ECE 3574: Applied Software Design:

Memory Management, Event-driven Code

Today we are going to look at two simple approaches to replacing the C/C++ library heap allocator and then introduce event driven programming in the context of embedded systems.

- Stacks
- Pools
- Interrupt handlers
- Event-driven code
- Introduction to FreeRTOS (if time)

From last time: alternatives to dynamic memory allocation

The real issue with dynamic allocation is different sized objects. We can get predictability using:

- stacks: pushes/pops happen at one end so object size does not matter. This mimics the way automatic allocation works.
- pool: allocate fixed size blocks that can be recycled without causing holes. This is an allocator for single sized objects.

We use specialized algorithms for allocation.

Stack example

Lets look at a simple implementation of a stack based allocator.
See `stack.h`, and `stack_ex.cpp`.

Pool example

Lets look at a simple implementation of a pool based allocator.
See `ring_buffer.h`, `pool.h`, and `pool_ex.cpp`.

Embedded programming requires handling a wide variety of time priorities

- control systems have hard deadlines - you must read ones or more sensor values and update one or more output values every X ms.
- human I/O, e.g. keypad and LCD, has softer deadlines - you show the character corresponding to the last key-press on the LCD within Y $\sim$ 500 ms
- remote I/O, say logging to a serial port or responding to a request via an internal http server might have delay times in the seconds.

How do you balance these different tasks?

Solution 1: serial execution

```
for( ;; ){
    read_update_control(); // task 1
    read_update_keypad(); // task 2
    respond_http(); // task 3
}
```

In this solution the time for tasks 1+2+3 must be less than the tightest deadline, that of task 1.

Solution 2: events

```
for( ;; ){
    read_update_control(); // task 1
    read_keypad_generate_event(); // task 2
    check_http_generate_event(); // task 3
    process_events_if_time();
}
```

Here task 2 and 3 do minimal work related to IO and generate events on some kind which are queued. These are executed if time is available.

There are two basic approaches to IO in embedded systems

- polling, spinning in a loop, checking the status of a port
- interrupts, code that gets called automatically when in interrupt occurs (interrupt service routine or ISR)

Both can be used to generate events.

Event handling

Event handling should be deterministic and kept short. This generally leads to state machines or event handlers.

- the state machine does minimal work then sets the next state
- an event handler does minimal work and generates another event

This requires chunking the functionality into deterministic pieces.

Example: state machine

Using a state machine to debounce a keypad using polling while updating a control loop.
See `state_ex.cpp`.

Example: event handlers

The same example using an event design.
See `event_ex.cpp`.

It can be tedious to chunck functions into deterministic pieces

Another approach is to preempt running code using a (minimal) OS. These are called *real-time* operating systems (RTOS).
Recall, preemptive multi-tasking is the dominant form of operating systems:

- The interruption and change of executing code is called a *context switch*.
- The RTOS kernel schedules code according to a priority-based scheme.

Operation of an RTOS scheduler

Any concurrently running code is a *task*.
The OS keeps two lists/queues of tasks:

- running: tasks sorted in priority order (heap) that can execute
- waiting/pending: tasks that are waiting for an interrupt or timer counter

Timers and counters are used for guaranteeing timing. For example a control task waits on a timer interrupt that occurs every X ms. The ISR moves the task from waiting to running. Since it has the highest priority it gets chosen to run next.

Context switches

Once a task has been selected to run (scheduled) the kernel needs to:

- restore the registers to when the task was last run
- restore the stack pointer for the task
- adjust the program counter to the instruction for the task that was to be executed

The latter is done using a `ret` instruction (return from function) in a cooperative kernel, and a `reti` instruction (return from interrupt) for a preemptive kernel.
In both cases each task has its own stack and the kernel stores the context for each task at the top of its stack, keeping the per-task stack pointer in kernel memory.

Pseudo-code for yield

```
save the PC, SP and registers for the current task
select next task to run from the priority queue
restore the SP and registers from the new task's stack
ret to the next instruction in the scheduled task
```

How does yield know which task is running?
How does it know which task to return to?

Example: function call on x86_64

On x86 the convention is:

- after call instruction: `%rip` points at first instruction of function, `%rsp+4` points at first argument, `%rsp` points at return address
- after ret instruction: `%rip` contains return address, `%rsp` points at arguments pushed by caller

To switch the task yield uses assembly to change these registers. See `main.cpp` and `main.s`.

A complication here is that most processors support two modes of operation

- kernel mode, any valid instruction can be executed
- user mode, the instructions are restricted, e.g. in/out for reading from ports

This is how an operating system supports privileges.
To perform a restricted operation a user mode program does a *system call*, which raises a trap exception in the CPU, which switches the CPU to kernel mode and jumps into code previously set by the kernel in the exception table. The kernel executes the restricted instructions necessary to complete the operation and switches the CPU back to user mode before returning.
This is less common in embedded systems since this is rather inefficient and most tasks require direct access to IO.

A preemptive context switch is similar but it is caused by a timer interrupt.

The kernel sets up an ISR. When the interrupt fires, the CPU looks up what ISR to run, saves the return address (PC) and jumps to the ISR.
The ISR pseudo-code

```
save the SP and registers for the current task
select next task to run from the priority queue
restore the SP and registers from the new task's stack
reti
```

Note the PC is saved automatically by the interrupt sequence and restored automatically after the ISR returns.
(I am ignoring nested interrupts here)

Example: precise temperature control with an RTOS

Suppose there is an embedded system with a keypad, LCD, a temperature sensor, and current control to a heater coil.

- The user can monitor and program a cycle of precise temperatures through the keypad/LCD.
- The current must be updated every 20ms.
- Another cycle can be programmed while one is running.
- An http server responds to requests rendering an html page with the current stats and cycle programming.

The control task is given a priority of 3. The user interface (keypad/LCD) is given a priority of 2. Other tasks such as http server get a priority of 1.

Example: FreeRTOS

FreeRTOS is a popular RTOS for embedded systems.

- small, fits into 6-12k of ROM
- preemptive or cooperative scheduling
- provides mutexes and semaphores
- provides a message passing implementation
- can uses tasks or co-routines

The core implementation is just three source files. The API is C.

FreeRTOS tasks

In FreeRTOS you implement *tasks*, functions that never return

```
void vATaskFunction( void *pvParameters )
{
    for( ;; )
    {
     // Task application code here.
    }
}
```

In main you call `xTaskCreate` for the tasks with a priority parameter and then start the scheduler.

Next Actions and Reminders

That's it for the semester's content.

- Next time, we review the course and revisit the learning objectives
- Project 3- Final due Tuesday at 11:59 PM. Be sure to tag and push by then.

Please, be sure to fill out the SPOT survey!