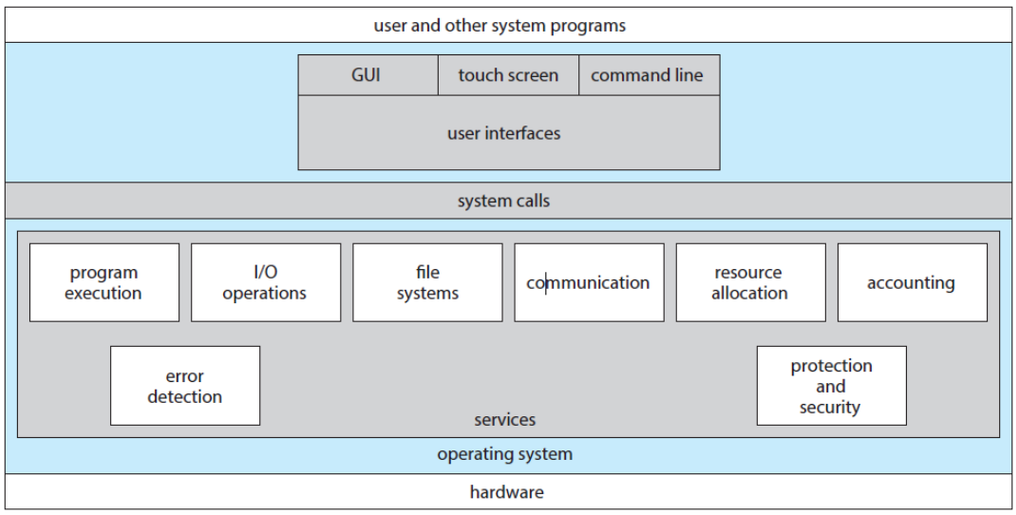
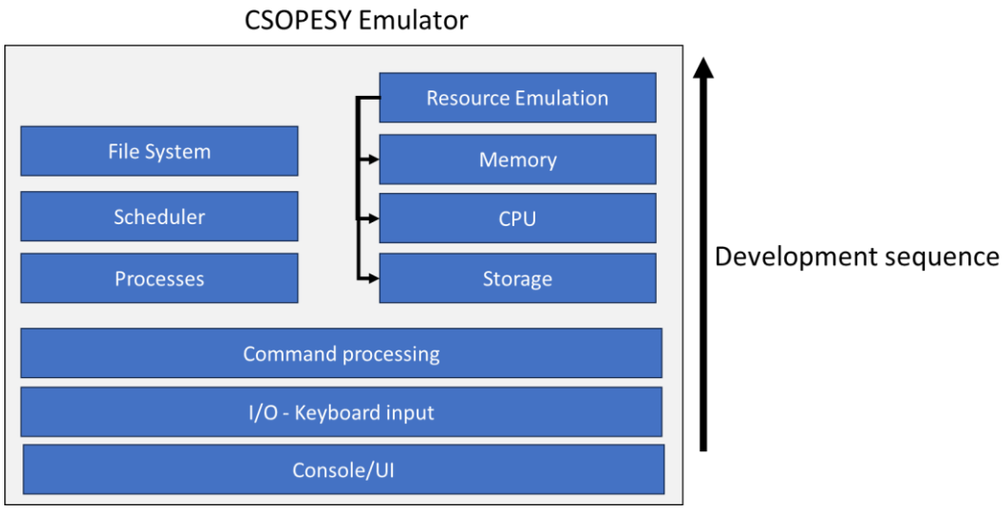
Background

- To create our emulator from scratch, we need to borrow fundamental OS structures → serve as templates and building blocks for our OS emulator.

Operating system services

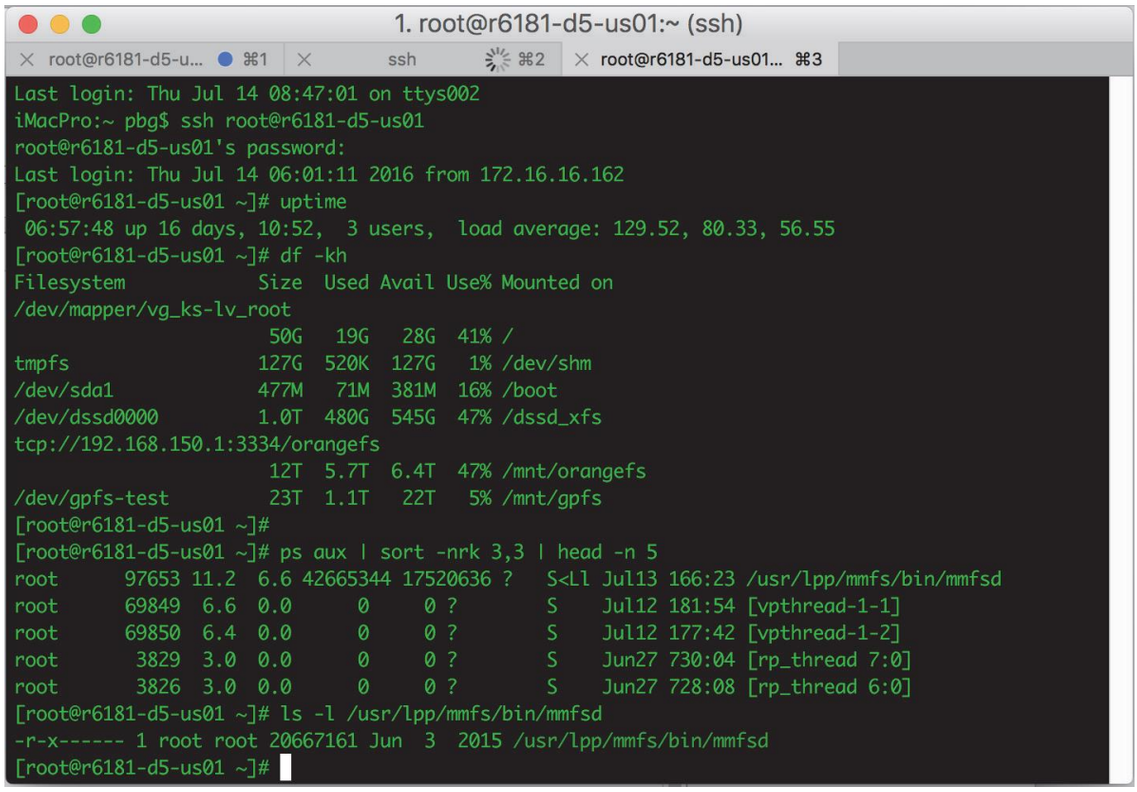


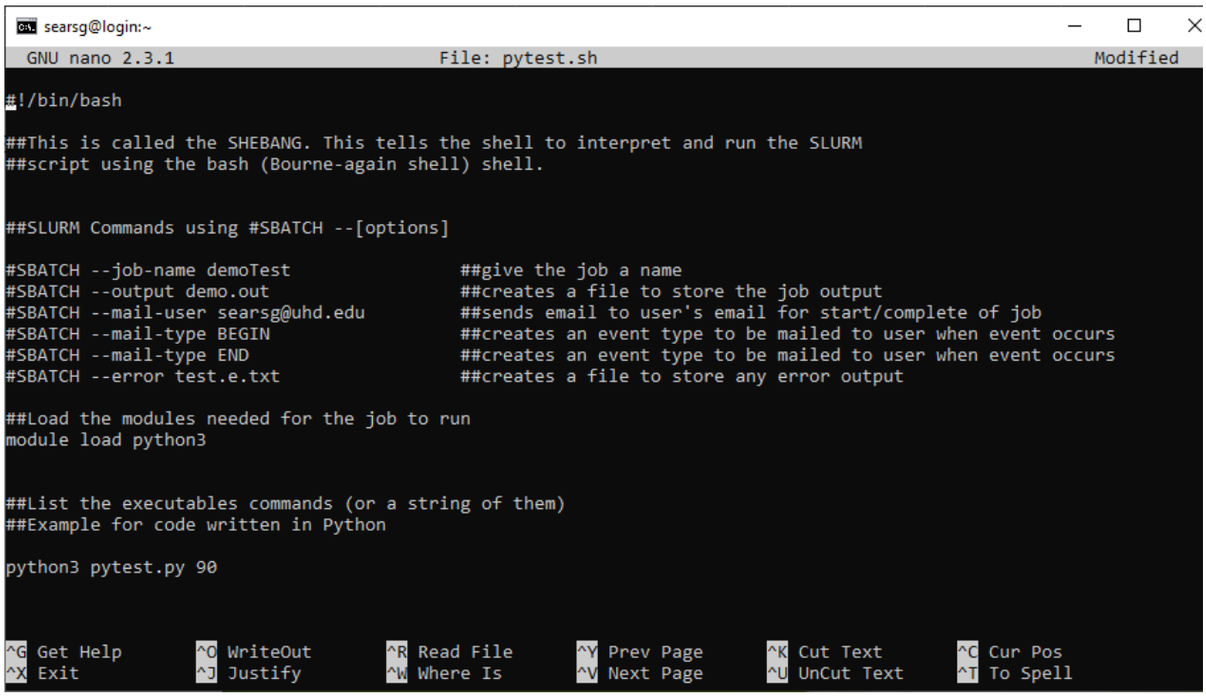


- We show a side-by-side comparison of a typical view of commercial OS services available, and what services we need to develop/emulate.

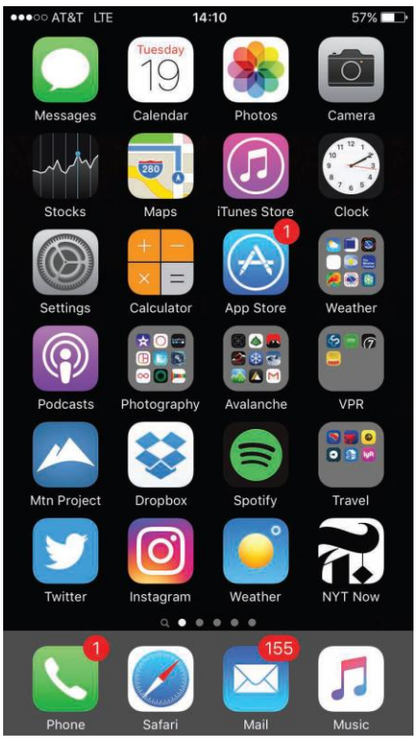
User interface

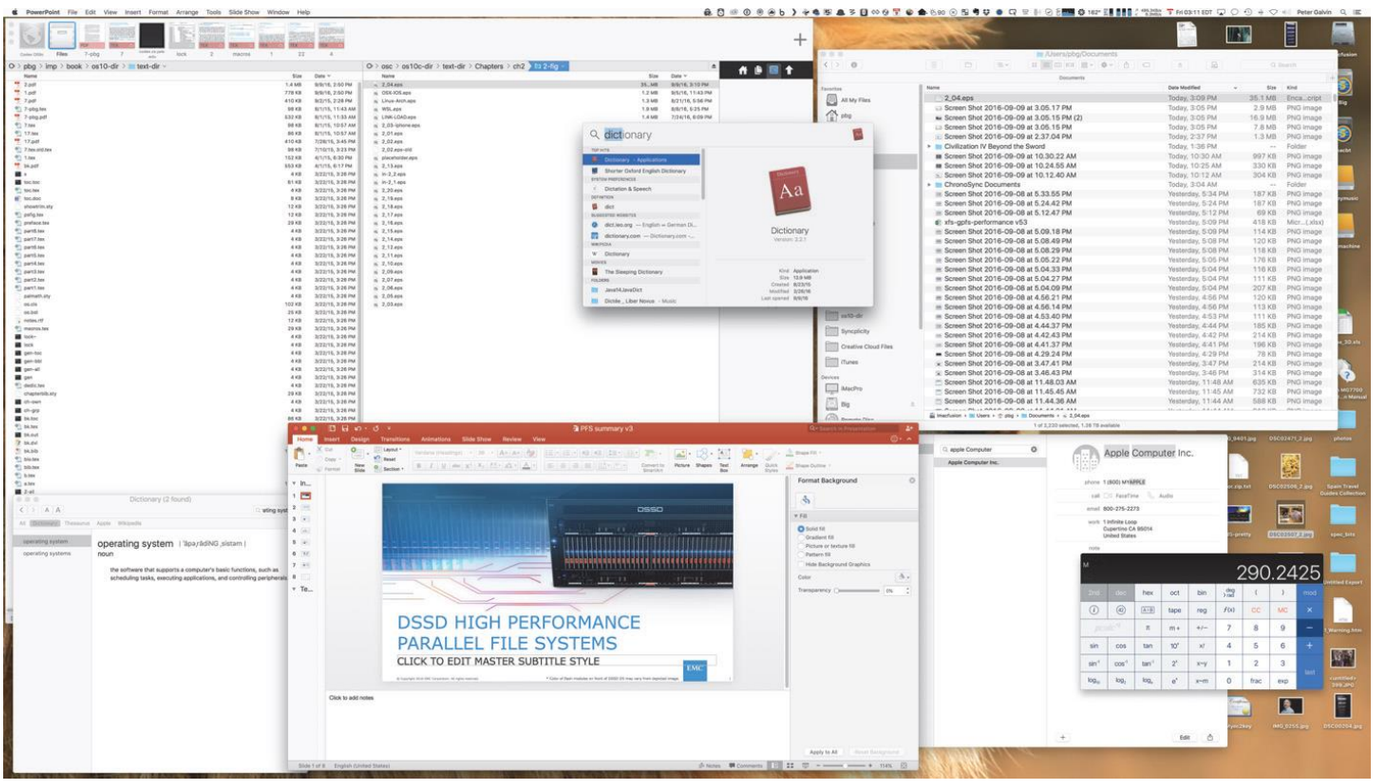
- Refers to display interface and reading of peripheral inputs → keyboard/mouse? Touch screen?
- Early OS started with command-line interfaces (CLI), which uses text commands and keyboard for entering them → what our OS emulator will look like.





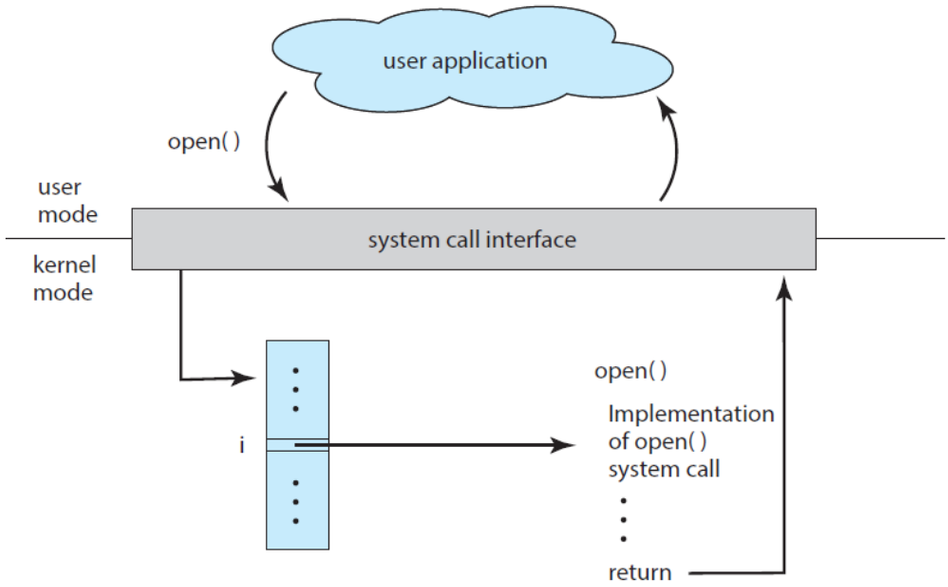
Other interfaces:





Since we will be pursuing a **command-line-interface style emulator**, essential features are:

- Command interpreter
  - Most operating systems, including Linux, UNIX, and Windows, treat the command interpreter as a special program that is running when a process is initiated or when a user first logs on (on interactive systems).
  - Virtual machines that could be rented on the cloud (e.g. Google VM) merely provides a Linux shell with Python and ML libraries pre-installed.
  - Developing a command interpreter require processing of commands received from keyboard input.
    - Maintains a list of commands recognizable: mkdir, cd, tmux, etc.
    - Tokenization of command. E.g. cd "root/example/folder1/subfolder1" → means of splitting the String into tokens using "/" as the separator → checking and validating each directory as the pointer moves to "subfolder1"
- System calls
  - Are libraries/interfaces developed that exposes services in an operating system. E.g. print function, read/write of files.
  - Also refers to system APIs. Several programming languages have system APIs that are automatically included/imported in every program. E.g. print/println, class/object management.

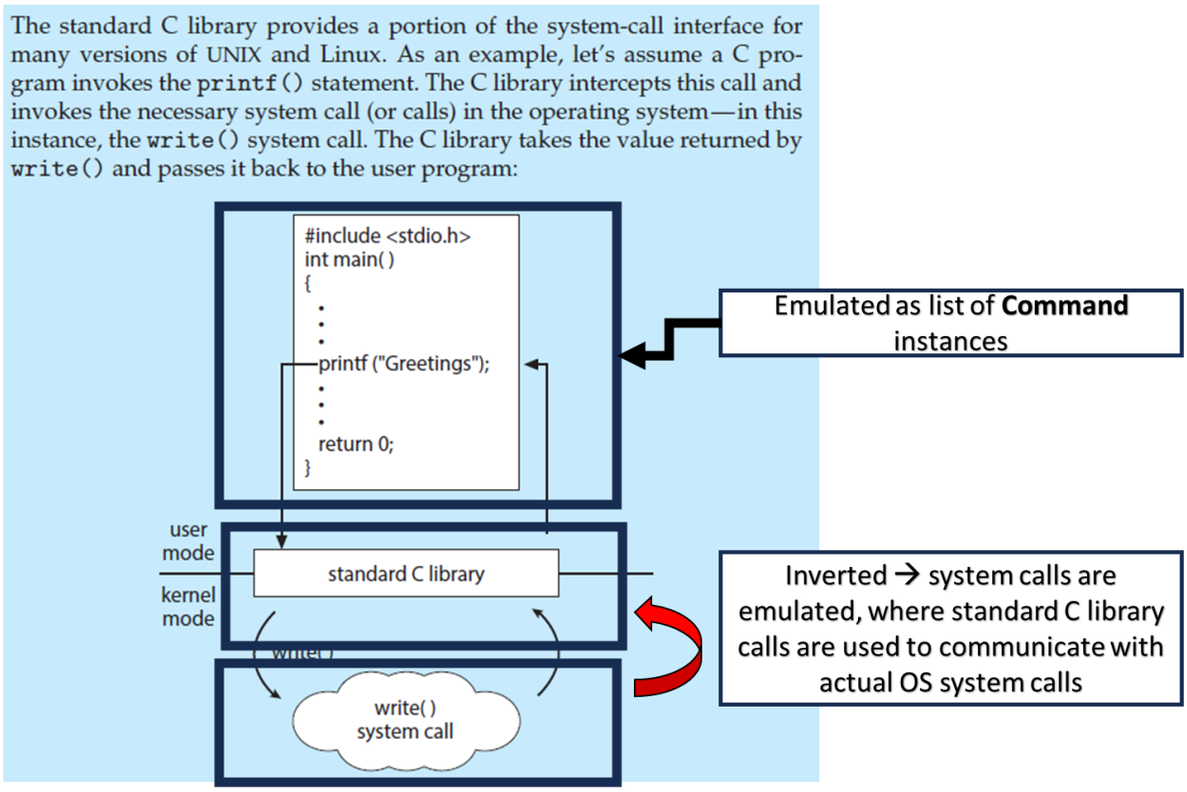


- The following are example system calls:
  - Process control
    - create process, terminate process
    - load, execute
    - get process attributes, set process attributes
    - wait event, signal event
    - allocate and free memory
  - File management
    - create file, delete file
    - open, close
    - read, write, reposition
    - get file attributes, set file attributes
  - Device management
    - request device, release device
    - read, write, reposition
    - get device attributes, set device attributes
    - logically attach or detach devices
  - Information maintenance
    - get time or date, set time or date
    - get system data, set system data
    - get process, file, or device attributes
    - set process, file, or device attributes
  - Communications
    - create, delete communication connection
    - send, receive messages
    - transfer status information
    - attach or detach remote devices
  - Protection
    - get file permissions
    - set file permissions

We show here a table of how we will develop this in our CSOPESY emulator.

System call	Commercial OS	CSOPESY emulator

Process control	X	Processes are simulated. Each process will have a pre-defined set of simple commands to execute → mock I/O read/write, print command, delay.
File management	X	File directory is simulated. All files are stored inside a single .csopesy file.
Device management	X	Simulated. Devices and contexts are configured.
Information maintenance	X	Simulated, except no time-date support.
Communications	X	Not supported.
Protections	X	No protection feature



The figure above shows what a typical compiled C program executed in a commercial OS. How does it work in our emulator?

- C program → emulated process with list of commands
- System calls and its associated library is inverted → system calls are emulated, where standard C++ library calls are used to communicate with actual OS system calls.
- In other words, we have one "emulation" layer containing features that essentially make up a simple OS shell → process scheduler, memory management, file management, I/O and display interface. The emulation layer "simulates/translates/restricts" certain process commands as we create an OS that performs its duties. Recall operating system as coordinator.

OTHER OPERATING SYSTEM SERVICES

Program execution

- Basic idea: To run a program, it must be loaded into memory, and a register dictates the flow of instructions to be executed. How our programs going to be executed?
- As mentioned earlier, our representation of a program/process only consists of placeholder commands.
- The following will be devised: command flow (sequence of executation), process scheduling, resource and memory allocation.

I/O operations

- A running program may require I/O, which may involve a file or an I/O device. For specific devices, special functions may be desired (such as reading from a network interface or writing to a file system).
- For efficiency and protection, users usually cannot control I/O devices directly → e.g. directly controlling the printer head.
- In our OS emulator, request for a device/driver will be simulated → simply have a table of boolean/int flags, indicating the availability of arbitrary devices (e.g. available/unavailable, 3 devices, 1 device, etc).

File-system manipulation

- Programs would often need access to certain files and directories.
- They also need to create and delete themby name, search for a given file, and list file information.

- Some operating systems include permissions management to allow or deny access to files or directories based on file ownership.
- Emulator: Directory systems will be emulated → N editable files are "combined" together into a single .csopesy file.
- The .csopesy file resembles your disk storage. Multiple .csopesy files resembles multiple disk storages. E.g. C:/, D:/, E:/

Communication

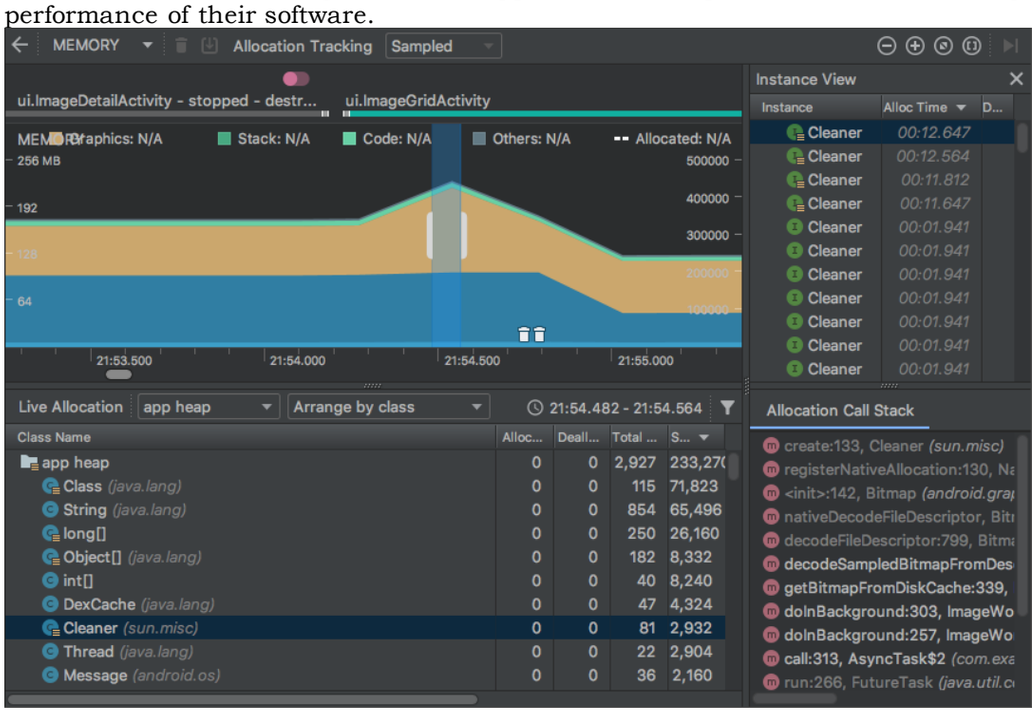
- Processes may exchange information with other processes. Scenario: Consider mobile OS (iOS/Android) → Active programs with running background services. Facebook application (foreground), sending notifications (background service).
- Some OS support concepts of shared memory across different processes. Example: Windows. Developers code their programs to access the shared memory pool: https://learn.microsoft.com/en-us/windows/win32/memory/creating-named-shared-memory
- Emulator: There will be no support for communication services.

Resource allocation

- When there are multiple processes running at the same time, resources must be allocated to each of them.
- The operating system manages many different types of resources. Some (such as CPU cycles, main memory, and file storage) may have special allocation codes, whereas others (such as I/O devices) may have much more general requests and release codes.
- For instance, in determining how best to use the CPU, operating systems have CPU-scheduling routines that take into account the speed of the CPU, the process that must be executed, the number of processing cores on the CPU, and other factors.
- There may also be routines to allocate printers, USB storage drives, and other peripheral devices.
- There must be a mechanism in dealing with possible deadlocks on resources → accessing a common resource across all processes.
- Emulator: The bulk of the emulator design would be on resource allocation.

Logging

- We want to keep track of which programs use how much and what kinds of computer resources.
- Most OS have a UI of usage statistics. Application developers also have means of profiling the performance of their software.



- Emulator: Basic logging must be supported. E.g. Viewing of running processes, checking resource availability, CPU and memory utilization, printing messages.

Protection and security

- Protection involves ensuring that all access to system resources is controlled. Some API calls must be restricted or granted privileges. E.g. Android application requesting for permission to access GPS.
- Security of the system would typically require each user to authenticate. Thus, OS must have user account support.
- Other security must be in place for programs and resources. E.g. Do not allow viewing of machine code.
- Emulator: We will not be implementing protection and security features.

Activities

- Designing support for your **user interface - console states**. Recall that you created your placeholder commands. It's time to recognize these commands and create a new console layout for each command.
    - The drawing must only be done in the same console window.
- Hint on console state: There must be a mechanism to store different console layouts → refer to ConsoleManager for implementation.
- Using the reference console manager, provide proposed pseudocodes of its implementation. You may add additional functions. Be prepared to be called in class to present your proposed solution.

Some hints on designing console states. Header file of the console manager class is seen below.

```
#pragma once
#include <memory>
#include <vector>
#include "AConsole.h"
#include <unordered_map>
#include <Windows.h>
#include "TypedefRepo.h"

const String MAIN_CONSOLE = "MAIN_CONSOLE";
const String MARQUEE_CONSOLE = "MARQUEE_CONSOLE";
const String SCHEDULING_CONSOLE = "SCHEDULING_CONSOLE";
const String MEMORY_CONSOLE = "MEMORY_CONSOLE";

class ConsoleManager
{
public:
    typedef std::unordered_map<String, std::shared_ptr<AConsole>> ConsoleTable;

    static ConsoleManager* getInstance();
    static void initialize();
    static void destroy();

    void drawConsole() const;
    void process() const;
    void switchConsole(String consoleName);
    void returnToPreviousConsole();
    void exitApplication();
    bool isRunning() const;

    HANDLE getConsoleHandle() const;

    void setCursorPosition(int posX, int posY) const;

private:
    ConsoleManager();
    ~ConsoleManager() = default;
    ConsoleManager(ConsoleManager const&) {}; // copy constructor is private
    ConsoleManager& operator=(ConsoleManager const&) {}; // assignment operator is private*/
    static ConsoleManager* sharedInstance;

    ConsoleTable consoleTable;
    std::shared_ptr<AConsole> currentConsole;
    std::shared_ptr<AConsole> previousConsole;

    HANDLE consoleHandle;
    bool running = true;
};
```

The constructor of the console manager is seen below:

```cpp
ConsoleManager::ConsoleManager()
{
    this->running = true;

    //initialize consoles
    this->consoleHandle = GetStdHandle(STD_OUTPUT_HANDLE);

    const std::shared_ptr<MainConsole> mainConsole = std::make_shared<MainConsole>();
    const std::shared_ptr<MarqueeConsole> marqueeConsole = std::make_shared<MarqueeConsole>();
    const std::shared_ptr<SchedulingConsole> schedulingConsole = std::make_shared<SchedulingConsole>();
    const std::shared_ptr<MemorySimulationConsole> memoryConsole = std::make_shared<MemorySimulationConsole>();

    this->consoleTable[MAIN_CONSOLE] = mainConsole;
    this->consoleTable[MARQUEE_CONSOLE] = marqueeConsole;
    this->consoleTable[SCHEDULING_CONSOLE] = schedulingConsole;
    this->consoleTable[MEMORY_CONSOLE] = memoryConsole;

    this->switchConsole(MAIN_CONSOLE);

}
```