Concurrent processes and real-time scheduling

55:036
Embedded Systems and System Software

Concurrent processes and real-time scheduling

- Often, an embedded system must carry out more than one task simultaneously
  - monitoring multiple sensors
  - controlling multiple devices
  - etc.
- A single embedded processor may be responsible for multiple tasks
- An embedded system may utilize multiple embedded processors

Simple Example of Concurrency in Embedded Systems

ISR
USART
Keyboard

 ISR
USART
Display

Concurrency Pitfalls—An Example

Customer1
int transAmt1, N;
for (i=0; i<N; i++) {
  deposit(transAmt1);
  withdraw(transAmt1);
}

Customer2
int transAmt2, N;
for (i=0; i<N; i++) {
  deposit(transAmt2);
  withdraw(transAmt2);
}

Bank Account
int balance = 0;
void deposit(int amount) {
  balance = balance + amount;
}
void withdraw(int amount) {
  balance = balance - amount;
}
Consider these two operations being performed concurrently. Net Result: 
Balance should be reduced by $30
Concurrency Pitfalls—An Example

1. read (old) balance
2. add amount to old balance
3. write updated balance

Bank Account
int balance = 0;
void deposit(int amount)  {
    balance = balance + amount;
}
void withdraw(int amount)  {
    balance = balance – amount;
}

Customer1
int transAmt1, N;
for (i=0; i<N; i++)  {
    deposit (20);
    withdraw(20);
}

Customer2
int transAmt2, N;
for (i=0; i<N; i++)  {
    deposit (50);
    withdraw(50);
}

1. read (old) balance
2. add amount to old balance
3. write updated balance

E.g. old balance = $100

$100 + 20

Updated Balance = $120
Concurrency Pitfalls—An Example

Bank Account

```java
int balance = 0;
void deposit(int amount) {
    balance = balance + amount;
}
void withdraw(int amount) {
    balance = balance – amount;
}
```

Customer1

```java
int transAmt1, N;
for (i=0; i<N; i++) {
    deposit (20);
    withdraw(20);
}
```

Customer2

```java
int transAmt2, N;
for (i=0; i<N; i++) {
    deposit (50);
    withdraw(50);
}
```

1. read (old) balance
2. add amount to old balance
3. write updated balance

1. read (old) balance
2. subtract amount from balance
3. write updated balance

RESULT IS WRONG!!!
SHOULD BE $70

Heartbeat Monitoring System

Task 1:
- Read pulse
- If pulse < Lo then
  - Activate Siren
- If pulse > Hi then
  - Activate Siren
- Sleep 1 second
- Repeat

Task 2:
- If B1/B2 pressed then
  - Lo = Lo +/– 1
- If B3/B4 pressed then
  - Hi = Hi +/– 1
- Sleep 500 ms
- Repeat

Set-top Box

Task 1:
- Read Signal
- Separate Audio/Video
- Send Audio to Task 2
- Send Video to Task 3
- Repeat

Task 2:
- Wait on Task 1
- Decode/output Audio
- Repeat

Task 3:
- Wait on Task 1
- Decode/output Video
- Repeat

Simple Concurrent Process Example

```java
ConcurrentProcessExample() {
    s = ReadX();
    p = ReadY();
    if (concurrent) {
        PrintHelloWorld(x) and PrintGoodbyeWorld(y);
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
        while (true) {
            print "Goodbye world."
            delay(1);
        }
    }
    PrintHelloWorld(x) {
        while (true) {
            print "Hello world."
            delay(1);
        }
    }
    PrintGoodbyeWorld(y) {
Process

- A sequential program, typically an infinite loop
  - Executes concurrently with other processes
  - We are about to enter the world of “concurrent programming”

- Other terms: task, thread

- 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

```
Process A() {
  // Decode packet
  // Communicate packet to B
}
```
```
void Process B() {
  // Get packet from A
  // Display packet
}
```

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

- Shared buffer
  - Count = # of valid data items in buffer
  - Process A produces data items and stores in buffer
  - Process B consumes data items from buffer

- Error when both processes try to update count concurrently
  - 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

```
data_type buffer[N];
int count = 0;
void processA() {
  int tail;
  while( 1 ) {
    produce(&data);
    while( count==N );/*loop*/
    buffer[tail] = data;
    tail = (tail + 1) % N;
    count = count + 1;
  }
}
```
```
void processB() {
  int head;
  while( 1 ) {
    while( count==0 );/*loop*/
    data = buffer[head];
    head = (head + 1) % N;
    count = count - 1;
    consume(&data);
  }
}
```
```
void main() {
  create_process(processA);
  create_process(processB);
}
```
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

```c
void processA() {
  while( 1 ) {
    produce(&data)
    send(B, &data);
    /* region 1 */
    receive(B, &data);
    consume(&data);
  }
}
```

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
  - 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

```c
int data_type buffer[N];
int count = 0;
mutex count_mutex;

void processA() {
  int tail;
  while( 1 ) {
    produce(&data);
    while( count==N ); /*loop*/
    buffer[tail] = data;
    tail = (tail + 1) % N;
    count_mutex.lock();
    count = count + 1;
    count_mutex.unlock();
  }
}

void processB() {
  int head;
  while( 1 ) {
    while( count==0 ); /*loop*/
    data = buffer[head];
    head = (head + 1) % N;
    count_mutex.lock();
    count = count - 1;
    count_mutex.unlock();
    consume(&data);
  }
}
```

Implementing Mutex-Locks

- Implementation of robust mutex locks is not a simple matter
- Often, hardware support is needed—more about this later
- Real-time operating systems typically provide locking, synchronization, and communication primitives—more about this later also.
Software Implementation of Mutex

• Note that a simple shared lock bit (or byte) doesn't work:

  e.g:

  ```c
  int mutex_lock;

  //CRITICAL SECTION
  mutex_lock = 0;
  ```

  ```c
  while (mutex_lock) { /*wait*/}
  ```

  ```c
  mutex_lock = 1;
  //CRITICAL SECTION
  mutex_lock = 0;
  ```

  ```c
  while (mutex_lock) { /*wait*/}
  ```

  ```c
  mutex_lock = 1;
  //CRITICAL SECTION
  mutex_lock = 0;
  ```

  WHY NOT???

P1 sees mutex_lock ==0

P2 sees mutex_lock ==0
Software Implementation of Mutex

- Note that a simple shared lock bit (or byte) doesn’t work: **WHY NOT???**
e.g.: ```c
int mutex_lock;

1 P1 sees mutex_lock == 0
2 P2 sees mutex_lock == 0
while (mutex_lock) { /*wait*/}
mutex_lock = 1;
//CRITICAL SECTION
mutex_lock = 0;
3 P1 sets mutex_lock = 1
4 P2 sets mutex_lock = 1
```

Requirements for a “correct” Mutex-lock

- Mutual exclusion
  - Guarantees only on process at a time in the Critical Section
- **WHAT ELSE???**

Software Implementation of Mutex

- Note that a simple shared lock bit (or byte) doesn’t work: **WHY NOT???**
e.g.: ```c
int mutex_lock;

1 P1 sees mutex_lock == 0
2 P2 sees mutex_lock == 0
while (mutex_lock) { /*wait*/}
mutex_lock = 1;
//CRITICAL SECTION
mutex_lock = 0;
3 P1 sets mutex_lock = 1
4 P2 sets mutex_lock = 1
```

Requirements for a “correct” Mutex-lock

- Mutual exclusion
  - Guarantees only on process at a time in the Critical Section
- Deadlock-free
  - Deadlock is a state where processes are permanently blocked
- Fairness (bounded waiting)
  - When multiple processes are “competing for” the critical section, they take turns fairly
- Progress
  - If critical section is not locked a requested process can gain access without waiting
Dekker's Algorithm

```plaintext
f0 := false;
f1 := false;
turn = 0; // or 1

p0: f0 = true;
    while (f1)
        if (turn != 0)
            f0 = false;
            while (turn != 0)
                f0 = true;
            // critical section
            turn = 1;
        f0 = false;

p1: f1 = true;
    while (f0)
        if (turn != 1)
            f1 = false;
            while (turn != 1)
                f1 = true;
            // critical section
            turn = 0;
        f1 = false;

// critical section
```

Mutual exclusion is insured: each process sets its flag BEFORE checking the other flag
If both flags are set, turn determines which one gets to proceed

Turn also insures fairness
**Dekker’s Algorithm**

- What about the other two properties: Deadlock freedom; progress.
  - Will leave it to you to confirm that the algorithm satisfies these properties
- Dekker’s algorithm can be easily extended to more than two processes
  - This extension is sometimes referred to as “Peterson’s algorithm”

**Mutual Exclusion/Critical Regions—Practical Considerations**

- In a single processor application, temporarily disabling interrupts is generally an effective way of implementing mutual exclusion

**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)
      - A executes lock operation on mutex2 (and acquires it)
      - A and B both execute in critical sections 1 and 2, respectively
      - A executes lock operation on mutex1
        - A blocked until B unlocks mutex2
      - B executes lock operation on mutex2
        - 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

**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

Monitor example: consumer-producer
Implementation: multiple processes sharing a 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 or real-time executive like RTOS
  - Language or library may rely on operating system to handle processes

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 (cooperative)
    - Only determines which process is next after current process finishes execution—i.e. voluntarily relinquishes the processor
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; can for 4 ms then is preempted
  - Process suspended for 14 ms, then back for the remaining 1 ms
  - Completed within 4 + 14 + 10 ms which means 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 Scheduling Example

Fixed-priority scheduling with two tasks, T1=50, C1=25, T2=100, C2=40

<table>
<thead>
<tr>
<th>Process</th>
<th>Period</th>
<th>Worst-case ex. time</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>50 ms</td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td>100 ms</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task</th>
<th>Deadline</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>50 ms</td>
<td>5</td>
</tr>
<tr>
<td>T2</td>
<td>100 ms</td>
<td>1</td>
</tr>
</tbody>
</table>

Rate Monotonic Scheduling Example

Fixed-priority scheduling with two tasks, T1=50, C1=25, T2=100, C2=40

<table>
<thead>
<tr>
<th>Task</th>
<th>Deadline</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>50 ms</td>
<td>5</td>
</tr>
<tr>
<td>T2</td>
<td>100 ms</td>
<td>1</td>
</tr>
</tbody>
</table>

Rate Monotonic Scheduling Example

Fixed-priority scheduling with two tasks, T1=50, C1=25, T2=100, C2=40

<table>
<thead>
<tr>
<th>Task</th>
<th>Deadline</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td>50 ms</td>
<td>5</td>
</tr>
<tr>
<td>T2</td>
<td>100 ms</td>
<td>1</td>
</tr>
</tbody>
</table>

Note: Processor Utilization = U = C1/T1 + C2/T2 = 0.5 + 0.4 = 0.9
Not All Tasks Are Schedulable

Some task sets aren’t schedulable (T1=50 C1=25, T2=75, C2=30)

U = .5 + .43 = .93

These tasks aren’t RMA schedulable even though U < 1.

Source: http://www.embedded.com/story/OEG20020221S0089

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.—i.e. most complex embedded systems
  – 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

Source: http://www.embedded.com/story/OEG20020221S0089
Real-time operating systems (RTOS)--Continued

• Symbian OS
  – Developed by a consortium of Mobile Phone manufacturers
  – Intended for “smart-phone” applications
  – Widely supported by development tools and environments
    • Java
    • Borland C++
    • etc
  – Platforms: x86, ARM, MIPS, Hitachi SuperH

Real-time operating systems (RTOS)--Continued

• There are a number of very low-overhead RTOS’s suitable for use in small embedded applications
  – PICC RTOS
  – FreeRTOS

PICC RTOS

• Supported by PICC PWH Complier
• Cooperative (non-preemptive) multitasking
• inter-task message passing

PICC RTOS Constructs

#USE RTOS
The CCS Real Time Operating System (RTOS) allows a PIC micro controller to run regularly scheduled tasks without the need for interrupts. This is accomplished by a function (RTOS_RUN()) that acts as a dispatcher. When a task is scheduled to run, the dispatch function gives control of the processor to that task. When the task is done executing or does not need the processor anymore, control of the processor is returned to the dispatch function which then will give control of the processor to the next task that is scheduled to execute at the appropriate time. This process is called cooperative multi-tasking.

Example:  #use rtos(timer=0, minor_cycle=20ms)
PICC RTOS—Continued

#TASK
Each RTOS task is specified as a function that has no parameters and no return. The #task directive is needed just before each RTOS task to enable the compiler to tell which functions are RTOS tasks. An RTOS task cannot be called directly like a regular function can.

```c
#task(rate=1000ms,max=100ms) // can be called
void The_first_rtos_task ( )
{
    printf("1\n\r");
}
```

PICC RTOS—Continued

- RTOS Functions:
  - RTOS_RUN()
  - RTOS_WAIT(sem)
  - RTOS_SIGNAL(sem)
  - RTOS_MESSAGE_SEND(task, byte)
  - RTOS_MSG_READ()
  - etc.
- See PICC Compiler Reference Manual for details

FreeRTOS

- Open source RTOS for embedded processors and microcontrollers
- Fully preemptive scheduler
- Support for message queues, semaphores, etc.
- Low overhead
  - Kernel requires 4-5 KB of Program Memory
  - 100-200 bytes of Data Memory
- Ports available for most microcontrollers and embedded processors
- www.freertos.org