2020/2021: Sistemas Operativos (*Operating Systems*) - 2º MIEIC

8.mar.2021

Exercise Sheet 3

Processes, threads and coordination

1. Add recursion to Exercise 12 of FE1:
   - write a parallel program that runs through all the file hierarchy of the current directory, presenting at the standard out the full name of all the regular files found and their size. Use a new process for each directory in the hierarchy.
2. Study the example program `sexec` and change it to new programs:
   a. one in that `execve()` is replaced by one of the related functions of the standard C library function.
   b. another in that the parent process waits for the child to finish and prints a message showing the termination code;
   c. yet another that shows that `USER` was changed after the `exec`.
3. Create 3 processes with a parent-child-grandchild "kinship" and with `SIGINT` and `SIGUSR1` signal handlers installed. (Note: there is a picture in "Supplementary material" that might be of enlightening for this exercise.)
   a. See that all processes belong to the same *session* and the same *group* and identify the "group leader".
      - Check that the session has at least one other group, whose *leader* is also the "session leader".
        (Hint: "`pstree -p`" and "`ps j`" might be useful.)
   b. From a terminal, send any of the above signals to the leader of the parent-child-grandchild group and check if any of the other processes of the group receives them.
   c. Press CTRL-C on the keyboard and check which processes receive the corresponding signal.
4. Clarify the difference between *semaphores* and *mutexes*.
5. What is the "big deal" with the solution for accessing critical zones exemplified in the following code segment? Is this solution incorrect because it allows *race conditions* to occur?

```
    int i;     // process number: 0 or 1
    int turn;  // shared variable for access to critical region

 enter_region(i)
    { while (turn != i) ; } // wait for turn
 leave_region(i)
    { turn = 1-i; }         // give turn away
```

6. Regarding the question on slide 18 of the sheets for "*Chap 3: Coordination*" (code here repeated), explain the circumstances in which the child process waits forever.

```
 void fsin() { ... }

 int main() {
    struct sigaction ss;
 ... // set up ss with handler fsin();
```

```
sigaction(SIGUSR1, &ss, NULL);

int f = fork();
switch (f) {
   case -1: perror("fork"); exit(1);
   case 0:  pause(); printf("hello!"); break;
   default: printf("World: "); kill(f, SIGUSR1);
   }
} // main()
```

7. Study and try the sample program `count`.
   a. Check the access problems to the shared variable, which are more visible as the number of threads increases.
      (Tip: the use of pipes (`|`) and the `uniq` utility (options '`c`' and '`d`') makes it particularly comfortable to spot the problems.
   b. Change the program by writing a new program, `count_ok`, that solves the observed access problems.
8. The following code is a "non-solution" for the *Dining Philosophers* classic problem. Show that it suffers from risk of:
   i. deadlock;
   ii. *livelock* (or *starvation*).

```
#define N 5       // number of philosophers

void philosopher(int i) {
   while (TRUE) {
     think();             // philosopher is thinking
     take_fork(i);        //take left fork
     take_fork((i+1) % N);// take right fork
     eat();               // yum-yum, spaghetti
     put_fork(i);         // put left fork back on the table
     put_fork((i+1) % N); // put right fork back on the table
   }
} //philosopher
```

9. Consider the classical synchronization problem *Sleeping Barber*. Clearly identify the race conditions to avoid regarding:
   - over-booking (two customers on same seat);
   - deadlock (customer and barber both sleeping).
10. Consider the classical synchronization problem *Readers and Writers*. Clearly identify some problems regarding:
    - writer starvation or reader starvation, due to race conditions;
    - low throughput, due to excessive zeal in avoiding starvation.
11. In the sample program `count`, the access to the variable shared by multiple threads was supposed to be regulated, but the attempt was unsuccessful; you probably fixed the problem with `count_ok` (Exercise 7.b).
    Consider, now, the following changes that are meant to transform this situation into another one, of the *bounded buffer* type, and call the new program `bbcount`.
    - the `MAXCOUNT` constant, should be named `BUF_SIZE` and take the value `20`;
    - the threads created, whose total number is specified in the command line, must be of 2 types:
      - *consumer thread* - whose function **decreases** the value of the `count` variable;
      - *producer thread* - whose function **increases** that variable.
        (As a mere suggestion, create new threads, one of each type in turn, until the specified total number of threads is reached.)
    - To control the program termination, define a constant, `MAX_ITER`, with the

value `100000` and create a new shared variable `iter` that counts the number of accesses (iterations) to the (bounded) buffer variable `count`;
    - use the synchronization primitives that you find convenient to ensure that the program operates error-free and without deadlock;
    - check if the final result is correct by studying the printed messages with the `uniq` utility.

    Regarding what is not specified here, make reasonable assumptions and implement them.

12. If you solved the previous problem (`bbcount`) with semaphores as the main synchronization primitive, write another solution, this time with condition variables; if you did not use semaphores, write another solution using them.
13. ... (additional exercises on Moodle for people with free time)