IPC with fifos and shared memory

Objective

You'll have N > 1 processes, that will play the hot potato game. Some processes will start with a hot potato, and in order not to burn their hands, will immediately give it to another process, and the potato will keep changing processes, and eventually cool down after a predefined number of switches and stop switching.

How

Each process will be executed N times as:

```
./play er -b hasp otat oornot -s n a meof s har e dmemo ry -f filewith f ifona m es
```

haspotatoornot: can be 0 or a positive integer. If it is zero, the process will start with empty hands, i.e. without a hot potato. Otherwise, the number will denote how many times the potato has to be switched between processes in order to cool down.

nameofsharedmemory: as the name implies, this will be the name of the posix named shared memory segment used for internal IPC. For each potato switching between processes it should contain the number of switches left until its cool down. You can identify each potato using the pid of the process that had it originally.

filewithfifonames : this ascii file will contain the names of the fifoes that will be used to communicate among the processes. The file will contain only one fifo pathname per line; make sure they are unique.

A process will start either with a potato or not. It will open the filewithfifonames file, read the fifonames of all the processes, and attach itself to the shared memory segment. You can use the shared memory segment to make sure each process selects a different name from the list of fifos. The process will create the fifo name it has selected.

If the process starts with a potato, it will write to the shared memory segment the id and number of cooldown switches left for this potato and then send the potato to another process, using a random fifo from the file filewithfifonames. The message sent to the destination process, must contain the pid of the sending process, not the number of cooldown switches. You are free to organize the message structure as you see fit in terms of delimitation. Once sent, the sender must print something like:

```
pid=3456 sending potato number 3456 to aliVeliFifo; this is switch number 1
```

If the process does not start with a potato, or if it has already sent its potato, then it will wait for incoming potatoes from its own fifo. When a potato arrives, it will print something like:

```
pid=3498 receiving potato number 3456 from aliVeliFifo
```

The receiver will then update the number of switches left on the shared memory segment for this potato by decrementing it by 1, **you'll need to protect access to it through an unnamed posix**

semaphore. If there are still switches left for this potato, then it will send it again to some other random process. If the switch count is down to zero then it will print:

pid=3498; potatonumber 3456 has cooled down.

And then continue to wait for other incoming potatoes. If however, this was the last potato switching processes in the system (i.e. all the other potatoes are already cold), then the process will free its allocated resources, and let know all the processes on the system to terminate by sending a termination message through the fifo. It is up to you to determine the termination message's content and structure.

Evaluation
You program will be evaluated with a different input file of the same structure, an arbitrary number of potatoes, and an arbitrary number of switches per potato.

Requirements:
- In case of CTRL-C to any of the processes, all the process must terminate gracefully. -
You are not allowed to create additional files.

Tips
None.

Rules:
1) Compilation warning (with respect to the -Wall flag); we don’t accept compilation error and warnings.
2) We need a makefile to
compile the program easy
4) We need a report with details to understand your program design and problem solving.
6) If the required command line arguments are missing/invalid, your program must print usage information and exit.
8) The program crashes/freezes and/or doesn't produce expected output with normal input: then work will not be accepted.
9) We don’t want memory leak in this program.
10) We don’t want zombie processes. Number of
zombie processes must be 0.
11) We don’t want Deadlock of any kind due to poor
synchronization
12) Use of poor synchronization: busy waiting/trylock/trywait/timedwait/sleep or any other form other than those specified at project: is not be
accepted.
14) In case of an arbitrary error, exit by printing to stderr a nicely formatted informative message. Otherwise: project is not
be accepted

- report must be in English.
- Your makefile must only compile the program, not run it!
- Do not submit any binary executable files. The Project managers will compile them on their own boxes.