ALBERT-LUDWIGS-
UNIVERSITÄT FREIBURG

Network Protocol Design and Evaluation

Exercise 6

Stefan Rührup

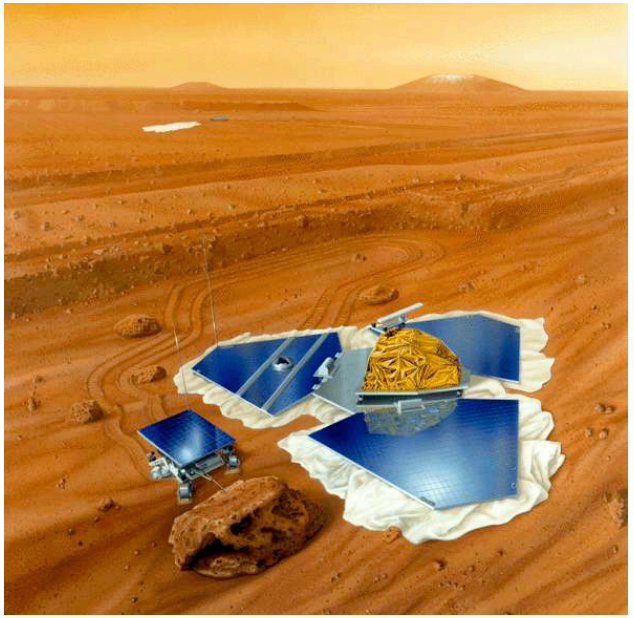
University of Freiburg
Computer Networks and Telematics

Summer 2009

Task 1

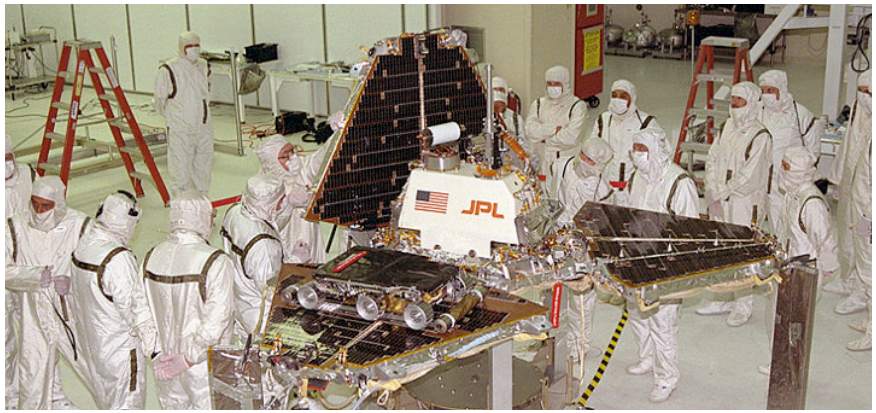
Task 1 *The Pathfinder Problem*

You are chief engineer at JPL and responsible for the Mars Pathfinder Mission. After the spacecraft has landed and released the rover, it is expected to transmit data to the earth. Unfortunately, the contact to the craft is lost at unpredictable moments. You suspect an automatic software reset after a process is blocked. There is a process for gathering meteorological data (low priority) and another process that consumes data (high priority). Both access an internal bus an use a semaphore that restricts the access.



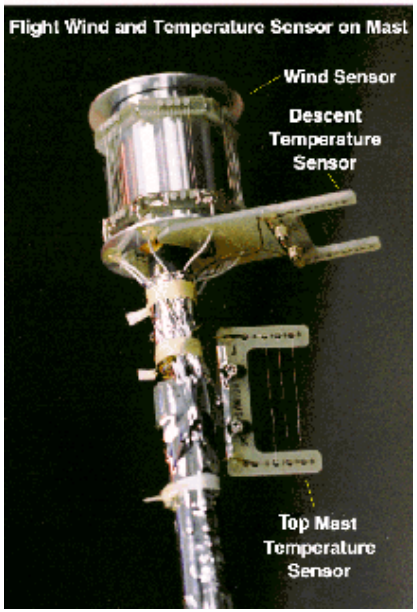
Mars Pathfinder

- Launched by NASA in Dec. 1996
- Landed on Mars on July 4, 1997
- Equipped with several instruments, e.g. the Atmospheric Structure Instrument/ Meterology (ASI/MET) Package
- ‘Low cost mission’: < $150 Million



Picture Credit: NASA

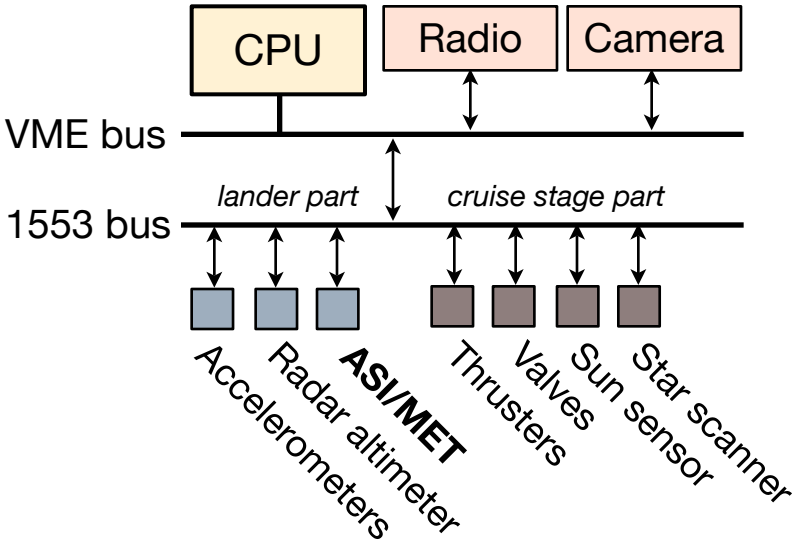
The ASI/MET system

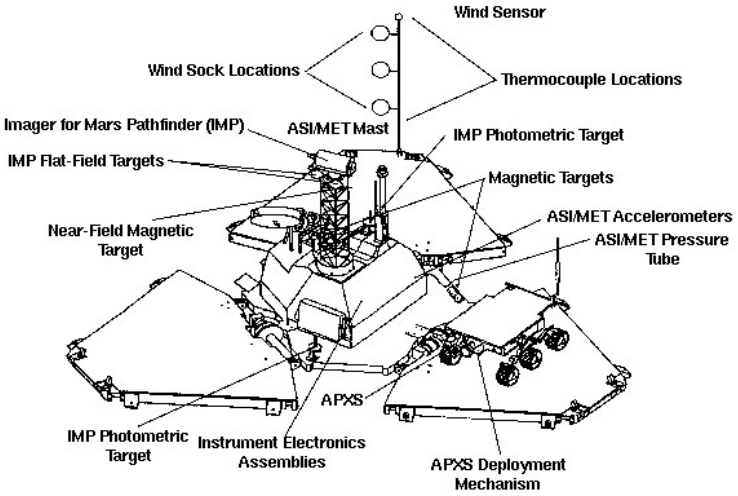


Picture Credit: NASA
http://marsprogram.jpl.nasa.gov/

The Pathfinder System (1)

Simplified system overview:

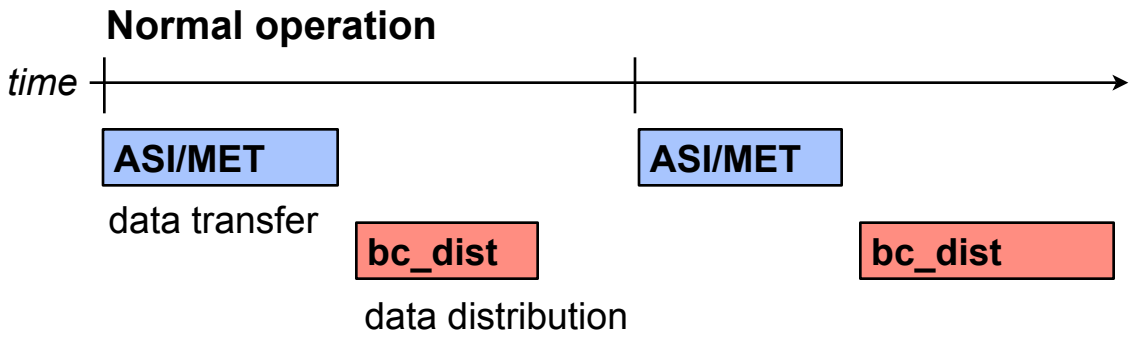




Picture Credit: NASA
http://marsprogram.jpl.nasa.gov/

The Pathfinder System (2)

- Data exchange on the 1553 bus:
  - Instruments, e.g. ASI/MET (low priority)
  - Data distribution process “bc_dist” (high priority)
  - Both share a resource (guarded by a semaphore)



The Promela Model

bc_dist

ASI/MET

```
mtype = { free, busy, idle, waiting, running };

show mtype h_state = idle;
show mtype l_state = idle;
show mtype mutex = free;

active proctype high() /* can run at any time */
{
end:  do
      :: h_state = waiting;
          atomic { mutex == free -> mutex = busy };
          h_state = running;

          /* critical section - consume data */

          atomic { h_state = idle; mutex = free }
     od
}

active proctype low() provided (h_state == idle)
{                              /* scheduling rule */
end:  do
      :: l_state = waiting;
          atomic { mutex == free -> mutex = busy};
          l_state = running;

          /* critical section - produce data */

          atomic { l_state = idle; mutex = free }
      od
}
```

pathfinder.pml (see SPIN's example directory)

Task 1.1

Task 1.1: Analyse the model with SPIN

- **Verification (check for invalid end states):**

```
>spin -a pathfinder.pml
>cc pan.c -o pan
>./pan
pan: invalid end state (at depth 3)
pan: wrote pathfinder.trail

(Spin Version 5.1.7 -- 23 December 2008)
Warning: Search not completed
     + Partial Order Reduction

Full statespace search for:
     never claim           - (none specified)
     assertion violations  +
     acceptance   cycles   - (not selected)
     invalid end states    +

State-vector 20 byte, depth reached 4, errors: 1
        5 states, stored
        1 states, matched
        6 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)
```

Task 1.1

- **Guided simulation:**

```
>spin -t -p pathfinder.pml
>Starting low with pid 0
Starting high with pid 1
  1:  proc  0 (low) line  27 "pathfinder" (state 1)    [l_state = waiting]
  2:  proc  0 (low) line  28 "pathfinder" (state 2)    [((mutex==free))]
  2:  proc  0 (low) line  28 "pathfinder" (state 3)    [mutex = busy]
  2:  proc  0 (low) line  29 "pathfinder" (state 4)    [l_state = running]
  3:  proc  1 (high) line  43 "pathfinder" (state 1)   [h_state = waiting]
spin: trail ends after 3 steps
#processes: 2
           h_state = waiting
           l_state = running
           mutex = busy
  3:  proc  1 (high) line  44 "pathfinder" (state 5)
  3:  proc  0 (low) line  33 "pathfinder" (state 8)
2 processes created
```

Task 1.1

- **Checking for livelocks:**

  First we set a progress state label

```
...

active proctype low() provided (h_state == idle)
{                              /* scheduling rule */
end:   do
       :: l_state = waiting;
          atomic { mutex == free -> mutex = busy};
progress: l_state = running;

          /* critical section - produce data */

          atomic { l_state = idle; mutex = free }
       od
}

...
```

Task 1.1

- **Check for non-progress cycles (without fairness):**

```
>spin -a pathfinder.pml
>cc pan.c -o pan -DNP
>./pan -l
pan: non-progress cycle (at depth 2)
pan: wrote pathfinder.trail

(Spin Version 5.1.7 -- 23 December 2008)
Warning: Search not completed
      + Partial Order Reduction

Full statespace search for:
      never claim           +
      assertion violations  + (if within scope of claim)
      non-progress cycles   + (fairness disabled)
      invalid end states    - (disabled by never claim)

State-vector 24 byte, depth reached 9, errors: 1
        5 states, stored
        0 states, matched
        5 transitions (= stored+matched)
        0 atomic steps
hash conflicts:         0 (resolved)
```

Task 1.1

- **Guided simulation:**

There is a cycle where the low priority process is suppressed

(can happen without fairness!)

```
>spin -t -p pathfinder.pml
Starting low with pid 0
Starting high with pid 1
spin: couldn't find claim (ignored)
  2:  proc  1 (high) line  43 "pathfinder" (state 1)   [h_state = waiting]
  <<<<<START OF CYCLE>>>>>
  4:  proc  1 (high) line  44 "pathfinder" (state 2)   [((mutex==free))]
  4:  proc  1 (high) line  44 "pathfinder" (state 3)   [mutex = busy]
  6:  proc  1 (high) line  45 "pathfinder" (state 5)   [h_state = running]
  8:  proc  1 (high) line  49 "pathfinder" (state 6)   [h_state = idle]
  8:  proc  1 (high) line  49 "pathfinder" (state 7)   [mutex = free]
 10:  proc  1 (high) line  43 "pathfinder" (state 1)   [h_state = waiting]
spin: trail ends after 10 steps
#processes: 2
            h_state = waiting
            l_state = idle
            mutex = free
 10:  proc  1 (high) line  44 "pathfinder" (state 4)
 10:  proc  0 (low) line  26 "pathfinder" (state 10) <valid end state>
2 processes created
>
```

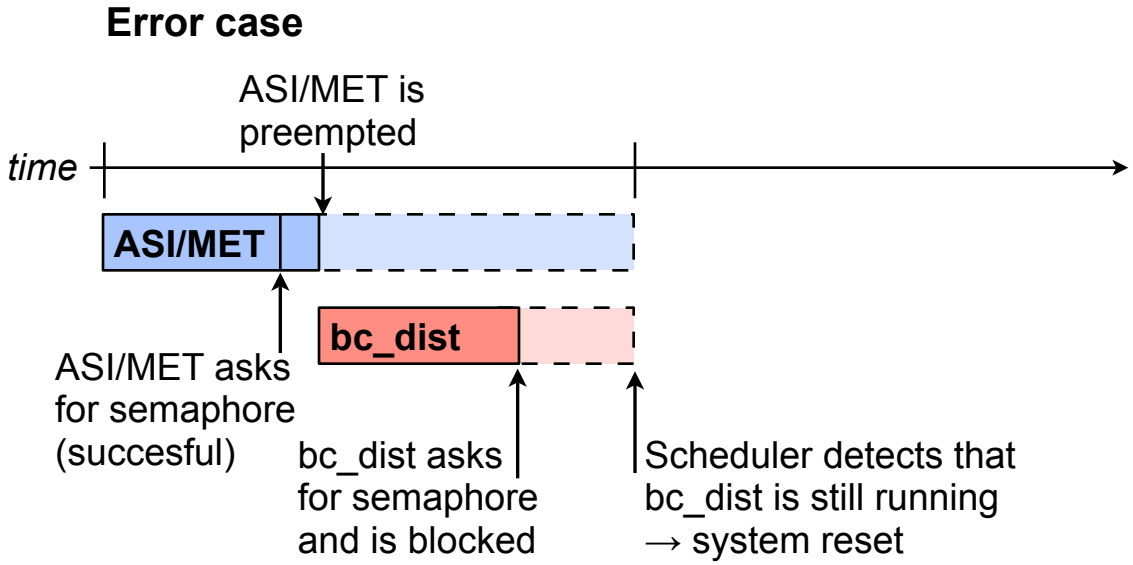
Task 1.1

- **Check for non-progress cycles again, with fairness:**

```
>spin -a pathfinder.pml
>cc pan.c -o pan -DNP
>./pan -l -f

(Spin Version 5.1.7 -- 23 December 2008)
      + Partial Order Reduction

Full statespace search for:
      never claim           +
      assertion violations  + (if within scope of claim)
      non-progress cycles   + (fairness enabled)
      invalid end states    - (disabled by never claim)

State-vector 24 byte, depth reached 20, errors: 0
       22 states, stored (33 visited)
       20 states, matched
       53 transitions (= visited+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

...
```

The Pathfinder Problem

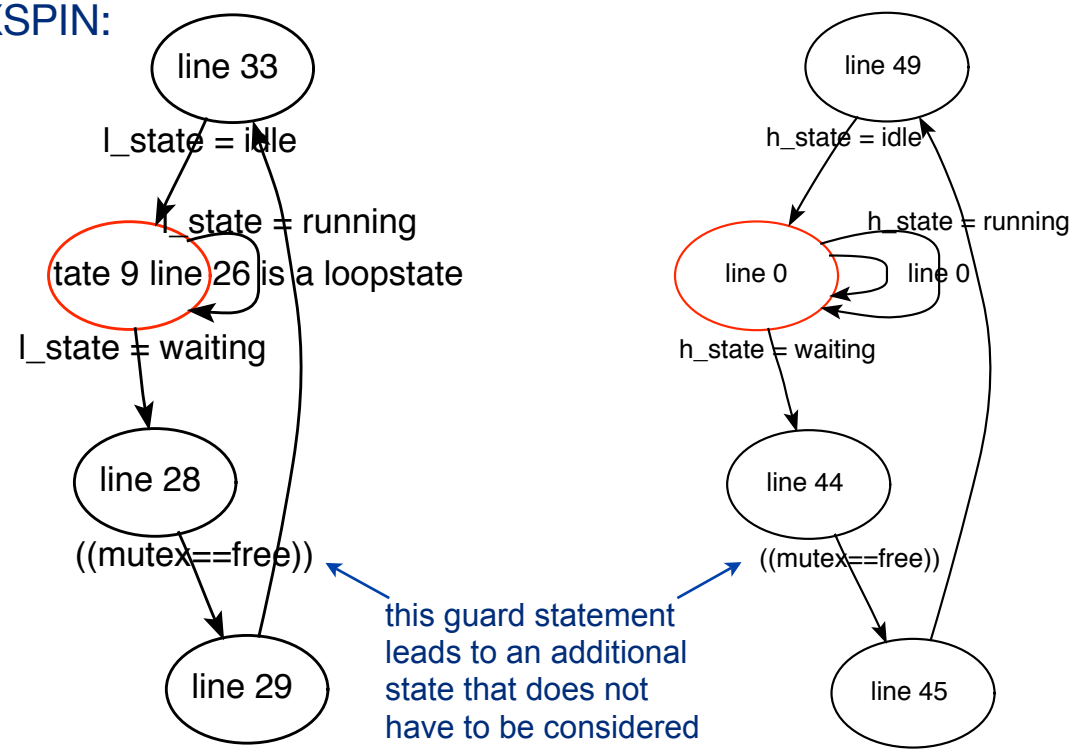
- The problem: Priority Inversion
  ASI/MET can be preempted by a higher priority process while still holding the semaphore. This leads to a deadlock.



Task 1.2

- Describe the processes in form of automata.

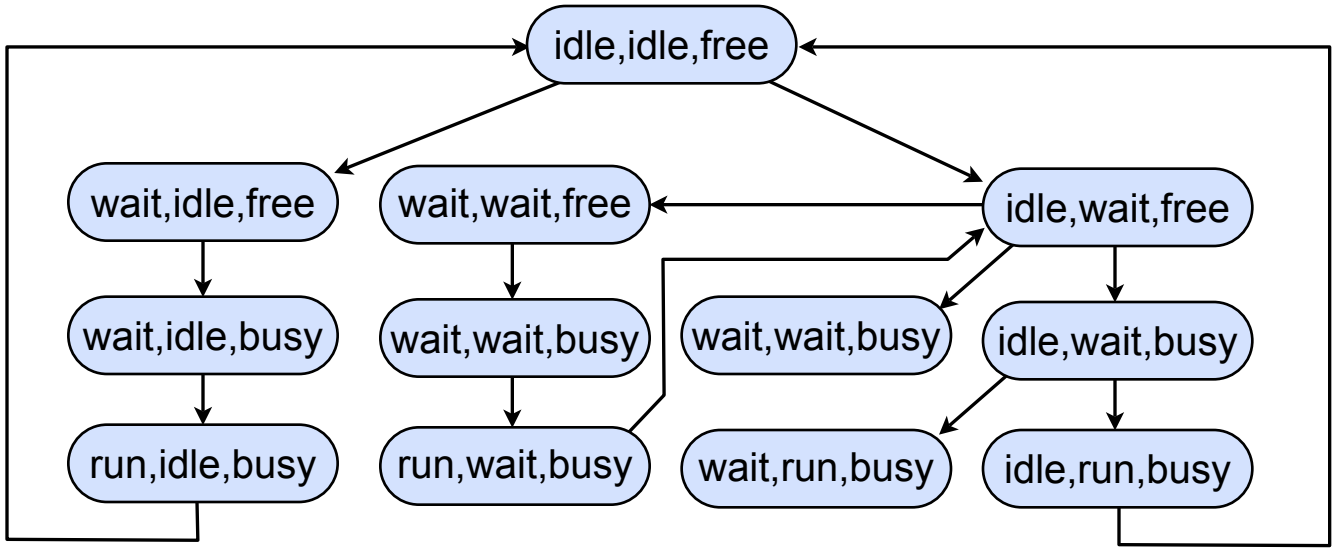
Generated with XSPIN:



Process states:
idle
waiting
running

Task 1.2

- Derive the complete state space for the two processes and the mutex state.

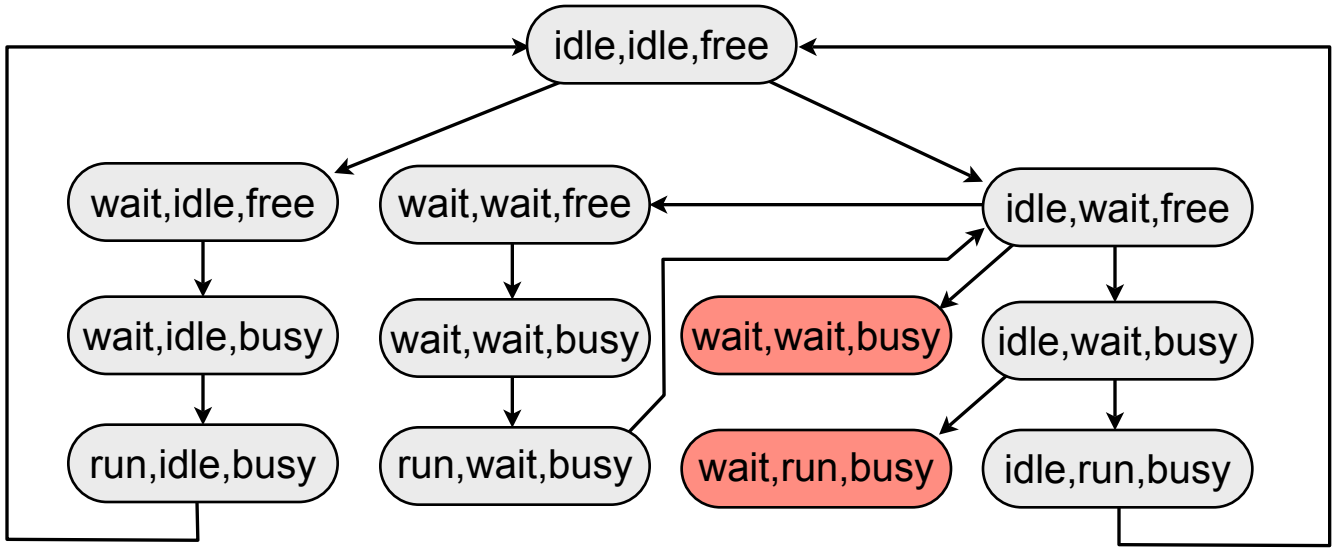


State description: (high, low, mutex state)

[Holzmann 2003]

Task 1.2

- **Deadlock states:**

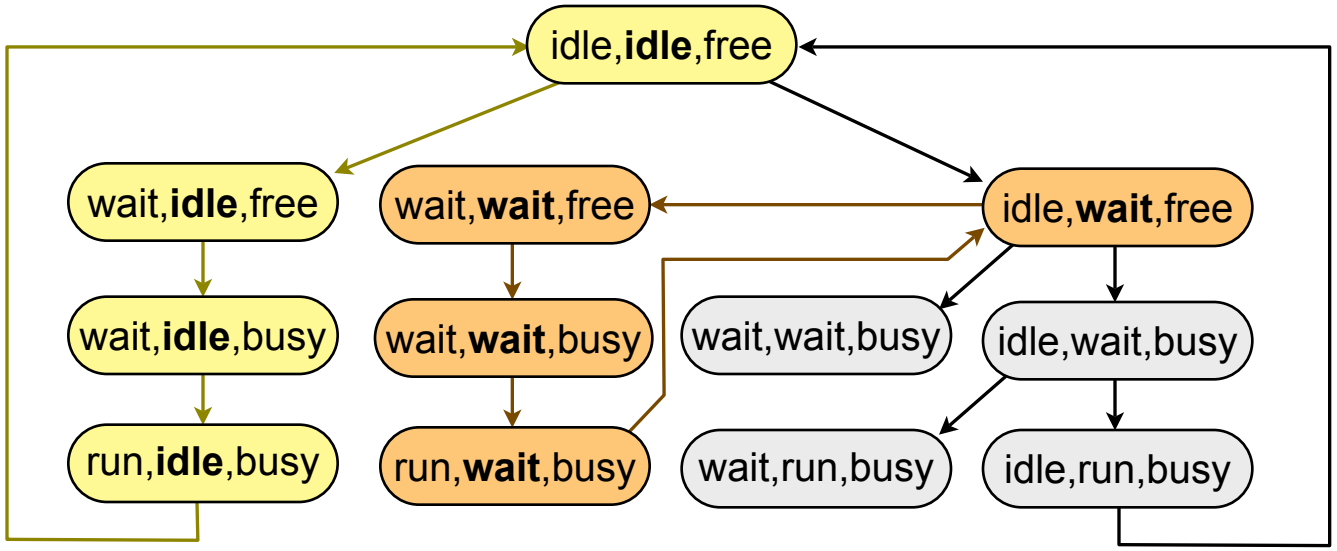


State description: (high, low, mutex state)

[Holzmann 2003]

Task 1.2

- **Non-progress cycles of the low priority process:**



State description: (high, low, mutex state)

[Holzmann 2003]

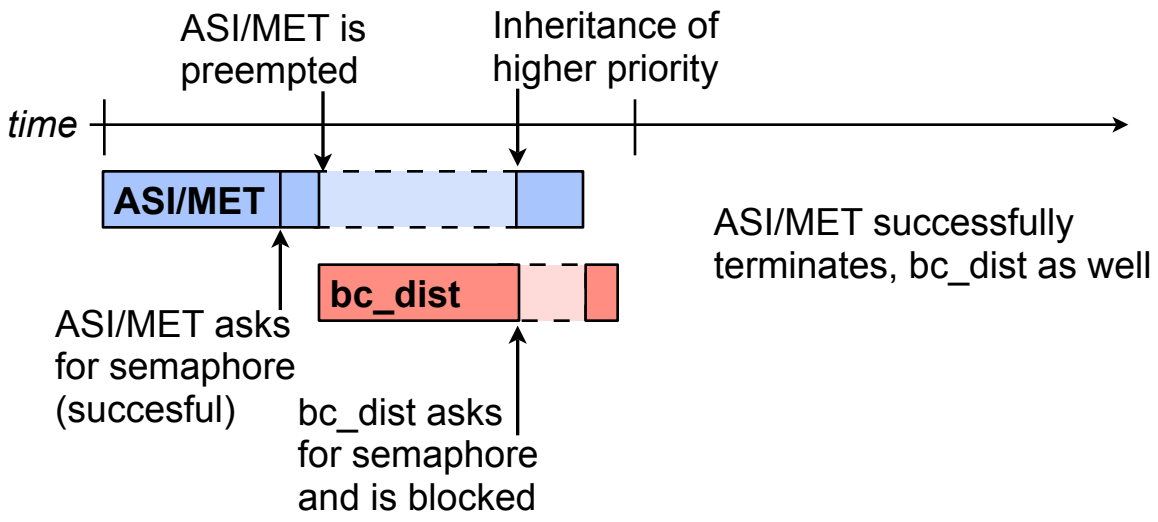
Solution of the Problem (1)

- Changes: The low priority process keeps on running once it enters the critical section

```
...

active proctype low() provided (h_state == idle)
{                              /* scheduling rule */
end:   do
       :: (h_state == idle) -> l_state = waiting;
          (h_state == idle) -> atomic { mutex == free -> mutex = busy};
progress: (h_state == idle) -> l_state = running;

          /* critical section - produce data */

          (h_state == idle) -> atomic { l_state = idle; mutex = free }
       od
}

...
```

Solution of the Problem (2)

- General solution: **Priority Inheritance**
  If a process blocks a higher priority process, it *inherits* the priority of the blocked process.



More Information

- **Glenn Reeves: “What really happened on Mars”**
  http://trs-new.jpl.nasa.gov/dspace/bitstream/2014/19020/1/98-0192.pdf
- Mike Jones’ page on the pathfinder problem
  http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/
- The Priority Inversion Problem
  http://en.wikipedia.org/wiki/Priority_inversion (see References therein)
- The validation model:
  G.J. Holzmann, “Designing Executable Abstractions”, 2nd Workshop on Formal Methods in Software Practice, 1998, pp.103-108.
  G.J. Holzmann, “The SPIN Model Checker”, Addison-Wesley, 2003

Task 2

Task 2 *LTL and Never Claims*

1. Specify the recurrence of a property p in LTL. Describe the Büchi automaton for the negated formula and give the corresponding never claim.

2. You want to check an invariant property p, but you have already used the never claim. You define another process by

```
active proctype invariant() {
    do :: assert(p) od
}
```

What is the problem of this solution (hint: think of timeouts)?
Is there an alternative?

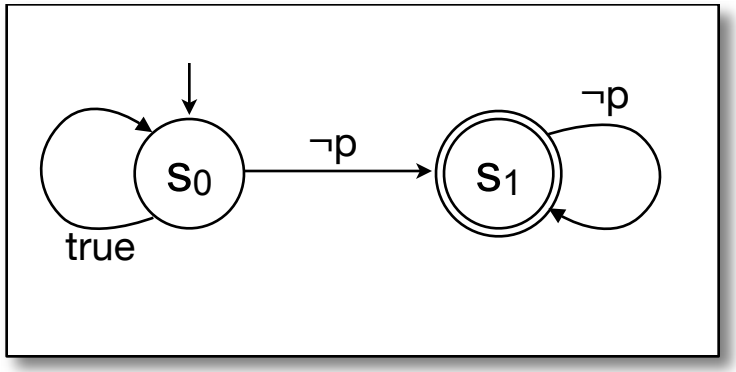
Task 2

Task 2.1 Specify the recurrence of a property p in LTL.

Recurrence: $\Box\Diamond p$ (Always eventually p, see templates)

Describe the Büchi automaton for the negated formula and give the corresponding never claim.

$\neg\Box\Diamond p \equiv \Diamond\neg\Diamond p \equiv \Diamond\Box\neg p$



Automaton for $\Diamond\Box\neg p$

```
never {
S0_init:
     if
     :: (!(p)) -> goto accept_S1
     :: true -> goto S0_init
     fi;
accept_S1:
     if
     :: (!(p)) -> goto accept_S1
     fi;
}
```

Task 2.2

Task 2.2 You want to check an invariant property p, but you have already used the never claim. You define another process by

```
active proctype invariant() {
    do :: assert(p) od
}
```

What is the problem of this solution?

The invariant process is always executable. Thus, deadlocks will remain undetected and timeout statements (if used) become never executable.

Task 2.2

Task 2.2 ... Is there an alternative?

The straightforward modification is the following. We leave out the do loop:

```
active proctype invariant() {
    assert(p)
}
```

This adds unnecessary overhead, because the validator has to check two steps (valid assertion and process termination) instead of one. A better alternative is to guard the assertion:

```
active proctype invariant() {
    atomic { !p -> assert(p) }
}
```

[http://spinroot.com/spin/Man/invariance.html]