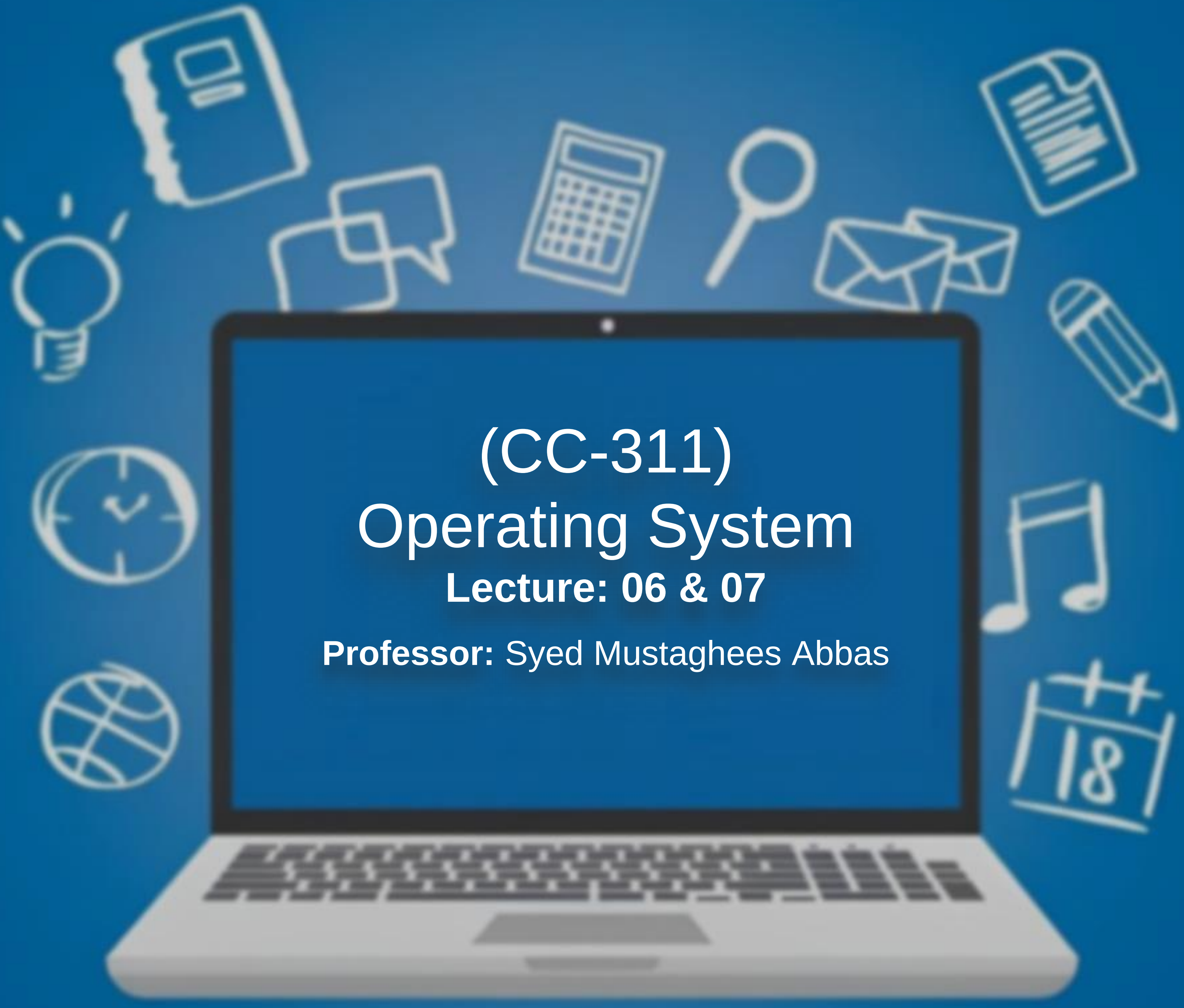


(CC-311) Operating System

Lecture: 06 & 07

Professor: Syed Mustaghees Abbas

Communication in Client-Server Systems

Client
Server

Sockets

- A **socket** is defined as an endpoint for communication.
- A pair of processes communicating over a network employ a pair of sockets, one for each process.
- A socket is identified by an **IP address** concatenated with a **port number**.
- Usually used in client-server architecture. The server waits for incoming client requests by listening to a specified port.
- A port is piece of software which is used as docking point in your machine, where remote application can communicate. This is analogy to the physical ports for entering in to a country from different sea ports.

Sockets (Continue)

- Port numbers can vary from 0 to 65535, so total we can get 65536 ports
- This is because limitation in TCP/IP stack where the port number field is just 16bit size. So we get only 2^{16} ports (65536 available ports)
- **Well known ports** are from 0 to 1023 (total 2^{10} = 1024 ports)
- Servers implementing specific services (such as telnet, ftp, and http) listen to well-known ports (a **telnet server** listens to **port 23**, an **ftp server** listens to **port 21**, and a **web**, or **http server** listens to **port 80**).

Sockets (Continue)

- Well known port is a designated port for particular well-known service such as web server, mail server, ftp server etc.
  - ✓ 20 – FTP Data (For transferring FTP data)
  - ✓ 21 – FTP Control (For starting FTP connection)
  - ✓ 80 – HTTP/WWW(apache)
  - ✓ 443 – HTTPS(HTTP+SSL for secure web access)

Sockets (Continue)

- When a client process initiates a request for a connection, it is assigned a port by the host computer. This port is some arbitrary number greater than 1024.

- For example, if a client on host X with IP address 146.86.5.20 wishes to establish a connection with a web server (which is listening on port 80) at address 161.25.19.8, host X may be assigned port 1625. The connection will consist of a pair of sockets: (146.86.5.20:1625) on host X and (161.25.19.8:80) on the web server. This situation is illustrated in Figure. The packets traveling between the hosts are delivered to the appropriate process based on the destination port number.

Sockets (Continue)

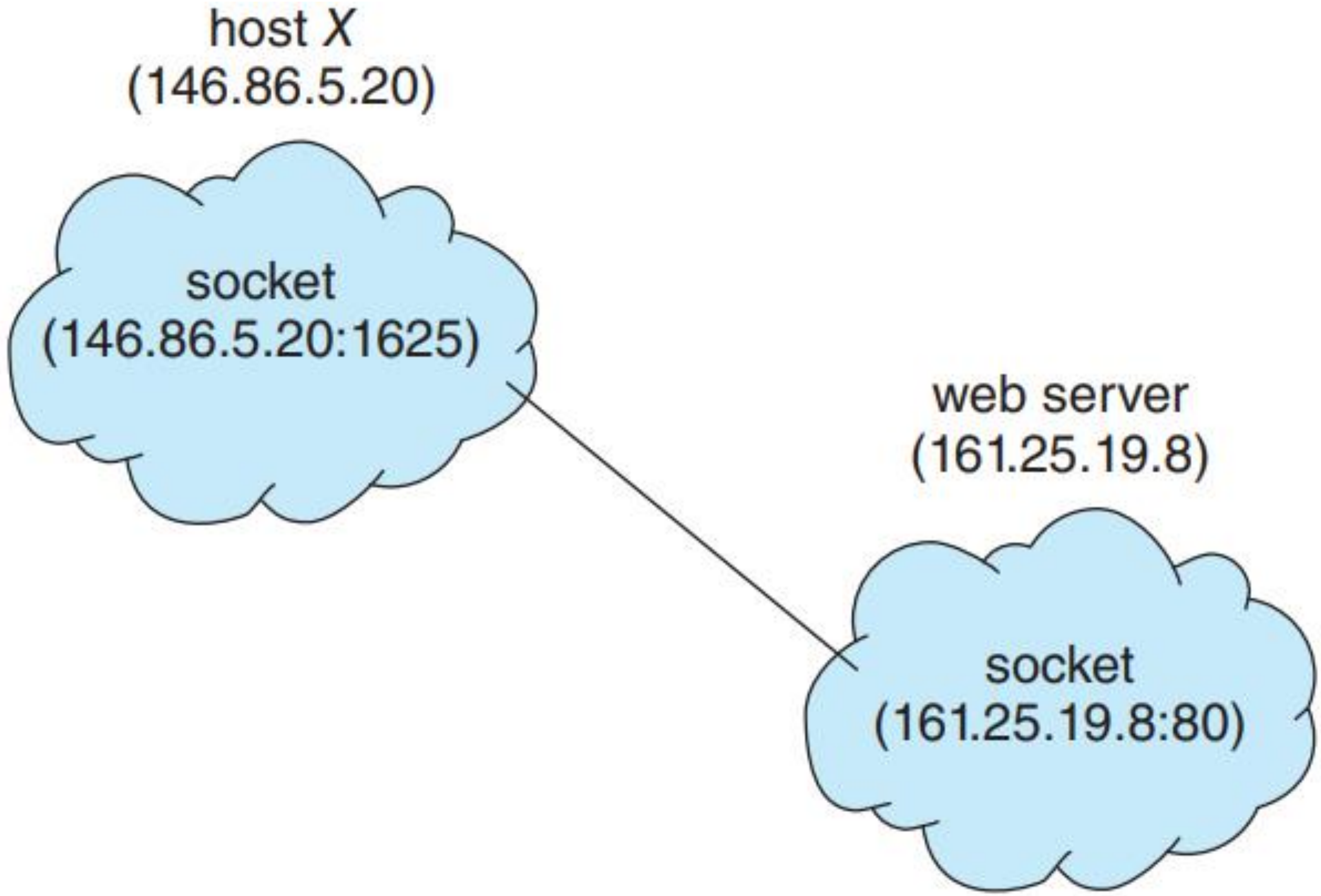


Figure Communication using sockets.

Sockets (Continue)

- There are two widely used socket types:
  1. **Stream sockets**
  2. **Datagram sockets**.
- **Stream sockets** treat communications as a continuous stream of characters
- Stream sockets **use** (connection-oriented) **TCP** (Transmission Control Protocol), which is a reliable, stream oriented protocol or.
- **Datagram sockets** have to read entire messages at once.
- datagram sockets **use** (connection-less) **UDP** (User Datagram Protocol), which is unreliable and message oriented.

Problems with Sockets

- Sockets forces us to design our distributed applications using a read/write (input/output) interface which is not how we generally think about application design and how different functional blocks of an application communicate.

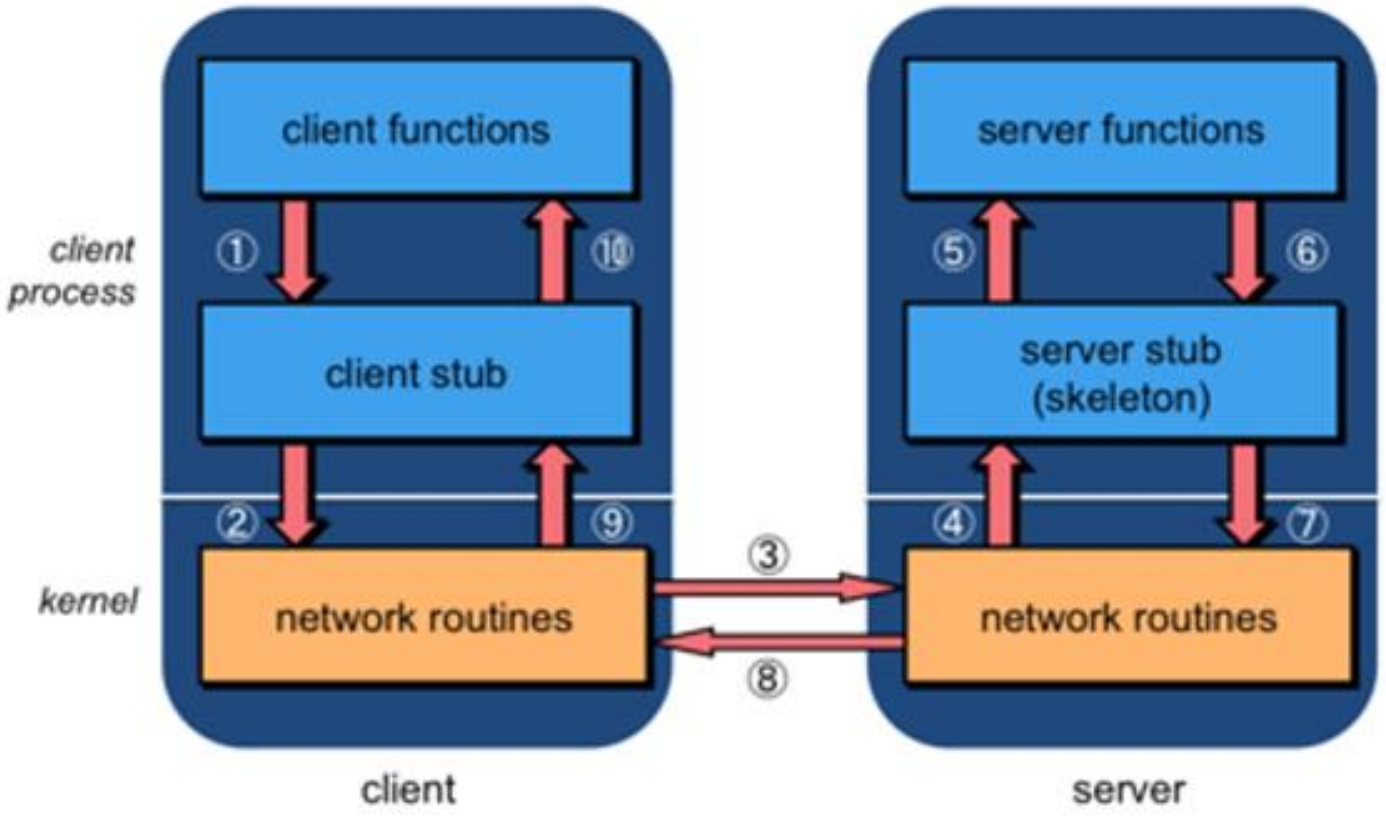
- In designing single-process applications, the procedure call is usually the standard, most popular, and most familiar interface model.

- If we want to make distributed computing look like centralized computing, input-output-based streams are not the way to accomplish this.

Remote Procedure Calls

- RPC was designed as a way to abstract the procedure-call mechanism for use between systems with network connections.
- Messages exchanged in RPC communication are well structured.
- Each message is addressed to an RPC daemon listening to a port on the remote system, and each contains an identifier of the function to execute and the parameters to pass to that function.
- The function is then executed as requested, and any output is sent back to the requester in a separate message.
- The semantics of RPCs allow a client to invoke a procedure on a remote host as it would invoke a procedure locally.

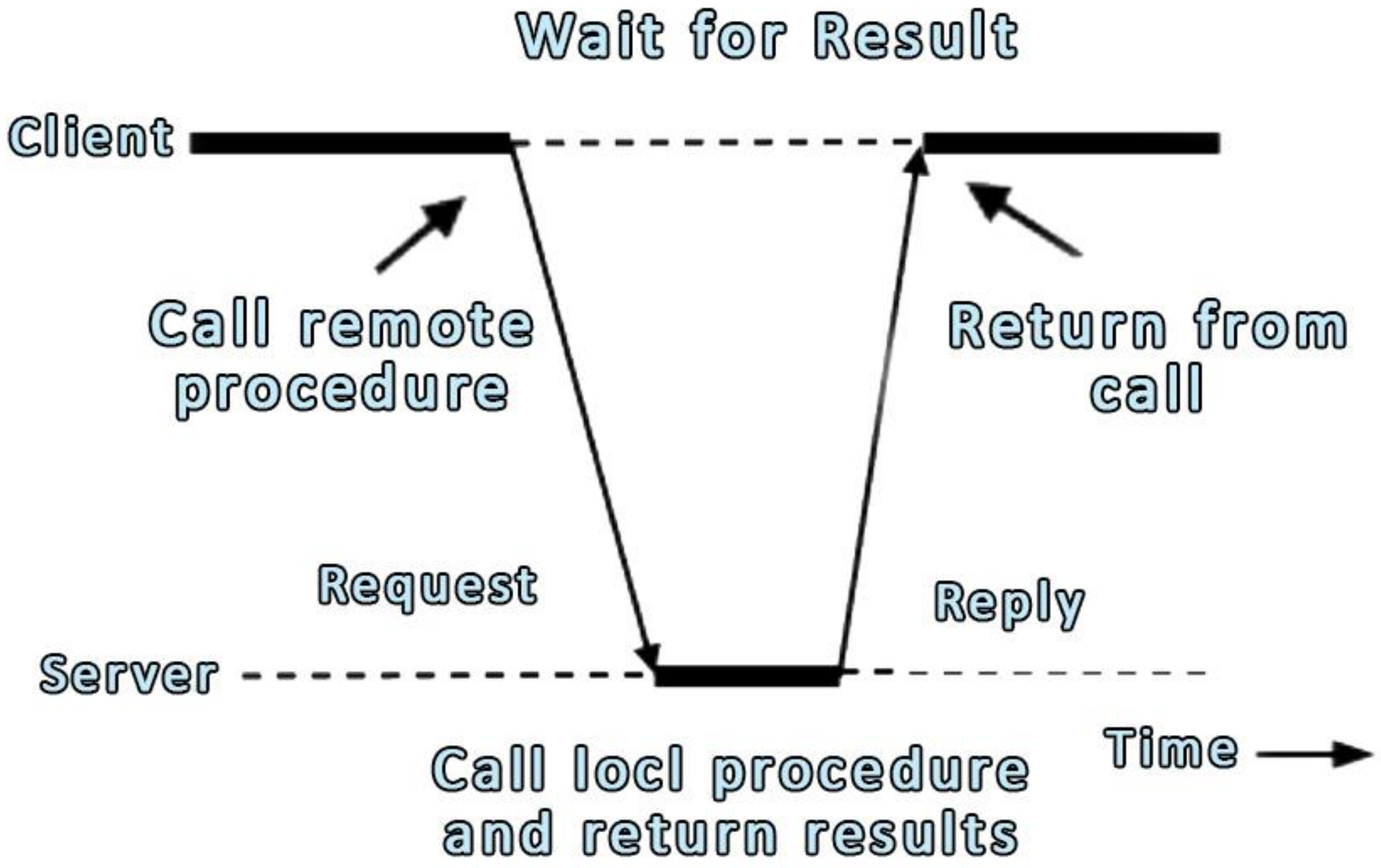
Remote Procedure Calls (Continue)



Remote Procedure Calls (Continue)

- The RPC system hides the details that allow communication to take place by providing a **stub** on the client side.
- Separate stub exists for each separate remote procedure. RPC system calls the appropriate stub, passing it the parameters provided to the remote procedure.
- This stub locates the port on the server and then transmits a message to the server using message passing.
- A similar stub on the server side receives this message and invokes the procedure on the server. If necessary, return values are passed back to the client using the same technique.

Remote Procedure Calls (Continue)



Steps of a Remote Procedure Call

1. Client procedure calls client stub in normal way
2. Client stub builds message, calls local OS
3. Client's OS sends message to remote OS
4. Remote OS gives message to server stub
5. Server stub unpacks parameters, calls server
6. Server does work, returns result to the stub
7. Server stub packs it in message, calls local OS
8. Server's OS sends message to client's OS
9. Client's OS gives message to client stub
10. Stub unpacks result, returns to client

Remote Method Invocation (RMI)

- RMI is a Java feature similar to RPCs.
- RMI allows a process to invoke a method on a remote object.
- Objects are considered remote if they reside in a different Java virtual machine (JVM).
- RPCs support procedural programming, whereby only remote procedures or functions can be called. In contrast, RMI is **object-based:** It supports invocation of methods on remote objects.
- Parameters to remote procedures are ordinary data structures in RPC; with RMI, it is possible to pass objects as parameters to remote methods.

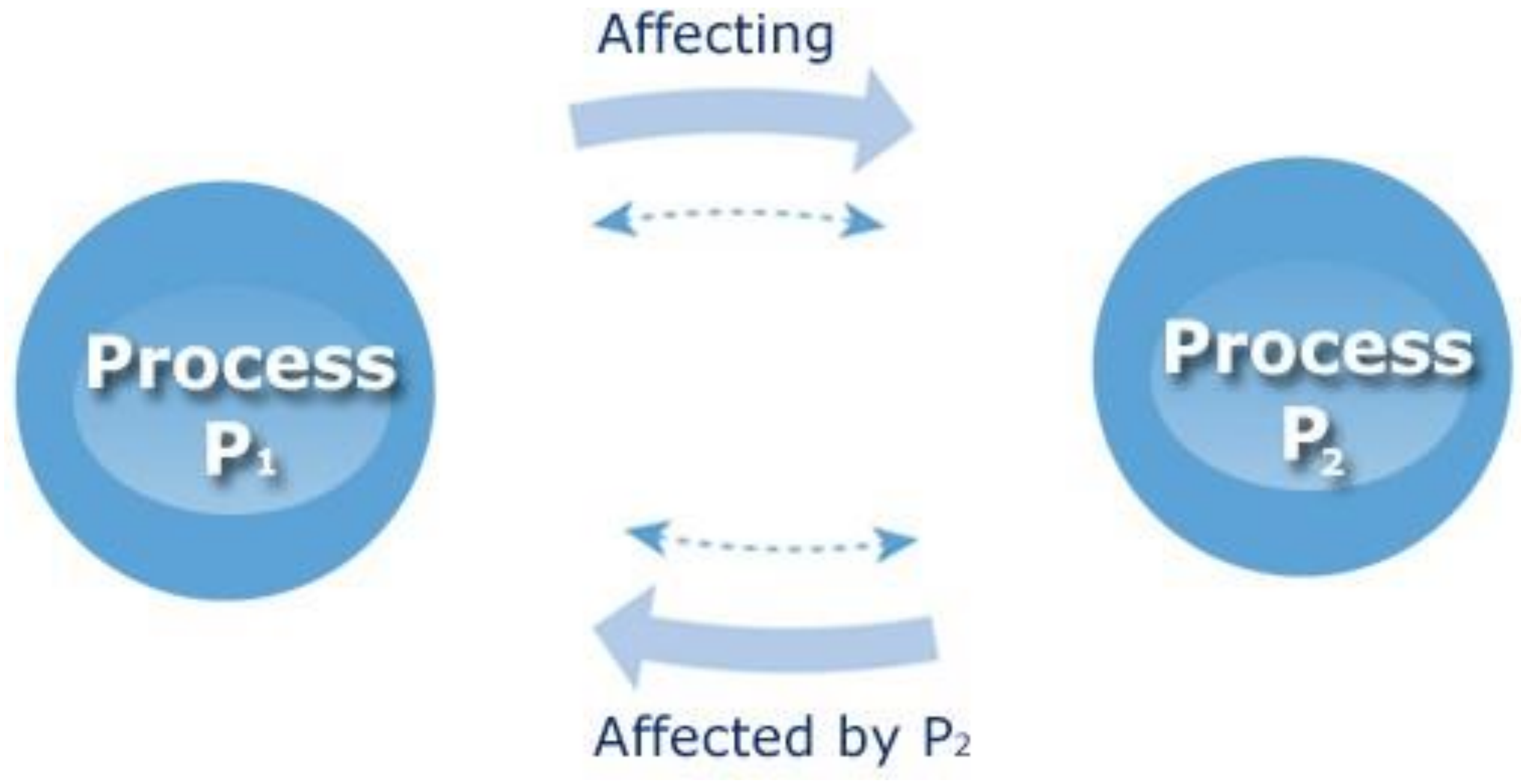
Remote Method Invocation (Continued)

- RMI implements the remote object using **stubs** and **skeletons**. A stub is a proxy for the remote object; it resides with the client.
- When a client invokes a remote method, the stub for the remote object is called. This client-side stub is responsible for creating a parcel consisting of the name of the method to be invoked on the server and the marshalled parameters for the method.
- The stub then sends this parcel to the server, where the skeleton for the remote object receives it.
- The skeleton is responsible for unmarshalling the parameters and invoking the desired method on the server.
- The skeleton then marshals the return value (or exception, if any) into a parcel and returns this parcel to the client. The stub unmarshals the return value and passes it to the client.

Process Synchronization

Process Synchronization

- Concurrent access to shared data may result in data inconsistency.



Process Synchronization: Problem

- Although both producer and consumer routines are correct separately, they may not function correctly when executed concurrently.
- For example the variable count is currently 5 and that the producer and consumer processes execute the statements "count++" and "count--" concurrently.
- Following the execution of these two statements, the value of the variable count may be 4, 5, or 6! .(correct result, though, is count == 5)

Race Condition

In machine language

- count++ could be implemented as

```
T1 register1 = count
T2 register1 = register1 + 1
T3 count = register1
```

- count-- could be implemented as

```
S1 register2 = count
S2 register2 = register2 - 1
S3 count = register2
```

- Consider this execution interleaving with "count = 5" initially:

  T1: producer execute register1 = count {register1 = 5}

  T2: producer execute register1 = register1 + 1 {register1 = 6}

  S1: consumer execute register2 = count {register2 = 5}

  S2: consumer execute register2 = register2 – 1 {register2 = 4}

  T3: producer execute count = register1 {count = 6 }

  S3: consumer execute count = register2 {count = 4}

- What will be the value of Count=?

Race Condition (Continued)

- Notice that we have incorrect state "count == 4", indicating that four buffers are full, when, in fact, five buffers are full.
- This happened because we allowed both processes to manipulate the variable count (**shared**) concurrently.
- When several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a **race condition**.
- To guard against the race condition, we need to ensure that only one process at a time can be manipulating the variable count, hence the need for processes synchronization.

The Critical-Section Problem

- In concurrent programming, a critical section is a piece of code that accesses a shared resource (data structure or device) that must not be concurrently accessed by more than one thread of execution.

- The critical-section problem is to design a protocol that the processes can use to cooperate.

- Each process must request permission to enter its critical section. The section of code implementing this request is the **entry section**. The critical section may be followed by an **exit section**. The remaining code is the **remainder section**.

The Critical-Section Problem (Continued)

```
do {

    entry section

        critical section

    exit section

        remainder section

} while (true);
```

Figure General structure of a typical process P_i.

Solution to Critical-Section Problem

A solution to the critical-section problem must satisfy the following three requirements:

Mutual Exclusion: If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.

Progress: If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely.

Bounded Waiting: There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Peterson’s Solution to Critical Section Problem

- A software-based solution to the critical-section problem is known as Peterson's solution.
- It is **restricted to two processes** that alternate execution between their CS and remainder sections.
- Peterson's solution may not work correctly on modern architectures because of machine-language instructions (like example of variable count in producer-consumer in previous slides).
- We present the solution because it provides a good algorithmic description of solving the critical-section problem and addresses the requirements of mutual exclusion, progress, and bounded waiting requirements.

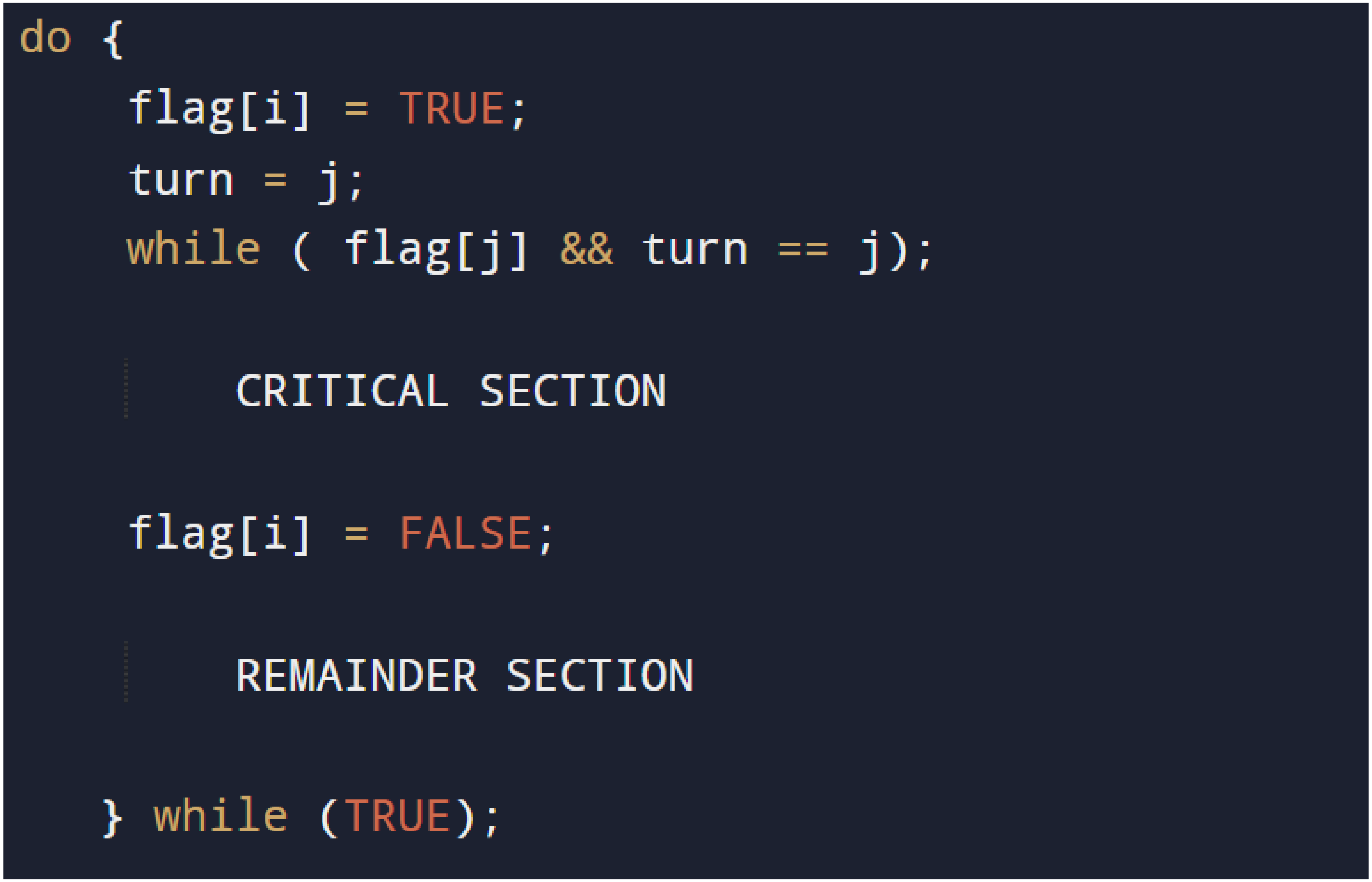
Peterson’s Solution to Critical Section Problem (Continued)

- Processes are numbered P_0 and P_1.
- Peterson's solution requires two data items to be shared between the two processes:
  - `int turn;`
  - `Boolean flag[2]`
- The variable turn indicates whose turn it is to enter the critical section.
- The flag array is used to indicate if a process is ready to enter the critical section. flag[i] = true implies that process P_i is ready!

Peterson’s Solution to Critical Section Problem (Continued)

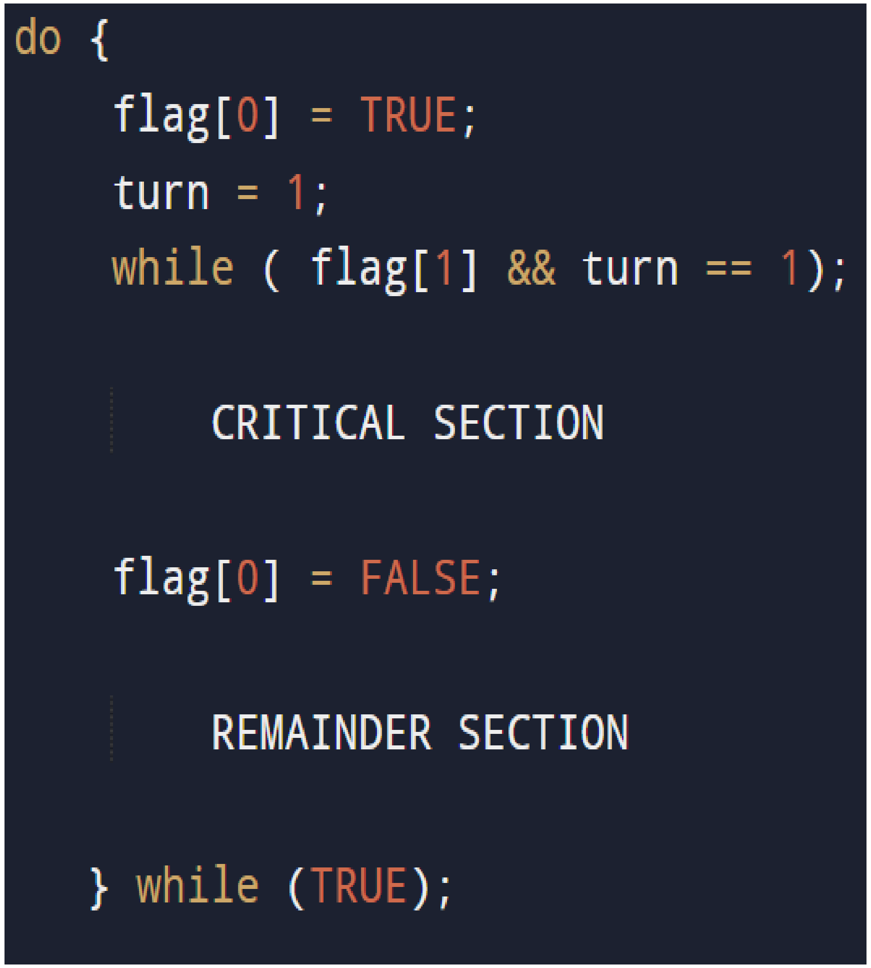
- To enter the critical section, process Pi first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so.
- If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time.
- Only one of these assignments will last; the other will occur but will be overwritten immediately.

Algorithm for Process P_i

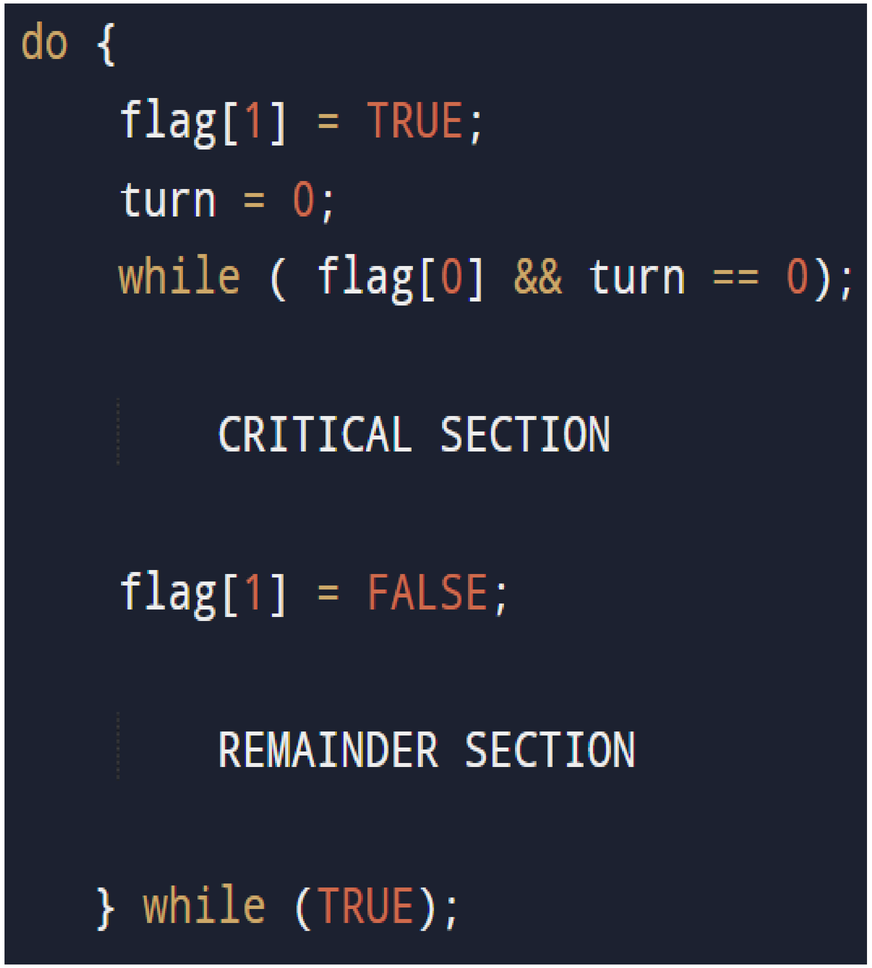


```
do {
    flag[i] = TRUE;
    turn = j;
    while ( flag[j] && turn == j);

        CRITICAL SECTION

    flag[i] = FALSE;

        REMAINDER SECTION

   } while (TRUE);
```

Algorithm for Process P_0 and P_1



```
do {
    flag[0] = TRUE;
    turn = 1;
    while ( flag[1] && turn == 1);

        CRITICAL SECTION

    flag[0] = FALSE;

        REMAINDER SECTION

   } while (TRUE);
```



```
do {
    flag[1] = TRUE;
    turn = 0;
    while ( flag[0] && turn == 0);

        CRITICAL SECTION

    flag[1] = FALSE;

        REMAINDER SECTION

   } while (TRUE);
```

Peterson’s Solution to Critical Section Problem (Continued)

- We now prove that this solution is correct. We need to show that:

  1) Mutual exclusion is preserved?

  2) The progress requirement is satisfied?

  3) The bounded-waiting requirement is met?

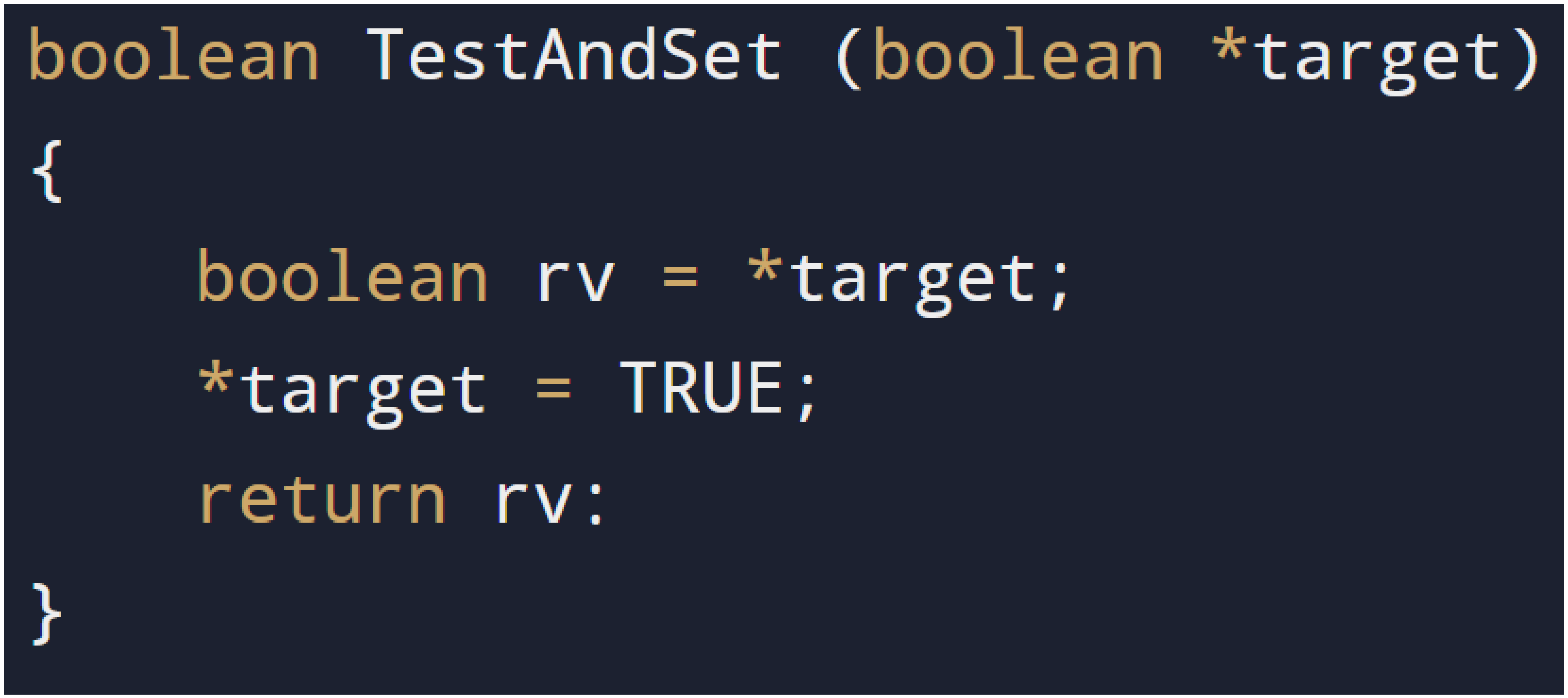
Synchronization Hardware

- Previously we studied a software based solution to critical section problem (Peterson solution).
- Many systems provide hardware support for critical section code.
- **Uniprocessors** – We could disable interrupts for preamption so when a shared data is being modified no preamption will occur.
  - ✓ Generally too inefficient on **multiprocessor** systems. Disabling interrupts on a multiprocessor is very time consuming, as the message is passed to all the processors
- Modern machines provide special atomic hardware instructions

  Atomic = non-interruptable

TestAndSet() instruction

- The important characteristic is that this instruction is executed atomically.
- If two TestAndSet() instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.



```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv:
}
```

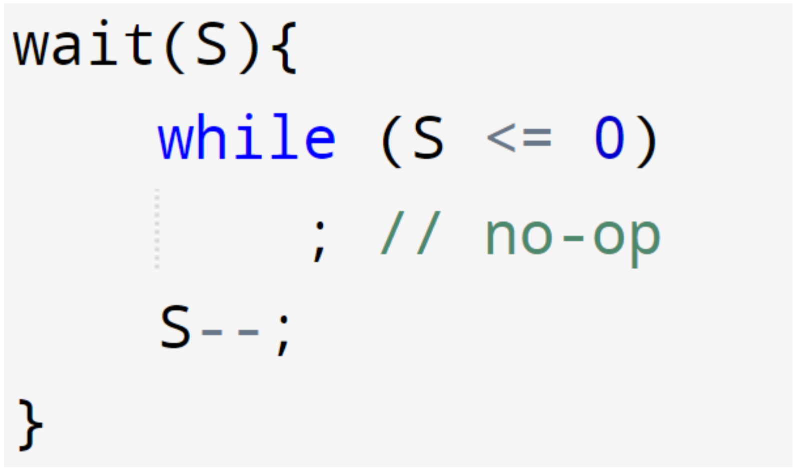
TestAndSet() instruction (Continued)

- If the machine supports the TestAndSet () instruction, then we can implement mutual exclusion by declaring a global Boolean variable lock, initialized to false.
- Solution using TestAndSet()
  Shared Boolean variable lock is initialized to false.

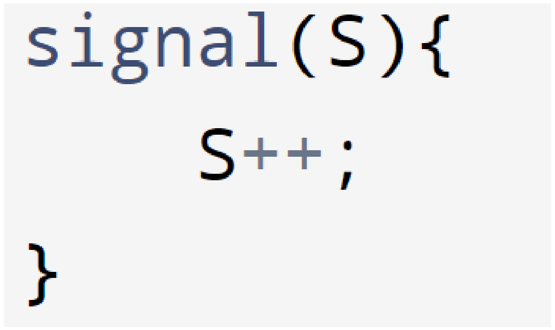
```
do {
    while ( TestAndSet (&lock ))
        ;   // do nothing

        //  critical section

    lock = FALSE;

        //  remainder section

    } while ( TRUE);
```

Semaphore

- Another synchronization tool is called semaphore. A semaphore S is an integer variable that is accessed only through two standard *atomic operations:* **wait ()** and **signal ()**.
- Wait is represented by **P()** and signal by **V()**.



```
wait(S){
    while (S <= 0)
        ; // no-op
    S--;
}
```



```
signal(S){
    S++;
}
```

Semaphore (Continued)

- **Binary semaphore** – integer value can range only between 0 and 1; can be simpler to implement.
  - Also known as *mutex locks*
- **Counting semaphore** – integer value can range over an unrestricted domain.
  - Used to control access to a given resource consisting of a finite number of instances.
  - Semaphore is initialized to the number of resources available. Each process performs a **wait()** operation on the semaphore (thereby decrementing the count).
  - When a process releases a resource, it performs a **signal()** operation (incrementing the count).
  - When the count for the semaphore goes to 0, all resources are being used. After that, processes that wish to use a resource will block until the count becomes greater than 0.

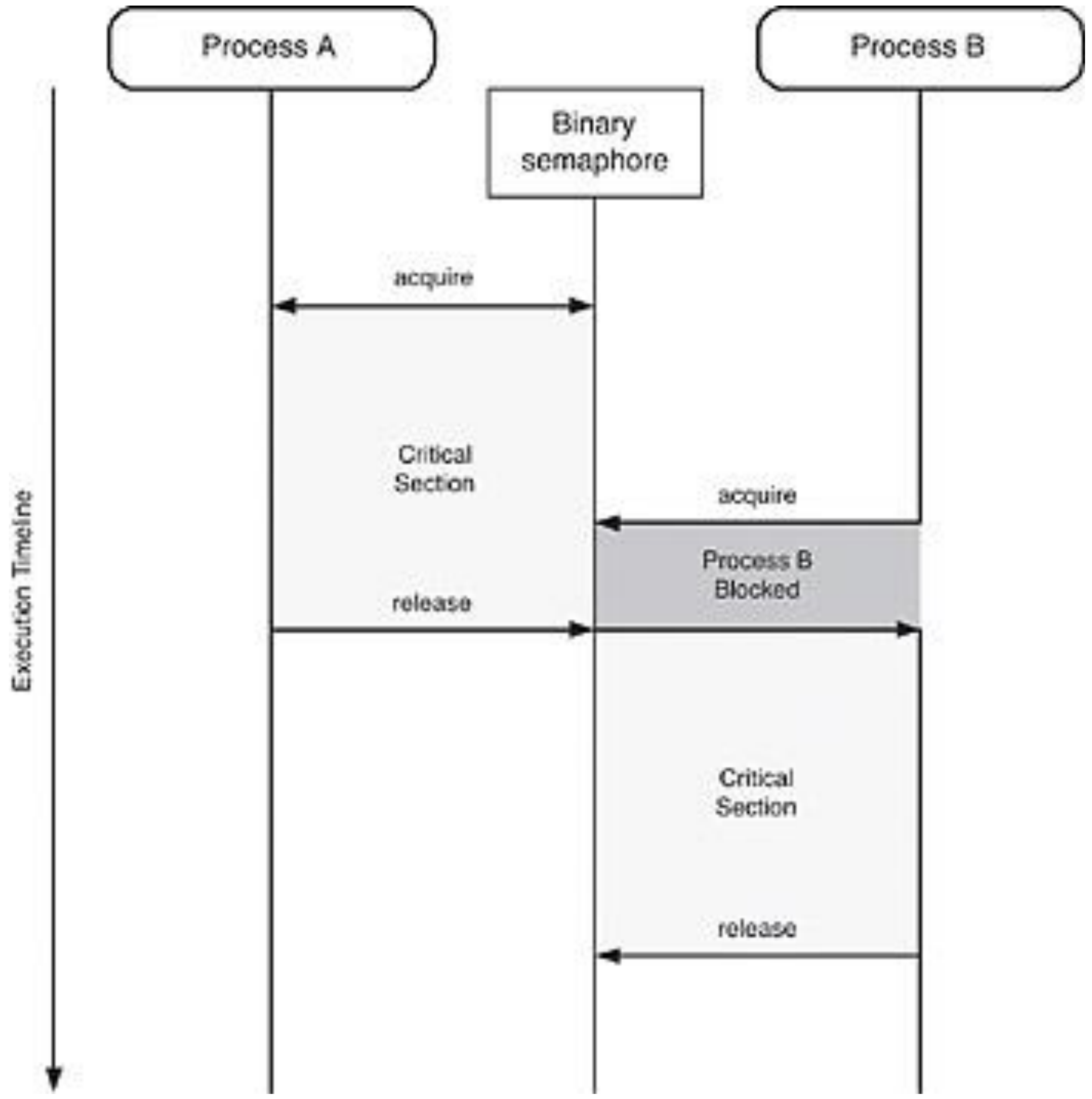
Semaphore (Continued)

- **Mutual Exclusion**
  We can use binary semaphores to deal with the critical-section problem for multiple processes. The n processes share a binary semaphore (mutex lock) initialized to 1.

- Each process P, is organized as shown

```
•  Semaphore S; // initialized to 1
•  wait (S);
          Critical Section
•  signal (S);
```

Semaphore (Continued)



Semaphore (Continued)

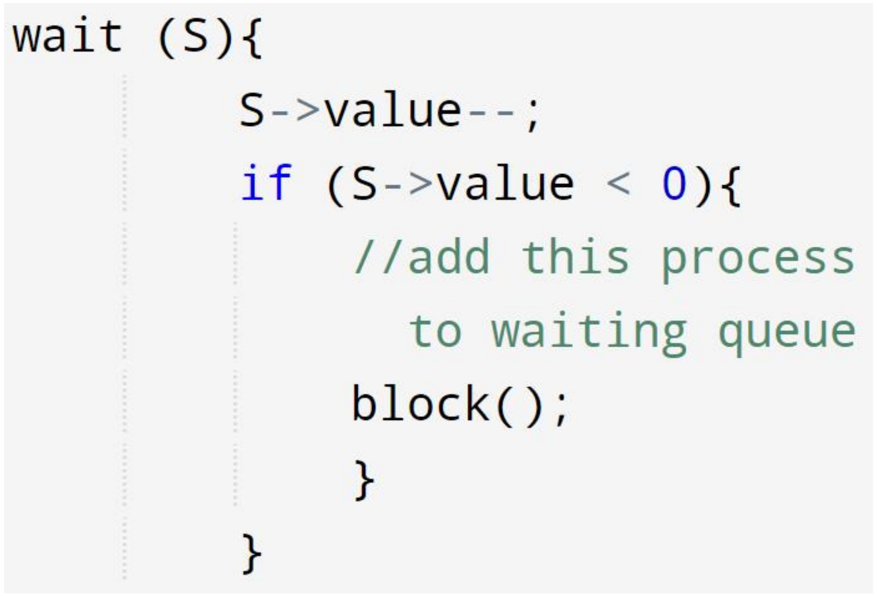
- **Implementation**– Main disadvantage of the semaphore definition given previously is that every process waiting for semaphore loops continuously in the entry code.
- Looping wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **spinlock** (process "spins" while waiting for the lock).
- To eliminate busy waiting when a process has to wait for semaphore it is *blocked*.
- The block operation places a process into a waiting queue associated with the semaphore. Hence a waiting process is not scheduled so no CPU cycles are wasted.

Semaphore (Continued)

➢ **Implementation**– semaphores under this definition is defined as a "C" struct:

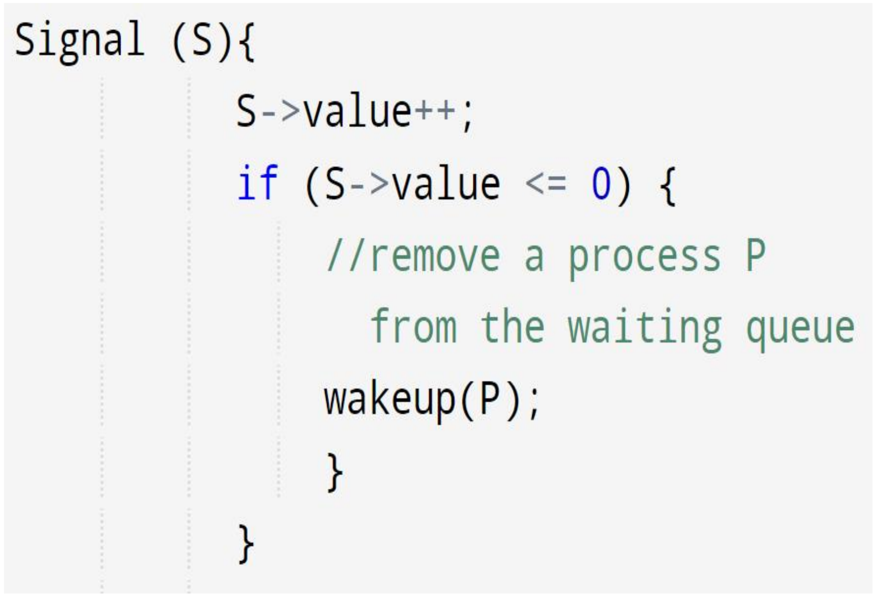
```
typedef struct {
        int value;
        struct process *list;
} semaphore;
```

- **Implementation of wait():**



```
wait (S){
        S->value--;
        if (S->value < 0){
            //add this process
              to waiting queue
            block();
            }
        }
```

- **Implementation of Signal():**



```
Signal (S){
        S->value++;
        if (S->value <= 0) {
            //remove a process P
              from the waiting queue
            wakeup(P);
            }
        }
```

Semaphore (Continued)

- The block() operation suspends the process that invokes it.
- The wakeup(P) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.
- Note that, although under the classical definition of semaphores with busy waiting the semaphore value is never negative, this implementation may have negative semaphore values.
- If the semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.

Deadlock and Starvation

- **Deadlock** – two or or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes

- Let **S** and **Q** be two semaphores initialized to 1

```
        P0                      P1
   wait (S);               wait (Q);
   wait (Q);               wait (S);
        .                       .
        .                       .
        .                       .
 Signal (S);             signal (Q);
 Signal (Q);             signal (S);
```

- **Starvation** – indefinite blocking.   A process may never be removed from the semaphore queue in which it is suspended.