MONASH University

Information Technology

FIT2100 Operating Systems

THREADS AND CONCURRENCY

Week 7
Semester 2 2024
(Reading: Tanenbaum, Chapter 2)

Dr Charith Jayasekara
Faculty of Information Technology

OUTLINE

- The Thread Model
- Thread Usage
- Thread Implementations
- POSIX Threads
- Concurrency Problems
- Race Condition
- Mutual Exclusion

LEARNING OUTCOMES

- Upon the completion of Week 7, you should be able to:
  - ◻ Understand the distinction between process and thread
  - ◻ Describe the basic thread models
  - ◻ Explain the difference between user-level threads and kernel-level threads
  - ◻ Discuss thread management in Unix/Linux
  - ◻ Discuss the problems related to concurrency
  - ◻ Understand the concept of a race condition
  - ◻ Understand the concept of critical regions
  - ◻ Describe the mutual exclusion requirements

THE THREAD MODEL

THREADS

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**

THREADS

WHAT IS A THREAD?

- 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.**

THREADS

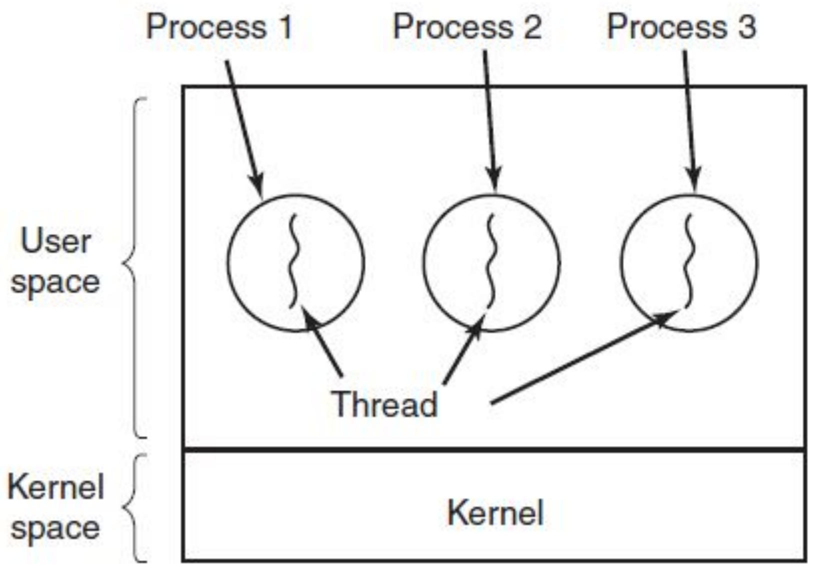
THREADS WITHIN A PROCESS

- 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.

THE THREAD MODEL

SINGLE-THREADED APPROACH

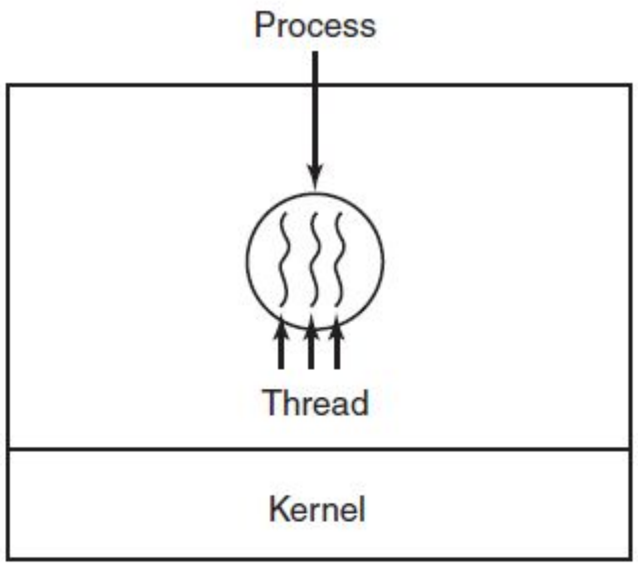
- A single thread of execution per process.
- The concept of a thread is not recognised — referred as a single-threaded approach.
- Example: MS-DOS, Windows 3.1



THE THREAD MODEL

MULTI-THREADED APPROACH

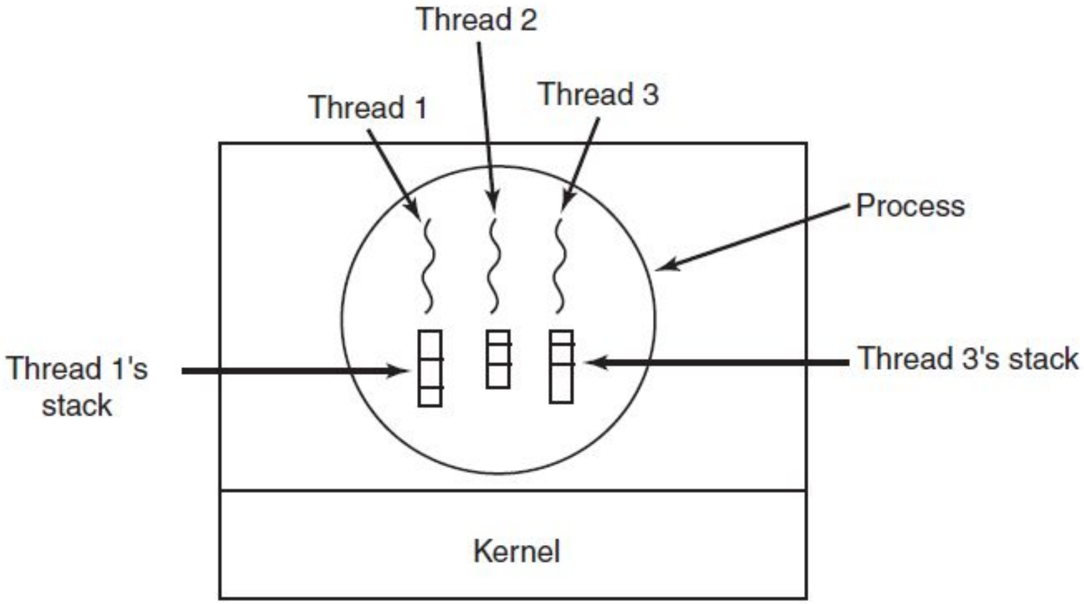
- One process with multiple threads.
- Example: Windows, Linux.



THE THREAD MODEL

PER-PROCESS vs PER-THREAD ITEMS

Per process items	Per thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	



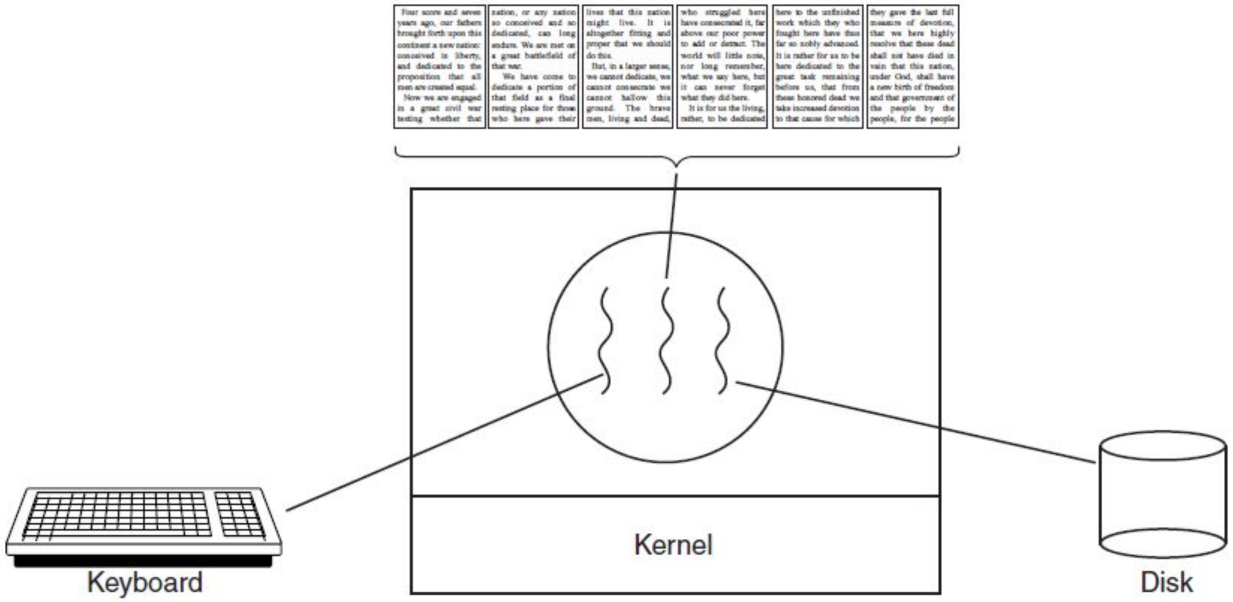
Each thread has its own stack.

- The first column lists some items *shared* by all threads in a process.
- The second one lists some items *private* to each thread.

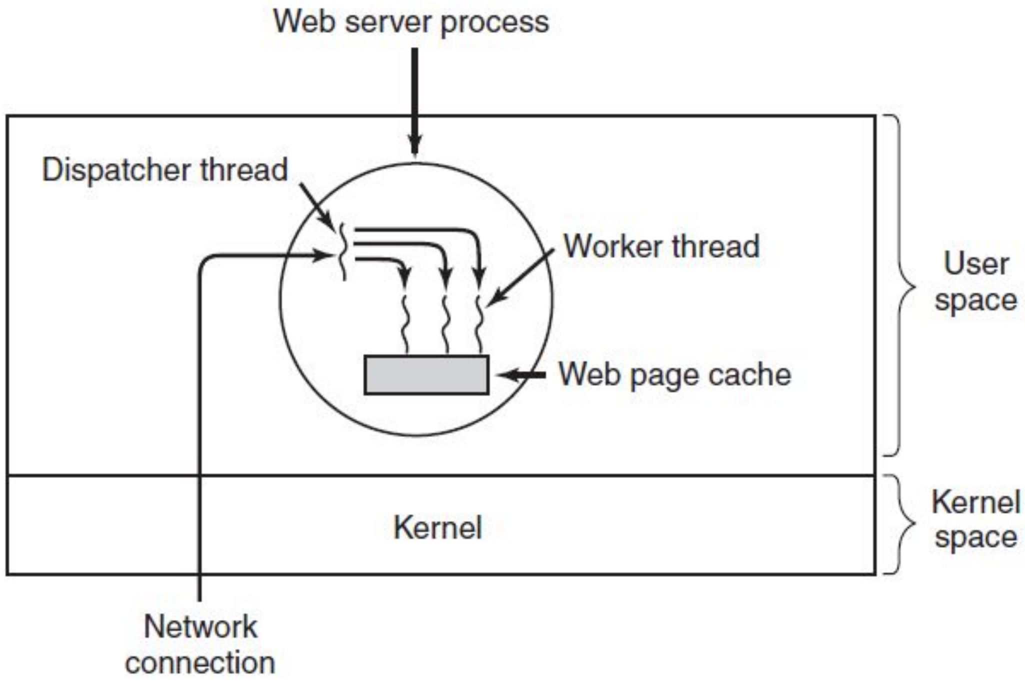
THREAD USAGE

THREAD USAGE

EXAMPLES OF MULTI-THREADED APPLICATIONS



A word processor with three threads.



A multithreaded Web server.

THREAD USAGE

MULTI-THREADED WEB SERVER IMPLEMENTATION

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

(a)

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

(b)

A rough outline of the code for a multi-threaded web server
(a) Dispatcher thread. (b) Worker thread.

THREAD USAGE

BENEFITS OF USING THREADS

- **Speed**: Takes less time to create a new thread than a process.
- **Speed**: Takes less time to terminate a thread than a process.
- **Speed**: Switching between two threads takes less time than switching between processes.
- **Speed**: Communication between threads within the same process is faster than communication between different processes.

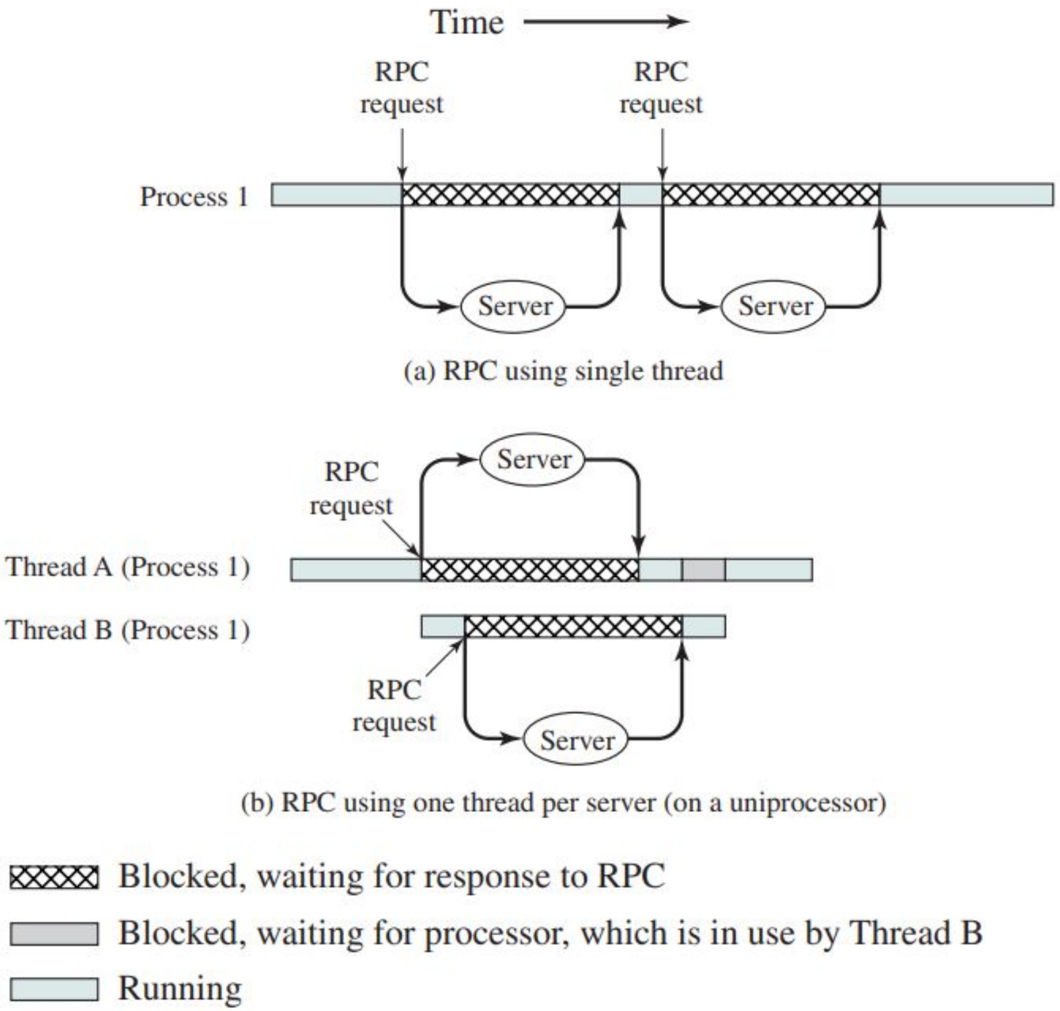
DRAWBACKS OF USING THREADS

- **Reliability**: A bug in one thread can disable multiple threads, or all threads.
- **Security**: If one thread is compromised, usually all threads are compromised.
- **Consistency**: Concurrency problems may arise with shared data structures
- **Portability**: Support for multithreading depends on CPU type, and performance gains may not be portable across different CPUs even of the same nominal architecture

THREAD USAGE

BENEFITS OF USING THREADS: REMOTE PROCEDURE CALL (RPC) EXAMPLE

- For a single-threaded approach, 2 RPC requests are made one after another by Process 1 (Fig. a).
- Process 1 is completely blocked while waiting for a response to arrive from the server (Fig. a).
- For a multi-threaded approach, Thread A sends the first RPC request and gets blocked (Fig. b).
- Thread B gets some CPU time and sends the second RPC request (Fig. b).
- The process completes quickly as the second RPC response arrives shortly after the first RPC response. (Fig. b).
- **Assumption:** Server is also multi-threaded. OR Different servers handled the RPC requests.



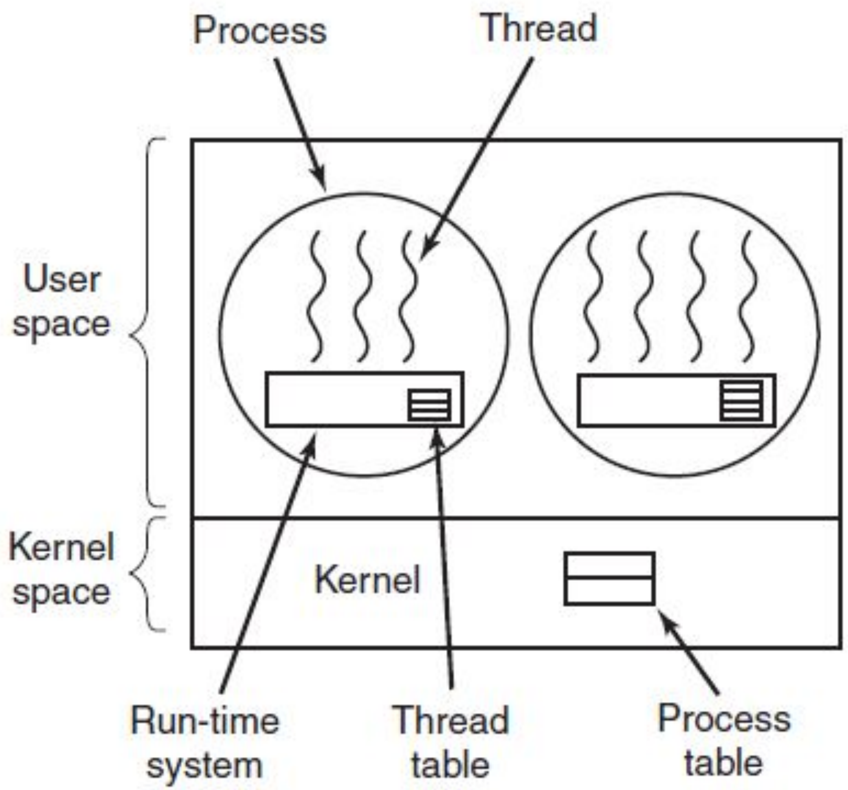
MONASH University

THREAD IMPLEMENTATIONS

THREAD IMPLEMENTATIONS

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.



THREAD IMPLEMENTATIONS

USER LEVEL THREADS (ULTs)

- **Advantages:**

❑ Thread switching does not require kernel mode privileges.

❑ Scheduling can be application specific.

❑ ULTs can run on any Operating Systems.

- **Disadvantages:**

❑ In a typical OS, many system calls are *blocking*.

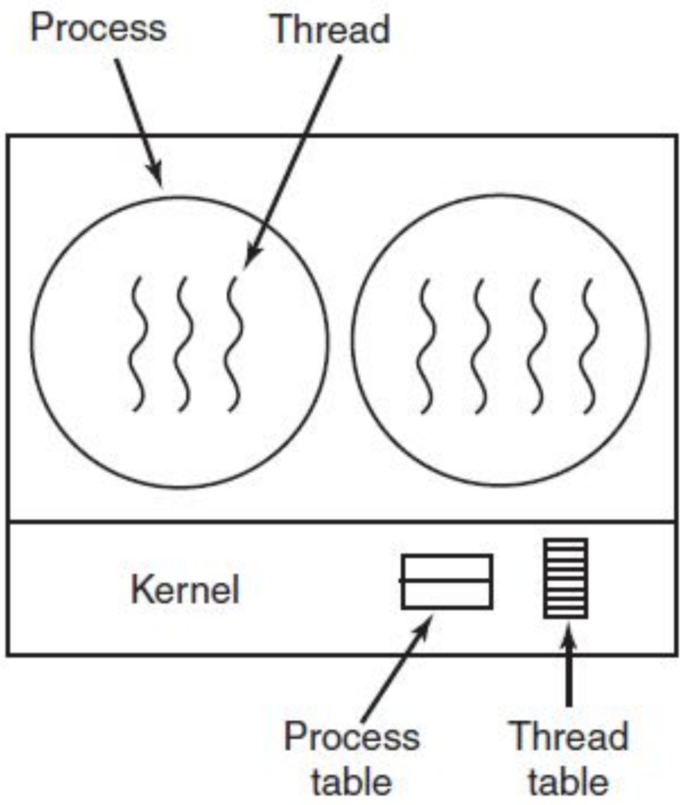
❑ When a ULT executes a system call, not only is that thread gets blocked, but all of the threads within the process are also blocked.

❑ In a pure ULT strategy, a multi-threaded application cannot take the full advantage of multiprocessing.

THREAD IMPLEMENTATIONS

KERNEL LEVEL THREADS (KLTs)

- 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



THREAD IMPLEMENTATIONS

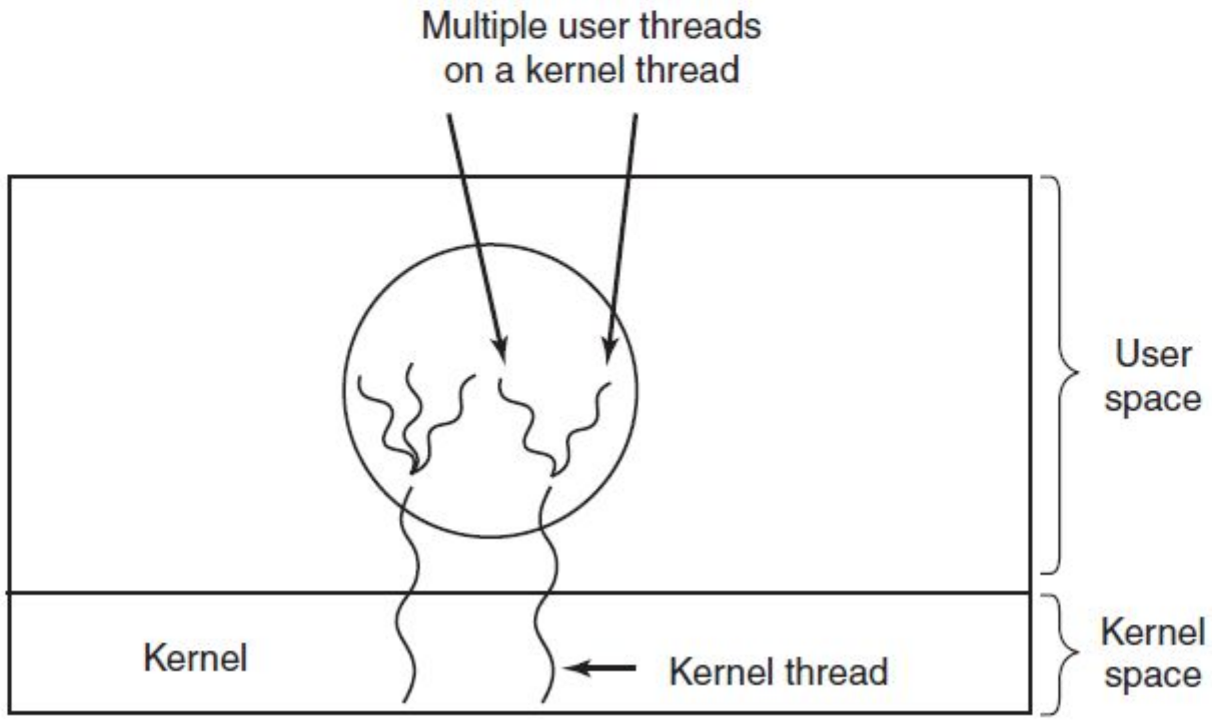
KERNEL LEVEL THREADS (KLTs)

- **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.

- **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.

THREAD IMPLEMENTATIONS

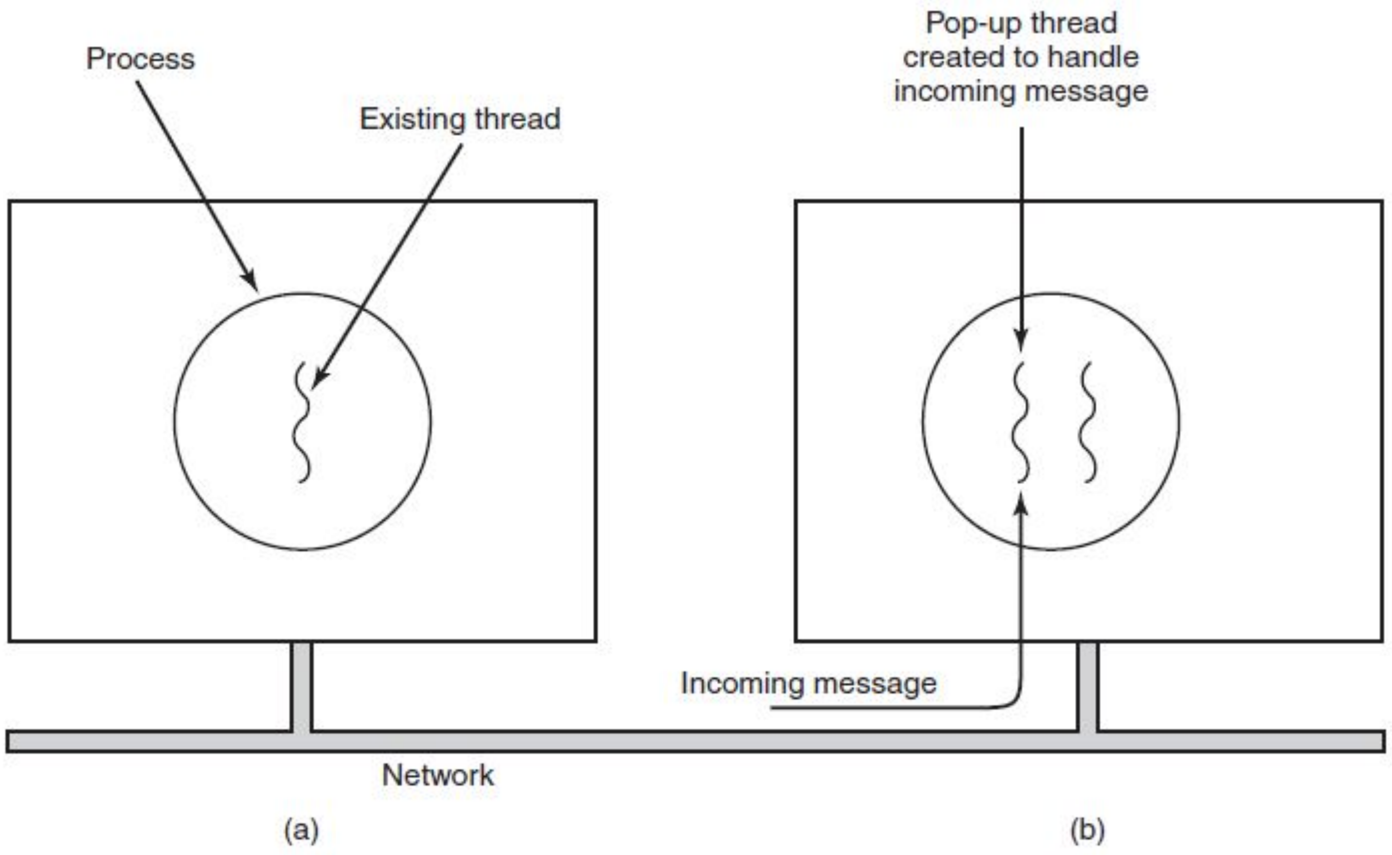
HYBRID IMPLEMENTATIONS



Multiplexing user-level threads
onto kernel-level threads.

THREAD IMPLEMENTATIONS

POP UP THREADS



Creation of a new thread when a message arrives:
(a) Before the message arrives. (b) After the message arrives.

MONASH University

POSIX THREADS (PTHREADS)

POSIX THREADS

PTHREADS LIBRARY

- 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

POSIX THREADS

PTHREADS EXAMPLE PROGRAM

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

#define NUMBER_OF_THREADS     10

void *print_hello_world(void *tid)
{
        /* This function prints the thread's identifier and then exits. */
        printf("Hello World. Greetings from thread %d\n", tid);
        pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
        /* The main program creates 10 threads and then exits. */
        pthread_t threads[NUMBER_OF_THREADS];
        int status, i;

        for(i=0; i < NUMBER_OF_THREADS; i++) {
                printf("Main here. Creating thread %d\n", i);
                status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);
```

MONASH University

POSIX THREADS

PTHREADS EXAMPLE PROGRAM

```
    int status, i;

    for(i=0; i < NUMBER_OF_THREADS; i++) {
        printf("Main here. Creating thread %d\n", i);
        status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

        if (status != 0) {
            printf("Oops. pthread_create returned error code %d\n", status);
            exit(-1);
        }
    }
    exit(NULL);
}
```

CONCURRENCY PROBLEMS

CONCURRENCY PROBLEMS

WHY CONCURRENCY PROBLEMS OCCUR

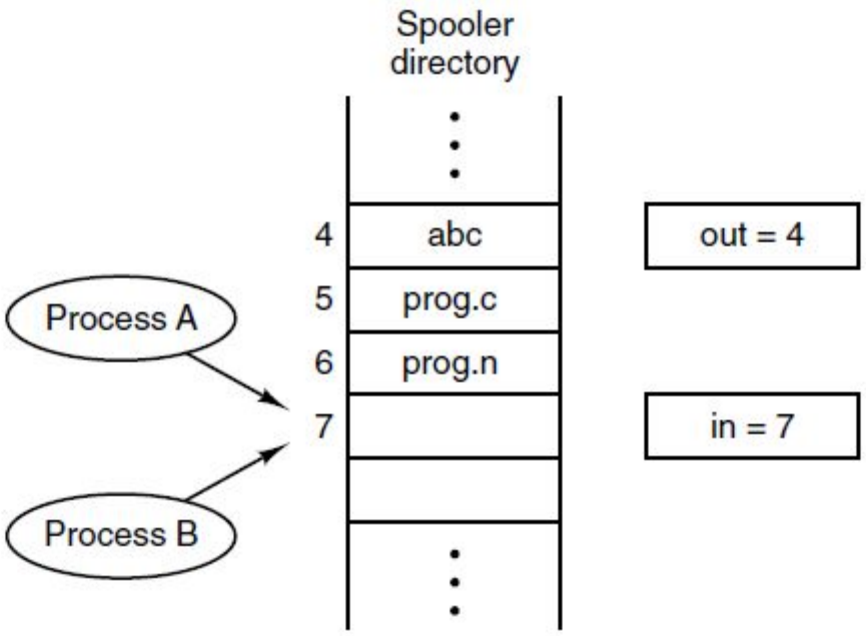
- ❑ Processes that are working together may share some common storage or memory that each one can read and write to communicate between themselves.
- ❑ Processes need to access a shared resource in the system.
- ❑ Concurrent access to shared data may result in data inconsistency.
- ❑ Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes.
- ❑ Difficult to locate programming errors — results are non-deterministic and not reproducible.
- ❑ The problem exists in both multiprogramming on uni-processor and multi-processors.

RACE CONDITIONS

RACE CONDITIONS

WHAT IS RACE CONDITION

- **A Race Condition** is a situation where two or more processes are reading or writing some shared data and the final result depends on who runs when.
- 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 or threads must be synchronised.



Two processes want to access shared memory at the same time.

MONASH University

RACE CONDITION

A RACE CONDITION IN 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 — Process A and Process B — make a call to this procedure.

RACE CONDITION

RACE CONDITION IN UNIPROCESSOR MULTIPROGRAMMING

- Consider the following procedure:

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

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

RACE CONDITION

RACE CONDITION IN UNIPROCESSOR MULTIPROGRAMMING

- Process A:

```
char_in = getchar();



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

- Process B:

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

TIME

MUTUAL EXCLUSION

MUTUAL EXCLUSION

WHAT IS MUTUAL EXCLUSION (MUTEX)

- Suppose `n` processes all **competing** to use some shared data.
- Each process has a code segment — **critical region** — where the shared data is accessed or manipulated.
- **Mutual exclusion** refers to the requirement of ensuring that no two concurrent processes are in their critical region at the same time.
- Mutual Exclusion is a basic requirement in concurrency control, to prevent **race conditions.**
- Ensure that when one process is executing in its critical region, **no other process is allowed in its critical region.**

MUTUAL EXCLUSION

REQUIREMENTS TO AVOID RACE CONDITIONS

- No two processes may be simultaneously inside their critical regions.
- No assumptions may be made about speeds or the number of CPUs.
- No process running outside its critical region may block other processes.
- No process should have to wait forever to enter its critical region.

MUTUAL EXCLUSION

PREVENTING A RACE CONDITION BY USING MUTUAL EXCLUSION

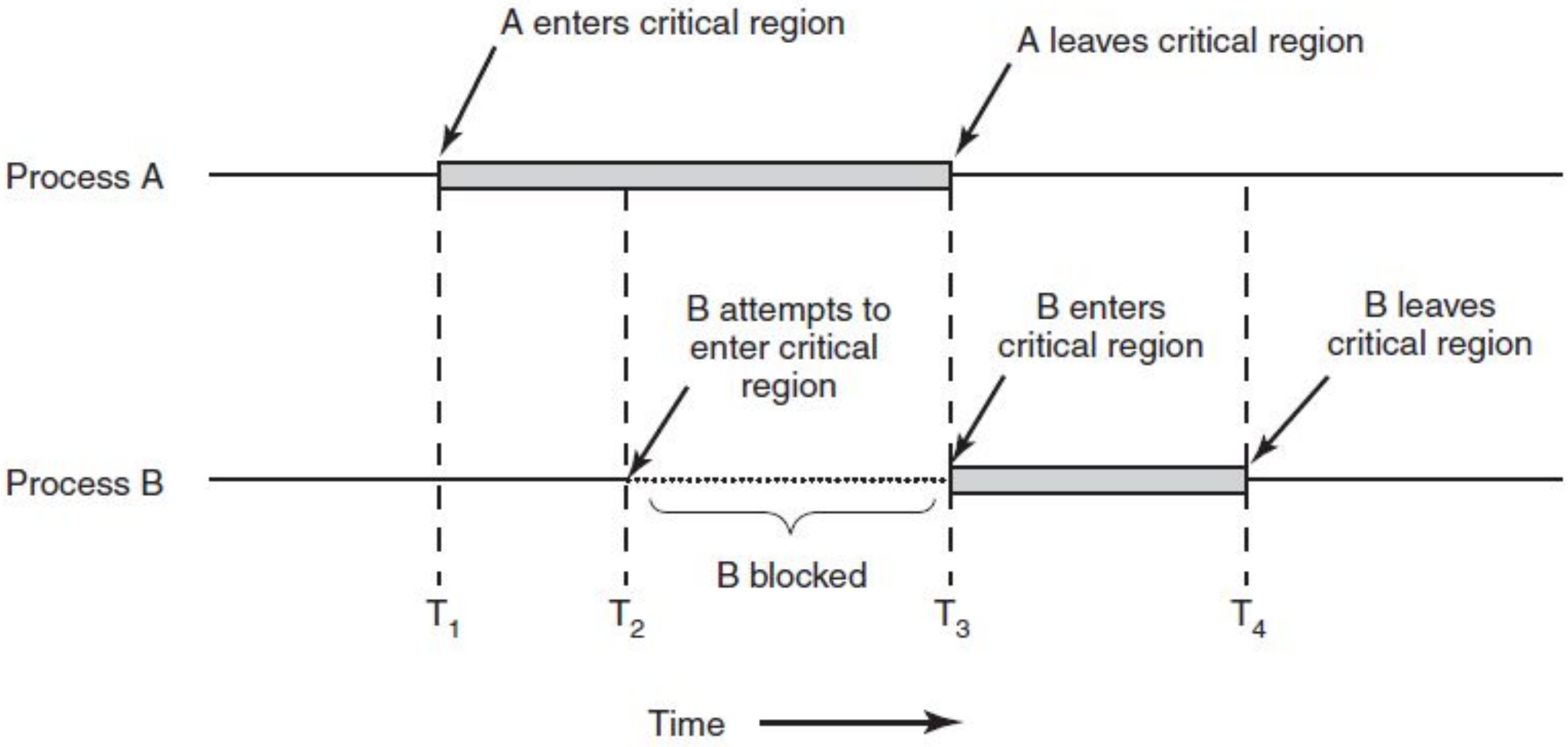
- Consider the following procedure:

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

MUTUAL EXCLUSION

PREVENTING A RACE CONDITION BY USING MUTUAL EXCLUSION



MUTUAL EXCLUSION

RACE CONDITION IN MULTIPROCESSOR MULTIPROGRAMMING

- Same problem arises even when the processes — A and B 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.

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.

MUTUAL EXCLUSION

RACE CONDITIONS WITH 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.

MONASH University

MUTUAL EXCLUSION

RACE CONDITION EXAMPLE WITH MULTIPLE SHARED DATA RESOURCES

- `Process A`

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

- `Process B`

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

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

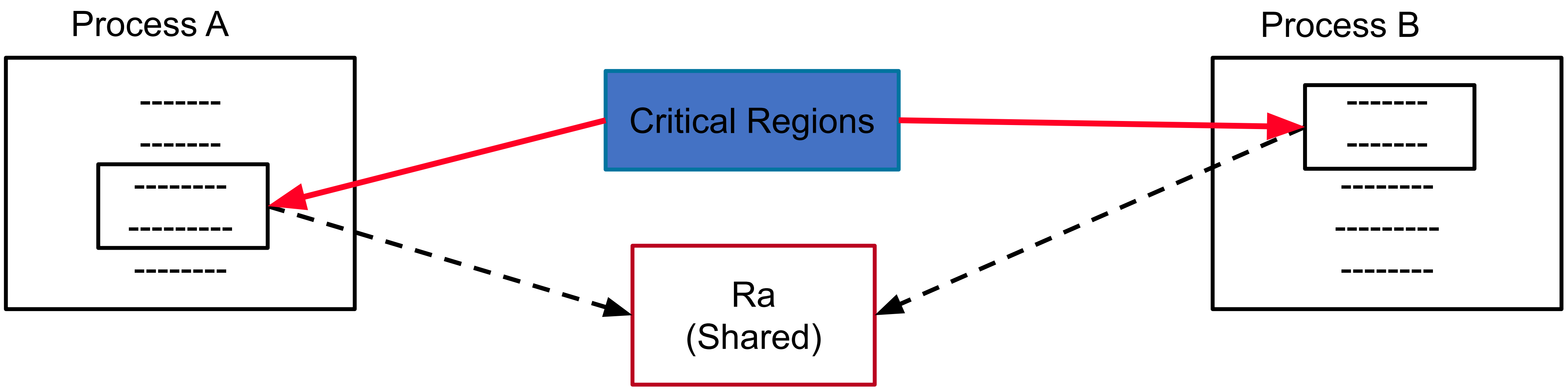
```
a = a + 1 ;   /* {PA}  */
b = 2 * b ;   /* {PB}  */
b = b + 1 ;   /* {PA}  */
a = 2 * a ;   /* {PB}  */
```

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

- **Solution:** access to both a and b should be enclosed in the same critical region.

RACE CONDITION

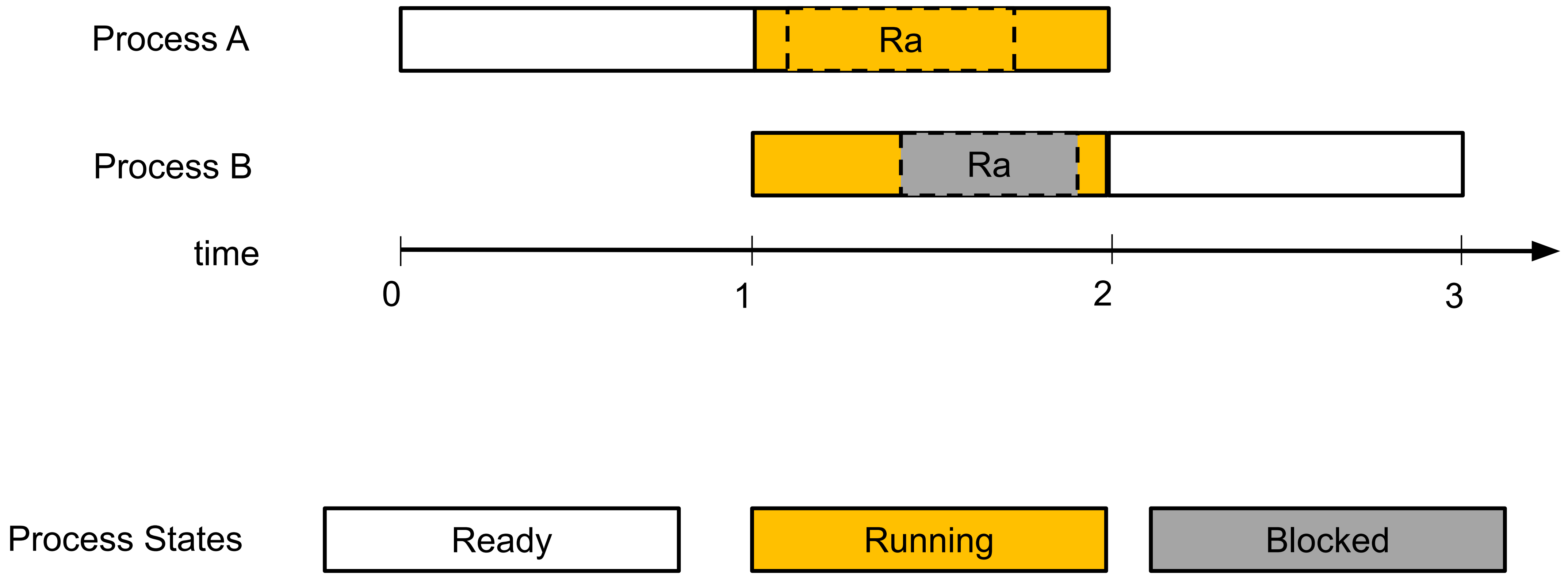
REVISIT



MONASH University

RACE CONDITION

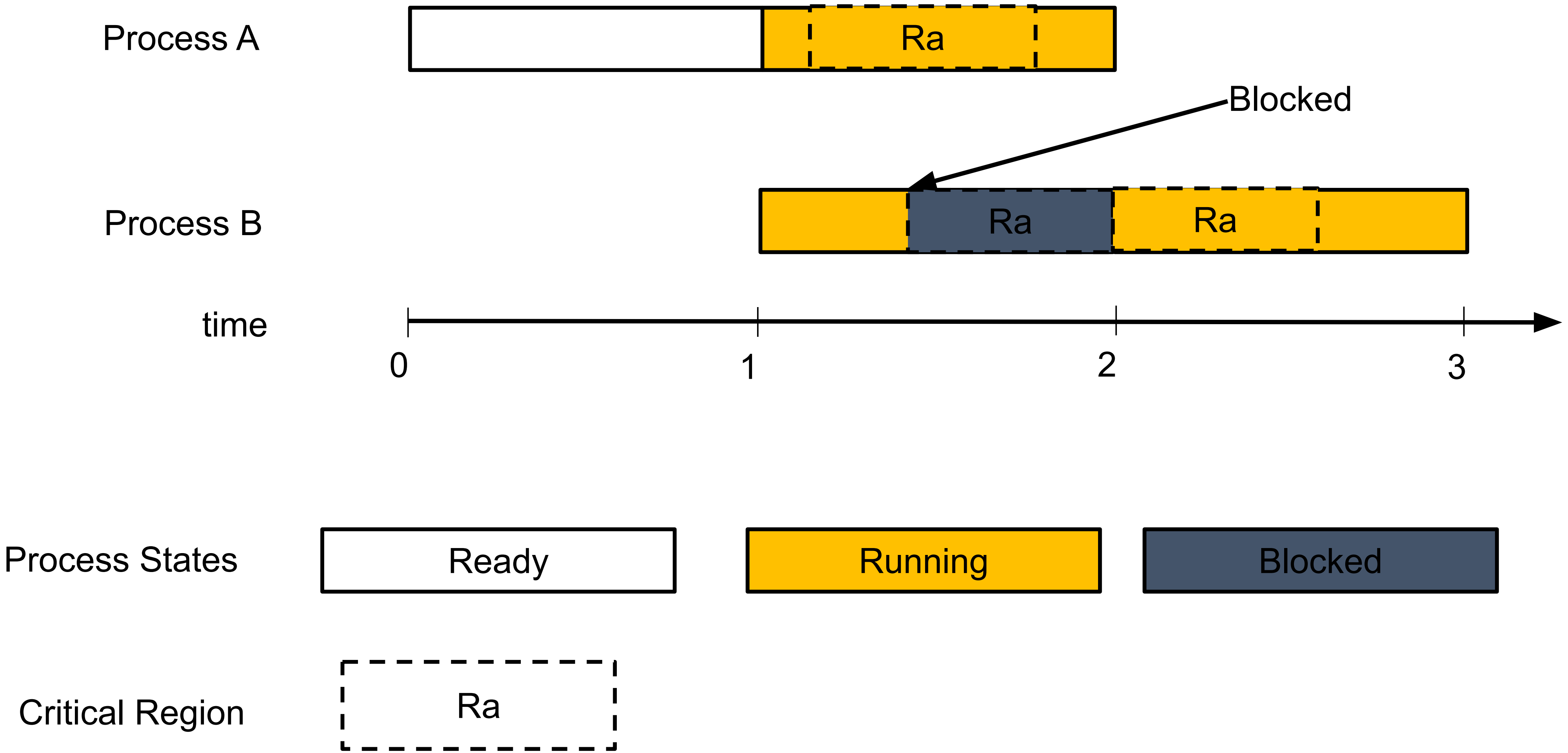
REVISIT



MONASH University

RACE CONDITION

RACE CONDITION PREVENTION WITH MUTUAL EXCLUSION



CONCURRENCY PROBLEMS

MORE CONCURRENCY PROBLEMS

- **Mutual Exclusion causes additional concurrency problems!!!**

- **Deadlock**: two or more processes are waiting indefinitely for the other processes to release the system resources.
- **Starvation**: indefinite blocking of a process.

Summary

- So far we have discussed
  - Introduction to threads
  - Thead Model
  - Thread implementations
  - Concurrency Problems

- Next week
  - Advanced Concurrency

- Reading
  - Tanenbaum, Chapter 2 (4th Edition)
  - Stallings, Chapter 7 (7th Edition)