Operating Systems Lab Project Report

Alessandro Cheli - Università di Pisa
A.A. 2019-2020

June 10, 2020

1 Project Overview

This is the full project and not the simplified version. The following list contains an overview of the project structure. Header files are implicitly omitted from the list where a C source file is specified:

- **Source Files:** `manager.c` and `supermarket.c` files contain the main functions for the corresponding processes. `lqueue.c` contains a generic (pointer to void) implementation of a FIFO queue, which relies on `linked_list.c`. `conc_lqueue.c` contains a concurrent wrapper to the linked list implementation, relying on mutexes and condition variables defined in the standard POSIX threads library. For simplicity, concurrent access to the queue is not done through fine-grained locking. Also, as suggested by Arpaci-Dusseau in the book *Operating Systems: Three Easy Pieces*, a single lock approach may result faster because acquiring and releasing locks for each element of the list can introduce significant overhead, therefore each queue instance includes a single mutual exclusion lock. I considered an hybrid approach where a lock is used every n elements, but discarded the idea because it was adding an unnecessary layer of complexity to the project. `cashcust.c` contains definitions of the cashier and customer data types, worker threads and miscellaneous methods. They are defined in the same file because the data structures and threads reference each other, `ini.c` contains a tiny ANSI C library for loading .ini config files. It is the only external dependency in the project and the source code repository is available at `https://github.com/rxi/ini`. It is released under the MIT license.
- **Header Only Files:** `logger.h` contains logging macros. The macros report the line number and surrounding function name of the place of invocation in the source code, and allow various log levels to be filtered. The default level in production is NOTICE, while DEBUG and NEVER are used for debugging. `config.h` contains macro constants for configuration variables defaults and the IPC textual protocol message definitions. `util.h` contains miscellaneous utility functions and macros. `globals.h` contains declarations of global flags used for quitting on SIGHUP, SIGQUIT and SIGINT. Those flags are only set by signal handlers.
- **Config Files:** The `manager` and `supermarket` executables accept the `-c` command line option that specifies the path to a .ini configuration files. If some of the values are not defined in the .ini file, sane defaults are included. Example configuration files, also used in tests, can be found in the `examples/` folder.
- **Shell scripts:** `memplot.sh` contains a script for memory profiling and plotting through `GNUplot` for this report. `analisi.sh` accepts a supermarket log file and produces a short report by using the echo, cat, grep, sed, awk, cut, tr, sort, uniq and bc UNIX utilities. `autotexrebuild.sh` is a simple shell script used in development that rebuilds and shows the LaTeX report as soon as it is modified by using `inotifywait`.

2 Design Choices

The queue data structure is generic and therefore is used for both message exchange between threads and for representing customers enqueued to cashiers. The concurrent FIFO queue data structure has methods for both blocking and non-blocking dequeueing. The nonblocking method reports an error when the queue is empty, instead of waiting on a condition variable. Signal handling is done synchronously in the manager process by a designated thread, which waits on the masked signals using the `sigwait(3)` function. This was needed because the manager process must forward signals to connected client processes by using `kill(2)`, and this required access to a mutally excluded array of process IDs. Instead, the signal handler in the supermarket is a normal signal handler and is registered by using the `sigaction` system call, as it only needs to set two global flags of type `volatile sig_atomic_t`, which are in turn used by other parts of the application to check every loop iteration if the current thread should be terminated, either by emptying the cashier queues or destroying them brutally.

In the supermarket process all cashiers and customers are active entities, represented by a data structure and by a corresponding worker thread. Cashier threads are terminated and joined when they are closed, and threads are created when cashiers are opened. Customers threads are created when they enter the supermarket and destroyed when they exit. The supermarket process also contains four additional threads. There are two threads designated for message handling: `inmsg_worker` and `outmsg_worker`. The former reads messages from the socket and applies the manager's decisions of opening or closing cashiers and allowing customers out (which are always allowed). To do so, the inbound message worker must have access to most data and synchronization structures. The outbound message worker instead, simply reads messages from a monolithic concurrent queue used in most threads in the process and sends the messages to the manager process through the UNIX socket, deallocating the consumed message buffers after a failed or successful write on the socket.

The other two helper threads in the supermarket process, gather data from queues at regular, configurable intervals. The cashier poller thread regularly enqueues a message for the manager that contains the size of every cashier's queue (size is -1 if the cashier is closed). Upon receiving a queue size poll, the manager undergoes a simple decision process and answers accordingly to the undercrowded/overcrowded tresholds (called S2 and S1 in the specification). Priority is given to overcrowding (opening new cashiers when needed). The customer re-enqueue worker thread, iterates through every cashier's queue at regular intervals. Each enqueued customer has a fixed random chance to be removed from the queue and be re-enqueued to an open cashier with the shortest queue. The algorithm for choosing a cashier is in fact very simple. It just chooses the cashier which queue is the shortest. To do so, customers must have access to the whole array of cashiers data structures.

Only MT safe library functions have been used in multithreaded environments. Random values are obtained with `rand_r` and seeds are different for every thread. Time differences are measured using `clock()`;

3 IPC

As requested in the full project specification, IPC is achieved between the manager and supermarket processes by using an UNIX socket. The manager acts as the server and can handle 2 clients at the same time by default, through a manager-worker scheme, bounded by a shared variable counting active connections. Regarding the manager-supermarket IPC I have opted for a simple textual protocol because, although less efficient than direct binary communication, it is portable and easily debuggable by using standard command line tools like netcat, and could be also used for TCP/IP sockets. There is a short handshake process that must go on when a client connects to a manager server. This is because the manager needs to keep track of the PIDs of the supermarket processes in case it needs to forward a SIGHUP, SIGQUIT or SIGINT signal. After receiving a poll, the manager may decide to open or close a cashier, or to not do anything. With the values used for testing, it seems that the number of open cashiers tends to stabilize, as well as the average number of customers enqueued to a cashier at any given time. Other than queue size polls and cashier opening and closing orders, IPC is also needed when customers have to ask to "get out". Before any customer thread can terminate normally, it has to follow a "let me out - OK" communication scheme with the manager process. Customers are always allowed out of the supermarket.

4 Tools Used and Debugging

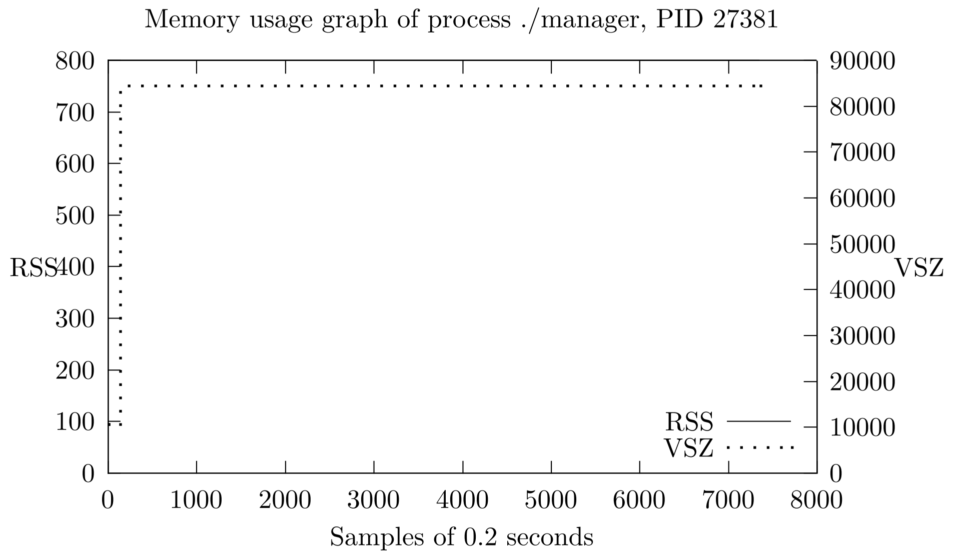
I chose `clang` version 10 as the development compiler because of the presence of additional debugging utilities such as the thread sanitizer, which has turned out an helpful tool for debugging data races and reported many potential bugs. The project is submitted with `gcc` as default because it is readily available on any system this project might be tested on. The default Makefile target builds "production" executables without debugging symbols, optimized with the `-O3` flag. To get code quality reports I have used static code analyzers such as `clang-analyzer` and `cppcheck`. `clang-analyzer` includes an utility called `scan-build` which allows to perform static analysis on the code at build time, wrapping the `make` command. Executables have been extensively tested for memory leaks by using `valgrind`. Data races have been reported by both static analyzers and clang's thread sanitizer. Deadlocks have been debugged by using `gdb`. `gdb` was signaled `SIGCONT` to pause the execution of a process when one or more threads got stuck waiting on a condition variable or a mutex. Thread backtraces were then inspected without the need for classic breakpoint debugging. `tectonic` was used as the LaTeX compiler for this report.

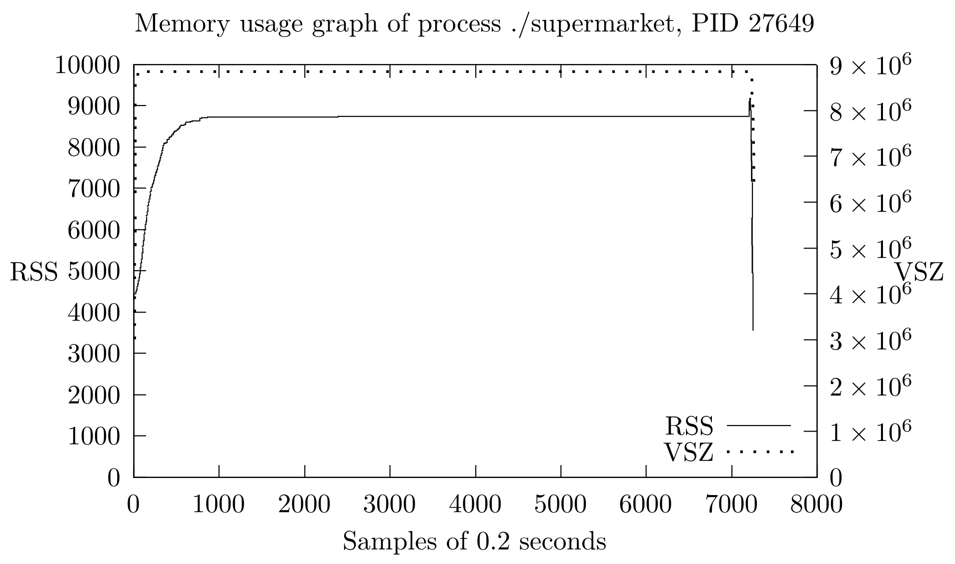
5 Testing

The project has been tested on different Linux distributions. It has not been tested on macOS and is not guaranteed to be portable because some syscall options have been used where the manual specifies that those options were introduced in Linux. Valgrind was used to make sure that there were no "definite" leaks. There may be indirect leaks when terminating the processes during the execution of some standard library functions. Bad things could happen if the manager and supermarket communicate while configured with different parameters. For simplicity, I assumed that no protection mechanism is needed and that the user will test the project only by using the same configuration file. Another method could be that the configuration values used by both processes are the same when establishing a connection.

6 Memory Usage Plots

To check for the absence of substantial memory leaks, memory usage was tracked and plotted using `GNUplot`. The script is available in `memtest.sh`. For the purpose of plotting the following two figures, both programs were tested by using large quantities of threads and resources. Units are in KB.




Memory usage graph of process ./supermarket, PID 27649
RSS
VSZ
Samples of 0.2 seconds
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
10000
1×10^6
2×10^6
3×10^6
4×10^6
5×10^6
6×10^6
7×10^6
8×10^6
9×10^6
RSS
VSZ