Demand Paging + Disk!

https://tinyurl.com/CS111A-water
Today’s reflection

● What is your favorite food?
Game Plan

- Demand Paging
- Admin
- Magnetic Disk
Addressing Review

- Let’s say I’m running on a system with 3-bit addresses. What does this look like?
Addressing Review

- Let’s say I’m running on a system with 3-bit addresses. What does this look like?

Bit 0  Bit 1  Bit 2
Addressing Review

- Let’s say I’m running on a system with 3-bit addresses. What does this look like?
Addressing Review

- Let’s say I’m running on a system with 3-bit addresses. What does this look like?

- Address 1
  - Bit 0: 0
  - Bit 1: 0
  - Bit 2: 1
Addressing Review

- Let’s say I’m running on a system with 3-bit addresses. What does this look like?

```
0 1 0
```

Address 2

- Bit 0
- Bit 1
- Bit 2
Addressing Review

- Let’s say I’m running on a system with 3-bit addresses. What does this look like?

Bit 0  Bit 1  Bit 2

Address 6
Addressing Review

- Let’s say I’m running on a system with 3-bit addresses. What does this look like?

1 1 0

Bit 0 Bit 1 Bit 2

Address 6
Addressing Review

● Let’s say I’m running on a system with 3-bit addresses. What does this look like?

  ○ With 3 bits, how many unique addresses can I make?

<table>
<thead>
<tr>
<th>Bit 0</th>
<th>Bit 1</th>
<th>Bit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Address 6

\[2^N\] unique numbers for \(N\) given bits

If given 12 offset bits in a page \(\Rightarrow \) 2^{12} addresses in the page
Addressing Review

- Let’s say I’m running on a system with 3-bit addresses. What does this look like?
  - With 3 bits, how many unique addresses can I make?
    - 8 unique values (0 -> 7). This is $2^3$.

```
1 1 0
```

Address 6

Bit 0 Bit 1 Bit 2
Addressing Review

- Let’s say I’m running on a system with 3-bit addresses. What does this look like?
  - To generalize, given $N$ bits, we can represent $2^N$ unique numbers!

<table>
<thead>
<tr>
<th>Bit 0</th>
<th>Bit 1</th>
<th>Bit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Address 6
Addressing Review

- Let’s say I’m running on a system with 3-bit addresses. What does this look like?
  - To generalize, given $N$ bits, we can represent $2^N$ unique numbers!

<table>
<thead>
<tr>
<th>Bit 0</th>
<th>Bit 1</th>
<th>Bit 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Address 6

How many unique addresses can I represent with 4-byte addresses?
Demand Paging

- With the **paging** implementation of virtual memory, all of memory was split up into 4KB regions called **pages**

Green pages are active
Demand Paging

- With the **paging** implementation of virtual memory, all of memory was split up into 4KB regions called **pages**

Example stack page for Process A

(how did I know it was a stack page?)
Demand Paging

- With the **paging** implementation of virtual memory, all of memory was split up into 4KB regions called **pages**

In this figure, the orange regions in process A (and the blue regions in process B) are currently **unmapped** (i.e. unallocated)

The allocated regions of process memory don’t need to be contiguous. For example, adjacent stack pages in process B can be in very different places in physical memory

Physical addresses can be anywhere, even on disk!
Demand Paging

- Demand Paging fixes the issue of page map size by *only creating a page map entry* (i.e. allocating a page of physical memory) when more memory is requested by the user.
  - This is a “lazy” approach to memory management, because we don’t act until we have to.
  - When the user does anything that requires more memory, the OS stops the program, creates a new Page Map entry out of physical memory, and maps it into the virtual address space of the process.
    - This is called a Page Fault
Demand Paging

- Demand Paging fixes the issue of page map size by *only creating a page map entry* (i.e. allocating a page of physical memory) when more memory is requested by the user.
  - This is a “lazy” approach to memory management, because we don’t act until we have to.
  - When the user does anything that requires more memory, the OS stops the program, creates a new **Page Map** entry out of physical memory, and maps it into the virtual address space of the process.
    - This is called a **Page Fault**
  - Similarly, if a page that was **swapped out to disk** gets referenced by a process, a **Page Fault** is generated that must load the page in from disk and then re-map it into the **Page Map**.
Game Plan

- VM Review
- Clock Algorithm
- Admin
- Final Exam Review!
Virtual Memory 2 🔒 Motivating Ideas:

- Processes must be *isolated* from each other. This is a cornerstone of computer security.
- Processes should believe that they have the *entire* memory space available to them, without worrying about our limited available DRAM.
Virtual Memory 2 🕵️ Motivating Ideas:

- Processes must be *isolated* from each other. This is a cornerstone of computer security.
- Processes should believe that they have the *entire* memory space available to them, without worrying about our limited available DRAM.
Paging Out To Disk

- **Demand Paging**
  - Pages of physical memory are mapped to processes on demand (during execution)
  - If our physical memory fills up, we can copy some pages to disk and reuse those physical pages!

<table>
<thead>
<tr>
<th>Physical page #</th>
<th>WR?</th>
<th>PR?</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>
Paging Out To Disk

- **Demand Paging**
  - Pages of physical memory are mapped to processes *on demand* (during execution)
  - If our physical memory fills up, we can copy some pages to disk and reuse those physical pages!

<table>
<thead>
<tr>
<th>Physical page #</th>
<th>WR?</th>
<th>PR?</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

Let’s assume at this point, our DRAM is completely full! All space for pages has been allocated!
Paging Out To Disk

- **Demand Paging**
  - Pages of physical memory are mapped to processes *on demand* (during execution)
  - If our physical memory fills up, we can copy some pages to disk and reuse those physical pages!

<table>
<thead>
<tr>
<th>Physical page #</th>
<th>WR?</th>
<th>PR?</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

Now, let’s assume I page fault on a *virtual address* that maps to *index 5* in my page map.
Paging Out To Disk

- **Demand Paging**
  - Pages of physical memory are mapped to processes *on demand* (during execution)
  - If our physical memory fills up, we can copy some pages to disk and reuse those physical pages!

Now, let's assume I page fault on a *virtual address* that maps to **index 5** in my page map.
Paging Out To Disk

- **Demand Paging**
  - Pages of physical memory are mapped to processes *on demand* (during execution)
  - If our physical memory fills up, we can copy some pages to disk and reuse those physical pages!

<table>
<thead>
<tr>
<th>Physical page #</th>
<th>WR?</th>
<th>PR?</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

Why can’t I just make the mapping and continue as normal?
Paging Out To Disk

- **Demand Paging**
  - Pages of physical memory are mapped to processes *on demand* (during execution)
  - If our physical memory fills up, we can copy some pages to disk and reuse those physical pages!

```
<table>
<thead>
<tr>
<th>Physical page #</th>
<th>WR?</th>
<th>PR?</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>
```

DRAM is full! Even though our page map has available entries, it’s likely that other processes are using lots of memory.
Paging Out To Disk

- **Demand Paging**
  - Pages of physical memory are mapped to processes *on demand* (during execution)
  - If our physical memory fills up, we can copy some pages to disk and reuse those physical pages!

<table>
<thead>
<tr>
<th>Physical page #</th>
<th>WR?</th>
<th>PR?</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

At this point, we need to pick a page in DRAM and write it out to disk.
Paging Out To Disk

- **Demand Paging**
  - Pages of physical memory are mapped to processes *on demand* (during execution)
  - If our physical memory fills up, we can copy some pages to disk and reuse those physical pages!

At this point, we need to pick a page in DRAM and write it out to disk.

This is called “paging” or “swapping” to disk.
Paging Out To Disk

- **Demand Paging**
  - Pages of physical memory are mapped to processes *on demand* (during execution)
  - If our physical memory fills up, we can copy some pages to disk and reuse those physical pages!

Now, notice that the selected page has a Present bit of 0, implying its data is no longer stored in DRAM.

We also clear out the physical page # field to avoid confusion.

---

Page swapping algorithm

<table>
<thead>
<tr>
<th>Physical page #</th>
<th>WR?</th>
<th>PR?</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>
Paging Out To Disk

- **Demand Paging**
  - Pages of physical memory are mapped to processes *on demand* (during execution)
  - If our physical memory fills up, we can copy some pages to disk and reuse those physical pages!

<table>
<thead>
<tr>
<th>Physical page #</th>
<th>WR?</th>
<th>PR?</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>7</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>
Clock Algorithm

- While there are many page replacement techniques, the one you’ll use in CS111 is called the Clock Algorithm.
Clock Algorithm

- While there are many page replacement techniques, the one you’ll use in CS111 is called the Clock Algorithm.
  - It’s an example of an LRU algorithm (least-recently-used)
  - Key idea -> swap out the page that was touched last by any program.
Clock Algorithm

- While there are many page replacement techniques, the one you’ll use in CS111 is called the **Clock Algorithm**.
  - It’s an example of an **LRU algorithm (least-recently-used)**
  - Key idea -> swap out the page that was touched last by any program.
    - It’s not easy to do this precisely without taking up lots of space (i.e. an exact timestamp)
Clock Algorithm

- Where have we seen LRU used before this quarter?

4.4 BSD scheduler
Clock Algorithm

- Where have we seen LRU used before this quarter?

- Why do we care so much about our page replacement Algorithm? What’s the worst that could happen?

Bad page replacement leads to constant disk read/write for each memory accesses => Thrashing
Clock Algorithm

We add a “reference” bit to our page map entry.

When a page is referenced in the page map, its reference bit is set to 1.
Clock Algorithm

We add a “reference” bit to our page map entry

When a page is referenced in the page map, its reference bit is set to 1

Every time the algorithm runs, it cycles through the page map (from the entry it last swapped), and sets all reference bits that are 1 to 0
Clock Algorithm

We add a “reference” bit to our page map entry

When a page is referenced in the page map, its reference bit is set to 1

Every time the algorithm runs, it cycles through the page map (from the entry it last swapped), and sets all reference bits that are 1 to 0

As soon as the algorithm finds a page with Ref bit 0, it stops and swaps that page out
Clock Algorithm

Between runs of the Clock Algorithm, some pages will be referenced, and others won’t be!

```
<table>
<thead>
<tr>
<th>Physical page #</th>
<th>WR?</th>
<th>PR?</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>E</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>D</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>X</td>
<td>X</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>A</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```
Clock Algorithm

Between runs of the Clock Algorithm, some pages will be referenced, and others won’t be!

Pages that aren’t referenced between runs will have their reference bit **still be 0**. Hence, we can swap them out.
Clock Algorithm

Let’s see an example run of the Clock Algorithm

Assume all pages start out with a reference bit of 0

Every time a page is accessed, set its reference bit to 1
Clock Algorithm

Let’s see an example run of the Clock Algorithm

Assume all pages start out with a reference bit of 0

Every time a page is accessed, set its reference bit to 1

* Program runs, time passes *
Clock Algorithm

Let’s see an example run of the Clock Algorithm

Assume all pages start out with a reference bit of 0

Every time a page is accessed, set its reference bit to 1

* Program runs, time passes *

<table>
<thead>
<tr>
<th>Physical page #</th>
<th>WR?</th>
<th>PR?</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>E</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>D</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>J</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>ZZ</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>F</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>C</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>B</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>A</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Clock Algorithm

Let’s see an example run of the Clock Algorithm

Assume all pages start out with a reference bit of 0

Every time a page is accessed, set its reference bit to 1

* Program runs, time passes *

Now the Clock Algorithm gets invoked.

Self Check: What would cause the Clock Algorithm to be called?
Clock Algorithm

Let’s see an example run of the Clock Algorithm

Assume all pages start out with a reference bit of 0

Every time a page is accessed, set its reference bit to 1

* Program runs, time passes *

**Now** the Clock Algorithm gets invoked. Self Check: What would cause the Clock Algorithm to be called?
Clock Algorithm

Let’s see an example run of the Clock Algorithm

Assume all pages start out with a reference bit of 0

Every time a page is accessed, set its reference bit to 1

* Program runs, time passes *

Now the Clock Algorithm gets invoked. Self Check: What would cause the Clock Algorithm to be called?
Clock Algorithm

Let’s see an example run of the Clock Algorithm

Assume all pages start out with a reference bit of 0

Every time a page is accessed, set its reference bit to 1

* Program runs, time passes *

Now the Clock Algorithm gets invoked.

Self Check: What would cause the Clock Algorithm to be called?
Clock Algorithm

Let’s see an example run of the Clock Algorithm

Assume all pages start out with a reference bit of 0

Every time a page is accessed, set its reference bit to 1

* Program runs, time passes *

Now the Clock Algorithm gets invoked.

Self Check: What would cause the Clock Algorithm to be called?
Clock Algorithm

Let’s see an example run of the Clock Algorithm

Assume all pages start out with a reference bit of 0

Every time a page is accessed, set its reference bit to 1

* Program runs, time passes *

Now the Clock Algorithm gets invoked. Self Check: What would cause the Clock Algorithm to be called?
Clock Algorithm

Let’s see an example run of the Clock Algorithm

Assume all pages start out with a reference bit of 0

Every time a page is accessed, set its reference bit to 1

* Program runs, time passes *

Now the Clock Algorithm gets invoked.
Self Check: What would cause the Clock Algorithm to be called?
Clock Algorithm

Let's see an example run of the Clock Algorithm

Self Check #2

Assuming no other pages are referenced, which page # will be removed next?
Clock Algorithm

Another question:

There is a special case for the Clock Algorithm – if the page it selects to swap out has its “writeable” bit set to 0 (meaning data is read-only) what happens? Is this case good or bad?

Good! It’s identical to the copy on disk.
Clock Algorithm

One last question:

There are two ways to run the Clock Algorithm. One is over a page table of only the Page Map Entries from your own process (called local replacement), and the other approach scans over all Page Map Entries to pick a swap candidate (called global replacement).

Give 1 pro and 1 con of both approaches.
Alternatives to Page Replacement

Page Replacement is expensive, mostly because we need to write to disk.

While the Clock Algorithm runs, we’re not utilizing one of our CPU cores (even though we context switch once the Disk write starts)

Luckily, Operating Systems have a few approaches to make this cost a little better…
Page Fetching (Prefetching)

When a Page Fault occurs, the OS may elect to read in more than one page into memory.

Which pages would it bring in? Why would it do this?

Bring in many pages in the hope that we'll use them all. "Locality" if I need memory, I might use memory around this spot very soon.
Page Fetching (Prefetching)

When a Page Fault occurs, the OS may elect to read in *more* than one page into memory.

Which pages would it bring in? Why would it do this?

What is a downside of prefetching?

Could have internal fragmentation
Game Plan

- Demand Paging
- Admin
- Magnetic Disk
Administrativa

- A5 is due this Thursday, 5/18! I’ll be having office hours that morning, 9-11AM in Old Union (2nd floor)
Computer Scientists *LOVE* being confusing with jargon. It’s just one of the things we need to deal with until we fix the culture 🙁

Let’s end this confusion once and for all with a single (animated) slide!

If you’ve ever heard of DRAM, this is the *hardware* that supports program memory. Stands for (*Dynamic Random Access Memory*). This really just means “you can select arbitrary bytes from this memory in any order without seeing a hit in performance.”
Disk

- Computer Scientists LOVE being confusing with jargon. It’s just one of the things we need to deal with until we fix the culture.

- Let’s end this confusion once and for all with a single (animated) slide!

  This stack-heap thing that we learned about in CS107 is called "Main Memory," "Program Memory," or simply "Memory." If you’ve ever heard of DRAM, this is the hardware that supports program memory. DrAM stands for (Dynamic Random Access Memory). This really just means "you can select arbitrary bytes in any order without seeing a hit in performance."

  Memory is what we call “volatile,” which means (in this case) that when you cut the power, whatever stuff was being stored in the memory goes away!
Disk

- Computer Scientists LOVE being confusing with jargon. It’s just one of the things we need to deal with until we fix the culture 😔
- Let’s end this confusion once and for all with a single (animated) slide!

Storage hardware like disks are slower than memory hardware like DRAM. The tradeoff is that they have much more capacity, and they’re cheaper.

The granularity at which hardware can access the (storage) disk is at the sector granularity. Although disk support random access, it’s significantly slower.

*Granularity is a great word that systems engineers like to use. It means “the standard size,” and usually refers to the smallest size at which hardware can do something.
Disk

This is a Disk! It consists of a series of circular platters.
Disk

This is a Disk! It consists of a series of circular platters.

When a read/write request is made, the disk mechanism must choose the correct arm/head corresponding to the platter that contains the data (more about how we know which platter is the right one in a few weeks!). We call the smallest unit a disk can store a sector.

The arm of the disk must move to the correct ring and wait for the data to rotate under the head.
Disk

The **key takeaway** here is that to read/write from disk, something *mechanical* has to happen. (i.e. the head has to physically move)

In RAM, memory is read/written digitally (via electricity!).

How slow *is* a disk I/O?
Disk

The **key takeaway** here is that to read/write from disk, something *mechanical* has to happen. (i.e. the head has to physically move)

In RAM, memory is read/written digitally (via electricity!).

How slow *is* a disk I/O?
Talking to Disk

How does the OS actually communicate with I/O devices like Disk?
Talking to Disk

How does the OS actually communicate with I/O devices like Disk?

In physical memory (i.e. not directly accessible to user programs, but visible to the OS) are a few 4-byte items called “device registers”

You can think of device registers as a bare-bones API for a computer to talk to the hardware device.
Talking to Disk

How does the OS actually communicate with I/O devices like Disk?

In physical memory (i.e. not directly accessible to user programs, but visible to the OS) are a few 4-byte items called "device registers"

You can think of device registers as a bare-bones API for a computer to talk to the hardware device.

Specifically, disk offers 3 sets device registers:

- Parameters for the disk (i.e. the sector number to start reading from)
- Control Bits (set by OS) to get the disk going (i.e. start read)
- Status Bits (set by Disk) to indicate when an I/O has completed.
Talking to Disk

Because we cannot busy-wait for a disk I/O, we need some way for the CPU to know when a read or write is done! What should we do?