

Administrative matter

- Homework #4
 - Due Thursday 10/24, 6:10PM
 - Update/clarification posted yesterday

- Midterm
 - Tuesday 10/29, in class
 - Covers
 - 85% ESD: chapter 1-8 (with emphasis on later chapters)
 - 15%: Labs 1-6 (with emphasis on later labs)



State Machine and Concurrent Process Models

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



Introduction

- Describing embedded system's behavior can be hard
 - Complexity increasing with increasing IC capacity
 - Past: washing machines, small games, etc. (100's lines of code)
 - Today: TV set-top boxes, Cell phone, etc. (1000's lines of code)
 - Desired behavior often not fully understood in beginning
- Many implementation bugs due to description error



Describing the processing behavior

- Sequential program computation model
 - Assembly languages
 - C, or other higher level languages
 - More powerful constructs
- But there are more advance computational models
 - Handle more complex systems
 - Easier to describe system at a higher level
 - Flow, parallel computation, priority, scheduling...
 - Detect specification error early, not waituntil prototyping stage
- English?
 - Powerful language
 - But not formal nor precise
 - Cfr, legal documents



Computation model

- Assists in understanding and designing behavior
- Provides
 - Set of simpler objects
 - Simpler rules for composing those objects
 - Simpler execution semantics of composed objects
- Models commonly used for embedded systems:
 - Sequential program model
 - Set of statements
 - Rules for putting statements one after another
 - Semantics stating how statements executed one at a time
 - ...



Common models for embedded systems

- Sequential program model
 - Statement, and rules for executing statements one after another
- Communicating process model
 - Describe multiple sequential programs running concurrently
- State machine model
 - Monitoring control inputs and reacting by setting control outputs
- Dataflow model
 - Transforming streams of input data to streams of output data
- Object-oriented model
 - Breaking complex software into simpler, well-defined pieces
- Combination of models may be used for one design



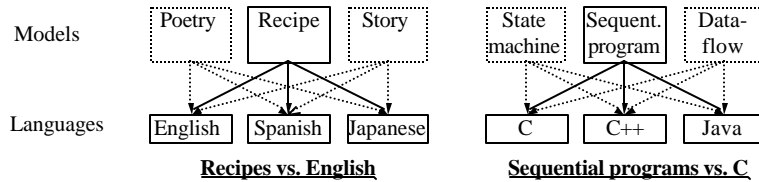
Models vs. Languages

- Common point of confusion
- Computation models describe system behavior
 - Conceptual notion
 - E.g., recipe, poetry, sequential program, state machine
- Languages capture models
 - Concrete form
 - E.g., English, Spanish, C, Stateflow
- Variety of languages can capture one model
 - E.g., C, C++, Java
 - sequential program model
- One language can capture variety of models
 - E.g., C++
 - sequential program model, object-oriented model, state machine model
- Certain languages better suited for certain models



Models vs. Languages

- Fettuccine Alfredo
 - In Recipe form with English
 - In Story form with Japanese
- Elevator controller
 - In Sequential Program with C
 - In Dataflow with C++?



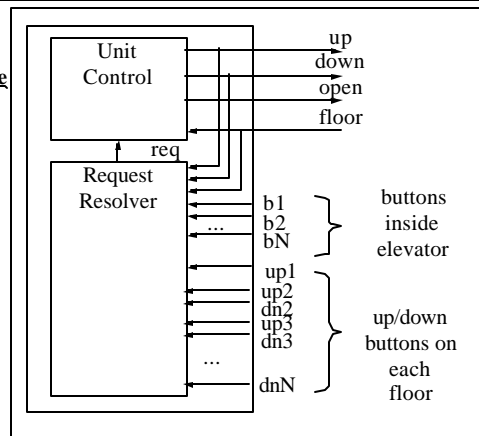
Elevator controller example

“Move the elevator either up or down to reach the requested floor. Once at the requested floor, open the door for at least 10 seconds, and keep it open until the requested floor changes. Ensure the door is never open while moving. Don’t change directions unless there are no higher requests when moving up or no lower requests when moving down...”

English description
(partial)

System interface

- Request Resolver resolves various floor requests into single requested floor
- Unit Control moves elevator to this requested floor



Introductory example

Sequential program model with concurrency construct

```

Inputs: int floor; bit b1..bN; up1..upN-1; dn2..dnN;
Outputs: bit up, down, open;
Global variables: int req;

void UnitControl()                void RequestResolver()
{                                  {
  up = down = 0; open = 1;         while (1)
  while (1) {                      ...
    while (req == floor);         req = ...
    open = 0;                     ...
    if (req > floor) { up = 1; }   }
    else {down = 1;}             void main()
    while (req != floor);         {
    up = down = 0;               Call concurrently:
    open = 1;                   UnitControl() and
    delay(10);                   RequestResolver()
  }                               }
}

```

11

CS122A: Embedded System Design, Fall 02

Outline

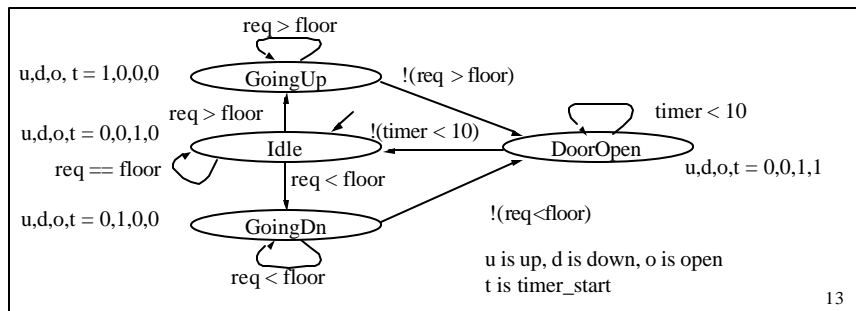
- Models v. Languages
- State Machine Model
 - FSM/FSMD
 - HCFSM and Statecharts Language
 - Program-State Machine (PSM) Model
- Concurrent Process Model
 - Communication
 - Synchronization
 - Implementation
- Dataflow Model
- Real-Time Systems

12

CS122A: Embedded System Design, Fall 02

Finite-state machine (FSM) model

- Describe system as:
 - Set of possible states
 - E.g., Idle, GoingUp, GoingDn, DoorOpen
 - Possible transitions from one state to another based on input
 - E.g., $!(req > floor)$, $!(timer < 10)$, etc.
 - Actions that occur in state or when transitioning
 - E.g., In GoingUp, $u,d,o,t = 1,0,0,0$ *UnitControl process using a state machine*



CS122A: Embedded System Design, Fall 02

Formal definition

- An FSM is a 6-tuple $F \langle S, I, O, F, H, s_0 \rangle$
 - S is a set of all states $\{s_0, s_1, \dots, s_l\}$
 - I is a set of inputs $\{i_0, i_1, \dots, i_m\}$
 - O is a set of outputs $\{o_0, o_1, \dots, o_n\}$
 - F is a next-state function $(S \times I \rightarrow S)$
 - H is an output function $(S \rightarrow O)$
 - s_0 is an initial state
- Moore-type
 - Associates outputs with states (as given above, H maps $S \rightarrow O$)
- Mealy-type
 - Associates outputs with transitions (H maps $S \times I \rightarrow O$)
- Shorthand notations to simplify descriptions
 - Implicitly assign 0 to all unassigned outputs in a state
 - Implicitly AND every transition condition with clock edge
 - If FSM is synchronous

CS122A: Embedded System Design, Fall 02

Finite-state machine with datapath model (FSMD)

- FSMD extends FSM to allow for data type and variable storage
 - FSMs use only Boolean data and state storage
- 7-tuple $\langle S, I, O, V, F, H, s_0 \rangle$
 - S is a set of states $\{s_0, s_1, \dots, s_l\}$
 - I is a set of inputs $\{i_0, i_1, \dots, i_m\}$
 - O is a set of outputs $\{o_0, o_1, \dots, o_n\}$
 - V is a set of variables $\{v_0, v_1, \dots, v_n\}$
 - F is a next-state function $(S \times I \times V \rightarrow S)$
 - H is an action function $(S \rightarrow O + V)$
 - s_0 is an initial state
- I, O, V may represent complex data types (i.e., integers, floating point, etc.)
- F, H may include arithmetic operations
- H is an action function not just an output function
 - Describes variable updates as well as outputs
- Complete system state now consists of current state, s_i , variable values

15

CS122A: Embedded System Design, Fall 02

Describing a system as a state machine

- Steps to describe system's behavior as FSMD:
 - List all possible states
 - Descriptive name for each
 - Declare all variables
 - For each state, list possible transitions, with conditions, to other states
 - For each state and/or transition, list associated actions
- Ensure exclusive and complete state exiting transition conditions
- No two conditions can be true at same time
 - Otherwise nondeterministic state machine
- One condition must be true at any given time
 - Implicit transitions should be avoided when first learning

16

CS122A: Embedded System Design, Fall 02

State machine vs. sequential program model

- Different thought process used with each model
- State machine:
 - Encourages designer to think of
 - all possible states and transitions based on all possible input conditions
- Sequential program model:
 - Transform data through series of instructions
 - Instructions may be iterated and conditionally executed
- State machine description excels in some cases
 - More natural means of computing in those cases
 - Not due to graphical representation (state diagram)
 - Would still have same benefits if textual language used (i.e., state table)
 - (Sequential program model could use graphical flowchart)



CS122A: Embedded System Design, Fall 02

17

Capturing state machines in sequential programming language

- Still, most popular development tools use SP languages
 - C, C++, Java, Ada, VHDL, Verilog, etc.
 - Development tools are complex and expensive, and not easy to replace
 - Must protect investment
- Two approaches to capturing SM model with SP languages
- Front-end tool approach
 - Additional tool installed to support state machine language
 - Graphical and/or textual state machine languages
 - May support graphical simulation
 - Automatically generate code in SP language (input to development tool)
 - Drawback: must support additional tool (costs, upgrades, training, etc.)
- Language subset approach
 - Most common approach



CS122A: Embedded System Design, Fall 02

18

Language subset approach

- Capturing SM constructs in equivalent set of SP constructs
 - Follow strict set of rules (template)
- Used with software (C) and hardware languages (VHDL)
- E.g., capturing UnitControl state machine in C
 - Enumerate all states (#define)
 - Declare state variable initialized to initial state (IDLE)
 - Single switch statement branches to current state's case
 - Each case may have actions
 - up, down, open, timer_start
 - Each case checks transition conditions to determine next state
 - if(...) {state = ...;}

19

CS122A: Embedded System Design, Fall 02

Language subset approach

```

#define IDLE0
#define GOINGUP1
#define GOINGDN2
#define DOOROPEN3
void UnitControl() {
    int state = IDLE;
    while (1) {
        switch (state) {
            IDLE: up=0; down=0; open=1; timer_start=0;
                if (req==floor) {state = IDLE;}
                if (req > floor) {state = GOINGUP;}
                if (req < floor) {state = GOINGDN;}
                break;
            GOINGUP: up=1; down=0; open=0; timer_start=0;
                if (req > floor) {state = GOINGUP;}
                if (!(req>floor)) {state = DOOROPEN;}
                break;
            GOINGDN: up=1; down=0; open=0; timer_start=0;
                if (req < floor) {state = GOINGDN;}
                if (!(req<floor)) {state = DOOROPEN;}
                break;
            DOOROPEN: up=0; down=0; open=1; timer_start=1;
                if (timer < 10) {state = DOOROPEN;}
                if (!(timer<10)){state = IDLE;}
                break;
        }
    }
}

```

UnitControl State machine in sequential programming language

20

CS122A: Embedded System Design, Fall 02

General template

```

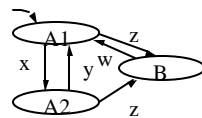
#define S0      0
#define S1      1
...
#define SN      N
void StateMachine() {
    int state = S0; // or whatever is the initial state.
    while (1) {
        switch (state) {
            S0:
                // Insert S0's actions here & Insert transitions Ti leaving S0:
                if( T0's condition is true ) {state = T0's next state; /*actions*/}
                if( T1's condition is true ) {state = T1's next state; /*actions*/}
                ...
                if( Tm's condition is true ) {state = Tm's next state; /*actions*/}
                break;
            S1:
                // Insert S1's actions here
                // Insert transitions Ti leaving S1
                break;
            ...
            SN:
                // Insert SN's actions here
                // Insert transitions Ti leaving SN
                break;
        }
    }
}

```

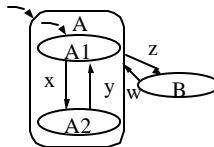
HCFSM and the Statecharts language

- Hierarchical/concurrent state machine model (HCFSM)
 - Extension to state machine model to support hierarchy and concurrency
 - States can be decomposed into another state machine
 - Conversely, several states grouped as one state
 - With hierarchy has identical functionality as Without hierarchy, but has one less transition (z)
 - OR-decomposition
 - States can execute concurrently
 - AND-decomposition

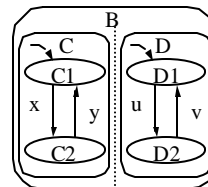
Without hierarchy



With hierarchy

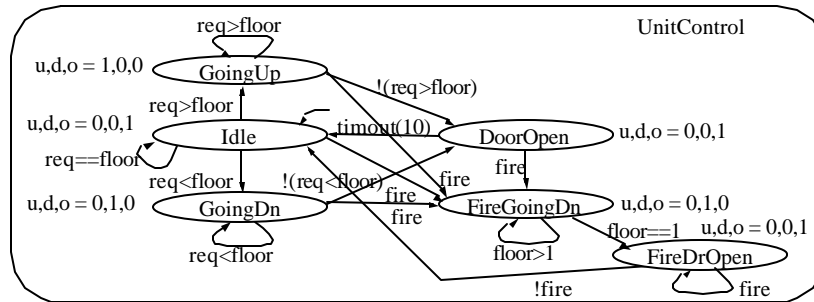


Concurrency

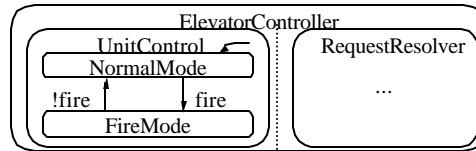


UnitControl with FireMode

- FireMode
 - When *fire*, immediately move elevator to 1st floor and open door



Without hierarchy

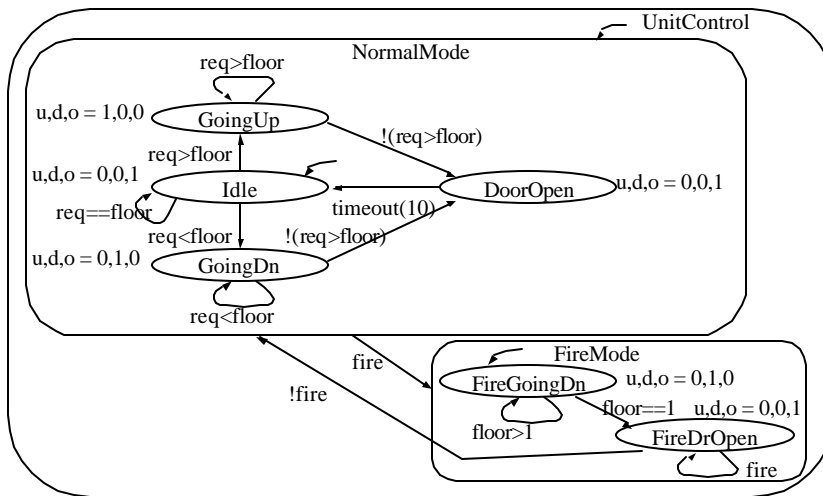


With concurrent RequestResolver

23

UnitControl with FireMode

With hierarchy



24

Program-state machine model (PSM)

- Merges HCFSM with sequential program model
- Allows sequential program code to define state's action
 - Could still define with state machine also
- Stricter hierarchy than HCFSM used in Statecharts
 - PSM requires transition between sibling states only
 - PSM requires entry to subroutine from one point only
- Includes SP notion of program-state completing
 - Reaching end of code means program-state is complete
- PSM has 2 types of transitions
 - Transition-immediately (TI)
 - taken regardless of source program-state
 - Transition-on-completion (TOC)
 - taken only if condition is true AND source program-state is complete
 - SpecCharts: extension of VHDL to capture PSM model
 - SpecC: extension of C to capture PSM model

25

CS122A: Embedded System Design, Fall 02

PSM description of ElevatorController

- NormalMode and FireMode are described as SP
- Black square
 - Indicates !fire is a TOC transition
 - Transition only after FireMode completed

ElevatorController

UnitControl

NormalMode

```

up = down = 0; open = 1;
while (1) {
  while (req == floor);
  open = 0;
  if (req > floor) { up = 1;}
  else {down = 1;}
  while (req != floor);
  open = 1;
  delay(10);}
                    
```

RequestResolver

```

...
req = ...
...
                    
```

FireMode

```

up = 0; down = 1; open = 0;
while (floor > 1);
up = 0; down = 0; open = 1;
                    
```

!fire ↑
fire ↓

int req;

26

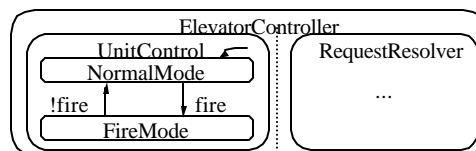
CS122A: Embedded System Design, Fall 02

Role of appropriate model and language

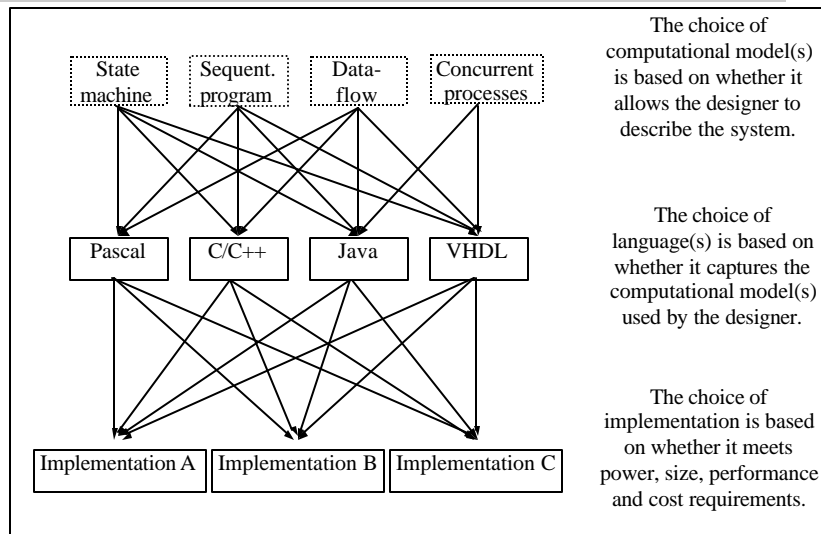
- Finding appropriate model to capture design is important
 - Model shapes the way we think of the system
 - Originally thought of sequence of actions, wrote sequential program
 - First wait for requested floor to differ from target floor
 - Then, we close the door
 - Then, we move up or down to the desired floor
 - Then, we open the door
 - Then, we repeat this sequence
 - To create SM, we thought in terms of states and transitions among states
 - When system must react to changing inputs, SM might be best model
 - HCFSM described FireMode easily, clearly

Role of appropriate model and language

- Language should capture model easily
 - Ideally should have features that directly capture constructs of model
 - FireMode would be very complex in sequential program
 - Checks inserted throughout code
 - While other factors may force choice of different model
 - Structured techniques can be used to adapt from one to another
 - E.g., Template for state machine capture in sequential program language



Implementation



29

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

30

CS122A: Embedded System Design, Fall 02

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);
  }
}
    
```

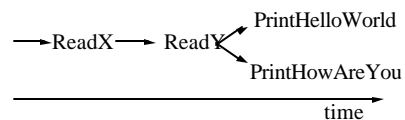
Simple concurrent process example

31

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

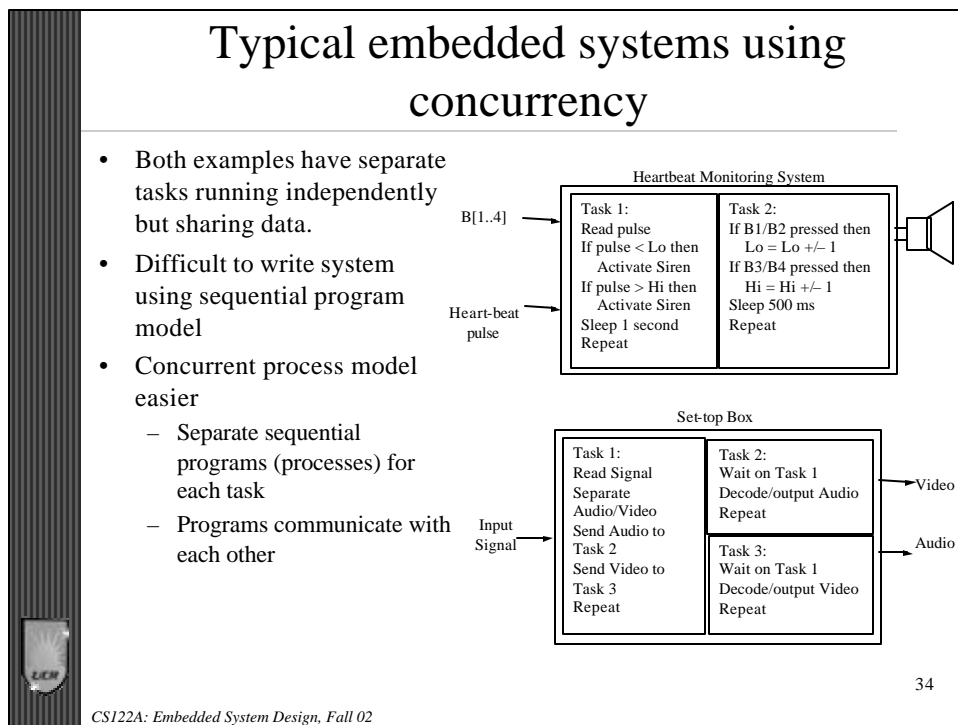
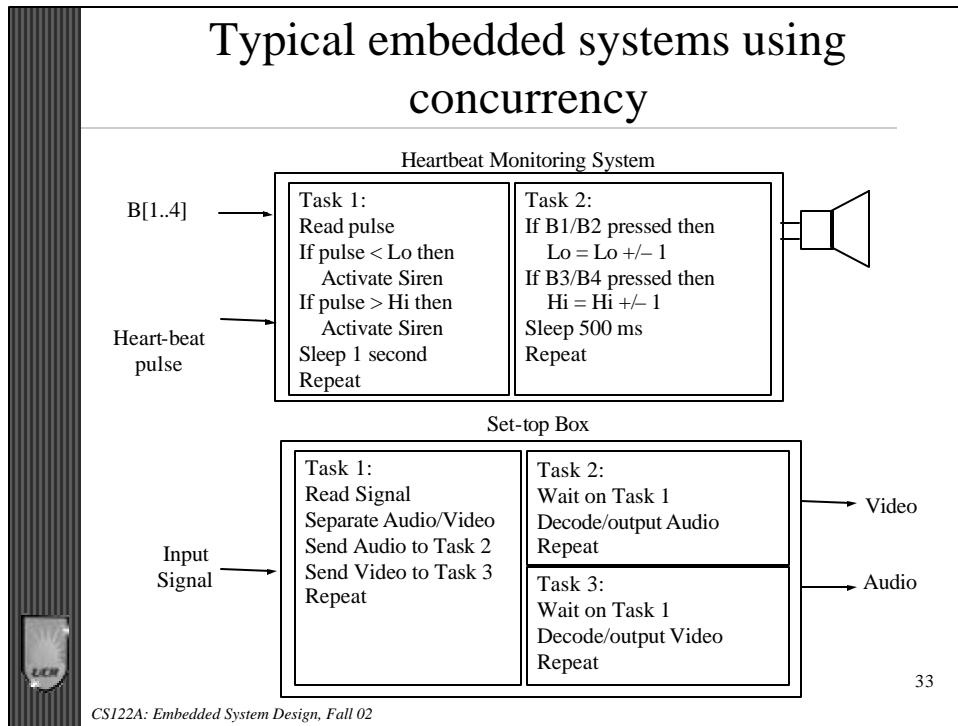
```

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

32



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

35

CS122A: Embedded System Design, Fall 02

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

36

CS122A: Embedded System Design, Fall 02

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



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



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 */
    }
}
```

39

CS122A: Embedded System Design, Fall 02

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:
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

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 use 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

42

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: }

```

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: }

```

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



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



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

49

CS122A: Embedded System Design, Fall 02

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;
06: void processA() {
07:     int i;
08:     while( 1 ) {
09:         produce(&data);
10:         cs_mutex.lock();
11:         if( count == N ) buffer_empty.wait(cs_mutex);
13:         buffer[count] = data;
14:
15:         count = count + 1;
16:         cs_mutex.unlock();
17:         buffer_full.signal();
18:     }
19: }
20: void processB() {
21:     int i;
22:     while( 1 ) {
23:         cs_mutex.lock();
24:         if( count == 0 ) buffer_full.wait(cs_mutex);
26:         data = buffer[count-1];
27:
28:         count = count - 1;
29:         cs_mutex.unlock();
30:         buffer_empty.signal();
31:         consume(&data);
32:     }
33: }

```

50

CS122A: Embedded System Design, Fall 02

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;
06: void processA() {
07:     int i;
08:     while( 1 ) {
09:         produce(&data);
10:         cs_mutex.lock();
11:         if( count == N ) buffer_empty.wait(cs_mutex);
13:         buffer[count] = data;
14:
15:         count = count + 1;
16:         cs_mutex.unlock();
17:         buffer_full.signal();
18:     }
19: }
20: void processB() {
21:     int i;
22:     while( 1 ) {
23:         cs_mutex.lock();
24:         if( count == 0 ) buffer_full.wait(cs_mutex);
26:         data = buffer[count-1];
27:
28:         count = count - 1;
29:         cs_mutex.unlock();
30:         buffer_empty.signal();
31:         consume(&data);
32:     }
33: }

```

51

CS122A: Embedded System Design, Fall 0

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;
06: void processA() {
07:     int i;
08:     while( 1 ) {
09:         produce(&data);
10:         cs_mutex.lock();
11:         if( count == N ) buffer_empty.wait(cs_mutex);
13:         buffer[count] = data;
14:
15:         count = count + 1;
16:         cs_mutex.unlock();
17:         buffer_full.signal();
18:     }
19: }
20: void processB() {
21:     int i;
22:     while( 1 ) {
23:         cs_mutex.lock();
24:         if( count == 0 ) buffer_full.wait(cs_mutex);
26:         data = buffer[count-1];
27:
28:         count = count - 1;
29:         cs_mutex.unlock();
30:         buffer_empty.signal();
31:         consume(&data);
32:     }
33: }

```

52

CS122A: Embedded System Design, Fall 0

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

53

CS122A: Embedded System Design, Fall 02

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:       count = count + 1;
13:       buffer_full.signal();
14:     }
15:   }
16:   void processB() {
17:     int i;
18:     while( 1 ) {
19:       if( count == 0 ) buffer_full.wait();
20:       data = buffer[count-1];
21:
22:       count = count - 1;
23:       buffer_empty.signal();
24:       consume(&data);
25:     }
26:   }
27: }
28:
29:

```

54

CS122A: Embedded System Design, Fall 02

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:     }
12:     count = count + 1;
13:     buffer_full.signal();
14:   }
15:   void processB() {
16:     int i;
17:     while( 1 ) {
18:       if( count == 0 ) buffer_full.wait();
19:       data = buffer[count-1];
20:     }
21:     count = count - 1;
22:     buffer_empty.signal();
23:     consume(&data);
24:   }
25: }
26:
27:
28:
29:

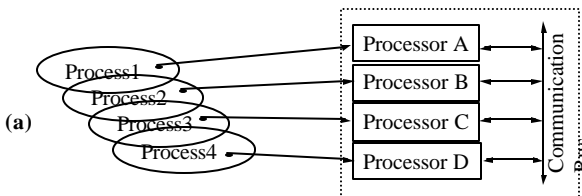
```

55

CS122A: Embedded System Design, Fall 02

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



56

CS122A: Embedded System Design, Fall 02

Concurrent process model: implementation

- (b) One general-purpose processor running all processes
 - Most processes don't use 100% of processor time ^(a)
 - 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

CS122A: Embedded System Design, Fall 02

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
 - }

58

CS122A: Embedded System Design, Fall 02

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 scheduling
 - Less overhead (no operating system)
 - More complex/harder to maintain

59

CS122A: Embedded System Design, Fall 02

Processes vs. threads

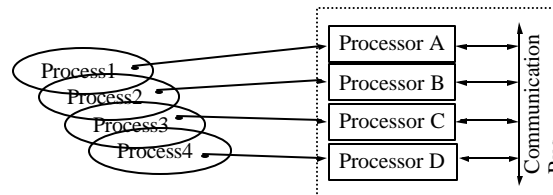
- Different meanings than operating system terminology
- Regular processes
 - Heavyweight process
 - Own virtual address space (stack, data, code)
 - System resources (e.g., open files)
- Threads
 - Lightweight process
 - Subprocess within process
 - Only program counter, stack, and registers
 - Shares 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

60

CS122A: Embedded System Design, Fall 02

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

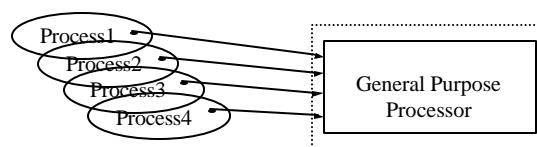


61

CS122A: Embedded System Design, Fall 02

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



62

CS122A: Embedded System Design, Fall 02

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

63

CS122A: Embedded System Design, Fall 02

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

64

CS122A: Embedded System Design, Fall 02

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

65

CS122A: Embedded System Design, Fall 02

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

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

EDF, priority for now

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

66

CS122A: Embedded System Design, Fall 02

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

67

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

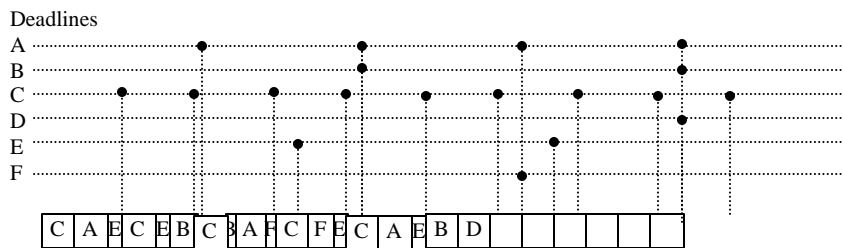
Process	Period	Priority
A	25 ms	5
B	50 ms	3
C	12 ms	6
D	100 ms	1
E	40 ms	4
F	75 ms	2

68

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



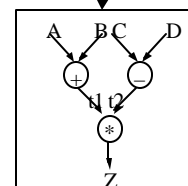
69

CS122A: Embedded System Design, Fall 02

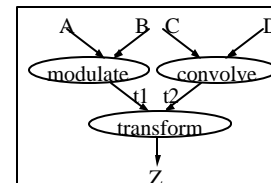
Dataflow model

- 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



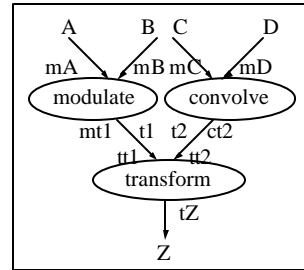
Nodes with more complex transformations

70

CS122A: Embedded System Design, Fall 02

Synchronous dataflow

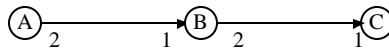
- Data flows at fixed rate with DSP
- Multiple tokens consumed/produced per firing
- SDF model takes advantage of
 - Each edge labeled with number of tokens consumed/produced each firing
 - Can statically schedule nodes, so can easily use sequential program model
 - Don't need real-time operating system 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

71

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

72

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



Real-time systems

- 2 or more concurrent processes with stringent time constraints
 - E.g., set-top boxes must decode 20 frames/sec to look right
 - Other examples with stringent time constraints are:
 - digital cell phones
 - navigation and process control systems
 - assembly line monitoring systems
 - multimedia and networking systems
 - etc.
 - Communication/synchronization between processes is critical
 - Therefore, concurrent process model best suited for describing these systems
- Real-Time Operating Systems (RTOS)
 - Provide scheduling and interfacing for Real-time embedded systems



Real-time operating systems (RTOS)

- Windows CE
 - Built specifically for embedded systems and appliance market
 - Scalable real-time 32-bit platform
 - Supports Windows API
 - Used for systems designed to interface with Internet
 - Preemptive priority scheduling with 256 priority levels per process
 - Kernel is 400 Kbytes
- QNX
 - RT microkernel w/ processes providing POSIX/UNIX compatibility
 - Microkernels typically support only the most basic services
 - Optional resource managers allow scalability to huge MP systems
 - Preemptive process using FIFO, adaptive, or priority-driven scheduling
 - 32 priority levels per process
 - Microkernel < 10 Kbytes

75