Computation structures

Support for problem-solving lesson #5

Some recalls

What is a program?

- It's a set of instructions that are sequentially executed by the processor (or one of its cores).

Programs can be written in various languages (C, Java, Perl, PHP, …).

- We will focus on C for the problem-solving lessons and for the second assignment.

Example of C program:

```
#include <stdio.h>
int main() {
	printf("Hello World!\n");
	getchar();
	return 0;
}
```

Storing elements

Variables can be stored in the memory.

- Their accessibility depends on their context.
- The C language imposes that the type of a variable is known at its declaration.

Example:

```
int globalVariable = 0; //Can be accessed from any function
int main() {
        int localVariable = 0; // Is only accessible from the local function
        return 0;
}
```

Pointers

A pointer is a special variable whose content is not a regular value (int, char, …) but an *address* that links to another variable

Example:

```
#include <stdio.h>
int main() {
	int* pointer; // This is a pointer to an integer
	int variable = 5;
	pointer = &variable; //The pointer now contains the address of the variable
	printf("%d",pointer); //Displays the address of the variable (e.g. 2337492)
	printf("%d",*pointer); //Displays the content of the variable (i.e. 5)
	return 0;
}
```

Structures

A *structure* is a custom data type that can contain several types of data at once.

Example:

```
#include <stdio.h>
struct my_struct {
        int integerVariable;
        char character;
};
typedef struct my_struct MyStruct;
int main() {
        struct my_struct object1;          //OK
        MyStruct object2;                  //Still OK
        my_struct object3;                 //Compiling error
        object1.integerVariable = 2;       //Use '.' to access the content of a structure
        MyStruct* pointer = &object2;
        pointer->character = 'S';          //Use '->' to access the content of a structure referenced by a pointer
        (*pointer).character = 'S';        //Equivalent to the upper line
        return 0;
}
```

Unions

A *union* is very similar to a *structure* (custom data type that can contain several types of data at once), but where the memory is shared across all fields.

Example:

Sign Mantissa Exponent

2400,012 = **01000000 10100010 11000000 00000110**
00100100 11011101 00101111 00011011

Exponent

```
#include <stdio.h>
union my_union {
        int integerVariable;
        double doubleVariable;
};
typedef union my_union MyUnion;
int main() {
        MyUnion mu;                                  //Declares the union
        mu.integerVariable = 5;                      //Sets the value for the integer part
        printf("%d\n",mu.integerVariable);           //Displays "5"
        mu.doubleVariable = 2400.012;
        printf("%lf\n",mu.doubleVariable);           //Displays "2400,012000"
        printf("%d",mu.integerVariable);             //Displays "618475291". Integer value has changed!
        return 0;
}
```

Unions

A *union* is very similar to a *structure* (custom data type that can contain several types of data at once), but where the memory is shared across all fields.

Example:

```
#include <stdio.h>
union my_union {
        int integerVariable;
        double doubleVariable;
};
typedef union my_union MyUnion;
int main() {
        MyUnion mu;                                  //Declares the union
        mu.integerVariable = 5;                      //Sets the value for the integer part
        printf("%d\n",mu.integerVariable);           //Displays "5"
        mu.doubleVariable = 2400.012;
        printf("%lf\n",mu.doubleVariable);           //Displays "2400,012000"
        printf("%d",mu.integerVariable);             //Displays "618475291". Integer value has changed!
        return 0;
}
```

00100100 11011101 00101111 00011011 = 618475291

Stack and heap

Untill now, all variables (or pointers) were allocated on the stack.

- This is also where the program code is stored
- This memory space is very short (typically, a few megabytes)

We should allocate large memory storages on the heap

```
#include <stdio.h>
#include <stdlib.h>
int main() {
        int* pointer; // This is a pointer to an integer
        pointer = malloc(1024*sizeof(int)); //The pointer now links to an array of 1024 integers on the heap
        //malloc can fail (not enough available memory) and would return NULL
        free(pointer); //Always free the allocated memory (beware of memory leaks and duplicate frees)
        pointer = NULL; // Good practice : After a free, set the variable to NULL
        return 0;
}
```

Alternatives and loops

You might want your program to behave in various ways depending on the input or on the variables content

- You can use the **if** statement to test a condition and split your code into two parts (similar to the branch in assembly). **switch … case** is also an option for multiple parts.
- You can use the **while** statement (or the **for** statement) to create loops

```
#include <stdio.h>
int main() {
		int variable = 5;
		if (variable > 0) { // Will enter this part of the code
				while(variable > 0) { // The following block will be executed untill the condition is false (beware of infinite loops!)
						variable = variable – 1; //Alternatively, you could have written "variable--;"
						printf("Variable is now : %d\n",variable);
				}
		} else { //With the variable currently set to 5, this part will never be reached
				printf("Variable was not strictly positive");
		}
		return 0;
}
```

Arrays and accessors

If you need several variables of the same type, you might want to use an array.

Arrays' content can be accessed through the **[ ]** accessor (recommended) or using pointer arithmetics (**not** recommended)

```
#include <stdlib.h>
int main() {
		int array[10]; //This is an array of 10 integers
		int* pointer; // This is a pointer to an integer (or an integer array)
		pointer = malloc(1024*sizeof(int)); //The pointer now links to an array of 1024 integers on the heap
		//malloc can fail (not enough available memory) and would return NULL
		if(pointer == NULL) {
				//Not enough memory, we just quit (we could have displayed a message, here)
				return 1; //1 often means : There has been an error
		} else {
				array[5] = 4; //Beware of "out of bounds"
				*(pointer+5) = 4; //Will work perfectly
				pointer[5] = 4; //Also legitimate and preferable
				free(pointer); //Always free the allocated memory (beware of memory leaks and duplicate frees)
				pointer = NULL; // Good practice : After a free, set the variable to NULL
		}
		return 0;
}
```

Character strings

A string is an array of characters. It can be manipulated by specific functions, like strcpy(), strcmp() or strlen().

```
#include <stdio.h>
#include <string.h>
int main() {

   char string[10];

   memset(string, '\1', sizeof(string));          //"string" now contains {1,1,1,1,1,1,1,1,1,1}
   strcpy(string, "Hello");                       //"string" now contains {'H','e','l','l','o',0,1,1,1,1}
   printf("String contains : %s\n", string);    //Displays "String contains : Hello"
   strcpy(string, "Isn't this too long?");        //"string" now contains {'I','s','n',''','t',' ','t','h','i','s'}
                                                  //Buffer overflow! Very dangerous. Prefer strncpy.
   return(0);
}
```

Functions

Keeping the whole logic in the main() function would rapidly make any program hard to read, thus maintain.

You can (and should) separate the code logic into different functions.

```
#include <stdio.h>
int myFunction(int a) {
		return a+1;
}
void anotherFunction(int* a) {
		*a += 1;
}
int main() {
		int a = 1;
		//Passing by value/variable
		a = myFunction(a);
		//Passing by reference
		anotherFunction(&a);
		printf("Value of a : %d",a);  //Will display "Value of a : 3"
		return 0;
}
```

Shared memory and semaphores

Using system V

- System V requires some includes (like 'sys/ipc.h')
- Obtaining a unique key : **key_t ftok(char ***_pathname_**, int** _proj_id_**);**
- Creating a shared memory : **int shmget(key_t** _key_**, size_t** _size_**, int** _shmflg_**);**
- Attaching a shared memory : **char *shmat (int** _shmid_**, char *shmaddr, int** _shmflg_ **)**
- Creating a semaphore : **int semget (key_t** _key_**, int** _nsems_**, int** _semflg_**);**
- Wait and signal : **int semop(int** _semid_**, struct sembuf ***_sops_**, unsigned** _nsops_**);**
- Other operations on semaphores : **int semctl (int** _semid_**, int** _semno_**, int** _cmd_**, ...);**
- Online doc : http://www.tldp.org/LDP/lpg/node21.html

Simple process management

Let's say you wrote the following code:

```
int main() {
	int variable = 5;
	while(variable > 0) {	//This will loop forever
	}
	return 0;
}
```

How can you regain access to the shell?

- POSIX signals can help you. These signals are sent to the process and the normal execution can be interrupted during any non-atomic instruction.
- Most known signals are SIGINT (typically by typing CTRL-C; can be handled in a routine) and SIGKILL (using the kill() instruction; not interceptable)

Simple process management (continued)

In practice:

Pressed CTRL-C (SIGINT)

Adding "&" makes the process run in background

The "ps" command displays useful information about running processes

Killing a process using the "kill" command (sends the SIGKILL signal)

Process is indeed terminated

```
Sam@SH-SYSTMOD /cygdrive/d/Ulg/Cours de Computation Structures/latex/work/SDO/MM
X/R5/Coding
$ ./whileLoop.exe


Sam@SH-SYSTMOD /cygdrive/d/Ulg/Cours de Computation Structures/latex/work/SDO/MM
X/R5/Coding
$ ./whileLoop.exe &
[1] 6716

Sam@SH-SYSTMOD /cygdrive/d/Ulg/Cours de Computation Structures/latex/work/SDO/MM
X/R5/Coding
$ ps
      PID    PPID    PGID     WINPID   TTY     UID    STIME COMMAND
     7044       1    7044       7044  ?       1001 13:16:02 /usr/bin/mintty
     6716    6164    6716       7600  pty0    1001 14:32:02 /cygdrive/d/Ulg/Cour
s de Computation Structures/latex/work/SDO/MMX/R5/Coding/whileLoop
     7076    6164    7076       7816  pty0    1001 14:32:03 /usr/bin/ps
     6164    7044    6164       7252  pty0    1001 13:16:02 /usr/bin/bash

Sam@SH-SYSTMOD /cygdrive/d/Ulg/Cours de Computation Structures/latex/work/SDO/MM
X/R5/Coding
$ kill 6716

Sam@SH-SYSTMOD /cygdrive/d/Ulg/Cours de Computation Structures/latex/work/SDO/MM
X/R5/Coding
$ ps
      PID    PPID    PGID     WINPID   TTY     UID    STIME COMMAND
     4396    6164    4396       3356  pty0    1001 14:32:11 /usr/bin/ps
     7044       1    7044       7044  ?       1001 13:16:02 /usr/bin/mintty
     6164    7044    6164       7252  pty0    1001 13:16:02 /usr/bin/bash
[1]+  Terminated              ./whileLoop.exe

Sam@SH-SYSTMOD /cygdrive/d/Ulg/Cours de Computation Structures/latex/work/SDO/MM
X/R5/Coding
$
```

IPCs management

In practice:

"ipcs" displays the current IPCs

"ipcrm" removes an ipc given its ID

- "-s" for semaphores
- "-m" for shared memory segments
- "-q" for message queues (see later)

IPCs are effectively deleted

```
ms812:~/cpp/test$ ipcs

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status
0x4d280b4d 2260993    hiard      666        41         0

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x4d280b4d 1933312    hiard      666        5

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

ms812:~/cpp/test$ ipcrm -s 1933312
ms812:~/cpp/test$ ipcrm -m 2260993
ms812:~/cpp/test$ ipcs

------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch     status

------ Semaphore Arrays --------
key        semid      owner      perms      nsems

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages

ms812:~/cpp/test$
```

Creating several processes at once

The fork() command will duplicate the current process (and the variables and current instructions are also duplicated)

```
#include <stdio.h>
#include  <sys/types.h>
#include <unistd.h>
int main() {
		pid_t pid = fork();
		//pid < 0 → process creation failed
		//Anything passed this point will be executed by two processes
		//pid == 0 → newly created process (son)
		//pid > 0 → creator process (father)
		…
		return 0;
}
```

Exercise 1

a) How many processes (at most) are created by the following program?

```
void main() {
        fork();
        fork();
        fork();
}
```

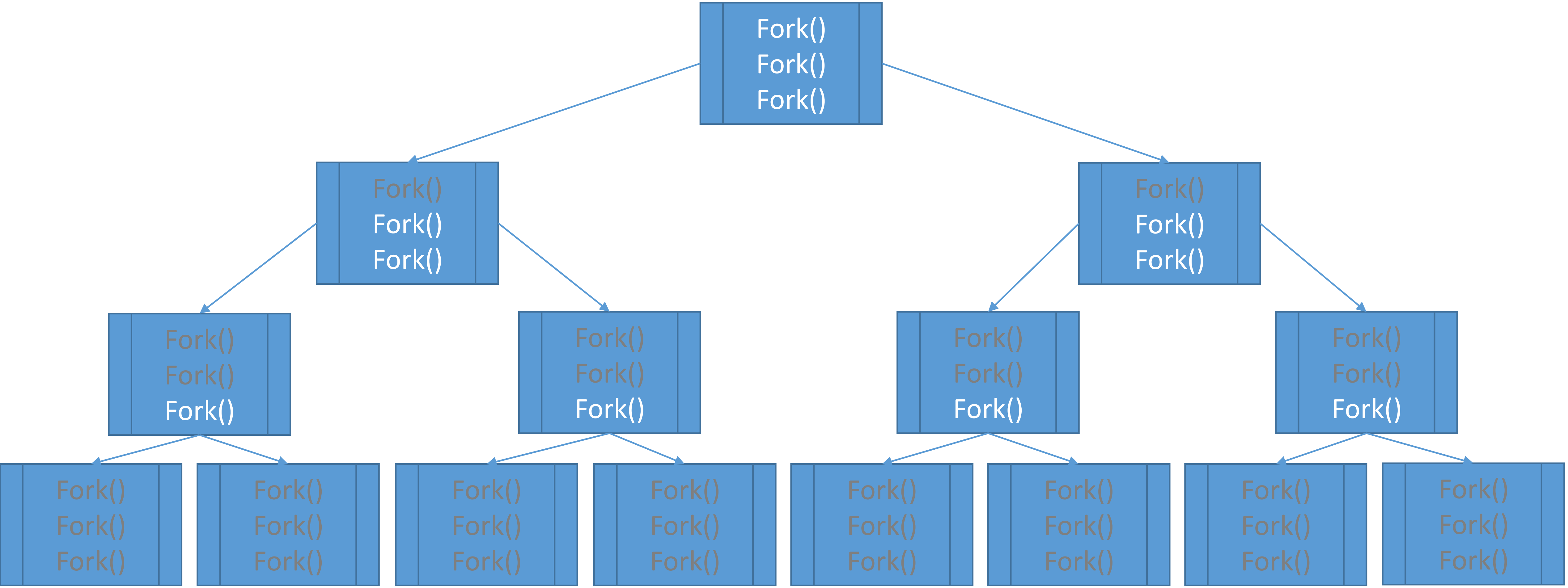
A. 3

B. 4

C. 8

D. Depends on the OS

What happens?



Exercise 1

b) How many processes (at most) are created by the following program?

```
void main() {
        for (int i = 0; i < 11; i++)
                fork();
}
```

In the previous example, 3 forks lead to $2^3 = 8$ processes.
In this example, we have 11 forks, so $2^{11} = 2048$ processes.

What if I only want 11 processes?

fork() returns a value that is the process id of the created process (the son) if we are in the calling process (the father) or 0 if we are in the created process.

```
#include <stdio.h>
#include  <sys/types.h>
#include <unistd.h>
void main() {
		pid_t pid = 0;
		for (int i = 0; i < 11; i++) {
				pid = fork();
				//If we are in the son process
				if(pid == 0) {
						//stop the loop
						i = 11;

						//Start working
						printf("I'm process number  %d\n", getpid());
				}
}
```

Exercise 2

Reminder

- Semaphores are synchronization mechanisms that can be seen as a positive (or null) integers.
- Two operations can be performed on semaphores : wait() and signal().
- signal() will increase the value of the semaphore by one unit.
- wait() will:
  - decrease the value of the semaphore by one unit if this value is > 0;
  - block the calling process otherwise until this value can be decreased again (in blocking mode) or exit without getting the lock (in non-blocking mode).
- wait() and signal() are atomic operations.

Exercise 2

A producer process writes integer numbers into a buffer zone with N slots in such a way that three consumer processes (C_1, C_2 and C_3) can read them.

The consumers must access the buffer zone one at a time in an orderly fashion: C_1, then C_2, then C_3, then C_1 and so forth.

Each element in the buffer will be read by one and only one consumer.

Use the C language to implement the code of the consumer processes and the producer process.

Exercise 2

Things to pay attention to:

- We have no hand on the context switching. All we can do is block/unblock the processes to gain some control.
- The producer cannot produce any value if the buffer is full.
- The consumers cannot consume any value if the buffer is empty.
- Each consumer must wait its turn before consuming a value.
- An element can only be consumed by one and only one consumer
- We must ensure that no deadlock (nor livelock) will ever happen.

Exercise 2

Let's first write the code for features, without any synchronization control

```
shared int in = 0;
shared int buffer[N];
```

```
//The producer calls this function in a while loop
int out = 0;
append(int x) {

        buffer[out] = x;
        out = (out+1)%N;

}
```

```
//Each consumer calls this function in a while loop
//The "who" parameter is the ID (0,1 or 2)
int value;
int take(int who) {


        value = buffer[in];
        in = (in+1)%N;


}
```

Exercise 2

The producer cannot produce any value if the buffer is full

```
shared semaphore free = N;


shared int in = 0;
shared int buffer[N];
```

```
//The producer calls this function in a while loop
int out = 0;
append(int x) {
        wait(free);
        buffer[out] = x;
        out = (out+1)%N;

}
```

```
//Each consumer calls this function in a while loop
//The "who" parameter is the ID (0,1 or 2)
int value;
int take(int who) {


        value = buffer[in];
        in = (in+1)%N;
        signal(free);

}
```

Exercise 2

The consumers cannot consume any value if the buffer is empty

```
shared semaphore free = N;
shared semaphore todo = 0;

shared int in = 0;
shared int buffer[N];
```

```
//The producer calls this function in a while loop
int out = 0;
append(int x) {
        wait(free);
        buffer[out] = x;
        out = (out+1)%N;
        signal(todo);
}
```

```
//Each consumer calls this function in a while loop
//The "who" parameter is the ID (0,1 or 2)
int value;
int take(int who) {

        wait(todo);
        value = buffer[in];
        in = (in+1)%N;
        signal(free);

}
```

Exercise 2

- Each consumer must wait its turn before consuming a value.
- An element can only be consumed by one and only one consumer

```
shared semaphore free = N;
shared semaphore todo = 0;
shared semaphore available[3] = {1,0,0};
shared int in = 0;
shared int buffer[N];
```

```
//The producer calls this function in a while loop
int out = 0;
append(int x) {
        wait(free);
        buffer[out] = x;
        out = (out+1)%N;
        signal(todo);
}
```

```
//Each consumer calls this function in a while loop
//The "who" parameter is the ID (0,1 or 2)
int value;
int take(int who) {
        wait(available[who]);
        wait(todo);
        value = buffer[in];
        in = (in+1)%N;
        signal(free);
        signal(available[(who+1)%3);
}
```

Exercise 2

Do I need to deem "**in = (in+1)%N**" a critical section?

No, because thanks to **available**, I can be sure that only one process will execute that code at a time

```
shared semaphore free = N;
shared semaphore todo = 0;
shared semaphore available[3] = {1,0,0};
shared int in = 0;
shared int buffer[N];
```

```
//The producer calls this function in a while loop
int out = 0;
append(int x) {
        wait(free);
        buffer[out] = x;
        out = (out+1)%N;
        signal(todo);
}
```

```
//Each consumer calls this function in a while loop
//The "who" parameter is the ID (0,1 or 2)
int value;
int take(int who) {
        wait(available[who]);
        wait(todo);
        value = buffer[in];
        in = (in+1)%N;
        signal(free);
        signal(available[(who+1)%3);
}
```

Exercise 2

Can I switch these two waits?

If the producer loops indefinetely, yes. If we have finite amount of data, no.

```
shared semaphore free = N;
shared semaphore todo = 0;
shared semaphore available[3] = {1,0,0};
shared int in = 0;
shared int buffer[N];
```

```
//The producer calls this function in a while loop
int out = 0;
append(int x) {
        wait(free);
        buffer[out] = x;
        out = (out+1)%N;
        signal(todo);
}
```

```
//Each consumer calls this function in a while loop
//The "who" parameter is the ID (0,1 or 2)
int value;
int take(int who) {
        wait(available[who]);
        wait(todo);
        value = buffer[in];
        in = (in+1)%N;
        signal(free);
        signal(available[(who+1)%3);
}
```

Exercise 2

Can I switch these two signals?

Yes, but it might be more "politically correct" to unlock the producer before the next consumer

```
shared semaphore free = N;
shared semaphore todo = 0;
shared semaphore available[3] = {1,0,0};
shared int in = 0;
shared int buffer[N];
```

```
//The producer calls this function in a while loop
int out = 0;
append(int x) {
        wait(free);
        buffer[out] = x;
        out = (out+1)%N;
        signal(todo);
}
```

```
//Each consumer calls this function in a while loop
//The "who" parameter is the ID (0,1 or 2)
int value;
int take(int who) {
        wait(available[who]);
        wait(todo);
        value = buffer[in];
        in = (in+1)%N;
        signal(free);
        signal(available[(who+1)%3);
}
```

Exercise 2 (solution)

1

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#include <ctype.h>
#include <string.h>

union semun {
	int val;                        /* value for SETVAL */
	struct semid_ds *buf;           /* buffer for IPC_STAT, IPC_SET */
	unsigned short int *array;      /* array for GETALL, SETALL */
	struct seminfo *__buf;          /* buffer for IPC_INFO */
};

#define SEGSIZE 10

int semid;
int *segptr;

main(int argc, char *argv[])
{
	key_t key, keysem;
	pid_t pid;
	int   shmid;
	int   id, cntr;
	union semun semopts;


	/* Create unique key via call to ftok() */
	key = ftok(".", 'M');
	keysem = ftok(".", 'S');
```

2

```c
/* Open the shared memory segment - create if necessary */
if((shmid = shmget(key, (SEGSIZE+1)*sizeof(int), IPC_CREAT|IPC_EXCL|0666)) == -1)
{
	printf("Shared memory segment exists - opening as client\n");

	/* Segment probably already exists - try as a client */
	if((shmid = shmget(key, (SEGSIZE+1)*sizeof(int), 0)) == -1)
	{
		perror("shmget");
		exit(1);
	}
}
else
{
	printf("Creating new shared memory segment\n");
}

/* Attach (map) the shared memory segment into the current process */
if((segptr = (int *)shmat(shmid, 0, 0)) == (int *)-1)
{
	perror("shmat");
	exit(1);
}

//Creating the semaphore array
printf("Attempting to create new semaphore set with 5 members\n");

if((semid = semget(key, 5, IPC_CREAT|IPC_EXCL|0666)) == -1)
{
	fprintf(stderr, "Semaphore set already exists!\n");
	exit(1);
}
```

Exercise 2 (solution; cont'd)

3

```
semopts.val = SEGSIZE;
semctl(semid, 0, SETVAL, semopts);
semopts.val = 0;
semctl(semid, 1, SETVAL, semopts);
semopts.val = 1;
semctl(semid, 2, SETVAL, semopts);
semopts.val = 0;
semctl(semid, 3, SETVAL, semopts);
semopts.val = 0;
semctl(semid, 4, SETVAL, semopts);


//Creating the three consumer processes
id = 0;
for(cntr = 0; cntr < 3; cntr++)
{
        pid = fork();
        if(pid < 0)
        {
                perror("Process creation failed");
                exit(1);
        }
        if(pid == 0)
        {
                //This is a son
                consumer(id);
                cntr = 3;
        }
        else
        {
                //This is the father
                id++;
        }
}
```

4

```
                                //We enter the producer's code
                                producer();
}

void locksem(int sid, int member)
{
    struct sembuf sem_lock={ 0, -1, 0};
    if( member<0 || member>4) {
        fprintf(stderr, "semaphore member %d out of range\n", member);
        return;
    }
    sem_lock.sem_num = member;
    if((semop(sid, &sem_lock, 1)) == -1)
    {
        fprintf(stderr, "Wait failed\n");
        exit(1);
    }
}

void unlocksem(int sid, int member)
{
    struct sembuf sem_unlock={ member, 1, 0};
    int semval;
    if( member<0 || member>4) {
        fprintf(stderr, "semaphore member %d out of range\n", member);
        return;
    }
    sem_unlock.sem_num = member;
    /* Attempt to unlock the semaphore set */
    if((semop(sid, &sem_unlock, 1)) == -1)
    {
        fprintf(stderr, "Signal failed\n");
        exit(1);
    }
}
```

Exercise 2 (solution; cont'd)

5

```
writeshm(int index, int value)
{
	segptr[index] = value;
	if(index > 0)
		printf("(Producer) Wrote %d\n", value);
	fflush(stdout);
}

int readshm(int id, int index)
{
	if(index > 0)
		printf("(Consumer %d) Read %d\n", (id+1), segptr[index]);
	return segptr[index];
}

removeshm(int shmid)
{
	shmctl(shmid, IPC_RMID, 0);
	printf("Shared memory segment marked for deletion\n");
}

producer()
{
		int out = 0;
		int value = 0;
		while(1 == 1) //While true
		{
				locksem(semid,0);
				writeshm(out+1,value);
				value++;
				out = (out+1)%SEGSIZE;
				unlocksem(semid,1);
		}
}
```

6

```
consumer(int id)
{
		int value;
		int in;
		while(1 == 1) //While true
		{
				locksem(semid,2+id);
				locksem(semid,1);
				in = readshm(id,0);
				value = readshm(id,in+1);
				in = (in+1)%SEGSIZE;
				writeshm(0,in);
				unlocksem(semid,0);
				unlocksem(semid,2+((id+1)%3));
		}
}
```

Exercise 2 (execution)

```
ms805:~/cpp/test$ ./R5_ex2
Creating new shared memory segment
Attempting to create new semaphore set with 5 members
(Producer) Wrote 0
(Producer) Wrote 1
(Producer) Wrote 2
(Producer) Wrote 3
(Producer) Wrote 4
(Producer) Wrote 5
(Producer) Wrote 6
(Producer) Wrote 7
(Producer) Wrote 8
(Producer) Wrote 9
(Consumer 1) Read 0
(Consumer 2) Read 1
(Producer) Wrote 10
(Producer) Wrote 11
(Consumer 3) Read 2
(Consumer 1) Read 3
(Consumer 2) Read 4
(Producer) Wrote 12
(Producer) Wrote 13
(Producer) Wrote 14
(Consumer 3) Read 5
(Consumer 1) Read 6
(Consumer 2) Read 7
(Producer) Wrote 15
(Producer) Wrote 16
(Producer) Wrote 17
(Consumer 3) Read 8
(Consumer 1) Read 9
(Consumer 2) Read 10
(Producer) Wrote 18
(Producer) Wrote 19
(Producer) Wrote 20
(Consumer 3) Read 11
(Consumer 1) Read 12
(Consumer 2) Read 13
(Producer) Wrote 21
(Producer) Wrote 22
(Producer) Wrote 23
```

- This will loop forever until I hit CTRL-C to send a SIGINT signal.
- But the created IPCs are not removed, so I need to manually clear the system (using 'ipcs' and 'ipcrm')
- There should be a better way to exit this program.

Exercise 2

How to quit properly

- Have a fifth process that waits user inputs (e.g. blocked on "getchar()").
- At that point, it will set a variable (e.g. "stop") in shared memory to 1.
- The producers and consumers now loop on "stop == 0".
- On loop exit,
  - The producer should produce 3 more elements (to unlock the possibly locked consumers);
  - The producer and the three consumers should make a signal to another semaphore (init. at 0) then exit;
- The fifth process makes 4 waits on that semaphore.
- Then it properly deletes the semaphores and the shared memory
  - semctl(semid, 0, IPC_RMID, 0); (for semaphores)
  - shmctl(shmid, IPC_RMID, 0); (for shared memory)

Exercise 2

Prevent global variables

- The given example uses global variables.

- This is bad, because your program behaviour depends on a variable that any function can change, so it makes test units useless (e.g. function *A* works fine, but you work on function *B* and create a side-effect that changes a global variable used by *A*. Now function *A* triggers an error and nothing indicates that the error comes from *B*).

- In our example, **int semid** and **int *segptr** should have been declared local and passed to any function that requires it (i.e. **consumer()**, **producer()**, **readshm()** and **writeshm()**) .

Exercise 3

A producer process P_1, two modifier processes M_1 and M_2 and a consumer process C_1 share a buffer of K slots.

P_1 writes integer numbers into the buffer (you can represent the number generation by using the **generate()** function).

Each number is firstly read and modified by M_1 (**foo()**), then read and modified by M_2 (**foo2()**).

Once these two modifications happened, the result is consumed by C_1 and the corresponding slot in the buffer is freed.

Use the C language to implement the code of the producer process, the modifier processes and the consumer process.

Exercise 3

Things to pay attention to:

- The producer cannot produce any value if the buffer is full.
- The consumer and the modifiers cannot consume any value if the buffer is empty.
- We must ensure that each element is handled by the processes in this order :
  $P_1 \rightarrow M_1 \rightarrow M_2 \rightarrow C_1$
- We must ensure that no deadlock (nor livelock) will ever happen.

Exercise 3

Let's first write the code for features, without any synchronization control

```
shared int buffer[K];
```

```
//The producer
int pi = 0;
produce() {
    while(true) {
        int x = generate();

        buffer[pi] = x;
        pi = (pi+1)%K;

    }
}
```

```
//The first modifier
int m1i = 0;
modify1() {
    while(true) {

        int x = buffer[m1i];
        buffer[m1i] = foo(x);
        m1i = (m1i+1)%K;

    }
}
```

```
//The second modifier
int m2i = 0;
modify2() {
    while(true) {

        int x = buffer[m2i];
        buffer[m2i] = foo2(x);
        m2i = (m2i+1)%K;

    }
}
```

```
//The consumer
int ci = 0;
consume() {
    while(true) {

        int x = buffer[ci];
        ci = (ci+1)%K;

    }
}
```

Exercise 3

The producer cannot produce any value if the buffer is full.

```
shared semaphore empty = K;
shared int buffer[K];
```

```
//The producer
int pi = 0;
produce() {
    while(true) {
        int x = generate();
        wait(empty);
        buffer[pi] = x;
        pi = (pi+1)%K;

    }
}
```

```
//The first modifier
int m1i = 0;
modify1() {
    while(true) {

        int x = buffer[m1i];
        buffer[m1i] = foo(x);
        m1i = (m1i+1)%K;

    }
}
```

```
//The second modifier
int m2i = 0;
modify2() {
    while(true) {

        int x = buffer[m2i];
        buffer[m2i] = foo2(x);
        m2i = (m2i+1)%K;

    }
}
```

```
//The consumer
int ci = 0;
consume() {
    while(true) {

        int x = buffer[ci];
        ci = (ci+1)%K;
        signal(empty);
    }
}
```

Exercise 3

The first modifier cannot modify anything if the buffer is empty

```
shared semaphore pdone = 0;

shared semaphore empty = K;
shared int buffer[K];
```

```
//The producer
int pi = 0;
produce() {
    while(true) {
        int x = generate();
        wait(empty);
        buffer[pi] = x;
        pi = (pi+1)%K;
        signal(pdone);
    }
}
```

```
//The first modifier
int m1i = 0;
modify1() {
    while(true) {
        wait(pdone);
        int x = buffer[m1i];
        buffer[m1i] = foo(x);
        m1i = (m1i+1)%K;

    }
}
```

```
//The second modifier
int m2i = 0;
modify2() {
    while(true) {

        int x = buffer[m2i];
        buffer[m2i] = foo2(x);
        m2i = (m2i+1)%K;

    }
}
```

```
//The consumer
int ci = 0;
consume() {
    while(true) {

        int x = buffer[ci];
        ci = (ci+1)%K;
        signal(empty);
    }
}
```

Exercise 3

The second modifier cannot modify anything if the buffer is empty

```
shared semaphore pdone = 0;
shared semaphore mdone1 = 0;

shared semaphore empty = K;
shared int buffer[K];
```

```
//The producer
int pi = 0;
produce() {
    while(true) {
        int x = generate();
        wait(empty);
        buffer[pi] = x;
        pi = (pi+1)%K;
        signal(pdone);
    }
}
```

```
//The first modifier
int m1i = 0;
modify1() {
    while(true) {
        wait(pdone);
        int x = buffer[m1i];
        buffer[m1i] = foo(x);
        m1i = (m1i+1)%K;
        signal(mdone1);
    }
}
```

```
//The second modifier
int m2i = 0;
modify2() {
    while(true) {
        wait(mdone1);
        int x = buffer[m2i];
        buffer[m2i] = foo2(x);
        m2i = (m2i+1)%K;

    }
}
```

```
//The consumer
int ci = 0;
consume() {
    while(true) {

        int x = buffer[ci];
        ci = (ci+1)%K;
        signal(empty);
    }
}
```

Exercise 3

The consumer cannot consume anything if the buffer is empty

```
shared semaphore pdone = 0;
shared semaphore mdone1 = 0;
shared semaphore mdone2 = 0;
shared semaphore empty = K;
shared int buffer[K];
```

```
//The producer
int pi = 0;
produce() {
    while(true) {
        int x = generate();
        wait(empty);
        buffer[pi] = x;
        pi = (pi+1)%K;
        signal(pdone);
    }
}
```

```
//The first modifier
int m1i = 0;
modify1() {
    while(true) {
        wait(pdone);
        int x = buffer[m1i];
        buffer[m1i] = foo(x);
        m1i = (m1i+1)%K;
        signal(mdone1);
    }
}
```

```
//The second modifier
int m2i = 0;
modify2() {
    while(true) {
        wait(mdone1);
        int x = buffer[m2i];
        buffer[m2i] = foo2(x);
        m2i = (m2i+1)%K;
        signal(mdone2);
    }
}
```

```
//The consumer
int ci = 0;
consume() {
    while(true) {
        wait(mdone2);
        int x = buffer[ci];
        ci = (ci+1)%K;
        signal(empty);
    }
}
```

Exercise 3

Why are **pi**, **m1i**, **m2i** and **ci** local variables (rather than in the shared memory)?

Because only one process will ever access these variables

```
shared semaphore pdone = 0;
shared semaphore mdone1 = 0;
shared semaphore mdone2 = 0;
shared semaphore empty = K;
shared int buffer[K];
```

```
//The producer
int pi = 0;
produce() {
    while(true) {
        int x = generate();
        wait(empty);
        buffer[pi] = x;
        pi = (pi+1)%K;
        signal(pdone);
    }
}
```

```
//The first modifier
int m1i = 0;
modify1() {
    while(true) {
        wait(pdone);
        int x = buffer[m1i];
        buffer[m1i] = foo(x);
        m1i = (m1i+1)%K;
        signal(mdone1);
    }
}
```

```
//The second modifier
int m2i = 0;
modify2() {
    while(true) {
        wait(mdone1);
        int x = buffer[m2i];
        buffer[m2i] = foo2(x);
        m2i = (m2i+1)%K;
        signal(mdone2);
    }
}
```

```
//The consumer
int ci = 0;
consume() {
    while(true) {
        wait(mdone2);
        int x = buffer[ci];
        ci = (ci+1)%K;
        signal(empty);
    }
}
```

Exercise 3

Can I swith these two lines (for each process)?

Yes, because the variables are not in the shared memory and thus don't need to be protected

```
shared semaphore pdone = 0;
shared semaphore mdone1 = 0;
shared semaphore mdone2 = 0;
shared semaphore empty = K;
shared int buffer[K];
```

```
//The producer
int pi = 0;
produce() {
    while(true) {
        int x = generate();
        wait(empty);
        buffer[pi] = x;
        pi = (pi+1)%K;
        signal(pdone);
    }
}
```

```
//The first modifier
int m1i = 0;
modify1() {
    while(true) {
        wait(pdone);
        int x = buffer[m1i];
        buffer[m1i] = foo(x);
        m1i = (m1i+1)%K;
        signal(mdone1);
    }
}
```

```
//The second modifier
int m2i = 0;
modify2() {
    while(true) {
        wait(mdone1);
        int x = buffer[m2i];
        buffer[m2i] = foo2(x);
        m2i = (m2i+1)%K;
        signal(mdone2);
    }
}
```

```
//The consumer
int ci = 0;
consume() {
    while(true) {
        wait(mdone2);
        int x = buffer[ci];
        ci = (ci+1)%K;
        signal(empty);
    }
}
```