

## Section: Demand Paging

### 1 Announcements

- Projects due thursday
- Questions?

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: Why do we need demand paging

Lets go back to one of the fundamental things that an OS provides: the illusion that every process has 4GB of memory (for a 32 bit machine). Now most of us don't have 4GB of memory so we need to find a way to fit all the applications in the memory space such that they all fairly share the physical memory. In addition we want to make sure that the sum of the memory that all the applications can use can easily exceed the total amount of memory we have.

**Question: Why is it important that the total amount of memory needed by all the applications is greater than the total physical memory?**

Thus we need to figure out a way for different processes to be able to map their address space to their physical pages that are valid. The idea we use is paging: that is evict pages of other applications or our application to make room for data that is currently needed by the processor. There aren't that many subtle ideas here, so I encourage you to read the book on why it is important to page. The main focus of this section will be analyzing different algorithms that pick the right page to evict and different access streams that can lead to a really bad page faulting scheme.

### 3 What is a Page Fault

Before we talk about paging algorithms and how to draw stuff out of the memory it is important to talk about what exactly constitutes inside a page fault and how they occur. When a processor does a load or store it first asks the TLB for a cached copy of the translation from virtual memory to physical memory. The TLB either has the translation

(the fast path) or has to declare a TLB miss and let the Kernel walk the page tables to find the translation.

In the normal case we just walk through our page tables to find the translation to refill the TLB. This assumes that that physical page is actually mapped in the memory. However, what happens when that's not the case and we need to pull out the page from disk and put it back into the memory so that we can start using it. If we know that the page is on disk, it means that the physical page itself is not in the memory and therefore we have an invalid page table entry for that page. Assuming, we are in a steady state and all the physical memory pages are full we need to come up with a way to evict other pages and find out what is the best one to evict. (If the system is not in steady state and there are free pages available, it is trivially easy to find an empty page) This is again the study of many research groups around the country simply because the best strategy is very much application dependent and there is no single right or wrong answer.

## 4 Replacement Strategies

There are many different strategies of page replacement algorithms we can use. I'll just highlight them here but encourage you to read the book for a full treatment about what their advantages and disadvantages are.

- **FIFO.** In FIFO replacement the page we choose to replace is the oldest page to arrive. I.e. the page that has been using the memory the longest.

**Question: What are the advantages of FIFO? What are the disadvantages?**

- **MIN or OPT.** This the "optimal" replacement strategy. The page that is evicted is the page that won't be used for the longest in the *future* ... sounds a bit difficult. One of the interesting suggestions raised in class is pre-profiling your applications so that you know.

**Question: Is it possible to pre-profile your application to get optimal replacement strategy. What assumptions does this make? How feasible are they?**

- **LRU.** The least recently used strategy replaces the page that hasn't been used for the longest time. The implication is that the page won't be used in the near future either so it's a good candidate to evict.

**Question: Under what conditions will LRU break down and become the worst policy? Why then is it still a very popular replacement strategy?**

*(Hint:)The problem is called thrashing.*

*Exercise: Given the page replacement policy's above and 4 physical pages, how many faults will the strategies incur if the access pattern for virtual pages is:*

*1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2?*

*This problem is a simplified version of 9.13 in the book.*

## 5 LRU replacement approximations

In the last section we said that LRU is an approximation for the optimal “longest time till next use” algorithm (also named OPT or MIN). However, even LRU is a bit tricky to implement since it requires the use of some sort of time stamp and an  $O(n)$  algorithm to search through all the pages in the memory to try to find the least recently used page and we have to traverse the list every single time. So we try to do a bit better by approximating LRU. Enter the clock algorithm.

### 5.1 The Clock Algorithm

In the clock algorithm we basically give the pages a chance to fight for its survival in the memory. Every time a page is accessed its reference bit is set to 1. In order to replace the page we follow the algorithm in Algorithm 1.

**Question:** Why is it called the *second-chance* clock algorithm?

---

**Algorithm 1** Second chance clock algorithm

---

```
... assume that the reference bits are set on each access ...
... assume that the clockHand variable is initialized to 0 on bootup ...
... assume refBit[i] refers to the reference bit in the  $i^{th}$  physical page ...
while refBit[clockHand] == 1 do
    refBit[clockHand] = 0
    clockHand = (clockHand + 1) % numPhysicalPages
end while
choose physicalPages[clockHand] for replacement.
clockHand = (clockHand + 1) % numPhysicalPages
```

---

### 5.2 The N-chance Clock Algorithm

The problem with the clock algorithm as described above is that it only gives the pages one-round of evictions before they get kicked out. What happens if the frequency of use is just beyond this threshold, is it fair to treat it the same category as a page that has been used once and will be never used again? The answer for us designers is no and we can do better. Enter the N-chance algorithm. In the N-chance algorithm the page has to have not been used in N sweeps and therefore it is an even better candidate for replacement.

**Question:** What is the design tradeoff here. What happens with very large N?

Algorithm 2 shows the N-chance clock algorithm in all its glory.

**Exercise:** Given the same access pattern in the above exercise, calculate which pages get evicted when with the second-chance clock algorithm and the N-chance clock algorithm with  $N=4$  and  $N=5$ . In each case how many sweeps do we have to do?

---

**Algorithm 2** N-chance clock algorithm

---

```
... assume that the reference bits are set on each access ...
... assume that the clockHand variable is initialized to 0 on bootup ...
... assume refBit[i] refers to the reference bit in the  $i^{th}$  physical page ...
... assume that counter[i] refers to the sweep counter n the  $i^{th}$  page...
while true do
  while refBit[clockHand] == 1 do
    refBit[clockHand] = 0
    counter[clockHand] = 0
    clockHand = (clockHand + 1) % numPhysicalPages
  end while
  if counter[clockHand] == N then
    choose physicalPages[clockHand] for replacement.
    break out of loop.
  else
    counter[clockHand] ++
    clockHand = (clockHand + 1) % numPhysicalPages
  end if
end while
```

---

### 5.3 An enhanced clock algorithm

In the above section we make no distinction on whether a page is dirty or not however this is a very big distinction that could buy us a huge performance win. The reason is that dirty pages must be written to disk *before* the new pages can be read off the disk. While clean pages allow us to simply invalidate the page (assuming we don't care about security and zeroing out the page). Thus most of the time we can simply use one disk I/O instead of two which results in a big savings. The enhanced version of the clock algorithm takes advantage of this.

### 5.4 Second Chance List Algorithm

In lecture, we saw that the VAX employed an algorithm in which it used two lists. The basic idea here is that we keep two lists: the active and the second chance. When we get a page fault and have to evict a page from the active list we push out the top page and then add a page at the bottom of the list (think FIFO). The page that gets pushed out gets put onto the second chance list (the bottom of another stack). There are now two possibilities: (1) when we take a page fault on the active list the page is actually in the second chance list so it is simply brought back to the active list or (2) the page is neither on the active list nor the second chance list so we toss out the oldest page which is the top element and find a new page to replace.

**Question:** What are the pros and cons of this approach?

**Question:** How many “artificial” page faults are created with this scheme?

## 5.5 Clock Algorithm with a free list

The final replacement algorithm we saw in lecture was a free-list based approach to replacement. When the clock algorithm finds a page to evict we put it on a free list and then pull a page off our free list to use for the new page. The advantage of the free list is that it is a list of pages that have been given the following notice “you are about to be evicted, pack up your belongings and ship them out if needed.” Thus these pages can start initiating disk I/O if they were marked as dirty and when it comes time to actually grab a free page, the OS doesn’t have to wait for the data to get flushed out, it can simply invalidate and move on. So in a sense this is merely an optimization to initiate the disk I/O early so the entire system doesn’t have to stall waiting for the page to get flushed out. The lecture covered this idea pretty well so I won’t go into too much detail. If you have taken or are taking CS152, this is very similar to the idea of a victim cache.

## 6 Implications of Paging

Until now we have only seen this idea of a page table and inverted page tables. In essence they are tables that give you information about what process is using what page in the page table. However now we have to deal with the possibility that the pages don’t actually exist so we need to set the page table entries to invalid. Seems like the simple thing to do. However lets take a look at what the implications of using paging are.

### 6.1 Core Map

How do we manage sharing? We need a way to make sure that when we evict a page all processes that had some sort of reference to that page are marked as invalid. This necessitates a map of the physical memory such that for each entry of the map (i.e. the physical page) we know what processes are using that page. This is termed the core map.

### 6.2 Lazy Loading

Most computers (like many of their designers) are very lazy about doing stuff and only do stuff if they absolutely have to. Lets say that we want to load a process into the memory its code sections. The problem however is that we aren’t really sure if we need to load in all the code. In MS word, the chances that you’ll use all of the features word has to offer in one-sitting are pretty rare and most of the time you’ll be using subsets of the features. So instead, we mark the page table entry for that section of the object file (or coff file as you guys know it) as invalid but we also give it a pointer to where we can find it on disk. Thus on the first access to this page of code, it is loaded from the file on disk and into the memory (and thus only loaded when needed). When it comes time to evict the page, we treat it like normal memory and flush it out to the swap space (if it is read only, flush it the first time around only). Thus with this scheme parts of the application that are never used are never even brought through the memory system and just stay on disk. We can thus load only portions of the code that will be used.

### 6.3 Precise Interrupts

Lets say that process is trying to issue a load and it has to take a page fault. What do we do? In our standard world from CS61C this is very easy. We simply stall the pipeline until the page fault has been cleared up and move on with our lives. However modern architectures are not as simple as CS61C makes them out to be. Most modern processors employ a mechanism called out-of-order execution. The idea is that you find instructions that don't have dependencies on each other and then execute them out of order and leverage the instruction level parallelism available in the processor. However what about a page fault in this context: how do we make sure that the instructions after our load are never executed and instructions before the load are done. This is where making precise interrupts gets tricky. In order to fix this problem the use a device called a reorder buffer. The details of this are not really relevant to this class however just realize that on modern processors creating a precise interrupt (an interrupt so that all the instructions before the instruction that caused the exception or the occurrence interrupt are completely finished and all the instructions after have the effect of not even starting). If you want to get the full gory details, take CS152 and CS252.