Operating Systems Principles and Programming

Principes et programmation des systèmes d'exploitation

Albert Cohen

Albert.Cohen@inria.fr

École Polytechnique — Master 1 — INF583

2013–2014

More Contact Information

Albert Cohen: senior research scientist at INRIA

PARKAS group at ENS: http://www.di.ens.fr/ParkasTeam.html
Parallelism of synchronous Kahn networks

Organization

Practical Information

- 9 lectures (slides in English) and 9 labs (in French)
- Oral examination
- Questions are welcome
- If you are lost, do not wait before asking for help

Prerequisites

- Attending lectures and labs
- Programming, reading code and documentation after lab hours

http://www.enseignement.polytechnique.fr/informatique/INF583

Contents

Course

- Principles and design of operating systems
- Operating system programming
- Concrete examples

Labs

- Corrections for most exercises
- Balanced between principles, algorithms, system design, kernel internals and system programming

Outline

1. Survival Kit

- Technical Appendix: *l'annexe*
- The Shell and Classical UNIX Filters: cf. INF422
- Helper Tools for System Programming

Help Yourself

UNIX man pages

- Read man pages:
  http://www.linuxmanpages.com or http://linux.die.net/man
  - Quick reference in French:
    http://www.blaess.fr/christophe/documents.php?pg=001
  - BusyBox: shell for embedded systems:
    http://www.enseignement.polytechnique.fr/informatique/INF422/busybox.html
- Command-line usage
  - `$ man 1 command` (UNIX command)
  - `$ man 2 system_call` (primitive system calls)
  - `$ man 3 library_call` (e.g., C library, system call front-end stubs)
  - Warning: multiple entries with the same name may appear in different sections of the man pages
    → run `$ man -k name` if you are not sure
  - The SEE ALSO section at the bottom of most man pages is an important way to navigate through this valuable source of precise/reference information

C Language and Standard Library

This Is Not a Programming Course

You may like C or not...
But C and Operating Systems are meant to work together
There is no choice but to learn and practice it!

Getting Help

- Definitive reference: C language book by B. Kernighan and D. Ritchie
- Use quick reference card and online C tutorial (see INF583 web page)

1. Survival Kit

- Technical Appendix: *l'annexe*
- The Shell and Classical UNIX Filters: cf. INF422
- Helper Tools for System Programming

1. Survival Kit

- Technical Appendix: *l'annexe*
- The Shell and Classical UNIX Filters: cf. INF422
- Helper Tools for System Programming

1. Survival Kit

- Technical Appendix: *l'annexe*
- The Shell and Classical UNIX Filters: cf. INF422
- Helper Tools for System Programming

Processes and System Calls

Processes: `ps/pstree`

- All processes: `ps -ef` or `ps -efww` (full command line)
- Easier to use `pstree` to inspect child threads

System Call Trace: `strace`

- Traces the sequence of system calls (with their arguments and return status)
- Verbose, but fast and useful to debug concurrent programs

Open files: `lsof/fuser`

- List open files
- List processes using a specific file

Other Helper Tools

Debugger: `gdb`

- Read the technical appendix (*l'annexe*), page 93
- Always compile with `-g`

Project manager: `make`

- A `Makefile` is provided with the labs
- You are encouraged to learn how to write one and extend it if needed

Shell and Internationalization Tricks

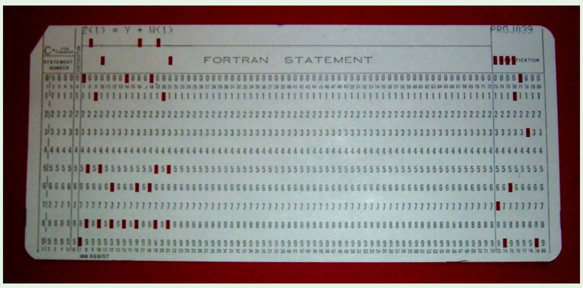
- Learn the basic functions of your *shell*
  Which shell am I using? `$ echo $SHELL`
- Hint: switch to another language (to English a priori)
  With `tcsh`: `$ setenv LANG C` or `$ setenv LANG fr_FR`
  With `bash`: `$ export LANG=C` or `$ export LANG=fr_FR`
  List internationalizations: `$ locale -a | more`

2. An Operating System, What For?

- Operating System Tasks
- Survey of Operating System Principles

Batch Processing

Punched Cards



Is it Enough?

There exist more interactive, complex, dynamic, extensible systems!

They require an *Operating System* (OS)

Operating System Tasks and Principles

Tasks
- Resource management
- Separation
- Communication

$\leftrightarrow$

Principles
- Abstraction
- Security
- Virtualization

2. An Operating System, What For?

- Operating System Tasks
- Survey of Operating System Principles

The Role of the Kernel: Separation, Communication

- The kernel is a *process manager*, not a process
- It runs with higher privileges (enforced by the microprocessor)
  - *User mode*: restricted instructions and access to memory
  - *Kernel mode*: no restriction
- User processes switch to kernel mode when requesting a service provided by the kernel
  - *Context switch*
  - *System call*

The Role of the Kernel: Resource Management

Control

- Bootstrap the whole machine
  Firmware, BIOS, EFI, boot devices, initialization sequence
- Configure I/O devices and low-level controllers
  Memory-mapped I/O, hardware interrupts
- Isolate and report errors or improper use of protected resources
  Kernel vs. user mode, memory protection, processor exceptions

Allocate

- Distribute processing, storage, communications, in time and space
  Process/task, multiprocessing, virtual memory, file system, networking ports
- Multi-user environment
  Session, identification, authorization, monitoring, terminal
- Fair resource use
  Scheduling, priority, resource limits

2. An Operating System, What For?

First OS Principle: Abstraction

Goal

- Simplify, standardize
  - Kernel portability over multiple hardware platforms
  - Uniform interaction with devices
  - Facilitate development of device drivers
  - Stable execution environment for the user programs

Main Abstractions

1. Process
2. File and file system
3. Device
4. Virtual memory
5. Naming
6. Synchronization
7. Communication

Process Abstraction

Single Execution Flow

- Process: *execution context of a running program*
- Multiprocessing: *private address space* for each process
  - Address spaces isolation enforced by the kernel and processor (see *virtual memory*)

Multiple Execution Flows

- Within a process, the program "spawns" multiple execution flows operating within the same address space: the *threads*
- Motivation
  - Less information to save/restore with the processor needs to switch from executing one thread to another (see *context switch*)
  - Communication between threads is trivial: shared memory accesses
- Challenge: threads need to *collaborate* when they *concurrently* access data
  - Pitfall: looks simpler than distributed computing, but hard to keep track of data sharing in large multi-threaded programs, and even harder to get the threads to collaborate correctly (non-deterministic behavior, non-reproducible bugs)

File and File System Abstractions

- *File*: *storage* and *naming* in UNIX
- *File System* (FS): repository (specialized database) of files
- Directory tree, absolute and relative pathnames
  `/` `.` `..` `/dev/hda1` `/bin/ls` `/etc/passwd`
- File types
  - Regular file or hard link (file name alias within a single file system)
    `$ ln pathname alias_pathname`
  - Soft link: short file containing a pathname
    `$ ln -s pathname alias_pathname`
  - Directory: list of file names (a.k.a. hard links)
  - Pipe (also called FIFO)
  - Socket (networking)
- Assemble multiple file systems through *mount points*
  Typical example: `/home` `/usr/local` `/proc`
- Common set system calls, independent of the target file system

Device Abstraction

- Device special files
  - *Block*-oriented device: disks, file systems
    `/dev/hda` `/dev/sdb2` `/dev/md1`
  - *Character*-oriented device: serial ports, console terminals, audio
    `/dev/tty0` `/dev/pts/0` `/dev/usb/hiddev0` `/dev/mixer` `/dev/null`

Virtual Memory Abstraction

- Processes access memory through *virtual addresses*
  - Simulates a large *interval* of memory addresses
  - Expressive and efficient address-space protection and separation
  - Hides kernel and other processes' memory
  - Automatic *translation* to *physical addresses* by the CPU (MMU/TLB circuits)
- *Paging* mechanism
  - Provide a protection mechanism for memory regions, called *pages*
  - The kernel implements a *mapping* of physical pages to virtual ones, different for every process
- *Swap* memory and file system
  - The ability to suspend a process and virtualize its memory allows to store its pages to disk, saving (expensive) RAM for more urgent matters
  - Same mechanism to migrate processes on NUMA multi-processors

Naming Abstraction

- Hard problem in operating systems
  - Processes are separated (logically and physically)
  - Need to access *persistent* and/or *foreign* resources
  - Resource *identification* determines large parts of the programming interface
  - Hard to get it right, general and flexible enough
- Good examples: /-separated filenames and pathnames
  - Uniform across complex directory trees
  - Uniform across multiple devices with *mount points*
  - Extensible with *file links* (a.k.a. aliases)
  - Reused for many other naming purposes: e.g., UNIX sockets, POSIX Inter-Process Communication (IPC)
- Could be better
  - INET addresses, e.g., 129.104.247.5, see the never-ending IPv6 story
  - TCP/UDP network ports
- Bad examples
  - Device numbers (UNIX internal tracking of devices)
  - Older UNIX System V IPC
  - MSDOS (and Windows) device letters (the ugly `C:\`)

Concurrency Abstraction

Synchronization

- Interprocess (or interthread) synchronization interface
  - Waiting for a process status change
  - Waiting for a signal
  - Semaphores (IPC)
  - Reading from or writing to a file (e.g., a pipe)

Communication

- Interprocess communication programming interface
  - Synchronous or asynchronous signal notification
  - Pipe (or FIFO), UNIX Socket
  - Message queue (IPC)
  - Shared memory (IPC)
- OS interface to network communications
  - INET Socket

Second OS Principle: Security

Basic Mechanisms

- Identification
  `/etc/passwd` and `/etc/shadow`, sessions (login)
  UID, GID, effective UID, effective GID
- Isolation of processes, memory pages, file systems
- Encryption, signature and key management
- Logging: `/var/log` and `syslogd` daemon
- Policies:
  - Defining a security policy
  - Enforcing a security policy

Enhanced Security: Examples

- SELinux: http://www.nsa.gov/selinux/papers/policy-abs.cfm
- Android security model: http://code.google.com/android/devel/security.html

Third OS Principle: Virtualization

"Every problem can be solved with an additional level of indirection"

2. An Operating System, What For? – Survey of Operating System Principles

Third OS Principle: Virtualization

"Every problem can be solved with an additional level of indirection"

Standardization Purposes

- Common, portable interface
- Software engineering benefits (code reuse)
  - Example: Virtual File System (VFS) in Linux = superset API for the features found in all file systems
  - Another example: drivers with SCSI interface emulation (USB mass storage)
- Security and maintenance benefits
  - Better isolation than processes
  - Upgrade the system transparently, robust to partial failures

Third OS Principle: Virtualization

"Every problem can be solved with an additional level of indirection"

Compatibility Purposes

- Binary-level compatibility
  - Processor and full-system virtualization: emulation, binary translation (*subject of the last chapter*)
  - Protocol virtualization: IPv4 on top of IPv6
- API-level compatibility
  - Java: through its virtual machine and SDK
  - POSIX: even Windows has a POSIX compatibility layer
  - Relative binary compatibility across some UNIX flavors (e.g., FreeBSD)

3. System Calls

- Principles and Implementation
- POSIX Essentials

Kernel Interface

Challenge: Interaction Despite Isolation

- How to isolate processes (in memory)...
- ... While allowing them to request help from the kernel...
- ... To access resources (in compliance with security policies)...
- ... And to interact

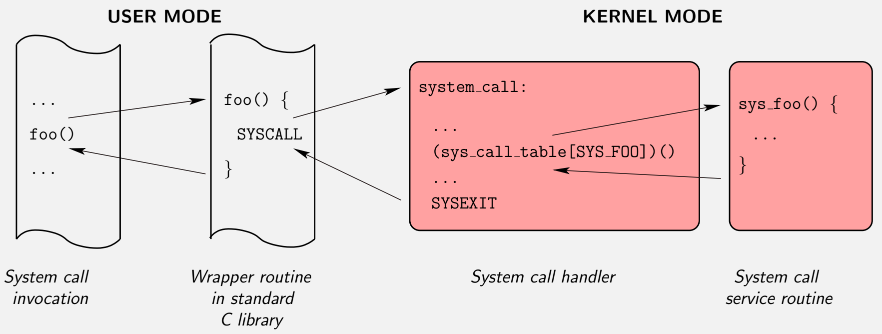
3. System Calls

- Principles and Implementation
- POSIX Essentials

System Call Principles

Information and Control Flow Across Priviledge Levels

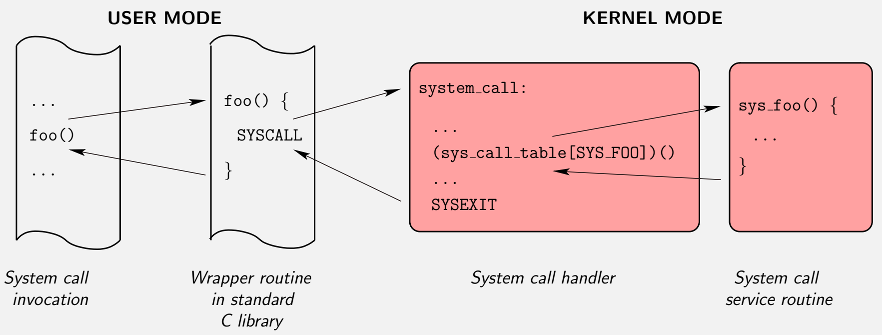
- Multiple indirections, switching from *user mode* to *kernel mode* and back (much more expensive than a function call)



System Call Implementation

C Library Wrapper

- All system calls defined in OS-specific header file
  Linux: `/usr/include/sys/syscall.h` (which includes `/usr/include/bits/syscall.h`)
- System call handlers are numbered
- C library wraps processor-specific parts into a plain function



System Call Implementation

Wrapper's Tasks

1. Move parameters from the user stack to processor registers
   Passing arguments through registers is easier than playing with both user and kernel stacks at the same time
2. Switch to kernel mode and jump to the system call handler
   Call processor-specific instruction (`trap`, `sysenter`, ...)
3. Post-process the return value and compute `errno`
   Linux: typically negate the value returned by the service function

Handler's Tasks

1. Save processor registers into the *kernel mode stack*
2. Call the service function in the kernel
   Linux: array of function pointers indexed by system call number
3. Restore processor registers
4. Switch back to user mode
   Call processor-specific instruction (`rti`, `sysexit`, ...)

System Call Implementation

Verifying the Parameters

- Can be call-specific
  E.g., checking a file descriptor corresponds to an open file
- General (coarse) check that the address is outside kernel pages
  Linux: less than `PAGE_OFFSET`
- Delay more complex page fault checks to address translation time
  1. Access to non-existent page of the process
     → no error but need to allocate (and maybe copy) a page on demand
  2. Access to a page outside the process space
     → issue a segmentation/page fault
  3. The kernel function itself is buggy and accesses and illegal address
     → call `oops()` (possibly leading to "kernel panic")

3. System Calls

- Principles and Implementation
- POSIX Essentials

POSIX Standard

Portable Operating System Interface

- IEEE POSIX 1003.1 and ISO/IEC 9945 (latest standard: 2004)
- Many subcommittees

Portability Issues

- POSIX is portable and does not evolve much,
- ... but it is still too high level for many OS interactions
  E.g., it does not specify file systems, network interfaces or power management
- UNIX applications deal with portability with
  - C-preprocessor conditional compilation
  - Conditional and multi-target `Makefile` rules
  - GNU `configure` scripts to generate `Makefile`s
  - Shell environment variables (`LD_LIBRARY_PATH`, `LD_PRELOAD`)

System Calls Essentials

Return Values and Errors

- System calls return an `int` or a `long` (sometimes hidden behind a POSIX standard type for portability)
  $\geqslant \mathbf{0}$ if execution proceeded normally
  $-\mathbf{1}$ if an error occurred
- When an error occurs, `errno` is set to the error code
  - Global scope, thread-local, `int` variable
  - It carries *semantical information* not available by any other mean
  - It is *not* reset to 0 before a system call
- `#include <errno.h>`

System Calls Essentials

Error Messages

- Print error message: `perror()` (see also `strerror()`)

Sample Error Codes

`EPERM`: Operation not permitted
`ENOENT`: No such file or directory
`ESRCH`: No such process
`EINTR`: Interrupted system call
`EIO`: I/O error
`ECHILD`: No child process
`EACCESS`: Access permission denied
`EAGAIN`/`EWOULDBLOCK`: Resource temporarily unavailable

System Calls Essentials

Standard Types

- `#include <sys/types.h>`
- Purpose: portability
- Alias for an integral type (`int` or `long` in general)

Examples

`clock_t`: clock ticks since last boot
`dev_t`: major and minor device numbers
`uid_t/gid_t`: user and group identifier
`pid_t`: process identifier
`mode_t`: access permissions
`sigset_t`: set of signal masks
`size_t`: size (unsigned, 64 bits in general)
`time_t`: seconds since 01/01/1970

System Calls Essentials

Interrupted System Calls

- Deliverling a *signal* interrupts system calls
- Hardware interrupts do not interrupt system calls (the kernel supports nesting of control paths)
- Rule 1: fail if the call did not have time to produce any effect

  Typically, return `EINTR`
- Rule 2: in case of partial execution (for a call where it means something), do not fail but return information allowing to determine the actual amount of partial progress

  See e.g., `read()` and `write()`

Trace of System Calls

```
$ strace ./hello
execve("./hello", ["./hello"], [/* 36 vars */])   = 0
brk(0)                                            = 0x0804a000
...
access("/etc/ld.so.nohwcap", F_OK)                = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)                = 3
fstat64(0x3, 0xffd1c12c)                          = 0
mmap2(NULL, 100777, PROT_READ, MAP_PRIVATE, 3, 0) = 0xf7f2e000
close(3)                                          = 0
...
open("/lib32/libc.so.6", O_RDONLY)                = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\1\000"..., 512) = 512
fstat64(0x3, 0xffd1c1c8)                          = 0
mmap2(NULL, 1336944, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3, 0) = 0xf7de7000
...
close(3)                                          = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xf7de6000
...
munmap(0xf7f2e000, 100777)                        = 0
fstat64(0x1, 0xffd1c9bc)                          = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xf7f46000
write(1, "Hello world!\n", 13)                    = 13
exit_group(0)                                     = ?
```

4. Files and File Systems

- Principles
- Structure and Storage
- Kernel Structures
- System Calls
- Directories
- Extended File Descriptor Manipulation
- Device-Specific Operations
- I/O Redirection
- Communication Through a Pipe

4. Files and File Systems

- Principles
- Structure and Storage
- Kernel Structures
- System Calls
- Directories
- Extended File Descriptor Manipulation
- Device-Specific Operations
- I/O Redirection
- Communication Through a Pipe

Storage Structure: Inode

Index Node

- UNIX distinguishes *file data* and *information about a file* (or meta-data)
- File information is stored in a structure called *inode*
- Attached to a particular device

Attributes

File Type
Number of hard links (they all share the same inode)
File length in bytes
Device identifier (DID)
User identifier (UID, file owner)
User group identifier (GID, user group of the file)
Timestamps: last status change (e.g., creation), modification, and access time
Access rights and file mode

Possibly more (non-POSIX) attributes, depending on the file system

Inode: Access Rights

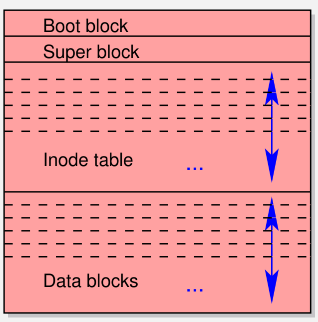
- Classes of file accesses
  - **user:** owner
  - **group:** users who belong to the file's group, excluding the owner
  - **others:** all remaining users
- Classes of access rights
  - **read:** *directories: controls listing*
  - **write:** *directories: controls file status changes*
  - **execute:** *directories: controls searching (entering)*
- Additional file modes
  - **suid:** with **execute**, the process gets the file's UID
    directories: nothing
  - **sgid:** with **execute**, the process gets the file's GID
    directories: created files inherit the creator process's GID
  - **sticky:** loosely specified semantics related to memory management
    directories: files owned by others cannot be deleted or renamed

4. Files and File Systems

File System Storage

General Structure

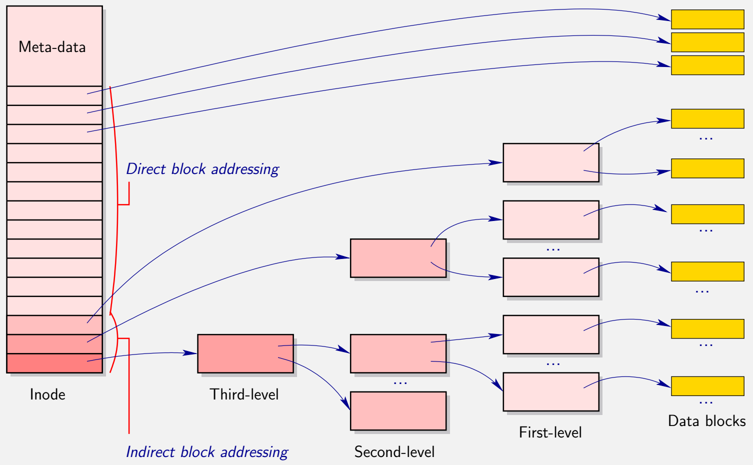
- Boot block
  - Bootstrap mode and “bootable” flag
  - Link to data blocks holding boot code
- Super block
  - File system status (mount point)
  - Number of allocated and free nodes
  - Link to lists of allocated and free nodes
- Inode table
- Data blocks
  - Note: directory = list of file names



Simplified file system layout

Inode: Data Block Addressing

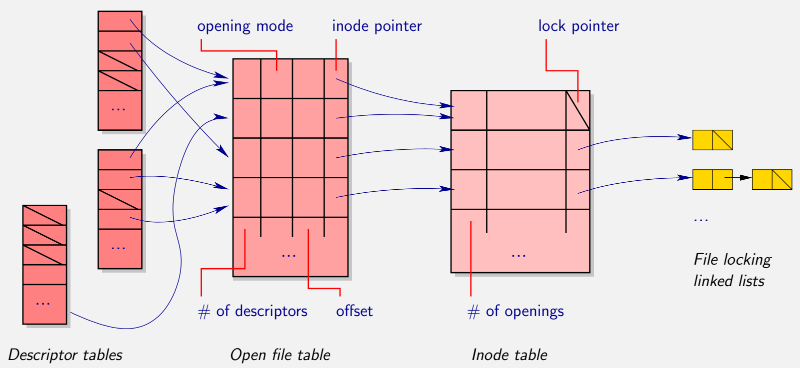
- Every Inode has a table of block addresses
- Addressing: direct, one-level indirect, two-levels indirect, ...



4. Files and File Systems

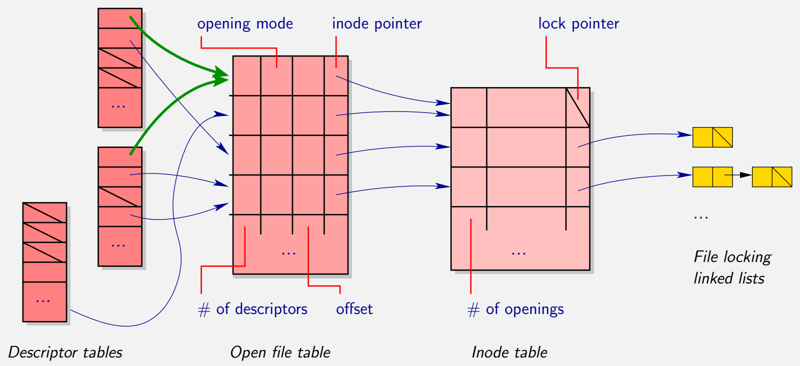
I/O Kernel Structures

- One table of *file descriptor*s per process: **0**:`stdin`, **1**:`stdout`, **2**:`stderr`
- Table of *open file*s (status, including opening mode and offset)
- Inode table (for all open files)
- File locks (*see chapter on advanced synchronization*)



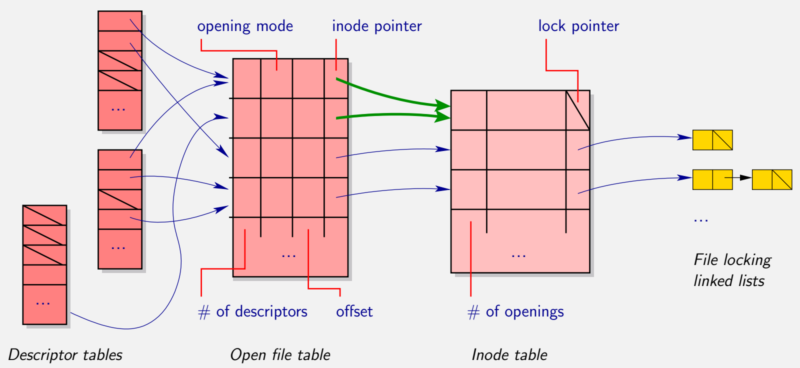
I/O Kernel Structures

- Example: file descriptor aliasing
  - E.g., obtained through the `dup()` or `fork()` system calls



I/O Kernel Structures

- Example: open file aliasing
  - E.g., obtained through multiple calls to `open()` on the same file
  - Possibly via hard or soft links



4. Files and File Systems

I/O System Calls

Inode Manipulation

- `stat()` `access()`, `link()`, `unlink()`, `chown()`, `chmod()`, `mknod()`, ...
- Note: many of these system calls have `l`-prefixed variants (e.g., `lstat()`) that *do not* follow soft links
- Note: many of these system calls have `f`-prefixed variants (e.g., `fstat()`) operating on *file descriptors*
  Warning: they are not to be confused with C library functions

4. Files and File Systems – System Calls

I/O System Calls

Inode Manipulation

- `stat()` `access()`, `link()`, `unlink()`, `chown()`, `chmod()`, `mknod()`, ...
- Note: many of these system calls have `l`-prefixed variants (e.g., `lstat()`) that *do not* follow soft links
- Note: many of these system calls have `f`-prefixed variants (e.g., `fstat()`) operating on *file descriptors*
  Warning: they are not to be confused with C library functions

File descriptor manipulation

- `open()`, `creat()`, `close()`, `read()`, `write()`, `lseek()`, `fcntl()`...
- We will describe `dup()` when studying redirections
- Note: `open()` may also create a new file (hence a new inode)
- Use `fdopen()` and `fileno()` to get a file C library `FILE*` from a file descriptor and reciprocally, but *do not mix C library and system call I/O on the same file (because of C library internal buffers)*

I/O System Call: `stat()`

Return Inode Information About a File

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

int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
int fstat(int fd, struct stat *buf);
```

Error Conditions

- The system call returns **0** on success, **−1** on error
- A few possible `errno` codes

  `EACCES`: search (enter) permission is denied for one of the directories in the prefix of `path`

  `ENOENT`: a component of `path` does not exist — file not found — or the path is an empty string

  `ELOOP`: too many symbolic links encountered when traversing the path

I/O System Call: stat()

Inode Information Structure

```
struct stat {
  dev_t      st_dev;     // ID of device containing file
  ino_t      st_ino;     // Inode number
  mode_t     st_mode;    // Protection
  nlink_t    st_nlink;   // Number of hard links
  uid_t      st_uid;     // User ID of owner
  gid_t      st_gid;     // Group ID of owner
  dev_t      st_rdev;    // Device ID (if special file)
  off_t      st_size;    // Total size, in bytes
  blksize_t st_blksize; // Blocksize for filesystem I/O
  blkcnt_t   st_blocks;  // Number of blocks allocated
  time_t     st_atime;   // Time of last access
  time_t     st_mtime;   // Time of last modification
  time_t     st_ctime;   // Time of last status change
};
```


I/O System Call: stat()

Deciphering st_mode

Macros to determine file type

- S_ISREG(m): is it a regular file?
- S_ISDIR(m): directory?
- S_ISCHR(m): character device?
- S_ISBLK(m): block device?
- S_ISFIFO(m): FIFO (named pipe)?
- S_ISLNK(m): symbolic link?
- S_ISSOCK(m): socket?

File type constants

- S_IFREG: regular file
- S_IFDIR: directory
- S_IFCHR: character device
- S_IFBLK: block device
- S_IFFIFO: FIFO (named pipe)
- S_IFLNK: symbolic link
- S_IFSOCK: socket

I/O System Call: stat()

Deciphering st_mode

Macros to determine access permission and mode
Usage: *flags* and *masks* can be *or'ed* and *and'ed* together, and with st_mode

Constant	Octal value	Comment
S_ISUID	04000	SUID bit
S_ISGID	02000	SGID bit
S_ISVTX	01000	sticky bit
S_IRWXU	00700	mask for file owner permissions
S_IRUSR	00400	owner has read permission
S_IWUSR	00200	owner has write permission
S_IXUSR	00100	owner has execute permission
S_IRWXG	00070	mask for group permissions
S_IRGRP	00040	group has read permission
S_IWGRP	00020	group has write permission
S_IXGRP	00010	group has execute permission
S_IRWXO	00007	mask for permissions for others
S_IROTH	00004	others have read permission
S_IWOTH	00002	others have write permission
S_IXOTH	00001	others have execute permission

I/O System Call: access()

Check Whether the Process Is Able to Access a File

```
#include <unistd.h>

int access(const char *pathname, int mode);
```

Access Mode Requests

- R_OK: check for read permission
- W_OK: check for write permission
- X_OK: check for execute permission
- F_OK: check for the existence of the file

Error Conditions

- The system call returns **0** on success, $-\mathbf{1}$ on error
- A few original errno codes
  - EROFS: write access request on a read-only filesystem
  - ETXTBSY: write access request to an executable which is being executed

I/O System Call: link()

Make a New Name (Hard Link) for a File

```
#include <unistd.h>

int link(const char *oldpath, const char *newpath);
```

See also: symlink()

Error Conditions

- Cannot hard-link directories (to maintain a tree structure)
- The system call returns **0** on success, $-\mathbf{1}$ on error
- A few original errno codes
  - EEXIST: newpath already exists (link() preserves existing files)
  - EXDEV: oldpath and newpath are not on the same file system

I/O System Call: unlink()

Delete a Name and Possibly the File it Refers To

```
#include <unistd.h>

int unlink(const char *pathname);
```

Error Conditions

- The system call returns **0** on success, $-\mathbf{1}$ on error
- An original errno code
  - EISDIR: attempting to delete a directory (see rmdir())

I/O System Call: chown()

Change Ownership of a File

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

int chown(const char *path, uid_t owner, gid_t group);
int lchown(const char *path, uid_t owner, gid_t group);
int fchown(int fd, uid_t owner, gid_t group);
```

Error Conditions

- The system call returns **0** on success, **−1** on error
- An original `errno` code
  - `EBADF`: the descriptor is not valid

I/O System Call: chmod()

Change Access Permissions of a File

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

int chmod(const char *path, mode_t mode);
int fchmod(int fildes, mode_t mode);
```

Access Permissions

- Build `mode` argument by *or'ing* the access mode constants
  E.g., `mode = S_IRUSR | S_IRGRP | S_IROTH; // 0444`

Error Conditions

- The system call returns **0** on success, **−1** on error

I/O System Call: mknod()

Create any Kind of File (Inode)

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

int mknod(const char *pathname, mode_t mode, dev_t dev);
```

File Type

- Set `mode` argument to *one* of the file type constants *or'ed* with any combination of access permissions
  E.g., `mode = S_IFREG | S_IRUSR | S_IXUSR; // Regular file`
- If `mode` is set to `S_IFCHR` or `S_IFBLK`, `dev` specifies the *major* and *minor* numbers of the newly created device special file
- File is created with permissions (`mode & ~current_umask`) where `current_umask` is the process's mask for file creation (see `umask()`)

I/O System Call: mknod()

Create any Kind of File (Inode)

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

int mknod(const char *pathname, mode_t mode, dev_t dev);
```

Error Conditions

- The system call returns **0** on success, **−1** on error
- A few original errno codes

  EEXIST: newpath already exists (mknod() preserves existing files)
  ENOSPC: device containing pathname has no space left for a new node

I/O System Call: open()/creat()

Open and Possibly Create a File

```
#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);
int creat(const char *pathname, mode_t mode);
```

Return Value

- On success, the system call returns a (non-negative) *file descriptor*
  - Note: it is the process's *lowest-numbered file descriptor not currently open*
- Return **−1** on error

I/O System Call: open()/creat()

Open and Possibly Create a File

```
#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);
int creat(const char *pathname, mode_t mode);
```


Acces Permissions

- File is created with permissions (mode & ~current_umask) where current_umask is the process's mask for file creation (see umask())

I/O System Call: open()/creat()

Open and Possibly Create a File

```
#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);
int creat(const char *pathname, mode_t mode);
```

Flags

- Access mode set to *one* of O_RDONLY, O_WRONLY, O_RDWR
  Note: opening a file in read-write mode is very different from opening it twice in read then write modes (see e.g. the behavior of lseek())
- Possibly *or'ed* with O_APPEND, O_CREAT, O_EXCL, O_TRUNC, O_NONBLOCK, ...

I/O System Call: close()

Close a File Descriptor

```
#include <unistd.h>

int close(int fd);
```

Remarks

- When closing the last descriptor to a file that has been removed using unlink(), the file is effectively deleted
- It is sometimes desirable to flush all pending writes (persistent storage, interactive terminals): see fsync()

Error Conditions

- The system call returns **0** on success, $-\mathbf{1}$ on error
- It is important to check error conditions on close(), to avoid losing data

I/O System Call: read()

Read From a File Descriptor

```
#include <unistd.h>

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

Semantics

- Attempts to read *up to* count bytes from file descriptor fd into the buffer starting at buf
  - Return immediately if count is **0**
  - May read less than count bytes: it is not an error
    E.g., close to end-of-file, interrupted by signal, reading from socket...
- On success, *returns the number of bytes effectively read*
- Return **0** if at *end-of-file*
- Return $-\mathbf{1}$ on error (hence the signed ssize_t)

I/O System Call: read()

Read From a File Descriptor

```
#include <unistd.h>

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

Error Conditions

- Important errno codes

  EINTR: call interrupted by a signal *before anything was read*
  EAGAIN: non-blocking I/O is selected and no data was available

I/O System Call: write()

Write to File Descriptor

```
#include <unistd.h>

ssize_t write(int fd, const void *buf, size_t count);
```

Semantics

- Attempts to write *up to* count bytes to the file file referenced by the file descriptor fd from the buffer starting at buf
  - Return immediately if count is **0**
  - May write less than count bytes: it is not an error
- On success, *returns the number of bytes effectively written*
- Return $-\mathbf{1}$ on error (hence the signed ssize_t)

Error Conditions

- An original errno code

  ENOSPC: no space left on device containing the file

Example: Typical File Open/Read Skeleton

```
void my_read(char *pathname, int count, char *buf)
{
  int fd;
  if ((fd = open(pathname, O_RDONLY)) == -1) {
    perror("‘my_function’: ‘open()’ failed");
    exit(1);
  }
  // Read count bytes
  int progress, remaining = count;
  while ((progress = read(fd, buf, remaining)) != 0) {
    // Iterate while progess or recoverable error
    if (progress == -1) {
      if (errno == EINTR)
        continue; // Interrupted by signal, retry
      perror("‘my_function’: ‘read()’ failed");
      exit(1);
    }
    buf += progress; // Pointer artithmetic
    remaining -= progress;
  }
}
```

4. Files and File Systems

- Principles
- Structure and Storage
- Kernel Structures
- System Calls
- Directories
- Extended File Descriptor Manipulation
- Device-Specific Operations
- I/O Redirection
- Communication Through a Pipe

Directory Traversal

Directory Manipulation (C Library)

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);
struct dirent *readdir(DIR *dir);
int closedir(DIR *dir);
```

`$ man 3 opendir`, `$ man 3 readdir`, etc.

Directory Entry Structure

```
struct dirent {
  long d_ino;                   // Inode number
  off_t d_off;                  // Offset to this dirent
  unsigned short d_reclen;      // Length of this d_name
  char d_name[NAME_MAX+1];      // File name ('\0'-terminated)
}
```


Example: Mail Folder Traversal

```
int last_num(char *directory)
{
  struct dirent *d;
  DIR *dp;
  int max = -1;
  dp = opendir(directory);
  if (dp == NULL) {
    perror("‘last_num’: ‘opendir()’ failed");
    exit(1);
  }
  while ((d = readdir(dp)) != NULL) {
    int m;
    m = atoi(d->d_name); // Parse string into ‘int’
    max = MAX(max, m);
  }
  closedir(dp);
  return max; // -1 or n >= 0
}
```

Example: Mail Folder Traversal

```
void remove_expired(char *directory, int delay, int last_num)
{
  struct dirent *d;
  time_t now;
  struct stat stbuf;
  DIR *dp = opendir(directory);
  if (dp == NULL) {
    message(1, "‘remove_expired’: ’opendir()’ failed");
    return;
  }
  time(&now);
  while ((d = readdir(dp)) != NULL) {
    int m = atoi(d->d_name);
    if (m >= 0 && m != last_num) {
      if (stat(d->d_name, &stbuf) != -1 && stbuf.st_mtime < now - delay)
        unlink(d->d_name);
    }
  }
  closedir(dp);
}
```


4. Files and File Systems

I/O System Call: fcntl()

Manipulate a File Descriptor

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

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
```

Some Commands

F_GETFL: get the file status flags

F_GETFL: set the file status flags to the value of arg

- No access mode (e.g., O_RDONLY) and creation flags (e.g., O_CREAT), but accepts O_APPEND, O_NONBLOCK, O_NOATIME, etc.

And many more: descriptor behavior options, duplication and locks, I/O-related signals (terminals, sockets), etc.

- See chapter on processes and on advanced synchronization

I/O System Call: fcntl()

Manipulate a File Descriptor

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

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
```

Return Value

- On success, fcntl() returns a (non-negative) value which depends on the command
  - F_GETFD: the descriptor's flags
  - F_GETFD: **0**
- Return **−1** on error

4. Files and File Systems

Device-Specific Operations

I/O "Catch-All" System Call: ioctl()

- Implement operations that do not directly fit into the stream I/O model (read and write)
- Typical examples
  - Block-oriented devices: CD/DVD *eject* operation
  - Character-oriented devices: *terminal* control
- Prototype

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

int ioctl(int fd, int request, char *argp);
```

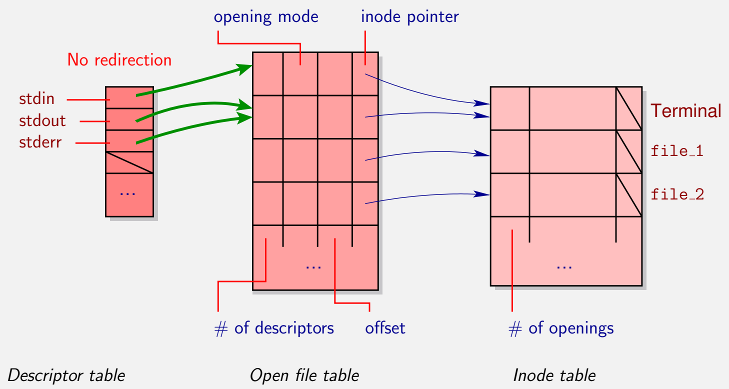
fd: open file descriptor
request: device-dependent request code
argp: buffer to load or store data (its size and structure is request-dependent)

4. Files and File Systems

I/O Redirection

Example

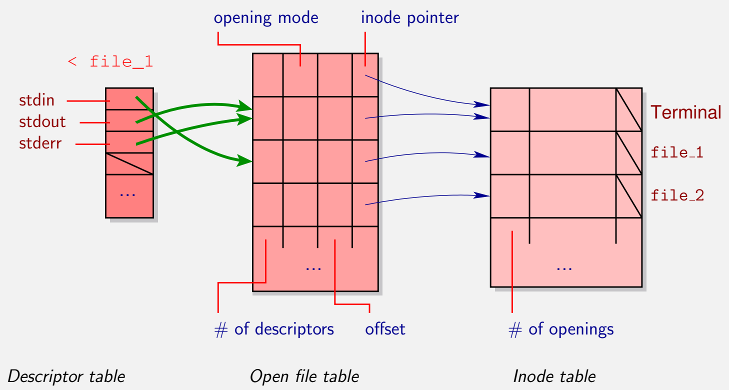
No redirection



I/O Redirection

Example

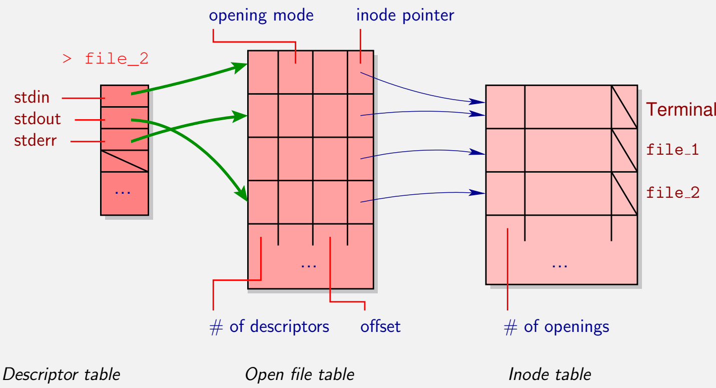
Standard input redirection



I/O Redirection

Example

Standard output redirection

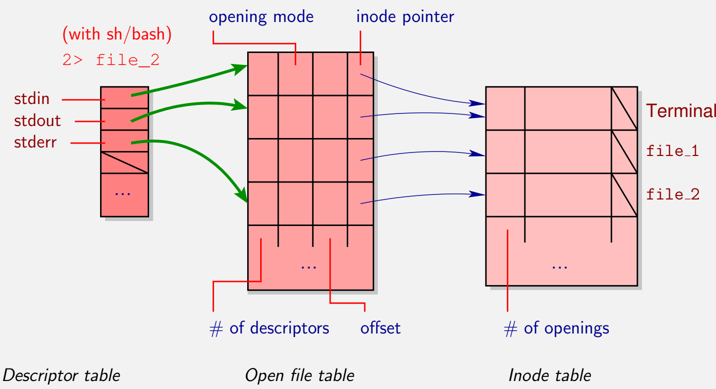


4. Files and File Systems – I/O Redirection

I/O Redirection

Example

Standard error redirection



4. Files and File Systems – I/O Redirection

I/O System Call: dup()/dup2()

Duplicate a File Descriptor

```
#include <unistd.h>

int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Return Value

- On success, `dup()`/`dup2()` return a *file descriptor*, a copy of `oldfd`
  - For `dup()`, it is the process's *lowest-numbered descriptor not currently open*
  - `dup2()` uses `newfd` instead, closing it before if necessary
  - Clears the flags of the new descriptor (see `fcntl()`)
  - Both descriptors share one single open file (i.e., one offset for `lseek()`, etc.)
- Return $-\mathbf{1}$ on error

Error Conditions

- An original `errno` code

  `EMFILE`: too many file descriptors for the process

Redirection Example

```
$ command > file_1   // Redirect stdout to file_1

{
  close(1);
  open("file_1", O_WRONLY | O_CREAT, 0777);
}
```

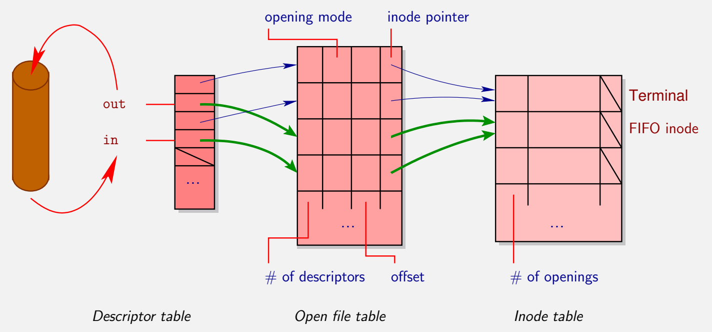
```
$ command 2>&1       // Redirect stderr to stdout

{
  close(2);
  dup(1);
}
```


4. Files and File Systems

FIFO (Pipe)

Principles

- Channel to stream data among processes
  - Data traverses the pipe *first-in* (write) *first-out* (read)
  - Blocking read and write by default (bounded capacity)
  - Illegal to write into a pipe without reader
  - A pipe without writer simulates *end-of-file*: `read()` returns **0**
- Pipes have *kernel* persistence



I/O System Call: pipe()

Create a Pipe

```
#include <unistd.h>

int pipe(int p[2]);
```

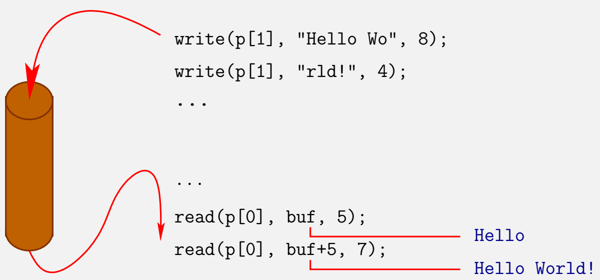
Description

- Creates a pipe and stores a pair of file descriptors into p
  - p[0] for reading (O_RDONLY)
  - p[1] for writing (O_WRONLY)
- Return **0** on success, **−1** if an error occurred

FIFO Communication

Unstructured Stream

- Like ordinary files, data sent to a pipe is unstructured: it does not retain "boundaries" between calls to write() (unlike IPC message queues)



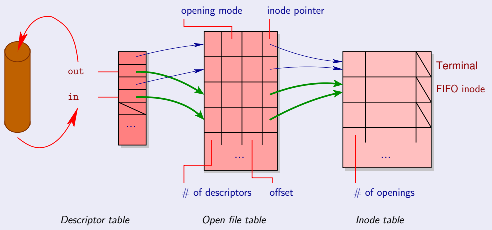
FIFO Communication

Writing to a Pipe

- Writing to a pipe without readers delivers of SIGPIPE
  - Causes termination by default
  - Otherwise causes write() to fail with error EINTR
- PIPE_BUF is a constant $\geqslant$ **512** (**4096** on Linux)
- Writing **n** bytes in blocking mode
  - **n** $\leqslant$ PIPE_BUF: atomic success (**n** bytes written), block if not enough space
  - **n** $>$ PIPE_BUF: non-atomic (may be interleaved with other), blocks until **n** bytes have been written
- Writing **n** bytes in non-blocking mode (O_NONBLOCK)
  - **n** $\leqslant$ PIPE_BUF: atomic success (**n** bytes written), or fails with EAGAIN
  - **n** $>$ PIPE_BUF: if the pipe is full, fails with EAGAIN; otherwise a partial write may occur

FIFOs and I/O Redirection

- Question: implement
  `$ ls | more`
- Solution
  - `pipe(p)`
  - `fork()`
  - Process to become `ls`
    - `close(1)`
    - `dup(p[1])`
    - `execve("ls", ...)`
  - Process to become `more`
    - `close(0)`
    - `dup(p[0])`
    - `execve("more", ...)`
- Short-hand: `$ man 3 popen`



FIFO Special Files

Named Pipe

- Special file created with `mkfifo()` (front-end to `mknod()`)
  - See also `mkfifo` command
- Does not store anything on the file system (beyond its inode)
  - Data is stored and forwarded in memory (like an unnamed pipe)
- Supports a rendez-vous protocol
  - Open for reading: blocks until another process opens for writing
  - Open for writing: blocks until another process opens for reading
- Disabled when opening in `O_NONBLOCK` mode

5. Processes and Memory Management

5. Processes and Memory Management

Logical Separation of Processes

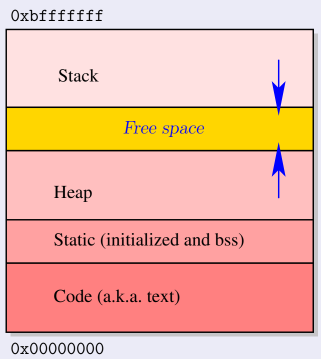
Kernel Address Space for a Process

- Process *descriptor*
  - Memory mapping
  - Open file descriptors
  - Current directory
  - Pointer to kernel stack
- Kernel stack
  - Small by default; grows in extreme cases of nested interrupts/exceptions
- Process table
  - Associative table of PID-indexed process descriptors
  - Doubly-linked tree (links to both children and parent)

Logical Separation of Processes

User Address Space for a Process

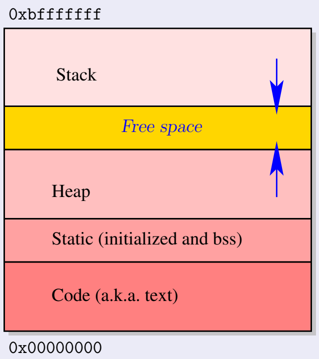
- Allocated and initialized when loading and executing the program
- *Memory accesses in user mode are restricted to this address space*



Logical Segments in Virtual Memory

Per-Process Virtual Memory Layout

- *Code* (also called *text*) segment
  - Linux: ELF format for object files (`.o` and executable)
- *Static Data* segments
  - Initialized global (and C `static`) variables
  - Uninitialized global variables
    - Zeroed when initializing the process, also called *bss*
- *Stack* segment
  - Stack frames of function calls
  - Arguments and local variables, also called `auto`matic variables in C
- *Heap* segment
  - Dynamic allocation (`malloc()`)



System Call: `brk()`

Resize the Heap Segment

```
#include <unistd.h>

int brk(void *end_data_segment);

void *sbrk(intptr_t displacement);
```

Semantics

- Sets the *end* of the data segment, which is also the end of the heap
  - `brk()` sets the address directly and returns **0** on success
  - `sbrk()` adds a displacement (possibly **0**) and returns the *starting* address of the new area (it is a C function, front-end to `sbrk()`)
- Both are *deprecated* as “programmer interface” functions, i.e., they are meant for kernel development only

Memory Address Space Example

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

double t[0x02000000];

void segments()
{
  static int s = 42;
  void *p = malloc(1024);

  printf("stack\t%010p\nbrk\t%010p\nheap\t%010p\n"
         "static\t%010p\nstatic\t%010p\ntext\t%010p\n",
         &p, sbrk(0), p, t, &s, segments);
}

int main(int argc, char *argv[])
{
  segments();
  exit(0);
}
```

Memory Address Space Example

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

double t[0x02000000];

void segments()
{
  static int s = 42;
  void *p = malloc(1024);

  printf("stack\t%010p\nbrk\t%010p\nheap\t%010p\n"
         "static\t%010p\nstatic\t%010p\ntext\t%010p\n",
         &p, sbrk(0), p, t, &s, segments);
}

int main(int argc, char *argv[])
{
  segments();
  exit(0);
}
```

Sample Output

stack	0xbff86fe0
brk	0x1806b000
heap	0x1804a008
static (bss)	0x08049720
static (initialized)	0x080496e4
text	0x080483f4

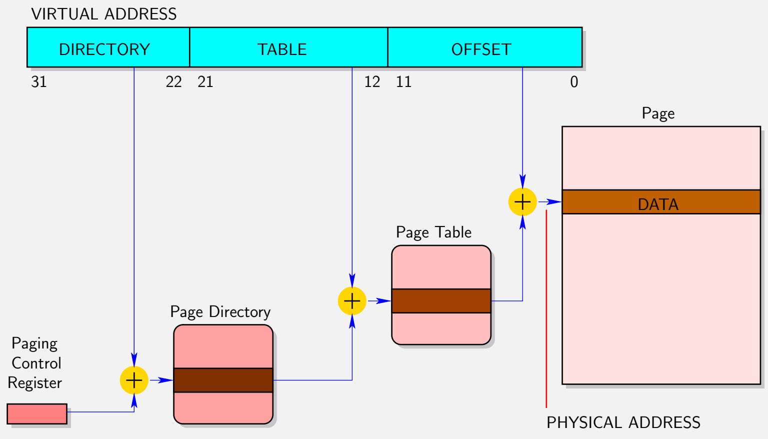
5. Processes and Memory Management

Introduction to Memory Management

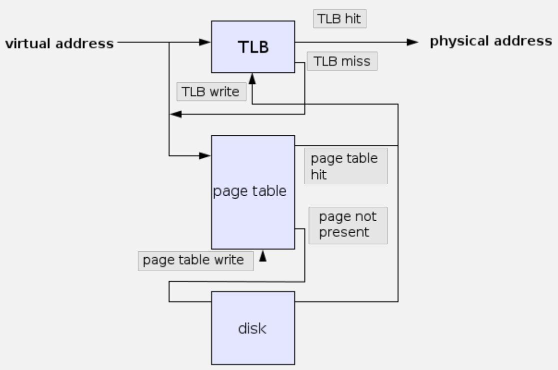
Paging Basics

- Processes access memory through *virtual* addresses
  - Simulates a large *interval* of memory addresses
  - Simplifies memory management
  - Automatic *translation* to *physical* addresses by the CPU (MMU/TLB circuits)
- *Paging* mechanism
  - Provide a protection mechanism for memory regions, called *pages*
  - Fixed 2^n page size(s), e.g., 4kB and 2MB on x86
  - The kernel implements a *mapping* of physical pages to virtual ones
    - *Different for every process*
- Key mechanism to ensure *logical separation* of processes
  - Hides kernel and other processes' memory
  - Expressive and efficient address-space protection and separation

Address Translation for Paged Memory



Page Table Actions



Page Table Structure(s)

Page Table Entry

- Physical address
- Valid/Dirty/Accessed
- Kernel R/W/X
- User R/W/X

Physical Page Mapping

E.g., Linux's `mem_map_t` structure:

- `counter` – how many users are mapping a physical page
- `age` – timestamp for swapping heuristics: Belady algorithm
- `map_nr` – Physical page number

Plus a free area for page allocation and dealocation

Saving Resources and Enhancing Performance

Lazy Memory Management

- Motivation: high-performance memory allocation
  - *Demand-paging*: delay the allocation of a memory page and its *mapping* to the process's virtual address space until the process *accesses* an address in the range associated with this page
  - Allows *overcommitting*: more economical than eager allocation (like overbooking in public transportation)
- Motivation: high-performance process creation
  - *Copy-on-write*: when cloning a process, do not replicate its memory, but mark its pages as "*need to be copied on the next write access*"
  - Critical for UNIX
    - Cloning is the only way to create a new process
    - Child processes are often short-lived: they are quickly overlapped by the execution of another program (see `execve()`)

Software Caches

- Buffer cache for block devices, and page cache for file data
- Swap cache to keep track of clean pages in the swap (disk)

C Library Function: `malloc()`

Allocate Dynamic Memory

```
#include <stdlib.h>

void *malloc(size_t size);
```

Semantics

- On success, returns a pointer to a *fresh interval* of `size` bytes of *heap* memory
- Return `NULL` on error
- See also `calloc()` and `realloc()`

C Library Function: `malloc()`

Allocate Dynamic Memory

```
#include <stdlib.h>

void *malloc(size_t size);
```

Semantics

- On success, returns a pointer to a *fresh interval* of `size` bytes of *heap* memory
- Return `NULL` on error
- See also `calloc()` and `realloc()`

- Warning: many OSes *overcommit* memory by default (e.g., Linux)
  - Minimal memory availability check and optimistically return non-`NULL`
  - Assume processes will not use all the memory they requested
  - When the system really runs out of free physical pages (after all swap space has been consumed), a kernel heuristic selects a non-`root` process and kills it to free memory for the requester (quite unsatisfactory, but often sufficient)

System Call: `free()`

Free Dynamic Memory

```
#include <stdlib.h>

void free(void *ptr);
```

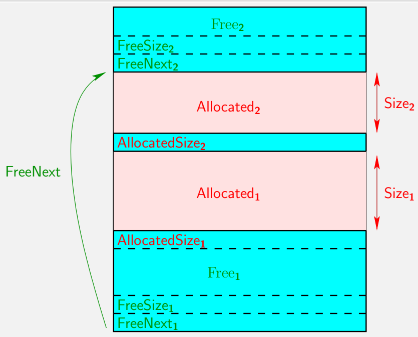
Semantics

- Frees the memory interval pointed to by `ptr`, which *must* be the return value of a previous `malloc()`
- Undefined behaviour if it is not the case
  (very nasty in general, because the bug may reveal much later)
- No operation is performed if `ptr` is `NULL`
- The dedicated `valgrind` tool instruments memory accesses and system calls to track memory leaks, phantom pointers, corrupt calls to `free()`, etc.

Memory Management of User Processes

Memory Allocation

- Appears in every aspect of the system
  - Major performance impact: highly optimized
- *Free list*: record linked list of free zones in the *free* memory space only
  - Record the address of the *next free zone*
  - Record the size of the allocated zone prior to its effective bottom address



Memory Management of User Processes

Memory Allocation

- Appears in every aspect of the system
  - Major performance impact: highly optimized
- *Buddy system*: allocate contiguous pages of physical memory
  - Coupled with free list for intra-page allocation
  - Contiguous physical pages improve performance (better TLB usage and DRAM control)

Intervals:
A: 64kB
B: 128kB
C: 64kB
D: 128kB

Empty	1024					
Allocate A	A	64	128	256		512
Allocate B	A	64	B	256		512
Allocate C	A	C	B	256		512
Allocate D	A	C	B	D	128	512
Free C	A	64	B	D	128	512
Free A	128		B	D	128	512
Free B	256			D	128	512
Free D	1024					

5. Processes and Memory Management

- Process Abstraction
- Introduction to Memory Management
- **Process Implementation**
- States and Scheduling
- Programmer Interface
- Process Genealogy
- Daemons, Sessions and Groups

Process Descriptor

Main Fields of the Descriptor

State	ready/running, stopped, zombie...
Kernel stack	typically one memory page
Flags	e.g., `FD_CLOEXEC`
Memory map	pointer to table of memory page descriptors (maps)
Parent	pointer to parent process (allow to obtain PPID)
TTY	control terminal (if any)
Thread	TID and thread information
Files	current directory and table of file descriptors
Limits	resource limits, see `getrlimit()`
Signals	signal handlers, masked and pending signals

Operations on Processes

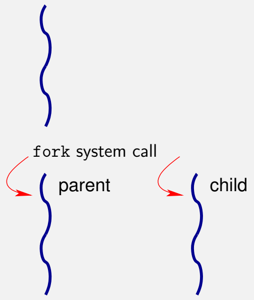
Basic Operations on Processes

- Cloning
  `fork()` system call, among others
- Joining (*see next chapter*)
  `wait()` system call, among others
- Signaling events (*see next chapter*)
  `kill()` system call, signal handlers

Creating Processes

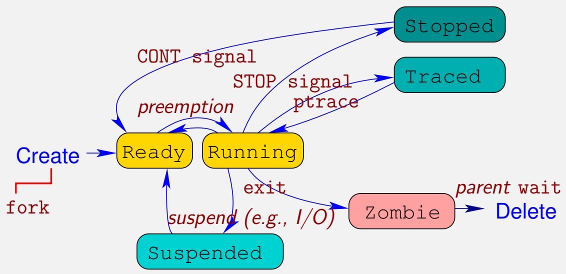
Process Duplication

- Generate a clone of the *parent* process
- The *child* is almost identical
  - It executes the same program
  - In a copy of its virtual memory space



5. Processes and Memory Management

Process States

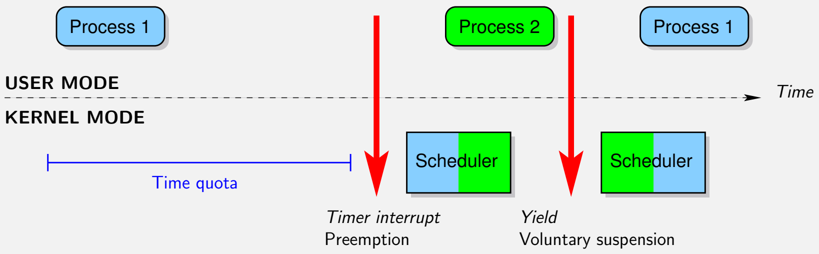


- Ready (runnable) process waits to be scheduled
- Running process make progress on a hardware thread
- Stopped process awaits a continuation signal
- Suspended process awaits a wake-up condition from the kernel
- Traced process awaits commands from the debugger
- Zombie process retains termination status until parent is notified
- Child created as Ready after `fork()`
- Parent is Stopped between `vfork()` and child `execve()`

Process Scheduling

Preemption

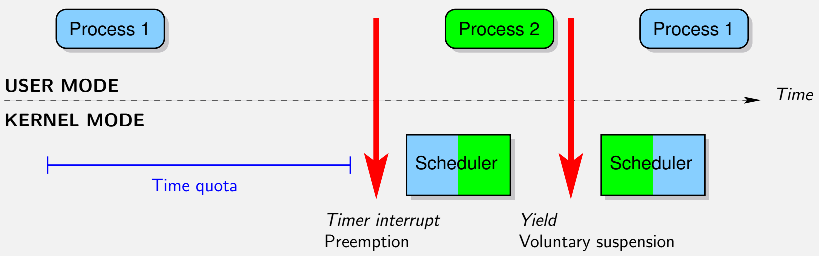
- Default for multiprocessing environments
- Fixed *time quota* (typically $\mathbf{1}\mathrm{ms}$ to $\mathbf{10}\mathrm{ms}$)
- Some processes, called *real-time*, may not be preempted



Process Scheduling

Voluntary Yield

- Suspend execution and yield to the kernel
  - E.g., I/O or synchronization
  - Only way to enable a context switch for *real-time* processes



5. Processes and Memory Management

System Call: fork()

Create a Child Process

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

pid_t fork();
```

Semantics

- The *child* process is identical to its *parent*, except:
  - Its PID and PPID (parent process ID)
  - Zero resource utilization (initially, relying on copy-on-write)
  - No pending signals, file locks, inter-process communication objects
- *On success, returns the child PID in the parent, and* **0** *in the child*
  - Simple way to detect "from the inside" which of the child or parent runs
  - See also `getpid()`, `getppid()`
- Return −**1** on error
- Linux: `clone()` is more general, for both *process* and *thread* creation

System Call: fork()

Create a Child Process

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

pid_t fork();
```

Typical Usage

```
switch (cpid = fork()) {
  case -1:                   // Error
    perror("‘my_function’: ‘fork()’ failed");
    exit(1);
  case 0:                    // The child executes
    continue_child();
    break;
  default:                   // The parent executes
    continue_parent(cpid);   // Pass child PID for future reference
}
```


System Call: execve() and variants

Execute a Program

```
#include <unistd.h>

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

Semantics

- Arguments: absolute path, argument array (a.k.a. vector), environment array (shell environment variables)
- On success, the call *does not return*!
  - Overwrites the process's *text*, *data*, *bss*, *stack* segments with those of the loaded program
  - Preserve PID, PPID, open file descriptors
    - Except if made `FD_CLOEXEC` with `fcntl()`
  - If the file has an SUID (resp. SGID) bit, set the *effective* UID (resp. GID) of the process to the file's *owner* (resp. group)
  - Return −**1** on error

System Call: execve() and variants

Execute a Program

```
#include <unistd.h>

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

Error Conditions

- Typical errno codes
  - EACCES: execute permission denied (among other explanations)
  - ENOEXEC: non-executable format, or executable file for the wrong OS or processor architecture

System Call: execve() and variants

Execute a Program: Variants

```
#include <unistd.h>

int execl(const char *path, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execlp(const char *file, const char *arg, ...);
int execvp(const char *file, char *const argv[]);
int execle(const char *path, const char * arg, ..., char *const envp[]);
int execve(const char *filename, char *const argv[], char *const envp[]);
```

Arguments

- execl() operates on NULL-terminated argument list
  Warning: arg, the *first argument* after the pathname/filename corresponds to argv[0] (the program name)
- execv() operates on argument array
- execlp() and execvp() are $PATH-relative variants (if file does not contain a '/' character)
- execle() also provides an environment

5. Processes and Memory Management – Programmer Interface

System Call: execve() and variants

Execute a Program: Variants

```
#include <unistd.h>

int execl(const char *path, const char *arg, ...);
int execv(const char *path, char *const argv[]);
int execlp(const char *file, const char *arg, ...);
int execvp(const char *file, char *const argv[]);
int execle(const char *path, const char * arg, ..., char *const envp[]);
int execve(const char *filename, char *const argv[], char *const envp[]);
```

Environment

- Note about environment variables
  - They may be manipulated through getenv() and setenv()
  - To retrieve the whole array, declare the global variable
    extern char **environ;
    and use it as argument of execve() or execle()
  - More information: $ man 7 environ

117 / 352

I/O System Call: fcntl()

Manipulate a File Descriptor

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

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
```

Some More Commands

F_GETFD: get the file descriptor flags

F_SETFD: set the file descriptor flags to the value of arg
Only FD_CLOEXEC is defined: sets the file descriptor to be closed upon calls to execve() (typically a security measure)

I/O System Call: fcntl()

Manipulate a File Descriptor

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

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, long arg);
```


Return Value

- On success, fcntl() returns a (non-negative) value which depends on the command

  F_GETFD: the descriptor's flags
  F_GETFD: **0**
- Return **−1** on error

System Call: _exit()

Terminate the Current Process

```
#include <unistd.h>

void _exit(int status);
```

Purpose

- Terminates the calling process
  - Closes any open file descriptor
  - Frees all memory pages of the process address space (except shared ones)
  - Any child processes are inherited by process **1** (init)
  - The parent process is sent a SIGCHLD signal (ignored by default)
  - If the process is a *session leader* and its *controlling terminal* also controls the session, disassociate the terminal from the session and send a SIGHUP signal to all processes in the *foreground group* (terminate process by default)
- The call never fails and *does not return*!

System Call: `_exit()`

Terminate the Current Process

```
#include <unistd.h>

void _exit(int status);
```

Exit Code

- The *exit code* is a *signed byte* defined as `(status & 0xff)`
- **0** means normal termination, non-zero indicates an error/warning
- There is no standard list of exit codes
- It is collected with one of the `wait()` system calls

System Call: `_exit()`

C Library Front-End: `exit()`

```
#include <stdlib.h>

void exit(int status);
```

- Calls any function registered through `atexit()` (in reverse order of registration)
- Use this function rather than the low-level `_exit()` system call

5. Processes and Memory Management

Bootstrap and Processes Genealogy

Swapper Process

Process 0

- *One per CPU* (if multiprocessor)
- Built from scratch by the kernel and runs in kernel mode
- Uses *statically*-allocated data
- Constructs memory structures and initializes virtual memory
- Initializes the main kernel data structures
- Creates kernel threads (swap, kernel logging, etc.)
- Enables interrupts, and creates a kernel thread with $\mathrm{PID} = \mathbf{1}$

Bootstrap and Processes Genealogy

Init Process

Process 1

- *One per machine* (if multiprocessor)
- Shares all its data with process **0**
- Completes the initalization of the kernel
- Switch to user mode
- Executes `/sbin/init`, becoming a regular process and burying the structures and address space of process **0**

Executing `/sbin/init`

- Builds the OS environment
  - From `/etc/inittab`: type of bootstrap sequence, control terminals
  - From `/etc/rc*.d`: scripts to run system *daemon*s
- Adopts all orphaned processes, continuously, until the system halts
- `$ man init` and `$ man shutdown`

Process Tree

Simplified Tree From `$ pstree | more`

```
init-cron
   |-dhclient3
   |-gdm---gdm-+-Xorg
   |           `-x-session-manag---ssh-agent
   |-5*[getty]
   |-gnome-terminal-+-bash-+-more
   |                |      `-pstree
   |                |-gnome-pty-helper
   |                `-{gnome-terminal}
   |-klogd
   |-ksoftirqd
   |-kthread-+-ata
   |         |-2*[kjournald]
   |         `-kswapd
   |-syslogd
   `-udevd
```

5. Processes and Memory Management

Example: Network Service Daemons

Internet "Super-Server"

- `inetd`, initiated at boot time
- Listen on specific ports — listed in `/etc/services`
  - Each configuration line follows the format:
    `service_name port/protocol [aliases ...]`
    E.g., `ftp 21/tcp`
- Dispatch the work to predefined daemons — see `/etc/inetd.conf` — when receiving incomming connections on those ports
  - Each configuration line follows the format:
    `service_name socket_type protocol flags user_name daemon_path arguments`
    E.g., `ftp stream tcp nowait root /usr/bin/ftpd`

Process Sessions and Groups

Process Sessions

- Orthogonal to process hierarchy
- Session ID = PID of the leader of the session
- Typically associated to user *login*, interactive *terminals*, *daemon* processes
- The *session leader* sends the `SIGHUP` (*hang up*) signal to every process belonging to its session, and only if it belongs to the *foreground* group associated to the *controlling terminal* of the session

Process Groups

- Orthogonal to process hierarchy
- Process Group ID = PID of the group leader
- General mechanism
  - To distribute signals among processes upon global events (like `SIGHUP`)
  - Interaction with terminals, e.g., stall background process writing to terminal
  - To implement *job control* in shells
    `$ program &`, **Ctrl-Z**, `fg`, `bg`, `jobs`, `%1`, `disown`, etc.

System Call: setsid()

Creating a New Session and Process Group

```
#include <unistd.h>

pid_t setsid();
```

Description

- If the calling process is not a process group leader
  - Calling process is the leader and only process of a new group and session
  - Process group ID and session ID of the calling process are set to the PID of the calling process
  - Calling process has no controlling terminal any more
  - Return the session ID of the calling process (its PID)
- If the calling process is a process group leader
  - Return -1 and sets `errno` to `EPERM`
  - Rationale: a process group leader cannot "resign" its responsibilities

System Call: setsid()

Creating a Daemon (or Service) Process

- A *daemon process* is detached from any terminal, session or process group, is adopted by `init`, has no open standard input/output/error, has / for current directory
- "Daemonization" procedure
  1. Call `signal(SIGHUP, SIG_IGN)` to ignore `HUP` signal (see signals chapter)
  2. Call `fork()` in a process **P**
  3. Terminate parent **P**, calling `exit()` (may send `HUP` to child if session leader)
  4. Call `setsid()` in child **C**
  5. Call `signal(SIGHUP, SIG_DFL)` to reset `HUP` handler (see signals chapter)
  6. Change current directory, close descriptors 0, 1, 2, reset `umask`, etc.
  7. Continue execution in child **C**
- Note: an alternative procedure with a double `fork()` and `wait()` in the grand-parent is possible, avoiding to ignore the `HUP` signal

System Call: setsid()

Creating a Daemon (or Service) Process

- A *daemon process* is detached from any terminal, session or process group, is adopted by `init`, has no open standard input/output/error, has / for current directory
- "Daemonization" procedure
  1. Call `signal(SIGHUP, SIG_IGN)` to ignore `HUP` signal (see signals chapter)
  2. Call `fork()` in a process **P**
  3. Terminate parent **P**, calling `exit()` (may send `HUP` to child if session leader)
  4. Call `setsid()` in child **C**
  5. Call `signal(SIGHUP, SIG_DFL)` to reset `HUP` handler (see signals chapter)
  6. Change current directory, close descriptors 0, 1, 2, reset `umask`, etc.
  7. Continue execution in child **C**
- Note: an alternative procedure with a double `fork()` and `wait()` in the grand-parent is possible, avoiding to ignore the `HUP` signal

See, `getsid()`, `tcgetsid()`, `setpgid()`, etc.
See also `daemon()`, not POSIX but convenient integrated solution

6. Process Event Flow

- Monitoring Processes
- Signals
- Typical Applications
- Advanced Synchronization With Signals

Motivating Example

Shell Job Control

Monitoring stop/resume cycles of a child process

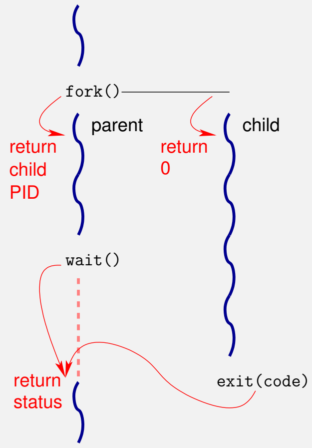
```
$ sleep 60
Ctrl-Z                      // Deliver SIGTSTP

// Shell notified of a state change in a child process (stopped)
[1]+  Stopped       sleep   // Recieved terminal stop signal
$ kill -CONT %1             // Equivalent to fg
sleep                       // Resume process
Ctrl-C                      // Deliver SIGINT
                            // Terminate process calling _exit(0)

// Shell notified of a state change in a child process (exited)
$
```

How does this work?
Signal: most primitive form of communication (presence/absence)

6. Process Event Flow

- Monitoring Processes
- Signals
- Typical Applications
- Advanced Synchronization With Signals

Monitoring Processes



System Call: wait() and waitpid()

Wait For Child Process to Change State

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

pid_t wait(int *status_pointer);
pid_t waitpid(pid_t pid, int *status_pointer, int options);
```

Description

- Monitor state changes and return PID of
  - Terminated child
  - Child stopped or resumed by a signal
- If a child terminates, it remains in a *zombie* state until wait() is performed to retrieve its state (and free the associated process descriptor)
  - Zombie processes do not have children: they are adopted by init process (**1**)
  - The init process always waits for its children
  - Hence, a zombie is removed when its parent terminates

System Call: wait() and waitpid()

Wait For Child Process to Change State

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

pid_t wait(int *status_pointer);
pid_t waitpid(pid_t pid, int *status_pointer, int options);
```

Whom to Wait For

- $\texttt{pid} > 0$: waitpid() suspends process execution until child specified by pid changes state, or returns immediately if it already did
- $\texttt{pid} = 0$: wait for any child in the same process group
- $\texttt{pid} < -1$: wait for any child in process group -pid
- $\texttt{pid} = -1$: wait for any child process

Short Cut

wait(&status) is equivalent to waitpid(-1, &status, 0)

System Call: wait() and waitpid()

Wait For Child Process to Change State

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

pid_t wait(int *status_pointer);
pid_t waitpid(pid_t pid, int *status_pointer, int options);
```

How to Wait

- Option WNOHANG: do not block if no child changed state
  Return **0** in this case
- Option WUNTRACED: report stopped child
  (due to SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU signals)
- Option WCONTINUED: report resumed child
  (due to SIGCONT signal)

System Call: wait() and waitpid()

Wait For Child Process to Change State

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

pid_t wait(int *status_pointer);
pid_t waitpid(pid_t pid, int *status_pointer, int options);
```


State Change Status

- If non-NULL status_pointer, store information into the int it points to
  WIFEXITED(status): true if child terminated normally (i.e., _exit())
  WEXITSTATUS(status): if the former is true, child exit status
  (lower **8** bits of status)
  WIFSIGNALED(status): true if child terminated by signal
  WTERMSIG(status): if the former is true, signal that caused termination
  WIFSTOPPED(status): true if child stopped by signal
  WSTOPSIG(status): if the former is true, signal that caused it to stop
  WIFCONTINUED(status): true if child was resumed by delivery of SIGCONT

System Call: wait() and waitpid()

Wait For Child Process to Change State

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

pid_t wait(int *status_pointer);
pid_t waitpid(pid_t pid, int *status_pointer, int options);
```


Error Conditions

- Return -1 if an error occurred
- Typical error code
  ECHILD, **calling** wait(): if all children were configured to be *unattended*
  (a.k.a. *un-waited for*, i.e., not becoming zombie when terminating, *see* sigaction())
  ECHILD, **calling** waitpid(): pid is not a child or is *unattended*

Process State Changes and Signals

Process State Monitoring Example

```
int status;
pid_t cpid = fork();
if (cpid == -1) { perror("fork"); exit(1); }
if (cpid == 0) {                  // Code executed by child
  printf("Child PID is %ld\n", (long)getpid());
  pause();                        // Wait for signals
} else                            // Code executed by parent
  do {
    pid_t w = waitpid(cpid, &status, WUNTRACED | WCONTINUED);
    if (w == -1) { perror("waitpid"); exit(1); }
    if (WIFEXITED(status))        // Control never reaches this point
      printf("exited, status=%d\n", WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
      printf("killed by signal %d\n", WTERMSIG(status));
    else if (WIFSTOPPED(status))
      printf("stopped by signal %d\n", WSTOPSIG(status));
    else if (WIFCONTINUED(status)) printf("continued\n");
  } while (!WIFEXITED(status) && !WIFSIGNALED(status));
```


Process State Changes and Signals

Running the Process State Monitoring Example

```
$ ./a.out &
Child PID is 32360
[1] 32359
$ kill -STOP 32360
stopped by signal 19
$ kill -CONT 32360
continued
$ kill -TERM 32360
killed by signal 15
[1]+  Done                    ./a.out
$
```


6. Process Event Flow

Process Synchronization With Signals

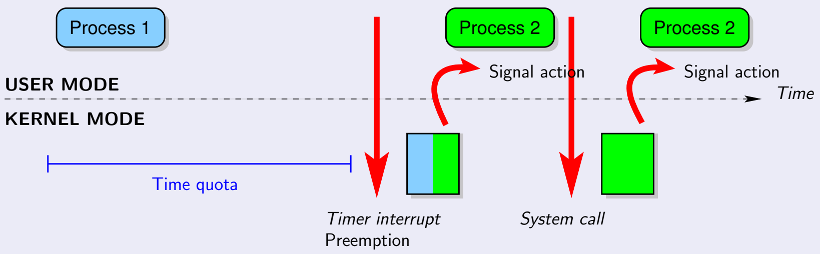
Principles

- Signal *delivery* is *asynchronous*
  - Both sending and recieving are asynchronous
  - Sending may occur during the signaled process execution or not
  - Recieving a signal may interrupt process execution at an arbitrary point
- A signal *handler* may be called upon signal delivery
  - It runs in *user mode* (sharing the user mode stack)
  - It is called "*catching the signal*"
- A signal is *pending* if it has been delivered but not yet handled
  - Because it is currently *blocked*
  - Or because the kernel did not yet check for its delivery status
- *No queueing* of pending signals

Process Synchronization With Signals

Catching Signals

- Signal caught when the process *switches from kernel to user mode*
  - Upon context switch
  - Upon return from system call



System Call: `kill()`

Send a Signal to a Process or Probe for a Process

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

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

Whom to Deliver the Signal

- $\texttt{pid} > 0$: to `pid`
- $\texttt{pid} = 0$: to all processes in the group of the current process
- $\texttt{pid} < -1$: to all processes in group $-\texttt{pid}$
- $\texttt{pid} = -1$: to all processes the current process has permitssion to send signals to, except himself and `init` (**1**)

System Call: kill()

Send a Signal to a Process or Probe for a Process

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

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

Existence and Permission

- No signal sent if sig is **0**, but error checks are performed
- UID or EUID of the sender must match the UID or EUID of the reciever

Error Conditions

- Return **0** on success, **−1** on error
- Possible errno codes
  - EINVAL: an invalid signal was specified
  - EPERM: no permission to send signal to any of the target processes
  - ESRCH: the process or process group does not exist

List of The Main Signals

SIGHUP[0]: terminal hang up
SIGINT[0]: keyboard interrupt (**Ctrl-C**)
SIGQUIT[0,1]: keyboard quit (**Ctrl-**\)
SIGKILL[0,3]: unblockable kill signal, terminate the process
SIGBUS/SIGSEGV[0,1]: memory bus error / segmentation violation
SIGPIPE[0]: broken pipe (writing to a pipe with no reader)
SIGALRM[0]: alarm signal
SIGTERM[0]: termination signal (kill command default)
SIGSTOP[3,4]: suspend process execution,
SIGTSTP[4]: terminal suspend (**Ctrl-Z**)
SIGTTIN/SIGTTOU[4]: terminal input/output for background process
SIGCONT[2]: resume after (any) suspend
SIGCHLD[2]: child stopped or terminated
SIGUSR1/SIGUSR2[0]: user defined signal 1/2

More signals: `$ man 7 signal`

[0] terminate process
[1] dump a core
[2] ignored by default
[3] non-maskable, non-catchable
[4] suspend process

System Call: signal()

ISO C Signal Handling (pseudo UNIX V7)

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
// Alternate, "all-in-one" prototype
void (*signal(int signum, void (*handler)(int)))(int);
```

Description

- Install a new handler for signal signum
  - SIG_DFL: default action
  - SIG_IGN: signal is ignored
  - Custom handler: function pointer of type sighandler_t
- Return the previous handler or SIG_ERR
- Warning: deprecated in multi-threaded or real-time code compiled with -pthread or linked with -lrt
  Some of the labs need threads, use sigaction

System Call: signal()

ISO C Signal Handling (pseudo UNIX V7)

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
// Alternate, "all-in-one" prototype
void (*signal(int signum, void (*handler)(int)))(int);
```

When Executing the Signal Handler

- The signum argument is the caught signal number
- Blocks (defers) nested delivery of the signal being caught
- Asynchronous execution w.r.t. the process's main program flow
  - Careful access to global variables (much like threads)
  - Limited opportunities for system calls
    Explicit list of "safe" functions: `$ man 2 signal`

System Call: pause()

Wait For Signal

```
#include <unistd.h>

int pause();
```

Description

- Suspends the process until it is delivered a signal
  - That terminate the process (pause() does not return...)
  - That causes a signal handler to be called
- Ignored signals (SIG_IGN) do *not* resume execution
  In fact, they never interrupt any system call
- Always return -1 with error code EINTR

6. Process Event Flow

System Call: alarm()

Set an Alarm Clock for Delivery of a SIGALRM

```
#include <unistd.h>

int alarm(unsigned int seconds);
```

Description

- Deliver SIGALRM to the calling process after a delay (non-guaranteed to react immediately)
- Warning: the default action is to terminate the process

System Call: alarm()

C library function: sleep

```
unsigned int sleep(unsigned int seconds)
```

- Combines signal(), alarm() and pause()
- Uses the same timer as alarm() (hence, do not mix)
- See also setitimer()

Putting the Process to Sleep

```
void do_nothing(int signum)
{
  return;
}
void my_sleep(unsigned int seconds)
{
  signal(SIGALRM, do_nothing); // Note: SIG_IGN would block for ever!
  alarm(seconds);
  pause();
  signal(SIGALRM, SIG_DFL);    // Restore default action
}
```


More Complex Event Flow Example

Shell Job Control

Monitoring stop/resume cycles of a child process

```
$ top
Ctrl-Z                        // Deliver SIGTSTP

[1]+  Stopped          top   // Stop process
$ kill -CONT %1               // Resume (equivalent to fg)
                              // Recieve SIGTTOU and stop
[1]+  Stopped          top   // Because of background terminal I/O
$ kill -INT %1
                              // SIGINT is pending, i.e.
[1]+  Stopped          top   //   did not trigger an action yet
$ fg
top
                              // Terminate process calling exit(0)
$
```

6. Process Event Flow

- Monitoring Processes
- Signals
- Typical Applications
- Advanced Synchronization With Signals

Advanced Synchronization With Signals

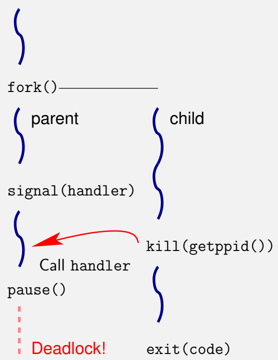
Determinism and Atomicity

- ISO C (pseudo UNIX V7) signals are error-prone and may lead to uncontrollable run-time behavior: historical *design flaw*
  - Example: install a signal handler (`signal()`) before suspension (`pause()`)
  - What happens if the signal is delivered in between?

Advanced Synchronization With Signals

Determinism and Atomicity

- ISO C (pseudo UNIX V7) signals are error-prone and may lead to uncontrollable run-time behavior: historical *design flaw*
  - Example: install a signal handler (`signal()`) before suspension (`pause()`)
  - What happens if the signal is delivered in between?



Advanced Synchronization With Signals

Determinism and Atomicity

- ISO C (pseudo UNIX V7) signals are error-prone and may lead to uncontrollable run-time behavior: historical *design flaw*
  - Example: install a signal handler (`signal()`) before suspension (`pause()`)
  - What happens if the signal is delivered in between?
    Asynchronous signal delivery
    → Possible deadlock
    → Hard to fix the bug

6. Process Event Flow – Advanced Synchronization With Signals

Advanced Synchronization With Signals

Determinism and Atomicity

- ISO C (pseudo UNIX V7) signals are error-prone and may lead to uncontrollable run-time behavior: historical *design flaw*
  - Example: install a signal handler (`signal()`) before suspension (`pause()`)
  - What happens if the signal is delivered in between?
    Asynchronous signal delivery
    → Possible deadlock
    → Hard to fix the bug
- Solution: atomic (un)masking (a.k.a. (un)blocking) and suspension

6. Process Event Flow – Advanced Synchronization With Signals

Advanced Synchronization With Signals

Determinism and Atomicity

- ISO C (pseudo UNIX V7) signals are error-prone and may lead to uncontrollable run-time behavior: historical *design flaw*
  - Example: install a signal handler (`signal()`) before suspension (`pause()`)
  - What happens if the signal is delivered in between?
    Asynchronous signal delivery
    → Possible deadlock
    → Hard to fix the bug
- Solution: atomic (un)masking (a.k.a. (un)blocking) and suspension
- Lessons learned
  - Difficult to tame low-level concurrency mechanisms
  - Look for *deterministic* synchronization/communication primitives (enforce functional semantics)

System Call: sigaction()

POSIX Signal Handling

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
                          struct sigaction *oldact);
```

Description

- Examine and change the action taken by a process on signal delivery
- If act is not NULL, it is the new action for signal signum
- If oldact is not NULL, store the current action into the struct sigaction pointed to by oldact
- Return **0** on success, **−1** on error

System Call: sigaction()

POSIX Signal Handling

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
                          struct sigaction *oldact);
```


Error Conditions

- Typical error code

  EINVAL: an invalid signal was specified, or attempting to change the action for SIGKILL or SIGSTOP
  Calling sigaction() with NULL second and third arguments and checking for the EINVAL error allows to check whether a given signal is supported on a given platform

System Call: sigaction()

POSIX Signal Action Structure

```
struct sigaction {
  void (*sa_handler)(int);
  void (*sa_sigaction)(int, siginfo_t*, void*);
  sigset_t sa_mask;
  int sa_flags;
}
```

Description

sa_handler: same function pointer as the argument of signal()
(it may also be set to SIG_DFL or SIG_IGN)

sa_sigaction: handler with information about the context of signal delivery
(excusive with sa_handler, should not even be initialized if sa_handler is set)

sa_mask: mask of blocked signals when executing the signal handler

sa_flags: bitwise or of handler behavior options

System Call: sigaction()

POSIX Signal Action Structure

```
struct sigaction {
  void (*sa_handler)(int);
  void (*sa_sigaction)(int, siginfo_t*, void*);
  sigset_t sa_mask;
  int sa_flags;
}
```

SA_NOCLDSTOP: if signum is SIGCHLD, no notification when child processes stop (SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU) or resume (SIGCONT)

SA_NOCLDWAIT: if signum is SIGCHLD, "leave children *unattended*", i.e., do not transform terminating children processes into zombies

SA_SIGINFO: use sa_sigaction field *instead* of sa_handler

- siginfo_t parameter carries signal delivery context
- $ man 2 sigaction for (lengthy) details

A few others: reset handler after action, restart interrupted system call, etc.

The sigsetops Family of Signal-Set Operations

```
$ man 3 sigsetops

#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

Description

- Respectively: empty set, full set, add, remove, and test whether a signal belong to the sigset_t pointed to by set
- The first four return **0** on success and −**1** on error
- sigismember() returns **1** if signum is in the set, **0** if not, and −**1** on error
- See also the non-portable sigisemptyset(), sigorset(), sigandset()

Simple sigaction Example

```
int count_signal = 0;

void count(int signum) {
  count_signal++;
}

// ...

{
  struct sigaction sa;

  sa.sa_handler = count;                // Signal handler
  sigemptyset(&sa.sa_mask);             // Pass field address directly
  sa.sa_flags = 0;
  sigaction(SIGUSR1, &sa, NULL);

  while (true) {
    printf("count_signal = %d\n", count_signal);
    pause();
  }
}
```

System Call: sigprocmask()

Examine and Change Blocked Signals

```
#include <signal.h>

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

Semantics

- If `set` is not `NULL`, `how` describes the behavior of the call
  - `SIG_BLOCK`: blocked $\longleftarrow$ blocked $\cup$ `*set`
  - `SIG_UNBLOCK`: blocked $\longleftarrow$ blocked $-$ `*set`
  - `SIG_SETMASK`: blocked $\longleftarrow$ `*set`
- If `oldset` is not `NULL`, store the current mask of blocked signals into the `sigset_t` pointed to by `oldset`
- Return **0** on success, $-\mathbf{1}$ on error

6. Process Event Flow – Advanced Synchronization With Signals

System Call: sigprocmask()

Examine and Change Blocked Signals

```
#include <signal.h>

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


Remarks

- Unblockable signals: `SIGKILL`, `SIGSTOP`
  (attempts to mask them are silently ignored)
- Use `sigsuspend()` to unmask signals before suspending execution

System Call: sigpending()

Examine Pending Signals

```
#include <signal.h>

int sigpending(sigset_t *set);
```

Semantics

- A signal is *pending* if it has been delivered but not yet handled, because it is currently blocked
  (or because the kernel did not yet check for its delivery status)
- Stores the set of pending signals into the `sigset_t` pointed to by `set`
- Return **0** on success, $-\mathbf{1}$ on error

System Call: sigsuspend()

Wait For a Signal

```
#include <signal.h>

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

Semantics

- Perform the two following operations *atomically* w.r.t. signal delivery
  1. Set mask as the temporary set of masked signals
  2. Suspend the process until delivery of an *unmasked*, *non-ignored* signal
- When recieving a non-terminating, non-ignored signal, execute its handler, *and then*, *atomically* restore the previous set of masked signals and resume execution
- Always return -1, typically with error code EINTR

6. Process Event Flow – Advanced Synchronization With Signals

System Call: sigsuspend()

Wait For a Signal

```
#include <signal.h>

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

Typical Usage

- *Prevent early signal delivery between unmasking and suspension*
  1. Call sigprocmask() to disable a set of signals
  2. Perform some critical operation
  3. Call sigsuspend() to atomically enable some of them and suspend execution
- Without this atomic operation (i.e., with signal() and pause())
  1. A signal may be delivered *between* the installation of the signal handler (the call to signal()) and the suspension (the call to pause())
  2. Its handler (installed by signal()) may be triggered *before the suspension* (the call to pause())
  3. Handler execution *clears the signal* from the process's pending set
  4. The suspended process deadlocks, waiting for an already-delivered signal

Example With Signals and Memory Management

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

struct sigaction sa;
char *p;

void catch(int signum) {       // Catch a segmentation violation
  static int *save_p = NULL;
  if (save_p == NULL) { save_p = p; brk(p+1); }
  else { printf("Page size: %d\n", p - save_p); exit(0); }
}

int main(int argc, char *argv[]) {
  sa.sa_handler = catch; sigemptyset(&sa.sa_mask); sa.sa_flags = 0;
  sigaction(SIGSEGV, &sa, NULL);
  p = (char*)sbrk(0);
  while (1) *p++ = 42;
}

$ page
Page size: 4096
```

Command-Line Operations on Processes

- Cloning and executing
  `$ program arguments &`
- Joining (waiting for completion)
  `$ wait [PID]`
- Signaling events
  `$ kill [-signal] PID`
  `$ killall [-signal] process_name`
  Default signal `TERM` terminates the process
- `$ nohup`: run a command immune to *hang-up* (`HUP` signal)

7. Communication and Synchronization

- Message Queues
- Advanced Memory Management
- Shared Memory Segments

7. Communication and Synchronization

- Message Queues
- Advanced Memory Management
- Shared Memory Segments

IPC: Message Queues

Queueing Mechanism for Structured Messages

- Signals
  - Carry no information beyond their own delivery
  - Cannot be queued
- FIFOs (pipes)
  - Unstructured stream of data
  - No priority mechanism
- Message queues offer a loss-less, *structured*, *priority-driven* communication channel between processes
  `$ man 7 mq_overview`

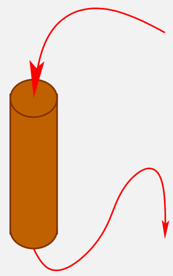
Implementation in Linux

- Message queue files are single inodes located in a specific *pseudo-file-system*, mounted under `/dev/mqueue`
- Must link the program with `-lrt` (real-time library)

Structured Communication

Priority, Structured Queues

- Maintain message boundary
- Sort messages by priority



```
mq_send(mqdes, " World!", 7, 20);
mq_send(mqdes, "Hello", 5, 31);

mq_getattr(mqdes, &mq_attr);
msg_len = mq_attr.mq_msgsize;
s = mq_receive(mqdes, buf, msg_len, NULL);
mq_receive(mqdes, buf+s, msg_len, NULL);
```

Hello

Hello World!

System Call: `mq_open()`

Open and Possibly Create a POSIX Message Queue

```
#include <mqueue.h>

mqd_t mq_open(const char *name, int flags);
mqd_t mq_open(const char *name, int flags, mode_t mode,
              struct mq_attr *attr);
```

Description

- Analogous to `open()`, but not mapped to persistent storage
  - `name`: must begin with a "`/`" and may not contain any other "`/`"
  - `flags`: only `O_RDONLY`, `O_RDWR`, `O_CREAT`, `O_EXCL`, `O_NONBLOCK`; and `FD_CLOEXEC` flag is set automatically
  - `mode`: `S_IRUSR`, `S_IWUSR`, `S_IXUSR`, etc.
  - `attr`: attributes for the queue, see `mq_getattr()`
    Default set of attributes if `NULL` or not specified
- Return a message queue descriptor on success, -1 on error

System Call: mq_getattr() and mq_setattr()

Attributes of a POSIX Message Queue

```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *mq_attr)
int mq_setattr(mqd_t mqdes, struct mq_attr *mq_newattr,
               struct mq_attr *mq_oldattr)
```

Description

- The mq_attr structure is defined as

```
struct mq_attr {
  long mq_flags;   // Flags: 0 or O_NONBLOCK
  long mq_maxmsg;  // Maximum # of pending messages (constant)
  long mq_msgsize; // Maximum message size (bytes, constant)
  long mq_curmsgs; // # of messages currently in queue
};
```

- Return **0** on success, **−1** on error

System Call: mq_send()

Send a Message To a POSIX Message Queue

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, char *msg_ptr,
            size_t msg_len, unsigned int msg_prio)
```

Description

- Enqueues the message pointed to by msg_ptr of size msg_len into mqdes
- msg_len must be less than or equal to the mq_msgsize attribute of the queue (see mq_getattr())
- msg_prio is a non-negative integer specifying message priority
  0 is the lowest priority, and **31** is the highest (portable) priority
- By default, mq_send() blocks when the queue is full (i.e., mq_maxmsg currently in queue)
- Return **0** on success, **−1** on error

System Call: mq_receive()

Receive a Message From a POSIX Message Queue

```
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
                   size_t msg_len, unsigned int *msg_prio)
```

Description

- Removes the oldest message with the highest priority from mqdes
- Stores it into the buffer pointed to by msg_ptr of size msg_len
- msg_len must be greater than or equal to the mq_msgsize attribute of the queue (see mq_getattr())
- If msg_prio is not null, use it to store the priority of the received message
- By default, mq_receive() blocks when the queue is empty
- Return the number of bytes of the received message on success, **−1** on error

System Call: mq_close()

Close a POSIX Message Queue Descriptor

```
#include <mqueue.h>

int mq_close(mqd_t mqdes);
```

Description

- Also remove any notification request attached by the calling process to this message queue
- Return **0** on success, **−1** on error

System Call: mq_unlink()

Unlink a POSIX Message Queue File

```
#include <mqueue.h>

int mq_close(const char *name);
```

Description

- Message queues have *kernel* persistence
- Similar to unlink()

Other System Calls

- mq_notify(): notify a process with a signal everytime the specified queue receives a message while originally empty
- mq_timedreceive() and mq_timedsend(): receive and send with timeout

7. Communication and Synchronization

Memory and I/O Mapping

Virtual Memory Pages

- *Map* virtual addresses to physical addresses
  - Configure MMU for page translation
  - Support growing/shrinking of virtual memory segments
  - Provide a protection mechanism for memory *pages*
- Implement copy-on-write mechanism (e.g., to support `fork()`)

Memory and I/O Mapping

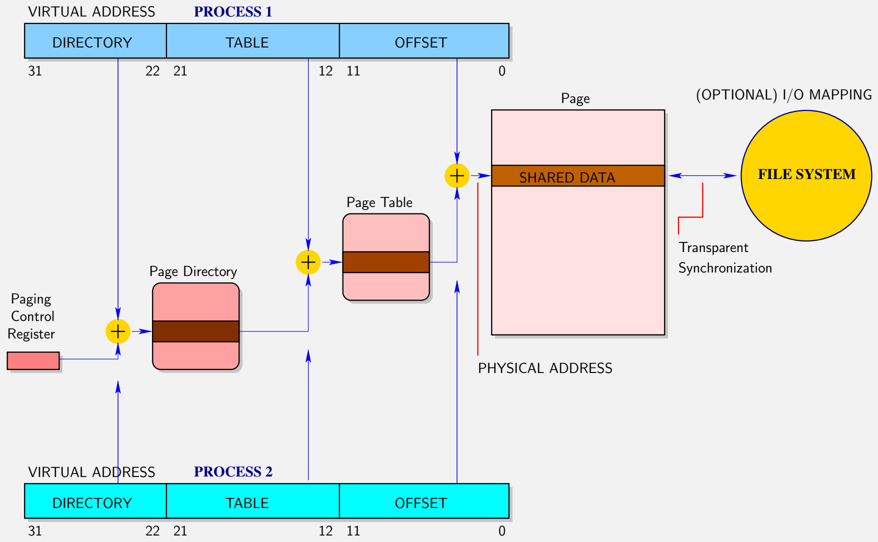
Virtual Memory Pages

- *Map* virtual addresses to physical addresses
  - Configure MMU for page translation
  - Support growing/shrinking of virtual memory segments
  - Provide a protection mechanism for memory *pages*
- Implement copy-on-write mechanism (e.g., to support `fork()`)

I/O to Memory

- *Map* I/O operations to simple memory load/store accesses
- Facilitate sharing of memory pages
  - Use file naming scheme to identify memory regions
  - Same system call to implement private and shared memory allocation

Memory and I/O Mapping



System Call: mmap()

Map Files or Devices Into Memory

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

void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
```

Semantics

- Allocate length bytes from the process virtual memory, starting at the start address or any fresh interval of memory if start is NULL
- Map to this memory interval the a file region specified by fd and starting position offset
- start address must be multiple of memory page size; almost always NULL in practice
- Return value
  - Start address of the mapped memory interval on success
  - MAP_FAILED on error (i.e., (void*)-1)

System Call: mmap()

Map Files or Devices Into Memory

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

void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
```


Memory Protection: the prot Argument

- It may be PROT_NONE: access forbiden
- Or it may be built by *or'ing* the following flags
  - **PROT_EXEC:** data in pages may be executed as code
  - **PROT_READ:** pages are readable
  - **PROT_WRITE:** pages are writable

System Call: mmap()

Map Files or Devices Into Memory

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

void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
```


Memory Protection: the flags Argument

- Either
  - **MAP_PRIVATE:** create a private, copy-on-write mapping; writes to the region do not affect the mapped file
  - **MAP_SHARED:** share this mapping with all other processes which map this file; writes to the region affect the mapped file
  - **MAP_ANONYMOUS:** mapping not associated to any file (fd and offset are ignored); underlying mechanism for growing/shrinking virtual memory segments (including stack management and malloc())

System Call: mmap()

Map Files or Devices Into Memory

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

void *mmap(void *start, size_t length, int prot, int flags,
           int fd, off_t offset);
```

Error Conditions

- EACCESS: fd refers to non-regular file or prot incompatible with opening mode or access rights
  Note: modes O_WRONLY, O_APPEND are forbidden
- ENOMEM: not enough memory

Error Signals

- SIGSEGV: violation of memory protection rights
- SIGBUS: access to memory region that does not correspond to a legal position in the mapped file

System Call: munmap()

Delete a Memory Mapping for a File or Device

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

int munmap(void *start, size_t length);
```

Semantics

- Delete the mappings for the specified address and range
- Further accesses will generate invalid memory references
- Remarks
  - start must be multiple of the page size (typically, an address returned by mmap() in the first place)
    Otherwise: generate SIGSEGV
  - All pages containing part of the specified range are unmapped
  - Any pending modification is synchronized to the file
    See msync()
  - Closing a file descriptor does not unmap the region
- Return **0** on success, **−1** on error

7. Communication and Synchronization

- Message Queues
- Advanced Memory Management
- Shared Memory Segments

IPC: Shared Memory Segments

Naming Shared Memory Mappings

- Question: how do processes *agree* on a *sharing* a *physical memory* region?

7. Communication and Synchronization – Shared Memory Segments

IPC: Shared Memory Segments

Naming Shared Memory Mappings

- Question: how do processes *agree* on a *sharing* a *physical memory* region?
  - *Sharing* is easy: call `mmap()` with `MAP_SHARED` flag
  - *Agreeing* is the problem

175 / 352

7. Communication and Synchronization – Shared Memory Segments

IPC: Shared Memory Segments

Naming Shared Memory Mappings

- Question: how do processes *agree* on a *sharing* a *physical memory* region?
  - *Sharing* is easy: call `mmap()` with `MAP_SHARED` flag
  - *Agreeing* is the problem
- Solution: use a *file name* as a meeting point

175 / 352

IPC: Shared Memory Segments

Naming Shared Memory Mappings

- Question: how do processes *agree* on a *sharing* a *physical memory* region?
  - *Sharing* is easy: call `mmap()` with `MAP_SHARED` flag
  - *Agreeing* is the problem
- Solution: use a *file name* as a meeting point
- Slight problem... one may not want to waste disk space for transient data (not persistent accross system shutdown)
  - `MAP_ANONYMOUS` solves this problem... but looses the association between the file and memory region to implement the rendez-vous

IPC: Shared Memory Segments

Naming Shared Memory Mappings

- Question: how do processes *agree* on a *sharing* a *physical memory* region?
  - *Sharing* is easy: call `mmap()` with `MAP_SHARED` flag
  - *Agreeing* is the problem
- Solution: use a *file name* as a meeting point
- Slight problem... one may not want to waste disk space for transient data (not persistent accross system shutdown)
  - `MAP_ANONYMOUS` solves this problem... but looses the association between the file and memory region to implement the rendez-vous

Implementation in Linux

- Shared memory files are single inodes located in a specific *pseudo-file-system*, mounted under `/dev/shm`
- Must link the program with `-lrt` (real-time library)

System Call: `shm_open()`

Open and Possibly Create a POSIX Shared Memory File

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

int shm_open(const char *name, int flags, mode_t mode);
```

Description

- Analogous to `open()`, but for files specialized into "shared memory rendez-vous", and not mapped to persistent storage

  `name`: must begin with a "/" and may not contain any other "/"

  `flags`: only `O_RDONLY`, `O_RDWR`, `O_CREAT`, `O_TRUNC`, `O_NONBLOCK`; and and `FD_CLOEXEC` flag is set automatically

  `mode`: `S_IRUSR`, `S_IWUSR`, `S_IXUSR`, etc.

System Call: shm_open()

Open and Possibly Create a POSIX Shared Memory File

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

int shm_open(const char *name, int flags, mode_t mode);
```

Allocating and Sizing a Shared Memory Segment

- The first mmap() on a shared memory descriptor allocates memory and maps it to virtual memory of the calling process
- Warning: the size of the allocated region is not yet stored in the descriptor
  - Need to *publish* this size through the file descriptor
  - Use a generic file-sizing system call

```
int ftruncate(int fd, off_t length);
```


System Call: shm_unlink()

Unlink a POSIX Shared Memory File

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

int shm_unlink(const char *name);
```

Description

- Shared memory files have *kernel* persistence
- Similar to unlink()
- close() works as usual to close the file descriptor after the memory mapping has been performed
- Neither close() nor unlink() impact shared memory mapping themselves

About Pointers in Shared Memory

Caveat of Virtual Memory

1. The value of a pointer is a *virtual memory address*

About Pointers in Shared Memory

Caveat of Virtual Memory

1. The value of a pointer is a *virtual memory address*
2. Virtual memory is *mapped differently* in every process
   - In general, a pointer in a shared memory segment does not hold a valid address for all processes mapping this segment

7. Communication and Synchronization – Shared Memory Segments

About Pointers in Shared Memory

Caveat of Virtual Memory

1. The value of a pointer is a virtual memory address
2. Virtual memory is mapped differently in every process
   - In general, a pointer in a shared memory segment does not hold a valid address for all processes mapping this segment

3. Big problem for *linked data structures* and function pointers

7. Communication and Synchronization – Shared Memory Segments

About Pointers in Shared Memory

Caveat of Virtual Memory

1. The value of a pointer is a virtual memory address
2. Virtual memory is mapped differently in every process
   - In general, a pointer in a shared memory segment does not hold a valid address for all processes mapping this segment
3. Big problem for linked data structures and function pointers

4. Mapping to a specified address is a fragile solution
   - The `start` argument of `mmap()`

About Pointers in Shared Memory

Caveat of Virtual Memory

1. The value of a pointer is a *virtual memory address*
2. Virtual memory is *mapped differently* in every process
   - In general, a pointer in a shared memory segment does not hold a valid address for all processes mapping this segment
3. Big problem for *linked data structures* and function pointers
4. Mapping to a specified address is a fragile solution
   - The `start` argument of `mmap()`
5. Pointers relative to the base address of the segment is another solution (cumbersome: requires extra pointer arithmetic)

7. Communication and Synchronization – Shared Memory Segments

About Pointers in Shared Memory

Caveat of Virtual Memory

1. The value of a pointer is a *virtual memory address*
2. Virtual memory is *mapped differently* in every process
   - In general, a pointer in a shared memory segment does not hold a valid address for all processes mapping this segment
3. Big problem for *linked data structures* and function pointers
4. Mapping to a specified address is a fragile solution
   - The `start` argument of `mmap()`
5. Pointers relative to the base address of the segment is another solution (cumbersome: requires extra pointer arithmetic)
6. Note: the problem disappears when forking *after* the shared memory segment has been mapped

8. Concurrency and Mutual Exclusion

Concurrent Resource Management

Concurrency Issues

- Multiple *non-modifying accesses* to *shared resources* may occur in parallel without conflict
- Problems arise when *accessing a shared resource* to *modify its state*
  - Concurrent file update
  - Concurrent shared memory update
- General problem: enforcing *mutual exclusion*

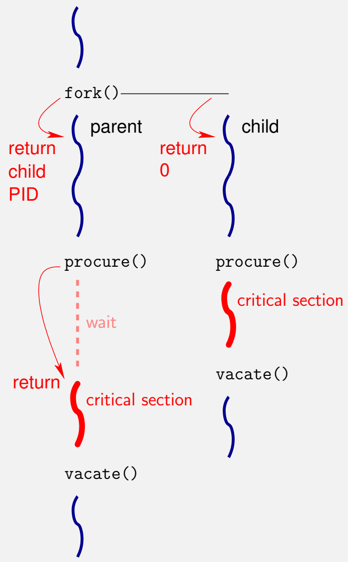
Principles of Concurrent Resource Management

Critical Section

- Program section accessing shared resource(s)
- *Only one process can be in this section at a time*

Mutual Exclusion

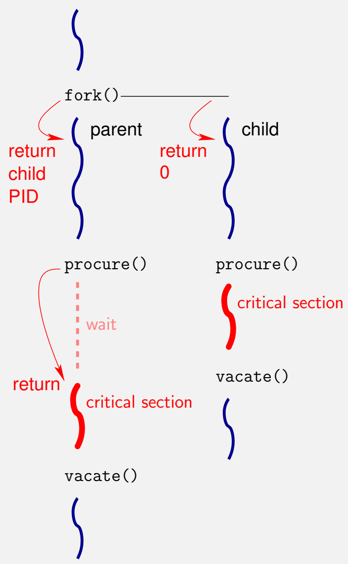
- Make sure at most one process may enter a critical section
- Typical cases
  - Implementing file locks
  - Concurrent accesses to shared memory



Principles of Concurrent Resource Management

Source of Major Headaches

- *Correctness*: prove process alone in critical section
- *Absence of deadlock*, or detection and lock-breaking
- *Guaranteed progress*: a process enters critical section if it is the only one to attempt to do it
- *Bounded waiting*: a process waiting to enter a critical section will eventually (better sooner than later) be authorized to do so
- *Performance*: reduce overhead and allow parallelism to scale



8. Concurrency and Mutual Exclusion

- Mutual Exclusion in Shared Memory
- Semaphores
- Mutual Exclusion and Deadlocks
- File Locks
- System V IPC

Mutual Exclusion in Shared Memory

Dekker's Algorithm

```
int try0 = 0, try1 = 0;
int turn = 0; // Or 1

// Fork processes sharing variables try0, try1, turn
// Process 0                               // Process 1
try0 = 1;                                  try1 = 1;
while (try1 != 0)                           while (try0 != 0)
  if (turn != 0) {                           if (turn != 1) {
    try0 = 0;                                  try1 = 0;
    while (try1 != 0) { }                      while (try0 != 0) { }
    try0 = 1;                                  try1 = 1;
  }                                          }
turn = 0;                                  turn = 1;
// Critical section                        // Critical section
try0 = 0;                                  try1 = 0;
// Non-critical section                    // Non-critical section
```


Mutual Exclusion in Shared Memory

Peterson's Algorithm

```
int try0 = 0, try1 = 0;
int turn = 0; // Or 1

// Fork processes sharing variables try0, try1, turn
// Process 0                               // Process 1
try0 = 1;                                  try1 = 1;
turn = 0;                                  turn = 1;
while (try1 && !turn) { }                  while (try0 && turn) { }
// Critical section                        // Critical section
try0 = 0;                                  try1 = 0;
// Non-critical section                    // Non-critical section
```

- Unlike Dekker's algorithm, enforces fair turn alternation
- Simpler and easily extensible to more than two processes

Shared Memory Consistency Models

Memory Consistency

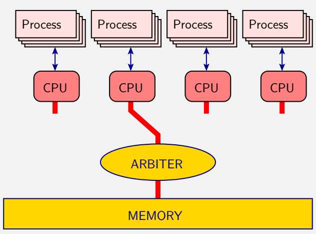
- "When the outcome of a memory access is visible from another process"
- Dekker and Petersen algorithms require the strongest memory model: *sequential consistency*
- Definition of sequential consistency by Leslie Lamport:
  "*The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*"

8. Concurrency and Mutual Exclusion – Mutual Exclusion in Shared Memory

Shared Memory Consistency Models

Memory Consistency

- "When the outcome of a memory access is visible from another process"
- Dekker and Petersen algorithms require the strongest memory model: *sequential consistency*
- Definition of sequential consistency by Leslie Lamport:
  "*The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*"



8. Concurrency and Mutual Exclusion – Mutual Exclusion in Shared Memory

Shared Memory Consistency Models

Memory Consistency

- "When the outcome of a memory access is visible from another process"
- Dekker and Petersen algorithms require the strongest memory model: *sequential consistency*
- Definition of sequential consistency by Leslie Lamport:
  "*The result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program.*"

Weak Consistency Models

- Hardware, run-time libraries and compilers prefer *weaker consistency models*
  - Compiler optimizations: loop-invariant code motion, instruction reordering
  - Hardware/run-time optimizations: out-of-order superscalar execution (local), out-of-order cache coherence (multi-processor)
- Impossibility result for mutual exclusion: Attiya et al. POPL 2011

Memory Consistency Examples

Paradoxical Example

```
int f = 1;
int x = 0;

// Fork processes sharing variables f, x
// Process 0                                  // Process 1
while (f) { }                                 x = 1;
printf("x = %d\n", x);                        f = 0;
```

Analysis

- What is the value of x printed by Process 0?
  (assuming no other process may access the shared variables)

8. Concurrency and Mutual Exclusion – Mutual Exclusion in Shared Memory

Memory Consistency Examples

Paradoxical Example

```
int f = 1;
int x = 0;

// Fork processes sharing variables f, x
// Process 0                                  // Process 1
while (f) { }                                 x = 1;
printf("x = %d\n", x);                        f = 0;
```

Analysis

- What is the value of x printed by Process 0?
  (assuming no other process may access the shared variables)
  - **1** with sequential consistency

8. Concurrency and Mutual Exclusion – Mutual Exclusion in Shared Memory

Memory Consistency Examples

Paradoxical Example

```
int f = 1;
int x = 0;

// Fork processes sharing variables f, x
// Process 0                                  // Process 1
while (f) { }                                 x = 1;
printf("x = %d\n", x);                        f = 0;
```

Analysis

- What is the value of x printed by Process 0?
  (assuming no other process may access the shared variables)
  - **1** with sequential consistency
  - May be **0** with weaker models

Solution: Hardware Support

Serializing Memory Accesses

- Memory *fences* (for the hardware and compiler)
  - Multiprocessor
    → In general, commit all pending memory and cache coherence transactions
  - Uniprocessor (cheaper and weaker)
    → Commit all local memory accesses
  - Can be limited to read or write accesses
  - Depending on the memory architecture, cheaper implementations are possible
  - Forbids cross-fence code motion by the compiler
- ISO C `volatile` attribute (for the compiler)
  - `volatile int x`
    Informs the compiler that asynchronous modifications of `x` may occur
  - No compile-time reordering of accesses to volatile variables
  - Never consider accesses to volatile variables as dead code
- Combining fences and volatile variables fixes the problems of Dekker's and Peterson's algorithms
- Modern programming languages tend to merge both forms into more abstract constructs (e.g., Java 5)

Solution: Hardware Support

Atomic Operations

- Fine grain *atomic operations* permit higher performance than synchronization algorithms with fences
  - *Atomic Exchange*: exchange value of a register and a memory location, atomically
  - *Test-and-Set*: set a memory location to **1** and return whether the old value was null or not, atomically
    - Can be implemented with atomic exchange

```
int test_and_set(int *lock_pointer) {
  int old_value = 0;
  if (*lock_pointer)
    old_value = atomic_exchange(lock_pointer, 1);
  return old_value != 0;
}
```


Solution: Hardware Support

Atomic Operations

- Fine grain *atomic operations* permit higher performance than synchronization algorithms with fences
  - More powerful: *Compare-and-Swap* (cannot be implemented with the former)

```
bool compare_and_swap(int *accum, int *dest, int newval) {
  if (*accum == *dest) {
      *dest = newval;
      return true;
  } else {
      *accum = *dest;
      return false;
  }
```

  - Many others, implementable with atomic exchange or compare-and-swap, with or without additional control flow

8. Concurrency and Mutual Exclusion

- Mutual Exclusion in Shared Memory
- Semaphores
- Mutual Exclusion and Deadlocks
- File Locks
- System V IPC

From Simple Locks to Semaphores

```
void lock(volatile int *lock_pointer) {
  while (test_and_set(lock_pointer) == 1);
}
void unlock(volatile int *lock_pointer) {
  *lock_pointer = 0 // Release lock
}

int lock_variable = 1;
void lock_example() {
  lock(&lock_variable);
  // Critical section
  unlock(&lock_variable);
}
```

Generalization to Countable Resources: Semaphores

- Atomic increment/decrement primitives
  - **P()** or `procure()` from "*proberen*" (Dutch for "to wait")
  - **V()** or `vacate()` from "*verhogen*" (Dutch for "to increment")
- May use *simple lock* to implement atomicity

Semaphore

Unified Structure and Primitives for Mutual Exclusion

- Initialize the semaphore with `v` instances of the resource to manage

```
void init(semaphore s, int v) {
  s.value = v;
}
```

- Acquire a resource (entering a critical section)

```
void procure(semaphore s) {
  wait until (s.value > 0);
  s.value--;     // Must be atomic with the previous test
}
```

  Also called `down()` or `wait()`
- Release a resource (leaving a critical section)

```
void vacate(semaphore s) {
  s.value++;     // Must be atomic
}
```

  Also called `up()`, `post()` or `signal()`

Heterogeneous Read-Write Mutual Exclusion

Read-Write Semaphores

- Allowing *multiple readers* and a *single writer*

```
void init(rw_semaphore l) {
  l.value = 0;   // Number of readers (resp. writers)
                 //   if positive (resp. negative)
}
void procure_read(rw_semaphore l) {
  wait until (l.value >= 0);
  l.value++;     // Must be atomic with the previous test
}
void vacate_read(rw_semaphore l) {
  l.value--;     // Must be atomic
}
void procure_write(rw_semaphore l) {
  wait until (l.value == 0);
  l.value = -1;  // Must be atomic with the previous test
}
void vacate_write(rw_semaphore l) {
  l.value = 0;
}
```


IPC: Semaphores

POSIX Semaphores

- Primitives: `sem_wait()` (`procure()`) and `sem_post()` (`vacate()`)
  - `sem_wait()` blocks until the value of the semaphore is greater than **0**, then decrements it and returns
  - `sem_post()` increments the value of the semaphore and returns
- They can be named (associated to a file) or not
- `$ man 7 sem_overview`

Implementation in Linux

- Semaphore files are single inodes located in a specific *pseudo-file-system*, mounted under `/dev/shm`
- Must link the program with `-lrt` (real-time library)

System Call: `sem_open()`

Open and Possibly Create a POSIX Semaphore

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int flags);
sem_t *sem_open(const char *name, int flags,
                mode_t mode, unsigned int value);
```

Description

- Arguments `flags` and `mode` allow for a subset of their values for `open()`
  - `flags`: only `O_CREAT`, `O_EXCL`; and `FD_CLOEXEC` flag is set automatically
  - `mode`: `S_IRUSR`, `S_IWUSR`, `S_IXUSR`, etc.
- `value` is used to initialize the semaphore, defaults to **1** if not specified
- Return the address of the semaphore on success
- Return `SEM_FAILED` on error (i.e., `(sem_t*)0`)

System Call: sem_wait()

Lock a POSIX Semaphore

```
#include <semaphore.h>

int sem_wait(sem_t *sem);
```

Description

- Block until the value of the semaphore is greater than **0**, then decrements it and returns
- Return **0** on success, -1 on error

System Call: sem_post()

Unlock a POSIX Semaphore

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

Description

- Increment the value of the semaphore pointed to by `sem`
- Return **0** on success, -1 on error

System Call: sem_close()

Close a POSIX Semaphore Structure

```
#include <semaphore.h>

int sem_close(sem_t *sem);
```

Description

- Similar to `close()` for semaphore pointers
- Undefined behavior when closing a semaphore other processes are currently blocked on

System Call: sem_unlink()

Unlink a POSIX Semaphore File

```
#include <semaphore.h>

int sem_unlink(const char *name);
```

Description

- Semaphores files have *kernel* persistence
- Similar to unlink()

Other System Calls

- sem_init() and sem_destroy(): create unnamed semaphores and destroy them (equivalent to combined sem_close() and sem_unlink())
- sem_getvalue(): get the current value of a semaphore
- sem_trywait() and sem_timedwait(): non-blocking and timed versions of sem_wait()

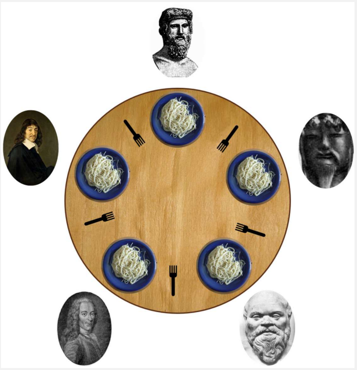
8. Concurrency and Mutual Exclusion

Mutual Exclusion and Deadlocks

Dining Philosophers Problem

- Due to Edsger Dijkstra and Tony Hoare
  - Eating requires two chopsticks (more realistic than forks...)
  - A philosopher may only use the closest left and right chopsticks

  Multiple processes acquire multiple resources
- Deadlock: all philosophers pick their left chopstick, *then* attempt to pick their right one
- Hard to debug non-reproducible deadlocks



Mutual Exclusion and Deadlocks

Preventing Deadlocks

- Eliminate symmetric or cyclic acquire/release patterns
  - Not always possible/desirable

Avoiding Deadlocks

- Use higher-level mutual exclusion mechanisms
  - Monitors
  - Atomic transactions
- Dynamic deadlock avoidance
  - Build a graph of resource usage
  - Detect and avoid cycles
  - Banker's algorithm for counted resources

Mutual Exclusion and Deadlocks

Breaking Deadlocks

1. Timeout
2. Analyze the situation
3. Attempt to reacquire different resources or in a different order

Beyond Deadlocks

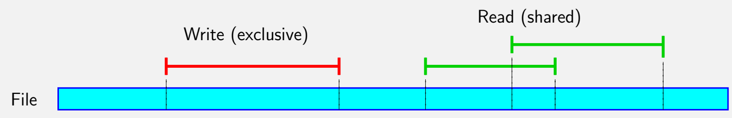
- Livelocks (often occurs when attempting to break a deadlock)
- Aim for fair scheduling: bounded waiting time
- Stronger form of fairness: avoid priority inversion in process scheduling

8. Concurrency and Mutual Exclusion

Alternative: I/O Synchronization With Locks

Purpose

- Serialize processes accessing the same region(s) in a file
- When at least *one process is writing*
- Two kinds of locks: *read* (a.k.a. *shared* and *write* (a.k.a. *exclusive*)
- Two independent APIs supported by Linux
  - POSIX with `fcntl()`
  - BSD with `flock()`



I/O System Call: `fcntl()`

Manipulate a File Descriptor

```c
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, struct flock *lock);
```

Main Commands

`F_DUPFD`: implements `dup()`

`F_GETLK`/`F_SETLK`/`F_SETLKW`: acquire, test or release *file region* (a.k.a. *record*) lock, as described by third argument `lock`

8. Concurrency and Mutual Exclusion – File Locks

I/O System Call: `fcntl()`

Manipulate a File Descriptor

```c
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, struct flock *lock);
```

Return Value

- On success, `fcntl()` returns a (non-negative) value which depends on the command, e.g.,

  `F_DUPFD`: the new file descriptor

  `F_GETLK`/`F_SETLK`/`F_SETLKW`: **0**
- Return -1 on error

I/O System Call: `fcntl()`

Manipulate a File Descriptor

```c
#include <unistd.h>
#include <fcntl.h>

int fcntl(int fd, int cmd);
int fcntl(int fd, int cmd, struct flock *lock);
```

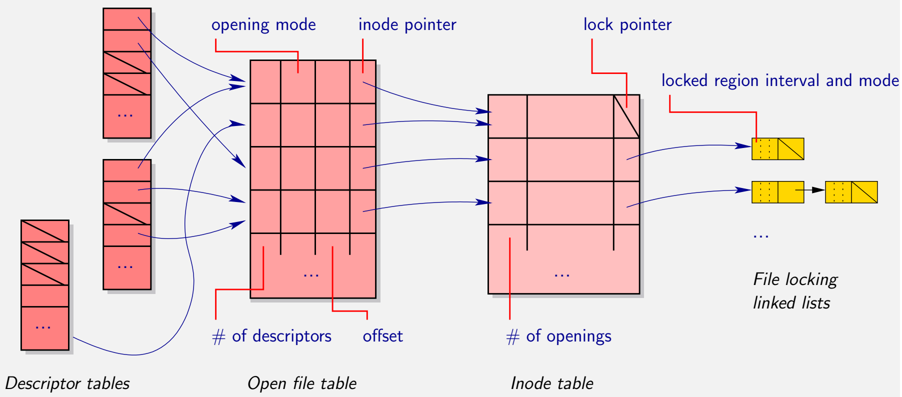
About File Locks

- `fcntl()`-style locks are POSIX locks; *not* inherited upon `fork`
- BSD locks, managed with the `flock()` system call, inherited upon `fork()`
- Both kinds are *advisory*, preserved across `execve()`, fragile to `close()` (releases locks), removed upon termination, and supported by Linux
- `$ man 2 fcntl` and `$ man 2 flock`
- Linux supports SVr3 mandatory `fcntl()`-style locks (mount with `-o mand`)
  - Disabled by default: very deadlock-prone (especially on NFS)
  - Linux prefers *leases* (adds signaling and timeout)

More About File Locks

Implementation Issues

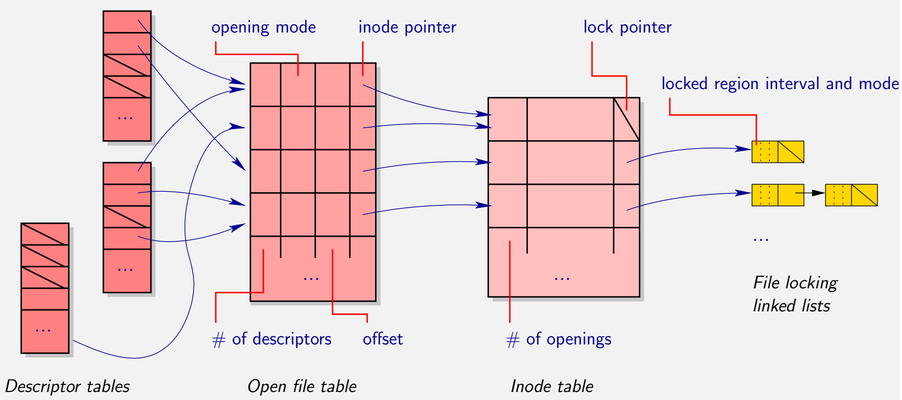
- Locks are associated with *open file entries*:
  → Lost when closing all descriptors related to a file



More About File Locks

Implementation Issues

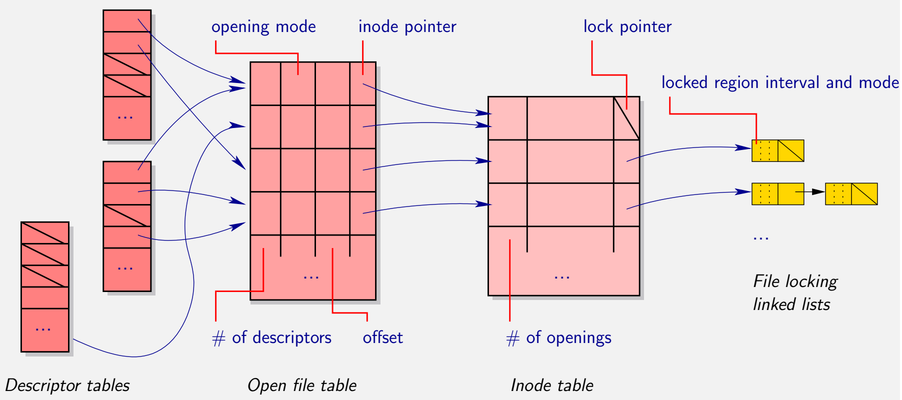
- Incompatible with C library I/O (advisory locking and buffering issues)



More About File Locks

Consequences for the System Programmer

- File locks may be of some use for *cooperating* processes only
- Then, why not use *semaphores*?



8. Concurrency and Mutual Exclusion

System V IPC

Old IPC Interface

- Shared motivation with POSIX IPC
  - Shared memory segments, message queues and semaphore sets
  - Well-defined semantics, widely used, but widely criticized API
  - `$ man 7 svipc`
- But poorly integrated into the file system
  - Uses (hash) *keys* computed from unrelated files
    `$ man 3 ftok`
  - Conflicting and non-standard naming
  - Ad-hoc access modes and ownership rules
- Eventually deprecated by POSIX IPC in 2001

9. Threads

- Applications
- Principles
- Programmer Interface
- Threads and Signals
- Example
- Threads and Mutual Exclusion
- Logical Threads vs. Hardware Threads

Lightweight Shared Memory Concurrency

Motivations

- Finer-grain concurrency than processes
  - Reduce cost of process creation and context switch
  - ≈ lightweight processes (save the process state)
- Implement shared-memory parallel applications
  - Take advantage of cache-coherent parallel processing hardware

9. Threads

- Applications
- Principles
- Programmer Interface
- Threads and Signals
- Example
- Threads and Mutual Exclusion
- Logical Threads vs. Hardware Threads

Multi-Threaded Applications

Thread-Level Concurrency

- Many algorithms can be expressed more naturally with independent computation flows
- Reactive and interactive systems: safety critical controller, graphical user interface, web server, etc.
- Client-server applications, increase modularity of large applications without communication overhead
- Distributed component engineering (CORBA, Java Beans), remote method invocation, etc.

Multi-Threaded Applications

Thread-Level Parallelism

- Tolerate latency (I/O or memory), e.g., creating more logical threads than hardware threads
- Scalable usage of hardware resources, beyond instruction-level and vector parallelism
- Originate in server (database, web server, etc.) and computational (numerical simulation, signal processing, etc.) applications
- Now ubiquitous on multicore systems: Moore's law translates into performance improvements through thread-level parallelism only

9. Threads

Principles

Thread-Level Concurrency and Parallelism

- A single process may contain multiple *POSIX threads*, a.k.a. *logical threads*, or simply, *threads*
  - Share a *single memory space*
    - Code, static data, heap
  - Distinct, *separate stack*
- Impact on operating system
  - Schedule threads and processes
  - Map POSIX threads to hardware threads
  - Programmer interface compatibility with single-threaded processes
- `$ man 7 pthreads`

Threads vs. Processes

Shared Attributes

- PID, PPID, PGID, SID, UID, GID
- Current and root directories, controlling terminal, open file descriptors, record locks, file creation mask (`umask`)
- Timers, signal settings, priority (`nice`), resource limits and usage

Threads vs. Processes

Shared Attributes

- PID, PPID, PGID, SID, UID, GID
- Current and root directories, controlling terminal, open file descriptors, record locks, file creation mask (`umask`)
- Timers, signal settings, priority (`nice`), resource limits and usage

Distinct Attributes

- Thread identifier: `pthread_t` data type
- Signal mask (`pthread_sigmask()`)
- `errno` variable
- Scheduling policy and real-time priority
- CPU affinity (NUMA machines)
- Capabilities (Linux only, `$ man 7 capabilities`)

Threads vs. Processes

Shared Attributes

- PID, PPID, PGID, SID, UID, GID
- Current and root directories, controlling terminal, open file descriptors, record locks, file creation mask (`umask`)
- Timers, signal settings, priority (`nice`), resource limits and usage

Distinct Attributes

- Thread identifier: `pthread_t` data type
- Signal mask (`pthread_sigmask()`)
- `errno` variable
- Scheduling policy and real-time priority
- CPU affinity (NUMA machines)
- Capabilities (Linux only, `$ man 7 capabilities`)

To use POSIX threads, compile with `-pthread`

9. Threads

System Call: `pthread_create()`

Create a New Thread

```
#include <pthread.h>

int pthread_create(pthread_t *thread, pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

Semantics

- The new thread calls `start_routine(arg)`
- The `attr` argument corresponds to thread attributes, e.g., it can be *detached* or *joinable*, see `pthread_attr_init()` and `pthread_detach()`
  - If `NULL`, default attributes are used (it is *joinable* (i.e., not *detached*) and has default (i.e., non *real-time*) scheduling policy
- Return **0** on success, or a non-null error condition; stores identifier of the new thread in the location pointed to by the `thread` argument
- Note: `errno` is *not* set

System Call: pthread_exit()

Terminate the Calling Thread

```
#include <pthread.h>

void pthread_exit(void *retval);
```

Semantics

- Terminates execution
  - After calling cleanup handlers; set with pthread_cleanup_push()
  - Then calling finalization functions for thread-specific data, see pthread_key_create()
- The retval argument (an arbitrary pointer) is the return value for the thread; it can be consulted with pthread_join()
- Called implicitely if the thread routine returns
- pthread_exit() never returns

System Call: pthread_join()

Wait For Termination of Another Thread

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **thread_return);
```

Semantics

- Suspend execution of the calling thread until thread *terminates* or is *canceled*, see pthread_cancel()
- If thread_return is not null
  - Its value is the pointer returned upon termination of thread
  - Or PTHREAD_CANCELED if thread was canceled
- thread must not be *detached*, see pthread_detach()
- Thread resources are *not* freed upon termination, only when calling pthread_join() of pthread_detach(); watch out for memory leaks!
- Return **0** on success, or a non-null error condition
- Note: errno is *not* set

Thread-Local Storage

Thread-Specific Data (TSD)

- *Private memory* area associated with each thread
- Some global variables need to be private
  - Example: errno
  - More examples: OpenMP programming language extensions
  - General compilation method: *privatization*
- Implementation: pthread_key_create()

Finalization Functions

- Privatization of non-temporary data may require
  - *Copy-in*: broadcast shared value into multiple private variables
  - *Copy-out*: select a private value to update a shared variable upon termination
- Memory management (destructors) for dynamically allocated TSD

9. Threads

Threads and Signals

Sending a Signal to A Particular Thread

→ `pthread_kill()`
Behaves like `kill()`, but *signal actions and handlers are global to the process*

Blocking a Signal in A Particular Thread

→ `pthread_sigmask()`
Behaves like `sigprocmask()`

Suspending A Particular Thread Waiting for Signal Delivery

→ `sigwait()`
Behaves like `sugsuspend()`, suspending thread execution (thread-local) and blocking a set of signals (global to the process).

9. Threads

Example: Typical Thread Creation/Joining

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <sys/times.h>

#define NTHREADS 5

void *thread_fun(void *num) {
  int i = *(int *)num;

  printf("Thread %d\n", i);         // Or pthread_self()

  // ...
  // More thread-specific code
  // ...

  pthread_exit(NULL);         // Or simply return NULL
}
```


Example: Typical Thread Creation/Joining

```
pthread_t threads[NTHREADS];

int main(int argc, char *argv[]) {
  pthread_attr_t attr;
  int i, error;
  for (i = 0; i < NTHREADS; i++) {
    pthread_attr_init(&attr);
    int *ii = malloc(sizeof(int)); *ii = i;
    error = pthread_create(&threads[i], &attr, thread_fun, ii);
    if (error != 0) {
      fprintf(stderr, "Error in pthread_create: %s \n", strerror(error));
      exit(1);
    }
  }
  for (i=0; i < NTHREADS; i++) {
    error = pthread_join(threads[i], NULL);
    if (error != 0) {
      fprintf(stderr, "Error in pthread_join: %s \n", strerror(error));
      exit(1);
    }
  }
}
```


9. Threads

System Call: pthread_mutex_init()

Initialisation of a mutex

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *attr);
```

Semantics

- Perform mutex initialization
- The mutex variable has to be shared among the threads willing to use the same lock; initialization has to occur exactly one time
  - For re-using an already initialized mutex see pthread_mutex_destroy
- The attr argument is the mutex type attribute: it can be *fast*, *recursive* or *error checking*; see pthread_mutexattr_init()
  - If NULL, *fast* is assumed by default
- Return **0** on success, or a non-null error condition
- Initialization can also be performed statically with default attributes by using: pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

System Call: pthread_mutex_*un*lock()

Acquiring/Releasing a lock

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Semantics of pthread_mutex_lock

- Block the execution of the current thread until the lock referenced by mutex becomes available
  - Attemtping to re-lock a mutex after acquiring the lock leads to different behaviour depending on mutex attributes (see previous slide)
- The system call is *not* interrupted by a signal
- Return **0** on success, or a non-null error condition

Semantics of pthread_mutex_unlock

- Release the lock (if acquired by the current thread)
- The lock is passed to a blocked thread (if any) depending on schedule
- Return **0** on success, or a non-null error condition

System Call: pthread_mutex_*try/timed*lock()

Acquiring a lock without blocking

```
#include <pthread.h>
int pthread_mutex_trylock(pthread_mutex_t * mutex);
int pthread_mutex_timedlock(pthread_mutex_t * mutex,
                            struct timespec * abs_timeout);
```

Semantics of pthread_mutex_trylock

- Try to acquire the lock and return immediately in case of failure
- Return **0** on success, or a non-null error condition

Semantics of pthread_mutex_timedlock

- Block the execution of the current thread until the lock becomes available or until abs_timeout elapses
- Return **0** on success, or a non-null error condition

Read/Write Locks

Principles

- Allow concurrent read and guarantee excluse write
- Similar API to regular mutexes
  - pthread_rwlock_init() – initialize a read/write lock
  - pthread_rwlock_rdlock() – get a shared read lock
  - pthread_rwlock_wrlock() – get an exclusive write lock
  - pthread_rwlock_unlock() – unlock an exclusive write or shared read lock
  - pthread_rwlock_tryrdlock() – get a shared read lock w/o waiting
  - pthread_rwlock_trywrlock() – get an exclusive write lock w/o waiting
  - pthread_rwlock_timedrdlock() – get a shared read lock with timeout
  - pthread_rwlock_timedwrlock() – get an exclusive write lock with timeout

Condition Variables

Overview

- Producer-Consumer synchronization mechanism
- Block the execution of a thread until a boolean predicate becomes true
- Require dedicated instructions to wait without busy-waiting

Principles

- A mutex is used to atomically test a predicate, and according to its value:
  - either the execution continues
  - or the execution is blocked until it is *signaled*
- Once signaled, the thread waiting on the condition resumes
- The mutex prevents race-conditions when a thread is going to wait while being signaled

System Call: pthread_cond_wait()

Blocking a thread according to a given condition

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond,pthread_mutex_t *mutex);
```

Semantics

- Atomically block the execution of a thread and release the mutex lock
- Once the condition variable cond is signaled by another thread, atomically reacquire the mutex lock and resume execution
- Return **0** on success, or a non-null error condition
- Like mutex variables, condition variables have to be initialized with a system call
- pthread_cond_timedwait() can also resume the execution after the end of a given timeout

System Call: pthread_cond_*signal/broadcast* ()

Signaling or broadcasting a condition

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

Semantics

- Signal *one* (pthread_cond_signal) or *every* (pthread_cond_broadcast) threads waiting on the condition variable cond.
- If no thread is waiting, nothing happens. Signal is *lost*.
- Return **0** on success, or a non-null error condition

Example: Typical use of Condition Variables

```
int x, y; //  Shared variables
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

void *thread_one(void *param) {
   // ...
   pthread_mutex_lock(&mutex);
   while (x <= y) {
      pthread_cond_wait(&cond, &mutex);
   }
   //  Now we can be sure that x > y
   pthread_mutex_unlock(&mutex);
   //  No more guarantee on the value of x > y
}
void *thread_two(void *param) {
    // ...
    pthread_mutex_lock(&mutex);
    // modification of x and y
    // no need to send a signal if the predicate is false
    if (x > y)
      pthread_cond_broadcast(&cond);
    pthread_mutex_unlock(&mutex);
}
```


pthread Implementation: Futexes

Futexes Overview

- Futex: fast userspace mutex
- *Low level* synchronization primitives used to program *higher-level* locking abstractions
- Appeared recently in the Linux kernel *(since 2.5.7)*
- Rely on:
  - a *shared integer in user space* to synchronize threads
  - two system calls (*kernel space*) to make a thread wait or to wake up a thread
- *Fast:* most of the time only the shared integer is required
- *Difficult to use:* no deadlock protection, subtle correctness and performance issues
- For more information: read *futexes are tricky* by Ulrich Drepper http://people.redhat.com/drepper/futex.pdf

9. Threads

Logical Threads vs. Hardware Threads

Logical Thread Abstraction

Multiple *concurrent* execution contexts of the same program, cooperating over a single memory space, called *shared address space* (i.e., shared data, consistent memory addresses across all threads)

Among the different forms of logical thread abstrations, *user-level* threads do not need a processor/kernel context-switch to be scheduled

Mapping Logical to Hardware Threads

The hardware threads are generally exposed directly as operating system kernel threads (POSIX threads); these can serve as *worker threads* on which user-level threads can be mapped
Mapping strategies: one-to-one, many-to-one ("green" threads), *many-to-many*

Logical Threads vs. Hardware Threads

Thread "Weight"

1. Lightest: run-to-completion coroutines
   $\longrightarrow$ indirect function call
2. Light: coroutines, fibers, protothreads, cooperative user-level threads
   $\longrightarrow$ garbage collector, cactus stacks, register checkpointing
3. Lighter: preemptive user-level threads
   $\longrightarrow$ preemption support (interrupts)
4. Heavy: kernel threads (POSIX threads)
   $\longrightarrow$ context switch
5. Heavier: kernel processes
   $\longrightarrow$ context switch with page table operations (TLB flush)

Task Pool

General approach to schedule user-level threads

- Single task queue
- Split task queue for scalability and dynamic load balancing

More than one pool may be needed to separate ready threads from waiting/blocked threads

Task Pool: Single Task Queue

Simple and effective for small number of threads

Caveats:

- The single shared queue becomes the point of contention
- The time spent to access the queue may be significant as compared to the computation itself
- Limits the scalability of the parallel application
- Locality is missing all together

Task Pool: Split Task Queue

Work Sharing

Threads with more work push work to threads with less work A centralized scheduler balances the work between the threads

Work Stealing

A thread that runs out of work tries to steal work from some other thread

The Cilk Project

- Language for dynamic multithreaded applications
- C dialect
- Developed since 1994 at MIT in the group of Charles Leiserson
  `http://supertech.csail.mit.edu/cilk`
  Now part of Intel Parallel Studio (and TBB, ArBB)
- Influenced OpenMP tasks (OpenMP 3.0), and other coroutine-based parallel languages

Fibonacci in Cilk

- Tasks are (nested) coroutines
- Two keywords:
  - `spawn` *function*`()` to indicate that the function call *may* be executed as a coroutine
  - `sync` to implement a *synchronization barrier*, waiting for *all previously spawned tasks*

```
cilk int fib(int n) {
  if (n < 2)
    return n;
  else {
    int x, y;
    x = spawn fib(n-1);
    y = spawn fib(n-2);
    sync;
    return (x+y);
  }
}
```


10. Network Interface

- Principles
- Connectionless Communications
- Connection-Based Communications
- Programmer Interface
- Threaded Server Model
- Distributed Systems

OS Abstraction for Distributed I/O

Challenges

- Abstract multiple *layers* of multiple *networking protocol stacks*
- Cross-system synchronization and communication primitives
- Extend classical I/O primitives to distributed systems

10. Network Interface

- Principles
- Connectionless Communications
- Connection-Based Communications
- Programmer Interface
- Threaded Server Model
- Distributed Systems

Open Systems Interconnection (OSI)

Basic Reference Model

- Layer 7: Application layer — RPC, FTP, HTTP, NFS
- Layer 6: Presentation layer — XDR, SOAP XML, Java socket API
- *Layer 5: Session layer — TCP, DNS, DHCP*
- *Layer 4: Transport layer — TCP, UDP, RAW*
- *Layer 3: Network layer — IP*
- Layer 2: Data Link layer — Ethernet protocol
- Layer 1: Physical layer — Ethernet digital signal processing

OS Interface

- Abstract layers 3, 4 and 5 through special files: *sockets*

Socket Abstraction

What?

- Bidirectional communication channel across systems called *hosts*

10. Network Interface – Principles

Socket Abstraction

What?

- Bidirectional communication channel across systems called *hosts*

Networking Domains

- *INET*: Internet Protocol (IP)
- *UNIX*: efficient host-local communication
- And many others (IPv6, X.25, etc.)

- `$ man 7 socket`
- `$ man 7 ip` or `$ man 7 ipv6` (for INET sockets)
- `$ man 7 unix`

251 / 352

10. Network Interface – Principles

Socket Abstraction

What?

- Bidirectional communication channel across systems called *hosts*

Socket Types

- STREAM: *connected* FIFO streams, reliable (error detection and replay), without message boundaries, much like `pipe`s across hosts
- DGRAM: *connection-less*, unreliable (duplication, reorder, loss) exchange of messages of fixed length (datagrams)
- RAW: direct access to the raw protocol (not for UNIX sockets)
- Mechanism to *address* remote sockets depends on the socket type
  - `$ man 7 tcp` Transmission Control Protocol (TCP): for STREAM sockets
  - `$ man 7 udp` User Datagram Protocol (UDP): for DGRAM sockets
  - `$ man 7 raw` for RAW sockets
- Two classes of INET sockets
  - IPv4: 32-bit address and 16-bit port
  - IPv6: 128-bit address and 16-bit port

251 / 352

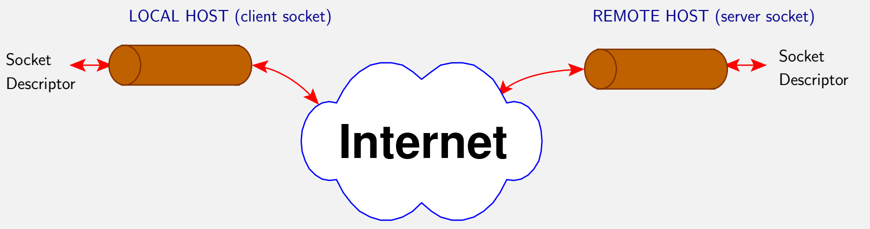
10. Network Interface

Scenarios for Socket-to-Socket Connection

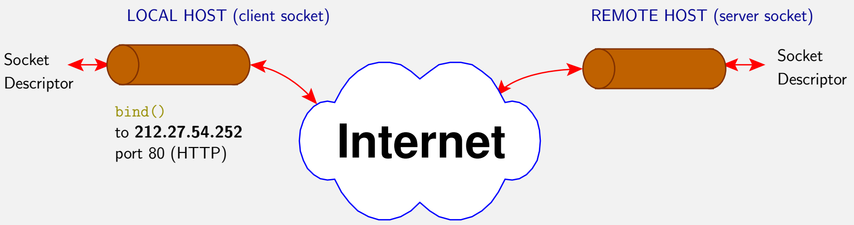
Direct Communication Scenario

- Create a socket with `socket()`
- Bind to a local address with `bind()`
- In the remote host, go through the first **2** steps exchanging the roles of local and remote addresses
- Only DGRAM (UDP) sockets can be operated that way
- Note: port numbers only provide a partial support for a rendez-vous protocol: unlike named FIFOs, no synchronization is enforced

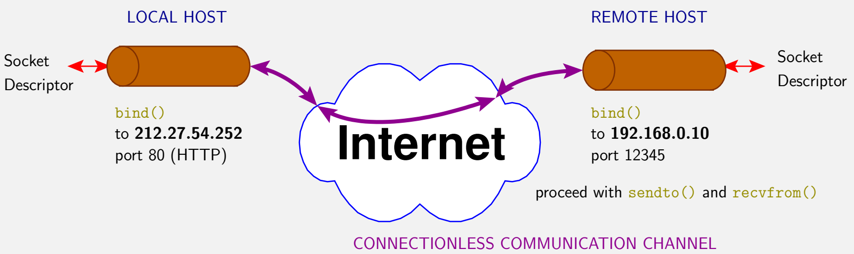
Establishing a Connectionless Channel



Establishing a Connectionless Channel



Establishing a Connectionless Channel



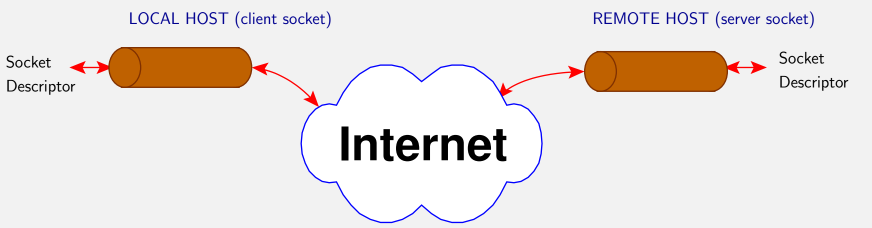
10. Network Interface

Scenarios for Socket-to-Socket Connection

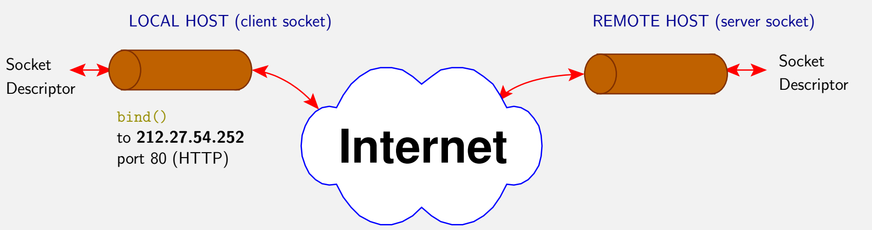
TCP Abstraction: Creation of a Private Channel

- Create a socket with `socket()`
- Bind to a local address with `bind()`
- Call `listen()` to tell the socket that new connections shall be accepted
- Call `accept()` to wait for an incoming connection, returning a new socket associated with a private channel (or "session") for this connection
- In the remote host, go through the first two steps exchanging the roles of local and remote addresses, and calling `connect()` instead of `bind()`
- The original pair of sockets can be reused to create more private channels
- Reading or writing from a "yet unconnected" connection-based socket raises `SIGPIPE` (like writing to a pipe without readers)

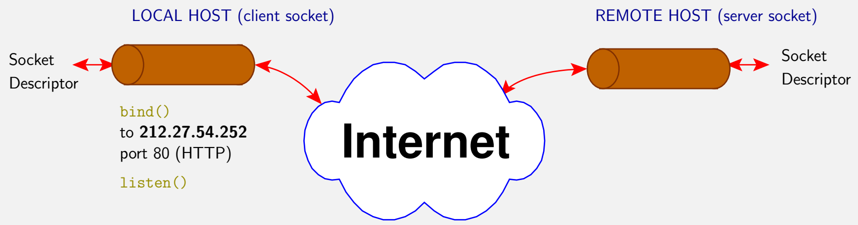
Establishing a Connection-Based Channel



Establishing a Connection-Based Channel

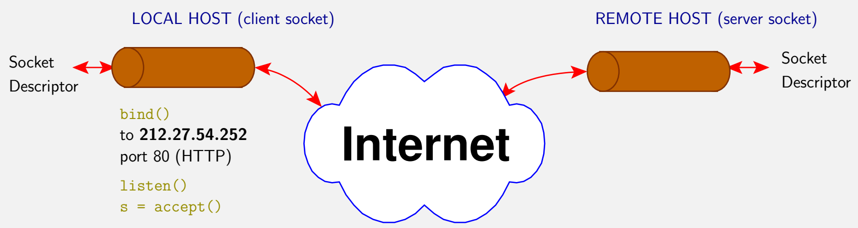


Establishing a Connection-Based Channel



10. Network Interface – Connection-Based Communications

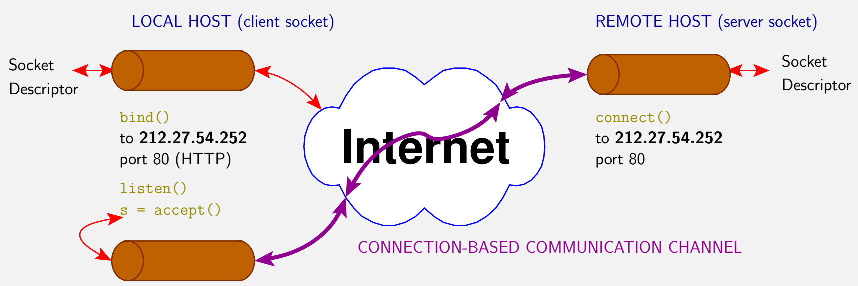
Establishing a Connection-Based Channel



257 / 352

10. Network Interface – Connection-Based Communications

Establishing a Connection-Based Channel



257 / 352

10. Network Interface

Establishing a Socket for Incoming Connections

```
#include <netdb.h>
#include <sys/socket.h>

int establish(unsigned short portnum) {
  int s; char myname[MAXHOSTNAME+1]; struct sockaddr_in sa;
  memset(&sa, 0, sizeof(struct sockaddr_in));    // Clear our address
  gethostname(myname, MAXHOSTNAME);              // Who are we?
  struct hostent *hp = gethostbyname(myname);    // Get our address info
  if (hp == NULL) return -1;                     // We do not exist!?
  memcpy(&sa.sin_addr.s_addr, hp->h_addr, hp->h_length); // Host address
  sa.sin_family = hp->h_addrtype;                // And address type
  sa.sin_port = htons(portnum);                  // And our big-endian port
  s = socket(AF_INET, SOCK_STREAM, 0))           // Create socket
  if (s < 0) return -1;                          // Socket creation failed
  int my_true = 1; // Immediate reuse of the local port after closing socket
  int so_r = setsockopt(*s, SOL_SOCKET, SO_REUSEADDR, &my_true, sizeof(my_true));
  // Wait 10 seconds after closing socket for reliable transmission
  struct linger my_linger = { .l_onoff = 1, .l_linger = 10 };
  so_r |= setsockopt(*s, SOL_SOCKET, SO_LINGER, &my_linger, sizeof(my_linger));
  if (so_r) { perror ("setsockopt"); close (*s); return -1; }
  if (bind(s, &sa, sizeof(sa), 0) < 0)           // Bind address to socket
    { close(s); return -1; }
  return s;
}
```


Waiting for Incoming Connections

```
int wait_for_connections(int s) {      // Socket created with establish()
  struct sockaddr_in sa;               // Address of socket
  int i = sizeof (sa);                 // Size of address
  int t;                               // Socket of connection

  listen(s, 3);                        // Max # of queued connections
  if ((t = accept(s, &sa, &i)) < 0)    // Accept connection if there is one
    return -1;
  return t;
}
```

Opening an Outgoing Connection

```
int call_socket(char *hostname, unsigned short portnum) {
  struct sockaddr_in sa;
  struct hostent *hp;
  int a, s;

  if ((hp = gethostbyname(hostname)) == NULL) {          // Do we know
    errno = ECONNREFUSED;                                // The host's address?
    return -1;                                           // No
  }

  memset(&sa, 0, sizeof(sa));
  memcpy(&sa.sin_addr, hp->h_addr, hp->h_length);        // Set address
  sa.sin_family = hp->h_addrtype;                        // And type
  sa.sin_port = htons(portnum);                          // And big-endian port

  if ((s = socket(hp->h_addrtype, SOCK_STREAM, 0)) < 0)  // Get socket
    return -1;
  if (connect(s, &sa, sizeof(sa)) < 0) {                 // Connect
    close(s); return -1;
  }
  return s;
}
```


Communicating Through a Pair of Sockets

Connected Socket I/O

- System calls `read()` and `write()` work as usual on *connected* sockets (otherwise raise `SIGPIPE`)
- System calls `recv()` and `send()` refine the semantics of `read()` and `write()` with additional flags to control socket-specific I/O (out-of-band, message boundaries, etc.)
- System call `shutdown(int sockfd, int how)` causes all or part of a full-duplex TCP connection to shut down, according to the value of `how`: `SHUT_RD`, `SHUT_WR`, `SHUT_RDWR` respectively disallow receptions, transmissions, and both receptions and transmissions; this call is important to avoid dead-locks or to simulate end-of-file through TCP connections (analog to selectively closing pipe descriptors)

Connection-Less Socket I/O

- A *single* DGRAM (UDP) socket can be used to communicate
- System calls: `recvfrom()` and `sendto()`

10. Network Interface

Application: Threaded Server Model

Dynamic Thread Creation

1. A *main thread* `listen`s for a `connection` request on a *predefined port*
2. After `accept`ing the request, the server creates a thread to handle the request and resumes `listen`ing for another request
3. The thread `detach`es itself, performs the request, closes the `socket` in response to the client's closing and returns

⟶ the thread function takes the socket returned from the `accept()` system call as a parameter

10. Network Interface – Threaded Server Model

Application: Threaded Server Model

Dynamic Thread Creation

1. A *main thread* `listen`s for a `connection` request on a *predefined port*
2. After `accept`ing the request, the server creates a thread to handle the request and resumes `listen`ing for another request
3. The thread `detach`es itself, performs the request, closes the `socket` in response to the client's closing and returns

⟶ the thread function takes the socket returned from the `accept()` system call as a parameter

Worker Pool

1. A *main thread* plays the role of a *producer*
2. A *bounded* number of *worker threads* play the role of *consumers*
3. The main thread `listen`s for `connection` requests and asks the workers to process them, e.g., enqueuing a task/coroutine on the worker's work list

10. Network Interface – Threaded Server Model

Application: Threaded Server Model

Dynamic Thread Creation

1. A *main thread* `listen`s for a `connection` request on a *predefined port*
2. After `accept`ing the request, the server creates a thread to handle the request and resumes `listen`ing for another request
3. The thread `detach`es itself, performs the request, closes the `socket` in response to the client's closing and returns

⟶ the thread function takes the socket returned from the `accept()` system call as a parameter

Worker Pool

1. A *main thread* plays the role of a *producer*
2. A *bounded* number of *worker threads* play the role of *consumers*
3. The main thread `listen`s for `connection` requests and asks the workers to process them, e.g., enqueuing a task/coroutine on the worker's work list

More Information and Optimizations: `http://www.kegel.com/c10k.html`

10. Network Interface

- Principles
- Connectionless Communications
- Connection-Based Communications
- Programmer Interface
- Threaded Server Model
- Distributed Systems

Distributed Systems and Protocols

RFC: Request for Comments

- RFC-Editor: `http://www.rfc-editor.org/rfc.html`
- IETF: Internet Engineering Task Force
- IANA: Internet Assigned Numbers Authority

Open Systems Interconnection (OSI)

Basic Reference Model

- *Layer 7: Application layer* — *RPC, FTP, HTTP, NFS*
- *Layer 6: Presentation layer* — *XDR, SOAP XML, Java socket API*
- *Layer 5: Session layer* — *TCP, DNS, DHCP*
- Layer 4: Transport layer — TCP, UDP, RAW
- Layer 3: Network layer — IP
- Layer 2: Data Link layer — Ethernet protocol
- Layer 1: Physical layer — Ethernet digital signal processing

Interface

- Abstract layers 4, 5 and 6 through dedicated protocols
- Virtualize distributed system resources over these protocols
  Cloud services: storage and computation resources, applications

More Information on Distributed Systems

- Look for information on each individual protocol
- Overview of distributed Computing: http://en.wikipedia.org/wiki/Distributed_computing
- Distributed Operating Systems (Andrew Tanenbaum): http://www.cs.vu.nl/pub/amoeba/amoeba.html
- Peer to peer: http://en.wikipedia.org/wiki/Peer-to-peer

→ See INF570 course

11. Kernel Design

- Interrupts and Exceptions
- Low-Level Synchronization
- Low-Level Input/Output
- Devices and Driver Model
- File Systems and Persistent Storage
- Memory Management
- Process Management and Scheduling
- Operating System Trends
- Alternative Operating System Designs

Bottom-Up Exploration of Kernel Internals

Hardware Support and Interface

- Asynchronous events, switching to kernel mode
- I/O, synchronization, low-level driver model

Operating System Abstractions

- File systems, memory management
- Processes and threads

Specific Features and Design Choices

- Linux 2.6 kernel
- Other UNIXes (Solaris, MacOS), Windows XP and real-time systems

11. Kernel Design

- Interrupts and Exceptions
- Low-Level Synchronization
- Low-Level Input/Output
- Devices and Driver Model
- File Systems and Persistent Storage
- Memory Management
- Process Management and Scheduling
- Operating System Trends
- Alternative Operating System Designs

Hardware Support: Interrupts

- Typical case: electrical signal asserted by external device
  - Filtered or issued by the *chipset*
  - Lowest level hardware synchronization mechanism
- Multiple priority levels: Interrupt ReQuests (IRQ)
  - Non-Maskable Interrupts (NMI)
- Processor switches to kernel mode and calls specific *interrupt service routine* (or *interrupt handler*)
- Multiple drivers may share a single IRQ line
  → IRQ handler must identify the source of the interrupt to call the proper service routine

Hardware Support: Exceptions

- Typical case: unexpected program behavior
  - Filtered or issued by the *chipset*
  - Lowest level of OS/application interaction
- Processor switches to kernel mode and calls specific *exception service routine* (or *exception handler*)
- Mechanism to implement *system calls*

11. Kernel Design

Hardware Interface: Kernel Locking Mechanisms

Low-Level Mutual Exclusion Variants

- Very short critical sections
  - Spin-lock: active loop polling a memory location
- Fine grain
  - Read/write lock: traditional read/write semaphore
  - Seqlock: high-priority to writes, speculative (restartable) readers
  - Read-copy update (RCU) synchronization: zero/low-overhead concurrent readers, concurrent writers in special cases
- Coarse grain
  - Disable preemption and interrupts
  - The "big kernel lock"
    - Non scalable on parallel architectures
    - Only for very short periods of time
    - Now mostly in legacy drivers and in the virtual file system

Hardware Interface: Spin-Lock

Example

- Busy waiting

```
do {
  while (lock == 1) { pause_for_a_few_cycles(); }
  atomic { if (lock == 0) { lock = 1; break; } }
} while (lock == 0);
// Critical section
lock = 0;
// Non-critical section
```

Applications

- Wait for short periods, typically less than **1 μs**
  - As a proxy for other locks
  - As a *polling* mechanism
  - Mutual exclusion in interrupts
- Longer periods would be wasteful of computing resources

Beyond Locks: Read-Copy Update (RCU)

Principles

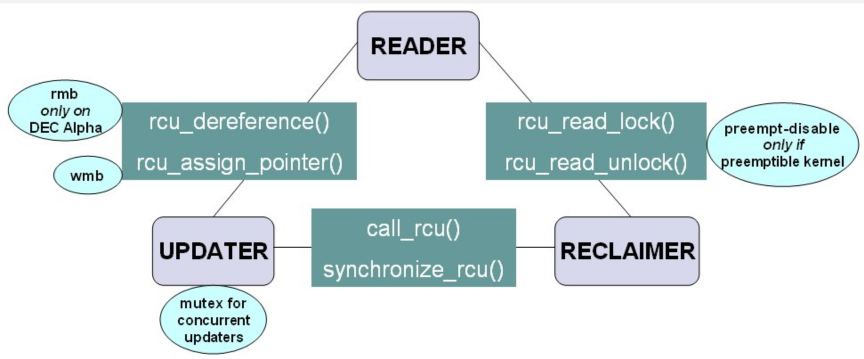
- Synchronization mechanism to improve scalability and efficiency
- RCU supports concurrency between a *single updater and multiple readers*
- Reads are kept atomic by maintaining multiple versions of objects — *privatization* — and ensuring that they are not freed up until all pre-existing read-side critical sections complete
- In non-preemptible kernels, RCU's read-side primitives have zero overhead
- Mechanisms:
  1. Publish-Subscribe mechanism (for insertion)
  2. Wait for pre-existing RCU readers to complete (for deletion)
  3. Maintain multiple versions of recently updated objects (for readers)

Beyond Locks: More About RCU

Programming Interface

- `rcu_read_lock()` and `rcu_read_unlock()`: delimit a RCU "read-side critical" section; important for kernel premption
- `rcu_assign_pointer()`: assign a new value to an RCU-protected pointer, with the proper fences (memory barriers)
- `rcu_dereference()`: return a pointer that may be safely dereferenced (i.e., pointing to a consistent data structure)
- `synchronize_rcu()`: blocks until all current read-side critical sections have completed, but authorize new read-side critical sections to start and finish

Beyond Locks: More About RCU



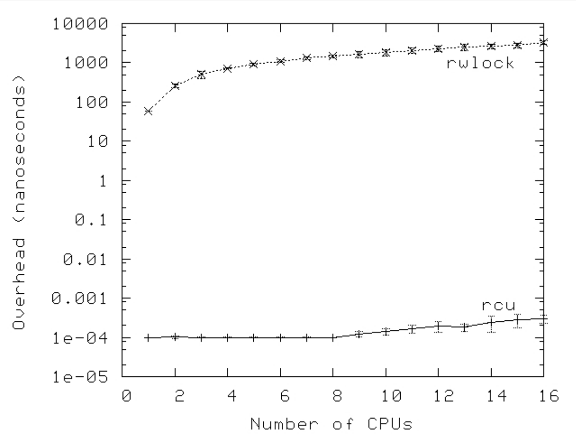
Beyond Locks: More About RCU

Toy but Correct Implementation on Non-Preemptible Kernels

```
void rcu_read_lock(void) { }
void rcu_read_unlock(void) { }
void synchronize_rcu(void) {
  /* force wait when kernel is not preemptible */
  attempt_to_switch_context_on_all_cpus();
}
#define rcu_assign_pointer(p, v) ({ \
    smp_wmb(); \
    (p) = (v); \
  })
#define rcu_fetch_pointer(p) ({ \
    typeof(p) _pointer_value = (p); \
    smp_rmb(); /* not needed on all architectures */ \
    (_pointer_value); \
  })
```

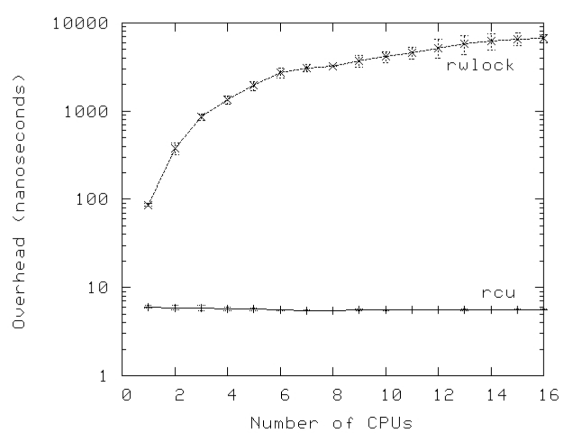

Beyond Locks: More About RCU

Overhead of RCU on *non-preemptible* 16-CPU Intel x86 at 3GHz



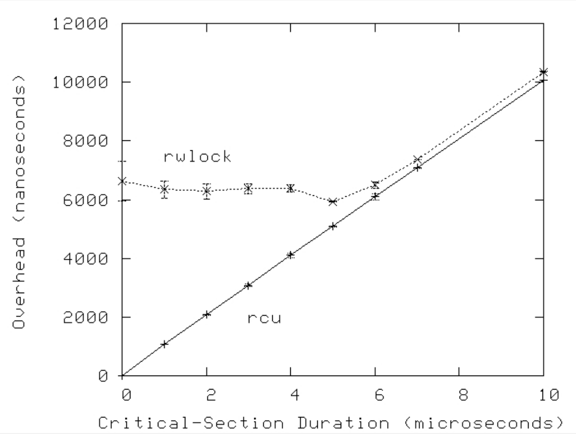
Beyond Locks: More About RCU

Overhead of RCU on *preemptible* 16-CPU Intel x86 at 3GHz



Beyond Locks: More About RCU

Total execution time of rwlock and RCU vs execution time in the critical section(s)



Beyond Locks: More About RCU

Online Resources

From the RCU wizard: Paul McKenney, IBM

http://www.rdrop.com/users/paulmck/RCU

11. Kernel Design

Hardware Support: Memory-Mapped I/O

External Remapping of Memory Addresses

- Builds on the chipset rather than on the MMU
  - Address translation + redirection to device memory or registers
- Unified mechanism to
  - Transfer data: just load/store values from/to a memory location
  - Operate the device: reading/writing through specific memory addresses actually sends a command to a device
    Example: *strobe* registers (writing anything triggers an event)
- Supports Direct Memory Access (DMA) block transfers
  - Operated by the DMA controller, not the processor
  - Choose between *coherent* (a.k.a. synchronous) or *streaming* (a.k.a. non-coherent or asynchronous) DMA mapping

Hardware Support: Port I/O

Old-Fashioned Alternative

- Old interface for x86 and IBM PC architecture
- Rarely supported by modern processor instruction sets
- Low-performance (ordered memory accesses, no DMA)

11. Kernel Design

Hardware Interface: Device Drivers

Overview

- Abstracted by system calls or kernel processes
- Manage buffering between device and local buffer
- Control devices through memory-mapped I/O (or I/O ports)
- Devices trigger interrupts (end of request, buffer full, etc.)
- Many concurrency challenges (precise synchronization required)
- Multiple layers for portability and reactivity

Hardware Interface: Driver Model in Linux

Low-Level Device Driver

- Automatic configuration: “plug’n’play”
  - Memory mapping
  - Interrupts (IRQ)
- Automatic configuration of device mappings
  - *Device numbers: kernel anchor for driver interaction*
  - Automatic assignment of *major* and *minor* numbers
    - At *discovery-time*: when a driver recognizes the signature of a device (e.g., PCI number)
    - At boot-time or plug-time
  - Hot pluggable devices

Hardware Interface: Driver Model in Linux

Device Special Files

- *Block*-oriented device
  Disks, file systems: `/dev/hda` `/dev/sdb2` `/dev/md1`
- *Character*-oriented device
  Serial ports, console terminals, audio: `/dev/tty0` `/dev/pts/0` `/dev/usb/hiddev0` `/dev/mixer` `/dev/null`
- *Major* and *minor* numbers to (logically) project device drivers to device special files

Hardware Interface: Driver Model in Linux

Low-Level Statistics and Management

- Generic device abstraction: `proc` and `sysfs` pseudo file systems
  - Class (`/sys/class`)
  - Module (parameters, symbols, etc.)
  - Resource management (memory mapping, interrupts, etc.)
  - Bus interface (PCI: `$ lspci`)
  - Power management (sleep modes, battery status, etc.)

Block Device

```
$ cat /sys/class/scsi_device/0:0:0:0/device/block:sda/dev
8:0

$ cat /sys/class/scsi_device/0:0:0:0/device/block:sda/sda3/dev
8:3
```


Hardware Interface: Driver Model in Linux

Kernel Objects and Events

- Main concept: `kobject`
  - Abstraction for devices, drivers, temporary structures, etc.
  - Representation (path) in `sysfs`
  - Type, parent pointer (hierarchy), reference count (garbage collection)
  - Ability to send *uevent*s to publish the state of the kernel object
  - Define which of these *uevent*s are exported to userspace, e.g., to be monitored by low-level daemons
- One application: automatic device node creation: `udev`
  - Userspace tools: `man udev`, `udevd` daemon, `udevadm` command
  - `udevadm info --export-db`
  - `udevadm monitor`
  - ...

Hardware Interface: Driver Model in Linux

Device Driver Examples

- *Device name: application anchor to interact with the driver*
- User level
- Reconfigurable rules
- Hot pluggable devices

Block Device

```
$ cat /sys/class/scsi_device/0:0:0:0/device/uevent
DEVTYPE=scsi_device
DRIVER=sd
PHYSDEVBUS=scsi
PHYSDEVDRIVER=sd
MODALIAS=scsi:t-0x00
$ cat /sys/class/scsi_device/0:0:0:0/device/block:sda/dev
8:0
$ cat /sys/class/scsi_device/0:0:0:0/device/block:sda/sda3/dev
8:3
```

Hardware Interface: Driver Model in Linux

Device Driver Examples

- *Device name: application anchor to interact with the driver*
- User level
- Reconfigurable rules
- Hot pluggable devices

Network Interface

```
$ cat /sys/class/net/eth0/uevent
PHYSDEVPATH=/devices/pci0000:00/0000:00:1c.2/0000:09:00.0
PHYSDEVBUS=pci
PHYSDEVDRIVER=tg3
INTERFACE=eth0
IFINDEX=2
```


Driver Model: Concurrency Challenges

Cost of Abstraction and Concurrency

- Complex kernel control paths

Typical Kernel Control Path: Swap Memory

1. Page fault of user application
2. Exception, switch to kernel mode
3. Lookup for cause of exception, detect access to swapped memory
4. Look for name of swap device (multiple swap devices possible)
5. Call non-blocking kernel I/O operation
6. Retrieve device major and minor numbers
7. Forward call to the driver
8. Retrieve page (possibly swapping another out)
9. Update the kernel and process's page table
10. Switch back to user mode and proceed

Concurrency Challenges

Concurrent Execution of Kernel Control Path

- Modern kernels are multi-threaded for reactivity and performance
  - Other processes
  - Other kernel control paths (interrupts, preemptive kernel)
  - Deferred interrupts (softirq/tasklet mechanism)
  - Real-time deadlines: timers, buffer overflows (e.g., CDROM)
- Shared-memory parallel architectures
  - Amdahl's law: minimize time spent in critical sections
  - Parallel execution of non-conflicting I/O

11. Kernel Design

- Interrupts and Exceptions
- Low-Level Synchronization
- Low-Level Input/Output
- Devices and Driver Model
- File Systems and Persistent Storage
- Memory Management
- Process Management and Scheduling
- Operating System Trends
- Alternative Operating System Designs

File Systems

Virtual File System

- Mounting multiple file systems under a common tree
  - `$ man mount`
- Superset API for the features found in modern file systems
  - Software layer below POSIX I/O system calls
  - Full support of UNIX file systems
  - Integration of pseudo file systems: `/proc`, `/sys`, `/dev`, `/dev/shm`, etc.
  - Support foreign and legacy file systems: FAT, NTFS, ISO9660, etc.

Modern File Systems: EXT3 and NTFS

Features

- Transparent defragmentation
- Unbounded file name and size
- Sparse bitmap blocks (avoid waste of resources)
- Support large disks and minimize down-time with *journaling*
  - Maximal protection: support logging of all data and meta-data blocks
  - Minimal overhead: logging of meta-data blocks only (traditional method on SGI's XFS and IBM's JFS)
- Atomic (transactional) file operations
- Access control Lists (ACL)

Modern File Systems: EXT3 and NTFS

Features

- Transparent defragmentation
- Unbounded file name and size
- Sparse bitmap blocks (avoid waste of resources)
- Support large disks and minimize down-time with *journaling*
  - Maximal protection: support logging of all data and meta-data blocks
  - Minimal overhead: logging of meta-data blocks only (traditional method on SGI's XFS and IBM's JFS)
- Atomic (transactional) file operations
- Access control Lists (ACL)

Notes About Linux EXT3

- Compatible with EXT2
- Journalization through a specific block device
- Use a hidden file for the log records

Modern File Systems: EXT3 and NTFS

Features

- Transparent defragmentation
- Unbounded file name and size
- Sparse bitmap blocks (avoid waste of resources)
- Support large disks and minimize down-time with *journaling*
  - Maximal protection: support logging of all data and meta-data blocks
  - Minimal overhead: logging of meta-data blocks only (traditional method on SGI's XFS and IBM's JFS)
- Atomic (transactional) file operations
- Access control Lists (ACL)

Notes About Windows NTFS

- Optimization for small files: "resident" data
- Direct integration of compression and encryption

Disk Operation

Disk Structure

- Plates, tracks, cylinders, sectors
- Multiple R/W heads
- Quantitative analysis
  - Moderate peak bandwidth in continuous data transfers
    E.g., up to 3Gb/s on SATA (Serial ATA), 6Gb/s on SAS (Serial Attached SCSI)
    Plus a read (and possibly write) cache in DRAM memory
  - Very high latency when moving to another track/cylinder
    A few milliseconds on average, slightly faster on SAS

Request Handling Algorithms

- Idea: queue pending requests and select them in a way that minimizes *head movement* and *idle plate rotation*
- Heuristics: variants of the *"elevator" algorithm* (depend on block size, number of heads, etc.)
- Strong influence on process scheduling and preemption: *disk thrashing*

11. Kernel Design

Hardware Support for Memory Management

Segmentation (Old-Fashioned)

- Hardware to separate types of memory (code, data, static, etc.)
- Supported by x86 but totally unused by Linux/UNIX

Paging

- Hardware memory protection and address translation (MMU)
- *At each context switch*, the kernel reconfigures the page table
  - Implementation: assignment to a control register at each context switch
  - Note: this flushes the TLB (cache for address translation), resulting in a severe performance hit in case of scattered physical memory pages
- Use *large pages* for the kernel and for long-lasting memory regions
  - E.g., file system, data base caches, arrays for numerical computing
- Page *affinity* policy for modern cache-coherent architectures

Kernel Mode Memory Management

Classes of Addressable Memory

- Zone allocator

  `ZONE_DMA`: lower 16MB on x86
  `ZONE_NORMAL`: above 16MB and below 896MB on x86 32bits
  `ZONE_HIGHMEM`: above 896MB and below 4096MB on x86 32bits, empty on 64bits

Allocation of Physical Pages

- `kmalloc()` and `kmap()` vs. `malloc()` and `mmap()`
- User processes: contiguous physical memory pages improves performance (TLB usage), but not mandatory
- Kernel: in general, allocation of *lists of non-contiguous physical memory pages*
  - With lower bounds on the size of each contiguous part
  - Note: operates under a specific critical section (multiple resource allocation)

Adaptive Memory Management

Memory Allocation

- *Slab allocator* (original design: Sun Solaris)
  - Caches for special-purpose pools of memory (of fixed size)
  - Learn from previous (de)allocations and anticipate future requests
  - Optimizations for short-lived memory needs
    - E.g., inode cache, block device buffers, etc.
    - Multipurpose buffers from 2^5 to 2^{22} bytes
    - Many other kernel internal buffers
  - `$ man slabinfo` and `$ cat /proc/slabinfo`

11. Kernel Design

Low-Level Process Implementation

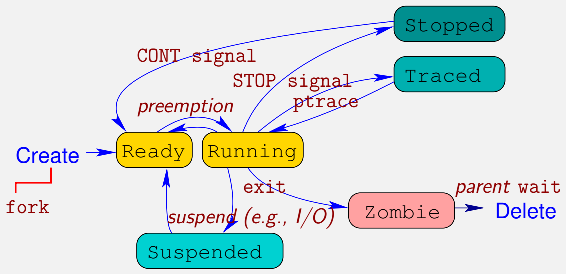
Hardware Context

- Saved and restored by the kernel upon context switches
- Mapped to some hardware thread when running
- Thread *affinity* policy for modern cache-coherent architectures

Heavyweight/Lightweight Processes

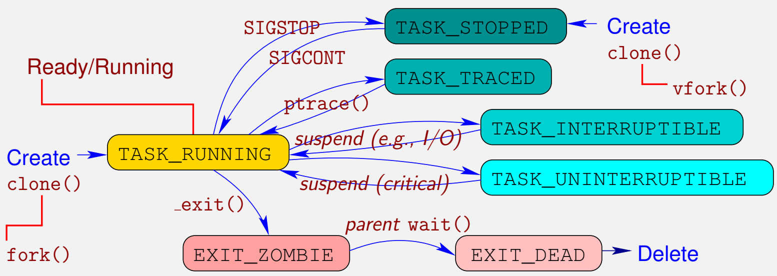
- Implement *one-to-one* model: one *user*-level thread mapped on one *kernel*-level thread
  - Unlike *user*-level threading libraries like the OCaml threads which implement a *many-to-one* model
- Generic `clone()` system call for both threads and processes
  - Setting which attributes are shared/separate
  - Attaching threads of control to a specific execution context

Generic Process States



- `Ready` (runnable) process waits to be scheduled
- `Running` process make progress on a hardware thread
- `Stopped` process awaits a continuation signal
- `Suspended` process awaits a wake-up condition from the kernel
- `Traced` process awaits commands from the debugger
- `Zombie` process retains termination status until parent is notified
- Child created as `Ready` after `fork()`
- Parent is `Stopped` between `vfork()` and child `execve()`

Linux Process States



Notes About Linux

- Context switch does *not* change process state
- Special "non-interruptible" state for critical and real-time I/O

Process Scheduling

Distribute Computations Among Running Processes

- Infamous optimization problem
- Many heuristics... and objective functions
  - Throughput?
  - Reactivity?
  - Deadline satisfaction?
- General (failure to) answer: *time quantum* and *priority*
  - Complex dynamic adaptation heuristic for those parameters
  - `nice()` system call
  - `$ nice` and `$ renice`

Process Scheduling

Scheduling Algorithm

- Process-dependent semantics
  - *Best-effort* processes
  - *Real-time* processes

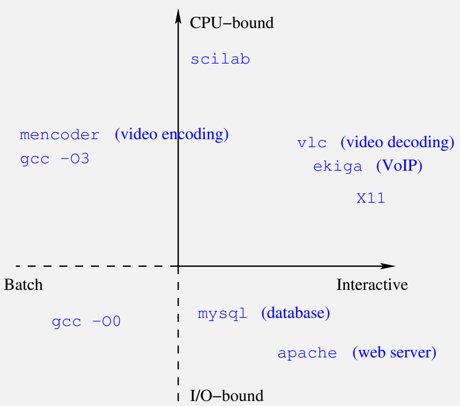
Scheduling Heuristic

- Multiple *scheduling queues*
  - Semantics: split processes according to scheduling algorithm (e.g., preemptive or not)
  - Performance: avoid high-complexity operations on priority queues (minimize context-switch overhead)
- Scheduling *policy*: prediction and adaptation

Scheduling Policy

Classification of Best-Effort Processes

- Two independent features
  - I/O behavior
  - Interactivity



Scheduling Policy

Real-Time Processes

- Challenges
  - Reactivity and low response-time variance
  - Avoid *priority inversion*: priorities + mutual exclusion lead to *priority inversion* (partial answer: *priority inheritance*)
  - Coexistence with normal, time-sharing processes
- `sched_yield()` system call to relinquish the processor voluntarily without entering a suspended state
- Policies: *FIFO* or *round-robin (RR)*

11. Kernel Design

Operating System Trends

Design for Modularity

- Goals
  - Minimize memory overhead (embedded systems)
  - Handle a variety of hardware devices and software services
  - Incremental compilation of the kernel
- *Kernel modules*
  - Linux kernel modules (`/lib/modules/*.ko`) and Windows kernel DLLs
  - Run specific functions on behalf of the kernel or a process
  - Dynamic (un)loading and configuration of device drivers

Operating System Trends

Design for Maintainability

- Downsizing: *microkernel*
  - Execute most of the OS code in user mode (debug, safety, adaptiveness)
  - The kernel only implements synchronization, communication, scheduling and low-level paging
  - User mode system processes implement memory management, device drivers and system call handlers (through specific access authorizations)

Operating System Trends

Microkernels

- Successes
  - Mach: NeXT, MacOS X
  - Chorus (from INRIA project, secure OS)
  - Model for very small kernels (smart cards, eCos)
- Drawbacks
  - Message passing overhead (across processes and layers)
  - Most of the advantages can be achieved through modularization
  - Diminishing returns on full-size kernels
- Extreme: *exokernel* enforce separation and access control only

Operating System Trends

Miscellaneous Trends

- Bundling of a kernel with a variety of higher level libraries, component systems, development kits, graphical interfaces, network tools, etc.
  - If OS $\neq$ kernel, where are the limits of the OS?
- Scalable performance: better support for NUMA
  - Affinity to a core/processor/node, page and process migration
  - Paging and scheduling aware of physical distribution of memory
  - Linux 2.6 as some of the most sophisticated support (see SGI Altix)
- Tuning of kernel policies
  - Custom process and I/O scheduling, paging, migration...
    E.g., IBM Research's K42 linux-compatible kernel
  - Access control models
    E.g., NSA's SELinux

11. Kernel Design

Alternative Designs

General-Purpose Systems

- Single-user "file and device managers": CP/M, MSDOS and Windows 3.1
- Unprotected single-user systems: MacOS 1–9, AmigaOS, OS/2, Windows 95
- Non-UNIX multi-user systems: Multics, VMS, OS/360, Windows NT, Windows 2000 and XP
- Modern workstation/server systems: Windows Vista, Solaris, Linux, MacOS X
- Modern embedded systems: SymbianOS, Blackberry, Windows Mobile, Linux, MacOS X

Real-Time Systems

- Examples of RTOS: pSOS+, VxWorks, VRTX, uiTRON, RTAI

We will quickly survey original features of Windows XP and RTOSes

Windows

Prominent operating system?

Quick Figures – 2007

- 90% of desktops
- 66% of servers
- 24% of smartphones
  - Linux 99% of DSL boxes
  - Linux 52% of web servers and 85% supercomputers

Quick Figures – 2011

- 85% of desktops
- 36% of web servers
- 1% of smartphones sold
  - Linux 99% of DSL boxes
  - Linux 56% of web servers and 91% supercomputers
  - Linux 55% of smartphones sold

Windows

Programmer Interface

- *Win32 API* is the default low-level user interface
- Most of POSIX is supported (see also *cygwin*, *mingw*)
- Note: documentation is not fully available to the public

Windows

File Systems and File Names

- Volume-based file system, no unified mount tree, no VFS
  - Historical legacy from CP/M: `A:`, `C:`, etc.
- Use a *swap file* rather than *swap partition* on UNIX
  - Enhanced flexibility and ease of configuration, lower performance
  - Frequent thrashing problems due to kernel control paths with conflicting memory requirements
- Flat *registry* of environment variables and configuration parameters
  - Combines the equivalent of UNIX's `/etc` and environment variables, plus GNOME's `GConf` files in one single associative table
  - Very fragile database: discouraged manual intervention by Microsoft itself in 1998!

Windows

Processes and Threads

- Multiple execution contexts called *subsystems*
- Multiple hardware threads per subsystem, similar to POSIX threads
  - Threads and subsystems are totally distinct objects, unlike Linux, but closer to other UNIX threading models
- Implementation of the *many-to-many* threading model
  - Support the mapping of multiple *user*-level threads to multiple *kernel*-level threads: `fiber` library

Windows

Processes Communications: Ports and Messages

- Cooperation through Mach-like messages
  - Subsystems have *ports* (for rendez-vous and communication)
    - A client subsystem opens a handle to a server subsystem's *connection port*
    - It uses it to send a connection request
    - The server creates *two communication ports* and returns one to the client
    - Both exchange messages, with or without callbacks (asynchronous message handling)
    - Implementation through *virtual memory mappinig* (small messages) or copying
  - Primary usage: *Remote Procedure Calls* (RPC) called *Local Procedure Calls* (LPC)

Windows

Processes Communications: Asynchronous Procedure Call

- Windows does *not* implement signals natively
- More expressive mechanism: *Asynchronous Procedure Call* (APC)
  - Similar to POSIX *message queues with callbacks* (or handlers)
  - APCs are queued (unlike signals)
  - Can be used to simulate signals (more expensive)

Windows

Thread Scheduling

- The Windows scheduler is called the *dispatcher*
- Similar support for real-time thread domains and time-quantum/priority mechanisms in Linux
- Original features (Windows XP)
  - Strong penalization of I/O-bound processes
  - Extend the time-quantum of the foreground subsystem whose window has graphical focus by a factor of 3!

Real-Time Operating System (RTOS)

Time-Dependent Semantics

- Motivations: enforce delay/throughput constraints
- Hypotheses
  - Short-lived processes (or reactions to events)
  - *Predictable* execution time, known at *process execution (launch)* or *reaction time*
- Tradeoffs
  - *Hard* real time: missing deadlines is not tolerable
  - *Soft* real time: missing deadlines is undesirable, but may happen to allow a higher priority task to complete

Real-Time Process Scheduling

Guarantees

- Periodic system: static *schedulability* dominates flexibility

$$\mathbf{T_i} = \text{execution time}, \ \mathbf{P_i} = \text{execution period:} \ \sum_i \frac{\mathbf{T_i}}{\mathbf{P_i}} < 1$$

- Aperiodic system: online acceptation/rejection of processes
- Beyond preemption and delay/throughput control, RTOSes may offer *reactivity* and *liveness* guarantees

Constraints

- Real-time scheduling requires static information about processes (e.g., bounds on execution time) and may not be compatible with many services provided by a general-purpose OSes

Trends in Real-Time Systems

Real-Time Features in a General-Purpose OS

- Modern OSes tend to include more and more real-time features
  - Predictable media-processing
  - High-throughput computing (network routing, data bases and web services)
  - Support hard and soft real-time
  - Example: *Xenomai* for Linux

Trends in Real-Time Systems

No-OS Approach

- Real-time operating systems are too complex to model and verify
  - Incremental approach: very simple RTOS with fully verifiable behavior
- Yet, *most critical systems do not use an OS at all*
  - Static code generation of a (reactive) scheduler, tailored to a given set of tasks on a given system configuration
  - Synchronous languages: Lustre (Scade), Signal, Esterel
    → main approach for closed systems like flight controllers (Airbus A320–A380)

- See Gérard Berry's lecture-seminars at Collège de France (live, in French)
  `http://www.college-de-france.fr/default/EN/all/inn_tec`

12. Introduction to Virtual Machines

- Modern Applications
- Challenges of Virtual Machine Monitors
- Historical Perspective
- Classification

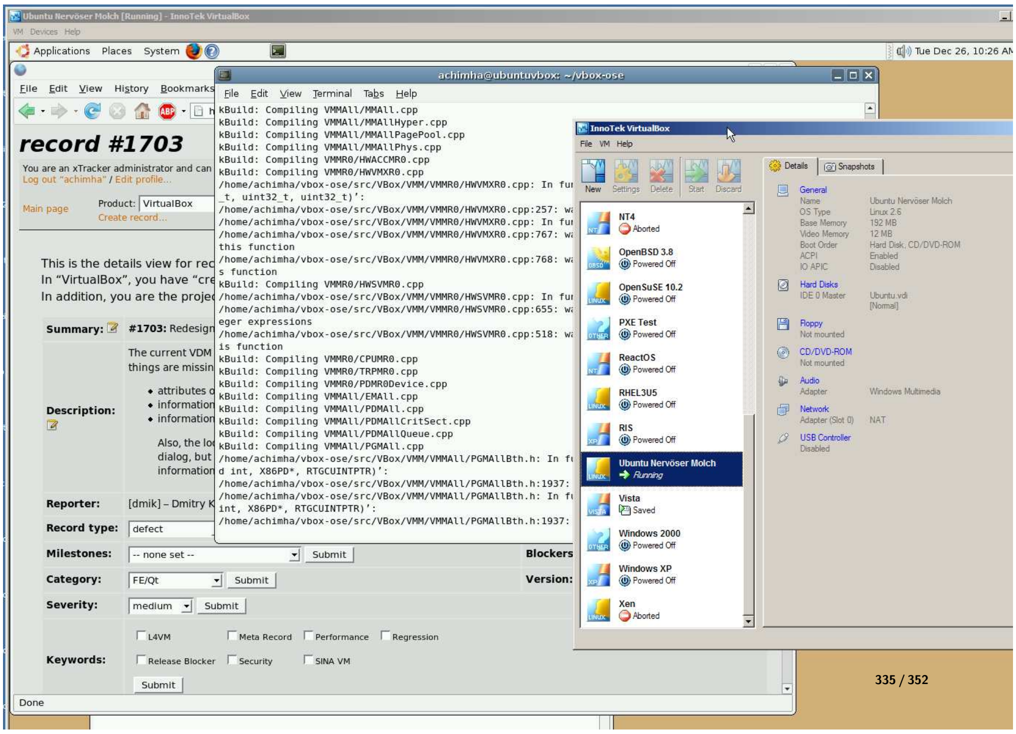
References

- First Attempt to Formalize and Classify: Popek and Goldberg, 1974
- The core element of a virtual machine: the *virtual machine monitor*
  - The virtual machine is analogous to an operating system, and the virtual machine monitor to its kernel

- James E. Smith and Ravi Nair: *Virtual Machines: Versatile Platforms for Systems and Processes*, 2005

12. Introduction to Virtual Machines

- Modern Applications
- Challenges of Virtual Machine Monitors
- Historical Perspective
- Classification



2007: Virtualization Everywhere

Machine-Level Virtualization

- *VMware*, *Parallels Desktop*, *Virtual Box* (Linux, MacOS X, Windows)
  - Driven by the convergence of server, desktop and embedded computing
  - Break some of the artificial constraints imposed by proprietary software
  - VMs replace processes in a secure environments: all communications use high-level distributed system interfaces on top of INET sockets
  - Build feature-rich kernels over small, device-specific kernels

2007: Virtualization Everywhere

Processor-Level Virtualization

- *VMware*, *Virtual Box*, *QEMU*, *Rosetta*
  - Translate machine instructions of the guest processor to run on the host system
  - Fast translation schemes: binary translation, code caches, adaptive translation, binary-level optimization, link-time optimization

2007: Virtualization Everywhere

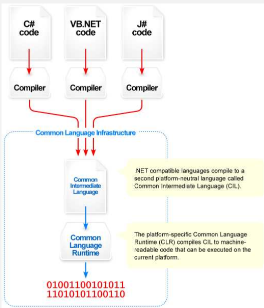
System-Level Virtualization

- Para-Virtualization with *Xen* (Linux only)
  - Ease OS development
  - Customize access control of embedded VMs in a secure environment
  - Virtual hosting, remote administration consoles

2007: Virtualization Everywhere

Language-Level Virtualization

- Abstract machine integrated into the semantics of a programming language
  - *JVM* (Sun Java)
  - *ECMA CLI* (MS .NET)
- Features
  - Portability, code size improvements, original dynamic optimizations
  - High-productivity features (garbage collection, distributed components)
  - Sandbox (robustness, security management, fault-tolerance)



12. Introduction to Virtual Machines

Virtual Machine Monitor

Classical Challenges

- Influence of the *guest-host* relationship
  - Homogeneous: intercept any *guest*-specific action to redirect it to the *host*'s interface
  - Heterogeneous: instruction-set *emulation* and *binary translation*
- Excessive memory usage
- Project drivers of the guest operating system to host devices

Virtual Machine Monitor

Virtualization of Priviledged Code

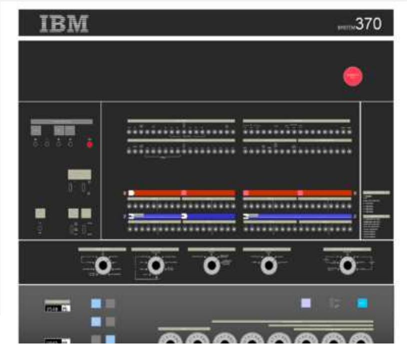
- Common issues
  - Memory-mapped I/O
  - Exceptions and interrupts
  - Esoteric machine language instructions
- Good design: IBM System/360 instruction set and I/O architecture
- Bad design: Intel x86 and IBM PC I/O
  - Port I/O, accesses to system buses, memory-mapped I/O, control registers and exceptions/interrupts *could not* be reconfigured to (selectively) trigger *host* exceptions
  - Require a conservative emulation layer to execute priviledged code
  - Fixed in the Core Duo 2 processor: *native virtualization*

12. Introduction to Virtual Machines

Historical Perspective

IBM **VM System/370**: 1967

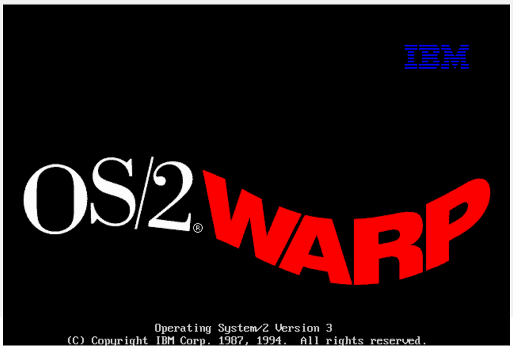
- First virtual machine
  - Offer long-term portability of System/360 applications over a wide range of machines and peripherals, although the processor's machine instructions where quite different
- Implementation: binary translation and emulation of foreign code
  - Unpriviledged code
  - Compatible guest and host system API



Historical Perspective

IBM **OS/2**: 1987

- Provide modern OS features to legacy MSDOS applications
  - Multitasking (non-preemptive at first)
  - Virtual memory (protection and management of more than 640kB of RAM)
- Implementation: embed a 1MB MSDOS memory image in the virtual memory frame of a single process (called task)
- Initially supported by Microsoft... then came Windows 3.1 and then NT



Historical Perspective

Apple (and Transitive) **Rosetta**: 1994 and 2006

- Execute Motorola 680x0 code on PowerPC
  - Emulation and some binary translation
  - User code only
- Execute PowerPC code on x86
  - Priviledged code, support for a full-fledged UNIX OS
- Compatible guest and host system API, low-overhead implementation
- Original project: *DAISY* in 1992 (IBM Research, PowerPC → VLIW instruction set)
- Enhancements
  - Full system virtualization (heterogeneous): *VMware*
  - Performance (emphasis on dynamic optimization), multi-OS (Linux, HPUX, Windows): *IA36EL* (Intel)

Historical Perspective

Transmeta Crusoe: 2000

- Translate x86 code to a VLIW ISA
  - Binary translation code embedded on the chip itself: *Code Morphing*
  - Pros: low overhead (avoids instruction cache pollution, dedicated hardware), energy and bandwidth savings (on-chip memory accesses)
  - Cons: peep-hole optimizations only, hard to maintain precise exception semantics
- Discrete advantage: fix hardware bugs, shorter testing (most expensive)
- Untold advantage: hide original processor specifications, including energy management and proprietary VLIW instruction set

12. Introduction to Virtual Machines

Taxonomy of Virtual Machine Monitors

Software Implementations

- *Hypervisor*: most general case, when native virtualization is not possible or to implement a *sandbox*
- Emulation and full-system virtualization: e.g., *QEMU*, *VMware*
- Computer architecture simulation (cycle-accurate or transaction-level): e.g., *Simics* (Chalmers), *UNISIM* (INRIA, Princeton, UPC)
- Binary translation, code cache and dynamic optimization: e.g., *DAISY* (IBM), *Dynamo* and *DELI* (HPLabs) *Rosetta* (Apple), *IA32EL* (Intel)
- Para-virtualization: resource sharing, security and sandboxing, e.g., *Xen* or *User Mode Linux*

Taxonomy of Virtual Machine Monitors

Hardware Implementations

- Homogeneous instruction sets
  - Native virtualization with a *lightweight hypervisor*: "trap" on specific instructions or address ranges, e.g., *Parallels Desktop* or *kvm* (with QEMU) on Intel Core 2 and later x86 processors
  - Reduce exception overhead and cost of switching to kernel mode
- Heterogeneous instruction sets
  - Support to accelerate instruction decoding (PowerPC)
  - Additional instruction pipeline stages (x86 $\longrightarrow$ internal RISC microcode)
  - Hybrid binary translation to reduce overhead: *Code Morphing* (Transmeta Crusoe),

Virtualization Stack

Intricate Example

Java application
↓ Just-in-time compilation and Java virtual machine
MacOS X PowerPC
↓ Binary Translation (Rosetta)
MacOS X x86
↓ Full system virtualization (Parallels Desktop)
Linux x86
↓ Para-virtualization (Xen)
Linux x86
↓ Binary translation (Transmeta Code Morphing)
Transmeta Crusoe (VLIW)

Merci !