CprE 588
Embedded Computer Systems

Prof. Joseph Zambreno
Department of Electrical and Computer Engineering
Iowa State University

Lecture #3 – Models of Computation
Introduction

- Describing embedded system’s processing behavior
  - Can be extremely difficult
    - Complexity increasing with increasing IC capacity
      - Past: washing machines, small games, etc.
        - Hundreds of lines of code
      - Today: TV set-top boxes, Cell phone, etc.
        - Hundreds of thousands of lines of code
    - Desired behavior often not fully understood in beginning
      - Many implementation bugs due to description mistakes/omissions
  - English (or other natural language) common starting point
    - Precise description difficult to impossible
    - Example: Motor Vehicle Code – thousands of pages long...

An Attempt at Precision

• California Vehicle Code
  • Right-of-way of crosswalks
    • 21950. (a) The driver of a vehicle shall yield the right-of-way to a pedestrian crossing the roadway within any marked crosswalk or within any unmarked crosswalk at an intersection, except as otherwise provided in this chapter.
    • (b) The provisions of this section shall not relieve a pedestrian from the duty of using due care for his or her safety. No pedestrian shall suddenly leave a curb or other place of safety and walk or run into the path of a vehicle which is so close as to constitute an immediate hazard. No pedestrian shall unnecessarily stop or delay traffic while in a marked or unmarked crosswalk.
    • (c) The provisions of subdivision (b) shall not relieve a driver of a vehicle from the duty of exercising due care for the safety of any pedestrian within any marked crosswalk or within any unmarked crosswalk at an intersection.
  • All that just for crossing the street (and there’s much more)!
Models and Languages

- How can we (precisely) capture behavior?
  - We may think of languages (C, C++), but computation model is the key
- Common computation models:
  - Sequential program model
    - Statements, rules for composing statements, semantics for executing them
  - Communicating process model
    - Multiple sequential programs running concurrently
  - State machine model
    - For control dominated systems, monitors control inputs, sets control outputs
  - Dataflow model
    - For data dominated systems, transforms input data streams into output streams
  - Object-oriented model
    - For breaking complex software into simpler, well-defined pieces
Models vs. Languages

- Computation models describe system behavior
  - Conceptual notion, e.g., recipe, sequential program
- Languages capture models
  - Concrete form, e.g., English, C
- Variety of languages can capture one model
  - E.g., sequential program model → C, C++, Java
- One language can capture variety of models
  - E.g., C++ → sequential program model, object-oriented model, state machine model
- Certain languages better at capturing certain computation models

### Diagram

**Models**
- Poetry
- Recipe
- Story

**Languages**
- English
- Spanish
- Japanese
- C
- C++
- Java

**Recipes vs. English**

**Sequential programs vs. C**
Text vs. Graphics

• Models versus languages not to be confused with text versus graphics
• Text and graphics are just two types of languages
  • Text: letters, numbers
  • Graphics: circles, arrows (plus some letters, numbers)

X = 1;
Y = X + 1;

\[ X = 1 \]
\[ Y = X + 1 \]
An Introductory Example

- Simple elevator controller
  - *Request Resolver* resolves various floor requests into single requested floor
  - *Unit Control* moves elevator to this requested floor
- Try capturing in C...

Partial English description:

“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…”

System interface:
An Introductory Example (cont.)

Sequential program model

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

void RequestResolver()
{
    while (1)
    {
        req = ...
    }
}

void main()
{
    Call concurrently:
    UnitControl() and RequestResolver()
}

Partial English description

“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…”

You might have come up with something having even more if statements.
An Introductory Example (cont.)

- Trying to capture this behavior as sequential program is a bit awkward
- Instead, we might consider an FSM model, describing the system as:
  - Possible states
    - E.g., Idle, GoingUp, GoingDn, DoorOpen
  - Possible transitions from one state to another based on input
    - E.g., req > floor
  - Actions that occur in each state
    - E.g., In the GoingUp state, u,d,o,t = 1,0,0,0 (up = 1, down, open, and timer_start = 0)
- Try it...
Finite-State Machine (FSM) Model

*UnitControl* process using a state machine

```
Idle

- req > floor
- req == floor
- req < floor

GoingUp

- !req > floor
- timer < 10

DoorOpen

- timer < 10
- !req < floor
- !req > floor

GoingDn

- req < floor
- !req < floor

u, d, o, t = 1, 0, 0, 0
- u, d, o, t = 0, 0, 1, 0
- u, d, o, t = 0, 1, 0, 0
- u, d, o, t = 0, 0, 1, 1

u is up, d is down, o is open
- t is timer_start
```
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 \( \{s_0, s_1, \ldots, s_l\} \)
  - \( I \) is a set of inputs \( \{i_0, i_1, \ldots, i_m\} \)
  - \( O \) is a set of outputs \( \{o_0, o_1, \ldots, o_n\} \)
  - \( F \) 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 (FSM is synchronous)
FSM with Datapath (FSMD)

- FSMD extends FSM: complex data types and variables for storing data
  - FSMs use only Boolean data types and operations, no variables
- FSMD: 7-tuple \(<S, I, O, V, F, H, s_0>\)
  - \(S\) is a set of states \(\{s_0, s_1, \ldots, s_l\}\)
  - \(I\) is a set of inputs \(\{i_0, i_1, \ldots, i_m\}\)
  - \(O\) is a set of outputs \(\{o_0, o_1, \ldots, o_n\}\)
  - \(V\) is a set of variables \(\{v_0, v_1, \ldots, v_n\}\)
  - \(F\) 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, \(s_i\), and values of all variables
Describing a System using an FSM

1. List all possible states
2. Declare all variables (none in this example)
3. For each state, list possible transitions, with conditions, to other states
4. For each state and/or transition, list associated actions
5. For each state, ensure exclusive and complete exiting transition conditions
   - No two exiting conditions can be true at same time
     - Otherwise nondeterministic state machine
   - One condition must be true at any given time
     - Reducing explicit transitions should be avoided when first learning

![Diagram of FSM with transitions and conditions](attachment:image.png)
FSM vs. Sequential Program Model

- Different thought process used with each model
- State machine:
  - Encourages designer to think of all possible states and transitions among states based on all possible input conditions
- Sequential program model:
  - Designed to transform data through series of instructions that may be iterated and conditionally executed
- State machine description excels in many 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)
    - Besides, sequential program model could use graphical representation (i.e., flowchart)
In-Class Design Example

- Try Capturing Other Behaviors with an FSM
  - Answering machine blinking light when there are messages
  - A simple telephone answering machine that answers after 4 rings when activated
  - A simple crosswalk traffic control light
Language-Based FSM Design

• Despite benefits of state machine model, most popular development tools use sequential programming language
  • C, C++, Java, Ada, VHDL, Verilog, etc.
  • Development tools are complex and expensive, therefore not easy to adapt or replace
    • Must protect investment
• Two approaches to capturing state machine model with sequential programming language
  • 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 sequential programming language that is input to main development tool
    • Drawback: must support additional tool (licensing costs, upgrades, training, etc.)
  • Language subset approach
    • Most common approach...
Language-Based FSM Design (cont.)

- Follow rules (template) for capturing state machine constructs in equivalent sequential language constructs.
- Used with software (e.g., C) and hardware languages (e.g., VHDL).
- 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 has actions
    - up, down, open, timer_start
  - Each case checks transition conditions to determine next state
    - if(...) {state = ...;}

```c
#define IDLE0
#define GOINGUP1
#define GOINGDN2
#define DOOROPEN3

void UnitControl() {
  int state = IDLE;
  while (1) {
    switch (state) {
      case 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;
      case GOINGUP:
        up=1; down=0; open=0; timer_start=0;
        if (req > floor) {state = GOINGUP;}
        if (!(req > floor)) {state = DOOROPEN;}
        break;
      case GOINGDN:
        up=1; down=0; open=0; timer_start=0;
        if (req < floor) {state = GOINGDN;}
        if (!(req < floor)) {state = DOOROPEN;}
        break;
      case 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
#define S0 0
#define S1 1
...
#define SN N

void StateMachine() {
    int state = S0; // or whatever is the initial state.
    while (1) {
        switch (state) {
            case S0:
                // Insert S0’s actions here & Insert transitions $T_i$ leaving $S0$:
                if( $T_0$’s condition is true ) {state = $T_0$’s next state; /*actions*/ }
                if( $T_1$’s condition is true ) {state = $T_1$’s next state; /*actions*/ }
                ... 
                if( $T_m$’s condition is true ) {state = $T_m$’s next state; /*actions*/ }
                break;
            case S1:
                // Insert S1’s actions here
                // Insert transitions $T_i$ leaving S1
                break;
            ...
            case SN:
                // Insert SN’s actions here
                // Insert transitions $T_i$ leaving SN
                break;
        }
    }
}

Jan 27-29, 2009  CprE 588 – Embedded Computer Systems  Lect-03.18
HCFSMs and Statecharts

- Hierarchical/concurrent state machine model (HCFSM)
  - Extension to state machine model to support hierarchy and concurrency
  - States can be decomposed into another state machine
    - With hierarchy has identical functionality as Without hierarchy, but has one less transition ($z$)
    - Known as OR-decomposition
  - States can execute concurrently
    - Known as AND-decomposition
- Statecharts
  - Graphical language to capture HCFSM
  - timeout: transition with time limit as condition
  - history: remember last substate OR-decomposed state $A$ was in before transitioning to another state $B$
    - Return to saved substate of $A$ when returning from $B$ instead of initial state
UnitControl with FireMode

- FireMode
  - When fire is true, move elevator to 1st floor and open door
  - w/o hierarchy: getting messy!
  - w/ hierarchy: simple!

Without hierarchy

With concurrent RequestResolver

With hierarchy
Program State Machine Model (PSM)

- Program-state’s actions can be FSM or sequential program
  - Designer can choose most appropriate
- Stricter hierarchy than HCFSM used in Statecharts
  - transition between sibling states only, single entry
- Program-state may “complete”
  - Reaches end of sequential program code, OR
  - FSM transition to special complete substate
- 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

```
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);
}
```

```
int req;
```

- **NormalMode** and **FireMode** described as sequential programs
- Black square originating within **FireMode** indicates !fire is a TOC transition
  - Transition from **FireMode** to **NormalMode** only after **FireMode** completed
Appropriate Model + Language

- Finding appropriate model to capture embedded system is an important step
  - 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 state machine, we thought in terms of states and transitions among states
      - When system must react to changing inputs, state machine might be best model
        - HCFSM described FireMode easily, clearly
  - 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
    - Other factors may force choice of different model
      - Structured techniques can be used instead
        - E.g., Template for state machine capture in sequential program language
Concurrent Process Model

- Describes functionality of system in terms of two or more concurrently executing subtasks
- Many systems easier to describe with concurrent process model because inherently multitasking
- E.g., simple example:
  - Read two numbers $X$ and $Y$
  - Display “Hello world.” every $X$ seconds
  - Display “How are you?” every $Y$ seconds
- More effort would be required with sequential program or state machine 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

```
Sample input and output
```

```
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)
...

Subroutine execution over time

Concurrent Process Example

```
Dataflow Model

- Derivative of concurrent process model
- Nodes represent transformations
  - May execute concurrently
- Edges represent flow of tokens (data) from one node to another
  - May or may not have token at any given time
- When all of node’s input edges have at least one token, node may fire
- When node fires, it consumes input tokens processes transformation and generates output token
- Nodes may fire simultaneously
- Several commercial tools support graphical languages for capture of dataflow model
  - Can automatically translate to concurrent process model for implementation
  - Each node becomes a process

\[ Z = (A + B) \times (C - D) \]
Concurrent Processes

- Consider two examples having 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

Heartbeat Monitoring System

B[1..4] 
Heart-beat pulse 

Task 1:
Read pulse
If pulse < Lo then
Activate Siren
If pulse > Hi then
Activate Siren
Sleep 1 second
Repeat

Task 2:
If B1/B2 pressed then
Lo = Lo +/- 1
If B3/B4 pressed then
Hi = Hi +/- 1
Sleep 500 ms
Repeat

Set-top Box

Task 1:
Read Signal
Separate Audio/Video
Send Audio to Task 2
Send Video to Task 3
Repeat

Task 2:
Wait on Task 1
Decode/output Audio
Repeat

Task 3:
Wait on Task 1
Decode/output Video
Repeat
Processes

• A sequential program, typically an infinite loop
  • Executes concurrently with other processes
  • We are about to enter the world of “concurrent programming”

• Basic operations on processes
  • Create and terminate
    • Create is like a procedure call but caller doesn’t wait
      • Created process can itself create new processes
    • Terminate kills a process, destroying all data
      • In HelloWorld/HowAreYou example, we only created processes
  • Suspend and resume
    • Suspend puts a process on hold, saving state for later execution
    • Resume starts the process again where it left off
  • Join
    • A process suspends until a particular child process finishes execution
Inter-Process Communication

- Processes need to communicate data and signals to solve their computation problem
  - Processes that don’t communicate are just independent programs solving separate problems
- Basic example: producer/consumer
  - Process A produces data items, Process B consumes them
  - E.g., A decodes video packets, B displays decoded packets on a screen
- How do we achieve this communication?
  - Two basic methods
    - Shared memory
    - Message passing

```c
int encoded_packets;

void processA() {
    // Decode packet
    // Communicate packet to B
}

void processB() {
    // Get packet from A
    // Display packet
}
```
Shared Memory

- Processes read and write shared variables
  - No time overhead, easy to implement
  - But, hard to use – mistakes are common
- Example: Producer/consumer with a mistake
  - 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 the following execution sequence occurs. Say “count” is 3.
    - 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

```c
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[i] = data;
09:     i = (i + 1) % N;
10:     count = count + 1;
11:   }
12: }
13: void processB() {
14:   int i;
15:   while( 1 ) {
16:     while( count == 0 );/*loop*/
17:     data = buffer[i];
18:     i = (i + 1) % N;
19:     count = count - 1;
20:     consume(&data);
21:   }
22: }
23: void main() {
24:   create_process(processA);
25:   create_process(processB);
26: }
```
Message Passing

- Data explicitly sent from one process to another
  - Sending process performs special operation, *send*
  - Receiving process must perform special operation, *receive*, to receive the data
  - Both operations must explicitly specify which process it is sending to or receiving from
  - Receive is blocking, send may or may not be blocking
- Safer model, but less flexible
Shared Memory (cont.)

- 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
  - 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
      - 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 unlock operation performed by acquiring process when it exits critical section
      - These processes will then be placed in runnable state and will compete for lock again
Shared Memory Producer-Consumer

- The primitive *mutex* is used to ensure critical sections are executed in mutual exclusion of each other.
- Following the same execution sequence as before:
  - A/B execute lock operation on *count_mutex*
  - Either A or B will acquire lock
    - Say B acquires it
    - A will be put in blocked state
  - 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 operation
    - A is placed in runnable state again
  - 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 now has correct value of 3

```c
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[i] = data;
10:     i = (i + 1) % N;
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[i];
21:     i = (i + 1) % N;
22:     count_mutex.lock();
23:     count = count - 1;
24:     count_mutex.unlock();
25:     consume(&data);
26:   }  
27: }
28: void main() {  
29:   create_process(processA);
30:   create_process(processB);
31: }
```
Avoiding Deadlock

- **Deadlock**: A condition where 2 or more processes are blocked waiting for the other to unlock critical sections of code
  - Both processes are then in blocked state
  - Cannot execute unlock operation so will wait forever
- Example code has 2 different critical sections of code that can be accessed simultaneously
  - 2 locks needed (mutex1, mutex2)
- Following execution sequence produces 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*
  - **DEADLOCK**!
- One deadlock elimination protocol requires locking of numbered mutexes in increasing order and two-phase locking (2PL)
  - Acquire locks in 1st phase only, release locks in 2nd phase

```c
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: }
```
Synchronizing Processes

- Sometimes concurrently running processes must synchronize their execution
  - When a process must wait for:
    - another process to compute some value
    - reach a known point in their execution
    - signal some condition
  - Recall producer-consumer problem
    - processA must wait if buffer is full
    - processB must wait if buffer is empty
    - This is called busy-waiting
      - Process executing loops instead of being blocked
      - CPU time wasted
  - More efficient methods
    - Join operation, and blocking send and receive discussed earlier
      - Both block the process so it doesn’t waste CPU time
    - Condition variables and monitors
Condition Variables

- Condition variable is an object that has 2 operations, signal and wait
- When process performs a wait on a condition variable, the process is blocked until another process performs a signal on the same condition variable
- 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
Condition Variables (cont.)

- 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
  - 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
  - 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

```c
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);
12:     buffer[i] = data;
13:     i = (i + 1) % N;
14:     count = count + 1;
15:     cs_mutex.unlock();
16:     buffer_full.signal();
17:   }
18: }
19: void processB() {
20:   int i;
21:   while( 1 ) {
22:     cs_mutex.lock();
23:     if( count == 0 ) buffer_full.wait(cs_mutex);
24:     data = buffer[i];
25:     i = (i + 1) % N;
26:     count = count - 1;
27:     cs_mutex.unlock();
28:     buffer_empty.signal();
29:     consume(&data);
30:   }
31: }
33: void main() {
35:   create_process(processA); create_process(processB);
37: }
```
Monitors

- Collection of data and methods or subroutines that operate on data similar to an object-oriented paradigm
- 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
Monitors (cont.)

- 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
  - 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[i] = data;
11:       i = (i + 1) % N;
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[i];
21:       i = (i + 1) % N;
22:       count = count - 1;
23:       buffer_empty.signal();
24:       consume(&data);
25:       buffer_full.signal();
26:     } 
27:   }
28: } /* end monitor */
29: void main() {
30:   create_process(processA);
31:   create_process(processB);  
32: } 
```
Implementation

- Mapping of system’s functionality onto hardware processors:
  - captured using computational model(s)
  - written in some language(s)
- Implementation choice independent from language(s) choice
- Implementation choice based on power, size, performance, timing and cost requirements
- Final implementation tested for feasibility
  - Also serves as blueprint/prototype for mass manufacturing of final product

The choice of computational model(s) is based on whether it allows the designer to describe the system.

The choice of language(s) is based on whether it captures the computational model(s) used by the designer.

The choice of implementation is based on whether it meets power, size, performance and cost requirements.
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
  - (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
Concurrent Process Implementation

- 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., simple Hello World concurrent program from before would look like:
    ```
    I = 1; T = 0;
    while (1) {
        Delay(I); T = T + 1;
        if X modulo T is 0 then call PrintHelloWorld
        if Y modulo T is 0 then call PrintHowAreYou
    }
    ```
- Can use multitasking operating system
  - Much more common
  - Operating system schedules processes, allocates storage, and interfaces to peripherals, etc.
  - Real-time operating system (RTOS) can guarantee execution rate constraints are met
  - Describe concurrent processes with languages having built-in processes (Java, Ada, etc.) or a sequential programming language with library support for concurrent processes (C, C++, etc. using POSIX threads for example)
- Can convert processes to sequential program with process scheduling right in code
  - Less overhead (no operating system)
  - More complex/harder to maintain
Processes v. Threads

- Different meanings when 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
Suspending, Resuming, and Joining

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

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

- Must meet timing requirements when multiple concurrent processes implemented on single general-purpose processor
  - Not true multitasking
- Scheduler
  - Special process that decides when and for how long each process is executed
  - Implemented as preemptive or nonpreemptive scheduler
- Preemptive
  - Determines how long a process executes before preempting to allow another process to execute
    - Time quantum: predetermined amount of execution time preemptive scheduler allows each process (may be 10 to 100s of milliseconds long)
  - Determines which process will be next to run
- Nonpreemptive
  - Only determines which process is next after current process finishes execution
Scheduling Priority

- Process with highest priority always selected first by scheduler
  - Typically determined statically during creation and dynamically during execution
- FIFO
  - Runnable processes added to end of FIFO as created or become runnable
  - Front process removed from FIFO when time quantum of current process is up or process is blocked
- Priority queue
  - Runnable processes again added as created or become runnable
  - Process with highest priority chosen when new process needed
  - If multiple processes with same highest priority value then selects from them using first-come first-served
  - Called priority scheduling when nonpreemptive
  - Called round-robin when preemptive
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 regardless of its period
    - 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

- **Rate monotonic scheduling**
  - Processes with shorter periods have higher priority
  - Typically used when execution deadline = period

- **Deadline monotonic scheduling**
  - Processes with shorter deadlines have higher priority
  - Typically used when execution deadline < period

---

### Rate monotonic

<table>
<thead>
<tr>
<th>Process</th>
<th>Period</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>25 ms</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td>50 ms</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>12 ms</td>
<td>6</td>
</tr>
<tr>
<td>D</td>
<td>100 ms</td>
<td>1</td>
</tr>
<tr>
<td>E</td>
<td>40 ms</td>
<td>4</td>
</tr>
<tr>
<td>F</td>
<td>75 ms</td>
<td>2</td>
</tr>
</tbody>
</table>

### Deadline monotonic

<table>
<thead>
<tr>
<th>Process</th>
<th>Deadline</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>17 ms</td>
<td>5</td>
</tr>
<tr>
<td>H</td>
<td>50 ms</td>
<td>2</td>
</tr>
<tr>
<td>I</td>
<td>32 ms</td>
<td>3</td>
</tr>
<tr>
<td>J</td>
<td>10 ms</td>
<td>6</td>
</tr>
<tr>
<td>K</td>
<td>140 ms</td>
<td>1</td>
</tr>
<tr>
<td>L</td>
<td>32 ms</td>
<td>4</td>
</tr>
</tbody>
</table>
Real-Time Systems

• Systems composed of 2 or more cooperating, concurrent processes with stringent execution time constraints
  • E.g., set-top boxes have separate processes that read or decode video and/or sound concurrently and must decode 20 frames/sec for output to appear continuous
  • 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 and synchronization between processes for these systems is critical
• Therefore, concurrent process model best suited for describing these systems
Real-Time Operating Systems

- Provide mechanisms, primitives, and guidelines for building real-time embedded systems
- Windows CE
  - Built specifically for embedded systems and appliance market
  - Scalable real-time 32-bit platform
  - Supports Windows API
  - Perfect for systems designed to interface with Internet
  - Preemptive priority scheduling with 256 priority levels per process
  - Kernel is 400 Kbytes
- QNX
  - Real-time microkernel surrounded by optional processes (resource managers) that provide POSIX and UNIX compatibility
    - Microkernels typically support only the most basic services
    - Optional resource managers allow scalability from small ROM-based systems to huge multiprocessor systems connected by various networking and communication technologies
  - Preemptive process scheduling using FIFO, round-robin, adaptive, or priority-driven scheduling
  - 32 priority levels per process
  - Microkernel < 10 Kbytes and complies with POSIX real-time standard
Summary

- Computation models are distinct from languages
- Sequential program model is popular
  - Most common languages like C support it directly
- State machine models good for control
  - Extensions like HCFSM provide additional power
  - PSM combines state machines and sequential programs
- Concurrent process model for multi-task systems
  - Communication and synchronization methods exist
  - Scheduling is critical
- Dataflow model good for signal processing