ECE437/CS481

M02A: PROCESSES & THREADS PROCESSES CONCEPTS

Chapter 3.1-3.2

Xiang Sun

The University of New Mexico

Programs & Processes

❑ What is a program?

- "A program contains codes & static data stored in a file" (von Neumann architecture)
  - ✓ Source file/code – before compilation
  - ✓ Binary file/code – ready to be loaded

❑ What is a process?

- "A process is a program that is being executed"
  - ✓ A program can execute many times, so corresponding to many processes
  - ✓ Different process have their own address spaces & process contexts

Process Context

❑ Process context includes

- ➢ Execution information
  - ✓ CPU register set image (e.g., Stack Pointer (SP), Instruction Pointer (IP), etc.)
  - ✓ Process stack - containing temporary data (e.g., subroutine arguments, temporary variables, return addresses)

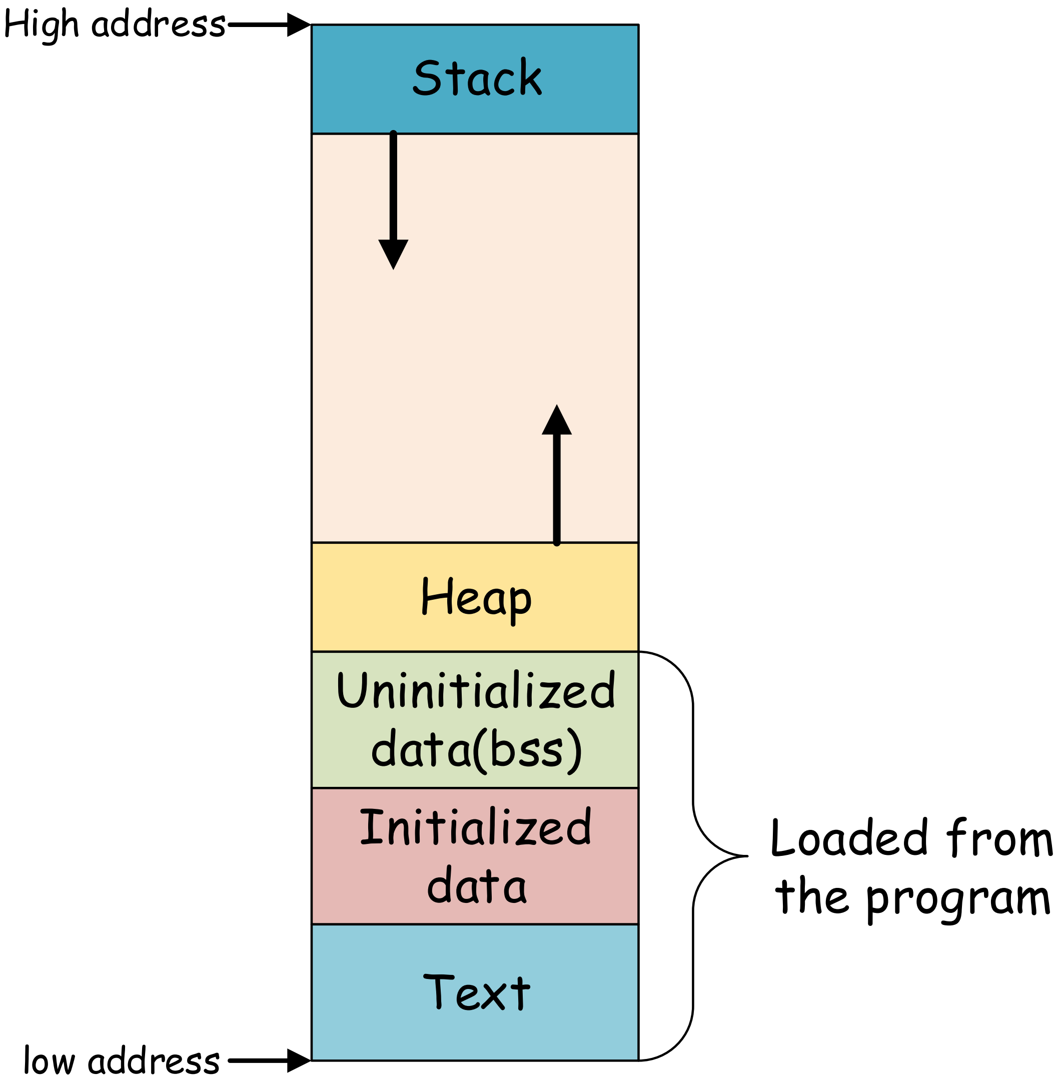
`%rax`	Return value	`%r8`	Argument #5
`%rbx`	Callee saved	`%r9`	Argument #6
`%rcx`	Argument #4	`%r10`	Caller saved
`%rdx`	Argument #3	`%r11`	Caller Saved
`%rsi`	Argument #2	`%r12`	Callee saved
`%rdi`	Argument #1	`%r13`	Callee saved
`%rsp`	Stack pointer	`%r14`	Callee saved
`%rbp`	Callee saved	`%r15`	Callee saved

Process Context

- Process context includes
  - Environment information
    - ✓ Memory address space or memory map, i.e., regions of memory that have been allocated to the process
    - ✓ Resources, e.g., open file table
  - System attributes
    - ✓ process identification or Process ID (PID)
    - ✓ process state information, e.g., waiting, ready, etc.
    - ✓ process control information, e.g., priority

Process Memory Map

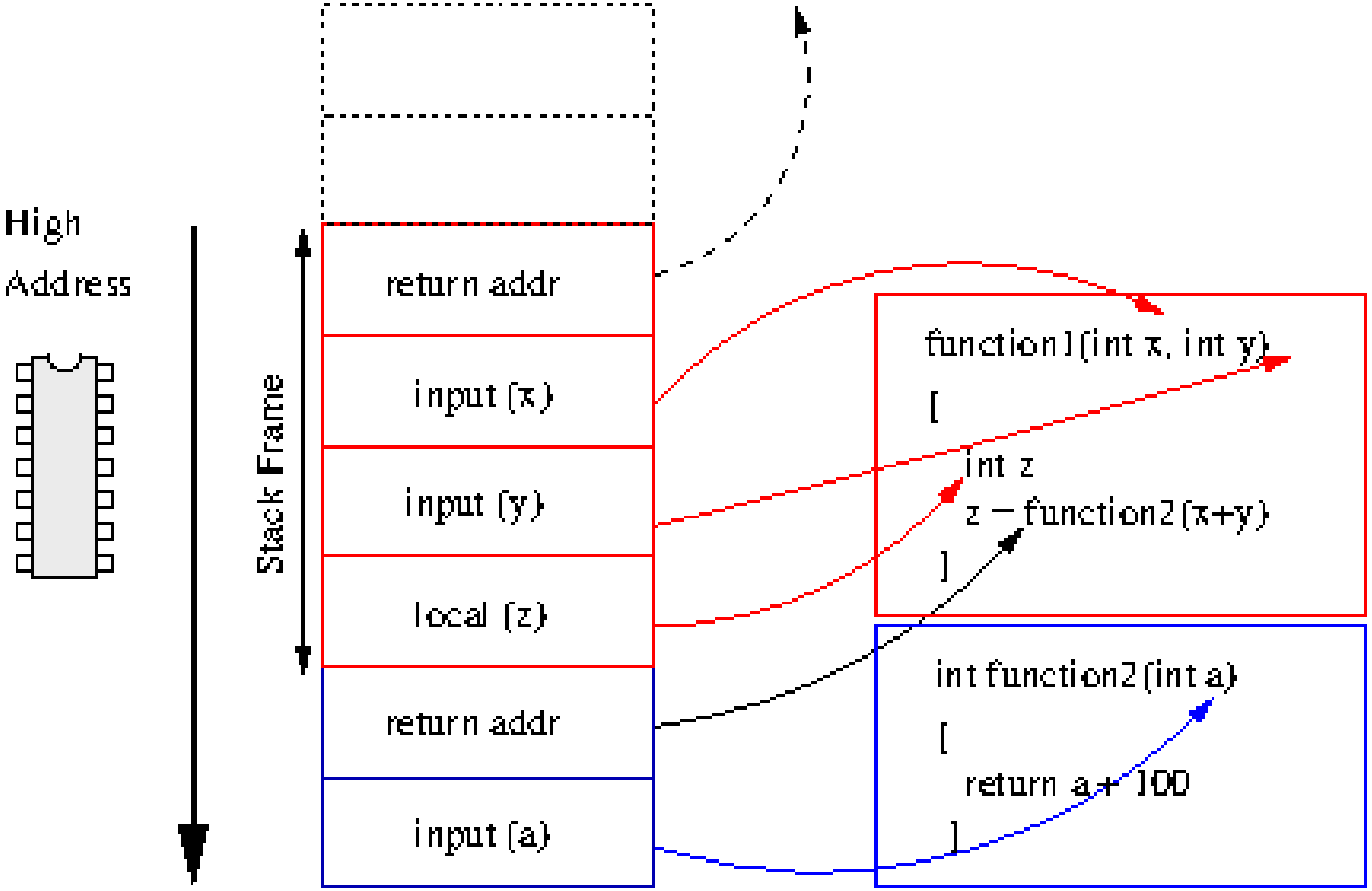
- Process's **memory layout** in Unix/Linux
  - **Text (code)**: the machine instructions to be executed. Usually, it stays unmodified over the lifecycle of the process.
  - **Initialized data**
  - **Uninitialized data**
  - **Heap**: dynamically allocated memory
    - ✓ grow via a system call to request more memory
  - **Stack:** memory space to manage function calls, returns, parameter passing, and local variables.
    - ✓ grow and shrink as the depth of function calls varies



Process Memory Map

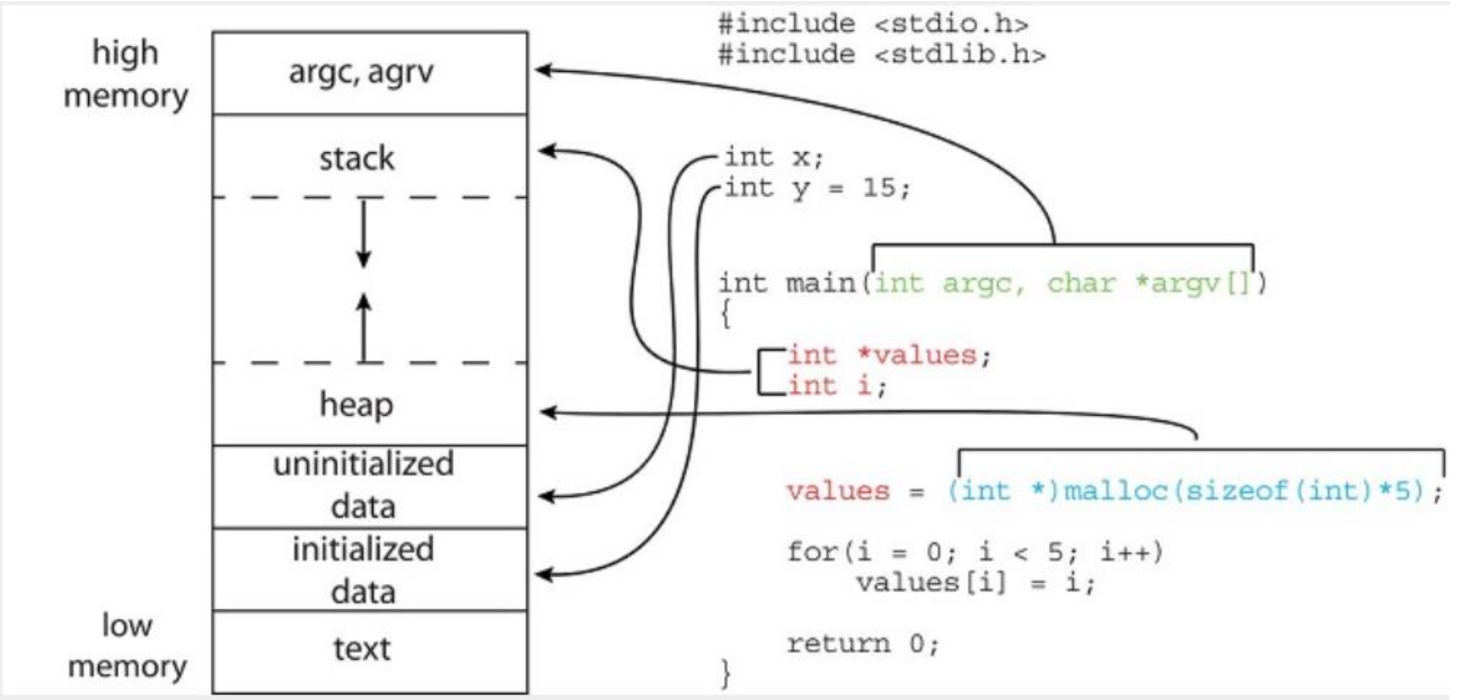
❑ Process's memory layout in Unix/Linux

- ➢ Once a subfunction is invoked, a new stack frame is created.
- ➢ The stack frame is an area of memory which usually contains the return address, input arguments to the subfunction, and local variables.
- ➢ The stack starts at a high address in memory and progressively gets lower.



Process Memory Map

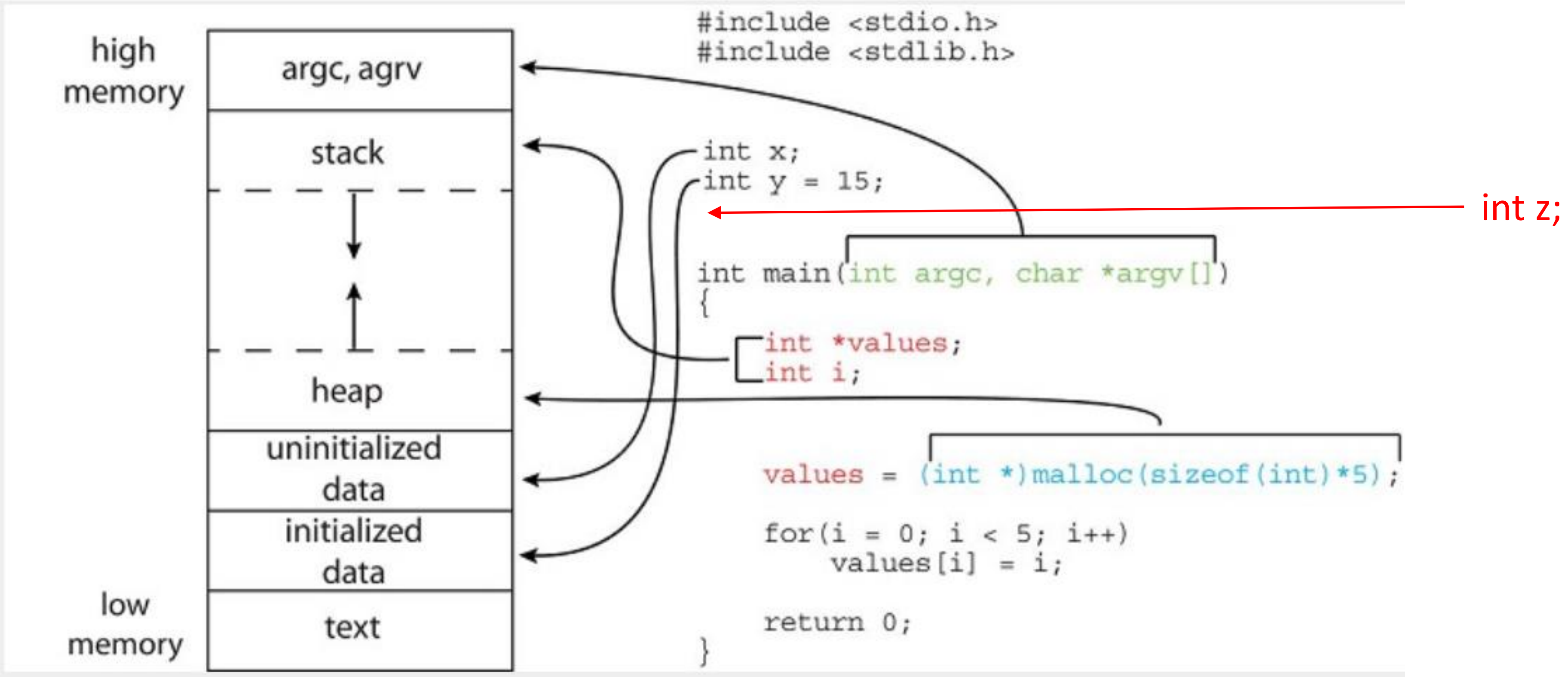
- An example of memory layout of a C program
  - The global data section is divided into different sections for (a) initialized data and (b) uninitialized data.
  - The arguments and local variables in the main() function are stored in the stack.
  - Malloc() create a memory space for heap.
  - Run *size <fileName>* can see size (in bytes) of different parts.



text	data	bss	dec	hex	filename
1158	284	8	1450	5aa	memory

Process Memory Map

❑ An example of memory layout of a C program



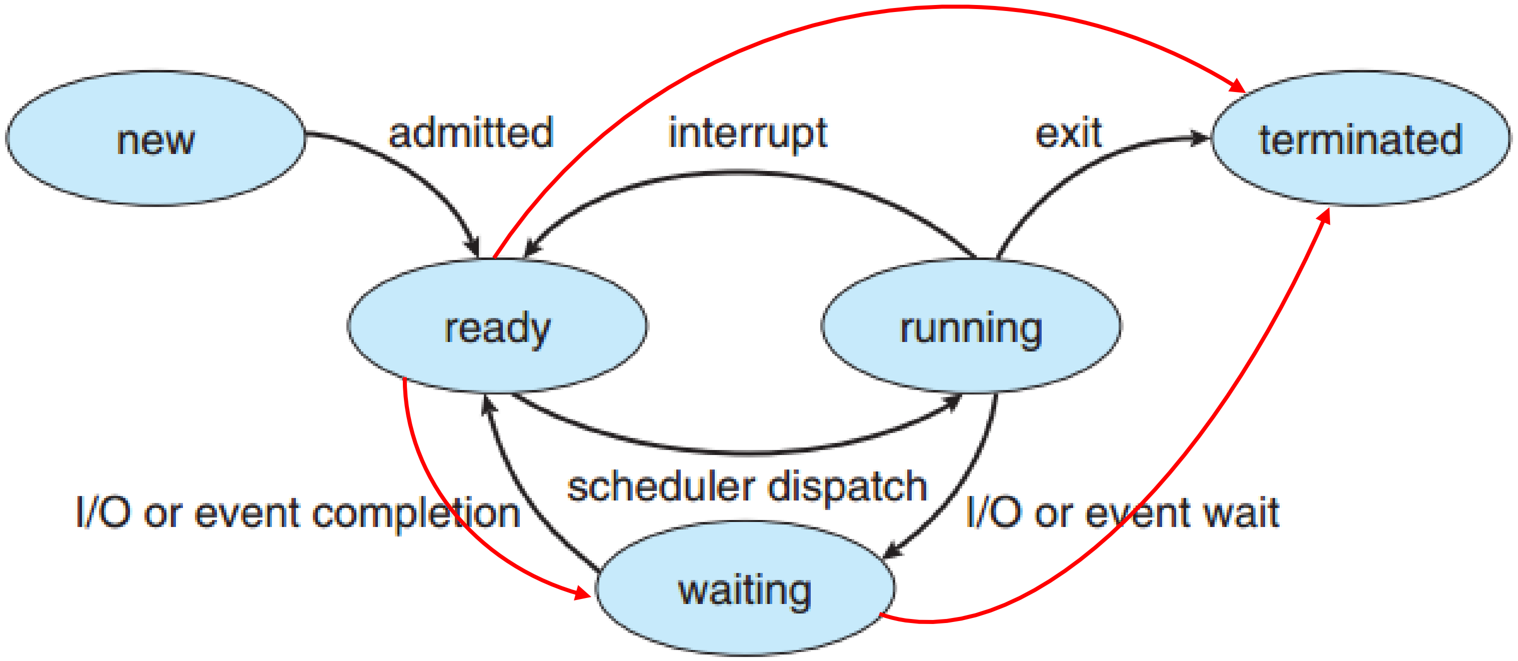
text	data	bss	dec	hex	filename
1158	284	8	1450	5aa	memory

Process State

- **New** (Start)
  - The process is being created but has not yet admitted to the pool of executable processes
    - Admit --- go to Ready state: The OS will move a process to the Ready state when the system has enough memory space.
- **Ready**
  - The process is waiting to be assigned to a processor.
- **Terminated** (Exit):
  - The process has finished the execution.
- **Running**
  - The process is currently being executed.
  - In a single processor computer, at most one process can be in this state. But it is not true for today's multicore systems.
- **Waiting** (Blocked)
  - The process is waiting for some event to occur.

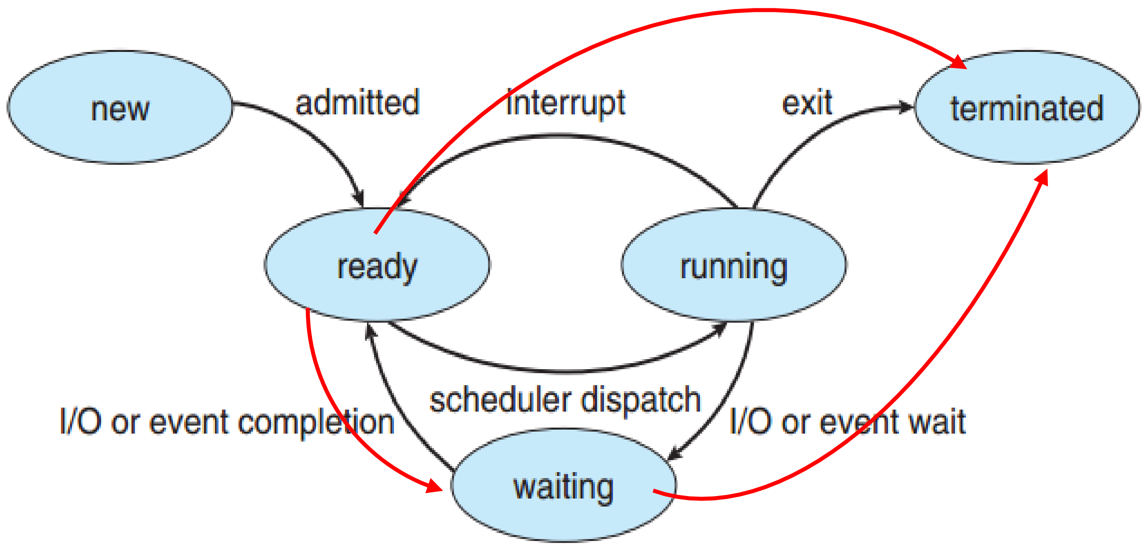
Process State

- ❑ Ready→Running/Waiting/Terminated
  - ➢ Scheduler dispatch --- go to the Running state
  - ➢ Suspend --- go to the Waiting state
  - ➢ Kill --- go to the Terminated state



Process State

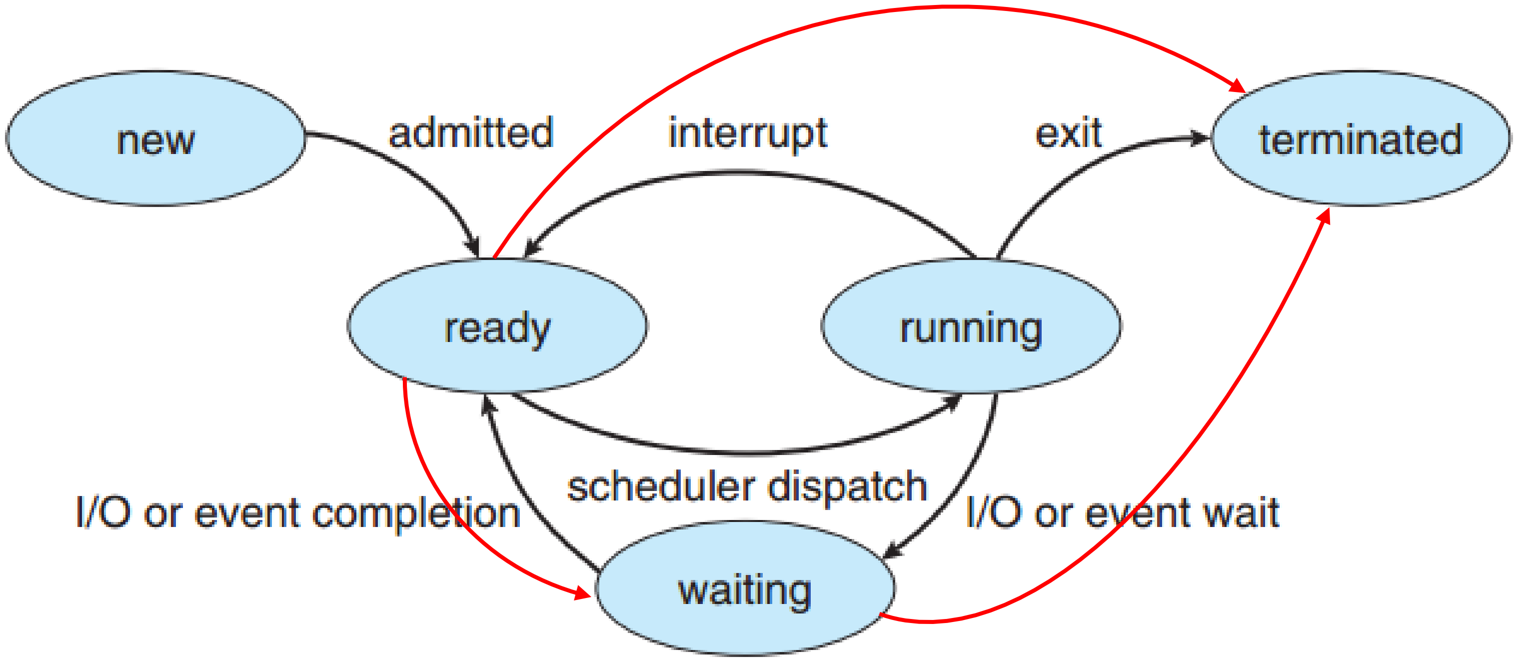
- Running→Ready/Waiting/Terminated
  - Yield (i.e., the running process voluntarily release control of the processor) --- go to the Ready state
  - Time slice is up (i.e., the running process has reached the maximum allowable time) --- go to the Ready state
  - Arrival of high priority process (i.e., the running process will be preempted due to its lower priority) --- go to the Ready state
  - I/O request --- go to the Waiting state
  - Suspend --- go to the Waiting state
  - Terminate --- go to the Terminated state



Process State

❑ Waiting→Ready/Terminated

- ➢ I/O complete, wakeup --- go to the Ready state
- ➢ Kill --- go to the terminated state



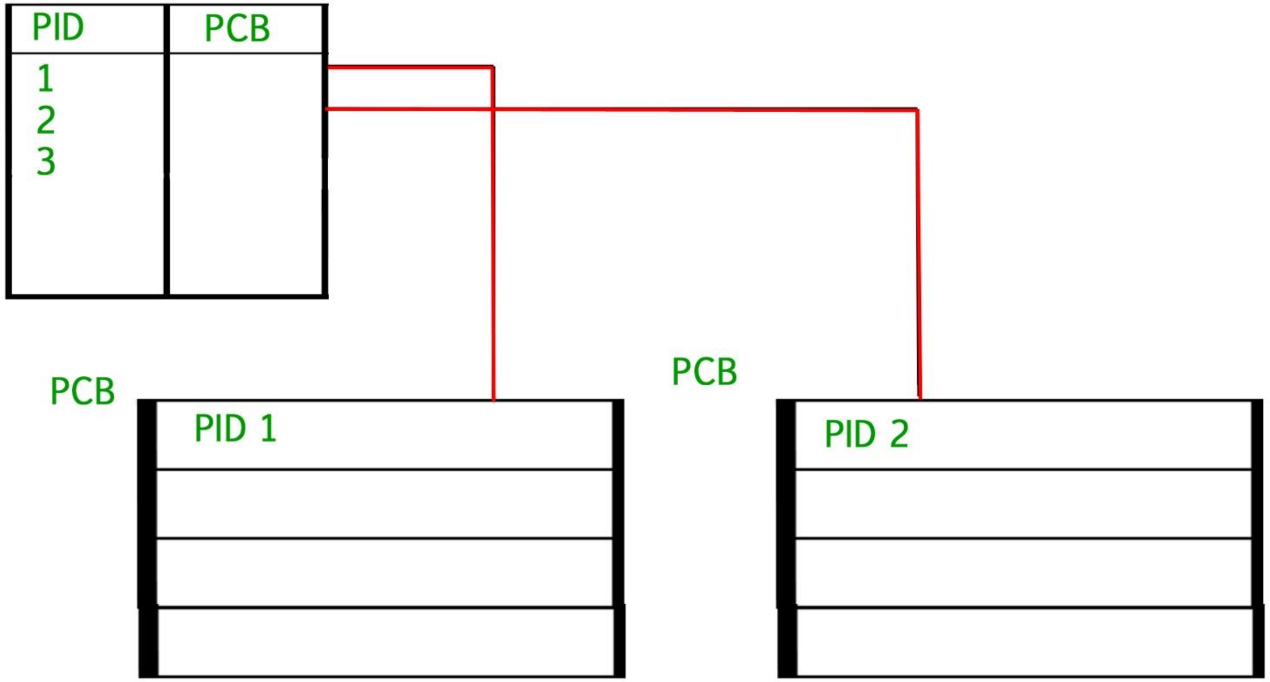
Process State

Process Control Block (PCB)

- Each process is represented in the OS by a PCB
- An important data structure to keep the context of a process
  - PID: a unique integer as process ID
  - UID: user ID who is executing this process
  - Program Counter: A pointer to the address of the next instruction to be executed for this process.
  - Process state: the current process state
  - Event: any event for which the process may be waiting
  - Memory management info: information of page table, memory limits, segment table, etc.
  - CPU registers: values of CPU registers where a process is swapped out.
  - Scheduling priority : parameters for the process scheduling
  - Accounting info: timing, usage, ...
  - IO status information: a list of I/O devices allocated to the process.
- In linux:
  - defined in task_struct (/usr/src/linux/include/linux/sched.h)---over 95 fields!!!

❑ Process table/list

- The OS maintains a process table/list, which contains PIDs and pointers.



Process table and process control block

State Queues

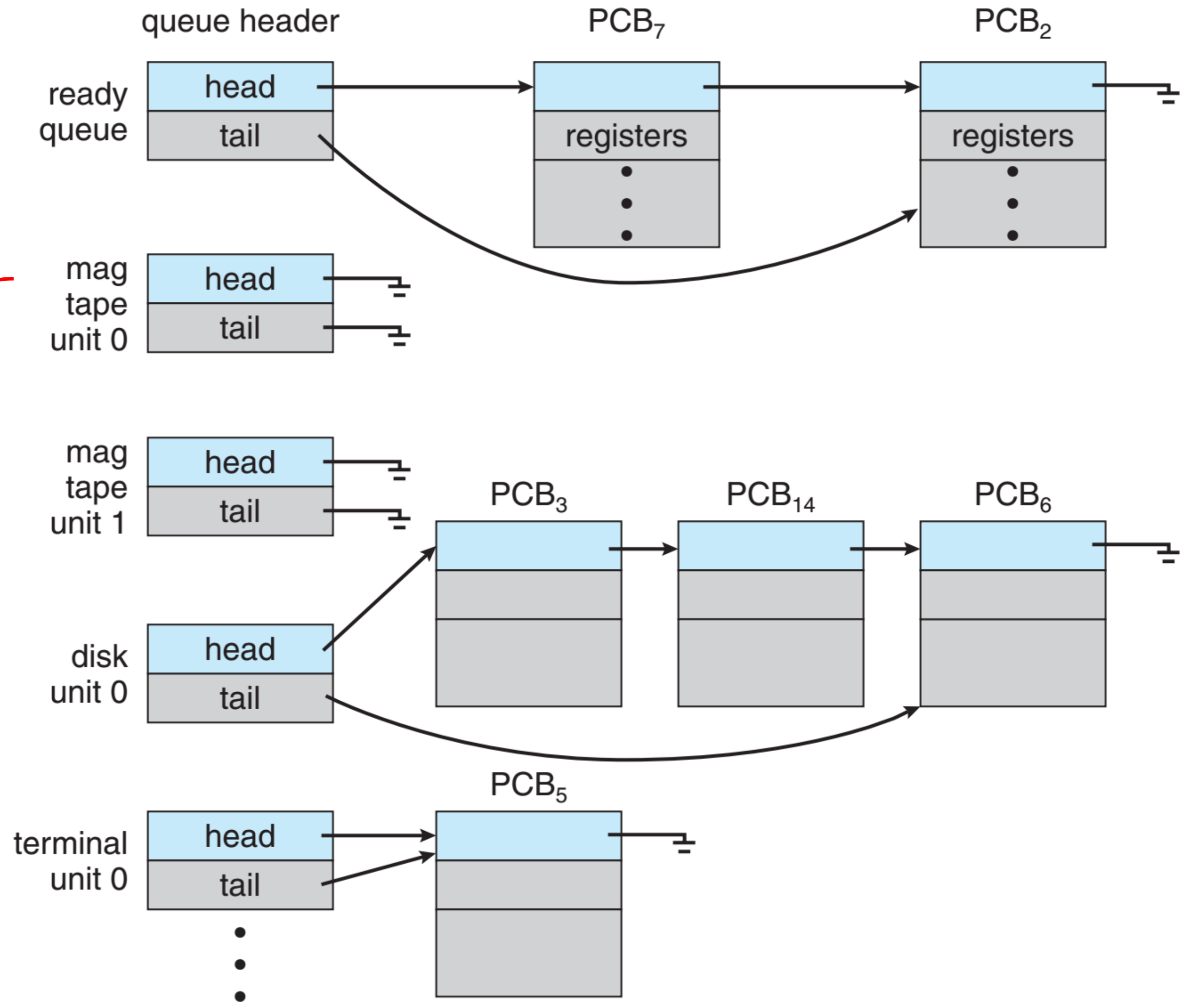
- The OS maintains a collection of queues that represent the state of all processes in the system
  - Typically, one queue stands for a state, e.g., ready,
  - Each PCB is queued onto a specific state queue based on its status.
  - Once a process changes its state, its PCB is unlinked from one queue, and linked onto another.

State Queues

- ➢ There may be many waiting queues, one for each type of wait (particular device, timer, message, …)
- ➢ Running queue? Terminated queue?

Ready queue

Waiting queue



Process Context Switch

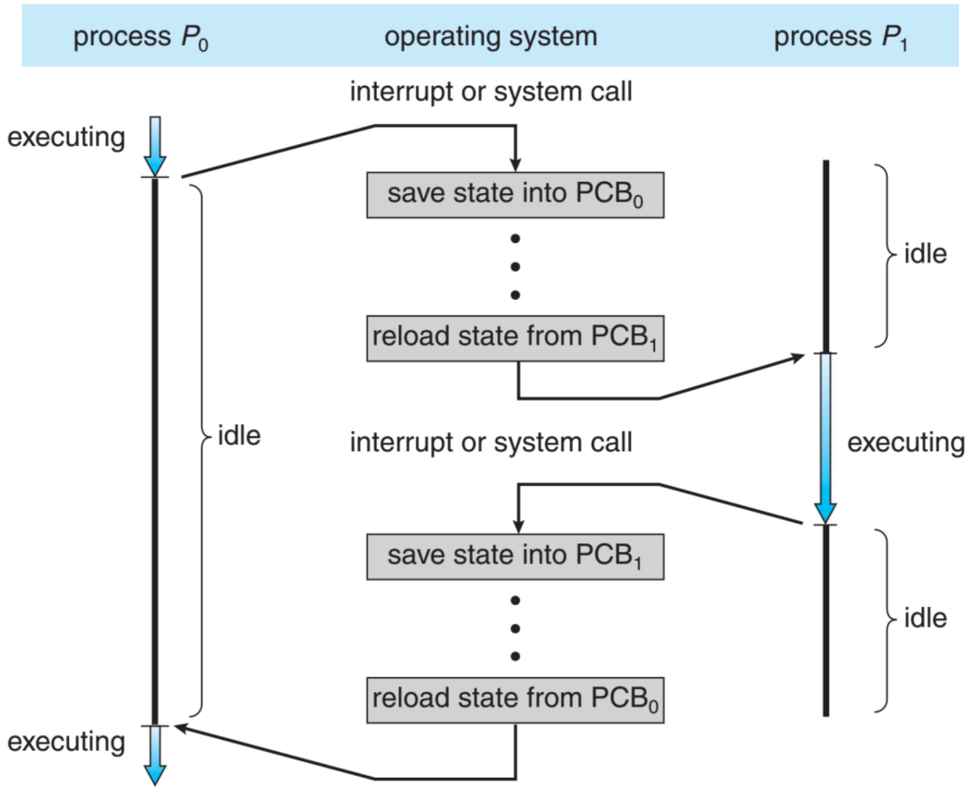
❑ Process context switch

---Switching of the CPU from process A to another, requiring storing the state of process A such that it can be restored, and execution resumed from the same point later.

- Why need context switch

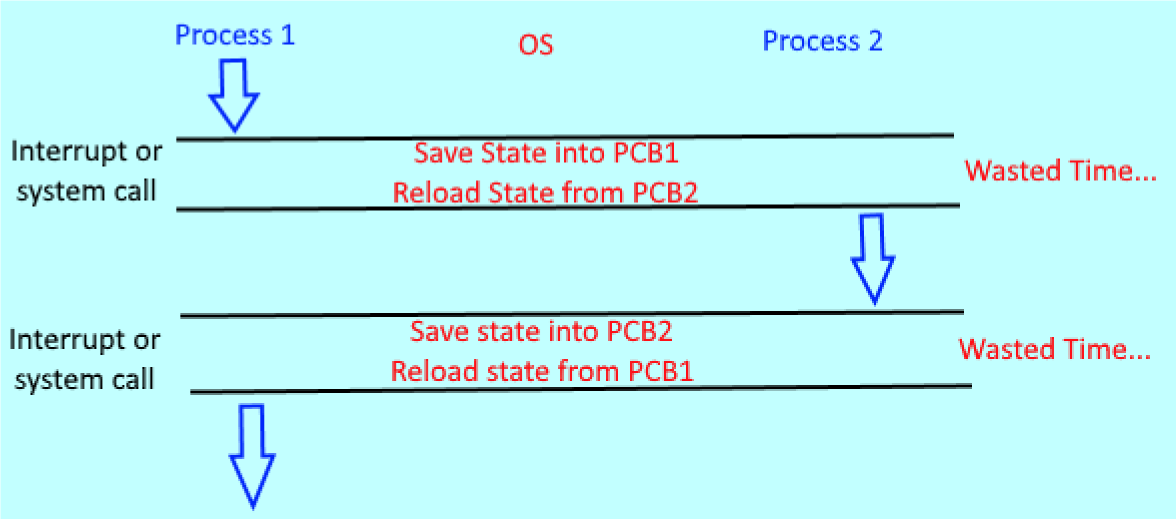
--- improve CPU utilization

- Steps of context switch
  1. save the registers' data of the current process.
  2. change process state (e.g., running->waiting) and put it to corresponding queue.
  3. select another process from the ready queue.
  4. change the newly selected process' state (e.g., ready->running) and reload the registers' data from its PCB.
  5. (re)start computation of the new process.



Process Context Switch

- **Overhead:**
  - The time incurred by a context switch. The CPU does no useful work while switching.
  - Translation lookaside buffer (TLB) flushing.



- **Tradeoff:**
  - Context switch can improve the CPU utilization; however, context switch may incur overheads, which may reduce CPU utilization.

Process Context Switch

- When to do context switch
  - I/O system call
  - Time slice is up (timer interrupt)
  - Arrival of higher priority process
  - Yield by process itself (yield())
  - Termination of process

- Difference between context switch and mode switch?
  - The CPU is executing the same process when mode switch occurs.
  - The CPU is executing a different process when context switch occurs.

- Not all the system calls trigger context switch.