Lab 2: Process synchronization

Created by Léon Mugwaneza (April 30, 2008)[1]

1. Introduction

You are to write some simple programs using shared memory and semaphores to make processes communicate and synchronize. As in lab 1, you are free write these programs in any programming language you wish, as long as you can do POSIX system calls from it. The assignment has been prepared for Java or C.

There are two main points of this assignment: to make you comfortable with semaphores, and to use unix shared memory. You should read up on semaphores in the lectures notes or in the textbook before doing the lab.

There are two different interfaces to semaphores in the POSIX standard: POSIX 1003.1b semaphores and System V semaphores. In this lab we will use System V semaphores. To compensate for the very difficult interface of SystemV, there is a simplified interface included in the lab package that is similar to, but not identical to the 1003.1b interface. It contains the usual create, destroy, P (or wait), and V (or signal) operations.
Also to make you focus on process synchronization, there is a simplified interface for SystemV shared memory in the lab package. This interface contains operations to create, destroy, attach, and detach a shared memory segment. To use a shared memory segment a process must perform the following steps :

- Create a shared memory segment and get a handle to the newly created segment from the operating system. This step is not necessary if the process already has a handle to a segment (inherited from parent for example).
- Attach the segment in its own address space. In the simplified interface of the lab package the area of the address space in which the shared segment is "mapped" is determined by the operating system. The attach operation returns the address of that area (a long in java and a pointer in C).
- Access the segment (read and/or write).
- Detach the segment from the process's address space.
- Destroy the segment (if no other process will do that).

Note that as java does not have pointers, we cannot directly read or write in shared memory. The lab package simplified interface to System V shared memory contains 2 helper methods (write and read) to write and read an array of bytes to shared memory. You will thus need to convert data types to or from arrays of bytes. The lab package contains such conversion functions for `int` (normally you will not need to convert other data types).

Java : The lab package contains these java files:
- `Semaphore.java` contains a simplified interface to System V semaphores.
- `Shared_memory.java` contains a simplified interface to System V shared memory.
- `Synchro.java` contains a starting point simple program you will use in part 1. You will have to modify this program to experiment process synchronisation.
- `Account.java` contains a starting point program you will use in part 2. In this program, two processes are modifying a shared variable.

The lab package contains also simple scripts to compile or launch the programs.

C : The lab package contains these header files (.h) and C files (.c) :
- `semaphore.h` defines the interface for semaphore.c
- `semaphore.c` is an implementation of the simplified interface to System V semaphores.
- `shared_memory.h` defines the interface to Shared_memory.c
- `shared_memory.c` is an implementation of the simplified interface to System V shared memory.
- `synchro.c` contains a starting point simple program you will use in part 1. You will have to modify this program to experiment process synchronisation.
- `account.c` contains a starting point program you will use in part 2. In this program, two processes are modifying a shared variable.

The lab package contains also makefiles to compile the programs.

2. The assignment

This assignment is composed of 2 parts. In part 1 you will use semaphores to synchronize independent processes. The processes no shared data (except the semaphores used for synchronization). In part 2, we add shared data and use semaphores to protect processes' critical sections where shared data are accessed. The assignment assumes your are familiar with operations on unix processes as experimented in lab1.

[1] Some material are from "lab2 : The dining philosophers" by Magnus Johansson

Part 1: Semaphores as a tool to synchronize independent processes

Familiarize yourself with the program in file `Synchro.java` if you are doing the assignment in java or `synchro.c` if you are a doing the assignment in C. To understand this program, you will need also to understand the simplified interface (not the implementation) to semaphores (file `Semaphore.java` if you are using java, file `semaphore.h` if you are using C).
In this program, a parent process creates 2 children processes A and B, and waits for their termination.
Each child process performs 5 iterations. Each iteration, it just displays the iteration number and sleeps for some random amount of time (to make the output easy to follow!).

i. Compile and run the program.
   a. Could you predict the output before execution? Why?
   b. Adjust the sleeping duration of one process (or both processes) to have a different output (ie another interleaving processes' traces). Could you predict this output? Why?

☛ **Note your answers to the questions above (i. a and i.b) in a text file named step1.1. You will have to hand in these answers.**

ii. We want to synchronize process A with process B such that process A is not allowed to start iteration *i* before process B has terminated its own iteration *i-1*.
   a. Modify the program to implement this synchronization using a semaphore.
   b. Change the processes' sleeping duration to make sure that although the output might be different, our synchronization constraint is always respected.

Note: Do not forget to destroy the semaphore before your program terminate. Semaphores are not removed automatically at program termination. You can see your semaphores (and others') using the shell command `ipcs` (Inter Process Communication Status). Semaphores can be removed manually using the command `ipcrm` (Inter Process Communication ReMove).

☛**Save your work in a file named Synchro2.java or synchro2.c. You will have to hand in that file.**

iii. We want now to make the 2 processes have a rendez-vous after each iteration. That is the 2 processes will perform their iterations in lockstep (both perform iteration 0, then both perform iteration 1, etc.).
   a. Modify the program to implement this rendez-vous synchronization using 2 semaphores.
   b. Change the processes' sleeping duration. Does the output change? What are the possible outputs?

☛**Save your work in a file named Synchro3.java or synchro3.c. You will have to hand in that file together with your answers to the questions in iii.b.**

iv. We want now to implement a rendez-vous between more than 2 processes (ie iterations are performed in lockstep). That is each process will perform its iteration 0, then each process performs its iteration 1, etc.
   a. Modify the program to make the parent create a third child (name it C) which like A and B performs 5 iterations displaying its name and the iteration number (for each iteration).
   b. Use semaphores to make the 3 children perform their iterations in lockstep. How many semaphores did you use? Is it possible to use less?

☛**Save your work in a file named Synchro4ab.java or synchro4ab.c. You will have to hand in that file together with your answers to the questions in iv.b.**

   c. How many semaphores are necessary to implement a rendez-vous between n processes without using a "helper process"? Describe (**do not implement**) a solution using that number of semaphores.

☛**Note your answers to the questions in iv.c in a text file named step1.4c. You will have to hand in these answers.**

   d. It is possible to implement a rendez-vous between n processes using a "rendez-vous helper process" and **only 3 semaphores**. Each process involved in the rendez-vous signals to the helper it has arrived (operation V on a semaphore) and then waits that the helper announces that all the processes have arrived (operations P on a semaphore). The rendez-vous helper waits for every participating process to arrive (n P operations on a semaphore), then resumes all the processes (n V operations on a semaphore).
   Implement this solution for the case of 3 processes in a rendez-vous (that is you will have one process more than in iv.b.

☛**Save your work in a file named Synchro4d.java or synchro4d.c. You will have to hand in that file.**

Part 2 : Semaphores as a tool for mutual exclusion between process sharing memory

Familiarize yourself with the program in file `Account.java` if you are doing the assignment in java or `account.c` if you're a doing the assignment in C. To understand this program, you will need also to understand the simplified interfaces (not the implementations) to semaphores and shared memory (files `Semaphore.java` and `Shared_memory.java` if you are using java, files `semaphore.h` and `shared_memory.h` if you are using C).
In this program, a parent process creates a shared memory segment, attaches the segment in its address space, and write 1000 in a variable in the shared memory before creating 2 children processes A and B. After that, the parent waits for children termination and destroys the shared memory segment.

Each child process performs 5 iterations. Every iteration process A adds 200 to the shared variable, and process B adds 100 to the shared variable. To help race conditions appear, both processes sleep for a random amount of time between the time they read the shared variable and the time they write back the shared variable after modification.

Note that, as for semaphores, your programs should remove the shared memory segments before termination. Shared memory segments are not automatically removed at program termination. You can see your shared memory segments (and others') using the shell command `ipcs` (the same command as for semaphores). Shared memory segments can be removed manually using the command `ipcrm` (the same command as for semaphores).

i. Compile and run the program.
   a. Could you predict the output before execution? Why?
   b. Adjust the sleeping duration of one process (or both processes) to have a different output. Could you predict this output? Why?

   ☛ **Note your answers to the questions i.a and i.b in a text file named step2.1. You will have to hand in these answers.**

ii. We now want to coordinate process A and process B so that we do not loose some modifications on the shared variable (imagine they are depositing money on my bank account).
   a. Identify each process's critical section (note the first and last instructions of the critical sections).
   b. Use a semaphore to ensure mutual exclusion between the 2 processes for the access to their critical sections.

   ☛ **Save your work in a file named Account2.java or account2.c. You will have to hand in that file together with your answers to the question in ii.a.**

iii. We want to add a new process (name it C), which behaves the same as processes A and B except it increments the shared variable only by 50.
   a. How many semaphores do you need to insure that no modification of the shared variable is lost?
   b. Modify the program to make the parent create process C, and use semaphores (the number of which you have just determined in a. above) to insure no modification of the shared variable is lost.

   ☛ **Save your work in a file named Account3.java or account3.c. You will have to hand in that file together with your answers to the question in iii.a.**

iv. We are finished with the account example. Consider now the bounded buffer problem described in lecture 7. Assume 5 producers and 10 consumers, and a buffer of size 10 items. Each producer performs 20 iterations. Every iteration a producer writes an item composed of 2 integer values (its pid and the value of the iteration number) to the buffer. Each consumer performs 10 iterations. Every iteration, a consumer reads an item composed of 2 integer values from the buffer (a pid of a producer and an iteration number), and then displays its own pid followed by the 2 integers read from the buffer. To help race conditions appear, make processes sleep for a random amount of time inside critical section and also outside critical section (it produces or consumes the item while sleeping!).
   a. Implement the semaphore based solution described in lecture 7 for this instance of the bounded buffer problem.
   b. Change the order of P operations (on mutex and fullBuffers or emptyBuffers) to see what happens. Does deadlock or starvation appear? Put back the P operations in the right order.
   c. Change the order of V operations (on mutex and fullBuffers or emptyBuffers) to see what happens. Does deadlock appear or starvation? Put back the V operations in the right order.

   ☛ **Save your work in a file named Bounded_buffer.java or bounded_buffer.c. You will have to hand in that file together with your answers to the questions in iv.b and iv.c.**