Embedded Systems Design and Modeling

Chapter 11

Multitasking

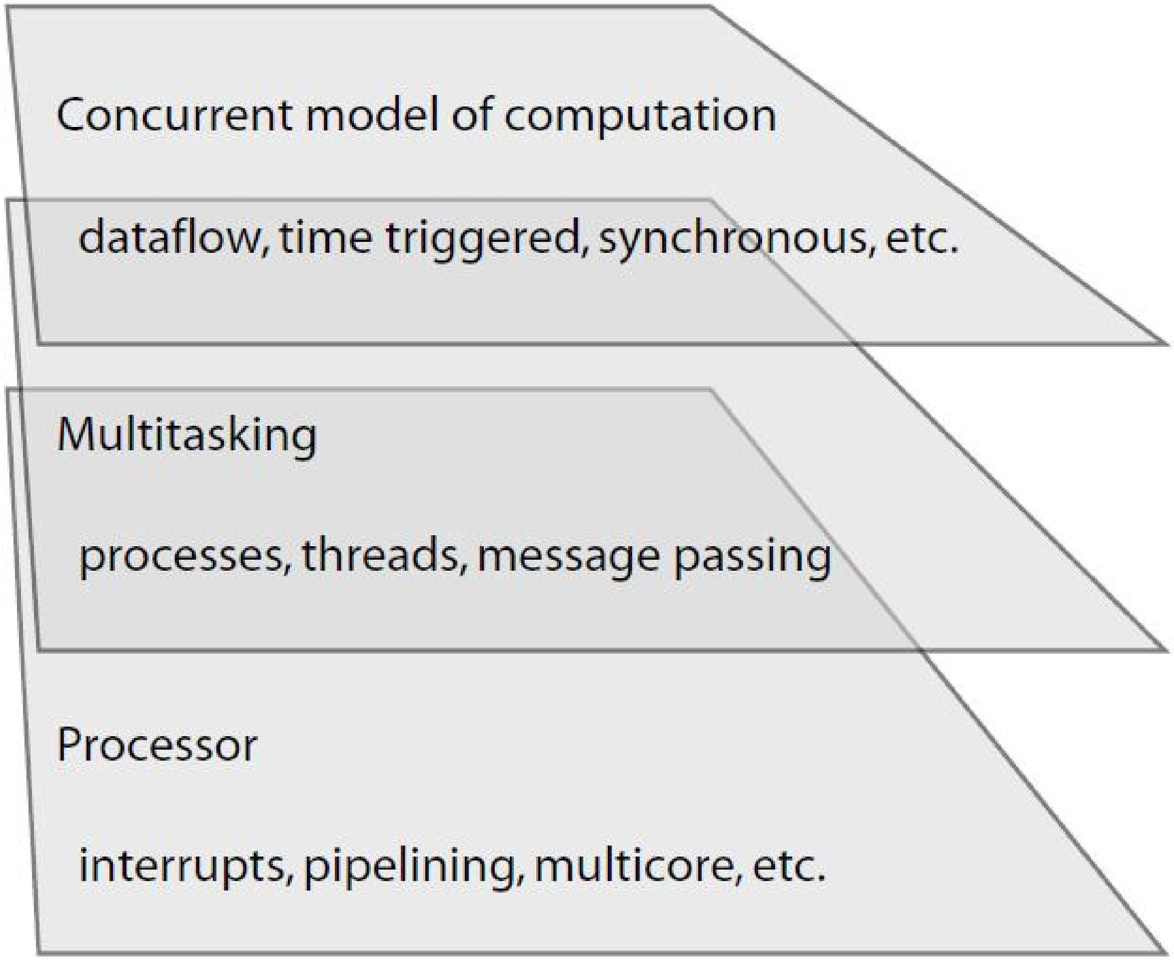
Outline

- Concurrency levels of implementation
- Concurrency motivation
- Basic concepts
- Imperative programs and FSM’s
- Threads and their problems
- Processes and their communications

Concurrency Implementation

- Concurrency can be implemented and modeled at different levels of abstraction:
  - High level issues were considered in the concurrent MoC's (Chapter 6)
  - Low level issues are implemented in hardware and software in processors (Chapters 7 to 10 which are not covered in this course)
  - There are mid-level mechanisms to implement concurrency in software (focus of this chapter)
- Shown graphically in the next slide

Concurrency Abstraction Levels



Justification

- Motivations for and uses of concurrency:
  1. Improving responsiveness by giving priority to time-sensitive tasks (HW and SW interrupts):
     - Long-running, less critical programs give way to more critical tasks
     - This reduces latency from stimulus to response
  2. Improving performance by allowing multi-processor execution of a program (exploiting parallelism in hardware)
  3. Creating the illusion of simultaneity by running multiple programs on a single hardware (multitasking)
  4. Direct control over the real-time constraints:
     - Needed when a task has to be executed at or finished by an exact time

Basic Concepts

- Multitasking definition: (apparently) simultaneous execution of multiple tasks
- Design tools that support concurrent MoC’s usually handle the issues at higher levels of abstraction
- The high-level description is expected to automatically generate lower level implementations of concurrency
- So the designers may not need to handle concurrency issues at lower levels

Basic Concepts (Continued)

- But if concurrency needs to be done at mid-level, then OS usually takes care of it
- Even if the OS handles it, it is still tricky
- If not done by OS, it is even harder to handle it and requires deep understanding of the concurrency issues
- Our only tool at mid-level:
  - Programs written in one of the programming languages
  - All are inherently imperative

Imperative Programs

- Imperative programs: consist of a sequence of operations or instructions
- All existing programming languages are imperative in nature
- How to model and implement concurrency using imperative programs?
  - Extended FSM's can model the behavior of an imperative program with fixed and bounded variables
  - But there is no one-to-one correspondence due to potential complexity of a program

Mid-Level Concurrency Challenges

- The potential complexity of a program’s data structure can be a problem:
  - A theoretically unbounded data structure (like linked list) cannot be described accurately by an FSM
- Adding concurrency makes it even more complex:
  - Handling it at mid-level is very difficult and error-prone (as will be shown)
- Threads are used to handle some of these difficulties

Threads

- Threads are imperative pieces of code that run concurrently and share memory
- The most common form of threads: interrupts (even without OS involvement)
- It is possible to create threads at a higher level than an interrupt:
  - OS provides a collection of procedures (called API) for use by programmers
  - Might even allow creation of processor-independent and OS-independent code
  - Example: Pthreads (POSIX threads)

Pthreads

- Pthreads is an API implemented by many operating systems, both real-time and not
- A library of C procedures
- Standardized by the IEEE in 1988
- Currently implemented in most modern operating systems
- Pthreads may not be apparently visible in a programming language, but they are still used behind the scene
- Example: Java

Pthreads Usage Example

```c
#include <pthread.h>
#include <stdio.h>
void* printN(void* arg) {
    int i;
    for (i = 0; i < 10; i++) {
        printf("My ID: %d\n", *(int*)arg);
    }
    return NULL;
}
int main(void) {
    pthread_t threadID1, threadID2;
    void* exitStatus;
    int x1 = 1, x2 = 2;
    pthread_create(&threadID1, NULL, printN, &x1);
    pthread_create(&threadID2, NULL, printN, &x2);
    printf("Started threads.\n");
    pthread_join(threadID1, &exitStatus);
    pthread_join(threadID2, &exitStatus);
    return 0;
}
```

Thread Implementation

- Needs a scheduler:
  1. In general, thread scheduling and predicting their behavior is difficult
  2. Without an OS, multithreading is achieved with interrupts and timing is determined by external events
  3. Generic and non-real-time OS's (like Linux, Windows, …) provide thread libraries (like Pthreads) and provide no fixed guarantees about when threads will execute => no guarantee on concurrency

Thread Implementation (Cont'd)

4. Real-time operating systems (RTOS's), like RTLinux and Windows CE, support a variety of ways of controlling when threads execute (process creation and killing, priorities, preemption policies, deadlines, ...)

Notes About Threads

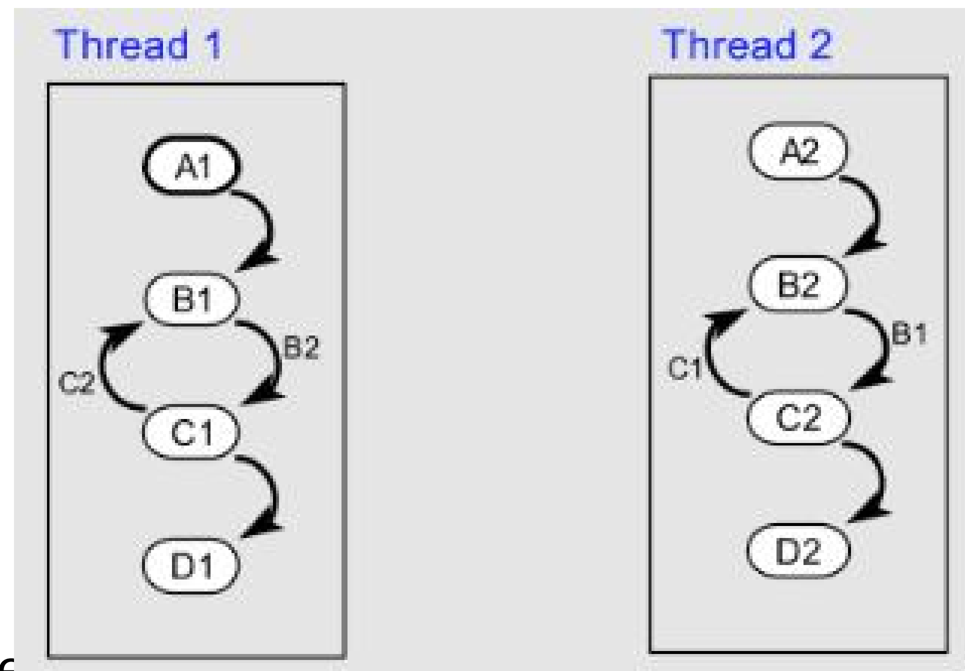
- Threads may or may not begin running when created
- A thread may be suspended between any two *atomic* instructions to execute another thread and/or interrupt service routine
  - Atomic instructions: typically assembly language instructions, not high-level language statements
  - States or transitions can also represent atomic instructions

Notes (Continued)

- Threads can often be given priorities, and these may or may not be respected by the operating system or the thread scheduler
- Threads may block on semaphores
- If two threads compete to access the same resource (race condition), the result may depend on the order of their accesses
- To prevent failures in these cases mutual exclusion locks (mutex) are used

Mutual Exclusion Lock

- In this figure, each state represents an atomic instruction
- Choose one thread arbitrarily
- Advance to a next state if guards are satisfied
- Repeat this!
- If done properly, one thread has to wait for the other before it can proceed



Mutex

- A mutual exclusion lock prevents any two threads from simultaneously accessing or modifying a shared resource
- The code between the lock and unlock is a critical section
- At any one time, only one thread can be executing code in the critical section
- A programmer may need to ensure that all accesses to a shared resource are similarly protected by locks

Risk of Deadlock

- Deadlock is when one or more thread is permanently blocked waiting to acquire locks
- Some possible ways to prevent deadlocks:
  1. Use only one lock in an entire program
     - Loses modular programming, misses real-time constraints
  2. Enable/disable interrupts when accessing a shared resource
     - Doesn't work if a thread is suspended due to other reasons

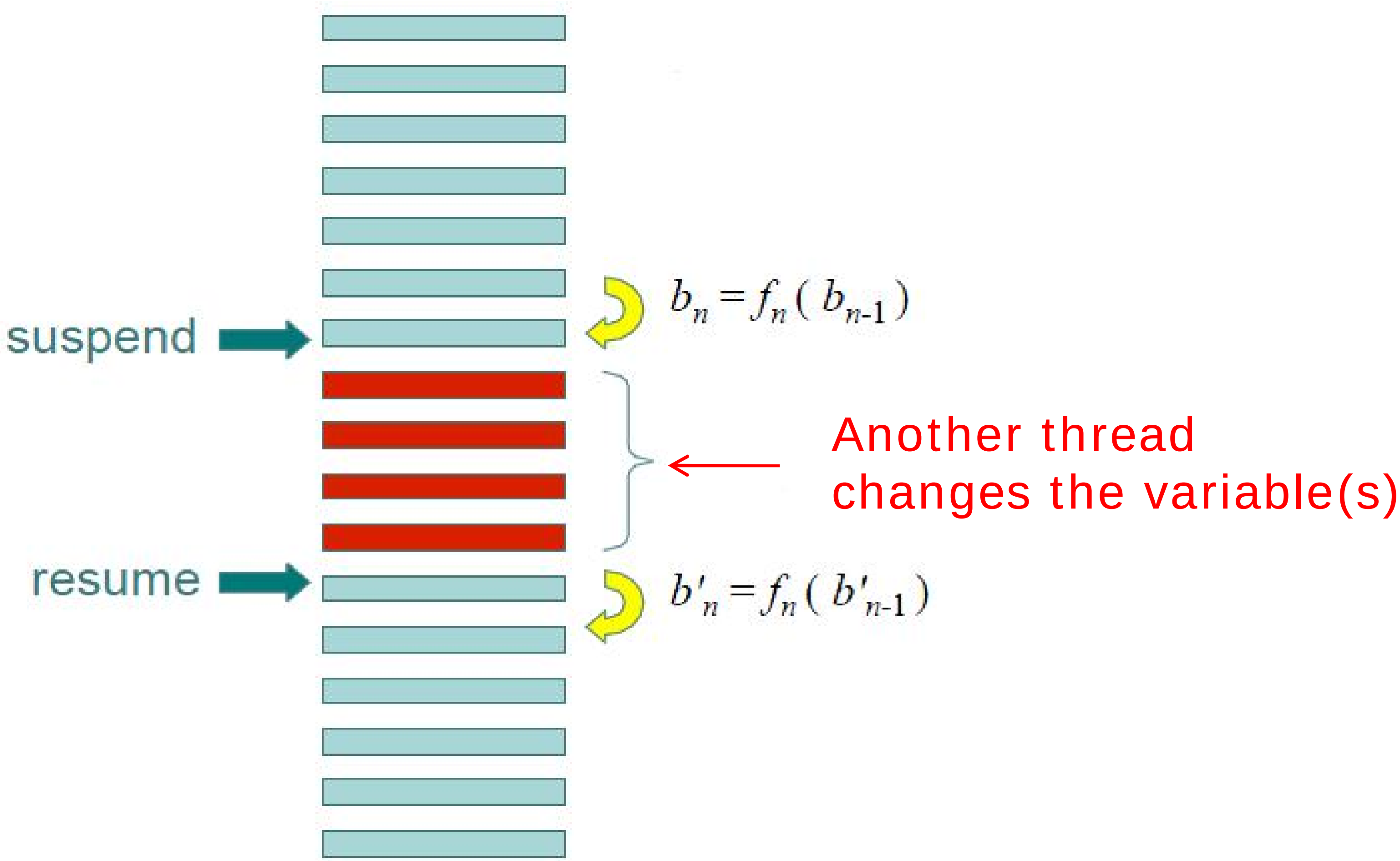
Deadlock Preventions (Cont'd)

- Some possible ways to prevent deadlocks:

3. Ensure all threads acquire and release their locks in the same order
    - Very hard to guarantee:
        - For example when a team is developing the code
    - Sometimes impossible:
        - For example when a code needs to be called, it acquires a lock but other locks may have to be released first => the code may be suspended
4. Other solutions exist but (almost) all of them impose tight constraints on the program or the programmer(s)

Memory Consistency Problem

- The value of shared variables depend on the order of threads execution



Memory Consistency Solutions

- Generate a memory consistency model:
  - A model to determine the effects of the threads on each other's variables
- Simplest model:
  - Assume variables are updated in the order of program instructions (sequential consistency)
- Problems:
  1. Sequential consistency is impossible with Pthreads
  2. Instructions can be reordered by the compiler and/or the hardware

Summarizing Thread Problems

- The main problem is in our multithread model:
  - From the perspective of any thread, the entire state of the universe can change between any two atomic actions (itself an ill-defined concept)
  - This notion makes them very nondeterministic
- The programmer's job is to prune away the nondeterminism by imposing constraints on execution order (e.g., mutexes) and limiting shared data accesses (e.g., OO design)
- Conclusion: threads are not the best way to create concurrency

Alternative: Processes

- Process: an imperative program which is a collection of threads with its own unshared memory space
  - Communication between processes must occur via OS facilities (like pipes or files)
    - Well-written procedures needed in a library by experts
    - Usually requires some form of hardware support:
      - Memory management unit (MMU)
  - The memory is not visible to other processes
  - Segmentation faults are attempts to access memory not allocated to the process

Inter-Process Communication

- Common process communication techniques:
  - File system:
    - Data can outlive a process
    - Simple but restricts file access to one process at a time
  - Message passing:
    - Communication takes place in form of messages passed in a repository called shared memory space
    - Prohibits direct sharing of data
    - Application programs can only access thru OS

Process Communication (Cont'd)

  - Optimum buffer size needed to avoid deadlock or memory waste
  - Can still suffer from deadlock
  - Can be order-dependent (non-deterministic)
- Final conclusion: solve concurrency issues at higher levels of abstraction (as much as possible)
- Chapter 11 homework assignments: 1 thru 5
- Due date: Tuesday 1403/3/1