Universidade Estadual de Campinas
Instituto de Computação
MC504 Sistemas Operacionais

T01

CPU Virtualization
The Process Abstraction

Arthur João Catto, PhD

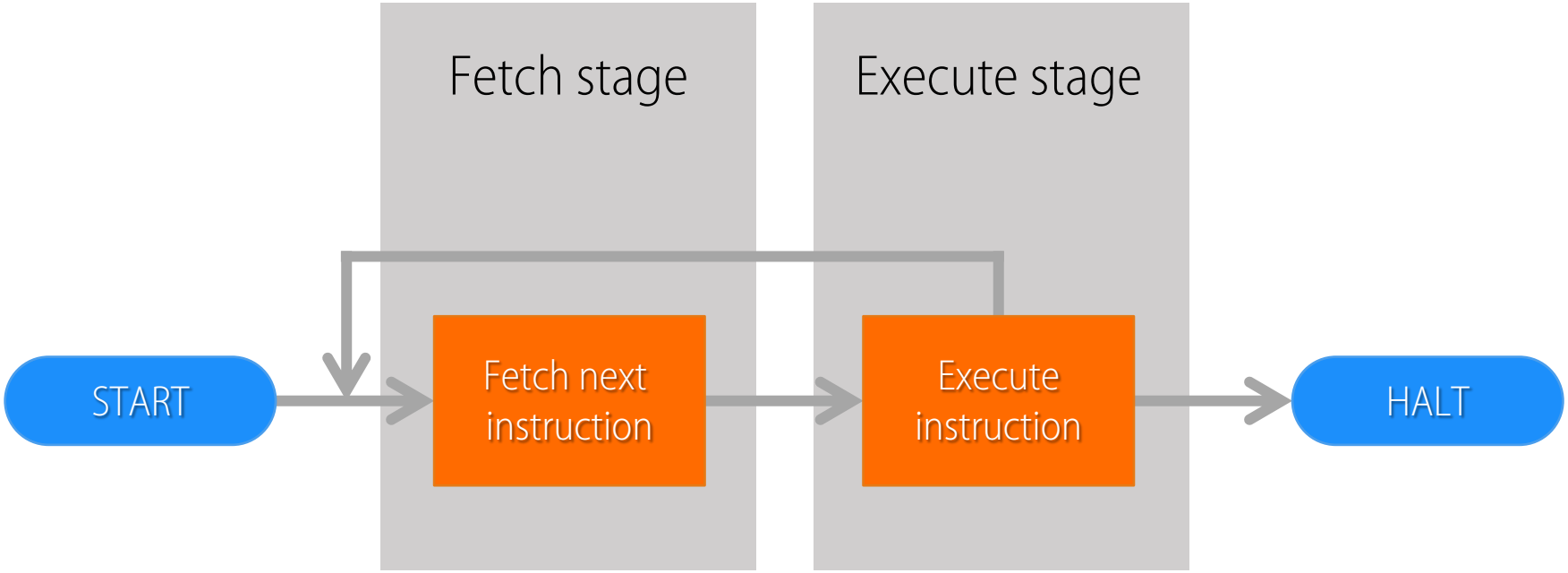
2º semestre de 2018

Essential Question

- How can the OS provide the illusion of a nearly endless supply of physical CPUs when there are only a few of them available?

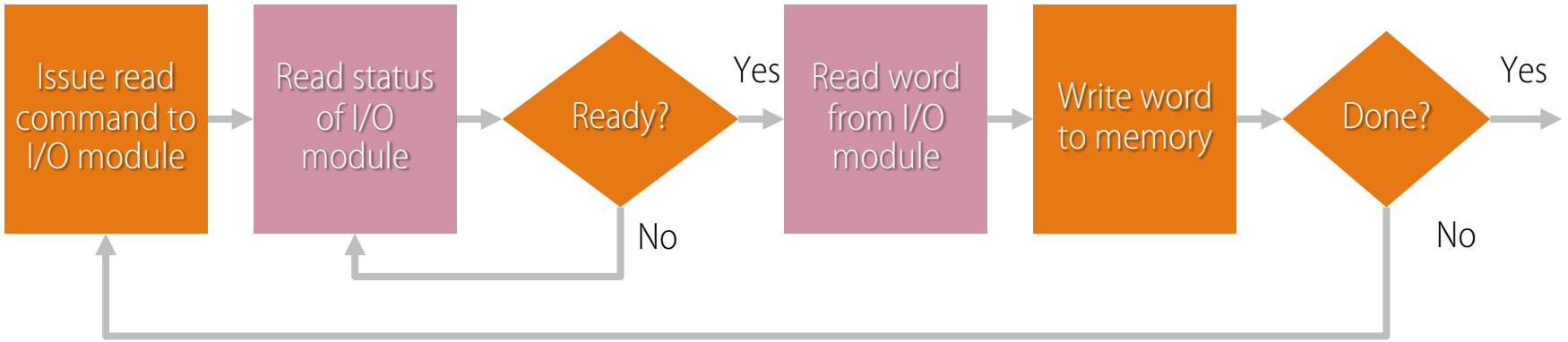
Introduction

The operation of a simple computer



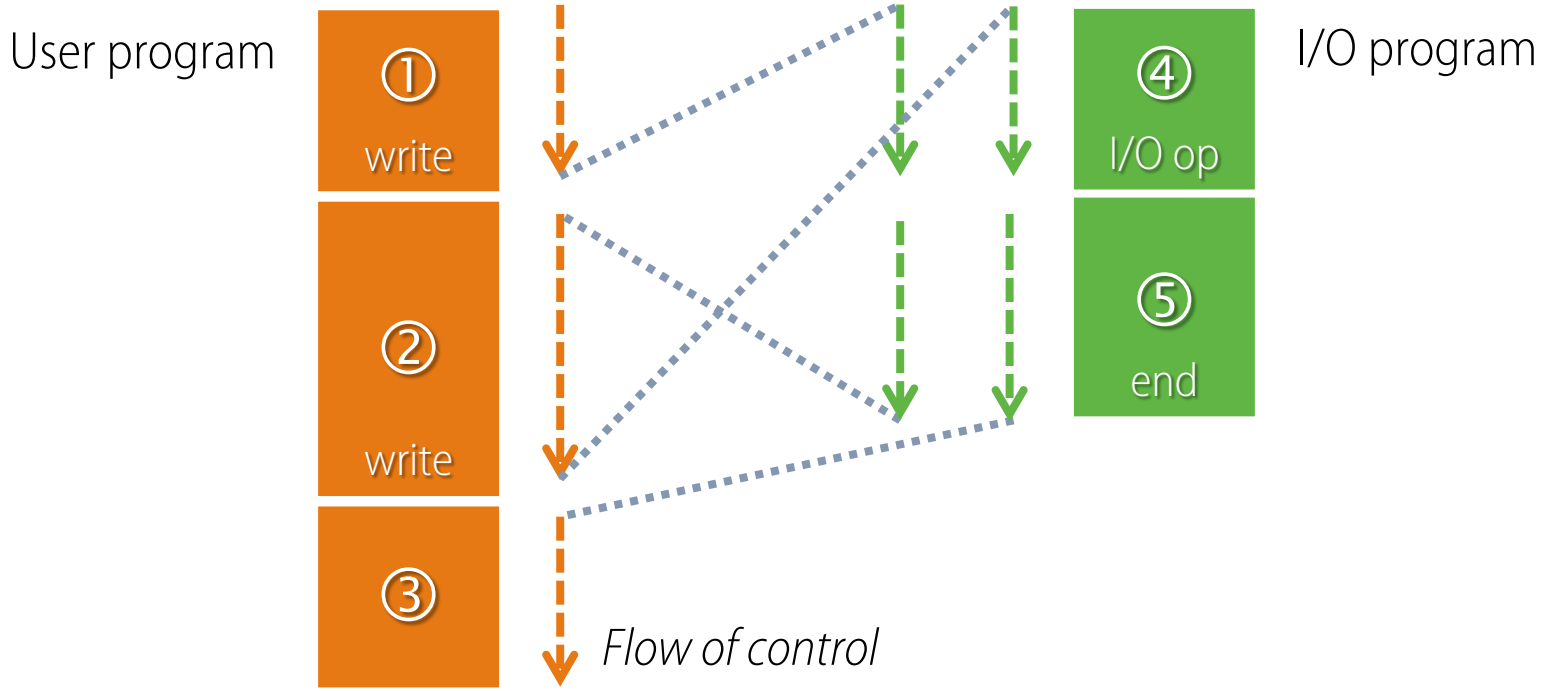
Simple computer I/O

Programmed I/O



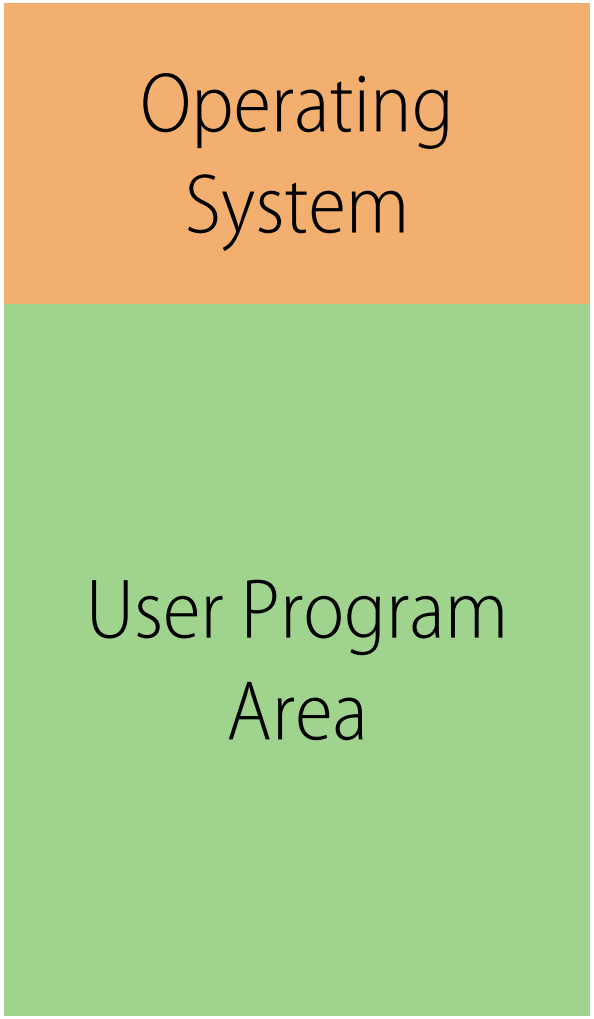
- I/O module performs the action, not the processor
- Sets appropriate bits in the I/O status register
- Processor checks status until operation is complete before proceeding to the next instruction

Flow of Control in Programmed I/O



Flow of time

Memory scheme in a simple batch system

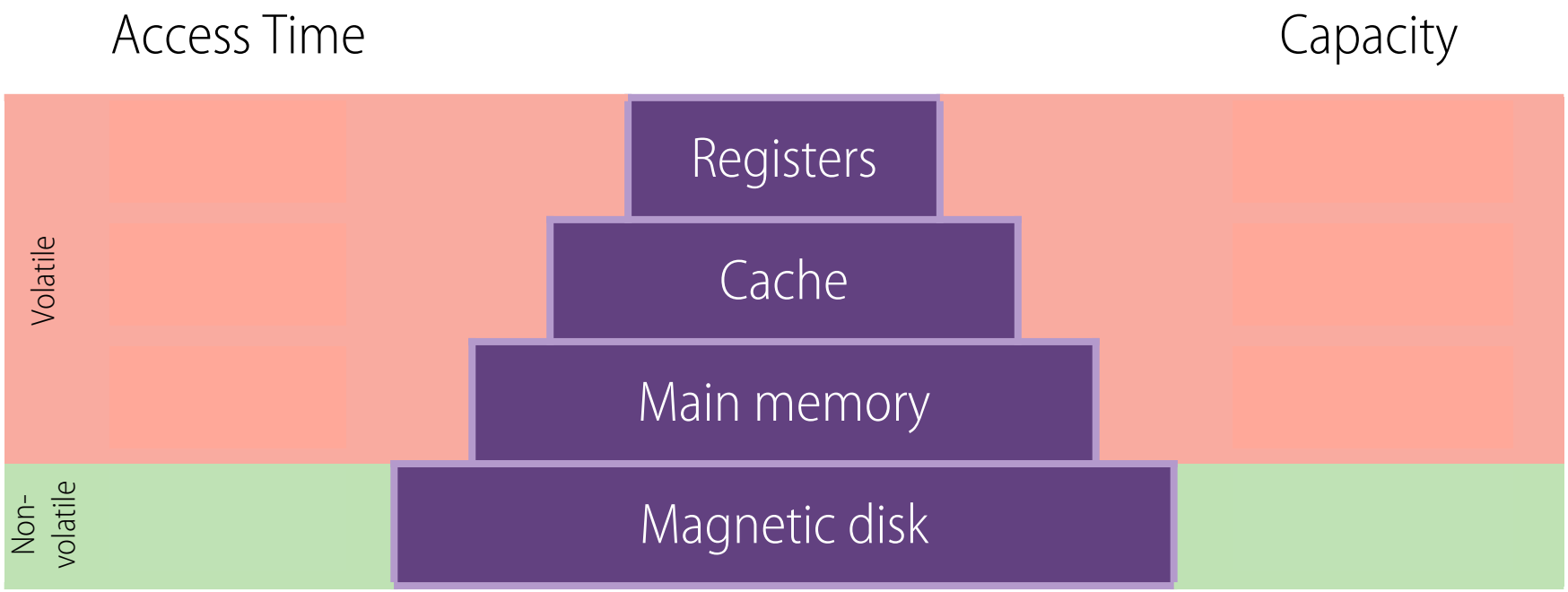


Uniprogramming

- Processor must wait for I/O instruction to complete before proceeding

What is the problem with waiting?

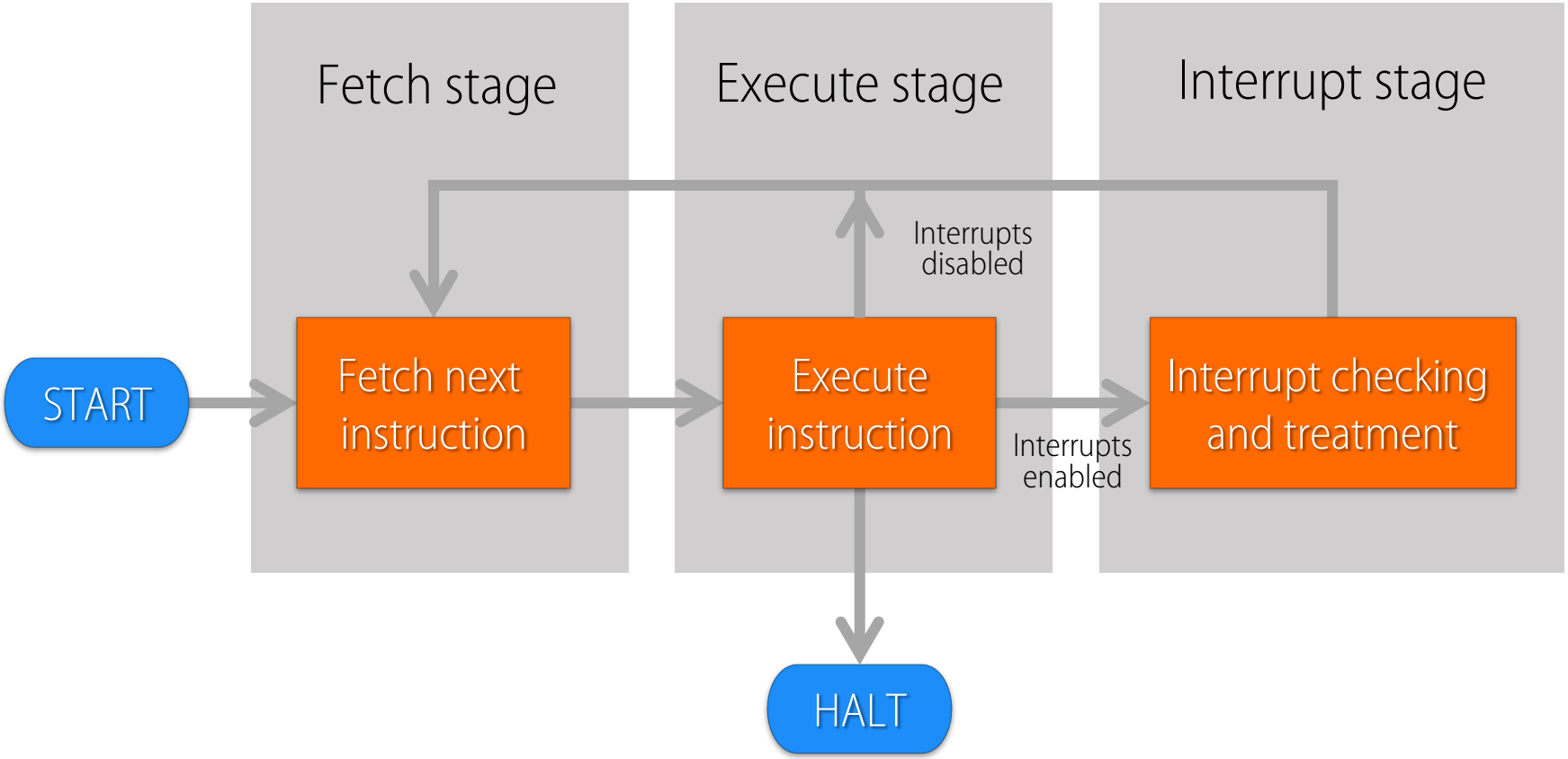
I/O devices are too slow



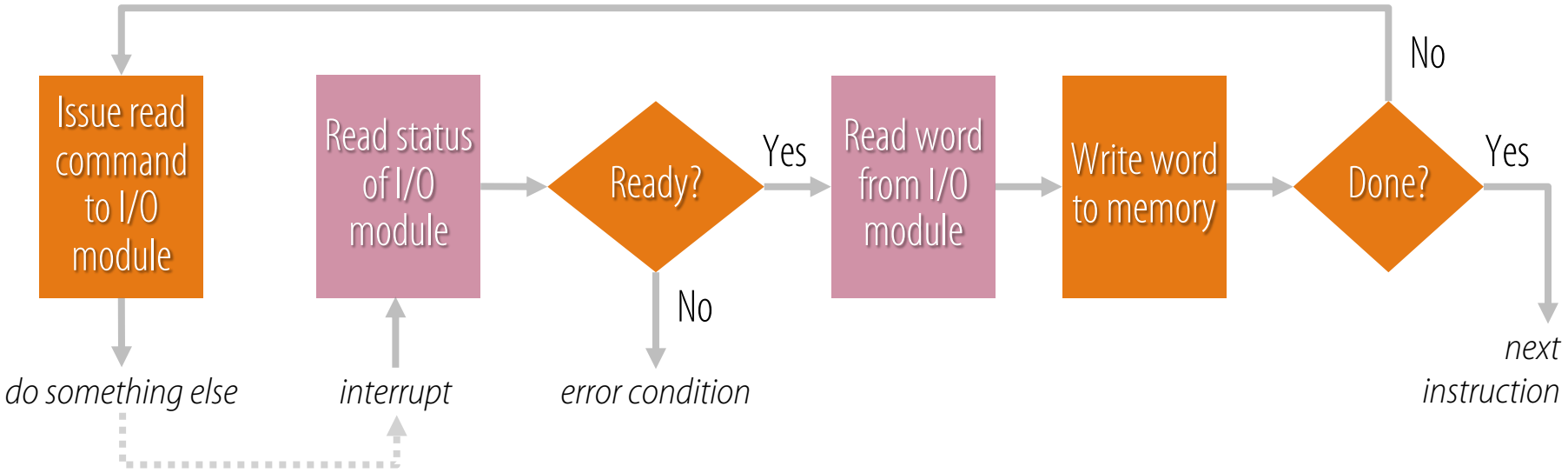
- The numbers are very rough approximations as of today.

Basic instruction cycle with interrupts

- To enable the use of idle CPU time during I/O, the I/O module signals the end of the operation by means of an **interrupt**.

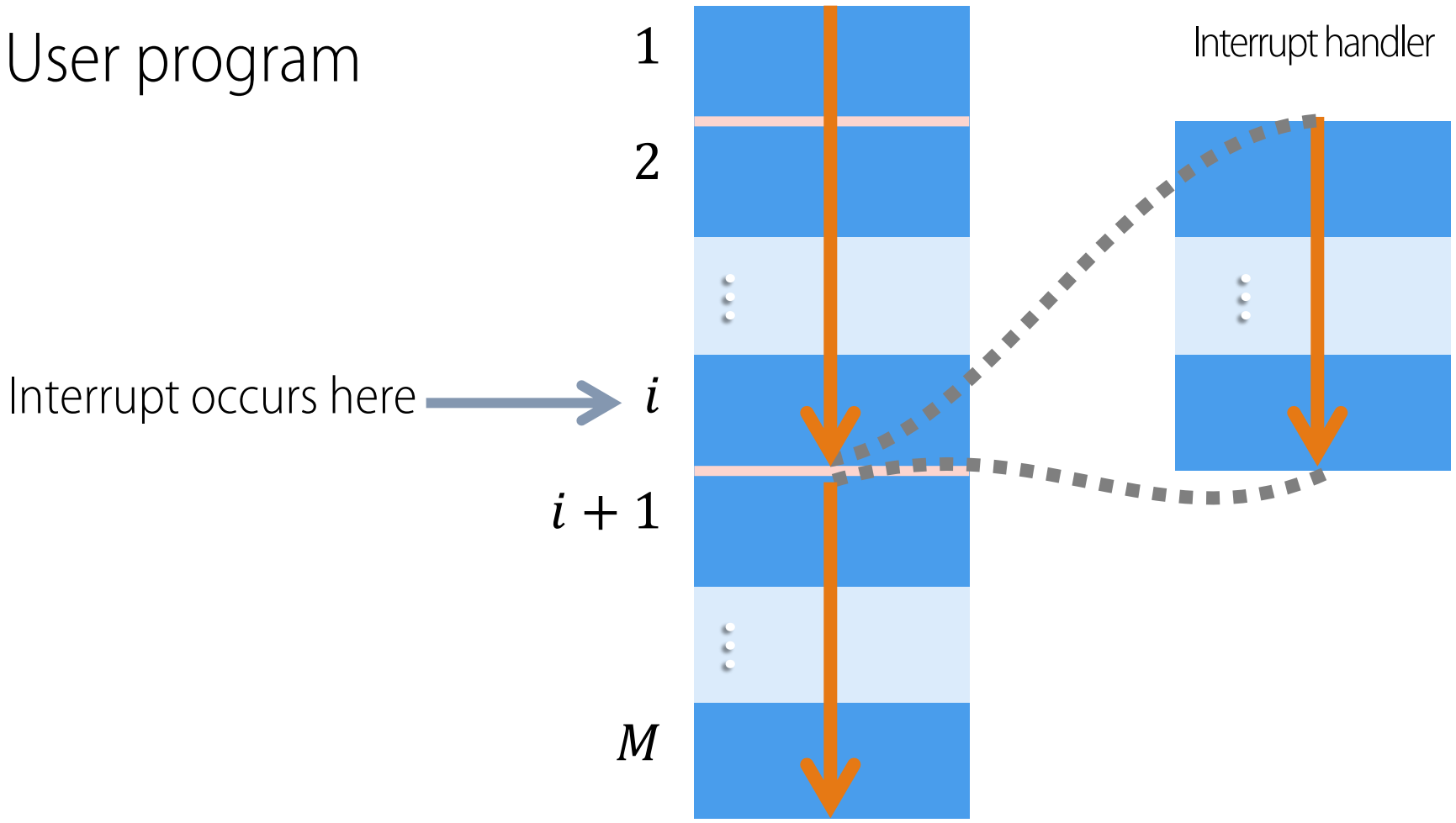


Interrupt-driven I/O or how to harness slack time

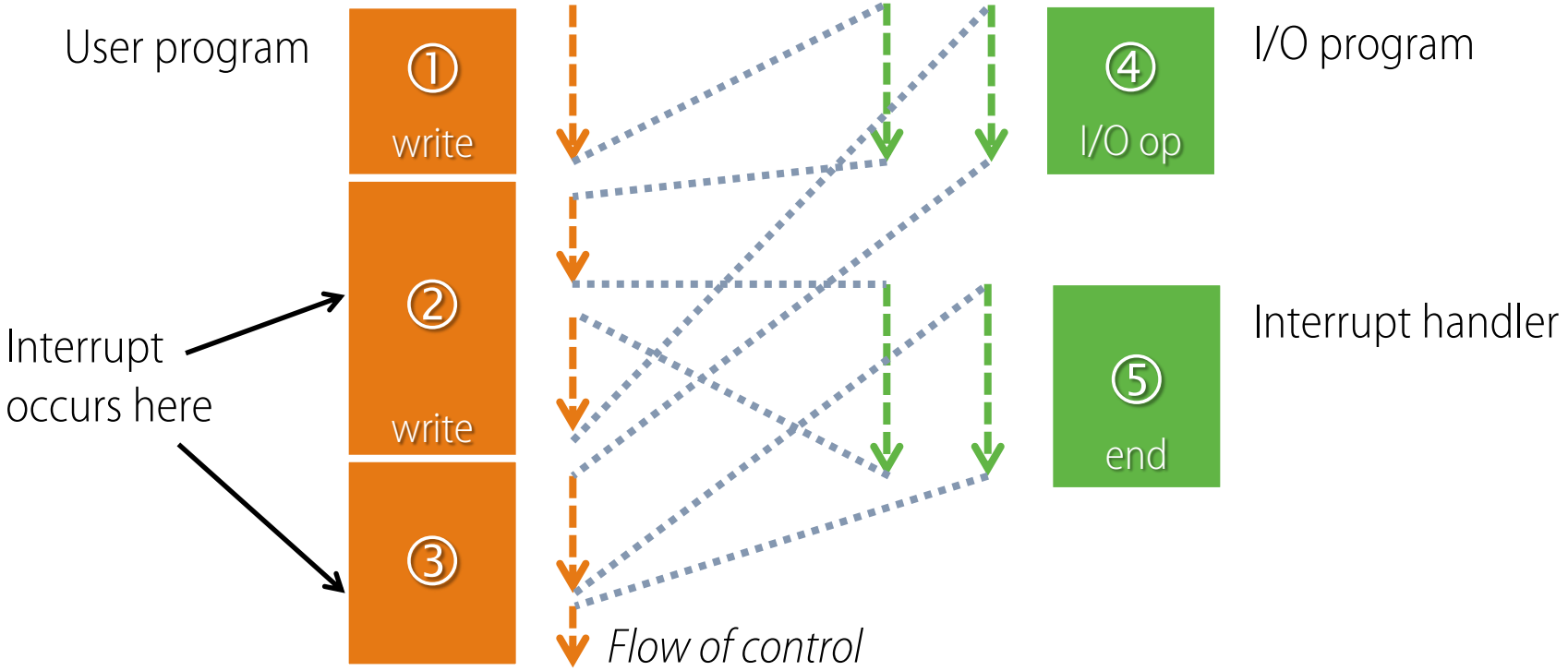


- Processor is interrupted when I/O module ready to exchange data
- Processor saves context of program executing and begins executing interrupt-handler
- No needless waiting
- Consumes a lot of processor time because every word read or written passes through the processor

Transfer of control under an interrupt

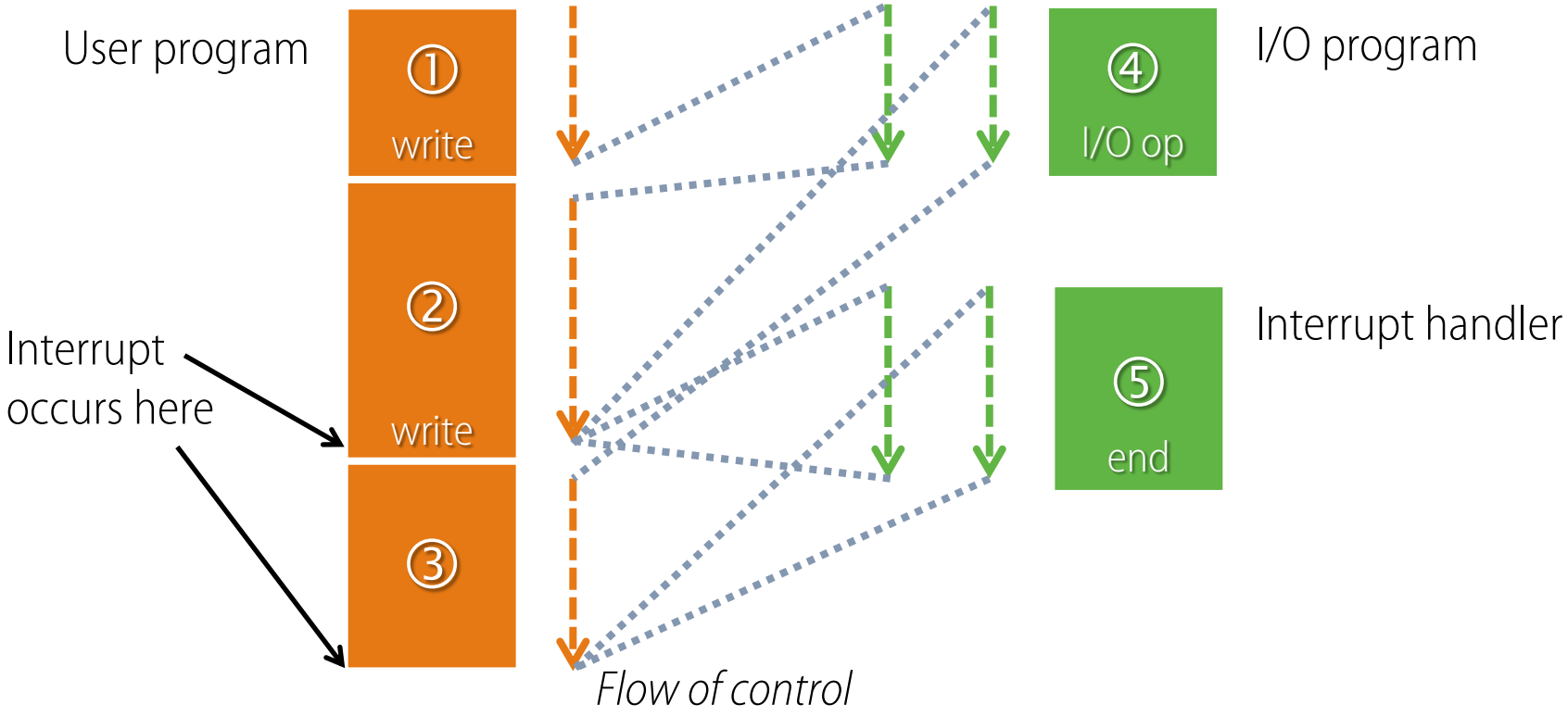


Flow of Control with Short I/O Wait

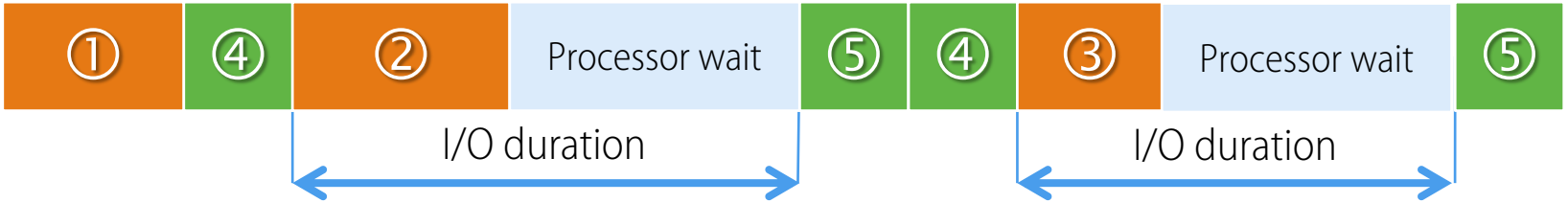


Flow of time

Flow of Control with Long I/O Wait

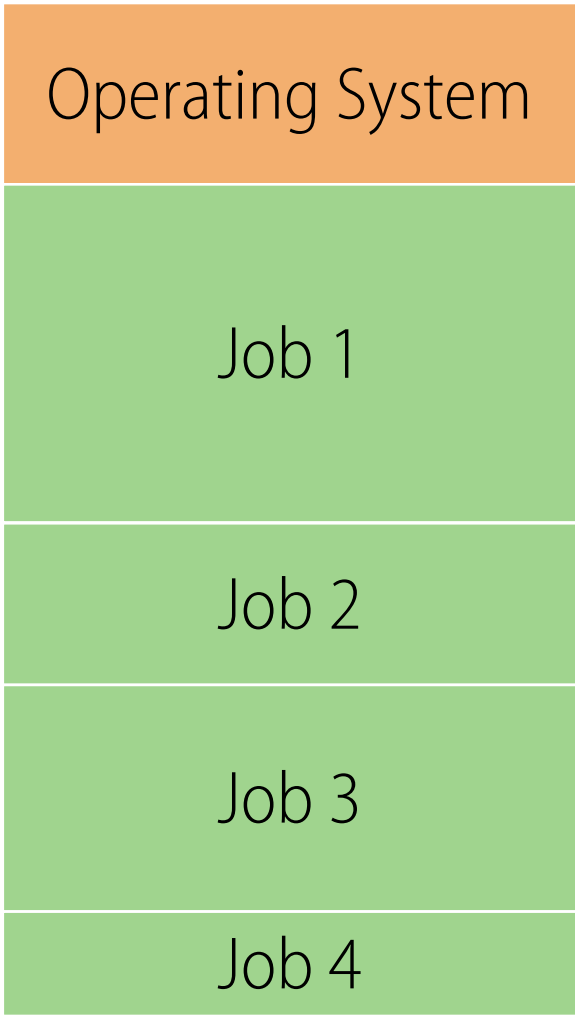


Flow of time



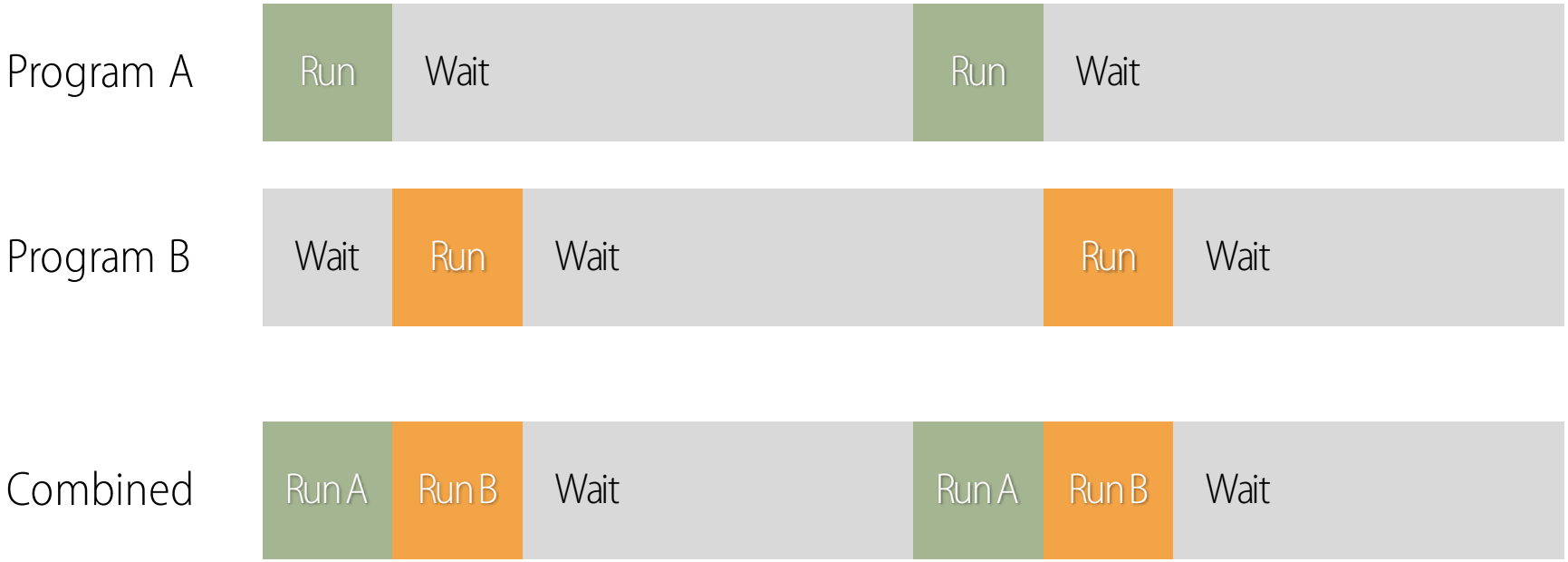
Multiprogrammed batch systems

- Single user cannot keep CPU and I/O devices busy at all times
- Multiprogramming organizes jobs (code and data) so CPU always has one to execute
- A subset of total jobs in system is kept in memory
- One job selected and run via job scheduling
- When it has to wait (for I/O for example), OS switches to another job



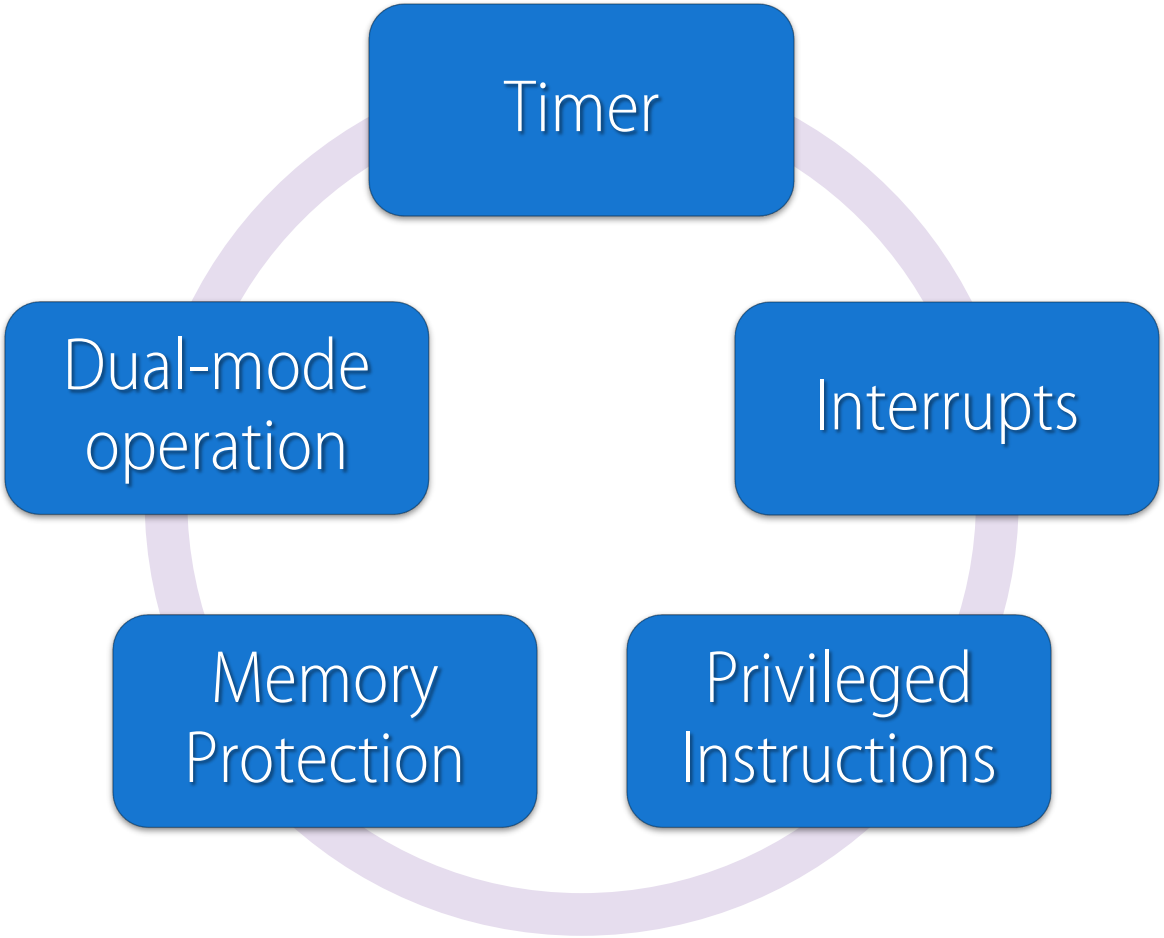
Multiprogramming two jobs

- When one job needs to wait for I/O, the processor can switch to another job

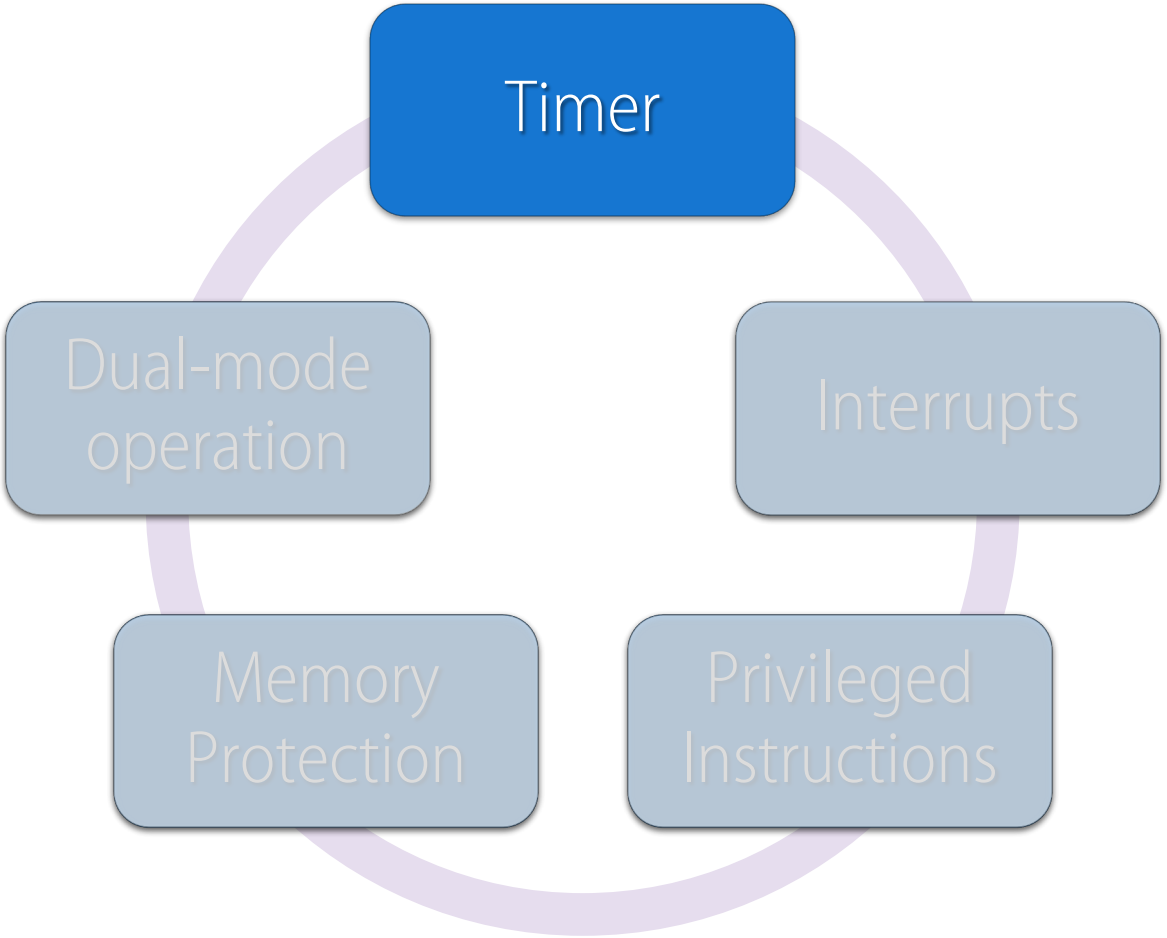


Multiprogramming three jobs

What is required in hw for multiprogramming?

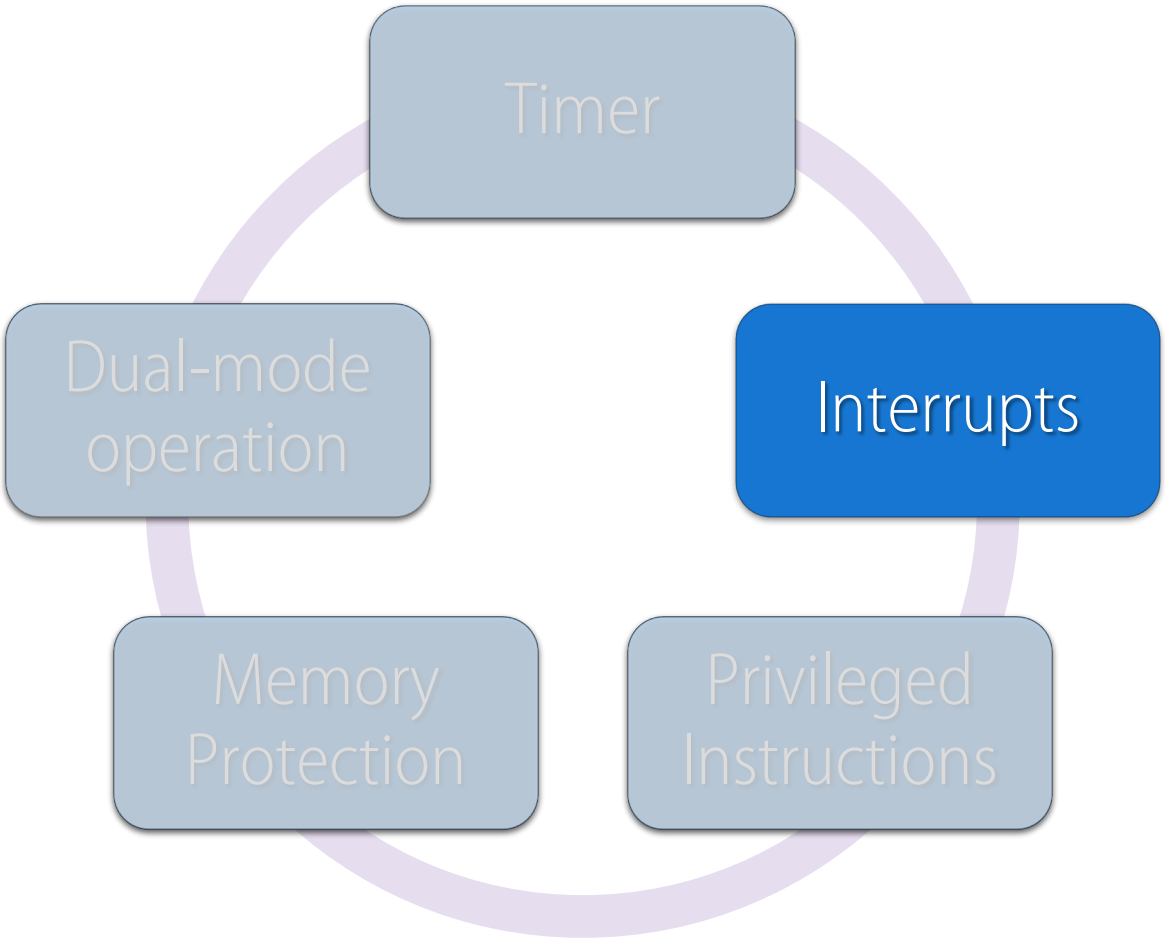


Hardware features for multiprogramming



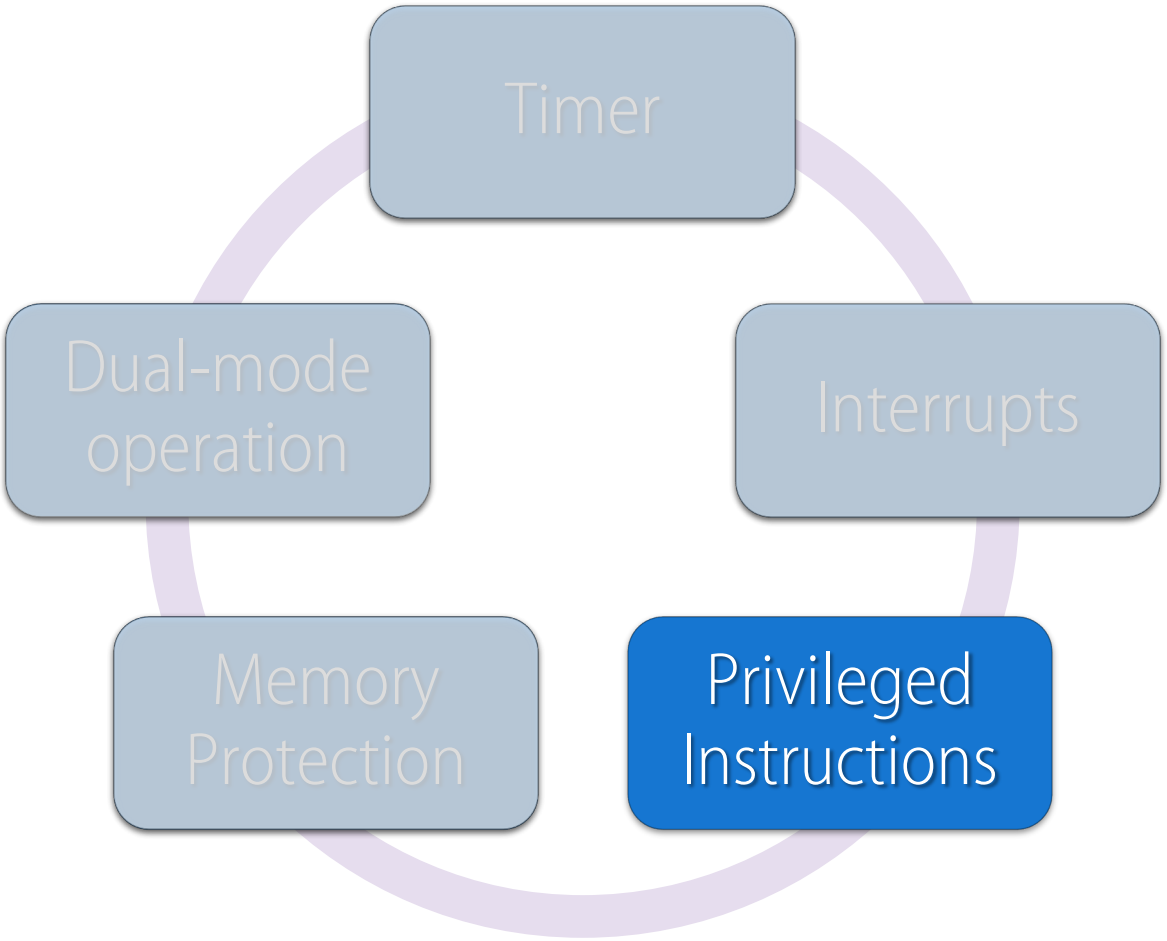
- Enables interrupting a user job after a certain amount of time

Hardware features for multiprogramming



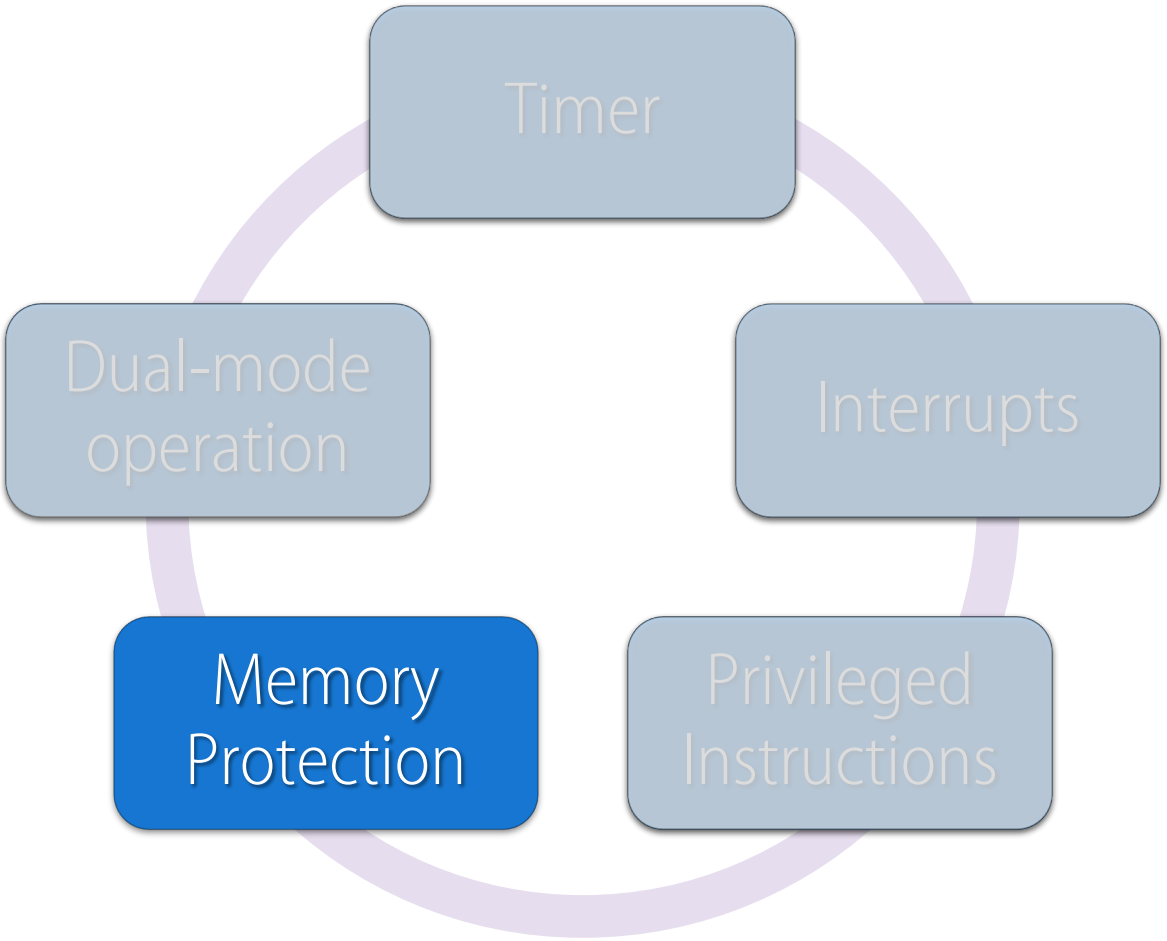
- Enable switching between jobs while waiting for I/O to complete

Hardware features for multiprogramming



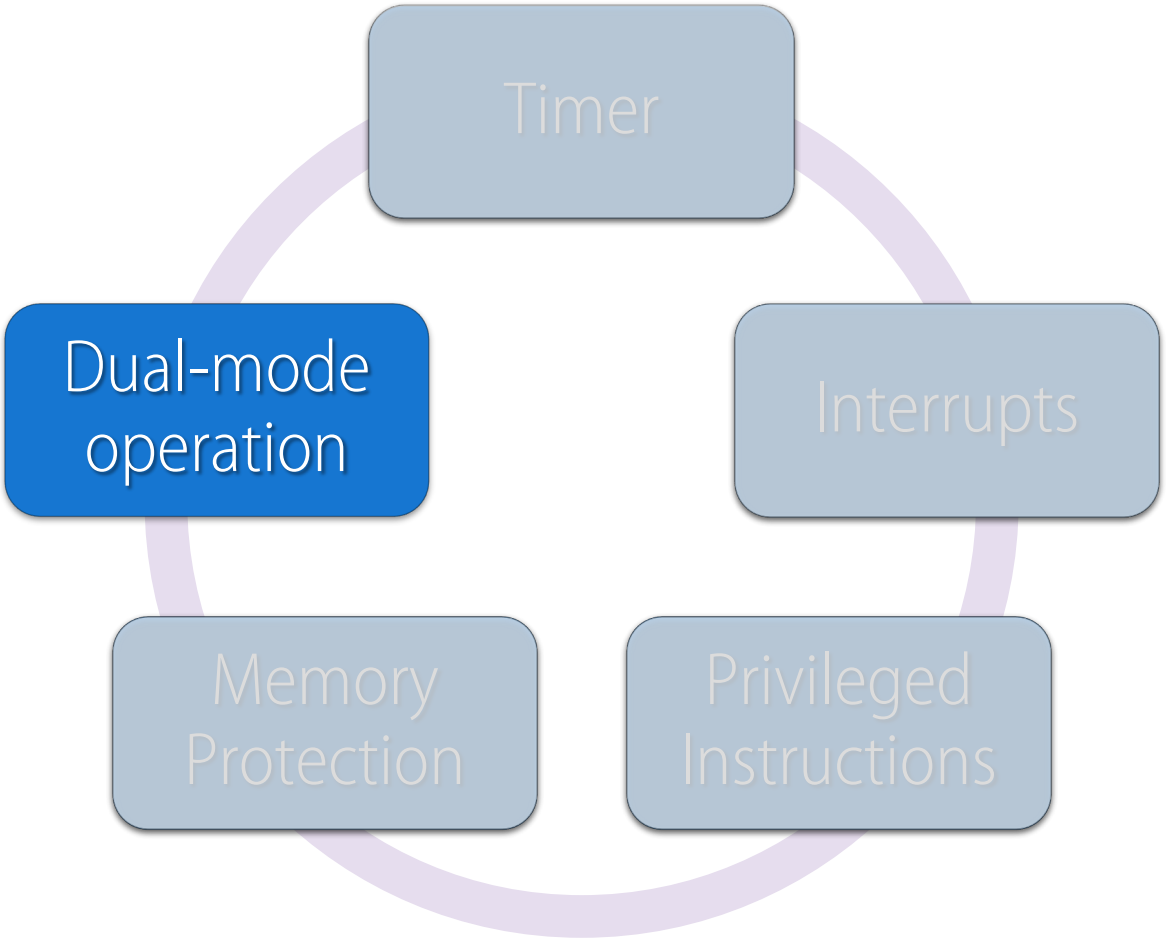
- User jobs cannot execute certain machine level instructions

Hardware features for multiprogramming



- A job cannot alter the memory area of the system or of another job

Hardware features for multiprogramming



- Jobs execute in user mode
  - Privileged instructions may not be executed
- OS executes in system (aka kernel) mode
  - Privileged instructions may be executed
  - Protected areas of memory may be accessed

The Process Abstraction

A process can be understood as …

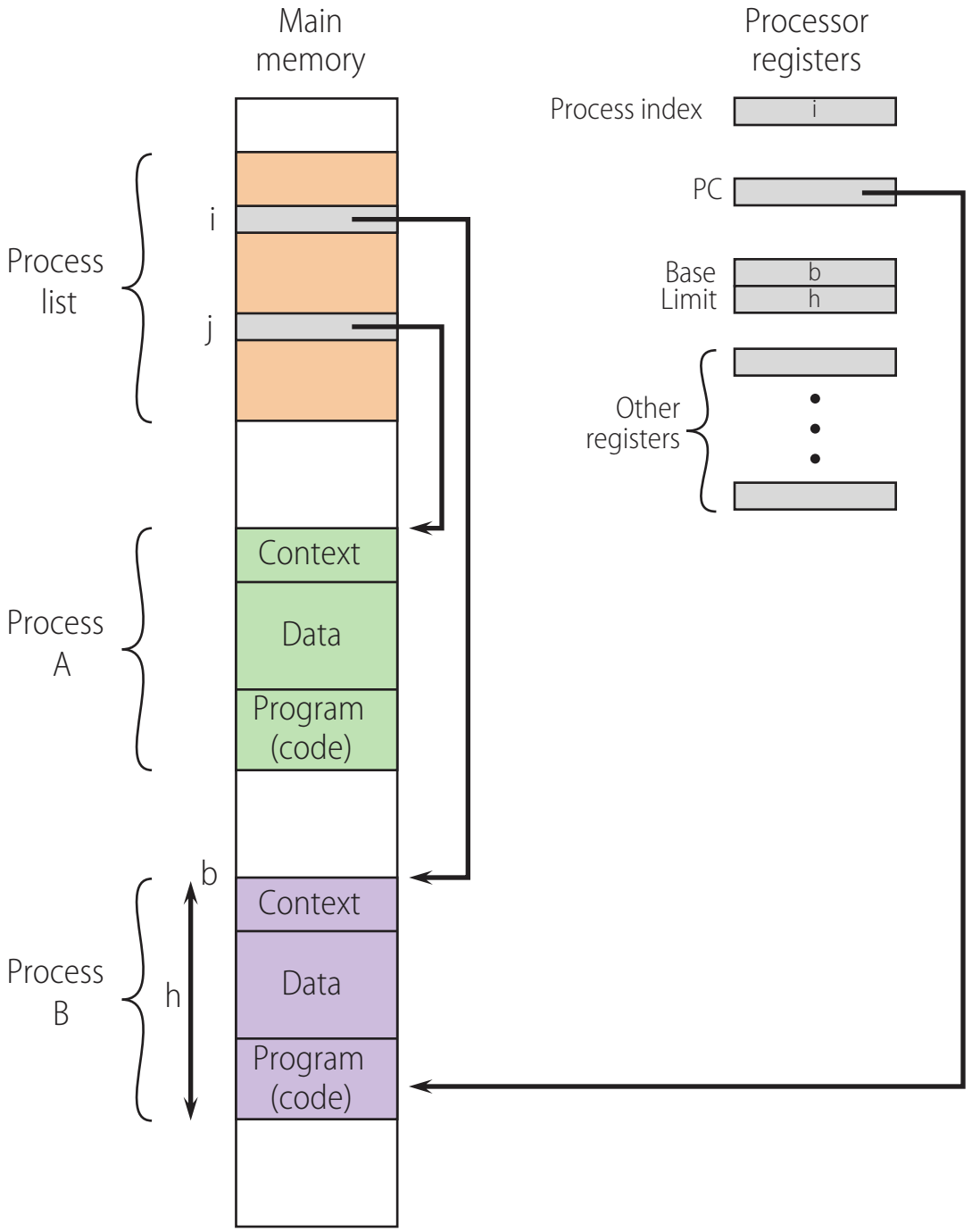
- A program in execution
- An instance of a program running on a computer
- An entity that can be assigned to and executed on a processor
- A unit of activity characterized by
  - a sequence of program instructions to be executed during its lifetime
  - an evolving state and
  - an associated set of required system services

A process . . .

- Consists of three components
  - An executable program
  - Associated data needed by the program
  - Execution context of the program aka process state
    - All information the operating system needs to manage the process
- Needs resources to accomplish its task
  - CPU, memory, I/O, files
  - Initialization data
- On termination, requires reclaim of any reusable resources

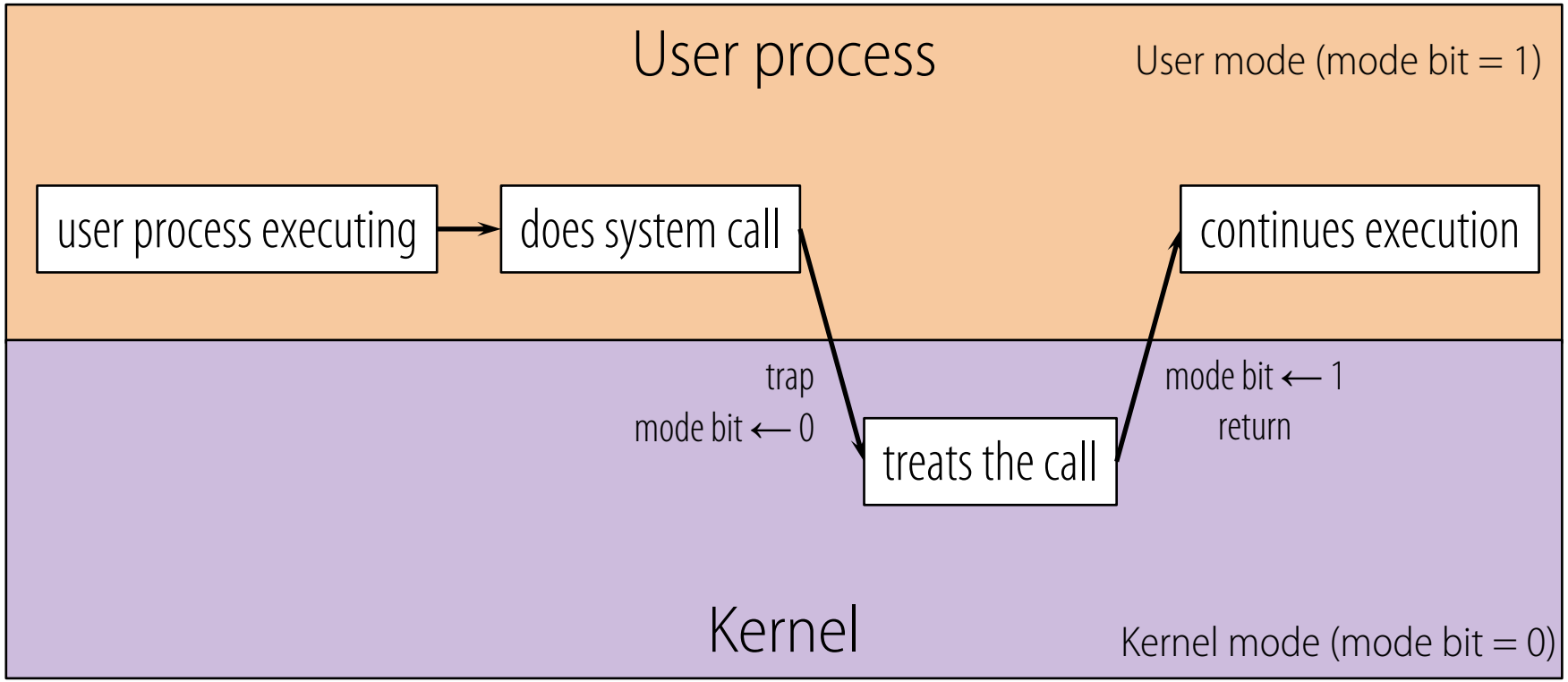
Simple process implementation

- A few concerns…
  - Does a process need to be wholly loaded in memory?
  - Can processes be switched in and out of memory?
  - If so, does a process need to be reloaded to the same memory addresses where it has been before?



How do a process and the OS interact?

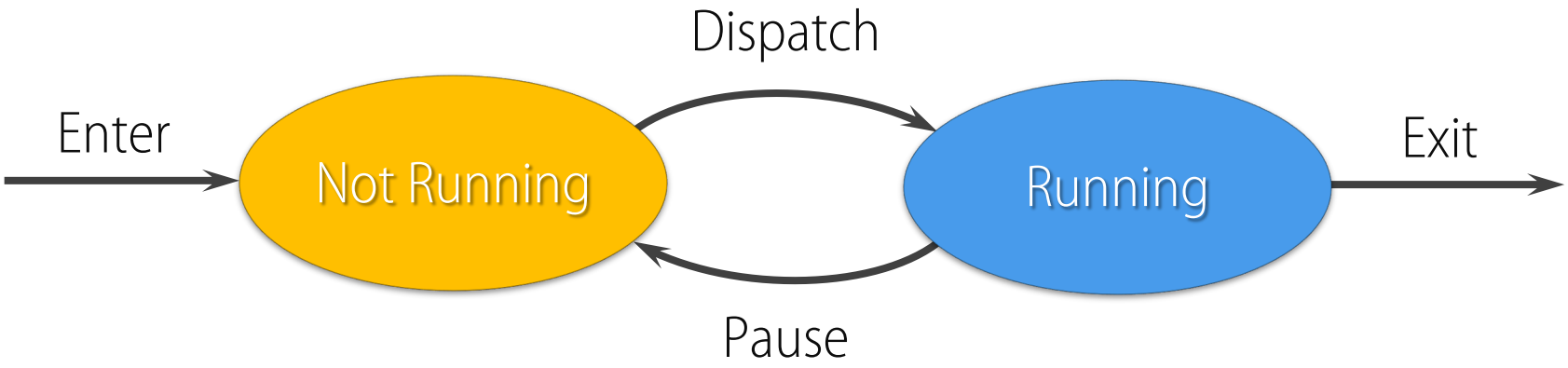
- Via system calls and dual-mode operation



Process States

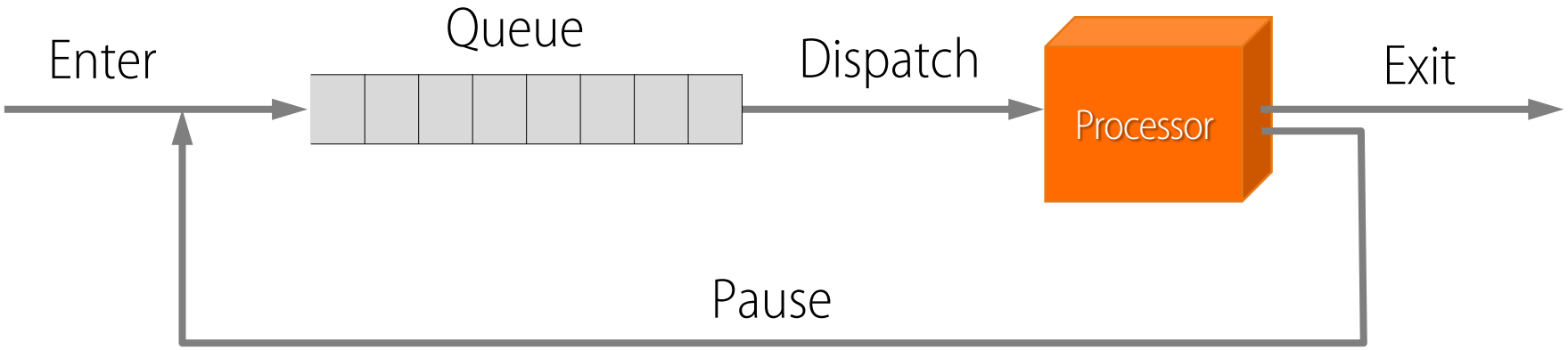
Two-state process model

- A process may be in one of two states
  - Running
  - Not-running

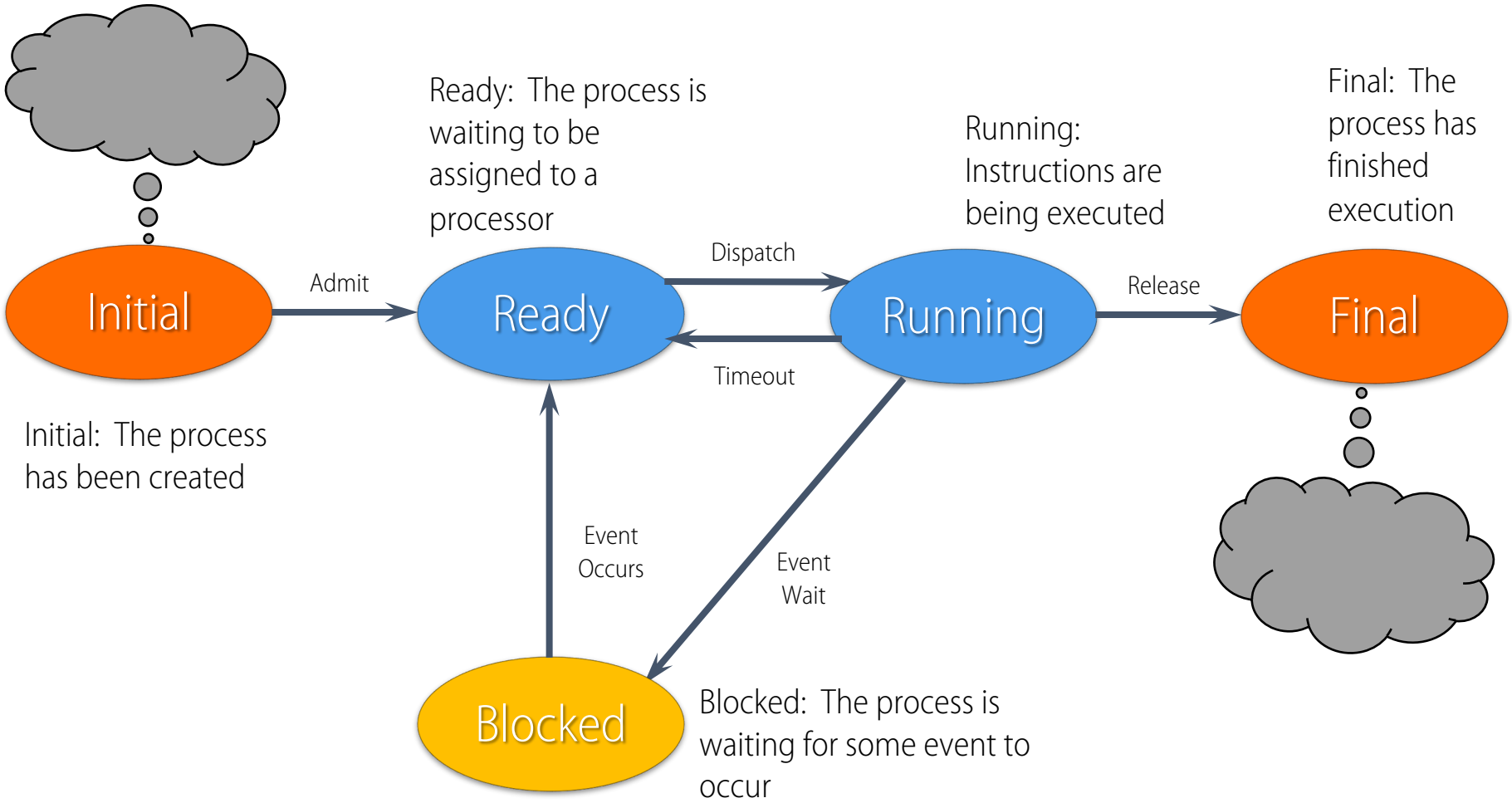


Queuing diagram for the 2-state model

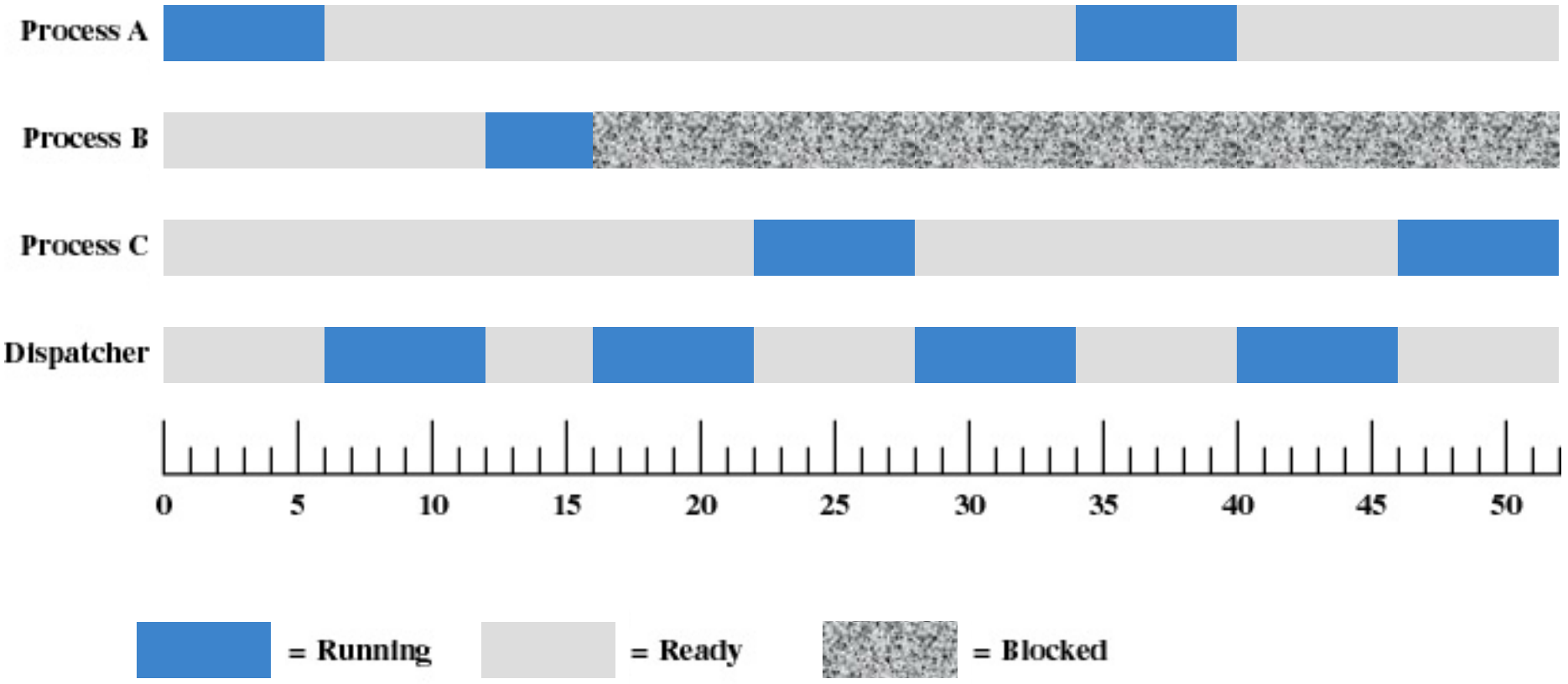
- What's wrong with this model?



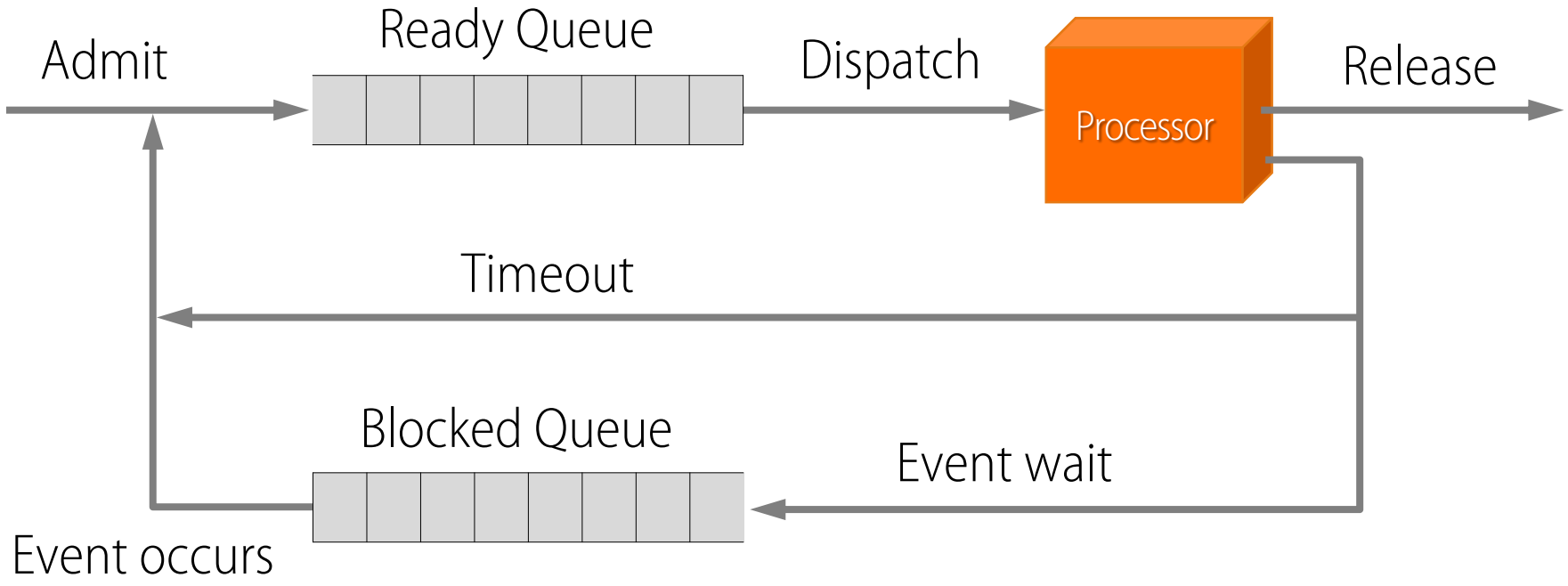
Process life-cycle in the 5-state model



States of the example processes



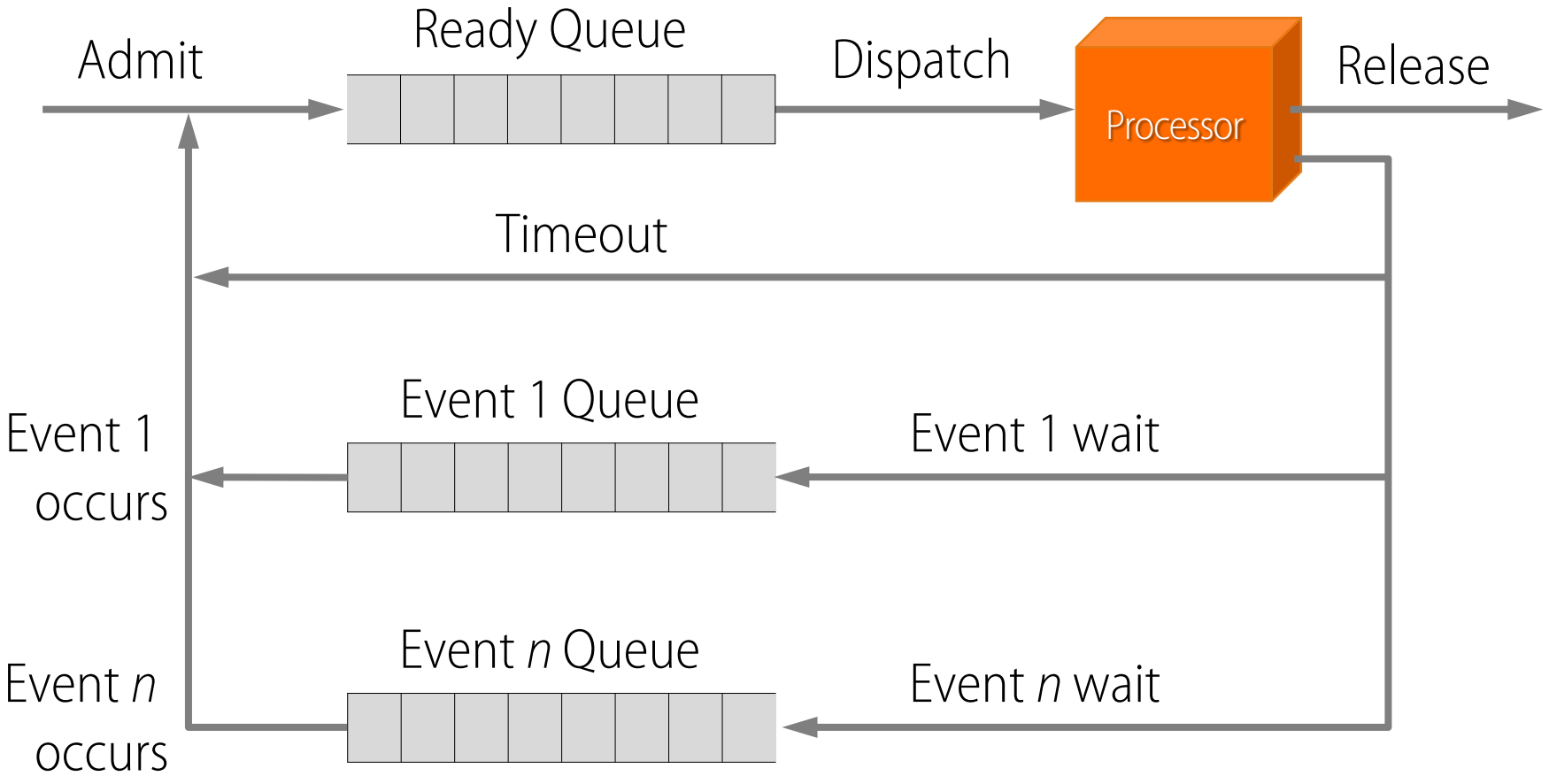
Queuing diagram for the 5-state model



Implementation of priorities

- When scheduling is based on process priority it makes sense to create several “ready” queues, one for each priority level.
- In this way, the OS can easily locate the highest priority “ready” process that has been waiting for the longest time.

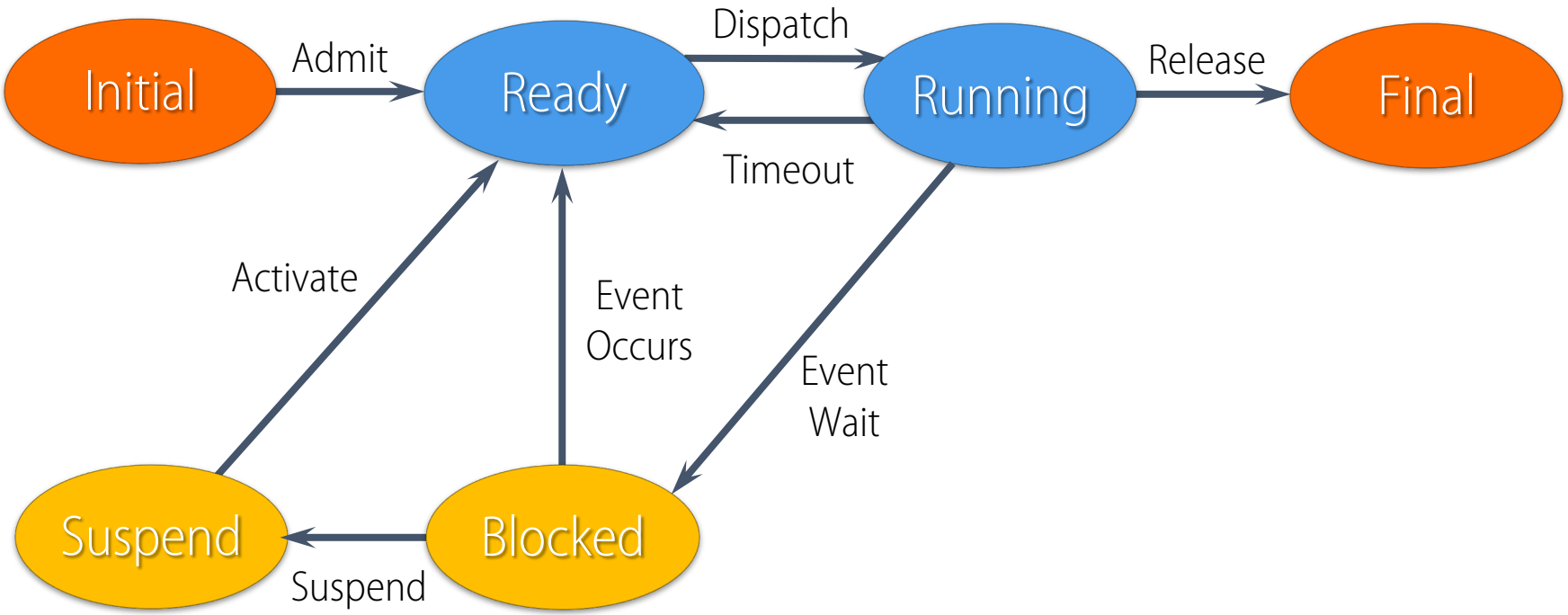
5-state model with several blocked queues



Process suspension

- Processor is faster than I/O so all processes could be waiting for I/O
- The OS can swap these blocked processes to disk to free up more memory
- *Blocked* state becomes *suspend* state when process has been swapped to disk

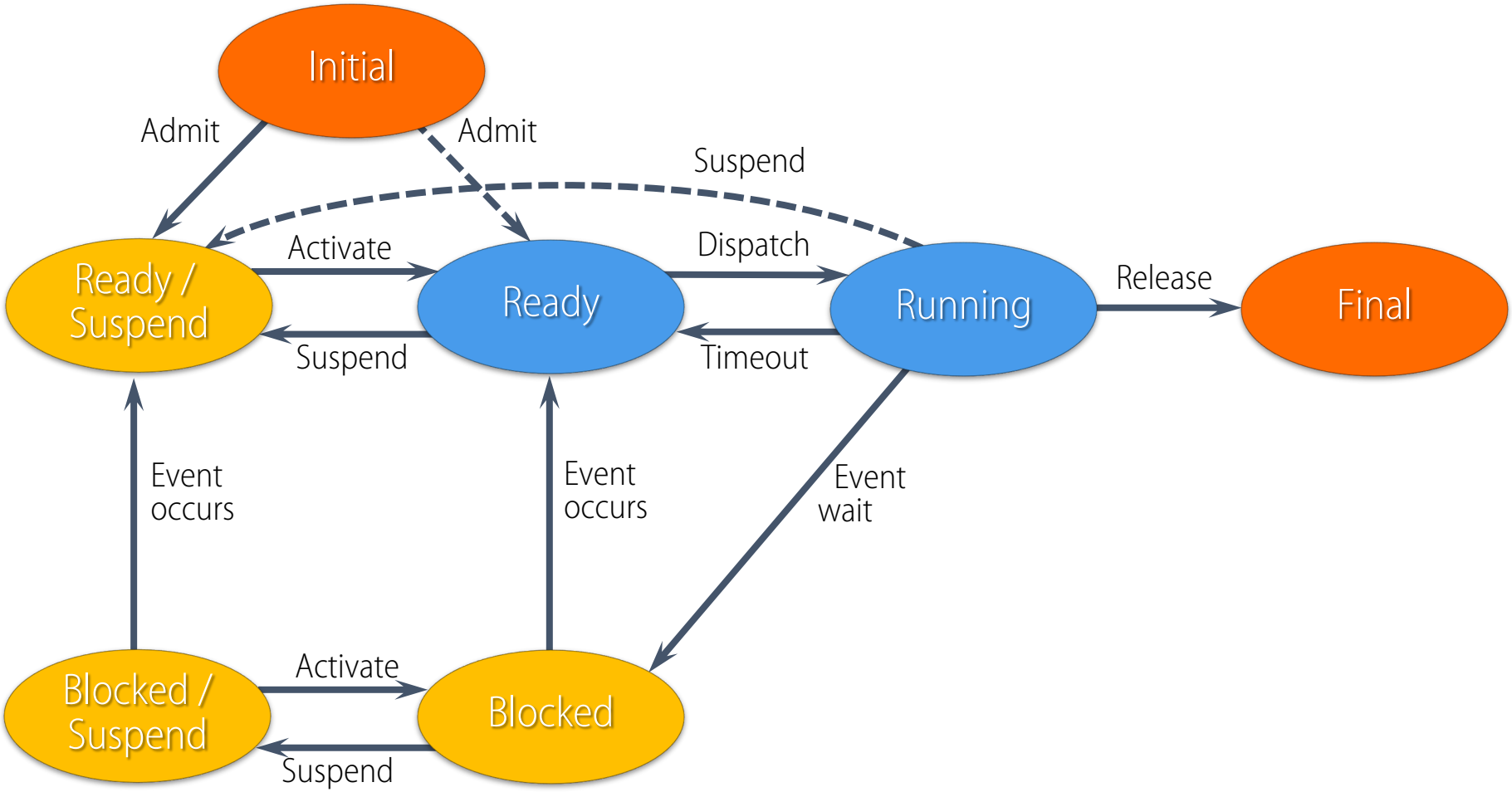
A 6-state process life-cycle model



A model with 2 “suspend” states

- There are two independent concepts
  - A process may be ready or blocked
  - A process may be suspended or not
- Four states are required to model such 2×2 arrangement
  - Ready
  - Blocked
  - Ready/Suspended
  - Blocked/Suspended

A 7-state process life-cycle model



Suspended process characteristics

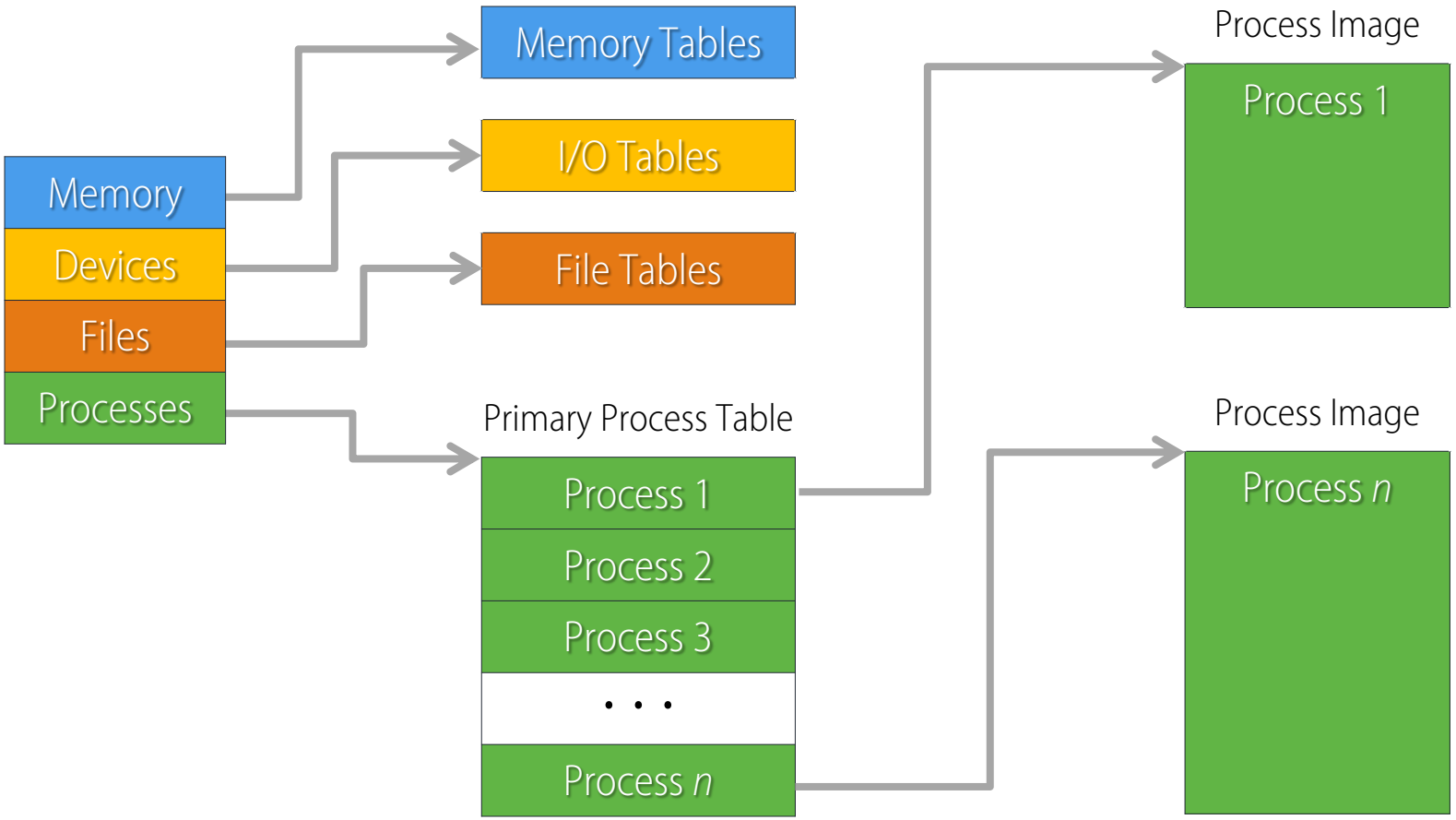
- It is not available for immediate execution
- It may be waiting for an event
  - In this case, the blocked condition is independent from the suspend condition
- It may have been suspended by an agent and now its state can only be changed by that agent.

Process Description

Operating system control structures

- Information about the current status of each process and resource
- Tables are constructed for each entity managed by the operating system
  - Memory
  - I/O
  - Files
  - Processes
- Such tables must be linked somehow.
- To create the tables, the OS must know the details of the executing environment.

General structure of OS control tables



Memory Tables

- Allocation of main memory to processes
- Allocation of secondary memory to processes
- Protection attributes for access to shared memory regions
- Information needed to manage virtual memory

I/O Tables

- I/O device is available or assigned
- Status of I/O operation
- Location in main memory being used as the source or destination of the I/O transfer

File Tables

- Existence of file
- Location of file on secondary memory
- Current file status
- File attributes

- Often this information is maintained by a file management system

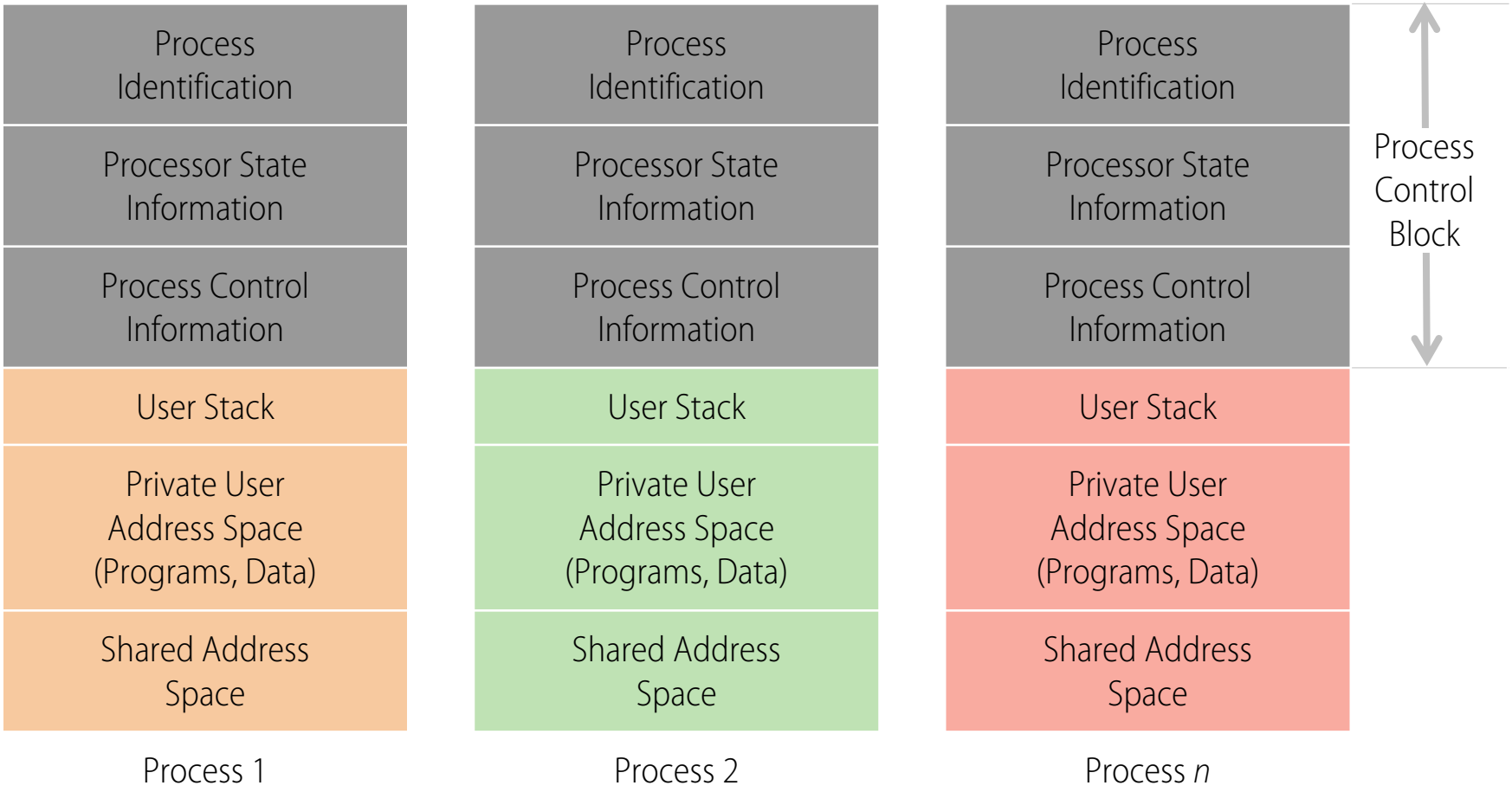
Process Table

- Where process is located
- Attributes in the process control block
  - Program
  - Data
  - Stack

Typical elements of a process image

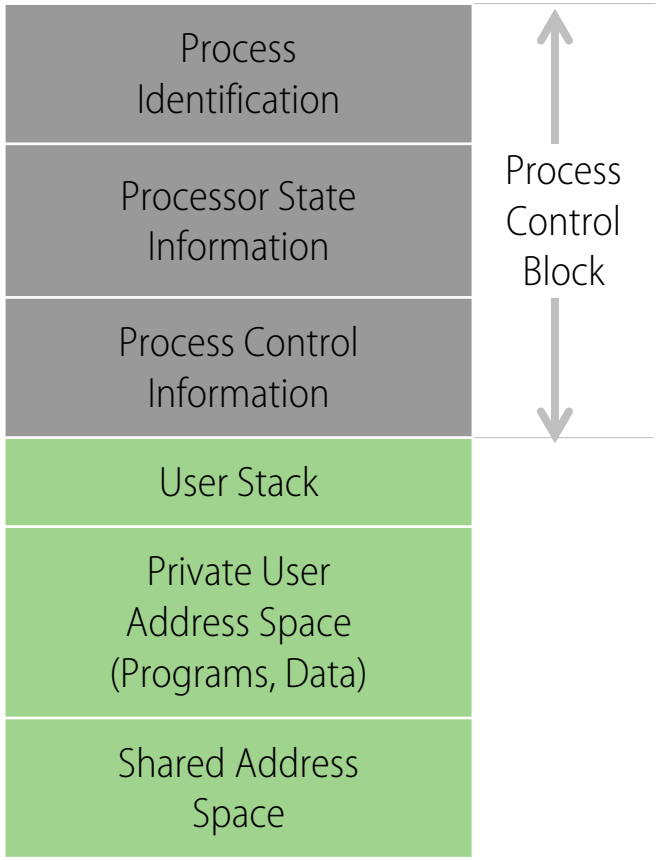
User data	• The modifiable part of the user space. • May include program data, a user stack area and programs that may be modified.
User program	• The code of the program to be executed.
System stacks	• Used as temporary storage for parameters, intermediate results and return addresses for procedure and system calls.
PCB Process Control Block	• Data needed by the OS to control the process, such as *process id, processor state,* etc.

User processes in virtual memory



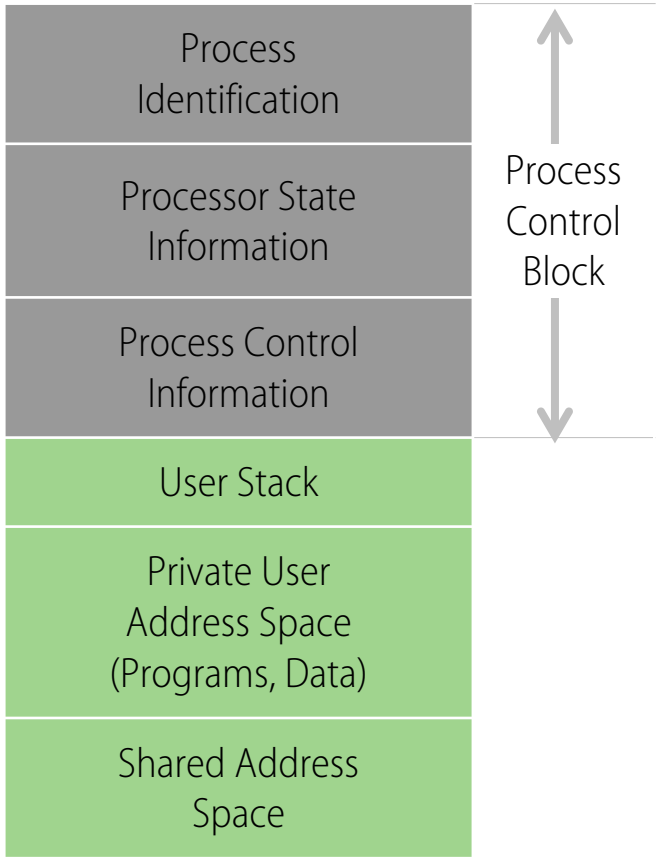
Process Control Block

- Process identification
  - Identifiers
    - Identifier of this process
    - Identifier of the process that created this process (parent process)
    - User identifier
    - Etc.



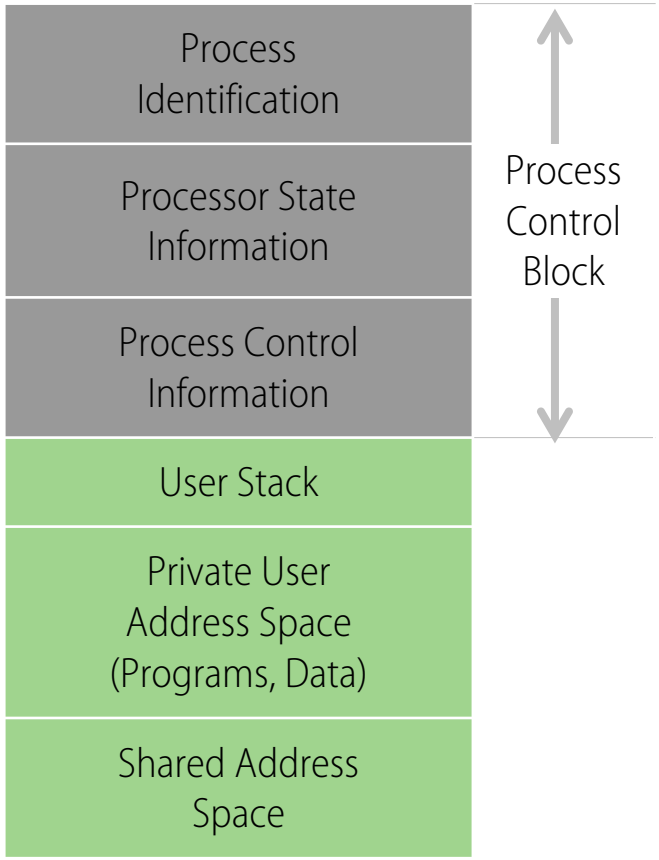
Process Control Block

- Processor State Information
  - User-Visible Registers
    - A user-visible register is one that may be referenced by means of the machine language that the processor executes while in user mode.
    - Typically, there are from 8 to 32 of these registers, although some RISC implementations have over 100.



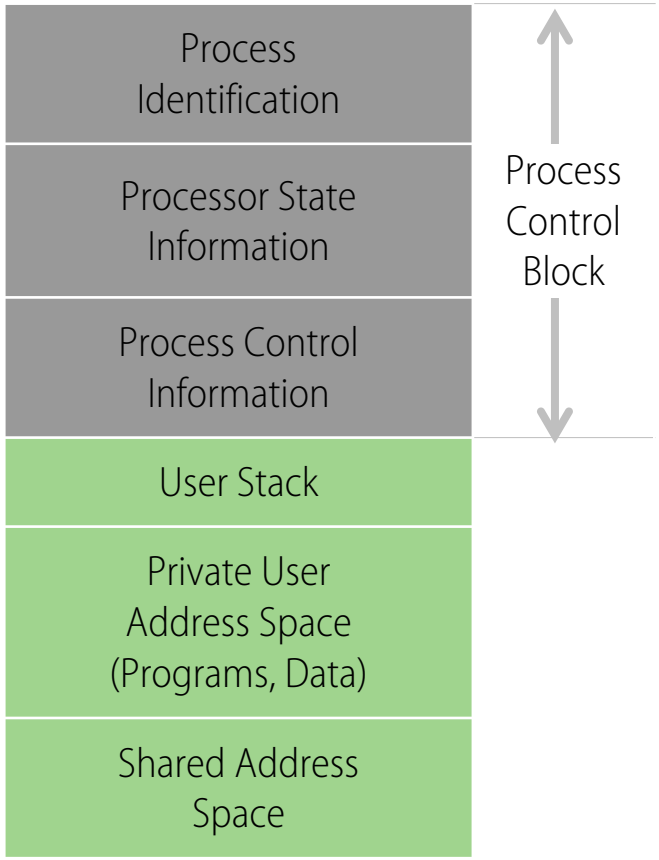
Process Control Block

- Processor State Information
  - Program Status Word (PSW)
    - A variety of processor registers that are employed to control the operation of the processor, e.g.
      - Program counter
      - Condition codes
      - Status information



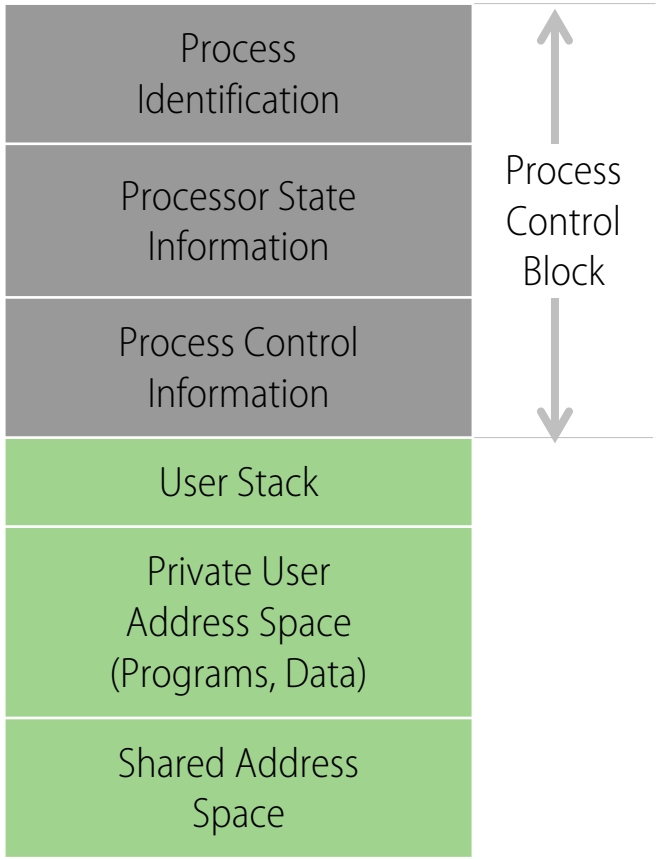
Process Control Block

- Processor State Information
  - Stack Pointers
    - Each process has one or more last-in-first-out (LIFO) system stacks associated with it.
    - A stack is used to store parameters and calling addresses for procedure and system calls.
    - The stack pointer points to the top of the stack.



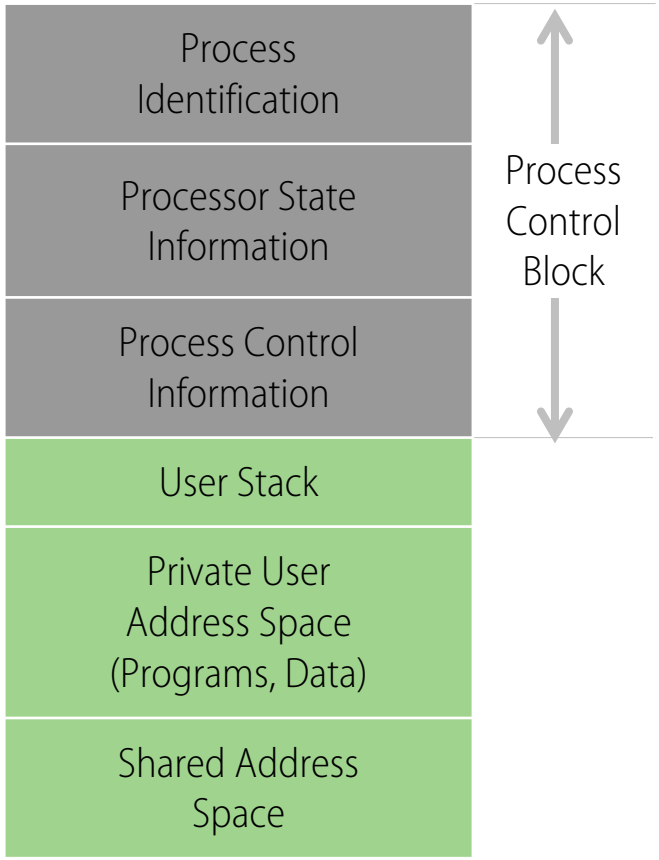
Process Control Block

- Process Control Information
  - Scheduling and State Information
    - Information needed by the OS to perform its scheduling function, e.g.
      - Process state
      - Priority
      - Scheduling-related information
      - Event



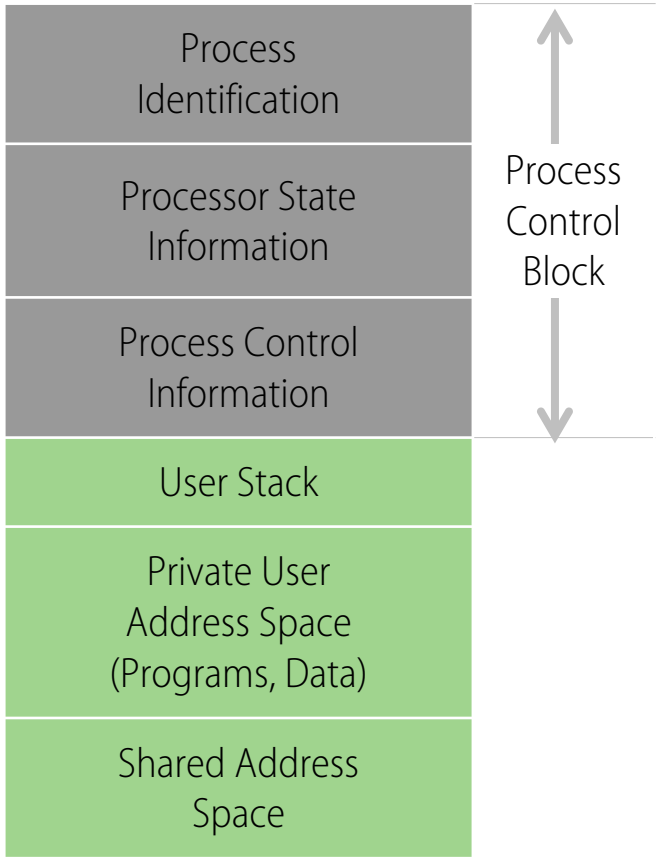
Process Control Block

- Process Control Information
  - Data Structuring
    - A process may be linked to another process in a queue, ring, or some other structure.
      - For example, all processes in a waiting state for a particular priority level may be linked in a queue.
    - A process may exhibit a parent-child (creator-created) relationship with another process.
      - The PCB may contain pointers to other processes to support these structures.



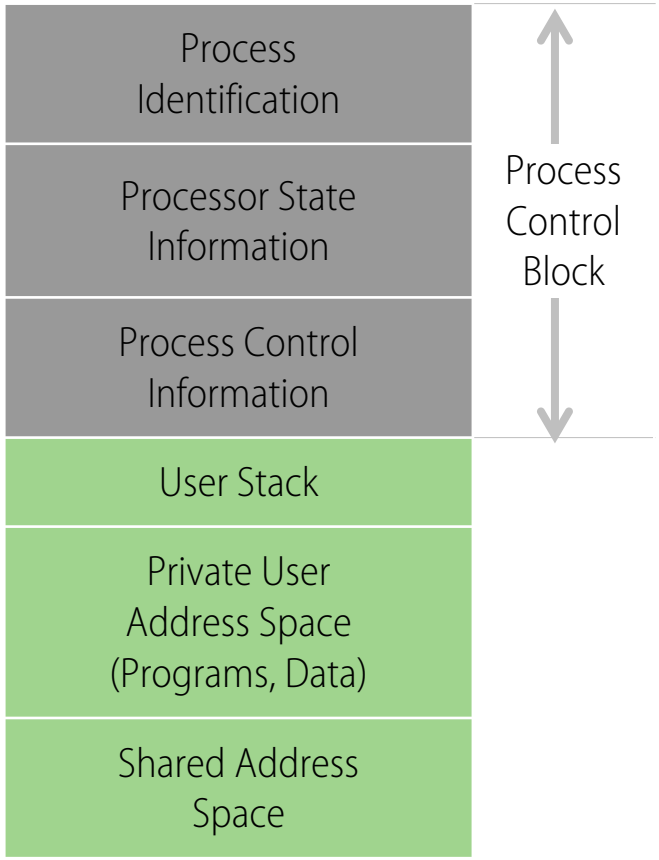
Process Control Block

- Process Control Information
  - Interprocess Communication
    - Various flags, signals, and messages may be associated with communication between two independent processes and kept in the PCB.
  - Process Privileges
    - Processes may be granted privileges in terms of the memory that may be accessed, the types of instructions that may be executed and the use of system utilities and services.

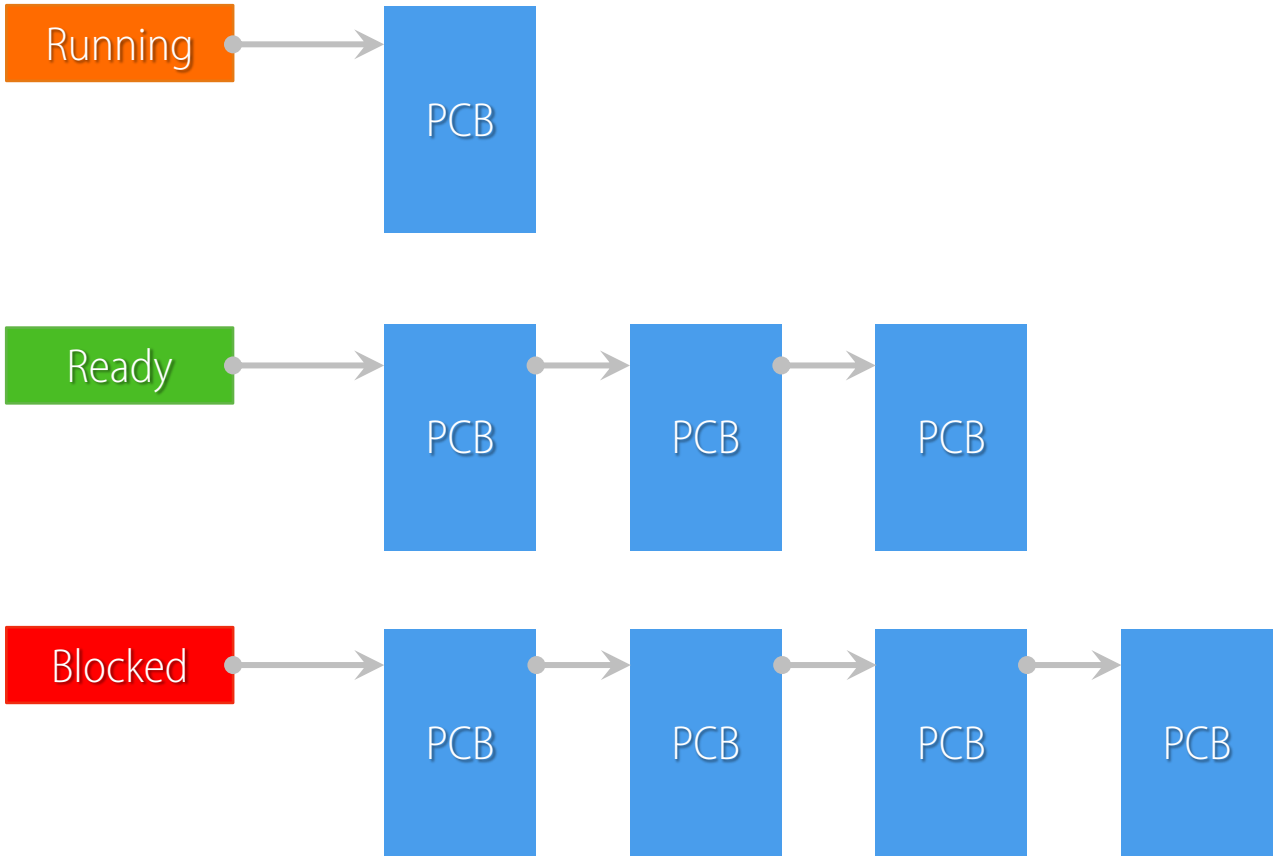


Process Control Block

- Process Control Information
  - Memory Management
    - This section may include pointers to segment and/or page tables that describe the virtual memory assigned to this process.
  - Resource Ownership and Utilization
    - Resources controlled by the process may be indicated, e.g. opened files.
    - A history of utilization of the processor or other resources may also be included.
      - This information may be needed by the scheduler.

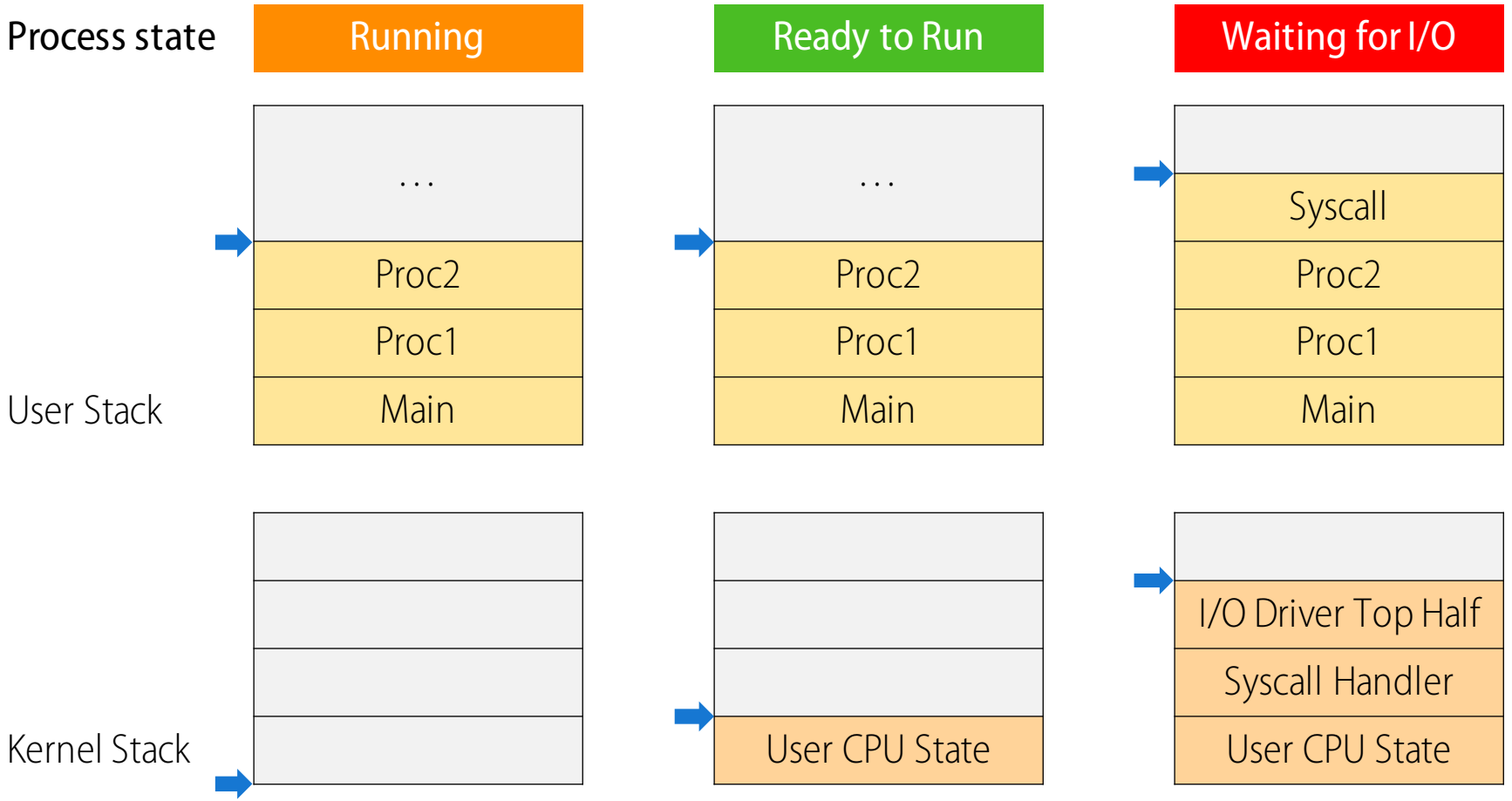


Process List Structures



User and Kernel Stacks

- Most operating systems allocate a kernel interrupt stack for each user-level process.



xv6, a simple Unix-like teaching operating system

- xv6 is a teaching operating system developed in the summer of 2006 for MIT's operating systems course, 6.828: Operating System Engineering, which will be cited in several examples during our course.
- A detailed description of xv6 and additional material can be found at https://pdos.csail.mit.edu/6.828/2017/xv6.html
- The latest xv6 source (licensed under the traditional MIT license) is available via `git clone git://github.com/mit-pdos/xv6-public.git`
- A booklet giving the sources with line numbers is available at https://pdos.csail.mit.edu/6.828/2017/xv6/xv6-rev10.pdf.
- A 100-page textbook that may help you to read through xv6 and learn about the main ideas in operating systems is also available at https://pdos.csail.mit.edu/6.828/2017/xv6/book-rev10.pdf.
  - The line numbers in this book refer to the above source booklet.

xv6: saved registers for kernel control switches

```
17  // Saved registers for kernel context switches.
18  // Don't need to save all the segment registers (%cs, etc),
19  // because they are constant across kernel contexts.
20  // Don't need to save %eax, %ecx, %edx, because the
21  // x86 convention is that the caller has saved them.
22  // Contexts are stored at the bottom of the stack they
23  // describe; the stack pointer is the address of the context.
24  // The layout of the context matches the layout of the stack in swtch.S
25  // at the "Switch stacks" comment. Switch doesn't save eip explicitly,
26  // but it is on the stack and allocproc() manipulates it.
27  struct context {
28    uint edi;
29    uint esi;
30    uint ebx;
31    uint ebp;
32    uint eip;
33  };
```

xv6: the different states a process can be in

```
35    enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

xv6: information tracked about each process

```
37  // Per-process state
38  struct proc {
39    uint sz;                     // Size of process memory (bytes)
40    pde_t* pgdir;                // Page table
41    char *kstack;                // Bottom of kernel stack for this process
42    enum procstate state;        // Process state
43    int pid;                     // Process ID
44    struct proc *parent;         // Parent process
45    struct trapframe *tf;        // Trap frame for current syscall
46    struct context *context;     // swtch() here to run process
47    void *chan;                  // If non-zero, sleeping on chan
48    int killed;                  // If non-zero, have been killed
49    struct file *ofile[NOFILE];  // Open files
50    struct inode *cwd;           // Current directory
51    char name[16];               // Process name (debugging)
52  };
```

Simulation homeworks

- At the end of many chapters of the textbook you will find simulation homeworks, which come in the form of simulators you run to make sure you understand some piece of the material.
- The simulators are generally python programs that enable you both to generate different problems (using different random seeds) as well as to have the program solve the problem for you (with the `-c` flag) so that you can check your answers.
- Running any simulator with a `-h` or `--help` flag will provide with more information as to all the options the simulator gives you.
- The README provided with each simulator gives more detail as to how to run it. Each flag is described in some detail therein.