[CPEN 331. Fall 2019](https://sites.google.com/view/cpen331-fall2019/home)

* [Home](https://sites.google.com/view/cpen331-fall2019/home)
* [Calendar](https://sites.google.com/view/cpen331-fall2019/calendar)
* [Assignments](https://sites.google.com/view/cpen331-fall2019/assignments)

**Assignment 6**

**Objectives**

The goal of this assignment is to design a proper virtual memory system in OS161. The existing virtual memory system in OS161, called DUMBVM, is, umm, dumb! A process's size is limited by the number of TLB entries (i.e., 64 pages). In addition, while kmalloc() correctly manages sub-page allocations (i.e., memory requests for under 4KB), single and multiple-page requests are not properly returned to the system, meaning that address space pages are discarded after use, severely limiting the lifetime of your system.

In this assignment we will adapt OS/161 to take full advantage of the simulated hardware by implementing management of the MIPS software-managed Translation Lookaside Buffer (TLB). You will write the code to manage this TLB. You will also write the code to implement paging—the mechanism by which memory pages of an active process can be sent to disk when memory is needed, and restored to memory when required by the program. This permits many processes to share limited physical memory while providing each process with the abstraction of a very large virtual memory. Finally, you will correctly handle page reclamation, allowing your system to reclaim memory when processes exit and (in theory, at least) run forever.

**Your Mission**

* Implement the code that services TLB faults.
* Add paging to your operating system.
* Discuss how you can tune your TLB and page replacement algorithms.
* Add the sbrk() system call, so that the malloc() we give you works.

**Background and Resources**

In this section we reiterate some salient details about virtual memory, paging and TLB that you might have missed during lectures. *The content was adapted from the original published by Margo Seltzer.*

**Structure of the TLB entries:**

In the System/161 machine, each TLB entry includes a 20-bit virtual page number and a 20-bit physical page number as well as the following five fields:

* global: 1 bit; if set, ignore the pid bits in the TLB.
* valid: 1 bit; set if the TLB entry contains a valid translation.
* dirty: 1 bit; enables writing to the page referenced by the entry; if this bit is 0, the page is only accessible for reading.
* nocache: 1 bit; unused in System/161. In a real processor, indicates that the hardware cache will be disabled when accessing this page.
* pid: 6 bits; a process or address space ID that can be used to allow entries to remain in the TLB after a process switch.

All these bits/values are maintained by the operating system. When the valid bit is set, the TLB entry contains a valid translation. This implies that the virtual page is present in physical memory. ATLB miss occurs when no TLB entry can be found with a matching virtual page and address space ID (unless the global bit is set in which case the address space ID is ignored) with the valid bit set.

**Paging**

The operating system creates the illusion of unlimited memory by using physical memory as a cache of virtual pages. Paging relaxes the requirement that all the pages in a process's virtual address space must be in physical memory. Instead, we allow a process to have pages either on disk or in memory. When the process issues an access to a page that is on disk, a page fault occurs. The operating system must retrieve the page from disk and bring it into memory. Pages with valid TLB entries must always be in physical memory. This means two things:

* If you evict a page from memory, you must invalidate the corresponding TLB entry, or to make sure that you never evict pages whose translations are present in any of the TLBs,
* A reference to a page on disk will always generate a TLB fault.

At the time of a TLB fault, the hardware generates a TLB exception, trapping to the operating system. The operating system then checks its own page table to locate the virtual page requested. If that page is currently in memory but wasn't mapped by the TLB, then all we need to do is update the TLB. However, the page might be on disk. If this is the case, the operating system must:

1. Allocate a place in physical memory to store the page;
2. Read the page from disk,
3. Update the page table entry with the new virtual-to-physical address translation;
4. Update the TLB to contain the new translation; and
5. Resume execution of the user program.

Notice that when the operating system selects a location in physical memory in which to place the new page, the space may already be occupied. In this case, the operating system must **evict** that other page from memory. If the page has been modified or does not currently have a copy on disk, then the old page must first be written to disk before the physical page can be reallocated. If the old page has not been modified and already has a copy on disk, then the write to disk can be avoided. The appropriate page table entry must be updated to reflect the fact that the page is no longer in memory.

As with any caching system, performance of your virtual memory system depends on the policy used to decide which things are kept in memory and which are evicted. On a page fault, the kernel must decide which page to replace. Ideally, it will evict a page that will not be needed soon. Many systems (such as UNIX) avoid the delay of synchronously writing memory pages to disk on a page fault by writing modified pages to disk in advance, so that subsequent page faults can be completed more quickly.

**Your Mission**

* Implement the code that services TLB faults.
* Add paging to your operating system.
* Discuss how you can tune your TLB and page replacement algorithms.
* Add the sbrk() system call, so that the malloc() we give you works.

**Useful resources**

1. The first resource you should refer to, especially if you missed any lectures are the lecture slides. They are posted on the course calendar page.

2. If you ever evict a page whose virtual-to-physical translation might be in a TLB, you need to also evict the corresponding TLB entry from the right TLB. Invalidating TLB entries is called TLB shootdown. Please refer to [David Holland's writeup on TLB shootdown](http://www.google.com/url?q=http%3A%2F%2Fece.ubc.ca%2F~os161%2Fdownload%2Ftlb-shootdown.txt&sa=D&sntz=1&usg=AFQjCNFW0_8IJYf-n2cACi7kWbyeYRtkYw) for more information on how to implement TLB shootdown in OS161.

3. The design of parts of the virtual memory system is closely tied to the hardware. So as part of this assignment, you will need to become familiar with the [MIPS memory-management unit (MMU) and the translation lookaside buffer (TLB)](http://www.google.com/url?q=http%3A%2F%2Fos161.eecs.harvard.edu%2Fdocumentation%2Fsys161-2.0.2%2Fmips.html&sa=D&sntz=1&usg=AFQjCNHOKgqHd0FNRUwjcdCsMdD3yoId4g).

**Completing the Assignment**

**Step 1. Prepare**

Make sure you have a working implementation of the parts of the system that were subject of previous assignments.

If you would like to use the solution set, it will be distributed to you separately. It is your responsibility to merge it into your codebase; TAs are there to help.

**Step 2. Configure your kernel**

Starting from this assignment dumbvm files will no longer be included into your system. To make that happen, configure and compile your system as follows.

cd kern/conf

./config GENERIC

cd ../compile/GENERIC

bmake depend

bmake

bmake install

You are still welcome to look at dumbvm code, to understand what it was trying to do and how it was cutting the corners, but your VM system must be designed from scratch and will most certainly look very different from dumbvm.

**How to begin?**

Implementing the VM system might seem daunting at first. Here is provide you some tips on how to break down your workflow. These are only suggestions, you do not have to follow them.

**Understand why dumbvm is dumb.**

Study the code in kern/arch/mips/vm/dumbvm.c and kern/arch/mips/vm/addrspace.c. These files are no longer included in the configuration for this assignment, because they do barely enough to get your system running, but they certainly do not implement a sensible virtual memory system. Understand **why** this code is not the proper implementation of a VM system based on your understanding of what the VM system must do. Where is dumbvm cutting corners? Why is the way it is handling VM faults too rudimentary?

**Design your data structures**

What data structures will you use to represent page tables? What about keeping track of free memory?

**Relax the assumption of needing contiguous blocks of physical memory.**

If you have fully understood why dumbvm is dumb, you probably realized that one of the reasons is that it grabs a contiguous chunk of physical memory to back a region of virtual memory. A simple way to start implementing your VM system is to set up your VM, so that, unlike dumbvm, it is able to back the virtual regions by a collection of individual physical pages. Then add the code that returns freed physical pages to the free pool when the process exits and updates you map of free pages. ***We recommend that you begin your assignment by completing this step and making sure your kernel passes km1-km4 tests.*** *You can get a good amount of points for this assignment even if you don't finish anything else!*

Dumbvm allocated all the physical pages needed for the process virtual address space when the process is created. When you are running under physical memory pressure (and we will certainly ask you to configure sys161 with limited physical memory), you won't be able to do this. So you should modify your system, so that it gives the user process one physical page at a time, on demand -- when there is a page fault.

**Add TLB eviction**

Dumbvm crashes your kernel if the TLB runs out of space. Eliminate this problem by evicting existing TLB entries when the TLB is full.

**Implement sbrk()**

The sbrk() system call is needed to support dynamic memory allocation in user programs. In order to implement it, your VM system will need to support a virtual memory region called a heap, which grows as sbrk() calls are made. Add support to your VM system to extend the heap region on sbrk() and to properly handle TLB faults to virtual pages in that area.

**Add support for fork()**

While dumbvm had very simple support for fork(), the source files you would be using for your new VM system have empty stubs in critical functions, like as\_copy(). Your next step is to implement as\_copy() and any other function that are necessary in your system to make sure that fork() works correctly. You are not required to support the copy-on-write functionality for this assignment.

**Add support for writing pages to disk**

We recommend that you use a separate disk for swapping pages. Treating the swap disk as a raw device is one convenient option. OS161 provides a free raw disk that you can use for this purpose. Just get a vnode by calling vfsopen on lhd0raw: and write pages at the offsets dictated by your virtual memory system.

**Implement your page eviction algorithm**

Once your system is handling TLB faults and knows how to swap pages to disk you are ready to implement whatever page eviction algorithm you desire. Don't forget that if you ever evict a page whose virtual-to-physical translation might be in any TLB, you must invalidate the corresponding TLB entry. (See more on TLB shootdown in the resources section at the beginning of this document).

**Design your synchronization protocol**

**Step 3. Testing the assignment**

Test your assignment using the tests that we will use to mark it, as described below.

**Submitting the assignment**

Tag your submission as you have done in the previous assignments and push the code and tags to your repo.

**How we will grade your assignment**

100 points maximum:

* 60 for tests
* 39 for code quality
* 1 for no issues with git

**Tests (60 points max): Run all tests with 512 KB of RAM and 5 MB swap.**

* km1, km2, km3: 30 points max (10 each)
* forktest, multiexec: 15 points max (7.5 each)
* sbrktest, malloctest: 7 points max (3.5 each)
  + Skip tests 9-11 for sbrktest and test 19-21 for malloctest).
* huge, palin, matmult, triplemat, quintmat, sort, triplesort, quintsort: 8 points max (1 each).

**If any of the above tests failed, we re-run only those tests with 16MB of RAM.**

For any test that passes with 16MB of RAM, we give 1/2 the points assigned for passing it with 512KB.

**IMPORTANT:** These partial marks are only to be awarded if the you implemented you own VM system as opposed to just reusing DUMBVM.

**Code quality:** 39 points maximum.

As usual, up to 3 points. Take the final number and multiply by 13, so the maximum is 39 points.

We give **one poin**t to any submission that did not have any git/compilation/tagging or other logistical issues.

**Tips on debugging**

**Using DEBUG macros**

You probably noticed that OS161 has DEBUG macros scattered all over the code. The first argument to the macro is the flag identifying the subsystem printing the debug message. For the VM system, for example, the flag is DB\_VM. You probably know by know that to enable debugging a particular subsystem, you have to set the global variable dbflags to the flag of the corresponding subsystem. For the VM subsystem, you would do: dbflags = DB\_VM. Very conveniently, you can set the dbflags variable in the debugger. After you attach to the running kernel, simply type: set dbflags = 32. DB\_VM happens to equal 32, and that statement will turn on debugging messages in the VM subsystem.

**Using kprinf**

Using print statements are very helpful for debugging. **Beware**, however, that printing debug messages inside your VM system may cause strange and unexpected behaviour. Printing a message requires sending characters to a console, and each sent character causes the thread to acquire a semaphore and wait on a wchan. That will in turn cause a context switch and call as\_activate, which may clear the TLB (if your as\_activate is implemented that way). So you are printing debug messages within vm\_fault you could be invalidating the TLB again and again, in recursive calls to vm\_fault. If you run into this situation, you may want to disable debug messages (set dbflags = 0 in gdb) and obtain the needed information by setting breakpoints and printing variables within the debugger.

**Debugging Bus errors and other nasty traps**

Some traps are particularly nasty and difficult to debug. For example, if you get a bus error, which would happen if your code attempts to access an illegal address, the exception handler code will print the exception program counter, i.e., the code address that triggered the bus error, but not the data address, if the faulting address was indeed the culprit of the problem. The MIPS hardware simply does not make this information available.

To simplify debugging, use a tool os161-addr2line from the OS161 binary tools collection. The binary tool will tell you the source location of the instruction address that triggered the trap. For instance, if I had a bus error at address 0x80001780, as in:

panic: Bus error exception, PC=0x80001780

I can find out which line it corresponds to by invoking os161-addr2line from my ~/os161/root directory like this:

os161-addr2line -e kernel 0x80001780

The output that I see is:

/Users/sasha/os161/src/kern/compile/GENERIC/../../../common/libc/string/memcpy.c:73 (discriminator 2)

Now I can set a breakpoint in memcpy.c:73, figure out which invocation of this line causes the bus error and examine the data values that it manipulates to see if anything looks suspicious.

**Examining TLB entries**

A very simple and helpful debugging technique is to set a breakpoint in your vm\_fault function, right after you've written an entry into a TLB, examining the values that you have written and making sure that they are what you expect. Printing from that function could be tricky, as explained above, but examining the values from the debugger doesn't have this problem.

**0xdeadbeef**

kmalloc() can be used in a very helpful debug mode where it fills all free pages with a series of values 0xdeadbeef. This can be very helpful to detect situations where your kernel is accessing memory that it shouldn't or that has not been properly initialized. To enable this mode, in kmalloc.c replace the line:

#undef CHECKBEEF

with:

#define CHECKBEEF

**BE PARANOID!**

Put lots of asserts in the code, use error-checking obsessively to make sure you are not corrupting the kernel memory or doing other bad things. A common error is to try to page into a physical address of 0. When converted to the kernel address, this turns into 0x80000000, which is the address where the interrupt handlers are stored! All hell will break loose if you overwrite your interrupt handlers!

**When ALL HELL BREAKS LOOSE**

Ok, so suppose that despite having lots of asserts in your code you actually did do something very bad, and your system is behaving in a completely unexplainable fashion: your paging code never receives a notification that I/O is complete I/O, your interrupt handler goes into a weird loop, your are getting weird assertions and your debugger turns up very weird kernel addresses that don't even look like valid kernel addresses. These and many other symptoms could be a sign that you are overwriting your interrupt handler. Why would you ever do that? The problem is that in the kernel you have access to all addresses, so if you are ever trying to write a NULL address, which could happen if you didn't error-check the return value from kmalloc or if you are using uninitialized variables, you could be putting data at physical address 0, which is where your interrupt handlers are located!

A simple way to check whether or not you are overwriting the interrupt handlers is to, well, print the contents of the corresponding memory! First, get the correct contents of that memory location. Attach to the kernel with the debugger and before you let your kernel run (and screw itself up), print the contents of the memory where the interrupt handlers should be, like so:

(gdb) print (char[512])\*(char\*)0x80000000

You should see something like this:

That is the contents of the first virtual addresses of the kernel after the interrupt handlers are copied into place. If you overwrote them, you will see something different.