What is an Operating System?

Provides environment for executing programs:
- Process abstraction for multitasking/concurrency: Scheduling
- Hardware abstraction layer (device drivers)
- Filesystems
- Communication

We will focus on concurrency and real-time issues

Do I Need One?

Not always
Simplest approach: cyclic executive
for (;;)
  do part of task 1
  do part of task 2
  do part of task 3

Handling an Interrupt

1. Program runs normally
2. Interrupt occurs
3. Processor state saved
4. Interrupt routine runs
5. “Return from Interrupt” instruction runs
6. Processor state restored
7. Normal program execution resumes

Cyclic Executive

Advantages
- Simple implementation
- Low overhead
- Very predictable

Disadvantages
- Can’t handle sporadic events
- Everything must operate in lockstep
- Code must be scheduled manually

Interrupts

Some events can’t wait for next loop iteration:
- Communication channels
- Transient events

Interrupt: environmental event that demands attention
- Example: “byte arrived” interrupt on serial channel

Interrupt routine code executed in response to an interrupt
A solution: Cyclic executive plus interrupt routines

Cyclic Executive Plus Interrupts

Works fine for many signal processing applications
56001 has direct hardware support for this style

Insanely cheap, predictable interrupt handler:
- When interrupt occurs, execute a single user-specified instruction
- This typically copies peripheral data into a circular buffer
- No context switch, no environment save, no delay

Drawbacks of CE + Interrupts

- Main loop still runs in lockstep
- Programmer responsible for scheduling
- Scheduling static
- Sporadic events handled slowly

Interrupt Service Routines

Most interrupt routines do as little as possible
- Copy peripheral data into a buffer
- Indicate to other code that data has arrived
- Acknowledge the interrupt (tell hardware)

Additional processing usually deferred to outside
E.g., Interrupt causes a process to start or resume running
Objective: let the OS handle scheduling, not the interrupting peripherals
Cooperative Multitasking

A cheap alternative
Non-preemptive
Processes responsible for relinquishing control
Examples: Original Windows, Macintosh
A process had to periodically call `get_next_event()` to let other processes proceed
Drawbacks:
  Programmer had to ensure this was called frequently
  An errant program would lock up the whole system
Alternative: preemptive multitasking

Concurrency Provided by OS

Basic philosophy:
Let the operating system handle scheduling, and let the programmer handle function
Scheduling and function usually orthogonal
Changing the algorithm would require a change in scheduling
First, a little history

Batch Operating Systems

Original computers ran in batch mode:
Submit job & its input
Job runs to completion
Collect output
Submit next job
Processor cycles very expensive at the time
Jobs involved reading, writing data to/from tapes
Costly cycles were being spent waiting for the tape!

Timesharing Operating Systems

Way to spend time while waiting for I/O: Let another process run
Store multiple batch jobs in memory at once
When one is waiting for the tape, run the other one
Basic idea of timesharing systems
Fairness primary goal of timesharing schedulers
Let no one process consume all the resources
Make sure every process gets equal running time

Aside: Modern Computer Architectures

Memory latency now becoming an I/O-like time-waster.
CPU speeds now greatly outstrip memory systems.
All big processes use elaborate multi-level caches.

An Alternative:
Certain high-end chips (e.g., Intel's Xeon) now contain two or three contexts. Can switch among them “instantly.”
Idea: while one process blocks on memory, run another

Priority-based Scheduling

Typical RTOS has on fixed-priority preemptive scheduler
Assign each process a priority
At any time, scheduler runs highest priority process ready to run (processes can be blocked waiting for resources).
Process runs to completion unless preempted

Real-Time Is Not Fair

Main goal of an RTOS scheduler: meeting deadlines
If you have five homework assignments and only one is due in an hour, you work on that one
Fairness does not help you meet deadlines

Typical RTOS Task Model

Each task a triplet: (execution time, period, deadline)
Usually, deadline = period
Can be initiated any time during the period
Initiation
Execution Time
Deadline

\[ p = (2, 8, 8) \]

Example: Fly-by-wire Avionics

Hard real-time system with multirate behavior

- Gyros/accel
  - INU 1 kHz
  - Pitch ctrl. 500 Hz
- GPS
  - Throttle ctrl. 250 Hz
- Sensor
  - Air data 1 kHz
- Stick
  - Joystick 500 Hz
- Rudder
  - Elevator
  - Aileron
- Aileron 1
  - 1 kHz
- Aileron 2
  - 1 kHz
- Elevator
  - 1 kHz
- Rudder
  - 1 kHz

Example

Fly-by-wire Avionics

Hard real-time system with multirate behavior

- Gyros/accel
  - INU 1 kHz
  - Pitch ctrl. 500 Hz
- GPS
  - Throttle ctrl. 250 Hz
- Sensor
  - Air data 1 kHz
- Stick
  - Joystick 500 Hz
- Rudder
  - Elevator
  - Aileron
- Aileron 1
  - 1 kHz
- Aileron 2
  - 1 kHz
- Elevator
  - 1 kHz
- Rudder
  - 1 kHz
**Priority-based Preemptive Scheduling**

Always run the highest-priority runnable process

```
A | A | A
B   B   B
C   C
B   A   B   A   B
```

**Solutions to equal priorities**

- Simply prohibit: Each process has unique priority
- Time-slice processes at the same priority
  - Extra context-switch overhead
  - No starvation dangers at that level
- Processes at the same priority never preempt
  - More efficient
  - Still meets deadlines if possible

**Rate-Monotonic Scheduling**

Common way to assign priorities
Result from Liu & Layland, 1973 (JACM)
Simple to understand and implement:
  - Processes with shorter period given higher priority

<table>
<thead>
<tr>
<th>Period</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1 (high)</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>4 (low)</td>
</tr>
</tbody>
</table>

**Key RMS Result**

Rate-monotonic scheduling is optimal:
If there is a fixed-priority schedule that meets all deadlines, then RMS will produce a feasible schedule
Task sets do not always have a schedule
Simple example: \( P_1 = (10, 20, 20) \) \( P_2 = (5, 9, 9) \)
Requires more than 100% processor utilization

**When Is There an RMS Schedule?**

\[
U = \sum_i \frac{c_i}{p_i} 
\]

No schedule can possibly exist if \( U > 1 \) No processor can be running 110% of the time
Fundamental result: RMS schedule exists if
\[
U < n \left(2^{1/n} - 1\right)
\]
Proof based on case analysis (\( P_1 \) finishes before \( P_2 \))

**EDF Scheduling**

RMS assumes fixed priorities.
Can you do better with dynamically-chosen priorities?
Earliest deadline first:

*Processes with soonest deadline given highest priority*

**When Is There an RMS Schedule?**

Asymptotic result:
If the required processor utilization is under 69%,
RMS will give a valid schedule
Converse is not true. Instead:
If the required processor utilization is over 69%,
RMS might still give a valid schedule, but there is no guarantee

**RMS Missing a Deadline**

\( p_1 = (2, 4, 4) \), \( p_2 = (3, 6, 6) \), 100% utilization

```
1   1   1   1   1
2   2   2
1   2   1   2
```

Changing \( p_2 = (2, 6, 6) \) would have met the deadline and reduced utilization to 83%.
EDF Meeting a Deadline

\[ p_1 = (2, 4, 4), \quad p_2 = (3, 6, 6), \quad 100\% \text{ utilization} \]

Key EDF Result

Earliest deadline first scheduling is optimal:

If a dynamic priority schedule exists, EDF will produce a feasible schedule

Earliest deadline first scheduling is efficient:

A dynamic priority schedule exists if and only if utilization is no greater than 100%

Static Scheduling More Prevalent

RMA only guarantees feasibility at 69\% utilization, EDF guarantees it at 100\%

EDF is complicated enough to have unacceptable overhead

More complicated than RMA: harder to analyze

Less predictable: can't guarantee which process runs when

Priority Inversion

RMS and EDF assume no process interaction, often a gross oversimplification

Priority Inversion

Lower-priority process effectively blocks a higher-priority one

Lower-priority process's ownership of lock prevents higher-priority process from running

Nasty: makes high-priority process runtime unpredictable

Priority Inheritance

Solution to priority inversion

Increase process's priority while it possesses a lock

Level to increase: highest priority of any process that might want to acquire same lock

I.e., high enough to prevent it from being preempted

Danger: Low-priority process acquires lock, gets high priority and hogs the processor

So much for RMS

Summary

Cyclic executive—A way to avoid an RTOS

Adding interrupts helps somewhat

Interrupt handlers gather data, acknowledge interrupt as quickly as possible

Cooperative multitasking, but programs don't like to cooperate
Summary

Preemptive Priority-Based Multitasking—Deadlines, not fairness, the goal of RTOSes

Rate-monotonic analysis

- Shorter periods get higher priorities
- Guaranteed at 69% utilization, may work higher

Earliest deadline first scheduling

- Dynamic priority scheme
- Optimal, guaranteed when utilization 100% or less

Summary

Priority Inversion

- Low-priority process acquires lock, blocks higher-priority process
- Priority inheritance temporarily raises process priority
- Difficult to analyze