Basic Functions of OS
• OS controls resources:
  • who gets the CPU;
  • when I/O takes place;
  • how much memory is allocated.
• Application programs run on top of OS services
• Challenge: manage multiple, concurrent tasks.

Example: Engine Control
• Concurrent tasks:
  • spark plug firing
  • crankshaft sensing
  • fuel/air mixture
  • oxygen sensor
• Tasks have different periods / rates
  • Must finish within the period
  • Rate of spark plug control relies on engine speed

Example: Sensor Motes
• Sensing
  • Sampling (multiple) sensors in different rates
• Communication
  • Send data
  • Receive data
• Computation
  • Data processing and aggregation
  • Routing

Process
• 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 context:
  • Data in registers, PC, status, memory.
  • Stored in activation record
• OS manages processes.
Processes and CPUs

- Activation record
  - process context.
- Context switch:
  - current CPU context goes out;
  - new CPU context goes in.

Co-Operative Multitasking

- Process management:
  - hides context switching mechanism;
  - relies on processes to voluntarily give up CPU.
- Each process allows a context switch at cswitch() call.
- Separate scheduler chooses which process runs next.

Preemptive Multitasking

- No more voluntary release of CPU
  - Operating System (OS) is now in charge
- Most powerful form of multitasking:
  - OS controls when context switches;
  - OS determines what process runs next.
- Use periodic timer interrupts to call OS to switch contexts

Problems with Co-Operative Multitasking

- Programming errors can keep other processes out:
  - process never gives up CPU;
  - process waits too long to switch, missing input.

Preemptive Context Switching

- Timer interrupt gives control to OS, which saves interrupted process’s state in an activation record.
- OS chooses next process to run.
- OS installs desired activation record as current CPU state.

Process Scheduling

- Who gets CPU?
  - Usually priority-based
    - Priority can be based on deadline, requested time, or...
- When context switch occurs?
  - Low priority tasks are running while high priority tasks are waiting
- We have three classes to introduce different scheduling algorithms
  - Key issue of real-time systems
Process States

- A process can be in one of three states:
  - executing on the CPU;
  - ready to run;
  - waiting for data.

Process Management

- OS keeps track of:
  - process priorities;
  - scheduling state;
  - process control block.
- Processes may be created:
  - statically before system starts;
  - dynamically during execution.
- OS controls when context switches and what process runs.

Process Scheduling

Embedded vs. General-Purpose

- 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.

Priority-Driven Scheduling

- Every process has a priority.
- CPU goes to the highest-priority ready process.
- Variants
  - Fixed vs. dynamic priority
  - Preemptive vs. non-preemptive

Scheduling Example

- Each process has a fixed priority (1 highest);
- P1: priority 1; P2: priority 2; P3: priority 3.

Preemptive Priority Scheduling

- Most common real-time scheduling approach
  - Real-Time POSIX
  - Real-time priorities in Linux, Solaris, and Windows
  - Most RTOS: VxWorks …
- Not the only possible way
  - Clock-driven scheduling (e.g., round robin)
  - Reservation-based scheduling
  - Proportional share scheduling
  - FIFO scheduling
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?
- Timing violations: 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.

POSIX

- IEEE standards designed to provide application portability between Unix variants.
  - IEEE 1003.1 defines a Unix-like OS interface.
  - IEEE 1003.2 defines the shell and utilities
  - IEEE 1003.4 defines real-time extensions.
- Supported by many operating systems
  - Variants of UNIX: AIX, HP-UX, Solaris, Linux
  - Many commercial RTOS, e.g., VxWorks
  - Windows provides similar services

Processes in POSIX

Create a process with fork:
- parent process keeps executing the old program;
- child process executes a new program.

Create a New Process using fork()

- A process can create a child process by making a copy of itself
- Parent process is returned with the child process ID
- Child process gets a return value of 0

```c
child_id = fork();
if (child_id == 0) {
    /* child operations */
    execv("mychild", childargs);
    exit(0);
} else {
    /* parent operations */
    wait(&csstatus);
    exit(0);
}
```

Overlay a Process using execv()

- Child process usually runs different code
- Use execv() to overlay the code of a process
- Parent uses wait() to wait for child process to finish and then release its memory

```c
childid = fork();
if (childid == 0) {
    execv("child", childargs);
    exit(0);
} else {
    parent_stuff();
    wait(&status);
    exit(0);
}
```

Real-Time Operating Systems

- Definition
  - Multitasking OS intended for real-time applications
- RTOS facilitates the creation of a real-time system but does not guarantee real-time
  - Key factors are bounded interrupt latency and bounded thread switching latency
  - how quick and/or predictable to a particular event
  - Scheduling, inter-task communication, resource sharing, etc
- Types of RTOS
  - Proprietary kernels
  - Real-time extensions to general-purpose OS

file with child code
Proprietary Kernels

- Homegrown kernels
  - Highly specialized for specific applications
    - e.g., nuclear power plant
  - Less common
- Commercial RTOS

Features for Efficiency

- Small
- Minimal set of functionality
- Fast and time bounded context switch
- Fast and time bounded response to interrupts
- Fixed or variable partitions of memory
  - May not support paging or virtual memory
  - Often support locking code and data in memory
- Sequential file that can accumulate data at fast rate
  - May be memory-based

Features for Real-Time

- Preemptive priority scheduling
  - At least 32 priority levels, commonly 128-256 priority levels
  - Priority inheritance/ceiling protocol
  - Usually does not directly support Earliest Deadline First (EDF)
- System calls
  - Bounded execution times
  - Short non-preemptable code
- High-resolution system clock
  - Resolution down to nanoseconds
  - But it takes about a microsecond to process a timer interrupt

Real-Time Extensions to General-Purpose OS

- Generally slower and less predictable than RTOS
- Much greater functionality and development support
- Standard interfaces
- Useful for soft real-time and complex applications

Real-Time Linux

- **Compliant kernels:** Modified native RTOS
  - Functionality and semantics of Linux are emulated
    - e.g., LynxOS (LynuxWorks)
- **Dual kernels:** Hard real-time kernel sits below Linux
  - Real-time kernel traps all interrupts and schedules all processes
  - Linux runs as a low-priority process
  - No memory protection between the two kernels
    - e.g., RT-Linux (FSM, Finite State Machine, Labs)
- **Core kernel modifications:** patches
  - Preemptive kernel, priority inheritance, high resolution timer, etc
    - e.g., TimeSys Linux, Monta Vista Linux

Linux Scheduling

- Real-time scheduling class
  - Scheduled based on fixed priority
  - SCHED_FIFO: First-In-First-Out
  - SCHED_RR: Round-Robin
- Non-real-time scheduling class (SCHED_OTHER)
  - Priority is adjusted dynamically to favor I/O bound threads
- Default
  - Real-time: 0 – 99
  - Non-real-time: 100 – 139
Summary

• Operating systems
• Process states and management
• Context switch
• Process scheduling
• POSIX
• Real-Time OS
  • Proprietary kernels
  • Real-time extensions to general-purpose OS
    • Real-time Linux
    • Linux scheduling

Reading

• Textbook: Chapters 1 and 2