Real Time Programming
Deadlock, producer / consumer model and conditional variables

A - General instructions

Exercises realization:

- Programs are in C language and must run in Linux.
- Favor a clear and readable style
- Name functions and variables with appropriate names
- Comment on your code
- Look after your indentation
- **Each exercise must be the subject of a separate C program (.c file).**
- **The answers to the questions should be placed as comments at the beginning of the file.** corresponding exercise or in a text file.
- **The compilation must not return any error or warning.**

Delivery of exercises:

- Each file must be named with the exercise number (ex: "exo1" or "exercise1").
- **Send your work by email to** roula.osta@gmail.com
- **Indicate [PTR] at the beginning of the subject (with the brackets).**
- **Indicate your first and last name! (Otherwise I do not know who to give the note!)**

B - Exercise Statements

Exercise 1 - The philosophers' dinner and the deadlock

Get the exo1_philo.c file from your instructor.

- Examine the code, compile it and run it.
- Observe that for the moment several philosophers can use the same range (represented by a semaphore) at the same time.
- Add the necessary instructions to wait for a fork or release a fork so that a fork can only be used by one philosopher at a time.
- Re-compile the code and launch it: you need to get an deadlock. Copy and paste commands entered from the console to compile and launch the program, as well as a example of returning the program when a deadlock occurs. Put these lines in comment at the end of your file.

Exercise 2 - Solution: Restrict the number of eaters

Take the code from the previous file and try to solve the problem of deadlock by limiting the number of philosophers who eat at the same time thanks to the semaphore numEating.
Question: What is the maximum value of NUM_EATING to avoid deadlock?

Exercise 3 - Alternative solution

Take again the code of exercise 1 and implement another solution to the problem of deadlock preventing the circular wait. Add comments to report and explain the instructions you have added / modified.

Exercise 4 - The Producer / Consumer Model

Here is a reminder of the structure of a program using the producer / consumer model:

```
mutex m = 1; // protect buffer access
semaphore full = 0; // presence of a buffer containing a message?
semaphore empty= n; // limit to n the number of buffers
buf_type buffer [n];
producer () {
buf_type * tp, * new;
while (1) {
new = produce (); // create a new element
P (empty); //Is there a free buffer?
P (m);
tp = obtain (buffer); // getting from buffer
V (m);
copy (new, tp);
P (m);
put (tp, buffer); // put buffer in buffer list
V (m);
V (full); // indicates the presence of a full buffer
}
}
consumer () {
buf_type * tp;
while (1) {
P (full); // is there a message to consume?  //(*)
P (m);                                        //(*)
tp =get (buffer); // get the buffer full
V (m);
consume (tp);
P (m);
put (tp, buffer); // put buffer in buffer list
V (m);
V (empty); // notify the release of the consumed buffer
}
}
```

Question: What happens if we reverse P (full) and P (m) reported by (*) in the consumer? Describe in detail what happens.

Place your answer in a text file.

Exercise 5 - Producer / consumer model, implementation 1/2

Get the file exo5_prod_cons.c from your instructor.

- Examine the code
- Comment on parent and child functions. Explain what they do.
- Compile the program and launch it.

Copy and paste commands entered on the console to compile and launch the program, as well as a example of return of the program. Put these lines in comment in your file.
Question: Comparing this program to the previous producer / consumer model, explain what prevents this program from working properly.

Exercise 6 - Producer / consumer model, implementation 2/2
Add the necessary instructions for the previous program to work as the template producer / consumer given in Exercise 3.
Question: Does the program work with multiple producers and multiple consumers? If this is not the case for your program, give an example of the result obtained in console and explain the origin of the problem.

Exercise 7 - Active waiting

- Get the exo7_var_cond.c file from your instructor.
- Examine the code, compile it and run it.
- Comment on the parent and child functions. Explain what they do.

One of the threads uses an active waiting ("polling"). Explain it and where this is done by adding comments to the code.
Copy and paste commands entered on the console to compile and launch the program, as well as an example back from the program. Put these lines in comment at the end of your file.

Exercise 8 - Conditional variables
The previous program is bad. Eliminate the active waiting and replace it with instructions using the conditional variable "ready_cakes". Have the parent warn all children when a batch of cakes is ready.