OPERATING SYSTEMS

PROCESS MANAGEMENT

Part 2: Process Management

Chapter 3: Process Concept

WHY DO WE HAVE PROCESSES

Early computers, with limited resources, and no multiprocessing …

Load the entire program into memory, and run

WHY DO WE HAVE PROCESSES

Early computers, with limited resources, and no multiprocessing …

Load the entire program into memory, and run

Shared and/or Multiprocessor Computers …

Multiple programs running at the same time

WHY DO WE HAVE PROCESSES

Early computers, with limited resources, and no multiprocessing …

Load the entire program into memory, and run

Drawbacks/Advantages

Shared and/or Multiprocessor Computers …

Multiple programs running at the same time

Drawbacks/Advantages

WHY DO WE HAVE PROCESSES

Early computers, with limited resources, and no multiprocessing …

Load the entire program into memory, and run

Drawbacks/Advantages

- Only 1 program running at any one time
- No sharing of resources
- No security protocols
- Program in control of all architecture elements

Shared and/or Multiprocessor Computers …

Multiple programs running at the same time

Drawbacks/Advantages

WHY DO WE HAVE PROCESSES

Early computers, with limited resources, and no multiprocessing …

Load the entire program into memory, and run

Drawbacks/Advantages

- Only 1 program running at any one time
- No sharing of resources
- No security protocols
- Program in control of all architecture elements

Shared and/or Multiprocessor Computers …

Multiple programs running at the same time

Drawbacks/Advantages

- Shared CPUs
- Concurrent processes
- One program shouldn't see or interfere with other programs

WHY DO WE HAVE PROCESSES

- Q: Can a single program utilize multiple processes?

Early computers, with limited resources, and no multiprocessing …

Load the entire program into memory, and run

Drawbacks/Advantages

- Only 1 program running at any one time
- No sharing of resources
- No security protocols
- Program in control of all architecture elements

Shared and/or Multiprocessor Computers …

Multiple programs running at the same time

Drawbacks/Advantages

- Shared CPUs
- Concurrent processes
- One program shouldn't see or interfere with other programs

WHY DO WE HAVE PROCESSES

- Q: Can a single program utilize multiple processes?
- Worksheet Q2: What could be the advantages of using multiple processes for a single program?

Early computers, with limited resources, and no multiprocessing …

Load the entire program into memory, and run

Drawbacks/Advantages

- Only 1 program running at any one time
- No sharing of resources
- No security protocols
- Program in control of all architecture elements

Shared and/or Multiprocessor Computers …

Multiple programs running at the same time

Drawbacks/Advantages

- Shared CPUs
- Concurrent processes
- One program shouldn't see or interfere with other programs

WHY DO WE HAVE PROCESSES

- Q: Can a single program utilize multiple processes?
- What could be the advantages?
  - Performance
  - Modularity
  - Reliability
  - Scheduling

Early computers, with limited resources, and no multiprocessing …

Load the entire program into memory, and run

Drawbacks/Advantages

- Only 1 program running at any one time
- No sharing of resources
- No security protocols
- Program in control of all architecture elements

Shared and/or Multiprocessor Computers …

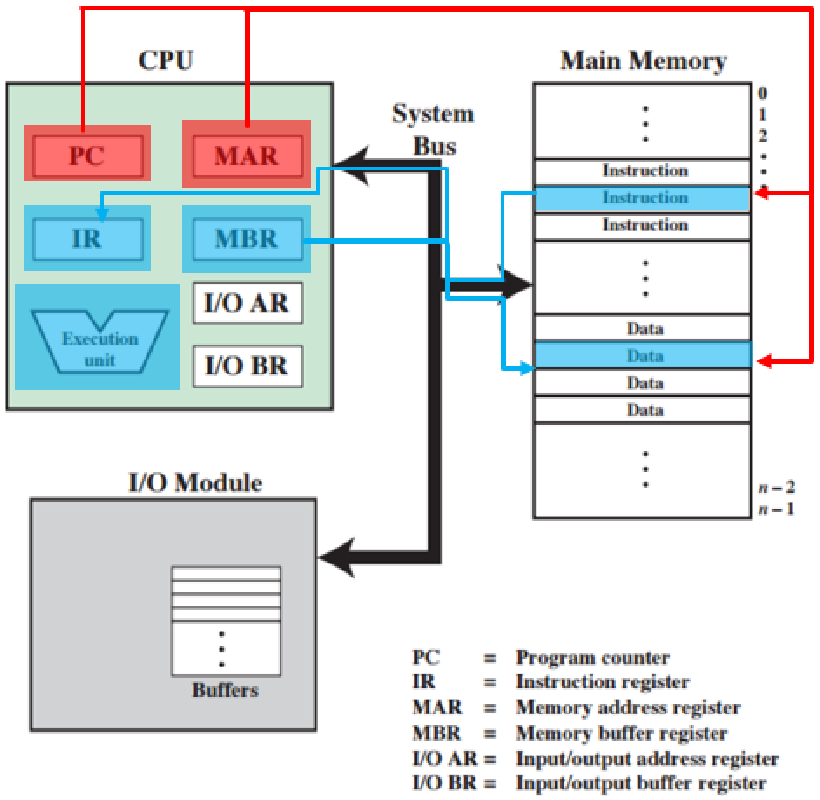
Multiple programs running at the same time

Drawbacks/Advantages

- Shared CPUs
- Concurrent processes
- One program shouldn't see or interfere with other programs

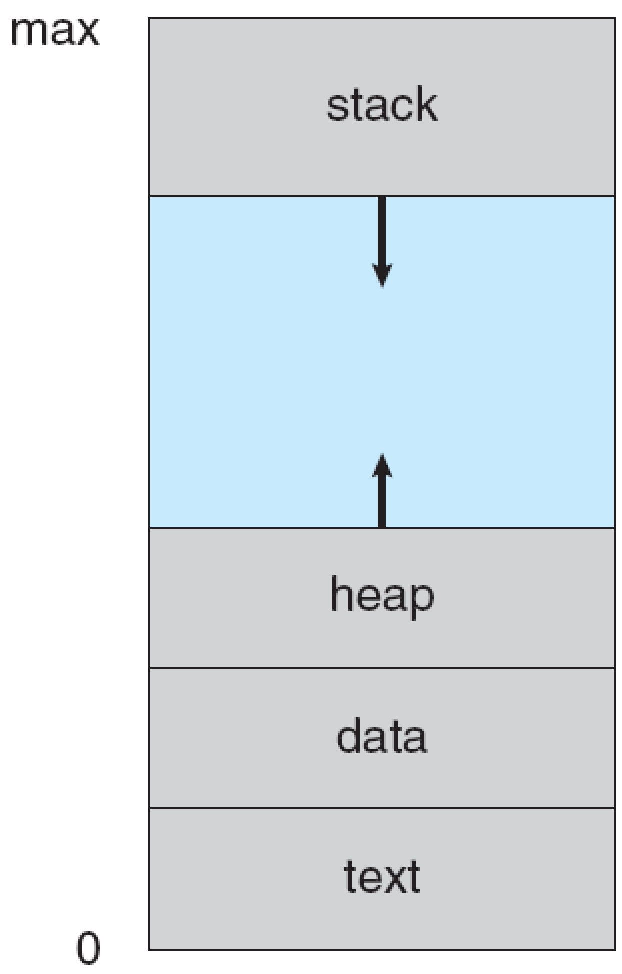
PROCESS

- A process is a program in execution. It is a unit of work within the system.
- Program is a ***passive entity***, process is an ***active entity***.



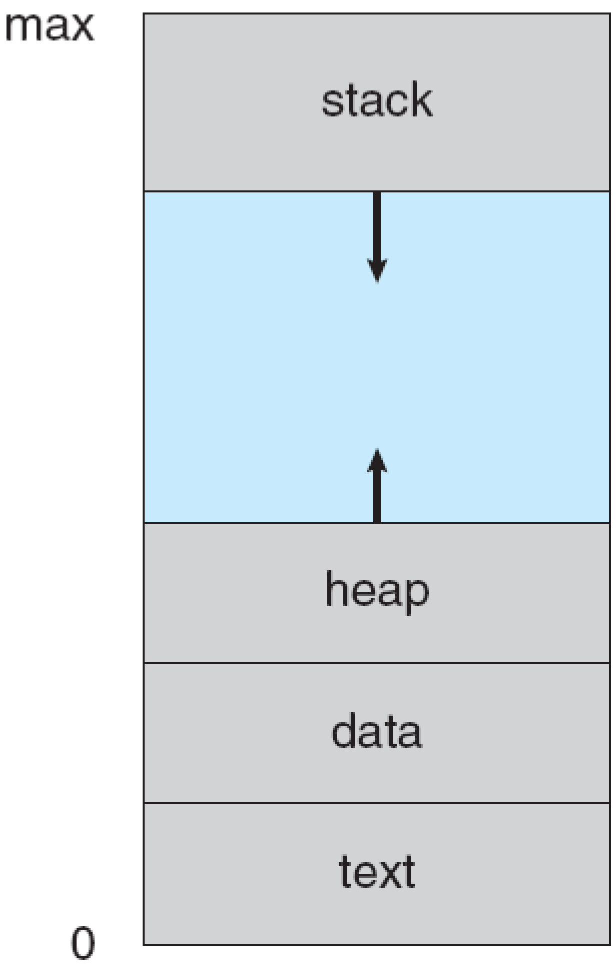
PROCESS MEMORY

- The program code, also called **text section**



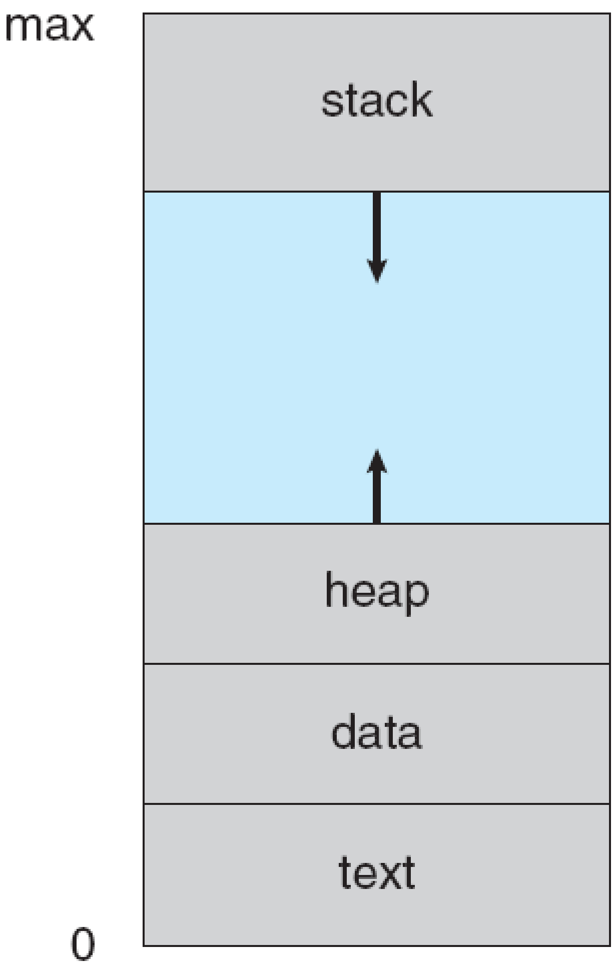
PROCESS MEMORY

- The program code, also called **text section**
- **Stack** containing temporary data
  - Function parameters, return addresses, local variables



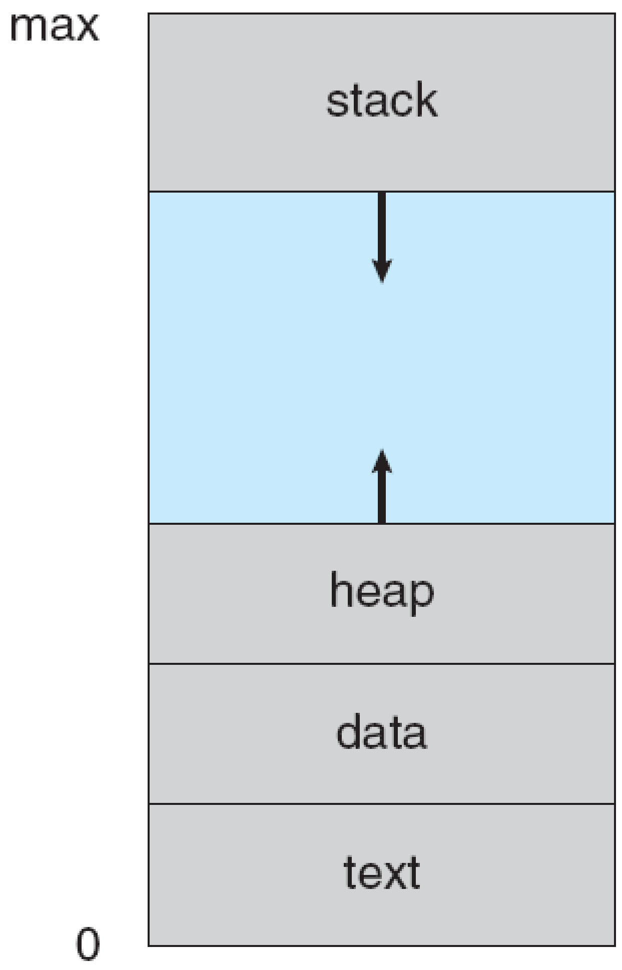
PROCESS MEMORY

- The program code, also called **text section**
- **Stack** containing temporary data
  - Function parameters, return addresses, local variables
- **Data section** containing global variables

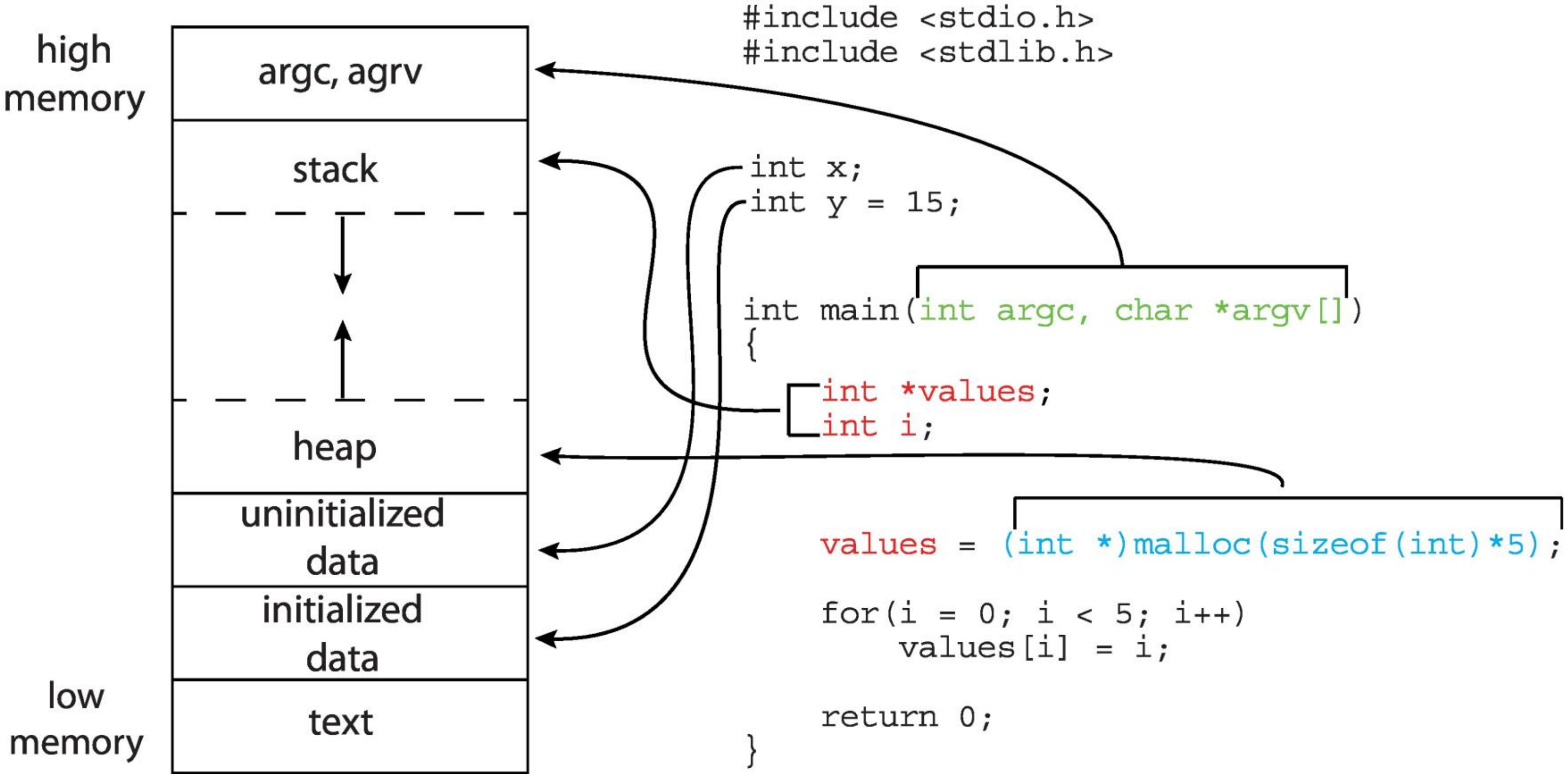


PROCESS MEMORY

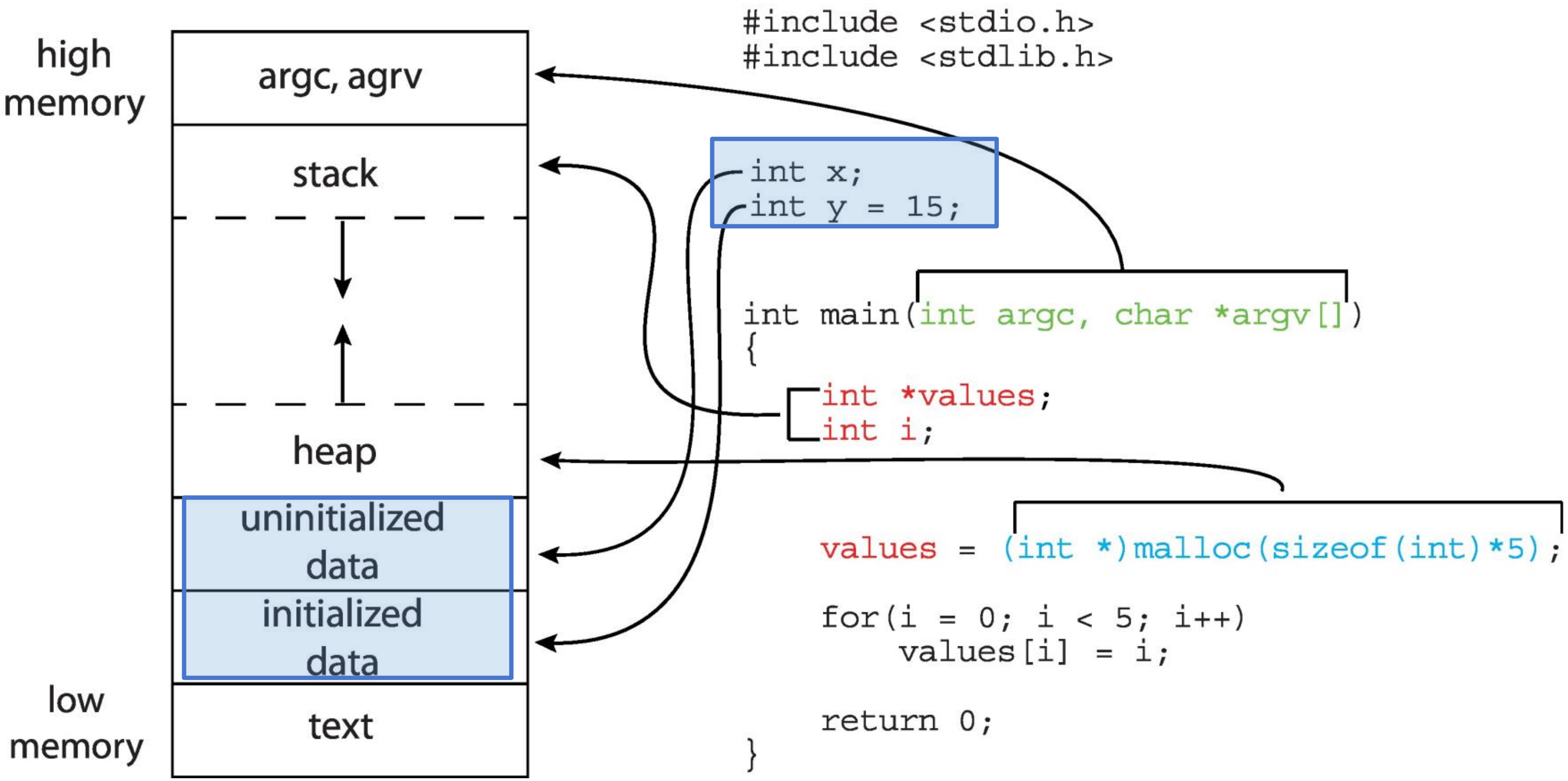
- The program code, also called **text section**
- **Stack** containing temporary data
  - Function parameters, return addresses, local variables
- **Data section** containing global variables
- **Heap** containing memory dynamically allocated during run time



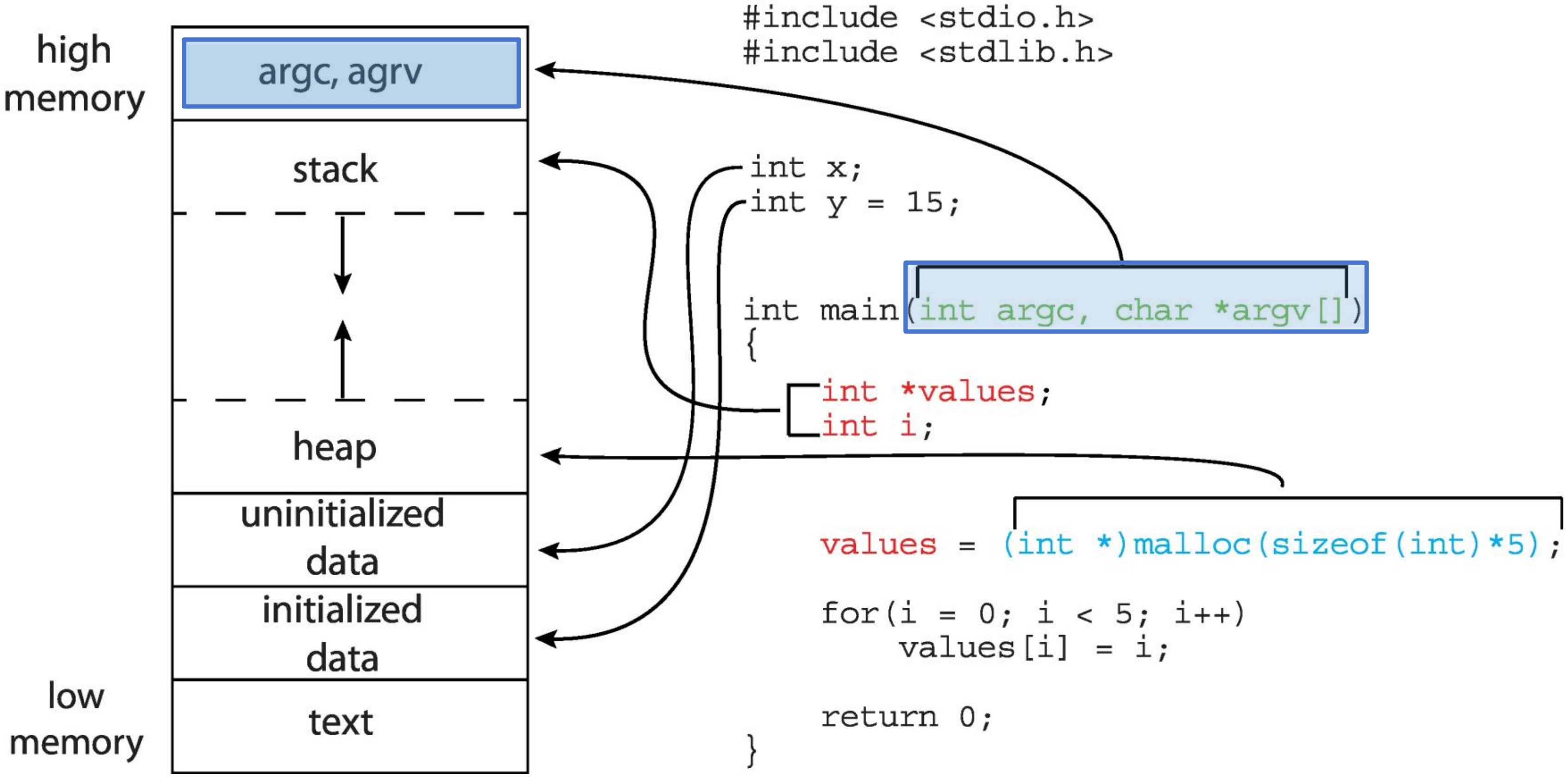
PROCESS MEMORY



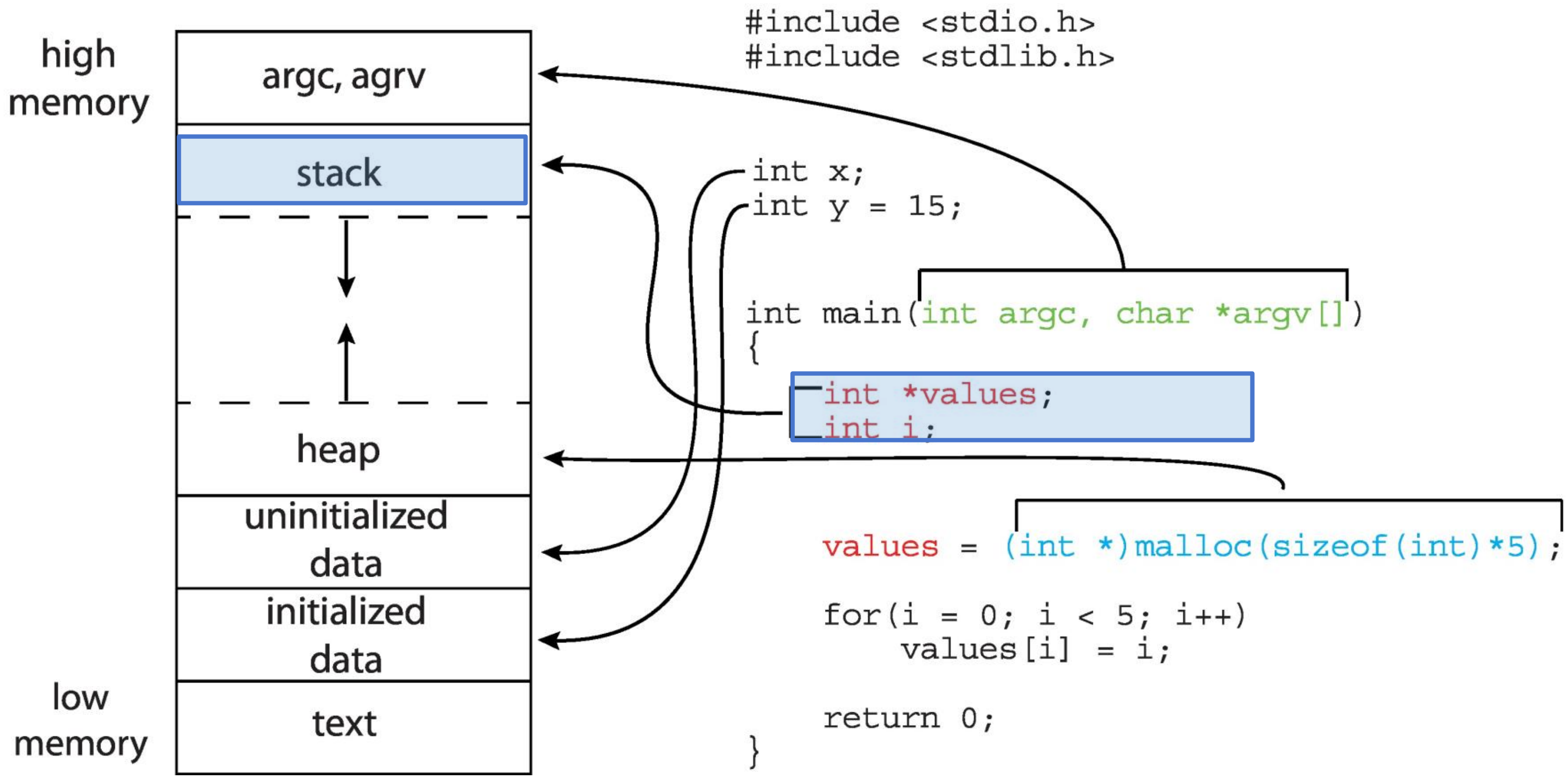
PROCESS MEMORY



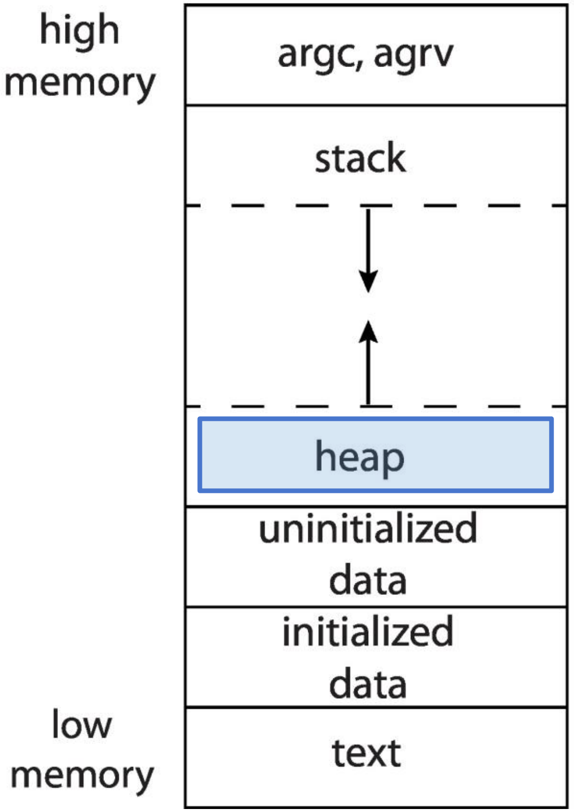
PROCESS MEMORY



PROCESS MEMORY



PROCESS MEMORY




```c
#include <stdio.h>
#include <stdlib.h>

int x;
int y = 15;

int main(int argc, char *argv[])
{
    int *values;
    int i;

    values = (int *)malloc(sizeof(int)*5);

    for(i = 0; i < 5; i++)
        values[i] = i;

    return 0;
}
```

PROCESS STATE

- As a process executes, it changes **state**

PROCESS STATE

- As a process executes, it changes **state**
  - **Running:** CPU has this process code running.
  - **Not Running:**

PROCESS STATE

- As a process executes, it changes **state**
  - **Running:** CPU has this process code running.
  - **Not Running:**
    - **New**
    - **Terminated**
    - **Waiting on a Resource/Event**
    - **Ready**

PROCESS STATE

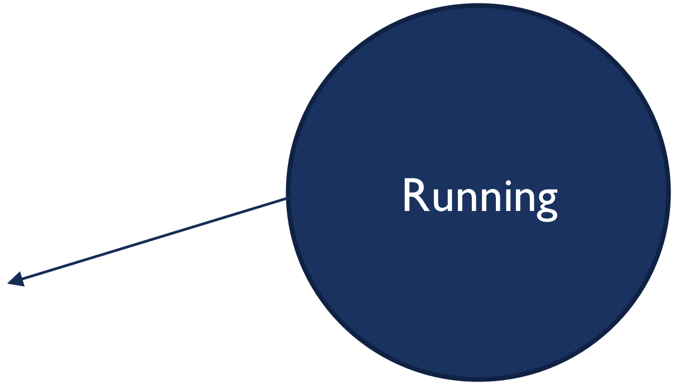
- As a process executes, it changes **state**
  - **Running:** CPU has this process code running.
  - **Not Running:**
    - **New**
    - **Terminated**
    - **Waiting on a Resource/Event**
    - **Ready**
- The scheduler, a kernel program, handles process dispatching.

CONTEXT SWITCHING

- Context Switching: Switching from one thread/process to another.
- Q:What could cause context switching?

CONTEXT SWITCHING

- Context Switching: Switching from one thread/process to another.
- Q:What could cause context switching?

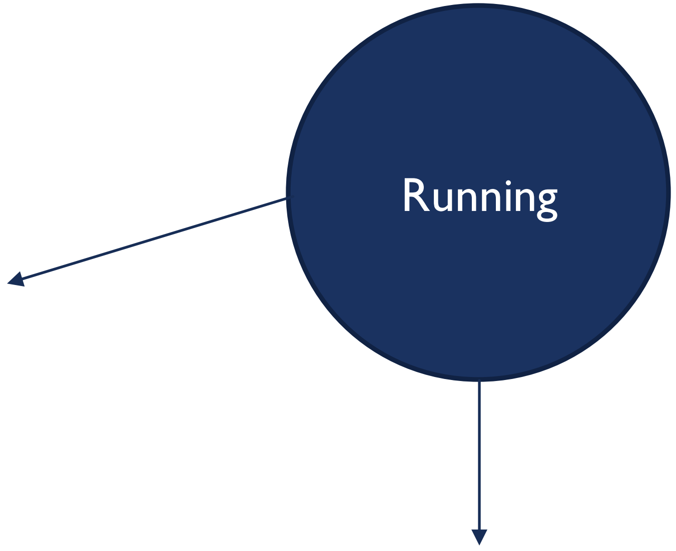


Scheduler:
process "turn" or
time slot is over,
other processes
need to run.

CONTEXT SWITCHING

- Context Switching: Switching from one thread/process to another.
- Q:What could cause context switching?

Scheduler:
process "turn" or
time slot is over,
other processes
need to run.



Process ***explicitly*** initiates
a system call to use a
kernel function.

WESTERN
WASHINGTON UNIVERSITY

CONTEXT SWITCHING

- Context Switching: Switching from one thread/process to another.
- Q: What could cause context switching?

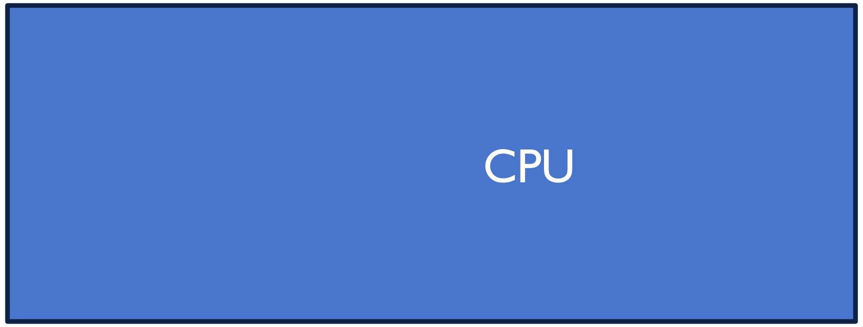
Running

Scheduler: process "turn" or time slot is over, other processes need to run.

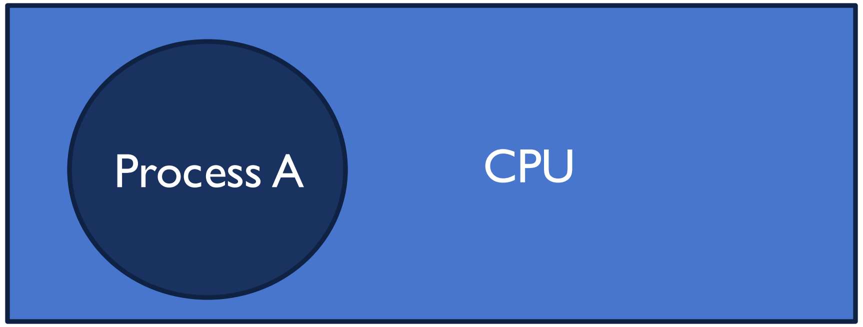
Faults/Error: Arithmetic exception, page faults, invalid address …

Process initiates a system call to use a kernel function.

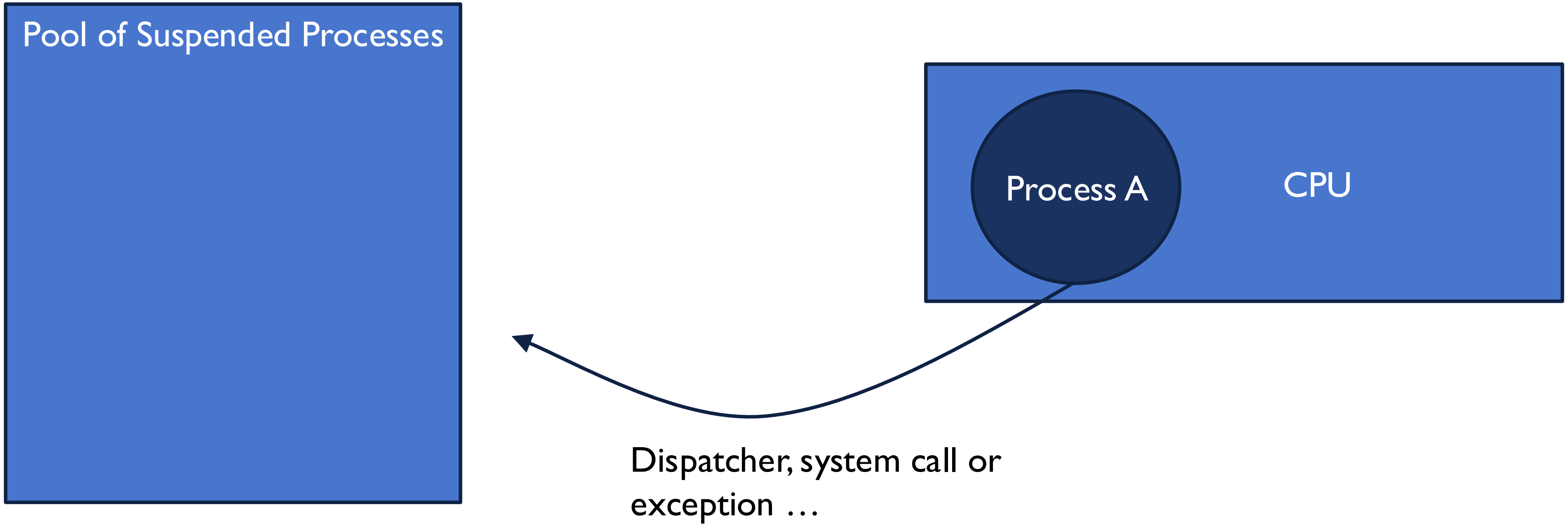
CONTEXT SWITCHING



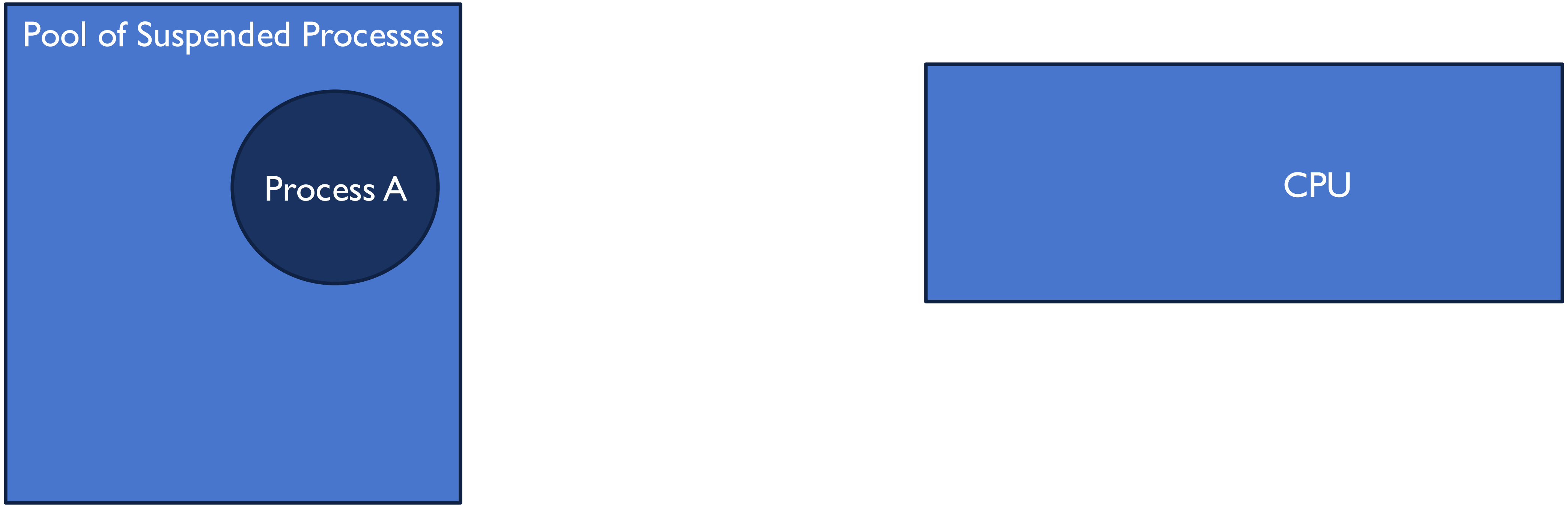
CONTEXT SWITCHING



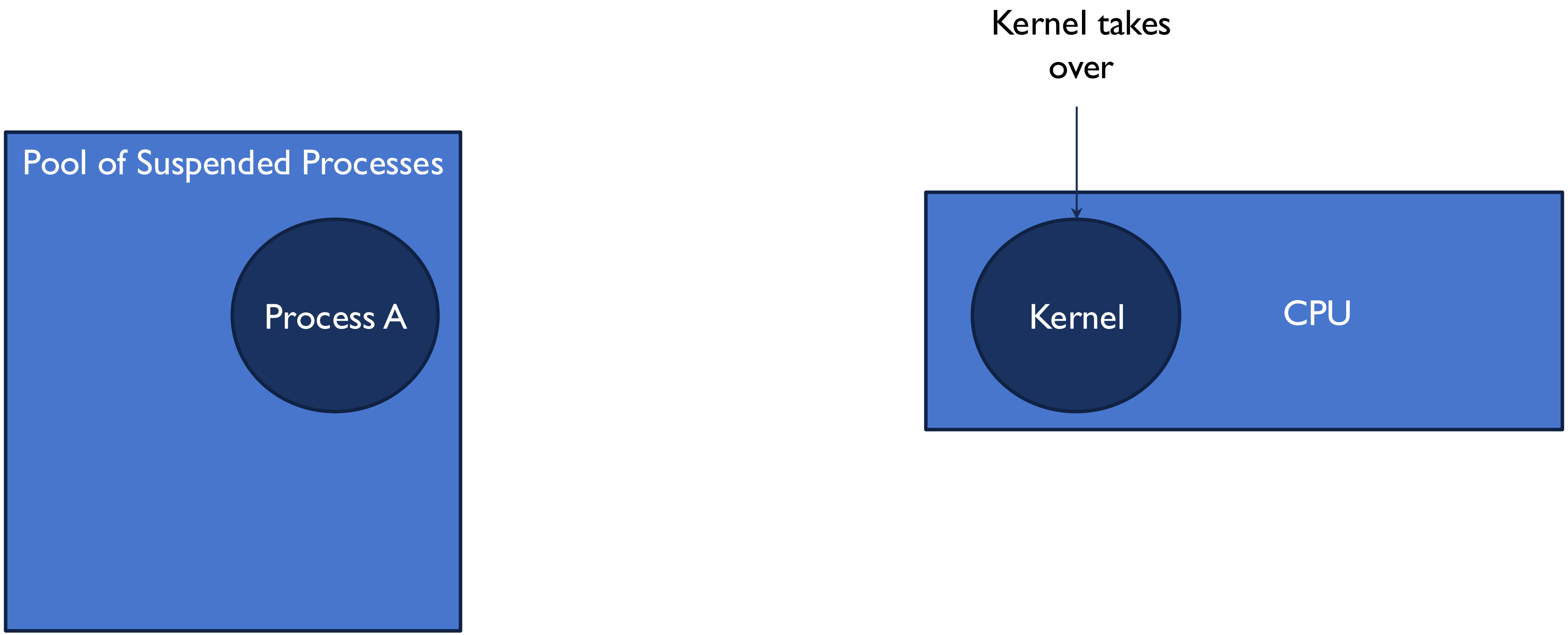
CONTEXT SWITCHING



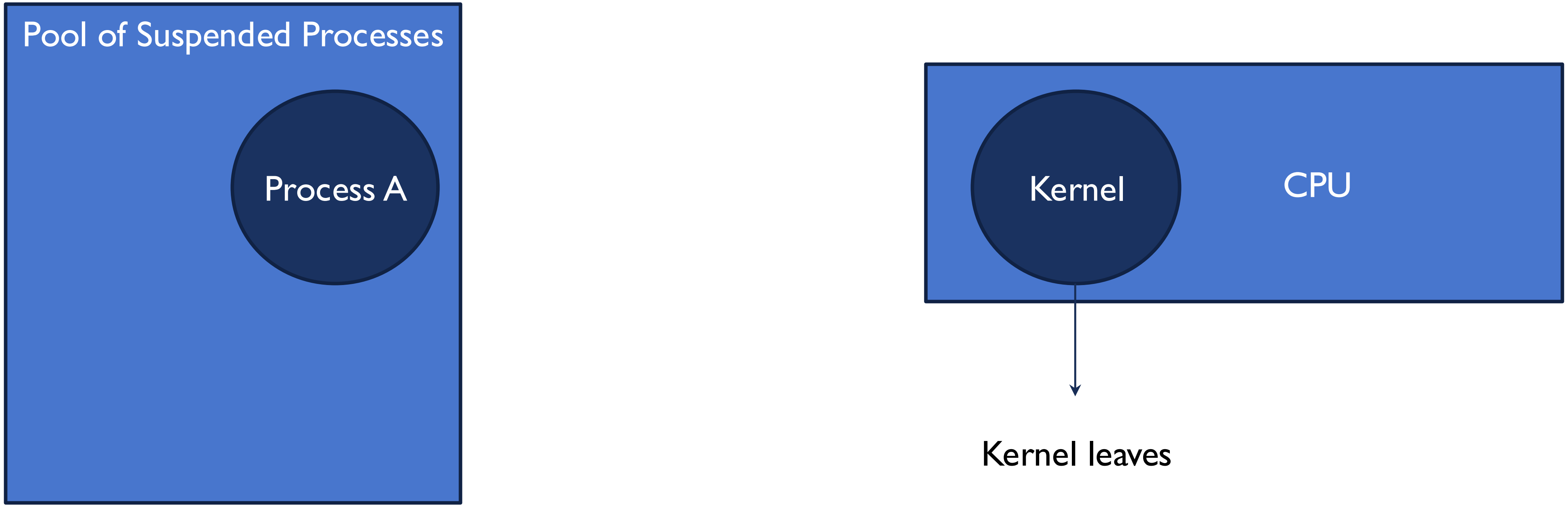
CONTEXT SWITCHING



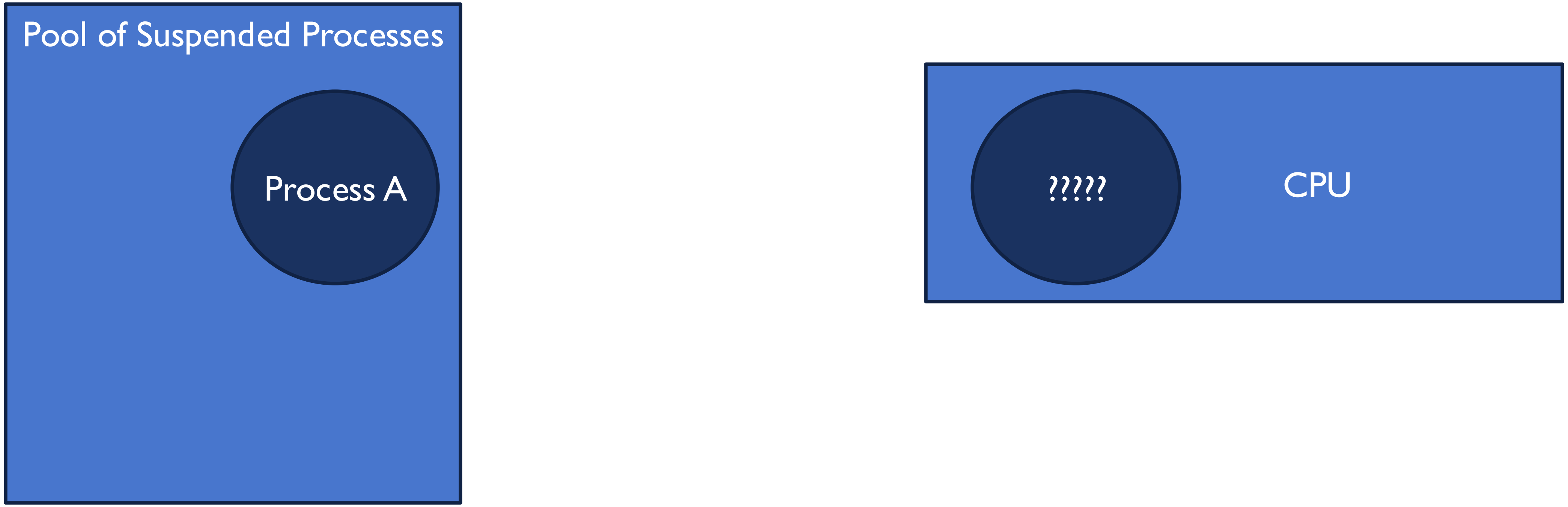
CONTEXT SWITCHING



CONTEXT SWITCHING

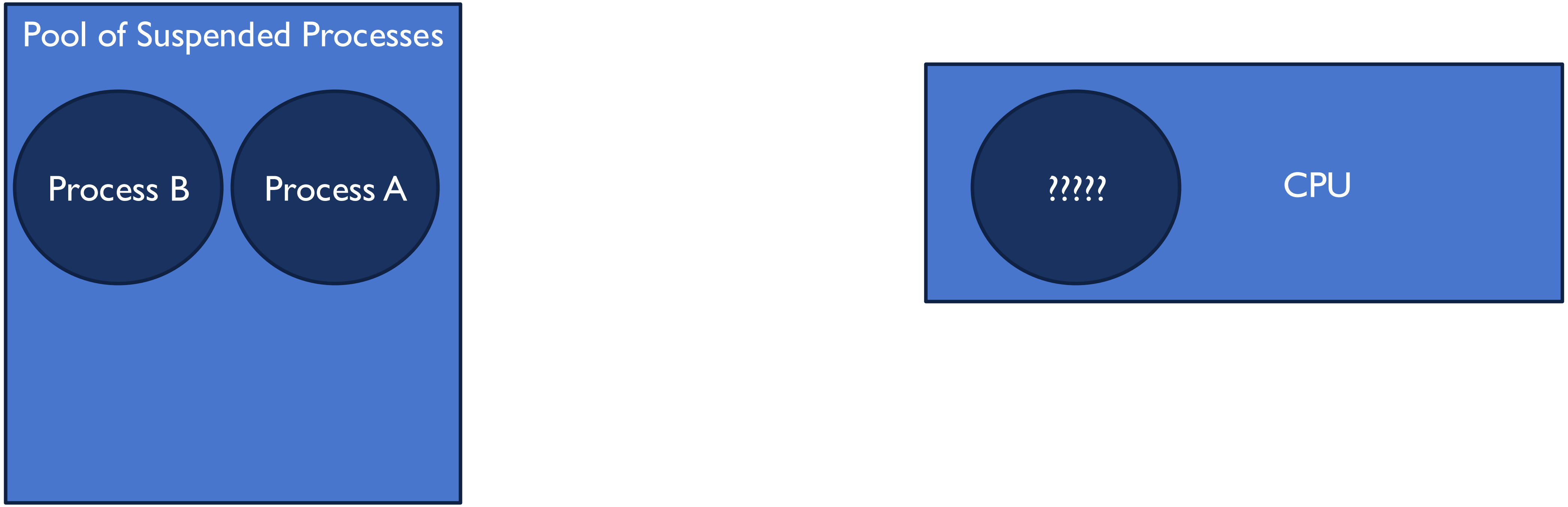


CONTEXT SWITCHING



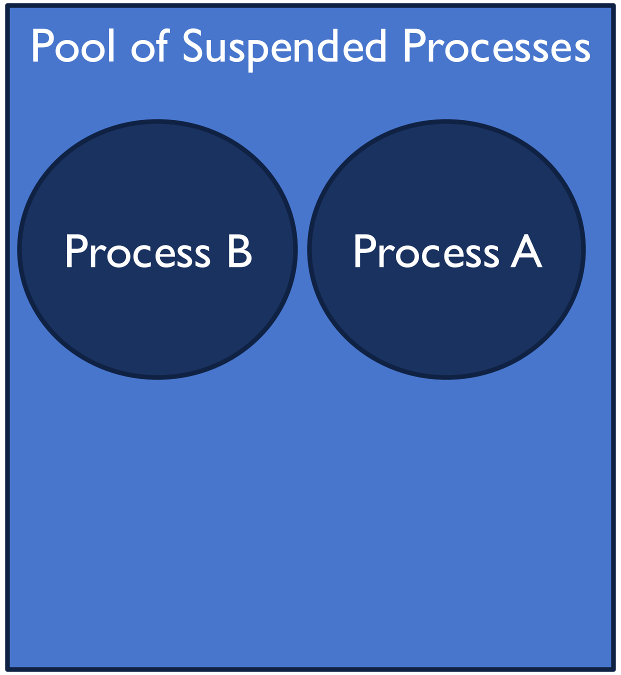
CONTEXT SWITCHING

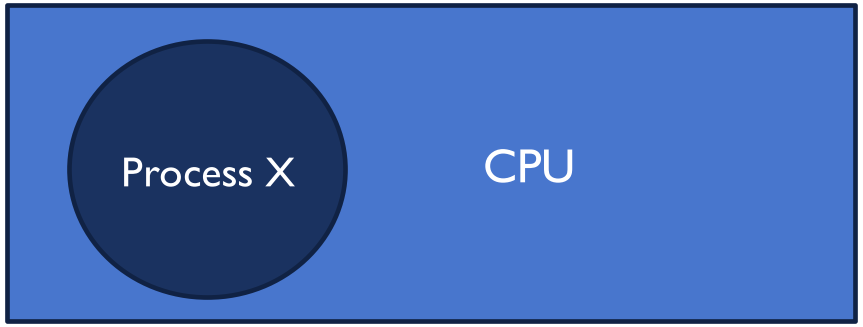
Which process should take over the CPU?



CONTEXT SWITCHING

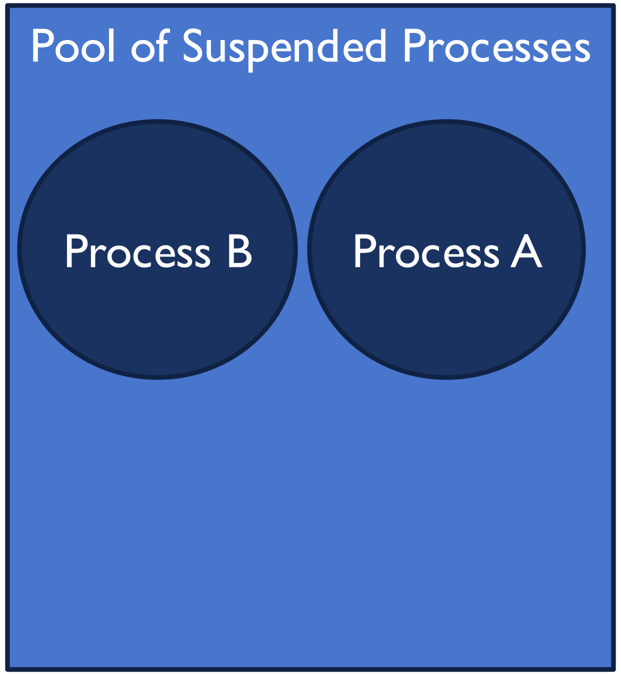
Which process should take over the CPU?





We don't know which process will take over …

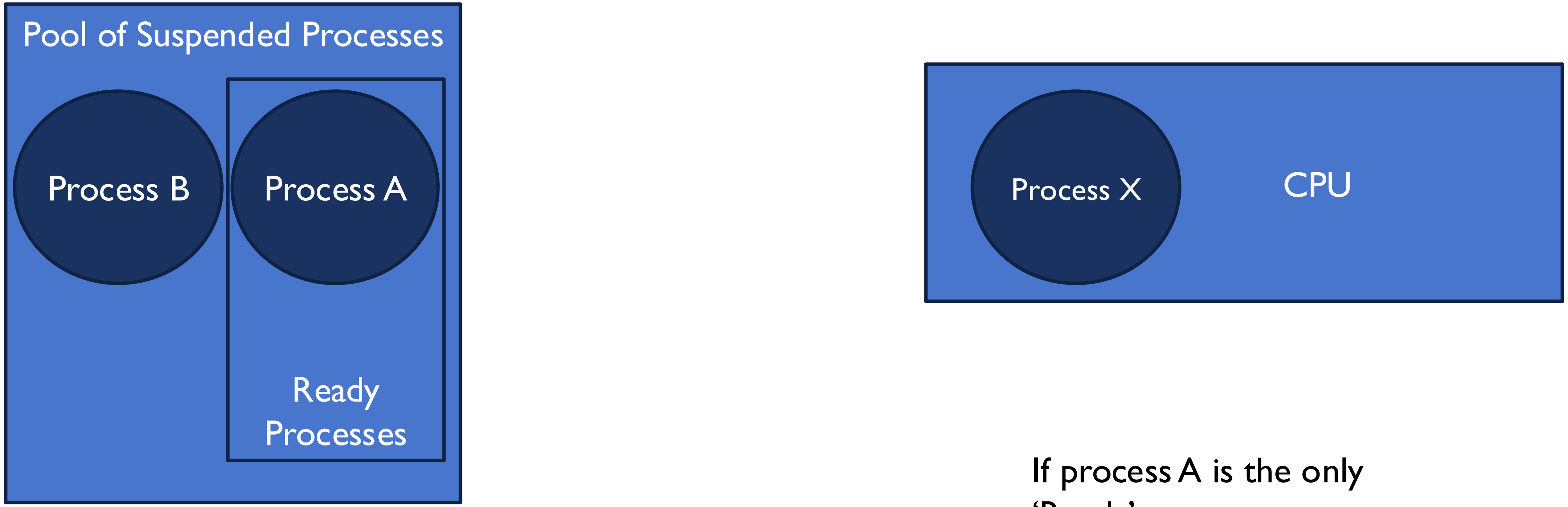
CONTEXT SWITCHING



Suspended Process can have different states:

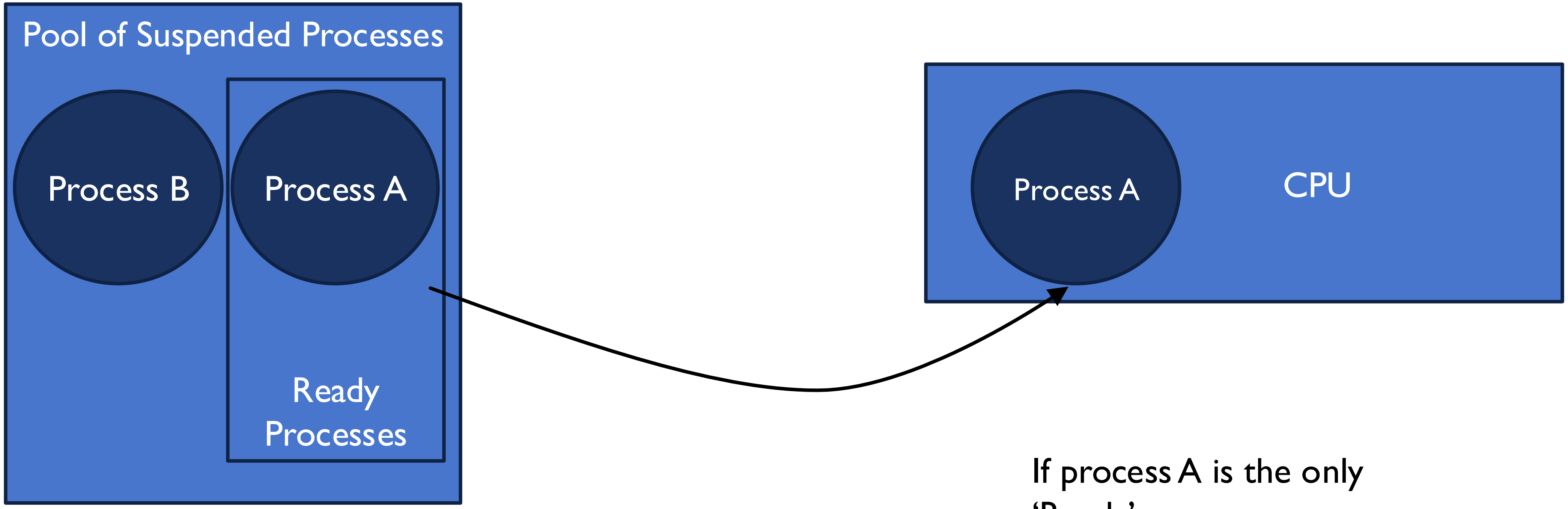
- Waiting: on resource or event …
- Ready

CONTEXT SWITCHING



If process A is the only 'Ready' process among suspended processes, then it's picked for execution.

CONTEXT SWITCHING



If process A is the only 'Ready' process among suspended processes, then it's picked for execution.

CONTEXT SWITCHING

Worksheet Question:

- All these event will result in a suspended process. Which will be in "Waiting" state and which will be in "Ready" state?

Running

1. Scheduler: process "turn" or time slot is over, other processes need to run.

Ready or Waiting?

2. Process ***explicitly*** initiates a system call to use a kernel function.

Ready or Waiting?

3. Faults/Error: Arithmetic exception, page faults, invalid address …

Ready or Waiting?

CONTEXT SWITCHING

Worksheet Question:

- All these event will result in a suspended process. Which will be in "Waiting" state and which will be in "Ready" state?

Running

1. Scheduler: process "turn" or time slot is over, other processes need to run.

Ready

2. Process ***explicitly*** initiates a system call to use a kernel function.

Ready or Waiting?

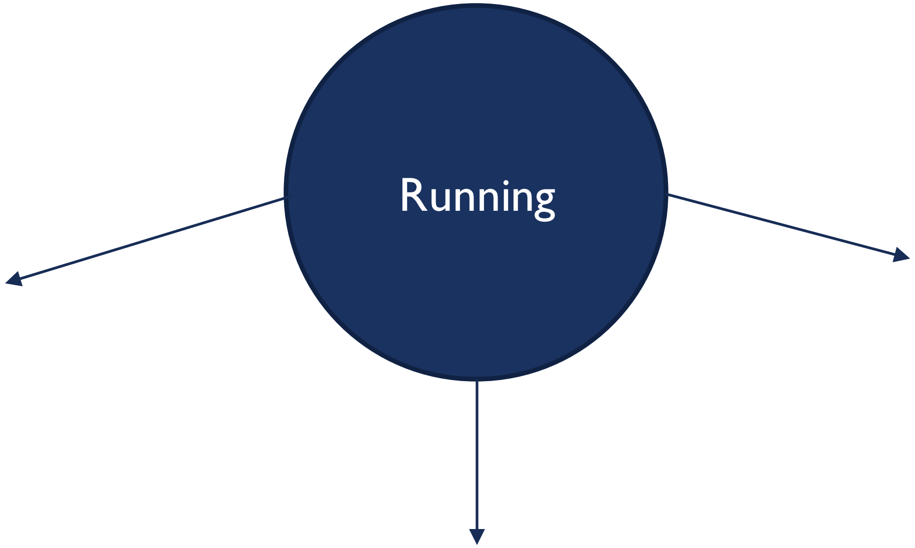
3. Faults/Error: Arithmetic exception, page faults, invalid address …

Ready or Waiting?

CONTEXT SWITCHING

Worksheet Question:

- All these event will result in a suspended process. Which will be in "Waiting" state and which will be in "Ready" state?



1. Scheduler: process "turn" or time slot is over, other processes need to run.

Ready

2. Process ***explicitly*** initiates a system call to use a kernel function.

Waiting

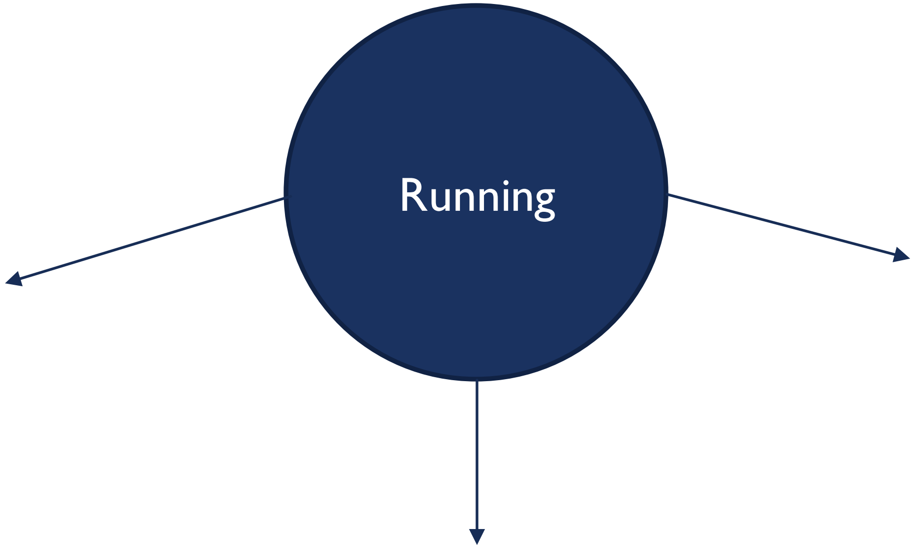
3. Faults/Error: Arithmetic exception, page faults, invalid address …

Ready or Waiting?

CONTEXT SWITCHING

Worksheet Question:

- All these event will result in a suspended process. Which will be in “Waiting” state and which will be in “Ready” state?



1. Scheduler: process “turn” or time slot is over, other processes need to run.

Ready

2. Process ***explicitly*** initiates a system call to use a kernel function.

Waiting

3. Faults/Error: Arithmetic exception, page faults, invalid address …

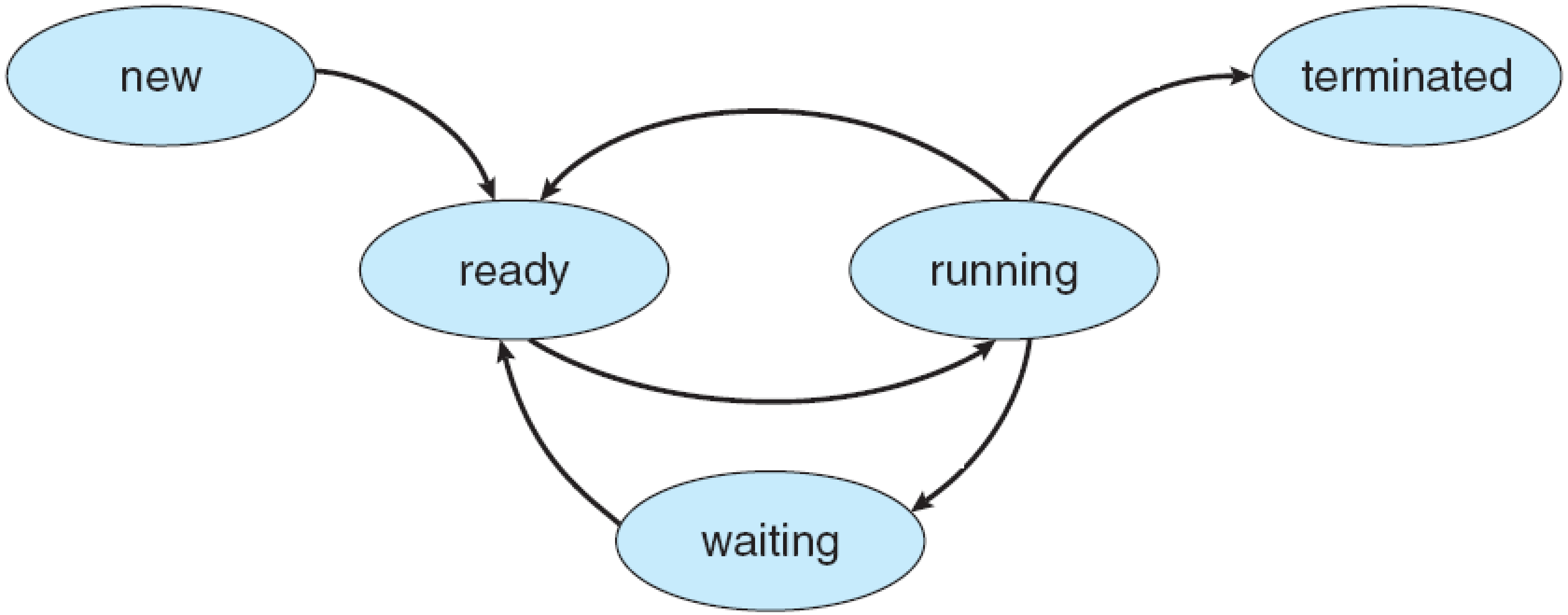
Waiting

WESTERN WASHINGTON UNIVERSITY

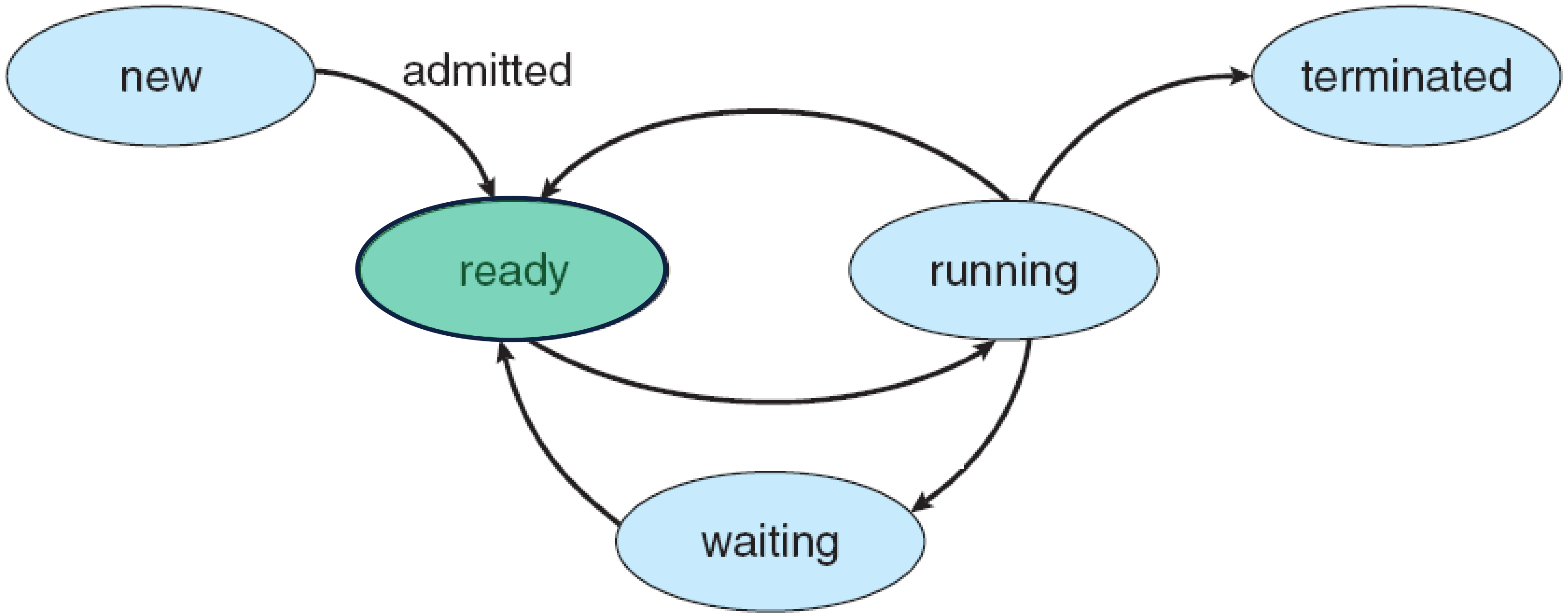
PROCESS STATE

- As a process executes, it changes **state**
  - **New**: The process is being created
  - **Ready**: The process is waiting to be assigned to a processor
  - **Waiting**: The process is waiting for some event to occur; can not execute
  - **Running**: Instructions are being executed
  - **Terminated**: The process has finished execution; waiting to be deleted and its resources released.
- The scheduler, a kernel program, handles process dispatching.

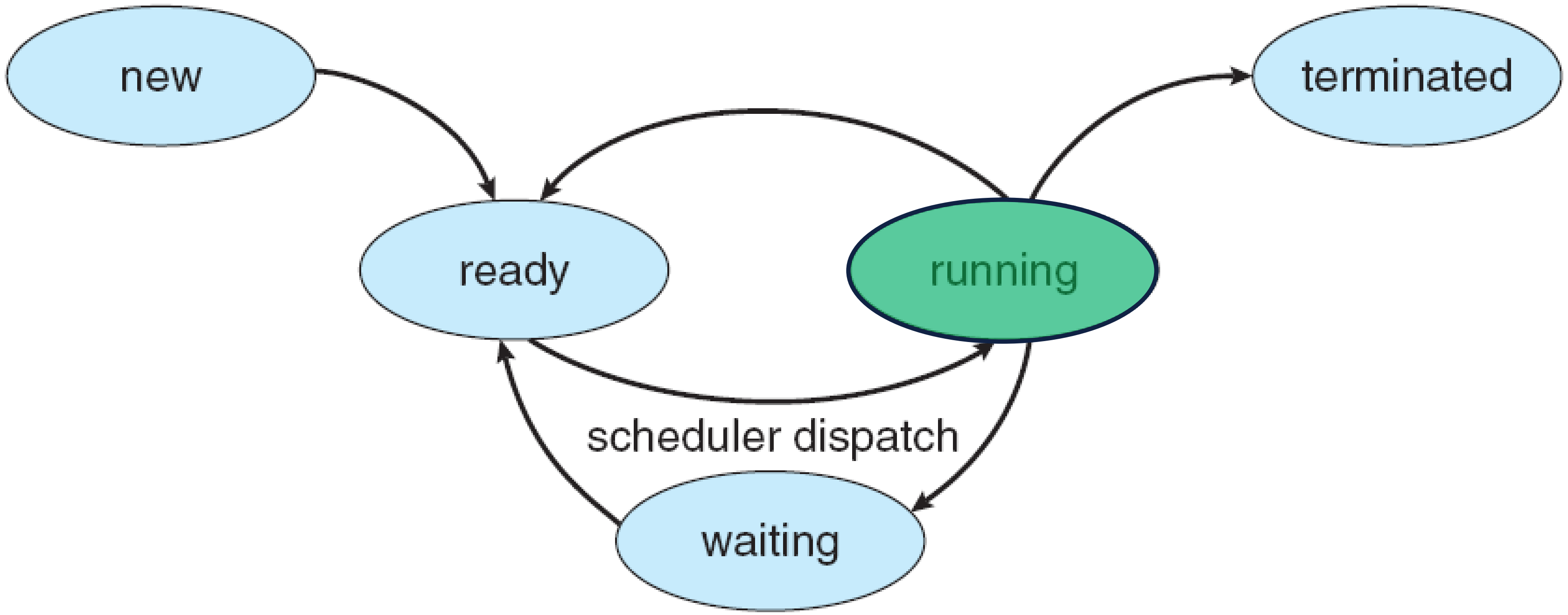
PROCESS STATE



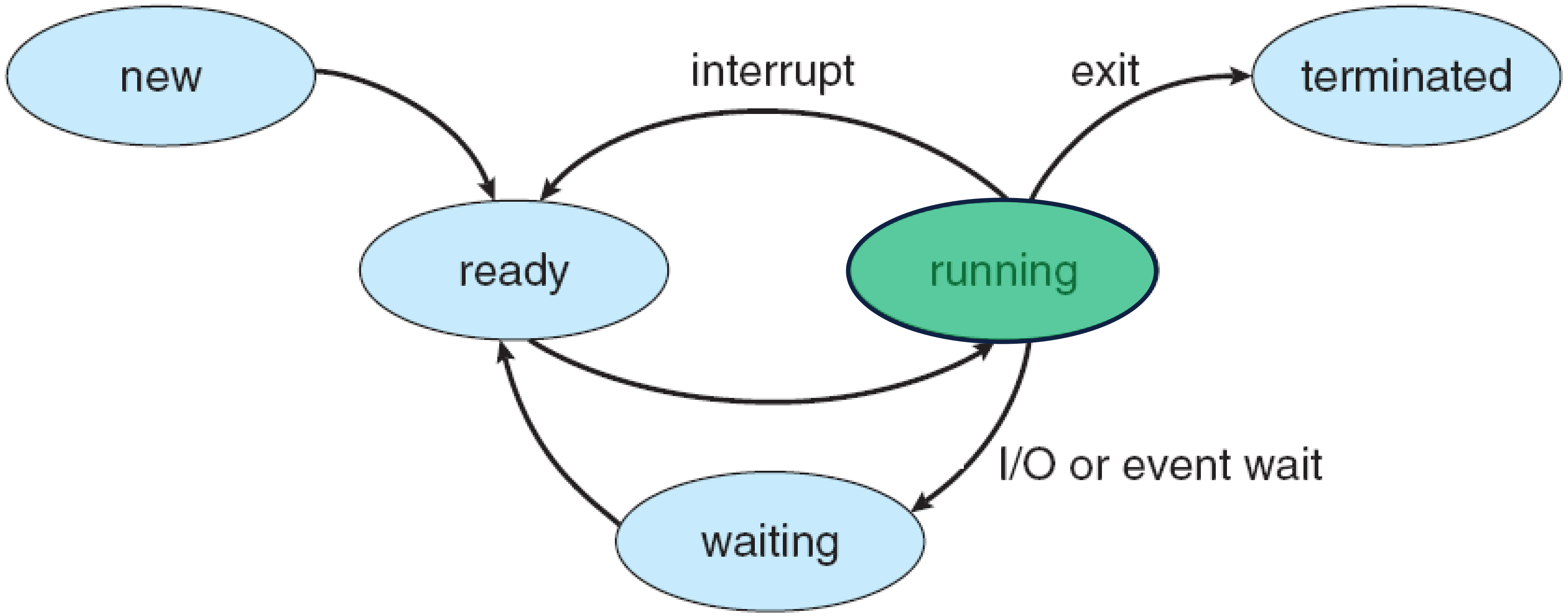
PROCESS STATE



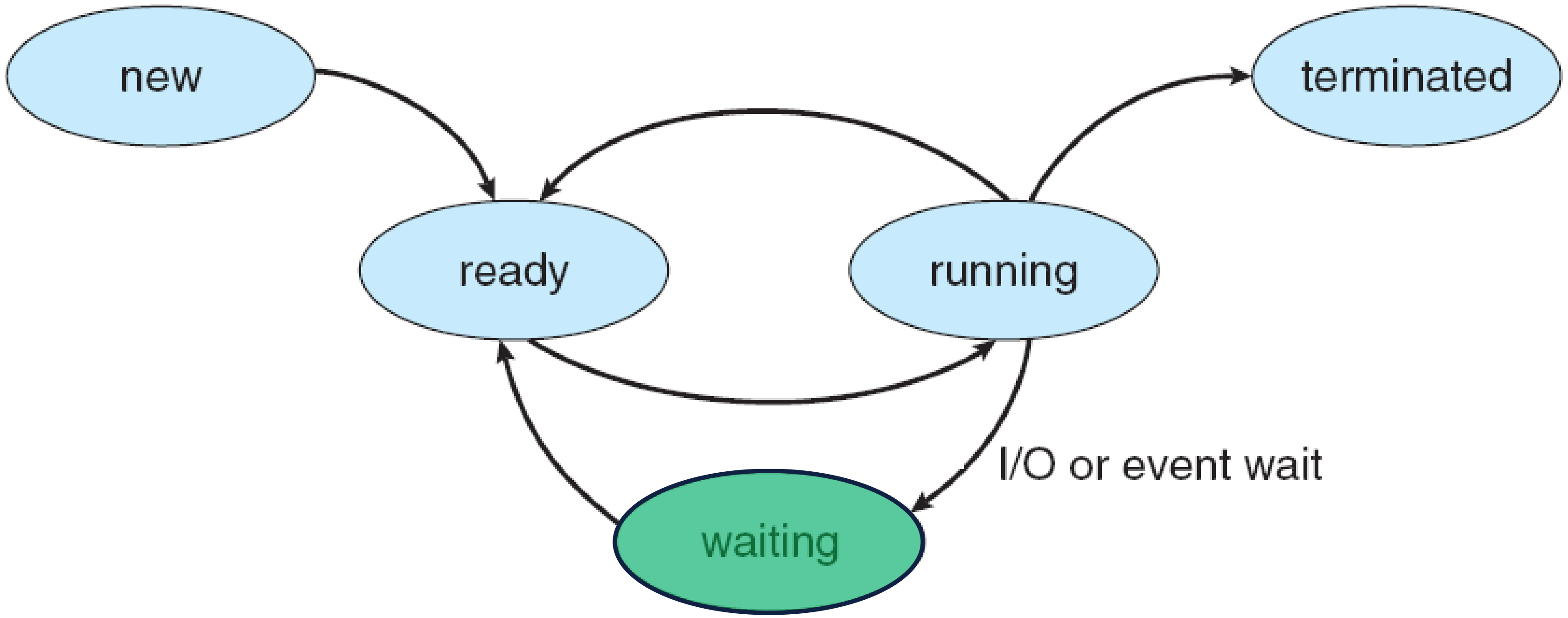
PROCESS STATE



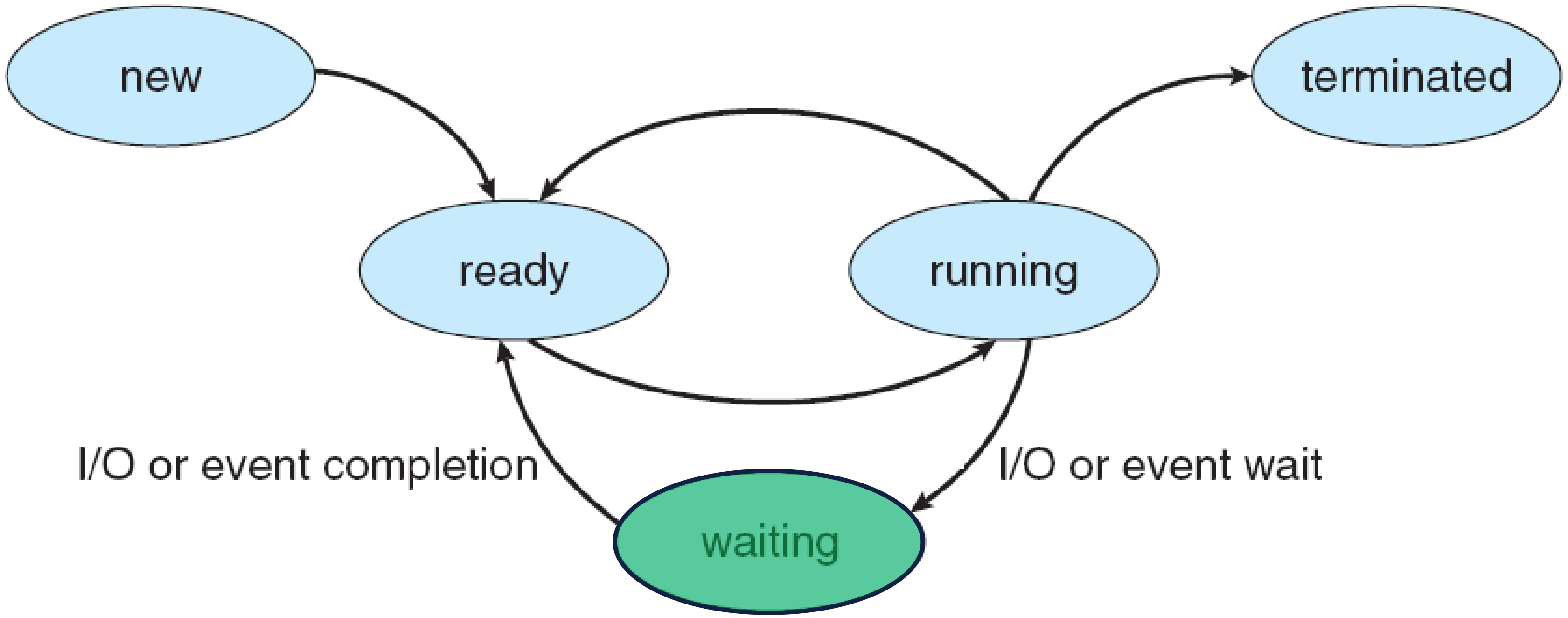
PROCESS STATE



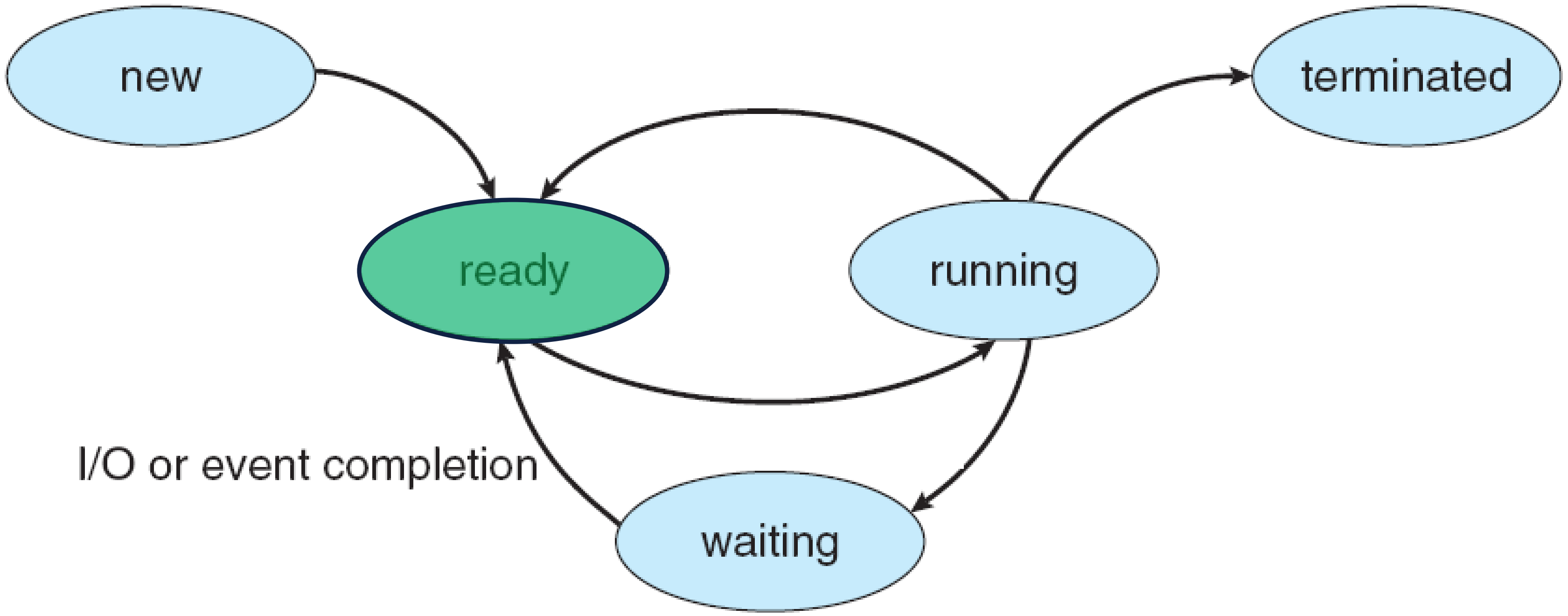
PROCESS STATE



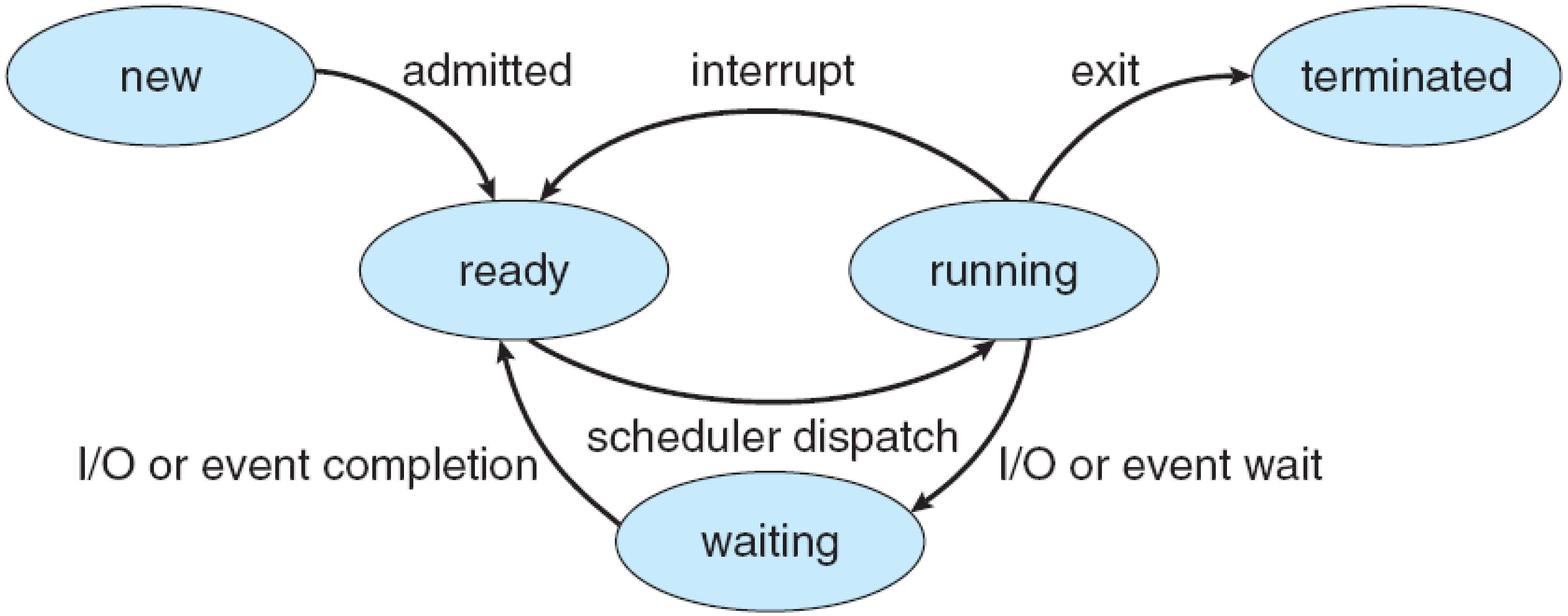
PROCESS STATE



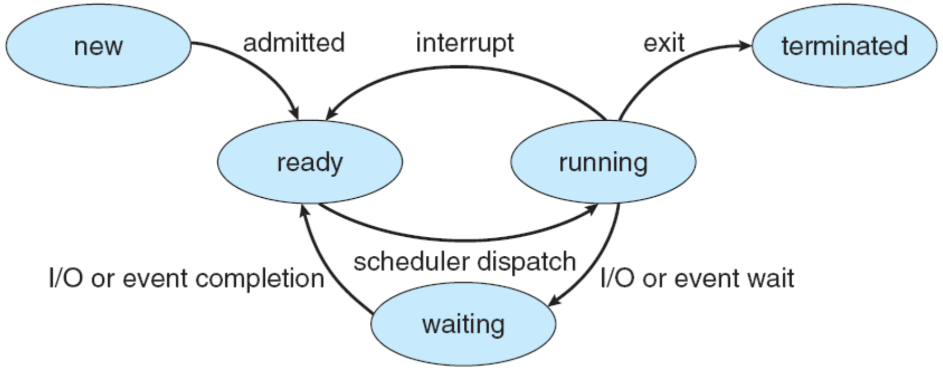
PROCESS STATE

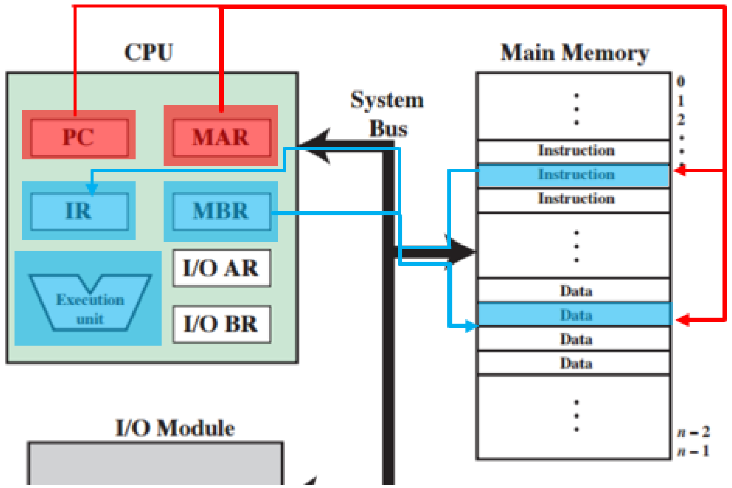


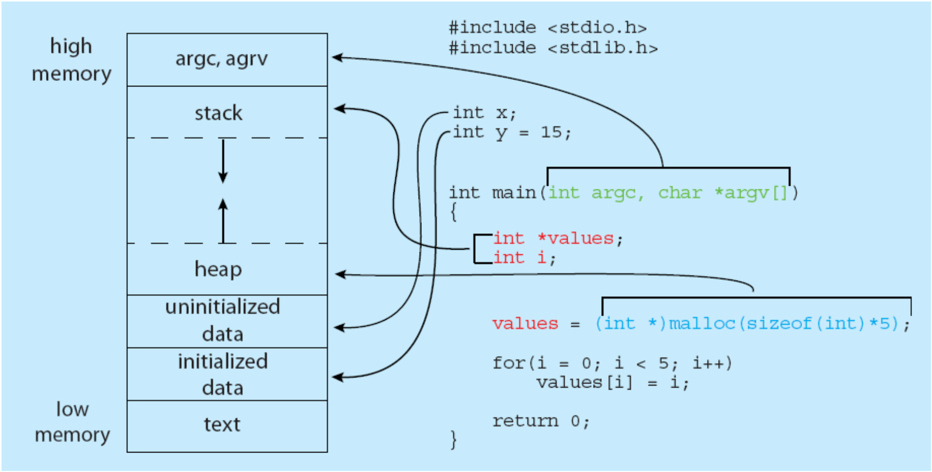
PROCESS STATE



PROCESS SNAPSHOT

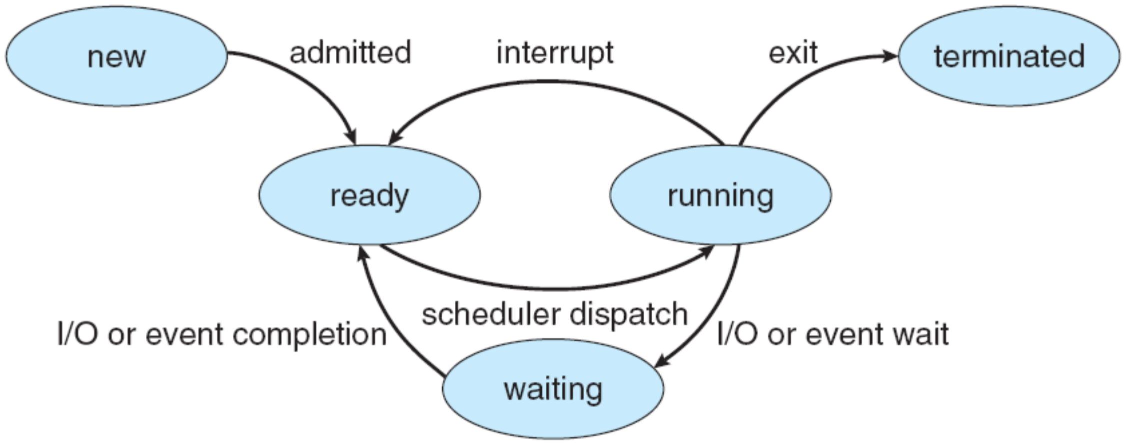






PROCESS CONTROL BLOCK: PROCESS SNAPSHOT

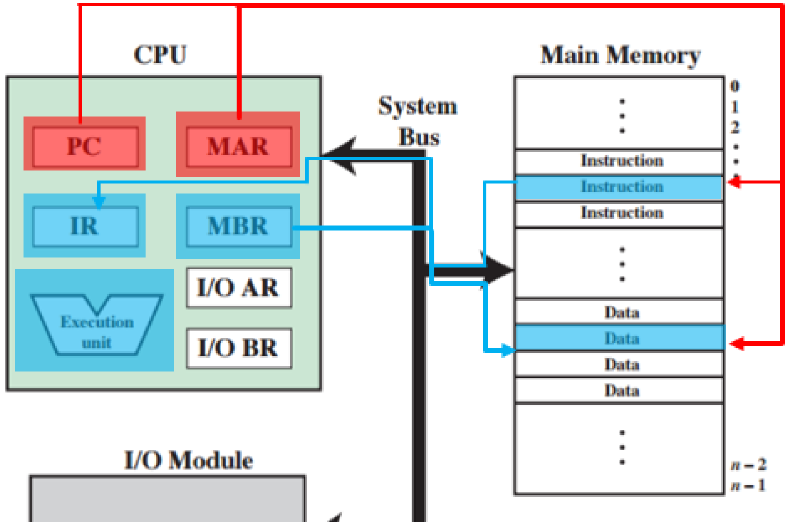
- One of a several current states



State

PROCESS CONTROL BLOCK: PROCESS SNAPSHOT

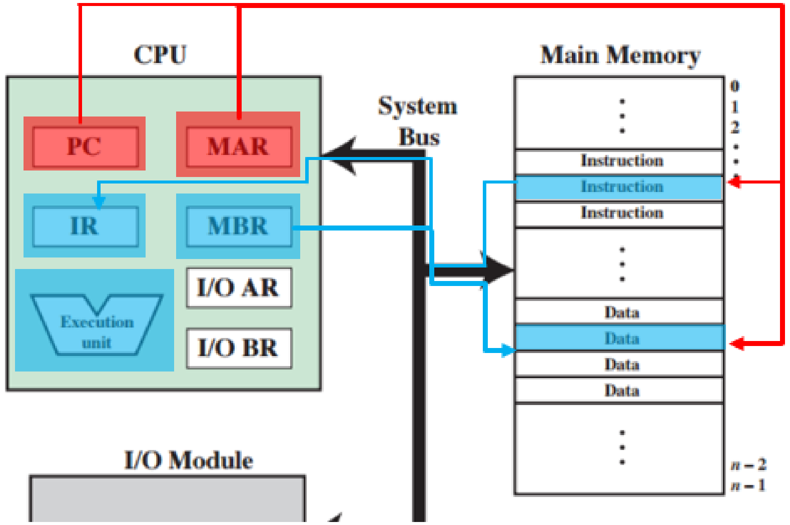
- One of a several current states
- Address of next instruction for process



State
Program Counter

PROCESS CONTROL BLOCK: PROCESS SNAPSHOT

- One of a several current states
- Address of next instruction for process
- Depending on the architecture, the type and count of specific registers varies … regardless, it is their content that is important



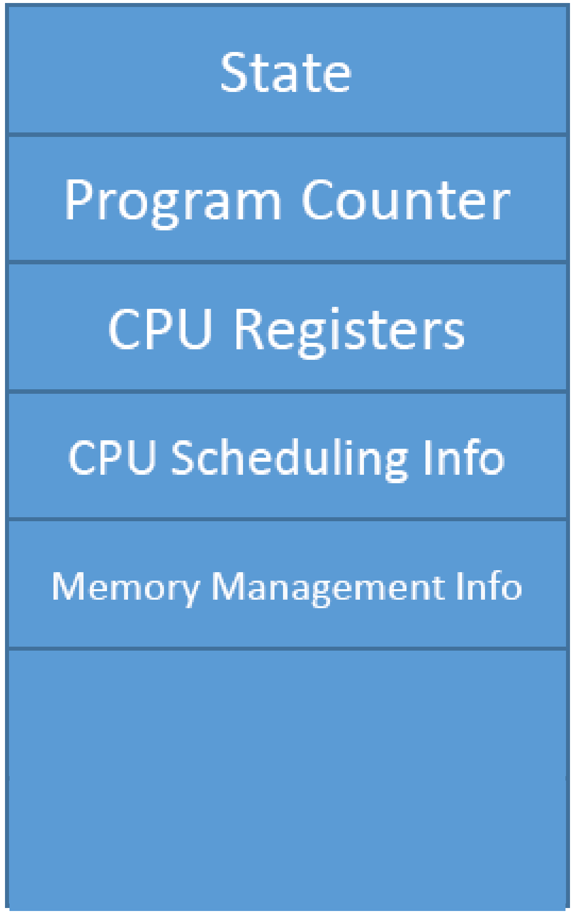
State
Program Counter
CPU Registers

PROCESS CONTROL BLOCK: PROCESS SNAPSHOT

- One of a several current states
- Address of next instruction for process
- Depending on the architecture, the type and count of specific registers varies … regardless, it is their content that is important
- Process priority, pointers to scheduling queues, etc.

PROCESS CONTROL BLOCK: PROCESS SNAPSHOT

- One of a several current states
- Address of next instruction for process
- Depending on the architecture, the type and count of specific registers varies … regardless, it is their content that is important
- Process priority, pointers to scheduling queues, etc.
- Page tables, segment tables, etc.



PROCESS CONTROL BLOCK: PROCESS SNAPSHOT

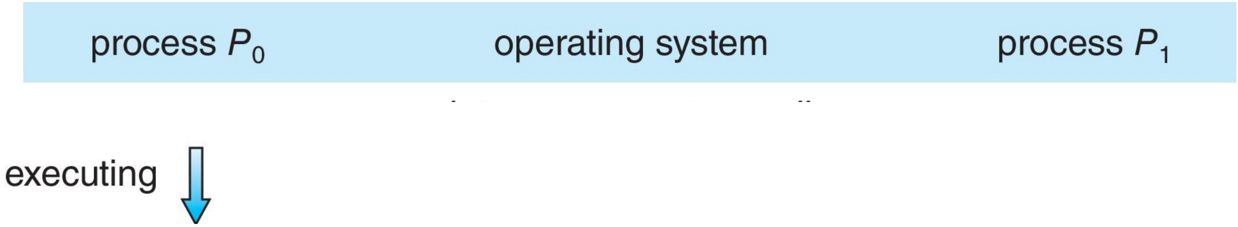
- One of a several current states
- Address of next instruction for process
- Depending on the architecture, the type and count of specific registers varies … regardless, it is their content that is important
- Process priority, pointers to scheduling queues, etc.
- Page tables, segment tables, etc.
- Amount of CPU already used, time limits, job or process number
- List of I/O devices allocated to/in use by process, list of open files, etc.

WESTERN WASHINGTON UNIVERSITY

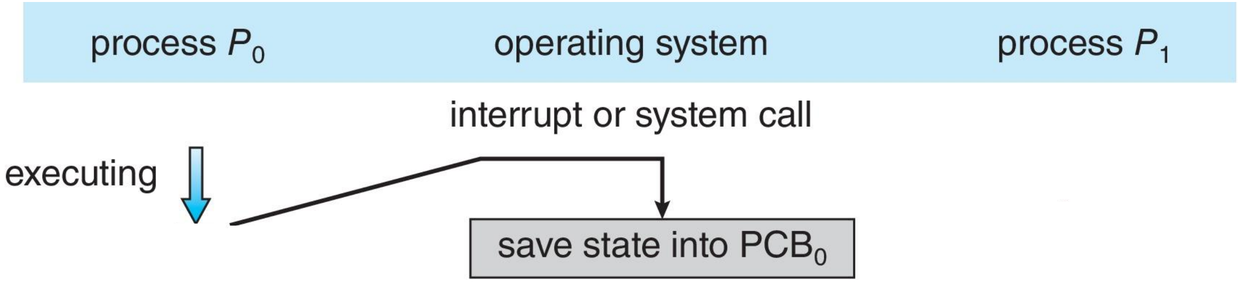
CONTEXT SWITCHING

process P_0	operating system	process P_1

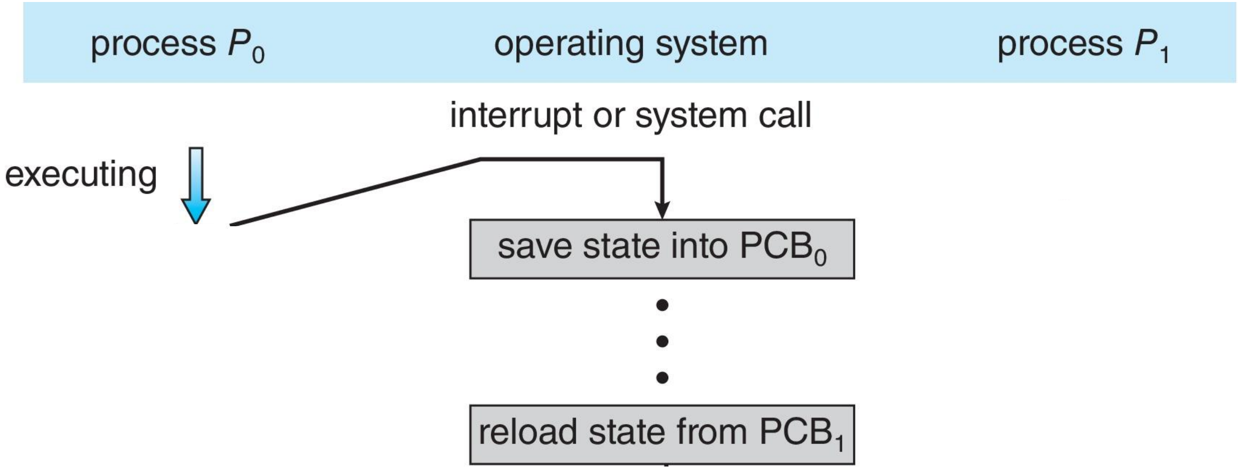
CONTEXT SWITCHING



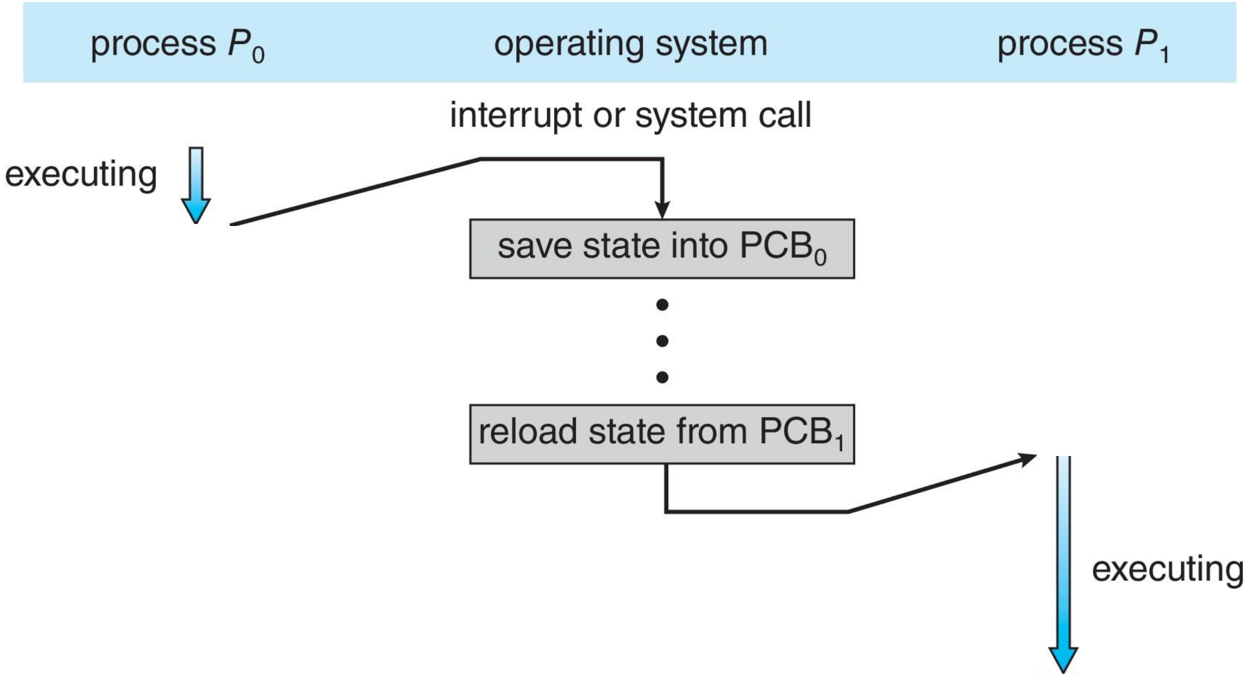
CONTEXT SWITCHING



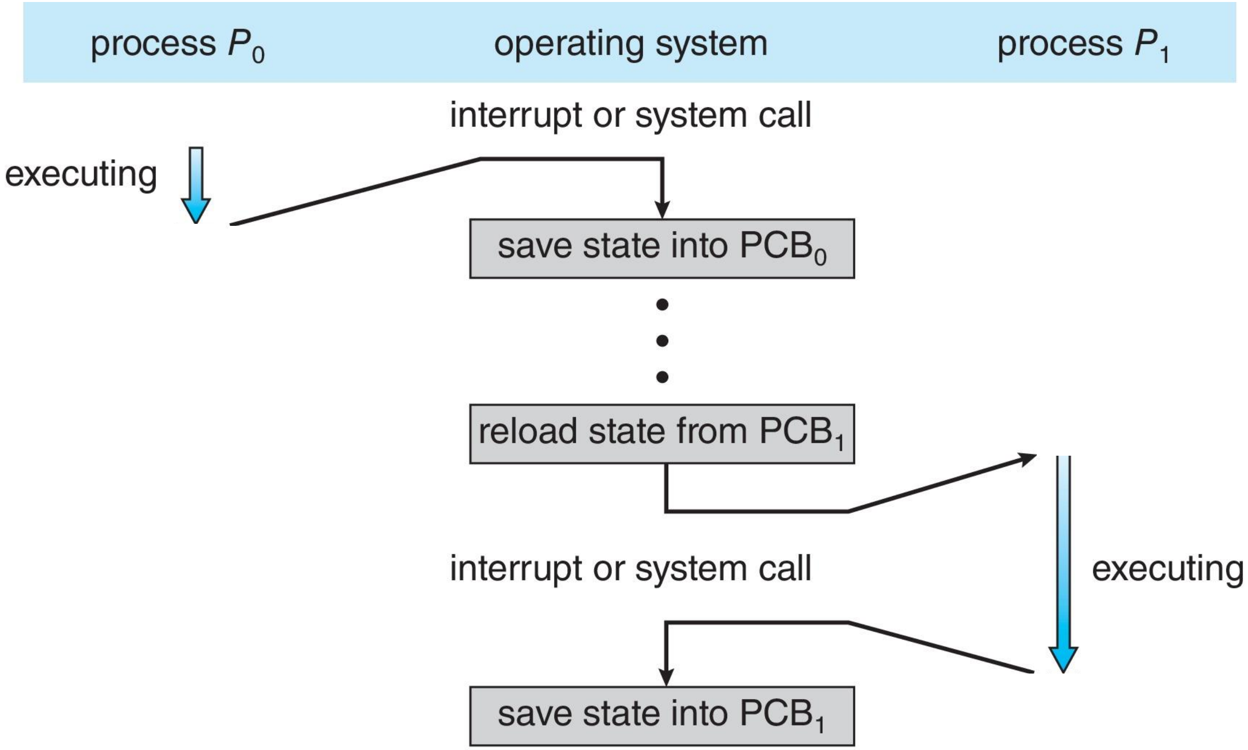
CONTEXT SWITCHING



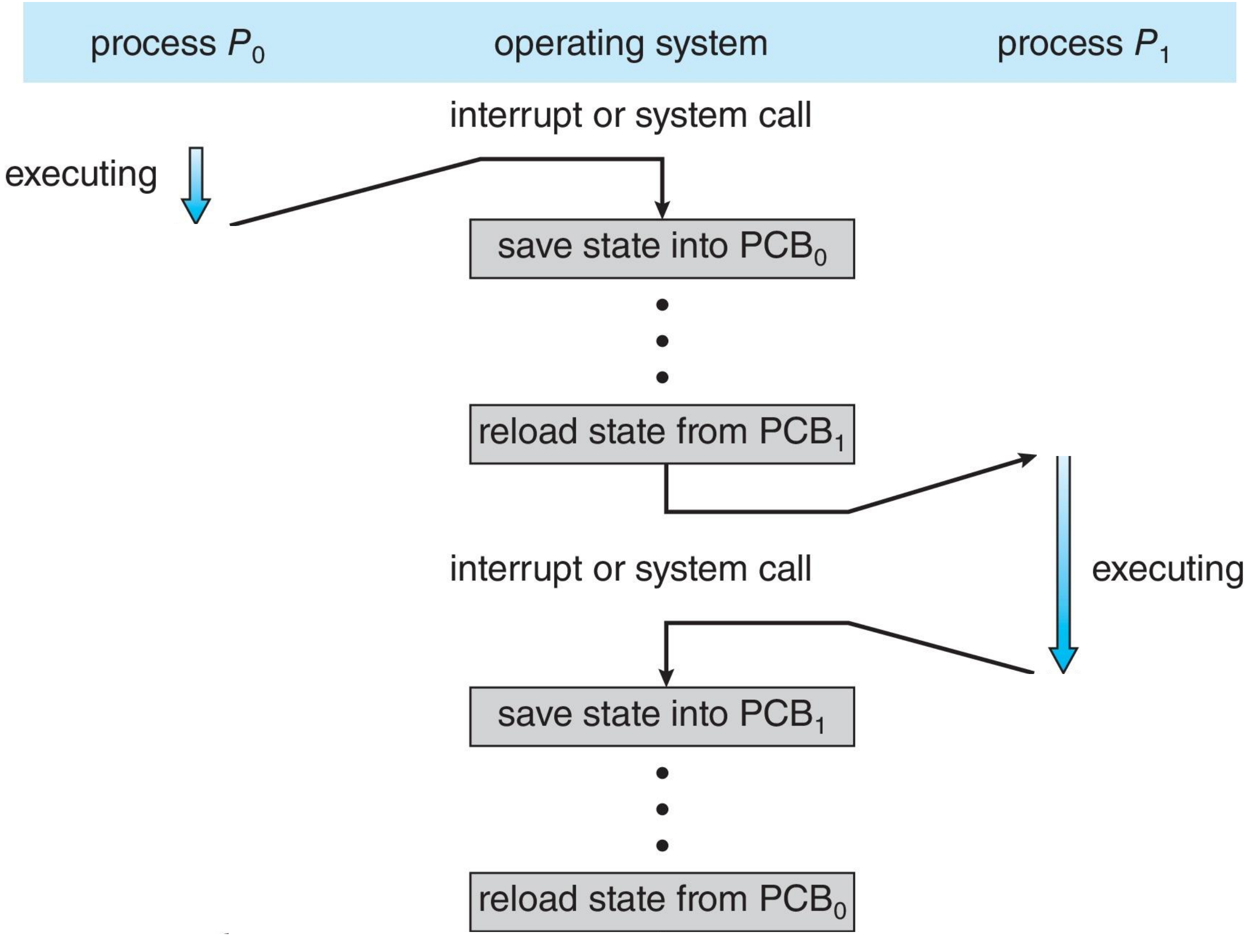
CONTEXT SWITCHING



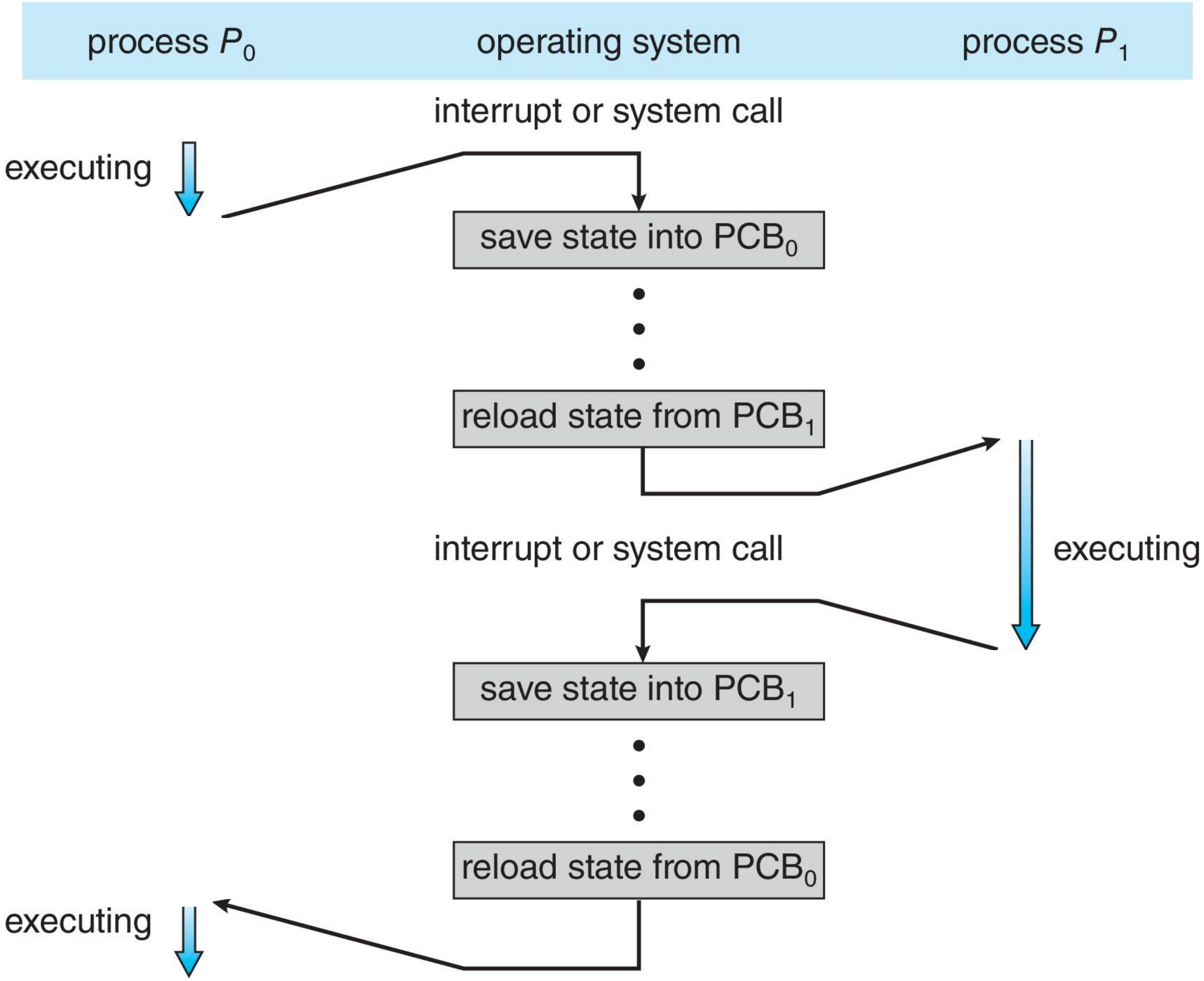
CONTEXT SWITCHING



CONTEXT SWITCHING

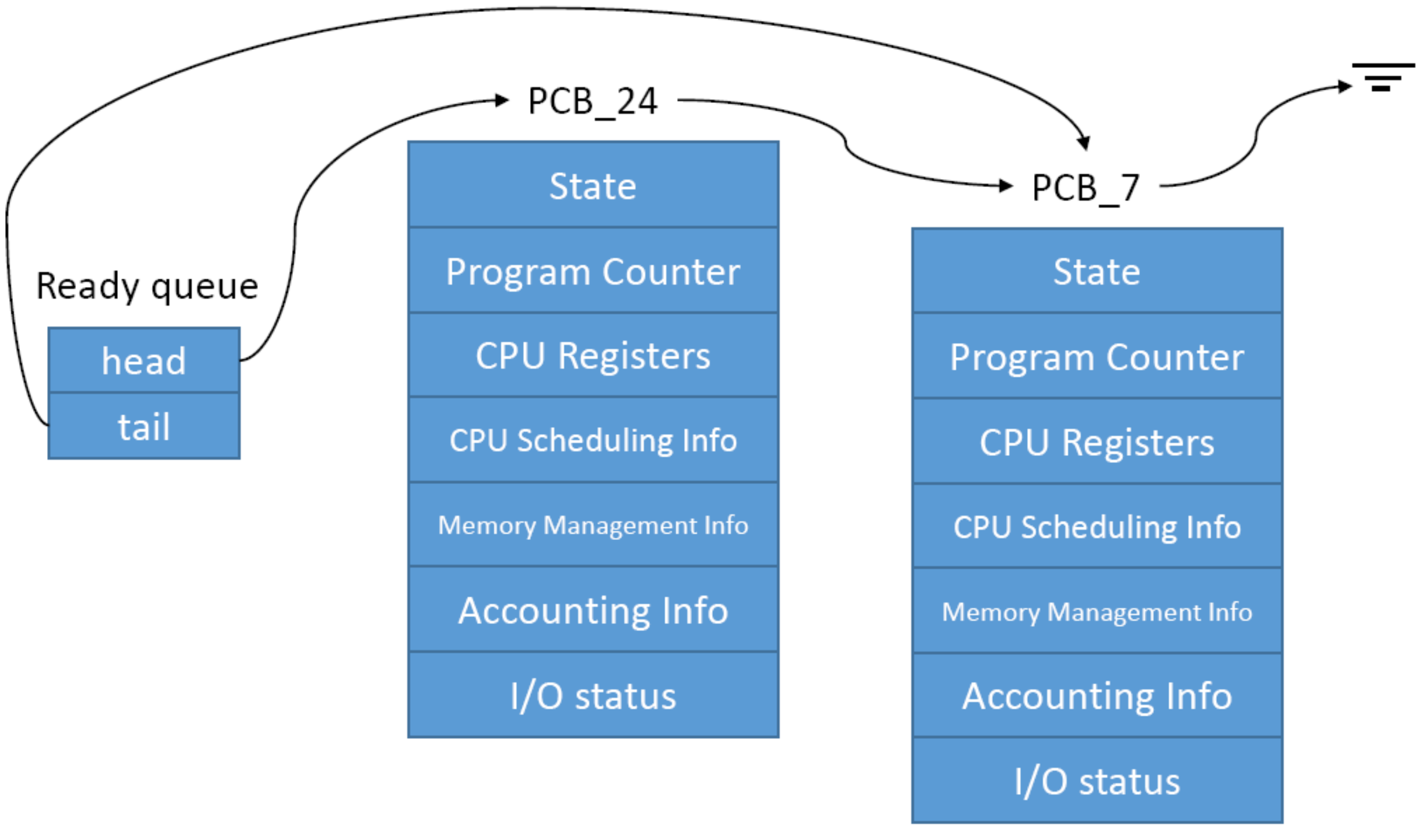


CONTEXT SWITCHING



PROCESS QUEUES

- **Process scheduler** selects among available processes for next execution on CPU core
- Maintains **scheduling queues** of processes
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Wait queues** – set of processes waiting for an event (i.e. I/O)
  - Processes migrate among the various queues



DATA STRUCTURES

- Process Control Block need to be stored in a queue …
- What type of data structure would be most suitable?

DATA STRUCTURES

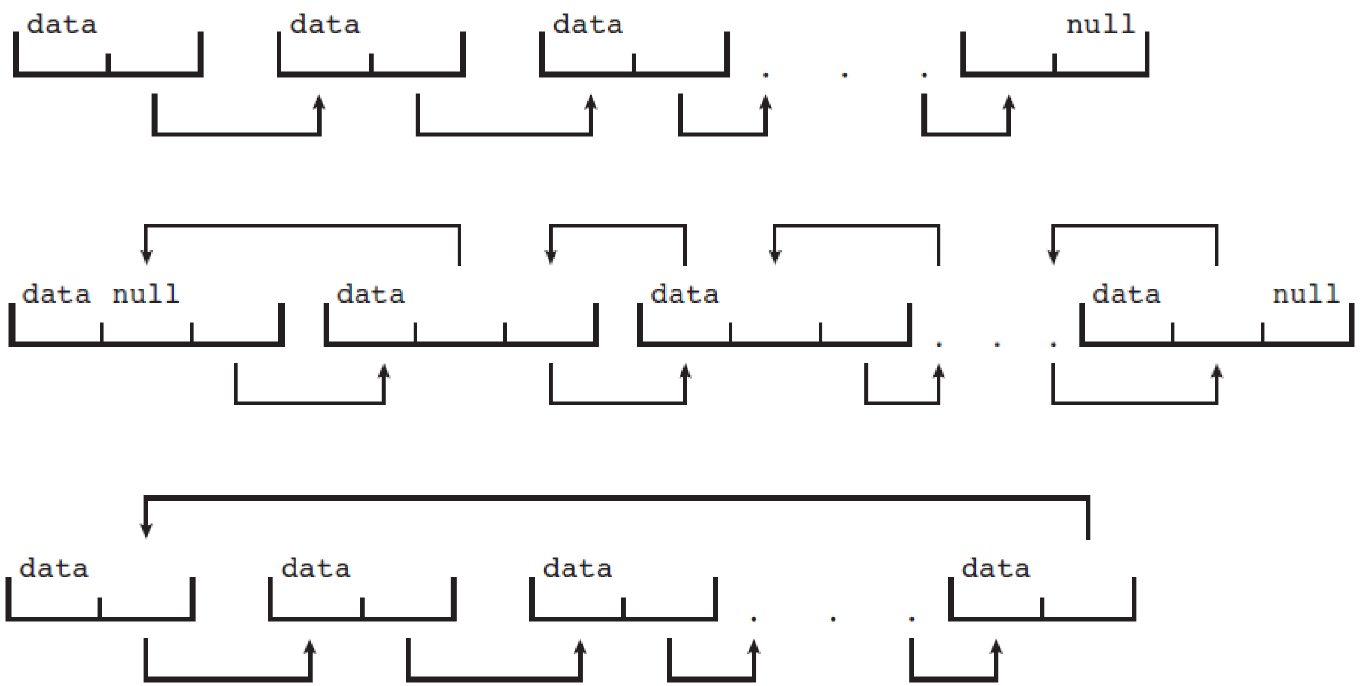
- Process Control Block need to be stored in a queue …
- What type of data structure would be most suitable?
  - Lists
  - Heap
  - Hash Tables

LISTS

Lists

Doubly linked lists

Circularly linked lists

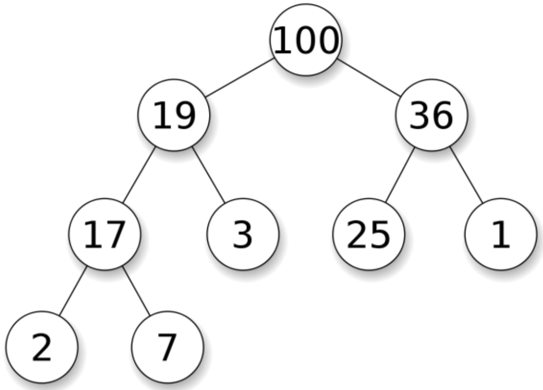


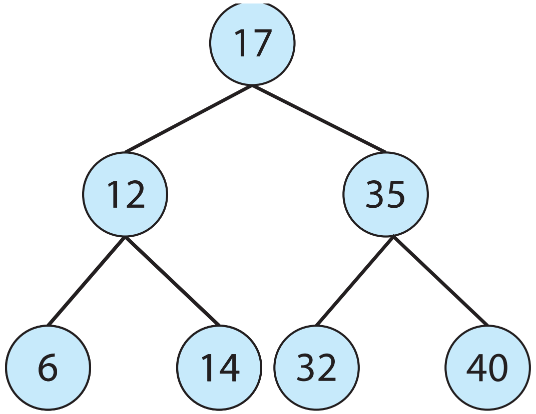
- Generally used when to access data in some set order.
- Great performance for FIFO and LIFO.
- Doubly linked lists reduces penalty for searching.

HEAP/TREES

- **Heap**
  child <= parent

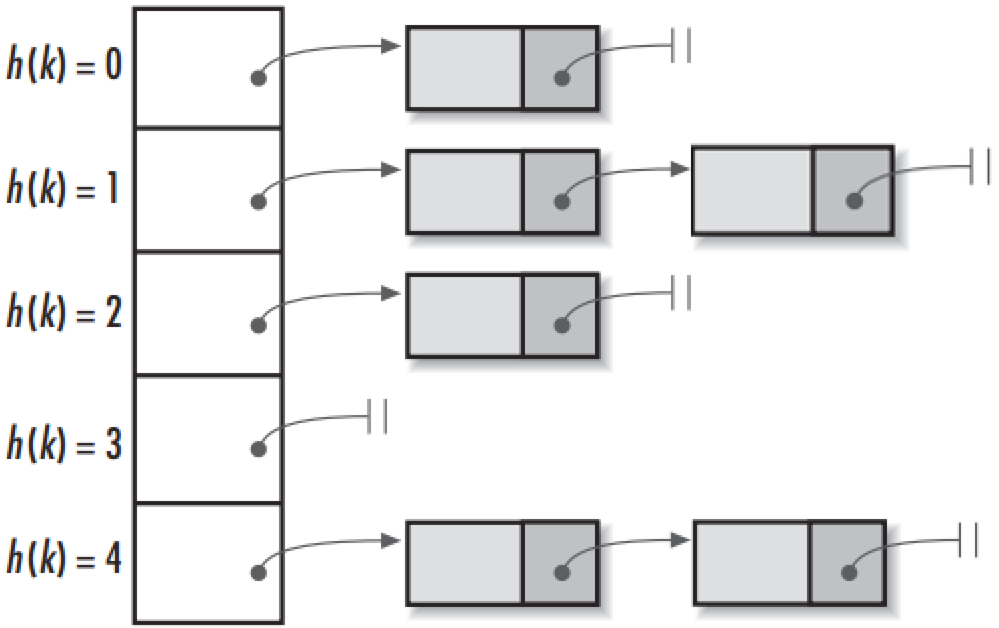
- **Binary search tree**
  left <= right
  - Search performance is *O(n)*
  - **Balanced binary search tree** is *O(log n)*
- Constant time for finding minimum O(1)
- Removing min: O(log(n))





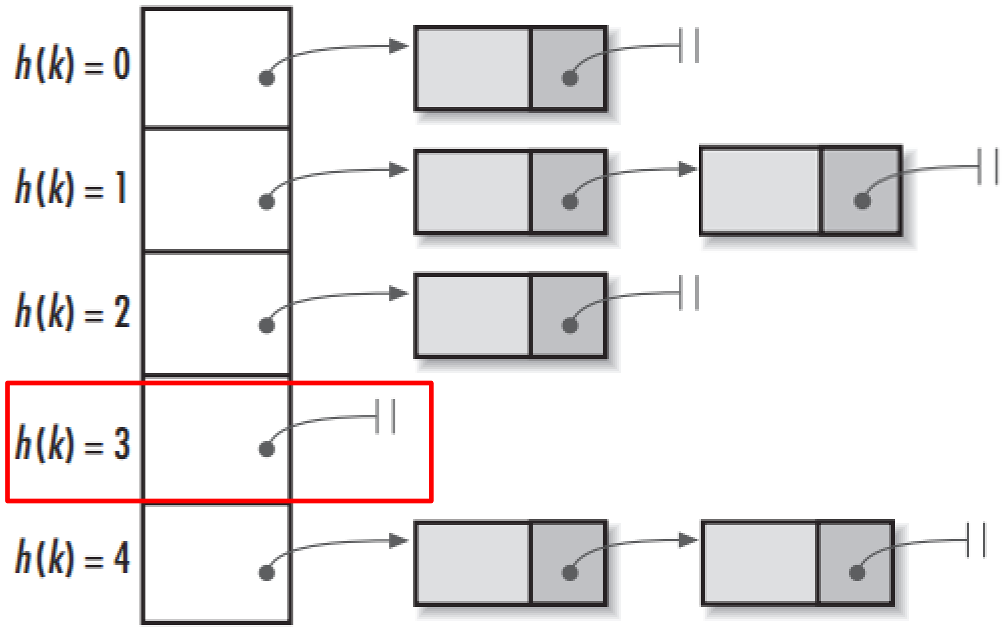
HASH TABLES

- Created by hashing the key of the value being stored.
- Reduces access time at the cost of memory: some keys are never used.



HASH TABLES

- Created by hashing the key of the value being stored.
- Reduces access time at the cost of memory: some keys are never used and are.
- Disadvantage: expensive memory.



WHICH DATA STRUCTURE TO CHOOSE?

Worksheet question:

- What would be the most suitable data structure when we're using:
  - A ready queue with different priority for each process.
  - Process Stack
  - Accessing Random Page in Memory

WHICH DATA STRUCTURE TO CHOOSE?

Worksheet question:

- What would be the most suitable data structure when we're using:
  - A ready queue with different priority for each process. → Heap
  - Process Stack
  - Accessing Random Page in Memory

WHICH DATA STRUCTURE TO CHOOSE?

Worksheet question:

- What would be the most suitable data structure when we're using:
  - A ready queue with different priority for each process. → Heap
  - Process Stack → List
  - Accessing Random Page in Memory

WHICH DATA STRUCTURE TO CHOOSE?

Worksheet question:

- What would be the most suitable data structure when we're using:
  - A ready queue with different priority for each process. → Heap
  - Process Stack → List
  - Accessing Random Page in Memory → Hash Table

PROCESS CREATION

Q: How is a process created?

PROCESS CREATION

Q: How is a process created?

Depending on OS (UNIX-ish versus Windows), there are two primary methods of process creation

PROCESS CREATION

Q: How is a process created?

Depending on OS (UNIX-ish versus Windows), there are two primary methods of process creation

Clone (UNIX, using `fork()`)

- Copy a current process
- New process is the “same” program that was copied
- Running a “new” program is via a different call (`exec()`)

PROCESS CREATION

Q: How is a process created?

Depending on OS (UNIX-ish versus Windows), there are two primary methods of process creation

Clone (UNIX, using `fork()`)

- Copy a current process
- New process is the "same" program that was copied
- Running a "new" program is via a different call (`exec()`)

Create New (Windows, using `CreateProcess()`)

- Create process and specifies program at same system call
- Very little inherited from parent
- Much more overhead than using `fork()`

PROCESS CREATION

Q: How is a process created?

Depending on OS (UNIX-ish versus Windows), there are two primary methods of process creation

system calls implemented by the operating system.

Clone (UNIX, using `fork()`)

- Copy a current process
- New process is the “same” program that was copied
- Running a “new” program is via a different call (`exec()`)

Create New (Windows, using `CreateProcess()`)

- Create process and specifies program at same system call
- Very little inherited from parent
- Much more overhead than using `fork()`

PROCESS CREATION

process creation system call must be called by a running program.

system calls implemented by the operating system.

Clone (UNIX, using `fork()`)

- Copy a current process
- New process is the "same" program that was copied
- Running a "new" program is via a different call (`exec()`)

Create New (Windows, using `CreateProcess()`)

- Create process and specifies program at same system call
- Very little inherited from parent
- Much more overhead than using `fork()`

PROCESS CREATION

process creation system call must be called by a running program.

We need a process to create a process!

system calls implemented by the operating system.

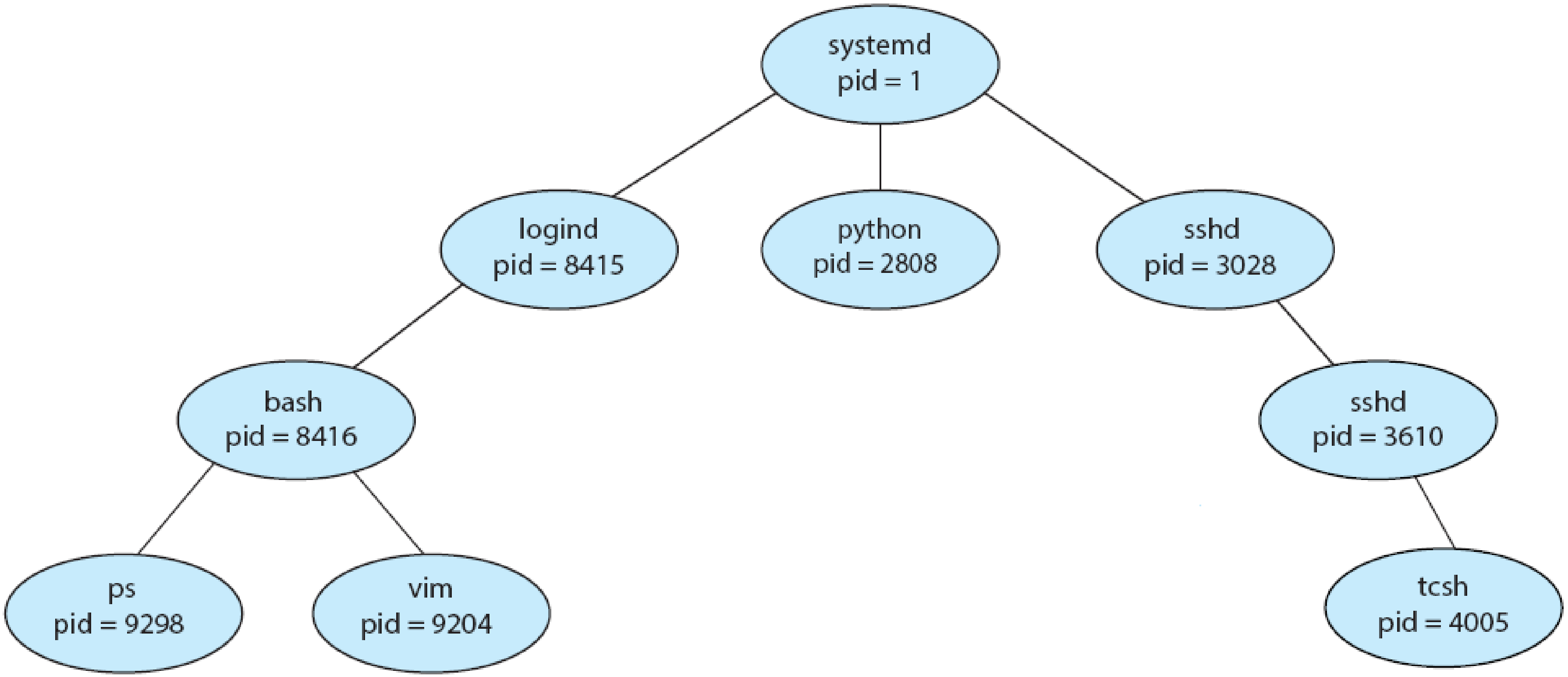
Clone (UNIX, using `fork()`)

- Copy a current process
- New process is the "same" program that was copied
- Running a "new" program is via a different call (`exec()`)

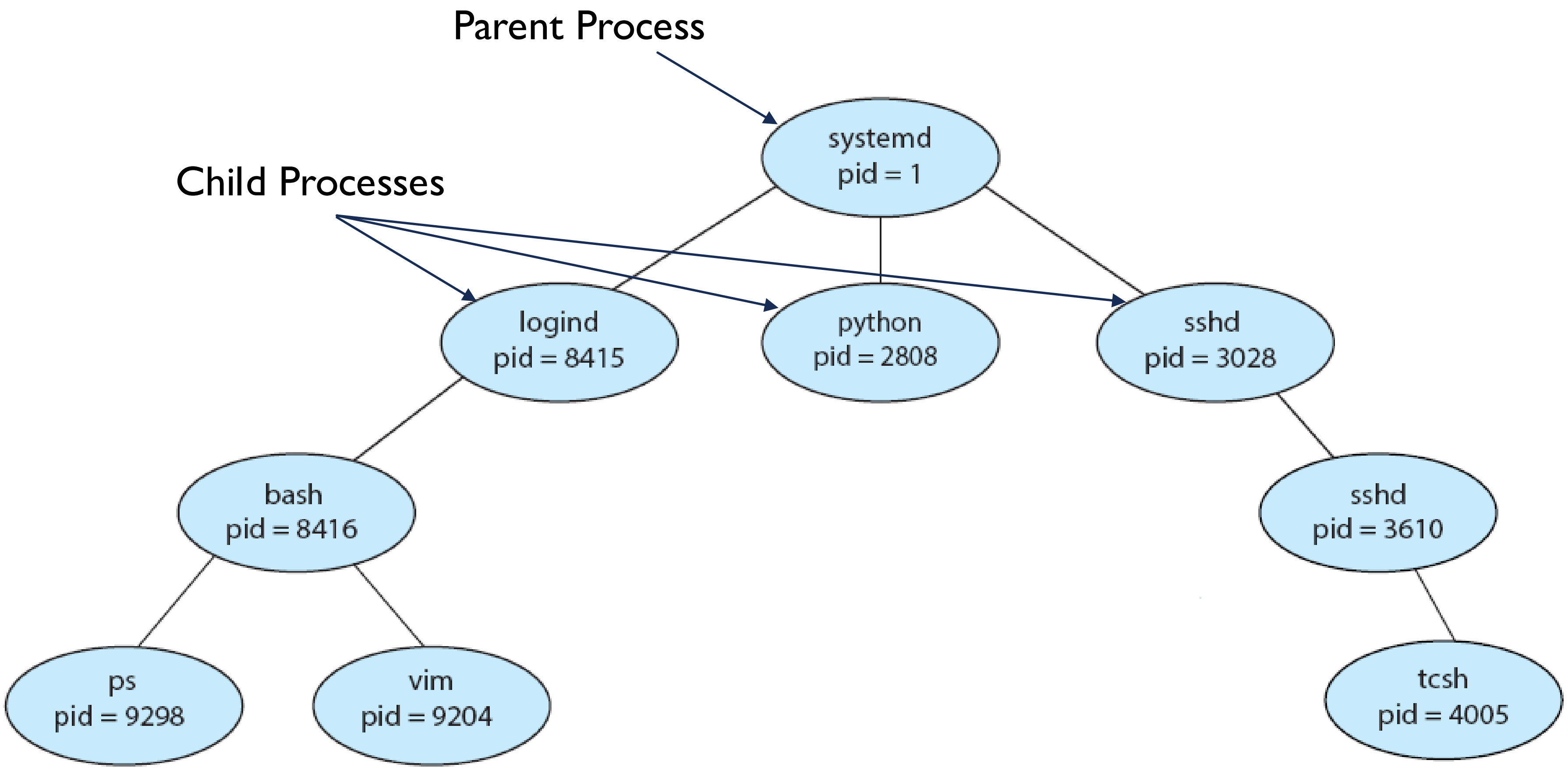
Create New (Windows, using `CreateProcess()`)

- Create process and specifies program at same system call
- Very little inherited from parent
- Much more overhead than using `fork()`

PROCESS TREE



PROCESS TREE



PARENT-CHILD RESOURCE SHARING

- Process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources

PARENT-CHILD RESOURCE SHARING

- Process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources → This is the default behavior

PARENT-CHILD RESOURCE SHARING

- Process identified and managed via a **process identifier** (**pid**)
- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources → This is the default behavior
    - Cloning is different than sharing!
    - Cloning would copy the memory space, sharing would use the same space (might cause write hazards).

PARENT-CHILD RESOURCE SHARING

- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate using the wait() system call.

UNIX: `fork()`

- Worksheet Q1: What would the output be?

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{

    // make two process which run same
    // program after this instruction
    fork();

    printf("Hello world!\n");
    return 0;
}
```

UNIX: `fork()`

- Worksheet Q1: What would the output be?

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{

    // make two process which run same
    // program after this instruction
    fork();

    printf("Hello world!\n");
    return 0;
}
```

Output:

```
Hello world!
Hello world!
```

UNIX: `fork()`

Worksheet Q2: What would be the output?

```c
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

UNIX: `fork()`

```c
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

Worksheet Q2: What would be the output?

Output:

```
hello
hello
hello
hello
hello
hello
hello
hello
```

$n = 3, 2^3 = 8$

UNIX: `fork()`

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

Worksheet Q2: What would be the output?

Output:

```
hello
hello
hello
hello
hello
hello
hello
hello
```

$n = 3,\ 2^3 = 8$

Q: How to run a new process with a different program?

UNIX: `exec()`

Q: How to run a new process with a different program?

UNIX: `exec()`

Q: How to run a new process with a different program?

- **`fork()`** system call creates new process
- **`exec()`** system replaces the process memory space with a new program, practically "overwriting" the process.

UNIX: `exec()`

Q: How to run a new process with a different program?

- **`fork()`** system call creates new process
- **`exec()`** system replaces the process memory space with a new program, practically "overwriting" the process.
- So how do you create a *new process* with a *new program*?

UNIX: `exec()`

Q: How to run a new process with a different program?

- **`fork()`** system call creates new process
- **`exec()`** system replaces the process memory space with a new program, practically “overwriting” the process.
- So how do you create a *new process* with a *new program*?
- **`fork()`** → **`exec()`**

UNIX: `exec()`

Q: How to run a new process with a different program?

- **`fork()`** system call creates new process
- **`exec()`** system replaces the process memory space with a new program, practically "overwriting" the process.
- So how do you create a *new process* with a *new program*?
- **`fork()`** → **`exec()`**
- How to distinguish between parent and child process?

UNIX: exec()

Q: How to run a new process with a different program?

- **fork()** system call creates new process
- **exec()** system replaces the process memory space with a new program, practically "overwriting" the process.
- So how do you create a *new process* with a *new program*?
- **fork() → exec()**
- How to distinguish between parent and child process?

```c
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

UNIX: `fork()`

- **`fork()`**
- returns a negative value if it fails.

UNIX: `fork()`

- **`fork()`**
- returns a negative value if it fails
- returns the child's 'pid' (process ID) for in the parent process
- returns '0' in the child process.

UNIX: `fork()`

- **`fork()`**
- returns a negative value if it fails
- returns the child's 'pid' (process ID) for in the parent process
- returns '0' in the child process.

```c
/* fork a child process */
pid = fork();
if (pid < 0) { /* error occured */
    fprintf(stderr, "Fork Failed");
}
else if (pid == 0) {
    /* code 1 */
}
else {
    /* code 2 */
}
```

UNIX: `fork()`

- **`fork()`**
- returns a negative value if it fails
- returns the child's 'pid' (process ID) for in the parent process
- returns '0' in the child process.

Worksheet Q3:

Where would you call the exec() in this code to create a new process with a new code?

```
/* fork a child process */
pid = fork();
if (pid < 0) { /* error occured */
    fprintf(stderr, "Fork Failed");
}
else if (pid == 0) {
    /* code 1 */
}
else {
    /* code 2 */
}
```

UNIX: `fork()`

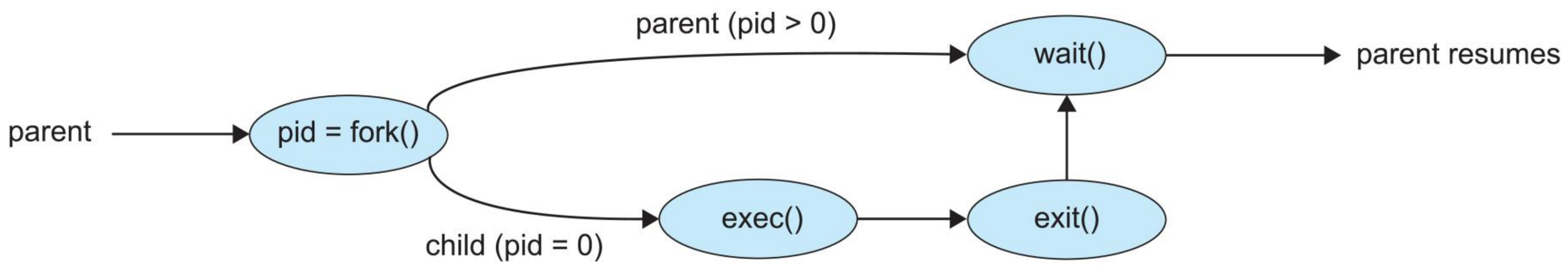
- **`fork()`**
- returns a negative value if it fails
- returns the child's 'pid' (process ID) for in the parent process
- returns '0' in the child process.

Worksheet Q3:

Where would you call the exec() in this code to create a new process with a new code?

```c
/* fork a child process */
pid = fork();
if (pid < 0) { /* error occured */
    fprintf(stderr, "Fork Failed");
}
else if (pid == 0) {
    exec()
}
else {
    /* code 2 */
}
```

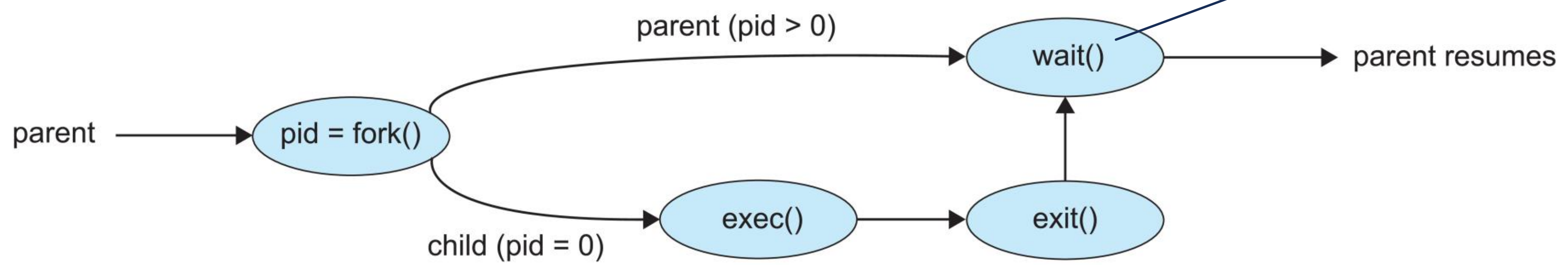
fork() → exec()



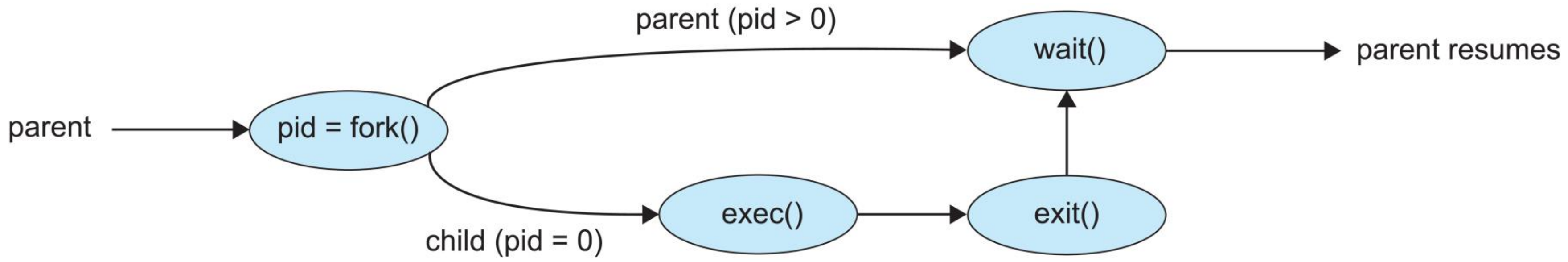
parent (pid > 0)
wait()
parent resumes
parent
pid = fork()
exec()
exit()
child (pid = 0)

fork() → exec()

Suspends current process until one of its children are terminated.

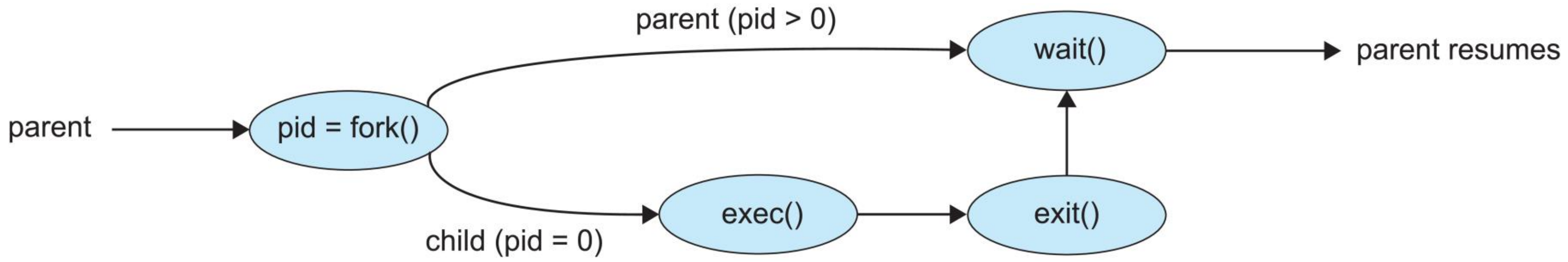


fork() → exec()



Question: There is an inefficiency for using the fork() then exec() system calls to run a new program. What is it?

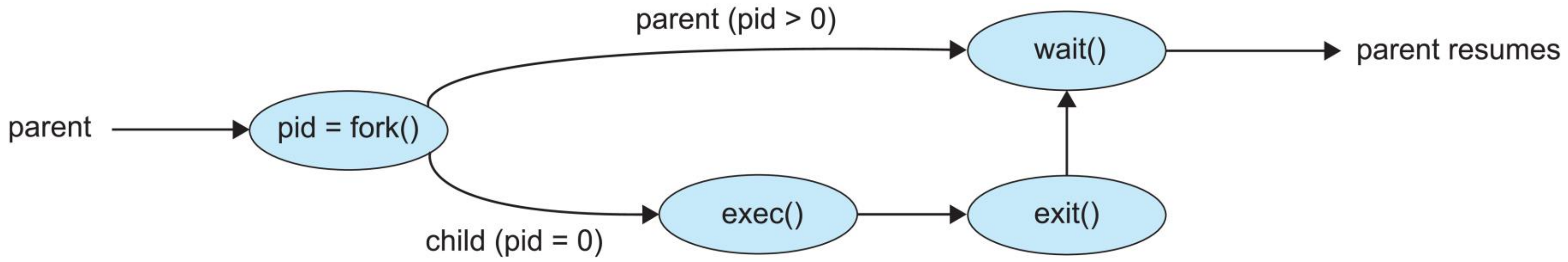
fork() → exec()



Question: There is an inefficiency for using the fork() then exec() system calls to run a new program. What is it?

- We're wasting resources copying a process only to destroy it and overwrite it …

fork() → exec()



Question: There is an inefficiency for using the fork() then exec() system calls to run a new program. What is it?

- We're wasting resources copying a process only to destroy it and overwrite it …
- This has been fixed in later version of linux kernels where the "copy" does not really start until the kernel notices that the new process is being used as is.

PROCESS TERMINATION

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process resources are deallocated by operating system

PROCESS TERMINATION

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process resources are deallocated by operating system
- Parent process may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:

PROCESS TERMINATION

- Process executes last statement and then asks the operating system to delete it using the `exit()` system call.
  - Returns status data from child to parent (via `wait()`)
  - Process resources are deallocated by operating system
- Parent process may terminate the execution of children processes using the `abort()` system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Parent suspects child process is not running correctly
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

PROCESS TERMINATION

- Process executes last statement and then asks the operating system to delete it using the **`exit()`** system call.
  - Returns status data from child to parent (via **`wait()`**)
  - Process resources are deallocated by operating system
- Parent process may terminate the execution of children processes using the **`abort()`** system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Parent suspects child process is not running correctly
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates
  - *Some* operating systems do not allow child to exist if its parent has terminated. If a process terminates, then all its children must also be terminated.

MULTI-PROCESS ARCHITECTURE

- Single programs can utilize multiple processes.
- Advantages:
  - Can operate multiple tasks at the same time.
  - Easier to program/manage: System will handle multitasking rather than programmer.
  - Reliability: if one process fails, only the tasks related to it stop.

MULTI-PROCESS ARCHITECTURE

- Many web browsers ran as single process (some still do)
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is a multiprocess program with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
  - **Plug-in** process for each type of plug-in

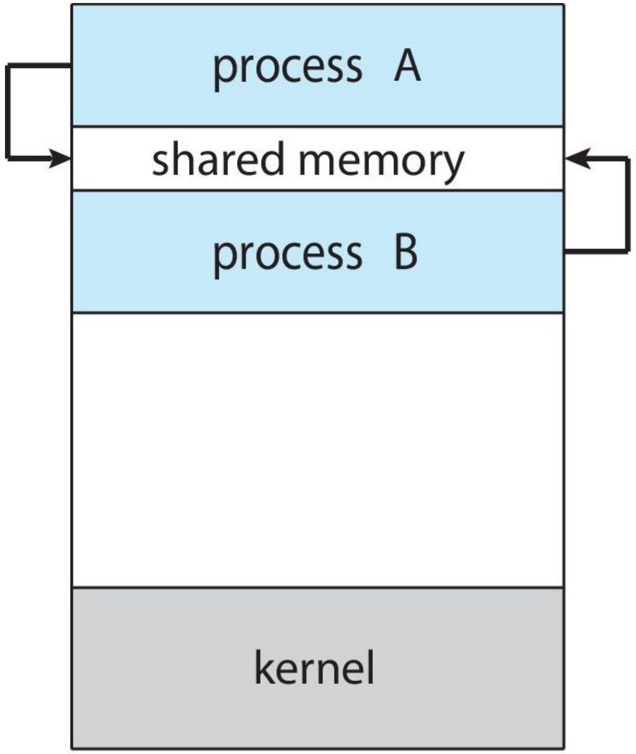
IPC: INTER-PROCESS COMMUNICATION

- Two primary methods:
  - Shared Memory
  - Messaging

SHARED MEMORY VS MESSAGE PASSING

(a) Shared memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Advantage: very fast and efficient

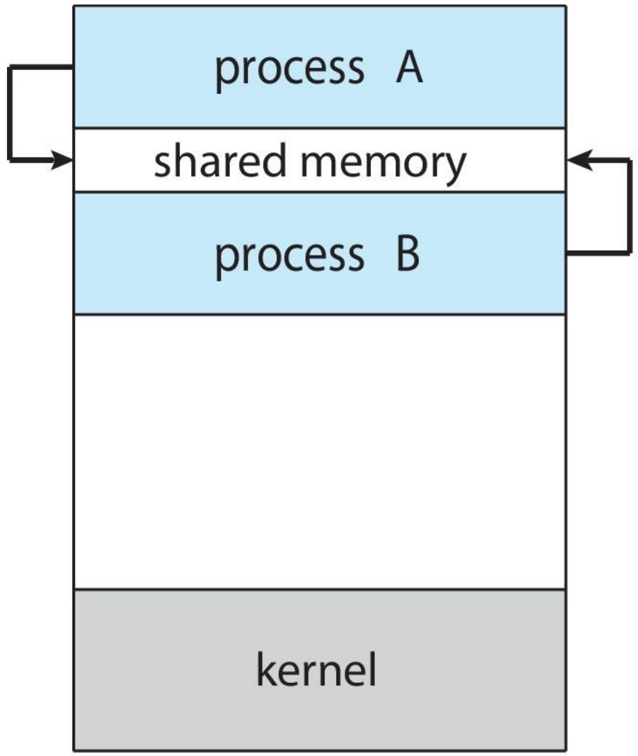


(a)

SHARED MEMORY VS MESSAGE PASSING

(a) Shared memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Advantage: very fast and efficient
- Disadvantage?

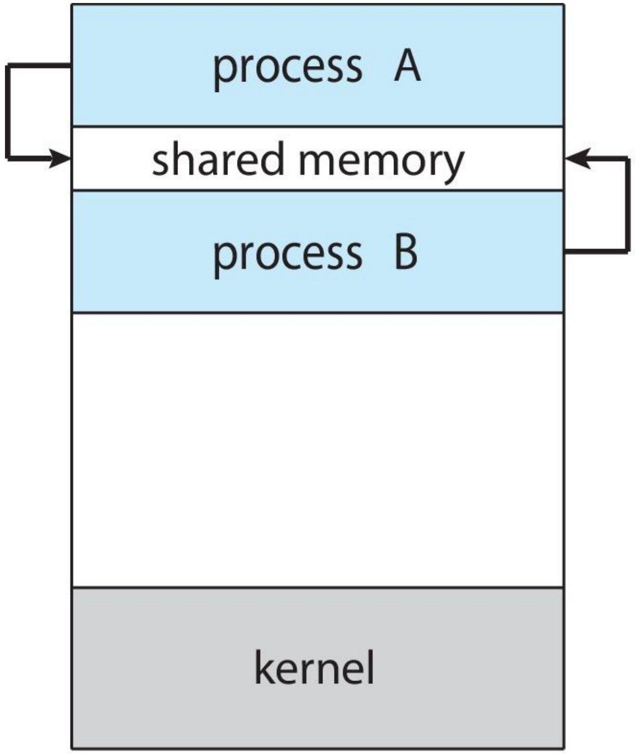


(a)

SHARED MEMORY VS MESSAGE PASSING

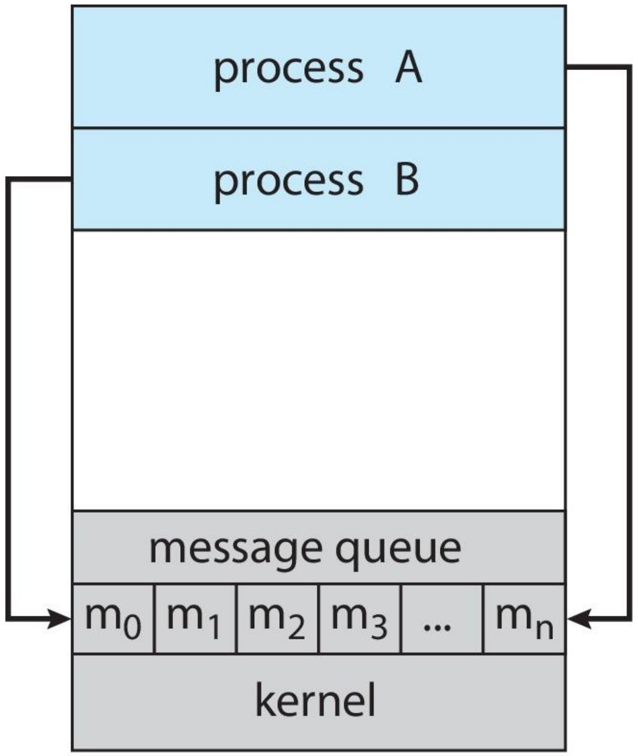
(a) Shared memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Advantage: very fast and efficient
- Disadvantage?
- OS needs to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is discussed in great details in later chapters.



(a)

SHARED MEMORY VS MESSAGE PASSING



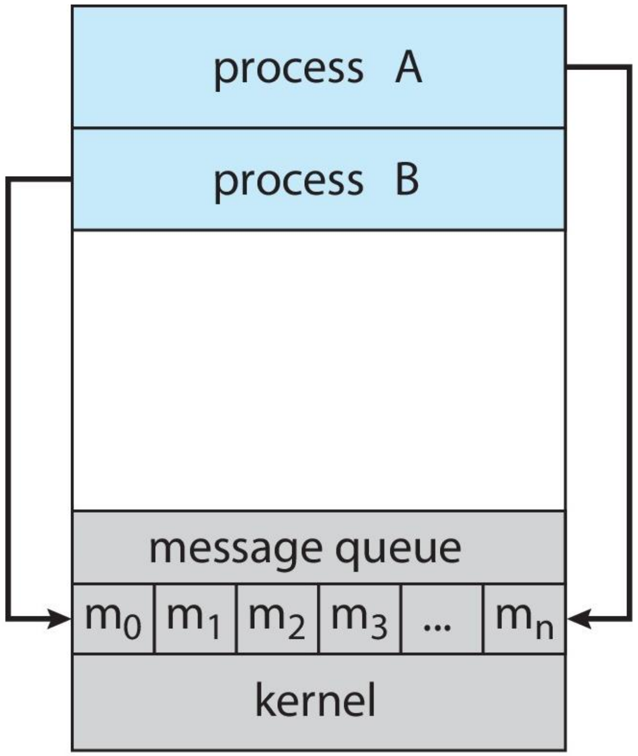
(b)

(b) Message passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)

SHARED MEMORY VS MESSAGE PASSING

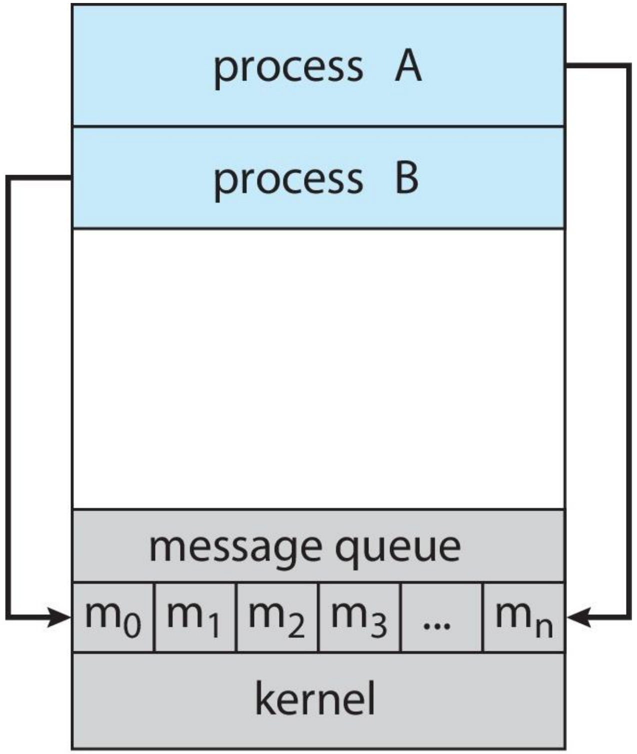
(b) Message passing



(b)

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable

SHARED MEMORY VS MESSAGE PASSING

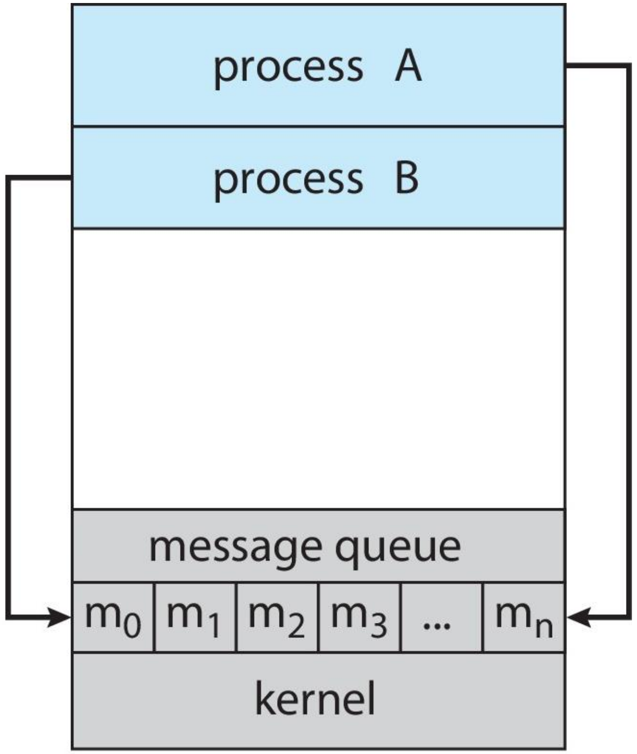


(b)

(b) Message passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable
- Advantage: Can be easily synchronized or even used for synchronization, which will be discussed later.

SHARED MEMORY VS MESSAGE PASSING

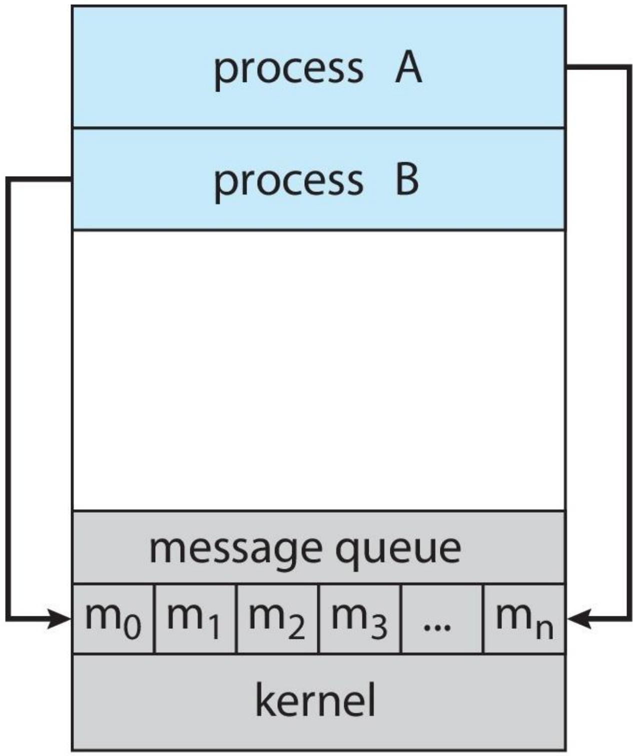


(b)

(b) Message passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable
- Advantage: Can be easily synchronized or even used for synchronization, which will be discussed later.
- Disadvantage?

SHARED MEMORY VS MESSAGE PASSING



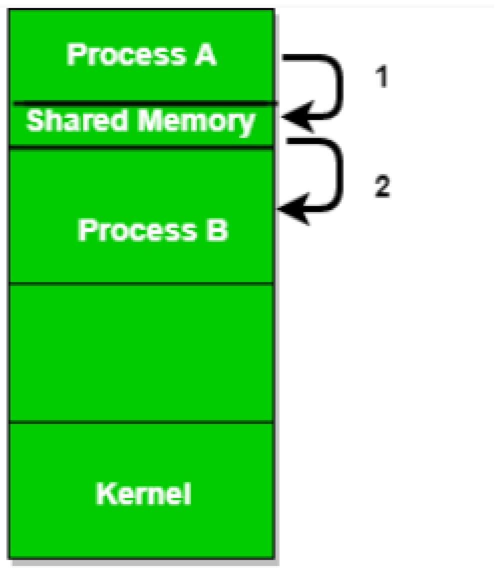
(b)

(b) Message passing

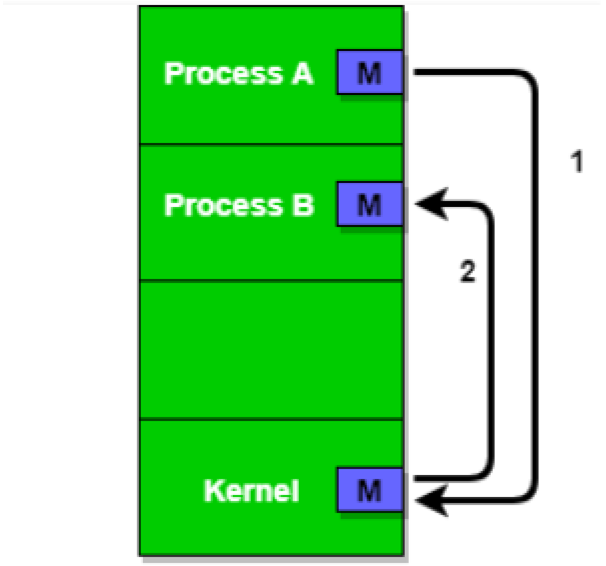
- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable
- Advantage: Can be easily synchronized or even used for synchronization, which will be discussed later.
- Disadvantage: Requires more operations and more read/writes than shared memory.

SHARED MEMORY VS MESSAGE PASSING

(a) Shared memory



(b) Message passing



POSIX SHARED MEMORY

SHARED MEMORY EXAMPLES: POSIX SHARED MEMORY

- POSIX: Portable Operating System Interface
- Standards specified by the IEEE Computer Society
- Objective: OS have unified API allowing portability of software.

POSIX SHARED MEMORY

- POSIX: Portable Operating System Interface
- Standards specified by the IEEE Computer Society
- Objective: OS have unified API allowing portability of software.

Most Linux systems are partially or fully compliant to the POSIX standards making it fairly easy to port code.

POSIX SHARED MEMORY

- POSIX Shared Memory
  - Process first creates shared memory segment

    ```
    shm_fd = shm_open(name, O_CREAT);
    ```

  - Also used to open an existing segment.
  - Set the size of the object

    ```
    ftruncate(shm_fd, 4096);
    ```

  - **Map shared memory object to the process's address space**

    ```
    void* addr = mmap(NULL, 1024, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd , 0);
    ```

IPC - MESSAGE IMPLEMENTATIONS

- Pipes
- Sockets
- Local/Remote Procedure Calls

PIPES

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)

PIPES

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes
- Windows calls these **anonymous pipes**

BLOCKING VS UNBLOCKING

- **Blocking** is considered **synchronous**
  - **Blocking send --** the sender is blocked until the message is received
  - **Blocking receive --** the receiver is blocked until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking send --** the sender sends the message and continue
  - **Non-blocking receive --** the receiver receives:
    - A valid message, or
    - Null message

BLOCKING VS UNBLOCKING

- **Blocking** is considered **synchronous**
  - **Blocking send --** the sender is blocked until the message is received
  - **Blocking receive --** the receiver is blocked until a message is available

Blocked send/receive can result in indefinite wait or even infinite wait …
This can freeze the whole program …

Worksheet Q1:What would be possible solution to this problem?

BLOCKING VS UNBLOCKING

- **Blocking** is considered **synchronous**
  - **Blocking send --** the sender is blocked until the message is received
  - **Blocking receive --** the receiver is blocked until a message is available

Blocked send/receive can result in indefinite wait or even infinite wait …
This can freeze the whole program …

Worksheet Q1:What would be possible solution to this problem?

- Timeout: set max wait time and return with "null" or error.

BLOCKING VS UNBLOCKING

- **Blocking** is considered **synchronous**
  - **Blocking send --** the sender is blocked until the message is received
  - **Blocking receive --** the receiver is blocked until a message is available

Blocked send/receive can result in indefinite wait or even infinite wait …
This can freeze the whole program …

Worksheet Q1:What would be possible solution to this problem?

- Timeout: set max wait time and return with "null" or error.
- Interrupt blocked read/write thread and wake it up.

BLOCKING VS UNBLOCKING

- **Blocking** is considered **synchronous**
  - **Blocking send --** the sender is blocked until the message is received
  - **Blocking receive --** the receiver is blocked until a message is available

Blocked send/receive can result in indefinite wait or even infinite wait …
This can freeze the whole program …

Worksheet Q1:What would be possible solution to this problem?

- Timeout: set max wait time and return with "null" or error.
- Interrupt blocked read/write thread and wake it up.
- Create a worker thread just for the read/write. The program can continue running in main thread.

PIPES IN UNIX

- The vertical bar | is the pipe operator in unix shell.
- The transferred data is never saved in a file, it is simply communicated to the other process.
- Unnamed or "ordinary" pipes are destroyed after the process completes execution.

Syntax :

```
command_1 | command_2 | command_3 | .... | command_N
```

```
$ ls -l | more
```

PIPES IN UNIX

- The vertical bar | is the pipe operator in unix shell.
- The transferred data is never saved in a file, it is simply communicated to the other process.
- Unnamed or "ordinary" pipes are destroyed after the process completes execution.
- Named pipes can be created by the mknod() system call with the 'FIFIO' option:
  `mknod("mypipe",SIFIFO,0)`
- You can also use mkfifo("name",0666)
- Named pipes allow communication between any two processes

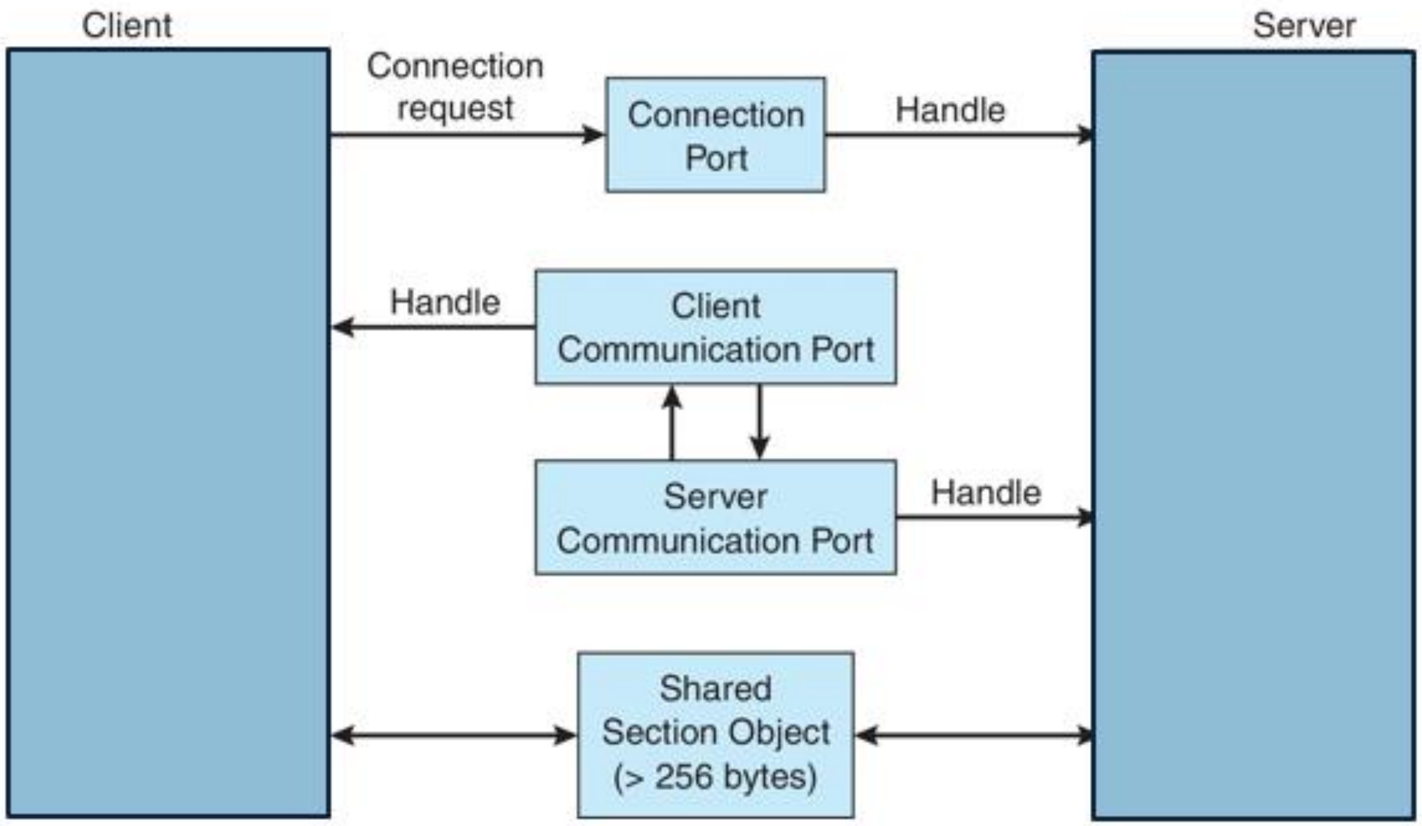
Syntax :

```
command_1 | command_2 | command_3 | .... | command_N
```

```
$ ls -l | more
```

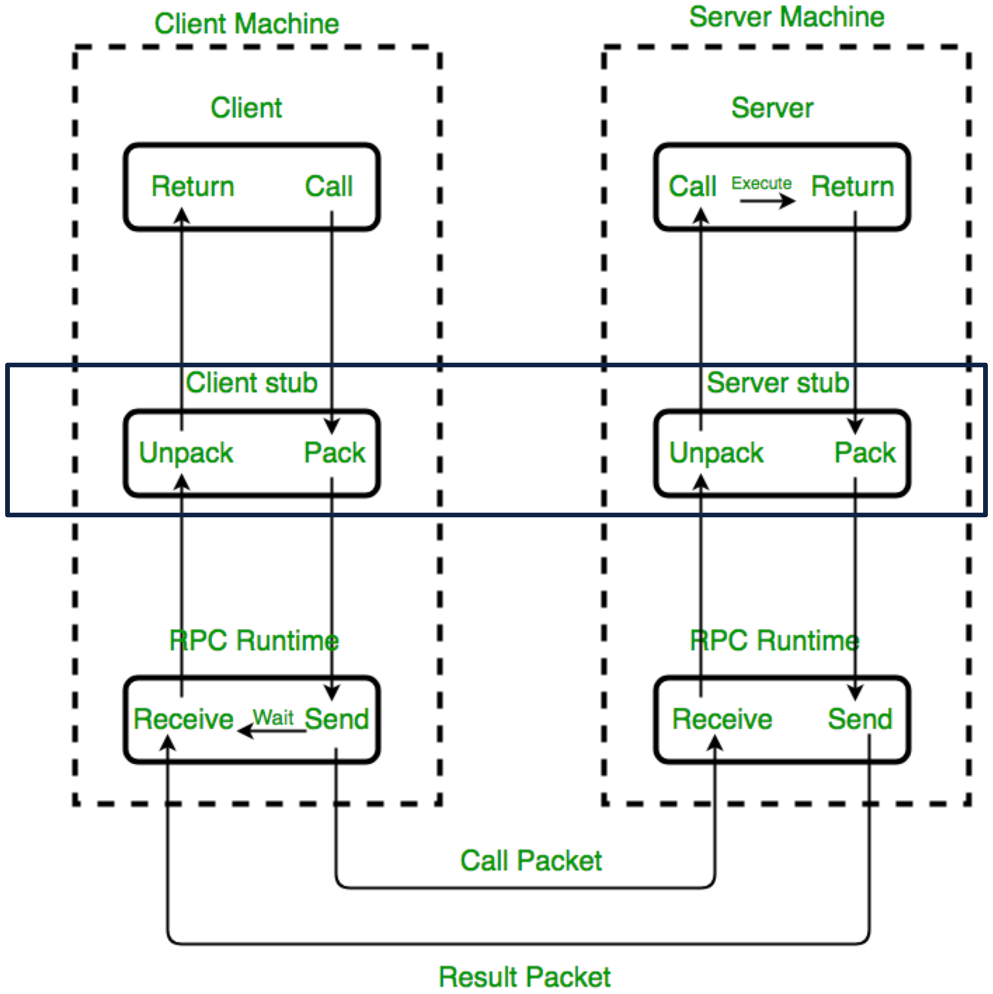
LOCAL PROCEDURE CALLS IN WINDOWS

- Message-passing centric via **advanced local procedure call** (**LPC**) facility
  - Only works between processes on the same system
  - Uses ports to establish and maintain communication channels



REMOTE PROCEDURE CALLS

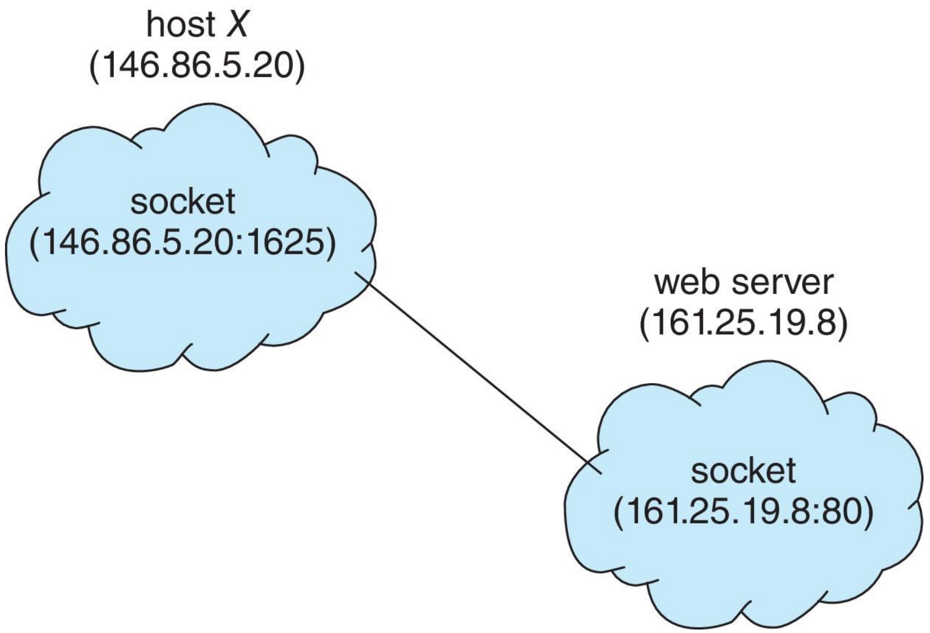
- Remote procedure call (RPC) abstracts procedure calls between processes on networked systems
  - Again uses ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
- The client-side stub locates the server and **marshalls** (packs) the parameters
- The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server



Implementation of RPC mechanism

SOCKETS

- A **socket** is defined as an endpoint for communication
- Concatenation of IP address and **port** – a number included at start of message packet to differentiate network services on a host
- The socket **161.25.19.8:1625** refers to port **1625** on host **161.25.19.8**
- Communication consists between a pair of sockets
- All ports below 1024 are ***well known***, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running



REVIEW QUESTIONS

- Q1: What is the difference between Linux fork() and Windows clone()?

- Fork(): clones current process.
- CreateProcess(): creates a new process.

REVIEW QUESTIONS

- Q2: Fill out the empty lines with parent or child.

```
int main()
{
        if (fork()== 0)
                printf("hello from 1._____\n");
        else
        {
                printf("hello from 2.______\n");
                wait(NULL);
                printf("3.______ has terminated\n");
        }

        printf("Bye\n");
        return 0;
}
```

REVIEW QUESTIONS

- Q2: Fill out the empty lines with parent or child.

```
int main()
{
        if (fork()== 0)
                printf("hello from child\n");
        else
        {
                printf("hello from ______\n");
                wait(NULL);
                printf("______ has terminated\n");
        }

        printf("Bye\n");
        return 0;
}
```

REVIEW QUESTIONS

- Q2: Fill out the empty lines with parent or child.

```
int main()
{
    if (fork()== 0)
        printf("hello from child\n");
    else
    {
        printf("hello from parent\n");
        wait(NULL);
        printf("_______ has terminated\n");
    }

    printf("Bye\n");
    return 0;
}
```

REVIEW QUESTIONS

- Q2: Fill out the empty lines with parent or child.

```
int main()
{
    if (fork()== 0)
        printf("hello from child\n");
    else
    {
        printf("hello from parent\n");
        wait(NULL);
        printf("child has terminated\n");
    }

    printf("Bye\n");
    return 0;
}
```

REVIEW QUESTIONS

- Q3: Advantages of IPC using shared memory?

REVIEW QUESTIONS

- Q3: Advantages of IPC using shared memory?
- Very fast
- Data needs to be written once and read once

REVIEW QUESTIONS

- Q4: Disadvantages of IPC using shared memory?

REVIEW QUESTIONS

- Q4: Disadvantages of IPC using shared memory?
- Requires careful implementation
- Requires synchronization support
- Too complicated when many processes are involved.

CHAPTER 4: THREADS

Thread : basic unit of CPU utilization

CHAPTER 4: THREADS

Thread : basic unit of CPU utilization

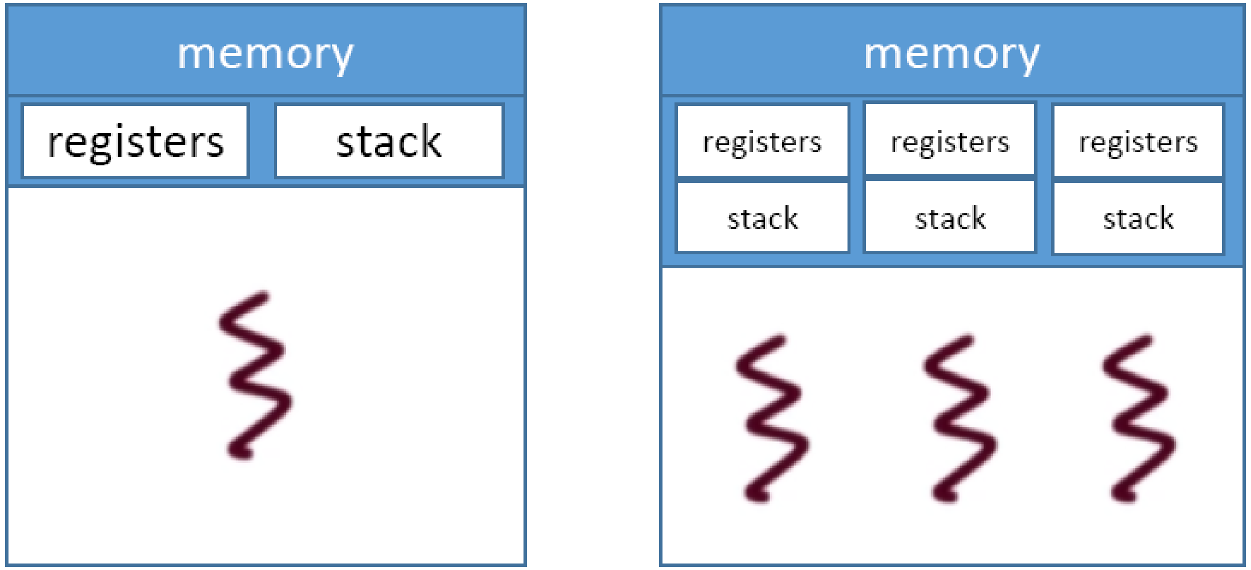
- A process needs at least one "thread" to run.
- A process is in "running" state if one of its threads are running.
- What actually runs on the CPU is the "thread" and not the process.

THREAD

- A process needs at least one "thread" to run.
- A process is in "running" state if one of its threads are running.
- What actually runs on the CPU is the "thread" and not the process.

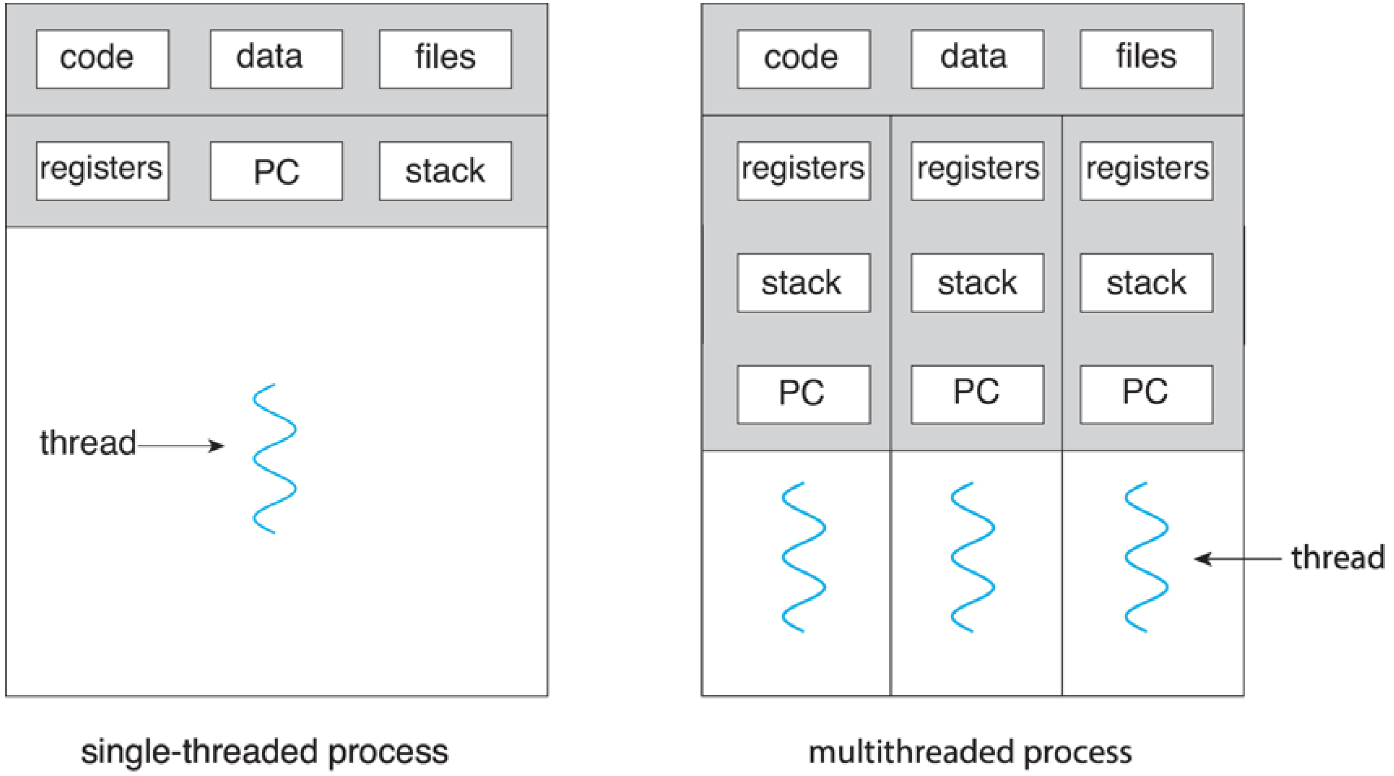
Thread : basic unit of CPU utilization

Single-threaded process vs multi-threaded processes



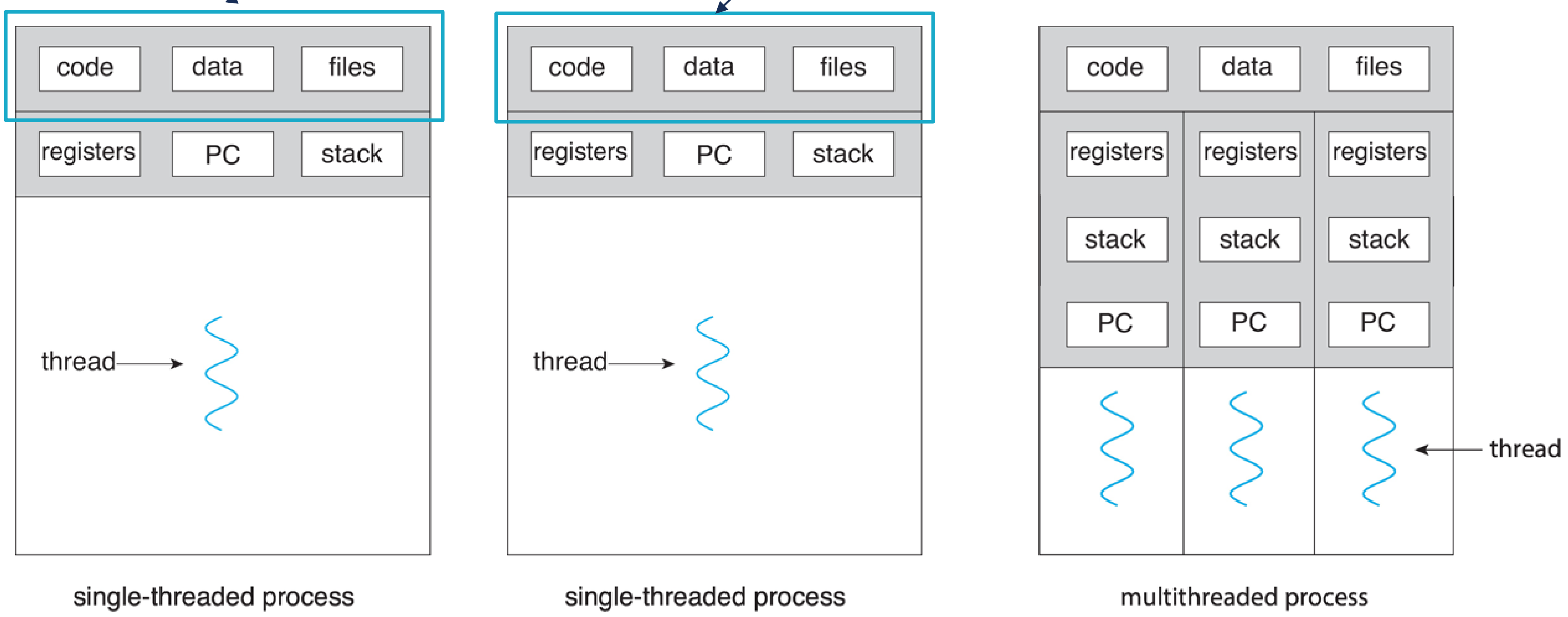
THREAD VS PROCESS

- Threads share memory but have different registers, stack and PC.
- Processes do not share memory.



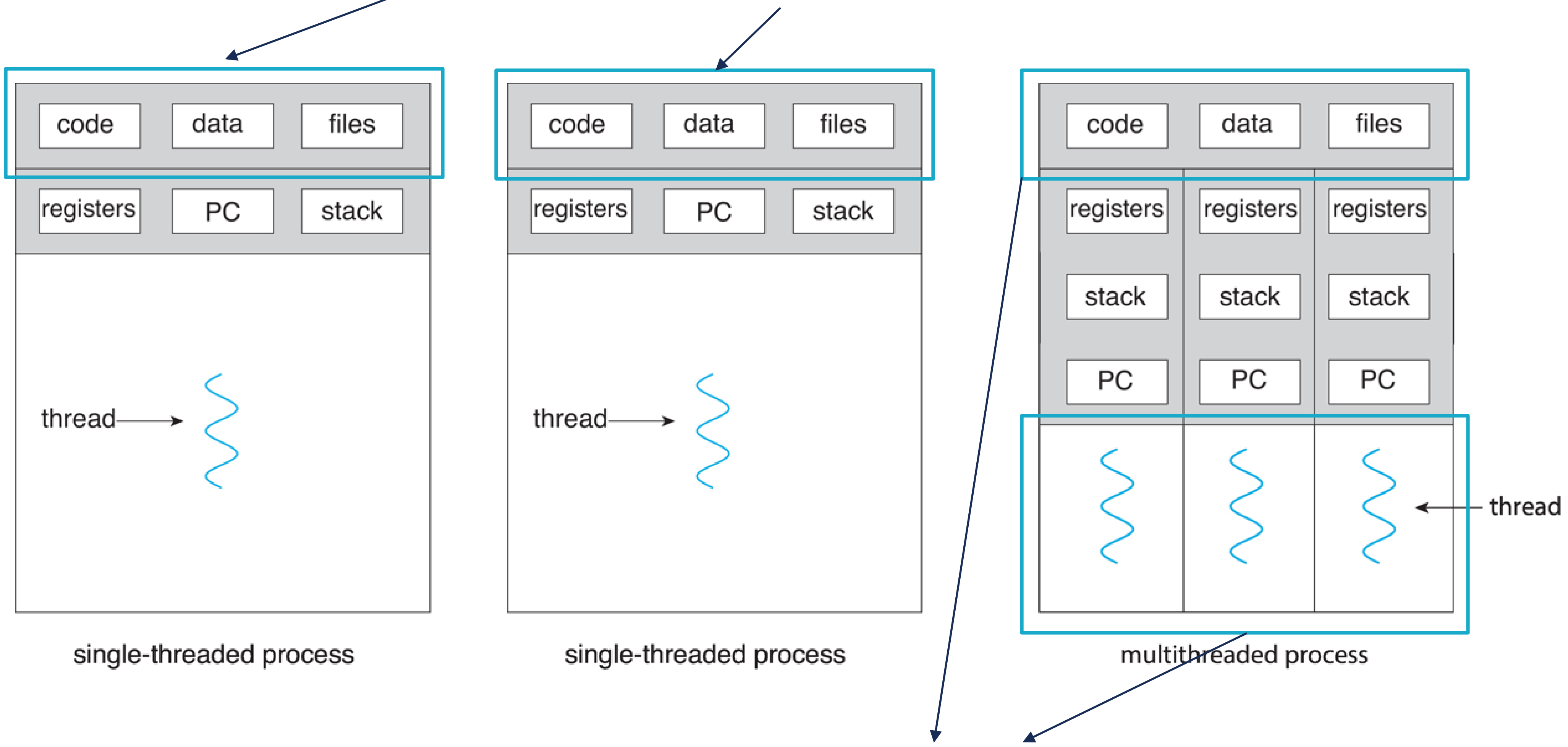
THREAD VS PROCESS

Two different processes have two different memory spaces



THREAD VS PROCESS

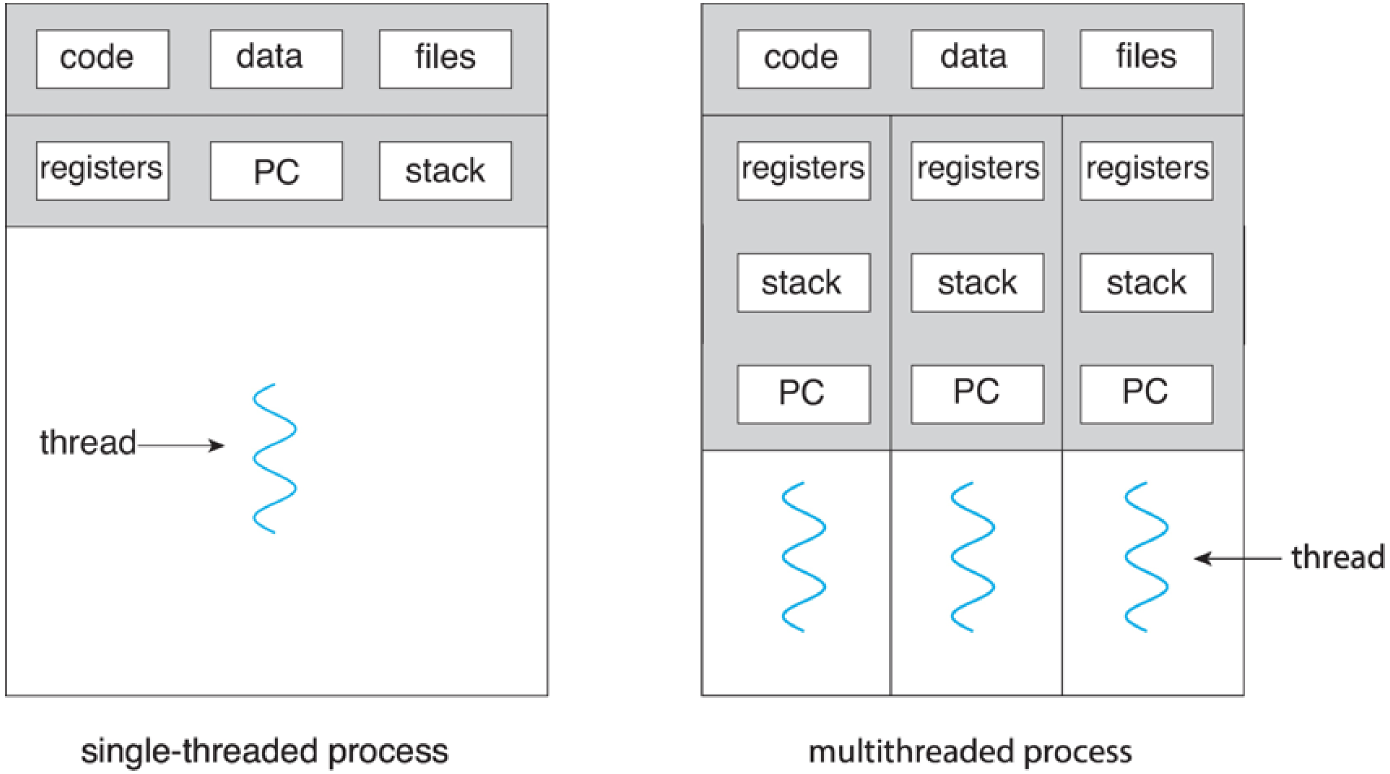
Two different processes have two different memory spaces



Threads of the same process share most of the memory space but have different stacks, registers, PC …

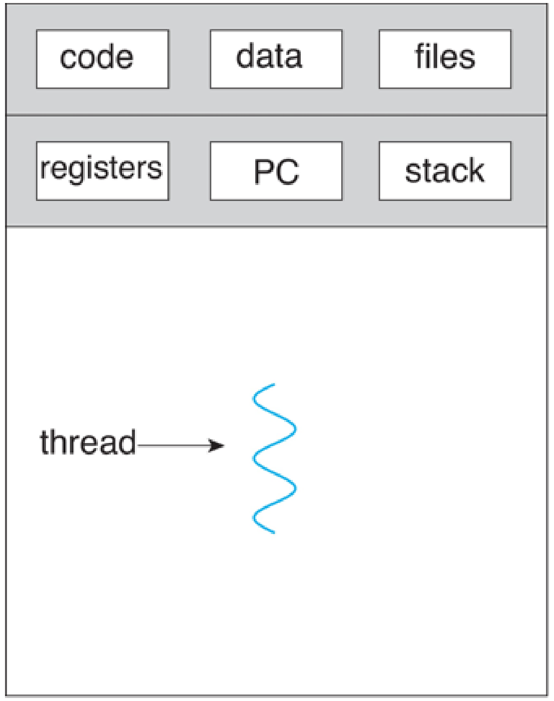
BENEFITS OF MULTITHREADING

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces

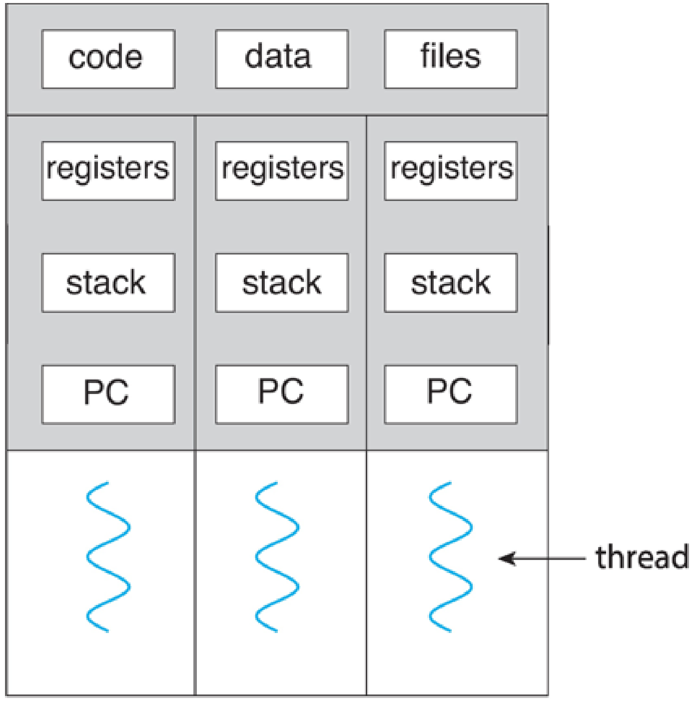


BENEFITS OF MULTITHREADING

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing



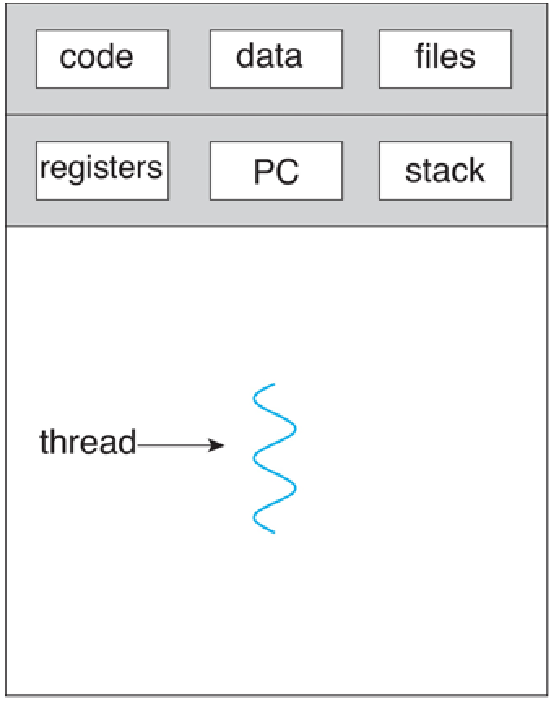
single-threaded process



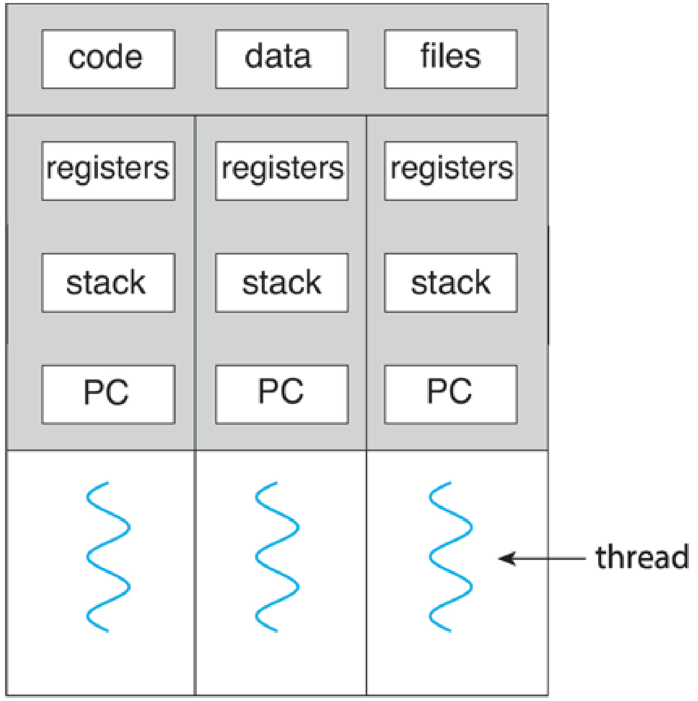
multithreaded process

BENEFITS OF MULTITHREADING

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than process switching



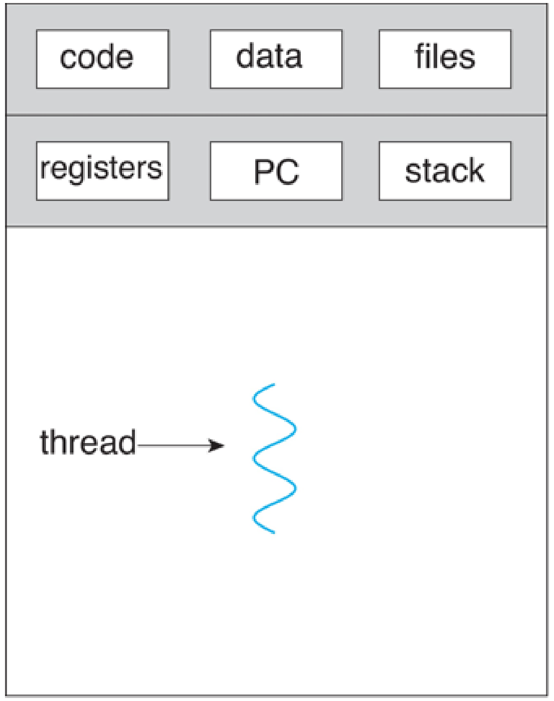
single-threaded process



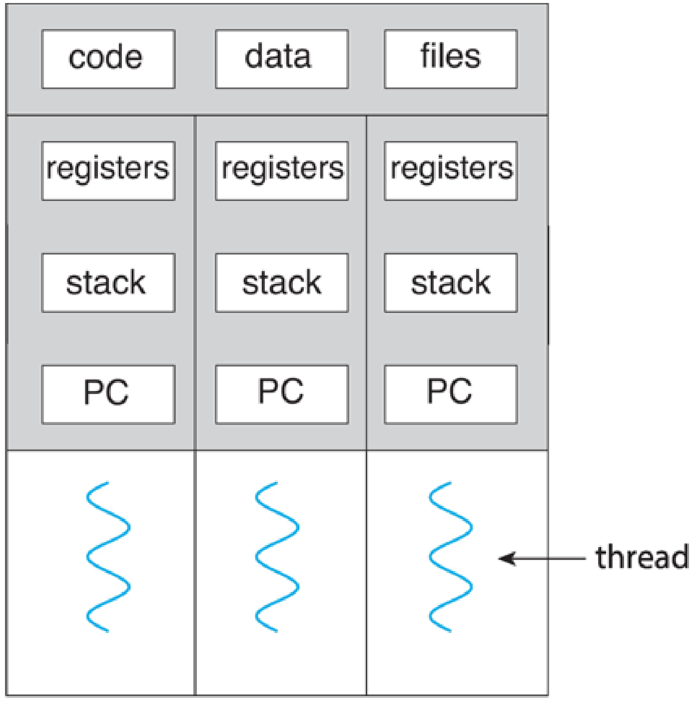
multithreaded process

BENEFITS OF MULTITHREADING

- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than process switching
- **Scalability –** process can take advantage of multicore architectures by having many threads running simultaneously.



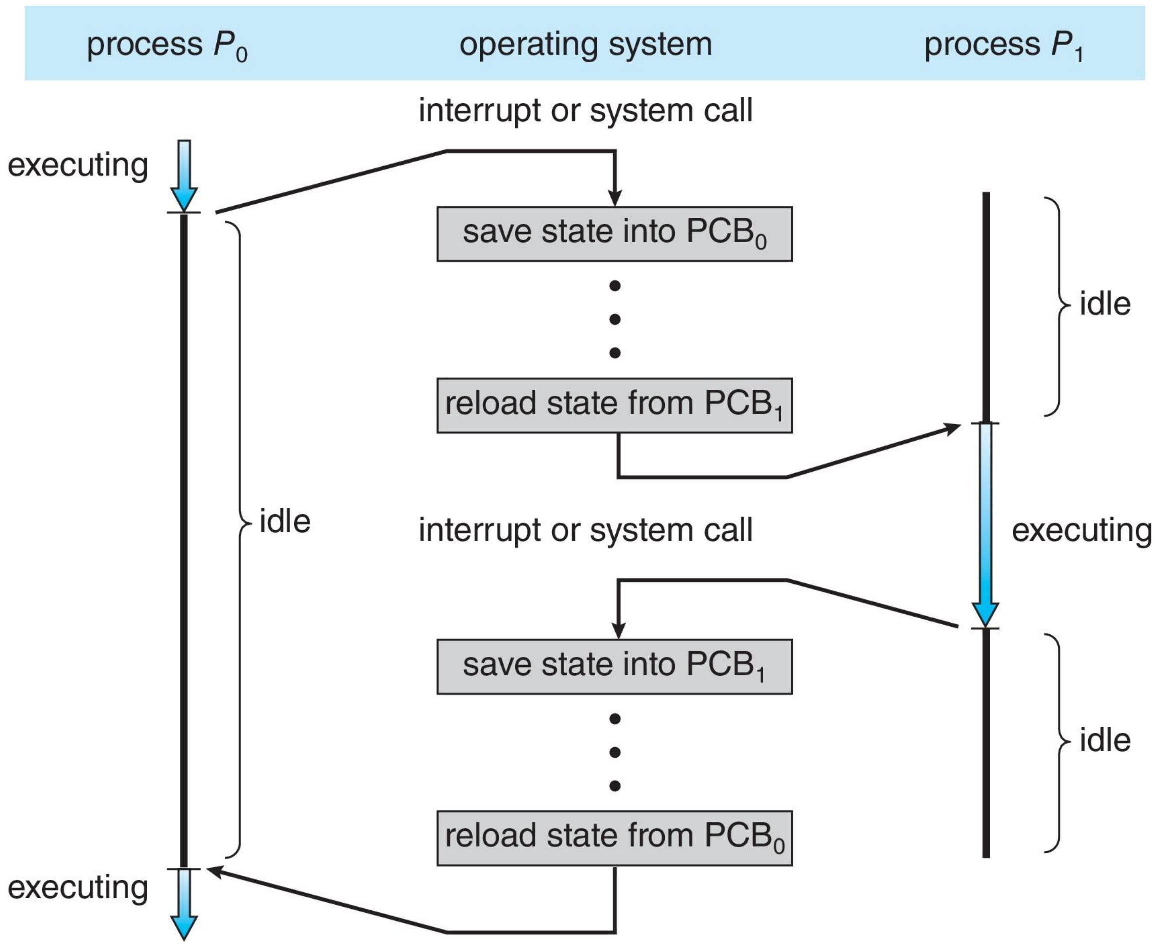
single-threaded process



multithreaded process

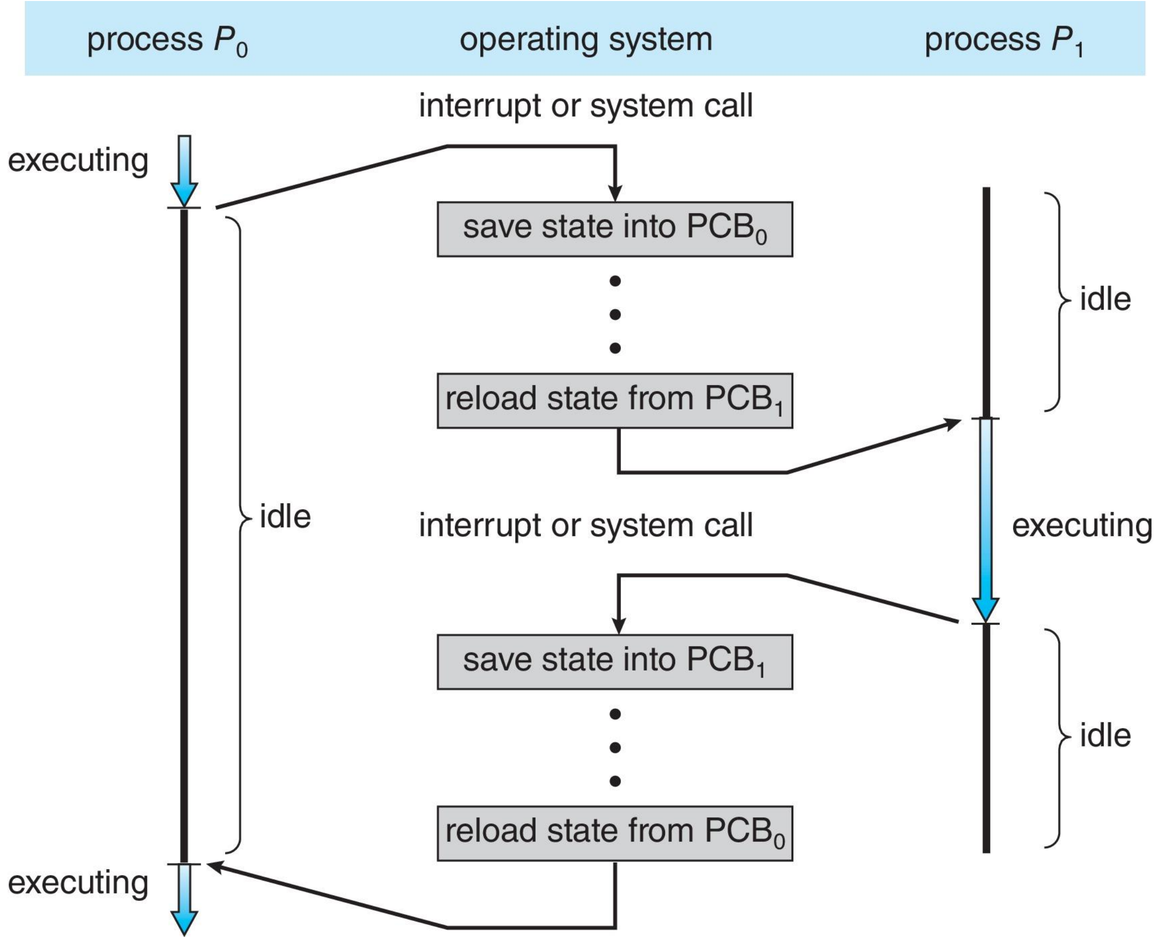
SIMULTANEOUS MULTI-THREADING

- Stalls are sometimes too short to justify process/ or thread switching.



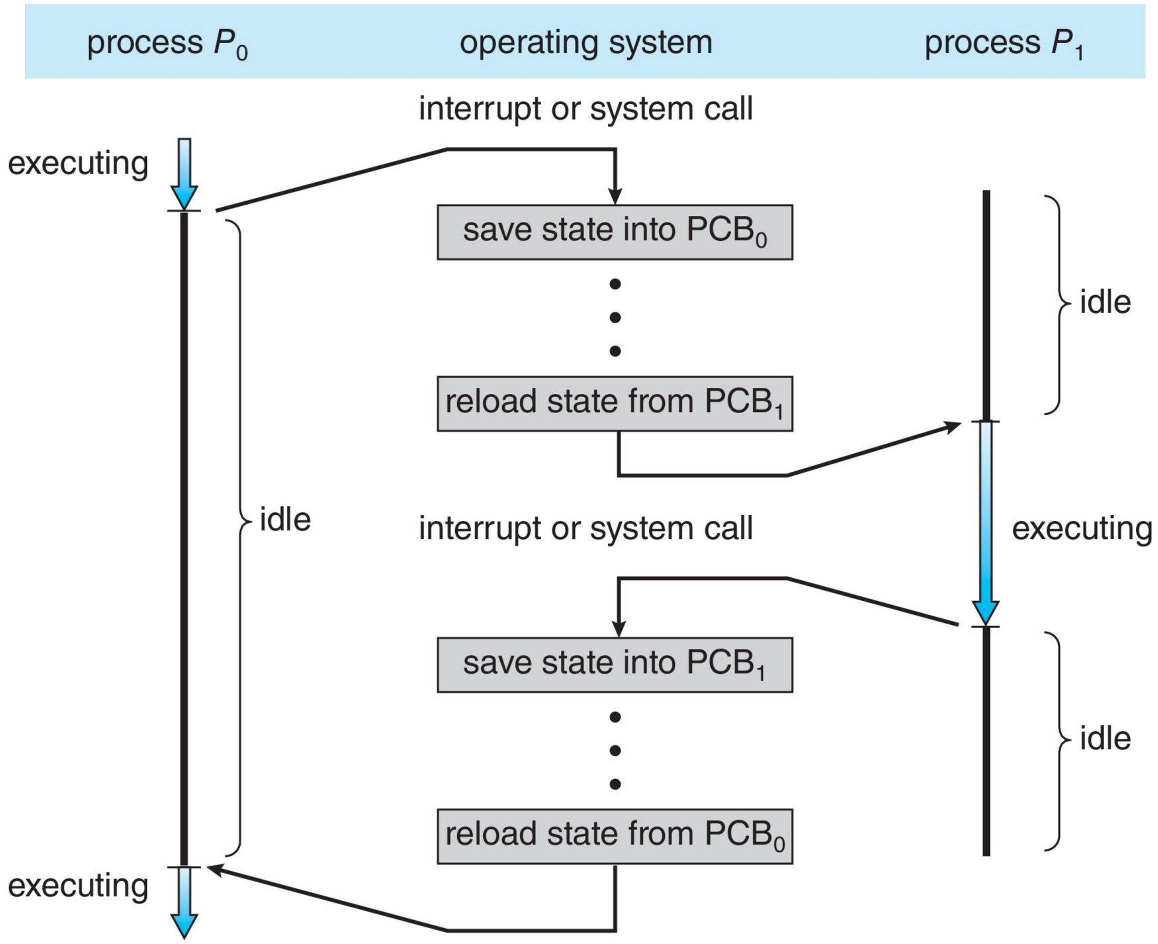
SIMULTANEOUS MULTI-THREADING

- Stalls are sometimes too short to justify process/ or thread switching.
- Short stalls occur very frequently.



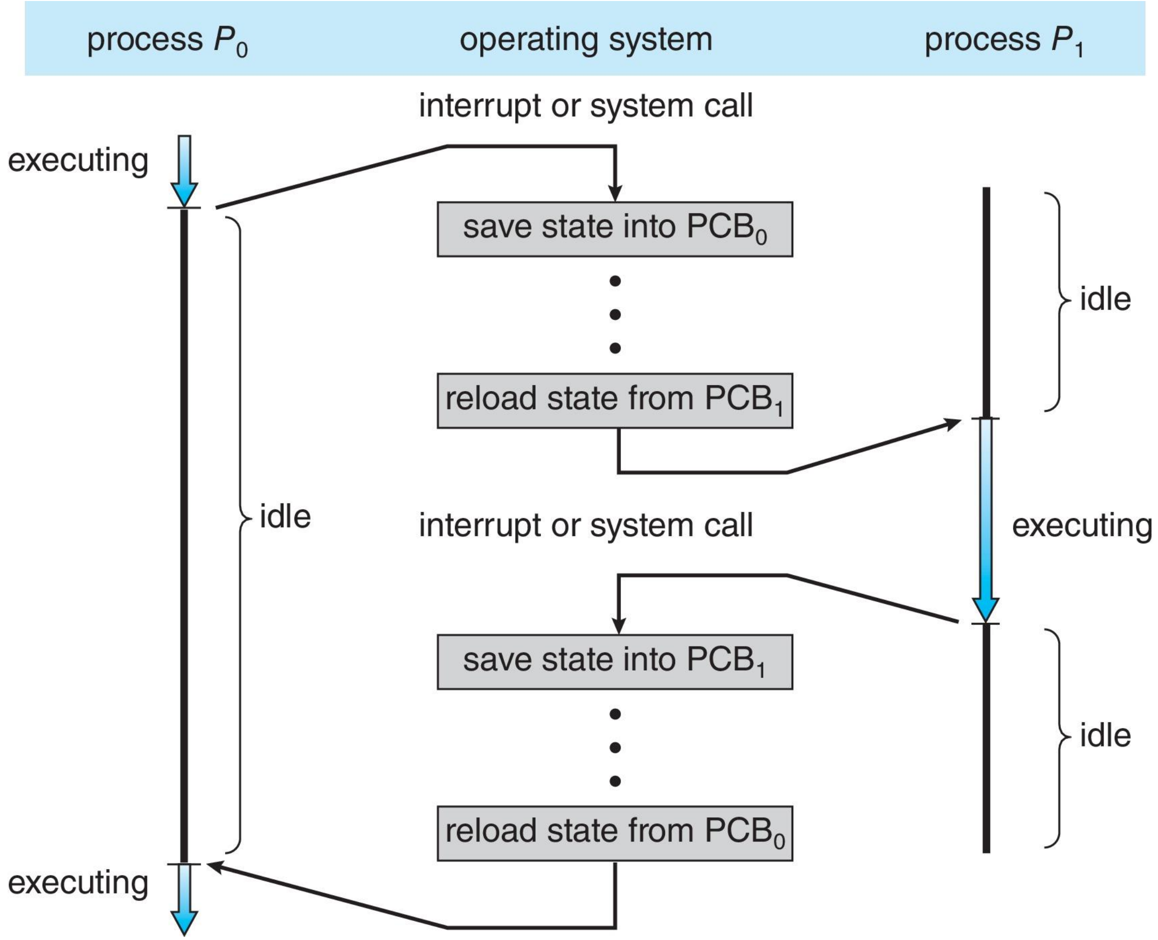
SIMULTANEOUS MULTI-THREADING

- Stalls are sometimes too short to justify process/ or thread switching.
- Short stalls occur very frequently.
- Solution: Simultaneous Multi Threading (SMT)

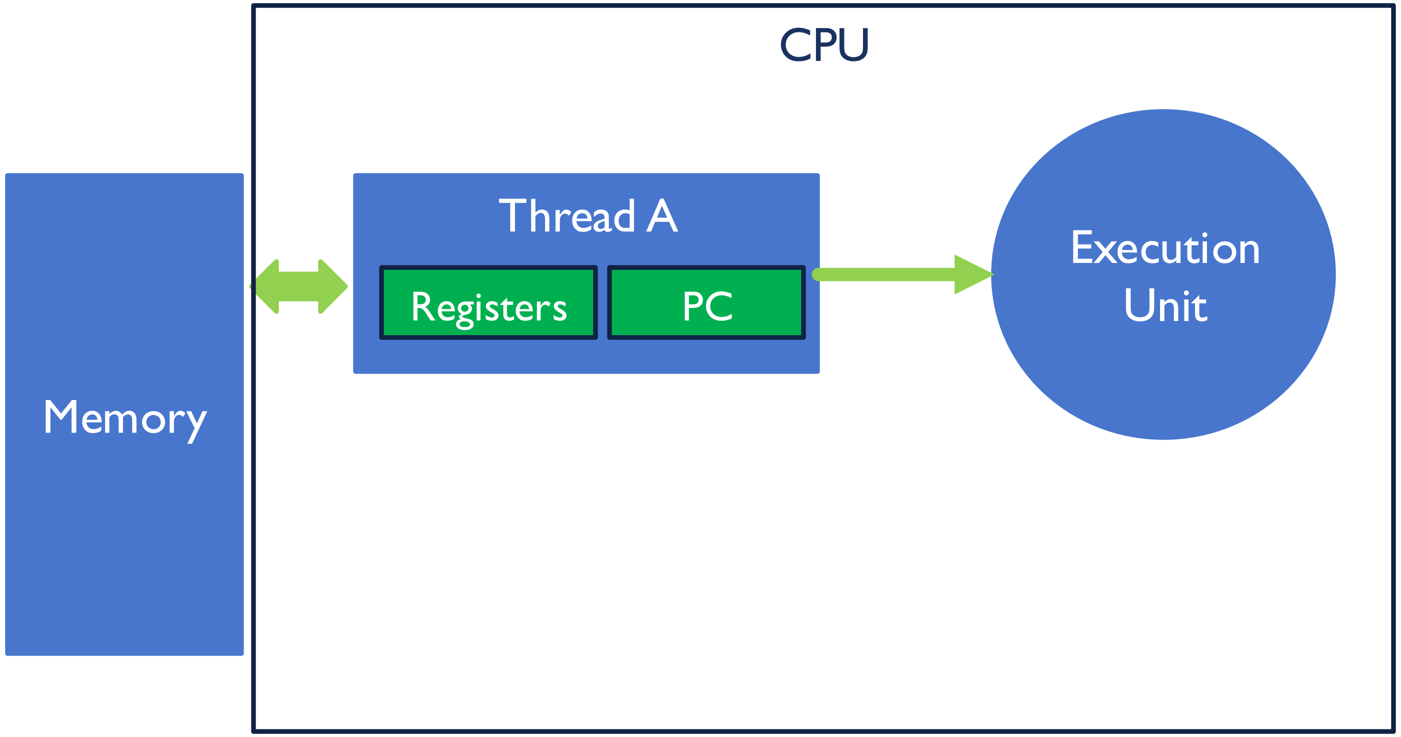


SIMULTANEOUS MULTI-THREADING

- Stalls are sometimes too short to justify process/ or thread switching.
- Short stalls occur very frequently.
- Solution: Simultaneous Multi Threading (SMT)
- Allows multiple thread to run on a single core/processor by exploiting stalls.

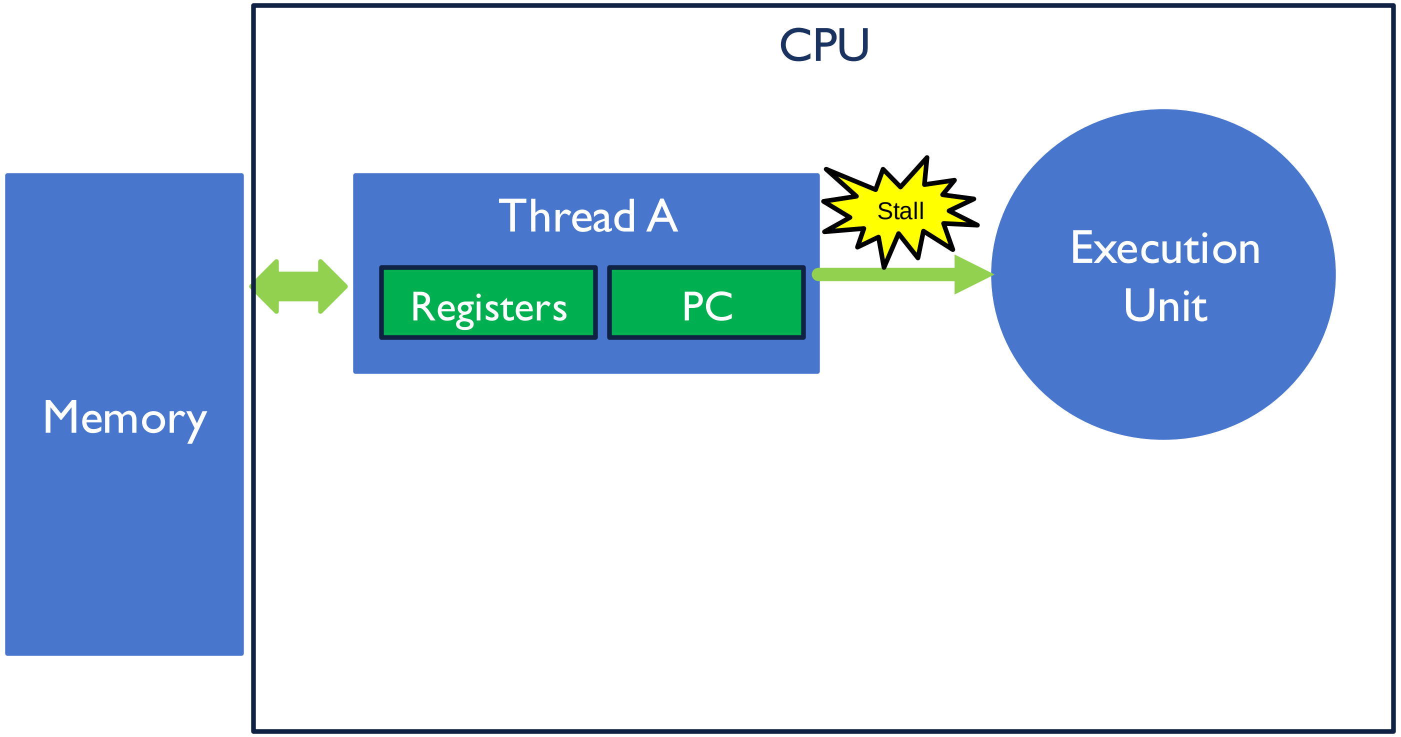


SIMULTANEOUS MULTI-THREADING



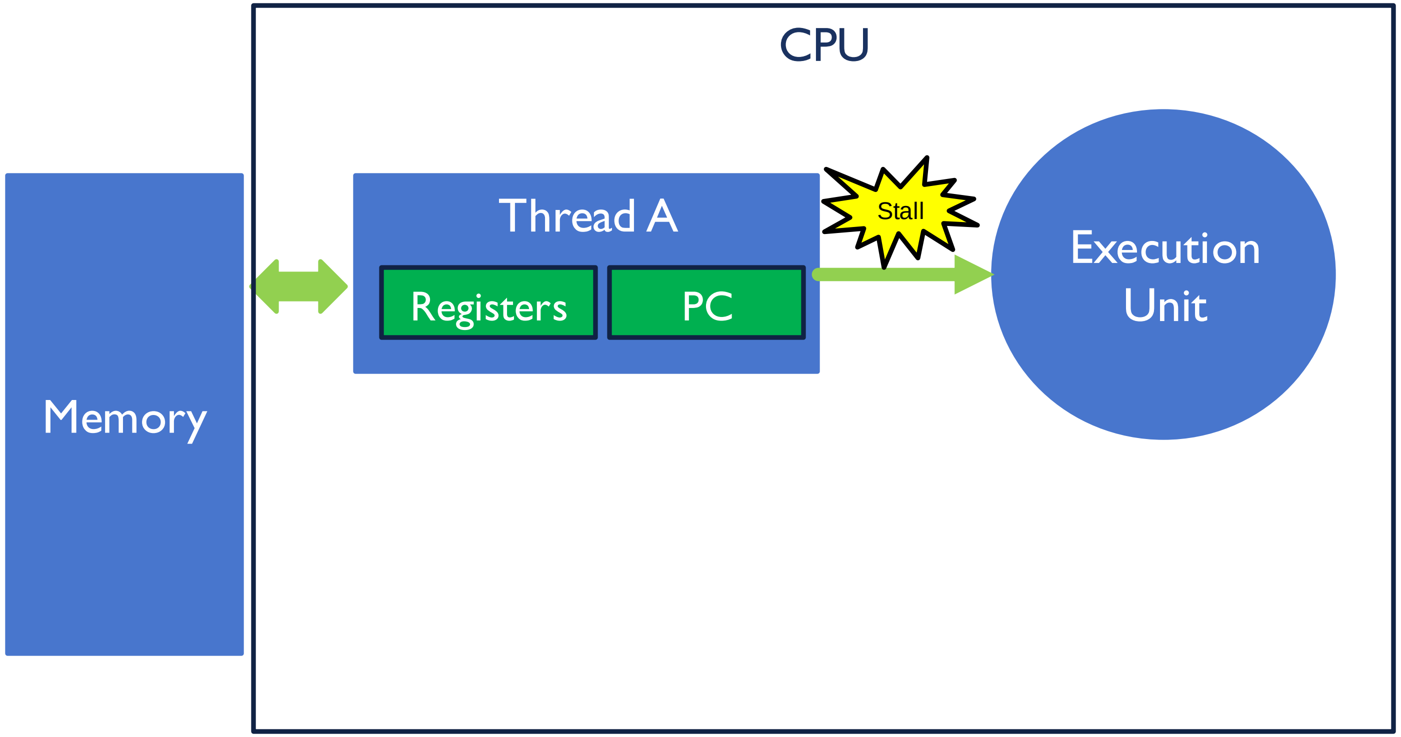
SIMULTANEOUS MULTI-THREADING

- Stall Occurs



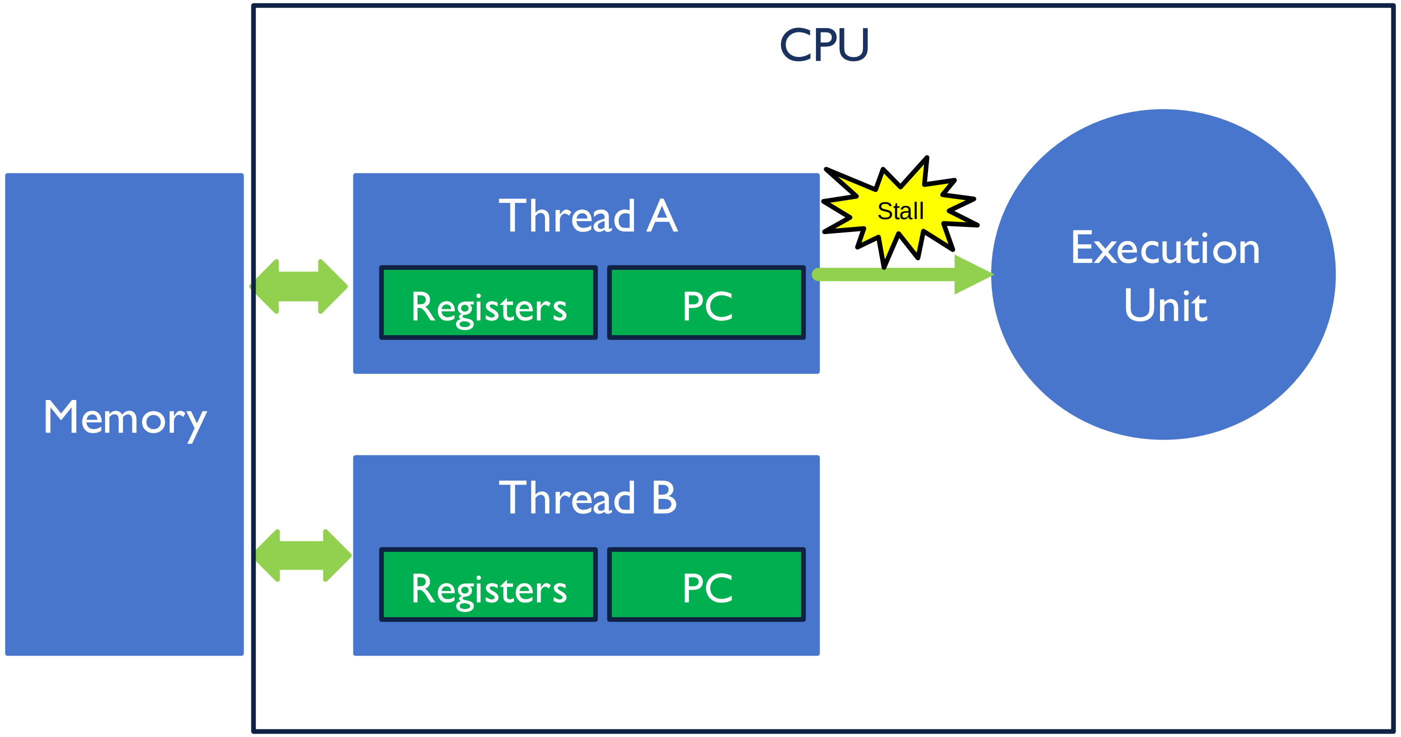
SIMULTANEOUS MULTI-THREADING

- Stall Occurs
- Short Stall: Cache Miss: 4~25 cycle. Not worth saving loading.
- CPU time is wasted whether we switch or not.



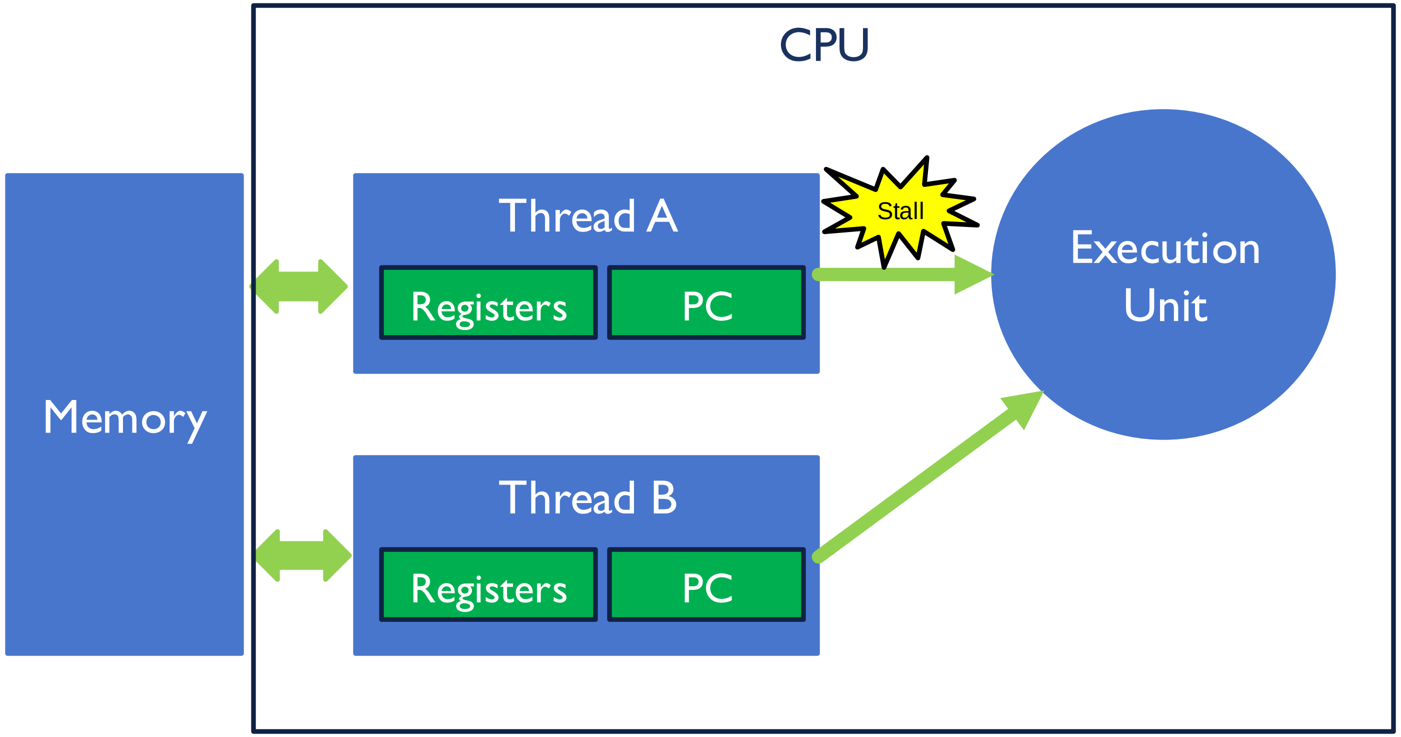
SIMULTANEOUS MULTI-THREADING

- What if we had another set of registers to support a non-executing thread?



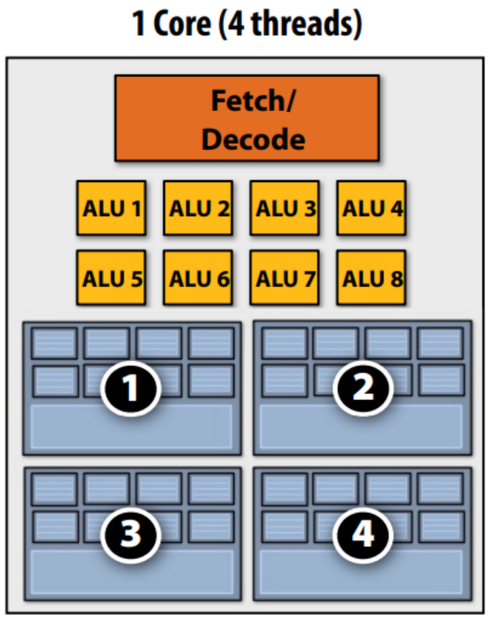
SIMULTANEOUS MULTI-THREADING

- What if we had another set of registers to support a non-executing thread?
- Thread B can begin executing immediately when thread A stalls.



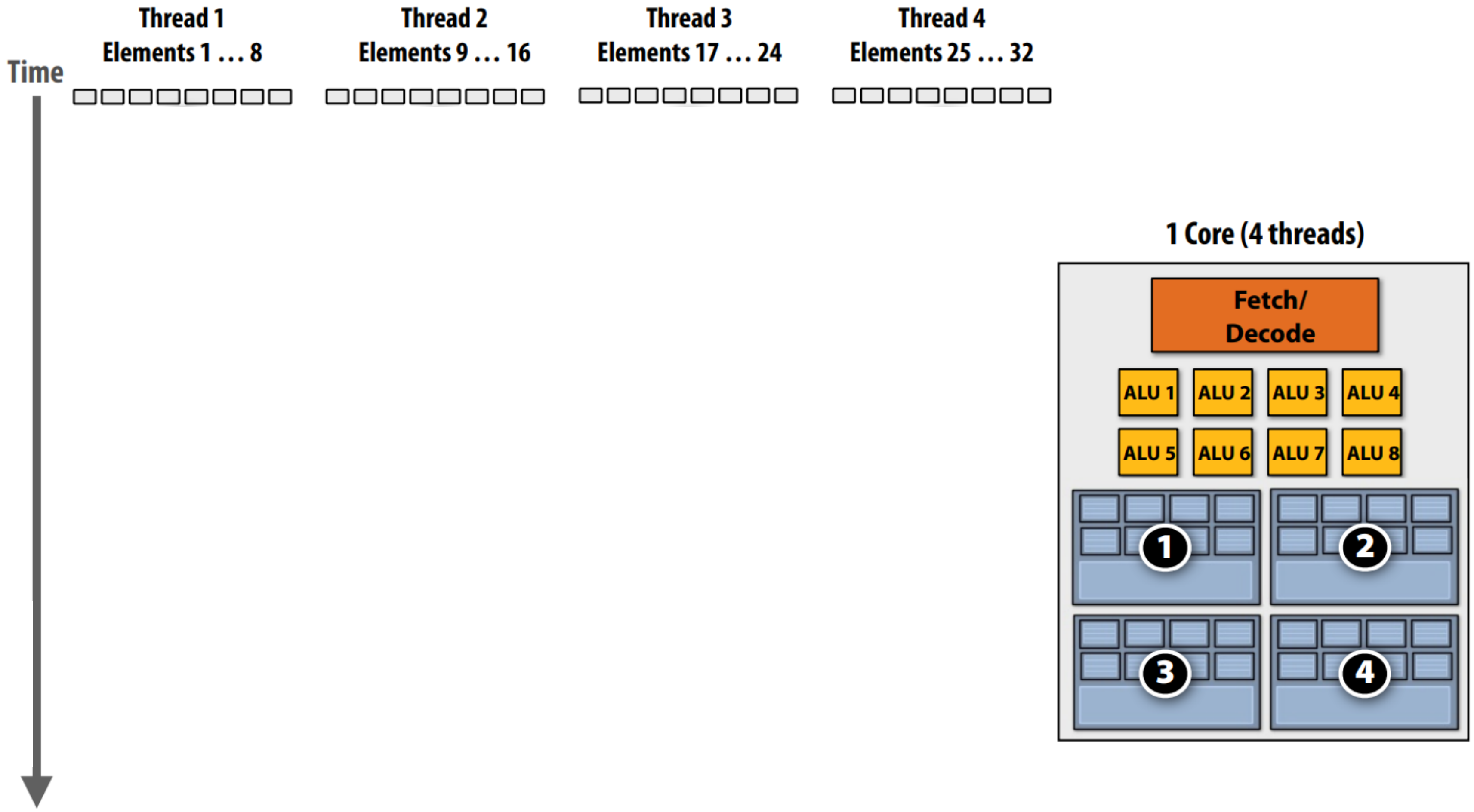
SIMULTANEOUS MULTI-THREADING

- Modern processors support multiple threads per core.
- Cores also have multiple ALUs allowing single thread to execute multiple instructions.



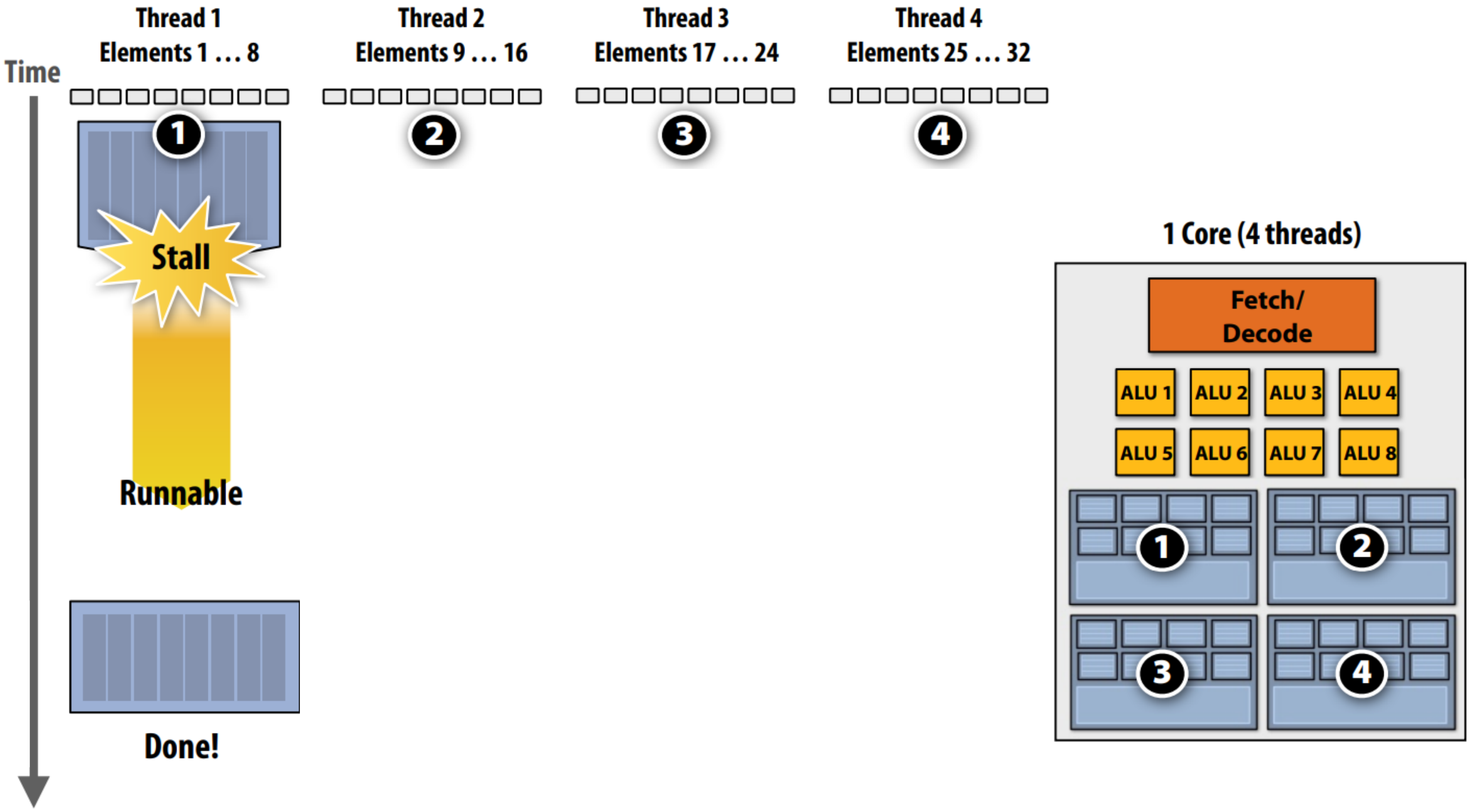
SIMULTANEOUS MULTI-THREADING

- Modern processors support multiple threads per core.
- Cores also have multiple ALUs allowing single thread to execute multiple instructions.
- When once thread stalls, another starts executing, no CPU time is lost.



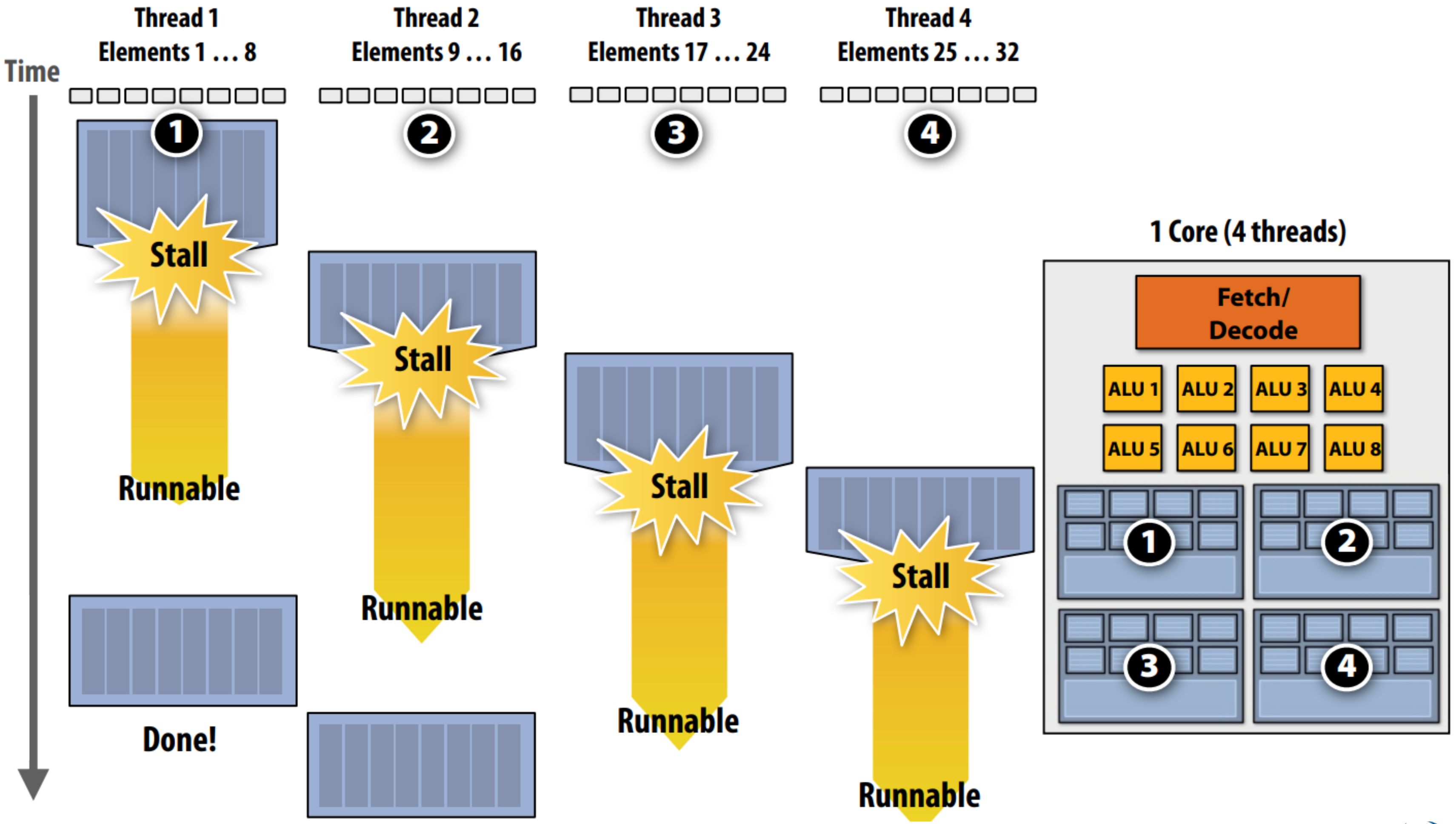
SIMULTANEOUS MULTI-THREADING

- Modern processors support multiple threads per core.
- Cores also have multiple ALUs allowing single thread to execute multiple instructions.
- When once thread stalls, another starts executing, no CPU time is lost.

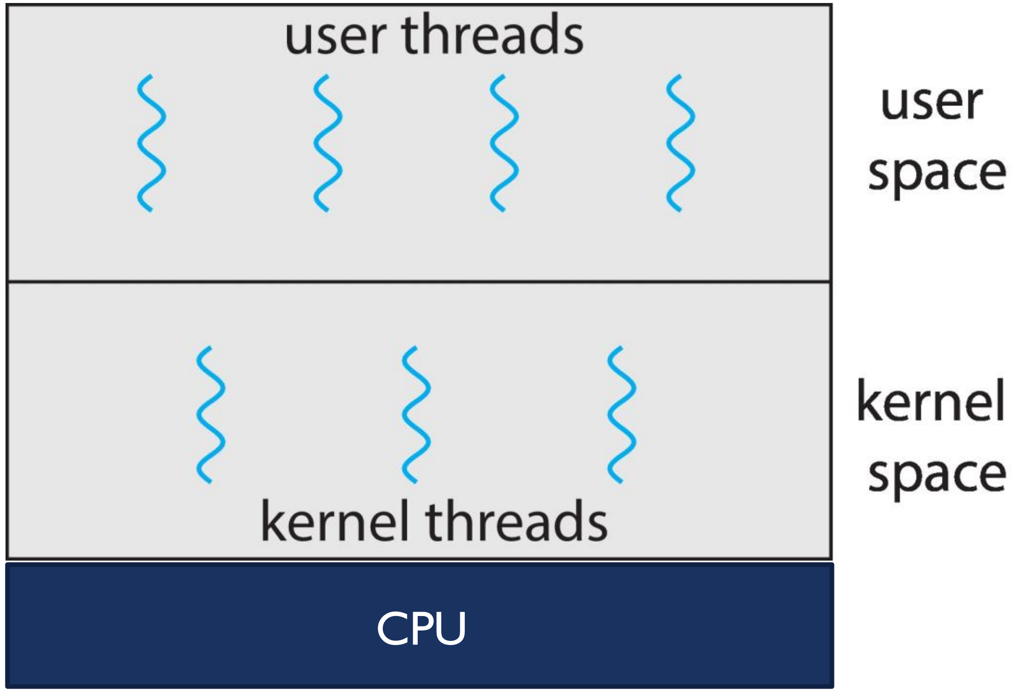


SIMULTANEOUS MULTI-THREADING

- Modern processors support multiple threads per core.
- Cores also have multiple ALUs allowing single thread to execute multiple instructions.
- When once thread stalls, another starts executing, no CPU time is lost.

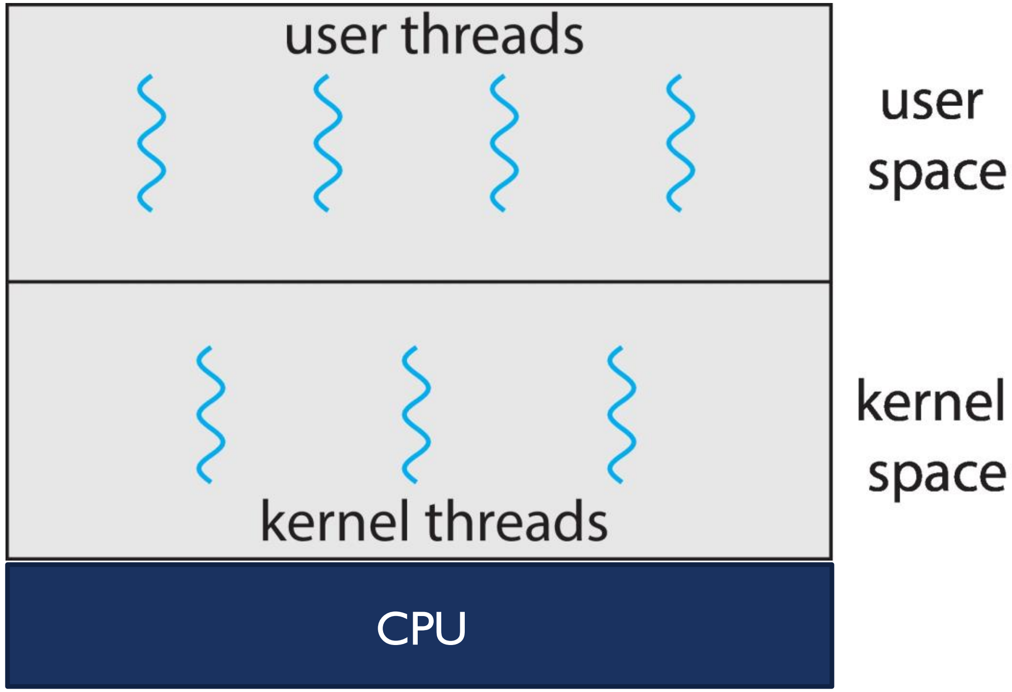


USER THREADS VS KERNEL THREADS

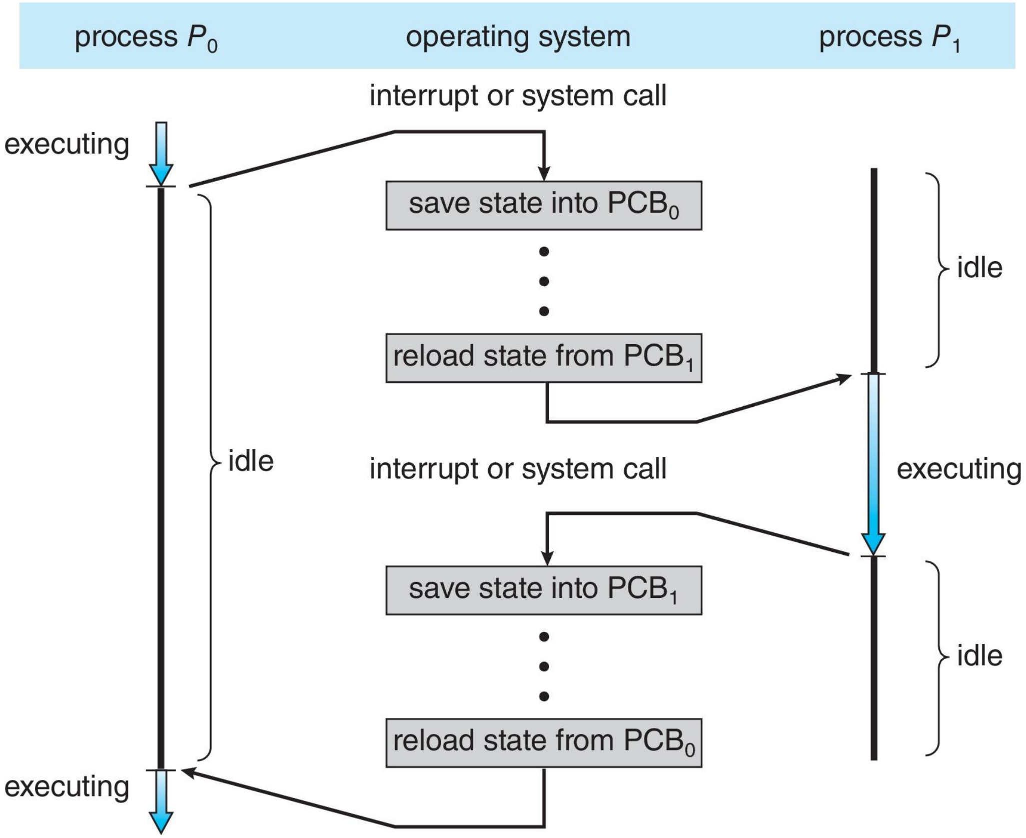


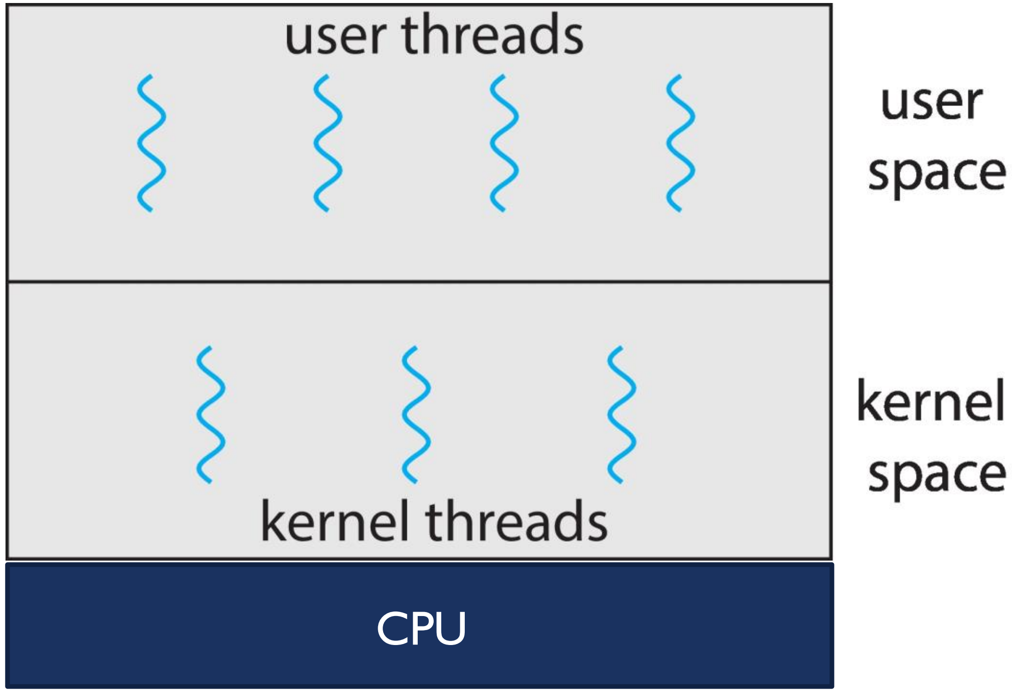
USER THREADS VS KERNEL THREADS

- User 'threads' are virtual/fake unless supported by a kernel thread.
- Any running thread is linked and is running on a kernel thread.



USER THREADS VS KERNEL THREADS

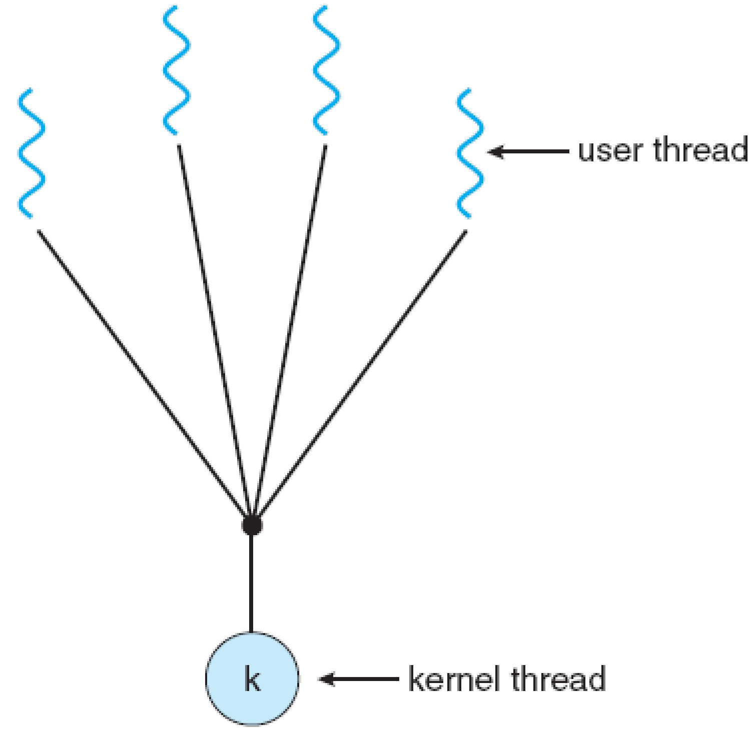




WESTERN
WASHINGTON UNIVERSITY

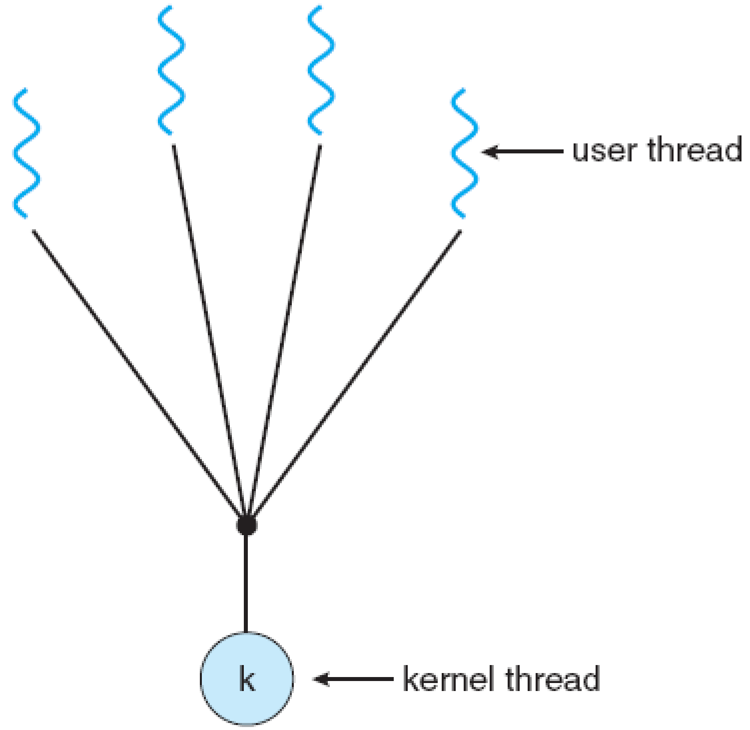
MANY TO ONE MAPPING

- Many user-level threads mapped to single kernel thread.
- No actual multithreading.
- One thread blocking causes all to block.
- Multiple threads may not run in parallel.



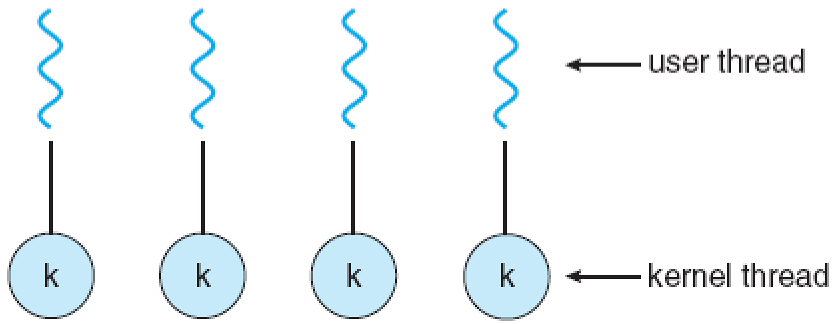
MANY TO ONE MAPPING

- Many user-level threads mapped to single kernel thread.
- No actual multithreading.
- One thread blocking causes all to block.
- Multiple threads may not run in parallel.
- Purpose: compatibility; allows multithreaded applications to run on systems that do no support multithreading.
- Few systems currently use this model.
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



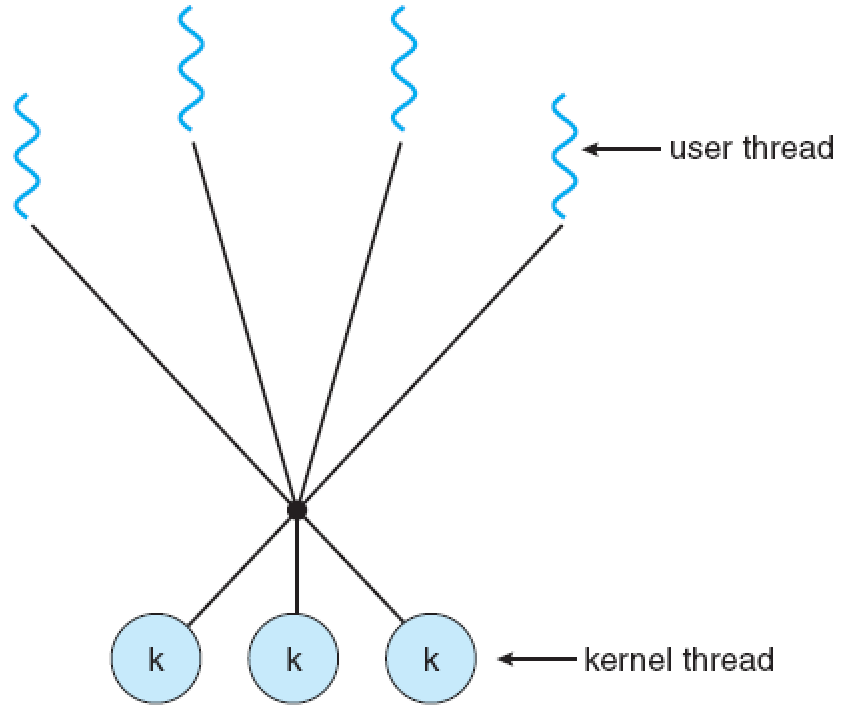
ONE TO ONE MAPPING

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux



MANY TO MANY

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Windows with the *ThreadFiber* package
- Not very common as modern operating systems enough resources these days.



THREAD LIBRARIES: THREAD CREATION

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS

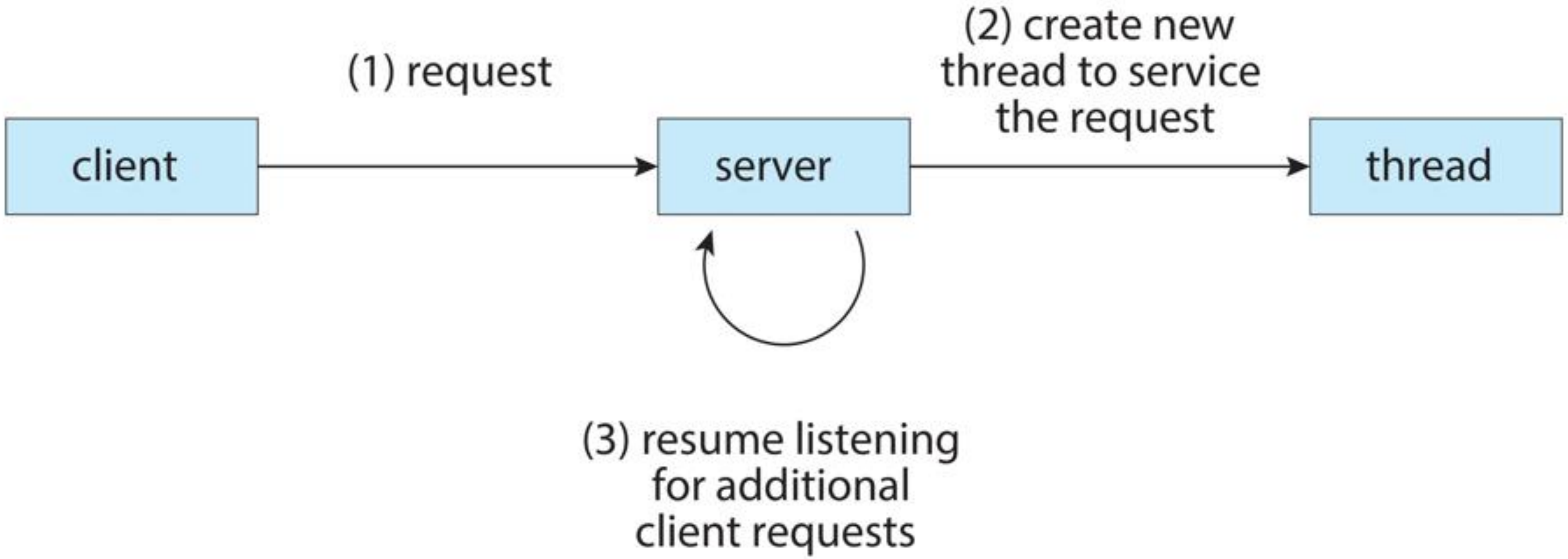
THREADS IN LINUX

- Linux refers to them as ***tasks*** rather than ***threads***
- Thread creation is done through **`clone()`** system call
- **`clone()`** allows a child task to share the address space of the parent task (process)
  - Flags control behavior

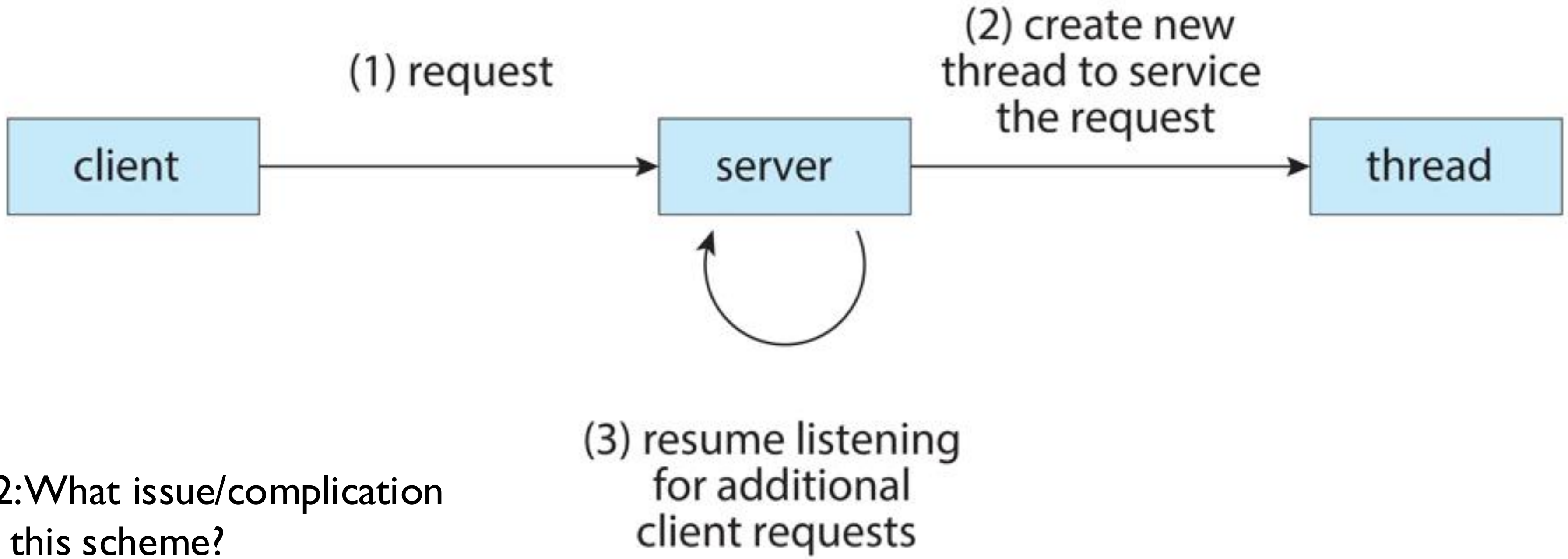
flag	meaning
CLONE_FS	File-system information is shared.
CLONE_VM	The same memory space is shared.
CLONE_SIGHAND	Signal handlers are shared.
CLONE_FILES	The set of open files is shared.

- **`struct task_struct`** points to process data structures (shared or unique)

MULTITHREADING FOR SERVERS

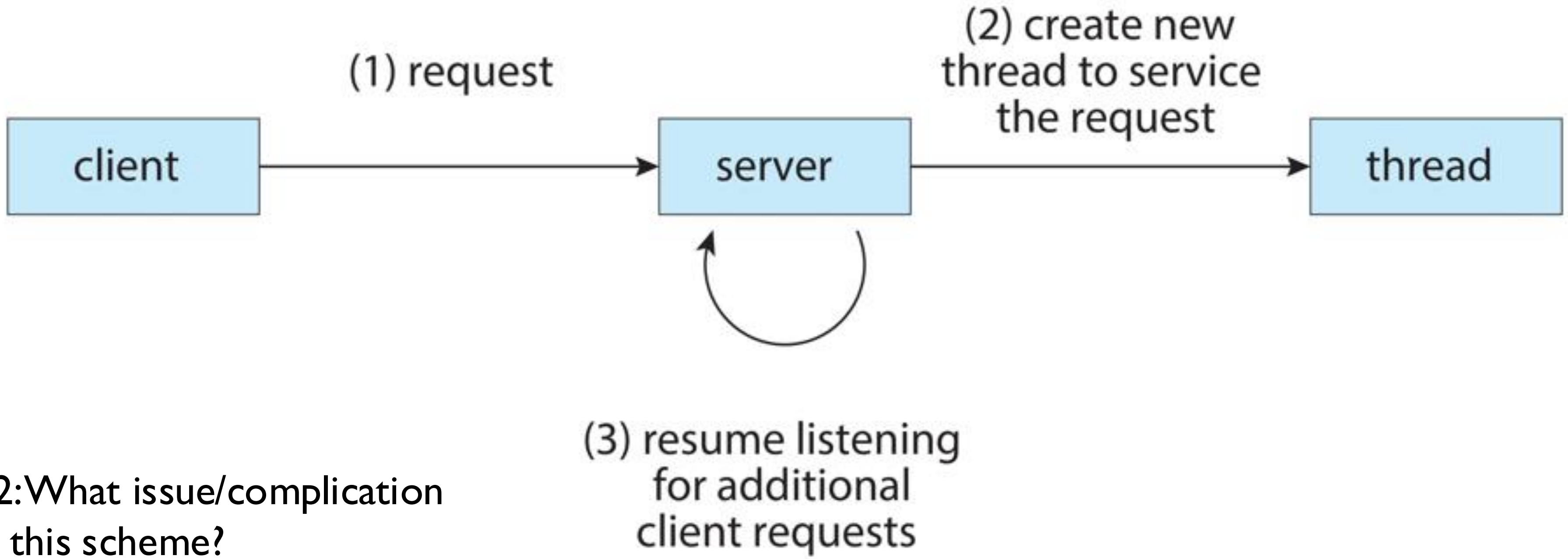


MULTITHREADING FOR SERVERS



Worksheet Q2: What issue/complication could occur in this scheme?

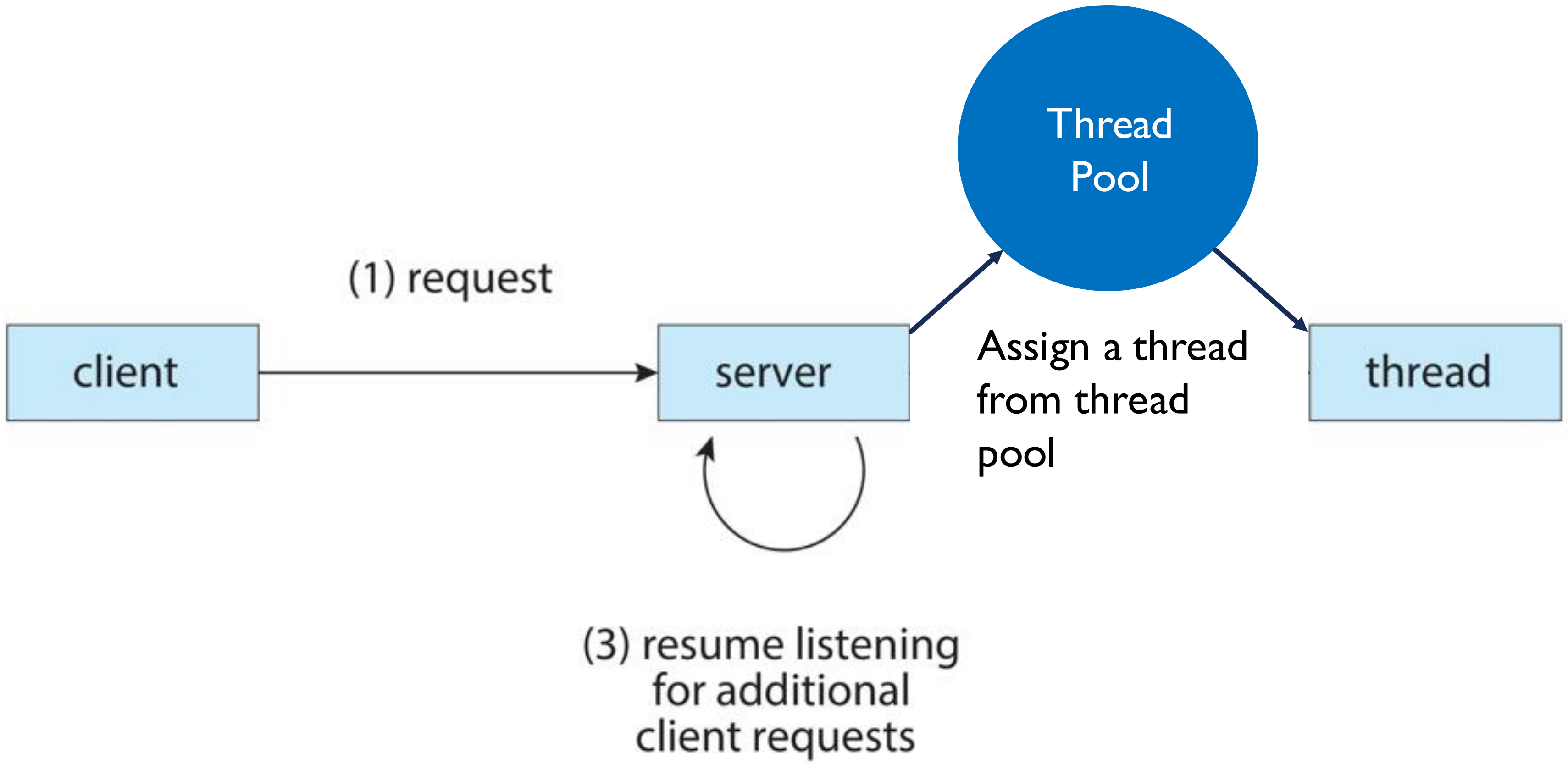
MULTITHREADING FOR SERVERS



Worksheet Q2: What issue/complication could occur in this scheme?

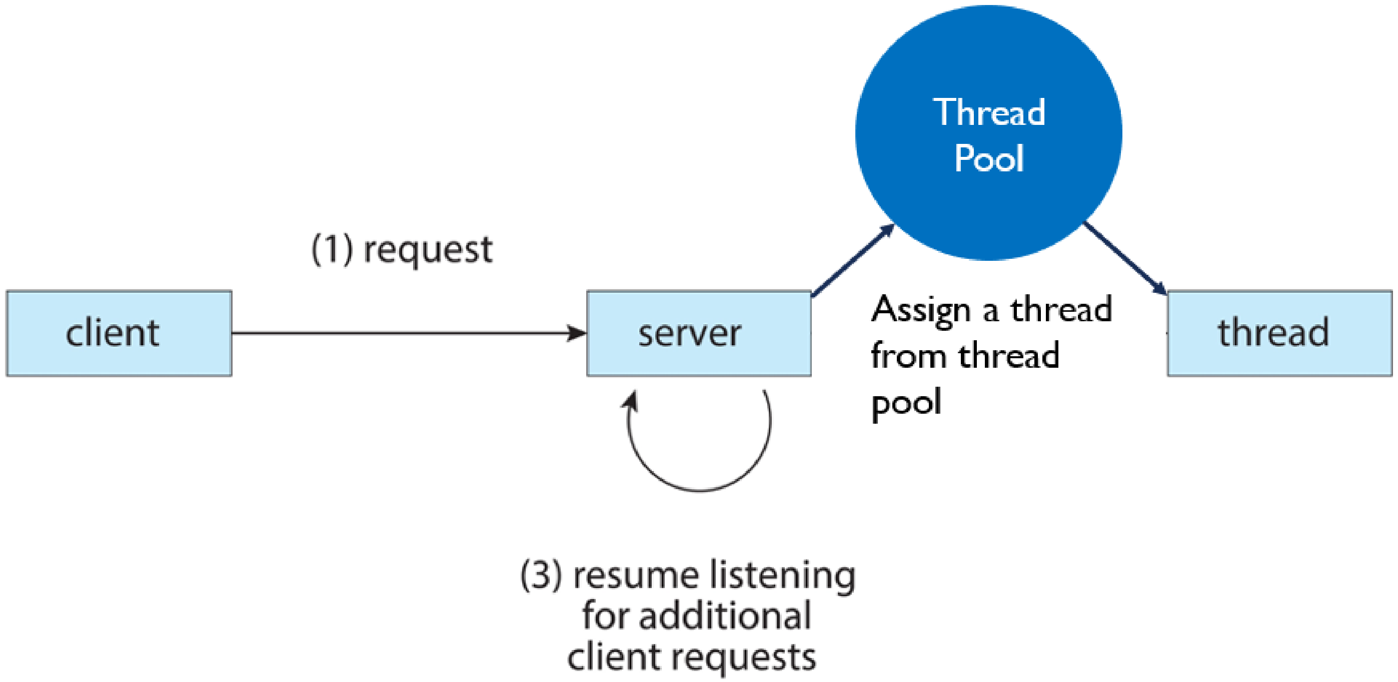
Possibility of creating too many threads that can crash the system!

THREAD POOLS FOR SERVERS



THREAD POOLS

- Create a number of threads in a pool where they await work
- Advantages:
  - Slightly faster to service a request with an existing thread than creating a new thread.
  - Allows the number of threads in the application(s) to be bound to the size of the pool and avoiding crashes or overloading the system.



THREAD TERMINATION

• Asynchronous cancellation	A thread immediately terminates the target thread
• Deferred cancellation	The target thread periodically “checks in” to find out if it should be terminated; if so, it does so in an orderly fashion

THREAD TERMINATION

• Asynchronous cancellation	A thread immediately terminates the target thread
• Deferred cancellation	The target thread periodically “checks in” to find out if it should be terminated; if so, it does so in an orderly fashion

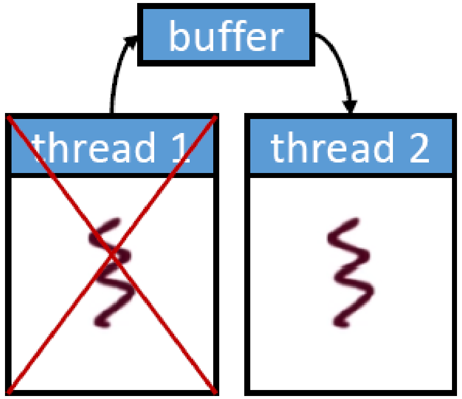
Q: Why might asynchronous thread cancellation be problematic?

THREAD TERMINATION

- Asynchronous cancellation — A thread immediately terminates the target thread
- Deferred cancellation — The target thread periodically “checks in” to find out if it should be terminated; if so, it does so in an orderly fashion

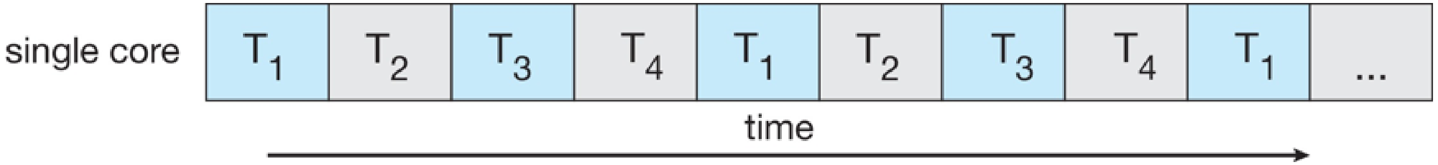
Q: Why might asynchronous thread cancellation be problematic?

Threads take on distinct roles in a computation/process, and information might need to be shared among threads. Terminating a thread prematurely might impact the thread that is killed, AND the thread that was in the process of communication with the killed thread.

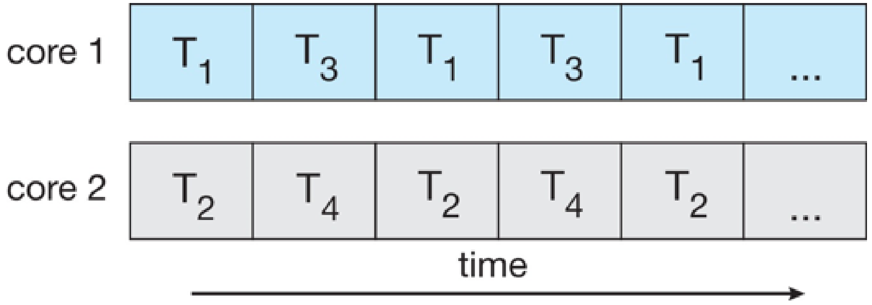


PARALLELISM AND CONCURRENCY

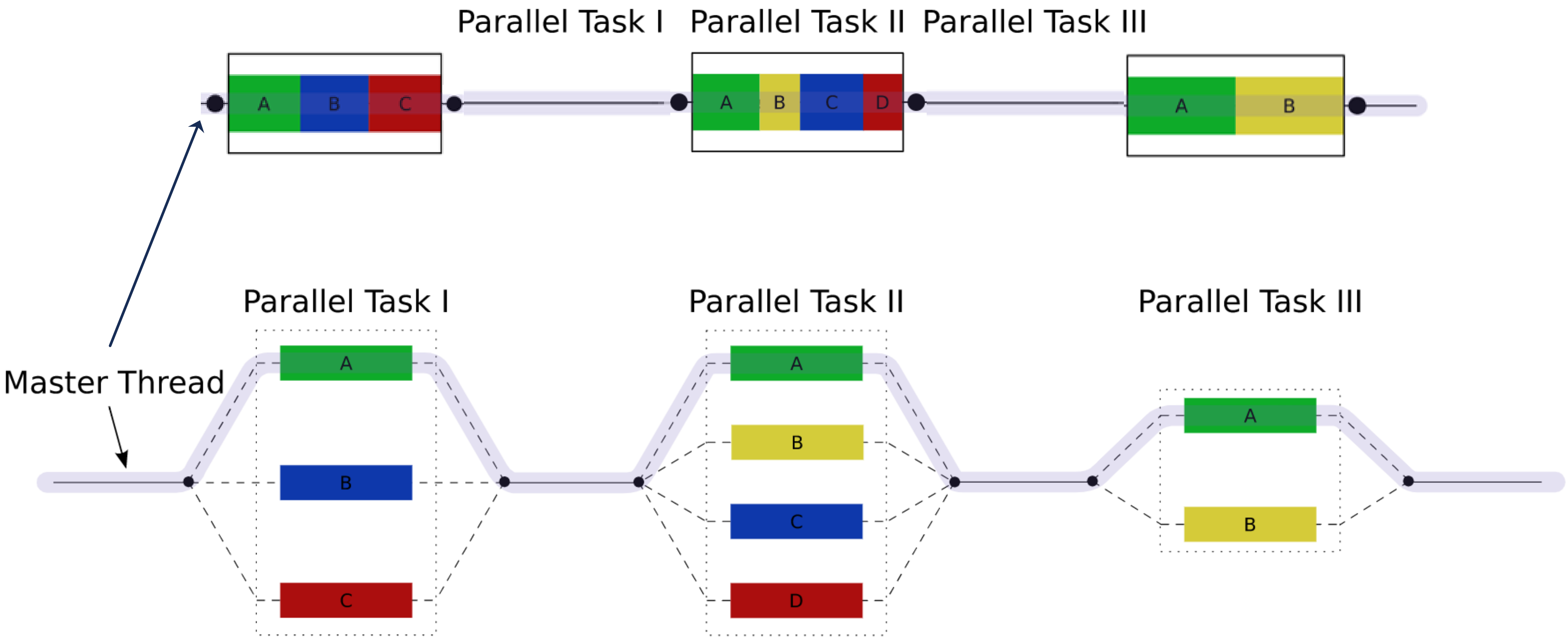
- **Concurrent execution on single-core system:**



- **Parallelism on a multi-core system:**



FORK-JOIN PARALLEL TASKS



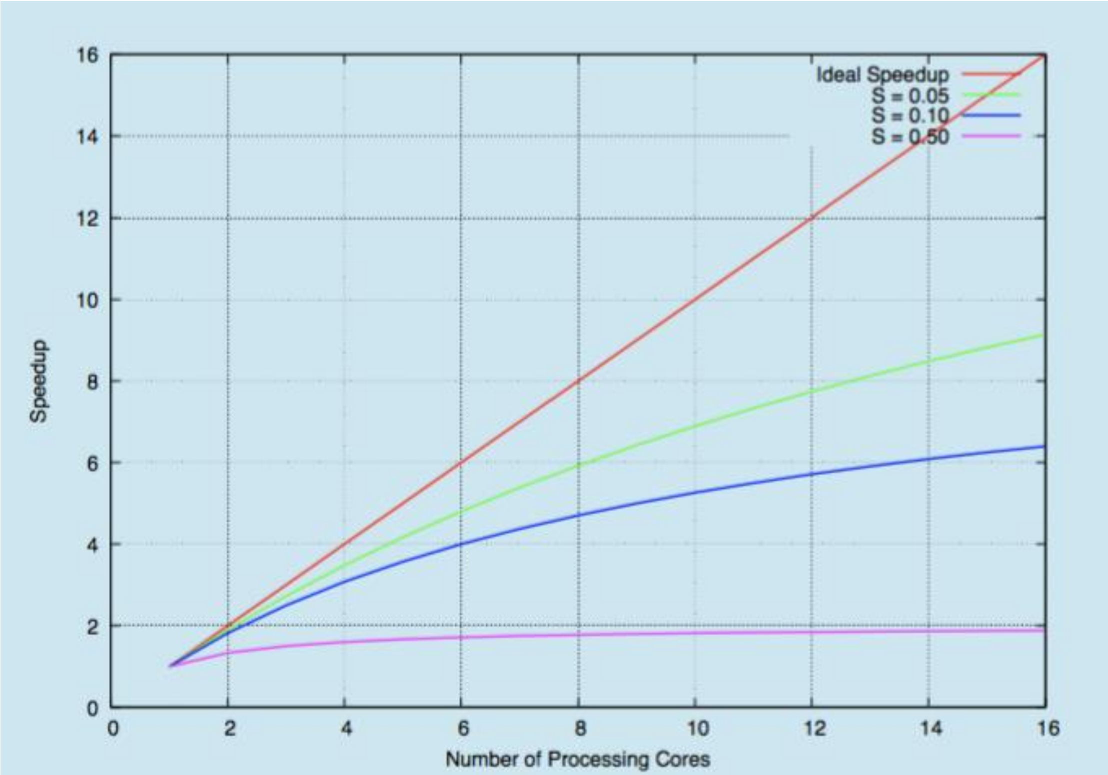
PERFORMANCE ENHANCEMENT CAP: AMDAHL'S LAW

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- *S* is serial portion
- *N* processing cores

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

AMDAHL'S LAW


Ideal Speedup
S = 0.05
S = 0.10
S = 0.50
Speedup
Number of Processing Cores
0
2
4
6
8
10
12
14
16

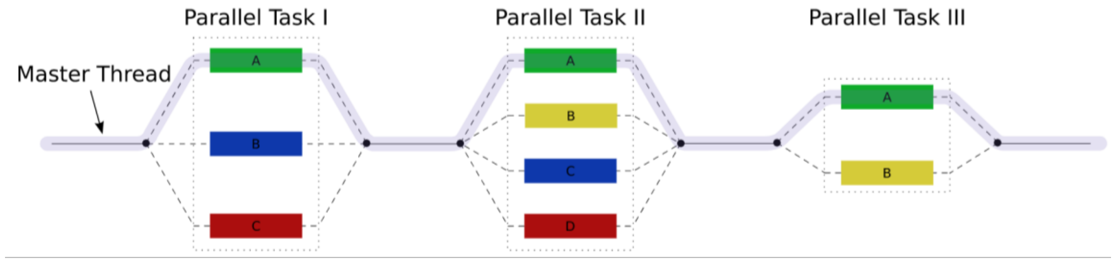
OpenMP (OPEN MULTI-PROCESSING) API

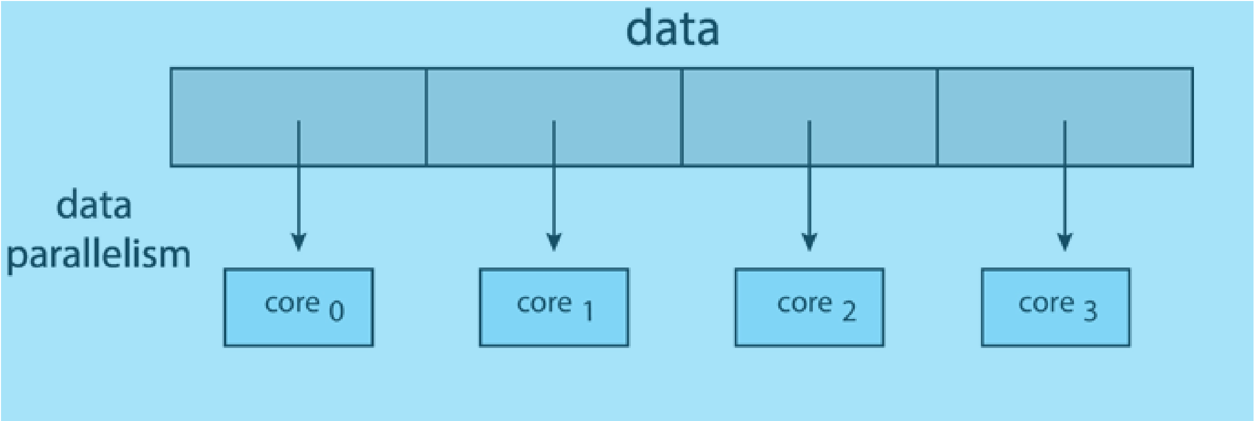
- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments

```c
#include <omp.h>
#include <stdio.h>

int main() {
    int i, n = 100;
    double x[n], y[n], sum = 0.0;

    // This directive tells the compiler to parallelize the loop
    #pragma omp parallel for reduction(+:sum)
    for (i = 0; i < n; i++) {
        x[i] = i * 1.0;
        y[i] = i * 2.0;
        sum += x[i] * y[i];
    }

    printf("Sum = %f\n", sum);
    return 0;
}
```

>gcc -fopenmp -o example example.c

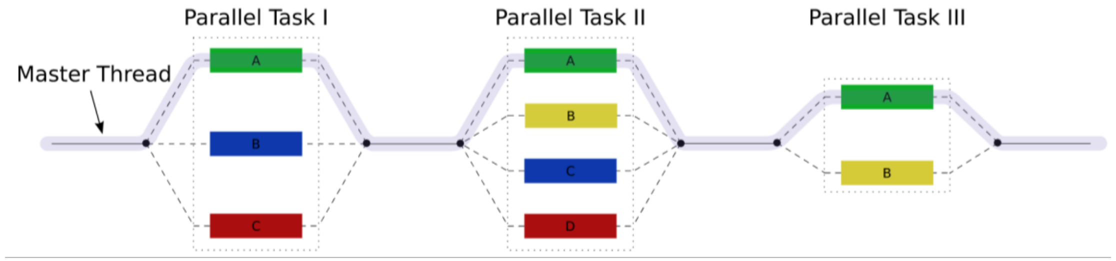
DATA PARALLELISM

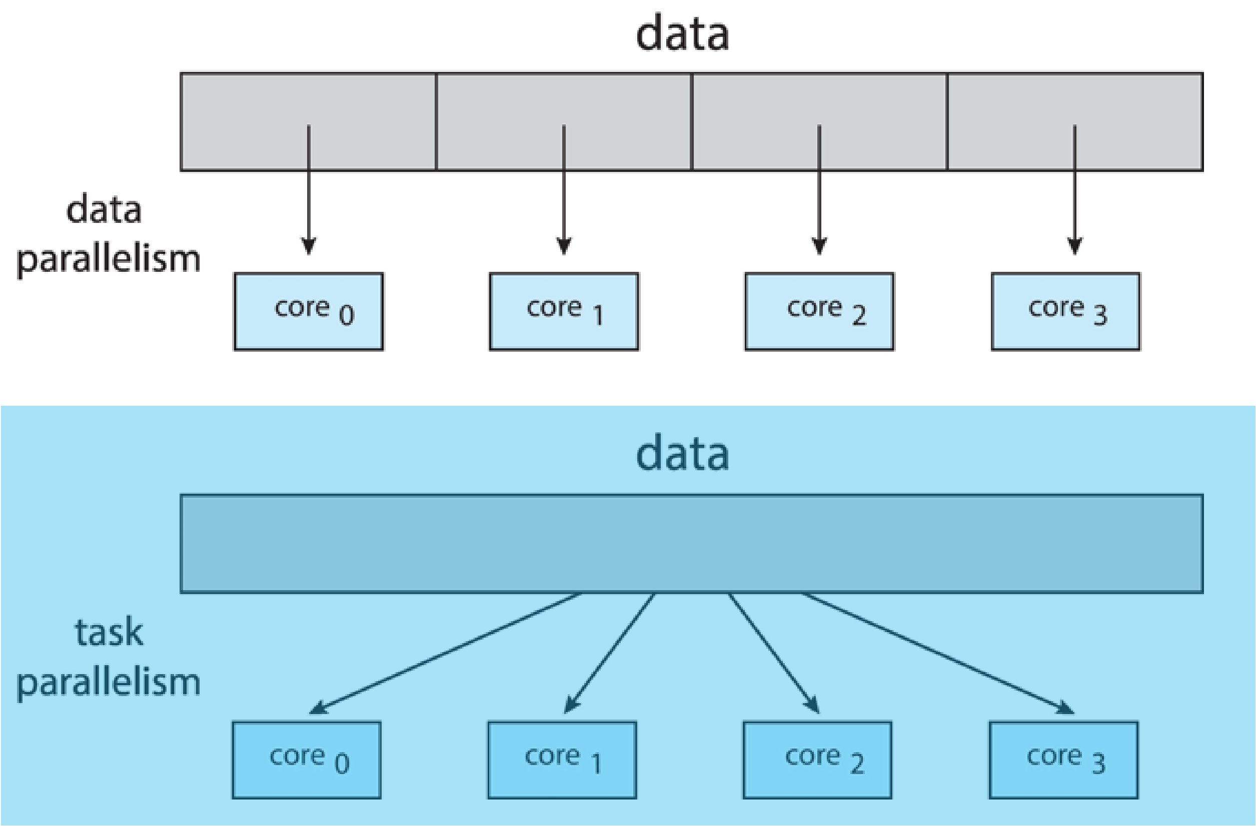




- When we're exploiting data parallelism, correct implementation of multithreading is simple.
- Each thread is working independently and there are no risks of overwrites and synchronization is not needed.

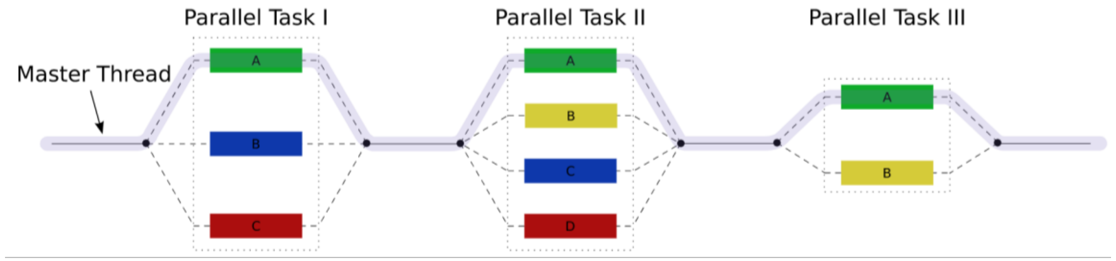
TASK PARALLELISM

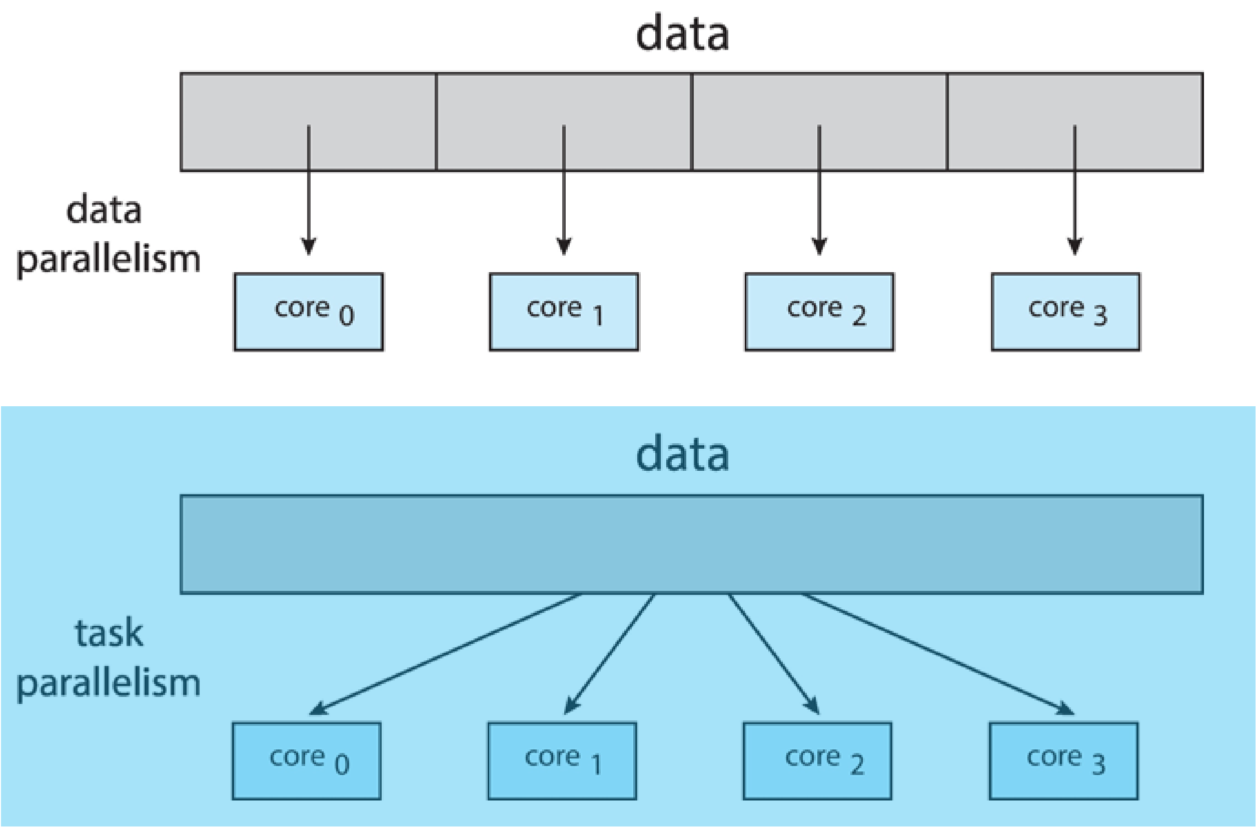




- This is not the case when we have 'task' parallelism when multiple threads are operating on the same data …

TASK PARALLELISM





- This is not the case when we have 'task' parallelism when multiple threads are operating on the same data …
- Threads could read expired data or overwrite fresh data that has not been processed yet …

WORKSHEET Q3

- Program Output?

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    printf("hello\n");
    fork();
    printf("bye\n");
    return 0;
}
```

WORKSHEET Q3

- Program Output?

(A)

```
hello
bye
hello
bye
bye
bye
```

(B)

```
hello
bye
bye
hello
bye
bye
```

(C)

```
hello
hello
bye
bye
bye
bye
```

```c
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    printf("hello\n");
    fork();
    printf("bye\n");
    return 0;
}
```

WORKSHEET Q3

- Program Output?

(A)

```
hello
bye
hello
bye
bye
bye
```

✓

(B)

```
hello
bye
bye
hello
bye
bye
```

✓

(C)

```
hello
hello
bye
bye
bye
bye
```

✓

Scheduler is unpredictable!

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    printf("hello\n");
    fork();
    printf("bye\n");
    return 0;
}
```

WORKSHEET Q3

- Program Output?

(A)	(B)	(C)
hello	hello	hello
bye	bye	hello
hello	bye	bye
bye	hello	bye
bye	bye	bye
bye	bye	bye
✓	✓	✓

```c
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    printf("hello\n");
    fork();
    printf("bye\n");
    return 0;
}
```

WORKSHEET Q3

- Program Output?

(A)	(B)	(C)
hello	hello	hello
bye	bye	hello
hello	bye	bye
bye	hello	bye
bye	bye	bye
bye	bye	bye
✓	✓	✓

- Print two 'hellos' and four 'byes'
- Start with 'hello', and end with two 'bye'

```c
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    printf("hello\n");
    fork();
    printf("bye\n");
    return 0;
}
```

CONCURRENCY

- A major advantage of multi-processing and multi threading is the ability to process data concurrently.
- One major issue:
  - **Scheduler is unpredictable!**
- Need to ensure that the processes/threads are **independent.**

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    printf("hello\n");
    fork();
    printf("bye\n");
    return 0;
}
```

hello	hello	hello
bye	bye	hello
hello	bye	bye
bye	hello	bye
bye	bye	bye
bye	bye	bye
✓	✓	✓