PROGRAMMING INTERFACE

Slides adapted from Tom Anderson's "Operating Systems: Principles and Practice"

Functions of the Operating System

- Creating and managing processes
  - fork, exec, wait
- Performing I/O
  - open, read, write, close
- Communicating between processes
  - pipe, dup, select, connect
- Thread management
- Memory management
  - brk
- Networking
  - socket
- Authentication and security
- …

Shell

- A shell is a job control system
  - Allows programmer to create and manage a set of programs to do some task
  - Windows, MacOS, Linux all have shells

- Example: to compile a C program
  - cc –c sourcefile1.c
  - cc –c sourcefile2.c
  - In –o program sourcefile1.o sourcefile2.o

Question

- If the shell runs at user-level, what system calls does it make to run each of the programs?
  - Ex: cc, ln

Windows CreateProcess

- System call to create a new process to run a program
  - Create and initialize the process control block (PCB) in the kernel
  - Create and initialize a new address space
  - Load the program into the address space
  - Copy arguments into memory in the address space
  - Initialize the hardware context to start execution at ``start''
  - Inform the scheduler that the new process is ready to run

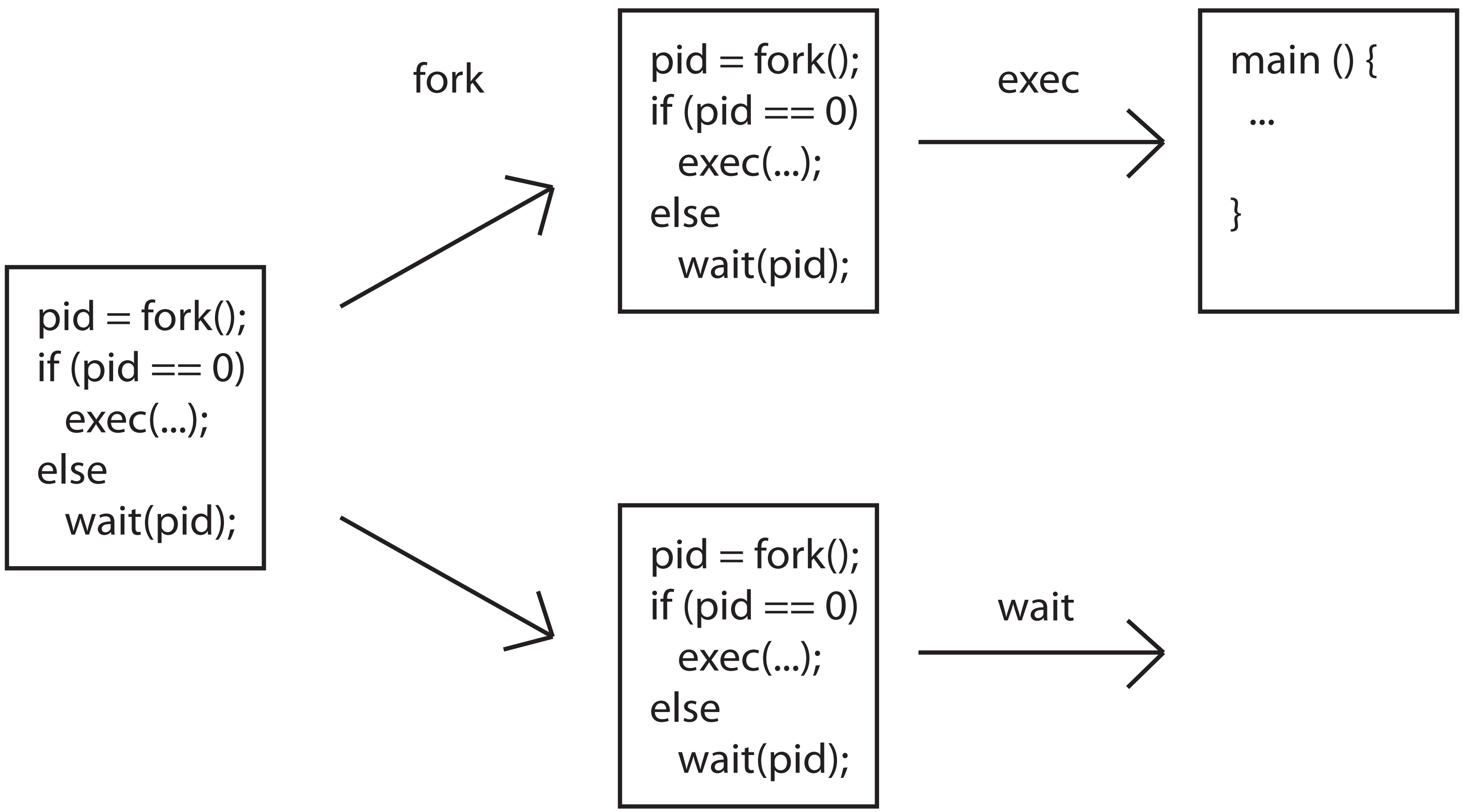
Windows CreateProcess API (simplified)

```
if (!CreateProcess(
    NULL,   // No module name (use command line)
    argv[1], // Command line
    NULL,   // Process handle not inheritable
    NULL,   // Thread handle not inheritable
    FALSE,  // Set handle inheritance to FALSE
    0,      // No creation flags
    NULL,   // Use parent's environment block
    NULL,   // Use parent's starting directory
    &si,    // Pointer to STARTUPINFO structure
    &pi )   // Pointer to PROCESS_INFORMATION structure
)
```

UNIX Process Management

- UNIX **fork** – system call to create a copy of the current process, and start it running
  - No arguments!
- UNIX **exec** – system call to change the program being run by the current process
- UNIX **wait** – system call to wait for a process to finish
- UNIX **signal** – system call to send a notification to another process

UNIX Process Management



Question: What does this code print?

```
int child_pid = fork();

if (child_pid == 0) {   // I'm the child process
    printf("I am process #%d\n", getpid());
    return 0;
}
else {                           // I'm the parent process
    printf("I am parent of process #%d\n", child_pid);
    return 0;
}
```

Questions

- Can UNIX fork() return an error? Why?
- Can UNIX exec() return an error? Why?
- Can UNIX wait() ever return immediately? Why?

Implementing UNIX fork

- Steps to implement UNIX **fork**
  - Create and initialize the process control block (PCB) in the kernel
  - Create a new address space
  - Initialize the address space with a copy of the entire contents of the address space of the parent
  - Inherit the execution context of the parent (e.g., any open files)
  - Inform the scheduler that the new process is ready to run

Implementing UNIX exec

- Steps to implement UNIX **exec**
  - Load the program into the current address space
  - Copy arguments into memory in the address space
  - Initialize the hardware context to start execution at “start"

UNIX I/O

- Uniformity
  - All operations on all files, devices use the same set of system calls: open, close, read, write
- Open before use
  - Open returns a handle (file descriptor) for use in later calls on the file
- Byte-oriented
- Kernel-buffered read/write
- Explicit close
  - To garbage collect the open file descriptor

UNIX File System Interface

- UNIX file open is a Swiss Army knife:
  - Open the file, return file descriptor
  - Options:
    - if file doesn't exist, return an error
    - If file doesn't exist, create file and open it
    - If file does exist, return an error
    - If file does exist, open file
    - If file exists but isn't empty, nix it then open
    - If file exists but isn't empty, return an error
    - …

Interface Design Question

- Why not separate syscalls for open/create/exists?

```
if (!exists(name))
     create(name);  // can create fail?
fd = open(name);    // does the file exist?
```

Implementing a Shell

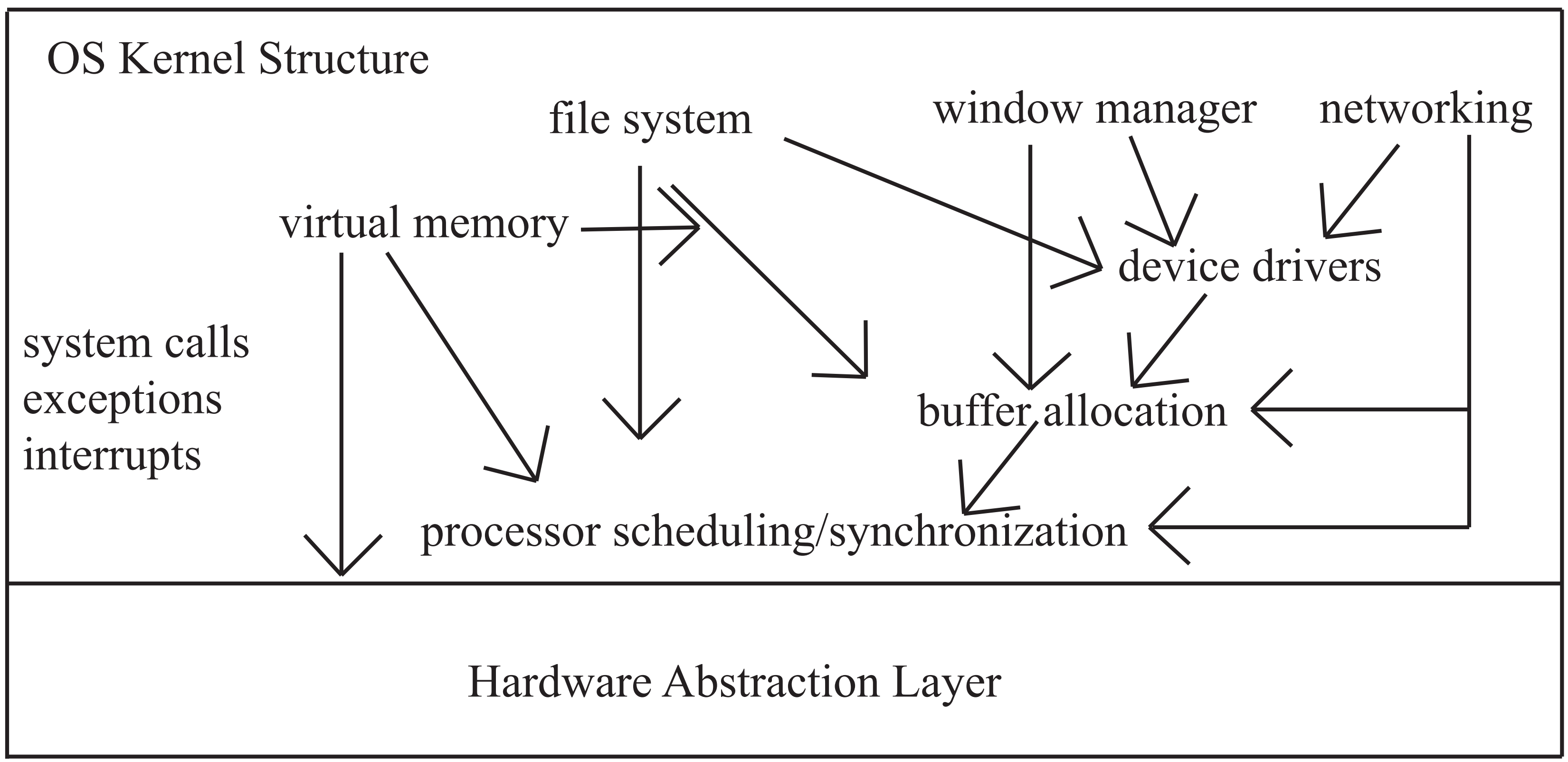
```
char *prog, **args;
int child_pid;

// Read and parse the input a line at a time
while (readAndParseCmdLine(&prog, &args)) {
   child_pid = fork();      // Create a child process
    if (child_pid == 0) {
       exec(prog, args); // I'm the child process. Run program
      // NOT REACHED
    }
    else {
       wait(child_pid); // I'm the parent, wait for child
       return 0;
    }
}
```

OPERATING SYSTEM STRUCTURE

Monolithic kernels vs microkernels

Slides adapted from Tom Anderson's "Operating Systems: Principles and Practice"

Monolithic Kernel



HAL and Device Drivers

- **HAL**: portable interface to machine-specific operations within the kernel.

- Huge number of different types of physical I/O devices, manufactured by a large number of companies.
  → 70% of the code in the Linux kernel was in device specific software.

- **Dynamically loadable device driver:**
  - software to manage a specific device or interface or chipset
  - added to the operating system kernel after the kernel starts running

- At boot, the operating system starts with a small number of device drivers
  → Queries the I/O buses for devices, and loads the drivers
  → 90% of the errors in an OS are due to bugs in the device drivers

Monolithic Kernel - System Call

Process P running in User level

Call function
read (fd, buffer, nbytes)

```
push nbytes
push buffer
push fd
call read
…
```

```
mov eax, 5 # sys_read entry
…
int 0x80 # system call
…
ret
```

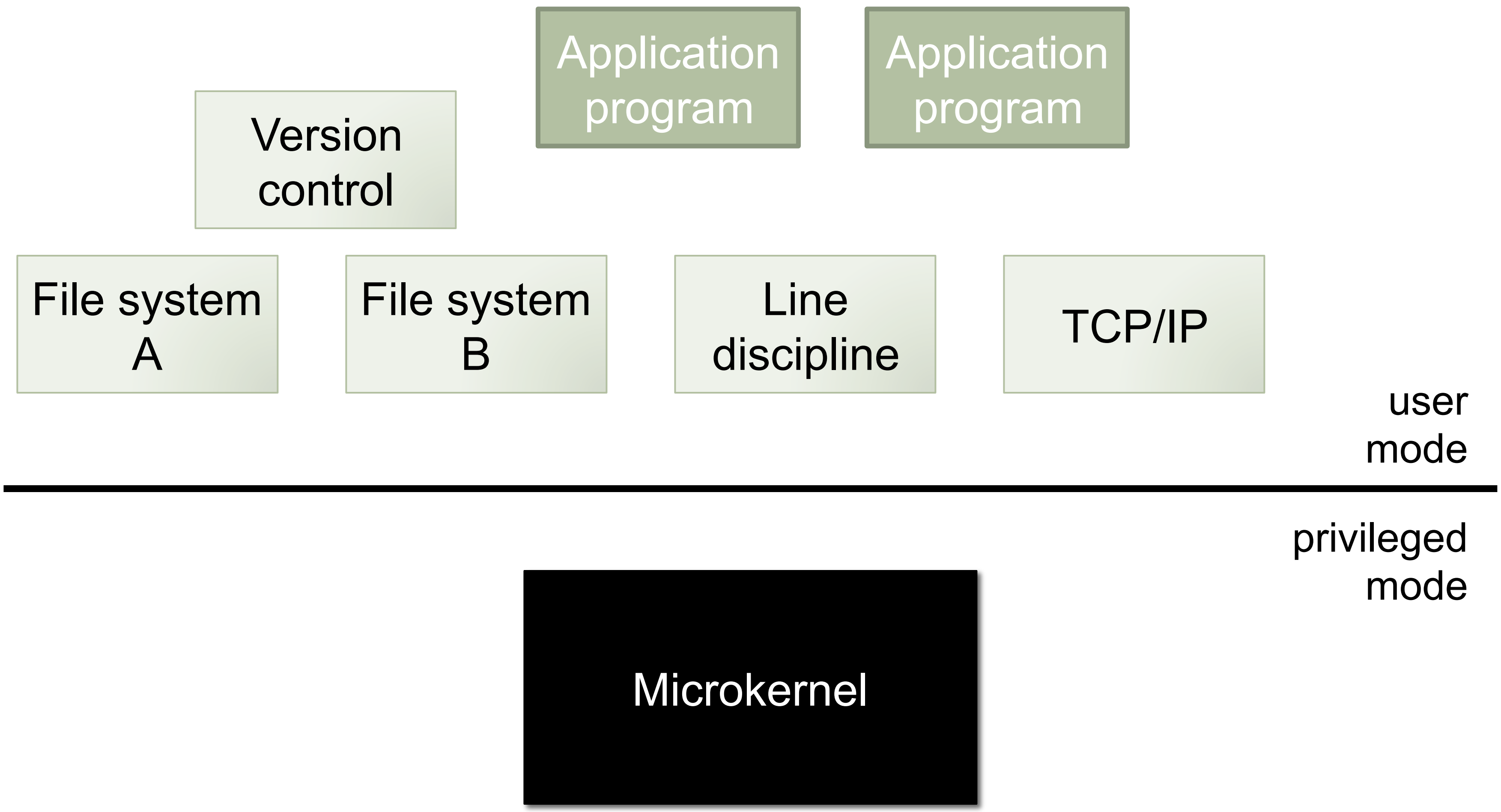
Process P running in Kernel level

Execution of system call read

Software
interrupt

```
syscall_table[eax]
…
iret
```

OS Services as User Apps



Why?

- It's cool …

- Assume that OS coders are incompetent, malicious, or both …
  - OS components run as protected user-level applications

- Extensibility
  - easier to add, modify, and extend user-level components than kernel components

Implementation Issues

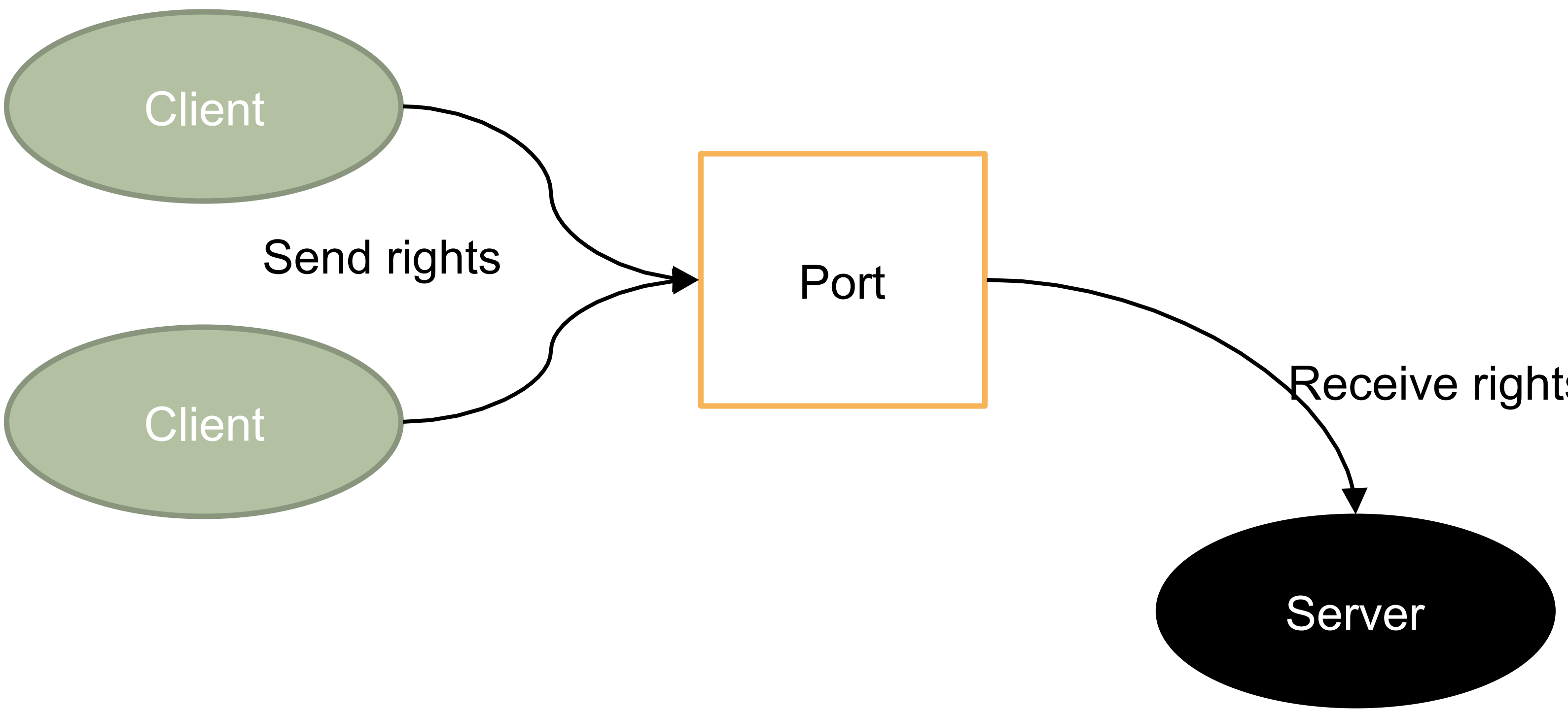
- How are modules linked together?
- How is data moved around efficiently?

Mach

- Developed at CMU, then Utah
- Early versions shared kernel with Unix
  - basis of NeXT OS
- Later versions still shared kernel with Unix
  - basis of OSF/1
    - basis of Macintosh OS X
- Even later versions actually functioned as working microkernel
  - basis of GNU/HURD project
    - HURD: HIRD of Unix-replacing daemons
    - HIRD: HURD of interfaces representing depth

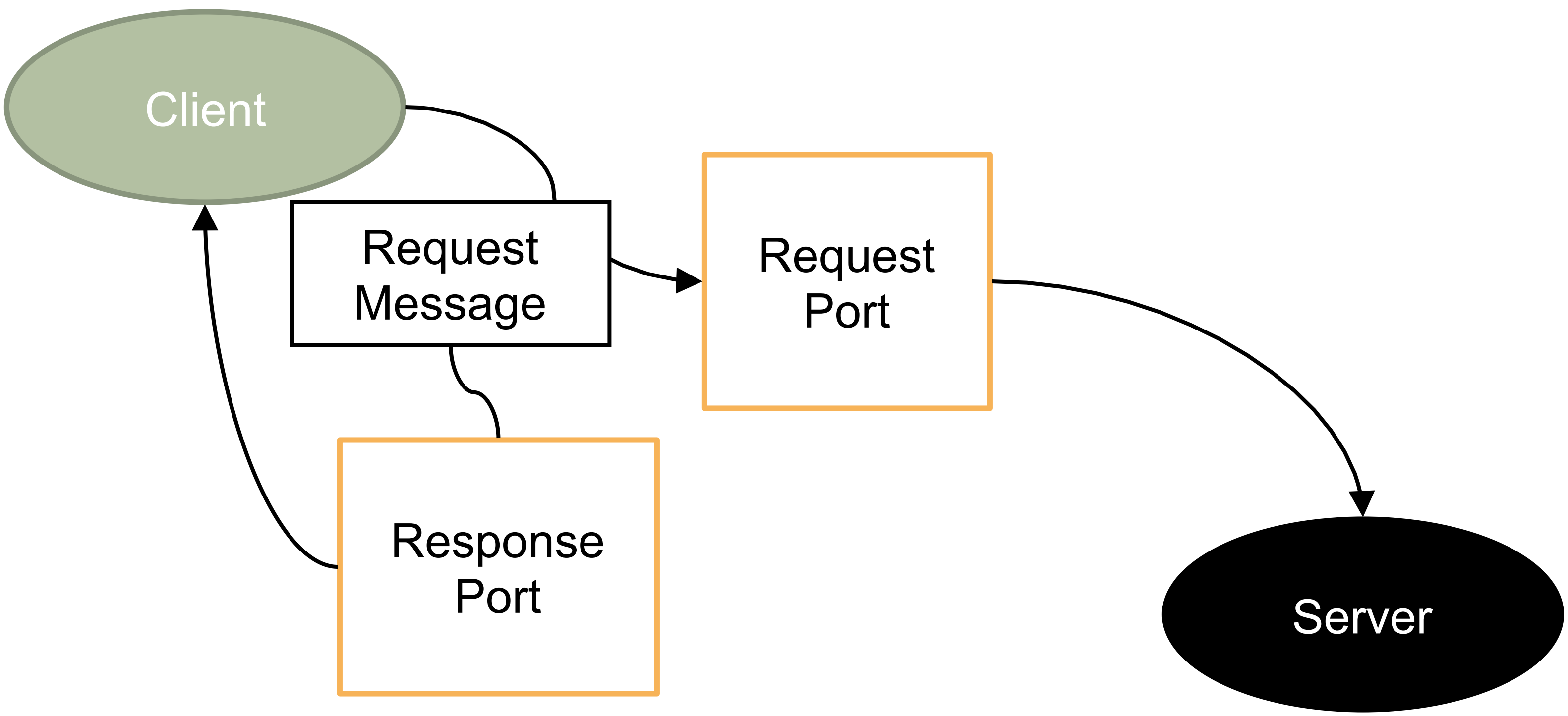
Mach Ports (1)

- Linkage construct



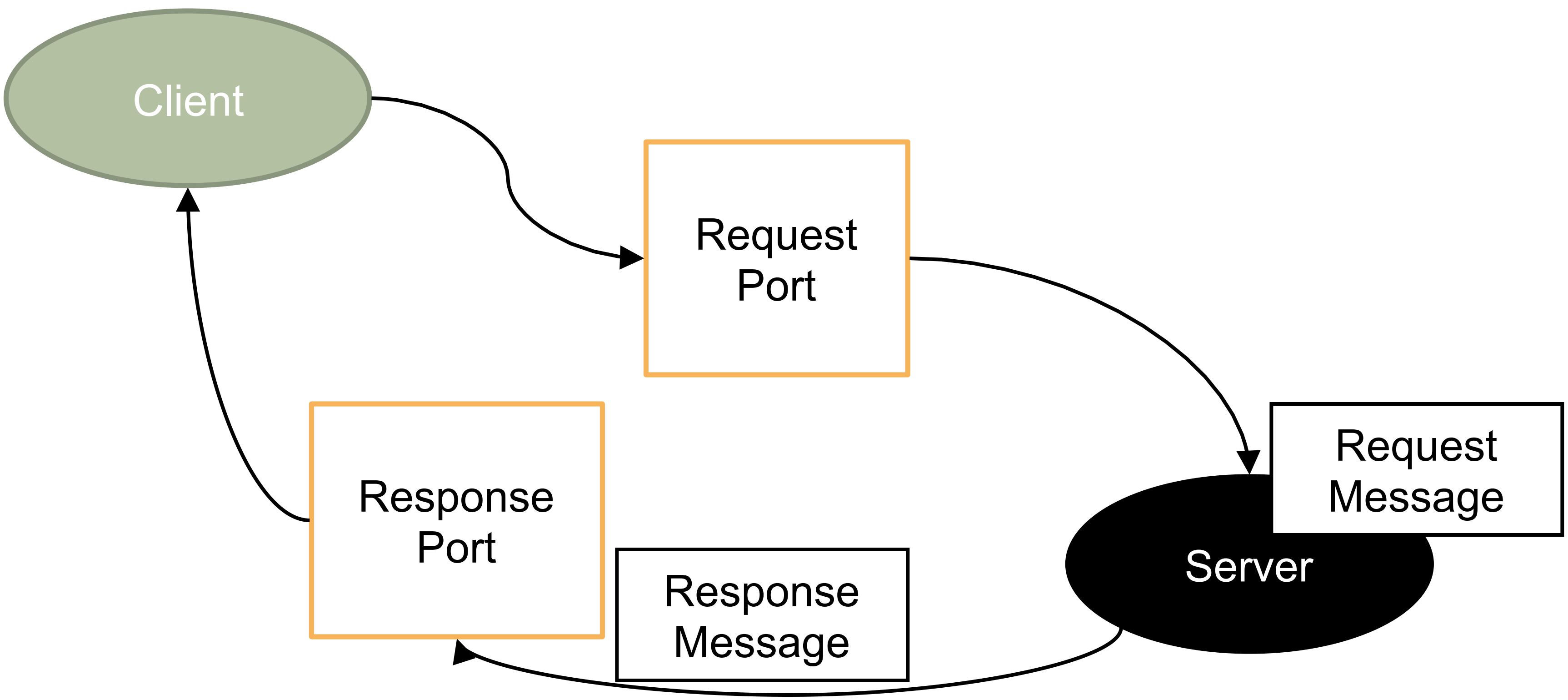
Mach Ports (2)

- Communication construct



Mach Ports (3)

- Communication construct

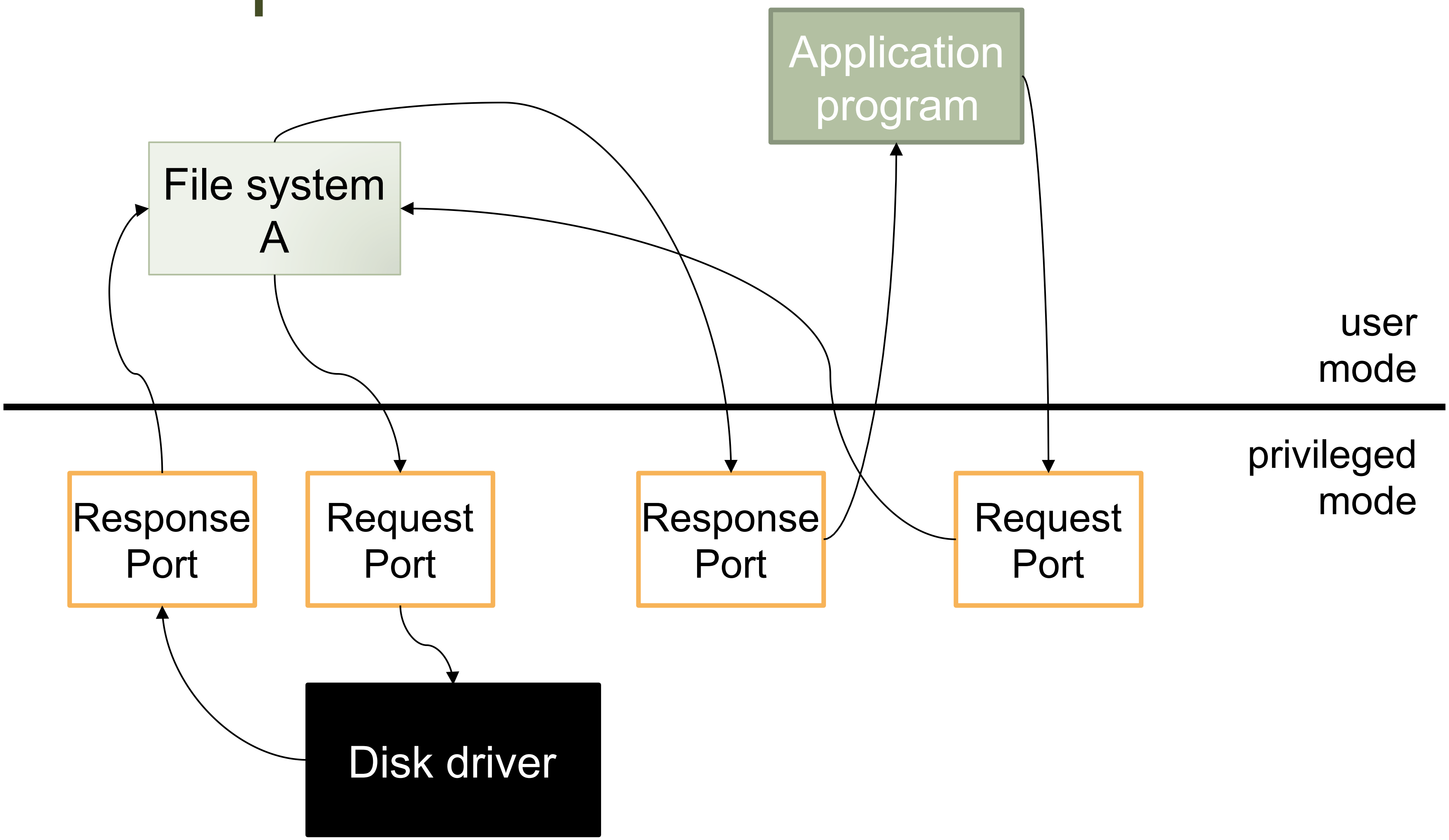


RPC

- Ports used to implement *remote procedure calls*
  - Communication across process boundaries

- if procedures are on same machine …
  - local RPC

Example


Application program
File system A
user mode
privileged mode
Response Port
Request Port
Response Port
Request Port
Disk driver

Successful Microkernel Systems

-
-
- …

Attempts

- Windows NT 3.1
  - graphics subsystem ran as user-level process
  - moved to kernel in 4.0 for performance reasons
- Macintosh OS X
  - based on Mach
  - all services in kernel for performance reasons
- HURD
  - based on Mach
  - services implemented as user processes
  - no one uses it, for performance reasons …