Processes and operating systems

- Motivation for processes.
- The process abstraction.
- Context switching.
- Multitasking.
- Processes and UML.

Why multiple processes?

- Processes help us manage timing complexity:
  - multiple rates
  - multimedia
  - automotive
  - asynchronous input
  - user interfaces
  - communication systems

Example: engine control

- Tasks:
  - spark control
  - crankshaft sensing
  - fuel/air mixture
  - oxygen sensor
  - Kalman filter

Life without processes

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

Co-routines

- Co-routine 1
  - ADR r14, co2a
  - co1a ...
  - ADR r13, co1b
  - MOV r15, r14
  - co1b ...
  - ADR r13, co1c
  - MOV r15, r14
  - co1c ...

Co-routine 2
  - co2a ...
  - ADR r13, co2b
  - MOV r15, r13
  - co2b ...
  - ADR r13, co2c
  - MOV r15, r13
  - co2c ...

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.

**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 at cswitch() call.
- Separate scheduler chooses which process runs next.

**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.
**Processes and UML**

- A process is an active class—-independent thread of control.

```
processClass1
  myAttributes
  myOperations()
    Signals
    start
    resume
```

**UML signals**

- Signal: object that is passed between processes for active communication:

```
  acomm: dataSignal
```

**Designing with active objects**

- Can mix normal and active objects:

```
p1: processClass1
  a: rawMsg
  w: wrapperClass
    ahat: fullMsg
    master: masterClass
```

**Processes and operating systems**

- Operating systems.

**Operating systems**

- The operating system controls resources:
  - who gets the CPU;
  - when I/O takes place;
  - how much memory is allocated.

- The most important resource is the CPU itself.
  - CPU access controlled by the scheduler.

**Process state**

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

- OS needs to keep track of:
  - process priorities;
  - scheduling state;
  - process activation record.
- Processes may be created:
  - statically before system starts;
  - dynamically during execution.

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.

Priority-driven scheduling

- Each process has a priority.
- CPU goes to highest-priority process that is ready.
- Priorities determine scheduling policy:
  - fixed priority;
  - time-varying priorities.

Priority-driven scheduling example

- Rules:
  - each process has a fixed priority (1 highest);
  - highest-priority ready process gets CPU;
  - process continues until done.
- Processes
  - P1: priority 1, execution time 10
  - P2: priority 2, execution time 30
  - P3: priority 3, execution time 20

Priority-driven scheduling example

- P3 ready t=18
- P2 ready t=0
- P1 ready t=15

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?
Process initiation disciplines

- **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.

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.

Example: Space Shuttle software error

- Space Shuttle's first launch was delayed by a software timing error:
  - Primary control system PASS and backup system BFS.
  - BFS failed to synchronize with PASS.
  - Change to one routine added delay that threw off start time calculation.
  - 1 in 67 chance of timing problem.

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

- 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:
    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

- Message passing on a network:
Process data dependencies

- One process may not be able to start until another finishes.
- Data dependencies defined in a task graph.
- All processes in one task run at the same rate.

Other operating system functions

- Date/time.
- File system.
- Networking.
- Security.

Processes and operating systems

- Scheduling policies:
  - RMS;
  - EDF.
- Scheduling modeling assumptions.
- Interprocess communication.
- Power management.

Metrics

- How do we evaluate a scheduling policy:
  - Ability to satisfy all deadlines.
  - CPU utilization—percentage of time devoted to useful work.
  - Scheduling overhead—time required to make scheduling decision.

Rate monotonic scheduling

- RMS (Liu and Layland): widely-used, analyzable scheduling policy.
- Analysis is known as Rate Monotonic Analysis (RMA).

RMA model

- All processes 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 $i$; $t_i$ is period of process $i$.

**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: $P_1, P_2, P_3, P_4$.
- Critical instant: $P_1, P_1, P_1, P_1$.

**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 example**

- Time: $0, 5, 10$.
- Processes: $P_1, P_2, P_1, P_1$.

**RMS CPU utilization**

- Utilization for $n$ processes is
  \[ \sum_{i=1}^{n} \frac{T_i}{t_i} \]
- As number of tasks approaches infinity, maximum utilization approaches 69%.
**RMS CPU utilization, cont'd.**
- RMS cannot use 100% of CPU, even with zero context switch overhead.
- Must keep idle cycles available to handle worst-case scenario.
- However, RMS guarantees all processes will always meet their deadlines.

**RMS implementation**
- Efficient implementation:
  - scan processes;
  - choose highest-priority active process.

**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 analysis**
- EDF can use 100% of CPU.
- But EDF may fail to miss a deadline.

**EDF implementation**
- On each timer interrupt:
  - compute time to deadline;
  - choose process closest to deadline.
  - Generally considered too expensive to use in practice.

**Fixing scheduling problems**
- What if your set of processes is unschedulable?
  - Change deadlines in requirements.
  - Reduce execution times of processes.
  - Get a faster CPU.
**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**

- Give priorities to system resources.
- Have process inherit the priority of a resource that it requests.
  - Low-priority process inherits priority of device if higher.

**Data dependencies**

- Data dependencies allow us to improve utilization.
  - Restrict combination of processes that can run simultaneously.
  - P1 and P2 can’t run simultaneously.

**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.

**Interprocess communication**

- OS provides interprocess communication mechanisms:
  - Various efficiencies;
  - Communication power.

**Signals**

- A Unix mechanism for simple communication between processes.
- Analogous to an interrupt—forces execution of a process at a given location.
  - But a signal is caused by one process with a function call.
- No data—can only pass type of signal.
### Signals in UML

More general than Unix signal—may carry arbitrary data:

```
<<signal>>
sSig
p : integer
```

```
<<send>>
someClass
sigbehavior()
```

---

### 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 behavior is usually much worse than execution time with good cache behavior.

---

### Power optimization

- **Power management**: determining how system resources are scheduled/used to control power consumption.
  - OS can manage for power just as it manages for time.
  - OS reduces power by shutting down units.
    - May have partial shutdown modes.

---

### Power management and performance

- Power management and performance are often at odds.
  - Entering power-down mode consumes
    - energy,
    - time.
  - Leaving power-down mode consumes
    - energy,
    - time.

---

### Simple power management policies

- **Request-driven**: power up once request is received. Adds delay to response.
  - **Predictive shutdown**: try to predict how long you have before next request.
    - May start up in advance of request in anticipation of a new request.
    - If you predict wrong, you will incur additional delay while starting up.
**Probabilistic shutdown**

- Assume service requests are probabilistic.
- Optimize expected values:
  - Power consumption;
  - Response time.
- Simple probabilistic: shut down after time $T_{\text{off}}$, turn back on after waiting for $T_{\text{off}}$.

**Advanced Configuration and Power Interface**

- ACPI: open standard for power management services.

**ACPI global power states**

- G3: mechanical off
- G2: soft off
  - S1: low wake-up latency with no loss of context
  - S2: low latency with loss of CPU/cache state
  - S3: low latency with loss of all state except memory
  - S4: lowest-power state with all devices off
- G1: sleeping state
- G0: working state