

# *ECE390*

## *Computer Engineering II*

*Lecture 21*



**Dr. Zbigniew Kalbarczyk**  
**University of Illinois at Urbana- Champaign**

### *Outline*



- Multitasking, concurrency, real-time
  - DOS processes
  - Multitasking
  - Interrupt driven multitasking

# *Multitasking, Concurrency, Real-Time*

---

- Depending on how the timesharing is administered by the operating system, it can appear as though the programs are executing simultaneously
- Multitasking is difficult to achieve when running DOS
- Multitasking is not trivial, but it is not that difficult when you write an application with multitasking specifically in mind
  - you can, however, write programs that multitask under DOS if you take some precautions

## *Issues*

---

- *How do we schedule tasks so that they meet their deadlines?*
- *How do we coordinate operations performed by different tasks?*
- *How do we control access to shared resources?*
- *How do we ensure consistency of shared data?*

## *DOS Processes*

---

- Under DOS several programs can be loaded into memory at one time
- DOS provides ability to run one program at the time
- When a DOS application is running, it can load and execute some other program using the DOS EXEC function
- When an application (the parent) runs a second program (the child) the child process executes to completion and then returns to the parent
- DOS provides several functions (members of INT 21) to
  - load and execute program code (function 4Bh)
  - terminate processes (function 4Ch)
  - obtain the exit status for the process (function 4Dh)

## *Shared Memory*

---

- Problem with running different DOS programs as part of single application is inter-process communication
- How do all these programs talk to one another?
- The ideal solution would be to keep a copy of various variables in shared memory (memory that appears in the address space of two or more processes)
- Most modern multitasking operating systems provide for shared memory
- MS-DOS does not support shared memory
  - solution is to write a resident program that provides this capability

## *Multitasking*

---

- A primary drawback of DOS processes, even when using shared memory, is that each program executes to completion before returning control to the parent process
- A common paradigm is for two programs to swap control of the CPU back and forth while executing
- The basic idea behind multitasking is that one process runs for a period of time (the time quantum or time slice) and the timer interrupts the process
- The timer ISR saves the state of the process and then switches control to another process
- A timer interrupt that switches between processes is a dispatcher

## *Multitasking*

---

- Designing a dispatcher we need to decide on a policy for the process scheduling algorithm
  - **Round-robin**
    - place all processes on the queue and then rotate them
    - tradeoff between a time quantum and overhead in switching between different processes
  - **Preemptive scheduling**
    - higher priority process can interrupt lower priority process
  - **Non-Preemptive scheduling**
    - processes run until they finish.

# Multitasking Synchronization

- Cooperative concurrently executing processes must be synchronize
- The producer-consumer problem (famous problem from operating system theory)
  - one or more processes that produce data and write the data to a shared buffer
  - one or more consumers that read data from this buffer
  - system must ensure that the producers do not produce more data than the buffer can hold
  - consumers do not remove data from an empty buffer
  - preserve integrity of the buffer data structure by allowing access to only one process at a time


Z. Kalbarczyk

ECE390

## Multitasking Synchronization (example problem)

- Consider the following fragment of a consumer code (after fetching a byte of data from a circular buffer)

```
DEC          word [MyBuffer.Count]
INC          word [MyBuffer.OutPtr]
CMP          word [MyBuffer.OutPtr], MaxBufSize
JB          .NoWrap
MOV          word [MyBuffer.OutPtr], 0
```



.NoWrap:

- If after the interrupt (as it is marked) the control is transfers to another consumer which reenters this portion of code, the second consumer will malfunction
- The end result is that two consumer processes fetch the same data and then skip a byte in the circular buffer

Z. Kalbarczyk

ECE390

## *Multitasking Synchronization Critical Section*

---

- The previous problem is easy to solve by recognizing the fact that the code that manipulates the buffer data is a critical region (section)
- Restrict the execution of the critical region to one process at the time
- Reentrancy can be prevented by turning the interrupts off while in the critical region

CLI: CLear Interrupt enable

- Retain full control of CPU

STI: SeT Interrupt enable

- Allow other routines to be serviced

## *Multitasking Synchronization Critical Section*

---

- Turning the Interrupts off does not always work
  - critical regions may take considerable amount of time (seconds, minutes) and you cannot leave the interrupts off for that amount of time
  - critical region can call a procedure that turns the interrupts back on and you have no control over this (e.g., procedures that call DOS)
- Synchronization objects known as semaphores provides an efficient, general purpose mechanism for protecting critical regions
  - when a process wants to use a given resource , it waits on the semaphore
  - if no other process is currently using the resource, then the wait call sets the semaphore to in-use and returns to the process

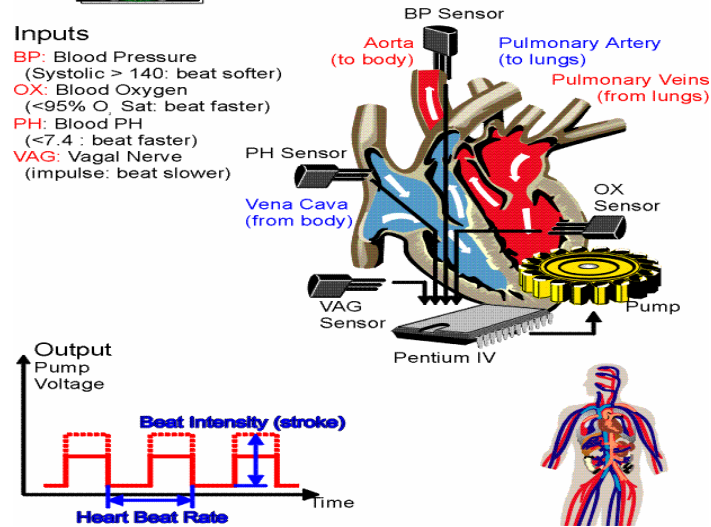
# Real-Time ISR Scheduling

## Artificial Heart Example



### JARVIC 2004

Artificial Hearts for Real Lives



Z. Kalbarczyk

ECE390

## Assumptions

- Assume that all requests are buffered
- If multiple tasks occur while the CPU is busy, they will be sequentially processed when the task is scheduled until
  - the CPU is interrupted with the higher priority interrupt or
  - there are no more tasks to execute
- Interrupts with lower-numbered priorities are always serviced before higher-numbered priorities
- The time to switch between interrupts is negligible
- Tasks are independent of each other

Z. Kalbarczyk

ECE390

## *Real-Time ISR Scheduling (cont.)*

---

- Consider ISRs for the Jarvic 2003 artificial heart

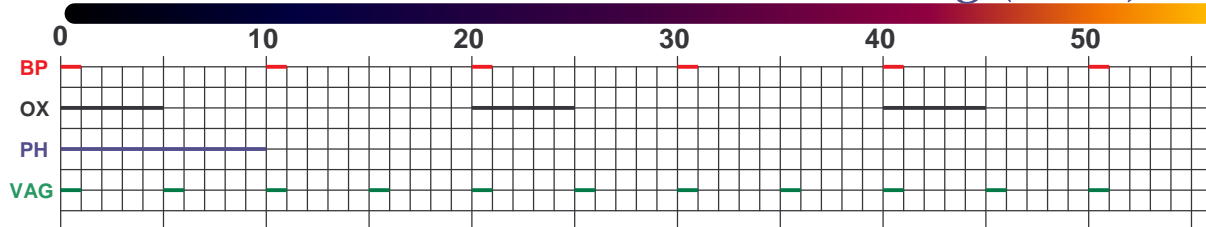
Task	Pri.	Run Time	Deadline	Event Freq.	Event Period	CPU Load
BP	1	1 ms	3 ms	100 Hz	10 ms	10%
OX	2	5 ms	20 ms	50 Hz	20 ms	25%
PH	2	10 ms	20 ms	10 Hz	100 ms	10%
VAG	3	1 ms	25 ms	200 Hz	5 ms	20%

## *Real-Time Characteristics of the Jarvic-Heart*

---

- Base on the data in the table determine the worst-case completion time for each of the tasks for two cases:
  - preemptive scheduling
  - non-preemptive scheduling

# Real-Time ISR Scheduling (cont.)



Task	Priority	Run Time [ms]	Deadline [ms]	Event Freq. [Hz]	Event Period [ms]	CPU Load (EventFreq x RunTime) [%]
<b>BP</b>	1	1	3	100	10	10
<b>OX</b>	2	5	20	50	20	25
<b>PH</b>	2	10	20	10	100	10
<b>VAG</b>	3	1	25	200	5	20

All tasks non-preemptive

$$BP = PH(10) + BP(1) = 11$$

$$OX = VAG(1) + BP(1) + PH(10) + BP(1) + OX(5) = 18$$

$$PH = VAG(1) + BP(1) + OX(5) + PH(10) = 17$$

$$VAG = PH(10) + BP(1) + BP(1) + OX(5) + VAG(1) = 18$$

Z. Kalbarczyk

All tasks preemptive

$$BP = 1$$

$$OX = BP(1) + PH(9) + BP(1) + PH(1) + OX(5) = 17$$

$$PH = BP(1) + OX(5) + PH(4) + BP(1) + PH(6) = 17$$

$$VAG = BP(1) + OX(5) + PH(4) + BP(1) + PH(6) + VAG(1) = 18$$

ECE390