How to Solve the Santa Claus Problem

Mordechai Ben-Ari
Department of Science Teaching
Weizmann Institute of Science
Rehovot 76100 Israel
ntbenari@wis.weizmann.ac.il, benari@acm.org

SUMMARY

John Trono published a new exercise in concurrent programming—the Santa Claus problem—and provided a solution based on semaphores [12]. His solution is incorrect because it assumes that a process released from waiting on a semaphore will necessarily be scheduled for execution. We give a simple solution in Ada 95 using higher order synchronization primitives: protected objects and rendezvous. We then give solution in Java, though this solution is not as elegant as the Ada 95 solution because the Java synchronization primitives are rather limited. The problem demonstrates that semaphores, designed for low-level mutual exclusion, are not appropriate for solving difficult concurrent programming problems.

INTRODUCTION

John Trono published a new exercise in concurrent programming [12] with the aim of providing educators with a more challenging problem than simple mutual exclusion: a problem with three distinct process types. His solution is rather complex, using ten semaphores and two global variables. Unfortunately, his solution is incorrect under the usual semantics of concurrent programming which require that a program be correct in *any* interleaving of the primitive statements of the processes [2]. Once a process has executed a semaphore instruction (either `P` or `Q`, there is nothing that requires that the process be scheduled for execution. Thus it is difficult to get a group of processes to execute "together", which is the behavior required by the problem specification.

There are semaphore algorithms that could probably be adapted to solve the problem [9], but these are extremely complex and not suitable for introductory

courses in concurrency. We give a simple solution using the more advanced synchronization primitives of Ada 95 [7]:

Protected object A monitor-like construct that enables a group of processes to be collected and then released together.

Rendezvous A synchronous construct that enables one process to wait simultaneously for more than one possible event.

Finally, we present a solution using so-called monitors in Java[1] [5]. In fact, Java synchronized methods are significantly less powerful than true monitors, and the solution is accordingly complex.

THE SANTA CLAUS PROBLEM

The problem:

> Santa Claus sleeps at the North pole until awakened by either all of the nine reindeer, or by a group of three out of ten elves. He performs one of two indivisible actions:
>
> - If awakened by the group of reindeer, Santa harnesses them to a sleigh, delivers toys, and finally unharnesses the reindeer who then go on vacation.
> - If awakened by a group of elves, Santa shows them into his office, consults with them on toy R&D, and finally shows them out so they can return to work constructing toys.
>
> A waiting group of reindeer must be served by Santa before a waiting group of elves. Since Santa's time is extremely valuable, marshalling the reindeer or elves into a group must not be done by Santa.

We now give an outline of the solution using semaphores, omitting the details of the calculations of the global variables and the semaphores protecting them. There is a single Santa process which waits on the semaphore `Santa`; it will be awakened by the ninth reindeer or the third elf. If awakened by the last reindeer, Santa executes as follows: awaken the eight other reindeer, harness all nine reindeer to the sleigh, deliver the toys and unharness the reindeer. The algorithm for the elves is similar.

```
loop
  P(Santa);
  if All_Reindeer_Ready then
    for All_Waiting_Reindeer loop
      V(Reindeer_Wait);
```

[1] Java is a trademark of Sun Microsystems, Inc.

```
    end loop;
    for All_Reindeer loop
      V(Harness);
    end loop;
    Deliver_Toys;
    for All_Reindeer loop
      V(Unharness);
    end loop;
  else   -- All_Elves_Ready
    for All_Waiting_Elves loop
      V(Elf_Wait);
    end loop;
    for All_Elves loop
      V(Invite_In);
    end loop;
    Consult;
    for All_Elves loop
      V(Show_Out);
    end loop;
  end if;
end loop;
```

Turning to the reindeer processes, the first eight reindeer returning from vacation are blocked on the semaphore `Reindeer_Wait`; the ninth reindeer wakes Santa who proceeds to awaken the other eight blocked reindeer. The elf processes are similar.

```
loop
  if Is_Last_Reindeer then
    V(Santa);
  else
    P(Reindeer_Wait);
  end if;
  P(Harness);
  Deliver_Toys;
  P(Unharness);
end loop;
```

The problem with this solution is the assumption that once the reindeer have been released from waiting on `Reindeer_Wait`, they will be harnessed and deliver toys *together with* Santa. But nothing prevents a malicious scheduler from scheduling Santa, who now proceeds to harness the reindeer, deliver the toys himself, and unharness the reindeer, before the reindeer have executed `Deliver_Toys` or even `P(Harness)`! Recall that `V` operations never block; a `V` operation on a strong semaphore will *release* blocked processes, but such ready

processes do not necessarily run. In fact, a truly miserly scheduler could now have Santa retire and consult indefinitely, to the consternation of the reindeer freezing in the stable.[2]

A SOLUTION IN ADA 95

This section shows how the concurrency primitives of Ada 95 can be used to develop an elegant solution to the Santa Claus problem. For a comprehensive treatment of concurrent programming in Ada 95, see [4].

There are two distinct problems that must be solved. The first is to synchronize a set of processes (reindeer or elves) and to release them as a group; the second is to show how the Santa process can provide more than one distinct service.

The solution to the first synchronization problem in Ada 95 uses *protected objects*. Protected objects are similar to monitors in that they consist of a set of variables and operations, encapsulated within the protected object so that the variables can be accessed only by these operations. The operations within the protected object are subject to mutual exclusion, and at most one can be executed by a task at any one time. Operations may be read-only functions (not subject to mutual exclusion), procedures or entries. A procedure is just executed under mutual exclusion, while an entry has a queue where calling processes wait if the entry is closed.

Classical monitors [6] use condition variables which must be explicitly signalled and waited upon. Ada 95 uses instead a construct similar to conditional critical section: a Boolean expression called a *barrier* is associated with each entry. If the barrier is true, a task calling the entry is allowed to execute the entry (subject to overall mutual exclusion on the entire protected object); if the barrier is false, tasks calling the entry are queued in FIFO order.[3] Upon completion of a protected operation, all barriers are re-evaluated; if any are true, one queued task waiting on one of the open barriers is released and allowed to execute the entry.

Due to the similarity of the code for the reindeer and the elves, we have generalized the solution to any `Group`. The protected object `Room` is enclosed within a generic package and instantiated for both reindeer and elves. `Team` is a generic parameter giving the group type.[4] `G` is the group name and is used to call one of the elements of an entry family in the Santa task.

```
generic
  type Team is (<>);
```

[2] Trono [Personal communication] notes that the problem can be fixed with another pair of semaphores.

[3] Other queueing policies, in particular priority-based policies, are possible; see Annex D Real-Time Systems of [7].

[4] The notation `(<>)` means that the generic actual parameter can be an enumeration type (such as the names of the reindeer), not just an integer type.

```
  G: in Groups;
```

The specification of the protected object `Room` contains an entry for processes to `Register` and a procedure `Open_Door` to re-open the entry after it has been closed. The private part, which is not visible to the calling processes, contains the state variables,[5] as well as an additional entry `Wait_for_Last_Member`, where calling processes are requeued pending the arrival of the last member of the group.

```
protected Room is
  entry Register;
  procedure Open_Door;
private
  entry Wait_for_Last_Member;
  Waiting:   Integer := 0;
  Entrance:  Door_State := Open;
  Exit_Door: Door_State := Closed;
end Room;
```

The implementation of the protected operations are in the the body of the protected object. All registering tasks that are members of the group, except the last one, immediately `requeue` on an additional, internal entry `Wait_for_Last_Member`, waiting for the barrier `Exit_Door` to open. The last member to register opens this barrier.

In the Santa Claus problem, it is particularly difficult to ensure that one group of elves does not overtake another. In this solution, the semantics of protected objects ensure that tasks blocked on a barrier have priority over new tasks attempting to enter the protected object. The last task to be released after requeueing wakes Santa and closes the barrier `Entrance` to ensure that no overtaking tasks join the group on the Santa server queue.

```
protected body Room is
  entry Register when Entrance = Open is
  begin
    Waiting := Waiting + 1;
    if Waiting < Group_Size then
      requeue Wait_for_Last_Member;
    else
      Waiting := Waiting - 1;
      Entrance := Closed;
      Exit_Door := Open;
    end if;
  end Register;
```

[5] One variable could be used for both barriers, but this formulation seems easier to explain.

```
  entry Wait_for_Last_Member when Exit_Door = Open is
  begin
    Waiting := Waiting - 1;
    if Waiting = 0 then
      Exit_Door := Closed;
      requeue Santa.Wake(Group);
    end if;
  end Wait_for_Last_Member;

  procedure Open_Door is
  begin
    Entrance := Open;
  end Open_Door;
end Room;
```

The `select`-statement is ideal for solving the second synchronization problem by implementing a server. The `select`-statement enables the server to block while waiting for more than one possible client. The semantics of the `select`-statement ensure that once a rendezvous has been commenced with one alternative of the statement, it will execute to completion. Thus, once Santa is wakened by the last reindeer, he will harness *all* the reindeer, deliver the toys and unharness the reindeer, before even attempting a rendezvous with the elves (and conversely).

```
loop
  select
    accept Wake(Reindeer_Group);
    for R in Reindeer loop
      accept Harness;
    end loop;
    Deliver_Toys;
    for R in Reindeer loop
      accept Unharness;
    end loop;
  or
    when Wake(Reindeer_Group)'Count = 0 =>
      accept Wake(Elf_Group);
    for E in Elf_Team loop
      accept Invite_In;
    end loop;
    Consult;
    for E in Elf_Team loop
      accept Show_Out;
    end loop;
  end select;
```

```
end loop;
```

The reindeer tasks are trivial: they simply call the protected object to register and then call the `Harness` and `Unharness` entries of the Santa process. The elf tasks are similar.

```
task body Reindeer_Task is
begin
  loop
    On_Vacation;
    Synchronize_Reindeer.Room.Register;
    Santa.Harness;
    Deliver_Toys;
    Santa.Unharness;
  end loop;
end Reindeer_Task;
```

PRIORITIES

We have not yet discussed giving priority to the reindeer.[6] Recall that in the semaphore solution, Santa initially releases the reindeer, but there is no guarantee that the reindeer actually deliver toys before elves consult Santa, because the malicious scheduler could choose to always schedule elves in preference to reindeer. We consider the last reindeer to be "in the stable" only after completing its final protected operation. We must ensure that (a) the last reindeer calls `Santa.Wake` without releasing mutual exclusion, and (b) that Santa initiates a rendezvous with the reindeer in preference to the elves.

The obvious solution to (a) is to have the last reindeer call the entry `Wake` in the Santa task.

```
if Waiting = 0 then
  Exit_Door := Closed;
  Santa.Wake(G);
end if;
```

Unfortunately, this is illegal in Ada because a *potentially blocking* operation, such as an entry call, cannot be called from within a protected operation.

The next obvious solution is to have the protected object return a parameter to the reindeer tasks so that the last reindeer to be released can then call `Santa.Wake`. This is also not acceptable, because it is possible that between the return from the protected object and the call of `Santa.Wake`, the elves will be scheduled and awaken Santa.

The correct solution is to use the `requeue`-statement.

[6] As usual in concurrent programming, we solve the problem using synchronization primitives and not process priorities.

```
if Waiting = 0 then
  Exit_Door := Closed;
  requeue Santa.Wake(G);
end if;
```

The meaning of this `requeue`-statement is that the last task which had been blocked on the entry `Wait_for_Last_Member` of the protected object, is transferred to the queue for the entry `Santa.Wake(G)` of the Santa task. Since a task can be requeued only on an entry with either no parameters or with the same parameter signature as the original call, no *new* blocking operation is needed. The task is requeued *before* the end of the protected action contained the entry call, so race conditions are prevented.

To solve (b) we use a *guard* on the accept statement for the elves.

```
when Wake(Reindeer_Group)'Count = 0 =>
  accept Wake(Elf_Group);
```

Once the last reindeer calls `Santa.Wake`, when Santa is scheduled he will rendezvous with the reindeer even if the last of a group of elves has previously called `Santa.Wake`.

However, it is still possible to construct a race condition in which elves will overtake the reindeer. Suppose that the last of a group of elves has called `Santa.Wake`, then `Santa` evaluates the guard finding that no reindeer task is waiting on `Santa.Wake`. Now the reindeer are scheduled and the last reindeer finally calls `Santa.Wake`. Since guards in `select`-statements (unlike barriers in protected objects) are not re-evaluated, the rendezvous may be with the elves, even though the reindeer are also ready. The reason is that the default choice between open alternatives is arbitrary.

We must ensure that the Santa process gives priority to the call from a reindeer process even after checking the guards. In Ada 95, a compiler that implements the Annex D Real-Time Systems supports the specification of a *queueing policy*, in particular, one in which alternatives are tried in textual order.[7]

```
pragma Queueing_Policy(Priority_Queueing);
```

TERMINATION

The simplest way to terminate the program is to have the main program execute `abort` on all the tasks. Note that you will have to modify the `requeue`-statement to include the `with abort` clause; otherwise, reindeer and elves that are waiting for their group to assemble will not be aborted.

The Santa task need not be explicitly aborted; since it is a server task, a `terminate` alternative can be used.

[7] This is similar to `PRI ALT` of occam.

```
select
  accept Wake(Reindeer_Group);
    ...
or
  when Wake(Reindeer_Group)'Count = 0 =>
    accept Wake(Elf_Group);
      ...
or
  terminate;
end select;
```

Graceful termination would be more difficult to achieve. `Santa_Task` is relatively easy to terminate using an explicit call from the main program to an additional alternative in the `select`-statements. Normally, one can depend on the semantics of Ada to terminate the calling (client) tasks: when the server has terminated, the exception `Tasking_Error` is raised in the calling tasks, and they terminate gracefully. However, if the client task is blocked on an entry queue for a protected object, it must first be released; this would require some additional programming.

GENERALIZATION

The solution easily generalizes to include synchronization with additional groups such as suppliers, tax auditors and safety inspectors. The generic package containing the protected object must be instantiated for each additional group, and the family of entries extended for additional alternatives in the `select`-statement in the Santa task. This is a well-know Ada technique; see [1], p. 436.

A SOLUTION IN JAVA

Java [5] is a new object-oriented language developed at Sun Microsystems, Inc. While superficially similar to C++, Java does not contain dangerous features inherited from C, such as explicit pointers, unchecked array access and file scope.

In addition to the language, the developers defined the *Java Virtual Machine (JVM)*, designed to ensure portability of Java programs. Note that the Java language and the JVM are two independent entities. For example, a compiler from Ada 95 to the JVM has been released [11].

Java includes constructs for defining and synchronizing concurrent processes. A class that is derived from the class `Thread` can be `start`'ed, creating a concurrent thread of execution. Synchronization is achieved either by executing `suspend` and `resume` directly on a thread, or indirectly, by thread executing `wait` and `notify` within a synchronized object. A method declared as `synchronized` is executed under mutual exclusion: only one thread at a time can be executing

any of the `synchronized` methods of an object. A thread that executes `wait` is blocked until another thread executes `notify`.

Java uses *monitors* ([5], page 399) to synchronize threads. However, there are significant differences between classical monitors ([6], see also [2]) and the Java construct:[8]

- The encapsulated variables of a monitor are accessible only under mutual exclusion. Java does not *require* that all of the methods of a class (intended to be used as a monitor) be declared `synchronized`. This is intended to allow concurrent execution of methods which simply return the value of a variable, but also makes it possible to subvert the protection mechanism. Ada 95 allows functions unsynchronized access to variables of a protected object, but the compiler checks that the functions do not modify the variables.

- Monitors can declare an arbitrary number of condition variables, each with its own queue of blocked processes. Similarly, in Ada 95 a protected object can contain an arbitrary number of entires, each with its own queue. However, in Java there is one queue per object, so there is no way of selectively awakening threads.

- The queues of monitor condition variables and Ada 95 barriers are FIFO; in Java, when there are several waiting tasks, it is unspecified which one is `notify`'ed. This makes it very difficult to program starvation-free algorithms.

- Monitors and Ada 95 specify *immediate resumption*: once a condition is signalled, a waiting process is activated in preference to a new process attempting to enter the monitor. This means that an awakened process need not re-check the condition it was waiting on, but can assume that once the signalling process has established the condition, no intervening process can destroy it. This is not true in Java, so awakened threads are often required to re-check their conditions.

Here is the outline of the synchronized object for grouping reindeer and elves.

```
class Group
{
  synchronized void Register()
  {
    while (Entrance == Closed) wait();
    Waiting++;
    if (Waiting < GroupSize)
```

[8] Brinch Hansen, who was one of the contributors to the monitor concept, showed how they could be used in practical systems programming. See the volume of his collected articles [3]. Of particular interest is Chapter 21 which is a history of the development of monitors.

```
      while (ExitDoor == Closed) wait();
    else {
      Entrance = Closed;
      ExitDoor = Open;
      notifyAll();
    }
    Waiting--;
    if (Waiting == 0)
      SantaClaus.Santa.SantaMonitor.Wake(GroupID);
  }

  synchronized void OpenDoor()
  {
    Entrance = Open;
    ExitDoor = Closed;
    notifyAll();
  }

  private int Waiting = 0;
  private int Entrance = Open;
  private int ExitDoor = Closed;
}
```

The problem with this solution is its high-overhead: all waiting threads are notified (`notifyAll`), and they each proceed to re-check the condition. Compare this with the Ada 95 solution where the private entry `Wait_for_Last_Member` has its own queue; both barriers are evaluated only once per completion of an operation, and only one waiting task is awakened.

While monitors are appropriate for protecting data from concurrent access, they do not appear to be suited to programming processes which offer more than one service. To synchronize between Santa Claus on the one hand, and the reindeer and elf groups on the other, a Java monitor was created that implements an explicitly programmed state machine. The methods `Who` and `Reset` are called by the Santa thread, and the method `Wake` is called by the last reindeer or elf.

```
class Monitor
{
  private static final int Sleeping     = 0;
  private static final int Harnessing   = 1;
  private static final int ShowingIn    = 2;
  private static final int Delivering   = 3;
  private static final int Consulting   = 4;
  private static final int UnHarnessing = 5;
  private static final int ShowingOut   = 6;
```

```
  private int State = Sleeping;

  synchronized void Wake(int GroupID)
  {
    while (State != Sleeping) wait();
    if (GroupID == SantaClaus.ReindeerID)
      State = Harnessing;
    else  // GroupID == SantaClaus.ElvesID
      State = ShowingIn;
    notifyAll();
  }

  synchronized int Who()
  {
    while ((State != Delivering) && (State != Consulting))
      wait();
    if (State == Delivering)
      return SantaClaus.ReindeerID;
    else // State == Consulting
      return SantaClaus.ElvesID;
  }

  synchronized void Reset(int GroupID)
  {
    if (GroupID == SantaClaus.ReindeerID)
      State = UnHarnessing;
    else // GroupID == SantaClaus.ElvesID
      State = ShowingOut;
    notifyAll();
   }

  synchronized void Harness(int r) ...
  synchronized void UnHarness(int r) ...
  synchronized void EnterOffice(int e) ...
  synchronized void LeaveOffice(int e) ...
}
```

Note that here too all waiting threads are awakened because there are no condition variables. Also note that since Java queues are not FIFO, race conditions are possible.

RELATED WORK ON JAVA

Sivilotti and Chandy [10] have developed a Java library of reusable classes that implement higher-order synchronization primitives such as locks and barriers. Their primitives are primarily useful for terminating algorithms since they are not starvation-free.

I attempted to use modifications of their primitives to simplify the above program by declaring two `Barrier` threads within `Register`. However, this was not successful because calling a synchronized method of a `Barrier` from the synchronized method `Register` obviously leads to deadlock! Removing the synchronization from `Register` would require a complete redesign of the program, and it is not clear that the result would be any more elegant.

Lea [8] has written a comprehensive book on Java concurrency techniques. The reader is invited to develop a full solution of the Santa Claus problem in Java using some of the techniques he describes such as explicitly programming a queue.

CONCLUSION

In spite of its simple formulation, the Santa Claus problem has proven to be a challenging exercise in concurrency. However, this excellent problem does not lend itself to a elementary solution using semaphores. While semaphores can and should be used to introduce concurrency *concepts*, I do not believe that they are appropriate for writing concurrent *programs*. Once the basic concepts are understood, students should be given stronger, more structured constructs such as monitors, or one of the synchronous primitives easily available in languages such as Ada, SR, Linda and occam.

The combination of protected objects and rendezvous in Ada 95 gives the flexibility of choosing the most appropriate synchronization primitive for each use. Java primitives, however, seem to be too weak for directly programming concurrent algorithms. Much work will be required to develop and standardize a concurrency library in Java.

The Ada 95 program was written using the GNAT compiler[9], and the Java program was written using Sun's JDK.[10] The author will send a copy of the programs by e-mail upon request.

ACKNOWLEDGEMENT

I would like to thank Ted Baker for clarifying the semantics of `requeue`.

[9] `ftp://cs.nyu.edu/pub/gnat`
[10] `http://java.sun.com`

REFERENCES

[1] J. Barnes. *Programming in Ada 95*. Addison-Wesley, Reading, MA, 1995.

[2] M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice-Hall International, Hemel Hempstead, 1990.

[3] P. Brinch Hansen. *The Search for Simplicity: Essays in Parallel Programming*. IEEE Computer Society Press, Los Alamitos, CA, 1996.

[4] A. Burns and A. Wellings. *Concurrency in Ada*. Cambridge University Press, Cambridge, 1995.

[5] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[6] C.A.R. Hoare. Monitors: an operating system structuring concept. *Communications of the ACM*, 17(10):549–557, 1974.

[7] Intermetrics. *Ada 95 Language Reference Manual*, 1995. ANSI/ISO/IEC 8652:1995.

[8] D. Lea. *Concurrent Programming in Java*. Addison-Wesley, Reading, MA, 1997.

[9] J.M. Morris. A starvation-free solution to the mutual exclusion problem using semaphores. *Information Processing Letters*, 8:76–80, 1979.

[10] P.A.G. Sivilotti and K.M. Chandy. Towards high-confidence distributed systems with java: Reliable thread libraries. In *11th International Conference on Systems Engineering*, pages 9–11, Las Vegas, NV, 1996. http://www.cs.caltech.edu/~paolo/publications/icse.ps.

[11] S.T. Taft. Programming the internet in Ada 95. In *Ada Europe '96*, 1996. http://www.intermetrics.com/~stt/adajava_paper/.

[12] J.A. Trono. A new exercise in concurrency. *SIGCSE Bulletin*, 26(3):8–10, 1994. Corrigendum: 26(4):63.