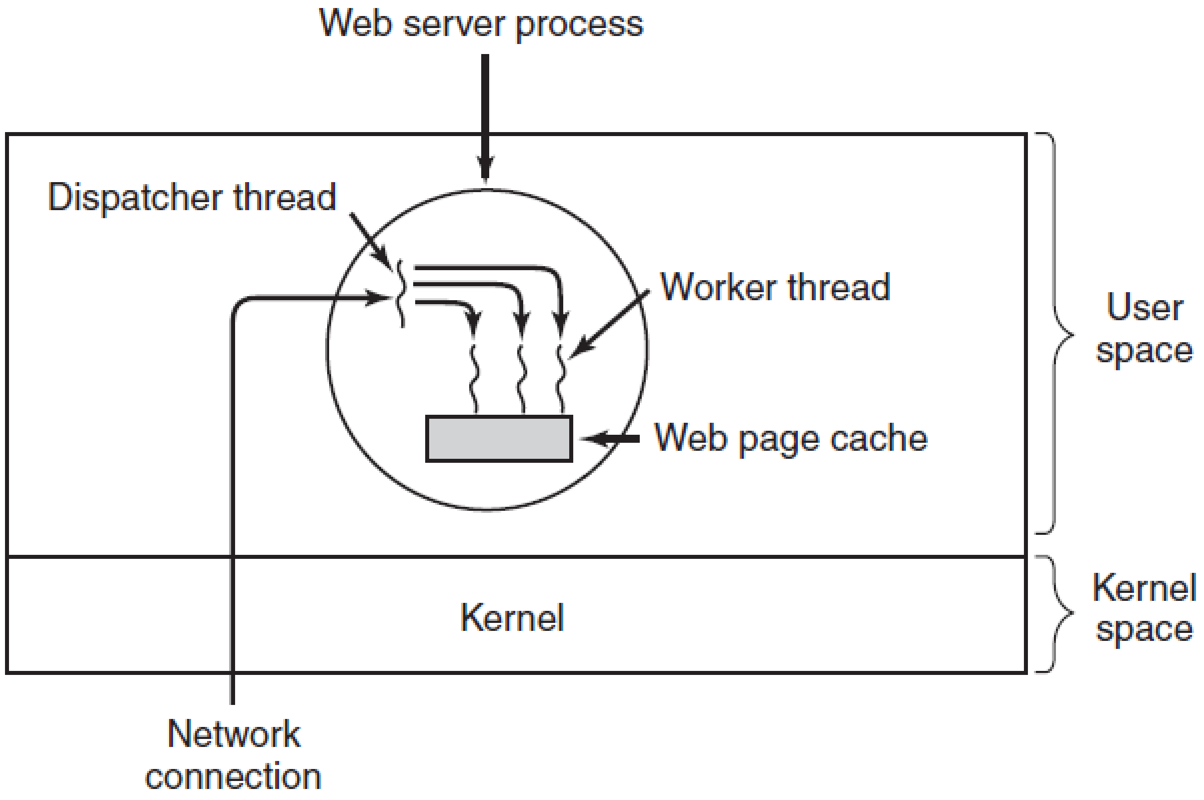
Agenda

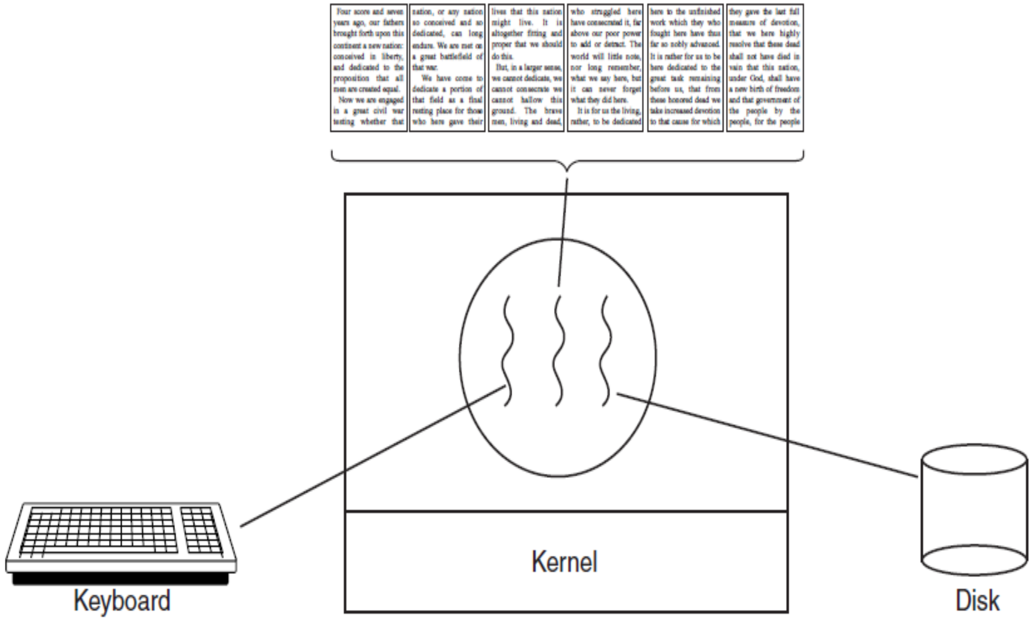
- Assignment 1 due September 29!
- Today's lecture
  - Multiprogramming in Java
  - IPC
  - Issues with Concurrency
  - Synchronization
- Reading: Sections 3.4, 4.4-4.5, 6.1-6.2
- Next : More on Synchronization

Multiprogramming Examples

- Web Server

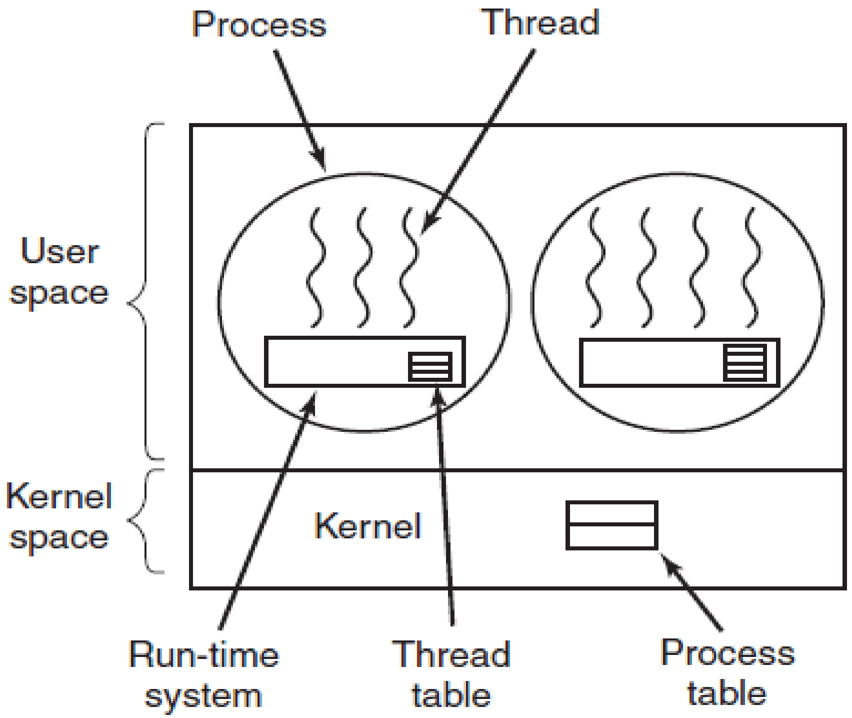


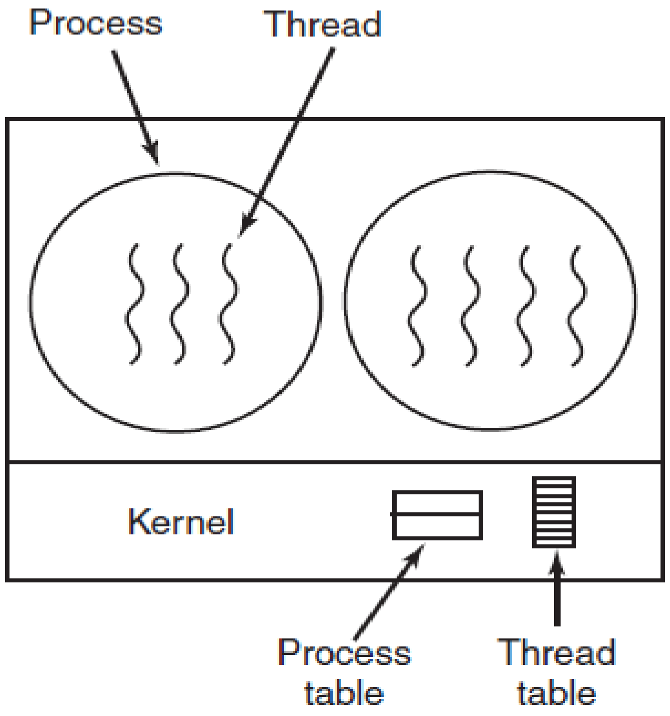
- Word processor



Thread Libraries

- Two primary ways of implementing threads
  - Library entirely in user space
  - Kernel-level library supported by the OS
- **Provide** programmer with API for creating and managing threads
  - POSIX Pthreads
  - Win32
  - Java





Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

Win32 Threads

- Kernel level library
- Shared data declared globally as with Pthreads

Java Threads

- Java threads are managed by the JVM
  - Implemented using host thread library
  - Win32 API on Windows
  - Pthreads on Unix and Linux
- User thread – kernel thread mapping
  - Depends on the host operating system
  - One-to-one model on Windows XP
- Java threads may be created by:
  - Implementing the Runnable interface

```
public interface Runnable
{
    public abstract void run();
}
```

Java Threads – Example Program

```java
class Sum
{
        private int sum;

        public int get() {
                return sum;
        }

        public void set(int sum){
                this.sum = sum;
        }
}
```

```java
class Summation implements Runnable
{
    private int upper;
    private Sum sumValue;

        public Summation(int upper, Sum  sumValue)
        {
                this.upper = upper;
                this.sumValue = sumValue;
        }

        public void run() {
                int sum = 0;

                for (int i = 0; i <= upper; i++)
                sum += i;

                sumValue.set(sum);
        }
}
```

Java Threads - Example Program

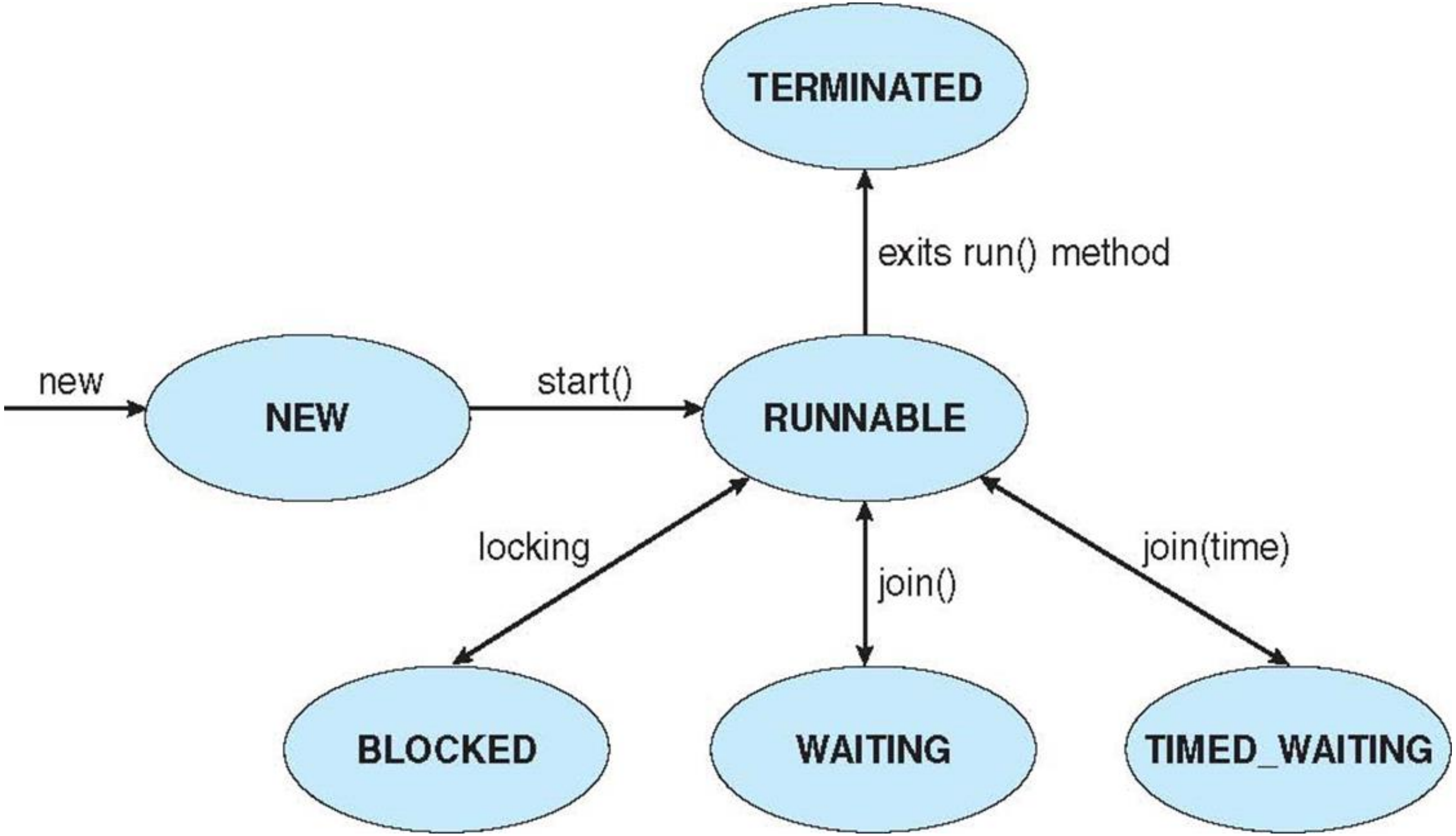
```java
public static void main(String[] args) {
        if (args.length != 1) {
                System.err.println("Usage Driver <integer>");
                System.exit(0);
        }
        if (Integer.parseInt(args[0]) < 0) {
                System.err.println(args[0] + " must be >= 0");
                System.exit(0);
        }

        // Create the shared object
        Sum sumObject = new Sum();
        int upper = Integer.parseInt(args[0]);

        Thread worker = new Thread(new Summation(upper, sumObject));
        worker.start();

        try {
        worker.join();
        } catch (InterruptedException ie) { }
        System.out.println("The sum of " + upper + " is " + sumObject.get());
}
```

1. Allocate memory and initialize thread
2. Call the run () method

Shared data by passing reference to objects

Wait for thread termination

Java Thread States



Threading Issues

- Semantics of **fork()** and **exec()** system calls
  - Does **fork()** duplicate only the calling thread or all threads?
- **Thread cancellation** of **target thread**
  - Asynchronous or deferred
- **Signal** handling
- **Thread pools**
- **Thread-specific data**
- **Scheduler activations**

Thread Cancellation

- Terminating a thread before it has finished

- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
    - JAVA API provides the Interrupt() method which sets the interrupt status of the target thread
    - Threads periodically check their interrupt status
    - If interrupt status set, clean up and terminate

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred

- A **signal handler** is used to process signals.
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled

- Options:
  - Deliver the signal to the thread to which the signal applies (e.g., synchronous signals)
  - Deliver the signal to every thread in the process (e.g., terminating signal <control><c>)
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
- Allows the number of threads in the application(s) to be bound to the size of the pool
- `Java.util.concurrent` package includes an API for thread pools

Thread Specific Data

- Allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

Scheduler Activations

- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number of kernel threads

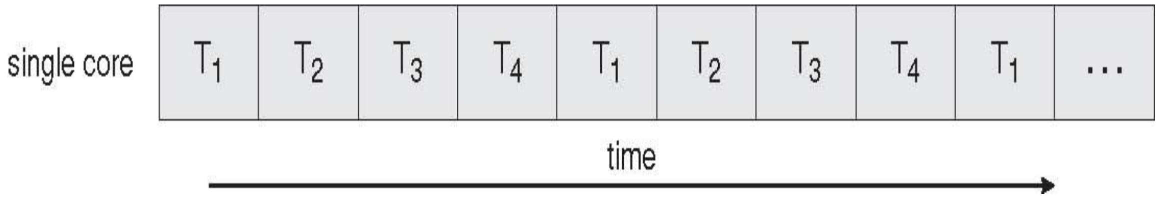
Concurrent Processes - Revisited

- Most OS provide process and thread abstractions to allow creation of multithread and multi-process application
- Hence, we talk about **concurrent processes**

Concurrent Processes - Revisited

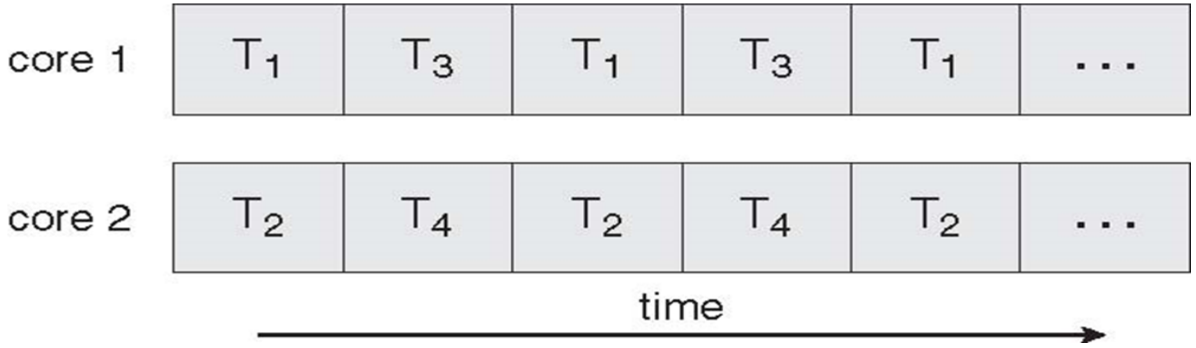
Concurrent Execution

- 2 or more tasks **seem** to be performed simultaneously
- Implemented in most modern PCs
- How does it work?
- Recall: Scheduling based on Time-slicing and pre-emption



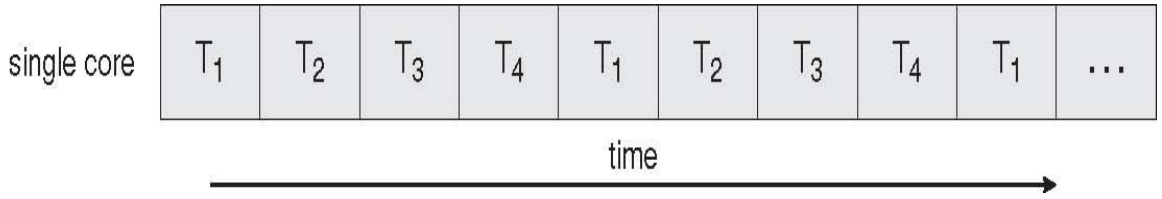
Parallel Execution

- 2 or more tasks **are** performed simultaneously
- Requires cooperation from hardware
  - Multiple CPUs
  - CPU with multicores
  - Multithreaded CPU



Concurrent Execution

- Multitasking, multiprocessing, or multiprogramming
  - Processes **seem** to be running simultaneously
- Achieved using Time slicing and Preemption
  - A process, Tx, gets to run for 1 time quantum (e.g., 10-100ms)
  - When quantum expires, Tx is interrupted
  - Scheduler selects Ty to run next
  - Tx is placed in the ready queue
  - Ty runs from where it was last interrupted
  - Repeat



Cooperating Processes

- Processes within a system may be **independent** or **cooperating**
- **Cooperating** process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing: e.g., files
  - Computation speedup: e.g., distribute workload among CPUs
  - Modularity: e.g. client/server architecture
  - Convenience: e.g., facilitate maintenance, security

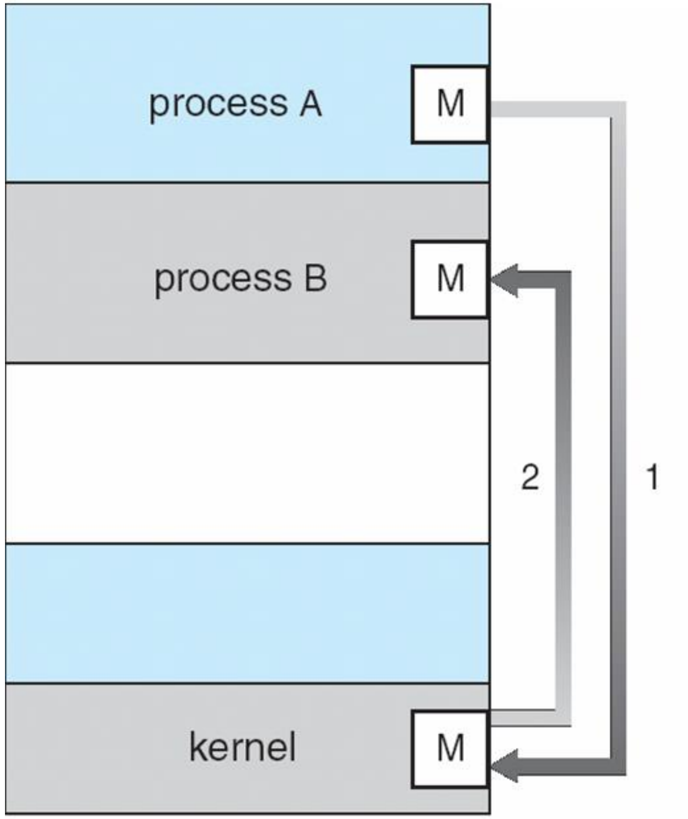
Cooperating Processes

- Cooperating processes need
  - **Interprocess communication** (**IPC**): exchanging data from one process to another
  - Synchronization: coordinating process activities
    - All threads of a process share the same address space and other resources
    - Any alteration of a resource by one thread affects the other threads in the same process
- The OS typically provides support for both IPC and synchronization

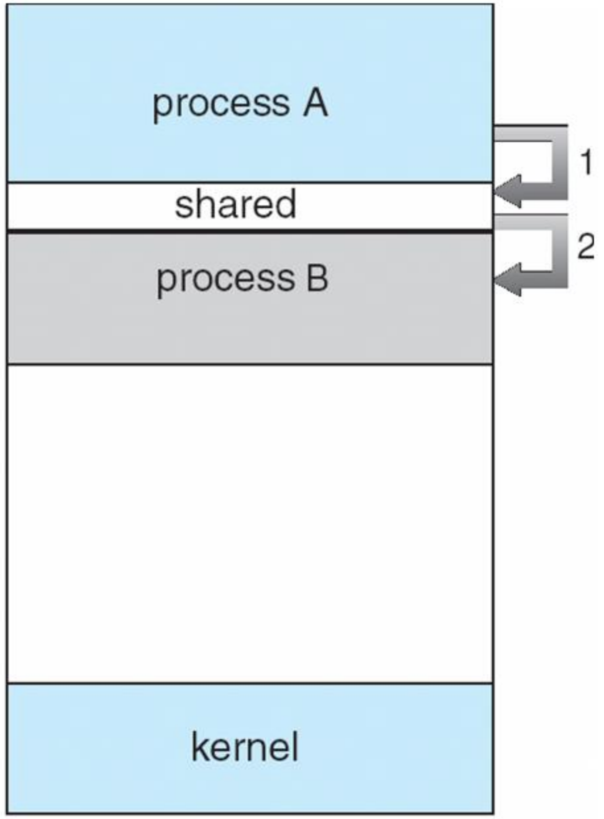
Interprocess Communication

- Cooperating processes need **interprocess communication (IPC)**
- Two models of IPC
  - Shared memory
    - E..g, global variables
    - E.g., threads
  - Message passing
    - E.g., e-mail, TCP/IP

Message passing



Shared Memory



Difficulties of Concurrency

- **By Definition, a cooperating** process can affect or be affected by the execution of another process
  - Recall: OS tries to prevent one process from accessing another process's memory
  - But IPC uses shared memory!!
- Concurrent access to shared data may result in data inconsistency
  - Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- Difficult for the OS to manage the allocation of resources optimally
- Difficult to locate programming errors as results are not deterministic and reproducible

Producer-Consumer Problem

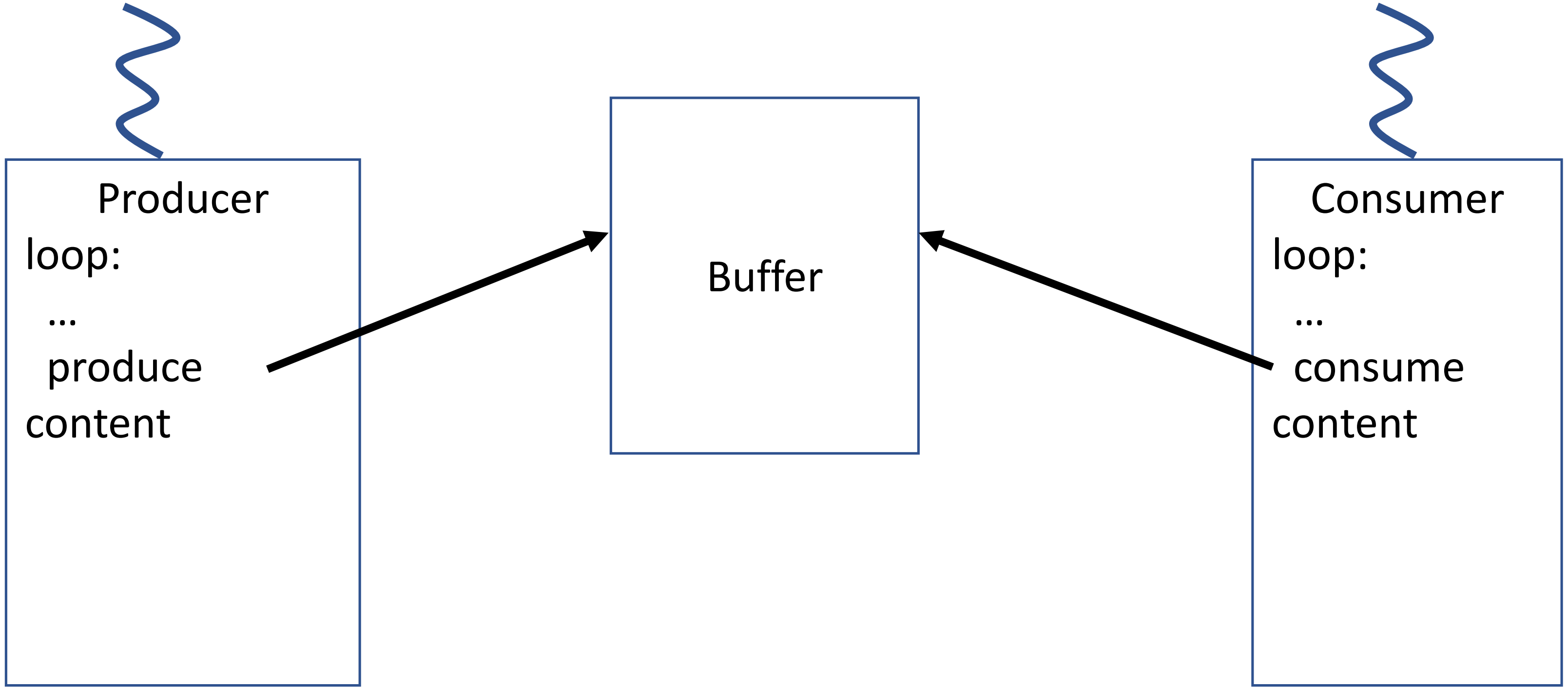
- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process
  - Word processor produces files to be consumed by the printer
  - Client/server paradigm
    - Server: web server produces files
    - Client: web browser consumes files

Producer-Consumer Problem

- The Producer-Consumer Problem consists of:
  - 2 or more processes or threads
  - 1 or more processes produce data
  - 1 or more processes consume the data
  - A shared buffer (or queue)
  - Producer(s) must safely pass the data to consumer(s)
- Two approaches
  - *unbounded-buffer:* places no practical limit on the size of the buffer
    - Producer can produce at will
    - Consumer can consume only if the buffer is not empty
  - *bounded-buffer*: assumes that there is a fixed buffer size
    - Producer can produce only if the buffer is not full
    - Consumer can consume only if the buffer is not empty

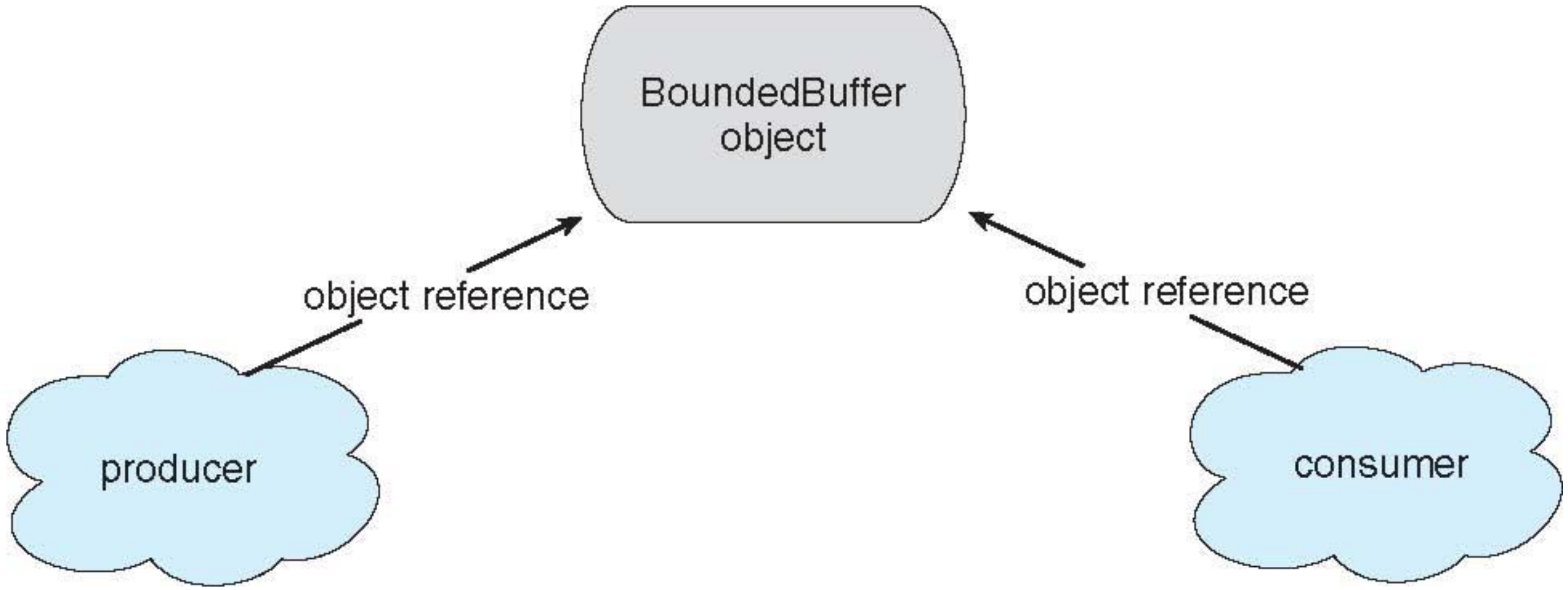
Producer-Consumer Problem

- Synchronization Challenges:
  - Empty buffer
  - Full buffer
  - Shared variables



Simplified Producer-Consumer

Simulating Shared memory in Java



Java Threads – Producer - Consumer

```java
public interface Buffer <E>
{
// producers call this method
public void insert(E item);

// consumers call this method
public E remove();
}
```

```java
public class BufferImpl<E> implements Buffer<E>{
private static final int BUFFER_SIZE = 5;
private E[] elements;
private int in, out, count;

public BufferImpl() {
count = 0;
in = 0;
out = 0;
elements = (E[]) new Object[BUFFER_SIZE];
}

public void insert(E item) {} // producers call this method
public E remove() {} // consumers call this method
}
```

Java Threads – Producer - Consumer

```java
// producers call this method
public void insert(E item) {

while (count == BUFFER_SIZE)
; // do nothing -- no free space

// add an element to the buffer
elements[in] = item;
in = (in + 1) % BUFFER_SIZE;
++count;
}
```

Wait until buffer is not full

```java
// consumers call this method
public E remove() {
E item;

while (count == 0)
; // do nothing - nothing to consume

// remove an item from the buffer
item = elements[out];
out = (out + 1) % BUFFER_SIZE;
--count;

return item;
}
```

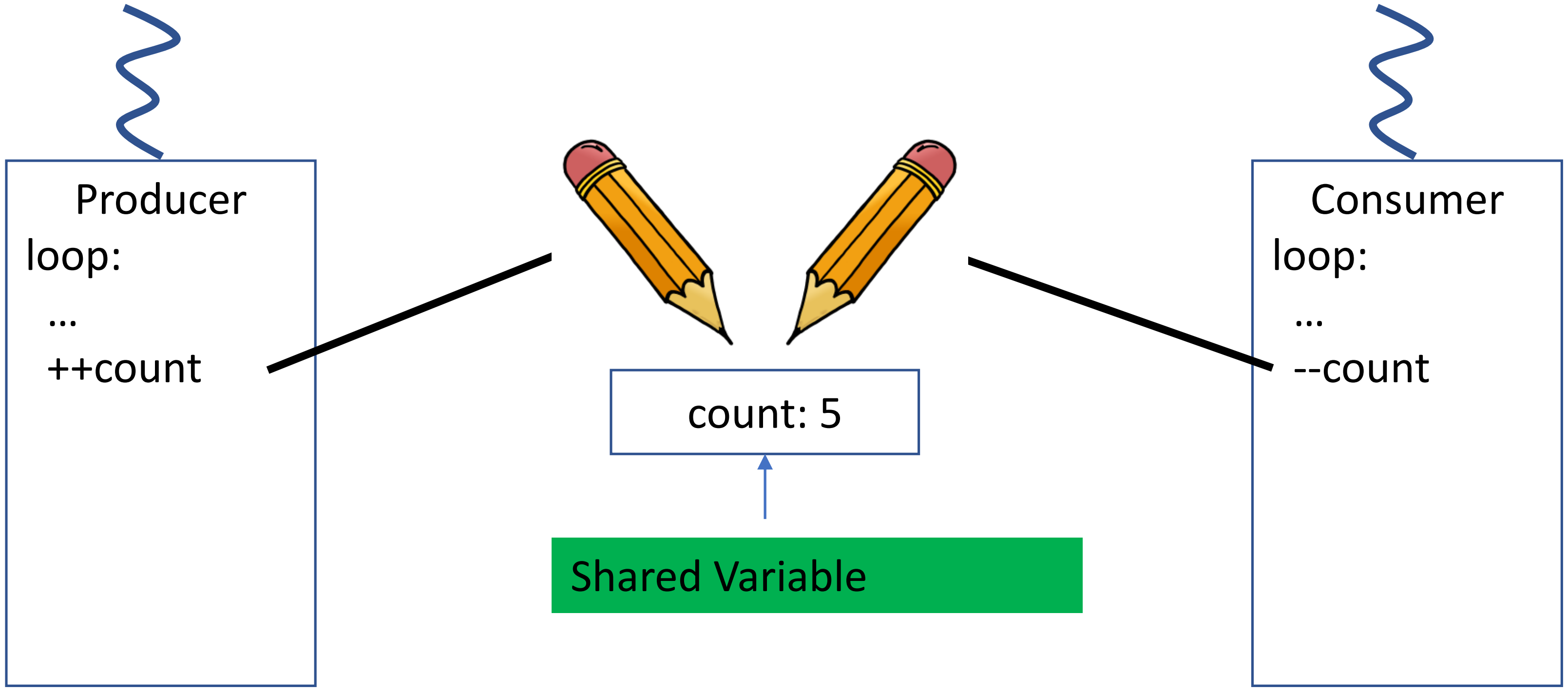
Wait until buffer is not empty

Producer - Consumer

```java
public static void main(String[] args) {
// create the shared object
Buffer<String> boundedBuffer = new BufferImpl<String>();

// create the producer and consumer threads
Thread producer = new Thread(new Producer(boundedBuffer));
Thread consumer = new Thread(new Consumer(boundedBuffer));

// start the threads
producer.start();
consumer.start();
}
```

Simplified Producer-Consumer

Synchronization Challenge: what is the value of `count` assuming producer and consumer threads run concurrently?



Sychronization – Shared Variable

- Machine instructions for ++`count`

```
move count → regA
inc regA
move regA → count
```

- Machine instructions for --`count`

```
move count → regB
dec regB
move regB → count
```

- The Producer and Consumer threads execute these instructions concurrently
- Problem: These instructions can be interleaved!

Interleaving Scenario 1

Instruction	count	regA	regB	
`move count → regA`	5	5	?	
`inc regA`	5	6	?	
`move regA → count`	6	6	?	Context switch
`move count → regB`	6	6	6	
`dec regB`	6	6	5	
`move regB → count`	5	6	5	

Correct: **`count`** has the same value before and after an increment and decrement!

Interleaving Scenario 2

Instruction	count	regA	regB	
`move count → regA`	5	5	?	Context switch
`move count → regB`	5	5	5	
`dec regB`	5	5	4	
`move regB → count`	4	5	4	Context switch
`Inc regA`	4	6	4	
`move regA → count`	6	6	4	

Incorrect: **`count`** has the value as if decrement never happened!

Interleaving Scenario 3

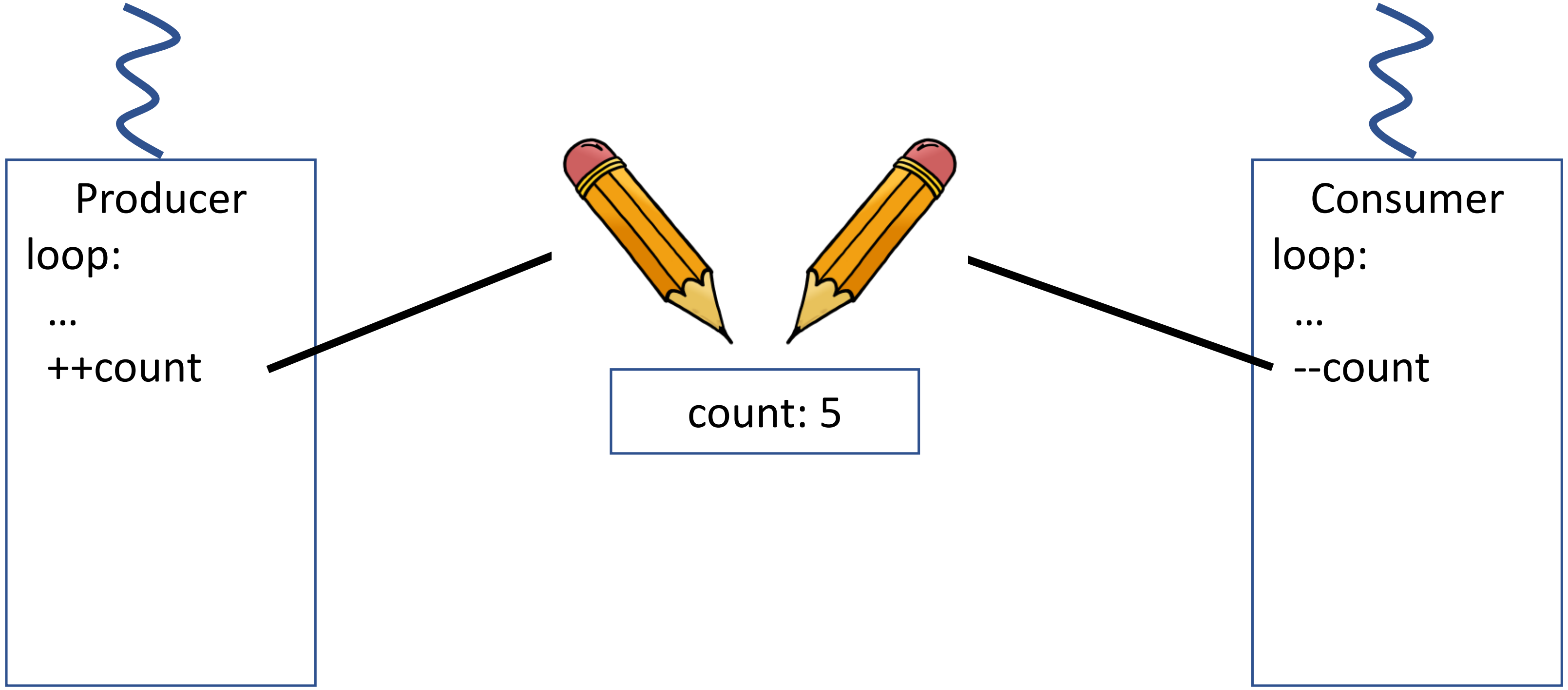
Instruction	count	regA	regB	
`move count → regB`	5	?	5	
`dec regB`	5	?	4	Context switch
`move count → regA`	5	5	4	
`inc regA`	5	6	4	
`move regA → count`	6	6	4	Context switch
`move regB → count`	4	6	4	

Incorrect: **count** has the same value as if increment never happened!

Thread Scheduling

- What was happening?
  - The same code is executed in each scenario
  - The threads were scheduled differently
  - Both threads write to `count`
  - Different behavior depending on when the writes occur
- **BUT,** thread scheduling can be non-deterministic!
- Scheduling in scenarios 2 and 3 resulted in **destructive interference**

Race Condition



- Two processes want to access shared memory at the same time
- A race condition is when *destructive interference* can occur.
- With increasing parallelism due to increasing number of cores, race condition are becoming more common

Critical Region (Section)

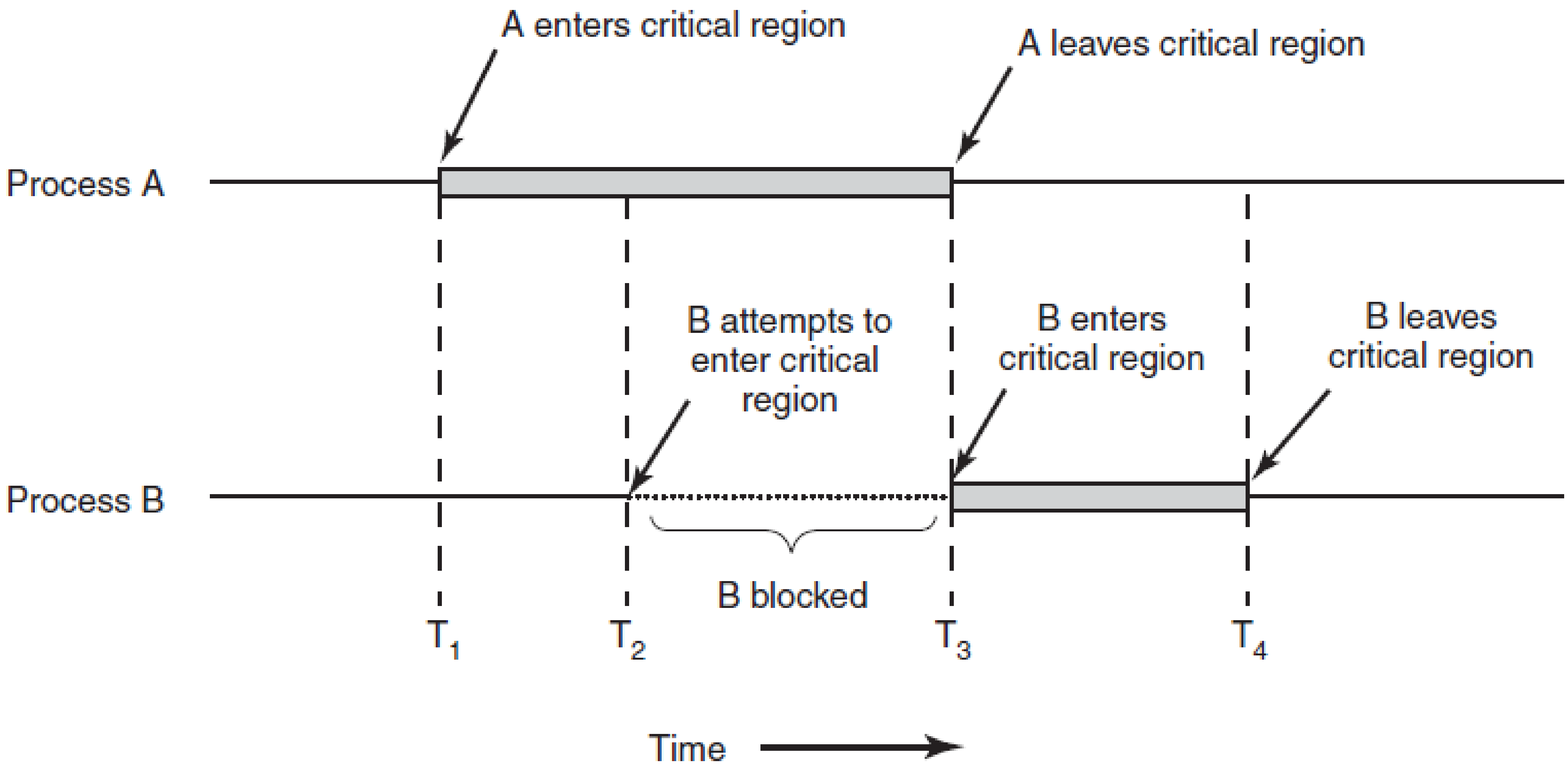
- Part of the program where shared memory is accessed
- Prone to race conditions
- **Problem**: Multiple threads executing concurrently in a critical section can cause errors
- How do we prevent this from happening?

Critical Regions

Requirements to avoid race conditions:

1. **Mutual exclusion**: No two processes may be simultaneously inside their critical regions.
2. **Scheduler independent**: No assumptions may be made about speeds or the number of CPUs.
3. **Allows progress**: No process running outside its critical region may block other processes.
4. **Starvation free**: No process should have to wait forever to enter its critical region.

Mutual Exclusion Using Critical Region



Next…

- Solutions to synchronization issues
  - Software solutions
  - Hardware solutions