COMP2310/COMP6310:
Concurrent and Distributed Systems
Semester 2 2014

COMP2310 Assignment 2: A Distributed Bakery

Deadline: 17:00 on Friday 24 October

(Please report any errors, omissions and ambiguities to the course lecturer)
(Amendments to this document after its release from draft form will be marked in blue)

This assignment is worth 12% of your total course mark. It will be marked out of 24.

Submission

This assignment must be submitted electronically. This can be done using the either of the following commands (according to your enrollment):

```
submit comp2310 ass2 distBakery.c
submit comp2310 ass2 DistBakery.java *.java
```

(and similarly for `comp6310`).

Extensions and Late Penalties

See the course Assessment page.

Objectives

- To implement a concurrent algorithm using two Posix-based concurrency architectures: forks and sockets (or other form of IPC).
- To gain experience in using the message-passing paradigm with a rendezvous-like client server interaction, and be able to contrast it the with the shared object / monitor paradigm.
- To gain experience in Posix system calls and libraries related towards communication and concurrency, including using appropriate defensive programming methods.

Requirements

You are required to implement the Bakery simulation of Assignment 1 where the system components, both active and passive, are implemented using separate Unix processes which communicate by passing messages. There must be an active process for each customer and the cook. The passive process corresponds to the bakery monitor.

The active processes send *action request* messages to the monitor process, which receives these requests. It may need to defer a particular request until the bakery state is ready to perform it (this corresponds to *condition synchronization* in the message passing paradigm). When the bakery state is ready to accept the action, the monitor process must make the corresponding call to the provided BakeryState function (for Java, BakeryState method), which updates the bakery's state and logs the action. Action parameters decided by the monitor process (e.g. ticket numbers, server ids and number of buns received) must then be communicated back to the respective active process(es) (as an *action reply*). The monitor process may assume the active processes make action requests in their proper sequence.

Your main program has the same command line synopsis as for Assignment 1 (the `-f` option is no longer relevant and may be ignored) and it must call the initialization functions in the BakeryParam and BakeryState modules (for Java, it should construct a `BakeryParam` object (as per Assignment 1), and pass that object to the BakeryState constructor). It should then create all required processes and begin the simulation. It may use the C SocketWrapper library (especially if using sockets for IPC).

The `BakeryState` module will terminate the calling (monitor) process when the required number of events are logged. However it will not terminate the other (active) processes: these must terminate themselves when an attempt to read an *action reply* fails.

Your system should follow General Guidelines for Writing Monitors (read `process' for `thread') with respect to sleeps between actions and desirable action interleavings. For full marks, your system should also minimize the number of useless topup actions, i.e. the cook attempts to topup a server who already has *NB* buns.

Please note the following for your program:

- It will be tested on a 4-core Linux machine like those in the CS student labs, and it must work on those machines!
- It should be written with good programming style (C code should not produce any warnings when compiled with the `-Wall` option!). It should be adequately commented and formatted to a standard width of 80 characters, so as to enable the reader (this includes your tutor/marker!) to easily read it. A standard indentation of 2 or 4 spaces should also be used. Identifiers should be meaningful and (unless their role is trivial) commented where declared, any constants should be named rather than hard-coded, and defensive programming should be used. The latter includes checks being made for system calls that can fail due to resource limitations (anything that creates a file descriptor or process); once detected, a message should be generated with a call to the system error function `perror()` and the process should exit with a status of 2. It also includes using `assert` to perform other kinds of checks (including overflow of any fixed-length data structures that you use). Any unused file descriptors in a process should be closed as soon as possible. Note that the `BakeryState` module will check *action requests* for validity, so it is not necessary that your monitor process do this as well.
- Your program must have a preamble making a declaration about the work you have submitted. The provided template files (distBakery.c, DistBakery.java) come with such a preamble. You must add your name and student number where indicated. If you need to modify the declaration, e.g. you did in fact receive substantial assistance from another person who is not course staff, you should modify the text to say so. Code obtained from the internet will receive no marks even if acknowledged.

Marks will be allocated to approximately as follows:

Constraint	**% mark**
NC = 1, *NS = 1*, no cook	60

$NC \leq 2$ $NS = 1$, no cook	70
$NC \leq 2$ $NS \leq 2$, no cook	75
$NC \leq 2$ $NS \leq 2$	85
no restriction	100

It is recommended that you develop your program in the above stages. It is also recommended that you use fixed-length messages between pairs of processes (e.g. a customer's *action request* might be a structure containing action name, ticket number, server id, etc attributes. If an attribute is not relevant to the action, it is simply ignored when received).

The Extended BakeryParam Class

Java implementations will have to fork-exec child versions of the same program to implement the customer etc processes. The easiest way to enable a child instance to distinguish itself from the parent is via passing to the child a modified command line parameter list. However the child process still needs to know things like the value of *NC* and hence use `BakeryParam` as well. To support this, the `BakeryParam` module will parse command lines of the format:

```
java -ea DistBakery [-C cid] [-P port] [-g] [-c] [-f] [NC] [NS] [NB]
```

The `-C` and `-P` options should never be used to invoke the simulation; they should only be inserted into the command line parameter list of the child processes (before they get created). *cid* is the id of the child process (use any convention you like to number them) and *port* is the port number of the listening socket of the parent. BakeryParam has the following extra methods to the version of Assignment 1:

```
boolean isChild();  // returns whether the -C option was present
int getChildId();   // returns cid from the -C cid option (default -1)
int getPortNo();    // returns port from the -P port option (default 0)
void sleepEvents(); // sleep for some random time in 0..SleepFactor-1 ms
```

Compilation and Running of Codes On-site

On the CS student system, you can compile your C program with the command

```
$ gcc -Wall -I $COMP2310/include -o distBakery distBakery.c $COMP2310/lib
/bakery.a
```

If you use Java, your program must be compilable by the command

```
$ javac DistBakery.java
```

and be able to be run by:

```
$ java -ea DistBakery ...
```

Compilation and Running of Codes Off-site

See this file. Suffice to say it may be tricky especially on Windoze....