Processes

Getting to know you

- A process is an abstraction that supports running programs.
- A sequential stream of execution in its own address space.
- A process is NOT the same as a program!
- So, two parts to a process:
  1. Sequential execution: no concurrency inside a process—everything happens sequentially.
  2. Address space: all process state—everything that interacts with the stream of execution.

Anatomy of a Process

Of Programs and Processes

- More to a process than just a program.
  - I run `ls`, you run `ls`—same program—but different processes.
- Less to a process than a program.
  - A program can invoke many processes to get the job done (e.g., `cc` starts up `cpp`, `cc1`, `cc2`, as—each a different process (and program!)).
Process Life Cycle

- Processes are always either executing, waiting to execute or waiting for an event to occur.

1. **New**
   - Process transitions from New to Ready when it becomes runnable.

2. **Ready**
   - Process transitions from Ready to Running when kernel schedules it.

3. **Start**
   - Process transitions from Ready to Running when kernel schedules it.
Process Life Cycle

Processes are always either executing, waiting to execute or waiting for an event to occur.

- **Start**
- **Ready**
- **Running**

Process transitions from Running to Waiting when they are blocked, waiting for an event to occur (e.g. I/O).

- **Start**
- **Ready**
- **Running**
- **Waiting**

Is that it?
Process Life Cycle

- Processes are always either executing, waiting to execute or waiting for an event to occur.

**Process transitions from Running to Ready on an interrupt:**
- Start
- Ready
- Running
- Waiting

**Process transitions from Running to Done on exit():**
- Start
- Ready
- Running
- Waiting
- Done
What happens on a read()?

Start
Ready
Running
Waiting
Done

Process Manipulation

Basic process manipulation: creation, program loading, exiting, ...

Example: Unix Operating system

- Creation and deletion: fork(), exec(), wait(), exit()
- Process signaling: kill()
- Process control: ptrace(), nice(), sleep()

Process Manipulation in Unix

- The system creates the first process (sysproc)
- The first process creates other processes in a tree like structure:
  - the creator is called the parent process
  - the created is called the child process
- Unix system interface includes a call to create processes
  - fork()

Unix’s fork()

- Creates a child process such that it inherits:
  - identical copy of all parent’s variables & memory
  - identical copy of all parent’s CPU registers (except one)
- Both parent and child execute at the same point after fork() returns:
  - for the child, fork() returns 0
  - for the parent, fork() returns the process identifier of the child
- Simple implementation of fork():
  - allocate memory for the child process
  - copy parent’s memory and CPU registers to child’s
  - Expensive !!

Can one reduce overhead without changing semantics?
Using Unix’s fork()

- The execution context for the child process is a copy of the parent’s context at the time of the call.

```c
main {
    int childPID;
    childPID = fork();
    if(childPID == 0) {
        // child continues here
        exec("program", argc, argv0, argv1, ...);
    } else { // parent continues here
        ...
    }
}
```

General Purpose Process Creation

In the parent process:
```c
main {
    int childPID;
    childPID = fork();
    if(childPID == 0) {
        // child continues here
        exec("program", argc, argv0, argv1, ...);
    } else { // parent continues here
        ...
    }
}
```

Program Loading: exec()

- The exec() call allows a process to “load” a different program and start execution at _start.
- It allows a process to specify the number of arguments (argc) and the string argument array (argv).
- If the call is successful:
  - it is the same process ...
  - but it runs a different program !!

Two implementation options:
- overwrite current memory segments with the new values
- allocate new memory segments, load them with the new values, and deallocate old segments

Orderly Termination: exit()

- After the program finishes execution, it calls exit()
- This system call:
  - takes the “result” of the program as an argument
  - closes all open files, connections, etc.
  - deallocates memory
  - deallocates most of the OS structures supporting the process
  - checks if parent is alive:
    - If so, it holds the result value until parent requests it; in this case, process does not really die, but it enters the zombie/defunct state
    - If not, it deallocates all data structures, the process is dead
  - cleans up all waiting zombies
The wait() System Call

A child program returns a value to the parent, so the parent must arrange to receive that value.

The wait(&value) system call serves this purpose:
1. it puts the parent to sleep waiting for a child's result
2. when a child calls exit(), the OS unblocks the parent and returns the address of value passed by exit() as a result of the wait call (along with the pid of the child)
3. if there are no children alive, wait() returns immediately
4. also, if there are zombies waiting for their parents, wait() returns one of the values immediately (and deallocates the zombie)

Tying all together: the Unix shell

while(! EOF)
read input
handle regular expressions
int pid = fork() // create child
if (pid == 0) { // child here
    exec("program", argc, argv0, ...);
}
else { // parent here
    ...
}

Translates <CTRL-C> to the kill() system call with SIGKILL
Translates <CTRL-Z> to the kill() system call with SIGSTOP
Allows input-output redirection, pipes, and other things we will see later.