Processes and thread management

Operating Systems Design

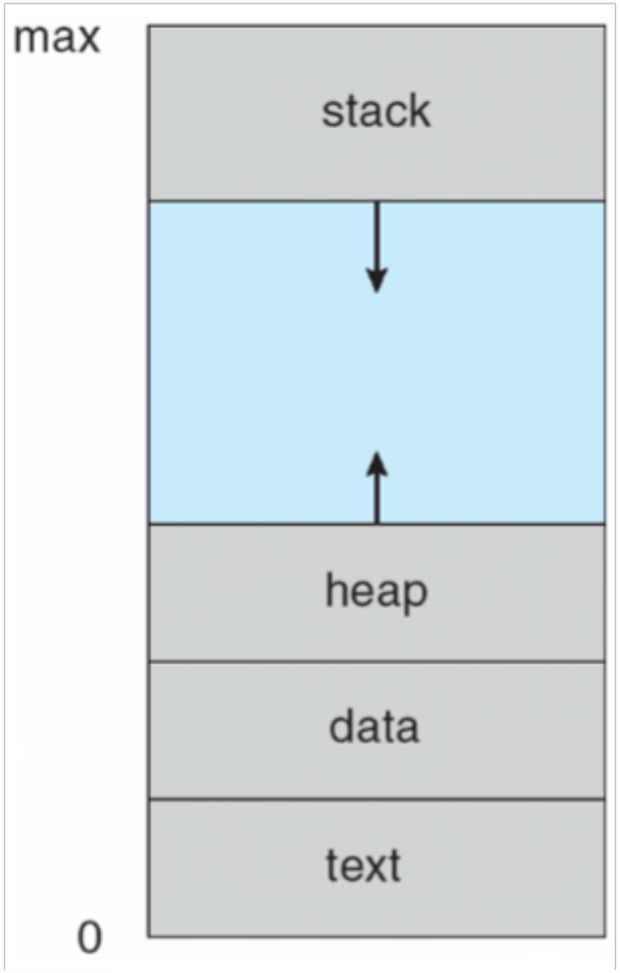
References

- Silberschatz ***Operating System Concepts Ninth Edition*** 2012. Chapters 3, 4, and 6

Process Concept

- An operating system executes a variety of programs:
  - Batch system – jobs
  - Time-shared systems – user programs or tasks
- Textbook uses the terms *job* and *process* almost interchangeably
- Process – a program in execution; process execution must progress in sequential fashion
- A process includes:
  - program counter
  - stack
  - data section

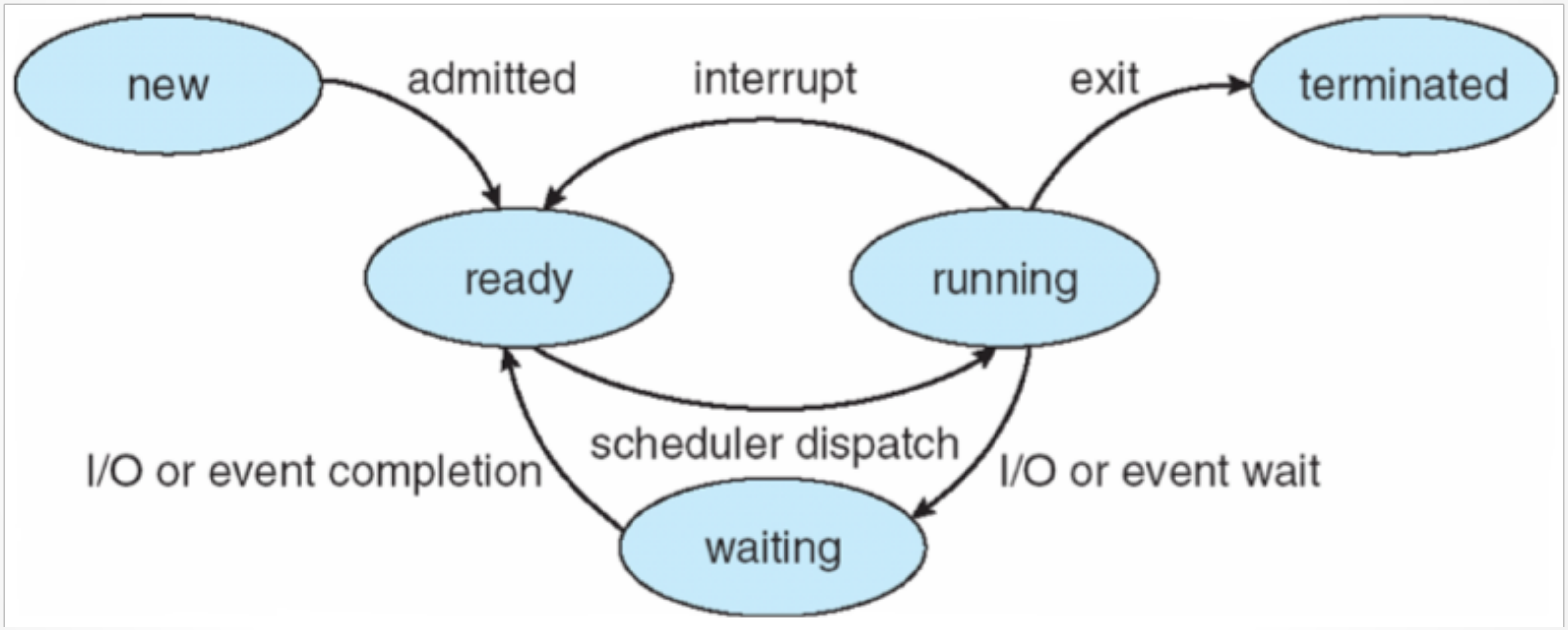
Process in Memory



Process State

- 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

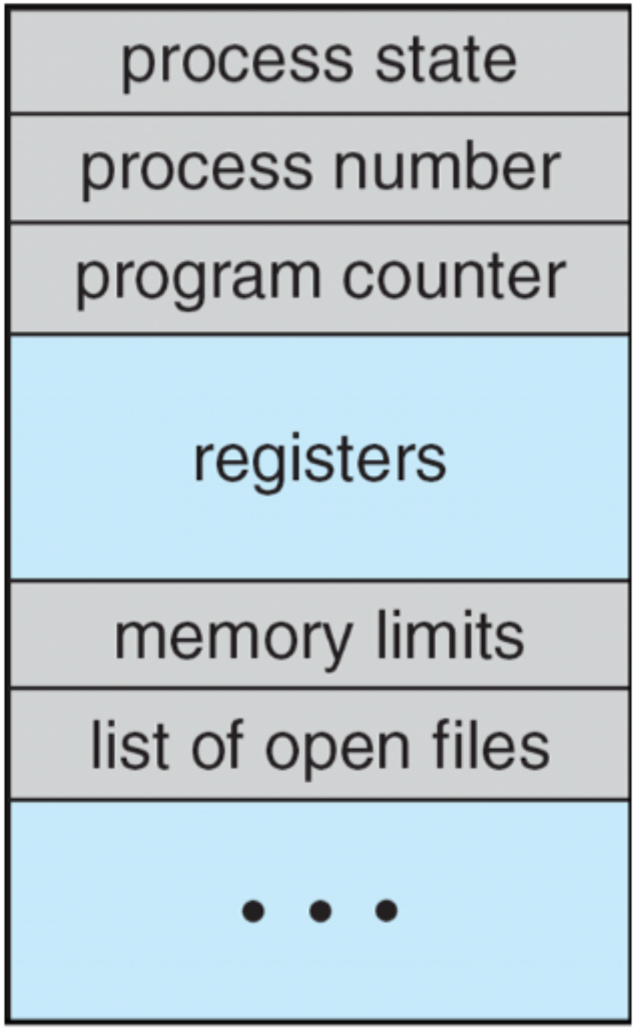
Diagram of Process State



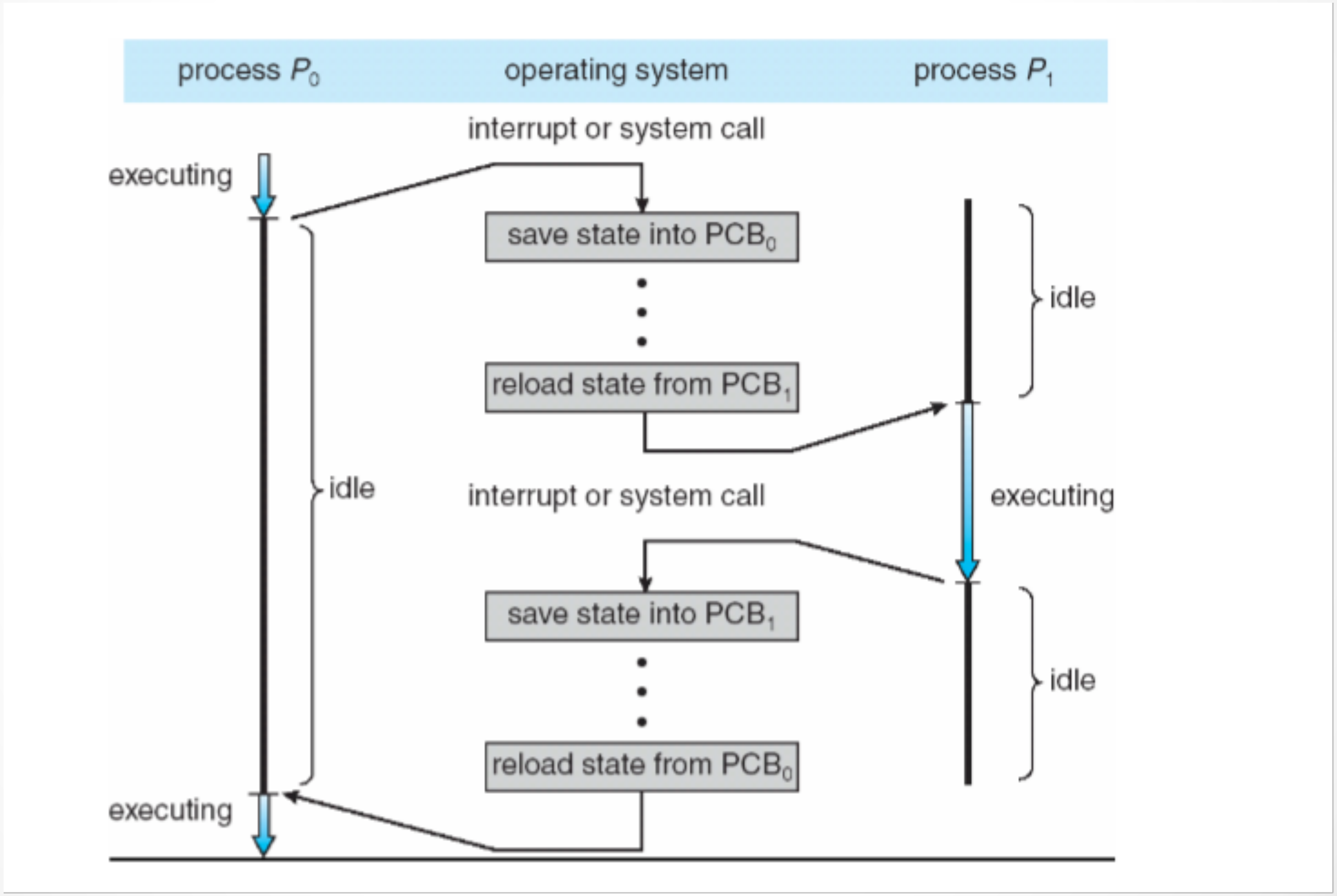
Process Control Block (PCB)

- Information associated with each process
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information
- PCBs are stored in a global **process table**

Process Control Block (PCB)



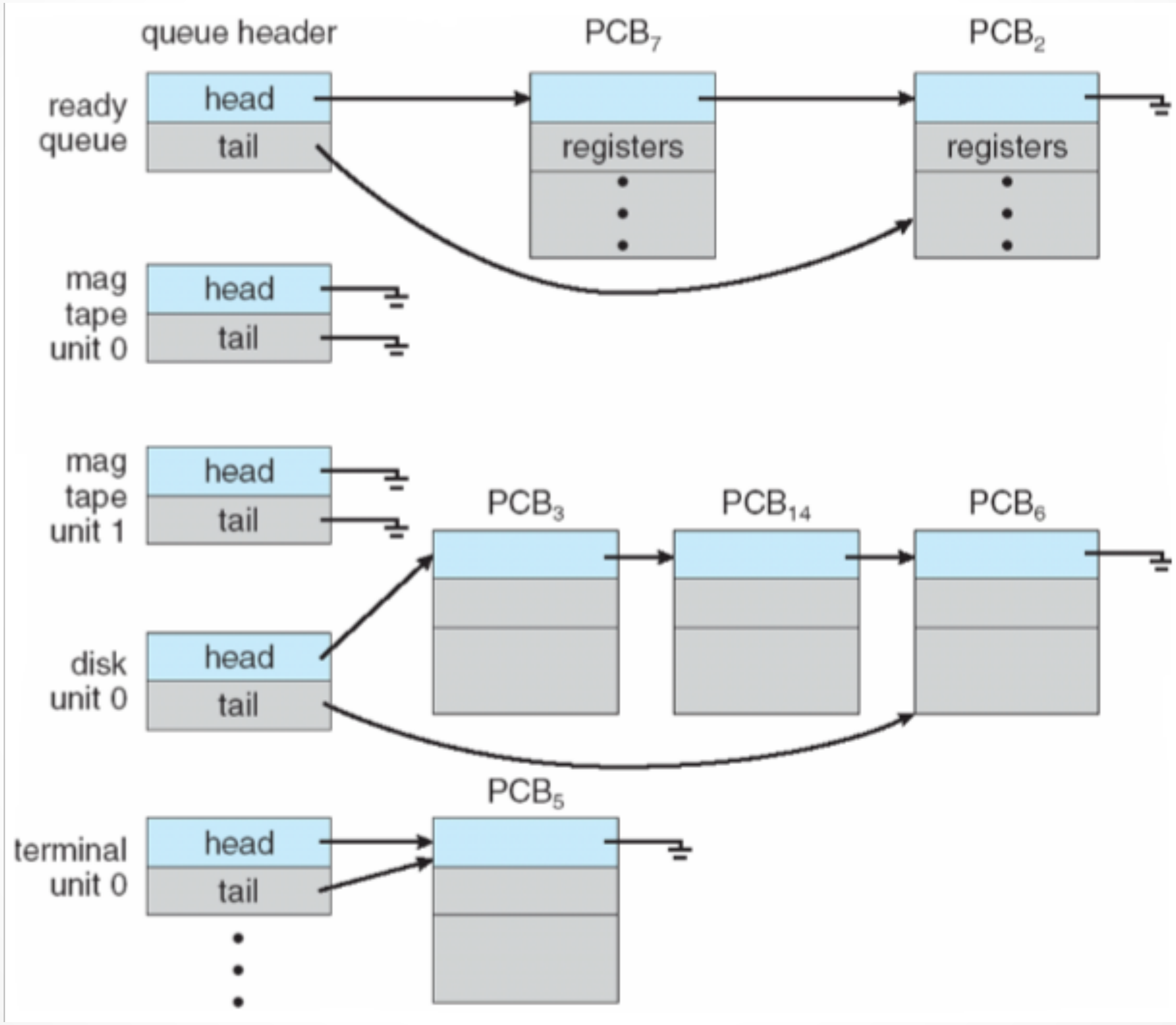
CPU Switch From Process to Process



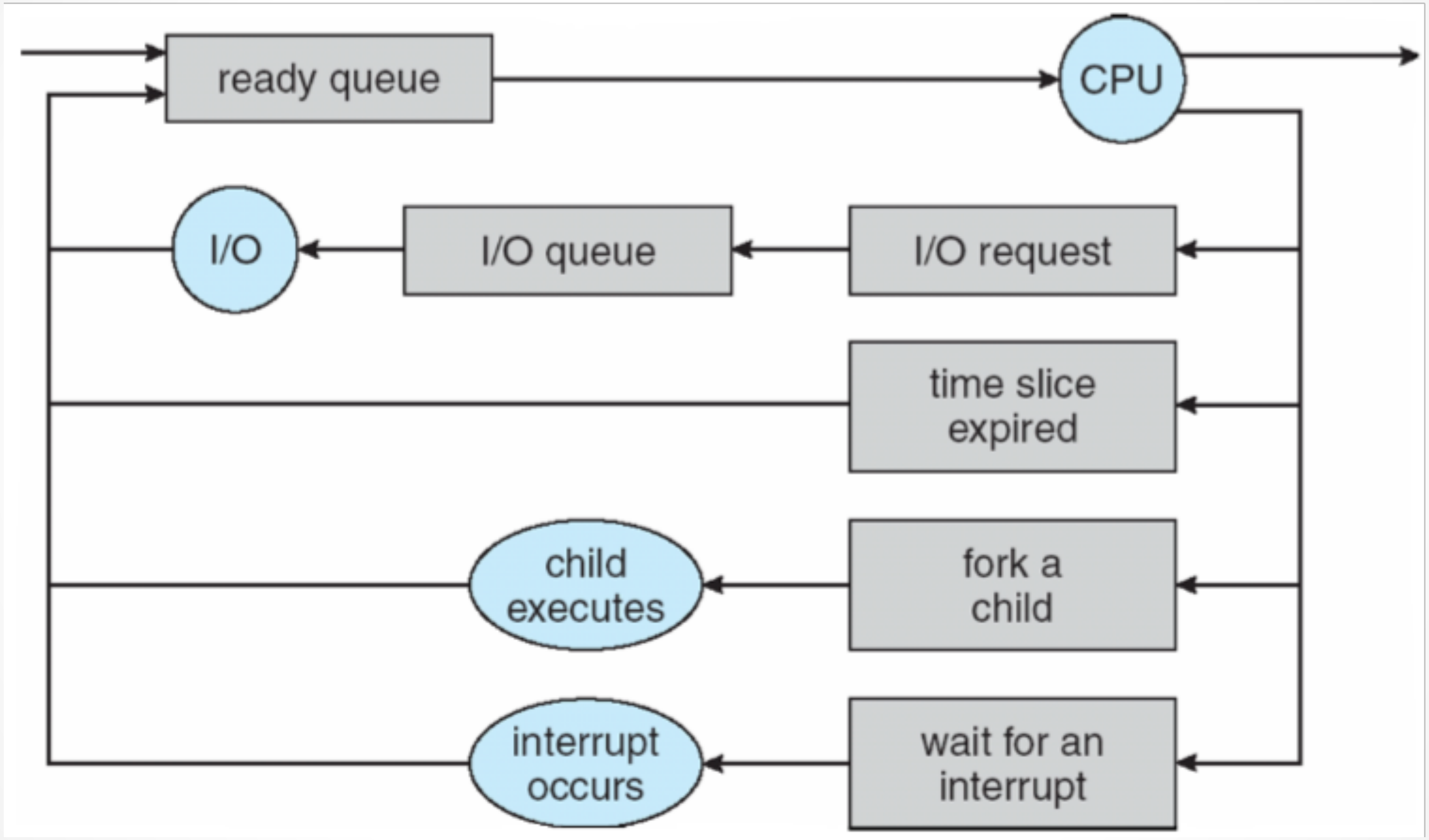
Process Scheduling Queues

- **Job queue** – set of all processes in the system
- **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
- **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Ready Queue And Various I/O Device Queues



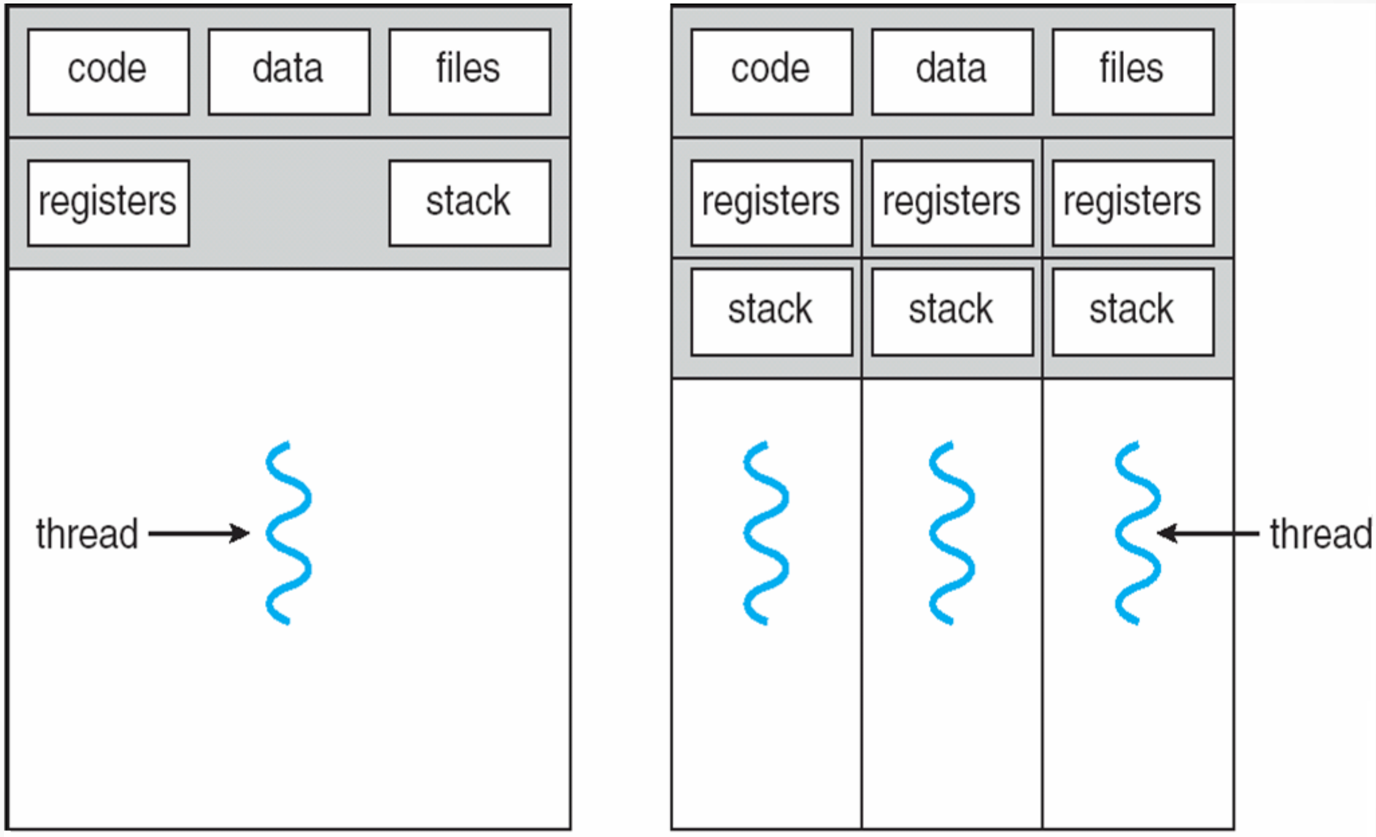
Representation of Process Scheduling



Threads and processes

- Thread = a lightweight process
- Threads of one process share
  - Data section (heap)
  - Code
  - Open files
  - Signals
- Each thread of a process has his own
  - Stack
  - Register state

Single and Multithreaded Processes



single-threaded process

multithreaded process

Threads

- Two types of thread implementations
  - User threads
  - Kernel threads
- A thread library provides programmer with API for creating and managing threads
- In the first programming assignment you will implement a user thread library

User Threads

- Thread management done by user-level thread library
- No kernel support
  - The kernel is not aware of the existence of several threads
    - If one blocks for I/O the kernel may infer that the whole process blocks for I/O and schedule another process

Kernel Threads

- Supported by the Kernel
  - If one thread within one process blocks for I/O the kernel knows how to schedule another thread from the same process

- Most OS support kernel threads
  - Windows
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Scheduler

- Selects from among the processes/threads in memory that are ready to execute, and allocates the CPU to one of them
- CPU scheduling decisions may take place when a process/thread:
    1. Switches from running to waiting state
    2. Switches from running to ready state
    3. Switches from waiting to ready
    4. Terminates
- Scheduling under 1 and 4 is **nonpreemptive (voluntary)**
- All other scheduling is **preemptive (involuntary)**

Dispatcher (activator)

- Dispatcher module gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- Dispatch latency – time it takes for the dispatcher to stop one process and start another running

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)

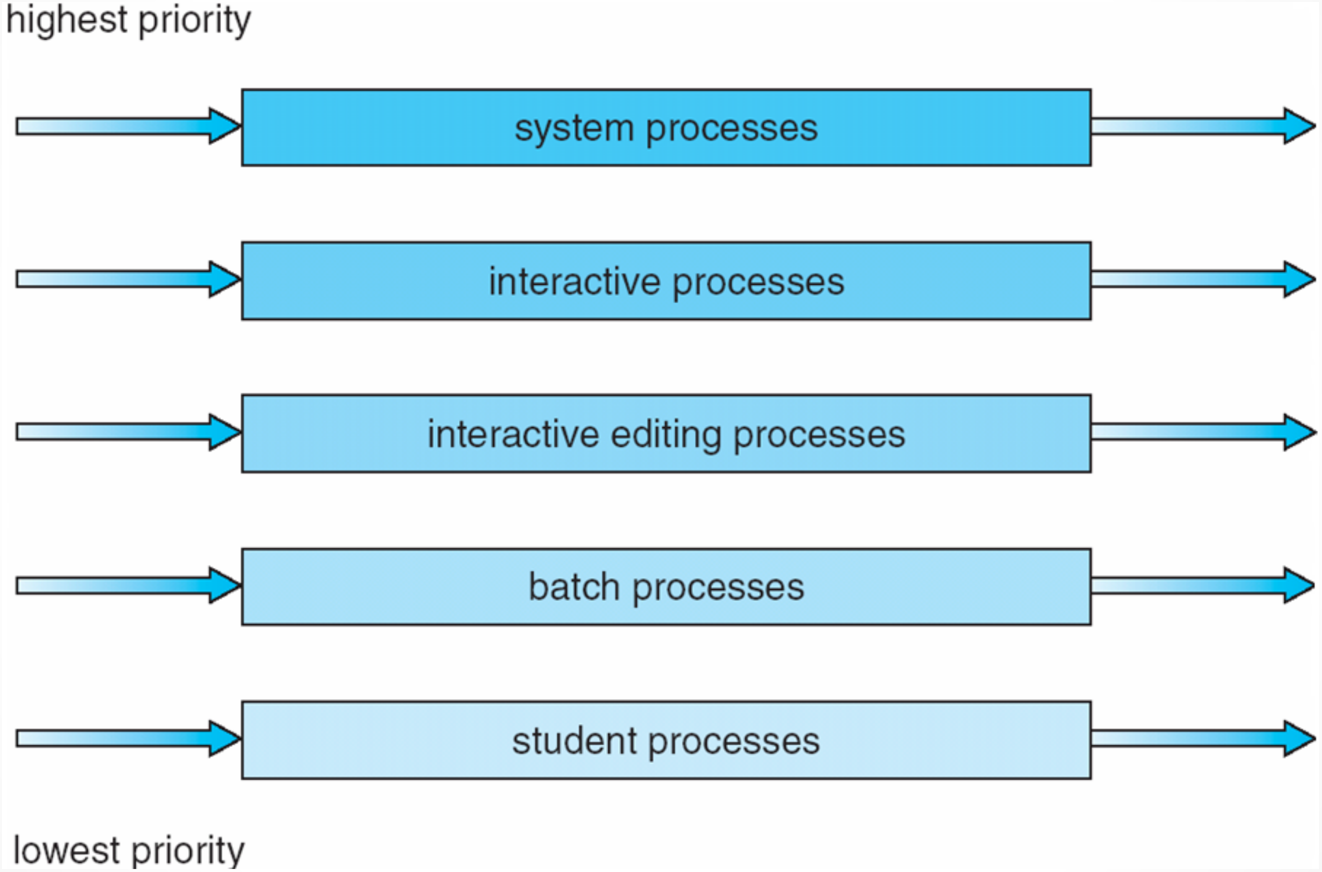
Scheduling Algorithm Optimization Criteria

- Max CPU utilization
- Max throughput
- Min turnaround time
- Min waiting time
- Min response time

Main scheduling algorithms

- FCFS (non-preemptive)
- Round-Robin (preemptive)
- SJF
  - Preemptive
  - Non-preemptive
- Priority-based
  - Preemptive
  - Non-preemptive
- Multi-level queue

Multilevel Queue Scheduling



Task context and scheduling

- The context of a user task is standardized by POSIX

```
typedef struct ucontext {
        struct ucontext uc_link; /* context that will be resumed 
when the current context terminates*
        sigset_t          uc_sigmask; /* set of  signals  blocked 
in  this  context */
        stack_t           uc_stack; /* stack  used  by this 
context */
        mcontext_t        uc_mcontext; /* machine-specific 
representation  of 
       the  saved  context */
        ...
} ucontext_t;
```

Manipulation of the user context

- Getcontext(ucontext_t *ucp): initializes a context
- Setcontext(const ucontext_t *ucp): starts to run the ucp context on the CPU
- Makecontext(ucontext_t *ucp, void *func(), int argc, …): creates a new thread using the ucp context previously created by getcontext
- Swapcontext(ucontext_t *old_ucp, ucontext_t *new_ucp): starts to run the new_ucp context on the CPU and saves the currently running context into old_ucp