Lab 2: Process Synchronization

Operating System
Spring 2003

Lab Assistant

Alexandre David
www.docs.uu.se/~adavid/courses/os/osvt2003.html
email: adavid@it.uu.se
room: 1237
postbox: 26 (4th floor, building 1)
phone: 018 - 471 7341

Questions are numbered **Qx** and **programs Px**. Programs are to be handed with their description, see section 6. When you answer, give the section number and the question number, and use **sentences** to answer. Answers of the type "yes" "3" "no" will be rejected. Answers have to be given on the special answer sheet included in the lab package. You should have unpacked the "lab package" now (from lab1), it contains a directory `lab2` for this lab. **Go inside this directory first.**

1 Introduction

In this assignment you will learn about interprocess communication (IPC) using what is commonly known as *System V IPC*, a set of mechanisms for shared memory, semaphores, and message queues. The three mechanisms share many similarities in the way they are called and in the information the system maintains about them.

Semaphores are objects that can be used for synchronization between processes. They are described in more detail in a later section.

Shared memory is used for data objects that are common to several processes. Normally the memory area belonging to a process is private, and other processes cannot easily see data that is stored there. Shared memory allows us to create data objects that are shared by several processes. When one process makes a change to a shared object, the change is seen by all processes using the object.

Since the programming interface to the System V IPC mechanisms is rather complex, a simpler one is provided for this lab. It is described in the course directory `/stud/docs/kurs/os`, in `ipc.h`. The functions described there are available in the function library `libipc.a`.

2 Getting Started

In this part of the assignment you will use the shell commands `ipcs` and `ipcrm` to manage IPC objects. **Read the manual pages** for `ipcs(1)` and `ipcrm(1)`.

The program `ipc-create`, located in the lab package (under `lab2`), will create a number of semaphores and shared memory segments. After creating the objects, the program will wait until a key is pressed before removing them again.

Run `./ipc-create`. While it is waiting, use `ipcs` with the right flags, in another xterm to examine the objects that were created[1].

Q1: Who owns the objects you see?
Q2: How big are the shared memory segments (smallest and largest size)?
Q3: How many semaphores were created?

Press a key to let the program finish, and run `ipcs` again to make sure that everything was removed.

Q4: What happens if you run the program, and while it is paused, use `ipcrm`, in another xterm, to manually remove one or more of the IPC objects?

Run the program again, but this time interrupt it at the pause, using `Ctrl-C`, so that the IPC objects are not cleaned up (verify this with `ipcs`).

Q5: What happens when you try to run it once more after the interruption?

Try to determine the system imposed limits on the number of semaphores and shared memory segments. Use `ipcrm` afterwards to manually remove any remaining objects. (If there are a lot of objects to remove, you can use the program `ipc-free` to remove *most* of them, but note that there will always be some objects left behind that need to be removed by hand.)

3 Process Synchronization

In this section we look at two types of synchronization: *mutual exclusion* and *rendezvous*. But first we need to define semaphores.

A semaphore is an abstract data type for which there are typically two operations defined: *wait* (or p) and *signal* (or v)[2]. Each semaphore uses an internal counter to represent a number of available resources, for example how many processes are allowed simultaneous access to an object. The number of resources can never be less than zero.

Each time a process does *wait* on a semaphore, the counter is decremented and the process is allowed to proceed. However if the counter was at zero, then *wait* will cause the calling process to block until another process increments the counter again. *Signal* increments the counter, and if any processes are blocked waiting for the semaphore at the time, one of them (but we don't know which one) will be unblocked and allowed to proceed.

When we create a semaphore we decide how many resources it will initially represent, i.e. we set the initial counter value. Then we can use *wait* to "check out" a resource when we need it, and *signal* to "check in" the resource when we are done. The semaphore will make sure that we never can check out more resources than we have available.

[1] Make sure that both xterms are running on the same machine.
[2] Note that *wait* and *signal* are not the same as `wait(2)` and `signal(2)`, but rather standard abstract terms normally used when discussing semaphores. The actual function names you will use in this lab are **`semwait()`** and **`semsignal()`**, and are defined in `ipc.h`.

3.1 Mutual Exclusion

When two processes are using data in shared memory it is important that they do not both attempt to modify the data at the same time, because otherwise we may end up with unpredictable results. Therefore we need to protect the integrity of shared objects by synchronizing the processes that use them, so that only one process at a time can access each object. The concept whereby at most one process at a time is allowed access to an object is known as *mutual exclusion*.

In order to achieve mutual exclusion we need to define, for each shared object, the sections of code where the object is accessed. These are known as *critical sections*, and by limiting access to the critical sections we also limit access to the data.

For each shared object we then define a semaphore with a count of one. Before entering each critical section we *wait* for the semaphore, and at the end we *signal* it, thereby allowing another waiting process to proceed.

Sometimes we have groups of shared objects that need to be used together in some manner, and a certain relationship must be maintained between the objects. In this case, we treat the entire group as though it were a single object, and define the critical section around the group of accesses to the individual objects.

3.2 Rendezvous

A *rendezvous* is a technique we can use when we have two processes running, and we want to synchronize their execution at a certain point. We may know that process 1 will reach point **A** before process 2 reaches point **B**, and wish them to synchronize at these points (figure 1). when process 1 reaches point **A** it should wait there until process 2 reaches point **B**.

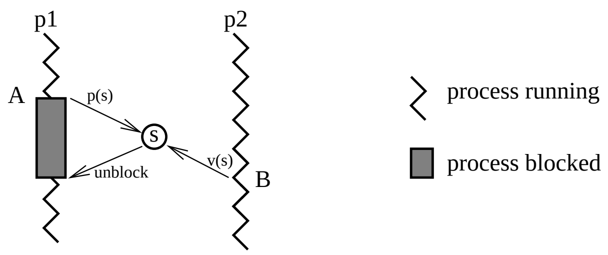

Figure 1: Rendezvous

If we know that one process will reach the rendezvous ahead of the other one, then we can use a semaphore to do the synchronization. A semaphore is created with a count of zero, and the first process to reach the rendezvous calls *wait* on the semaphore. Since there are no resources available, it will block.

When the second process subsequently reaches the rendezvous, it calls *signal* to unblock the first process. At that point, the two processes have synchronized and both can proceed.

3.3 Simple Synchronization Problem

Read the file `ipc.h` in the lab directory `lab2` to see what IPC primitives are available.

In `pencils.c` there is a program that simulates a pencil factory. The program creates a number of processes to make pencils, and one process to provide them with supplies when necessary. Read through the source code to familiarize yourself with it.

Some of the data (the supplies themselves and the number of manufactured pencils) is in shared memory and needs to be protected by a semaphore to avoid being corrupted. In addition, a rendezvous must be added to synchronize the supplier with the manufacturers. The supplier should wait at the rendezvous point until it is released by one of the manufacturers needing parts. Make sure that the supplier is called only once each time there is a shortage of parts.

P1: Add the necessary synchronization mechanisms to the program so that it works properly.

Make sure that:

- Only one process will simultaneously access, (read or write), each shared memory.
- The supplier is only running when there is a shortage of supplies.
- When the manufacturers discover that there is a shortage of supplies the supplier shall only be called *once*.
- No more pencils than the specified number are made.
- All processes are exiting correctly.

Hint: there are several working solutions, but you don't need to change any of the existing code to solve this part of the assignment. You only need to add some *wait* (using `semwait` or `p`) and *signal* (using `semsignal` or `v`) commands to the `manufacture` and `supply` functions in appropriate places. You will have two semaphores specified, one with an initial count of 0 and one with an initial count of 1. Only the code for the modified `manufacture` and `supply` functions, together with a description of your implementation, needs to be included in your lab report.

There is another example in `/stud/docs/kurs/os`: `tennis.c`, where two processes play tennis with each other. You may wish to refer to this example for clues.

Look at the included `Makefile` to see how to compile your program. A simple `make` should be enough.

4 Classical Synchronization Problem

P2: Choose one of the following classical synchronization problems and implement a deadlock-free solution, using shared memory for the common data structures and semaphores to synchronize access. Each of the "actors" in the problem should be a separate process that runs independently of the others.

Together with your deadlock-free implementation and your detailed program description you need to answer the following questions:

Q6: Describe what deadlock and starvation mean in terms of your particular problem.
The four necessary conditions for deadlock are: *Shared resource*, *Hold and wait*, *No preemption* and *Circular wait*.
Q7: Map these definitions to your particular problem and give a convincing argument, in terms of these four necessary deadlock conditions, why your solution is deadlock-free.
Q8: Is your implementation starvation-free too? If so, show how you have avoided starvation. If not, describe how your solution might be modified to make it starvation-free.

Dining Philosophers : Choice # 1 A number of philosophers are eating chinese food at a round table. Each philosopher is independent, and spends his time first thinking then eating for random times, until no food remains on the table. There is only one chopstick between each two philosophers, so although more than one philosopher can eat at the same time, two neighbours cannot.

In order to eat, a philosopher must obtain both chopsticks nearest him, one at a time. When he has eaten for some time, he puts both chopsticks back on the table and continues thinking, then eating, etc. This continues until no food remains. Food comes from a large plate in the center of the table that is shared by all the philosophers. Note that philosopers *must not* leave the table before all food has been eaten, since this changes the nature of the problem.

Make sure it is clear from your run-time example that two philosophers can (and sometimes do) eat at the same time. Give at least two run-time examples of your program when 4 respectively 5 philosophers are dining.

Narrow Bridge : Choice # 2 A city is built on two islands connected by a narrow bridge. There are many cars driving throughout the city and occasionally crossing the bridge. The bridge is only wide enough for traffic in one direction at a time.

Because the bridge is narrow the cars must also travel slowly while crossing the bridge (i.e. it should take some time). There is no traffic light.

When a car decides to cross the bridge, one of three situations can occur:

i) the bridge is free, in which case the car may cross.

ii) the bridge is occupied, and the traffic on the bridge is travelling in the right direction, so the car is allowed to cross.

iii) the bridge is occupied, but the traffic is in the wrong direction. The car must either wait until the bridge is free, or come back later and try again.

Make sure that you have enough cars, and that they decide to cross the bridge often enough that all three of the traffic situations can occur. Use one process to represent one car.

Under the course directory `/stud/docs/kurs/os`there is a program `bridge_queue` which might be used as an inspiration on how the program should work.

5 Important

- Leave no stray resources. Make sure your programs always clean up all allocated IPC objects before exiting.
- If you are still debugging your code and resources are being left behind, then use `ipcrm` to clean up manually.
- Busy waiting or polling is not allowed! In a multitasking system we need to use resources efficiently, especially the cpu. All **synchronization** must be done with **semaphores**, and all **delays** with `sleep(3C)` so that the process blocks. *No synchronization with* `sleep(3C)` *is allowed.*
- If you want to use random numbers have a look at `drand48(3C)`, (don't forget the random number generator with `srand48(3C)`).
- *Always* check return values. Most system calls return a value to indicate if they were successful, and if not, the reason for failure. They do fail sometimes! In these cases you should use `perror(3C)` to print a message describing the failure.

- `perror(3C)` is a special message function that understands the error values returned by most system calls. Use it! But note also that `perror(3C)` is not a general purpose function, it will only indicate the status of system calls, not C library functions and other functions. Use `fprintf(stderr,...)` when these fail.
- Leave *no* zombies. If you create a process, you need to wait for it too (with `wait(2)`). All processes should exit properly, and the main process must clean up after them. This means that the main process must keep track of how many processes have been *successfully* started, and never exit without waiting for each of them.
- The "real" interface to the System V IPC functions is rather complex so a simple one has been provided for the lab. It is declared in `ipc.h` and defined in `libipc.a`.

5.1 FAQ

How to print? Use `a2ps -Ppr1411` *yourfile.c*.

I got segmentation fault, what is that? Your program tried to read/write in forbidden memory zones. It is typical from a pointer not initialized.

Bus error Fatal error, typically you tried `wait(2)` in your program and you did not notice that `wait(2)` meant look at `wait` in the *section 2* of the man pages.

What does kill do? It sends a signal to a process, to terminate it generally (depends on the options), it is similar to clicking to the close button of a window. `kill -9` is a nasty signal which cannot be ignored and that always kills a running or sleeping process, i.e. not a zombie. See the manual pages on kill for complete information.

Why compilation uses -Wall -Werror? To warn on all possible errors and treat them as errors… to force you to have some discipline with C because it is a rather dirty language.

What is make? It is an utility widely used in programming: it takes a file (`Makefile` most often) in a special format, which contains *dependency rules*, then it calls the proper programs to build the *targets*. The result is that you type `make` and depending on your last modifications, the right programs are rebuilt. It is very convenient to use it, that's why a Makefile is provided in the lab. You may have a look at it if you are curious. Have a look at the manual and `www.csd.uu.se/documentation/programming/make` for more information.

Command not found? You are trying `pstree` (for example) and you expect it work, but you get an error. Check that you are in the right directory (`pwd`) and try `./pstree`. The error comes from the fact that "." (your current directory is not in your path: `echo $PATH` to check this).

6 Lab Report

A proper report is to be handed in for the lab. It should include at least the following:

- The cover page provided at the end of this description.
- Answers to the given questions.
- A description of your implementation. Explain in abstract terms (not program syntax) how it works and why you chose your particular solution to the problem. Use diagrams if you like.
- Paths to readable source files, executable files, and the shell commands used to start the programs.
- Informative listings of test runs.
- Listings of well-commented and well-structured programs. Use the command `a2ps(1)` to get nice printouts.

This assignment must be handed in no later than June 1st, 2003.

Lab 2: Process Synchronization

Operating System
Spring 2003

Lab Assistant

Alexandre David
www.docs.uu.se/~adavid/courses/os/osvt2003.html
email: adavid@it.uu.se
room: 1237
postbox: 26 (4th floor, building 1)
phone: 018 - 471 7341

I/We have solved these labs independently, and each of us has actively participated in the development of all of the lab solutions.

Lab solver 1	Lab solver 2
.... Signature	.... Signature
.... Name	.... Name
.... Personal number	.... Personal number
.... Username	.... Username

....
Path to source files