


### Administrative matter

- Homework #4
  - Due Thursday 10/24, 6:10PM
  - Update/clarification posted yesterday
- Midterm
  - Tuesday 10/29, in class
  - Bring a calculator!
  - Covers
    - 85% ESD: chapter 1-8 (with emphasis on later chapters)
    - 15%: Labs 1-6 (with emphasis on later labs)



1

### Outline

- Models v. Languages
- State Machine Model
  - FSM/FSMD
  - HCFSM and Statecharts Language
  - Program-State Machine (PSM) Model
- Concurrent Process Model
  - Communication
  - Synchronization
  - Implementation
- Dataflow Model
- Real-Time Systems

2

### Concurrent process model

- Function contains two or more concurrently executing subtasks
- Many systems easier to describe with concurrent process model
  - inherently multitasking
- E.g., Read 2 numbers X and Y
  - Display "Hello world." every X seconds
  - Display "How are you?" every Y seconds
- More effort required with SP or SM model

```

ConcurrentProcessExample() {
  x = ReadX()
  y = ReadY()
  Call concurrently:
  PrintHelloWorld(x) and
  PrintHowAreYou(y)
}
PrintHelloWorld(x) {
  while( 1 ) {
    print "Hello world."
    delay(x);
  }
}
PrintHowAreYou(x) {
  while( 1 ) {
    print "How are you?"
    delay(y);
  }
}
    
```

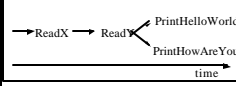
Simple concurrent process example

3

### Concurrent process model

```

ConcurrentProcessExample() {
  x = ReadX()
  y = ReadY()
  Call concurrently:
  PrintHelloWorld(x) and
  PrintHowAreYou(y)
}
PrintHelloWorld(x) {
  while( 1 ) {
    print "Hello world."
    delay(x);
  }
}
PrintHowAreYou(x) {
  while( 1 ) {
    print "How are you?"
    delay(y);
  }
}
    
```



Subroutine execution over time

```

Enter X: 1
Enter Y: 2
Hello world. (Time = 1 s)
Hello world. (Time = 2 s)
How are you? (Time = 2 s)
Hello world. (Time = 3 s)
How are you? (Time = 4 s)
Hello world. (Time = 4 s)
...
    
```

Sample input and output

Simple concurrent process example

4

### Typical embedded systems using concurrency

#### Heartbeat Monitoring System

<b>Task 1:</b> Read pulse If pulse < Lo then Activate Siren If pulse > Hi then Activate Siren Sleep 1 second Repeat	<b>Task 2:</b> If B1/B2 pressed then Lo = Lo +/- 1 If B3/B4 pressed then Hi = Hi +/- 1 Sleep 500 ms Repeat
--	--

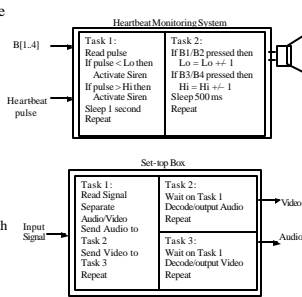
#### Set-top Box

<b>Task 1:</b> Read Signal Separate Audio/Video Send Audio to Task 2 Send Video to Task 3 Repeat	<b>Task 2:</b> Wait on Task 1 Decode/output Audio Repeat
	<b>Task 3:</b> Wait on Task 1 Decode/output Video Repeat

5

### Typical embedded systems using concurrency

- Both examples have 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



6

### Process

- A unit of execution
- Executes concurrently with other processes in model
- Thought of as infinite loop executing sequential statements
  - May never terminate under normal condition
- Process states:
  - Running
    - Currently being executed
  - Runnable
    - Ready and executable
    - It can't be running because of resource constraints, just waiting its turn
  - Blocked
    - Not ready to be executed
    - Could be waiting for a device/processor to complete

7

CS122A: Embedded System Design, Fall02

### Basic operations: create and terminate

- Computation model defines objects and operations on objects
  - Process is object encapsulating some portion of system functionality
  - Basic operations: create, terminate, suspend, resume, and join
- Create
  - Creates a new process
    - Initializes associated data
    - Starts execution
  - Like procedure call in SP model except process does not wait for return
    - Both execute concurrently
- Terminate
  - Terminates an executing process
    - Destroys all associated data
  - Performed by one process on another
    - For handling exceptions like detecting error condition

8

CS122A: Embedded System Design, Fall02

### Basic operations: suspend, resume, and join

- Suspend
  - Suspends execution of already created process
    - State and current instruction location must be saved
- Resume
  - Resume execution of suspended process
    - Restore state and start at saved current instruction location
- Join
  - Process suspended until to-be-joined process ends execution
  - Important for process synchronization

9

CS122A: Embedded System Design, Fall02

### Communication among processes

- Communication b/t processes key for concurrent process model
- Two common methods used: shared memory, message passing
- Shared memory
  - Processes communicate by reading/writing same memory locations
  - Very efficient, easy to implement
  - May be error prone (no control synchronization)
- Message passing
  - Data exchanged between two processes in explicit fashion
    - Sending process performs special operation, "send"
    - Receiving process must perform "receive", to receive the data
    - Must explicitly specify destination and source, respectively

10

CS122A: Embedded System Design, Fall02

### Message Passing

```

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 */
    }
}
    
```

11

CS122A: Embedded System Design, Fall02

### Consumer-producer problem (incorrect solution)

- Shared memory
  - 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

```

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[count] = data;
09:         count = count + 1;
10:     }
11: }
12: void processB() {
13:     int i;
14:     while( 1 ) {
15:         while( count == 0 ); /*loop*/
16:         data = buffer[count-1];
17:         count = count - 1;
18:         consume(&data);
19:     }
20: }
21: void main() {
22:     create_process( processA );
23:     create_process( processB );
24: }
    
```

12

CS122A: Embedded System Design, Fall02

### Consumer-producer problem (incorrect solution)

- Error when both processes try to update *count* concurrently (lines 10 and 19) and the following execution sequence occurs:
  - 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[count] = data;
09:
10:         count = count + 1;
11:     }
12: }
13: void processB() {
14:     int i;
15:     while( 1 ) {
16:         while( count == 0 ) ; /*loop*/
17:         data = buffer[count-1];
18:
19:         count = count - 1;
20:         consume(&data);
21:     }
22: }
23: void main() {
24:     create_process(processA);
25:     create_process(processB);
26: }
    
```

CS122A: Embedded System Design, Fall 02

### Mutual exclusion

- Certain sections of code should not be performed concurrently
- Critical section
  - Possibly noncontiguous section of code where simultaneous updates, by multiple processes to a shared memory location, can occur
  - Critical section routine used to prevent simultaneous updates
- 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
  - Allow/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 acquiring process unlock and exits critical section
  - These processes will then be placed in runnable state
    - compete for lock again

14

CS122A: Embedded System Design, Fall 02

### Consumer-producer problem

- *mutex* is used to ensure critical sections are executed in mutual exclusion
- Following the same execution sequence as before....

```

01: data_type buffer[N];
02: int count = 0;
03: mutex count_mutex;
04: void processA() {
05:     int i;
06:     while( 1 ) {
07:         produce(&data);
08:         while( count == N ) ; /*loop*/
09:         buffer[count] = data;
10:
11:         count_mutex.lock();
12:         count = count + 1;
13:         count_mutex.unlock();
14:     }
15: }
16: void processB() {
17:     int i;
18:     while( 1 ) {
19:         while( count == 0 ) ; /*loop*/
20:         data = buffer[count-1];
21:
22:         count_mutex.lock();
23:         count = count - 1;
24:         count_mutex.unlock();
25:         consume(&data);
26:     }
27: }
    
```

CS122A: Embedded System Design, Fall 02

### Consumer-producer problem

- A/B execute lock operation on *count\_mutex*
- Either A or B acquire lock
  - Say B acquires it
  - A will be put in blocked
- 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*
  - A is placed in runnable
- 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* has correct value of 3

```

01: data_type buffer[N];
02: int count = 0;
03: mutex count_mutex;
04: void processA() {
05:     int i;
06:     while( 1 ) {
07:         produce(&data);
08:         while( count == N ) ; /*loop*/
09:         buffer[count] = data;
10:
11:         count_mutex.lock();
12:         count = count + 1;
13:         count_mutex.unlock();
14:     }
15: }
16: void processB() {
17:     int i;
18:     while( 1 ) {
19:         while( count == 0 ) ; /*loop*/
20:         data = buffer[count-1];
21:
22:         count_mutex.lock();
23:         count = count - 1;
24:         count_mutex.unlock();
25:         consume(&data);
26:     }
27: }
    
```

CS122A: Embedded System Design, Fall 02

### Deadlock

- A condition where 2 or more processes are blocked waiting for the other to unlock
  - Cannot execute unlock so will wait forever
- Example has 2 different critical sections of code that can be accessed simultaneously
  - 2 locks needed (*mutex1*, *mutex2*)
  - Following sequence produces deadlock...

```

01: mutex mutex1, mutex2;
02: void processA() {
03:     while( 1 ) {
04:         ...
05:         mutex1.lock();
06:         /* critical section 1 */
07:         mutex2.lock();
08:         /* critical section 2 */
09:         mutex2.unlock();
10:         /* critical section 1 */
11:         mutex1.unlock();
12:     }
13: }
14: void processB() {
15:     while( 1 ) {
16:         ...
17:         mutex2.lock();
18:         /* critical section 2 */
19:         mutex1.lock();
20:         /* critical section 1 */
21:         mutex1.unlock();
22:         /* critical section 2 */
23:         mutex2.unlock();
24:     }
25: }
    
```

17

CS122A: Embedded System Design, Fall 02

### 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*
- Various deadlock detection and deadlock resolving algorithm available

```

01: mutex mutex1, mutex2;
02: void processA() {
03:     while( 1 ) {
04:         ...
05:         mutex1.lock();
06:         /* critical section 1 */
07:         mutex2.lock();
08:         /* critical section 2 */
09:         mutex2.unlock();
10:         /* critical section 1 */
11:         mutex1.unlock();
12:     }
13: }
14: void processB() {
15:     while( 1 ) {
16:         ...
17:         mutex2.lock();
18:         /* critical section 2 */
19:         mutex1.lock();
20:         /* critical section 1 */
21:         mutex1.unlock();
22:         /* critical section 2 */
23:         mutex2.unlock();
24:     }
25: }
    
```

18

CS122A: Embedded System Design, Fall 02

### Outline

- Models v. Languages
- State Machine Model
  - FSM/FSMD
  - HCFSM and Statecharts Language
  - Program-State Machine (PSM) Model
- Concurrent Process Model
  - Communication
  - Synchronization
  - Implementation
- Dataflow Model
- Real-Time Systems

19

### Synchronization among processes

- Concurrently running processes may need to synchronize
  - When a process must wait for:
    - another process to compute some value
    - Other processes to reach known points in their execution
    - Other processes signal some conditions
- Recall producer-consumer problem
  - processA must wait if buffer is full
  - processB must wait if buffer is empty
  - Busy-waiting
    - Process executing loops instead of being blocked
    - CPU time wasted
- More efficient methods
  - Join operation, and blocking send and receive
  - Both block the process so it doesn't waste CPU time
    - Condition variables and monitors

20

### Condition variables

- Condition variable is an object that has
  - Signal operation
  - Wait operation
- Process performs a wait on a condition variable
  - the process is blocked until another process performs a signal
    - Not busy-waiting, CPU is free to perform other task explicitly
- How is this done?
  - Process A acquires lock on a mutex
    - Process A performs wait, passing this mutex
    - Causes mutex to be unlocked
  - Process B can now acquire lock on same mutex
  - Process B enters critical section
    - Computes some value and/or make condition true
  - Process B performs signal when condition true
    - Causes process A to implicitly reacquire mutex lock
    - Process A becomes runnable

21

### Condition variable example: consumer-producer

- 2 condition variables
  - *buffer\_empty*
    - Signals at least 1 free location available in *buffer*
  - *buffer\_full*
    - Signals at least 1 valid data item in *buffer*
- *processA*:
  - produces data item
  - acquires lock (*cs\_mutex*) for critical section
  - checks value of *count*

```

01: data_type buffer[N];
02: int count = 0;
03: mutex cs_mutex;
04: condition buffer_empty, buffer_full;
05: void processA() {
06:   int i;
07:   while( 1 ) {
08:     produce(&data);
09:     cs_mutex.lock();
10:     if( count == N ) buffer_empty.wait(cs_mutex);
11:     buffer[count] = data;
12:     count = count + 1;
13:     cs_mutex.unlock();
14:     buffer_full.signal();
15:   }
16: }
17: void processB() {
18:   int i;
19:   while( 1 ) {
20:     cs_mutex.lock();
21:     if( count == 0 ) buffer_full.wait(cs_mutex);
22:     data = buffer[count-1];
23:     count = count - 1;
24:     cs_mutex.unlock();
25:     buffer_empty.signal();
26:     consume(&data);
27:   }
28: }
    
```

22

### Condition variable example: consumer-producer

- if *count = N, buffer is full*
  - performs wait operation on *buffer\_empty*
  - this releases the lock on *cs\_mutex* allowing *processB* to enter critical section, consume data item and free location in *buffer*
  - *processB* then performs signal

```

01: data_type buffer[N];
02: int count = 0;
03: mutex cs_mutex;
04: condition buffer_empty, buffer_full;
05: void processA() {
06:   int i;
07:   while( 1 ) {
08:     produce(&data);
09:     cs_mutex.lock();
10:     if( count == N ) buffer_empty.wait(cs_mutex);
11:     buffer[count] = data;
12:     count = count + 1;
13:     cs_mutex.unlock();
14:     buffer_full.signal();
15:   }
16: }
17: void processB() {
18:   int i;
19:   while( 1 ) {
20:     cs_mutex.lock();
21:     if( count == 0 ) buffer_full.wait(cs_mutex);
22:     data = buffer[count-1];
23:     count = count - 1;
24:     cs_mutex.unlock();
25:     buffer_empty.signal();
26:     consume(&data);
27:   }
28: }
    
```

23

### Condition variable example: consumer-producer

- if *count < N, buffer is not full*
  - *processA* inserts data into *buffer*
  - increments *count*
  - signals *processB* making it runnable if it has performed a wait operation on *buffer\_full*

```

01: data_type buffer[N];
02: int count = 0;
03: mutex cs_mutex;
04: condition buffer_empty, buffer_full;
05: void processA() {
06:   int i;
07:   while( 1 ) {
08:     produce(&data);
09:     cs_mutex.lock();
10:     if( count == N ) buffer_empty.wait(cs_mutex);
11:     buffer[count] = data;
12:     count = count + 1;
13:     cs_mutex.unlock();
14:     buffer_full.signal();
15:   }
16: }
17: void processB() {
18:   int i;
19:   while( 1 ) {
20:     cs_mutex.lock();
21:     if( count == 0 ) buffer_full.wait(cs_mutex);
22:     data = buffer[count-1];
23:     count = count - 1;
24:     cs_mutex.unlock();
25:     buffer_empty.signal();
26:     consume(&data);
27:   }
28: }
    
```

24

### Monitors

- Monitor guarantees only 1 process can execute inside monitor at a time
- (a) Process X executes while Process Y has to wait
- (b) Process X performs wait on a condition
  - Process Y allowed to enter and execute
- (c) Process Y signals condition Process X waiting on
  - Process Y blocked
  - Process X allowed to continue executing
- (d) Process X finishes executing in monitor or waits on a condition again
  - Process Y made runnable again

25

### Monitor example: consumer-producer

- Single monitor encapsulates both processes along with *buffer* and *count*
- One process will be allowed to begin executing first
- If *processB* allowed to execute first

```

01: Monitor {
02:   data_type buffer[N];
03:   int count = 0;
04:   condition buffer_full, condition buffer_empty;
05:   void processA() {
06:     int i;
07:     while( 1 ) {
08:       produce(&data);
09:       if( count == N ) buffer_empty.wait();
10:       buffer[count] = data;
11:     }
12:   }
13:   void processB() {
14:     int i;
15:     while( 1 ) {
16:       if( count == 0 ) buffer_full.wait();
17:       data = buffer[count-1];
18:       count = count - 1;
19:       buffer_empty.signal();
20:       consume(&data);
21:     }
22:   }
23: }
    
```

26

### Monitor example: consumer-producer

- Will execute until it finds *count = 0*
- Will perform wait on *buffer\_full* condition variable
- *processA* now allowed to enter monitor and execute
- *processA* produces data item
- finds *count < N* so writes to *buffer* and increments *count*
- *processA* performs signal on *buffer\_full* condition variable
- *processA* blocked
- *processB* reenters monitor and continues execution, consumes data, etc.

```

01: Monitor {
02:   data_type buffer[N];
03:   int count = 0;
04:   condition buffer_full, condition buffer_empty;
05:   void processA() {
06:     int i;
07:     while( 1 ) {
08:       produce(&data);
09:       if( count == N ) buffer_empty.wait();
10:       buffer[count] = data;
11:       count = count + 1;
12:       buffer_full.signal();
13:     }
14:   }
15:   void processB() {
16:     int i;
17:     while( 1 ) {
18:       if( count == 0 ) buffer_full.wait();
19:       data = buffer[count-1];
20:       count = count - 1;
21:       buffer_empty.signal();
22:       consume(&data);
23:     }
24:   }
25: }
    
```

27

### 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

28

### Concurrent process model: implementation

- (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

29

### Implementation: multiple processes sharing single processor

- Can 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
  - E.g., Hello World concurrent program would look like:
 

```

• I = 1; T = 0;
• while (1) {
•   Delay(I); T = T + 1;
•   if T modulo X is 0 then call PrintHelloWorld
•   if T modulo Y is 0 then call PrintHowAreYou
• }
                    
```

30

### Implementation: multiple processes sharing single processor

- Can use multitasking operating system
  - Much more common
  - Operating system
    - schedules processes
    - allocates storage
    - interfaces to peripherals
  - Real-time operating system (RTOS) guarantee execution rate constraints
  - Languages having built-in processes
    - Java, Ada
  - Sequential programming language with support for concurrent processes
    - C, C++, etc. using POSIX threads
- Can in-line processes to sequential program with static scheduling
  - Less overhead (no operating system)
  - More complex/harder to maintain

31  
CS122A: Embedded System Design, Fall02

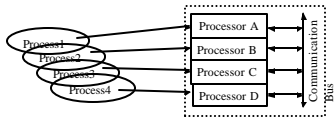
### Processes vs. threads

- Different meanings than 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
- In embedded system, distinctions are blurred

32  
CS122A: Embedded System Design, Fall02

### Implementation: suspending, resuming, and joining

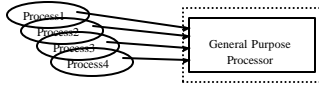
- Multiple processes mapped to single-purpose processors
  - Built into processor's implementation
  - Extra input signal that is asserted when process suspended
  - Additional logic needed for determining process completion
    - Extra output signals indicating process done



33  
CS122A: Embedded System Design, Fall02

### Implementation: suspending, resuming, and joining

- Multiple processes mapped to single general-purpose processor
  - Built into programming language or special multitasking library
    - POSIX
  - Language or library may rely on operating system to handle



34  
CS122A: Embedded System Design, Fall02

### Implementation: process scheduling

- Multiple concurrent processes implemented on single GPP
  - Not true multitasking
  - Must meet timing constraints
- Scheduler
  - Special process that decides when/for how long each process is executed
  - Implemented as preemptive or nonpreemptive scheduler
  - Preemptive
    - Determines when a process is preempted to allow another process to execute
    - Determines which process will be next to run
  - Nonpreemptive
    - Only determines which process is next after current process finishes

35  
CS122A: Embedded System Design, Fall02

### Scheduling

- FIFO
  - Runnable processes added to end of FIFO as created or become runnable
  - Front process removed from FIFO
    - When current process finished
    - When current process preempted
- Priority Scheduling
  - Process with highest priority always selected first by scheduler
  - Determined statically during creation
    - SDF, RMS
  - Determined dynamically during execution
    - EDF
- Priority queue
  - Runnable processes added as created or become runnable
  - Process with highest priority chosen when new process needed
    - May be preemptive or non-preemptive

36  
CS122A: Embedded System Design, Fall02

### Priority assignment

- **Period of process**
  - Repeating time interval the process must complete one execution within
    - E.g., period = 100 ms
    - Process must execute once every 100 ms
  - Usually determined by the description of the system
    - E.g., refresh rate of display is 27 times/sec
    - Period = 37 ms
- **Execution deadline**
  - Amount of time process must be completed by after it has started
    - E.g., execution time = 5 ms, deadline = 20 ms, period = 100 ms
    - Process must complete execution within 20 ms after it has begun
  - An example run
    - Process begins at start of period, runs for 4 ms then is preempted
    - Process suspended for 14 ms, then runs for the remaining 1 ms
    - Completed within 4 + 14 + 1 = 19 ms which meets deadline of 20 ms
    - Without deadline process could be suspended for much longer

37

CS122A: Embedded System Design, Fall02

### Priority assignment

- **Rate monotonic scheduling (RMS)**
  - Processes with shorter periods have higher priority
  - Typically used when execution deadline = period
  - Static priority
- **Earliest Deadline First (EDF)**
  - Processes with shorter deadlines have higher priority
  - Typically used when execution deadline < period
  - Dynamic priority

Process	Period	Priority
A	25 ms	5
B	50 ms	3
C	12 ms	6
D	100 ms	1
E	40 ms	4
F	75 ms	2

Process	Deadline	Priority
G	17 ms	5
H	50 ms	2
I	32 ms	3
J	10 ms	6
K	140 ms	1
L	32 ms	4

38

CS122A: Embedded System Design, Fall02

### Rate Monotonic Scheduling

- **Assumption:**
  - Each task has fixed and unique priority
  - Each task is periodic, with constant and known period
  - Tasks are independent, i.e. they don't talk to each other
  - Each task is completed before it is called again
    - Feasible priority assignment and scheduling
  - Each task has a constant and known runtime
- Then RMS is optimal
  - If RMS assignment not feasible, then system not schedulable

Rate monotonic

Process	Period	Priority
A	25 ms	5
B	50 ms	3
C	12 ms	6
D	100 ms	1
E	40 ms	4
F	75 ms	2

39

CS122A: Embedded System Design, Fall02

### Rate Monotonic Scheduling

- RMS also give least upper bound on processor usage
  - Given n tasks, if the processor usage is less than  $n(2^{1/n} - 1)$ 
    - The set of task is schedulable
- E.g. If each task run 1 ms
  - In 600ms window, the processor usage is:
    - $24 + 12 + 50 + 6 + 15 + 8 = 115$ , usage is  $115/600 = 0.1917$
    - $6(2^{1/6} - 1) = 0.7348$
    - RMS schedule is valid
- E.g. if each task run 5 ms, usage will be  $575/600 = .9585$ 
  - Unschedulable
    - Under the given assumption

Process	Period	Priority
A	25 ms	5
B	50 ms	3
C	12 ms	6
D	100 ms	1
E	40 ms	4
F	75 ms	2

40

CS122A: Embedded System Design, Fall02

### Earliest Deadline First

- At any given time, task with the earliest deadline goes
  - Obviously, very high overhead
    - Have to keep checking deadline
- Dynamic Priority
- Feasible if processor usage < 1
- For task running 5ms (usage .9585)
  - Assume deadline=period

Process	Period	Priority
A	25 ms	5
B	50 ms	3
C	12 ms	6
D	100 ms	1
E	40 ms	4
F	75 ms	2

41

CS122A: Embedded System Design, Fall02

### Dataflow model

- Derivative of concurrent process model
- Nodes represent transformations
  - May execute concurrently
- Edges represent flow of tokens (data)
- Node may fire
  - When all of node's inputs have at least one token
- When node fires
  - It consumes input tokens and generates output token
- Nodes may fire simultaneously
- Commercial tools available
  - Graphical capture of dataflow model
  - Automatic synthesis
    - Translation to concurrent process model
    - Each node becomes a process

$$Z = (A + B) * (C - D)$$

Nodes with arithmetic transformations

Nodes with more complex transformations 42

42

CS122A: Embedded System Design, Fall02

### Synchronous dataflow

---

- Data flows at fixed rate with DSP
- Multiple tokens consumed/produced per firing
- SDF model takes advantage of
  - Each edge labeled with number of tokens consumed/produced each firing
  - Can statically schedule nodes, so can easily use sequential program model
    - Don't need real-time operating system and its overhead
- Algorithms developed for getting "single-appearance" schedules
  - Only one statement needed to call each node's associated procedure
    - Allows procedure inlining without code explosion, reducing overhead

**Synchronous dataflow**

43

CS122A: Embedded System Design, Fall02

### Single Appearance Schedules

---

- ABCBCCC
  - Not SAS, require buffer 2+3=5
- A(2B)(4C)
  - SAS, but require buffer 2+4=6 (same as ABBCCCC)
- A(2 B(2 C))
  - SAS, and require buffer 2+2=4 (same as ABCCBCC)

```

run A;
For (l=0; l<2; l++){
  run B;
  for(j=0;j<2;j++){
    run C;
  }
}
    
```

44

CS122A: Embedded System Design, Fall02