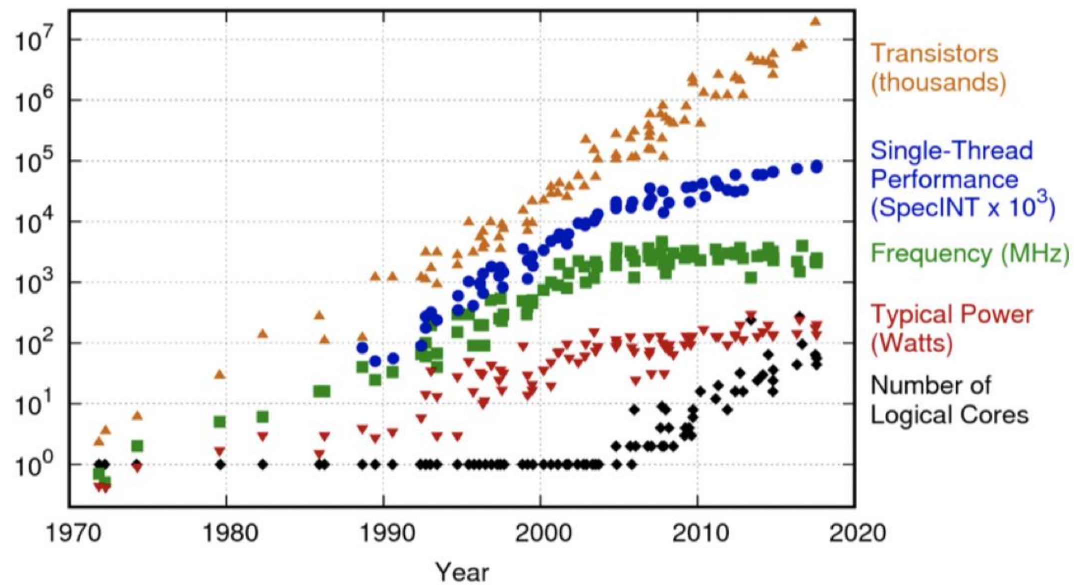
Operating Systems

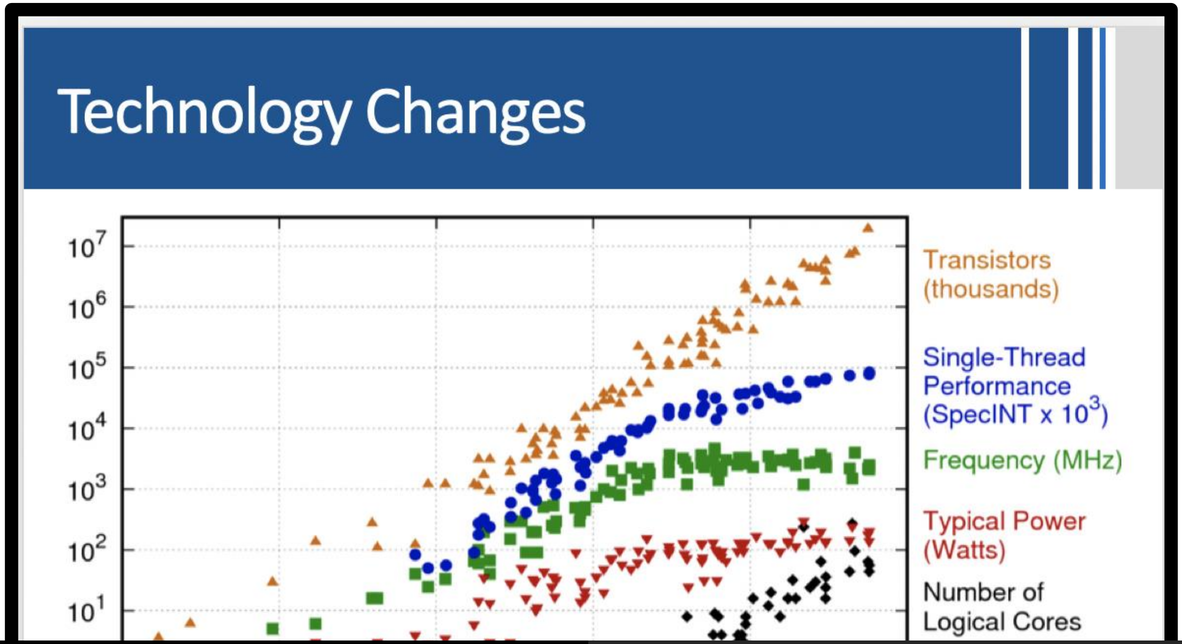
David Hay

Technology Changes



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2017 by K. Rupp

24

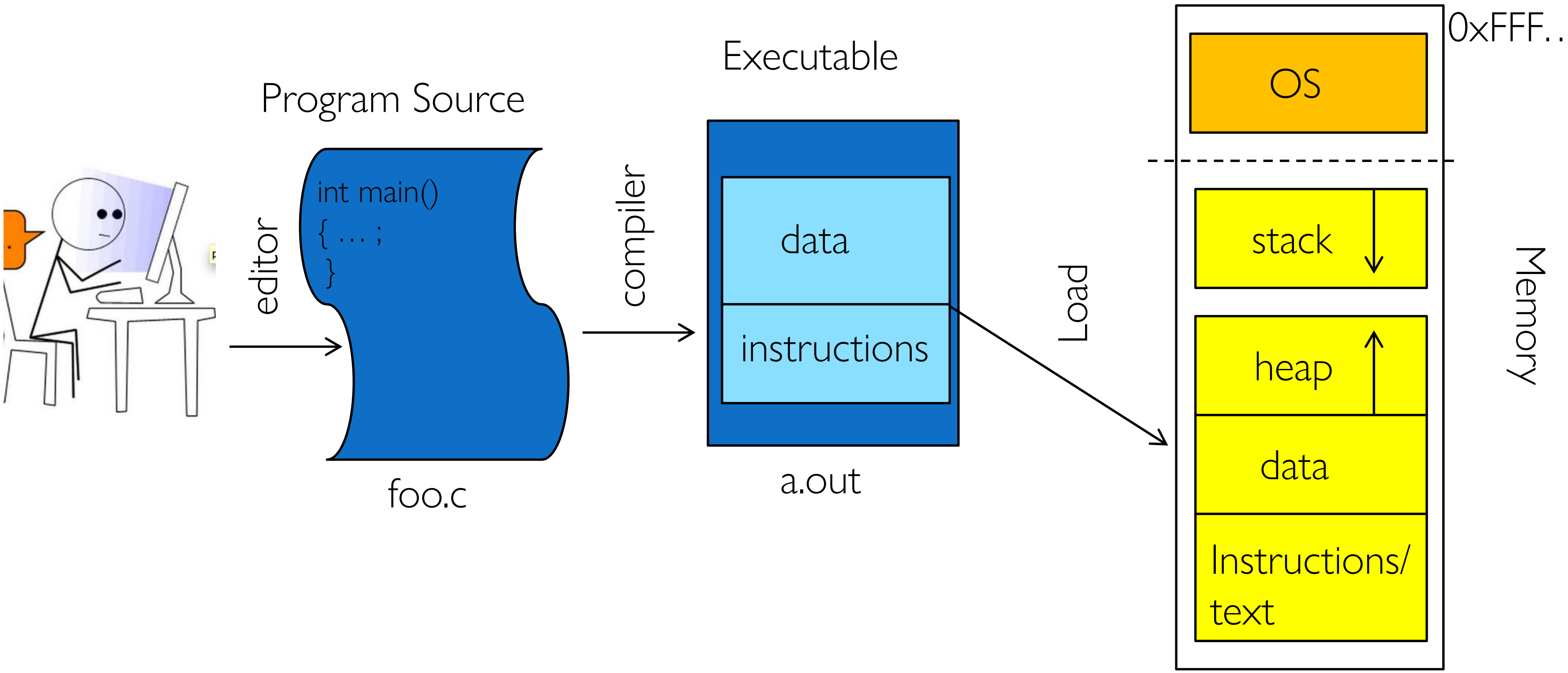


Gordon Moore, Intel Co-Founder, Dies at 94

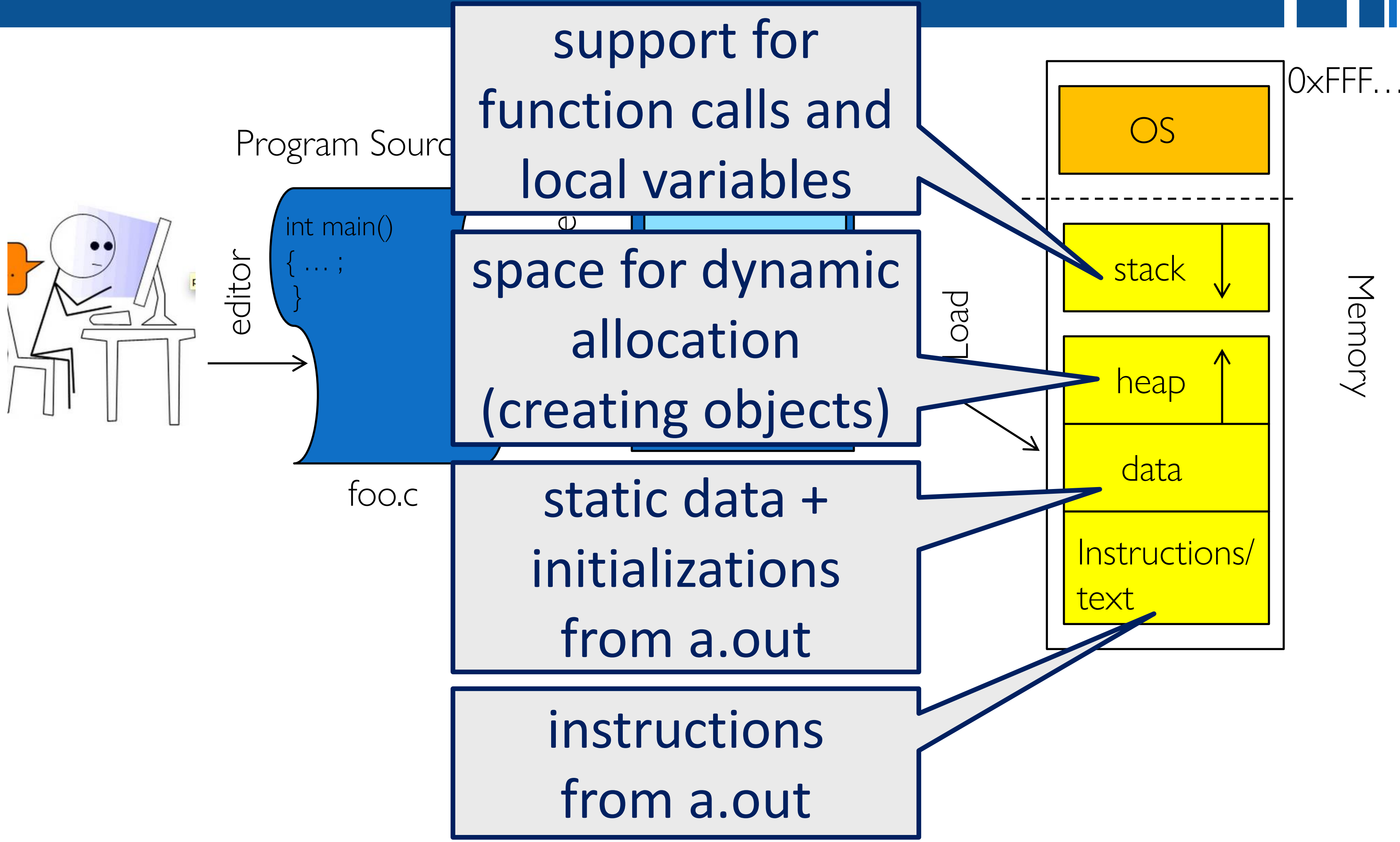
Moore, who set the course for the future of the semiconductor industry, devoted his later years to philanthropy.

NEXT SUBJECT: PROCESS MANAGEMENT

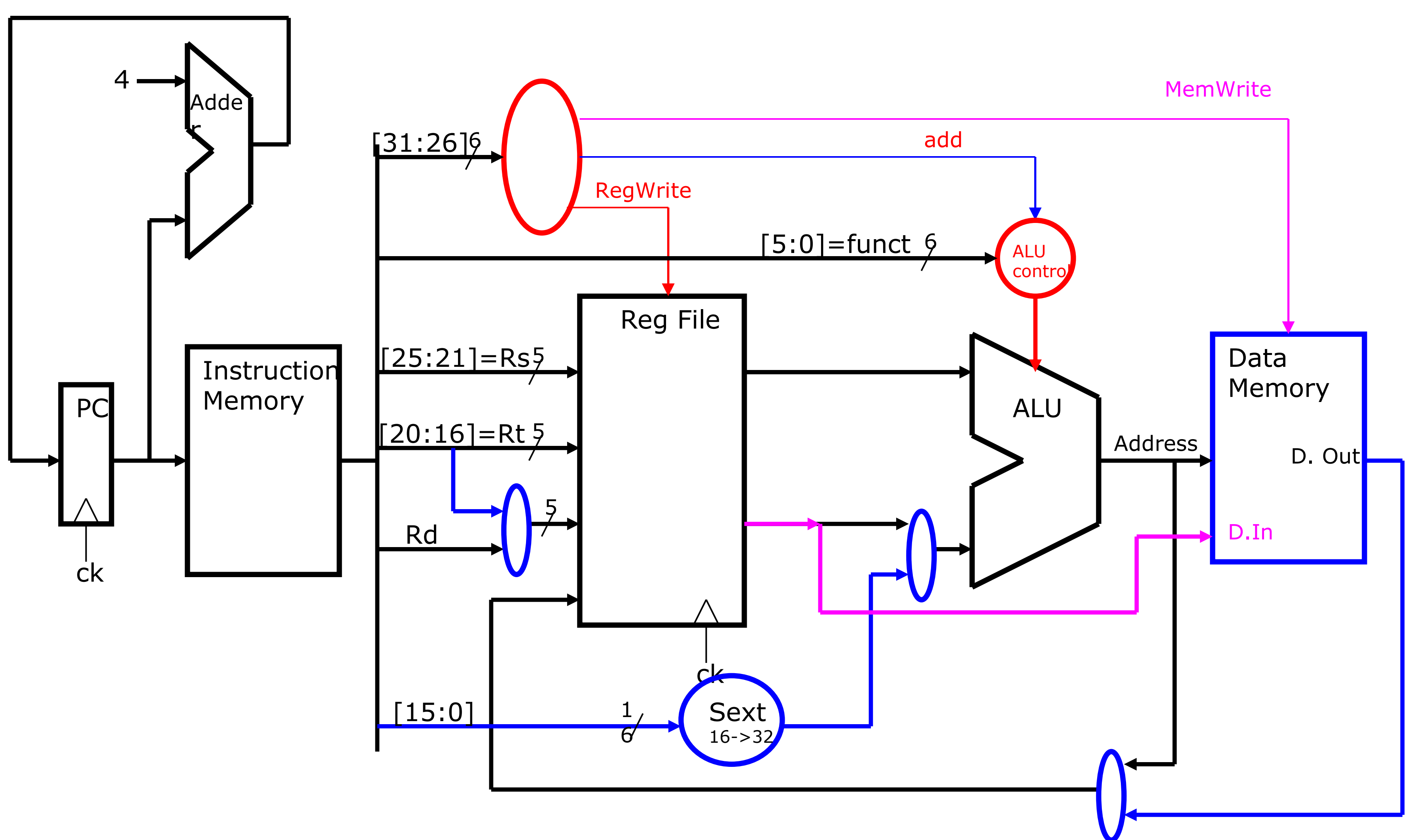
How does OS Run Programs?



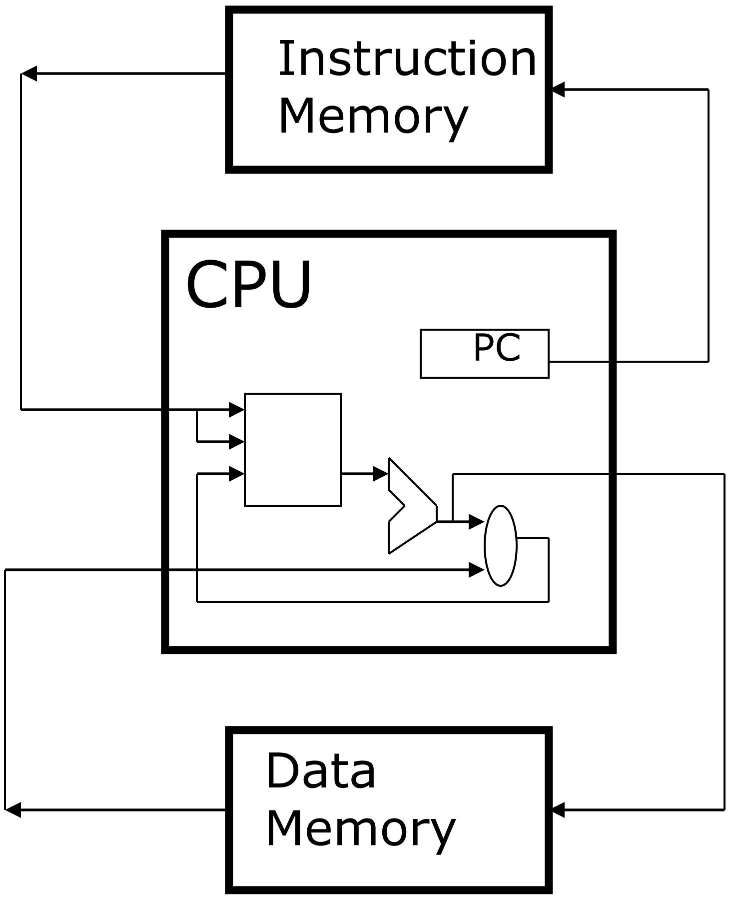
How does OS Run Programs?



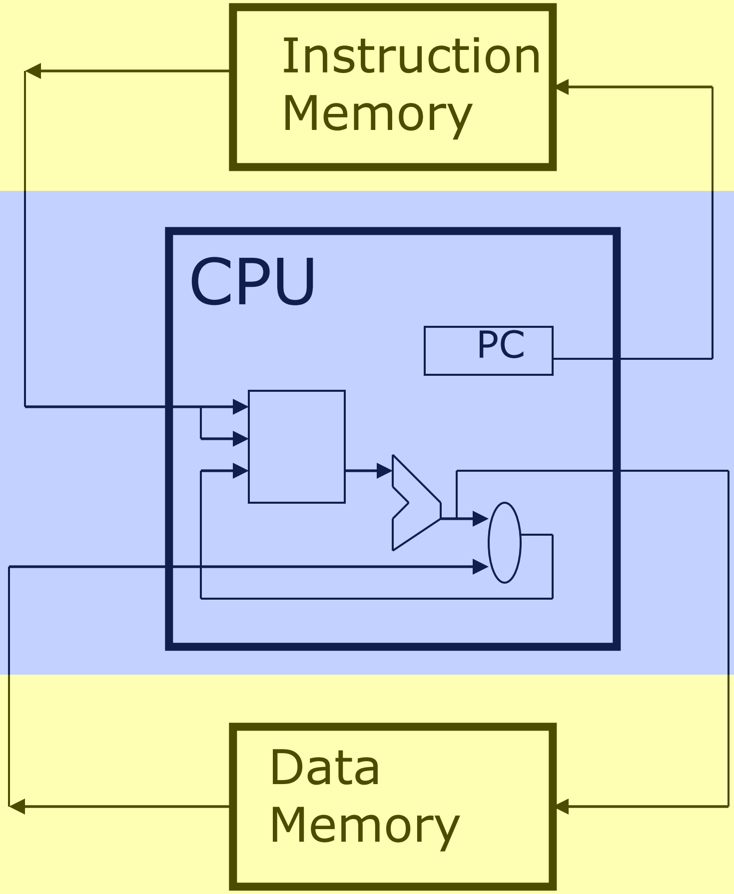
A CPU capable of R-type & lw/sw instructions



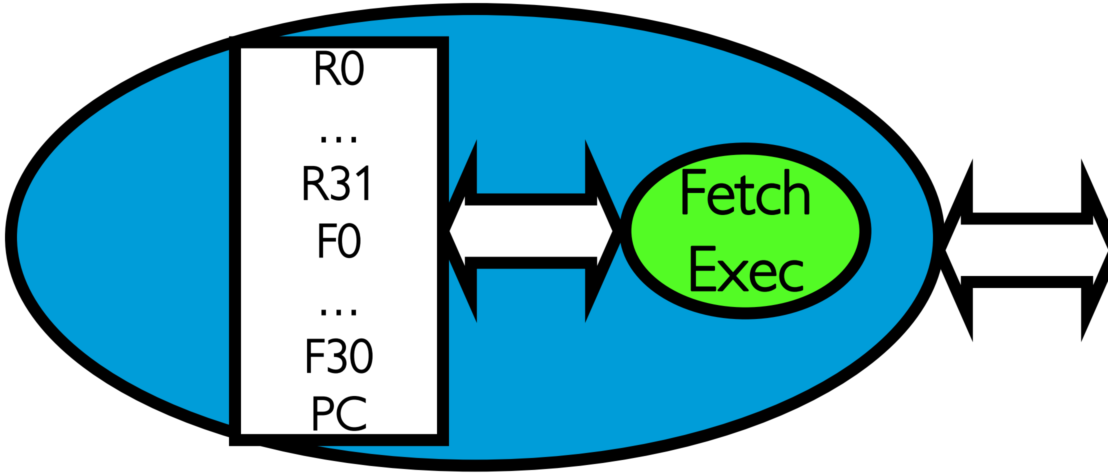
Where is the CPU?



Where is the CPU?



What happens during program execution?



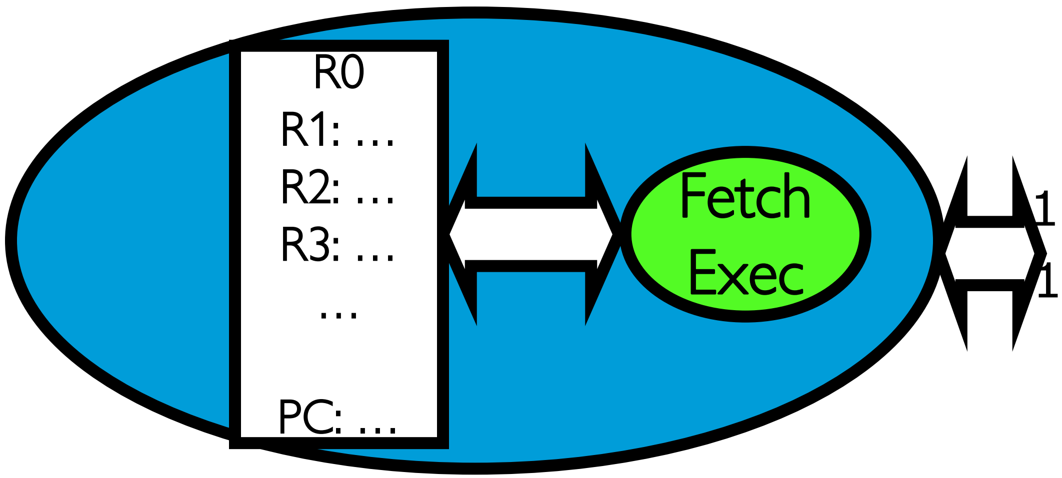
- Execution sequence:
  - Fetch Instruction at PC and save it in IR
  - Decode instruction in IR
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

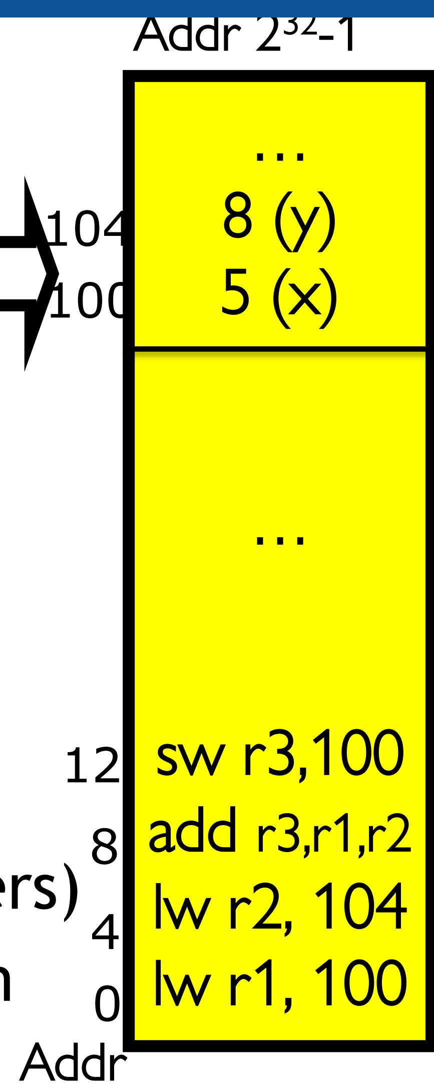
Example: x=x+y

Suppose x is in location 100, y is in location 104 in the data memory

```
lw r1, 100
lw r2, 104
add r3, r1, r2
sw r3,100
```

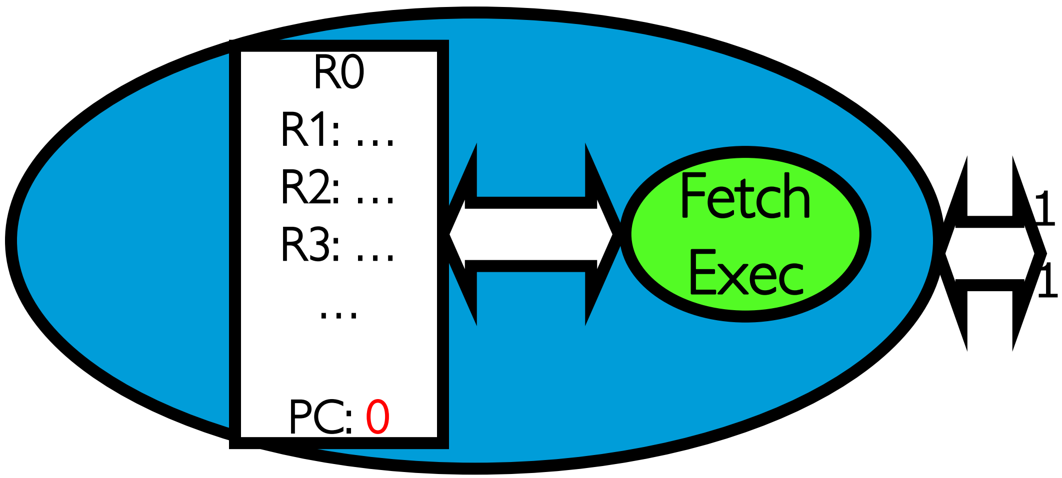
Example: Suppose x=5, y=8

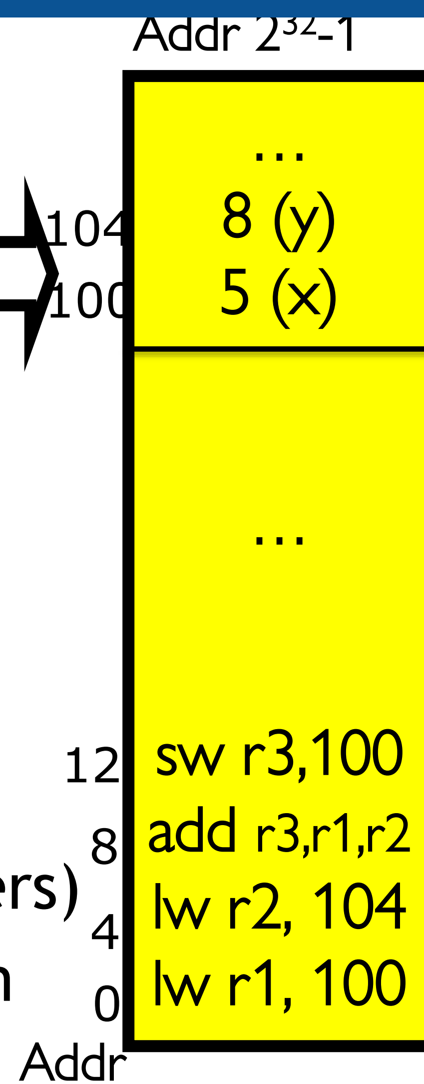




- Execution sequence:
  - Fetch Instruction at PC save it in IR
  - Decode instruction in IR
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

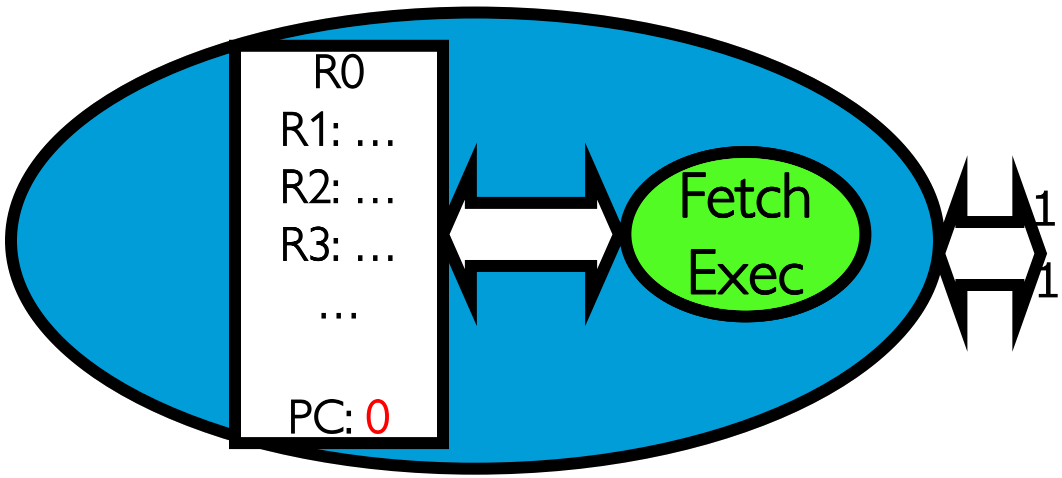
Example: Suppose x=5, y=8

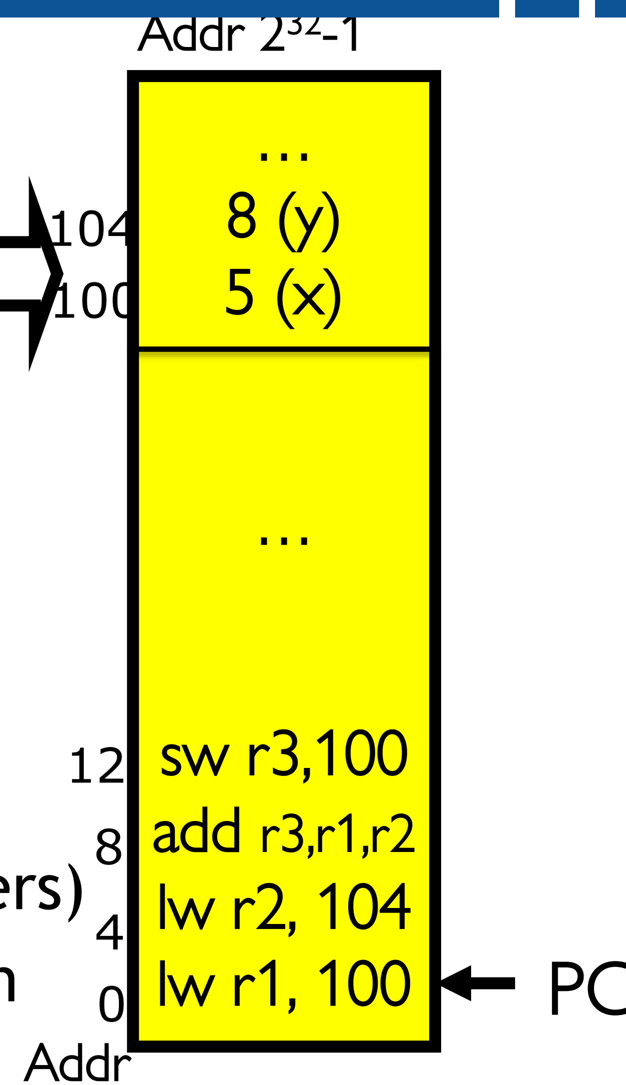




- Execution sequence:
  - Fetch Instruction at PC save it in IR
  - Decode instruction in IR
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

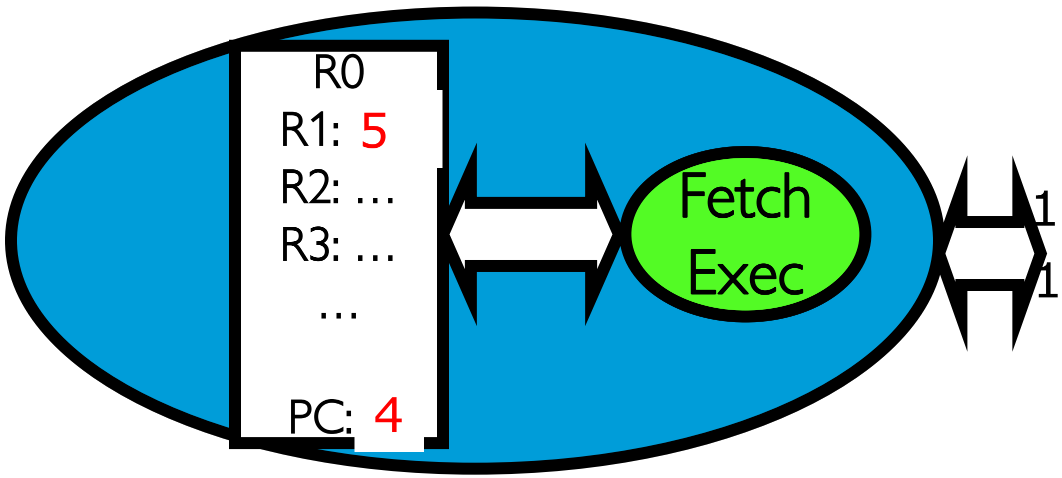
Example: Suppose x=5, y=8

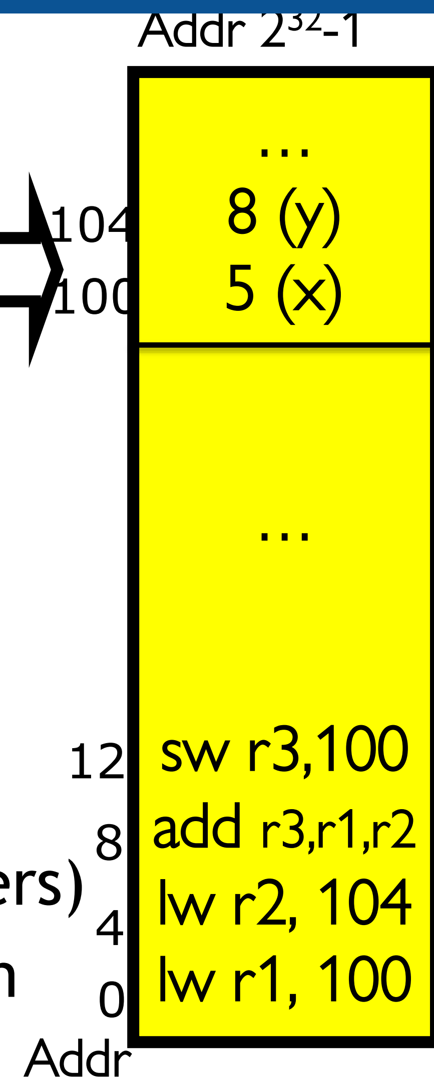




- Execution sequence:
  - Fetch Instruction at PC save it in IR
  - Decode instruction in IR
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

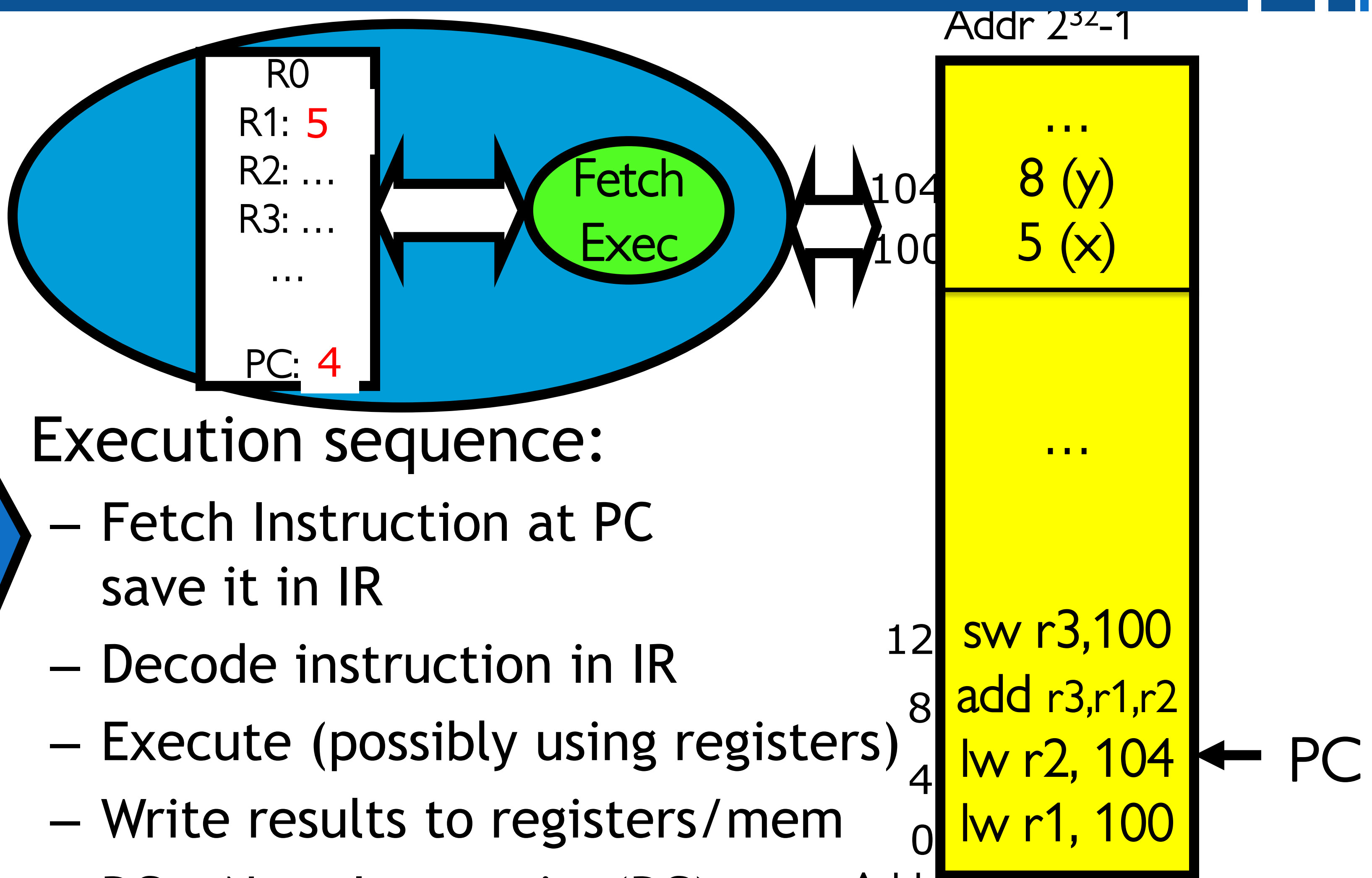
Example: Suppose x=5, y=8





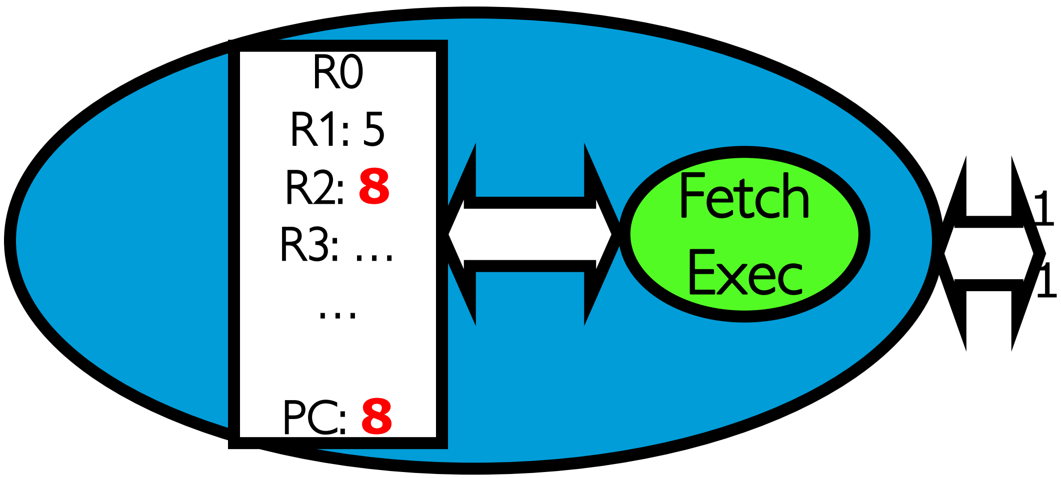
- Execution sequence:
  - Fetch Instruction at PC save it in IR
  - Decode instruction in IR
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

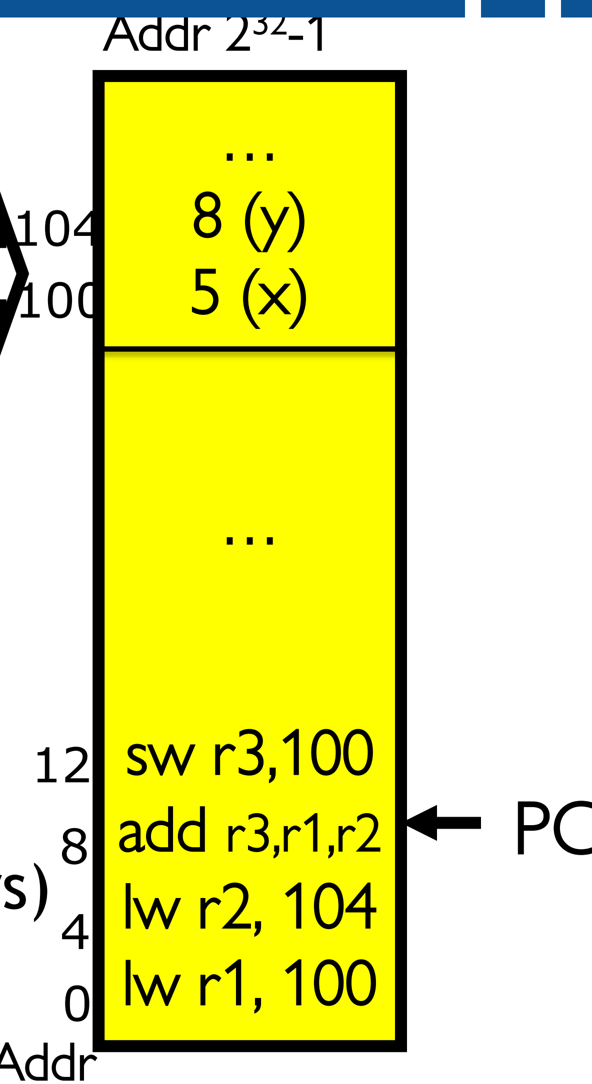
Example: Suppose x=5, y=8



- Execution sequence:
  - Fetch Instruction at PC save it in IR
  - Decode instruction in IR
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

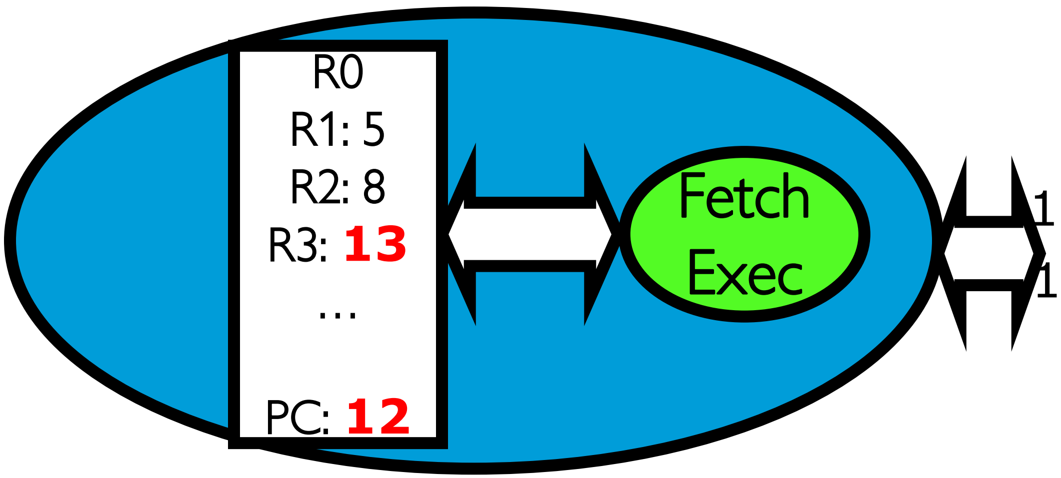
Example: Suppose x=5, y=8

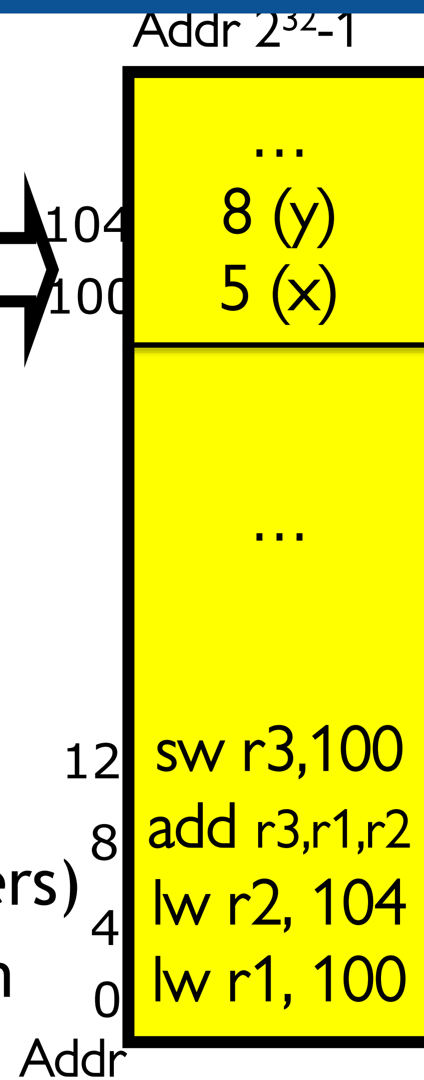




- Execution sequence:
  - Fetch Instruction at PC save it in IR
  - Decode instruction in IR
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

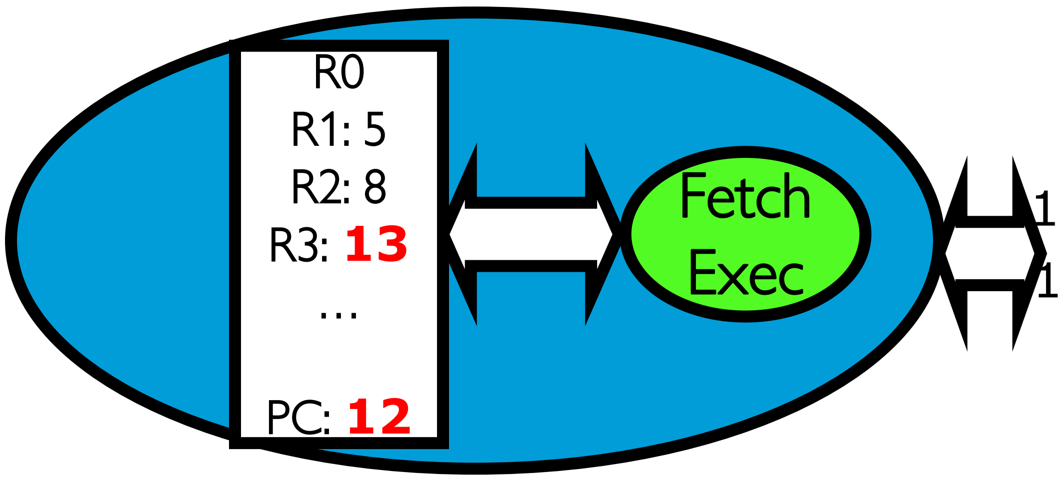
Example: Suppose x=5, y=8

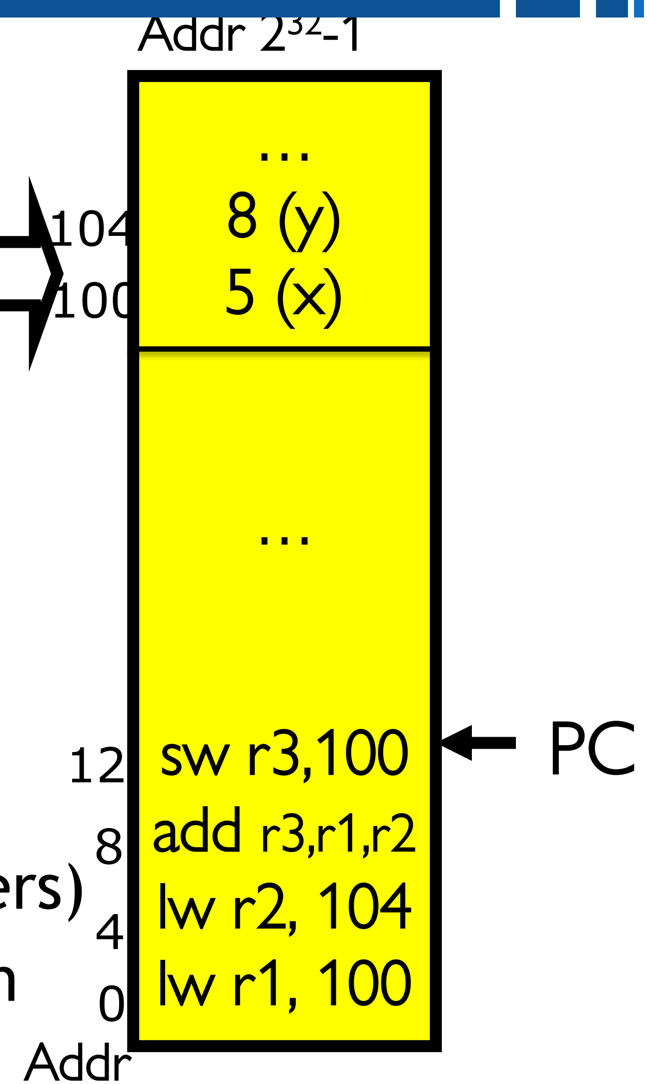




- Execution sequence:
  - Fetch Instruction at PC
    save it in IR
  - Decode instruction in IR
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

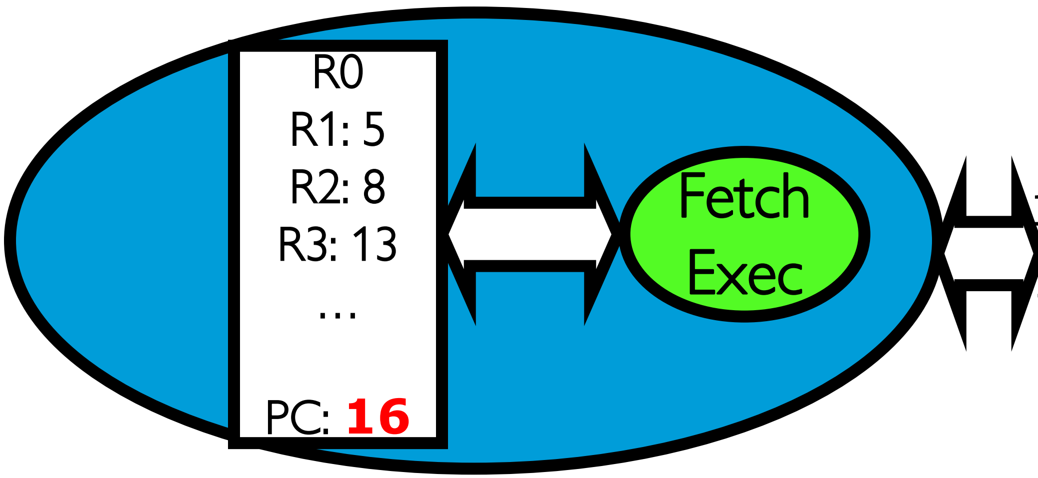
Example: Suppose x=5, y=8

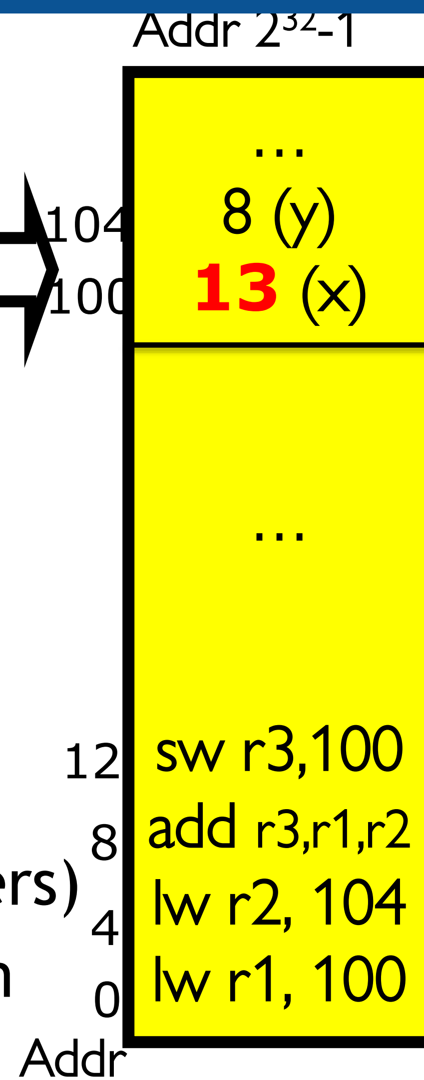




- Execution sequence:
  - Fetch Instruction at PC save it in IR
  - Decode instruction in IR
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

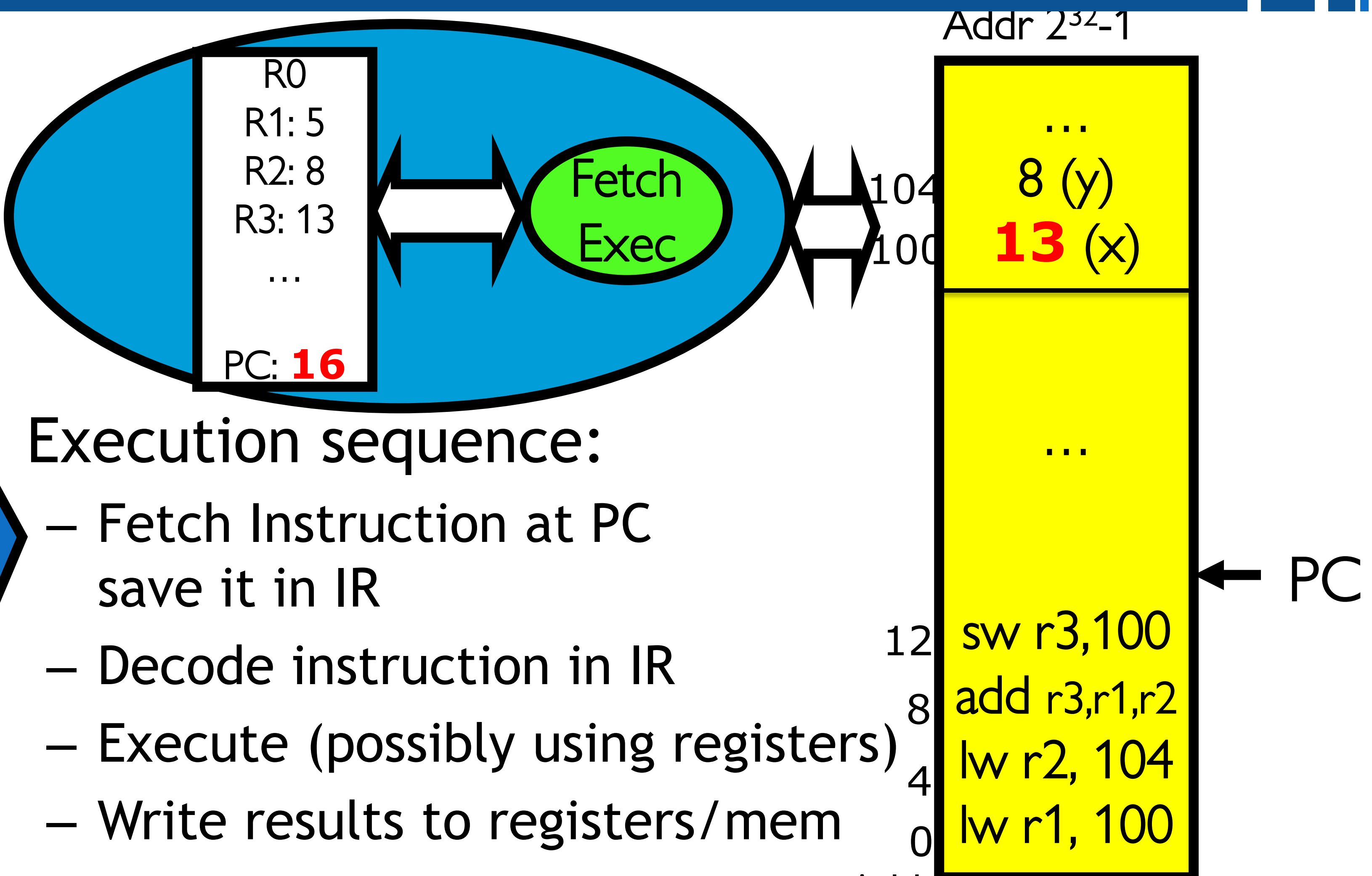
Example: Suppose x=5, y=8





- Execution sequence:
  - Fetch Instruction at PC save it in IR
  - Decode instruction in IR
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

Example: Suppose x=5, y=8



- Execution sequence:
  - Fetch Instruction at PC save it in IR
  - Decode instruction in IR
  - Execute (possibly using registers)
  - Write results to registers/mem
  - PC = Next Instruction(PC)
  - Repeat

Context (of Execution)

- The value of the registers:
  - Program Counter
  - All other registers in the CPU
  - There are other “special” registers in the CPU
    - Stack pointer, heap pointer, etc.
    - Generally points only to the “root” of the data
- What’s written in the memory:
  - Code/text, Data, Stack, Heap
- Some Internal data in the OS relevant to the program
  - E.g., user, priority, etc.

Context (of Execution)

- The value of the registers:
  - Program Counter
  - All other registers in the CPU
  - There are other "special" registers in the CPU
    - Stack pointer, heap pointer, etc.
    - Generally points only to the "root" of the data

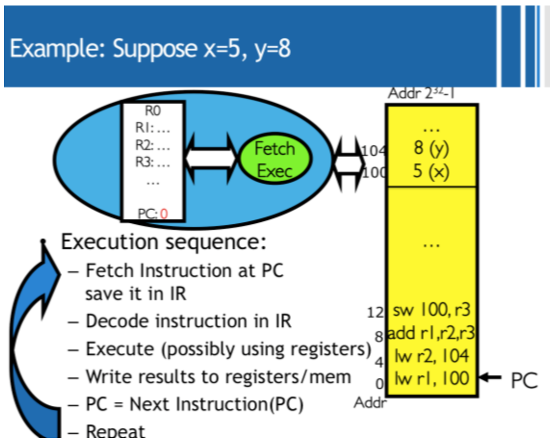
In CPU

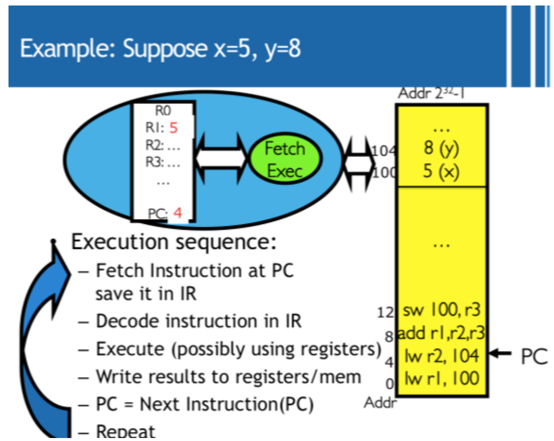
- What's written in the memory:
  - Code/text, Data, Stack, Heap

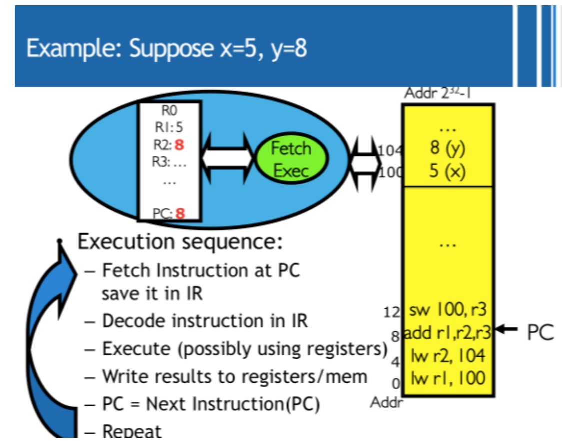
In Memory

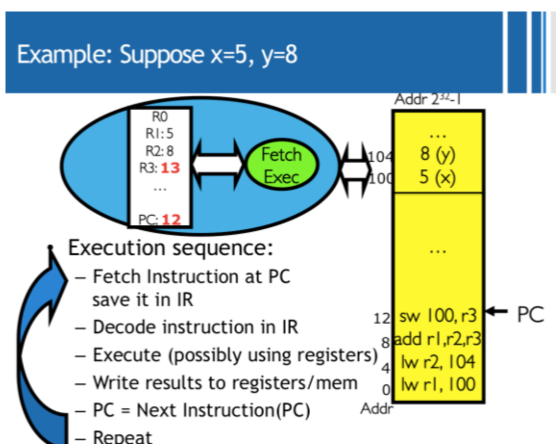
- Some Internal data in the OS relevant to the program
  - E.g., user, priority, etc.

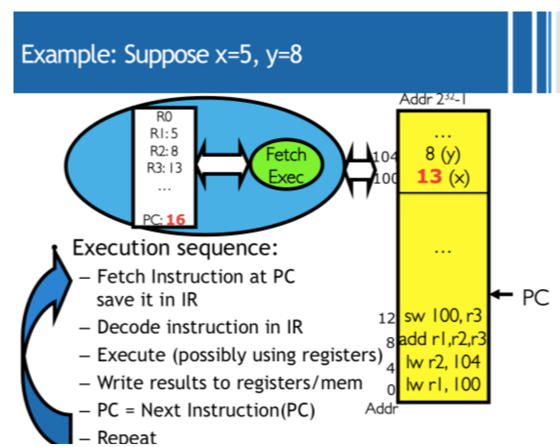
Different Contexts in the Example











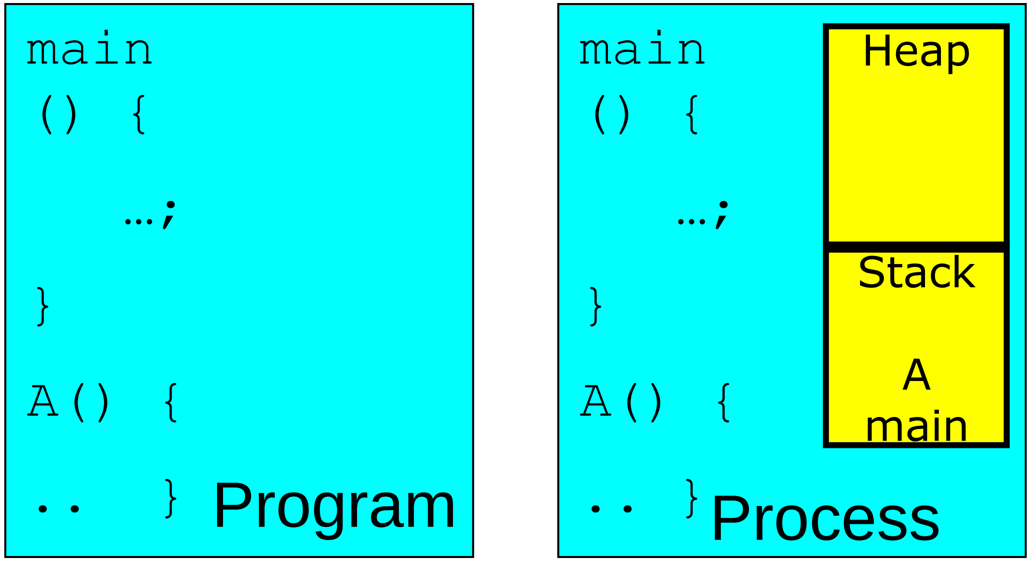
Processes תהליכים

- **A process is an instance of a program execution**
  - an abstraction of “an individual computer”
- The processes are OS objects
  - Processes provide context of execution
  - The process exists for the duration of the execution
  - Also called “job”
- A process is executing on a CPU when it is resident in the CPU registers.
  - For non-resident processes, the context part that was “in CPU” is stored in a special location in memory

Process vs. Program

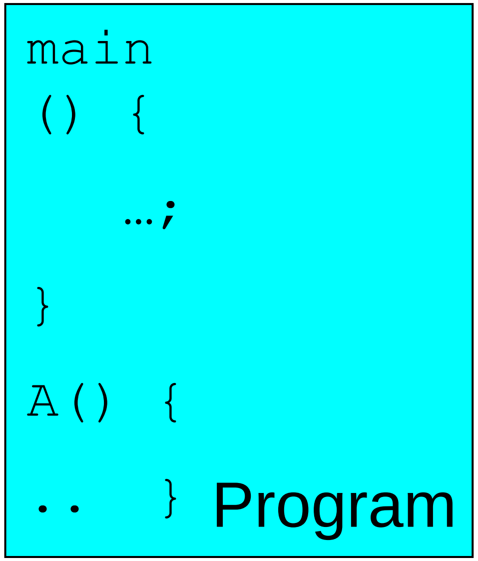
Process vs. Program

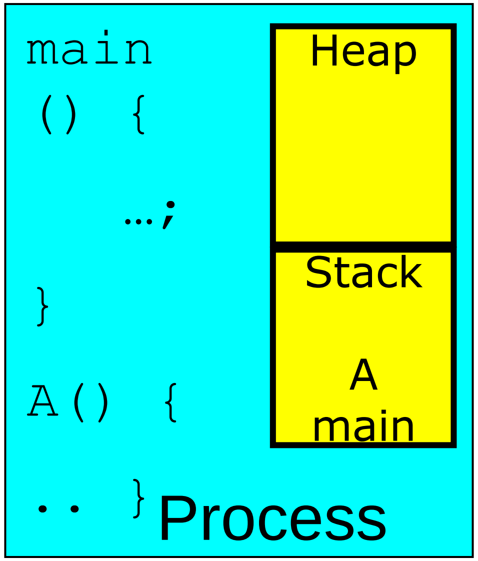
- Program is passive entity, process is active
  - Program becomes process when executable file loaded into memory
  - Done by the **loader**, which is a component of the OS



Process vs. Program

- Program is passive entity, process is active
  - Program becomes process when executable file loaded into memory
  - Done by the **loader**, which is a component of the OS

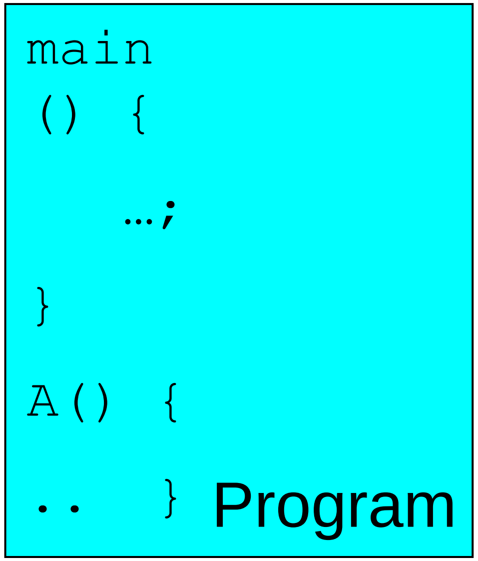


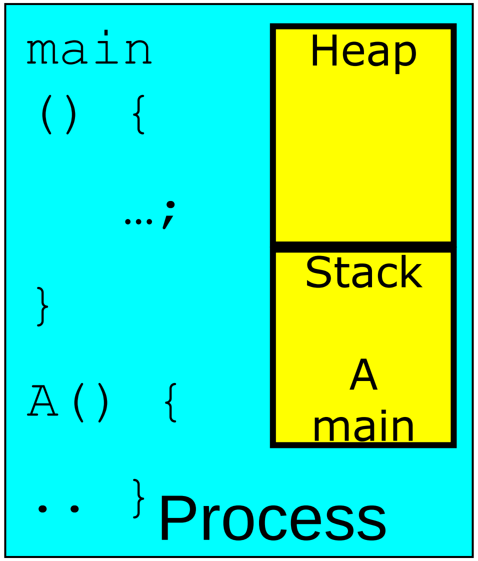


- More to a process than just a program:
  - Program is just part of the process state

Process vs. Program

- Program is passive entity, process is active
  - Program becomes process when executable file loaded into memory
  - Done by the **loader**, which is a component of the OS

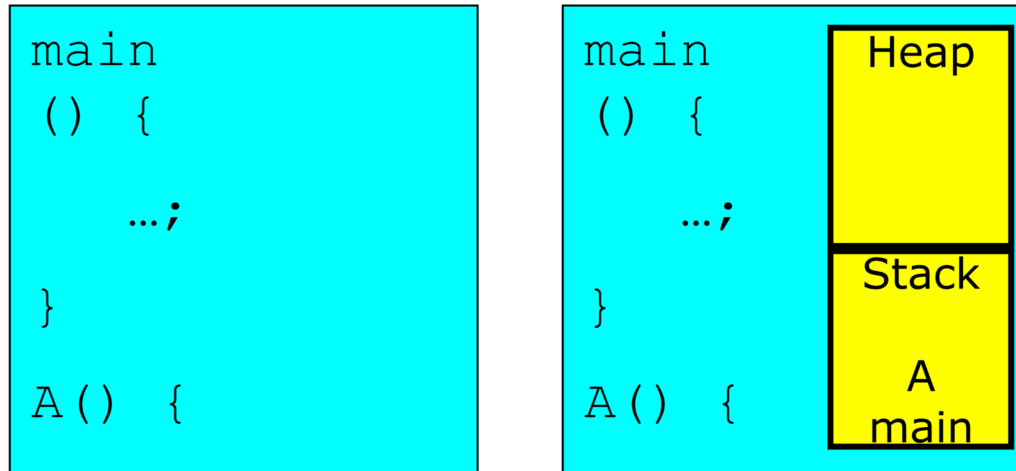




- More to a process than just a program:
  - Program is just part of the process state
- Less to a process than a program:
  - A program can invoke more than one process

Process vs. Program

- Program is passive entity, process is active
  - Program becomes process when executable file loaded into memory
  - Done by the **loader**, which is a component of the OS



- More to a proce
  - Program is ju
- Less to a proces
  - A program ca

Baking Analogy	
Program	Recipe
CPU/Processor	Baker
Data	Ingredients
Process	Baking the cake

Address Space of Process (Simplified)

Address Space of Process (Simplified)

- Address space $\Rightarrow$ the set of accessible addresses in the **memory** by the process

Address Space of Process (Simplified)

- Address space $\Rightarrow$ the set of accessible addresses in the **memory** by the process
- Given by two parameters: ***base*** and ***bound***. This imply that the process can access addresses only in [***base***,***base***+***bound***]. Verified by the CPU in hardware.
  - Is PC in [***base***,***base***+***bound***]?
  - **Protection/Isolation:** processes cannot access each other memory

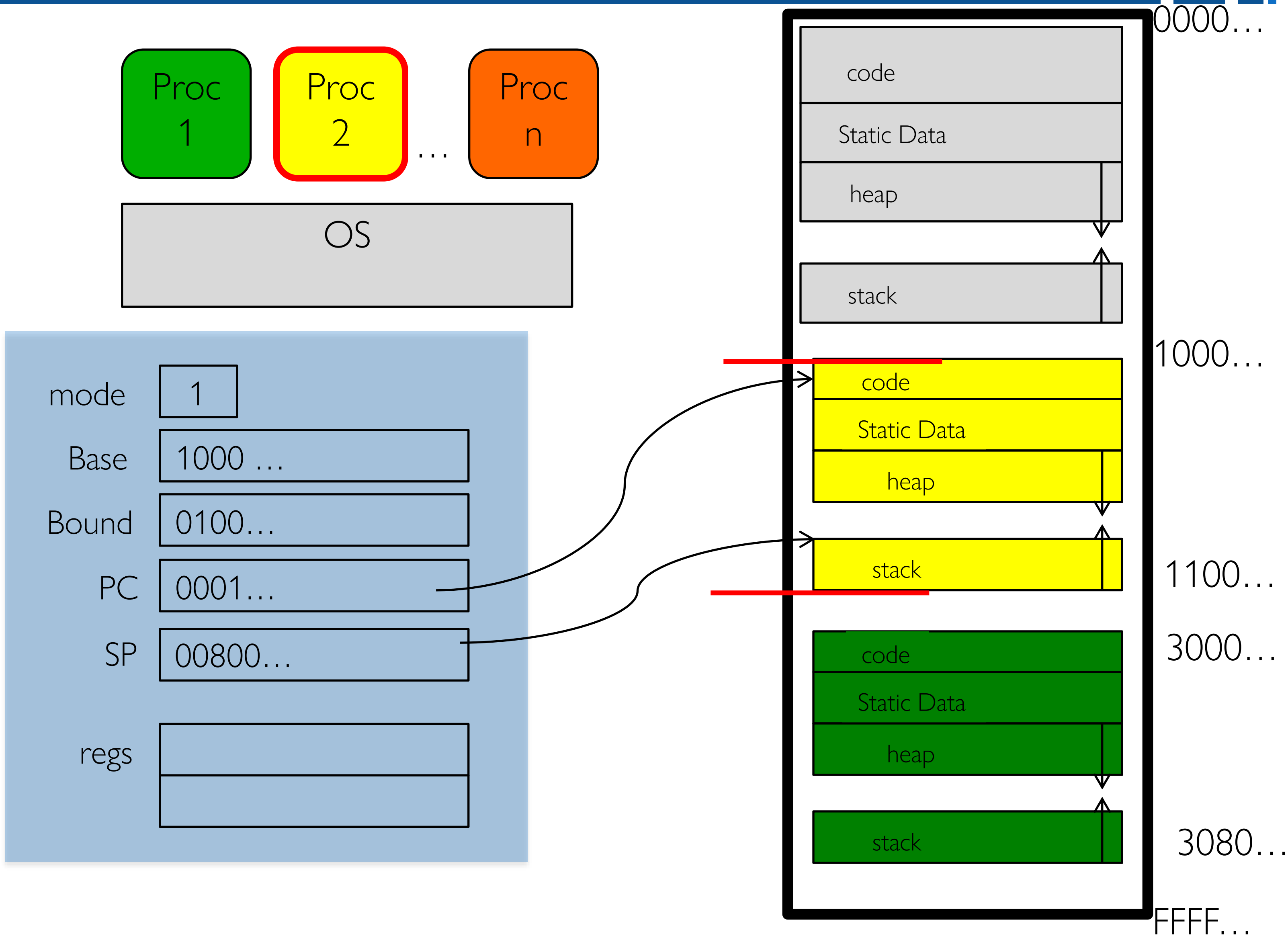
Address Space of Process (Simplified)

- Address space $\Rightarrow$ the set of accessible addresses in the **memory** by the process
- Given by two parameters: ***base*** and ***bound***. This imply that the process can access addresses only in [***base***,***base+bound***]. Verified by the CPU in hardware.
  - Is PC in [***base***,***base+bound***]?
  - **Protection/Isolation:** processes cannot access each other memory
- Address translation: when a process specifies address ***x*** it actually means ***base+x***
  - Same pointer address in different processes point to different memory

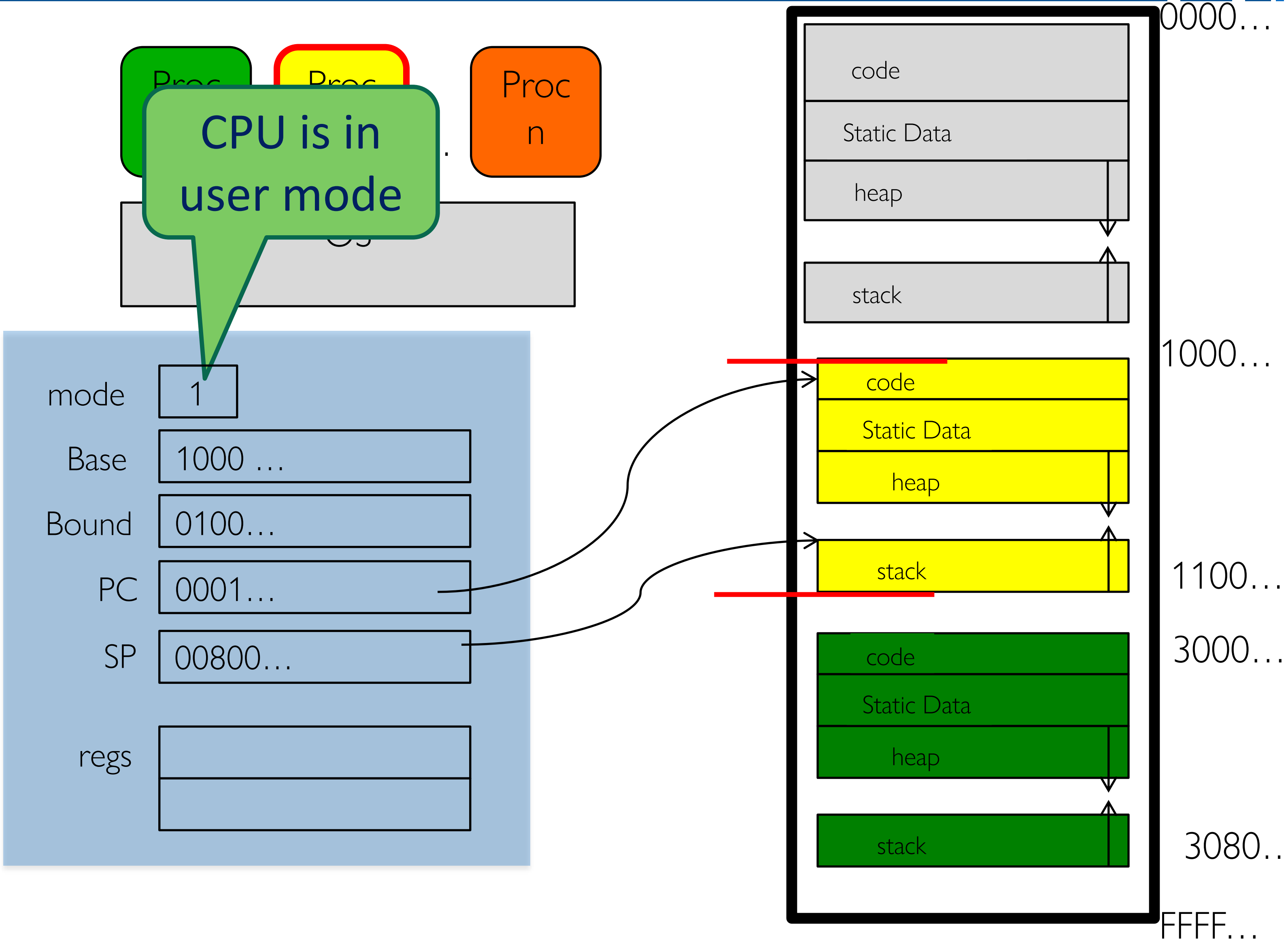
Address Space of Process (Simplified)

- Address space $\Rightarrow$ the set of accessible addresses in the **memory** by the process
- Given by two parameters: ***base*** and ***bound***. This imply that the process can access addresses only in [***base***,***base***+***bound***]. Verified by the CPU in hardware.
  - Is PC in [***base***,***base***+***bound***]?
  - **Protection/Isolation:** processes cannot access each other memory
- Address translation: when a process specifies address ***x*** it actually means ***base***+***x***
  - Same pointer address in different processes point to different memory
- OS address space is called kernel memory, the rest is user memory

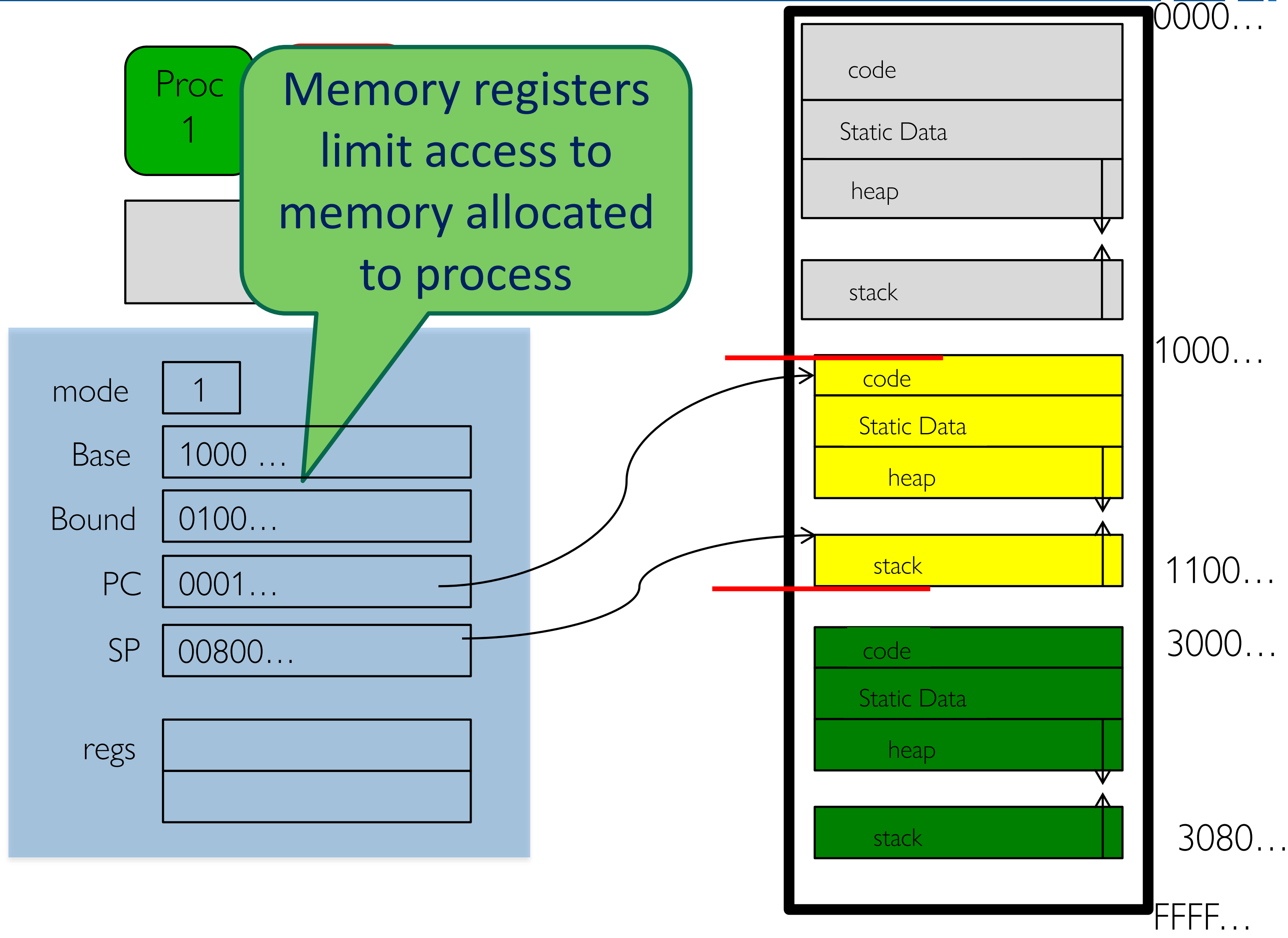
Multiple Processes



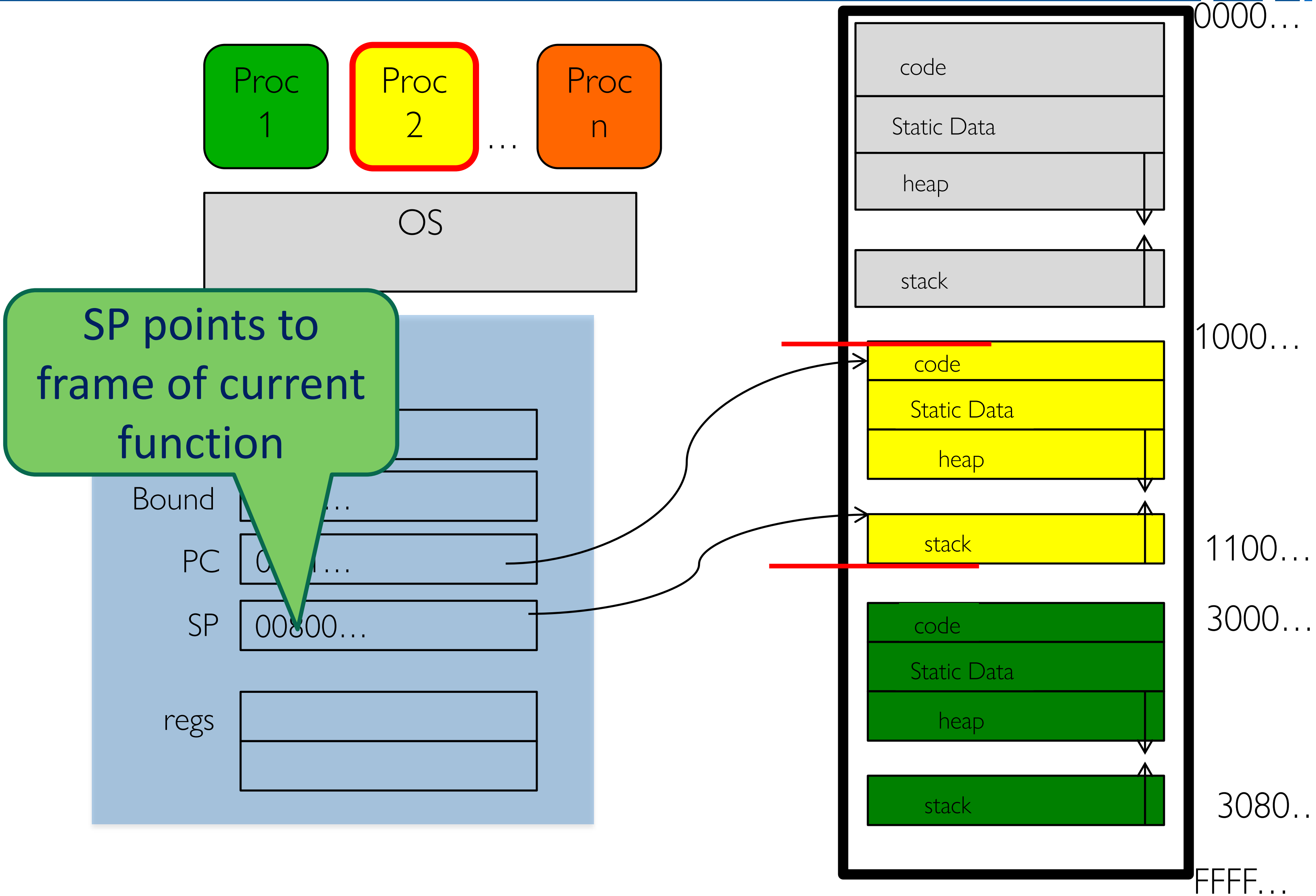
Multiple Processes



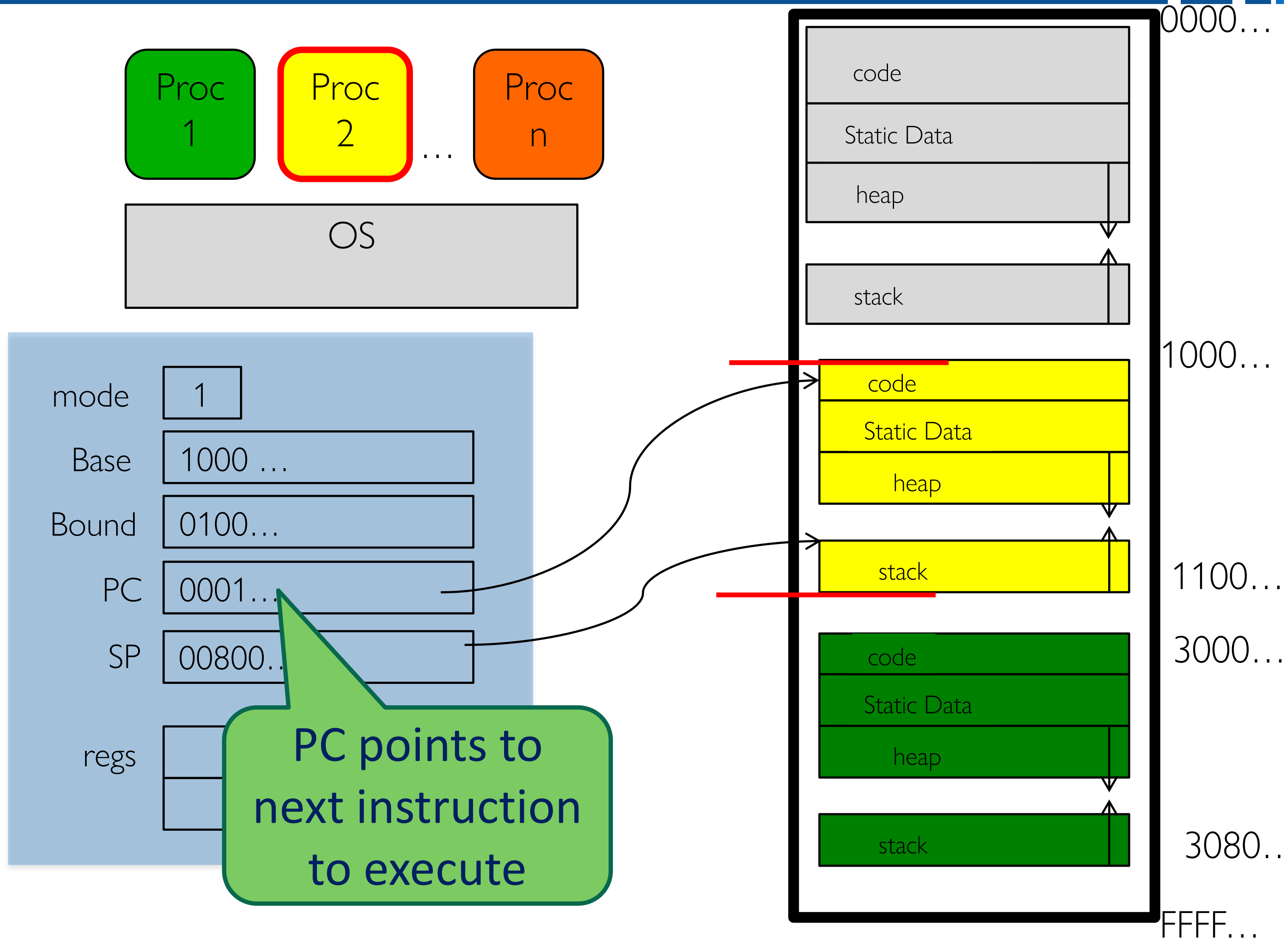
Multiple Processes



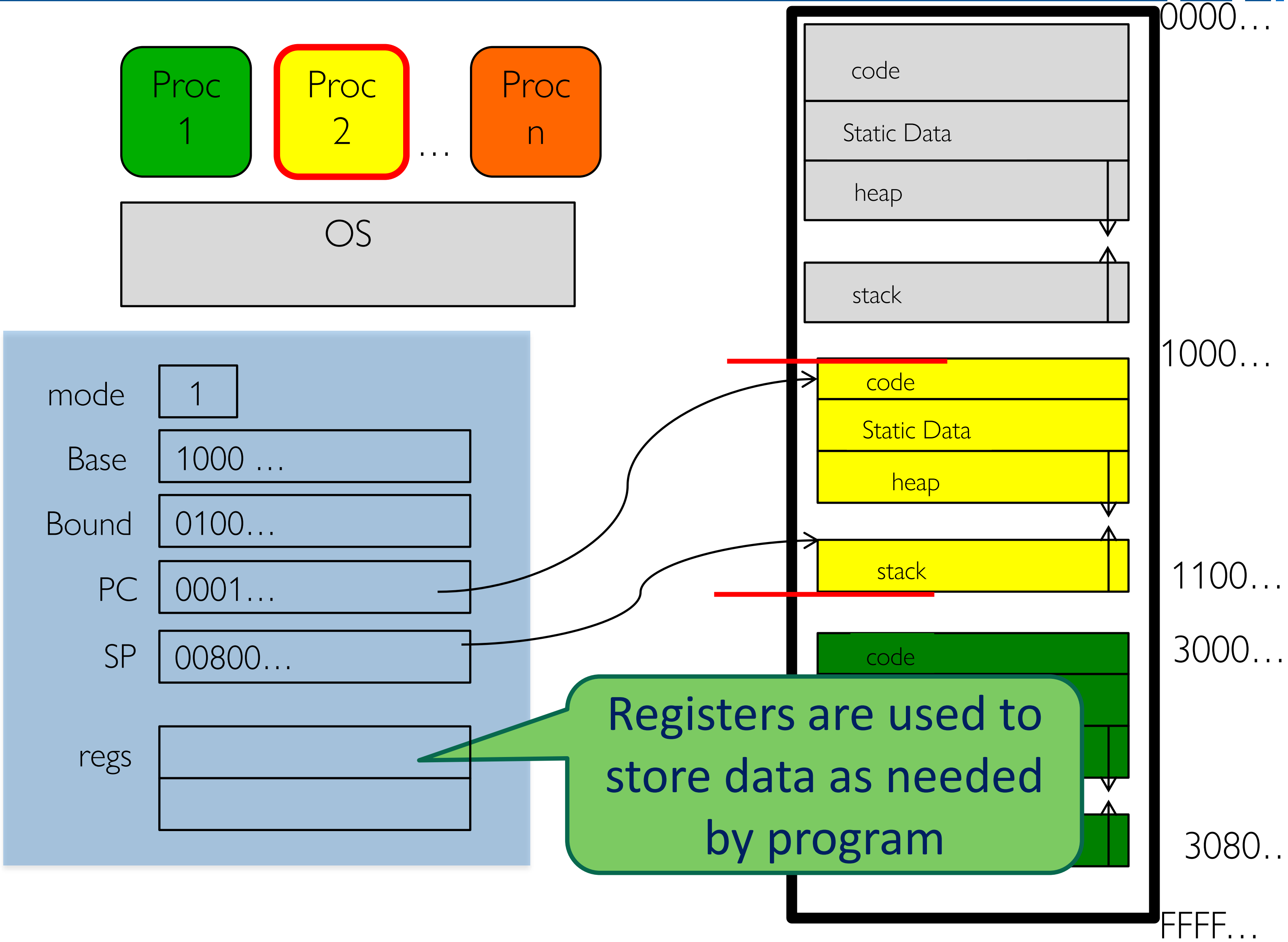
Multiple Processes



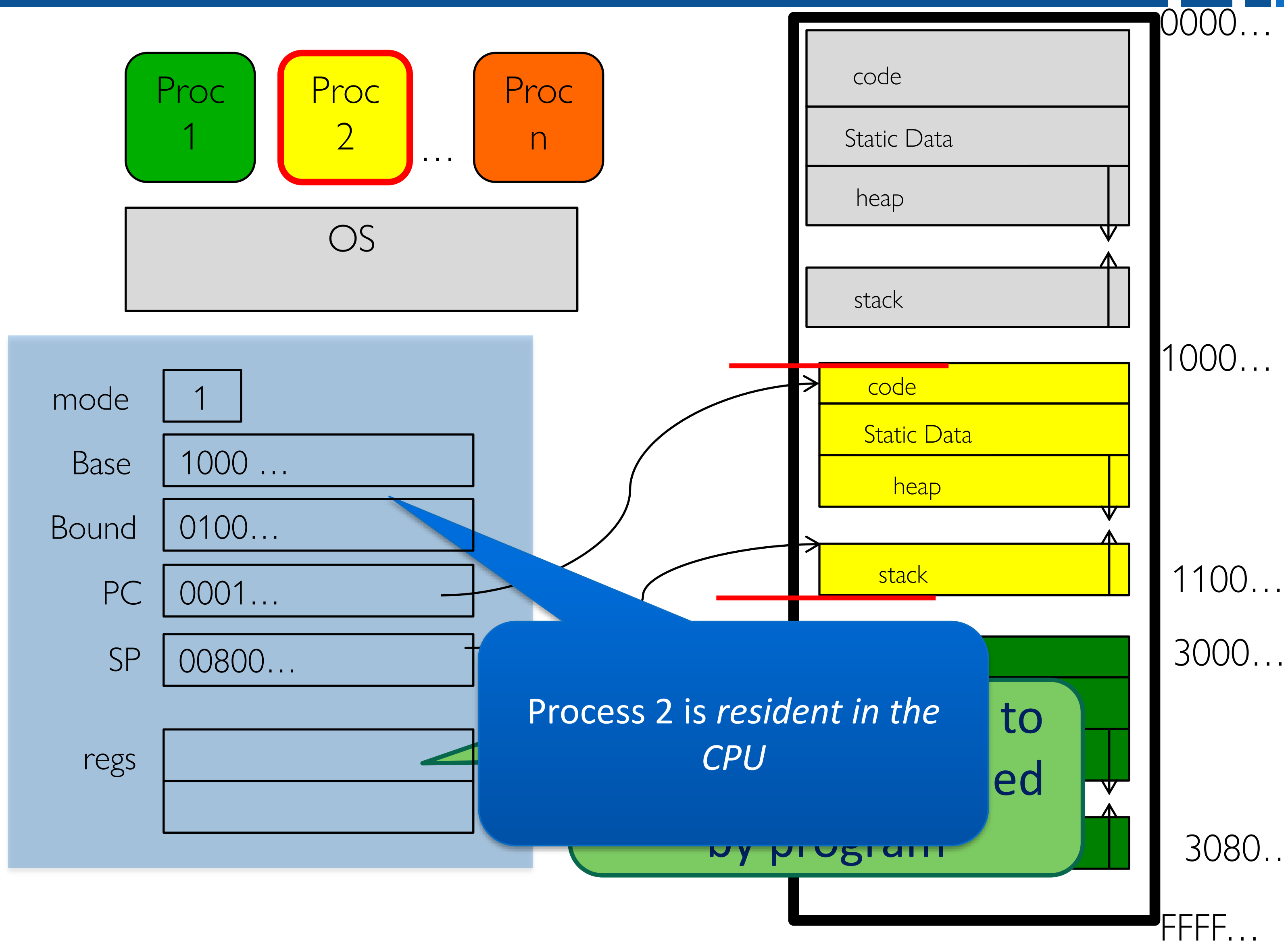
Multiple Processes



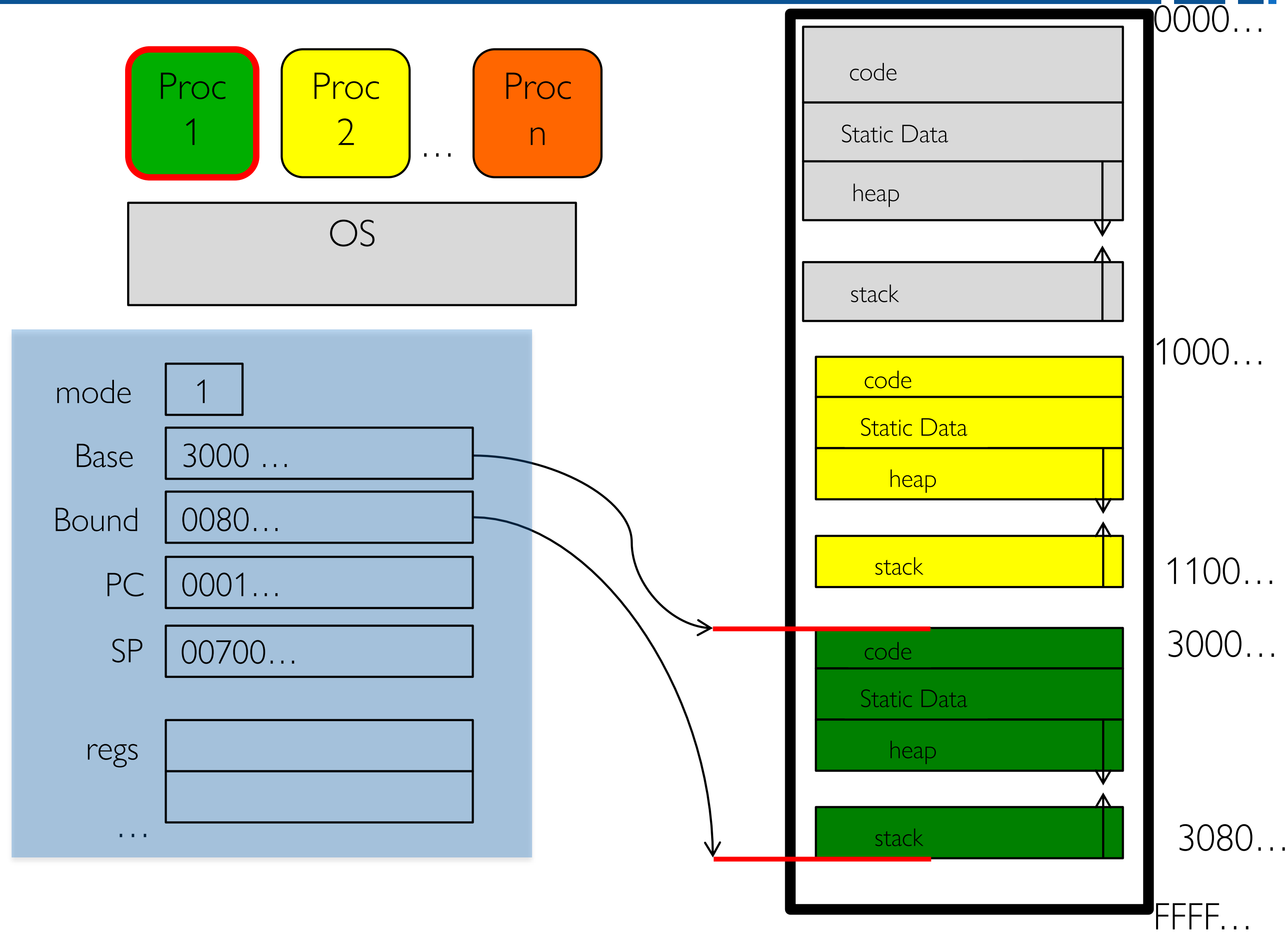
Multiple Processes



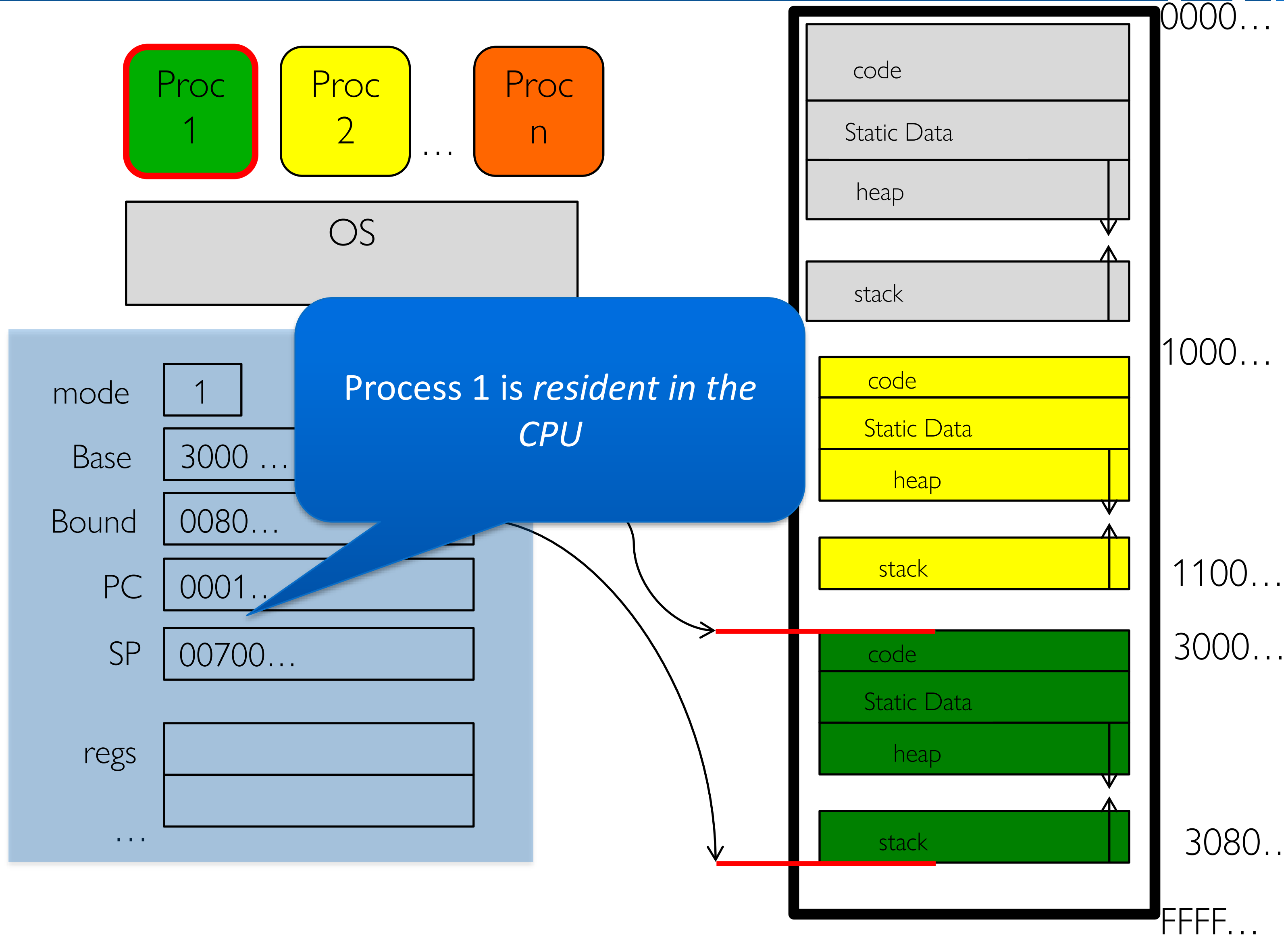
Multiple Processes



Multiple Processes



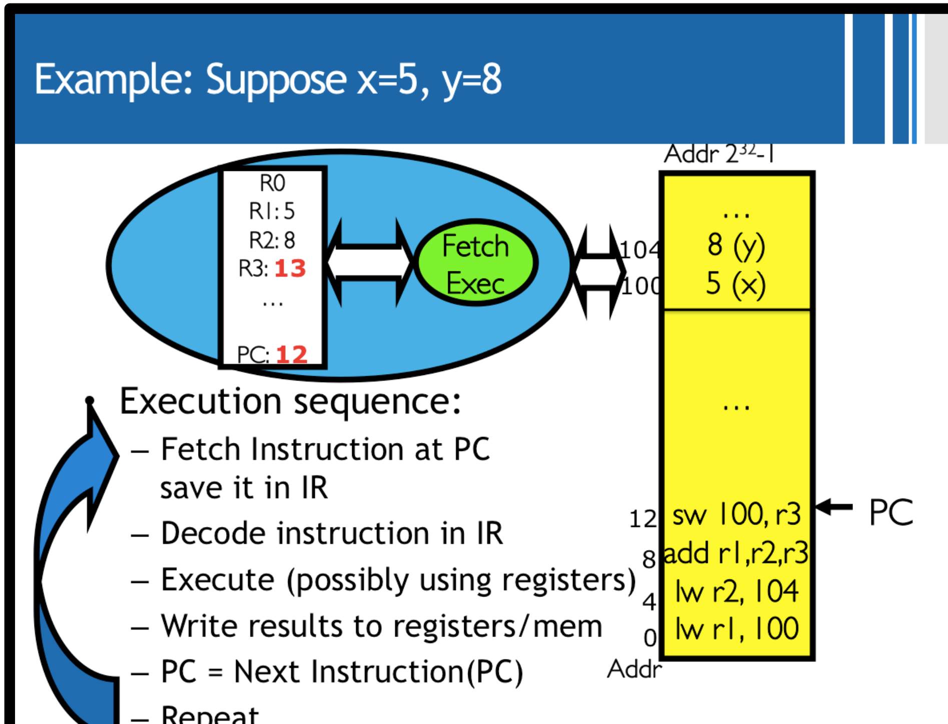
Multiple Processes



What should be saved when switching between process 1 to process 2?

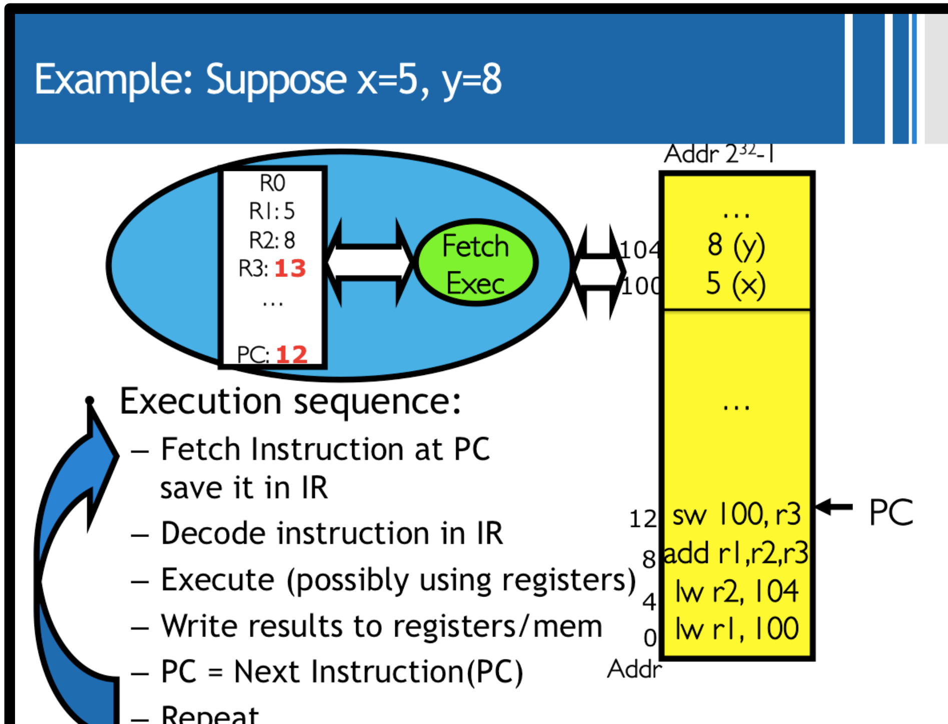
What should be saved when switching between process 1 to process 2?

- Process 2 context



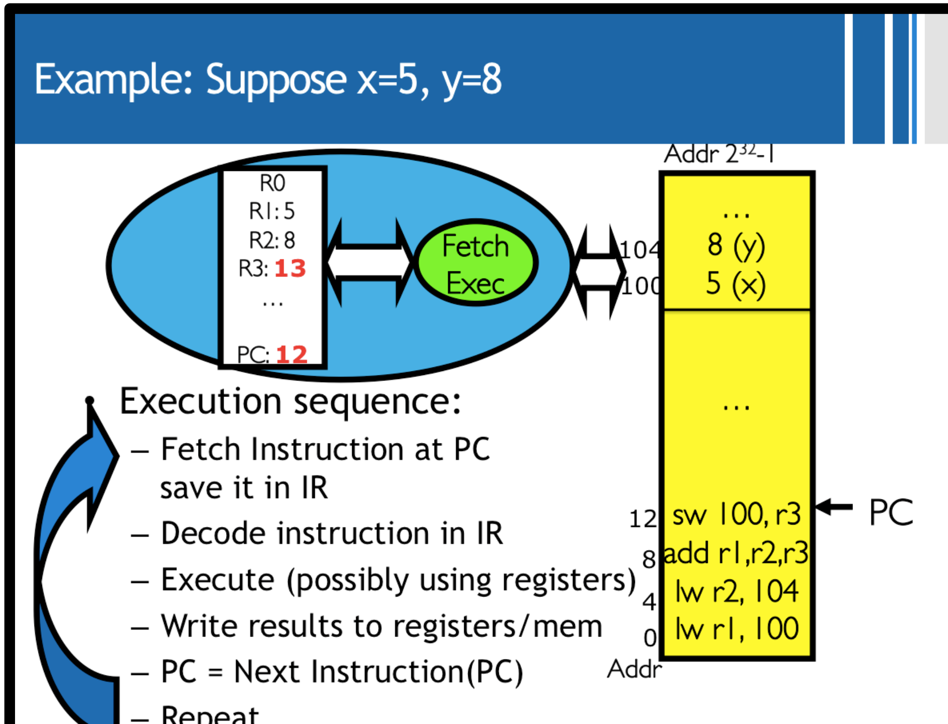
What should be saved when switching between process 1 to process 2?

- Process 2 context
- Everything in memory stays in memory (no need to do anything)
- Everything in CPU should be copied



What should be saved when switching between process 1 to process 2?

- Process 2 context
- Everything in memory stays in memory (no need to do anything)
- Everything in CPU should be copied
- Example: save PC but not code, save SP but not entire stack



Process Control Block (PCB)

How the process is represented in the OS

Process ID
Process state
User
Accounting info
Priority
Allocated memory
Open files
Open commun. channels
Storage space for CPU state (registers)

- Context
- Additional OS information needed:
  - Memory management
  - Accounting information
  - I/O status information
  - User/permissions
  - …

Process Control Block (PCB)

How the process is represented in the OS

Process ID
Process state
User
Accounting info
Priority
Allocated memory
Open files
Open commun. channels
Storage space for CPU state (registers)

Needed for protection: what is the process allowed to do?

- Context

S

needed:

nagement

  - Accounting information
  - I/O status information
  - User/permissions
  - …

Process Control Block (PCB)

How the process is represented in the OS

Process ID
Process state
User
Accounting info
Priority
Allocated memory
Open files
Open commun. channels
Storage space for CPU state (registers)

Needed for scheduling decisions: when should the process run?

- Context
- Additional OS information needed:
  - …nagement
  - … information
  - … information
  - User/permissions
  - …

Process Control Block (PCB)

How the process is represented in the OS

Process ID
Process state
User
Accounting info
Priority
Allocated memory
Open files
Open commun. channels
Storage space for CPU state (registers)

Point to relevant data in other OS data structures

- Context
- Additional OS information needed:
  - Memory management
  - Accounting information
  - I/O status information
  - ssions

Process Control Block (PCB)

How the process is represented in the OS

Process ID
Process state
User
Accounting info
Priority
Allocated memory
Open files
Open commun. channels
Storage space for CPU state (registers)

For when process is not running; will be restored when scheduled to run again

- Context
- Additional OS information needed:
  - Memory management
  - Accounting information
  - I/O status information
  - User/permissions
  - …

The Life-cycle of a Process

- 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
  - –
  - –

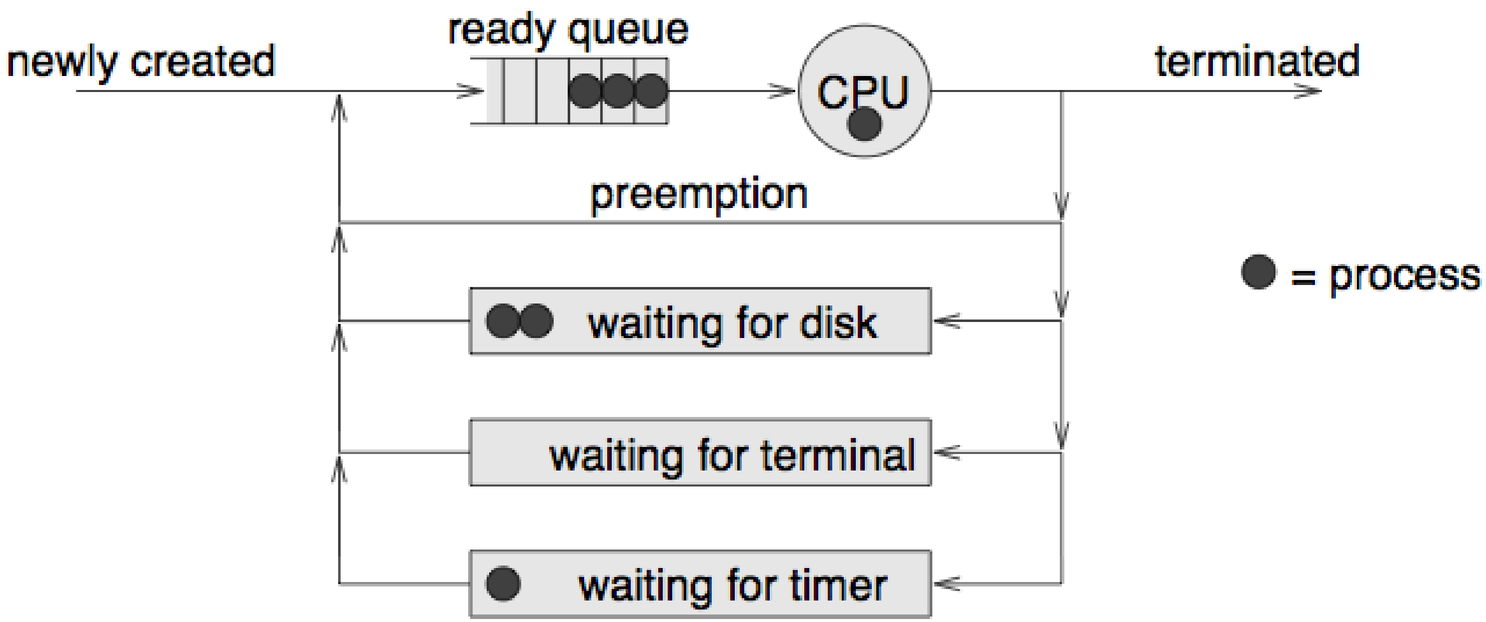


Diagram of Process State

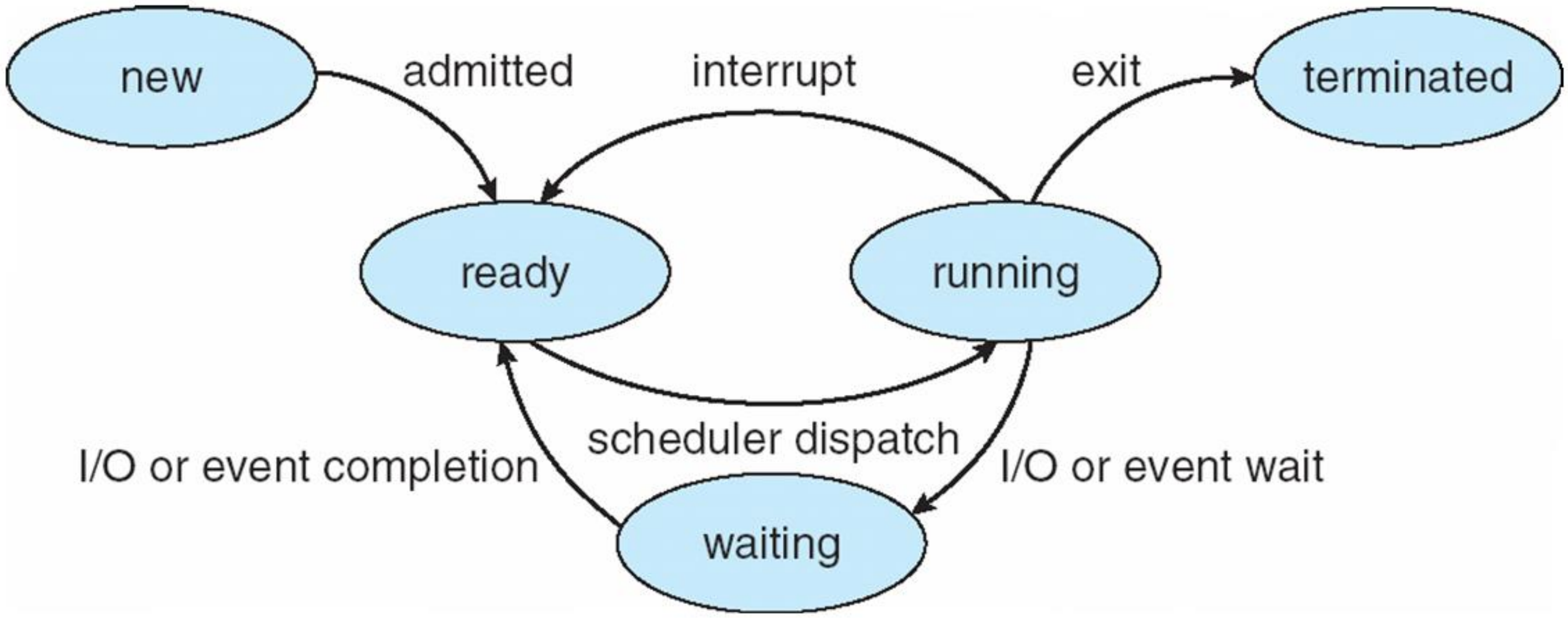
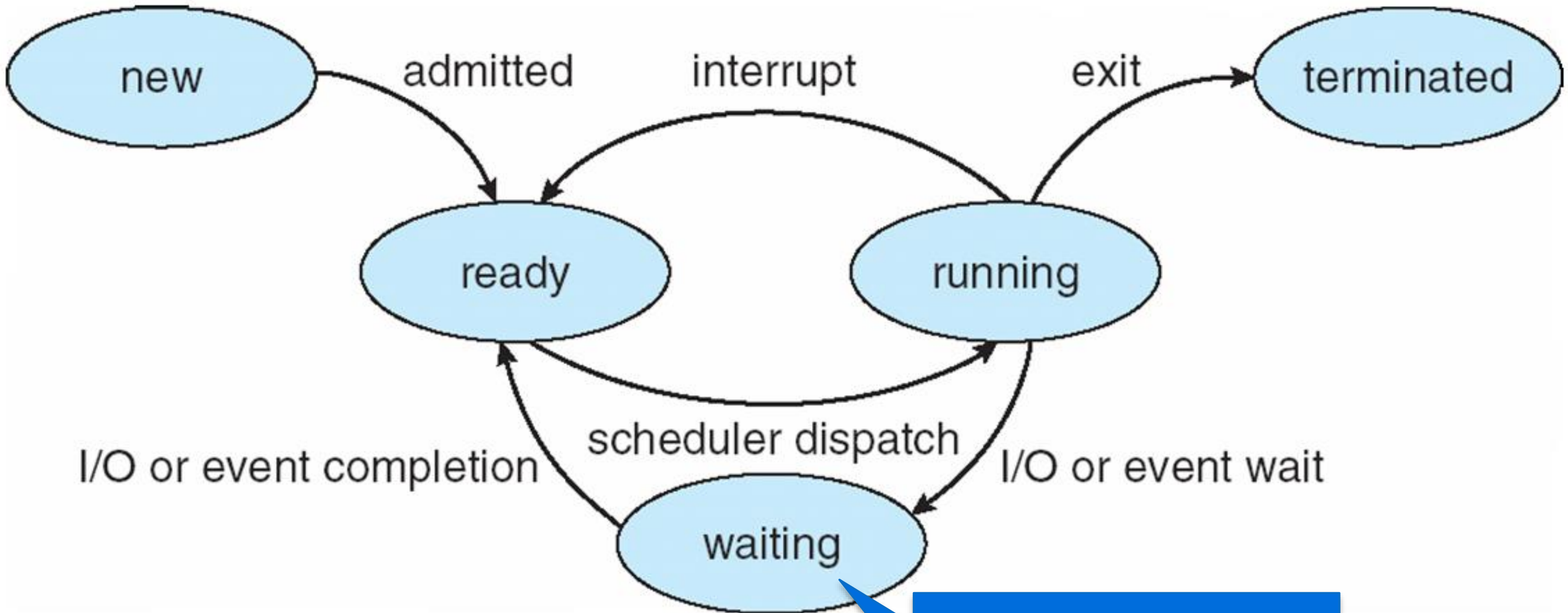
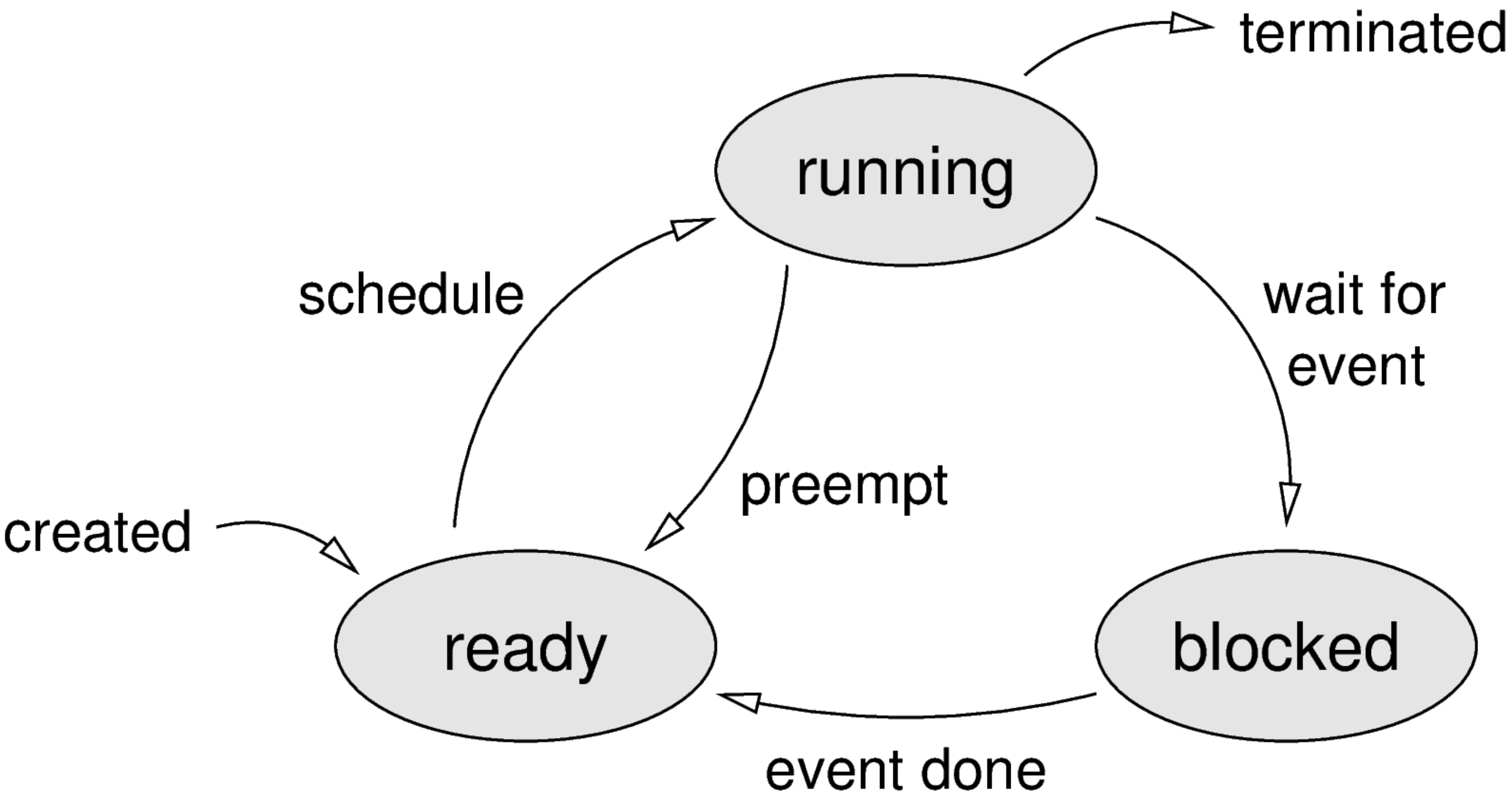


Diagram of Process State



a.k.a. Blocked

Diagram of Process State

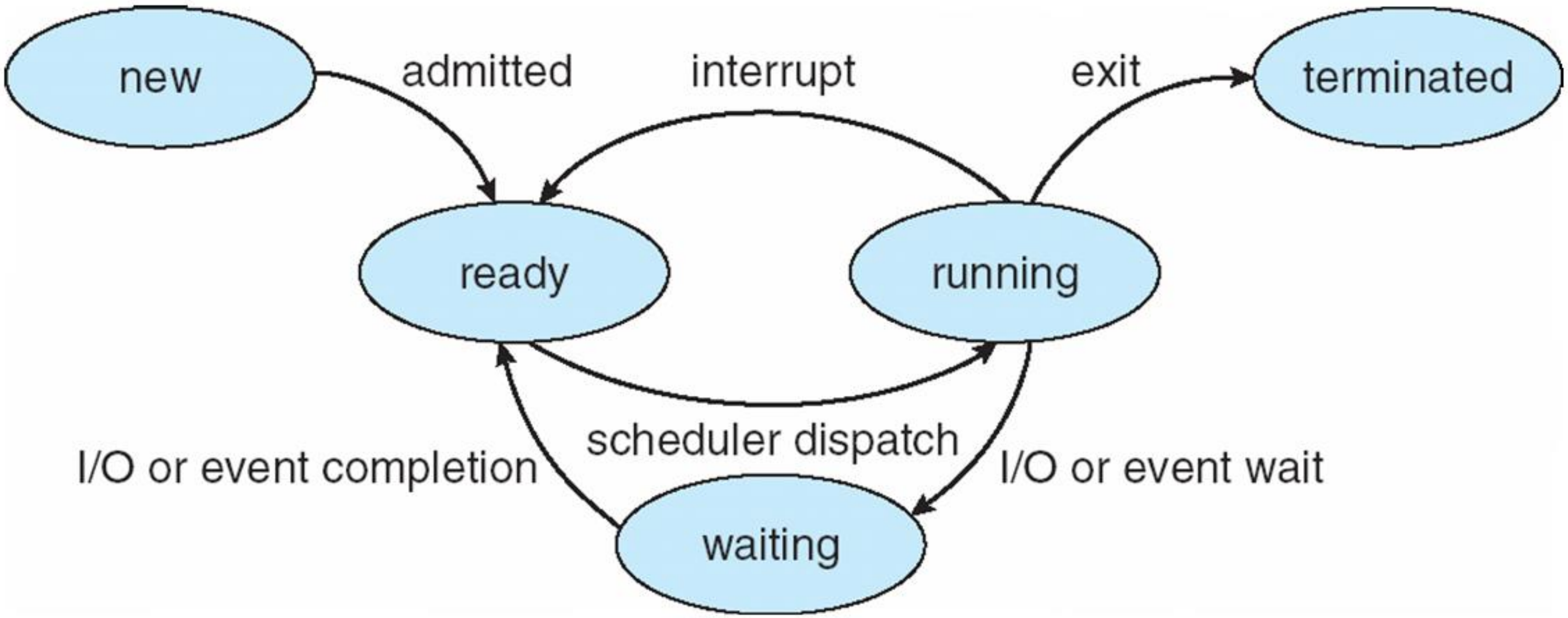


Scheduling

- Which process should get to run on the CPU and for how long?

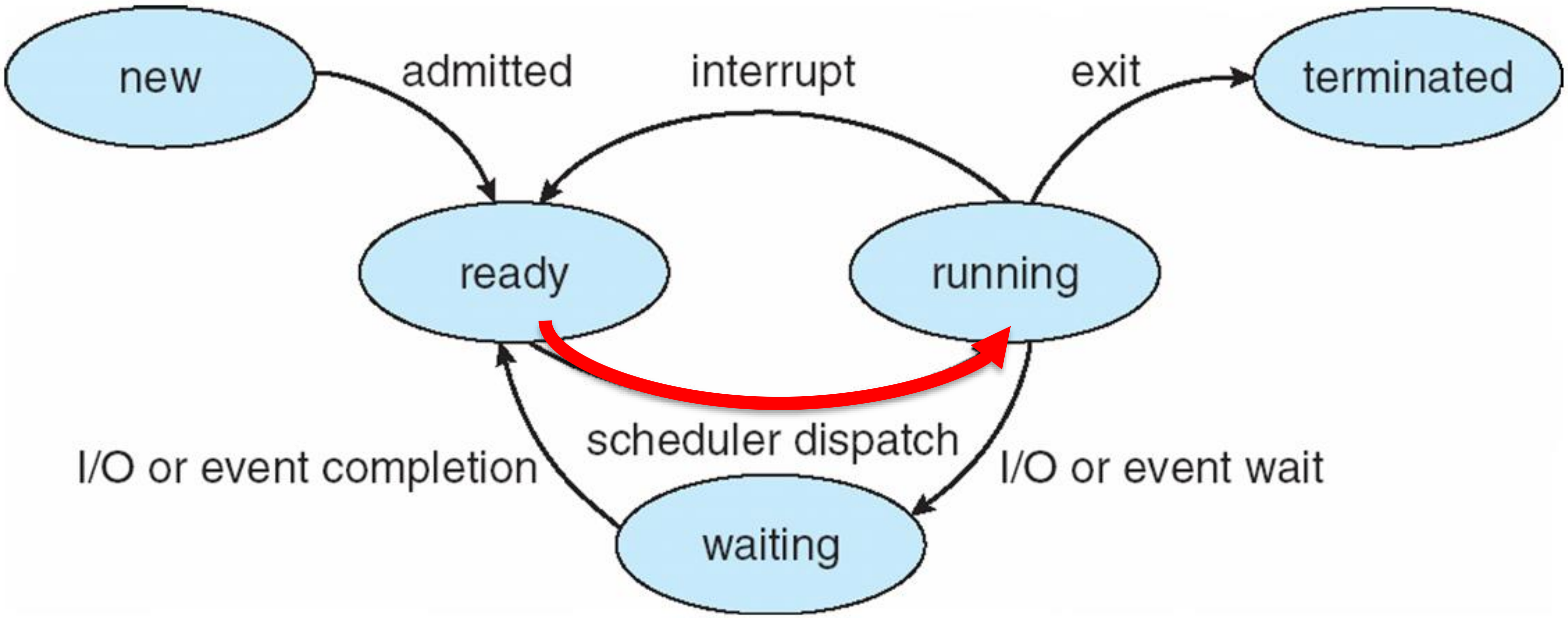
Scheduling

- Which process should get to run on the CPU and for how long?



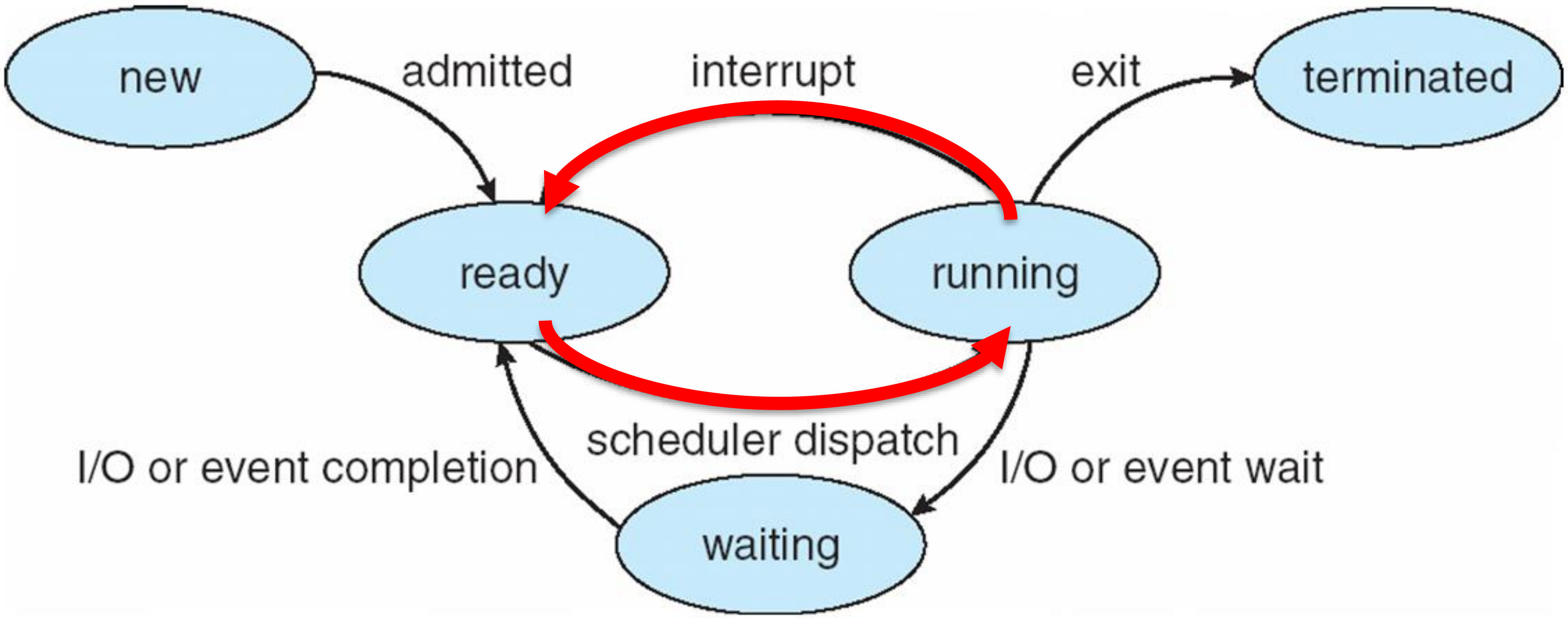
Scheduling

- Which process should get to run on the CPU and for how long?



Scheduling

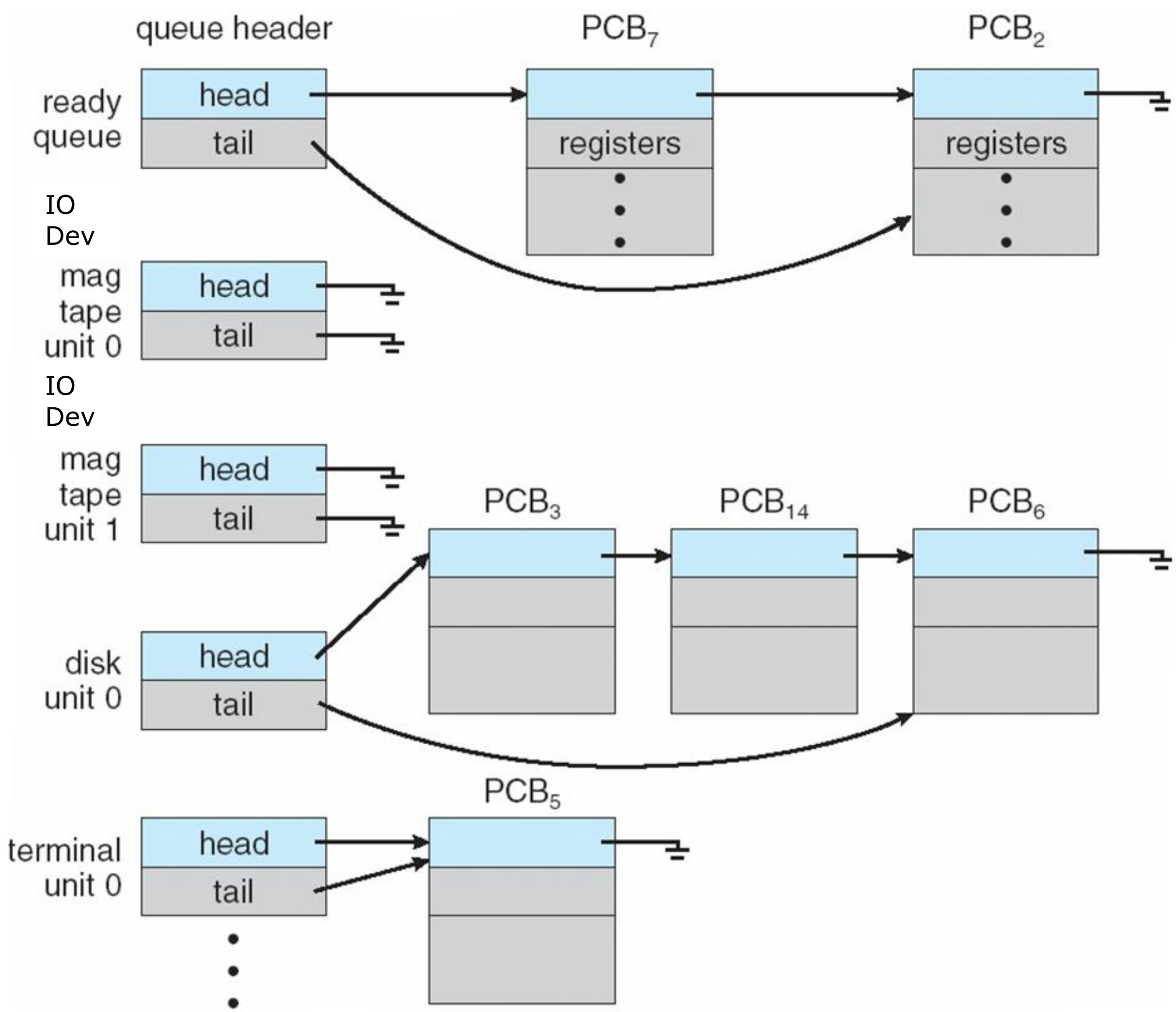
- Which process should get to run on the CPU and for how long?



A Scheduler and a Dispatcher

- **CPU Scheduler:** Decide which process should run on the CPU (and sometimes for how long)
  - "Short-term scheduling"; tens/hundreds times a second
- **Dispatcher:** The module responsible on executing the CPU scheduler decisions:
  - Context switch
  - Switching back to user mode

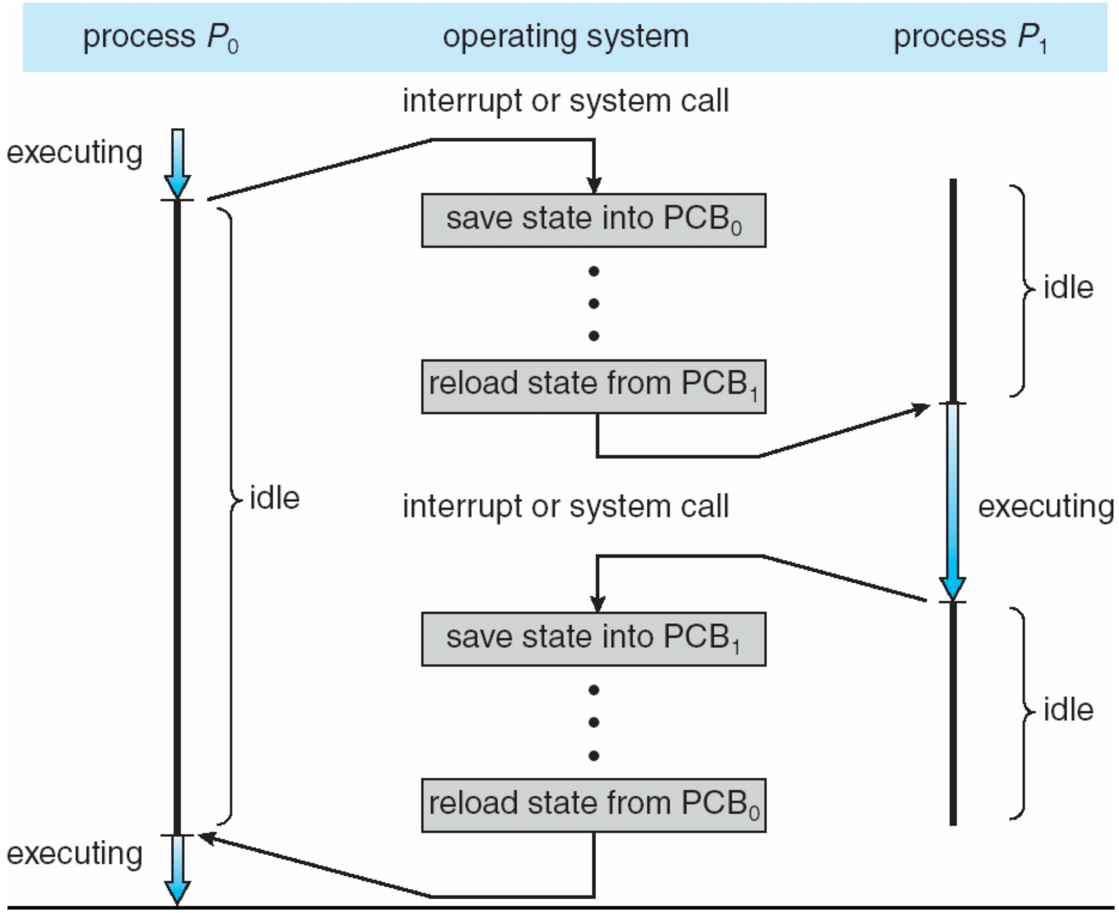
Ready Queue And Various I/O Device Queues



Context Switch between Processes

- Save processor context, including program counter and other registers
- Update the PCB with the new state and any accounting information
- Move PCB to appropriate queue - ready, blocked/waiting
- Select another process for execution
- Update the PCB of the process selected
- Update memory-management data structures
- Restore context of selected process

Context Switch between Processes



When to Switch a Process

- Interrupts
  - Clock → process has executed a full time-slice
  - I/O
- Memory fault
  - memory address is in virtual memory
  - More on this later
- System call
- Exception
  - When error occurred

Example: Process Creation using fork()

- A new process can be created by the **fork()** system call
- Child and parent are identical
  - child returns a 0
  - parent returns child process id (nonzero)
- Both parent and child execute next line

```
int pid;
int status = 0;

pid = fork()
if (pid != 0)
{
        /* parent */
        .....
}
else
{
        /* child */
        .....
}
```

Simple Example

1

```
int pid;
int status = 0;

pid = fork()
if (pid != 0)
{
        /* parent */
        printf("a");
}
else
{
        /* child */
        printf("b");
}
```

Simple Example

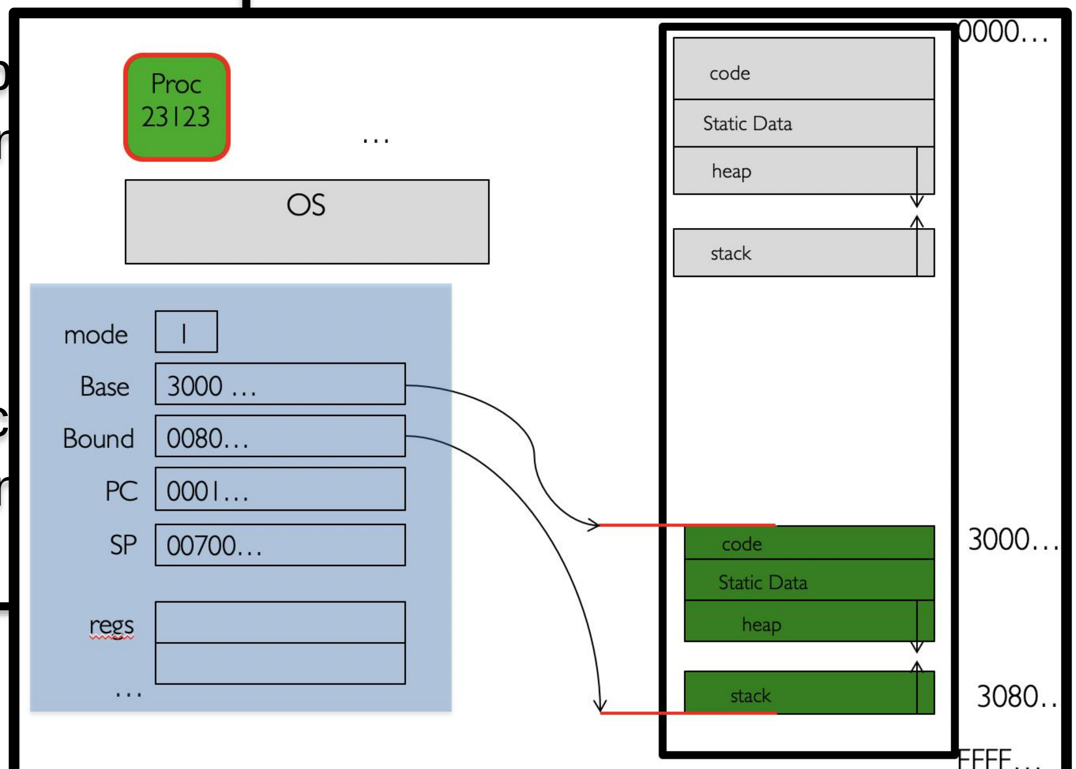
Process state = Running
Process number = 23123
Program counter
registers
Memory limits
List of open files
….

```
int pid;
int status = 0;

pid = fork()
if (pid != 0)
{
        /* parent */
        printf("a");
}
else
{
        /* child */
        printf("b");
}
```

Simple Example

Process state = Running
Process number = 23123
Program counter
registers
Memory limits
List of open files
….

```
int pid;
int status = 0;

pid = fork()
if (pid != 0)
{
        /* p
        prin
}
else
{
        /* c
        prin
}
```



Simple Example

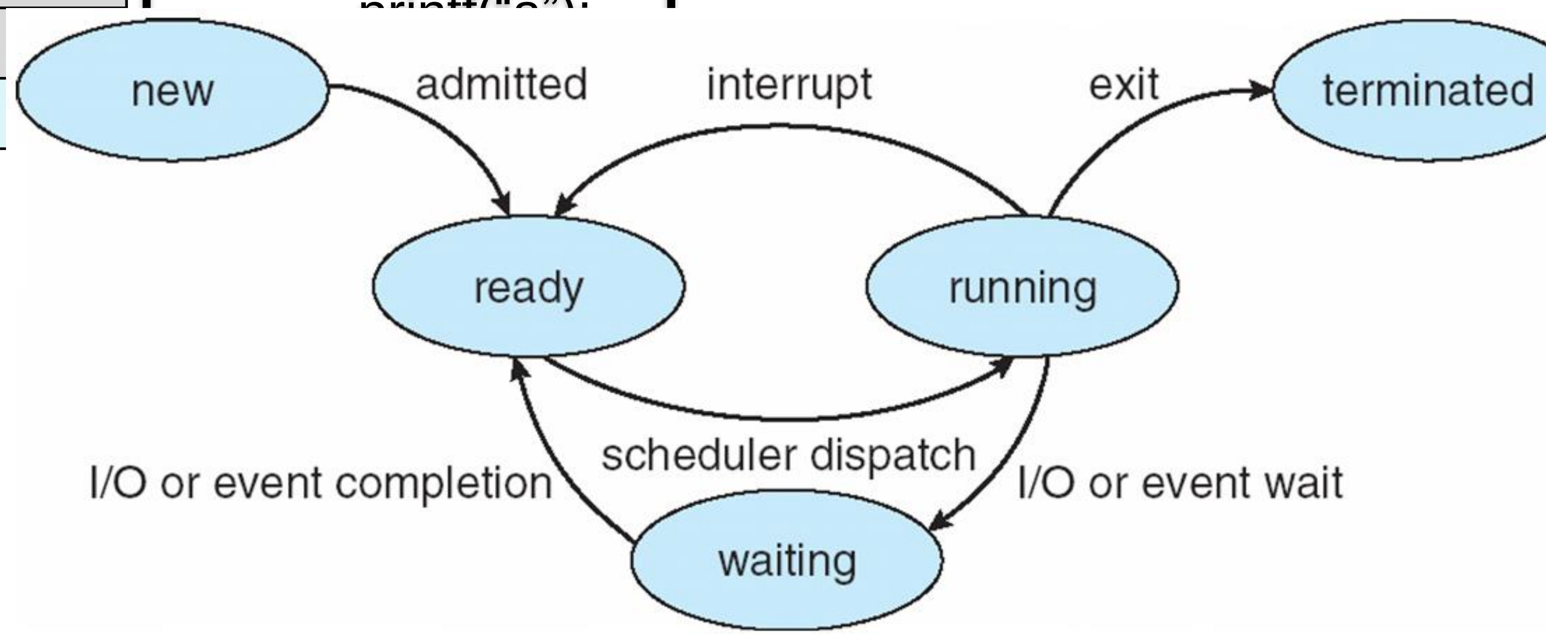
Process state = Running
Process number = 23123
Program counter
registers
Memory limits
List of open files
….

```
int pid;
int status = 0;

pid = fork()
if (pid != 0)
{
        /* parent */
        printf("a");
}
else
{
        /* child */
        printf("b");
}
```

Simple Example

Process state = Running
Process number = 23123
Program counter
registers
Memory limits
List of open files
….

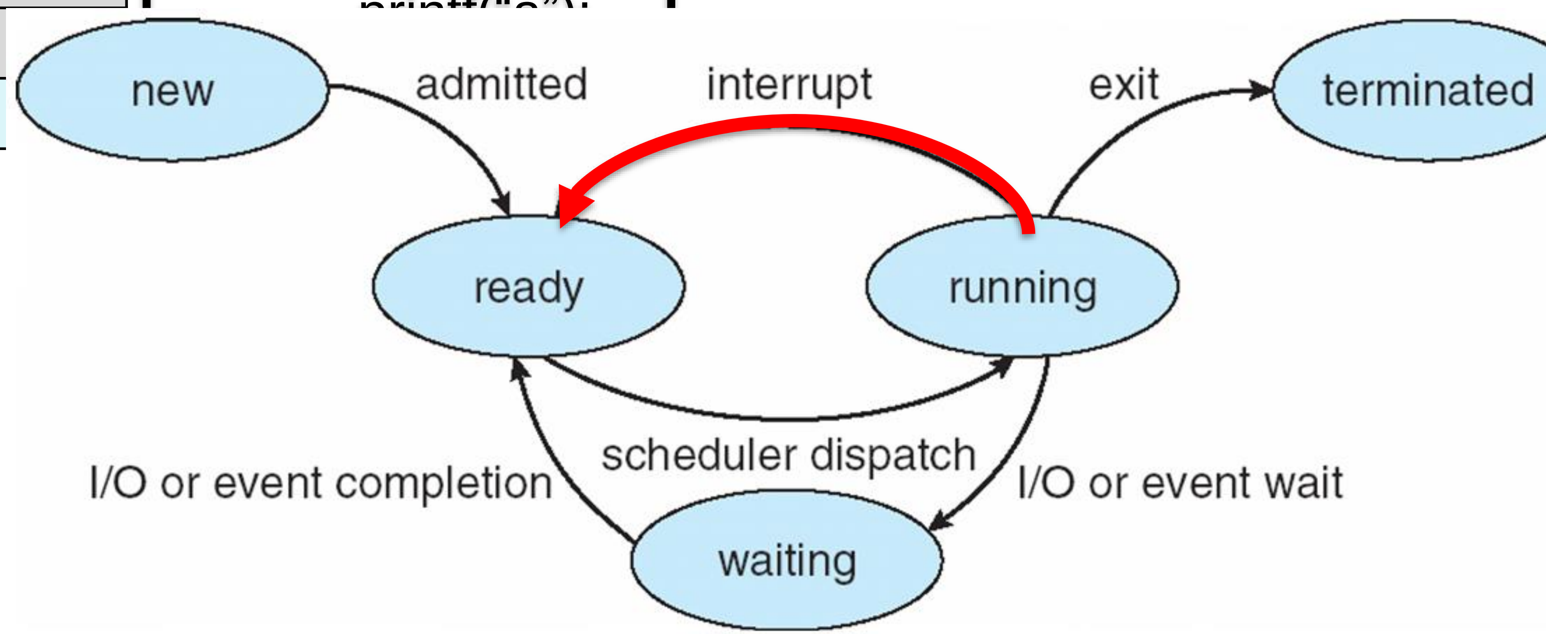
```
int pid;
int status = 0;

pid = fork()
if (pid != 0)
{
        /* parent */
```



Simple Example

Process state = Running
Process number = 23123
Program counter
registers
Memory limits
List of open files
….

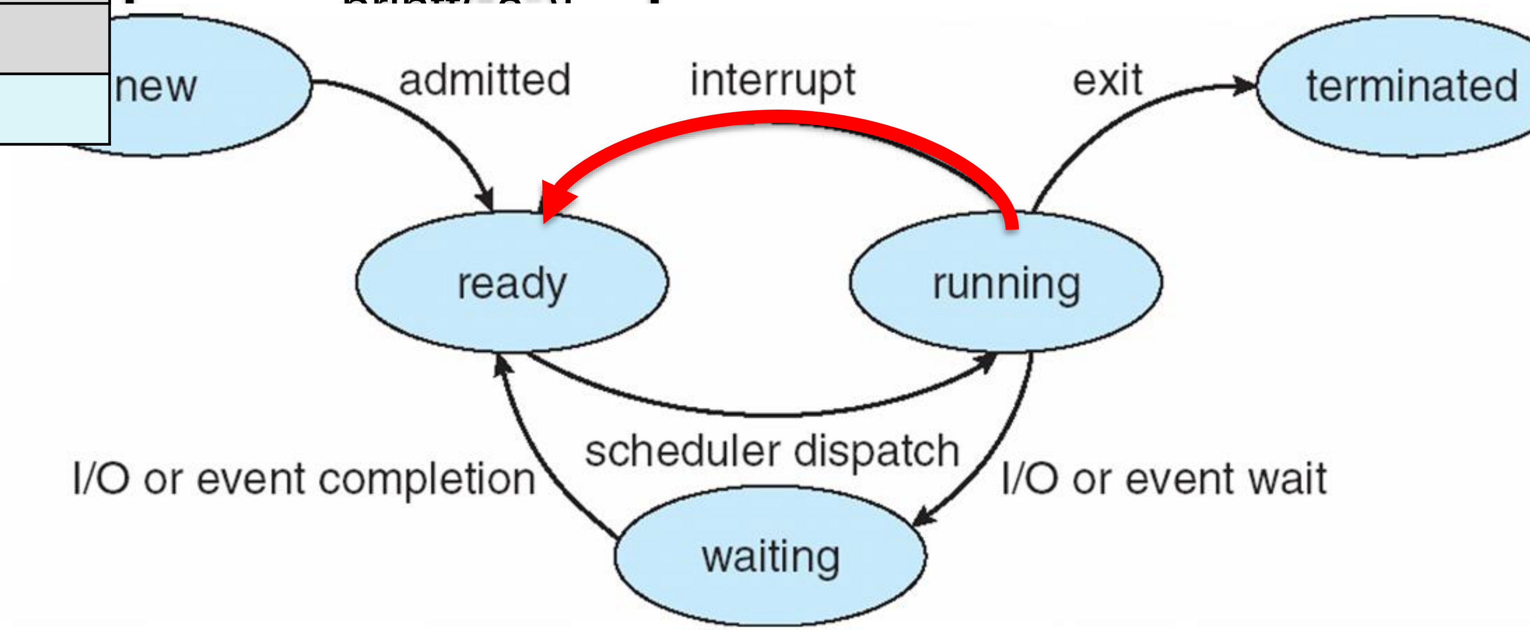
```
int pid;
int status = 0;

pid = fork()
if (pid != 0)
{
        /* parent */
```



Simple Example

Process state = Ready
Process number = 23123
Program counter
registers
Memory limits
List of open files
….

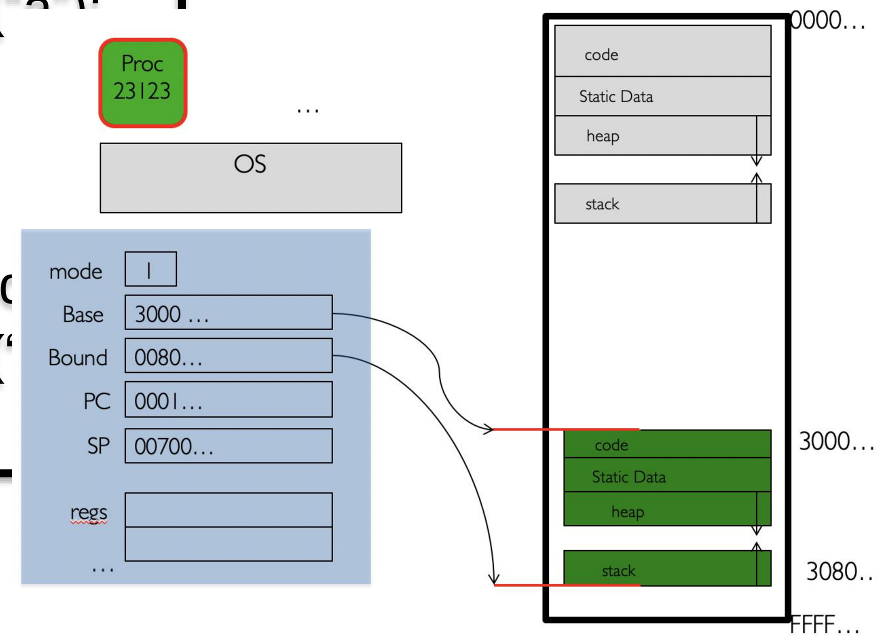
```
int pid;
int status = 0;

pid = fork()
if (pid != 0)
{
            /* parent */
```



Simple Example

Process state = Ready
Process number = 23123
Program counter
registers
Memory limits
List of open files
….

```
int pid;
int status = 0;

pid = fork()
if (pid != 0)
{
        /* parent */
        printf(“
}
else
{
        /* chil
        printf(“
}
```

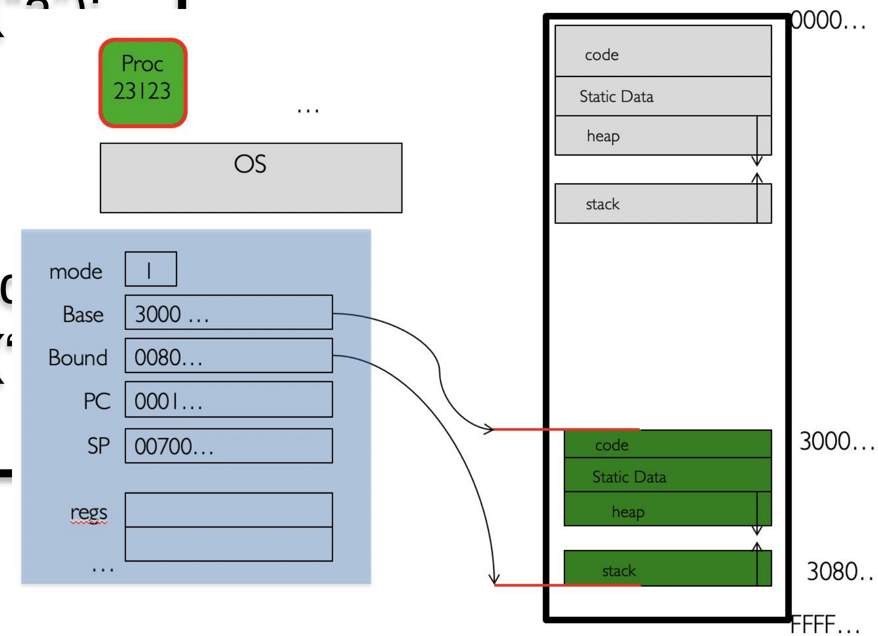


Simple Example

Process state = Ready
Process number = 23123
Program counter
registers
Memory
List of o
....

```
int pid;
int status = 0;

pid = fork()
if (pid != 0)
{
    /* parent */
    printf("a");

    /* chil
    printf("
```

Process state = Ready
Process number = 24212
Program counter
registers
Memory limits
List of open files
....

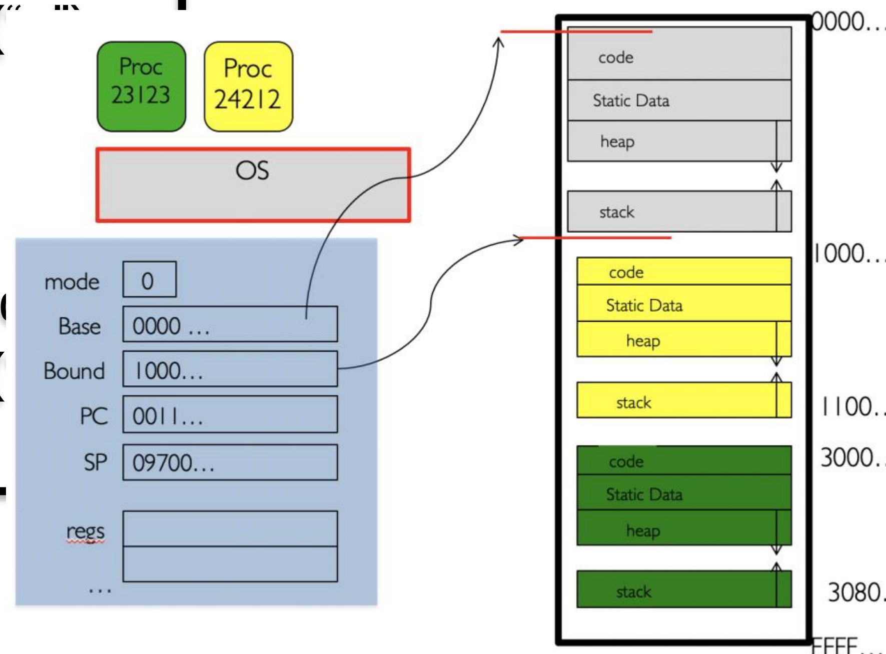


Simple Example

Process state = Ready
Process number = 23123
Program counter
registers
Memory
List of o
....

```
int pid;
int status = 0;

pid = fork()
if (pid != 0)
{
  /* parent */
  printf("  ")

  /* chil
  printf(
```

Process state = Ready
Process number = 24212
Program counter
registers
Memory limits
List of open files
....

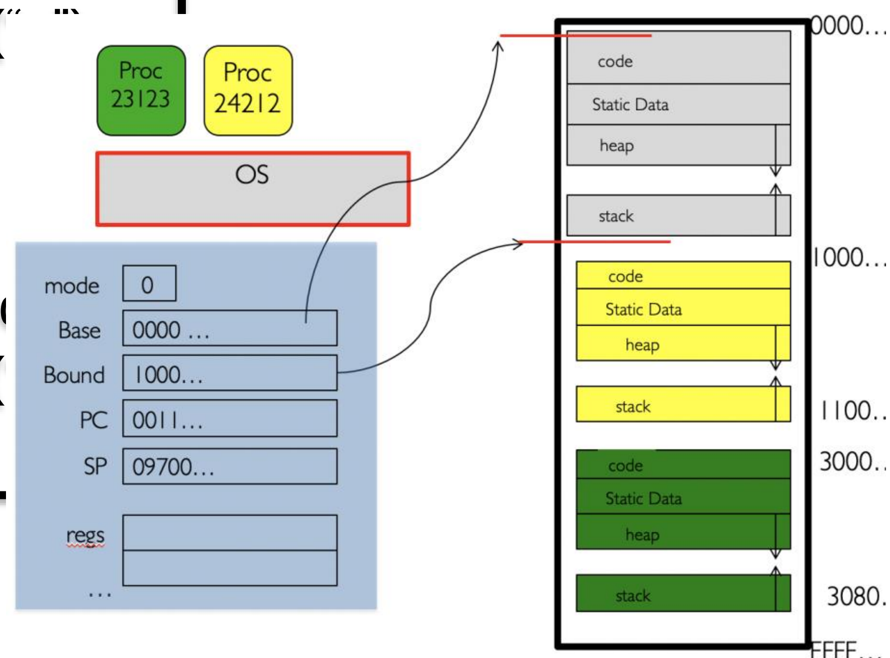


Simple Example

Process state = Ready
Process number = 23123
Program counter
EAX = 24212 registers
Memor
List of o
....

```
int pid;
int status = 0;

pid = fork()
if (pid != 0)
{
  /* parent */
  printf("...");

  /* chil
  printf(
```

Process state = Ready
Process number = 24212
Program counter
EAX = 0 registers
Memory limits
List of open files
....

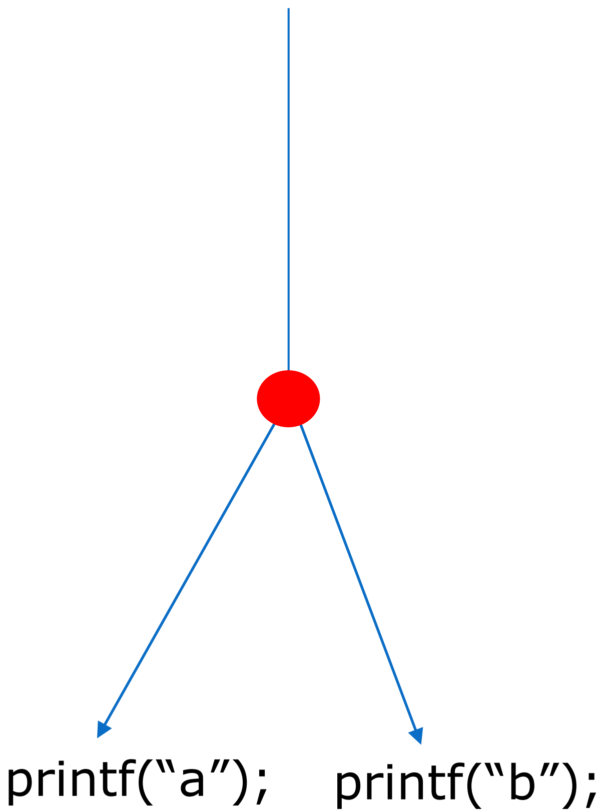


Simple Example

Process state = Ready
Process number = 23123
Program counter
EAX = 24212
registers
Memor
List of o
....

Process state = Ready
Process number = 24212
Program counter
EAX = 0
registers
Memory limits
List of open files
....

```
int pid;
int status = 0;

pid = fork()
if (pid != 0)
{
            /* parent */
            printf(“a”);
}
else
{
            /* child */
            printf(“b”);
}
```

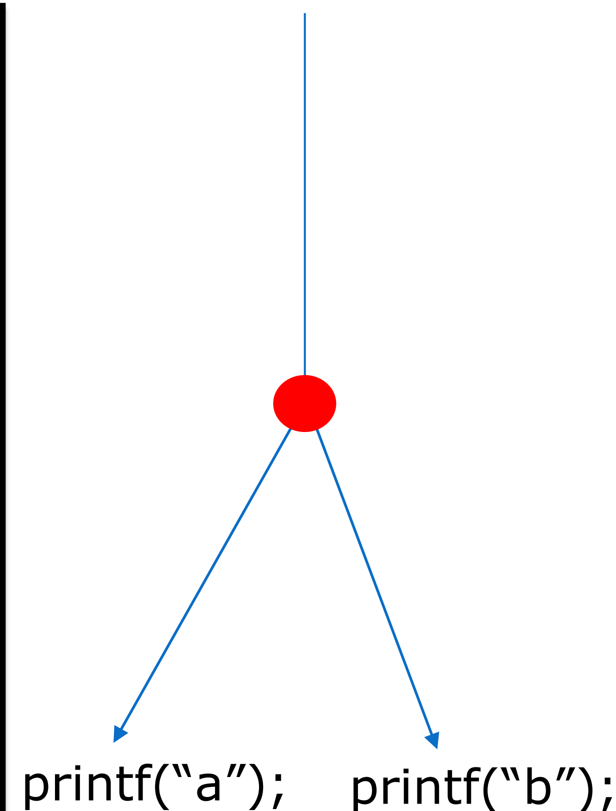


Simple Example

Process state = Ready
Process number = 23123
Program counter
EAX = 24212
registers
Memor
List of o
....

Process state = Ready
Process number = 24212
Program counter
EAX = 0
registers
Memory limits
List of open files
....

```
int pid;
int status = 0;

pid = fork()
if (pid != 0)
{
            /* parent */
            printf("a");
}
else
{
            /* child */
            printf("b");
}
```



Two possible outputs:
ab
ba

Communication between Processes

- Sometimes processes want to cooperate with each other:
  - E.g. share information, break large task into multiple tasks
  - On the other hand, processes should be protected from each other

Communication between Processes

- Sometimes processes want to cooperate with each other:
  - E.g. share information, break large task into multiple tasks
  - On the other hand, processes should be protected from each other

→ Inter-process communication (IPC) is provided by the OS, through specific system calls → very high overhead

Inter-process Communication (IPC)

- Many ways to achieve that:
  - Specify a segment in the memory that is *shared*, and all processes can access it
    - Many times implemented through the file system
    - One writes to a file, one reads from it
  - Exchanging messages through communication channel (e.g., the socket interface)

Inter-process Communication (IPC)

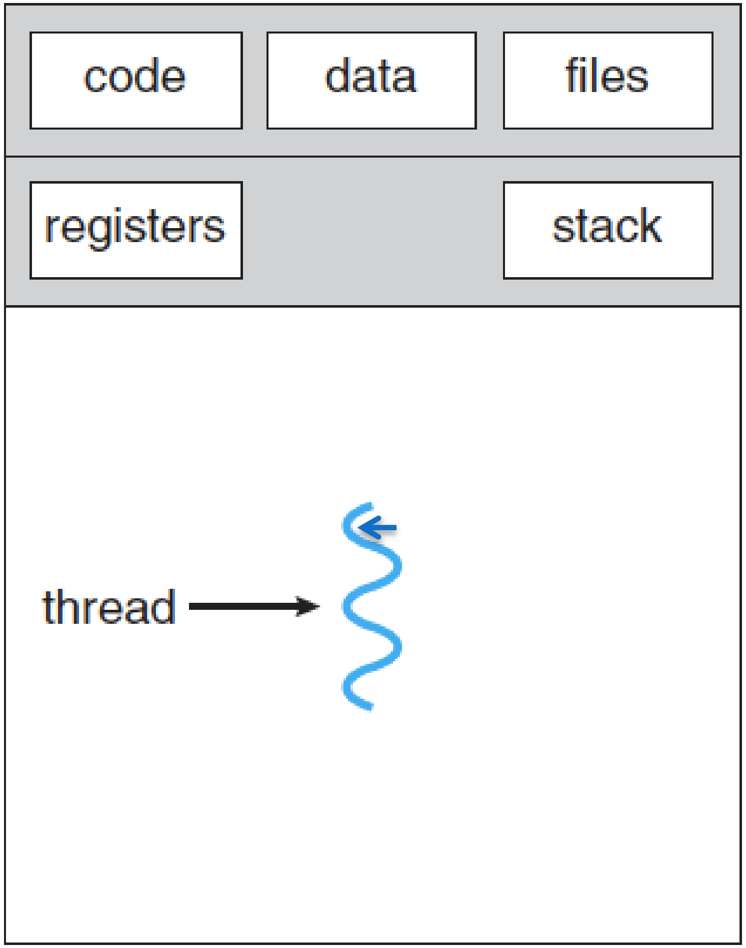
- Many ways to achieve that:
  - Specify a segment in the memory that is *shared*, and all processes can access it
    - Many times implemented through the file system
    - One writes to a file, one reads from it
  - Exchanging messages through communication channel (e.g., the socket interface)
- Sometimes classified to either shared-memory or message-passing systems

Inter-process Communication (IPC)

- Many ways to achieve that:
  - Specify a segment in the memory that is *shared*, and all processes can access it
    - Many times implemented through the file system
    - One writes to a file, one reads from it
  - Exchanging messages through communication channel (e.g., the socket interface)
- Sometimes classified to either shared-memory or message-passing systems
- Create many **synchronization** problems.

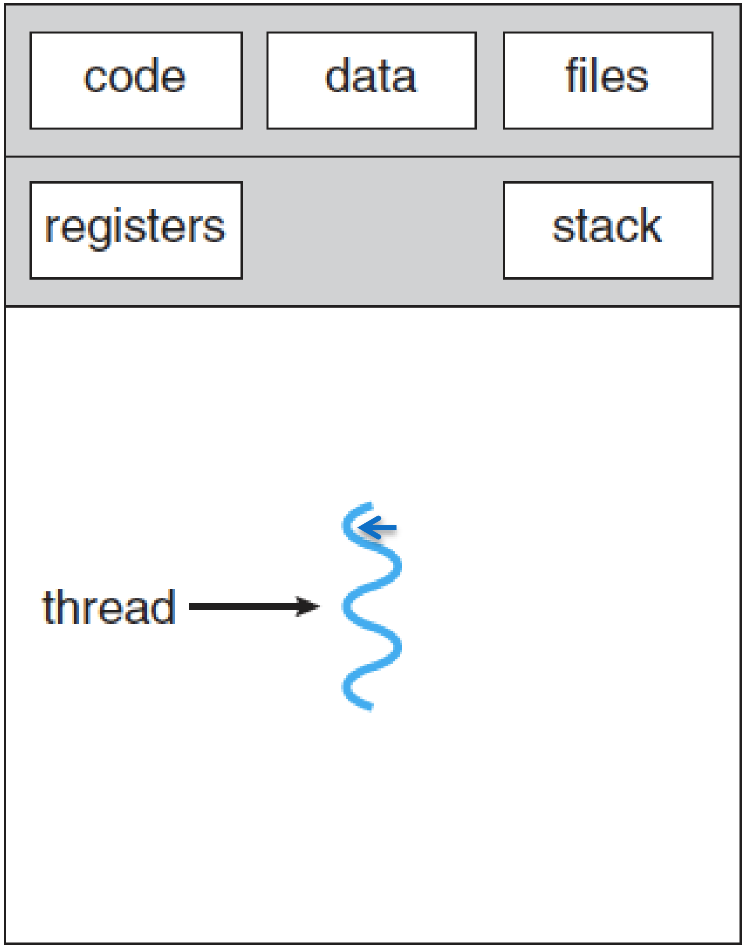
NEXT CONCEPT: THREADS

Thread of Control



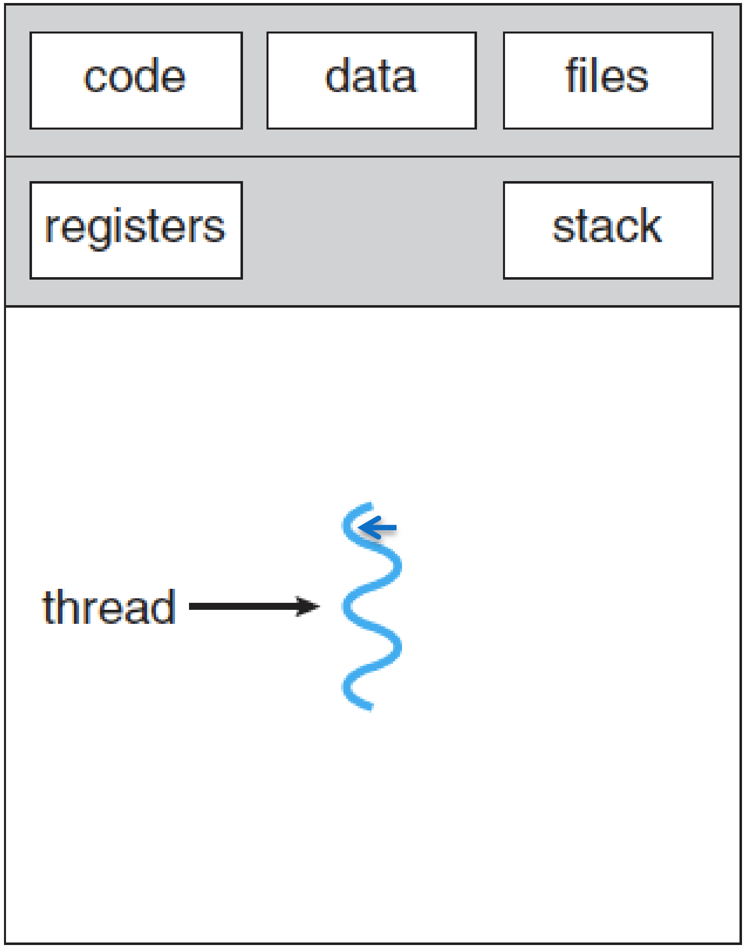
Thread of Control

- Thread: path of execution
- Up until now, each process had one thread of control
  - Defined by PC and stack



Thread of Control

- Thread: path of execution
- Up until now, each process had one thread of control
  - Defined by PC and stack
- Multithreading: multiple independent execution paths
  - Execute different parts of the same code concurrently
  - Share context (memory contents, open files)



Examples of Concurrent Tasks

- Many applications need to perform more than one task at once
  - Web browser: retrieve data from the network; display images of data already retrieved
  - Word processing: process keystrokes; display graphics; perform spell check on the background
  - Web server: process many requests while listening to new requests

How to do these tasks concurrently?

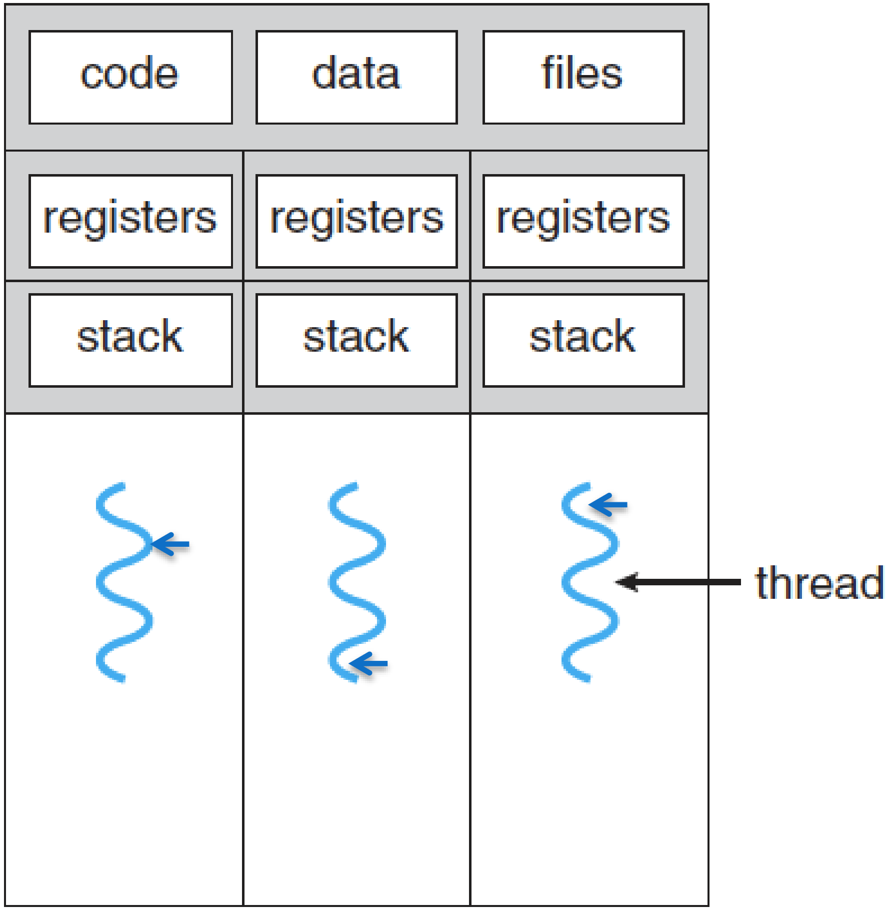
How to do these tasks concurrently?

- <u>**No concurrency:**</u> the word processor will wait on a keystroke → evicting the CPU → No other of its tasks will run until the keystroke is processed after a keyboard interrupt

How to do these tasks concurrently?

- **No concurrency:** the word processor will wait on a keystroke → evicting the CPU → No other of its tasks will run until the keystroke is processed after a keyboard interrupt
- **Each task as a process**:
  1. A lot of IPC → Very clumsy, expensive, slow.
  2. The code of the program is in memory again and again (e.g., for each web request)

Multi-threaded Processes



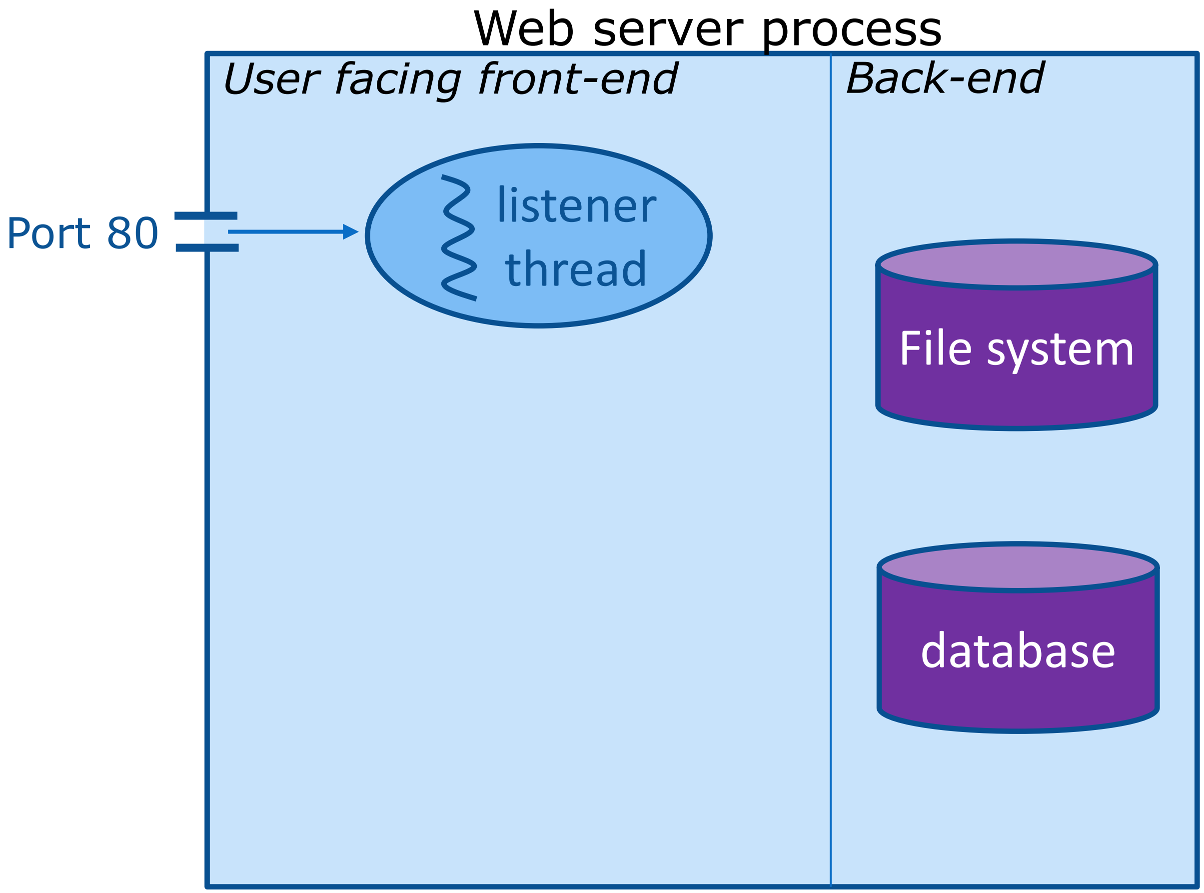
- Each thread has its own registers (including PC) and stack
- All threads share
  - The code of the program
  - All other data (global variables and the heap)
  - Files

Multi-Threaded Processes

- A **process** virtualizes the *computer*
  - Each running application sees an abstract computer dedicated to itself
- A **thread** virtualizes the *CPU*
  - An application can be structured as if it will run on multiple CPUs
  - Regardless of how many are actually available
    - Concurrency: threads share a single physical CPU (time slicing)
    - True parallelism: threads run on distinct processors

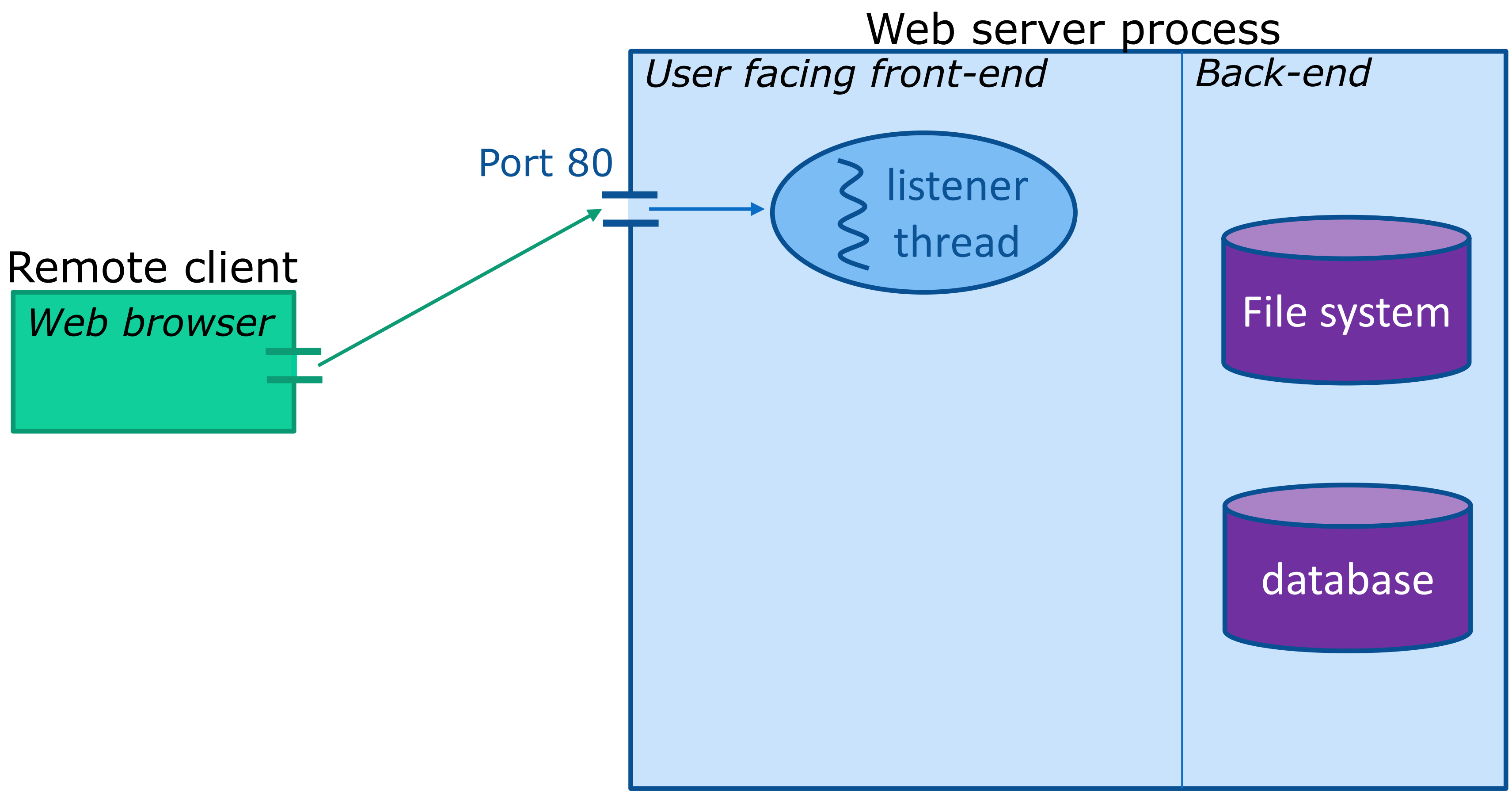
Example: Multi-Threaded Web Server

Initially the server has a single thread listening for requests



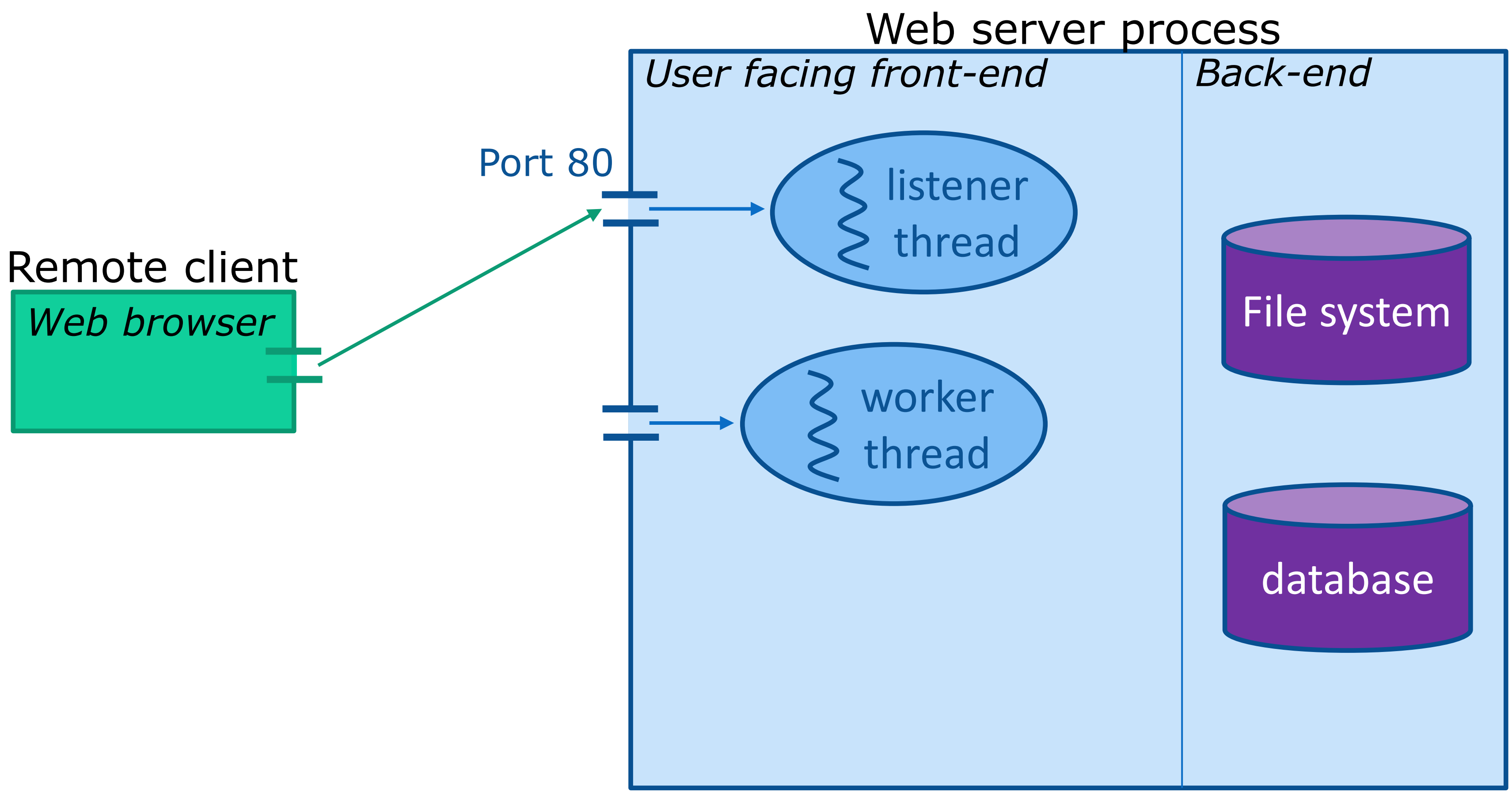
Example: Multi-Threaded Web Server

A client connects to the server



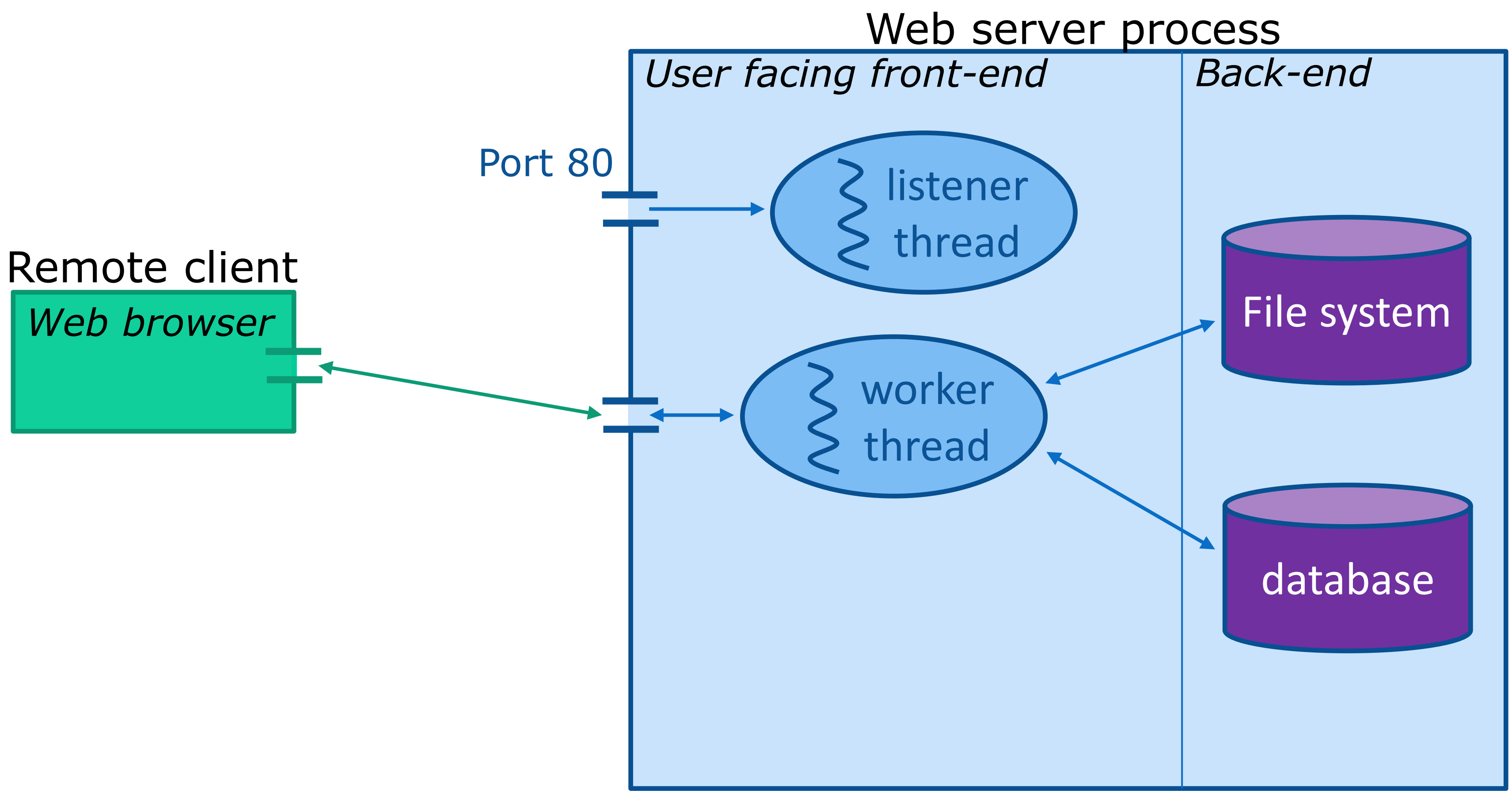
Example: Multi-Threaded Web Server

The listener creates a worker thread to handle the request



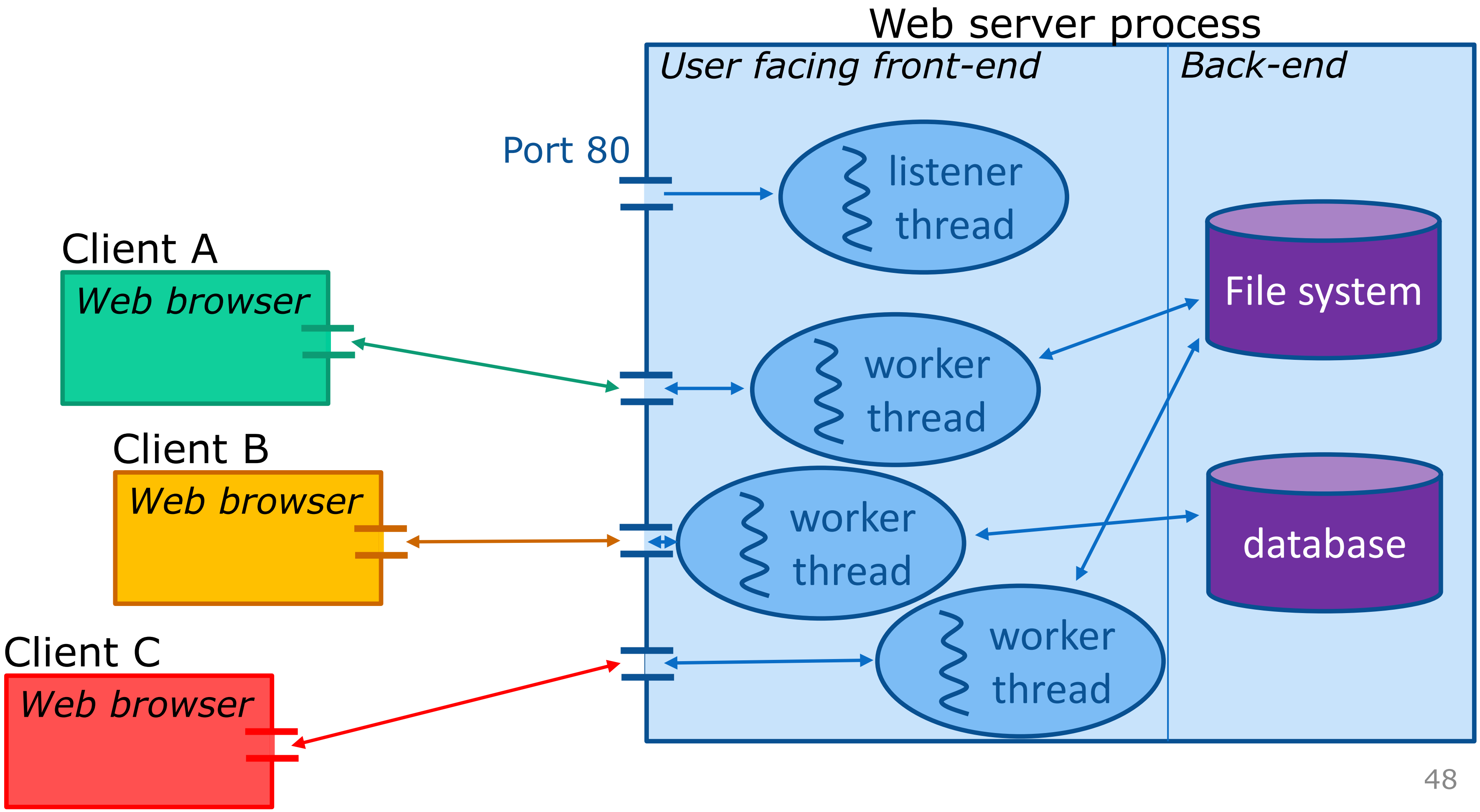
Example: Multi-Threaded Web Server

The worker thread handles the request



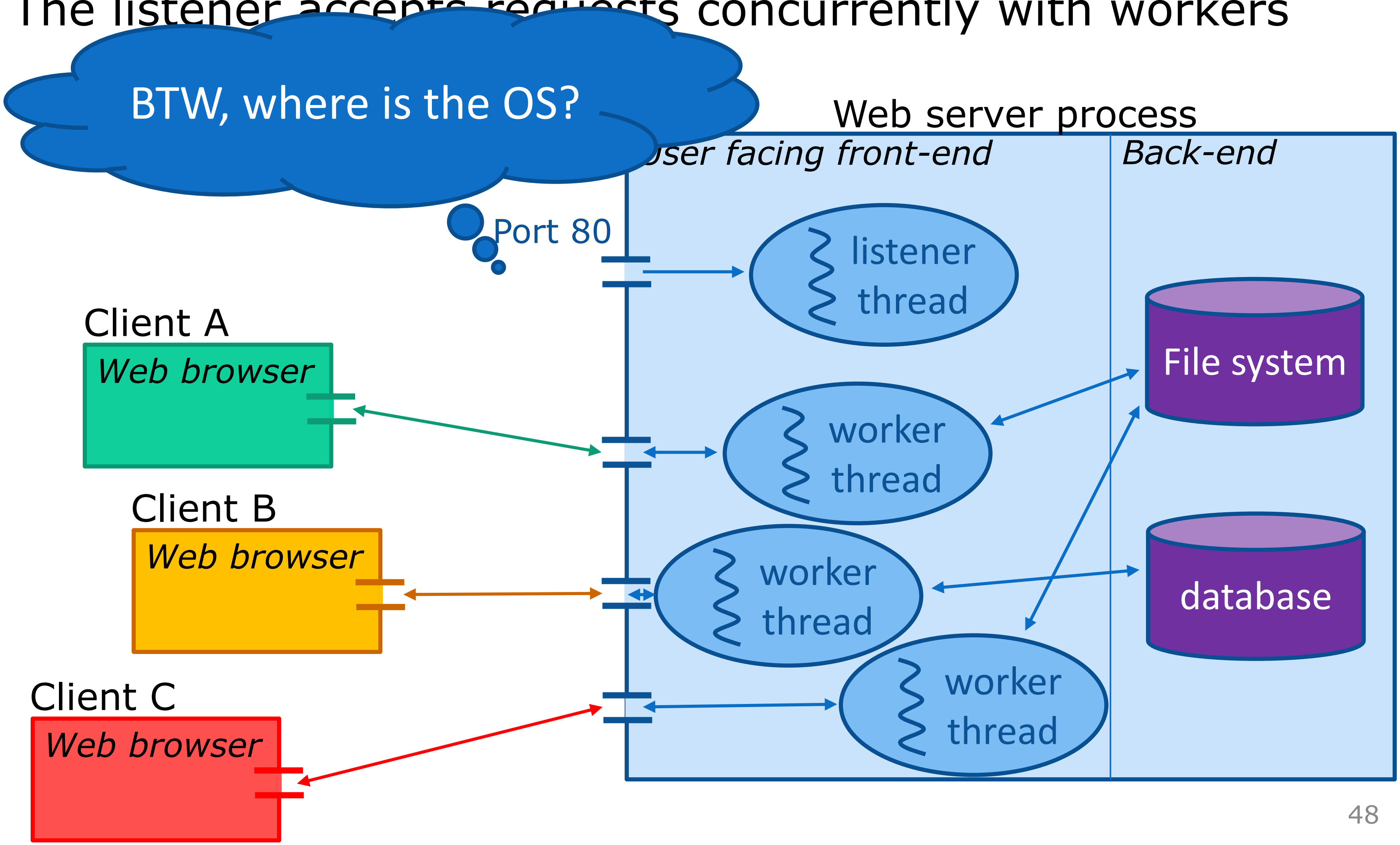
Example: Multi-Threaded Web Server

The listener accepts requests concurrently with workers



Example: Multi-Threaded Web Server

The listener accepts requests concurrently with workers



Kernel-Level Threads

- Kernel maintains context information for the process and the threads

Kernel-Level Threads

- Kernel maintains context information for the process and the threads
- Blocking a thread does not have to block the process

Kernel-Level Threads

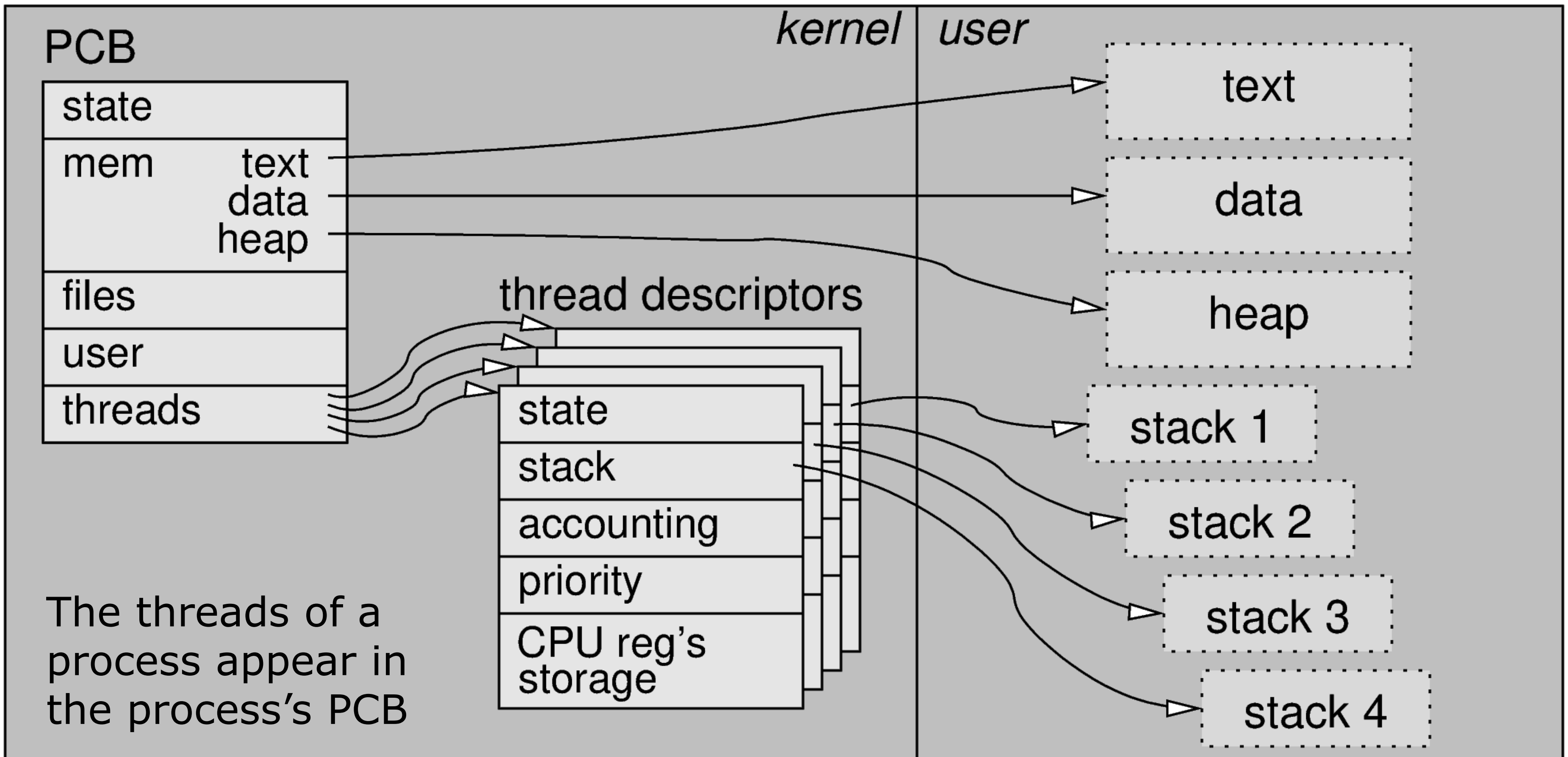
- Kernel maintains context information for the process and the threads
- Blocking a thread does not have to block the process
- **Disadvantage**: Switching between threads requires the kernel

Kernel-Level Threads

- Kernel maintains context information for the process and the threads
- Blocking a thread does not have to block the process
- **Disadvantage**: Switching between threads requires the kernel
- **Terminology**:
  - *Kernel-level threads* are managed by the kernel but run in **user** space (used for structuring the application)
  - *Kernel* threads are managed by the kernel and run in **kernel** space (used for structuring the OS)

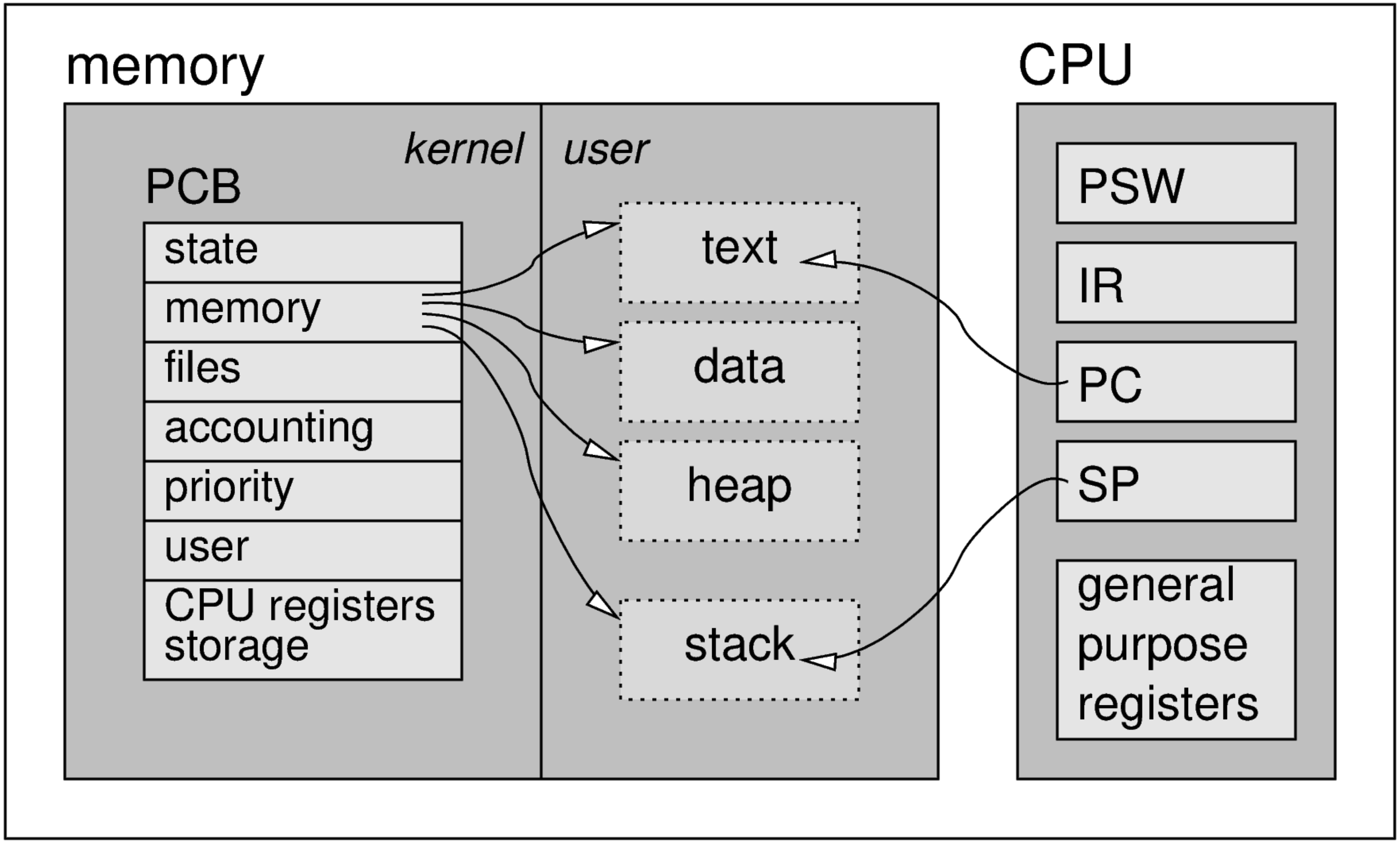
Implementing Kernel-Level Threads

A threads package managed by the kernel (very similar to processes)



Implementing Kernel-Level Threads

(For comparison a process looks like this:)



User-Level Threads

- All thread management is done by the user application/library (in user-mode)

User-Level Threads

- All thread management is done by the user application/library (in user-mode)
- The kernel is not aware of the existence of threads

User-Level Threads

- All thread management is done by the user application/library (in user-mode)
- The kernel is not aware of the existence of threads
- Thread switching does not require kernel mode privileges

User-Level Threads

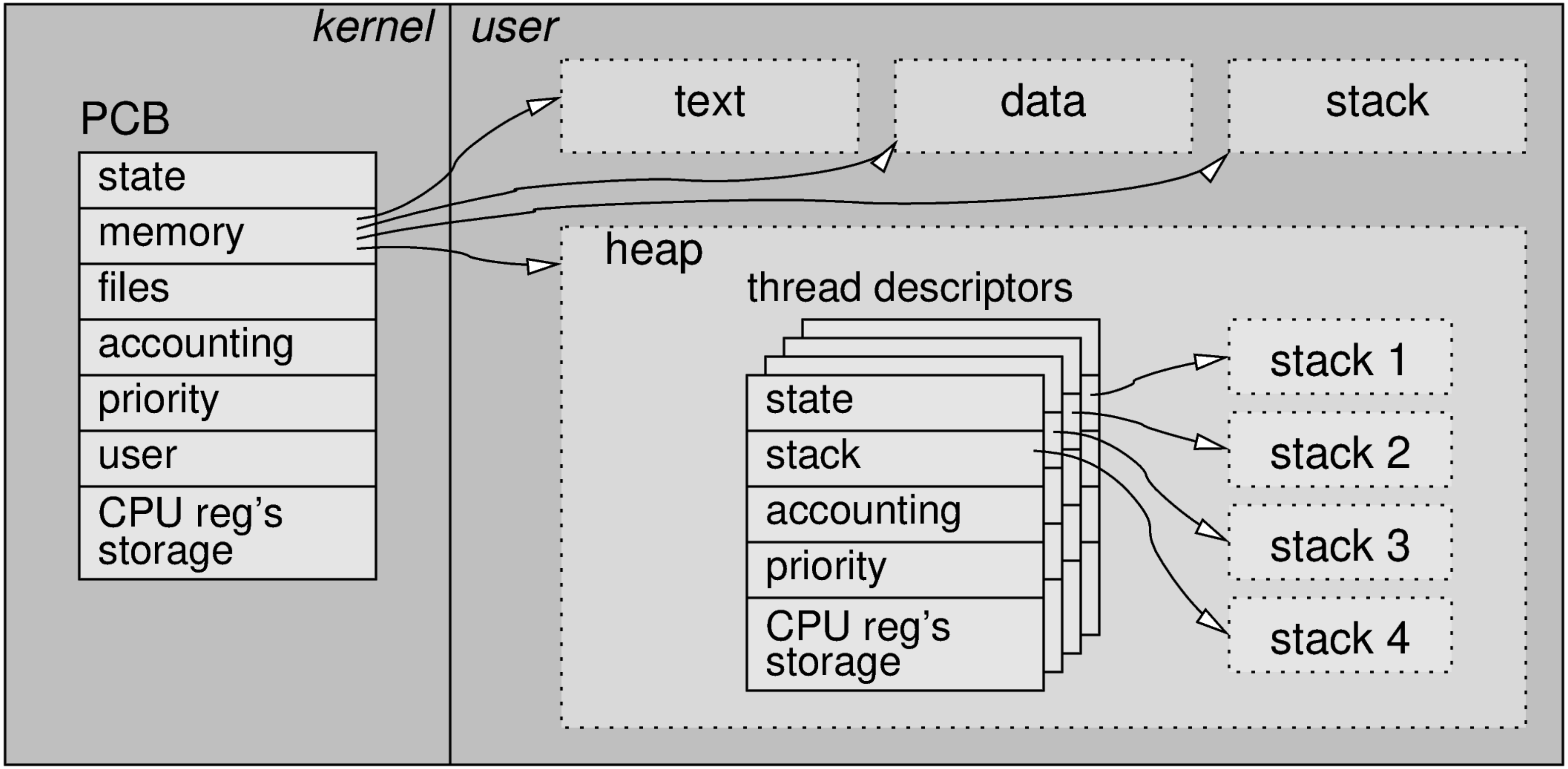
- All thread management is done by the user application/library (in user-mode)
- The kernel is not aware of the existence of threads
- Thread switching does not require kernel mode privileges
- Scheduling is application specific

User-Level Threads

- All thread management is done by the user application/library (in user-mode)
- The kernel is not aware of the existence of threads
- Thread switching does not require kernel mode privileges
- Scheduling is application specific
- <u>Major Problems:</u>
  - System calls (including I/O) by threads <u>block the process</u>
  - A single thread can monopolize the time-slice thus starving the other threads within the task

Implementing Threads in User Space

A user-level threads package



Summary: Benefits of Threads

- 30-100 times faster to **create** a new thread than a process
- Less time to **terminate** a thread than a process
- ~5 times faster to **switch** between two threads within the same process
- Threads within the same process share memory and files ➔ they can **communicate** without invoking the kernel

Summary: Threads vs. Processes

processes	*Kernel-level threads*	*User-level threads*
protected from each other, require operating system to communicate	share address space, simple communication, useful for application structuring	
high overhead: all operations require a kernel trap, significant work	medium overhead: operations require a kernel trap, but little work	low overhead: everything is done at user level
independent: if one blocks, this does not affect the others		if a thread blocks the whole process is blocked
can run in parallel on different processors in a multiprocessor		all share the same processor so only one runs at a time
system specific API, programs are not portable		the same thread library may be available on several systems
one size fits all		application-specific thread management is possible

Is the CPU "aware" of the existence of:

- User-level threads
- kernel-level threads
- Both user- and kernel-level threads
- Neither user- nor kernel-level threads

"Hardware-Level" Threads: Hyper-Threading

- Intel's technology since Pentium 4 (2002)
  - Single-core, single-processor

“Hardware-Level” Threads: Hyper-Threading

- Intel’s technology since Pentium 4 (2002)
  - Single-core, single-processor
- Appears to the OS as if there are several CPUs, although there is one

"Hardware-Level" Threads: Hyper-Threading

- Intel's technology since Pentium 4 (2002)
  - Single-core, single-processor
- Appears to the OS as if there are several CPUs, although there is one
- Management, switching between the hyper-threads is done by the CPU in hardware
  - More efficient than kernel-level threads (and user-level threads)

"Hardware-Level" Threads: Hyper-Threading

- Intel's technology since Pentium 4 (2002)
  - Single-core, single-processor
- Appears to the OS as if there are several CPUs, although there is one
- Management, switching between the hyper-threads is done by the CPU in hardware
  - More efficient than kernel-level threads (and user-level threads)
- Today, usually hyper-threading provide 2 hyper-thread for each core
  - 4-core machine with hyper-threading appears to the OS as 8-core machine

Process Creation

- When you login a process is created
- This process runs a shell / desktop GUI
- When you type a command, the shell creates a new process to run it
- When you double click on an icon, the desktop manager creates a new process to run the app
- Any process can make a system call to create additional processes

Process Termination

- Normal exit (voluntary)
- Error exit (voluntary)
  - The process discover a fatal error :
    - **cc foo.c** when **no file foo.c**
- Fatal error (involuntary)
  - Dividing by zero
- Killed by another process (involuntary)
  - Using signals, as you have seen in the tutorial