Embedded Systems Design: A Unified Hardware/Software Introduction

Chapter 8: State Machine and Concurrent Process Model

Outline

- Models vs. 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 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...

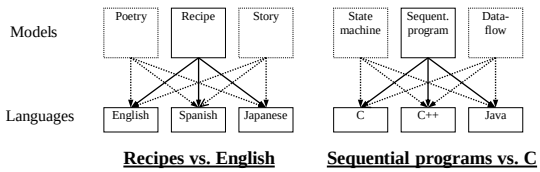
An example of trying to be precise in English

- 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

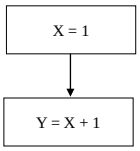


- Computation models describe system behavior
  - Conceptual notion, e.g., recipe, sequential program
- Languages capture models
  - Concrete form, e.g., English, C
- Variety of languages can capture one model
  - E.g., sequential program model → C,C++, Java
- One language can capture variety of models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

Text versus 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;
```



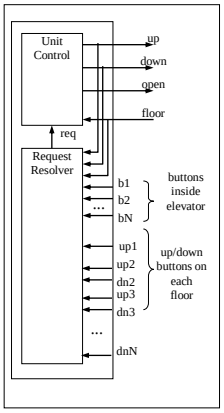
Introductory example: An elevator controller

- 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



Elevator controller using a sequential program model

Sequential program model

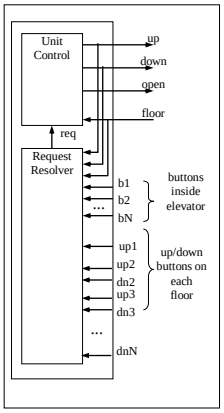
```
Inputs: int floor; bit b1..bN; up1..upN-1; dn2..dnN;
Outputs: bit up, down, open;
Global variables: int req;

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;
    open = 1;
    delay(10);
  }
}

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

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

You might have come up with something having even more if statements.

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

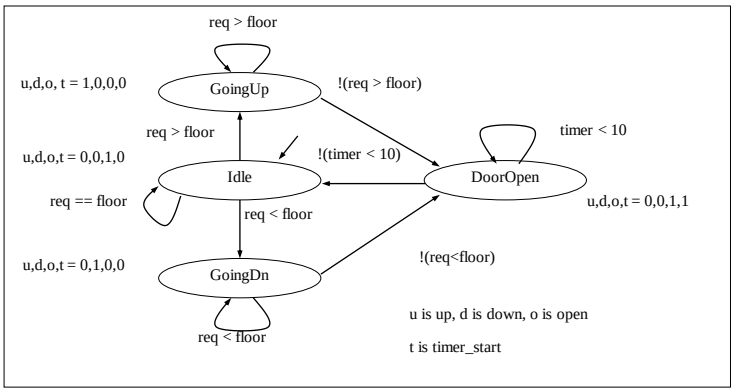


Finite-state machine (FSM) model

- 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**



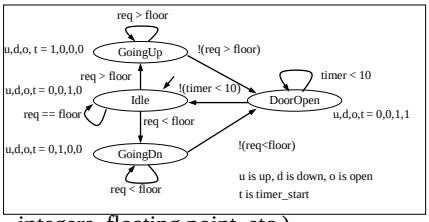
Formal definition

- An FSM is a 6-tuple $F<S, I, O, F, H, s_0>$
  - 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 x $I \rightarrow S$)
  - 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 x $I \rightarrow O$)
- 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)

Finite-state machine with datapath model (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, $\underline{V}$, 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 $\{o_0, o_1, \ldots, o_n\}$
  - **$\underline{V}$ is a set of variables $\{v_0, v_1, \ldots, v_n\}$**
  - F is a next-state function (S x I x $V \rightarrow S$)
  - H is an **action** function ($S \rightarrow O + V$)
  - s_0 is an initial state

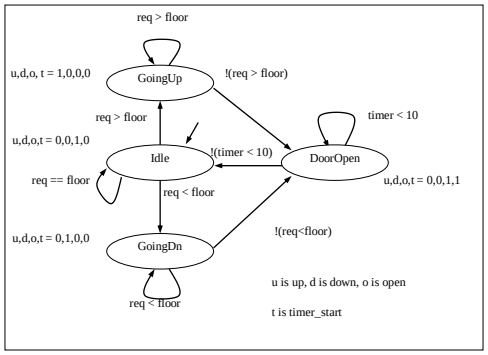
We described UnitControl as an FSMD



- 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 as a state machine

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



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 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)

Try Capturing Other Behaviors with an FSM

- E.g., Answering machine blinking light when there are messages
- E.g., A simple telephone answering machine that answers after 4 rings when activated
- E.g., A simple crosswalk traffic control light
- Others

Capturing state machines in sequential programming language

- 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 subset approach

- 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 = …;}

```
#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**

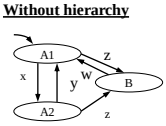
General template

```
#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;
      }
   }
}
```

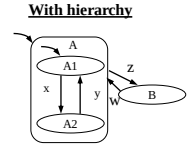

HCFSM and the Statecharts language

- Hierarchical/concurrent state machine model (HCFSM)
  - Extension to state machine model to support hierarchy and concurrency
  - States can be decomposed into another state machine
    - *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

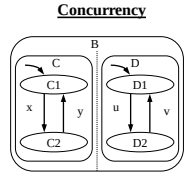
Without hierarchy



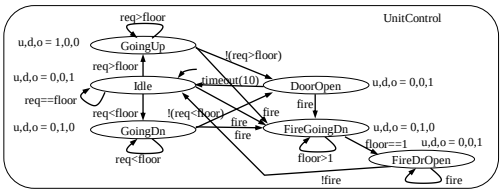
With hierarchy



Concurrency

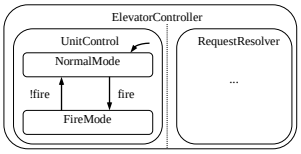


UnitControl with *FireMode*



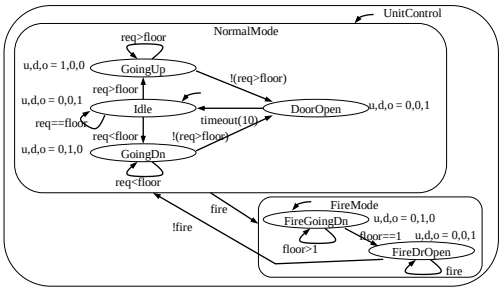
Without hierarchy

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



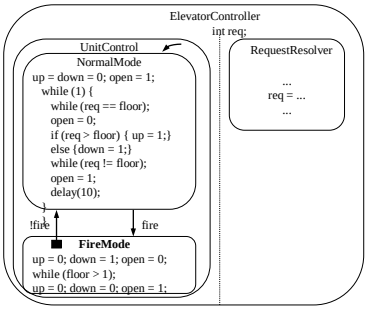
With concurrent *RequestResolver*

With hierarchy



Program-state machine model (PSM): HCFSM plus sequential program model

- 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



- *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

Role of appropriate model and language

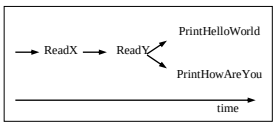
- Finding appropriate model to capture embedded system is an important step
  - Model shapes the way we think of the system
    - Originally thought of sequence of actions, wrote sequential program
      - First wait for requested floor to differ from target floor
      - Then, we close the door
      - 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

```
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

- 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



Subroutine execution over time

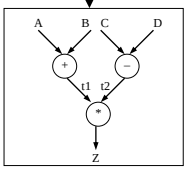
```
Enter X: 1
Enter Y: 2
Hello world.   (Time = 1 s)
Hello world.   (Time = 2 s)
How are you?   (Time = 2 s)
Hello world.   (Time = 3 s)
How are you?   (Time = 4 s)
Hello world.   (Time = 4 s)
...
```

Sample input and output

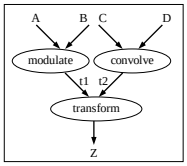
Dataflow model

- 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) * (C - D)



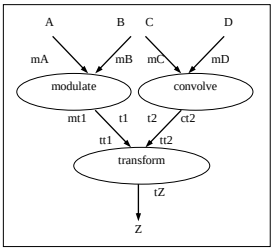
Nodes with arithmetic transformations



Nodes with more complex transformations

Synchronous dataflow

- With digital signal-processors (DSPs), data flows at fixed rate
- Multiple tokens consumed and produced per firing
- Synchronous dataflow model takes advantage of this
  - 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
- How would you map this model to a sequential programming language? Try it...
- Algorithms developed for scheduling nodes into "single-appearance" schedules
  - Only one statement needed to call each node's associated procedure
    - Allows procedure inlining without code explosion, thus reducing overhead even more

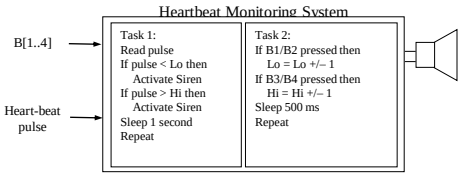


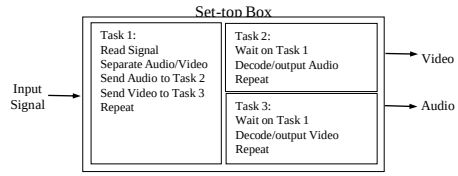
Synchronous dataflow

Concurrent processes and real-time systems

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



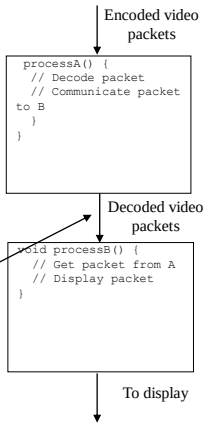


Process

- 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 HelloWord/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

Communication among processes

- 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 display decoded packets on a screen
- How do we achieve this communication?
  - Two basic methods
    - Shared memory
    - Message passing



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

```
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

- 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

```
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 */
  }
}
```

Back to Shared Memory: 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
- 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

Correct Shared Memory Solution to the Consumer-Producer Problem

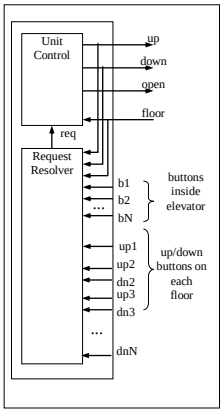
- The primitive *mutex* is used to ensure critical sections are executed in mutual exclusion of each other
- Following the same execution sequence as before:
  - *A*/*B* execute *lock* operation on *count_mutex*
  - Either *A* **or** *B* will acquire *lock*
    - Say *B* acquires it
    - *A* will be 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

```
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: }
```

Process Communication

- Try modeling "req" value of our elevator controller
  - Using shared memory
  - Using shared memory and mutexes
  - Using message passing

System interface



A Common Problem in Concurrent Programming: 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

```
mutex mutex1, mutex2;
void processA() {
  while( 1 ) {
    …
    mutex1.lock();
    /* critical section 1 */
    mutex2.lock();
    /* critical section 2 */
    mutex2.unlock();
    /* critical section 1 */
    mutex1.unlock();
  }
}
void processB() {
  while( 1 ) {
    …
    mutex2.lock();
    /* critical section 2 */
    mutex1.lock();
    /* critical section 1 */
    mutex1.unlock();
    /* critical section 2 */
    mutex2.unlock();
  }
}
```

Synchronization among 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 variable example: consumer-producer

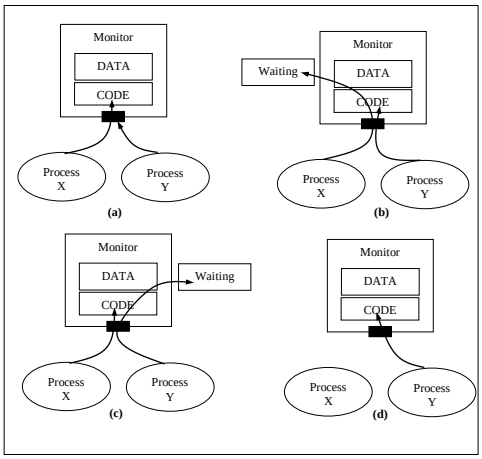
- 2 condition variables
  - *buffer_empty*
    - Signals at least 1 free location available in *buffer*
  - *buffer_full*
    - Signals at least 1 valid data item in *buffer*
- *processA*:
  - produces data item
  - acquires lock (*cs_mutex*) for critical section
  - checks value of *count*
  - if *count* = *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*

Consumer-producer using condition variables

```
data_type buffer[N];
int count = 0;
mutex cs_mutex;
condition buffer_empty, buffer_full;
void processA() {
  int i;
  while( 1 ) {
    produce(&data);
    cs_mutex.lock();
    if( count == N ) buffer_empty.wait(cs_mutex);
    buffer[i] = data;
    i = (i + 1) % N;
    count = count + 1;
    cs_mutex.unlock();
    buffer_full.signal();
  }
}
void processB() {
  int i;
  while( 1 ) {
    cs_mutex.lock();
    if( count == 0 ) buffer_full.wait(cs_mutex);
    data = buffer[i];
    i = (i + 1) % N;
    count = count - 1;
    cs_mutex.unlock();
    buffer_empty.signal();
    consume(&data);
  }
}
void main() {
  create_process(processA); create_process(processB);
}
```


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



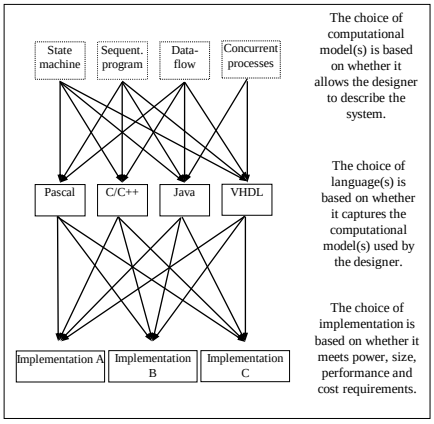
Monitor example: consumer-producer

- 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
  - 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;
06:   void processA() {
07:     int i;
08:     while( 1 ) {
09:       produce(&data);
10:       if( count == N ) buffer_empty.wait();
12:       buffer[i] = data;
13:       i = (i + 1) % N;
14:       count = count + 1;
15:       buffer_full.signal();
16:     }
17:   }
18:   void processB() {
19:     int i;
20:     while( 1 ) {
21:       if( count == 0 ) buffer_full.wait();
23:       data = buffer[i];
24:       i = (i + 1) % N;
25:       count = count - 1;
26:       buffer_empty.signal();
27:       consume(&data);
28:       buffer_full.signal();
29:     }
30:   }
31: }  /* end monitor */
32: void main() {
33:   create_process(processA); create_process(processB);
35: }
```

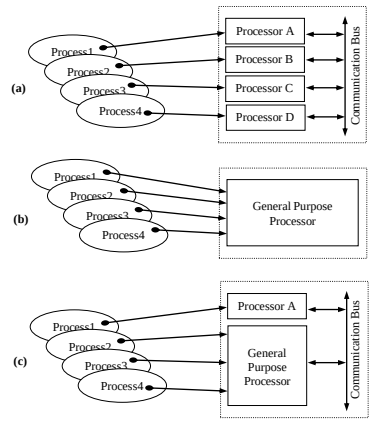

Implementation

- 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



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



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., 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 vs. 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

Implementation: 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

Implementation: 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

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

Deadline monotonic

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

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 (RTOS)

- 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