

# Architecture of Digital Systems II

## 4 Processes and Operating Systems

# Process and Operating Systems

In many embedded applications, a single large program running on the microprocessor is not feasible. If multiple operations must be performed at varying points in time and at different rates, a single program becomes too complex, hard to verify for function and performance, and hard to debug and maintain.

In this chapter, we discuss two important abstractions, the **process** and the **operating system (OS)**, which both help to manage the complexity of software design.

The process defines the state of an executing program with respect to a single task, out of many to be performed by the system.

The operating system provides the mechanism for switching execution between the processes.

**Real-time operating systems (RTOSs)** provide functionality to satisfy the numerous real-time requirements typically placed on embedded systems.

# Multirate Systems

Many embedded systems are **multirate** systems: certain operations must be performed periodically, and each operation is executed at its own rate.

Every operation is performed by an individual process. The **period** of a process is the time between two successive executions of the process. The **rate** of a process is the inverse of its period.

The rate of processes is not necessarily constant – depending on the application it may vary over time.

*Example:* Automotive engine controller

- highest-rate process is firing the spark plugs; rate depends on rotational speed
- lower-rate processes: reading crankshaft position, computing fuel injection parameters,
- even lower-rate processes: reading temperature and gas sensors, reading throttle settings

# Co-Routine

An early approach to multi-tasking was the co-routine.

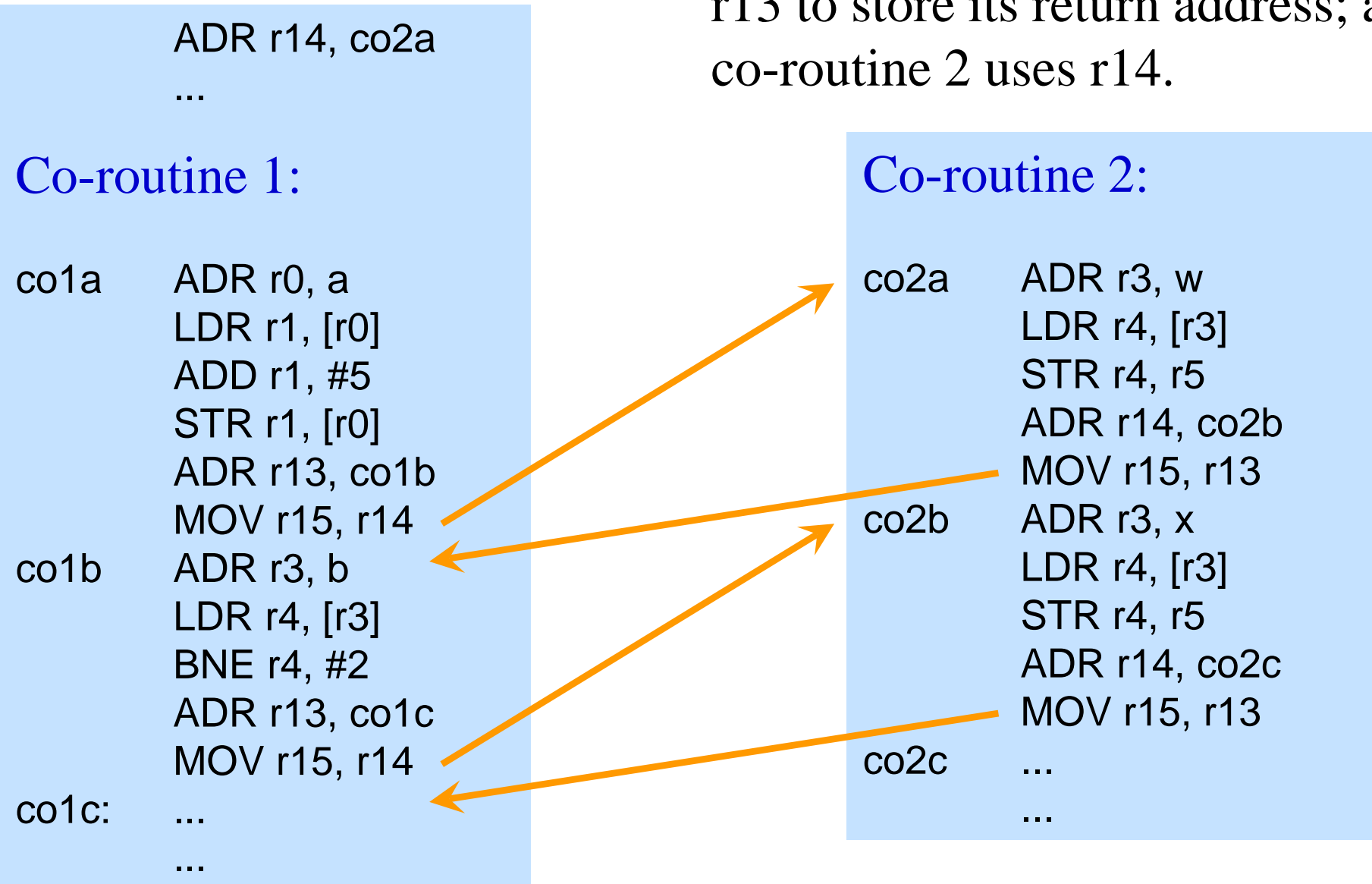
A co-routine is similar to a subroutine, however, it has several **entry points**.

Each task is implemented as a co-routine. The co-routines decide when to yield the CPU to another task, explicitly jumping into the corresponding co-routine.

The co-routines must, of course, not destroy any of the register contents used by other co-routines.

# Co-Routine: Example

In this example, co-routine 1 uses r13 to store its return address; and co-routine 2 uses r14.



## (Co-Routines)

Some high-level languages support co-routines (e.g., Modula2, Simula, C).

The advantages of multi-tasking based on co-routines are:

- co-routines are simple and easy to implement
- no "full-blown" operating system is needed
- switching between tasks can be very fast

The disadvantages are:

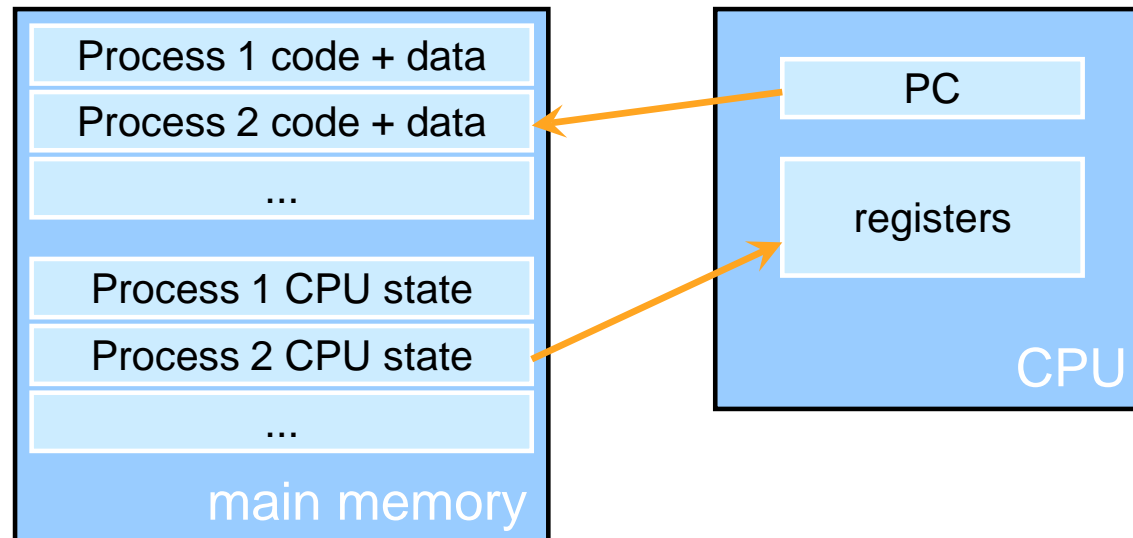
- flow of control may be difficult to understand
- "hard-coded" switching between co-routines – inflexible
- adding or removing tasks is tedious and error-prone

Co-routines are suitable for small-sized systems. They are not widely used today.

# Multi-Tasking using Processes

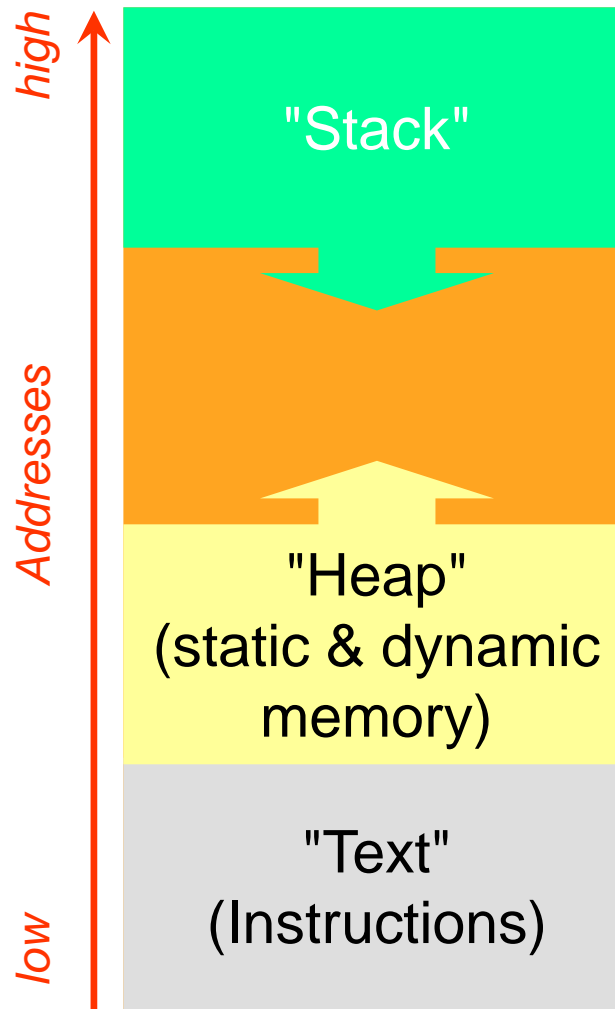
Today, most embedded systems use a more advanced multi-tasking concept. Switching between tasks is handled by a specialized system component – the **scheduler**. Individual tasks are handled by processes. The scheduler decides when to assign the CPU to a task. The scheduler is part of the operating system.

A process is the unique execution of a program. A process is defined by its program code and its data. The data lives in the main memory as well as in the CPU registers.



# Main memory representation of a Process

Typically, a process is represented in a designated area of main memory in the following way:



Each process has its own stack and heap. The heap is used to allocate memory on demand (e.g., using `malloc()`). The stack may grow from one end of the memory area, the heap from the other.



# Processes and Memory Management

Processes should only modify their assigned memory areas and keep out of those of other processes. It is useful to protect processes from each other by restricting memory access.

Using [virtual memory management](#), a process can be assigned its own address range. Every process has its own address translation table. The translation tables are managed by the operating system.

In systems with memory management units (MMUs), the hardware can initiate an exception if a prohibited memory access is attempted by a process.

As of today, many embedded systems do not have MMUs because of cost and performance. Usually, the processes that will be running are known at design time and can be tested and verified to stay within their memory bounds.

# Threads

**Threads** are a "light-weight" version of processes often used in embedded systems. They have their own distinct sets of values for CPU registers but share the same main memory space. This means that

- communication among threads can be simple and fast
- no MMU is needed
- however: care must be taken so that a thread does not inadvertently destroy data used by another thread!

Most general-purpose computer operating systems support both: (standard) *processes* and light-weight *threads*.

In embedded systems, operating systems often provide only threads.

# Multi-Tasking Operating Systems - Example

**POSIX** (“Portable Operating System Interface”) is a set of standards defining the application programming interface (API) of multi-tasking Unix-like operating systems. POSIX is standardized in the IEEE 1003 set of standards. Operating Systems complying to the standard are source-code compatible, i.e., applications that use only POSIX-standard library functions can be compiled and run on a new POSIX platform without modification.

Some examples of POSIX-compliant operating systems are **GNU/Linux**, **Solaris**, and **AIX**.

The POSIX standard has been extended to support real-time requirements. Many real-time operating systems are POSIX-compliant.

## Example: Processes in POSIX

A new POSIX process can only be created by duplicating an existing process. If intended, the new process may start a new program.

The `fork()` system function makes an identical copy of the calling process. The original process is called *parent*, the new process is called *child*. In both processes, the function returns. However, in the parent process, it returns the child process' ID; in the child process, it returns 0.

The `execv()` system function overlays a process with a new program execution.

### Example:

```
pid_t pid = fork();
if (pid == 0) {
    /* we are child */
    execv("prog", childargs);

    ...
} else if (pid == -1) {
    /* an error occurred */
} else {
    /* we are parent */
    parent_stuff();
    wait(&childstatus);
}
```

# Context Switching

At any one time, only one process is running on a (single-core) CPU. **Context switching** is the mechanism to move the CPU from one executing process to another. Context switching should be fast.

In a context switch, the operating system needs to replace the context of one process by that of another. The context is stored in the **process control block (PCB)** and includes

- PC
- stack pointer
- general-purpose registers
- status registers
- return address registers
- pointers to memory address translation tables
- ...

# Cooperative Multi-Tasking

We will look at two forms of multitasking, *cooperative* and *preemptive*.

In a **cooperative multitasking** system processes give up the CPU to other processes voluntarily. A process decides autonomously when to initiate a context switch.

This concept is similar to the co-routine discussed earlier. The main difference is that a process doesn't directly call another process but calls the scheduler. The scheduler decides which process runs next.

The starting point is a standard procedure call mechanism. A process calls the scheduler. The scheduler saves the current process's CPU state and registers and loads the next process's context. It restores CPU state such that a subroutine return from the scheduler will move control to the new process.

# Cooperative Multitasking: Example

## Process 1:

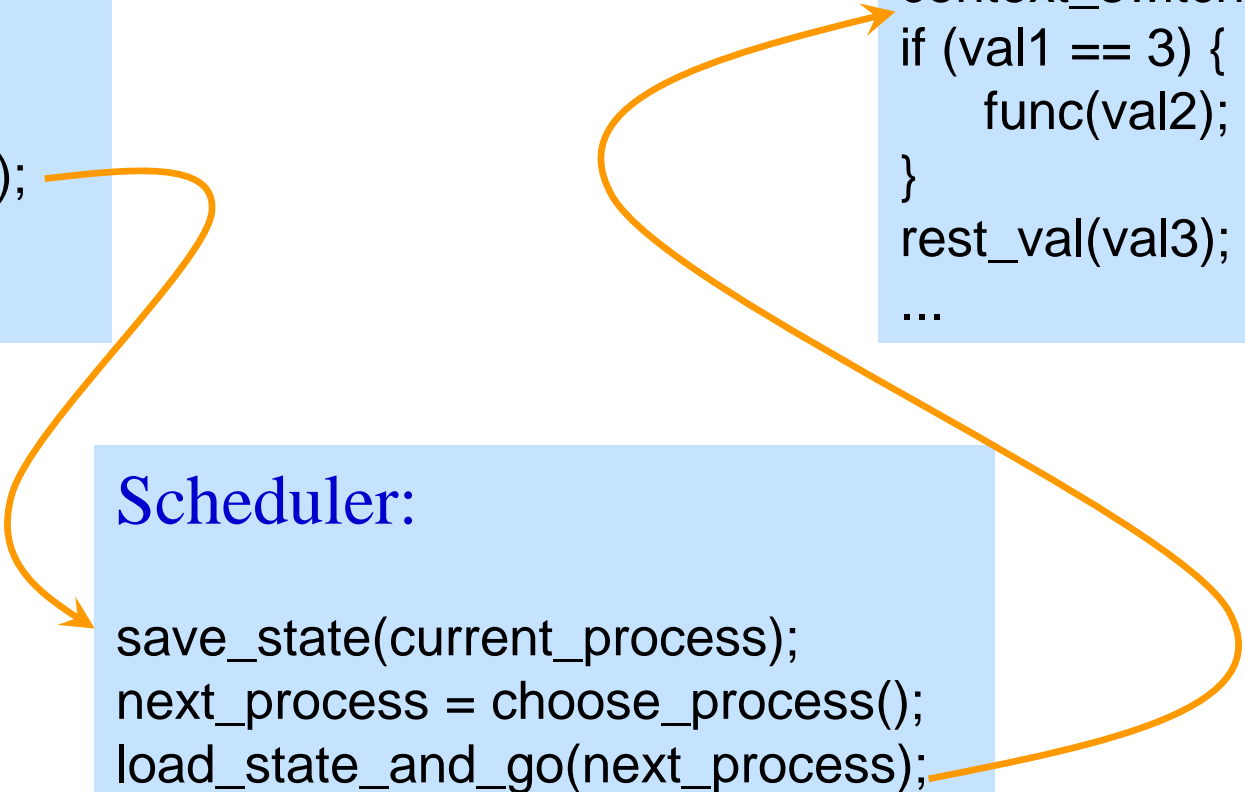
```
if (x > 29) {  
    sub1(y);  
} else {  
    sub2(y, z);  
context_switch();  
proc_a(a, b, c);  
...
```

## Process 2:

```
proc_data(r, s, t);  
context_switch();  
if (val1 == 3) {  
    func(val2);  
}  
rest_val(val3);  
...
```

## Scheduler:

```
save_state(current_process);  
next_process = choose_process();  
load_state_and_go(next_process);
```



# (Cooperative Multitasking)

Cooperative multitasking has some advantages over the co-routine concept:

- new processes can be added more easily to the system,
- with a central scheduler, there is flexibility in choosing which process to call next – the scheduler may give priorities to more important / more time-critical processes.

A major problem, however, is that the system only works properly if all processes adhere to the cooperative scheme. If only a single buggy process doesn't give up the CPU, the system may lock up. Or if processes execute for too long before invoking a context switch, real-time constraints for the system may be violated.



# Preemptive Multitasking

An improvement to cooperative multitasking is to take the CPU away from a process automatically after a certain amount of time. This is called **preemptive multitasking**.

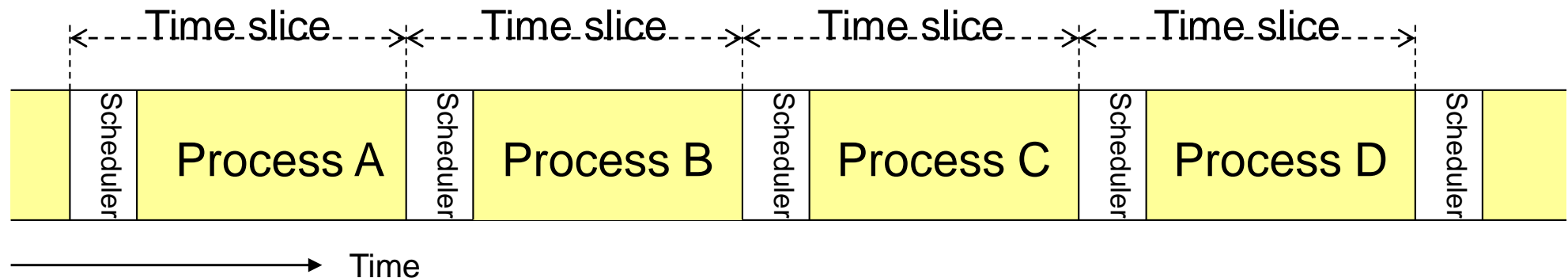
The system hardware contains a system timer that generates periodic interrupts to the CPU.



The interrupt causes control to be transferred to the interrupt handler. The handler calls the process scheduler of the operating system to switch context. The new process's context is restored. The interrupt return address is modified so that return from the interrupt resumes execution of the new process.

# Preemptive Multitasking

Using a periodic system timer interrupt basically chops CPU time into *time slices*.



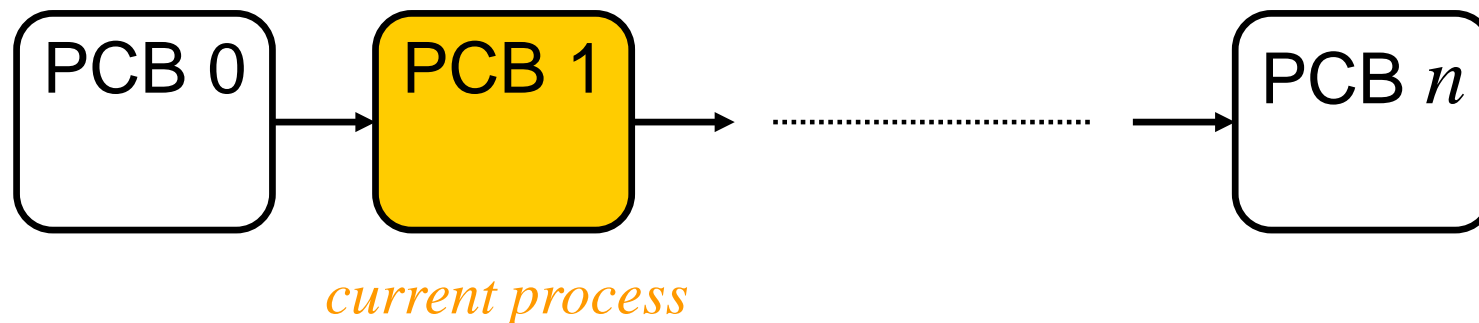
The period of the system timer minus the time needed for context switching is the maximum time a process may run uninterrupted. Choosing the length of the time slice (or **quantum**) is critical – making it too long increases the latency of processes; making it too short increases the overhead of context switching and reduces overall system performance.

# Scheduling

The job of the scheduler is to determine the process that is to run next. There are many possibilities for a **scheduling policy**.

A simple policy is **round-robin scheduling**:

All process control blocks are kept in a linked list. The processes are activated one after the other. After the end of the list is reached the scheduler starts with the first process again.



However, activating a process that is not doing anything except waiting for an I/O device or another process is not efficient.

# Process State

The operating system considers a process to be in one of three basic **scheduling states**: *waiting*, *ready* or *running*.

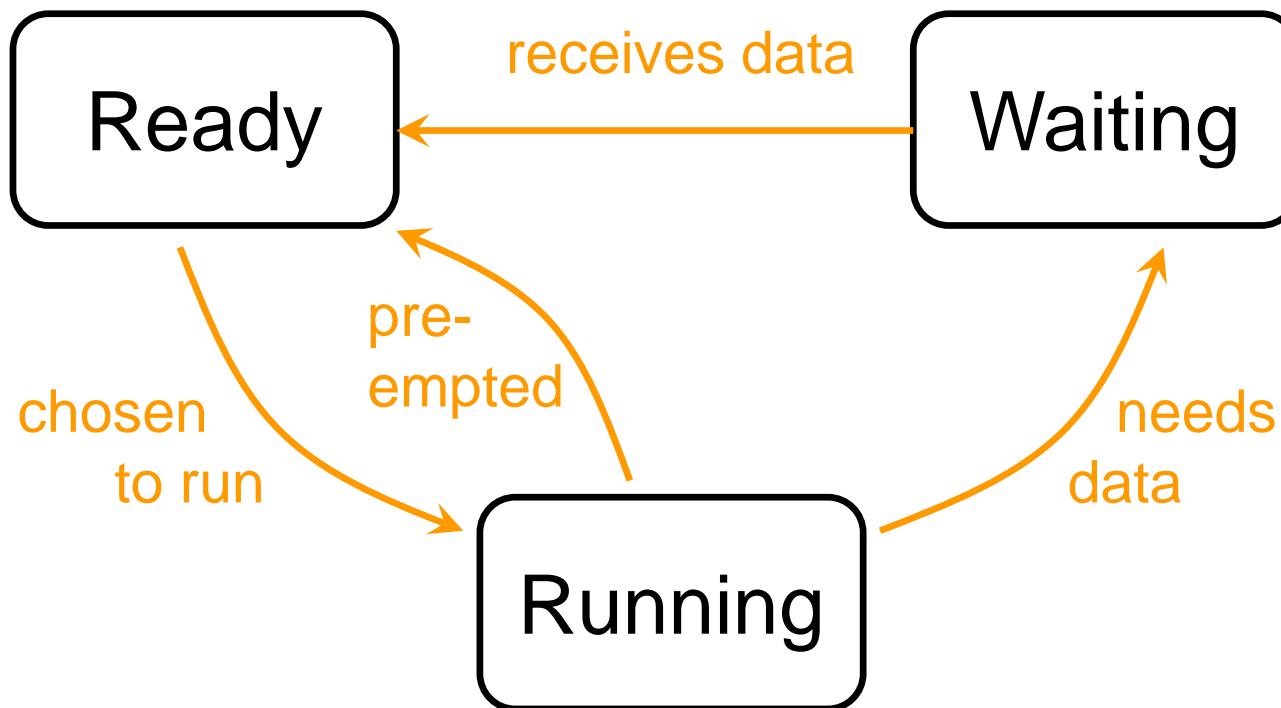
There is, at any one time, only a single process in **running** (**executing**) state: that is the process currently being executed by the CPU.

All other processes that could be executed are in the **ready** state – the scheduler chooses the next running process from the set of ready processes.

A process that is waiting for some event, e.g., data from an I/O device or another process, is in the **waiting** (**blocked**) state.

# Process State

The scheduler maintains the state of a process. The state is stored in the process control block. The basic state transitions are:



A waiting process is set to the ready state when the event that the process was waiting for has occurred, e.g., data became available.

# Process Priorities

It is common to assign priorities to processes and to select the next running process based on the priority information. More important or more time-critical tasks can be assigned higher process priorities. The scheduler selects among the ready processes the one with the highest priority.

Priorities may be fixed (*static priorities*) or may change during execution (*dynamic priorities*).

The priority of a process is stored together with the information on process state and CPU state in the process control block.

# Scheduling Policies

A **scheduling policy** defines how the scheduler selects the next running process from the set of ready processes. In priority-based scheduling policies, this amounts to determining the process priorities.

Many scheduling policies exist, each one tailored for a specific application scenario.

Scheduling policies vary with respect to the timing requirements they can handle and how efficiently they make use of the CPU resource (**CPU utilization**).

Scheduling policies also differ in implementation effort and in the amount of **scheduling overhead** – the CPU time spent in the scheduler and not in the processes.

In this course, we present only a simplified view on scheduling.

# Timing Requirements on Processes

Timing requirements are expressed with the following terms:

The **release time** is the time when a process goes from the waiting state to the ready state. The **initiation time** is the time at which a process actually begins execution on the CPU.

A **deadline** specifies when a process must be finished.

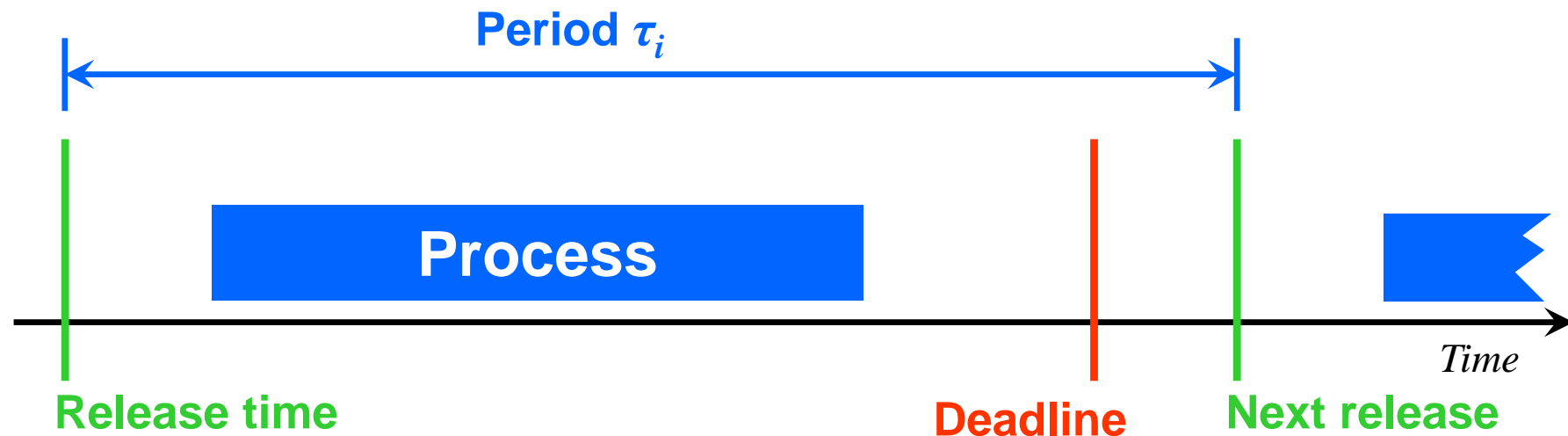


**Aperiodic processes** are initiated by events (e.g., data ready).



# (Timing Requirements on Processes)

Periodic processes have a period associated with them. They are executed once per period.



The period is also called the **initiation interval**.

The inverse of the period is the **rate**.

A **rate requirement** specifies how often a process must be initiated per unit of time.

# Rate-Monotonic Scheduling

A scheduling policy commonly employed in embedded systems is **rate-monotonic scheduling (RMS)**.

The analysis of RMS is based on the following assumptions:

- All processes  $P_i$  run with a constant period  $\tau_i$  and constant execution time  $T_i$
- Deadlines of processes are at the end of their periods
- Single CPU
- Scheduling overhead is neglected
- No dependencies between processes

Under these assumptions it is possible to show that *the optimal scheduling policy is to statically assign priorities according to their period – the shorter the period of a process, the higher its priority.* RMS is therefore a **static scheduling policy**. It provides high CPU utilization while ensuring that all processes meet their deadlines.

# RMS Example

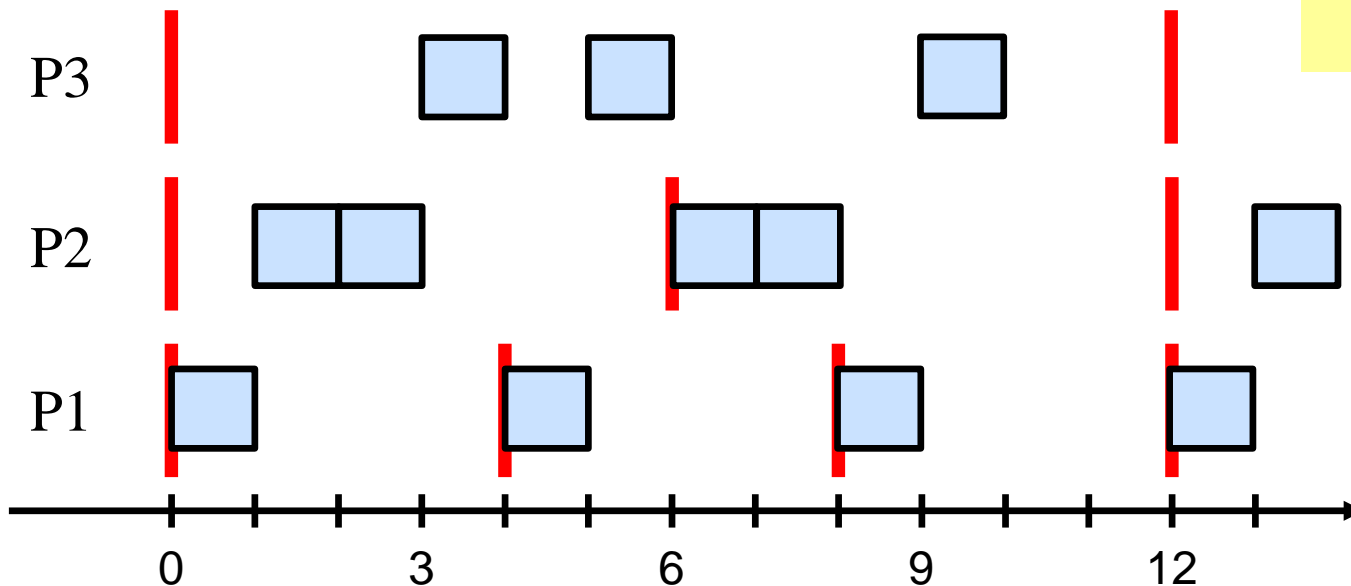
Consider a set of  $n$  processes. Does a feasible RMS schedule exist?

Priority	Process	Period $\tau_i$	Execution time $T_i$
1	P1	4	1
2	P2	6	2
3	P3	12	3

## CPU Utilization

Fraction of CPU time actually spent in processes:

$$U = \sum_{i=1}^n \frac{T_i}{\tau_i}$$



$$U = \frac{10}{12} \approx 83\%$$

# Schedulability Tests

A schedulability test allows to analyze a given set of processes whether a feasible schedule under a particular scheduling policy exists.

For rate-monotonic scheduling (RMS):

**Sufficient condition** for schedulability:

$$U = \sum_{i=1}^n \frac{T_i}{\tau_i} \leq n(2^{1/n} - 1)$$

Note: The sequence  $n(2^{1/n} - 1)$  is monotonically decreasing. It has the limit

$$\lim_{n \rightarrow \infty} n(2^{1/n} - 1) = \ln 2 \approx 69\%$$

Hence, another, (weaker) sufficient condition for RMS schedulability is:

$$U \leq 69\%.$$

# Earliest-Deadline-First Scheduling

**Earliest-Deadline-First** Scheduling (EDF) is another well-known and simple scheduling policy: it assigns priorities in the order of deadline. The highest priority is given to the process whose deadline approaches next. Priorities must be recalculated at every termination of a process (**dynamic scheduling policy**).

EDF can achieve 100% CPU utilization (assuming zero-overhead scheduling). However, missing of deadlines is harder to predict.

**Necessary and sufficient condition** for schedulability:

$$U = \sum_{i=1}^n \frac{T_i}{\tau_i} \leq 1$$

Implementation of EDF is more complex than that of RMS since the pool of processes must be kept sorted in order of deadline. This requires additional computation time in the scheduler.

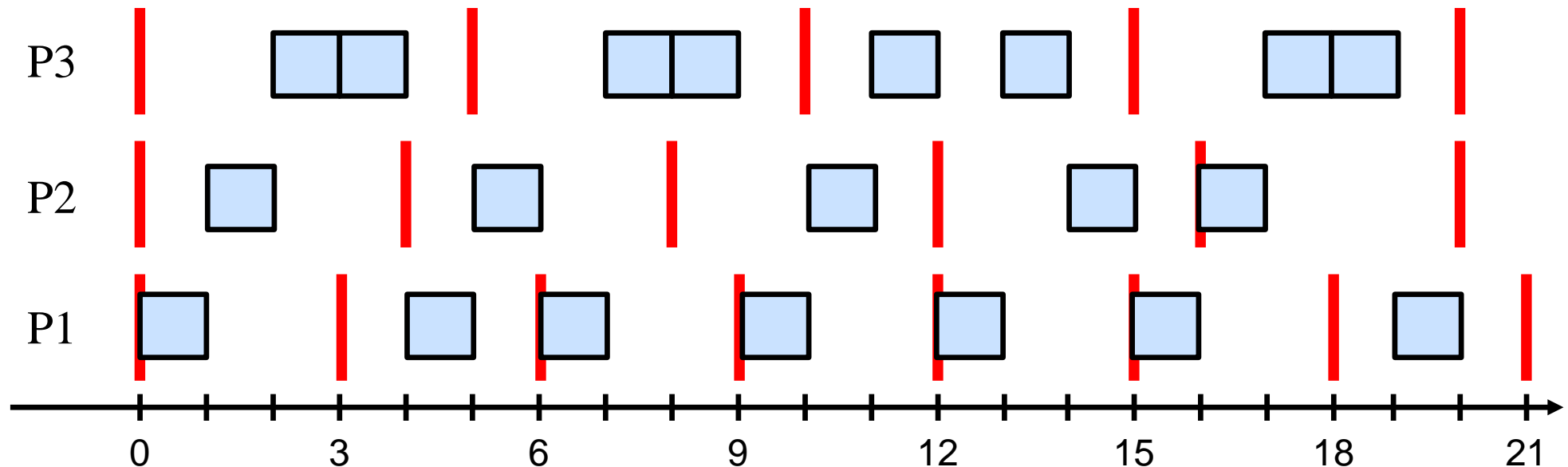
# EDF Example

Consider a set of processes. Does a feasible EDF schedule exist?

Process	Period $\tau_i$	Execution time $T_i$
P1	3	1
P2	4	1
P3	5	2

$$U = \sum_{i=1}^n \frac{T_i}{\tau_i} = \frac{1}{3} + \frac{1}{4} + \frac{2}{5}$$

$$U = \frac{59}{60} < 1 \rightarrow \text{Yes, schedule exists.}$$



# Inter-Process Communication

Very often processes need to communicate, sending each other data or signaling events for synchronization. The operating system provides mechanisms for **inter-process communication (IPC)**.

Communication may be either blocking or non-blocking.

In **blocking** communication, a process sends data or signals an event and then goes into the *waiting/blocking* state until it receives a response.

In **non-blocking** communication, a process sends some data and then proceeds with its computations immediately.

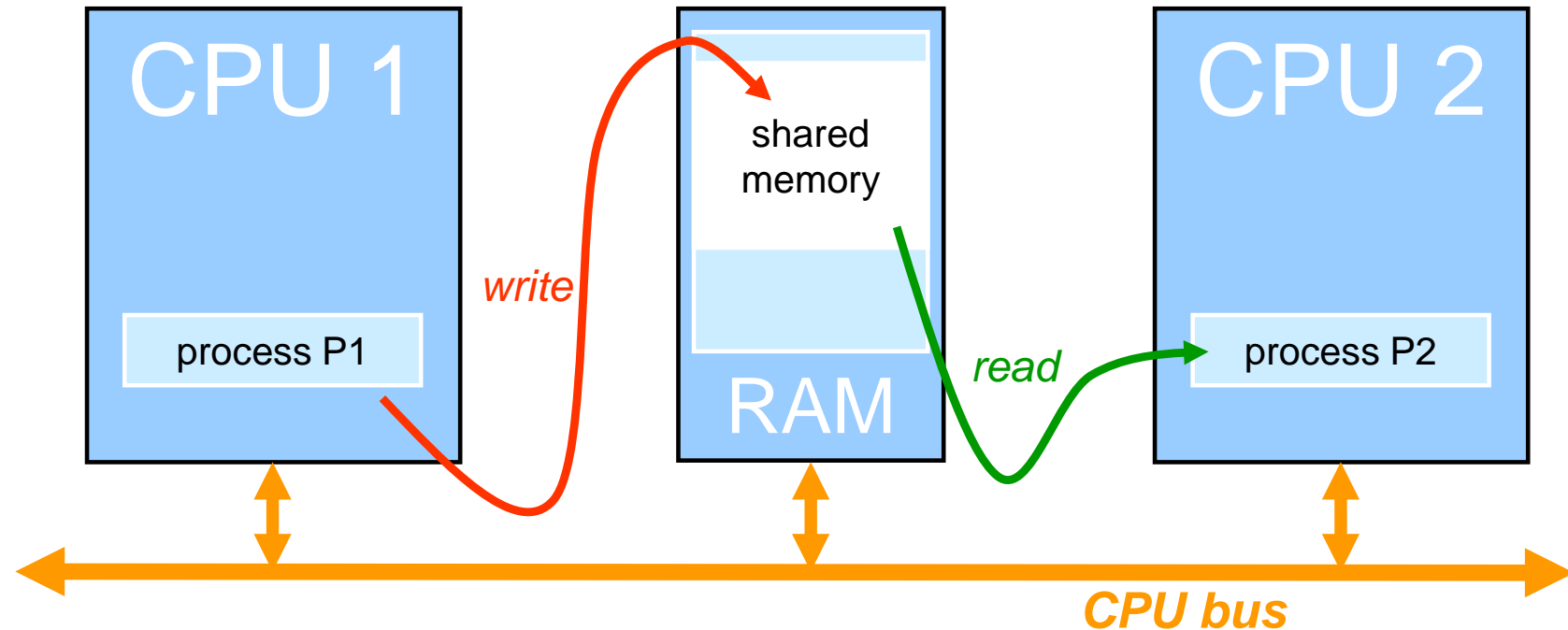
In practice, two IPC concepts are common:

- shared memory
- message passing

Functionally, these concepts are equivalent – we can construct either concept using the other.

# Shared Memory

**Shared memory** is an area in main memory that is accessible by two or more communicating partners, e.g., processes or threads. (Threads always share a common memory space.)



*Example: two processes communicating through shared memory*



# Example: Shared Memory Communication (1)

We consider an example in a Java-like language with two threads communicating through shared memory.

The class `m` defines the shared memory. Its `main()` method creates two threads, a producer, `p`, and a consumer, `c`.

```
public class m {  
    final public static int BUFSIZE = 100;  
    final public static char EOF = 26;  
    public static int in = 0, out = 0;  
    public static char buffer[] = new char[BUFSIZE];  
  
    public static producer p;  
    public static consumer c;  
  
    public static void main(String args[]) {  
        p = new producer();  
        c = new consumer();  
        p.start();  
        c.start();  
    }  
  
    public static int next(int k) {  
        if (k < BUFSIZE-1) return k+1; else return 0;  
    }  
}
```

## Example: Shared Memory Communication (2)

The producer class generates data and writes it into the buffer.

If the buffer is full, the producer thread goes to sleep (state *waiting*) until reactivated by the consumer. If the buffer was empty before the write, the producer thread wakes up the consumer thread, (putting it into state *ready*).

```
class producer extends Thread {
    public void run() {
        char c = 0;
        while (c != m.EOF) {
            c = getchar();
            boolean was_empty = (m.in == m.out);
            boolean is_full = (m.next(m.in) == m.out);
            if (is_full)
                suspend();           // go to sleep
            m.buffer[m.in] = c;       // write into buffer
            m.in = m.next(m.in);      // advance write ptr
            if (was_empty) // now one datum available
                m.c.resume();        // wake consumer
        }
    }
    private char getchar() { ... }
}
```

## Example: Shared Memory Communication (3)

The consumer class reads data from the buffer and prints it to the console.

If the buffer is empty the consumer thread goes to sleep (state *waiting*) until reactivated by the producer.

If the buffer was full before the read the consumer thread wakes up the producer thread (putting it into state *ready*).

```
class consumer extends Thread {
    public void run() {
        char c = 0;
        while (c != m.EOF) {
            boolean was_full = (m.out == m.next(m.in));
            boolean is_empty = (m.in == m.out);
            if (is_empty)
                suspend();           // go to sleep
            c = m.buffer[m.out];      // read from buffer
            m.out = m.next(m.out);    // advance read ptr
            if (was_full)             // now one mem-cell available
                m.p.resume();        // wake producer
            System.out.print(c);
        }
    }
}
```

# Race Condition

This example design has a major flaw:

The two threads run concurrently. The access of one thread to the shared memory may be interleaved by the other thread. Consider for example the case where, just after the consumer has checked for an empty buffer and determined that `is_empty == true`, the producer puts a datum into the buffer. It sends the consumer a wake-up signal, but this signal is lost because the consumer is still running. Shortly after, the consumer evaluates the `is_empty` flag and goes to sleep. The producer keeps writing data into the buffer until it is full and then goes to sleep, also.

This is called a **race condition**:

The system breaks if the wake-up signal arrives before the consumer goes to sleep.

The problem can be solved using semaphores.

# Semaphore

A **semaphore** is a protected variable controlling access to a shared resource such as shared memory (Dijkstra, 1968).

In the general case, a semaphore is a counter. It can only be accessed through the following operations:

```
P(Semaphore s)    // Acquire Resource
{
    wait until s > 0, then s := s-1;
}
```

```
V(Semaphore s)    // Release Resource
{
    s := s+1;
    if (s = 1) then wake waiting processes;
}
```

```
Init(Semaphore s, Integer v)
{
    s := v;
}
```

These operations must be *atomic*, i.e., they may not be interrupted by other processes or threads.

Semaphores are functions of the operating system.

# Example with Semaphores (1)

We add semaphores to our example.

We assume that there be system procedures called

`sem_wait(s)` –  
the P(s) operation

`sem_post(s)` –  
the V(s) operation

`sem_init(s, v)` –  
the `init(s, v)`  
operation

```
public class m {  
    final public static int BUFSIZE=100;  
    final public static char EOF = 26;  
    public static int in = 0, out = 0;  
    public static char buffer[] = new char[BUFSIZE];  
    public static producer p;  
    public static consumer c;  
  
    // semaphores:  
    public static int filled, available;  
  
    public static void main(String args[]) {  
        sem_init(filled, 0);  
        sem_init(available, BUFSIZE-1);  
        p = new producer();  
        c = new consumer();  
        p.start();  
        c.start();  
    }  
    public static int next(int k) { ... } // as before  
}
```

## Example with Semaphores (2)

The producer calls `sem_wait(available)` to find out whether there is space in the buffer. If `available==0` then the thread is suspended. Only after the consumer has read out a datum and called `sem_post(available)`

will the producer be reactivated to write into the buffer.

After writing, the producer calls `sem_post(filled)` on the filled semaphore to indicate that data is ready.

```
class producer extends Thread {
    public void run() {
        char c = 0;
        while (c != m.EOF) {
            c = getchar();
            sem_wait(available);    // P()
            m.buffer[m.in] = c;    // write into buffer
            m.in = m.next(m.in);    // advance write ptr
            sem_post(filled);      // V()
        }
    }
    private char getchar() { ... }
}
```

## Example with Semaphores (3)

The consumer calls `sem_wait(filled)` to find out whether there is data ready. If `filled==0` then the thread is suspended. Only after the producer has written a datum into the buffer and called `sem_post(filled)`

```
class consumer extends Thread {  
    public void run() {  
        char c = 0;  
        while (c != m.EOF) {  
            sem_wait(filled);           // P()  
            c = m.buffer[m.out];        // read from buffer  
            m.out = m.next(m.out);      // advance read ptr  
            System.out.print(c);  
            sem_post(available);        // V()  
        }  
    }  
}
```

will the consumer be reactivated to read from the buffer. After reading, the producer calls `sem_post(available)` on the available semaphore to indicate that buffer space has become available.

---

*The semaphore "system calls" shown here are not standard Java!*



# Mutex

A **mutex** is the special case of a *binary* semaphore (the counter is initialized with 1). It is also called **lock**. A mutex is usually used to protect a single resource from being accessed by more than one process/thread simultaneously (hence the name from "mutual exclusion").

The sequence of program statements in a process/thread that access the resource is called **critical section**. The critical sections must be guarded by the mutex operations. These are:

```
Lock(Mutex m)
{
    wait until m = 1;
    m = 0;
}
```

```
Unlock(Mutex m)
{
    m = 1;
    wake waiting processes;
}
```

These operations need to be atomic. – Operating systems and some programming languages have built-in support for mutexes.

# Mutex: POSIX Example

Consider two threads that both read and write to a global variable.

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

```
static int X = 0;           // a global variable
```

```
pthread_mutex_t M = init_value; // the mutex
```

```
void thread1() {
```

```
    int g;
```

```
    ...
```

```
    pthread_mutex_lock(&M);
```

```
    g = X;           // read
```

```
    g = g+1;
```

```
    X = g;           // write
```

*critical  
section*

```
    pthread_mutex_unlock(&M);
```

```
    ...
```

```
}
```

```
void thread2() {
```

```
    int g;
```

```
    ...
```

```
    pthread_mutex_lock(&M);
```

```
    g = X;           // read
```

```
    g = g+1;
```

```
    X = g;           // write
```

*critical  
section*

```
    pthread_mutex_unlock(&M);
```

```
    ...
```

```
}
```

The lock ensures that no concurrent access occurs on the variable.

# Message Passing

**Message Passing** is another common concept for inter-process communication.

The communicating units do not share a common area in memory, but rather pass on packets of data between each other. Message passing is used frequently in parallel computing (computer clusters) and distributed systems.

An operating system may support some form of message passing. It then provides functions for sending and receiving messages. It maintains the buffers used for intermediate storage of messages and it ensures synchronization of the send and receive operations.

# Message-passing Primitives

Messages are exchanged between communicating threads using two primitives, *SEND* and *RECV*.

The primitives have the following parameters:

- a communication address, identifying the communicating partner thread,
- the message (a sequence of bytes),
- the length of the message (number of bytes),
- additional information, e.g., tags identifying the type of the message, or control flags.

There can be synchronous and asynchronous versions of these primitives.

# Synchronous vs. Asynchronous SEND/RECV

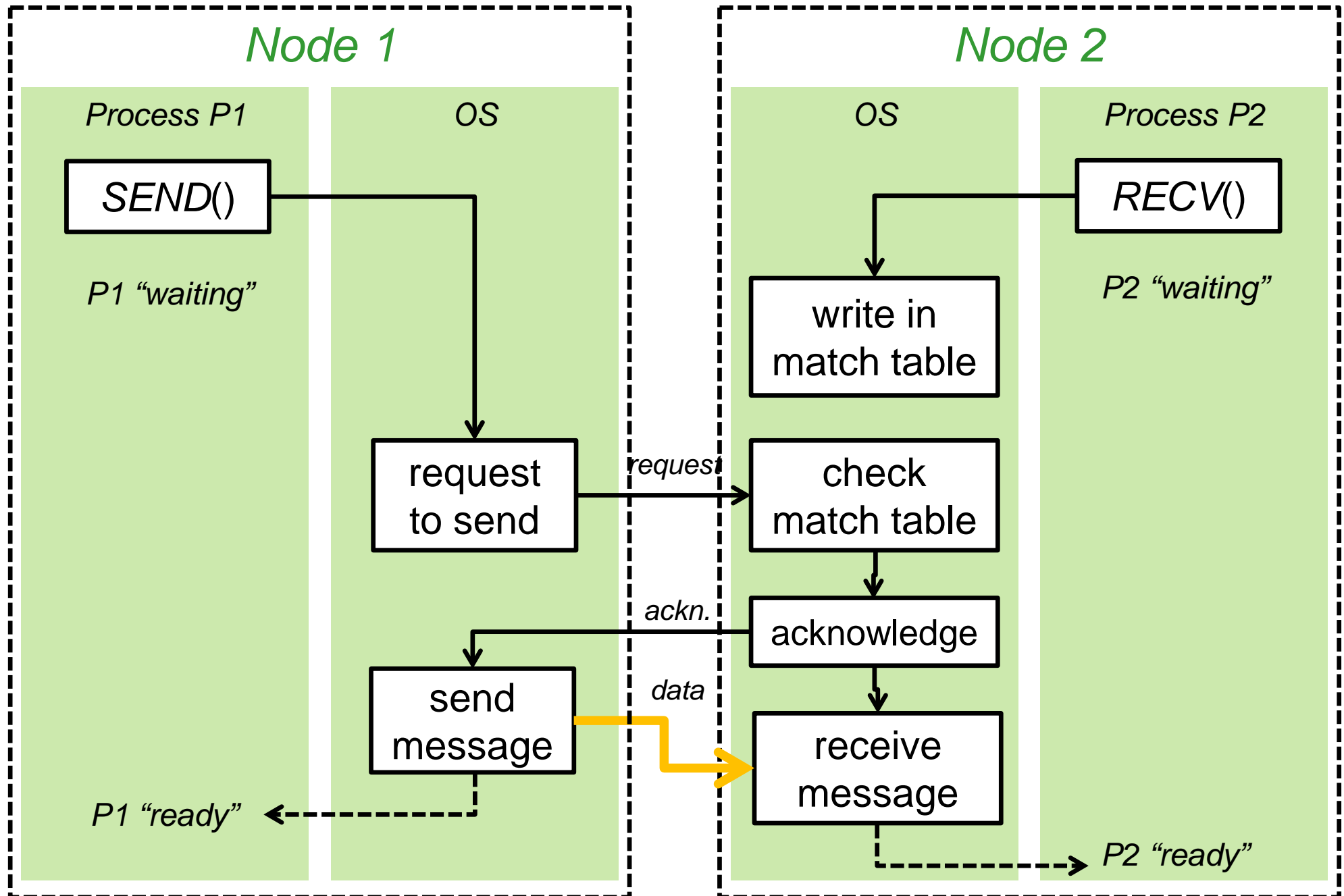
In synchronous *SEND/RECV*, the function call blocks until the communication has completed. Synchronization between the threads is implicit in the functions, and naturally avoids data races between the sending and the receiving thread.

There are two disadvantages with synchronous message passing:

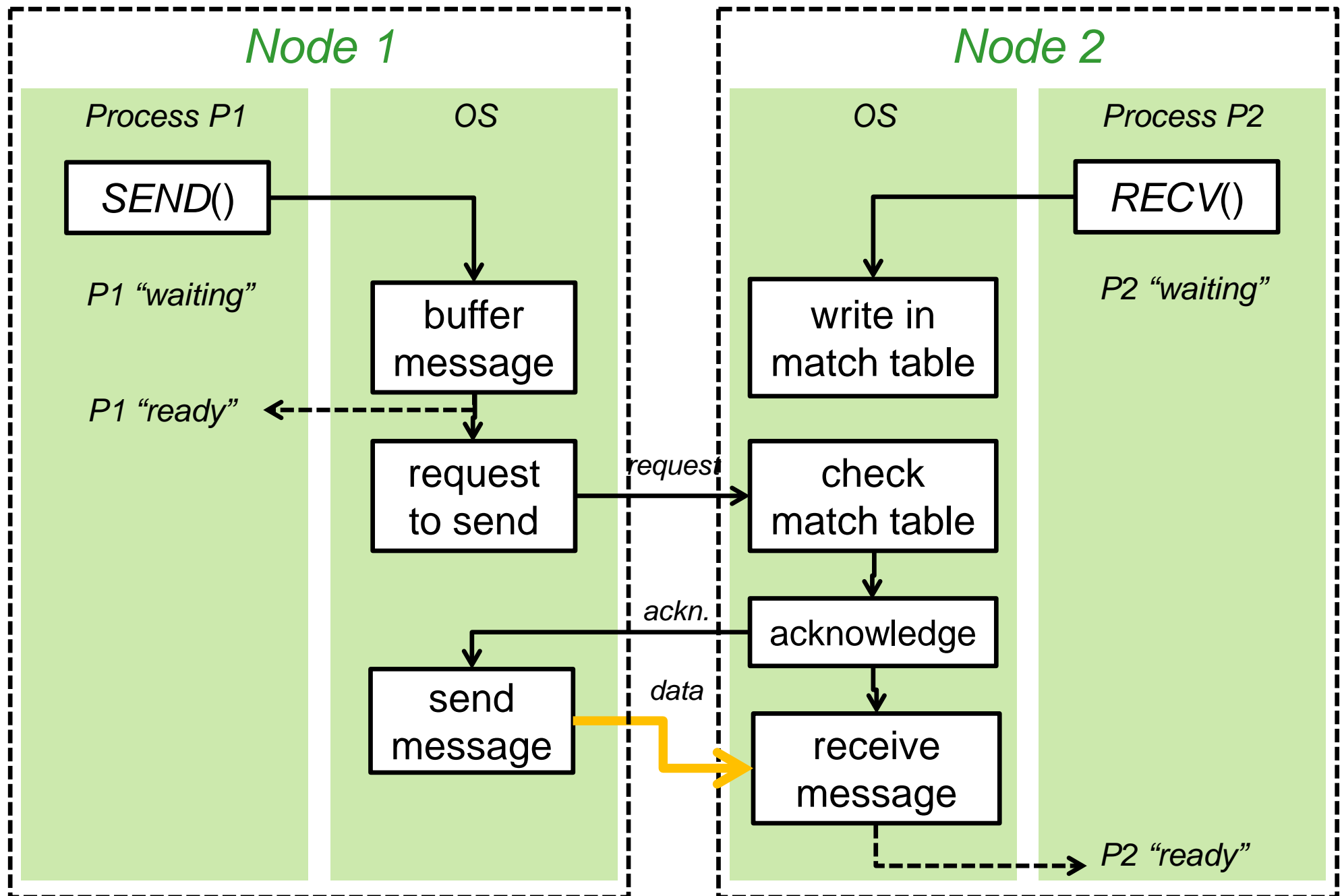
- possibility of deadlock
- blocking does not allow to overlap communication with computation

In asynchronous message passing, the message is buffered in message buffers so that *SEND/RECV* do not have to wait for the corresponding call on the other side of the communication, but can return immediately.

# Synchronous SEND / Synchronous RECV



# Asynchronous SEND / Synchronous RECV



# Example: POSIX Message Queues

POSIX provides system calls for *message queues*.

A new message queue can be requested from the operating system using a `msgget()` call:

```
#include <sys/msg.h>

int queue_id = msgget(...);
```

Messages are user-defined. They must follow the following format:

```
struct mymsg {
    long messagetype;
    char data[N];
};
```



## (Example: POSIX Message Queues)

A process may send a message using a **msgsnd()** call. Example:

```
struct mymsg snd_message = { 15, "Hello world,\n" };  
  
msgsnd(queue_id, &snd_message, sizeof(snd_message), flags);
```

A second process may read the message using a **msgrcv()** call. Example:

```
struct mymsg rcv_message;  
  
ssize_t nread  
= msgrcv(queue_id, &rcv_message, sizeof(rcv_message), msgtype, flags);
```