Embedded Software Programming

Computer Science & Engineering Department
Arizona State University
Tempe, AZ 85287

Dr. Yann-Hang Lee
yhlee@asu.edu
(480) 727-7507
Event and Time-Driven Threads

\[ \text{taskSpawn (name, priority, options, stacksize, main, \ldots)} \]

- **initialization**
  - external trigger?
    - ISR: to set/clear events
    - Take actions and change system state
  - start_time = time()
    - computation
      - Sleep(period - (time() - start_time))
Multiple Events in One Thread

```c
void compute()
{
    if (event1) then action1;
    if (event2) then action2;
    if (event3) then action3;
    .
}

or
{
    for (i=0, i<n, i++)
        if event[i] then action[i];
}
```

```c
void compute()
{
    if (event1) then action1;
    else if (event2) then action2;
    else if (event3) then action3;
    .
}

or
{
    for (i=0, i<n, i++)
        if event[i] then {
            action[i];
            break;
        }
}
```
Real-time System Specification

- **Logical correctness requirements:**
  - The computation produces correct outputs.
  - Models of computation to describe inputs and computations
  - Additional requirements on resource, security, reliability, etc.
  - Finite state machine
    - good for control logic and protocols,
    - transition and activity
  - Data flow – modular computations that are triggered by the availability of input data.

- **Temporal correctness requirements:**
  - The computation produces outputs at the right time
  - When the computation can get started and should be completed
# Specification Patterns

<table>
<thead>
<tr>
<th>Category</th>
<th>Pattern</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Duration (stimuli and responses)</td>
<td>minimum duration</td>
<td>The system has a minimum 'off' period of 120 seconds before it reenters the cranking mode.</td>
</tr>
<tr>
<td></td>
<td>maximum duration</td>
<td>The system can only operate in engine cranking mode for no longer than 10 seconds at one time.</td>
</tr>
</tbody>
</table>
| Periodic                       | bounded recurrence   | The ABS controller checks for wheel skidding every 10 milliseconds."
| Real-time order                | bounded response     | The detection of and response to rapid deceleration must occur within 0.015 seconds.                                                |
|                                | bounded invariance    | If Error 502 is received, then the braking system is inhibited for 10 seconds.                                                         |

RT Specification in FSM

- Duration of staying in a state
- Periodic activity in a state
- Bounded response for each transition
- Accumulated delay between multiple transitions
- Hierarchical FSM
  - a state encloses a FSM
  - enter a state $\rightarrow$ activate a FSM
- Concurrent FSM
  - FSMs run in parallel (active simultaneously)
RT Specification in Data Flow

- The execution of actors is driven by data or clock
- Communication mechanism and delay
- End to end delay of each data stream
  - delay at each stage
  - scheduling of actors
  - input stream

Diagram:
- navigation
- sensor processing
- integrated navigation
- guidance processing
- flight plan processing
Task model

- **Periodic task** $T_i$: (examples ??)
  - constant (or bounded) period, $p_i$: inter-release time between two consecutive jobs
  - phase $\phi_i$, utilization $\mu_i = e_i / p_i$, deadline (relative) $D_i$

- **Aperiodic and sporadic**: (examples ??)
  - uncertain interarrival times but with a minimum separation
  - aperiodic: with a soft or no deadline
  - sporadic: with a hard deadline
Terminology: Temporal Parameters

- **Release time:**
  - fixed \((r)\), jitter \([r-\Delta, r+\Delta]\), sporadic or aperiodic

- **Execution time:**
  - uncertainty from memory refresh, contention due to DMA, cache misses, interrupts, OS overhead
  - execution path variations

- **WCET:** a “deterministic” parameter for the worst-case execution time
  - a conservative measure
  - an assumption to make scheduling and validation easier
  - how can you measure the WCET of a job?
Terminology

- **Hard deadline**: late result is little or no value, or may lead to disaster
  - need to validate (can you guarantee it?)

- **Soft deadline**: late result may still be useful
  - probability of missing deadlines
  - 95% of telephone switch connects in 10 seconds

- **How serious is serious?**

- **Tardiness**:
  - \( \min \{ 0, \text{deadline} - \text{completion time} \} \)

- **Usefulness**:
  - function of tardiness

![Diagram of tardiness and usefulness](chart.png)
Task Functional Parameters

- **Preemptivity**: suspend the executing job and switch to the other one
  - should a job (or a portion of job) be preemptable
  - context switch: save the current process status (PC, registers, etc.) and initiate a ready job
  - transmit a UDP package, write a block of data to disk, a busy waiting loop

- **Preemptivity of resources**: concurrent use of resources or critical section
  - lock, semaphore, disable interrupts

- **How can a context switch be triggered?**
  - Assume you want to preempt an executing job -- why
    - a higher priority job arrives
    - run out the time quantum
Synchronous or Asynchronous (1)

- **Synchronous:**
  - atomic reactions indexed by a global logical clock,
  - each reaction computes new events for its outputs based on its internal state and on the input values
  - the communication of all events between components occur synchronously during each reaction.

- **Cycles of reading inputs, computing reaction and producing outputs**
  - Synchronous = 0-delay = within the same cycle
  - No interference between I/O and computation

- **Simulink? and other synchronous languages**

- **Why?**
  - deterministic semantics in the presence of concurrency.
a high priority task B arrives and receives inputs from A (from A_1 or A_2?)

if Pri(A) > Pri(C)>Pri(B), depending upon the execution time of C, B may receive inputs from A_1 or A_2
Synchronous or Asynchronous (3)

- If execution time = 0, then the computation is determined by the order of arrivals, not the arrival instances, nor execution time
- Can we memorize the arrival order and then fetch data from buffer
Reader/Writer -- ISR and Buffering

- **Input**: single producer (ISR) and single consumer (thread)
- **If a read is initialized by the thread**
  - calls "read" with a buffer of $n$ bytes
  - initiate IO operation, enable interrupt
  - ISR reads input and store in the buffer.
  - If done, single the completion
- **Blocking or nonblocking**
  - in thread context (e.g. vxWorks) – semaphore, lock
  - in kernel context (Linux) – wait queue
- **Guarded access**
  - Lock (mutex) and interrupt lock (disable)
Deadlock

- **Permanent blocking of a set of threads that are competing for a set of resources.**

- **Necessary conditions for a deadlock**
  - Mutual exclusion: holds a resource in a non-sharable mode.
  - Hold-and-wait: must hold at least one allocated resource while awaiting one or more resources held by other processes.
  - No preemption: Resources not forcibly removed from a process holding it.
  - Circular wait: a closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain.
Resource Allocation Graph

- If graph contains no cycles ⇒ no deadlock.
- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock.
  - if several instances per resource type, possibility of deadlock.

(A. Silberschatz, G. Gagne, and P. B. Galvin, "Operating System Concepts")
Deadlock Prevention

- **Detect unsafe locking**
  - search resource allocation graph for a cycle

- **Eliminate mutual exclusion:**
  - not possible in most cases
  - spooling makes I/O devices sharable

- **Eliminate hold-and-wait**
  - request all resources at once
  - release all resources before a new request
  - release all resources if current request blocks

- **Eliminate circular wait**
  - order all resources: \( \text{SEQ}(R_i) \neq \text{SEQ}(R_j) \)
  - process must request in ascending order
Distributed Deadlock Prevention

- **Resources that cause deadlock**
  - locks, guarded methods
  - limited number of threads

- **Based on wait-for graphs (Chandy-Misra-Haas algorithm).**
  - A process wishing to see if it is involved in a deadlock sends a probe message on all outgoing request arcs.
  - The message is a triple (original blocked process, current sender, current destination).
  - Any process receiving such a message passes it on on any outgoing arcs, with modified sender and receiver.
  - If the message returns to the original sender then a cycle exists.
Overrun Management

- **Imprecise computation**
  - trades off precision for timeliness during a transient overload.
  - A task consists of two or more logical parts: a mandatory part and at least one optional part.
    - The mandatory part must meet deadline constraint
    - The optional part only affect the quality of result

- **Implementation**
  - Synchronous approach: polling after each iteration of optional computation
  - Separate tasks for mandatory and for optional parts
    - The optional task (OT) sends results back to the mandatory task (MT)
    - When running out the allocated execution time, MT puts out results and kill OT
  - Asynchronous transfer of control (Exception)
Asynchronous Transfer of Control

- Communication between threads may be either synchronous or asynchronous.
- Asynchronous communication
  - Resumption (through signals and signal handling) model and termination model
- If some change in the system environment needs immediate attention
  - Time out on a computation
  - Terminate a thread
  - Terminate one loop of computation
- A controversial issue
  - Difficult to write correct code
  - Release resources
  - Performance penalty
C and C++ Exceptions

- C does not define any exception handling facilities
- C++ exception: try, throw, and catch
  - Cannot throw in signal handler
- To implement ATC model, it is necessary to save the status of a program's registers etc. on entry to an exception domain and then restore them if an exception occurs.
  - The POSIX facilities of `setjmp` and `longjmp` can be used for this purpose
  - Finalization: done by the programmer
- Language support of ATC: Ada and RTSJ
The function will throw *DivideByZero* as an exception
- caught by an exception-handling catch statement that catches exceptions of type `int`.
- The necessary construction is a try catch system.
- So, a program that checks for exceptions and may have exceptions thrown must be enclosed in a try block.

```cpp
const int DivideByZero = 10;
//....
double divide(double x, double y)
{
    if(y==0)
    {
        throw DivideByZero;
    }
    return x/y;
}
```

```cpp
try
{
    divide(10, 0);
}
catch(int i)
{
    if(i==DivideByZero)
    {
        cerr<<"Divide by zero error";
    }
}
```

**setjmp and longjmp**

- **setjump**
  - saves the program status and returns a 0

- **longjmp**
  - restores the program status and results in the program abandoning its current execution and restarting from the position where setjump was called
  - this time setjump returns the values passed by longjmp

- **See the example in the next slide**
  - Program status is saved in a global variable *jumper* of type *jum_buf*.
  - One may need different exception handlers in different functions. So how will SigHandler() know which *jumper* to use?
/* The simplest error handling based on setjmp() and longjmp() */

jmp_buf jumper;

void sigHandler()
{
    if (overrun) longjmp(jumper, -1); /* can't divide by 0 */
    return;
}

void SomeTask (void)
{
    int result;
    if (setjmp(jumper) == 0)
    {
        result = SomeComputation; /* continue working and save result */
    }
    else
    {
        put_out(result); /* overrun, send out the saved result */
    }
}