Module 07: Introduction to Operating Systems and Program Execution

Unit 1: Functionalities of OS and the Concept of Process

ITSC 2181 Introduction to Computer Systems
College of Computing and Informatics

UNIVERSITY OF NORTH CAROLINA
CHARLOTTE

Module Overview: Introduction to Operating Systems and Program Execution

- Provides a foundational understanding of what the role of an operating system is in managing hardware resources and running programs.
- Exploring key concepts such as system calls, interrupts, and processes
- Explores the process lifecycle, and how Unix-based operating systems create and manage processes

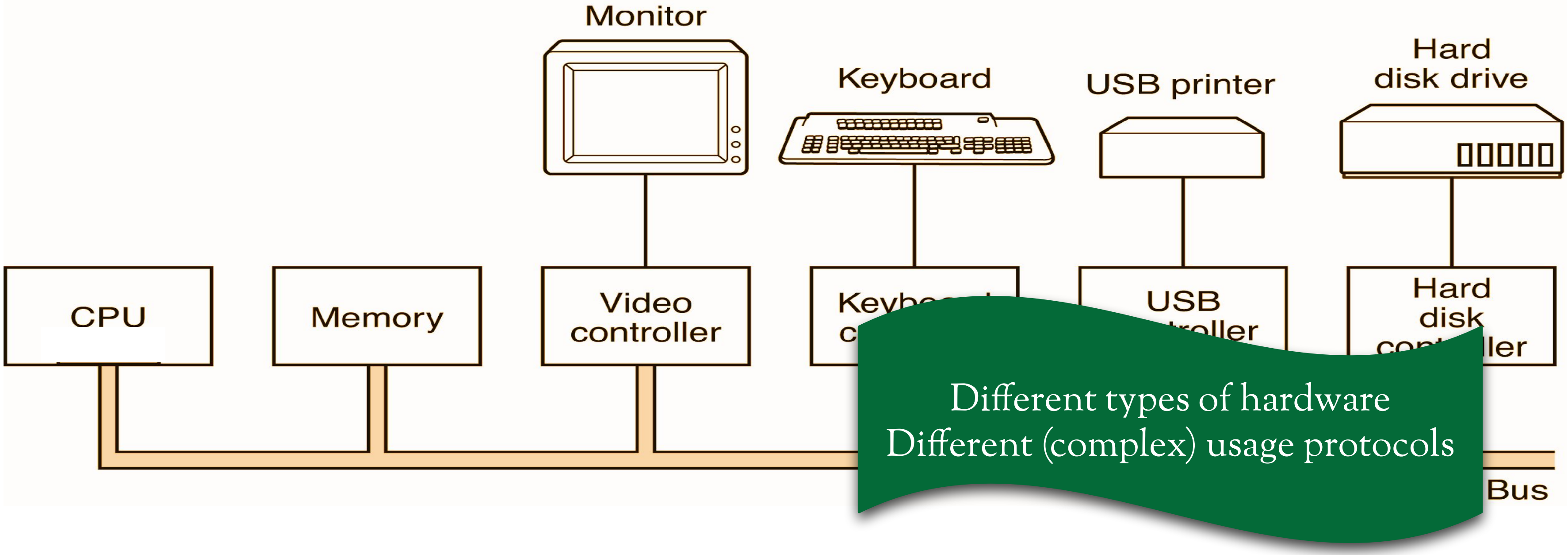
Why

- Equips students with essential knowledge for their academic journey and future careers in the field of computing.
- Gain foundational knowledge about how programs are executed on a computer system.

Objectives

- Check the Learning Guide of each Unit of the Module
  - ☛ **Unit 1: Functionalities of OS and the Concept of Process**
  - Unit 2: Process Lifecycle and Process Creation
  - Unit 3: Thread and Memory of a Process (If time permits)

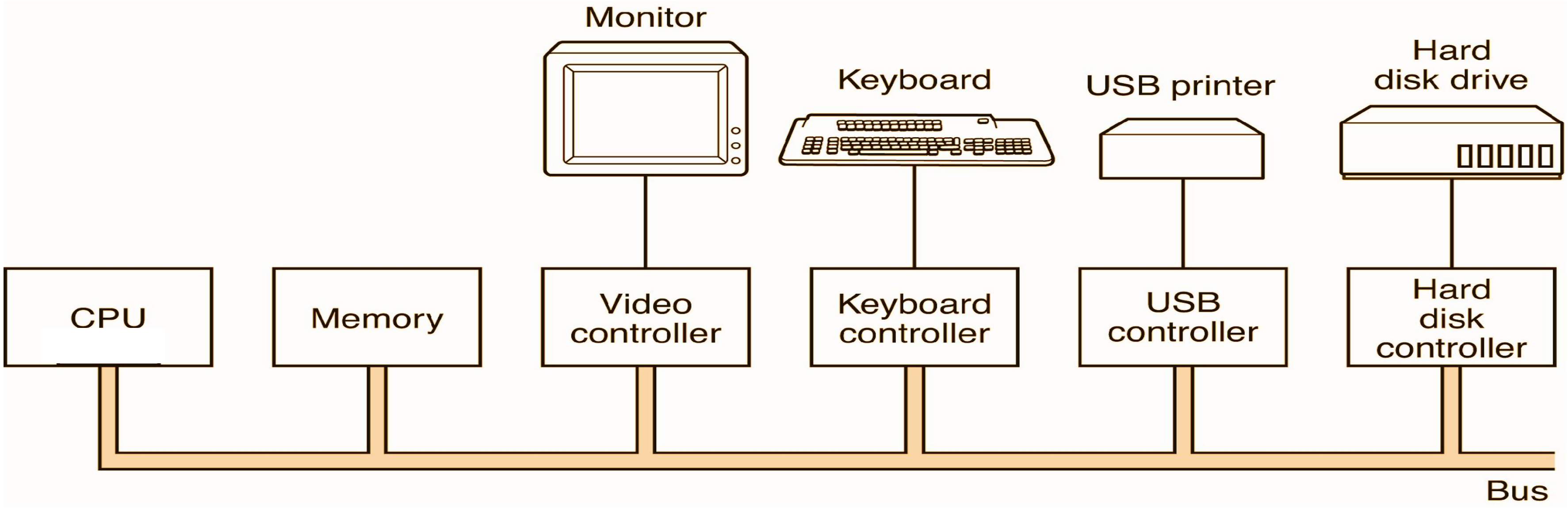
Managing Hardware Complexity



Different types of hardware
Different (complex) usage protocols

- Low level hardware controller detail too complicated for application programs / users
- Hardware state can get messed up through use of incorrect protocols

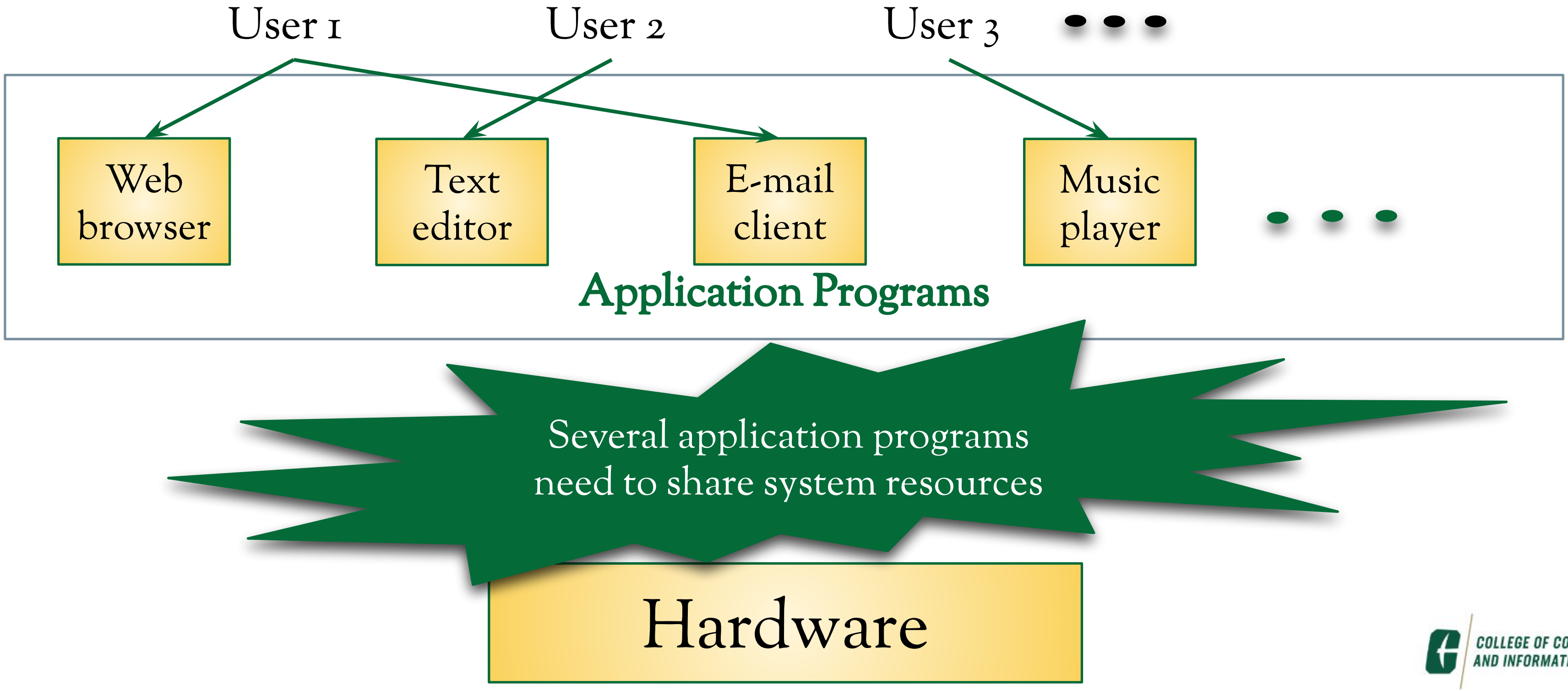
Managing Hardware Complexity



Abstraction

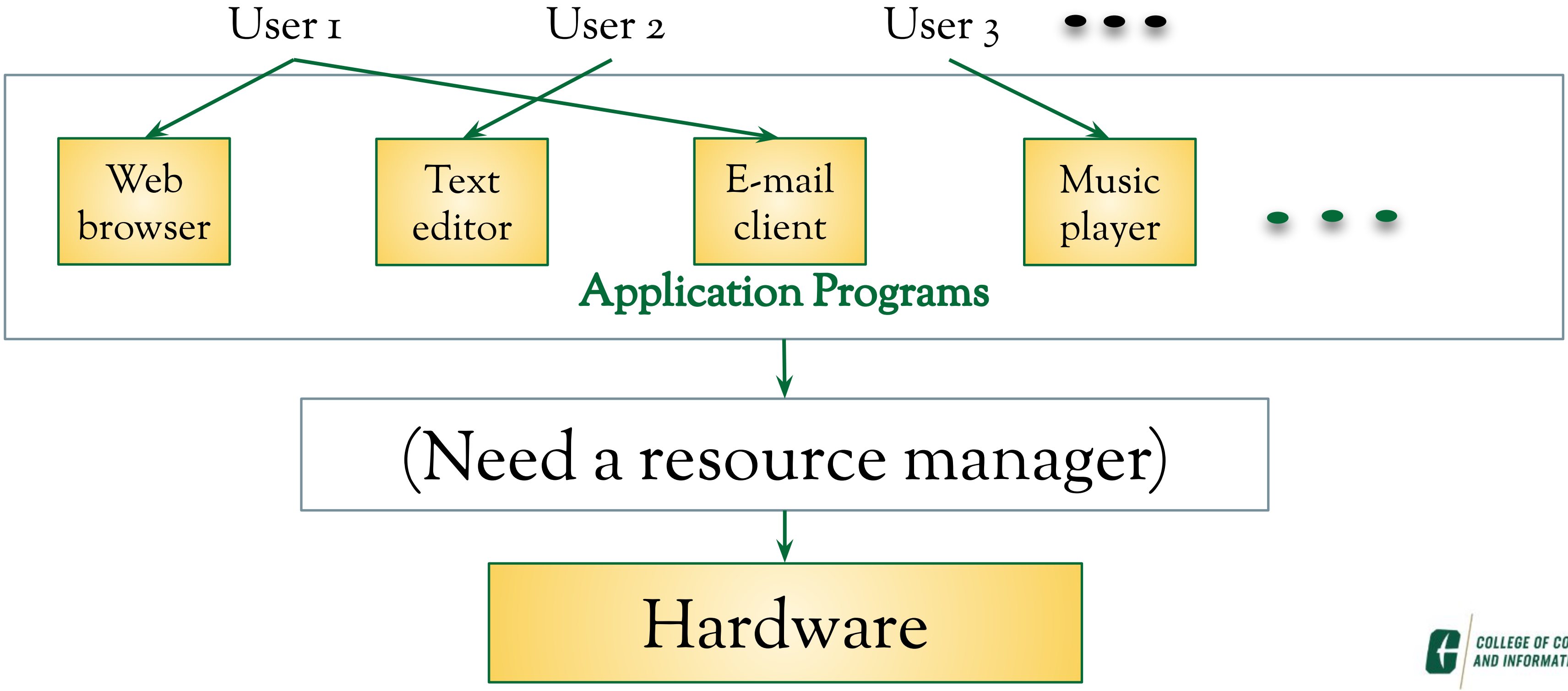
- Need special software that
  - Knows how to interact with hardware controllers
  - Provides simpler external interface to application programs / users

Managing Shared Resources



COLLEGE OF COMPUTING AND INFORMATICS

Managing Shared Resources

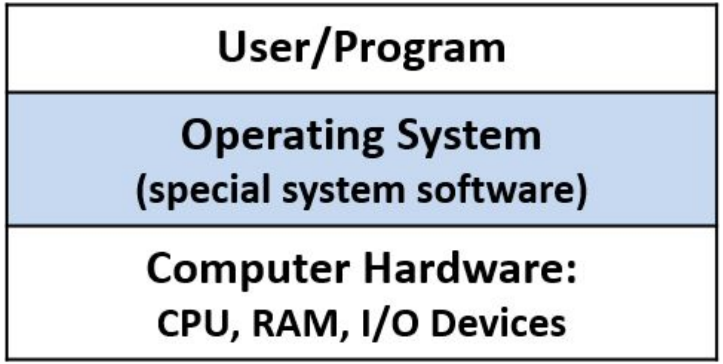


Operating System (*OS*)

- Special system software between computer hardware & application programs

- Makes computer hardware easy to use via *abstractions*
    - Ensures safety (protects hardware, prevents & handles errors)
    - May provide multiple levels of abstraction

- Acts as a *resource manager*
    - Allows multiple application programs & users to share resources
    - Ensures fair, efficient & protected access to resources



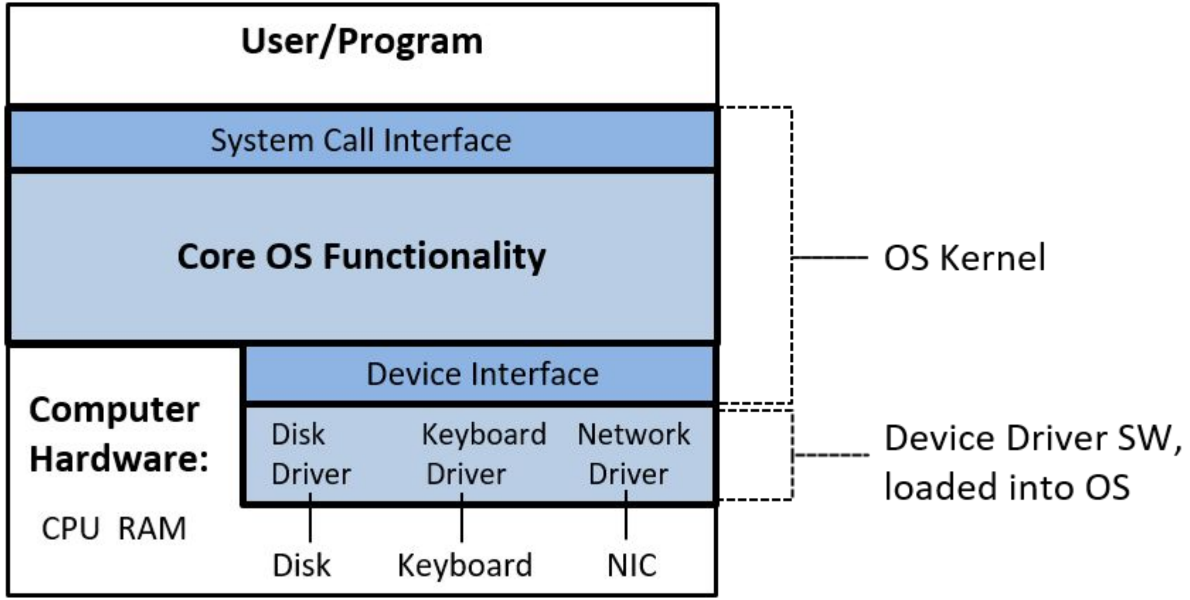
User/Program

Operating System
(special system software)

Computer Hardware:
CPU, RAM, I/O Devices

Operating System (*OS*)

- OS ***kernel*** implements core functionality and provides:
  - Programming interface for users of the system → *system call interface*
  - Interface for interacting with hardware devices (e.g., disk driver, network driver,...) → *device interface*



To manage complexity…

- OS design typically separates *mechanism* from *policy*
  - I.e., separates *how* from *what*/*when*/*which*

- *Mechanism*
  - Data structures/operations used to implement abstraction/service

- *Policy*
  - Procedures/rules to guide selection of action from possible alternatives

Protection

Need structures/*mechanisms* that ensure:

Protection of hardware (CPU, memory, I/O devices)

Protection between multiple applications/users

Protection

System operation split into two *modes*

User mode	*Kernel* (monitor / supervisor / system) mode
● Execution on behalf of user → *protected* mode ● No direct access to hardware ● Can execute only *subset* of instructions ● Can access only *restricted* memory areas	● Execution on behalf of operating system → *privileged* mode ● Complete access to hardware ● Can execute *any* instruction ● Can access *any* memory area

Hardware support for modes

- System maintains *mode bit* indicating current mode
- If privileged operation is attempted in user mode
  - It must be prevented from taking place
  - System must be notified
- These are achieved using an *exception*
  - *Synchronous **interrupt*** → type of interrupt caused by current instruction

Interrupt

- *External event that causes change in flow of current execution*
- Sequence of activities…
  - CPU executing instructions
  - *Interrupt* occurs
  - CPU
    - Finishes current instruction
    - See that there’s an interrupt
    - Saves current state
    - Services interrupt
    - Resumes normal activity

Consequence of modes

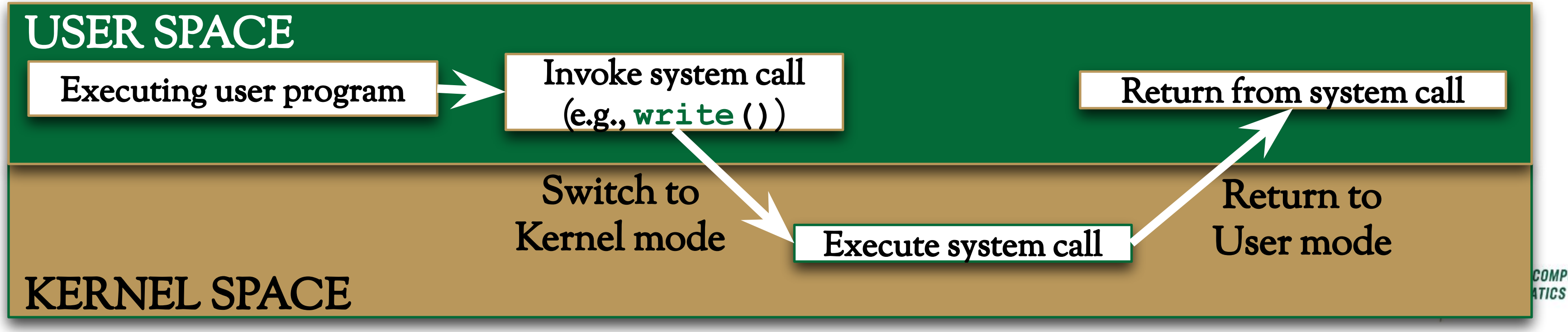
- Need special *mechanism* for applications to access OS services

- **System call** is the answer
  - *Interface* between running programs and OS
  - Provides *controlled entry* into kernel for privileged operation
  - Makes sure access is performed in specific *well defined* way

System Call

- Causes system to switch to *kernel* mode
  - ***Trap*** – a kind of *synchronous interrupt* – used to achieve this
  - **In general, any interrupt causes switch to kernel mode**

- Typically invoked using assembly language instructions
  - Systems generally provide *library* or *API* to invoke system call
  - Library function serves as wrapper for actual system call



USER SPACE

Executing user program

Invoke system call (e.g., `write()`)

Return from system call

Switch to Kernel mode

Execute system call

Return to User mode

KERNEL SPACE

Services provided by the OS

- Program execution
  - Load program & data, schedule and execute program
- Memory management
  - Manage main memory; ensure programs can’t mess with other programs’ memory
- File management
  - Create, read, write files
  - Access control for files
- I/O management
  - Safe and controlled access to I/O devices

Services provided by the OS

- Information maintenance
  - Get/set system time/date
- Communication services
  - Communication b/w programs
- User management
  - Authentication and access for users of system
- Error management
  - Detect & handle errors
- Accounting services
  - Collect statistics, monitor performance

Program Execution

As we know…

- Computer system essentially used to execute/run programs
    - May run several different programs concurrently
        - E.g., e-mail client, browser, editor, music player
    - May run multiple instances of same program concurrently
        - E.g., Multiple instances of browser, editor
- Need some way to represent running programs internally

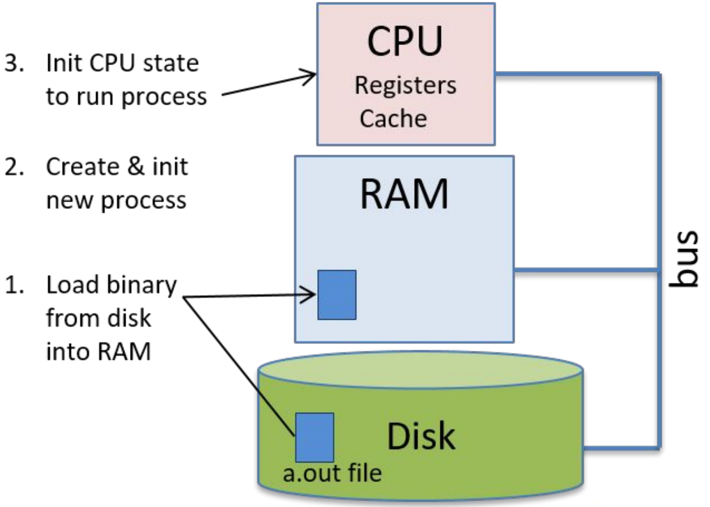
Process

- Abstraction for a *running program*
  - Represents an *activity* of some kind – hence the name!
  - Used by OS to manage concurrently running programs
- A *process* is not equivalent to a *program*
  - TextEdit → program
  - Specific running instance of TextEdit → process
- More to *process* than just the program code
  - Also includes program data and execution **context**
  - Owns resources (e.g., memory)

Running a program

- When a program needs to be run
  - OS loads program binary into main memory (if it’s not already there)
  - Creates a process to represent the new program instance
    - New process is assigned its own **unique ID**

Note: *If the same program is run multiple times, separate processes are created to represent each instance of the program.*


3. Init CPU state to run process
2. Create & init new process
1. Load binary from disk into RAM
CPU
Registers
Cache
RAM
Disk
a.out file
bus