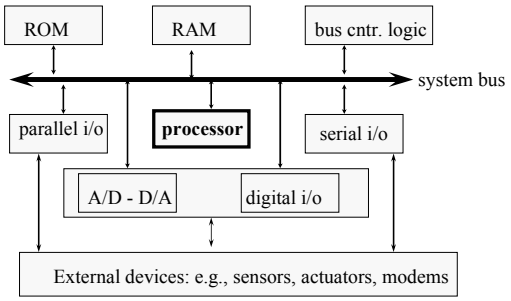
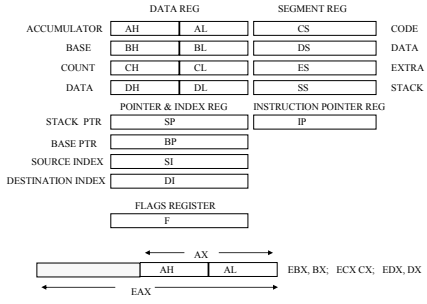
Final Exam

- You are allowed to bring in one 8''x11'' NOTES. Do you photocopy lecture slides.
- Final covers lab, homework and lectures

Small Embedded Applications



Programming View of Processor Registers



Addressing Modes

- Register addressing: MOV AX, BX
- Immediate addressing: MOV AX, 100H
- Direct memory addressing
  - MOV AX, 1000H
  - MOV DS, AX (MOV DS,1000H is not allowed)
  - MOV [7000H], AX
- Indirect register addressing: MOV AX, [SI], where the content of SI will be added into the left shifted DS content.
  - DS = 1000H and SI = 7000H,
  - Effective address: 10000H + 7000H = 17000H.
- Base register addressing: MOV AX, [BX+4H]
  - DS = 100H, BX = 600H
  - Effective address: 1000H + 600H + 4H = 1604H
- The default use of DS can be replaced by specifying another segment register, e.g., ES
  - MOV AX, ES:[100H]

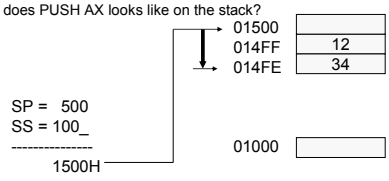
Addressing Modes

- Base index addressing: MOV [BP+SI], AH or MOV [BP][SI], AH
  - DS = 2000H, BP = 4000H, SI = 800H
  - effective address: 20000H + 4000H + 800H = 24800H
- Register relative addressing: MOV EDX, port[EBP]
  - the value of the port address at stack location (port+EBP) is loaded into EDX
- I/O port addressing
  - IN AL, 40H or OUT AL, 40H //read or write to port 40H
  - IN AL, DX or OUT DX, AL //port address stored in DX

Stack

Suppose that
- the stack segment starts at location 1000 H, SS = 100 H
- the stack size 500H, SP = 500 H
- AX = 1234 H

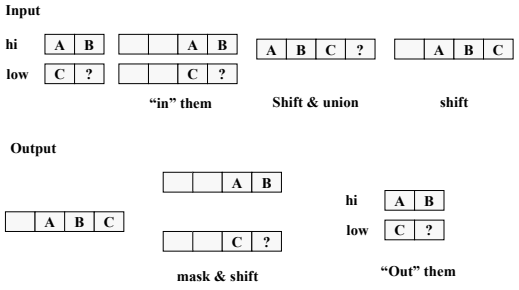
- What does PUSH AX looks like on the stack?



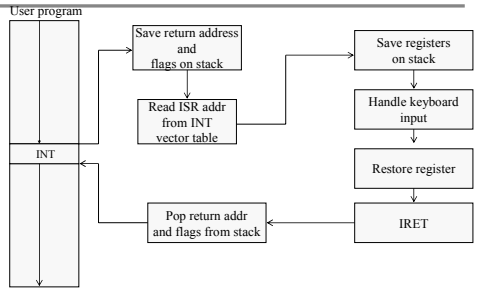
A/D and D/A Cards Addressing

- Port addresses
  - base address
  - analog input and output ports
  - digital input and output ports
- Control, status and configuration register ports:
  - Reset A/D or D/A registers
  - Initiate A/D conversions
  - Disable/enable Interrupts, Bipolar/Uni-polar selections
  - Status register: e.g., bit x A/D conversion done, bit y bipolar/uni-polar selected.
- Trade-off between single-ended mode and differential mode
- A/D and D/A mappings
  - A to D: ((Va - Va_lower_limit)/(Va_range))*(Vd_range)
  - D to A (Vd /Vd_range)*(Va_range) + Va_lower_limit
  - the odd and even mapping problem

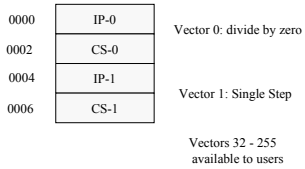
A/D - D/A Cards (12 Bit Registers)



Review of the Interrupt Model



Review: Interrupt Vector Table



Example: INT 01H, where the ISR address is computed by multiplying the interrupt type number, 01H, by 4.

Help Your Computer to Find the Right ISR

- Supposed that interrupt number 34H has not been used and we would like to store the ISR at memory location 30400H. Code segment starts at 30000H. That is, we would like the ISR starts with an offset 400H to the segment base adr. 30000H.
- IP = _____H (1)
- CS = _____H (2)
- INT Vector Address = _____ H (3)
- 1) 400 H (2) 3000 H 3) D0H (34H*4H)

- 0 PUSHA (4)
- 1 MOV DS, 0 (5)
- 2 MOV AX, 400H (6)
- 3 MOV [D0H], AX (7)
- 4 MOV AX, 3000H (8)
- 5 MOV [D1H], AX (9)
- 6 POPA (10)
- There are 2 bugs, can you spot and correct them?

- not allowed 9) D2H (2 byte)

Example: Maskable Nested Priority Interrupt

Supposed priority 3 interrupt arrives first, followed immediately priority level 5 (higher interrupt)

interrupt level 3 ISR
- save context + the old priority mask
- set priority mask level 3 (level 3 or lower disabled)
- enable interrupt
- serving interrupt

→ interrupt level 5 ISR
- •save context including the old priority mask
- •set priority mask level 5 (level 5 or lower disabled)
- •enable interrupt
- •serving interrupt
- •restore context + the old priority mask
- •IRET

←
- •serving interrupt
- •restore context + old priority mask
- •IRET

Periodic Tasks

- Periodic tasks are a common construction in real time applications, in Unix we have a primitive called sleep().
- Suppose that we want to control a device using a 20 msec task with hard real time requirements:

```
loop
    1. start_time = read_clock()
    2. wake_up_time = start_time + 20 msec
    3.   //read sensor data from the device
    4.   //do work
    5. current_time = read_clock()
    6.   //send control data to the device
    7. sleep(wake_up_time - current_time)
end_loop
```

- How many potential timing problems can you find? That is, how many “tough” real time question you should raise before accepting it?

Writing a Real-Time Periodic Task

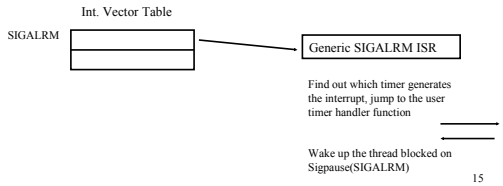
```
//include header files signal.h, time.h, errno.h, stdio.h //errno.h allows to decode the return code
void timer_handler( ) // it is invoked by the timer, not called by your software
{do your device I/O}  // use global variables to communicate between main and handler


void main () {
//ask user to input the task rate, max volt and min volt. Check for validity. If the resolution is too high and
      if the voltages are too high/low
//create your timer and initialize it
//sets up your SIGALRM handler

while (1){
sigpause(SIGALRM); //wait for signal SIGALRM
//do your computation, make the handler code as small as possible to reduce jitter.}
}
```

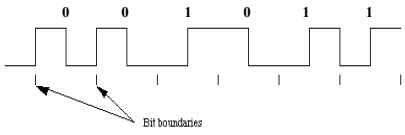

How Does Sigpause Work?

- When you call sigpause, your thread is blocked on a semaphore associated with SIGALGM.
- When your timer handler is done with data I/O part and the control returns to the generic ISR of SIGALGM, the generic handler unlocks the semaphore and wakes up the thread that does the computation part.



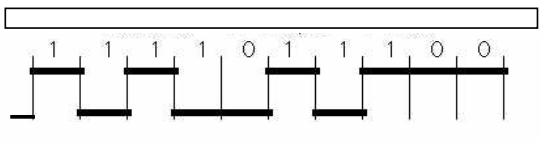
Manchester Code

- Two conflicting requirements
  - Synchronization of sender’s and receiver’s clocks
  - Bit rates. Cables have capacitance. The more is the transition, the higher is the frequencies and the more is the demand on the quality of cables
- Manchester code is “clock friendly”. It is used in 10 Mbit Ethernet. It always has a transition in the middle. One to zero transition is 0. Zero to one transition is 1. So what is the code here?

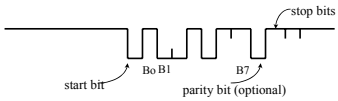


Non Return to Zero Inverted (NRZI) Code

- However, Manchester code is not bit rate friendly. It has one transition per bit, so that high speed USB and 100 Mbit dump it and use non zero return invert code.
- If there is a transition at the beginning, it is 1. Otherwise, it is zero. What is this code under NRZI?



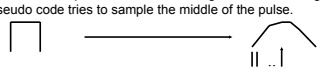
Asynchronous Communication Basics



- Sender and receivers clocks runs at the SAME rate but not synchronized.
- The clocks run 8x, 16x, or 64x of the bit rate. The idle (normal) line logic is 1, no frame being transmitted.
- when the line drops to zero after idle, it signals the start of a frame
- after 7 (or 8 if parity is enabled) bits, sender goes to 1 for at least 1 bit (stop bit) to signal the end of a frame. (Is it necessary? Why?)
- Efficiency: 7/(7 + 3) = 70%

PseudoCode for Receiving Frames

//in real world, the pulse will not be rectangular after traveling a distance. The following pseudo code tries to sample the middle of the pulse.



- Loop
  - Wait for a start bit(high-low transition) ; //causes the UART to start //receiving the data
  - Set bit-counter to zero
  - While (bit-counter<8) //the frame consists of 7 bits
    - set tick-counter to count from 0 to 7 //the clock is 16x
    - sample the signal and store the bit //approx middle of bit time
    - set tick-counter to count from 0 to 7 again
    - increment bit-counter
  - EndWhile
  - Check stop bit ; stop bit causes UART to go back to Idle state
  - Report error if there is no stop bit
- EndLoop

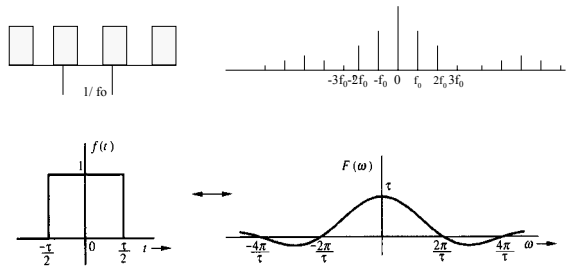
Digital Phase Lock Loop

- A phase lock loop (PLL) is an electronic circuit that adjusts a local oscillator so as to keep a constant phase angle relative to a reference signal
- In digital data communication, the signal are square waves (1's and 0's).A digital phase lock loop (DPLL) is a FSM that adjust the local clock (oscillator), so that the sending side's and receiving side's clocks are synchronized with respect to the reference signal.

Operation of DPLL

- Suppose that we use 8x sampling, Manchester code and that our synchronization patter is three bits of 1 in a row, the following is what we've got in the buffer (80H, NRZI is actually used in USB)
  - 00001111000011110000111 1 This is perfect
  - 11 00001111000011110000011 000011 ticks were lost
- Identify how many bits is too fast (slow) by sliding "the window"
- Adjusted the local clock (count in the clock counter) accordingly

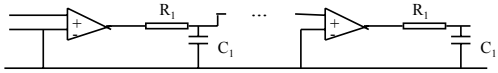
Review: Fourier Series and Transform



Review: Nyquist Rate and Aliasing

- To avoid aliasing: you must sample at least twice as fast as the highest frequency in the signal.
- When you use DSP, the statement above means that you need to sample at 2 times the highest frequency in (useful signal + noise) contained in the electronic signal.
- Alternatively, you can use an analog filter to get rid of the high frequency noise first,
  - using a low pass filter (anti-aliasing filter) to reduce the high frequency noise before sampling
  - sample at least twice of the highest signal frequency of interests
- When a signal is contaminated by noises within the signal spectrum, model based filters such as Kalman filter is called for.

A Simple Low Pass Filter



$$|a(\omega)| = |a_0| \frac{1}{\sqrt{1+(\frac{\omega}{\omega_1})^2}} \cdots \frac{1}{\sqrt{1+(\frac{\omega}{\omega_1})^2}}$$

$$\theta(\omega) = -(\arctan \frac{\omega}{\omega_1} + \ldots + \arctan \frac{\omega}{\omega_1})$$

cut off frequency: $\omega_1 = 1/R_1C_1$

Example

Suppose that we let the cut off frequency at 100 Hz and the filter has three stages.

- For a 300 Hz noise, the noise is reduced by a factor of :

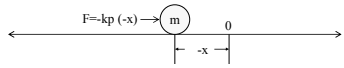
$$\left(\frac{1}{\sqrt{1+(300/100)^2}}\right)^3 = 0.031$$

- For a 30 Hz signal the phase delay is:

$$3 * \arctan(0.3) = 50 \text{ degree}$$

A Simple System with Proportional Control

- Consider a marble on a flat and perfectly leveled table.
  - Any point can be an equilibrium point (just pick one)
  - Its motion can be described by Newton's law F = ma, or x'' = F/m
  - suppose that we want to keep the marble at x = 0, by applying proportional control: F = $-k_p$ x.
    - The feedback is negative since if the marble *position error* is negative, it pushes with a positive force and vice versa.
    - K_p is a positive integer known as proportional control constant.



Marble under Proportional Control

- The system trajectories can be obtained by solving the differential equations by using Laplace transform.
  X'' = ($-k_p$ * x) /m,
  $s^2 + k_p/m = 0$, L(x'') = s^2, L(x') = s, L(x) = 1
  s = + - j sqrt(k_p/m), eigenvalues
- As you might has guessed, the marble will oscillate with frequency ω = sqrt(k_p/m) radians/sec.
- The system is marginally stable, since the real part of eigenvalue is zero.

Marble Under P+D Control

To determine of if a system is stable, we check if the real part of the system's eigenvalue is negative.

X'' = F/m, where F = -Kp x - Kd x'
= -Kd/m x' - Kp/m x

X'' + Kd/m x' + Kp/m x = 0

s^2 + Kd/m s + Kp/m = 0

s =[-Kd/m +- sqrt($(Kd/m)^2$- 4 Kp/m)]/2

The marble under P+D control is stable, because RE(S)= -Kd/2m < 0.
Generally,by picking the appropriate Kp and Kd, we can achieve stability as indicated by the eigenvalue of the system.

Integral Control

- An effective way to deal with steady state errors due to load changes is to add an integral control.
  - Use it lightly since too large may lead to
    - overshoot
    - oscillations
    - even instability
  - Check the system eigenvalues to make sure that the system is not too close to the edge of instability.

PID Control of Marble

X'' = F/m, where F = -Kp x - Kd x' - $\int$x

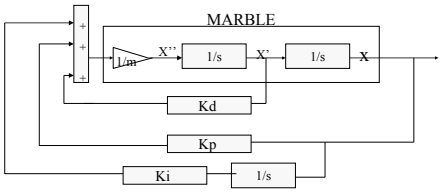
s^2 + Kd/m s + Kp/m + ki/m 1/s = 0

s^3 + kd/m s^2 + kp/m s + ki/m = 0

We can solve the equation to see if the real part of the eigenvalues are still negative. Better yet, use MATLAB

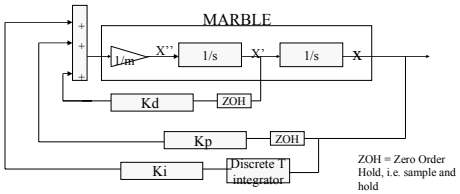
> [A, B, C, D] = linmod('marble')
>eig(A)

Marble with PID controller



[A, B, C, D] = linmod('marble')

Digital Control Simulation

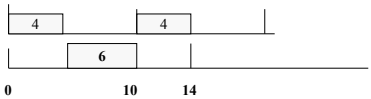


[A, B, C, D] = dlinmod('marble')

Measure of Merit

	Real time systems	*Non-real time systems*
performance	Schedulable Utilization (schedulability)	*throughput*
responsiveness	Worst case response time of each tasks	*Average response time*
overload	*Stability (getting critical tasks done)*	*Fairness*

Less Than 100% Utilization but not Schedulable



4/10 + 6/14 = 0.83

note: if tasks periods are harmonic, e.g, 2, 4, 8, 16 etc, the utilization bound is 1

Schedulability: UB Test

- Utilization bound(UB) test[Liu73]: a set of n independent periodic tasks scheduled by the rate monotonic algorithm will always meet its deadlines, for all task phasing, if

$$\frac{C_1}{T_1} + \ldots + \frac{C_n}{T_n} \leq U_{(n)} = n(2^{1/n} - 1)$$

- U(1) = 1.0 U(4) = 0.756 U(7) = 0.728
- U(2) = 0.828 U(5) = 0.743 U(8) = 0.724
- U(3) = 0.779 U(6) = 0.734 U(9) = 0.720

- For harmonic task sets, the utilization bound is U(n)=1.00 for all n. For large n, the bound converges to *ln 2* ~ 0.69.

- Conventions, task 1 has shorter period than task 2 and so on.

Exact Schedulability Test

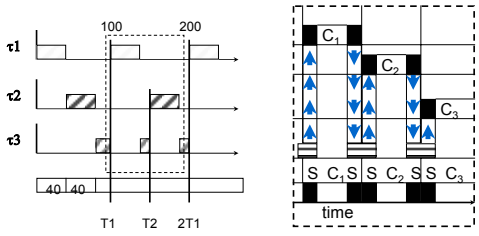
$$a_{n+1} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{a_n}{T_j} \right\rceil C_j \qquad \text{where } a_0 = \sum_{j=1}^{i} C_j$$

Test terminates when $a_{n+1} > T_i$ (not schedulable) or when $a_{n+1} = a_n \leq T_i$ (schedulable).

The subscript to *a* indicates the number of iterations in the calculation. The index *i* indicates it is the *ith* task being checked.

The index *j* runs from 1 to i-1, i.e all the higher priority tasks. Recall from the convention - task 1 has a higher priority than task 2 and so on.

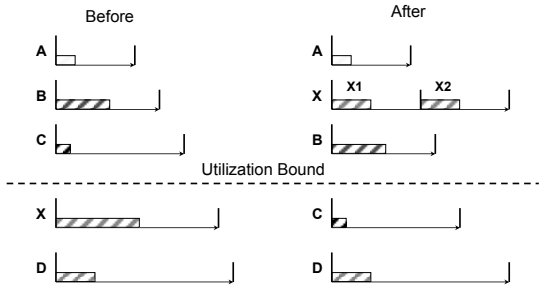
Modeling Task Switching as Execution Time



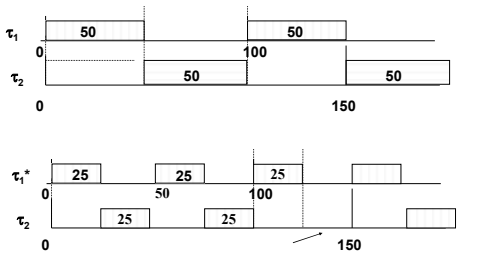
Two scheduling actions per task (start and end of period)

OS

Promoting a Critical Task



Using Period Transformation to Improve Processor Utilization

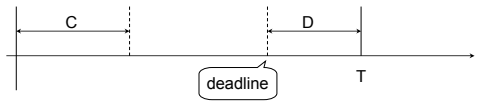


25 unused units are created by period transformation which can now be used by these two tasks.

Modeling Preperiod Deadlines

Suppose task τ, with compute time C and period T, has a preperiod deadline D.

- In UB tests, pre-period deadline can be modeled as if the task has a longer execution time (C+D), because if the task has execution time (C+D) can finish before time T, then we know it must finish D units before T if it has only execution time C.
- In exact schedulability test, just move the deadline from T to (T - D)



Task Switching and Pre-period Deadline

Suppose that task 2 has D_2 unit of preperiod deadline, we just add D_2 to task 2's execution time ***LOCALLY*** (Why?)

$$\tau_1 \quad \frac{(C_1 + 2S)}{T_1} \le U(1)$$

$$\tau_2 \quad \frac{(C_1 + 2S)}{T_1} + \frac{(C_2 + 2S + D_2)}{T_2} \le U(2)$$

$$\tau_3 \quad \frac{(C_1 + 2S)}{T_1} + \frac{(C_2 + 2S)}{T_2} + \frac{(C_3 + 2S)}{T_3} \le U(3)$$

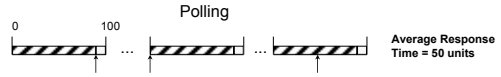
Aperiodic: The Fundamental Idea

- Rate monotonic scheduling is a periodic framework. To handle apperiodics, we must convert the aperiodic event service to into a periodic framework.
- Except in the case of using interrupt handler, the basic idea is to periodically allocate CPU cycles to each stream of aperiodic requests.
  - polling
  - Handle it within an interrupt handler
  - sporadic server

Polling

- The simplest approach is polling:
  - buffer all aperiodic requests
  - serve the buffered requests periodically with
    - a budget C
    - and a period T
    - the priority is assigned according to the service periodic (higher rate, higher priority just like periodic tasks)
  - The utilization of a polling server is simply C/T
  - NOTE: each time, the server will keep serve buffered request until either
    - all the buffered requests are serviced, unused budget, if any, will be discarded
    - or the budget C runs out. In this case, the service suspends until the beginning of next period with a new C budget again

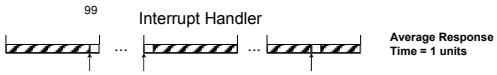
Performance of a Polling Server



Service delay of a polling server is, on average, roughly half of the the period.

- polling looks at request buffers only once per period. (like a city bus)
- higher polling rate will give better response time.
- low polling will have lower overhead

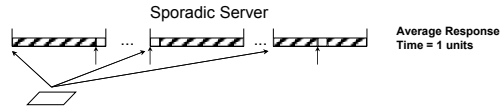
Using Interrupt Handler



- Handle aperiodic requests within interrupt handler gives the best performance, since interrupt handlers runs at priority higher than applications
- Precisely for the same reason, a larger amount of such interrupts would cause deadlines of periodic tasks to missed.
- It is a solution with serious side effects. Use it ONLY as a last resort for short fuse hard deadline aperiodic request such as power failure warning.

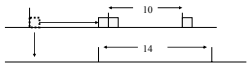
Sporadic Server

- Sporadic server is similar to paying an employee a monthly (periodic) salary but each person to spend his/her money whenever they like (aperiodic usages)
- The idea is to deposit a budget the beginning of each period and aperiodic requests, when arrive, can use the budget if there is any.
- (If there is no budget left, wait until there is)
- Unused budget cannot be carry over to next period



Sporadic Server Implementation Issues

- The idea is to make the service of aperiodic events to behave like a regular periodic task, i.e. if the sporadic server has a budget C and a period of T_s, then it should preempt any lower priority task, of period T, ceil(T/T_s) times.



- Because of the deferred usage of the budget, a straightforward implementation would create more than ceil(T/T_s) occurrences of the preemption.
- In the example above, it creates 3 preemptions, more than ceiling (14/10)

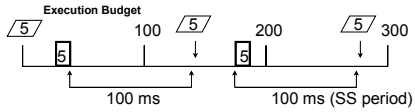
Sporadic Server (SS)

Modeled as periodic tasks

- Fixed execution budget (C)
- Replenishment interval (T)

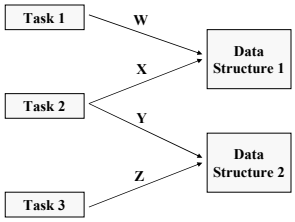
Priority is based on T, adjusted to meet requirements

Replenishment occurs one "period" after it is used up. (Like refill your motor oil if it is low). More aggressive replenishment is possible and would give somewhat better response time.



Sample Problem: Worst-Case Blocking Times

- You need to understand how basic priority inheritance protocol and priority ceiling protocol work, including nested semaphores.



Worst Case Blocking Time (BIP)

- Un-nested semaphores
  - First, examine task 3. Recall that *blocking* is when a lower priority task delays the execution of a higher priority task. Since task 3 has the lowest priority, its blocking time will be 0.
  - For task 2, it could only be blocked by task 3. Thus, its B is equal to Z.
  - Similarly, task 1's B is equal to X. Because semaphores cannot be nested, data structure 2 will not affect task 1.
- Nested semaphores
  - Task 3 and 2 are the same as the un-nested case.
  - For task 1, imagine the worst case sequence of events.

Worst-Case Blocking Times (BIP)

- Nested semaphores
  - The worst sequence for Task 1:
    1. Task 3 acquires data structure 2.
    2. Task 2 acquires data structure 1.
    3. Task 2 asks for data structure 2 (but cannot get it).
       - What is Task 3's priority now?
       - Task 3 has priority of 2.
    4. Task 1 asks for data structure 1.
       - What are Task 2's and 3's priorities now?
       - They both have priority of 1.
    5. Task 3 finishes its critical section (Z units).
    6. Task 2 acquires data structure 2 and finishes its critical section (Y units).
    7. Task 2 finishes its critical section for data structure 1 (X units).
  - Total time: X + Y + Z.

Summary

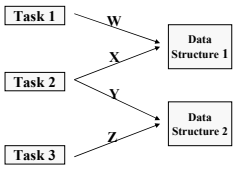
- For a nested semaphore itself, the blocking time it may inflict is the entire nested locks, that is x+y

BIP

- Nested semaphore may link the blocking time from a semaphore that is not directly used by a task. In this case, the blocking time that can be caused by z.

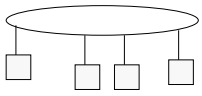
PCP

- Under PCP, the linking is not possible. If Task 3 locks data structure 2 first, Task 2 cannot lock data structure 1 (why?).



Simple Network Scheduling

- In distributed systems, signals are captured at one place, transmitted over a network and then utilized by the destination node, for example, video data captured at one place, sent over the network and displayed at another node.
- The simplest form of a deterministic network is a token ring such as FDDI, a 100 M Bit/sec LAN.
- Under FDDI, the messages queued at each node can be queued in either FIFO or priority order. We should use priority queueing and assign messages with GRMS priorities.



Network Scheduling: FDDI

- FDDI can be configured into synchronous mode for real time applications. We shall assume that all messages are periodic. (If not, we will use sporadic servers, so that the resulting aperiodic data streams can be analyzed as if they are periodic.)
  - The source node is responsible to remove its own packets when they come around the ring.
  - U_i is the percentage of ring bandwidth that will be used by a node to transmit data on the ring. $U_i = U_{i1} + U_{i2} + \ldots + U_{in}$, U_{ik}
  - For example: at node 2, there are two periodic messages
  - Message 1 with transmission time 1 and period 10
  - Message 2 with transmission time 2 and period 40
  - $U_2 = U_{21} + U_{22} = 0.1 + 0.05 = 0.15$

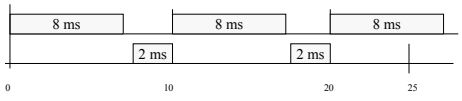
FDDI (cont.)

- Walk time, W, the time a token circulates the ring once when there is no transmission.
- Node i's hold time, H_i, is maximum time that Node i can hold the token and transmit data.
- In synchronous mode, the target token rotation time, TTRT, is the sum of walk time and hold times around the ring.
- TTRT is a design parameter and should be no longer than the shortest period of all the periodic messages.
- In token ring, hold times are used for transmission. Walk time represents the sum of each node's waiting time for the token. Walk time is overhead.

Proportional Allocation of Bandwidth

- How can we determine the hold time for each node?
- Proportional allocation is a simple algorithm that makes hold time at each node proportion to its bandwidth requirements:
  - Initial holding time: H_i = (TTRT - W)* (U_i/U), where Ui is the total utilization of node i and U is the total bandwidth of the ring
  - If a particular node cannot schedule all its messages while others can, adjustments to Hi's can be made.

Equivalent Task Set



Task 1: C1 = 8 ms, T1 = 10

Task 2: C2 = 4 ms, T2 = 25

At each node, the "Task 1" is known as the TTRT Task, its execution time is (TTRT – HOLD TIME of THIS NODE) and the period is just TTRT.

Think about this: from the viewpoint of THIS node, every 10 ms, the rings is taken away for (10 – 2) = 8 ms, as if there were a high priority task with period 10 ms and executive time 8 ms.