Programs, Processes, Threads

Reading: Stevens - Chapters 7/8

(Focus on 7.1-7.6 , 8.1-8.6 , 8.10)

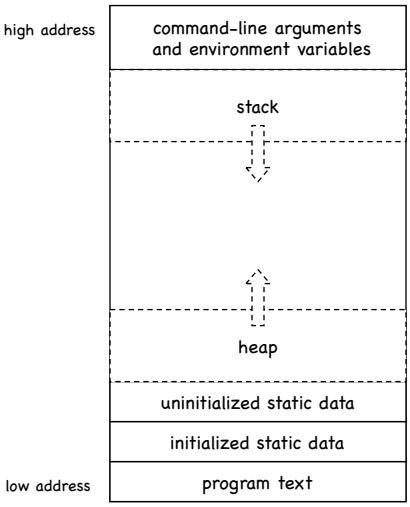
Processes Management

- What **is** a process?
- How to **control** processes.
- How to allocate the available resources to the execution of the processes (**scheduling**)
- How to coordinate processes among themselves (**synchronization**)

Processes and Process Control

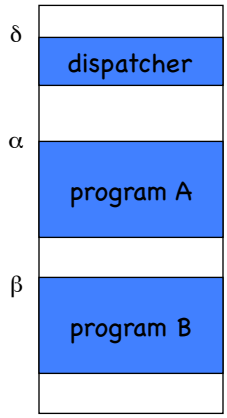
- Q: What is a process?
- *A:*
  - *Process* as execution of a *Program*
  - We can trace the execution of a process
  - Process as minimal entity for resource allocation (for example memory).

Simple Memory Layout of a Running Program



The Execution Trace of Processes

- Two processes and a dispatcher



Traces of processes **A** and **B**

α	β
α+1	β+1
α+2	β+2
α+3	β+3
α+4	β+4
α+5	β+5
α+6	β+6
α+7	β+7
α+8	β+8
α+9	β+9
α+10	β+10
α+11	β+11

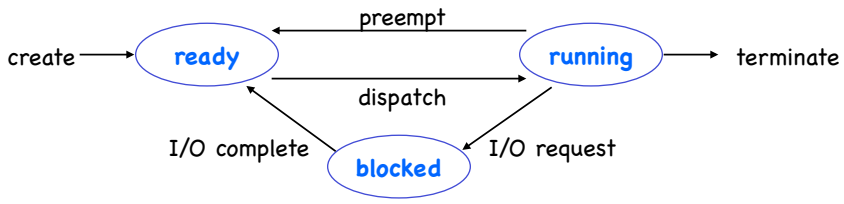
Trace of **dispatcher**

δ
δ+1
δ+2
δ+3
δ+4

β
β+1
β+2
β+3
β+4
δ
δ+1
δ+2
δ+3
δ+4
α
α+1
α+2
α+3
α+4
δ
δ+1
δ+2
δ+3
δ+4
β+5
β+6
β+7
...

States of a Process

- *User view*: A process is executing continuously
- *In reality*: Several processes compete for the CPU and other resources
- A process may be
  - **running**: it holds the CPU and is executing instructions
  - **blocked**: it is waiting for some I/O event to occur
  - **ready**: it is waiting to get back on the CPU



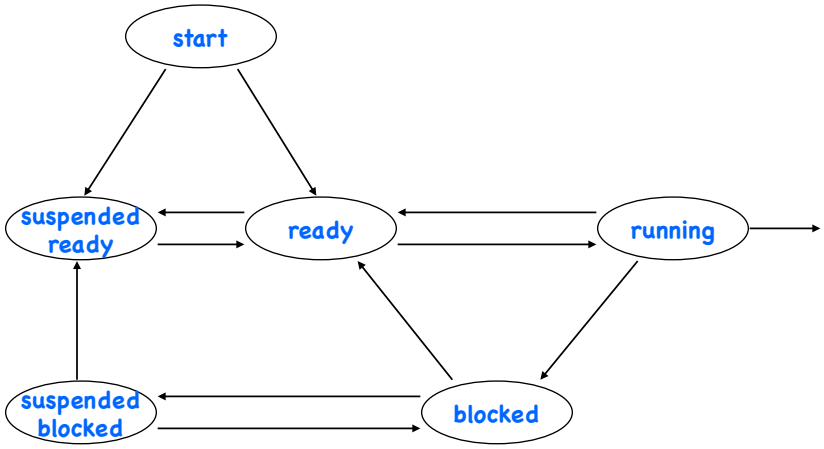
Process Creation

- **When?**
  - Submission of a batch job
  - User logs on
  - Create process to provide service such as printing
  - Spawned by existing processes
- **How?**
  - In UNIX:
    all processes created by **fork()** system call

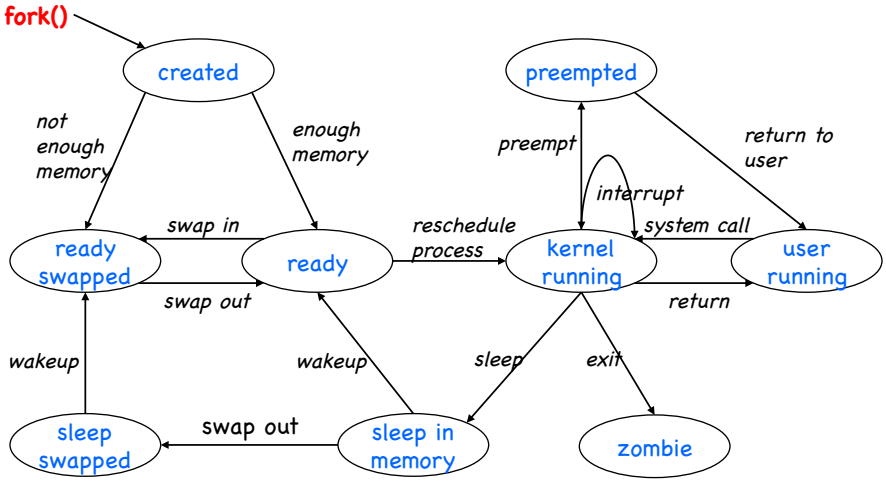
Example: Vanilla Command Interpreter

```
char command[MAX_COMMAND_LENGTH];
do {
  command = read_command(stdin);
  if (fork() != 0) {
    /* parent */
    if (last_char(command) != '&') {
      /* run in foreground, i.e. wait */
      waitpid(-1, &status, ...);
    }
  }
  else {
    /* child */
    execve(command, ...);
  }
} while (strcmp(command, "exit") != 0); /* ?!? */
```

Suspended Processes

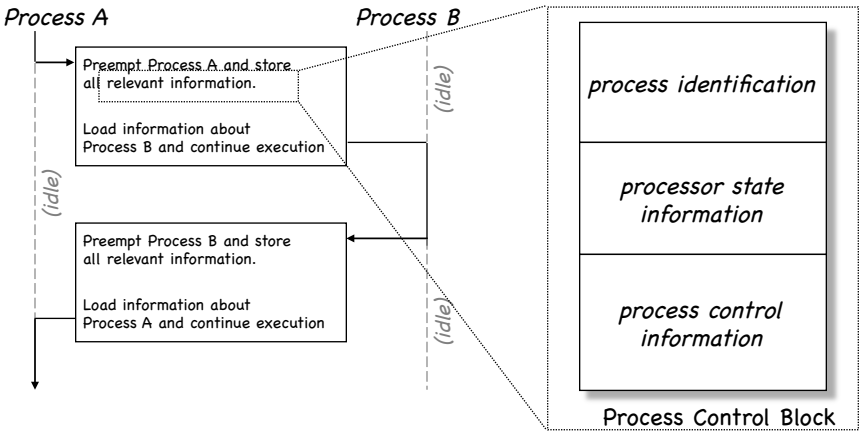


Processes in UNIX



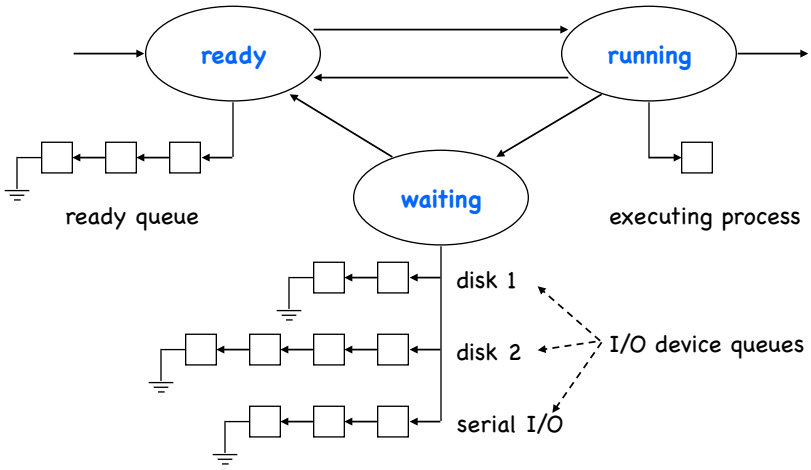
The Process Control Block (PCB)

- Mechanism of a process switch:



- The PCB contains all information specific to a process.

Example for the Use of PCBs: Process Queues

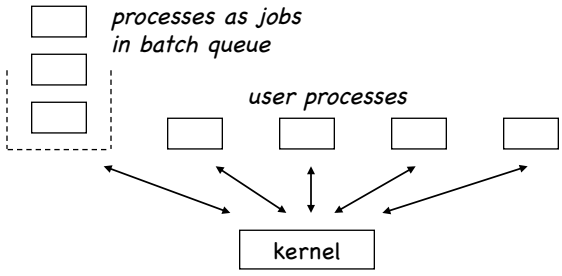


Programs, Processes, Threads

- Programs, Processes, and Threads
- Processes and Threads in UNIX

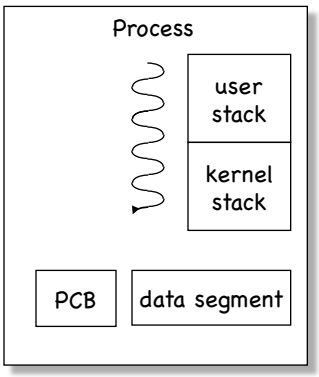
Threads

- Traditionally, processes interact very little:



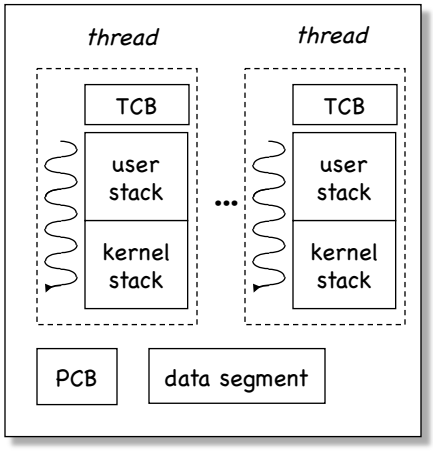
- This is not true in modern systems: Some applications may want to have multiple, tightly-coupled "processes".

Problems with traditional (heavy-weight) processes



- Heavy-weight processes have **separate address spaces**.
  - Process creation is expensive
  - Process switch is expensive
  - Sharing memory among processes is non-trivial

Threads

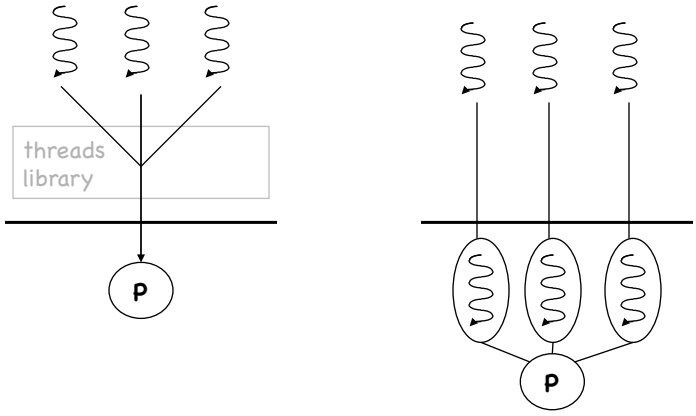


- **Threads share address space:**
  - Thread creation much simpler than process creation (no need to create and initialize address space, etc.)
  - Thread switch simple
  - Threads fully share the address space
- **Convenience**
  - communication between threads
- **Efficiency**
  - multiprogramming within a process
  - multiprocessors

User-Level *vs.* Kernel-Level Threads

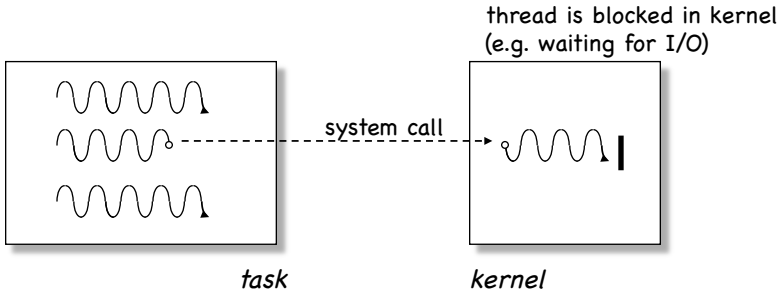
User-Level Threads
kernel is not aware of threads

Kernel-Level Threads
all thread management done in kernel

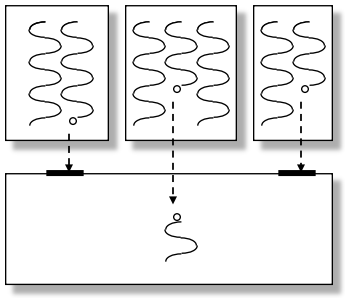


Potential Problems with Threads

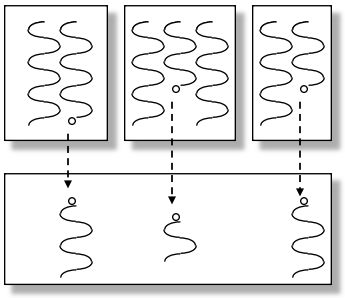
- General: Several threads run in the same address space:
  - Protection must be explicitly programmed (by appropriate thread synchronization)
  - Effects of misbehaving threads limited to task
- User-level threads: Some problems at the interface to the kernel: With a single-threaded kernel, as system call blocks the entire task.



Singlethreaded vs. Multithreaded Kernel



- Protection of kernel data structures is trivial, since only one process is allowed to be in the kernel at any time.



- Special protection mechanism is needed for shared data structures in kernel.

Light-weight Processes