UNIX NETWORK PROGRAMMING

Anupama Potluri
Dept. of Computer and Information Sciences
University of Hyderabad

Overview

- Process Control
- Signals
- Pipes and FIFOs
- Mutexes and Condition Variables
- Posix Semaphores
- Shared Memory
- Socket Programming

PROCESS CONTROL

Process Control – Process Creation

- Other than the **initd, swapper and pagedaemon** processes, every other process in Unix is created by a call to the system call *fork()*.
  - Ex.1: Explore whether swapper in a Linux system is created by kernel using *ps* command.
- *fork()* creates a copy of the program and both parent and child execute simultaneously.
- **Copy-on-write:** Data, stack and heap of a parent are not copied. The protection is set to read-only until the parent or child attempts to write, at which time a copy is made.

Program 8.1 from APUE by W.Richard Stevens

```
int  glob =6;
char  buf [ ]= “a write to stdout \n”;


int main(void) {
  int var;
  pid_t pid;
  var = 88;

  if (write(STDOUT_FILENO, buf, sizeof(buf)-1) !=
      sizeof(buf)-1)  printf(“write error”);
  printf (“before fork \n”);
  if ((pid = fork()) > 0 ){ sleep(2);}
  else if ( pid == 0 ) {  /* child */
      glob++;
      var++;
   } else { printf(“Error forking\n”); }
   printf(“pid = %d,glob =%d,var =%d\n ”, getpid(),
           glob, var);
}
```

Parent and Child Processes – File Sharing

- When a process is created using *fork()*, it inherits all the **open file descriptors** from the parent. It also inherits : *current working directory, environment, file mode creation mask, signal mask etc.*
- What is different in parent and child? The *return value from fork, process ID, parent process ID etc.*
- **File Sharing:** Parent and child share the same file offset so that when one process writes to the file, the new offset is visible to the other process. Otherwise, they will overwrite each other.
- `Ex.2: Open a file in a program, fork and write to the file in both parent and child – use both unbuffered and buffered write and see the results.`

Process Termination

- Normal termination: *exit()* - called implicitly at the end of every program
- Abnormal termination: *abort()*
- What happens when the parent dies ahead of child?
- What happens when the child dies ahead of parent – *Zombie processes* – what are the issues with zombies?
- Waiting for the child: *wait()* and *waitpid()*

```
pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int
  options); /* WNOHANG – non-blocking */
```

Replacing the Program of a Process

- *exec()* and its variants replace the program that is executing in a process with a new one – i.e., the data, stack, heap and text are replaced with that of the new program
- Properties inherited from the calling process:
  - *pid, ppid, pwd, file mode creation mask, file locks, process signal mask, pending signals etc.*

```
int execl(const char *path, const char *arg,
  ...);
int execv(const char *path, char *const
  argv[]);
```

Program to demonstrate the use of *exec()* variants

```
int main(int argc, char **argv)
{
   .... Initialization ....
   if ((child_pid = fork()) == -1) {
            printf("Error forking\n");
            exit(3);
    } else if (child_pid == 0) { /* Child  */
            sprintf(fd_string, "%d", fd);
            execl("/home/anupama/unp/test/child",
                  fd_string, argv[2], NULL);
    } else {                          /* Parent */
            printf(“In Parent...\n”);
    }
}
```

Process Control – Summary

- Every process can be created only by another process in Unix except for *init* using *fork()*.
- Child processes created with *fork()* execute in parallel with their parents. **copy-on-write** saves the kernel from copying all of parent's data to the child process unless required.
- New programs can be executed in the context of an existing process using the family of *exec()* calls.
- Zombie processes are created when a child dies and a parent has not waited for its exit status.
- A concurrent server needs to take care of Zombie processes or risk running out of process space.

SIGNALS

Signals – an Introduction

- Signals are software interrupts – a way of handling **asynchronous** events in a program.
- Signals you have come across: *SIGINT, SIGTERM, SIGSEGV, SIGBUS, SIGCHLD, SIGKILL*
- Actions to take on Signals:
  - **Ignore**, **Catch** and Default Action
- *signal()* : The system call to handle signals

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

sighandler_t signal(int signum, sighandler_t
  handler);

SIG_IGN: Handler to ignore a signal
```

Example Program for installation of a Signal Handler

```
static void my_handler(int signo);
int main(void) {
    if (signal(SIGUSR1, my_handler) == SIG_ERR) {
       printf(“Error installing handler for SIGUSR1”);
    else if (signal(SIGUSR2, my_handler) == SIG_ERR) {
       printf(“Error installing handler for SIGUSR2”);
    else if (signal(SIGQUIT, SIG_IGN) == SIG_ERR) {
       printf(“Error installing handler for SIGQUIT”);
    }
    for (; ;) pause();
}

static void my_handler(int signo) {
    if (signo == SIGUSR1)
        printf(“Received signal SIGUSR1”);
    else if (signo == SIGUSR2)
        printf(“Received signal SIGUSR1”);
    else
        printf(“Error: received signal %d\n”, signo);
}
```

Deficiencies of *signal()* system call

- Problems with the *signal()* call:
  - it helps to **catch** or **ignore** a signal but you cannot **block** a signal.
  - Sometimes a signal is lost as shown below:

```
int my_sigint_hndlr();
...
signal(SIGINT, my_sigint_hndlr);
...
int my_sigint_hndlr(void) {
    signal(SIGINT, my_sigint_hndlr);
...
}
```

How to send and wait for signals

- Sending Signals : *raise(), kill()*
  - `int raise(int sig);`
  - `int kill(pid_t pid, int sig);`
  - `unsigned int alarm(unsigned int seconds);`
  - `void abort(void);`
- Waiting for Signals : *pause(), sleep()*
  - `int pause(void);`
  - `unsigned int sleep(unsigned int seconds);`
- SIGCHLD : signal sent automatically to parent when child dies.

POSIX Signals (Reliable)

- **Signal Sets** – *sigset_t* : data structure that contains **one bit per signal** – each can be set or unset using functions such as *sigemptyset(), sigfillset(), sigaddset(), sigdelset()* and queried using *sigismember()*.
- **Blocking and Unblocking Signals:** *sigprocmask()*

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

  - *how: SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK*

- **Catching Signals reliably:** *sigaction()*

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

POSIX Signals (contd.)

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

- **Get Pending Signals and Suspend Process:**
  - `int sigpending(sigset_t *set);`
  - `int sigsuspend(const sigset_t *mask);`

Program 10.12 from APUE, W.Richard Stevens

```
Sigfunc *signal(int signo, Sigfunc *func) {
  struct sigaction    act, oact;

  act.sa_handler = func;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;

  if (sigaction(signo, &act, &oact) < 0)
    return SIG_ERR;

  return(oact.sa_handler);
}
```

Program 10.15, APUE, W.Richard Stevens.

```
static void sig_quit(int);

int  main(void) {
  sigset_t newmask, oldmask, zeromask, pendmask;

  if (signal(SIGQUIT,sig_quit)== SIG_ERR)
     err_sys("cant catch SIGQUIT");
  /* Block SIGQUIT and save current signal mask */
  sigemptyset(&newmask);
  sigaddset(&newmask,SIGQUIT);

  if (sigprocmask(SIG_BLOCK, &newmask, &oldmask) < 0)
     err_sys("SIG_BLOCK error");
  sleep(5);

  if (sigpending(&pendmask) < 0)
    err_sys("sigpending error");
```

contd...

```
  if (sigismember(&pendmask, SIGQUIT))
    printf("\nSIGQUIT pending");

  /* allow all signals and pause */
  if (sigsuspend(&zeromask) != -1) {
    printf("sigsuspend error\n");
  }

  /* Reset signal mask which unblocks SIGQUIT */
  if (sigprocmask (SIG_SETMASK, &oldmask, NULL) < 0 )
    printf("SIG_SETMASK error\n");

  exit(0);
}

static void sig_int(int signo)
{
  printf("in __FUNCTION__\n");
  return;
}
```

Reentrant Functions

- **Reentrant Functions:** Those functions whose behavior is not affected by being interrupted in the middle of the execution and called from another context.
- Consider the code below: Is is reentrant or not?

```
int my_func(void) {
    static int index = 0;
    index++;
    while (index != 1);
    ...
}
```

Signals – Summary

- Signals are asynchronous events that can be caught, ignored or the default action can be taken.
- Most signals terminate the program unless caught. A few such as SIGCHLD are ignored by default
- Non-POSIX signals are unreliable in nature and do not allow signals to be blocked.
- POSIX signals are reliable, allow signals to be caught reliably, blocked or can unblock and suspend a program without loss of signals.
- It is important to have reentrant functions when handling signals since the program execution sequence is hard to predict.

Some Exercises in Signals

1. In program 10.1, APUE, W.Richard Stevens, remove the for (; ;) statement and run the program multiple times and observe the output. Explain the output.
2. Create a process in which a file is opened for RW. Then, fork a new process. Alternately, write from parent and child to the file – e.g., “From Parent(pid)” and “From Child(pid)” - using signals.
3. Repeat Ex.2 above as follows: After fork(), exec a new program in child. Remember that a child inherits the open descriptors after exec also. Without re-opening the file, write alternately from parent and child processes.

PIPES and FIFOs

Pipes and FIFOs - Introduction

- **Pipes** are a form of IPC only between **related processes**.
- **FIFOs**, a.k.a, **Named Pipes,** can be used between **unrelated processes** by virtue of having a name.
- Pipes and FIFOs are **unidirectional** in nature. Two pipes or FIFOs are needed if both the processes need to read and write to them.
  - `Q: Do you need IPCs between threads or processes or both?`
- Pipes and FIFOs return file descriptors and can be treated exactly as files. In fact, for Unix, the **world** is a **file**.

IPC Persistence

- Three types of persistence are possible based on the life of the IPC object:
  - **Process-persistent** : exists until last process with this object open closes the object or exits – e.g., Pipes, FIFOs, mutex, condition variable
  - **Kernel-persistent** : exists until the kernel reboots or IPC object is explicitly deleted – e.g., named semaphores, shared memory
  - **File-persistent** : exists until IPC object is explicitly deleted – e.g., some implementations of named semaphores and shared memory

PIPES

- A pipe is created with the following function:
  - `int pipe(int filedes[2]);`
- This function returns **two** file descriptors – one for read, *fd[0]* and the other for write, *fd[1]* : this establishes one unidirectional pipe between the processes.
- For a **duplex pipe,** two calls to *pipe()* are done. Then, the parent closes *fd[0]* of pipe1 and *fd[1]* of pipe2 whereas the child closes *fd[1]* of pipe1 and *fd[0]* of pipe2.
- The shell command | is nothing but the creation of a pipe between the two processes that are executed.

Example Program to illustrate use of Pipes

```
int main(int argc, char **argv) {
  int    pipe1[2], pipe2[2];  pid_t  cpid;
  char   str[MAXLEN];

  pipe(pipe1); pipe(pipe2);
  if (fork() == 0) {
    close(pipe1[1]); close(pipe2[0]);
    strcpy(str, “Sending data to parent”);
    write(pipe2[1], str, strlen(str));
    read(pipe1[0], str, MAXLEN);
    printf("%s\n", str);
  } else if (cpid > 0) {
    close(pipe1[0]); close(pipe2[1]);
    strcpy(str, "Sending data to child");
    write(pipe1[1], str, strlen(str));
    read(pipe2[0], str, MAXLEN);
    printf("%s\n", str);
  } else printf(“Error forking\n”);
  exit(0);
}
```

FIFOs

- FIFOs need two function calls to be opened:
  - `int mkfifo(const char *pathname, mode_t mode);`
  - `int open(const char *pathname, int flags, mode_t mode);`
- If duplex communication is required, two FIFOs have to be opened. The parent opens *pipe1* for write and *pipe2* for read and the child does the opposite.
- The sequence of calls for opening the FIFOs is important as it can, otherwise, lead to a deadlock. (see e.g. in next slide)

Program 4.16, UNP vol.2, W.Richard Stevens

```
#define FIFO1 “/tmp/fifo.1”
#define FIFO2 “/tmp/fifo.2”

void client(int, int), server (int,int);

int main (int argc , char **argv)
{
  int readfd, writefd;
  pid_t childpid;

  /* create two FIFOs; OK if they already exist */
  if ((mkfifo (FIFO1, FILE_MODE) < 0) &&
      (errno != EEXIST))
       printf(“can't create %s\n”,FIFO1);

  if ((mkfifo (FIFO2,FILE_MODE) < 0) &&
      (errno != EEXIST)) {
    unlink (FIFO1);
    printf(“Can't create %s \n“,FIFO2);
  }
```

contd....

```
  if ((childpid = Fork() )== 0){ /* child */
     readfd = Open(FIFO1,O_RDONLY,0);
     writefd = Open(FIFO2,O_WRONLY,0);
     server(readfd,writefd);
     exit (0);
  }

  /* PARENT */
  writefd = Open(FIFO1,O_WRONLY,0);
  readfd = Open(FIFO2,O_RDONLY,0);
  client(readfd,writefd);
  waitpid (childpid, NULL,0);
  close(readfd);
  close(writefd);

  unlink(FIFO1);
  unlink(FIFO2);
  exit(0);
}
```

Properties of Pipes and FIFOs

- A descriptor can be set to **non-blocking** using the O_NONBLOCK flag in *open()*.
- Two constants define the operations of Pipes and FIFOs
  - **OPEN_MAX** : max. no. of descriptors open by a process
  - **PIPE_BUF** : max. amount of data that can be atomically transferred using Pipes and FIFOs
- Write to a pipe or FIFO not open for reading (i.e., closed) results in the signal SIGPIPE and the process terminates.
  - `Q: No signal is generated when a read is issued from a pipe that has been closed. WHY?`

Properties of Pipes and FIFOs (contd.)

- When a descriptor is set to non-blocking, its return value for *write()* depends on the number of bytes to write and the amount of space currently available.
  - If (write-bytes < PIPE_BUF) **and** (avail-space(pipe) < write-bytes) return **EAGAIN**
  - If (write-bytes > PIPE_BUF) **and** (no-avail-space) return **EAGAIN**
  - If (write-bytes > PIPE_BUF) **and** (avail-space(pipe) < write-bytes) return **actual_bytes_written**

Pipes and FIFOS – Summary

- Pipes and FIFOs are unidirectional IPC mechanisms with process persistence. For duplex communication, two Pipes or FIFOs must be created.
- Pipes can be used only between related processes.
- FIFOs can be used between any two arbitrary processes that know the name of the FIFO.
- FIFOs must be opened in the right order to **avoid deadlock** between the client and server processes.
- No. of Pipes or FIFOs open by a process is limited by the parameter **OPEN_MAX** and the amount of data that can be written to them is limited by **PIPE_BUF**.

Mutexes and Condition Variables

Mutexes and Condition Variables - Introduction

- Mutexes and Condition variables are from the Posix thread library (pthread) and are for synchronization between threads in a process.
- Can be used for processes if they are stored in memory that is shared between processes (as in shared memory)
- Mutex is for mutual exclusion.
- Condition Variable allows a process to wait until a condition.

Mutexes

- Primary functions are to **lock** and **unlock** such that only one thread is allowed to acquire the lock.
- Typical code for a critical section looks like this:

```
pthread_mutex_t fastmutex =
  PTHREAD_MUTEX_INITIALIZER;
retval =  pthread_mutex_lock(&fastmutex);
... Critical Section ...
retval = pthread_mutex_unlock(&fastmutex);
```

```
#define  MAXITEMS      100
#define  MAXTHREADS    10

int     nitems;

struct share_s {
  pthread_mutex_t   mutex;
  int               buff[MAXITEMS];
  int               nput;
  int               nval;
} shared = {PTHREAD_MUTEX_INITIALIZER};
```

```
int main(int argc, char **argv)
{
  int             i, nthr;
  pthread_t       prod[MAXTHREADS], cons;

  nthr   = atoi(argv[1]);
  nitems = nthr;
  shared.nput = shared.nval = 0;

  for (i = 0; i < nthr; i++) {
    pthread_create(&prod[i], NULL, produce, &i);
  }
  for (i = 0; i < nthr; i++) {
    pthread_join(prod[i], NULL);
  }
  pthread_create(&cons, NULL, consume, NULL);
  pthread_join(cons, NULL);

  exit(0);
}
```

```
static void *produce(void *arg)
{
  pthread_mutex_lock(&shared.mutex);
  if (shared.nput >= nitems) {
    pthread_mutex_unlock(&shared.mutex);
    return NULL;
  }

  shared.buff[shared.nput] = shared.nval;
  shared.nput++;
  shared.nval += 10;
  pthread_mutex_unlock(&shared.mutex);

  return NULL;
}
```

```
static void *consume(void *arg)
{
  int          i;

  for (i = 0; i < nitems; i++) {
    printf("buff[%d] = %d\n", i,
           shared.buff[i]);
  }

  return NULL;
}
```

Condition Variables

- Condition Variables allow a process to wait for a condition.
- A condition variable **always** has a mutex associated with it since we need to have mutual exclusion to it.
- Either a single process or multiple processes can be woken up by a single condition variable.

```
int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_wait(pthread_cond_t *cond,
  pthread_mutex_t *mutex);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

Condition Variables (contd.)

- A typical code segment with condition variables looks like this:

```
SIGNAL-PROCESS:
    pthread_mutex_lock(&var.mutex);
    set condition true
    pthread_cond_signal(&var.cond);
    pthread_mutex_unlock(&var.mutex);
WAIT-PROCESS:
    pthread_mutex_lock(&var.mutex);
    while (condition is FALSE)
        pthread_cond_wait(&var.cond, &var.mutex);
        modify condition
    pthread_mutex_unlock(&var.mutex);
```

Some Exercises for Mutexes and Condition Variables

1. Create threads in a process and put them to sleep and after waking up, print their thread ID and die. Let the main thread wait for the child tid status. What is the order in which they print? Try multiple times.
2. Repeat the above after removing the pthread_wait() from the main thread. What is the output?
3. Repeat the above such that a global variable is updated by the threads. The global variable starts with value 0 and when it is 0, the main thread prints its thread ID and increments it. Then, thread 1 must wake up, print its thread ID, increment the global variable and die and so on. The main thread waits for the status of all threads before it dies.

Mutexes and Condition Variables – Summary

- Mutexes are for mutual exclusion – i.e., acquire lock and update some global variables and then unlock. Only one program can be executing the critical section.
- Condition Variables are needed for waiting for a condition. A condition variable is always associated with a mutex since the condition variable is accessed by multiple threads which should access it in a mutually exclusive manner.

POSIX Semaphores

POSIX Semaphores - Introduction

- Semaphores are **kernel-persistent** IPC that can be used to synchronize between **threads or processes**.
- Two types : **memory-based** and **named** semaphores
- A **mutex** is a **binary semaphore** but is typically implemented as a different IPC so that it has much **less overhead** than a semaphore.
- Typically, semaphores are **counting semaphores**, **one per resource available**. When all available resources are already in use, a process needing them has to wait until they are released.

Semaphores vs Mutexes and Conditon Variables

- Mutex needs to be unlocked by the same thread as that which locked it; whereas, a semaphore is usually posted to by one process and waited for by another.
- A mutex is either locked or unlocked – only two states and cannot handle waiting for a condition.
- A semaphore allows both mutual exclusion and wait. It also does not lose a post unlike condition variables whose signal can be lost.

Semaphores – Function Calls

- The following are the various function calls used to create, open and use semaphores:

```
sem_t *sem_open(const char *name, int oflag, ...);
int sem_close(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_post(sem_t * sem);
int sem_unlink(const char *name);
```

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

#define  MAXITEMS      10

int     nitems;

struct share_s {
  sem_t               mutex, nempty, nstored;
  int                 buff[MAXITEMS];
} shared;
```

```
int main(int argc, char **argv)
{
  pthread_t       prod, cons;
  nitems   = atoi(argv[1]);

  sem_init(&shared.mutex, 0, 1);
  sem_init(&shared.nempty, 0, MAXITEMS);
  sem_init(&shared.nstored, 0, 0);

  pthread_create(&prod, NULL, produce, NULL);
  pthread_create(&cons, NULL, consume, NULL);
  pthread_join(prod, NULL);
  pthread_join(cons, NULL);

  sem_destroy(&shared.mutex);
  sem_destroy(&shared.nempty);
  sem_destroy(&shared.nstored);

  exit(0);
}
```

```
static void *produce(void *arg)
{
  int         i;

  for (i = 0; i < nitems; i++) {
    sem_wait(&shared.nempty);
    sem_wait(&shared.mutex);
    shared.buff[i % MAXITEMS] = i;
    sem_post(&shared.mutex);
    sem_post(&shared.nstored);
  }

  return NULL;
}
```

```
static void *consume(void *arg)
{
  int          i;

  for (i = 0; i < nitems; i++) {
    sem_wait(&shared.nstored);
    sem_wait(&shared.mutex);
    if (shared.buff[i % MAXITEMS] != i)
      printf("buff[%d] = %d\n", i,
             shared.buff[i % MAXITEMS]);
    sem_post(&shared.mutex);
    sem_post(&shared.nempty);
  }

  return NULL;
}
```

Semaphores – Summary

- Semaphores are typically kernel-persistent IPC but can also be file-persistent as in the case of memory-based semaphores that are implemented in shared memory.
- Semaphores can handle mutual exclusion as well as waiting for a condition.
- They are more reliable than condition variables.

Shared Memory

Shared Memory - Introduction

- Shared Memory is an IPC with **least latency**.
- Users need to handle synchronization **explicitly**.
- Two types of POSIX shared memory:
  - Memory-mapped files
  - Shared memory objects

mmap – A utility for Shared Memory

- *mmap()* **maps** a file or a shared memory object into the **address space** of a process.
- Used with regular files to provide memory-mapped I/O.
- Used with *shm_open()* to provide shared memory between unrelated processes.

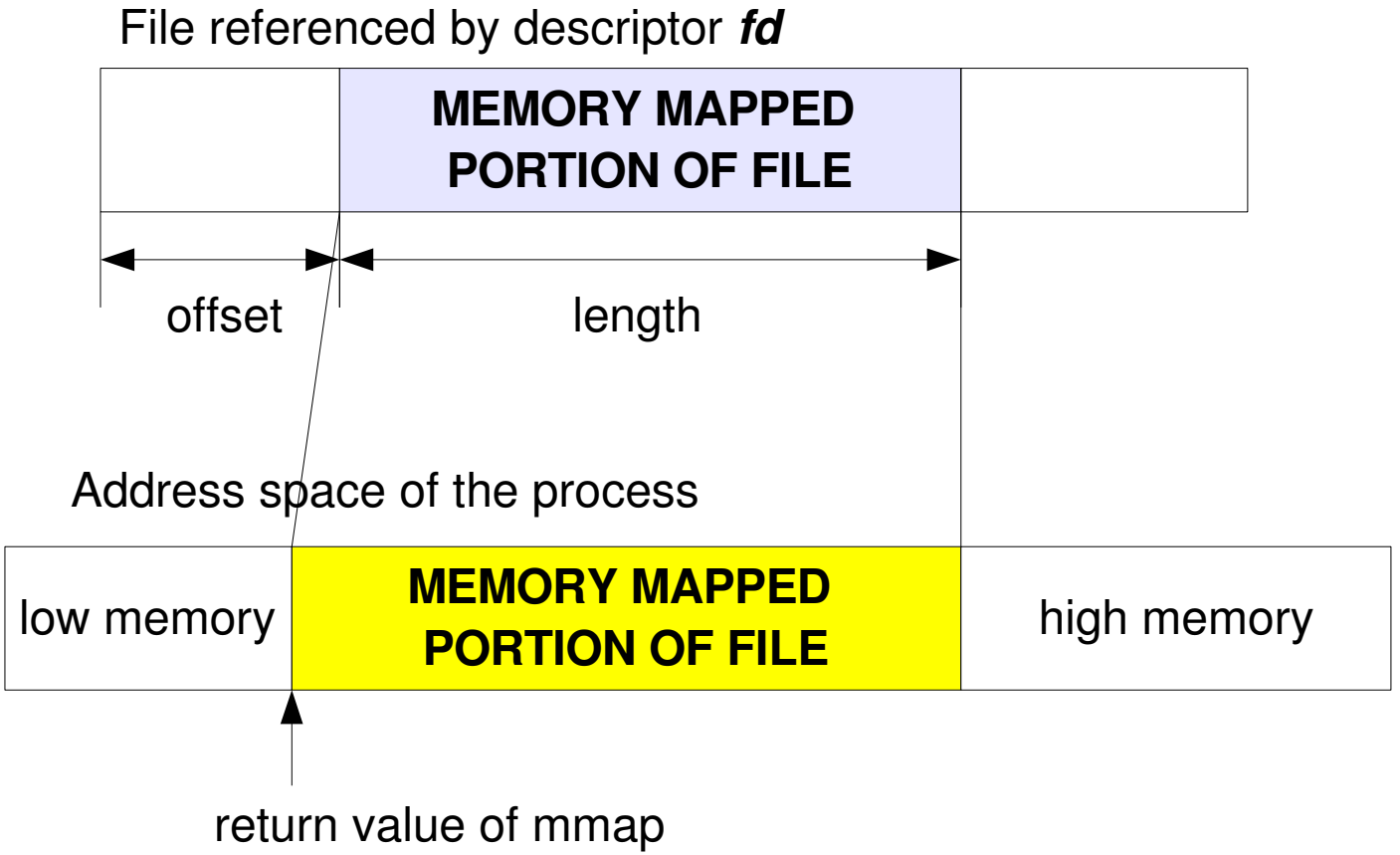
```
void  * mmap(void *start, size_t length, int prot,
  int flags, int fd, off_t offset);

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

prot: PROT_READ, PROT_WRITE, etc.

flags: MAP_SHARED, MAP_PRIVATE etc.

Example of memory-mapped file



Shared Memory Example

(UNP vol.2, W.Richard Stevens)

```
int main(int argc, char **argv) {
    unsigned char  *ptr;
    ...
    fd = shm_open(argv[1], O_RDWR, FILE_MODE);
    ptr = mmap(NULL, MY_SHARED_SIZE,
          PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

    close(fd);

    for (i = 0; i < MY_SHARED_SIZE; i++)
        *ptr++ = i % 256;

    shm_unlink(ptr);
}
```

Shared Memory - Summary

- Shared Memory is an IPC with **least latency**.
- It requires more work from the users in terms of explicit synchronization.
- We use the function *mmap()* to map a shared memory object into the address space of a process.
- Unless explicitly removed through *shm_unlink()* call, the shared memory object is **kernel-persistent**.

SOCKET PROGRAMMING

Socket Programming – Introduction

- Sockets are another IPC mechanism in Unix.
- Can be used to exchange data between processes on same machine or different machines
- Always use **network byte order** for all structures.
- Macros provided for conversion to network byte order: *htons(), htonl(), ntohs() and ntohl()*.
- Network data is **not** ASCII characters only and is treated as a stream of bytes. As such byte manipulation functions such as *bzero(), bcopy()* or the ANSI C variants *memset(), memcpy(), memcmp()* **must** be used.

Socket Programming – Data Structures

Socket Address structure to hold the IP address of the machine – look in */usr/include/netinet/in.h* in Linux.

```
struct in_addr {
   in_addr_t    s_addr;
};
struct sockaddr_in {
   uint8_t         sin_len;
   sa_family_t     sin_family;
   in_port_t       sin_port;
   struct in_addr  sin_addr;
   char            sin_zero[8];
};
```

Socket Programming – Utility Functions

Functions to convert from the human-readable dotted-decimal notation of an IP address to unsigned long and vice versa are :

```
int inet_aton(const char *cp, struct in_addr *inp);
   - Given dotted-decimal notation, returns IP in inp
in_addr_t inet_addr(const char *cp);
   - Returns IP given dotted-decimal notation
char *inet_ntoa(struct in_addr in);
   - Given IP in unsigned long, returns dotted-decimal
     notation string
```

Sample TCP Server Program

```
int main(int argc, char ** argv){
   int                  listenfd, connfd;
   struct sockaddr_in   servaddr, cliaddr;
   socklen_t            len = sizeof(cliaddr);

   listenfd = socket(AF_INET, SOCK_STREAM, 0);
   bzero(&servaddr, sizeof(servaddr));
   servaddr.sin_family = AF_INET;
   servaddr.sin_addr.s_addr = INADDR_ANY;
   servaddr.sin_port = htons(SERV_PORT);
   bind(listenfd, &servaddr, sizeof(servaddr));
   listen(listenfd, 5);
   for (; ;) {
      connfd = accept(listenfd, &cliaddr, &len);
      printf("Connection from client %s\n",
               inet_ntoa(cliaddr.sin_addr));
        read and write until condition is FALSE...
        close(connfd);
   }
}
```

Sample TCP Client Program

```
int main(int argc, char ** argv){
   int                    connfd;
   struct sockaddr_in     servaddr;
   char                  *serv_addr = “172.16.88.12”;

   connfd = socket(AF_INET, SOCK_STREAM, 0);
   bzero(&servaddr, sizeof(servaddr));
   servaddr.sin_family = AF_INET;
   inet_aton(serv_addr, &servaddr.sin_addr);
   servaddr.sin_port = htons(SERV_PORT);
   connect(connfd, &servaddr, sizeof(servaddr));

   while (!condition) {
      ...write() and read() data from server...
   }
   close(connfd);
}
```

Socket Function Calls – Data Structures and Exceptions

- Incomplete and Complete Queues
- Connection abort before *accept()* returns - ECONNABORTED
- Server terminates and Client is waiting for input on another *fd* – SIGPIPE error
- Server crashes – ETIMEDOUT vs EDESTUNREACH
- Server reboots – ECONNRESET

I/O Multiplexing: *select()*

- Need for I/O Multiplexing – especially with blocking file descriptors

```
int select(int n, fd_set *readfds,
      fd_set *writefds, fd_set *exceptfds,
      struct timeval *timeout);
```

- *select()* returns when
  - any of the descriptors in **read-set** are ready
  - any of the descriptors in **write-set** are ready
  - any of the descriptors in **exception-set** are ready
  - **timeout** has occurred

Example Program Showing use of *select()*

```
int main(int argc, char **argv) {

   int       connfd;
   fd_set    rset;
   ...Open socket and accept connection as earlier...

   FD_ZERO(&rset);
   for (; ;) {
      FD_SET(stdin, &rset);
      FD_SET(connfd, &rset);
      maxfd = max(stdin, connfd) + 1;
      select(maxfd, &rset, NULL, NULL, NULL);
      if (FD_ISSET(stdin, &rset))
         printf("Input from stdin\n");
      else if (FD_ISSET(connfd, &rset))
         printf("Input from socket\n");
      else
         printf("Error\n");
   }
}
```

Socket Options

- Getting and Setting some options for sockets helps in controlling the features of sockets.
- Examples:
  - SO_KEEPALIVE along with TCP_KEEPALIVE
  - SO_REUSEADDR
  - SO_LINGER (controls what TCP does with outstanding data on the socket)
  - IP_TTL
  - TCP_NODELAY (disable Nagle Algorithm)

Socket Options Function Calls

Two functions are defined to get and set socket options:

```
int getsockopt(int s, int level, int optname,
  void *optval, socklen_t *optlen);


int setsockopt(int s, int level, int optname,
  const void *optval, socklen_t optlen);
```

Example UDP Server

```
int main(int argc, char **argv) {
   int        sockfd, addrlen;
   struct sockaddr_in servaddr, cliaddr;

   addrlen = sizeof(servaddr);
   sockfd = socket(AF_INET, SOCK_DGRAM, 0);
   servaddr.sin_family = AF_INET;
   servaddr.sin_port = htons(SERV_PORT);
   servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
   bind(sockfd, &servaddr, addrlen);

   for (; ;) {
      n = recvfrom(sockfd, mesg, MAXBYTES, 0,
       &cliaddr, &addrlen);
      printf("Recvd %s from client\n", mesg);
      sendto(sockfd, mesg, n, 0, &cliaddr, addrlen);
   }
}
```

Elementary Name and Address Conversions

- To connect to a machine, typically, we need to contact the DNS server to get a name-to-address translation.
- Functions that allow this are:

```
struct hostent *gethostbyname(const char *name);

struct hostent *gethostbyaddr(const void *addr,
  int len, int type);

struct servent *getservbyname(const char *name,
  const char *proto);

struct servent *getservbyport(int port, const
  char *proto);
```

Advanced I/O

- ***recv*** and ***send*** functions

```
ssize_t recv(int sockfd, void *buff, size_t
  nbytes, int flags);

ssize_t send(int sockfd, const void *buff, size_t
  nbytes, int flags);
```

flags is one of the following:

MSG_DONTROUTE, MSG_DONTWAIT, MSG_OOB, MSG_PEEK, MSG_WAITALL

Advanced I/O (contd.)

- Scatter and Gather I/O functions – ***writev*** and ***readv***

```
ssize_t readv(int fd, const struct iovec *iov,
  int iovcnt);

ssize_t writev(int fd, const struct iovec *iov,
  int iovcnt);

struct iovec {

  void *iov_base;

  int iov_len);

}
```

Data Structures Needed for Scatter/Gather Example

```
typedef struct data_s{
  int       opcode;
  int       len;
} data_t;

typedef struct std_s {
  int   rollno;
  char  name[64];
  int   cgpa;
} std_t;

#define ADD_STUDENT 1
```

Example Client using ***writev***

```
int main(int argc, char **argv)
{
  int  fd, rc;
  std_t s;
  data_t d;
  struct iovec        iov[2];

  s.rollno = 10;   s.cgpa = 9;
  strcpy(s.name, "APCS");

  /* Open socket and connect to server */
  ....
  d.opcode = ADD_STUDENT;   d.len    = sizeof(s);
  iov[0].iov_base = &d; iov[0].iov_len  = sizeof(d);
  iov[1].iov_base = &s;iov[1].iov_len  = sizeof(s);
  rc = writev(fd, iov, 2);

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

Example Server using ***readv***

```
int main(int argc, char **argv)
{
  int  fd, rc;
  std_t s;
  data_t d;
  struct iovec        iov[2];

  /* Open socket, bind, listen and accept conn. */
  ......

  iov[0].iov_base = &d; iov[0].iov_len  = sizeof(d);
  iov[1].iov_base = &s;iov[1].iov_len  = sizeof(s);
  rc = readv(fd, iov, 2);
  printf("Opcode = %d\n", d.opcode);
  printf("Roll no = %d, Name = %s, CGPA = %d\n",
  s.rollno, s.name, s.cgpa);

  close(connfd); close(listenfd);
  exit(0);
}
```

Socket Programming - Summary

- TCP server opens a passive socket whereas a client opens an active socket.
- TCP sockets have **incomplete** and **complete** queues associated with them.
- UDP clients need to timeout as packets can get lost.
- Always use SO_REUSEADDR socket option.
- Use of *select()* allows a process to wait on multiple file descriptors for **read**, **write** and **exception** conditions.
- DNS resolution is done by *gethostbyname()* function.

Daemon Processes

- Properties of a daemon:
  - It is a process that **starts at bootup** time and runs in the **background** until the system is shut down
  - Has no **controlling terminal**
  - Error messages are logged using the **syslog daemon**
  - Examples are all the standard network servers

Daemon Initialization Function (UNP Vol.1, Fig. 12.4)

```
int daemon_init(void) {
   pid_t          pid;

   if ((pid = fork()) != 0)
     exit(0);

   /* Make the child process session leader */
   setsid();

   /* Ignore SIGHUP signal                    */
   signal(SIGHUP, SIG_IGN);
   if ((pid = fork()) != 0)
     exit(0);

   chdir("/");
   umask(0);

   /* Close all unused file descriptors */
   return 0;
}
```

syslog Function

syslog is a function that allows daemons to log different types of messages that help with debugging/troubleshooting a system/service.

The prototype for ***syslog*** is

```
void syslog(int priority, char *format, ...);
```

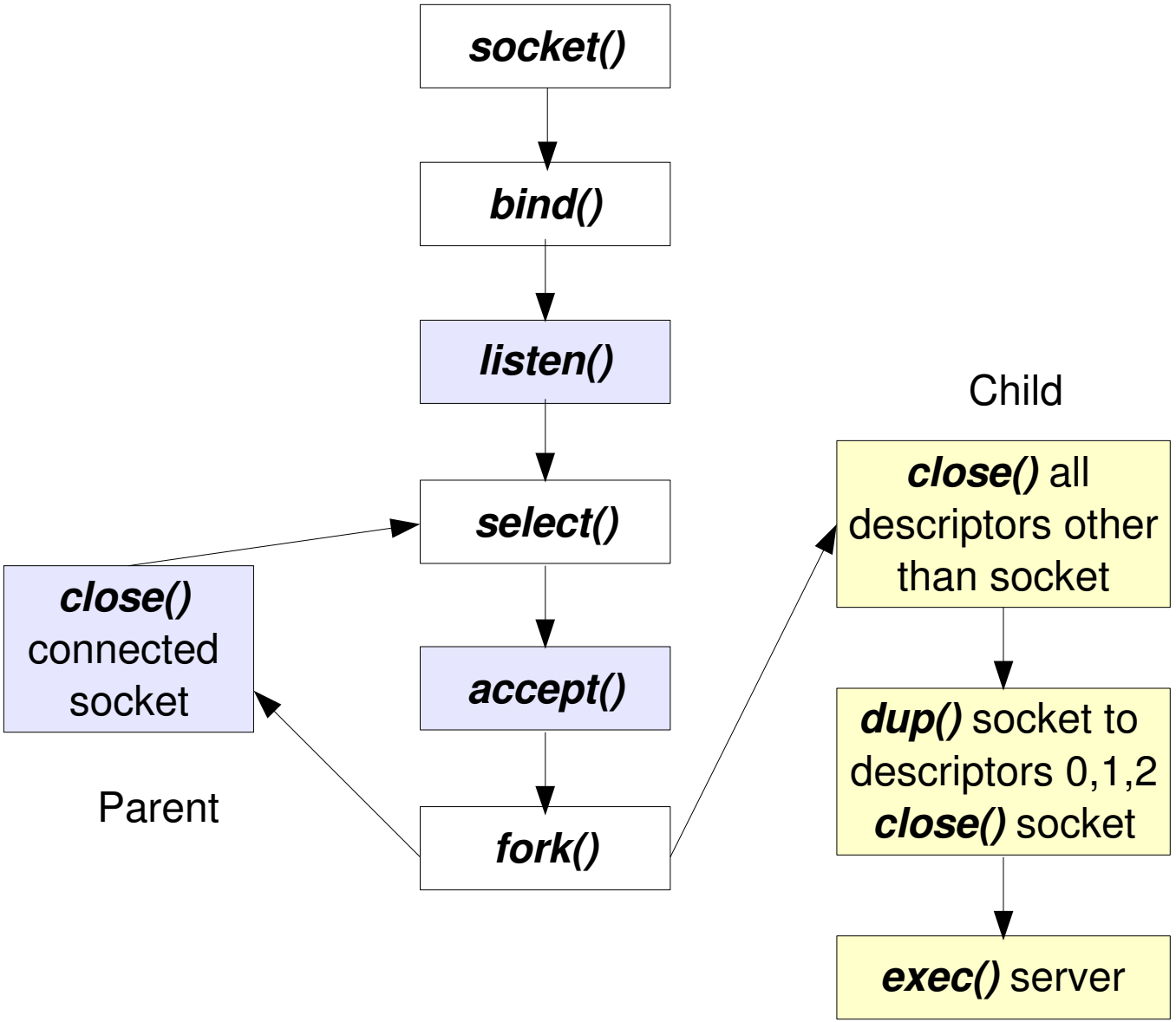
priority is a combination of *level* and *facility* of the message, *format* is as in a ***printf***.

LOG_ERR, LOG_CRIT, LOG_WARNING, LOG_INFO, LOG_DEBUG are some levels.

inetd Superserver

- **inetd** allows efficiency in writing network servers by:
  - simplification of writing daemon processes
  - allows a single process to wait for clients of multiple services instead of multiple processes most of which are idle
  - Examples are Telnet, Rlogin, FTP etc.
  - The services **inetd** handles are stated in the configuration file /etc/inetd.conf in Unix (in Linux, it is in /etc/xinetd.conf and in /etc/xinetd.d/* files.

Flowchart for **inetd** Superserver



Telnet *inetd* configuration file

```
service telnet
{
   socket_type     = stream
   protocol        = tcp
   wait            = no
   user            = root
   server          = /usr/sbin/in.telnetd
   disable         = yes
}
```

REFERENCES

- *Advanced Programming in the Unix Environment*, W.Richard Stevens (for Process Control, Signals)
- *Unix Network Programming, vol. 2,* W.Richard Stevens (for Pipes, FIFOs, Mutexes, Condition Variables, Semaphores and Shared Memory)
- *Unix Network Programming, vol. 1,* W.Richard Stevens (for Socket programming)