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.
Why multiple processes?

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

- Thermometer and heating element
  - Measure and update power at a rate (i.e. 100ms)
  - Hard real-time deadline
- Keypath: asynchronous
- Display: periodic, but soft real-time
- Can receive requests through network
  - New setpoint
  - Request data (temperature)
  - New control parameters
Life without processes

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

```cpp
A_code();
...
B_code();
...
if (C) C_code();
...
A_code();
...
switch (x) {
  case C: C();
  case D: D();
  ...
```
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. Reentrant code should avoid global variables, static local variables and other variables that could be modified while another thread is using it.
Context switching

- Who controls when the context is switched?
- How is the context switched?
Co-operative multitasking

- Hides context switching mechanism.
- Relies on processes to give up CPU.
- Each process allows a context switch at context_switch() 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?
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:
Flow of control with preemption
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.
A process can be in one of three states:
- \textcolor{red}{executing} on the CPU;
- \textcolor{red}{ready} to run;
- \textcolor{red}{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

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

P3 ready $t=18$
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 T**: interval between process activations.
- **Initiation interval or rate**: $1/T$
- **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.
Rate-monotonic scheduling

- N tasks: Periods $T_i$ and execution times $C_i$
- Assign higher priority to higher rate
- All tasks are guaranteed to complete before the end of their periods if

$$\sum \frac{C_i}{T_i} \leq Bound$$

Bound

- 1 for harmonic task set
- $n(\sqrt[n]{2-1})$
- $n=1$
- $\ln 2 = 0.69$ for large n
If bound is exceeded, an analysis of the specific task set is needed, to verify whether or not it will be schedulable.
RMA Example

- 3 harmonic tasks: \((T_i, C_i)\), \(P1=(4,1)\), \(P2=(5,2)\), \(P3=(20,5)\)

\[
\sum \frac{C_i}{T_i} = 0.9 \leq 1.0
\]

- Example: 2 tasks with \(P1=(2, 0.9)\), \(P2=(5, 2.3)\) Bound is exceeded (sum = 0.91, bound = 0.83) but can still be scheduled (P1 runs from 0 to .9, 2 to 2.9, 4 to 4.9, P2 runs from .9 to 2, 2.9 to 4, and 4.9 to 5.0)
Priority inversion

- A task H is forced to wait for a lower-priority task L because both need access to a resource that need to be protected. L grab the resource, then it is preempted by H which tries to grab the resource but can't; if a medium priority tasks M exist, it can also preempt L causing delay in L releasing the resource.

- Cure: Priority inheritance: L inherits priority of H so that M can not preempt L
Deadlock

- Mutual deadlocks can occur with the basic priority inheritance protocol.
- Task 1 wants to lock L1, then L2
- Task 2 tries to lock L2 and then L1
- Task 2 locks L2 first, before getting preempted by task 1, which then locks L1. Now, tasks 1 and 2 will be mutually deadlocked. This scenario can happen with any sequence of 2 or more tasks.
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.
Critical regions

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

- **Examples**: writing shared memory; accessing I/O device.
Semaphores

- **Semaphore**: controls access to critical regions and shared resources.

- **Protocol:**
  - Get access to semaphore with \( P() \)
  - Perform critical region operations.
  - Release semaphore with \( V() \).
  - If call to \( P() \) by process P1 fails because process P2 has taken the semaphore, P1 waits until P2 calls \( V() \) and releases the semaphore
Message passing

- Used instead of shared memory.
Other operating system functions

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