Today

• Project 1

• Project 2

• User space vs. kernel space
  (this may be useful for tomorrow’s quiz ☺)
Office Hours

• Gary Kimura
  - MWF 12:00-1:00 CSE 476
  - (after class)

• Mark Zbikowski
  - MWF 9:30-10:30 CSE 591
  - (before class)
Project 1

• **Due January 26, 11:59 pm**
  - same time as Project 2
  - you can resubmit until then

• **Asynchronous I/O extra credit**
  - this is actually a lot of work
  - if you want to do it:
    ... read Chapter 9 of *Windows Internals*
    ... especially “I/O Processing”

• **Questions?**
This is broken when run on a multiprocessor. Why?
NTSTATUS
NTReadFile(...) {
    ...
    int tmp = CSE451Info.readcalls;
    CSE451Info.readcalls = tmp+1;
    return status;
}
NTSTATUS
NTReadFile(...) {

    int tmp = CSE451Info.readcalls;
    CSE451Info.readcalls = tmp+1;
    return status;
}

Thread 1
NTReadFile("foo.txt")

Thread 2
NTReadFile("bar.c")

tmp 5
CSE451Info.readcalls 5

int tmp = CSE451Info.readcalls;
CSE451Info.readcalls = tmp+1;
return status;

int tmp = CSE451Info.readcalls;
CSE451Info.readcalls = tmp+1;
return status;

tmp 5
CSE451Info.readcalls 6

✘
Visualizing the bug as a bad interleaving
How can we fix this data race?
(you need to do this for Project 2)

• Use a \textit{mutex}
  - short for “MUtual EXclusion”
  - \texttt{Acquire(mutex)} begins a critical section
  - \texttt{Release(mutex)} ends a critical section

• Sometimes called a \textit{lock}
  - \texttt{Lock(lock)} same as \texttt{Acquire(mutex)}
  - \texttt{Unlock(lock)} same as \texttt{Release(mutex)}

• Let’s see how it works …
How can we fix this data race?
(you need to do this for Project 2)

Thread 1
{
    ...
    Acquire(&SomeMutex);
    int tmp = CSE451Info.readcalls;
    CSE451Info.readcalls = tmp+1;
    Release(&SomeMutex);
    ...
}

Thread 2
{
    ...
    Acquire(&SomeMutex);
    int tmp = CSE451Info.readcalls;
    CSE451Info.readcalls = tmp+1;
    Release(&SomeMutex);
    ...
}

Cannot complete until Thread 1 releases SomeMutex
Project 2

• Two goals
  - make Project 1 thread safe
  - support event histories (see project doc)

• Due January 26, 11:59 pm
  - available on course website
Today

• Project 1

• Project 2

• User space vs. kernel space
  (this may be useful for tomorrow’s quiz 😊)
User space vs. Kernel space

User Space
- process

Kernel Space
- Isolate processes
- Multiplex hardware safely

Hardware
- cpus
- hard disks
- memory
- devices
What’s in a process?

**Process**
- stack (per thread)
- data
- code
- threads (at least one)
- private memory (address space)
- open files
- net connections
- etc.

**Thread**
- stack ptr
- instruction ptr
- floating point
- etc.
- registers

**Resources**
- etc.
Life of an Operating System

1. **Boot (initialize)**
2. **Pick a thread to run**
3. **Jump to user space: run that thread**
4. **Jump to kernel space: handle a service**
What causes a transition to kernel mode?

• Interrupts
  - caused by hardware ("asynchronous")
  - e.g.: timer fires, network packet arrives, etc.

• Exceptions
  - caused by software
  - traps: expected or intentional exceptions
    e.g., making a system call
  - faults: unexpected or error exceptions
    e.g., segfault (p=NULL; *p)
    divide by zero
    try to execute a privileged instruction in user mode
How do we pass data to/from a system call?

```
#include "a.h"
int main() {
    ...
    kernbuf
    {
        ...
        return &kernbuf;
    }
}
```
How do we pass data to/from a system call?

User Space

buf = ReadFile(...);
print(buf);

Kernel Space

#include “a.h”
int main() {
...
return &kernbuf;
}

BUG: user can’t access kernel memory!
How do we pass data to/from a system call?

OK: kernel can access user memory
How do we pass data to/from a system call?

Evil user program:

```c
#include "a.h"

int main() {
    ReadFile((char*)0xfff23456, ...);
    // manufacture a buffer ptr
    // hope we get lucky and it points at
    // a kernel data structure!
}
```

```c
ReadFile(char* userbuf, int userlen) {
    ...
    memcpy(userbuf, kernbuf, sz);
    return;
}
```

```
user = "Tom"
passwd = "12345"
```

Kernel must validate user buffers!

Kernel memory:

```
#include "a.h"

int main() {
    ....
}
```
Evil user program:

```c
#include "a.h"
int main() {
    ....
    kernbuf = ReadFile((char*)0xfff23456, ...);
    // manufacture a buffer ptr
    // hope we get lucky and it points at
    // a kernel data structure!
    ReadFile(char* userbuf, int userlen) {
        ...
        ProbeForWrite(userbuf, userlen, ..);
        // fails if userbuf is not valid
        // memory in user space
        ...
        memcpy(userbuf, kernbuf, sz);
        ...
    }
    ...
    user = "Tom"
    passwd = "12345"
    #include "a.h"
    int main() {
        ....
    }
    kernel memory
}```