Operating System Concepts

Lecture 4: Process Abstraction

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

Today's class

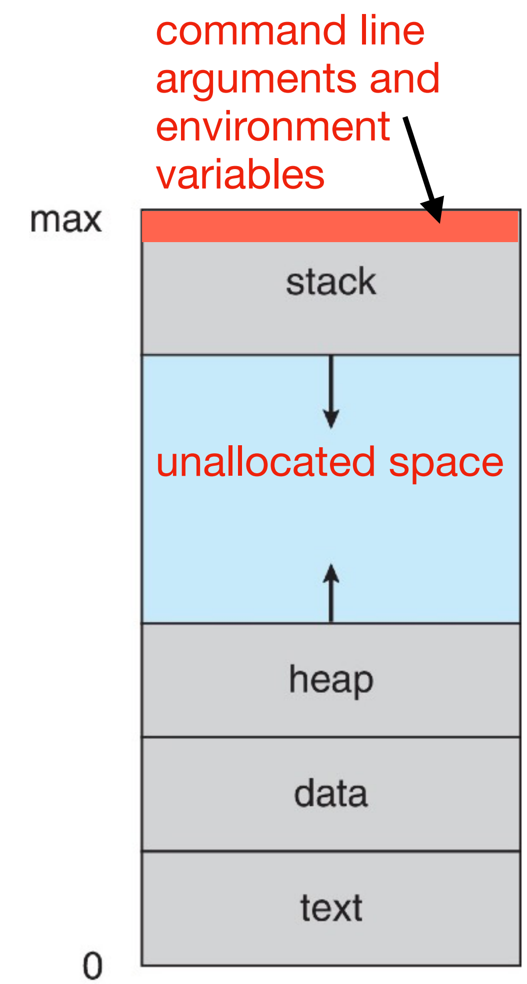
- Process abstraction
  - How does OS create this abstraction?
  - Why is it useful?
  - What happens during a context switch? What are the roles of the **dispatcher** and **scheduler**?

Process abstraction

- A process is a program during execution. It is an execution environment with restricted rights
  - has its own resources (CPU registers, memory to contain program code and data, file descriptors, etc.)
  - encapsulates one or more **threads** that share process resources
  - is characterized by a **unique identifier** (PID)
- Different processes may run different instances of the same program (process $\neq$ program)
  - e.g., you can run several instances of a web browser
- Why do we need this abstraction?
  - necessary for concurrent execution and protection

Memory layout for process

- Multiple sections of a process
  - text section containing the program code
  - data section containing global variables (initialized and uninitialized)
  - stack containing temporary data, function parameters, return addresses, and local variables
  - heap containing memory that is dynamically allocated at runtime using **`malloc`** library function or the **`brk/sbrk`** system call



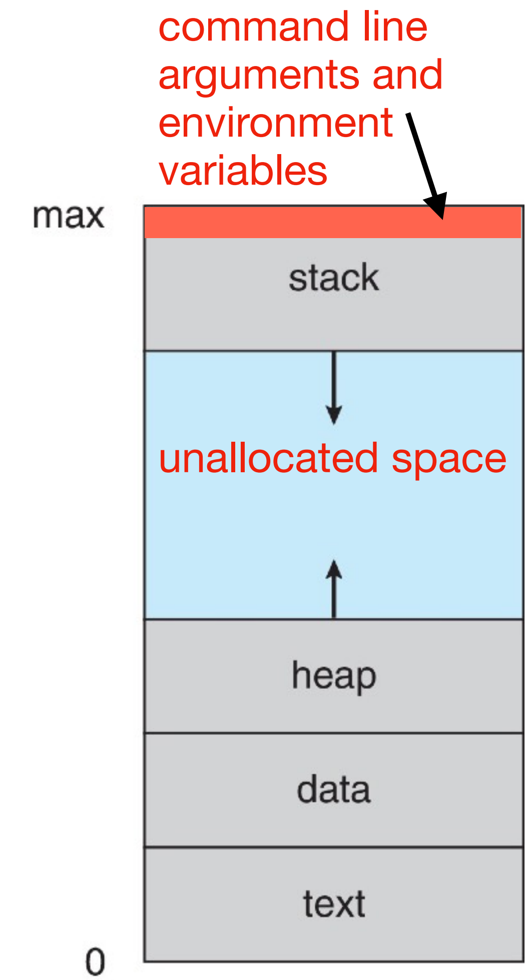
Programmer's view of memory: single space containing this one process only

Which variables are allocated on the stack?

```
#include <stdio.h>
void foo (int n) {
   int i, a[5], *b;
   if (n == 0) return;
   b = new int[n];
   printf ("foo(%d): %p,%p,%p,%p \n", n, &i, a, &b, b);
   foo(n-1);
}
int main ( ) { foo(10); }
```

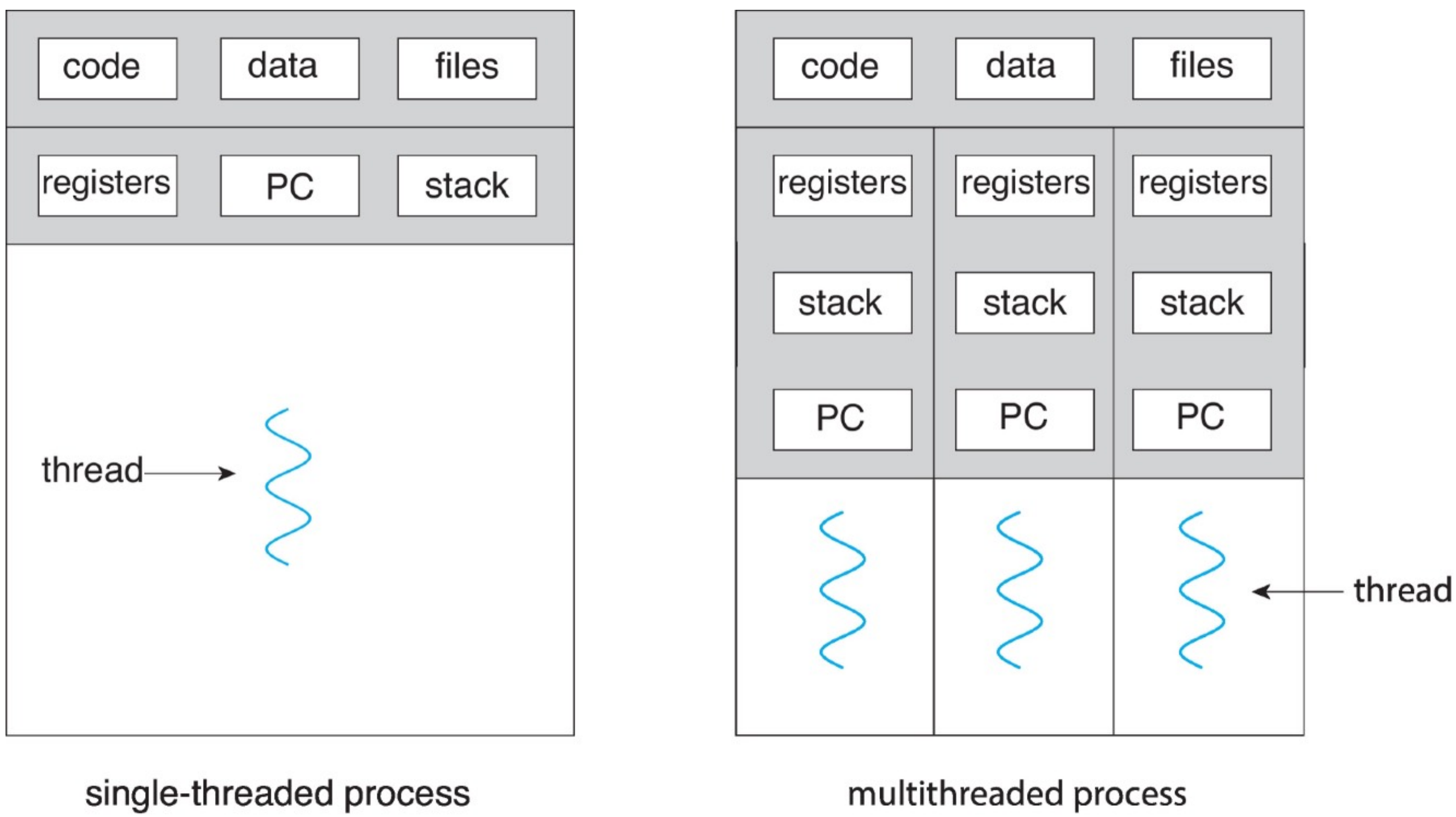
Memory layout of a process

- Multiple sections of a process
  - text section (fixed size) containing the program code
  - data section (fixed size) containing global variables (initialized/uninitialized)
  - stack containing temporary data, function parameters, return addresses, and local variables
  - heap containing memory that is dynamically allocated at runtime using `malloc` library function or the **`brk/sbrk`** system call
- Each process has a distinct and isolated **address space** (i.e., set of addresses that can be accessed by its code)
  - addresses in the executable file are as if it is loaded at memory address 00000000
  - these addresses need to be adjusted when the program is **relocated** to somewhere else
  - no process can read or write memory of another process
- Hardware translates virtual addresses to physical addresses



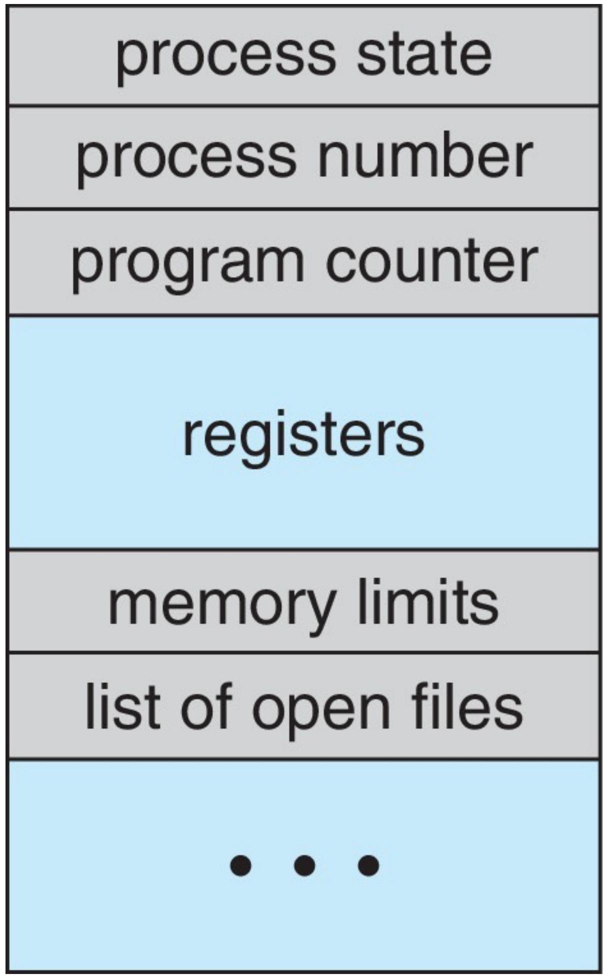
Single vs. multi-threaded process

- A thread is a sequential execution stream of instructions
- The address space of the process is shared among its threads
  - sharing heap, text, static data sections in addition to file descriptors
- Threads can execute simultaneously on different cores of a multicore system
  - e.g. in a word processor you can simultaneously type a character and run the spell checker!

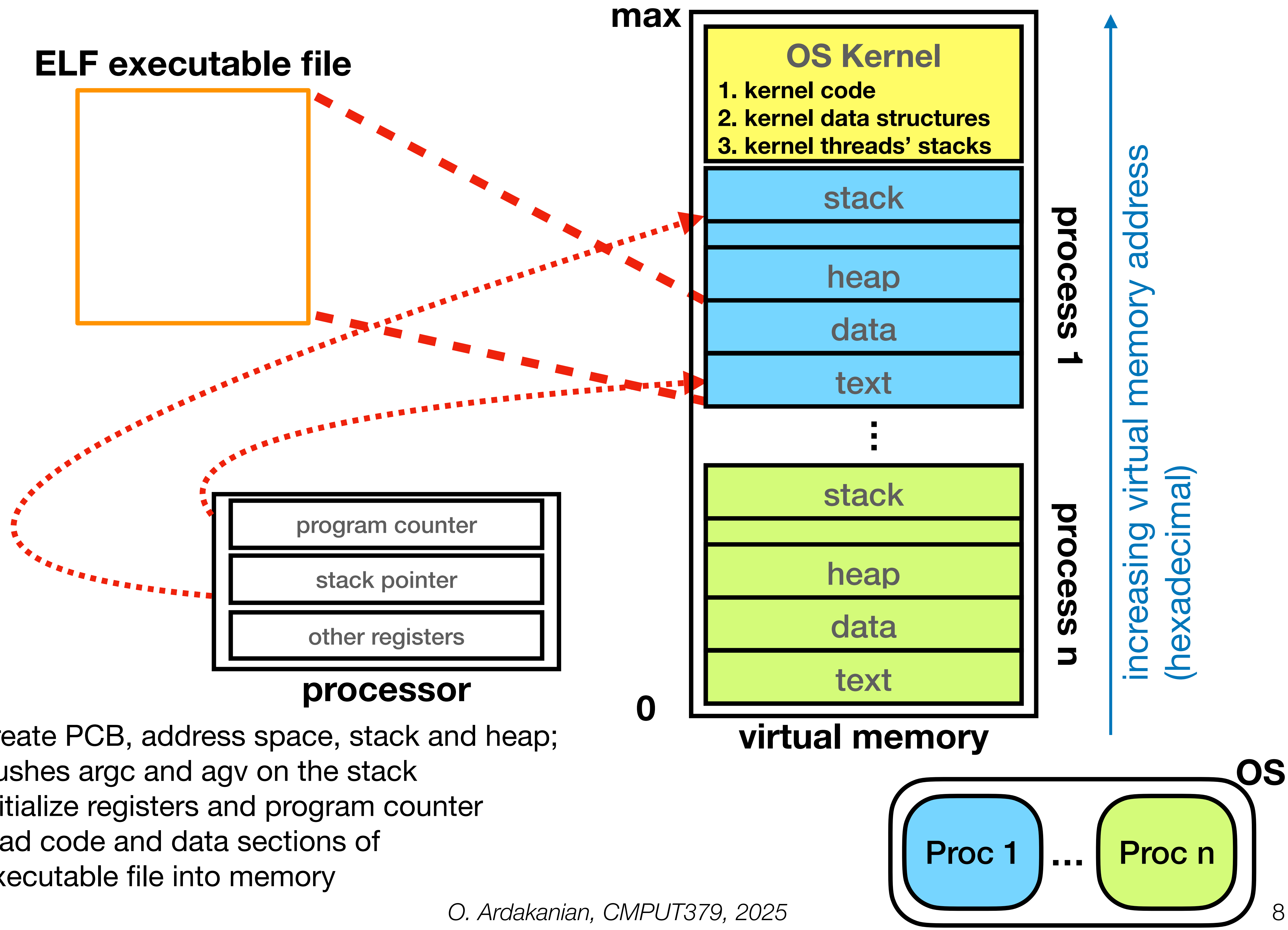


Process control block

- For every process, OS keeps the state of process execution (metadata) in a **process control block** (PCB)
- PCB is a kernel data structure in memory; it represents run-time information about the process, defining its **context**
  - Process status (running, ready, blocked/waiting)
  - Process ID (PID) and its children's PIDs
  - CPU registers, including program counter (PC), stack pointer (SP), base/relocation and limit registers, page-table base register (PTBR), and general-purpose registers
  - Thread control block(s)
  - Accounting information (e.g., execution time, time elapsed since start)
  - Scheduling information (e.g., priorities, queue pointers for state queues)
  - Set of OS resources in use (e.g., list of open files, I/O devices allocated to the process)
  - Current working directory
  - Username of owner
  - ...
- PCB in Linux is represented by the C structure called `task_struct`
  It is defined in <linux/sched.h>
  - `task_struct` contains `mm_struct` which represents the address space of a given process



Loading — revisited

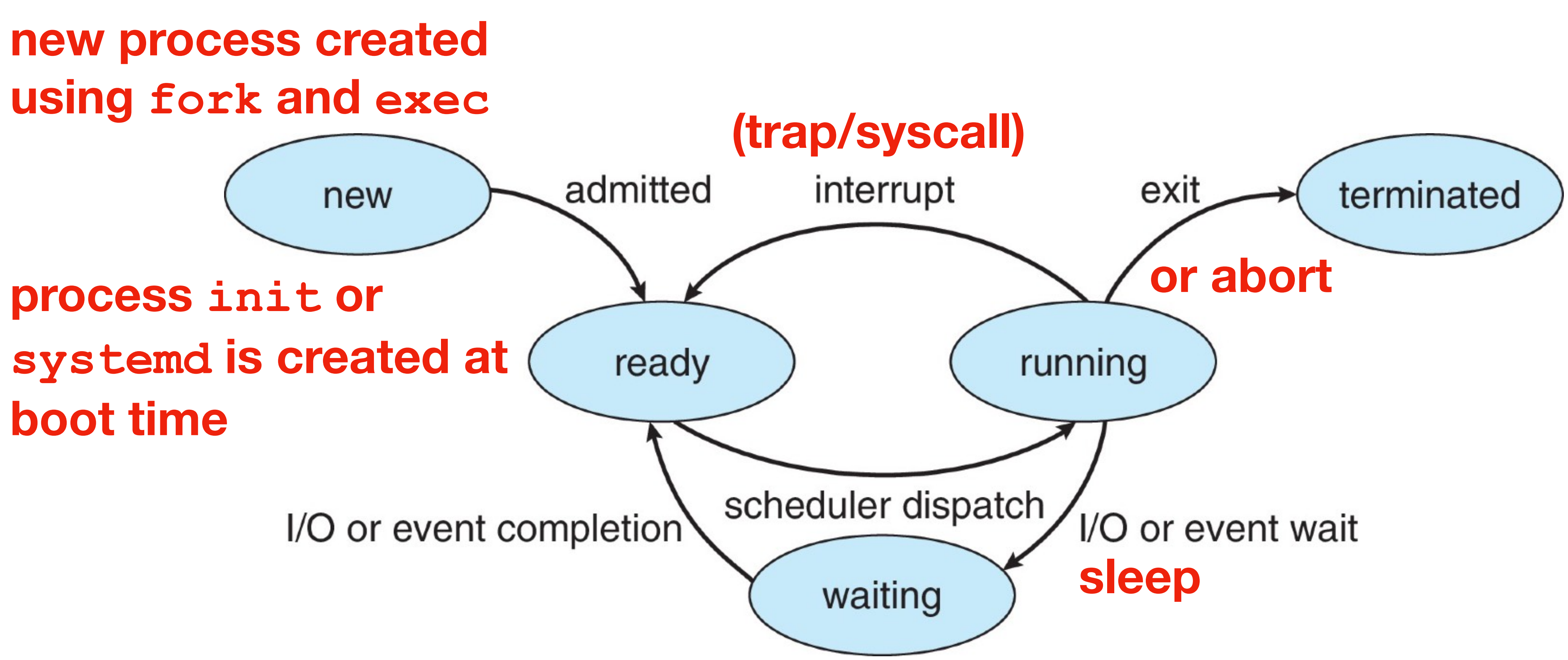


1. create PCB, address space, stack and heap; pushes argc and agv on the stack
2. initialize registers and program counter
3. load code and data sections of executable file into memory

Keeping track of processes

- OS juggles many processes at a time
  - only one process can be running per core (the kernel maintains a pointer to this process)
  - but many processes can be in ready and waiting states
- OS puts PCBs of the active processes in appropriate queues
  - ready queue (organized by the process-scheduling priority, the arrival time, etc.)
  - wait queue for each device
  - **zombie** queue (terminated processes that have an entry in the process table are kept in that queue until they are reaped by their parent)
- state change happens as a result of the process actions (e.g., termination or invoking system calls), OS actions (scheduling), and external actions (hardware interrupts)

Process/thread lifecycle

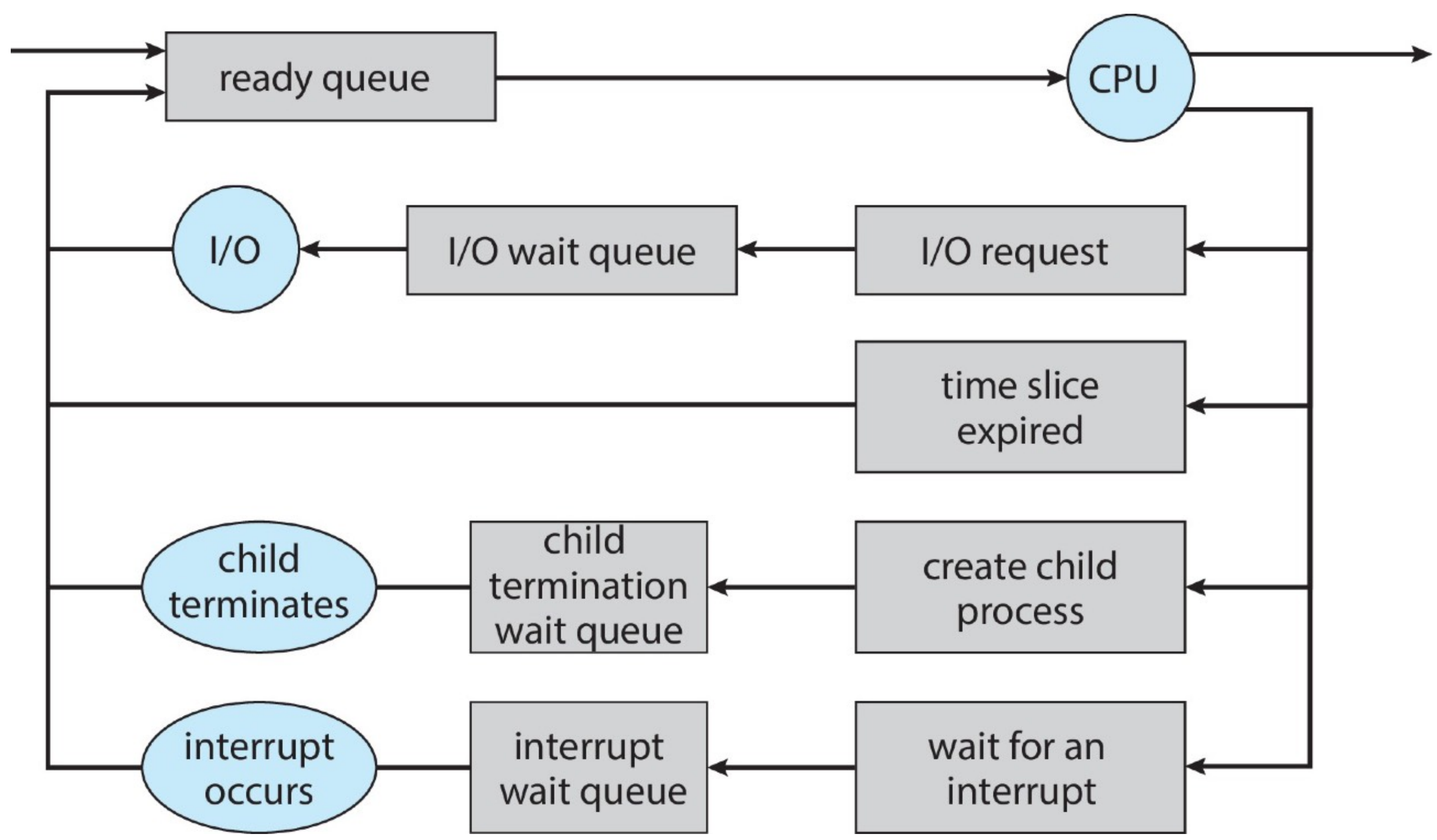


Zombie and orphan processes

- a process that has terminated, but its parent has not yet read its exit status becomes zombie
- a process becomes orphan when its parent terminates while it is still running

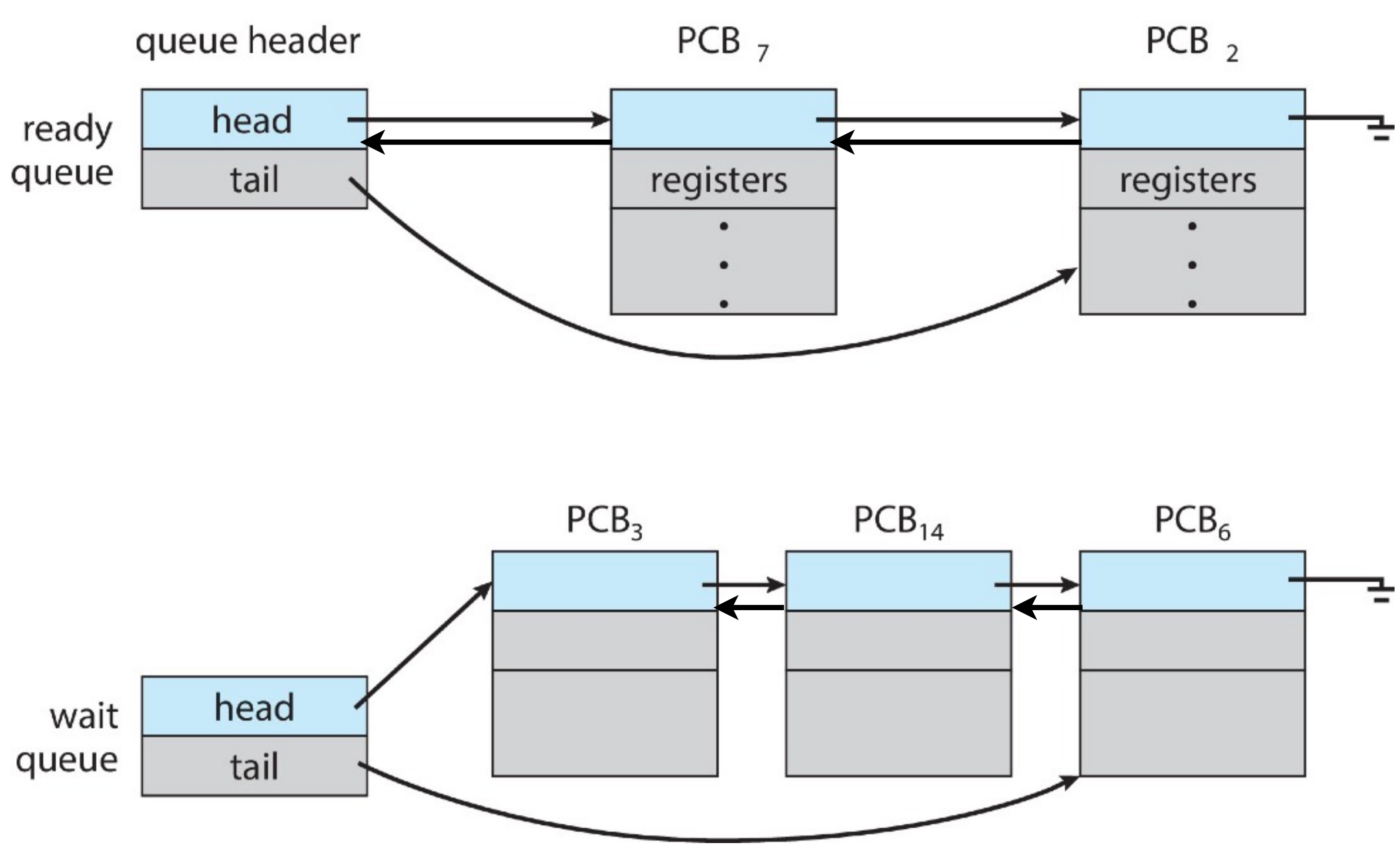
Multiprogramming

- Only <u>one</u> process is active at a time (per CPU core)
- OS gives out CPU to different processes in the ready queue
  - the number of processes currently in memory is the **degree of multiprogramming**



Scheduling

- The scheduler maintains a data structure (e.g., a **doubly linked list**) of PCBs
  - PCBs are moved from one queue to another queue
  - job queue: every process in the system
  - ready queue: processes residing in the main memory, waiting to run on CPU
  - device queue: processes waiting for a particular I/O device, each device has its own queue



Scheduling

- The scheduler selects a process among processes in the ready queue to run
  - selected process runs on CPU
  - if no process left in the ready queue, the CPU runs an idle process
  - scheduling can be performed for fairness, minimum latency, providing real-time guarantees, etc.
- The (short-term) scheduler makes a decision very fast (<10 ms) and is called very often (e.g., every 100ms)

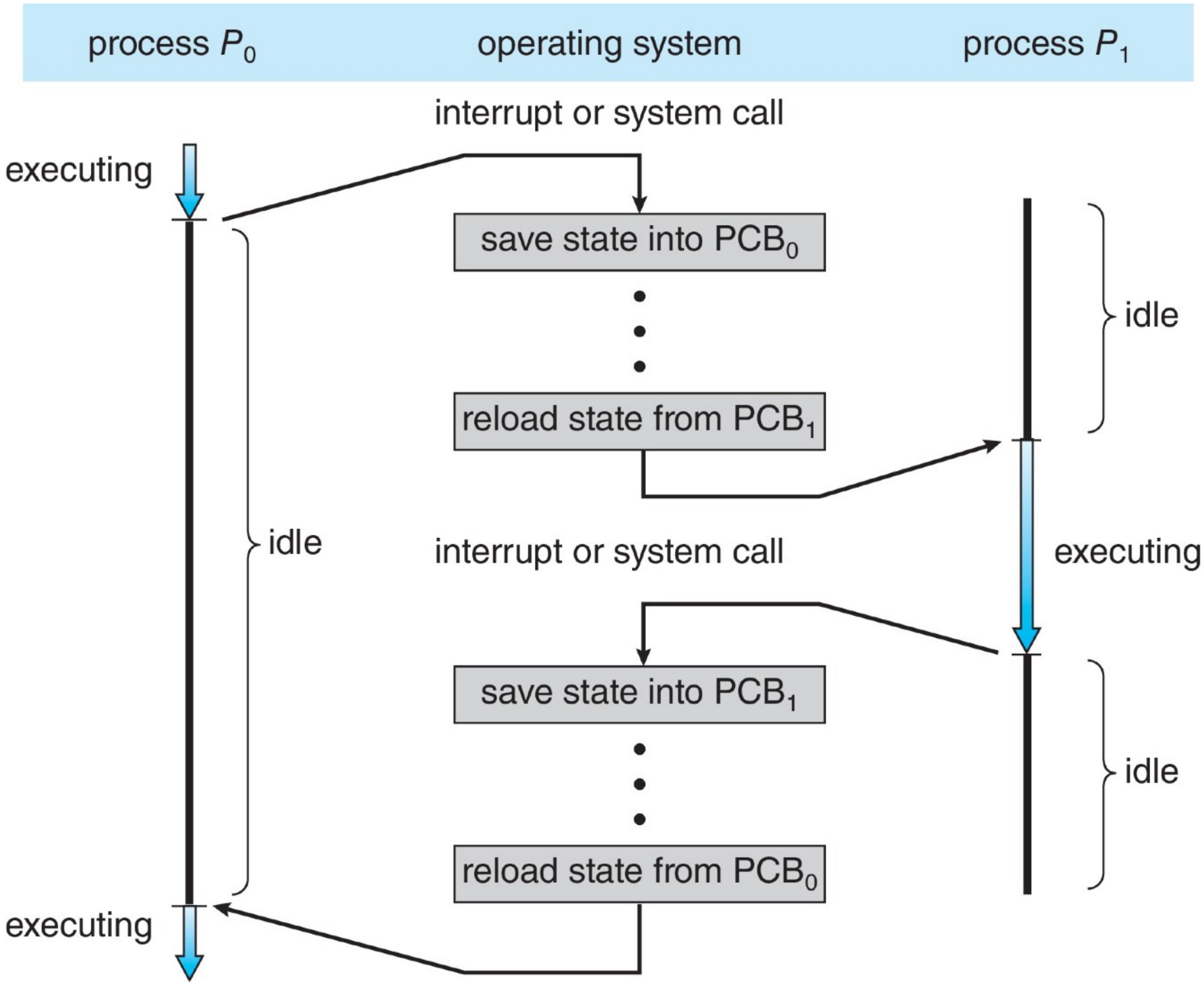
Context switching

- Definition: stopping a process and starting another one
  - it is a relatively expensive operation
  - time-sharing systems may do hundreds of context switches per second
  - the **context** includes the value of CPU registers, the process state, and memory management information
- OS starts executing a process in the ready state by loading hardware registers (PC, SP, etc) from its PCB
  - while a process is running, the CPU modifies the Program Counter (PC), Stack Pointer (SP), registers, etc.
- When OS stops executing a process, it saves the values of the registers (PC, SP, etc.) into its PCB
  - so that they can be restored the next time the process is selected for running on the CPU

Tradeoff

- Context switching is pure **overhead,** i.e., the system does no useful work
  - the cost of a context switch and the time between switches are closely related
    - but fast context switching is necessary for responsiveness
  - OS must balance the context switch frequency with the scheduling requirement (response time, fairness, etc.)

Context switching



Scheduler and dispatcher

- The scheduler selects a process to run next
- The dispatcher makes it happen
  - performs context switching
  - switches to user mode
  - jumps to the proper location in the user program