Fundamentos de los Sistemas Operativos (FSO)

Departamento de Informática de Sistemas y Computadoras (DISCA)
Universitat Politècnica de València

Part 2: Process Management

Unit 5
Execution threads

fSO

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

DISCA

- **Goals**
  - Understanding **concurrent programming** concept
  - Being aware of the **difference** between **execution thread** and **process**
  - Understanding **execution thread** implementation models
  - Knowing the problems related to **sharing memory** (global variables) in multithreaded applications

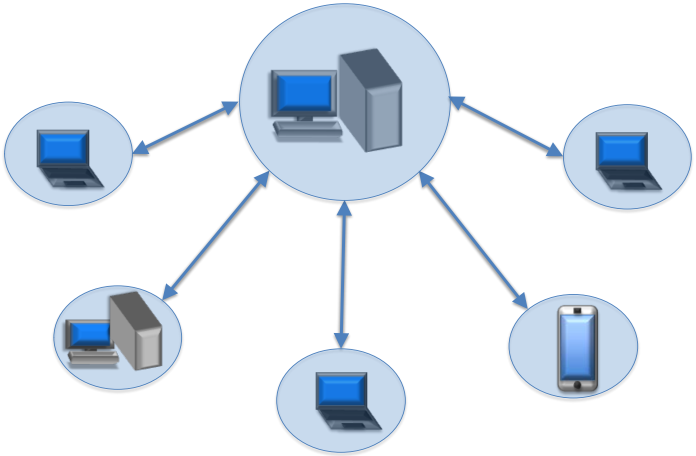
- **Bibliography**
  - "Operating system concepts" Silberschatz 9th Edition, Chapter 4
  - "Sistemas operativos: una visión aplicada" Carretero 2nd Edition

Content

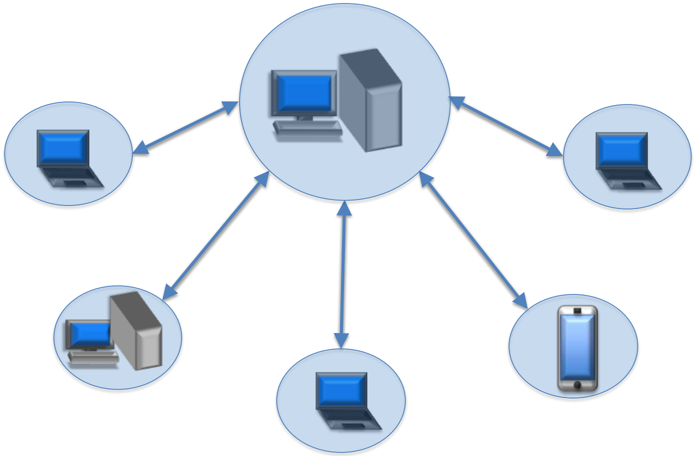
- **Concurrent programming**
- Execution thread concept
- Execution thread models
- Synchronization requirement
- Race condition concept

Concurrent programming

- **Concurrent programming definition:**
  - A single program that solves a problem by using several simultaneous activities (real life is commonly concurrent)
  - If activities REALLY operate simultanously then there is parallelism and completion time can be reduced
- **Examples**:
  - **Web servers** process several HTTP demands concurrently in order to minimize server input waiting time
  - **Multiplayer computer games** where every player is managed by an independent activity



- “Activities” in a concurrent program
  - Work together
    - » They inter-communicate to interchange data (shareing memory and/or using message passing)
    - » Their flow control time lines are synchronized
- There are two choices to implement “activies” in a computer:
  - **Processes**
  - **Execution threads**



Content

- Concurrent programming
- **Execution thread concept**
- Execution thread models
- Synchronization requirement
- Race condition concept

Execution thread concept

- A process is an abstract entity composed by:

Process

Resouce assignmet unit

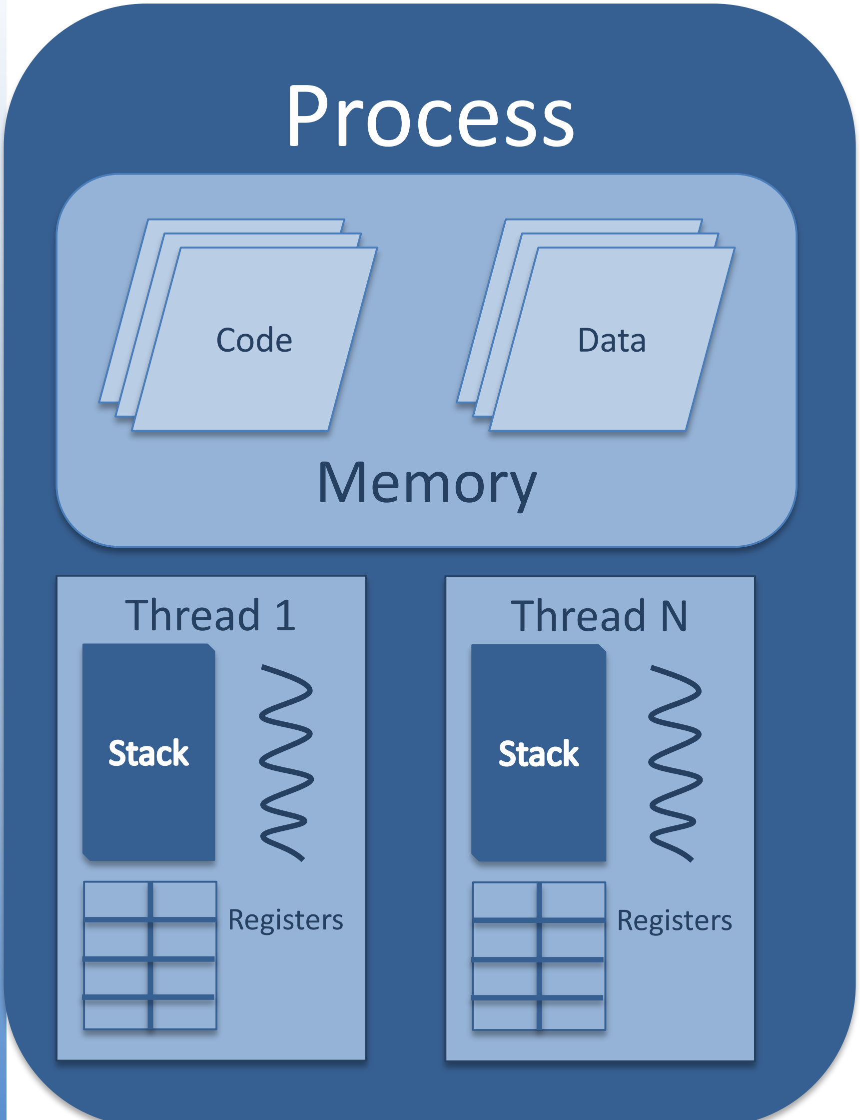
- Memory space
- Open files
- Software resources
- etc

CPU assignment unit

- Stack
- CPU context
- Execution thread ID

- The SO can separate this assignment units inside a process

Execution thread concept



- **Execution thread:** Basic unit for CPU assignment
- **Process** = Resource assignment unit with at least one execution thread
- Execution threads defined **inside a process** share:
  - Code
  - Data
  - Process assigned resources
- Every thread has its **own attributes**:
  - Thread ID
  - Stack
  - Program counter
  - CPU registers

Execution thread concept

- Implementation

TCB (Thread Control Block)

Identification
- Thread ID

Context
- Program counter
- Stack pointer
- Data registers
- Flags register
- Condition codes

Control
- State
- Event
- Scheduling info

- **Attributes**
  - Threads have few attributes
    - TCB is much more compact than PCB
  - Threads are lighter than processes
    - Shared resources info is stored in process PCB

- **Threads vs processes**
  - From the **system point of view**
    - It costs less…
      - » To create a thread inside an existing process than to create a new process
      - » To finish a thread than a process
      - » To switch context between two threads inside a process than between two processes
  - From the **programmer point of view**
    - Multithread programming gives an **easier concurrent programming model** in which communication is:
      - » **More natural**. All threads inside a process share global variables and files
      - » **More efficient**. Many times it is not requiered to ask for kernel services

Content

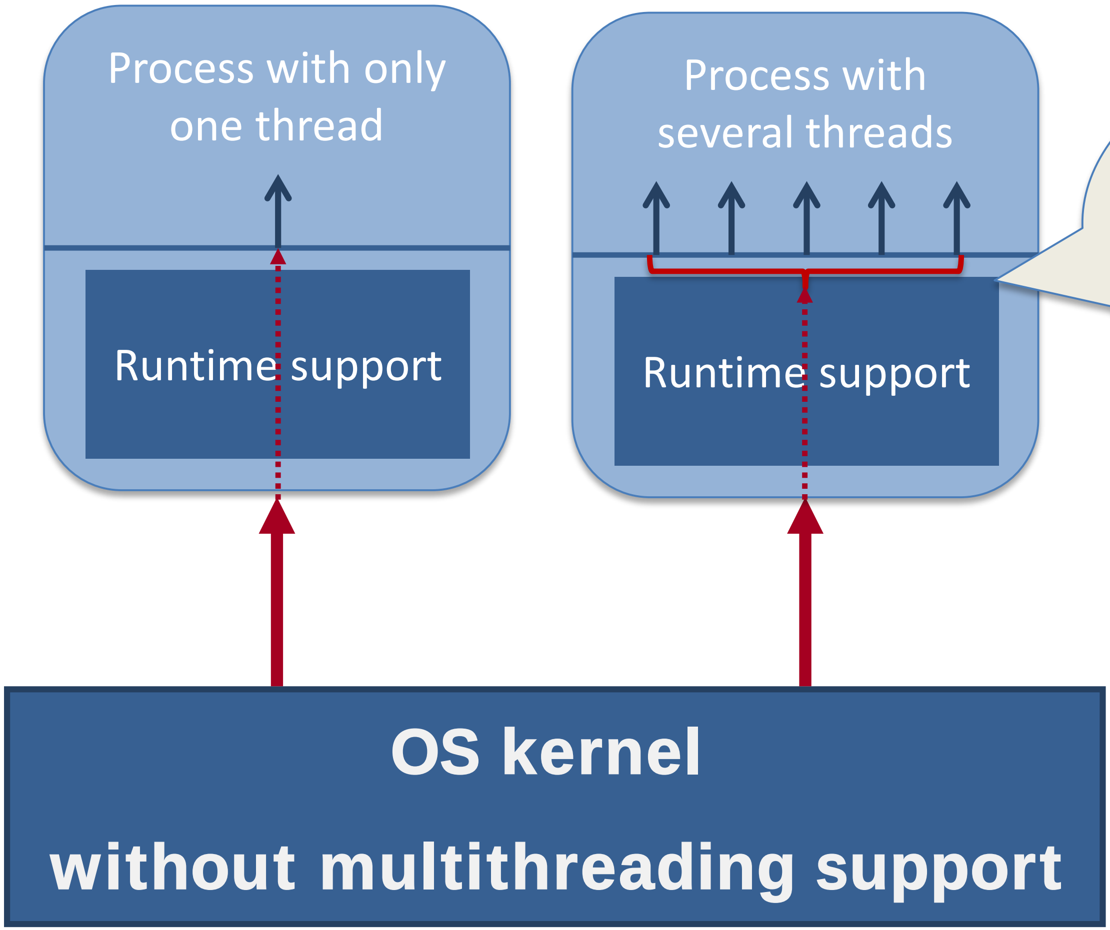
- Concurrent programming
- Execution thread concept
- **Execution thread models**
- Synchronization requirement
- Race condition concept

Execution thread models

- Multithread programming requires **support** the **execution thread abstraction** based on:
  - Data structures with threads attributes (TCBs)
  - Inter thread communication and synchronization operations
- Depending where this support is provided there are three multithread programming models:
  - User level
  - Kernel level
  - Hybrid

Execution thread models

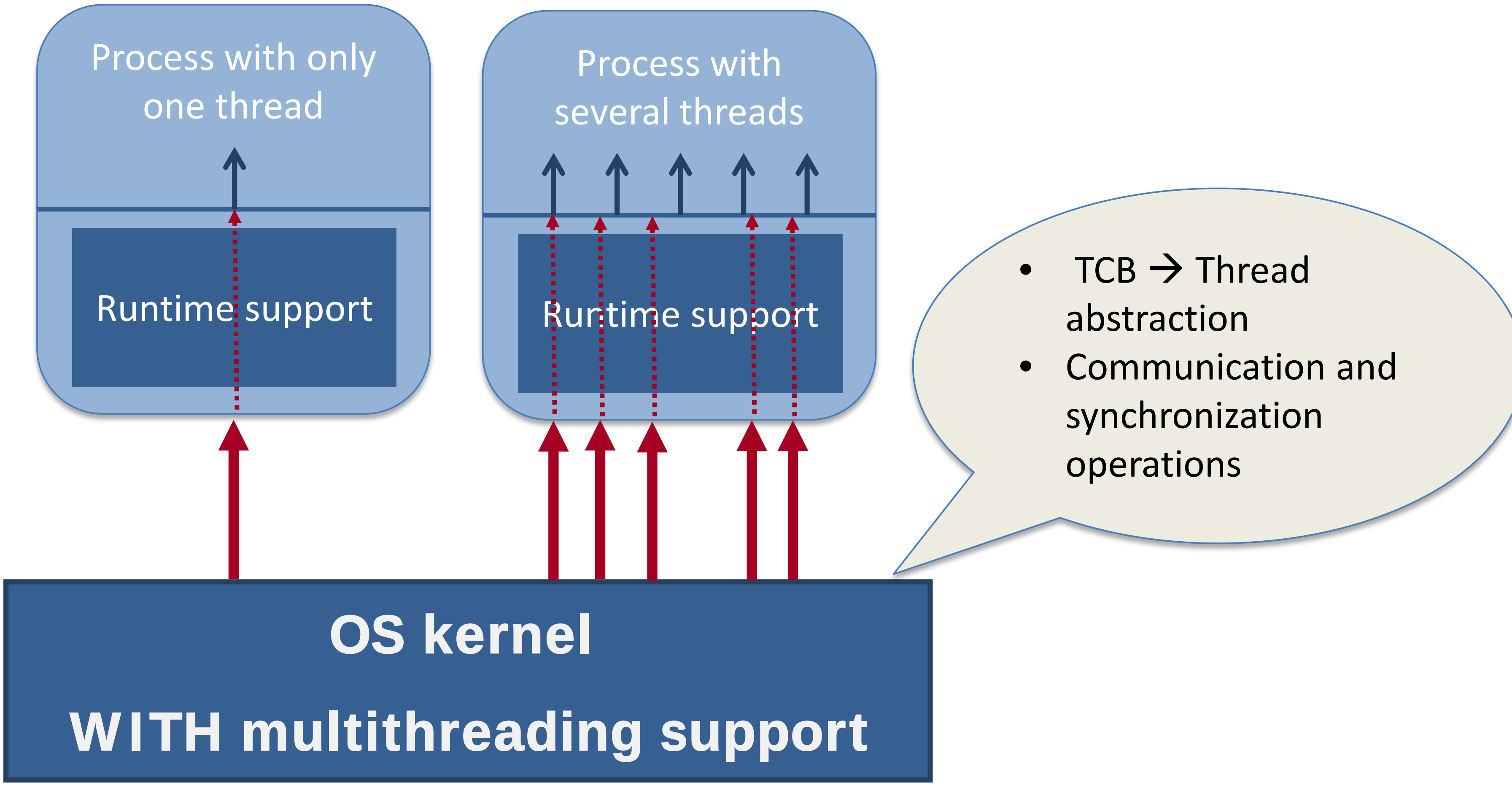
- **User level threads**
  - Multithreading support is provided by the **progamming language runtime**



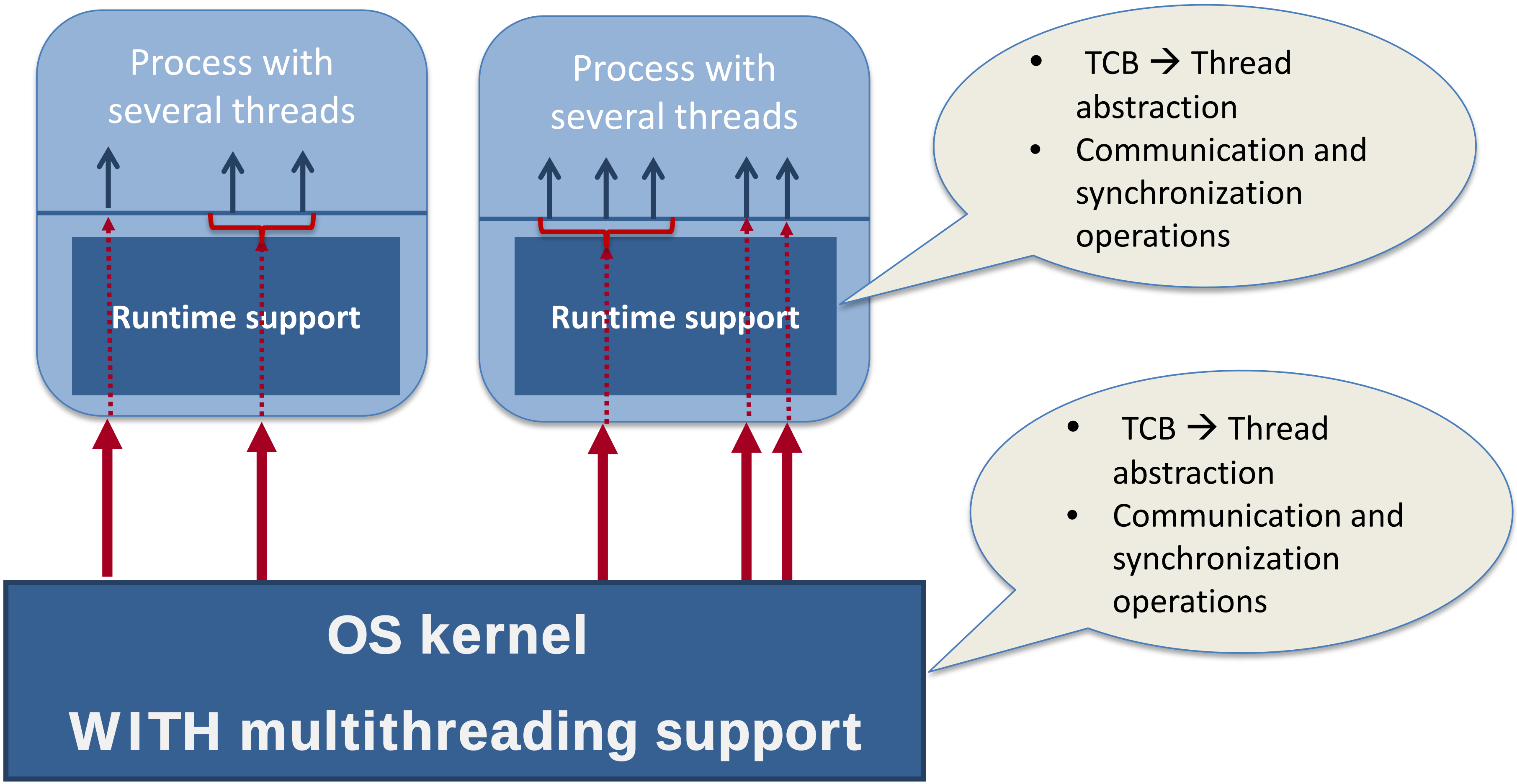
- TCB → Thread abstraction
- Communication and synchronization operations

Execution thread models

- **Kernel level threads**
  - Multithreading support is provided by the OS kernel by means of system calls



- **Hybrid model**
  - Multithreading support is provided by both programming language runtime and SO kernel, maximum programming flexibility and performance



Content

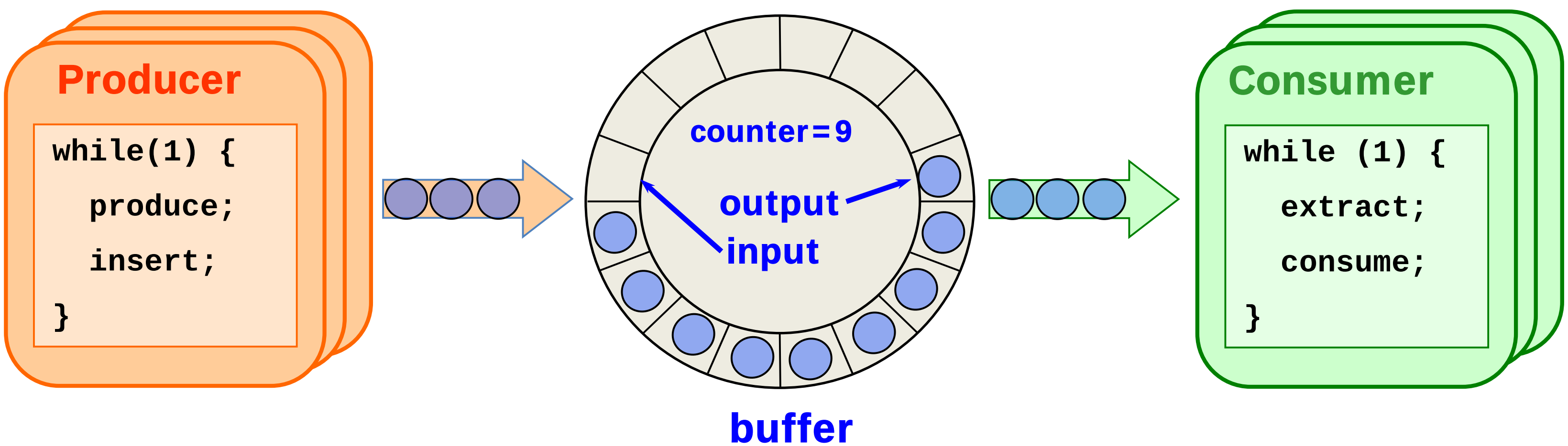
- Concurrent programming
- Execution thread concept
- Execution thread models
- **Synchronization requirement**
- Race condition concept

Synchronization requirement

- Concurrency is fundamental
  - Both at application and system level
- Concurrent programming deals with the following aspects:
  - Communication
  - Resource sharing
  - Synchronzation
  - CPU time reservation
- Concurrency **is present in both:**
  - Multiprocessor/multicomputer systems
  - Monoprocessor multiprogrammed/time sharing systems
- Concurrency can happen at three levels:
  - Several application running at once
  - Several processes and/or threads inside a given application
  - Several activities (i.e. deamons) inside the OS

Synchronization requirement

- Example: **producer/consumer problem with a bounded buffer**
  - A common communication model consists in items (data chunks) interchange from a source entity (producer) to a destination entity (consumer)
  - There is a bounded buffer (circular list) that avoids waits when there is a temporal speed unbalance between the producer and the comsumer:
    - It the buffer gets full the producer must wait
    - It the buffer gets empty the consumer must wait



Synchronization requirement

- Producer and consumer code sketch

Shared by producers and consumers

```
#define N  20
int  buffer[N];
int  input=0, output=0, counter=0;
```

"Active waiting" loops

```
void *func_prod(void *p) {
  int item;

  while(1) {
    item = produce();

    while (counter == N)
      /* empty loop */ ;

    buffer[input] = item;
    input = (input + 1) % N;
    counter = counter + 1;
  }
}
```

```
void *func_cons(void *p) {
  int item;

  while(1) {
    while (counter == 0)
      /* empty loop */ ;

    item = buffer[output];
    output = (output + 1) % N;
    counter = counter - 1;

    consume(item);
  }
}
```

"`counter`" and "`buffer`" are shared by producer and consumer threads
With several producer and consumers, "`input`" is shared by all producers and "`output`" by all consumers

Synchronization requirement

- **Producer/Consumer** multithreading issues
  - Producer and consumer threads **are executed concurrently**
    - **Shared variable** access
  - Threads are **selected to execution independently**
  - **Context switch** between threads **is performed by the scheduler**, programmers have no control about when and how context switch happens

A **race condition** happens when: *"Under concurrent execution a code is susceptible of incorrect execution"*

Content

- Concurrent programming
- Execution thread concept
- Execution thread models
- Synchronization requirement
- **Race condition concept**

Race condition concept

- If we suppose that:
  - Initially "counter" is 5
  - A producer does "counter = counter + 1;"
  - A consumer does "counter = counter - 1;"

"counter" final result must be 5

Producer:

```
counter = counter + 1;
```

```
lw reg1, counter
addi  reg1, reg1, 1
sw reg1, counter
```

Consumer:

```
counter = counter - 1;
```

```
lw reg2, counter
addi  reg2, reg2, -1
sw reg2, counter
```

But if the operation sequence is:

Context switches

T	Thread	Operation	reg1	reg2	counter
0	Prod.	**lw** reg1, counter	5	?	5
1	Prod.	**addi** reg1, reg1, 1	6	?	5
2	Cons.	**lw** reg2, counter	?	5	5
3	Cons.	**addi** reg2, reg2, -1	?	4	5
4	Cons.	**sw** reg2, counter	?	4	4
5	Prod.	**sw** reg1, counter	6	?	6

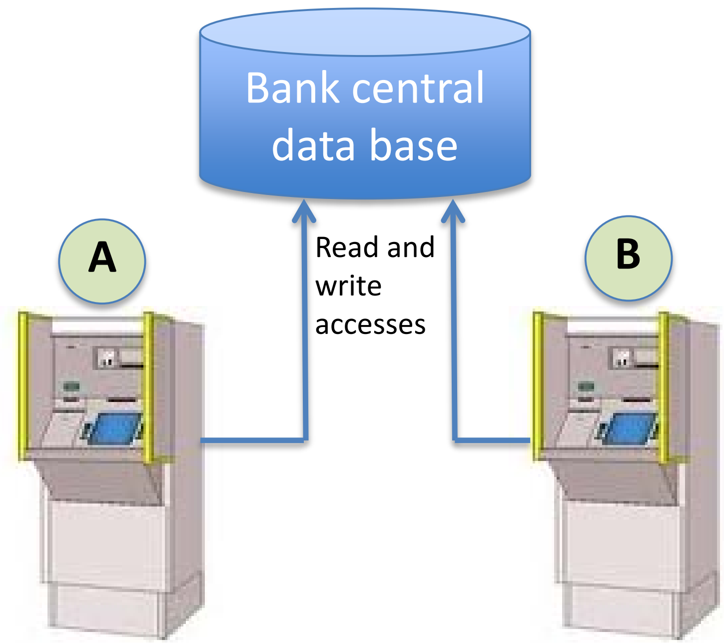
Incorrect

Race condition concept

- Suppose we want to take money from two ATM simultaneously

Take 20 Euros:

```
S=CheckBalance()
Si S>=20 {
  GiveMoney()
  NewBalance(S-20)
}
```



- Suppose that the actual account balance is 100 Euros and that the following operations are performed

Context switches

A: CheckBalance ()
B: CheckBalance ()
B: GiveMoney()
B: NewBalance(80)
A: GiveMoney()
A: NewBalance(80)

¡We got 40 euros and the account balance is 80!
Obviously money delivery through ATMs doesn't work this way

Checking and updating an accout are performed atomically, in the mean time other ATMs must wait

Race condition concept

- **Race condition definition:**

A **race condition** happens when the set of **concurrent operations** on a **shared variable** leave the variable in an **inconsistent state** according to the operations performed

- Race conditions appear because:
  - Programmers worry about sequential correction of their programs, but **they don't know when context switches happen**
  - **The OS doesn't know the dependecies between threads** in execution neither if it is convenient or not to perform a context switch at a certain moment

Race condition concept

- Race conditions are difficult to debug because they are due to thread interaction in time, being their isolated codes correct
  - Inconsistency use to happen from time to time only when a context switch happens randomly in an incovenient code place
  - Then try and error testing is not an adecuate procedure to detect or solve race conditions
- Multithreaded programs MUST avoid race conditions
  - Programmers have no control over context switch then programs must execute correctly regardless of the code places where context switches happen

Access to shared variables must be synchronized

Exercise 1

- In a system with kernel threads support four threads, H1, H2, H3 and H4 arrive with the following processing demands:

Thread	Arrival	Burst sequence
H1	0 (1st)	6 CPU + 2 I/O + 1 CPU
H2	0 (2nd)	6 CPU + 2 I/O + 1 CPU
H3	0 (3rd)	2 CPU + 3 I/O + 1 CPU + 3 I/O + 1 CPU
H4	0 (4th)	2 CPU + 3 I/O + 1 CPU + 3 I/O + 1 CPU

I/O is performed in one single device and delivers access following FCFS policy. What will be the **mean waiting time** if the scheduler uses the following algorithms?

a) SRTF

b) RR (q=2)

Exercise 2

- In a system **without** kernel threads support four threads, H1, H2, H3 and H4 arrive with the following processing demands:

Process	Thread	Arrival	Burst sequence
A	H1	0 (1st)	6 CPU + 2 I/O + 1 CPU
A	H2	0 (2nd)	6 CPU + 2 I/O + 1 CPU
B	H3	0 (3rd)	2 CPU + 3 I/O + 1 CPU + 3 I/O + 1 CPU
B	H4	0 (4th)	2 CPU + 3 I/O + 1 CPU + 3 I/O + 1 CPU

The programming language **run-time** schedules threads applying a **FCFS policy**. I/O is served by a single device with FCFS policy. What will be the **mean waiting time** if the system scheduler uses the following scheduling algorithms?

a) SRTF

b) RR (q=2)