Real time systems interact asynchronously with external entities and must cope with multiple threads of control and react to events - the executing programs need to share the CPU.

Concurrency in a computer application is when two or more sequential programs execute “at the same time” resulting in two or more sequences of machine instructions (executed one-after-another or interleaving on a CPU).

Concurrency is implemented on either operating system level or programming language level.

NOTE: when concurrency is implemented on multiple processors - we define it as distributed or parallel computing.
Task - Definition

- **Task**: a step-by-step complete sequence of instructions, *spawned for execution*, with a defined execution time designed to fulfill one of the system functions
- Each task may be executed **concurrently** with other tasks
- Each task is a separate *non-dormant* programming construct (implemented as a **process** or a **thread**) with its own execution sequence and the context (program counter, registers, memory, stack, etc.)

Patterns of Task Interactions (in Concurrent Problems)

- **Independent**: the tasks operate on separate parts of a problem, requiring no interaction
- **Competing**: the tasks require use of the same resource (CPU, shared data, display, file, peripheral)
- **Cooperating**: the tasks operate on common problem parts (overlapping at a few well defined points) and cooperate with each other at these points to solve the problem
Example: Concurrent Application Design

- Four tasks require specific time
- Execute all tasks in a sequence of one loop

```
loop
T1(200 ms)
T2(300 ms)
T3(100 ms)
T4(400 ms)
end loop
```

What is the required response time?

\[400 + 200 + 300 + 100 = 1000 \text{ ms}\]

Rationale for Concurrency CYCLIC EXECUTIVE

- Event E3, that must be handled by T3, occurs when T4 starts execution
**Rationale for Concurrency**

**ROUND-ROBIN**

- Execute each task in a separate loop
- Assign each task 50 ms time slice

```plaintext
loop T1(200 ms) end loop
loop T2(300 ms) end loop
loop T3(100 ms) end loop
loop T4(400 ms) end loop
```

Event E3, that must be handled by T3, occurs when T4 starts execution

What is the required response time?

The T3 response takes two time slices

\[
50 + 50 + 50 + 50 + 50 + 50 + 50 + 50 = 400 \text{ ms}
\]

**Rationale for Concurrency**

**NON-PREEMPTIVE PRIORITY**

- Schedule task Ti for execution when activated by event Ei
- Lower number task has a higher priority
- When event Ei occurs, complete currently executing task and schedule the Ti (according to its priority)

```plaintext
loop T1(200 ms) end loop
loop T2(300 ms) end loop
loop T3(100 ms) end loop
loop T4(400 ms) end loop
```

Event E3, that must be handled by T3, occurs when T4 starts execution

What is the required response time?

\[
400 + 100 = 500 \text{ ms}
\]
Schedule task Ti for execution when activated by event Ei
• Lower number task has a higher priority
• When event Ei occurs and currently executing task Tj (j > i), interrupt Tj and execute Ti

What is the required response time?

100 = 100 ms

Concurrently executed programs imply sharing of resources - they require synchronization and communication facilities
• Synchronization - we want to execute concurrent components of program “in step”
• Communication - we want to pass data between concurrent program components (but only at the appropriate time)
Concurrency - Potential Problems

- **Deadlock** - a task is blocked waiting for a resource that shall never be available (e.g. due to a circular wait of more than one tasks)
- **Livelock** - a task executes an idle loop waiting for a condition that will never happen (e.g., the task that is to set this condition is not doing so)
- **Non-reentrant Code** - the interrupted code may have corrupted data upon resuming its execution (due to overwriting data on subsequent code invocations)

Concurrency - Potential Problems (2)

- **Race condition** - different results are obtained depending upon which task will execute a specific instruction first
- **Mutual exclusion violation** - two tasks have access to shared data (or resource) at the same time which may lead to data corruption, operation on corrupted data, or resource contention and malfunction
- **Priority Inversion** - tasks of lower priority execute while task of higher priority waits for execution
  - (Can you think of a way this might happen?)
Ordering and Determinism

- **Sequential** programs are totally ordered - we can always determine which statement will be executed next (the steps of an algorithm are always performed in a *well defined* order).
- For **concurrent** programs the next statement to be executed is much harder to determine - it could be from any of the executing concurrently processes.
- The order of the activities involved in the execution of an algorithm can only be *partially defined*.

Example: Non-Determinism

- The order or execution of a set of concurrent processes cannot be predetermined.

P1

N := N-1

P2

N := N+1

Assuming N is initialized to 3
What is the final value of N?
Pick one: 1, 2, 3, 4, 5,...
Example: Non-Determinism (cont)

P1:
P11: fetch N
P12: decrease N by 1
P13: store N

P2:
P21: fetch N
P22: increase N by 1
P23: store N

<table>
<thead>
<tr>
<th>Case 1</th>
<th>p11</th>
<th>p12</th>
<th>p13</th>
<th>p21</th>
<th>p22</th>
<th>p23</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case 2</th>
<th>p11</th>
<th>p21</th>
<th>P12</th>
<th>p22</th>
<th>p13</th>
<th>p23</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Case 3</th>
<th>p11</th>
<th>p21</th>
<th>p22</th>
<th>p23</th>
<th>p12</th>
<th>p13</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Task Implementation
Operating System Level

- **As a process** - executing program with its data and context
  - a “heavy” separate executing programming construct
  - scalability and modularity
  - good protection via operating system
  - time penalty on process overhead
- **As a thread** - unit of work executing sequentially
  - a “light” programming construct executing within a process
  - fast creation and communication
  - common data area
  - sensitive to data corruption
**Process vs. Thread**

<table>
<thead>
<tr>
<th>Process vs. Thread</th>
<th>1:1</th>
<th>1:M</th>
</tr>
</thead>
<tbody>
<tr>
<td>single process / single thread</td>
<td>M:1</td>
<td>M:M</td>
</tr>
<tr>
<td>multiple process / single thread</td>
<td></td>
<td>multiple process / multiple thread</td>
</tr>
</tbody>
</table>

**Task Activation**

**Periodic Activation**
- defined time interval between two consecutive activation events
- must be completed before the next activation

```
begin loop
  { body of task }
  delay period
end
```

OR

```
begin loop
  wait for time point
  { body of task }
end
```

**Sporadic Activation**
- triggered by an asynchronous event
- must be completed within a defined time limit

```
begin loop
  wait for event
  { body of task }
end
```
Task States - a Simple Approach

- **RUNNING**: Executing with access to CPU
- **READY**: has everything it needs (but no CPU)
- **BLOCKED**: waiting for something (to be ready)

![Diagram showing task states]

Two Common Task Scheduling Mechanisms

- **Preemptive Priority** Scheduling
  - Kernel schedules the tasks according to task priority and scheduling options
  - Kernel preempts a low priority task to execute a higher priority task when the latter is ready

- **Round-Robin** Scheduling
  - A time-slice is defined with each task is assigned time processor for duration defined by the time slice
  - Upon time-slice expiration the next task is assigned to processor
Example: Preemptive Priority

- Context switch occurs when a higher priority task is ready to run

Example: Round-Robin with Time-Slicing

- To allow equal priority tasks to preempt each other, time slicing must be turned on:
  - `kernelTimeSlice(ticks)`
  - If `ticks = 0`, time slicing is turned off
- Priority scheduling always takes precedence
- Round-robin only applies to tasks of the same priority
- Priority-based rescheduling can happen any time at the pre-emption point
- Round-robin rescheduling can only happen at specific clock tick defined by the time slice
Mechanism by which an application reacts to multiple discrete real-world events creating the appearance of many threads of execution running concurrently

Kernel interleaves the task execution on the basis of a scheduling algorithm

All real-time kernels provide multitasking environment (e.g.: VxWorks wind)

The wind kernel is that part of VxWorks which directly manages tasks
Allocates the CPU to tasks according to the VxWorks scheduling algorithm (to be discussed)
Uses Task Control Blocks (TCBs) to keep track of tasks
  One per task
  Declared as WIND_TCB data structure in taskLib.h
  RTOS controls information about state, task priority, delay timer breakpoint list, error status, I/O redirections, etc.
  CPU context information: PC, CPU registers, FPU registers
Types of Context Switches

- Synchronous context switches occur because the executing task:
  - pends, delays, or suspends itself
  - makes a higher priority task ready to run
  - (less frequently) lowers its own priority, or exits

- Asynchronous context switches occur when an ISR:
  - makes a higher priority task ready to run
  - (less frequently) suspends the current task or lowers its priority

When does context switch occur?

- Under the control of kernel at specified preemption points (usually the real clock time tick):
  - when the task completes
  - when the task becomes pending, delayed, or suspended
  - when the task time slice expires

- Under the control of hardware (after the completion of currently executing instruction) as a response to an interrupt/exception
How does context switch happen?

• The created task is placed on the ready queue (in the priority order)
• The first task from the ready queue starts execution
• When the software context switch occurs, the next task from ready queue shall execute
• When the interrupt occurs, the ISR (interrupt service routine) executes in the hardware context
• Context switch time (µsec range) may be critical for a real-time system performance

Detailed Insight into Task States

• **NON-EXISTING** - not yet created (or terminated)
• **DORMANT** - created but not yet activated (or completed but not yet terminated)
• **RUNNING** - executing with access to processor
• **INTERRUPTED** - when system responds to hardware interrupt
• **READY** - ready to execute
• **PENDING or BLOCKED** - waiting for a resource which is not available
• **DELAYED or WAITING** - waiting on a delay or timer to expire
• **SUSPENDED** - stopped for debugging or other purpose (not able to execute - *may be used with READY, PENDING, or DELAYED*)
Kernel maintains the current state of each task using queues for:
Delayed; Ready; Pending; Suspended

Many vs. Few Tasks

• Benefits of many tasks:
  • faster response time
  • cohesion and modularity
  • Encapsulation

• Benefits of few tasks:
  • data sharing
  • easy communication and synchronization
  • less memory
  • less switching
Distinction between:
- **creation** - elaboration of a declaration of code segment,
- **activation** - explicit or implicit operation resulting in the actual execution of a task

Activation can be by:
- continuing execution of the activating task (UNIX fork, Ada task)
- executing specific list of statements concurrently (Occam cobegin-coend)

Process termination can be **explicit** (UNIX kill, Ada abort) or **implicit** via completion

In some cases the completed task can still not terminate until (because) it has active dependent tasks

Task Actions

- Task can:
  - spawn another task
  - delete another task
  - protect itself from other tasks
  - delete itself (exit)
  - suspend another task
  - resume another
  - restart another task
  - delay another task
Creating VxWorks Task

```c
int taskSpawn (name, priority, options, stackSize, entryPt, arg1, ..., arg10)
```

- **name**: Task name, if `NULL` gives a default name
- **priority**: Task priority 0-255 (0 highest)
- **options**: Task options e.g. `VX_UNBREAKABLE`
- **stackSize**: Size of stack to be allocated in bytes
- **entryPt**: Address of code to start executing (initial PC)
- **arg1, ..., arg10**: Up to 10 arguments to entry-point routine

- Returns a task id or `ERROR` if unsuccessful

**Example:**
```
newTid = taskSpawn("tMyTask", 150, 0, 20000, myRoutine, arg1, arg2, 0,0,0,0,0,0,0,0)
```

VxWorks Task Control

- **Task Creation and Activation**
  - `taskSpawn/taskCreate/taskActivate`

- **Task Names and ID**
  - `taskName: get name from ID`
  - `taskIdTold: get tid from name`
  - `taskIdSelf: gets ID of calling task`
  - `taskIdVerify: verify that ID is valid`

- **TaskOptions**
  - `taskOptionsGet/taskOptionsSet: get or set task options`

- **Task Information**
  - `taskIdListGet/taskInfoGet`
  - `taskPriorityGet`
  - `taskRegsGet/taskRegsSet`
  - `taskIsReady/taskIsSuspended`
VxWorks Task Control (Cont.)

- Control round-robin scheduling
  - `kernelTimeSlice(N)`
  - N is the slice duration - number of clock ticks (0 - no time slicing)
- Change priority of task
  - `taskPrioritySet(N)`. N is the new priority (0-255)
- Disable/enable task rescheduling
  - `taskLock/taskUnlock`. Function to disable/enable context switching
- Disable/enable interrupts
  - `intLock/intUnlock`. Function to disable/enable interrupts

VxWorks Task Control (Cont.)

- Task Deletion (BIG RED FLAG)
  - `taskDelete(tid)`
  - `taskDelete(0) ??`
  - `exit`. (Same as taskDelete (0)) but code is still stored in TCB
- Task Control
  - `taskSuspend/taskResume`
  - `taskDelay(number of ticks)`
  - `taskRestart`. Task is reinitialized with original arguments and tid
VxWorks Task ID’s and Names

- Uniquely assigned by kernel when task is created
- May be reused after task exits
- A task ID of zero refers to task making call (self)
- Relevant taskLib routines:
  - `taskIdSelf( )` Get ID of calling task
  - `taskIdListGet( )` Array with ID’s of all existing tasks
  - `taskIdVerify( )` Verify a task ID is valid
- Task names provided for human convenience
  - Typically used only from the shell (during development)
  - Within programs, use task IDs
- By convention, task names start with a “t” followed by an integer

Task Priorities and Stack

- Range from 0 (highest) to 255 (lowest)
- Timing requirements rather than hazy ideas about task importance should govern priorities (theory exists)
- One can manipulate priorities dynamically (not recommended)
  - `taskPriorityGet (tid, &priority)`
  - `taskPrioritySet (tid, priority)`
- Task stack allocated from system memory pool when task created
  - Task stack has fixed size after creation
  - The kernel reserves space from the stack, making the stack space actually available slightly less than the stack space requested
  - Exceeding stack size (“stack crash”) causes unpredictable system behavior
VxWorks Task Creation and Options

• During time critical code, task creation can be unacceptably time consuming
• To reduce creation time, a task can be spawned with the VX_NO_STACK_FILL option bit set
• Alternatively, spawn a task at system start-up which blocks immediately, and waits to be made ready when needed
• Can be bitwise or’ed together when the task is created:
  • VX_FP_TASK Add floating point support
  • VX_NO_STACK_FILL Don’t fill stack with 0xee
  • VX_UNBREAKABLE Disable breakpoints
  • VX_DEALLOC_STACK Deallocate stack and TCB at exit

• Use taskOptionsGet( ) to inquire about a tasks options
• Use taskOptionsSet( ) to set or unset a task option

VxWorks Task Deletion and Restart

• taskDelete (tid). Deleting specified task
  • De-allocating the TCB and stack: exit (code)
  • does not return global resources, i.e. allocated memory and open file descriptors
  • the same as taskDelete(0) except code is stored in the tasks TCB before the task is terminated
  • can be examined postmortem if VX_DEALLOC_STACK is off

• taskRestart (tid). Task is reinitialized with original arguments and tid
  • The stack is reset and cleared
  • Global resources used by the task are not reset
  • Global and static variables do not get reset
  • file descriptors will be left open
  • any memory allocated by the task will not be freed at restart
**VxWorks Task Suspend/Resume and Delay**

- **taskSuspend (tid).** Makes task ineligible to execute (safest to have a task suspend itself)

- **taskResume (tid)** removes suspension
  - **taskSuspend()** and **taskResume()** used for debugging and development purposes

- To delay a task for a specified number of system clock ticks: STATUS taskDelay (tics)
  - To poll every 1/5 second:
    ```
    FOREVER
    {
    taskDelay (sysCkRateGet() / 5)
    ...
    }
    ```

- Only accurate if clock rate is a multiple of five ticks/second

- Use **sysCkRateSet()** to change the clock rate

---

**Lab Work**

- **Task Basics**
- **Task Control**
- **Task Delay**
Lab 1: Due by class time on Thursday 09/25
Reading Assignment: Chapter 6 (Semaphores)