Reminders

• Quiz tomorrow (2/22)
• Project 4 is up
  – Due Wednesday, 3/13
  – Group project
Topics for Today

• Project 3 Recap
• Virtual Address Spaces
• Project 4
Project 3 Recap

- How was performance?
  - Async vs. Sync?
  - Sync # of threads?
  - Async # of calls?
  - Buffer size?
Project 3 Recap

• Calls to disk are all sequential access!
  – Seems like concurrency won’t help much…
Project 3 Recap

• Calls to disk are all sequential access!
  – Seems like concurrency won’t help much…
• Disk Caching!
  – Optimizations to keep pages in memory
Project 3 Recap

- Write caching

- Read caching
Project 3 Recap

- Disk scheduler can minimize amount of I/O between memory and disk
- Delay write to disk as long as possible
- Reads **must** be immediate
  - If a write occurs on a file, a read on the same file must fetch from disk
Storage Latency: How Far Away is the Data?

- 10^9: Tape/Optical Robot (2,000 Years)
- 10^6: Disk (2 Years)
- 100: Memory (1.5 hr)
- 10: On Board Cache (10 min)
- 2: On Chip Cache
- 1: Registers (1 min)

© 2004 Jim Gray, Microsoft Corporation
Topics for Today

• Project 3 Recap
• Virtual Address Spaces
• Project 4
Virtual Address Spaces

- Wait, if pointers are just numbers ...
  - how does each process get a private memory space?
  - how does the kernel get a private memory space?
  - how does the kernel access process memory?
Virtual Address Spaces

here is a pointer

process address space

physical memory

p: 0x0041ab8fe023ecd5
Virtual Address Spaces

here is a pointer
p: 0x0041ab8fe023ecd5

process address space

page table

physical memory
Virtual Address Spaces

On context switch:
- install page table for the new process in hw
  (on x86: write pointer to %cr3 register)
Virtual Address Spaces

- Great, that explains how processes are isolated
- What about the kernel?
  - how does the kernel get a private memory space?
  - how does the kernel access user memory?

```c
NtReadFile(void* userbuf) {
    ...
    CSE451.readcalls++;
    ...
}
```
Virtual Address Spaces

P₁ address space

user space

kernel space

P₂ address space

user space

kernel space

physical memory
Virtual Address Spaces

Kernel and user share the address space:
- don’t need to install a new page table when entering the kernel
- this is how system calls access user space

```c
NtReadFile(void* userbuf) {
    ...
    CSE451.readcalls++;
    ...
}
```

system call
Virtual Address Spaces

- How is the kernel isolated from the user?
How is the kernel isolated from the user?
- Set protection bits in the page table:
  - hw triggers a page fault if user code tries to access kernel memory
Virtual Address Spaces

- So user and kernel share the address space. Great! What could possibly go wrong?

```c
NtReadFile(void* userbuf, int userlen)
{
    ... 
    memcpy( userbuf, FileData, FileDataSize );
}
```

- What if `userbuf` is invalid?
  - e.g. NULL or points at an unmapped page

- The kernel will segfault!
Virtual Address Spaces

- So user and kernel share the address space. Great! What could possibly go wrong?

```c
NtReadFile(void* userbuf, int userlen)
{
    ...
    memcpy(userbuf, FileData, FileDataSize);
}
```

- What if `userbuf` points into kernel space?
  - e.g. malicious user code “guesses” a pointer value
- The kernel data structures will be corrupted!
Always validate user pointers in the kernel

- Check that user pointers point at \textbf{user} memory, not kernel memory

  - user space
  - user buffer
  - kernel space

  
  Windows

  \texttt{ProbeForWrite( userbuf, length );}

  Linux

  \texttt{access\_ok( userbuf, length );}

- Guard kernel code that accesses user pointers against segfaults

  Windows

  \begin{verbatim}
  try {
    memcpy( userbuf, FileData, FileDataLen );
  } except {
    x = GetExceptionCode();
    ... // oops, handle segfault
  }
  \end{verbatim}

  Linux

  \begin{verbatim}
  copy\_to\_user( userbuf, FileData, FileDataLen );
  // copy\_to\_user deals with a segfault if it happens
  \end{verbatim}
Always validate user pointers in the kernel

- An example from Project 2:

```c
NtQuerySystemInformation( Cse451* info, ... )
{
    ....
    // copy event buffer to user space
    memcpy( info->buffer, CseEventBuffer, info->bufferSize );
    ....
}
```
Always validate user pointers in the kernel

- An example from Project 2.
  Added a fix. Is this enough? What could go wrong?

```c
NtQuerySystemInformation( Cse451* info, ... )
{
    ....
    ProbeForWrite( info->buffer, info->bufferSize );
    try {
        memcpy( info->buffer, CseEventBuffer, info->bufferSize );
    } except {
        ....
    }
}
Always validate user pointers in the kernel

- An example from Project 2.
  Added a fix. Is this enough? What could go wrong?

What if another thread changes info->buffer after ProbeForWrite and before memcpy?

```c
NtQuerySystemInformation( Cse451* info, ... )
{
    ....
    ProbeForWrite( info->buffer, info->bufferSize );
    try {
        memcpy( info->buffer, CseEventBuffer, info->bufferSize );
    } except {
        ....
    }
}
```

**Buggy user code example**

- Thread 1: `NtQuerySystemInformation(info);`
- Thread 2: `info->buffer = 0xfff...;` (a kernel address)
Always validate user pointers in the kernel

An example from Project 2.
The full fix:

```c
NtQuerySystemInformation( Cse451* info, ... )
{
    ....
    tmpBuffer = info->buffer;  // capture pointer
    tmpSize   = info->bufferSize;
    ....
    ProbeForWrite( tmpBuffer, tmpSize );
    try {
        memcpy( tmpBuffer, CseEventBuffer, tmpSize );
    } except {
        ....
    }
}```
Topics for Today

• Project 3 Recap
• Virtual Address Spaces
• Project 4
Goals: Modify the FAT file system to
- Make all directories sortable
- Compact directories
The FAT File System

Goal of FAT: store files and directories!

Size of FAT
Size of data area
Size of each cluster
Location of root directory
The FAT File System

Goal of FAT: store files and directories!

Each cluster either:
- Stores data for a file or...
- Stores lists of files in a directory (dirent)

Size of FAT
Size of data area
Size of each cluster
Location of root dirent
The FAT File System

![Diagram of FAT File System]

Goal of FAT: store files and directories!

Each cluster either:
- Stores data for a file
- Stores lists of files in a directory (dirent)

File Allocation Table
- Linked list of clusters
- As many entries as there are clusters

Size of FAT
Size of data area
Size of each cluster
Location of root dirent
The FAT File System

• So, how do we get files?
The FAT File System

[Diagram showing FAT and data area with pointers and cluster information for files named "file1.txt", "file2.txt", "subdir", "x0.txt", "x1.txt", "x2.txt", "y1.txt", "y2.txt", "y3.txt" and their corresponding clusters and pointers.]
Project 4

- Goal: keep dirents sorted in each directory
  - Note: This means implementing your own sorting algorithm!

PACKED_DIRENT (from fat.h)

<table>
<thead>
<tr>
<th>Field</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>FileName</td>
<td>&quot;file1.txt&quot;</td>
</tr>
<tr>
<td>LastWriteTime</td>
<td>...</td>
</tr>
<tr>
<td>FirstClusterOfFile</td>
<td>1</td>
</tr>
<tr>
<td>FileSize</td>
<td>4052</td>
</tr>
</tbody>
</table>
Project 4

- Kernel data structures: on-disk (fat.h)
  - PACKED_BOOT_SECTOR (boot info, etc – don’t modify)
  - BIOS_PARAMETER_BLOCK (boot info, etc – don’t modify)
  - PACKED_DIRENT(DIRENT struct)
- Kernel data structures: in-memory (fatstruct.h)
  - VCB (info about mounted volume)
  - FCB (cached files)
  - DCB (cached directories)
Project 4

VCB → Root DCB → DCB → DCB → FCB (opened file)

FCB (opened file)
Project 4

• Resort dirent when:
  – Creating a new file (name, extension, cluster number)
  – Closing a file (timestamp, size)
  – Re-sorting the entire dirent
Topics for Today

- Project 3 Recap
- Virtual Address Spaces
- Project 4