OPERATING SYSTEMS, DV1628/DV1629

—

LAB 1: PROCESSES, THREADS, AND SYNCHRONIZATION

Håkan Grahn
Blekinge Institute of Technology

2021 fall
uppdated: 2021-08-24

The objective of this laboratory assignment is to study how user-level processes and threads work, as well as study various synchronization and communication mechanisms for processes and threads.

The laboratory assignments should be conducted, solved, implemented, and presented in groups of two students! It is ok to do them individually, but groups of more than two students are not allowed.

Home Assignment 1. Preparations

Read through these laboratory instructions and do the **home assignments**. The purpose of the home assignments is to do them *before* the lab sessions to prepare your work and make more efficient use of the lab sessions.

Read the following sections in the course book [1]:

- Section 2.1: Processes
- Section 2.2: Threads
- Section 2.3: Interprocess communication
- Section 6.1 and 6.2: Resources and Introduction to deadlocks
- Section 8.1: Multiprocessors
- Section 10.3: Processes in Linux

End of home assignment 1.

Home Assignment 2. Plagiarism and collaboration

You are encouraged to work in groups of two. Groups larger than two are not accepted.

Discussions between laboratory groups are positive and can be fruitful. It is normally not a problem, but watch out so that you do not cross the border to cheating. For example, you are *not allowed* to share solution approaches, solutions to the different tasks, source code, output data, results, etc.

The submitted solution(s) to the laboratory assignments and tasks, should be developed by the group members only. You are not allowed to copy code from somewhere else, i.e., neither from your student fellows nor from the internet or any other source. The only other source code that is allowed to use is the one provided with the laboratory assignment.

End of home assignment 2.

1 Introduction

A computer system consists of one or several processors. Today, most processors are so called multicore processors, i.e., the processor contains several cpu cores internally. In this laboratory assignment we will study how we can divide the work of a program, either using processes or using threads. We will also study the memory model for processes and threads, how they share data and information, and how we synchronize between processes and threads.

The programs in this lab assignment are tested on Ubuntu Linux, and thus work in the lab room (G332). However, they should also work on any Linux distribution you may have at home. The source code files and programs necessary for the laboratory are available on Canvas.

If nothing else is said/written, you shall compile your programs using:

```
gcc -O2 -o output_file_name sourcefile1.c ... sourcefileN.c
```

When you compile and link a parallel program written using pthreads, you shall add "`-lpthread`" to the compile command in order to link the pthreads library to your application:

```
gcc -O2 -o output_file_name sourcefile1.c ... sourcefileN.c -lpthread
```

All commands are written in a Linux terminal or console window.

2 Examination and grading

Present and discuss your solutions with a teacher / lab assistant when all tasks are done.

When you have discussed your solutions orally, prepare and submit a `tar`-file or `zip`-file containing:

- **Source code:** The source-code for working solutions to the tasks in this laboratory assignment. Specifically, your well-commented source code for **Task 1**, **Task 2**, **Task 4**, **Task 6**, **Task 9**, **Task 11**, **Task 13**, **Task 15**, **Task 16**, and **Task 17**.
- **Written report:** You should write a short report (approximately 2-3 pages) describing your answers to the questions in the assignment and your implementations. Submit answers to all questions posed in the tasks, including measurements etc. The format of the report must be pdf.

All material (except the code given to you in this assignment) must be produced by the laboratory group alone.

The examiner may contact you within a week, if he/she needs some oral clarifications on your code or report. In this case, all group members must be present at that oral occasion.

3 Processes

In this first part of laboratory 1, we will study how user-level processes work. A process is created when we start a program, and processes can also crete new processes.

Home Assignment 3. `fork()` and `exec()`

Study the man pages of `fork()` and `exec()` so you understand how these system calls work and their differences. (hint, e.g., write `man fork` in the terminal on a Linux system)

End of home assignment 3.

The standard way of creating new user-level processes on Unix systems is the system call `fork()`. On Linux we also have a system call `clone()`, that also creates new processes. One of the main differences between `fork()` and `clone()` is that `clone()` allows child processes to share parts of its execution context with its parent.

The program in Listing 1 is a small example of how we can create new processes using the system call `fork()`.

Task 1. Fork example

Compile and execute the program in `fork.c` (see Listing 1).

Which process is the parent and which is the child process?

The variable `i` is used in both processes. Is the value of `i` in one process affected when the other process increments it? Why/Why not?

Modify the program in two ways:

- Let the program create a third process, that writes 1000 "C" (similarly to "A" and "B")
- In the parent process, print out the process id (pid) of each of the child processes.

Which are the process identities (pid) of the child processes?

End of task 1.

4 Process communication and synchronization

An important property of processes is memory protection and isolation, i.e., what happens inside on process should not affect the memory and data of other processes. However, in some situations we need to pass data from one process to another, i.e., using inter process communication (IPC). One approch is *sending messages* and another is communication through a *shared memory* area.

When using shared memory for process communication, we need to allocate a piece of memory that both processes have access to. The system call to allocate such shared memory from the operating system is `shmget()`. Then each of the processes can attach or detach themselves to/from that piece of memory using `shmat()` and `shmdt()`. Shared memory communication between processes can be done using either memory-mapped files or an allocated memory segment. In our examples we will use an allocated memory segment.

Home Assignment 4. Shared memory functions

Study the man pages of `shmget()`, `shmop()`, `shmat()`, `shmdt()`, and `shmctl()` so you understand how these system calls work and their differences.

Study the program example in Listing 2 (`shmem.c`).

What does the program do?

End of home assignment 4.

In the case with only *one number* in the buffer, we have a case similar to the one described in Tanenbaum's book on pages 123–124 ('Strict Alternation'). When we increase the buffer size, we do not want to restrict the ordering between the processes to strictly arternating turns. In contrast, we would like the processes to be able to access the buffer in any order. However, we must ensure two important restrictions:

1. we cannot put a new number into the buffer if the buffer is full, and
2. we cannot remove any number is the buffer is empty.

The problem that may arise otherwise is that we may lose/drop numbers or fetch/read the same number multiple times.

Task 2. Shared memory buffer

Modify the program in Listing 2 (`shmem.c`) so the buffer contains 10 numbers instead of only one number. Implement it as a *circular, bounded buffer*, i.e.,

- the items shall be fetched by the consumer in the same order at they were put into the buffer by the producer,
- when you come to the end of the buffer you start over from the first buffer place, and
- it is not allowed to fill the buffer with 10 items before the consumer runs / fetches items in the buffer.

The producer, i.e., the parent process, shall wait a random time between 0.1s - 2s each time it has put a number into the buffer. Similarly, the consumer, i.e., the child process, shall also wait a random time between 0.1s - 2s each time it has fetched a number from the buffer.

Comment you program and explain where and how the problems described above can occur.

End of task 2.

In the previous example, there is a risk that both processes may read and update the same data simultaneously. Therefore, we need to have some mechanism to protect the data. A common way to do that is to introduce critical sections, where only one process at the time is allowed to access the shared data.

In the following example (`semaphore.c`), we introduce POSIX semaphores as a way to implement critical sections. POSIX semaphores come in two versions: *named semaphores*, that can be used between processes, and *unnamed semaphores*, that are memory-based and can only be used between threads in the same process or on a memory region shared between processes (such as the one that create in the previous example). In this example we will look at named semaphores.

Home Assignment 5. Semaphores

Read the man pages for `sem_open()`, `sem_post()`, `sem_wait()`, `sem_close()` and `sem_unlink()`.

End of home assignment 5.

Home Assignment 6. Semaphore example

Study the program example in Listing 3 (`semaphore.c`).

What does the program do?

End of home assignment 6.

Task 3. Semaphore example

Compile and execute the program example in Listing 3 (`semaphore.c`).

You compile it with:

```
gcc -O2 -o semaphore semaphore.c -lrt -lpthread
```

Be sure that you clearly understand how the semaphores are allocated and initialized, and how they are used.

What are the initial values of the semaphores `sem_id1` and `sem_id2`, respectively?

End of task 3.

Task 4. Shared memory buffer with semaphores

Copy the program you developed in Task 2, the one with a circular buffer containing 10 slots.

Modify the new program by adding proper synchronizations using semaphores.

Make sure the program executes correctly by using semaphores, i.e., there should be no risk for dropping numbers or reading numbers several times.

Hint: Think about which initial values that are suitable for the semaphores.

End of task 4.

5 Message queues

Sending messages is an alternative to using shared memory for inter-process communication (IPC). In the following examples we will create a process that send messages to another process using the messaging primitives available in Linux/Unix. In this type of IPC, the processes do not share memory, instead they let the kernel handle the tranfser of messages between the processes. There are two different message queue APIs in Linux: System V and POSIX, and they provide similar functionality. System V message queues have been around for a long time, thus they are widely used and in essence all Unix systems implement them. POSIX message queues are newer, thus they are developed based on the experience on System V messege queues.

In our laboratory, we are going to use System V message queues. The basic functionality we are interested in is the ability to send a message from one process to another and the ability to recieve a message from another process. The first is implemented in a call named `msgsnd()` and the other in a call named `msgrcv()`. In this scenario, one process sends a message to the kernel, and then the kernel locates the other process and delivers the message to that process.

Home Assignment 7. Message queues

Read the man pages for `msgget()`, `msgsnd()`, `msgrcv()`, and `msgctl()`.

End of home assignment 7.

In the following example (`msgqsend.c` and `msgqrecv.c`), we introduce how inter-process communication can be done using System V message queues. The example is taken and modified from Tutorialspoint.[1]

Home Assignment 8. Message queue example

Study the program example in Listing 4 (`msgqsend.c`) and Listing 5 (`msgqrecv.c`).

What do the programs do?

End of home assignment 8.

Task 5. Message queue example

Compile the programs in Listing 4 (`msgqsend.c`) and Listing 5 (`msgqrecv.c`).

When you execute them, run them in two different terminal windows. Further, make sure you start `msgqsend` first.

Why do you need to start `msgqsend` first?

End of task 5.

Task 6. Message queue modification

Modify the programs in Listing 4 (`msgqsend.c`) and Listing 5 (`msgqrecv.c`) to send and recieve a sequence of integers instead.

A sequence of 50 integers should be generated automatically with random values in the range INT_MIN < x < INT_MAX.

Hint: The size of an `int` is different from the size of a `char`.

End of task 6.

6 Threads

Threads are different to processes in how they share resources such as memory and files. A process can contain one or several threads. For example, a sequential program is executed in one process with only one single executing thread. In this laboratory we will use POSIX Threads, or pthreads for short.

Home Assignment 9. Thread creation

Read the manual pages for the following functions:

- `pthread_create()`
- `pthread_exit()`
- `pthread_join()`

Especially, make sure you understand the different arguments (input parameters) to the different functions.

End of home assignment 9.

[1] `https://www.tutorialspoint.com/inter_process_communication/inter_process_communication_message_queues.htm`

In the first thread example, see Listing 6 (`pthreadcreate.c`), we will study how threads are created and terminated.

Task 7. `pthread_create()` example

Compile and execute the program in Listing 6 (`pthreadcreate.c`). Remember the `-lpthread` flag to the compiler.

Make sure you understand what it does and why.

What does the program print when you execute it?

End of task 7.

Sometimes we would like to send arguments (input parameters) to a thread we create, and sometimes we also want the thread to return some data. A tricky thing is that a thread can only have *one* argument and that argument should be of type `void*`.

In the exemple in Listing 7 (`pthreadcreate2.c`), we show an example of how we can pass arguments to a thread when we create the thread. Two data items, `id` and `numThreads` are put in a `struct`, and then a `void*` pointer is set to point to that struct, and finally, the `void*` pointer is passed to the thread function. In the child threads, the input parameter is typecasted back to the `struct` type, and then we can use the different values passed to the child thread.

Task 8. `pthread_create()` example 2

Compile and execute the program in Listing 7 (`pthreadcreate2.c`).

Make sure you understand what it does and why.

Why do we need to create a new `struct threadArgs` for each thread we create?

End of task 8.

When we want to return values from a child thread, we cannot simply execute a `return` statement as in a normal function, since the thread function must have return type `void`. A solution is to return values to the parent thread through the same `struct` as the input parameters were passed. In this situation, we cannot allow the child to deallocate the `struct` where the arguments are passed to the child thread, as in Task 8 (Listing 7). Instead, the parent thread is responsible for both allocating and deallocating the argument `struct`.

Task 9. `pthread_create()`, how to handle return values

In Listing 8 (`pthreadcreate3.c`), we provide a program where the parent thread allocates (and deallocates) an array of `structs` of type `threadArgs`, one `struct` for each child thread. The `threadArgs struct` passed to each child thread can then be used to pass values from the child thread to the parent thread.

Your task is to:

- Add a new variable in the struct `threadArgs`, called `squaredId`.
- In the child thread, take the thread id, square it, and return the value. Use the `threadArgs struct` for communicating values between the parent and the child.
- In the main program, when all threads have terminated, print the squared id for each thread (this value should be returned by each thread).

End of task 9.

7 Thread communication and synchronization

Since all threads in the same process share memory, they can easily communicate with each other by reading and writing data to variables in the shared memory. However, as in the case for processes, we need to protect the shared variables from simultaneous modification by different threads. A mechanism the achieve this protection is mutex variables. Therefore, we will in this part of the laboratory study how mutex synchronization works.

Home Assignment 10. Thread synchronization

Read the manual pages for the following functions:

- `pthread_mutex_init()`
- `pthread_mutex_lock()`
- `pthread_mutex_unlock()`

End of home assignment 10.

Task 10. Bank account

In Listing 9 (`bankaccount.c`) you find a simple simulation of a bank account where deposits and withdrawals are done. Each thread does either 1000 deposits (odd thread id ⇒ calls `deposit()`) or 1000 withdrawals (even thread id ⇒ calls `withdraw()`).

If everything goes well, the saldo at the end of the execution should be 0, given an even number of threads.

Execute the program a number of times (with an even number of threads, e.g., 8, 16, 32, and 64 threads).

Does the program execute correctly? Why/why not?

End of task 10.

The problem that occur is that two (or more) threads try to read and update the same variable at the same time, and their accesses to the shared variable (in this case `bankAccountBalance`) are interleaved. When updating a variable, a thread reads the value of the variable from the memory, updates the value (a local operation in a processor register), and finally writes the new value back to memory. When two threads do this concurrently, the execution may look like the one in Figure 1 and is called a *race condition*. Note that this situation may occur also on single-core processors, e.g., if an interrupt and process context switch happens between line 1 and 2. Programs with data races may have unpredictable behavior, and can cause bugs that are very hard to find and reproduce (so called "Hiesenbugs").

Time	Thread 1 (on cpu1)		Thread 2 (on cpu2)	
1	proc_reg = mem_var	(reg_value == 1)		
2			proc_reg = mem_var	(reg_value == 1)
3	proc_reg++	(reg_value == 2)	proc_reg++	(reg_value == 2)
4	mem_var = proc_reg	(mem_value == 2)		
5			mem_var = proc_reg	(mem_value == 2)

Figure 1: Possible execution when two threads try to update the same variable `mem_var` concurrently, assuming that `mem_var==1` before the execution. The situation is referred to as a *race condition*. The correct value of the code sequence should be 3, not 2.

The solution to the problem in Figure 1 is to enclose accesses to shared variable in critical sections. Then we can guarantee mutual exclusion, i.e., only one thread at the time can access and modify a shared variable.

Task 11. Bank account, correction

Correct the program in Listing 9 (`bankaccount.c`) by introducing proper synchronizations on shared variables.

The shared variable(s) that are protected and accessed in critical sections should be locked as short time as possible, i.e., you are supposed to keep the critical sections are short os possible.

End of task 11.

8 Dining professors

The Dining Philosophers is a well-known synchronization problem, that involves potential livelock and deadlock issues. The problem is described on pages 167–170 in Tanenbaum's book. In the problem, five philosophers are sitting around a table and would like to eat. Each of them has a bowl and there are five chopsticks, see Figure 2. Unfortunately, they need two chopsticks in order to eat. Thus, they need to share and sychronize with each other.

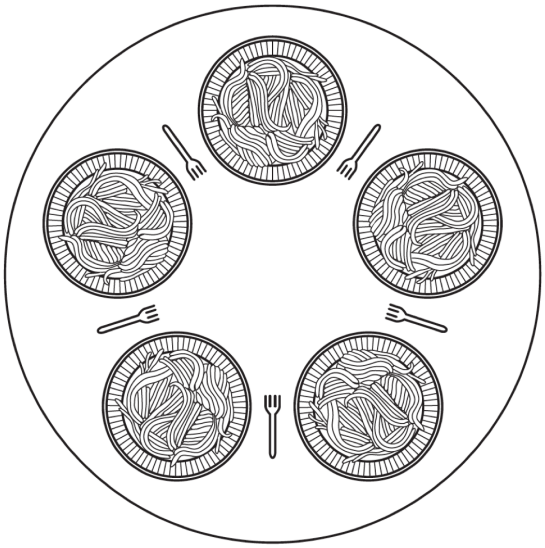

Figure 2: The Dining Philosophers

In this section, we are going to implement Dining Professors. Five operating system professors, Tanenbaum, Bos, Lamport, Stallings, and Silberschatz, are sitting around a round table and eat noodles. To the left of each bowl, there is a chopstick. Each professor needs two chopsticks to eat. Thus, the professors need to share the chopsticks with each other, e.g., Bos needs to share with Tanenbaum and Lamport.

Each professor sits and think for a while, and then becomes hungry. He will then pick up his left chopstick (if the chopstick is not taken by the colleague to the left). Then, the professor thinks a while before he picks up the chopstick to the right (if it is not taken). When the professor has both chopsticks, he can eat. When he has eaten for a while, he puts down both chopsticks and starts to think again.

Task 12. Dining Professors, implementation

Implement Dining Professors with five professors according to the description above. The program shall contain five threads, where each thread represents one professor. The program must guarantee that only one professor at the time can use a chopstick.

- Let the professors think a random time between two and ten seconds.
- Take the left chopstick.
- When they have gotten the left chopstick, let them think in one to three seconds.
- Take the right chopstick.
- Let the professor eat for a random time between 10-20 seconds, and then put down both chopsticks.
- Write information to the terminal when the professors go from one state to another, e.g., "Lars: thinking -> got left chopstick". Also indicate when a professor tries to take a chopstick.

An implementation done in line with the description above is not deadlock-free, i.e., it may result in a deadlock. Explain why, i.e., which conditions lead to the deadlock?

End of task 12.

Task 13. Dining Professors, deadlock-free implementation

Modify the program from Task 12 so the program is deadlock-free, i.e., you can guarantee that no deadlock can occur.

Which are the four conditions for deadlock to occur?

Which of the four conditions did you remove in order to get a deadlock-free program, and how did you do that in your program?

End of task 13.

9 Parallelism and performance

Threads and processes are often used to introduce concurrency, i.e., the ability to do multiple things at the same time. There can be many reasons for that, e.g., handling independent requests to a server or handling different tasks such as spell checking and autobackup at the same time in a word processing application. However, another important reason to introduce concurrency is performance. When a concurrent program execute on multiple cores, it is said to execute in parallel. Ideally, if a parallel program execute on two cores it could run twice a fast, i.e., we can have a speedup of 2. Speedup is defined as:

$$Speedup = \frac{T_{sequential}}{T_{parallel}} \tag{1}$$

where $T_{sequential}$ is the execution time of sequential program and $T_{parallel}$ is the execution time of parallel program.

Unfortunately, it is often very hard to get as high speedup as we have cores running the parallel program. When we have as high speedup as we have cores, i.e., speedup of 4 on four cores, we call it *linear speedup*. However, in most cases the speedup is sub-linear, i.e., less than the ideal one. There are many reasons for that, e.g., sequential parts of the application, synchronization overhead, scheduling overhead, etc. In the following sections we will study two different applications, *matrix multiplication* and *Quicksort*. The first one is relatively easy to get close to linear speedup, while the second one is much harder.

When measuring the execution time, it looks a bit different if you have `bash` och `tcsh` as default shell in your terminal / console. In the lab room (G332) `tcsh` is default, but in a standard Ubuntu/Linux distribution `bash` is default. You usually can find out which shell you run by executing the command (if the environment variable `SHELL` is set corrrectly, which is unfortunately not always done):

```
echo $SHELL
```

If you are running `bash`, you should measure the execution time with:

```
time command
```

The value of "real" is the wall clock time that we are interested in.

If you are running `tcsh`, you should measure the execution time with:

```
/usr/bin/time command
```

The value of "elapsed" is the wall clock time that we are interested in.

9.1 Parallelization of a simple application

We start by parallelizing a simple application, *matrix multiplication*. A sequential version of the program is found in the file `matmulseq.c`, see Listing 10.

Task 14. Sequential matrix multiplication

Compile and execute the program `matmulseq.c`.

How long time did it take to execute the program?

End of task 14.

Task 15. Parallel matrix multiplication

We will now parallelize the matrix multiplication program using threads. Parallelize the program according to the following assumptions:

- Parallelize only the matrix multiplication, but not the initialization.
- A new thread shall be created for each row to be calculated in the matrix.
- The row number of the row that a thread shall calculate, shall be passed to the new thread as a paramater to the new thread.
- The main function shall wait until all threads have terminated before the program terminates.

Compile and link your parallel version of the matrix multiplication program. Don't forget to add `-lpthread` when you link your program. We will now measure how much faster the parallel program is as compared to the sequential one.

Execute the program and measure the execution time.

How long was the execution time of your parallel program?

Which speedup did you get (as compared to the execution time of the sequential version, where $Speedup = T_{sequential}/T_{parallel}$)?

End of task 15.

A performance liming factor when writing and executing parallel applications is if there is any sequential execution path in the parallel program. For example, critical sections introduce sequential parts in a program, although they are necessary for the correctness of the program. In our matrix multiplication example, we have a sequential part in the initialization of the matrices.

Task 16. Parallelization of the initialization

Parallelize the initialization of the matrices also, i.e., the function `init_matrix`. Use one thread to initialize each of the rows in the matrices `a` and `b`. Compile, link, and execute your program.

How fast did the program execute now?

Did the program run faster or slower now, and why?

End of task 16.

In the previous tasks we have had a large number of threads (1024) to perform the matrix multiplication. An alternative would be to let each thread calculate several rows in the resulting matrix. This will then result in a larger granularity, i.e., more work is done per thread. An advantage is then that the scheduling overhead will decrease (fewer threads to administrate and schedule), but a disadvantage is that the amount of parallelism is lower which may result in lower speedup.

Task 17. Changing granularity

Rewrite your program so it only creates as many threads as there are processors to execute on, e.g., when we run the program on 4 processors, we create 4 threads that each calculate 1024/4 = 256 rows in the matrix. Hardcode the number of threads to use. Execute your program and measure the execution time.

Which is the execution time and speedup for the application now?

Do these execution times differ from those in Task 15? If so, why? If not, why?

Does it make any difference now if `init_matrix` is parallelized or not?

End of task 17.

References

[1] Andrew S. Tanenbaum and Herbert Bos, "Modern Operating Systems, 4th ed", *Pearson Education Limited*, ISBN-10: 0-13-359162-X, 2015.

A Program listings

Listing 1. `fork.c`

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv)
{
    pid_t pid;
    unsigned i;
    unsigned niterations = 1000;
    pid = fork();
    if (pid == 0) {
        for (i = 0; i < niterations; ++i)
            printf("A = %d, ", i);
     } else {
        for (i = 0; i < niterations; ++i)
            printf("B = %d, ", i);
    }
    printf("\n");
}
```

Listing 2. `shmem.c`

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define SHMSIZE 128
#define SHM_R 0400
#define SHM_W 0200

int main(int argc, char **argv)
{
    struct shm_struct {
        int buffer;
        unsigned empty;
    };
    volatile struct shm_struct *shmp = NULL;
    char *addr = NULL;
    pid_t pid = -1;
    int var1 = 0, var2 = 0, shmid = -1;
    struct shmid_ds *shm_buf;

    /* allocate a chunk of shared memory */
    shmid = shmget(IPC_PRIVATE, SHMSIZE, IPC_CREAT | SHM_R | SHM_W);
    shmp = (struct shm_struct *) shmat(shmid, addr, 0);
    shmp->empty = 0;
    pid = fork();
    if (pid != 0) {
        /* here's the parent, acting as producer */
        while (var1 < 100) {
            /* write to shmem */
            var1++;
            while (shmp->empty == 1); /* busy wait until the buffer is empty */
            printf("Sending %d\n", var1); fflush(stdout);
            shmp->buffer = var1;
            shmp->empty = 1;
        }
        shmdt(addr);
        shmctl(shmid, IPC_RMID, shm_buf);
```

```
    } else {
        /* here's the child, acting as consumer */
        while (var2 < 100) {
            /* read from shmem */
            while (shmp->empty == 0); /* busy wait until there is something */
            var2 = shmp->buffer;
            shmp->empty = 0;
            printf("Received %d\n", var2); fflush(stdout);
        }
        shmdt(addr);
        shmctl(shmid, IPC_RMID, shm_buf);
    }
}
```

Listing 3. `semaphore.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <semaphore.h>
#include <fcntl.h> /* For O_* constants */

const char *semName1 = "my_sema1";
const char *semName2 = "my_sema2";

int main(int argc, char **argv)
{
    pid_t pid;
    sem_t *sem_id1 = sem_open(semName1, O_CREAT, O_RDWR, 1);
    sem_t *sem_id2 = sem_open(semName2, O_CREAT, O_RDWR, 0);
    int i, status;

    pid = fork();
    if (pid) {
        for (i = 0; i < 100; i++) {
            sem_wait(sem_id1);
            putchar('A'); fflush(stdout);
            sem_post(sem_id2);
        }
        sem_close(sem_id1);
        sem_close(sem_id2);
        wait(&status);
        sem_unlink(semName1);
        sem_unlink(semName2);
    } else {
        for (i = 0; i < 100; i++) {
            sem_wait(sem_id2);
            putchar('B'); fflush(stdout);
            sem_post(sem_id1);
        }
        sem_close(sem_id1);
        sem_close(sem_id2);
    }
}
```

Listing 4. `msgqsend.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define PERMS 0644
struct my_msgbuf {
    long mtype;
    char mtext[200];
};

int main(void) {
    struct my_msgbuf buf;
    int msqid;
    int len;
    key_t key;
    system("touch msgq.txt");

    if ((key = ftok("msgq.txt", 'B')) == -1) {
        perror("ftok");
        exit(1);
    }

    if ((msqid = msgget(key, PERMS | IPC_CREAT)) == -1) {
        perror("msgget");
        exit(1);
    }
    printf("message queue: ready to send messages.\n");
    printf("Enter lines of text, ^D to quit:\n");
    buf.mtype = 1; /* we don't really care in this case */

    while(fgets(buf.mtext, sizeof buf.mtext, stdin) != NULL) {
        len = strlen(buf.mtext);
        /* remove newline at end, if it exists */
        if (buf.mtext[len-1] == '\n') buf.mtext[len-1] = '\0';
        if (msgsnd(msqid, &buf, len+1, 0) == -1) /* +1 for '\0' */
            perror("msgsnd");
    }
    strcpy(buf.mtext, "end");
    len = strlen(buf.mtext);
    if (msgsnd(msqid, &buf, len+1, 0) == -1) /* +1 for '\0' */
        perror("msgsnd");

    if (msgctl(msqid, IPC_RMID, NULL) == -1) {
        perror("msgctl");
        exit(1);
    }
    printf("message queue: done sending messages.\n");
    return 0;
}
```

Listing 5. `msgqrecv.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define PERMS 0644
struct my_msgbuf {
    long mtype;
    char mtext[200];
};

int main(void) {
    struct my_msgbuf buf;
    int msqid;
    int toend;
    key_t key;

    if ((key = ftok("msgq.txt", 'B')) == -1) {
        perror("ftok");
        exit(1);
    }

    if ((msqid = msgget(key, PERMS)) == -1) { /* connect to the queue */
        perror("msgget");
        exit(1);
    }
    printf("message queue: ready to receive messages.\n");

    for(;;) { /* normally receiving never ends but just to make conclusion */
                /* this program ends with string of end */
        if (msgrcv(msqid, &buf, sizeof(buf.mtext), 0, 0) == -1) {
            perror("msgrcv");
            exit(1);
        }
        printf("recvd: \"%s\"\n", buf.mtext);
        toend = strcmp(buf.mtext,"end");
        if (toend == 0)
        break;
    }
    printf("message queue: done receiving messages.\n");
    system("rm msgq.txt");
    return 0;
}
```

Listing 6. `pthreadcreate.c`

```
#include <stdio.h>
#include <pthread.h> // pthread types and functions

void* child() {
    printf("This is the child thread.\n");
}

int main(int argc, char** argv) {
    pthread_t thread; // struct for child-thread info
    // spawn thread:
    pthread_create(&thread, // the handle for it
        NULL, // its attributes
        child, // the function it should run
        NULL); // args to that function

    printf("This is the parent (main) thread.\n");
    pthread_join(thread, NULL); // wait for child to finish
    return 0;
}
```

Listing 7. `pthreadcreate2.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

struct threadArgs {
    unsigned int id;
    unsigned int numThreads;
};

void* child(void* params) {
    struct threadArgs *args = (struct threadArgs*) params;
    unsigned int childID = args->id;
    unsigned int numThreads = args->numThreads;
    printf("Greetings from child #%u of %u\n", childID, numThreads);
    free(args);
}

int main(int argc, char** argv) {
    pthread_t* children; // dynamic array of child threads
    struct threadArgs* args; // argument buffer
    unsigned int numThreads = 0;
    // get desired # of threads
    if (argc > 1)
        numThreads = atoi(argv[1]);
    children = malloc(numThreads * sizeof(pthread_t)); // allocate array of handles
    for (unsigned int id = 0; id < numThreads; id++) {
        // create threads
        args = malloc(sizeof(struct threadArgs));
        args->id = id;
        args->numThreads = numThreads;
        pthread_create(&(children[id]), // our handle for the child
            NULL, // attributes of the child
            child, // the function it should run
            (void*)args); // args to that function
    }
    printf("I am the parent (main) thread.\n");
    for (unsigned int id = 0; id < numThreads; id++) {
        pthread_join(children[id], NULL );
    }
    free(children); // deallocate array
```

```
    return 0;
}
```

Listing 8. `pthreadcreate3.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

struct threadArgs {
    unsigned int id;
    unsigned int numThreads;
};

void* child(void* params) {
    struct threadArgs *args = (struct threadArgs*) params;
    unsigned int childID = args->id;
    unsigned int numThreads = args->numThreads;
    printf("Greetings from child #%u of %u\n", childID, numThreads);
}

int main(int argc, char** argv) {
    pthread_t* children; // dynamic array of child threads
    struct threadArgs* args; // argument buffer
    unsigned int numThreads = 0;
    // get desired # of threads
    if (argc > 1)
        numThreads = atoi(argv[1]);
    children = malloc(numThreads * sizeof(pthread_t)); // allocate array of handles
    args = malloc(numThreads * sizeof(struct threadArgs)); // args vector
    for (unsigned int id = 0; id < numThreads; id++) {
        // create threads
        args[id].id = id;
        args[id].numThreads = numThreads;
        pthread_create(&(children[id]), // our handle for the child
            NULL, // attributes of the child
            child, // the function it should run
            (void*)&args[id]); // args to that function
    }
    printf("I am the parent (main) thread.\n");
    for (unsigned int id = 0; id < numThreads; id++) {
        pthread_join(children[id], NULL );
    }
    free(args); // deallocate args vector
    free(children); // deallocate array
    return 0;
}
```

Listing 9. `bankaccount.c`

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

// Shared Variables
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
double bankAccountBalance = 0;

void deposit(double amount) {
    bankAccountBalance += amount;
}

void withdraw(double amount) {
    bankAccountBalance -= amount;
}

// utility function to identify even-odd numbers
unsigned odd(unsigned long num) {
    return num % 2;
}

// simulate id performing 1000 transactions
void do1000Transactions(unsigned long id) {
    for (int i = 0; i < 1000; i++) {
        if (odd(id))
            deposit(100.00); // odd threads deposit
        else
            withdraw(100.00); // even threads withdraw
    }
}

void* child(void* buf) {
    unsigned long childID = (unsigned long)buf;
    do1000Transactions(childID);
    return NULL;
}

int main(int argc, char** argv) {
    pthread_t *children;
    unsigned long id = 0;
    unsigned long nThreads = 0;
    if (argc > 1)
        nThreads = atoi(argv[1]);
    children = malloc( nThreads * sizeof(pthread_t) );
    for (id = 1; id < nThreads; id++)
        pthread_create(&(children[id-1]), NULL, child, (void*)id);
    do1000Transactions(0); // main thread work (id=0)
    for (id = 1; id < nThreads; id++)
        pthread_join(children[id-1], NULL);
    printf("\nThe final account balance with %lu threads is $%.2f.\n\n", nThreads, bankAccountBalance);
    free(children);
    pthread_mutex_destroy(&lock);
    return 0;
}
```

Listing 10. `matmulseq.c`

```
/*****************************************************************************
 *
 * Sequential version of Matrix-Matrix multiplication
 *
 *****************************************************************************/

#include <stdio.h>
#include <stdlib.h>

#define SIZE 2048

static double a[SIZE][SIZE];
static double b[SIZE][SIZE];
static double c[SIZE][SIZE];

static void
init_matrix(void)
{
    int i, j;

    for (i = 0; i < SIZE; i++)
        for (j = 0; j < SIZE; j++) {
            /* Simple initialization, which enables us to easy check
             * the correct answer. Each element in c will have the same
             * value as SIZE after the matmul operation.
             */
            a[i][j] = 1.0;
            b[i][j] = 1.0;
        }
}

static void
matmul_seq()
{
    int i, j, k;

    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            c[i][j] = 0.0;
            for (k = 0; k < SIZE; k++)
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
        }
    }
}

static void
print_matrix(void)
{
    int i, j;

    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++)
            printf(" %7.2f", c[i][j]);
        printf("\n");
    }
}

int
main(int argc, char **argv)
{
    init_matrix();
    matmul_seq();
    //print_matrix();
}
```