James Guerrera-Sapone - A20365203

Robert Judka - A20348051

**myMemory System Call Manual (MAN) Page**

**NAME**

myMemory – display current physical memory usage in terms of pages allocated

**SYNOPSIS**

**#include “user.h”**

**int myMemory();**

**DESCRIPTION**

**myMemory**() traverses the *page table directory* and each *page table* to **count**

**the pages based on their flagged bits**.

There are three key flag bits to account for:

* *present bit*: checks if page is accessible by process
  + page tables shared between kernel and user process
* *user bit*: checks if page is accessible by user
* *writable bit*: checks if process can write to page

**RETURN VALUE**

Zero is always returned regardless of the state of system call.

**ERRORS**

Any error that occurs while traversing the paging structure will result in a thrown

exception (most likely indicating an error elsewhere in the system).

**NOTES**

Based on the implementation of the paging mechanism in xv6, the pages allocated

Is not always an accurate representation of the memory usage of a process.

Because the page table is shared between the kernel and the user process,

pages that appear to be *present* to a process are not always actually readily

usable by the user process. Thus, the system call displays the memory

information in relation to the kernel and user space.

**myMemory System Call Implementation**

**xv6 memory details**

A screenshot of a cell phone

Description generated with very high confidence

xv6 includes various techniques when defining the page table of a user process. First, the user process shares the page table with the kernel which maintains mappings for the user space and the kernel space (as pictured above). xv6 has a fixed-size stack (of size one) and an inaccessible stack-guard page used to prevent the stack from going beyond its one page. The xv6 data/text page(s) are also readable and writable by the user process and kernel, which count towards a process’s overall allocated pages.

With the implementation used in this system call, there was no need to make any changes to the xv6 memory management code. Also, there is no separate *accessible bit* as xv6 uses the *present* and *user bits* for that purpose.

The code for determining the allocated memory and displaying the information is described below.

**xv6/proc.c**

**A screenshot of a computer screen

Description generated with very high confidence**

**myproc()** returns a pointer to a process’s *proc* structure which contains the attributes for the process. The *proc* structure has an attribute *pgdir* which will be used for finding a process’s

page table.

**A screenshot of a computer screen

Description generated with very high confidence**

Before dealing with the page table structures, the memory descriptor counters get initialized.

Each counter is described as follows:

* *kernel\_p* counts the number of pages allocated to the kernel
* *kernel\_w* counts the pages writable by the kernel
* *user\_p* counts the number of pages allocated to the user
* *user\_w* counts the pages writable by the user

The reason for splitting up all these counts is to be able to display a more fine-grained summary

of the allocated memory. Because a user process shares its page table with the kernel, we felt it

was important to differentiate between the two.

**A screenshot of a computer screen

Description generated with very high confidence**

Working under the assumption that the address space is contiguous (as described in the xv6 documentation), we can iterate through the entire *p->pgdir* array and check for any page table directories that have the *present bit* set. For any page table directory that is present, we can perform an address translation to get the virtual address.

Here, we can assume the predetermined number of page directory entries (1024) as the maximum iterations to perform.

**A screenshot of a computer screen

Description generated with very high confidence**

After getting the virtual address to the page table, and working under the same assumption as above, we can iterate through the computed *pgdir* array and check for any page table entries that have the *present bit* set. For any page table entry that is present, we increment the kernel’s allocated page counter (because the page table is shared, any present page, regardless of the other bits, is considered allocated to the kernel).

Same as above, we can assume the predetermined number of page table entries (1024) as the maximum iterations to perform.

**A screenshot of a computer screen

Description generated with very high confidence**

If the page is present in the page table, the next flag to check is the *user bit* which marks a page as specific to a user process. For any page allocated for the user process, we increment the user’s allocated page counter and check the *write bit* to see if a page is writable by the user. If the *write bit* is set, we increment the user’s writable page counter.

**A screenshot of a computer screen

Description generated with very high confidence**

We then check the *write bit* for every present page and increment the kernel’s writable page counter for any page that has the *write bit* set.

**A screenshot of a computer screen

Description generated with very high confidence**

After each page table directory and page table entry have been traversed, the memory information is displayed using the count values computed while traversing the tables. As mentioned earlier, the reason for displaying both kernel and user counts is because both entities share the same table so to get a clearer idea of the memory allocated, knowing values for both would be beneficial.