

# CSE 344 System Programming

---

Süleyman Gölböl  
1801042656

## Introduction, POSIX and fundamental concepts

Spring 2020-2021

Erchan Aptoula

Institute of Information Technologies

Office : 253

# Introduction

---

## What is this course about ?

- The aim of the course is to make the students familiar with system related calls of a POSIX operating system
- What do “system calls” do ?
  - Talk to the Operating System (O/S)
  - Access Hardware
  - Perform low level programming

## So in this course we will

- Look at the Standard C Library and how we can use it
- Look at Standards for O/S level programming
- Look at how the theory of O/Ss is applied to the practice of actual systems programming

# Programming Interface

---

*Programming interfaces* allow applications to perform tasks such as :

- File I/O, creating and deleting files and directories, creating new processes, executing programs, setting timers, communicating between processes and threads on the same computer, and communicating between processes residing on different computers connected via a network.

This set of low-level interfaces is also known as the *system programming interface*

# The Core of the Operating System: The Kernel

---

The term *operating system* is commonly used with two different meanings:

- To denote the entire package consisting of the **central software managing a computer's resources** and all of the accompanying **standard software tools**, such as **command-line interpreters, graphical user interfaces, file utilities, and editors**.
- More narrowly, to refer to the central software that manages and allocates computer resources (i.e., the CPU, RAM, and devices).

The term *kernel* is often used as a synonym for the second meaning.

# Tasks performed by the kernel

---

Among other things, the kernel performs the following tasks :

- Process scheduling
- Memory management
- Provision of a file system
- Creation and termination of processes
- Access to devices
- Networking
- Provision of an application programming interface (API) through system calls.

# Kernel Tasks: Process Scheduling

---

A computer has one or more central processing units (CPUs), which execute the instructions of programs.

Most UNIX systems, are *preemptive multitasking* operating systems

- **Multitasking** means multiple processes (i.e., running programs) can simultaneously reside in memory and each may receive use of the CPU(s).
- **Preemptive** means the rules governing which processes receive use of the CPU and for how long are determined by the kernel process scheduler (rather than by the processes themselves)

# Kernel Tasks: Memory Management

**Physical memory** (RAM) is a limited resource that the kernel must share among processes in an equitable and efficient fashion.

Most modern operating systems, employ **virtual memory management**, a technique that confers two main advantages:

- ✗ Processes are **isolated from one another and from the kernel**, so that one process can't read or modify the memory of another process or the kernel.
- ✗ Only **part of a process needs to be kept in memory**, thereby lowering the memory requirements of each process and allowing more processes to be held in RAM simultaneously. This leads to better CPU utilization, since it increases the likelihood that, at any moment in time, there is at least one process that the CPU(s) can execute.

# Kernel Tasks

---

## *Provision of a file system:*

The kernel provides a file system on disk, allowing files to be created, retrieved, updated, deleted, and so on.

## *Creation and termination of processes:*

The kernel can load a new program into memory, providing it with the resources (e.g., CPU, memory, and access to files) in order to run it. Such an instance of a **running program is termed as a “process”**.

Once a process has completed execution, the kernel ensures that the resources it uses are freed for subsequent reuse by later programs.

# Kernel Tasks

---

## ***Accessing to devices:***

The devices attached to a computer allow communication of information between the computer and the outside world, permitting input / output.

The kernel provides programs with an **interface that standardizes and simplifies access to devices**, while at the same time arbitrating access by multiple processes to each device.

## ***Networking:***

The kernel transmits and receives network messages (packets) on behalf of user processes. This task includes routing of network packets to the target system.

# Kernel Tasks

---

- Multiuser operating systems provide users with the abstraction of a virtual private computer; that is, each user can log on to the system and operate largely independently of other users.
- Users can run programs, (each gets a share of the CPU) and operates in its own virtual address space,
- All programs can independently access devices and transfer information over the network.
- The kernel resolves potential conflicts in accessing hardware resources, so users and processes are generally unaware of the conflicts.

# Kernel mode and the user mode

---

- Modern processor architectures typically allows the CPU to operate in at least two different modes: ***user mode* and *kernel mode***. Hardware instructions allow switching from one mode to the other.
- Correspondingly, **areas of virtual memory can be marked as being part of user space or kernel space**. When running in user mode, the CPU can access only memory that is marked as being in user space; attempts to access memory in kernel space result in a hardware exception. When running **in kernel mode, the CPU can access both user and kernel memory space**.
- Certain operations can be performed only while the processor is operating in kernel mode.

# Process versus kernel views of the system

---

- A running system typically has numerous processes. For a process, many things happen asynchronously. An executing process **doesn't know when it will next time out**, which other processes will then be scheduled for the CPU (and in what order), or when it will next be scheduled. The delivery of *signals* and the occurrence of *interprocess communication* events are mediated by the kernel, and can occur at any time for a process.
- Many things happen transparently for a process. A process doesn't know **where it is located in RAM** or, whether a particular part of its memory space is currently resident in memory or held in the swap area. Similarly, a process doesn't know where on the disk drive the files it accesses are being held; it simply refers to the files by name.

# Process versus kernel views of the system

- A process operates in isolation; it **can't directly communicate with another process**.
- A process **can't itself create a new process** or even end its own existence.
- A process can't communicate directly with the input and output devices attached to the computer.
- The **kernel decides which process will next obtain access to the CPU**, when it will do so, and for how long.



# Process versus kernel views of the system

---

- The **kernel maintains data structures containing information about all running processes** and updates these structures as processes are created, change state, and terminated.
- The **kernel maintains all of the low-level data structures** that enable the file names used by programs to be translated into physical locations on the disk.
- The **kernel also maintains data structures that map the virtual memory of each process into the physical memory** of the computer and the swap area(s) on disk.
- **All communication** between processes is done via mechanisms provided by the kernel.

# The Shell

---

- A **shell** is a special-purpose program designed to read commands typed by a user and execute appropriate programs in response to those commands.
- Such a program is also known as a ***command interpreter***
- The term *login shell* is used to denote the process that is created to run a shell when the user first logs in.
- On some operating systems the command interpreter is an integrated part of the kernel, **on UNIX systems, the shell is a user process**. Many different shells exist, and different users on the same computer can simultaneously use different shells.

# Users and Groups

Each user on the system is uniquely identified, and users may belong to groups

## Users

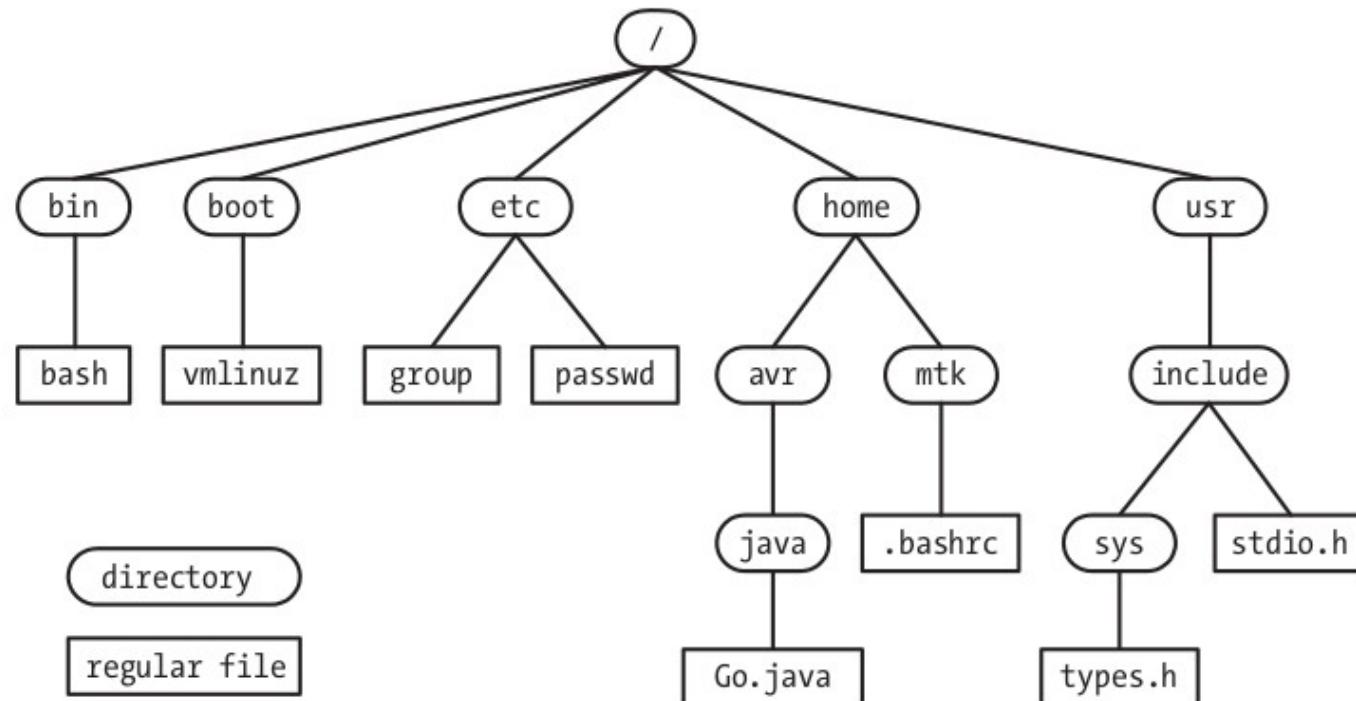
- Every user of the system has a unique login name (*username*) and a corresponding numeric user ID (UID).

## Groups

- For administrative purposes (for controlling access to files and other system resources) it is useful to organize users into groups. Each group is identified by its *group name*, *Group ID* (GID) and *user list*

# Single Directory Hierarchy, Directories, Links, and Files

- The kernel maintains a hierarchical directory structure to organize all files in the system where each disk device has its own directory hierarchy.



# Single Directory Hierarchy, Directories, Links, and Files

- At the base of this hierarchy is the **root directory**, named / (slash). All files and directories are children or further removed descendants of the root directory
- Within the file system, **each file is marked with a *type***, indicating what kind of file it is.
- File types include **devices, pipes, sockets, directories, and symbolic links**.
- A *directory* is a special file whose contents take the form of a table of filenames coupled with references to the corresponding files.

# Single Directory Hierarchy, Directories, Links, and Files

- The filename-plus-reference association is called a *link*, and files may have multiple links, and thus multiple names, in the same or in different directories
- Every directory contains at least two entries:
  - . (dot), which is a link to the directory itself, and
  - .. (dot-dot), which is a link to its parent directory, the directory above it in the hierarchy (except the root directory)
- For the root directory, the dot-dot entry is a link to the root directory itself (thus, `../` equates to `/`).

# Directories, Links, and Files

---

- A *pathname* is a string consisting of an optional initial slash (/) followed by a series of filenames separated by slashes.
- All but the last of these component filenames identifies a directory
- A pathname is read from left to right; each filename resides in the directory specified by the preceding part of the pathname.
- An *absolute pathname* begins with a slash ( / ) and specifies the location of a file with respect to the root directory.
- A *relative pathname* specifies the location of a file relative to a process's *current working directory*

# File I/O

---

- One of the distinguishing features of the I/O model on UNIX systems is the concept of *universality of I/O*.
- The **same system calls** (`open()`, `read()`, `write()`, `close()`, ...) are used to perform I/O on all types of files, including devices.
- Many applications and libraries interpret the *newline* character (ASCII code 10 decimal) as terminating one line of text and commencing another.

# File Descriptors

- The I/O system calls refer to open files using a *file descriptor*, a non-negative integer. A file descriptor is typically obtained by a call to *open()*, which takes a pathname argument specifying a file upon which I/O is to be performed.
- A process inherits three open file descriptors when it is started by the shell:
  - **descriptor 0 is standard input**, the file from which the process takes its input;
  - **descriptor 1 is standard output**, the file to which the process writes its output;
  - and **descriptor 2 is standard error**, the file to which the process writes error messages and notification of exceptional or abnormal conditions

# Processes

- When the process terminates, all such resources are released for reuse by other processes. Other resources, such as the CPU and network bandwidth, are renewable, but must be shared equitably among all processes.
- A process is logically divided into the following parts, known as segments:
  - *Text*: the instructions of the program.
  - *Data*: the static variables used by the program.
  - *Heap*: an area from which programs can dynamically allocate extra memory.
  - *Stack*: a piece of memory that grows and shrinks as functions are called and return. Also used to allocate storage for local variables and function call linkage information.

# Processes Creation and Program Execution

- A process can create a new process using the *fork()* system call. The process that calls *fork()* is referred to as the parent process, and the new process is referred to as the child process. The kernel creates the child process by making a duplicate of the parent process.
- The child inherits copies of the parent's data, stack, and heap segments, which it may then modify independently of the parent's copies
- The child process goes on either to execute a different set of functions in the same code as the parent, or, frequently, to use the *execve()* system call to load and execute an entirely new program. An *execve()* call destroys the existing text, data, stack, and heap segments, replacing them with new segments based on the code of the new program

# Processes Creation and Program Execution

---

## Process ID and parent process ID

- Each process has a unique integer process identifier (PID). Each process also has a parent process identifier (PPID) attribute, which identifies the process that requested the kernel to create this process. (PIDs wrap when maxed)

## Process termination and termination status

- A process can terminate in one of two ways: by requesting its own termination using the `_exit()` system call, or by being killed by the delivery of a signal.
- The process yields a termination status, a small nonnegative integer value that is available for inspection by the parent process using the `wait()` system call.

# Deamon Processes

---

A **daemon** is a special-purpose process that is created and handled by the system in the same way as other processes, but is distinguished by the following characteristics:

- **It is long-lived.** A daemon process is often started at system boot and remains in existence until the system is shut down
- **It runs in the background,** and has no controlling terminal from which it can read input or to which it can write output.
- Examples of daemon processes include *syslogd*, which records messages in the system log, and *httpd*, which serves web pages via the Hypertext Transfer Protocol

# Environment List

**Each process has an environment list**, which is a set of environment variables that are maintained within the user-space memory of the process.

- Each element of this list consists of a name and an associated value. When a new process is created via *fork()*, it inherits a copy of its parent's environment. Thus, the environment provides a mechanism for a parent process to communicate information to a child process.
- When a process replaces the program that it is running using *exec()*, the new program either inherits the environment used by the old program or receives a new environment specified as part of the *exec()* call

# Memory Mapping

The ***mmap()*** system call creates a new *memory mapping* in the calling process's virtual address space.

- Mappings fall into two categories:
  - A *file mapping* maps a region of a file into the calling process's virtual memory. Once mapped, the file's contents can be accessed by operations on the bytes in the corresponding memory region. The pages of the mapping are automatically loaded from the file as required.
  - By contrast, an *anonymous mapping* doesn't have a corresponding file. Instead, the pages of the mapping are initialized to 0

# Memory Mapping

**The memory in one process's mapping may be shared with mappings in other processes.**

This can occur either because:

- two processes map the same region of a file
- or because a child process created by *fork()* inherits a mapping from its parent.

Memory mappings serve a variety of purposes, including initialization of a process's text segment from the corresponding segment of an executable file, allocation of new (zero-filled) memory, file I/O (memory-mapped I/O), and inter process communication (via a shared mapping)

# Static and Shared Libraries

---

- An *object library* is a file containing the compiled object code for a (usually logically related) set of functions that may be called from application programs
- Placing code for a set of functions in a single object library eases the tasks of program creation and maintenance
- Modern UNIX systems provide two types of object libraries *static libraries* and *shared libraries*.

# Static Libraries

---

- **Static libraries** were the only type of library on early UNIX systems. A static library is essentially **a structured bundle of compiled object modules**.
- To use functions from a static library, we specify that library in the link command used to build a program. After resolving the various function references from the main program to the modules in the static library, **the linker extracts copies of the required object modules from the library and copies these into the resulting executable file**.

# Static Libraries

---

- The fact that each statically linked program includes its **own copy** of the object modules required from the library creates a number of **disadvantages**.
- One is the **duplication** of object code in different executable files wastes disk space. A corresponding waste of memory occurs when statically linked programs using the same library function are executed at the same time; each program requires its own copy of the function to reside in memory.
- Additionally, if a library function requires **modification**, then, after recompiling that function and adding it to the static library, all applications that need to use the updated function must be **relinked** against the library.

# Shared Libraries

---

**Shared libraries** were designed to address the problems with static libraries.

- If a program is linked against a shared library, then, instead of copying object modules from the library into the executable, the linker just writes a record into the executable to indicate that at run time the executable needs to use that shared library.
- When the executable is loaded into memory at run time, a program called the **dynamic linker ensures** that the shared libraries required by the executable are **found** and **loaded** into memory, and performs **run-time linking** to resolve the function calls in the executable to the corresponding definitions in the shared libraries.

# IPC and Synchronization

---

A running operating system consists of numerous processes, many of which operate independently of each other.

- Some processes, however, cooperate to achieve their intended purposes, and these processes **need methods of communicating** with one another and **synchronizing** their actions.
- One way for processes to communicate is by reading and **writing information in disk files**. However, for many applications, this is too slow and inflexible

# Interprocess Communication (IPC)

Modern UNIX implementations provides a rich set of mechanisms for interprocess communication, including the following:

- *signals* (to indicate that an error has occurred)
- *pipes* and *FIFOS* (to transfer data between processes)
- *sockets* (to transfer data between processes on the same or even on different computers)
- *file locking* (so a process can lock a region of the file and prevents others not to use it)
- *message queues* (exchange packets of data between processes)
- *semaphores* (to synchronize the actions of processes)
- *shared memory* (so that different processes can share the same memory page)

# Signals

---

- Signals are often described as “**software interrupts**.” The arrival of a signal informs a process that some event or exceptional condition has occurred.
- There are various types of signals, each of which identifies a different event or condition. Each signal type is identified by a different integer, defined with symbolic names of the form **SIGxxxx**
- Signals are sent to a process **by the kernel, by another process** (with suitable permissions), or **by the process itself**
- Within the shell, the ***kill* command** can be used to send a signal to a process.
- The ***kill()* system call** provides the same facility within programs

# Signals

- When a process receives a signal, it takes one of the following actions, depending on the signal:
  - it **ignores** the signal;
  - it is **killed** by the signal;
  - it is **suspended** until later being resumed by receipt of a special-purpose signal
- For most signal types, instead of accepting the default signal action, a program can choose to ignore the signal or to establish a *signal handler*.
- A signal handler is a programmer-defined function that is automatically invoked when the signal is delivered to the process. This function performs some action appropriate to the condition that generated the signal

# Threads

---

- In most modern operating systems each process can have **multiple *threads* of execution**.
- One way of envisaging threads is as a set of processes that share the same virtual memory.
- Each thread is executing the same program code and **shares the same data area and heap**. However, **each thread has its own stack** containing local variables and function call linkage information
- Threads can communicate with each other **via the global variables that they share**. The threading API provides ***condition variables* and *mutexes***, which are primitives that enable the threads of a process to communicate and synchronize their actions, in particular, their use of shared variables

# Threads

---

- The primary **advantages of using *threads*** are that they make it **easy to share data** (via global variables) between cooperating *threads* and that some algorithms transpose more naturally to a *multithreaded* implementation than to a *multiprocess* implementation.
- A *multithreaded* application can transparently take advantage of the possibilities for parallel processing on multiprocessor hardware.

# Client-Server Architecture

---

A ***client-server application*** is one that is broken into two component processes:

- a ***client***, which asks the server to carry out some *service* by sending it a request message
- a ***server***, which examines the client's request, performs appropriate actions, and then sends a response message back to the client.

Typically, the client application interacts with a user, while the server application provides access to some shared resource. There are multiple instances of client processes communicating with a few (if not only one) instances of the server process

# Realtime

---

- ***Realtime* applications are those that need to respond in a timely fashion to input.** Frequently, such input comes from an external sensor or a specialized input device, and output takes the form of controlling some external hardware.
- The defining factor is that **the response is guaranteed to be delivered within a certain deadline** time after the triggering event.
- The provision of realtime responsiveness, especially where short response times are demanded, requires support from the underlying operating system
- Examples of applications with realtime response requirements include **automated assembly lines, bank ATMs, and aircraft navigation systems.**

# BIL 344 System Programming

Week 2

---

System Programming Concepts :

- System Calls, Library Functions, Standard C library...

Processes :

- Processes and Programs
- Memory Layout of a process
- Command line arguments, Environment list ...

Memory Allocation :

- Allocating memory on the Heap
- Allocating memory on the Stack

# System Calls

---

A *system call* is a controlled entry point into the kernel, allowing a process to request the kernel to perform some action on the process's behalf.

The kernel makes a range of services accessible to programs via the system call application programming interface.

These services include, creating a new process, performing I/O, creating a pipe for interprocess communication and so on.

...show the list

# System Calls

---

A system call changes the processor state from *user mode* to *kernel mode*, so that the CPU can access protected kernel memory.

The set of system calls is fixed. Each system call is identified by a unique number, normally not visible to the programs.

Each system call may have a set of arguments that specify information to be transferred from user space to kernel space and vice-versa.

# System Calls

---

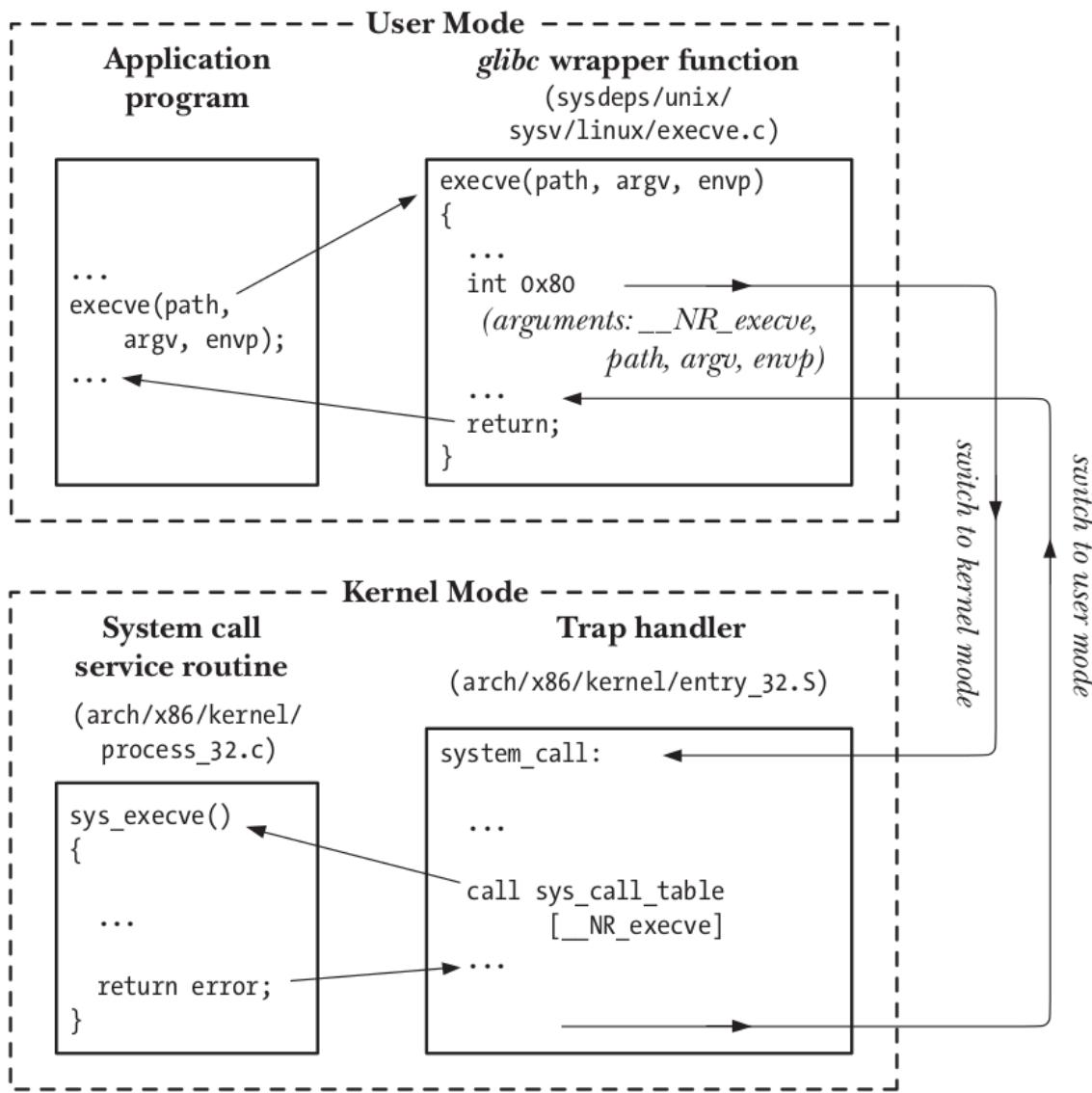
In **kernel mode**, the executing code has **complete and unrestricted access** to the underlying hardware. It can execute **any CPU instruction** and reference **any memory address**. Kernel mode is generally reserved for the lowest-level, most trusted functions of the operating system. Crashes in kernel mode are **catastrophic**; they will halt the entire PC.

In **user mode**, the executing code has **no ability to directly access** hardware or reference memory. Code running in user mode **must delegate** to system APIs to access hardware or memory. Due to the protection afforded by this sort of isolation, crashes in user mode are always **recoverable**. Most of the code running on your computer will execute in user mode.

# System Calls

From a programming point of view, invoking a system call looks much like calling a C function.

However, behind the scenes, many steps occur during the execution of a system call.



# What is really going on ?

1. The application program makes a system call by invoking **a wrapper function** in the C library.
2. The wrapper function must make all of the system call arguments available to the system call ***trap-handling routine***. These arguments are passed to the wrapper via the stack, but the **kernel expects them in specific registers**. The wrapper function copies the arguments to these registers.  
Example for the `write(1, msg, len)` system call:

```
movl  $len,%edx      # third argument: message length
movl  $msg,%ecx      # second argument: pointer to message to write
movl  $1,%ebx         # first argument: file handle (stdout)
movl  $4,%eax         # system call number (sys_write)
int   $0x80            # call kernel
```

# What is really going on ?

---

3. Since all system calls enter the kernel in the same way, the kernel needs some method of identifying the system call. To permit this, the wrapper function copies the system call number into a specific *CPU register*.
4. The wrapper function executes a trap machine instruction (int 0x80 in our example), which causes the processor to switch from user mode to kernel mode and execute code pointed to by location 0x80 (128 decimal) of the system's trap vector.

```
movl    $len,%edx      # third argument: message length
movl    $msg,%ecx      # second argument: pointer to message to write
movl    $1,%ebx        # first argument: file handle (stdout)
movl    $4,%eax        # system call number (sys_write)
int    $0x80            # call kernel
```

## What is really going on ?

---

5. In response to the trap to location 0x80 , the kernel invokes its *system\_call()* routine to handle the trap. This handler:
  - a) Saves register values onto the kernel stack
  - b) Checks the validity of the system call number
  - c) Invokes the appropriate system call service routine, which is found by using the system call number to index a table of all system call service routines
  - d) Restores register values from the kernel stack and places the system call return value on the stack
  - e) Returns to the wrapper function, simultaneously returning the processor to user mode.

# What is really going on ?

---

6. If the return value of the system call service routine indicated an error, the wrapper function sets the global variable *errno* using this value. The wrapper function then returns to the caller, providing an integer return value indicating the success or failure of the system call.

/usr/include/errno.h

More recent x86-32 architectures implement the `sysenter` instruction, which provides a faster method of entering `kernel mode` than the conventional `int 0x80` trap instruction. The use of `sysenter` is supported in the 2.6 `kernel` and from `glibc 2.3.2` onward.

# Library Functions

---

- A *library function* is simply one of the multitude of functions that constitutes a programming language (standard C in our case). The purposes of these functions are very diverse (like opening a file, converting a time to a human-readable format, and comparing two character strings).
- Many library functions don't make any use of system calls. However, some library functions are layered on top of system calls
- Often, library functions are designed to provide a more caller-friendly interface than the underlying system call.

```
FILE *fopen(const char *path, const char *mode);  
int open(const char *pathname, int flags, mode_t mode);
```

# Standard C library

---

- There are different implementations of the standard C library on the various UNIX implementations. The most commonly used implementation on Linux is the GNU C library (*glibc*)

# Error Handling

---

- Nearly all system call and library function returns some type of status value indicating whether the call succeeded or failed. This status value should always be checked to see whether the call succeeded. If it did not, then appropriate action should be taken—at the very least, the program should display an error message warning that something unexpected occurred.
- Many hours of debugging time can be wasted due to a check not made on the status return of a system call or library function that “couldn’t possibly fail”
- The manual page for each system call documents the possible return values of the call, showing which value(s) indicate an error. Usually, an error is indicated by a return of `-1`.

...

```
fd = open(pathname, flags, mode);           /* system call to open a file */
if (fd == -1) {
/* Code to handle the error */
}
...

if (close(fd) == -1) {
/* Code to handle the error */
}
```

...

# Error Handling (system calls)

- When a system call fails, it sets the global integer variable *errno* to a positive value that identifies the specific error. Including the `<errno.h>` header file provides a declaration of *errno*, as well as a set of constants for the various error numbers.

```
cnt = read(fd, buf, numbytes);
if (cnt == -1) {
    if (errno == EINTR)
        fprintf(stderr, "read was interrupted by a signal\n");
else {
    /* Some other error occurred */
}
```

- When checking for an error, one should always first check if the function return value indicates an error, and only then examine *errno* to determine the cause of the error.
- `/usr/include/asm-generic/errno-base.h`

# Error Handling (library functions)

- A common course of action after a failed system call is to print an error message based on the *errno* value. The *perror()* and *strerror()* library functions are provided for this purpose.
- *perror()* prints to stderr the string pointed to by its *msg* argument, followed by a message corresponding to the current value of *errno*.

```
#include <stdio.h>

void perror(const char *msg);
```

- The *strerror()* function returns the error string corresponding to the error number given in its *errnum* argument.

```
#include <string.h>

char *strerror(int errnum);
```

# Error Handling (library functions)

```
#include <stdio.h>

int main () {
    FILE *fp = fopen("file.txt", "r");
    if( fp == NULL ) {
        perror("Error: ");
        return(-1);
    }
    fclose(fp);
    return(0);
}
```

Program output:

```
Error: : No such file or directory
```

# Error Handling (library functions)

---

The various library functions return different data types and different values to indicate failure. For our purposes, library functions can be divided into the following categories

- Some library functions **return error information in exactly the same way as system calls: a `-1` return value, with `errno` indicating the specific error**. Errors from these functions can be diagnosed in the same way as errors from system calls.
- Some library functions **return a value other than `-1` on error**, but nevertheless set `errno` to indicate the specific error condition. The `perror()` and `strerror()` functions can be used to diagnose these errors.
- Other **library functions don't use `errno` at all**. The method for determining the existence and cause of errors depends on the particular function and is documented in the function's manual page. For these functions, it is **a mistake to use `errno`, `perror()`, or `strerror()` to diagnose errors**.

# Processes and Programs

---

A *process* is an instance of an executing program.

A **program** is a file containing a range of information that describes how to construct a process at run time. This information includes:

- ***Binary format identification***: Each program file includes meta-information describing the **format of the executable file**. This enables the kernel to interpret the remaining information in the file.
- ***Machine-language instructions***: These encode the algorithm of the program
- ***Program entry-point address***: This identifies the location of the instruction at which execution of the program should commence
- ***Data***: The program file contains values used to initialize variables and also literal constants used by the program (e.g., strings).

# Processes

---

- ***Symbol and relocation tables***: These describe the locations and names of functions and variables within the program. These tables are used for a variety of purposes, including debugging and run-time symbol resolution (dynamic linking).
- ***Shared-library and dynamic-linking information***: The program file includes fields listing the shared libraries that the program needs to use at run time and the pathname of the dynamic linker that should be used to load these libraries.
- ***Other information***: The program file contains various other information that describes how to construct a process.

**One program may be used to construct many processes, or, many processes may be running the same program.**

# Processes

---

- A process is an abstract entity, defined by the kernel, to which system resources are allocated in order to execute a program
- From the **kernel's point of view, a process consists of user-space memory containing program code and variables used by that code, and a range of kernel data structures** that maintain information about the state of the process
- The information recorded in the kernel data structures includes various identifier numbers (IDs) associated with the process, **virtual memory tables, the table of open file descriptors, information relating to signal delivery and handling, process resource usages and limits, the current working directory**, and a host of other information

# Process ID and Parent Process ID

Each process has a process ID (PID), a positive integer that uniquely identifies the process on the system.

The *getpid()* system call returns the process ID of the calling process.

```
#include <uninst.h>

pid_t getpid(void);
                                         Always successfully returns process ID of caller
```

Each process has a parent—the process that created it. A process can find out the process ID of its parent using the *getppid()* system call.

```
#include <uninst.h>

pid_t getppid(void);
                                         Always successfully returns process ID of parent of caller
```

# Process ID and Parent Process ID

---

The parent process ID attribute of each process represents the tree-like relationship of all processes on the system. The parent of each process has its own parent, and so on, going all the way back to process 1, *init*, the ancestor of all processes.

If a child process becomes orphaned because its “birth” parent terminates, then the child is adopted by the *init* process, and subsequent calls to *getppid()* in the child return 1

PIDS wrap up when maxed from a safe min number (e.g. 300)

# Memory Layout of a Process

The memory allocated to each process is composed of a number of parts, usually referred to as *segments*. These segments are as follows :

- The ***text segment*** contains the machine-language instructions of the program run by the process. The text segment is made **read-only** so that a process doesn't accidentally modify its own instructions via a bad pointer value.
- The ***initialized data segment*** contains global and static variables that are explicitly initialized.
- The ***uninitialized data segment*** contains global and static variables that are not explicitly initialized
- The ***stack*** is a dynamically growing and shrinking segment containing stack frames.
- The ***heap*** is an area from which memory (for variables) can be dynamically allocated at run time.

```

#include <stdio.h>
#include <stdlib.h>

char globBuf[65536];           /* Uninitialized data segment */
int primes[] = { 2, 3, 5, 7 };  /* Initialized data segment */

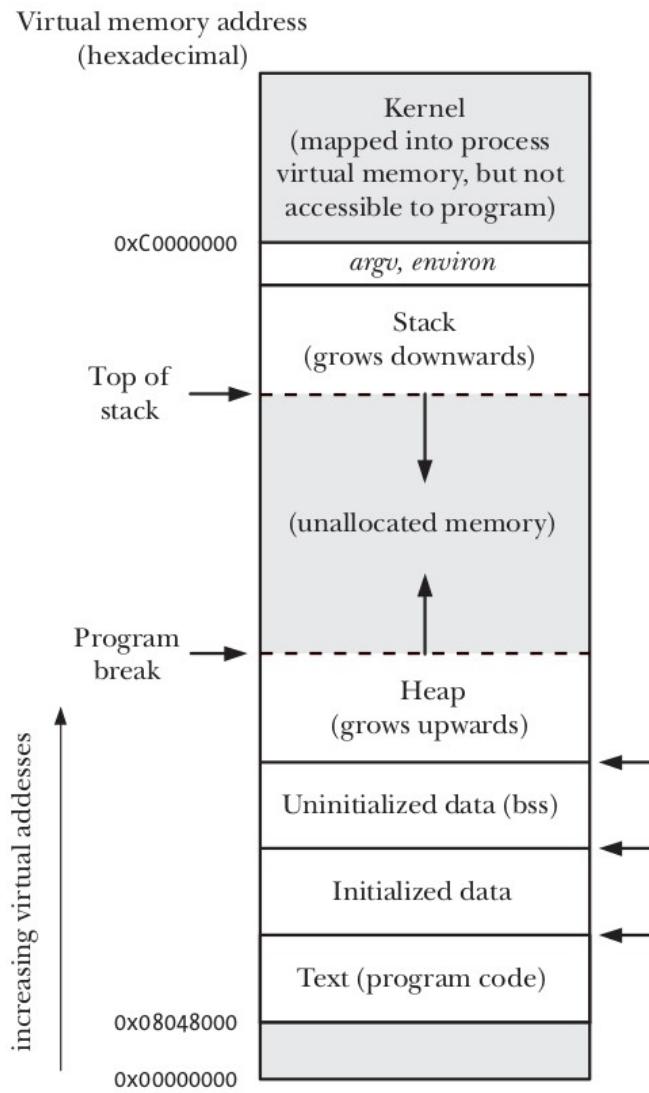
static int square(int x)        /* Allocated in frame for square() */
{
int result;                   /* Allocated in frame for square() */
result = x * x;
return result;                /* Return value passed via register */
}

static void doCalc(int val)    /* Allocated in frame for doCalc() */
{
printf("The square of %d is %d\n", val, square(val));
if (val < 1000) {
    int t;                   /* Allocated in frame for doCalc() */
    t = val * val * val;
    printf("The cube of %d is %d\n", val, t);
}
}

int main(int argc, char *argv[]) /* Allocated in frame for main() */
{
static int key = 9973;          /* Initialized data segment */
static char mbuf[10240000];    /* Uninitialized data segment */
char *p;                       /* Allocated in frame for main() */
p = malloc(1024);              /* Points to memory in heap segment */
doCalc(key);
exit(EXIT_SUCCESS);
}

```

# Typical memory layout of a process (x86)



**Why have 2 segments for initialized and uninitialized data?**

**Optimization!**

**Otherwise, if everything was in the data segment, it would contain a lot of zeroes; initialization would be slower because of skipping RAM spots; it would be way worse with embedded programs.**

**cf. size command**

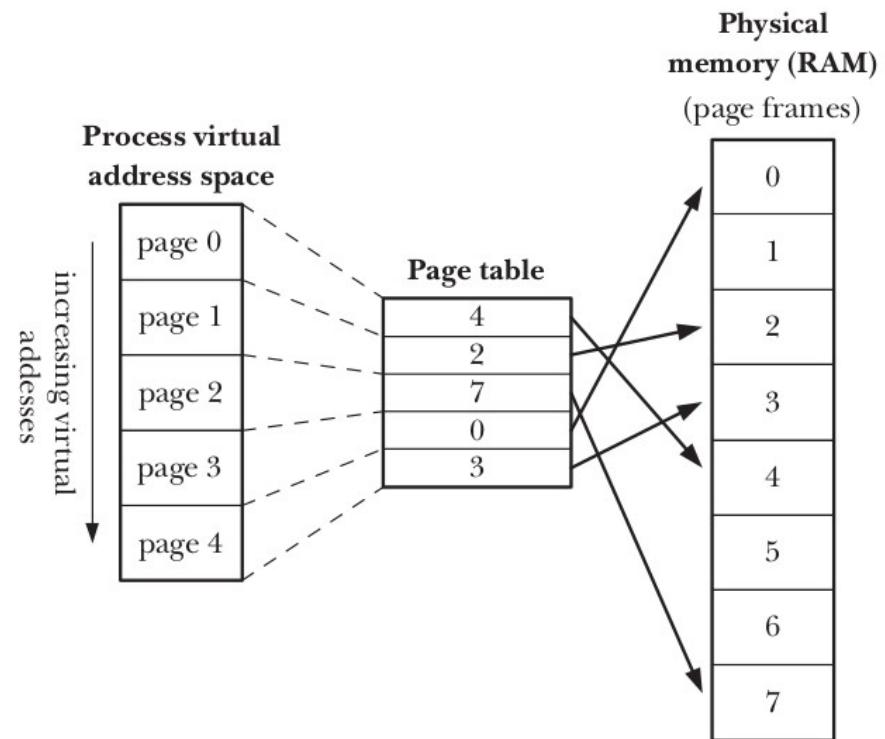
# Virtual Memory Management

- Modern Operating systems employ a technique known as *virtual memory management*. The aim is to make efficient use of both the CPU and RAM (physical memory) by exploiting a property that is typical of most programs: *locality of reference*
- Most programs demonstrate two kinds of locality:
  - *Spatial locality* is the tendency of a program to reference memory addresses that are near those that were recently accessed
  - *Temporal locality* is the tendency of a program to access the same memory addresses in the near future that it accessed in the recent past

By using locality of reference, it **is possible to execute a program while maintaining only part of its address space** in physical memory (RAM)

# Virtual Memory Management

- The kernel maintains a ***page table*** for each process. The page table describes the location of each page in the process's virtual address space (the set of all virtual memory pages available to the process).
- Each entry in the page table either indicates the location of a virtual page in RAM or indicates that it currently resides on disk.



# Virtual Memory Management

---

- Not all address ranges in the process's virtual address space require page-table entries. Typically, large ranges of the potential virtual address space are unused, so that it isn't necessary to maintain corresponding page-table entries.
- **If a process tries to access an address for which there is no corresponding page-table entry, it receives a SIGSEGV signal.**
- A process's range of valid virtual addresses can change over its lifetime, as the kernel allocates and deallocates pages (and page-table entries) for the process.
- Virtual memory management separates the virtual address space of a process from the physical address space of RAM.

# The Stack and Stack Frames

---

- The **stack grows and shrinks linearly as functions are called and return**. On most other UNIX implementations, the stack resides at the high end of memory and grows downward (toward the heap).
- A special-purpose register, the ***stack pointer***, tracks the current top of the stack. Each time a function is called, an additional frame is allocated on the stack, and this frame is removed when the function returns.
- Sometimes, the term ***user stack*** is used to distinguish the stack we describe here from the ***kernel stack***.
- The **kernel stack is a per-process memory region** maintained in kernel memory that is used as the stack for execution of the functions called internally during the execution of a system call

# Command-Line Arguments ( *argc* , *argv* )

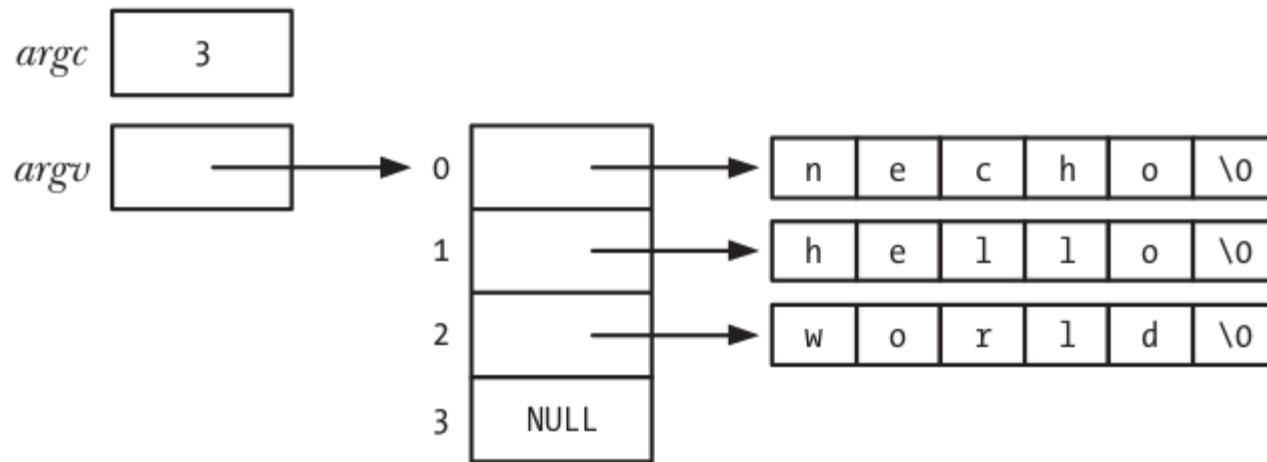
- Every C program must have a function called *main()*, which is the point where execution of the program starts. When the program is executed, the command-line arguments (the separate words parsed by the shell) are made available via two arguments to the function *main()* , *argc* and *argv*
- The first argument, *int argc*, indicates how many command-line arguments there are.
- The second argument, *char \*argv[]*, is an array of pointers to the command-line arguments, each of which is a null-terminated character string.
- The first of these strings, in *argv[0]*, is the name of the program itself. The list of pointers in *argv* is terminated by a *NULL* pointer (i.e., *argv[argc]* is *NULL* )

```

#include "tlpi_hdr.h"

int main(int argc, char *argv[])
{
    int j;
    for (j = 0; j < argc; j++)
        printf("argv[%d] = %s\n", j, argv[j]);
    exit(EXIT_SUCCESS);
}
/* necho.c */

```



Alternatively, since `argv` list is terminated by NULL value the body of the `main()` above can be implemented as

```

char **p;

for (p = argv; *p !=NULL; p++)
    puts(*p);

```

# Environment List

---

- Each process has an associated array of strings called the *environment list* (simply the *environment*). Each of these strings is a definition of the form *name*=*value*. Thus, the environment represents a set of name-value pairs that can be used to hold arbitrary information. The names in the list are referred to as *environment variables*.
- When a new process is created, it inherits a copy of its parent's environment. Since the child gets a copy of its parent's environment at the time it is created, this transfer of information is one-way and once-only. After the child process has been created, either process may change its own environment, and these changes are not seen by the other process.

# Environment List

- Within a C program, the **environment list** can be accessed using the global variable `char **environ` (a NULL-terminated list of pointers).

```
#include <stdlib.h>
#include <unistd.h>

extern char **environ;

int main(int argc, char *argv[])
{
    char **ep;
    for (ep = environ; *ep != NULL; ep++)
        puts(*ep);
    exit(EXIT_SUCCESS);
} /* display env.c */
```

# Environment List

In order to manipulate the environment in a C program you may also use

```
#include <stdlib.h>

char *getenv(const char * name );
int putenv(char * string );
int setenv(const char * name , const char * value , int overwrite );
int unsetenv(const char * name );
int clearenv(void) ;
```

# Memory Allocation

---

## Allocating Memory on the Heap

- A process can allocate memory by increasing the size of the heap, a variable size segment of contiguous virtual memory that begins just after the uninitialized data segment of a process and grows and shrinks as memory is allocated and freed.
- The current limit of the heap is referred to as the *program break*
- To allocate memory, C programs normally use the *malloc* family of functions (**they are not system calls**).

# Memory Allocation

## Adjusting the Program Break: *brk()* and *sbrk()*

- Resizing the heap is just requesting the kernel to adjust its idea of where the process's program break is.
- Initially, the program break lies just past the end of the uninitialized data segment. After the program break is increased, the program may access any address in the newly allocated area, but no physical memory pages are allocated yet. The kernel automatically allocates new physical pages on the first attempt by the process to access addresses in those pages.
- Traditionally, the UNIX system has provided two system calls for manipulating the program break,

```
#include <unstd.h>

int brk(void * end_data_segment );
    /* Returns 0 on success, or -1 on error */
void *sbrk(intptr_t increment );
    /* Returns previous program break on success, or (void) -1 on error*/
```

Newer implementations use **mmap** and **sbrk** to handle the heap

# Memory Allocation

---

## Allocating Memory on the Heap: *malloc()* and *free()*

C programs use the *malloc* family of functions to allocate and deallocate memory on the heap. These functions offer several advantages over *brk()* and *sbrk()*. In particular, they:

- are standardized as part of the C language;
- are easier to use in threaded programs;
- provide a simple interface that allows memory to be allocated in small units;
- allow us to arbitrarily deallocate blocks of memory, which are maintained on a free list and recycled in future calls to allocate memory.

# Memory Allocation

- The *malloc()* function allocates size bytes from the heap and returns a pointer to the start of the newly allocated block of memory. The allocated memory is “not initialized”.

```
#include <stdlib.h>

void *malloc(size_t size);

/* Returns pointer to allocated memory on success*/
```

- If memory could not be allocated (perhaps because we reached the limit to which the program break could be raised), then *malloc()* returns NULL and sets errno to indicate the error.
- Although the possibility of failure in allocating memory is small, **all calls to *malloc()*, and the related functions, should check for this error return.**

# Memory Allocation

- The *free()* function deallocates the block of memory pointed to by its *ptr* argument, which should be an address previously returned by *malloc()*

```
#include <stdlib.h>

void *free(void *ptr);
```

- In general, *free()* doesn't lower the program break, but instead adds the block of memory to a list of free blocks that are recycled by future calls to *malloc()*.

## To *free()* or not to *free()* ?

- When a process terminates, all of its memory is returned to the system, including heap memory allocated by functions in the *malloc* package. In programs that allocate memory and continue using it until program termination, it is common to omit calls to *free()*, relying on this behavior to automatically free the memory.
- This can be especially useful in programs that allocate many blocks of memory, since adding multiple calls to *free()* could be expensive in terms of CPU time, as well as perhaps being complicated to code.
- Although relying on process termination to automatically free memory is acceptable for many programs, there are many reasons why it can be desirable to explicitly free all allocated memory.

# Other Methods of Allocating Memory on the Heap

- As well as *malloc()*, the C library provides a range of other functions for allocating memory on the heap, the C library provides a range of other functions for allocating memory on the heap

```
#include <stdlib.h>

void *calloc(size_t numitems , size_t size );
    /*Returns pointer to allocated memory on success, or NULL on error*/
```

The *calloc()* function allocates memory for an array of identical items. The *numitems* argument specifies how many items to allocate, and *size* specifies their size. After allocating a block of memory of the appropriate size, *calloc()* returns a pointer to the start of the block.

**Note that *calloc()* initializes the allocated memory to 0 whereas *malloc* doesn't!**

# Other Methods of Allocating Memory on the Heap

- The *realloc()* function is used to resize a block of memory previously allocated by one of the functions in the *malloc* package.

```
#include <stdlib.h>

void *realloc(void * ptr , size_t size );
    /*Returns pointer to allocated memory on success, or NULL on error*/
```

The *ptr* argument is a pointer to the block of memory that is to be resized. The *size* argument specifies the desired new size of the block. On success, *realloc()* returns a pointer to the location of the resized block (this may be different from its location before the call). On error, *realloc()* returns NULL and leaves the block pointed to by *ptr* untouched

# Allocating Memory on the Stack: *alloca()*

- Like the functions in the *malloc* package, *alloca()* allocates memory dynamically. However, instead of obtaining memory from the heap, *alloca()* obtains memory from the stack by increasing the size of the stack frame.

```
#include <alloca.h>

void *alloca(size_t size );
                                         /*Returns pointer to allocated block of memory */
```

The *size* argument specifies the number of bytes to allocate on the stack. The *alloca()* function returns a pointer to the allocated memory as its function result.

- We need not—indeed, must not—call *free()* to deallocate memory allocated with *alloca()*. Likewise, it is not possible to use *realloc()* to resize a block of memory allocated by *alloca()*. You cannot not use *alloca()* within a function argument list.

# Allocating Memory on the Stack: *alloca()*

- Using *alloca()* to allocate memory has a few advantages over *malloc()*. One of these is that allocating blocks of memory is faster with *alloca()* than with *malloc()*, as *alloca()* is implemented by the compiler as inline code that directly adjusts the stack pointer.
- Furthermore, *alloca()* doesn't need to maintain a list of free blocks
- It is automatically freed when the allocating function returns
- Alloca: is VERY FAST for dynamic allocation but dangerous (stack overflow risks), and unsupported by ANSI-C, so it limits portability. Use with caution or use *malloc* instead!

# BIL 344 System Programming

Week 3

---

*Files and low level I/O in the UNIX world*

- *open/close/read/write/lseek/fcnt/ioctl*
- *Redirection*
- *File implementation in UNIX*
- *File stats*
- *Temporary files*
- *Relationship between file descriptors and open files*
- *Buffering and stdio*

# Files

---

"On a UNIX system, everything is a file; if something is not a file, it is a process."<sup>\*</sup>

Directories: **files** containing the names of files in them

Programs, texts, images, videos: **files**; either binary or ASCII

Devices, e.g. monitor, cpu, gpu, printer: all represented as **files**

Consequently, it is highly important to know how to handle them!

<sup>\*</sup> minor exceptions apply

# File types

## Regular

```
$ ls -l *
-rw-r--r-- 1 greys greys 1024 Mar 29 06:31 text
```

## Directory

```
$ ls -ld *
-rw-r--r-- 1 greys greys 1024 Mar 29 06:31 text
d-rwxr-xr-x 2 greys greys 4096 Aug 21 11:00 mydir
```

Device file (**c**: streams sequential data one by at a time, **b**: provides random access to blocks of data)

```
$ ls -al /dev/loop0 /dev/ttys0
brw-rw---- 1 root disk 7, 0 Sep 7 05:03 /dev/loop0
crw-rw-rw- 1 root tty 3, 48 Sep 7 05:04 /dev/ttys0
```

# File types

---

## Named Pipe (IPC)

```
$ ls -al /dev/xconsole
p r w - - - - 1 root adm 0 Sep 25 08:58 /dev/xconsole
```

## Symbolic link

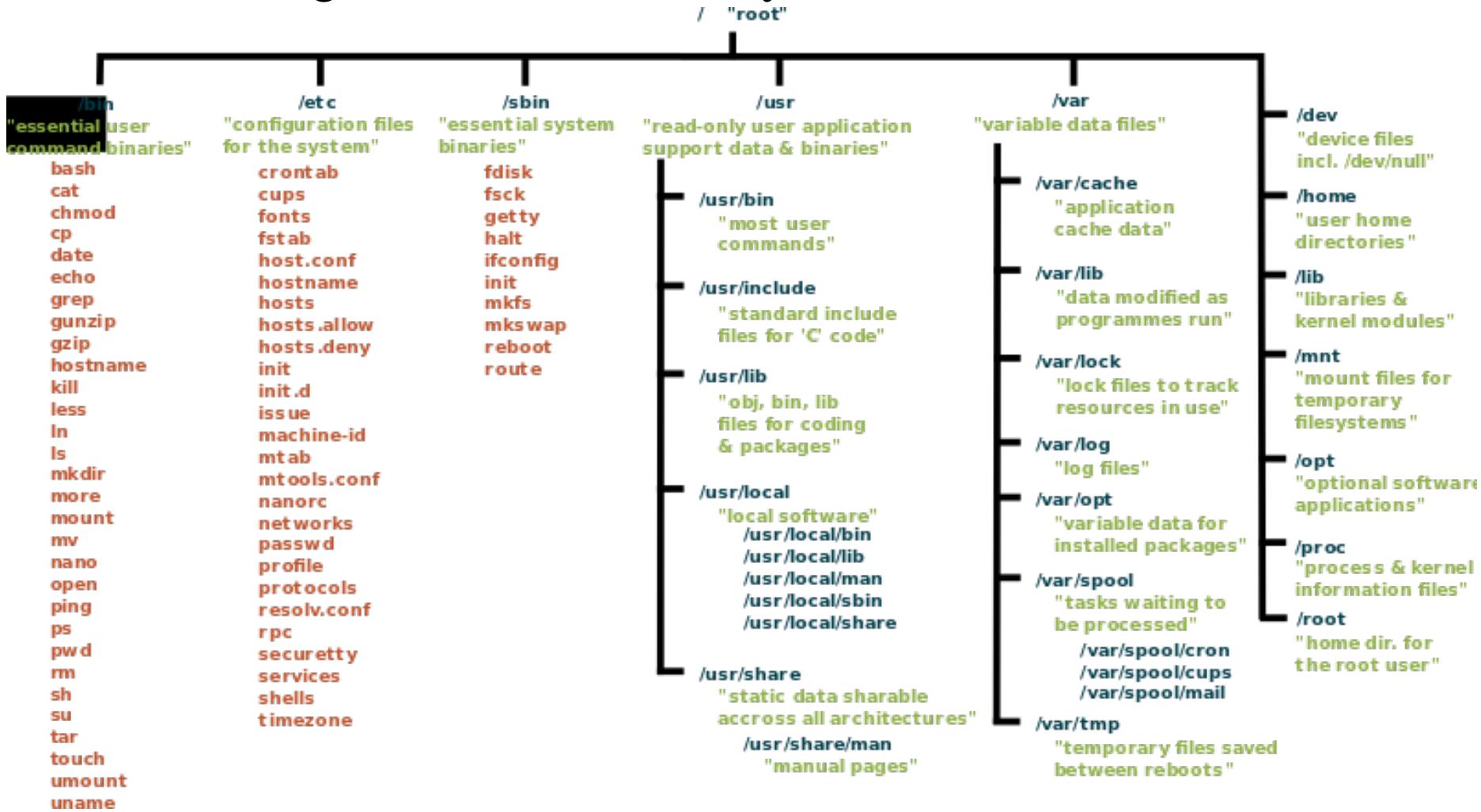
```
$ ls -al hosts
l r w x r w x r w x 1 greys www-data 10 Sep 25 09:06 hosts -> /etc/host
```

## Socket (IPC)

```
$ ls -al /dev/log
s r w - r w - r w - 1 root root 0 Sep 7 05:04 /dev/log0
```

# Files

All files are organized within a **file system**; a rooted tree of directories.



# Files

---

Unlike windows where each drive has a letter that's the root of its own FS, in UNIX, drives, partitions, removable media and even network shares can be **mounted**, and thus the entire volume's FS appears as a directory. The root is always denoted by / ("slash")

**/bin**: binaries, ls, cp, etc.

**/home** : user directories

**/boot**: files needed for booting the system

**/etc**: system-wide configuration files

**/dev**: file representations of devices      **/lib**: libraries

**/proc**: files representing runtime system information

**/usr**: read-only stuff: non-system critical binaries, libraries, resources

**/var**: variable data files: logs, temporary files, mail, print jobs, etc.

# Files

---

An arbitrary location or address within this tree structure is known as a **path**, e.g.: /home/erhan/courses/bil344/week3.pdf

Every directory contains the files: “.” and “..” that represent respectively the current and parent directories.

If a path starts with / then it's an **absolute or fully qualified path**, otherwise, the program prepends the absolute path of the current working directory;

e.g, you are located at: /home/erhan/courses/bil344

..../bil464/midterm.pdf -> /home/erhan/courses/bil464/midterm.pdf

# Files

All system calls that deal with files (of any type) refer to them through **file descriptors**; i.e a small non-negative integer.

**All programs** start with 3 open files that are opened on their behalf by the shell:

File descriptor	Purpose	POSIX name	<i>stdio</i> stream
0	standard input	STDIN_FILENO	<i>stdin</i>
1	standard output	STDOUT_FILENO	<i>stdout</i>
2	standard error	STDERR_FILENO	<i>stderr</i>

# Files

---

There are four key system calls upon which programming libraries (fopen, fclose, fwrite, etc) rely for file I/O:

- `fd = open(pathname, flags, mode)`

Opens the file *pathname* and returns its file descriptor

- `numread= read(fd, buffer, count)`

Read at most count bytes from the open file fd and stores in buffer

- `numwritten=write(fd, buffer, count)`

Writes up to count bytes from buffer into the open file fd

- `status=close(fd)`

Is called after all I/O operations are completed, and releases resources

# open

---

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

## Flag examples:

O\_RDONLY: read only; O\_WRONLY: write only

O\_RDWR: read and write

O\_CREAT: create a new file

O\_APPEND: any data written to the file will be appended to its end

O\_TRUNC: discard previous content

O\_EXCL: used together with O\_CREAT, returns error if the file already exists

O\_NONBLOCK: if the file cannot be opened, instead of blocking, returns an error

...and more..

**Returns** the fd or -1 in case of error.

# open

**mode**: is an octal number specifying the permissions of the newly created file (in conjunction with umask and the access permissions of the parent directory).

POSIX defines symbolic names for the permission masks so that you can specify them independently of the underlying implementation (defined in `sys/stat.h`)

`S_I(R|W|X)(USR|GRP|OTH)`

e.g.

`S_IRUSR`: read access for user

`S_IWGRP`: write access for group

`S_IXOTH`: execution permission for others

Having **read** permission on a file grants the right to read the contents of the file. Read permission on a directory implies the ability to list all the files in the directory.

**Write** permission implies the ability to change the contents of the file (for a file) or create new files in the directory (for a directory).

**Execute** permission on files means the right to execute them, if they are programs. (Files that are not programs should not be given the execute permission.) For directories, execute permission allows you to enter the directory

# Example: open

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
    /* The path at which to create the new file.  */
    char* path = argv[1];
    /* The permissions for the new file.  */
    mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH;

    /* Create the file.  */
    int fd = open (path, O_WRONLY | O_EXCL | O_CREAT, mode);
    if (fd == -1) {
        /* An error occurred.  Print an error message and bail.  */
        perror ("open");
        return 1;
    }

    return 0;
}
```

# close

---

Even though when a process terminates the OS closes all open fd's associated with that process, it is good practice to `close` a file once you are done with it.

```
#include <unistd.h>
int close(int fd);
```

**Returns** zero on success and -1 on error.

Open file descriptors use kernel resources (every process needs a table to keep track of its open files). The typical limit is 1024 file descriptors per process. You can adjust this limit through the `getrlimit` and `setrlimit` system calls.

# write

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

It writes up to count bytes from the buffer pointed by buf **to the current offset** of the file referred to by the file descriptor fd. It might write less than count due to a signal interruption, etc. The data to write need not be a character string; it works with arbitrary bytes.

**Returns** the number of bytes written or -1 on error.

Error examples:

EBADF: fd is not a valid file descriptor or is not open for writing.

ENOSPC: the device containing the file referred to by fd has no room for the data.

# Example: write

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <time.h>
#include <unistd.h>

/* Return a character string representing the current date and time. */

char* get_timestamp ()
{
    time_t now = time (NULL);
    return asctime (localtime (&now));
}

int main (int argc, char* argv[])
{
    /* The file to which to append the timestamp. */
    char* filename = argv[1];
    /* Get the current timestamp. */
    char* timestamp = get_timestamp ();
    /* Open the file for writing. If it exists, append to it;
       otherwise, create a new file. */
    int fd = open (filename, O_WRONLY | O_CREAT | O_APPEND, 0666);
    /* Compute the length of the timestamp string. */
    size_t length = strlen (timestamp);
    /* Write the timestamp to the file. */
    write (fd, timestamp, length);
    /* All done. */
    close (fd);
    return 0;
}
```

example on how to append a timestamp to a file

**r:4, w: 2, x: 1**

**0666: rw-rw-rw-**

```
% ./timestamp tsfile
% cat tsfile
Thu Feb 1 23:25:20 2001
% ./timestamp tsfile
% cat tsfile
Thu Feb 1 23:25:20 2001
Thu Feb 1 23:25:47 2001
```

**Bad example: does not check whether the system calls succeeded.**

# read

---

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count)
```

Similar to write in principle. Returns the number of bytes read, 0 on EOF, -1 on error.

**Warning:** in the UNIX world file lines are separated by the newline character '\n' (ASCII 10). In the windows world, lines are separated by two characters: a carriage return '\r' (ASCII 13) **and** a newline character.

So if your file was saved in a windows environment, and you read it in a UNIX environment, do not be alarmed when you see the ^M expression (corresponding to the carriage return character) at the end of every line.

# Example: read

```
#include <fcntl.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
    unsigned char buffer[16];
    size_t offset = 0;
    size_t bytes_read;
    int i;

    /* Open the file for reading.  */
    int fd = open (argv[1], O_RDONLY);

    /* Read from the file, one chunk at a time.  Continue until read
     * "comes up short", that is, reads less than we asked for.
     * This indicates that we've hit the end of the file.  */
    do {
        /* Read the next line's worth of bytes.  */
        bytes_read = read (fd, buffer, sizeof (buffer));
        /* Print the offset in the file, followed by the bytes themselves.  */
        printf ("0x%06x : ", offset);
        for (i = 0; i < bytes_read; ++i)
            printf ("%02x ", buffer[i]);
        printf ("\n");
        /* Keep count of our position in the file.  */
        offset += bytes_read;
    }
    while (bytes_read == sizeof (buffer));
    /* All done.  */
    close (fd);
    return 0;
}
```

print the hexadecimal dump of a file

% ./hexdump hexdump

0x000000 : 7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
0x000010 : 02 00 03 00 01 00 00 00 c0 83 04 08 34 00 00 00
0x000020 : e8 23 00 00 00 00 00 00 34 00 20 00 06 00 28 00
0x000030 : 1d 00 1a 00 06 00 00 00 34 00 00 00 34 80 04 08
...

# Example: copying a file

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/stat.h>

#define READ_FLAGS O_RDONLY
#define WRITE_FLAGS (O_WRONLY | O_CREAT | O_EXCL)
#define WRITE_PERMS (S_IRUSR | S_IWUSR)

/* function definitions */
int copyfile(int fromfd, int tofd);

int main(int argc, char *argv[]) {
    int bytes;
    int fromfd, tofd;

    if (argc != 3) {
        fprintf(stderr, "Usage: %s from_file to_file\n", argv[0]);
        return 1;
    }

    if ((fromfd = open(argv[1], READ_FLAGS)) == -1) {
        perror("Failed to open input file");
        return 1;
    }

    if ((tofd = open(argv[2], WRITE_FLAGS, WRITE_PERMS)) == -1) {
        perror("Failed to create output file");
        return 1;
    }

    bytes = copyfile(fromfd, tofd);
    printf("%d bytes copied from %s to %s\n", bytes, argv[1], argv[2]);
    /* the return closes the files */
}
```

# Example: copying a file

```
#include <errno.h>
#include <unistd.h>
#define BLKSIZE 1024

int copyfile(int fromfd, int tofd) {
    char *bp;
    char buf[BLKSIZE];
    int bytesread;
    int byteswritten = 0;
    int totalbytes = 0;

    for ( ; ; ) {
        while (((bytesread = read(fromfd, buf, BLKSIZE)) == -1) &&
               (errno == EINTR)) ;           /* handle interruption by signal */
        if (bytesread <= 0)           /* real error or end-of-file on fromfd */
            break;
        bp = buf;
        while (bytesread > 0) {
            while(((byteswritten = write(tofd, bp, bytesread)) == -1 ) &&
                   (errno == EINTR)) ;           /* handle interruption by signal */
            if (byteswritten < 0)           /* real error on tofd */
                break;
            totalbytes += byteswritten;
            bytesread -= byteswritten;
            bp += byteswritten;
        }
        if (byteswritten == -1)           /* real error on tofd */
            break;
    }
    return totalbytes;
}
```

# Iseek

---

A file descriptor remembers its position in a file. As you read or write the position advances depending on the number of bytes read or written. If you want to move arbitrarily within a file then:

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

**offset**: new position. The third argument determines how to interpret the second arg.

whence=SEEK\_SET: number of bytes from the start of the file (only positive)

whence=SEEK\_CUR: number of bytes from the current position (positive or negative)

whence=SEEK\_END: number of bytes from the end of the file (positive or negative)

**Returns** the new position from the beginning. Cannot be used with sockets.

# Example: lseek

```
lseek(fd, 0, SEEK_SET);          /* Start of file */
lseek(fd, 0, SEEK_END);          /* Next byte after the end of the file */
lseek(fd, -1, SEEK_END);         /* Last byte of file */
lseek(fd, -10, SEEK_CUR);        /* Ten bytes prior to current location */
lseek(fd, 10000, SEEK_END);       /* 10001 bytes past last byte of file */
```

Listing B.5 (*lseek-huge.c*) Create Large Files with *lseek*

```
#include <fcntl.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>                  size_t length = (size_t) atoi (argv[2]) * megabyte;

int main (int argc, char* argv[])      /* Open a new file.  */
{
    int zero = 0;                      int fd = open (filename, O_WRONLY | O_CREAT | O_EXCL, 0666);
    const int megabyte = 1024 * 1024;    /* Jump to 1 byte short of where we want the file to end.  */
    char* filename = argv[1];           lseek (fd, length - 1, SEEK_SET);
                                        /* Write a single 0 byte.  */
                                        write (fd, &zero, 1);
                                        /* All done.  */
                                        close (fd);

                                        return 0;
}
```

# lseek

Using `lseek-huge`, we'll make a 1GB (1024MB) file. Note the free space on the drive before and after the operation.

```
% df -h .
Filesystem      Size  Used Avail Use% Mounted on
/dev/hda5        2.9G  2.1G  655M  76% /
% ./lseek-huge bigfile 1024
% ls -l bigfile
-rw-r-----  1 samuel    samuel  1073741824 Feb  5 16:29 bigfile
% df -h .
Filesystem      Size  Used Avail Use% Mounted on
/dev/hda5        2.9G  2.1G  655M  76% /
```

No appreciable disk space is consumed, despite the enormous size of `bigfile`. Still, if we open `bigfile` and read from it, it appears to be filled with 1GB worth of 0s. For instance, we can examine its contents with the `hexdump` program of Listing B.4.

```
% ./hexdump bigfile | head -10
0x000000 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000010 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000020 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000030 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000040 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0x000050 : 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
...
...
```

**a.k.a sparse files**

**e.g. when representing the drive of a virtual machine.**

**It'll have very little data in the beginning, and keep filling with time.**

**No use of allocating XYZGB from day 1.**

These “magic” file holes are a nice property of UNIX file systems. In windows environments this would lead to an actual 1GB file.

# ioctl

## Non standard I/O operations: ioctl

```
#include <sys/ioctl.h>
int ioctl(int d, int request, ...);
```

```
#include <fcntl.h>
#include <linux/cdrom.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
    /* Open a file descriptor to the device specified on the command line. */
    int fd = open (argv[1], O_RDONLY);
    /* Eject the CD-ROM. */
    ioctl (fd, CDROMEJECT);
    /* Close the file descriptor. */
    close (fd);

    return 0;
}
```

It manipulates the underlying device parameters of special files.

It requires detailed understanding of the device represented by the fd.

It's beyond our scope.

Example: ejecting a cdrom.

# fcntl

## Advanced file operations: fcnlt

```
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, int cmd, ... /* arg */ );
```

- It can manipulate the flags associated with a fd (same ones used during opening).
- It can duplicate file descriptors
- It can lock/unlock files - very useful for inter process communication...

To place a lock on a file, first create and zero out a struct flock variable. Set the `l_type` field of the structure to `F_RDLCK` for a read lock or `F_WRLCK` for a write lock. Then call `fcntl`, passing a file descriptor to the file, with the `F_SETLK` operation code, and a pointer to the `struct flock` variable. If another process holds a lock that prevents a new lock from being acquired, `fcntl` blocks until that lock is released.

# Example: fcntl

Listing 8.2 (*lock-file.c*) Create a Write Lock with *fcntl*

```
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>

int main (int argc, char* argv[])
{
    char* file = argv[1];
    int fd;
    struct flock lock;

    printf ("opening %s\n", file);
    /* Open a file descriptor to the file. */
    fd = open (file, O_WRONLY);
    printf ("locking\n");
    /* Initialize the flock structure. */
    memset (&lock, 0, sizeof(lock));
    lock.l_type = F_WRLCK;
    /* Place a write lock on the file. */
    fcntl (fd, F_SETLKW, &lock);
```

Only one process can hold a write-lock for a given fd.

Many can hold a read-lock.

In case of an already acquired lock,  
the call to fcntl will block the process.

```
printf ("locked; hit Enter to unlock... ");
/* Wait for the user to hit Enter. */
getchar ();

printf ("unlocking\n");
/* Release the lock. */
lock.l_type = F_UNLCK;
fcntl (fd, F_SETLK, &lock);

close (fd);
return 0;
}
```

# Redirection

Under normal circumstances a process reads from standard input, outputs its result to standard output and in case of error, it is sent to standard error. We can however redirect them!

In Bourne-style shells: `$ ./myscript > results.log 2>&1`

i.e. redirect stdout to `results.log` and redirect stderr to wherever stdout points to; so both stdout and stderr end at `results.log`

```
$ ./myscript 2>&1 | less
```

Both stderr and stdout of “myscript” become stdin of “less”

`&:` modifies a file into a fd; otherwise 2 is redirected to a file named “1”

# Redirection

---

How does redirection work behind the scenes?

```
#include <unistd.h>
int dup2(int oldfd, int newfd);
```

It makes a duplicate of the file descriptor given in `oldfd` using the descriptor number supplied in `newfd`. If the file descriptor specified in `newfd` is already open, it closes it first; e.g.

```
dup2(1, 2) // 2>&1
```

first closes `stderr`, then replaces it with a copy of `stdout`.

Or you can use `fcntl` instead of `dup2`:

```
newfd = fcntl(oldfd, F_DUPFD, startfd);
```

Uses as `newfd` the lowest unused file descriptor greater than or equal to `startfd`

# File implementation

---

The designers of POSIX have separated file data and meta-data.

All the meta-data concerning a given file reside in a fixed-length structure called **inode** (short for index node).

The inode contains information about the **file size**, the **file location**, the **owner** of the file, the **time of creation**, **time of last access**, **time of last modification**, **permissions** and so on.

In addition to descriptive information about the file, the inode contains pointers to the first few data blocks of the file. If the file is large, the indirect pointer is a pointer to a block of pointers that point to additional data blocks. If the file is still larger, the double indirect pointer is a pointer to a block of indirect pointers. If the file is really huge, the triple indirect pointer contains a pointer to a block of double indirect pointers.

# File implementation

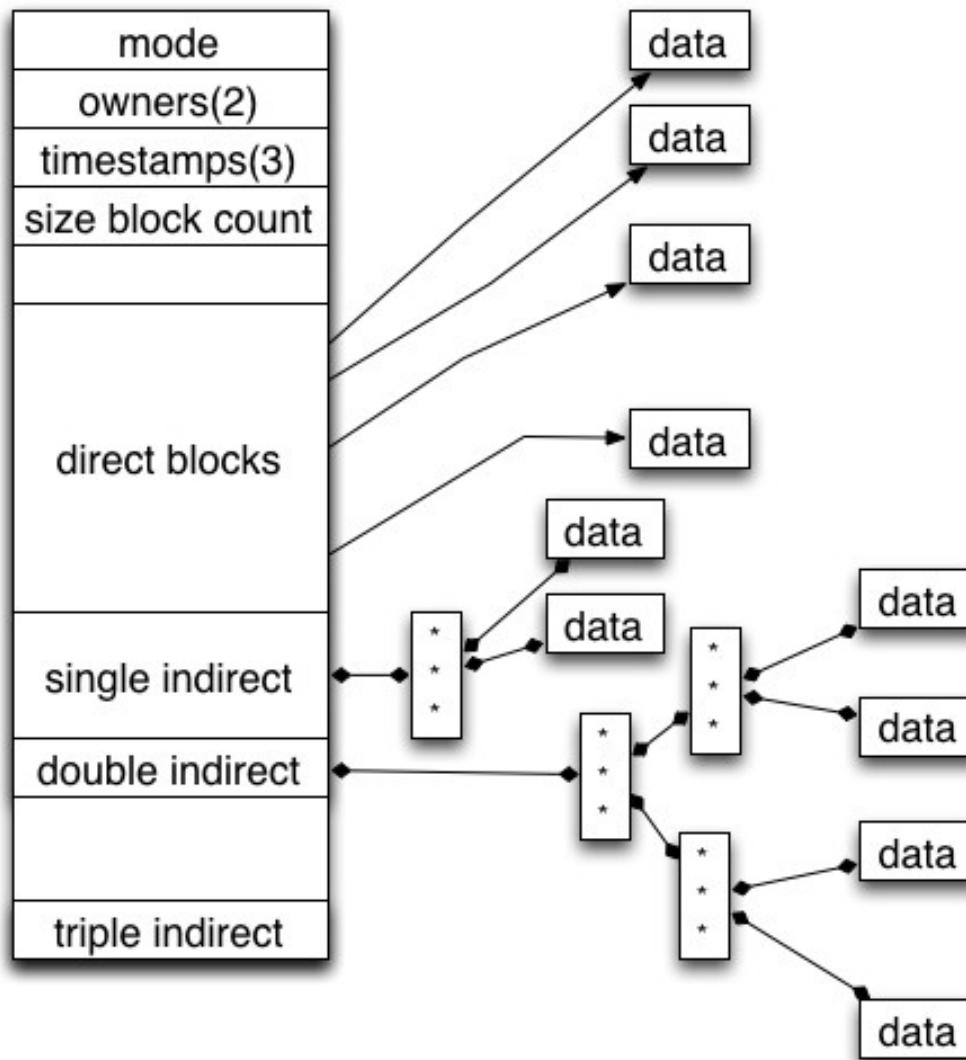


Figure 12.9 UNIX inode

Enables the allocation of data into non-contiguous data blocks

Exercise:  
calculate the  
max supported  
file size on a 64-  
bit system with  
1KB block size.

# Directory implementation

**Directories** in UNIX are basically associate arrays of filenames and inode numbers; e.g.

```
1167010 .
1158721 ..
1167626 subdir
132651 barfile
132650 bazfile
```

*The inode itself does not contain the filename.* When a program references a file by pathname, the operating system traverses the file system tree to find the filename and inode number in the appropriate directory.

Once it has the inode number, the operating system can determine other information about the file by accessing the inode.

# Directory implementation

A directory implementation that contains only names and inode numbers has the following advantages.

1. **Changing the filename** requires changing only the directory entry. A file can be moved from one directory to another just by moving the directory entry, as long as the move keeps the file on the same partition.
2. **Only one physical copy** of the file needs to exist on disk, but the file may have several names or the same name in different directories. Again, all of these references must be on the same physical partition.
3. Directory entries are of variable length because the filename is of variable length. Directory entries are small, since most of the information about each file is kept in its inode. Manipulating small variable-length structures can be done efficiently. **The larger inode structures are of fixed length.**

# Links

---

UNIX directories have two types of links—links and symbolic links

- A link is an association between a filename and an inode, sometimes called a hard link,
- A symbolic link, sometimes called a soft link, is a file that stores a string used to modify the pathname when it is encountered during pathname resolution
- Each inode contains a count of the number of hard links to the inode.
- When a file is created, a new directory entry is created and a new inode is assigned.
- Additional hard links can be created with

`ln newname oldname`

or with

```
#include <unistd.h>  
  
int link(const char *oldpath, const char *newpath);
```

# Links

---

A new hard link to an existing file creates a new directory entry but assigns no other additional disk space.

- A new hard link increments the link count in the inode.
- A hard link can be removed with the rm command or the unlink system call:

```
#include <unistd.h>
int unlink(const char *pathname);
```

- These decrement the link count.
- The inode and associated disk space are freed when the count is decremented to 0.

# Links

A symbolic link is a special type of file that contains the name of another file.

- A reference to the name of a symbolic link causes the operating system to use the name stored in the file, rather than the name itself.
- Symbolic links are created with the command:

```
ln -s newname oldname or
```

```
#include <unistd.h>  
int symlink(const char *target, const char *linkpath);
```

- Symbolic links do not affect the link count in the inode.
- Unlike hard links, symbolic links can span filesystems.

# Links

```
$echo "cat" > file1          // 3 bytes + \n = 4 bytes
$touch "dog" > file2
$ls -li
4986415 -rw-r--r--  1 erhan erhan          4 Mar  7 10:02 file1
4986713 -rw-r--r--  1 erhan erhan          4 Mar  7 10:03 file2
$ln file1 link
$ln -s file2 slink
$ls -li
4986415 -rw-r--r--  2 erhan erhan          4 Mar  7 10:02 link
4985804 lrwxrwxrwx  1 erhan erhan          5 Mar  7 10:06 slink -> file2
4986415 -rw-r--r--  2 erhan erhan          4 Mar  7 10:02 file1
4986713 -rw-r--r--  1 erhan erhan          4 Mar  7 10:03 file2
$readlink slink
file2
$ rm file1; ls -li link          // or unlink instead of rm
4986415 -rw-r--r--  1 erhan erhan 4 Mar  7 10:02 link
```

# Temporary files

Some programs need to create temporary files that are used only while the program is running, and these files should be removed when the program terminates.

The `mkstemp` call generates a unique filename based on a template supplied by the caller and opens the file, returning a file descriptor that can be used with I/O system calls.

```
#include <stdlib.h>
int mkstemp(char * template);
```

**Returns** the file descriptor on success, or -1 on error

Typically, a temporary file is unlinked (deleted) soon after it is opened, using the `unlink` system call

# Temporary files

---

```
int fd;
char template[] = "/tmp/somestringXXXXXX";

fd = mkstemp(template);
if (fd == -1)
    errExit("mkstemp");
printf("Generated filename was: %s\n", template);
unlink(template);      /* Name disappears immediately, but the file
                           is removed only after close() */

/* Use file I/O system calls - read(), write(), and so on */

if (close(fd) == -1)
    errExit("close");
```

# File stats

You can access the inode information through the `stat` calls

```
#include <sys/stat.h>

int stat(const char *pathname, struct stat *statbuf);
int lstat(const char *pathname, struct stat *statbuf);
int fstat(int fd, struct stat *statbuf);
```

All return 0 on success, or -1 on error

`stat` works with filenames

`fstat` works with file descriptors

`lstat` is similar to `stat`, except that if the named file is a symbolic link, information about the link itself is returned, rather than the file to which the link points

# File stats

```
struct stat {  
    dev_t      st_dev;          /* IDs of device on which file resides */  
    ino_t      st_ino;          /* I-node number of file */  
    mode_t     st_mode;          /* File type and permissions */  
    nlink_t    st_nlink;         /* Number of (hard) links to file */  
    uid_t      st_uid;          /* User ID of file owner */  
    gid_t      st_gid;          /* Group ID of file owner */  
    dev_t      st_rdev;          /* IDs for device special files */  
    off_t      st_size;          /* Total file size (bytes) */  
    blksize_t  st_blksize;        /* Optimal block size for I/O (bytes) */  
    blkcnt_t   st_blocks;         /* Number of (512B) blocks allocated */  
    time_t     st_atime;          /* Time of last file access */  
    time_t     st_mtime;          /* Time of last file modification */  
    time_t     st_ctime;          /* Time of last status change */  
};
```

# File stats

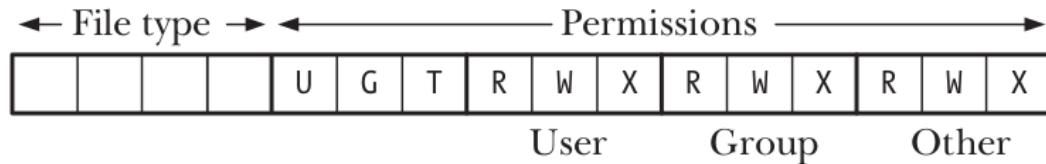


Figure 15-1: Layout of *st\_mode* bit mask

The file type can be extracted by AND'ing (&) with the constant **S\_IFMT**.

```
if ((statbuf.st_mode & S_IFMT) == S_IFREG)
printf("regular file\n")
```

But since this is a common operation there are macros for it.

# File stats

**Table 15-1:** Macros for checking file types in the *st\_mode* field of the *stat* structure

Constant	Test macro	File type
S_IFREG	S_ISREG()	Regular file
S_IFDIR	S_ISDIR()	Directory
S_IFCHR	S_ISCHR()	Character device
S_IFBLK	S_ISBLK()	Block device
S_IFIFO	S_ISFIFO()	FIFO or pipe
S_IFSOCK	S_ISSOCK()	Socket
S_IFLNK	S_ISLNK()	Symbolic link

```
if (S_ISREG(statbuf.st_mode))  
    printf("regular file\n");
```

# File stats

---

```
#include <unistd.h>
#include <stdio.h>
#include <sys/stat.h>
#include <sys/types.h>

int main(int argc, char **argv)
{
    if(argc != 2) return 1;

    struct stat fileStat;
    if(stat(argv[1],&fileStat) < 0) return -1;

    printf("Information for %s\n",argv[1]);
    printf("-----\n");
    printf("File Size: \t\t%d bytes\n",fileStat.st_size);
    printf("Number of Links: \t%d\n",fileStat.st_nlink);
```

# File stats

```
printf("File inode: \t\t%d\n", fileStat.st_ino);

printf("File Permissions: \t");
printf( (S_ISDIR(fileStat.st_mode)) ? "d" : "-");
printf( (fileStat.st_mode & S_IRUSR) ? "r" : "-");
printf( (fileStat.st_mode & S_IWUSR) ? "w" : "-");
printf( (fileStat.st_mode & S_IXUSR) ? "x" : "-");
// similarly for GRP and OTH
printf("\n\n");

printf("The file %s a symbolic link\n", (S_ISLNK(fileStat.st_mode))
? "is" : "is not");

return 0;
}
```

# File stats

---

```
$ ./testProgram testfile.sh
```

Information for testfile.sh

```
-----  
File Size:          36 bytes  
Number of Links:    1  
File inode:         180055  
File Permissions:   -rwxr-xr-x
```

The file is not a symbolic link

# File stats

```
[root@desktop /root] # ls -l  
total 558414
```

Disk usage in terms of data  
blocks

type	access modes	# of links	owner	group	size (bytes)	modification date and time	name
d	rwxr-xr-x	5	root	root	1024	Dec 23 13:48	GNUstep
-	rw-r--r--	1	root	root	331	Feb 11 10:19	Xrootenv.0
-	rw-rw-r--	1	root	root	490	Jan 6 15:07	audio.cddb
-	rw-r--r--	1	root	root	45254876	Jan 6 15:08	audio.wav
d	rwxr-xr-x	2	root	root	1024	Feb 20 16:41	axhome
-	rw-r--r--	1	root	root	900	Jan 18 20:15	conf
d	rwxr-xr-x	2	root	root	1024	Dec 25 10:03	corel
-	rw-r--r--	1	root	root	915	Jan 18 20:57	firewall
d	rwxrwxr-x	2	root	root	1024	Jan 6 15:42	linux
d	rwx-----	2	root	root	1024	Jan 4 02:19	mail
d	rwxr-xr-x	3	root	root	1024	Jan 4 01:49	mirror
-	rwxr--r--	1	root	root	29	Dec 27 15:07	openn
d	rwxr-xr-x	3	root	root	1024	Dec 26 13:24	scan
d	rwxrwxr-x	3	root	root	1024	Jan 4 02:34	sniff

# Relationship between open files and processes

---

There is no one-to-one correspondence between file descriptors and open files. It is possible-and useful-to have multiple descriptors referring to the same open file. These file descriptors may be open in the same process or in different processes.

The kernel maintains 3 data structures

- the per-process file descriptor table (with fd flags);
- the system-wide table of open file descriptions (offset, inode, signal settings, etc)
- the file system i-node table (file type, owner, location, etc)

# Relationship between open files and processes

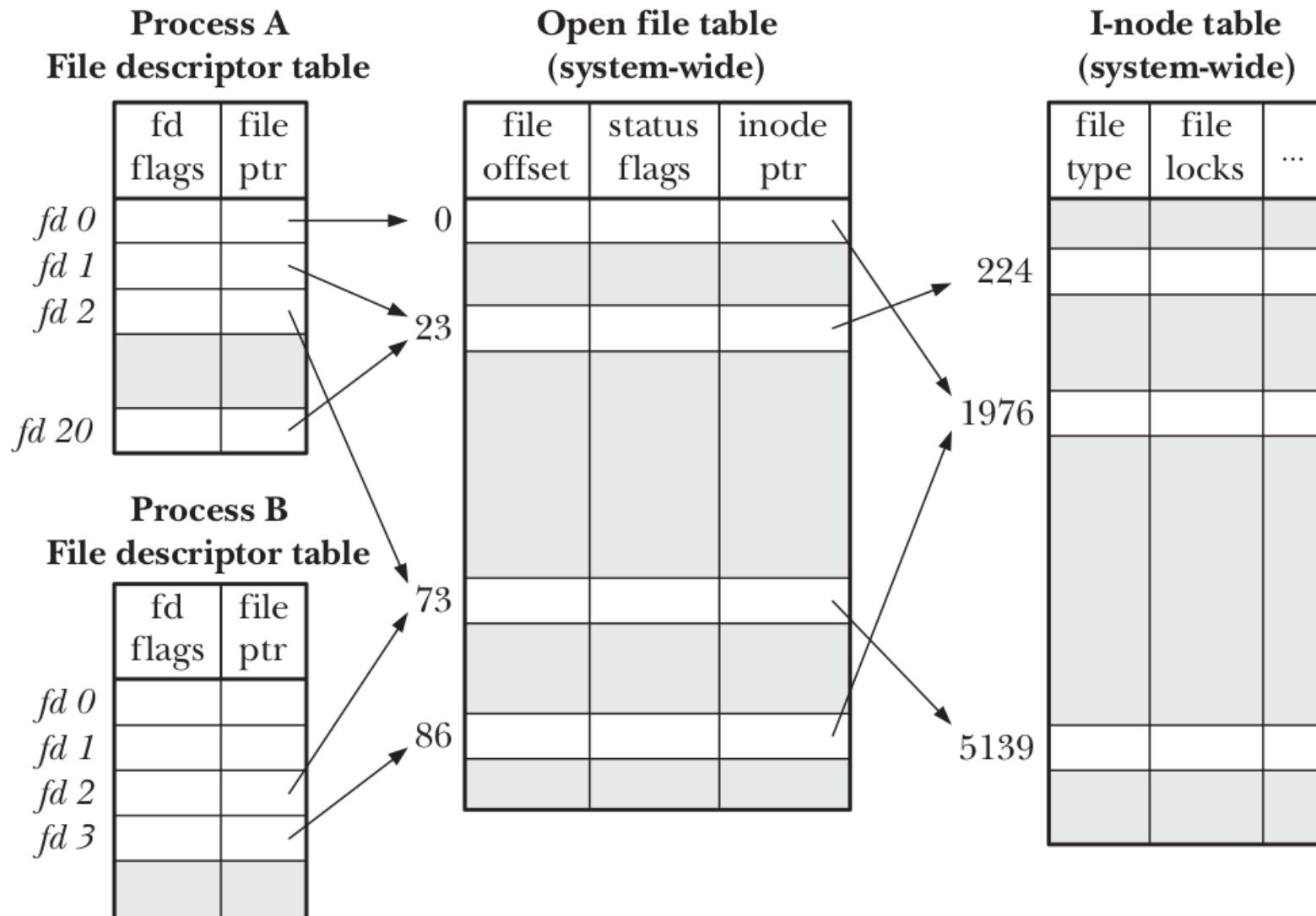


Figure 5-2: Relationship between file descriptors, open file descriptions, and i-nodes

# Relationship between open files and processes

Process A has two fds referring to the same file #23 (and offset) due to:

- a call to `dup()` or `fcntl()` for duplicating a fd.

Process A and B have two fds referring to the same file #73 (and offset):

- because probably A and B have a parent child relationship, and `fork()` clones the parent process along with its fd table, but both fds (2 and 2) point to the same file description entry, and hence same offset!

Process A has a fd (0), and process B has a fd (3), with distinct **offsets** that however point to the same file (i.e. same i-node: 1976):

- either because A and B called `open()` independently on the same file
- or it can also happen when the same process calls twice `open()` on the same file.

# Buffering

When working with disk files, the `read()` and `write()` system calls don't directly initiate disk access. Instead, they simply copy data between a user-space buffer and a buffer in the kernel buffer cache. For example, the following call transfers 3 bytes of data from a buffer in user-space memory to a buffer in kernel space:

```
write(fd, "abc", 3);
```

At this point, `write()` returns. At some later point, the kernel writes (flushes) its buffer to the disk. (Hence, we say that the system call is not synchronized with the disk operation.) If, in the interim, another process attempts to read these bytes of the file, then the kernel automatically supplies the data from the buffer cache, rather than from (the outdated contents of) the file (a similar scenario is valid for `read`).

# Buffering

---

The kernel performs the same number of disk accesses, regardless of whether we perform 1000 writes of a single byte or a single write of a 1000 bytes. However, the latter is preferable, since it requires a single system call, while the former requires 1000. Although much faster than disk operations, system calls nevertheless take an appreciable amount of time, since the kernel must trap the call, check the validity of the system call arguments, and transfer data between user space and kernel space.

Let's look at the effect of the buffer size on duplicating a large file.

# Buffering

**Table 13-1:** Time required to duplicate a file of 100 million bytes

BUF_SIZE	Time (seconds)			
	Elapsed	Total CPU	User CPU	System CPU
1	107.43	107.32	8.20	99.12
2	54.16	53.89	4.13	49.76
4	31.72	30.96	2.30	28.66
8	15.59	14.34	1.08	13.26
16	7.50	7.14	0.51	6.63
32	3.76	3.68	0.26	3.41
64	2.19	2.04	0.13	1.91
128	2.16	1.59	0.11	1.48
256	2.06	1.75	0.10	1.65
512	2.06	1.03	0.05	0.98
1024	2.05	0.65	0.02	0.63
4096	2.05	0.38	0.01	0.38
16384	2.05	0.34	0.00	0.33
65536	2.06	0.32	0.00	0.32

# Buffering and File pointers

---

Buffering of data into large blocks to reduce system calls is exactly what is done by the C library I/O functions (e.g., `fprintf()`, `fscanf()`, `fgets()`, `fputs()`, `fputc()`, `fgetc()`) when operating on disk files. Thus, using the `stdio` library relieves us of the task of buffering data for output with `write()` or input via `read()`.

Be careful as these higher level functions handle files in terms of pointers to `FILE` structures.

# the FILE structure

---

```
typedef struct {  
    char *fpos; // Current position of file pointer (absolute address)  
    void *base; /* Pointer to the base of the file */  
    unsigned short handle; /* File handle */  
    short flags; /* Flags (see FileFlags) */  
    short unget; /* 1-byte buffer for ungetc (b15=1 if non-empty) */  
    unsigned long alloc; // # of currently allocated bytes for the file  
    unsigned short buffincrement; /* # of bytes allocated at once */  
} FILE;
```

# Buffering

The `setvbuf()` function controls the form of buffering employed by the stdio library.

```
#include <stdio.h>

int setvbuf(FILE * stream , char * buf , int mode , size_t size
);
```

**Returns** 0 on success, or nonzero on error. Valid for all subsequent stdio operations.

`stream` : the stream upon which the buffering will be applied

`buf` : the buffer

`size` : size of the buffer in bytes

# Buffering

---

mode can be one of the following:

`_IONBF`: no buffering, immediate reads/writes, default of `stderr`

`_IOLBF`: Employ line-buffered I/O. This flag is the default for streams referring to terminal devices. For output streams, data is buffered until a newline character is output (unless the buffer fills first). For input streams, data is read a line at a time.

`_IOFBF`: Employ fully buffered I/O. Data is read or written (via calls to `read()` or `write()`) in units equal to the size of the buffer. This mode is the default for streams referring to disk files.

# Buffering

---

```
#define BUF_SIZE 1024
static char buf[BUF_SIZE];

if (setvbuf(stdout, buf, _IOFBF, BUF_SIZE) != 0)
    errExit("setvbuf");
```

# Buffering

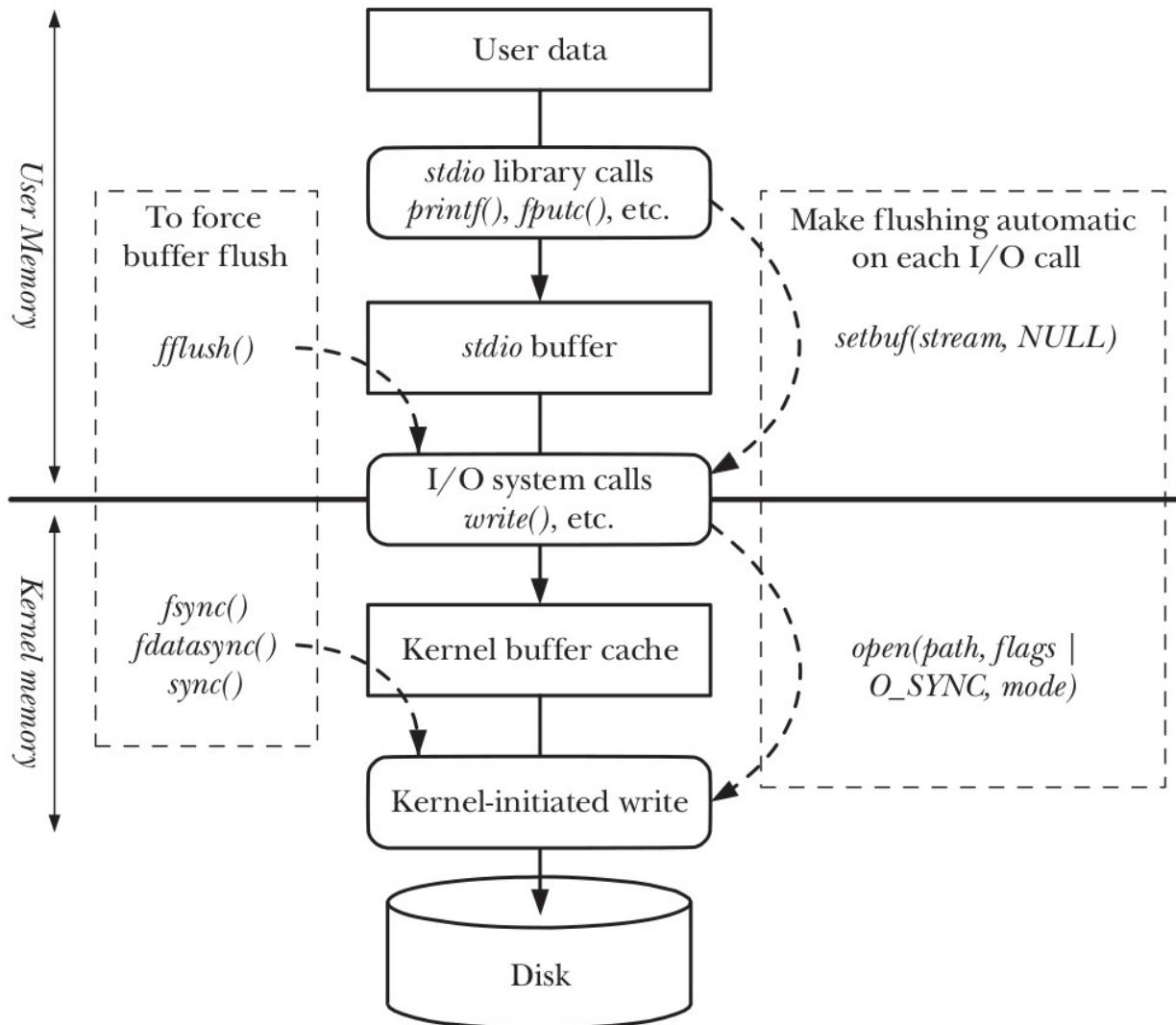
---

Regardless of the current buffering mode, at any time, we can force the data in a stdio output stream to be written (i.e., flushed to a kernel buffer via `write()`) using the `fflush()` library function. This function flushes the output buffer for the specified stream.

```
#include <stdio.h>
int fflush(FILE * stream);
```

Returns 0 on success, EOF on error

# Buffering



**O\_SYNC** guarantees that the call will not return before all data has been transferred to the disk (as far as the OS can tell). This still does not guarantee that the data isn't somewhere in the harddisk write cache, but it is as much as the OS can guarantee.

Figure 13-1: Summary of I/O buffering

# BIL 344 System Programming

Week 4

---

## *Signals and processes*

- *Signal fundamentals*
- *Signal blocking, ignoring, handling*
- *Creating a new process*
- *Terminating a process*

# Signals

Signals are mechanisms for communicating with, and manipulating processes.

A signal is a notification sent to a process that an event has occurred.

Think of them as **software interrupts**. They are similar to hardware interrupts in the sense that they interrupt the normal flow of execution.

It is **impossible** to predict exactly when a signal is going to arrive.

Examples: dividing by zero, referencing an inaccessible memory location, pressing CTRL-C...all lead to **signal generation**.

# Signals

---

Signals are **asynchronous**; when a process receives a signal, it processes the signal immediately, without finishing the current function or even the current line of code.

When a signal is **delivered** to a process, the process can:

- Ignore it
- Take the default action associated with that signal: termination, suspension of execution, core dump file generation, etc.
- Execute a **signal handler**; i.e. a custom function written by you, that takes appropriate action; e.g. in case of CTRL-C, make sure all data/settings are saved properly.

# Signals

---

If a signal handler is used, the currently executing program is paused, the signal handler is executed, and, when the signal handler returns, the program resumes from where it left off.

Every signal is associated with a unique integer number (starting from 1) and is referenced by a symbolic name (defined in **signal.h**) that starts with SIG; SIGSEGV, SIGTERM, SIGINT, etc.

Make sure you include **signal.h** in your source files when working with signals.

SIGUSR1 and SIGUSR2 are reserved for user use.

# Signal commands

## sigint

# Signals

<i>signal</i>	<i>description</i>	<i>default action</i>
SIGABRT	process abort	implementation dependent
SIGALRM	alarm clock	abnormal termination
SIGBUS	access undefined part of memory object	implementation dependent
SIGCHLD	child terminated, stopped or continued	ignore
SIGCONT	execution continued if stopped	continue
SIGFPE	error in arithmetic operation as in division by zero	implementation dependent
SIGHUP	hang-up (death) on controlling terminal (process)	abnormal termination
SIGILL	invalid hardware instruction	implementation dependent
SIGINT	interactive attention signal (usually Ctrl-C)	abnormal termination
SIGKILL	terminated (cannot be caught or ignored)	abnormal termination
SIGPIPE	write on a pipe with no readers	abnormal termination
SIGQUIT	interactive termination: core dump (usually Ctrl-\)	implementation dependent
SIGSEGV	invalid memory reference	implementation dependent
SIGSTOP	execution stopped (cannot be caught or ignored)	stop
SIGTERM	termination	abnormal termination
SIGTSTP	terminal stop	stop
SIGTTIN	background process attempting to read	stop
SIGTTOU	background process attempting to write	stop
SIGURG	high bandwidth data available at a socket	ignore
SIGUSR1	user-defined signal 1	abnormal termination
SIGUSR2	user-defined signal 2	abnormal termination

# Signals

Signals can be sent by the kernel, by other processes, or even by the user using the `kill` system call (also available as a shell command):

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

Returns 0 on success, and -1 on error and sets `errno`.

The name `kill` derives from the fact that historically, many signals have the default action of terminating the process.

- ✓ `pid > 0` signal sent to the process of that pid
- ✓ `pid == -1` sent to all processes for which it has permission to send

# Signals

If no process matches the specified pid, `kill()` fails and sets `errno` to `ESRCH` (“No such process”).

A process needs appropriate permissions to be able to send a signal to another process; you cannot go around killing other users’ processes.

- The init process (pid 1) is special; it can only be sent signals for which it has a handler installed; this prevents accidental kills.
- For all others, a process can send another process a signal if their user ids match (there are some intricacies involved).
- `SIGCONT` is an exception; (can be sent to any process in the same session).

# Signals

---

Tip: you can use the `kill` system call to send the **null signal** (0) to test for the existence of a specific pid.

It will either send no signal (successfully since the null signal does not exist) or return with ESRCH (no such process).

A process can also send a signal to itself, either through

```
kill(getpid(), sig);
```

or through the `raise` system call

```
#include <signal.h>
int raise(int sig);
```

## for example: Blocking Ctrl C signal **Signal mask** → Dominating

For each process (in fact **for each thread** – the kernel sends the signal to a random thread if none block it), the kernel maintains a signal mask - a set of signals whose delivery to the process is currently blocked. If a signal that is blocked is sent to a process, delivery of that signal is delayed until it is unblocked by being removed from the process signal mask.

If delays that signal

```
#include <signal.h>
```

```
int sigprocmask(int how, const sigset_t * set, sigset_t * oldset);
```

Returns 0 on success, or -1 on error

→ with this we can add or remove other signals.

The `sigprocmask()` system call can be used at any time to explicitly add signals to, and remove signals from the signal mask.

# Signal mask

```
#include <signal.h>  
int sigprocmask(int how, const sigset_t* set, sigset_t* oldst);
```

how: how should the mask be changed

- SIG\_BLOCK: the signals in set are added into the signal mask
- SIG\_UNBLOCK: the signals in set are removed from the signal mask
- SIG\_SETMASK: set becomes the signal mask

Some signals, such as SIGSTOP and SIGKILL, cannot be blocked. If an attempt is made to block these signals, the system ignores the request without reporting an error.

# Signal set

And what about the `sigset_t` type? Easy to manipulate:

```
#include <signal.h>
int sigaddset(sigset_t *set, int signo);
```

Adds `signo` into the set

```
int sigdelset(sigset_t *set, int signo);
```

Removes `signo` from the set

```
int sigemptyset(sigset_t *set);
```

Initializes a signal set to contain no members

```
int sigfillset(sigset_t *set);
```

Initializes a set to contain all signals

```
int sigismember(const sigset_t *set, int signo);
```

Test membership of `signo` in the set

'intmask' in olduğu hâli hepisi:

## Signal (un)blocking example

```
#include <math.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
/* a program that blocks and unblocks SIGINT */
int main(int argc, char *argv[]) {
    int i;
    sigset_t intmask;
    int repeatfactor;
    double y = 0.0;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s repeatfactor\n", argv[0]);
        return 1;
    }
    repeatfactor = atoi(argv[1]);
    if ((sigemptyset(&intmask) == -1) || (sigaddset(&intmask, SIGINT) == -1))
    {
        perror("Failed to initialize the signal mask");
        return 1;
    }
    for ( ; ; ) {
        if (sigprocmask(SIG_BLOCK, &intmask, NULL) == -1)
            break;
        fprintf(stderr, "SIGINT signal blocked\n");
        for (i = 0; i < repeatfactor; i++)
            y += sin((double)i); /* blocked signals calculations */
        fprintf(stderr, "Blocked calculation is finished, y = %f\n", y);
        if (sigprocmask(SIG_UNBLOCK, &intmask, NULL) == -1)
            break;
        fprintf(stderr, "SIGINT signal unblocked\n");
        for (i = 0; i < repeatfactor; i++)
            y += sin((double)i); /* unblocked signals calculations */
        fprintf(stderr, "Unblocked calculation is finished, y=%f\n", y);
    }
    perror("Failed to change signal mask");
    return 1;
}
```

Sig ismember();

## Pending signals

If a process receives a signal that it is currently blocking, that signal is added to the process's set of pending signals. When (and if) the signal is later unblocked, it is then delivered to the process. To determine which signals are pending for a process, we can call

```
#include <signal.h>
int sigpending(sigset_t * set );
```

Returns 0 on success, or -1 on error

We can then examine set using `sigismember()`

The set of pending signals is only a mask; it indicates whether or not a signal has occurred, but not how many times it has occurred. **Pending signals DO NOT queue.**

It will be blocked from beginning of it to the end of it.

## Signal handlers

Handling signals either by catching or ignoring them, is done through the `sigaction` call:

```
#include <signal.h>
int sigaction(int signo, const struct sigaction *act, struct
sigaction *oldact);
```

`signo`: signal number for action

`act`: the action to take

`oldact`: receives the previous action towards `signo`

When a handler function is invoked on a signal, that signal is automatically blocked (in addition to any other signals that are already in the process's signal mask) during the time the handler is running.

# Signal handlers

```
struct sigaction {  
    //SIG_DFL, SIG_IGN or pointer to function  
    void (*sa_handler)(int);  
    // additional signals to be blocked during execution of  
    // handler  
    sigset_t sa_mask;  
    // special flags and options  
    int sa_flags;  
    // obsolete, don't use this  
    void(*sa_restorer) (void);  
};
```

SIG\_DFL: default action

SIG\_IGN: ignore signal

The handler cannot return anything, and receives only signo as param.

# Signal handling example

Listing 3.5 (sigusr1.c) Using a Signal Handler

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

sig_atomic_t sigusr1_count = 0;

void handler (int signal_number)
{
    ++sigusr1_count;
}

int main ()
{
    struct sigaction sa;
    memset (&sa, 0, sizeof (sa));
    sa.sa_handler = &handler;
    sigaction (SIGUSR1, &sa, NULL);

    /* Do some lengthy stuff here. */
    /* ... */

    printf ("SIGUSR1 was raised %d times\n", sigusr1_count);
    return 0;
}
```

*Don't use signd() because it's obsolete. Use sigaction() instead. Doing so will block a signal during handler's execution.*

# Signal handling

The following code segment sets the signal handler for SIGINT to mysighand

```
struct sigaction newact;  
newact.sa_handler = &mysighand; /* set the new handler */  
newact.sa_flags = 0;  
  
/* no special options */  
if ((sigemptyset(&newact.sa_mask) == -1) || (sigaction(SIGINT,  
&newact, NULL) == -1))  
    perror("Failed to install SIGINT signal handler")
```

Always check

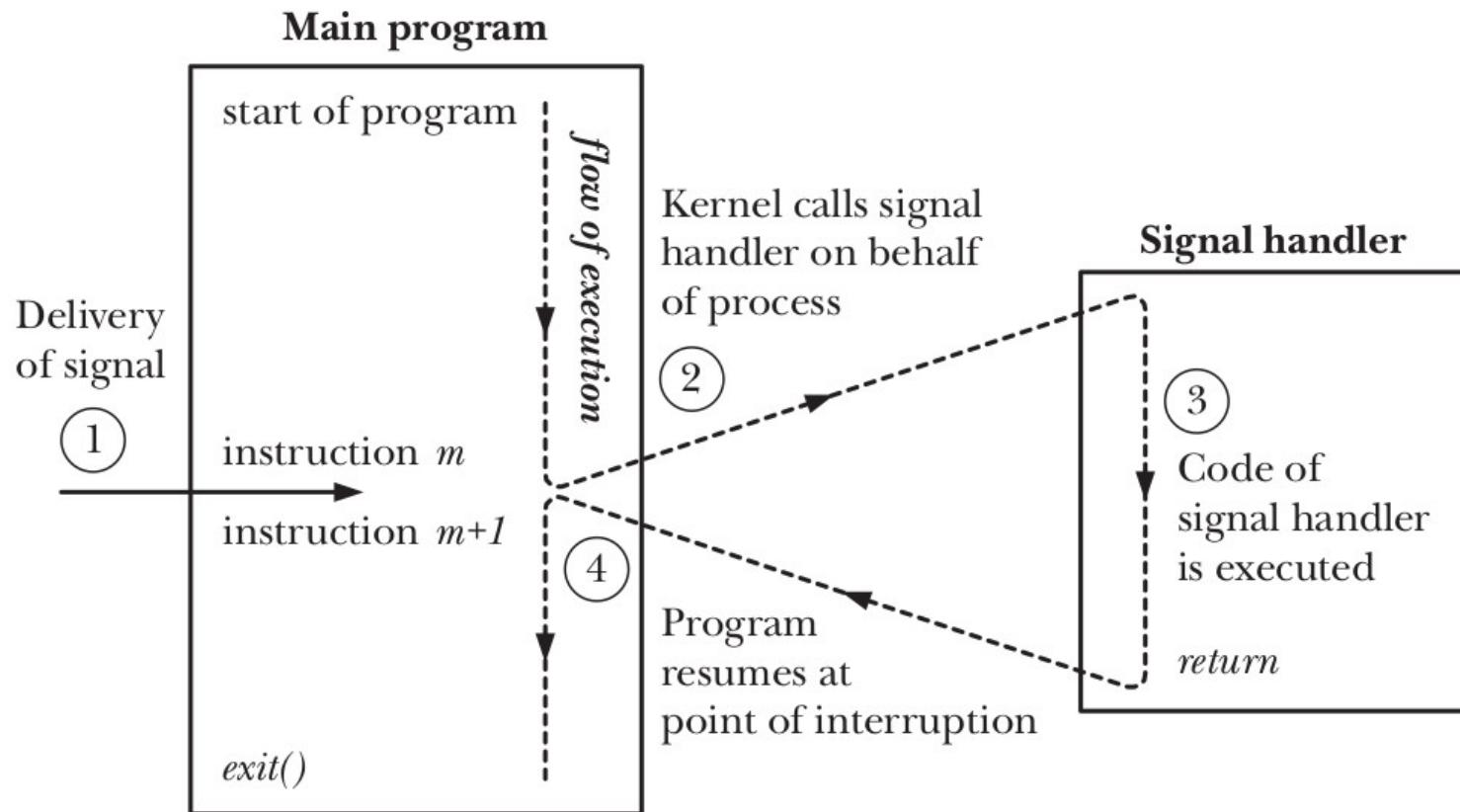
# Signal handling

The following code segment causes the process to ignore SIGINT if the default action is in effect for this signal.

```
struct sigaction act;  
/* Find the current SIGINT handler */  
if (sigaction(SIGINT, NULL, &act) == -1)  
    perror("Failed to get old handler for SIGINT");  
else if (act.sa_handler == SIG_DFL) {    If succeeded  
    /* if SIGINT handler is default */  
    act.sa_handler = SIG_IGN;  
    /* set new SIGINT handler to ignore */    → set ignore  
    if (sigaction(SIGINT, &act, NULL) == -1)  
        perror("Failed to ignore SIGINT");  
}
```

# Signal handling

Invocation of a signal handler may interrupt the main program flow at any time; the kernel calls the handler on the process's behalf, and when the handler returns, execution of the program resumes at the point where the handler interrupted it.



# Signal handling

The `sa_flags` field is a bit mask specifying various options controlling how the signal is handled. The following bits can be OR'ed (|)

`SA_RESTART`: automatically restart system calls interrupted by this signal handler. *Don't rely on it. Doesn't support on all posix systems.*

`SA_SIGINFO`: invoke the signal handler with additional arguments providing further information about the signal.

And more: `SA_RESETHAND`, `SA_NOCLDSTOP`, etc.

Important  
Problem  
(How to synchronize them)

## Waiting for signals

The UNIX signaling mechanism also serves as a way of waiting for an event without **busy waiting**.

Never USE

If use it on hw  
it's fail.

Busy waiting: testing continuously (within a loop) whether a certain event occurred by using CPU cycles.

The alternative is to suspend the process until a signal arrives notifying it that the event occurred (hence the CPU is free).

→ **SIGSUSPEND** ⇒ Suspend until signal arrives

POSIX system calls for suspending processes until a signal occurs: sleep, nanosleep, pause, sigsuspend...

# Waiting for signals

The sleep() function suspends execution of the calling process for the number of seconds specified in the seconds argument or until a signal is caught (thus interrupting the call).

Avoid. (Yeterde yarınlaşsa birşey olabilir.)

```
#include <unistd.h>  
unsigned int sleep(unsigned int seconds );
```

Returns 0 on normal completion, or number of unslept seconds if prematurely terminated

nanosleep is a more powerful version of sleep that operates at higher resolution.

Also avoid

## Waiting for signals

Calling `pause()` suspends execution of the process until the call is interrupted by a signal handler (or until an unhandled signal terminates the process).

```
#include <unistd.h>
int pause(void);
```

Always returns `-1` with `errno` set to `EINTR`

```
for (;;)           /* Loop forever, waiting for signals */
    pause();       /* Block until a signal is caught */
```

```
sigset_t prevMask, intMask;
struct sigaction sa;

sigemptyset(&intMask);
sigaddset(&intMask, SIGINT);

sigemptyset(&sa.sa_mask);
sa.sa_flags = 0;
sa.sa_handler = handler;

if (sigaction(SIGINT, &sa, NULL) == -1)
    errExit("sigaction");

/* Block SIGINT prior to executing critical section. (At this
   point we assume that SIGINT is not already blocked.) */

if (sigprocmask(SIG_BLOCK, &intMask, &prevMask) == -1)
    errExit("sigprocmask - SIG_BLOCK");

/* Critical section: do some work here that must not be
   interrupted by the SIGINT handler */

/* End of critical section - restore old mask to unblock SIGINT */

if (sigprocmask(SIG_SETMASK, &prevMask, NULL) == -1)
    errExit("sigprocmask - SIG_SETMASK");

/* BUG: what if SIGINT arrives now... */

pause();                                     /* Wait for SIGINT */
```

S

Why is pause insufficient?  
Because a signal can arrive  
between unblocking and pause  
and it will be lost

example shows incorrectly  
unblocking and  
waiting for a signal

Critical  
section

/\* Critical section: do some work here that must not be  
interrupted by the SIGINT handler \*/

/\* End of critical section - restore old mask to unblock SIGINT \*/

if (sigprocmask(SIG\_SETMASK, &prevMask, NULL) == -1)  
 errExit("sigprocmask - SIG\_SETMASK");

/\* BUG: what if SIGINT arrives now... \*/

/\* Wait for SIGINT \*/

# Waiting for signals

Mask veriyor.  
Maskenin işaretini alıyor.  
Maskenin işaretini alıyor.  
Maskenin işaretini alıyor.  
Maskenin işaretini alıyor.

The solution: *İmgek信号блоки, сигналері белгілерінің блоки, маске*

```
#include <signal.h>
```

```
int sigsuspend(const sigset_t * mask);
```

Returns -1 with errno set to EINTR

*Thi's way, we can avoid busy waiting.*

**Atomically** unblocks a signal and suspends the process.

Equivalent to (atomic):

```
/* Assign new mask */  
sigprocmask(SIG_SETMASK, &mask, &prevMask);  
pause();  
/* Restore old mask */  
sigprocmask(SIG_SETMASK, &prevMask, NULL);
```

Main idea

Block the process  
efficiently until a  
certain signal arrives and  
is handled.

# Dealing with signals

There are 3 delicate issues when dealing with signals.

## 1) Handling errors that use errno

If errno is set within the signal handler, that could mean that its previous eventually unprocessed value from the main program is lost!  
Solution: in the signal handler, save and restore errno.

```
void myhandler(int signo) {  
    int esaved = errno;  
    // do dangerous stuff that might cause errno to be set.  
    errno = esaved; // once you are done, restore errno's value  
}
```

*Take backup of errno.  
After do dangerous  
stuff, restore  
errno value.*

# Dealing with signals

2) Whether POSIX system calls that are interrupted by signals should be restarted.

A signal arrives in the middle of a system call. It is handled, and now we return to the call. Should it continue, restart, cancel?

By default, the system call will fail with the error EINTR (“Interrupted function”). You can use this feature for manual restarts:

*Use this macro*

```
while ((cnt = read(fd, buf, BUF_SIZE)) == -1 && errno == EINTR)
    continue; /* Do nothing loop body */
if (cnt == -1)
    perror("read");
```

# Dealing with signals

---

If this becomes a major concern you can turn it into a macro too.

```
#define NO_EINTR(stmt) while ((stmt) == -1 && errno == EINTR);  
  
NO_EINTR(cnt = read(fd, buf, BUF_SIZE));  
if (cnt == -1)  
    perror("read"); /* read() failed with other than EINTR */
```

Unfortunately the `SA_RESTART` flag we saw earlier, is not supported by all system calls, for historical reasons.

# Dealing with signals

3) The last issue is about which system calls to call in a signal handler.

System calls that rely on global/static variables and data structures are **not safe** (the majority of stdio calls: `printf`, `scanf`, etc).

→ *Don't use inside a Signal handler. Instead use variable. Exit as soon as possible.*

By **not safe** we mean that if you call them in the signal handler, you no longer have the guarantee that they will behave as expected when you return to the main program.

Unfortunately the list of async-safe functions is relatively short. Almost none of the functions in the **standard C library** are on it.

# Signal handlers

Because signals are asynchronous, the main program may be in a very **fragile state** when a signal handler function executes. Therefore, you should avoid performing any I/O operations or calling most library and system functions from signal handlers.

A signal handler should perform the minimum work necessary to respond to a signal, and then return control to the main program (or terminate the program).

In most cases, this consists simply of recording the fact that a signal occurred. Signals may arrive even while handling them...this is very hard to debug.

# Signal handlers

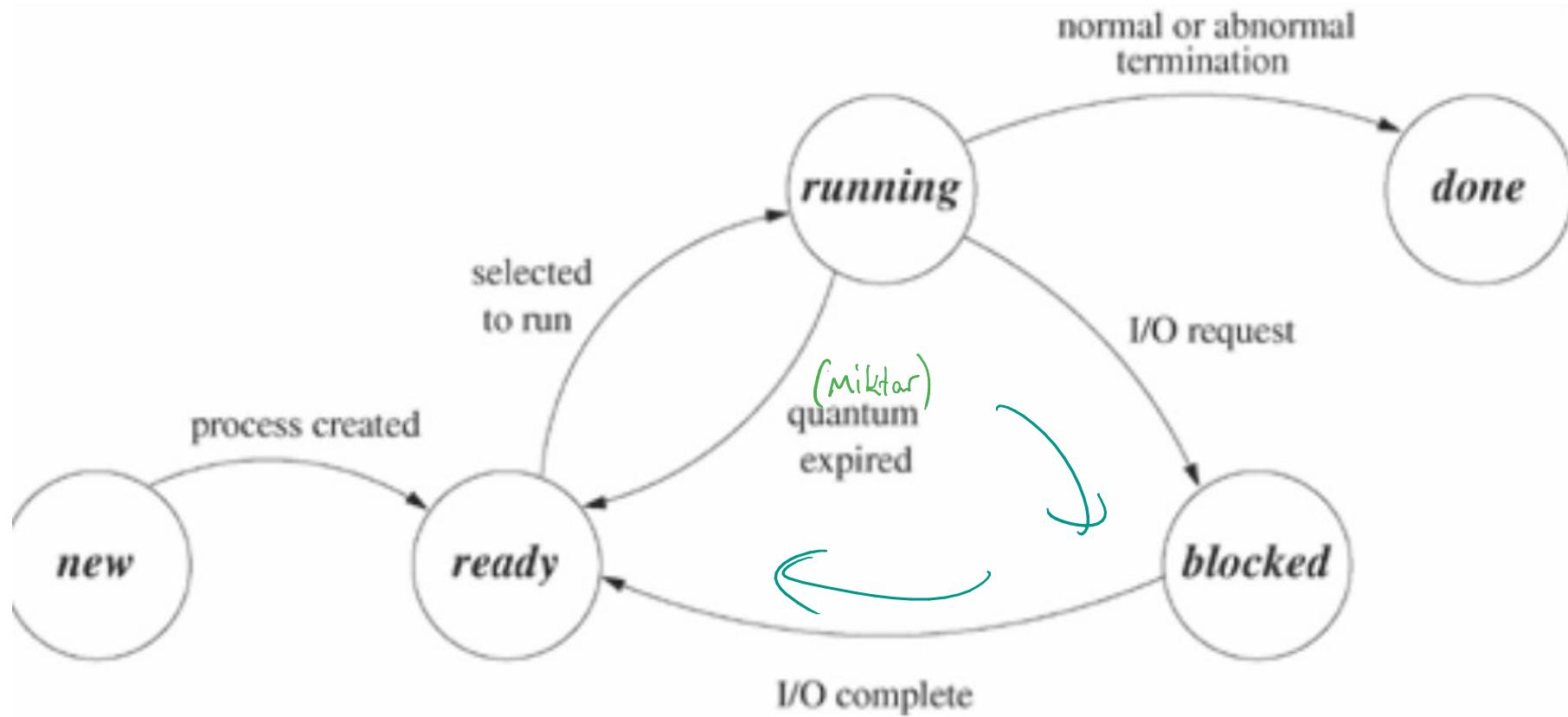
One way of handling this fragile condition in loop based programs is:

```
signal_handler:  
    sets a flag variable  
main:  
loop:  
    do lengthy stuff  
    if flag is set:  
        exit elegantly  
end_loop
```

Just set a flag variable.  
Use the bin flag to control/edict Z.  
Set a flag variable.  
Set a flag variable.  
Set a flag variable.  
Set a flag variable.  
Set a flag variable.

# Process state

The state of a process in a simple operating system



But how **exactly** are processes created?

# Process creation

Two common techniques are used for creating a new process.

The first is relatively simple but should be used sparingly (or rather never) because it is inefficient and has considerable security risks.

The second technique is more complex but provides greater flexibility, speed, and security.

# Process creation

The `system` function **in the standard C library** provides an easy way to execute a command from within a program, much as if the command had been typed into a shell.

In fact, `system` creates a subprocess running the standard Bourne shell (`/bin/sh`) and hands the command to that shell for execution.

```
#include <stdlib.h>  
int system(const char * command );
```

*Function in C, not a system call*

It returns the exit status of the shell command. If the shell itself cannot be run, it returns 127; if another error occurs, it returns -1.

# Process creation

Listing 3.2 (*system.c*) Using the *system* Call

---

```
#include <stdlib.h>

int main ()
{
    int return_value;
    return_value = system ("ls -l /");
    return return_value;
}
```

---

Don't use it, it's slow.

**Pros:** it's simple, error & signal handling are taken care of.

**Cons:** a) inefficient since it creates 2 processes, one for the shell, and one more for the commands b) it's subject to the features, limitations, and **security flaws** of the system's shell. Since you can't rely on the **availability** of any particular shell, it's not portable either.

# Process creation

The windows API has the spawn family of functions to create new processes by being given just the name of the program to run.

POSIX

In the UNIX world the creation of a new process is done in **2 steps!**

Amas bir process'in başka process'ı galistirmasi,

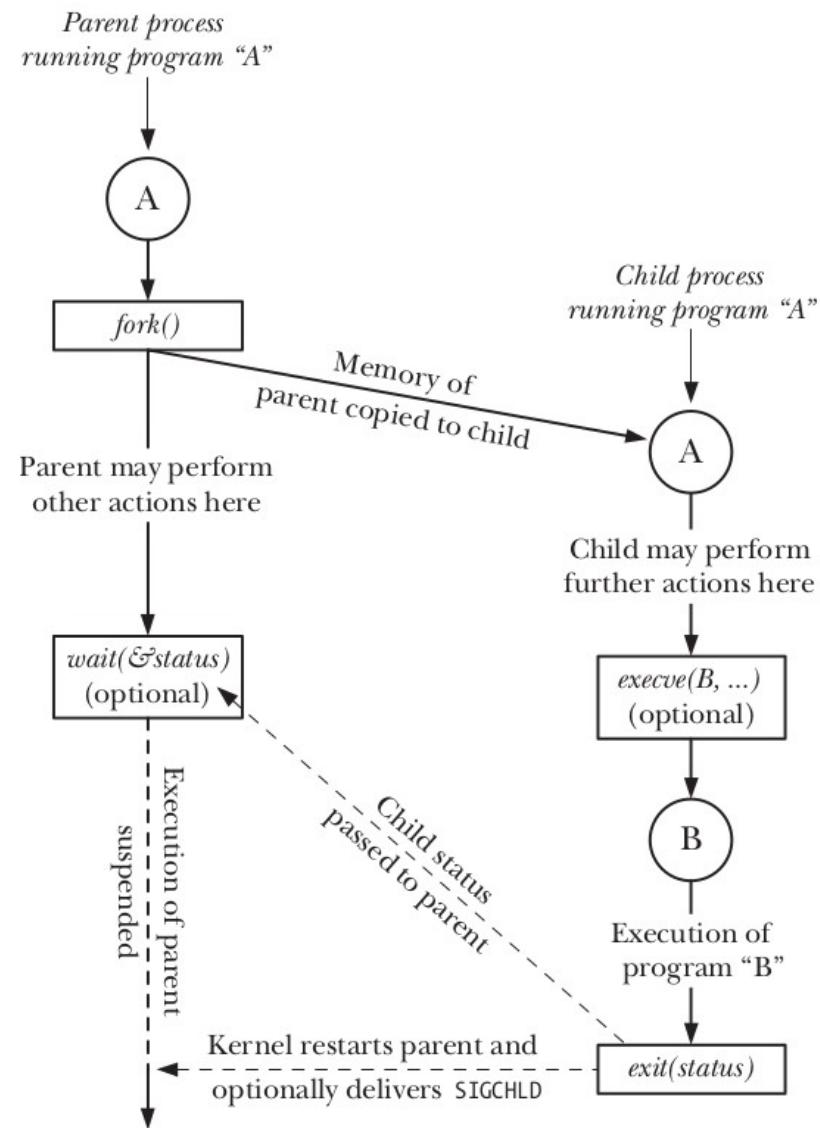
- 1) fork: that makes a child process that is an exact copy of its parent process klon yaratma
- 2) exec: causes a particular process to cease being an instance of one program and to instead become an instance of another program Sonlandırma

exec ve

exec vb

i

# Process creation



# Process creation

Why create multiple instances of the same process? In many applications, it can be a useful way of dividing up a task.

For example, a network server process may listen for incoming client requests and create a new child process to handle each request; meanwhile, the server process continues to listen for further client connections.

Dividing tasks up in this way often makes application design simpler. It also permits greater concurrency (i.e., more tasks or requests can be handled simultaneously).

This is accomplished through `fork`

*expensive (slow)*  
Because that instead of creating processes, we create threads.

# Process creation

```
#include <unistd.h>
pid_t fork(void);
```

In parent: returns process id of child on success, or  $-1$  on error;  
in successfully created child: always returns 0

The key point to understanding `fork()` is to realize that after it has completed its work, two processes exist, and, in each process, execution continues from the point where `fork()` returns.

Because no process ever has a process id of zero, this makes it easy for the program to know whether it is now running as the parent or the child process.

# Process creation

Listing 3.3 (*fork.c*) Using *fork* to Duplicate a Program's Process

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main ()
{
    pid_t child_pid;

    printf ("the main program process ID is %d\n", (int) getpid ());

    child_pid = fork (); → If 0, then we are in child process.
    if (child_pid != 0) {
        printf ("this is the parent process, with id %d\n", (int) getpid ());
        printf ("the child's process ID is %d\n", (int) child_pid);
    } → If it enters here, we are in parent.
    else
        printf ("this is the child process, with id %d\n", (int) getpid ());

    return 0;
}
```

# Process creation

The child process is created as an exact copy of its parent, except for its process id. This means a copy of everything: data, heap and stack.

Same file descriptor.

What about open files? The child process receives duplicates of all of the parent's file descriptors. The descriptors in the parent and the child refer to the same open files. The open file description contains the current file offset (as modified by read(), write(), and lseek()) and the open file status flags (set by open()).

Consequently, these attributes of an open file are shared between the parent and child. For example, if the child updates the file offset, this change is visible through the corresponding descriptor in the parent.

# Process creation

```
#include <stdio.h>

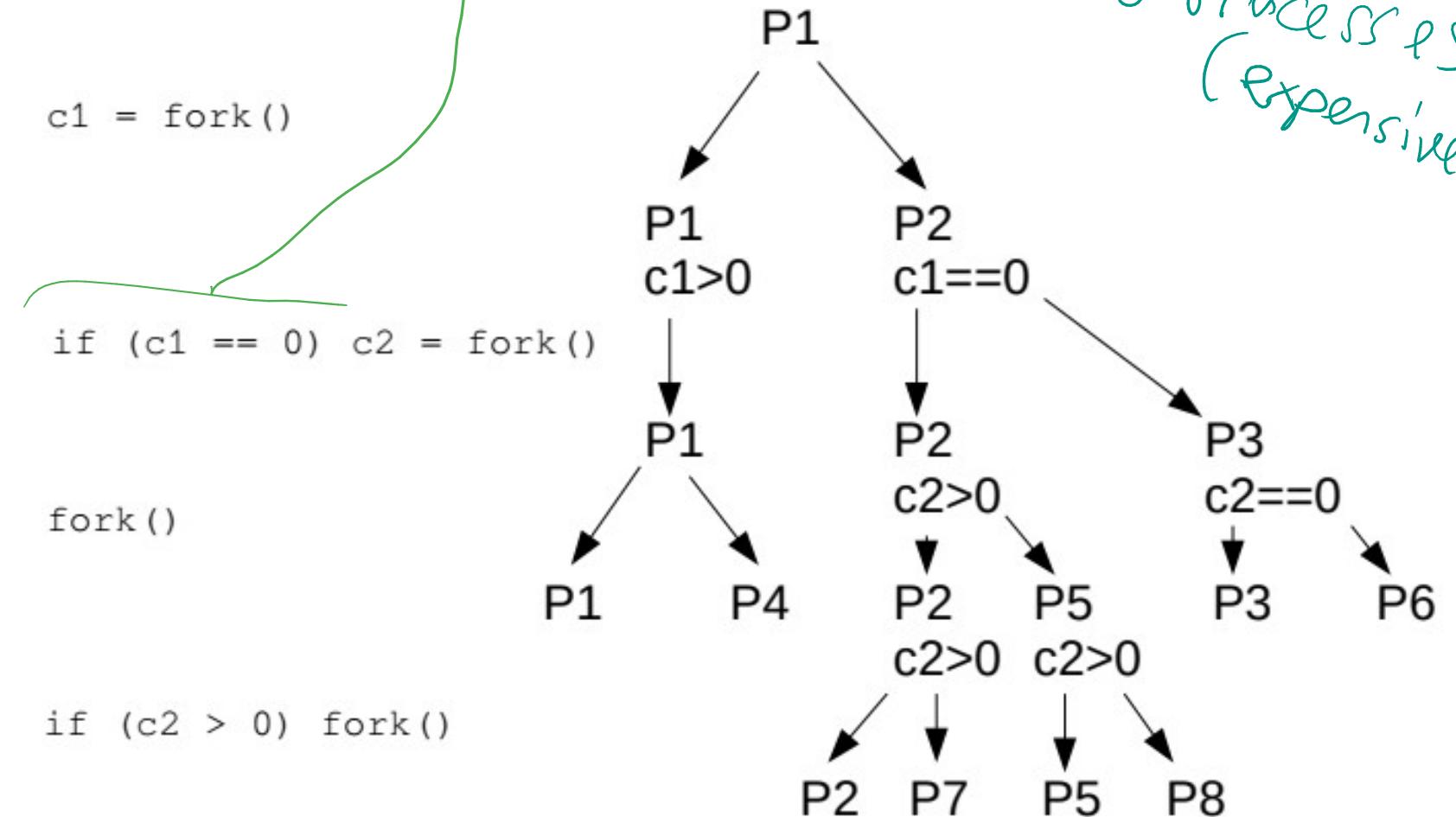
int main(void) {
    pid_t c1, c2; /* process ids */
    c2 = 0;
    c1 = fork(); /* fork number 1 */
    if (c1 == 0) c2 = fork(); /* fork number 2 */
    fork(); /* fork number 3 */
    if (c2 > 0) fork(); /* fork number 4 */
}
```

*it's only in child process*

**How many processes have been formed ?? By which process ??**

# Process creation

If we didn't have if we would have  $2^4 = 16$  processes (expensive)



We don't know if child or parent will execute first.

## Process scheduling

After a `fork()`, it is **indeterminate** which process - the parent or the child - next has access to the CPU. (On a multiprocessor system, they may both simultaneously get access to a CPU.)

Applications that implicitly or explicitly rely on a particular sequence of execution in order to achieve correct results are **open to failure** due to race conditions.

Such bugs can be hard to find, as their occurrence depends on scheduling decisions that the kernel makes according to system load.

Use the `nice` command to control the “priority” of processes.

# Process termination

---

**Normally**, a process terminates in one of two ways.

Either the executing program calls the `exit` function, or the program's main function returns. Each process has an exit code: a number that the process returns to its parent. The exit code is the argument passed to the `exit` function, or the value returned from `main`.

```
#include <stdlib.h>
void exit(int status);
```

By convention

- 0 status: successful completion
- < 0 status: failure
- > 0 status: their meaning is application specific

# Process termination

With most shells, it's possible to obtain the exit code of the most recently executed program using the special `$?` variable. Example:

```
% ls /
bin  coda  etc  lib        misc  nfs  proc  sbin  usr
boot  dev  home  lost+found  mnt  opt  root  tmp  var
% echo $?
0
% ls bogusfile
ls: bogusfile: No such file or directory
% echo $?
1
```

# Process termination

Calling `exit` leads to

- execution of exit handlers
- flushing of all stdio buffers

An exit handler is a function to be executed when the process terminates. It is registered through `atexit`:

```
#include <stdlib.h>  
int atexit(void (* func )(void));
```

*free any memory that's allocated.  
execute the function given  
parameter.*

More generally, at process termination:

- All open files and streams are closed
- All file locks held by the process are released

# Process termination

The second way of termination is the **abnormal** way, in response to a signal such as SIGSEGV, SIGBUS, SIGINT, SIGTERM and SIGABRT.

Their default disposition is to terminate the target process.

The most powerful termination signal is SIGKILL which ends a process immediately and cannot be blocked or handled by a program.

*If it wants to terminate, terminates*

# Example

---

```
/* This example replaces the signal mask and then suspends execution. */
#define _POSIX_SOURCE
#include <stdio.h>
#include <signal.h>
#include <time.h>
#include <unistd.h>

void catcher(int signum) {
    switch (signum) {
        case SIGUSR1: puts("catcher caught SIGUSR1");
                       break;
        case SIGUSR2: puts("catcher caught SIGUSR2");
                       break;
        default:      printf("catcher caught unexpected signal %d\n",
                           signum);    // bad idea to call printf in handler
    }
}
```

## Example continued

```
int main() {  
    sigset_t sigset;  
    struct sigaction sact;  
    time_t t;  
  
    if (fork() == 0) {  
        sleep(10); → Make sure that child process get chance to execute  
        puts("child is sending SIGUSR2 signal - which should be blocked");  
        kill(getppid(), SIGUSR2);  
        sleep(5);  
        puts("child is sending SIGUSR1 signal - which should be caught");  
        kill(getppid(), SIGUSR1); → Send this process  
        exit(0);  
    }  
}
```

*(There is no guarantee one will happen before other. If you want guarantee, you need to use synchronization)*

# Example continued

```
sigemptyset(&sact.sa_mask);
sact.sa_flags = 0;
sact.sa_handler = catcher;
if (sigaction(SIGUSR1, &sact, NULL) != 0)
    perror("1st sigaction() error");
else if (sigaction(SIGUSR2, &sact, NULL) != 0)
    perror("2nd sigaction() error");
else {
    sigfillset(&sigset);           Sets and removes
    sigdelset(&sigset, SIGUSR1);
    time(&t);
    printf("parent waiting for child to send SIGUSR1 at %s", ctime(&t));
    if (sigsuspend(&sigset) == -1) // blocks all except for SIGUSR1
        perror("sigsuspend() returned -1 as expected");
    printf("sigsuspend is over at %s", ctime(time(&t)));
}
```

# Example continued

Output

```
parent is waiting for child to send SIGUSR1 at Fri Jun 16 12:30:57 2001
child is sending SIGUSR2 signal - which should be blocked
child is sending SIGUSR1 signal - which should be caught
catcher caught SIGUSR2
catcher caught SIGUSR1
sigsuspend() returned -1 as expected: Interrupted function call
sigsuspend is over at Fri Jun 16 12:31:12 2001
```

*↳ Depends which one first*

Why are there sleep calls in the child process? What could happen if we remove them?

# BIL 344 System Programming

Week 5

---

*Processes continued...*

- *Waiting for children processes*
- *Orphans, zombies,*
- *Daemons*

# is it stopped properly Monitoring Child Processes

In many application designs, a parent process needs to know when one of its child processes changes state - when the child terminates or is stopped by a signal.

This lecture will focus on two techniques used to monitor child processes: the *wait()* system call (and its variants) and the SIGCHLD signal.

The wait system call suspends execution of the calling thread until one of its children terminates.

The wait system call suspends execution of the calling thread until child specified by pid argument has changed state.

# The wait () system call

```
#include <sys/wait.h>
```

```
pid_t wait(int *status);
```

↳ The process that called goes to the blocked queue.

Returns process ID of terminated child, or -1 on error

- The **wait()** system call waits for one of the children of the calling process to terminate and returns the termination status of that child in the buffer pointed by **status**
- If no (previously unwaited-for) child of the calling process has yet terminated, the call blocks until one of the children terminates. If a child has already terminated by the time of the call, **wait()** returns immediately.
- If **status** is not NULL, information about how the child terminated is returned in the integer to which it points.

```

#include <sys/wait.h>
#include <time.h>
#include "curr_time.h"           /* Declaration of currTime() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int numDead;      /* Number of children so far waited for */
    pid_t childPid;  /* PID of waited for child */
    int j;

    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sleep-time...\n", argv[0]);

    setbuf(stdout, NULL);          /* Disable buffering of stdout */
    Print without buffering directly
    for (j = 1; j < argc; j++) {   /* Create one child for each argument */
        switch (fork()) {
            case -1:
                errExit("fork");

            case 0:                  /* Child sleeps for a while then exits */
                printf("[%s] child %d started with PID %ld, sleeping %s "
                       "seconds\n", currTime("%T"), j, (long) getpid(), argv[j]);
                sleep(getInt(argv[j], GN_NONNEG, "sleep-time"));
                exit(EXIT_SUCCESS);

            default:                 /* Parent just continues around loop */
                break;
        }
    }

    numDead = 0;
    for (;;) {                   /* Parent waits for each child to exit */
        childPid = wait(NULL);
        if (childPid == -1) {
            if (errno == ECHILD) {
                printf("No more children - bye!\n");
                exit(EXIT_SUCCESS);
            } else {                /* Some other (unexpected) error */
                errExit("wait");
            }
        }

        numDead++;
        printf("[%s] wait() returned child PID %ld (numDead=%d)\n",
               currTime("%T"), (long) childPid, numDead);
    }
}

```

**\_exit() does not call onexit  
methods conversely to  
exit()** *before*

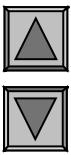


*There is no guarantee which one execute first, / finish*

```

$ ./multi_wait 7 1 4
[13:41:00] child 1 started with PID 21835, sleeping 7 seconds
[13:41:00] child 2 started with PID 21836, sleeping 1 seconds
[13:41:00] child 3 started with PID 21837, sleeping 4 seconds
[13:41:01] wait() returned child PID 21836 (numDead=1)
[13:41:04] wait() returned child PID 21837 (numDead=2)
[13:41:07] wait() returned child PID 21835 (numDead=3)
No more children - bye!

```



# The *waitpid()* System Call

```
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);

>Returns process ID of child, 0, or -1 on error
```

- If a parent process has created multiple children, it is not possible to *wait()* for the completion of a specific child; we can only wait for the next child that terminates.
- If no child has yet terminated, *wait()* always blocks. Sometimes, it would be preferable to perform a nonblocking wait so that if no child has yet terminated, we obtain an immediate indication of this fact.
- Using *wait()*, we can find out only about children that have terminated. It is not possible to be notified when a child is stopped by a signal (such as SIGSTOP or SIGTTIN) or when a stopped child is resumed by delivery of a SIGCONT signal.

## ***waitpid ()***

---

The return value and status arguments of *waitpid()* are the same as for *wait()*.

The pid argument enables the selection of the child to be waited for, as follows :

- If pid is greater than 0, wait for the child whose process ID equals pid
- If pid equals 0, wait for any child in the same process group as the caller
- If pid is less than –1, wait for any child whose process group identifier equals the absolute value of pid.
- If pid equals –1, wait for any child. The call `wait(&status)` is equivalent to the call `waitpid(-1, &status, 0)`.



```
#include <sys/wait.h>
#include "print_wait_status.h" /* Declares printWaitStatus()*/
#include "tlpi_hdr.h"

int main(int argc, char *argv[])
{
    int status;
    pid_t childPid;
    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [exit-status]\n", argv[0]);

    switch (fork()) {
        case -1: errExit("fork");

        case 0: /* Child: either exits with given status or loops waiting for signals */
            printf("Child started with PID = %ld\n", (long) getpid());
            if (argc > 1) /* Status supplied on command line? */
                exit(getInt(argv[1], 0, "exit-status"));
            else /* Otherwise, wait for signals */
                for (;;)
                    pause();
            exit(EXIT_FAILURE); /* Not reached, but good practice */
        default: /* Parent: wait on child till it either exits or is terminated by a signal */
        for (;;) {
            childPid = waitpid(-1, &status, WUNTRACED | WCONTINUED);
            if (childPid == -1)
                errExit("waitpid");
            printf("waitpid() returned: PID=%ld; status=0x%04x (%d,%d)\n",
                   (long) childPid, (unsigned int) status, status >> 8, status & 0xff);
            printWaitStatus(NULL, status);
            if (WIFEXITED(status) || WIFSIGNALED(status))
                exit(EXIT_SUCCESS);
        }
    }
}
```

Macros in <sys/wait.h>

## What is going on ?

---

- The *printWaitStatus()* function is used in Listing 26-3. This program creates a child process that either loops continuously calling *pause()* or, if an integer command-line argument was supplied, exits immediately using this integer as the exit status.
- In the meantime, the parent monitors the child via *waitpid()*, printing the returned status value and passing this value to *printWaitStatus()*.
- The parent exits when it detects that the child has either exited normally or been terminated by a signal.

# Process Termination from a Signal Handler

Inside handler, perform min num of instructions.

- In some circumstances, we may wish to have certain cleanup steps performed before a process terminates. For this purpose, we can arrange to have a handler catch such signals, perform the cleanup steps, and then terminate the process.
- If we do this, we should bear in mind that the termination status of a process is available to its parent via `wait()` or `waitpid()`.
- For example, calling `_exit(EXIT_SUCCESS)` from the signal handler will make it appear to the parent process that the child terminated successfully

# Process Termination from a Signal Handler

- If the child needs to inform the parent that it terminated because of a signal, then the child's signal handler should first disestablish itself, and then raise the same signal once more. The signal handler would contain code such as the following:

```
void handler(int sig)
{
    /* Perform cleanup steps */

    signal(sig, SIG_DFL);
    raise(sig);

    /* Disestablish handler */
    /* Raise signal again */
}
```

*Catch the signal in child process.*

*Keep the minimum for setting flag variable; use handler.*

*Kodda dan  
sikarmak*

*Most sophisticated version of wait family*

## The `waitid()` System Call

Like `waitpid()`, `waitid()` returns the status of child processes. However, `waitid()` provides extra functionality that is unavailable with `waitpid()`

```
#include <sys/wait.h>

int waitid(idtype_t idtype , id_t id , siginfo_t * infop , int options );
```

Returns 0 on success or if WNOHANG was specified and there were no children to wait for, or -1 on error

They rarely dominate same time.

## Orphans and Zombies

The lifetimes of parent and child processes are usually not the same—either the parent outlives the child or vice versa. This raises two questions:

- Who becomes the parent of an orphaned child? The orphaned child is adopted by init, the ancestor of all processes, whose process ID is 1. In other words, after a child's parent terminates, a call to `getppid()` will return the value 1 (This can be used as a way of determining if a child's true parent is still alive)  
*When parent terminates, it is adopted by init(ancestor)*
- What happens to a child that terminates before its parent has had a chance to perform a `wait()`? The point here is that, although the child has finished its work, the parent should still be permitted to perform a `wait()` at some later time to determine how the child terminated. The kernel deals with this situation by turning the child into a zombie.

# Zombies

- Regarding zombies, UNIX systems imitate the movies—a zombie process can't be killed by a signal, not even the (silver bullet) SIGKILL. This ensures that the parent can always eventually perform a *wait()*  
*they still occupy rows on table. We should always cleanup after child. (MW Rule)*
- When the parent does perform a *wait()*, the kernel removes the zombie, since the last remaining information about the child is no longer required. On the other hand, if the parent terminates without doing a *wait()*, then the *init* process adopts the child and automatically performs a *wait()*, thus removing the zombie process from the system.
- If a parent creates a child, but fails to perform a *wait()*, then an entry for the zombie child will be maintained indefinitely in the kernel's process table. If a large number of such zombie children are created, they will eventually fill the kernel process table, preventing the creation of new processes.

# Zombies

---

Cost of zombie processes:

- All the memory and resources allocated to a process are deallocated when the process terminates using the `exit()` system call. **But the process's entry in the process table is still available.**
- They occupy a PID, of which a finite amount is available.  
e.g. linux: 32.768, FreeBSD: 99.999, Solaris: 999.999, etc.
- PIDs wrapup when maxed.

# The SIGCHLD Signal

The termination of a child process is an event that occurs asynchronously. A **parent can't predict when one of its children will terminate**. We have already seen that the parent should use *wait()* (or similar) in order to prevent the accumulation of zombie children, and have looked at **two ways** in which this can be done:

- The parent can **call *wait()*, or *waitpid()*** without specifying the WNOHANG flag, in which case the call will block if a child has not already terminated.
- The parent can periodically perform **a nonblocking check (a poll)** for dead children via a call to *waitpid()* specifying the WNOHANG flag

**Both of these approaches are inconvenient .**

On the one hand, we may not want the parent **to be blocked** waiting for a child to terminate. On the other hand, **making repeated nonblocking *waitpid()* calls wastes CPU time** and adds complexity to an application design.

AS soon as you see this sign,  
that means a child process (at least 1) is terminated.

# Establishing a Handler for SIGCHLD

- The **SIGCHLD** signal is sent to a **parent process** whenever one of its children terminates. By default, this signal is ignored
- A common way of reaping <sup>toplanaat, biomek</sup> dead children processes is to establish a **handler for the SIGCHLD** signal. This signal is delivered to a parent process **whenever one of its children terminates**, and optionally when a child is stopped by a signal.
- ~~Alternatively, but somewhat less portably, a process may elect to set the disposition of SIGCHLD to SIG\_IGN, in which case the status of terminated children is immediately discarded (and thus can't later be retrieved by the parent), and the children don't become zombies.~~

# Establishing a Handler for SIGCHLD

```
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/wait.h>

sig_atomic_t child_exit_status;

void clean_up_child_process (int signal_number)
{
    /* Clean up the child process.  */
    int status;
    wait (&status);
    /* Store its exit status in a global variable.  */
    child_exit_status = status;
}

int main ()
{
    /* Handle SIGCHLD by calling clean_up_child_process.  */
    struct sigaction sigchld_action;
    memset (&sigchld_action, 0, sizeof (sigchld_action));
    sigchld_action.sa_handler = &clean_up_child_process;
    sigaction (SIGCHLD, &sigchld_action, NULL);

    /* Now do things, including forking a child process.  */
    /* ... */

    return 0;
}
```

# Executing a New Program: `execve()`

→ Replace current process content  
with the target process's!

- The `execve()` system call loads a new program into a process's memory. During this operation, the old program is discarded, and the process's stack, data, and heap are replaced by those of the new program.
- The most frequent use of `execve()` is in the child produced by a `fork()`, although it is also occasionally used in applications without a preceding `fork()`.
- Various library functions, all with names beginning with `exec`, are layered on top of the `execve()` system call. Each of these functions provides a different interface to the same functionality. The loading of a new program by any of these calls is commonly referred to as an **exec operation**, or simply by the notation `exec()`.

# execve()

```
#include <uninst.h>

int execve(const char * pathname , char *const argv [], char *const envp []);
```

Never returns on success; returns -1 on error

The *pathname* argument contains the pathname of the new program to be loaded into the process's memory.

The *argv* argument specifies the command-line arguments to be passed to the new program. The value supplied for *argv[0]* corresponds to the command name.

The final argument, *envp*, specifies the environment list for the new program. The *envp* argument corresponds to the *environ array* of the new program; it is a NULL - terminated list of pointers to character strings of the form *name=value*

# execve()

Since it replaces the program that called it, a **successful execve()** **never returns**. We never need to check the **return value** from execve(); it will **always be -1**. The very fact that it returned informs us that an error occurred. Among the errors that may be returned in errno are the following:

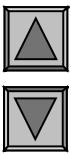
**EACCES** : The pathname argument doesn't refer to a regular file, the file doesn't have execute permission enabled, or one of the directory components of pathname is not searchable

**ENOENT** : The file referred to by pathname doesn't exist.

**ENOEXEC** : The file referred to by pathname is marked as being executable, but it is not in a recognizable executable format. Possibly, it is a script that doesn't begin with a line specifying a script interpreter.

**ETXTBSY** : The file referred to by pathname is open for writing by another process

**E2BIG** : The total space required by the argument list and environment list exceeds the allowed maximum



```
#include "tlpi_hdr.h"

int main(int argc, char *argv[])
{
    char *argVec[10];                                /* Larger than required */
    char *envVec[] = { "GREET=salut", "BYE=adieu", NULL };

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s pathname\n", argv[0]);

    argVec[0] = strrchr(argv[1], '/');           /* Get basename from argv[1] */
    if (argVec[0] != NULL)                            → returns string after '/'.
        argVec[0]++;
    else
        argVec[0] = argv[1];
    argVec[1] = "hello world";
    argVec[2] = "goodbye";
    argVec[3] = NULL;                                /* List must be NULL-terminated */

    execve(argv[1], argVec, envVec); → Loads a new program into the process' memory.
    errExit("execve");                            /* If we get here, something went wrong */

}
```

Imagine the input from the commandline is '/bin/ls'



```
#include "tlpi_hdr.h"
extern char **environ;

int main(int argc, char *argv[])
{
    int j;
    char **ep;

    for (j = 0; j < argc; j++)
        printf("argv[%d] = %s\n", j, argv[j]);

    for (ep = environ; *ep != NULL; ep++)
        printf("environ: %s\n", *ep);

exit(EXIT_SUCCESS);

}
```

\$ ./t\_execve ./envargs

argv[0] = envargs  
argv[1] = hello world  
argv[2] = goodbye  
environ: GREET=salut  
environ: BYE=adieu

↗ *environmental variables*

# The exec() library functions

```
#include <uninst.h>

int execle(const char * pathname , const char * arg , ...
           /* , (char *) NULL, char *const envp [] */ );
int execlp(const char * filename , const char * arg , ...
           /* , (char *) NULL */);

int execvp(const char * filename , char *const argv []);
int execv(const char * pathname , char *const argv []);
int execl(const char * pathname , const char * arg , ...
           /* , (char *) NULL */);
```

Change  
the form of  
Parameters

None of the above returns on success; all return -1 on error

## Summary of differences between the exec() functions

**pathname:** go to file at given path

**name+PATH:** search for the file in the directories specified by PATH

Function	Specification of program file (-, p)	Specification of arguments (v, l)	Source of environment (e, -)
<i>execve()</i>	pathname	array	<i>envp</i> argument
<i>execle()</i>	pathname	list	<i>envp</i> argument
<i>execlp()</i>	filename + PATH	list	caller's <i>environ</i>
<i>execvp()</i>	filename + PATH	array	caller's <i>environ</i>
<i>execv()</i>	pathname	array	caller's <i>environ</i>
<i>execl()</i>	pathname	list	caller's <i>environ</i>

## Signals and `exec()`

---

- During an `exec()`, the text of the existing process is discarded. This text may include **signal handlers** established by the calling program. Because **the handlers disappear**, the kernel resets the dispositions of all handled signals to `SIG_DFL` .
- The dispositions of **all other signals are left unchanged** by an `exec()`.
- Signals should not be blocked or ignored across an `exec()` of an arbitrary program (here, “arbitrary” means a program that we did not write) It is acceptable to block or ignore signals when execing a program we have written or one with known behavior with respect to signals (OPTIONAL).

# Example of fork+exec

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
Yunus Jamat
/* Spawns a child process running a new program. PROGRAM is the name
   of the program to run; the path will be searched for this program.
   ARG_LIST is a NULL-terminated list of character strings to be
   passed as the program's argument list. Returns the process ID of
   the spawned process. */
int spawn (char* program, char** arg_list)
{
    pid_t child_pid;
    /* Duplicate this process. */
    child_pid = fork ();
    if (child_pid != 0)
        /* This is the parent process. */
        return child_pid;
    else {
        /* Now execute PROGRAM, searching for it in the path. */
        execvp (program, arg_list);
        /* The execvp function returns only if an error occurs. */
        fprintf (stderr, "an error occurred in execvp\n");
        abort ();
    }
}

int main ()
{
    /* The argument list to pass to the "ls" command. */
    char* arg_list[] = {
        "ls",      /* argv[0], the name of the program. */
        "-l",
        "/",
        NULL      /* The argument list must end with a NULL. */
    };
    /* Spawns a child process running the "ls" command. Ignore the
       returned child process ID. */
    spawn ("ls", arg_list);
    printf ("done with main program\n");
    return 0;
}
```

★ The child process' pid is never 0.  
fork returns 0 to the child to tell it that it's the child.

★ The child process' pid is the value that fork returns to the parent.

★ Fork (when succeeds) returns twice,  
once in the child, once in the parent.

Function	Specification of program file (-, p)	Specification of arguments (v, l)	Source of environment (e, -)
execve()	pathname	array	envp argument
execle()	pathname	list	envp argument
execvp()	filename + PATH	list	caller's environ
execvp()	filename + PATH	array	caller's environ
execv()	pathname	array	caller's environ
execl()	pathname	list	caller's environ

→ Contains ls command to execute.

# Executing a Shell Command: *system()*

ps aux?  
↓ List the currently running processes and PIDs.

- The *system()* function allows the calling program to execute an arbitrary shell command.
- Let's see how can one implement *system()* using *fork()*, *exec()*, *wait()*, and *exit()*.

```
#include <stdlib.h>  
  
int system(const char *command);
```

→ don't use it on homeworks.

The *system()* function creates a child process that invokes a shell to execute a command.

# system()

---

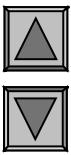
The principal **advantages** of *system()* are **simplicity** and **convenience**:

- We don't need to **handle the details** of calling *fork()*, *exec()*, *wait()*, and *exit()*.
- **Error and signal handling** are performed by *system()* on our behalf.
- Because *system()* uses the shell to execute command, all of the usual **shell processing, substitutions, and redirections are performed on command** before it is executed. This makes it easy to add an “execute a shell command” feature to an application.

First it creates shell and then executes (inefficiency)  
**system()**

- The **main cost** of *system()* is **inefficiency**.
- Executing a command using *system()* requires the **creation of at least two processes**—one for the shell and one or more for the command(s) it executes—each of which performs an *exec()*.
- If **efficiency or speed** is a requirement, it is preferable to use explicit *fork()* and *exec()* calls to execute the desired program.

If shell has security problem, because of *system()* also  
our system is not safe



```
#include <sys/wait.h>
#include "print_wait_status.h"
#include "tlpi_hdr.h"
#define MAX_CMD_LEN 200

int main(int argc, char *argv[])
{
    char str[MAX_CMD_LEN];           /* Command to be executed by system() */
    int status;                      /* Status return from system() */

    for (;;) {                      /* Read and execute a shell command */
        printf("Command: ");
        fflush(stdout);
        if (fgets(str, MAX_CMD_LEN, stdin) == NULL)
            break;                   /* break at end-of-file */

        status = system(str);
        printf("system() returned: status=0x%04x (%d,%d)\n",
               (unsigned int) status, status >> 8, status & 0xff);

        if (status == -1) {
            errExit("system");
        }
        else {
            if (WIFEXITED(status) && WEXITSTATUS(status) == 127)
                printf("(Probably) could not invoke shell\n");
            else                      /* Shell successfully executed command */
                printWaitStatus(NULL, status);
        }
    }
    exit(EXIT_SUCCESS);
}
```

# Sample run:

```
$ ./t_system
Command: whoami
mtk
system() returned: status=0x0000 (0,0)
child exited, status=0
Command: ls | grep XYZ
system() returned: status=0x0100 (1,0)
child exited, status=1
Command: exit 127
system() returned: status=0x7f00 (127,0)
(Probably) could not invoke shell
Command: sleep 100
Type Control-Z to suspend foreground process group
[1]+  Stopped                  ./t_system
$ ps | grep sleep
29361 pts/6    00:00:00 sleep
$ kill 29361
$ fg
./t_system
system() returned: status=0x000f (0,15)
child killed by signal 15 (Terminated)
Command: ^D$
```

*Shell terminates with the status of...  
its last command (grep), which...  
found no match, and so did an exit(1)*

*Actually, not true in this case*

*Find PID of sleep*

*And send a signal to terminate it  
Bring t\_system back into foreground*

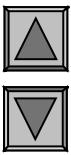
*Type Control-D to terminate program*

# Implementing *system()*

To implement *system()*, we need to use *fork()* to create a child that then does an *execl()* to implement a command like

*Execute shell*  
*I*  
`execl("/bin/sh", "sh", "-c", command, (char *) NULL);`  
*Shell will execute target "command"*

To collect the status of the child created by *system()*, we use a *waitpid()* call that specifies the child's process ID.



```
#include <unistd.h>
#include <sys/wait.h>
#include <sys/types.h>

int system(char *command)
{
    int status;
    pid_t childPid;

    switch (childPid = fork()) {
        case -1: /* process creation error */
            return -1;

        case 0: /* Child */
            execl("/bin/sh", "sh", "-c", command, (char *) NULL);
            _exit(127);
            /* Failed exec; not supposed to reach this line */

        default: /* Parent */
            if (waitpid(childPid, &status, 0) == -1)
                return -1; → if wait fails.
            else
                return status;
    }
}
```

Using `wait()` would not suffice, as `wait()` waits for any child, could accidentally collect the status of some other child created by the calling process.

# Treating signals correctly inside `system()`

- What **adds complexity** to the implementation of `system()` is the correct **treatment with signals**.
- The first signal to consider is SIGCHLD . Suppose that the **program calling `system()`** is also directly creating children, and has established **a handler for SIGCHLD** that performs its own `wait()`. In this situation, when a SIGCHLD signal is generated by the termination of the child created by `system()`, it is possible that **the signal handler of the main program will be invoked**—and collect the child’s status—before `system()` **has a chance to call `waitpid()`**.
- Therefore, `system()` must block delivery of SIGCHLD while it is executing

# Treating signals correctly inside system()

- The other signals to consider are those generated by the terminal interrupt (usually Control-C) and quit (usually Control-\) characters, SIGINT and SIGQUIT , respectively
- SIGINT and SIGQUIT should be ignored in the calling process while the command is being executed.

If you want to implement system(), use ns.

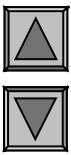
## Beyond the parent-child relation

**Why a session?** When a user logs out of a system, the kernel needs to terminate all the processes the user had running, the session id helps in this regard. The session's ID is the same as the pid of the process that created the session through the setsid() system call. That process is known as the session leader for that session group. All of that process's descendants are then members of that session unless they specifically remove themselves from it.

**Controlling terminal** Every session is tied to a terminal from which processes in the session get their input and to which they send their output. The terminal to which a session is related is called the controlling terminal (or controlling tty) of the session. A terminal can be the controlling terminal for only one session at a time.

**Why a group?** e.g. ls | grep "[aA].\*gz" | more

To enable job control; stop/suspend/continue collections of "related" processes.



```
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>
#include <sys/types.h>
#include <errno.h>

int system(const char *command)
{
    sigset_t blockMask, origMask;
    struct sigaction saIgnore, saOrigQuit, saOrigInt, saDefault;
    pid_t childPid;
    int status, savedErrno;

    if (command == NULL) /* Is a shell available? */
        return system(":") == 0;

    sigemptyset(&blockMask); /* Block SIGCHLD */
    sigaddset(&blockMask, SIGCHLD);
    sigprocmask(SIG_BLOCK, &blockMask, &origMask);

    saIgnore.sa_handler = SIG_IGN; /* Ignore SIGINT and SIGQUIT */
    saIgnore.sa_flags = 0;
    sigemptyset(&saIgnore.sa_mask);
    sigaction(SIGINT, &saIgnore, &saOrigInt);
    sigaction(SIGQUIT, &saIgnore, &saOrigQuit);
```

```

switch (childPid = fork()) {
case -1:                                /* fork() failed */
    status = -1;
    Break;                                /* Carry on to reset signal attributes */

case 0: /* Child: exec command */
    saDefault.sa_handler = SIG_DFL;
    saDefault.sa_flags = 0;
    sigemptyset(&saDefault.sa_mask);

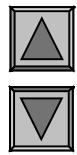
    if (saOrigInt.sa_handler != SIG_IGN) sigaction(SIGINT, &saDefault, NULL);
    if (saOrigQuit.sa_handler != SIG_IGN) sigaction(SIGQUIT, &saDefault, NULL);

    sigprocmask(SIG_SETMASK, &origMask, NULL);

    execl("/bin/sh", "sh", "-c", command, (char *) NULL);
    _exit(127);                            /* We could not exec the shell */

default: /* Parent: wait for our child to terminate */
    while (waitpid(childPid, &status, 0) == -1) {
        if (errno != EINTR) {                /* Error other than EINTR */
            status = -1;
            break;                            /* So exit loop */
        }
    }
    break;
}
/* Unblock SIGCHLD, restore dispositions of SIGINT and SIGQUIT */
savedErrno = errno;                      /* The following may change 'errno' */
sigprocmask(SIG_SETMASK, &origMask, NULL);
{ sigaction(SIGINT, &saOrigInt, NULL);
  sigaction(SIGQUIT, &saOrigQuit, NULL);
errno = savedErrno;
return status;
}

```



# Daemons

A daemon is a process with the following characteristics:

- **It is long-lived.** Often, a daemon is created at system startup and runs until the system is shut down.
- **It runs in the background and has no controlling terminal.** The lack of a controlling terminal ensures that the kernel never automatically generates any job-control or terminal-related signals (such as SIGINT, SIGTSTP, and SIGHUP) for a daemon.
- It is a convention (not universally observed) that **daemons have names ending with the letter *d*.**

Database servers are usually daemons  
server processes  
web servers

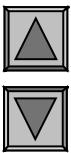
# Creating a Daemon

To become a daemon, a program performs the following steps:

- Perform a `fork()`, after which the **parent exits and the child continues** (the shell returns to its prompt) *inherit controlling terminal*.
- The child process calls `setsid()` to start a new session and free itself of any association with a controlling terminal (otherwise closing the terminal will kill the daemon)
- If the daemon might later open a terminal device, then we must take steps to ensure that the device does not become the controlling terminal
- **Clear the process umask to ensure that, when the daemon creates files and directories, they have the requested permissions.**
- **Change the process's current working directory, typically to the root directory**
- **Close all open file descriptors** that the daemon has inherited from its parent
- After having closed file descriptors 0, 1, and 2, a daemon normally **opens `/dev/null`** and uses `dup2()` (or similar) to make all those descriptors refer to this device; this way neither the daemon nor its children write to the terminal when running + detach from tty
  - duplicate so they all go to null.*

session ID → process group ID → process ID

*→ Make sure that nobody has a controlling signal.*



```
#ifndef BECOME_DAEMON_H                                /* Prevent double inclusion */
#define BECOME_DAEMON_H

/* Bit-mask values for 'flags' argument of becomeDaemon() */
#define BD_NO_CHDIR          01      /* Don't chdir("/") */
#define BD_NO_CLOSE_FILES     02      /* Don't close all open files */
#define BD_NO_REOPEN_STD_FDS  04      /* Don't reopen stdin, stdout, and stderr to
*                                   * /dev/null */
#define BD_NO_UMASK0          010    /* Don't do a umask(0) */
#define BD_MAX_CLOSE          8192   /* Maximum file descriptors to close if
*                                   * sysconf(_SC_OPEN_MAX) is indeterminate */

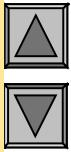
int becomeDaemon(int flags);

#endif
```

we have to close 0, 1 and 2

D

stdin, stdout and stderr are closed so that the daemon can detach successfully from the tty it was started from and also so that the daemon (or its child processes) won't write to the tty when it's running.



```
#include <sys/stat.h>
#include <fcntl.h>
#include "become_daemon.h"
#include "tlpi_hdr.h"

int becomeDaemon(int flags)                                /* Returns 0 on success, -1 on error */
{
    int maxfd, fd;

    switch (fork()) { To create clone of the parent process
        case -1: return -1;
        case 0: break;
        default: _exit(EXIT_SUCCESS);
    }

    if (setsid() == -1)                                     /* Become leader of new session, dissociate from tty */
        return -1;                                         /* can still acquire a controlling terminal */

    switch (fork()) { To guarantee, controlling terminal will not be acquired by accident.
        case -1: return -1;           /* thanks to 2nd fork, there is no way of acquiring a tty */
        case 0: break;
        default: _exit(EXIT_SUCCESS);
    }

    if (!(flags & BD_NO_UMASK0))                           /* Cancel inherited mask
                                                               and set our own mask. */
        umask(0);                                         /* Clear file mode creation mask */

    if (!(flags & BD_NO_CHDIR))                           /* Change to root directory */
        chdir("/"); Change to root directory
    if (!(flags & BD_NO_CLOSE_FILES)) {                   /* Close all open files */
        maxfd = sysconf(_SC_OPEN_MAX);
        if (maxfd == -1)                                 /* Limit is indeterminate... */
            maxfd = BD_MAX_CLOSE;                      /* so take a guess */

        for (fd = 0; fd < maxfd; fd++)
            close(fd);
    }
}
```

```

if (!(flags & BD_NO_REOPEN_STD_FDS)) {
    close(STDIN_FILENO); /* Reopen standard fd's to /dev/null */

    fd = open("/dev/null", O_RDWR);

    if (fd != STDIN_FILENO) /* 'fd' should be 0 */
        return -1;

    if (dup2(STDIN_FILENO, STDOUT_FILENO) != STDOUT_FILENO)
        return -1;

    if (dup2(STDIN_FILENO, STDERR_FILENO) != STDERR_FILENO)
        return -1;
}
return 0;
}

```

To point all of them  
to new device.

They will never have  
controlling terminal.

But if you want  
to handle them,  
daemons can handle  
them.

# Daemon

- Daemons **perform specific tasks**, such as providing a network login facility or serving web pages. To become a daemon, a program performs a standard sequence of steps, including calls to *fork()* and *setsid()*.
- Where appropriate, daemons should correctly handle the arrival of the SIGTERM and SIGHUP signals. The **SIGTERM signal should result in an orderly shutdown** of the daemon, while the **SIGHUP signal provides a way to trigger the daemon to reinitialize itself by rereading its configuration file and reopening any log files it may be using.**

Traditionally set

# BIL 344 System Programming

Week 6

## *Introduction to InterProcess Communication (IPC)*

- *Pipes*
- *FIFOs*

Inter  
Process  
Communication

# Interprocess Communication

---

IPC refers to the mechanisms an operating system provides to allow processes to manage shared data.

IPC facilities are divided into three broad categories:

- 1) **Communication**: exchanging data between processes
- 2) **Synchronization**: synchronizing the actions of processes
- 3) **Signals**: they have various uses; including communication & synchronization between processes.

# Interprocess Communication

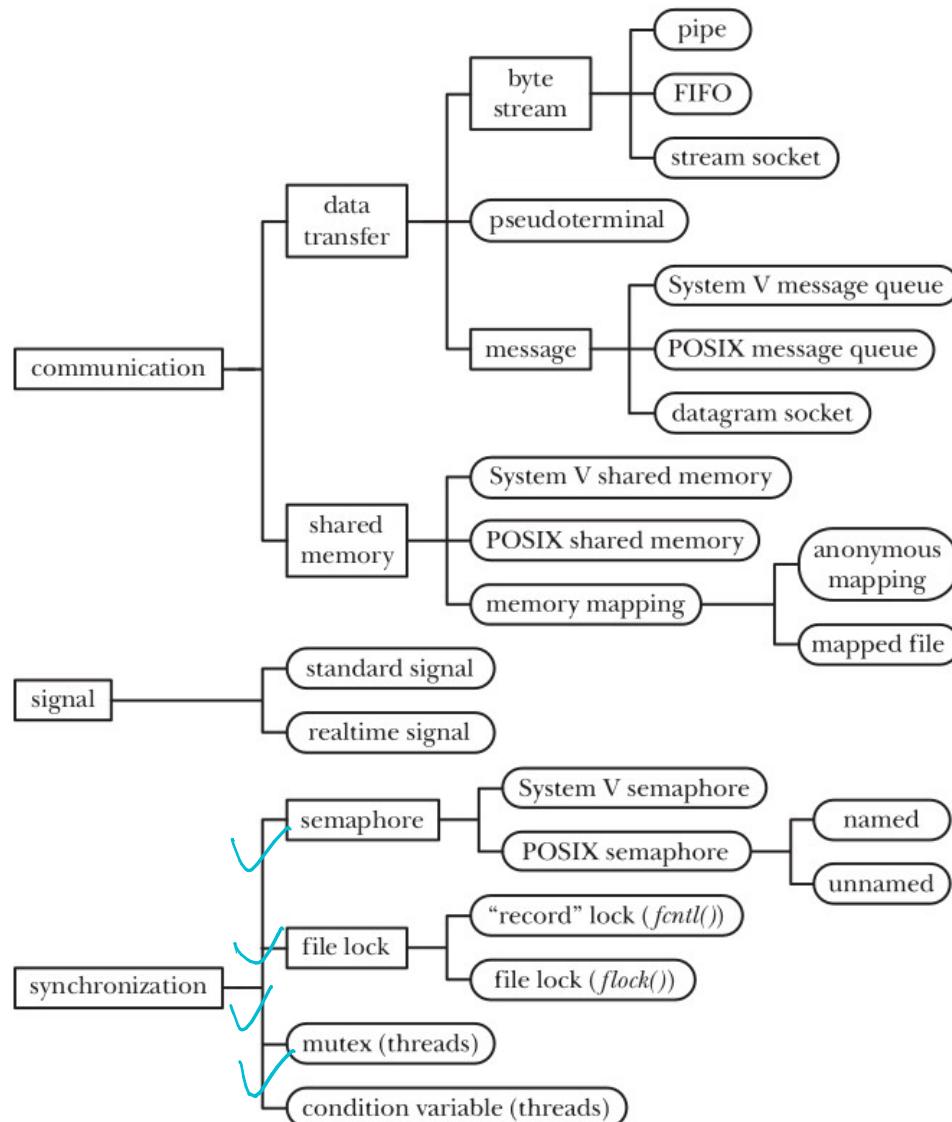


Figure 43-1: A taxonomy of UNIX IPC facilities

# Interprocess Communication

An overview of communication options:

- 1) **Shared memory** permits processes to communicate by simply reading and writing to a **shared memory page**.
- 2) **Mapped memory** is similar to shared memory, **except that it is associated with a file in the filesystem**.
- 3) **Pipes** permit **sequential** communication from one process to a related process.
- 4) **FIFOs** are similar to pipes, except that **unrelated processes can communicate** because the pipe is given a name in the filesystem.
- 5) **Sockets** support communication between unrelated processes even on different computers. *On distinct systems  
only option*  
and more (message queues, etc)..

# Interprocess Communication

IPC criteria:

- Whether they restrict communication to related processes (processes with a common ancestor), to unrelated processes sharing the same filesystem, or to any computer connected to a network
- Whether a communicating process is limited to only writing data or only reading data
- The number of processes permitted to communicate
- Whether the communicating processes are synchronized by the IPC; for example, a reading process halts until data is available to read

# Interprocess Communication

---

An overview of synchronization options:

- 1) **Semaphore**: a kernel maintained non-negative integer that can solve many of the commonly encountered synchronization issues.
- 2) **File locks**: are a synchronization method explicitly designed to coordinate the actions of multiple processes operating on the same file.
- 3)  **Mutexes & condition variables (i.e. monitors)**: are intended to be used between threads and can solve (in theory) any synchronization issue.

# Interprocess Communication

IPC continues to grow beyond POSIX and UNIX and has led to the development of modern technologies such as:

- Remote procedure calls (Java RMI, JSON-RPC, SOAP, .net remote)
- Distributed object models (CORBA, sessions)

and many more.

*→ a function located on one machine can call a procedure on another machine*

# Interprocess Communication

Pipes are the oldest Unix IPC method.

A pipe is a communication device that permits unidirectional communication. Data written to the “write end” of the pipe is read back from the “read end.”

→ one process can only send data with one pipe to other process  
doesn't go both ways

Pipes are serial devices; the data is always read from the pipe in the same order it was written.

Typically, a pipe is used to communicate between two threads in a single process or between parent and child processes.

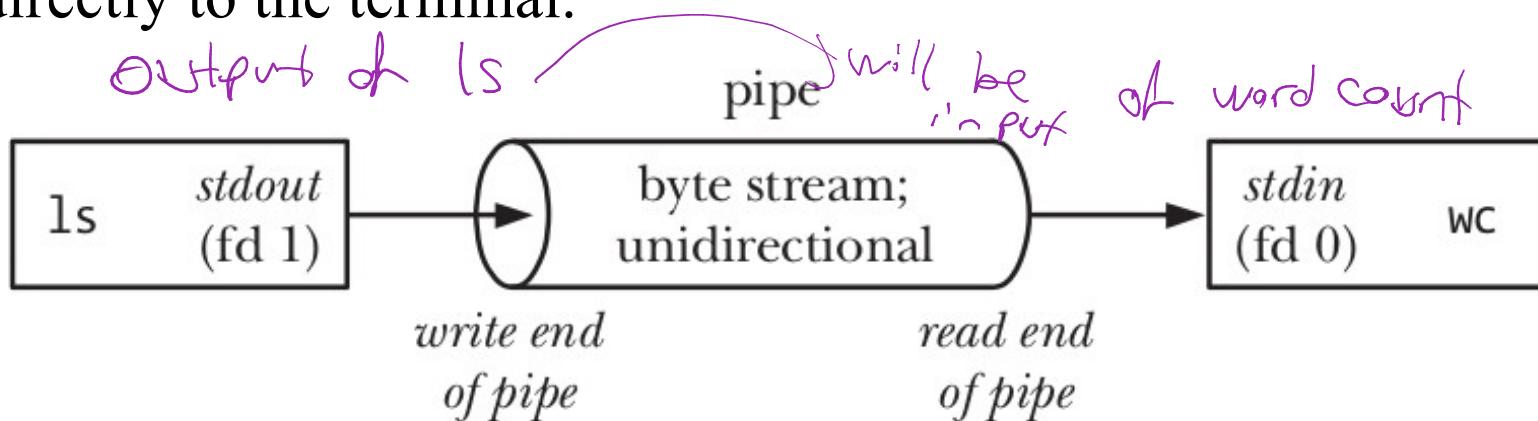
For <sup>usually</sup> <sub>parent</sub> processes

# Interprocess Communication

In a shell, the symbol | creates a pipe. For example, this shell command causes the shell to produce two child processes, one for ls and one for wc :

⇒ ls | wc -l  
↓  
pipe

The shell also creates a pipe connecting the standard output of the ls subprocess with the standard input of the wc process. The filenames listed by ls are sent to wc in exactly the same order as if they were sent directly to the terminal.



# Interprocess Communication

- The following command line inputs have the same effect

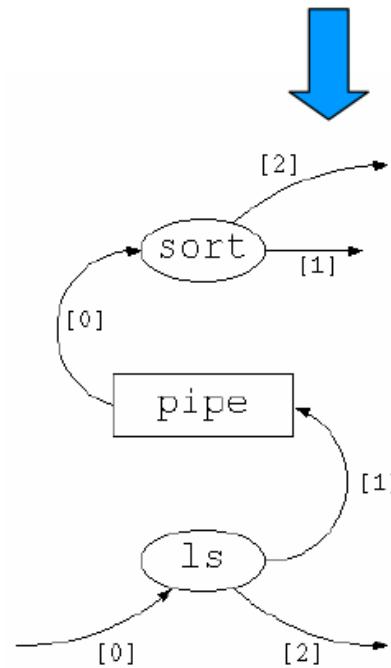
```
#> ls -l > my.file  
#> sort -n < my.file
```

Save  
thing

```
#> ls -l | sort -n
```



The output of `ls -l` is written to myfile  
sort command uses myfile as an input



sort  
file descriptor table  
[0] pipe *read*  
[1] standard *output*  
[2] standard *error*

ls  
file descriptor table  
[0] standard *input*  
[1] pipe *write*  
[2] standard *error*

# Interprocess Communication

A pipe is simply a buffer maintained in kernel memory. This buffer has a finite and limited capacity (often 64KB).

*If into the pipe, full, it will be blocked. Not able to write anything further*  
If the writer process writes faster than the reader process consumes the data, and if the pipe cannot store more data, the writer process **blocks** until more capacity becomes available.

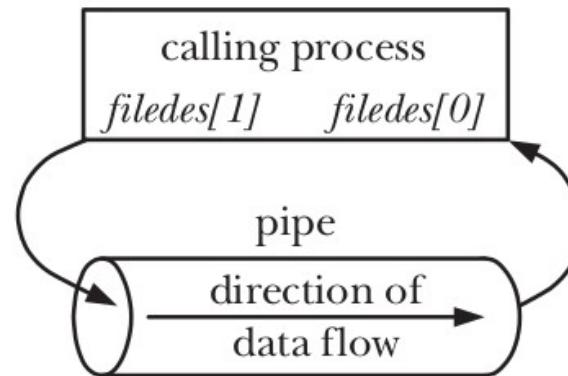
If the reader tries to read but no data is available, it **blocks** until data becomes available. Thus, the pipe automatically synchronizes the two processes.

*Automatically  
Synchronize*

# Interprocess Communication

The `pipe()` system call creates a new pipe.

```
#include <unistd.h>
int pipe(int filedes [2]);
```



**Figure 44-2:** Process file descriptors after creating a pipe

Returns 0 on success, or  $-1$  on error.

First supply an integer array of size 2. Then the call to `pipe()` stores the reading file descriptor in array position 0 and the writing file descriptor in position 1.

# Interprocess Communication

Example:

```
int pipe_fds[2];
int read_fd;
int write_fd;
✓pipe (pipe_fds);          // never forget error control
read_fd = pipe_fds[0]; ✓
write_fd = pipe_fds[1]; ✓
```

Data written to the file descriptor `write_fd` can be read back from `read_fd`.

# Interprocess Communication

A process's file descriptors cannot be passed to unrelated processes; however, when the process calls `fork`, file descriptors are copied to the new child process. Thus, pipes can connect only related processes.

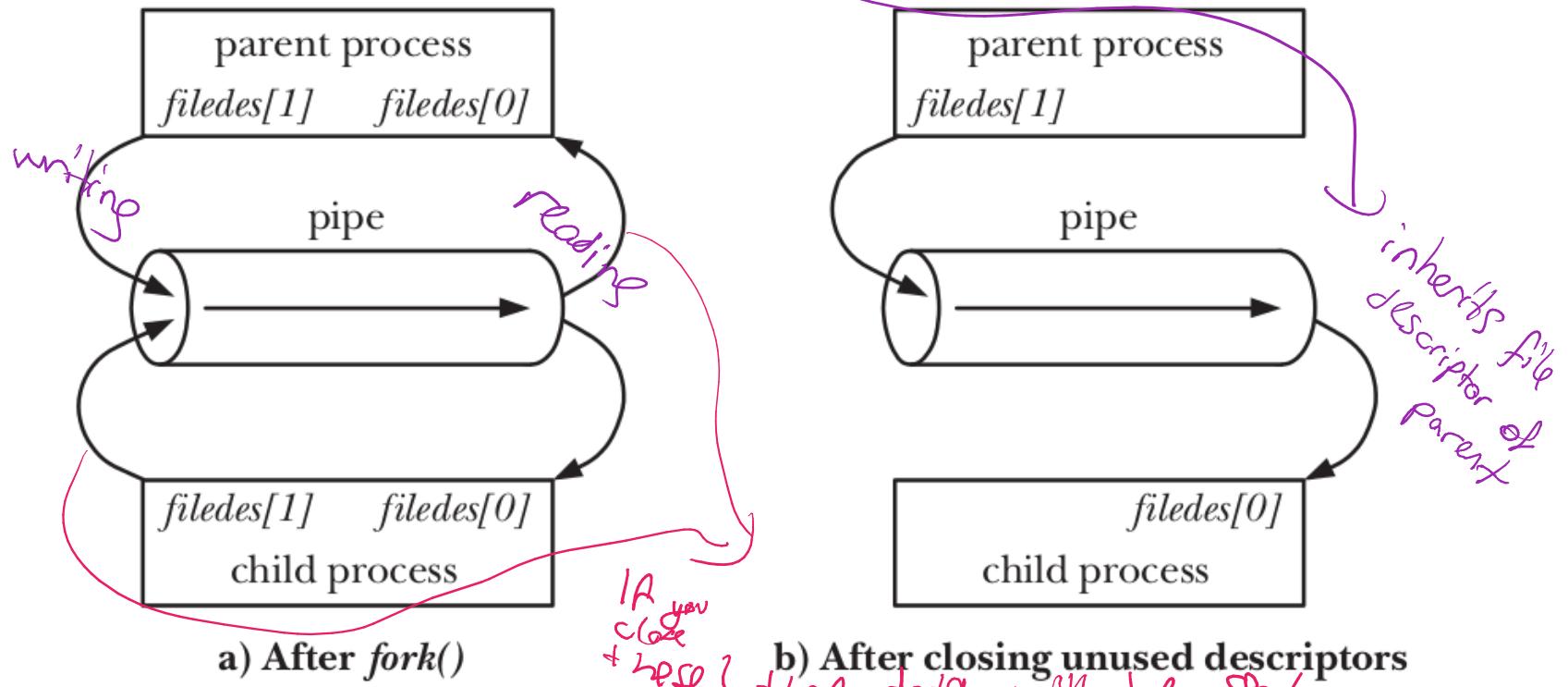


Figure 44-3: Setting up a pipe to transfer data from a parent to a child

# Interprocess Communication

**Listing 44-1:** Steps in creating a pipe to transfer data from a parent to a child

```
int filedes[2];

if (pipe(filedes) == -1)                      /* Create the pipe */
    errExit("pipe");

switch (fork()) {                                /* Create a child process */
case -1:
    errExit("fork");

case 0: /* Child */
    if (close(filedes[1]) == -1)                /* Close unused write end */
        errExit("close");

    /* Child now reads from pipe */
    break;

default: /* Parent */
    if (close(filedes[0]) == -1)                /* Close unused read end */
        errExit("close");

    /* Parent now writes to pipe */
    break;
}
```

# Interprocess Communication

Listing 5.7 (*pipe.c*) Using a Pipe to Communicate with a Child Process

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

/* Write COUNT copies of MESSAGE to STREAM, pausing for a second
   between each. */

void writer (const char* message, int count, FILE* stream)
{
    for (; count > 0; --count) {
        /* Write the message to the stream, and send it off immediately. */
        fprintf (stream, "%s\n", message);
        fflush (stream); → buffer, denizler
        /* Snooze a while. */
        sleep (1);
    }
}

/* Read random strings from the stream as long as possible. */

void reader (FILE* stream)
{
    char buffer[1024];
    /* Read until we hit the end of the stream. fgets reads until
       either a newline or the end-of-file. */
    while (!feof (stream) → until end of line character
          && !ferror (stream)
          && fgets (buffer, sizeof (buffer), stream) != NULL)
        fputs (buffer, stdout); prints buffer
}
```

Example (part 1/2)

# Interprocess Communication

## Example (part 2/2)

```
int main ()  
{  
    int fds[2];  
    pid_t pid;  
  
    /* Create a pipe.  File descriptors for the two ends of the pipe are  
       placed in fds.  */  
    pipe (fds);  
    /* Fork a child process.  */  
    pid = fork ();  
    if (pid == (pid_t) 0) {  
        FILE* stream;  
        /* This is the child process.  Close our copy of the write end of  
           the file descriptor.  */  
        close (fds[1]);  
        /* Convert the read file descriptor to a FILE object, and read  
           from it.  */  
        stream = fdopen (fds[0], "r"); → Creates file descriptor to FILE*  
        reader (stream);  
        close (fds[0]);  
    }  
    else {  
        /* This is the parent process.  */  
        FILE* stream;  
        /* Close our copy of the read end of the file descriptor.  */  
        close (fds[0]);  
        /* Convert the write file descriptor to a FILE object, and write  
           to it.  */  
        stream = fdopen (fds[1], "w");  
        writer ("Hello, world.", 5, stream); → writing this string into the stream  
        close (fds[1]);  
    }  
  
    return 0;  
}
```

# Interprocess Communication

# Symbol

When a pipe is created, the file descriptors used for the two ends of the pipe are the next lowest-numbered descriptors available. Since, in normal circumstances, descriptors 0, 1, and 2 are already in use for a process, some higher-numbered descriptors will be allocated for the pipe. *it's gonna use available lowest number for file descriptor*

So how do we bring about the situation where two programs that read from `stdin` and write to `stdout` are connected using a pipe, such that the standard output of one program is directed into the pipe and the standard input of the other is taken from the pipe?

To connect one file descriptor of one command to zero file descriptor of other command  
By duplicating By duplicating file descriptors!

# Interprocess Communication

```
int pfd[2];
pipe(pfd);
/* Allocates (say) file descriptors 3 and 4 for pipe */
/* Other steps here, e.g., fork() */
close(STDOUT_FILENO); /* Free file descriptor 1 */Now 1 is free
dup(pfd[1]); /* dup uses lowest free file descriptor, i.e., fd 1 */
```

*Stdout will go to the writing end.*

The end result of the above steps is that the process's standard output is bound to the write end of the pipe. A corresponding set of calls can be used to bind a process's standard input to the read end of the pipe.

# Interprocess Communication

However `dup` assumes that 0, 1 and 2 are already open.

If however 0 was closed for some reason, `pfld[1]` would become its clone instead of 1.

That's why it's a better idea to use `dup2()` instead of `close()` and `dup()`

```
/* Close fd 1, and reopen it as the write end of pipe */  
dup2(pfd[1], STDOUT_FILENO); // choose specifically!
```

Specifically saying a copy of stdout

# Interprocess Communication

After duplicating `pf[1]`, we now have two file descriptors referring to the write end of the pipe: descriptor 1 and `pf[1]`. Since unused pipe file descriptors should be closed, after the `dup2()` call, we close the superfluous descriptor:

```
if (pf[1] != STDOUT_FILENO) { // just in case writing end of pipe
    dup2(pf[1], STDOUT_FILENO);
    close(pf[1]);
}
```

We have 2 file descriptors pointing to the writing end of pipe.  
1. We need to close both with close().

# Interprocess Communication

Listing 44-4: Using a pipe to connect *ls* and *wc*

pipes/pipe\_ls\_wc.c

```
#include <sys/wait.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int pfd[2];                                /* Pipe file descriptors */

    if (pipe(pfd) == -1)                         /* Create pipe */
        errExit("pipe");

    switch (fork()) {
        case -1:
            errExit("fork");

        case 0:          /* First child: exec 'ls' to write to pipe */
            if (close(pfd[0]) == -1)                /* Read end is unused */
                errExit("close 1");

            /* Duplicate stdout on write end of pipe; close duplicated descriptor */

            if (pfd[1] != STDOUT_FILENO) {
                if (dup2(pfd[1], STDOUT_FILENO) == -1)
                    errExit("dup2 1");
                if (close(pfd[1]) == -1)
                    errExit("close 2");
            }
            execlp("ls", "ls", (char *) NULL);      /* Writes to pipe */
            errExit("execlp ls");

        default:         /* Parent falls through */
            break;
    }
}
```

Example (part 1/2)

Check if already in place  
If not explicitly copy it  
Now closed and offered as the writing end of the pipe  
Everything you write (unterminated etc) will go to the writing end of pipe

# Interprocess Communication

```
switch (fork()) {  
    case -1:  
        errExit("fork");  
  
    case 0:             /* Second child: exec 'wc' to read from pipe */  
        if (close(pfd[1]) == -1)           /* Write end is unused */  
            errExit("close 3");  
  
        /* Duplicate stdin on read end of pipe; close duplicated descriptor */  
  
        if (pfd[0] != STDIN_FILENO) {      /* Defensive check */  
            if (dup2(pfd[0], STDIN_FILENO) == -1)  
                errExit("dup2 2");  
            if (close(pfd[0]) == -1)  
                errExit("close 4");  
        }  
        execlp("wc", "wc", "-l", (char *) NULL); /* Reads from pipe */  
        errExit("execlp wc");  
  
    default:            /* Parent falls through */  
        break;  
}  
  
/* Parent closes unused file descriptors for pipe, and waits for children */  
  
if (close(pfd[0]) == -1)  
    errExit("close 5");  
if (close(pfd[1]) == -1)  
    errExit("close 6");  
if (wait(NULL) == -1)  
    errExit("wait 1");  
if (wait(NULL) == -1)  
    errExit("wait 2");  
  
exit(EXIT_SUCCESS);  
}
```

Example (part 2/2)

Using scarf, fgets doesn't matter it will always read it from the reading end of pipe.

If there is no thing to read, it will block until gets something.

# Interprocess Communication

The inherent synchronization mechanism of pipes can be exploited for general purpose synchronization as well.

Imagine a parent process that wants to wait/block until all of its children have accomplished their respective tasks (a.k.a. *synchronization barrier*); could you solve this through signals?

We can solve this with semaphores or mutexes

But How to solve with pipes?

# Interprocess Communication

It's solvable through pipes as well: the parent builds a pipe before creating the child processes.

Each child inherits a file descriptor for the write end of the pipe and closes this descriptor once it has completed its action.

After all of the children have closed their file descriptors for the write end of the pipe, the parent's `read()` from the pipe will complete, returning end-of-file (0). At this point, the parent is free to carry on to do other work.

*will receive EOF. So it will understand all reached Rendezvous point.*

(Note that closing the unused write end of the pipe in the parent is essential for the correct operation of this technique; otherwise, the parent would block forever when trying to read from the pipe.)

# Interprocess Communication

The following is an example of what we see when we use the program in Listing 44-3 to create three children that sleep for 4, 2, and 6 seconds:

```
$ ./pipe_sync 4 2 6
08:22:16 Parent started
08:22:18 Child 2 (PID=2445) closing pipe
08:22:20 Child 1 (PID=2444) closing pipe
08:22:22 Child 3 (PID=2446) closing pipe
08:22:22 Parent ready to go
```

Example (part 1/3)

**Listing 44-3:** Using a pipe to synchronize multiple processes

```
----- pipes/pipe_sync.c
#include "curr_time.h"          /* Declaration of currTime() */
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int pfd[2];                /* Process synchronization pipe */
    int j, dummy;
    if (argc < 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s sleep-time...\n", argv[0]);

    setbuf(stdout, NULL);      /* Make stdout unbuffered, since we
                                terminate child with _exit() */
    printf("%s Parent started\n", currTime("%T"));

    if (pipe(pfd) == -1)
        errExit("pipe");
```

*Establish the pipe in kernel memory*

# Interprocess Communication

```
for (j = 1; j < argc; j++) {  
    switch (fork()) {  
        case -1:  
            errExit("fork %d", j);  
  
        case 0: /* Child */  
            if (close(pfd[0]) == -1) /* Read end is unused */  
                errExit("close");  
  
            /* Child does some work, and lets parent know it's done */  
  
            sleep(getInt(argv[j], GN_NONNEG, "sleep-time")); /* Simulate processing */  
            printf("%s Child %d (PID=%ld) closing pipe\n",  
                   currTime("%T"), j, (long) getpid());  
            if (close(pfd[1]) == -1)  
                errExit("close");  
            /* Child now carries on to do other things... */  
  
            _exit(EXIT_SUCCESS);  
  
        default: /* Parent loops to create next child */  
            break;  
    }  
}
```

Example (part 2/3)

For every child  
/\* Read end is unused \*/

Close reading end  
Child has reached to rendezvous point,

# Interprocess Communication

```
/* Parent comes here; close write end of pipe so we can see EOF */  
  
if (close(pfd[1]) == -1) /* Write end is unused */  
    errExit("close");  
  
/* Parent may do other work, then synchronizes with children */  
  
if (read(pfd[0], &dummy, 1) != 0)  
    fatal("parent didn't get EOF");  
printf("%s Parent ready to go\n", currTime("%T"));  
  
/* Parent can now carry on to do other things... */  
  
exit(EXIT_SUCCESS);  
}
```

Example (part 3/3)

Parent closes its end writing pipe.

If there is at least one more child process that hasn't closed, parent will be blocked.

Only return has closed writing end of child pipe.

Only way of it's return is read to return with an end of file. (all file desc to be closed)

# Interprocess Communication

A common use for pipes is to execute a shell command and either read its output or send it some input. The `popen()` and `pclose()` functions are provided to simplify this task.

Not system call  
(C library)

```
#include <stdio.h>
```

```
FILE *popen(const char * command , const char * mode);
```

↳ Execute the command given and connect pipe to the end of this command.  
Returns file stream, or NULL on error

```
int pclose(FILE * stream);
```

Returns termination status of child process, or -1 on error

# Not recommended (Security Fisly) Involves creating instance executing a shell. And we don't know safe (like system())

## Interprocess Communication

The `popen()` function creates a pipe, and then forks a child process that execs a shell, which in turn creates a child process to execute the string given in command.

The mode argument is a string that determines whether the calling process will read from the pipe (mode is `r`) or write to it (mode is `w`).

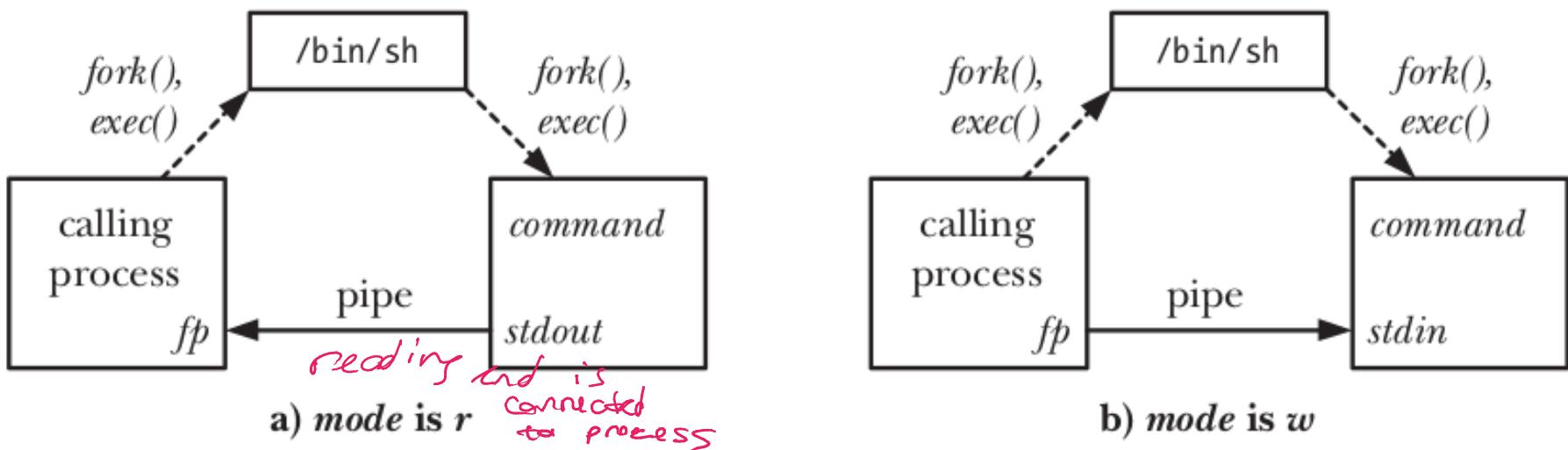


Figure 44-4: Overview of process relationships and pipe usage for `popen()`

# Interprocess Communication

Listing 5.9 (*popen.c*) Example Using *popen*

```
#include <stdio.h>
#include <unistd.h>

int main ()
{
    FILE* stream = popen ("sort", "w");
    fprintf (stream, "This is a test.\n");
    fprintf (stream, "Hello, world.\n");
    fprintf (stream, "My dog has fleas.\n");
    fprintf (stream, "This program is great.\n");
    fprintf (stream, "One fish, two fish.\n");
    return pclose (stream);
}
```

Eliminates the need for  
dup2,  
pipe,  
fork,  
exec,  
fdopen.

Security-wise, `popen()` is risky (like `system()`) as it involves executing a shell; you can find various exploit examples online for both, usually involving executing commands with a high-privilege user account.

# Interprocess Communication

Our second IPC tool is **FIFO**.

A FIFO is simply a pipe that has a name in the filesystem.

*Can be used by 2 processes executing on the same system.*

Any process can open or close the FIFO; **the processes on either end of the pipe need not be related to each other**. FIFOs are also called *named pipes*.

Just as with pipes, a FIFO has a write end and a read end, and data is read from the pipe in the same order as it is written. This fact gives FIFOs their name: *first-in, first-out*.

# Interprocess Communication

We can create a FIFO from the shell using the mkfifo command. Specify the path to the FIFO on the command line. For example, create a FIFO in /tmp/fifo by invoking this:

```
$ mkfifo /tmp/fifo
$ ls -l /tmp/fifo
p  prw-rw-rw- 1 erhan  users  0 Jan 18 14:04  /tmp/fifo
```

*Not an ordinary file, that's a FIFO.*

When applied to a FIFO (or pipe), fstat() and stat() return a file type of S\_IFIFO in the st\_mode field of the stat structure. When listed with ls -l, a FIFO is shown with the type **p** in the first column.

# Interprocess Communication

In one window, read from the FIFO by invoking the following:

```
$ cat < /tmp/fifo
```

In a second window, write to the FIFO by invoking this:

```
$ cat > /tmp/fifo
```

*What you write on 1 terminal,  
will appear on other terminal.*

Then type in some lines of text. Each time you press Enter, the line of text is sent through the FIFO and appears in the first window. Close the FIFO by pressing Ctrl+D in the second window.

Remove the FIFO with this line:

```
$ rm /tmp/fifo
```

# Interprocess Communication

The `mkfifo()` call creates a new FIFO with the given pathname.

```
#include <sys/stat.h>
```

```
int mkfifo(const char * pathname , mode_t mode );
```

Returns 0 on success, or -1 on error

The `mode` argument specifies the permissions for the new FIFO. These permissions are specified by ORing the desired combination of constants: S\_I(R|W|X)(USR|GRP|OTH)

# Interprocess Communication

Access a FIFO just like an ordinary file. To communicate through a FIFO, one process must open it for writing, and another process must open it for reading. Either low-level I/O functions (`open`, `write`, `read`, `close`) or C library I/O functions (`fopen`, `fprintf`, `fscanf`, `fclose`) may be used.

For example, to write a buffer of data to a FIFO using low-level I/O routines, you could use this code:

```
int fd = open (fifo_path, O_WRONLY) ;  
write (fd, data, data_length) ;  
close (fd) ;
```

# Interprocess Communication

To read a string from the FIFO using C library I/O functions, you could use this code:

```
FILE* fifo = fopen (fifo_path, "r");
fscanf (fifo, "%s", buffer);
fclose (fifo);
```

A FIFO can have multiple readers or multiple writers. Bytes from each writer are written atomically up to a maximum size of PIPE\_BUF (4KB on Linux). Chunks from simultaneous writers can be interleaved. Similar rules apply to simultaneous reads.

*At least 2 processes have open to fifo in order to continue.*

The FIFO must be opened on both ends before data can be passed. Normally, opening the FIFO blocks until the other end is opened also (use the O\_NONBLOCK flag in `open()` if you don't want this).

# Interprocess Communication

The parent reads what its child has written to a named pipe

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/wait.h>
#define BUFSIZE 256
#define FIFO_PERM (S_IRUSR | S_IWUSR)
/* function predefinitions */
int dofifochild(const char *fifoName, const char *idString);
int dofifoparent(const char *fifoName);

int main (int argc, char *argv[]) {
    pid_t childpid;

    if (argc != 2) {                                /* command line has pipe name */
        fprintf(stderr, "Usage: %s pipename\n", argv[0]);
        return 1;
    }
    if (mkfifo(argv[1], FIFO_PERM) == -1) {          /* create a named pipe */
        if (errno != EEXIST) {
            fprintf(stderr, "[%ld]:failed to create named pipe %s: %s\n",
                    (long)getpid(), argv[1], strerror(errno));
            return 1;
        }
    }
    if ((childpid = fork()) == -1){
        perror("Failed to fork");
        return 1;
    }
    if (childpid == 0)                                /* The child writes */
        return dofifochild(argv[1], "this was written by the child");
    else
        return dofifoparent(argv[1]);
}
```

Example (part 1/3)

# Interprocess Communication

The child writes to the pipe and returns

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include "restart.h"
#define BUFSIZE 256

int dofifochild(const char *fifoName, const char *idString) {
    char buf[BUFSIZE];
    int fd;
    int rval;
    ssize_t strsize;

    fprintf(stderr, "[%ld]:(child) about to open FIFO %s...\n",
            (long) getpid(), fifoName);
    while (((fd = open(fifoName, O_WRONLY)) == -1) && (errno == EINTR));
    if (fd == -1) {
        fprintf(stderr, "[%ld]:failed to open named pipe %s for write: %s\n",
                (long) getpid(), fifoName, strerror(errno));
        return 1;
    }
    rval = snprintf(buf, BUFSIZE, "[%ld]:%s\n", (long) getpid(), idString);
    if (rval < 0) {
        fprintf(stderr, "[%ld]:failed to make the string:\n", (long) getpid());
        return 1;
    }
    strsize = strlen(buf) + 1;
    fprintf(stderr, "[%ld]:about to write...\n", (long) getpid());
    rval = r_write(fd, buf, strsize);
    if (rval != strsize) {
        fprintf(stderr, "[%ld]:failed to write to pipe: %s\n",
                (long) getpid(), strerror(errno));
        return 1;
    }
    fprintf(stderr, "[%ld]:finishing...\n", (long) getpid());
    return 0;
}
```

r\_write is the repeated form of write, in case of EINTR  
Available at  
<http://usp.cs.utsa.edu/usp/programs/chapter06/>

Example (part 2/3)

# Interprocess Communication

The parent reads what was written to a named pipe.

Example (part 3/3)

```
#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include "restart.h"
#define BUFSIZE 256
#define FIFO_MODES 0_RDONLY

int doifoparent(const char *fifoName) {
    char buf[BUFSIZE];
    int fd;
    int rval;

    fprintf(stderr, "[%ld]:(parent) about to open FIFO %s...\n",
            (long) getpid(), fifoName);
    while (((fd = open(fifoName, FIFO_MODES)) == -1) && (errno == EINTR)) ;
    if (fd == -1) {
        fprintf(stderr, "[%ld]:failed to open named pipe %s for read: %s\n",
                (long) getpid(), fifoName, strerror(errno));
        return 1;
    }
    fprintf(stderr, "[%ld]:about to read...\n", (long) getpid());
    rval = r_read(fd, buf, BUFSIZE);
    if (rval == -1) {
        fprintf(stderr, "[%ld]:failed to read from pipe: %s\n",
                (long) getpid(), strerror(errno));
        return 1;
    }
    fprintf(stderr, "[%ld]:read %.*s\n", (long) getpid(), rval, buf);
    return 0;
}
```

## Pipes, fifos and the client-server model

The **client–server model** is a distributed application structure that partitions tasks between the providers of a resource or service, called **servers**, and service requesters, called **clients**.

e.g. http server and web browser

*Apache, Microsoft Web Server*

This is opposed to the **peer-to-peer** (p2p) model, where two or more programs/computers (or simply *peers*) pool their resources and communicate in a decentralized system (each peer can act as both client and server).

e.g. bittorrent based file sharing

## Pipes, fifos and the client-server model

Client-server models have many flavors depending on the number of servers, number of clients, type of communication between them (simple request, request-reply), way of handling the clients (iterative, concurrent), etc.

*Server clients must be executed on the same system.*

*In windows FIFO's can be established on different systems,*

Consider the common scenario of a single server with multiple clients using fifo based communication, where the server provides the (trivial) service of assigning unique sequential numbers to each client that requests them.

The first impulse is usually to create a fifo between the server and the clients...this is bad idea; why?

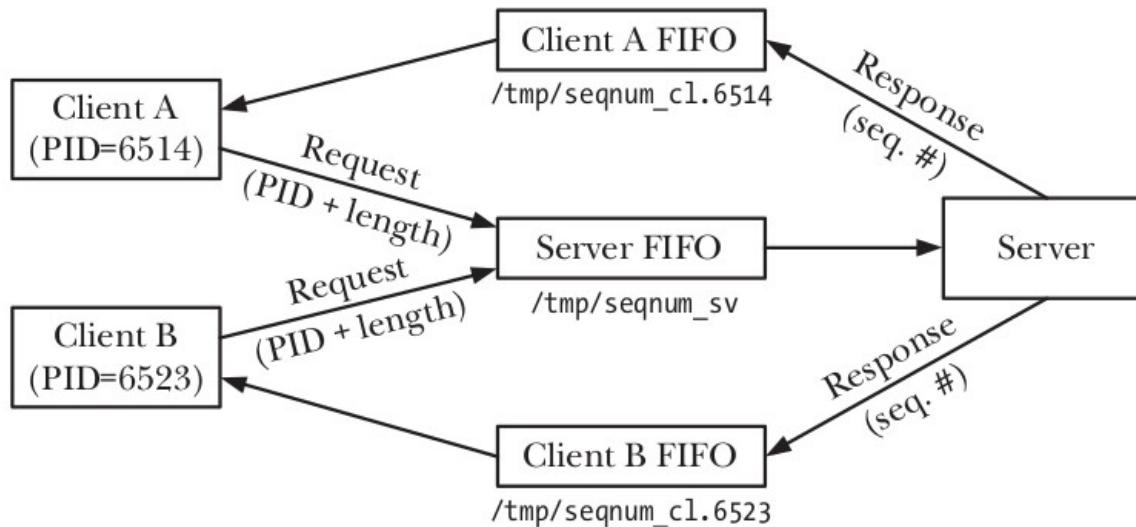
## Pipes, fifos and the client-server model

The first impulse is usually to create a fifo between the server and the clients...this is bad idea; why?

Because multiple clients would race to read from the FIFO, and possibly read each other's response messages rather than their own.

Therefore, each client creates a unique FIFO that the server uses for delivering the response for that client, and the server needs to know how to find each client's FIFO; either with a pre-agreed name template or by sending the fifo name as part of the request message.

# Pipes, fifos and the client-server model



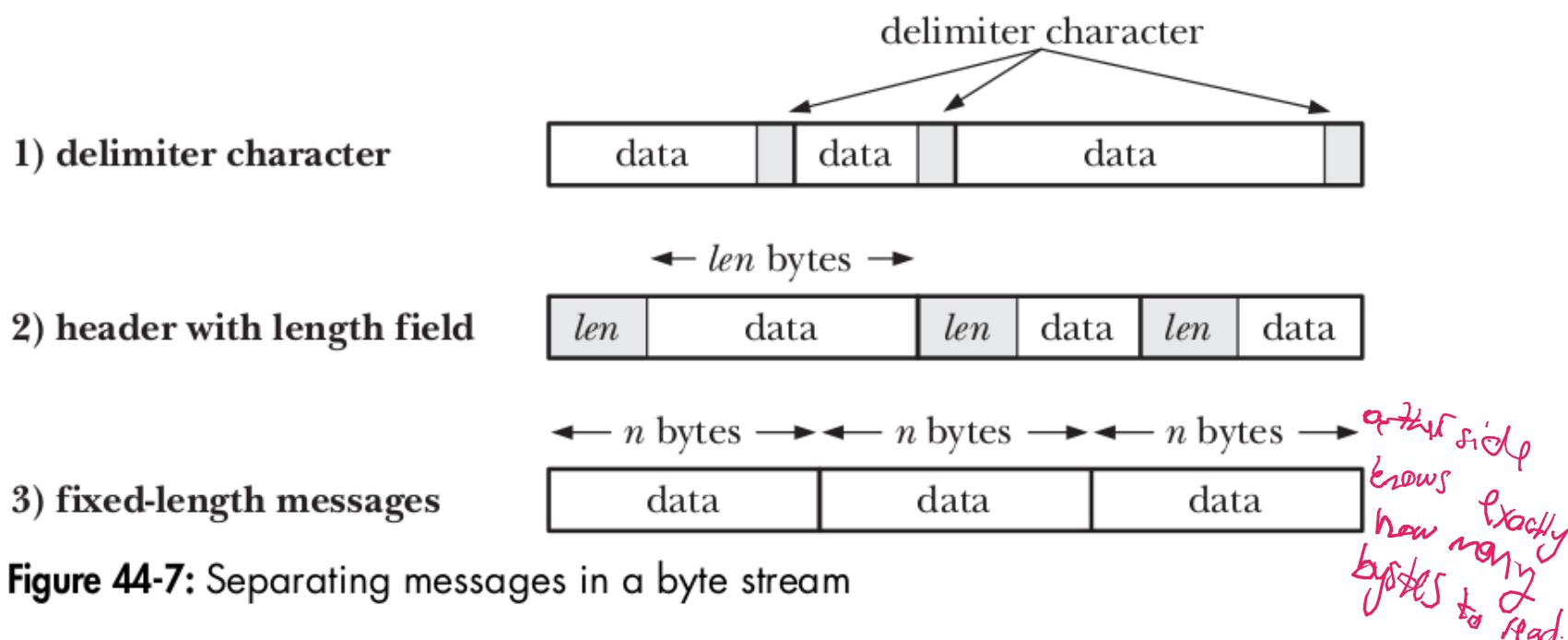
You will add a new FIFO every time that a client connects to the server.

**Figure 44-6:** Using FIFOs in a single-server, multiple-client application

For instance the names of the client fifos can consist of the pids of the clients, thus easily ensuring uniqueness.

# Pipes, fifos and the client-server model

Another practical issue to consider that **the data in pipes and FIFOs is a byte stream**; boundaries between multiple messages are not preserved. This means that when multiple messages are being delivered to a single process, such as the server in our example, then the sender and receiver must agree on some convention for separating the messages; e.g.



# Pipes, fifos and the client-server model

**Listing 44-6:** Header file for fifo\_seqnum\_server.c and fifo\_seqnum\_client.c

---

pipes/fifo\_seqnum.h

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include "tlpi_hdr.h"

#define SERVER_FIFO "/tmp/seqnum_sv"
                  /* Well-known name for server's FIFO */
#define CLIENT_FIFO_TEMPLATE "/tmp/seqnum_cl.%ld"
                  /* Template for building client FIFO name */
#define CLIENT_FIFO_NAME_LEN (sizeof(CLIENT_FIFO_TEMPLATE) + 20)
                  /* Space required for client FIFO pathname
                     (+20 as a generous allowance for the PID) */

struct request {
    pid_t pid; process : ✓          /* Request (client --> server) */
    int seqLen;           /* PID of client */
                           /* Length of desired sequence */
};

struct response {
    int seqNum;           /* Response (server --> client) */
                           /* Start of sequence */
};
```

One Server FIFO  
Multiple Client FIFO

# Pipes, fifos and the client-server model

```
#include <signal.h>
#include "fifo_seqnum.h" That's not concurrently,  
sequential example

int
main(int argc, char *argv[])
{
    int serverFd, dummyFd, clientFd;
    char clientFifo[CLIENT_FIFO_NAME_LEN];
    struct request req;
    struct response resp;
    int seqNum = 0; /* This is our "service" */
    /* Create well-known FIFO, and open it for reading */

    umask(0); /* So we get the permissions we want */
    if (mkfifo(SERVER_FIFO, S_IRUSR | S_IWUSR | S_IWGRP) == -1
        && errno != EEXIST)
        errExit("mkfifo %s", SERVER_FIFO);
    serverFd = open(SERVER_FIFO, O_RDONLY);
    if (serverFd == -1)
        errExit("open %s", SERVER_FIFO);

    /* Open an extra write descriptor, so that we never see EOF */
    dummyFd = open(SERVER_FIFO, O_WRONLY);
    if (dummyFd == -1)
        errExit("open %s", SERVER_FIFO);
    if (signal(SIGPIPE, SIG_IGN) == SIG_ERR)
        errExit("signal"); Instead use Sigaction
```

## Iterative server (part 1/2)

- creates the fifo
  - must be run before clients
  - open will block until a client opens as well
  - second open ensures that the server doesn't see EOF if all clients close the write end of the FIFO
  - SIGPIPE: occurs when writing without a reader;  
SIGPIPE kills by default;  
ignore it so as to receive EPIPE instead
- to keep pre-connection  
to ignore EPIPE  
if client don't the server*

# Pipes, fifos and the client-server model

```
for (;;) {                                /* Read requests and send responses */
    if (read(serverFd, &req, sizeof(struct request))
        != sizeof(struct request)) {
        fprintf(stderr, "Error reading request; discarding\n");
        continue;                            /* Either partial read or error */
    }

    /* Open client FIFO (previously created by client) */

    snprintf(clientFifo, CLIENT_FIFO_NAME_LEN, CLIENT_FIFO_TEMPLATE,
             (long) req.pid);
    clientFd = open(clientFifo, O_WRONLY);
    if (clientFd == -1) {                  /* Open failed, give up on client */
        errMsg("open %s", clientFifo);
        continue;
    }

    /* Send response and close FIFO */

    resp.seqNum = seqNum;
    if (write(clientFd, &resp, sizeof(struct response))
        != sizeof(struct response))
        fprintf(stderr, "Error writing to FIFO %s\n", clientFifo);
    if (close(clientFd) == -1)
        errMsg("close");

    seqNum += req.seqLen;                  /* Update our sequence number */
}
```

Iterative server (part 2/2)

block until  
reading

↳ we will be ready to respond to the next client.

# Pipes, fifos and the client-server model

```
#include "fifo_seqnum.h"

static char clientFifo[CLIENT_FIFO_NAME_LEN];

static void           /* Invoked on exit to delete client FIFO */
removeFifo(void)
{
    unlink(clientFifo);
}

int
main(int argc, char *argv[])
{
    int serverFd, clientFd;
    struct request req;
    struct response resp;
    if (argc > 1 && strcmp(argv[1], "--help") == 0)
        usageErr("%s [seq-len...]\\n", argv[0]);

    /* Create our FIFO (before sending request, to avoid a race) */

    umask(0);           /* So we get the permissions we want */
    snprintf(clientFifo, CLIENT_FIFO_NAME_LEN, CLIENT_FIFO_TEMPLATE,
             (long) getpid());
    if (mkfifo(clientFifo, S_IRUSR | S_IWUSR | S_IWGRP) == -1
        && errno != EEXIST)
        errExit("mkfifo %s", clientFifo);
    if (atexit(removeFifo) != 0) {    }  
    }  
    In order to  
    register remove file
    errExit("atexit");
```

Client (part 1/2)

```
$ ./fifo_seqnum_server &
[1] 5066
$ ./fifo_seqnum_client 3
0
$ ./fifo_seqnum_client 2
3
$ ./fifo_seqnum_client
5
```

If you are connecting to another machine, it's not fifo. we use socket.  
(Communication between processes  
located on different machine)

## Pipes, fifos and the client-server model

```
/* Construct request message, open server FIFO, and send request */  
  
req.pid = getpid();  
req.seqLen = (argc > 1) ? getInt(argv[1], GN_GT_0, "seq-len") : 1;  
  
serverFd = open(SERVER_FIFO, O_WRONLY);  
if (serverFd == -1)  
    errExit("open %s", SERVER_FIFO);  
  
if (write(serverFd, &req, sizeof(struct request)) !=  
    sizeof(struct request))  
    fatal("Can't write to server"); } write request  
  
/* Open our FIFO, read and display response */  
  
clientFd = open(clientFifo, O_RDONLY);  
if (clientFd == -1)  
    errExit("open %s", clientFifo);  
  
if (read(clientFd, &resp, sizeof(struct response)) !=  
    sizeof(struct response)) } wait for  
    fatal("Can't read response from server"); response & server  
  
printf("%d\n", resp.seqNum);  
exit(EXIT_SUCCESS); }
```

Client (part 2/2)

# BIL 344 System Programming

Week 7

---

Synchronization between processes

- Semaphores
- Shared Memory

# Semaphores

---

A semaphore is a kernel-maintained integer whose value is restricted to being greater than or equal to 0. Various operations (i.e., system calls) can be performed on a semaphore, including :

- setting the semaphore to an absolute value;
- adding a number to the current value of the semaphore;
- subtracting a number from the current value of the semaphore;
- waiting for the semaphore value to be equal to 0.

If semaphore is trying to be -1,  
it will be blocked because  
it cannot be -1.

## Semaphores

The last two of these operations may cause the calling process to block.

When lowering a semaphore value, the kernel blocks any attempt to decrease the value below 0.

Similarly, waiting for a semaphore to equal 0 blocks the calling process if the semaphore value is not currently 0.

In both cases, the calling process remains blocked until some other process alters the semaphore to a value that allows the operation to proceed, at which point the kernel wakes the blocked process.

# Semaphores

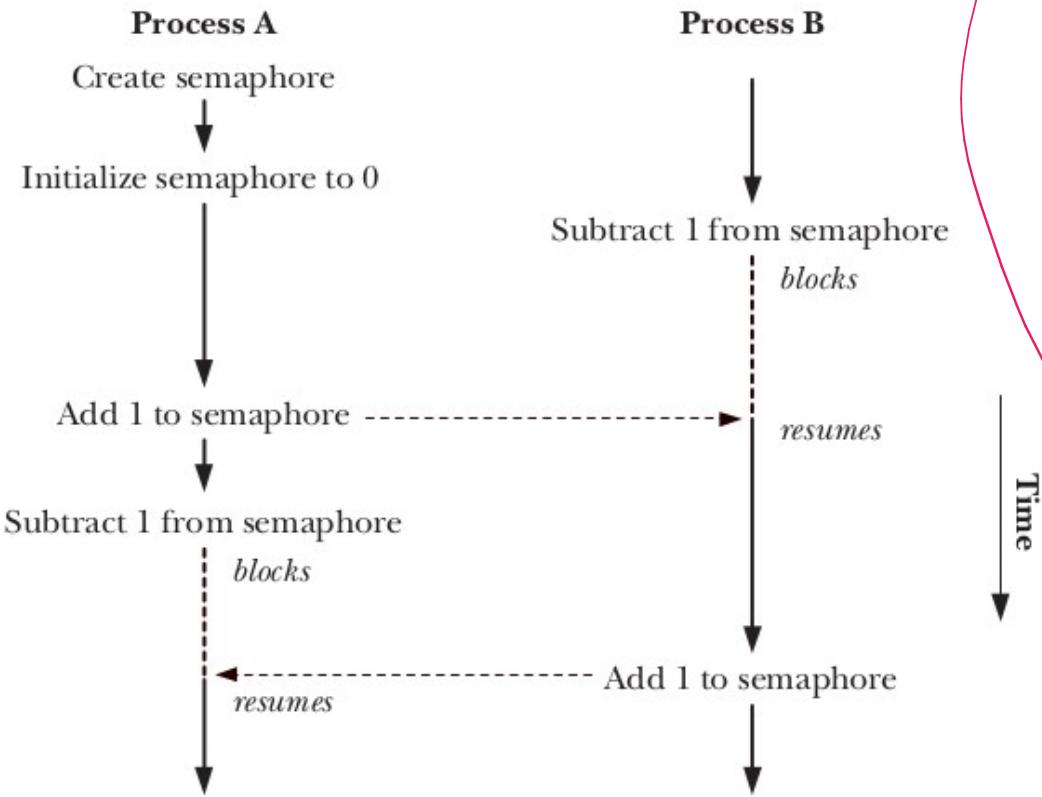


Figure illustrates the use of a semaphore to synchronize two processes

# Semaphores

---

In terms of controlling the actions of a process, a semaphore has no meaning in and of itself.

Its meaning is determined only by the associations given to it by the processes using the semaphore. Typically, processes agree on a convention that associates a semaphore with a shared resource, such as a region of shared memory.

Other uses of semaphores are also possible, such as synchronization between parent and child processes after *fork()*.

# Semaphores

---

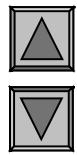
In POSIX:SEM terminology, the `wait` and `signal` operations are called *semaphore lock* and *semaphore unlock*, respectively.

We can think of a semaphore as an integer value and a list of processes waiting for a signal operation.

Semaphore implementations use **atomic operations** of the underlying operating system to ensure correct execution.

Suppose *process 1* must execute statement **a** before *process 2* executes statement **b**.

The semaphore sync enforces the ordering in the following pseudocode, provided that sync is initially 0.



Process 1 executes:

```
a;  
signal(&sync);
```

Process 2 executes:

```
wait(&sync);  
b;
```

If there are processes waiting for semaphore,  
changed from blocked queue to ready queue.

Posting  $\Rightarrow$  increase by 1

Be careful about deadlock.

What happens in the following pseudocode if semaphores **S** and **Q** are both initialized to 1?

Process 1 executes:

```
for( ; ; ) {  
    wait(&Q);  
    wait(&S);  
    a;  
    signal(&S);  
    signal(&Q);  
}
```

Process 2 executes:

```
for( ; ; ) {  
    wait(&S);  
    wait(&Q);  
    b;  
    signal(&Q);  
    signal(&S);  
}
```

If both waits for each other.

Care about order of executing.

You can only increment by 1 atomically. Not more.

# Semaphores

In this course, we will focus on two types of semaphore implementations :

*Simple* ← – POSIX Semaphores (or simply semaphore)

*Powerful and complicated* ← – SYSTEM V Semaphores (semaphore sets)

For a relatively simpler introduction to semaphore concept students are referred to old CSE 244 slides and notes.

# POSIX Semaphores

POSIX: SEM specifies **two types** of semaphores:

- **Named semaphores**: This type of semaphore has a name. By calling `sem_open()` with the same name, unrelated processes can access the same semaphore.  
*If success, return pointer to the semaphore.*
- **Unnamed semaphores**: This type of semaphore doesn't have a name; instead, it resides at an agreed-upon location in memory. Unnamed semaphores can be shared between processes or between a group of threads.  
*Shared memory*

\* note that some systems might not have a full implementation of POSIX semaphores (i.e. Linux 2.4)

# Opening a named semaphore

```
#include <fcntl.h>           /* Defines O_* constants */
#include <sys/stat.h>         /* Defines mode constants */
#include <semaphore.h>

sem_t *sem_open(const char * name , int oflag , ...
               /* mode_t mode , unsigned int value */ );
```

Returns pointer to a semaphore on success, or SEM\_FAILED on error and errno is set  
// #define SEM\_FAILED ((sem\_t \*) 0)

The `sem_open()` function creates and opens a new named semaphore or opens an existing semaphore.

Regardless of whether we are creating a new semaphore or opening an existing semaphore, `sem_open()` returns a pointer to a `sem_t` value

In Unix Everything is a file even semaphores.

## POSIX Semaphores: Named Semaphores

Never make a copy of semaphore.

Note that the results are **undefined** if we attempt to perform operations (`sem_post()`, `sem_wait()`, and so on) on a **copy** of the `sem_t` variable pointed to by the return value of `sem_open()`.

In other words, the following use of `sem2` is not permitted:

```
sem_t *sp, sem2;
sp = sem_open(...);
sem2 = *sp;
sem_wait(&sem2);
```

When a child is created via `fork()`, it inherits references to **all of the named semaphores** that are open in its parent. After the `fork()`, the parent and child can use these semaphores to synchronize their actions.

## Closing a Semaphore

When a process opens a named semaphore, the system records the association between the process and the semaphore.

The `sem_close()` function terminates this association, releases any resources that the system has associated with the semaphore for this process, and decreases the count of processes referencing the semaphore.

```
#include <semaphore.h>

int sem_close(sem_t *sem)
```

Returns 0 on success, -1 on error

**Closing a semaphore does not delete it.** For that purpose, we need to use `sem_unlink()`.

# Removing a Named Semaphore

The `sem_unlink()` function removes the semaphore identified by name and marks the semaphore to be destroyed once all processes cease using it.

```
#include <semaphore.h>

int sem_unlink(const char *name)
```

**Returns 0 on success, -1 on error**

# Semaphore Operations

```
#define _XOPEN_SOURCE 600          /* only required for timedwait case */
#include <semaphore.h>

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);

int sem_timedwait(sem_t * sem , const struct timespec * abs_timeout );
/* !!!!!! not available on all UNIX Implementations !!!!!! */
```

Returns 0 on success, -1 on error

The `sem_wait()` function decrements (decreases by 1) the value of the semaphore referred to by `sem`.

If the semaphore currently has a value greater than 0, `sem_wait()` returns immediately. If the value of the semaphore is currently 0, `sem_wait()` blocks until the semaphore value rises above 0; at that time, the semaphore is then decremented and `sem_wait()` returns. The `sem_trywait()` function is a nonblocking version of `sem_wait()` (returns the EAGAIN error in lieu of blocking).

# Semaphore Operations: posting/signalling

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

⚠️ No guarantee which process will continue after sem-post

Returns 0 on success, -1 on error

The `sem_post()` function increments (increases by 1) the value of the semaphore referred to by `sem`.

If the value of the semaphore was 0 before the `sem_post()` call, and some other process (or thread) is blocked waiting to decrement the semaphore, then that process is awoken, and its `sem_wait()` call proceeds to decrement the semaphore.

If multiple processes (or threads) are blocked in `sem_wait()`, then, if the processes are being scheduled under the default time-sharing policy, it is indeterminate which one will be awoken and allowed to decrement the semaphore.

# Retrieving the Current Value of a Semaphore

The `sem_getvalue()` function returns the current value of the semaphore referred to by `sem` in the `int` pointed to by `sval`.

```
#include <semaphore.h>

int sem_getvalue(sem_t * sem , int * sval );
```

Returns 0 on success, -1 on error

If one or more processes (or threads) are currently blocked waiting to decrement the semaphore's value, then the value returned in `sval` depends on the implementation.

There are two possibilities: 0 or a negative number whose absolute value is the number of waiters blocked in `sem_wait()` (On Linux the former behavior is adapted)

# POSIX Semaphores: Unnamed Semaphores

Unnamed semaphores (also known as memory-based semaphores) are variables of type `sem_t` that are stored in memory allocated by the application.

The semaphore is made available to the processes or threads that use it by placing it in an area of memory that they share.

Operations on unnamed semaphores use the same functions that are used to operate on named semaphores. In addition, two further functions are required:

- The `sem_init()` function initializes a semaphore and informs the system of whether the semaphore will be shared between processes or between the threads of a single process.  
*No access to other*
- The `sem_destroy(sem)` function destroys a semaphore

# Initializing an Unnamed Semaphore

The `sem_init()` function initializes the unnamed semaphore pointed to by `sem` to the value specified by `value`.

```
#include <semaphore.h>

int sem_init(sem_t * sem , int pshared , unsigned int value);
```

pshared  
0 or  
non-0

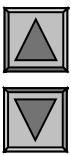
>Returns 0 on success, -1 on error

The `pshared` argument indicates whether the semaphore is to be shared between threads or between processes.

*will be a global variable will be shared.*

- If `pshared` is 0, then the semaphore is to be shared between the threads of the calling process. In this case, `sem` is typically specified as the address of either a global variable or a variable allocated on the heap
- If `pshared` is nonzero, then the semaphore is to be shared between processes. In this case, `sem` must be the address of a location in a region of shared memory (a POSIX shared memory object, a shared mapping created using `mmap()`, or a System V shared memory segment).

## Take a peek into the future (first the thread function definition )



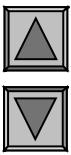
```
#include <semaphore.h>
#include <pthread.h>
#include "tlpi_hdr.h"

static int glob = 0;
static sem_t sem;

static void *threadFunc(void *arg)          /* Loop 'arg' times incrementing 'glob' */
{
    int loops = *((int *) arg);
    int loc, j;
    for (j = 0; j < loops; j++) {
        if (sem_wait(&sem) == -1)
            errExit("sem_wait");
        loc = glob;
        loc++;
        glob = loc;
        if (sem_post(&sem) == -1)
            errExit("sem_post");
    }
    return NULL;
}
```

→ Protecting the access to semaphore.

# Take a peek into the future



```
int main(int argc, char *argv[])
{
    pthread_t t1, t2;
    int loops, s;
    loops = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-loops") : 10000000;

    /* Initialize a thread-shared mutex with the value 1 */
    if (sem_init(&sem, 0, 1) == -1)
        errExit("sem_init");
    /* Create two threads that increment 'glob' */
    s = pthread_create(&t1, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");
    s = pthread_create(&t2, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");
    /* Wait for threads to terminate */
    s = pthread_join(t1, NULL); → It's like wait. Waiting for termination.
    if (s != 0)
        errExitEN(s, "pthread_join");
    s = pthread_join(t2, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");
    printf("glob = %d\n", glob);
    exit(EXIT_SUCCESS);
}
```

It's like wait. Waiting for termination.

# Destroying an Unnamed Semaphore

The `sem_destroy()` function destroys the semaphore `sem`, which must be an unnamed semaphore that was previously initialized using `sem_init()`. It is safe to destroy a semaphore only if no processes or threads are waiting on it.

```
#include <semaphore.h>
int sem_destroy(sem_t * sem);
```

*Before calling; make sure no one waiting for it.*

**Returns 0 on success, -1 on error**

After an unnamed semaphore segment has been destroyed with `sem_destroy()`, it can be reinitialized with `sem_init()`. An unnamed semaphore should be destroyed **before its underlying memory is deallocated**. (!!! beware of leaks !!!)

# Comparisons with Other Synchronization Techniques

## POSIX semaphores versus System V semaphores

- The POSIX semaphore interface does not support waiting/ decrementing atomically for  $n > 1$
- The POSIX semaphore interface is simpler than the System V semaphore interface. This simplicity is achieved without loss of functional power.
- POSIX named semaphores eliminate the initialization problem associated with System V semaphores (sem. sets)

*If you can solve with System V,  
you can also solve with POSIX  
even requires more work.*

# Comparisons with Other Synchronization Techniques

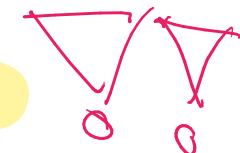
## *Difference* POSIX semaphores versus Pthreads mutexes

POSIX semaphores and Pthreads mutexes can both be used to synchronize the actions of threads within the same process, and their performance is similar.

However, **mutexes** are usually preferable, because the ownership property of mutexes enforces good structuring of code. By contrast, one thread can increment a semaphore that was decremented by another thread. This flexibility can lead to poorly structured synchronization designs.

*Only the thread locked mutex can unlock it.*

*Others can unlock it.*



# Comparisons with Other Synchronization Techniques

## POSIX semaphores versus Pthreads mutexes

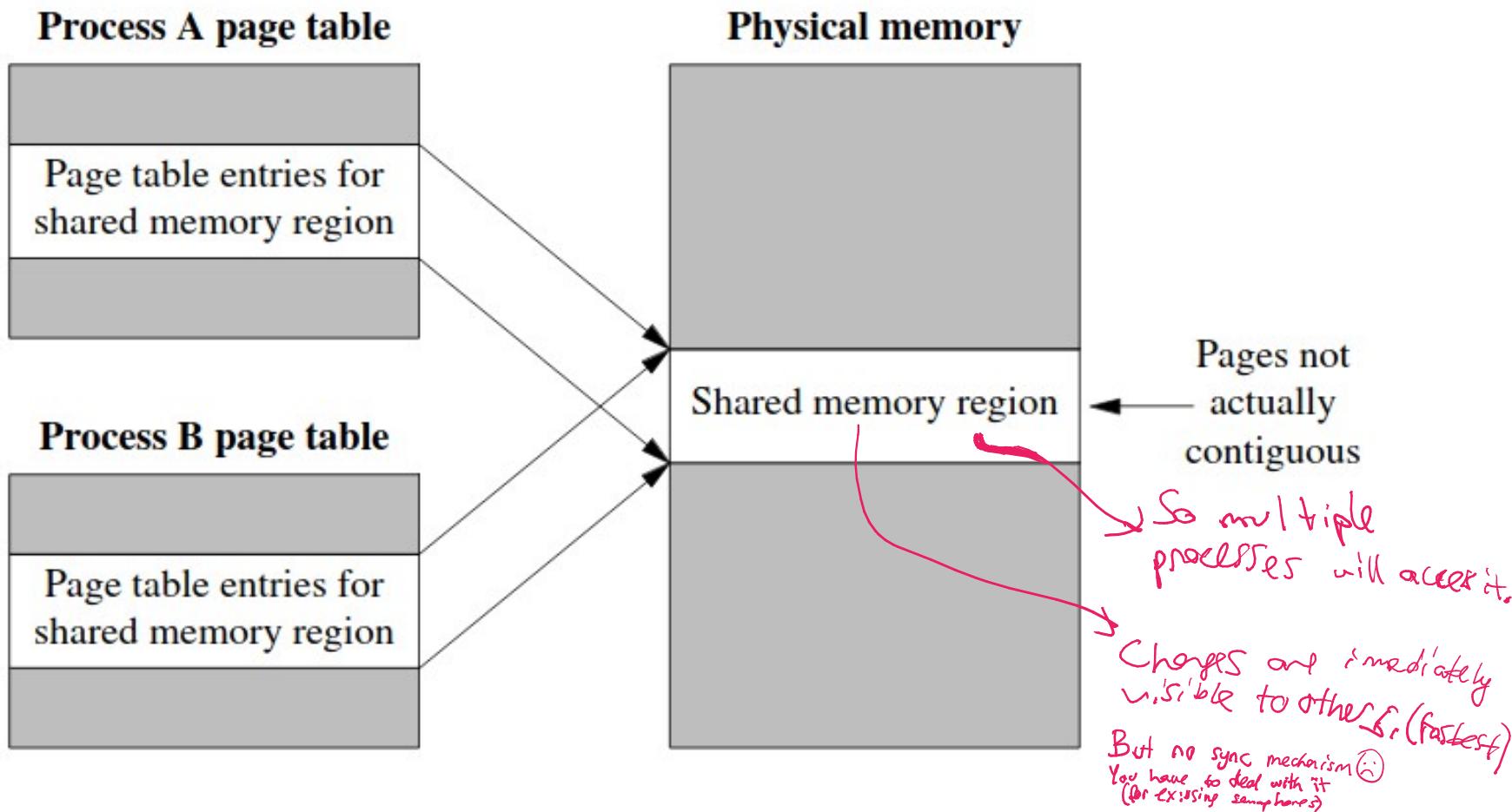
*↳ Have problem with Signal.*

There is one circumstance in which mutexes can't be used in a multi-threaded application and semaphores may therefore be preferable.

Because it is **async-signal safe**, the `sem_post()` function can be used from within a signal handler to synchronize with another thread. This is not possible with mutexes, because the Pthreads functions for operating on mutexes **are not async-signal-safe**. However, because it is usually preferable to deal with asynchronous signals by accepting them using `sigwaitinfo()` (or similar), rather than using signal handlers, this advantage of semaphores over mutexes is seldom required.

# Shared Memory

- Shared memory allows two or more processes to share the same region of physical memory.



# Shared Memory

- Since a shared memory segment becomes part of a process's user-space memory, no kernel intervention is required for IPC. All that is required is that one process copies data into the shared memory; that data is immediately available to all other processes sharing the same segment.
- This provides fast IPC by comparison with techniques such as pipes or message queues, where the sending process copies data from a buffer in user space into kernel memory and the receiving process copies in the reverse direction.
- The fact that IPC using shared memory is not mediated by the kernel implies that some method of synchronization is required so that processes don't simultaneously access the shared memory

# Other Shared Memory Techniques

---

- The unix world has many flavors

- 1) **System V shared memory**

- Original shared memory mechanism, still widely used
  - Sharing between unrelated processes

- 2) **Shared mappings –mmap(2)**

- Shared anonymous mappings (between related processes)
  - Shared file mappings (between unrelated processes, backed by file in FS)

- 3) **POSIX shared memory**

- Sharing between unrelated processes, without overhead of filesystem I/O
  - Intended to be simpler and better than older APIs

They must have already agreed about  
the name of shared memory segment

## POSIX Shared Memory

POSIX shared memory allows the user to share a mapped region between unrelated processes without needing to create a corresponding mapped file. To use a POSIX shared memory object, we perform two steps:

- 1) Use the `shm_open()` function to open an object with a specified name. The `shm_open()` function is analogous to the `open()` system call. It either creates a new shared memory object or opens an existing object.
- 2) Pass the file descriptor obtained in the previous step in a call to `mmap()` that specifies `MAP_SHARED` in the flags argument. This maps the shared memory object into the process's virtual address space.

## In the case of Linux

Kernel  
Reboot if

- They are implemented as files in a dedicated **tmpfs** filesystem
  - tmpfs is a virtual memory filesystem that employs swapspace when needed
- Objects have **kernel persistence**
  - Objects exist until explicitly deleted, or system reboots (**so, exiting does not mean it is deleted automagically**)
  - Can map an object, change its contents, and unmap
  - Changes will be visible to next process that maps object

# Creating Shared Memory Objects

The `shm_open()` function creates and opens a new shared memory object or opens an existing object. The arguments to `shm_open()` are analogous to those for `open()`.

```
#include <fcntl.h>                                /* Defines O_* constants */
#include <sys/stat.h>                             /* Defines mode constants */
#include <sys/mman.h>

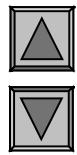
int shm_open(const char * name , int oflag , mode_t mode );
```

Returns file descriptor on success, or `-1` on error

When a new shared memory object is created, it initially has zero length. This means that, after creating a new shared memory object, we normally call `ftruncate()` to set the size of the object before calling `mmap()`. Following the `mmap()` call, we may also use `ftruncate()` to expand or shrink the shared memory object as desired.

It's attached to segment.

## Example Program : > \$ pshm\_create -c /demo\_shm 10000



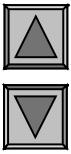
```
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include "tlpi_hdr.h"

static void usageError(const char *progName) {
    fprintf(stderr, "Usage: %s [-cx] name size [octal-perms]\n", progName);
    fprintf(stderr, " -c Create shared memory (O_CREAT)\n");
    fprintf(stderr, "-x Create exclusively (O_EXCL)\n");
    exit(EXIT_FAILURE);
}

int main(int argc, char *argv[])
{
    int flags, opt, fd;
    mode_t perms;
    size_t size;
    void *addr;

    flags = O_RDWR;
    while ((opt = getopt(argc, argv, "cx")) != -1) {
        switch (opt) {
            case 'c': flags |= O_CREAT; break;
            case 'x': flags |= O_EXCL; break; → If already exist, return error.
            default: usageError(argv[0]);
        }
    }
}
```

## Example Program :cont...



```
if (optind + 1 >= argc)
    usageError(argv[0]);


size = getLong(argv[optind + 1], GN_ANY_BASE, "size");
perms = (argc <= optind + 2) ? (S_IRUSR | S_IWUSR) :
    getLong(argv[optind + 2], GN_BASE_8, "octal-perms");

/* Create shared memory object and set its size */
fd = shm_open(argv[optind], flags, perms);
if (fd == -1)
    errExit("shm_open");
if (ftruncate(fd, size) == -1)
    errExit("ftruncate");

/* Map shared memory object */ Then kernel can attach whenever it sees good.
addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
if (addr == MAP_FAILED)
    errExit("mmap");

exit(EXIT_SUCCESS);
}
```

# Using Shared Memory Objects

---

- The programs given on the next two slides demonstrate the use of a shared memory object to transfer data from one process to another. The first program copies the string contained in its second command-line argument into the existing shared memory object named in its first command-line argument.
- Before mapping the object and performing the copy, the program uses *truncate()* to resize the shared memory object to be the same length as the string that is to be copied.

## Example Program :writing to a shared memory object

---

pshm/pshm\_write.c

```
#include <fcntl.h>
#include <sys/mman.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int fd;
    size_t len;           /* Size of shared memory object */
    char *addr;

    if (argc != 3 || strcmp(argv[1], "--help") == 0)
        usageErr("%s shm-name string\n", argv[0]);

    fd = shm_open(argv[1], O_RDWR, 0);      /* Open existing object */
    if (fd == -1)
        errExit("shm_open");

    len = strlen(argv[2]);    → Length of string to write,
    if (ftruncate(fd, len) == -1)          /* Resize object to hold string */
        errExit("ftruncate");
    printf("Resized to %ld bytes\n", (long) len);

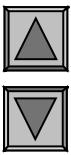
    addr = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0); → Attach memory space .
    if (addr == MAP_FAILED)
        errExit("mmap");

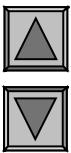
    if (close(fd) == -1)
        errExit("close");           /* 'fd' is no longer needed */

    printf("copying %ld bytes\n", (long) len);
    memcpy(addr, argv[2], len);           /* Copy string to shared memory */
    exit(EXIT_SUCCESS);
}
```

---

pshm/pshm\_write.c





```
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include "tlpi_hdr.h"

int
main(int argc, char *argv[])
{
    int fd;
    char *addr;
    struct stat sb;

    if (argc != 2 || strcmp(argv[1], "--help") == 0)
        usageErr("%s shm-name\n", argv[0]);

    fd = shm_open(argv[1], O_RDONLY, 0); /* Open existing object */
    if (fd == -1)
        errExit("shm_open");

    /* Use shared memory object size as length argument for mmap()
     * and as number of bytes to write() */

    if (fstat(fd, &sb) == -1) → Get memory space (bytes)
        errExit("fstat");

    addr = mmap(NULL, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
    if (addr == MAP_FAILED)
        errExit("mmap");

    if (close(fd) == -1); /* 'fd' is no longer needed */
        errExit("close");

    write(STDOUT_FILENO, addr, sb.st_size); → Where data is
    printf("\n"); → Size of string
    exit(EXIT_SUCCESS);
}
```

# Removing Shared Memory Objects

When a shared memory object is no longer required, it should be removed using `shm_unlink()`.

```
#include <sys/mman.h>

int shm_unlink(const char * name );
```

Returns 0 on success, or -1 on error

The `shm_unlink()` function removes the shared memory object specified by `name`. Removing a shared memory object doesn't affect existing mappings of the object (which will remain in effect until the corresponding processes call `munmap()` or terminate), but prevents further `shm_open()` calls from opening the object. Once all processes have unmapped the object, the object is removed, and its contents are lost.

## Overall...

Inter process communication

- Shared Memory provides fast IPC, and applications typically must use a semaphore (or other synchronization primitive) to synchronize access to the shared region.
- Once the shared memory region has been mapped into the process's virtual address space, it looks just like any other part of the process's memory space.
- The system places the shared memory regions within the process virtual address space .
- Assuming that we don't attempt to map a shared memory region at a fixed address, we should ensure that all references to locations in the region are calculated as offsets (rather than pointers), since the region may be located at different virtual addresses within different processes .

# Overall...

e.g. a linkedlist stored in shared memory.

```
struct node{  
    DATA d;  
    struct node * next;  
}  
  
struct node * shm = (struct node *) mmap(...);
```

Assume process P1 is attached to shm at address 10.000.

In which case next could point to address 10.030 for instance.

→ If every part has 30 bytes  
10000 + 30

If process P2 is attached to the shm at address 55.000 then what happens when it tries to access shm->next?

→ So if you use shared memory segment;  
instead of linked list, use array indexes.  
(Don't use pointers inside shared memory segment)

# Overall...

Solution: use indexes instead of pointers in shared memory segments.

```
struct node{  
    DATA d;  
    int next;  
};
```

The linkedlist becomes an array `shm` of nodes

e.g. `shm[shm[0].next].d`

In the general case pointers need to be replaced by offsets w.r.t. the base address of the shared memory segment.

## Overall...

fork() : a child process inherits its address space from its parent process, hence all shared memory segments are inherited at the same addresses.

Use ~~unmap~~

exit() : all shared memory segments are detached prior to the destruction of the process.

exec() : same as exit()

# BIL 344 System Programming

Week 8

## Synchronizing with semaphores: classic problems

- Producer/consumer
- Dining philosophers
- Cigarette smokers
- Synchronization barrier
- Readers/writers
- Sleeping barber

Monitor  
more complex

# Synchronization

Example: a stack consisting of a shared array T and an index S

<b>push (v)</b>	<b>v</b> $\leftarrow$ <b>pop</b>	<b>T</b> = {1, 2, 3, 4, 5}
++S	v=T [ S ]	S = 4
T [ S ]=v	--S	T = {1, 2, 3, 4, 6}
	return v	S = 4

**Process 1 pushes 6**

++S

**context switch**

→ Change execution  
another process

Change from  
one task to another  
task

**Process 2 pops #!\_?**

v=T [ S ] // wrong  
--S  
return v

**context switch**

T [ S ]=v // wrong

# Synchronization

There is **no way** of stopping process scheduling or predicting it reliably.

A **critical section** is a block of code using a shared resource, such as the stack in our example.

The golden rule is that at any given moment, there must be at most one process (or thread) inside a critical section.

This way even if the kernel schemes against us, no harm can come to our resources.

# Synchronization

Assuming that m is a semaphore initialized to 1.

Process p1	Process p2	Should be same semaphore
wait (m)	wait (m)	
push (v)	v = pop ()	// critical section
post (m)	post (m)	

A semaphore acquiring only the values 0 and 1 is called a binary semaphore, or a mutex (**mutual exclusion**).

Generally semaphores are used to model “a number of available resources” (that’s why they are called **counting semaphores**).

# Consumer must wait until there is sth to consume; (to solve; named semaphore full)

## Synchronization

The **producer-consumer** model is by far the most widely encountered synchronization model. A producer process produces data, and a consumer process consumes the said data.

1) Unbounded buffer case: the consumer process must execute only if there is something to consume in the buffer; otherwise it must wait.

The `full=0` semaphore represents the number of products in the buffer

<u>Most frequently encountered problem</u>	
<u>Producer</u>	<u>Consumer</u>
<code>while(true)</code>	<code>while(true)</code>
<code>    add(buffer, data)</code>	<code>    wait(full) <i>Until full</i></code>
<code>    post(full)</code>	<code>    take(buffer) <i>is greater than 0, block when 1, decrement</i></code>

Semaphore

This should work...right?

locking  
storage area

## Synchronization

What happens if both processes enter the buffer at the same time?

Then the buffer becomes corrupted like our stack! We need to protect the access to the critical section through a semaphore/mutex  $m = 1$ !

Initially, it's unlocked.  $\rightarrow$  Surround CS with wait and post.

Producer

while (true)

wait (m)

add (buffer, data)

post (m)

post (full)

Consumer

while (true)

wait (full)

wait (m)

take (buffer)

post (m)

Surrounding  
critical section

At most one process in critical section.

We solved the underflow problem, but what about the overflow problem?

# Synchronization

## 2) Bounded buffer case

Now we also have an upper limit to our buffer. The producer must not produce if the buffer is full! Empty spaces are now a resource too!

Semaphore full: number of products in the buffer

Semaphore empty: number of empty spaces in the buffer

Semaphore m: concurrent access lock

*meaning of none of the  
Semaphore is up to us.*

Initially

full=0 // no products

empty=N // size of the buffer

m=1 // unlocked

*Represents  
empty spaces*

# Synchronization

## Producer

```
while(true)
```

Order is important, if was reverse, the producer will infinitely.

```
wait(empty)  
wait(m)  
add(buffer, data)  
post(m)  
post(full)
```

Locking storage area

## Consumer

```
while(true)
```

```
wait(full)  
wait(m)  
take(buffer)  
post(m)  
post(empty)
```

At any given moment  $\text{empty} + \text{full} \leq N$

# Synchronization

The order of waits is crucial! Let's see what happens if we exchange them.



Producer	Consumer
while(true)	while(true)
<b>wait(m)</b>	wait(full)
<b>wait(empty)</b>	wait(m)
add(buffer, data)	take(buffer)
post(m)	post(m)
post(full)	post(empty)

Imagine the producer getting the lock and then encountering a full buffer..the system will be blocked indefinitely!

# Synchronization

What happens when the consumer needs more than 1 resource?

Process P1 needs 3 resources and process P2 needs 2 resources.

Initially we have  $s=2$  resources.

	P1	P2
<b>context switch</b>	wait(s) // $s=1$	
		wait(s) // $s=0$
<b>context switch</b>	wait(s) // blocked	wait(s) // blocked
	wait(s) // blocked	<i>When semaphore value is 0, block.</i>

*Deadlock*

It's a pity, process 2 could have been served with 2 resources; now they'll have to wait until some other process calls post.

# Synchronization

Calling `wait k times` is not the same as an atomic wait decreasing the semaphore by k.

This functionality is provided readily by System V semaphores; (POSIX semaphores can do it too, albeit indirectly :)

P1

Atomic decrement by 3

`wait(s, 3)`

// work

`post(s, 3)`

Atomic (Sufficient)

P2

`wait(s, 2)`

// work

`post(s, 2)`

If value is 3  
decrement,  
if less  
blocks

# Synchronization

**Calling `wait`  $k$  times is not the same as an atomic `wait` decreasing the semaphore by  $k$ .**

This functionality is provided readily by System V semaphores; (POSIX semaphores can do it too, albeit indirectly :)

P1

```
wait(s, 3)
```

```
// work
```

```
post(s, 3)
```

P2

```
wait(s, 2)
```

```
// work
```

```
post(s, 2)
```

# System V IPC

---

Unix System V (System 5, SysV) is one of the first commercial versions of Unix (AT&T, 1983). As of 2020 there are 3 variants based on it: IBM AIX, HP-UX and Oracle's Solaris

System V IPC is widely supported by \*x flavors, and is older than POSIX (IEEE-1988-2017); is nowadays present in POSIX-XSI

Linux has supported System V IPC since before, while full POSIX IPC has been available since kernel > 2.6

System V IPC has the same tools as POSIX IPC, with varying degrees of differences

# Some info on standards

- POSIX (Portable Operating System Interface) IEEE (1988-)
- SUS (Single Unix Specification) IEEE and The Open Group (industry) (mid 1990s) a.k.a. POSIX XSI (“extended posix”)
- POSIX is a subset of SUS
- Certified SUS systems: AIX, HP-UX, macOS, Solaris,etc
- Mostly POSIX/SUS compliant: Android NDK, FreeBSD, Linux, OpenBSD, etc
- Certification is not free.

Atomic decrement of multiple values  
**System V semaphores**  
Atomic decrement of multiple semaphores

System V semaphores are more powerful than POSIX semaphores, but are heavy-weight; main difference: this is a set of semaphores

General steps:

- Create or open a semaphore set using `semget()`
- Initialize the semaphores in the set using `semctl()` `SETALL` or `SETVAL` operation
- Perform operations on semaphore values using `semop()`
- When all processes have finished using the semaphore set, remove the set using the `semctl()` `IPC_RMID` operation

# System V **semaphores**

It's like  
for counting

```
#include <sys/types.h>
```

```
#include <sys/sem.h>
```

```
int semget(key_t key, int nsems, int semflg);
```

Returns the semaphore set identifier on success, or -1 on error

The semget() system call creates a new semaphore set or obtains the identifier of an existing set (uninitialized values)

**key**: unique integer identifier of the IPC object; usually set to IPC\_PRIVATE or generated through ftok()

**nsems**: number of semaphores in the set

**semflg**: bit mask of permissions

# System V semaphores

```
#include <sys/types.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int cmd /*, union semun */);
```

Returns a nonnegative integer on success and -1 on error

The `semctl()` system call performs a variety of control operations on a semaphore set or on an individual semaphore within a set

semid: identifier of the semaphore set

semnun: number of particular semaphore in the set (starts from 0)

cmd: operation to be performed (GET/SETVAL, GET/SETALL, IPC\_STAT, etc)

Depending on `cmd`'s value a fourth argument is supplied

# System V semaphores

Arguments and return values for all semctl commands are conveyed by semun, that must be **defined by your programs** as follows:

*If you want, you have to define it*

```
union semun {  
    int val; /* individual semaphore value */  
    struct semid_ds * buf; /* semaphore data structure */  
    unsigned short* array; /* multiple semaphore values */  
    struct seminfo* __buf; /* linux specific */  
};
```

*→ usually unnecessary*

semctl allows access to a wealth of information: PID of process that last modified a semaphore, number of processes currently waiting for a semaphore, time of last change, etc

# System V semaphores

```
/* Initialize a binary semaphore with a value of 1. */
int binary_semaphore_initialize (int semid)
{
    union semun argument;
    unsigned short values[1];
    values[0] = 1;
    argument.array = values;
    return semctl (semid, 0, SETALL, argument);
}
```

↓  
If you use this  
then this is unnecessary

# System V semaphores

```
#include <sys/types.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned int nsops);
```

Returns 0 on success, or -1 on error

The `semop()` system call performs one or more operations on the semaphores

`semid`: identifier of the semaphore set

`sops`: array that contains the operations to be performed

`nsops`: size of `sops`

```
struct sembuf {
    unsigned short sem_num; /* Semaphore number (starts from 0) */
    short sem_op;           /* Operation to be performed */
    short sem_flg; /* Operation flags (IPC_NOWAIT and SEM_UNDO) */
};
```

# System V semaphores

```
struct sembuf {  
    unsigned short sem_num; /* Semaphore number */  
    short sem_op;           /* Operation to be performed */  
    short sem_flg;          /* Operation flags (IPC_NOWAIT and SEM_UNDO) */  
};
```

`sem_op > 0`: added to the semaphore; **post**

`sem_op == 0`: block until the semaphore is zero; **zero** *Only continue when 0*

`sem_op < 0`: subtracted from the semaphore; **wait**

In case of a signal, `semop` is interrupted and `errno` is set to `EINTR`. It is not automatically restarted.

*Use max which keeps on while (--) & if errno != --)*

`IPC_NOWAIT`: unblocking operation, `errno` is set to `EAGAIN` instead

`SEM_UNDO`: the operation will be automatically undone when the process terminates

# System V semaphores

The fact that semaphore creation and initialization must be performed by separate system calls, instead of in a single atomic step, leads to possible race conditions when initializing a semaphore.

Solutions:

- 1) **avoidance**: ensure that a single process is in charge of creating & initializing the semaphore
- 2) A trick with the **sem\_otime** field in the semid\_ds data structure based on the historical (and now standard) fact that the field is set to 0 upon creation and only changes with a semop. The process that ~~does not~~ create the semaphore can wait until the first process has both initialized the semaphore and executed a no-op semop() call that updates the sem\_otime field, but does not modify the semaphore's value

↳ Another process will check if 0

Your textbook has a full example

# System V semaphores

---

## Limits (customizable)

```
$ ipcs -l      // on my machine
----- Semaphore Limits -----
max number of arrays = 32000
max semaphores per array = 32000
max semaphores system wide = 1024000000
max ops per semop call = 500
semaphore max value = 32767
```

## Overall

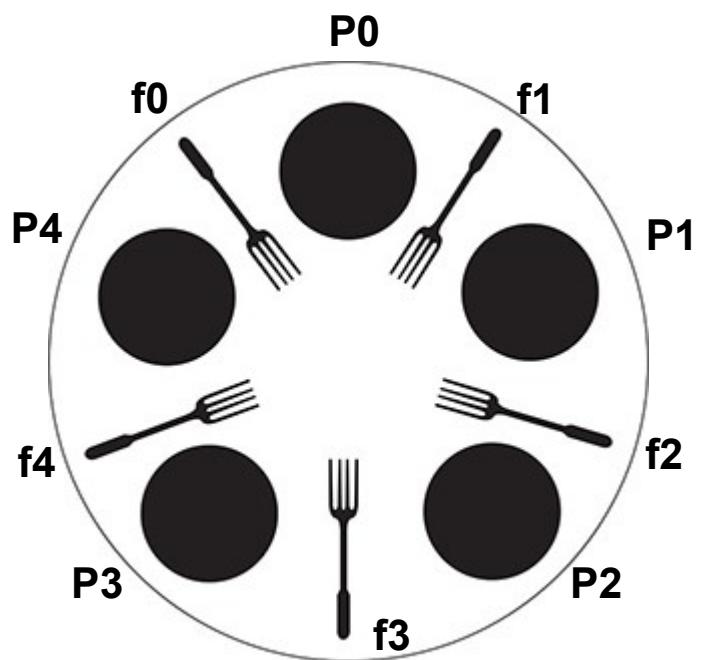
- Complex API, initialization race condition, keys
- + set granularity

# Synchronization

The **dining philosophers** is a classic synchronization problem introduced by Dijkstra.

Five philosophers are sitting around a dinner table, with a fork in between each pair of adjacent philosophers.

In order to eat, a philosopher needs to pick up the two forks that lie at the philosopher's left and right sides

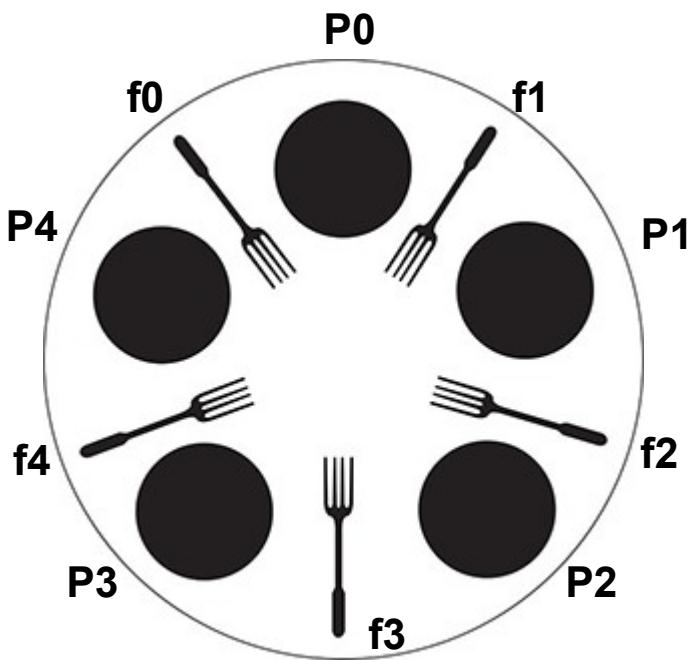


# Synchronization

Each philosopher alternates between thinking (non-critical section) and eating (critical section).

Since the forks are shared, there is a synchronization problem between philosophers (processes or threads).

The forks are our shared resources, so we'll have a semaphore representing each of them.



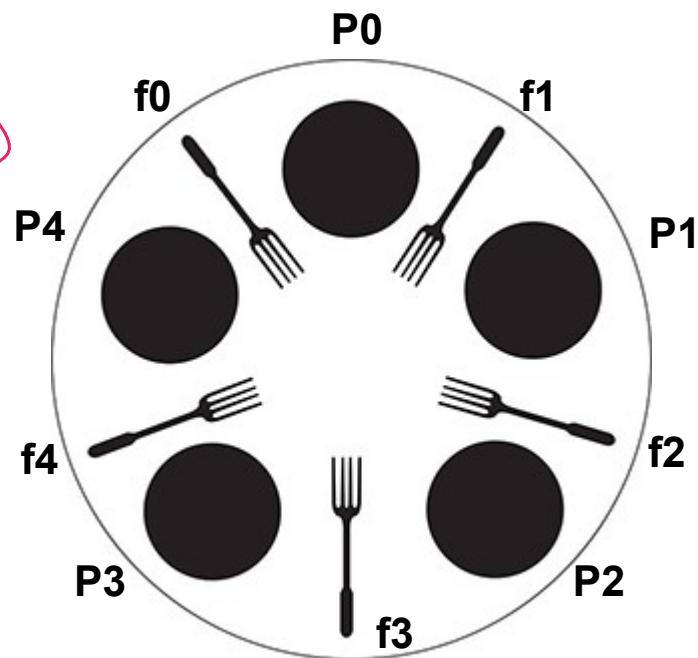
# Synchronization

A first attempt at solving the problem:

P2

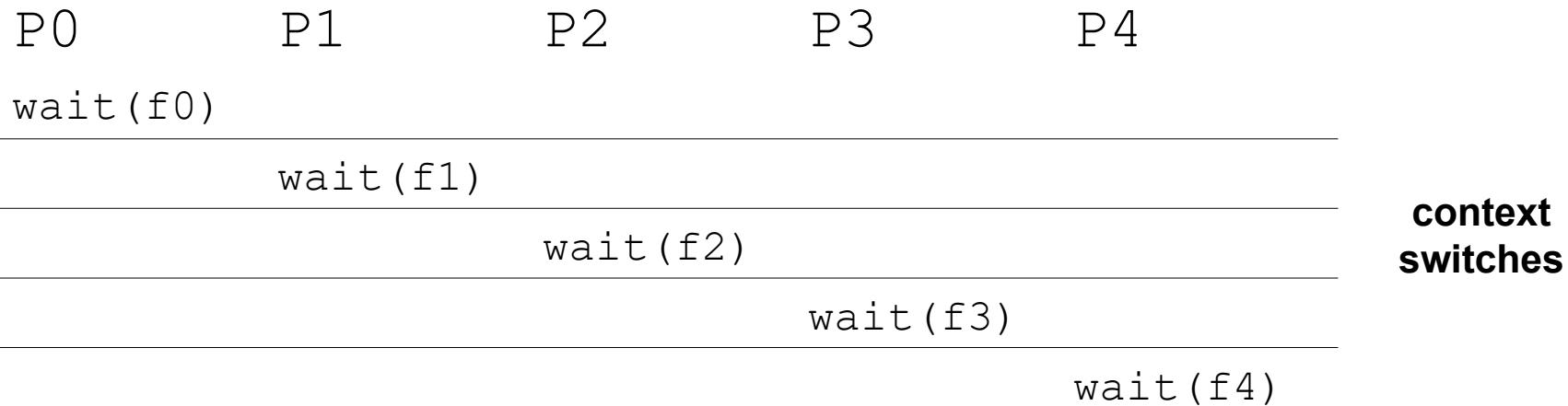
```
think()      // non critical section  
wait(f2)    // get fork  
wait(f3)    // get fork  
eat()       // crit. section  
post(f2)    // put down fork  
post(f3)    // put down fork
```

Get 2 forks or no fork.



# Synchronization

This scheduling can happen if the kernel dislikes you:



All philosophers end up starving even though 2 of them could have been served.

Main challenge: a philosopher must either get both forks, if available, or otherwise none!

# Synchronization-SysV

Easy to solve with IPC/System V semaphores

```
int forks = semget(IPC_PRIVATE, 5, IPC_CREAT|IPC_EXCL|0600);  
...  
int getFork(int i){  
    // for the i-th philosopher  
    struct sembuf ops[2]; 2 atomic operation  
    ops[0].sem_num = i; // semaphore to process  
    ops[1].sem_num = (i+1) % 5; // semaphore to process  
    ops[0].sem_op = ops[1].sem_op = -1; + decrement (Lock)  
    ops[0].sem_flg = ops[1].sem_flg = 0; → for now, unconnected  
    return semop(forks, ops, 2);  
}
```

*Don't care about key*

*5 semaphores*

*Either 2 or nothing (be blocked)*

## Synchronization-POSIX

---

More complicated with POSIX semaphores: if only four philosophers are allowed at the table at a time, deadlock is impossible!

If there are only four philosophers at the table, then in the worst case each one picks up a fork. Even then, there is a fork left on the table, and that fork has two neighbors, each of which is holding another fork. Therefore, either of these neighbors can pick up the remaining fork and eat.

We can control the number of philosophers at the table with a semaphore named **footman** that is initialized to 4 (multiplex pattern: enforces an upper limit on the number of concurrent threads/processes)

# Synchronization-POSIX

```
get_forks(i)
  wait(footman)
  wait(left_fork(i))
  wait(right_fork(i))

put_forks_down(i)
  post(left_fork(i))
  post(right_fork(i))
  post(footman)
```

Lim, if active  
philosophers

This may be a better s.

Get rid of deadlock  
footman (concurrency pattern)

2 forks guarantee

## Synchronization-POSIX

---

This solution also guarantees that no philosopher starves. Imagine that you are sitting at the table and both of your neighbors are eating. You are blocked waiting for your right fork.

Eventually your right neighbor will put it down, because eat can't run forever. Since you are the only thread/process waiting for that fork, you will necessarily get it next.

By a similar argument, you cannot starve waiting for your left fork.

# Synchronization

Another classic problem are the **cigarette smokers** (1971).

Assume a cigarette requires three ingredients to make and smoke: tobacco, paper, and matches.

There are three smokers around a table, each of whom has an infinite supply of one of the three ingredients — one smoker has an infinite supply of tobacco, another has paper, and the third has matches.

# Synchronization

---

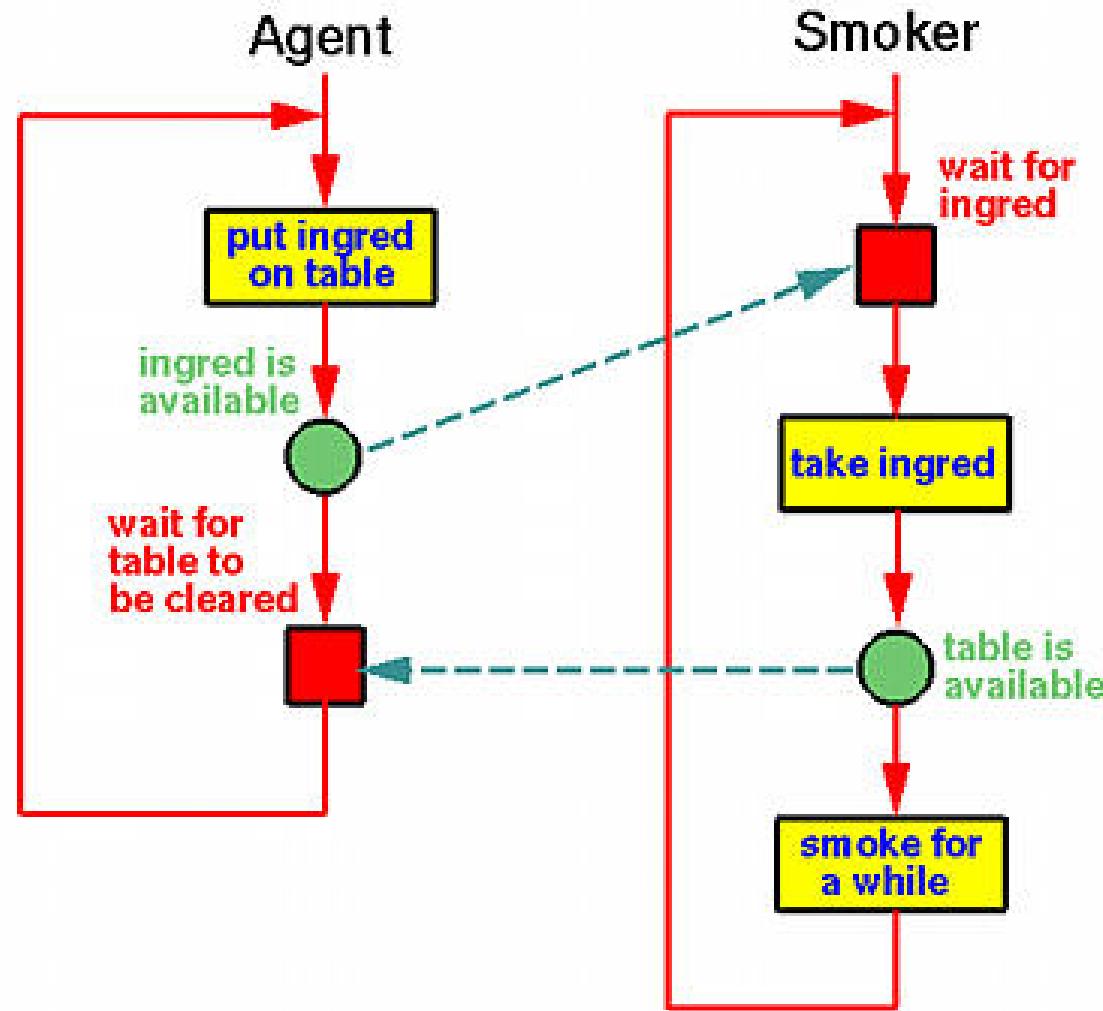
There is also a non-smoking agent who enables the smokers to make their cigarettes by arbitrarily (non-deterministically) selecting two of the supplies to place on the table.

The smoker who has the third supply should remove the two items from the table, using them (along with their own supply) to make a cigarette, which they smoke for a while.

Once the smoker has finished his cigarette, the agent places two new random items on the table. This process continues forever.

The ingredients are resources so we'll have one semaphore for each.

# Synchronization



# Synchronization

A first attempt (agentSem=1, tobacco=paper=matches=0)

[smoker-has matches]	[agent]
while(true)	while(true)
wait(tobacco)	wait(agentSem)
wait(paper)	post(paper)
get_ingred.() // crit. sect.	post(tobacco)
smoke()	
post(agentSem)	

Does it look good...?

# Synchronization

Looks good? No..

[has matches]

**wait (tobacco)**

wait (paper)

get\_ingred.()

smoke()

post (agentSem)

[has tobacco]

**wait (paper)**

wait (matches)

get\_ingred.()

smoke()

post (agentSem)

What if the agent brings **tobacco and paper**, but one smoker gets the tobacco and the other the paper? None will be able to smoke, the system will be deadlocked (good for the smokers, bad for the system)!

# Synchronization-SysV

Similarly to the dining philosophers, each smoker must either get both ingredients, if available, or otherwise none; in order to avoid effectively the deadlocks. e.g.:

```
[has matches]  
while(true) {  
    wait(tobacco, paper)  
    get_ingred()  
    smoke()  
    post(agentSem)  
}
```

**wait(tobacco, paper)** → Atomic wait for both  
Get both or don't get

# Synchronization-POSIX

A bit messy with POSIX semaphores. We have **three** helper processes called “pushers” that respond to the signals from the agent, keep track of the available ingredients, and signal the appropriate smoker

## Flags

isMatch=isTobacco=isPaper=0

**Semaphores** (3 for pushers, 3 for smokers) + 1 for agent +  
agentSem=1, tobacco=paper=matches=0 and a mutex m  
tobacco2=paper2=matches2=0 // for the smokers

The boolean variables indicate whether or not an ingredient is on the table. The pushers use tobacco2 to signal the smoker with tobacco, and the other semaphores likewise

Wait's for tobacco

## Synchronization

### Pusher A

```
wait (tobacco)  
wait (m)  
if isPaper:  
    isPaper=False  
    post (matches2)  
elif isMatch :  
    isMatch=False  
    post (paper2)  
else :  
    isTobacco=True  
post (m)
```

This pusher wakes up any time there is tobacco on the table.

If it finds `isPaper` true, it knows that Pusher B has already run, so it can signal the smoker with matches.

Similarly, if it finds a match on the table, it can signal the smoker with paper.

One pusher  
per ingredient

# Synchronization

## Pusher A

```
wait (tobacco)
wait (m)
if isPaper:
    isPaper=False
    post (matches2)
elif isMatch :
    isMatch=False
    post (paper2)
else :
    isTobacco=True
post (m)
```

## Pusher B

```
wait (paper)
wait (m)
if isTobacco:
    isTobacco=False
    post (matches2)
elif isMatch :
    isMatch=False
    post (tobacco2)
else :
    isPaper=True
post (m)
```

## Pusher C

```
wait (matches)
wait (m)
if isTobacco:
    isTobacco=False
    post (paper2)
elif isPaper :
    isPaper=False
    post (tobacco2)
else :
    isMatch=True
post (m)
```

# Synchronization

## **Smoker A**

wait (tobacco2)

get\_ingredients ()

smoke ()

post (agentSem)

## **Smoker B**

wait (paper2)

get\_ingredients ()

smoke ()

post (agentSem)

## **Smoker C**

wait (matches2)

get\_ingredients ()

smoke ()

post (agentSem)

## **Agent**

wait (agentSem)

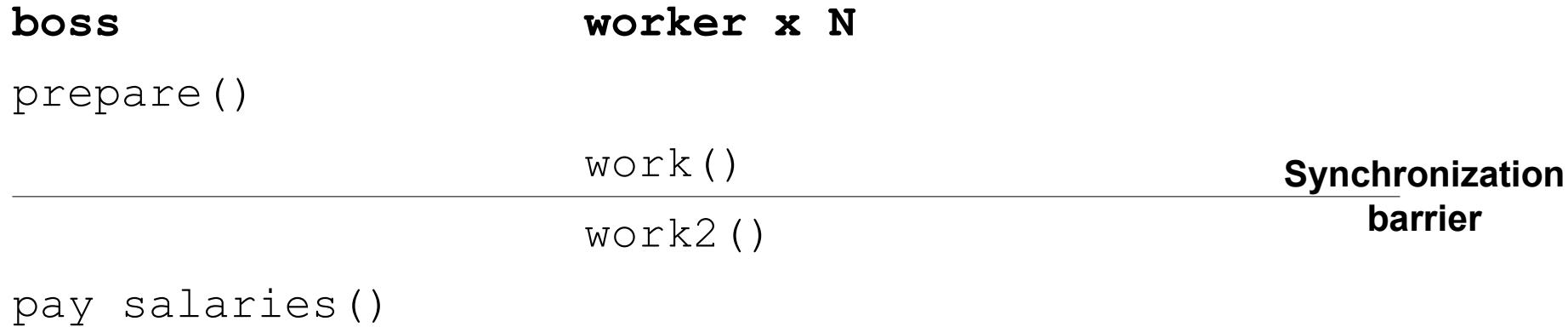
post (paper)

post (matches)

# Synchronization

The **synchronization barrier** is another often encountered problem. We have  $N$  processes (or threads), and any of them reaching this point must stop and cannot proceed unless all other threads/processes have reached this barrier.

e.g. worker processes and a boss process: the boss process does not pay their salary, unless all workers have completed a required task.



# Synchronization-SysV

We have a single semaphore T initialized at N.

Every process/thread reaching the rendez-vous point will call wait on it, and then wait for it to become zero. If T becomes zero, that means everyone has reached the barrier.

```
// worker  
wait(T)  
zero(T) // wait for T to become zero
```

zero is a call specific to System V/IPC semaphores.

When they move all together (barrier) waiting point

# Synchronization-SysV

```
int barr_init(int semid, int num, int N)
{return semctl(semid, SETVAL, N); } // initialization

int barr_wait(int semid, int num) {
    struct sembuf w,z;           // two distinct operations
    w.sem_num = z.sem_num = num;  → Not atomic
    w.sem_flg = z.sem_flg = 0;
    w.sem_op = -1;                // wait
    z.sem_op = 0;                  // zero
    return sem_op(semid,&w,1) !=-1 &&           // WAIT
           sem_op(semid,&z,1) !=-1? 0: -1; // ZERO
}
```

# Synchronization-POSIX

---

And with POSIX semaphores..

```
variable count=0
semaphores mutex=1, barrier=0
```

count keeps track of how many processes have arrived.

mutex provides exclusive access to count so that processes can increment it safely

barrier is locked (zero or negative) until all processes arrive; then it should be unlocked (1 or more)

# Synchronization-POSIX

And with POSIX semaphores..

```
wait (mutex)  
    count = count + 1
```

```
post (mutex)
```

```
if count == n:  
    post (barrier)
```

```
wait (barrier)  
post (barrier) → Conf. nice door  
random point
```

# Synchronization-POSIX

---

The **readers-writers** is another classic synchronization problem (1971).

We have a shared resource that two types of processes (or threads) access:

- The readers: that do not modify the resource
- The writers: that modify the resource

Readers can access the data in any order and number they like. However at most one writer is allowed to write at any given moment. And of course no reader should be reading while a writer is writing.

# BAD EXAMPLE

# Synchronization-POSIX

## READER

```
wait(mutex)           // avoid race
++readers
if(readers == 1)     // 1st reader
    wait(rsc)         // no writers allowed
post(mutex)
```

**read() // read the data**

```
wait(mutex)
--readers
```

```
if(readers == 0)     // last reader
    post(rsc)         // let the writer enter
post(mutex)
```

*Lock up // release when no readers left*

**Initially: mutex=1, rsc=1**

## WRITER

```
wait(rsc)
write()
post(rsc)
```

*While blocking, we are reading another reader may arrive, writer can get in.*

# Synchronization-POSIX

readers: the number of active readers

Shared locks: readers  
Exclusive locks: writers

rsc: makes sure we have only one writer

mutex: makes sure the shared variable readers is modified safely

While a writer is writing, the first reader will be blocked at `wait(rsc)` and the subsequent ones at `wait(mutex)`

In the database world this is known as a “lock”.

Readers ask the Database Management System (DBMS) for a “**shared lock**” and writers ask for a “**exclusive lock**”.

# Synchronization-POSIX

However, imagine the following scenario:

Reader1 is reading

Writer1 is blocked at wait(rsc)

Reader2 is reading

Reader1 exits (Writer1 is still blocked)

Reader3 and Reader4 are reading

Reader2 exits (Writer1 is still blocked)

Reader4 exits (Writer1 is still blocked)

Reader5 is reading...

i.e. if the readers are too many, a writer might have to wait indefinitely.

Not me man

# Synchronization-POSIX

---

Solution: prioritize writers!

i.e. no writer, once added to the queue, shall be kept waiting longer than absolutely necessary. This is also called **writers-preference**.

This is accomplished by forcing every reader to lock and release a “readtry” semaphore individually. The writers on the other hand don't need to lock it individually.

Only the first writer will lock the “readtry” and then all subsequent writers can simply use the resource as it gets freed by the previous writer. The very last writer must release the “readtry” semaphore, thus opening the gate for readers to try reading.

New  
rmutex → lock

# Synchronization-POSIX

## READER

```
wait(readTry)          // a reader is trying to enter
wait(rmutex)          // avoid race condition with other readers
readcount++;          // report yourself as a reader → We can see that
if (readcount == 1)    // if you are the first reader
    wait(rsc) → reader // lock the resource and prevent writers
post(rmutex);          // allow other readers
post(readTry)          // you are done trying to access the resource
```

## **read()**

```
wait(rmutex)          // avoids race condition with readers
readcount--;          // indicate you're leaving
if (readcount == 0)    // if you are last reader leaving
    post(rsc)          // you must release the locked resource
post(rmutex)          // release exit section for other readers
```

readTry=1, rmutex=1, rsc=1, readcount=0

Last reader release exclusive lock  
(Unlock shared variable)

~~Wm~~  
~~mutex~~  $\rightarrow$  wlock

# Synchronization-POSIX

## WRITER

```
    ↗ Check if there are other writers
wait(wmutex)    // avoids race conditions
                because we need to
                protect "writecount" variable.
writecount++;   // report yourself as a writer entering
if (writecount == 1) // if you're the first writer
    wait(readTry) // no new readers!
post(wmutex)    // release entry section
                ↗ Only one
                writer will
                have
                put an update
                on database
                at once
wait(rsc)       // prevents other writers
write()        // only 1 writer allowed here
post(rsc)       // release resource
wait(wmutex)    // reserve exit section
writecount--;   // indicate you're leaving
if (writecount == 0) // checks if you're the last writer
    post(readTry) // if you're the last writer, unlock the readers
post(wmutex)
```

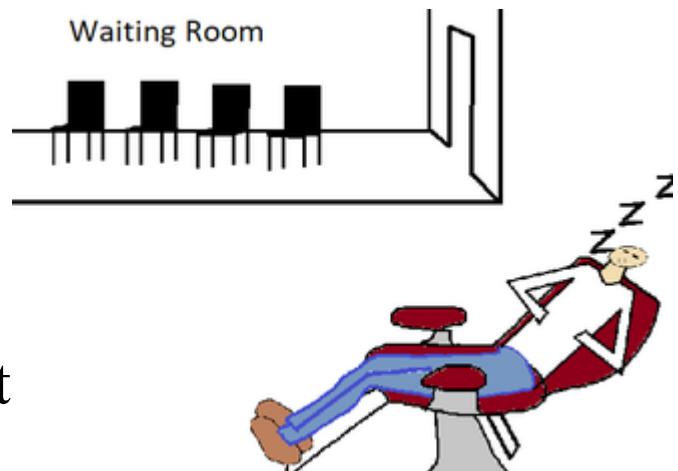
wmutex=1

# Synchronization-POSIX

Another famous problem is the **sleeping barber**; also attributed to Dijkstra (1965).

The barber shop has one barber and two rooms. The waiting room with N chairs, and the cutting room with a single chair.

**Barber:** When he finishes cutting a customer's hair, he dismisses the customer and goes to the waiting room to see if there are others waiting. If there are, he brings one of them back to the chair and cuts his hair. If there are none, he returns to the chair and sleeps in it



# Synchronization-POSIX

---

**Customer:** each customer, when he arrives, looks to see what the barber is doing. If the barber is sleeping, the customer wakes him up and sits in the cutting room chair. If the barber is cutting hair, the customer stays in the waiting room. If there is a free chair in the waiting room, the customer sits in it and waits his turn. If there is no free chair, the customer leaves.

All actions (cutting hair, etc) can take an unknown amount of time. This can cause a lot of issues.

# Synchronization-POSIX

---

**Issues:** for instance a customer may arrive and observe that the barber is cutting hair, so he goes to the waiting room. While they're on their way, the barber finishes their current haircut and goes to check the waiting room. Since there is no one there (the customer not having arrived yet), he goes back to their chair and sleeps. The barber is now waiting for a customer, but the customer is waiting for the barber.

Or, two customers may arrive at the same time when there happens to be a single seat in the waiting room. They observe that the barber is cutting hair, go to the waiting room, and both attempt to occupy the single chair.

binary  
Semaphore

## Synchronization-POSIX

// mutex; whether the barber is ready to cut or not

Semaphore barberReady = 0

// mutex to control access to the number of waiting room seats

Semaphore accessWRSeats = 1

// the number of customers currently waiting at the waiting room

Semaphore custReady = 0

// total number of free seats in the waiting room

int numberOffreeWRSeats = N

# Synchronization-POSIX

---

## Barber

```
while(true)  
  wait(custReady)    // acquire a customer - if none, sleep  
  wait(accessWRSeats)      // there is a customer  
  numberOfWorkingSeats += 1 // increase # of free seats  
  post(barberReady)        // ready to cut.  
  post(accessWRSeats)      // no need for chair lock any more  
  cutting_hair()
```

# Synchronization-POSIX

## Customer

```
wait(accessWRSeats)           // access waiting room chairs
if(numberOfFreeWRSeats > 0)   // If there are any free seats:
    numberOfFreeWRSeats -= 1  // sit down in a chair
    post(custReady)          // notify the barber that there is a customer
    post(accessWRSeats)       // release the chairs' lock
    wait(barberReady)         // wait until the barber is ready
    have_haircut()
else                           // there are no free seats
    post(accessWRSeats)       // release the lock
    leave_without_a_haircut()
```

*Get lock and access*

# BIL 344 System Programming

Week 9

---

## POSIX Threads

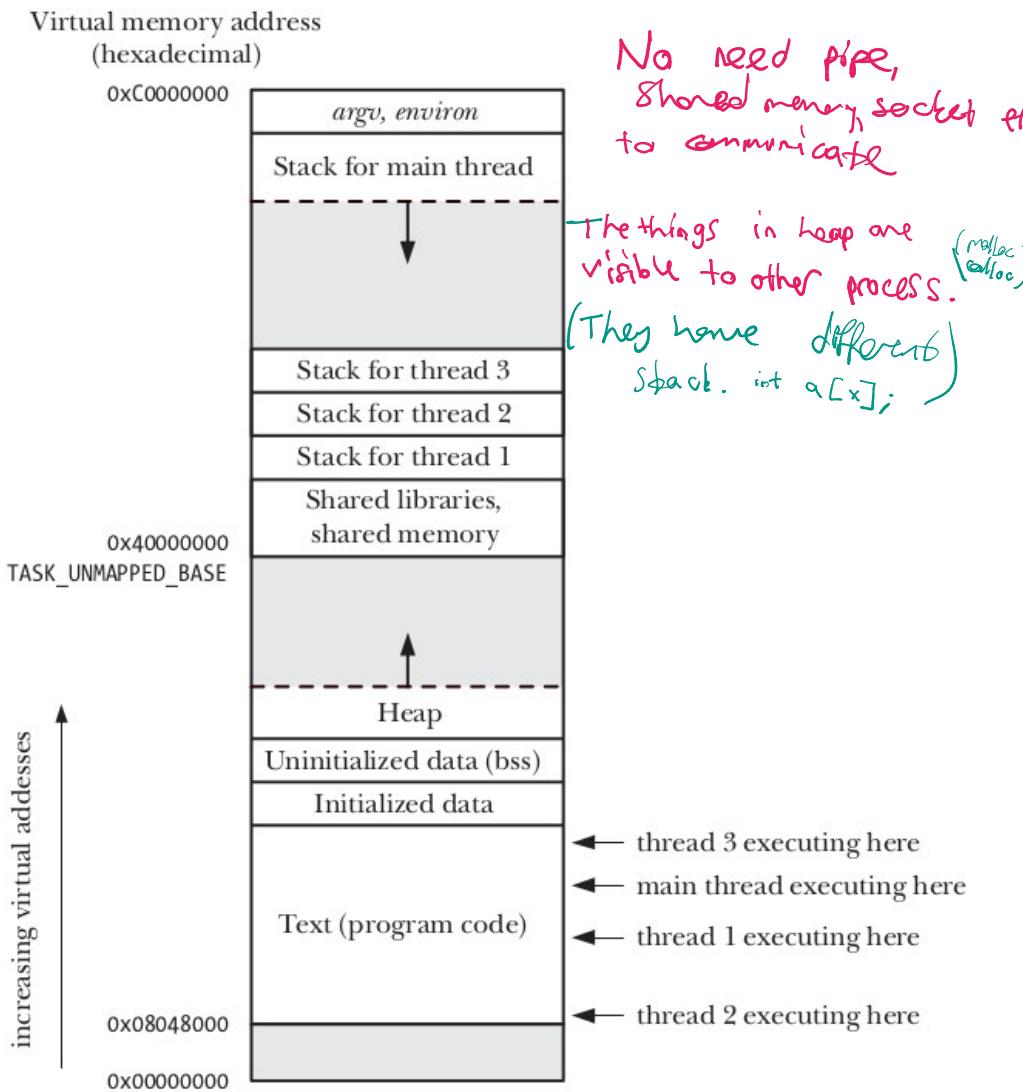
- Introduction to Threads
- POSIX Threads
- The Pthread API
- Threads vs Processes

# Threads

---

- Like processes, threads are a mechanism that permits an application to perform multiple tasks concurrently. A single process can contain multiple threads
- All threads are independently executing the same program, and they all share the same global memory, including the initialized data, uninitialized data, and heap segments.
- The threads in a process can execute concurrently. On a multiprocessor system, multiple threads can execute in parallel. If one thread is blocked on I/O, other threads are still eligible to execute.

# Threads in a Process



- As it can be seen from the figure all of the per-thread stacks reside within the same virtual address space. This means that, given a suitable pointer, it is possible for threads to share data on each other's stacks.
- This is occasionally useful, but it requires careful programming to handle the dependency that results from the fact that a local variable remains valid only for the lifetime of the stack frame in which it resides.
- Failing to correctly handle this dependency can create bugs that are hard to track down.

Figure 29-1: Four threads executing in a process (Linux/x86-32)

# Threads : Introduction

Threads offer advantages over processes in certain applications.

To give an example consider a network server design in which a parent process accepts incoming connections from clients, and then uses *fork()* to create a separate child process to handle communication with each client

While this approach works well for many scenarios, it does have the following limitations in some applications :

- It is difficult to share information between processes. Since the parent and child don't share memory, we must use some form of interprocess communication in order to exchange information between processes.
- Process creation with *fork()* is relatively expensive. The need to duplicate various process attributes such as page tables and file descriptor tables means that a *fork()* call is still time-consuming.

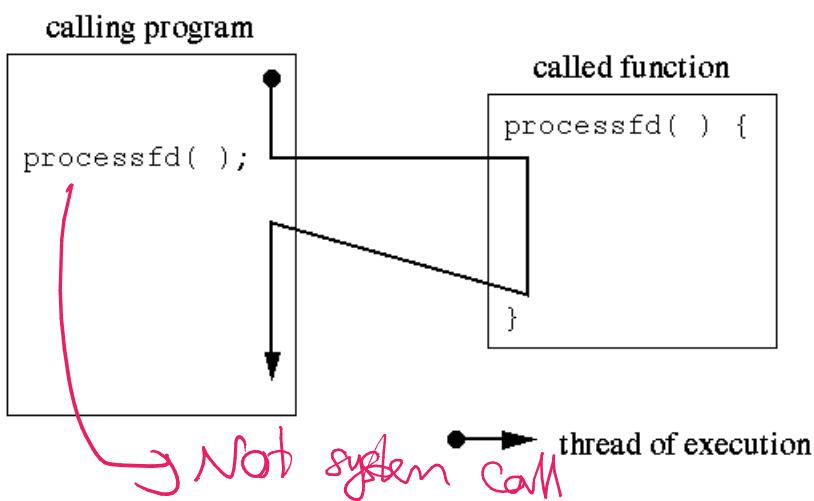
# Threads : Motivation

---

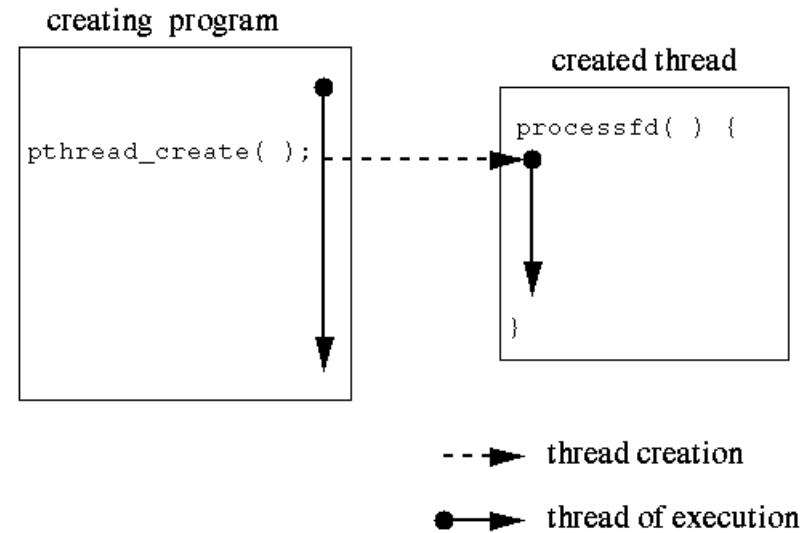
Threads address both of these problems:

- Sharing information between threads is easy and fast. It is just a matter of copying data into shared (global or heap) variables. However, in order to avoid the problems that can occur when multiple threads try to update the same information, we must employ some synchronization techniques.
- Thread creation is faster because many of the attributes that must be duplicated in a child created by *fork()* are instead shared between threads. In particular, copy-on-write duplication of pages of memory is not required, nor is duplication of page tables

# Threads : function call vs a Thread



Program that makes an ordinary call to `processfd` has a single thread of execution.



Program that creates a new thread to execute `processfd` has two threads of execution.

- When a new thread is created it runs concurrently with the creating process.
  - When creating a thread you indicate which function the thread should execute.

# Threads : function call vs a Thread

- A function that is used as a thread must have a special format.
- It takes a single parameter of type pointer to void and returns a pointer to void.
- The parameter type allows any pointer to be passed. This can point to a structure, so in effect, the function can use any number of parameters.
- The processfd function used above might have prototype:

```
void *processfd(void *arg);
```

- Instead of passing the file descriptor to be monitored directly, we pass a pointer to it.

You never know (unless you take precaution),  
which thread will run for how long/ which order.



## Function call and thread example

```
#include <stdio.h>
#include "restart.h"
#define BUFSIZE 1024

void docommand(char *cmd, int cmdsize);

void *processfd(void *arg) { /* process commands read from file descriptor */
    char buf[BUFSIZE];
    int fd;
    ssize_t nbytes;

    fd = *((int *) (arg));
    for ( ; ; ) {
        if ((nbytes = r_read(fd, buf, BUFSIZE)) <= 0)
            break;
        docommand(buf, nbytes);
    }
    return NULL;
}
```

### Function call

```
void *processfd(void *);
int fd;

processfd(&fd);
```

### Create a new thread to execute the function

```
void *processfd(void *arg);

int error;
int fd;
pthread_t tid;

if (error = pthread_create(&tid, NULL, processfd, &fd))
    fprintf(stderr, "Failed to create thread: %s\n",
    strerror(error));
```

# Thread Management

---

- A thread package usually includes functions for thread creation and thread destruction, scheduling, enforcement of mutual exclusion and conditional waiting
- A typical thread package also contains a runtime system to manage threads transparently (i.e., the user is not aware of the runtime system).
- When a thread is created, the runtime system allocates data structures to hold the thread's ID, stack and program counter value.
- The thread's internal data structure might also contain scheduling and usage information. The threads for a process share the entire address space of that process.

Some C functions  
are not thread safe.  
For example different threads executing printf. (most C functions are not thread safe)

## Thread Management

- When a thread allocates space for a return value, some other thread is responsible for freeing that space.  
*Whenever possible, a thread should clean up its own mess rather than requiring another thread to do it.*
- When creating multiple threads, do not reuse the variable holding a thread's parameter until you are sure that the thread has finished accessing the parameter.* As the variable is passed by reference, it is a good practice to use a separate variable for each thread.

# Thread Safety

---

- A hidden problem with threads is that they may call library functions that are not thread-safe, possibly producing spurious results
- A function is thread-safe if multiple threads can execute simultaneous active invocations of the function without interference.
- POSIX specifies that all the required functions, including the functions from the standard C library, be implemented in a thread-safe manner except for the specific functions
- Those functions whose traditional interfaces preclude making them thread-safe must have an alternative thread-safe version.

# Thread Safety

---

- Another interaction problem occurs when threads access the same data.
- In more complicated applications, a thread may not exit after completing its assigned task. Instead, a worker thread may request additional tasks or share information.

# User Threads versus Kernel Threads

Difference between User Level Threads and Kernel Level Threads (Important question)

- The two traditional models of thread control are user-level (green) threads and kernel-level threads.
- User-level threads usually run on top of an existing operating system. These threads are **invisible to the kernel** and compete among themselves for the resources allocated to their encapsulating process
- The threads are scheduled by a thread runtime system that is part of the process code
- Programs with user-level threads usually link to a special library in which each library function is enclosed by a jacket
- The jacket function calls the thread runtime system to do thread management before and possibly after calling the jacketed library function.

Concurrent (User Level Threads)

One library runs  $t_1$   
then stops and runs  $t_2$   
then again  $t_1$

## User Level Threads

Very fast  
and easy to schedule

- Functions such as `read` or `sleep` can present a problem for user-level threads because they may cause the process to block.
- To avoid blocking the entire process on a blocking call, the user-level thread library replaces each potentially blocking call in the jacket by a nonblocking version.
- The thread runtime system tests to see if the call would cause the thread to block. If the call would not block, the runtime system does the call right away. If the call would block, however, the runtime system places the thread on a list of waiting threads, adds the call to a list of actions to try later, and picks another thread to run.
- All this control is invisible to the user and to the operating system.

# User Level Threads

- User-level threads have low overhead, but they also have some disadvantages.
- The user thread model, which assumes that the thread runtime system will eventually regain control, can be thwarted by CPU-bound threads.
- A CPU-bound thread rarely performs library calls and may prevent the thread runtime system from regaining control to schedule other threads.  
*One thread can dominate scheduling,  
that's dangerous.  
~100 user-level threads have this problem.*
- The programmer has to avoid the lockout situation by explicitly forcing CPU-bound threads to yield control at appropriate points.  
*Starvation*      *Not truly parallel.*

# Kernel-level Threads

*Can run ~*

- With kernel-level threads, the kernel is aware of each thread as a schedulable entity and threads compete systemwide for processor resources *Can run on other process or some CPU*
- The scheduling of kernel-level threads can be almost as expensive as the scheduling of processes themselves, but kernel-level threads can take advantage of multiple processors.
- The synchronization and sharing of data for kernel-level threads is less expensive than for full processes, but kernel-level threads are considerably more expensive to manage than user-level threads.

# Hybrid threads

- Hybrid thread models have advantages of both user-level and kernel-level models by providing two levels of control
- The user writes the program in terms of user-level threads and then specifies how many kernel-schedulable entities are associated with the process
- The user-level threads are mapped into the kernel-schedulable entities at runtime to achieve parallelism. The level of control that a user has over the mapping depends on the implementation

# User Threads versus Kernel Threads

- The user-level threads are called threads and the kernel-schedulable entities are called lightweight processes
- The POSIX thread scheduling model is a hybrid model that is flexible enough to support both user-level and kernel-level threads in particular implementations of the standard.
- The model consists of two levels of scheduling—threads and kernel entities. The threads are analogous to user-level threads. The kernel entities are scheduled by the kernel. The thread library decides how many kernel entities it needs and how they will be mapped.

# User Threads versus Kernel Threads

---

- User-level threads run on top of an operating system
  - Threads are invisible to the kernel.
  - Link to a special library of system calls that prevent blocking
  - Have low overhead
  - CPU-bound threads can block other threads
  - Can only use one processor at a time.
- Kernel-level threads are part of the OS.
  - Kernel can schedule threads like it does processes.
  - Multiple threads of a process can run simultaneously on multiple CPUs.
  - Synchronization is more efficient than for processes but less than for user-level threads.

# Pthreads API

## Pthread data types :

The Pthreads API defines a number of data types, some of which are listed

Data type	Description
<i>pthread_t</i>	Thread identifier
<i>pthread_mutex_t</i>	Mutex
<i>pthread_mutexattr_t</i>	Mutex attributes object
<i>pthread_cond_t</i>	Condition variable
<i>pthread_condattr_t</i>	Condition variable attributes object
<i>pthread_key_t</i>	Key for thread-specific data
<i>pthread_once_t</i>	One-time initialization control context
<i>pthread_attr_t</i>	Thread attributes object

# Pthreads API

---

## Threads and errno :

- In the traditional UNIX API, errno is a global integer variable. However, this doesn't suffice for threaded programs.
- If a thread made a function call that returned an error in a global errno variable, then this would confuse other threads that might also be making function calls and checking errno (race condition)

# Pthreads API

---

## Return value from Pthreads functions:

- The traditional method of returning status from system calls and some library functions is to return 0 on success and –1 on error, with `errno` being set to indicate the error.
- The functions in the Pthreads API do things differently. All Pthreads functions return 0 on success or a positive value on failure. The failure value is one of the same values that can be placed in `errno` by traditional UNIX system calls

# POSIX Thread management functions

POSIX function	description
pthread_cancel	terminate another thread
pthread_create	create a thread
pthread_detach	set thread to release resources
pthread_equal	test two thread IDs for equality
pthread_exit	exit a thread without exiting process
pthread_kill	send a signal to a thread
pthread_join	wait for a thread
pthread_self	find out own thread ID

- Most POSIX thread functions return 0 if successful and a nonzero error code if unsuccessful. **They do not set errno, so the caller cannot use perror to report errors**

# PThreads API : Thread Creation

When a program is started, the resulting process consists of a single thread, called the initial or main thread.

The *pthread\_create()* function creates an additional (new) thread.

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                  void *(*start)(void *), void *arg);
```

Returns 0 on success, or a positive error number on error

The new thread commences execution by calling the function identified by start with the argument arg . The thread that calls *pthread\_create()* continues execution with the next statement that follows the call.

# Pthreads API : Thread Termination

---

The execution of a thread terminates in one of the following ways:

- The thread's start function performs a return specifying a return value for the thread.
- The thread calls *pthread\_exit()* .
- The thread is canceled using *pthread\_cancel()*
- Any of the threads calls *exit()*, or the main thread performs a *return* (in the *main()* function), which causes all threads in the process to terminate immediately.

# Pthreads API : Thread Termination

The *pthread\_exit()* function terminates the calling thread, and specifies a return value that can be obtained in another thread by calling *pthread\_join()*.

```
include <pthread.h>
void pthread_exit(void *retval);
```

↑ thread ends  
→ \_exit() → All thread ends,

Calling *pthread\_exit()* is equivalent to performing a return in the thread's start function, with the difference that *pthread\_exit()* can be called from any function that has been called by the thread's start function.

If the main thread calls *pthread\_exit()* instead of calling *exit()* or performing a return , **then the other threads continue to execute**

# Pthreads API : Thread IDs

- POSIX threads are referenced by an ID of type `pthread_t`. A thread can find out its ID by calling `pthread_self`.

```
#include <pthread.h>
pthread_t pthread_self(void);
```

→ return : 1 (not 0)

- Since `pthread_t` may be a structure, use `pthread_equal` to compare thread IDs for equality. The parameters of `pthread_equal` are the thread IDs to be compared.

```
#include <pthread.h>
pthread_t pthread_equal(pthread_t t1, pthread_t t2);
```

→ return 0 if equal

- If `t1` equals `t2`, `pthread_equal` returns a nonzero value. If the thread IDs are not equal, `pthread_equal` returns 0

# Pthreads API : Detaching and Joining

- When a thread exits, it does not release its resources unless it is a detached thread.
- ✓ The `pthread_detach` function sets a thread's internal options to specify that storage for the thread can be reclaimed when the thread exits.
- Detached threads do not report their status when they exit.
- Threads that are not detached are joinable and do not release all their resources until another thread calls `pthread_join` for them or the entire process exits. The `pthread_join` function causes the caller to wait for the specified thread to exit (similar to `waitpid` at the process level)
- To prevent memory leaks, long-running programs should eventually call either `pthread_detach` or `pthread_join` for every thread.

Do not -> use `detach`  
or `join` to prevent leak.

↳ If not detached,  
another thread must call `join` to  
prevent memory leaks.

# PThreads API : Joining

- A nondetached thread's resources are not released until another thread calls `pthread_join` with the ID of the terminating thread as the first parameter.
- **The `pthread_join` function suspends the calling thread until the target thread, specified by the first parameter, terminates.**
- The `value_ptr` parameter provides a location for a pointer to the return status that the target thread passes to `pthread_exit` or `return`. If `value_ptr` is `NULL`, the caller does not retrieve the target thread's return status

```
#include <pthread.h>
int pthread_join(pthread_t thread, void **value_ptr);
```

# Pthreads API : Joining

---

threads/simple\_thread.c

```
#include <pthread.h>
#include "tlpi_hdr.h"

static void *
threadFunc(void *arg)
{
    char *s = (char *) arg;

    printf("%s", s);

    return (void *) strlen(s);
}

int
main(int argc, char *argv[])
{
    pthread_t t1;
    void *res;
    int s;

    s = pthread_create(&t1, NULL, threadFunc, "Hello world\n");
    if (s != 0)
        errExitEN(s, "pthread_create");

    printf("Message from main()\n");
    s = pthread_join(t1, &res);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("Thread returned %ld\n", (long) res);

    exit(EXIT_SUCCESS);
}
```

One it is detached  
no join.

when it's detached,  
when exit() calls  
Everything terminates.

---

threads/simple\_thread.c

# Pthreads API : Detaching

- The `pthread_detach` function has a single parameter, `thread`, the thread ID of the thread to be detached.

```
#include <pthread.h>  
  
int pthread_detach(pthread_t thread);
```

- If successful, `pthread_detach` returns 0. If unsuccessful, `pthread_detach` returns a nonzero error code
- Once a thread has been detached, it is no longer possible to use `pthread_join()` to obtain its return status, and the thread can't be made joinable again. Detaching a thread doesn't make it immune to a call to `exit()` in another thread or a return in the main thread. In such an event, all threads in the process are immediately terminated, regardless of whether they are joinable or detached. To put things another way, `pthread_detach()` simply controls what happens after a thread terminates, not how or when it terminates.

## Pthreads API : Thread Attributes

---

- As mentioned earlier that the *pthread\_create()* *attr* argument, whose type is *pthread\_attr\_t*, can be used to specify the attributes used in the creation of a new thread.
- We'll just mention that these attributes include information such as the location and size of the thread's stack, the thread's scheduling policy and priority and whether the thread is joinable or detached.

# Thread Attributes Example code

---

*from threads/detached\_attrib.c*

```
pthread_t thr;
pthread_attr_t attr;
int s;

s = pthread_attr_init(&attr);                  /* Assigns default values */
if (s != 0)
    errExitEN(s, "pthread_attr_init");

s = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
if (s != 0)
    errExitEN(s, "pthread_attr_setdetachstate");

s = pthread_create(&thr, &attr, threadFunc, (void *) 1);
if (s != 0)
    errExitEN(s, "pthread_create");

s = pthread_attr_destroy();                  /* No longer needed */
if (s != 0)
    errExitEN(s, "pthread_attr_destroy");
```

---

*from threads/detached\_attrib.c*

## Thread Attributes Example code

---

The code shown creates a new thread that is made detached at the time of thread creation.

This code first initializes a thread attributes structure with default values, sets the attribute required to create a detached thread, and then creates a new thread using the thread attributes structure.

Once the thread has been created, the attributes object is no longer needed, and so is destroyed.

# Threads Versus Processes

---

This lecture we briefly considered some of the factors that might influence our choice of whether to implement an application as a group of threads or as a group of processes.

We begin by considering the advantages of a multithreaded approach :

- Sharing data between threads is easy. By contrast, sharing data between processes requires more work .
- Thread creation is faster than process creation; context-switch time may be lower for threads than for processes

# Threads Versus Processes

---

Using threads can have some disadvantages compared to using processes

- When programming with threads, we need to ensure that the functions we call are thread-safe or are called in a thread-safe manner. Multiprocess applications don't need to be concerned with this.
- A bug in one thread (e.g., modifying memory via an incorrect pointer) can damage all of the threads in the process, since they share the same address space and other attributes. By contrast, processes are more isolated from one another.

# Threads Versus Processes

---

- Each thread is competing for use of the finite virtual address space of the host process. In particular, each thread's stack and thread-specific data (or thread local storage) consumes a part of the process virtual address space, which is consequently unavailable for other threads. Although the available virtual address space is large, this factor may be a significant limitation for processes employing large numbers of threads or threads that require large amounts of memory. By contrast, separate processes can each employ the full range of available virtual memory .

# Threads Versus Processes

The following are some other points that may influence our choice of threads versus processes:

- Dealing with signals in a multithreaded application requires careful design. (As a general principle, it is usually desirable to avoid the use of signals in multithreaded programs.) *Only when absolutely necessary.*
- In a multithreaded application, all threads must be running the same program. In a multiprocess application, different processes can run different programs.
- Aside from data, threads also share certain other information (e.g., file descriptors, signal dispositions, current working directory, and user and group IDs). This may be an advantage or a disadvantage, depending on the application.

# Summary

---

- In a multithreaded process, multiple threads are concurrently executing the same program. All of the threads share the same global and heap variables, but each thread has a private stack for local variables. The threads in a process also share a number of other attributes, including process ID, open file descriptors, signal dispositions, current working directory, and resource limits.
- The key difference between threads and processes is the easier sharing of information that threads provide, and this is the main reason that some application designs map better onto a multithread design than onto a multiprocess design. Threads can also provide fast performance for some operations, but this factor is usually secondary in influencing the choice of threads versus processes.

# Summary

---

- Threads are created using *pthread\_create()*. Each thread can then independently terminate using *pthread\_exit()* (If any thread calls exit(), then all threads immediately terminate).
- Unless a thread has been marked as detached, it must be joined by another thread using *pthread\_join()*, which returns the termination status of the joined thread. *Detach or join*

# BIL 344 System Programming

Week 10

---

## Synchronizing with threads

- Mutexes
- Condition variables
- Classic problems and their solutions in terms of threads

If there is a common variable  
that is shared by multiple threads (an integer)  
when it comes to accessing you need  
precaution.

Semaphores  
→ mutexes → condition variables

# Thread synchronization

Threads are a beautiful concept, they are:

- “**lighter**” than processes in terms of memory use
- **faster** in terms of creation and context switching
- **easier** to establish inter-thread communication as they share the process memory.

However, all this comes at a cost, and that cost is solving the associated synchronization problems. You have already seen (counting) semaphores. You can use Posix/IPC semaphores with threads as well to solve **some** of your synchronization problems.

Posix threads offer us two more synchronization tools that enable us to solve (theoretically) **any** synchronization problem: **mutexes** and **condition variables**.

# Thread synchronization

When faced with a critical section (i.e. a block of code executed in parallel, posing a synchronization risk), there is **no way** of avoiding an eventual rescheduling by the kernel.

All we can (and should) do, is make sure no other thread/process accesses the critical section while there is already a thread/process inside it.

We achieved this with semaphores by surrounding critical sections with wait and post calls:

`wait(s)`

`//critical section, common variable, common data structure, etc`

`post(s)`

either locked (or owned) by a thread  
or unlocked

## Thread synchronization

Threads employ **mutexes (mutual exclusion)** for the same purpose.

A mutex  $m$  can be in either of two states: **locked (owned)** or **unlocked (not owned)**. It can change states through `lock()` and `unlock()` calls.

Once a mutex  $m$  is locked/owned by a thread  $t_1$ , all other threads trying to lock  $m$  will block, **until  $t_1$  unlocks  $m$** .

Thread  $t_1$

`lock (m)`

`push (v)`

`unlock (m)`

Thread  $t_2$

`lock (m)`

`v = pop ()`

`// critical section`

`unlock (m)`

# Mutex synchronization

Creating a mutex (**dynamically**, with eventually custom attributes)

```
pthread_mutex_t mutex;  
pthread_mutex_init (&mutex, NULL); // NULL for default attrs.
```

```
#include <pthread.h>  
int pthread_mutex_init(pthread_mutex_t * m ,  
                      const pthread_mutexattr_t * attr);
```

Returns 0 on success, or a positive error number on error

or **statically** (with default attributes; convenient for global variables)

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

# Thread synchronization

Attention: calling `pthread_mutex_init()` on an already initialized mutex leads to undefined behavior!

Don't call twice for same mutex.

Once you no longer need to use a mutex, call `pthread_mutex_destroy()`

if statically created, no need

```
#include <pthread.h>
```

```
int pthread_mutex_destroy(pthread_mutex_t * mutex);
```

Returns 0 on success, or a positive error number on error

Make sure it is unlocked before destroying.

It is not necessary to call `pthread_mutex_destroy()` on a mutex that was statically initialized using `PTHREAD_MUTEX_INITIALIZER`

Never make copies of mutexes.  
If you need additional ones, create a new one.

Make sure it's not locked when you call it!

And don't make mutex **copies**!

# Thread synchronization

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t * m );
int pthread_mutex_unlock(pthread_mutex_t * m );
```

Both return 0 on success, or a positive error number on error

Assuming that `m` is a mutex, and `t1, t2` are threads using it.

What happens when ->	t1 calls lock	t1 calls unlock	t2 calls lock	t2 calls unlock
<b>While m is owned by t1</b>	depends on the type of m	m is unlocked	t2 is blocked <small>Other cannot block.</small>	undefined/error depending on type
<b>while m is not owned</b>	t1 locks m	undefined/error depending on type  <small>↳ Don't call unlock, if doesn't belong to you.</small>	t2 locks m	undefined/error depending on type

# Thread synchronization

Rules (the exact behavior depends on the type of the mutex):

- A single thread may not lock the same mutex twice.
- A thread may not unlock a mutex that it doesn't currently own (i.e., that it did not lock).
- A thread may not unlock a mutex that is not currently locked.

**Normal/Fast:** default, no checks, deadlocks in case of double lock (waits for itself)

**Recursive:** keeps count of locks, and only unlocks when the count is zero (e.g. in case your critical section is located inside a recursive function; credits to Yusuf Karaarslan)

**Errorcheck:** slower, but returns errors on all 3 scenarios

mutex attribute

# Thread synchronization

Example on how to set the mutex type:

```
pthread_mutex_t mtx;
pthread_mutexattr_t mtxAttr;
int s, type;

s = pthread_mutexattr_init(&mtxAttr);
if (s != 0)
    errExitEN(s, "pthread_mutexattr_init");

s = pthread_mutexattr_settype(&mtxAttr, PTHREAD_MUTEX_ERRORCHECK);
if (s != 0)
    errExitEN(s, "pthread_mutexattr_settype");

s = pthread_mutex_init(mtx, &mtxAttr);
if (s != 0)
    errExitEN(s, "pthread_mutex_init");

s = pthread_mutexattr_destroy(&mtxAttr);          /* No longer needed */
if (s != 0)
    errExitEN(s, "pthread_mutexattr_destroy");
```

# Thread synchronization

## Common errors with mutexes

- Different orders of locks across threads

Thread A

```
1. pthread_mutex_lock(&mutex1);  
2. pthread_mutex_lock(&mutex2);  
blocks
```

Thread B

```
1. pthread_mutex_lock(&mutex2);  
2. pthread_mutex_lock(&mutex1);  
blocks
```

- Relocking an already locked mutex
- Trying to unlock another thread's mutex

*bad design, try to avoid.*

Tip: you can check safely whether a mutex is locked using `pthread_mutex_trylock()`; it returns zero if the lock is acquired or an immediate error otherwise.

# Thread synchronization

Important: is a mutex **identical** to a binary semaphore?

No. Because conversely to semaphores, mutexes revolve around the core concept of ownership. Semaphores don't have an owner, they can be increased through “`post()`” by any thread/process. A mutex can be unlocked/released **only** by its owner.

**Mutexes** are excellent for controlling access to critical sections.

**Semaphores** are great for implementing shared counters.

In practice however we encounter far more complex situations; for those we have **condition variables**.

↳ Theoretically, we can solve any kind of sync problem with cv.

# Thread synchronization

Examples of real-world synchronization problems:

- wait for  $t$  to acquire the value of 17
- wait for  $x+y > 13$
- wait for some buffer to fill up to at least 85%... etc.

Common <sup>Ortak</sup> denominator of all problems: **wait for a certain condition or logical predicate to be satisfied.**

A condition variable  $c$  possesses three main methods:

- 1)  $cwait(c,?)$ : blocks the thread (?: is a hidden parameter, more on this soon)
- 2)  $signal(c)$ : awakens a thread blocked on  $c$  because it called  $cwait(c,?)$
- 3)  $broadcast(c)$ : awakens all threads blocked on  $c$  because they called  $cwait(c,?)$

In Java  
wait  
notify  
notifyAll

# Thread synchronization

A condition variable **knows nothing** about the condition that it's waiting for.

## Example

T1, T2 and T3: threads, c: condition variable,  $x=y=0$

T3 must not advance unless  $x+y > 13$

*This is wrong (Need to protect with a mutex)*

```
T1
  x += 5;
  signal(c);

T2
  y += 7;
  signal(c);

T3
  while(! (x + y > 13))
    cwait(c, ?);
```

With every call of `signal()`, T3 checks whether the condition is satisfied; if yes, it continues, otherwise it goes back to sleep.

# Thread synchronization

T3: test condition	FALSE	cwait and sleep again	x=0, y=0
T1: x+=5	signal	T3 awake and ready	x=5, y=0
...			
T3: test condition	FALSE	cwait and sleep again	x=5, y=0
T2: y+=7	signal	T3 awake and ready	x=5, y=7
T3: test condition	FALSE	cwait and sleep again	x=5, y=7
T1: x+=5	signal	T3 awake and ready	x=10, y=7
T3: test condition	TRUE	exit loop	x=10, y=7

The while clause could be about any logical predicate, that's what enables condition variables to be used with any problem.

# Thread synchronization

---

## Issues

The variables involved in the condition are shared between T1, T2 and T3. Let's say the condition is satisfied, and T3 exits the while loop. How do we know the condition is still satisfied at that point? We don't!

All we know is that the condition was satisfied certainly for at least a brief moment in time. Some thread could have possibly changed them back to unwanted values.

Conclusion: condition variables are **never used alone**, but **always with a mutex**.

- 1) Signal and broadcast calls must be made always under an exclusive lock.**
- 2) The condition must be tested under an exclusive lock.**

# Thread synchronization

Conclusion:

- 1) Signal and broadcast calls must be made always under an exclusive lock.
- 2) The condition must be tested under an exclusive lock.

T1	T2	T3
lock (m)	lock (m)	lock (m)
$x += 5;$	$y += 7;$	while (! ( $x + y > 13$ )) {
signal (c);	signal (c);	unlock (m) <span style="color: orange;">→ unlock so others can use m</span>
unlock (m)	unlock (m)	cwait (c, ?) <span style="color: orange;">→ like sleep</span>
		lock (m) <span style="color: orange;">→ Eager T1 vs T2</span>
		}
		...
		unlock (m) <span style="color: orange;">badly synchronized</span>

Are we done?

# Thread synchronization

**No!** What if T1 or T2 call signal while T3 is between unlock and cwait?  
The signal will be lost, and T3 will wait for the next signal which might never arrive.

T1	T2	T3
lock (m)	lock (m)	lock (m)
$x += 5;$	$y += 7;$	<code>while (! (x + y &gt; 13)) {</code>
<code>signal (c);</code>	<code>signal (c);</code>	<b><code>unlock (m)</code></b>
unlock (m)	unlock (m)	<b><code>cwait (c, ?);</code></b>
		<code>lock (m)</code>
		}
		...
		unlock (m)

# Thread synchronization

The truth is you cannot solve this at user level. That is why cwait receives not one but two parameters: **the condition variable to operate on and a mutex**.

And thus we talk about a monitor = a lock and zero or more condition variables

cwait (c, m)

This way when a signal arrives (due to `signal()` or `broadcast()`) the mutex is first locked and then cwait returns (**atomically**).

AND

When calling cwait, the mutex is first unlocked (**atomically**).

# Thread synchronization

In other words, all three statements are combined into one.

T3

lock (m)

while (! (x + y > 13)) {

**unlock (m)**

**cwait (c, ?);**

**lock (m)**

}

...

unlock (m)

T3

lock (m)

while (! (x + y > 13)) {

**cwait (c, m)**

}

... // safe to proceed on x, y

unlock (m)

*pthread cond-wait*  
↳ Wait for condition  
variable to be signaled  
or broadcast.

*✓ Atomically  
(no context switch  
gerekçeli)*

Sinyal'ıki threadleri uyandır ve veriyi global 'den alırlar.

Toplunda n tane thread olacak ve hepsi bekleyeceler.

# Thread synchronization

Example: the **bounded producer-consumer** problem.

int count=0: number of products, N upper limit, mutex m for storage  
Condition variables empty and full.

Producer

```
for(;;)
    lock(m)
    while(count == N)
        cwait(empty, m) Wait for
                        consumer broadcast (signal
                        empty conditional variable
                        until empty)
    produce_item()
    count++
    broadcast(full)
    unlock(m)
```

Consumer

```
for(;;)
    lock(m)
    while(count == 0)
        cwait(full, m)
    consume_item()
    count--
    broadcast(empty)
    unlock(m)
```

# Thread synchronization

Example: synchronization barrier with N threads.

Condition variable c, mutex m, arrived = 0

Checks on  $\Sigma$  the last thread

that reach to rendezvous point.

If not the last  
go to sleep

If the last,  
it will be  
arrived = N  
then it will  
broadcast.

```
lock (m)
++arrived
if(arrived < N)      // if this thread is not last
    cwait (c, m)      // then wait for others
else
    broadcast (c)    // i'm last, awaken the other N-1
unlock (m)
```

Always should be inside loop.

# Thread synchronization

---

## Example: **readers-writers**

### Reader

```
wait until no writers
access database
check out -- wake up waiting writer
```

### Writer

```
wait until no readers or writers
access database
check out -- wake up waiting readers or writer
```

# Thread synchronization

State variables

number of active readers AR = 0

→ We can have multiple active reader  
but only one active writer.

number of active writers AW = 0

number of waiting readers WR = 0

number of waiting writers WW = 0

Condition variable okToRead

Condition variable okToWrite

Lock m

# Thread synchronization

```
Reader ()  
    lock (m);  
    while ((AW + WW) > 0) { // if any writers, wait  
        WR++; // waiting reader  
        cwait (okToRead, m);  
        WR--;  
    }  
    AR++; // active reader  
    unlock (m);  
    Access DB  
    lock (m);  
    AR--;  
    if (AR == 0 && WW > 0) → Check if they are last  
        signal (okToWrite, m); // active reader to leave  
        the database to leave  
        if there is a writer,  
unlock (m); → Not broadcast,  
because one writer should enter. (Syallerdeki signal ile aynı değil.)
```

# Thread synchronization

```
Writer()
{
    lock(m);
    while ((AW + AR) > 0) { // if any readers or writers, wait
        WW++; // waiting writer
        cwait(okToWrite, m);
        WW--;
    }
    AW++; // active writer
    unlock(m);
    Access DB
    lock(m);
    AW--;
    if (WW > 0) // give priority to other writers
        signal(okToWrite, m);
    else if (WR > 0) // if there are no writers that waits
        broadcast(okToRead, m); // then signal to readers. (All readers)
    unlock(m);
}
```

*because broadcast*

# Thread synchronization

The dining philosophers: 5 condition variables, one for every fork (resource). The status array stores every fork's status: free or inUse

one for each of the resources/forks.

```
lock (m)
while (status[i]==inUse || status[(i+1) % 5]==inUse)
    cwait (c[i], m); → Sağında veya solundaki
                      forklardan biri bile müsait
                      değilse bekle.
status[i]=status[(i+1) % 5]=inUse → First, set fork in use
                                         then unlock so others
                                         cannot touch it.
unlock (m)
//eat
lock (m)
status[i]=status[(i+1) % 5]=free Ate so set free
signal (c[i], m) → Maden sağındaki /Solundaki gatalları
                           bıraktım, sağındaki /Solundaki filozofları
                           uyandıracım o zaman.
signal (c[(i+1) % 5], m)
unlock (m)
```

# Thread synchronization

**Cigarette smokers:** c condition variable, m mutex, s=0 semaphore

Agent

```
for(;;)  
    get_two_random_ingredients()  
    lock(m)  
    deliver_ingredients()  
    broadcast(c)      // tell all that the ingredients are here  
    unlock(m)         // they can take them  
    wait(s)           // wait for the cigarette to be ready
```

→ Signal all, but only one of them will be successful.

# Thread synchronization

---

## Smoker

```
for(;;)
    lock(m)
    while(!ingredient_enough())      // while missing ingredients
        cwait(c,m)                  // sleep
    obtain_ingredients()
    unlock(m)                      // ingredients taken, no need for lock
    prepare_cigarette()
    post(s)                        // cigarette ready
```

# Thread synchronization

Similarly to mutexes, condition variables can be allocated statically or dynamically.

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

or

```
#include <pthread.h>  
int pthread_cond_init(pthread_cond_t * cond ,  
                      const pthread_condattr_t * attr );
```

Returns 0 on success, or a positive error number on error

- ✓ Use `NULL` in place of `attr` for default parameters.
- ✓ Do not re-initialize (with `init`) an already initialized cond. var. → undefined!
- ✓ Copying a cond. variable → undefined behavior. Work on the originals!

# Thread synchronization

```
#include <pthread.h>

int pthread_cond_signal(pthread_cond_t * cond ) ;

int pthread_cond_broadcast(pthread_cond_t * cond ) ;

int pthread_cond_wait(pthread_cond_t * cond ,
                      pthread_mutex_t * mutex ) ;
```

All return 0 on success, or a positive error number on error

**The thread to be awoken by `pthread_cond_signal()` is unpredictable.**

*↳ we cannot know which thread is going to wake up.*

Don't use. Bad design.

## Thread synchronization

```
#include <pthread.h>

int pthread_cond_timedwait (pthread_cond_t * cond ,
                           pthread_mutex_t * mutex ,
                           const struct timespec * abstime );
```

Returns 0 on success, or a positive error number on error

Same as `pthread_cond_wait`, but it waits at most for `abstime`, after which (it doesn't wake from sleep) and it returns `ETIMEDOUT`

# Thread synchronization

When an automatically or dynamically allocated condition variable is no longer required, then it should be destroyed using `pthread_cond_destroy()`.

```
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t * cond );
```

Returns 0 on success, or a positive error number on error

It is not necessary to call `pthread_cond_destroy()` on a condition variable that was statically initialized using `PTHREAD_COND_INITIALIZER`.

Make sure there are no threads waiting for the condition variable before destroying it. You can use it again if you want to; just make sure you call `pthread_cond_init()` first.

# BIL 344 System Programming

Week 11

## Socket Programming

- Sockets : An Introduction
- Unix Domains
- Fundamentals of TCP/IP Networks

En az 2 sunucu var demek,  
POSIX uyumlu  
2 bilgisayar arası

Şimdi ilk PC'yi deskefliyor.

2 PC iletişim kuracaksı P2P veya 2 node kuralları.

Server tarafında Socket oluştur. (adres, portunu oluştur.)

Açığımız anda bize file descriptor veriyor.

Sadece PC'ler arası değil,  
aynı PC'de pipe gibi kullanılabili

Aynı PC üzerinde socket kuruyorsa,  
2 desya olarak ifade ediliyor.  
(Socket'i kurarken buna belirtiyorsa ➔ AF\_UNIX, dizer 194)

# Overview

---

In a typical client-server scenario, applications communicate using sockets as follows:

- Each application creates a socket. A socket is the “apparatus” that allows communication, and both applications require one.
- The server binds its socket to a well-known address (name) so that clients can locate it.

A socket is created using the *socket()* system call, which returns a file descriptor used to refer to the socket in subsequent system calls:

*fd = socket(domain, type, protocol);*

# Communication Domain

---

Sockets exist in a communication domain, which determines:

- The method of identifying a socket (i.e., the format of a socket “address”);
- The range of communication (i.e., either between applications on the same host or between applications on different hosts connected via a network).

# Sockets

As of today modern operating systems support at least the following domains:

- The *UNIX* (AF\_UNIX) domain allows communication between applications on the same host.
- The *IPv4* ( AF\_INET) domain allows communication between applications running on hosts connected via an Internet Protocol version 4 (IPv4) network.
- The *IPv6* ( AF\_INET6) domain allows communication between applications running on hosts connected via an Internet Protocol version 6 (IPv6) network.

\*Although IPv6 is designed as the successor to IPv4, the latter protocol is still widely used.

# Socket Types

- Every socket implementation provides at least two types of sockets : **stream** and **datagram**.
- These socket types are supported in both the UNIX and the Internet domains.

Property	Socket type	
	Stream	Datagram
Reliable delivery?	Y	N
Message boundaries preserved?	N	Y
Connection-oriented?	Y	N

Mesela C1-b sibh secip nıracısan,

Wiþer PC'ler bunu yapır. (Broadcast yapırız sun PC)

Stream'de; eger bilgi ulasmadıysa bir datha paket yolluyor. (Message passing)

Datagram'da datha iyi, deðan ediyorsan. (Paketi var mı yok mu biliþi var.)

Güvenlik

Herhangi bir sırada yollandıysa  
o Stream'a; basarına kadar  
yollanır ian garanti veriyor.

A, B yolla

A, A, B gidelbilir.  
B, A gidelbilir.

(Stream'de  
bu nüfakar  
olmaz.)

İyi yani.

Zarf gibi gidiyor.

(bası, sunu hakkında bilmiyoruz.)

→ Stream altýný pipe gibi

→ file gibi no boundaries

TCP(stream), UDP(datagram)

# Stream Sockets

---

Stream sockets ( `SOCK_STREAM` ) provide a *reliable, bidirectional, byte-stream* communication channel. By the terms in this description, we mean the following:

- **Reliable** means that we are guaranteed that either the transmitted data will arrive intact at the receiving application, exactly as it was transmitted by the sender , or that we'll receive notification of a probable failure in transmission.
- Bidirectional means that data may be transmitted in either direction between two sockets.
- Byte-stream means that, as with pipes, there is no concept of message boundaries

# Stream Sockets

---

- A stream socket is similar to using a pair of pipes to allow bidirectional communication between two applications, with the difference that (Internet domain) sockets permit communication over a network.
- Stream sockets operate in connected pairs. For this reason, stream sockets are described as connection-oriented. The term **peer socket** refers to the socket at the other end of a connection; peer address denotes the address of that socket; and peer application denotes the application utilizing the peer socket. Sometimes, the term **remote** is used synonymously with **peer**.
- Analogously, sometimes the term **local** is used to refer to the application, socket, or address for this end of the connection. A stream socket can be connected to only one peer

# Datagram Sockets

- Datagram sockets (SOCK\_DGRAM) allow data to be exchanged in the form of messages called datagrams. With datagram sockets, message **boundaries are preserved**, but data transmission is **not reliable**. Messages may arrive out of order, be duplicated, or not arrive at all.
- Unlike a stream socket, a datagram socket doesn't need to be connected to another socket in order to be used.
- In the Internet domain, datagram sockets employ the User Datagram Protocol (UDP), and stream sockets (usually) employ the Transmission Control Protocol (TCP).
- Instead of using the terms Internet domain datagram socket and Internet domain stream socket, we'll often just use the terms **UDP socket** and **TCP socket**, respectively.

*datagram sockets*

*stream sockets*

# Socket System Calls

The key socket system calls are the following:

- The ***socket()*** system call creates a new socket.
- The ***bind()*** system call binds a socket to an address. Usually, a server employs this call to bind its socket to a well-known address so that clients can locate the socket.
- The ***listen()*** system call allows a stream socket to accept incoming connections from other sockets.
- The ***accept()*** system call accepts a connection from a peer application on a listening stream socket, and optionally returns the address of the peer socket.  
*Bekleyen sığır madura okşırır  
Accept doesn't return.*
- The ***connect()*** system call establishes a connection with another socket.  
*İlhan'ın connect'i bekliyor.*

# Socket I/O

- Socket I/O can be performed using the conventional *read()* and *write()* system calls, or using a range of socket-specific system calls (e.g., *send()*, *recv()*, *sendto()*, and *recvfrom()*).
- By default, these system calls block if the I/O operation can't be completed immediately.
- Nonblocking I/O is also possible, by using the *fcntl()* **F\_SETFL** operation to enable the **O\_NONBLOCK** open file status flag. → (Tıkınmadık isteniyorsa hata ile geri dönmeyi sağlayır.)

Bind → Listen + Accept → Yeni bir fd veriyorsa  
(ilk fd özgür.)

# Creating A Socket

- The *socket()* system call creates a new socket

```
#include <sys/socket.h>
```

```
int socket(int domain, int type, int protocol);
```

bizim derseimde  
her紫eim 0 (header'lam ularna vs.)

Returns file descriptor on success, or -1 on error

- The domain argument specifies the communication domain for the socket. The type argument specifies the socket type. This argument is usually specified as either `SOCK_STREAM`, to create a stream socket, or `SOCK_DGRAM`, to create a datagram socket.
- The protocol argument is always specified as 0 for the socket types specified in this course. Nonzero protocol values are used with some socket types that we don't describe. For example, protocol is specified as `IPPROTO_RAW` for raw sockets (`SOCK_RAW`).
- On success, *socket()* returns a file descriptor used to refer to the newly created socket in later system calls.

# Binding a Socket to an Address

The *bind()* system call binds a socket to an address.

```
#include <sys/socket.h>
```

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

Returns 0 on success, or -1 on error

ip v4 → 4 byte

The *sockfd* argument is a file descriptor obtained from a previous call to *socket()*. The *addr* argument is a pointer to a structure specifying the address to which this socket is to be bound. The type of structure passed in this argument depends on the socket domain. The *addrlen* argument specifies the size of the address structure.

# Generic Socket Address Structures

- The `addr` and `addrlen` arguments to `bind()` require some further explanation.
- For each socket domain, a different structure type is defined to store a socket address. However, because system calls such as `bind()` are generic to all socket domains, they must be able to accept address structures of any type. In order to permit this, the sockets API defines a generic address structure, `struct sockaddr`.  
*Polymorphism  
Supertype ⇒ Alt fñeri vor.*
- The only purpose for this type is to cast the various domain-specific address structures to a single type for use as arguments in the socket system calls.

## struct sockaddr

The sockaddr structure is typically defined as follows:

```
struct sockaddr {  
    sa_family_t sa_family;          /* Address family (AF_* constant) */  
    char        sa_data[14];         /* Socket address (size varies  
                                    according to socket domain) */  
};
```

- This structure serves as a template for all of the domain-specific address structures.
- Each of these address structures begins with a family field corresponding to the *sa\_family* field of the sockaddr structure.
- The value in the family field is sufficient to determine the size and format of the address stored in the remainder of the structure.

# Stream Sockets

---

The operation of stream sockets can be explained by analogy with the telephone system:

- The *socket()* system call, which creates a socket, is the equivalent of installing a telephone. In order for two applications to communicate, each of them must create a socket.
- Communication via a stream socket is analogous to a telephone call. One application must connect its socket to another application's socket before communication can take place.
- Once a connection has been established, data can be transmitted in both directions between the applications until one of them closes the connection using *close()*. Communication is performed using the conventional *read()* and *write()* system calls or via a number of socket specific system calls (such as *send()* and *recv()*) that provide additional functionality.

# Stream Sockets

Two sockets are connected as follows:

- One application calls *bind()* in order to bind the socket to a well-known address, and then calls *listen()* to notify the kernel of its willingness to accept incoming connections.
- The other application establishes the connection by calling *connect()*, specifying the address of the socket to which the connection is to be made.
- The application that called *listen()* then accepts the connection using *accept()*. If the *accept()* is performed before the peer application calls *connect()*, then the *accept()* blocks

*Sınıfı doldurmadığında (accept'te)*

# Stream Sockets

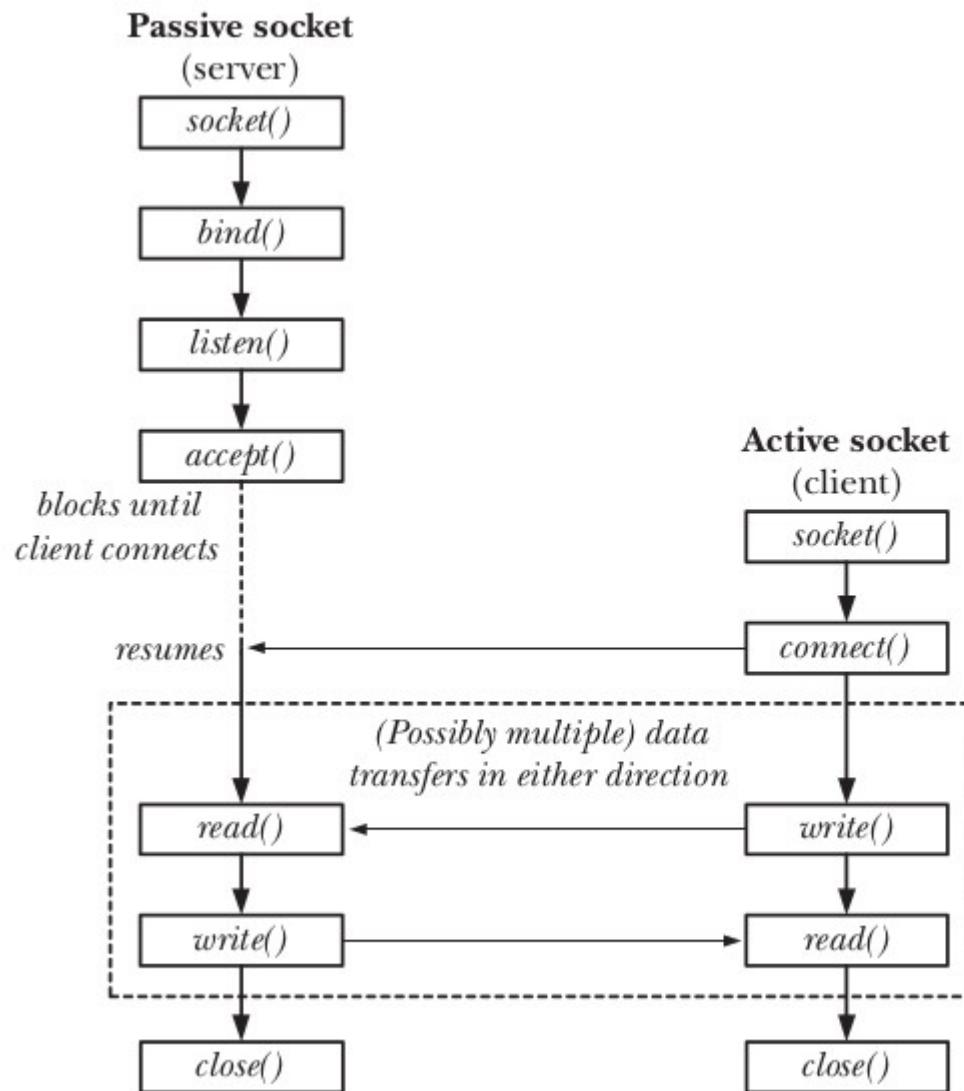
Stream sockets are often distinguished as being either **active** or **passive**:

- By default, a socket that has been created using *socket()* is **active**. An active socket can be used in a *connect()* call to establish a connection to a passive socket. This is referred to as performing an active open.
- A passive socket (also called a listening socket) is one that has been marked to allow incoming connections by calling *listen()*. Accepting an incoming connection is referred to as performing a passive open

Tüm işleri yaparak başka bağlantı gelirse bakılır. (Thread oluşturma da yarar.)  
Farklı kullanıson çok yarar. Veya thread oluşturma. Bu yarar threadleri en en başka oluştur.

Sistem programlama optimizeye yapıyor.  
Accept te aynı bağlantıda socket'i kullanmayı dikkat et.

# Overview of systems calls : Stream Sockets



# Listening for incoming Connections

The *listen()* system call marks the stream socket referred to by the file descriptor *sockfd* as passive. The socket will subsequently be used to accept connections from other (active) sockets.

```
#include <sys/socket.h>  
  
int listen(int sockfd, int backlog);
```

Eğer accept() kabul ederken o sırada başka  
bağlantı gelirse backlog ontarın sayısını laffediyorsun.  
(örneğin 5; ana backlog'a giremeye sizce olabilir.)  
Ancak en kısa sürede handle etmek

Returns 0 on success, or -1 on error

We can't apply *listen()* to a connected socket—that is, a socket on which a *connect()* has been successfully performed or a socket returned by a call to *accept()*.

The kernel must record some information about each pending connection request so that a subsequent *accept()* can be processed. The backlog argument allows us to limit the number of such pending connections

# Accepting a Connection

The *accept()* system call accepts an incoming connection on the listening stream socket referred to by the file descriptor *sockfd*. If there are no pending connections when *accept()* is called, the call blocks until a connection request arrives.

```
#include <sys/socket.h>
```

```
int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);
```

→ adres varlığı

Returns file descriptor on success, or -1 on error

bağlantı alındı  
yeni socket dönüyor.

Aynı değil

Note that *accept()* creates a new socket, and it is this new socket that is connected to the peer socket that performed the *connect()*. A file descriptor for the connected socket is returned as the function result of the *accept()* call. The listening socket (*sockfd*) remains open, and can be used to accept further connections.

# Connecting to a Peer Socket

The `connect()` system call connects the active socket referred to by the file descriptor `sockfd` to the listening socket whose address is specified by `addr` and `addrlen`.

```
#include <sys/socket.h>

int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

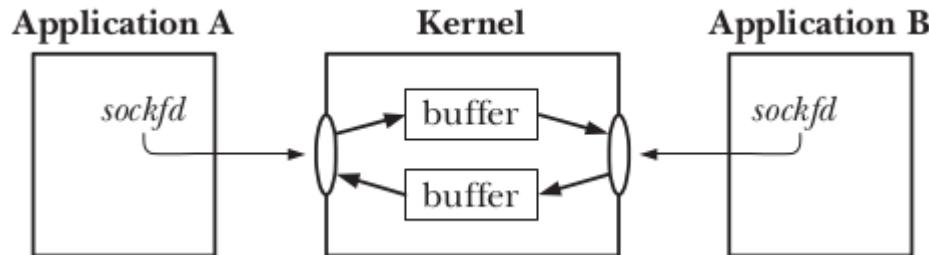
Returns 0 on success, or -1 on error

Yeni socket oluşturur. Bir bağlantıya (Socket) bağlanır. (Socket, bir socket oluşturmak için bir dala bir dala) (bir dala bir dala)

If `connect()` fails and we wish to reattempt the connection, the portable method of doing so is to close the socket, create a new socket, and reattempt the connection with the new socket.

# I/O on Stream Sockets

A pair of connected stream sockets provides a bidirectional communication channel between the two endpoints



The semantics of I/O on connected stream sockets are similar to those for pipes: to perform I/O use the `read()` and `write()` system calls. Since sockets are bidirectional both calls may be used on each end of the connection.

# Terminating a Connection

is `bitinice close()` ile kapat.

- The usual way of terminating a stream socket connection is to call **close()**. If multiple file descriptors refer to the same socket, then the connection is terminated when all of the descriptors are closed.
- Suppose that, after we close a connection, the peer application crashes or otherwise fails to read or correctly process the data that we previously sent to it. In this case, we have no way of knowing that an error occurred. If we need to ensure that the data was successfully read and processed, then we must build some type of acknowledgement protocol into our application. This normally consists of an explicit acknowledgement message passed back to us from the peer.  
yalnızca sonna hata süb��usa bawu bireyiz.  
ackidegi ḡiverme. lerdin bir acknowledgement l̄ur. Bitinice bildir.

# Datagram Sockets

→ Her zonf birbirinden bağımsız.  
Yolculuk trafi adres/kime gidecek belli olmaz.  
→ Porta bittiği gibi dönerini belli etti.  
Oluşan herage yolculuklarını belli etti.

The operation of datagram sockets can be explained by analogy with the postal system:

- The *socket()* system call is the equivalent of setting up a mailbox. Each application that wants to send or receive datagrams creates a datagram socket using *socket()*.
- In order to allow another application to send it datagrams (letters), an application uses *bind()* to bind its socket to a well-known address. Typically, a server binds its socket to a well-known address, and a client initiates communication by sending a datagram to that address

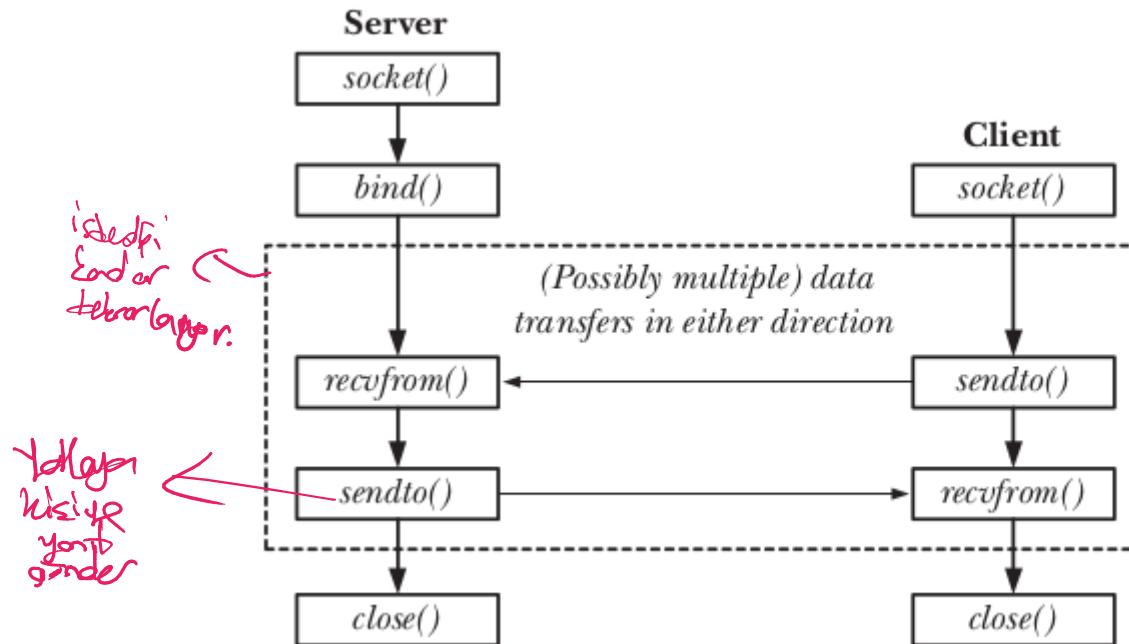
→ And to (içerisi)  
→ receive from

Yol (gödilimin gideceğinin  
ayrıca gideceğinin  
aynı sajda gideceğinin  
garantisi yok)

## Datagram Sockets

- To send a datagram, an application calls *sendto()*, which takes as one of its arguments the address of the socket to which the datagram is to be sent.
- In order to receive a datagram, an application calls *recvfrom()*, which may block if no datagram has yet arrived. Because *recvfrom()* allows us to obtain the address of the sender, we can send a reply if desired.
- When the socket is no longer needed, the application closes it using *close()*.
- There is no guarantee that they will arrive in the order they were sent, or even arrive at all. Since the underlying networking protocols may sometimes retransmit a data packet, the same datagram could arrive more than once.

# Datagram Sockets



Overview of system calls used with datagram sockets

# Exchanging Datagrams

The *recvfrom()* and *sendto()* system calls receive and send datagrams on a datagram socket.

```
#include <sys/socket.h>
```

↑ *bytes received*:  
buffer

```
ssize_t recvfrom(int sockfd, void *buffer, size_t length, int flags,  
                 struct sockaddr *src_addr, socklen_t *addrlen);
```

↳ *returning address buffer*

Returns number of bytes received, 0 on EOF, or -1 on error

```
ssize_t sendto(int sockfd, const void *buffer, size_t length, int flags,  
               const struct sockaddr *dest_addr, socklen_t addrlen);
```

Returns number of bytes sent, or -1 on error

The return value and the first three arguments to these system calls are the same as for *read()* and *write()*.

# Using `connect()` with Datagram Sockets

Datagram ile `connect()` kullanımı arounda deñirin.

`connect()` kullanımı: Yollayan kişi needen yolladı, bunu buna yazır. *ör. lokalis*

- Even though datagram sockets are connectionless, the `connect()` system call serves a purpose when applied to datagram sockets.
- Calling `connect()` on a datagram socket causes the kernel to record a particular address as this socket's peer. The term connected datagram socket is applied to such a socket. The term unconnected datagram socket is applied to a datagram socket on which `connect()` has not been called

*(Tek deñaya məsəs adres keşfədiyir.  
Gəl gəzəti yox.)*

Məca: `connect()` datagram ile kəllənməyi öneriyorum.

bir socket'la 100 kişiye yarınca yot.

Sağlamlara çok açık.  
Bilgi isten tamamında olmamalı.  
Ping'in amacı konuda belli/bizi ver mi gösterme.

## Using `connect()` with Datagram Sockets

After a datagram socket has been connected:

- Datagrams can be sent through the socket using `write()` (or `send()`) and are automatically sent to the same peer socket. As with `sendto()`, each `write()` call results in a separate datagram.
- Only datagrams sent by the peer socket may be read on the socket.
- The effect of `connect()` is asymmetric for datagram sockets. The above statements apply only to the socket on which `connect()` has been called, not to the remote socket to which it is connected (unless the peer application also calls `connect()` on its socket).

## Dealing with endian-ness across systems

Sending/receiving data across systems ignoring endian-ness is a recipe for disaster. Example:

e.g. an Intel (Little-endian) system sending a word to a Motorola system (Big-endian).

You could of course solve this by first sending some scouting messages to figure out the endian-ness of the target system, but there is a more standard approach.

## Dealing with endian-ness across systems

Before sending anything you always convert it to “network order” (big-endian).

(Olumadır once formasyondan geçir. kendi tarafın  
makinelerindeki benim makinemde yine söyle ayıra.)

And after receiving something, you first convert it into “host order” (whatever the endian-ness of the local host is..) and then use it.

16bit: ntohs and htons → host to network short  
32bit: ntohl and htonl → host to network long

e.g. the “`uint16_t htons(uint16_t n)`” function converts the unsigned short integer `n` from host byte order to network byte order

# Summary

---

- A typical **stream socket server** creates its socket using **socket()**, and then binds the socket to a well-known address using **bind()**. The server then calls **listen()** to allow connections to be received on the socket. Each client connection is then accepted on the listening socket using **accept()**, which returns a file descriptor for a **new socket** that is connected to the client's socket.
- A typical **stream socket client** creates a socket using **socket()**, and then establishes a connection by calling **connect()**, specifying the server's well-known address. After two stream sockets are connected, data can be transferred in either direction using **read()** and **write()**. Once all processes with a file descriptor referring to a stream socket endpoint have performed an implicit or explicit **close()**, the connection is terminated.

# Summary

---

- A typical **datagram socket server** creates a socket using *socket()*, and then binds it to a well-known address using *bind()*. Because datagram sockets are connectionless, the server's socket can be used to receive datagrams from any client.
- Datagrams can be received using *read()* or using the socket specific *recvfrom()* system call, which returns the address of the sending socket. A datagram socket client creates a socket using *socket()*, and then uses *sendto()* to send a datagram to a specified address. The *connect()* system call can be used with a datagram socket to set a peer address for the socket. After doing this, it is no longer necessary to specify the destination address for outgoing datagrams; a *write()* call can be used to send a datagram