Concurrent UNIX Processes and shared memory

The goal of this homework is to become familiar with concurrent processing in Unix/Linux using shared memory.

Problem: As Kim knew from her English courses, palindromes are phrases that read the same way right-to-left as they do left-to-right, when disregarding capitalization. Being the nerd she is, she likened that to her hobby of looking for palindromes on her vehicle odometer: 245542 (her old car), or 002200 (her new car).

Now in her first job after completing her undergraduate program in Computer Science, Kim is once again working with palindromes. As part of her work, she is exploring alternative security encoding algorithms based on the premise that the encoding strings are *not* palindromes. Her first task is to examine a large set of strings to identify those that are palindromes, so that they may be deleted from the list of potential encoding strings.

The strings consist of letters, $(a, \ldots, z, A, \ldots Z)$, and numbers $(0, \ldots, 9)$. Kim has a one-dimensional array of pointers, `mylist[]`, in which each element points to the start of a string. Each string terminates with the `NULL` character. Kim's program is to examine each string and print out all string numbers (subscripts of `mylist[]`) that correspond to palindromes in one file while the non-palindromes in another file.

Your job is obviously to write the code for Kim. The main program will read the list of palindromes from a file (one string per line) into shared memory and fork off processes. The children will test the index assigned to them and write the string into appropriate file, named `palin.out` and `nopalin.out`. You will have to use the code for multiple process IPC problem (Multiple processor solution in the notes) to protect the critical resources – the two files.

The multiple processes compete to get exclusive access to the file to write into, participating in the race condition. You will use the multiple process synchronization algorithm for critical section problem. Make sure you never have more than 20 processes in the system at any time. Add the pid of the child in the output. The preferred output format is:

```
PID   Index    String
```

The child process will be `exec`ed by the command

```
palin id xx
```

where `id` is the sequential identification number for the child (in the range (1,19)) and `xx` is the index number of the string to be tested in shared memory.

If a process starts to execute code to enter the critical section, it must print a message to that effect on `stderr`. It will be a good idea to include the time when that happens. Also, indicate the time on `stderr` when the process actually enters and exits the critical section. Within the critical section, wait for r_1 seconds before you write into the file, and then, wait for another r_2 seconds before leaving the critical section. r_1 and r_2 are random intervals in the range $[0, 2]$. For each child process, tweak the code so that the process requests and enters the critical section at most 5 times (`max_writes`), after which it terminates.

The code for each child process should use the following template:

```
for ( i = 0; i < 5; i++ )
{
    execute code to enter critical section;
    /* Critical section */
    sleep for random amount of time (between 0 and 2 seconds);
    write message into the file
    sleep for random amount of time (between 0 and 2 seconds);
    execute code to exit from critical section;
}
```

You will be required to create 19 separate `palin` processes from your main process. That is, the main process will just spawn the 19 child processes and wait for them to finish. It will also set up the signal handler. The *main* process also sets a timer at the start of computation to 60 seconds (make it configurable). If computation has not finished by this time, the *main* process kills all the spawned processes and then exits. Make sure that you print appropriate message(s) from both main and child processes.

In addition, the main process should print a message when an interrupt signal (`^C`) is received. Make sure that all the children/grandchildren are killed by `main` when this happens, and all the shared memory is deallocated. The grandchildren kill themselves upon receiving interrupt signal but print a message on `stderr` to indicate that they are dying because of an interrupt, along with the identification information. Make sure that the processes handle multiple interrupts correctly. As a precaution, add this feature only after your program is well debugged.

Implementation

The code for and child processes should be compiled separately and the executables be called `master` and `palin`. The program should be executed by

```
./master
```

Other points to remember: You are required to use `fork`, `exec` (or one of its variants), `wait`, and `exit` to manage multiple processes. Use `shmctl` suite of calls for shared memory allocation. Also make sure that you do not have more than twenty processes in the system at any time. You can do this by keeping a counter in `master` that gets incremented by `fork` and decremented by `wait`.

Hints

I highly suggest you do this problem incrementally. Do not start with child processes, but instead start it with a master and 2 children. Handle the shared memory first and make sure it is being done correctly. Make sure to verify that the memory was allocated and attached with appropriate error messages if something goes wrong.

You will need to set up shared memory in this project to allow the processes to communicate with each other. Please check the man pages for `shmget`, `shmctl`, `shmat`, and `shmdt` to work with shared memory.

You will also need to set up signal processing and to do that, you will check on the functions for `signal` and `abort`. If you abort a process, make sure that the parent cleans up any allocated shared memory before dying.

In case you face problems, please use the shell command `ipcs` to find out any shared memory allocated to you and free it by using `ipcrm`.

What to handin

Handin an electronic copy of all the sources, `README`, Makefile(s), and results. Create your programs in a directory called *username*.2 where *username* is your login name on hoare. Once you are done with everything, *remove the executables and object files*, and issue the following commands:

```
chmod 700 username.2

cp -p -r username.2 /home/hauschild/cs4760/assignment2
```

Do not forget `Makefile` (with suffix or pattern rules), `RCS` (or some other version control like Git), and `README` for the assignment. If you do not use version control, you will lose 10 points. I want to see the log of how the file was modified. Omission of a `Makefile` will result in a loss of another 10 points, while `README` will cost you 5 points.

Make sure that there is no shared memory left after your process finishes (normal or interrupted termination). Also, use relative path to execute the child. Your input data file should have at least 50 strings, one on each line.