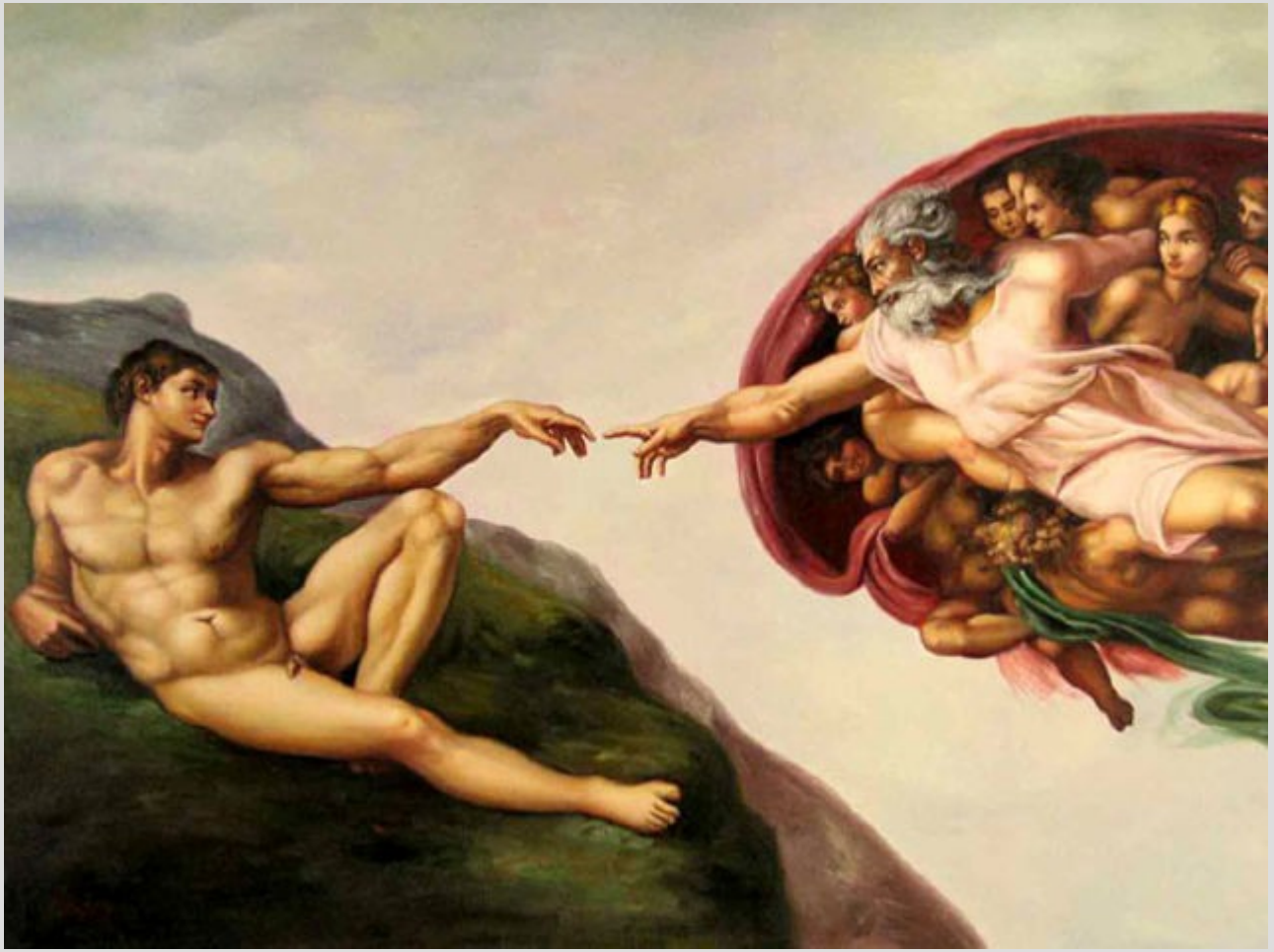
Parte 19

Processes



[Michelangelo – La Creazione di Adamo, 1511]

Process

- The fundamental concept in any operating system is the “**process**”
  - a process is an executing program
  - an OS can execute many processes at the same time → **Concurrency**

Process traces

- When a computer executes a sequential process, it goes through states
- At each step, the state of the process is changed
- State:
  - set of processor registers
  - set of process variables

Process history

- A program for computing GCD

```
int gcd(int a, int b)
{
        while (a!=b) {
                if (a < b) b = b – a;
                else a = a – b;
        }
        return a;
}
```

Steps	a	b
1	21	15
2	6	15
3	6	9
4	6	3
5	3	3

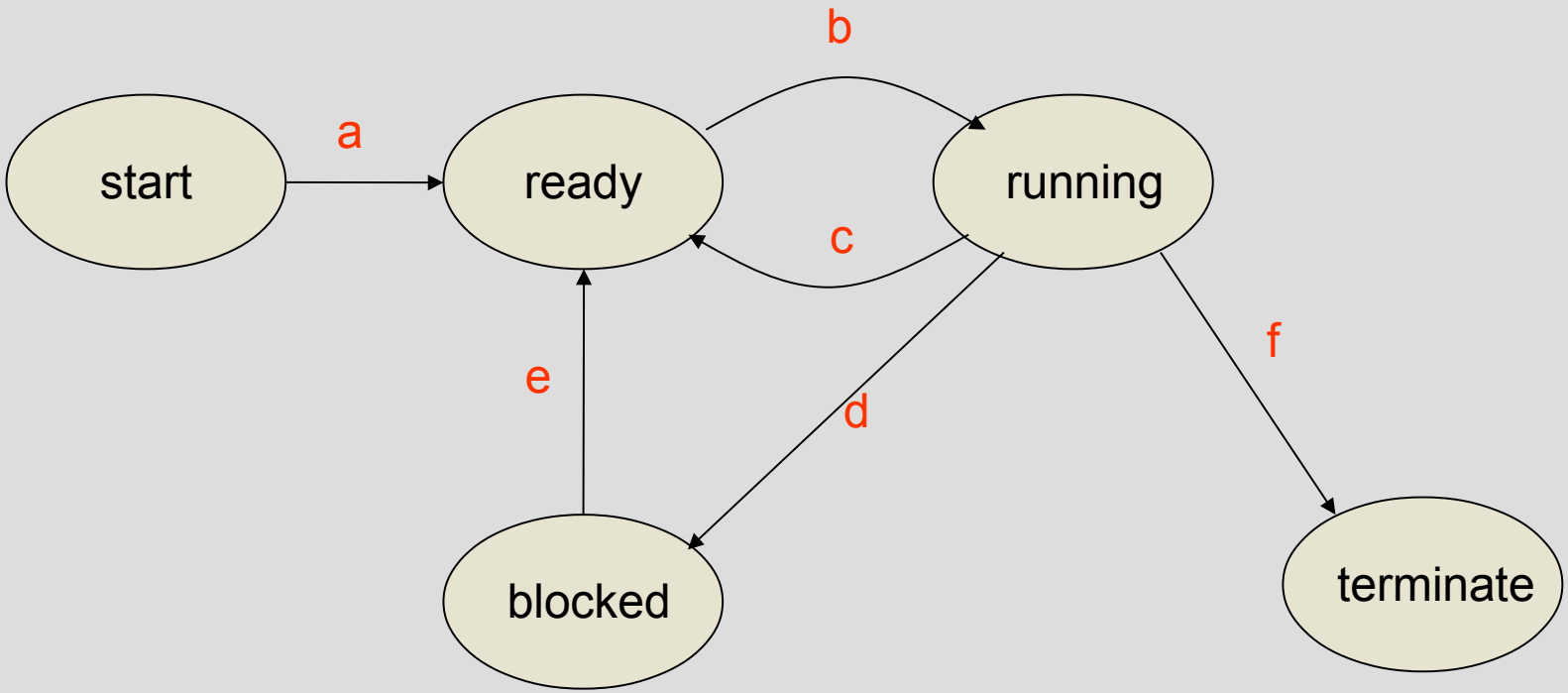
- Every time the program is launched, a new process is generated and run

Process states

- The OS can execute many processes at the same time
- Each process can be in one of the following states:
  - starting (the process is being created)
  - ready (the process is ready to be executed)
  - executing (the process is executing)
  - blocked (the process is waiting on a condition)
  - terminating (the process is about to terminate)

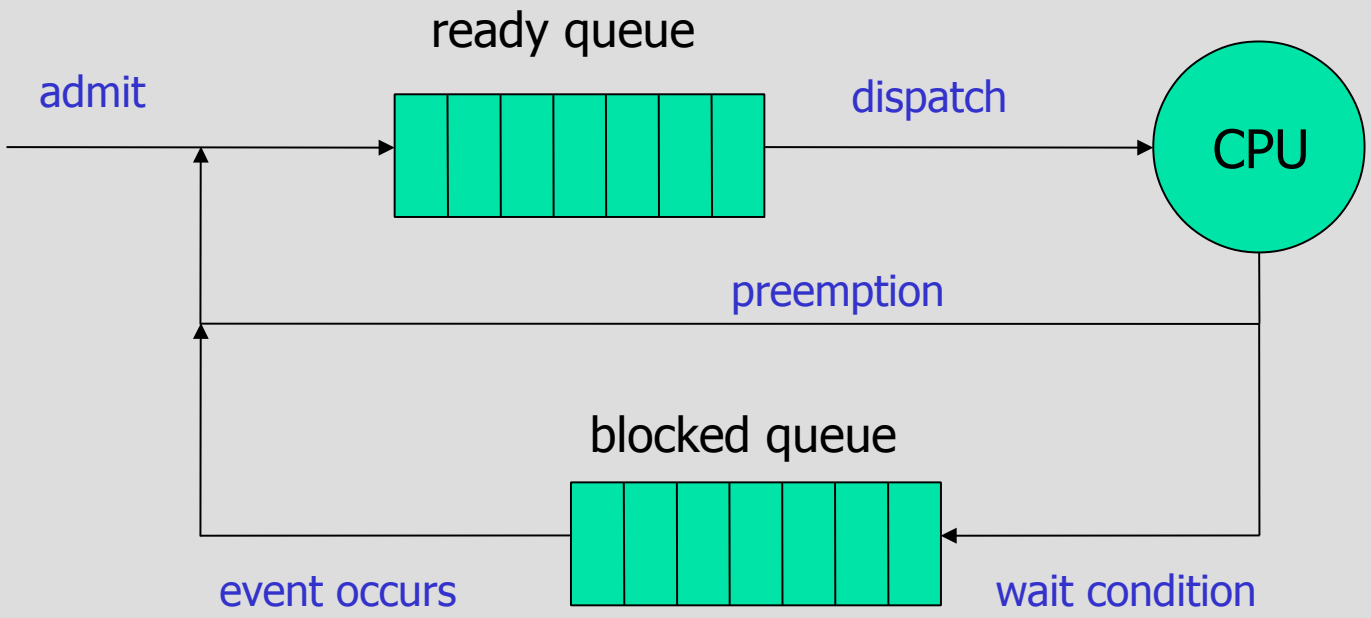
Process states

a)	Creation	the process is created
b)	Dispatch	the process is selected to execute
c)	Preemption	the process leaves the processor
d)	Wait on condition	the process is blocked on a condition
e)	Condition true	the process is unblocked
f)	Exit	the process terminates



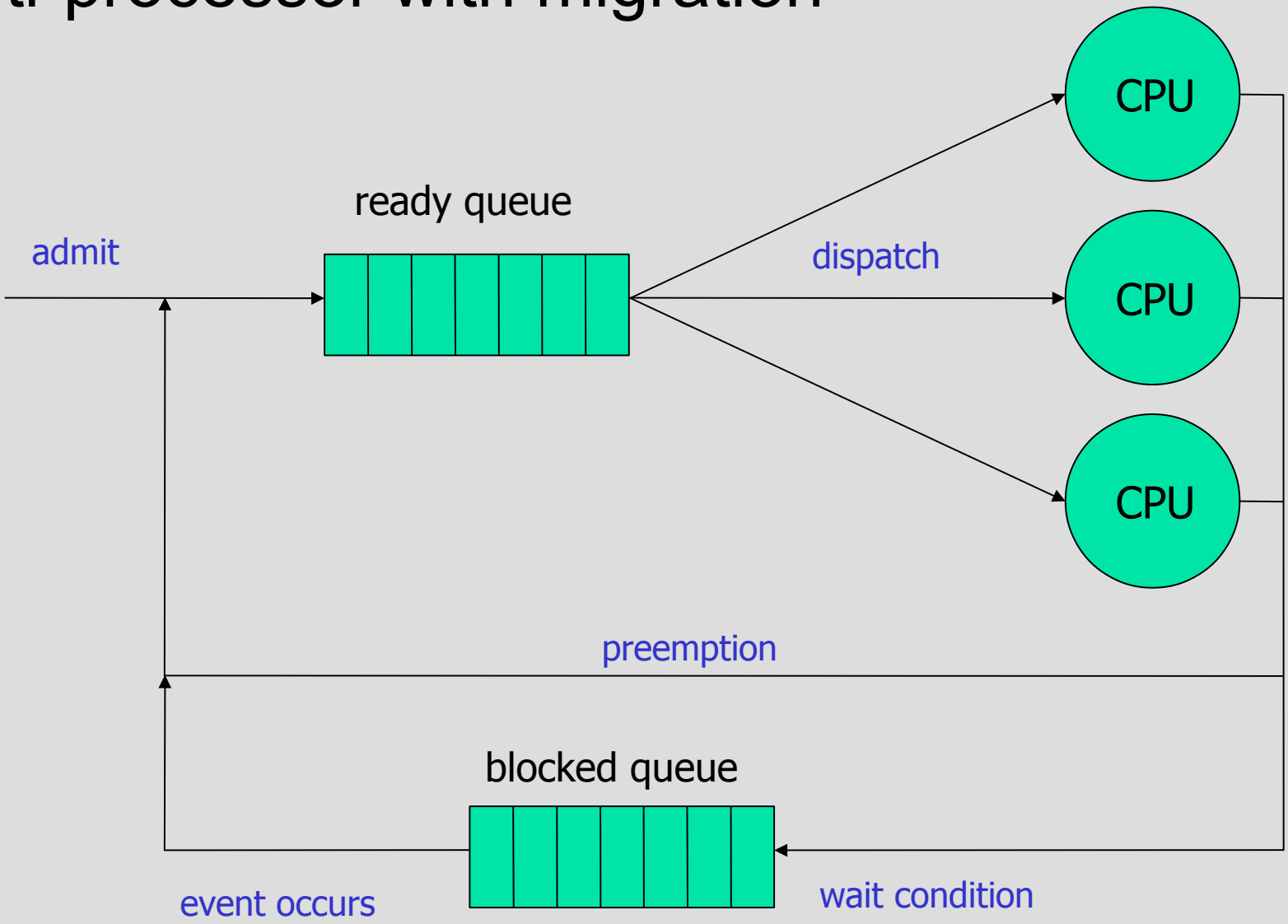
Process queues

- Single processor

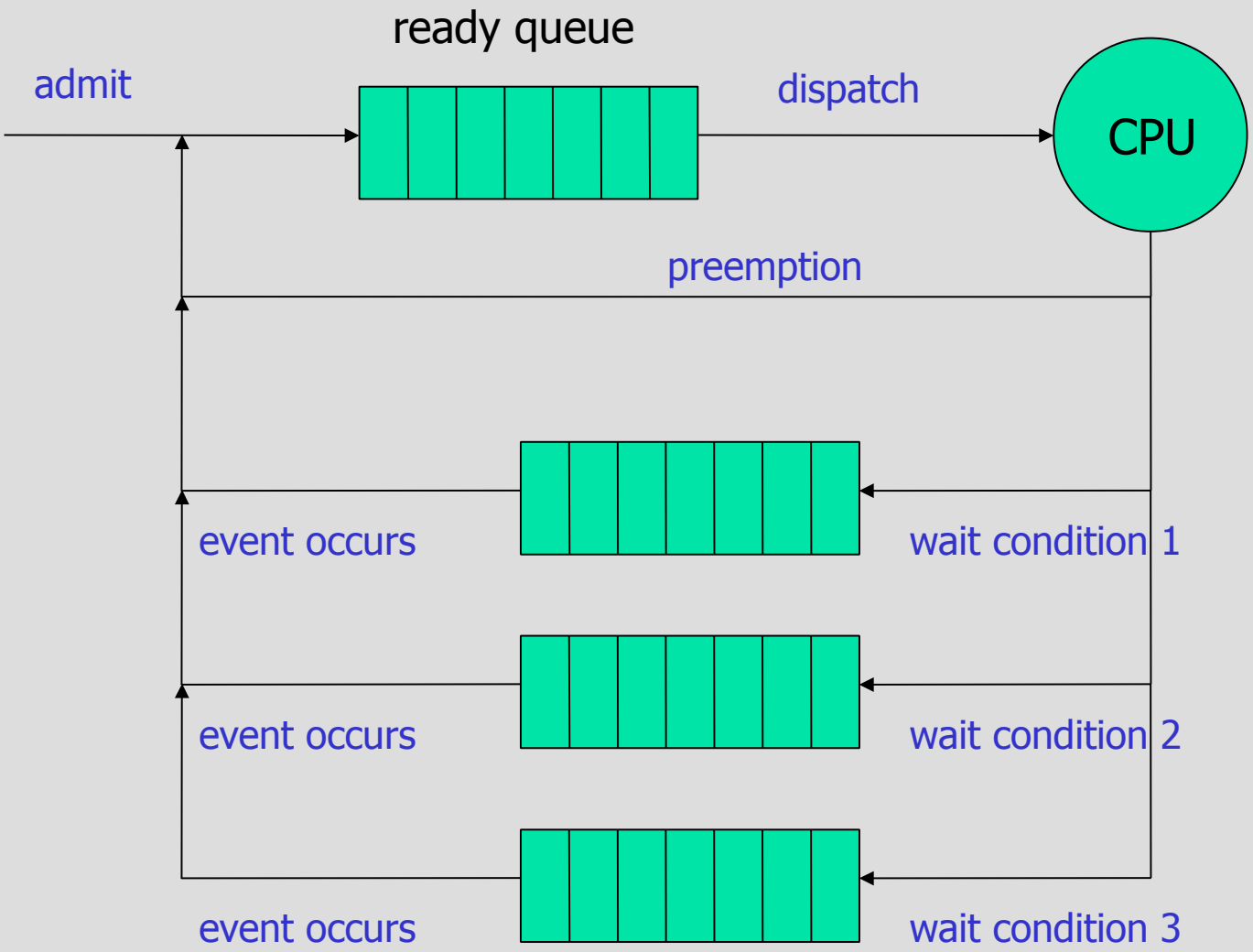


Process queues

- Multi-processor with migration

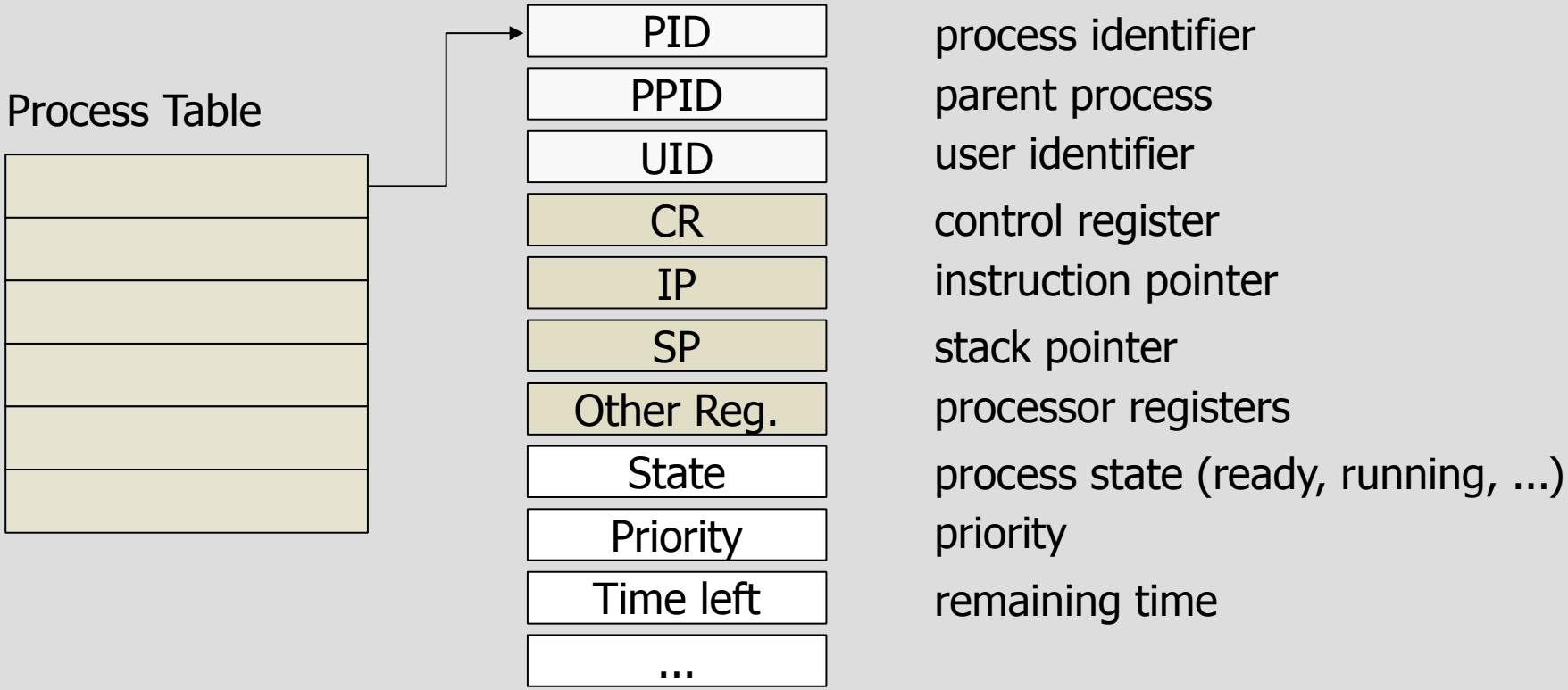


Multiple blocking queues



Process Control Block

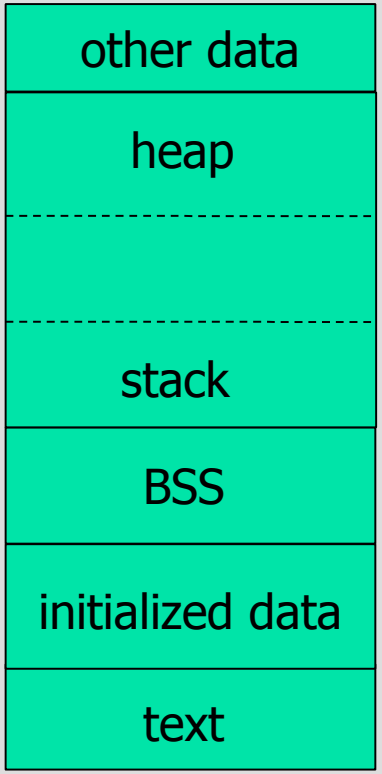
- Contains all data concerning a process
- PCBs are stored in the Process Table



The role of PCB

- PCB is a critical point of any OS
- Accessed by multiple OS routines:
  - the scheduler
  - the Virtual memory
  - the Virtual File System
  - interrupt handlers (I/O devices)
  - ...
- It can only be accessed by the OS!
- The user can access some of the information in the PCB by using appropriate system calls

Memory layout of a Process



Process heap (for dynamic memory)

Process stack

Data NOT initialized by the loader

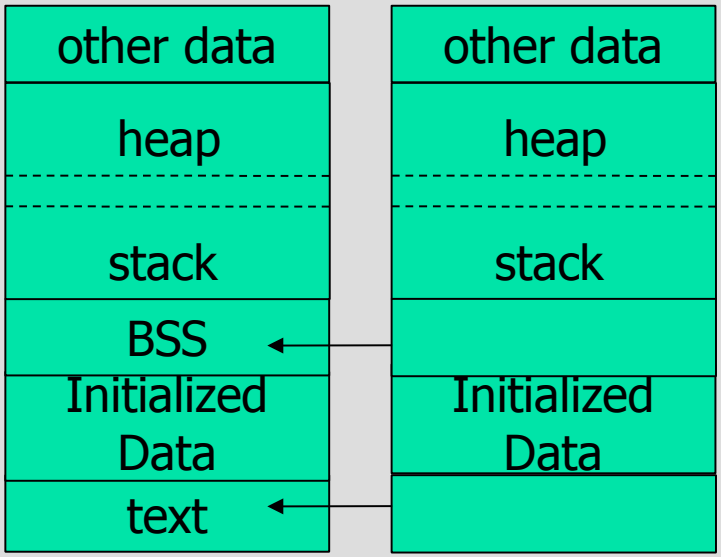
Data initialized by the loader

Process code

- In some system, the PCB is in the “other data” section
  - for example, Linux

Memory protection

- Every process has its own memory space
  - Part of it is “private to the process”
  - Part of it can be shared with other processes
  - Two processes of the same program may share the TEXT part
  - Two processes communicating by shared memory may share a portion of the data segment



Memory protection

- By default, two processes cannot share their memory
  - If one process tries to access a memory location outside its space, a processor exception is raised (trap) and the process is terminated
  - The famous “Segmentation Fault” error!!
  - Segmentation fault is also related to the Virtual Memory

Modes of operation (revised)

- Every modern processor supports at least two modes of operation
  - User
  - Supervisor
- A bit in the Control Register (CR) tells in which mode the processor is running
- OS routines run in supervisor mode
  - They need to operate freely on every part of the hardware with no restriction
- User code runs in user mode
- Mode switch
  - From user to supervisor mode or viceversa

Mode switch

- It can happen in one of the following cases
  - Interrupts or traps
    - In this case, before calling the interrupt handler, the processor goes in supervisor mode and disables interrupts
    - Traps are interrupts that are raised when a critical error occurs (e.g., division by zero, page fault, or for debugging purposes)
    - Returning from the interrupt restores the previous mode
  - Invoking a special instruction
    - This special instruction is used to implement system primitives
    - In the x86 family, it is the INT instruction
    - This instruction is similar to an interrupt
    - It takes a number that identifies a “service”
    - All OS calls are invoked by calling INT
    - Returning from the handler restores the previous mode

Example of system call

```
int fd,n;
char buff[100];

fd = open("Dummy.txt", O_RDONLY);
n =  read(fd, buff, 100);
```

- Saves parameters on the stack
- Executes INT 21h
  1. Change to supervisor mode
  2. Save context
  3. Execute the function open
  4. Restores the context
  5. IRET
- Back to user mode
- Delete parameters from the stack

- The "open" system call can potentially block the process!
  - in that case we have a "process switch"

Process switch

- It happens when
  - The process has been “preempted” by another higher priority process
  - The process blocks on some condition
  - In time-sharing systems, the process has completed its “round” and it is the turn of some other process
- We must be able to restore the process later
  - Therefore, we must save its state before switching to another process

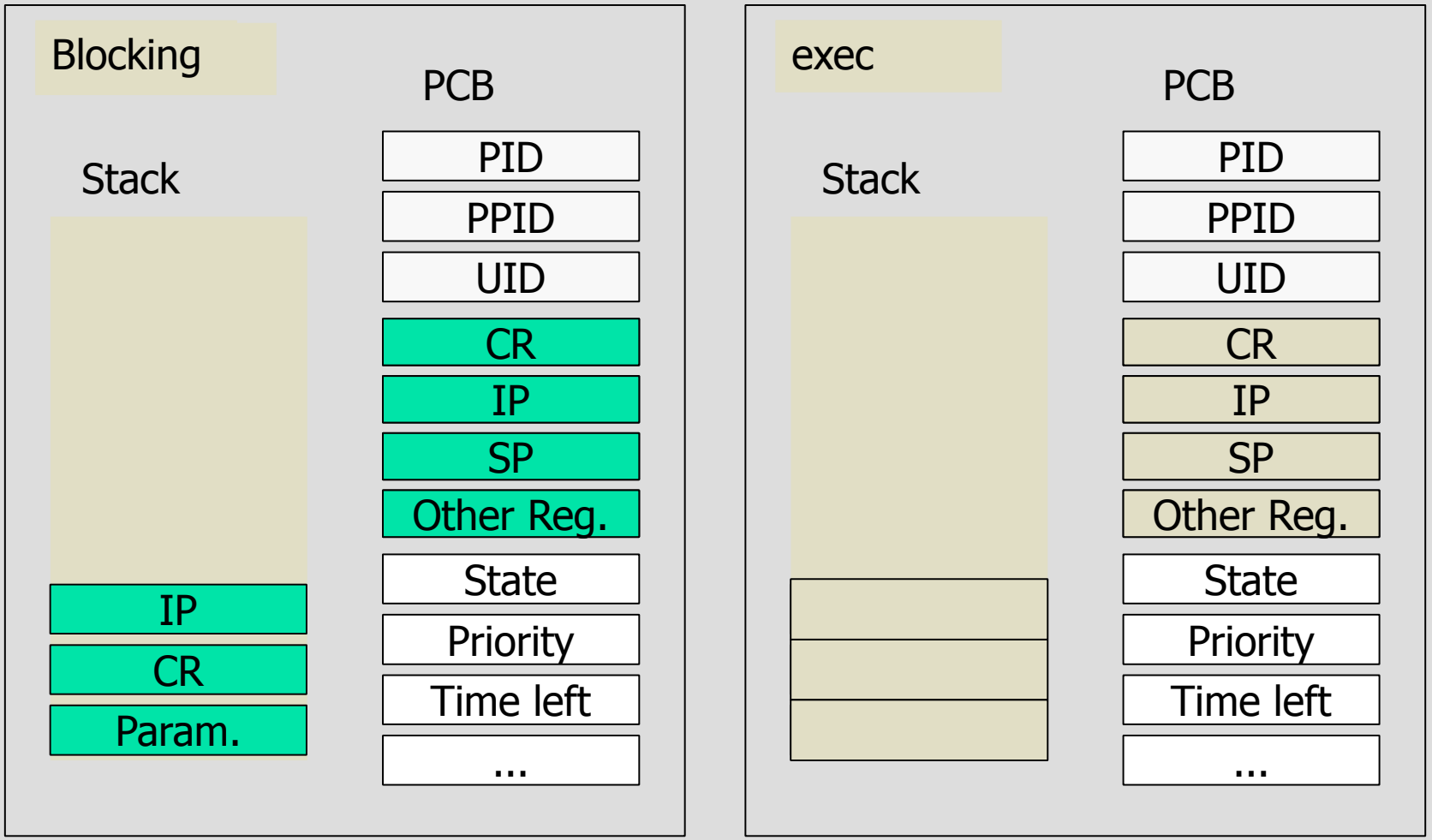
The “exec” pointer

- Every OS has one pointer (“exec”) to the PCB of the running process
  - The status of the “exec” process is RUNNING
- When a process switch occurs:
  - The status of the “exec” process is changed to BLOCKING or READY
  - The scheduler is called
  - The scheduler selects another “exec” from the ready queue

System call with process switch

- Saves parameters on stack
- Then:
  - Change to supervisor mode
  - Save context in the PCB of “exec” (including SP)
  - The process changes status and goes into BLOCKING mode
  - Calls the scheduler
    - Moves “exec” into the blocking queue
    - Selects another process to go into RUNNING mode
    - Now exec points to the *new* process
  - Restores the exec process
    - Restores the context of the “exec” PCB
      - Including SP (that means the stack is changed)
    - IRET
      - Returns to where the new process was interrupted

Process switches and stack changes

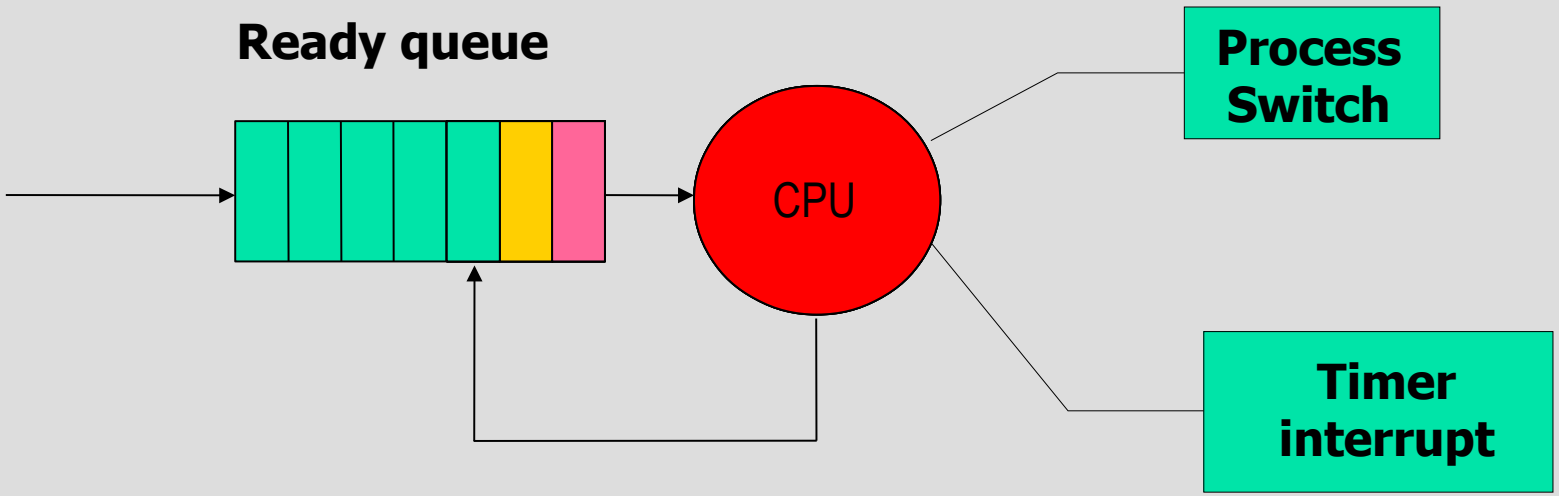


Process switch

- This is only an example
  - Every OS has little variations on the same theme
  - For example, in most cases, registers are saved on the stack, not on the PCB
- You can try and look into some real OS
  - Linux
  - Free BSD
  - every OS is different!

Time sharing systems

- In time sharing systems,
  - Every process can execute for at most one round
    - for example, 10msec
  - At the end of the round, the processor is given to another process



Interrupt with process switch

- When an interrupt arrives:
  - change to supervisor mode
  - save CR and IP
  - save processor context
  - execute the interrupt handler
  - call the scheduler
    - this may change the “exec” pointer
  - IRET

Causes for a process switch

- A process switch can be
  - Voluntary: the process calls a blocking primitive
    - For example, by calling a read() on a blocking device
  - Non-voluntary: an interrupt arrives that causes the process switch
    - It can be the timer interrupt in time-sharing systems
    - It can be an I/O device which unblocks a blocked process with a higher priority

Processes

Two aspects can be distinguished in a process:

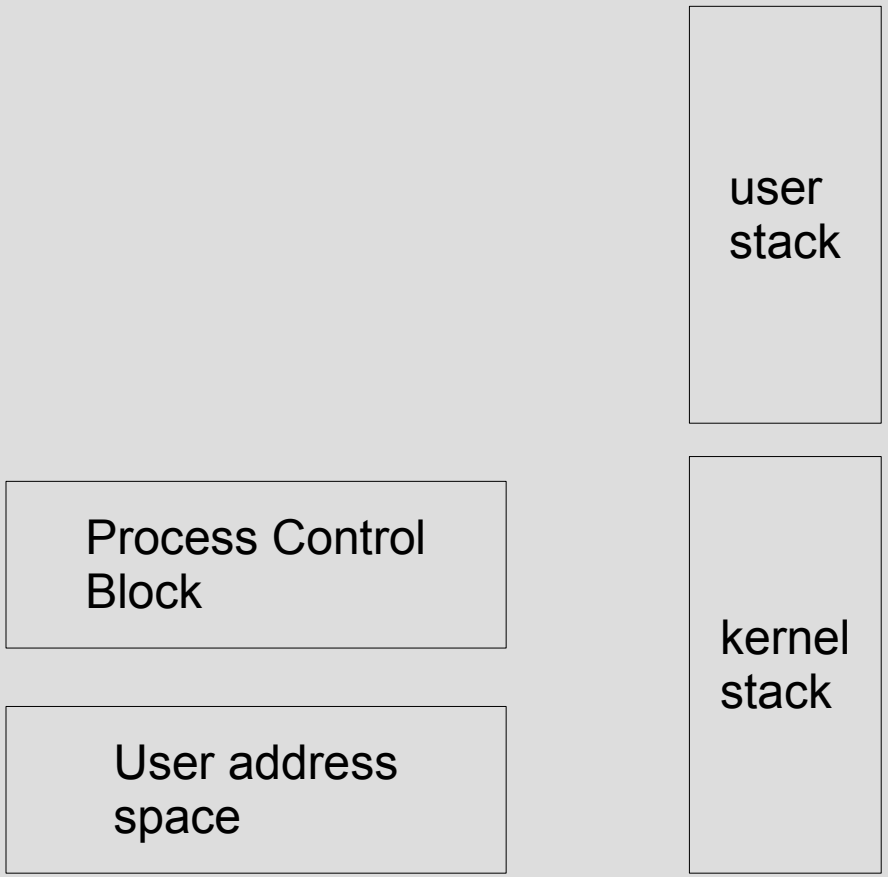
- Resource ownership
  - A process includes a virtual address space, a process image (code + data)
  - It is allocated a set of resources, like file descriptors, I/O channels, etc.
- Scheduling/execution
  - The execution of a process follows an execution path, and generates a trace (sequence of internal states)
  - It has a state (ready, running, etc.) and scheduling parameters (priority, time left in the round, etc.)

Multi-threading

- Many OS separate these aspects, by providing the concept of **thread**
- The process is the “resource owner”
- The thread is the “scheduling entity”
  - One process can consist of one or more *threads*
  - Threads are sometime called (improperly) lightweight processes
  - Therefore, one process can have many different (and concurrent) traces of execution!
  - Each thread belongs to exactly one process and no thread can exist outside a process

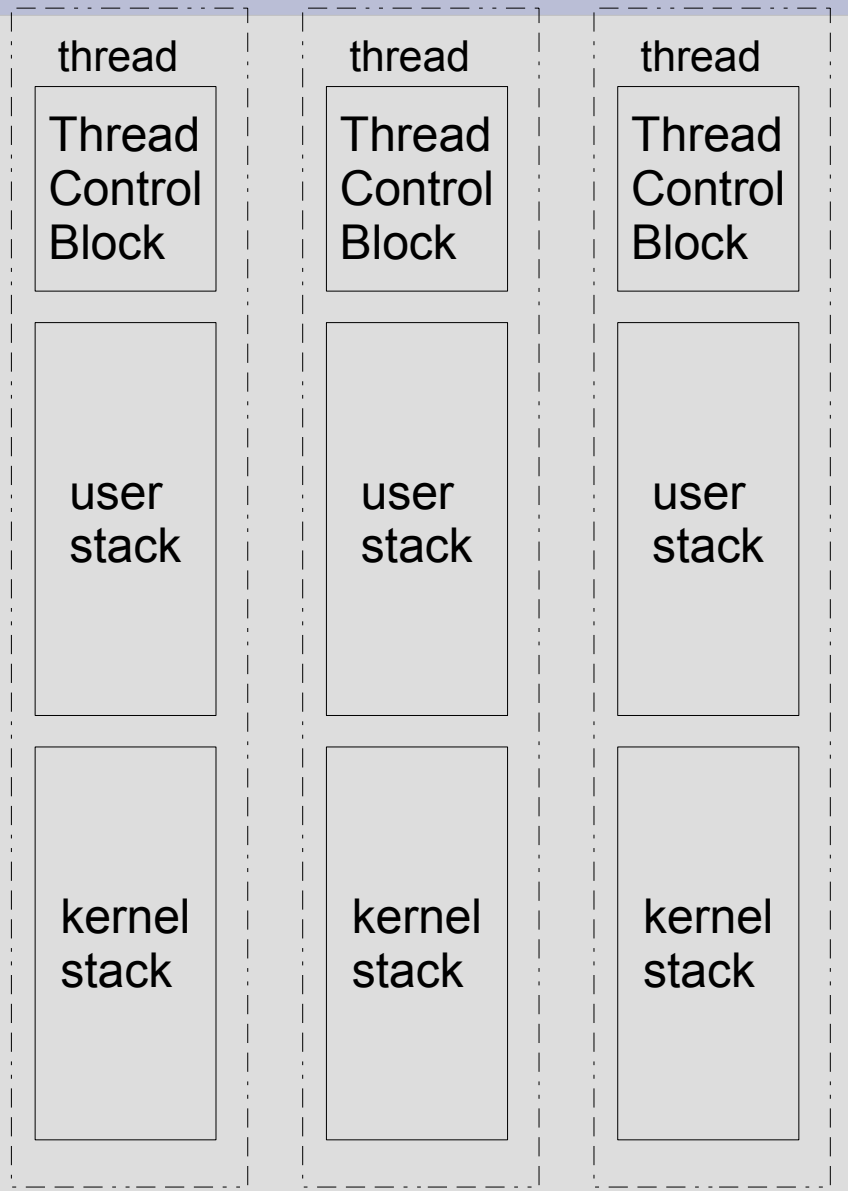
Single threaded Process Model

- In the single-threaded process model, one process has only one thread
  - one address space
  - one stack
  - one PCB only



Multi-threaded process model

- In the multi-threaded process model, each process can have many threads
  - One address space
  - One PCB but multiple TCBs (Thread Control blocks)
  - Many stacks
  - The threads are scheduled directly by the global scheduler



Process Control Block

User address space

Threads

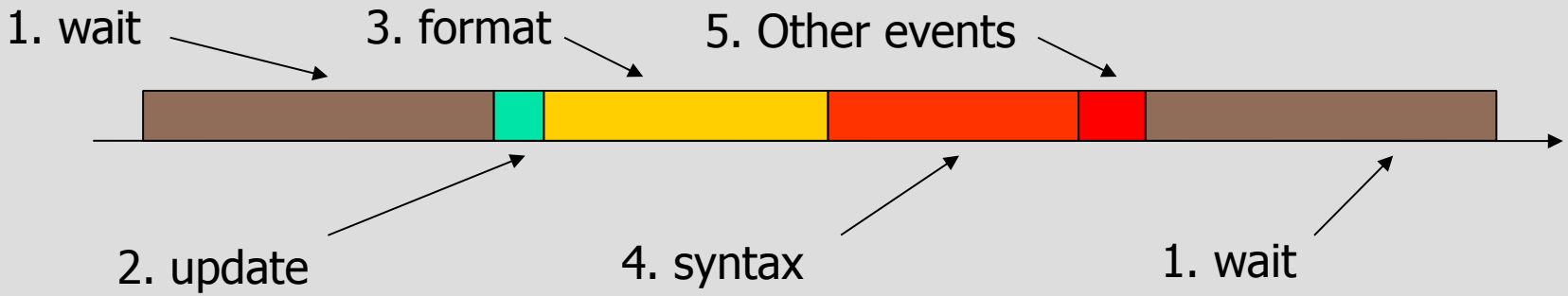
- Generally, processes do not share memory
  - To communicate between process, it is necessary to use OS primitives
  - Process switch is more complex because we have to change address space
- Two threads in the same process share the same address space
  - They can access the same variables in memory
  - Communication between threads is simpler
  - Thread switch has less overhead

Processes vs. Threads

- Processes are mainly used to compete for some resource
  - For example, two different users run two separate applications that need to print a file
  - The printer is a shared resource, the two processes compete for the printer
- Threads are mainly used to collaborate to some goal
  - For example, a complex calculation can be split into two parallel phases, where each thread does one phase
  - In a multi-processor machine the two threads go in parallel and the calculation becomes faster

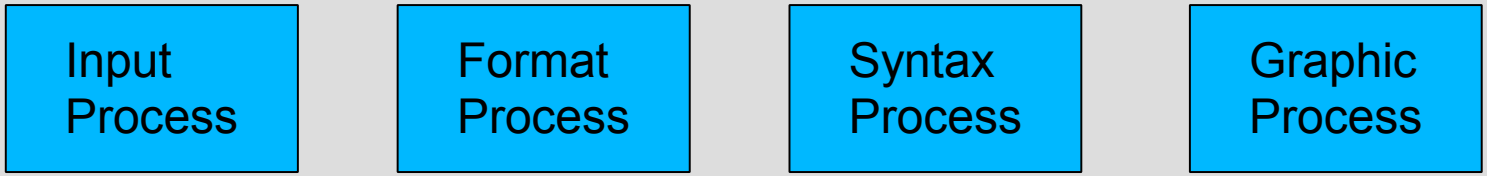
Example (1)

- Consider a Word Processor application
- Main cycle
  1. Wait for input from the keyboard
  2. Update the document
  3. Format the document
  4. Check for syntax errors
  5. Check for other events (i.e. temporary save)
  6. Return to 1
- One single process would be a waste of time!



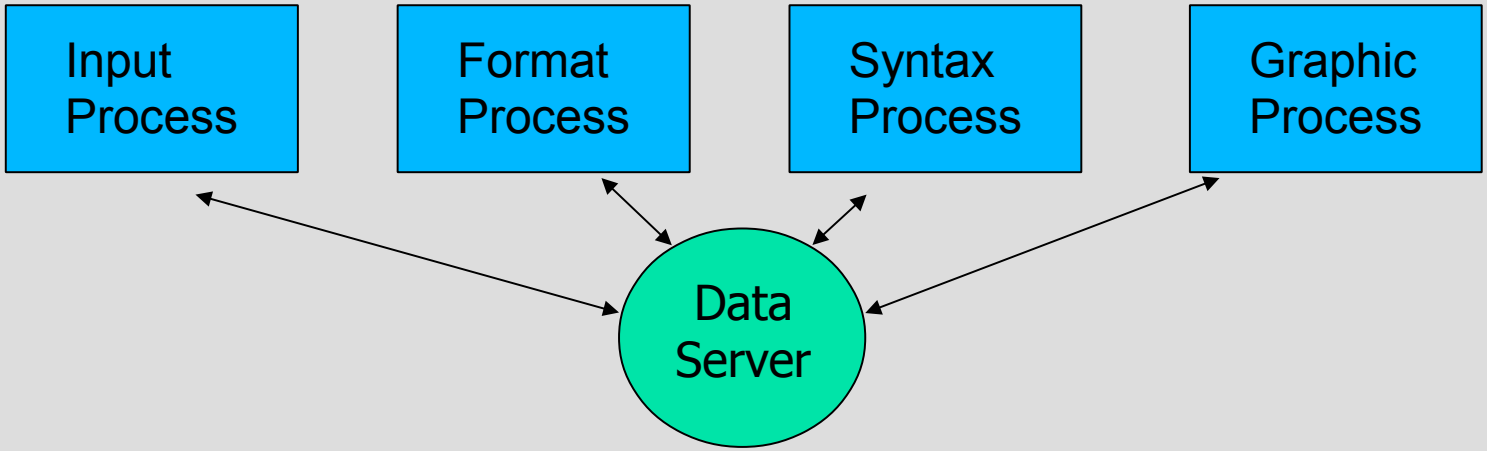
Example (2)

- Problems
  - Most of the time, the program waits for input
    - Idea: while waiting we could perform some other task
  - Activities 3 and 4 (formatting and syntax checking) are very time consuming
    - Idea: let's do them while waiting for input
- Solution with multiple processes
  - One process waits for input
  - Another process periodically formats the document
  - A third process periodically performs a syntax checking
  - A fourth process visualizes the document



Example (3)

- Problem with multiple processes
  - All processes need to access the same data structure: the document
  - Which process holds the data structure?
- Solution 1: message passing
  - A dedicated process holds the data, all the others communicate with it to read/update the data
  - Very inefficient!

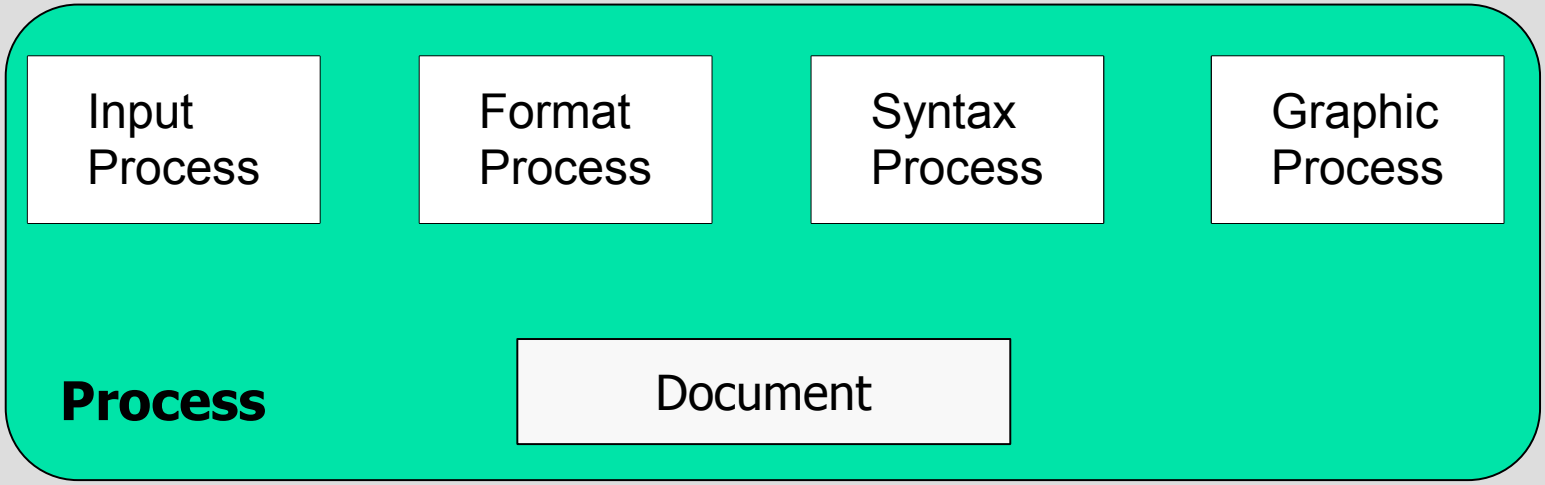


Example (4)

- Solution 2: shared memory
  - One process holds the data and makes that part of its memory shareable with the others
  - Still not very efficient:
    - Many process switches
    - Memory handling becomes very complex

Why using threads

- Speed of creation
  - Creating a thread takes far less time than a process
- Speed of switching
  - Thread switch is faster than process switch
- Shared memory
  - Threads of the same process run in same memory space
  - They can naturally access the same data!



Threads support in OS

- Different OS implement threads in different ways
  - Some OS supports directly only processes
    - Threads are implemented as “special processes”
  - Some OS supports only threads
    - Processes are threads’ groups
  - Some OS natively supports both concepts
    - for example Windows NT
- We will come back to this part later, after we have studied the problem of synchronization
- For now, we abstract away the different implementations

Summary

- Important concepts
  - Process: provides the abstraction of memory space
  - Thread: provide the abstraction of execution trace
  - The scheduler manages threads!
- Processes do not normally share memory
- Two threads of the same process share memory
- Threads may communicate in different ways:
  - Shared memory
  - Message passing
- In the following, we will only refer to threads