## Chapter 2, Section 1:
## Real-Time Systems Concepts

1. Foregroung/background systems and multitasking systems
2. Shared resources and critical sections
3. Reminder on mutual exclusion, deadlock and communication mechanism
4. Task model with priorities (preempted systems)
5. Interrupts





## 1. Foregroung/Background Systems and Multitasking Systems

1.1 *Foreground/background* systems

1.2 Migration from a *foreground/background* system to a multitasking system

1.3 Definition of a task (thread)

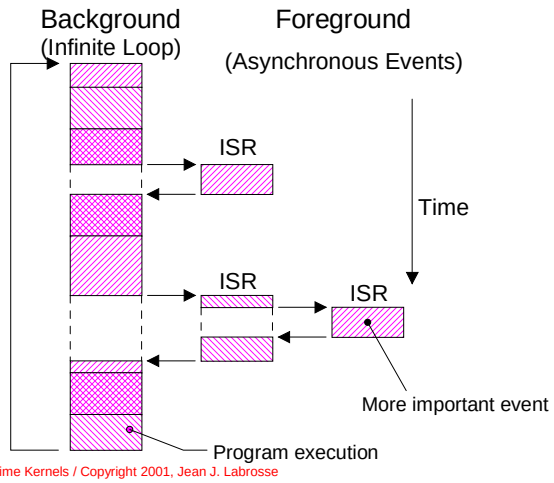
1.4 Concurrent programming

## 1.1 *Foreground/background systems*

- **Efficient for small systems of low complexity**
- **Infinite loop that call modules (Background also called *task level*)**
- **Interrupt Service Routines (ISRs) handle asynchronous events (foreground also called *ISR level*)**
  - **Timer interrupts**
  - **I/O interrupts**




## 1.1 *Foreground/background* systems




Real-Time Kernels / Copyright 2001, Jean J. Labrosse

## 1.1 *Foreground/background* system

*Example :*

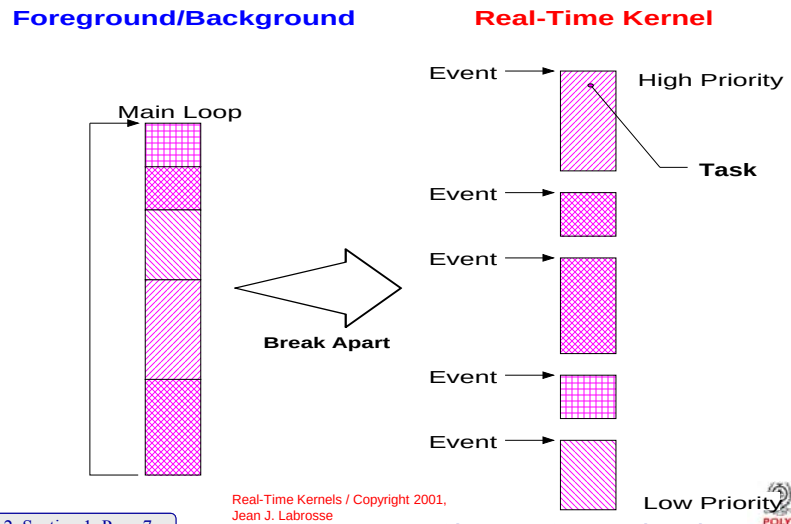
```
void main (void)          /* Background */
{
    Initialization;
    FOREVER {
        Read analog   inputs;
        Read discrete inputs;
        Perform monitoring functions;
        Perform control    functions;
        Update analog   outputs;
        Update discrete outputs;
        Scan keyboard;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




## 1.2 Migration from a *foreground/background* system to a multitasking system

- **For systems of higher complexity or to facilitate the reusing, a multitasking system is more appropriate.**

## 1.2 Migration from a *foreground/background* system to a multitasking system









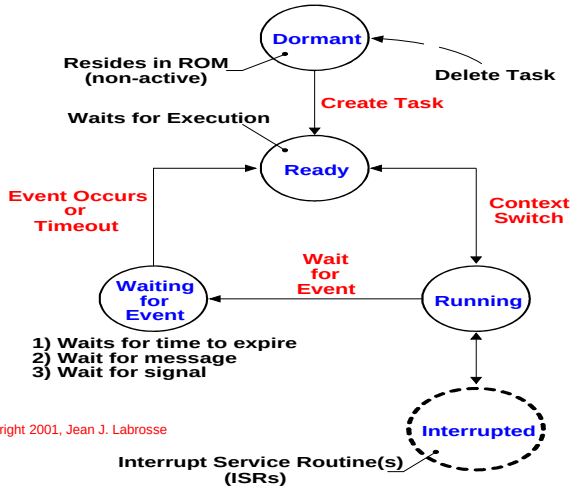



## 1.3 Definition of a task (thread)

- A task is a simple program that thinks it has the CPU all to itself
- It is typically an infinite loop that can be in any different states: DORMANT, READY, RUNNING, WAITING, ISR

```
void Task(void *pdata)
{
    Do something with 'argument' pdata;
    Task initialization;
    for (;;) {
        /* Processing (Your Code)                  */
        Wait for event;     /* Time to expire ...   */
                            /* Signal from ISR ...  */
                            /* Signal from task ... */
        /* Processing (Your Code)                  */
    }
}
```

# 1.3 Definition of a task (thread)





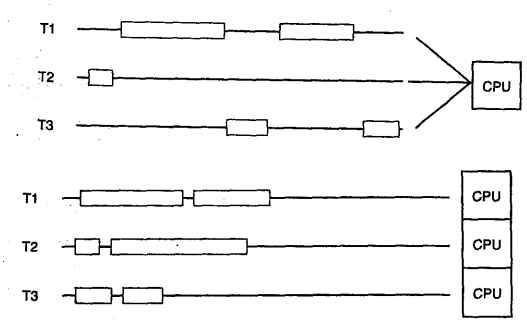







# 1.4 Concurrent programming

**Concurrent programming is employed to express the potential parallelism and to resolve the problems of synchronization and communication in multitasking systems.**




**Concurrent programming**

**Parallel programming**

**(AMP or SMP?)**

# 2. Shared resources and critical sections







ÉCOLE POLYTECHNIQUE MONTRÉAL

## 2.1 Shared Resource

- **A resource is an entity used by a task (I/O device, variable, array, data structure, etc.)**
- **A shared resource is a resource that can be used by more than one task.**
- **A resource can be corrupted (data corruption) when simultaneously consulted (e.g. read/write) by more than one task.**
  - Each task should gain exclusive access before accessing the shared resource. Otherwise, the sequence of execution can provide data corruption.

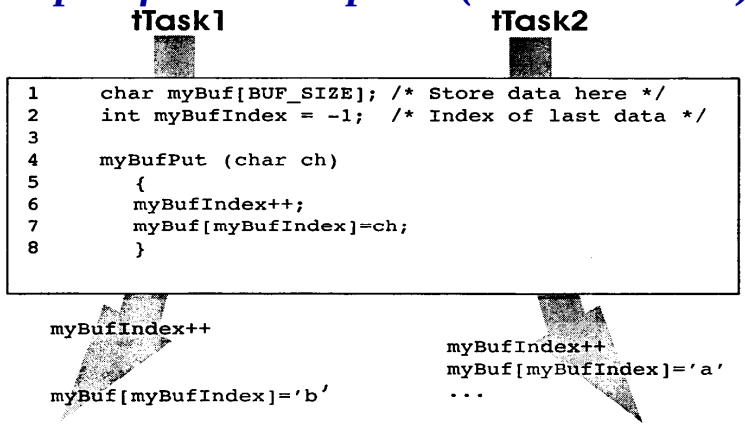




ÉCOLE POLYTECHNIQUE MONTRÉAL

## 2.1 Shared resource

***Example of data corruption (Race condition) :***










ÉCOLE POLYTECHNIQUE MONTRÉAL

## 2.2 Reentrancy

- **To avoid data corruption, functions of a real-time application must be reentrant.**
- **A *reentrant function* can be used by more than one task without fear of data corruption.**
- **A reentrant function can be interrupted at any time and resumed at a later time without loss of data.**

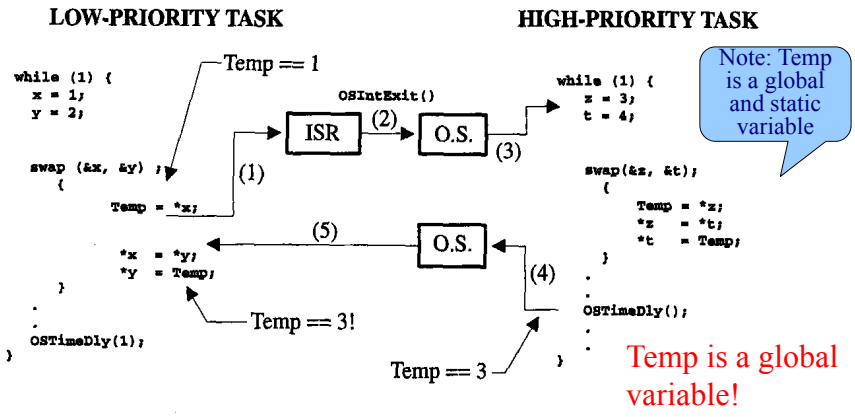




ÉCOLE POLYTECHNIQUE MONTRÉAL

## 2.2 Reentrancy

### *Example of a non reentrant function:*







## 2.2 Reentrancy

### *Mechanisms to assure reentrant function:*

- **Use only local variables**
- **Use mutex to assure the sharing of variables**
- **Disable interrupts before modifying the content of a global variable**
- **In multiprocessor systems, the following hypothesis is made to guarantee reentrancy:**

  ***In the case of a multiprocessor architecture, a variable can be modified by only one processor.***

## 2.3 Critical section

- **A *critical section* of code is code that needs to be treated indivisibly. Once the section of code starts executing, it must not be interrupted (even by the highest priority task).**
- **Example: instructions in the function** *myBufPut* (slide 13)





## 3. Reminder on mutual exclusion, deadlock and communication mechanism

3.1 Mutual exclusion mechanisms

3.2 Deadlock

3.3 Communication mechanisms

## 3.1 Mutual exclusion mechanisms

- The easiest way for tasks to communicate with each other is through shared data structures
- The most common methods for obtaining exclusive access to shared resources are:
  - disabling interrupts;
  - disabling scheduling;
  - performing test-and-set operations, and
  - the use of mutex, semaphore, spin locks, etc.





## 3.1 Mutual exclusion mechanisms

- Enabling/disabling interrupts facts:
  - Disabling interrupts affects interrupt latency and can cause interrupt to be missed
  - Appropriate to share a variable between a task and a ISR (Interrupt Service Routine)
  - Appropriate to share a variable between ISRs (Interrupt Service Routine)

## 3.1 Mutual exclusion mechanisms

- **Enabling/disabling scheduler**
  - **Not efficient when a variable is shared between a task and a ISR (Interrupt Service Routine) since that while the scheduler is locked interrupts may be enable**
  - **The behavior is very similar to that of a non-preemptive kernel**





## 3.1 Mutual exclusion mechanisms

- **TestAndSet:**
  - **Indivisibly (e.g. by the processor) or you must disable interrupts :**
    - **Test if a global variable is either 0 or 1**
    - **Set the variable to 1**

```
int TestAndSet(* lock) {
   int initial ;
   initial = lock;
   lock = 1;
   return initial;
}
```

## 3.1 Mutual exclusion mechanisms

- **Mutex and binary semaphore: 0 or 1**
- **Semaphore Counter: 0 to MAX (255, 65535, …)**
- **Spin locks: Busy waiting through an infinite loop in order to enter in a critical section.**

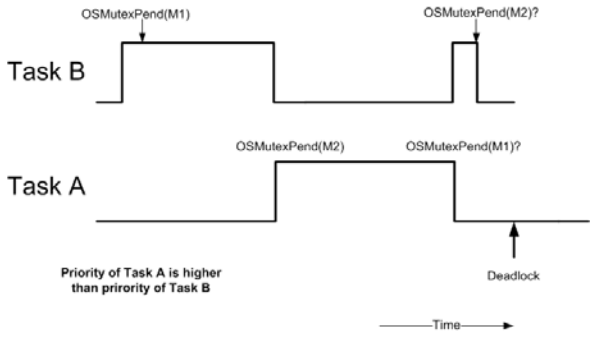
```
volatile int lock = 0;
void Critical() {
      while (TestAndSet(&lock) == 1);
      critical section //only one process can be in this section at a time
      lock = 0 //release lock when finished with the critical section
}
```





## 3.2 Deadlock

- **A deadlock is a situation in which two tasks are each unknowingly waiting for resources held by the other.**



## 3.2 Deadlock

- The simplest way to avoid a deadlock is for tasks to:
  - acquire all resources before proceeding,
  - acquire the resources in the same order, and
  - release the resources in the same order
- Most kernels allow to specify a timeout when acquiring a mutex. This features allows a deadlock to be broken.





## 3.3 Communication mechanisms

- Mailbox:
  - protected buffer that may contain a message
  - e.g. used for the sending of data periodically between two tasks
- Message queues (FIFO):
  - a message queue is used to send one or more messages to a task (e.g. array of mailbox)
  - e.g. used to support a data flow (stream) between two tasks.

## 3.3 Communication mechanisms

- **Synchronization**
  - **A task can be synchonized with an ISR (or another task) by using a semaphore. E.g. to indicate that a new task must be processed (*unilateral rendez-vous*)**
  - **Two tasks can synchronize their activities by using two semaphores (*bilateral rendez-vous*)**





## Unilateral rendez-vous

```
void  DrivFifo (void *data)
{
      msg dat;
      UBYTE err;
      data = data;
      dat.device = 1;
      while(1){
            OSSemPend(Irq1, 0, &err);
            dat.value = reg1;
            OSQPost(Fifo, (void*)&dat);
```

```
void  DevFifo (void *data)
{
      int i;
      unsigned short value = 0;
      data = data;
      while(1){
            for(i=0; i<100; i++) {
                  // Delay for 10 millisecond
                  OSTimeDlyHMSM(0, 0, 0, 10);
                  OSSemAccept(Irq1);
                  reg1 = value++;
                  OSSemPost(Irq1);
            }
            OSTimeDlyHMSM(0, 0, 1,400);
```

# Unilateral rendez-vous

```
void  DrivFifo (void *data)
{
      msg dat;
      UBYTE err;
      data = data;
      dat.device = 1;
      while(1){
            OSENTERCRITICAL
            OSSemPend(Irq1, 0, &err);
            dat.value = reg1;
            OSQPost(Fifo, (void*)&dat);
            OSEXITCRITICAL
        }
```

```
ISR  DevFifo (void *data)
{
      int i;
      unsigned short value = 0;
      data = data;
      while(1){
            for(i=0; i<100; i++) {
                  // Delay for 10 millisecond
                  OSTimeDlyHMSM(0, 0, 0, 10);
                  OSSemAccept(Irq1);
                  reg1 = value++;
                  OSSemPost(Irq1);
            }
            OSTimeDlyHMSM(0, 0, 1,400);
```




# Bilateral rendez-vous

```
Irq1 = OSSemCreate(0);
Irq2 = OSSemCreate(0);
```

```
T1
OSSemPost(Irq1)
OSSemPend(Irq2)       // wait T2
.
.
.
code 2
.
.
OSSemPost(Irq1)
```

```
T2

OSSemPend(Irq1);      // wait T1
.
.
code1
.
.
OSSemPost(Irq2);
OSSemPend(Irq1);
```

# 4. Task model with priorities (preempted systems)

## 4.1 Priorities and tasks

## 4.2 Non-Preemptive kernel

## 4.3 Preemptive kernel







## 4.1 Priorities and tasks

- **The priority of a task is a metric that characterize its importance.**
- **Higher (lower) priorities are assigned to critical tasks (e.g. in terms of timing constraints)**
- **The priority of a task may be**
  - *static* (fixed for all the execution).
  - *dynamic* (may change during the execution).
  - Note: Protocol-like priority inheritance that may change temporally the priority of a task. In this case, the task is not considered having a dynamic priority.

## 4.1 Priorities and tasks

*Assigning task priorities*

- **This is a difficult problem because of the complex nature of real-time systems. Not all the tasks are considered critical. They can be given low priorities. The most difficult tasks to assign is the one having hard real-time constraints.**
- **Interesting techniques exist to assign task priorities. An example is IMS (rate monotonic scheduling).**

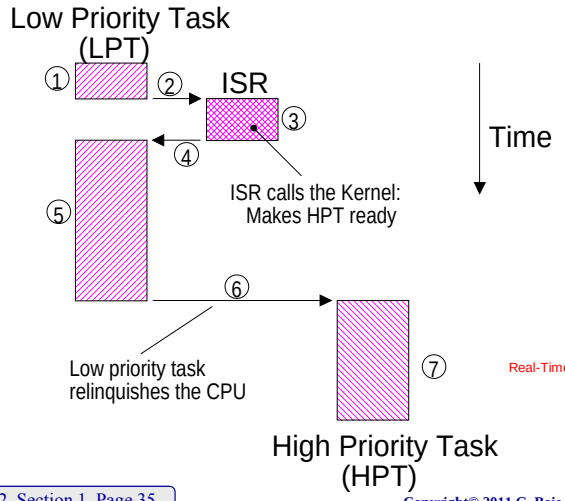




## 4.2 Non-preemptive kernel

- **Non-preemptive kernel (*cooperative multitasking*):**
  - **interrupt latency is relatively low**
  - **can use reentrant functions except for the synchronization with an ISR (unilateral rendez-vous)**
  - **less use of mutex**
  - **For HPT, delay (see 5 in the next slide) to obtain a new result from an ISR is higher and less determinism than preemptive kernel**

# 4.2 Non-preemptive kernel












# 4.2 Non-preemptive kernel

**1) An *LPT* is executing but it is interrupted by an event (e.g. peripheral interrupt)**

**2) &3)**
**ISR (Interrupt Service Routine) handles the event and makes a higher priority task (*HPT*) ready to run**

**4) Upon completion of the ISR, the CPU return to *LPT*.**

**5) When *LPT* is completed, it calls a service (software interrupt) that the kernel provides to relinquish the CPU to another task.**

**6) The kernel (scheduler) determines HPT as the next task to be executed.**

**7) The kernel performs a context switch to resume *HPT*.**

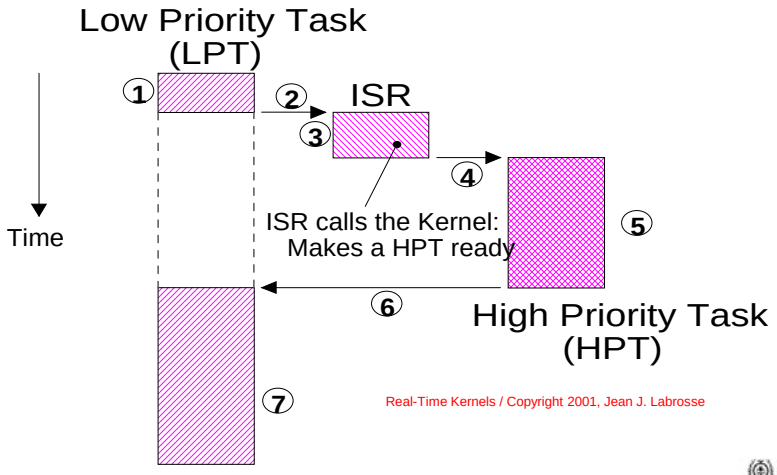
# 4.3 Preemptive kernel

- **Preemptive kernel:**
  - **most popular kernel for real-time**
  - **interrupt response high and determinism**
  - **the use of reentrant functions is a must**
  - **the use of one of four methods to shared resources is a must (e.g. the use of mutex).**





# 4.3 Preemptive kernel




Real-Time Kernels / Copyright 2001, Jean J. Labrosse

## 4.3 Preemptive kernel

1) A *LPT* is executing but it is interrupted by an event (e.g. peripheral interrupt)

2) & 3)
ISR (Interrupt Service Routine) handles the event and makes a higher priority task (*HPT*) ready to run

4) Upon completion of the ISR, it calls a service (software interrupt) that the kernel provides to relinquish the CPU to another task.

5) The kernel (scheduler) determines HPT as the next task to be executed and the kernel performs a context switch to resume *HPT*.

6) When *HPT* is completed, it calls a service that the kernel provides to relinquish the CPU to another task.

7) The kernel (scheduler) determines LPT as the next task to be executed.





# 5. Reminder about interrupts

## 5.1 What is an interrupt?

- **An interrupt is part of a communication mechanism between the CPU and the external world.**
- **An interrupt is a hardware mechanism used to inform the CPU that an asynchronous event has occurred and that it must be processed.**
- **In particular, external interrupts allow a microprocessor to process events when they occur, rather *continuously polling* an event to see if it has occurred.**





## 5.1 What is an interrupt?

**Interrupts may classified in 3 groups:**

- External interrupts (peripheral IRQ)
  - Asynchronous interrupts
- Software interrupts
  - Software interrupts are synchronous
- Exceptions (also called hardware interrupts)
  - Exceptions are either synchronous or asynchronous.

## 5.1 What is an interrupt?

### *Overview of an external interrupt*

- **Consider asynchronous (sporadic) data acquisition from a peripheral to a processor (ex. A/D converter).**
- **Two approaches are possible:**
  1. **Periodic polling to see if a new result is converted. It may increase the load of the CPU for nothing (no new data is available).**
  2. **An interrupt is raised when a new data is available, then the processor may process...**

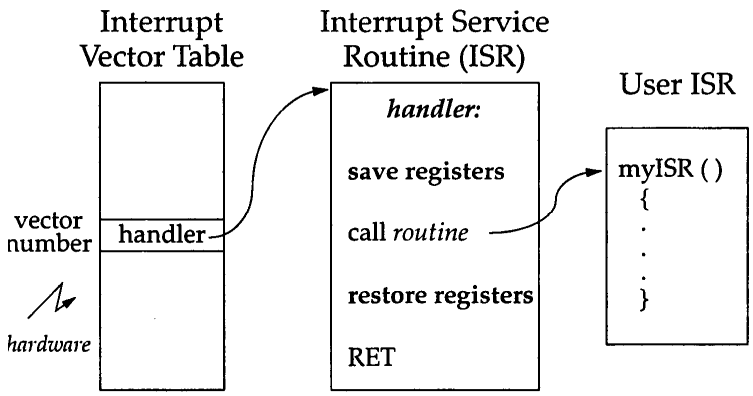




## 5.1 What is an interrupt?

- **When an interrupt is recognized, the CPU save part (or all) of its context (i.e. registers) and jump to an ISR for Interrupt Service Routine**
- **The ISR processes the event (I/O completion, end of the execution of a coprocessor, etc.)**
- **Upon completion of the ISR, it calls a service that the kernel provides to relinquish the CPU to another task (the choice of the task will depend the type of kernel, i.e. preemptive or not).**

# 5.1 What is an interrupt?












# 5.1 What is an interrupt?

- **Microprocessors provide an IDT (Interrupt Description Table), where each entry corresponds to a specific handler interrupt.**
- **Microprocessors allow interrupt to be nested.**
- **Microprocessors allow interrupts to be *ignored* and *recognized* (thus temporarily disabled) through the use of special (assembler) instructions.**

## 5.1 What is an interrupt?

- **As mentioned in section 4, interrupt can be temporarily disabled to assure the mutual exclusion of a critical section.**
- **Probably the most important specification of a real-time kernel is the maximum amount of time interrupts are disabled in a system.**
- **If an interrupt occurs during the disabling of interrupts, its response will be delayed. This delay is named *interrupt latency*.**





## 5. 2 Interrupt Latency, Response & Recovery

- **Interrupt latency**: maximum amount of time interrupts are disabled + Time to start executing the first instruction in the ISR
- **Interrupt response**: interrupt latency + the difference of time between the start of the interrupt and the start user code that handles the interrupt (ISR code).

## 5. 2 Interrupt Latency, Response & Recovery

- **Case of the non-preempted kernel:**
  - **Interrupt response =**
    - Interrupt latency +
    - Time to save the CPU's context
- **Case of a preempted kernel:**
  - **Interrupt response =**
    - Interrupt latency +
    - Time to save the CPU's context +
    - Time to update the number of nesting interrupts





## 5. 2 Interrupt Latency, Response & Recovery

- **Interrupt recovery is the time required for the processor to return to the interrupted code or to a higher priority task (case of a preempted kernel).**
- **Case of a non-preempted kernel:**
  - **Interrupt recovery =**
    - Time to restore the CPU's context +
    - Time to execute the return from interrupt instruction

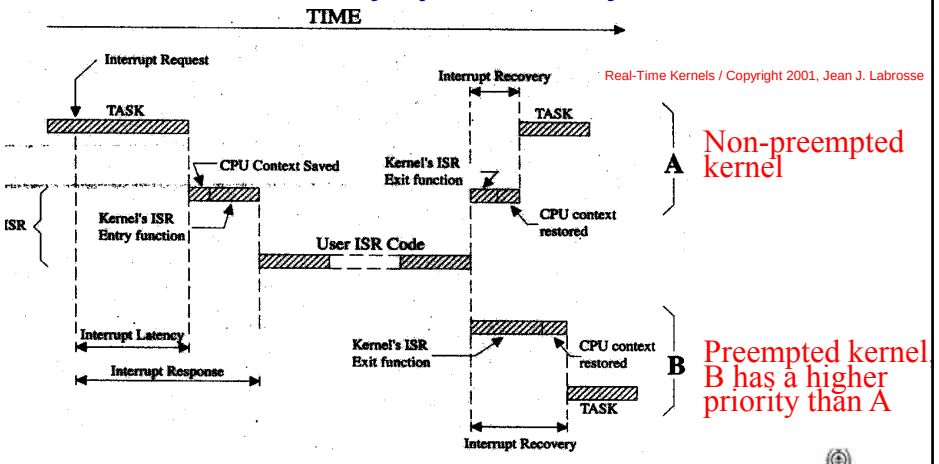
## 5. 2 Interrupt Latency, Response & Recovery

- **Case of a preempted kernel:**
  - **Interrupt recovery =**
    - Time to find the next task ready to run
    - Time to restore the CPU's context
    - Time to execute the return from interrupt instruction




## 5. 2 Interrupt Latency, Response & Recovery

***Summary of all the delays***







Copyright© 2011 G. Bois, M. De Nanclas, L. Filion

## 5. 2 Interrupt Latency, Response & Recovery

### *Non-maskable Interrupts*

- **Sometimes, an interrupt must be serviced as quickly as possible and cannot afford to have the latency imposed by a kernel, you might be able to use *nonmaskable interrupt* (*NMI*) provided by most of the microprocessors.**
- **Because the NMI cannot be disabled, interrupt latency, response, and recovery are minimal.**
- **NMI is used for drastic measures (for instance power down).**





## 5. 2 Interrupt Latency, Response & Recovery

### *Non-maskable Interrupts*

- **Disabling interrupts in the critical section of an ISR or a task do not avoid NMI**
- **Sharing a variable with an NMI needs a *TestandSet* assembler instruction (indivisibility).**

## 5.3 ISR Processing

### *Algorithm of an ISR*

1. **Save the CPU's context**
2. **Increment by 1 the ISR nesting counter**
3. **The core of the ISR is executed, for example:**
   - •Read and write from an external peripheral (UART, A/D converter, etc.)
   - •Then communicate these new values to a task through:
     - •The shared memory
     - •A messsage queue
     - •A global variable protected by a mutex through a unilateral rendez-vous





## 5.3 ISR Processing

### *Algorithm of an ISR*

4. **Decrement the ISR nesting counter and determine the highest priority task ready to run as the next task to be executed**
5. **Restore the CPU context of the task designated to run**

- **Remember, in steps 4 and 5, processor returns to the interrupted code (non-preempted kernel) or to the higher priority task (preempted kernel).**

## 5.3 ISR Processing

### *Interrupt stack*

- **Many microprocessors have their own stack to process ISR.**
  - stack is initialized at the boot
  - size of the stack must consider the worst case of nesting





## 5.3 ISR Processing

### *Interrupt stack*

- **x86, R6000, CPU-32, MC68060, MC68000 architectures do not have interrupt stacks, then they use the stack of the current task (task that was executed before the interrupt).**
  - Disadvantage: stack size of each task must consider an overhead for the worst case of nesting

## 5.3 ISR Processing

### *ISR characteristics*

- **Typically, ISR is a mixture of assembler and C code:**
  - **Assembler:**
    - **Saving context**
    - **Enabling/disabling interrupt**
    - **Switching context**
  - **C code**
    - **ISR core**
    - **Make a priority task ready to run and determine the top priority task (scheduling)**





## 5.3 ISR Processing

### *Caution on ISR*

- **In a real-time system, the goal is to minimize latency and response of interrupts in order to assure a minimum response time by the system.**
- **An ISR must not:**
  - be blocked (pending) on a mutex, semaphore, mailbox and queue.
  - Call dynamic allocation subroutines (ex.: malloc() )
  - Call I/O subroutine (ex.: printf())

## 5.3 ISR Processing

### *Caution on ISR*

- **An ISR must be completed as soon as possible since:**
  - ISRs of lower priorities can be waiting
  - tasks ready to run can be waiting
  - is difficult to debug...
- **Avoid floating point instructions inside ISR**
  - they are in general slow
  - FPU register must saved/restored

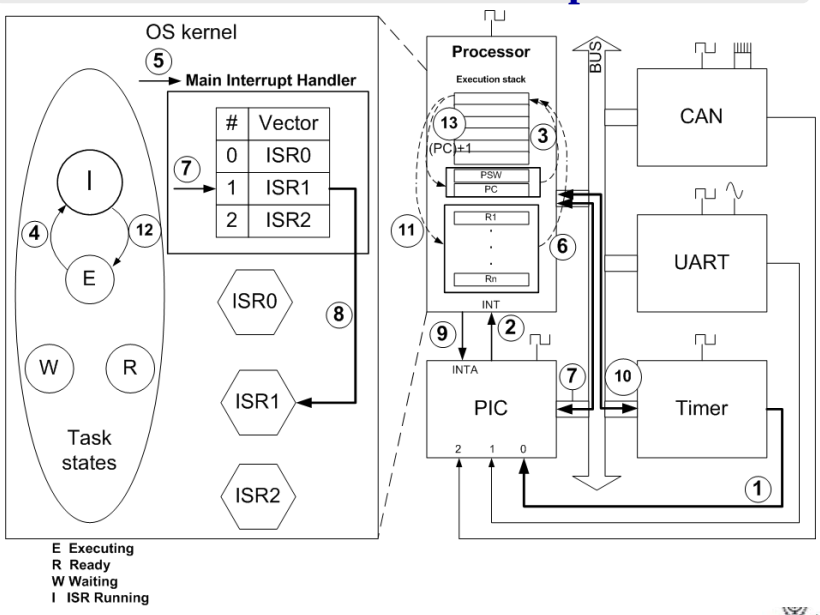




## 5.4 External interrupts

- **Interrupts from peripherals and external components in their way to a processor are generally first sent to a Processor Interrupt Controller (PIC)**
- **As mentioned previously, interrupts are more efficient (CPU usage) than polling, but require additional connections and pins on the microprocessor**
- **Next slides present a complete scenario of a typical interrupt using vector interrupt. To simplify we consider a non-preempted kernel.**

# 5.4 External interrupts








ÉCOLE POLYTECHNIQUE MONTRÉAL

# 5.4 External interrupts

1. **The timer enable its interrupt line.**
2. **The PIC recognizes the interrupt and then raises an interrupt to the processor**
3. **The processor complete its current instruction and saves PC and PSW registers**
4. **The current state of the task is now *interrupt* (i.e. and ISR is running)**
5. **Main interrupt handler is called**
6. **Complete the CPU saving of the current task (general purpose registers)**
7. **Interrupts are disabled and the main interrupt handler (main ISR) "asks" the controller what device(s) caused the interrupt**
8. **Through the interrupt vector the interrupt handler performs a branch to the right entry (dispatch to ISR1).**





ÉCOLE POLYTECHNIQUE MONTRÉAL

## 5.4 External interrupts

9. Interrupts are enabled and the processor acknowledges the PIC through the INTA pin
10. ISR1 load the new data from the timer
11. Context is restored (general registers)
12. The context of the interrupted task is resumed and its current status is now E (executed).
13. Through the execution of the RET instructions, the main interrupt handler restores PSW and PC+1





## 5.4 Case of the MicroBlaze (MB)

*Ref. http://www1.cs.columbia.edu/~soviani/cs4840hack/Microblaze_interrupts.html*

- MicroBlaze (Xilinx processor) is a good example of what happens for a typical interrupt.
- When an interrupt is detected, (if interrupts are enabled) MB stops executing the current code and jumps to address 0x00000010 (interrupt handler).
- Briefly, the main interrupt handler has to :
  1. first save the context (mainly g/p registers)
  2. acknowledge the interrupt
  3. service the interrupt
  4. restore the context and execute rtdi (return from interrupt)

# 5.4 Case of the MicroBlaze

- **Because Microblaze has only 1 interrupt pin, n interrupt controllers are needed when multiple devices emit interrupts**
- **The main interrupt handler will "ask" the controller what device(s) caused the interrupt and act accordingly (then branching to ISR*x*)**
  - **Here *x* must be smaller than 32. That is, a maximum number of 32 physical IRQs (Interrupt Request) to connect peripherals**
- **Further details will be seen later in the porting section of µC**





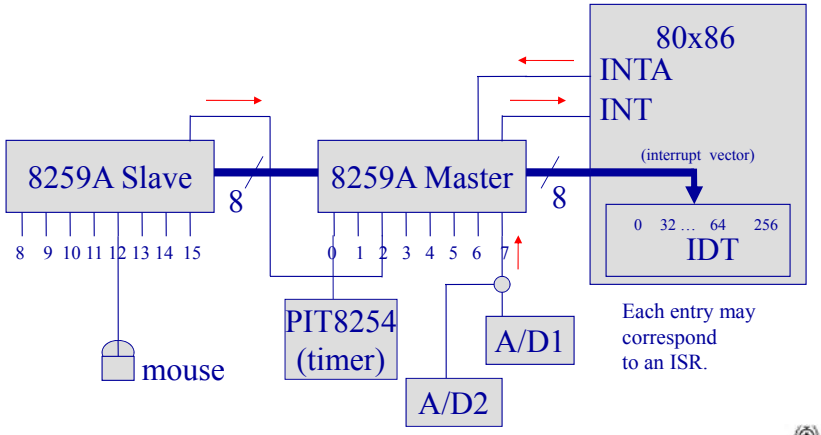
# 5.4 Case of the 80x86 processor

## *uCOS version, details in chapter 3*

```
OSTickISR: ; ISR0 (entry 0 of the IDT)

; Save context (PC and PSW have been saved by the interrupt handler,
; now registers are saved)
       pushad

; Acknowledge (through INTA pin) to the PIC8259
       mov  al,20h
       out  20h,al

; Standard uCOS processing.
       call OSIntEnter
       call OSTimeTick
       call OSIntExit

; The context is restored
       popad
       iretd
```

# 5.4 Case of the 80x86 processor

- **16 physical IRQs (Interrupt Request) pins and a maximum of 64 interrupt request lines.**

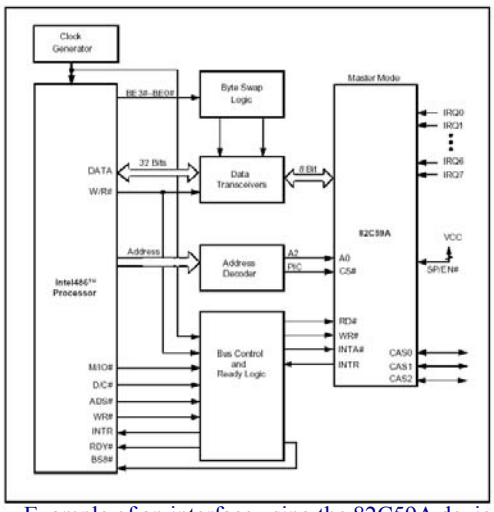








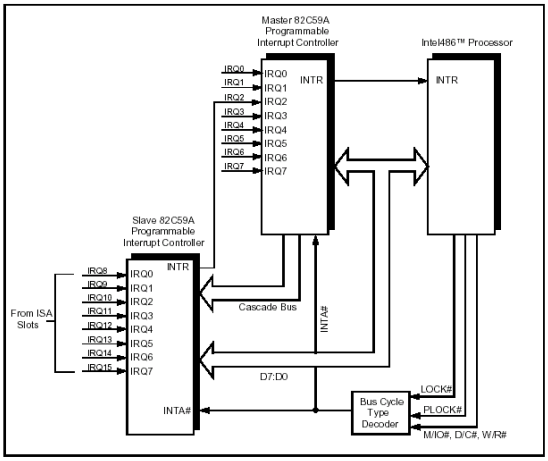

# 5.4 Case of the 80x86 processor




Example of an interface using the 82C59A device

Ref.
*Embedded Intel486™ Processor*

# 5.4 Case of the 80x86 processor




Ref.
*Embedded Intel486™ Processor*

Example of an interface using the two 82C59A devices





ÉCOLE POLYTECHNIQUE MONTRÉAL

# 5.4 Case of the 80x86 processor

## *IDT as vector interrupt*

- **A system table called Interrupt Descriptor Table (IDT), similar to the vector interrupt, associates each interrupt (external and software) or exception vector with the address of the corresponding interrupt or exception handler. The IDT must be properly initialized before the kernel enables interrupts.**





ÉCOLE POLYTECHNIQUE MONTRÉAL

# 5.4 Case of the 80x86 processor

- **Indexes 0 to 31 of the IDT vector are reserved for exceptions;**
- **Indexes 32 to 47 are reserved for external interrupts**
- **Indexes 47 to 355 are reserved for software interrupts**
- **In Linux, index 0x80 are used for software interrupt. This address is used as system call.**





# 5.4 Case of the 80x86 processor

| n (IDT Entry) | Exception | Exception Manager | Posix Signal |
|---|---|---|---|
| 0 | "Divide error" | Divide_error() | SIGFPE |
| 1 | "Debug" | Debug() | SIGTRAP |
| 2 | Réserver pour une NMI | Nmi() | None |
| 3 | "Breakpoint" | Int3() | SIGTRAP |
| 4 | "Overflow" | Overflow() | SIGSEGV |
| 5 | "Bounds check" | Bounds() | SIGSEGV |
| 6 | "Invalid opcode" | Invalid_op() | SIGILL |
| 7 | "Devide not available" | Devide_not_available() | SIGSEGV |
| 8 | "Double fault" | Double_fault() | SIGSEGV |
| 9 | "Coprocessor segment overrun" | Coprocessor_segment_overrun() | SIGFPE |
| 10 | "Invalid TSS" | Invalid_tss() | SIGSEGV |
| 11 | "Segment not present" | Segment_not_present() | SIGBUS |
| 12 | "Stack exception" | Stack_segment() | SIGBUS |
| 13 | "General protection" | General_protection() | SIGSEGV |
| 14 | "Page fault" | Page_fault() | SIGSEGV |
| 15 | Réservé à Intel | None | None |
| 16 | "Floating point error" | Coprocesseur_error() | SIGFPE |
| 17 | "Alignment check" | Alignment_check() | SIGSEGV |
| 18 à 31 | Développement futur | | |

Assignment of exceptions in IDT

- 17 exceptions (hardware interrupts)
- All exceptions are NMI
- The 3rd column indicates the name of the system function called by the ISRn (where $0 \geq n \leq 31$)

# 5.4 Case of the 80x86 processor

| n (PIC Pin No) | n + 32 (IDT Entry) | Peripheral |
|---|---|---|
| 0 | 32 | Timer |
| 1 | 33 | Keyboard |
| 2 | 34 | PIC Nesting |
| 3 | 35 | COM2 |
| 4 | 36 | COM1 |
| 6 | 38 | Floppy |
| 8 | 40 | System Clock |
| 11 | 43 | Network Interface |
| 12 | 44 | PS/2 Mouse |
| 13 | 45 | Math co. |
| 14 | 46 | Ctl IDE 1 |
| 15 | 47 | Ctl IDE 2 |

Assignment of external interrupts in IDT