Last Time

- Debugging
  - It’s a science – use experiments to refine hypotheses about bugs
  - It’s an art – creating effective hypotheses and experiments and trying them in the right order requires great intuition

Today

- Advanced threads
  - Thread example
  - Implementation review
  - Design issues
  - Performance metrics
  - Thread variations
- Example code from Ethernut RTOS

What’s an RTOS?

- Real-Time Operating System
  - Implication is that it can be used to build real-time systems
- Provides:
  - Threads
  - Real-time scheduler
  - Synchronization primitives
  - Boot code
  - Device drivers
- Might provide:
  - Memory protection
  - Virtual memory
- Is WinCE an RTOS? Embedded Linux?

Thread Example

- **We want code to do this:**
  1. Turn on the wireless network at time t_0
  2. Wait until time is $t_0 + t_{awake}$
  3. If communication has not completed, wait until it has completed or else time is $t_0 + t_{awake} + t_{wait\_max}$
  4. Turn off radio
  5. Go back to step 1

Threaded vs. Non-Threaded

```
void radio_wake_thread () {
  while (1) {
    radio_on();
    timer_set (&timer, T_AWAKE);
    wait_for_timer (&timer);
    timer_set (&timer, T_SLEEP);

    if (!communication_complete()) {
      timer_set (&wait_timer, T_WAIT_MAX);
      wait_cond (communication_complete() ||
                 timer_expired (&wait_timer));
    }
    radio_off();
    wait_for_timer (&timer);
  }
}
```

```
enum { ON, WAITING, OFF } state;

void radio_wake_event_handler () {
  switch (state) {
    case ON:
      if (expired(&timer)) {
        set_timer (&timer, T_SLEEP);
        if (!communication_complete) {
          state = WAITING;
          set_timer (&wait_timer,
                     T_MAX_WAIT);
        } else {
          turn_off_radio();
          state = OFF;
      }}
      break;
    case WAITING:
      if (communication_complete() ||
          timer_expired (&wait_timer)) {
        state = OFF;
        radio_off();
      }
      break;
    ...
```

Blocking

- **Blocking**
  - Ability for a thread to sleep awaiting some event
    - Like what?
  - Fundamental service provided by an RTOS

- **How does blocking work?**
  1. Thread calls a function provided by the RTOS
  2. RTOS decides to block the thread
  3. RTOS saves the thread's context
  4. RTOS makes a scheduling decision
  5. RTOS loads the context of a different thread and runs it

- **When does a blocked thread wake up?**

More Blocking

- When does a blocked thread wake up?
  - When some predetermined condition becomes true
  - Disk block available, network communication needed, timer expired, etc.
  - Often interrupt handlers unblock threads
- Why is blocking good?
  - Preserves the contents of the stack and registers
  - Upon waking up, thread can just continue to execute
- Can you get by without blocking?
  - Yes – but code tends to become very cluttered with state machines

Preemption

- When does the RTOS make scheduling decisions?
  - Non-preemptive RTOS: Only when a thread blocks or exits
  - Preemptive RTOS: every time a thread wakes up or changes priority
- Advantage of preemption: Threads can respond more rapidly to events
  - No need to wait for whatever thread is running to reach a blocking point
- Even preemptive threads sometimes have to wait
  - For example when interrupts are disabled, preemption is disabled too

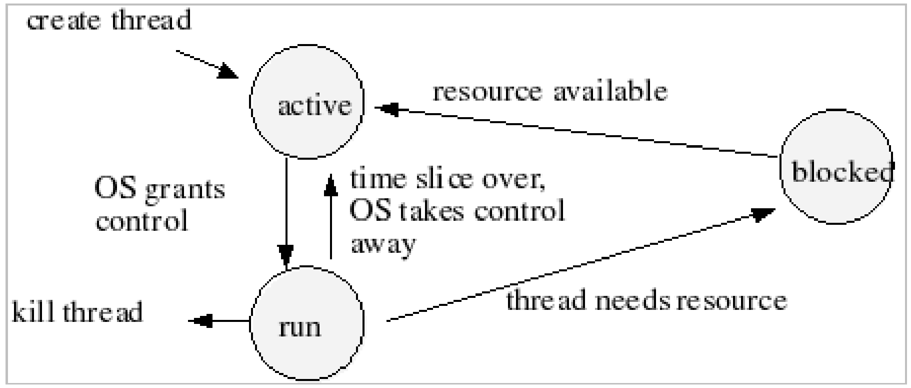
More Preemption

- Preemption and blocking are orthogonal
  - No blocking, no preemption – main loop style
  - Blocking, no preemption – non-preemptive RTOS
    - Also MacOS < 10
  - No blocking, preemption – interrupt-driven system
  - Blocking, preemption – preemptive RTOS

Thread Implementation

- TCB – thread control block
  - One per thread
  - A struct that stores:
    - Saved registers including PC and SP
    - Current thread state
    - All-threads link field
    - Ready-list / block-list link field
- Stack
  - Dedicated block of RAM per thread

Thread States



- Thread invariants
  - At most one running thread
    - If there's an idle thread then exactly one running thread
  - Every thread is on the "all thread" list
  - State-based:
    - Running thread
    - Blocked thread → On one blocked list
    - Active thread → On one ready list

Ethernut TCB

```
struct _NUTTHREADINFO {
    NUTTHREADINFO *volatile td_next;      /* Linked list of all threads. */
    NUTTHREADINFO *td_qnxt;         /* Linked list of all queued thread. */
    u_char td_name[9];              /* Name of this thread. */
    u_char td_state;                /* Operating state. One of TDS_ */
    uptr_t td_sp;                   /* Stack pointer. */
    u_char td_priority;             /* Priority level. 0 is highest priority. */
    u_char *td_memory;              /* Pointer to heap memory used for stack. */
    HANDLE td_timer;                /* Event timer. */
    HANDLE td_queue;                /* Root entry of the waiting queue. */
};

#define TDS_TERM        0   /* Thread has exited. */
#define TDS_RUNNING     1   /* Thread is running. */
#define TDS_READY       2   /* Thread is ready to run. */
#define TDS_SLEEP       3   /* Thread is sleeping. */
```

Scheduler

- **Makes a decision when:**
  - Thread blocks
  - Thread wakes up (or is newly created)
  - Time slice expires
  - Thread priority changes
- **How does the scheduler make these decisions?**
  - Typical RTOS: Priorities
  - Typical GPOS: Complicated algorithm
  - There are many other possibilities

```
u_char NutThreadSetPriority(u_char level) {
    u_char last = runningThread->td_priority;
    /* Remove the thread from the run queue and re-insert it with a new
     * priority, if this new priority level is below 255. A priotity of
     * 255 will kill the thread. */

    NutThreadRemoveQueue(runningThread, &runQueue);
    runningThread->td_priority = level;
    if (level < 255)
       NutThreadAddPriQueue(runningThread, (NUTTHREADINFO **) & runQueue);
    else
      NutThreadKill();

    /* Are we still on top of the queue? If yes, then change our status
     * back to running, otherwise do a context switch. */
   if (runningThread == runQueue) {
        runningThread->td_state = TDS_RUNNING;
    } else {
        runningThread->td_state = TDS_READY;
        NutEnterCritical();
        NutThreadSwitch();
        NutExitCritical();
    }
    return last;
}
```

Dispatcher

- Low-level part of the RTOS
- Basic functionality:
  - Save state of currently running thread
    - Important not to destroy register values in the process!
  - Restore state of newly running thread

- What if there's no new thread to run?
  - Usually there's an idle thread that is always ready to run
  - In modern systems the idle thread probably just puts the processor to sleep

Ethernut ARM Context

```
typedef struct {
    u_long csf_cpsr;
    u_long csf_r4;
    u_long csf_r5;
    u_long csf_r6;
    u_long csf_r7;
    u_long csf_r8;
    u_long csf_r9;
    u_long csf_r10;
    u_long csf_r11;          /* AKA fp */
    u_long csf_lr;
} SWITCHFRAME;
```

```
void NutThreadSwitch(void) attribute ((naked))
{
    /* Save CPU context. */
    asm volatile (              /* */
                "stmfd   sp!, {r4-r11, lr}"    /* Save registers. */
                 "mrs     r4, cpsr"     /* Save status. */
                 "stmfd   sp!, {r4}"    /* */
                 "str     sp, %0"   /* Save stack pointer. */
                 ::"m" (runningThread->td_sp) );

   /* Select thread on top of the run queue. */
    runningThread = runQueue;
    runningThread->td_state = TDS_RUNNING;

    /* Restore context. */
    __asm__ __volatile__(       /* */
                        "@ Load context"        /* */
                        "ldr     sp, %0"        /* Restore stack pointer. */
                        "ldmfd   sp!, {r4}"     /* Get saved status... */
                        "bic     r4, r4, #0xC0"      /* ...enable interrupts */
                        "msr     spsr, r4"      /* ...and save in spsr. */
                        "ldmfd   sp!, {r4-r11, lr}"     /* Restore registers. */
                        "movs    pc, lr"    /* Restore status and return. */
                        ::"m"(runningThread->td_sp) );
}
```

Thread Correctness

- Threaded software can be hard to understand
  - Like interrupts, threads add interleavings
- To stop the scheduler from interleaving two threads: use proper locking
  - Any time two threads share a data structure, access to the data structure needs to be protected by a lock

Thread Interaction Primitives

- Locks (a.k.a. mutexes)
  - Allow one thread at a time into critical section
  - Block other threads until exit
- FIFO queue (a.k.a. mailbox)
  - Threads read from and write to queue
  - Read from empty queue blocks
  - Write to empty queue blocks
- Message passing
  - Sending thread blocks until receiving thread has the message
  - Similar to mailbox with queue size = 0

Mixing Threads and Interrupts

- **Problem:**
  - Thread locks do not protect against interrupts
- **Solution 1:**
  - Mutex disables interrupts as part of taking a lock
  - What happens when a thread blocks inside a mutex?
- **Solution 2:**
  - Up to the user to disable interrupts in addition to taking a mutex

Thread Design Issues 1

- Static threads:
  - All threads created at compile time
- Dynamic threads:
  - System supports a "create new thread" and "exit thread" calls
- Tradeoffs – dynamic threads are:
  - More flexible and user-friendly
  - Not possible to implement without a heap
  - A tiny bit less efficient
  - Much harder to verify / validate

Thread Design Issues 2

- **Can threads be asynchronously killed?**
  - Alternative: Threads must exit on their own
- **Tradeoffs – asynchronous termination:**
  - Is sometimes very convenient
  - Raises a difficult question – What if killed thread is in a critical section?
    - Kill it anyway → Data structure corruption
    - Wait for it to exit → Defeats the purpose of immediate termination
  - Why do Windows and Linux processes not have this problem?

Thread Design Issues 3

- Are multiple threads at the same priority permitted?
- Tradeoffs – multiple same-priority threads:
  - Can be convenient
  - Makes data structures a bit more complex and less efficient
  - Requires a secondary scheduling policy
    - Round-robin
    - FIFO

Thread Design Issue 4

- How to determine thread stack sizes?
  - Use same methods as for non-threaded systems
  - Need to know how interrupts and stacks interact
- Possibilities
  1. Interrupts use the current thread stack
  2. Interrupts use a special system stack

Thread Performance Metrics

- Thread dispatch latency
  - Average care and worst case
- System call latency
  - Average case and worst case
- Context switch overhead
- RAM overhead
  - More or less reduces to heap manager overhead

Thread Variation 1

- **Protothreads are stackless**
- **Can block, but…**
  - **Blocking is cooperative**
  - **All stack variables are lost across a blocking point**
  - **Blocking can only occur in the protothread's root function**
- **Tradeoffs – protothreads are another design point between threads and events**

Thread Variation 2

- Preemption thresholds
  - Every thread has two priorities
    - P1 – regular priority, used to decide when the thread runs
    - P2 – preemption threshold, used to decide whether another thread can preempt currently running thread
  - If P1 == P2 for all threads, degenerates to preemptive multithreading
  - If P2 == max priority, degenerates to non-preemptive scheduling
- Key benefits:
  - Threads that are mutually nonpreemptive can share a stack
  - Reduces number of context switches

Thread Pros

- **Blocking can lead to clearer software**
  - **No need to manually save state**
  - **Reduces number of ad-hoc state machines**
- **Preemptive scheduling can lead to rapid response times**
  - **Only in carefully designed systems**
- **Threads compose multiple activities naturally**
  - **As opposed to cyclic executives**

Thread Cons

- Correctness
  - Empirically, people cannot create correct multithreaded software
  - Race conditions
  - Deadlocks
  - Tough to debug
- Performance
  - Stacks require prohibitive RAM on the smallest systems
  - Context switch overhead can hurt – might end up putting time critical code into interrupts

Thread Rules

- Always write code that is free of data races
- A data race is any variable that is…
  - Written by 1 or more threads
  - Shared between 2 or more threads
  - Not consistently protected by a lock
- For every variable in your code you should be able to say why there is not a data race on it

Thread Rules

- **You must be clear about**
  - **Your locking strategy**
  - **Your call graph**
  - **Where pointers might be pointing**
- **Would a program be free of data races if you disabled interrupts before accessing each shared variable, and enabled afterwards?**
- **Would it be correct?**
- **How long do you hold a lock in general?**

Thread Rules

- **Protect data any time its invariants are broken**
- **This means you have to know what the invariants are!**
- **Examples?**

Thread Rules

- **Always either:**
  - **Acquire only one lock at a time**
    - **Usually not practical**
  - **Assign a total ordering to locks and acquire them in that order**
    - **Requires coordination across developers**

Summary

- Threads have clear advantages for large systems
  - Blocking reduces the need to build state machines
  - Threads simplify composing a system from parts
- Threads have clear disadvantages
  - RAM overhead, for small systems
  - Correctness issues