CSE 237B Fall 2009
Tasks and Task Scheduling for Real Time

Rajesh Gupta
Computer Science and Engineering
University of California, San Diego.
The goal of task modeling and management is to understand the requirements of embedded software for application programming as well as for the operating system needs.

- Task management
- Task interaction

- Embedded Software as Tasks
  - Static and Dynamic Aspects of Task scheduling

- Memory Management: Stack and Heap
- Real-time kernels
- Commercial and research real-time operating systems
Tasks

• An embedded system typically has many activities (or tasks) occurring in parallel.
• A task represents an activity in the system.
• Historically, one task represents one sequential thread of execution;
  – however, multithreading allows multiple threads of control in the same task.
  – We will assume a single thread of control.
• The principles of concurrency are fundamental regardless of the granularity of the concurrent units (processes, tasks, or threads).
  – We will examine concurrency in terms of tasks.
Concurrent tasking means structuring a system into concurrent tasks.

Advantages of concurrent tasking

- A natural model for many real-time applications.
- Results in a separation of concerns of what each task does from when it does it. This usually makes the system easier to understand, manage, and construct.
- Can result in an overall reduction in system execution time by overlapping executions of independent tasks.
- Allows greater scheduling flexibility since time critical tasks with hard deadlines may be given a higher priority than less critical tasks.
- Identifying the concurrent tasks early in the design can allow an early performance analysis of the system.

However, concurrent tasking introduces complexity because of task interactions.
Task Interaction

• Often, tasks execute asynchronously, i.e., at different speeds, but may need to interact with each other.
• Three types of interactions are possible
  – communication
  – synchronization
  – mutual exclusion
• Communication is simply used to transfer data between tasks.
• Synchronization is used to coordinate tasks.
• Mutual exclusion is used to control access to shared resources.
Results of Interaction

• Task interactions lead to three types of behavior
  – independent
  – cooperating
  – competing

• Independent tasks have no interactions with each other.
• Cooperating tasks communicate and synchronize to perform some common operation.
• Competing tasks communicate and synchronize to obtain access to shared resources.
Task Implementation

- Two cases: dedicated versus shared resources.
- Implementing on dedicated resources (multiprocessing)
  - dedicate one processor for each task
  - connect processors using communication links such as a bus
  - Different arrangements are possible such as shared memory (one big memory shared by all but with local memories too) or distributed memory (all local memories).
- Implementing on shared resources
  - sharing the processor
  - sharing the memory
Shared Processor Implementation

• Issues in implementing tasks on a shared processor
  – how the processor is to be shared - what mechanisms are required to enable a processor executing one task to change its activity and execute another task
  – when the processor is to be shared - at what times, or as a result of what events, should the processor change from executing one task to executing another
  – which task should the processor direct its attention to, when sharing of the processor necessary (related to scheduling)

• How and when in serial execution
  – commence the next task at its starting point at the completion of the current task

• How and when in concurrent execution
  – commence the next task at the point where it previously left off when the current task gives up use of the processor
Shared Memory Implementation

• **Issues in implementing tasks on a shared memory**
  – provide enough memory to hold all the tasks, or
  – do code sharing and memory sharing

• **Code sharing through**
  – **serially re-useable code**
    • write the code in the subroutine shared (call it S) in such a way that it makes no assumptions about the values in its local variables when it is entered.
    • Using a lock and unlock pair, only one task can be made to use S at any time.
  – **re-entrant code**
    • In the above scheme, all the temporary areas that S needs reside in S. If these areas were to be part of the task currently using S, then it would consist of executable code only, and it could be executed by more than one task at a time, provided that S did not modify its own code in any way.
    • S uses the data areas indirectly, typically via a relocation pointer which is associated with each task and which is passed as a parameter when S is called.
Task Management

- A task can be in one of the states shown:

- task creation
  - In general, all tasks should be created before run time and remain dormant until needed.
  - This guarantees that the resource demands will be known and that performance can be evaluated with respect to real-time deadlines.
Task Modeling Issues

• Variations in the task models of concurrent programming languages are based on
  – structure
  – level of parallelism
  – granularity
  – initialization
  – termination
  – representation

• Structure
  – static: the number of tasks is fixed and known before run time.
  – dynamic: tasks are created at any time. The number of extant tasks is determined only at run time. For example, Ada and C.

• Level of parallelism
  – nested: tasks are defined at any level of the program text; in particular, they are allowed to be defined within other tasks. For example, Ada and C.
  – flat: tasks are defined only at the outermost level of the program text.
Task Modeling Issues -2

• **Granularity**
  – **coarse grain**: such a program contains relatively few big (long live history) tasks, e.g., Ada.
  – **fine grain**: such a program contains a large number of simple tasks.

• **Initialization**:  
  – when a task is created, it may need to supplied with information pertinent to its execution.
  – **Two ways to do that**
    • pass information in the form of parameters to the task
    • communicate explicitly with the task after it has started its execution
• **Termination under the following circumstances**
  – completion of execution of the task body
  – suicide, by execution of a “self-terminate” statement
  – abortion, through the explicit action of another task
  – occurrence of an un-trapped error condition
  – never; tasks are assumed to execute non-terminating loops
  – when no longer needed
Expressing Concurrency

- Representation: there are four basic mechanisms for expressing concurrent execution
  1. coroutines
  2. fork and join
  3. Cobegin/Coend
  4. explicit task declaration
1 Expressing Concurrency: Coroutines

• Like subroutines but allow control to pass explicitly between them in a symmetric rather than strictly hierarchical way.

• Control is passed from one coroutine to another by means of the “resume” statement which names the coroutine to be resumed.

• When a coroutine executes a resume, it stops executing but retains local state information so that if another coroutine subsequently resumes it, it can and will continue its execution.

• No run-time support system is needed as the coroutines themselves sort out their order of execution.

• In this scheme, tasks can be written by independent parties, and the number of tasks need not be known in advance.

• Certain languages such as Ada and Modula-2 have built-in support for coroutines.

• error-prone due to the use of global variables for communication

• Only one routine at a time.
Coroutines

coroutines
fork and join
cobegin and
coend
explicit task
declaration

A
Resume B
Resume A

B
Resume C

C
2 Expressing Concurrency: Fork and Join

- Fork specifies that a designated routine should start executing concurrently with the invoker of the fork.
- Join allows the invoker to synchronize with the completion of the invoked routine.
- Fork and join allow for *dynamic task creation* and provide a means of passing information to the child task via parameters.
  - *Usually only a single value is returned by the child on its termination.*
- Flexible but error-prone in use because they do not provide a structured approach to task creation.
- Available in Unix.
Fork and Join

Function F return …;
procedure P;

... 
C := fork F;
...
J := join C;
end P;

Consider two different scenarios: P finished before F, F finishes before P at J.
The completed process is suspended till join.
3 Expressing Concurrency: Cobegin and Coend

- **Cobegin**
  - a structured way of denoting the concurrent execution of a collection of statements
  - Tasks between a pair of cobegin and coend statements execute concurrently.
  - Can even support nesting of cobegins.
  - Occam-2 supports cobegins.

```plaintext
cobegin
  S1;
  S2;
  S3;
coend;
```
4 Explicit Task Declaration

- So far, sequential routines are composed to do parallel execution
- Explicit task declaration
  - Routines themselves state whether they will be executed concurrently.
  - Ada supports explicit task declaration by implicit task creation
    - all tasks declared within a block start executing concurrently at the end of the declarative part of that block.
  - Ada also supports dynamic task creation using the ‘new’ operator on a task type.
Example: Robot Arm Controller, Modula-1

MODULE main;
TYPE dimension = (xplane, yplane, zplane);

PROCESS Control (dim: dimension);
VAR position: integer; setting: integer;

BEGIN
Position:=0; (* rest position*)
LOOP
  new_setting (dim, setting);
  position:=position+setting;
  move_arm (dim, position);
END
END control;

BEGIN
  control(xplane);
  control(yplane);
  control(zplane);
END main.
Task Interaction : Communication

• Communication is based on
  – shared memory
  – message passing

• Shared memory-based communication
  – Each task may access or update pieces of shared information/data.

• Message passing-based communication
  – A direct transfer of information occurs from one task to another.

• Communication mechanisms
  – channels
  – pools
Communication Mechanisms

• Channels
  – provide the medium for items of information to be passed between one task and another
  – can hold more than one item at any time
  – usually have the items passing through in an ordered manner

• Pools
  – make items of information available for reading and/or writing by a number of tasks in the system
  – act as a repository of information; information does not flow within a pool
Implementing Communication

- **Channel** - provides a pipe of information passing from one task to another. For the tasks to run truly asynchronously, there must be some buffering of information; the larger the buffers, the greater the system flexibility.
  - queues
  - circular queues (or ring buffers or hoppers)
  - event flags
  - sockets and pipes

- **Pool** - usually takes the form of system tables, shared data areas, and shared files. Since a pool is shared by more than one task, it is essential to control strictly the access to information in pools.
  - mailboxes (or ports)
  - monitors

- In all cases involving a finite-sized structure, the size of the structure should be taken into account during the design phase of the system to prevent overflows.
Implementing Communication

• Queues
  – Items are placed on the tail of the queue by the sending task and removed from the head of the queue by the receiving task.
  – A common organization is First-In-First-Out (FIFO) organization in which the first item come in will be the first go out.
  – Items can have priorities and can be placed in the queue based on their priorities.
  – For large items such as arrays, it is better to use the address of the item in the queue. In this case, the producer task allocates the memory and the consumer task releases or reuses it.

• Circular queues
  – The underlying structure is a queue but the arrangement is like a ring in that items are placed into the slots in the
Implementing Communication

- Event flags
  - An event flag is associated with a set of related Boolean events. The flag maintains the state of the events and provides users access to read or modify the events. A task can wait for a particular event to change states.
  - In essence, they represent simulated interrupts, created by the programmer. Raising the event flag transfers control to the operating system, which can then invoke the corresponding handler. An example is the raise and signal facilities in C.
  - Liked by designers because they enable Boolean logic to be applied to events, e.g., a task can wait on the conjunction and/or disjunction of discrete events.
  - Poor mechanisms because they do not have content, and it is hard to decide who resets a flag’s state and what to do if a flag indicates the event is already set (or cleared).
Implementing Communication

• Sockets and pipes
  – most often associated with network-based systems and provide a reliable communication path
  – should be used if portability is more important than performance

• Mailboxes
  – A mailbox is a mutually agreed upon memory location that multiple tasks can use to communicate.
  – Each mailbox has a unique identification, and two tasks can communicate only if they have a shared mailbox.
  – uncommon in modern real-time systems

• Monitors
  – A monitor is defined over a channel or a pool and hides the internal structure of them.
  – A monitor is used to enforce synchronization (via condition variables) and mutual exclusion under the control of the compiler.
  – provide information hiding. Java uses monitors.
Task Synchronization

• Synchronization involves the ability of one task to **stimulate or inhibit** its own action or that of another task.
  – In other words, in order to carry out the activities required of it, a task may need to have the ability to say ‘stop’ or ‘go’ or ‘wait a moment’ to itself, or another task.

• Synchronization between two tasks centers around two significant events, **wait** and **signal**.
  – One task must wait for the expected event to occur, and the other task will signal that the event has occurred.

• Thus, synchronization can be implemented by assuming the existence of the following two procedures
  – **WAIT**(event), **SIGNAL**(event)

• **WAIT** and **SIGNAL** procedures are **indivisible** operations in that once begun, they must be completed and the processor cannot be swapped while they are being executed.
Implementing Synchronization

• **WAIT**(event)
  – causes the task to suspend activity as soon as the WAIT operation is executed, and it will remain suspended until such time as notification of the occurrence of an event is received.
  – Should the event have already occurred, the task will resume immediately.
  – A waiting task can be thought of as being in the act of reading event information from a channel or pool. Once this information appears, it can continue.

• **SIGNAL**(event)
  – broadcasts the fact that an event has occurred. Its action is to place event information in a channel or pool. This in turn may enable a waiting task to continue.

• **Implementing synchronization via semaphores**
  – a non-negative integer that can only be manipulated by WAIT and SIGNAL apart from the initialization routine
  – ‘event’ in WAIT and SIGNAL above refers to a semaphore
  – also used to manage mutual exclusion
Task Interaction - Mutual Exclusion

- Critical region
  - a sequence of statements that must appear to be executed indivisibly (or atomically)
- Mutual exclusion
  - the synchronization required to protect a critical region
  - can be enforced using semaphores
- Potential problems - due to improper use of mutual exclusion primitives
  - Deadlocks
  - Livelocks
  - Lockouts or starvation
  - Priority inversion
Mutual Exclusion Problems

• Deadlock
  – Two or more tasks are waiting indefinitely for an event that can be caused by only one of the waiting tasks.

• Livelock
  – Two or more tasks are busy waiting indefinitely for an event that can be caused by only one of the busy-waiting tasks.

• Lockout or starvation
  – One task that wished to gain access to a resource is never allowed to do so because there are always other tasks gaining access before it.

• Priority Inversion
  – Effective inversion in priority because of resource lock due to (transitively) dependent tasks.
Task Interaction - Mutual Exclusion

• If a task is free from livelocks, deadlocks, and lockouts, then it is said to possess **liveness**. This property implies that if a task wishes to perform some action, then it will, eventually, be allowed to do so.
  – In particular, if a task requests access to a critical section, then it will gain access within a finite time.

• Deadlocks are the most serious error condition among the three problems above. There are three possible approaches to address the issue of deadlock
  – **deadlock prevention**
  – **deadlock avoidance**
  – **deadlock detection and recovery**

• For a thorough discussion of these issues, refer to standard operating systems books,
  – e.g., Silberschatz and Galvin, because real-time systems use the same techniques.

Next Lecture: Structuring Embedded Software into multiple tasks.