Embedded Systems Programming

Overrun Management
(Module 23)

Yann-Hang Lee
Arizona State University
yhlee@asu.edu
(480) 727-7507

Summer 2014
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
C++ exception throw and catch

- 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;
}

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

(http://www.cprogramming.com/tutorial/exceptions.html)
**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 */
    }
}
Task Preemption

- **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