...the Linux philosophy is “laugh in the face of danger”. Oops. Wrong one. “Do it yourself”. That’s it. – Linux Torvalds
Motivation

• We have already run into some limitations of the standalone process model:
  – Single application, growing in complexity quickly
  – Lots of polling loops, deep nested ‘if’ statements

• We could continue in this direction, but a modern Operating System (OS) provides streamlined mechanisms for:
  – Preemptive multitasking
  – Device drivers
  – Memory management
  – File systems

• It would be insane to try to cover all the major issues involved in embedded OS in a single lecture
This Week’s Topic

- Embedded Operating System features
  - Processes and scheduling
    - Context switching
    - Scheduler policies
    - Real-Time Operating Systems (RTOS)
  - Atomic operations
  - Inter-processes communication
  - Virtual memory
  - Examples along the way:
    - Linux, POSIX, freeRTOS.org
    - ARM architecture support

- Reading: Wolf chapter 6, 3.5
Reactive Systems

• Respond to external events:
  – Engine controller
  – Seat belt monitor

• Requires real-time response:
  – System architecture
  – Program implementation

• May require a chain reaction among multiple processors
Tasks and Processes

- A task is a functional description of a connected set of operations
- Task can also mean a collection of processes
- A process is a **unique execution** of a program
  - Several copies of a program may run simultaneously or at different times
- A process has its own state:
  - registers
  - memory
- The operating system manages processes
Process State

- A process can be in one of three states:
  - executing on the CPU
  - ready to run
  - waiting for data
Embedded vs. General-Purpose Scheduling

• Workstations try to avoid starving processes of CPU access
  – Fairness == access to CPU

• Embedded systems must meet deadlines
  – Low-priority processes may not run for a long time
Preemptive Scheduling

• Timer interrupt gives CPU to kernel
  – Time quantum is smallest increment of CPU scheduling time

• Kernel decides what task runs next

• Kernel performs context switch to new context
Context Switching

• Set of registers that define a process’s state is its context
  – Stored in a record

• Context switch moves the CPU from one process’s context to another

• Context switching code is usually assembly code
  – Restoring context is particularly tricky
freeRTOS.org Context Switch

Diagram showing the context switch process involving various stages such as timer, vPreemptiveTick, portSAVE_CONTEXT, portRESTORE_CONTEXT, vTaskSwitchContext, and tasks 1 and 2.
freeRTOS.org Timer Handler

```c
void vPreemptiveTick( void ) {
    /* Save the context of the current task. */
    portSAVE_CONTEXT();
    /* Increment the tick count - this may wake a task. */
    vTaskIncrementTick();
    /* Find the highest priority task that is ready to run. */
    vTaskSwitchContext();
    /* End the interrupt in the AIC. */
    AT91C_BASE_AIC->AIC_EOICR = AT91C_BASE_PITC->PITC_PIVR;
    portRESTORE_CONTEXT();
}
```
ARM Context Switch

User mode

<table>
<thead>
<tr>
<th>r0</th>
<th>r1</th>
<th>r2</th>
<th>r3</th>
<th>r4</th>
<th>r5</th>
<th>r6</th>
<th>r7</th>
<th>r8</th>
<th>r9</th>
<th>r10</th>
<th>r11</th>
<th>r12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

IRQ

<table>
<thead>
<tr>
<th>r8</th>
<th>r9</th>
<th>r10</th>
<th>r11</th>
<th>r12</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FIQ

<table>
<thead>
<tr>
<th>r13 (sp)</th>
<th>r14 (lr)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Undefined

<table>
<thead>
<tr>
<th>r13 (sp)</th>
<th>r14 (lr)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Abort

<table>
<thead>
<tr>
<th>r13 (sp)</th>
<th>r14 (lr)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

SVC

<table>
<thead>
<tr>
<th>r13 (sp)</th>
<th>r14 (lr)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

STM

```
STM sp, {R0-1r}^ 
```

; Dump user registers above R13.

MRS

```
MRS R0, SPSR 
```

; Pick up the user status

STMDB

```
STMDB sp, {R0, lr} 
```

; and dump with return address below.

LDR

```
LDR sp, [R12], #4 
```

; Load next process info pointer.

CMP

```
CMP sp, #0 
```

; If it is zero, it is invalid

LDMDBNE

```
LDMDBNE sp, {R0, lr} 
```

; Pick up status and return address.

MSRNE

```
MSRNE SPSR_cxsf, R0 
```

; Restore the status.

LDMNE

```
LDMNE sp, {R0-1r}^ 
```

; Get the rest of the registers

NOP

```
NOP 
```

; and return and restore CPSR.

SUBSNE

```
SUBSNE pc, lr, #4 
```

; Insert "no next process code" here.
Real-Time Systems

• What is a real-time system?
• Which of the following is real-time?
  – A program that processes 100 video frames per second?
  – A program that that process 1 video frame per 10 seconds?

• A better name
  – “Get things done on time” Systems
• They are about getting things done on time, not getting things done fast
**Real-Time Systems: Key Terms/Concepts**

- **Task**
  - Cost: time for processor to complete task without interruptions
  - Release time: when task is ready to be run
  - Deadline: time by which task needs to be completed
  - Period: time between release times

- **Task-set schedule**: order in which tasks are allocated the CPU

- **Scheduling policy (algorithm)**: means by which (i.e., rules followed) to create a task-set schedule
Scheduling: Period vs Aperiodic

- **Periodic process**: executes on (almost) every period
- **Aperiodic process**: executes on demand
- Analyzing aperiodic process sets is harder---must consider worst-case combinations of process activations
Timing Requirements on Processes

• **Period**: interval between process activations
• **Initiation interval**: reciprocal of period
• **Initiation time**: time at which process becomes ready
• **Deadline**: time at which process must finish

• What happens if a process doesn’t finish by its deadline?
  – **Hard deadline**: system fails if missed
  – **Soft deadline**: user may notice, but system doesn’t necessarily fail
Priority-driven Scheduling

- Each process has a priority
- CPU goes to highest-priority process that is ready
- Priorities determine scheduling policy:
  - Fixed (Static) priority
  - Time-varying (Dynamic) priorities
Priority-driven Scheduling Example

• Rules:
  – Each process has a fixed priority (1 highest)
  – Highest-priority ready process gets CPU
  – Process will not self stop (i.e. block) until done
  – Pre-emptive scheduling

• Processes
  – P1: priority 1, execution time 10, release time 15
  – P2: priority 2, execution time 30, release time 0
  – P3: priority 3, execution time 20, release time 18
 Priority-driven Scheduling Example (cont.)

P1: priority 1, execution time 10, release time 15
P2: priority 2, execution time 30, release time 0
P3: priority 3, execution time 20, release time 18

P3 ready $t=18$

P2 ready $t=0$  P1 ready $t=15$

P2

P1

P2

P3

0  10  20  30  40  50  60

time
The Scheduling Problem

• Can we meet all deadlines?
  – Must be able to meet deadlines in all cases

• How much CPU horsepower do we need to meet our deadlines?
CPU Utilization

• T1: PPM update
  – Cost = 10 ms
  – Deadline = 25 ms
  – Period = 25 ms

• What is the CPU utilization of T1?
Scheduling Example (with preemption)

- **T1: PPM update**
  - Cost = 10 ms
  - Deadline = 25 ms
  - Period = 25 ms

- **T2: Video processing**
  - Cost = 20 ms
  - Deadline = 50 ms
  - Period = 50 ms

- **What rules to follow for scheduling**
  - Let’s say that the more often a task needs to run, the higher the priority (allow preemption)
  - Draw out schedule and see if we miss a deadline
Scheduling Example (no preemption)

- **T1: PPM update**
  - Cost = 10 ms
  - Deadline = 25 ms
  - Period = 25 ms

- **T2: Video processing**
  - Cost = 20 ms
  - Deadline = 50 ms
  - Period = 50 ms

- **What rules to follow for scheduling**
  - Let’s say that the more often a Task needs to run, the higher the priority (now allow NO preemption)
  - Is there a release pattern that can cause a task to miss a deadline?
Scheduling Metrics

• How do we evaluate a scheduling policy:
  – Ability to satisfy all deadlines (Feasibility)
  – CPU utilization---percentage of time devoted to useful work
  – Scheduling overhead---time required to make scheduling decision
Scheduling Metrics: Feasibility

• For previous preemption example
  – How long do we have to draw out the schedule before we know we will never miss a deadline?
  – What if we had 3 tasks with period 3ms, 4ms, and 7ms?
  – For a general task set, for how do we have to draw out the schedule?
Scheduling Metrics: Feasibility

• For previous preemption example
  – How long do we have to draw out the schedule before we know we will never miss a deadline?
  – What if we had 3 tasks with period 3ms, 4ms, and 7ms? Answer: 84 ms
  – For a general task set, how do we have to draw out the schedule? Answer: Lowest common multiple of Task periods (a task set’s Hyper Period). This is the time it takes before all Tasks release times synchronize after time = 0

• Is there a better way to determine is feasible (i.e. schedule using a given policy)? Yes! RMA
Rate Monotonic Scheduling

• **RMS** (Liu and Layland): widely-used, analyzable scheduling policy

• Analysis is known as **Rate Monotonic Analysis (RMA)**
  – All process run on single CPU
  – Zero context switch time
  – No data dependencies between processes
  – Process execution time is constant
  – Deadline is at end of period
  – Highest-priority ready process runs
Process Parameters

- $T_i$ is computation time of process $i$; $\tau_i$ is period of process $i$.  

Diagram:

- Period $\tau_i$ 
- Computation time $T_i$
Rate-Monotonic Analysis

- **Response time**: time required to finish process
- **Critical instant**: scheduling state that gives worst response time
- Critical instant occurs when all higher-priority processes are ready to execute
Critical Instant

interfering processes

P1  P1  P1  P1  P1

P2  P2  P2

P3  P3

P4

critical instant
RMS Priorities

• Optimal (fixed) priority assignment:
  – Shortest-period process gets highest priority
  – Priority inversely proportional to period
  – Break ties arbitrarily

• No fixed-priority scheme does better
RMS CPU Utilization

- Utilization for $n$ processes is
  $$-\sum_{i} T_i / \tau_i$$

- As number of tasks approaches infinity, maximum utilization approaches 69%
  - If the Task set Utilization $\leq 69\%$, then RMS is guaranteed to meet all deadlines
  - If Utilization $> 69\%$, then must draw schedule for the Lowest Common Multiple (LCM) of the Task set periods.
  - Positive: Quick way to determine Feasibility
  - Negative: Gives up about 30% of CPU Utilization
Earliest-Deadline-First Scheduling

- **EDF**: dynamic priority scheduling scheme
- Process closest to its deadline has highest priority
- Requires recalculating processes at every timer interrupt

- EDF can use 100% of CPU
  - But part of that 100% will be used for computing/updating Task priorities
Modified Scheduling Example

• T1: PPM update
  – Cost = 7 ms
  – Deadline = 12 ms
  – Period = 12 ms

• T2: Video processing
  – Cost 20.5 ms
  – Deadline 50 ms
  – Period = 50 ms

• Lets try RMS first
EDF Implementation

• On each timer interrupt:
  – Compute time to deadline
  – Choose process closest to deadline

• Generally considered too expensive to use in practice
Scheduling Problems

• What if your set of processes is unschedulable?
  – Change deadlines in requirements
  – Reduce execution times of processes
  – Get a faster CPU

• Note for RMS: If periods of task sets are “Harmonic” then RMS can handle 100% utilization
Fixed Priority Concern: Priority Inversion

• **Priority inversion**: low-priority process keeps high-priority process from running

• Improper use of system resources can cause scheduling problems:
  – Low-priority process grabs I/O device
  – High-priority device needs I/O device, but can’t get it until low-priority process is done

• Can cause deadlock
Solving Priority Inversion

• Priority Inheritance: Have process inherit the priority of the highest process being blocked
  – Can still have deadlock

• Priority Ceilings: Process can only enter a critical section of code, if no other higher priority process owns a resource that it may need.
  – Solves deadlock issue
Context-Switching Time

• Non-zero context switch time can push limits of a tight schedule
• Hard to calculate effects---depends on order of context switches
• In practice, OS context switch overhead is small (hundreds of clock cycles) relative to many common task periods (ms – μs)
Interprocess Communication

- Interprocess communication (IPC): OS provides mechanisms so that processes can pass data

- Two types of semantics:
  - **blocking**: sending process waits for response
  - **non-blocking**: sending process continues
IPC Styles

- **Shared memory:**
  - Processes have some memory in common
  - Must cooperate to avoid destroying/missing messages

- **Message passing:**
  - Processes send messages along a communication channel---no common address space
• Shared memory on a bus:
Race Condition in Shared Memory

- Problem when two CPUs try to write the same location:
  - CPU 1 reads flag and sees 0
  - CPU 2 reads flag and sees 0
  - CPU 1 sets flag to one and writes location
  - CPU 2 sets flag to one and overwrites location
Atomic Test-and-Set

• Problem can be solved with an atomic test-and-set:
  – Single bus operation reads memory location, tests it, writes it.

• ARM test-and-set provided by SWP (originally, more modern chips use LDREX, STREX):

  ADR r0,SEMAPHORE
  LDR r1,#1
  GETFLAG: SWP r1,r1,[r0]
  BNZ GETFLAG
Critical Regions

• **Critical region**: section of code that cannot be interrupted by another process

• Examples:
  – Writing shared memory
  – Accessing I/O device
Semaphores

- **Semaphore**: OS primitive for controlling access to critical regions

- **Protocol:**
  - Get access to semaphore with $P()$
  - Perform critical region operations
  - Release semaphore with $V()$
• Message passing on a network:
freeRTOS.org Queues

- Queues can be used to pass messages
- Operating system manages queues

```c
xQueueHandle q1;
q1 = xQueueCreate( MAX_SIZE, sizeof(msg_record));
if (q1 == 0) /* error */
xQueueSend(q1, (void *)msg, (portTickType)0);
/* queue, message to send, final parameter controls timeout */
if (xQueueReceive(q2, &(in_msg), 0);
/* queue, message received, timeout */
```
Signals

• Similar to a software interrupt.
• Changes flow of control but does not pass parameters.
  – May be typed to allow several types of signals.
  – Unix ^c sends kill signal to process.

```
<<signal>>
asig

p: integer

<<send>>
someClass

sigbehavior()
```

Mailbox

- Fixed memory or register used for interprocess communication
- May be implemented directly in hardware or by RTOS

```c
void post(message *msg) {
    P(mailbox.sem);
    copy(mailbox.data, msg);
    mailbox.flag = TRUE
    V(mailbox.sem);
}

bool pickup(message *msg) {
    bool pickup = FALSE;
    P(mailbox.sem);
    pickup = mailbox.flag;
    mailbox.flag = FALSE;
    copy(msg, mailbox.data);
    V(mailbox.sem);
    return(pickup);
}
```
### POSIX Process Creation

- fork() makes two copies of executing process
- Child process identifies itself and overlays new code

```c
if (childid == 0) {
    /* must be child */
    execv("mychild", childargs);
    perror("execv");
    exit(1);
}
else { /* is the parent */
    parent_stuff();
    wait(&cstatus);
    exit(0);
}
```
POSIX Real-Time Scheduling

- Processes may run under different scheduling policies
- `_POSIX_PRIORITY_SCHEDULING` resource supports real-time scheduling
- `SCHED_FIFO` supports RMS

```c
int i, my_process_id;
struct sched_param my_sched_params;
...

i = sched_setscheduler(my_process_id, SCHED_FIFO, &sched_params);
```
POSIX Interprocess Communication

• Supports counting semaphores in _POSIX_SEMAPHORES
• Supports shared memory

```c
i = sem_wait(my_semaphore); /* P */
/* do useful work */
i = sem_post(my_semaphore); /* V */

/* sem_trywait tests without blocking */
i = sem_trywait(my_semaphore);
```
POSIX Pipes

- Pipes directly connect programs
- pipe() function creates a pipe to talk to a child before the child is created

```c
if (pipe(pipe_ends) < 0) {
    perror("pipe");
    break;
}
childid = fork();
if (childid == 0) {
    childargs[0] = pipe_ends[1];
    execv("mychild", childargs);
    perror("execv");
    exit(1);
}
else { ... }
```
Memory Management Units

- Memory management unit (MMU) translates addresses:

  ![Diagram](image)

- CPU
- Logical address
- Memory management unit
- Physical address
- Main memory
Memory Management Tasks

- Allows programs to move in physical memory during execution
- Allows virtual memory:
  - Memory images kept in secondary storage
  - Images returned to main memory on demand during execution
- Page fault: request for location not resident in memory
Address Translation

• Requires some sort of register/table to allow arbitrary mappings of logical to physical addresses

• Two basic schemes:
  – segmented
  – paged

• Segmentation and paging can be combined (x86)
Segments and Pages

- page 1
- page 2
- segment 1
- memory
- segment 2
Segment Address Translation

- **segment base address**
- **logical address**
- **range check**
- **physical address**

- segment lower bound
- segment upper bound

+ range

- range error
Page Address Translation (cont.)

- Page i base
- Concatenate
  - Page
  - Offset
Page Table Organizations

- **flat**
- **tree**

- **page descriptor**
Caching Address Translations

• Large translation tables require main memory access

• **TLB**: cache for address translation
  – Typically small
ARM Memory Management

• Memory region types:
  – Section: 1 Mbyte block
  – Large page: 64 kbytes
  – Small page: 4 kbytes

• An address is marked as section-mapped or page-mapped

• Two-level translation scheme
ARM Address Translation

Translation table base register

descriptor
1st level table

1st index

concatenate

2nd index

concatenate

offset

concatenate

physical address

descriptor
2nd level table
Root file system is an essential component of any Linux system and contains many critical system components:
- Applications
- Configuration files
- Shared libraries
- Data files

Mounted after kernel initialization completes.
Contains first app run by initialization process.

Folders common to Desktop Linux:

- **/dev** - System devices (Chapter 3.1 topic)
- **/root** - Storage for super user files
  - Each user gets their own folder (e.g. /home/user)
  - Similar to “My Documents” in Windows
  - “root” user is different, that user’s folder is at /root
- **/mnt** - Mount point for other file systems
  - Linux only allows one root file system but other disks can be added by mounting them to a directory in the root file system
  - Similar to mapping a drive under Windows
- **/lib** - System libraries
  - Location of system shared object libraries
  - Similar to Windows “C:\Windows\System”
- **/sys and /proc** - Virtual file systems location
  - Exposes kernel parameters (kobjects) as files
  - Similar to Windows Registry
- **/usr** - Storage for user binaries
  - Similar to “Program Files” in Windows
  - Linux system programs are stored in here
Linux Device Driver Types

- Applications
- Kernel
- CPU
- Memory
- Devices
- Character Driver
  - Stream Oriented
- Network Driver
  - Packet Oriented

Random Access
Linux Device Driver Modularity

- Provide access to physical hardware resources
- Built-in or loaded at run-time (loadable modules)
- Can be multi-layered subsystems (USB, I2C, Ethernet)
System Call API

- Example library functions:
  - `fopen()`
  - `fread()`
  - `fwrite()`
  - `fseek()`
  - `fclose()`

- Primary way applications interact with the kernel
Traditional Device Drivers vs. sysfs

• Traditional /dev devices
  – Handles streaming data (i.e. audio/video)
  – Efficient exchange of binary data and structures rather than individual text strings
  – Protection from simultaneous access

• Device drivers under sysfs
  – Limited to simple single text value
  – Easy access to device data via both shell scripts and user space programs

• Weigh the tradeoffs to decide which solution is appropriate for your own application
sysfs Device Driver Example

GPIO7 is connected to MIO7 pin which controls LED LD9 on ZedBoard

Shell Commands:
/sys/class/gpio/gpio7 # echo 1 > value
/sys/class/gpio/gpio7 # cat value

C code:
```c
fprintf(file_led7, "%d", 1);  /* write */
fscanf(file_led7, "%d", &n_ch); /* read */
```
Acknowlegments

• These slides are inspired in part by material developed and copyright by:
  – Marilyn Wolf (Georgia Tech)
  – Fred Kuhns (Washington University in St. Louis)
  – Steve Furber (University of Manchester)
  – Ed Lee (UC-Berkeley)