



Processes and Operating Systems

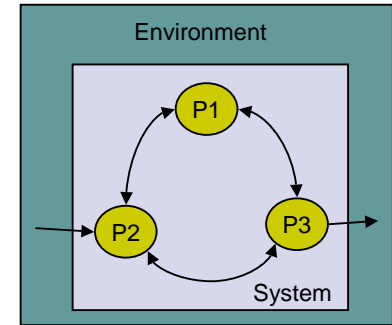
Ingo Sander
ingo@imit.kth.se

Based on slides by Wayne Wolf



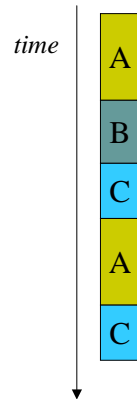
Why multiple processes?

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



Life without processes

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

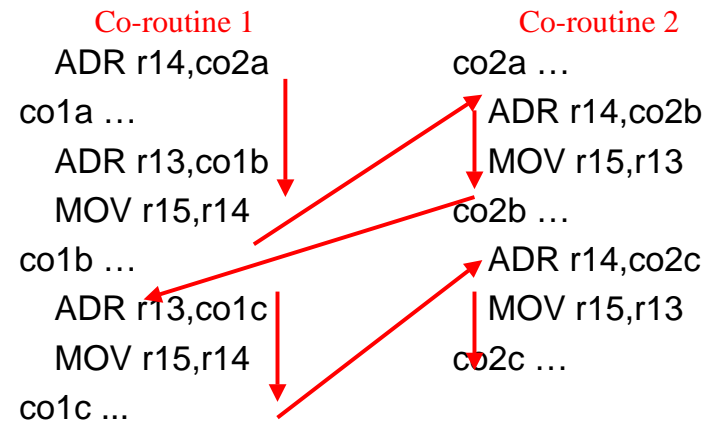


```

A_code();
...
B_code();
...
if (C) C_code();
...
A_code();
...
switch (x) {
  case C: C();
  case D: D();
  ...
}

```

Co-routines



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.

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

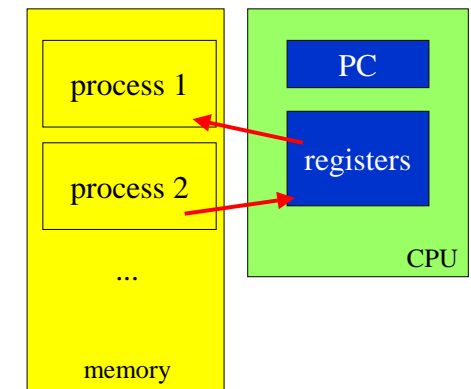


- A process is a **unique execution** of a program.
 - Several copies of a program may run simultaneously or at different times.
- A process has its own state:
 - registers;
 - memory.
- The operating system manages processes.

Processes and CPUs



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



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

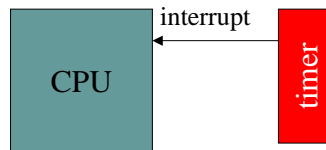


- Must copy all registers to activation record, keeping proper return value for PC.
- Must copy new activation record into CPU state.
- How does the program that copies the context keep its own context?

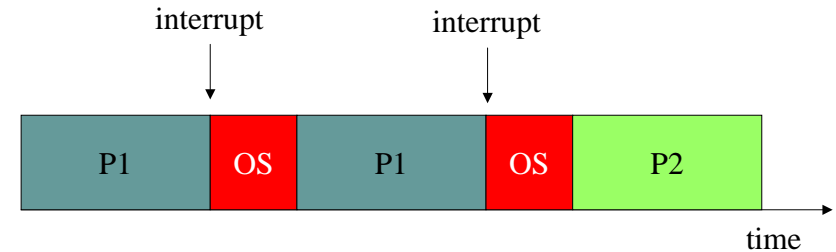
Preemptive multitasking



- 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.

Operating systems

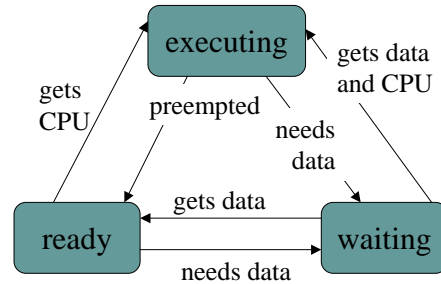


- The operating system controls resources:
 - who gets the CPU;
 - when I/O takes place;
 - how much memory is allocated.
- The most important resource is the CPU itself.
 - CPU access controlled by the scheduler.

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.

Scheduling



- Problem:
 - A set of tasks has to be implemented on one or more processors
 - Each task is accompanied by information (e.g. periodicity) and constraints (e.g. activation times).
- Objective:
 - To find a schedule that is *feasible*, i.e. a schedule where all requirements are met!

Scheduling



- In order to find a schedule, different algorithms have been proposed
- These algorithms use priorities in order to decide, which task is scheduled onto which processor at what time instant
- An *optimal* algorithm produces a feasible schedule, if it exists
- We only look at algorithm for single processors

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.

Our assumptions

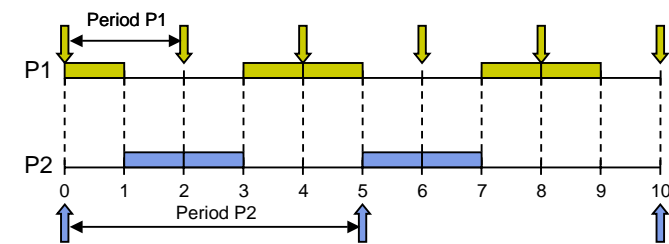


- Processes are periodic and are characterized by $P(c,p,d)$ where
 - c = compute time
 - p = period
 - d = deadline
- We assume that the period and the deadline are equal ($p = d$)!
- Processes may be preempted

First example



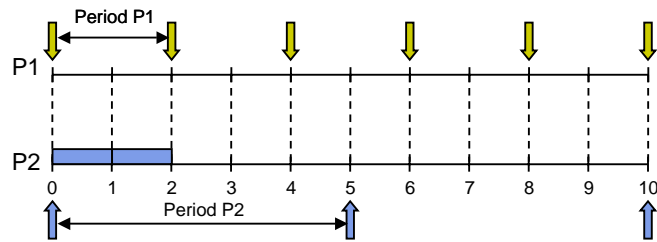
- $P1 = (1,2,2)$; $P2 = (2,5,5)$
- Policy: P1 has higher priority than P2



First example Changed priorities



- $P1 = (1,2,2)$; $P2 = (2,5,5)$
- Policy: $P2$ has higher priority than $P1$



P1 can not meet its first deadline!

Rate Monotonic Scheduling



- RMS assigns fixed priorities in reverse-order of period length
 - The shorter the period, the higher the priority
 - Intuition: Processes that require frequent action should receive higher priority
- RMS uses preemption
- RMS uses static priorities

Rate Monotonic Analysis (RMA) model

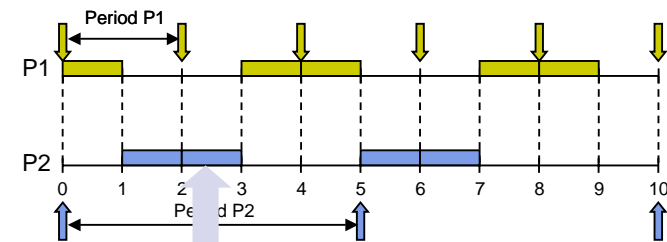


- All process run on single CPU
- Zero context switch time
- No data dependencies between processes
- Process execution time is constant
- Deadline is at end of period
- Highest-priority ready process runs

Is this an RMS schedule?



- $P1 = (1,2,2)$; $P2 = (2,5,5)$
- Policy: $P1$ has higher priority than $P2$

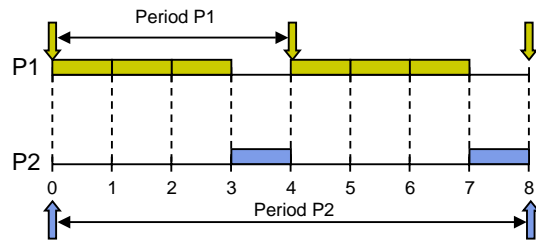


No! Preemption is not used!

Example Rate Monotonic Schedule



- $P1 = (3,4,4)$; $P2 = (2,8,8)$
- $P1$ has shorter period and is thus assigned higher priority!

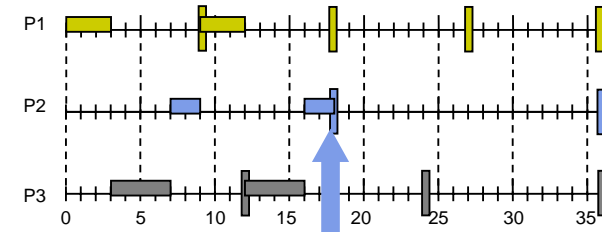


Deadlines are met => feasible schedule!

Another RMS Example



- $P1 = (3,9,9)$; $P2 = (5,18,18)$; $P3 = (4,12,12)$
- Priority: $P1 > P3 > P2$



P2 can not meet its first deadline!

Scheduling with dynamic priorities

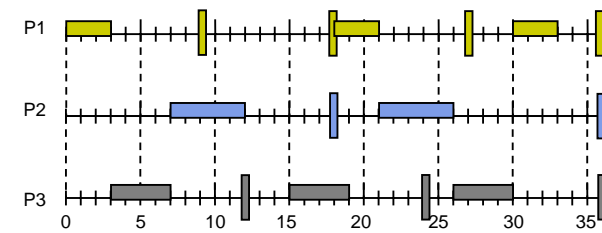


- RMS used static priorities
- Other algorithms use dynamic priorities, which give more flexibility, but are more complicated
- Example for dynamic priorities: *Earliest Deadline First (EDF)*
 - The schedule with the closest deadline is scheduled first

Example for EDF



- $P1 = (3,9,9)$; $P2 = (5,18,18)$; $P3 = (4,12,12)$
- Dynamic Priority



Feasible Schedule!

Properties of RMS



- If a schedule that meets the deadline with fixed priorities exists, RMS will produce a feasible schedule
- Scheduling with static priorities for n processors is always feasible, if the utilization

$$U \leq n(2^{1/n} - 1) \text{ where } U = \sum_{i=1}^n (c_i / p_i)$$

- Scheduling is always guaranteed, if the utilization is less than 69%

Properties for EDF



- Earliest Deadline First is optimal!
- Scheduling with dynamic priorities is feasible, if and only if $U \leq 1$

RMS implementation



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

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.

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.

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.

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

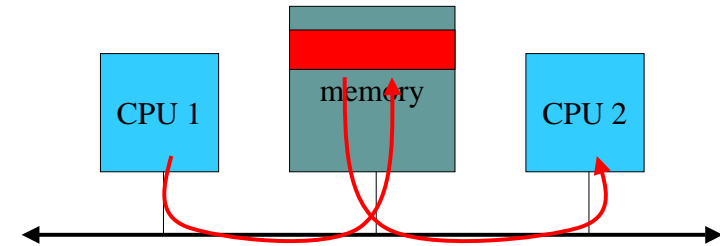


- Shared memory:
 - processes have some memory in common;
 - must cooperate to avoid destroying/missing messages.
- Message passing:
 - processes send messages along a communication channel---no common address space.

Shared memory



- Shared memory on a bus:



Critical Sections



- Simultaneous use of a shared resource can lead to disastrous consequences
- Resources that can only be used by one at a time, are called *serially reusable* and the use of the resource cannot be interrupted
- The code that accesses a serially reusable resource is called a *critical section* or *critical region*
- Thus there must be a mechanism that prevents a *collision*, i.e. the simultaneous use of such a resource)
- Examples:
 - Access to I/O-device
 - Writing to memory

Example Critical Regions



```
task1()
{
    printf("I am Task 1");
}

task2()
{
    printf("I am Task 2");
}
```

Possible Output: I am I am Task 2 Task1

Semaphores



- **Semaphore**: OS primitive for controlling access to critical regions.
- A variable S is used to lock a critical region!
- Protocol:
 - Get access to semaphore with **P(S)**.
 - Perform critical region operations.
 - Release semaphore with **V(S)**.
- P and V relate to the dutch words (Dijkstra)
 - "proberen" – "to test"
 - "verhogen" – "to wait"

Definitions of Access to Semaphores



```
void p(bool s)           void v(bool s)
{
    while (s == TRUE)    {
                        s := FALSE
                        ;
                        }
    s := TRUE;
}
```

Example Critical Regions



```
task1()
{
    p(s);
    printf("I am Task 1");
    v(s);
}
```

```
task2()
{
    p(s);
    printf("I am Task 2");
    v(s);
}
```

The critical region *printf(...)* is now protected by a semaphore!

Problems with Semaphores



In HLL:

```
void p(bool s)
{
    while (s == TRUE)
        ;
    s := TRUE;
}
```

In Assembler:

```
WHILE  ADR R1, S
        MOV R2, [R1]
        CMP R2, #1
        BEQ WHILE      ; S = TRUE?
        MOV R2, #1     ; S:= TRUE
```

- Assume process is interrupted between CMP and 2nd MOV by another routine that then unlocks the semaphore just a little bit earlier...
- At that moment R2 will have the value 0
- When the interrupted routine (this routine) continues, it sees the resource as free and will use it and may collide with another resource that also is allowed to use the resource => collision!

Atomic test-and-set



- Problem can be solved with an atomic test-and-set:
 - single bus operation reads memory location, tests it, writes it.
- In ARM test-and-set provided by SWP:


```
ADR r0,SEMAPHORE
LDR r1,#1
GETFLAG SWP r1,r1,[r0]
BNZ GETFLAG
```

 - SWP Rd, Rm, Rn swaps the contents between registers Rd, Rm and the memory location Rm
 - Rd = [Rn]; [Rn] = Rm
 - Rd = Rm => contents between Rd and [Rn] are swapped

September 27, 2004 2B1445 Embedded Systems 49

Deadlock



```
procedure Task A;
begin
  ...
  P(S);
  use_resource_1;
  ...
  P(R);
  use_resource_2;
  V(R);
  V(S);
end;
```

Task 1 waits for resource 2

```
procedure Task B;
begin
  ...
  P(R);
  use_resource_2;
  ...
  P(S);
  use_resource_1;
  V(S);
  V(R);
end;
```

Task 2 waits for resource 1

September 27, 2004

2B1445 Embedded Systems

50

Deadlock and Livelock



- In *deadlock* no process can satisfy its requirements because all are blocked
- In *livelock* (starvation) at least one process is satisfying its requirements, while one or more are not
 - Starvation occurs when a task does not receive sufficient resources to complete processing in its allocated time
- Deadlock is a serious problem because it cannot always be detected by testing and does only occur very infrequently, which makes it extremely dangerous!

September 27, 2004

2B1445 Embedded Systems

51

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.

September 27, 2004

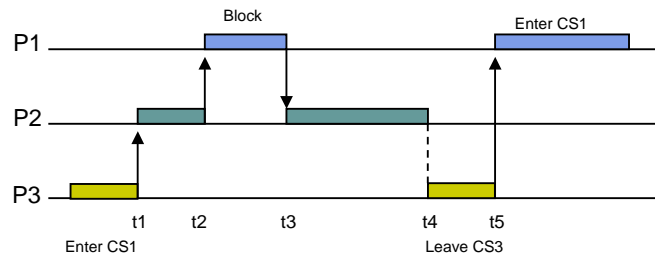
2B1445 Embedded Systems

52

Example for priority inversion



- Priority: $P1 > P2 > P3$
- P1 and P3 share resource R
P1:: begin ... lock(R);CS1;unlock(R) ... end;
P2:: begin End;
P3:: begin ... lock(R);CS3;unlock(R) ... end;



Possible solutions for priority inversion

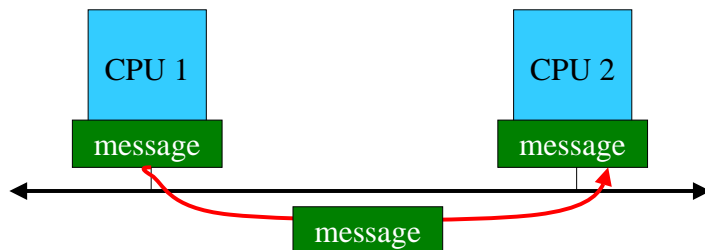


- Critical sections could be made non-preemptable
 - May work for short critical sections
- Critical sections are executed at the highest priority of any process that might use it
- Research has proposed solutions in form of priority inheritance protocols

Message passing



- Message passing on a network:



Message passing



- Message passing is an abstract communication scheme
- Processes communicate by sending and receiving messages
- A received message can initiate new actions (like an interrupt)
- Message passing can also be implemented on shared memory architectures

Message Passing by Mailbox



- Message Passing can be implemented by a "Mailbox-System"
- A memory location is defined which two or more processes can use to pass data
- Two operations are defined:
 - Procedure pend(var data, S:integer)
 - {data gets contents of mailbox S if any, suspend otherwise}
 - Procedure post(var data, S: integer)
 - {mailbox S gets data}
- The mailbox-system can be quite advanced in order to guarantee that no collisions occur
- The mailbox is similar to semaphores, processes who wait for "mail" are suspended when no mail is ready

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

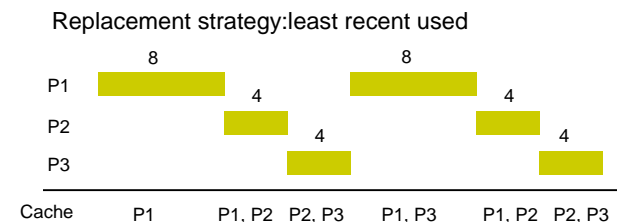


- Processes can cause additional caching problems.
 - Even if individual processes are well-behaved, processes may interfere with each other.
- Worst-case execution time with bad behavior is usually much worse than execution time with good cache behavior.

Example Cache Effects on Scheduling



Process	Worst Case	Average Case
P1	8	6
P2	4	3
P3	4	3

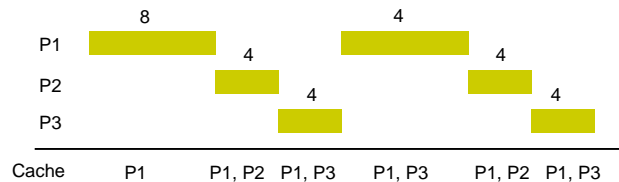


Example Cache Effects on Scheduling



Process	Worst Case	Average Case
P1	8	6
P2	4	3
P3	4	3

Replacement strategy: P1 is always in the cache!



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.

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_{on} , turn back on after waiting for T_{off} .