This class

- Sections 3.1—3.3
  - Processes, creation, scheduling and termination
- Learning objectives
  - How a process is represented inside the kernel
  - Why a process is a stateful entity, and how it transitions among states
  - Example process-managing system calls
  - Process pools/queues
  - The lifecycle of a process
What is a process?
To the operating system, a process is:

- Two data structures:
  - The loaded program in its memory space
  - The “process” data structure inside the kernel (Process Control Block)
The process on the memory

- It contains four main parts
  - **Stack** (function/procedure call stack, parameters and local variables)
  - **Heap** (dynamically allocated memory)
  - **Data** (statically allocated variables, constants, and hard-coded data)
  - **Text** (compiled program image)
The Process Control Block

- In general, it contains all the information about a process that is not stored in the main memory, e.g.:

<table>
<thead>
<tr>
<th>process state</th>
<th>process number</th>
<th>program counter</th>
</tr>
</thead>
<tbody>
<tr>
<td>registers</td>
<td>memory limits</td>
<td>list of open files</td>
</tr>
</tbody>
</table>

...
PCB and the process' memory layout

http://static.duarte.org/img/blogPosts/mm_struct.png
The state of a process

- In Linux:
  - R: running
  - S: sleeping in an interruptible wait,
  - D: waiting in uninterruptible disk sleep,
  - Z: zombie,
  - T: traced or stopped (on a signal)
  - W: paging.
How the kernel switches and schedules processes
How the kernel switches processes

![Diagram showing process switching](image-url)
Job (a.k.a. process) queuing

- A process stays in the CPU until...

- after which, it is placed into the ready queue
The various queues in the kernel

- Queues are usually implemented as linked lists
- Process queues may more accurately be called “process pools”
The various kinds of schedulers

- **Long-term scheduler** (or job scheduler)
  - selects which processes should be brought into the ready queue
  - is invoked very infrequently (seconds, minutes) ⇒ (may be slow)
  - controls the *degree of multiprogramming*

- **Medium-term scheduler**
  - selects which processes should be swapped out of main memory to improve performance of the more active ones

- **Short-term scheduler** (or CPU scheduler)
  - selects which process should be executed next and allocates CPU
  - is invoked very frequently (milliseconds) ⇒ (must be fast)

- **I/O-bound process** – spends more time doing I/O than computations, many short CPU bursts

- **CPU-bound process** – spends more time doing computations; few very long CPU bursts
The process life cycle
Oh yes, it looks like this...
An existing process calls the fork() system call to replicate itself

The original goes on about its business, or waits for the child (i.e. replica) process, while the child calls the exec() system call to run a specific program

The child process exits and the parent (i.e. original) process is woken up from waiting.
System calls on process creation and manipulation

- **execve** - execute a program
- **clone** - create a child process
- **fork** - create a child process
- **vfork** - create a child process and block parent
- **getpid** - get process identification
- **getppid** - get parent process identification
- **exit** - terminate the calling process

- **wait** - wait for process to change state
- **wait4** - wait for process to change state, BSD style
- **kill** - send a signal to a process
- **sigaction** - examine and change a signal action
- **signal** - ANSI C signal handling
Programming example
(from wait(2) man page)

```c
#include <sys/wait.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    pid_t cpid, w;
    int status;
    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (cpid == 0) { /* Code executed by child */
        printf("Child PID is %ld\n", (long) getpid());
        if (argc == 1)
            pause(); /* Wait for signals */
        _exit(atoi(argv[1]));
    } else { /* Code executed by parent */
        do {
            w = waitpid(cpid, &status, WUNTRACED | WCONTINUED);
            if (w == -1) {
                perror("waitpid");
                exit(EXIT_FAILURE);
            }
            if (WIFEXITED(status)) {
                printf("exited, status=%d\n", WEXITSTATUS(status));
            } else if (WIFSIGNALED(status)) {
                printf("killed by signal %d\n", WTERMSIG(status));
            } else if (WIFSTOPPED(status)) {
                printf("stopped by signal %d\n", WSTOPSIG(status));
            } else if (WIFCONTINUED(status)) {
                printf("continued\n");
            }
        } while (!WIFEXITED(status) && !WIFSIGNALED(status));
    exit(EXIT_SUCCESS);
    }
```
Process termination in UNIX

- Process executes last statement and asks the operating system to delete it (exit)
  - Output data from child to parent (via wait)
  - Process’ resources are deallocated by operating system
- Parent may terminate execution of children processes (abort)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If parent is exiting
    - Some operating system do not allow child to continue if its parent terminates
    - All children terminated - cascading termination