

# Processes

## Operating System Design – M1 Info

Instructor: Vincent Danjean  
Class Assistants: Florent Bouchez, Nicolas Fournel

September 16, 2014

# Outline

## Introduction

### User View of Processes

- Basic Unix/Linux System Call Interface

- Basic Process Management

### Kernel View of Processes

### Inter Process Communication

- Motivation

- Signals

- Shared Memory

- Bounded Buffer

- Pipes

### Inter Process Communications

- General Facts

- Sockets

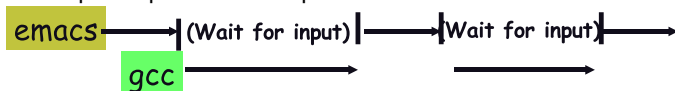
- MPI, RPC and Java RMI

# Processes

- ▶ **A process is an instance of a program running**
- ▶ **Modern OSes run multiple processes simultaneously**
- ▶ **Examples (can all run simultaneously):**
  - ▶ `gcc file_A.c` – compiler running on file A
  - ▶ `gcc file_B.c` – compiler running on file B
  - ▶ `emacs` – text editor
  - ▶ `firefox` – web browser
- ▶ **Non-examples (implemented as one process):**
  - ▶ Multiple firefox windows or emacs frames (still one process)
- ▶ **Why processes?**
  - ▶ Simplicity of programming
  - ▶ Higher throughput (better CPU utilization), lower latency

- ▶ **Multiple processes can increase CPU utilization**

- ▶ Overlap one process's computation with another's wait

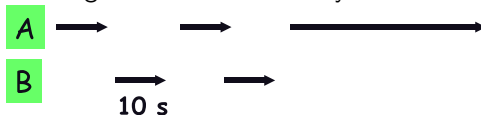


- ▶ **Multiple processes can reduce latency**

- ▶ Running *A* then *B* requires 100 sec for *B* to complete



- ▶ Running *A* and *B* concurrently makes *B* finish faster



- ▶ *A* slightly slower, but less than 100 sec unless *A* and *B* both completely CPU-bound

# Processes in the real world

- ▶ **Processes, parallelism fact of life much longer than OSES have been around**
  - ▶ E.g., say takes 1 worker 10 months to make 1 widget
  - ▶ Company may hire 100 workers to make 10,000 widgets
  - ▶ Latency for first widget  $\gg 1/10$  month
  - ▶ Throughput may be  $< 10$  widgets per month (if can't perfectly parallelize task)
  - ▶ Or  $> 10$  widgets per month if better utilization (e.g., 100 workers on 10,000 widgets never idly waiting for paint to dry)
- ▶ **You will see this with your assignments**
  - ▶ Don't expect labs to take  $1/3$  time with three people

# A process's view of the world

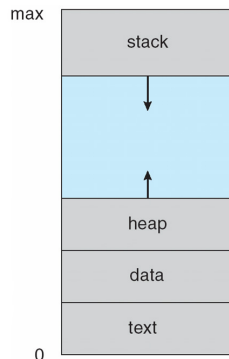
- ▶ **Each process has own view of machine**

- ▶ Its own address space
- ▶ Its own open files
- ▶ Its own virtual CPU (through preemptive multitasking)

- ▶ `*(char *)0xc000` **different in  $P_1$  &  $P_2$**

- ▶ **Greatly simplifies programming model**

- ▶ gcc does not care that firefox is running



- ▶ **Sometimes want interaction between processes**

- ▶ Simplest is through files: emacs edits file, gcc compiles it
- ▶ More complicated: Shell/command, Window manager/app.

# Outline

## Introduction

## User View of Processes

- Basic Unix/Linux System Call Interface

- Basic Process Management

## Kernel View of Processes

## Inter Process Communication

- Motivation

- Signals

- Shared Memory

- Bounded Buffer

- Pipes

- Inter Process Communications

  - General Facts

  - Sockets

  - MPI, RPC and Java RMI

# UNIX files I/O

- ▶ **Applications “open” files (or devices) by name**
  - ▶ I/O happens through open files
- ▶ `int open(char *path, int flags, /*mode*/...);`
  - ▶ flags: `O_RDONLY`, `O_WRONLY`, `O_RDWR`
  - ▶ `O_CREAT`: create the file if non-existent
  - ▶ `O_EXCL`: (w. `O_CREAT`) create if file exists already
  - ▶ `O_TRUNC`: Truncate the file
  - ▶ `O_APPEND`: Start writing from end of file
  - ▶ mode: final argument with `O_CREAT`
- ▶ **Returns file descriptor—used for all I/O to file**



# Error returns

- ▶ **What if open fails? Returns -1 (invalid fd)**
- ▶ **Most system calls return -1 on failure**
  - ▶ Specific kind of error in global int errno
- ▶ **#include <sys/errno.h> for possible values**
  - ▶ 2 = ENOENT “No such file or directory”
  - ▶ 13 = EACCES “Permission Denied”
- ▶ **perror function prints human-readable message**
  - ▶ `perror ("initfile");`  
→ “initfile: No such file or directory”

# Operations on file descriptors

- ▶ `int read (int fd, void *buf, int nbytes);`
  - ▶ Returns number of bytes read
  - ▶ Returns 0 bytes at end of file, or -1 on error
- ▶ `int write (int fd, void *buf, int nbytes);`
  - ▶ Returns number of bytes written, -1 on error
- ▶ `off_t lseek (int fd, off_t pos, int whence);`
  - ▶ whence: 0 – start, 1 – current, 2 – end
    - ▶ Returns previous file offset, or -1 on error
- ▶ `int close (int fd);`

# File descriptor numbers

- ▶ **File descriptors are inherited by processes**
  - ▶ When one process spawns another, same fds by default
- ▶ **Descriptors 0, 1, and 2 have special meaning**
  - ▶ 0 – “standard input” (`stdin` in ANSI C)
  - ▶ 1 – “standard output” (`stdout`, `printf` in ANSI C)
  - ▶ 2 – “standard error” (`stderr`, `perror` in ANSI C)
  - ▶ Normally all three attached to terminal
- ▶ **Example: `type.c`**
  - ▶ Prints the contents of a file to `stdout`

## Example: type.c

```
void typefile(char *filename)
{
    int fd, nread;
    char buf[1024];

    fd = open(filename, O_RDONLY);
    if (fd == -1) {
        perror(filename);
        return;
    }

    while ((nread = read(fd, buf, sizeof(buf))) > 0)
        write(1, buf, nread);

    close(fd);
}
```

# The rename system call

- ▶ `int rename (const char *p1, const char *p2);`
  - ▶ Changes name p2 to reference file p1
  - ▶ Removes file name p1
- ▶ **Guarantees that p2 will exist despite any crashes**
  - ▶ p2 may still be old file
  - ▶ p1 and p2 may both be new file
  - ▶ but p2 will always be old or new file
- ▶ `fsync/rename` **idiom used extensively**
  - ▶ E.g., emacs: Writes file `.#file#`
  - ▶ Calls `fsync` on file descriptor
  - ▶ `rename (".#file#", "file");`

# Creating processes

- ▶ `int fork (void);`
  - ▶ Create new process that is exact copy of current one
  - ▶ Returns *process ID* of new process in “parent”
  - ▶ Returns 0 in “child”
  - ▶ Actually, not `int` anymore, but `pid_t`
- ▶ `int getpid (void); int getppid (void);`
  - ▶ Returns *process ID* of the calling process (resp. of its parent)
- ▶ `int waitpid (int pid, int *stat, int opt);`
  - ▶ `pid` – process to wait for, or -1 for any
  - ▶ `stat` – will contain exit value, or signal
  - ▶ `opt` – usually 0 or `WNOHANG`
  - ▶ Returns process ID or -1 on error
- ▶ **Hierarchy of processes**
  - ▶ run the `pstree -p` command

# Deleting processes

- ▶ `void exit (int status);`
  - ▶ Current process ceases to exist
  - ▶ `status` shows up in `waitpid` (shifted)
  - ▶ By convention, status of 0 is success, non-zero error
- ▶ `int kill (int pid, int sig);`
  - ▶ Sends signal `sig` to process `pid`
  - ▶ `SIGTERM` most common value, kills process by default (but application can catch it for “cleanup”)
  - ▶ `SIGKILL` stronger, kills process always

# Process Termination

- ▶ **When a child terminates (either by calling `exit` or abnormally due to a fatal error or signal)**
  - ▶ An exit status is returned to the OS
  - ▶ Some of the process resources are deallocated by the operating system.
  - ▶ A `SIGCHLD` signal is sent to the parent
  - ▶ Parent should retrieve the exit status using `wait`. If it does not, then the child process will remain in the system as a **zombi**.
- ▶ **When a parent process terminates before its child, there are two options:**
  - ▶ Operating system does not allow child to continue if its parent terminates  $\leadsto$  cascading termination (VMS).
  - ▶ Re-parent the orphan (UNIX). The `init` process becomes the new parent and is specifically designed to handle orphan processes (and take care of zombies).



# Running programs

- ▶ `int execve (const char *prog, const char **argv, char **envp;)`
  - ▶ `prog` – full pathname of program to run
  - ▶ `argv` – argument vector that gets passed to main
  - ▶ `envp` – environment variables, e.g., `PATH`, `HOME`
- ▶ **Generally called through a wrapper functions**
  - ▶ `int execvp (char *prog, char **argv);`  
Search `PATH` for `prog`, use current environment
  - ▶ `int execlp (char *prog, char *arg, ...);`  
List arguments one at a time, finish with `NULL`
- ▶ **Example:** `minish.c`
  - ▶ Loop that reads a command, then executes it

## minish.c (simplified)

```
pid_t pid;
char **av;
void doexec() {
    execvp(av[0], av);
    perror(av[0]);
    exit(1);
}

/* ... main loop: */
for (;;) {
    parse_next_line_of_input(&av, stdin);

    switch (pid = fork()) {
    case -1:
        perror("fork");
        break;
    case 0:
        doexec();
    default:
        waitpid(pid, NULL, 0);
        break;
    }
}
```

# Manipulating file descriptors

- ▶ `int dup2 (int oldfd, int newfd);`
  - ▶ Makes `newfd` be the *copy* of `oldfd`, *closing* `newfd` first if necessary.
  - ▶ Two file descriptors will share same offset (`lseek` on one will affect both)
- ▶ `int fcntl (int fd, F_SETFD, val)`
  - ▶ Sets *close on exec* flag if `val = FD_CLOEXEC`, clears if `val = 0`
  - ▶ Makes file descriptor non-inheritable by spawned programs
- ▶ **Example:** `redirsh.c`
  - ▶ Loop that reads a command and executes it
  - ▶ Recognizes `command < input > output 2> errlog`

## redirsh.c

```
void doexec (void) {
    int fd;

    /* infile non-NULL if user typed "command < infile" */
    if (infile) {
        if ((fd = open (infile, O_RDONLY)) < 0) {
            perror (infile);
            exit (1);
        }
        if (fd != 0) {
            dup2 (fd, 0);
            close (fd);
        }
    }

    /* ... Do same for outfile -> fd 1, errfile -> fd 2 ... */

    execvp (av[0], av);
    perror (av[0]);
    exit (1);
}
```

# Why fork?

- ▶ **Most calls to `fork` followed by `execve`**
- ▶ **Could also combine into one `spawn` system call**
- ▶ **Occasionally useful to fork one process**
  - ▶ Unix *dump* utility backs up file system to tape
  - ▶ If tape fills up, must restart at some logical point
  - ▶ Implemented by forking to revert to old state if tape ends
- ▶ **Real win is simplicity of interface**
  - ▶ Tons of things you might want to do to child:  
Manipulate file descriptors, environment, resource limits, etc.
  - ▶ Yet `fork` requires *no* arguments at all

# Spawning process w/o fork

- ▶ Without fork, require tons of different options
- ▶ Example: Windows `CreateProcess` system call

```
BOOL CreateProcess(  
    LPCTSTR lpApplicationName, // pointer to name of executable module  
    LPTSTR lpCommandLine, // pointer to command line string  
    LPSECURITY_ATTRIBUTES lpProcessAttributes, // process security attr.  
    LPSECURITY_ATTRIBUTES lpThreadAttributes, // thread security attr.  
    BOOL blInheritHandles, // handle inheritance flag  
    DWORD dwCreationFlags, // creation flags  
    LPVOID lpEnvironment, // pointer to new environment block  
    LPCTSTR lpCurrentDirectory, // pointer to current directory name  
    LPSTARTUPINFO lpStartupInfo, // pointer to STARTUPINFO  
    LPPROCESS_INFORMATION lpProcessInformation // pointer to  
    PROCESS_INFORMATION );
```

# Outline

## Introduction

## User View of Processes

- Basic Unix/Linux System Call Interface

- Basic Process Management

## Kernel View of Processes

## Inter Process Communication

- Motivation

- Signals

- Shared Memory

- Bounded Buffer

- Pipes

- Inter Process Communications

  - General Facts

  - Sockets

  - MPI, RPC and Java RMI

# Implementing processes

- ▶ **OS keeps data structure for each proc**

- ▶ Process Control Block (PCB)
- ▶ Called `proc` in Unix, `task_struct` in Linux

- ▶ **Tracks state of the process**

- ▶ Running, runnable, blocked, etc.

- ▶ **Includes information necessary to run**

- ▶ Registers, virtual memory mappings, etc.
- ▶ Open files (including memory mapped files)

- ▶ **Various other data about the process**

- ▶ Credentials (user/group ID), signal mask, controlling terminal, priority, accounting statistics, whether being debugged, which system call binary emulation in use, ...

Process state
Process ID
User id, etc.
Program counter
Registers
Address space (VM data structs)
Open files

PCB



# Fork & Exec

## ▶ The fork system call creates a copy of the PCB

- ▶ Open files and memory mapped files are thus similar
- ▶ Open files are thus opened by both father and child. They should both close the files
- ▶ The pages of many memory segments are shared (text, r/o data,...)
- ▶ Many others are lazily copied (copy on write)

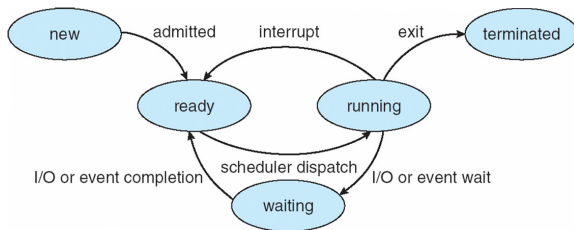
## ▶ The exec system call replaces the address space, the registers, the program counter by the one of the program to exec.

- ▶ Open files are thus inherited (hence, the need for the `fcntl` function sometimes)

Process state
Process ID
User id, etc.
Program counter
Registers
Address space (VM data structs)
Open files

PCB

# Process states




## ► Process can be in one of several states

- *new* & *terminated* at beginning & end of life
- *running* – currently executing (or will execute on kernel return)
- *ready* – can run, but kernel has chosen different process to run
- *waiting* – needs async event (e.g., disk operation) to proceed

## ► Which process should kernel run?

- if 0 runnable, run idle loop, if 1 runnable, run it
- if  $>1$  runnable, must make scheduling decision

# Scheduling

- ▶ **How to pick which process to run**
  - ▶ **Scan process table for first runnable?**
    - ▶ Expensive. Weird priorities (small pids better)
    - ▶ Divide into runnable and blocked processes
  - ▶ **FIFO?**
    - ▶ Put process on back of list, pull them off from front
- 
- The diagram illustrates a First-In-First-Out (FIFO) queue. It consists of a horizontal line with arrows at both ends. Four colored squares (blue, red, magenta, and yellow) are placed in a row along this line, connected by small horizontal arrows pointing from left to right. This represents a sequence of processes waiting to be executed in the order they arrived.
- ▶ **Priority?**
    - ▶ Give some process a better shot at the CPU

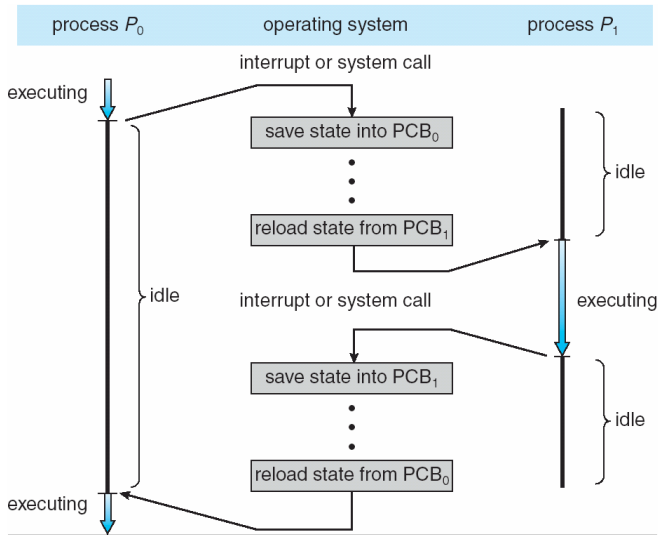
# Scheduling policy

- ▶ **Want to balance multiple goals**
  - ▶ *Fairness* – don't starve processes
  - ▶ *Priority* – reflect relative importance of procs
  - ▶ *Deadlines* – must do  $x$  (play audio) by certain time
  - ▶ *Throughput* – want good overall performance
  - ▶ *Reactivity* – minimize response time
  - ▶ *Efficiency* – minimize overhead of scheduler itself
- ▶ **No universal policy**
  - ▶ Many objectives, can't optimize for all
  - ▶ Conflicting goals (e.g., throughput or priority vs. fairness)
- ▶ **We will spend a lecture on this topic**

# Preemption

- ▶ **Can preempt a process when kernel gets control**
- ▶ **Running process can vector control to kernel**
  - ▶ System call, page fault, illegal instruction, etc.
  - ▶ May put current process to sleep—e.g., read from disk
  - ▶ May make other process runnable—e.g., fork, write to pipe
- ▶ **Periodic timer interrupt**
  - ▶ If running process used up quantum, schedule another
- ▶ **Device interrupt**
  - ▶ Disk request completed, or packet arrived on network
  - ▶ Previously waiting process becomes runnable
  - ▶ Schedule if higher priority than current running proc.
- ▶ **Changing running process is called a context switch**

# Context switch



# Context switch details

- ▶ **Very machine dependent. Typical things include:**
  - ▶ Save program counter and integer registers (always)
  - ▶ Save floating point or other special registers
  - ▶ Save condition codes
  - ▶ Change virtual address translations
- ▶ **Non-negligible cost**
  - ▶ Save/restore floating point registers expensive
    - ▶ Optimization: only save if process used floating point
  - ▶ May require flushing TLB (memory translation hardware)
    - ▶ Optimization: don't flush kernel's own data from TLB
  - ▶ Usually causes more cache misses (switch working sets)

# Outline

## Introduction

## User View of Processes

- Basic Unix/Linux System Call Interface

- Basic Process Management

## Kernel View of Processes

## Inter Process Communication

- Motivation

- Signals

- Shared Memory

- Bounded Buffer

- Pipes

- Inter Process Communications

  - General Facts

  - Sockets

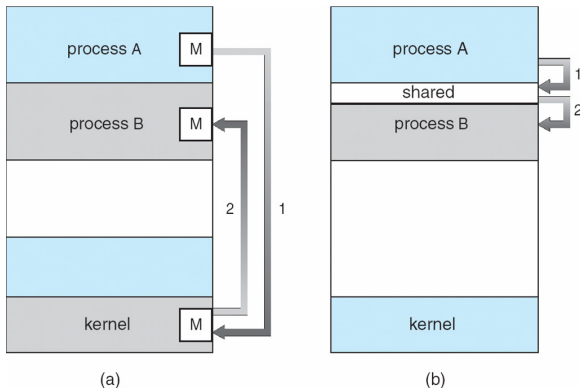
  - MPI, RPC and Java RMI



# Cooperating Processes

- ▶ **Independent process cannot affect or be affected by the execution of another process**
  - ▶ See next lectures on memory management
- ▶ **Cooperating process can affect or be affected by the execution of another process. Advantages:**
  - ▶ Information sharing
  - ▶ Computation speed-up
  - ▶ Modularity
  - ▶ Convenience

# Process Interaction



## ► How can processes interact in real time?

- (1) Through files but it's not really “real time”.
- (2) Through asynchronous signals or alerts but again, it's not really “real time”.
- (3) By sharing a region of physical memory
- (4) By passing messages through the kernel

# Asynchronous notification

## ► As we have seen earlier

- Children process send a SIGCHLD signal to their parents upon termination.
- One may send a SIGINT/SIGTERM/SIGSTOP/SIGKILL signal to CTRL-C/suspend (CTRL-Z)/terminate/kill a process using the kill function:

```
int kill (int pid, int sig);
```

- Upon reception of a signal, a given handler is called. This handler can be obtained and modified using the signal function:

```
typedef void (*sighandler_t)(int);
```

```
sighandler_t signal(int signum, sighandler_t handler);
```

## ► Some common signals:

- SIGSEGV (segfault), SIGFPE (floating-point exception), SIGALRM (timer alarm), SIGABRT (abort is caught by gdb), SIGILL (illegal instruction!), SIGCONT (resume if suspended)
- SIGUSR1, SIGUSR2

## ► Some signals cannot be blocked (SIGSTOP and SIGKILL)

# Illustrating shm

```
#define DELAY 1      /* secondss */

void handler(int signal_num)
{
    printf("Signal %d => ", signal_num);
    switch (signal_num) {
        case SIGTSTP:
            printf("Let's sleep!");
            kill(getpid(), SIGSTOP);
            printf("Waking up!");
            signal(SIGTSTP, handler);
            break;
        case SIGINT:
        case SIGTERM:
            printf("End of the program");
            exit(EXIT_SUCCESS);
            break;
    }
}
```

```
#include <stdlib.h>
#include <signal.h>
#include <errno.h>
#include <unistd.h>
#include <stdio.h>

int main(void)
{
    signal(SIGTSTP, handler);
    /* if control-Z */
    signal(SIGINT, handler);
    /* if control-C */
    signal(SIGTERM, handler);
    /* if kill processus */
    while (1) {
        sleep(DELAY);
        printf(".");
        fflush(stdout);
    }
    printf("fin");
    exit(EXIT_SUCCESS);
}
```

# Shared Memory Segment

- ▶ **A process can create a shared memory segment using:**

```
int shmget(key_t key, size_t size, int shmflg);
```

- ▶ The returned value identifies the segment and is called the `shmid`
- ▶ The key is used so that process indeed get the same segment.
- ▶ **The original owner of a shared memory segment can assign ownership to another user with `shmctl()`.**

- ▶ It can also revoke this assignment.

- ▶ **Once created, a shared segment should be attached to a process address space using**

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

- ▶ **It can be detached using `int shmdt(const void *shmaddr);`**
- ▶ **Can also be done with the `mmap` function**

# Illustrating shm

```
char c;
int shmid;
key_t key;
char *shm, *s;

key = 5678;

/* Create the segment */
if ((shmid = shmget(key, SHMSZ,
    IPC_CREAT | 0666)) < 0) {
    perror("shmget");
    exit(1);
}

/* Attach the segment */
if ((shm = shmat(shmid, NULL, 0)) ==
    (char *) -1) {
    perror("shmat");
    exit(1);
}
```

```
int shmid;
key_t key;
char *shm, *s;

key = 5678;

/* Locate the segment */
if ((shmid = shmget(key, SHMSZ, 0666))
    < 0) {
    perror("shmget");
    exit(1);
}

/* Attach the segment */
if ((shm = shmat(shmid, NULL, 0)) ==
    (char *) -1) {
    perror("shmat");
    exit(1);
}
```

# Producer-Consumer Problem

## ► Paradigm for cooperating processes

- Producer process produces information that is consumed by a consumer process
- unbounded-buffer places no practical limit on the size of the buffer
- bounded-buffer assumes that there is a fixed buffer size

## ► Shared-Memory Solution

```
/*Shared data structure*/
```

```
#define BUFFER_SIZE 10
typedef struct {
...
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

```
/*      Producer      */
```

```
item nextProduced;
while (1) {
    while(((in + 1)%BUFFER_SIZE)
           ==out)
        ; /* do nothing */
    buffer[in]=nextProduced;
    in=(in+1)%BUFFER_SIZE;
}
```

```
/*      Consumer      */
```

```
item nextConsumed;
while (1) {
    while (in == out)
        ; /* do nothing */
    nextConsumed=buffer[out];
    out=(out+1)%BUFFER_SIZE;
}
```

## ► Drawbacks:

- Solution is correct, but can only use BUFFER\_SIZE-1 elements
- Works only with one producer and one consumer
- Busy waiting

# Pipes

- ▶ `int pipe (int fds[2]);`
  - ▶ Returns two file descriptors in `fds[0]` and `fds[1]`
  - ▶ Writes to `fds[1]` will be read on `fds[0]`
  - ▶ When last copy of `fds[1]` closed, `fds[0]` will return EOF
  - ▶ Returns 0 on success, -1 on error
- ▶ **Operations on pipes**
  - ▶ `read/write/close` – as with files
  - ▶ When `fds[1]` closed, `read(fds[0])` returns 0 bytes
  - ▶ When `fds[0]` closed, `write(fds[1])`:
    - ▶ Kills process with SIGPIPE, or if blocked
    - ▶ Fails with EPIPE
- ▶ **Example: `pipesh.c`**
  - ▶ Sets up pipeline `command1 | command2 | command3 ...`



## pipesh.c (simplified)

```
void doexec (void) {
    int pipefds[2];
    while (outcmd) {
        pipe (pipefds);
        switch (fork ()) {
            case -1:
                perror ("fork"); exit (1);
            case 0:
                dup2 (pipefds[1], 1);
                close (pipefds[0]); close (pipefds[1]);
                outcmd = NULL;
                break;
            default:
                dup2 (pipefds[0], 0);
                close (pipefds[0]); close (pipefds[1]);
                parse_command_line (&av, &outcmd, outcmd);
                break;
        }
    }
}
/* ... */
}
```

# Inter Process Communications (IPC)

- ▶ **Mechanism for processes to communicate and to synchronize their actions**
- ▶ **Message system** processes communicate with each other **without resorting to shared variables**
- ▶ **IPC facility provides two operations:**
  - ▶ `send(message)` message size fixed or variable
  - ▶ `receive(message)`
- ▶ **If P and Q wish to communicate, they need to:**
  - ▶ establish a communication link between them
  - ▶ exchange messages via `send/receive`
- ▶ **Implementation of communication link**
  - ▶ physical (e.g., shared memory, hardware bus)
  - ▶ logical (e.g., logical properties)

# Implementation Issues

- ▶ 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 a 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 (also referred to as 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 and receive messages through mailbox
  - ▶ destroy a mailbox
- ▶ **Primitives are defined as:**
  - ▶ `send(A, message)` send a message to mailbox A
  - ▶ `receive(A, message)` receive a message from mailbox A

# Indirect Communication Issues

## ► Mailbox sharing

- P1, P2, and P3 share mailbox A
- P1, sends; P2 and P3 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 Issues

## ► Synchronization

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
- Non-blocking is considered asynchronous
- send and receive primitives may be either blocking or non-blocking

## ► Queue of messages attached to the link; implemented in one of three ways.

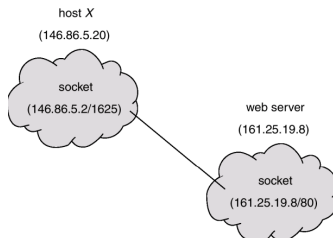
- Zero capacity 0 messages. Sender must wait for receiver (*rendezvous*)
- Bounded capacity finite length of n messages. Sender must wait if link full
- Unbounded capacity infinite length. Sender never waits

## ► Pipes, just like most I/Os are buffered

- Hence, when you pipe process, the initial producer process will “wait” for the child process to read the data

# Sockets

- ▶ A socket is defined as an endpoint for communication
- ▶ Concatenation of IP address and port
- ▶ The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- ▶ Communication consists between a pair of sockets and is bidirectionnal



```
(int) sock_id = socket(int domain, int type, int protocol);
(int) error bind(int sock_id,sockaddr localaddr, int addrlen);
(int) error = listen(int sock_id,int backlog);
(int) new_sock_id = accept(int sock_id, struct sockaddr *client_addr,
                          int * client_addrlen);
(int) error = connect(int sock_id, struct sockaddr *server_addr,
                     int * server_addrlen);
ssize_t recv(int sock_id,char * buffer,int len,int flags);
ssize_t send(int sock_id,char * buffer,int len,int flags);
(int) error = close(int sock_id);
```



# Sockets

To accept connections, the following steps are performed:

1. A socket is created with **socket**
2. The socket is bound to a local address using **bind** (*assigning a name to a socket*), so that other sockets may be **connected** to it
3. A willingness to accept incoming connections and a queue limit for incoming connections are specified with **listen**.
4. Connections are accepted with **accept**.

```
(int) sock_id = socket(int domain, int type, int protocol);
(int) error bind(int sock_id,sockaddr localaddr, int addrlen);
(int) error = listen(int sock_id,int backlog);
(int) new_sock_id = accept(int sock_id, struct sockaddr *client_addr,
                          int * client_addrlen);
(int) error = connect(int sock_id, struct sockaddr *server_addr,
                     int * server_addrlen);
ssize_t recv(int sock_id,char * buffer,int len,int flags);
ssize_t send(int sock_id,char * buffer,int len,int flags);
(int) error = close(int sock_id);
```

# Higher level APIs

- ▶ **Message Passing Interface (MPI)**
  - ▶ Used for High Performance Computing with high-speed network implementations
  - ▶ Proposes send/recv but many others (Isend/Irecv, collective operations)
  - ▶ Uses structured types instead of char \* (for portability)
- ▶ **Remote procedure call (RPC) abstracts procedure calls between processes on networked systems**
  - ▶ Stubs client-side proxy for the actual procedure on the server
  - ▶ The client-side stub locates the server and **marshalls** the parameters
  - ▶ The server-side stub receives this message, unpacks the marshalled parameters, and performs the procedure on the server
- ▶ **Remote Method Invocation**
  - ▶ Remote Method Invocation (RMI) is a Java mechanism similar to RPCs
  - ▶ RMI allows a Java program on one machine to invoke a method on a remote object