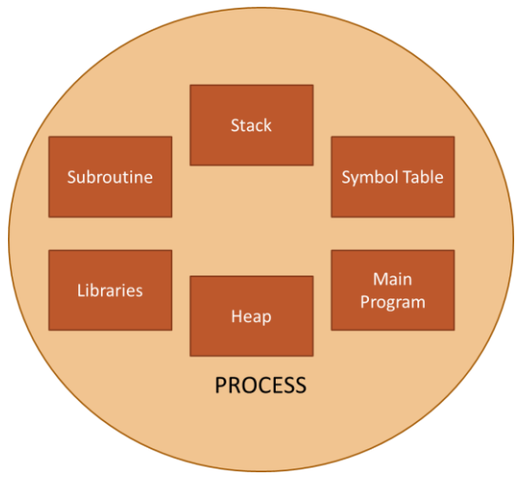
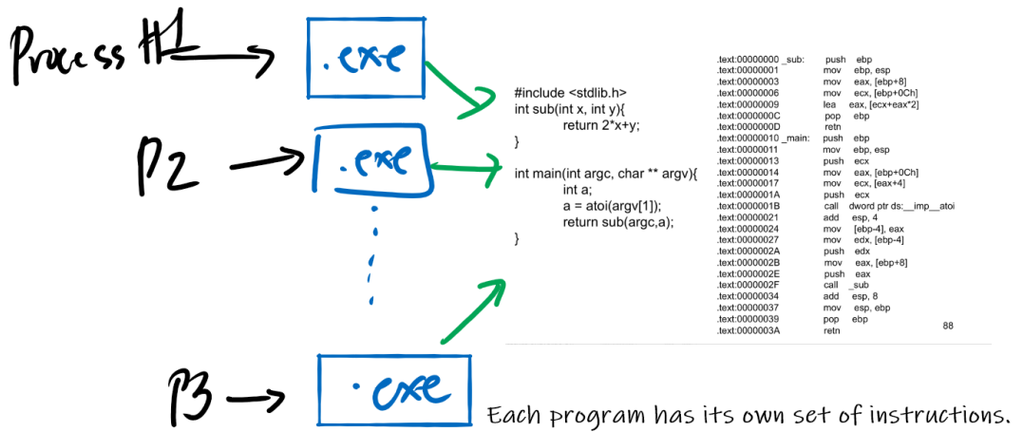
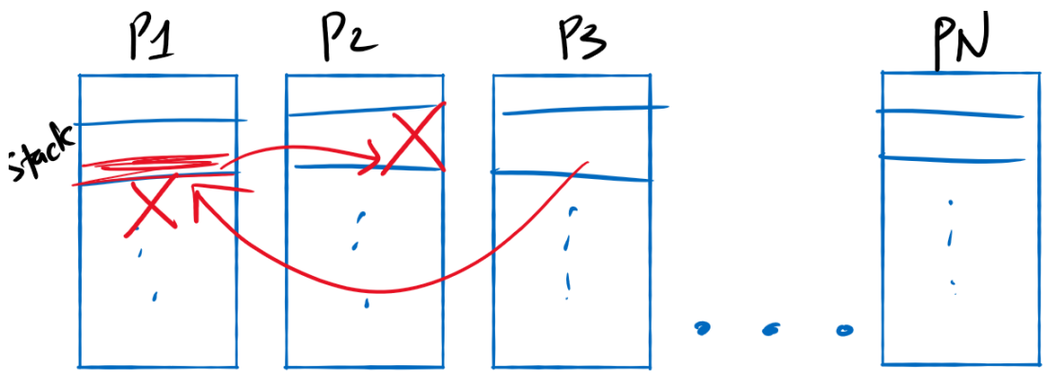
WHAT IS A PROCESS?

- One program = one process.
- A standalone application that contains memory, stack, heap, etc.
- In C++, we use fork() method to create a duplicate process of the program.
- Context switch between the process is time consuming.
- We don't typically practice developing programs using multiple processes.







Each process has its own allocation and cannot be accessed by any other process.

Brief Example of Multi-Processing in C++

Fork system call is used for creating a new process, which is called child process, which runs concurrently with the process that makes the fork() call (parent process). After a new child process is created, both processes will execute the next instruction following the fork() system call.

NOTE: Calling fork() is only applicable for UNIX systems! This is also why multiprocessing for games are rarely done, as game programs/engines are typically created in Windows systems.

Creates a new process (P2), then the new process proceeds with executing the next line.

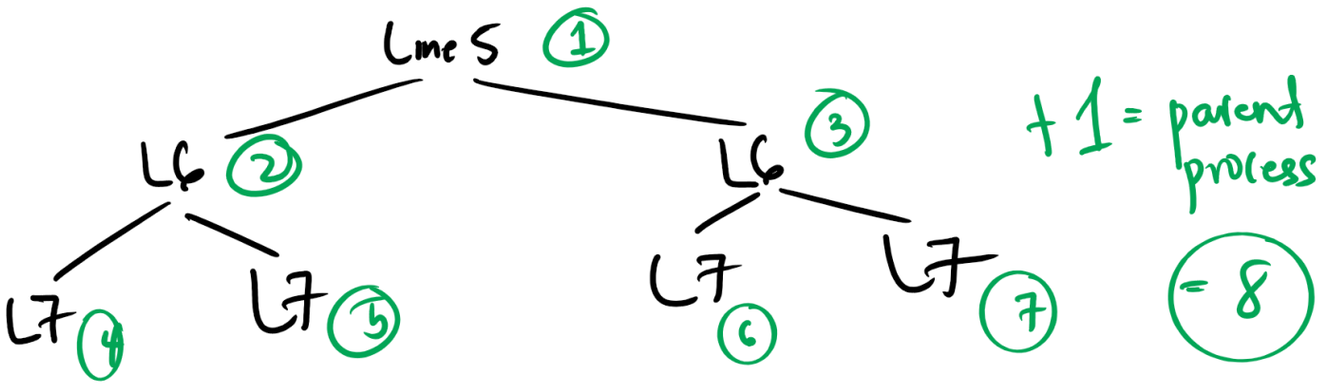
```c
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

Output:

```
hello
hello
hello
hello
hello
hello
hello
hello
```

P1 and P2 will then execute line 6 onwards.

This code will create processes exponentially!



Another Example

```c
// C++ program to demonstrate creating processes using fork()
#include <unistd.h>
#include <stdio.h>

int main()
{
    // Creating first child
    int n1 = fork();

    // Creating second child. First child
    // also executes this line and creates
    // grandchild.
    int n2 = fork();

    if (n1 > 0 && n2 > 0) {
        printf("parent\n");
        printf("%d %d \n", n1, n2);
        printf(" my id is %d \n", getpid());
    }
    else if (n1 == 0 && n2 > 0)
    {
        printf("First child\n");
        printf("%d %d \n", n1, n2);
        printf("my id is %d \n", getpid());
    }
    else if (n1 > 0 && n2 == 0)
    {
        printf("Second child\n");
        printf("%d %d \n", n1, n2);
        printf("my id is %d \n", getpid());
    }
    else {
        printf("third child\n");
        printf("%d %d \n", n1, n2);
        printf(" my id is %d \n", getpid());
    }
    return 0;
}
```

```
parent
28808 28809
 my id is 28807
First child
0 28810
my id is 28808
Second child
28808 0
my id is 28809
third child
0 0
```

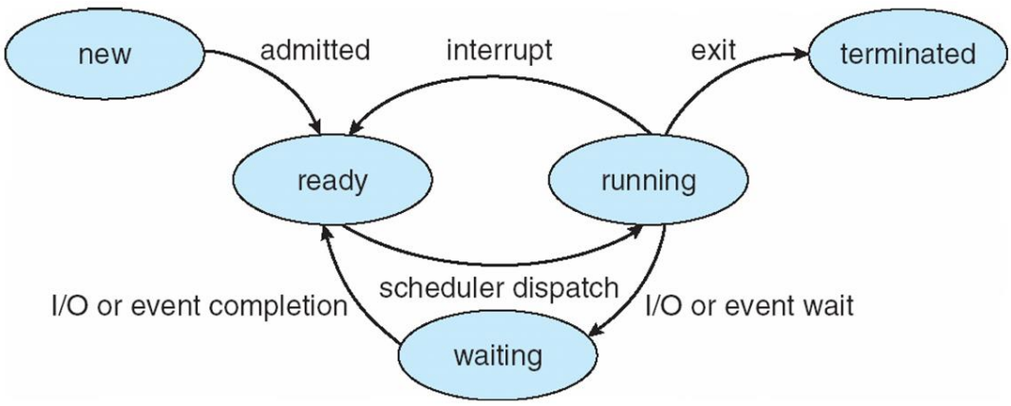
Process transition

- A program is in **passive** state when it is stored on disk (as an executable file).
- A program becomes a process when the executable file is loaded into memory (**active** state).
- Execution of program started via GUI mouse clicks, command line entry of its name, etc.
- One program can have several processes. E.g. consider a user opening a program multiple times.

Process states

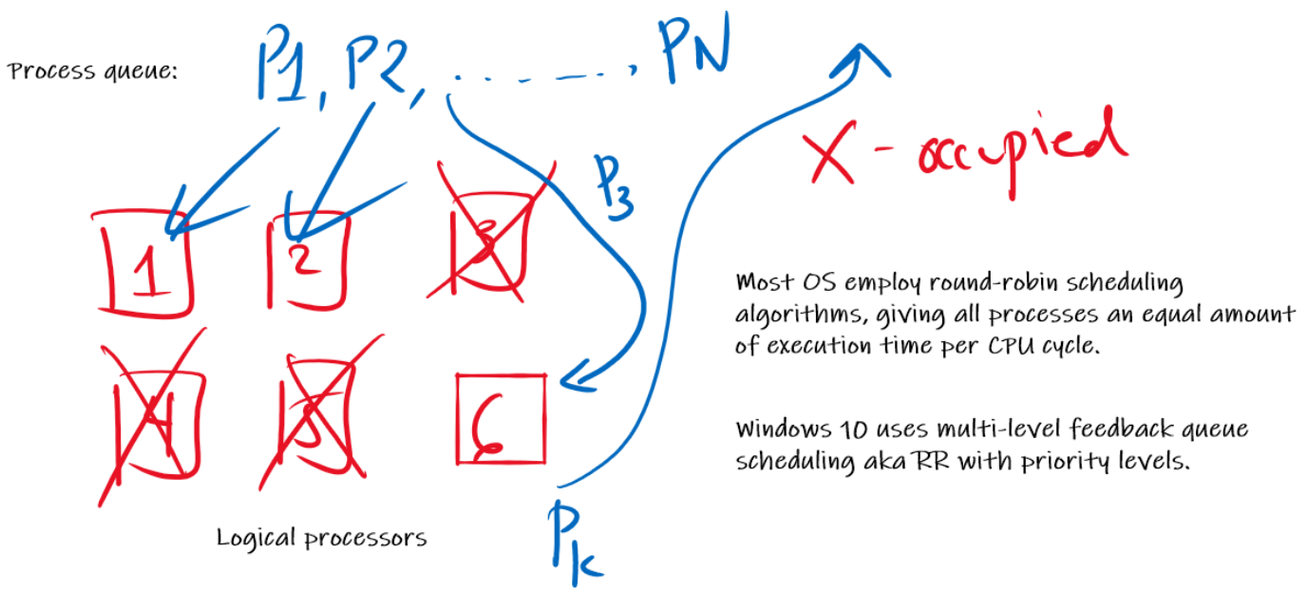
As a process executes, it changes **state**

- **new**: The process is being created
- **running**: Instructions are being executed
- **waiting**: The process is waiting for some event to occur
- **ready**: The process is waiting to be assigned to a processor
- **terminated**: The process has finished execution



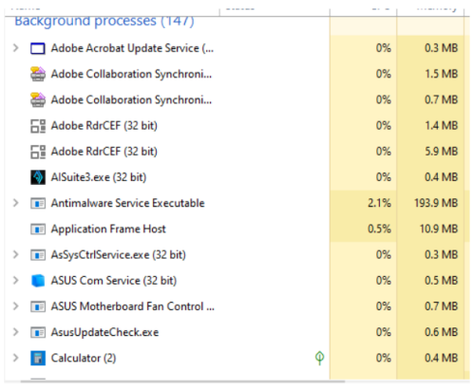
Very similar to application lifecycle for mobile.

Recall process/CPU scheduling



Processes can move to different states frequently. Consider this example:

OBSERVATION: Any active window implies that the process does not change state frequently.

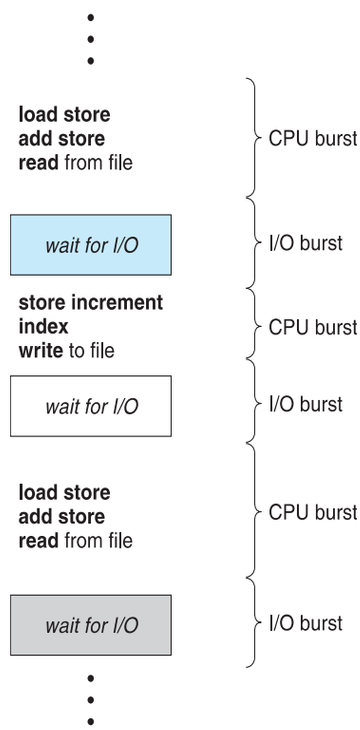


However, background processes, such as services, move to various states frequently.

REASON: Foreground processes has more priority, a background process may stop execution at any time when additional resources are needed.

CPU Scheduling – Basic Concepts

- One goal of the OS is to have maximum CPU utilization. Keep CPU busy at all time and finish as many processes as possible.
- A CPU can be on idle state such as the following case on the right. Program has "scattered" sections of I/O-related code. Whenever there is an I/O code, the process must be removed from the CPU and move to the device queue (and gets executed in the I/O system program).
- A commercial OS will never have long CPU idle times even if the user is not seeing anything on-screen.



Terms

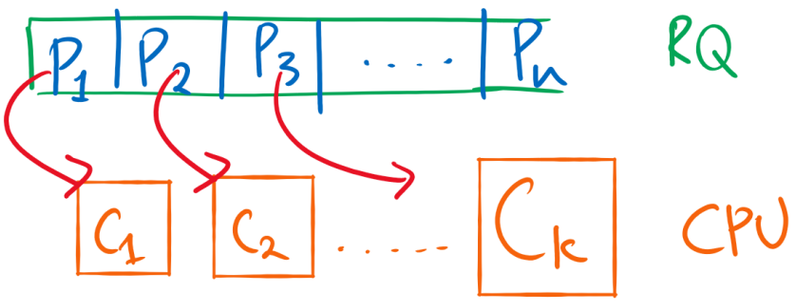
- CPU – not the physical CPU hardware. A logical unit for executing a process.
- Ready queue – holds processes that should be executed in the CPU.
- Short-term scheduler - selects from among the processes in ready queue and allocates to the CPU.
- Long-term scheduler – decides which processes should go first to the short-term scheduler. NOTE: Not all OS have this.
- CPU scheduling decisions may take place when a process:
  1. Switches from running to waiting state
  2. Switches from running to ready state
  3. Switches from waiting to ready
  4. Terminates

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment).

Demonstration of CPU Scheduling: Shortest-Job-First (Pre-Emptive)
Shortest-job-first, by the name itself, selects processes with the shortest CPU burst for scheduling. Consider the following example:

The ready queue holds pending processes



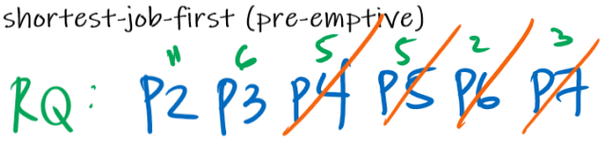
Each logical core can hold one process for execution.

When the process arrives at the ready queue

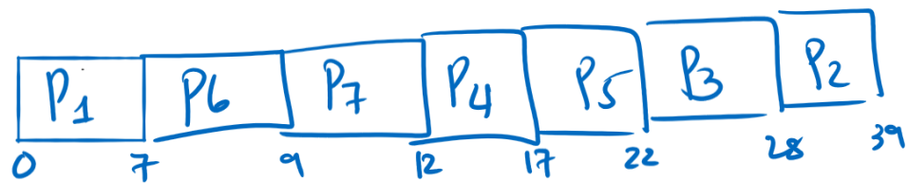
Total CPU time needed to finish it

Process	Arrival Time	CPU Burst
P1	0	7
P2	1	11
P3	2	6
P4	3	5
P5	4	5
P6	6	2
P7	7	3

Using shortest-job-first (pre-emptive)

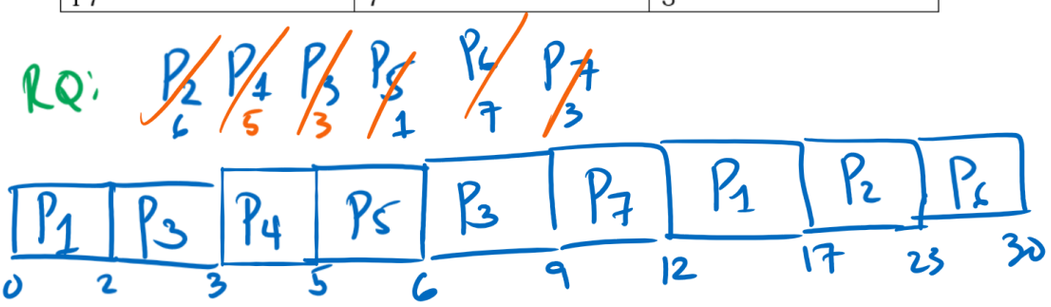


Gantt chart of CPU execution



To show how pre-emptive works

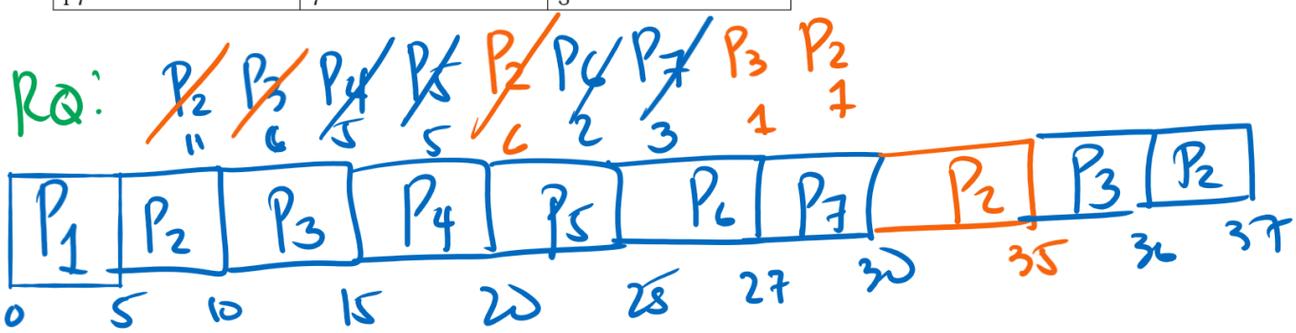
Process	Arrival Time	CPU Burst
P1	0	7
P2	1	~~11~~ 6
P3	2	~~6~~ 4
P4	3	~~5~~ 2
P5	4	~~5~~ 1
P6	6	~~2~~ 7
P7	7	3



Demonstration of CPU Scheduling: Round-Robin
Promotes fairness policy by cycling through all processes that arrive evenly. Each process receives a fair amount of CPU time for execution.

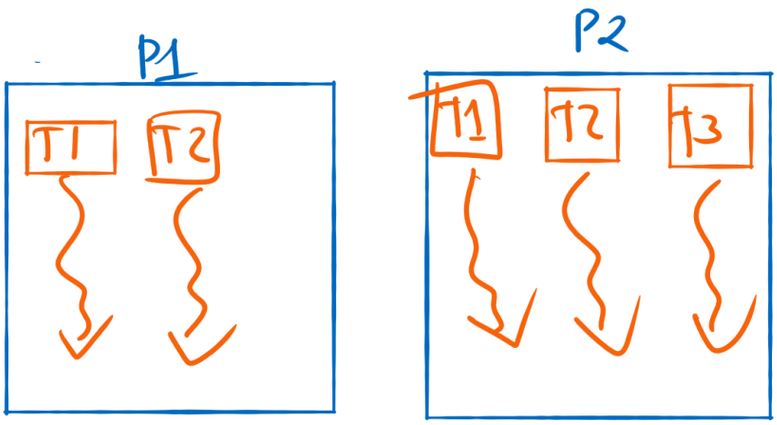
Using round-robin with CPU burst allocation of 5

Process	Arrival Time	CPU Burst
P1	0	7
P2	1	11
P3	2	6
P4	3	5
P5	4	5
P6	6	2
P7	7	3



IDEA: Promote fairness on all processes as much as possible

WHAT IS A THREAD?



- A thread is a lightweight process
- A sequence of instructions to execute in parallel with another thread.
- Since it is within a process, memory is shared.
- Multiple threads can access the same data structure, functions, etc.

Hands-On Activity: Create your first "Hello World" thread in C++.

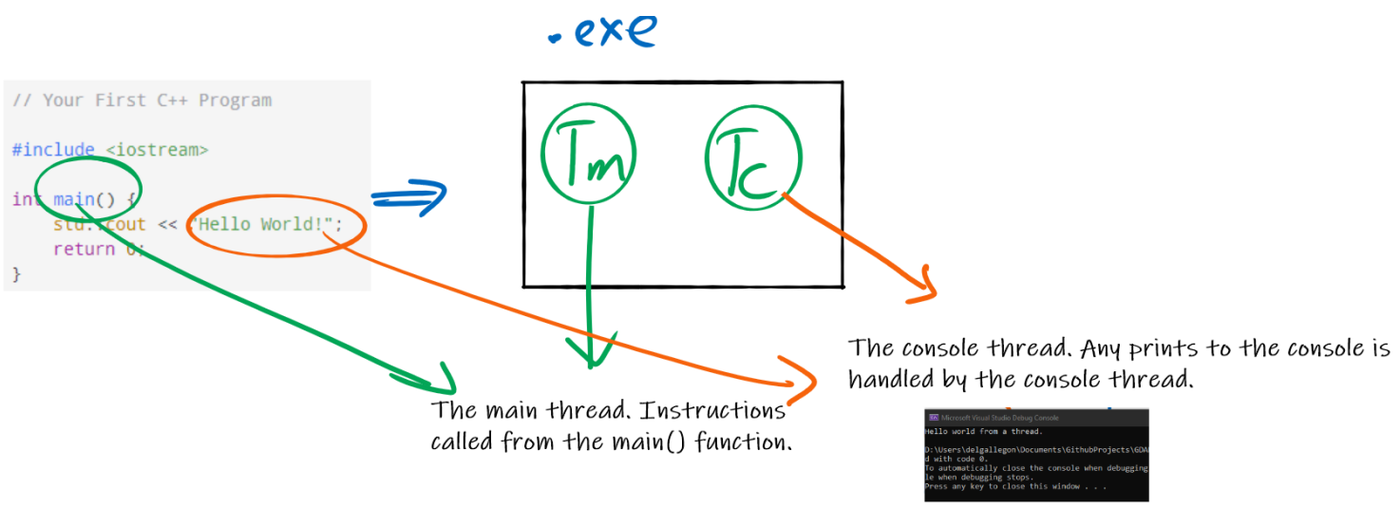
```
19  void testFunctionThread()
20  {
21      std::cout << "Hello world from a thread. " << std::endl;
22  }
23
24  int main() {
25      //createHWThreads();
26      std::thread myThread(testFunctionThread);
27  }
```

NOTE: Running the above program will trigger a runtime error. C++ has a guarding mechanism that should force the developer to call either join() or detach().

```
24  int main() {
25      //createHWThreads();
26      std::thread myThread(testFunctionThread);
27      myThread.join();
28  }
```

A Windows program – Default thread handling.

A default Windows program always has the following set of threads: **main thread, UI thread**. See illustration below.

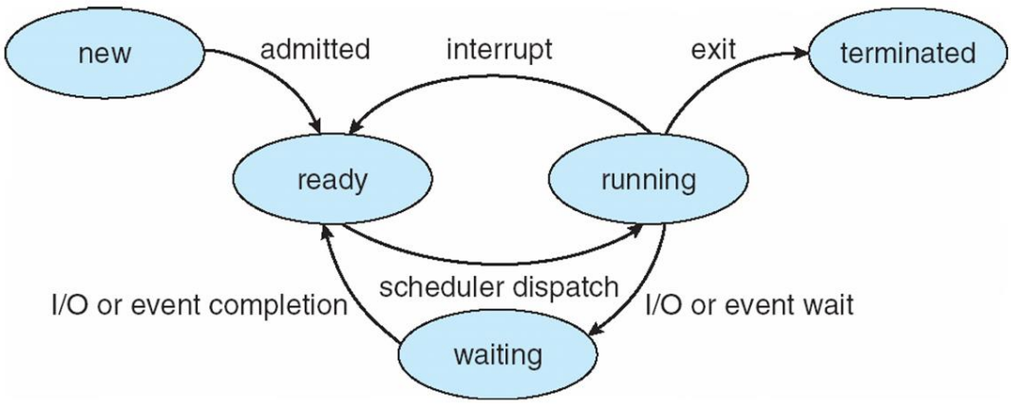


The console thread persistently runs until the main program is closed.

For programs that has a Window screen, such as game applications, there are actually 3 threads by default.

NOTE: All the spawned threads are independent from one another. Information is being passed to each thread through **message passing** (e.g. std::cout, render/draw frame in SFML/OPENGL).

Since threads are lightweight processes, they also undergo the same states as well.



ACTIVITY: For both Unity and Unreal engine, identify the different background threads that run when you use the engine. Explain the purpose of each thread.