Overview

Learning Objectives

This section should help us answer the following questions:

- What is a **process**?
- What is **Process API** and what calls are related to it?
- What are the different **states** a process can be in?
- What is a **Process Control Block** and what does it have to do with the **Process List**

Introduction

One of the OS's most basic abstractions is the **process**. A **process** is just a running program. The program itself is only a set of instructions waiting to run. The operating system then runs the application, making it usable.

Think about your computer, where you would want to run a web browser, mail app, game, etc. A normal system might look like it's running tens or even hundreds of processes at once. Instead of worrying about whether a CPU is available, you just run applications.

The OS may generate the illusion of many virtual CPUs through **virtualizing** the CPU. In reality, there is only one real CPU by executing one process, stopping it, and starting another (or a few).

CPU **time-sharing**, allows users to run as many concurrent programs as they like but at the risk of performance loss.

▼ TIP: Time (and Space) Sharing

Time-sharing is an essential technique used by an OS to share a resource. Many entities can share the same resource when used by one for some time and then by another, etc.

Space sharing is time sharing's partner, where a resource is divided (in space) among those who wish to use it. For example, disk space is a naturally shared resource; once a block is assigned to a file, it usually isn't assigned to another until the original has been deleted.

The OS will need both low-level hardware and high-level intelligence to **virtualize** the CPU correctly.

- **Mechanisms** are low-level techniques or protocols that provide essential functionality.
- **Policies** are decision-making algorithms within the OS. For example, given a choice of CPU applications, which should the OS run?
    - The **scheduling policy** will likely choose using information on which applications ran recently, which sorts of programs run, and performance measurements.

Which of the following is the OS's abstraction for running a program?

What is a Process?

A **process** is just a running application. We describe processes by listing the many system components it works with during execution.

To understand a process, we must understand its machine state: what a software can read or update.

What parts of the machine are important for the program to run?

- **Memory** contains both instructions and data that the executing program reads and writes. So the memory space that the process can access is it's **address space**.
- **Registers** are also part of the process's machine state; various instructions specifically read or change them, indicating their importance.
  - Interesting registers include:
    - A **Program Counter** tells us which instruction will execute next.
    - A **stack pointer** and its **frame pointer** manage to stack for parameters, variables, and return addresses.

Finally, applications often use **persistent storage devices**. This I/O information may include a list of the process's open files.

Process Creation

Let's explore how a process is created.

- How does the OS start a program?
- How does process creation work?

Programs live on a **disk** in some executable format. The OS has to do a few things before the program is ready to run.

1. Read the programs bytes in the disk
2. Load its code and any data by putting the bytes into memory, the process's address space.
3. Allocate memory for the **run-time stack** of the program (or just stack) and given to the program.
4. Prepare for potential I/O requests.

▼ **Stack**

C programs use the **stack** for local variables, function arguments, and return addresses.

The OS will also likely initialize the stack with arguments, especially the argc and argv arrays for the `main()` method.

The OS may also allocate memory for the program's **heap**. In C programs, the heap stores dynamically allocated data. We need them for data structures like linked lists, hash tables, and trees. The heat may start small, but a program may request more memory by using the `malloc()` API.

After setting the stage, the OS's final job is to initiate the program at the entry point, `main()`. The OS passes control of the CPU to the newly formed process by jumping to the main() function, begins execution, and is ready for the show!

Process States

Knowing what a process is and how it is created, let us now discuss the various states a process can be in:

- **Running**: A process is running on CPU and is executing instructions.
- **Ready**: A process is ready to run, but the OS has decided not to do so at this time.
- **Blocked**: A process performs an action that stops it from running until another event happens.
    - Like when a process requests I/O from a disk, it becomes blocked, letting other processes use the CPU.

The OS can shift a process from ready to executing at will. A process is **scheduled** when it goes from ready to running, and **descheduled** when it goes from running to ready.

Initiating an I/O operation blocks a process until some event (like I/O completion) happens, at which time the process returns to the ready state (and potentially immediately to running again, if the OS so chooses).

Let us examine an example of two processes transitioning through these stages. Imagine two processes running, each using only the CPU (no I/O). A trail of each process's status may look like the table below.

Time	Process$_0$	Process$_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	Process$_0$ finishes
5	-	Running	
6	-	Running	
7	-	Running	
8	-	Running	Process$_1$ finishes

In this next example, after some time, the first process issues an I/O. The process is then blocked, allowing the other to run.

Process$_0$ initiates an I/O and becomes blocked while waiting for it to complete. The OS see Process$_0$ is idle and launches Process$_1$. The I/O completes, returning Process$_0$ to ready. Process$_1$ finishes, and Process 0 runs and finishes.

Time	**Process$_0$**	**Process$_1$**	**Notes**
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	Process$_0$ initiates I/O
4	Blocked	Running	Process$_0$ blocked, Process$_1$ runs
5	Blocked	Running	
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	Process$_1$ finishes
9	Running	-	
10	Running	-	Process$_0$ finishes

Even in this example, the OS makes a lot of decisions.

1. First, the system had to decide whether to execute Process$_1$ while Process$_0$ performed an I/O, effectively improving resource usage.

2. Second, the system chose not to return to Process$_0$ after its I/O finished.

Do you think this was the best choice?

Exploring Process States

This program, process-run.py, shows how a process's state changes while it runs on a CPU. A process can be in a few different states:

- **RUNNING** - the process is using the CPU right now
- **READY** - the process could be using the CPU right now, but (alas) some other process is
- **WAITING** - the process is waiting on I/O
  (e.g., it issued a request to a disk)
- **DONE** - the process is finished executing

This simulation will show us how these process states change while a program runs and help us understand how they work.

Run the following command in the terminal to run the program and see its options.

```
python process-run.py -h
```

It's very important to understand the PROCESS LIST (specified by the -l or -processlist flags) as they tell you what each running program (or 'process') does. Processes are made of instructions, and each instruction can only do one of two things:

- use the CPU
- issue an IO (and wait for it to complete)

When a process uses the CPU (but does no IO), it should simply switch between being **RUNNING** on the CPU and being **READY** to run. Let's do a simple run with only one application that uses the CPU (it does no I/O). run the program with the following commands:

```
./process-run.py -l 5:100
```

The process we defined is "5:100," which means it should have 5 instructions and a 100% probability of each instruction being a CPU instruction.

If you use the -c flag, you will be able to see what happens to the process:

```
./process-run.py -l 5:100 -c
```

The process is simply in the RUN stage and finishes, consuming the CPU the whole time and without doing any I/Os.

Let’s complicate things by running two processes:

```
./process-run.py -l 5:100,5:100
```

In this scenario, two processes run, each using just the CPU. So what happens when the OS runs them? Let’s see:

```
./process-run.py -l 5:100,5:100 -c
```

As seen above, process 0 runs first, followed by process 1, which is **READY** to run but waits for 0 to finish. 0 finishes and advances to **DONE** while 1 runs. The trace ends at 1.

##

Fill in the blanks below to complete the statements

One More Example

Let's look at one more example. The process in this example just makes I/O requests. Here we specify that I/Os take 5 time units to complete with the flag -L.

```
./process-run.py -l 3:0 -L 5
```

How do you expect the execution traces to look? Let's see:

```
./process-run.py -l 3:0 -L 5 -c
```

See how the program only issues three I/Os. Each I/O causes the process to enter a WAITING state, leaving the CPU idle.

One additional CPU operation completes the I/O. To handle I/O start and completion with a single instruction is not practical, but is utilized here for simplicity.

Let's print some metrics (with the -p flag) to examine some overall behaviors:

```
./process-run.py -l 3:0 -L 5 -c -p
```

As you can see, the trace took 21 clock ticks to run, yet the CPU was only >30% occupied. However, the I/O device was extremely active. We'd want to keep all devices active to maximize resource use.

Additional Flags

There are a few other important flags to be aware of:

```
  -s SEED, --seed=SEED  the random seed
    this gives you way to create a bunch of different jobs
randomly

  -L IO_LENGTH, --iolength=IO_LENGTH
    this determines how long IOs take to complete (default is 5
ticks)

  -S PROCESS_SWITCH_BEHAVIOR, --switch=PROCESS_SWITCH_BEHAVIOR
                        when to switch between processes:
SWITCH_ON_IO, SWITCH_ON_END
    this determines when we switch to another process:
    - SWITCH_ON_IO, the system will switch when a process issues
an IO
    - SWITCH_ON_END, the system will only switch when the
current process is done

  -I IO_DONE_BEHAVIOR, --iodone=IO_DONE_BEHAVIOR
                        type of behavior when IO ends:
IO_RUN_LATER, IO_RUN_IMMEDIATE
    this determines when a process runs after it issues an IO:
    - IO_RUN_IMMEDIATE: switch to this process right now
    - IO_RUN_LATER: switch to this process when it is natural to
      (e.g., depending on process-switching behavior)
```

We'll explore this more in the lab for the section.

Data Structures

The OS is a program. Just like any other program, it has **data structures** that track important data. The OS will likely keep a **process list** to keep track of the states of all of the processes.

The **process list** keep track of:
* Which processes are ready
* Which ones are running
* Which ones become blocked, and
* When I/O events complete.

The code snippet to the left shows what kind of information an OS must track about each xv6 kernel process. Similar process structures in operating systems like Linux, Mac OS X, or Windows.

In the code, the OS tracks some important information about a process.

The **register context** holds the contents of a stopped process's registers. When a process is stopped, its registers are stored here. The OS can continue the process by restoring them (i.e., putting their values back into the physical registers).

The code also shows that a process can be in states other than running, ready, and blocked.

A system may have an **initial state** when it is built. Also, a process might be terminated but not cleaned up. This end state allows other processes to check the return code and make sure the process completed properly (usually, programs return zero in UNIX-based systems when they have accomplished a task successfully, and non-zero otherwise).

After that, the parent will make a `wait()` to wait for the child's completion and tell the OS to clear away any data structures related to the now-extinct process.

info

Data Structure: The Process List

These data structures are common in operating systems. The process list (also known as the task list) is the first. It’s one of the simplest ones, but every OS that can run many applications at once would have something like this structure to keep track of them all.

A **Process Control Block (PCB)** is a fancy way of saying a C structure that holds information about each process (also sometimes called a process descriptor).

##

Which of the following is a data structure that holds information about a particular process?

Summary

- A **process** is an OS abstraction for a running program. It can be described by:

1. It’s **state**.
    - **Running**: When a process is running on CPU and is executing instructions.
    - **Ready**: When a process is ready to run, but the OS has decided not to do so at this time.
    - **Blocked**: When it performs an operation that prevents it from running until another event happens. Like when a process requests I/O from a disk, it becomes blocked, letting other processes use the CPU.
2. The contents of it’s **address space**.
3. The contents of its CPU registers
    - Like the program counter and stack pointer
4. It’s I/O information.
    - Like which files are open and can be read or written

- The **process API** has calls that programs can make relating to a process.
    - **Create:** Creates a new process.
    - **Destroy:** Destroys a process.
    - **Wait:** Waits for another process to finish running.
    - **Miscellaneous Control:** Other controls like, pause and resume, are often available
    - **Status:** May include how long a process has been running and its current state.
- A **process list** shows all processes in the system. Each item is in a **process control block (PCB)**, a structure containing information about a particular process.

Lab Intro

Let's explore a program, `process-run.py` that lets us see how the state of a process changes as it runs on a CPU. A process can be in a few different states:

```
RUNNING - the process is using the CPU right now
READY   - the process could be using the CPU right now
          but (alas) some other process is
WAITING - the process is waiting on I/O
          (e.g., it issued a request to a disk)
DONE    - the process is finished executing
```

First, let's look at the different options we have when working with `process-run.py`. Copy and paste the following command in the terminal and run it.

```
./process-run.py -h
```

You may need to type `python` before the command.

```
python process-run.py -h
```

You should see the following available options:

```
Usage: process-run.py [options]

Options:
  -h, --help            show this help message and exit
  -s SEED, --seed=SEED  the random seed
  -l PROCESS_LIST, --processlist=PROCESS_LIST
                        a comma-separated list of processes to
run, in the
                        form X1:Y1,X2:Y2,... where X is the
number of
                        instructions that process should run,
and Y the
                        chances (from 0 to 100) that an
instruction will use
                        the CPU or issue an IO
  -L IO_LENGTH, --iolength=IO_LENGTH
                        how long an IO takes
  -S PROCESS_SWITCH_BEHAVIOR, --switch=PROCESS_SWITCH_BEHAVIOR
                        when to switch between processes:
SWITCH_ON_IO,
                        SWITCH_ON_END
  -I IO_DONE_BEHAVIOR, --iodone=IO_DONE_BEHAVIOR
                        type of behavior when IO ends:
IO_RUN_LATER,
                        IO_RUN_IMMEDIATE
  -c                    compute answers for me
  -p, --printstats      print statistics at end; only useful
with -c flag
                        (otherwise stats are not printed)
```

The PROCESS LIST (as specified by the `-l` or — process list options) is the most important option to understand because it dictates what each running program (or ‘process’) will do. A process is made up of instructions, each of which can only do one of two things:

- Use the CPU
- Issue an I/O request (and wait for it to complete)

When a process uses the CPU (and does no IO), it should simply alternate between being READY to run and RUNNING on the CPU. Here’s an example of a simple run in which only one program is running, and that program only uses the CPU (it does no IO).

```
./process-run.py -l 5:100
```

The process we specified is `5:100`, which means it should have 5 instructions and a 100% likelihood of each instruction being a CPU instruction.

Using the `-c` flag, which computes the answers for you, you can see what happens to the process:

```
./process-run.py -l 5:100 -c
```

This result isn't super interesting. The process stays in the `RUN` state and finishes. It uses the CPU the entire time, keeping the CPU busy and not issuing any I/O requests.

Let's make it more interesting by running two processes:

```
./process-run.py -l 5:100,5:100
```

In this situation, two separate processes are running, each of which is only using the CPU. What happens when they're run by the operating system? Let's have a look:

```
./process-run.py -l 5:100,5:100 -c
```

The process with "process ID" (or "PID") 0 runs first, while process 1 is `READY` to run but waits for 0 to finish. When 0 is finished, it enters the `DONE` state, and 1 continues to run. The trace is complete once 1 is completed.

Let's look at one more example. In this case, the process is only making I/O requests. With the flag -L, we specify that I/Os take 5 time units to complete.

```
./process-run.py -l 3:0 -L 5
```

Can you figure out what the execution trace will look like? Let's use `-c` to find out:

```
./process-run.py -l 3:0 -L 5 -c
```

This program issues three I/Os. When each one is issued, the process moves to a `WAITING` state. While the device is busy servicing the I/O, the CPU is **idle**

Another CPU action is performed to handle the completion of the I/O. It's worth noting that using a single instruction to handle I/O initiation and completion isn't really realistic. It's only used here for the sake of simplicity.

To see some overall behaviors, let's print some stats (use the same program as above, but with the `-p` flag):

```
Stats: Total Time 21
Stats: CPU Busy 6 (28.57%)
Stats: IO Busy  15 (71.43%)
```

The trace took 21 clock ticks to complete, however the CPU was only busy for about 30 of the time. On the other hand, the I/O device was super active. We'd prefer to keep all of the devices active in general because it's a better use of resources.

There are a few more important flags to be aware of:

```
  -s SEED, --seed=SEED  the random seed
    this gives you way to create a bunch of different jobs
randomly

  -L IO_LENGTH, --iolength=IO_LENGTH
    this determines how long IOs take to complete (default is 5
ticks)

  -S PROCESS_SWITCH_BEHAVIOR, --switch=PROCESS_SWITCH_BEHAVIOR
                        when to switch between processes:
SWITCH_ON_IO, SWITCH_ON_END
    this determines when we switch to another process:
    - SWITCH_ON_IO, the system will switch when a process issues
an IO
    - SWITCH_ON_END, the system will only switch when the
current process is done

  -I IO_DONE_BEHAVIOR, --iodone=IO_DONE_BEHAVIOR
                        type of behavior when IO ends:
IO_RUN_LATER, IO_RUN_IMMEDIATE
    this determines when a process runs after it issues an IO:
    - IO_RUN_IMMEDIATE: switch to this process right now
    - IO_RUN_LATER: switch to this process when it is natural to
      (e.g., depending on process-switching behavior)
```

Now let's get ready to explore more in our lab!

Lab 1

For the following questions, use the `-c` and `-p` flags to check your work before submitting.

Lab 2

Now we'll look at some of the other flags. The `-S` option is important because it controls how the system behaves when a process uses an I/O. When the `SWITCH_ON_END` flag is set, the system will not switch to another process while one is performing I/O, but will instead wait until the process is finished.

Let's run the programs again and change the switching behavior to switch to another process while one is WAITING for I/O.

It's also important to know what to do when an I/O completes. With `-I IO_RUN_LATER`, the process that issued the request is not guaranteed to automatically keep running when the I/O completes.

Execute the same processes again, but this time with `-I IO_RUN_ IMMEDIATE` set, which will run the process that issued the I/O immediately.