CSE 451: Operating Systems

Lab Section: Week 6
Today

• Project 3

• Virtual Address Spaces
  - Part II: fun virtual memory tricks

• Paging

(I have no idea what’s on tomorrow’s quiz 😞)
Project 3

• Due Wednesday, Feb 16 at 11:59pm
  - next week!

• Questions?
A rant on code optimization

“premature optimization is the root of all evil”
- Donald Knuth

• Write the simple version first
  - the profile (this tells you where time is spent)
  - then optimize

• You don’t need to super optimize every line of code!
"Numbers every engineer should know"
(from Jeff Dean, Google)

| Simple instruction | $<1$ ns |
| L1 cache reference | $<1$ ns |
| Main memory reference | $100$ ns |
| Mutex lock/unlock | $100$ ns |
| Compress 1Kb of data | $10,000$ ns |
| Send 2Kb over local network | $20,000$ ns |
| Read 1Mb sequentially from flash drive | $5,000,000$ ns |
| Read 1Mb sequentially from network | $10,000,000$ ns |
| Disk seek (random access) | $10,000,000$ ns |
| Read 1 Mb sequentially from disk | $30,000,000$ ns |
| Send packet CA→Netherlands→CA | $250,000,000$ ns |

Flash numbers added by me
Hard disk geometry

(what is a seek?)

seek: moving the arm

from a future lecture slide
Today

• **Project 3**

• **Virtual Address Spaces**
  - Part II: fun virtual memory tricks

• **Paging**
Virtual Address Spaces
(review)

\[ P_1 \text{ address space} \]

\[
\begin{array}{c}
\text{user space} \\
\hline
0 \quad 2^{64} - 1
\end{array}
\]

physical memory

\[ P_2 \text{ address space} \]

\[
\begin{array}{c}
\text{user space} \\
\hline
0 \quad 2^{64} - 1
\end{array}
\]
Virtual Address Spaces
(review)

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
<th>Protect Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0041ab...</td>
<td>✔</td>
<td>user</td>
</tr>
<tr>
<td>0xffffffff...</td>
<td>✘</td>
<td>user</td>
</tr>
</tbody>
</table>

P1 address space

user space

kernel space

2^64 - 1

P2 address space

user space

kernel space

2^64 - 1

physical memory
Page table protection bits

• **user** bit
  - we just saw this
  - used to hide kernel pages from user programs

• **present** bit
  - is there a physical page allocated for this virtual address?

• **writable** bit
  - is the page writable?
  - when unset, the page is read-only (we’ll see this in a bit)

• What if a protection bit is violated?
  - hardware triggers a page fault
  - OS decides what to do
VM trick (1): NULL pointers

P1 address space

- code
- data, stack
- kernel space

Goal: segfault on (*p) when (p=NULL)

How?
- use a null page!
- marked not present in the page table
VM trick (2): sharing

Does sharing make sense?

Pages are shared

$P_1$ address space

0 $\rightarrow$ \[ 2^{64}-1 \]$  

user space  

kernel space

$P_2$ address space

0 $\rightarrow$ \[ 2^{64}-1 \]$  

user space  

kernel space

physical memory
VM trick (2): sharing

A shared library:
- share code pages in multiple address spaces (saves space!)

Problem: can’t let P₂ write to P₁’s DLL!
- solution: map pages read-only
VM trick (2): sharing

P₁ address space

code  DLL  data, stack  kernel space

0  2^64-1

physical memory

page table

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
<th>Protect Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x0041ab...</td>
<td></td>
<td>✔ user  ❌ writable</td>
</tr>
</tbody>
</table>

pages mapped read-only

P₂ address space

code  DLL  data, stack  kernel space

0  2^64-1

page table

<table>
<thead>
<tr>
<th>Virtual Address</th>
<th>Physical Address</th>
<th>Protect Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x07eff...</td>
<td></td>
<td>✔ user  ❌ writable</td>
</tr>
</tbody>
</table>
VM trick (2): sharing

How do we know the address of memcpy?
- it depends on where the DLL was loaded!
- solution: jump table
VM trick (2): sharing

**P_1 address space**

- code
- data, stack
- kernel space

Jumper table initially empty

Library call indirects through *jump table*

```
jumpTable = {
    [0] = ?,
    [1] = ?,
    ...
    [42] = ?,
    ...
}
```
VM trick (2): sharing

Jump table fixed when DLL is loaded
- by a program called a loader
VM trick (3): fast system calls

Fast GetTime() syscall
- kernel writes current time to a special page
- mapped *writable* in kernel space
- mapped *read-only* in all processes

user reads time

kernel writes time
VM trick (4): fork

The UNIX fork() syscall:

```c
r = fork()  // spawns a new process
    // as a copy of this one
if (r > 0)  // in the parent (P1)
else if (r == 0)  // in the child (P2)
```

**P1 address space**

```
0 ----------- 2^{64}-1
```

- user space
- kernel space

**P2 address space**

```
0 ----------- 2^{64}-1
```

- user space *(copy of P1)*
- kernel space

**physical memory**
Efficient `fork()` via copy-on-write

- copy all page table mappings
- mark *read-only* in both processes
- lazy copy

**On P2 write:**
- page fault
- copy the page, mark *writable*
VM trick (4): fork
copy-on-write

On P₁ write:
- page fault
- only reference left!
  (P₂ has a private copy now)
- just mark writable
More VM tricks please!

See this excellent paper by Andrew Appel and Kai Li

“Virtual Memory Primitives for User Programs” (ASPLOS 1991)
- garbage collection, distributed shared memory, more ...

Check out Emery Berger’s work

Professor at UMASS
has made a career out of inventing VM tricks (among other things)
Today

- Project 3

- Virtual Address Spaces
  - Part II: fun virtual memory tricks

- Paging
Paging

• What if we need more pages than available in physical memory?
  - page to disk

• Isn’t this slow?
  - yes!
  - but processes have locality
Working Set

- $W(t, w)$
  - set of pages used in time $[t-w, t]$

- This is usually a small-ish subset of memory
  - demonstrated empirically

- Ideally: keep the working set in memory
  - page out everything else
  - see lecture slides for algorithms
Paging

- *local* page replacement

\[ P_1 \rightarrow \text{pages resident in memory} \]

\[ P_2 \rightarrow \text{pages swapped to disk} \]

On \( P_2 \) page fault: swap out a page from \( P_2 \)'s memory

*(fixed quota per process)*
Paging

• *global* page replacement

\[ P_1 \quad \text{pages resident in memory} \]

\[ P_2 \quad \text{on page fault: swap out *any* page to memory} \]

\[ \text{pages swapped to disk} \]
Paging

- **local** page replacement
  - fixed quota per process
  - why bad?
    - not globally optimal
    - e.g.: foreground tasks should get more pages

- **global** page replacement
  - no quotas
  - why bad?
    - more variability, possibility for unfairness
Working set

• When is the working set the entire program?
  - garbage collection! (mark-and-sweep...)
  - Java performance tanks when paging to disk