The Santa Claus Problem
In occam

CSC 789
Multiparadigmatic Programming
(c) Matt Pedersen, 2011
1 The Santa Clause Problem

The Santa Clause problem was originally described by John Trono as an exercise in concurrency.

“Santa Claus sleeps in his shop at the North Pole, and can only be wakened by either all nine reindeer being back from their year long vacation on the beaches of some tropical island in the South Pacific, or by some elves who are having some difficulties making the toys. One elf’s problem is never serious enough to wake up Santa (otherwise, he may never get any sleep), so, the elves visit Santa in a group of three. When three elves are having their problems solved, any other elves wishing to visit Santa must wait for those elves to return. If Santa wakes up to find three elves waiting at his shop’s door, along with the last reindeer having come back from the tropics, Santa has decided that the elves can wait until after Christmas, because it is more important to get his sleigh ready as soon as possible. (It is assumed that the reindeer don’t want to leave the tropics, and therefore they stay there until the last possible moment. They might not even come back, but since Santa is footing the bill for their year in paradise . . . This could also explain the quickness in their delivering of presents, since the reindeer can’t wait to get back to where it is warm.) The penalty for the last reindeer to arrive is that it must get Santa while the others wait in a warming hut before being harnessed to the sleigh.”

In the paper in which Trono posed the problem, he also proposed a solution using locks and mutexes; a typical approach to concurrency control. However, Ben-Ari et al. showed that Trono’s original solution did not function correctly, and proposed a corrected version.

All though this problem seems simple at first, and perhaps to have no ties to real life concurrent programming problems, we believe it serves as a perfect exercise in concurrency. If we think of the output of the program as control signals to external devices, for example, control systems in nuclear power plants or airplanes, then all of a sudden certain restrictions might need to be enforced on this output. The simplest example might be the ordering of the output messages (i.e., the order in which the signals are sent to the external devices if we were modeling a power plant or an airplane). We might need to enforce a Lamport happen-before relation between some or all of the outputs.

As a result of such restrictions, the more output that must be ordered, the more complex the implementation gets, and the reasoning about correctness becomes much harder.
2 The Assignment

The goal of this exercise is to implement the Santa Clause Problem in a non-process oriented language and then afterwards in occam and try to reason about your implementation informally.

I have written some of the code so you do not have to start from scratch.

The main issues that you have to deal with in this assignment are, apart from not deadlocking the code, to make sure that the reporting is done in the correct order. We shall look at what the right order is in the following sections.

Firstly, let us start by determining the processes that we need in the system.

3 The Processes

The obvious processes that must be present in the system are of course

- The Santa process.
- 10 Reindeer processes.
- 9 Elf processes.

In addition we need a display process whose job it is to collect all the output from the various processes and print it to the screen. I have written this process for you; it is called display and is located in the display.occ file which is included at the top of the handout code (santa.occ).

All processes who wish to report any action must have the writing end of a shared reporting channel. The display process holds the reading end, and all it ever does is execute a read-and-print loop.

The protocol to which the messages sent to the display process must adhere is defined in the protocol.occ file, which is also included at the top of the handout code.

The definition for the report channel in the handout code looks like this:

```
SHARED ! CHAN MESSAGE report:
```

and the MESSAGE protocol extends REINDEER.MSG, ELF.MSG, and SANTA.MSG. Reindeer send REINDEER.MSG messages, elves send ELF.MSG messages, and Santa sends SANTA.MSG messages. As an example, if an elf wants to send a message to the display process that it is working, it could look like this:

```
CLAIM report!
  report ! working; id
```

Since report is shared at the writing end, it must be claimed before it can be accessed. Note that the value being sent is the ‘working’ kind, which must be followed by an INT, in this case the id of the elf.
3.1 Synchronization

There are a number of different kinds of synchronization that we need to consider for this problem. The two first ones are as follows:

- All reindeer must return from vacation **before** notifying Santa. This means that all 10 reindeer must synchronize before Santa can be notified that they are back from vacation. Since all reindeer processes must participate in this synchronization we can use a full barrier synchronization. A process can synchronize on a barrier (on which it is already enrolled) like this:

```
SYNC barrier.name
```

After this synchronization among the reindeer, they can proceed to introduce themselves to Santa, who will then harness them one at a time, take them out delivering toys for a while, then return and unharness them one at a time and let them go back on vacation.

- A total of **exactly** 3 elves must have queued up outside Santa’s door before they attract Santa’s attention. However, since we have 9 elves, but only need 3 we cannot use a full barrier synchronization, so we need to create our own partial barrier. We actually need 2 slightly different versions of the partial barrier: One which 3 elves can synchronize on, and one which the same 3 elves can synchronize with Santa on. Here is the code for the 2 different partial barriers:

```plaintext
PROC p.barrier.knock (VAL INT n, CHAN BOOL a?, b?, knock!)
    WHILE TRUE
    SEQ
        SEQ i = 0 FOR n -- accept n processes
        BOOL any:
        a ? any
        knock ! TRUE -- knock on Santa’s door
        SEQ i = 0 FOR n
        BOOL any:
        b ? any -- accept the same n processes
    :

PROC p.barrier (VAL INT n, CHAN BOOL a?, b?)
    WHILE TRUE
    SEQ
        SEQ i = 0 FOR n -- accept n processes
        BOOL any:
        a ? any
        SEQ i = 0 FOR n -- accept n processes
```

```
If a process wishes to synchronize on either of these barriers, it must execute the following code:

CLAIM a!
  a ! TRUE
CLAIM b!
  b ! TRUE

where a! and b! are the writing ends of either the channels associated with p.barrier or p.barrier.knock. Since this operation must be done often, we have wrapped it in procedure called sync, which can be called like this:

sync (elves.a!, elves.b!)

Naturally we need one p.barrier and one p.barrier.knock process running as well as the Santa, the 10 reindeer and the 9 elves.

Having described the barrier processes let us turn to the internal workings of the 3 major processes.

4 The Reindeer Process

A reindeer starts out by reporting that it is on holiday. After a random amount of time it returns to the North Pole. We can use a timer to wait for a random time in the following way:

wait, my.seed := random (HOLIDAY.TIME, my.seed)
tim ? t
tim ? AFTER t PLUS wait

After returning from holiday a deer reports itself ready, and then waits for all the other deer to return from holiday. Once that has happened it sends it id to Santa, and waits for Santa to harness everyone (This can be done with a full barrier in which Santa is also enrolled!). Once everyone is ready the deer reports that it is delivering toys, and again waits for Santa and the other deer to finish delivering (Again that is a full barrier synch with Santa and the other deer). Then each deer can report that they are done and send their id to Santa to be unharnessed before going back on holiday.

The skeleton code for a reindeer looks like this:
PROC reindeer (VAL INT id, -- unique id
    VAL INT seed, -- seed for the random
    number generator
    BARRIER just.reindeer, -- barrier for the 10
    deer
    BARRIER Santa.reindeer, -- barrier for the 10
    deer and Santa
    SHARED CHAN INT to.Santa!, -- channel to Santa
    SHARED CHAN REINDEER.MSG report! -- reporting channel)

INITIAL INT my.seed IS seed:
TIMER tim:
INT t, wait:
SEQ
    -- warm up the random number generator
    SEQ i = 0 FOR 1000
    wait, my.seed := random (HOLIDAY.TIME, my.seed)
    WHILE TRUE
        -- the deer code goes here
    :

5  The Elf Process

An elf starts out by reporting that it is working, which it does for a random
amount of time. When an elf wants to consult with Santa, it reports that it is
ready to do so, and tries to synchronize on the partial barrier executed by
p.barrier.knock by calling the sync(...) procedure. If he makes it through
the barrier it is because 2 other elves were ready as well and a 'knock' to Santa
has been accepted by Santa. The elf then sends his id to Santa and synchro-
nizes with Santa and the other 2 elves though the p.barrier process. Once
that has happened an elf can now report that he is consulting, which goes on for
as long as Santa wants. This is again modeled by a synchronization with Santa
and the other 2 elves using the sync(...) procedure. Once done consulting,
an elf can report that he is done, and send his id to Santa as a good-bye greeting.

The code skeleton for an elf looks like this:

PROC elf (VAL INT id, seed,
        SHARED CHAN BOOL elves.a!, elves.b!, -- for synchronizing 3 elves.
        SHARED CHAN BOOL Santa.elves.a!, Santa.elves.b!, -- 3 elves
        & Santa
        SHARED CHAN INT to.Santa!, SHARED CHAN ELF.MSG report!)
INITIAL INT my.seed IS seed:
TIMER tim:
INT t, wait:
SEQ
   -- warm up random number generator
SEQ i = 0 FOR 1000
   wait, my.seed := random (WORKING.TIME, my.seed)
WHILE TRUE
   -- elf code here
:

6 The Santa Process

Santa Claus start out sleeping; the only thing that can wake his is either all 10 reindeer back from holiday or 3 elves that need to consult about toy making. The reindeer have priority over the elves.

If all 10 reindeer are back from holiday they will each send Santa their id, so if he ever received one reindeer id he knows he will receive another 9. When receiving the first reindeer id, Santa reports that the deer are ready, and for each reindeer id he also reports that he is harnessing it. Once everyone has been harnessed (i.e., once Santa has passed the full barrier that he enrolls on with the deer), he can take them out to deliver presents, so he reports mush, mush :-). This delivering of presents takes a random amount of time, after which he says woah and returns to the Pole; he then unharnesses each deer as they send him their id.

If 3 elves are ready to consult, a knock on the door will happen from the p.barrier.knock process; once this knock has happened Santa reports the elves ready, and in turn receives the id of the 3 waiting elves and greets them. He then synchronizes with the 3 elves before reporting that he is consulting. This takes a random amount of time, after which Santa reports that he is done consulting, and he synchronizes with the elves yet again and then as he receives their id tells them goodbye. Once either of these actions are done he can go back to sleep.

The skeleton code for the Santa process looks like this:

PROC Santa (VAL INT seed,
           CHAN BOOL knock?, CHAN INT from.reindeer?, from.elf?,
           BARRIER Santa.reindeer,
           SHARED CHAN BOOL Santa.elves.a!, Santa.elves.b!,
           SHARED CHAN SANTA.MSG report!)
INITIAL INT my.seed IS seed:
    TIMER tim:
    INT t, wait:
SEQ
-- warm up the random number generator
SEQ i = 0 FOR 1000
  wait, my.seed := random (DELIVERY.TIME, my.seed)
WHILE TRUE
  -- Santa code
:

7 The System

The following illustration shows the wiring of the entire system. The thick lines represent full barriers.

The following code makes up the main process of the Santa Claus Problem:

PROC Santa.claus.problem (CHAN BYTE screen!)
  TIMER tim:
  INT seed:
  SEQ
    tim ? seed
    seed := (seed >> 2) + 42

  BARRIER just.reindeer, Santa.reindeer:
  SHARED ! CHAN BOOL elves.a, elves.b:
  CHAN BOOL knock:
  SHARED ! CHAN BOOL Santa.elves.a, Santa.elves.b:
  SHARED ! CHAN INT reindeer.Santa, elf.Santa:
  SHARED ! CHAN MESSAGE report:
PAR

    -- start all your processes here

Apart from passing in the full barriers as arguments, processes must be enrolled in the barrier when they are started. This can be done in two different ways. The first way is a\texttt{ PAR ENROLL barrier.name} which is like a regular\texttt{ PAR}, except any process within that par is enrolled on the barrier named \texttt{barrier.name}. The second way is to use a\texttt{ PAR ... FOR ... ENROLL barrier1, barrier2, ... , barriers}. The\texttt{ PAR ENROLL} can also take any number of barriers.

8 Questions

1. Implement the code in C/C++, Java or any other language. (you may do the next step and implement the solution in occam first.)

2. Implement the missing pieces of the code and get it running.

3. Can your code execute in a way such that Santa reports “Ho-ho-ho ... some elves are here!” before all three elves have reported “Need to consult Santa”?

4. Can you argue that your code can never execute in a way such that an elf never reports “Working” or a reindeer never reports “On holiday” until Santa has reported either “Goodbye elf” (in the elf case) or “Un-harnessing reindeer” (in the reindeer case)?

5. What part of the code assures that the reindeer report “all toys delivered ... want a holiday” before Santa reports “un-harnessing reindeer”?

9 Notes

The handout code can be found on the website along with a file containing some sample output from my solution. The next page shows an example of this output.
Elf 0: working, :)  
Elf 1: working, :)  
Elf 2: working, :)  
Elf 3: working, :)  
Elf 4: working, :)  
Elf 5: working, :)  
Elf 6: working, :)  
Elf 7: working, :)  
Elf 8: working, :)  
Elf 9: working, :)  

Reindeer 0: on holiday ... wish you were here, :)  
Reindeer 1: on holiday ... wish you were here, :)  
Reindeer 2: on holiday ... wish you were here, :)  
Reindeer 3: on holiday ... wish you were here, :)  
Reindeer 4: on holiday ... wish you were here, :)  
Reindeer 5: on holiday ... wish you were here, :)  
Reindeer 6: on holiday ... wish you were here, :)  
Reindeer 7: on holiday ... wish you were here, :)  
Reindeer 8: on holiday ... wish you were here, :)  

Elf 3: need to consult santa, :(  
Elf 9: need to consult santa, :(  
Reindeer 7: back from holiday ... ready for work, :(  
Reindeer 2: back from holiday ... ready for work, :(  
Elf 4: need to consult santa, :(  

Santa: Ho-ho-ho ... some elves are here!  
Santa: hello elf 3 ...  
Santa: hello elf 9 ...  
Santa: hello elf 4 ...  
 Elf: 3: about these toys ... ???  
 Elf: 9: about these toys ... ???  
 Elf: 4: about these toys ... ???  

Santa: consulting with elves ...  
Reindeer 8: back from holiday ... ready for work, :(  
Reindeer 3: back from holiday ... ready for work, :(  
Reindeer 4: back from holiday ... ready for work, :(  

OK, all done - thanks!  
Elf 3: OK ... we'll build it, bye ... :(  
Elf 9: OK ... we'll build it, bye ... :(  
Elf 4: OK ... we'll build it, bye ... :(  

Santa: goodbye elf 3 ...  
 Elf 3: working, :)  
Santa: goodbye elf 9 ...  
 Elf 9: working, :)  
Santa: goodbye elf 4 ...
10 display.occ

PROC display(CHAN MESSAGE in?, CHAN BYTE screen!)
  WHILE TRUE
    INT id:
    in ? CASE
    holiday; id
      SEQ
        out.string ("*t*t*t*t*t*Reindeer ", 0, screen!)
        out.int (id, 0, screen!)
        out.string (": on holiday ... wish you were here, :)c*n", 0, screen!)
    deer.ready; id
      SEQ
        out.string ("*t*t*t*t*t*Reindeer ", 0, screen!)
        out.int (id, 0, screen!)
        out.string (": back from holiday ... ready for work, :(c*n", 0, screen!)
    deliver; id
      SEQ
        out.string ("*t*t*t*t*t*Reindeer ", 0, screen!)
        out.int (id, 0, screen!)
        out.string (": delivering toys ... la-di-da-di-da-di-da, :)*c*n", 0, screen!)
    deer.done; id
      SEQ
        out.string ("*t*t*t*t*t*Reindeer: ", 0, screen!)
        out.int (id, 0, screen!)
        out.string (": all toys delivered ... want a holiday, :(c*n", 0, screen!)
    working; id
      SEQ
        out.string ("*t*t*Elf ", 0, screen!)
        out.int (id, 0, screen!)
        out.string (": working, :)c*n", 0, screen!)
    elf.ready; id
      SEQ
        out.string ("*t*t*Elf: ", 0, screen!)
        out.int (id, 0, screen!)
        out.string (": need to consult santa, :(c*n", 0, screen!)
    consult; id
      SEQ
        out.string ("*t*t*Elf: ", 0, screen!)
        out.int (id, 0, screen!)
        out.string (": about these toys ... ??*c*n", 0, screen!)
    elf.done; id
      SEQ
        out.string ("*t*t*Elf ", 0, screen!)
        out.int (id, 0, screen!)
        out.string (": OK ... we’ll build it, bye ... :(c*n", 0, screen!)

11
reindeer.ready
  out.string ("Santa: Ho-ho-ho ... the reindeer are back!*c*n", 0, screen!)
harness; id
  SEQ
    out.string ("Santa: harnessing reindeer ", 0, screen!)
    out.int (id, 0, screen!)
    out.string (" ... *c*n", 0, screen!)
mush.mush
  out.string ("Santa: mush mush ...*c*n", 0, screen!)
woah
  out.string ("Santa: woah ... we're back home!*c*n", 0, screen!)
unharness; id
  SEQ
    out.string ("Santa: un-harnessing reindeer ", 0, screen!)
    out.int (id, 0, screen!)
    out.string (" ... *c*n", 0, screen!)
elves.ready
  out.string ("Santa: Ho-ho-ho ... some elves are here!*c*n", 0, screen!)
greet; id
  SEQ
    out.string ("Santa: hello elf ", 0, screen!)
    out.int (id, 0, screen!)
    out.string (" ... *c*n", 0, screen!)
consulting
  out.string ("Santa: consulting with elves ...*c*n", 0, screen!)
santa.done
  out.string ("OK, all done - thanks!*c*n", 0, screen!)
goodbye; id
  SEQ
    out.string ("Santa: goodbye elf ", 0, screen!)
    out.int (id, 0, screen!)
    out.string (" ... *c*n", 0, screen!)

:
11 protocol.occ

PROTOCOL REINDEER.MSG
CASE
  holiday; INT
  deer.ready; INT
  deliver; INT
  deer.done; INT
 :

PROTOCOL ELF.MSG
CASE
  working; INT
  elf.ready; INT
  consult; INT
  elf.done; INT
 :

PROTOCOL SANTA.MSG
CASE
  reindeer.ready
  harness; INT
  mush.mush
  woah
  unharness; INT
  elves.ready
  greet; INT
  consulting
  santa.done
  goodbye; INT
 :

PROTOCOL MESSAGE EXTENDS REINDEER.MSG, ELF.MSG, SANTA.MSG: