A New Exercise in Concurrency

John A. Trono
Saint Michael's College
One Winooski Park
Colchester, VT 05439

Abstract

This article presents an exercise to be assigned whenever undergraduates are introduced to the concepts of concurrency and semaphores. It also presents several possible strategies to solve said exercise along with a "coded" solution.

Introduction

Most students in the area of computer science experience only the sequential paradigm of computer programming in their first few years as an undergraduate. The introduction to ideas such as concurrency, interprocess communication, and process synchronization typically occurs in upper level courses. That is also when the students become acquainted with the notions of parallel processing and distributed systems. At St. Michael's College, it is in our junior level operating systems course that these topics are introduced, using such classic problems as the producers-consumers, the readers and writers, and the dining philosophers. These problems are contained in most operating systems texts, along with the typical homework exercises like the sleepy barber problem, the baboons crossing the bridge problem, and so on. It has been my experience that for most students, their first few attempts at solving these exercises (utilizing co-operating processes) suffer from logic errors analogous to beginning programmers; there is a learning curve and students need many, single problem, assignments over time, to gain proficiency in this area.

Background

Since the operating systems course is the first time that students at St. Michael's College have been exposed to "systems programming", I follow the traditional approach to this topic, in the order proscribed in the second edition of the text by Peterson and Silberschatz[8]. Even though currently popular texts have integrated concurrency into much earlier chapters (chapter 2 in Tanenbaum[11], and chapter 4 in Silberschatz and Galvin[10]), I feel it is still appropriate for me to discuss concurrency nearer the middle of the semester.

Along with the examples cited previously, the solution to the readers and writers problem, where priority is given to the writers[1], **instead** of the readers, is designed and discussed in class. This example is more involved than the case where readers are given priority and illustrates several key points when one is working with semaphores and a large number of autonomous processes. Also, several articles concerning how to implement counting semaphores, using only the binary semaphore operations PB and VB, are given to the students[2,3,4,5,7]. After some in-class debate, the students must select and justify one of the "solutions" in a short paper. While this is occurring in the lecture, the students are working on the exercises mentioned already, along with the agent and smokers problem[9,10,11] and the four-of-a-kind problem[6]. Considering the time spent introducing semaphores and message passing, concurrency comprises a third of the semester, along with a few quizzes on these topics, and some material on distributed systems.

About four years ago, when the last lecture for operating systems was held (which was a review for the final), a student asked me to go over another sample concurrency problem, and if possible, with more than the two types of processes as found in most of the problems covered so far. Fortunately, with Christmas two weeks away, I decided to attempt an entertaining example coordinating a Santa Claus process, nine reindeer processes and a large number of elf processes. Since then, it has always been one more example to use in class, and I share it with you now.

Problem Definition

Santa Claus sleeps in his shop up 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.

A Solution

The solution that has worked best over the years, and also appears to be the simplest, is written using C statements and pseudo-code. (Constants are also used in case the number of reindeer were to change, or if the group size of "solution-seeking" elves is modified.) Basically, the reindeer arrive, update the count of how many have arrived, and the last one wakes up Santa. An elf, upon discovering a problem, attempts to modify the count for the number of elves with a problem and either: waits outside Santa's shop if he/she is the first or second such elf; knocks on the door and wakes up Santa if that elf is the third one; or waits in the elves' shop until the elves currently with Santa start coming back. (The code for this solution can be found in the Appendix.)

Other Solutions

Without the counting semaphore **only_elves**, any solution must then either maintain a count of how many elves are waiting to have a problem solved, or, have the third elf "lock out" other elves until they leave Santa's shop. These solutions are typically more complicated but can work. One problem with the former solution approach is that students usually forget to prevent an elf who recently discovered a problem from sneaking in front of those who have been waiting and now can go to Santa's shop since the three elves are returning. Using the latter solution approach, students occasionally have too many elves going to Santa's shop, once the three elves have come back.

Conclusion

A new example for cooperating processes, using semaphores to synchronize their processing and to maintain mutual exclusion on two shared variables, has been presented. This new example has three different types of processes instead of the normal two used in most previously cited examples. It also forces one type (the elf) to monitor the behavior of other processes of that type, as well as "getting in sync" with a different process (Santa). Hopefully students will enjoy, benefit and remember this example as much as I enjoyed the dining philosophers example when I was an undergraduate. I hope this encourages others to share any new and innovative examples/exercises. Perhaps Snow White and the Seven Dwarves (add the Prince as the third process), and other such "whimsical" examples might make the material more interesting, our job more fun and most importantly, create an improved learning environment for our students.

Appendix

```
/* uses P and V for the wait and signal semaphore operations */

#define REINDEER 9 /* max # of reindeer */
#define ELVES    3 /* size of elf group */

                        Semaphores
  only_elves = 3,  /* 3 go to Santa */
  emutex = 1,      /* update elf_ct */
  rmutex = 1,      /* update rein_ct */
  rein_wait = 0,   /* block early arrivals            back from islands */
  sleigh = 0,      /* all reindeer wait
                   around the sleigh */
  done = 0,        /* toys all delivered*/
  santa_signal = 0,/* 1st 2 elves wait on          this outside Santa's shop */
  santa = 0,       /* Santa sleeps on this            blocked semaphore */
  problem = 0,     /* wait to pose the              question to Santa */
  elf_done = 0;    /* receive reply */

                        Shared integers
  rein_ct = 0;  /* # of reindeer back */
  elf_ct = 0;   /* # of elves w/problem */

                        Local Integers
  i;          /* for loop variable */

                    /* Reindeer Process */
for (;;){
  tan on the beaches in the Pacific
    until Christmas is close
  P(rmutex)
    rein_ct++
    if (rein_ct == REINDEER) {
      V(rmutex)
      V(santa)
    }
    else {
      V(rmutex)
      P(rein_wait)
    }
  /* all reindeer waiting to be
    attached to the sleigh */
  P(sleigh)
  fly off to deliver the toys
  P(done)
  head back to the Pacific islands
} /* end "forever" loop */
```

/* **Elf Process** */

```
for (;;){
  P(only_elves) /* only 3 elves "in"*/
    P(emutex)
      elf_ct++
      if (elf_ct == ELVES){
        V(emutex)
        V(santa) /* 3rd elf-wakes Santa */
      }
      else {
        V(emutex)
        P(santa_signal) /* wait outside
                        Santa's shop door */
      }
    P(problem)
    ask question /* Santa woke elf up*/
    P(elf_done)
  V(only_elves)
} /* end "forever" loop */
```

/* **Santa Process** */

```
for(;;){
  P(santa) /* Santa "rests" */
  /* mutual exclusion is not needed on
    rein_ct because if it is not equal
    to REINDEER, then elves woke up
    Santa */
  if (rein_ct == REINDEER){
    P(rmutex)
      rein_ct = 0 /* reset while reindeer blocked */
   V(rmutex);
    for (i = 0; i < REINDEER - 1; i++)
      V(rein_wait)
    for (i = 0; i < REINDEER; i++)
      V(sleigh)
    deliver all the toys and return
    for (i = 0; i < REINDEER; i++)
```

```
      V(done)
  }
  else { /* 3 elves have arrived */
    for (i = 0; i < ELVES - 1; i++)
      V(santa_signal)
    P(emutex)
      elf_ct = 0
    V(emutex)
    for (i = 0; i < ELVES; i++){
      V(problem)
      answer that question
      V(elf_done)
    }
  }
} /* end "forever" loop */
```

References

[1] Courtois, P.J., Heymans, F. and Parnas, D.L. *Concurrent Control with Readers and Writers*, Communications of the ACM, Vol. 10, # 10, 667-668, 1971.

[2] Hemmendinger, D. *A Correct Implementa-tion of General Semaphores*, ACM Operating Systems Review, Vol. 22, #3, 46-48, 1988.

[3] Hemmendinger, D. *Comments on "A Correct and Unrestrictive Implementation of General Semaphores"*, ACM Operating Systems Review, Vol 23, #1, 7-8, 1988.

[4] Hsieh, C.S. *Further Comments on Implementation of General Semaphores*, ACM Operating Systems Review, Vol 23, #1, 9-10, 1989.

[5] Kearns, P. *A Correct and Unrestrictive Implementation of General Semaphores*, ACM Operating Systems Review, Vol. 22 #4, 46-48, 1988.

[6] Holt, R.C., Graham, G.S., Lazowska E.D. and Scott, M.A. *Structured Concurrent Programming with Operating Systems Applications*, Addison Wesley, 1978. (Exercise #19, page 109.)

[7] Kotulski, L. *Comments on Imple-mentation of P and V Primitives with help of Binary Semaphores*, ACM Operating Systems Review, Vol. 22 #2, 53-59, 1988.

[8] Peterson, J.L., and Silberschatz, A. *Operating Systems Concepts*, Addison Wesley, 1985.

[9] Patil, S.S. *Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination Among Processes*, MIT Project MAC Computational Structures Group Memo, # 57, 1971.

[10] Silberschatz, A., and Galvin, P. *Operating Systems Concepts*, Addison Wesley, 1994.

[11] Tanenbaum, A.S. *Modern Operating Systems*, Prentice Hall, 1992.