CSE 451 Winter 2013

Section 2

Anton Osobov
aosobov@cs

Some material adapted from CSE 451, Winter 2011 and
Operating Systems Principles and Practices by Anderson and Dahlin

Reminders

- Quiz tomorrow (1/18)
- Project 2 is up
  - Due Wednesday, 1/30
  - Individual project
- Projects 3 and 4 will be done in groups
  - Groups were due to us yesterday, will be finalized by next week

Topics for Today

- Project 1
- System call parameter validation
- File handles

Project 1

- Questions/Comments?
- grepWin

Project 1

- Handling multiple returns from a function
  - Example: `NtReadFile`
  - Use a wrapper function

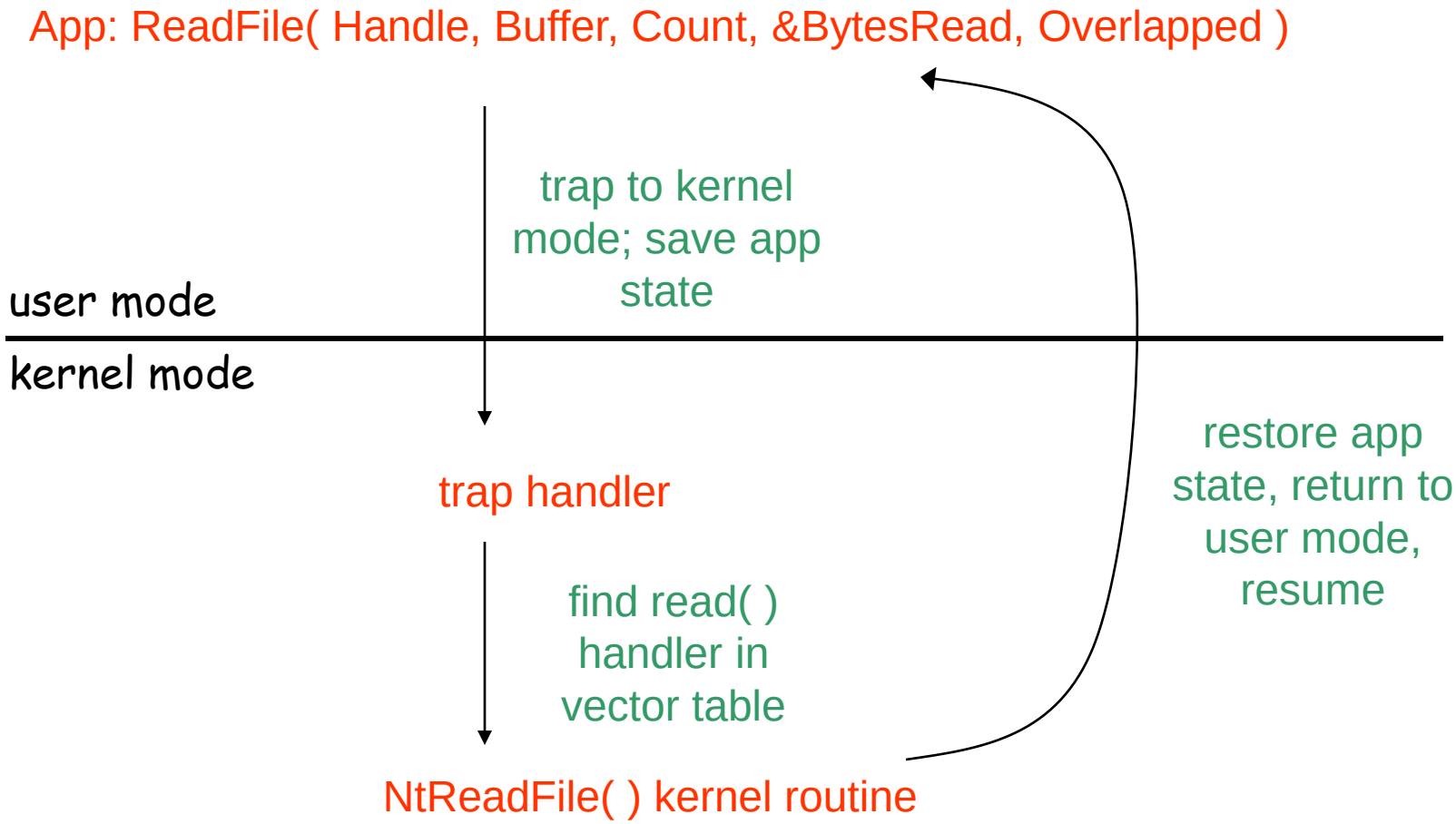
System Calls

- Provide user space applications with controlled access to OS services
- Necessary to protect the system from buggy or malicious code
- Requires special hardware support on CPU to detect a system call instruction and trap to the kernel

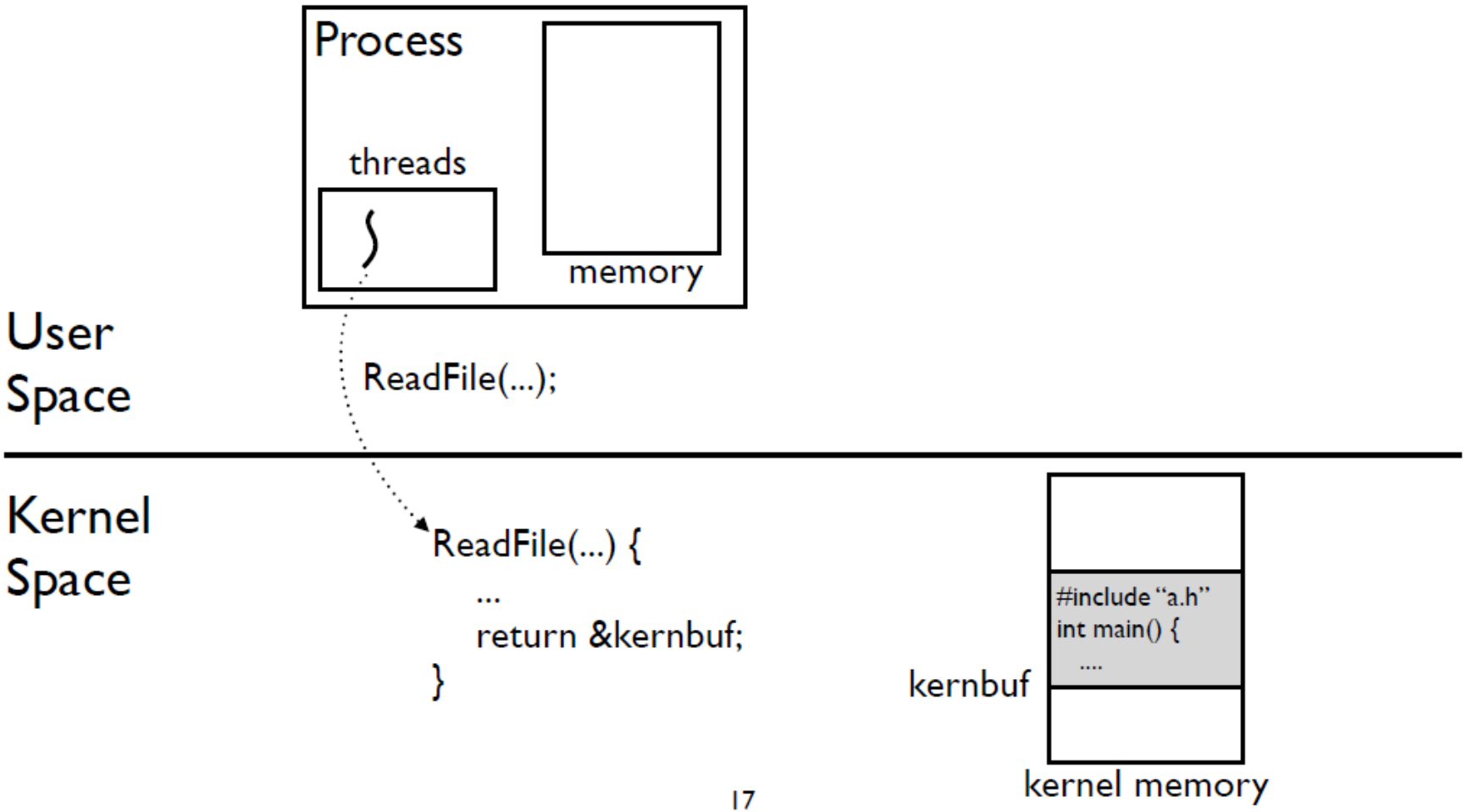
System Call Control Flow

- User application calls a user-level library routine (`NtQuerySystemInformation(), ReadFile(), etc.`)
- This routine is a stub
  - It calls the trap instruction and passes the number of the desired sys call
  - control is passed to the kernel
- The system call handler calls the appropriate function in the kernel
- The function executes and returns to interrupt handler, which return the result to the user space process

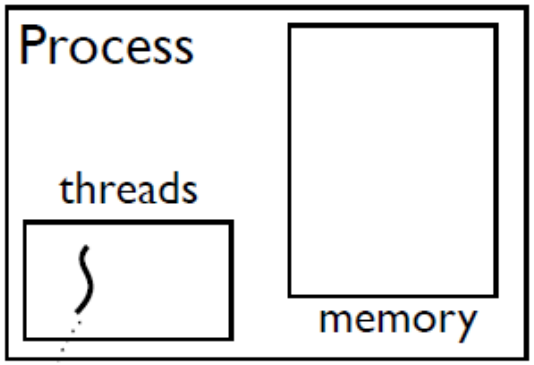
A Kernel Crossing Illustrated



How do we pass data to/from a system call?



How do we pass data to/from a system call?



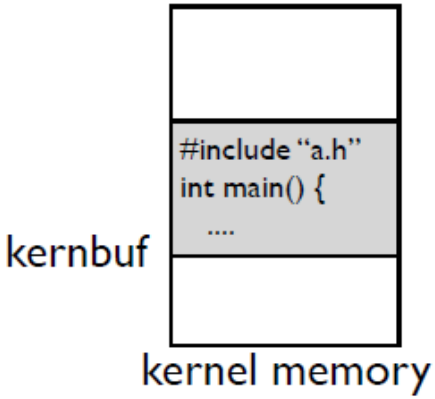
BUG: user can't access kernel memory!

User Space

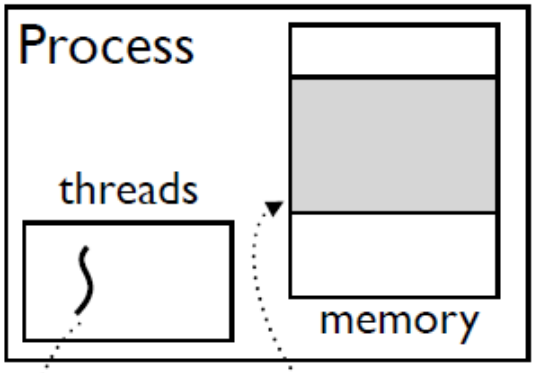
```
buf = ReadFile(...);
print(buf);
```

Kernel Space

```
ReadFile(...) {
    ...
    return &kernbuf;
}
```



How do we pass data to/from a system call?



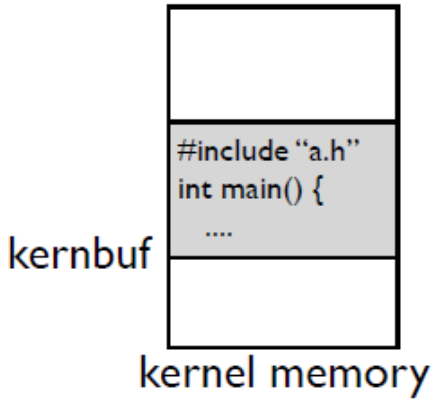
OK: kernel can access user memory

User Space

```
ReadFile(&buf, ...);
print(buf);
```

Kernel Space

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



How do we pass data to/from a system call?

Evil user program:

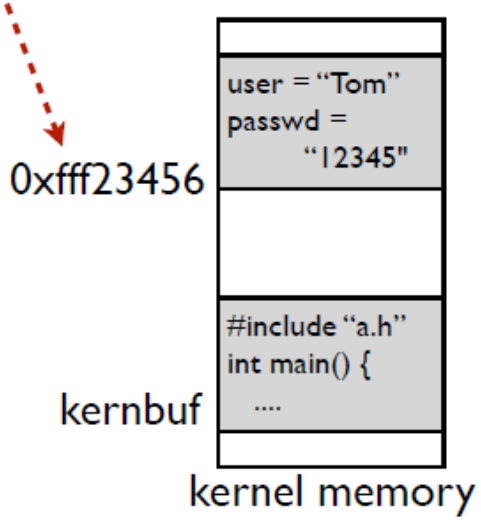
```
ReadFile((char*)0xfff23456, ...);
    // manufacture a buffer ptr
    // hope we get lucky and it points at
    // a kernel data structure!
```

User Space

Kernel Space

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

Kernel must validate user buffers!



How do we pass data to/from a system call?

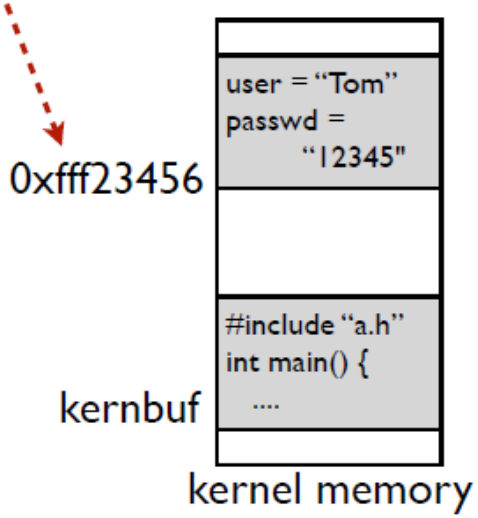
Evil user program:

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

User Space

Kernel Space

```
ReadFile(char* userbuf,
         int userlen) {
  ...
  ProbeForWrite(userbuf, userlen, ..);
    // fails if userbuf is not valid
    // memory in user space
  ...
  memcpy(userbuf, kernbuf, sz);
  ...
```



Kernel Sys Call Function Tasks

1. Locate system call arguments
   - Arguments to a sys call stored in user memory
   - Thus it is a virtual address
   - Must be checked to make sure it is within the users domain
   - Must be converted to a physical address

Kernel Sys Call Function Tasks

2. Validate Parameters
   - Kernel must protect itself from errors in the arguments
   - Example:
     - Normally a file name is a zero-terminated string
     - The name could be corrupted
     - It could point to memory outside the application's region
     - Could start inside the application's memory but extend beyond it
   - If an error is detected, kernel returns to the user with an error
- What happens if you change the parameters after the check?

Kernel Sys Call Function Tasks

3. Copy before check
   - Kernel copies sys call parameters into kernel memory before validating them
   - Used to prevent Time of use to time of check attacks

Kernel Sys Call Function Tasks

4. Copy back results
   - If sys call reads data into a buffer in user memory, that data needs to be copied from the kernel buffer
   - Kernel first checks the user address and converts it into a kernel address, then copies

Putting it all together

```
int KernelStub_Open() {
    char *localCopy[MaxFileNameSize + 1];

    // check that stack pointer is valid and that the arguments are stored at valid addresses
    if(!validUserAddressRange(userStackPointer, userStackPointer + size of arguments on stack))
        return error_code;

    // fetch pointer to file name from user stack, and convert to a kernel pointer
    filename = VirtualToKernel(userStackPointer);

    // make a local copy of the filename, inside the OS. This prevents the application from
    // changing the name after the check, but before the read
    if(!VirtualToKernelStringCopy(filename, localCopy, MaxFileNameSize))
        return error_code;

    // make sure local copy is null terminated
    localCopy[MaxFileNameSize] = 0;

    // check that the user is permitted to access this file
    if(!UserFileAccessPermitted(localCopy, current_process)
        return error_code;

    // now we can call the actual routine to open the file
    return Kernel_Open(localCopy);
}
```

Sys Calls and File Handles

- To start accessing a file, a process calls open() to get a file handle (file descriptor in linux)
- The OS requires that files be accessed through file handles and not by just passing the file path to read() and write()
- Why?

Sys Calls and File Handles

1. Path parsing and permission checking are only required when file is opened
   - No need to repeat on each read or write

Sys Calls and File Handles

1. Path parsing and permission checking are only required when file is opened
   - No need to repeat on each read or write
2. When a file is opened, OS creates a data structure that:
   - Keeps track of file's ID
   - Whether a process has read or write permissions
   - A pointer to the processes current position within the file