Operating Systems

ECE344, WINTER 2014
UNIVERSITY OF TORONTO

ASSIGNMENT 3: VIRTUAL MEMORY

Release date: Mar 14.
Due date: Apr 11, 11:59 pm. **TA responsible for this Lab**: Mike Qin (mikeandmore@gmail.com).

Objectives

After this assignment, you should:

- Become familiar with OS handling of TLB and page faults.
- Develop a solid intuition about virtual memory and what it means to support it.

Introduction

You have improved OS161 to the point that you can now run user processes. However, there are a number of shortcomings in the current system. 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`), a page that is allocated is not returned to the system, severely limiting the lifetime of your system.

In this assignment we will adapt OS161 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 correctly handle page reclamation, allowing your system to reclaim memory when processes exit. Finally, you will also write the code to implement demand paging, i.e., the mechanism by which memory pages of an active process are loaded on demand from disk, sent back 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.

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.
- [illegible] 6 bits: a context or address space ID that can be used to allow

- `pid`: 6 bits; a context or address space ID that can be used to allow entries to remain in the TLB after a context switch.

All these bits/values are updated 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. A **TLB 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 are always in physical memory. This means that 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.
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, the 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. Typically, this process is performed by a separate paging thread.

Your Mission

- Part 1
  - Implement the code that services TLB faults.

    - Implement the code to correctly allocate and deallocate kernel memory.
    - Implement the first piece of demand paging, demand loading of pages, so that the operating system loads pages from disk on demand.
- Part 2
    - Implement copy-on-write so that a child and parent can share pages after `fork` until one of them writes to a shared page.
    - Add the `sbrk()` system call, so that user-level `malloc()` works.
- Part 3
    - Implement the second piece of demand paging, swapping, so that the operating system can evict pages to disk when it runs out of memory.
- Part 4
    - Implement a paging thread that evicts dirty pages in advance so that page faults can be serviced without requiring disk I/O.
    - Tune the performance of your virtual memory system by optimizing the TLB and page replacement algorithms.

You will be tested on the four different parts above. We suggest you to spend roughly one week for each part.

Begin your assignment

Consult the ASST3 config file and notice that the `arch/mips/mips/dumbvm.c` file will be omitted from your kernel. You will undoubtedly need to add new files to the system for this assignment, e.g., `kern/vm/vm.c`. Be sure to update the file `kern/conf/conf.kern`, or, for machine-dependent files, `kern/arch/mips/conf/conf.arch`, to include any new files that you create. Take care to place files in the "correct" place, separating machine-dependent components from machine-independent components.

You should also now restrict your physical memory to 512 KB by editing the `ramsize` line in your `sys161.conf` file.

Next, you need to download some files for this assignment. Add these new files or update existing files in the `lib/libc/` and `testbin/` directories and commit these new files to your svn repository. These files implement user-level malloc and free, and add some new tests in the `testbin` directory.

Now, tag your repository exactly as shown below. The `$SVN344_SVN` variable setup is described here.

```
% svn copy $ECE344_SVN/trunk $ECE344_SVN/tags/asst3-start
```

Next configure and build OS161 for this assignment. The configuration and build instructions are similar to previous assignments.

Now you are now ready to begin assignment 3.

Design Questions

Problem 1

Assuming that a user program just accessed a piece of data at (virtual) address X, describe the conditions under which each of the following can arise. If the situation cannot happen, explain why it cannot occur.

- TLB miss, page fault
- TLB miss, no page fault
- TLB hit, page fault
- TLB hit, no page fault

Problem 2

A friend of yours who foolishly decided not to take this course, but who likes OS161, implemented a TLB that has room for only one entry, and experienced a bug that caused a user-level instruction to generate a TLB fault infinitely - the instruction never completed executing! Explain how this could happen. (Note that after OS161 handles an exception, it restarts the instruction that caused the exception.)

Problem 3

How many memory-related exceptions (i.e., hardware exceptions and other software exceptional conditions) can the following MIPS-like instruction raise? Explain the cause of each exception.

```
# load word from $0(contains zeros) offset 0x120 into register $3
lw  $3,0x0120($0)
```

TLB Handling

In this part of the assignment, you will modify OS161 to handle TLB faults. Additionally, you need to guarantee that the TLB state is initialized properly on a context switch. One implementation alternative is to invalidate all the TLB entries on a context switch. The entries are then re-loaded by taking TLB faults as pages are referenced. An alternative to invalidating the entries is to use the 6-bit address space IDs and maintain separate entries for each process in the TLB simultaneously. In this case, you will need to add support for tag handling in the TLB.

We recommend that you separate the implementation of the TLB entry replacement algorithm from the actual piece of code that handles the replacement. This will make it easy to experiment with different replacement algorithms if you wish to do so. Refer to the kernel config file section of Assignment 2 on how to add configuration options for TLB replacement policies.

Paging

In this part of the assignment, you will modify OS161 to handle page faults. When you have completed this problem, your system will generate an exception when a process tries to access an address that is not memory-resident and then handle that exception and continue running the user process.

You will need routines to move a page from disk to memory and from memory to disk.

Currently `execv` loads all program pages when the program is started. Replace this implementation so that pages are loaded on demand

Replace this implementation so that pages are loaded on demand. Similarly, change `fork` to load certain pages (e.g., text pages) on demand.

When the time comes to bring a page into memory, you will need to know which physical pages are currently in use. One way to manage physical memory is to maintain a **core map**, a sort of reverse page table. Instead of being indexed by virtual addresses, a core map is indexed by its physical page number and contains the virtual address and address space identifier for the virtual page currently associated with the page in physical memory. When you need to evict a page, you look up the physical address in the core map, locate the address space whose page you are evicting and modify the corresponding state information to indicate that the page will no longer be in memory. Then you can evict the page. If the page is dirty, it must first be written to the backing store or swap (discussed below).

Paging systems allow implementing copy-on-write (COW) based sharing of pages. With copy-on-write, virtual pages share the same physical frame, until either of the virtual page is updated. When either page is updated, a copy is made so that the physical frames are no longer shared. You need to modify fork so that the parent and child use copy-on-write pages. Mark a copy-on-write page as read-only temporarily. When a write is attempted, a page fault will occur. The page fault handler can determine that this page is temporarily read-only because the region should be writable (e.g., stack region). At this point, implement the page copying in the fault handler. You will need to change the core map implementation described above to store multiple virtual address mappings for each physical page.

Your paging system will also need to support page allocation requests generated by `kmalloc()`. You should review `kmalloc()` to understand how these requests are generated, so that your system will respond to them correctly. Currently, pages that are allocated to `kmalloc` are not returned to the system. You can use the core map to correctly implement allocated and freed pages.

You will need to decide how to implement backing store (the place on disk where you store virtual pages not currently stored in physical memory). The default `sys161.conf` includes two disks; you can use one of those disks for swapping. Please do swap to a disk and not somewhere else (such as a file). Also, be sure not to use that disk for anything else! To help prevent errors or misunderstandings, please have your system print the location of the swap space when it boots. You will have to figure out how to use the swap device (hint: use `vfs_open` and `VOP_STAT`).

You will need to store evicted pages and find them when you need them. You can maintain a bitmap that describes the space in your swap area. Think of the swap area as a collection of chunks, where each chunk holds a page. Use the bitmap to keep track of which chunks are full and which are empty. You can evict pages to empty chunks. For every evicted page in an address space, you need to keep track of the chunk in the swap area that holds the page contents. When there are too many pages to fit in physical memory, you can write (modified) pages out to swap.

On a page fault, you will need to add code to check whether a page has been swapped out. If so, the page should be read from the swap chunk and the chunk marked as available. How do you make this code work for shared pages that are swapped out?

In some systems, the writing of dirty pages to backing store is done in the background. As a result, when the time comes to evict a page, the page itself is usually clean (that is, it has been written to backing store, but not modified since then). You need to implement this mechanism in OS161 by creating a thread that periodically examines pages in memory and writes them to backing store if they are dirty.

Testing malloc() and free()

Now that OS161 has paging, you can support applications with larger address spaces. The `malloc()` and `free()` library functions are provided in the standard C library. Read the code and answer the following questions:

- **Question 1.** Consider the following (useless) program:

```
/* This is bad code: it doesn't do any error-checking */
#include<stdio.h>
int main (int argc, char **argv) {
        int i;
        void *start, *finish;
        void *res[10];
        start = sbrk(0);
        for (i = 0; i < 10; i++) {
                res[i] = malloc(10);
        }
        finish = sbrk(0);
        /* TWO */
        return 0;
}
```

  1. How many times does the system call `sbrk()` get called from within `malloc()`?
  2. On the i386 platform, what is the numeric value of `(finish - start)`?

- **Question 2.** Suppose that we now insert the following code at location `/* TWO */`:

```
        {
                void *x;
                free(res[8]); free(res[7]); free(res[6]);
                free(res[1]); free(res[3]); free(res[2]);
                x = malloc(60);  /* MARK */
        }
```

  Again on the i386, would `malloc()` call `sbrk()` when doing that last allocation at the marked line above? What can you say about `x`?

- **Question 3.** It is conventional for libc internal functions and variables to be prefaced with "__". Why do you think this is so?
- **Question 4.** The man page for malloc requires that "the pointer returned must be suitably aligned for use with any data type." How does our implementation of malloc guarantee this?

Note that the operation of `malloc()` and `free()` is a standard job interview question -- you should understand this code!

You are responsible for making the `malloc()` we give you work. This will involve writing an `sbrk()` system call.

Instrumentation and Tuning

In this section, we ask you to tune the performance of your virtual memory system. As a start, we ask that you implement the following memory system. As a start, we ask that you implement the following counters:

- The number of currently allocated physical pages.
- The number of TLB faults.
- The number of page faults.
- The number of page faults that require a synchronous write of a page.
- The total number of pages swapped over time.

You should add the necessary infrastructure to maintain these statistics as well as any other statistics that you think you will find useful in tuning your system.

Once you have completed all the problems in this assignment and added instrumentation, it is time to tune your operating system. We strongly encourage you to create a file `performance.txt` as your "lab notebook" for this section (you do not need to submit this file).

At a minimum, use the `matmult` and `sort` programs provided to determine your baseline performance (the performance of your system using the default TLB and paging algorithms). Experiment with different TLB and paging algorithms and parameters in an attempt to improve your performance. As before, predict what will happen as you change algorithms and parameters. Compare the measured to the predicted results; obtaining a different result from what you expect might indicate a bug in your understanding, a bug in your implementation, or a bug in your testing. Figure out where the error is! We suggest that you tune the TLB and paging algorithms separately and then together. You may not rewrite the `matmult` program to improve its performance.

You should add other test programs to your collection as you tune performance, otherwise your system might perform well at matrix multiplication and sorting, but little else. Try to introduce programs with some variation in their uses of the memory system.

At the end of this assignment, we will run some test programs to evaluate the overall performance of your virtual memory system.

Testing your code

By the end of this assignment, you will be able to run several programs in the `/testbin` directory. In addition to the programs that should run after assignment 2, you should be able to run `huge, malloctest, matmult, palin, parallelvm, stacktest, triplehuge, triplemat, triplesort`.

You will need to "bullet-proof" the OS161 kernel from user program errors. There should be nothing a user program can do to crash the operating system (with the exception of explicitly asking the system to halt). In particular, the `forkbomb` and the `forkexecbomb` programs should only cause your system to run out of swap but not crash or deadlock your kernel.

Strategy

The first step is understanding how TLB and page faults occur. To make this task easier, one person in your group should study TLB faults and the other should study page faults.

However, the time required to implement page fault handling is longer than the time required to implement TLB fault handling. You should divide the virtual memory implementation into several small and well-defined modules so that you can both work on it as soon as one of you has completed the TLB implementation. Get together as early as possible to share what you each have discovered.

Look at the code in `kern/arch/mips/mips/trap.c` to see how traps are handled by the kernel. (To see how traps are generated by the simulator, look at `sys161-1.14/mipseb/mips.c` available from the OS161 toolchain). Then examine the `vm_fault()` handler in `os161/src/kern/arch/mips/mips/dumbvm.c`. What changes must you add to support TLB and page faults?

Some of the key issues are:

- What will your page tables look like?
- What should you put in each PTE (page table entry)?
- What will your core map (or reverse page table) look like?
- In what order can TLB faults and page faults occur? For example, can a page fault occur without causing a TLB fault?

When you have completed your initial implementation of the TLB and virtual memory, one partner can begin experimenting with `matmult` while the other writes other test programs. Stay in touch and test each other's code.

Do your best to break your partner's implementation, then help him/her fix it. When you write test programs, think about verifying that your replacement algorithms are working correctly (i.e., "If I run this program it should generate exactly *n* TLB faults with your algorithm; does it?").

Review each other's designs. Think about how you might have implemented the different parts and compare ideas.

As your system begins to stabilize, begin to work on the performance tuning. Continue to test and fix bugs. Take turns testing and tuning. Be sure to keep a careful log of your performance experiments so you can trade it back and forth. Have fun, and good luck!

Preparing your assignment for submission

1. (Required) Once you are confident that you have completely done your assignment, run `make clean` from the `os161` directory. This will clean all generated files. Then use `svn status` in the `os161` directory to find out the status of all files. Any files that have a `?` before them are not in the repository. If you have created these files by hand, then add them to the repository using `svn add`. If these are generated files, use the instructions here to ignore these files.
2. (Required) Then run `svn commit` from the `os161` directory so that all modified files are checked in your repository. Use `svn status` in the `os161` directory again to make sure that all your modified source files are properly committed. Make sure that your partner's changes are also committed.
3. (Required) Tag your repository for the end of asst3:

```
% svn copy $ECE344_SVN/trunk $ECE344_SVN/tags/asst3-end
```

4. (No need to submit) We strongly encourage you to document your

4. (No need to submit) We strongly encourage you to document your design in a design document. Your design document should include the following:
    - A high level description of how you are approaching the problem.
    - A description of the implementation (e.g., new structures, why they were created, what they are encapsulating, what problems they solve). How did you go about adding TLB handling, paging, memory allocation and deallocation, brk,

      stack growth? Provide a description of your TLB and paging algorithms.

Testing your assignment with os161-tester

Please read the instructions for testing your code.