Based on

Embedded Systems: A Contemporary Design Tool
James Peckol

and

EE472 Lecture Notes Pack
Blake Hannaford, James Peckol, Shwetak Patel
Why would anyone want an OS?

- **Goal:** run multiple programs on the same HW “simultaneously”
  - i.e. multi-tasking…it means more than surfing Facebook during lecture

- **Problem:** how to share resources & avoid interference
  - To be shared: processor, memory, GPIOs, PWM, timers, counters, ADCs, etc
  - In embedded case, we may need to do the sharing while respecting “real time” constraints

- OS is responsible for **scheduling** the various jobs

- **Also:**
  - OS provides abstractions of HW (e.g. device drivers) that make code more portable & re-usable, as well as enabling sharing
  - Code re-use a key goal of ROS (“meta-operating system”)
  - Power: maintain state across power loss, power aware scheduling
Tasks / Processes, Threads

- Task or process
  - Unit of code and data… a program running in its own memory space
- Thread
  - Smaller than a process
  - A single process can contain several threads
  - Memory is shared across threads but not across processes

With just 1 task, it is either Running or Ready Waiting
Types of tasks

- Periodic --- Hard real time
  - Control: sense, compute, & generate new motor cmd every 10ms
  - Multimedia: sample audio, compute filter, generate DAC output every 22.73 μS
  - Characterized by
    - P, Period
    - C, Compute time (may differ from instance to instance, but C<=P)
    - D, Deadline (useful if start time of task is variable)
      - C < D < P

- Intermittent
  - Characterized by
    - C and D, but no P

- Background
  - Soft realtime or non-realtime
  - Characterized by
    - C only

- Complex
  - Examples
    - MS Word, Web server
  - Continuous need for CPU
  - Requests for IO or user input free CPU
Scheduling strategies

- Multiprogramming
  - Running task continues until a stopping point (e.g. waiting for an IO event)

- Real-time
  - Tasks must be completed before deadline

- Time sharing
  - Running task gives up CPU
  - Cooperative multitasking
    - App voluntarily gives up control
      - Old versions of Windows & Mac OS
      - Badly behaved apps hang the system
  - Preemptive multitasking
    - HW timer *preempts* currently executing task, returns control to OS
      - All versions of Unix

- Power aware
  - Research topic
State must be saved / restored to switch between tasks

- Program Counter (PC)
- Register values
- Processor status flags (Status Register)
- Stack Pointer (SP)
- Memory state
- Peripheral configurations
- Etc
Task states in a time-sharing system

- **Enter**
- **Ready Waiting**
- **Blocked/Waiting**
- **Running**
- **Exit**
Memory resource management

- Address space
  - Each process has a range of addresses it’s allowed to use

- Privilege level
  - Supervisory / kernel mode
  - User mode
    - Interrupt generated when a user process tries to operate outside its address space
    - “General protection fault” in x86
## Task Control Block (TCB)

<table>
<thead>
<tr>
<th>Task Control Block</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pointer</td>
</tr>
<tr>
<td>State</td>
</tr>
<tr>
<td>Process ID</td>
</tr>
<tr>
<td>Program Counter</td>
</tr>
<tr>
<td>Register contents</td>
</tr>
<tr>
<td>Memory limits</td>
</tr>
<tr>
<td>Open Files</td>
</tr>
<tr>
<td>Etc.</td>
</tr>
</tbody>
</table>

Also: scheduling information, memory management information, I/O status info
Task Control Block (TCB)

// The task control block
struct TCB
{
    void (*taskPtr)(void* taskDataPtr);
    void* taskDataPtr;
    void* stackPtr;
    unsigned short priority;
    struct TCB* nextPtr;
    struct TCB* prevPtr;
};

taskPtr is a pointer to a function
    The function’s param list has one arg, of type void*
stackPtr: each task has its own stack
Priority: what is the priority level of this task?
nextPtr & prevPtr: pointers to other TCBs
Scheduling
Time (for RTOS)

- Time slice $T$, Ticks
- $P_{\text{min}}$, shortest period of all tasks in system
- $T < P_{\text{min}}$, sometimes $T \ll P_{\text{min}}$
Scheduling goals

- **CPU Utilization**
  \[ U_{CPU} = 1 - \frac{\text{idle}}{\text{period}} \]
  In mainframe, 100% is best, but 100% not safe for realtime systems
  Goal: 40% low load, 90% high load

- **Throughput**

- **Turnaround time**

- **Waiting time**

- **Response time**
Scheduler types

- Infinite loop, aka non-preemptive Round Robin

```c
while(1) {
    task1_fn();
    task2_fn();
    task3_fn();
}

taskN_fn() {
    compute a little bit;
    return();
}
```
Scheduler types

- Synchronized Infinite loop
  - Top of loop waits for a HW clock
    ```
    while(1) {
        wait(CLOCK_PULSE);
        task1_fn();
        task2_fn();
        task3_fn();
    }
    ```
Scheduler types

- Preemptive round robin
  - AKA cyclic executive
  - All processes handled without priority
  - Starvation free
Scheduler types

- Preemptive priority based
  - Goal in non-RT OS is to allocate resources equitably…no process should perpetually lack necessary resources
  - Attach priorities to each process
    - Problem: priority inversion
      - A is highest priority process. It is blocked waiting for a result from C
      - B is 2\textsuperscript{nd} highest priority. It never blocks
      - C is 3\textsuperscript{rd} highest priority
      - Now B runs all the time and A never gets to…their priorities are effectively inverted…A is starved
    - Problem: deadlock
      - Catch 22 / Chicken - Egg: A is waiting for B, but B is waiting for A
      - One person has the pencil but needs the rule, the other has ruler but needs pencil
      - You can’t make coffee until you’re alert…but you’re not alert until you’ve had coffee

- Ways to avoid priority inversion
  - Make sure every job gets a minimum time slice
  - Priority inheritance
    - Does not prevent deadlock when there are circular dependencies
Scheduler types

- Preemptive priority based
  - Rate monotonic scheduling (RMS), for RTOS
    - Static priorities set based on job cycle duration---shorter job gets scheduled more often
    - Provide deterministic guarantees about response times (show using Rate Monotonic Analysis)

\[
U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n\left(2^{1/n} - 1\right)
\]

- Where
  - \(C_i\) is compute time
  - \(T_i\) is release period
  - \(n\) is # processes to be scheduled

\[
\lim_{n \to \infty} n\left(\sqrt[n]{2} - 1\right) = \ln 2 \approx 0.693147\ldots
\]

Roughly, RMS can meet deadlines when CPU < 69% used
End
Real-Time OSes and their communities

- Linux
  - RTLinux
  - RTAI
  - Xenomai

- Commercial
  - LynxOS
  - QNX
  - VxWorks
  - Windows CE
  - iRMX for Windows
  - OSE

- Embedded systems
  - FreeRTOS
  - μC/OS-II

- Sensor networks
  - TinyOS
  - Contiki

- Computational RFID
  - Dewdrop
  - MementOS

- Robotics [“meta OSes,” on top of Linux]
  - ROS
  - Player / Stage
  - Carmen
**RTLinux**

- Hard realtime RTOS microkernal runs entire Linux OS as a preemptive process
- Real time OS is virtual machine “host OS” … Linux kernal runs as “guest OS”
- Interrupts for realtime processing handled by realtime core
- Other interrupts forwarded to Linux, handled at lower priority than realtime interrupts
- Acquired by WindRiver, sold as Wind River Real-Time Core for Wind-River Linux
RTAI & Xenomai (Real time Linux)

- **RTAI==Real Time Application Interface**
  - Provides deterministic response to interrupts
  - Kernel patch allows RT system to take over key interrupts, leaves ordinary Linux to handle others
  - No patent restrictions (vs RTLinux)
  - Lowest feasible latencies

- **Xenomai**
  - Emphasizes extensibility rather than lowest latency
μC/OS-II

- www.ucos-ii.com
- Kernal only…supports
  - Scheduling
  - Message passing (mailboxes)
  - Synchronization (semaphores)
  - Memory management
  - Supports 64 priority levels…runs highest priority first
  - Does not support: IO devices, Files, networking

- Versions
  - mC/GUI
  - mC/USB-Bulk
  - mC/USB-MSD [for Mass Storage Devices]
FreeRTOS

- http://www.freertos.org/
- Another realtime kernel
- Many features similar to μC/OS-II
- Supports both tasks and co-routines
  - A co-routine does not have its own stack
  - Smaller memory footprint, more efficient
  - Restrictions on how/when to call etc required

Versions
- OpenRTOS
  - Commercial, supported
- SafeRTOS
  - Documented for safety critical applications
Contiki and TinyOS

- See Contiki slides
- More info:
  http://www.sics.se/contiki/wiki/index.php/Main_Page
DewDrop

- Energy-aware runtime (scheduler) for computational RFID
- Interesting to compare power aware scheduling to RTOS ("time-aware scheduling")
ROS

- Robot Operating System
  - Meta-operating system

See

- ros_overview.pdf
- ros_tutorial.pdf
Inter-task communication

- Shared variables
  - Global variable
  - Shared buffer: producer & consumer

Problem: mismatch in filling & emptying rates can lead to over- or underflow
Solution: always check empty / full before reading / writing
Inter-task communication

- Shared variables
  - Shared double buffer (ping pong buffer)

One buffer is being filled while the other is being emptied (also used for displays / graphics!)

Can generalize to n buffers…may be useful when producer generates data in fast short bursts
Inter-task communication

- Shared variables
  - Ring buffer
    - An implementation of a queue, used to let 2 processes communicate
    - FIFO (First In First Out)
    - Need to avoid under/overflow

D0 – D3: valid data
xx: junk
Inter-task communication

- Shared variables
  - Mailbox

  ![Diagram of Task T0 and Task T1 with post and pend actions]

  Interface
  ```
  post(mailbox, data) // post to mailbox
  pend(mailbox, data) // pend on mailbox
  ```

  A flag indicates that data has been posted…reading clears flag

  Variants: can implement as
  - a queue of length 1,
  - extensible queue (length n)
  - priority queue

  A way to share a critical resources

  Pend differs from poll since during pend, CPU can do other things
### Inter-task communication

- **Messaging / communication**
  - Generalize mailbox from “agreed-upon memory address accessed by defined interface” to more abstract address (which could be on another processor)
  - Inter-Process Communication (IPC)
    - send & receive instead of post & pend
Inter-task communication

- Messaging / communication
  - Direct
    - send (T1, message)  // send message to Task T1
    - receive (T0, message)  // receive message from Task T0
  - Indirect
    - send(M0, message)  // send message to mailbox M0
    - receive(M0, message)  // receive message from mailbox M0
    - Multiple tasks may be able to read from / write to a mailbox
Inter-task communication

- **Messaging / communication**
  - Messaging systems can be buffered in 3 different ways
    - Link has 0 capacity → *rendezvous* or *Idle RQ protocol*
      - RQ: “Repeat reQuest”
      - TX waits for RX to accept message [ACK, NACK, timeout]
      - AKA “stop and wait” or “synchronous”
    - Link has bounded capacity…queue length of n
    - Link has unbounded capacity → *continuous RQ protocol*
      - TX never has to wait
      - TX can send next packet before receiving ACK from previous packets
      - AKA “asynchronous”
  - NB: Idle RQ and Continuous RQ are examples of “backward error correction” (BEC) protocols, which manage re-transmission when errors are detected. Contrast with Forward Error Correction (FEC), which we discussed earlier with error correcting codes [Hamming, LDPC, Raptor, etc]
Task cooperation, synchronization, sharing

- Concurrent access to common data can result in data inconsistency, unexpected behavior, system failure
- Need to manage interactions of multiple tasks with common resources
Task cooperation, synchronization, sharing

- Bridge example
  - Critical section of roadway...can’t be occupied by both cars at once
  - Need to manage access to shared resource to avoid collisions
Task cooperation, synchronization, sharing

- Example: N item buffer

  ![Diagram showing task cooperation and synchronization]

  - Producer (Task T0):
    - Start in idle state.
    - Transition to write state if not full.
    - Transition to inc cnt state if not full.
    - Terminate.

  - Consumer (Task T1):
    - Start in idle state.
    - Transition to read state if not empty.
    - Transition to dec cnt state if not empty.
    - Terminate.
Task cooperation, synchronization, sharing

Example: N item buffer

Task T0 --- Producer
while (1)
  if not full
    add item
    increment count
  else
    wait for space
end while

Task T1 --- Consumer
while (1)
  if not empty
    get item
    decrement count
  else
    wait for item
end while

The variable count is a critical shared resource...its value can depend on how the two processes interleave at the lowest level...see next slide
Task cooperation, synchronization, sharing

Example of problem

**count++** implementation:

- `register1 = count`
- `register1 = register1 + 1`
- `count = register1`

**count--** implementation:

- `register2 = count`
- `register2 = register2 - 1`
- `count = register2`

Let `count = 5` initially. One possible concurrent execution of `count++` and `count--` is:

- `register1 = count {register1 = 5}`
- `register1 = register1 + 1 {register1 = 6}`
- `register2 = count {register2 = 5}`
- `register2 = register2 - 1 {register2 = 4}`
- `count = register1 {count = 6}`
- `count = register2 {count = 4}`

`count = 4` after `count++` and `count--`, even though we started with `count = 5`

Question: what other values can `count` be from doing this incorrectly?

“Race condition” --- result is determined by “which input gets to the output first”

Any SW or HW situation in which result depends critically on timing

*Problem is caused by inter-leaving of read & write operations on the same variable*
Task cooperation, synchronization, sharing

- Example of non-problem

**count++** implementation:

```
register1 = count
register1 = register1 + 1
count = register1
```

**count--** implementation:

```
register2 = count
register2 = register2 - 1
count = register2
```

Let count = 5 initially. One possible concurrent execution of count++ and count-- is

```
register1 = count {register1 = 5}
register1 = register1 + 1 {register1 = 6}
count = register1 {count = 6}
register2 = count {register2 = 6}
register2 = register2 - 1 {register2 = 5}
count = register2 {count = 5}
count = 5, the correct value
```

*This worked correctly because the operations modifying count were not interleaved*
Task cooperation, synchronization, sharing

- How to prevent problems due to concurrent access to shared resources?
  - Ensure that access to shared resource is *mutually exclusive*...only one process can access at time!
    - Mutual exclusion synchronization [locks]
    - Condition synchronization
  - Structure of a critical section
    ```
    while(1)
      non-critical code
      entry section
      critical section
      exit section
      non-critical code
    end while
    ```
Task cooperation, synchronization, sharing

Requirements to solve critical section problem

- Ensure mutual exclusion in critical region
- Prevent deadlock
- Ensure progress through critical section
- Ensure bounded waiting
  - Upper limit on the number of times a lower priority task can be blocked by a higher priority task

Definition: an *atomic operation* is guaranteed to terminate without being interrupted…all sub-steps comprising an atomic operation succeed or fail together
Mechanisms for implementing mutual exclusion

- Flags, embedded in an atomic operation

```
await (condition) { // await is “atomic wait”
    statements
} variable
```

Other tasks must be able to execute during `await`, otherwise deadlock can occur

Use `T0Flag` to mean Task 0 has lock; `T1Flag` means Task 1 has lock

```
await (!T1Flag) {T0Flag=True;}
await (!T0Flag) {T1Flag=True;}
```
Task cooperation, synchronization, sharing

- Mechanisms for implementing mutual exclusion
  - Flags
    - `count++` implementation:
      ```
      register1 = count
      register1 = register1 + 1
      count = register1
      ```
    - `count--` implementation:
      ```
      register2 = count
      register2 = register2 - 1
      count = register2
      ```

Task T0 --- Producer
```
while (1)
  if not full
    add item
    await(!T1Flag) {T0Flag=true;}
    count++
    T0Flag = false;
  else
    wait for space
end while
```

Task T1 --- Consumer
```
while (1)
  if not empty
    get item
    await(!T0Flag) {T1Flag=false;}
    count--
    T1Flag = false;
  else
    wait for item
end while
```
Task cooperation, synchronization, sharing

- **Mechanisms for implementing mutual exclusion**
  - Token passing: one token gets passed among tasks...only the task holding the token can access the resource

  - Problems:
    - Task holds on to token forever
    - Task with token crashes
    - Token lost or corrupted
    - Task terminates without releasing token
    - How to add new tasks?

- **Possible solutions?**
  - Add a system task which manages token, and has watchdog timer
  - Getting complicated though
Task cooperation, synchronization, sharing

- Mechanisms for implementing mutual exclusion
  - Interrupt management
    - In a single processor system, disable interrupts in critical section
    - Similar problems to token passing: badly behaved code can screw up
    - Similar solutions: use a watchdog timer (with higher priority interrupt level, one that does not get disabled by critical section)
Task cooperation, synchronization, sharing

- Mechanisms for implementing mutual exclusion
  - Semaphores
    - Used to indicate availability of critical variable
    - Simplest example: boolean S with two atomic access operations
      - wait: P(S)  P from Dutch proberen, to test
        - wait tests semaphore value, and if false, sets to true
        - wait has two parts, test and set, which must occur together atomically
      - signal: V(S)  V from Dutch verhogen, to increment
        - sets value to false
Task cooperation, synchronization, sharing

- **Mechanisms for implementing mutual exclusion**
  - **Semaphores**

```c
// implementation of semaphore
// notes: - wait must happen atomically!
//       - s should be initialized to false

void wait(s) {
    while (s); // do nothing while another process has s set
    s = TRUE; // now WE set s to be true to warn other processes
}

void signal(s) {
    s = FALSE; // Turn off warning for other processes
}
```
Task cooperation, synchronization, sharing

- Mechanisms for implementing mutual exclusion
  - Semaphores

// use of semaphores

Task T0 {
    ...
    wait(s)
    critical section
    signal(s)
    ...
}

Task T1 {
    ...
    wait(s)
    critical section
    signal(s)
    ...
}
Task cooperation, synchronization, sharing

- Mechanism for synchronization
  - Semaphores
    - Can also be used to enforce ordered execution of asynchronous tasks
    - Want \( f(x) \) to be called before \( g(y) \)
    - Use semaphore \( \text{sync} \) to do this

```c
// semaphores for synchronization
sync = true // initialization

Task T0 {
    ...
    f(x)
    signal(sync)
    ...
}

Task T1 {
    ...
    wait(sync) // wait
g(y)
    ...
}
```

Lock on critical section is called a spin lock, because T1 “spins” waiting for \( \text{sync} \) signal. Other activity can occur on the system while T1 is waiting, but T1 is not accomplishing anything while waiting.
Task cooperation, synchronization, sharing

- Mechanisms for implementing mutual exclusion
  - Semaphores
    - Can be non-binary: counting semaphores
    - Useful for managing a pool of identical resources
    - P and V, wait and signal, down and up, and other names used for semaphore access functions
  - vs Mutex [mutual exclusion]: same as binary semaphore, but
    - Mutex often has a notion of an “owner process” who must release mutex; semaphore usually has no owner
Example messaging system: ROS

- See ROS slides
ROS & multithreading in roscpp

- roscpp is the C++ implementation of ROS
  - roscpp provides a client library / API for C++ programmers
  - roscpp is the high performance option
  - vs rospy, python client library / API
- roscpp does not specify a threading model for apps
Single threaded spinning: `spin()`

```cpp
1 ros::init(argc, argv, "my_node");
2 ros::NodeHandle nh;
3 ros::Subscriber sub = nh.subscribe(...);
4 ... 
5 ros::spin();
```

- All user callbacks will be called from within `ros::spin()`
- `ros::spin()` does not return until node shuts down...instead, message handling events get processed
Single threaded spinning: `spinOnce()`

1. `ros::Rate r(10);` // 10 hz
2. `while (should_continue)`
3. `{`
4. `... do some work, publish some messages, etc. ...`
5. `ros::spinOnce();`
6. `r.sleep();`
7. `}
8
- **Call** `ros::spinOnce()` **periodically**
- `ros::spinOnce()` **calls all callbacks that are currently waiting to be processed**
- **Note:** `spin()` and `spinOnce()` are intended for single threaded apps
**Multi-threaded spinning**: `MultiThreadedSpinner()`

```cpp
t  ros::MultiThreadedSpinner spinner(4); // Use 4 threads
t  spinner.spin(); // spin() will not return until node has been shutdown
```

- Blocking spinner, similar to `spin()`
- You specify a number of threads
- Defaults to one thread per CPU core
Multi-threaded spinning: AsyncSpinner()

1 ros::AsyncSpinner spinner(4); // Use 4 threads
2 spinner.start();
3 ros::waitForShutdown();
4

- This example is equivalent to previous blocking example
- Call to start() is non-blocking---execution returns right away
- In a real use, you’d put useful code after the start(), instead of immediately doing waitForShutdown()