Concurrency

Lecture outline

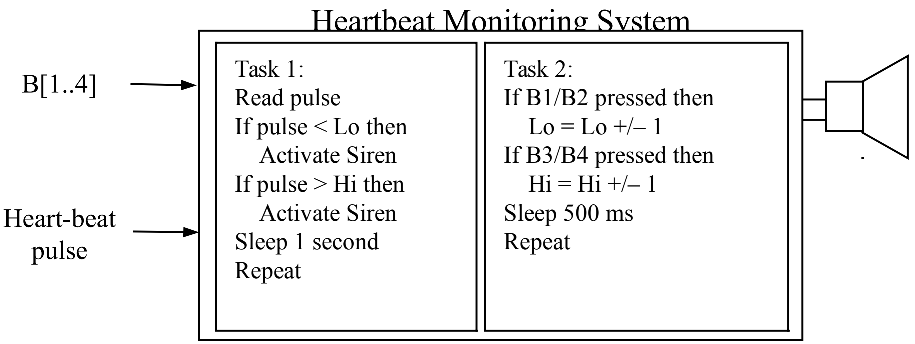
- Why concurrency?
- Foreground/background vs. multi-tasking systems
- Concurrent processes
  - Communication: message passing vs. shared memory
- Scheduling
- Bus scheduling

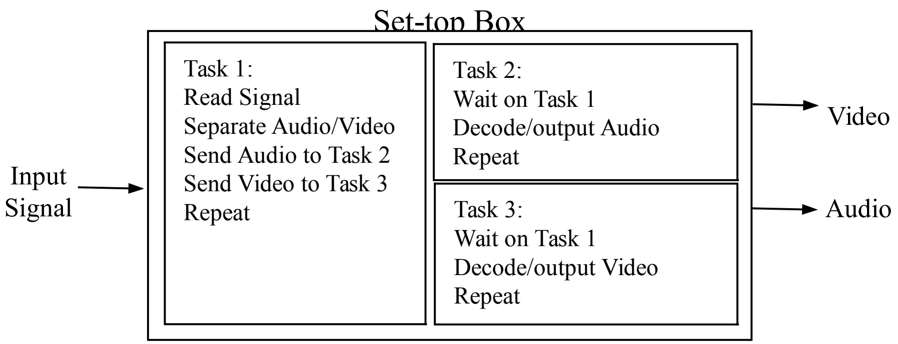
Lecture outline

- **Why concurrency?**
- Foreground/background vs. multi-tasking systems
- Concurrent processes
  - Communication: message passing vs. shared memory
- Scheduling
- Bus scheduling

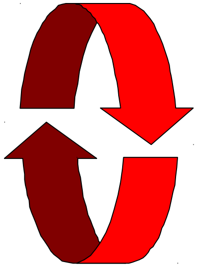
Why concurrency?

- Consider two examples having 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

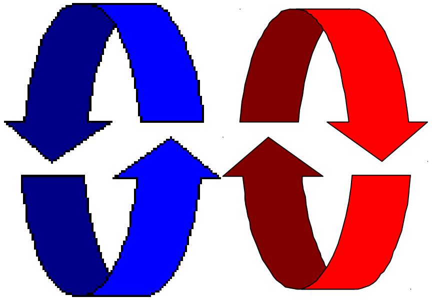




Concurrent Programs



A **sequential** program has a single thread of control.



A **concurrent** program has multiple threads of control allowing it perform multiple computations “in parallel” and to control multiple external activities which occur at the same time.

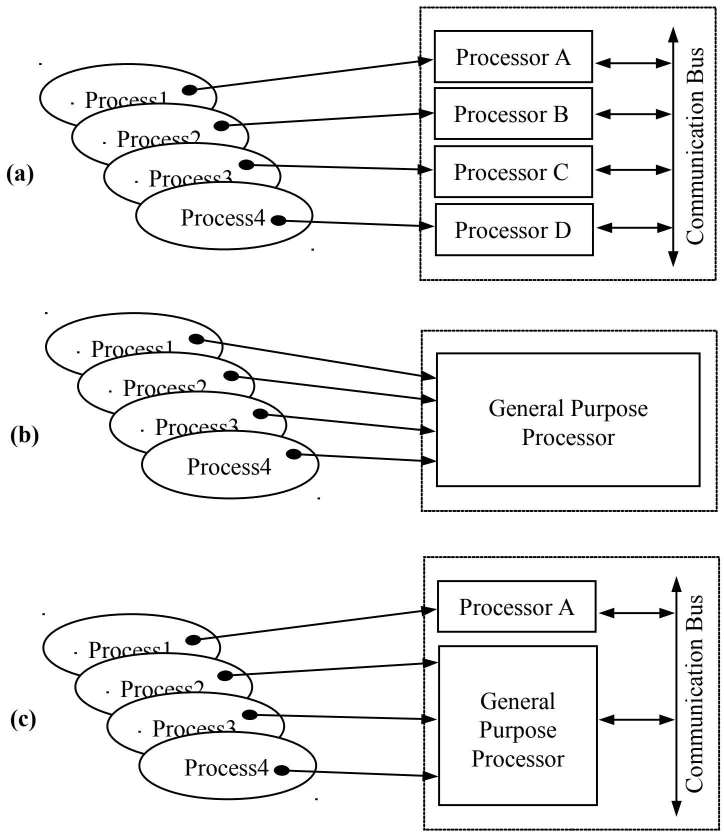
[Magee and Kramer 2006]

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 model: implementation

- Can use single and/or general-purpose processors
- (a) Multiple processors, each executing one process
  - True multitasking (parallel processing)
  - General-purpose processors
    - Use programming language like C and compile to instructions of processor
    - Expensive and in most cases not necessary
  - 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

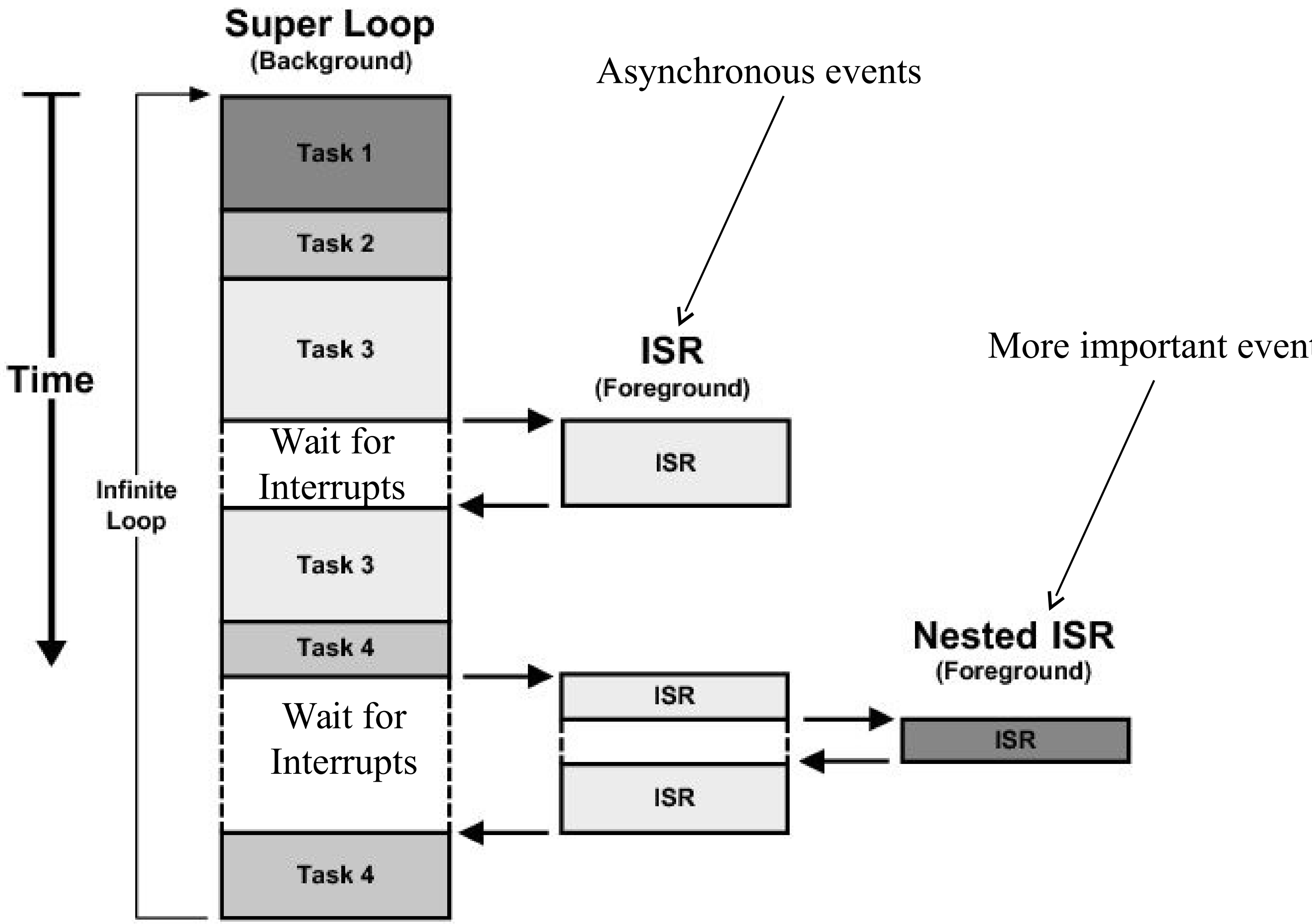
- Sharing global resources
- Management of allocation of resources
- Programming errors difficult to locate

Lecture outline

- Why concurrency?
- **Foreground/background vs. multi-tasking systems**
- Concurrent processes
  - Communication: message passing vs. shared memory
- 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 superloop)
- Interrupt Service Routines (ISRs) handle asynchronous events (foreground also called ISR level)
  - Timer interrupts
  - I/O interrupts


Super Loop
(Background)
Asynchronous events
Time
Infinite Loop
Task 1
Task 2
Task 3
ISR
(Foreground)
More important event
Wait for Interrupts
ISR
Task 3
Task 4
Nested ISR
(Foreground)
ISR
Wait for Interrupts
ISR
ISR
Task 4

Foreground/Background systems

- Critical tasks are handled by ISRs to ensure they perform in timely fashion
- Information for a background module that makes an ISR available is not processed until the background routine gets its turn to execute. This is called task-level response.
- The worst-case task-level response is depends on how long the background loop takes to execute and since the execution time is not constant, this is difficult to predict.
- High volume and low-cost microcontroller-based applications (e.g., microwaves, telephones,…) are designed are 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

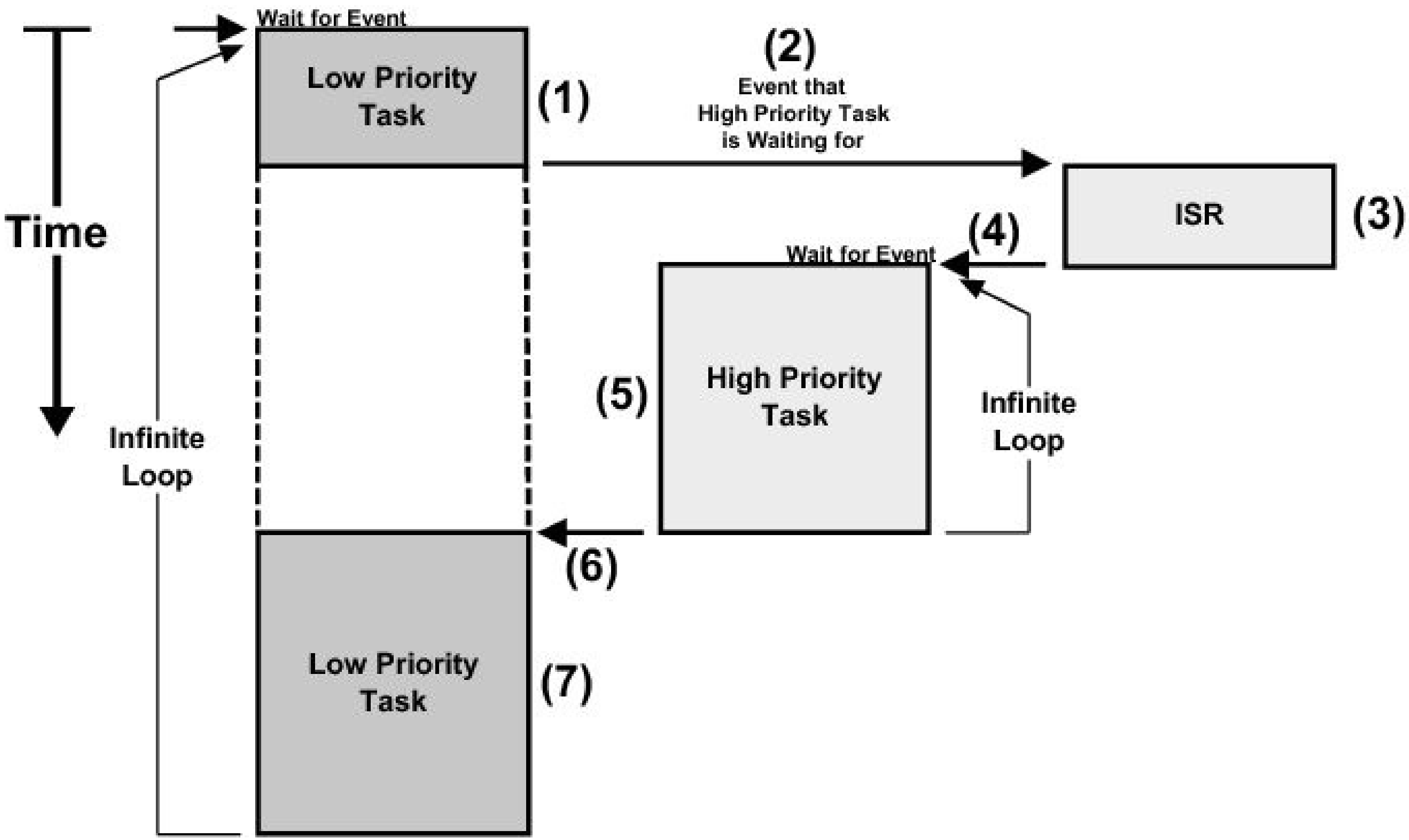
- All ‘tasks’ have the same priority!
  - Code executes in sequence
  - If an important event occurs it’s handled at the same priority as everything else!
  - You may need to execute the same code often to avoid missing an event.

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



Time
Wait for Event
Low Priority Task
(1)
(2)
Event that High Priority Task is Waiting for
ISR
(3)
(4)
Wait for Event
High Priority Task
(5)
Infinite Loop
(6)
Low Priority Task
(7)
Infinite Loop

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 is management --- scheduling and switching the CPU between several tasks --- is performed by the kernel of the real-time system

Lecture outline

- Why concurrency?
- Foreground/background vs. multi-tasking systems
- **Concurrent processes**
  - Communication: message passing vs. shared memory
- Scheduling
- Bus scheduling

Concurrent processes

- Only one process runs at a time while others are *suspended*.

- Processor switches from one process to another so quickly that it appears all processes are running simultaneously. Processes run *concurrently*.

- Programmer assigns *priority* to each process and the *scheduler* uses this to determine which process to run next.

Process

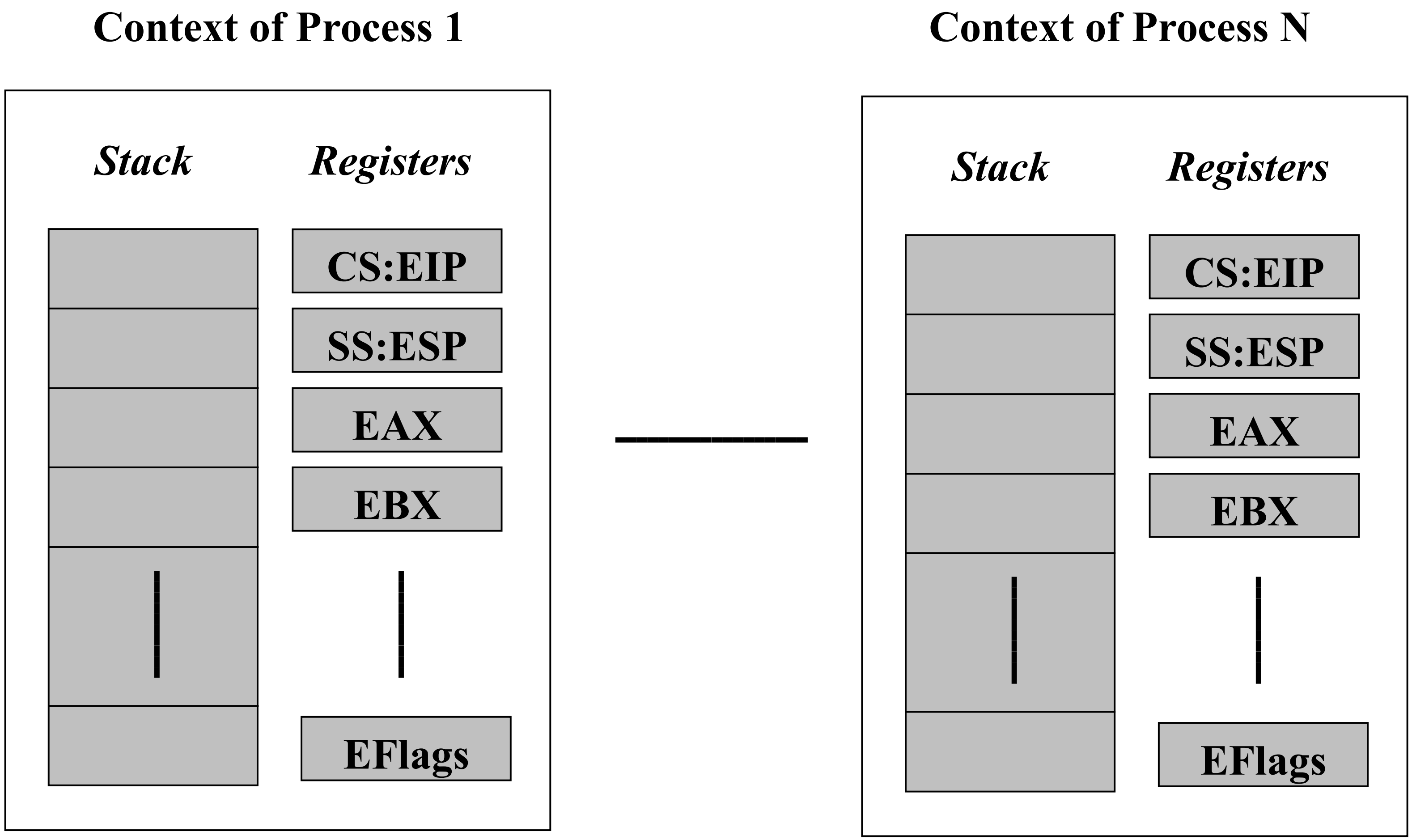
- A sequential program, typically an infinite loop
  - Executes concurrently with other processes

- Basic operations on processes
  - Create and terminate
    - Create is like a procedure call but caller doesn’t wait
      - Created process can itself create new processes
    - Terminate kills a process, destroying all data
  - Suspend and resume
    - Suspend puts a process on hold, saving state for later execution
    - Resume starts the process again where it left off
  - Join
    - A process suspends until a particular child process finishes execution

Processes vs. threads

- Different meanings in operating system terminology
- Regular processes
  - Heavyweight process
  - Own virtual address space (stack, data, code)
  - System resources (e.g., open files)
- Threads
  - Lightweight process
  - Subprocess within process
  - Only program counter, stack, and registers
  - Shares address space, system resources with other threads
    - Allows quicker communication between threads
  - Small compared to heavyweight processes
    - Can be created quickly
    - Low cost switching between threads

Each process maintains its own stack and register contents



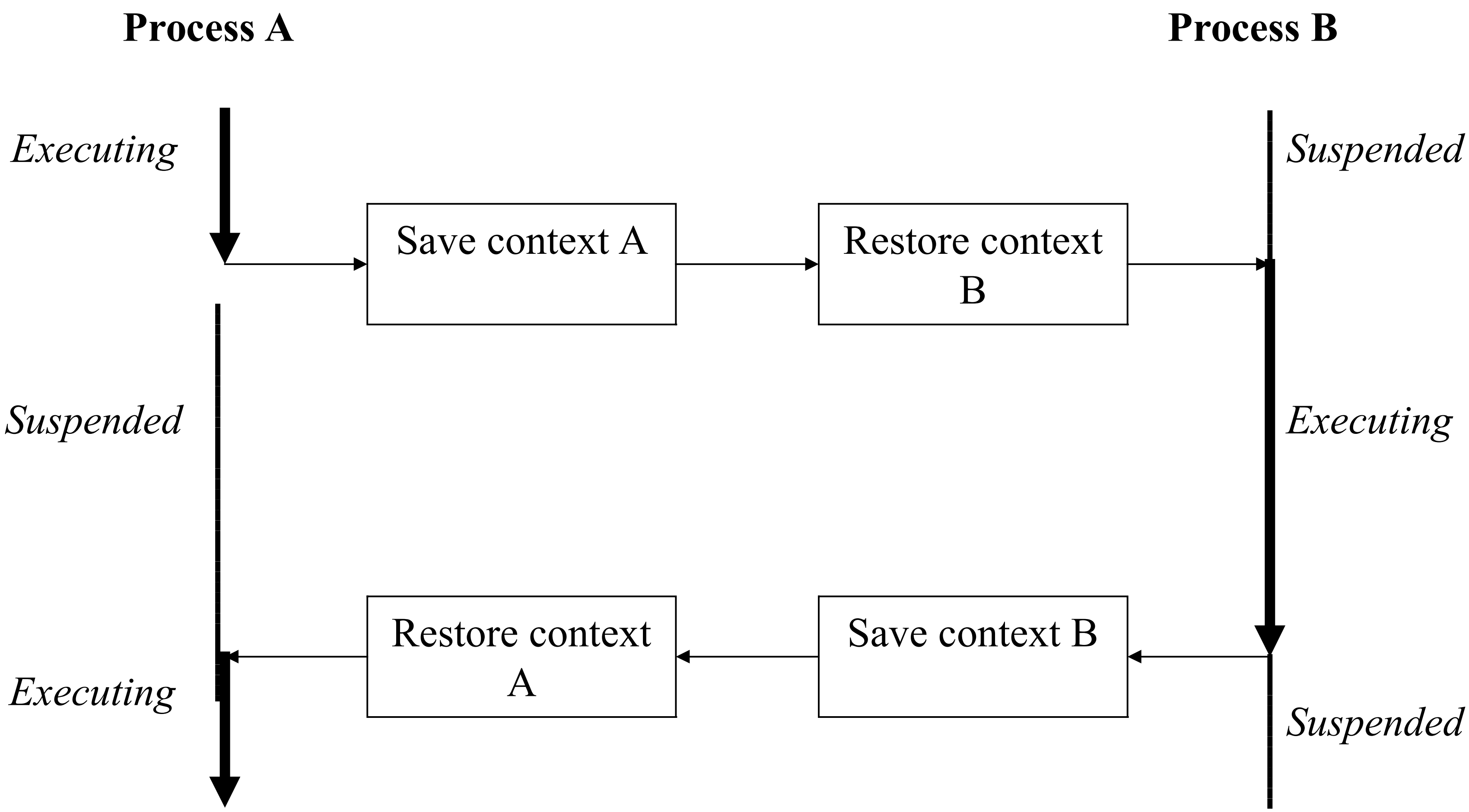
Context switching

- 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.

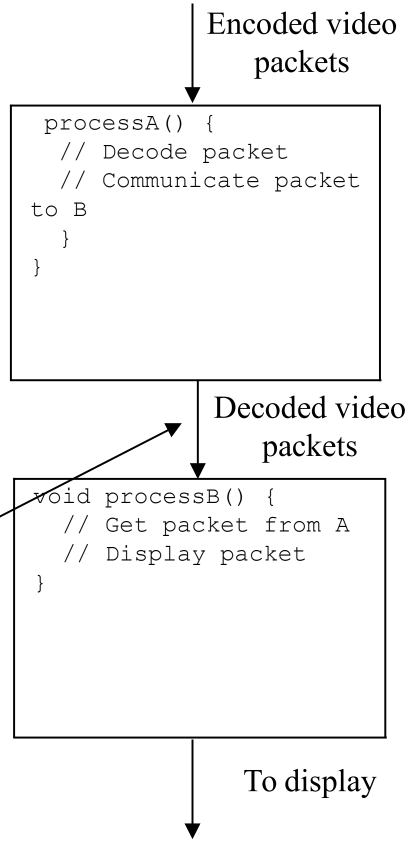
- Since CPU registers includes SS:ESP and CS:EIP, reloading context B reactivates process B's stack and returns to where it left off when it was last suspended.

Context switching, cont.



Communication among processes

- Processes need to communicate data and signals to solve their computation problem
  - Processes that don't 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



Shared Memory

- Processes read and write shared variables
  - No time overhead, easy to implement
  - But, hard to use – mistakes are common
- Example: Producer/consumer with a mistake
  - Share *buffer*[*N*], *count*
    - *count* = # of valid data items in *buffer*
  - *processA* produces data items and stores in *buffer*
    - If *buffer* is full, must wait
  - *processB* consumes data items from *buffer*
    - If *buffer* is empty, must wait
  - Error when both processes try to update *count* concurrently (lines 10 and 19) and the following execution sequence occurs. Say “count” is 3.
    - *A* loads *count* (*count* = 3) from memory into register R1 (R1 = 3)
    - *A* increments R1 (R1 = 4)
    - *B* loads *count* (*count* = 3) 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

```
01: data_type buffer[N];
02: int count = 0;
03: void processA() {
04:   int i;
05:   while( 1 ) {
06:     produce(&data);
07:     while( count == N );/*loop*/
08:     buffer[i] = data;
09:     i = (i + 1) % N;
10:     count = count + 1;
11:   }
12: }
13: void processB() {
14:   int i;
15:   while( 1 ) {
16:     while( count == 0 );/*loop*/
17:     data = buffer[i];
18:     i = (i + 1) % N;
19:     count = count - 1;
20:     consume(&data);
21:   }
22: }
23: void main() {
24:   create_process(processA);
25:   create_process(processB);
26: }
```

Message Passing

- 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

Correct Shared Memory Solution to the Consumer-Producer Problem

- 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* **<u>or</u>** *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!
- One deadlock elimination protocol requires locking of numbered mutexes in increasing order and two-phase locking (2PL)
  - Acquire locks in 1st phase only, release locks in 2nd phase

```
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();
  }
}
```

Summary: multiple processes sharing single processor

- Manually rewrite processes as a single sequential program
  - Ok for simple examples, but extremely difficult for complex examples
  - Automated techniques have evolved but not common
- Can use multitasking operating system
  - Much more common
  - Operating system schedules processes, allocates storage, and 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)
- Can convert processes to sequential program with process scheduling right in code
  - Less overhead (no operating system)
  - More complex/harder to maintain

Lecture outline

- Why concurrency?
- Foreground/background vs. multi-tasking systems
- Concurrent processes
  - Communication: message passing vs. shared memory
- **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 nonpreemptive scheduler

Preemptive vs non-preemptive

- 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 10 to 100s of milliseconds long)
  - Determines which process will be next to run

- Nonpreemptive
  - Only determines which process is next after current process finishes execution

Static vs dynamic scheduling

- Static (off-line)
  - complete a priori knowledge of the task set and its constraints is available
  - hard/safety-critical system

- Dynamic (on-line)
  - partial taskset knowledge, runtime predictions
  - firm/soft/best-effort systems, hybrid 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

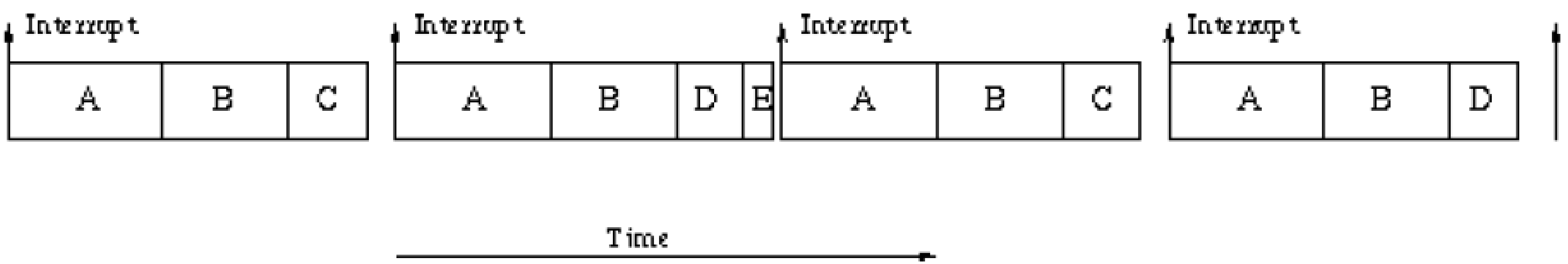
Process	Period	Comp. Time
A	25	10
B	25	8
C	50	5
D	50	4
E	100	2

```
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;
```

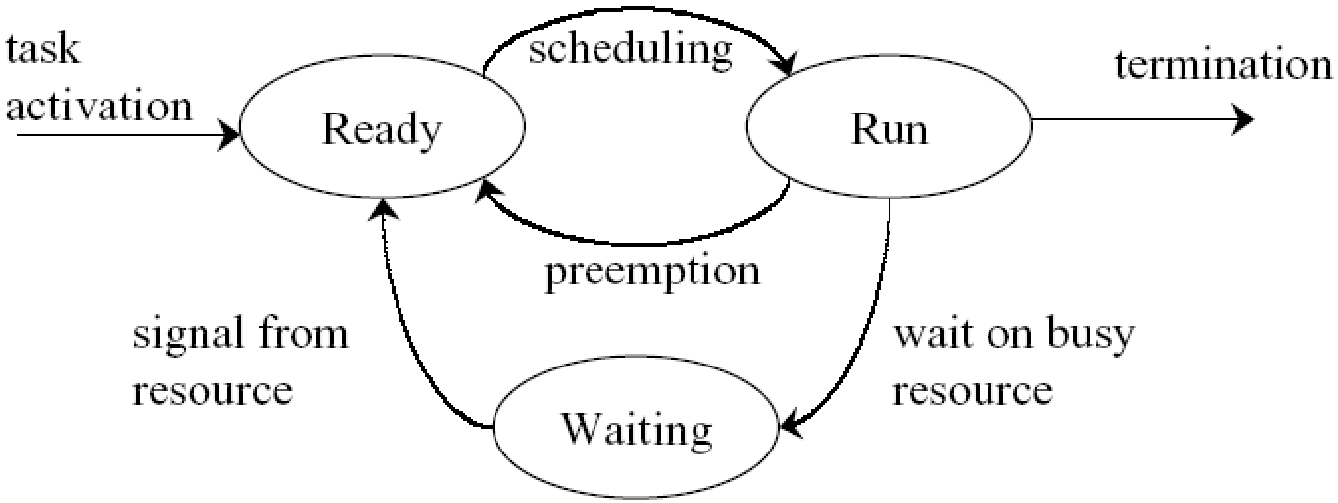


- What happens if the periods are not harmonic?

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 …

States of a process



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

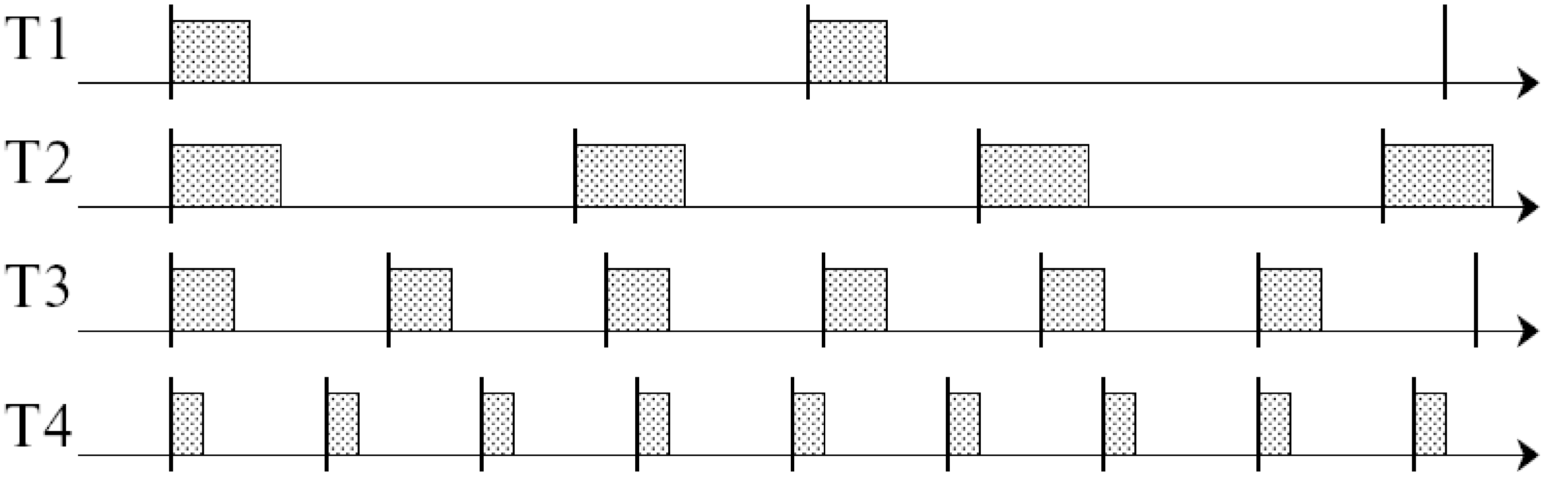
Rate Monotonic

- 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
- Preemptive
  - Unless stated otherwise

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

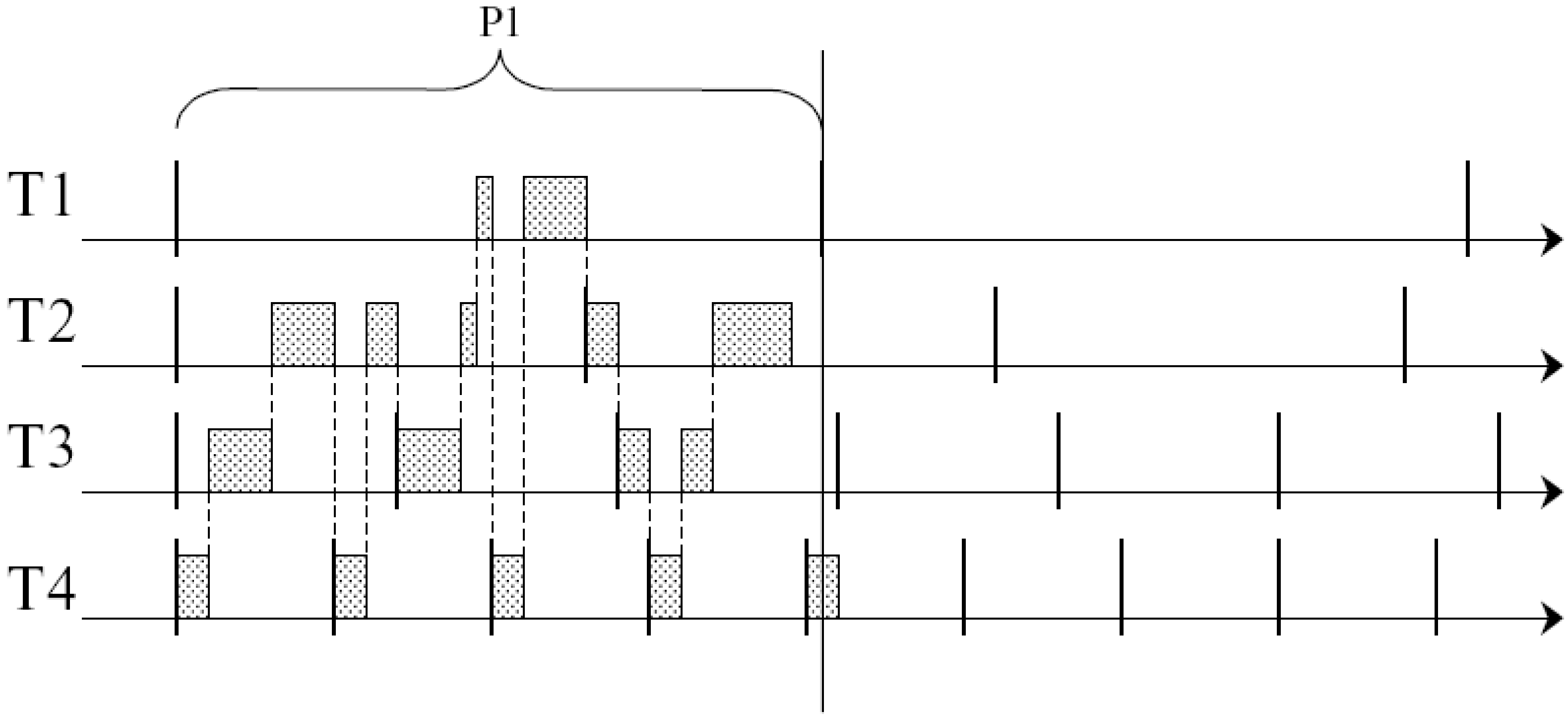
Example 1

- Assume we have the following task set
  - OBS: not scheduled yet …



Example 1 (cont'd)

- Scheduled with RM



Schedulability test for RM

- Sufficient, but not necessary:

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

- Necessary, but not sufficient:

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

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%

Example 2

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

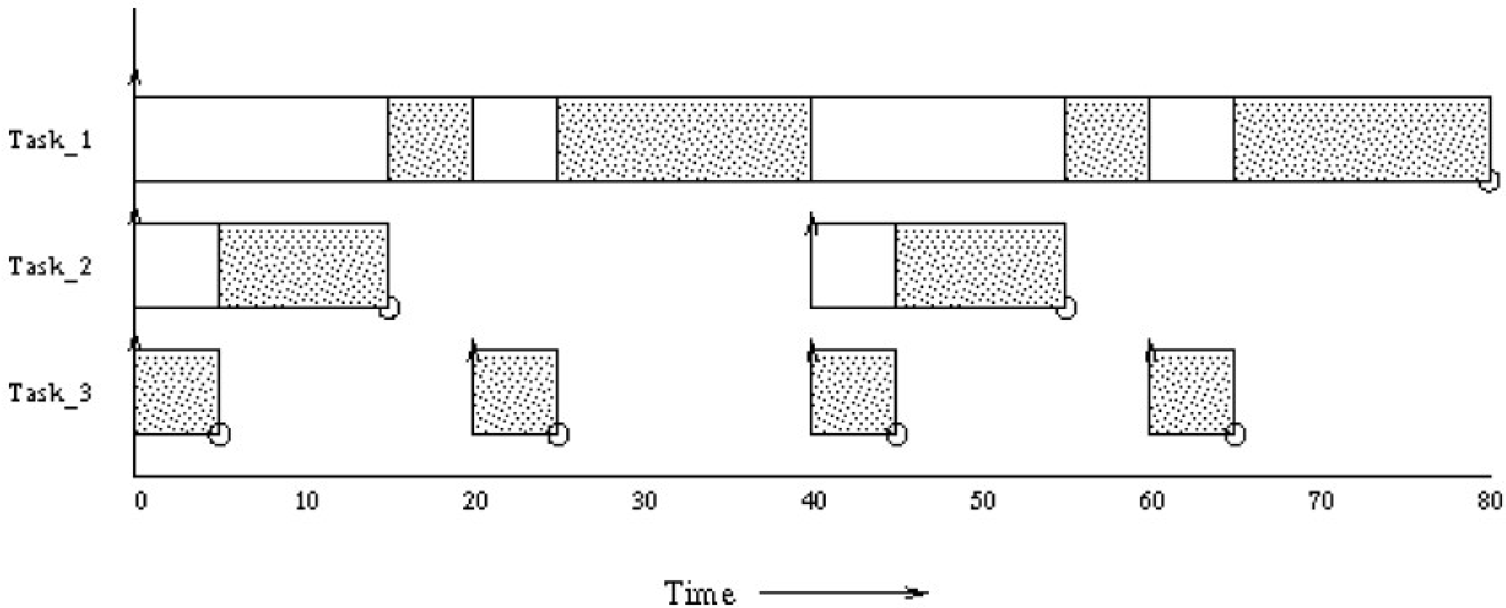
- Is this schedulable?

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:



Exact schedulability test

- The scedulability of a given taskset for RM can be decided by:
  - Drawing a schedule
  - Doing a response time analysis
- Complexity: Pseudo-polynomial time

Optimality of scheduling algorithms

- “A scheduler is optimal if it always finds a schedule when a schedulability test indicates there is one.”
  - Burns, 1991
- “An optimal scheduling algorithm is one that may fail to meet a deadline if no other scheduling algorithm can meet it.”
  - Stankovic et al., 1995
- “An optimal scheduling algorithm is guaranteed to always find a feasible schedule, given that a feasible schedule does exist.”
  - Hansson, 1998

Optimality of RM

- Rate Monotonic is optimal among fixed priority schedulers
  - If we assume the “Simple Process Model” for the tasks

What to do if not schedulable

- Change the task set utilitsation
  - 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 reevaluation?

- Assumes the “Simple task model”
  - Actually more relaxed: $D_i < T_i$

- Preemptive
  - Unless stated otherwise

Schedulability test for EDF

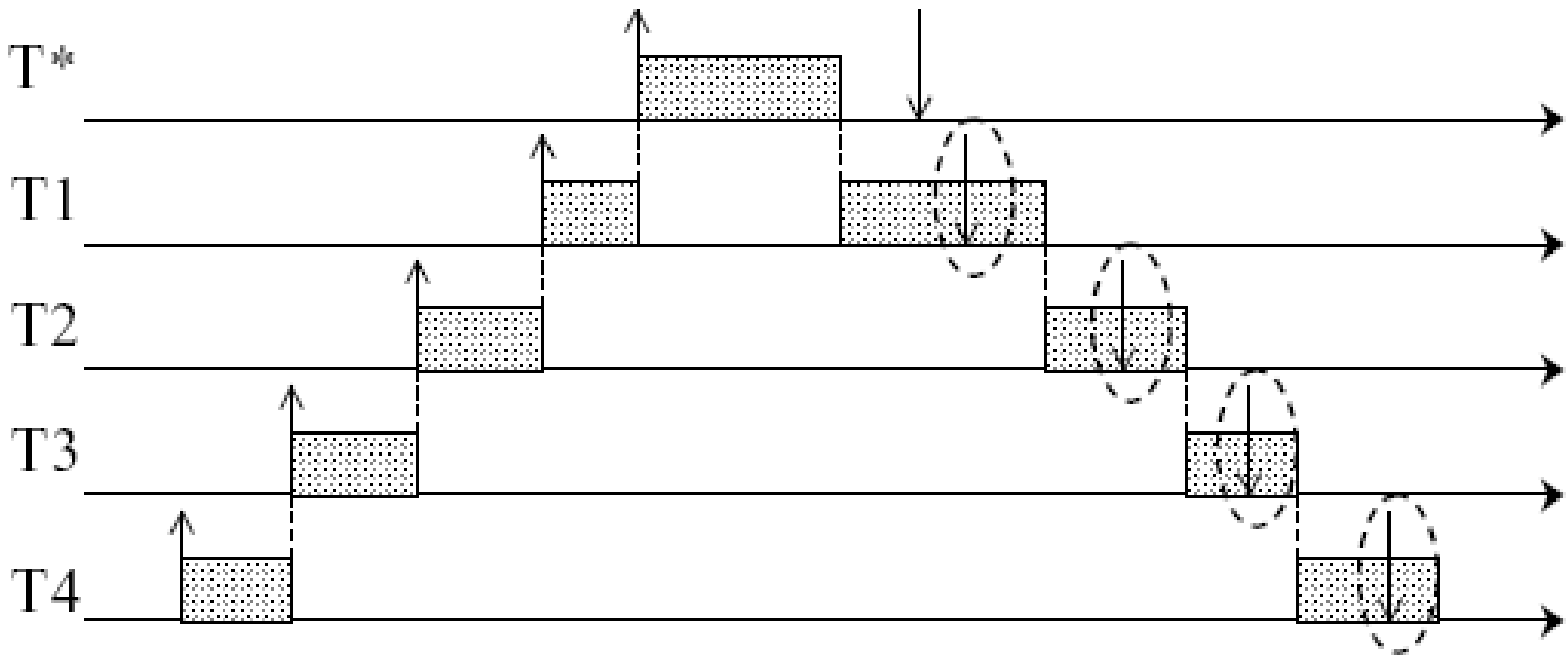
- Utilitsation test
  - Necessary and sufficient (exact)

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

Optimality of EDF

- EDF is optimal among dynamic priority schedulers
  - If we assume the “Simple Process Model” for the tasks
    - Or a more relaxed one where $D_i < T_i$

Domino Effect

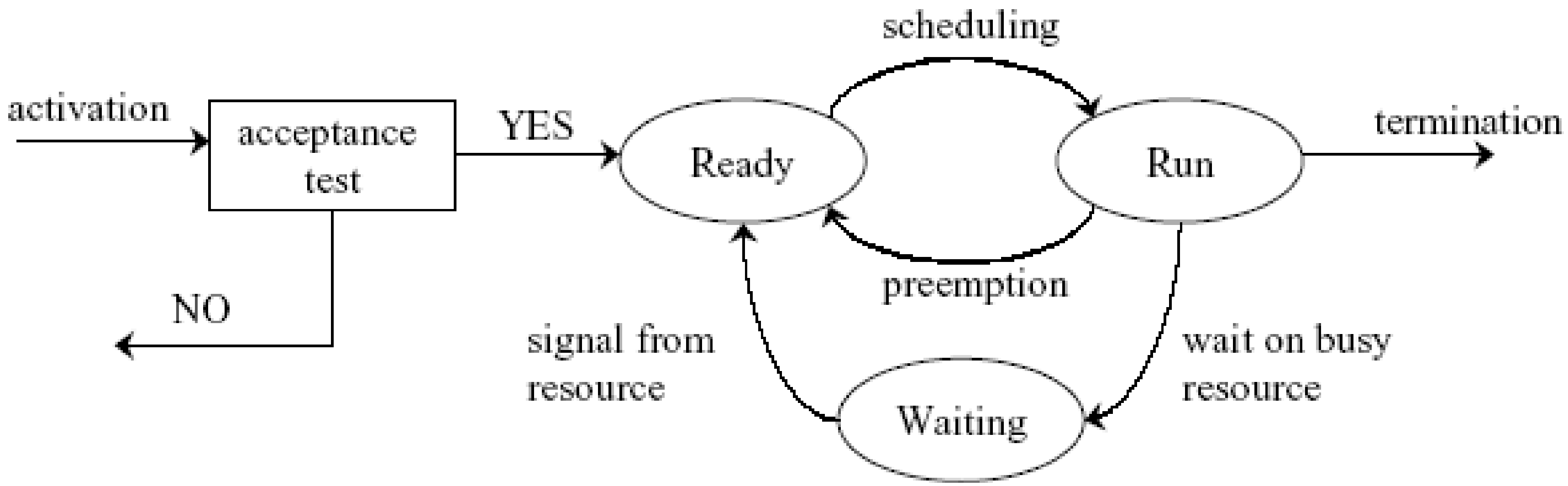


Domino effect!!!

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

Dynamic Scheduling



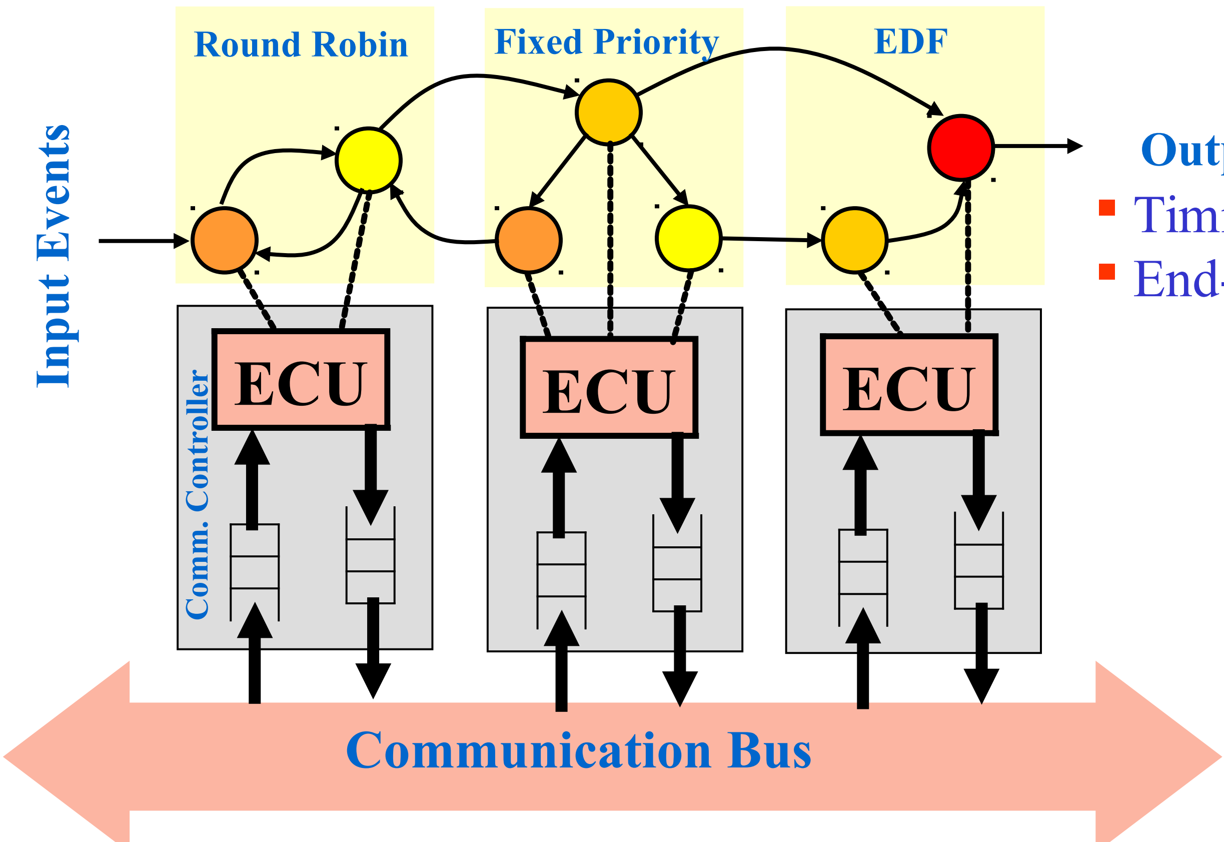
Lecture outline

- Why concurrency?
- Foreground/background vs. multi-tasking systems
- Concurrent processes
  - Communication: message passing vs. shared memory
- 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



- Timing Properties?
- End-to-end delay?

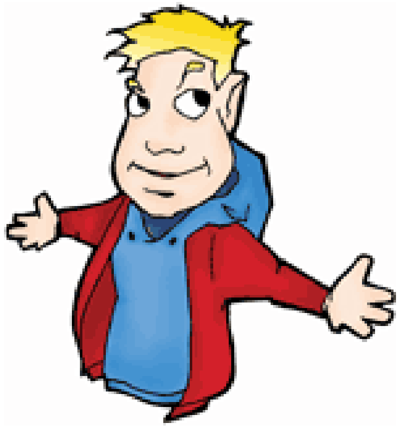
- Tasks have different activation rates and execution demands
- Each computation/communication element has a different scheduling/arbitration policy

Goal

- Time- and Event-triggered protocols – advantages and disadvantages
  - With the example of bus protocols in automotive systems

Time-Triggered and Event-Triggered Systems

TT

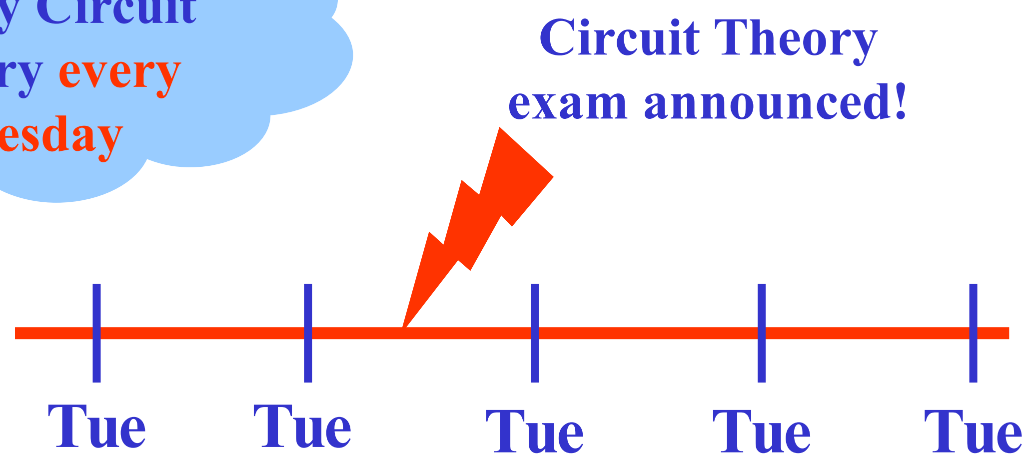


ET

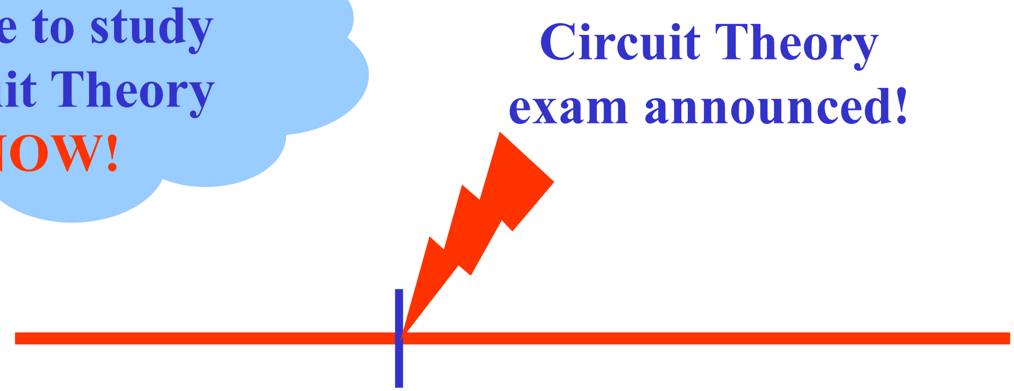
- TT and ET are two students studying at the university
- TT is well-organized and studies according to a predefined schedule
- ET takes life more casually and studies only when necessary

The Story of Two Students – TT and ET

I study Circuit Theory **every Tuesday**



I have to study Circuit Theory **NOW!**



The Story of Two Students – TT and ET

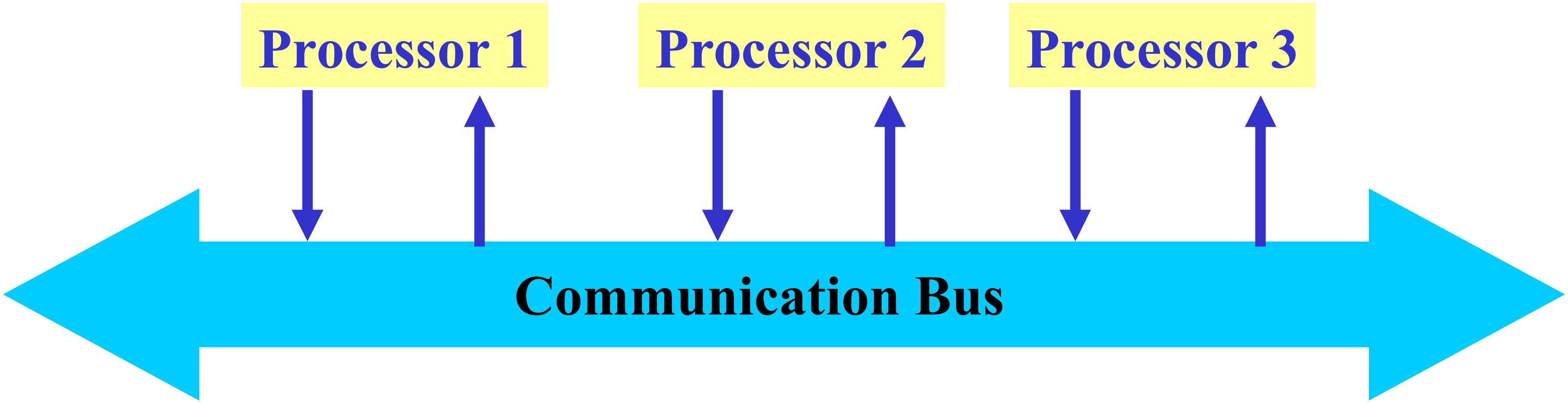
I study Circuit Theory every Tuesday

TT follows a time-driven or Time-Triggered protocol

I have to study Circuit Theory NOW!

ET follows a event-driven or Event-Triggered protocol

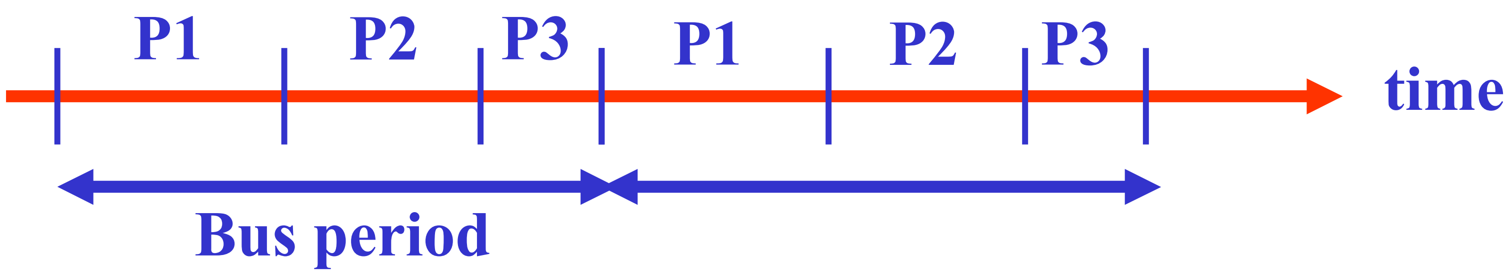
Bus Arbitration Policies



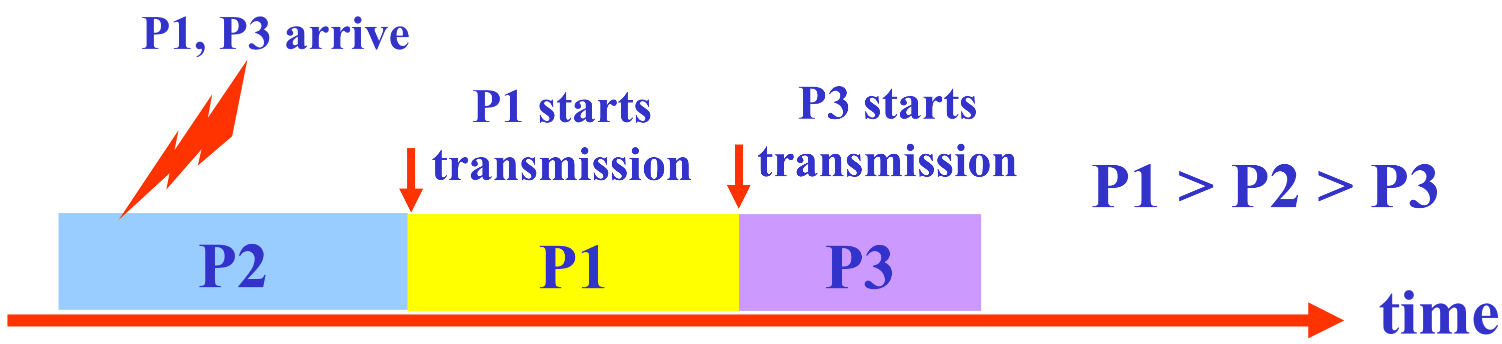
- When multiple processors want to transmit data at the same time, how is the contention resolved?
  - Using a bus arbitration policy, i.e. determine who gets priority
  - Examples of arbitration policies
    - **Time Division Multiple Access, Round Robin, Fixed Priority …**

Time-triggered arbitration

Time-triggered arbitration policy

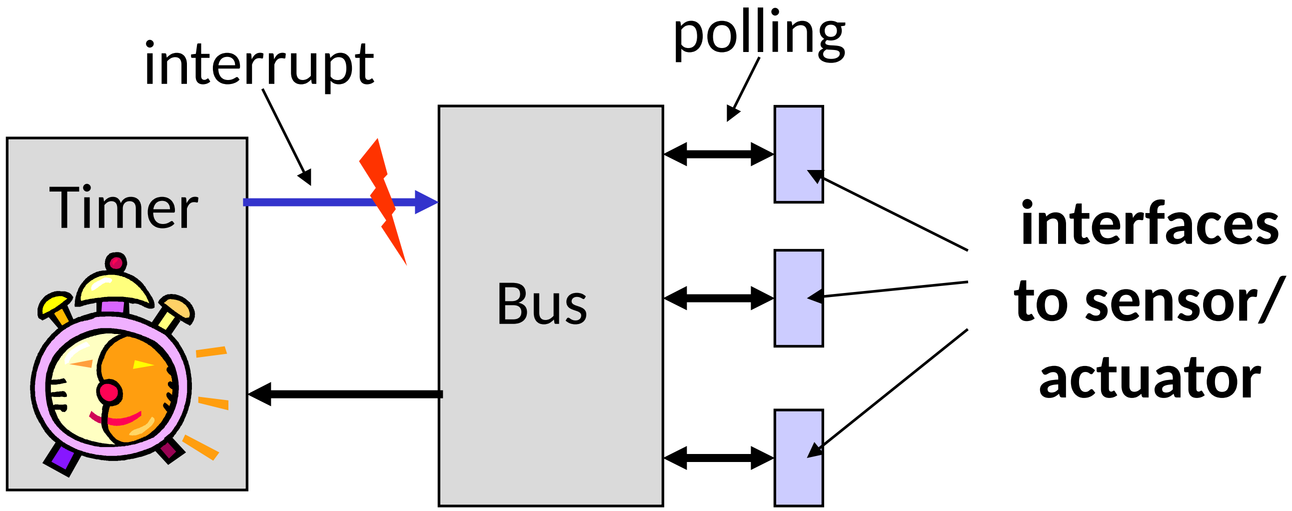


(Non preemptive) Event-triggered arbitration policy



P1 > P2 > P3

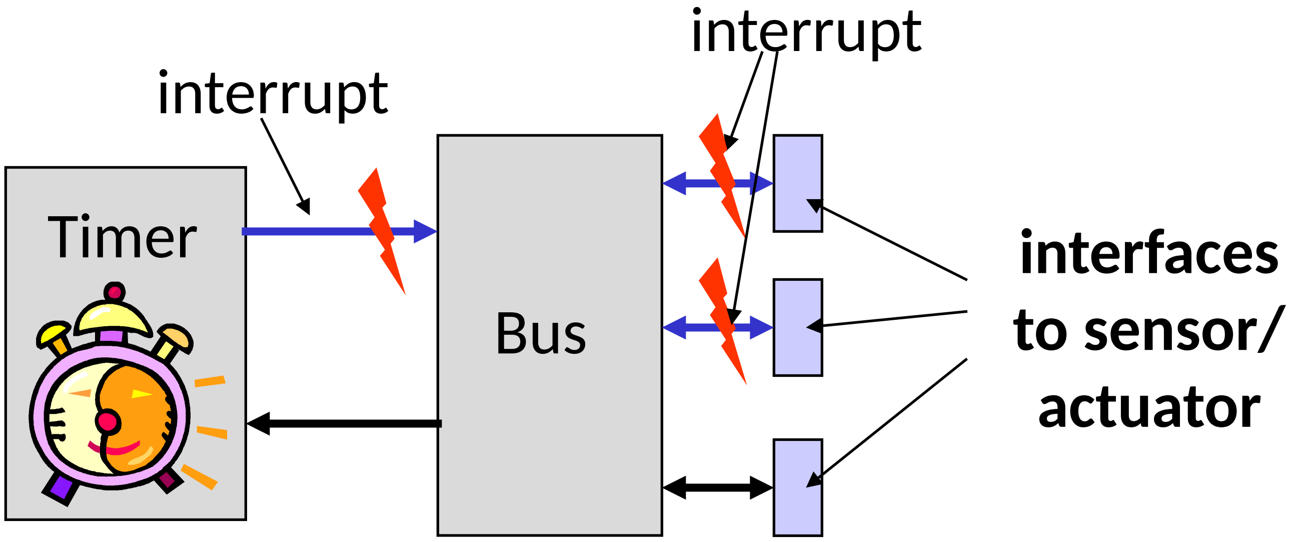
Bus Arbitration Policies



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

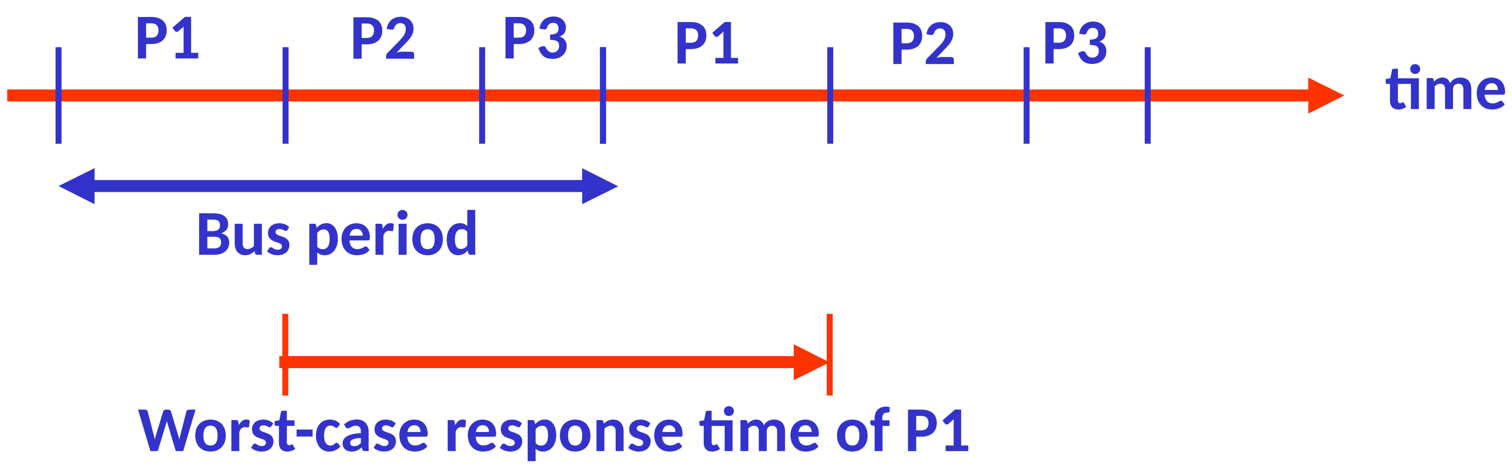
Bus Arbitration Policies



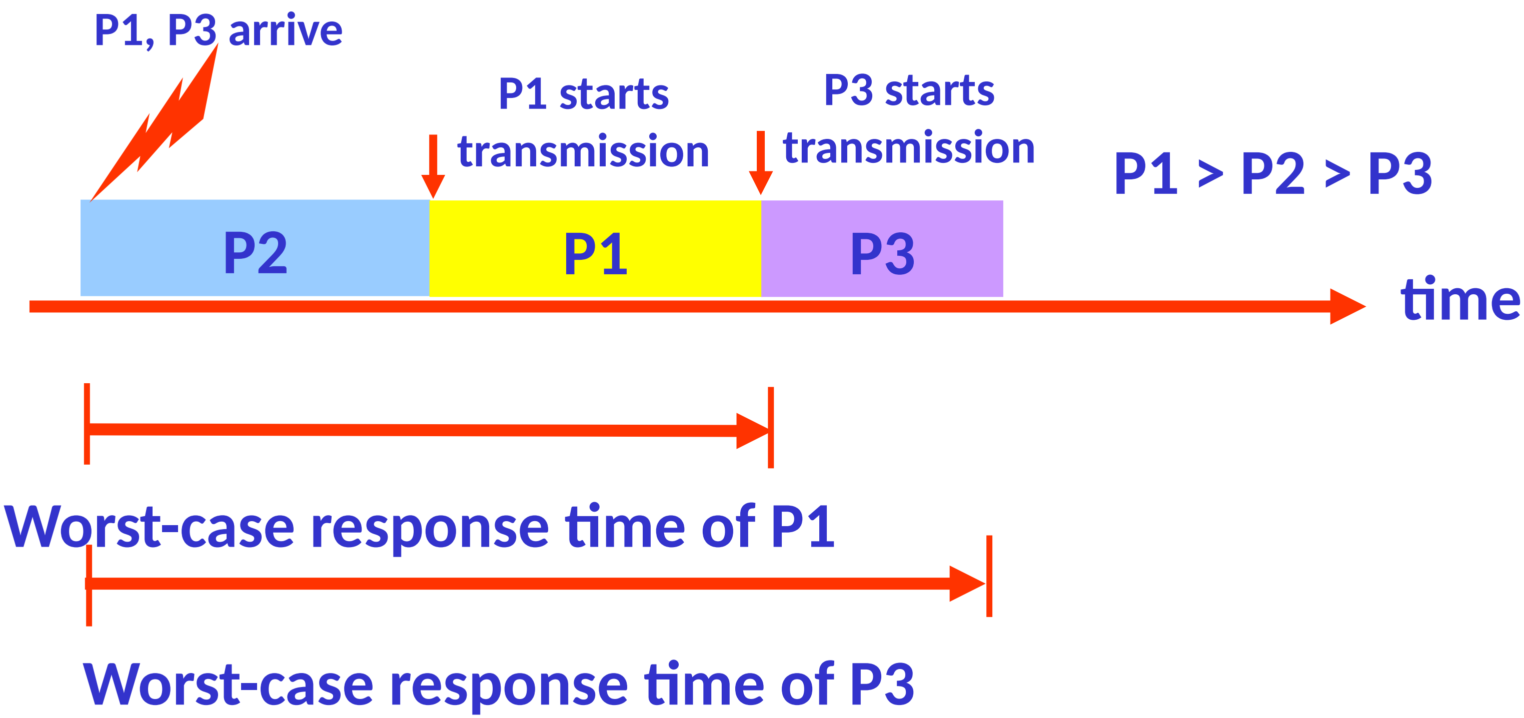
Event-Triggered Policy

- 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)** – mostly used for reliable/guaranteed communication. Also 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
  - ‘Does not provide’ hard real-time guarantees

Time-Triggered Vs Event-Triggered: Summary

- Both have their advantages and disadvantages
- Recently, combined protocols are being developed

	Time-Triggered	Event-Triggered
Hard Real-Time Guarantees	☺	×
Response Times	×	☺
Bus Utilization	×	☺
Flexibility	×	☺
Composability	☺	×

Conclusion

- TT and ET concepts also applicable to processors

- RMS and EDF are ET or TT?