Memory and Data Locality

(The C Memory Model)

https://cs.pomona.edu/classes/cs140/
Outline

Topics and Learning Objectives

• Motivate our discussion on memory
• Discuss memory access timing
• Discuss the C memory model
• Discuss locality

Exercise

• Sets and Lists (tangentially related)
Exercise
Process Memory

**Text:** contains your compiled code

**Data:** contains *initialized* static and global variables

**BSS:** contains uninitialized static and global variables

**Heap:** contains dynamically allocated memory

**Stack:** contains local variables
Process Memory

Text: contains your compiled code

Data: contains initialized static and global variables

BSS: contains uninitialized static and global variables

Heap: contains dynamically allocated memory

Stack: contains local variables
char* string = “hello”;
int iSize;

char* f(void)
{
    char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}

Read through the code for a few moments.
char* string = "hello";
int iSize;

char* f(void)
{
    char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
char* string = "hello";
int iSize;

char* f(void)
{
    char* p;
iSize = 8;
p = malloc(iSize);
return p;
}
char* string = “hello”;
int isize;

char* f(void)
{
    char* p;
    isize = 8;
    p = malloc(isize);
    return p;
}
What goes in Data?

```c
char* string = "hello";
int iSize;

char* f(void)
{
    char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```
char* string = "hello";
int iSize;

char* f(void)
{
    char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}

What goes in BSS?
What goes in BSS?

```c
char* string = "hello";
int isize;

char* f(void)
{
    char* p;
    isize = 8;
    p = malloc(isize);
    return p;
}
```
char* string = “hello”;
int iSize;

char* f(void)
{
    char* p;
iSize = 8;
p = malloc(iSize);
return p;
}
char* string = "hello";
int iSize;

char* f(void)
{
    char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
char* string = "hello";
int iSize;

char* f(void)
{
    char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
char* string = "hello";
int iSize;

char* f(void)
{
    char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
char* string = "hello";
int isize;

char* f(void)
{
    char* p;
    isize = 8;
    p = malloc(isize);
    return p;
}
Stack and Heap Resources

- https://www.cs.princeton.edu/courses/archive/fall07/cos217/lecture\ns/06MemoryAllocation-3x1.pdf
- https://www.usna.edu/Users/cs/aviv/classes/ic221/s16/lec/08/lec.html
Principle of Locality

- **Locality**: programs tend to reuse data and instructions near those they have used recently

- **Temporal locality**: recently referenced items are likely to be referenced again in the near future

- **Spatial locality**: items with nearby addresses tend to be referenced close together in time
Discontiguous data structures are the root of all evil. This is simply a fact. If you don't believe me, I'll try to convince you. Specifically, please say "no" to linked lists. OK. Please. Please say no to linked lists. There is almost nothing more harmful you can do to the performance of an actual modern microprocessor than to use a linked list data structure.

Chandler Carruth (Engineer at Google) @ CppCon 2014
Discontiguous data structures are the root of all evil. This is simply a fact. If you don't believe me, I'll try to convince you. Specifically, please say "no" to linked lists. OK. Please. Please say no to linked lists. There is almost nothing more harmful you can do to the performance of an actual modern microprocessor than to use a linked list data structure.
Notice that the amount of space gets bigger as you go down the hierarchy.
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1 Cache
4. On miss, looks in L2 Cache
5. On miss, looks in L3 Cache
6. On miss, finds it in memory (in “virtual” memory)
<table>
<thead>
<tr>
<th>Entity</th>
<th>Time (nanoseconds)</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>One cycle on a 3 GHz processor</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>L1 cache reference</td>
<td>0.5</td>
<td></td>
</tr>
<tr>
<td>Branch mis-predict</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7</td>
<td>14x L1 cache</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100</td>
<td>20x L2; 200x L1</td>
</tr>
<tr>
<td>Compress 1k bytes with Snappy</td>
<td>3,000</td>
<td></td>
</tr>
<tr>
<td>Send 1K bytes over 1 Gbps network</td>
<td>10,000</td>
<td></td>
</tr>
<tr>
<td>Read 4K randomly from SSD</td>
<td>150,000</td>
<td></td>
</tr>
<tr>
<td>Read 1 MB sequentially from main memory</td>
<td>250,000</td>
<td></td>
</tr>
<tr>
<td>Round trip with the same datacenter</td>
<td>500,000</td>
<td></td>
</tr>
<tr>
<td>Read 1 MB sequentially from SSD</td>
<td>1,000,000</td>
<td>4x main memory</td>
</tr>
<tr>
<td>Disk seek</td>
<td>10,000,000</td>
<td>20x datacenter round trip</td>
</tr>
<tr>
<td>Read 1 MB sequentially from Disk</td>
<td>20,000,000</td>
<td>80x main memory; 20x SSD</td>
</tr>
<tr>
<td>Send package CA -&gt; Netherlands -&gt; CA</td>
<td>150,000,000</td>
<td></td>
</tr>
</tbody>
</table>
Intel Core i7 cache hierarchy (2014)

Memory Layout:
- **Text**
- **Data**
- **BSS**
- **Heap**
- **Stack**

Processor package:
- **Core 0**
  - Regs
  - L1 d-cache
  - L1 i-cache
  - L2 unified cache
  - L3 unified cache (shared by all cores)
- **Core 3**
  - Regs
  - L1 d-cache
  - L1 i-cache
  - L2 unified cache

- 16 (rax, rbx...)
- 32kB each
- 256kB
- 2-20MB
- L4* 128 MB (Iris Pro)

Main memory
Intel Haswell

Latency (cycles)
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On **miss**, looks in L2
5. On **miss**, looks in L3
6. On **miss**, finds it in memory
   (in “virtual” memory)
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On miss, looks in L2
5. On miss, looks in L3
6. On miss, finds it in memory
   (in “virtual” memory)
1. CPU needs a piece of memory

2. Needs to load it into a register

3. Looks in L1

4. On miss, looks in L2

5. On miss, looks in L3

6. On miss, finds it in memory

(in “virtual” memory)
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On miss, looks in L2
5. On miss, looks in L3
6. On miss, finds it in memory
   (in “virtual” memory)
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On miss, looks in L2
5. On miss, looks in L3
6. On miss, finds it in memory (in “virtual” memory)
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On miss, looks in L2
5. On miss, looks in L3
6. On miss, finds it in memory

(in “virtual” memory)
1. CPU needs a piece of memory

2. Needs to load it into a register

3. Looks in L1

4. On miss, looks in L2

5. On miss, looks in L3

6. On miss, finds it in memory (in “virtual” memory)
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On miss, looks in L2
5. On miss, looks in L3
6. On miss, finds it in memory (in “virtual” memory)
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On miss, looks in L2
5. On miss, looks in L3
6. On miss, finds it in memory
   (in “virtual” memory)
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On miss, looks in L2
5. On miss, looks in L3
6. On miss, finds it in memory

(in “virtual” memory)
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On miss, looks in L2
5. On miss, looks in L3
6. On miss, finds it in memory
   (in “virtual” memory)
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On miss, looks in L2
5. On miss, looks in L3
6. On miss, finds it in memory (in “virtual” memory)
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On miss, looks in L2
5. On miss, looks in L3
6. On miss, finds it in memory (in “virtual” memory)
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On miss, looks in L2
5. On miss, looks in L3
6. On miss, finds it in memory (in “virtual” memory)
1. CPU needs a piece of memory
2. Needs to load it into a register
3. Looks in L1
4. On **miss**, looks in L2
5. On **miss**, looks in L3
6. On **miss**, finds it in memory
   (in “virtual” memory)
1. CPU needs a piece of memory

2. Needs to load it into a register

3. Looks in L1

4. On miss, looks in L2

5. On miss, looks in L3

6. On miss, finds it in memory

(in “virtual” memory)
std::vector vs std::list

Which has better locality? Meaning, which will work better with cache?
What does this picture look like for:
1. An Adjacency Matrix
2. An Adjacency List
What does extract min look like?

What does this picture look like for:
1. A Sorted Array
2. A Binary Search Tree
3. A Heap