Eager Evaluation Considered Harmful

Blomdell, Anders

2000

Document Version:
Publisher's PDF, also known as Version of record

Eager evaluation considered harmful

Anders Blomdell

Department of Automatic Control
Lund Institute of Technology
Box 118 S-221 00 Lund
anders.blomdell@control.lth.se

Abstract In real-time systems potentially unbounded loops and lazy (late) evaluation are often avoided in order to improve predictability. This paper will show that a commonly suggested way to implement semaphores can lead to unnecessary blocking of high-priority tasks. It also presents a scheme with lazy evaluation and (potentially) unbounded loops that gives less blocking of high-priority tasks.

Keywords blocking, mutual exclusion, priority inversion, real-time kernel, scheduling, semaphore.

Introduction

Often, real-time blocking primitives like semaphores and monitors are coded in such a way that scheduling decisions are taken as soon as a task releases its lock on the resource. These eager (early) evaluations are done with the belief that system predictability will be improved and the processor load be decreased ("...the increment of a semaphore immediately followed by its decrement is avoided..." [Burns and Wellings, 1997, **p** 215]). A common way to implement semaphores [Dijkstra, 1968] is the following [Brinch Hansen, 1973], [Buttazzo, 1997]:

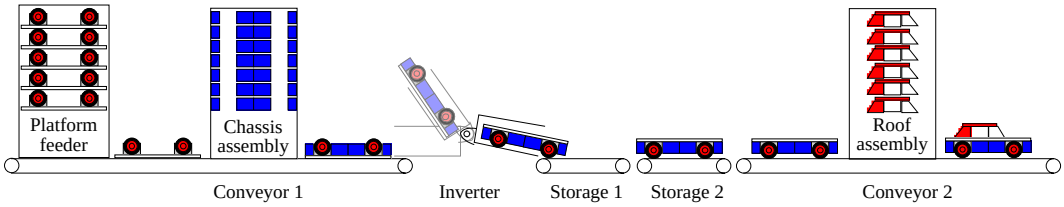
```
procedure wait(s : semaphore);

begin wait
  disable();
  if (s.count > 0) then
    s.count := s.count − 1;
  else
    insert(s.waiting, current);
    schedule();
  end if;
  reenable();
end wait;
```

```
procedure signal(s : semaphore);

begin signal
  disable();
  if (not empty(s.waiting)) then
    insert(readyqueue, first(s.waiting));
    schedule();
  else
    s.count := s.count + 1;
  end if;
  reenable();
end signal;
```

This paper presents a real-world situation where this implementation fails to give the expected timing, then the problem is analyzed and an alternative semaphore implementation is presented.

A Toy Example

In our lab we have a LEGO^{TM} car factory that assembles a number of LEGO bricks and prebuilt modules into finished LEGO cars.



At the far left of the factory, the *Platform feeder* loads prebuilt platforms on *Conveyor 1*. When a platform reaches the *Chassis assembly*, the conveyor is stopped and four LEGO bricks are mounted on the platform. The finished chassis is transported to the *Inverter* that moves it from its upside-down position to *Storage 1* conveyor. The chassis then passes *Storage 2* conveyor to *Conveyor 2* which transfers the chassis to the *Roof assembly*, where a roof is mounted. Finally the finished car is transferred out of the factory. In the figure the various motors and detectors are not shown. There are hard deadlines in the conveyor control, since failure to stop the transport or storage conveyors in time will lead to cars colliding or jamming the assembly stations.

The factory was recently equipped with a serial I/O system in order to be able to control it from an ordinary PC without any special I/O-cards. Due to physical reasons 3 digital I/O-units were used for the conveyors and 4 digital I/O-units were used for the feeder, assembly stations and inverter. The communication is done via the PC serial port (RS-232) at a speed of 9600 baud. The rest of this section will describe the software design for controlling the factory.

Since the software was intended to run under Windows NT, which only has a few priority levels available for each process, it was decided that control should be divided in two tasks; one high-priority task controlling the time critical conveyors and a lower-priority task controlling the feeder, assembly stations and inverter:

```
process ConveyorControl;

begin ConveyorControl
  t := now;
  loop
    conveyor_1_in := DigIn(1);
    storage_in := DigIn(2);
    conveyor_2_in := DigIn(3);
    Calculate conveyor control signals...
    DigOut(1, conveyor_1_out);
    DigOut(2, storage_out);
    DigOut(3, conveyor_2_out);
    t := t + T_conveyor;
    WaitUntil(t);
  end loop
end ConveyorControl;
```

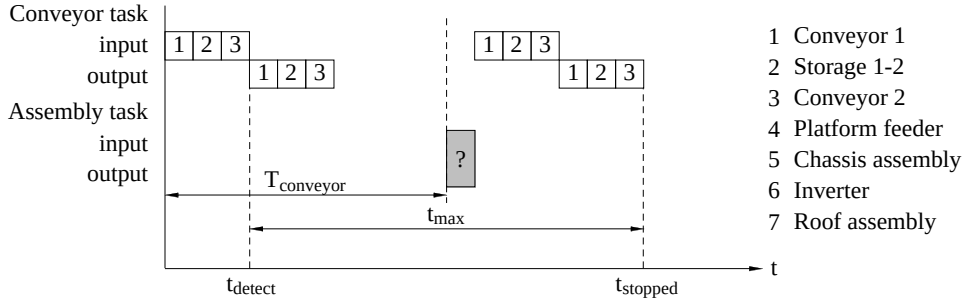
```
process AssemblyControl;

begin AssemblyControl
  t := now;
  loop
    feeder_in := DigIn(4);
    chassis_in := DigIn(5);
    inverter_in := DigIn(6);
    roof_in := DigIn(7);
    Calculate assembly control values...
    DigOut(4, feeder_out);
    DigOut(5, chassis_out);
    DigOut(6, inverter_out);
    DigOut(7, roof_out);
    t := t + T_assembly;
    WaitUntil(t);
  end loop
end AssemblyControl;
```

The serial port is a shared resource and is protected with a semaphore. The code for the digital I/O can briefly be described as:

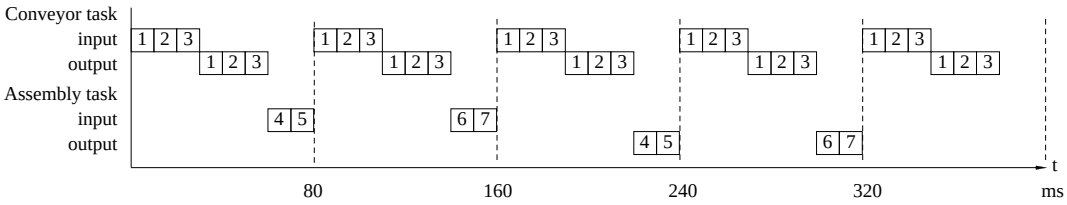
```
procedure DigIn(device : int) : byte;
begin DigIn
  wait(rs232semaphore);
    Send command to get data from device...
    Wait for data to arrive...
    Read result data...
  signal(rs232semaphore);
  Return result...
end DigIn;
```

```
procedure DigOut(device : int; data : byte);
begin DigOut
  wait(rs232semaphore);
    Send data to device...
    Wait for acknowledge to arrive...
  signal(rs232semaphore);
end DigOut;
```

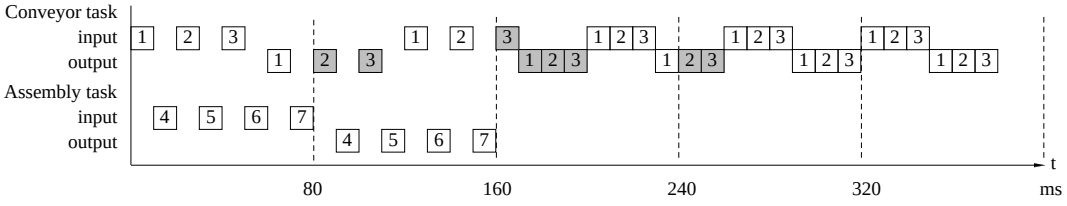
In the absence of higher priority tasks competing for the serial bus, the worst case timing for the conveyor control task occurs when: (a) a detector is triggered the moment after it has been polled; (b) the next sampling instant is delayed by an I/O-operation performed by the assembly control task.



Since the digital I/O-units require 10 bytes of transmitted data for each I/O-operation, each operation takes approximately 10 ms. Together with the the conveyor speed of 160 mm/s and the fact that conveyors has to be stopped within 20 mm from the detector we get $t_{\max} = \frac{20\,\text{mm}}{160\,\text{mm/s}} = 125\,\text{ms}$ and from the timing diagram we derive $T_{\text{conveyor}} \leq t_{\max} - 40\,\text{ms} = 85\,\text{ms}$. We then choose $T_{\text{conveyor}} = 80\,\text{ms}$ since the Windows NT time resolution is 10 ms. If we then schedule the lower-priority loop with $T_{\text{assembly}} = 400\,\text{ms}$ the total schedule for the RS-232 bus will be:



But when this system was implemented and run, we found that the system occasionally did misfeed cars. Logging of the activity revealed that the RS-232 bus was allocated as follows:



As we can see from the gray blocks of the high-priority process, 3 out of five deadlines are missed. It would be easy to assume that the use of a non-real-time operating system accounts for this error, but as the next section will show, it is

the textbook semaphore implementation that is the cause of the error.

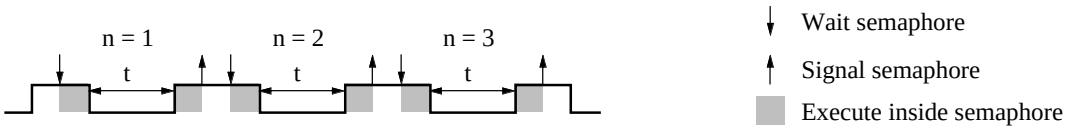
The Problem Analyzed

Let us consider what happens if we use the semaphore from the introduction to control access to some resource and the task is then blocked on some operation (e.g. protecting devices connected to a serial port and blocking on read/write operations). If multiple acquisitions are done from a high-priority task, the associated code could look like:

```
procedure acquire();

begin acquire
  wait(s); Claim device.
  Setup device for acquisition...
  Blocking wait for data to arrive.
  Acquire data...
  signal(s); Release device.
end acquire;
```

```
procedure highpriority();

begin highpriority
  for n := 1 to 3 do
    acquire();
  end for;
end highpriority;
```

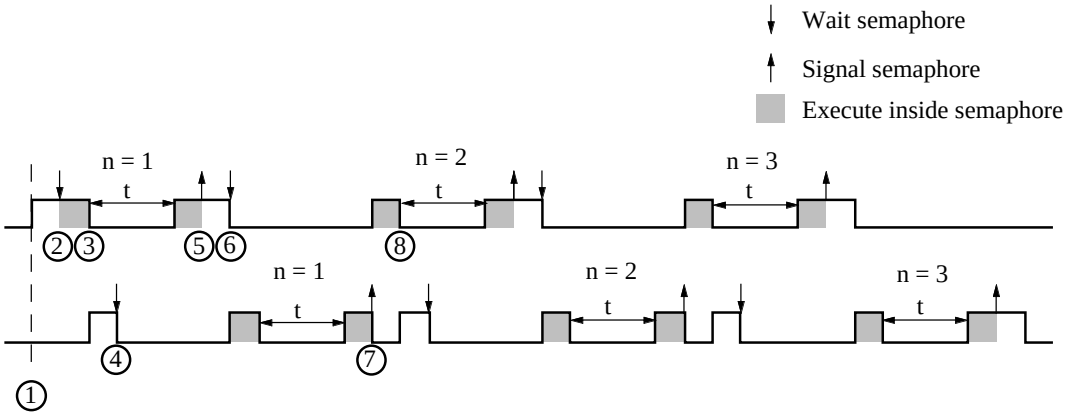
The associated timing diagram would then look like:



Now we introduce a similar low-priority task:

```
procedure lowpriority();

begin lowpriority
  for n := 1 to 3 do
    acquire();
  end for;
end lowpriority;
```

If the two tasks are released at the same time, one would intuitively think that the high-priority task would do all its acquisitions and then the low-priority task would do its acquisitions. But in reality we get the following behavior:



The sequence of events that leads to unexpected behavior is:

① Both tasks are released and the high-priority task (HP) starts executing.

② HP does a wait on the semaphore and enters the critical region.

③ HP blocks while waiting for data and the low-priority task (LP) starts executing.

④ LP does a wait on the semaphore, finds it occupied and is placed in the semaphore queue.

⑤ After finishing its wait for data, HP leaves the critical region by doing a signal on the semaphore, finds LP in the semaphore queue and grants LP the semaphore.

⑥ HP does a wait on the semaphore, finds it occupied and is placed in the semaphore queue. LP starts to execute.

⑦ LP signals the semaphore, finds HP in the semaphore queue and grants HP the semaphore. HP starts to execute.

⑧ The history repeats itself from ③.

The trouble starts at ⑤, where the high-priority task grants the semaphore to the low-priority task. The moment later (at ⑥), the high-priority task tries to grab the semaphore itself, but finds it occupied by the low-priority task and thus is forced to block until the semaphore is released.

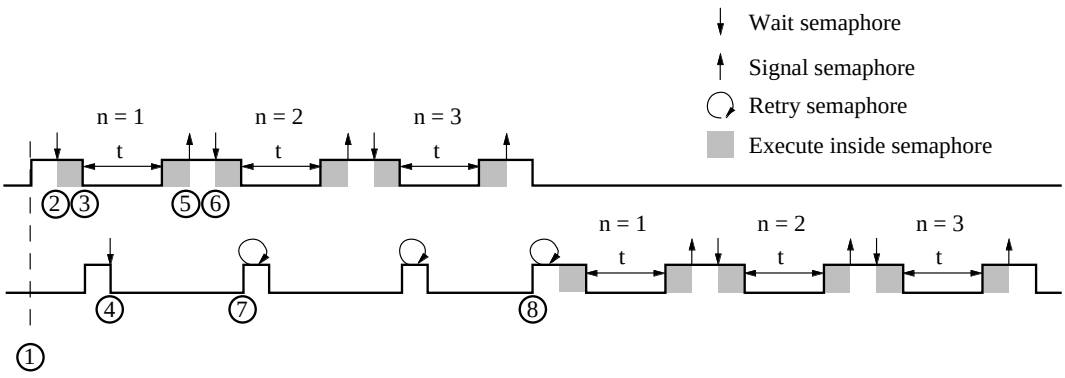
A Modest Proposal

The proposed way to correct this problem, is to postpone the decision of who gets the semaphore to the latest possible time. The code would then look like:

```
procedure wait(s : semaphore);

begin wait
  disable();
  while (s.count = 0) do
    insert(s.waiting, current);
    schedule();
  end while;
  s.count := s.count − 1;
  reenable();
end wait;
```

```
procedure signal(s : semaphore);

begin signal
  disable();
  s.count := s.count + 1;
  if (not empty(s.waiting)) then
    insert(readyqueue, first(s.waiting));
    schedule();
  end if;
  reenable();
end signal;
```

The timing diagram for the modified semaphore will then be:



① Both tasks are released and HP starts executing.

② – ④ See previous example.

⑤ After finishing its wait for data, HP leaves the critical region by doing a signal on the semaphore, increments the semaphore, finds LP in the semaphore queue and puts LP into the ready-queue.

⑥ HP does a wait on the semaphore, finds it free and enters the critical region a second time.

⑦ LP retries the semaphore, finds it occupied and is placed in the semaphore queue.

⑧ LP retries the semaphore for the third time, finds it free and can start its acquisition cycle.

The important feature of the presented solution, is that each task compete for the semaphore; i.e. no task grants a resource to another task. It is noteworthy that the proposed correction uses an unbounded loop, which is often considered unadvisable in real-time systems.

Conclusions

This paper has shown that improper implementation of blocking primitives for uni-processors can lead to the same kind of scheduling anomalies that frequently occurs on multi-processors.

Semaphores and priority based scheduling have been known and used by programmers for about 30 years. The combination of the two can lead to unexpected behavior (bugs). The implementation of both concepts is relatively simple, thus testing, code inspection and formal verification methods should have revealed the bug many times during these years. Still, at least one modern real-world system (i.e. Windows NT 4.0) contain the bug.

Nota Bene Even though this paper deals mostly with semaphores, it should be noted that all blocking primitives (e.g. monitors, mailboxes, etc) are suspectible to the problem. It must also be stressed that even though the effects of the bug resembles priority-inversion, the cause may well be present in the implementation of priority-inheritance and priority-ceiling protocols.

Bibliography

Brinch Hansen, P. (1973): *Operating System Principles*. Prentice-Hall, Englewood Cliffs, New Jersey.

Burns, A. and A. Wellings (1997): *Real–Time Systems and Their Programming Languages*. Addison–Wesley.

Buttazzo, G. (1997): *Hard Real–Time Computing Systems – Predictable Scheduling Algorithms and Applications*. Kluwer Academic Publishers.

Dijkstra, E. (1968): "Cooperating sequential processes." In Genuys, Ed., *Programming languages*. Academic Press N.Y.