Week 11 Lecture 1

NWEN 241
Systems Programming

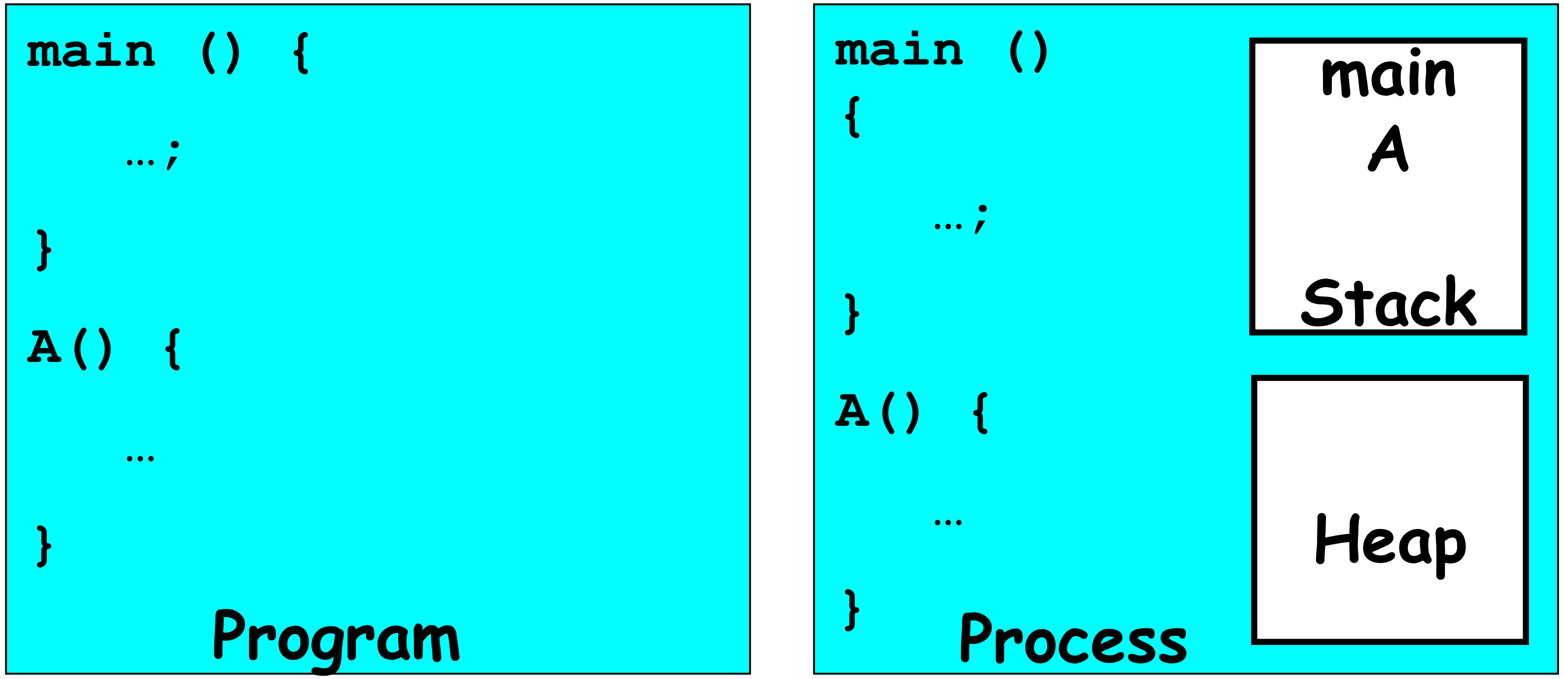
Alvin C. Valera

alvin.valera@ecs.vuw.ac.nz

Content

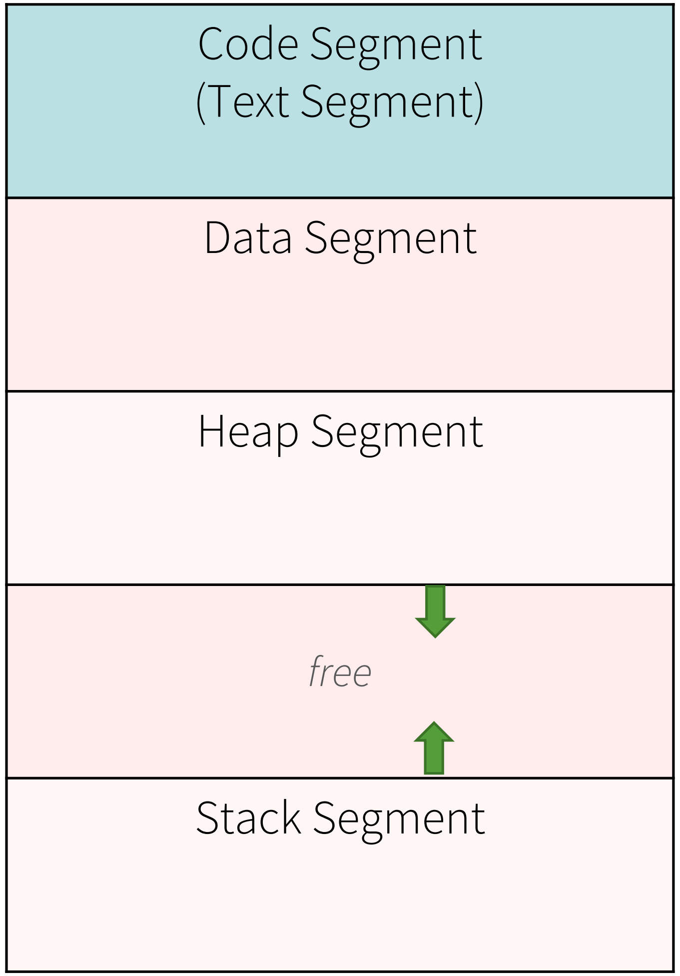
- Process management in the operating system
- Interprocess communication

Recap: Process vs program



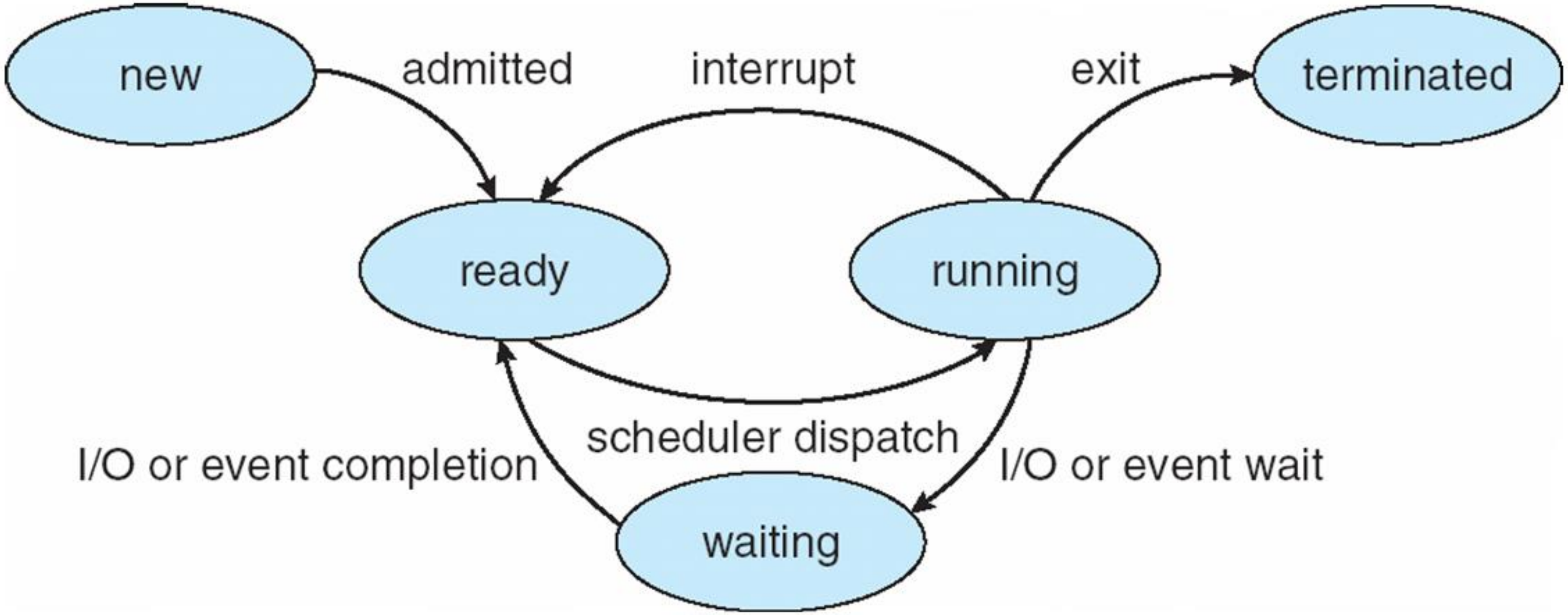
- Program is static, with the potential for execution
- Process is a program in execution and have a state
- One program can be executed several times and thus has several processes

Process in memory



- **Text / Code Segment**
  - Contains program's machine code
- Data spread over:
  - **Data Segment** – Fixed space for global variables and constants
  - **Stack Segment** – For temporary data, *e.g.*, local variables in a function; expands / shrinks as program runs
  - **Heap Segment** – For dynamically allocated memory; expands / shrinks as program runs

Recap: Process lifecycle

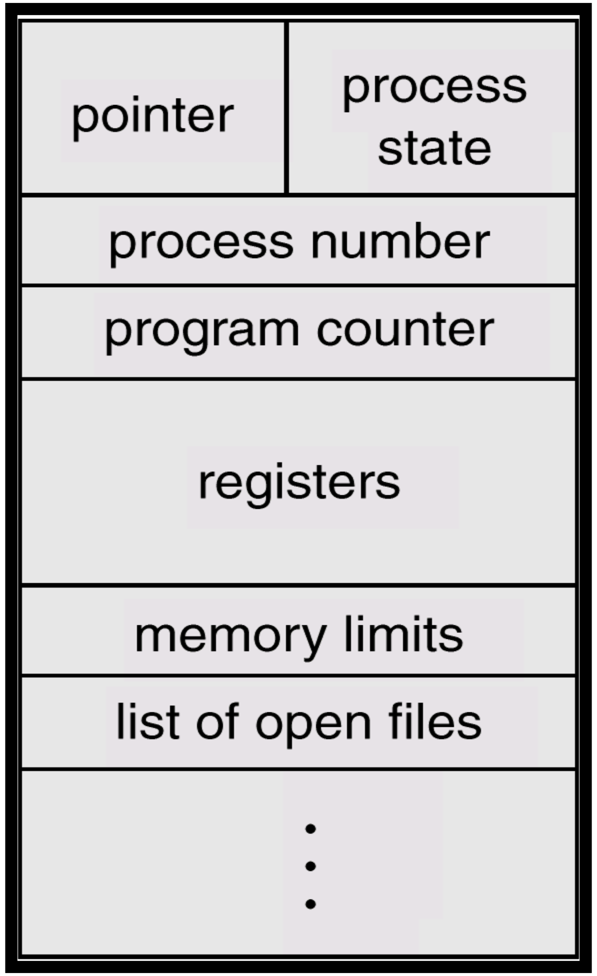


Process lifecycle

- 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

Process control block

- Information associated with each process
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information
- A process is named using its process ID (PID) or process #
- Data is stored in a process control block (PCB)



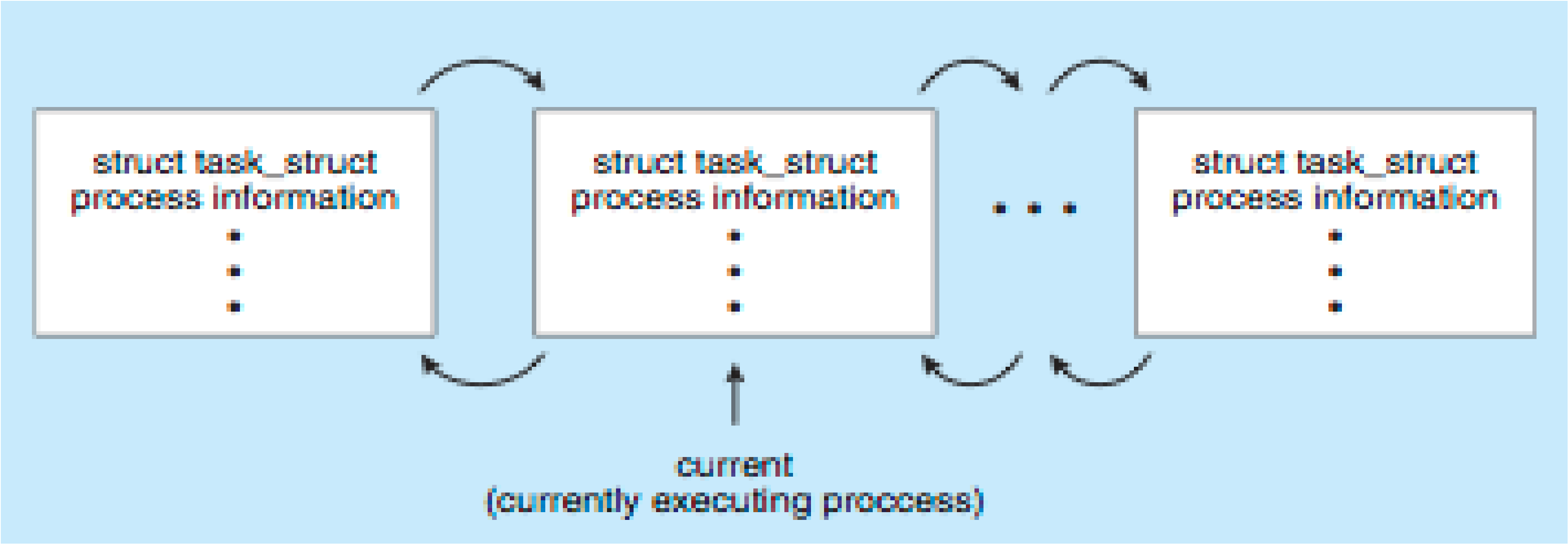
Process representation in Linux

- Represented by structure `task_struct`
  - See https://github.com/torvalds/linux/blob/master/include/linux/sched.h for more information
- Some of the structure members

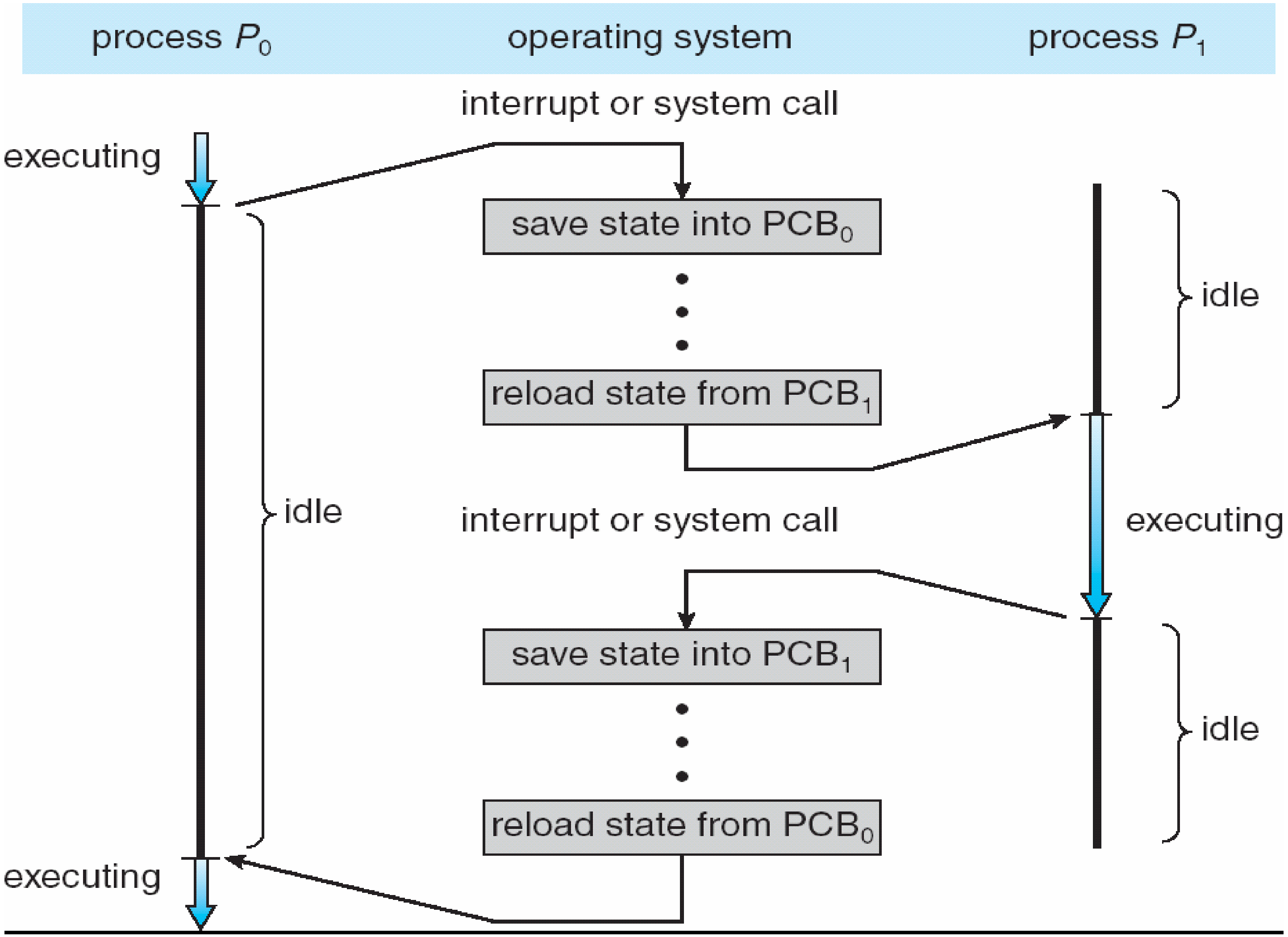
```
pid t_pid; /* process identifier */
long state; /* state of the process */
unsigned int time_slice /* scheduling information */
struct task_struct *parent; /* this process's parent */
struct list_head children; /* this process's children */
struct files_struct *files; /* list of open files */
struct mm_struct *mm; /* address space of this process */
```

Process representation in Linux

- Represented by structure `task_struct`
  - See https://github.com/torvalds/linux/blob/master/include/linux/sched.h for more information



Process switching



Context switch

- When CPU switches to another process, the system must **save the state** of the old process and load the **saved state** for the new process via a **context switch**

- Context-switch time is overhead
  - System does no useful work while switching
  - The more complex the OS and the PCB -> longer the context switch

- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU -> multiple contexts loaded at once

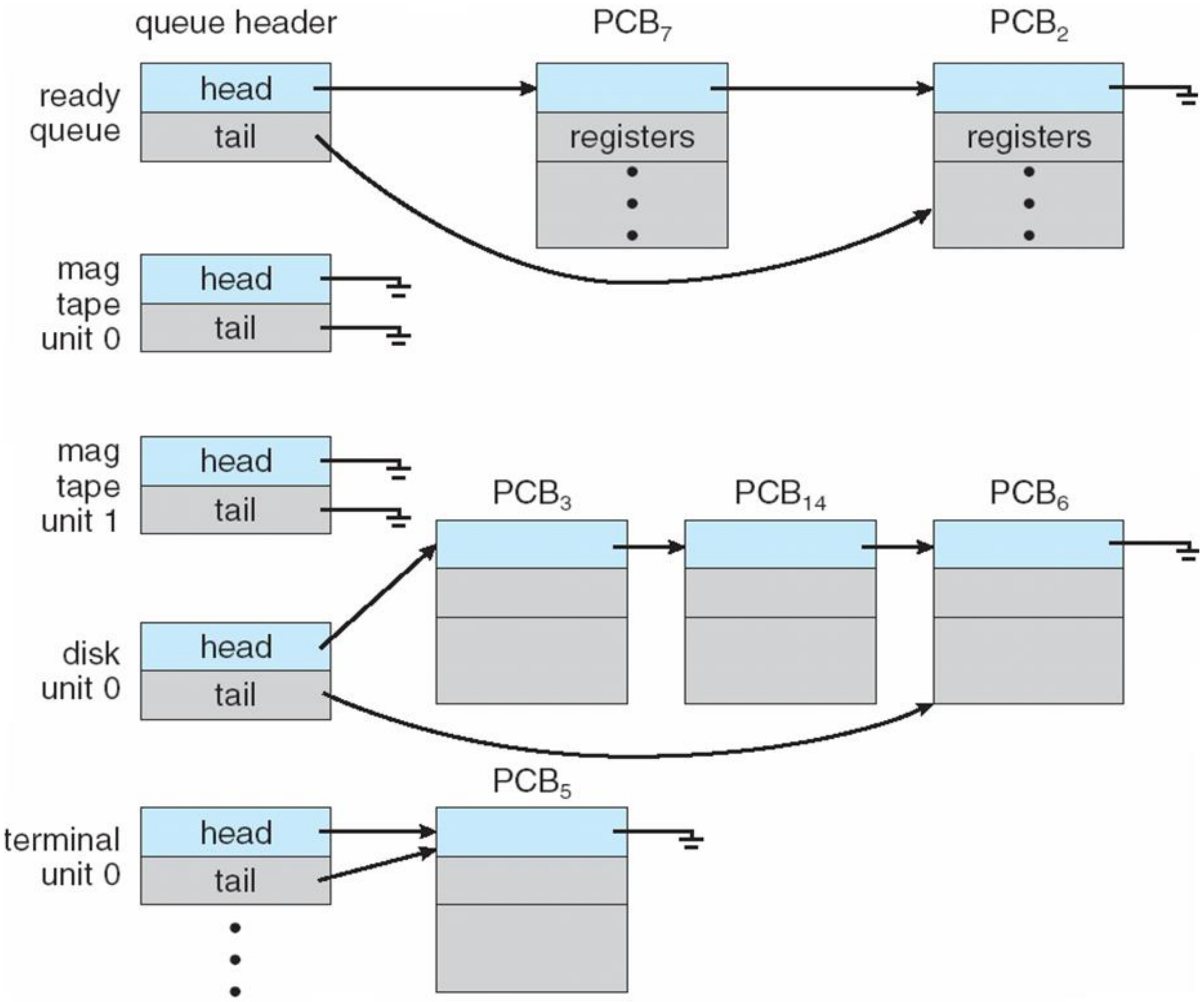
Why perform process switching?

- A running process may reach an instruction requiring user input
- While waiting for input, CPU is not doing anything
- To maximize CPU use, quickly switch ready processes onto CPU for time sharing

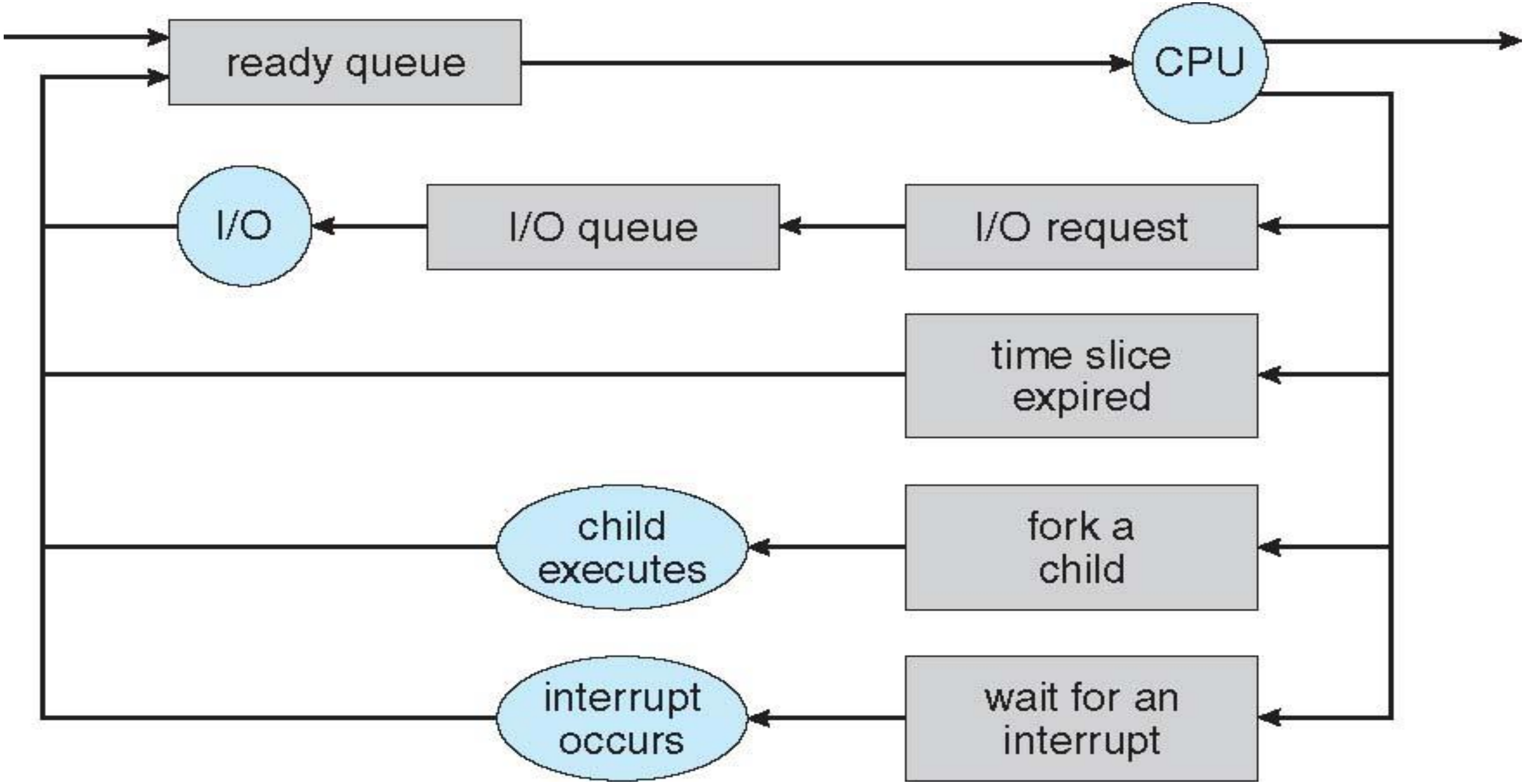
Process scheduling

- **Process scheduler** selects among ready processes for next execution on CPU
- Maintains **scheduling queues** of processes
  - **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



Process scheduling flow



Interprocess Communication

Independent process

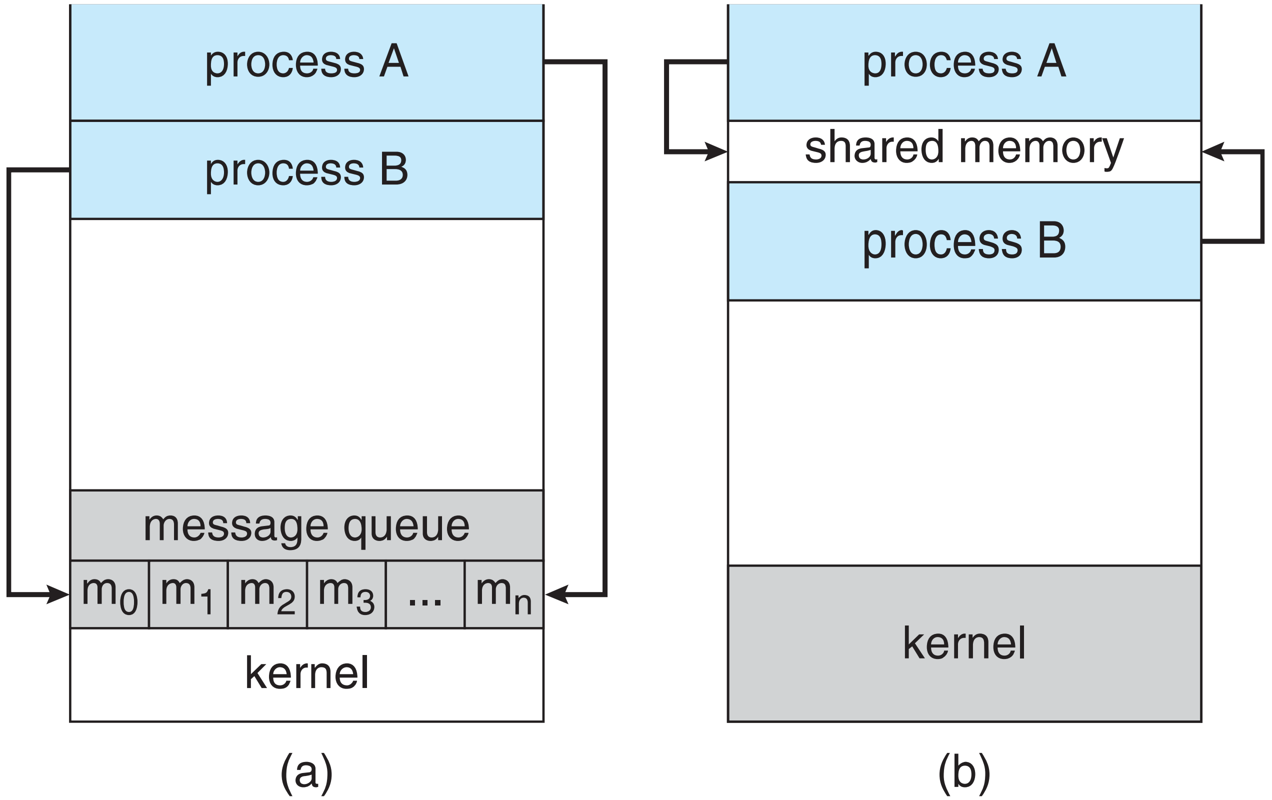
- So far, we have talked about **independent** processes: processes that don’t interact with other processes

- Processes can be designed to **cooperate** - process can affect or be affected by other processes, including sharing data

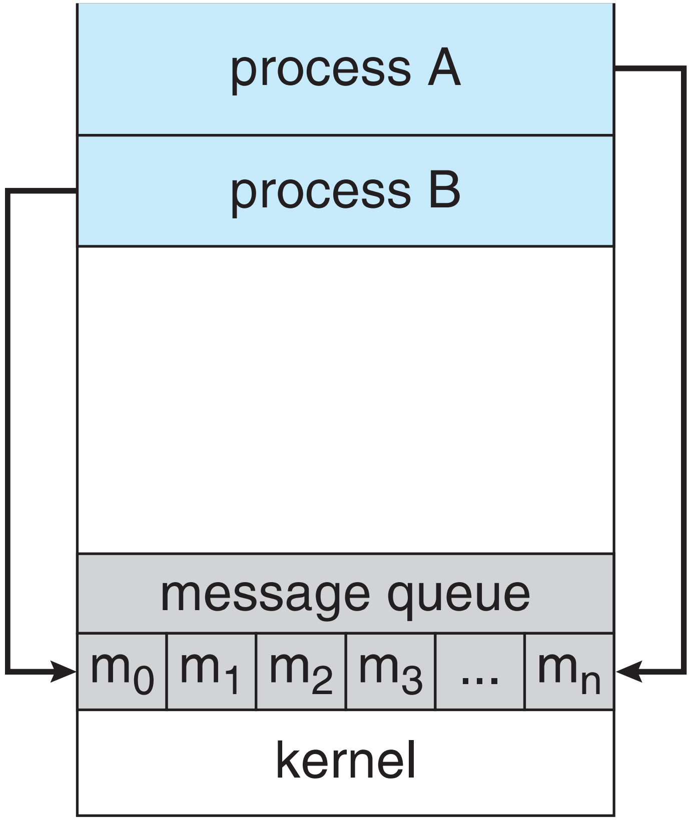
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience

Interprocess communication

- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
  - **Shared memory**
  - **Message passing**



Message passing



- Processes communicate with each other without resorting to shared variables

- IPC facility provides two primitive operations:
  - `send`(*message*)
  - `receive`(*message*)

- If *P* and *Q* wish to communicate, they need to:
  - establish a ***communication link*** between them
  - exchange messages via send/receive

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking / synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking / asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

Synchronization

- Different combinations possible
  - If both send and receive are blocking, we have a **rendezvous**
  - Producer-consumer interaction becomes trivial

- Producer-consumer paradigm: a cooperation paradigm in IPC

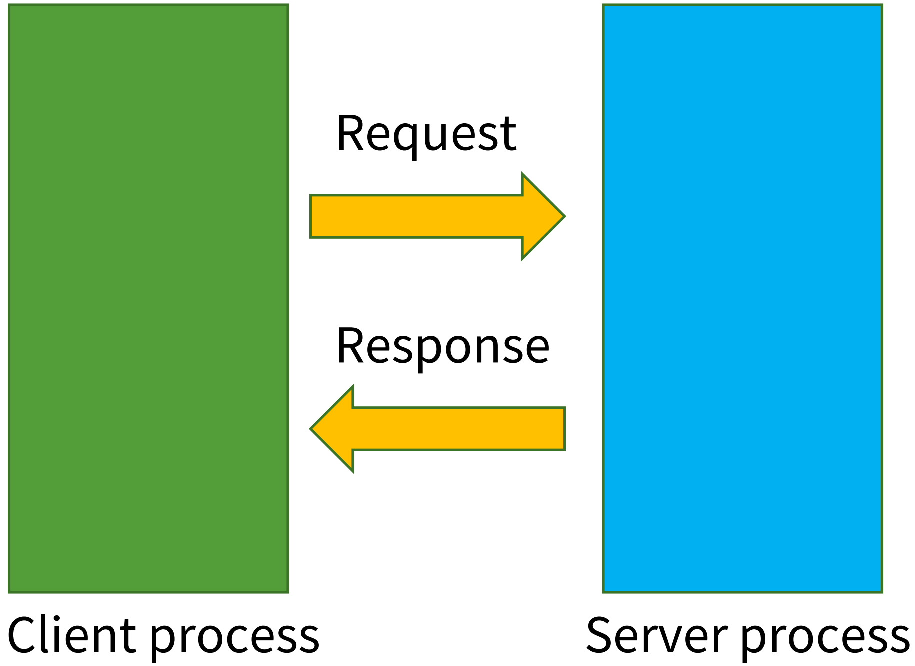
Buffering

- Queue of messages attached to the link

- Implemented in one of three ways:
  - Zero capacity – 0 messages
    Sender must wait for receiver (rendezvous)
  - Bounded capacity – finite length of *n* messages
    Sender must wait if link full
  - Unbounded capacity – infinite length
    Sender never waits

Client-server model

- Based on the producer-consumer model of process cooperation
- Client makes the request for some resource or service to the server process
- Server process handles the request and sends the response (result) back to the client



Client-server model

- Client process needs to know the existence and the address of the server

- However, the server does not need to know the existence or address of the client prior to the connection

- Once a connection is established, both sides can send and receive information

Client-server communication

- Remote Procedure Calls
- Pipes
- Remote Method Invocation (Java)
- **Sockets**

Next lecture

- Socket programming