MONASH University

MONASH
INFORMATION
TECHNOLOGY

FIT2100 Semester 2 2020

Lecture 7: Threads and Concurrency

(Reading: Stallings, Chapter 4, 5)

- WEEK 8

Lecture 7 (Part A): Learning Outcomes

❑ Upon the completion of this lecture, you should be able to:

- Understand the distinction between process and thread
- Describe the basic design issues for threads
- Explain the difference between user-level threads and kernel-level threads
- Discuss thread management in Unix/Linux

Reading from Stallings, Chapter 4 (4.1, 4.2, 4.6)

WHAT DO WE UNDERSTAND ABOUT PROCESSES?

- A process 'owns' resources
  - Space in main memory, open files, I/O devices, etc.
  - An independent **process image**
  - One process is prevented from interfering with another process's resources and image as allocated by the OS
- Scheduling and execution
  - A process follows a 'path' of execution
  - Execution may be interleaved with other processes
  - Scheduled and dispatched by the OS.

The Concept of Threads

- The unit of dispatching for execution is referred to as:
  - Thread or
  - Lightweight process

- The unit of resource ownership is referred to as:
  - Process or task

e.g. Windows or Unix

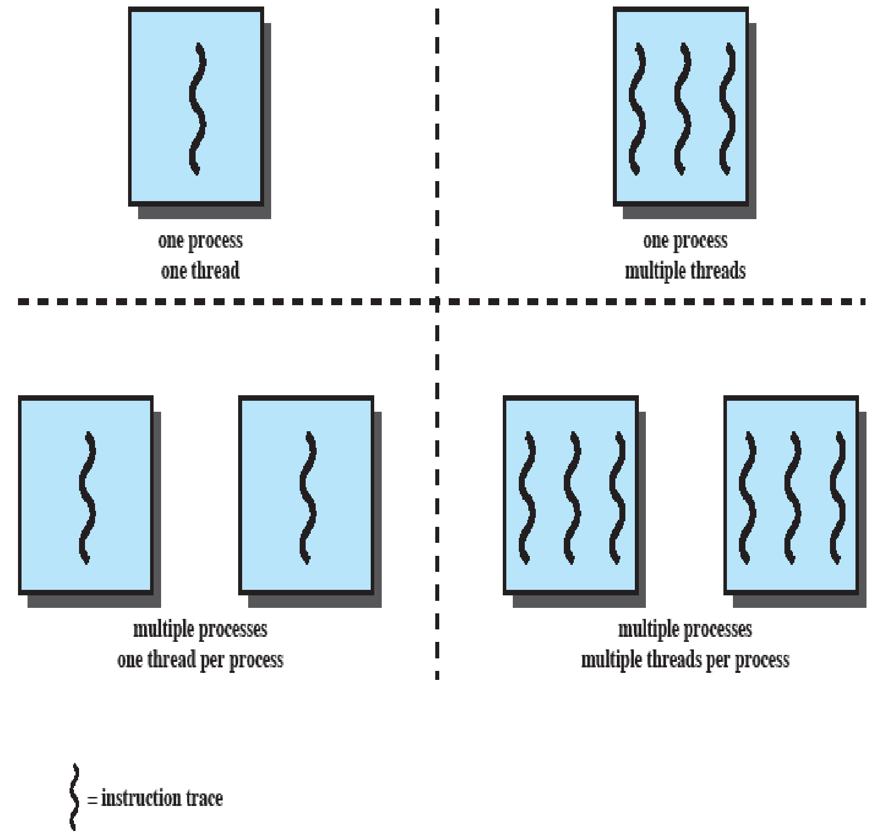
The entity that owns a resource is a process

The Concept of Threads

- The unit of dispatching for execution is referred to as:
  - Thread or
  - Lightweight process
- The unit of resource ownership is referred to as:
  - Process or task
- Multi-threading:
  - The ability of an OS to support multiple concurrent paths of execution within a single process
  - **1 process : multiple threads of execution.**

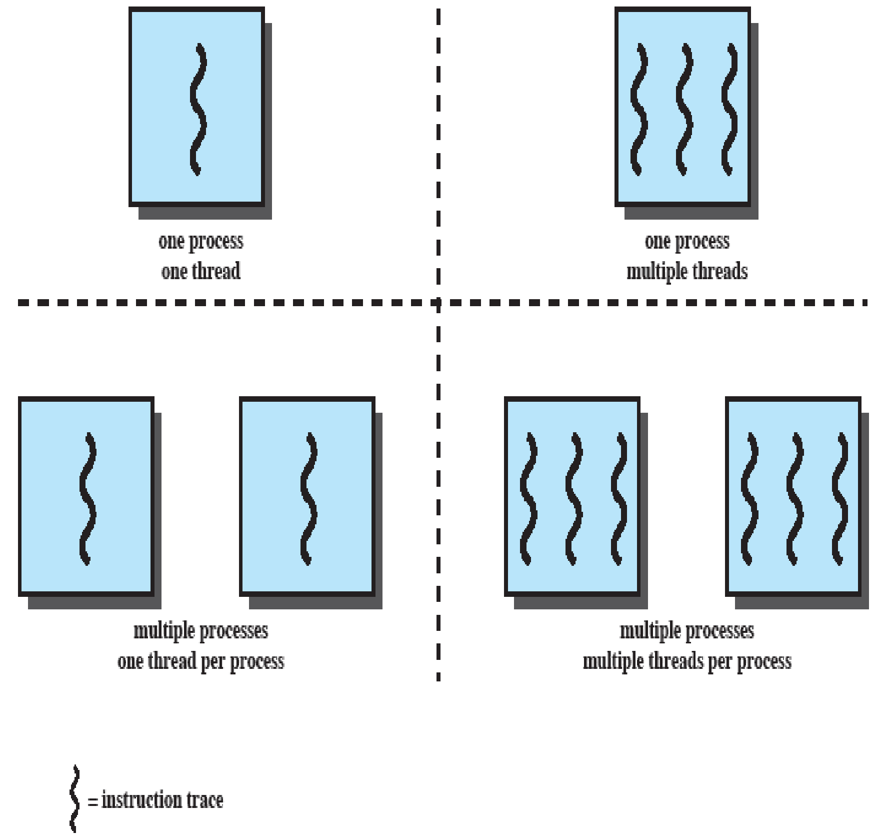
Single-Threaded Approaches

- Left side of figure:
  - A single thread of execution per process.
- The concept of a thread is not recognised — referred as a single-threaded approach.
- Example: MS-DOS, Windows 3.1



Multi-Threaded Approaches

- The right half of the figure depicts the multi-threaded approach.
- One process with multiple threads.
- Example: Windows, Linux.



The Concept of Processes (revisit)

- The unit of resource allocation and a unit of protection.
- A (virtual) address space that holds the process image.
- Protected access to:
  - Processor(s)
  - Other processes
  - Files
  - I/O resources

Interprocess communication

Threads within a Process

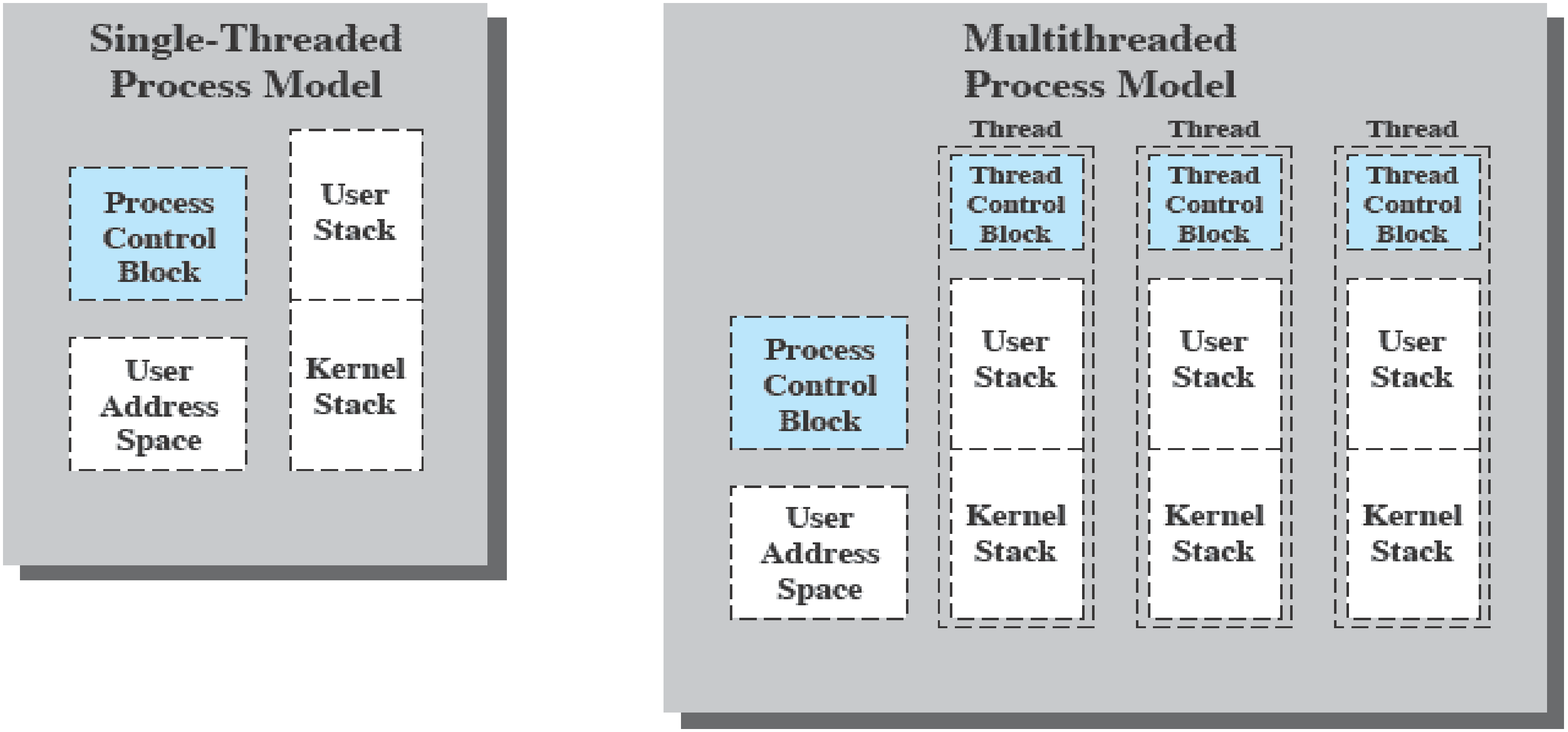
- Different part of a program may do different things and they can be executed concurrently to improve response time (or completion time).
  - Example: one thread may do a processor-bound task like rendering an image, while another thread responds to user interaction in the same program.
- If there is an interaction between different parts of the programs — concurrency control need to be applied.
- Example: accessing and modifying a common variable — mutual exclusion need to be satisfied.

Attributes of a Thread

Each thread has:

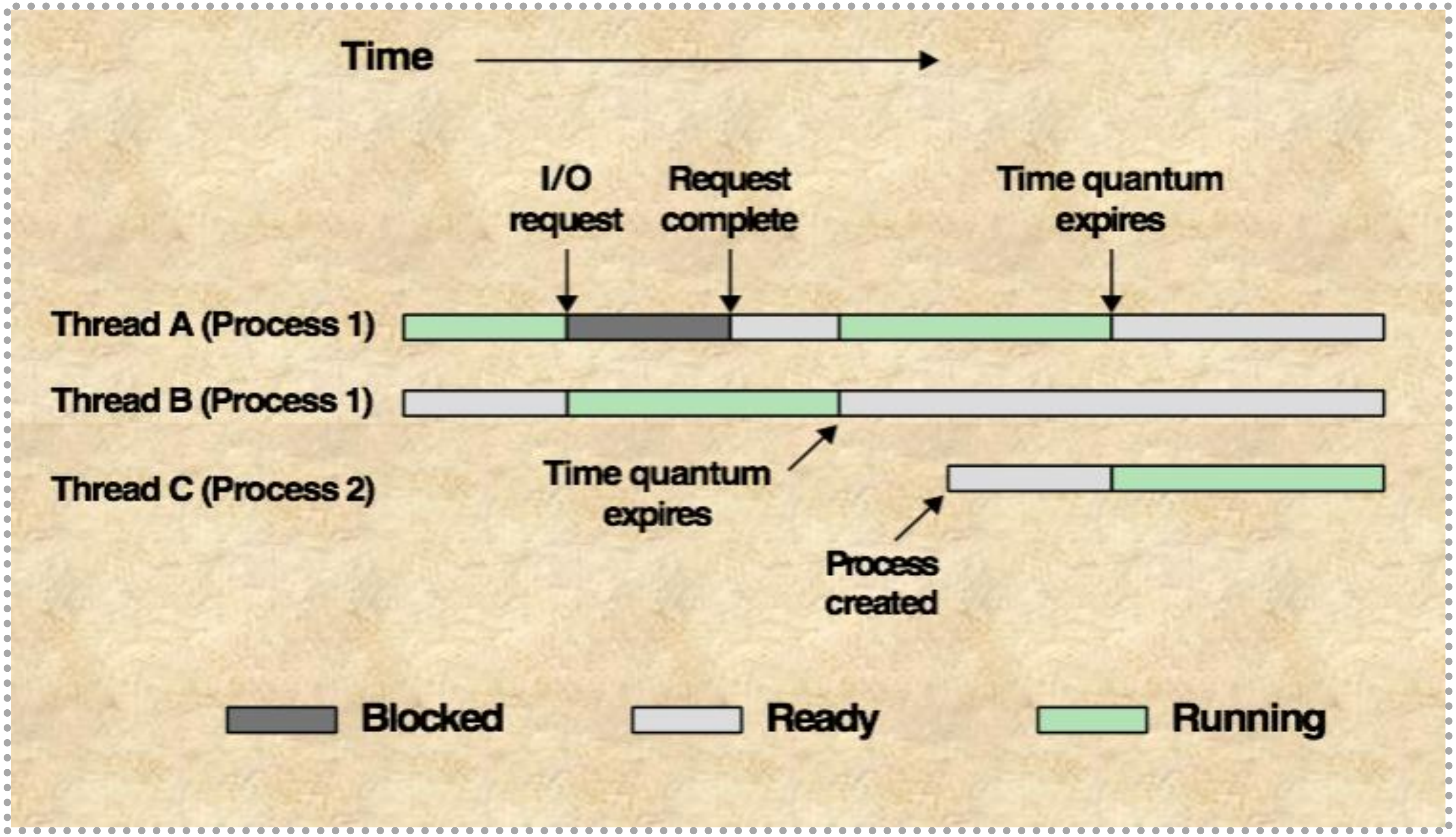
- an execution state (Running, Ready, etc.)
- saved thread context when not running
- an execution stack
- some per-thread static storage for local variables
- access to the memory and resources of its process (all threads of a process share this)

Threads vs. Processes



Single Threaded and Multithreaded Process Models

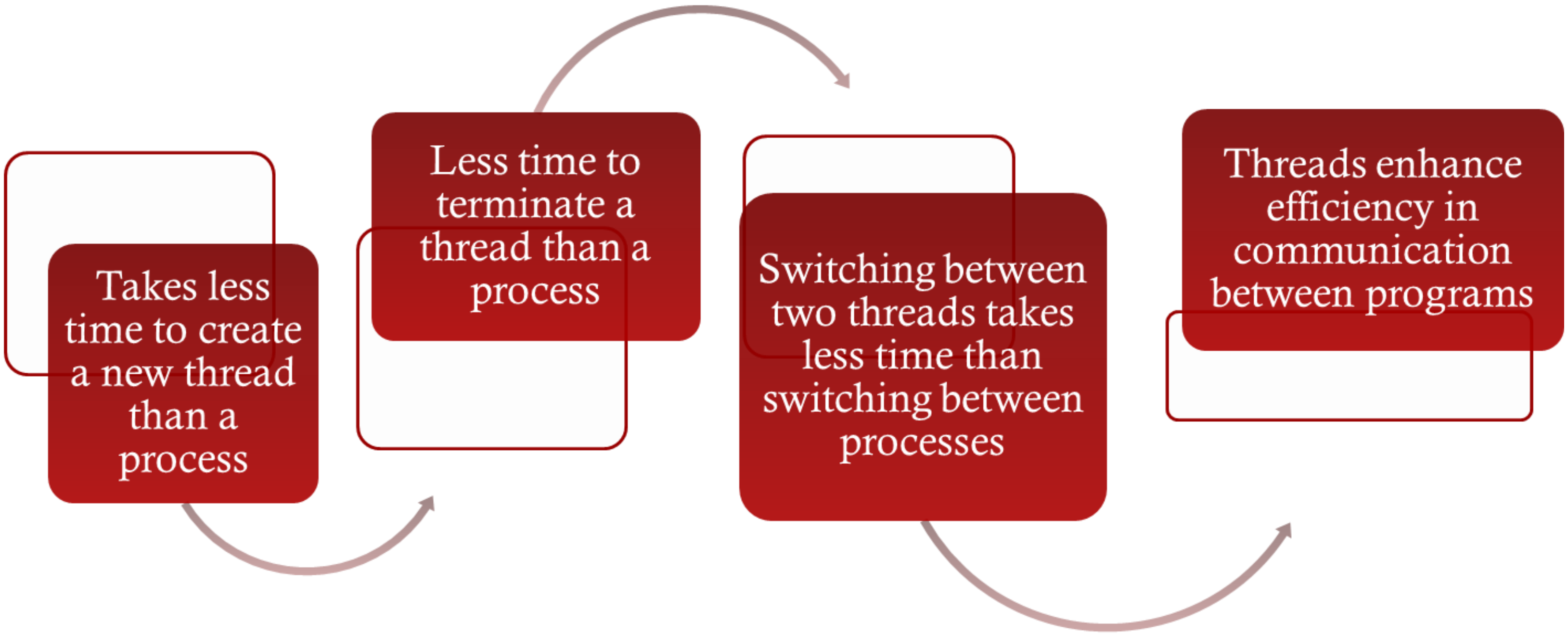
Thread States: Multi-threading on a Uniprocessor



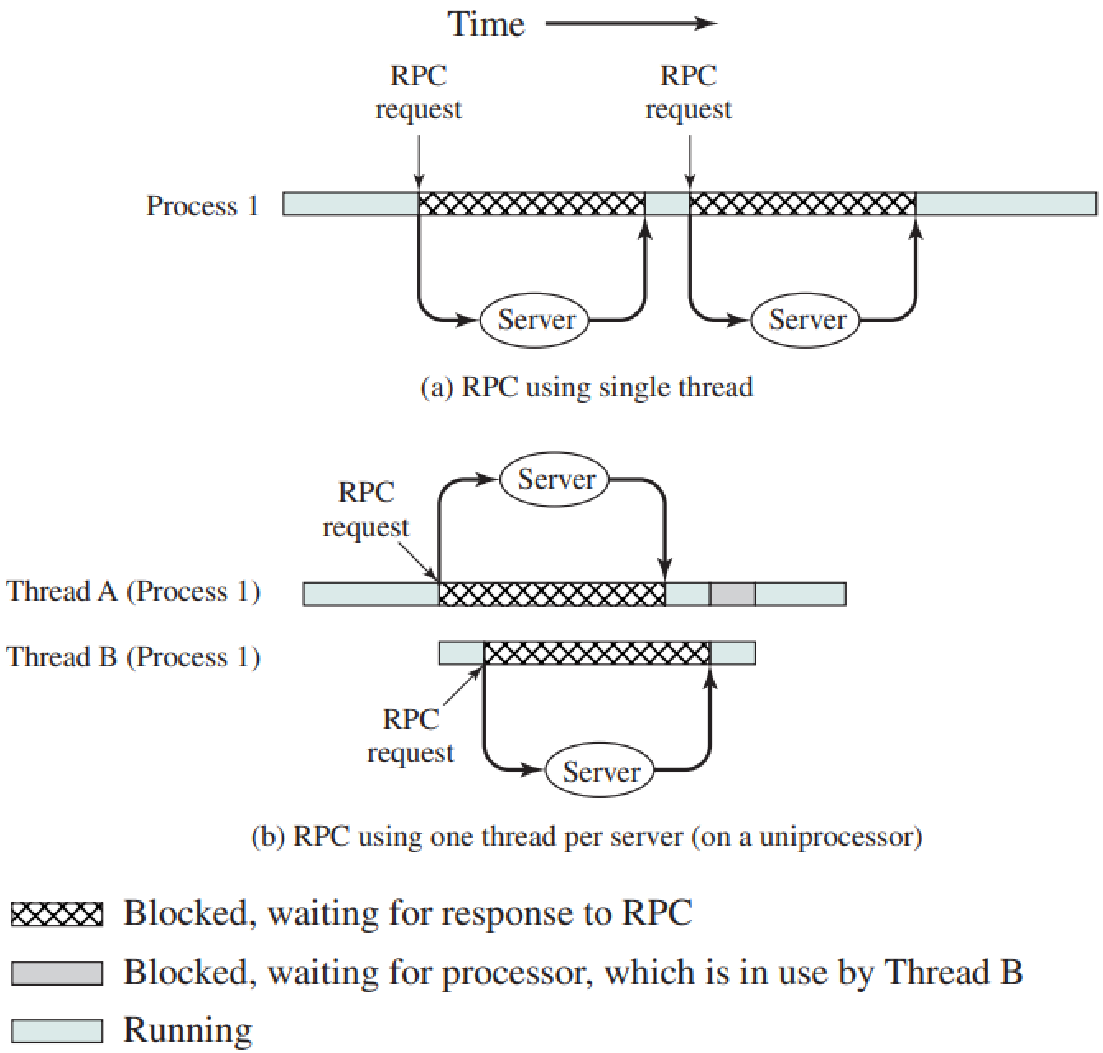
e.g. One thread may run while another thread is blocked.

Benefits of Threads

Threads == 'lightweight processes'



Benefits of Threads: Remote Procedure Call (RPC) Example



More on Threads

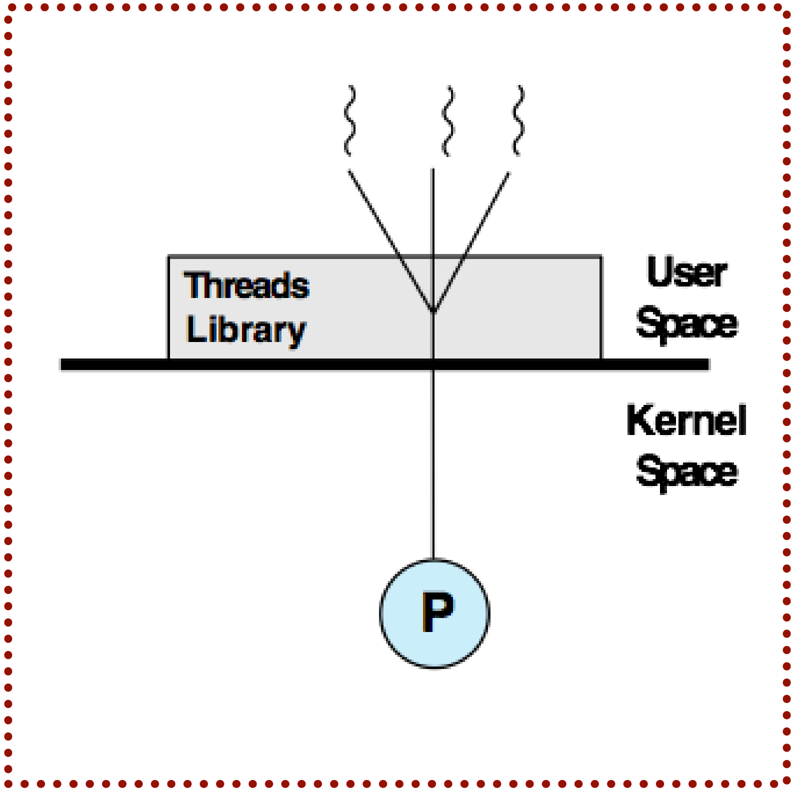
- For an OS that supports threads, scheduling and dispatching is done on a thread basis.
- Most of the state information dealing with execution is maintained in thread-level data structures:
  - Suspending a process involves suspending all threads of a process.
  - Termination of a process terminates all threads within the process.

All threads within a process share the same address space.

User-Level Thread (UTL) and Kernel-Level Thread (KLT)

User-Level Threads (ULTs)

- All thread management is done by the application.
- The kernel is not aware of the existence of threads.
- Any application can be programmed to be multi-threaded by using a threads library.
  - Even if OS does not support threads.



Pure user-level

ULTS: Advantages

Thread switching does not required kernel mode privileges

Scheduling can be application specific

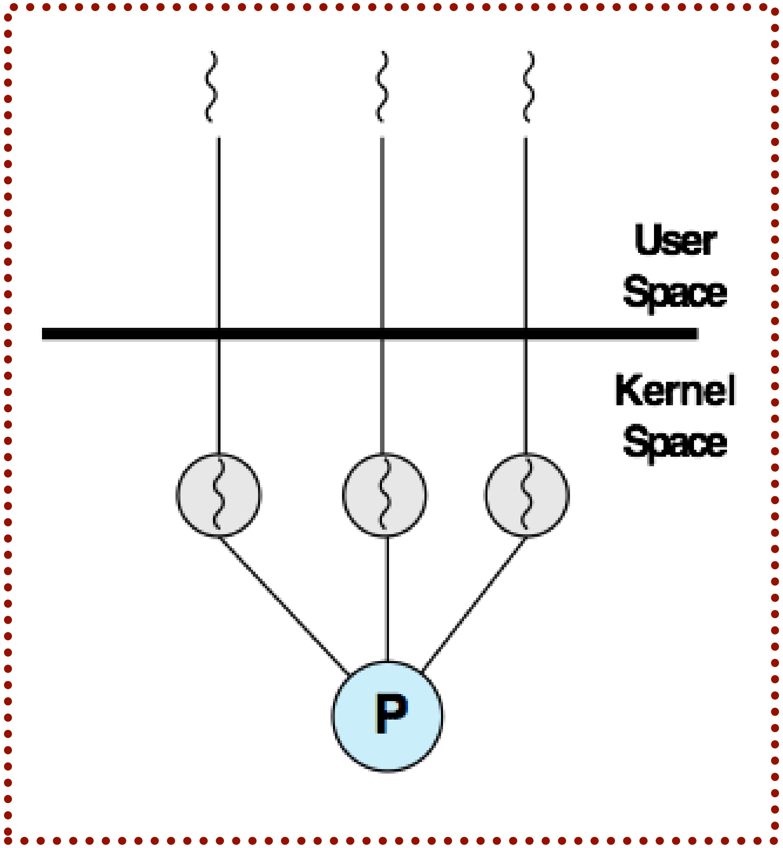
ULTs can run on any Operating System

ULTS: Disadvantages

- In a typical OS, many system calls are *blocking*.
- When a ULT executes a system call, not only is that thread gets blocked, but all of the threads within the process are also blocked.
- In a pure ULT strategy, a multi-threaded application cannot take the full advantage of multiprocessing.

Kernel-Level Threads (ULTs)

- ❑ Thread management is done by the kernel.
- ❑ No thread management is done by the application — through API to the kernel thread facility.
- ❑ Example: Windows, Linux



Pure kernel-level

KLTS: Advantages

- The kernel can simultaneously schedule multiple threads from the same process on **multiple processors**.
- If one thread in a process is blocked, the kernel can schedule another thread of the same process.
- The kernel routines can also be multi-threaded.

KLTS: Disadvantages

- The transfer of control from one thread to another thread within the same process requires a mode switch to the kernel.
  - Some overhead here.

How do Unix/Linux systems manage threads?

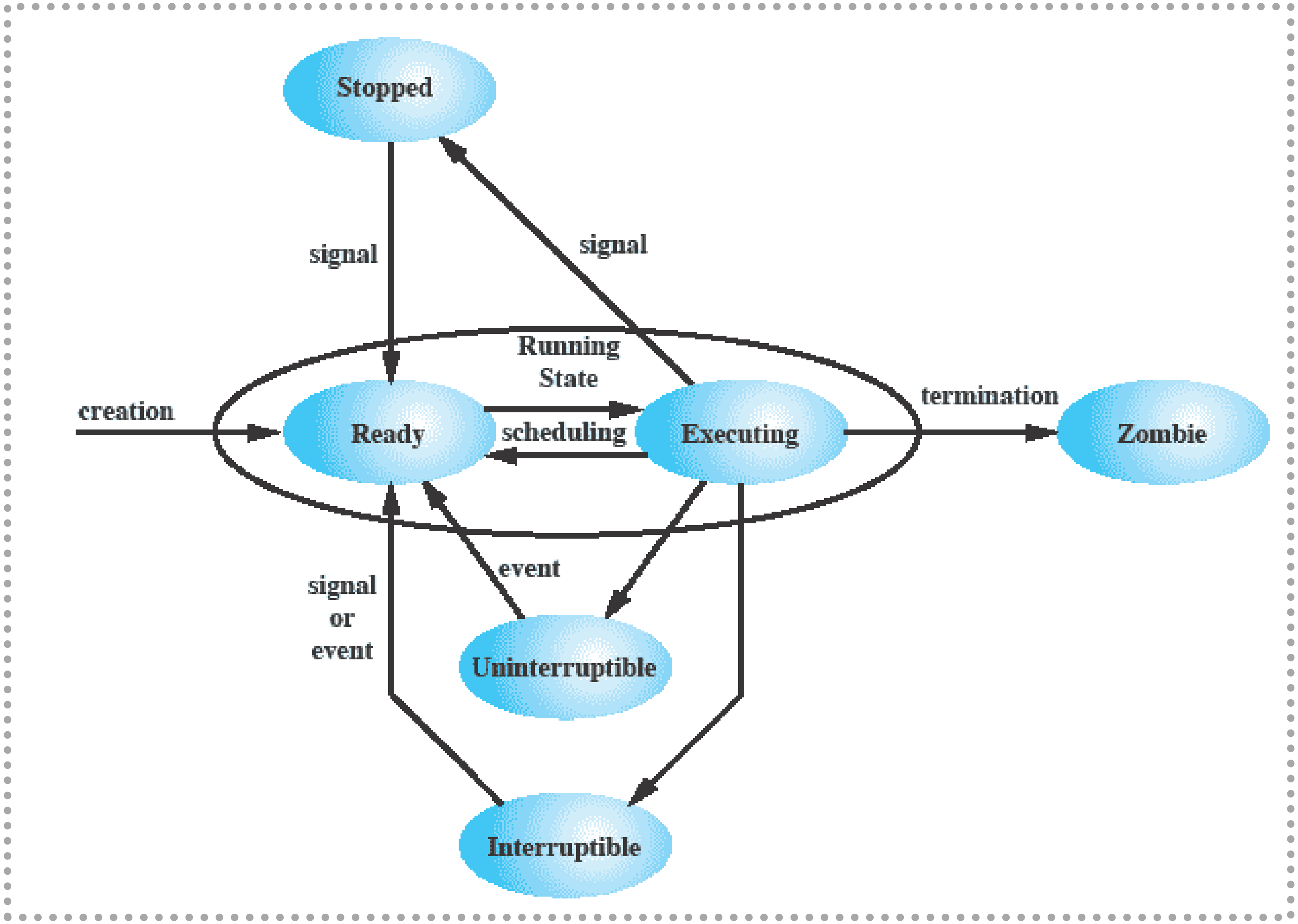
PTHREADS

- Threads in Linux are known as **pthreads**. Managed through a separate API.

- The **pthread** library must be included and linked into the program in order to use threads.
  - **`#include <pthread.h>`**
  - Add **`-lpthread`** to the end of the **gcc** command to **link** the program against the pthread library

- **`pthread_create()`** – spawn a new thread
- **`pthread_join()`** – wait for another thread to terminate

Linux: Process/Thread Model



Summary of Lecture 7 (Part A)

- The concept of process is related to resource ownership.
- The concept of thread is related to program execution.
- In multi-threaded system, multiple concurrent threads may be defined with a single process.
- Two types of threads: user-level and kernel level.

Reading from Stallings, Chapter 4: 4.1, 4.2 and 4.6

Lecture 7 (Part B): Learning Outcomes

- Upon the completion of this lecture, you should be able to:
  - Discuss the basic concepts related to concurrency
  - Understand the concept of race condition
  - Describe the mutual exclusion requirements

Concurrency: Terminology

atomic operation	A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.
critical section	A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
livelock	A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.
mutual exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
race condition	A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

What are the concurrency problems?

Concurrency: Problems

- Concurrent access to shared data may result in data inconsistency — **race condition**
- The problem exists in both multiprogramming on uni- and multi-processors
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- Difficult to locate programming errors — results are not deterministic and reproducible

Race Condition

- Occurs when multiple processes or threads read and write data items
- Final result depends on the order of execution
  - "Loser" of the race is the process that updates last and will determine the final value of the variable
- To prevent race conditions, concurrent processes must be synchronised

Concurrency Problem: Uniprocessor Multiprogramming

- Consider the following procedure:

```
void echo()
{
    char_in = getchar();
    char_out = char_in;
    putchar(char_out);
}
```

- Read a character from the keyboard and store in `char_in`.
- Transfer to `char_out` before being sent for display.
- Consider two different applications — P1 and P2 — make a call to this procedure.

Concurrency Problem: Uniprocessor Multiprogramming

- Consider the following procedure:

```
void echo()
{
    char_in = getchar();
    char_out = char_in;
    putchar(char_out);
}
```

- P1 invokes **`echo()`** and is interrupted immediately after **`getchar`** returns its value and stores it in **`char_in`** (e.g. **`x`**).
- P2 is activated and invokes **`echo()`** and read a char (e.g. **`y`**) and runs to completion of the procedure.
- When P1 resumes the value of **`x`** has been overwritten in **`char_in`** by process P2 and therefore its value **`x`** is lost.

Concurrency Problem: Uniprocessor Multiprogramming

- Process P1:

```
char_in = getchar();



char_out = char_in;
putchar(char_out);
```

- Process P2:

```

char_in = getchar();
char_out = char_in;
putchar(char_out);
```

TIME

Concurrency Problem: Uniprocessor Multiprogramming

- Consider the following procedure:

```
void echo()
{
    char_in = getchar();
    char_out = char_in;
    putchar(char-out);
}
```

- Assume only one process at a time to invoke and be in the `echo` procedure.
- P1 invokes `echo()` and is interrupted immediately after `getchar` returns its value and stores it in `char_in` (`x`).
- P2 is activated and invokes `echo()`.
- But since P1 is still inside the procedure, and currently suspended — P2 is blocked from entering the procedure.

Concurrency Problem: Summary

- P1 invokes the **`echo`** procedure and is interrupted immediately after **`getchar`** returns its value and stores it in **`char_in`** (e.g. **`x`**).
- P2 is activated and invokes **`echo`** procedure and since the **`echo`** procedure is used by process P1, P2 is blocked from further execution.
- At some later time, P1 is resumed and completes the execution of **`echo`** and the proper input character will be displayed.
- When P1 exits **`echo`**, this removes the block on P2.
- When P2 is later resumed, the **`echo`** procedure is successfully invoked.

How about concurrency problems with multiprocessors?

Concurrency Problems: Multiprocessor Multiprogramming

- Same problem arises even when the processes — P1 and P2 — runs on different processors accessing unprotected shared variables.

- The solution outlined in the previous slides can work here.

- Protecting and controlling access to shared resources are critical.

Question: Race Condition

- Assume P1 and P2 share two variables **`a`** and **`b`** with initial values of **`a`** = 1 and **`b`** = 2
- P1 executes the statement: **`a`** = **`a`** + **`b`**
- P2 executes the statement: **`b`** = **`a`** + **`b`**
- What values are **`a`** and **`b`** if P1 executes before P2?
- What values are **`a`** and **`b`** if P2 executes before P1?

What are the control problems with concurrent processes?

Control Problems

- Concurrent processes come into conflict when they are competing for the same system resource — I/O devices, memory, processor time, etc.

In the case of competing processes three control problems must be faced:

- **mutual exclusion**
- **deadlock**
- **starvation**

Mutual Exclusion

- Suppose `n` processes all competing to use some shared data.

- Each process has a code segment — **critical section** — where the shared data is accessed or manipulated.

- Ensure that when one process is executing in its critical section, no other process is allowed in its critical section.

Mutual Exclusion: Example

```
/* PROCESS 1 */

void P1
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

```
/* PROCESS 2 */

void P2
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

• • •

```
/* PROCESS n */

void Pn
{
  while (true) {
    /* preceding code */;
    entercritical (Ra);
    /* critical section */;
    exitcritical (Ra);
    /* following code */;
  }
}
```

To enforce mutual exclusion, two function are provided: entercritical and exitcritical with the resource as the argument.

Multiple Shared Data Resources

- The same problem exists even when processes access more than one shared resource.
- Processes must cooperate to ensure the shared data are properly managed.
- Control mechanisms are needed to ensure the integrity of the shared data.

Multiple Shared Data Resources: Example

- **P1**

```
a = a + 1 ;
b = b + 1 ;
```

- **P2**

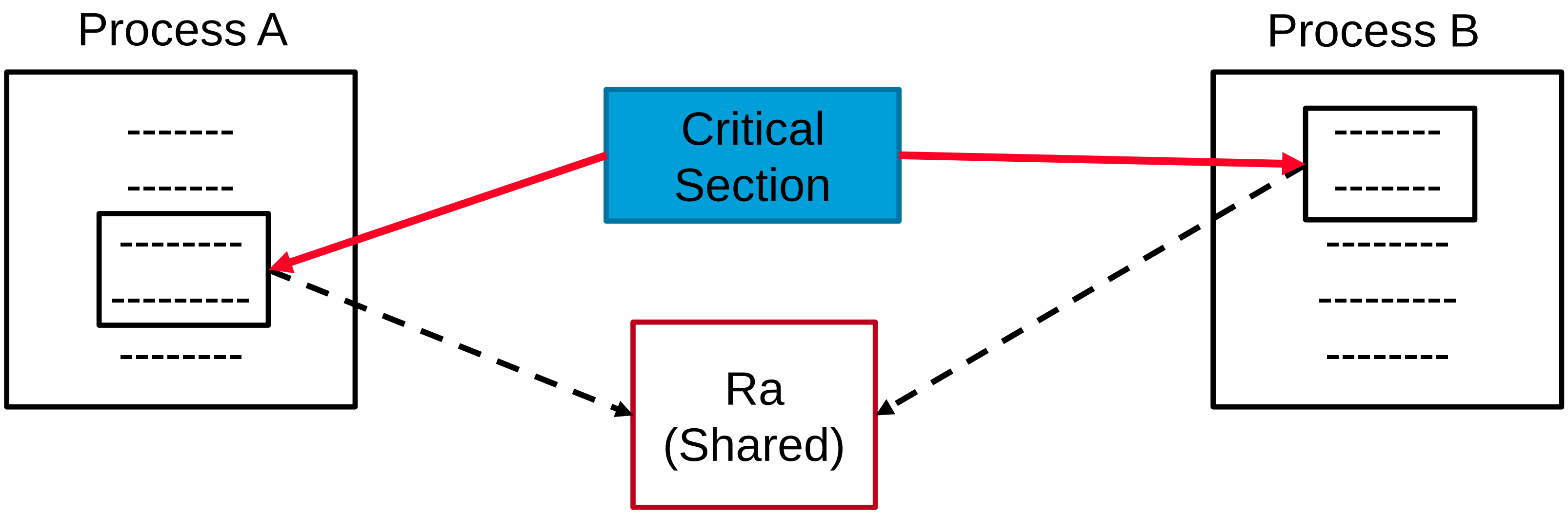
```
b = 2 * b ;
a = 2 * a ;
```

- Assuming that `a = b` at the beginning, and consider the following concurrent execution sequence:

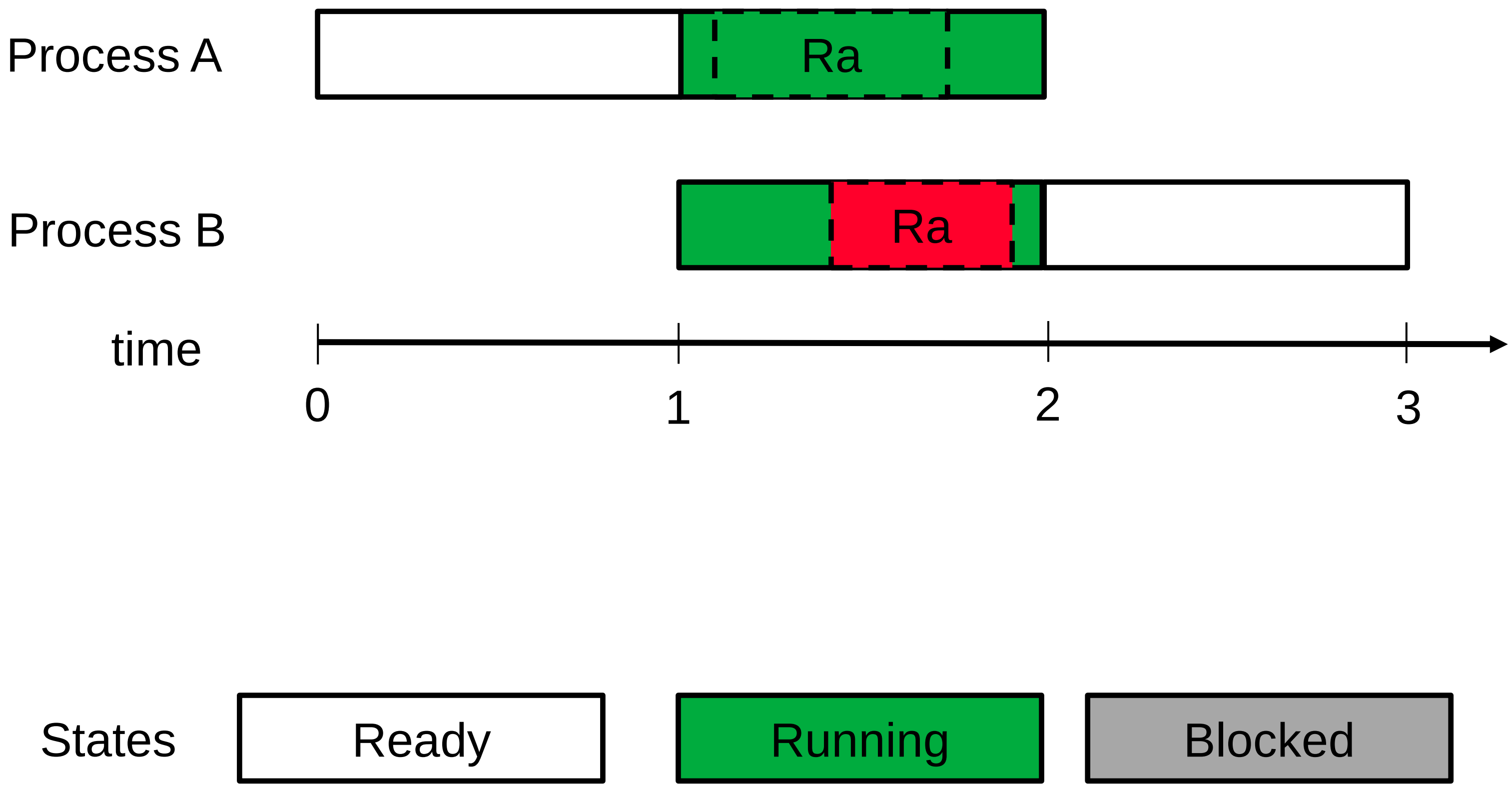
```
a = a + 1 ;   /* {P1} */
b = 2 * b ;   /* {P2} */
b = b + 1 ;   /* {P1} */
a = 2 * a ;   /* {P2} */
```

- At the end of this execution, the condition `a = b` ***no longer holds!***

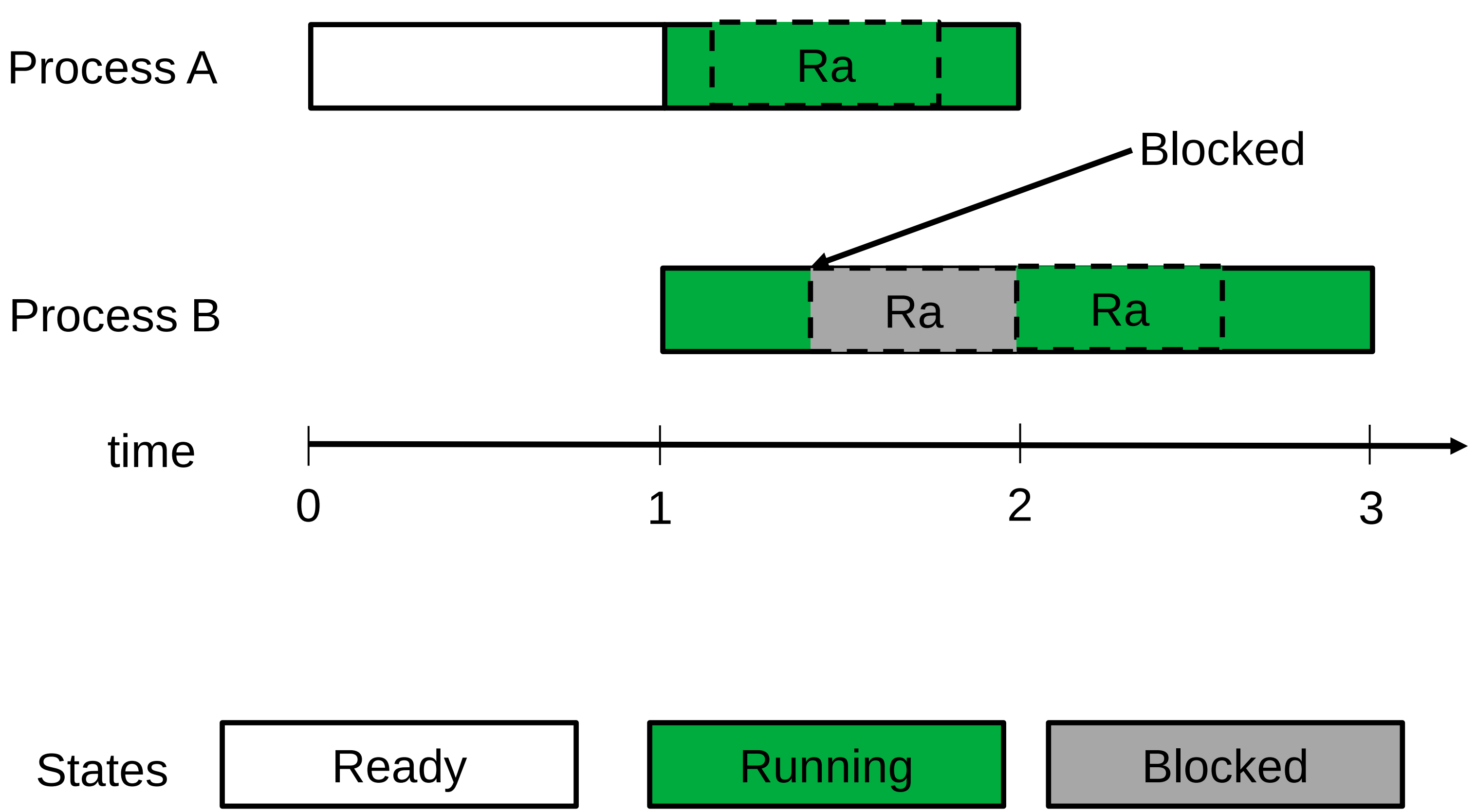
Race Condition (revisit)



Race Condition (revisit)



Solution: Mutual Exclusion



Additional Control Problems

- **Deadlock**: two or more processes are waiting indefinitely for the other processes to release the system resources.

- **Starvation**: indefinite blocking of a process.

Mutual Exclusion: Requirements

- Mutual Exclusion must be enforced.
- A process that halts must do so without interfering with other processes.
- A process must not be denied access to a critical section when there is no other process is using the shared resources (being manipulated by the critical section code).
- No assumptions are made about relative process speeds or the number of processors.
- A process remains inside its critical section for a finite time only.
- **No deadlock or starvation**.

Summary of Lecture 7 (Part B)

- Concurrency is the fundamental concern in supporting multiprogramming, multiprocessing, and distributed processing.
- Mutual exclusion is the condition where there is a set of concurrent processes — only one of which is able to access a given resource or perform a given function at any time.

Reading from Stallings, Chapter 5: 5.1

Next week: Concurrency mechanisms, deadlocks and starvation.