OPERATING SYSTEMS

UNIT 3: CONCURRENCY

Objectives

1. Review concurrency concepts and derived problems
2. Analyze how the OS contributes to solve concurrency problems
3. Analyze the inner workings of some synchronizations and communication mechanisms offered by the OS

Contents

1. **Review of Concurrent Programming concepts**
2. OS role in concurrency management
3. Synchronization mechanisms
4. Communication mechanisms
5. Deadlock

Review of Concurrent Programming concepts

1. What is Concurrency ?
2. Are there any differences between concurrency and parallelism?
3. What is the difference between concurrent independent and concurrent cooperating processes? Examples
4. Can they both need Communication and Synchronization Mechanisms? Examples
5. Why do we need Concurrency? Examples
6. Types of parallelization algorithms
7. Examples of communication and/or synchronization
8. What problems does concurrency present?

Review of Concurrent Programming concepts

Concurrency

- Simultaneous execution of tasks in the system that may have to share resources.

Parallelism: real concurrency

- The process execution is performed on different cores or processors

Apparent concurrency

- Concurrency is simulated. Processes exist simultaneously and their execution time is interleaved.
- A model with a single processor

Review of Concurrent Programming concepts

1. What is Concurrency ?
2. Are there any differences between concurrency and parallelism?
3. **What is the difference between concurrent independent and concurrent cooperating processes? Examples**
4. **Can they both need Communication and Synchronization Mechanisms? Examples**
5. Why do we need Concurrency? Examples
6. Types of parallelization algorithms
7. Examples of communication and/or synchronization
8. What problems does concurrency present?

Review of Concurrent Programming concepts

Types of concurrent processes

- **Independent**
  - Run without requiring the assistance or cooperation of other processes.
  - In general most of the processes are executed independently of one another.
- **Cooperating**
  - Designed to work together in an activity.
  - Must be able to communicate and interact among them.

Review of Concurrent Programming concepts

Interaction among processes (of any type)

- **Share or compete** for access to a physical or logical resource
  - E.g.: Two independent processes may compete for disk access
  - E.g.: Two processes want to change the content of a record in the database
- They **communicate or synchronize** to achieve a common goal
  - E.g.: (a + b) * (a-b) The process that performs the product can not start until addition and subtraction processes have not finished.
  - E.g.: $| V | = \text{sqrt}(v1^2 + v2^2 + \ldots + vN^2)$

Review of Concurrent Programming concepts

1. What is Concurrency ?
2. Are there any differences between concurrency and parallelism?
3. What is the difference between concurrent independent and concurrent cooperating processes? Examples
4. Can they both need Communication and Synchronization Mechanisms? Examples
5. **Why do we need Concurrency? Examples**
6. **Types of parallelization algorithms**
7. Examples of communication and/or synchronization
8. What problems does concurrency present?

Review of Concurrent Programming concepts

Need for concurrence

- Accelerates calculation.
  - Divide the task into parallel subtasks
- It allows interactive use.
  - Multiple users making queries. (e.g. a database)
  - It creates a thread for each one
- Making better use of the machine resources.
  - E.g. CPU

Parallelization of algorithms

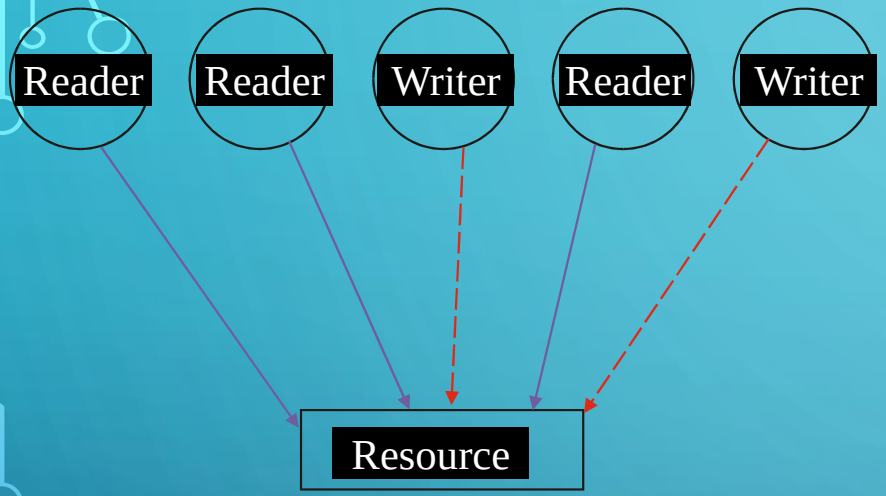
- Parallelization of tasks (independent tasks)
- Data Parallelization
- Hybrid (pipelines)

Review of Concurrent Programming concepts

1. What is Concurrency ?
2. Are there any differences between concurrency and parallelism?
3. What is the difference between concurrent independent and concurrent cooperating processes? Examples
4. Can they both need Communication and Synchronization Mechanisms? Examples
5. Why do we need Concurrency? Examples
6. Types of parallelization algorithms
7. **Examples of communication and/or synchronization**
8. What problems does concurrency present?

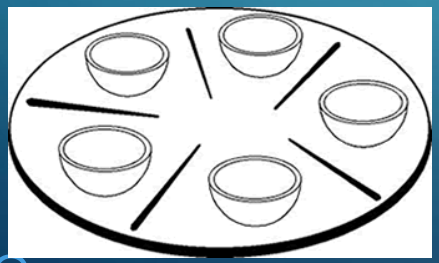
Review of Concurrent Programming concepts

Classical problems



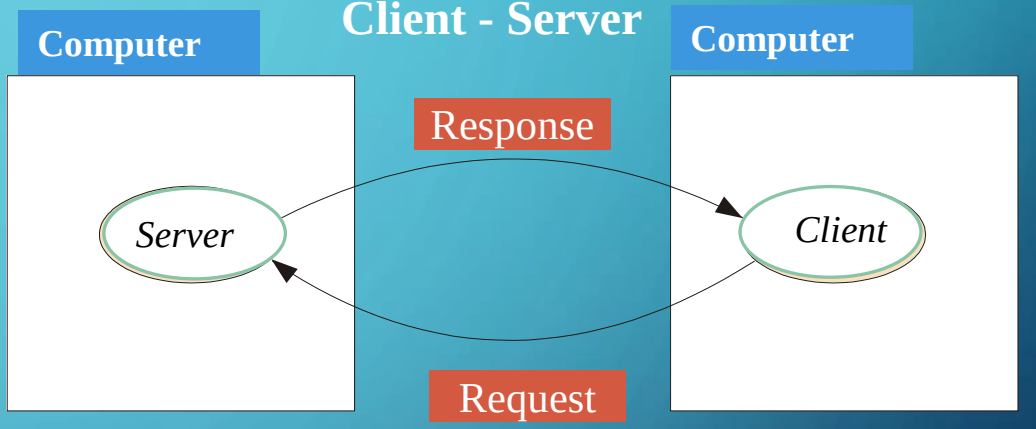
Readers and Writers

Limited resources

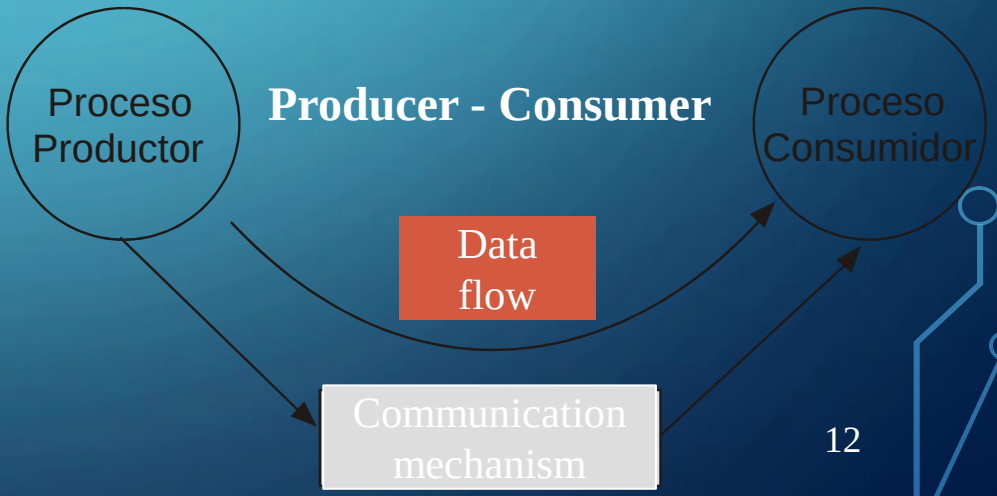


dining philosophers problem

Client - Server



Producer - Consumer



Review of Concurrent Programming concepts

1. What is Concurrency ?
2. Are there any differences between concurrency and parallelism?
3. What is the difference between concurrent independent and concurrent cooperating processes? Examples
4. Can they both need Communication and Synchronization Mechanisms? Examples
5. Why do we need Concurrency? Examples
6. Types of parallelization algorithms
7. Examples of communication and/or synchronization
8. **What problems does concurrency present?**

Review of Concurrent Programming concepts

Concurrency problems

1. ***Race Conditions:*** *The final result of the execution of several concurrent processes depends on the execution order.*

2. ***Deadlock:*** *permanent blockade of several processes competing for resources or that need to synchronize with each other.*

Review of Concurrent Programming concepts

Race conditions

```
LOAD R1, /res
LOAD R2, /total
ADD R1, R2
ST R1, /total
```

```
int total = 0;

int main() {
    thread_create( partial_addition(1, 50) );
    thread_create( partial_addition(51, 100) );

    return total;
}

void partial_addition(int from, int until) {
    int res = 0;
    for(int i = from; i <= until; i++) {
        res += i;
    }
    total = total + res;

    thread_exit(0);
}
```

Review of Concurrent Programming concepts

Race conditions

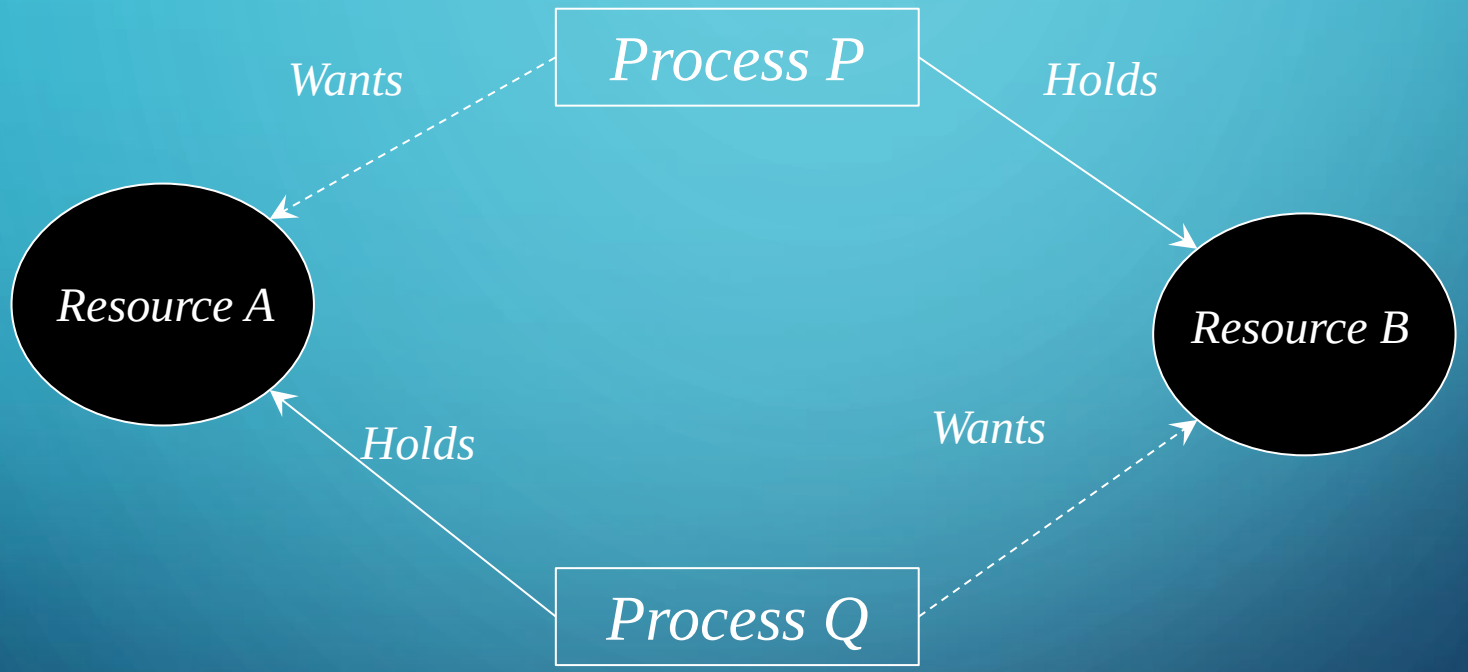
Processor can be switched in between any two instructions

total = 0
sum(1..50) → 1275
sum(51, 100) → 3775

What value has "total" after this execution path ?

Review of Concurrent Programming concepts

Deadlock



Review of Concurrent Programming concepts

Race Condition

- **Example** of race condition
  - Some assignments are not atomic operations (+=, -= ...)
- **Solutions** (race conditions)
  - Executing critical section with **mutual exclusion**

    (atomic execution of hazardous operations)
- **Critical section**
  - Code segment of a program that accesses shared resources with other processes, being exclusive access to resources essential.

Make sure that only one process accesses its critical section → synchronization mechanisms needed

Review of Concurrent Programming concepts

```
int total = 0;

int main() {
    thread_create( partial_addition(1, 50) );
    thread_create( partial_addition(51, 100) );

    return total;
}

void partial_addition(int from, int until) {
    int res = 0;
    for(int i = from; i <= until; i++) {
        res += i;
    }
    total = total + res;

    thread_exit(0);
}
```

Critical section

Review of Concurrent Programming concepts

What synchronization mechanisms do you know?

Are they language mechanisms or operating system mechanisms?

Review of Concurrent Programming concepts

Tools: **Synchronization Mechanisms**

For exclusive access to a critical section

- Lock (.Net) (synchronization between threads)
- Mutex (synchronization between processes or threads)
- Monitors (thread safety)

For restricted access to **N** concurrent processes or threads to a critical section

- Semaphores
- .... Other (Condition variables, critical regions ...)

Review of Concurrent Programming concepts

What synchronization mechanisms do you know?

Are they language mechanisms or operating system mechanisms?

CONTENTS

1. Review of Concurrent Programming concepts
2. **OS role in concurrency management**
3. Synchronization mechanisms
4. Communication mechanisms
5. Deadlock

OS role in concurrency management

How is synchronization included in a program?

1. Using language libraries
2. Using operating system calls to use primitive synchronization systems

In the end, the operating system is the one responsible for providing mechanisms for

1. **Synchronization**
2. **Communication**

OS role in concurrency management

Synchronization mechanisms

- Semaphores
- Mutex
- Conditional variables
- Conditional critical regions (language)
- Monitors (language)
- Signals

Communication mechanisms

- Archives
- Pipes
- Shared memory
- Message passing

Contents

1. Review of Concurrent Programming concepts
2. OS role in concurrency management
3. **Synchronization mechanisms**
4. Communication mechanisms
5. Deadlock

Synchronization Mechanisms: Semaphores

- Devised by Dijkstra in 1965
- Structure with three atomic operations defined:
  - *Initialization, P, and V.*
- All operations are atomic.

*Also used **wait(s)** and **signal(s)** instead of P (s) and V (s) respectively*

***No active standby.** The OS manages a queue of blocked processes*

```
P(s){
   s = s - 1;
   if (s < 0)
      block on s;
}
```

```
V(s){
   s = s + 1;
   if (s <= 0)
        wake up a process blocked on s;
}
```

The operating system implements these operations and offers them as services

Synchronization Mechanisms: Semaphores

Utility

1. **Solution to the critical section on mutual exclusion** *(Initializing S to 1)*

```
S = 1;
```

```
P(s);
   CRITCAL SECTION
V(s);
```

2. **Limiting the number of processes or threads concurrently accessing a critical section** (control units of a resource being used simultaneously) *(Initializing S to the number of resources)*

```
S = 4;
```

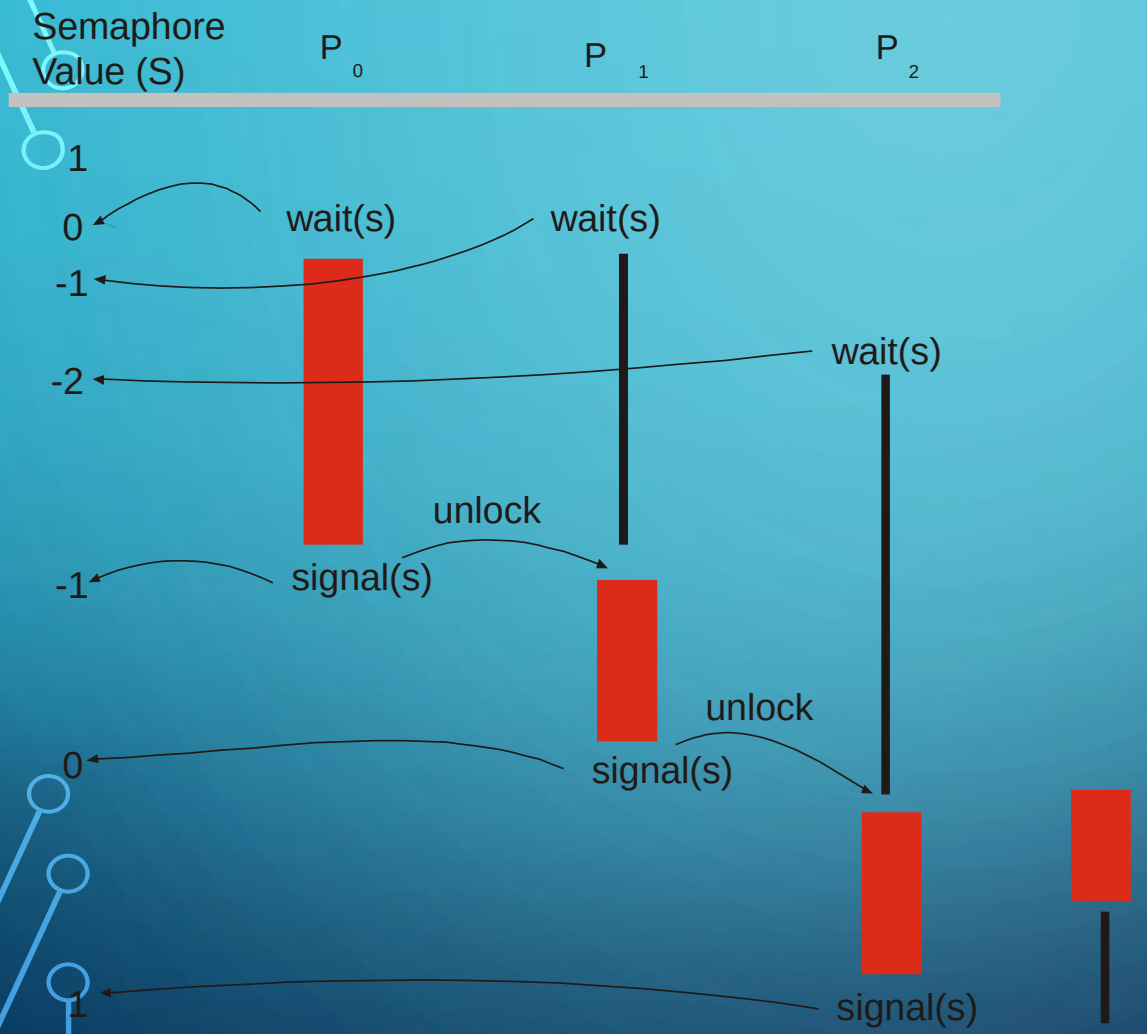
```
P(s);
   resource usage
V(s);
```

3. **Synchronization**

```
S = 0;
```

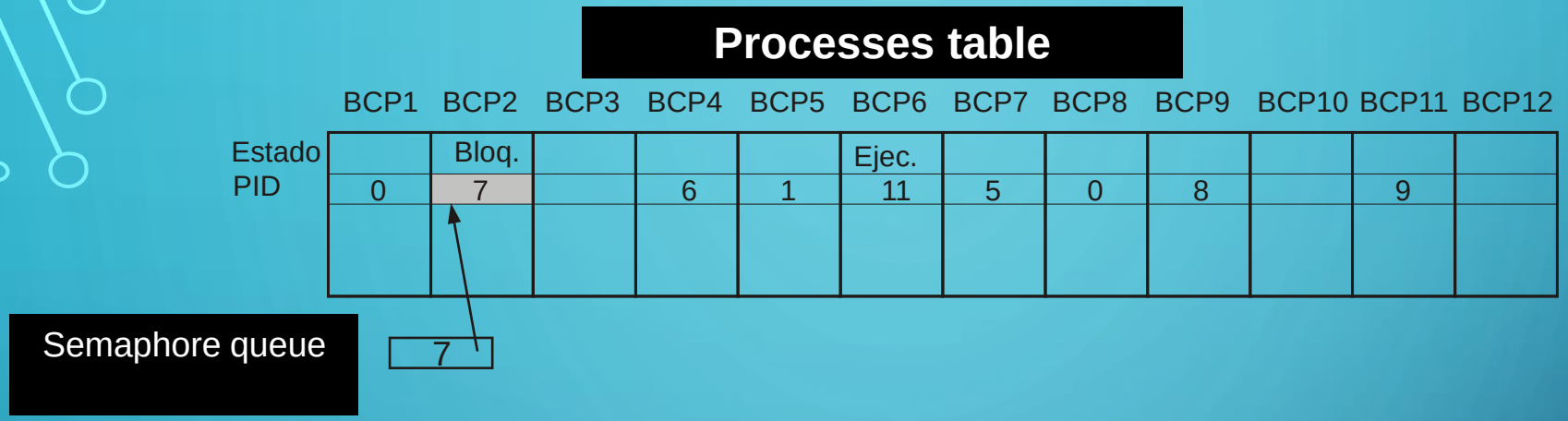
Thread 1	Thread 2
A = x + y	P(s);
V(s);	print(A);

Synchronization Mechanisms: Semaphores

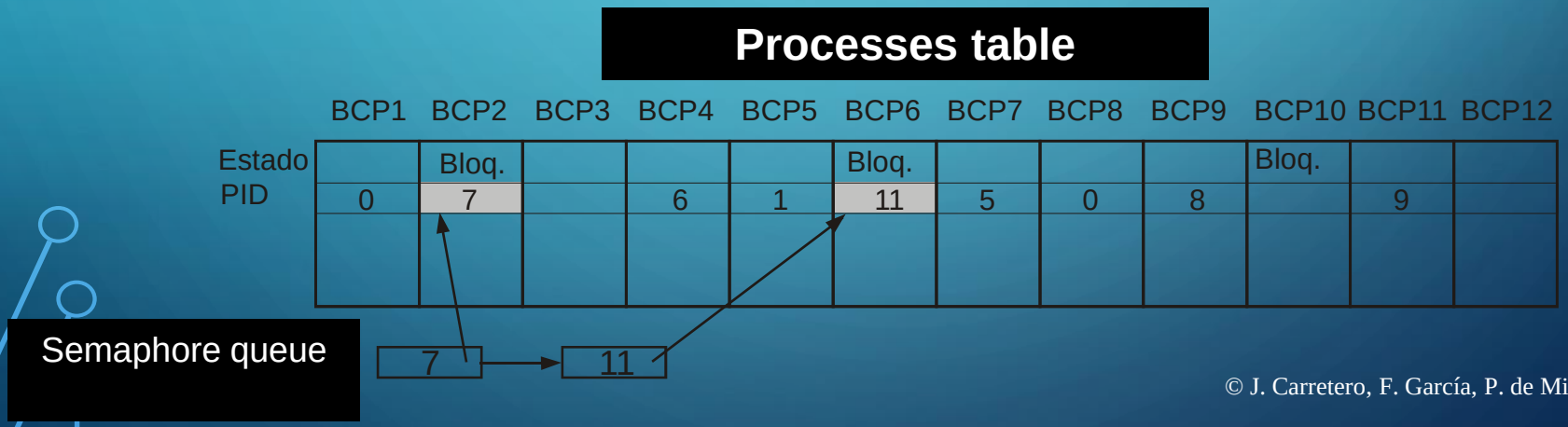


- *Wait(s) is equivalent to P(s)*
- *Signal(s) is equivalent to V(s)*

Synchronization Mechanisms: Semaphores



(a)



(b)

Synchronization Mechanisms: Mutex

- Special case of a semaphore
  - Initialized with value 1
- Only used from mutual exclusion on critical sections
- Operations:
  - Lock() (like wait() on sem.)
  - Unlock() (equiv. to signal())
- Lock(), unlock should be done by the same process

```
lock(m);
  <critical section>
unlock(m);
```

```
int total = 0;
Mutex m;

int main() {


void partial_addition(int from, int until) {
    int res = 0;
    for(int i = from; i <= until; i++) {
        res += i;
    }

    lock(m);
    total = total + res;
    unlock(m);
}
```

Conditional Variables

- Synchronization variable associated with a mutex
  - Used when in a critical section we need to wait for some other condition
- It blocks the thread until the condition is meet
  - When is blocked then the mutex is released
  - When the condition becomes TRUE the dormant threads are awaken.
    - They compete again for the mutex

- Operations:
  - Lock(), unlock() for the mutex,
  - c_wait(<variable>, <mutex>)
  - c_signal(<variable>)

Synchronization Mechanisms: Conditional Variables

```
lock(m);
   <critical section>
   while condition == FALSE {
        c_wait(c, m)
   }
  <rest of critical section>
unlock(m);
```

```
lock(m);
   <critical section>
   condition = TRUE
   c_signal(c)
unlock(m);
```

Synchronization Mechanisms: Conditional Variables

```
producer() {
    int pos = 0;

    for(;;) {
        lock(mutex);
        while( n_elements == BUFFER_SIZE) {
            c_wait(full, mutex);
        }
        buffer[ pos ] = datum;
        pos = (pos + 1) % BUFFER_SIZE;
        n_elements++;
        if (n_elements == 1) {
            c_signal( empty);
        }
        unlock(mutex);
    }
}
```

```
consumer() {
    int pos = 0;

    for(;;) {
        lock(mutex);
        while( n_elements == 0) {
            c_wait( empty, mutex );
        }
        datum = buffer[ pos ];
        pos = (pos + 1) % BUFFER_SIZE;
        n_elements--;
        if (n_elements == BUFFER_SIZE - 1) {
            c_signal( full );
        }
        unlock(mutex);

        <use datum>
    }
}
```

Synchronization Mechanisms: Signals

Sending signals between processes

- A process may get locked (sleep) until it receives a signal (***pause*** *call*).
- A process can awake another sending a signal (***kill*** call).

Considerations about signals

- A process may receive signals even if it not is waiting for them (are asynchronous).
- Signals are not queued. If a signal is pending and another of the same type is received only one registered.
- Signals may cause race conditions.

Synchronization Mechanisms: Signals

```
int R, S, M;
int a, b, c;

...

int pchildid = fork();
if (pchildid == 0) {
    R = a + b;
    kill( getppid(), SIGUSR1 );
    pause( SIGUSR2 );
    M = S + d
}
else {
    pause( SIGUSR1 );
    S = R + c
    kill(pchildid, SIGUSR2);
}
```

Synchronization Mechanisms: Signals

```
function(...) {

    ...
    ...
    kill( pid, SIGUSR1 );
    ...
    ...

}
```

```
void handler(int signal) {
    <signal handling>
}

run() {

    ...
    signal(SIGUSR1, handler)
    ...
    ...
    ...
    ...
    ...
    ...
    ...

}
```

```
void handler(int signal) {
    <signal handling>
}
```

Synchronization Mechanisms: Semaphores

TYPE	POSIX	WIN32	ACTION
Mutex	pthread_mutex_destroy	CloseHandle (mutex handle)	Mutex destruction
Mutex	pthread_mutex_init	CreateMutex	Mutex initialization
Mutex	pthread_mutex_unlock	ReleaseMutex	Unlock a mutex
Mutex	pthread_mutex_lock	WaitForSingleObject (mutex handle)	Lock a mutex
Semaphores	sem_open	CreateSemaphore	Creates a name semaphore
Semaphores	sem_init		Inits a no name semaphore
Semaphores	sem_open	OpenSemaphore	Open a named semaphore
Semaphores	sem_close	CloseHandle	Close a semaphore
Semaphores	sem_post	ReleaseSemaphore	Signal on a semaphore
Semaphores	sem_wait	WaitForSingleObject	Wait on a semaphore

Synchronization Mechanisms: Conditional Variables

TYPE	POSIX	WIN32	ACTION
Synchronization	pthread_cond_destroy	CloseHandle	Destroys a conditional variable in POSIX and a Win32 event when no longer referenced
Synchronization	pthread_cond_init	CreateEvent	Start a condition variable and an event
Synchronization	pthread_cond_broadcast	PulseEvent	Wake a lightweight processes blocked on a condition variable or event
Synchronization	pthread_cond_signal	SetEvent	Wake up a lightweight process blocked on a condition variable or event
Synchronization	pthread_cond_wait	WaitForSingleObject	Blocks to a process on a condition variable or event

Synchronization Mechanisms: Signals

TYPE	POSIX	Win32	Comments
Signals	pause		Suspends process until signal reception
Signals	kill		Sends a signal
Signals	sigemptyset, sigfillset, sigaddset, sigdelset, sigismember		Handling of signal sets
Signals	sigprocmask		Checks or changes the signal mask
Signals	sigpending		Gets not yet delivered signals
Signals	sigaction		Detailed Management of Signals
Signals	sigsetjmp, siglongjmp		Perform non-local jumps
Signals	sigsuspend		Specifies mask and suspends process until signal

Contents

1. Review of Concurrent Programming concepts
2. OS role in concurrency management
3. Synchronization mechanisms
4. **Communication mechanisms**
5. Deadlock

Communication mechanisms

Synchronization mechanisms

- Semaphores
- Mutex
- Conditional variables
- Conditional critical regions (language)
- Monitors (language)
- Signals

Communication mechanisms

- Archives
- Pipes
- Shared memory
- Message passing

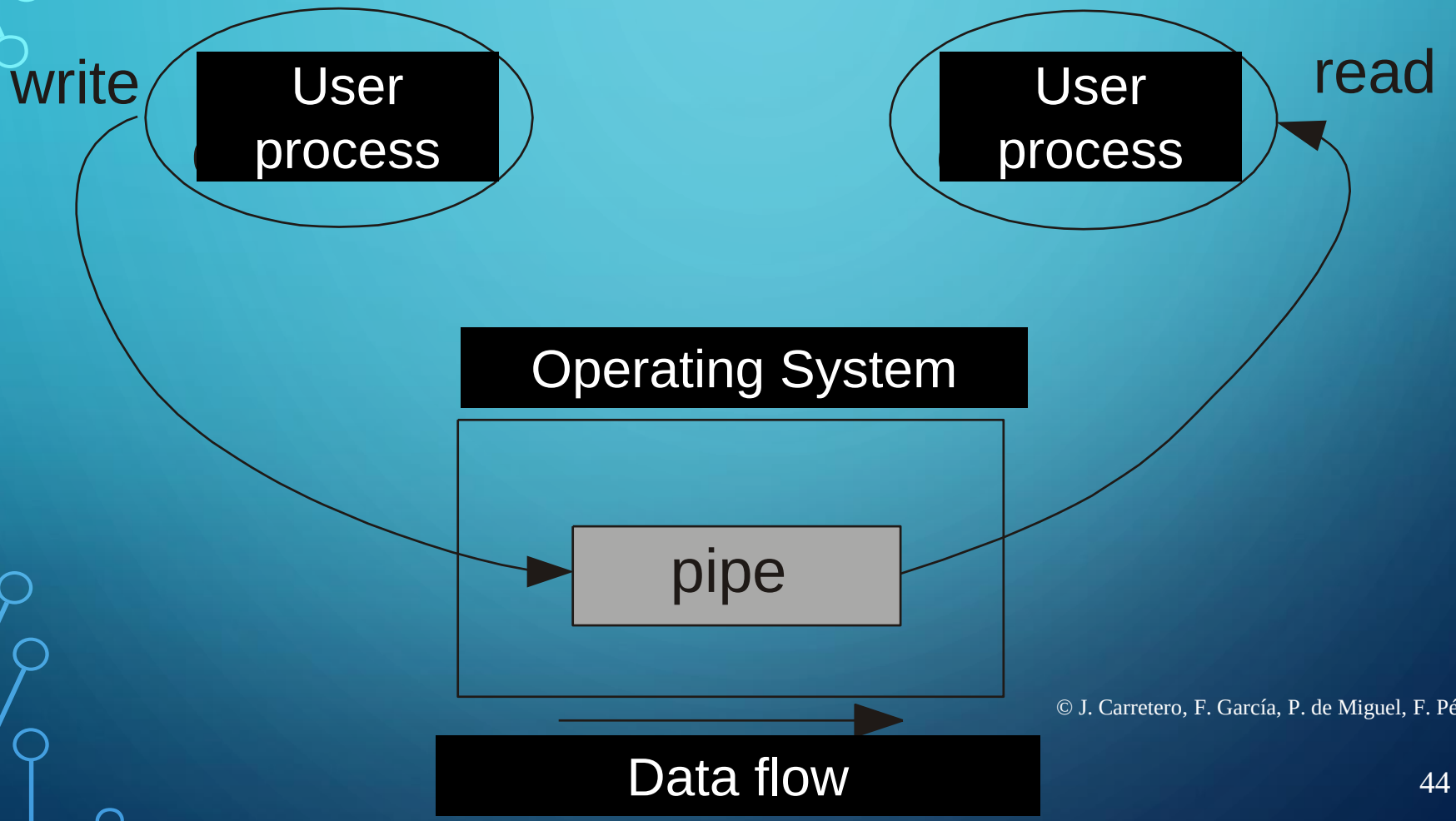
Files

- Files can be used for data sharing by several processes.
- They are easy to use. Communication based on read and write operations.
- Can communicate a potentially unlimited number of processes.
- It is inefficient, being slow operations.
- **You need a synchronization mechanism to order file access.**

```
fd file;
char buffer[128];

file = fopen("r+", "file.dat");
if (fork() != 0){
   buffer= "hola que tal";
   write(file, buffer);
}
else{
   read(file, buffer);
   println(buffer);
}
```

Pipes



Pipes

- Used in POSIX both as a communication and synchronization mechanism
- You use the same calls as with files to read and write. Both operations are atomic.
- It is a FIFO file: when reading data, you read the oldest in the pipe and then this data is removed.
- If a process tries to read from an empty pipe, the OS sleeps the process until data is available.
- If the pipe is full and you try to write, the process will sleep until there is room.
- It can be used by multiple processes, either reading or writing.

Pipes

```
int fildes[2];
char buffer[1000];

pipe(fildes); // opens the pipe
// fildes[0] for reading
// fildes[1] for writing
if (fork() != 0){
    bufer = "hola que tal";
    write(fildes[1], buffer, 12);
}
else{
    read(fildes[0], buffer, 12);
    println(buffer);
}
```

Communication mechanisms
Message passing

- To communicate processes on different machines
- Processes send and receive messages to communicate and synchronize.
- **Sockets is an implementation widely used**. Most commonly used in distributed applications
- It uses two basic operations:
  - send (destination, message)
  - receive (source, message

Message passing

- **Types of Communication:**
  - **Synchronous** (blocking sending and receiving): sender sleeps until the recipient receives the message. The receiver sleeps if the message has not arrived.
  - **Synchronous intermediate** (non-blocking sending and blocking receiving) the sender does not sleep but the receiver is blocked until it receives the message.
  - **Asynchronous** (non-blocking send and receive): Nobody waits.

Communication mechanisms: Message passing

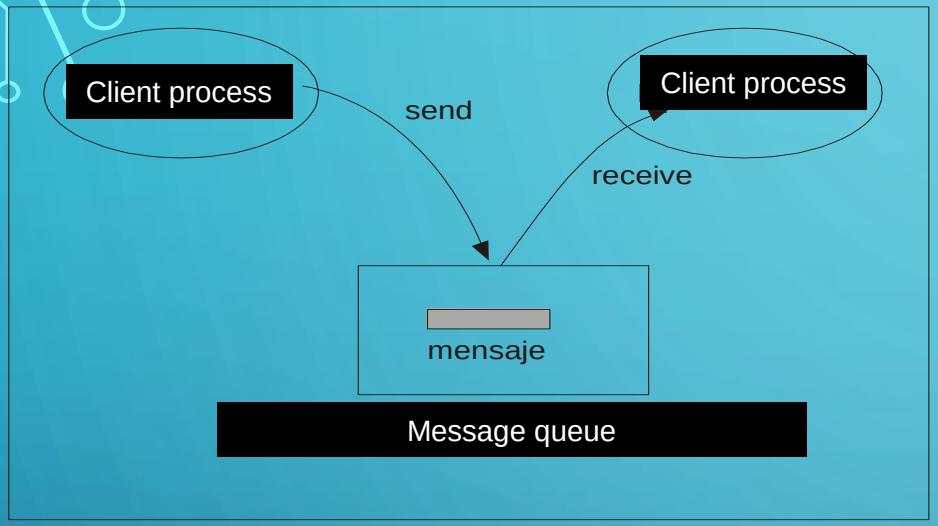
Communication modes

1. Direct communication Message sending to the process
2. Indirect communication Message sending to an intermediate structure from which the process takes it up
   - Message queues or mailboxes
   - Ports

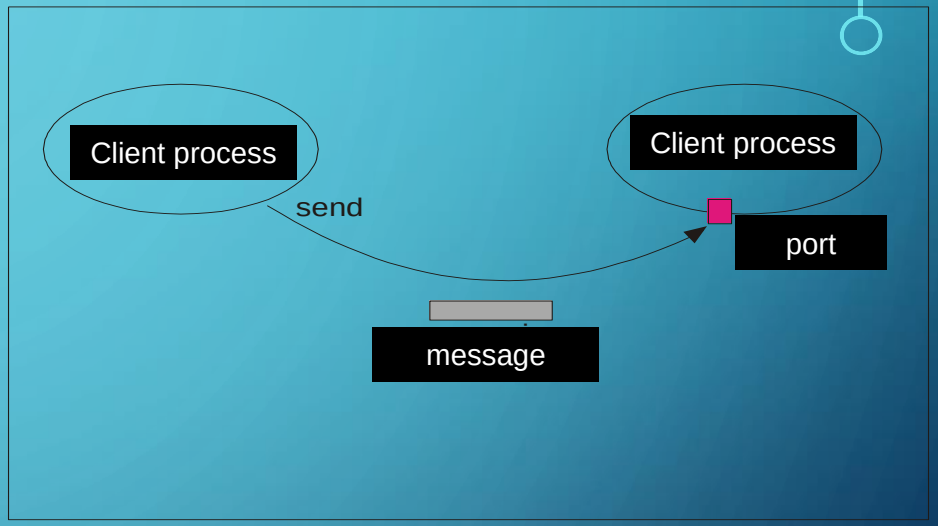
Message queue. The source and destination identifies an intermediate entity: a mailbox. There may be multiple senders and multiple receivers.

Ports. Specific case of a mailbox, where there is only one receiver, there may be multiple senders. The port typically belongs to the receiving process.
Sockets are an example of this.

Communication mechanisms: Message passing

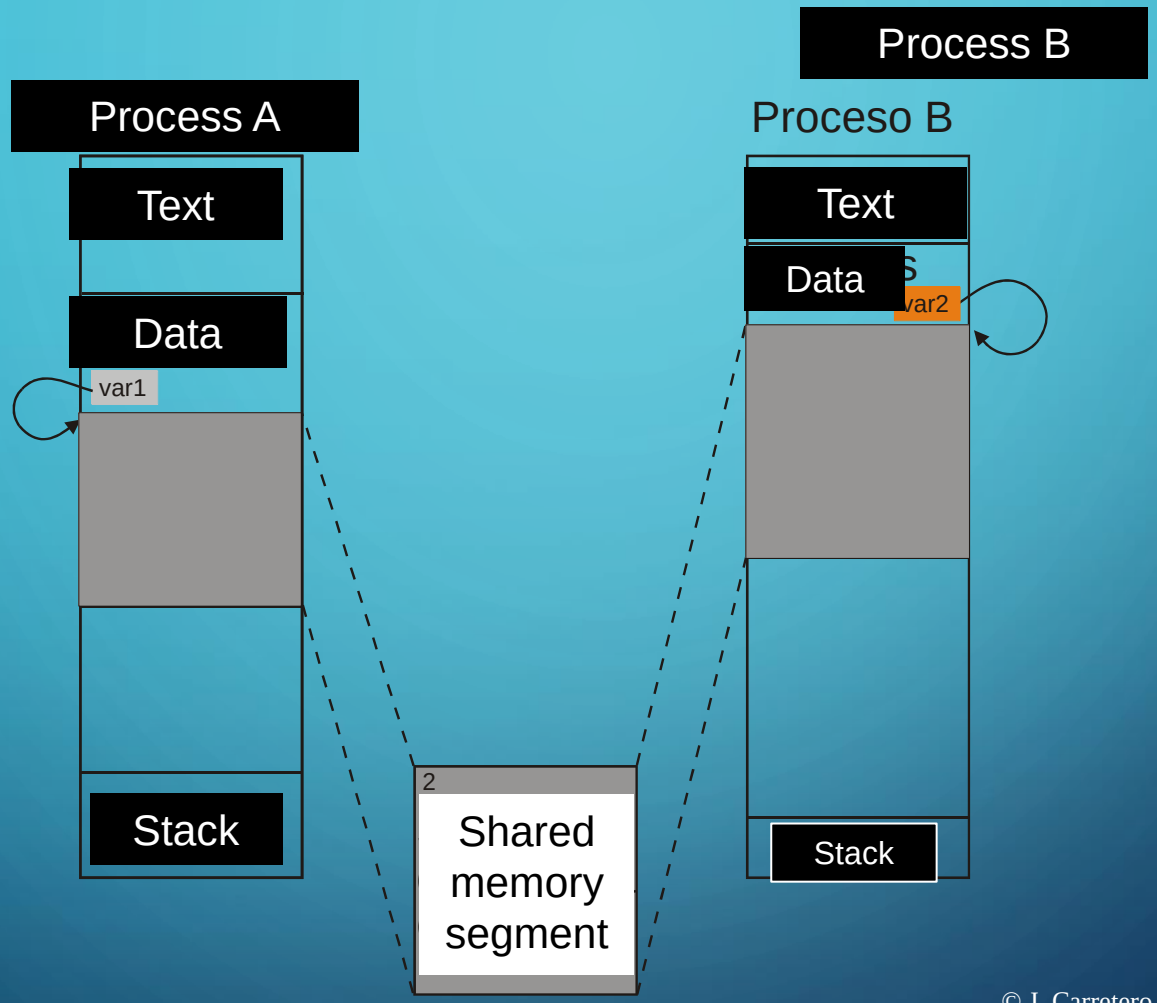


Message queue communication



Port communication

Communication mechanisms
Shared Memory



Shared Memory

- Shared memory is used to communicate (not to synchronize) processes within the same machine.
- The operating system allows multiple processes to access the same memory via specific system calls for shared memory creation.
- Processes can use this memory area to leave data that must be accessible by all.
- In a system based on threads all threads in the same process share memory without OS intervention.

Communication mechanisms

TYPE	POSIX	WIN32	ACTION
Communication	close	CloseHandle	Closes a pipe
Communication	mq_open	CreateFile	Opens a message queue in POSIX and mailslot in Win32
Communication	mq_open	CreateMailslot	Creates a message queue in POSIX and Win32 mailslot
Communication	mq_close	CloseHandle	Closes a message queue in POSIX and Win32 mailslot
Communication	mq_send	WriteFile	Sends data to a message queue in POSIX or to a Win32 mailslot
Communication	mq_receive	ReadFile	Receives data from a message queue in POSIX or a mailslot in Win32

Communication mechanisms: Shared Memory

TYPE	POSIX	WIN32	ACTION
Communication	mq_unlink	CloseHandle	Deletes a message queue in POSIX and Win32 mailslot when no longer referenced
Communication	mq_getattr	GetMailslotInfo	Gets attributes of a message queue in POSIX and Win32 mailslot
Communication	mq_setattr	SetMailSlotInfo	Sets the attributes of a message queue in POSIX and Win32 mailslot
Communication	mkfifo	CreateNamedPipe	Creates a named pipe
Communication	pipe	CreatePipe	Creates an unnamed pipe
Communication	dup, dup2 , fcntl	DuplicateHandle	Duplicates a file handle
Communication	read (tubería)	ReadFile	Reads data from a pipe
Communication	write (tubería)	WriteFile	Writes data to a pipe
Communication	close	CloseHandle	Closes a pipe

Contents

1. Review of Concurrent Programming concepts
2. OS role in concurrency management
3. Synchronization mechanisms
4. Communication mechanisms
5. **Deadlock**

Deadlock

Concurrency problems:

1. **Race Conditions:** The final result of the execution of several concurrent processes depends on the execution order.

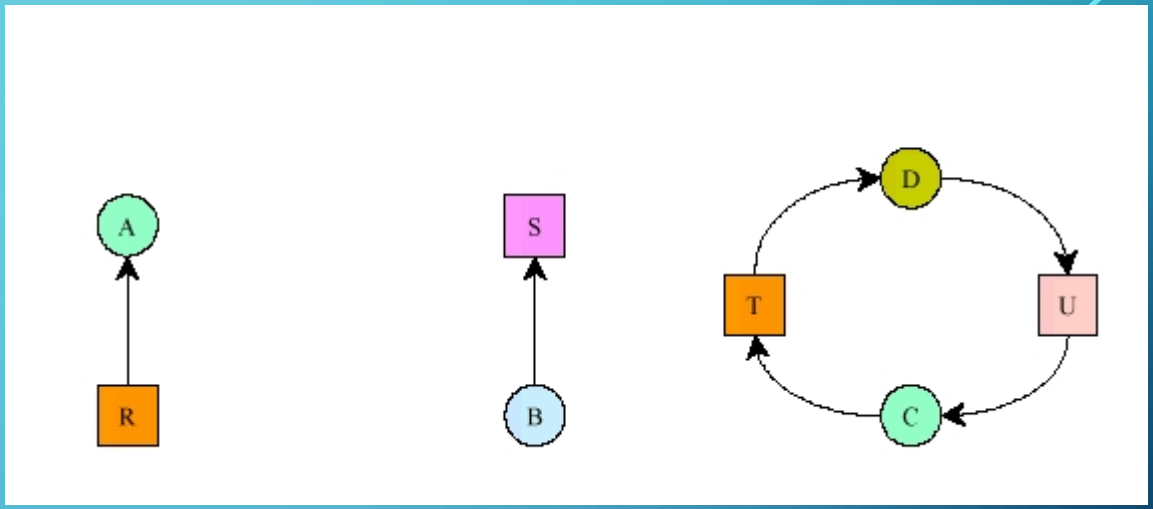
2. **Deadlock:** permanent blockade of several processes competing for resources or that need to synchronize with each other.

We need proper use of synchronization mechanisms

Deadlock

Deadlock

It is an anomaly that occurs when a process is waiting for an event that cannot happen.



- The root cause is the exclusive use of resources shared by multiple processes. There are usually sets of processes or threads involved.
- Not to be confused with *starvation* or *indefinite postponement*. This problem occurs due to poor resource allocation policy.

Deadlock examples

With just one process

The process is waiting for an event that will not happen. For example, *sleep until 12:00 January 1, 2000*.

With two processes

Process 1	Process 2
Request DVD recorder	Request printer
Request printer	Request DVD recorder
USE	USE
Release printer	Release DVD recorder
Release DVD recorder	Release printer

Deadlock examples

With two processes. Consider a system with 200 KB of memory

Process 1	Process 2
Request 80 KB	Request 70 KB
…	…
Request 60 KB	Request 80 KB
…	…
Release 140 KB	Release 150 KB

Both will be blocked if the first request of each process has been attended.

Deadlock examples

Can the philosophers' problem run into deadlock?

```
Philosopher(k) {
   repeat  {
      P(chopstick[k]);
      P(chopstick[k +1]% 5);

      eat();

      V(chopstick[k]);
      V(chopstick[k +1]);

      think();
    }
}
```

```
Philosopher(k) {
   repeat {
      P (table);
      P (chopstick[k]);
      P (chopstick[k +1]% 5);

      eat();

      V (chopstick[k]);
      V (chopstick[k +1]);
      V (table);

      think();
    }
}
```

What initial value must take the table semaphore to avoid Deadlock?

Deadlock conditions

4 conditions for deadlock

1. **Mutual exclusion**

   Processes require exclusive control of resources

2. **Hold and wait**

   Processes hold resources already allocated while waiting for additional resources

3. **No appropriation**

   Resources cannot be taken from the holder processes until they have finished using them

4. **Circular wait**

   There is a circular chain of processes in which each process has one or more resources needed by the next process in the chain

Avoiding Deadlock

1. **Prevention**

   It eliminates any possibility of occurrence

2. **Avoidance (prediction)**

   Deadlock is avoided when it is about to happen

3. **Detection**

   Deadlock allowed and detected

4. **Recovery**

   Deadlock allowed, it is automatically detected and recovered

Deadlock Prevention

- Deadlock prevention works by preventing one of the four conditions from occurring.

- 3 strategies
  - Denial of hold and wait
  - Denial of non-appropriation
  - Denial of circular wait

Deadlock Prevention
Denial of Hold and Wait

- Avoid waiting for additional resources
- The process must **request all resources at once.** It is either all or none.
- Disadvantages
  - Resources underutilization

Deadlock Prevention
Denial of Circular Wait

- Avoid circular waits
  - Resources are numbered
  - Processes must **request resources in linear order**.
    - A process requiring three resources must request them in numbering order.
- Disadvantages
  - Resources are picked before they are needed. Underutilization.

Deadlock Prevention
Denial of non-appropiation

- It is allowed to snatch resources from processes
- If a process, which already holds some resources, requests another one which is not given because it is being used by another process, then it **should release all held resources** and request all of them later.
- Disadvantages
  - Loss of work
  - Possibility of indefinite postponement

Deadlock Avoidance

- Does not prevent the 4 deadlock conditions
- Deadlock is avoided with adequate resource allocation
- Using the Banker's algorithm for avoidance

Banker's Algorithm

- Banker → Operating system
- Capital → Resources
- Customers → Processes
- Loan → Resource allocation
- Return → Resource release
- Bankruptcy → Deadlock

Deadlock Avoidance

- The processes that get all resources they need will eventually finish, and then they may return all the resources they requested.
- If a process is blocked waiting for a resource, it will never return the resources assigned.
- **Algorithm**
  - **The banker lends money only if he is sure he will be able to get his capital back.**
  - **When a process requests a resource, it is only granted if that allocation leads to a safe state**
- **Safe State**
  - The system is in a safe state if it knows that all processes can be completed in a finite time (can meet all their requests).

Deadlock Avoidance

Algorithm for safe status checking

- Repeat until there is no processes with pending requests
  - Find a process that may be assigned all pending resources
  - Add all assigned resources to free resources and then delete the process
- If eventually all processes can be deleted the state is safe

- Disadvantages
  - Need to know a priori the needs of each process
  - Overload

Deadlock Detection and Recovery

- It periodically checks the existence of deadlock by getting the processes and resources involved
- It determines if there are circular wait using **resource allocation graphs**
- It tries to reduce the graphs removing arrows between processes and resources
- If a graph can be reduced for all processes the system presents no Deadlock
- The reduction order is indifferent

- Once detected, it alerts the administrator to take the proper actions.

Deadlock Detection and Recovery

- Once a Deadlock is detected, the system will automatically try to recover the resources
- This requires removing one of the four conditions for deadlock.
- Possibilities
  - Kill processes
  - Suspend processes
- Problems
  - Decide which process to kill or suspend (priority, arbitrary, etc.)
  - Loss of work
  - Need checkpoints / restart (suspended)

Examples

Consider a system with five processes and three types of resources. The current system state is shown below. **It is in a safe state?**

Allocated

	A	B	C
p1	0	1	0
p2	2	0	0
p3	3	0	2
p4	2	1	1
p5	0	0	2

Maximum

	A	B	C
p1	7	5	3
p2	3	2	2
p3	9	0	2
p4	2	2	2
p5	4	3	3

Available

A	B	C
3	3	2

Examples

1.- We calculate the outstanding needs, (Max - Allocation)

Allocated

	A	B	C
p1	0	1	0
p2	2	0	0
p3	3	0	2
p4	2	1	1
p5	0	0	2

Maximum

	A	B	C
p1	7	5	3
p2	3	2	2
p3	9	0	2
p4	2	2	2
p5	4	3	3

Pending

	A	B	C
p1	7	4	3
p2	1	2	2
p3	6	0	0
p4	0	1	1
p5	4	3	1

Available

A	B	C
3	3	2

2.- We look for a safe sequence :<p4, p2, p5, p1, p3>