The third phase of Nachos is to investigate the use of virtual memory. A software-managed translation lookaside buffer (TLB) is used as a cache for page tables to provide the illusion of fast access to virtual page translation over a large address space. There is no new code for this assignment (the only change is that you need to compile with the "-DVM -DUSE_TLB" flags); the Makefile in the vm directory takes care of that. Your job is to write the code to manage the TLB and to implement virtual memory.

Page tables were used in assignment 2 to simplify memory allocation and to isolate failures from one address space from affecting other programs. For this assignment, the hardware knows nothing about page tables. Instead it only deals with a software-loaded cache of page table entries, called the TLB. On almost all modern processor architectures, a TLB is used to speed address translation. Given a memory address (an instruction to fetch, or data to load or store), the processor first looks in the TLB to determine if the mapping of virtual page to physical page is already known. If so (a TLB “hit”), the translation can be done quickly. But if the mapping is not in the TLB (a TLB “miss”), page tables and/or segment tables are used to determine the correct translation. On several architectures, including Nachos, the DEC MIPS and the HP Snakes, a “TLB miss” simply causes a trap to the OS kernel, which does the translation, loads the mapping into the the TLB and re-starts the program. This allows the OS kernel to choose whatever combination of page table, segment table, inverted page table, etc., it needs to do the translation. On systems without software-managed TLB’s, the hardware does the same thing as the software, but in this case, the hardware must specify the exact format for page and segment tables. Thus, software managed TLB’s are more flexible, at a cost of being somewhat slower for handling TLB misses. If TLB misses are very infrequent, the performance impact of software managed TLB’s can be minimal.

The illusion of unlimited memory is provided by the operating system by using main memory as a cache for the disk. For this assignment, page translation allows us the flexibility to get pages from disk as they are needed. Each entry in the TLB has a valid bit: if the valid bit is set, the virtual page is in memory. If the valid bit is clear or if the virtual page is not found in the TLB, a software page table is needed to tell whether the the page is in memory (with the TLB to be loaded with the translation), or the page must be brought in from disk. In addition, the hardware sets the use bit in the TLB entry whenever a page is referenced and the dirty bit whenever the page is modified.

When a program references a page that is not in the TLB, the hardware generates a TLB exception, trapping to the kernel. The operating system kernel then checks its own page table. If the page is not in memory, it reads the page in from disk, sets the page table entry to point to the new page, and then resumes the execution of the user program. Of course, the kernel must first find space in memory for the incoming page, potentially writing some other page back to disk, if it has been modified.

As with any caching system, performance depends on the policy used to decide which things are kept in memory and which are only stored on disk. On a page fault, the kernel must decide which page to replace; ideally, it will throw out a page that will not be referenced...
for a long time, keeping pages in memory those that are soon to be referenced. Another consideration is that if the replaced page has been modified, the page must be first saved to disk before the needed page can be brought in; many virtual memory systems (such as UNIX) avoid this extra overhead by writing modified pages to disk in advance, so that any subsequent page faults can be completed more quickly.

The Assignment

1. Implement software-management of the TLB. For this, you will need to implement some kind of software page translation, for handling TLB misses. Note that with the compile time flag -DUSE_TLB, the hardware no longer deals with page tables; thus, you need to do something about making sure the TLB state is set up properly on a context switch. Most systems simply invalidate all the TLB entries on a context switch; the entries get re-loaded as the pages are referenced. For item 2, your page translation scheme should keep track of the dirty and use flags for each page set by hardware in the TLB entry.

2. Implement virtual memory. For this, you will need routines to move a page from disk to memory and from memory to disk. We recommend that you use the Nachos file system as backing store. In order to find unreferenced pages to throw out on page faults, you will need to keep track of all of the pages in the system which are currently in use. A simple way to do this is to keep a “core map” (inverted page table) – instead of translating virtual page numbers to physical pages, a core map translates physical page numbers to the virtual pages that are stored there.

3. Evaluate the performance of your system in terms of TLB misses and page faults. Track the following three categories.

   1. Compulsory misses are those due to the first reference to a cached item; no matter what, you have to pull each referenced page off disk and put it into memory and into the TLB.

   2. Capacity misses are those due to the size of the cache; if the “working set” of the program is larger than main memory or the number of TLB entries, the program will incur misses. Capacity misses are those that would not occur in an infinite sized cache.

   3. Conflict misses are those due to the replacement policy of the cache. These would not occur if the cache used an “optimal” replacement policy, for the same program running on the same size cache.

Write some (two or three) “useful” user programs that demonstrate both a small and large number of each kind of miss, for both the TLB and paging from disk. In other words, write one test program that that demonstrates a small number of capacity TLB misses, then one that demonstrates a small number of capacity page faults, then one
that demonstrates a large number of capacity TLB misses, etc. As an example, both sort.c and matmult.c in the “test” directory demonstrate a large number of conflict misses for most standard paging policies.

For each test case, explain its performance on your system, and say how you might improve the performance of your system.

You will probably find it useful to reduce the size of main memory (in machine.h), to more quickly incur paging behavior.

**Deliverables**

The project is due 5/14. Although there is no formal walkthrough for this assignment, you might want to answer the following questions to help yourself learn the code.

1. Explain how address translation works in this assignment vs. how it worked in the previous one. Be specific. As you are reading the code, note that now the flags “VM” and “USE_TLB” are defined. This means that any code contained within an #ifdef VM until #endif will be included by the preprocessor and compiled.
2. Where is the TLB defined? How big is it?
3. When does the TLB get updated? Is there more than one place?
4. What happens on a TLB miss?
5. Do you need a page table? What data structures does the OS need to keep to handle memory references?
6. What page replacement policy will you use? Discuss implementation
7. What prefetching policy will you use? Discuss implementation
8. What cleaning policy will you use? Discuss implementation

**Getting the Code**

As mentioned above, there is no new code. You need to run a Makefile with different flags set. It is probably helpful to start with the code you implemented so far. So, you can go to your current nachos project directory (nachos-proj2). Go to the code directory and then:

```
    cp /home/nael/nachos-proj3/code/Makefile .
```

You should now be ready. Type make in the code directory and check that you get a nachos executable in the directory vm. If you run ./nachos -x ../test/halt you should get an assertion that the TLB is uninitialized.