ECEN 3753: Real-Time Operating Systems

Tasks

What Is A Task

- Sub-unit of program/application that can execute somewhat independently
- When using an RTOS, an application likely consists of more than one task.
- Each task may:
    - Generate data
    - Process data
    - Control hardware

Why Multi-Tasking

- In a typical embedded system, a program must wait for a physical operation to complete
  - Mechanical operations are *slow* relative to CPU execution.
- Polling loops are sub-optimal
  - Power: 100% Duty Cycle–max consumed
  - Latency: we can only benefit from operation completion at the point that we check it.
  - Coupling: If the CPU needs to do other work while waiting, this needs to be managed inside the polling loop.
- Using a multi-Tasking OS, if a program needs to wait for an operation to complete, a responsible Task can **block**.
  - A blocked Task must then be awakened by an interrupt or some signal from another Task.
  - Other Tasks can simply execute on the CPU while one task is waiting.

Compare to "multi-**t**asking", which could be done with a superloop, interrupt-driven quasi-concurrency, etc.

Program Execution: Processes, Threads, and Tasks

- Process
  - Heavyweight (relatively) unit of program execution.
  - Typically used in virtual memory systems and paged in to switch between processes.
  - Owns its own memory space, including heap.
  - Windows '.exe' (after it has started running)
- Thread
  - Lightweight (relatively) unit of program execution.
  - A process may spawn one or more threads.
  - Threads share the memory space (including global data) of their parent process, but have separate stacks for program execution.
  - TEND to be aligned more to concurrent subtasks
- Task
  - Lightweight unit of program execution, very similar to a thread.
  - TEND to align with less-concurrent subtasks in real-time, but allowing decoupling

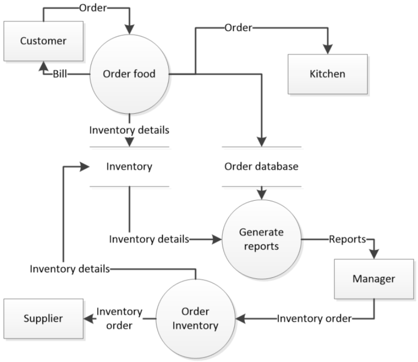
What Needs to Happen–Pulling our thoughts together

How We Can View, Model, and Document a Multi-Task Process

- Our Textbook shows Software Design Models, System Run Time Models, and some Task Development by examining System functions and sub-functions in an attempt to structure software (Chapter 1).
- Another method that can be useful is to work from use-cases through blocking events and messages, to diagram an algorithm (step 3 of the 7-Steps-to-Coding method), utilizing some tools that may be especially effective for this.
    - Data Flow Diagram (shows data between processing entities)
    - State Machine Diagram (shows (task) states, what transitions are caused by and effects)
    - Sequence Diagram (good to formally capture your use cases)

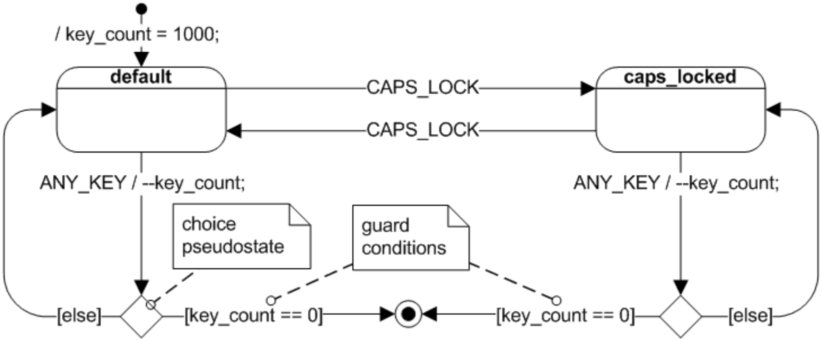
Data Flow Diagram

This tool can be especially useful in seeing the boundaries between tasks, by examining minimized data transfer boundaries, and action boundaries. Additionally, it may help you discriminate shared data stores from simple I/O data.



State Machine Diagram

This tool can be especially useful in seeing and documenting the changing behavior within a task, in reaction to external events or information.

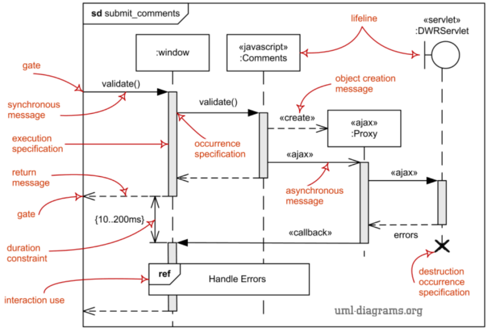


<image from https://en.wikipedia.org/wiki/UML_state_machine>

Extra-Task behavior and info passing will be covered in a few weeks, in Inter-Process Comms and Shared Data.

Sequence Diagram

This tool can be used to formally document well-defined interactions between different entities. It can be used to help clarify use cases, but due to its assumption of entities (e.g. tasks) it may be premature to use it when the task boundaries are not yet clear.



<image from https://www.uml-diagrams.org/sequence-diagrams.html>

Back to Some Nuts-and-Bolts. . .

Let's presume that we'll come up with the right algorithms, task divisions/messaging/etc.

What are the essentials that an RTOS juggles to make that all work?

Resources That A Program Needs To Execute

- CPU to run on
  - Only one task may run on a physical CPU at a time.
- Instructions to fetch
  - Program counter (PC) points to the current instruction being executed
- Mechanism to maintain the current state of the program we are executing, including:
  - What code is currently running
  - Where does code execution return to
    - After a function call
    - After an interrupt
    - After switching to a new task (context switch)
  - Local data
    - Registers
    - Stack
    - Thread-local storage

Program State: The Task Stack

- Each task has its own stack (All the same size?)
- Interrupt handlers often have their own stack
- When a task executes, information on the stack provides information about what code is executing at any point in time
  - Local variables
  - Parameters that have been passed from one function to another
  - Function return addresses
- On ARM, stacks typically grow... Aside: What does "upward"/"downward" mean to you?

Program State: The Task Stack

- Each task has its own stack (All the same size?)
- Interrupt handlers often have their own stack
- When a task executes, information on the stack provides information about what code is executing at any point in time
  - Local variables
  - Parameters that have been passed from one function to another
  - Function return addresses
- On ARM, stacks typically grow... Aside: What does "upward"/"downward" mean to you?
  - toward address 0

The use of the terms **Upward/Downward** can fuel holy wars. Some insist on drawing memory with address zero at the top, some at the bottom. We'll try to be agnostic, and use unambiguous terminology.

The ARM Application Binary Interface (ABI)

- Parameters typically in registers R0..R3, possibly on stack for longer parameter lists or large data structures
- Function return value in R0
- Stack pointer (SP = R13)
- Link Register (LR = R14) (return address)
  - Often saved on the stack on function entry (to allow for nested calls), popped into PC on return.
- Program Counter (PC = R15)
- R4..R11 are generally volatile and should be preserved (on the stack) across function calls if they are needed locally.
- Compiler may optimize around ABI
  - Compile time computations (parameters + return values)
  - Inline functions

The ARM Application Binary Interface (ABI)

- Parameters typically in registers R0..R3, possibly on stack for longer parameter lists or large data structures
- Function return value in R0
- Stack pointer (SP = R13)
- Link Register (LR = R14) (return address)
  - Often saved on the stack on function entry (to allow for nested calls), popped into PC on return.
- Program Counter (PC = R15)
- R4..R11 are generally volatile and should be preserved (on the stack) across function calls if they are needed locally.
- Compiler may optimize around ABI
  - Compile time computations (parameters + return values)
  - Inline functions

Why review calls/params/returns now?

The ARM Application Binary Interface (ABI)

- Parameters typically in registers R0..R3, possibly on stack for longer parameter lists or large data structures
- Function return value in R0
- Stack pointer (SP = R13)
- Link Register (LR = R14) (return address)
  - Often saved on the stack on function entry (to allow for nested calls), popped into PC on return.
- Program Counter (PC = R15)
- R4..R11 are generally volatile and should be preserved (on the stack) across function calls if they are needed locally.
- Compiler may optimize around ABI
  - Compile time computations (parameters + return values)
  - Inline functions

Why review calls/params/returns now?

Looking for parallels as we look to perform task switching, instead of simply considering hierarchical calls.

Anatomy of a Function: Parameter Passing [32b ARM]

- The first four parameters to a function are usually passed in R0..R3
  - 64-bit values can be passed in a pair of registers (R0..R1, or R2..R3)
- Subsequent parameters (after the first 4) are passed on the stack
- For large data structures:
  - Pointer parameters are always of a size_t that fits in a CPU register, and may be passed in registers (regardless of what data is being pointed to)
  - If passed by value, a large data structure is passed on the stack.
    - Compiler may generate calls to memcpy()–Star Wars: "wait for it..."
  - Always more efficient to pass large data by pointer rather than by value
    - Efficiency aside, do you see any risks with a pointer to a const?

Anatomy of a Function: Entry and Exit

- When we enter a function
  - The following registers may be saved on the stack to preserve the current CPU context
    - LR (function return address)
    - R4..R11 (scratch registers)
  - Params passed in R0..R3 are often re-saved in another register or on the stack before they are needed to make a nested call [1]
- When we exit a function
  - The function return value is copied into R0 (or R0..R1)
  - Values that were originally pushed onto the stack are restored, including the original LR which will be the PC to which execution resumes.

[1] Params are passed in same position [R0..R3] by subsequent nested calls require fewer register/stack copies.

Anatomy of a Function: Stack Frames

- If a function declares local data, it may be stored in a stack frame
- When entering a function, the compiler is free to reserve an amount of memory at the top of the stack for local/scratch data.
  - Large local data structures–again: ick.
  - With optimized ARM code, a stack frame is often unnecessary, since local data can also be stored in registers (R0..R11)
- If a stack frame is allocated, the register used to point into the frame[2] will generally be saved on the stack at function entry (and restored on exit)
- The memory allocated for a stack frame will generally be reclaimed when a function returns.

[2] there is not necessarily a Stack Frame Pointer–compilers are good at math, so SP is generally sufficient

Now for some Parallels (ISR-FunctionCall-TaskSwitch)

Anatomy of a Task Switch

- Each task maintains its own stack
- CPU registers are a shared resource, but the values in these registers at any point in time are specific to only the currently running task.
- When the Micrium OS wants to switch to a different task it needs to do the following (os_cpu_a.s):
  - Allocate stack space to save all the CPU registers for the currently running task
  - Save SP in the current task's control block, so that it can be restored later.
  - Save CPU registers into this reserved stack space.
  - Find the control block for the task we are about to switch in.
  - Load SP from the new task's control block.
  - Load any remaining CPU registers from the new tasks's stack.
  - Set PC to the new task (SWITCH!)

The terms OUT/IN may be replaced with FROM/TO in some environments. The terminology has not converged

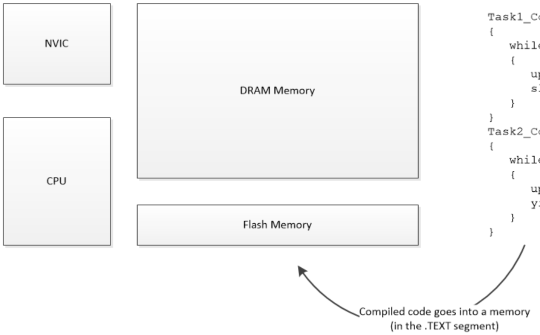
Creating a Task: What information is Needed

- Amount of memory to reserve for task stack
  - Each task owns its stack space
- Task name
- Entry point[3]
  - void entryFunction(void*)
- Task priority
- Task control block
  - Memory that the OS will use internally to manage/store task state
- Initialization options
  - Default time quanta (round-robin scheduling)
  - Is stack protection enabled
  - Is thread local storage enabled
  - etc.

[3] arg must be valid and accessible when the task starts (i.e. heap or global)

POV Switch: OS Internals to OS usage

Where does the Task Code go?



```
Task1_Code
{
   while (1)
   {
      updateOutput1(getAdcValue());
      sleep(1);
   }
}
Task2_Code
{
   while (1)
   {
      updateOutput2(getAdcValue());
      yield();
   }
}
```

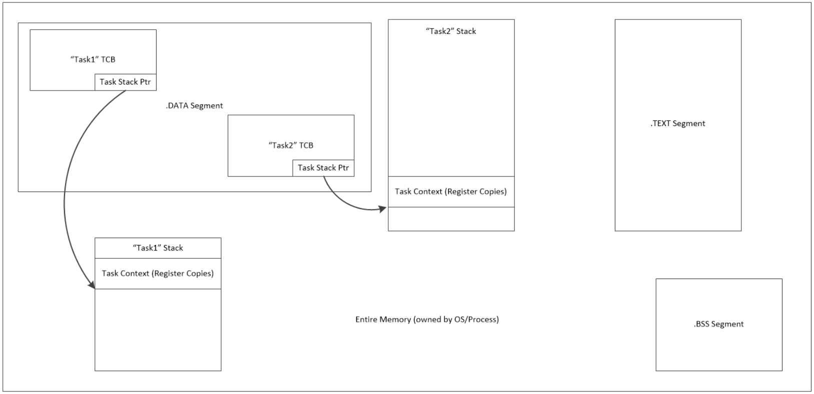
main() Constructs Tasks

OS Calls are made to set up Tasks:

```
OSInit()
OSTaskCreate("Task1", Task1_Code, ...);
OSTaskCreate("Task2", Task2_Code, ...);
   // or could create in Task1_Code, before while(1)
OSTaskStart(...);
```

Memory Breakdown

Where do the Task constructs go?



The TCB is defined by the user code (global variable) and passed in to micrium. See

https://doc.micrium.com/display/osiiidoc/OSTaskCreate

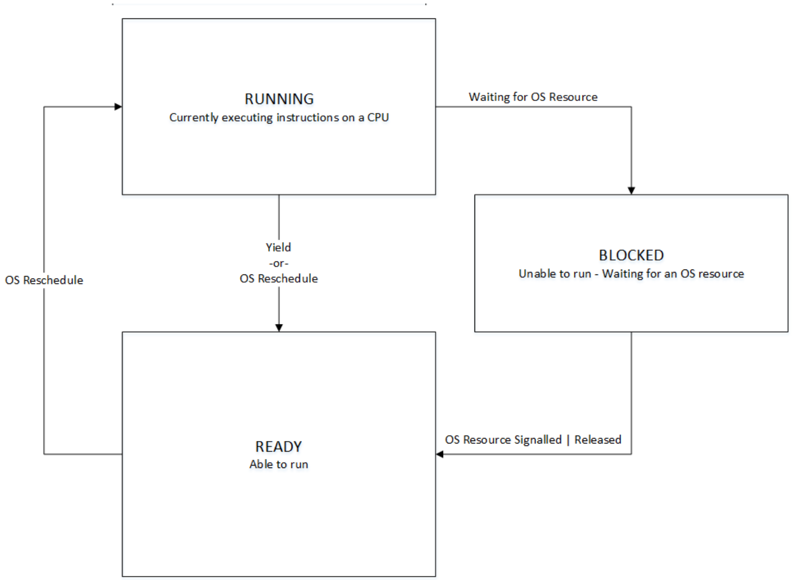
Starting the Micrium OS (from Main)

- OSInit(&err)
    - Creates OS system tasks
        - Tick Task
        - Idle Task
        - Timer Task
        - Statistics Task (optional)
- OSTaskCreate(. . . , &err)
    - Create your application main task
    - Your task can create as many additional tasks as it needs
- OSStart(&err)
    - Starts the OS
    - Schedules the first task to run
    - Does not return

Inside Your Task Entry Point

- Initialize data needed by your task
- Infinite loop
    - Do some work. . .
    - Give up CPU (block or yield)

Overview of Task States



RUNNING
Currently executing instructions on a CPU
Waiting for OS Resource
BLOCKED
Unable to run - Waiting for an OS resource
Yield
-or-
OS Reschedule
OS Reschedule
READY
Able to run
OS Resource Signalled | Released

Task State: RUNNING

- For single core systems, only one task at a time may run on a CPU.
- Multicore systems can run one task on each CPU simultaneously.
  - CPU affinity describes which task(s) prefer to run on which CPU
- A running task may yield the CPU and possibly allow another task to run
- A running task may block, in which case we always allow another task to run.

Task State: BLOCKED

- Sometimes referred to as a “suspended” or “pending” state.
- Waiting for an OS event, mutex, semaphore, timer, etc.
- Whenever the OS resource we are blocked on is released, the blocked task is made eligible again via the ready list.
- Unblocking a task does not make this task run automatically.

Task State: READY

- All tasks in the system that are not blocked might be scheduled to run at any point in time.
- The OS maintains a list of tasks that **could** be run at any point in time in a sorted ready list.
- Each time an OS kernel call is made, the OS might reschedule tasks.
- When the kernel determines that a task switch is needed, the head of the ready list is popped and becomes the currently running task.

What About Tasks that are Interrupted

- Only the currently running task may be interrupted.
- When an interrupt handler returns, it can either:
  - Return to the running task directly (no state transitions)
  - Return via the OS scheduler with a different task running.
- Inside an interrupt handler:
  - We may -not- call any OS functions that block or yield.
  - We may call OS functions that signal a task to become unblocked.
    - Task state transitions only occur after all nested interrupts return.
    - Only need to reschedule once.