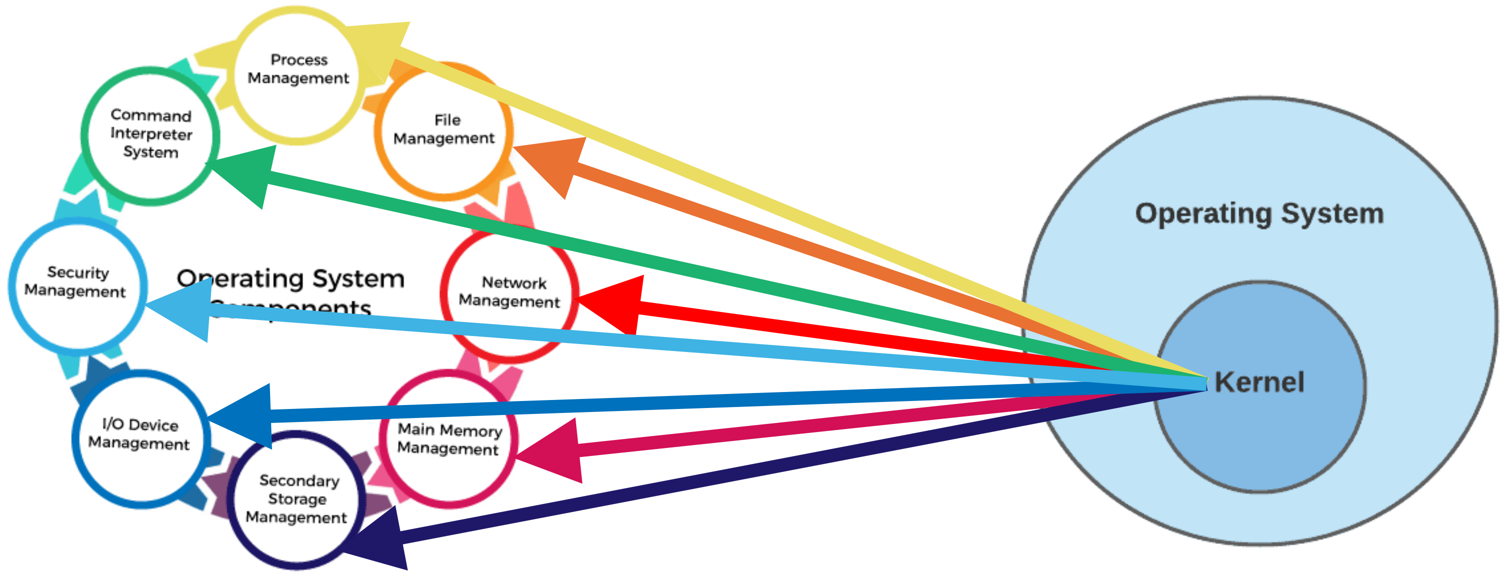
OS is a **program** that acts as an intermediary between a user of a computer and the computer hardware.

OS hides the complexity and limitations of hardware from application programmers.

Recap: A View of Operating System Services

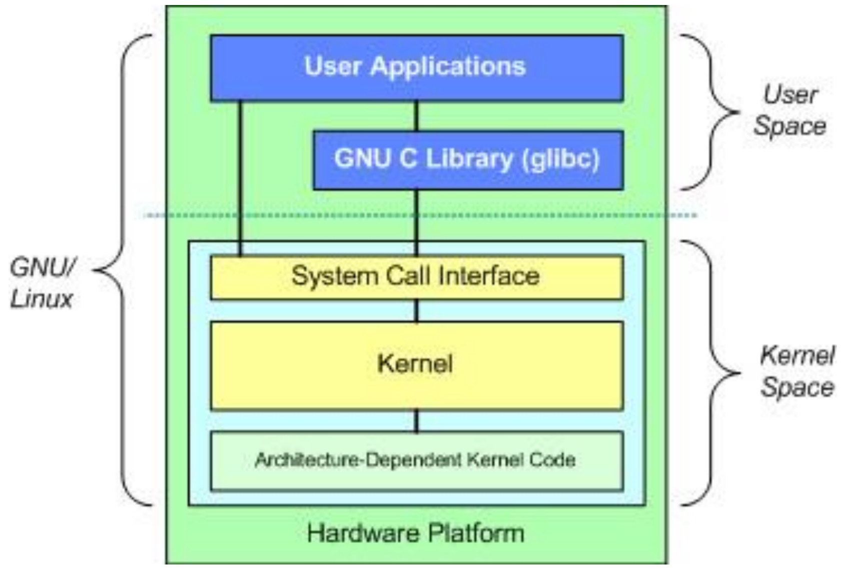


Recap: Kernel: The Core of the Operating System



Recap:
Dual-Mode Operations: Kernel Space and User Space

A mechanism to protect apps from crashing the OS



Kernel space is the memory area where the operating system kernel runs, with the highest level of privileges.

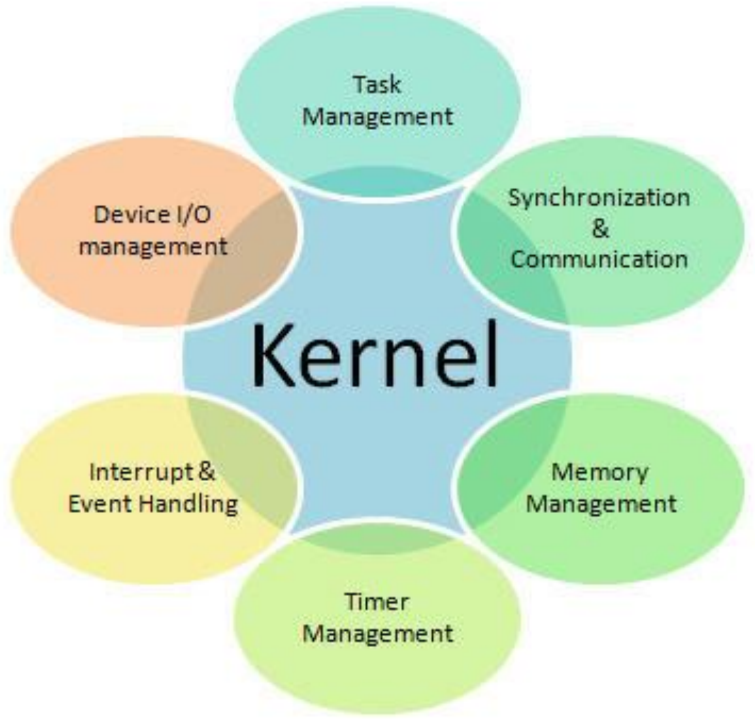
- **Function:** Manages system resources such as memory, CPU, and device drivers.
- **Security:** Due to its high privileges, kernel space code must be very stable and secure; any errors can cause system crashes.
- **Access:** Only code running in kernel mode can access kernel space; user mode code cannot directly access it.

User space is the memory area where regular applications run, with lower privileges.

- **Function:** Runs user applications like browsers, text editors, etc.
- **Security:** Errors in user space typically do not affect the overall system stability due to its lower privileges.
- **Access:** Code running in user mode can only access user space and must use system calls to interact with the kernel.

Important

Recap: Understanding the Kernel



- **Task Management:** Schedules and manages processes.
- **Memory Management:** Allocates memory to processes.
- **Device I/O Management:** Controls interactions with external devices.
- **Synchronization & Communication:** Ensures smooth process interaction.
- **Interrupt & Event Handling:** Responds to hardware events and interrupts.
- **Timer Management:** Manages system time and timers.

Recap: Kernel Design and Trends

- Monolithic, Layered, Microkernel, Modular
- Trend: Hybrid
  - Based on the **microkernel**, driven by the rise of distributed networks, enabling platforms like smart watches, smart TVs, and laptops to run the same OS kernel while loading corresponding modules, including monolithic ones.

Important

Why *Concurrency*?

- Allows multiple applications to run at the same time
  - Analogy: juggling

- What is an applications?
  - A Program that runs on the OS.

History Phase I: Hardware Expensive, Humans Cheap

- Hardware: mainframes
- OS: human operators
  - Handle one ***job*** (a unit of processing) at a time
  - Computer time wasted while operators walk around the machine room

IBM System/360

OS Design Goal in Phase I

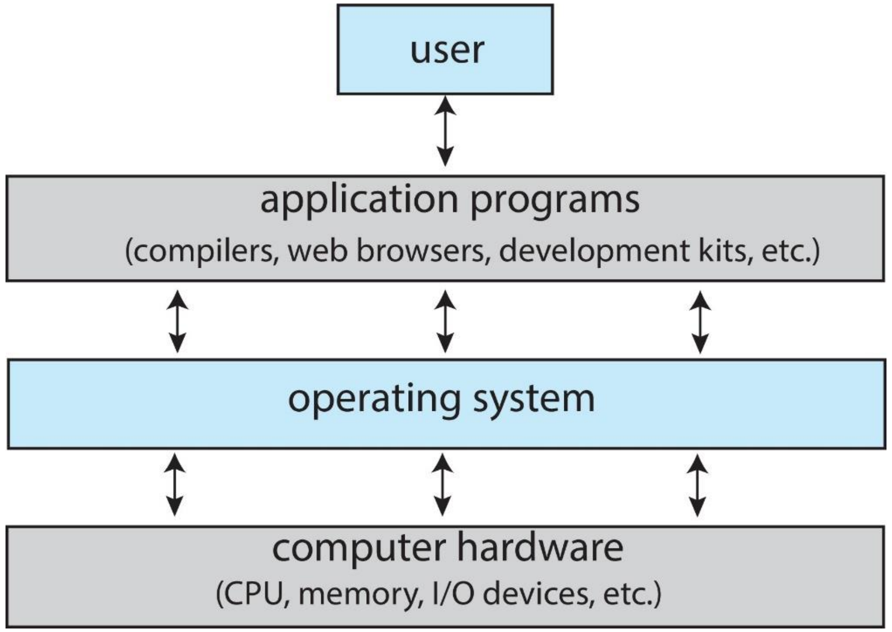
- Efficient use of the hardware
  - **Batch System:**
    - **Collects a batch of jobs before processing and processes them sequentially**
    - **Emphasizes throughput by reducing idle time during job collection and processing.**
  - **Multiprogramming:**
    - **Allows multiple programs to run simultaneously, improving CPU utilization by switching between jobs.**
    - **Efficiently handles both I/O-bound and CPU-bound jobs, minimizing idle CPU time.**
    - **Key Point: Focuses on maximizing system resource usage by running multiple tasks concurrently.**

Why *Concurrency*?

- Allows multiple applications to run at the same time
  - Analogy: juggling

- What is an applications?
  - A Program that runs on the OS.

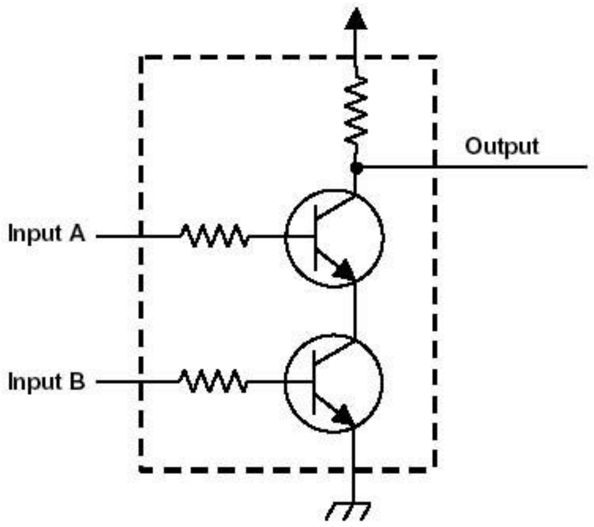
Abstract View of Components of Computer

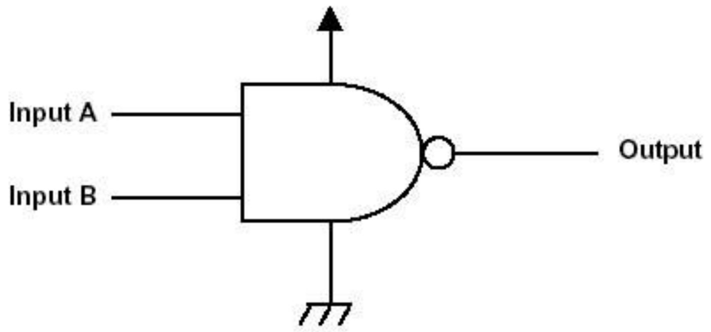

user
application programs
(compilers, web browsers, development kits, etc.)
operating system
computer hardware
(CPU, memory, I/O devices, etc.)

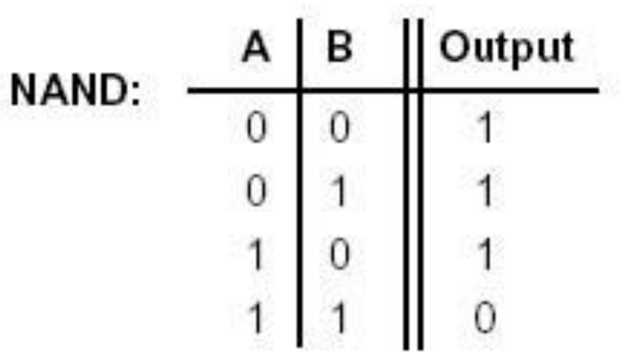
OS is a **program** that acts as an intermediary between a user of a computer and the computer hardware.

What is a program?

One Logic Gate





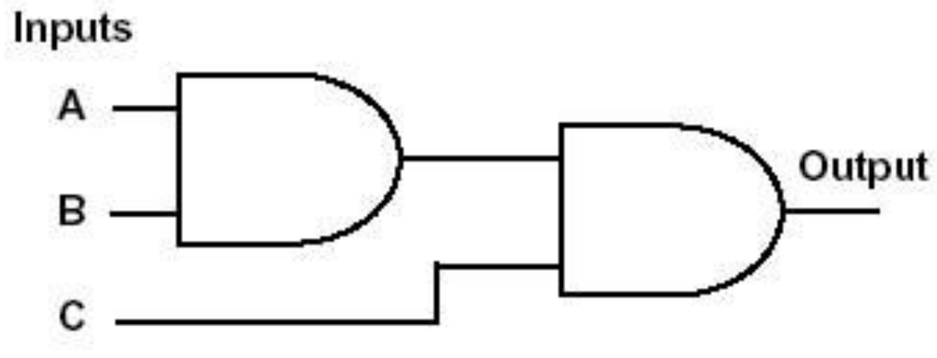


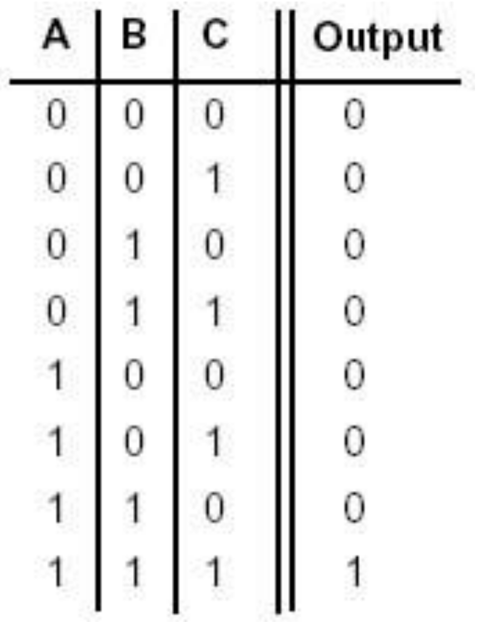
NAND:

A	B	Output
0	0	1
0	1	1
1	0	1
1	1	0

Truth Table

Two Logic Gates

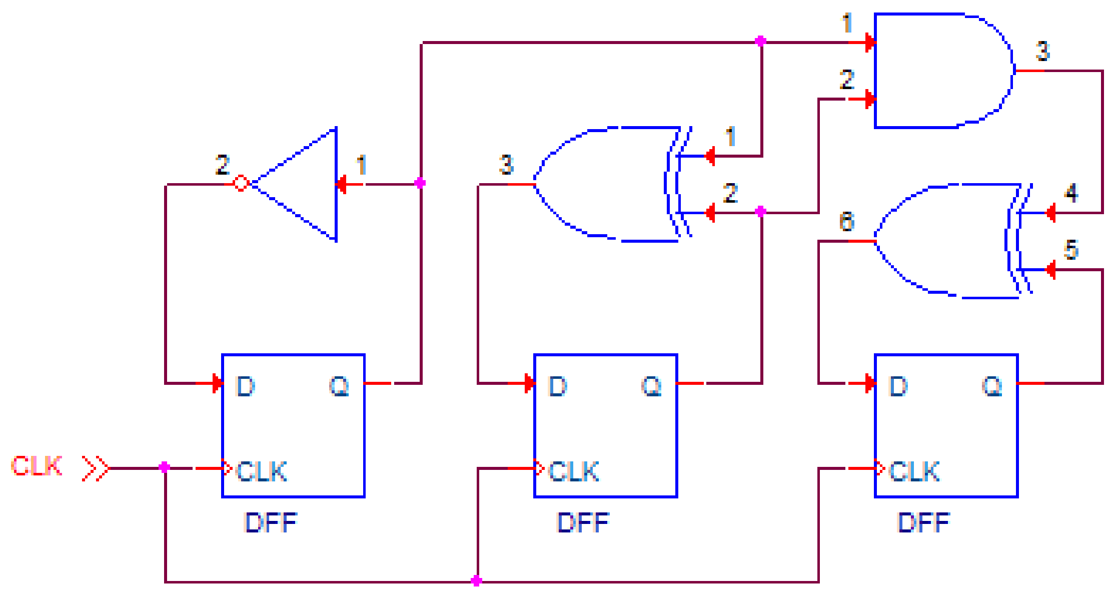




A	B	C	Output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

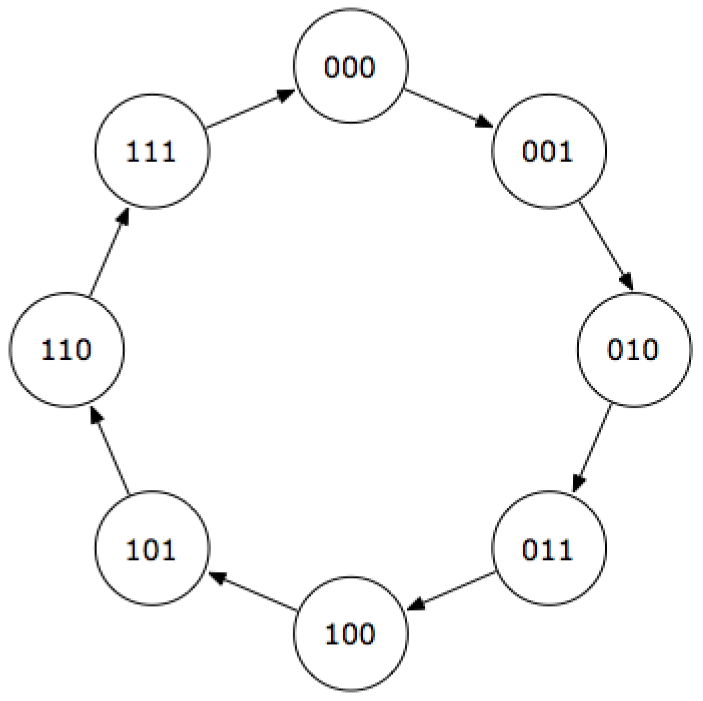
Truth Table

More Logic Gates



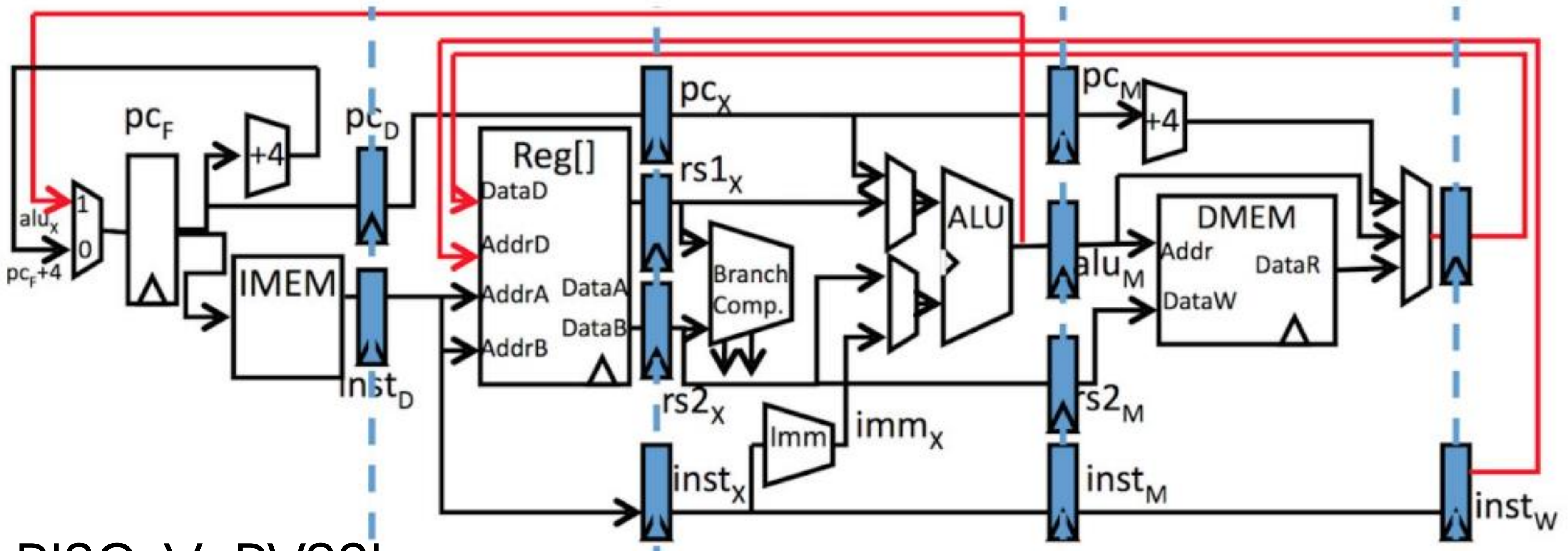
Q_2	Q_1	Q_0	D_2	D_1	D_0
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	0	1
1	0	1	1	1	0
1	1	0	1	1	1
1	1	1	0	0	0

Truth Table



Finite State Machine

Millions of Logic Gates



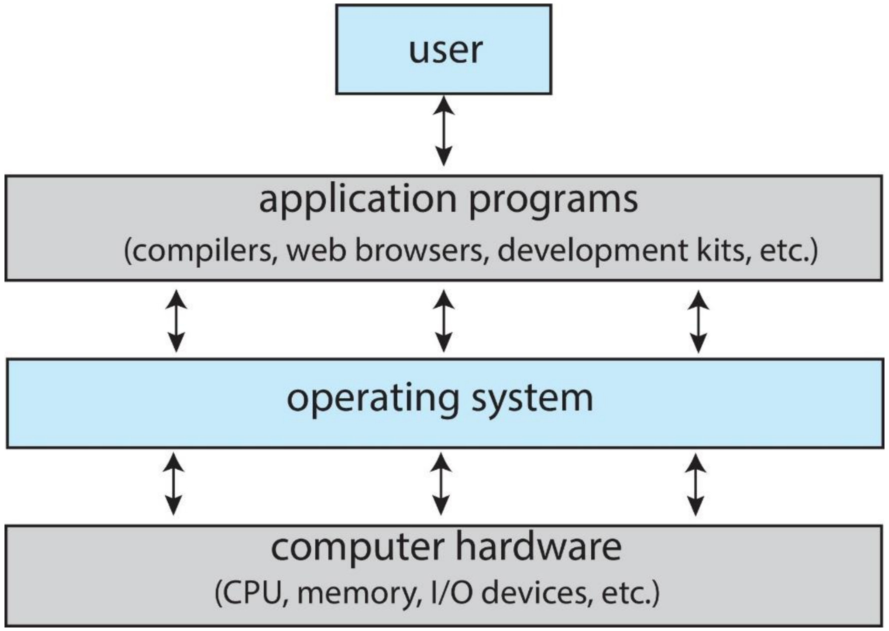
RISC-V-RV32I

31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

From CPU to Program

- CPU is a state machine.

- A program running on a CPU is inevitably also a state machine



user
application programs
(compilers, web browsers, development kits, etc.)
operating system
computer hardware
(CPU, memory, I/O devices, etc.)

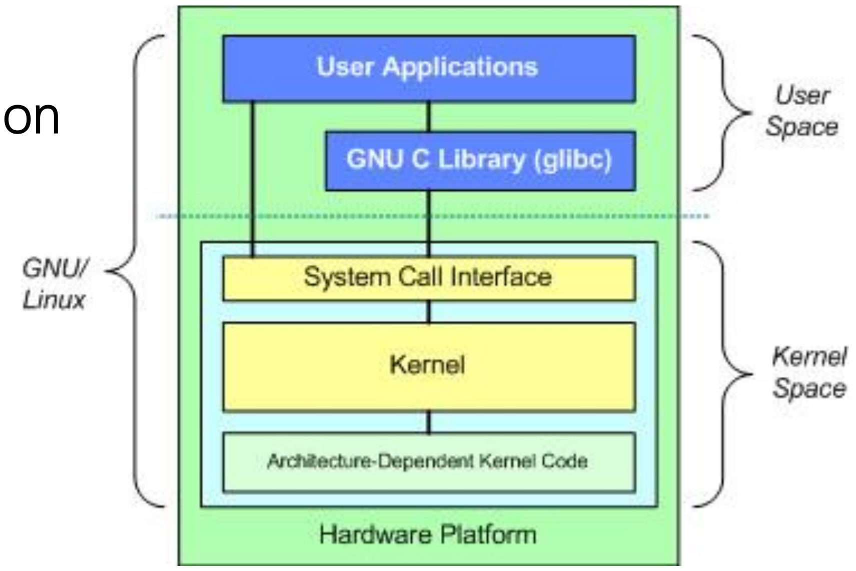
State in the State Machine of Program

- **Program Counter (PC)**: It indicates the position of the instruction currently being executed.
- **Register State:** The current values of all registers, including general-purpose and specialized registers.
- **Memory State:** The data stored in the program's memory, especially the values of local variables, global variables, etc.
- **Input/Output State:** The interaction state of the program with external devices, such as the keyboard or display.
- **Stack State:** The current state of the function call stack, including function calls, local variables, return addresses, etc.

From the Application’s Perspective

- **How an application works:**
  - Processing Computations and Logic:
    - Applications directly execute logic and arithmetic operations via the CPU, handling internal data and state changes, such as computation results, conditional checks, variable assignments, etc.
  - When an application needs hardware resources, it makes a **system call**.
  - The application passes **its state** to the OS.
  - The OS manages hardware and returns control to the application

Computation -> System Call -> Computation -> System Call …



OS hides the complexity and limitations of hardware from application programmers.

Example: Smallest size HelloWorld.c

```c
#include <stdio.h>


int main()
{
    printf("Hello, OS World!\n");
}
```

15960 bytes

```
#include <sys/syscall.h>

.globl _start
_start:
    movq $1, %rax              # write(
    movq $1, %rdi              #  fd=1,
    movq $st, %rsi             #  buf=st,
    movq $(ed - st), %rdx      #  count=ed-st,
    syscall                    # )

    movq $60, %rax     # exit(
    movq $1, %rdi              #  status=1
    syscall                    # )

st:
    .ascii "\033[01;31mHello, OS World\033[0m\n"
ed:
```

4856 bytes

Example:

https://github.com/xinliulab/COP4610_Operating_Systems/tree/main/Lecture_3_Concurrency_Processes_and_Threads

Why *Concurrency*?

- Allows multiple applications to run at the same time
    - Analogy: juggling

- Program is a state machine.

- What is the red ball?
    - A State.

Benefits of Concurrency

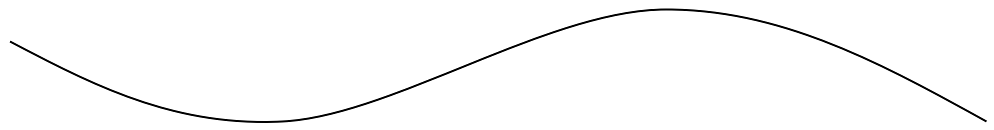
- Better performance
  - One application uses only the processor
  - One application uses only the disk drive
  - Completion time is shorter when running both concurrently than consecutively

Drawbacks of Concurrency

- Applications need to be protected from one another
- Additional coordination mechanisms among applications
- Overhead to switch among applications
- Potential performance degradation when running too many applications

Thread

- A sequential execution stream
  - The smallest CPU scheduling unit
  - Can be programmed as if it owns the entire CPU
    - Implication: an infinite loop within a thread won't halt the system
  - Illusion of multiple CPUs on a single-CPU machine



Thread Benefits

- Simplified programming model per thread
- Example: Microsoft Word
  - One thread for grammar check; one thread for spelling check; one thread for formatting; and so on…
  - Can be programmed independently
  - Simplifies the development of large applications

Address Space

- Contains all states necessary to run a program
  - Code, data, stack
  - Program counter
  - Register values
  - Resources required by the program
  - Status of the running program
- **A mechanism to protect one app from crashing another app**

Process

- An address space + at least one thread of execution
  - Address space offers protection among processes
  - Threads offer concurrency
- A fundamental unit of computation

Process =? Program

- ***Program***: a collection of statements in C or any programming languages
- Process: a running instance of the program, with additional states and system resources

- Two processes can run the same program
  - The code segment of two processes are the same program

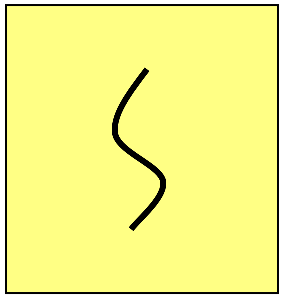
- A program can create multiple processes
  - Example: gcc, chrome

Real-life Analogy?

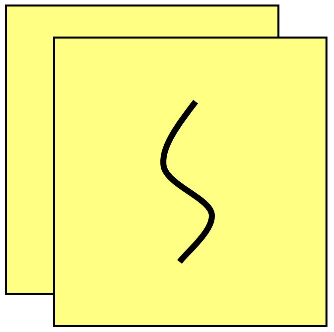
- Program: a recipe
- Process: everything needed to cook
    - e.g., kitchen
- Two chefs can cook the same recipe in different kitchens
- One complex recipe can involve several chefs

Some Definitions

- ***Uniprogramming***: running one process at a time

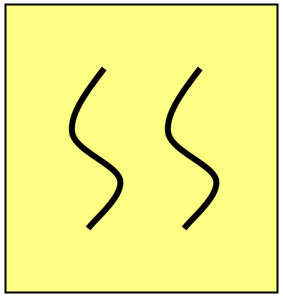


- ***Multiprogramming***: running multiple processes on a machine



Some Definitions

- *Multithreading*: having multiple threads per address space (threads share the same address space)
- *Multiprocessing*: running programs on a machine with multiple processors
- *Multitasking*: a single user can run multiple processes



Classifications of OSes

	Single address space	Multiple address spaces
Single thread	MS DOS, Macintosh	Traditional UNIX
Multiple threads	Embedded systems	Windows, iOS

Threads & Thread Control Block

- A thread owns a ***thread control block***
  - Execution states of the thread
  - The status of the thread
    - Running or sleeping
  - Scheduling information of the thread
    - e.g., priority

Threads & Dispatching Loop

- Threads are run from a ***dispatching loop***
  - Can be thought as a per-CPU thread
  - LOOP
    - Run thread
    - Save states
    - Choose a new thread to run ← ***Scheduling***
    - Load states from a different thread

Context switch (Save states; Choose a new thread to run; Load states from a different thread)

Simple? Not quite…

- How does the dispatcher regain control after a thread starts running?
- What states should a thread save?
- How does the dispatcher choose the next thread?

How does the dispatcher regain control?

- Two ways:
  1. Internal events (“Sleeping Beauty”)
     - Yield—a thread gives up CPU voluntarily
       - A thread is waiting for I/O
       - A thread is waiting for some other thread
  2. External events
     - Interrupts—a complete disk request
     - Timer—it’s like an alarm clock

What states should a thread save?

- Anything that the next thread may trash before a context switch
  - Program counter
  - Registers
  - Changes in execution stack

How does the dispatcher choose the next thread?

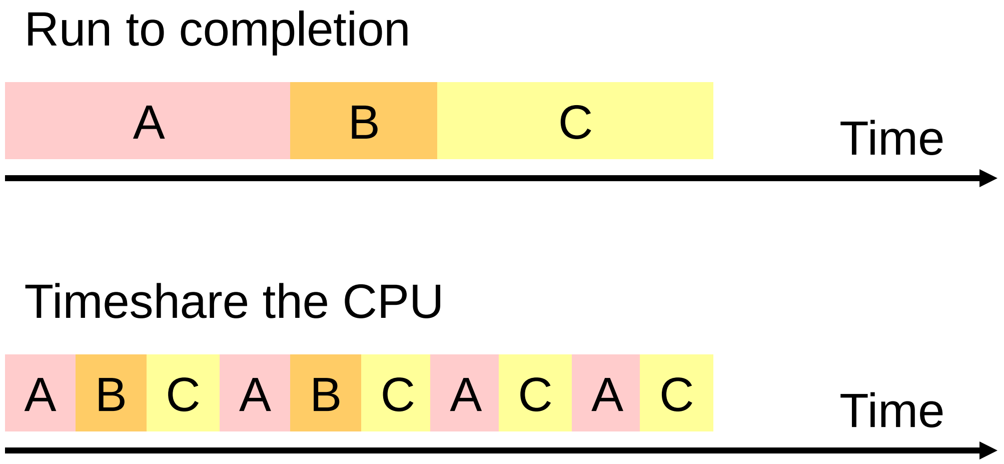
- The dispatcher keeps a list of threads that are ready to run
- If no threads are ready
  - Dispatcher just loops
- If one thread is ready
  - Easy

How does the dispatcher choose the next thread?

- If more than one thread are ready
  - We choose the next thread based on the scheduling policies
  - Examples
    - FIFO (first in, first out)
    - LIFO (last in, first out)
    - Priority-based policies

How does the dispatcher choose the next thread?

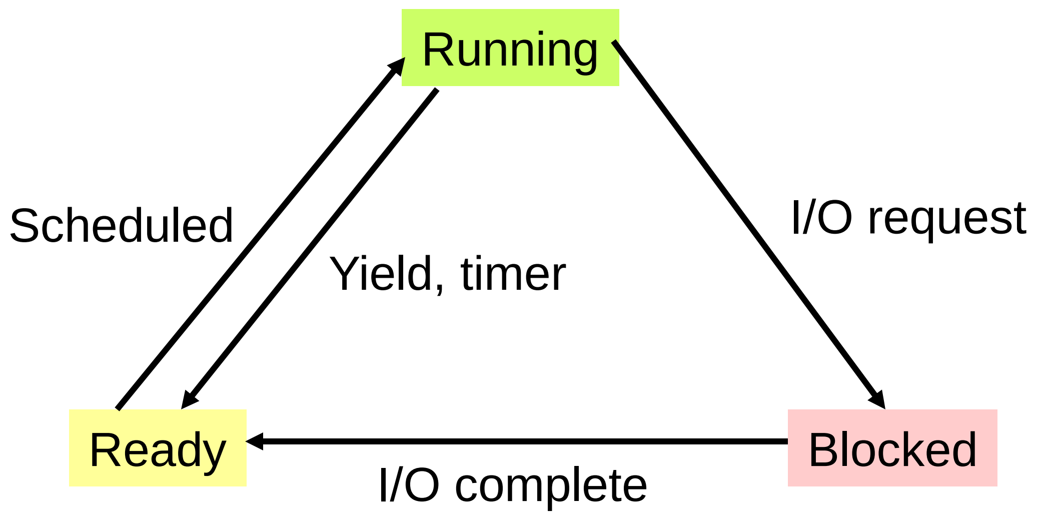
- Additional control by the dispatcher on how to share the CPU
  - Suppose we have three threads



Per-thread States

- Each thread can be in one of the three states
  1. *Running*: has the CPU
  2. *Blocked*: waiting for I/O or another thread
  3. *Ready to run*: on the ready list, waiting for the CPU

Per-thread State Diagram


$$\begin{tikzcd}
 & \text{Running} \arrow[dl, bend left, "\text{Yield, timer}"] \arrow[dr, "\text{I/O request}"] & \\
\text{Ready} \arrow[ur, bend left, "\text{Scheduled}"] & & \text{Blocked} \arrow[ll, "\text{I/O complete}"]
\end{tikzcd}$$

Important

For Multi-core Machines

- Each core has a dispatcher loop
  - Decide which thread will execute next
- One core has a global dispatcher loop
  - Decide which core to execute a thread

Parallelism vs. Concurrency

- *Parallel* computations
  - Computations can happen at the same time on separate cores
- *Concurrent* computations
  - One unit of computation does not depend on another unit of computation
    - Can be done in parallel on multiple cores
    - Can time share a single core (not parallel)

Real-life Example

- Two hands are playing piano in parallel (not concurrently)
  - Notes from left and right hands are dependent on each other
- Two separate groups singing ‘row row row your boat’ concurrently (and in parallel)

Amdahl’s Law

- Identifies potential performance gains from adding cores
  - P = % of program that can be executed in parallel
  - N = number of cores
- $speedup \le \frac{1}{(1-P)+\frac{P}{N}}$

Important

Amdahl's Law

- Example
  - P = 75% of program that can be executed in parallel
  - N = 2 cores
- $speedup \le \frac{1}{(1-0.75) + \frac{0.75}{2}}$ = 1.6

Takeaways

- OS is a state machine.
- Process, Thread, Address Space
- Thread Dispatch Loop
- Amdahl’s Law