Forfattere:
13.07-89 Rane Eriksen BVL193
dd.mm-yy Christian Efternavn NUMMERPLADE
14.06-79 Sebastian O. Jensen GJX653
Hold 1

Datalogisk Institut

Københavns Universitet

Lineær algebra for dataloger

G-Assignment 1

16. februar 2015

Indhold

Rane Eriksen BVL193
Christian Efternavn ??????
Sebastian O. Jensen GJX653

1 Task 1. A priority queue

notes

The heap data structure is a tree structure that is only concerned about the highest or lowest value which is always stored in the first entry of the tree.

Inserting a node is done by appending the node to the tail of the array which symbolizes the heap.
The newly inserted node is then compared with its parrent node - If the heap is a max heap the priority of the parrent node must be lower than the priority of the newly inserted node - which it is in our case - if it is higher, the two nodes swap places in the array, this process is done until the newly inserted node has reached the root of the tree or if the newly inserted node is lower or equal to its parrent.

The runtime is $\Theta(log\ n)$ since the newly inserted node must only compare itself with a parrent "height of the tree" times.

Removing the root node is very similar to inserting a node since removing the node is done by storing the root node in a temporary variable and moving the tail node to the root node.
The root node is then compared to its children and swapped with one of the children if either the priority of the left child node is bigger than the priority of the root node or the priority of the right child node is bigger than the priority of the root node.
This process is also repeated untill neither of the children has a node with a priority that is higher than their parrent.
Once this process is completed the the temporary variable is returned by the function.
The runtime of removing a node is also $\Theta(log\ n)$

This makes it a good structure for a priority queue, and can be used to improve the current priority queue with a running time of $\Theta(n^2)$

Implementation

void heap_initialize(heap *h)

initializes a new heap by allocating size for the root of the array and sets the size of the heap to zero which is used to indicate how many entries the root

contains.

void heap_clear(heap *h)

frees the allocated memory used by the heap and sets the "initialized" variable in the heap to zero - This is used to determine if a heap is correctly initialized in every other function other than insert.

size_t heap_size(heap *h)

returns the size of the heap which is the count of inserted nodes in the root.

void *heap_top(heap *h)

Returns the root node of the heap which is also the node with the highest priority.

void heap_insert(heap *h, void *item, int priority)

Appends the node to the heap, the newly inserted node is then compared to its parrents and sorted if needed.

void *heap_pop(heap *h)

returns the root node which is also the ndoe with the highest priority.
Before the node is returned the root of the heap is replaced with the tail of the heap and the heap is sorted in order to maintain the properties of the heap.

Unit testing

In our unit tests we are testing every function seperately.
Every test performed is done by expecting a value of comparing the expected value with the actual value.
The actual value is retrieved in two ways.
If the function returns anything else than void, then that's the value we are comparing with.
If the function returns void we are simply extracting the value from the heap since we have direct access to the heap.

The heap root has an object "void *thing" which can be anything and is a pointer to some object.

We chose, because of this fact, to only test against the expected priority instead of testing for an expected "thing".

What do we test and where?

The tests are initialized from main.c which calls:

- unit_test_heap_initialize
  Tests the allocated size of root in the heap
- unit_test_heap_clear
  Tests the alloceted size of the root in the heap
- unit_test_heap_size
  Tests the size of the heap root - Actually inserted nodes
- unit_test_heap_top
  Tests the priority of the heap roots top priority
- unit_test_heap_insert
  Tests if the heap root expands when the array is full by checking the allocated size of the heap root - It also tests the top priority of the heap root.
- unit_test_heap_pop
  Tests the top priority of the returned node by the function.

Rane Eriksen BVL193
Christian Efternavn ??????
Sebastian O. Jensen GJX653

2 Task 2. Buenos system calls for basic I/O

notes

Implementation

Which files did we modify/add?

- We modified: kernel/module.mk
  In this file, we have added read.c and write.c to FILES in order to have our implementation compiled along with the rest of buenos kernel.
- We added: kernel/read.h
  Contains the header file for the solution to syscall_read
- We added: kernel/read.c
  Contains the actual code for the solution to syscall_read
- We added: kernel/write.h
  Contains the header file for the solution to syscall_write
- We added: kernel/write.c
  Contains the actual code for the solution to syscall_write
- we modified: proc/syscall.c
  We expanded the switch to also handle syscall_read and syscall_write and we defined the variables A0, A1, A2, A3 and V0

kernel/module.mk

As stated before, we only added read.c and write.c to this file in order to have our solution compiled with the buenos kernel in order to use the solution.

kernel/read.h

In this file we we implemented the function *int syscall_read(uint32_t, uint32_t, uint32_t);* This function is explained in the next section.

kernel/read.c

int syscall_read(uint32_t f_handle, uint32_t buffer_addr, uint32_t length) takes three arguments:

1. uint32_t f_handle
   f_handle is a file handle which indicates which file to read from.
   In our implementation we do only care for the console which in the function is defined as "FILEHANDLE_STDIN", defined in proc/syscall.h
   Our implementation therefore distincts between 0 (FILEHANDLE_STDIN) and everything else.
   if the first argument is 0 the function will enter our implementation for reading from the console if it anything else than 0 our implementation will return -1 because we did not implement a solution for reading from files. -1 simply indicates an error to the function that called syscall_read

2. uint32_t buffer_addr
   buffer_addr is the address to the buffer which will contain the read bytes from the console one the function has completed.

3. uint32_t length
   length is the max size of bytes which may be read from the console - In our implementation we reserve one byte in the buffer_addr in order to append a null terminating string to the bytes read from the console.

In our implementation we are reading one byte at a time which allows our implementation to look for the return key. If the return key is pressed the function will stop reading from the console and the length of read bytes is returned. Everything read from the console is stored in buffer_addr which the caller function provided when calling syscall_read.
The caller function then has access to what was read from the console because it provided the address to the buffer to syscall_read.
The actual length of read bytes is available to the caller function in a somewhat different manner.
Once syscall_read returns the length of bytes read it is returned to proc/syscall.c however this functions returns void which means that we cannot return the value to the actual caller function, instead we are setting the variable V0 to the returned value - V0 is then returned to the caller function.

kernel/write.h

In this file we we implemented the function *int syscall_write(uint32_t, uint32_t, uint32_t);* This function is explained in the next section.

kernel/write.c

int syscall_write(uint32_t f_handle, uint32_t buffer_addr, uint32_t length) is very similar to *syscall_read* in the sence that it takes the exact same argu-

ments and they have the same meaning - The function is called in the same way and the return values are also the same.
The difference between the two functions is, that *syscall_write* writes to the console whereas *syscall_read* reads from the console.
syscall_write writes to the console one byte at a time untill it has written all the bytes provided by the buffer_addr or untill the function has written the maximum allowed bytes set by the caller function.
In order to determine if the function has written all the bytes provided - The functions looks for a null terminating string, if this is found before the maximum allowed bytes has been written, the functions stops writing to the console.

Once the functions stops writing to the console, the functions returns the length of actually written bytes.
It does not store what it wrote to the console anywhere nor does it delete the buffer provided by the caller function.

proc/syscall.c

Inside this files we defined A0, A1, A2, A3 and V0 in order to avoid writing $user\_context- > cpu\_regs[MIPS\_REGISTER\_A0]$ every time we needed the value of A0 for instance - This also ensures much cleaner and shorter code.
We implemented two cases inside the switch to handle system calls to syscall_read and syscall_write.
These function calls are provided with the arguments A1, A2 and A3 which corresponds to the file handle, buffer and the length.
V0 is set to value return by the function calls which enables the actual caller function to see the result of either syscall_read and syscall_write

unit testing