

Section: Address Translation (cont.)

1 Announcements

- Design doc sessions coming soon ... i'll locations when i get them
- If you think there are problems with your midterm, come see me.
- project 2 ... start coding once you have your review.

Disclaimer: These notes aren't substitutes for attending lecture or reading the book. Those have been created with careful revision and review. These notes are mainly intended to be a more thorough index of the lectures and reading so that you can go off and read the full treatment on the topic and be able to ask the right questions while reading those two sources. If you have comments or suggestions please send me email and I'll fix them up.

2 Recap of Problems on the Test

I encourage you all to take a look at the solutions and see where you went wrong once you get your tests back but i want to touch upon a couple of the problems on the test and make sure you understand the solutions. Considering the barrier problem was on an old midterm, it was interesting to see the number of "unique" solutions we got. Algorithm 1, Algorithm 2, and Algorithm 3 show the correct solution to the problem. Many people just wrote out solutions with while loops because they thought that Mesa always meant use while loops. While loops don't actually work in this case because they don't work for multiple and back-to-back barriers. Another common mistake was not to reset numWaiting. So make sure you understand the solution. Its important that you understand how condition variables work and how to use them properly.

Algorithm 1 Code for threadBegin()

```
BarrierLock.Acquire()  
numThreads ++  
BarrierLock.Release()
```

The next major problem was of course enqueue and dequeue. I encourage you guys to take a look at the solutions online, but realize that if you use more than one atomic instruction the operation is no longer atomic. This is very similar to how we implemented test-and-set with atomic swap. Any implementation that used two atomic-swaps lead to incorrect results.

Algorithm 2 Code for threadEnd()

```
BarrierLock.Acquire()
numThreads -;
if numThreads == numWaiting then
    numWaiting = 0
    BarrierCondVar.broadcast()
end if
BarrierLock.Release()
```

Algorithm 3 Code for enterBarrier()

```
BarrierLock.Acquire()
numWaiting++
if numWaiting == numThreads then
    numWaiting = 0;
    BarrierCondVar.broadcast()
else
    BarrierCondVar.Sleep();
end if
BarrierLock.Release()
```

Many people seemed to think the Banker's Algorithm actually gave each thread its max resources, and then ran each thread to completion sequentially. This caused many people to say (in 5f) T1 can get a lot more of each resource than it could because they said "Just run T2, T3, and T4. After that, we'll have a bunch of A, B, and C to give to T1." Likewise, many said (in 5e) that giving 2 of A to T1 will not cause deadlock because "by the time we get to it, we have [a ton] available".

Also, remember that deadlock does not mean that the machine halts or that all threads are stuck. It only means that some of the threads are in a cyclic dependency and cannot run.

Also, when given a listing of the maximum resource a thread might need, keep in mind that this is a MAX of what it MIGHT need. Likewise, when the Banker's Algorithm "detects deadlock", it is only detecting that deadlock is POSSIBLE, not guaranteed or already present. It does indicate, however, that the state is unsafe because we cannot guarantee that deadlock will not happen.

Finally, you can NEVER change the values in the max matrix. It is a listing of the maximum possible resources that the thread will EVER need. A "request" for more of resource X is asking for more of X to be ALLOCATED. The request can only be up to max - alloc for that resource. Anything higher is illegal and should be denied outright. When the request is granted, we increase the values in the ALLOC matrix, not the MAXIMUM matrix.

I'll hand out the results in your design reviews.

3 Nachos Project 2

3.1 The Cross Compiler

The cross compiler is available for download on the class website. The inst machines also have them installed by default so you shouldn't have to do too much messing around there. The cool part about the cross compiler is that it allows you to run mips code. Unlike normal C code we will be linking against our own versions of stdio and stdlib. These will be pointing into our Nachos implementation of these libraries. Notice that every machine usually has their own implementation of stdlib and stdio based on OS internals. Try writing hello world for nachos and other simple programs to understand how to use the cross-compiler framework.

I also encourage you to test your own syscalls by writing a lot of C code. If you want to test the bullet proofing, make sure you write code that stresses the corner cases .. especially reads and writes across multiple pages! I'll be carefully looking at those parts for your design doc.

3.2 Nachos Debug Flags

For this project Nachos has a set of debug flags that will be useful. Take a look at the readme for a full description on how to use and run these debug flags. The debug flags are especially useful in tracking down what types of memory access errors you are getting. In addition it'll show you how your operating system is interacting with the MIPS code.

Nachos offers the following debug flags:

- c: COFF loader info. Useful for seeing how the different object files are being loaded
- i: HW interrupt controller info. Helps understand when interrupts are getting enabled and disabled and what new interrupts are coming in.
- p: processor info. This will be very useful in this stage of the project. It gives you what addresses are being accessed and how they are getting translated by processor. Use this debug tool.
- m: disassembly . This dumps out the stream of MIPS instructions that the processor runs
- M: more disassembly. If you like MIPS, you'll love this
- t: thread info. Gives you information on which thread is being switched in and out.
- a: process info (formerly "address space", hence a). Gives you information on what coff files and how large the different address spaces are for the different processes you create.

3.3 Processor.java

Processor.java is where the nachos MIPS simulator is stored. It is based on SPIM simulator that you worked with in CS61C. Notice that there is a large switch statement here which contains the different instructions and how to run the different instructions. Most of the instructions are self-explanatory. However there a few ones i'll go over.

3.3.1 readMem v. readVirtualMem

`readMem` is the function that gets called when you do a load and store. Notice that this function first translates the virtual address and then writes into the processor. `readMem` will always be a one-word array copy since load only deals with words. However `readVirtualMem` must be able to handle bulk (multi-page) reads and therefore you have to properly segment your reads into multiple pages. I'll be checking for this in the design reviews. So realize that the processor doesn't interact with `readVirtualMem` and `writeVirtualMem`, its only the system call handlers that use these functions.

3.3.2 SystemCall Handlers

The weird part is that they arguments are all ints (i.e. 32 bit words) Its up to you to do the proper casting of these fields so that they mean the right thing. so if you get an int that is supposed to be a virtual address we can do all the techniques we learned about partitioning that 32 bit integer and thus getting the virtual page id and then the offset so we can properly address into the physical memory.

3.4 Synchronization

In addition, to these features, make sure to use the synchronization primitives like locks! An example would be to lock the free list of pages and make sure that it is a synchronized access. Another place would be getting the next process id. There can be really weird behavior if two processes grab the same id. I'll be looking for this very carefully.

3.5 Nachos Timers

Before we were relying on the interrupts being enabled and disabled for our tests to pass but every instruction that the simulator executes will advance the clock by 1 tick. Therefore threads can get preempted very easily in nachos and we'll get a lot more preemption. Some of the timing issues will be gone but you have to be more careful with synchronization.

Questions?

4 Address Translation Continued

Now that all the dry stuff about project details and midterms are over lets talk about how address translation in a bit more detail. I think Kubi did a good job of covering the problems with address translations and different schemes in lecture so i encourage you to take a look at that. There are two important concepts that i will go into much more detail on and depth. These are the Inverted Page Table and the TLB.

4.1 Page Table Entry

So far we have talked a lot about Page Table Entries containing this meta data about what a page table contains. Lets go a little bit more in detail about the different fields that the entry could have. The first obvious field is the physical page number, needed for the translation. Lets go over the other bits given in the example in class.

- V: the valid bit tells you whether or not the entry is valid (x86 calls it Present).
- W: the writable bit tells you whether the user has access to write the page. Most code segments are marked read-only.
- U: User accessible tells you whether or not the user can access this page. For example, the Process Control Block, which is a per process state is stored in the memory. However, we don't want the user mucking with this block since that affects protection. Therefore we can mark pages as user accessible or not.
- PWT: This means that even if the page is loaded into the memory we will still write the value all the way through to disk. If you are doing DB transactions and updating bank records, you don't want a page full of updates to get tossed because the power went out.
- PCD: This states that the page can't be read into the memory and that you have to access it from disk. (like the last one, its for users that don't want their data in an unprotected state).
- A: gives information on whether this page has been recently accessed (we'll get into this much more next week).
- D: whether a page is dirty or not.
- L: some intel thing ... that i wouldn't worry about too much.

We'll go a lot more in depth in each of these as the term goes on, but realize this meta data will be very useful to us.

4.2 Inverted Page Table

Forward page tables (ie. all the page tables we have learned until now) have one fundamental problem. You have to have some method of mapping all the entries in the page table even if most of the entries are empty. This method artificially increases the complexity to handle the generality. It may seem fine for 32-bit machines but once we move to 64-bit machines (which will happen very soon) the forward page tables will get very complicated and as system designers thats bad. The reason for all these issues is that the size of the page table is proportional to the virtual address space which can be very large. What we want to do instead is make it proportional to the size of the physical address space so that the size doesn't grow out of control and hence comes in the inverted page table.

The inverted page table (IPT) is simply a hash table with all the insertion/collision problems you learned about in CS61B. The input to the hash is process id and virtual page

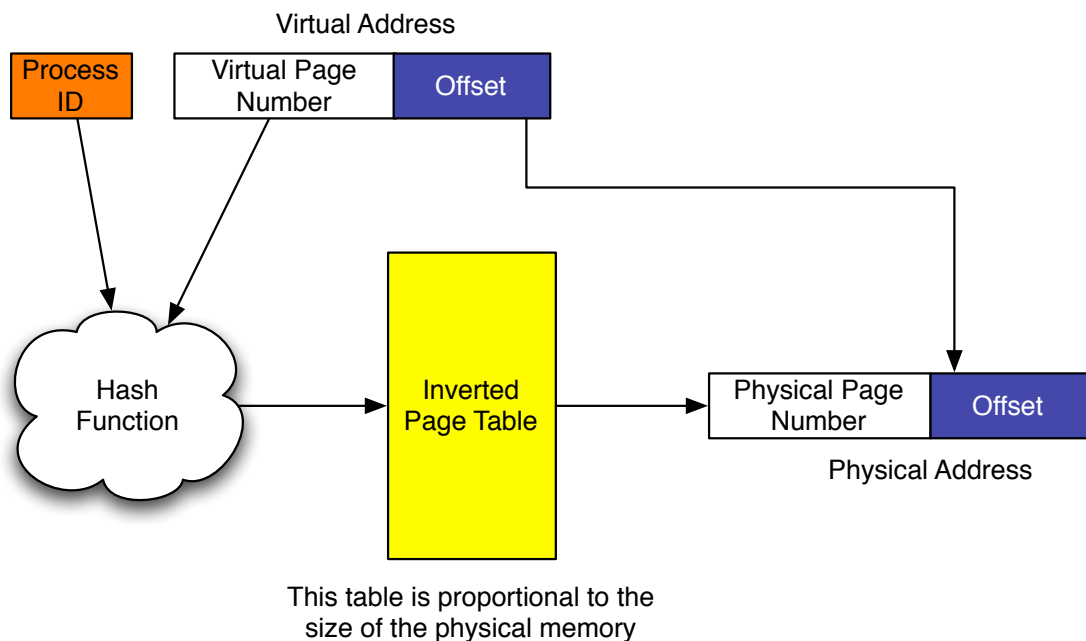


Figure 1: Inverted Page Table

number. The result in the table is the page table entry for a physical page which that virtual page number maps to. Notice that the time to access the page is now the time to run a hash function on the $\langle \text{pid}, \text{virtual page number} \rangle$ tuple and the time to access that entry. Figure 1 shows an example of a conceptual diagram of how an inverted page table would work.

Question: Why do we need to hash on both process ID as well as Virtual Page Number?

Question: Is the IPT swapped out like normal page tables during a context switch?

Question: If memory accesses were a lot slower than computation, would it make sense to use an IPT or a normal page table? Why?

Question: If computation was a lot slower than normal memory accesses, would it make sense to use an IPT or a page table? Why?

Question: Since the inverted page table is just a hash table, what does it mean to get a hash conflict? Why are hash conflicts really bad with IPTs ... what does it cause as an artifact?

4.3 Translation Lookaside Buffer (TLB)

So far, we've seen that all the mechanisms we have to translate addresses require us to do some memory look ups in order to get the actual physical address before we can actually

make the translation. But notice that there is a really nasty problem here. For every memory access we create two more just to do the translation and thus tripling our memory accesses. In modern systems where memory speed is the bottleneck, we will pay a big performance penalty. How can we get around this? Employ another big idea in systems and architecture, caching! If we use a translation once, chances are very likely that we'll use the data in that page again. So why not save that translation and so we can reuse it many times. This way when we do the translation once we store the result in a special cache. When we try to do the translation in future, this new translation is waiting for us ready to go and thus our memory access time only requires us to look up the TLB. Don't worry too much about how we do this in parallel with the caches just yet.

The TLB is filled as follows. When a processor requests an address it sends it to the TLB. The TLB either returns the entry (the fast path when there is a hit) and the translation is done. However when the entry is not in the TLB (either because its the first access or the TLB is full) then we have to go through the full mechanics of translating through the page tables and storing the resulting translation into the TLB. For future translations we now have it and can use it. Figure 2 shows an example for a 4 entry page table. Notice that when we miss in the TLB we then walk through the page table and put it in the TLB by evicting the least recently used entry (if all the slots are full). Then we try to access the TLB again and then by definition we will have a hit since the TLB was just filled in with the entry that we want. The hope is that the miss path is the uncommon path and the hit path is the common path.

Question: Since the TLB is a cache of recently used translations, do we need to flush it out on context switch? Why? What is a way around this?

Question: TLBs are usually fully associative. Normally we think of fully associative caches to be slow. Why is this ok in a TLB?

(Hint:) Think about Average Memory Access Time.

Question: What are the advantages of small TLBS?

Question: What are the advantages of large TLBs?

Algorithm 4 one way of scaling a matrix

```
int A[1024][1024]
for i = 0 to 1023 do
  for j = 0 to 1023 do
    A[i][j] = A[i][j] * 4;
  end for
end for
```

Exercise: Given that the page size is $4k$, the machine uses a two level page table, and we have a TLB that has 128 entries. How many TLB misses will Algorithm 4 incur? Algorithm 5? Lets say that the time to access memory is a constant 1 ms / per access and it takes an additional 250 ns to fill the TLB. Also assume that if we hit in the TLB, the translation time is 0 but we still incur the time to access the data. What is the total time savings between Algorithm 4 and Algorithm 5?

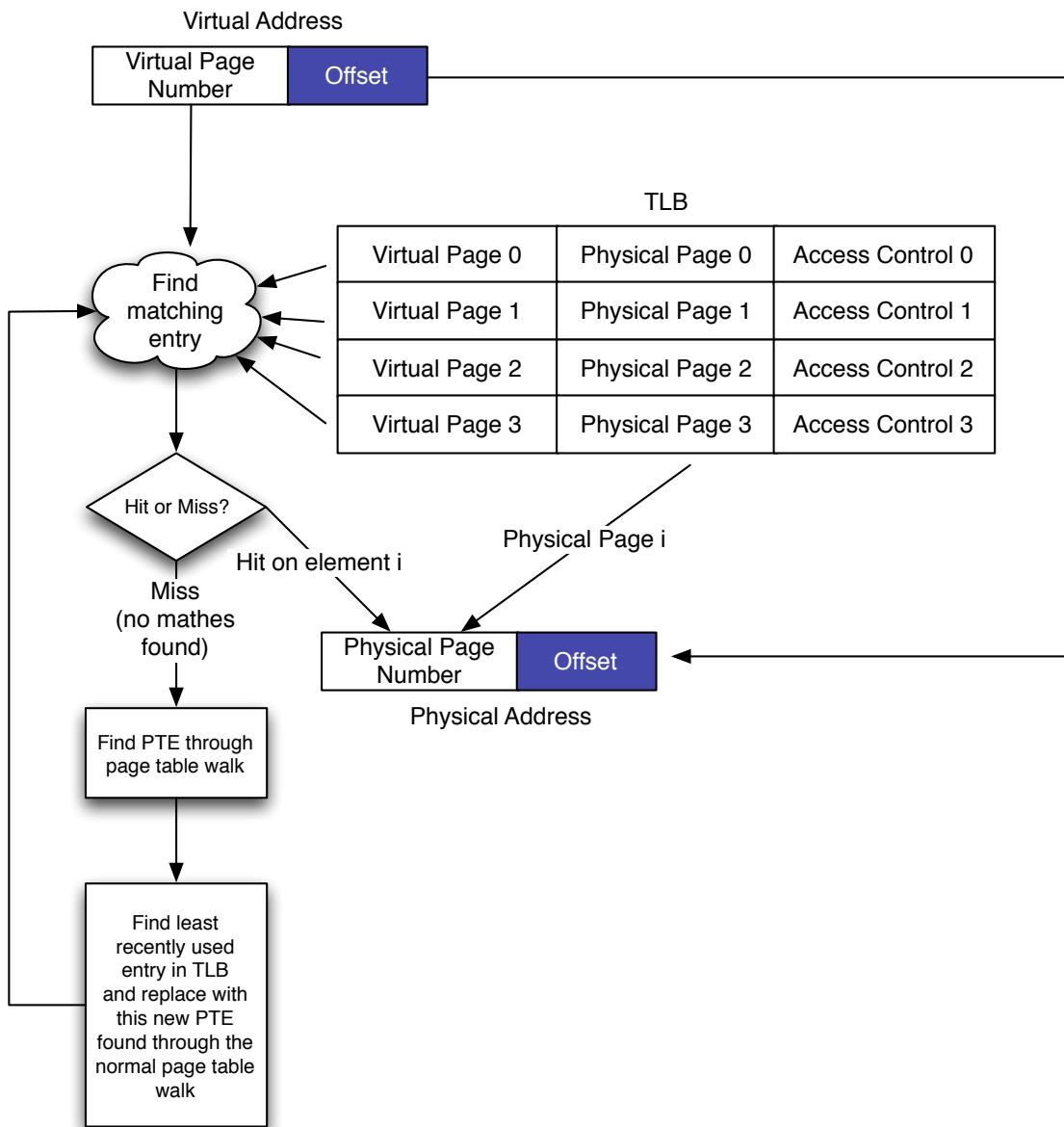


Figure 2: Translation Lookaside Buffer (TLB)

Algorithm 5 another way of scaling a matrix

```

int A[1024][1024]
for j = 0 to 1023 do
  for i = 0 to 1023 do
    A[i][j] = A[i][j] * 4;
  end for
end for

```
