UNIVERSITY OF COPENHAGEN

G1

Priority queue and syscall read/write

Mikkel Enevoldsen

Kristian Høi

Simon Van Beest

February 19, 2016

A Heap-based priority Queue

1. We have implemented the queue as a max heap. Our datastructure has three items: an int* root, int size and int capacity. The root is an integer pointer to the start of our heap. For navigating in our heap, we have created three macros in queue.h: PARENT, LEFT and RIGHT. size keeps track of the size of the heap
   int capacity keeps track of the items we have allocated space to.
   In queue_pop we simply return the value in the root. Then we set the last item of the queue, the root item, and call max_heapify which restores the max heap attribute.
   In queue_push we insert our new priority at the end of our heap. Then we bottom-up compare the priority with the parent and swap if needed.
   If the number of items is equal to the capacity we allocated, we call realloc. realloc changes the size of the memory block pointer depending on the argument pointer. By this we can add more capacity.
   According to the main page realloc can fail and return a null pointer. This is why we check if the returned pointer is different from NULL. Fortunately realloc does not change the original memory block if it fails.

2. queue_pop returns 1 if the queue_pop is called on an empty queue.
   A queue can be initialized again after queue_destroy has been called, since we only free the allocated space for the queue's items.

3. We added a test target to our Makefile. By running make test it runs the command pyhon3 bounce.py
   We have tried bounce.py with a different amount of tests ranging from 50-1000 and all did pass.

Kudos System Calls for Basic I/O

1. We have created the file rw.c wich contains various systemcalls to read and write. This has been places in userland, as it is a testfile and not part of the kernel.
   To handle these calls from userland we created read.c, read.h, write.c and write.h. These are placed in kudos/proc as the documentation of kudos suggests.

2. The files mentioned above is made following the hello syscall example presented at the exercises. The fact that the userland program and the kernel is seperated suggests that the implementation is bullet proof.

3. In the file, rw.c, we use syscall to access kernel mode for the I/O processes. The testing is based on visual feedback. We call syscall_write on a predefined string, and reads the user input with syscall_read and write it with syscall_write. If the test is successful it will print the user's input in the terminal. It is only possible to enter one character (or paste a string) at a time. We call syscall_halt to avoid kernel panic.