Outline

- Motivation for processes
- The process abstraction
- Context switching
- Multitasking
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

Why Multiple Processes?

Processes help us manage timing complexity
- multiple rates
  - multimedia
  - automotive

- asynchronous input
  - user interfaces
  - communication systems
Multi-rate Systems

- Tasks may be synchronous or asynchronous
- Synchronous tasks may recur at different rates
- Processes run at different rates based on computational needs of the tasks

Example: engine control

Tasks
- spark control
- crankshaft sensing
- fuel/air mixture
- oxygen sensor
- Kalman filter
- state machine
Typical Rates in Engine Control

<table>
<thead>
<tr>
<th>Variable</th>
<th>Full range time (ms)</th>
<th>Update period (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Engine spark timing</td>
<td>300</td>
<td>2</td>
</tr>
<tr>
<td>Throttle</td>
<td>40</td>
<td>2</td>
</tr>
<tr>
<td>Air flow</td>
<td>30</td>
<td>4</td>
</tr>
<tr>
<td>Battery voltage</td>
<td>80</td>
<td>4</td>
</tr>
<tr>
<td>Fuel flow</td>
<td>250</td>
<td>10</td>
</tr>
<tr>
<td>Recycled exhaust gas</td>
<td>500</td>
<td>25</td>
</tr>
<tr>
<td>Status switches</td>
<td>100</td>
<td>20</td>
</tr>
<tr>
<td>Air temperature</td>
<td>Seconds</td>
<td>400</td>
</tr>
<tr>
<td>Barometric pressure</td>
<td>Seconds</td>
<td>1000</td>
</tr>
<tr>
<td>Spark (dwell)</td>
<td>10</td>
<td>1</td>
</tr>
<tr>
<td>Fuel adjustment</td>
<td>80</td>
<td>8</td>
</tr>
<tr>
<td>Carburetor</td>
<td>500</td>
<td>25</td>
</tr>
<tr>
<td>Mode actuators</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>

Life Without Processes

- Code turns into a mess
  - interruptions of one task for another
  - spaghetti code

- Although it can still be developed under a strong discipline (e.g. round robin architecture)

```c
A_code();
...
B_code();
...
if (C) C_code();
...
A_code();
...
switch (x) {
    case C: C();
    case D: D();
    ...
```
Co-Routine Methodology

- Like subroutine, but caller determines the return address
- Co-routines voluntarily give up control to other co-routines
- Pattern of control transfers is embedded in the code
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

Processes and CPUs

- Activation record: copy of process state
- Context switch
  - current CPU context goes out
  - new CPU context goes in
Terms

- Thread (lightweight process): a process that shares memory space with other processes
- Reentrancy: ability of a program to be executed several times with the same results

Processes in POSIX

- Create a process with fork
  - parent process keeps executing old program
  - child process executes new program
fork()

- The fork process creates child:

  ```c
  childid = fork();
  if (childid == 0) {
      /* child operations */
  } else {
      /* parent operations */
  }
  ```

execv()

- Overlays child code:

  ```c
  childid = fork();
  if (childid == 0) {
      execv("mychild", childargs);
      perror("execv");
      exit(1);
  }
  ```
Context Switching

- Who controls when the context is switched?
- How is the context switched?

Co-operative Multitasking

- Improvement on co-routines
  - hides context switching mechanism
  - still relies on processes to give up CPU
- Each process allows a context switch call
- Separate scheduler chooses which process runs next
- Implementable with a function-queue architecture
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

Context Switching

- Must copy all registers to activation record, keeping proper return value for PC
- Must copy new activation record into CPU state
- How does the program that copies the context keep its own context?
Context Switching in ARM

- Save old process:

  STMIA r13,{r0-r14}^
  MRS r0,SPSR
  STMDB r13,{r0,r15}

- Start new process:

  ADR r0,NEXTPROC
  LDR r13,[r0]
  LDMDB r13,{r0,r14}
  MSR SPSR,r0
  LDMIA r13,{r0-r14}^
  MOVS pc,r14

Preemptive Multitasking

- Most powerful form of multitasking
- OS controls when contexts switches;
- OS determines what process runs next.
- Use timer to call OS, switch contexts
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
Why Not Use Interrupts?

- We could change the interrupt vector at every period, but
- We would need management code anyway
- We would have to know the next period’s process at the start of the current process

Evaluating Performance

- May want to test:
  - context switch time assumptions
  - scheduling policy
- Can use OS simulator to exercise process set, trace system behavior
Processes and Caches

- Processes can cause additional caching problems
  - Even if individual processes are well-behaved, processes may interfere with each other
- Worst-case execution time with bad cache behavior is usually much worse than execution time with good cache behavior

Outline

- Motivation for processes
- The process abstraction
- Context switching
- Multitasking

Next Class

- Scheduling policies