Unix Inter-Process Communication

Jiwei WANG
Jiwei.Wang@ywywy.com

January, 1999

Course Outline

- Unix IPC Overview
- Signal
- Shared Memory and Semaphore
- Socket and TCP/IP
- Pipe and Message queue
- Version Control with CVS
- Make and Makefile
- Cross Reference with id-util
- Unix File System Layout

1. Unix IPC Overview

- What is a process?
- What's in a process?
- Memory Layout of a Process
- Finding Out the IDs
- What is Inter-Process Communication?
- IPC Mechanisms
- IPC Properties
- Unix Standards
- Unix Implementations
- References

What is a Process?

What is a process?

A *program* is an executable binary file. An executing instance of a program is called a *process*.

Example:

/bin/ls is a binary program. Executing,

% ls

causes a process to be created and the current directory listing is shown as the result.

How to create a process

The only way to create a process is to invoke *fork*() system call.

All processes in a Unix system are spawned directly or indirectly by process *init*, (pid 1).

What's in a process?

A process has,

- A protected memory region. (Details in the next slide.)
- 6 IDs:

  Process ID, referred as PID: a non-negative integer, unique in the system.

  Parent PID: PID of the process that launches the current process.

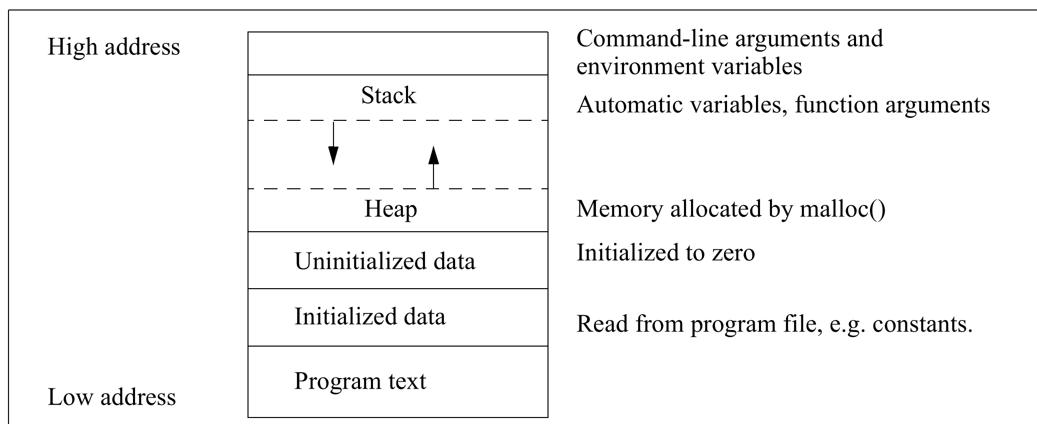
  User ID: the user id of the real process owner.

  Group ID: the group id of the real process owner.

  Effective user ID and effective group ID: used for file access permission check.
- Current working directory.
- Some system resources: such as file descriptors, shared memory, child process, CPU time limits, etc.

Memory Layout of a Process

Every process has its own memory region in an OS with memory protection.



Dereferencing a pointer outside the process memory causes a segmentation violation.

Finding Out the IDs

System calls to find out the IDs of a process:

```
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
pid_t getuid(void);
pid_t getgid(void);
pid_t geteuid(void); /* get effective uid */
pid_t getegid(void); /* get effective gid */
```

Process vs. Thread

Threads are multiple and interleaved independent execution points within a process. Each thread shares the same memory region of the process, but has its own stack.

Threads are “light-weight processes”. Thread creation can be 10~100 times faster than process creation.

What is Inter-Process Communication?

Definition

Inter-Process Communication (IPC) refers to various ways of *message passing* and *synchronization* between different processes.

IPC and Networking

- IPC:

  Message passing between processes on the same machine.

- Networking:

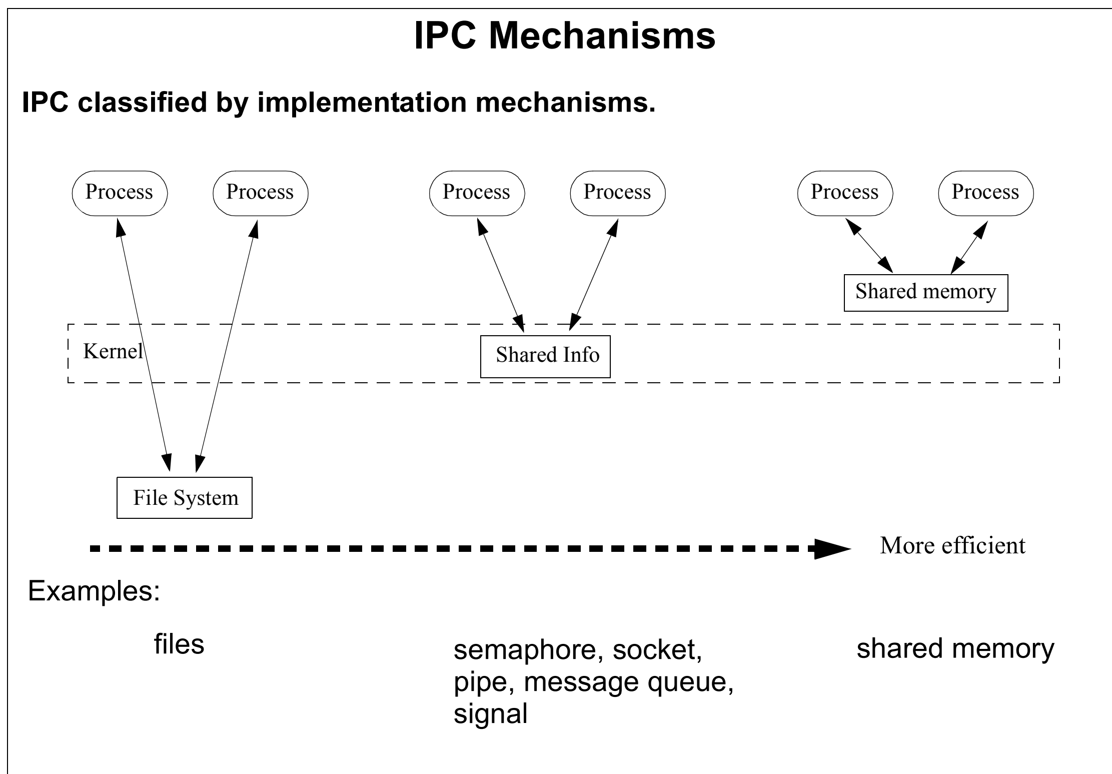
  Message passing between processes on different machines (across a network).

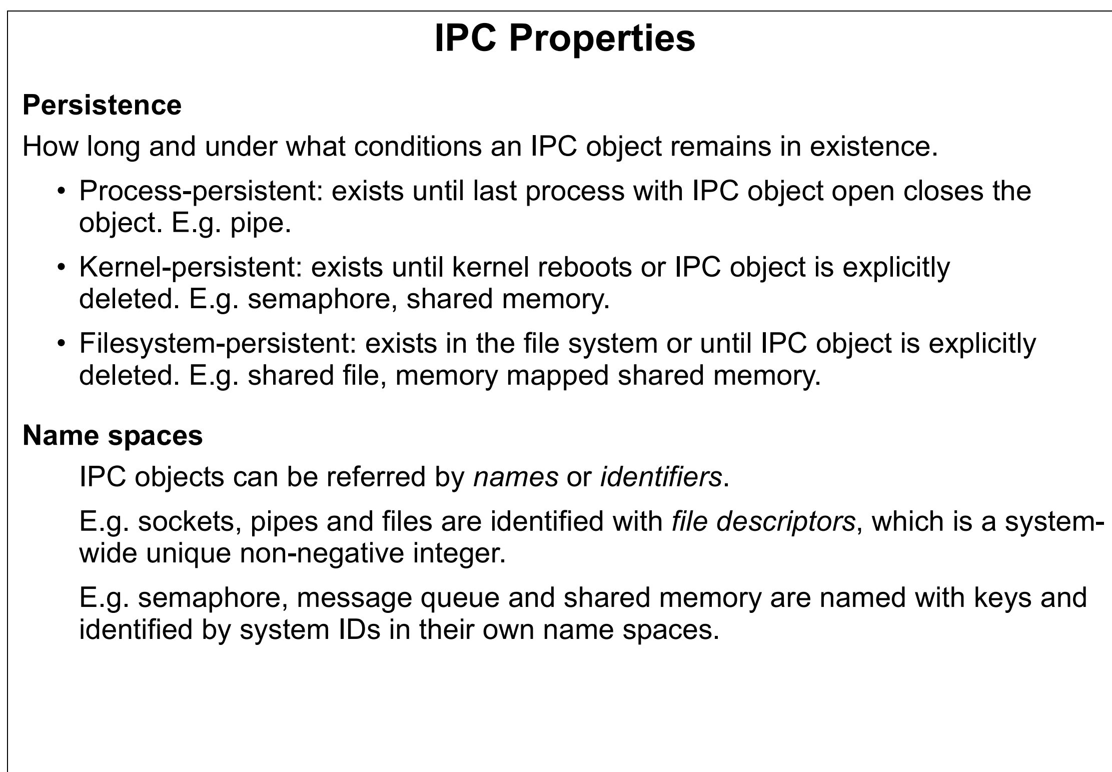
Examples

- Messages being passed from process “ls” to process “more”, via a pipe.

  % ls -las | more

- Closing an Xterm shuts down the csh program, via signal.
- Using web browser to access a WWW page, via TCP sockets.


IPC Mechanisms
IPC classified by implementation mechanisms.
Process
Process
Process
Process
Process
Process
Shared memory
Kernel
Shared Info
File System
More efficient
Examples:
files
semaphore, socket, pipe, message queue, signal
shared memory


IPC Properties
Persistence
How long and under what conditions an IPC object remains in existence.
• Process-persistent: exists until last process with IPC object open closes the object. E.g. pipe.
• Kernel-persistent: exists until kernel reboots or IPC object is explicitly deleted. E.g. semaphore, shared memory.
• Filesystem-persistent: exists in the file system or until IPC object is explicitly deleted. E.g. shared file, memory mapped shared memory.
Name spaces
IPC objects can be referred by names or identifiers.
E.g. sockets, pipes and files are identified with file descriptors, which is a system-wide unique non-negative integer.
E.g. semaphore, message queue and shared memory are named with keys and identified by system IDs in their own name spaces.

IPC Properties (Cont....)

Permission

The same as Unix file access permission, that is, readable, writable, executable for user, group, and others.

Throughput

Throughput is referred to the bandwidth of the message passing.

E.g. TCP/IP sockets over a 10-Base Ethernet cannot transmit more than 1MB/s.

Latency

Latency is referred to the delay incurred when passing a message.

Most IPC communication can be completed in micro-seconds, e.g. semaphore, signal. Some can be done in nano-seconds, e.g. shared memory.

Limitations

System limits for the number of IPC objects.

File descriptor based IPC cannot exceed the total system file descriptors.

Unix Standards

Standards define interface specifications.

Three influential standards:

- ANSI C standard (1989)

  Defines a set of C library functions. These C library functions are supported by all Unix implementations.

- IEEE Portable Operating System Interface (POSIX)

  1003.1 (1988, 1990): System Application Program Interface (API) for C.

  1003.2 (1992): Shell and Utilities.

- X/Open (a consortium of vendors and users)

  XPG3 (X/Open Portability Guide, Issue 3) (1989).

  “X/Open Single Unix Specification” (1995, 1998), also known as “Spec 1170” (system interfaces + headers + commands), or “Unix 95” and “Unix 98”.

Unix Implementations

All mainstream Unix systems derive from three implementations.

- AT&T System V Release 4. E.g. HP-UX, AIX, Ultrix, Solaris.
  Contributions: System V IPC, shared libraries, streams.
- UC Berkeley BSD 4.3 (Berkeley Software Distribution). E.g SunOS 3.
  Contributions: TCP/IP, socket, NFS, RPC.
- GNU & FSF (Free Software Foundation) Linux.
  Contributions: GNU utilities
- Real-time OS (RTOS), such as WindRiver VxWorks, Sun Chorus, and QNX.

Most Unix systems conform to some version of Posix and X/Open, but all offer additional features.

From IPC point of view

System V IPC and Berkeley socket are the most common inter-process communication mechanisms.

Choosing the Right IPC

The decision should be based on:

- Location of the process: Sharing data between processes on the same machine or over the network.

  Signal, pipe, shared memory, semaphore, and message queues work for processes on the same machine.

  Socket works for both situation.

  File and shared memory can work across network providing NFS and special program is running to replicate them.

- Performance requirement and Hardware architecture.

  E.g. Shared memory is more efficient than other IPC on SMP machines.

- Level of abstraction.

  Inter-operability between processes can be achieved by introducing a layer of abstraction, such as CORBA, DCOM and DCE.

Change Effective uid and gid

The effective user ID and group ID of a program can be changed at runtime from those of the executing user to those of the owner of the file. This grants additional file access permissions to the program.

The Setuid bit and Setgid bits are changed by adding 4000 and 2000 in the *chmod* command respectively.

Example

- Copy /bin/csh to /tmp
- Log in as root and execute commands,

```
% chown root:sys /tmp/csh
% chmod 6755 /tmp/csh
% ls -l /tmp/csh
-rwsr-sr-x   1 root     sys       353944 Jan 19 23:23 csh
```

- Create a text file, say /tmp/foo, read-writable only by root.
- Log in as another user. Try to read /tmp/foo, it should show permission denied.

- Execute the following command,

```
% /tmp/csh
% id
uid=1000(jiwei) gid=100(users) euid=0(root) egid=3(sys) groups=100(users)
```

- Now try to read /tmp/foo again. You can also modify /tmp/foo.

By now, you system security is entirely compromised. Any user can run /tmp/csh and modify system file, e.g. /etc/passwd file to make himself/herself a root. This is a well-known way of tempering a Unix system.

Please remember to remove /tmp/csh after the exercise.

Find out Process Size

The *ps* utility

On SunOS, Solaris and Linux,

% ps -aux

On HP-UX,

% ps -ef

USER	PID	%CPU	%MEM	SIZE	RSS	TTY	STAT	START	TIME	COMMAND
jiwei	1101	0.0	5.9	1296	884	p1	S	23:00	0:00	bash

RSS: Resident set size; kilobytes of program in memory.

SIZE: the process size in RAM and virtual memory, in KB.

Using *top*.

top is a free utility which provides an ongoing look at processor activity in real time.

% top

PID	USER	PRI	NI	SIZE	RSS	SHARE	STAT	LIB	%CPU	%MEM	TIME	COMMAND
1117	root	17	0	1068	864	468	R	0	27.5	5.8	0:04	xterm

System Administration References

"Unix System Administration Handbook" 2nd Edition, by Evi Nemeth, Garth Snyder, Scott Seebass, Trent R. Hein. Prentice Hall 1995. ISBN: 0-13-151051-7.

The most comprehensive guide to UNIX system administration. Covers six most popular variants of UNIX: Solaris, HP-UX, IRIX, SunOS, OSF/1, and BSDI.

Unix System Administration Independent Learning

A self-study course and resources for Unix system administration

http://www.uwsg.indiana.edu/usail/

Unix Guru Universe

Everything you want to know about Unix system administration.

http://www.ugu.com/

"Red Hat Linux Secret" 2nd Edition, by Naba Barkakati, IDG Books Worldwide 1998; ISBN: 076453175.

An excellent book on RedHat Linux.

GNU utilities

ftp://uiarchive.cso.uiuc.edu/pub/gnu/

References

"Advance Programming in the Unix Environment" by W. Richard Stevens. Addison-Wesley 1992. ISBN: 0-201-56317-7.

One of the best introductory book to Unix system and application programming. Clear and Lucid. With great examples.

"Unix Network Programming Volume 2: Interprocess Communication", 2nd Edition, by W. Richard Stevens. Addison-Wesley 1999. ISBN: 0-13-081081-9.

This is an essential reference to Unix IPC. It offers comprehensive guide to every form of IPC, including message passing, synchronization, shared memory, and Remote Procedure Calls (RPC).

"Unix Network Programming Volume 1: Networking APIs", 2nd Edition, by W. Richard Stevens. Addison-Wesley 1998. ISBN: 0-13-490012-X.

Another essential reference to Unix networking. It covers everything that programs communicate over networks, including TCP, UDP, broadcasting/multicasting, routing sockets, IPv4 and IPv6, and raw sockets, plus a section covering Posix threads.

References (Cont....)

"Magic Garden Explained: The Internals of Unix System V Release 4: An Open Systems Design" by Berny Goodheart, James Cox. Prentice Hall 1994; ISBN: 0-13-098138-9.

An authoritative, in-depth description of the internal workings and programmatic interface to the UNIX SVR4 kernel.

"The Design of the Unix Operating System" by Maurice J. Bach, Prentice Hall 1986; ISBN: 0-13-201799-7.

This is the first, and still, the most comprehensive book to describe the UNIX System V kernel--the internal algorithms, the structures and their relationship to the programming interface.

"Operating Systems: Internals and Design Principles" 3rd edition, by William Stallings, Prentice Hall 1997; ISBN: 0-13-887407-7.

This book offers a comprehensive up-to-date description of operating systems with an emphasis on internals and design issues. Excellent presentation on virtual memory, file system and distributed system. Covers both Windows NT and Unix.

2. Signal

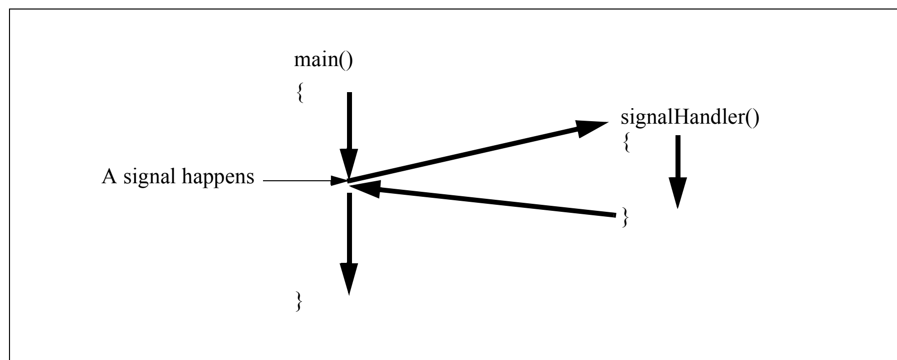
- Signal Concept
- Signal Categories
- Signal Properties
- ANSI C signal()
- signal() Example: ansiSig.c
- ANSI signal() Caveats
- Related System Calls
- System Commands
- Example Programs
- POSIX Signal Mask
- POSIX Signal Processing
- Posix Signal Example
- Posix Signal Caveats

Signal Concept

Definition

Signals are software interrupts, providing a means to handle asynchronous events.

Signal handling



Signal Categories

Signals can be classified into 5 categories.

- Process control.

  SIGHUP: hang-up, usually sent by the parent process to child processes when terminating.

  SIGKILL: kill (cannot be caught of ignored)

  SIGTERM: Software termination signal from kill

  SIGABRT: Process abort signal
- Job control.

  SIGCHLD: Child process has stopped or terminated.

  SIGSTOP: Stop signal (cannot be caught or ignored)

  SIGTSTP: Interactive stop signal

  SIGCONT: Continue if stopped

  SIGTTIN: Read from control terminal attempted by a member of a background process group

  SIGTTOU Write to control terminal attempted by a member of a background process group
- Environment change.

  SIGWINCH: indicates window size change.

- Hardware exception.

  SIGINT: Interrupt, generated by Ctrl-C

  SIGQUIT: quit, generated by Ctrl-\ Ctrl-break

  SIGILL: Illegal instruction (not reset when caught)

  SIGTRAP: trace trap (not reset when caught)

  SIGIOT: IOT instruction

  SIGEMT: EMT instruction

  SIGFPE: Floating point exception

  SIGBUS: bus error

  SIGPWR: power state indication

  SIGIO asynchronous I/O

  SIGURG urgent condition on IO channel
- Software condition.

  SIGSYS: bad argument to system call

  SIGPIPE: write on a pipe with no one to read it

  SIGALRM: alarm clock

  SIGSEGV: Segmentation violation

  SIGVTALRM: virtual timer alarm

  SIGPROF: profiling timer alarm

Signal Properties

Persistence

Process-persistent.

Name space

PID and signal numbers.

Permission

Processes of the same group can send each other signals. Superuser (root) can send signal to all processes.

Latency

A few hundred micro-seconds on general-purpose Unix (e.g. HP-UX, Linux) or a few micro-seconds on a RTOS (3.3 us on Pentium-166, QNX).

Limitations

A limited number of signals (normally 32). Two of them can be defined by users.

ANSI C signal()

ANSI C defines the simplest signal handling function.

```
#include <signal.h>
void (*signal(int sig, void (*func)(int)))(int);
```

signal() instantiates a signal handler for the given signal and returns the signal handler instantiated earlier.

Example

```
static void signalHandler(int); /* signal handler function */
signal(SIGHUP, &signalHandler); /* instantiate the signal handler */
```

Macros SIG_ERR, SIG_DFL and SIG_IGN.

```
signal(sig, SIG_DFL); /* reset to default signal handler. */
signal(sig, SIG_IGN); /* ignore the signal. */
```

This is normally sufficient for basic signal handling.

signal() Example: ansiSig.c

```
int receivedSignal_g = 0;
int receivedSignalNo_g = 0;
static void signalHandler(int);

int main(int argc, char* argv[])
{
   signal(SIGHUP, &signalHandler);
   while (receivedSignalNo_g != SIGTERM) {
      if (receivedSignal_g) {
         printf("Received signal: %d\n", receivedSignalNo_g);
         receivedSignal_g = 0; receivedSignalNo_g = 0;
      }
      sleep(60);
   }
}


static void signalHandler(int signalNumber)
{
   signal(signalNumber, signalHandler); /* re-establish the signal handler. */
   receivedSignal_g = 1;
   receivedSignalNo_g = signalNumber;
}
```


ANSI signal() Caveats

Be cautious about

- Signal handler is automatically reset to default after the signal occurs. Re-establish signal handler after receiving it.
- Signal can be occur during the handling of a signal. This causes signal() to be unreliable.
- signal() overwrites the previous signal handler. May need to consider signal handler chaining.
- Some system calls are not reentrant. These are mainly the “slow” system calls, e.g select(), I/O calls. Signals interrupt the current operation and causing it to return an error.

Related System Calls

Sending a signal

```
#include <signal.h>
int raise(int sig);             /* send a signal to itself */
int kill(pid_t pid, int sig);   /* send a signal to process PID */
```

kill() can be used to determine whether a process has terminated.

```
kill(aPid, 0);
```

If *kill()* returns -1 and *errno* is set to ESRCH, the aPid process is no longer around.

System call *abort()* is implemented with *raise(SIGABRT)*, sending SIGABRT to itself.

Waiting for a signal

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

System call *sleep()* is implemented with *alarm()* and *pause().*

System Commands

kill() - sends a signal to a process, or by default, terminates a process.

```
kill [-s signame] pid ...
kill [-s signum] pid ...
kill -l                 // list all signal values.
```

Obsolescent Versions:

```
kill -signame pid ...
kill -signum pid ...
```

Example Programs

ansiSig.c: Simple ANSI signal handling program.

- Start the program with: % ./go
- Send a SIGHUP to the program: % kill -SIGHUP pid
- Resize the window and observe the output.
- Terminate ./go with Ctrl-C.
- Modify ansiSig.c to catch signal SIGINT. Recompile it. The program can no longer be terminated with Ctrl-C. Use "kill -9 pid" to stop the program.

alarm.c: How to set an alarm to wake the process up. Using alarm() and pause().

- Launch the program and observe the outputs on epoch seconds.

sendSig.c and recSig.c : Measuring signal latency.

- Launch ./recSig in one window; Send a signal to it with "./sendSig pid 1" from another window; Compare the time stamps output by the programs.

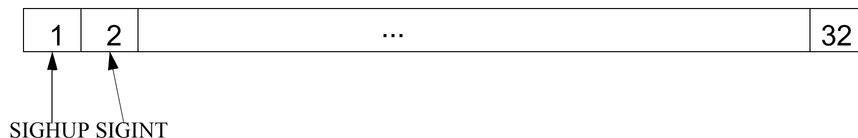
chaining.c: Example of chaining signal handlers.

- Run the program; Send it a signal and observe the output; Modify program to remove chaining and observe the results.

POSIX Signal

Signals are queued for a process. They can be *blocked* and remain in *pending* state until they are *unblocked* by the process. The size of the queue is implementation dependent.

Signal set is a mask for the signals. An example of a 32 bits signal set.



Signal set manipulation

```
int sigemptyset(sigset_t *set); /* initialize the set to unblock all signal */
int sigfillset(sigset_t *set); /* initialize the set to block all signal */
int sigaddset(sigset_t *set, int signum); /* block a given signal in a set */
int sigdelset(sigset_t *set, int signum); /* unblock a given signal in a set */
int sigismember(const sigset_t *set, int signum); /* test if a signal is blocked in the set */
```

POSIX Signal Processing

The sigaction struct

```
struct sigaction {
        void (*sa_handler)(int); /* address of a signal handler */
        sigset_t sa_mask; /* block certain signal during the execution of the handler */
        int sa_flags; /* signal options */ }
```

Signal handling functions

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
    /* install a signal handler for a signal, and return the previous sigaction struct.*/
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
    /* modify the signal mask of the current process. how can be SIG_BLOCK,
       SIG_UNBLOCK, SIG_SETMASK */
int sigpending(sigset_t *set);
    /* returns a set of signals which are blocked */
int sigsuspend(const sigset_t *mask);
    /* suspend the process until the specified signals (in the mask) occur */
```

Signal handler will not change after a signal occurs. Signal mask is replaced by the sigaction() during signal handling and restored afterwards.

Posix Signal Example

```
static void signalHandler(int);

int main(int argc, char* argv[])
{
    struct sigaction act;
    struct sigaction oldact;
    act.sa_handler = &signalHandler;
    sigfillset(&act.sa_mask); /* block all signal during signal handler */
    act.sa_flags = 0;
    sigaction(SIGHUP, &act, &oldact);

    while (receivedSignalNo_g != SIGTERM) {
        if (receivedSignal_g) { receivedSignal_g = 0; receivedSignalNo_g = 0; }
        sleep(60);
    }
}

static void signalHandler(int signalNumber)
{
    receivedSignal_g = 1;
    receivedSignalNo_g = signalNumber;
}
```

Posix Signal Caveats

- SIGKILL and SIGSTOP cannot be caught.
- Never do thing that could block (e.g. I/O) in a signal handler. This may make the program unresponsive to other events. Signal handler is meant to set a few flags only.
- signaction() overwrites the previous signal handler. May need to consider signal handler chaining.
- In some Unix system, multiple occurrences of the same signal could be delivered only once.

3.Semaphore

- Semaphore Concept
- Semaphore Programming Model
- Semaphore Properties
- System V Semaphore
- System V Semaphore Calls
- Flaw of System V Semaphore
- System V Semaphore Initialization
- Posix Semaphore
- Posix Named Semaphore
- Posix Memory-based Semaphore
- HP-UX 10.x Memory-based semaphore
- Related System Command
- Caveats
- The Dining Philosophers Problem
- Example Programs

Semaphore Concept

Definition

A *semaphore* is a primitive used to provide synchronization between processes on a Unix system (or threads in a process).

A *binary semaphore* can assume only the values 0 or 1.

A *counting semaphore* can take any value between 0 and N, where N is a non-negative number (greater than 1).

Operations on a Semaphore

- *Create* a semaphore

  Create a semaphore with an initial value.
- *Wait* for (or *lock*) a semaphore

  Test the value of the semaphore, wait (or block) if the value is less or equal to 0, and then decrements the value by 1 once it's greater than 0.
- *Post* to (or *unlock*) a semaphore

  Increment the semaphore value by 1.
- *Destroy* a semaphore

  Remove the semaphore from kernel.

Semaphore Programming Model

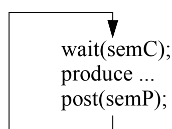
Mutual exclusion

Controlling exclusive access to a common shared resource, to prevent data corruption. For example, writing to a shared memory space between two processes has to be mutually exclusive. Use binary semaphore.

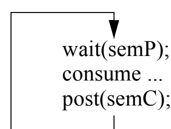
Synchronization

Synchronizing process executions. One process has to wait for one (or more) processes to complete their job before it can proceed. Use binary or counting semaphore.

Producer Process



Consumer Process



semC initialized to 1
semP initialized to 0

Resource control

Controlling the total usage of a shared resource. This ensures that the use of a certain resource is kept under the maximum allowable units. Use counting semaphore.

Semaphore Properties

Persistence

System V semaphore: kernel-persistent.

Posix semaphores: process-persistent, kernel-persistent or filesystem-persistent.

Name space

Semaphore Keys and IDs (System V). Keys are used when creating the semaphore and IDs are used in all the references after. File path (Posix semaphore).

Permission

Permission can be specified by the process, the same as a file.

Latency

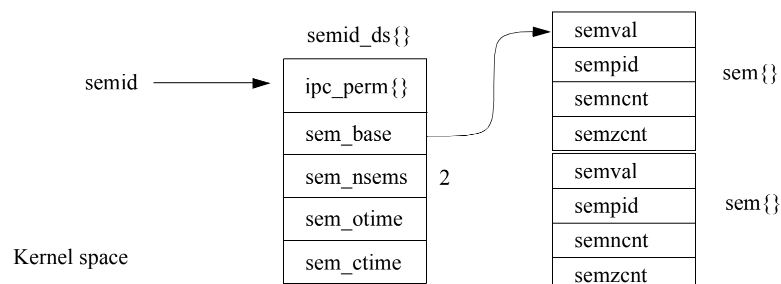
A pair of System V lock() and unlock() operations typically takes less than 20 micro-seconds on a general-purpose Unix (e.g. HP-UX, Linux). Posix memory-based semaphore takes less than 100 nano-seconds.

Limitations

Available on the same machine only.

System V Semaphore

System V semaphore is a set of counting semaphores (upto 25).



semget() creates the set of semaphore and returns an unique ID.

int semget(key_t key, int nsems, int oflag);

Key is a unsigned long integer. nsems specifies the number of semaphores.

oflag can be a file permission (e.g. 0666), bitwise-ORed with IPC_CREAT, or IPC_CREAT | IPC_EXCL.

IPC_CREAT: create if it does not already exist, otherwise proceeds without error.

IPC_CREAT | IPC_EXCL: create if it does not already exist, otherwise return an error.

System V Semaphore Calls (cont...)

semop() operates on the semaphore.

int semop(int semid, struct sembuf *sops, unsigned nsops)

struct *sembuf* including the following members:

```
short sem_num; /* semaphore number: 0 = first */
short sem_op;  /* semaphore operation */
short sem_flg; /* operation flags */
```

sem_num specifies to which semaphore the operation applies.

sem_op is an integer to be added onto the existing value.

Flags recognized in sem_flg are IPC_NOWAIT and SEM_UNDO. If an operation asserts SEM_UNDO, it will be undone when the process exits.

nsops specifies the number of sembuf.

Semaphore System Calls (cont...)

semctl() performs various control operations on a semaphore.

int semctl (int semid, int semnum, int cmd, union semun arg)

Common commands:

IPC_STAT	Obtain the current semaphore set data structure.
IPC_SET	Write to the semaphore structure
IPC_RMID	Remove the semaphore set and awaken all waiting processes.
GETVAL	The system call returns the value of semval
SETVAL	Set the value of semval
GETPID	The system call returns the value of sempid

Flaw of System V Semaphore

Flaw that cause great inconvenience

Semaphore creation and initialization are done with two different commands. Causing a potential race condition.

Semaphore structure initialization during creation.

ipc_perm{}: uid, gid to the creating process effective uid, gid.

sem_base: points to the first sem{} structure.

sem_nsems: sets to nsems.

sem_otime: sets to 0 and won't change until a semop() is executed.

sem_ctime: sets to current time.

sem{}: *not initialized*.

System V Semaphore Initialization

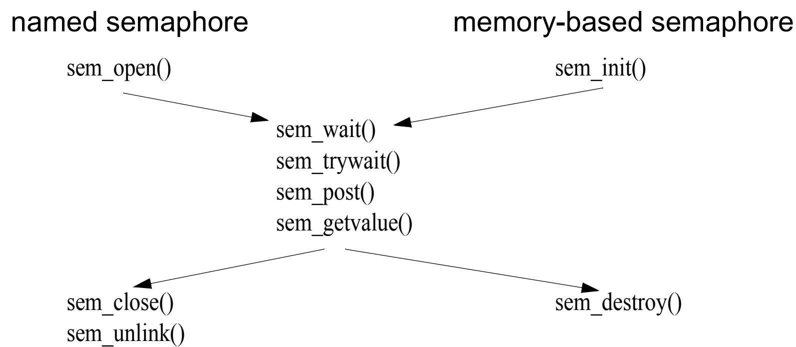
Proper initialization (sema.c)

```
    if ((id = semget(key, 3, IPC_CREAT | IPC_EXCL | 0666)) >= 0) {
      /* success, we're the first so initialize */
      semctl_arg.val = 0;
      semctl(id, 0, SETALL, semctl_arg);
    } else if (errno == EEXIST) {
/*Grab the id, no creation, also protect us from the semaphore being deleted */
      if ((id = semget(key, 3, 0666)) < 0) {
        if (errno == ENOENT) goto again;
      }
      semctl_arg.buf = &seminfo;
      for (i = 0; i < MAX_TRIES; i++) {
        semctl(id, 0, IPC_STAT, semctl_arg);
        if (semctl_arg.buf->sem_otime != 0)
          break;
        sleep(1);
      }
      if (i == MAX_TRIES) { /* after slept MAX_TRIES, semaphore is still not initialized */
        perror("semget OK, but semaphore not initialized");
        return -1;
      }
    }
```

Posix Semaphore

Posix defines named and memory-based semaphores. There's no requirement to maintain them in kernel.

Posix system calls are much simpler than those of System V.



There is no related system command for the Posix semaphores.

Posix Named Semaphore

Posix named semaphore shares the name space of the file system and can be implemented with files. So it can be filesystem-persistent or kernel-persistent.

```
sem_t *sem_open(const char *pathName, int oflag, mode_t mode, unsigned int value);
```

pathName: full valid path to a file.

oflag: can be

- 0: just open,
- O_CREAT: create and initialize if it doesn't exist, otherwise just open,
- O_CREAT | O_EXCL: create and initialize if it doesn't exist, otherwise return error.

mode: permission bits. Not require if oflag is 0.

value: initial value. Not require if oflag is 0.

```
int sem_close(sem_t *sem);
```

/* reduce reference count to the semaphore by 1 and destroy the semaphore if the count is 0. sem_close() is automatically done if the process exits. */

```
int sem_unlink(sem_t *sem);
```

/* remove the name of the semaphore. It has no effect on the existing reference, only prevents subsequent calls from succeeding.*/

Posix Memory-based Semaphore

Posix memory-based semaphore is initialized with sem_init().

```
int sem_init(sem_t *sem, int shared, int value);
```

sem: pointer to a sharable memory.

shared: 0 indicate within the same process, otherwise between processes.

sem_init() always initializes the semaphore with the given value.

```
int sem_destroy(sem_t *sem); /* remove the semaphore. */
```

Posix memory-based semaphore can be process-persistent (when shared is 0) or kernel-persistent.

```
int sem_wait(sem_t *sem);
   /* sleep if the semaphore is locked, otherwise decrement it by 1. */
int sem_trywait(sem_t *sem);
   /* return -1 immediately if the semaphore is locked, otherwise decrement it by 1. */
int sem_post(sem_t *sem); /* increment the semaphore by 1. */
int sem_getvalue(sem_t *sem); /* return the current value of the semaphore. */
```


HP-UX 10.x Memory-based semaphore

HP-UX 10.x also supports HP's own version of memory based semaphore which,

- supports binary semaphore only, and
- is about 10 times faster than System V semaphore.

```
#include <sys/mman.h>
msemaphore *msem_init(msemaphore *sem, int initial_value);
int msem_lock(msemaphore *sem, int condition);
   /* condition can be MSEM_IF_NOWAIT or zero. When MSEM_IF_NOWAIT is set and the
      semaphore is locked, the function returns with an error. */
int msem_unlock(msemaphore *sem, int condition);
   /* If condition is zero, the semaphore will be unlocked, whether or not any other processes
      are currently attempting to lock it. If condition is MSEM_IF_WAITERS, and some other
      process is waiting to lock the semaphore or the implementation cannot reliably deter-
      mine whether some process is waiting to lock the semaphore, the semaphore is un-
      locked by the calling process. If condition is MSEM_IF_WAITERS, and no process is
      waiting to lock the semaphore, the semaphore is not unlocked and an error is returned.
      */
int *msem_remove(msemaphore *sem);
   /* remove the semaphore structure, any further use of sem produces undefined results. */
```

Related System Command

List all the existing IPC objects

ipcs [-asmq] [-tclup]

-m shared memory segments
-q message queues
-s semaphore arrays
-a all (this is the default)

```
------ Semaphore Arrays --------
key          semid   owner   perms   nsems   status
0x12345678   2176    jiwei   666     3
```

Remove an existing IPC object

ipcrm -smq ID

-m shared memory segments
-q message queues
-s semaphore arrays

Caveats

- Be careful with multiple initialization in System V and Posix memory-based semaphores. Multiple initialization could corrupt the semaphore value.
- The fairness of semaphore waiting queue is not guaranteed in some Unix system.
- *Dead-lock* could happen if,
  - Multiple semaphores are nested.

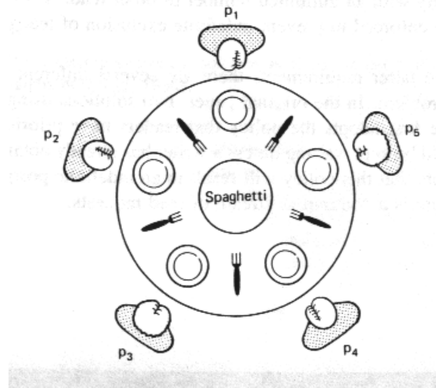
Process 1	Process 2
lock(semaphore-A);	lock(semaphore-B);
...	...
lock(semaphore-B);	lock(semaphore-A);
...	...
unlock(semaphore-B);	unlock(semaphore-A);
...	...
unlock(semaphore-A);	unlock(semaphore-B);

  - The process dies before it can unlock the semaphore. System V semaphore supports the SEM_UNDO to increase the semaphore by 1 when such an event happens.
- *Starvation* could happen when the semaphore waiting queue is not operated fairly. Some program waits too long (or forever) for the semaphore to be unlocked.

The Dining Philosophers Problem

Five philosophers, Pi, sit around a table in the middle of which is a bowl of spaghetti. There's a plate in front of each philosopher and five forks on the table, one between each two plates. To eat, a philosopher must pick up two forks next to his plate. After eating, the philosopher drops both forks.

The philosophers are considered process and the forks resources. The challenge is to design an algorithm to ensure no philosopher will starve.



The Dining Philosophers Problem (cont...)

Two concerns:

- Dead-lock: when each philosopher holds the fork on the left side.
- Starvation: Two philosophers, say P2 and P5, may conspire to starve P1 to death.

Suggested rules:

- Only one philosopher is allowed eating at any one time, with a fair waiting queue.
- At most 2 philosophers are allowed eating at any one time, and sequentialize the picking-up and putting-down of the forks, or randomize the sleep time.

Example Programs

sema.c and sema.h: This provides an simpler and easier to understand interface to the System V semaphore system calls. There are 7 routines available

id = sema_create(key, initval);	# create with initial value or open
id = sema_open(key);	# open (must already exist)
sema_wait(id);	# wait = P = down by 1
sema_signal(id);	# signal = V = up by 1
sema_value (id);	# return the value of the semaphore
sema_close(id);	# close the semaphore, remove it if no other process # uses it anymore.
sema_rm(id);	# remove the semaphore.

Other programs are all based upon sema.c.

createSem.c: Creates a semaphore with the specified value.

watchSemaphore.c: Continuously watches the value of a given semaphore.

benchSemaphore.c: Benchmarks the latency of a pair of lock/unlock semaphore operations.

Example Programs (cont...)

demoSemaphore.c: This program can be controlled form the keyboard to lock and unlock a specified semaphore. Multiple instances of the program can used to,

- demonstrate how binary and counting semaphore works, i.e. using createSem to create a semaphore with different values.
- demonstrate automated undoing the semaphore locking during process death, i.e. kill a process when it locks the semaphore.
- show semaphore queuing fairness, i.e. having two one process waiting for a locked semaphore, unlock the semaphore and see which process gets it.

Clarifications

Primitive

A function or operator which is built into a programming language or operating system, either for speed of execution or because it would be impossible to implement it in the language or OS.

Function typedef

```
typedef void (*PointerToIntFunction)(int);
   /* define a type which is a pointer to a function that takes int as its argument. */
```

With this new type, the signal() function,

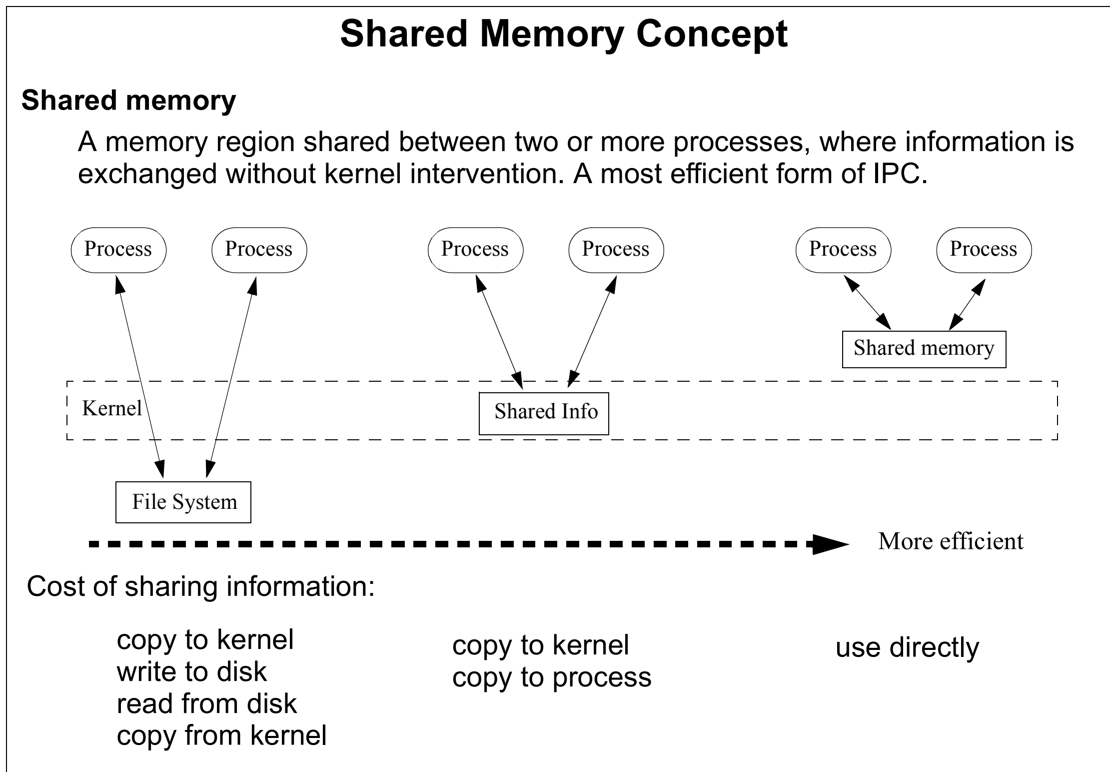
```
void (*signal(int signum, void (*handler)(int)))(int);
```

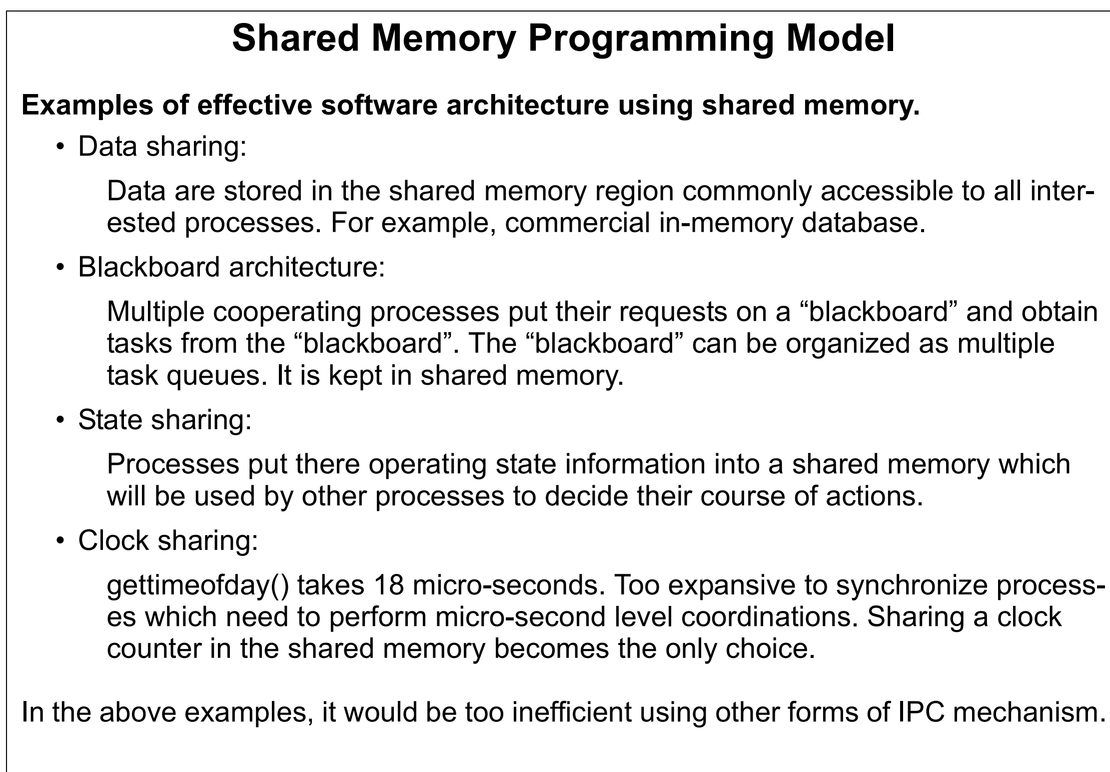
can be more clearly written as,

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


4. Shared Memory

- Shared Memory Concept
- Shared Memory Programming Model
- Shared Memory Properties
- System V Shared Memory
- Shared Memory Initialization
- Persistent Shared Memory
- Related System Calls
- Related System Commands
- Posix Shared Memory
- Caveats
- Example Programs


Shared Memory Concept
Shared memory
A memory region shared between two or more processes, where information is exchanged without kernel intervention. A most efficient form of IPC.
Process
Process
Process
Process
Process
Process
Shared memory
Kernel
Shared Info
File System
More efficient
Cost of sharing information:
copy to kernel
write to disk
read from disk
copy from kernel
copy to kernel
copy to process
use directly


Shared Memory Programming Model
Examples of effective software architecture using shared memory.
• Data sharing:
Data are stored in the shared memory region commonly accessible to all interested processes. For example, commercial in-memory database.
• Blackboard architecture:
Multiple cooperating processes put their requests on a “blackboard” and obtain tasks from the “blackboard”. The “blackboard” can be organized as multiple task queues. It is kept in shared memory.
• State sharing:
Processes put there operating state information into a shared memory which will be used by other processes to decide their course of actions.
• Clock sharing:
gettimeofday() takes 18 micro-seconds. Too expansive to synchronize processes which need to perform micro-second level coordinations. Sharing a clock counter in the shared memory becomes the only choice.
In the above examples, it would be too inefficient using other forms of IPC mechanism.

Shared Memory Properties

Persistence

Kernel-persistent, can be file system persistent.

Name space

Shared memory Keys and IDs (System V). Keys are used when creating the semaphore and IDs are used in all the references afterwards.

Permission

Permission can be specified by the process, the same as a file.

Latency

Memory access are very fast, down to nano-seconds level. Additional synchronization cost (i.e. semaphore cost) should be included.

Limitations

Available on the same machine only. There are programs which make shared memory distributed, e.g. MIT Alewife DSM (distributed shared memory).

System V Shared Memory

System V shared memory operations

- Allocating a shared memory segment: shmget().
- Attaching the process to the shared memory segment: shmat().
- Controlling and detaching the shared memory: shmdt() & shmctl().

Allocate a shared memory segment

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int oflag);
```

shmget() returns a shared memory ID if successful, otherwise -1.

key: shared memory key.

size: shared memory segment size.

oflag can be a file permission (e.g. 0666), bitwise-ORed with IPC_CREAT, or IPC_CREAT | IPC_EXCL.

IPC_CREAT: create if it does not already exist, otherwise proceeds without error.

IPC_CREAT | IPC_EXCL: create if it does not already exist, otherwise return an error.

System V Shared Memory (Cont...)

Attach to a shared memory segment

void *shmat (int shmid, void *shmaddr, int oflag);

shmat() return the shared memory segment address if successful, -1 otherwise.

shmid: shared memory id returned by shmget().

shmaddr: if it's 0 (null pointer), system selects the shared memory address, otherwise, the shared memory is attached to the address specified by *shmaddr*, when *oflag* is 0, or *shmaddr* rounded down to SHMLBA (low boundary address) when *oflag* is SHM_RND.

For portability reasons, it's recommended always setting shmaddr to 0 to allow system choosing the address.

Detach from a shared memory segment

int shmdt (const void *shmaddr);

Detach a shared memory segment from the process. When the process dies, all the shared memory segments are automatically detached.

System V Shared Memory (Cont...)

Control shared memory segments

int shmctl(int shmid, int cmd, struct shmid_ds *buf);

shmctl() operates on the shared memory segment.

cmd can be

- IPC_RMID: remove the shared memory segment.
- IPC_SET: modify the *shmid_ds* structure with *buf*
- IPC_STAT: return the current *shmid_ds* structure in *buf*.
- SHM_LOCK: prevents swapping of a shared memory segment.
  Requires access privilege (super-user or effective creator).
- SHM_UNLOCK: allows the shared memory segment to be swapped out.

```
struct shmid_ds {
        struct    ipc_perm shm_perm; /* operation perms */
        int shm_segsz;          /* size of segment (bytes) */
        time_t    shm_atime;          /* last attach time */
        time_t    shm_dtime;          /* last detach time */
        time_t    shm_ctime;          /* last change time */
        unsigned short shm_cpid; /* pid of creator */
        unsigned short shm_lpid; /* pid of last operator */
        short     shm_nattch;         /* no. of current attaches */
};
```

Shared Memory Initialization

Allow only one initialization

```
if ((shmId = shmget(0x12345678, 8, 0666 | IPC_CREAT | IPC_EXCL)) > 0) {
    initIt = 1; /* set the flag to initialize the shared resource */
}
/* failure could be caused by the existence of the shared memory */
else if ((shmId = shmget(0x12345678, 8, 0666)) < 0) {
    perror("shmget() error");
    exit(1);
}
if ((shmPtr = (char*) shmat(shmId, 0, 0)) == (char*) -1)   {
    perror("shmat() error");
    exit(1);
}
/* Grab a 32-bit word aligned portion, cast it as a pointer to an integer. */
producerAreaPtr = (int*) ((int)shmPtr & 0xfffffffc) + 1;

if (initIt) {
    (*shmPtr) = 0;
}
```

The initialization part should be protected with a semaphore to prevent access before initialization.

Persistent Shared Memory

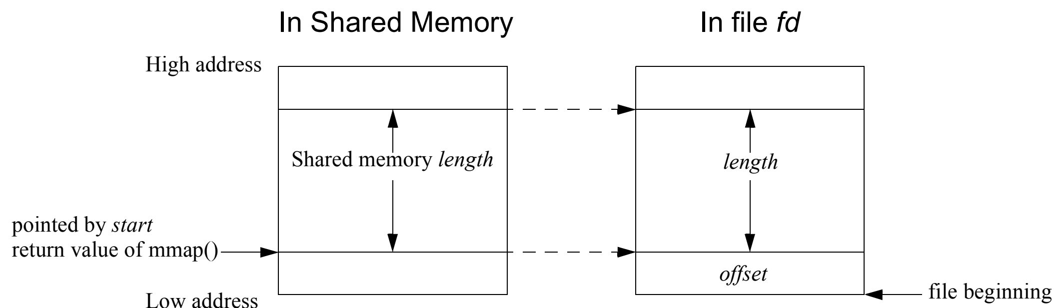
Shared memory can be mapped to a file and all the changes to the shared memory can be mirrored in a file. OS does synchronization between the two automatically.

mmap, munmap - map or unmap files or devices into memory

#include <unistd.h>

#include <sys/mman.h>

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



Persistent Shared Memory (cont...)

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

prot specifies the protection of the mapped region. It can be:

PROT_READ: data can be read

PROT_WRITE: data can be written

PROT_EXEC: data can be executed

PROT_NONE: data cannot be accessed

Common value is PROT_READ | PROT_WRITE.

flag specifies the data sharing in the mapped region. It can be:

MAP_SHARED: changes are shared between all processes.

MAP_PRIVATE: changes are private the each individual process.

MAP_FIXED: interpret the *addr* argument exactly. This ruins portability. Don't use it.

To removed the mapped memory,

int munmap(void *start, size_t length);

Any further reference to the memory region after munmap() causes a SIGSEGV or SIGBUS.

Related System Calls

A different way of defining a System V IPC key.

Mapping a file-name to System V IPC key

include <sys/types.h>

include <sys/ipc.h>

key_t ftok (char *pathname, int proj);

proj is an integer indicating different projects.

pathname is the absolute path of an existing file. If the file doesn't exist, ftok() returns -1.

If the file has been removed and recreated, ftok() will return a different key.

ftok() can also be used for semaphore keys.

Related System Commands

The same as IPC commands as that for semaphore. List all the existing shared memory segments

ipcs -m

Shared Memory:

	shmid	key	perms	owner	group
m	412	0x12345678	--rw-rw-rw-	jiwei	users

Remove an existing IPC object

ipcrm -m ID

Posix Shared Memory

Posix shared memory uses the mmap() to facilitate the shared memory functionality. This can be done in two ways:

- Using an ordinary file with *open()*, *close()*, etc. file operations.
- Using *shm_open()* and *shm_unlink()*.

Open a Posix shared memory object

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

shm_open() returns a nonnegative descriptor if OK, -1 on error.

name: full valid path name.

oflag: can be

0: just open,

O_CREAT: create and initialize if it doesn’t exist, otherwise just open,

O_CREAT | O_EXCL: create and initialize if it doesn’t exist, otherwise return error.

mode: permission bits. Not require if oflag is 0.

int shm_unlink(const char *name);

/* remove the name of the shared memory object. It has no effect on the existing reference, only prevents subsequent calls from succeeding.*/

Caveats

- Must synchronize accesses to prevent data corruption.
- Initialization should be done only once and need to prevent access before initialization.
- Memory in the shared memory area are not guaranteed to be word (32 bits or 64 bits) aligned. Casting (for int, long) could cause segmentation violation.
- System shared memory can be fragmented. A large contiguous block memory may not be available. Instead of claiming a large contiguous chunk, claim smaller chunks and chain them together.
- Shared memory can be swapped to virtual memory, which could seriously hinder application performance. We can lock the shared memory in RAM.
- Who should destroy shared memory segments?

  Shared memory segments contain valuable data and debugging information. It's recommended that no process should destroy any shared memory.

  Use *ipcrm* command to remove shared memory in a script at system start-up or shut-down.

Example Programs

consumer.c producer.c: Simulate synchronized producer and consumer programming model. Use two System V semaphores to perform the synchronization. Multiple producers and consumers can be executed at the same time (using the same shared memory and semaphore keys).

The producer increments a counter in the shared memory by 1 for its turn and the consumer read the producer counter and add 1000 and put it in its own area.

Usage:

./consumer 0x12345678

./producer 0x12345678

observer.c: Observe the shared memory areas of the producer and consumer. No semaphore is used because the observer does not modify the shared resource.

Usage:

./observer 0x12345678

Semaphore Protected Flags

A typical Unix system supports 60 semaphores. Though the number is configurable, 4096 is considered to be a large number.

Semaphore protected flags

Use flags to control operations on records and Use a single semaphore to protect flag operations.

This is beneficial when:

- There are a large amount of record need to be protected with semaphore.
- Operation on the record take a relatively long time.

Semaphore contention is minimized because the flag setting operation takes very little time.

Example Program

worker.c: Use flags to lock records. Operations on flags are protected by a semaphore.

There can be multiple worker program running. The workers randomly pick a record, lock it, increment the counter the record by 1 and then release the lock. If the flag of the chosen record is locked, it goes for a different record.

If the flag operation is not protect, the locking will fail on a multi-processor machine.

At the end, the sum of the counters should match up to the total operations by all the workers.

5. Process Priorities

Priority

A process's priority determines how much CPU time it will receive. Priority is used by scheduler to determine when and how long a process should run.

Processes priorities vary between schedulers. They are normally based on:

- who runs the process (user vs. system),
- which scheduler the process belongs to (time-sharing vs. real-time),
- relative process priority,
- the amount of time having been consumed and the length of waiting time.

Priority scale are different depending on OS. On HP-UX,

Priority and relative process priority can be observed with *top*.

Relative Process Priority

Relative process priority

Relative process priority is expressed with nice value.

nice

/usr/bin/nice [-n adjustment] [-adjustment] [command [arg...]]

nice runs the given command with its scheduling priority adjusted to the specified adjustment.

If no arguments are given, *nice* prints the current scheduling priority. If no adjustment is given, the priority of the command is incremented by 10.

The superuser can specify a negative adjustment. The priority can be adjusted by nice over the range of -20 (the highest priority) to 20 (the lowest).

Note: shells (csh, ksh) have built-in *nice* which has the default adjustment 4.

renice

renice nice-value [[-p] pid ...] [[-g] pgrp ...] [[-u] user ...]

Alter priority of running processes

Note: non-superuser can only decrease the nice value.

Related System Calls

On all System VR4 and 4.4 BSD system

```
#include <sys/time.h>
#include <sys/resource.h>
int getpriority(int which, int who);
int setpriority(int which, int who, int prio);
   /* Set and get relative priorities. Corresponds to the nice command. Depending on which
      value being PRIO_PROCESS, PRIO_PGRP, or PRIO_USER, who can be pid, gid or
      user id respectively */
#include <unistd.h>
int nice(int inc);
   /* operates on the current process. only super-user can have negative inc. */
```

On HP-UX, internal real-time priority can be modified with *rtprio()*.

```
#include <sys/rtprio.h>
int rtprio(pid_t pid, int prio);
   /* sets or reads the real-time priority of a process. When setting the real-time priority of
      another process, the uid, euid of the calling process must match (or with appropriate
      privileges) the uid of the process to be modified.*/
```


6. TCP/IP Socket

- Network Architecture
- OSI Reference Model
- Data Transmission in OSI Model
- TCP/IP Protocols
- TCP/IP Reference Model
- TCP Socket Concept
- TCP/IP Socket Programming Model
- TCP/IP Socket Properties
- TCP Client-Server Flowchart
- Client-side Socket API
- Server-side Socket API
- Socket Multiplexing
- Related System Calls
- Related System Commands
- Caveats
- Examples

Network Architecture

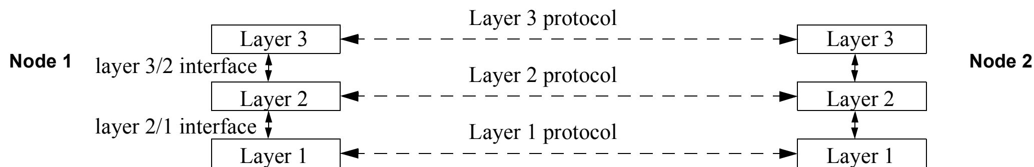
Layered Architecture

To reduce the design complexity, most networks are organized as a series of layers (levels). Each layer is built upon the one below it. There is no cross layer reference. This

- hides implementation details, and
- simplifies the network architecture.

Communication Protocol

Communication between two nodes are conducted between counter-part layers. The rules and conventions used in this communication are known as a *protocol*.



Network Architecture

A set of layers and protocols is called *network architecture*. A list of the protocols is called a *protocol stack*.

OSI Reference Model

Open Systems Interconnection (OSI) Reference Model

Defined in 1983 by the International Standards Organization (ISO). Intended to standardize protocol stacks in communication between open systems.

	Layer	Description
7	Application	Application related protocols.
6	Presentation	Dictates the abstract organization of the data.
5	Session	Allows network nodes to establish sessions.
4	Transport	Breaks down data into smaller units and reassembles them.
3	Network	Controls the operations of sub-nets.
2	Data Link	Makes the raw bits appear free of transmission errors.
1	Physical	Transmits raw bits over a communication channel.

Data Transmission in OSI Model

Each layer of protocol adds a header to the data packet.

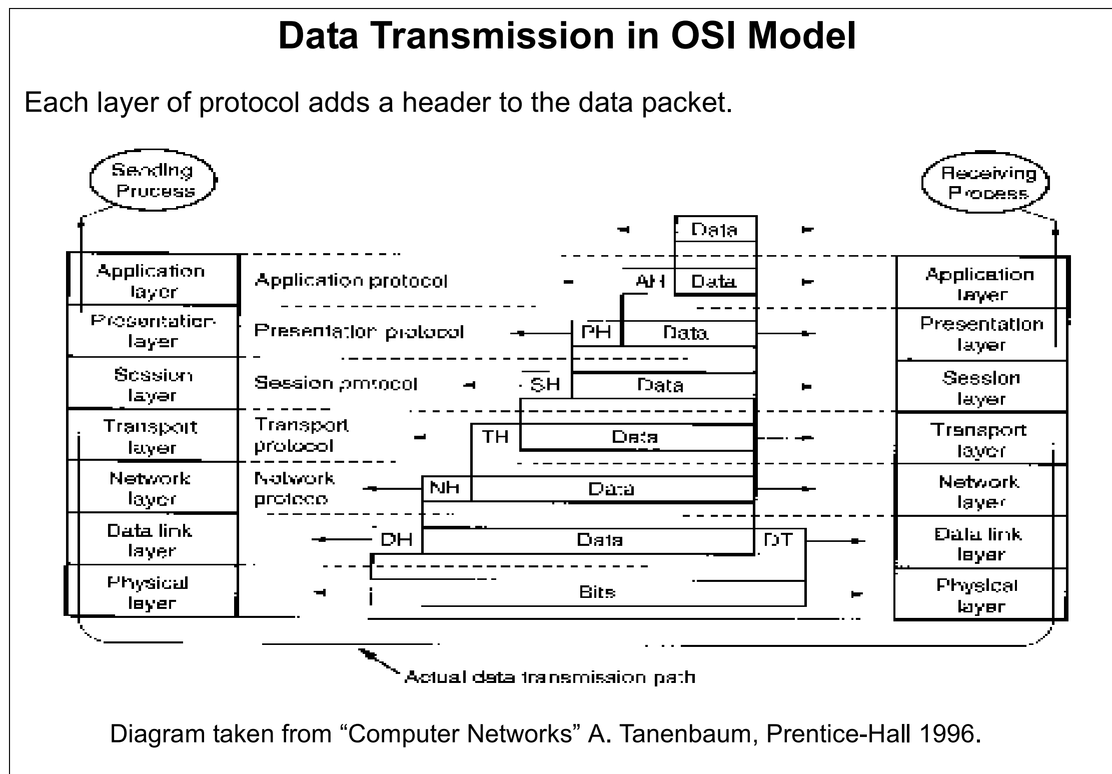


Diagram taken from “Computer Networks” A. Tanenbaum, Prentice-Hall 1996.

TCP/IP Protocols

IP: Internet Protocol

IP is a network layer protocol which defines the IP datagram as the unit of information passed across the Internet and provides the packet delivery service to TCP, UDP, etc.

IP datagrams are of variable size, to the maximum of 65535 bytes.

IP v4 was defined in the early 1980s. It uses 32-bit addresses. IP v6 was designed in the mid-1990s. It uses 128-bit addresses.

TCP: Transmission Control Protocol

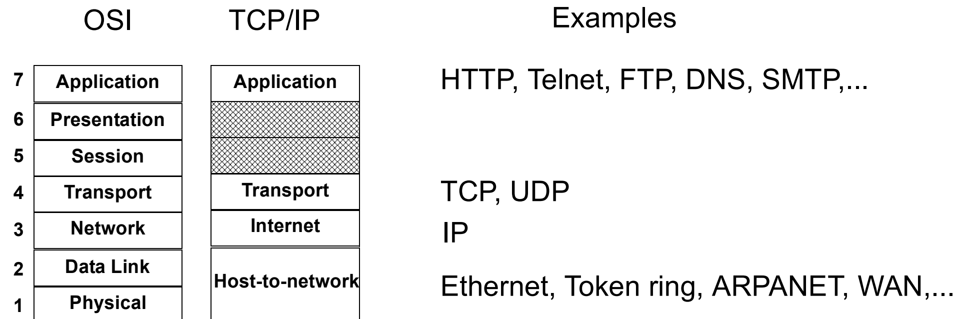
TCP is a transport layer protocol which provides a connection-oriented, reliable, full-duplex, stream service.

UDP: User Datagram Protocol

UDP is also a transport layer protocol which provides a connection-less, full-duplex, no-guarantee datagram delivery service.

TCP/IP Reference Model

The TCP/IP protocols were first defined in 1974 in DoD's ARPANET project. The TCP/IP reference model was later defined in 1985.



TCP Socket Concept

In this session, we focus on TCP sockets while leave UDP sockets to a later session.

Socket

A *socket* is defined with an IP address and a port number. For example, 10.0.0.1:80. A *socket pair*, which consists of two sockets, uniquely identifies a TCP connection.

Berkeley socket application programming interface

Berkeley socket is the common API for TCP/IP programming. Socket is also used for other communication protocols, e.g. Unix domain and Xerox Network Systems protocols. But TCP/IP takes the overwhelming majority.

TCP connection properties

- Connection-oriented. A connection needs to be initiated before the data transmission.
- Reliability. Providing acknowledgment and retransmission.
- Sequentialize data. The sequential order of data arrival is guaranteed.
- Flow control. Data are buffered in the sockets at both ends.
- Full-duplex. Two-way data transmission can be achieved at all time.

TCP/IP Socket Programming Model

Client-Server programming

The process that initiates a connection is called a *client process*. The process that expects connections from client processes is called a *server process*.

Note that this is different from the client-server concept in information processing, which a client requests a service/information provided by a server.

Flexible distributed computing

Client and server processes can collocate on the same node or reside on nodes across a network, without any code change.

TCP/IP Socket Properties

Persistence

TCP/IP sockets are process-persistent.

Name space

IP address + port number. 65536 socket ports in total. Root privilege is required for the first 1024 ports. Those ports are called services, defined in /etc/services.

Permission

Permission can be specified by the process, the same as a file.

Latency & Throughput

Over the network, close to 80-90% of the available bandwidth, about 7.5 MB/s on a switched 100Base-T Ethernet. On the same machine, over 5.5 MB/s on a Pentium-100.

On a quiet 100Base-T Ethernet, exchanging a 1024-byte packet between two hosts of a sub-net has a latency of ~0.1 ms.

Advantages

Node and network independent. There's very few limitations.

TCP Client-Server Flowchart

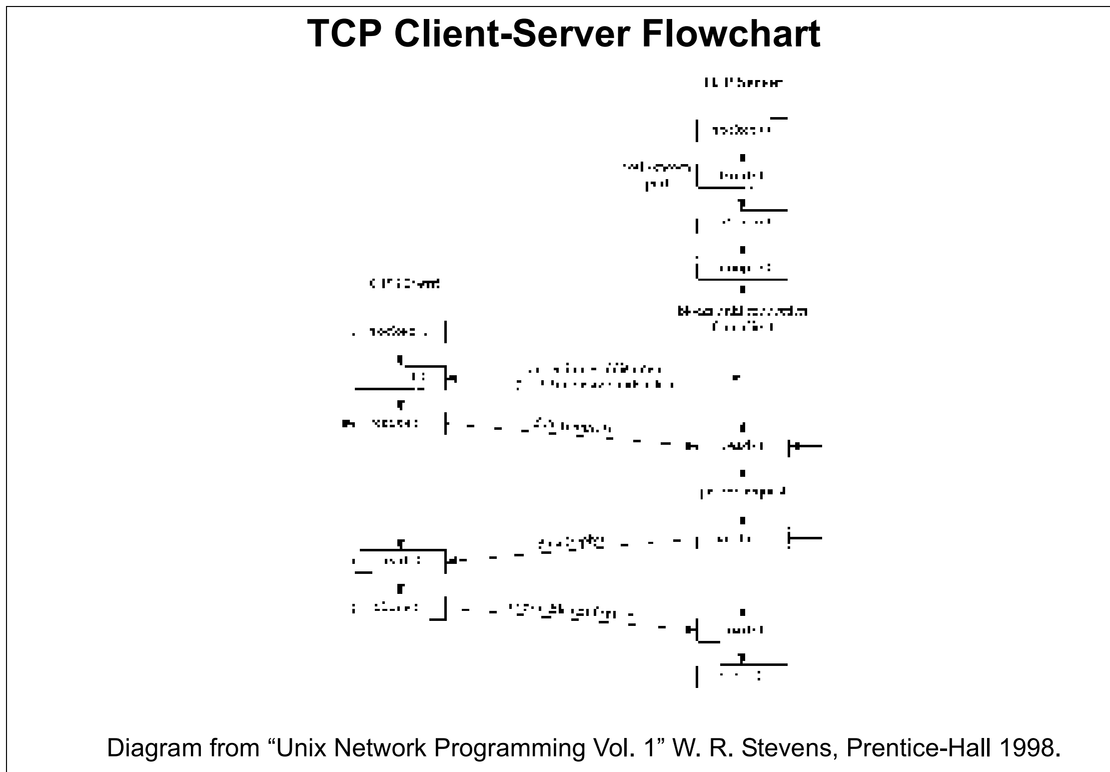


Diagram from “Unix Network Programming Vol. 1” W. R. Stevens, Prentice-Hall 1998.

Client-side Socket API

Create a socket

#include <sys/types.h>

#include <sys/socket.h>

int socket(int domain, int type, int protocol);

domain: specifies the protocol domain, can only be AF_INET.

type: specifies the semantics of communication. Currently defined types are:

- SOCK_STREAM: for TCP
- SOCK_DGRAM: for UDP
- SOCK_RAW: for raw socket
- SOCK_SEQPACKET: for Xerox Network Systems protocols
- SOCK_RDM: not implemented yet.

protocol: specifies a particular protocol to be used with the socket. This can be obtained with getprotobyname(), or use 0 when there’s only one protocol in the family.

#include <netdb.h>

struct protoent *getprotobyname(const char *name); /* obtain a protocol name structure */

e.g. ptrp = getprotobyname("tcp");

sd = socket(AF_INET, SOCK_STREAM, ptrp->p_proto)

-1 is returned if an error occurs, otherwise the return value is a descriptor referencing the socket.

Client-side Socket API (Cont...)

Connect to the remote host

```
#include <sys/types.h>
#include <sys/socket.h>
   int connect(int sockfd, struct sockaddr_in *serv_addr, int addrlen);
```

sockfd: the socket file descriptor.

addrlen: the size of serv_addr.

serv_addr:

```
   struct sockaddr_in {
        uint8_t          sin_len;    /* length of structure */
        sa_family_t      sin_family; /* AF_INET */
        in_port_t        sin_port;   /* 16bit TCP port number */
        struct in_addr   sin_addr;   /* 32-bit IPv4 address, network byte ordered */
   }
```

If the connection succeeds, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

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

Close the socket. It is the same for opened files.

Client-side Socket API (Cont...)

```
#include <sys/types.h>
#include <sys/socket.h>
   int send(int sockfd, const void *msg, int len, unsigned int flags);
```

sockfd: socket file descriptor.

msg: pointer to the message.

len: length of the message.

flags: normally 0. It can be MSG_OOB (process out-of-band data) or MSG_DONTROUTE (bypass routing, use direct interface).

The call returns the number of characters sent, or -1 if an error occurred.

```
#include <sys/types.h>
#include <sys/socket.h>
   int recv(int sockfd, void *buf, int len, unsigned int flags);
```

sockfd: socket file descriptor.

buf: buffer to save the message.

len: length of the buffer.

flags: normally 0. It can be MSG_OOB (process out-of-band data), MSG_PEEK (peek at incoming message), or MSG_WAITALL (wait for full request).

The call returns the number of bytes received, or -1 if an error occurred.

Server-side Socket API

Bind a socket with the local address

```
#include <sys/types.h>
#include <sys/socket.h>
   int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
```

sockfd: a socket file descriptor.

my_addr: current host address

```
struct sockaddr {
      uint8_t        sa_len;       /* length of structure */
      sa_family_t    sa_family;    /* AF_INET */
      char           sa_data[14];  /* protocol specific address */
}
```

This is a generic form of *sockaddr_in*. *sockaddr_in* can be casted to *sockaddr*.

addrlen: size of *my_addr*.

On success, zero is returned. On error, -1 is returned, and *errno* is set appropriately.

Server-side Socket API (Cont...)

Listen for connections

```
#include <sys/socket.h>
   int listen(int sockfd, int backlog);
```

sockfd: socket file descriptor.

backlog: maximum backlog length, usually 5. On a busy HTTP server, the *backlog* is best set to 32.

Convert the socket to a passive socket and specify a waiting queue.

Accept a connection

```
#include <sys/types.h>
#include <sys/socket.h>
   int accept(int sockfd, struct sockaddr *addr, int *addrlen);
```

sockfd: the listening socket.

addr: pointer to a *sockaddr* structure.]

addrlen: pointer to an int. it will be filled a the size of the *sockaddr* structure.

The *accept* call extracts the first connection request on the queue of pending connections, creates a new socket and allocates a new file descriptor for the socket.

The call returns -1 on error. If it succeeds, it returns a non-negative integer that is a descriptor for the accepted socket.

Socket Multiplexing

By default, all sockets are blocking which means read()/recv() blocks until there's data arrived at the socket, and write()/send() blocks until the data is sent out. (Non-blocking sockets are also possible. We shall skip the non-blocking sockets in this class.)

If a client or a server process needs to handle more than one socket, the process would prefer to know whether there's data on the socket before it commits on a read.

select()

```
#include <sys/time.h> #include <sys/types.h> #include <unistd.h>
   int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *time-
      out);
```

n: the number of fd to be tested. It should be the highest fd +1. The maximum fd_set size is 1024.

readfds: fds to be tested being ready for reading.

writefds: fds to be tested being ready for writing.

exceptfds: fds to be tested having an exception.

timeout: the maximum waiting time. If *timeout* is null, it waits forever. *timeval* is defined as,

```
struct timeval {long tv_sec; /* seconds */ long tv_usec; /* microseconds */};
```

select() returns the number of sockets which is ready, or -1 if there's an error.

Socket Multiplexing (Cont...)

Macros to manipulate fd_set

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>
   FD_ZERO(fd_set *set); /* clear all bits in fd_set */
   FD_CLR(int fd, fd_set *set); /*turn off the bit for fd in fd_set */
   FD_SET(int fd, fd_set *set); /* turn on the bit for fd in fd_set */
   FD_ISSET(int fd, fd_set *set); /* test if the r fd in fd_set is ready */
```

Example

```
/* Assume that we have two sockets fd1 and fd2 that we want to test whether any of
them is ready for reading */
fd_set myFdSet;
FD_ZERO(&myFdSet);
FD_SET(fd1, &myFdSet);
FD_SET(fd2, &myFdSet);
n = select(max(fd1,fd2)+1, &myFdSet, 0, 0, 0);
if (n > 0) {     if (FD_ISSET(fd1, &myFdSet) {/* fd1 is ready, work on it */}
                 else if (FD_ISSET(fd2, &myFdSet) {/* fd2 is ready, work on it */}
}
```

Related System Calls

Obtain local and remote socket address

```
#include <sys/socket.h>
    int getpeername(int sockfd, struct sockaddr *name, int *name-len);
        /*Get name of connected peer socket */
    int getsockname(int   sockfd, struct sockaddr * name, int * namelen)
        /* Get local socket name */
```

Network byte order vs. host byte order conversion routines

```
#include <netinet/in.h>
    unsigned long int htonl(unsigned long int hostlong);
        /*host byte to network byte, for long integer*/
    unsigned short int htons(unsigned short int hostshort);
        /*host byte to network byte, for short integer */
    unsigned long int ntohl(unsigned long int netlong);
        /* network byte to host byte, for long integer */
    unsigned short int ntohs(unsigned short int netshort);
        /* network byte to host byte, for short integer */
```


Related System Calls (Cont...)

IP address manipulation

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
    int inet_aton(const char *cp, struct in_addr *inp);
        /* converts the Internet host address from the numbers-and-dots notation into binary
        data */
    unsigned long int inet_addr(const char *cp);
        /* converts the Internet   host address from numbers-and-dots notation into binary
        data in network byte order
    char *inet_ntoa(struct in_addr in);
        /* converts the Internet   host address in given in network byte order to a string in
       standard numbers-and-dots notation */
    struct in_addr inet_makeaddr(int net, int host);
        /* makes an Internet   host address in network byte order by combining the network
        number with the local address, both in local host byte order. */
```

Related System Commands

netstat

Display network connections, routing tables, interface statistics, etc.

% netstat -a /* display TCP/UDP connections and states. hosts displayed in name*/

% netstat -an /* TCP/UDP connections and states. hosts displayed in number*/

% netstat -c /* print the selected table every second continuously on the screen until you interrupt it */

% netstat -i /* display the existing network interfaces */

Caveats

- Due to the un-reliability of networks, TCP/IP programs need to be careful with the condition of the connection. The connection can be broken or congested at any time. Check the return code of each operation.
- Operations to a broken connection result in a SIGPIPE signal and a -1 return code.
- Processes at both ends may likely to become stale or insane. Solution to this problem is to maintain a heartbeat between the client and server.
- Socket calls (i.e.connect(), accept(), send(), recv() and select()) are slow system calls. They can be interrupted by any signal. Check the error code to determine whether the call has been completed properly.
- A process that handles multiple socket need to avoid blocking on a socket, because of the responsiveness and the possibility of deadlock. Use select() to test the socket beforehand or use alarm() signal to break the call.
- Concurrent server can be programmed with forking additional processes/ threads, or non-blocking sockets. Forking processes has a simpler logic (example in socket-faq dir), but non-blocking sockets are more efficient.

Examples

httpGet.c

A simple http client. It connects to a http server and grabs the content of a specified page, without stripping off the HTTP header.

Usage:./httpGet requestString host [port]

Example:./httpGet "/index.htm" 191.72.1.1 8080

server.c

allocate a socket and then repeatedly execute the following:

- wait for the next connection from a client
- send a short message to the client
- close the connection
- go back to step (1)

client.c

allocate a socket, connect to a server, and print all message that the server sends to it.

Examples (Cont...)

remoteShell

The remoteShell sub-directory contains an extension to server.c/client.c. It implements a remote command execution shell. The client sends a Unix command to the server. The server executes the command and sends back the output of the command. The client displays the output to stdout.

Start the server:% rshellServer 9000

Send a command:% rshell 10.2.3.4 9000 "ls -l"

socket-faq

socket-faq directory has a couple of good examples on TCP and UDP sockets.

The Socket FAQ contains good answers to many TCP/IP socket related questions. A good socket libary, Simple Socket Library - SSL, can also be found on the FAQ page.

http://www.lcg.org/sock-faq/

TCP/IP performance measurement tool

HP’s Netperf is a good to measure TCP/IP throughput and latency.

http://www.netperf.org/netperf/NetperfPage.html