# CONCURRENCY

LECTURE V
TDDI11 Embedded Software

DEPT. COMPUTER AND INFORMATION SCIENCE (IDA)

LINKÖPINGS UNIVERSITET

## OUTLINE

- Why concurrency?
- Foreground / background vs. multi-tasking systems
- Concurrent processes and communication
- Scheduling
- Bus scheduling

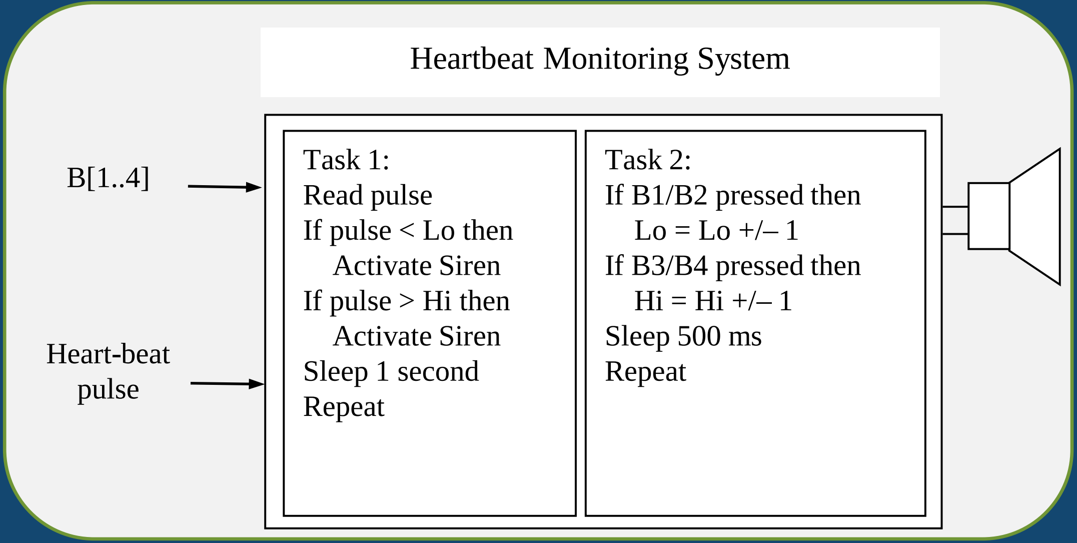
# WHY CONCURRENCY?

Separate tasks running independently but sharing data

Difficult to write system using sequential program model

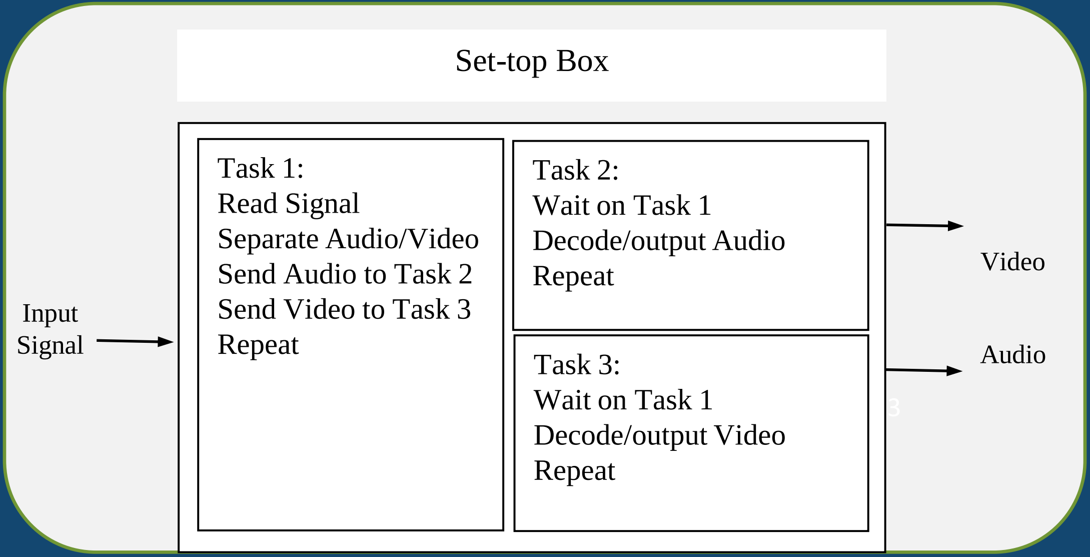
Concurrent process model easier

- Separate sequential programs (processes) for each task
- Programs communicate with each other



```
Task 1:
Read pulse
If pulse < Lo then
    Activate Siren
If pulse > Hi then
    Activate Siren
Sleep 1 second
Repeat
```

```
Task 2:
If B1/B2 pressed then
    Lo = Lo +/– 1
If B3/B4 pressed then
    Hi = Hi +/– 1
Sleep 500 ms
Repeat
```



```
Task 1:
Read Signal
Separate Audio/Video
Send Audio to Task 2
Send Video to Task 3
Repeat
```

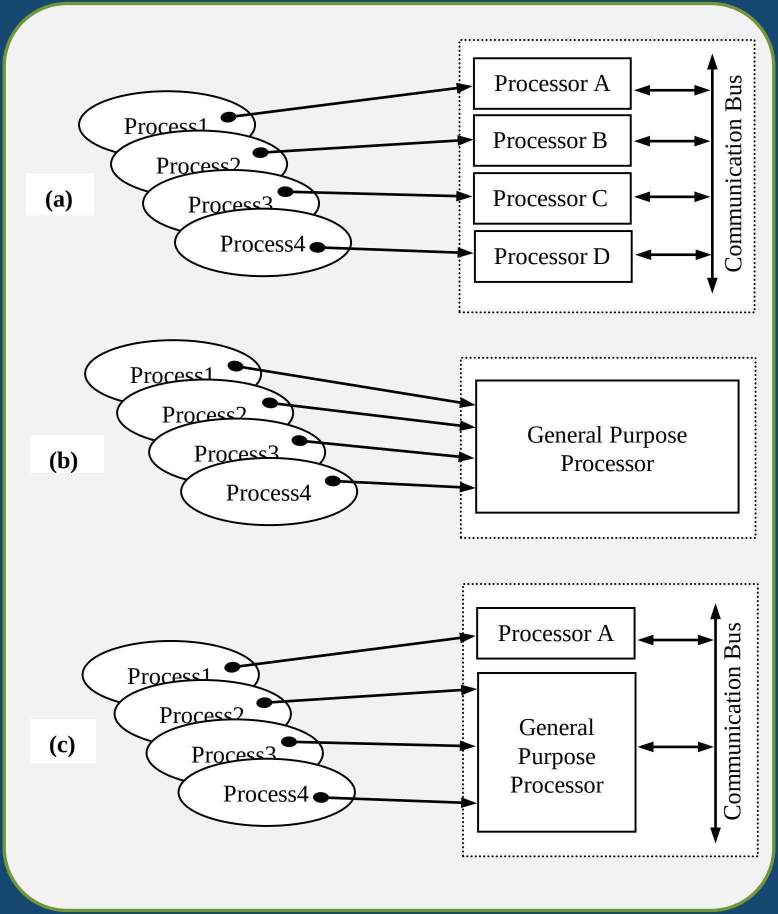
```
Task 2:
Wait on Task 1
Decode/output Audio
Repeat
```

```
Task 3:
Wait on Task 1
Decode/output Video
Repeat
```

## CONCURRENCY MANIFESTATIONS

- Multiple applications
  - Multiprogramming
- Structured application
  - Application can be a set of concurrent processes
- Operating-system structure
  - Operating system is a set of processes or threads

# CONCURRENT PROCESS MODELS: IMPLEMENTATION



a) Multiple processors, each executing one process
   1. True multitasking (parallel processing)
   2. General-purpose processors

      Expensive and in most cases not necessary
   3. Custom single-purpose processors

      More common

b) One general-purpose processor running all processes. Most processes don't use 100% of processor time. Can share processor time and still achieve necessary execution rates

c) Combination of (A) and (B). Multiple processes run on one general-purpose processor while one or more processes run on own single-purpose processor

## CHALLENGES WITH CONCURRENCY

- Sharing global resources
- Management of allocation of resources
- Programming errors difficult to locate

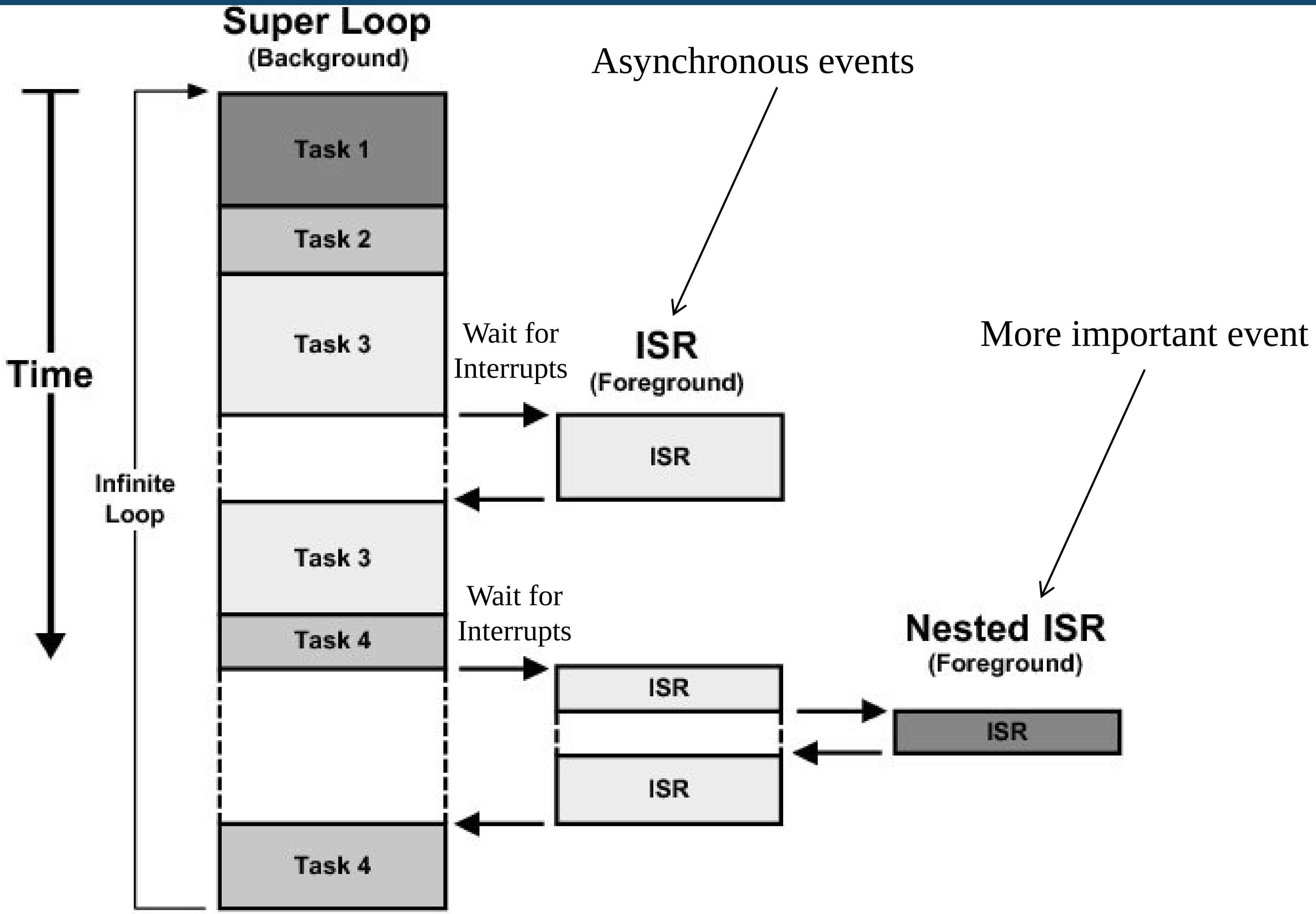
## OUTLINE

- Why concurrency?
- <u>Foreground / background vs. multi-tasking systems</u>
- Concurrent processes and communication
- Scheduling
- Bus scheduling

## FOREGROUND/BACKGROUND SYSTEMS

- Efficient for small systems of low complexity
- Infinite loop that call modules or tasks to perform the desired operations (also called task level or super-loop)
- Interrupt Service Routines (ISRs) handle asynchronous events (foreground also called ISR level)
  - Timer interrupts
  - I/O interrupts

# FOREGROUND/BACKGROUND SYSTEMS



## FOREGROUND/BACKGROUND SYSTEMS

- Critical tasks are handled by ISRs to ensure timeliness
- Information from ISR is not processed until the background routine gets its turn to execute. This is called task-level response.
- The worst-case task-level response depends on how long the background loop is
- High volume and low-cost microcontroller-based applications (e.g., microwaves, simple telephones,…) are designed as foreground/background systems

# FOREGROUND/BACKGROUND

```
/* Background */
void main (void) {
       Initialization;
       FOREVER {
                  Read analog inputs;
                  Read discrete inputs;
                  Perform monitoring functions;
                  Perform control functions;
                  Update analog outputs;
                  Update discrete outputs;
                  Scan keyboard;
                  Handle user interface;
                  Update display;
                  Handle communication requests;
                  Other...
                       }
}

/* Foreground */
ISR (void){
       Handle asynchronous event;
}
```

## FOREGROUND/BACKGROUND: ADVANTAGES

- Used in low cost embedded applications
- Memory requirements only depends on your application
- Single stack area for:
  - Function nesting
  - Local variables
  - ISR nesting
- Minimal interrupt latency for bare minimum embedded systems

## FOREGROUND/BACKGROUND: DISADVANTAGES

Background response time is the background execution time

- Non-deterministic, affected by if, for, while ...
- May not be responsive enough
- Changes as you change your code

## FOREGROUND/BACKGROUND: DISADVANTAGES

All “tasks” have the “same priority”!

– Code executes in sequence

– If an important event occurs it’s handled at the same priority as everything else!

– You may need to execute the same code often to avoid missing an event.

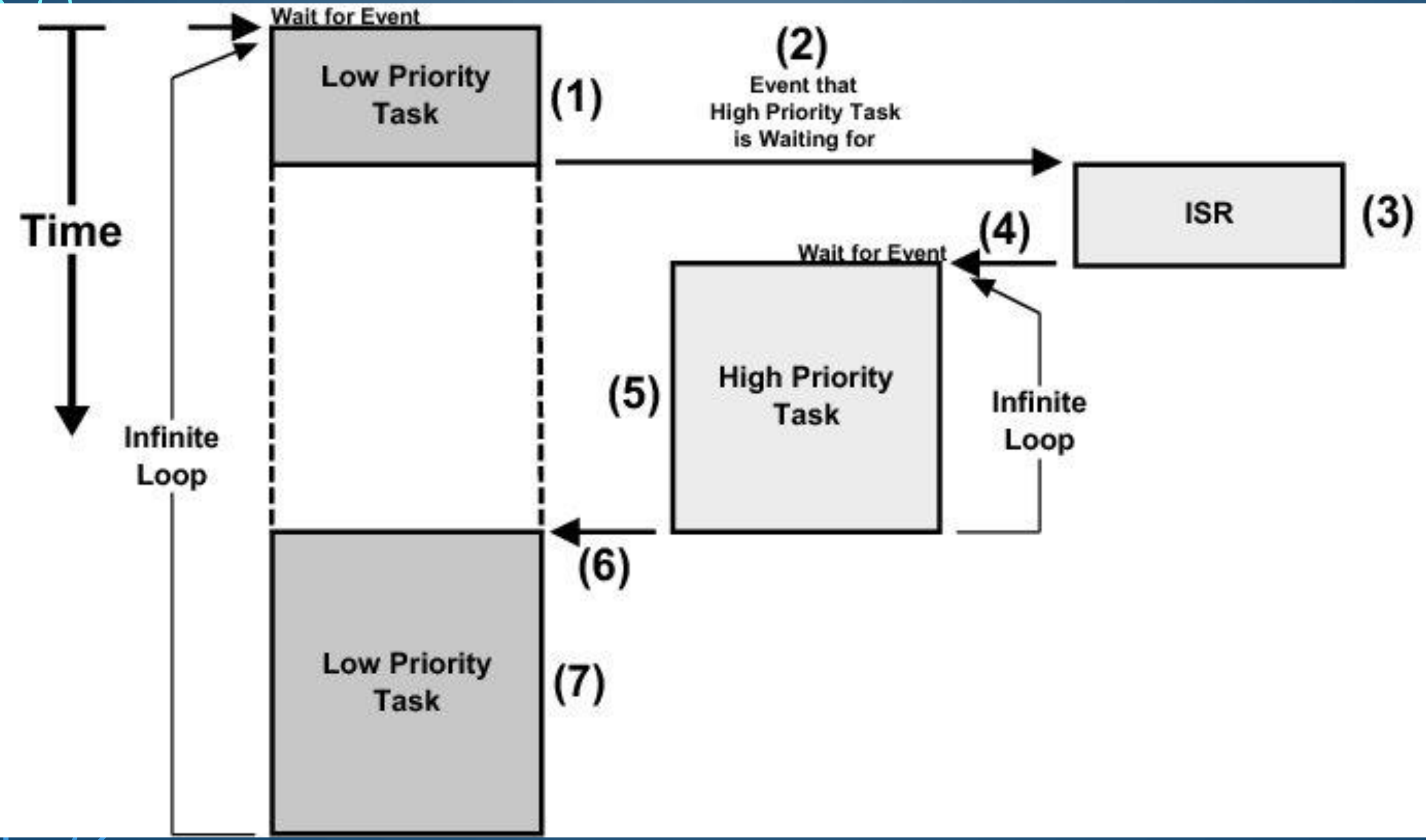
## FOREGROUND/BACKGROUND: DISADVANTAGES

- Code is harder to maintain and can become messy
  - Imagine the C program as the number of tasks increase!

## MIGRATE TO MULTI-TASKING SYSTEMS

- Each operation in the superloop/background is broken apart into a task, that by itself runs in infinite loop

## MIGRATE TO MULTI-TASKING SYSTEMS



## MULTI-TASKING SYSTEM

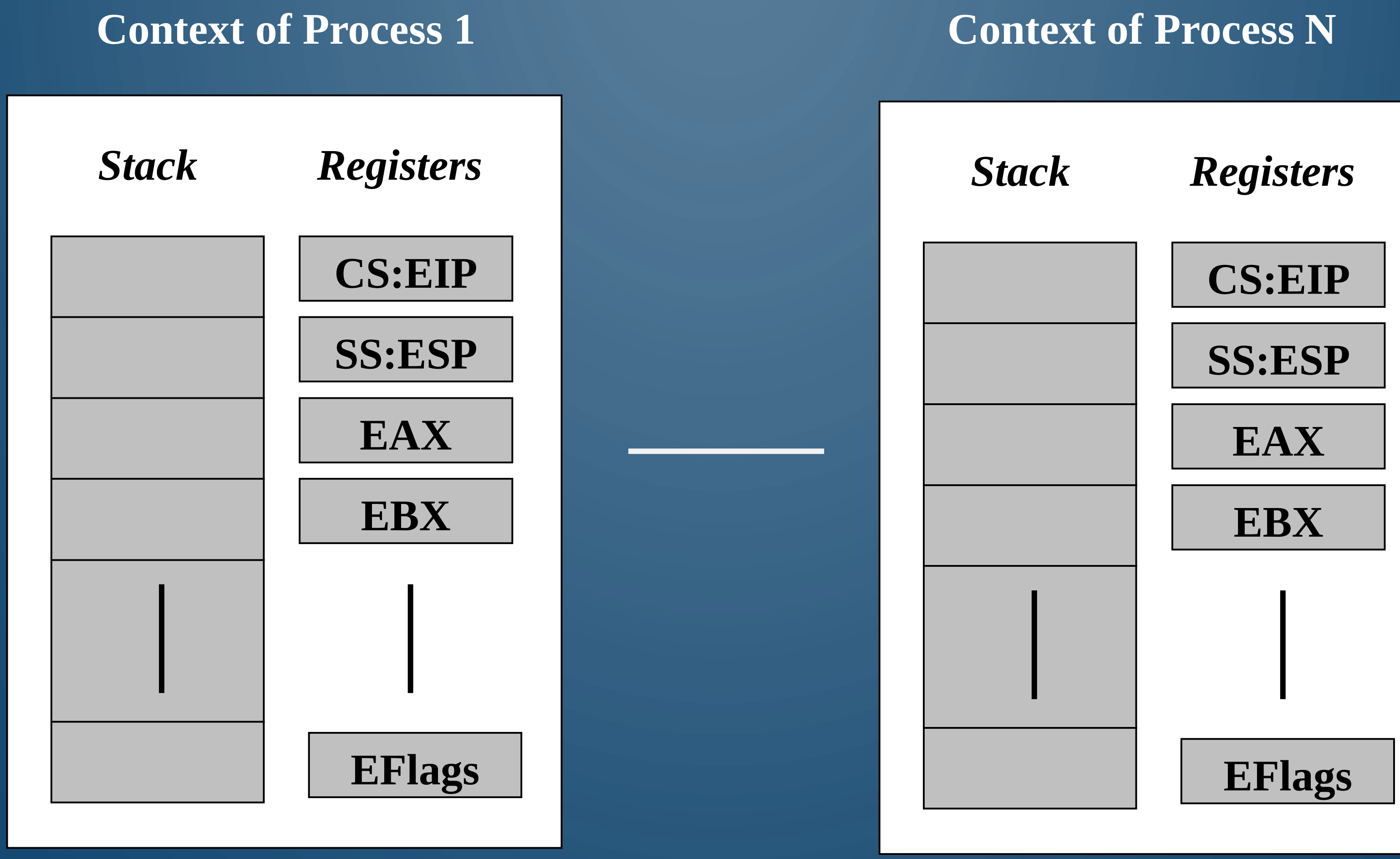
- Each task is a simple program that thinks it has the entire CPU to itself, and typically executed in an infinite loop.

- In the CPU only one task runs at any given time. This management --- scheduling and switching the CPU between several tasks --- is performed by the kernel of the real-time system

## OUTLINE

- Why concurrency?
- Foreground / background vs. multi-tasking systems
- <u>Concurrent processes and communication</u>
- Scheduling
- Bus scheduling

## EACH PROCESS MAINTAINS ITS OWN STACK AND REGISTER CONTENTS

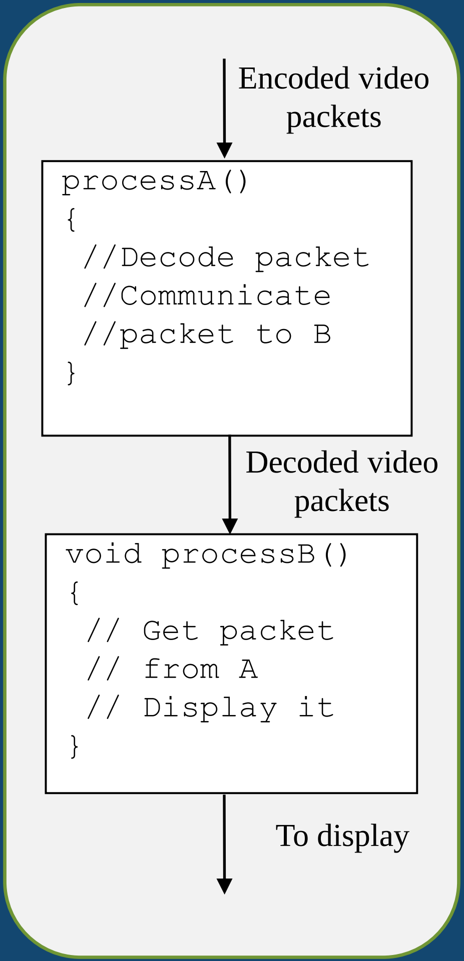


## CONTEXT SWITCHING

- Each process has its own stack and a special region of memory referred to as its *context*.
- A *context switch* from process "A" to process "B" first saves all CPU registers in context A, and then reloads all CPU registers from context B.
- Since CPU registers includes SS:ESP and CS:EIP, reloading context B reactivates process B's stack and returns to where it left off when it was last suspended.

# COMMUNICATION AMONG PROCESSES

- Processes need to communicate data and signals to solve their computation problem
  - Processes that do not communicate are just independent programs solving separate problems
- Basic example: producer/consumer
  - Process A produces data items, Process B consumes them
  - E.g., A decodes video packets, B display decoded packets on a screen
- How do we achieve this communication? Two basic methods:
  - Shared memory
  - Message passing



```
processA()
{
 //Decode packet
 //Communicate
 //packet to B
}
```

```
void processB()
{
 // Get packet
 // from A
 // Display it
}
```

# SHARED MEMORY

- Processes read and write shared variables
- No time overhead, easy to implement
- But, hard to use – mistakes are common

- Example: buggy producer(A)/consumer(B)
- Share *buffer*[*N*], *count (*# of valid data items in *buffer)*
- *processA* produces data items and stores in *buffer*
- *processB* consumes data items from *buffer*
- If *buffer* is full, processA must wait
- If *buffer* is empty, processB must wait

- Error when both update *count* concurrently. Say count is 3:
- *A* loads *count* from memory into register R1 (R1 = 3)
- *A* increments R1 (R1 = 4)
- *B* loads *count* from memory into register R2 (R2 = 3)
- *B* decrements R2 (R2 = 2)
- *A* stores R1 back to *count* in memory (*count =* 4)
- *B* stores R2 back to *count* in memory (*count* = 2)
- *count* now has incorrect value of 2

```
data_type buffer[N];
int count = 0;

void processA() {
  int i;
  while( 1 ) {
    produce(&data);
    while( count == N );/*loop*/
    buffer[i] = data;
    i = (i + 1) % N;
    count = count + 1;
  }
}

void processB() {
  int i;
  while( 1 ) {
    while( count == 0 );/*loop*/
    data = buffer[i];
    i = (i + 1) % N;
    count = count - 1;
    consume(&data);
  }
}

void main() {
  create_process(processA);
  create_process(processB);
}
```

# MESSAGE PASSING

- Data explicitly sent from one process to another
  - Sending process performs special operation, *send*
  - Receiving process must perform special operation, *receive*, to receive the data
  - Both operations must explicitly specify which process it is sending to or receiving from
- Safer model, but less flexible

```
void processA() {
  while( 1 ) {
    produce(&data)
    send(B, &data);
    /* region 1 */
    receive(B, &data);
    consume(&data);
  }
}
```

```
void processB() {
  while( 1 ) {
    receive(A, &data);
    transform(&data)
    send(A, &data);
    /* region 2 */
  }
}
```

## Back to Shared Memory: Mutual Exclusion

Certain sections of code should not be performed concurrently

- Critical section: section of code where simultaneous updates, by multiple processes to a shared memory location, can occur

When a process enters the critical section, all other processes must be locked out until it leaves the critical section.

Mutex:

- A shared object used for locking and unlocking segment of shared data
- Disallows read/write access to memory it guards
- Multiple processes can perform lock operation simultaneously, but only one process will acquire lock
- All other processes trying to obtain lock will be put in blocked state until unlock operation performed by acquiring process when it exits critical section
- These processes will then be placed in runnable state and will compete for lock again

# SHARED MEMORY (REV.)

The primitive *mutex* is used to ensure critical sections are executed in mutual exclusion of each other

Following the same execution sequence as before:

- *A*/*B* execute *lock* operation on *count_mutex*
- Either *A* **or** *B* will acquire *lock*
    - Say *B* acquires it
    - *A* will be put in blocked state
- *B* loads *count* (*count* = 3) from memory into register R2 (R2 = 3)
- *B* decrements R2 (R2 = 2)
- *B* stores R2 back to *count* in memory (*count* = 2)
- *B* executes *unlock* operation
    - *A* is placed in runnable state again
- *A* loads *count* (*count* = 2) from memory into register R1 (R1 = 2)
- *A* increments R1 (R1 = 3)
- *A* stores R1 back to *count* in memory (*count* = 3)

*Count* now has correct value of 3

```
data_type buffer[N];
int count = 0;
mutex count_mutex;

void processA() {
  int i;
  while( 1 ) {
    produce(&data);
    while( count == N );/*loop*/
    buffer[i] = data;
    i = (i + 1) % N;
    count_mutex.lock();
    count = count + 1;
    count_mutex.unlock();
  }
}

void processB() {
  int i;
  while( 1 ) {
    while( count == 0 );/*loop*/
    data = buffer[i];
    i = (i + 1) % N;
    count_mutex.lock();
    count = count - 1;
    count_mutex.unlock();
    consume(&data);
  }
}

void main() {
  create_process(processA);
  create_process(processB);
}
```

# A COMMON PROBLEM IN CONCURRENT PROGRAMMING: DEADLOCK

Deadlock: A condition where 2 or more processes are blocked waiting for the other to unlock critical sections of code

- Both processes are then in blocked state
- Cannot execute unlock operation so will wait forever

Example code has 2 different critical sections of code that can be accessed simultaneously

- 2 locks needed (mutex1, mutex2)
- Following execution sequence produces deadlock
  - *A* executes lock operation on *mutex1* (and acquires it)
  - *B* executes lock operation on *mutex2*( and acquires it)
  - *A*/*B* both execute in critical sections 1 and 2, respectively
  - *A* executes lock operation on *mutex2*
    - *A* blocked until *B* unlocks *mutex2*
  - *B* executes lock operation on *mutex1*
    - *B* blocked until *A* unlocks *mutex1*
  - DEADLOCK!

```
mutex mutex1, mutex2;

void processA() {
  while( 1 ) {
    …
    mutex1.lock();
    /* critical section 1 */
    mutex2.lock();
    /* critical section 2 */
    mutex2.unlock();
    /* critical section 1 */
    mutex1.unlock();
  }
}

void processB() {
  while( 1 ) {
    …
    mutex2.lock();
    /* critical section 2 */
    mutex1.lock();
    /* critical section 1 */
    mutex1.unlock();
    /* critical section 2 */
    mutex2.unlock();
  }
}
```

One deadlock elimination protocol requires locking of numbered mutexes in increasing order. This is typically combined with two-phase locking (2PL):

- Acquire locks in 1st phase only, release locks in 2nd phase

## SUMMARY: MULTIPLE PROCESSES SHARING SINGLE PROCESSOR

Manually rewrite processes as a single sequential program

- Less overhead (no operating system)
- More complex/harder to maintain
- Ok for simple examples, but extremely difficult for complex examples

Can use multitasking operating system

- Much more common
- Operating system schedules processes, allocates storage, interfaces to peripherals, etc.
- Real-time operating system (RTOS) can guarantee execution rate constraints are met
- Describe concurrent processes with languages having built-in processes (Java, Ada, etc.) or a sequential programming language with library support for concurrent processes (C, C++, etc. using POSIX threads for example)

## OUTLINE

- Why concurrency?
- Foreground / background vs. multi-tasking systems
- Concurrent processes and communication
- Scheduling
- Bus scheduling

## IMPLEMENTATION: PROCESS SCHEDULING

- Must meet timing requirements when multiple concurrent processes implemented on single general-purpose processor
- Scheduler
  - Special process that decides when and for how long each process is executed
  - Implemented as preemptive or non-preemptive scheduler

## PREEMPTIVE VS NON-PREEMPTIVE

- Time-Preemptive
  - Determines how long a process executes before preempting to allow another process to execute
  - Time quantum: predetermined amount of execution time preemptive scheduler allows each process (may be 10s to 100s of milliseconds long)
  - Determines which process will be next to run

- Non-preemptive
  - Only determines which process is next after current process finishes execution

## STATIC VS DYNAMIC SCHEDULING

Static (off-line)

- Complete a priori (i.e., before hand) knowledge of the task set and its constraints is available
- Suitable for hard/safety-critical system

Dynamic (on-line)

- Partial task set knowledge, requires runtime predictions
- Suitable for soft/best-effort systems, mixed criticality systems

## SCHEDULING APPROACHES

- Cyclic executives
- Fixed priority scheduling
  - RM - Rate Monotonic
  - DM - Deadline Monotonic Scheduling
- Dynamic priority scheduling
  - EDF - Earliest Deadline First
  - LSF - Least Slack First

# CYCLIC EXECUTIVE

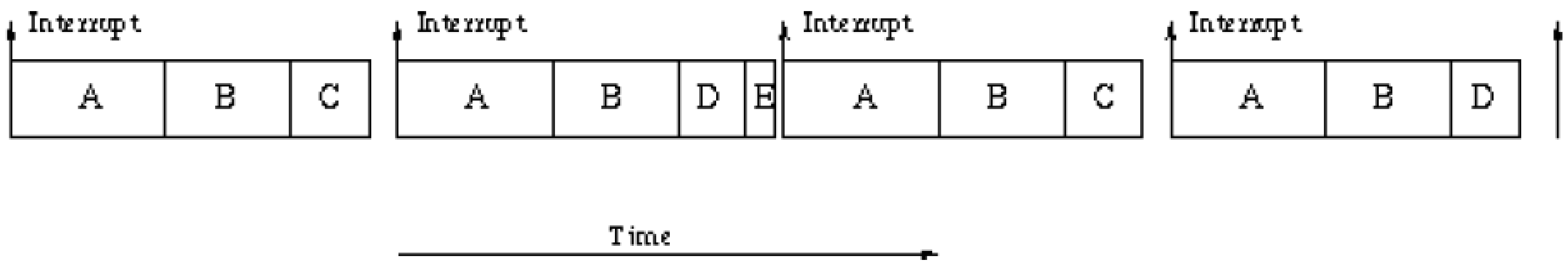
| Process | Period | Comp. Time |
|---|---|---|
| A | 25 | 10 |
| B | 25 | 8 |
| C | 50 | 5 |
| D | 50 | 4 |
| E | 100 | 2 |

```
loop
    Wait_For_Interrupt;
    Procedure_For_A;
    Procedure_For_B;
    Procedure_For_C;

    Wait_For_Interrupt;
    Procedure_For_A;
    Procedure_For_B;
    Procedure_For_D;
    Procedure_For_E;

    Wait_For_Interrupt;
    Procedure_For_A;
    Procedure_For_B;
    Procedure_For_C;

    Wait_For_Interrupt;
    Procedure_For_A;
    Procedure_For_B;
    Procedure_For_D;
end loop;
```



## PRIORITY-BASED SCHEDULING

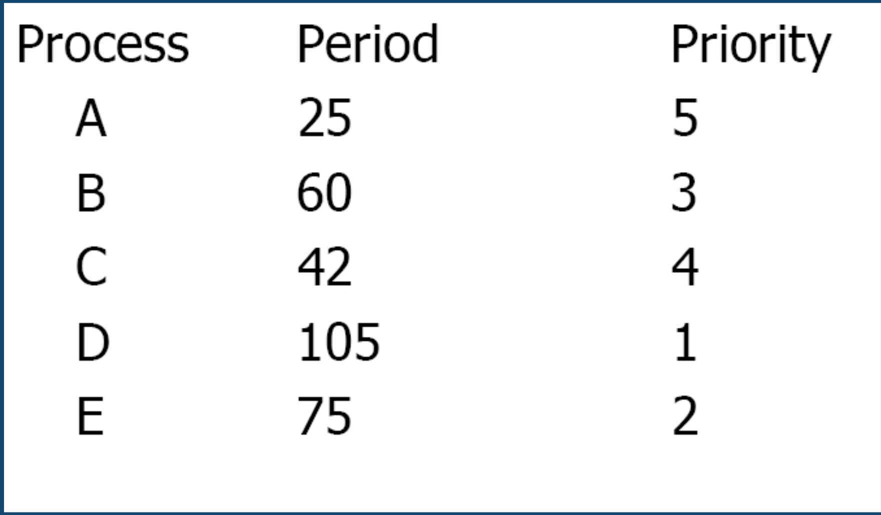
- Every task has an associated priority
- Run task with the highest priority
  - At every scheduling decision moment
- Examples:
  - Rate Monotonic (RM)
    - Static priority assignment
  - Earliest Deadline First (EDF)
    - Dynamic priority assignment
  - And many others …

## SCHEDULABILITY TEST

- Test to determine whether a feasible schedule exists
- Sufficient
    - + if test is passed, then tasks are definitely schedulable
    - - if test is not passed, we don’t know
- Necessary
    - + if test is passed, we don’t know
    - - if test is not passed, tasks are definitely not schedulable
- Exact
    - sufficient & necessary at the same time

# RATE MONOTONIC

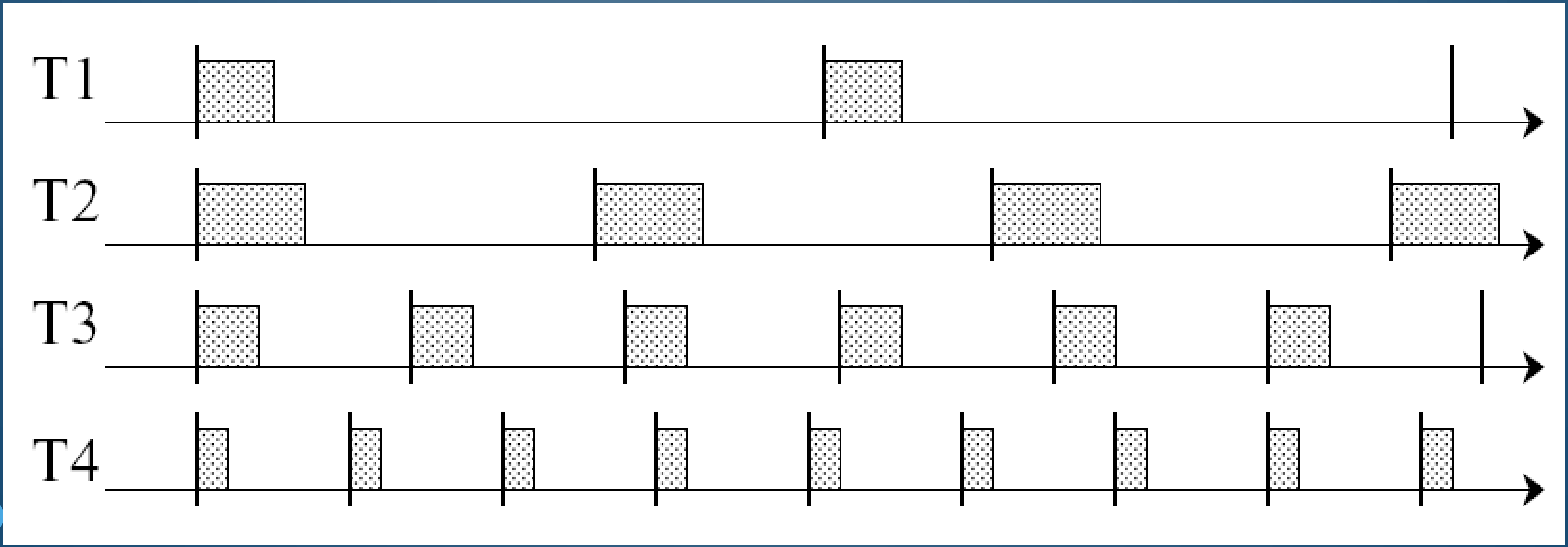
- Each process is assigned a (unique) priority based on its period; the shorter the period, the higher the priority
- Assumes the "Simple task model"
- Fixed priority scheduling
- Pre-emptive
  - Unless stated otherwise



| Process | Period | Priority |
|---|---|---|
| A | 25 | 5 |
| B | 60 | 3 |
| C | 42 | 4 |
| D | 105 | 1 |
| E | 75 | 2 |

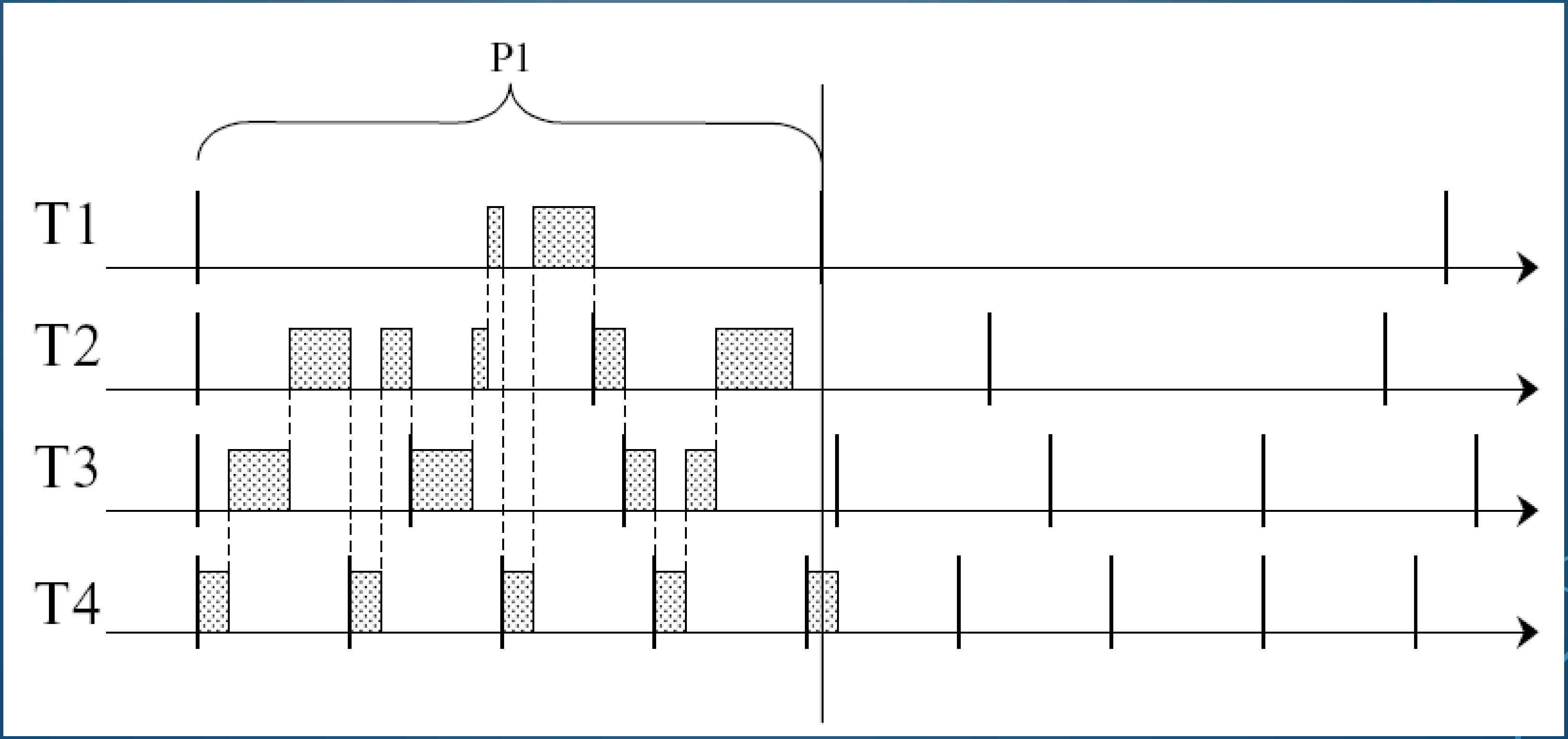
## EXAMPLE 1

Assume we have the following task set (not scheduled yet ...)



## EXAMPLE 1 (CONT'D)

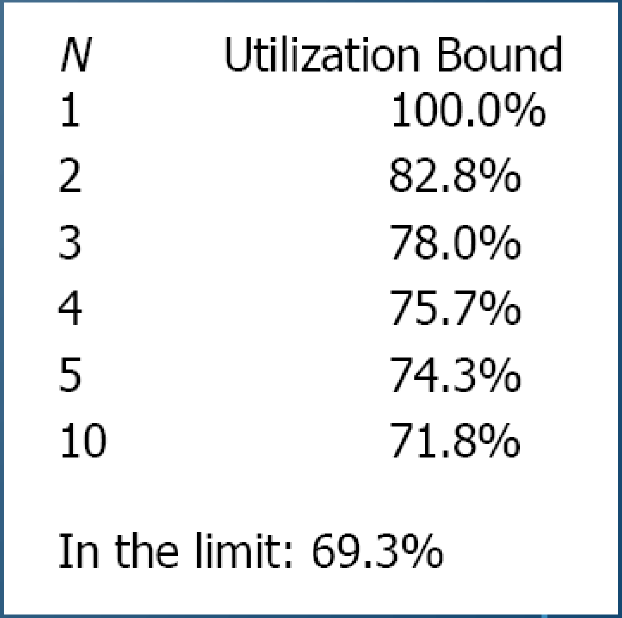
Scheduled with RM



## SCHEDULABILITY TEST FOR RM

Sufficient, but not necessary:

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq N\left(2^{1/N} - 1\right)$$



| *N* | Utilization Bound |
|---|---|
| 1 | 100.0% |
| 2 | 82.8% |
| 3 | 78.0% |
| 4 | 75.7% |
| 5 | 74.3% |
| 10 | 71.8% |

In the limit: 69.3%

Necessary, but not sufficient:

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$$

## EXAMPLE 2

| Taskset | P1 | P2 | P3 |
|---|---|---|---|
| Period (Ti) | 20 | 50 | 30 |
| WCET (Ci) | 7 | 10 | 5 |

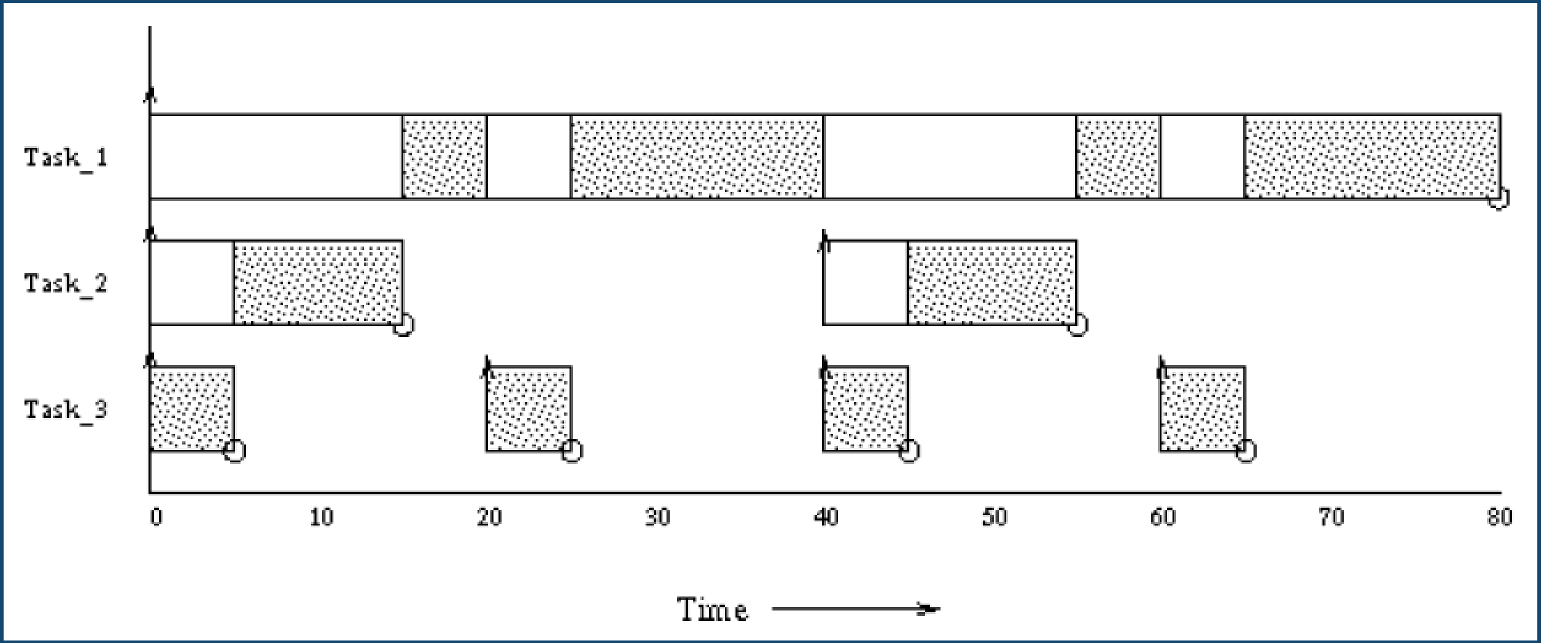
Is this schedulable?

## EXAMPLE 3

Taskset

| | Period | Comp. Time | Priority | Utilization |
|---|---|---|---|---|
| Task_1 | 80 | 40 | 1 | 0.50 |
| Task_2 | 40 | 10 | 2 | 0.25 |
| Task_3 | 20 | 5 | 3 | 0.25 |

Gantt chart:



## OPTIMALITY OF RM

Rate Monotonic is optimal among fixed priority schedulers if we assume the "Simple Process Model" for the tasks (e.g., no resource sharing, deadlines equal to periods, free context switch)

## WHAT TO DO IF NOT SCHEDULABLE

- Change the task set utilisation
    - by reducing $C_i$
        - code optimisation
        - faster processor

- Increase $T_i$ for some process
    - If your program and environment allows it

## RM CHARACTERISTICS

- Easy to implement.
- Drawback:
  - May not give a feasible  schedule even if processor is idle at some points.

## EARLIEST DEADLINE FIRST (EDF)

- Always runs the process that is closest to its deadline.
- Dynamic priority scheduling
  - Evaluated at run-time
  - What are the events that should trigger a priority re-evaluation?
- Assumes the “Simple task model” (e.g., no resource sharing, deadlines equal to periods, free context switch)
  - Actually more relaxed: $D_i < T_i$
- Pre-emptive
  - Unless stated otherwise

## SCHEDULABILITY TEST FOR EDF

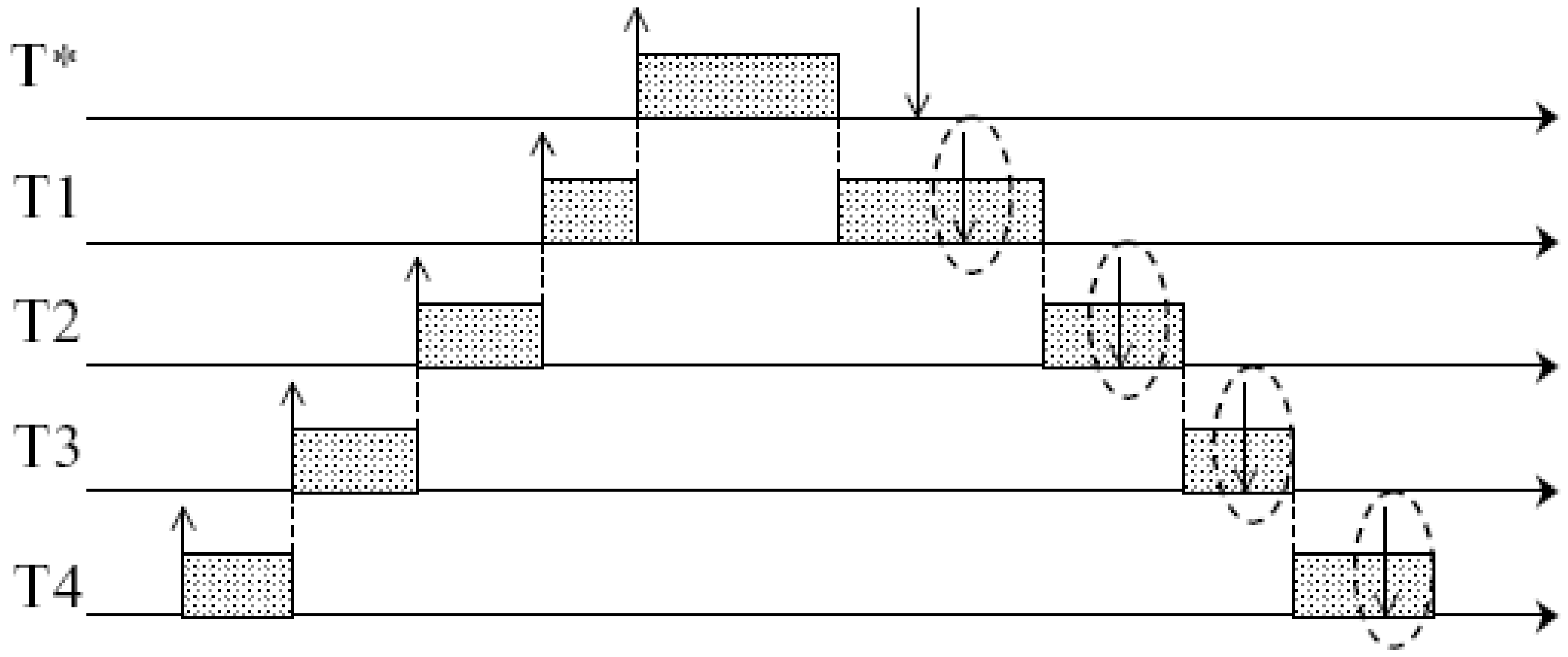
Utilisation test: Necessary and sufficient (exact!)

$$\sum_{i=1}^{N} \frac{C_i}{T_i} \leq 1$$

## OPTIMALITY OF EDF

EDF is optimal among dynamic priority schedulers if we assume the “Simple Process Model” for the tasks

# DOMINO EFFECT



Domino effect!!!

## EDF VS. RM

- EDF can handle tasksets with higher processor utilisation.
- EDF has simpler exact analysis
- RMS can be implemented to run faster at run-time
  - Depends on the OS
  - But they usually like fixed priorities more

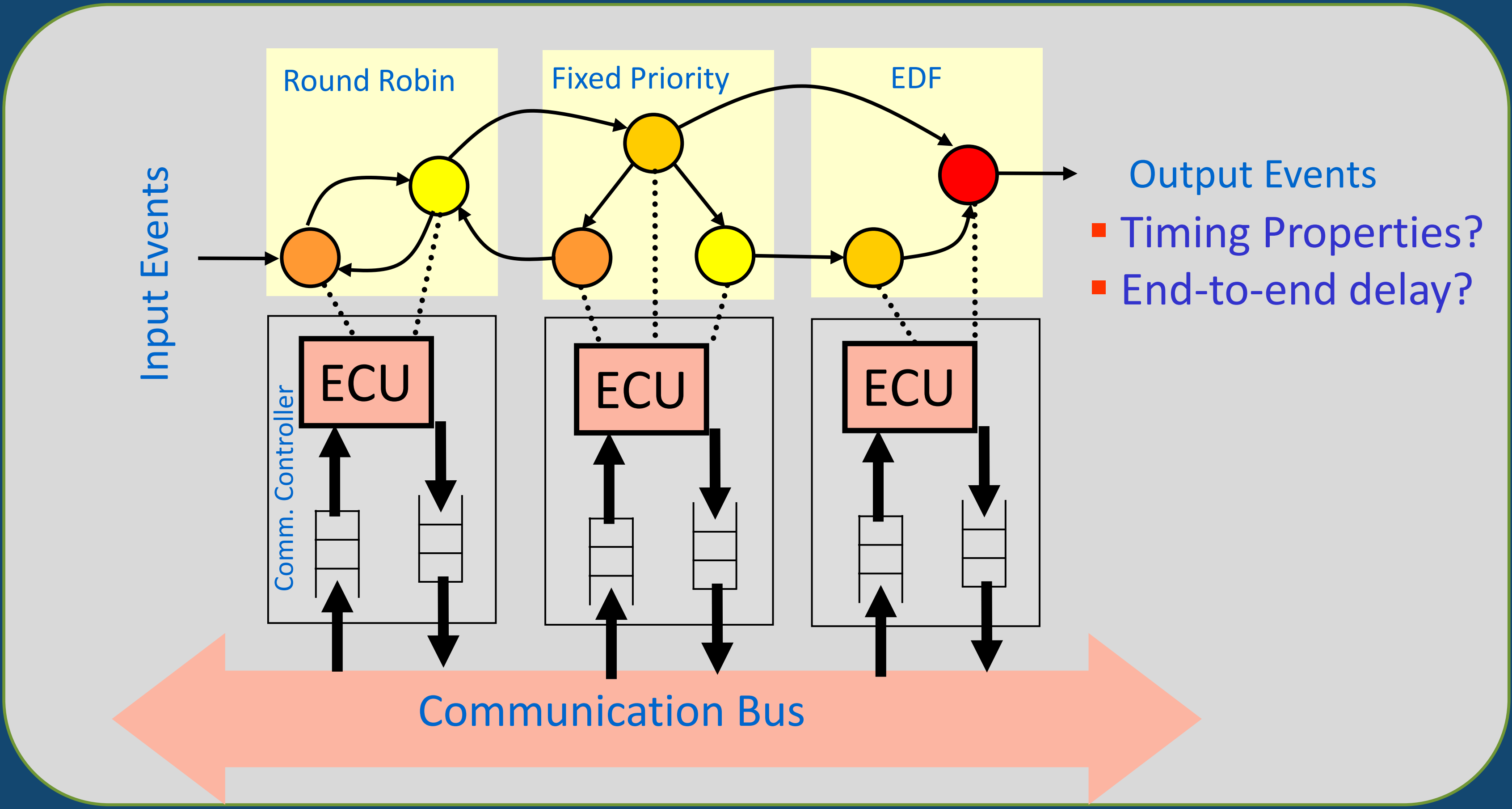
## OUTLINE

- Why concurrency?
- Foreground / background vs. multi-tasking systems
- Concurrent processes and communication
- Scheduling
- Bus scheduling

## BUS SCHEDULING

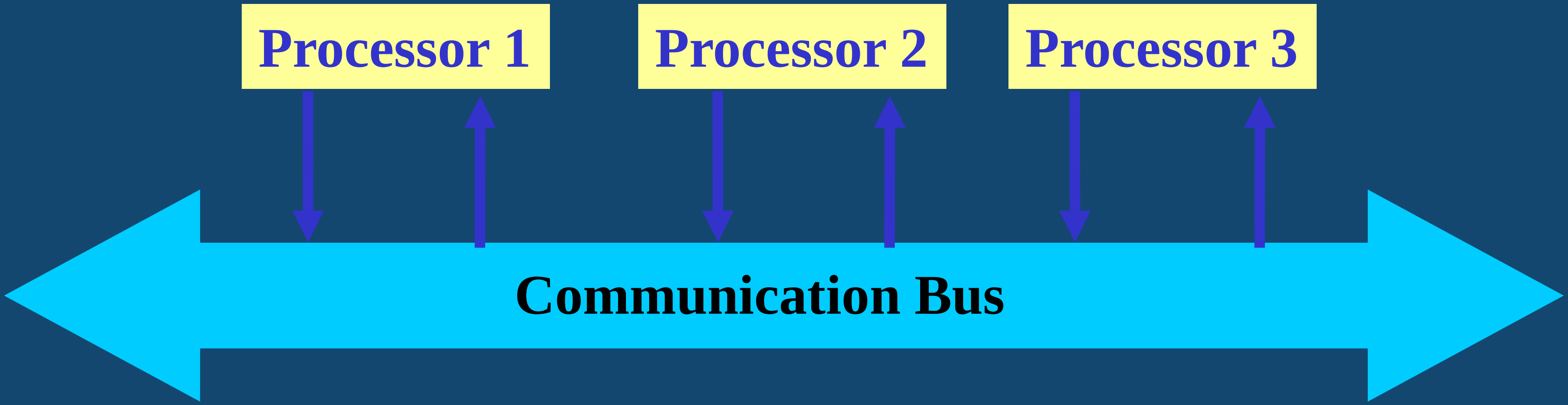
- So far we have studied the scheduling analysis on *one* processor
- However, as systems become more complex, multiple processors exist on a system
- MPSoCs in mobile devices, automotive electronics
- The different processors exchange messages over a *communication bus*!

## SYSTEM-LEVEL TIMING ANALYSIS PROBLEM



- Tasks have different activation rates and execution demands
- Each computation/communication element has a different scheduling/arbitration policy

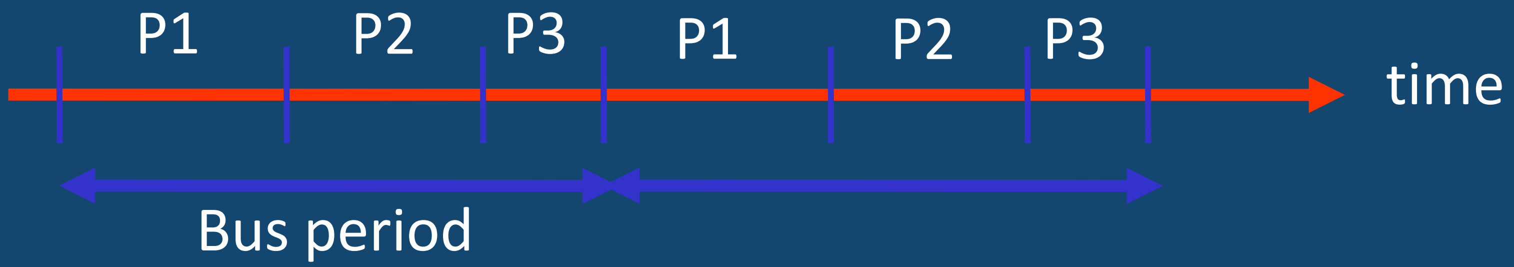
## BUS ARBITRATION POLICIES



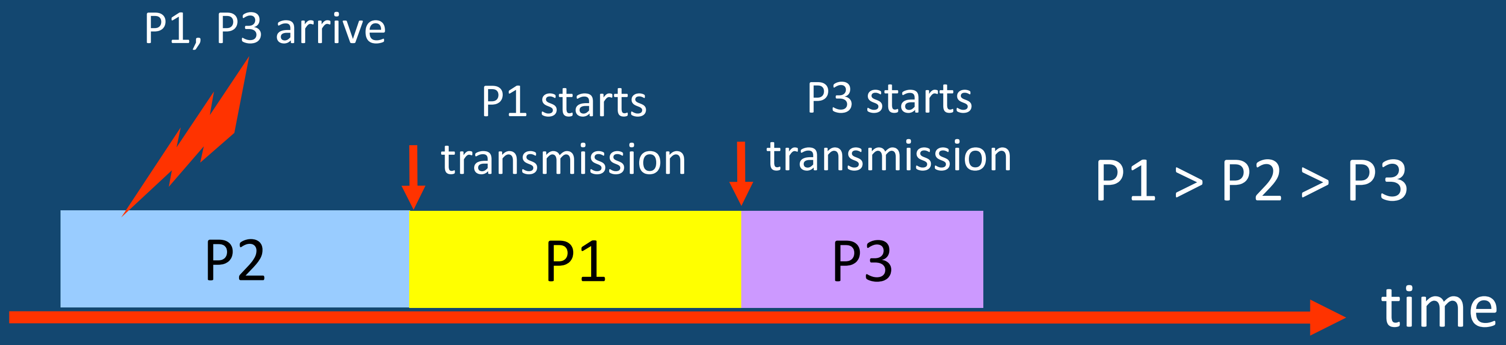
- When multiple processors want to transmit data at the same time, how is the contention resolved?
  - Using a bus arbitration policy, i.e. determine who gets priority
  - Examples of arbitration policies
    - Time Division Multiple Access, Round Robin, Fixed Priority …

## TIME/EVENT-TRIGGERED ARBITRATION

Time-triggered arbitration policy

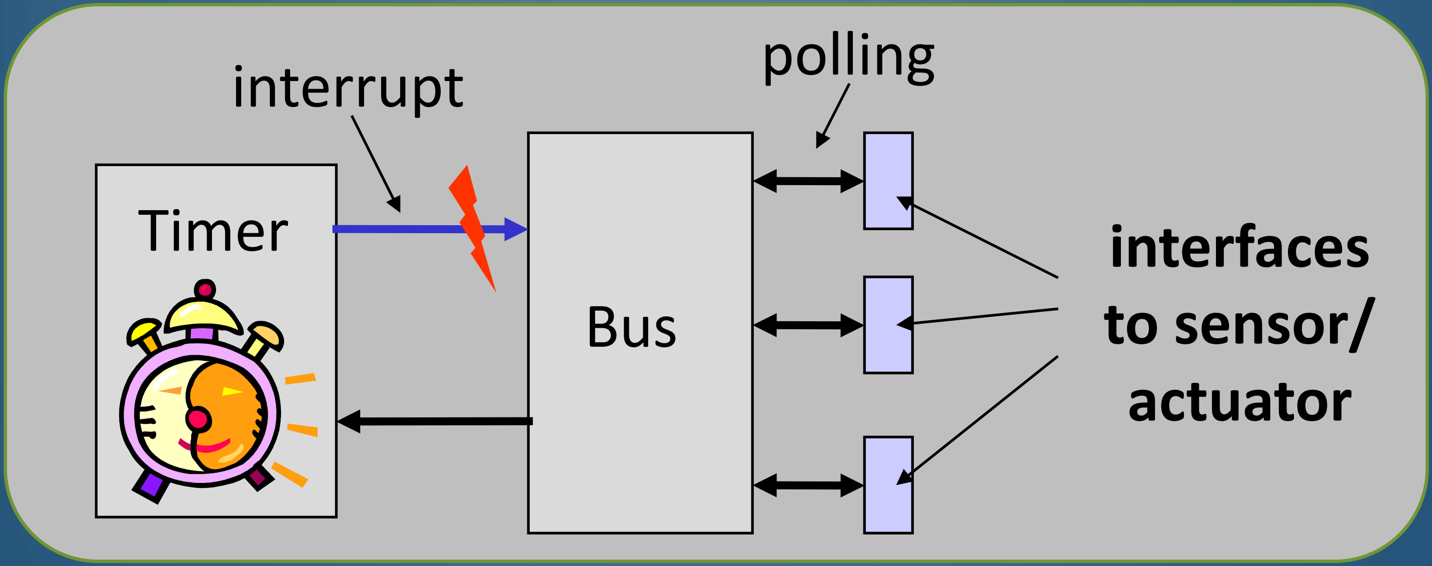


(Non preemptive) Event-triggered arbitration policy



P1 > P2 > P3

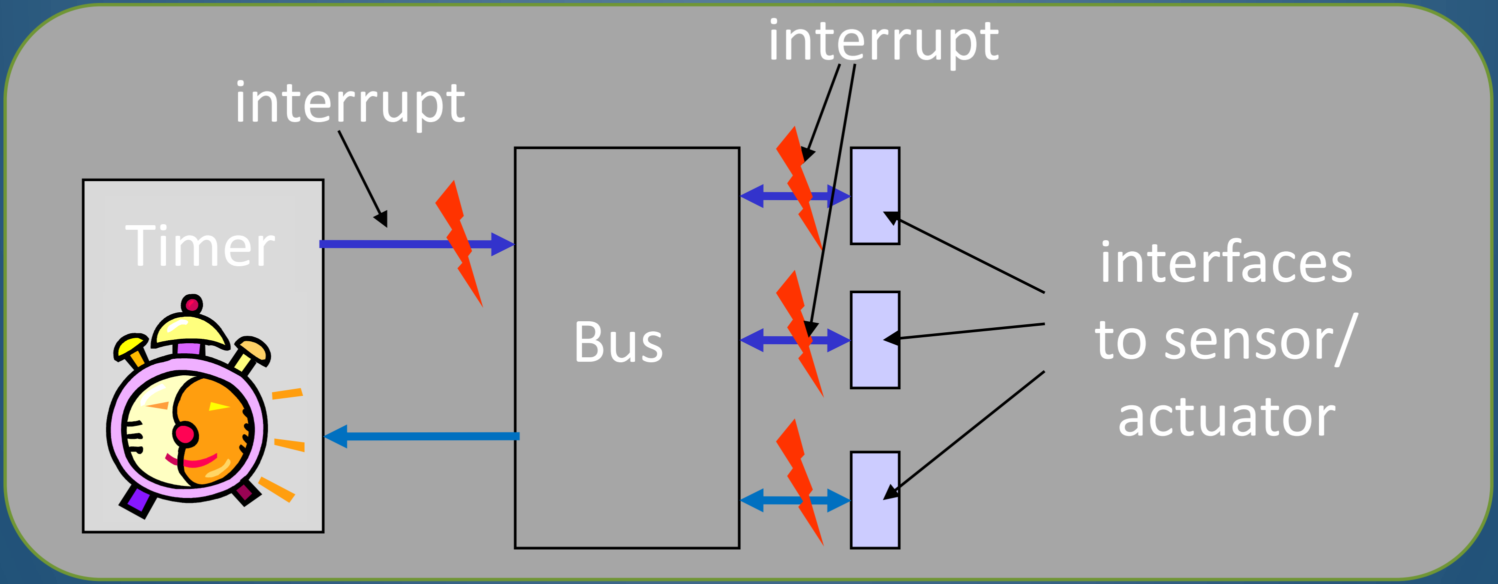
# BUS ARBITRATION POLICIES



**Time-Triggered Policy:**

- Only interrupts from the timer are allowed
- Events CANNOT interrupt
- Interaction with environment through polling
- Schedule is computed offline, deterministic behavior at runtime
- Example: Time Division Multiple Access (TDMA) policy

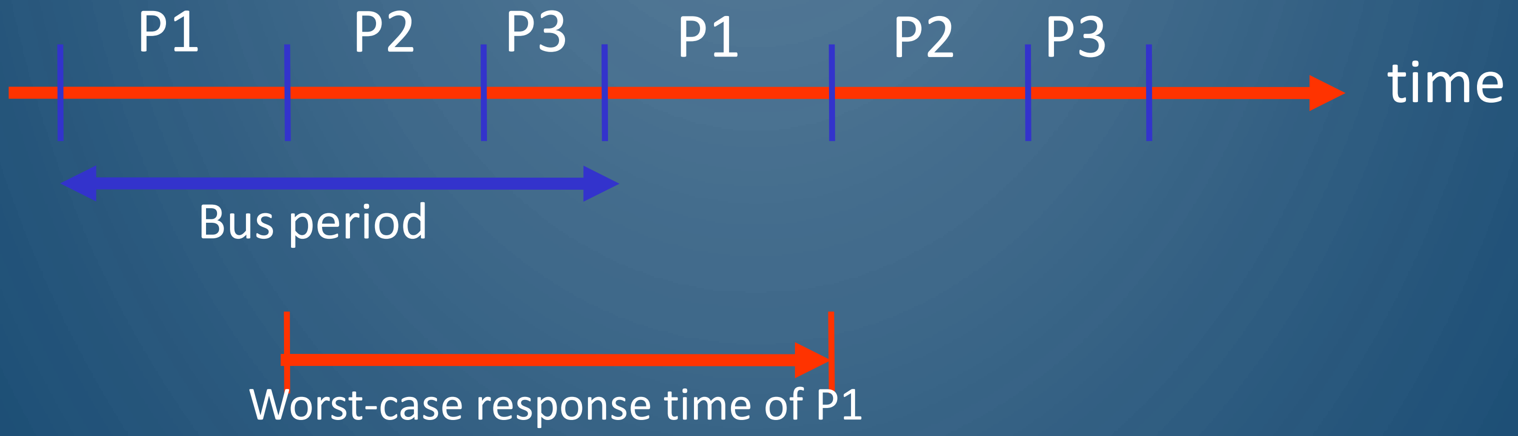
# BUS ARBITRATION POLICIES



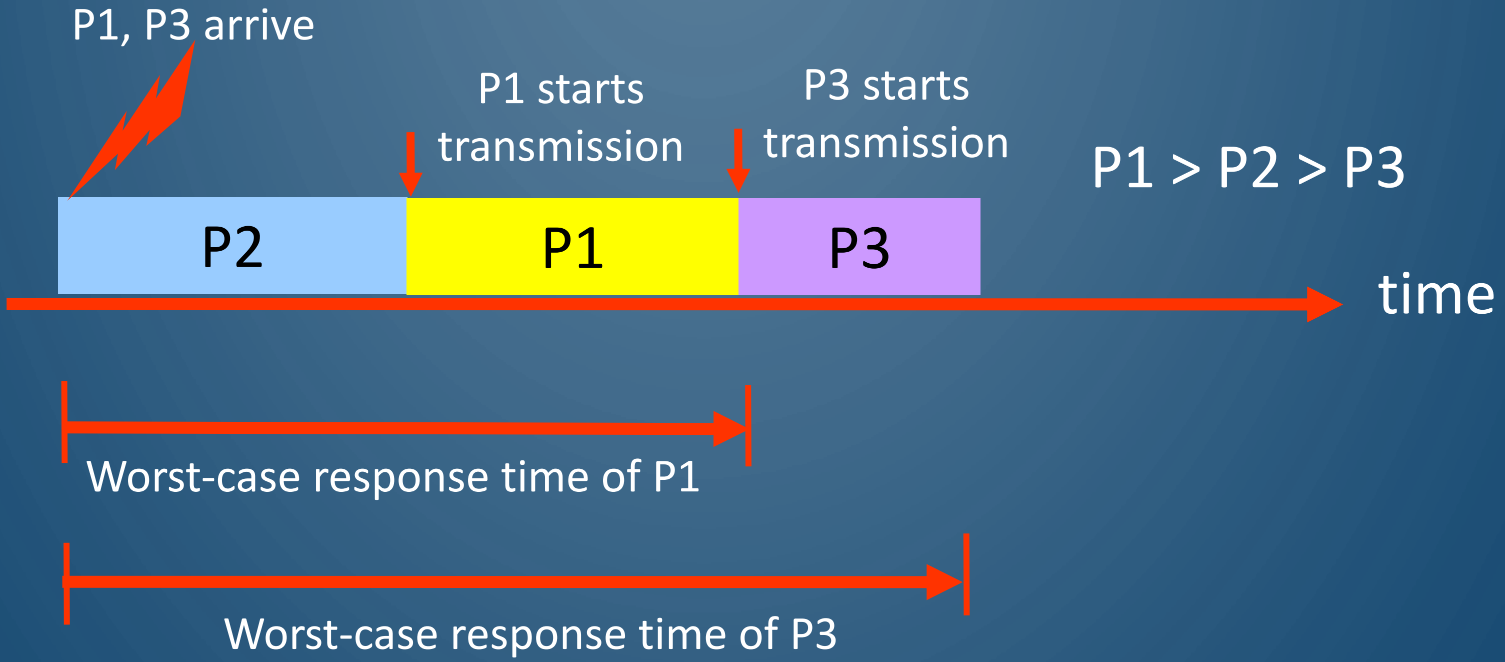
**Event-Triggered Policy:**

- Interrupts can be from the timer or from external events
- Interaction with environment through interrupts
- Schedule is dynamic and adaptive
- Response times ‘can’ be unpredictable
- Example: Fixed Priority scheduling policy

## COMPUTING RESPONSE TIMES IN TIME-TRIGGERED SYSTEMS



## COMPUTING RESPONSE TIMES IN EVENT-TRIGGERED SYSTEMS



## TWO WELL-KNOWN BUS PROTOCOLS

- Time-Triggered Bus Protocols:
  - Time-Triggered Protocol (TTP) – used in avionics
  - Based on Time Division Multiple Access (TDMA) policy

- Event-Triggered Bus Protocols:
  - Controller Area Network (CAN) – widely used for chassis control systems and power train communication
  - Based on fixed priority scheduling policy

## TIME-TRIGGERED VS EVENT-TRIGGERED: SUMMARY

- Both have their advantages and disadvantages

| | Time-Triggered | Event-Triggered |
|---|---|---|
| Response Times | ✕ | ☺ |
| Bus Utilization | ✕ | ☺ |
| Flexibility | ✕ | ☺ |
| Composability | ☺ | ✕ |