Systems Calls and IPC

COMS W4118

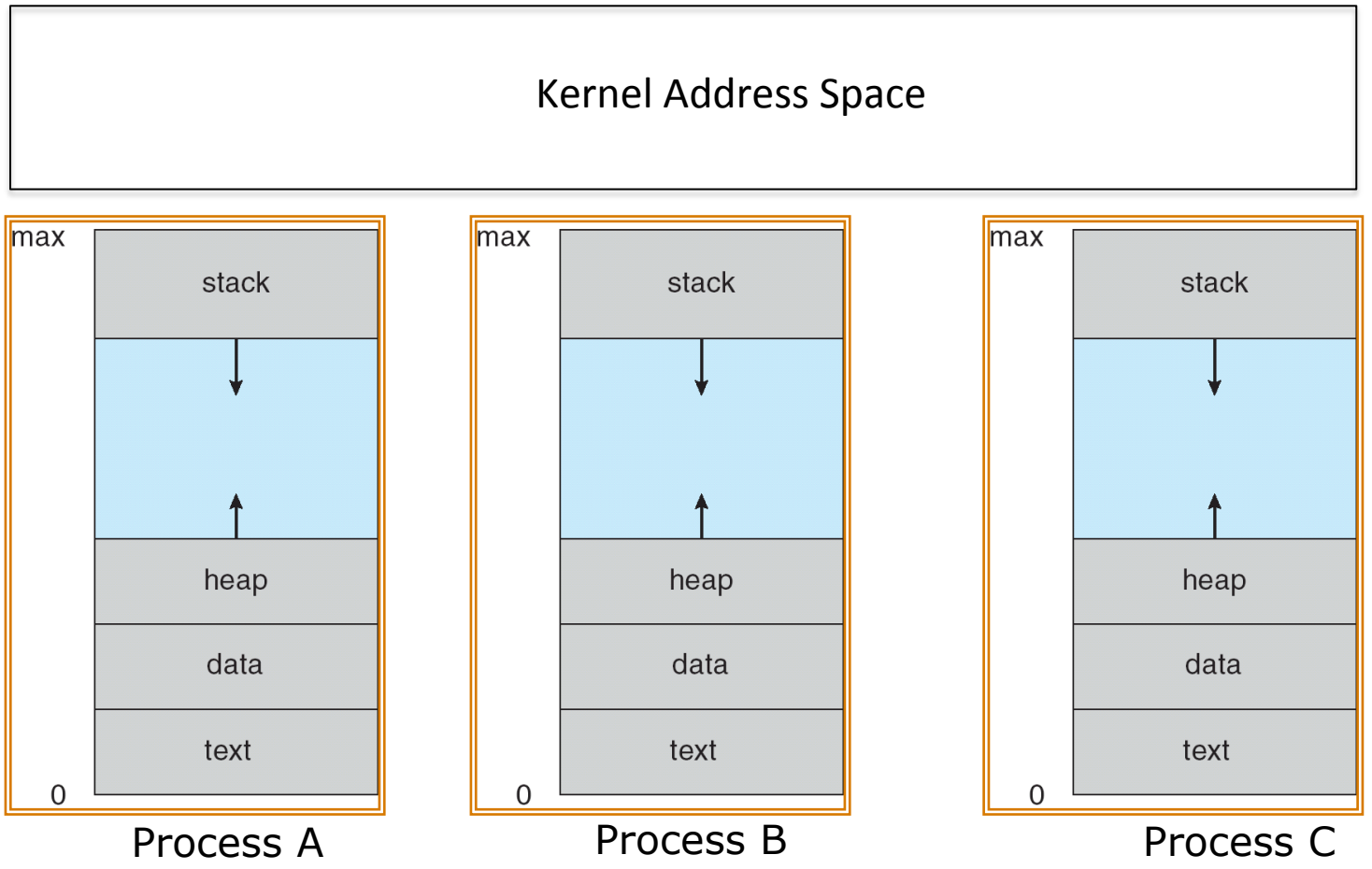
Prof. Kaustubh R. Joshi

krj@cs.columbia.edu

http://www.cs.columbia.edu/~krj/os

References: Operating Systems Concepts (9e), Linux Kernel Development, previous W4118s

Address Space Overview



- Processes can't access anything outside address space
- How do they communicate with outside world?

Outline

- System Calls
- Signals
- Co-operating Processes
- Shared Memory
- Message based IPC

System calls

- User processes cannot perform privileged operations themselves
- Must request OS to do so on their behalf by issuing system calls
- Basic concepts (today), more details on how done on Linux (later)

System Calls

- Programming interface to the services provided by the OS
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Program Interface (API)** rather than direct system call use
- E.g., Win32 API for Windows and POSIX API (UNIX, Linux, and Mac OS X)
- Why use APIs rather than system calls?
  - Exact mechanism to invoke varies for different hardware
  - Changes with type (e.g., x86 moved from int to sysenter)
  - Backward compatibility
  - Portability

Example of Standard API

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the `man` page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

ssize_t     read(int fd, void *buf, size_t count)
```

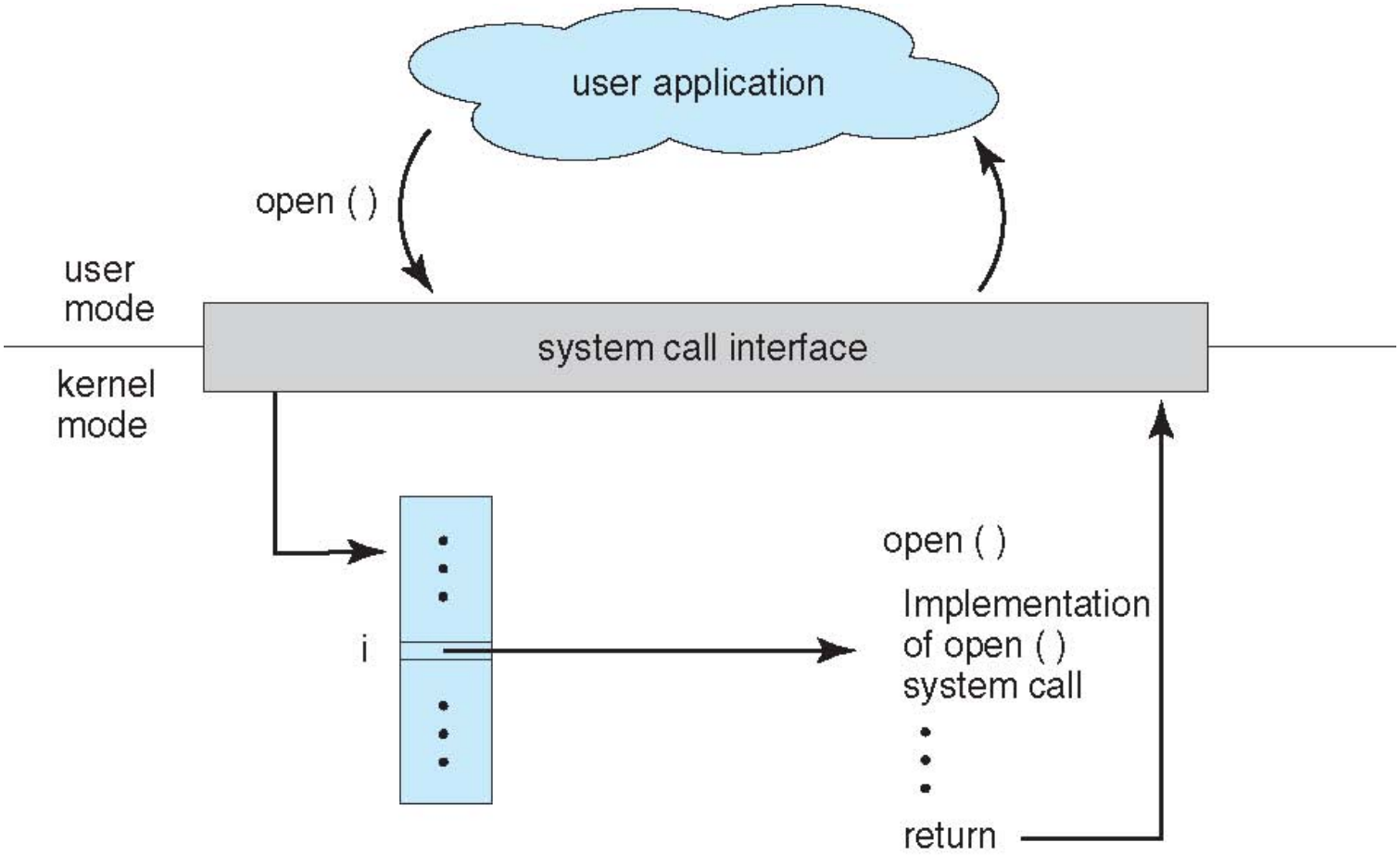
return value — function name — parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer where the data will be read into
- `size_t count`—the maximum number of bytes to be read into the buffer

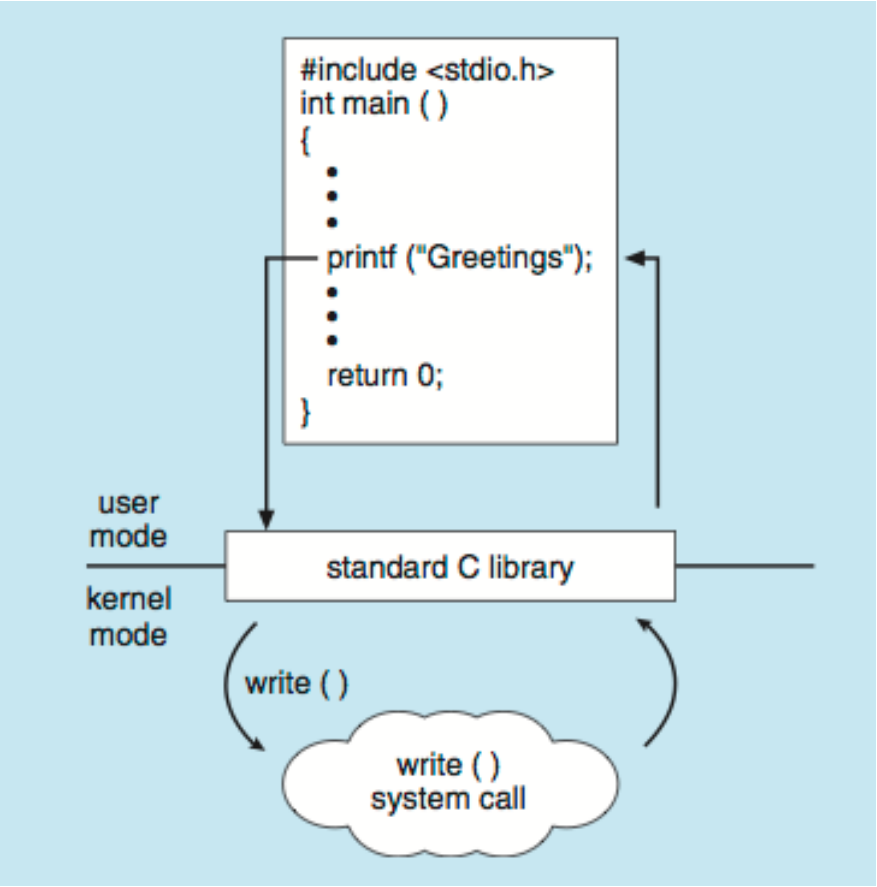
On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns −1.

Library vs. System Calls



Library vs. System Calls

- C program invoking printf() libc library call, which calls write() system call



Types of System Calls

- Process control
  - end, abort
  - load, execute
  - create process, terminate process
  - get process attributes, set process attributes
  - wait for time
  - wait event, signal event
  - allocate and free memory

  - Dump memory if error
  - **Debugger** for determining **bugs, single step** execution
  - **Locks** for managing access to shared data between processes

Types of System Calls

- File management
  - create file, delete file
  - open, close file
  - read, write, reposition
  - get and set file attributes
- Device management
  - request device, release device
  - read, write, reposition
  - get device attributes, set device attributes
  - logically attach or detach devices

Types of System Calls (Cont.)

- Information maintenance
  - get time or date, set time or date
  - get system data, set system data
  - get and set process, file, or device attributes

- Communications
  - create, delete communication connection
  - send, receive messages if **message passing model** to **host name** or **process name**
  - **Shared-memory model** create and gain access to memory regions
  - transfer status information
  - attach and detach remote devices

Types of System Calls (Cont.)

- Protection
  - Control access to resources
  - Get and set permissions
  - Allow and deny user access

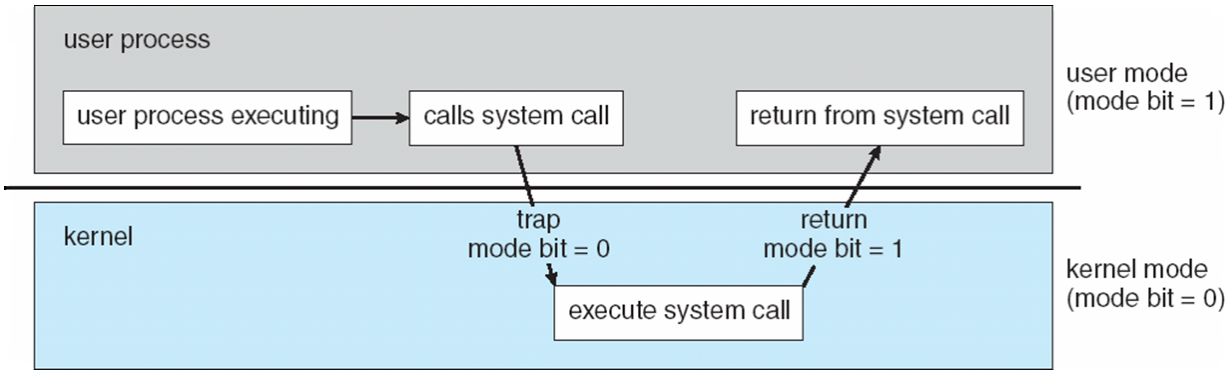
See “man syscalls” on any Linux system

Examples of Windows and Unix System Calls

	Windows	Unix
Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

System Call Dispatch

- How should actual system call be invoked?
  - Program can't see kernel namespace



- Need hardware support to change privilege level
- Traps
  - Type of interrupt
  - Software interrupts and exceptions
  - Software interrupts initiated by programmer
  - Exceptions occur automatically

Traps, Interrupts, Exceptions

```
for(;;) {
   if (interrupt) {
       n = get interrupt number
       call interrupt handler n
   }
   fetch next instruction
   run next instruction
}
```

```
for(;;) {
   fetch next instruction
   run next instruction {
       if (instr == "int n")
         call interrupt handler n
   }
   if (error or interrupt) {
       n = get error or interrupt type
       call interrupt handler n
   }
}
```

- On x86, int n (n=0:255) calls interrupts n
- Some interrupts are privileged
- Can't be called by user mode
- Others aren't, e.g., syscalls
- Processor transitions to privileged mode when handling interrupt

x86 Hardware Exceptions

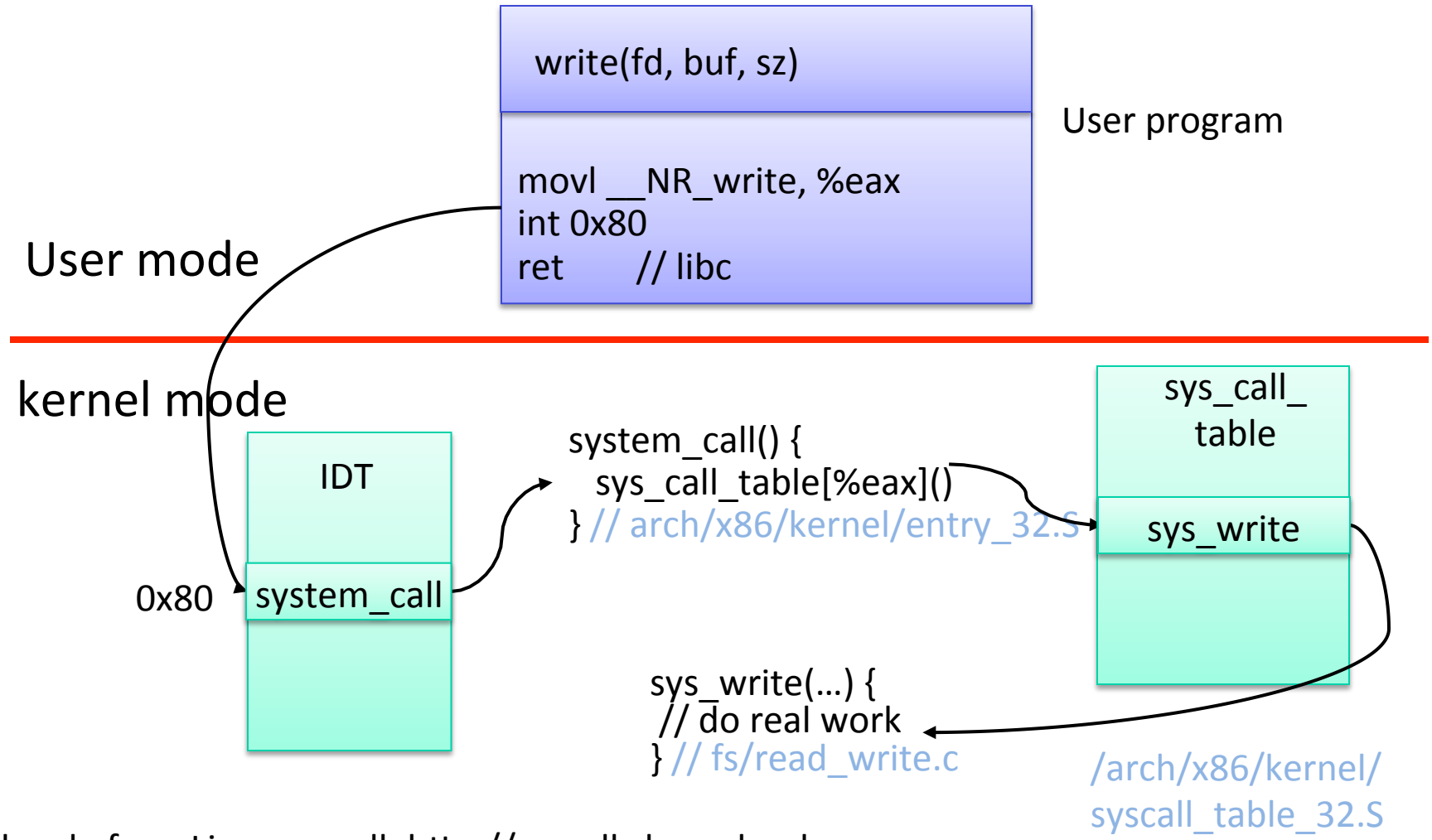
- Intel Reserved first 32 interrupts for exceptions.
- OS can use others: Linux used 0x80 for syscalls

- 0x00 Division by zero
- 0x01 Debugger
- 0x03 Breakpoint
- 0x04 Overflow
- 0x05 Bounds
- 0x06 Invalid Opcode
- 0x07 Coprocessor not available
- 0x08 Double fault
- 0x0A Invalid Task State Segment
- 0x0B Segment not present
- 0x0C Stack Fault
- 0x0D General protection fault
- 0x0E Page fault
- 0x10 Math Fault
- 0x11 Alignment Check
- 0x12 Machine Check
- 0x13 SIMD Floating-Point Exception

System call dispatch

1. Kernel assigns system call type a system call number
2. Kernel initializes system call table, mapping system call number to functions implementing the system call
   - Also called system call vector
3. User process sets up system call number and arguments
4. User process runs int X (on Linux, X=80h)
5. Hardware switches to kernel mode and invokes kernel's interrupt handler for X (interrupt dispatch)
6. Kernel looks up syscall table using system call number
7. Kernel invokes the corresponding function
8. Kernel returns by running iret (interrupt return)

Linux System Call Dispatch

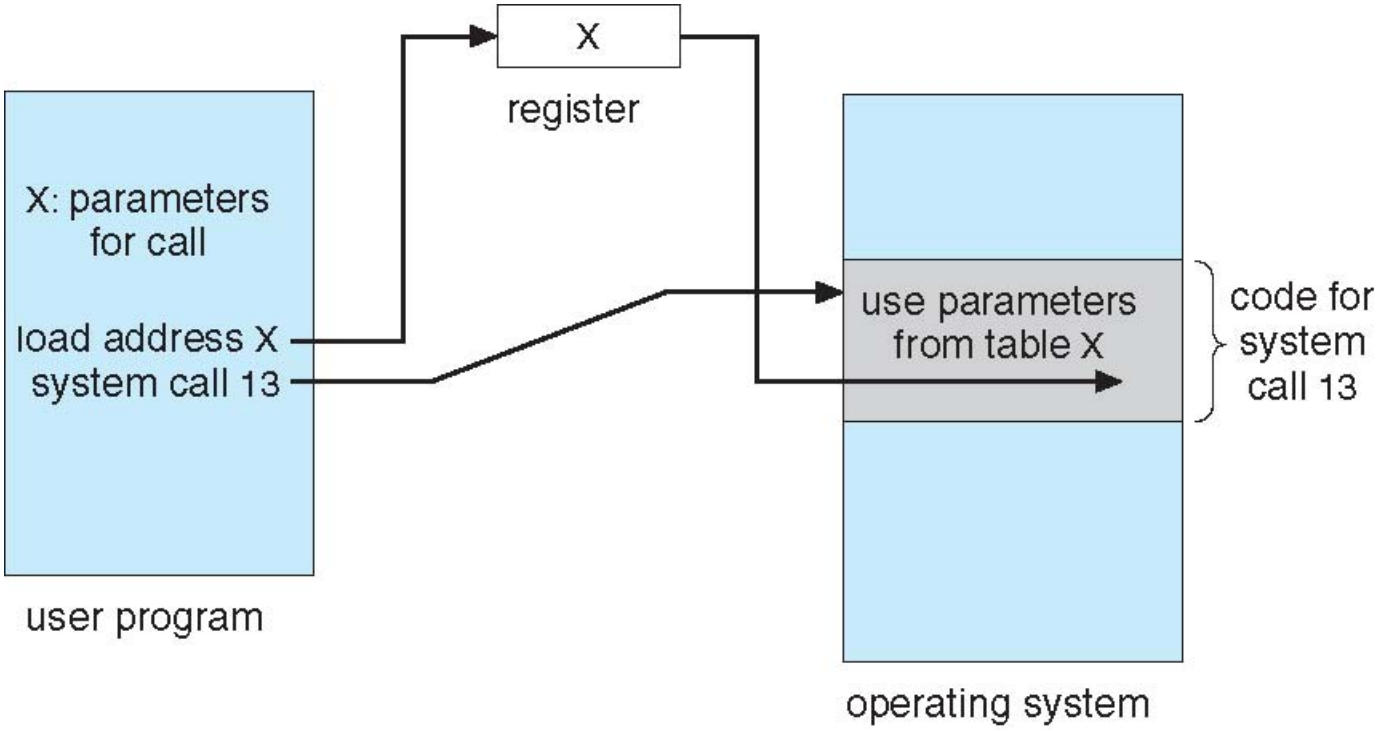


To find code for a Linux syscall: http://syscalls.kernelgrok.com

System call parameter passing

- Typical methods
  - Pass via registers (e.g., Linux)
    - More parameters than registers?
  - Pass via user-mode stack
    - Complex: user mode and kernel mode stacks
  - Pass via designated memory region
    - Address passed in register

Parameter Passing via Table



Linux System Call Parameter Passing

- Syscalls with less than 6 parameters passed in registers
  - %eax (syscall number), %ebx, %ecx, %esi, %edi, %ebp
- If more than 6 arguments
  - Pass pointer to block structure containing argument list
- Maximum size of argument is register size
  - Larger arguments passed as pointers
  - Stub code copies parameters onto kernel stack before calling syscall code (kernel stack, will study later)
- Use special routines to fetch pointer arguments
  - get_user(), put_user(), copy_to_user(), copy_from_user
  - Include/asm/uaccess.S
  - These functions can block. Why?
  - Why use these functions?
- OS must validate system call parameters

Linux system call naming convention

- Usually the user-mode wrapper foo() traps into kernel, which calls sys_foo()
  - sys_foo is implemented by DEFINEx(foo, …)
  - Expands to "asmlinkage long sys_foo(void)"
  - Where x specifies the number of parameters to syscall
  - Often wrappers to foo() in kernel
- System call number for foo() is __NR_foo
  - arch/x86/include/asm/unistd_32.h
  - Architecture specific
- All system calls begin with sys_

System Call from Userspace

- Generic syscall stub provided in libc
  - _syscalln
  - Where n is the number of parameters
- Example
  - To implement: ssize_t write(int fd, const void *buf, size_t count);
  - Declare:

```
#define __NR_write 4 /* Syscall number */
_syscall3(ssize_t, write, int, fd, const void*, buf, size_t count)
```

- Usually done in libc for standard syscalls

Tracing system calls in Linux

- Use the "strace" command (man strace for info)
- Linux has a powerful mechanism for tracing system call execution for a compiled application
- Output is printed for each system call as it is executed, including parameters and return codes
- ptrace() system call is used to implement strace
  - Also used by debuggers (breakpoint, singlestep, etc)
- Use the "ltrace" command to trace dynamically loaded library calls

System Call Tracing Demo

- ssh clic-lab.cs.columbia.edu
- pwd
- ltrace pwd
  - Library calls
  - setlocale, getcwd, puts: makes sense
- strace pwd
  - System calls
  - execve, open, fstat, mmap, brk: what are these?
  - getcwd, write

Interesting System Calls

- brk, sbrk: increase size of program data
  - void* sbrk(int bytes)
  - Accessed through malloc
- mmap
  - Another way to allocate memory
  - Maps a file into a process's address space
  - More a bit later

Outline

- System Calls
- Signals
- Co-operating Processes
- Shared Memory
- Message based IPC

Kernel to Process Notification

- Hardware exceptions
  - Those that kernel can't handle
  - E.g., divide by 0
- Timer callbacks
  - What does user want to do?
- Process termination
  - Run cleanup code
- How to notify?

UNIX Signals

- Signals
  - A very short message: just a small integer
  - A fixed set of available signals. Examples:
    - 9: kill
    - 11: segmentation fault
- Installing a handler for a signal
  - sighandler_t signal(int signum, sighandler_t handler);
- Blocking signals
  - sigprocmask(SIGBLOCK, sigset_t *set, sigset_t *oldset);
- Send a signal to a process
  - kill(pid_t pid, int sig)

POSIX Signals

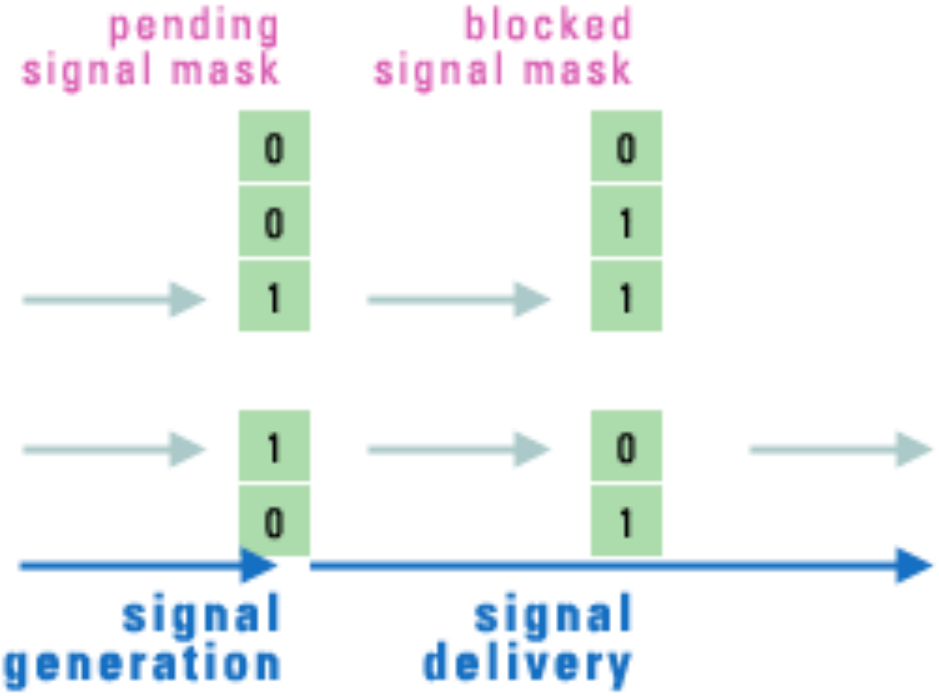
- What kinds of signals? Examples defined by POSIX
- Process Control
  - SIGINT: Ctrl+C, default: terminate process
  - SIGTSTP: Ctrl+Z, default: suspend process
  - SIGCONT: resume process stopped with SIGSTP
  - SIGQUIT: Ctrl+\, default: terminate and dump core
  - SIGKILL: kill process (cannot be caught or ignored)
- Exceptions or Errors
  - SIGSEGV: segmentation fault
  - SIGFPE: arithmetic exception (div by zero)
  - SIGPIPE: notify when writing to reader closed pipe
  - SIGILL: illegal instruction
  - SIGSYS: illegal syscall parameters
- External events
  - SIGALRM: timer set by alarm function
  - SIGCHLD: child process terminated or stopped
  - SIGUSR1, SIGUSR2: user defined (can be raised by kill)
  - SIGPOLL: asynchronous I/O event that user was waiting for has happened
  - SIGURG: socket has urgent out of band data available to read

Synchronous vs. Asynchronous

- When is a signal delivered?
- Synchronous signals
  - Raised in response to a process's own activities
  - E.g., error or exception (SIGSEGV, SIGFPE)
  - Process is not allowed to continue until signal is handled
- Asynchronous signals
  - Raised due to external events
  - Can occur at any time
  - No guarantees on when they are delivered

Signal Delivery

- Signal generation mask
- Signal delivery mask
- Scheduler checks before returning control
- Pending signals continue to be expressed until handled



Signal Handler Gotchas

- Closest thing in userspace to interrupts
- Similar issues (we'll see more later)
- Asynchronous: can happen anytime
- Can be nested (even signals of the same type)
  - New signal during signal handler routine
  - Can't use non re-entrant functions
- Can interrupt system call in progress
- Race conditions

Outline

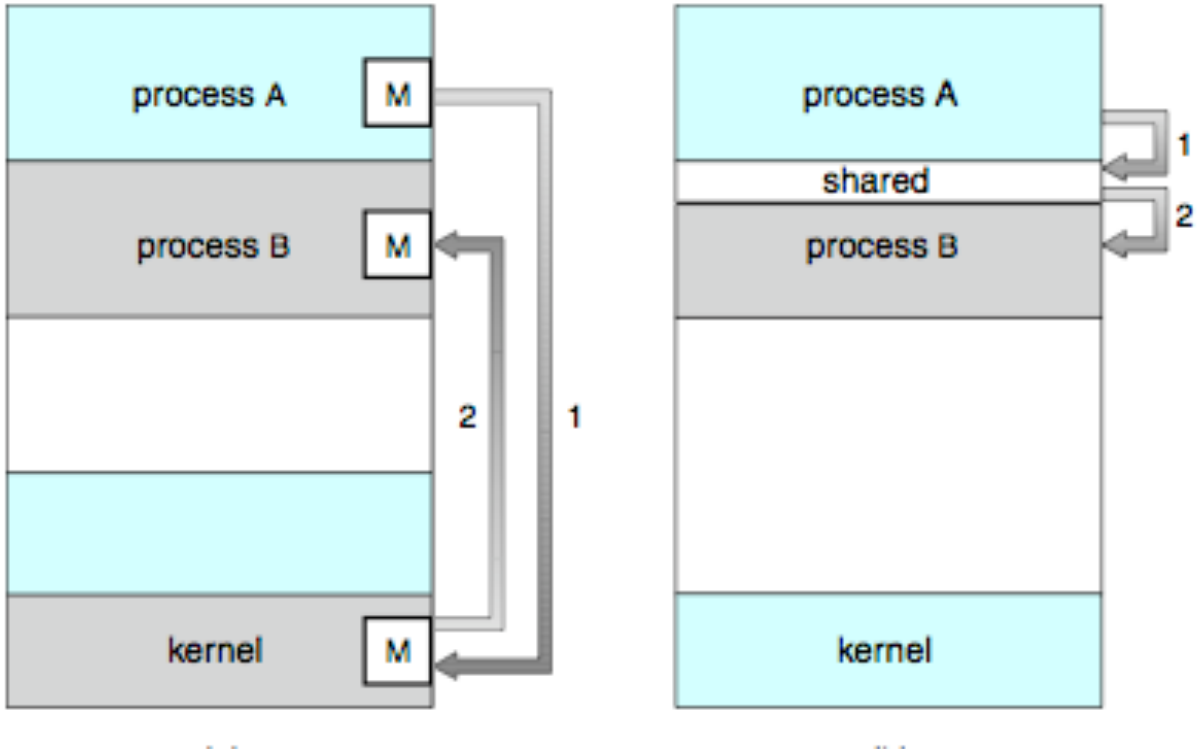
- System Calls
- Signals
- Co-operating Processes
- Shared Memory
- Message based IPC

Cooperating Processes

- Independent process cannot affect or be affected by the execution of another process.
- Cooperating process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity/Convenience
  - System services (microkernels)

Interprocess Communication Models

Message Passing **Shared Memory**



Message Passing vs. Shared Memory

- Message passing
  - Why good? All sharing is explicit → less chance for error
  - Why bad? Overhead. Data copying, cross protection domains
- Shared Memory
  - Why good? Performance. Set up shared memory once, then access w/o crossing protection domains
  - Why bad? Things change behind your back → error prone

Outline

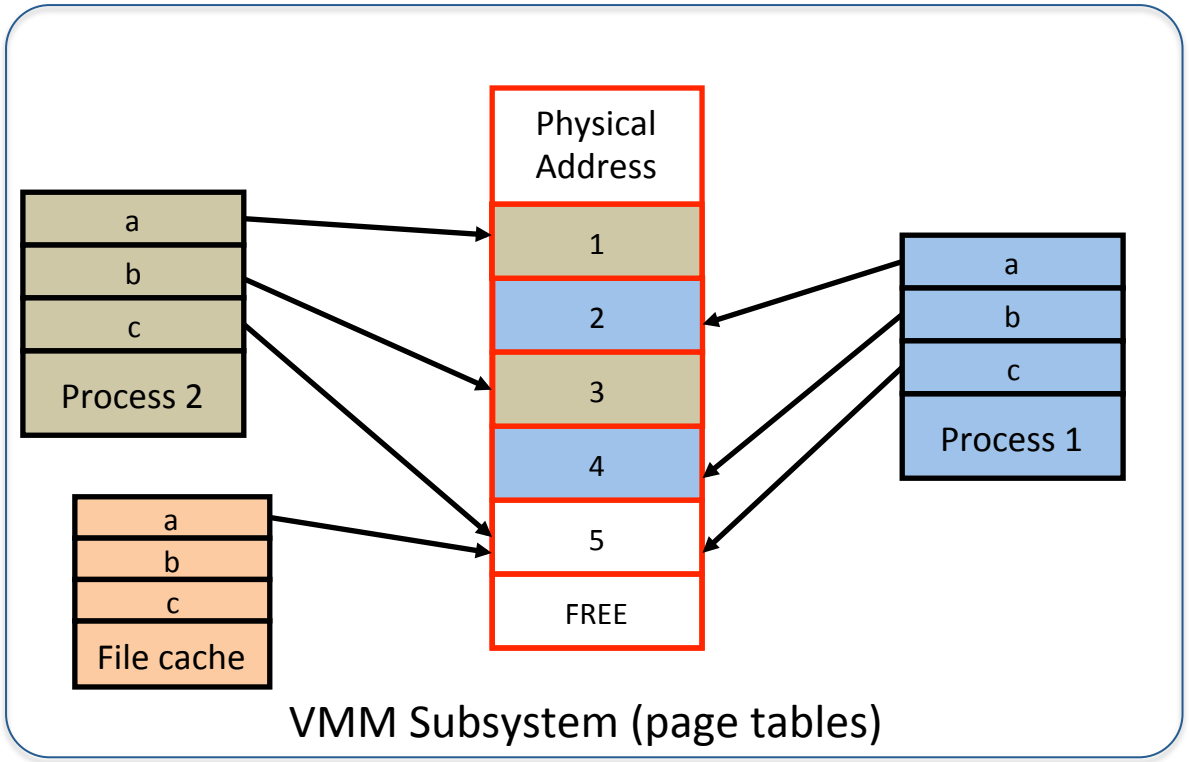
- System Calls
- Signals
- Co-operating Processes
- Shared Memory
- Message based IPC

IPC Example: UNIX Shared Memory

- int shmget(key_t key, size_t size, int shmflg);
  - Create a shared memory segment; returns ID of segment
  - key: unique key of a shared memory segment, or IPC_PRIVATE
- int shmat(int shmid, const void *addr, int flg)
  - Attach shared memory segment to address space of the calling process
  - shmid: id returned by shmget()
- int shmdt(const void *shmaddr);
  - Detach from shared memory
- Problem: synchronization! (later)

Implementing Shared Memory

- Need OS support
- Kernel maps same physical page to different address spaces



mmap Shared Memory

- mmap: map file to a processes address space
  - void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
  - "Magic memory"
  - Can share maps between processes
  - Any changes by one process are immediately reflected to other processes that have mapped
- Can create anonymous maps
  - Similar to UNIX shared memory

Outline

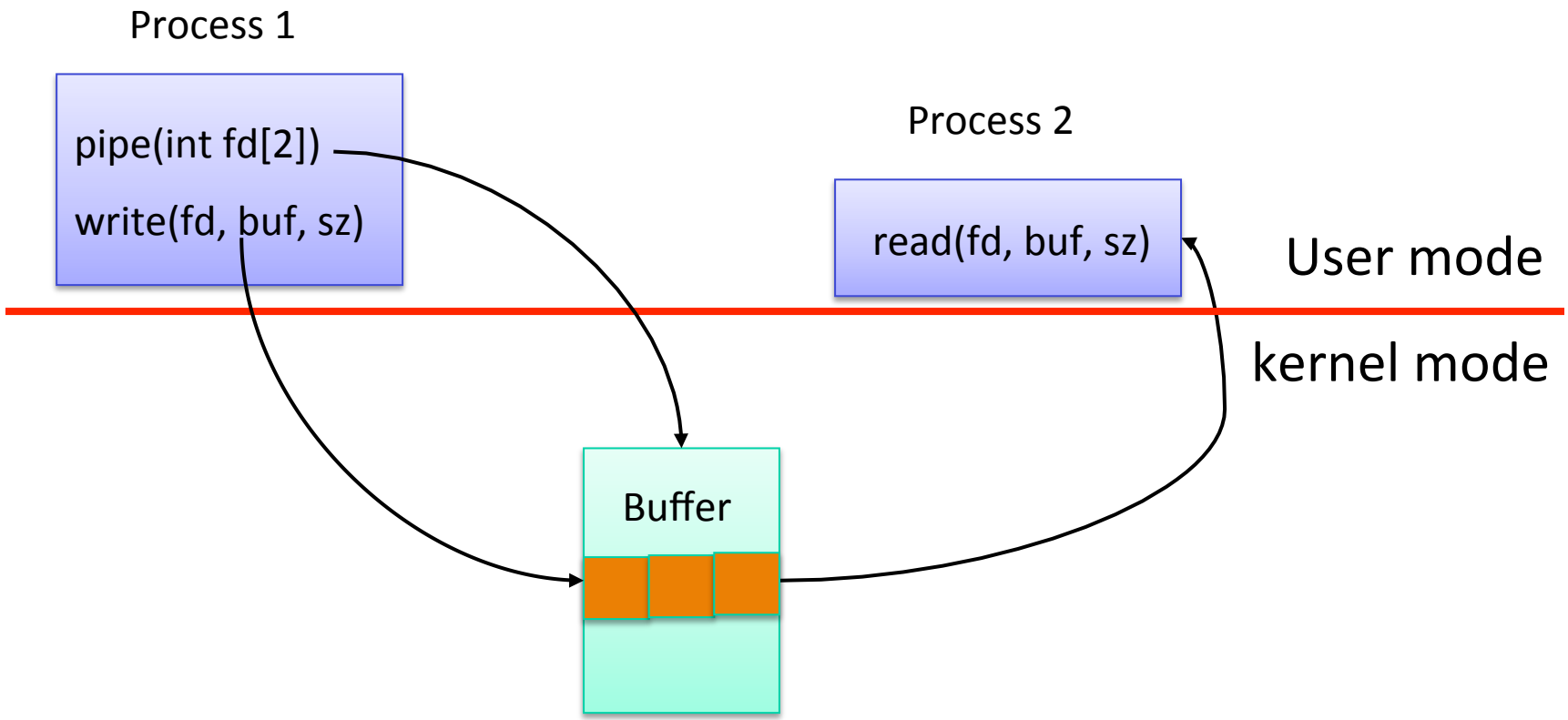
- Invoking Kernel Services
- Signals
- Co-operating Processes
- Shared Memory
- Message based IPC

Message Passing

- Explicit communication without resorting to shared variables
- Two basic operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- Processes wishing to communicate
  - establish a ***communication link*** between them
  - exchange messages via send/receive
- Implementation
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., direct or indirect, synchronous or asynchronous, automatic or explicit buffering)

Implementing Message Passing

- Data copied by kernel
- Kernel involved at every operation (unliked shared memory)



Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive message from process Q

- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional
- Operations
  - create a new mailbox
  - **send**(*A, msg*), **receive**(*A, msg*) –message to/from mailbox A
  - send and receive messages through mailbox
  - destroy a mailbox

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?

- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization

- Message passing may be either blocking or non-blocking

- **Blocking** is considered **synchronous**
  - **Blocking send:** sender blocks until the message is received
  - **Blocking receive:** receiver blocks until a message is available

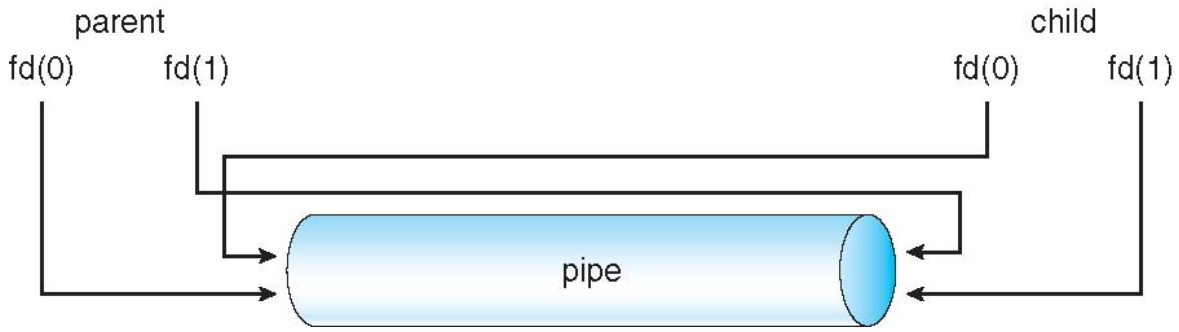
- **Non-blocking** is considered **asynchronous**
  - Sender, receiver return immediately
  - Sender may block if buffer full

Buffering

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

Ordinary Pipes

- Allow communication in standard producer-consumer style
- Producer writes to one end (the **write-end** of the pipe)
- Consumer reads from the other end (the **read-end** of the pipe)
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship



- Windows calls these **anonymous pipes**

Code example: UNIX pipe

- int pipe(int fds[2])
  - Creates a one way communication channel
  - fds[2] holds the returned two file descriptors
  - Bytes written to fds[1] will be read from fds[0]

```
int pipefd[2];
 pipe(pipefd);
    switch(pid=fork()) {
    case -1: perror("fork"); exit(1);
    case 0:  close(pipefd[0]);
             // write to fd 1
             break;
    default:  close(pipefd[1]);
             // read from fd 0
    break;
 }
```

Named Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

Using Named Pipes

- int mkfifo(const char* path, mode_t mode)
  - Creates a named pipe
  - Can read or write to it like a regular file
  - Seen as a file in the filesystem (ls, etc.)
  - Support multiple writers
  - Support multiple readers, but first to read gets the data

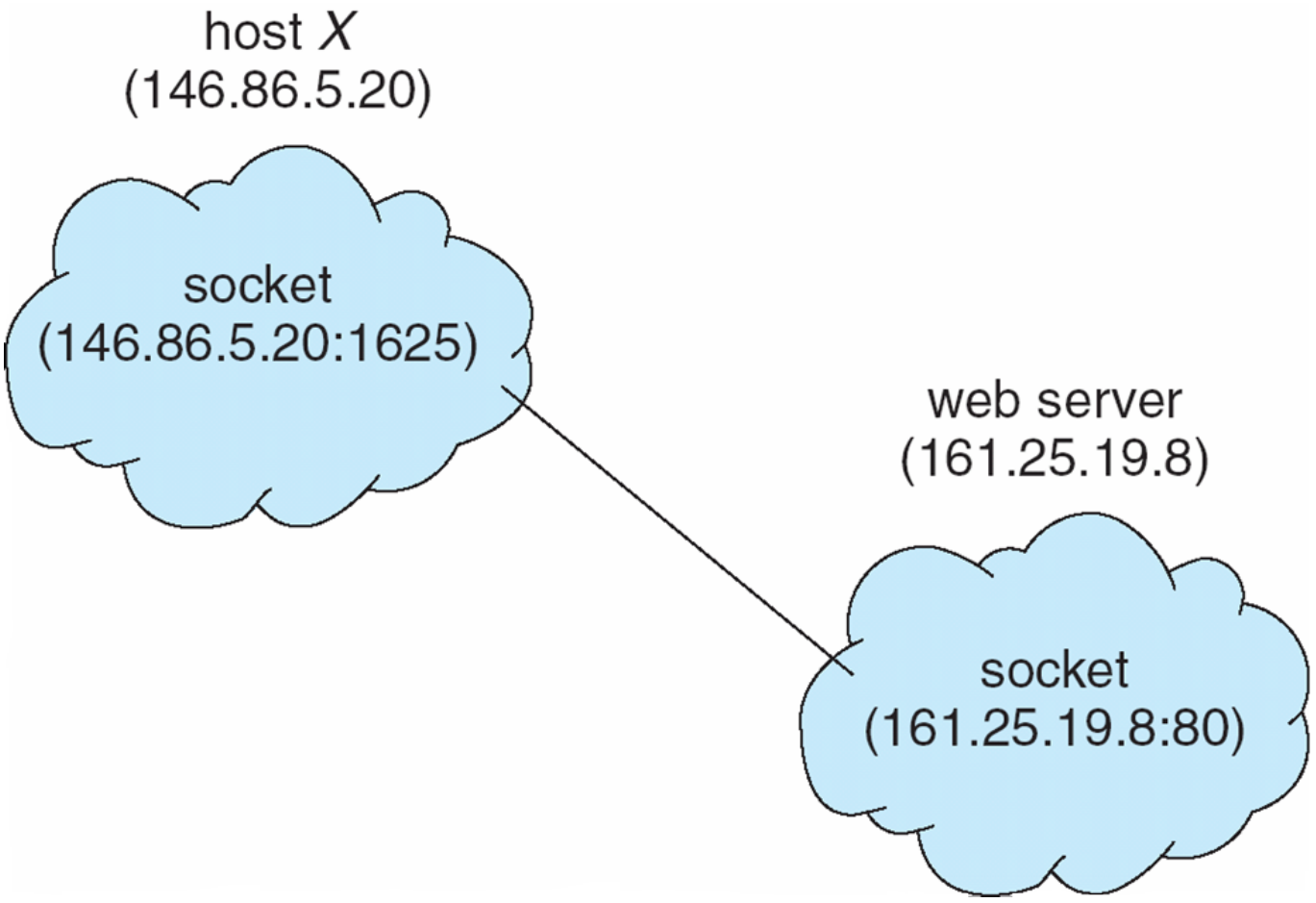
- Command line

```
$ mkfifo my_pipe
$ gzip -9 -c < my_pipe > out.gz &
$ cat file > my_pipe
$ ls -all my_pipe
$ prw------- 1 krj research 0 Feb  3 22:08 my_pipe
```

File-based IPC

- Everything as files
  - Core UNIX philosophy
- Normal Files
  - Persistent storage across reboots
- Devices (/dev/hda0)
  - Allow device I/O through reads and writes
- Named Pipes
  - IPC
- Special filesystems (/proc and /sys)
  - Communicate kernel information to userspace programs
  - Saw examples in last class (/proc/pid/maps)

Network Communication



Network Sockets

- Communication between endpoints on different machines
- A **socket** is defined as an endpoint for communication
- <IP address, **port**> – unique endpoint
- 5-tuple <src IP, src port, dst IP, dst port, protocol> designate unique communication channel
- Ports below 1024 are ***well known***, used for standard services
- Special IP address 127.0.0.1 (**loopback**) to refer to system on which process is running

Network (Berkeley) Sockets

- Code example (no error checking)

```
1.  int sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
2.  server.sin_family = AF_INET;
3.  server.sin_port = 22;
4.  server.sin_addr.s_addr = INADDR_ANY;
5.  bind(sock, (struct sockaddr *) &server, sizeof(server))) ;
6.  listen(sock, 5);
7.  for (;;) {
8.          msgsock = accept(sock, 0, 0);
9.          read(msgsock, buf, 1024);
10.         close(msgsock);
11.  }
12. close(sock);
```

UNIX Domain Sockets

- Same API as network sockets
  - Implemented as a local IPC similar to named pipes
  - Identified by pathname

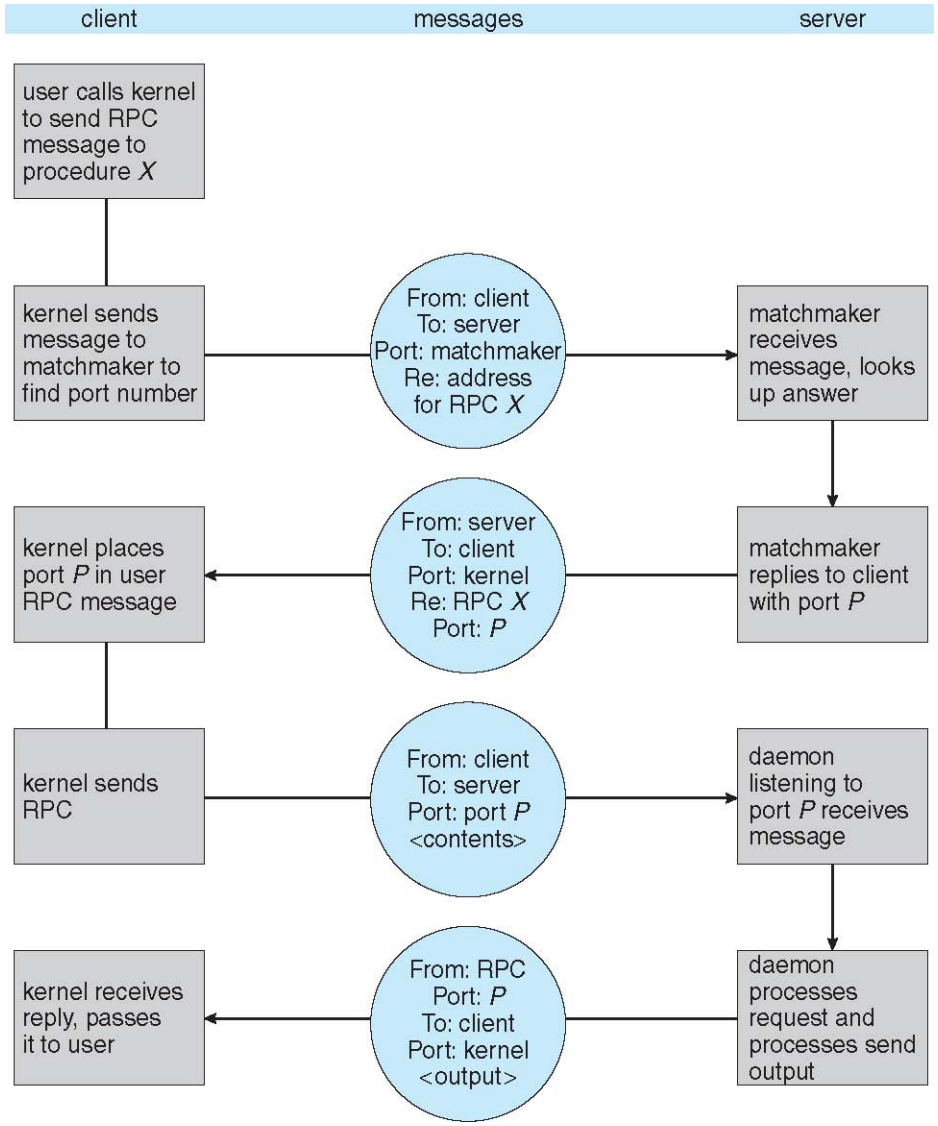
- Code example (no error checking)

```
1.  int sock = socket(AF_UNIX, SOCK_STREAM, 0);
2.  server.sun_family = AF_UNIX;
3.  strcpy(server.sun_path, "/tmp/my_socket");
4.  bind(sock, (struct sockaddr *) &server, sizeof(server))) ;
5.  listen(sock, 5);
6.  for (;;) {
7.      msgsock = accept(sock, 0, 0);
8.      read(msgsock, buf, 1024);
9.      close(msgsock);
10.  }
11. close(sock);
12. unlink("/tmp/my_socket");
```

Remote Procedure Calls

- RPC abstracts procedure calls between processes
  - Call, return, Ports for service differentiation
- **Stubs** – client-side proxy for the actual procedure on the server
  - The client stub locates server and **marshalls** the parameters
  - The server-side stub receives and unpacks the marshalled parameters, and performs the procedure on the server
- Data representation handled via **External Data Representation**
  - Account for different architectures
  - **Big-endian** and **little-endian**
- Remote communication has more failure scenarios than local
  - Deliver ***exactly once*** rather than ***at most once***
- OS provides a **matchmaker** service to connect client and server

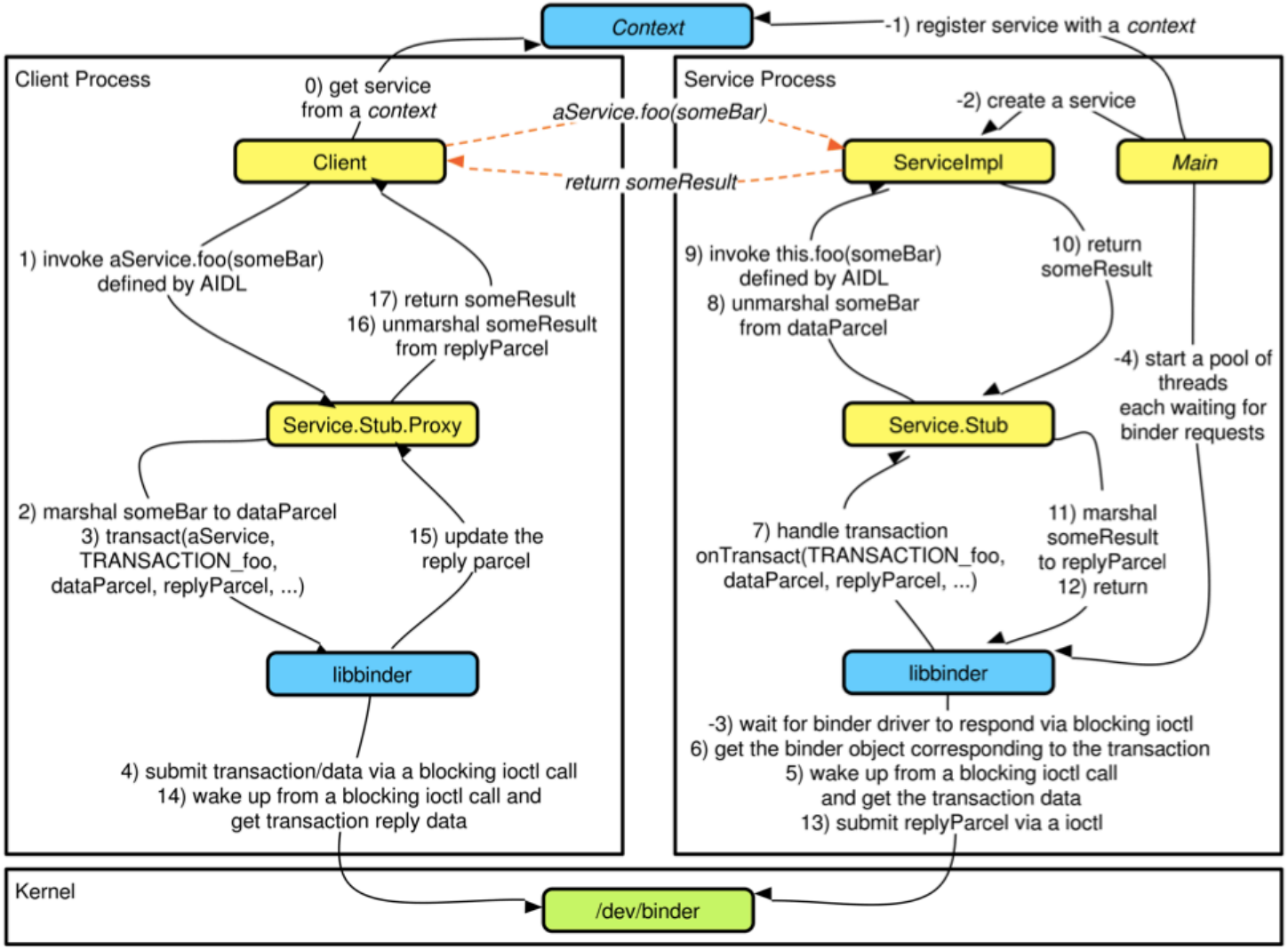
Execution of RPC



Android Binder

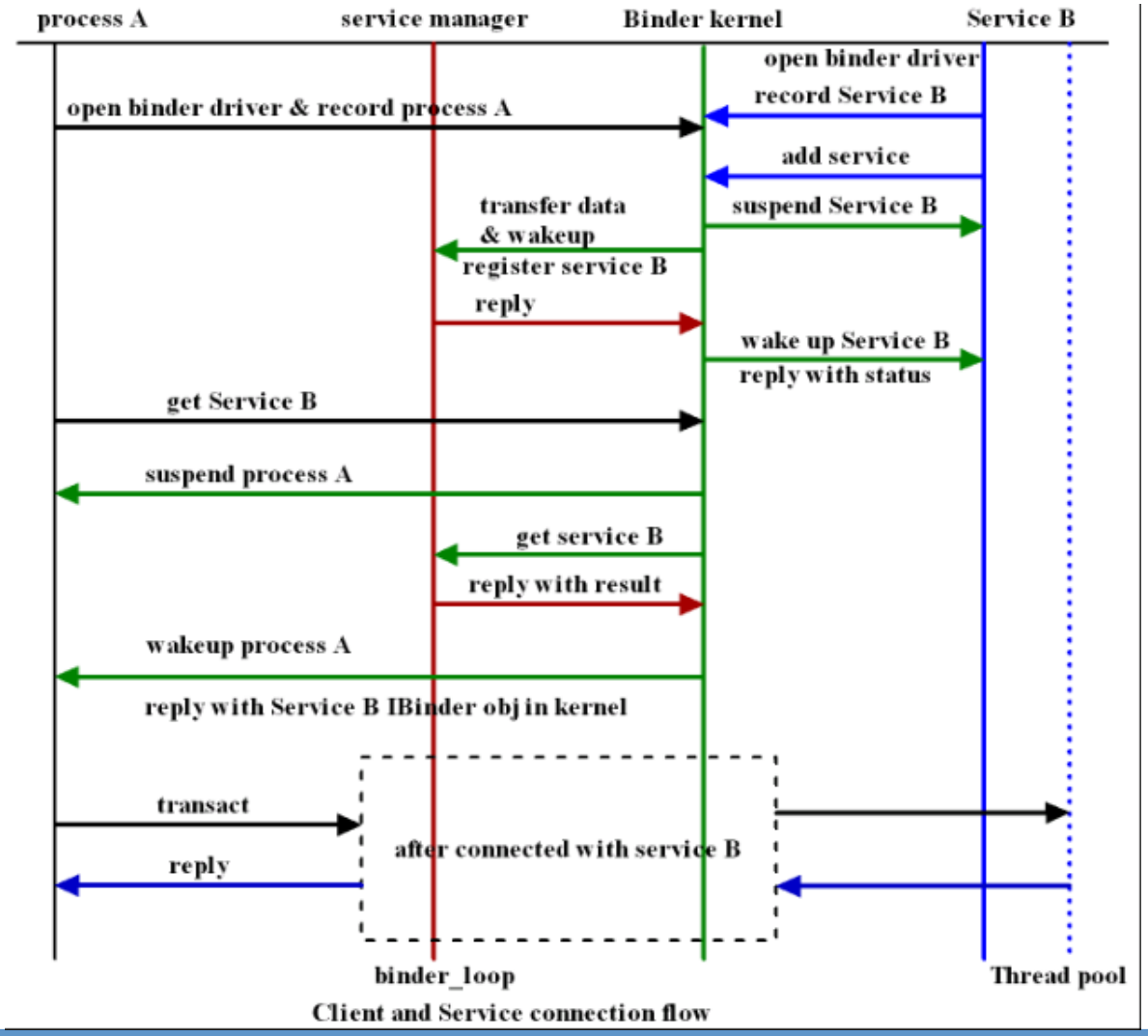
- Based on BeOS OpenBinder
  - Implemented as an I/O "device" (/dev/binder)
- Using from user-space
  - Called through Binder library from Java code
  - Apps write method definitions in AIDL
  - Automatically generate stubs for un/marshaling
- Tracking object lifetimes
  - Prevent remote objects from being deleted while being used by another process
- Naming service
  - Context manager (binder with id 0)
  - Implemented by servicemanager process
- More details
  - http://www.angryredplanet.com/~hackbod/openbinder/docs/html/BinderIPCMechanism.html
  - http://www.nds.rub.de/media/attachments/files/2012/03/binder.pdf

Android Binder Operation



Android Binder Message Flow

Source:http://lukejin.wordpress.com



Mixer

- Find your groupmates!