COMP 4736
Introduction to Operating Systems

01. Computer Architecture and Overview

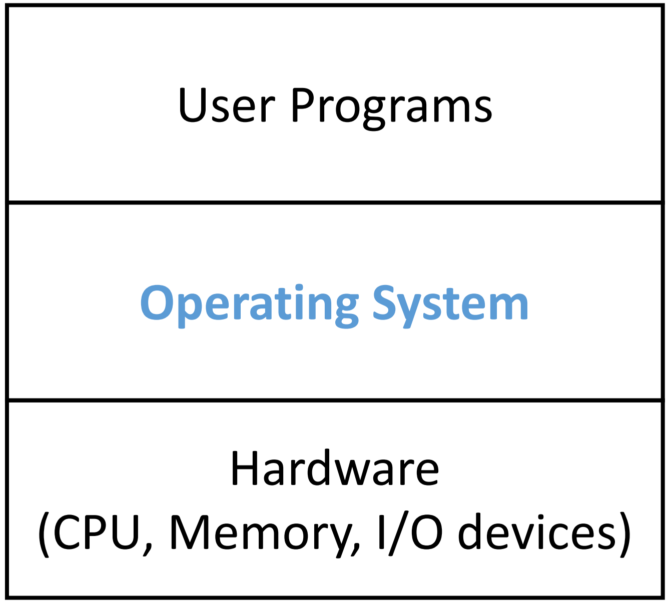
(OSTEP Ch. 2 & 4)

Ch. 02. Introduction to Operating Systems

(OSTEP Ch 2)

What is an Operating System?

- Middleware between **user programs** and **system hardware**.
- Manages hardware: CPU, main memory, I/O devices (disk, network card, mouse, keyboard, etc.)



What happens when a program runs?

(Recall: a program is a sequence of instructions and data.)

- The processor **fetches** an instruction from memory, …
- **decodes** it (i.e., figures out which instruction this is), …
- and **executes** it (e.g., add two numbers, access memory, check a condition, jump to a function, etc.).

- The processor then moves to the **next instructions**. It repeats until program completes.

Operating Systems (OS)

- Responsible for
  - Making it easy to **run** programs.
  - Allowing programs to **share** memory.
  - Enabling programs to **interact** with devices.

OS is in charge of making sure the system operates **correctly** and **efficiently**.

Virtualization

- The OS takes a **physical** resource and transforms it into a **virtual** form of itself.
  - Physical resource: processor, memory, disk, etc.
- The virtual form is more **general**, **powerful** and **easy-to-use**.
- Sometimes, we refer to the OS as a **virtual machine**.

System Call

- **System call** allows user to **tell the OS** what to do.
- The OS provides some interfaces (**APIs**, **standard library**).
- A typical OS exports a few hundred **system calls**.
    - Run programs
    - Access memory
    - Access devices

OS as a Resource Manager

- The OS **manages resources** such as CPU, memory and disk.
- The OS allows
  - Many programs to run → Sharing the **CPU**
  - Many programs to concurrently access their own instructions and data → Sharing **memory**
  - Many programs to access devices → Sharing **disks**

Virtualizing the CPU

- The system has a very large number of virtual CPUs.
- Turning a single CPU into a **seemingly infinite number** of CPUs.
- Allowing many programs to **seemingly run at once**
  → **Virtualizing the CPU**

Virtualizing the CPU (cpu.c)

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <assert.h>
#include "common.h"

int
main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "usage: cpu <string>\n");
        exit(1);
    }
    char *str = argv[1];
    while (1) {
        Spin(1);
        printf("%s\n", str);
    }
    return 0;
}
```

Virtualizing the CPU (Result 1)

```
prompt> gcc -o cpu cpu.c -Wall
prompt> ./cpu "A"
A
A
A
A
ˆC
prompt>
```

- Runs forever.
- We halt the program by pressing `Ctrl+C`.

Virtualizing the CPU (Result 2)

```
prompt> ./cpu A & ./cpu B & ./cpu C & ./cpu D &
[1] 7353
[2] 7354
[3] 7355
[4] 7356
A
B
D
C
A
B
D
C
A
...
```

Even though we have only **one processor**, all **four of programs** seem to be **running at the same time**!

Virtualizing the CPU

- Running multiple programs at once raises new questions.
  - E.g., if two programs want to run at the same time, which **should** run?
- It is handled by **policies** of the OS.
- We will study them as we learn about the basic **mechanisms** that OS implement, as a **resource manager**.

Virtualizing Memory

- The **physical memory** is an array of bytes.
- A program keeps all of its data structures in memory.

- **Read** memory (load):
  - Specify an address to be able to access the data
- **Write** memory (store):
  - Specify the data to be written to the given address

Virtualizing Memory (mem.c)

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include "common.h"

int
main(int argc, char *argv[])
{
    int *p = malloc(sizeof(int));                      // a1
    assert(p != NULL);
    printf("(%d) address pointed to by p: %p\n",
        getpid(), p);                                  // a2
    *p = 0;                                            // a3
    while (1) {
        Spin(1);
        *p = *p + 1;
        printf("(%d) p: %d\n", getpid(), *p);          // a4
    }
    return 0;
}
```

Virtualizing Memory (Result 1)

```
prompt> ./mem
(2134) address pointed to by p: 0x200000
(2134) p: 1
(2134) p: 2
(2134) p: 3
(2134) p: 4
(2134) p: 5
ˆC
```

- The newly allocated memory is at address 0x200000.
- It updates the value and prints out the result.

Virtualizing Memory (Result 2)

```
prompt> ./mem &; ./mem &
[1] 24113
[2] 24114
(24113) address pointed to by p: 0x200000
(24114) address pointed to by p: 0x200000
(24113) p: 1
(24114) p: 1
(24114) p: 2
(24113) p: 2
(24113) p: 3
(24114) p: 3
(24113) p: 4
(24114) p: 4
```

- It is as if each running program has its **own private memory**.
  - Each running program has allocated memory **at the same address**.
  - Each seems to be updating the value at 0x200000 independently.

Virtualizing Memory

- Each process accesses its own private **virtual address space**.

- The OS maps **address space** onto the **physical memory**.
- A memory reference within one running program **does not affect** the address space of other processes.
- Physical memory is a **shared resource**, managed by the OS.

Problems of Concurrency

- The OS is juggling **many things at once**, first running one process, then another, and so forth.

- Modern **multi-threaded programs** also exhibit the concurrency problem.

Concurrency (`thread.c`)

```
#include <stdio.h>
#include <stdlib.h>
#include "common.h"
#include "common_threads.h"

volatile int counter = 0;
int loops;

void *worker(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        counter++;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "usage: threads <value>\n");
        exit(1);
    }
```

Concurrency (`thread.c`)

```
    loops = atoi(argv[1]);
    pthread_t p1, p2;
    printf("Initial value : %d\n", counter);

    Pthread_create(&p1, NULL, worker, NULL);
    Pthread_create(&p2, NULL, worker, NULL);
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("Final value : %d\n", counter);
    return 0;
}
```

- The main program creates **two threads**.
  - **Thread**: a function running within the **same memory space**. Each thread start running in a routine called `worker()`.
  - `worker()`: increments a counter

Concurrency (Result)

- `loops` determines how many times each of the two workers will increment the shared counter in a loop.
  - `loops`: 1000.

```
prompt> gcc -o thread thread.c -Wall -pthread
prompt> ./thread 1000
Initial value : 0
Final value : 2000
```

  - `loops`: 100000.

```
prompt> ./thread 100000
Initial value : 0
Final value : 143012         // huh??
prompt> ./thread 100000
Initial value : 0
Final value : 137298         // what the??
```

Why is this happening?

- Increment a shared counter ➔ take three instructions.

1. Load the value of the counter from memory into register.
2. Increment it.
3. Store it back into memory.

- These three instructions do not execute **atomically**.
  ➔ Problem of **concurrency** can happen.

Persistence

- Devices such as DRAM store values in a **volatile**.
- **Hardware** and **software** are needed to store data **persistently**.

- Hardware: I/O device such as a **hard drive**, **solid-state drives** (**SSDs**)
- Software:
  - **File system** manages the disk.
  - File system is responsible for storing any **files** the user creates.

Persistence (`io.c`)

```c
#include <stdio.h>
#include <unistd.h>
#include <assert.h>
#include <fcntl.h>
#include <sys/types.h>

int main(int argc, char *argv[]) {
    int fd = open("/tmp/file", O_WRONLY|O_CREAT|O_TRUNC,
        S_IRWXU);
    assert(fd > -1);
    int rc = write(fd, "hello world\n", 13);
    assert(rc == 13);
    close(fd);
    return 0;
}
```

- `open()`, `write()`, and `close()` **system calls** are routed to the part of OS called the **file system**, which handles the requests.

Persistence

- What OS does in order to write to disk?
  - Figure out **where** on disk this new data will reside.
  - **Issue I/O** requests to the underlying storage device.

- File system handles system crashes during write.
  - **Journaling** or **copy-on-write**.
  - Carefully **ordering** writes to disk.

Design Goals

- Build up **abstraction**
  - Make the system convenient and easy to use.
- Provide high **performance**
  - **Minimize the overheads** of the OS.
  - OS must strive to provide virtualization without excessive overheads.
- **Protection** between applications
  - **Isolation**: Bad behavior of one does not harm other and the OS itself.

Design Goals

- High degree of **reliability**
  - The OS must also run non-stop.

- Other issues
  - **Energy-efficiency**
  - **Security**
  - **Mobility**

Some History

- IBM 701 — Electronic Data Processing Machine (1952)
- No OS. Operated manually.

Batch Processing

- IBM 709 —Data Processing System (1952)
- SHARE Operation System (SOS). Manages buffers and I/O devices.

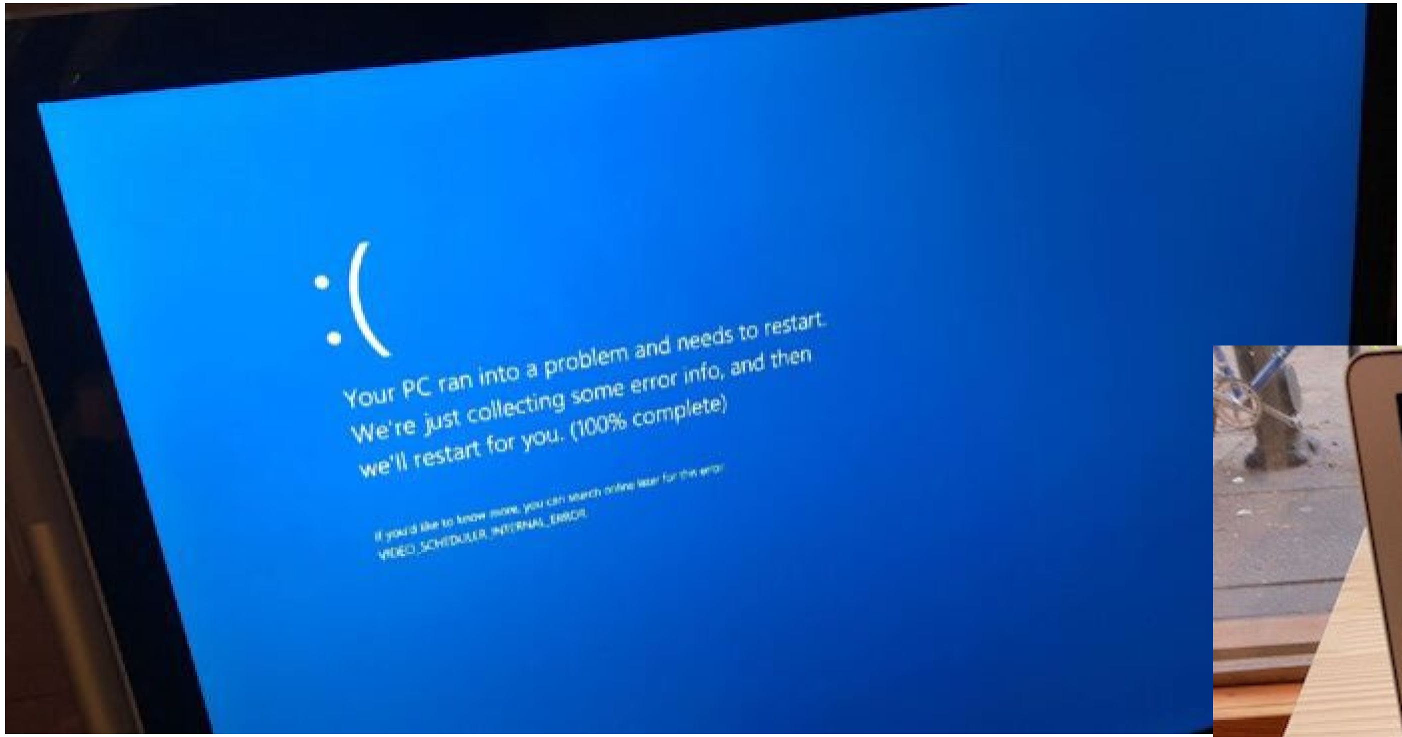
Multiprogramming

- PDP-11 — minicomputer (1970)
- Unix. Supports multiprogramming.

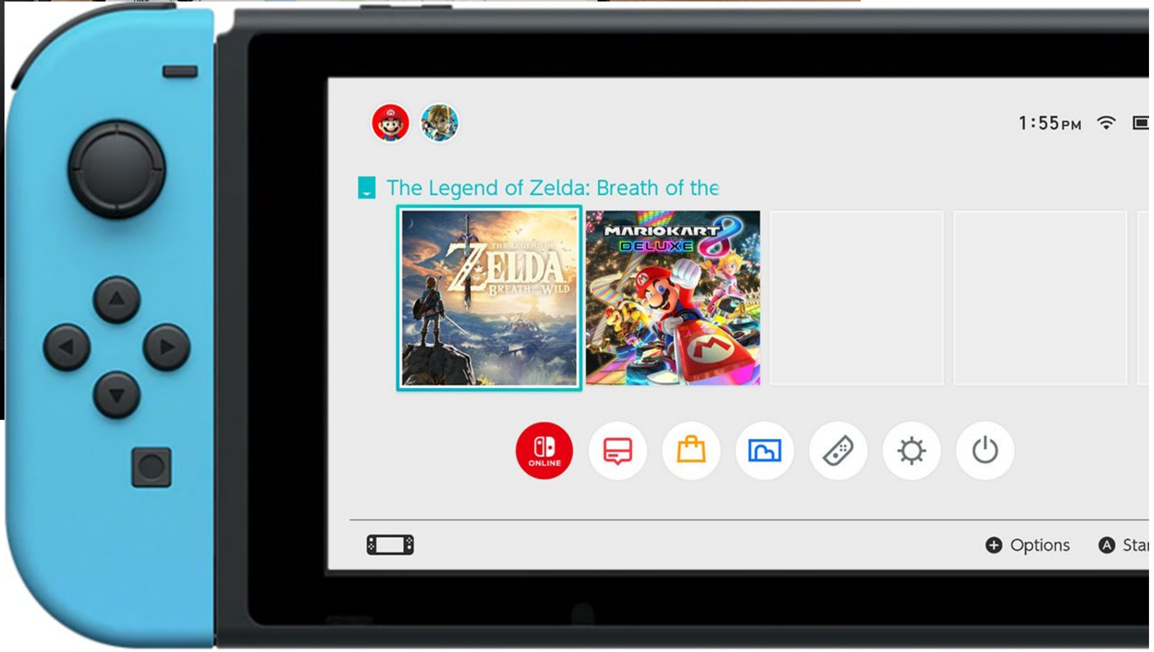
Personal Computer (PC)

- Apple II (1977) and IBM PC (1981)
- Disk Operating System (DOS) and Mac OS

Modern Era (and Error)



Mobile and IoT



Summary

- We have briefly introduced what an **operating system** (**OS**) is.
- Included in this course:
  - **Virtualization** of CPU and memory
    - Resource management. Allows multiple programs to run.
  - **Concurrency**
    - Data consistency. Data protection.
- Not included (but important topics nonetheless):
  - **Persistence** via devices and file systems.
  - **Networking**
  - **Graphics devices**
  - **Security**

References

- P. Brinch Hansen, “The Evolution of Operating Systems”, *Classic Operating Systems: From Batch Processing to Distributed Systems,* Springer-Verlag, New York, 2000.
- IBM Archives: IBM 701. https://www.ibm.com/ibm/history/exhibits/701/701_intro.html
- IBM Archives: 709 Data Processing System https://www.ibm.com/ibm/history/exhibits/mainframe/mainframe_PP709.html
- A, Hudson, “A brief tour of the PDP-11, the most influential minicomputer of all time”, Ars Technica, Mar 2022. https://arstechnica.com/gadgets/2022/03/a-brief-tour-of-the-pdp-11-the-most-influential-minicomputer-of-all-time/7/

Ch. 04. The Process

(OSTEP Ch. 4)

How to Provide the Illusion of Many CPUs?

- **CPU virtualization**
  - The OS can promote the illusion that many virtual CPUs exist.
  - **Time sharing**: Running one process, then stopping it and running another.

- The potential cost is **performance**.

Implementing Virtualization

- OS will need low-level machinery and high-level intelligence.

- **Mechanisms**: How to “context switch” between processes.
- **Policies**: Decides which process to run, e.g., a **scheduling policy**.

Process

A process is a **running program**.

- Comprising of a process:

- **Memory** (**address space**)
  - Instructions
  - Data section
- **Registers**
  - Program counter
  - Stack pointer

Process API

- These APIs are available on any modern OS.
- **Create**
  - Create a new process to run a program
- **Destroy**
  - Halt a runaway process
- **Wait**
  - Wait for a process to stop running
- **Miscellaneous Control**
  - Some kind of method to suspend a process and then resume it
- **Status**
  - Get some status info about a process

Process Creation

1. **Load** a program code into **memory**, into the address space of the process.
   - Programs initially reside on **disk** in **executable format**.
   - Modern OS perform the loading process **lazily**.
     - Loading pieces of code or data **only** as they are needed during program execution.

2. The program's run-time **stack** is allocated.
   - Use the stack for local variables, function parameters, and return address.
   - Initialize the stack with arguments ➔ `argc` and the `argv` array of `main()` function

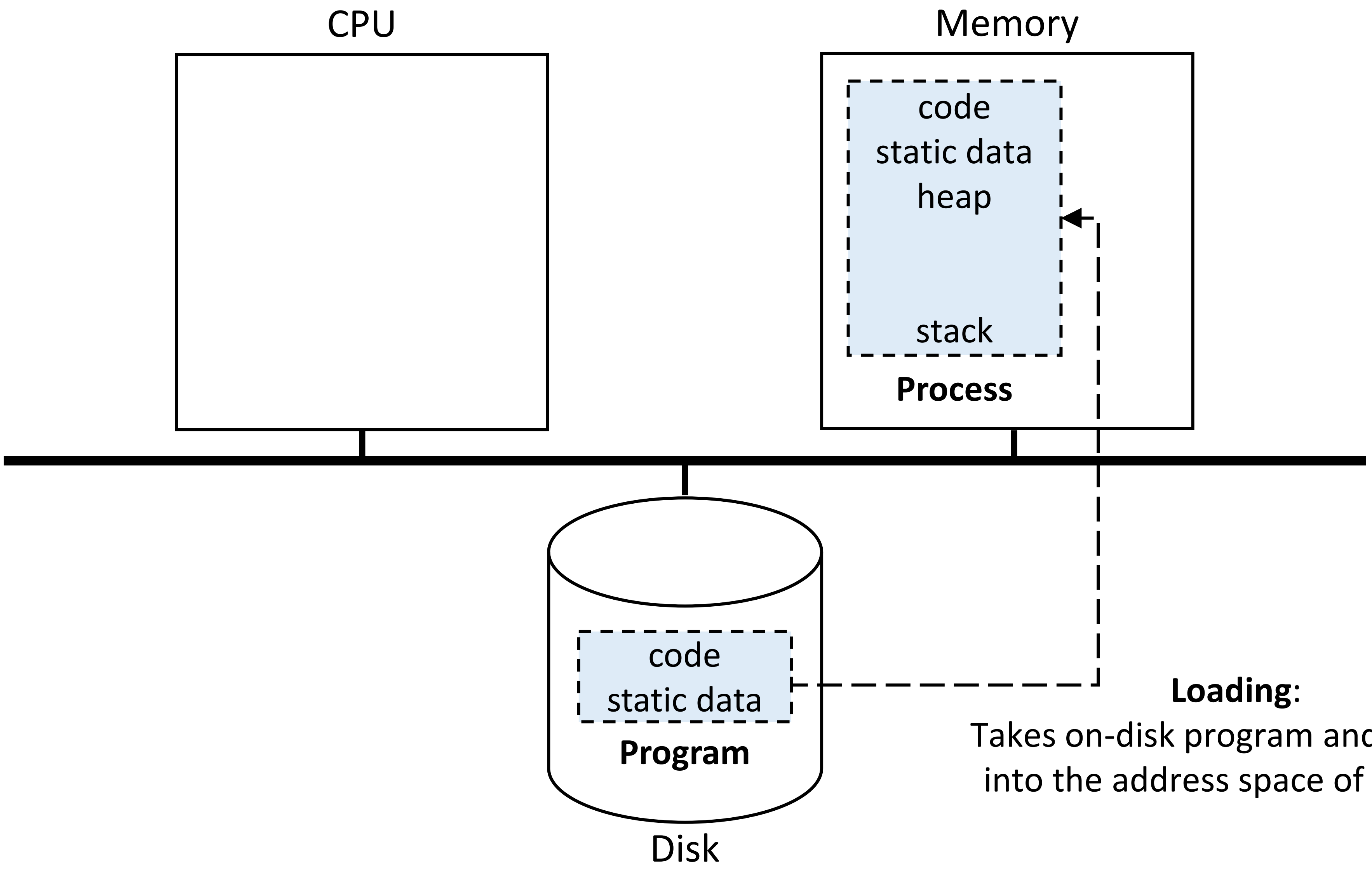
Process Creation

3. The program's **heap** is created.
   - Used for explicitly requested dynamically allocated data.
   - Program request such space by calling `malloc()` and free it by calling `free()`.

4. The OS do some other **initialization** tasks.
   - Input/output (I/O) setup
   - Each process by default has three open file descriptors.
     - Standard input, output and error

5. **Start the program** running at the entry point, namely `main()`.
   - The OS **transfers control** of the CPU to the newly-created process.

Loading: From Program To Process

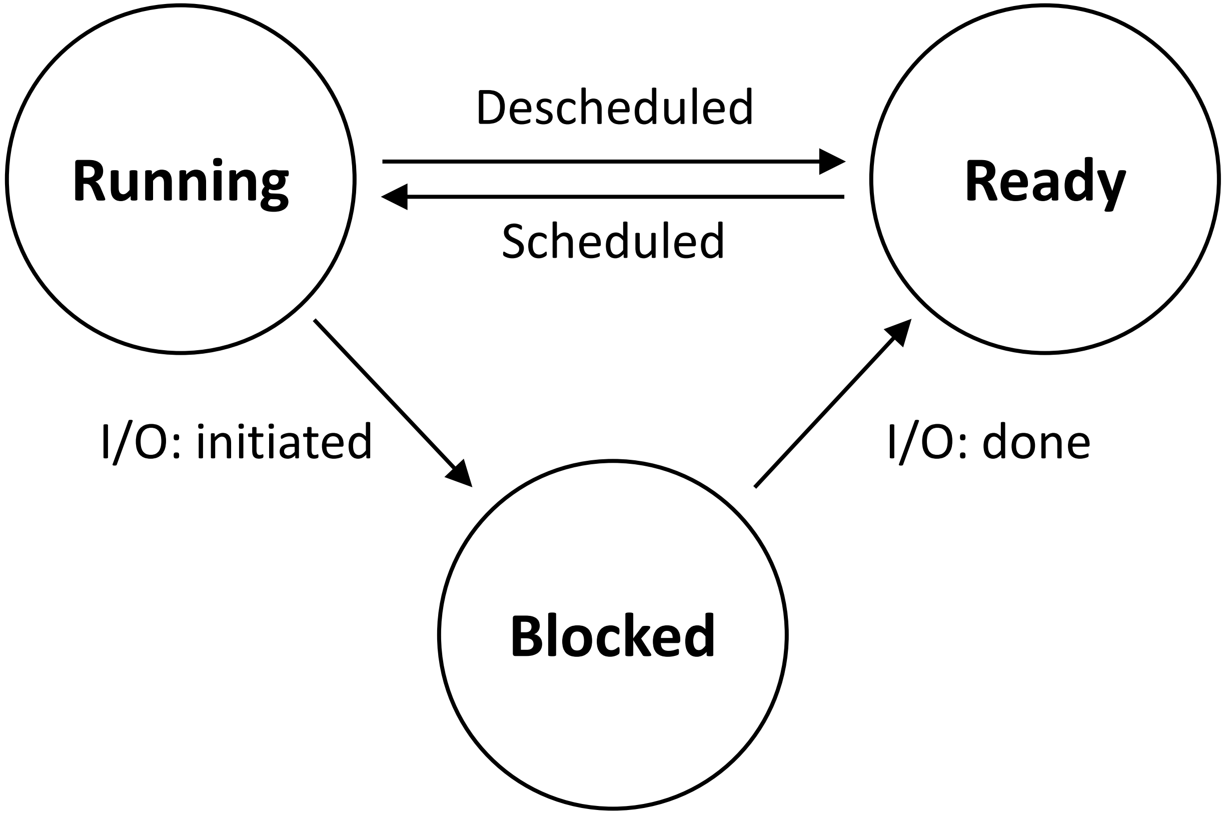


Loading:
Takes on-disk program and reads it into the address space of process

Process States

- A process can be one of three states.
- **Running**
  - A process is running on a processor.
- **Ready**
  - A process is ready to run but for some reason the OS has chosen not to run it at this given moment.
- **Blocked**
  - A process has performed some kind of operation.
  - When a process initiates an I/O request to a disk, it becomes blocked and thus some other process can use the processor.

Process: State Transitions



Tracing Process State: CPU Only

Time	Process_0	Process_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process_0 is now done
5	—	Running	
6	—	Running	
7	—	Running	
8	—	Running	Process_1 is now done

Tracing Process State: CPU and I/O

Time	Process_0	Process_1	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process_0 initiates I/O
4	Blocked	Running	Process_0 is blocked, so Process_1 runs
5	Blocked	Running	
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process_1 is now done
9	Running	—	
10	Running	—	Process_0 is now done

Data Structures

- OS maintains a data structure (e.g., **process list**) of all ready, running and blocked processes.

- **Process Control Block** (**PCB**)
  - A C-structure that contains information about each process.
  - **Register context**: a set of registers that define the state of a process

The xv6 Proc Structure

```
// the information xv6 tracks about each process
// including its register context and state
struct proc {
   char *mem;                     // Start of process memory
   uint sz;                       // Size of process memory
   char *kstack;                  // Bottom of kernel stack for this process
   enum proc_state state;         // Process state
   int pid;                       // Process ID
   struct proc *parent;           // Parent process
   void *chan;                    // If !zero, sleeping on chan
   int killed;                    // If !zero, has been killed
   struct file *ofile[NOFILE];    // Open files
   struct inode *cwd;             // Current directory
   struct context context;        // Switch here to run process
   struct trapframe *tf;          // Trap frame for the current interrupt
};
```

The xv6 Proc Structure

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
    int eip;
    int esp;
    int ebx;
    int ecx;
    int edx;
    int esi;
    int edi;
    int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

Summary

- We have introduced the most basic abstraction of the OS: the **process**.
- Low-level **mechanisms** are needed to switch between processes.
- High-level **policies** are required to schedule the processes in an intelligent way.