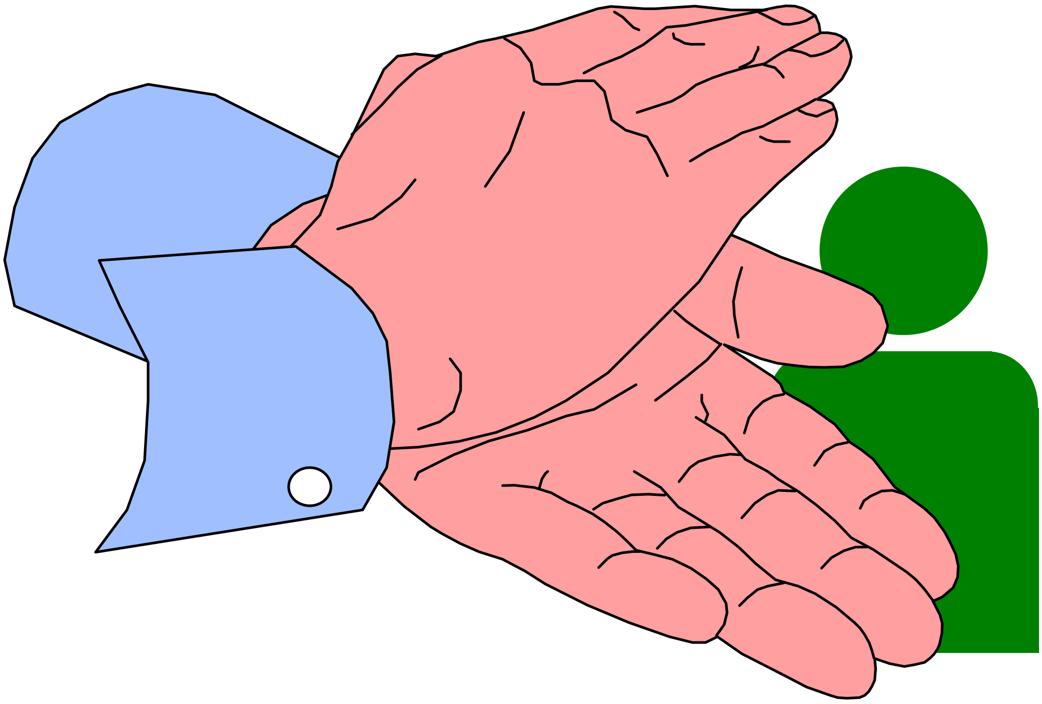
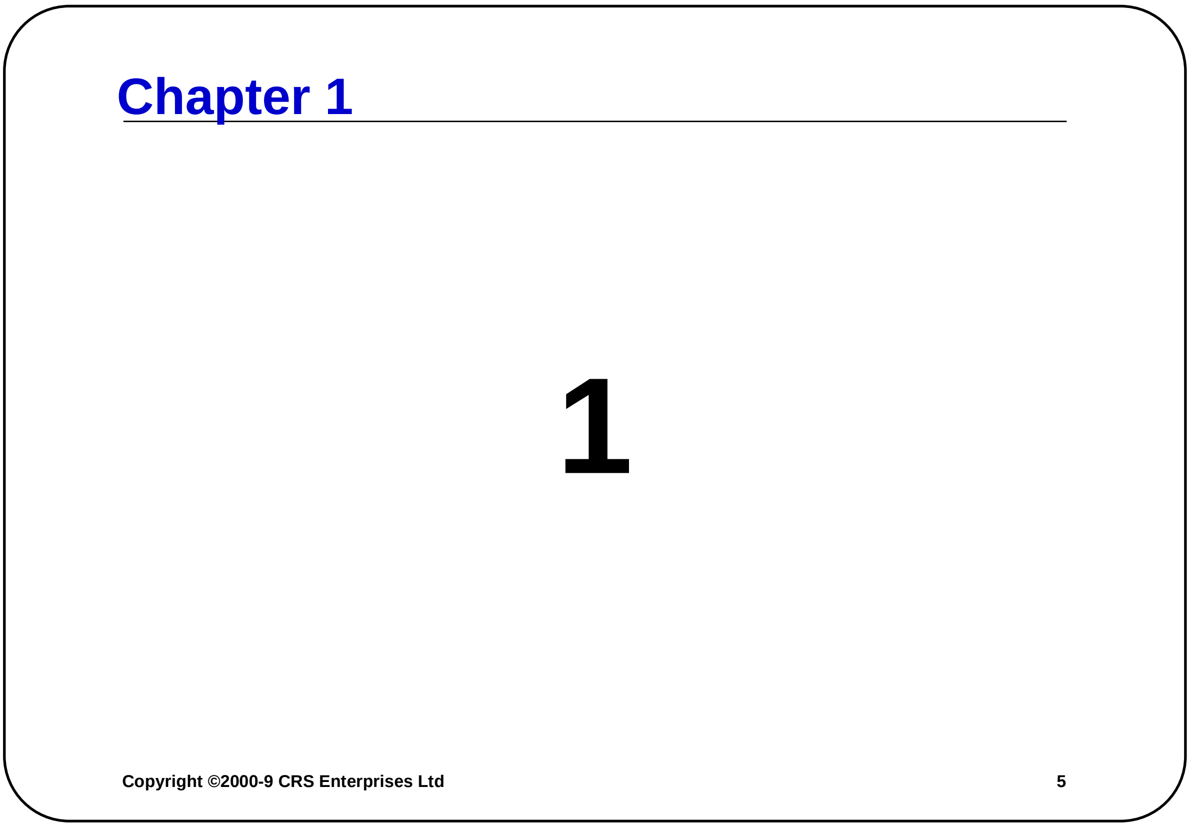
Linux Programming



Chris Seddon

seddon-software@keme.co.uk

Linux Programming



Chapter 1

1

The Unix Model

1

6

Unix for Programmers

Structure of Unix

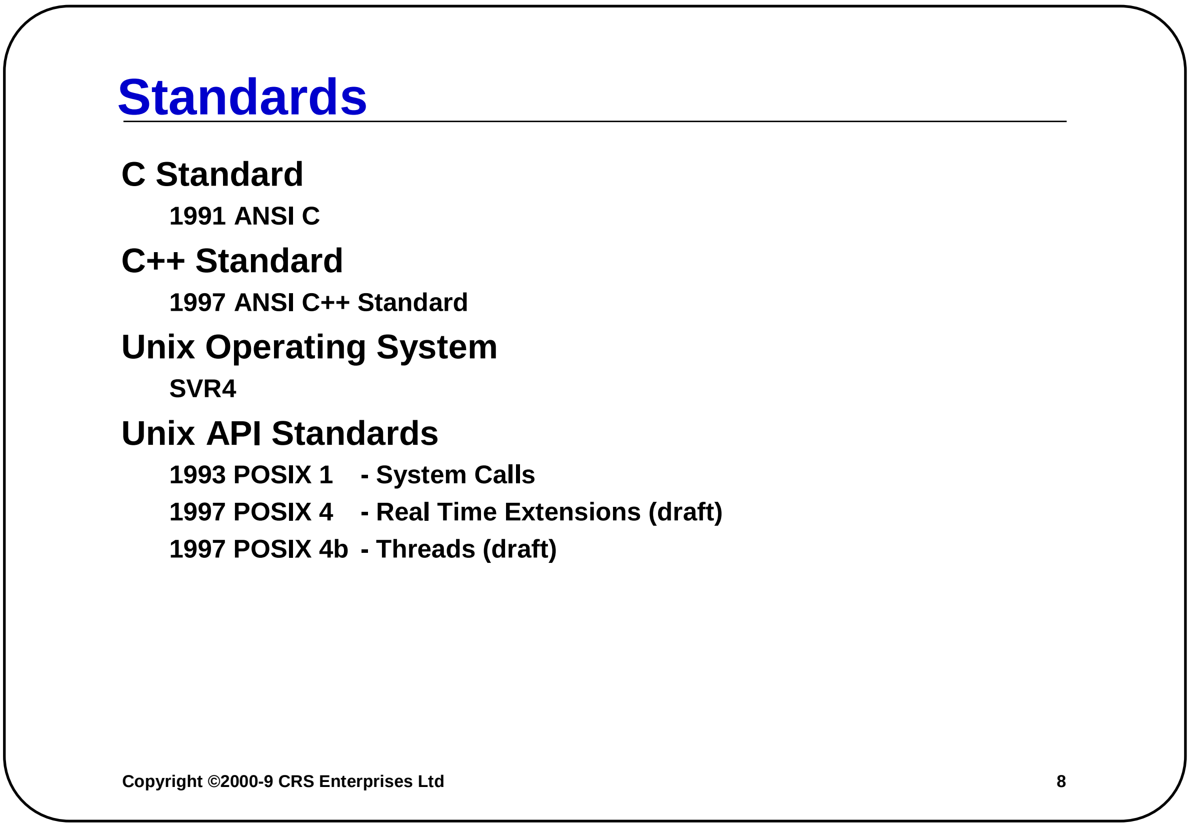
Debugging

Standards

Processes

System Calls and Library Functions

Error Handling



Standards

C Standard

1991 ANSI C

C++ Standard

1997 ANSI C++ Standard

Unix Operating System

SVR4

Unix API Standards

1993 POSIX 1 - System Calls
1997 POSIX 4 - Real Time Extensions (draft)
1997 POSIX 4b - Threads (draft)

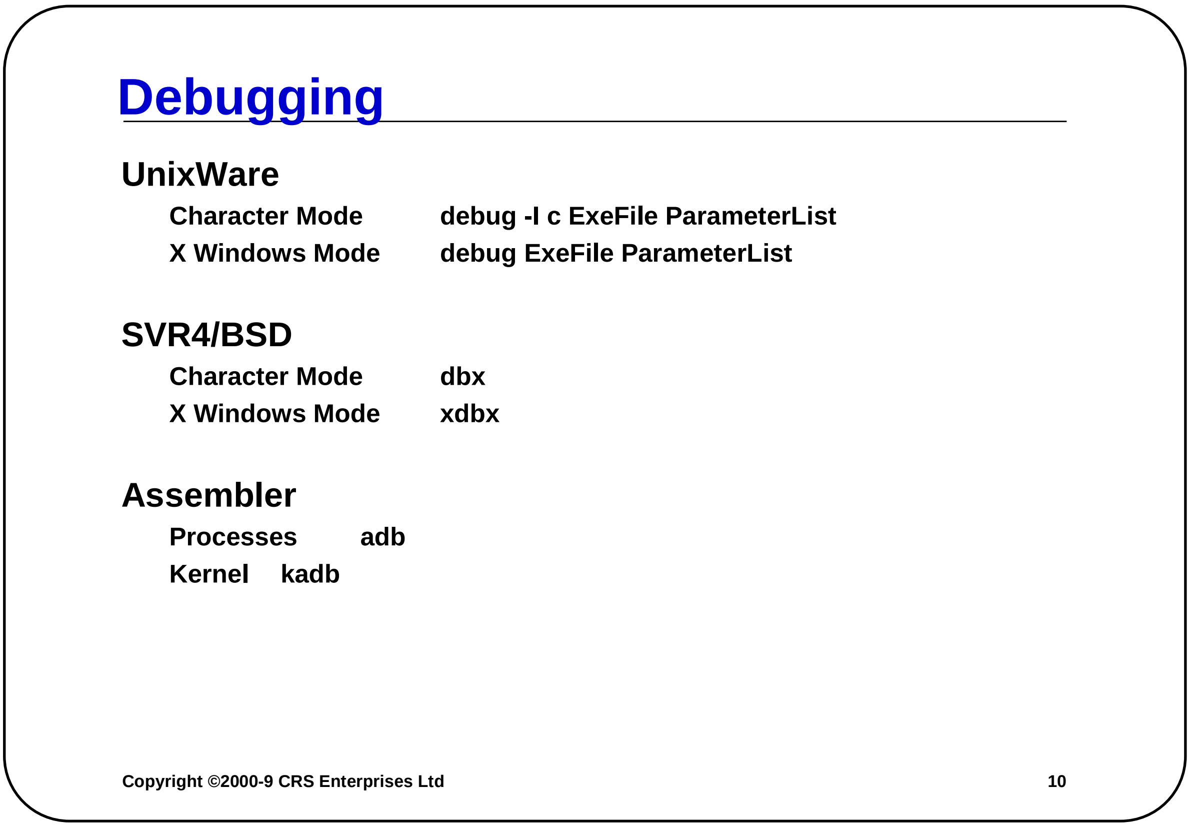
Documentation

Manual Pages

Volume 1	Unix Commands
Volume 2	System Calls
Volume 3	Library Functions
Volume 5	Data Structures

X Windows Help

Xman



Debugging

UnixWare

Character Mode	debug -I c ExeFile ParameterList
X Windows Mode	debug ExeFile ParameterList

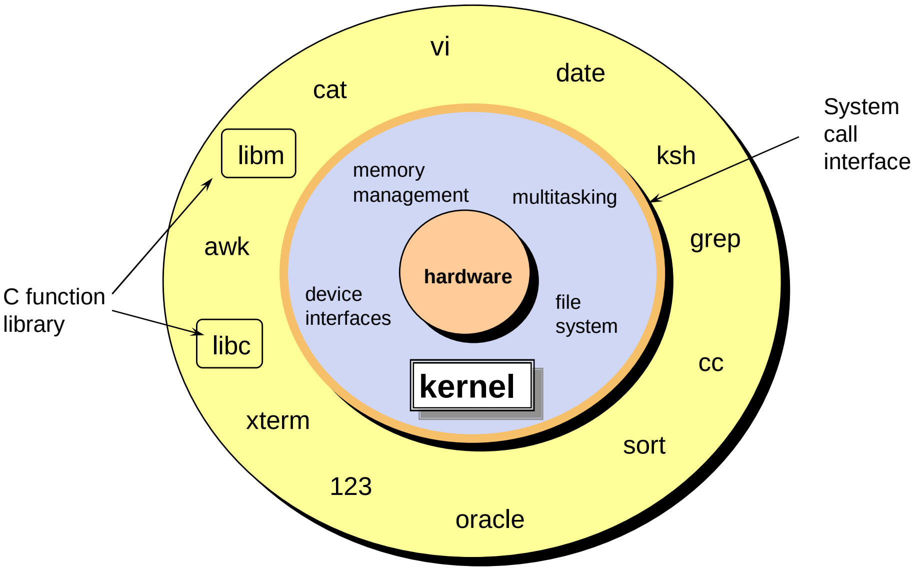
SVR4/BSD

Character Mode	dbx
X Windows Mode	xdbx

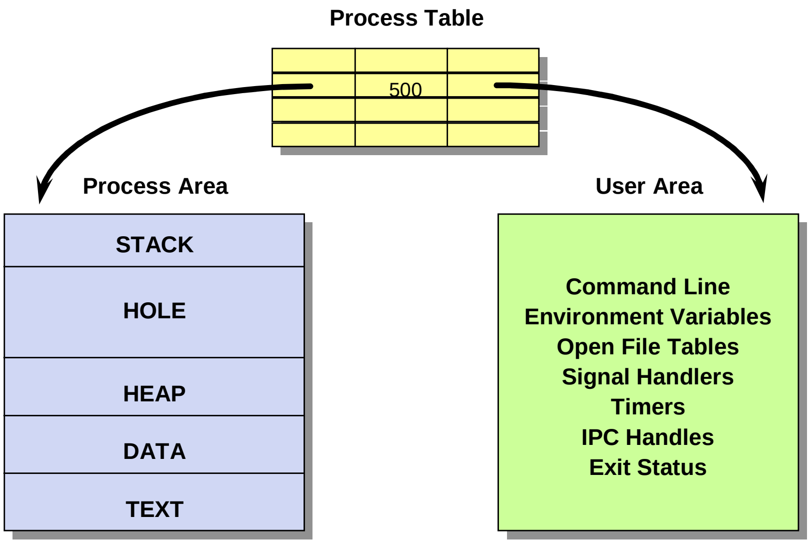
Assembler

Processes	adb
Kernel	kadb

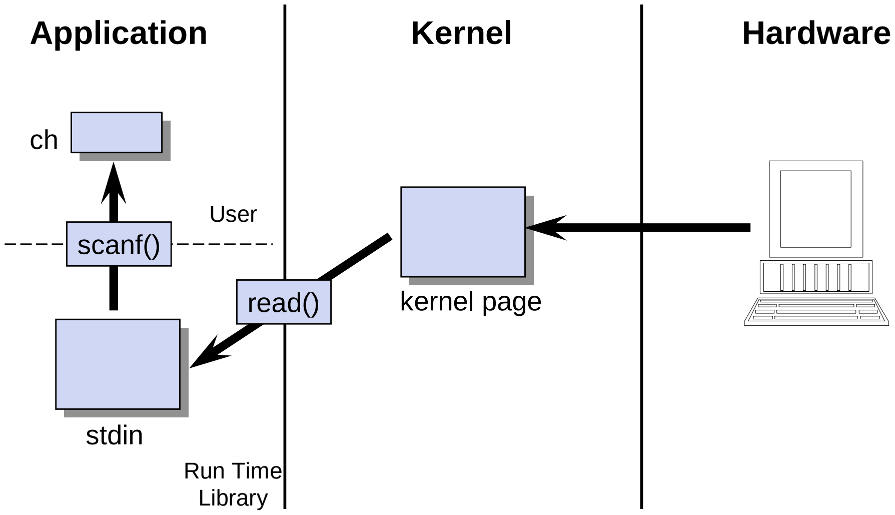
The Structure of Unix



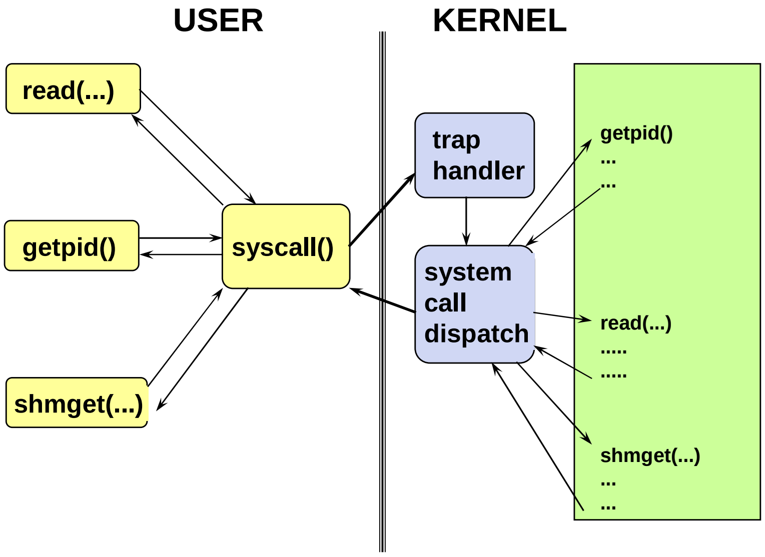
Processes

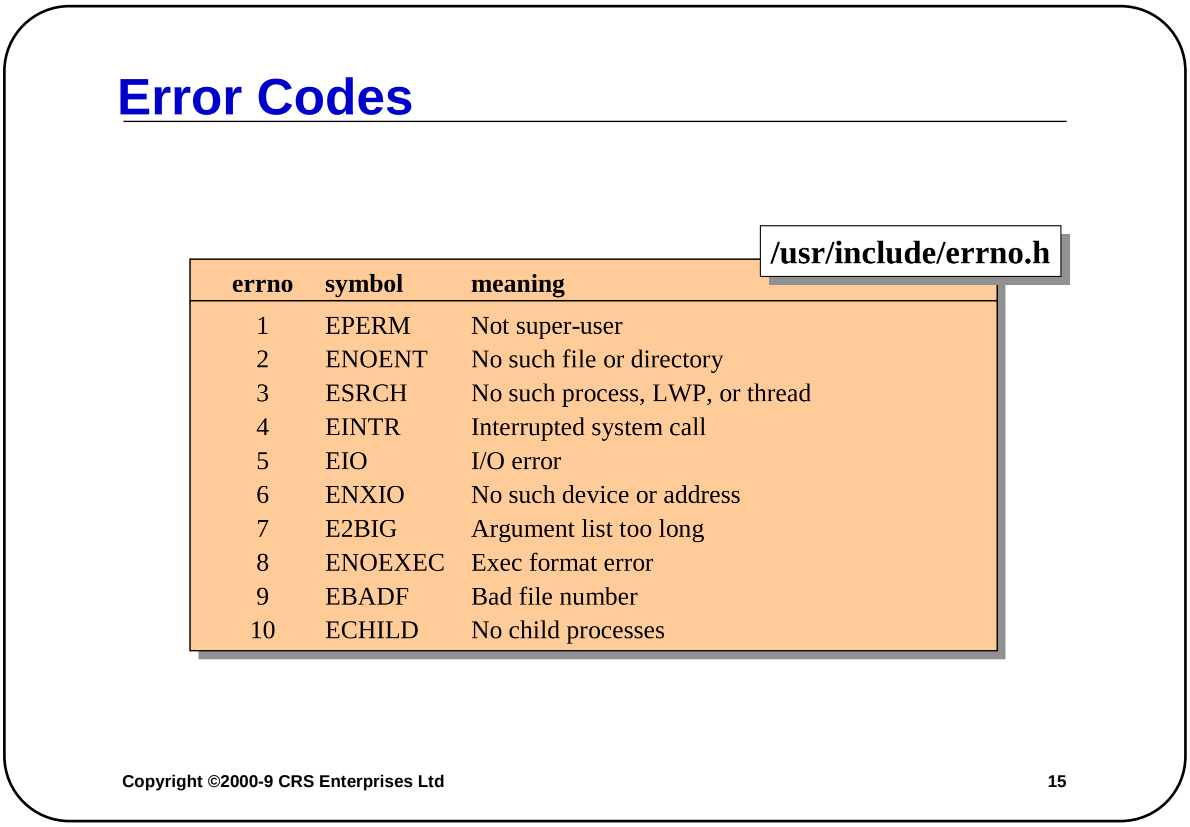

Process Table
500
Process Area
STACK
HOLE
HEAP
DATA
TEXT
User Area
Command Line
Environment Variables
Open File Tables
Signal Handlers
Timers
IPC Handles
Exit Status

System Calls and Library Functions



Making System Calls

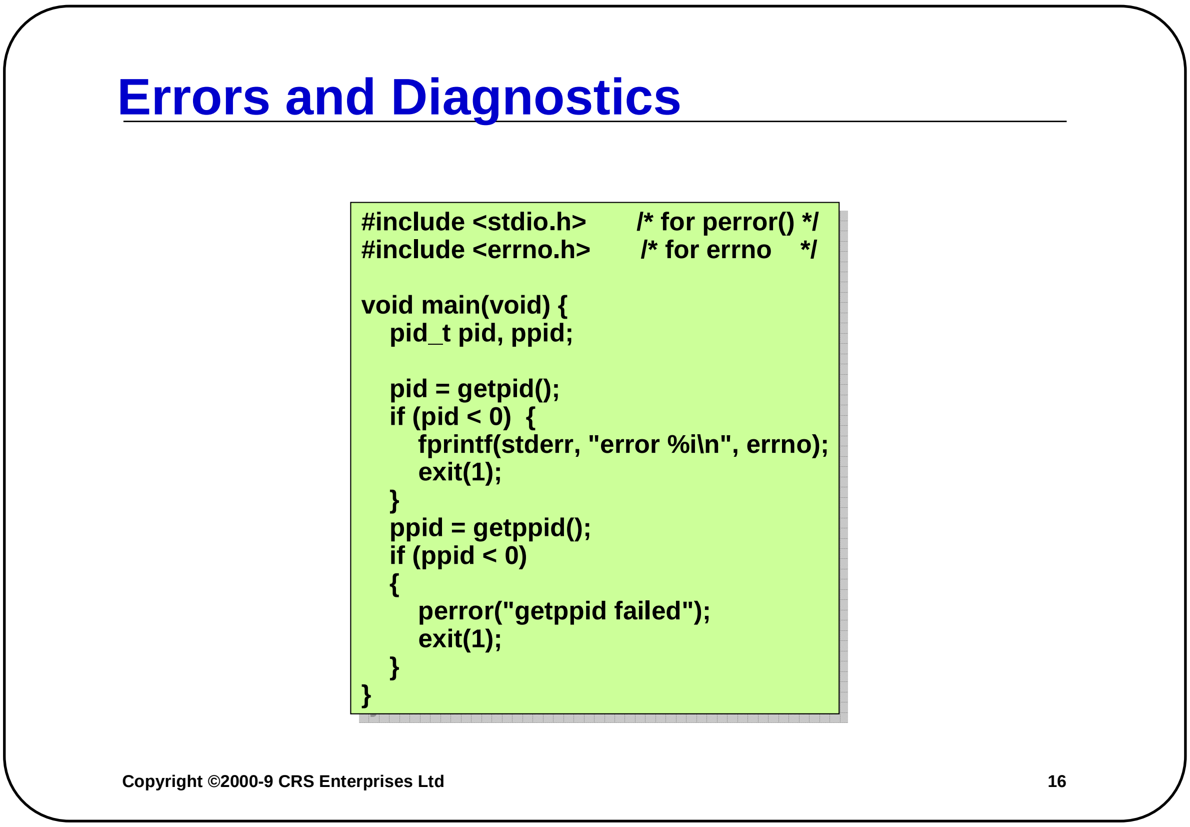




Error Codes

/usr/include/errno.h

errno	symbol	meaning
1	EPERM	Not super-user
2	ENOENT	No such file or directory
3	ESRCH	No such process, LWP, or thread
4	EINTR	Interrupted system call
5	EIO	I/O error
6	ENXIO	No such device or address
7	E2BIG	Argument list too long
8	ENOEXEC	Exec format error
9	EBADF	Bad file number
10	ECHILD	No child processes



Errors and Diagnostics

```
#include <stdio.h>       /* for perror() */
#include <errno.h>       /* for errno    */

void main(void) {
   pid_t pid, ppid;

   pid = getpid();
   if (pid < 0)  {
      fprintf(stderr, "error %i\n", errno);
      exit(1);
   }
   ppid = getppid();
   if (ppid < 0)
   {
      perror("getppid failed");
      exit(1);
   }
}
```

Copyright ©2000-9 CRS Enterprises Ltd 16

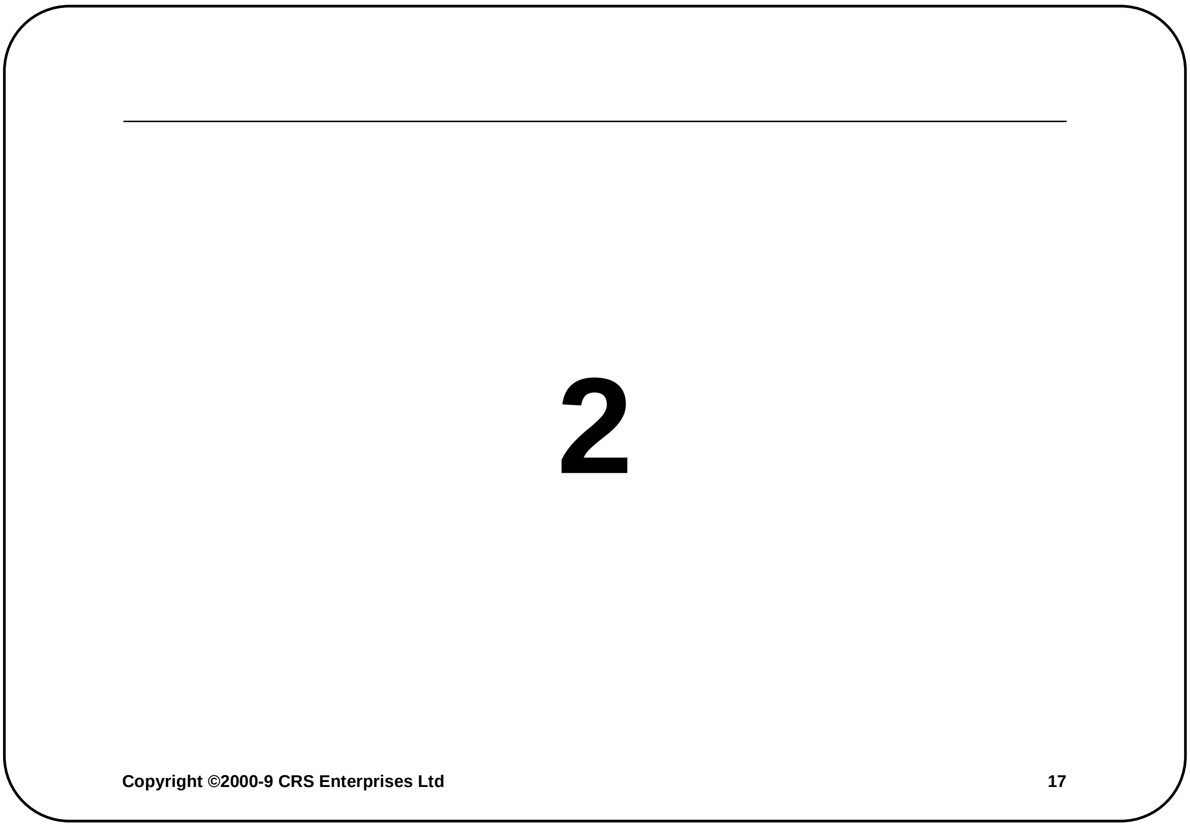
The example above shows two typical system calls:

getpid()	returns the process's id.
getppid()	returns the parent's process id.

In both cases we check the return code from the system call to see if it has failed. With getpid() we print a message that echoes the value of errno. Since we included the file <errno.h>, we have direct access to the variable. With getppid() we use perror() to print a meaningful message based on errno. perror() prints its parameter followed by the error message, so we may see something like:

get ppid failed: no such process

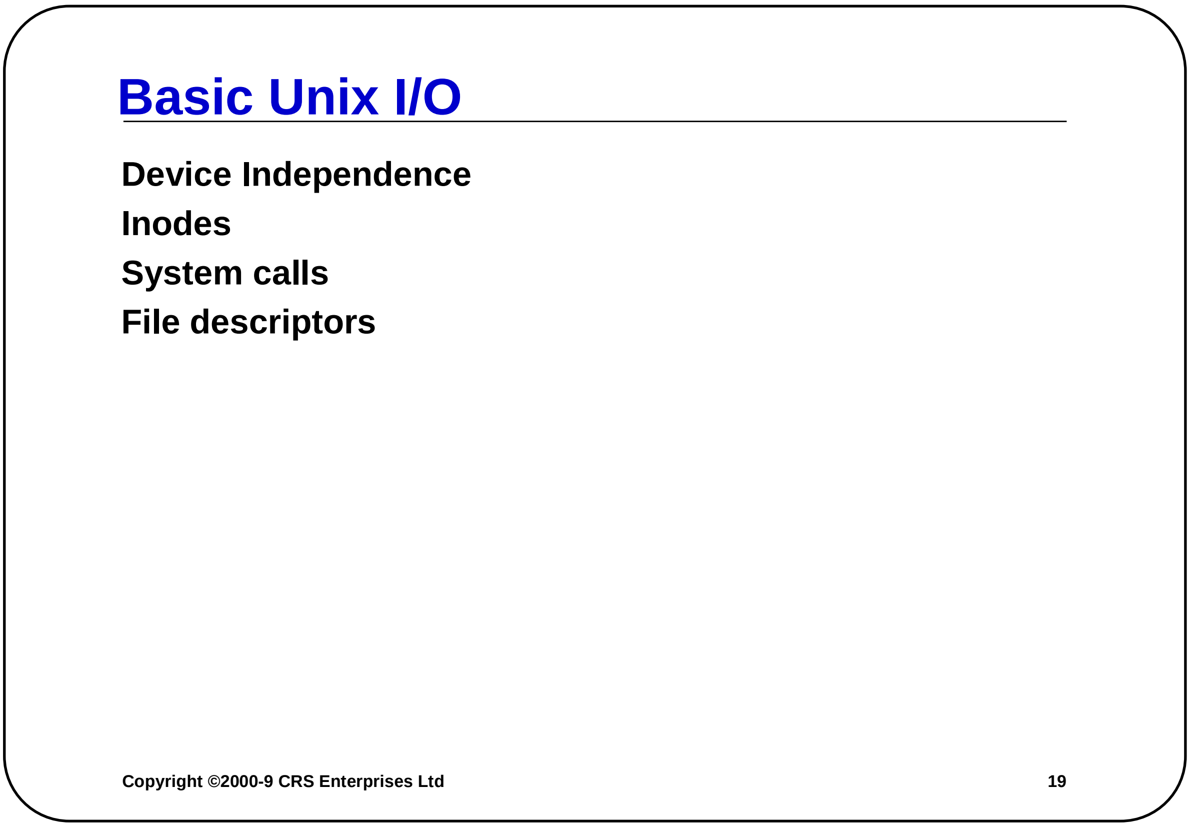
The manual pages for the system calls will contain definitive information about the possible error conditions and what may cause them. A list of error numbers and descriptions can be found under intro in volume 2 of the manual pages.



2

File Input and Output

2



Basic Unix I/O

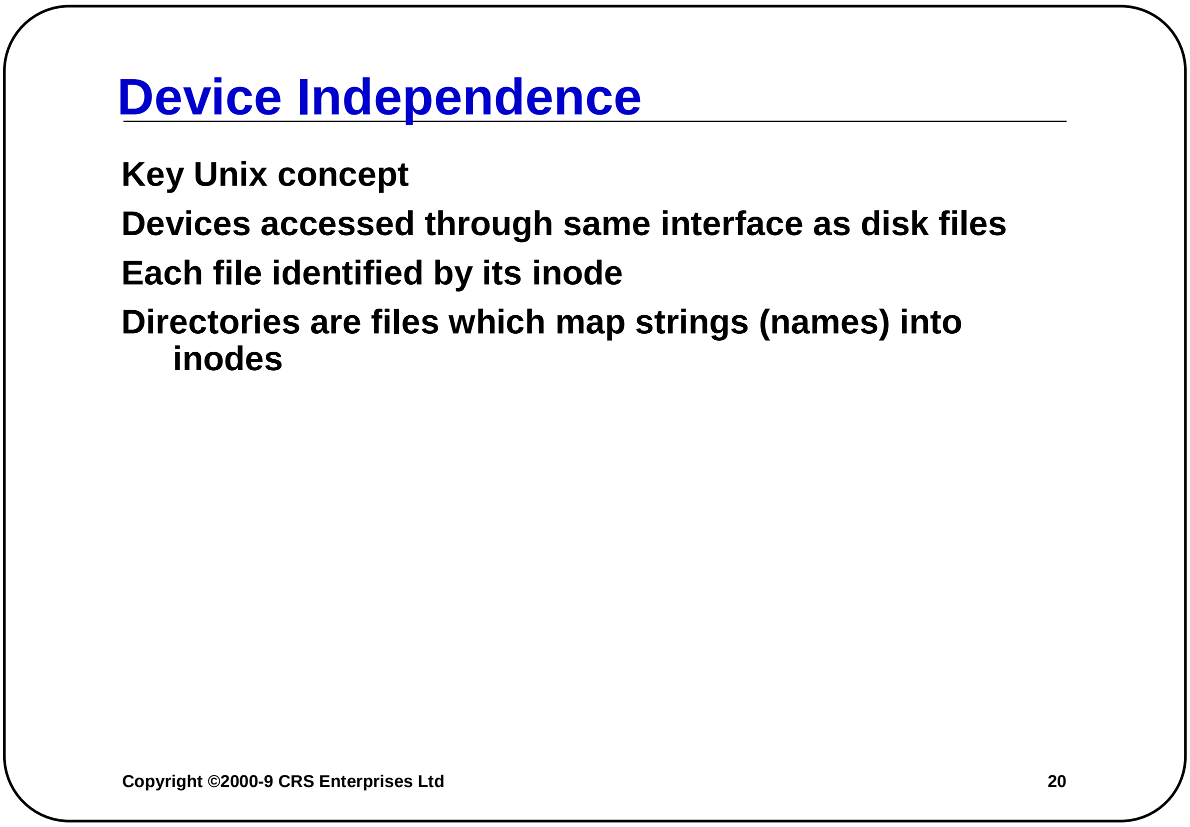
Device Independence

Inodes

System calls

File descriptors

This chapter will review standard I/O routines for accessing files, introduce the concept of the virtual file interface and explain inodes and file descriptors. The standard low-level I/O routines are also described in detail.



Device Independence

Key Unix concept

Devices accessed through same interface as disk files

Each file identified by its inode

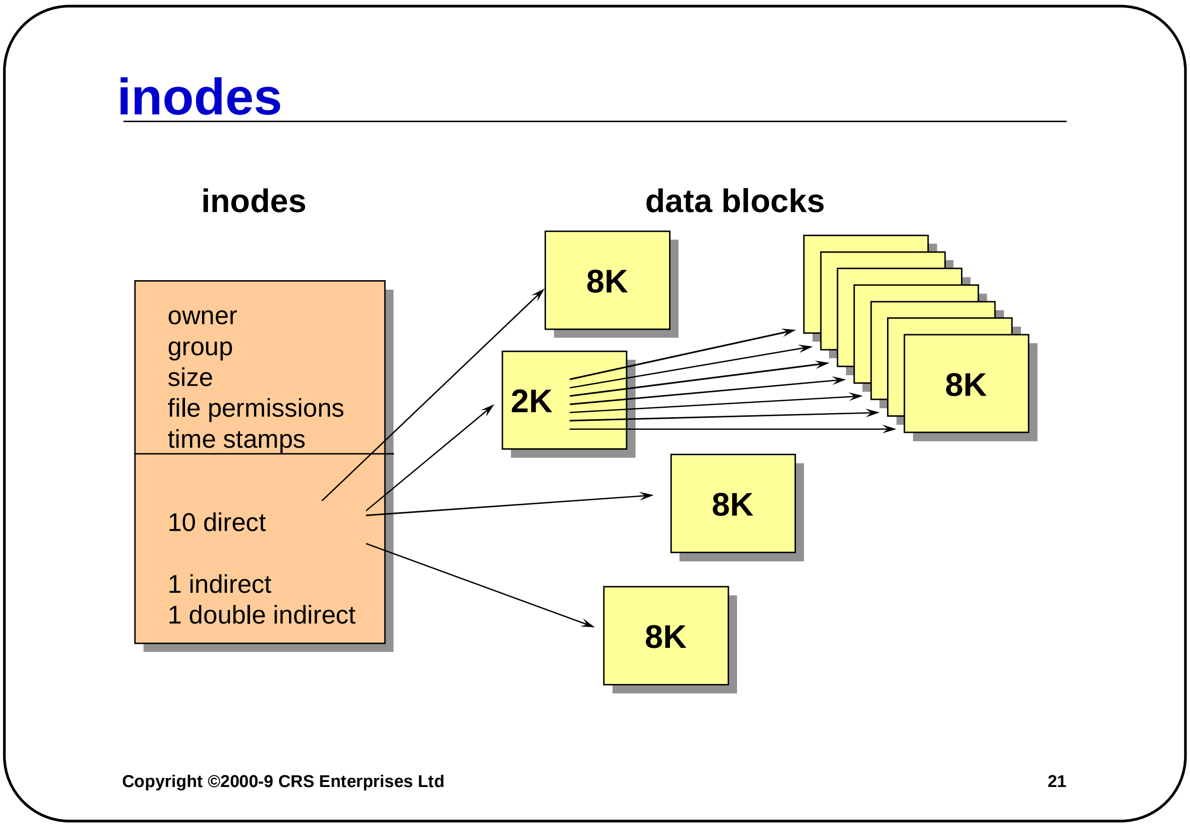
Directories are files which map strings (names) into inodes

Unix has a virtual-file interface to devices, which means that all devices are accessed in the same manner as files. The Unix file-system routines in the kernel recognize when a special file (e.g. a device) is accessed and redirect the I/O operations to the appropriate handling code. The device-handler routines then map the file operations into equivalent device operations.

In order to keep track of all the files in a Unix hierarchical file system, there is a unique list of all the files on a file system. This list, called the inode list, is stored as part of the file-system structure information on the disk.

A directory is a file that contains records that map names onto inode numbers. When accessing files by name, Unix reads each part of the pathname and looks in the next directory for the inode corresponding to that name. If this file is a directory, it becomes the directory to search for the next component of the pathname.

More than one directory entry can refer to the same inode number, thereby allowing Unix file systems to have multiple links to the same file. Links are important because they allow a single file to be known by more than one pathname. Without links, the two special names "." and ".." would have to be treated as special cases when evaluating a pathname; with links, they are simply names in the directory file.



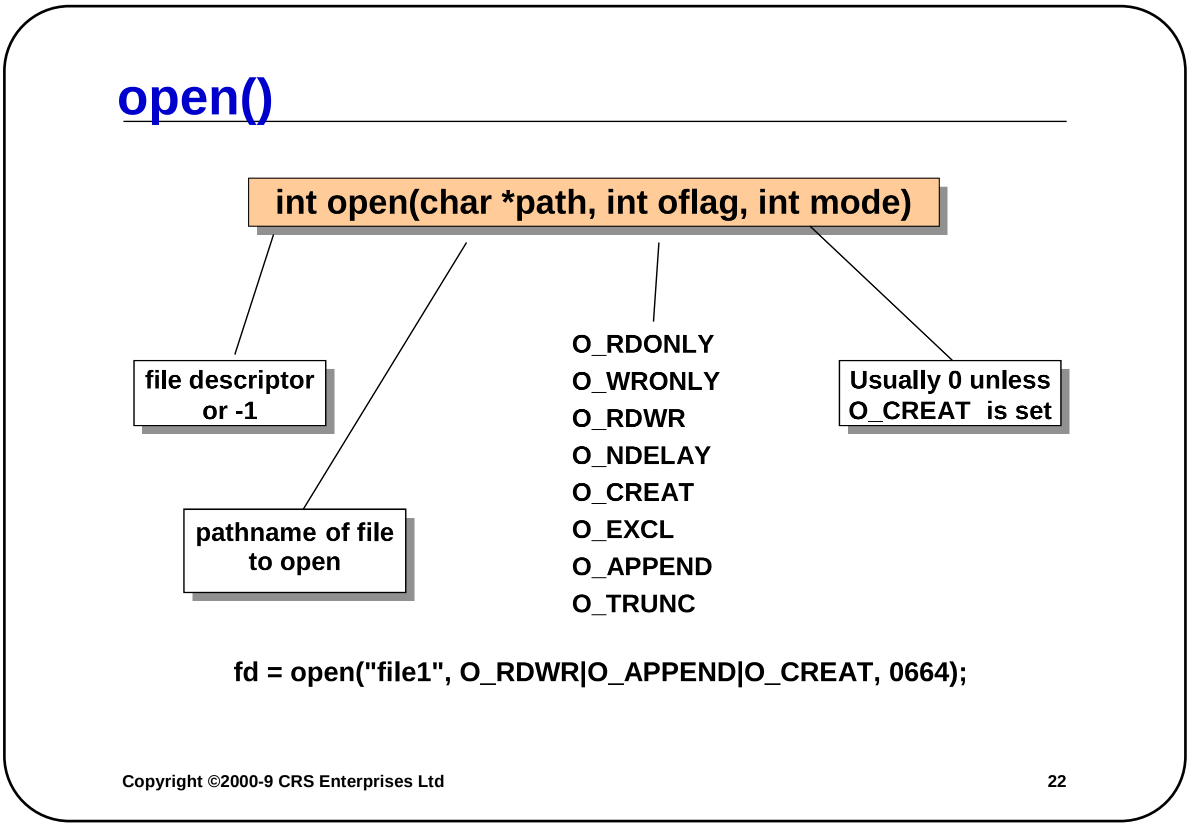
inodes

21

The inode is a data structure that describes the attributes of the file. These attributes include creation, modification and access times, owner and group ids, protection and attribute flags and type of file (file, directory, device, etc.). In the case of a regular file or directory, the inode also contains information that defines the blocks on the disk that contain the data. Every file has a single inode allocated to it. When the inode list is full, no new files can be created, even if the disk still has unused file-data blocks. A file's inode number can be obtained with the ls -i command.

A device file is fully defined by its inode; other types of file require allocated disk space to hold the data. The inode contains 13 pointers to data blocks. The first 10 point directly to the data blocks (direct pointers). The next pointer points to a block that contains pointers to the actual data blocks (indirect). The next inode pointer points to a block that contains pointers to the blocks that contain the pointers to the actual data blocks (double-indirect). The last inode pointer is a triple-indirect pointer (not used on modern systems).

Originally, the file system's data blocks were only 512 bytes. Using such a small block size gives very poor I/O performance and nowadays most systems use 8KB data blocks.



open()

int open(char *path, int oflag, int mode)

file descriptor or -1

pathname of file to open

O_RDONLY
O_WRONLY
O_RDWR
O_NDELAY
O_CREAT
O_EXCL
O_APPEND
O_TRUNC

Usually 0 unless O_CREAT is set

```
fd = open("file1", O_RDWR|O_APPEND|O_CREAT, 0664);
```

Copyright ©2000-9 CRS Enterprises Ltd 22

The open system call is used to open a file descriptor. File descriptors must be opened before they can be read from or written to. The first parameter is the full pathname of the file (relative or absolute) and the second parameter is the I/O access flags. The access flags contain one of the following:

O_RDONLY Open for read only.

O_WRONLY Open for write only.

O_RDWR Open for read and write.

Other modes can be used in conjunction with the access flags:

O_NDELAY If it is set, it will perform non-blocking I/O. Read will return immediately if no data present. Writes will initiate the write but not wait for completion. With this bit clear, reads will block until data arrives and writes will block until data has been written.

O_APPEND Set pointer to end of file prior to each write.

O_CREAT If the file exists, this has no effect. Otherwise, the file is created using the mode flag. See the next chapter for how to create files and other devices using this flag and other system calls.

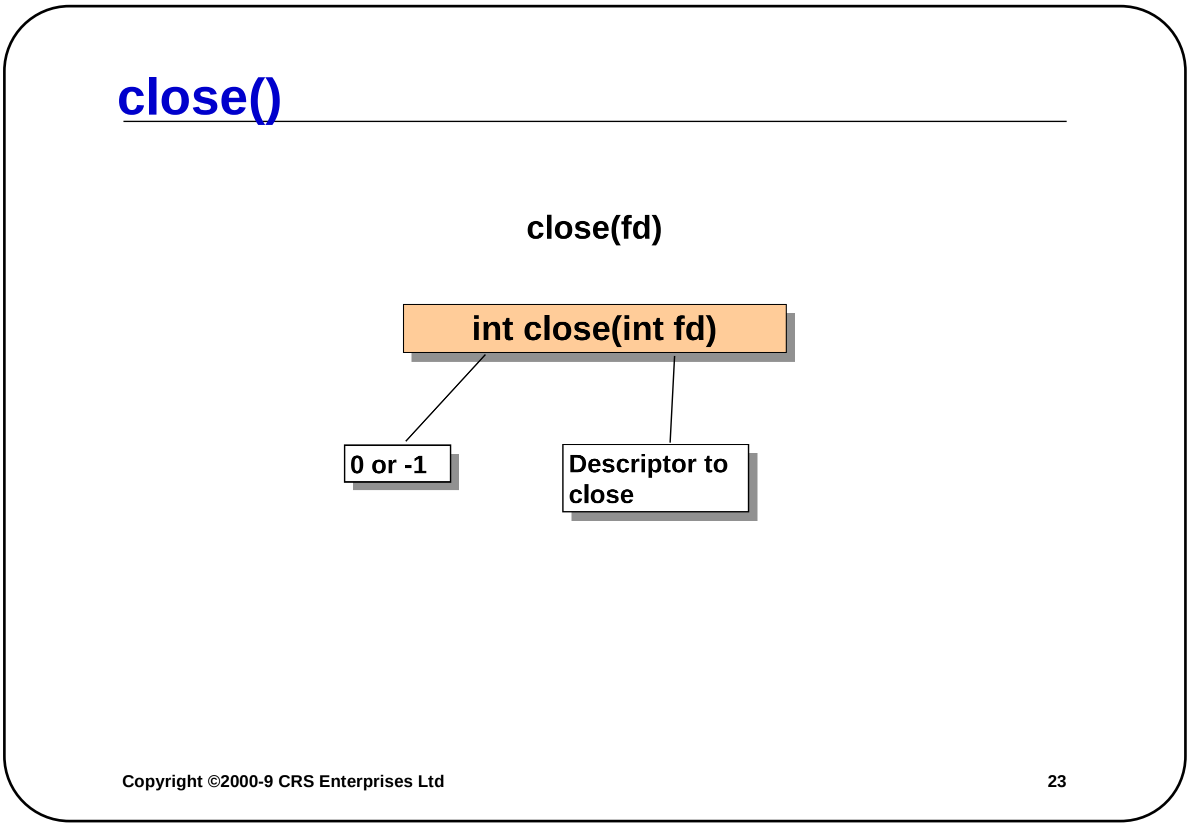
O_TRUNC If the file exists, length is truncated to 0.

O_EXCL If set with O_CREAT, the open will fail if the file exists.

The other bits can be "or"d into the flags to modify the I/O access. For example

```
fd = open(“file1”, O_WRONLY | O_CREAT | O_TRUNC, 0);
```

opens a file in write only mode (O_WRONLY), creates the file if it does not exist (O_CREAT) and clears out its contents if it does exist (O_TRUNC).



close()

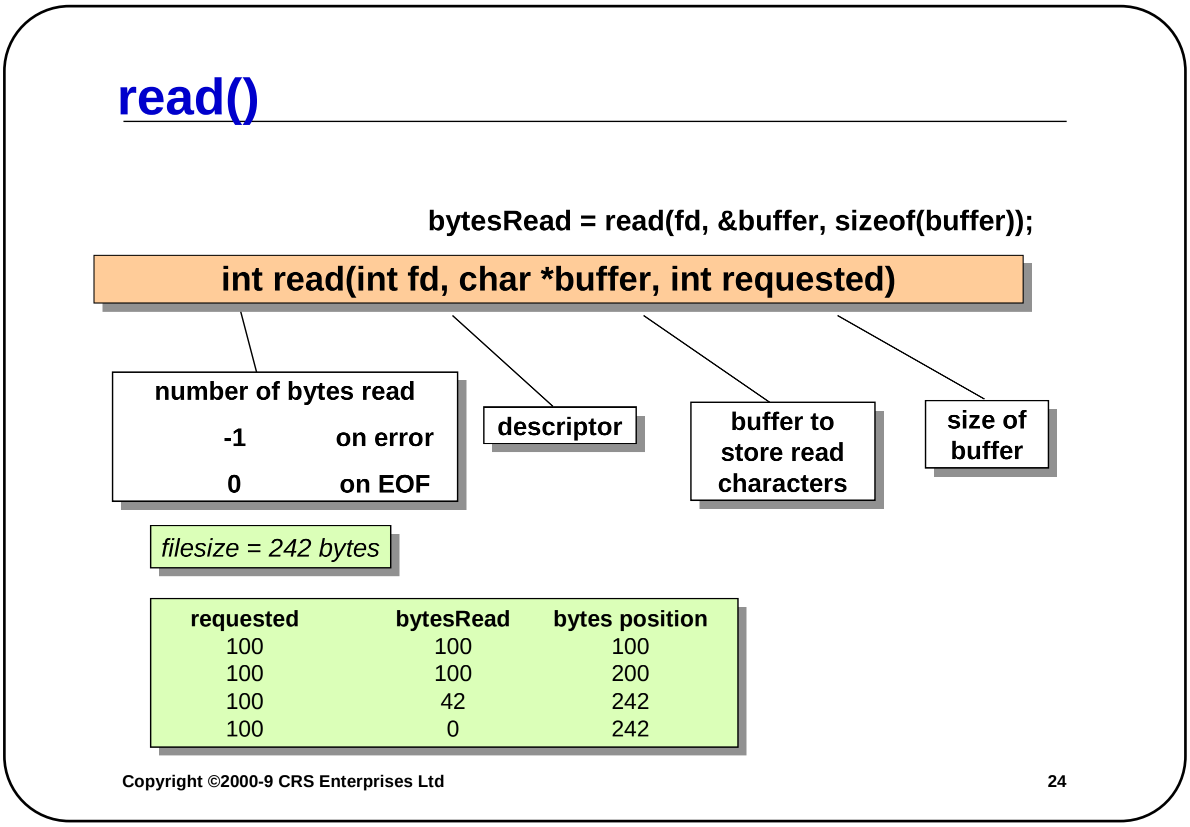
close(fd)

int close(int fd)

0 or -1

Descriptor to close

The close system call is used to close an open file descriptor. Close returns 0 on success and -1 on error, with errno set to indicate the error.



read()

```
bytesRead = read(fd, &buffer, sizeof(buffer));
```

```
int read(int fd, char *buffer, int requested)
```

number of bytes read

-1	on error
0	on EOF

descriptor

buffer to store read characters

size of buffer

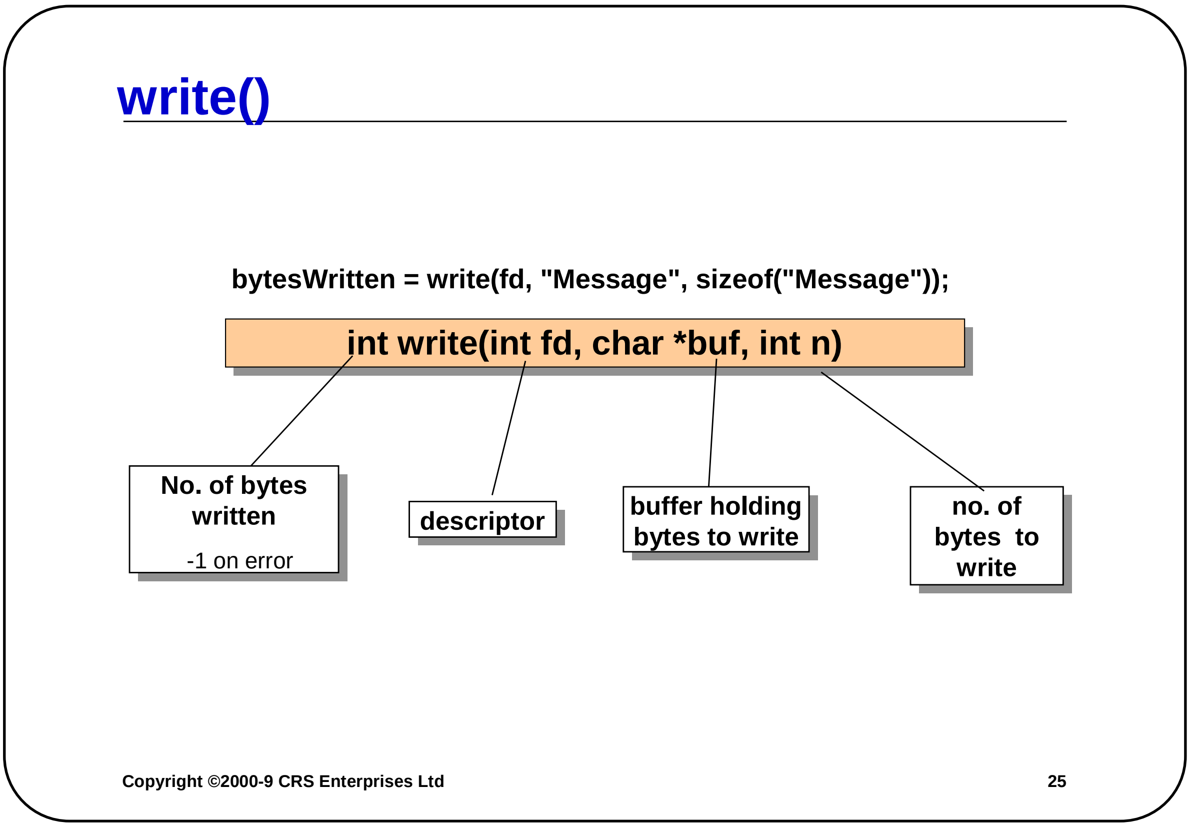
filesize = 242 bytes

requested	bytesRead	bytes position
100	100	100
100	100	200
100	42	242
100	0	242

The read system call is used to read data from a file or device. The data is stored in the specified buffer and the number of bytes written to the buffer are returned from the system call. A return value of 0 indicates end of file, and a return value of -1 indicates error (errno set to error code). The number of bytes read will be the number requested, except when the number of bytes left in a file is less than the number requested. When reading from serial lines (terminals), the number read will depend upon the mode of the terminal (see the description of terminals in the next chapter). In the simple case for terminals, a single line of data is read and saved in the buffer (including the terminating newline), unless this would exceed the number of bytes requested.

When reading from seeking devices (files), the data is read from the current file position and the file position is incremented by the number of bytes read (ready for the next read).

If the descriptor was opened with the O_NDELAY flag set, then reading from device or pipe that has no data available will return immediately with no data read. This situation is not the same as end of file.



write()

```
bytesWritten = write(fd, "Message", sizeof("Message"));
```

int write(int fd, char *buf, int n)

No. of bytes written

-1 on error

descriptor

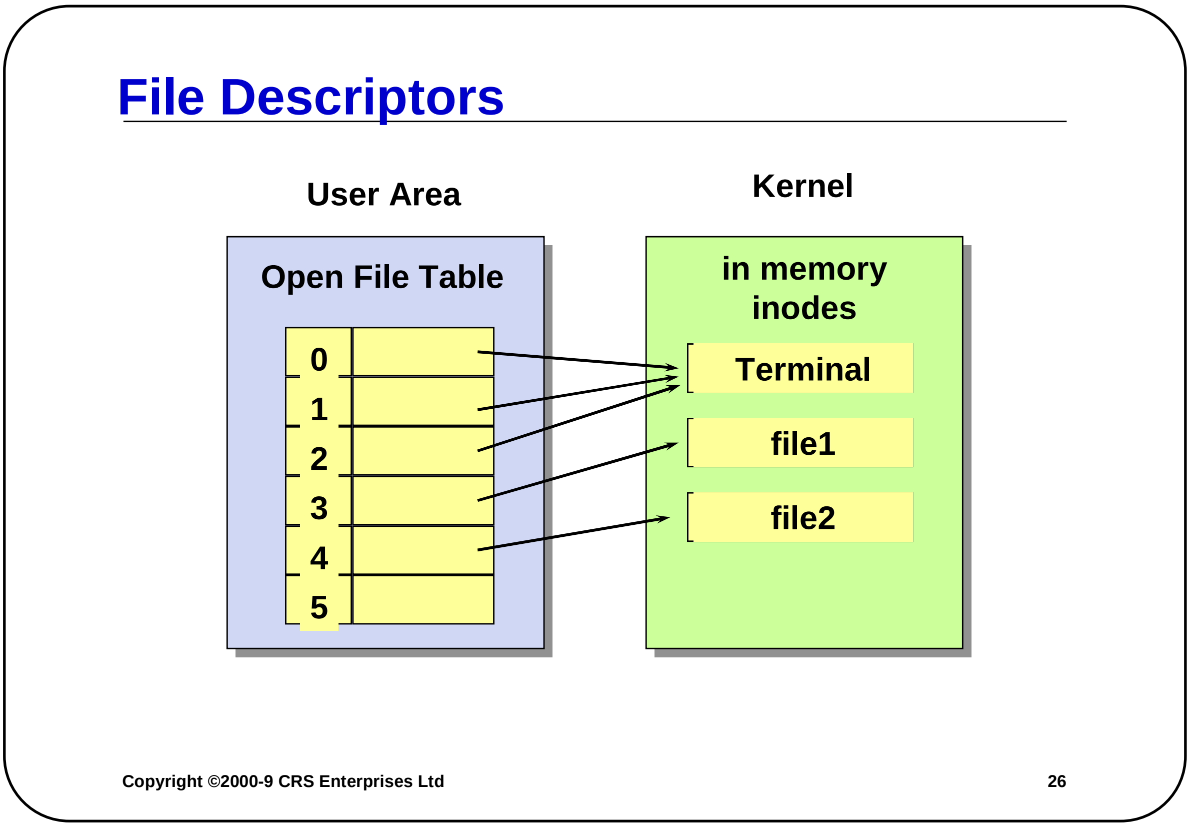
buffer holding bytes to write

no. of bytes to write

Copyright ©2000-9 CRS Enterprises Ltd 25

The write system call is used to write data to a file or device. The number of bytes written is returned from the call, unless an error occurred, in which case -1 is returned and errno is set to indicate the error.

When writing to seeking devices (files), the data is written at the current file position and the file position is incremented by the number of bytes written. If the O_APPEND flag was set on open, then the file pointer is always reset to the end of the file before the write takes place. Such writes are guaranteed to be atomic.



File Descriptors

Copyright ©2000-9 CRS Enterprises Ltd 26

Unix processes can identify files using file descriptors as well as the stream file type (FILE). File descriptors are non-negative integers that are used to index a file-control table local to each process. Standard file descriptors are used for a process's files which are open by default; these are:

0	Standard input.
1	Standard output.
2	Standard error output.

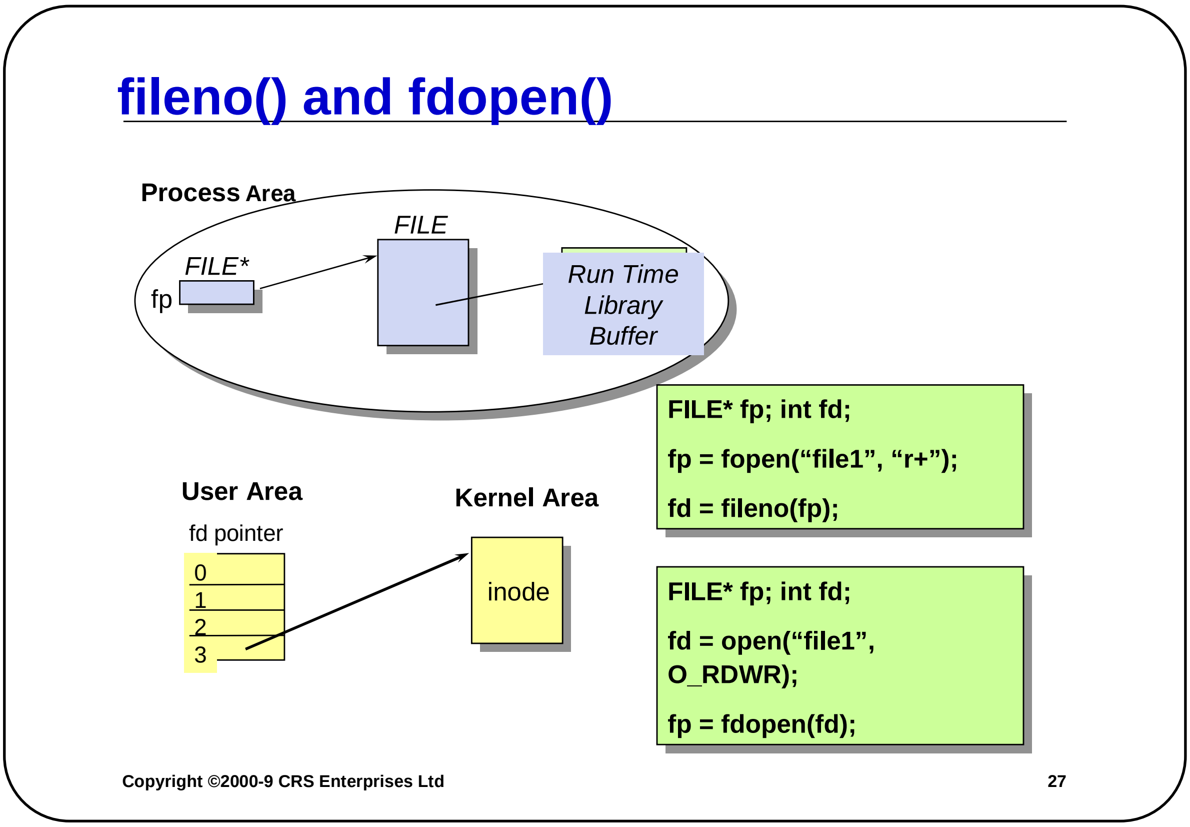
Note that the symbolic names stdin, stdout and stderr defined in the stdio.h header file refer to the FILE streams and not to the file descriptors.

The implementation of the stream file type is achieved using the lower-level file descriptors.

```
int fileno (FILE *stream)
```

will return the low-level file descriptor associated with a file stream. It is not advisable to intermix stream I/O and the low-level I/O system calls (described next) on the same open file.

Open file descriptors remain open across exec and fork system calls (described in a later chapter). This permits a child process to inherit a parent's open files (typically standard input, output and error).



fileno() and fdopen()

```
FILE* fp; int fd;
fp = fopen(“file1”, “r+”);
fd = fileno(fp);
```

```
FILE* fp; int fd;
fd = open(“file1”,
O_RDWR);
fp = fdopen(fd);
```

27

Unix processes can identify files using file descriptors as well as the stream file type (FILE). File descriptors are non-negative integers that are used to index a file-control table local to each process. Standard file descriptors are used for a process's files which are open by default; these are:

0	Standard input.
1	Standard output.
2	Standard error output.

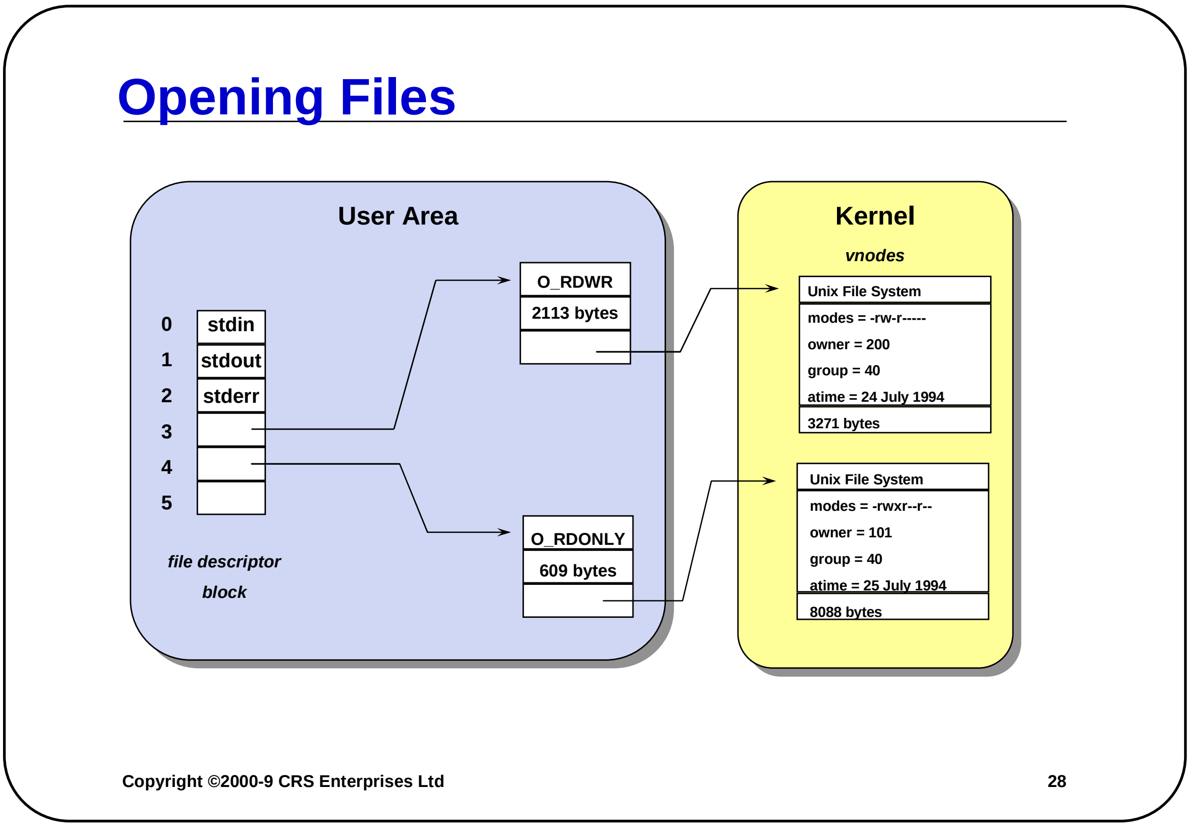
Note that the symbolic names stdin, stdout and stderr defined in the stdio.h header file refer to the FILE streams and not to the file descriptors.

The implementation of the stream file type is achieved using the lower-level file descriptors.

```
int fileno (FILE *stream)
```

will return the low-level file descriptor associated with a file stream. It is not advisable to intermix stream I/O and the low-level I/O system calls (described next) on the same open file.

Open file descriptors remain open across exec and fork system calls (described in a later chapter). This permits a child process to inherit a parent's open files (typically standard input, output and error).



Opening Files

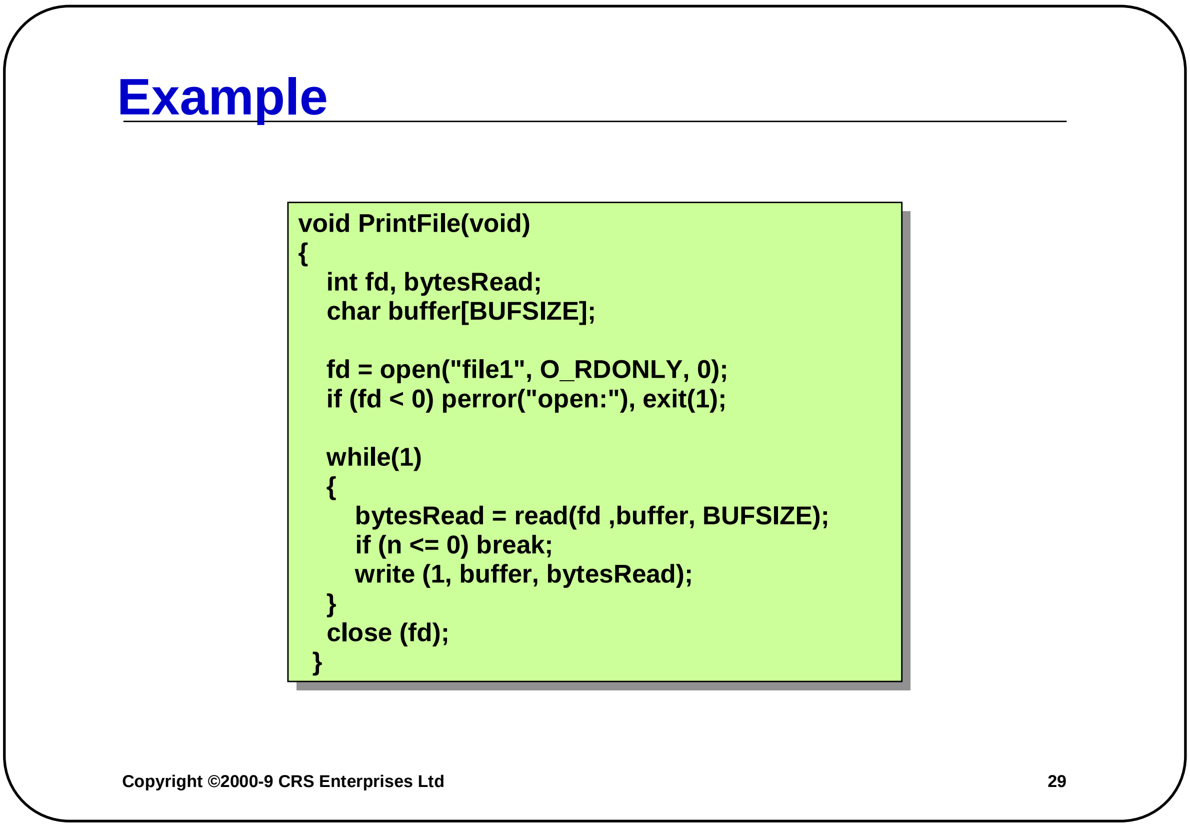
Copyright ©2000-9 CRS Enterprises Ltd 28

Each process has a file-descriptor table that contains information about every file the process has open and the flags associated with the file. If the process was run from a standard shell, it would almost certainly have the first three file descriptors allocated to *stdin*, *stdout* and *stderr*.

Each process also has an open file table. This table contains a pointer to the vnode structure, flags to indicate the mode in which the file was opened and the current offset in the file for read() and write() operations.

The kernel maintains a vnode table for every open file. This table is shared by all processes. The table indicates the file-system type (e.g. Unix, NFS, PC), a pointer to the inode of the file (Unix file systems only) and the size of the file.

It is important to realize that if several processes open the same file, then they share the vnode structure, but have their own open file tables. This enables different processes to maintain different file offsets, so that they can access different parts of a file simultaneously.

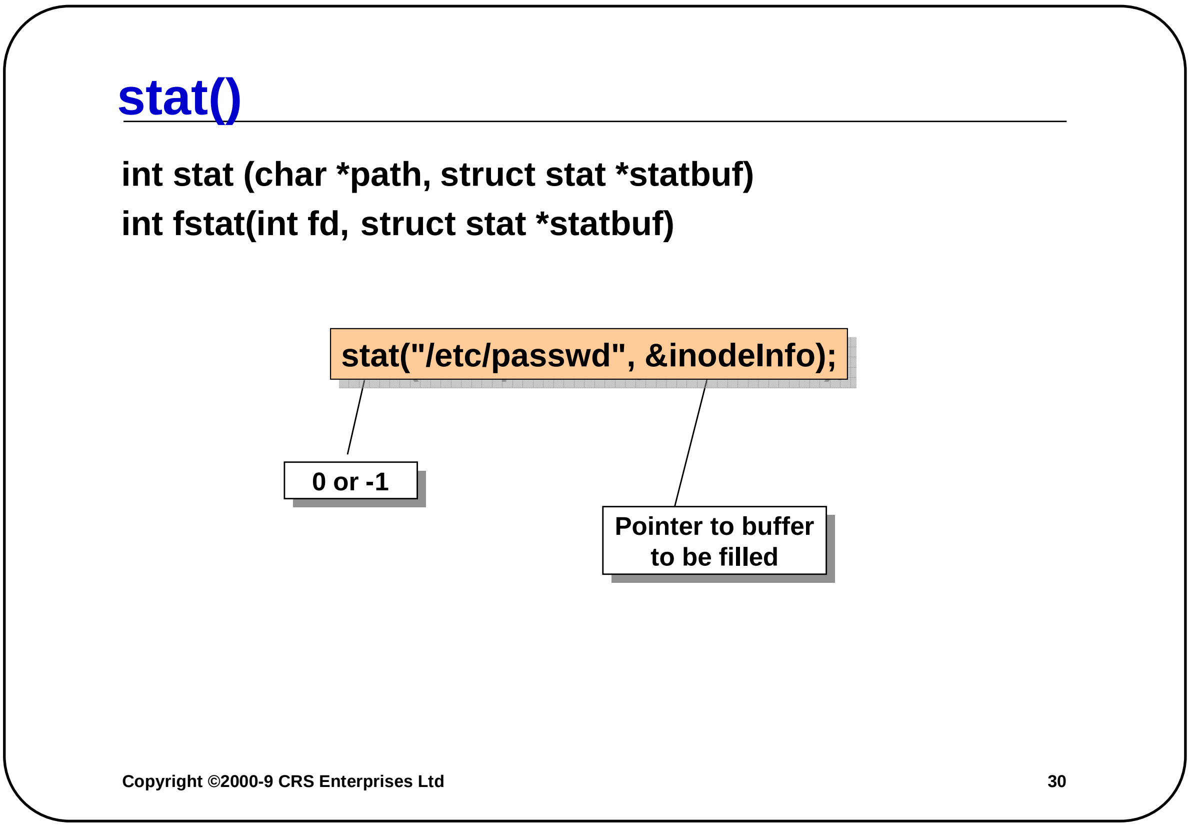


Example

```
void PrintFile(void)
{
    int fd, bytesRead;
    char buffer[BUFSIZE];

    fd = open("file1", O_RDONLY, 0);
    if (fd < 0) perror("open:"), exit(1);

    while(1)
    {
        bytesRead = read(fd ,buffer, BUFSIZE);
        if (n <= 0) break;
        write (1, buffer, bytesRead);
    }
    close (fd);
  }
```

29

This is a simple function to print a file to standard output. It makes no allowances for read errors. This function copies the data verbatim, with no breaking up of long lines.



stat()

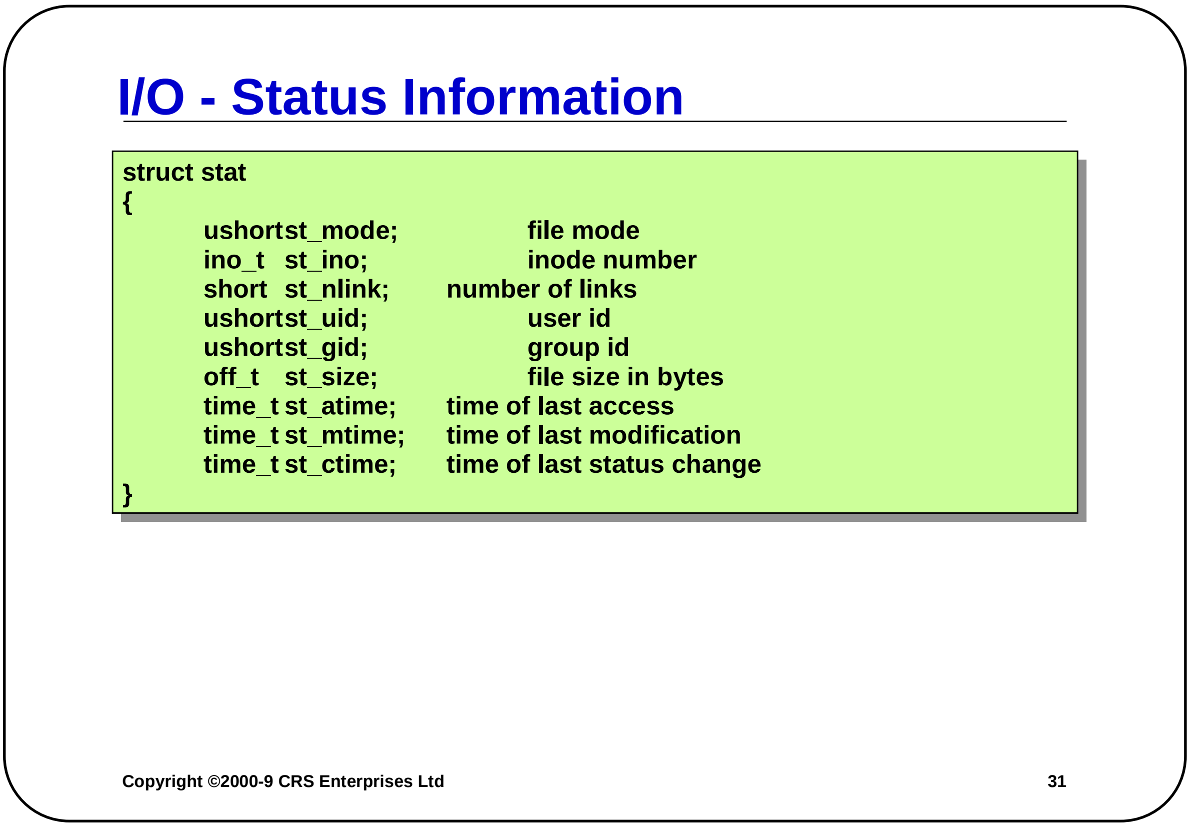
```
int stat (char *path, struct stat *statbuf)
int fstat(int fd, struct stat *statbuf)
```

```
stat("/etc/passwd", &inodeInfo);
```

0 or -1
Pointer to buffer to be filled

30

The stat and fstat system calls return (in the stat record) details of the appropriate file. This information is extracted from the inode for the file. fstat uses an open file descriptor, whereas stat uses a full filename (the file may or may not be open by this process). Stat and fstat return zero on success, -1 on error.



I/O - Status Information

```
struct stat
{
          ushortst_mode;              file mode
          ino_t   st_ino;             inode number
          short   st_nlink;       number of links
          ushortst_uid;               user id
          ushortst_gid;               group id
          off_t   st_size;            file size in bytes
          time_t st_atime;        time of last access
          time_t st_mtime;        time of last modification
          time_t st_ctime;        time of last status change
}
```

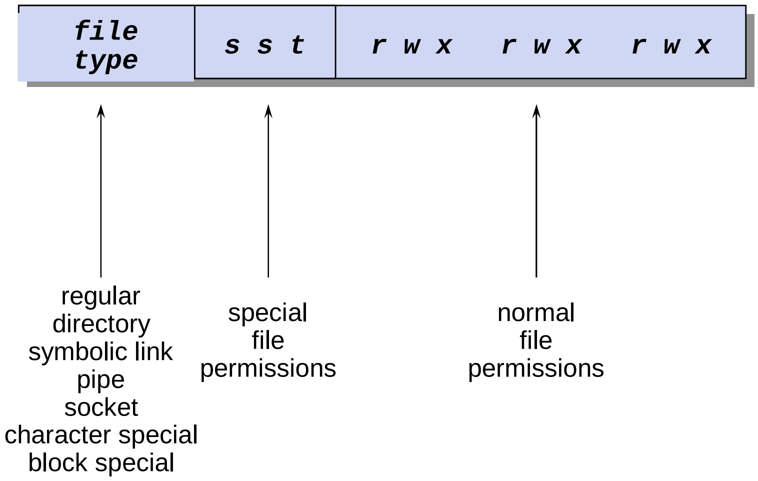
The stat data structure contains fields derived from the inode information. Some of the more obvious and interesting fields are shown on the slide. The st_mode field defines the status and protection flags for the file. Most of the other fields should be self explanatory.

The time fields define time in terms of seconds measured from 00:00 GMT, Jan 1, 1970. This is the standard Unix time-stamping mechanism (see the header file <time.h> for routines such as time, localtime, asctime, etc.).

The special data types are defined in the header file

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

st_mode



file type | s s t | r w x r w x r w x

regular
directory
symbolic link
pipe
socket
character special
block special

special
file
permissions

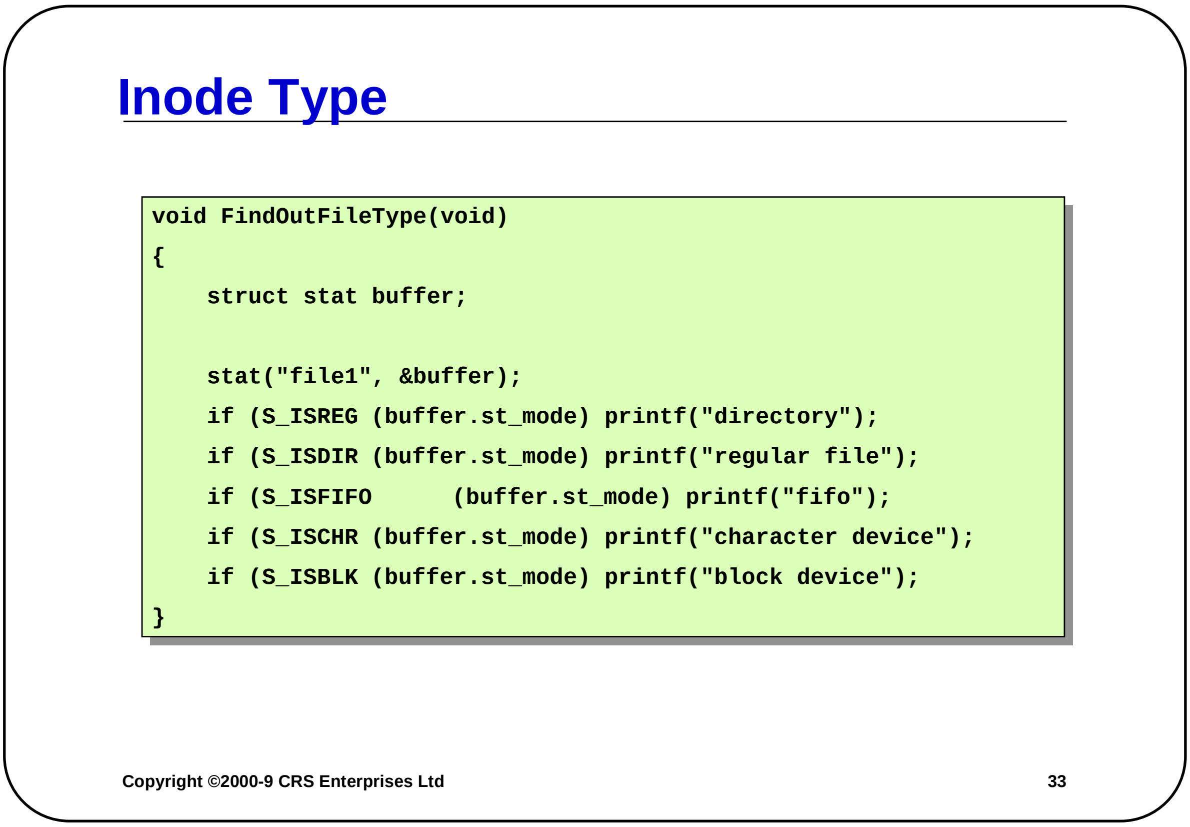
normal
file
permissions

Probably the most important field in the stat structure is the st_mode field. This has type mode_t, usually an unsigned short, with each bit representing one characteristic of the object.

The mode field indicates two main features of the object:

> The type of object, i.e. file, directory or one of the special file types such as device, pipe or socket.
>
> The access permissions for the object, including any setuid or setgid information.

The mode field is most easily analyzed with a collection of defined constants and macros from the include file <sys/stat.h>. The next slides show these in more detail.

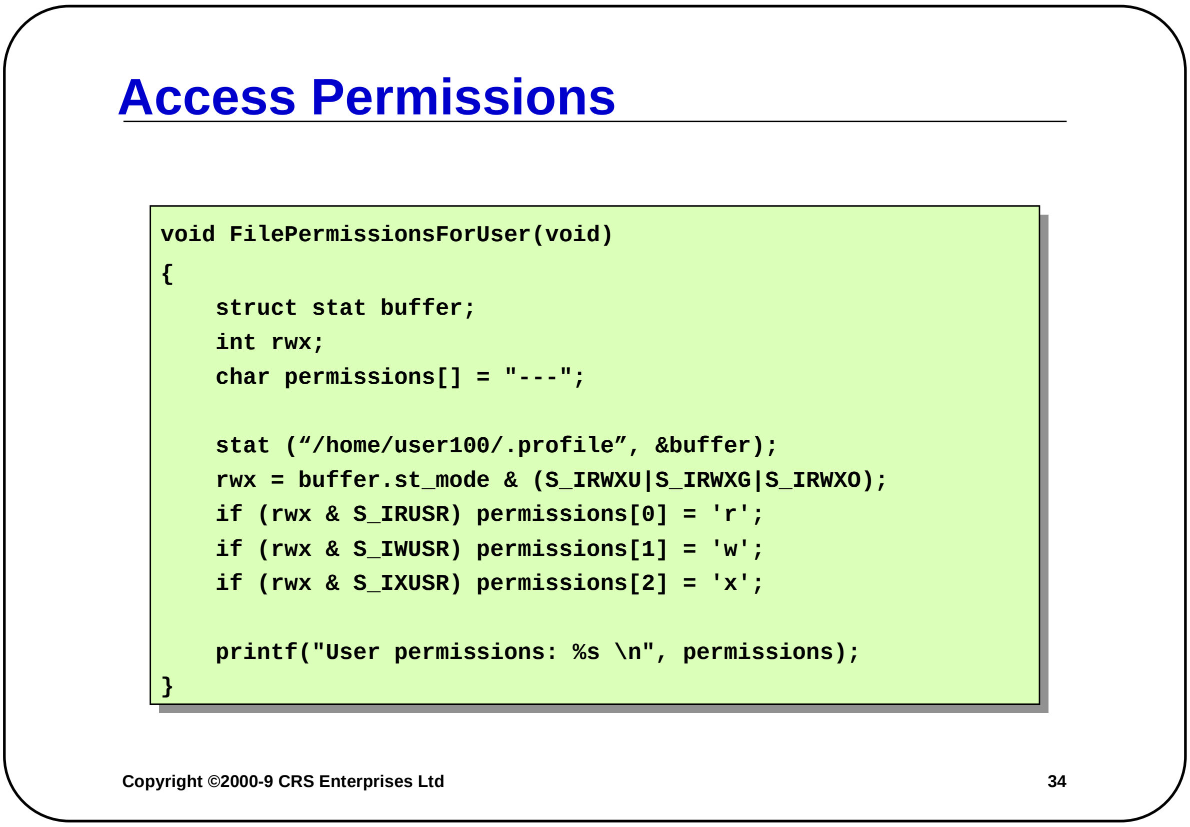


Inode Type

```
void FindOutFileType(void)
{
    struct stat buffer;

    stat("file1", &buffer);
    if (S_ISREG (buffer.st_mode) printf("directory");
    if (S_ISDIR (buffer.st_mode) printf("regular file");
    if (S_ISFIFO      (buffer.st_mode) printf("fifo");
    if (S_ISCHR (buffer.st_mode) printf("character device");
    if (S_ISBLK (buffer.st_mode) printf("block device");
}
```

An inode's object type is determined by examining its mode field, using facilities provided in the include file <sys/stat.h>:

S_ISREG(mode)	evaluates to TRUE (>0) if mode describes a file
S_ISDIR(mode)	is TRUE for a directory
S_ISCHR(mode)	is TRUE for a character-device special file
S_ISBLK(mode)	is TRUE for a block-device special file
S_ISFIFO(mode)	is TRUE for a named pipe
S_ISLNK(mode)	is TRUE for a symbolic link
S_ISSOCK(mode)	is TRUE for a socket



Access Permissions

```
void FilePermissionsForUser(void)
{
    struct stat buffer;
    int rwx;
    char permissions[] = "---";

    stat (“/home/user100/.profile”, &buffer);
    rwx = buffer.st_mode & (S_IRWXU|S_IRWXG|S_IRWXO);
    if (rwx & S_IRUSR) permissions[0] = 'r';
    if (rwx & S_IWUSR) permissions[1] = 'w';
    if (rwx & S_IXUSR) permissions[2] = 'x';

    printf("User permissions: %s \n", permissions);
}
```

34

The access permissions for the object are obtained using a set of defined constants. They can be used to test individual permissions or to obtain sets of permissions.

(mode & S_IRWXU)	generates the rwx permissions for the owner
(mode & S_IRWXG)	generates the permissions for the object's group
(mode & S_IRWXO)	generates the permissions for other users
(mode & S_IRUSR)	is TRUE if the owner has read permission
(mode & S_IWUSR)	is TRUE if the owner has write permission
(mode & S_IXUSR)	is TRUE if the owner has execute permission
(mode & S_IRGRP)	is TRUE if the group has read permission
(mode & S_IWGRP)	is TRUE if the group has write permission
(mode & S_IXGRP)	is TRUE if the group has execute permission
(mode & S_IROTH)	is TRUE if other users have read permission
(mode & S_IWOTH)	is TRUE if other users have write permission
(mode & S_IXOTH)	is TRUE if other users have execute permission

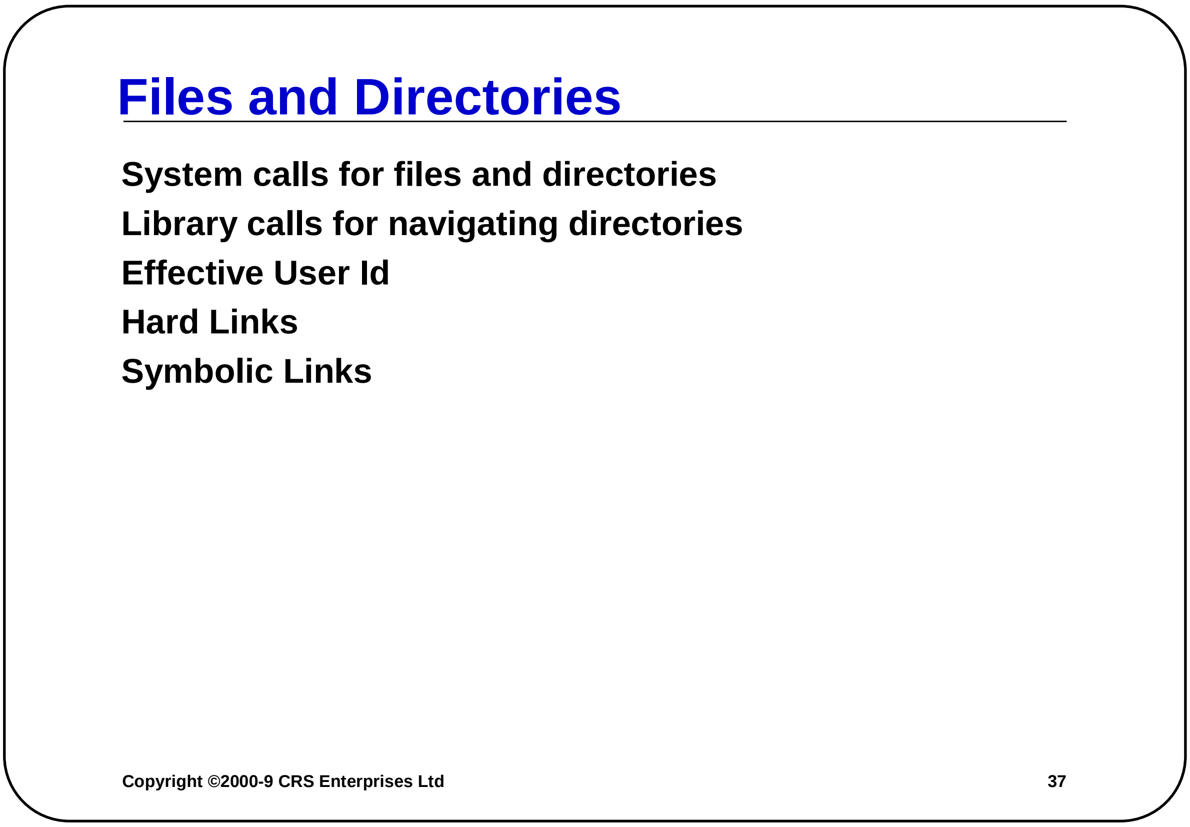
Some extra features can also be tested, i.e.:

(mode & S_ISUID)	is TRUE if the set-uid bit is set on the inode
(mode & S_ISGID)	is TRUE if the set-gid bit is set
(mode & S_IVTX)	is TRUE if the sticky bit is set

3

Files and Directories

3



Files and Directories

System calls for files and directories

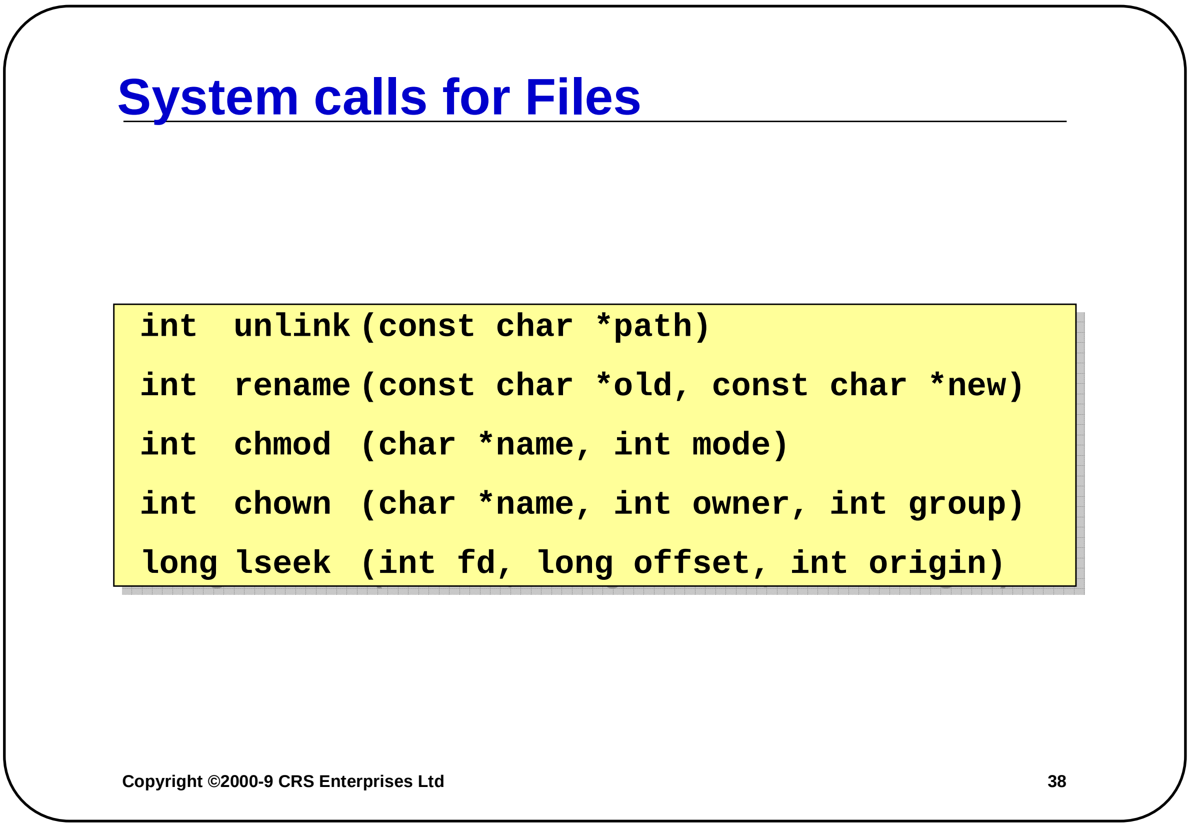
Library calls for navigating directories

Effective User Id

Hard Links

Symbolic Links

This chapter describes system calls used to manage files and directories. Routines to create files and devices are described, as are the general routines for renaming and removing files and changing access permissions. The system independent run time library routines for navigating directories is presented. The role of effective and real user ids is also discussed



System calls for Files

```
int  unlink (const char *path)
int  rename (const char *old, const char *new)
int  chmod  (char *name, int mode)
int  chown  (char *name, int owner, int group)
long lseek  (int fd, long offset, int origin)
```

The unlink() system call is used to remove a file's entry from a directory. The link reference count in the file's inode is decremented, and if this reaches zero, the file is deleted.

The rename() system call renames the specified file, which can be any form of file including directories and special files.

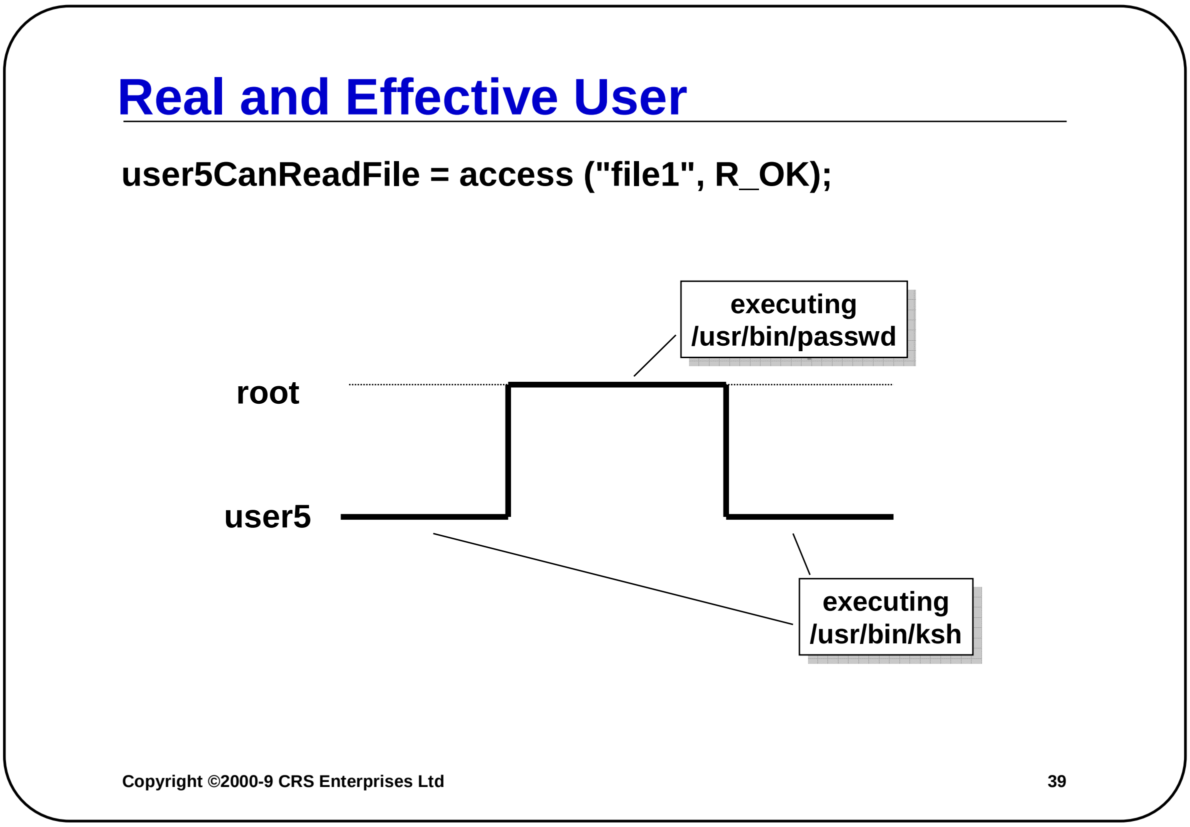
The chmod() system call changes the permission flags for the specified file. The set user id and set group id bits and sticky bit can be modified in addition to the access-control bits. Only the superuser and owner of a file can change the access flags.

The chown() system call changes the named file's owner and group to the specified ids. Only the superuser or file owner may change the ownership or group ids.

The lseek() system call sets the file position for the next read or write on the file represented by descriptor fd.

Example calls:

```
unlink ("file1");
rename ("OldFileName", "NewFileName");
chmod ("file1", 0600);
chown ("file1", 100, 50);
lseek (fd, 250000L, SEEK_SET);
```



Real and Effective User

user5CanReadFile = access ("file1", R_OK);

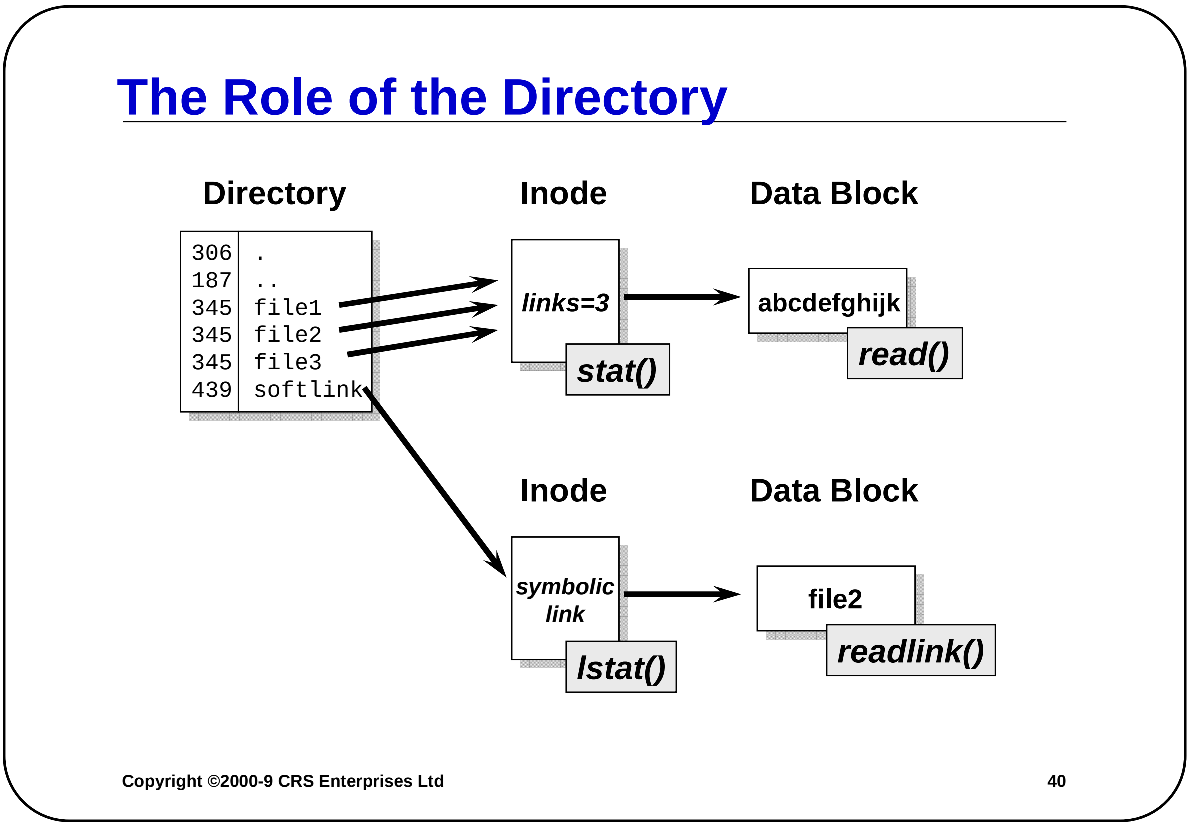
The access() system call is the only system call that checks file access against real user id. All other system calls respect the effective user id. A process's effective user id is the same as its real user id unless the executable file used to load the process into memory has its set user bit set.

In the example shown above, user5 executes the ksh, passwd and ksh executables in order. When the passwd executable is executed, because it is owned by root and has its set user bit is set, user5's effective user id is set to root. The ksh programs do not have set user bit set and therefore run with effective user id set to user5. While passwd is executing as root, it has permissions to modify any given file. However, the access() call can be used to check if user5 would normally have permission to modify the given file; the stat() call cannot be used because it respects effective user id.

The access system call returns zero if the requested access to the file is permitted. Requested file access is determined by the lowest three bits of the mode word, which correspond to the read (040), write (020) and execute (010) permission bits.

These can be represented mnemonically using the defined constants:

R_OK	The file can be read by the current process.
W_OK	The file can be written by the current process.
X_OK	The file can be executed by the current process.
F_OK	All intermediate directories in the path can be read and the file exists.



The Role of the Directory

```
306  .
187  ..
345  file1
345  file2
345  file3
439  softlink
```

40

The main job of a directory is to associate a name with an inode. A directory is a special type of file, that contains a table, mapping names onto inodes. When you access a file by name, Unix looks up the directory containing the file and finds out the inode number. It can then retrieve the inode from the "inode area" of the disk, and find out all it needs to know about the file (or directory).

A directory contains two types of links to inodes: hard links and symbolic links.

In this example we create a file called file1 (a hard link) and then create two further hard links using the Unix commands

```
print "ABCDEFGHIJKLM" > file1
ln file1 file2
ln file1 file3
```

As you can see from the diagram, file1, file2 and file3 all share the same inode. In reality, there is one file with 3 names.

We can create a symbolic link (soft link) using the Unix command

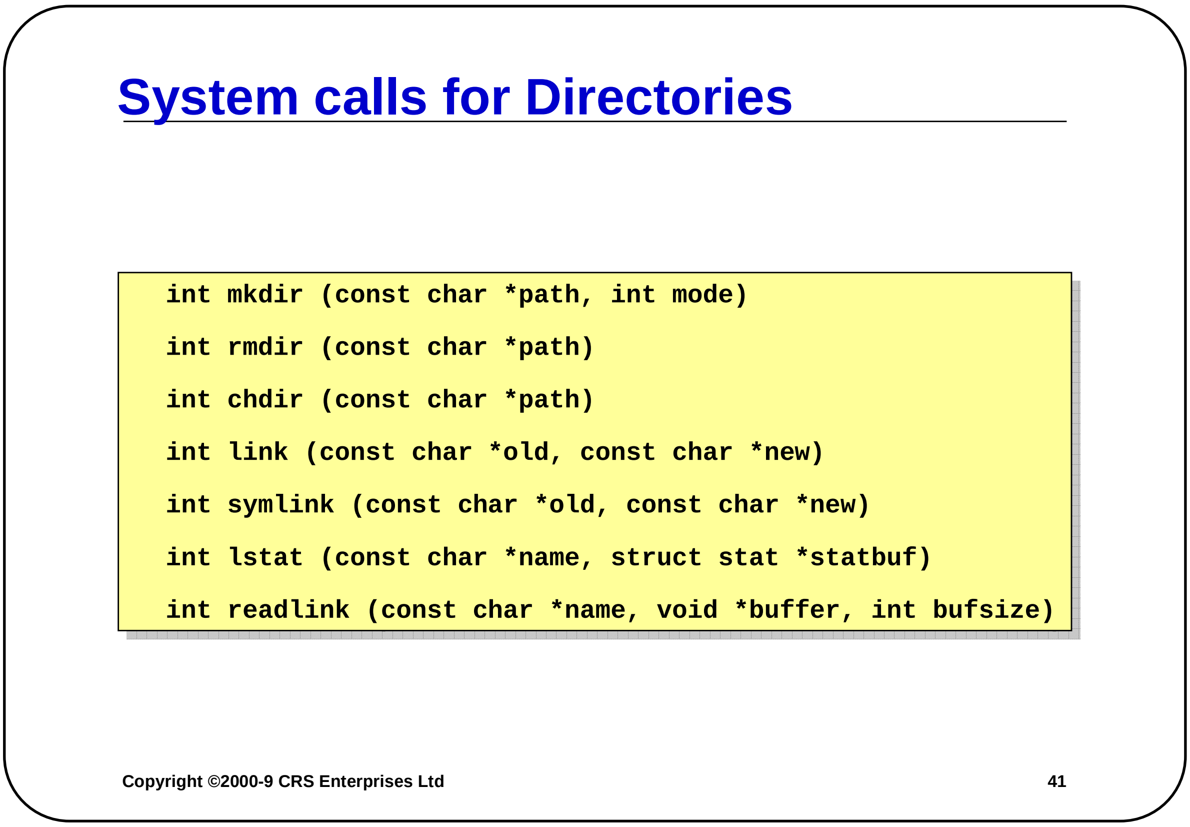
```
ln -s file2 softlink
```

Note that the symbolic link uses a different inode from before. Nevertheless, the commands

```
cat file1
cat file2
cat file3
cat softlink
```

all print the same information.

The system calls for extracting information from hard links are stat() and read()/write(). Symbolic links have their own system calls: lstat() and readlink().



System calls for Directories

```
int mkdir (const char *path, int mode)
int rmdir (const char *path)
int chdir (const char *path)
int link (const char *old, const char *new)
int symlink (const char *old, const char *new)
int lstat (const char *name, struct stat *statbuf)
int readlink (const char *name, void *buffer, int bufsize)
```

Directories are created with the mkdir() system call. When a directory is created, the entries for "." and ".." created at the same time. The link reference count for a new directory will be set to two; one for its primary name and one for the "." link created within the directory. The parent's directory link count will also be incremented by one for the new directory entry.

Empty directories are removed with rmdir(). The entries for "." and ".." are unlinked before the directory itself is deleted.

The chdir() system call sets the current working directory to the specified pathname. The directory must exist.

The link() system call creates additional filenames for an inode. A file's inode keeps a reference count of the number of links or names it is known by. A file is only deleted, and its data blocks reclaimed, when its reference count falls to zero and no process has the file open.

A symbolic link is created with the symlink() system call. System calls such as open() and stat(), when used on a symbolic link, automatically traverse the link. Symbolic links may point across file systems (partitions).

The system call lstat() is provided to extract the information stored in the symbolic link's inode.

The readlink() system call is provided to give access to the name string stored in the symbolic link.

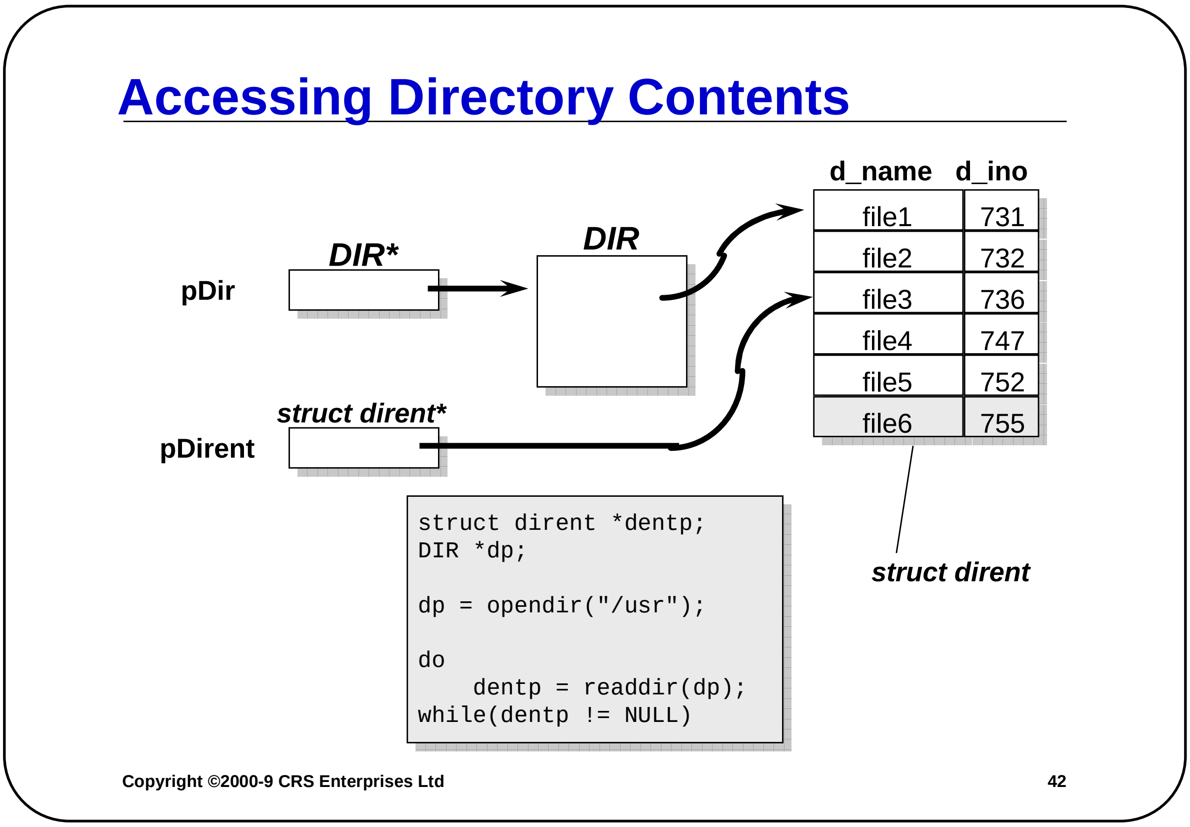
Example calls:

int mkdir (const char *path, int mode)

int rmdir (const char *path)

int chdir (const char *path)

int link (const char *old, const char *new)



Accessing Directory Contents

d_name	d_ino
file1	731
file2	732
file3	736
file4	747
file5	752
file6	755

```
struct dirent *dentp;
DIR *dp;

dp = opendir("/usr");

do
    dentp = readdir(dp);
while(dentp != NULL)
```

The directory include file dirent.h defines a number of library routines which simplify the interface to directories. The routines described on the next slide provide a common file-system-independent interface to directories. Their use is recommended in preference to opening and reading directories using the simple I/O operations open(), read(), lseek() and close().

The directory structure struct dirent is defined according to the underlying file system type. Two of the most common directory layouts are shown below:

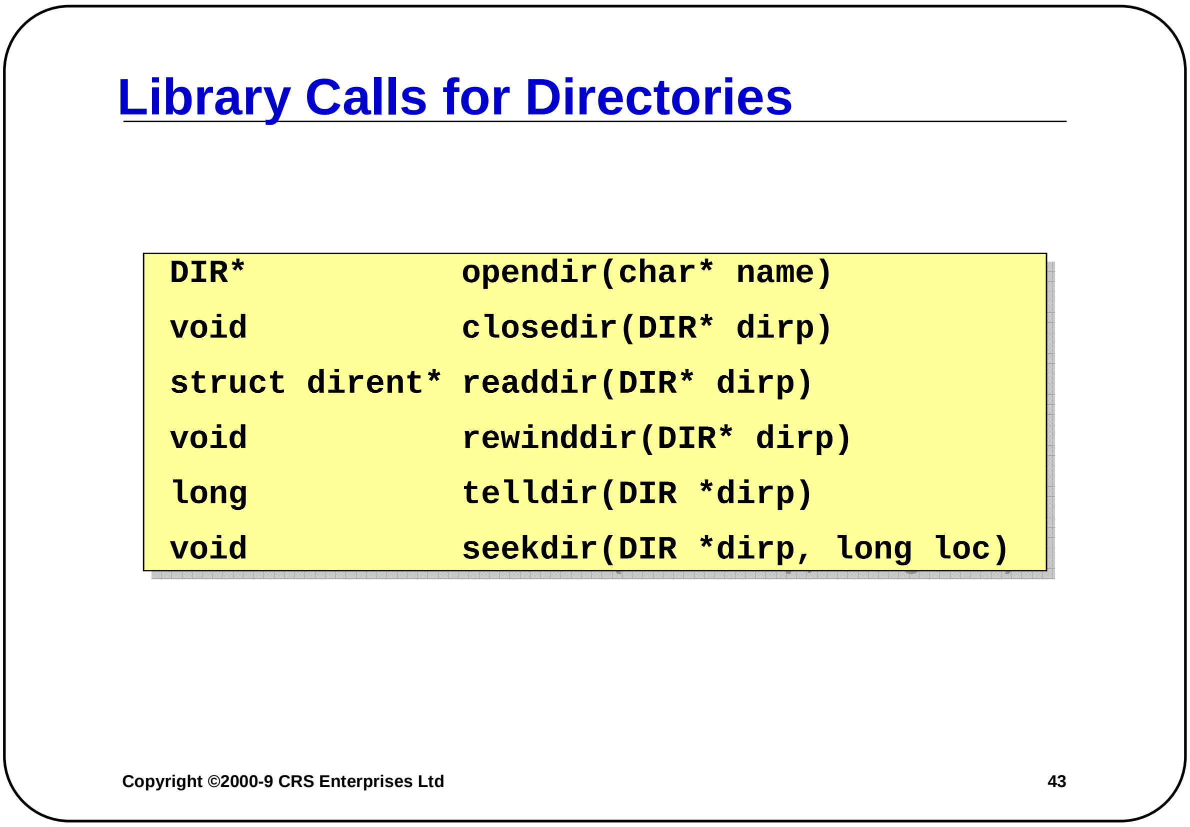
Fixed Length Filenames

```
struct dirent
{
        ino_t       d_ino;
        char        d_name[14];
}
```

Variable Length Filenames

```
struct dirent
{
        ino_t       d_ino;
        u_short d   namlen;
        char        d_name[];
}
```

Programs written using the directory routines and the common data fields d_ino and d_name will be portable.



Library Calls for Directories

```
DIR*            opendir(char* name)
void            closedir(DIR* dirp)
struct dirent*  readdir(DIR* dirp)
void            rewinddir(DIR* dirp)
long            telldir(DIR *dirp)
void            seekdir(DIR *dirp, long loc)
```

The opendir() call opens a directory for reading and returns a pointer to a directory stream control structure. This structure is allocated by the run time library code.

The closedir() call closes a directory and frees the memory allocated for the directory stream structure.

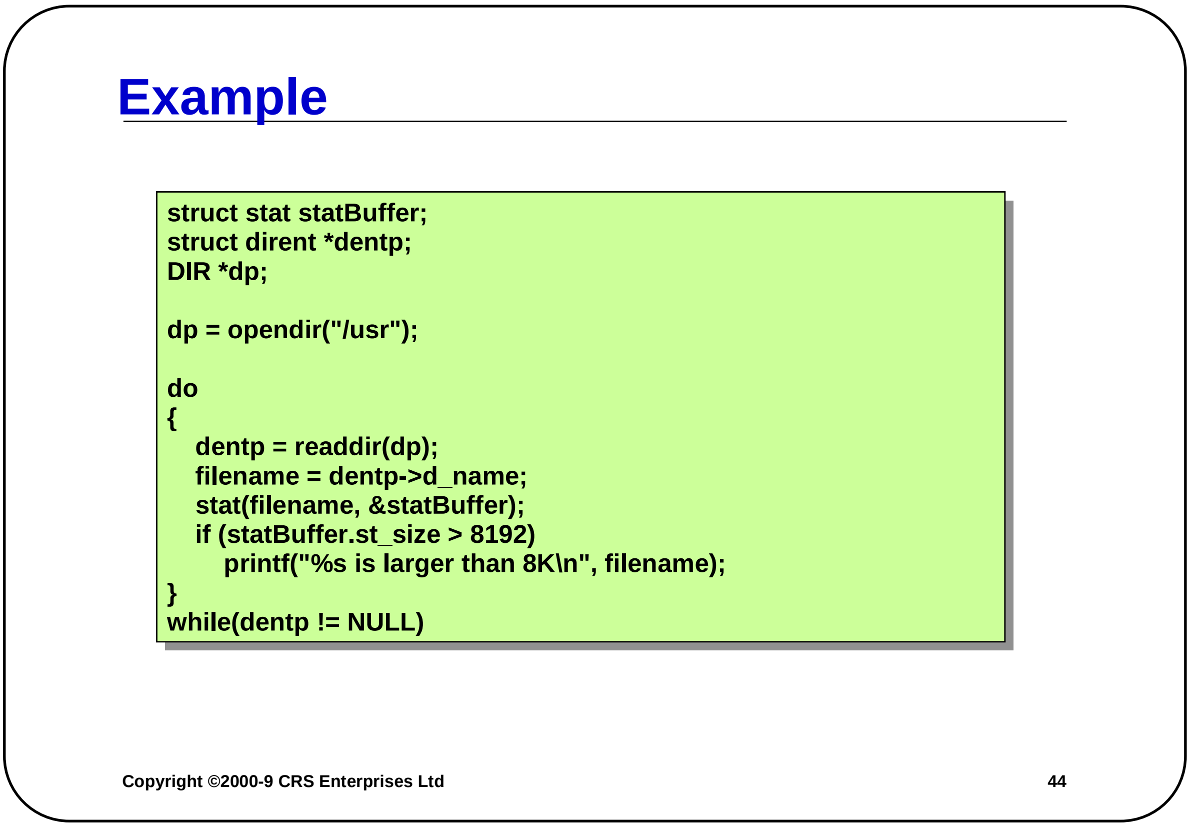
The readdir() call reads the next valid entry from the directory and returns a pointer to a dirent structure allocated within the DIR stream data. At the end of the directory, readdir() returns NULL. There is only one dirent structure in the DIR data, so subsequent calls to the readdir() library routine overwrite the data in this record.

The rewinddir() call resets a directory stream so that a directory's contents can be rescanned.

The telldir() call remembers the current location within a directory. The seekdir() call is used to position to a previously-remembered location.

Example calls:

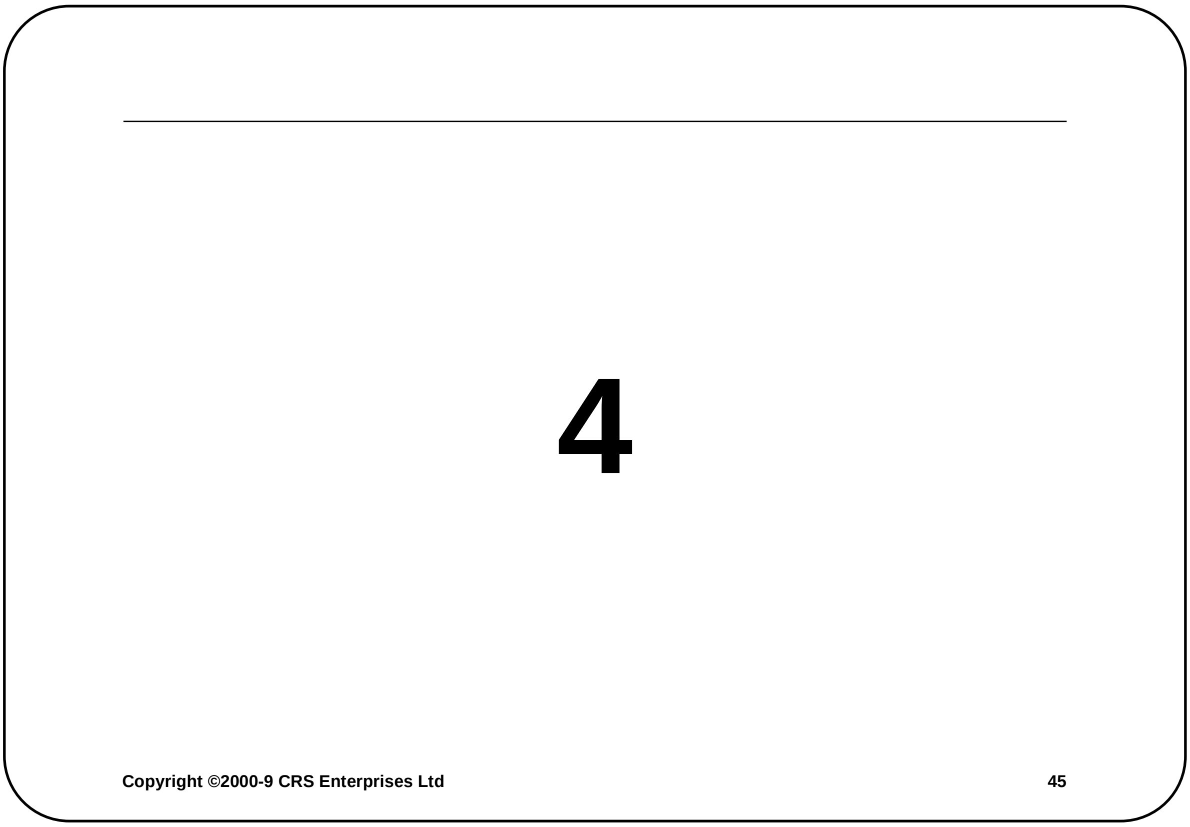
```
dirPtr = opendir("/tmp/dir1");
closedir (dirPtr)
direntPtr = readdir(dirPtr);
rewinddir(dirPtr)
offset = telldir(dirPtr)
seekdir(dirPtr, offset)
```



Example

```
struct stat statBuffer;
struct dirent *dentp;
DIR *dp;

dp = opendir("/usr");

do
{
    dentp = readdir(dp);
    filename = dentp->d_name;
    stat(filename, &statBuffer);
    if (statBuffer.st_size > 8192)
        printf("%s is larger than 8K\n", filename);
}
while(dentp != NULL)
```

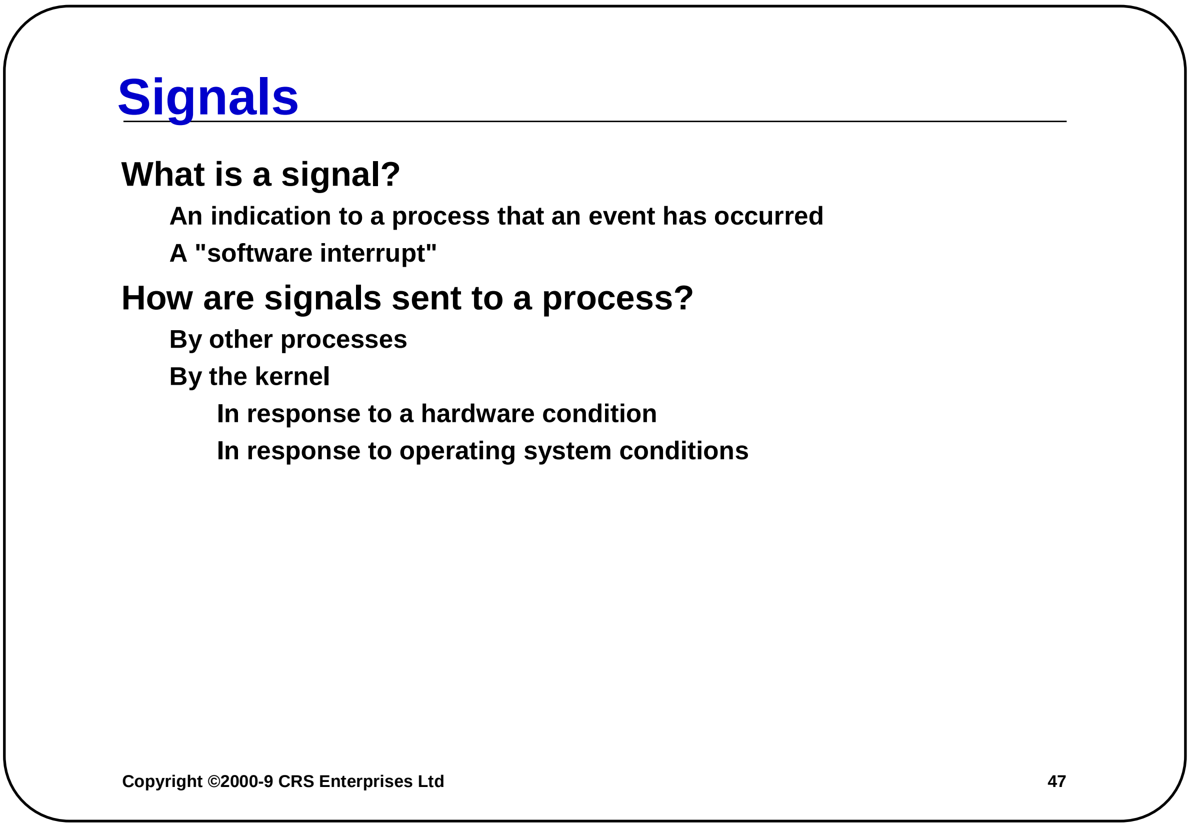
In this example we navigate through all entries in the directory /usr. As each filename is determined we use the stat() system call to check if the file is larger than 8K in size. The names of all such files are printed on standard output.



4

Signals

4



Signals

What is a signal?

- An indication to a process that an event has occurred
- A "software interrupt"

How are signals sent to a process?

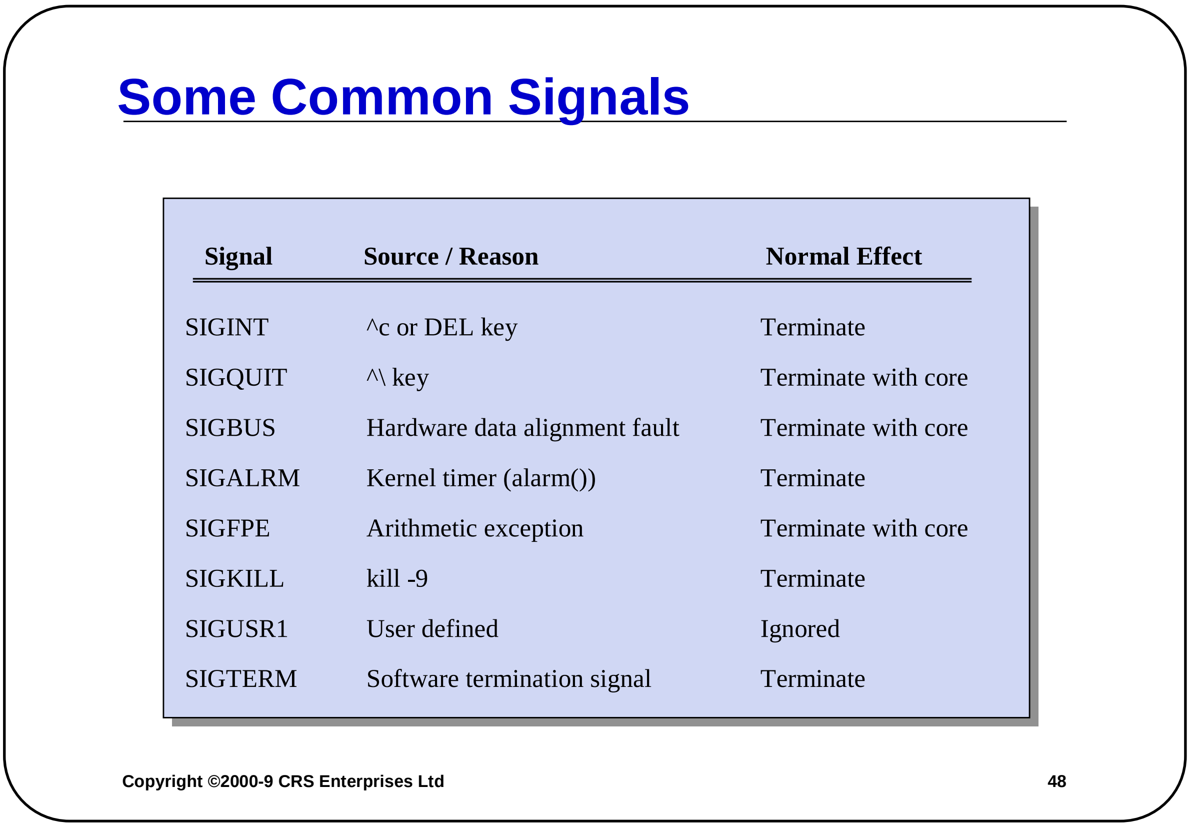
- By other processes
- By the kernel
  - In response to a hardware condition
  - In response to operating system conditions

Signals are used to inform processes when asynchronous events occur, i.e. a process does not have to explicitly wait for the event or look to see if the event has occurred. This has led to signals often being known as "software interrupts".

Signals can be sent to a process by another process using the kill() system call. They may also be sent to a process by the kernel in response to some condition which has arisen, either in the kernel itself or in the underlying hardware. For example, if a process attempts a divide-by-zero operation, the hardware in the arithmetic or floating-point unit will normally generate some form of exception. This is detected by the kernel, which arranges for the process to be sent a signal notifying it of an arithmetic exception.

Certain keys on the keyboard also cause a process to be sent a signal. For example, hitting the DEL or Ctrl-C key causes the current process to be sent an "interrupt" signal, which normally causes the process to terminate.

Modern Unix implementations define upwards of 30 different signals, the majority of which are common across all versions. The set of signals available is part of the virtual machine definition, which allows Unix to present a consistent interface to programmers across a wide range of hardware platforms. It is not necessary to be aware of the different hardware interrupt conditions, as they will be dealt with in the kernel, which translates them into the appropriate signals.

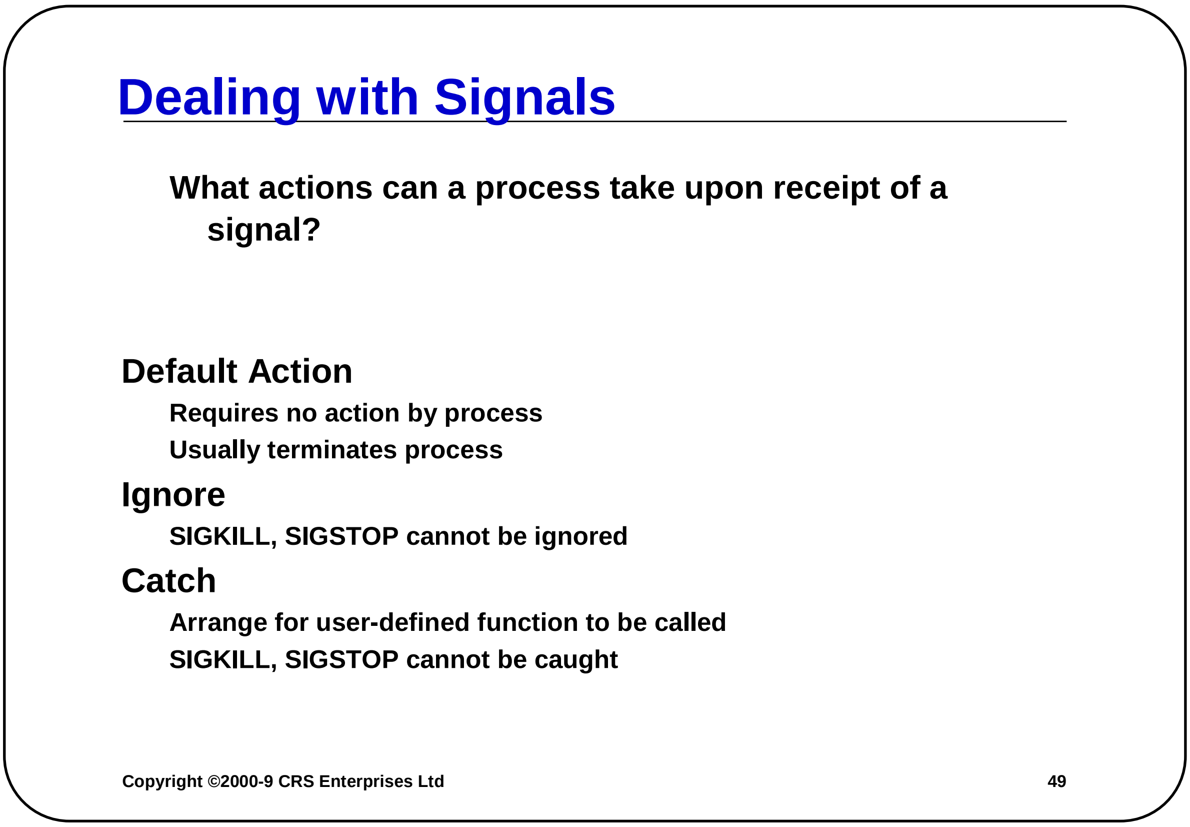


Some Common Signals

Signal	Source / Reason	Normal Effect
SIGINT	^c or DEL key	Terminate
SIGQUIT	^\ key	Terminate with core
SIGBUS	Hardware data alignment fault	Terminate with core
SIGALRM	Kernel timer (alarm())	Terminate
SIGFPE	Arithmetic exception	Terminate with core
SIGKILL	kill -9	Terminate
SIGUSR1	User defined	Ignored
SIGTERM	Software termination signal	Terminate

The slide illustrates some of the more common signals, together with the reasons why they are sent to a process and the normal effect on the process. If you want a signal to produce a different effect from the normal effect, you must write a signal handler function.

Signals are represented by small integer numbers. However, for ease of use, there are defined constants that can be used equivalently. These values are defined in <signal.h>. You cannot create your own signals if you have a POSIX.1 system You are allowed to define additional signals if you are using the POSIX.4 real time extensions.



Dealing with Signals

What actions can a process take upon receipt of a signal?

Default Action

- Requires no action by process
- Usually terminates process

Ignore

- SIGKILL, SIGSTOP cannot be ignored

Catch

- Arrange for user-defined function to be called
- SIGKILL, SIGSTOP cannot be caught

When a process receives a signal, there are three courses of action open to it:

Default

Every signal has a "default action" associated with it. If no other action is taken by a process, then the default action for the signal is taken by a process on receipt of that signal. In most cases, the default action is to terminate the process.

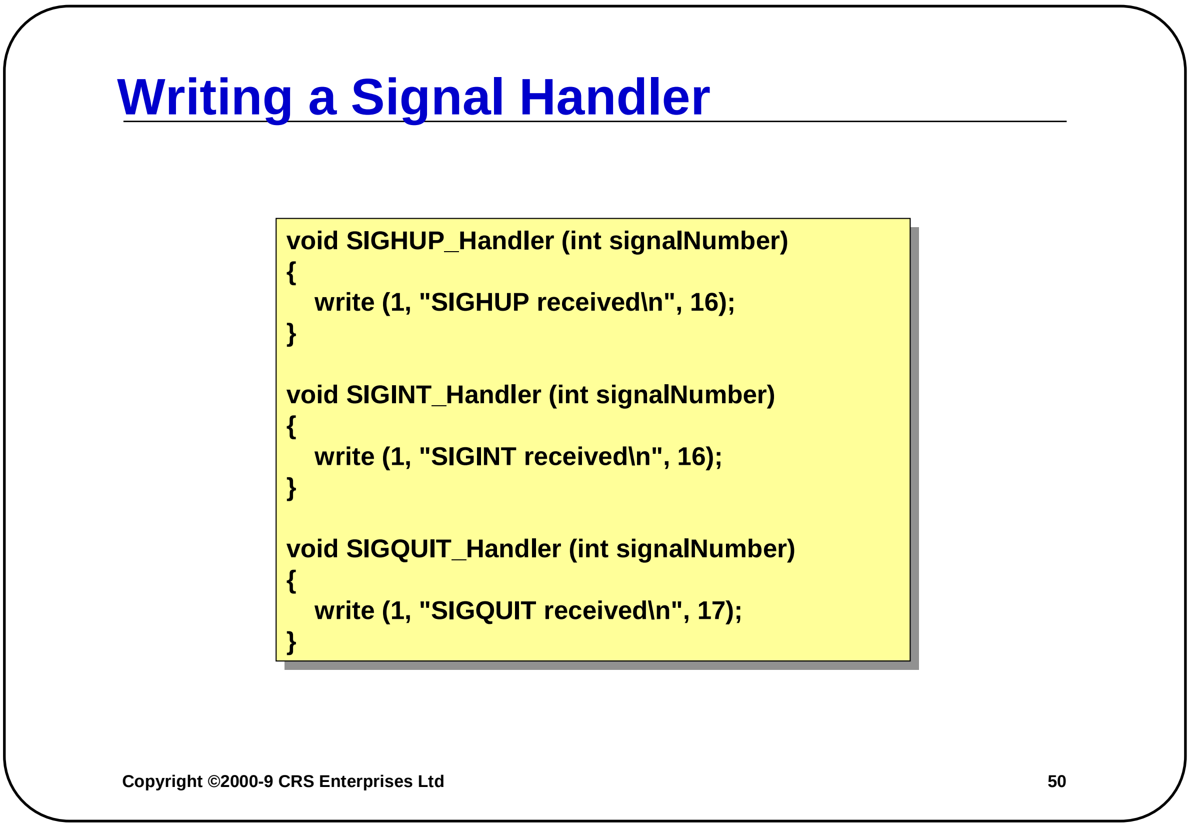
Many signals, particularly those associated with exceptional conditions, also cause a core image to be generated. The core image may then be analyzed with debugging tools to see what caused the exception.

Ignore

A process can choose to ignore a signal. In this case, the process will not be informed when it has been sent the signal. It is possible to ignore all signals, except SIGKILL and SIGSTOP.

Catch

A process may "catch" a signal and execute a specific routine when it receives the signal. The function to be executed is known as the signal handler. It is possible to catch all signals, except SIGKILL and SIGSTOP.



Writing a Signal Handler

```
void SIGHUP_Handler (int signalNumber)
{
   write (1, "SIGHUP received\n", 16);
}

void SIGINT_Handler (int signalNumber)
{
   write (1, "SIGINT received\n", 16);
}

void SIGQUIT_Handler (int signalNumber)
{
   write (1, "SIGQUIT received\n", 17);
}
```

50

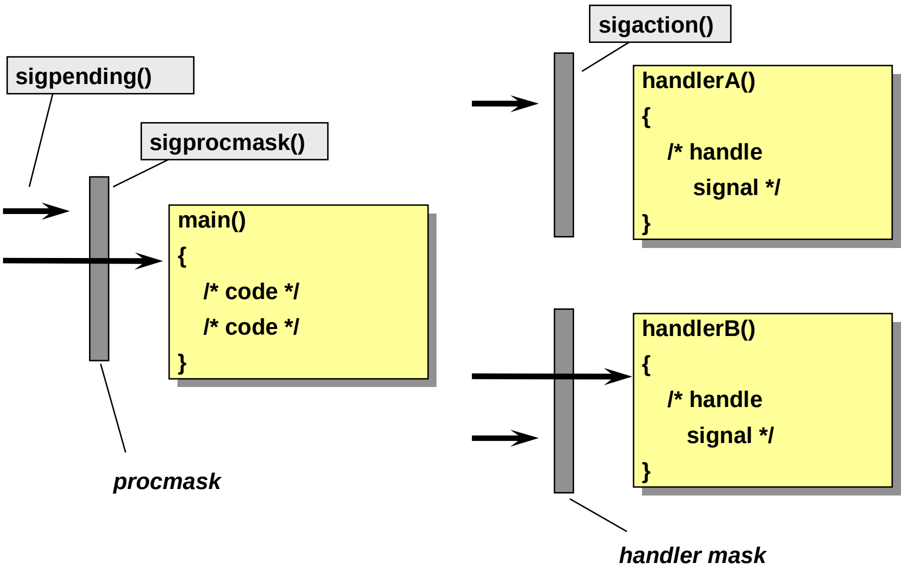
The slide shows sample signal-handler routines for SIGHUP, SIGINT and SIGQUIT. Each handler uses the write() system call to print a simple message. Do not use printf() inside a signal handler unless you are using a thread safe C Run Time Library; the standard implementation of printf it is not reentrant.

When a signal-handler routine is called, it is always passed the signal number of the signal that occurred. This allows the same routine to be installed for multiple signals and allows the routine to know which signal caused it to be called.

Some of the signals that are caused through hardware conditions also provide further information. For example, the SIGFPE signal may have been caused by divide by zero, overflow or underflow. A signal handler for SIGFPE will normally be able to access this information and work out which actual condition caused the signal. However, this facility is implementation dependent and will vary according to the system (hardware and version of Unix) being used.

When a handler routine returns, execution of the process will continue from the instruction where it was interrupted.

Unix Signal Model



sigpending()

sigprocmask()

```
main()
{
   /* code */
   /* code */
}
```

procmask

sigaction()

```
handlerA()
{
   /* handle
      signal */
}
```

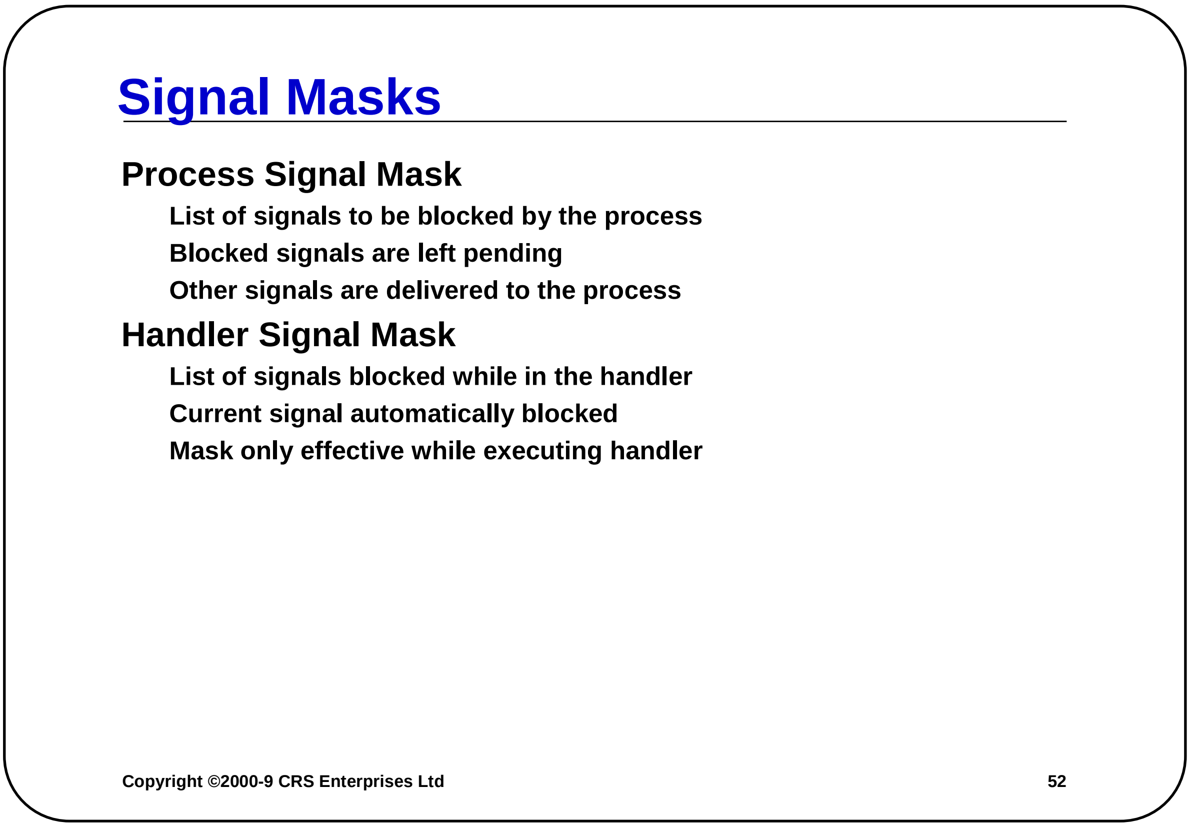
```
handlerB()
{
   /* handle
      signal */
}
```

handler mask

The Unix signal model is quite complicated. The process's code consists of a main thread plus several signal handlers. The main thread is executed until a signal is delivered. As soon as a signal arrives, execution of the main thread is suspended and the appropriate handler thread executed.

Each signal handler is protected from delivery of unwanted signals by defining a signal mask. Each handler thread has its own mask which you can define to BLOCK selected signals. For example, the main thread could block SIGINT and SIGHUP and one of the handlers block SIGUSR1. If one of the blocked signals is sent to your process, it is not delivered to the process until the mask is changed to allow the signal through. Such signals are deemed pending signals.

Each handler routine and signal mask is registered with the kernel by calling the sigaction() function. This function transfers the handler address and mask to a table in the process's user area. The kernel uses this table when delivering a signal to determine which thread to run. The main thread's signal mask is set by calls to sigprocmask().



Signal Masks

Process Signal Mask

- List of signals to be blocked by the process
- Blocked signals are left pending
- Other signals are delivered to the process

Handler Signal Mask

- List of signals blocked while in the handler
- Current signal automatically blocked
- Mask only effective while executing handler

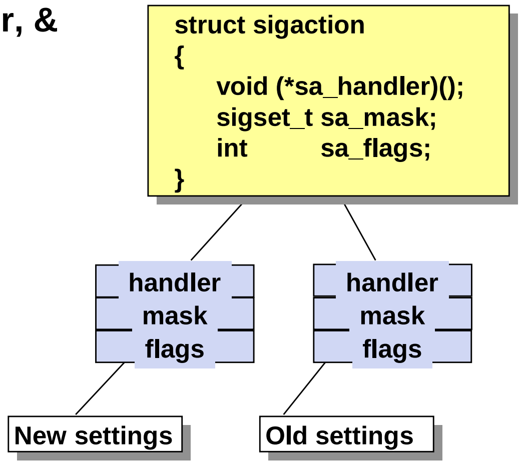
52

The process signal mask defines which signals are blocked by the process. If a signal is blocked, the kernel will ensure it is not delivered (and hence its signal handler is not called). Such a signal is left pending. A process is free to change its process signal mask at any time; this may cause pending signals to be delivered. The kernel remembers any pending signals and if the process signal mask is changed will immediately deliver any pending signals that have been unblocked.

The signal handler mask defines additional signals to be blocked while a handler is executing. Without the mechanism it would be very difficult to handle several signals delivered in quick succession. The signal being handled is automatically blocked in its own handler. This behaviour can be changed if you wish by setting the flags word of sigaction.

Installing a Signal Handler

sigaction (signalNumber, &

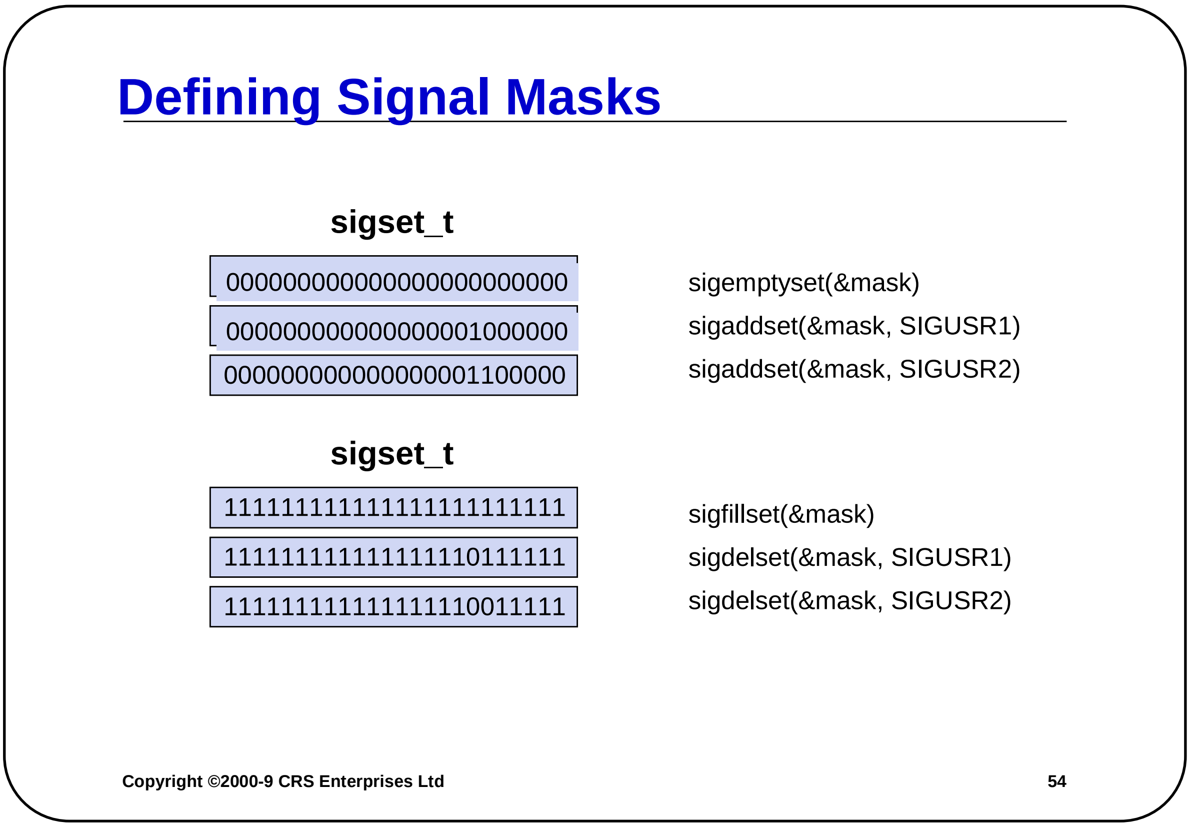


```
struct sigaction
{
      void (*sa_handler)();
      sigset_t sa_mask;
      int          sa_flags;
}
```

Signal handlers may be installed using the sigaction() system call. The key parameters in the sigaction() call are the two sigaction structures. Their elements are shown in the slide.

The first field is the address of the signal handler; SIG_IGN (ignore signal) and SIG_DFL (default handler) are valid as well as a user-defined function. The second field is a mask of signals to be blocked while the handler is executing. The third field allows special flags to be be set.

Either of the two sigaction parameters in a call to sigaction() may be NULL. Set the old setting to NULL if you are not interested in the value being replaced. If the new setting is NULL, then the routine provides a way of examining the current signal action without changing it.



Defining Signal Masks

sigset_t

000000000000000000000000	sigemptyset(&mask)
000000000000000001000000	sigaddset(&mask, SIGUSR1)
000000000000000001100000	sigaddset(&mask, SIGUSR2)

sigset_t

111111111111111111111111	sigfillset(&mask)
111111111111111110111111	sigdelset(&mask, SIGUSR1)
111111111111111110011111	sigdelset(&mask, SIGUSR2)

54

Signal masks provide a way of restricting the types of signals delivered to a process.

A process can set up an empty mask (no signals masked) using the sigemptyset() macro and then mask out signals one at a time using sigaddset().

Alternatively, a process can set up a filled mask (all signals masked) using the sigfillset() macro and then remove signals from the mask one at a time using sigdelset().

Other System Calls

```
sigprocmask    (options, &newMask, &oldMask);
sigpending     (&pendingMask);
sigismember    (&pendingMask, SIGHUP);
pause();
```

A process may set its process signal mask using the sigprocmask() function. A process normally sets the process signal mask soon after it fires up. The process mask can be changed at any time by a further call to sigprocmask(). The first parameter to sigprocmask() can be one of the following:

SIG_BLOCK The signals in the second parameter are added to the current signal mask.

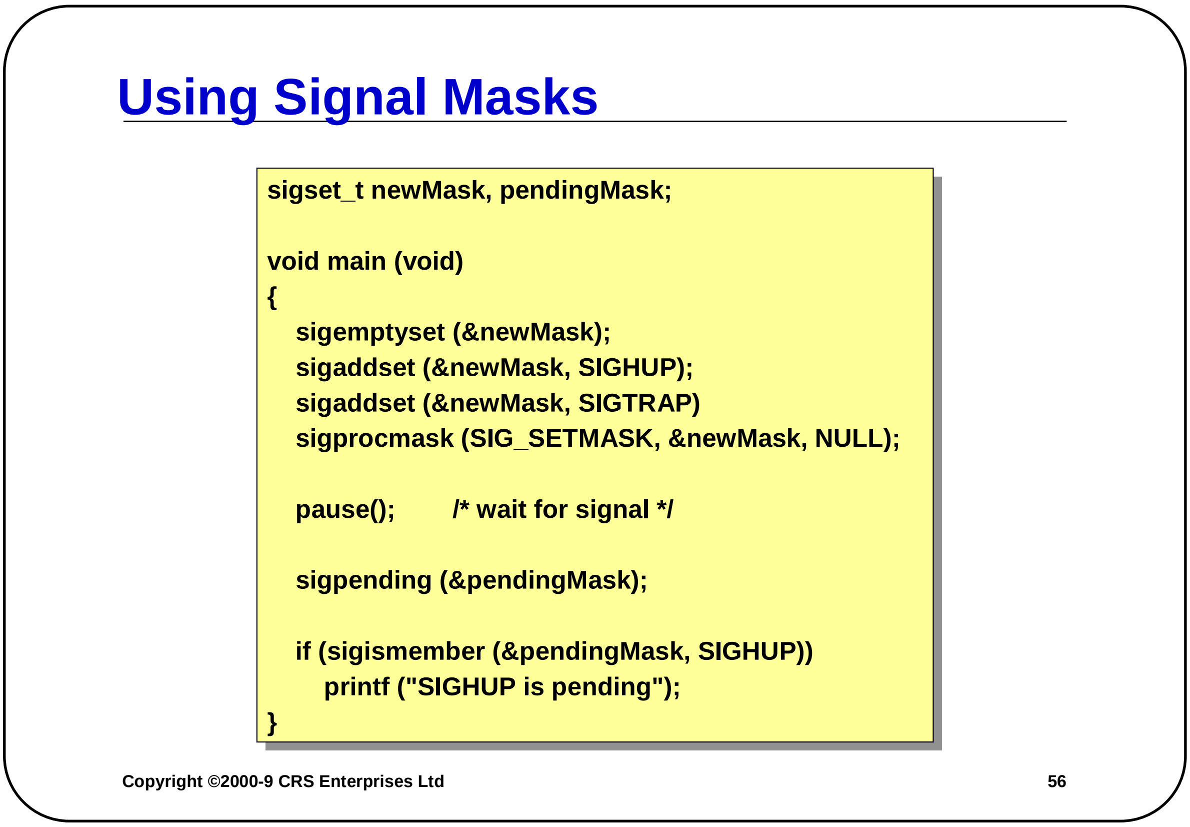
SIG_UNBLOCK The signals in the second parameter are removed from the current signal mask, i.e. they are unblocked.

SIG_SETMASK The signal mask is set explicitly to the parameter set.

The process mask is sometimes used to protect critical regions of code from being interrupted by a particular signal. The mask is set upon entry to the critical section and the cleared on leaving. If the unwanted signal is generated during execution of the critical code, it will be left pending and do no harm. It will be delivered as soon as the mask is reset.

If there are any signals pending for the process and the call to sigprocmask() unblocks the signal(s), then at least one of the signals will be delivered to the process before the routine returns. A process can examine the signals pending by using the sigpending() and sigismember() functions.

The pause() function puts a process to sleep until a signal (any signal) is delivered to the process. This is preferable to entering a tight infinite loop and thereby wasting CPU time.



Using Signal Masks

```
sigset_t newMask, pendingMask;

void main (void)
{
   sigemptyset (&newMask);
   sigaddset (&newMask, SIGHUP);
   sigaddset (&newMask, SIGTRAP)
   sigprocmask (SIG_SETMASK, &newMask, NULL);

   pause();       /* wait for signal */

   sigpending (&pendingMask);

   if (sigismember (&pendingMask, SIGHUP))
      printf ("SIGHUP is pending");
}
```

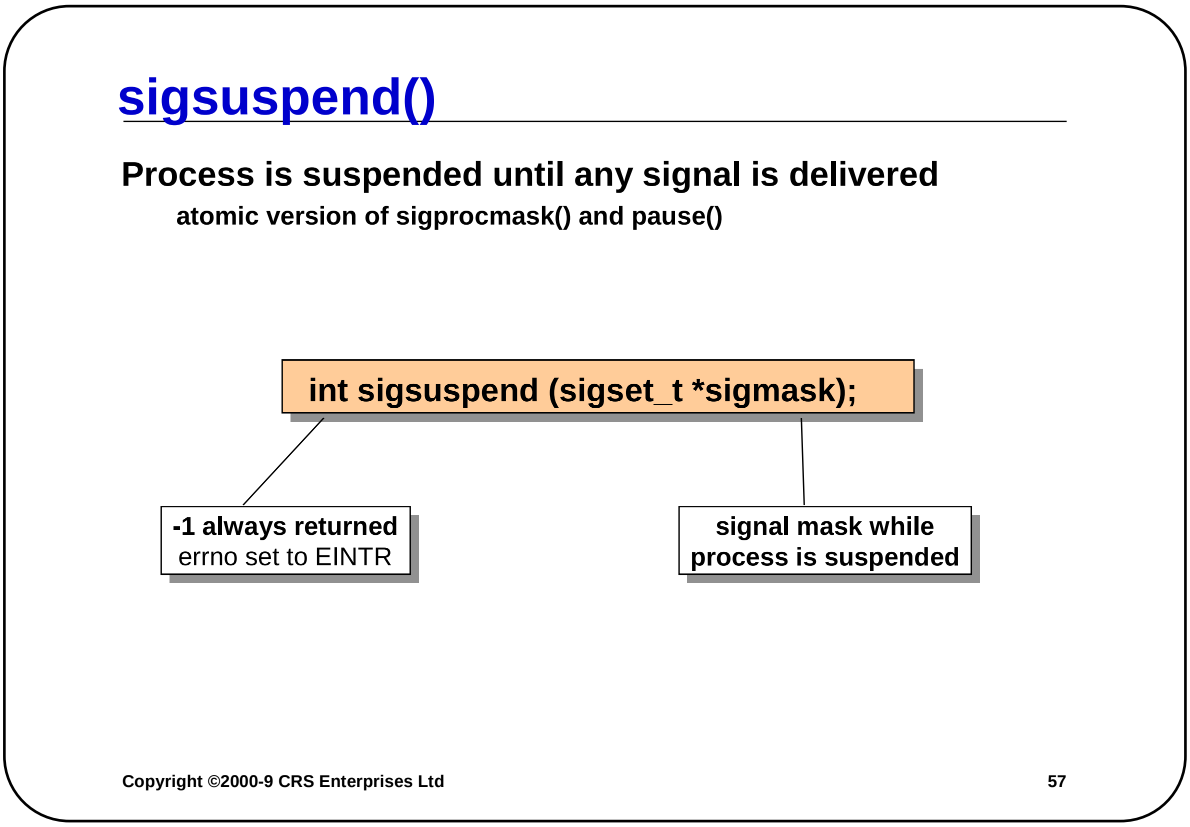
Copyright ©2000-9 CRS Enterprises Ltd 56

In this example a process sets its process signal mask to block SIGHUP and SIGTRAP. The process is not interested in its old process mask and therefore passes NULL as the third parameter to sigprocmask().

The process then puts itself to sleep and waiting for a signal other than SIGHUP or SIGTRAP.

When pause() returns (some other signal has been delivered), the process is awakened by the kernel and the appropriate signal handler called. SIGHUP and SIGTRAP are still blocked.

To find out if either of these signals are pending, the process calls sigpending(). This routine returns a mask containing a list of all pending signals. Calling sigismember() checks for an individual pending signal (SIGHUP).



sigsuspend()

Process is suspended until any signal is delivered

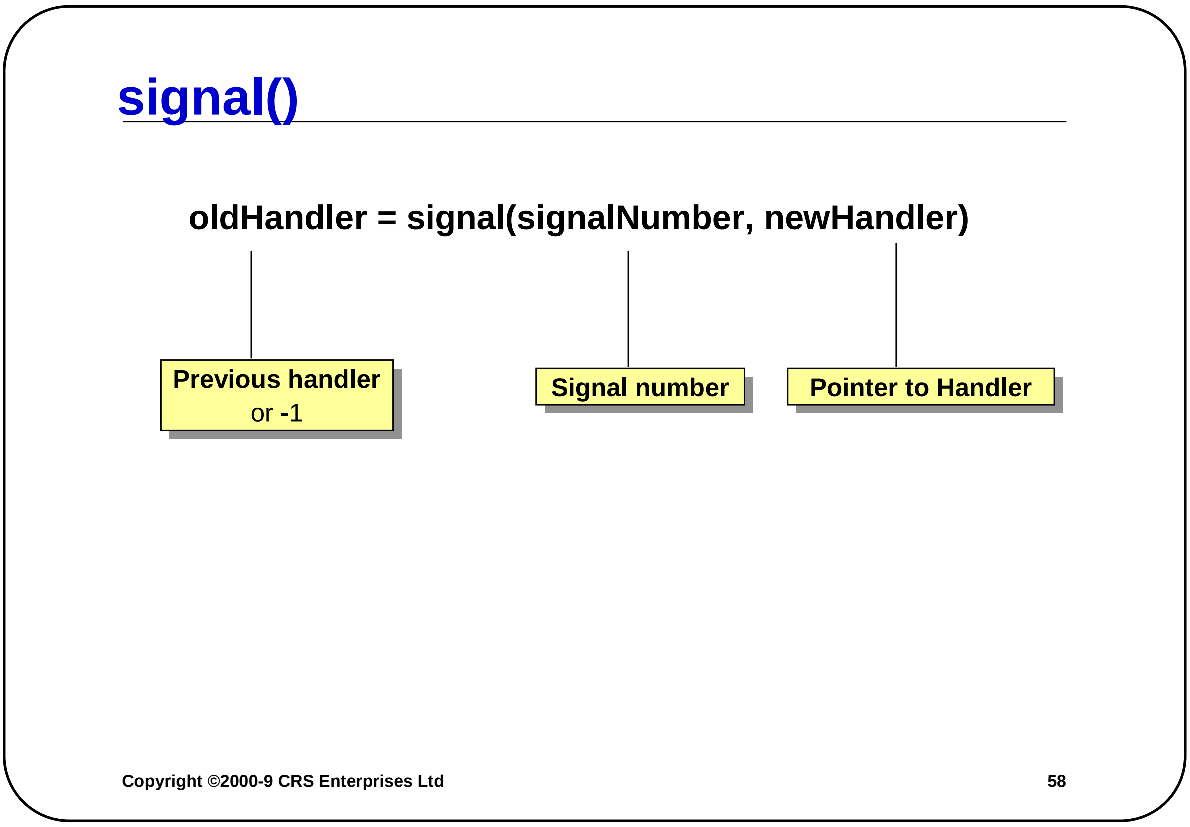
atomic version of sigprocmask() and pause()

int sigsuspend (sigset_t *sigmask);

sigsuspend() allows a process to be suspended while waiting for a signal to arrive. First, however, the routine changes the process mask. The process is resumed as soon as a signal is delivered.

sigsuspend() is an atomic version of sigprocmask() and pause(). It is used to avoid race conditions: a race condition is where one of two conditions could occur dependent on timing considerations.

Suppose that in the previous example we are only expecting one signal to be delivered after we change the process mask. This signal will wake us up from the pause() call. But what if this signal is delivered too early; just before we call pause() ? In that case the signal will be handled as usual, but after we subsequently call pause() we will sleep forever because no more signals will get delivered. This is a good example of a race condition.

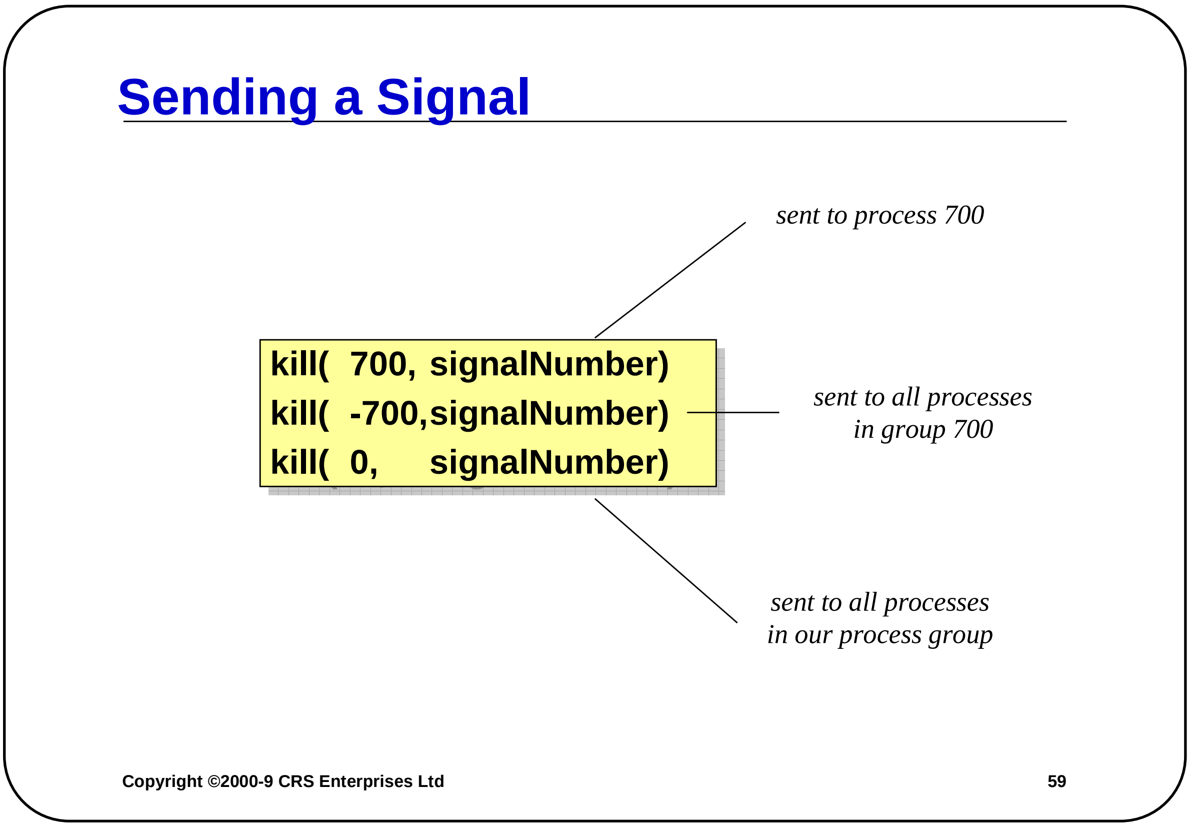


signal()

oldHandler = signal(signalNumber, newHandler)

58

The signal() system call was the original method of installing signal handlers before signal masks were introduced. All new code should use the more flexible sigaction() call. You need to be aware of the signal() system call because it is still found in a great many older applications.



Sending a Signal

```
kill(  700, signalNumber)
kill(  -700,signalNumber)
kill(  0,     signalNumber)
```

sent to process 700

sent to all processes in group 700

sent to all processes in our process group

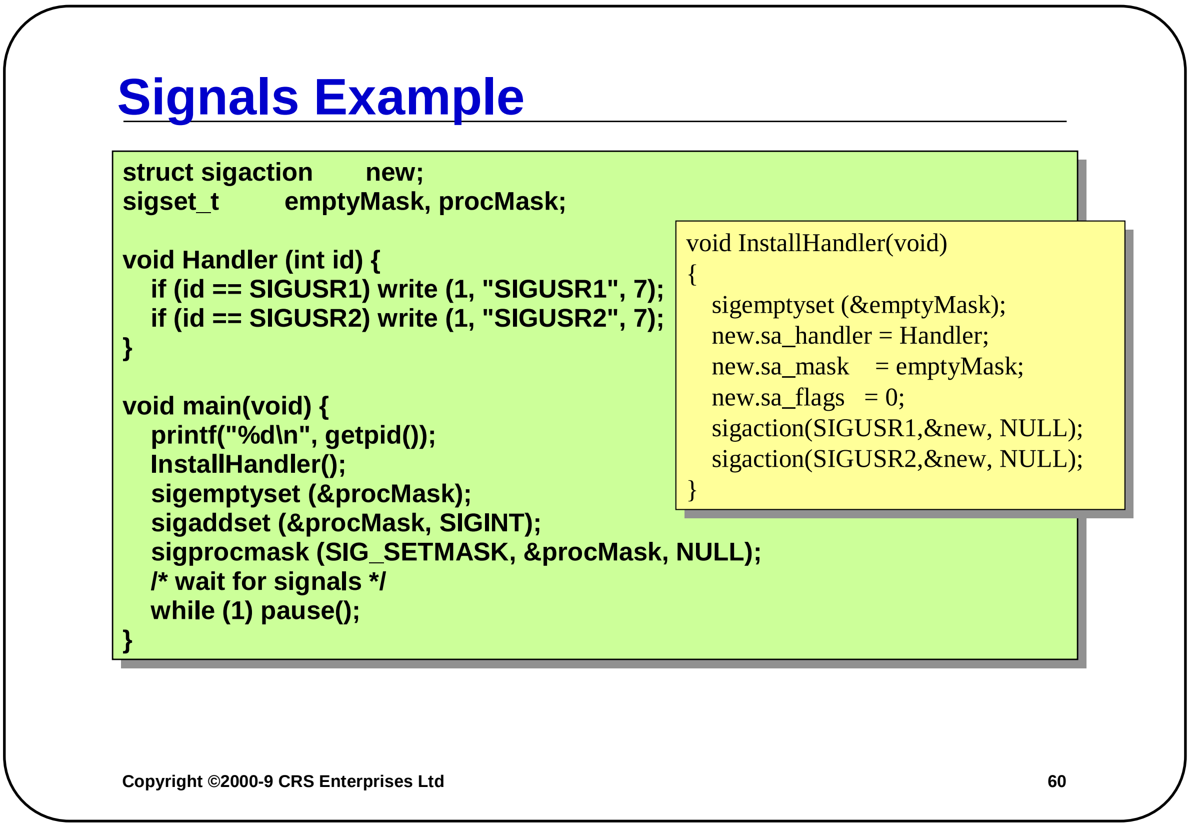
Copyright ©2000-9 CRS Enterprises Ltd 59

A process can send a signal using the kill() system call. Kill is a misnomer; the kill signal was the original signal implemented in early versions of Unix and hence the naming of the system call.

kill() allows a process to send a signal to a different process. For fairly obvious security reasons, a process with an effective user id of anything other than superuser may only send signals to processes with the same effective user id. Under normal circumstances, kill() will signal the specified process id. There are some special cases, however.

pid	users	effect
> 0	all	The signal is sent to process pid.
0	all	The signal is sent to all processes in the sender's process group.
-1	not root	The signal is sent to all processes whose real uid is the same as the sender's effective uid.
-1	root	The signal is sent to all user processes (i.e. all processes except pid 0 and 1).
< -1	all	The signal is sent to all processes in the process group abs(pid).

If the signal number parameter to kill() is 0, then no signal is sent, but the validity of the process id argument is checked. This checks if a particular process exists.



Signals Example

```
struct sigaction       new;
sigset_t        emptyMask, procMask;

void Handler (int id) {
    if (id == SIGUSR1) write (1, "SIGUSR1", 7);
    if (id == SIGUSR2) write (1, "SIGUSR2", 7);
}

void main(void) {
    printf("%d\n", getpid());
    InstallHandler();
    sigemptyset (&procMask);
    sigaddset (&procMask, SIGINT);
    sigprocmask (SIG_SETMASK, &procMask, NULL);
    /* wait for signals */
    while (1) pause();
}
```

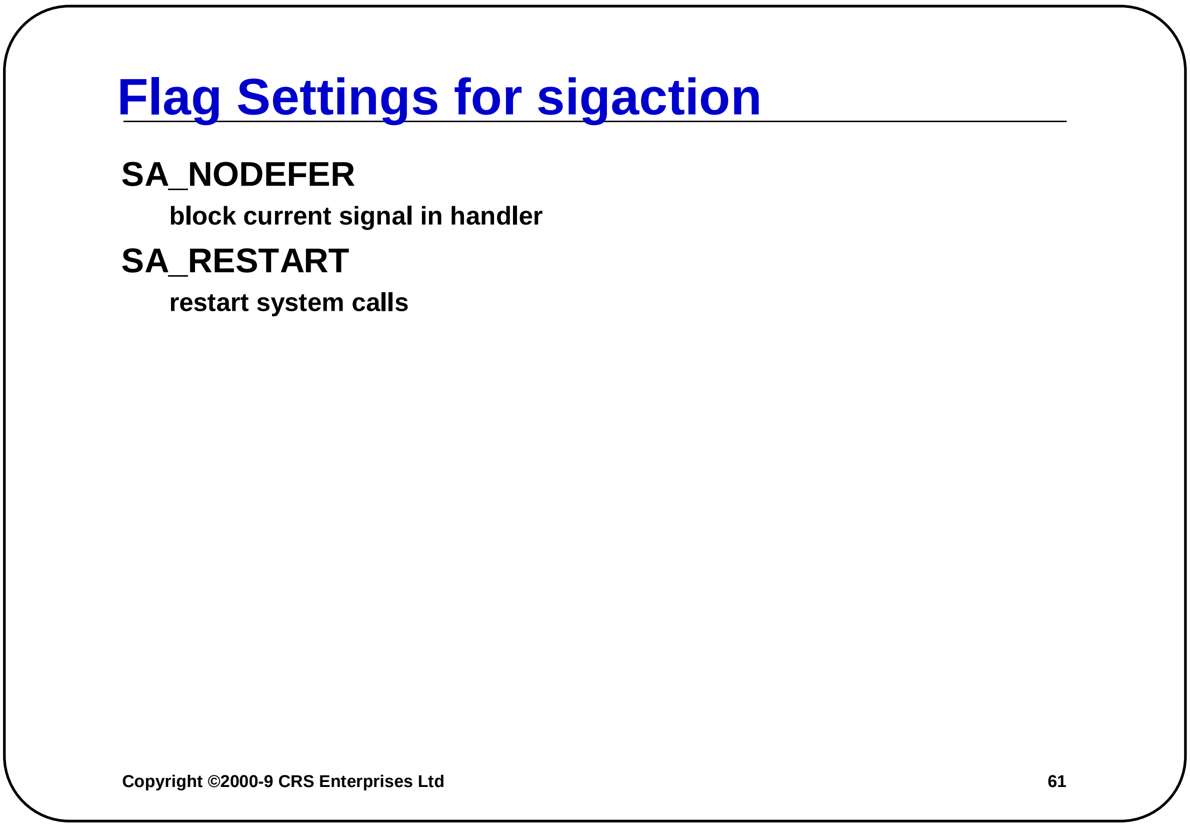
```
void InstallHandler(void)
{
    sigemptyset (&emptyMask);
    new.sa_handler = Handler;
    new.sa_mask    = emptyMask;
    new.sa_flags   = 0;
    sigaction(SIGUSR1,&new, NULL);
    sigaction(SIGUSR2,&new, NULL);
}
```

The idea in this code fragment is to use one signal handler for SIGUSR1 and SIGUSR2 and block delivery of SIGINT signals. This allows us to see how the sigaction() and sigprocmask() system calls work.

The sigaction() call installs the signal handler along with its signal mask and optional flags. We have chosen to use an empty mask (no signals blocked) and no flags. Since we are not interested in the state of the previous handler, we have used a NULL pointer as the third argument.

The sigprocmask() call allows us to define which signals are to be blocked in the current process. We have chosen to block the SIGINT signal.

Once the process has called pause(), we can send SIGUSR1 and SIGUSR2 signals to the process. The pause() system call will return and the handler will be called. If we try to send a SIGINT signal, the kernel blocks the signal and pause() does not return.



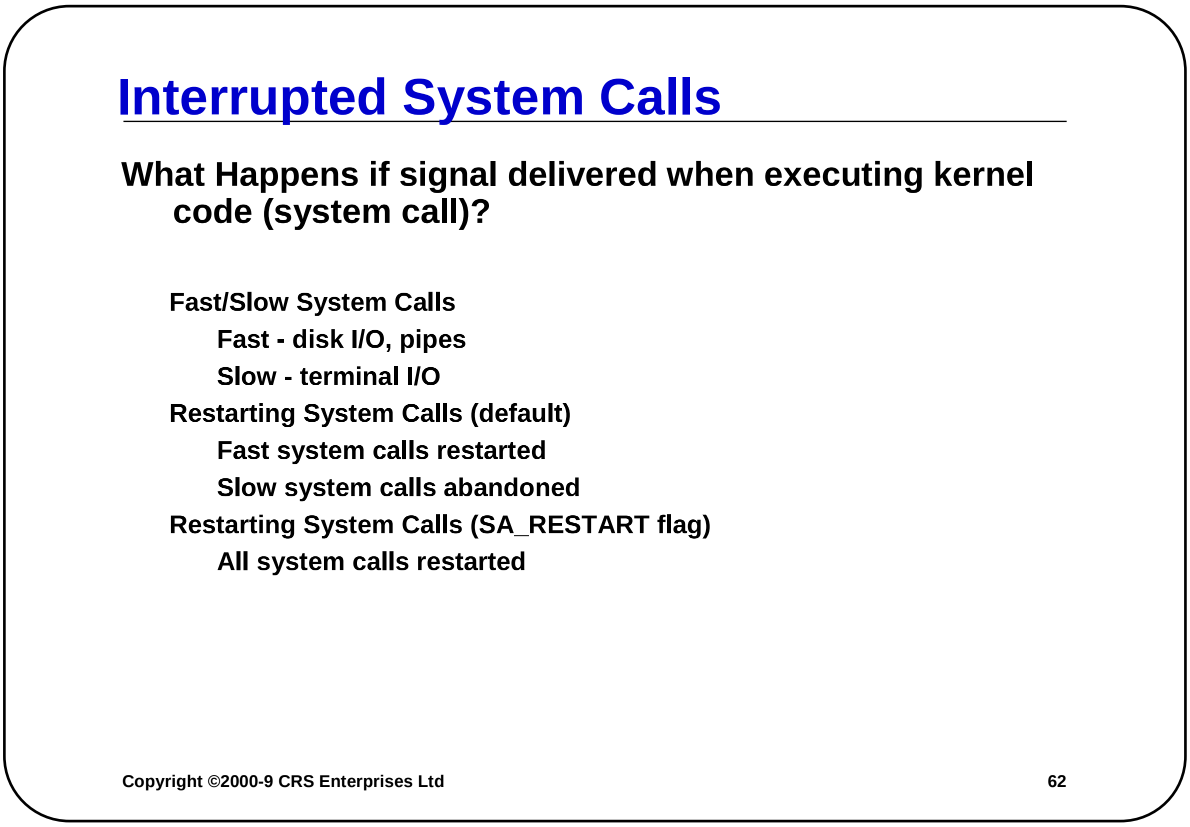
Flag Settings for sigaction

SA_NODEFER

block current signal in handler

SA_RESTART

restart system calls



Interrupted System Calls

What Happens if signal delivered when executing kernel code (system call)?

- **Fast/Slow System Calls**
  - **Fast - disk I/O, pipes**
  - **Slow - terminal I/O**
- **Restarting System Calls (default)**
  - **Fast system calls restarted**
  - **Slow system calls abandoned**
- **Restarting System Calls (SA_RESTART flag)**
  - **All system calls restarted**

62

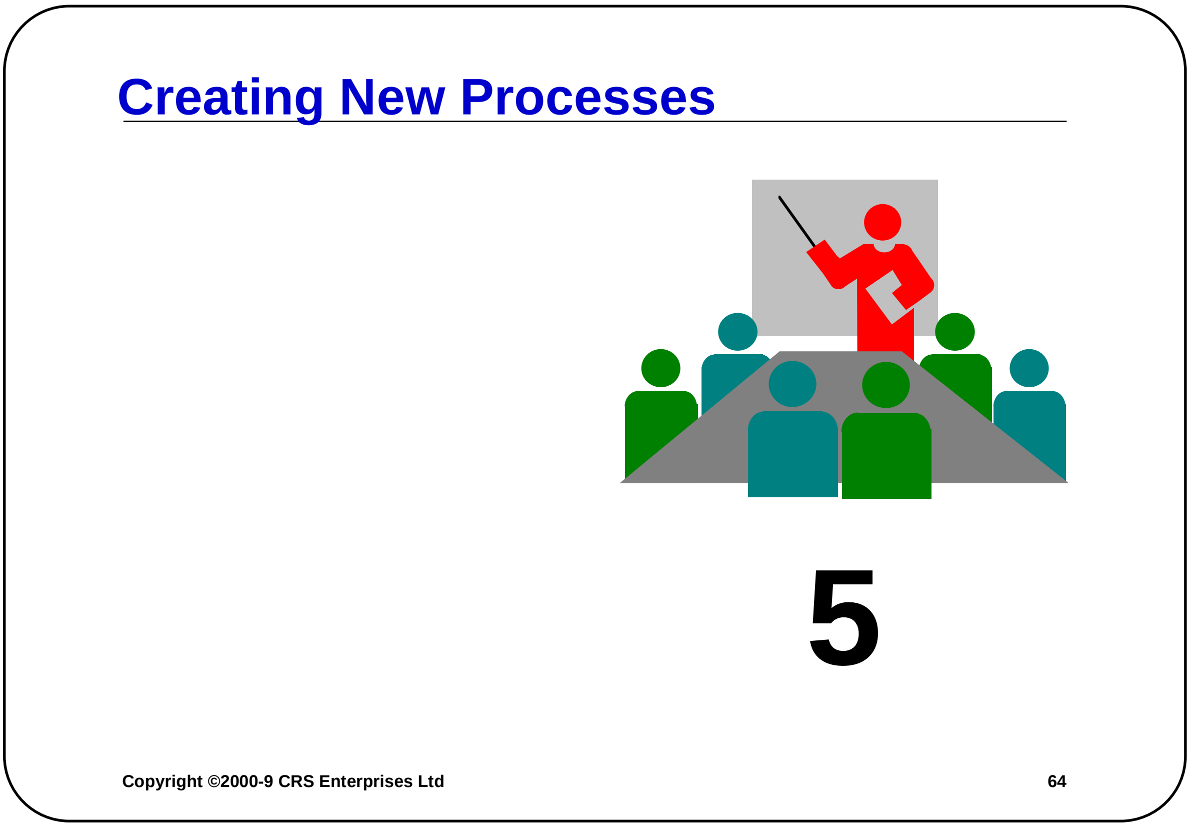
Normally a signal will be delivered while you are executing user code (the main thread or a handler thread). Execution resumes at the point of interruption after the handler completes. Should this behaviour be the same if the signal is delivered while the kernel code is executing a system call?

The answer to the question is usually yes. However, it is possible that the signal was generated because of a problem encountered while executing the system call itself. In this case the system call should be abandoned.

It has now been decided to classify system calls as either fast or slow. Slow system calls are loosely defined as calls that may never return (e.g. a read on a terminal device). These are the calls that may run into problems and be responsible for generating signals. For fast system calls it is assumed that if any problems arises, the kernel need not generate a signal (it just returns an error code).

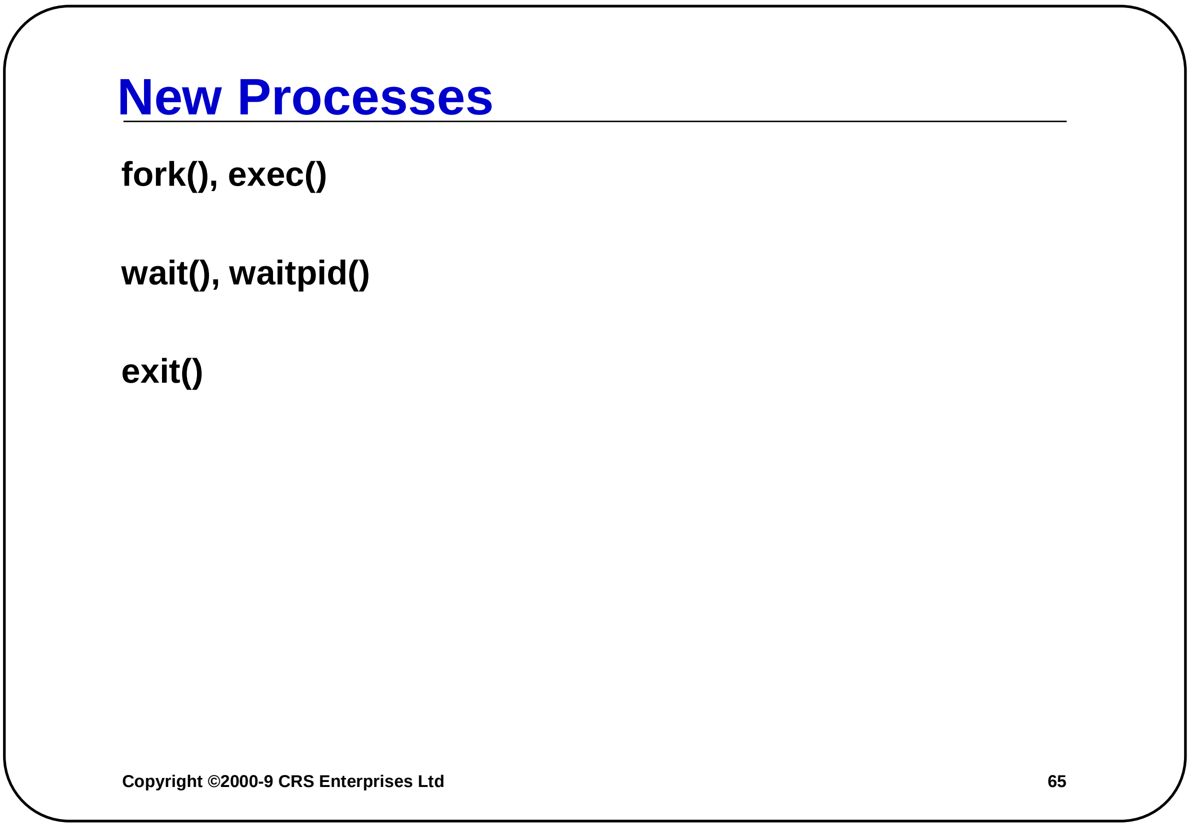
Slow system calls are not restarted unless you set the SA_RESTART flag of the sigaction system call. Fast system calls are always restarted.

5



Creating New Processes

5



New Processes

fork(), exec()

wait(), waitpid()

exit()

65

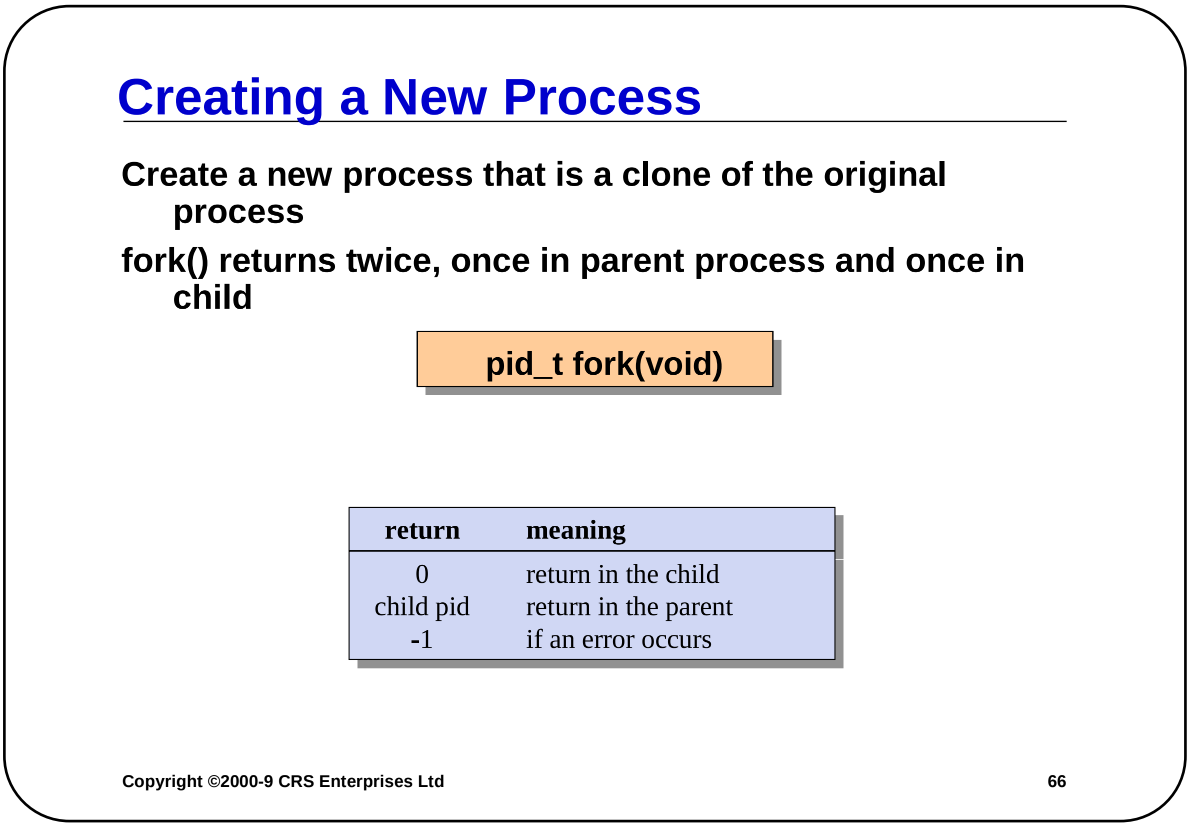
In this chapter we consider how to create new processes and synchronise processes.

Using the fork() system call is the only way to create a new process. fork() creates a clone process. The original process is called the parent; the new process is called the child.

The exec() system call does not create a new process, instead it overlays the process address space of the current process (p-area and u-area) with a new executable from the disk. The name of the executable is specified as the first parameter to exec().

Once a parent has created one of more child processes it has the option to wait or not to wait for them to complete. The wait() and waitpid() system calls allow the parent to wait.

When a child terminates it calls exit() to define a termination code. The parent can examine this code by calling wait() or waitpid().



Creating a New Process

Create a new process that is a clone of the original process

fork() returns twice, once in parent process and once in child

pid_t fork(void)

return	meaning
0	return in the child
child pid	return in the parent
-1	if an error occurs

66

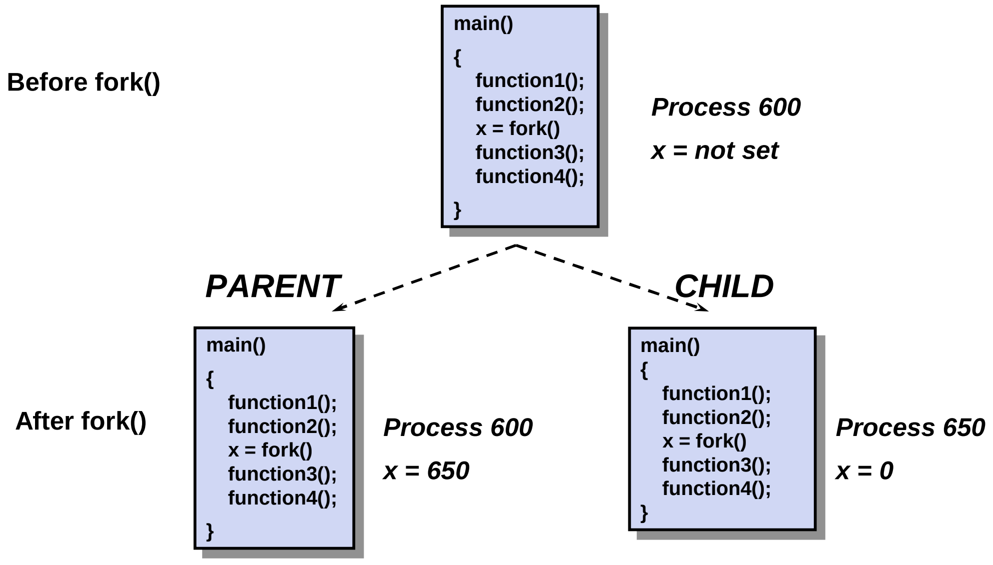
The fork() system call is used to create a new process. fork() causes the calling process to clone itself, creating a new process that is identical to the original except for pid. The new process is known as the child process; the calling process is known as the parent.

fork() takes no parameters. Since the child process is a clone of the parent, it will be executing the same instructions. Therefore, fork() returns twice; once in the child and once in the parent. The return values are:

-1	if an error occurs.
0	when fork() returns in the child process.
child pid	the pid of the newly-created child process when fork() returns in the parent process.

The fork() mechanism is the only way in which a new process can be created in Unix. The init process (pid of 1), is hand built by the kernel when the system starts. After this, every process in the system is created through execution of the fork() call.

fork()



Before fork()

```
main()
{
    function1();
    function2();
    x = fork()
    function3();
    function4();
}
```

Process 600

x = not set

PARENT

CHILD

After fork()

```
main()
{
    function1();
    function2();
    x = fork()
    function3();
    function4();
}
```

Process 600

x = 650

```
main()
{
    function1();
    function2();
    x = fork()
    function3();
    function4();
}
```

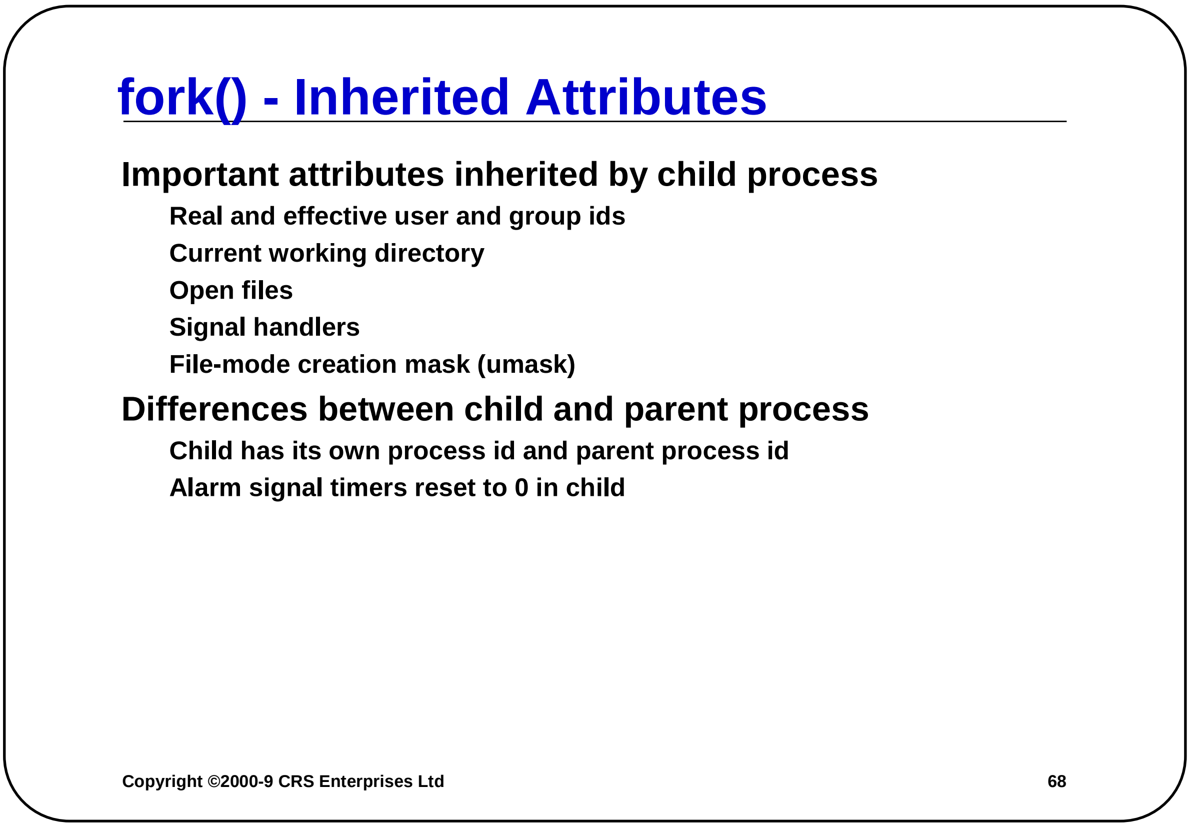
Process 650

x = 0

67

The fork() system call replicates a process in memory. Before fork() is called, only a parent process exists in memory. After fork() returns, two almost identical processes exist in memory: the original parent and the newly created child.

The parent and the child are distinct processes but their process address spaces are almost exact copies of one another; they execute the same code and have almost identical data areas. The only difference between the two processes is the value stored in the variable x returned by fork(). The parent's copy of x contains the process id of the child, whereas the child's copy of x contains zero.



fork() - Inherited Attributes

Important attributes inherited by child process

- Real and effective user and group ids
- Current working directory
- Open files
- Signal handlers
- File-mode creation mask (umask)

Differences between child and parent process

- Child has its own process id and parent process id
- Alarm signal timers reset to 0 in child

Copyright ©2000-9 CRS Enterprises Ltd 68

The fork() system call causes the creation of a new process that is a duplicate of the calling process. We have seen the the process address space is cloned; the same applies to the process's user area. Nearly all the process attributes stored in the u-area are clone, the exceptions being those attributes where duplication would be meaningless. Thus the child process inherits the majority of the parent process's attributes.

Attributes inherited:

The child process has the same real and effective user and group ids as the parent.

The child has the same current working directory as the parent.

Any files that were open in the parent process will also be open in the child. The child receives its own copy of the parent's descriptor table, but the actual files remain accessible through the descriptors.

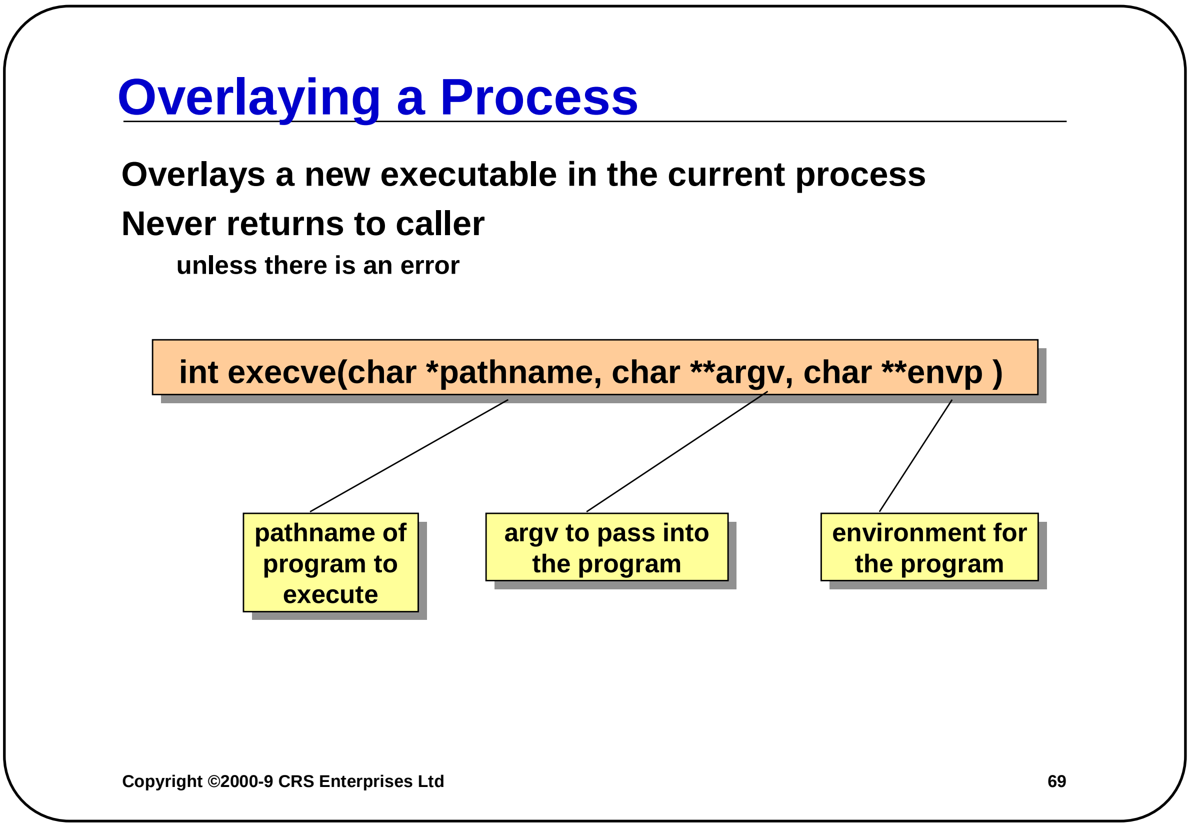
All signal-handling information is passed from parent to child.

The file-mode creation mask, represented by umask, is passed from parent to child.

Attributes not inherited:

The child will have a different process id and a different parent process id.

Any timers being accumulated for an alarm in the parent process will be reset to 0 in the child.



Overlaying a Process

Overlays a new executable in the current process

Never returns to caller

unless there is an error

int execve(char *pathname, char **argv, char **envp)

pathname of program to execute

argv to pass into the program

environment for the program

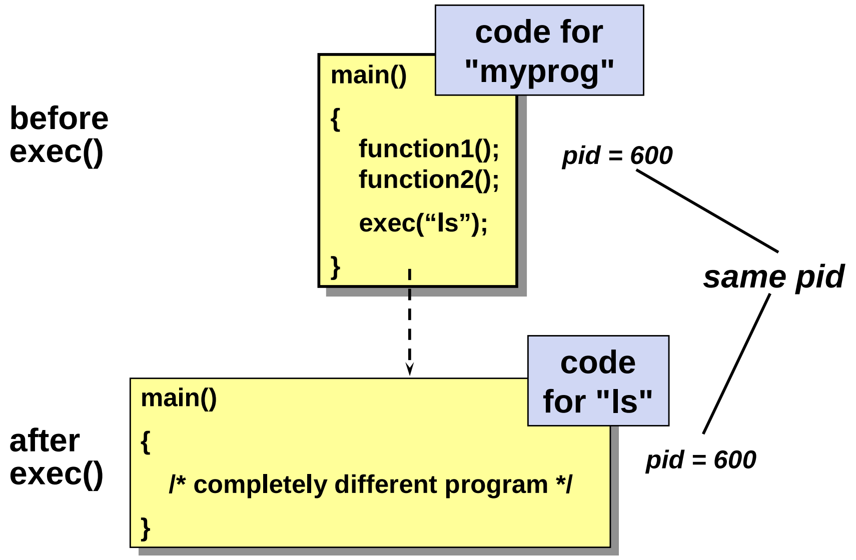
Copyright ©2000-9 CRS Enterprises Ltd 69

A process address space is overlayed when a process issues the execve() system call. This causes the current process to be "replaced" with the program in the file specified by the pathname parameter to the system call. The program will be executed with the argv vector as specified in the system call and in the environment specified by the envp parameter.

Because the execve() call causes the process to be overwritten, it will never return to the caller unless there has been an error that prevented the program being executed. It is important to realize that execve() does not cause the creation of a new process.

Although the p-area is replaced, the process's u-area is left unchanged. Thus the new code in the p-area retains all the attributes of the process.

exec()



before exec()

code for "myprog"

```
main()
{
    function1();
    function2();

    exec("ls");
}
```

pid = 600

same pid

after exec()

code for "ls"

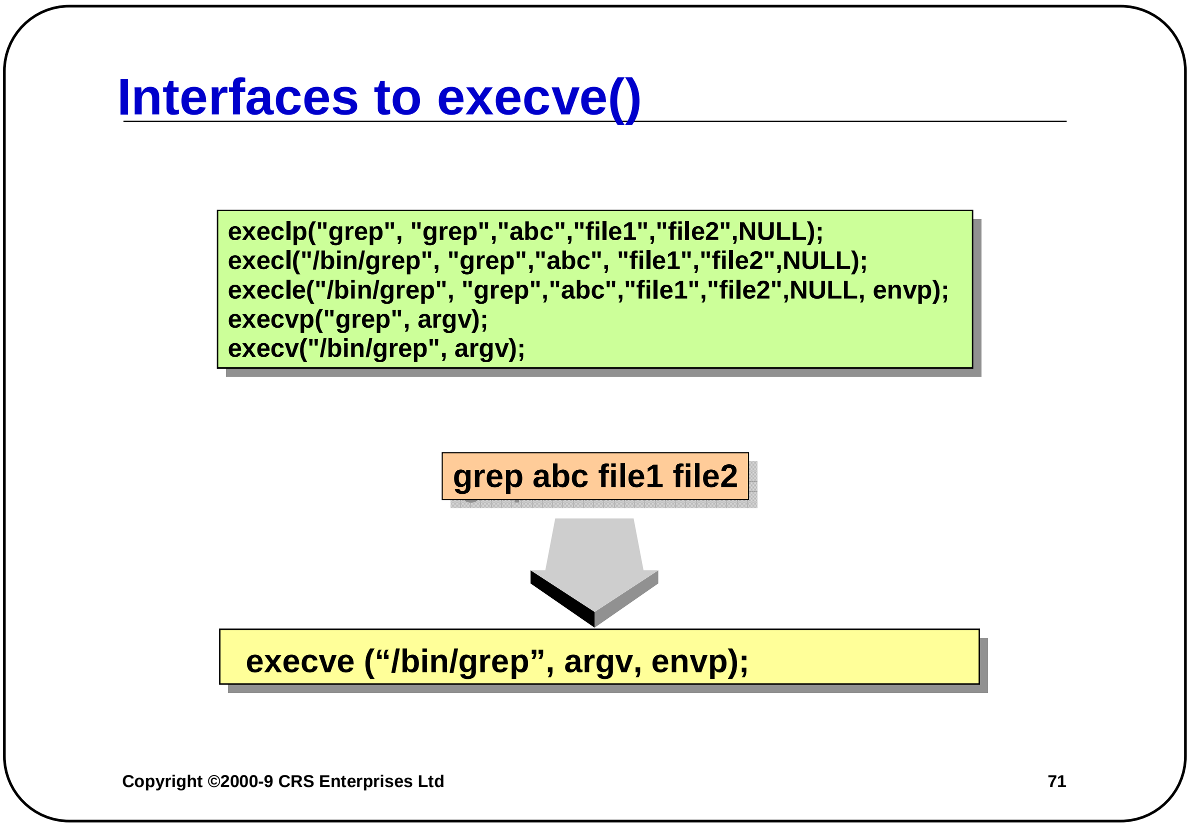
```
main()
{
    /* completely different program */
}
```

pid = 600

The execve() system call completely changes the process’s address space (both data and code sections). The new code is started at its entry point as if it were a new process.

The execve() system call leaves the process's user area almost unchanged. Most attributes of a process are retained across an execve() call. For example, the new code executing within the process inherits the open file table of the old code.

Note: If you do not want open files to remain open after an execve() call, make sure you set the CLOSE_ON_EXEC flag when you first open the file.



Interfaces to execve()

```
execlp("grep", "grep","abc","file1","file2",NULL);
execl("/bin/grep", "grep","abc", "file1","file2",NULL);
execle("/bin/grep", "grep","abc","file1","file2",NULL, envp);
execvp("grep", argv);
execv("/bin/grep", argv);
```

grep abc file1 file2

execve (“/bin/grep”, argv, envp);

There are a number of exec routines that accept different forms of the parameters, process them into the correct format and then call execve(). The functionality contained in these different versions is indicated by the letters following exec in the routine name:

l arguments are passed as individual elements

v accept the arguments in argv vector format.

p will search the directories in the current PATH to find filename and construct the full pathname before calling execve().

e specific environment can be passed to the program being executed

As well as execve(), we therefore have:

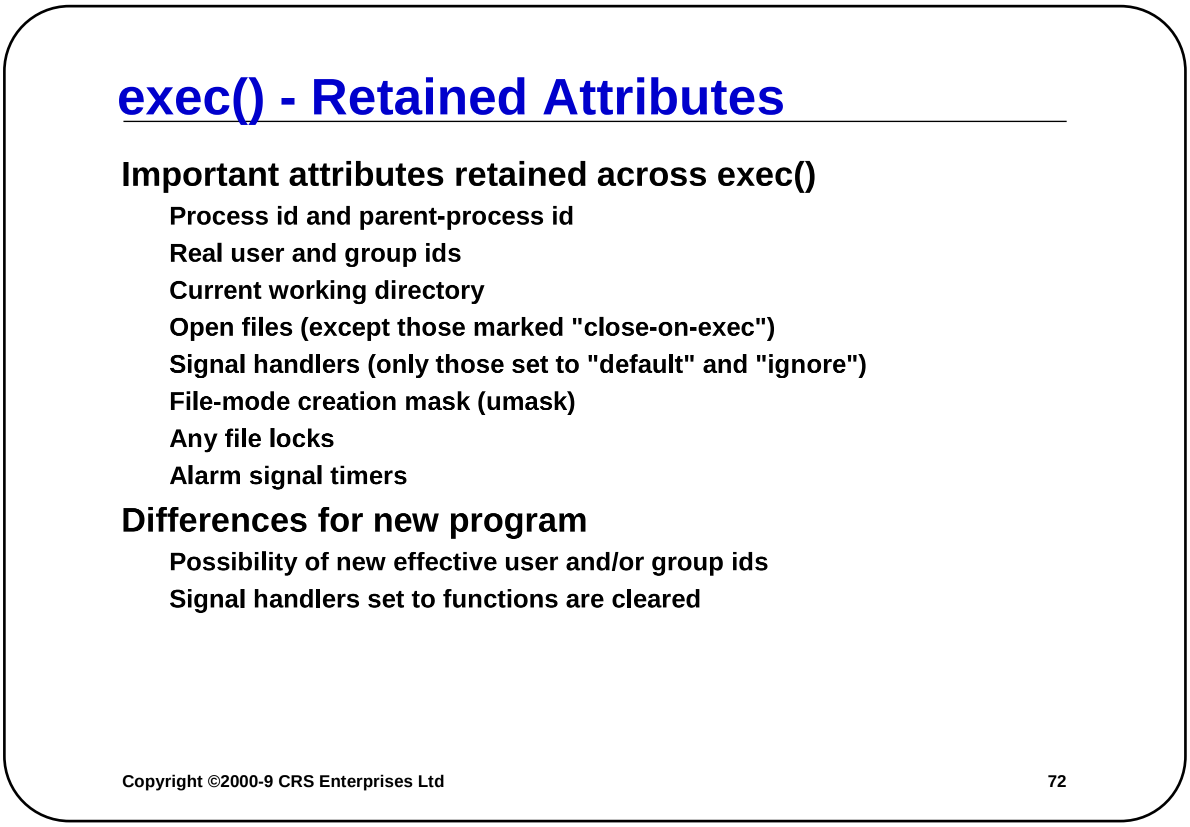
execlp Allow the arguments to be passed individually, treat the program name as a filename and use the current search path to find the file. Execute with a copy of the current environment.

execl Allow individual arguments, treat the program name as a full pathname and copy the current environment.

execle Allow individual arguments, treat the program name as a full pathname and use the supplied environment.

execvp Use the argv vector format for arguments, treat the program name as a filename and use the current search path to find the file. Execute with a copy of the current environment.

execv Use the argv vector format for the arguments, treat the program name as a full pathname and use a copy of the current environment.



exec() - Retained Attributes

Important attributes retained across exec()

- Process id and parent-process id
- Real user and group ids
- Current working directory
- Open files (except those marked "close-on-exec")
- Signal handlers (only those set to "default" and "ignore")
- File-mode creation mask (umask)
- Any file locks
- Alarm signal timers

Differences for new program

- Possibility of new effective user and/or group ids
- Signal handlers set to functions are cleared

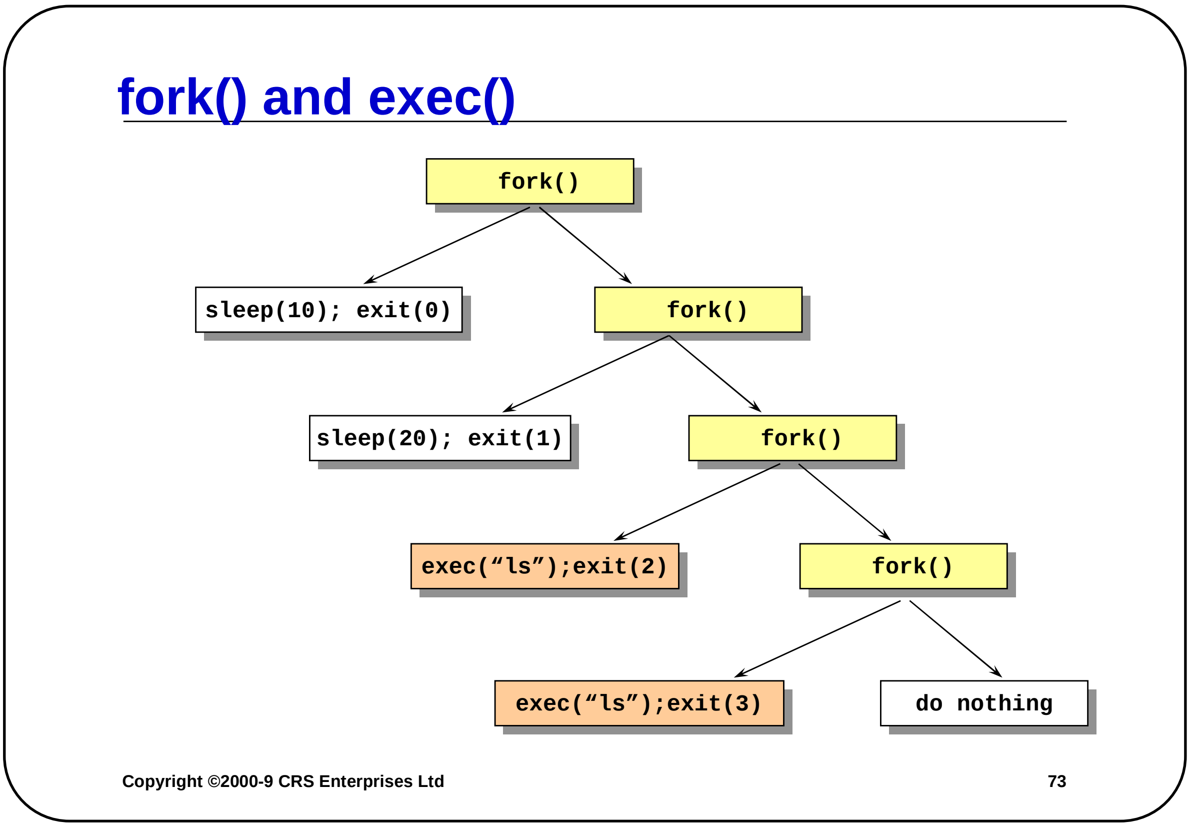
Copyright ©2000-9 CRS Enterprises Ltd 72

Most of a process's attributes will remain unchanged through a call to exec(). In particular:

> Since there is no new process created, the pid and parent pid will be the same.
>
> Real user-ids and group-ids will be the same.
>
> The current working directory will be the same.
>
> Open files will remain open, except those which have been opened using the close-on-exec flag.
>
> Any signal handlers set to take default action, or for the signal to be ignored, will be retained.
>
> The file mode creation mask, umask will be retained.
>
> Any locks held on files will be retained.
>
> Timers for alarm signals will be unchanged.

Some differences may exist between the calling process and the new program:

> The effective user and group id may change if the file is marked for set-uid or set-gid execution.
>
> Any signal actions set to call user-defined handlers will be cleared, since the addresses of the handler routines referred to the old program and will now be meaningless.



fork() and exec()

fork()

sleep(10); exit(0)

fork()

sleep(20); exit(1)

fork()

exec(“ls”);exit(2)

fork()

exec(“ls”);exit(3)

do nothing

The example shown opposite calls *fork*() four times to create four child processes. The four child processes together with the parent all execute simultaneously.

Since *fork*() replicates a parent's address space, each child has an exact copy of the parent's code. However, various if statements ensure that each child executes different sections of the code.

The parent calls *fork*() and then sleeps for 10 seconds before exiting.

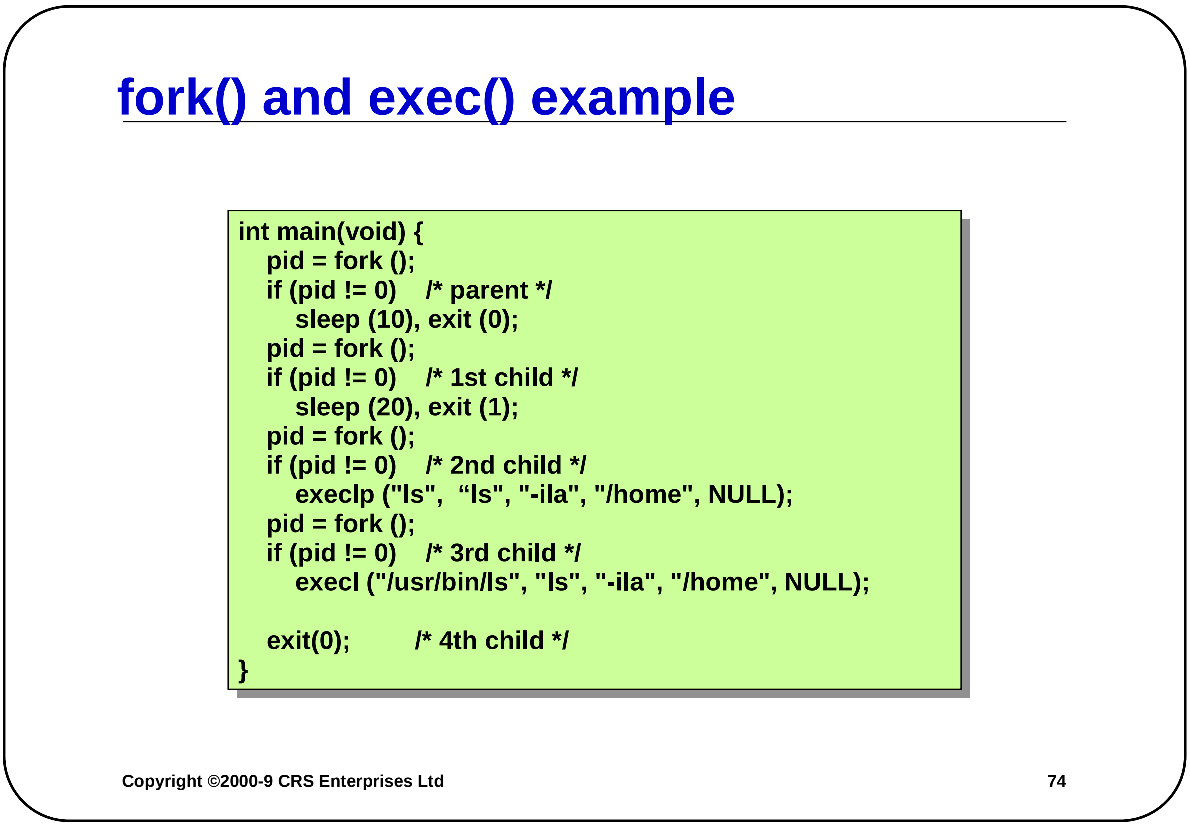
The first child calls *fork*() and then sleeps for 20 seconds before exiting.

The second child calls *fork*() and then overlays itself with the *ls* program.

The third child calls *fork*() and then overlays itself with the *ls* program.

The last child has no code to execute and therefore does nothing!

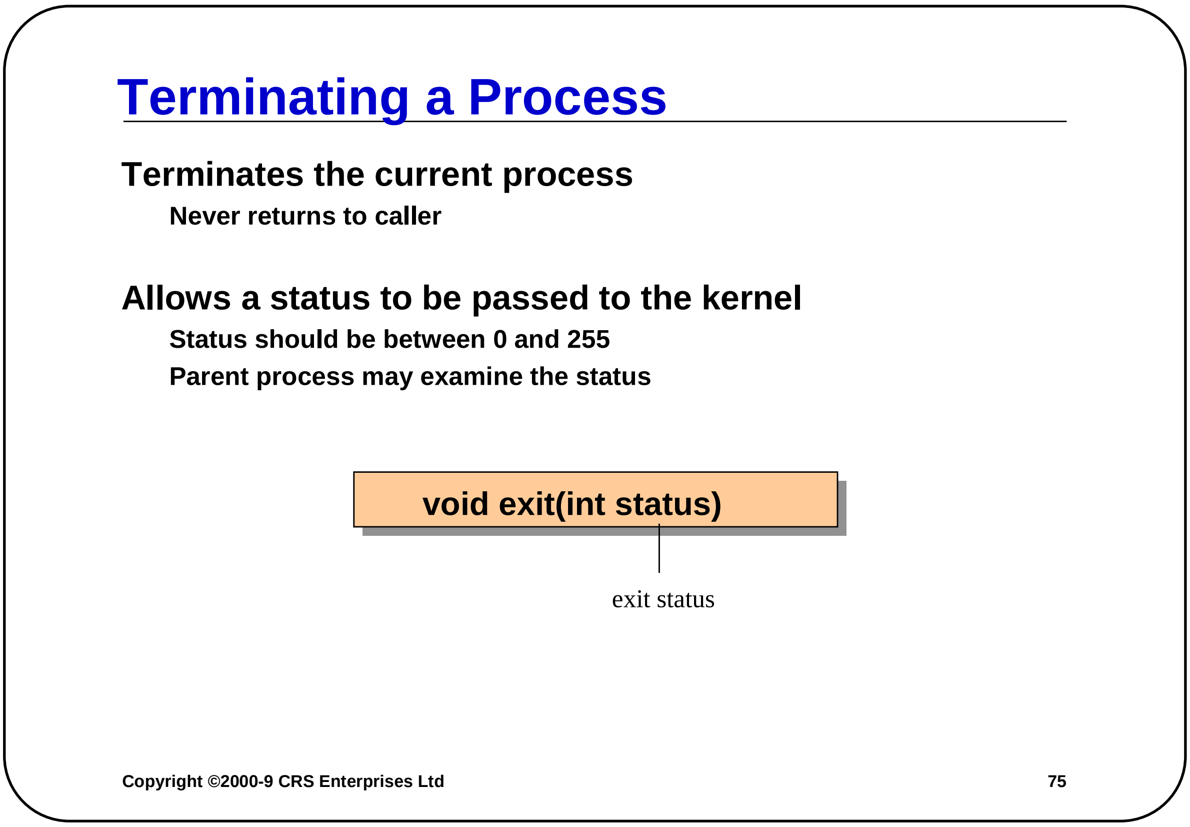
Be aware that *fork*() is called by one process, but returns in two processes. The return from *fork*() is zero for the child and non-zero for the parent.



fork() and exec() example

```
int main(void) {
    pid = fork ();
    if (pid != 0)    /* parent */
        sleep (10), exit (0);
    pid = fork ();
    if (pid != 0)    /* 1st child */
        sleep (20), exit (1);
    pid = fork ();
    if (pid != 0)    /* 2nd child */
        execlp ("ls",  “ls", "-ila", "/home", NULL);
    pid = fork ();
    if (pid != 0)    /* 3rd child */
        execl ("/usr/bin/ls", "ls", "-ila", "/home", NULL);

    exit(0);        /* 4th child */
}
```

This slide shows the implementation of the fork() and exec() pattern from the previous slide.



Terminating a Process

Terminates the current process

- **Never returns to caller**

Allows a status to be passed to the kernel

- **Status should be between 0 and 255**
- **Parent process may examine the status**

```
void exit(int status)
```

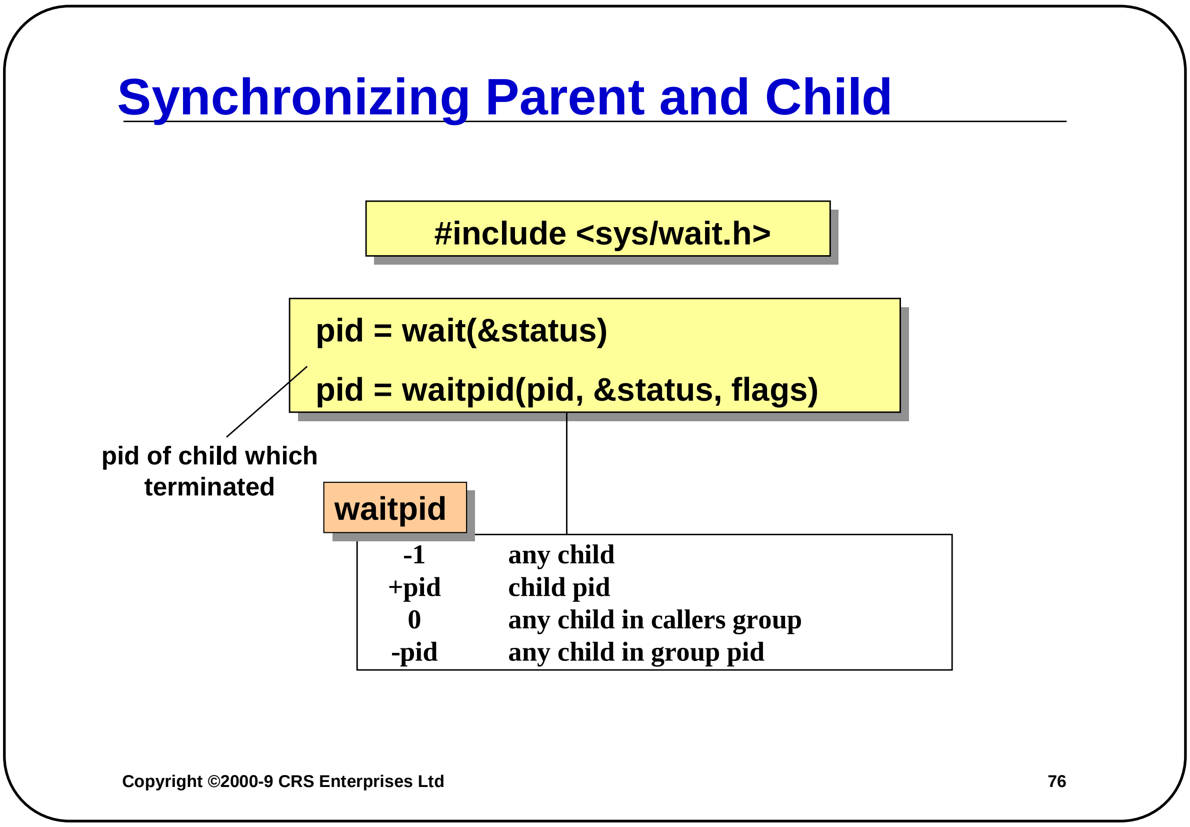
exit status

A process terminates through the exit() system call. A successful call to exec will never return. Any open files are closed and resources are freed.

exit() allows a process to pass a status value into the kernel, which can be examined at a later time by the process's parent (see later section on the wait() system call). The Unix convention is that an exit status of 0 indicates normal termination of the process; anything else indicates that an error condition occurred.

The exit status occupies only the low-order 8 bits of the status value (the kernel often manipulates the value and assumes only the low 8 bits are valid). Hence the exit status of a process should be between 0 and 255.

The stdio library provides a function with an identical name to the exit() system call. This has the same interface as the system call, but arranges for any pending stdio buffers to be flushed before calling the exit() system call. It is likely that the stdio function will be linked to a program by default, so a process wishing to access the system call directly, to avoid any buffer flushing from the stdio library, should use the alternative function interface _exit().



Synchronizing Parent and Child

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

```
pid = wait(&status)

pid = waitpid(pid, &status, flags)
```

pid of child which terminated

waitpid

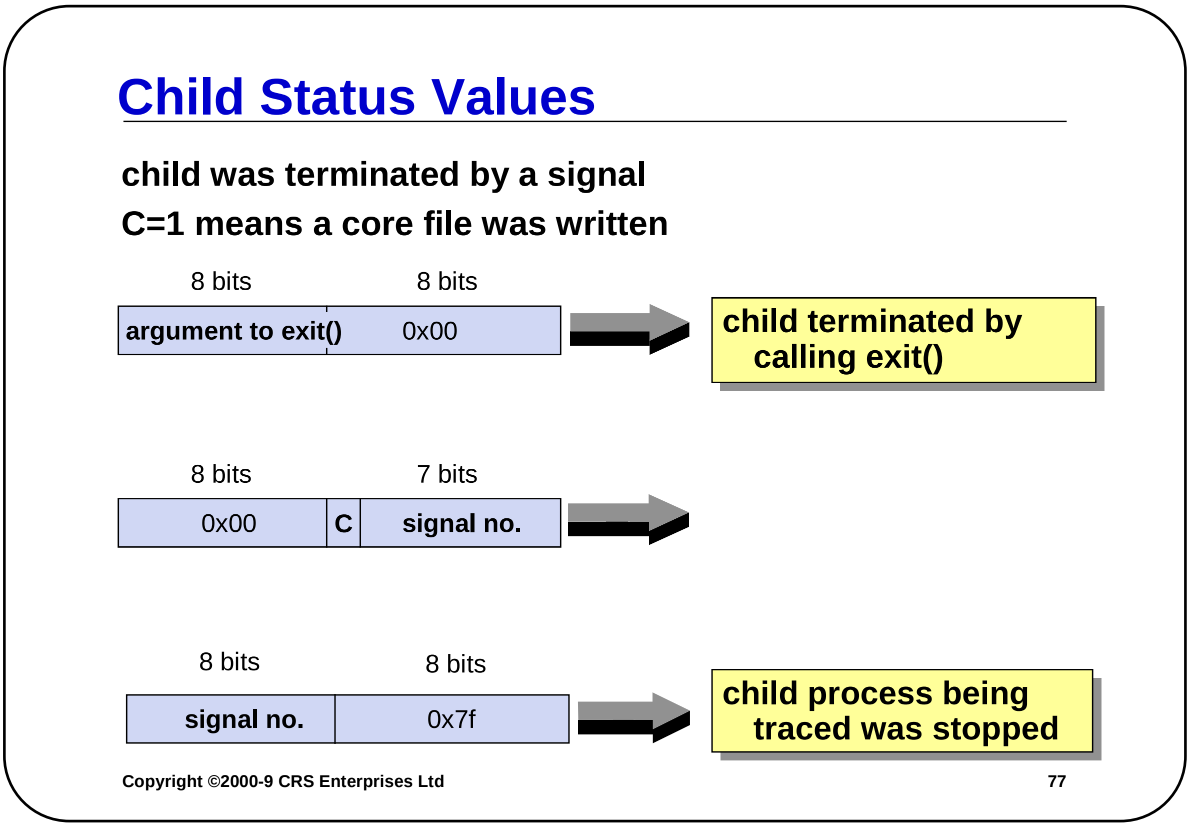
-1	any child
+pid	child pid
0	any child in callers group
-pid	any child in group pid

The wait() system call allows a process to wait for one of its child processes to finish execution and then examine the status of the child process to see the reason for its termination. If there are many child processes, there is no guarantee which will finish first. For this reason, wait() returns the pid of the child process that it has picked up. If there are child processes, but none has terminated, the calling process will be suspended until one of the child processes finishes.

The waitpid() system call allows a process to wait for specified child processes to finish execution rather than to wait for an arbitrary child to finish.

It is possible to examine the child's status on termination, via the status parameter from wait(). However, the exit status information is in an encoded form (see next slide on how to decode the information).

Note: When a child process terminates or is suspended, a SIGCLD is sent to its parent. A parent may then use wait() to determine which child has terminated. Obviously, wait() will return immediately in this situation.



Child Status Values

child was terminated by a signal

C=1 means a core file was written

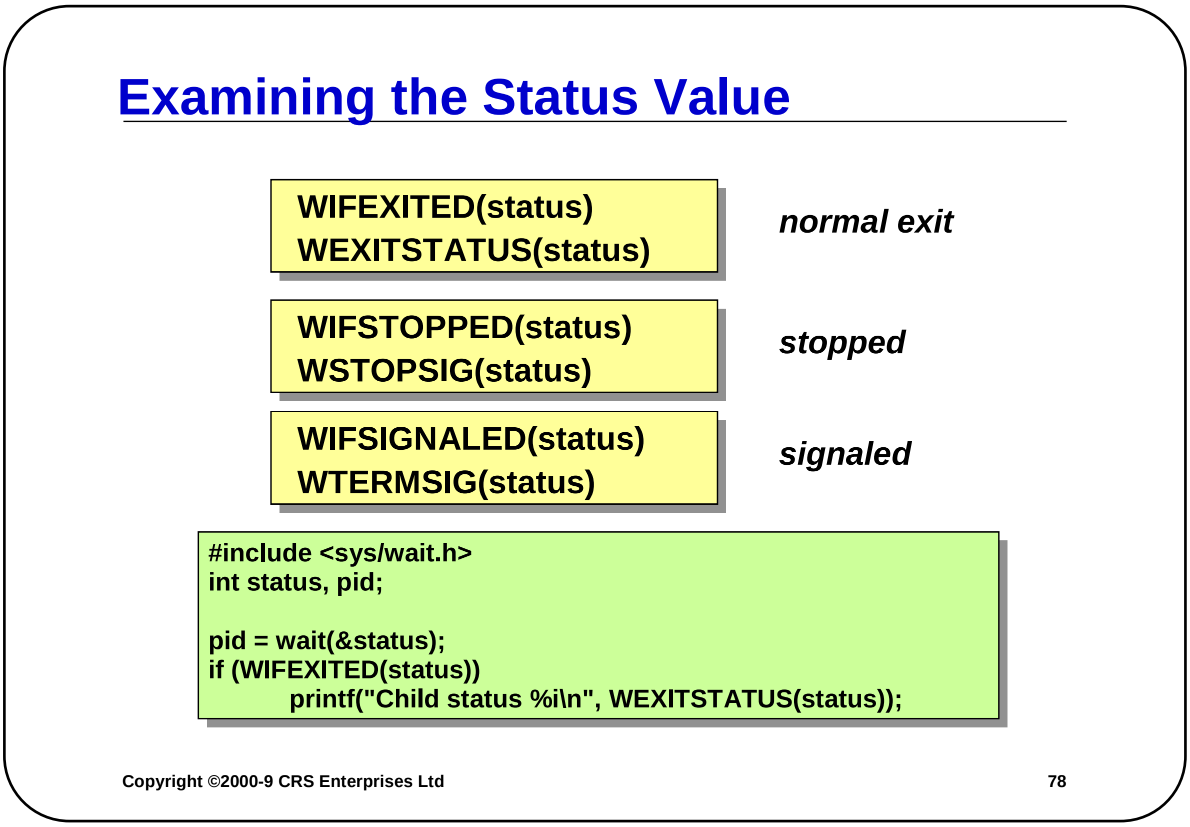
Copyright ©2000-9 CRS Enterprises Ltd

77

wait() and waitpid() return information about a child process when one of three circumstances occurs:

1. The child terminated by calling exit(). In this case, the low-order byte of the status word will be 0 and the high byte will contain the parameter that was passed to exit().
2. The child terminated on receipt of a signal. Here, the high byte will be 0 and the low 7 bits will contain the signal that terminated the process. If the C bit is set, then the process generated a core image on termination.
3. The child was being traced and stopped for some reason (usually on receipt of a signal). This situation occurs mostly when a debugger is being used to control the execution of a process. It is not common in normal use.

A set of macros is provided to simplify decoding of this information (see next slide).



Examining the Status Value

WIFEXITED(status)
WEXITSTATUS(status)

normal exit

WIFSTOPPED(status)
WSTOPSIG(status)

stopped

WIFSIGNALED(status)
WTERMSIG(status)

signaled

```
#include <sys/wait.h>
int status, pid;

pid = wait(&status);
if (WIFEXITED(status))
        printf("Child status %i\n", WEXITSTATUS(status));
```

Copyright ©2000-9 CRS Enterprises Ltd 78

The include file <sys/wait.h> defines a number of macros that are very useful for examining the status value returned by wait(). It is particularly good practice to use these macros, since they will be independent of any hardware-specific considerations of byte ordering and word size.

1. Normal exit

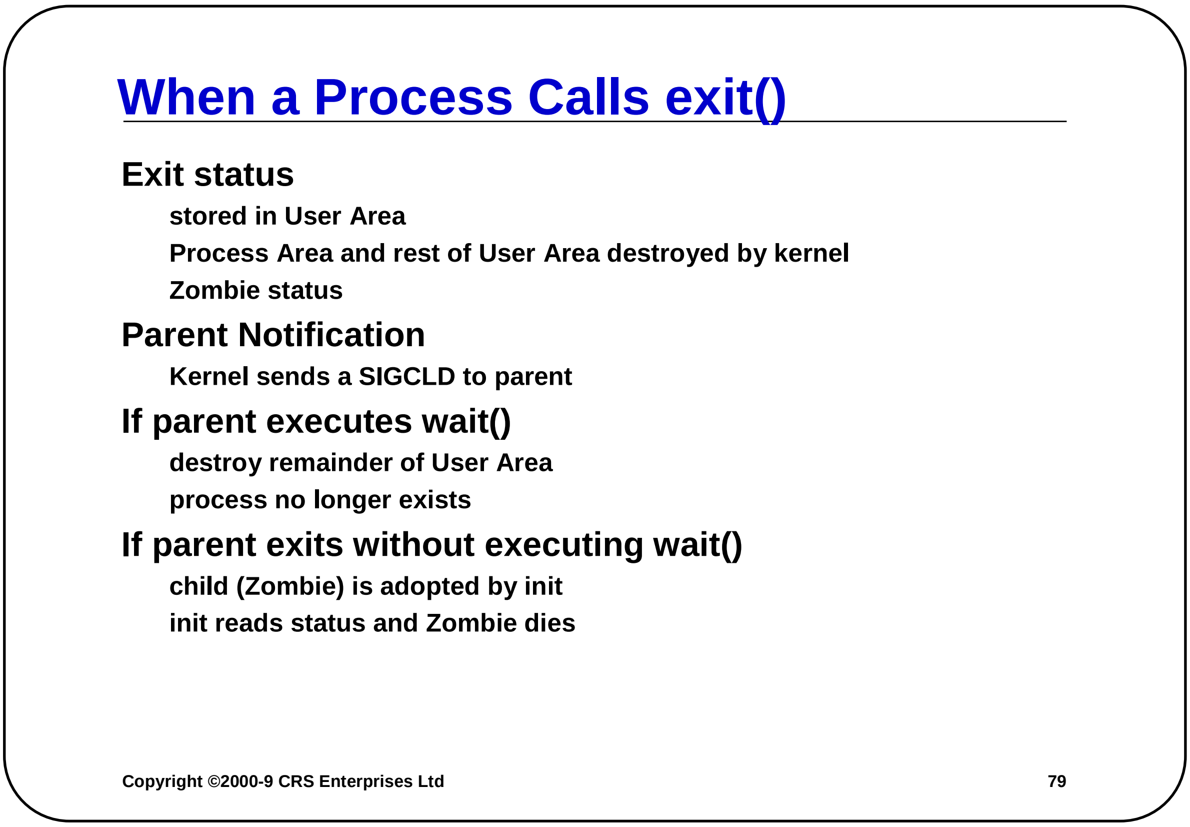
WIFEXITED(status)	true if the child process terminated by calling exit().
WEXITSTATUS(status)	"exit status" that the child passed to the exit() call.

2. Exit on receipt of a signal

WIFSIGNALED(status)	true if the child process terminated on receipt of a signal.
WTERMSIG(status)	number of the signal that caused the process to terminate.
WCOREDUMP(status)	true if a core image of the process was created.

3. Child stopped

WIFSTOPPED(status)	true if the child is a traced process that has stopped.
WSTOPSIG(status)	number of the signal that stopped the process.



When a Process Calls exit()

Exit status

- stored in User Area
- Process Area and rest of User Area destroyed by kernel
- Zombie status

Parent Notification

- Kernel sends a SIGCLD to parent

If parent executes wait()

- destroy remainder of User Area
- process no longer exists

If parent exits without executing wait()

- child (Zombie) is adopted by init
- init reads status and Zombie dies

We can now look in more detail at what happens when a process terminates with exit().

A parent process can call the wait() system call to find out information about its child processes when they terminate. Since the wait() call will suspend a process until a child terminates, this facility can be used as a primitive means of synchronization between parent and child processes.

The kernel sends the SIGCLD signal to a parent process when one of its children dies. SIGCLD is ignored by default, but is intended to inform a parent that it should execute a wait() call to pick up details about a child, allowing the child to be cleared from the system. Explicitly setting SIGCLD to be ignored lets the kernel know that a process is not interested in the exit status of its child processes and does not want to execute wait() for them. This prevents the child processes from becoming zombies.

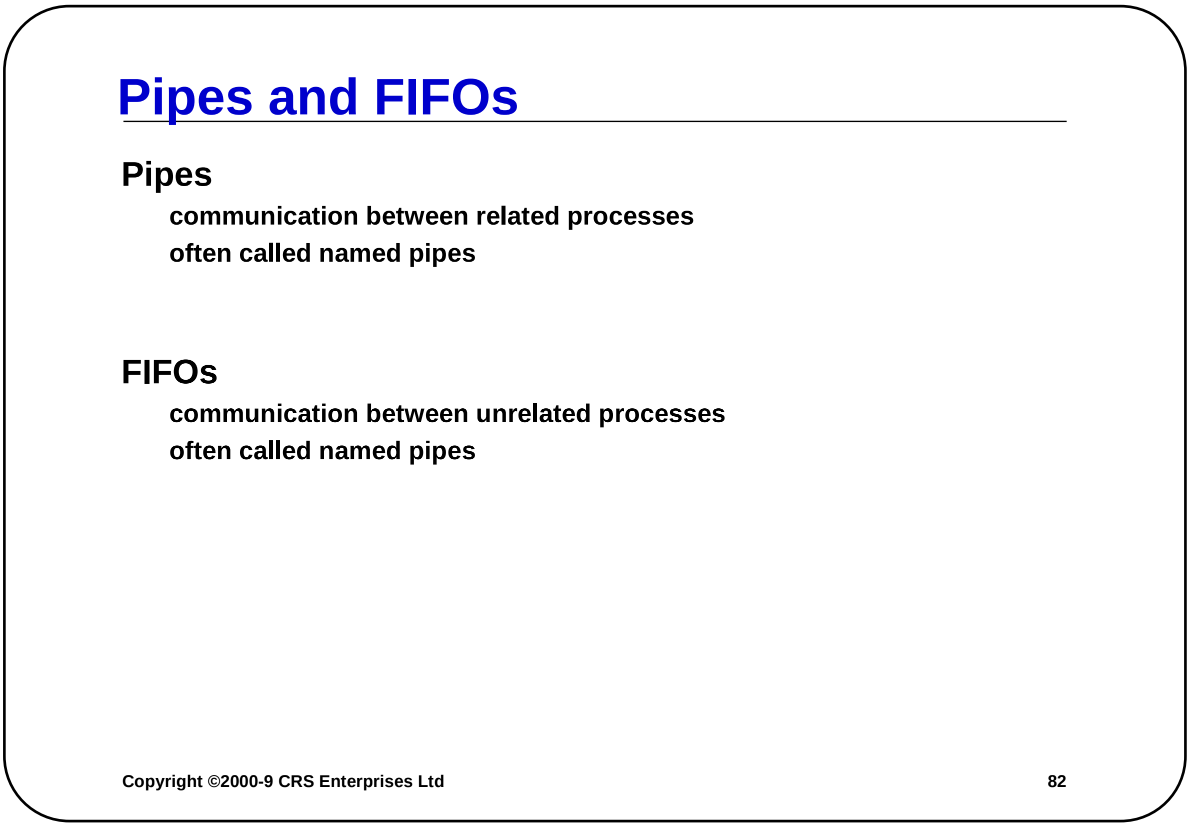
If the parent process has not yet executed a wait() call, the exiting process becomes a zombie process. In this state, all the resources (such as virtual memory) used by the process are freed, but the child process retains a small amount of kernel memory to store its exit status. This memory remains allocated until its parent executes wait().

If a parent process exits before its child processes have terminated, the parent pid of the child processes must be reset to some meaningful value. The init process (process 1) adopts all such orphan processes and reads their exit status as soon as possible. Thus init cleans up all zombie processes.

6

Pipes and FIFOs

6



Pipes and FIFOs

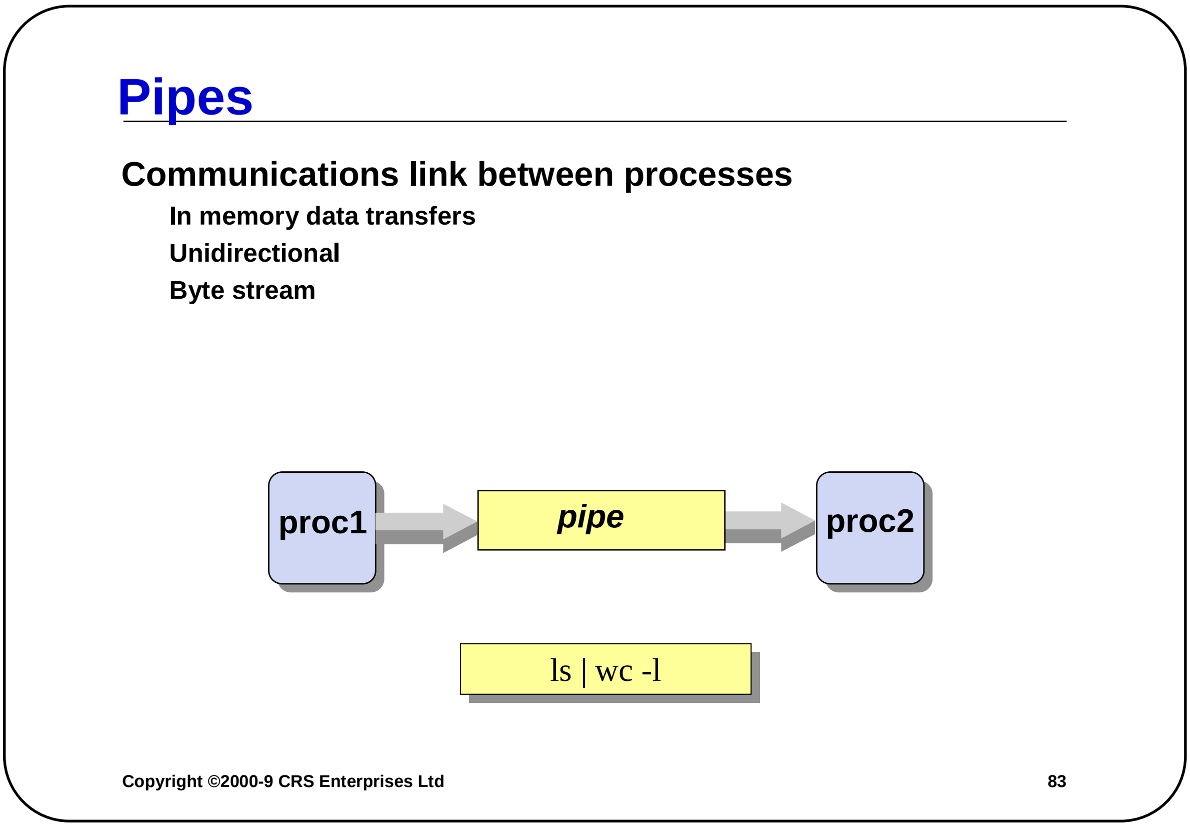
Pipes

- communication between related processes
- often called named pipes

FIFOs

- communication between unrelated processes
- often called named pipes

In this chapter we will investigate two of the oldest forms of inter-process communication in Unix: pipes and FIFOs (named pipes). Pipes can only be used in a parent/child process arrangement, whereas FIFOs can be used to communicate between any two processes.



Pipes

Communications link between processes

- In memory data transfers
- Unidirectional
- Byte stream

ls | wc -l

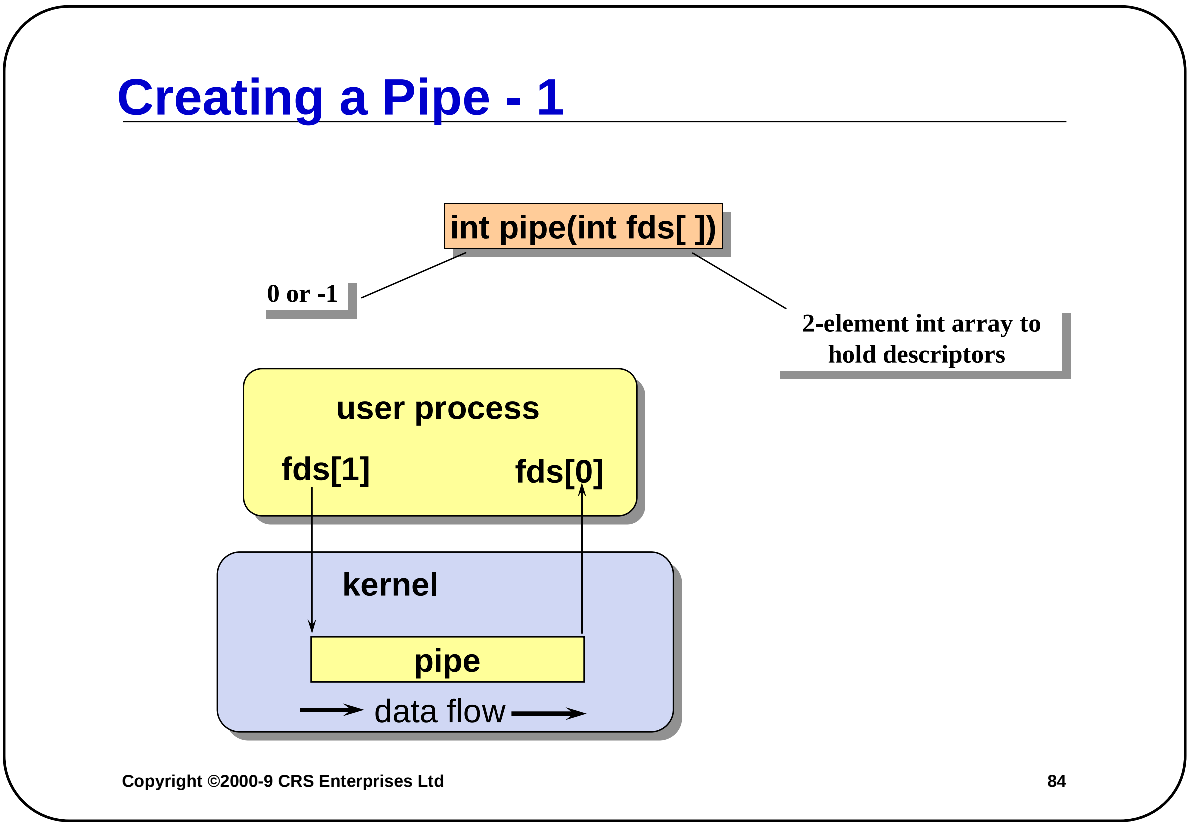
Pipes were the original method of inter-process communication in Unix. They are available in all versions.

A pipe provides a unidirectional link for one process to send information to another. The information is sent as a byte stream, similar to the idea of files being streams of bytes. There are no record boundaries in the data being sent through a pipe.

A pipe appears to a process as another I/O channel, accessed through a descriptor in the same way as a file or device. Random access is not allowed and data is accessed strictly on a first-in first-out basis; once read, data is removed from the pipe.

Pipes are often used on the command line. In the next few slides, we will investigate how the shell implements the pipeline:

```
ls | wc -l
```



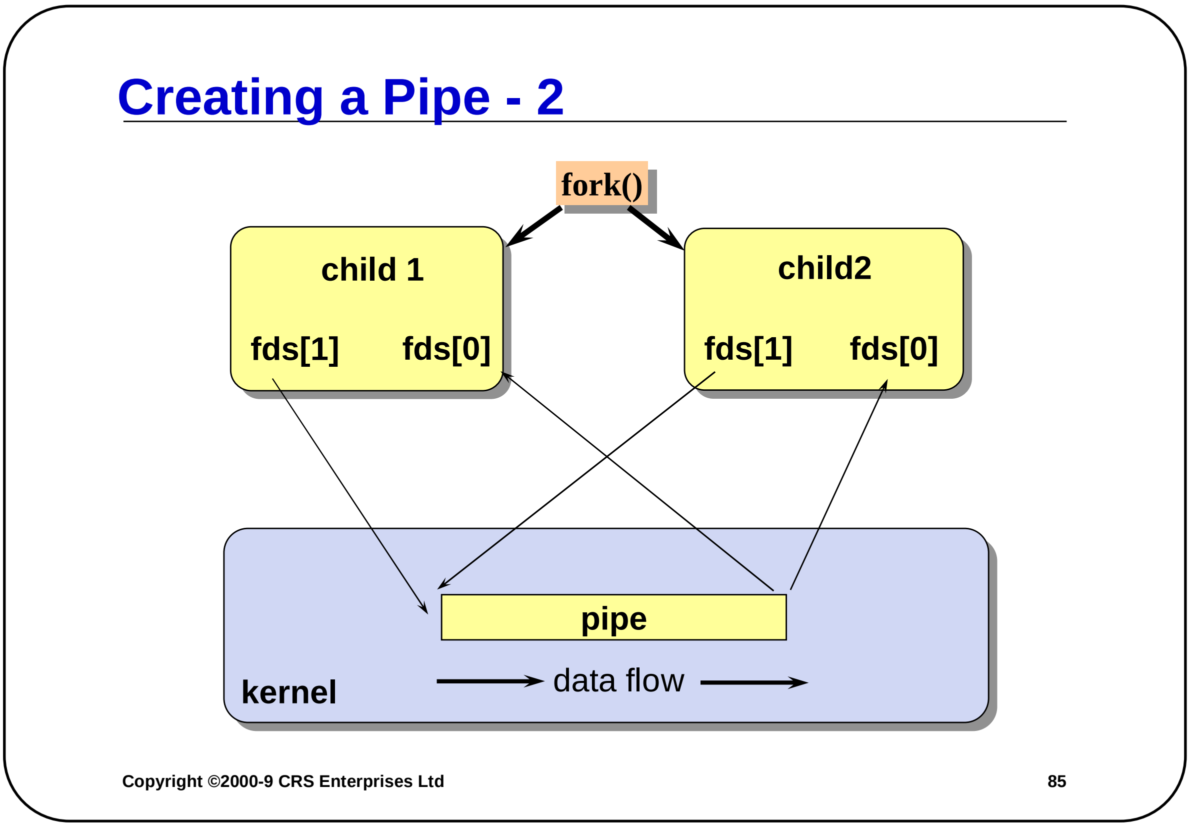
Creating a Pipe - 1

Copyright ©2000-9 CRS Enterprises Ltd 84

There are several stages in the procedure of creating a pipe between two processes. The first is to call pipe() which creates the pipe kernel data structures. The parameter fds is a 2-element integer array, which is intended to hold a pair of file descriptors.

If pipe() is successful, fds[0] will be set to a descriptor that can be used to read from the pipe and fds[1] to a descriptor that can be used to write to the pipe.

At this stage, we are still within the confines of a single process. If any data is written to the pipe on fds[1], the process can immediately read it on fds[0]. We therefore have a form of loopback. Clearly, pipes of this form are of limited use.



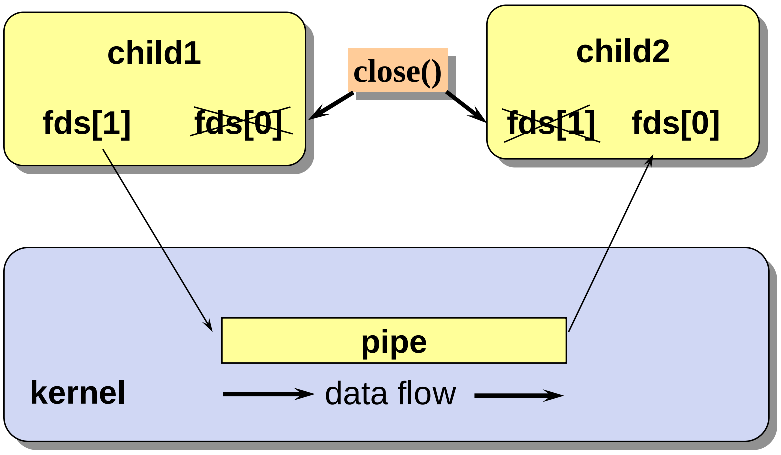
Creating a Pipe - 2

Copyright ©2000-9 CRS Enterprises Ltd 85

The next stage in setting up a pipe between processes is for the process to execute a fork() call.

As a result of the way in which process attributes are inherited across fork(), the child and parent processes will both inherit the descriptors that refer to the pipe. The pipe itself is not duplicated; only the descriptors that provide access to it. Now we have two child processes with access to the read and write file descriptors of the pipe.

Creating a Pipe - 3


child1

fds[1] ~~fds[0]~~

close()

child2

~~fds[1]~~ fds[0]

pipe

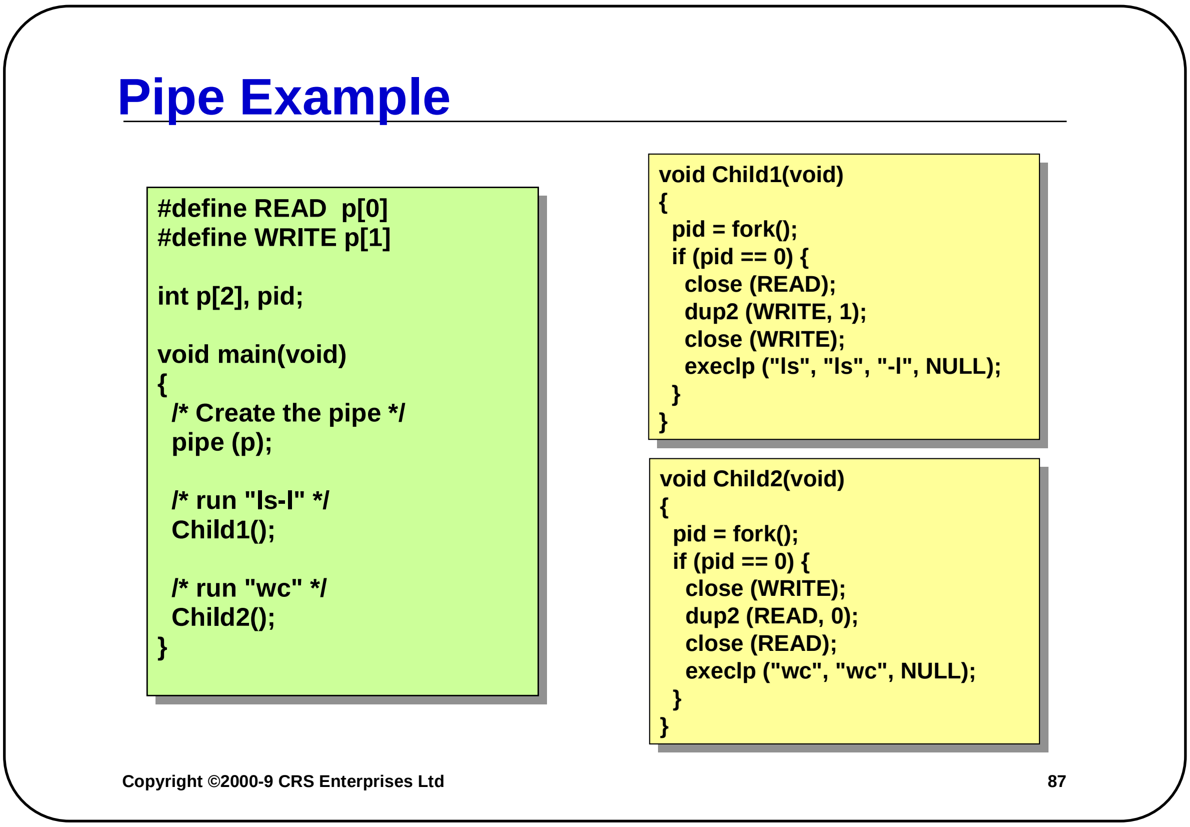
kernel

data flow

86

Pipes do not support duplex (bidirectional) communications. Therefore, the final stage of the pipe creation procedure is to close the write descriptor in the reader process and close the read descriptor in the writer process.

This leaves the situation as shown, with a single pipe connecting the two processes; one reading from the pipe and the other writing to it. The processes may now use read() and write() as though the descriptor referred to normal files.



Pipe Example

```
#define READ  p[0]
#define WRITE p[1]

int p[2], pid;

void main(void)
{
  /* Create the pipe */
  pipe (p);

  /* run "ls-l" */
  Child1();

  /* run "wc" */
  Child2();
}
```

```
void Child1(void)
{
  pid = fork();
  if (pid == 0) {
    close (READ);
    dup2 (WRITE, 1);
    close (WRITE);
    execlp ("ls", "ls", "-l", NULL);
  }
}
```

```
void Child2(void)
{
  pid = fork();
  if (pid == 0) {
    close (WRITE);
    dup2 (READ, 0);
    close (READ);
    execlp ("wc", "wc", NULL);
  }
}
```

Copyright ©2000-9 CRS Enterprises Ltd 87

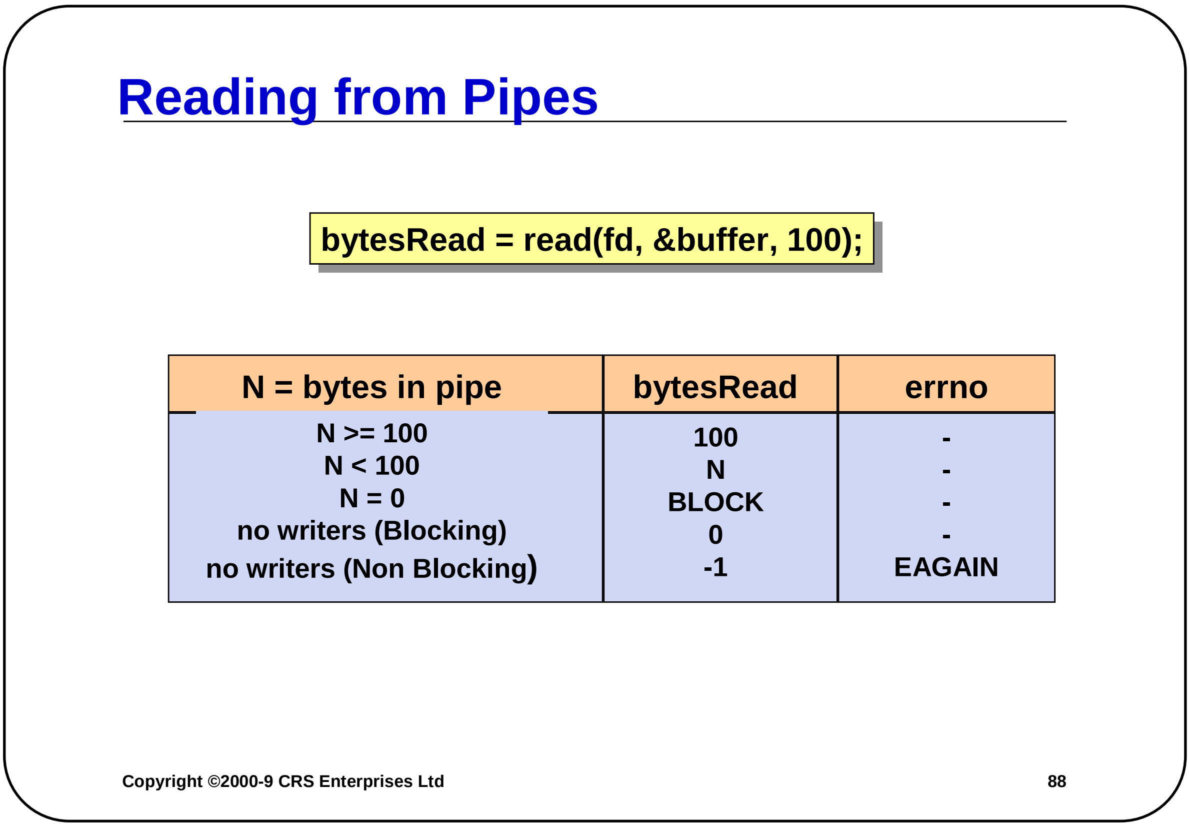
This example shows how a pipe can be created between two child processes. The example simulates the shell pipeline:

```
ls -l | wc
```

The call to the pipe() system call should be made before the fork() to create the child processes. The descriptors created by the pipe() will be inherited by both child processes, and all that remains is for each process to close the descriptor that corresponds to the end of the pipe that is not to be used. The parent also has both ends of the pipe open, but since it does not use the pipe this does no harm. In fact keeping the descriptors open in the parent can sometimes be an advantage. This removes the possibility of an error condition arising where there are no writer processes or no reader processes (see later).

Thereafter, either process may use read() or write() on the descriptor as though it refers to a regular file.

Notice the use of dup2() to duplicate standard input and output in the child processes. It is assumed that the program will be executed from a shell and therefore descriptors 0 (stdin), 1 (stdout) and 2 (stderr) will be inherited by each process.



Reading from Pipes

```
bytesRead = read(fd, &buffer, 100);
```

N = bytes in pipe	bytesRead	errno
N >= 100	100	-
N < 100	N	-
N = 0	BLOCK	-
no writers (Blocking)	0	-
no writers (Non Blocking)	-1	EAGAIN

Data is read from pipes using the read() system call. Certain special situations may occur:

If the pipe contains more than the requested number of bytes, the required number is returned. This is as we would expect and is analogous to read() from a file.

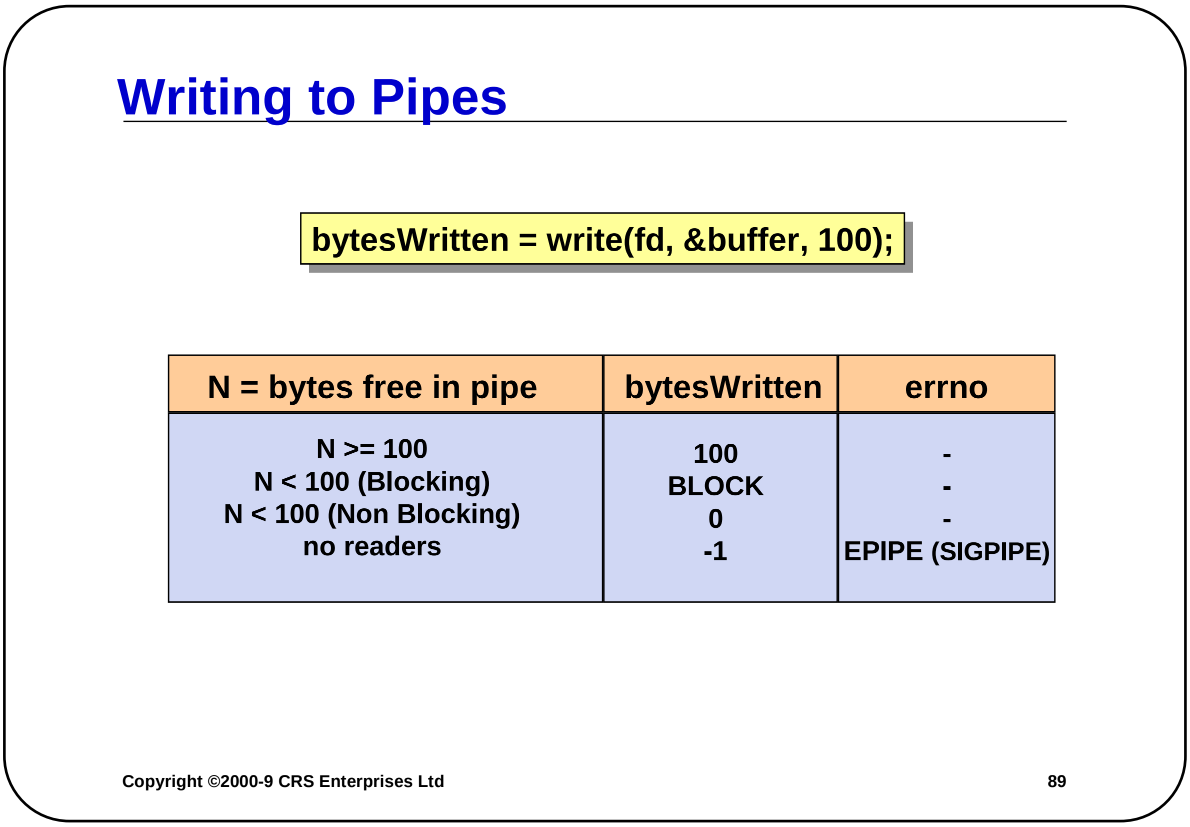
If the pipe contains fewer than the requested number of bytes, read() will return with the number of bytes in the pipe.

If the pipe contains no data, then one of several situations may occur:

The reading process will normally wait until data appears in the pipe, then read() will return that data as above. If the pipe is "disconnected", i.e. there are no processes writing to the pipe, then read() will return 0 (signifying EOF).

If the O_NDELAY flag is set on the pipe, read() will return 0 straight away. It is not possible to tell whether there are no writer processes or whether the pipe is simply empty.

The O_NONBLOCK flag may be set as a way of specifying non-blocking I/O. For an empty pipe read() will return -1 and errno will be set to EAGAIN.



Writing to Pipes

```
bytesWritten = write(fd, &buffer, 100);
```

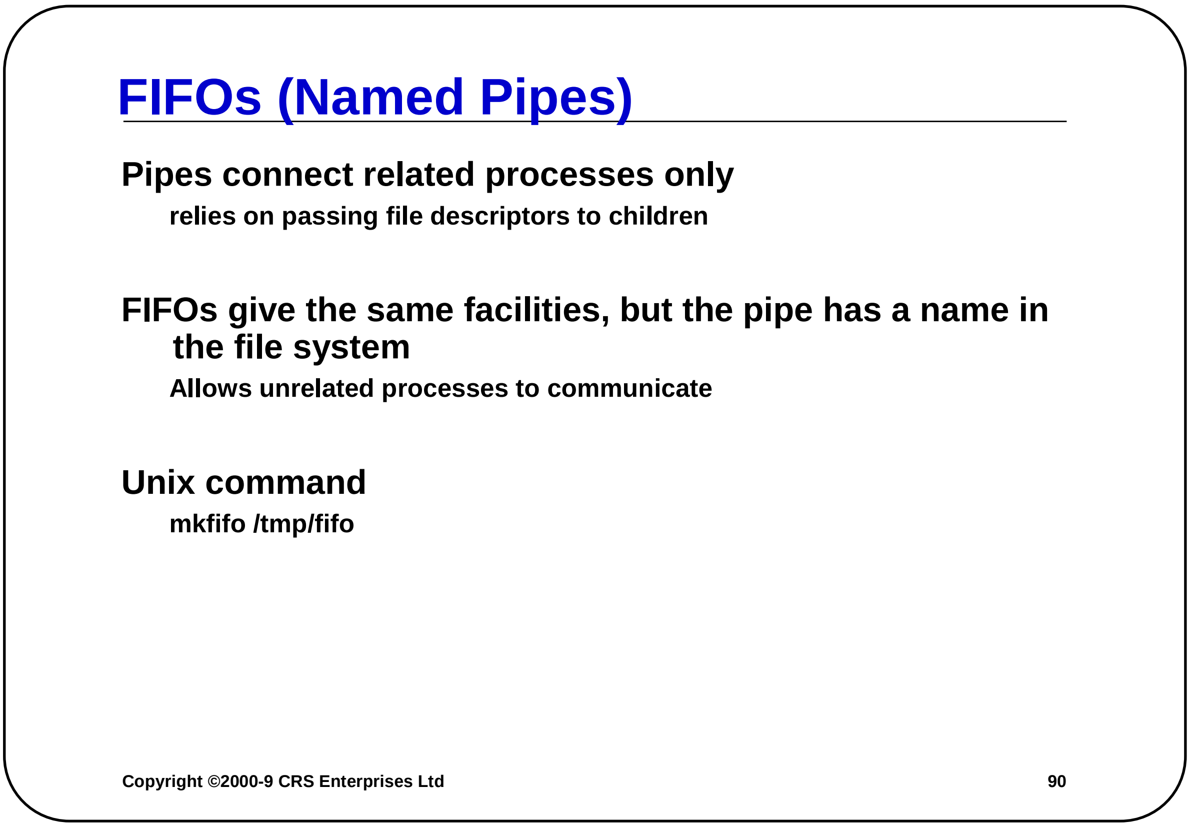
N = bytes free in pipe	bytesWritten	errno
N >= 100	100	-
N < 100 (Blocking)	BLOCK	-
N < 100 (Non Blocking)	0	-
no readers	-1	EPIPE (SIGPIPE)

89

The write() system call is used to put data into a pipe. As with reading from pipes, certain special situations may occur:

> If the amount of data written is less than the size of the pipe (PIPE_BUF), the write() is guaranteed to be atomic, i.e. the data will be written in a contiguous piece, regardless of how many processes are writing to the pipe.
>
> If the amount of data is larger than the pipe size, the write() is not guaranteed to be atomic. Normal calls to write() will return the number of bytes transferred, as for files.
>
> If the pipe is full, the write() call will block until there is enough space to transfer the data.
>
> If O_NDELAY is set, the write() will return the value of 0 immediately.

If there are no processes currently reading the pipe, the write() will fail, with error EPIPE. The signal SIGPIPE will be sent to the process. SIGPIPE has termination as its default action.



FIFOs (Named Pipes)

Pipes connect related processes only

- **relies on passing file descriptors to children**

FIFOs give the same facilities, but the pipe has a name in the file system

- **Allows unrelated processes to communicate**

Unix command

- **mkfifo /tmp/fifo**

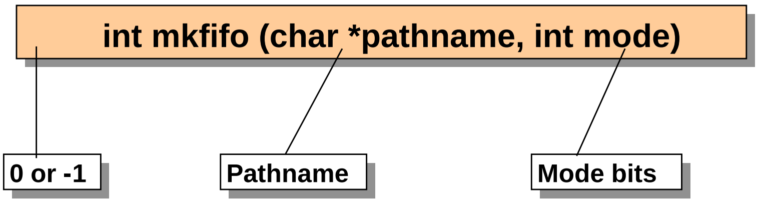
90

Pipes can be very useful, but processes must be related to use them. An alternative to pipes (unnamed pipes) SVR4 defines FIFOs (named pipes). FIFOs allow processes which are not related to pass data in the same way as pipes.

A FIFO has a name in the file system just like any regular file. It can be accessed and protected like a file, but the data-transmission rules are the same as those for unnamed pipes.

Creating a FIFO

```
mkfifo("/tmp/fifo",  0666);
```



int mkfifo (char *pathname, int mode)

0 or -1

Pathname

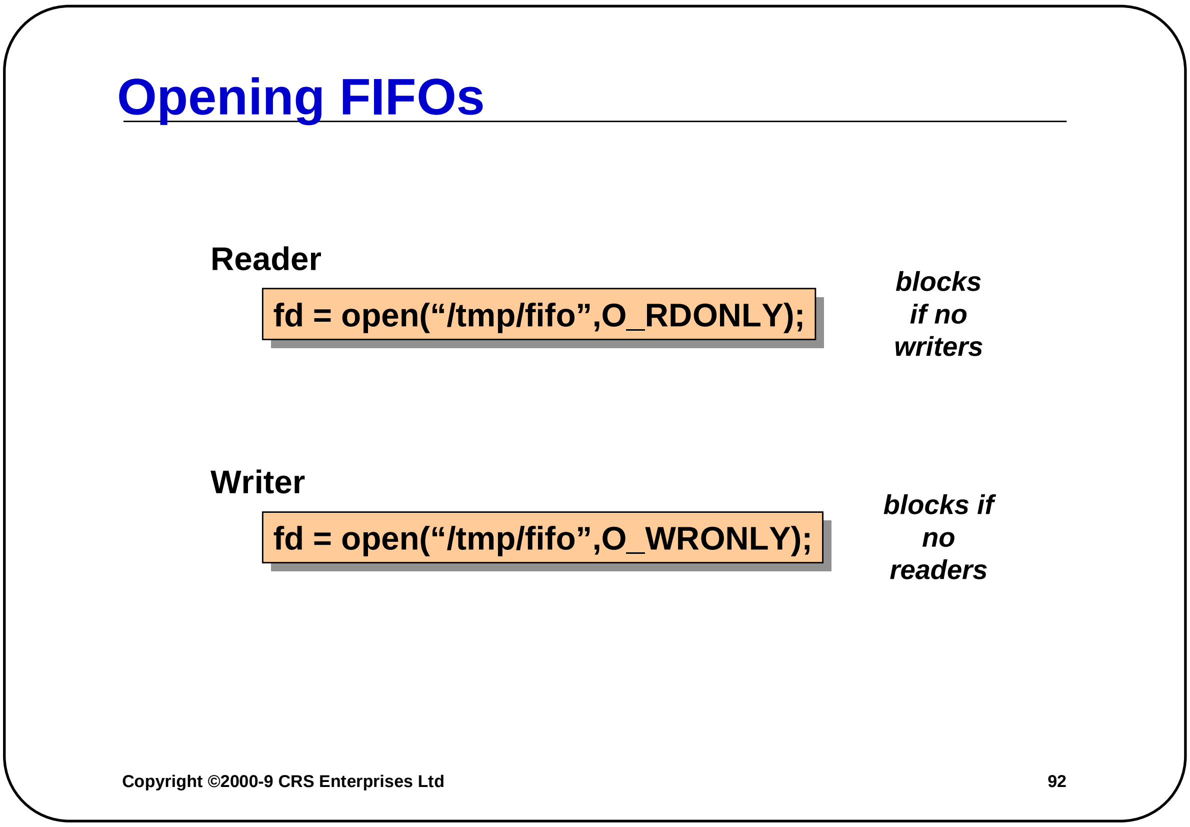
Mode bits

Named pipes are created using the mkfifo() system call:

```
mkfifo ( char *pathname, int mode );
```

The mkfifo command can also be used to create a FIFO:

```
/etc/mkfifo pathname
```



Opening FIFOs

Reader

```
fd = open(“/tmp/fifo”,O_RDONLY);
```

blocks if no writers

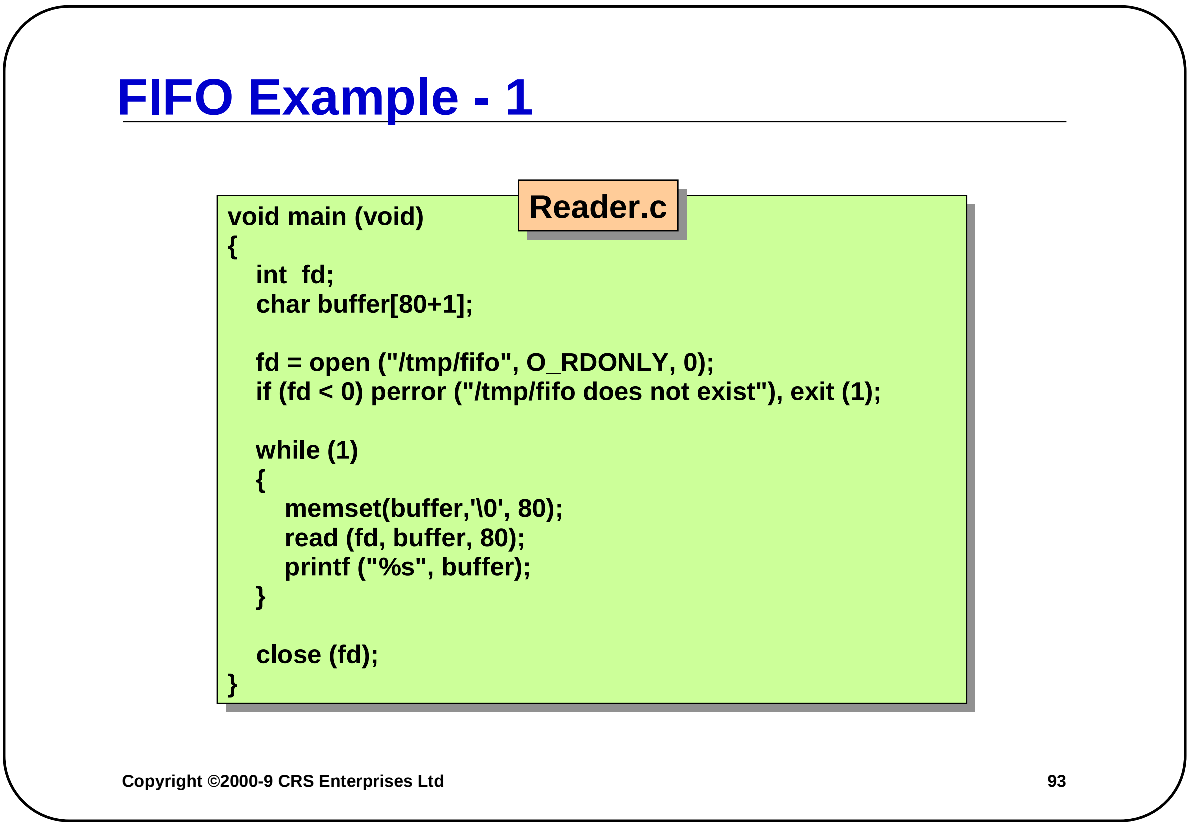
Writer

```
fd = open(“/tmp/fifo”,O_WRONLY);
```

blocks if no readers

When a process opens a FIFO for reading or writing the open() call will block until a connection is made with another process. Once the connection is established, reading and writing from FIFOs follows the same rules as for pipes.

If the writer process closes the FIFO, its partner will get 0 returned from every subsequent read() system call (blocking mode) unless another process reopens the FIFO. In such situations you can either poll the FIFO for activity (non zero return from read()) or close and then reopen the FIFO. If you reopen the FIFO you will block until a new connection is established.



FIFO Example - 1

Reader.c

```
void main (void)
{
   int  fd;
   char buffer[80+1];

   fd = open ("/tmp/fifo", O_RDONLY, 0);
   if (fd < 0) perror ("/tmp/fifo does not exist"), exit (1);

   while (1)
   {
      memset(buffer,'\0', 80);
      read (fd, buffer, 80);
      printf ("%s", buffer);
   }

   close (fd);
}
```

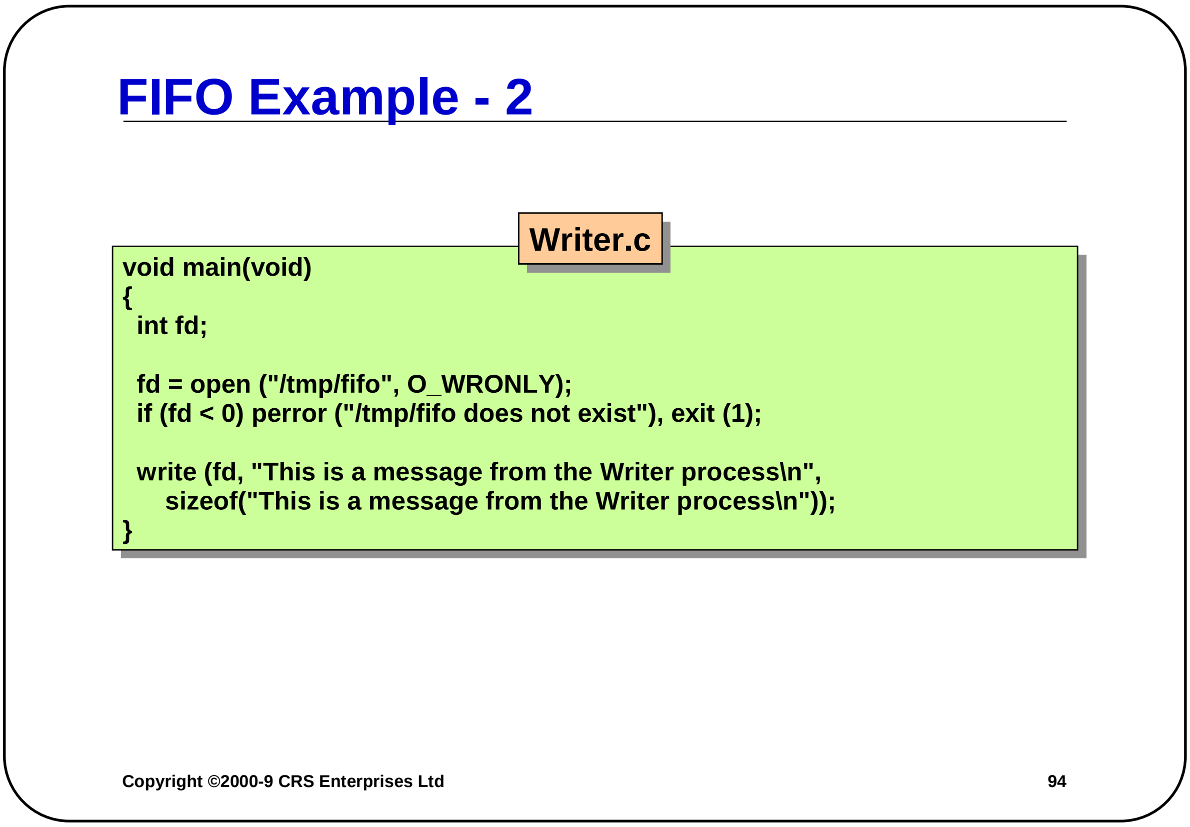
The example code above and that on the next slide show how to use a named pipe or FIFO between two unrelated processes.

This example assumes that the FIFO has already been created using the mkfifo command:

```
mkfifo /tmp/fifo
```

before the reader and writer process are executed.

The reader process will block as soon as it reaches the open() system call, because there are no writer processes using the FIFO.



FIFO Example - 2

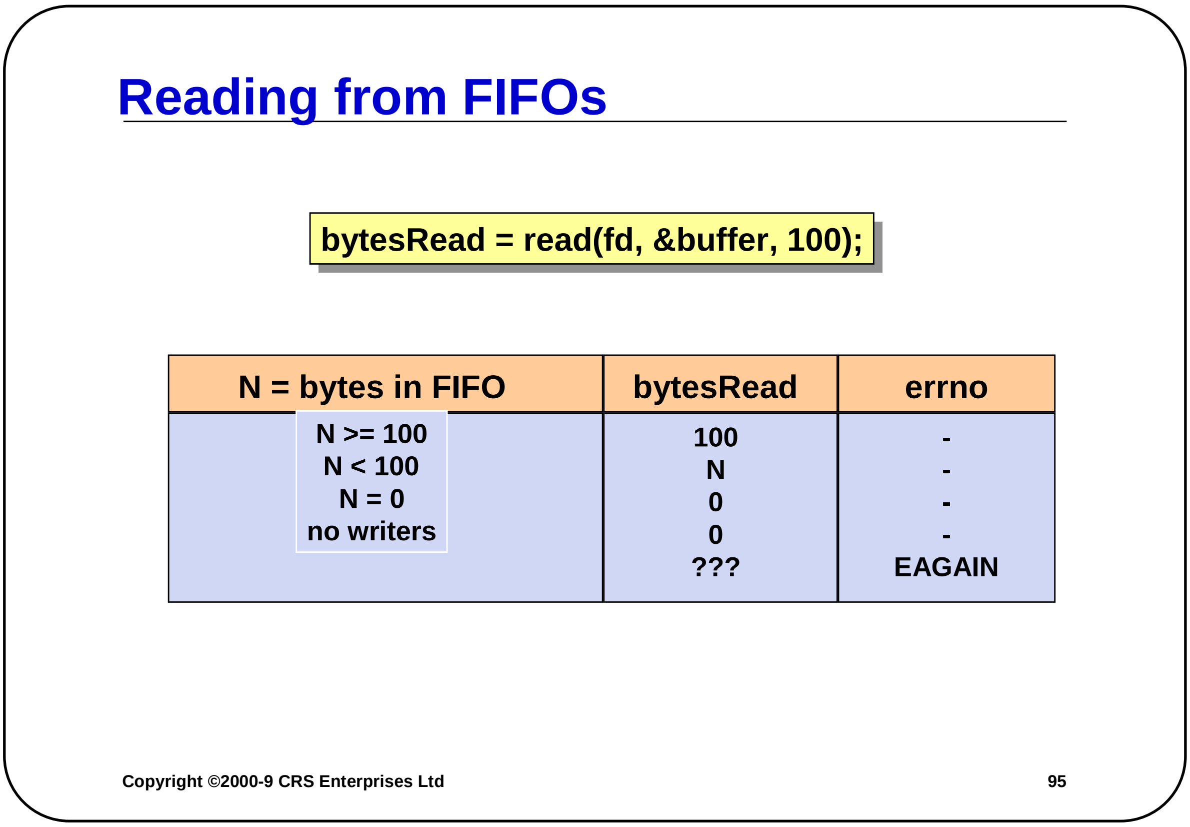
Writer.c

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

  fd = open ("/tmp/fifo", O_WRONLY);
  if (fd < 0) perror ("/tmp/fifo does not exist"), exit (1);

  write (fd, "This is a message from the Writer process\n",
      sizeof("This is a message from the Writer process\n"));
}
```

94

Once the reader process has started the writer can be executed. As soon as the writer opens the FIFO, the reader will unblock from its open() call and then block on its read() call.

The writer will write data to the FIFO and then exit. The reader will then unblock from its read() system call and extract the data from the FIFO.



Reading from FIFOs

```
bytesRead = read(fd, &buffer, 100);
```

N = bytes in FIFO	bytesRead	errno
N >= 100	100	-
N < 100	N	-
N = 0	0	-
no writers	0	-
	???	EAGAIN

Copyright ©2000-9 CRS Enterprises Ltd 95

Data is read from pipes using the read() system call. Certain special situations may occur:

If the pipe contains more than the requested number of bytes, the required number is returned. This is as we would expect and is analogous to read() from a file.

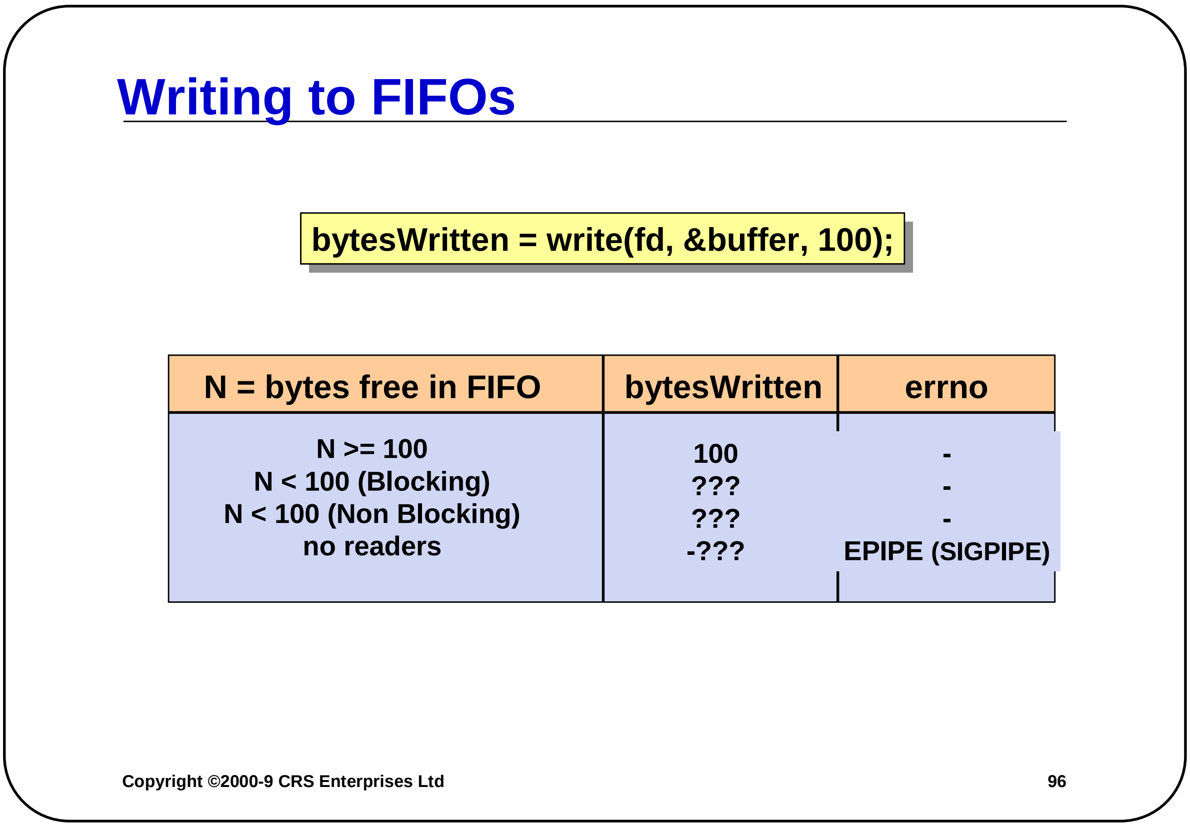
If the pipe contains fewer than the requested number of bytes, read() will return with the number of bytes in the pipe.

If the pipe contains no data, then one of several situations may occur:

The reading process will normally wait until data appears in the pipe, then read() will return that data as above. If the pipe is "disconnected", i.e. there are no processes writing to the pipe, then read() will return 0 (signifying EOF).

If the O_NDELAY flag is set on the pipe, read() will return 0 straight away. It is not possible to tell whether there are no writer processes or whether the pipe is simply empty.

In POSIX systems, the flag O_NONBLOCK may be set as a way of specifying non-blocking I/O. If so, then read() will return -1 and errno will be set to EAGAIN.



Writing to FIFOs

```
bytesWritten = write(fd, &buffer, 100);
```

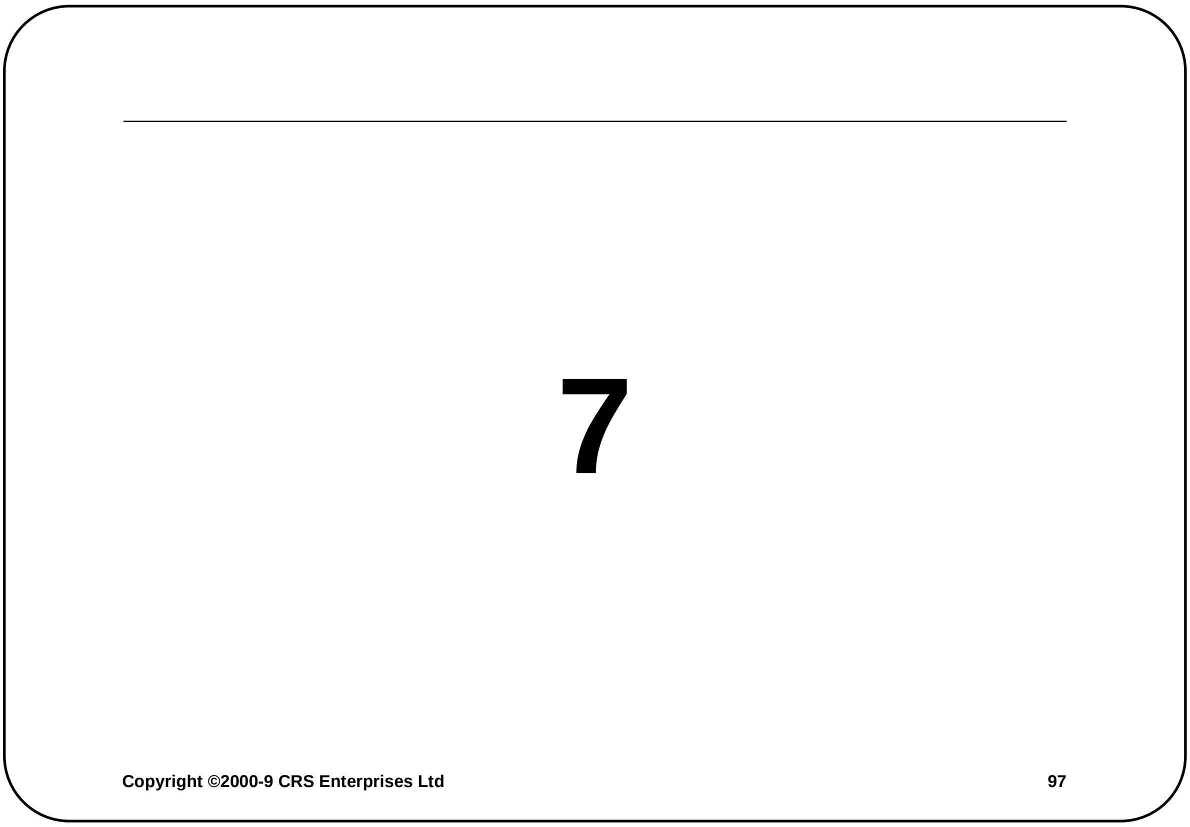
N = bytes free in FIFO	bytesWritten	errno
N >= 100	100	-
N < 100 (Blocking)	???	-
N < 100 (Non Blocking)	???	-
no readers	-???	EPIPE (SIGPIPE)

Copyright ©2000-9 CRS Enterprises Ltd 96

The write() system call is used to put data into a pipe. As with reading from pipes, certain special situations may occur:

> If the amount of data written is less than the size of the pipe (PIPE_BUF), the write() is guaranteed to be atomic, i.e. the data will be written in a contiguous piece, regardless of how many processes are writing to the pipe. If the amount of data is larger than the pipe size, the write() is not guaranteed to be atomic. Normal calls to write() will return the number of bytes transferred, as for files.
>
> If the pipe is full, the write() call will block until there is enough space to transfer the data. If O_NDELAY is set, the write() will return the value of 0 immediately.

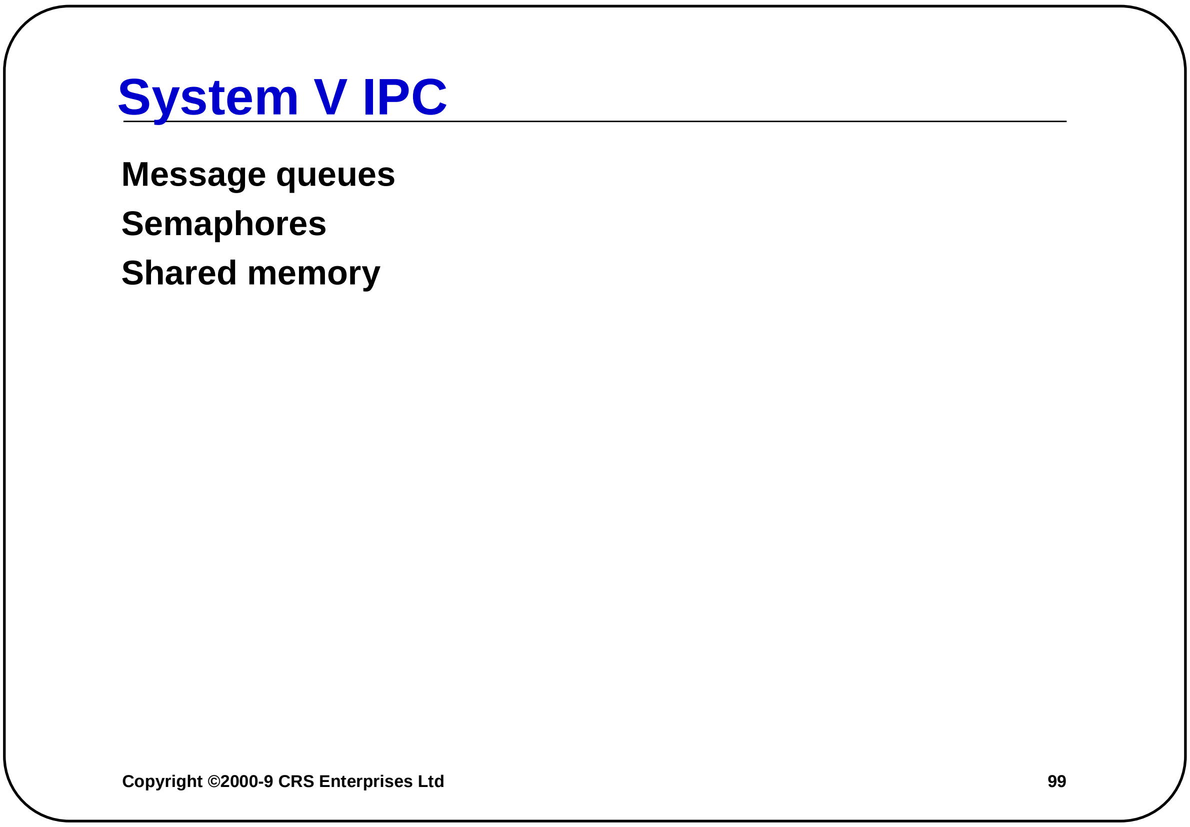
If there are no processes currently reading the pipe, the write() will fail, with error EPIPE. The signal SIGPIPE will be sent to the process. SIGPIPE has termination as its default action. The write() call returns only if the process ignores or catches SIGPIPE, then returns from the signal handler.



7

System V IPC

7



System V IPC

Message queues

Semaphores

Shared memory

99

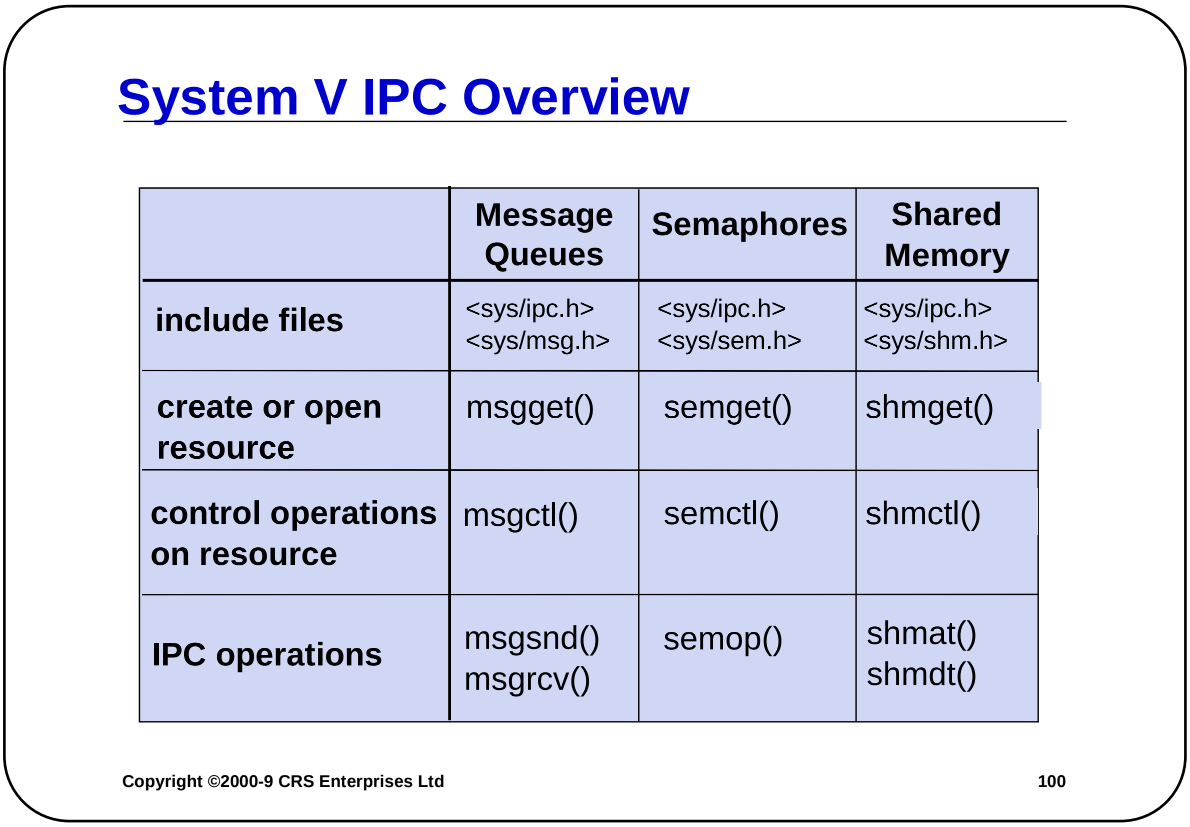
Three forms of IPC were developed by AT&T in the early 1980s and have now become a standard part of SR4:

Message queues

Semaphores

Shared memory

Although now available in most versions of Unix, they still tend to be known as System V IPC. This section will give an overview of these mechanisms.



System V IPC Overview

	Message Queues	Semaphores	Shared Memory
include files	<sys/ipc.h> <sys/msg.h>	<sys/ipc.h> <sys/sem.h>	<sys/ipc.h> <sys/shm.h>
create or open resource	msgget()	semget()	shmget()
control operations on resource	msgctl()	semctl()	shmctl()

IPC operations	msgsnd()
	msgrcv()
		semop()	shmat()
			shmdt()

This table shows the basic facilities available with the System V IPC mechanisms and how to access them. There are a number of common factors linking all three areas:

All require the include file <sys/ipc.h> to be included.

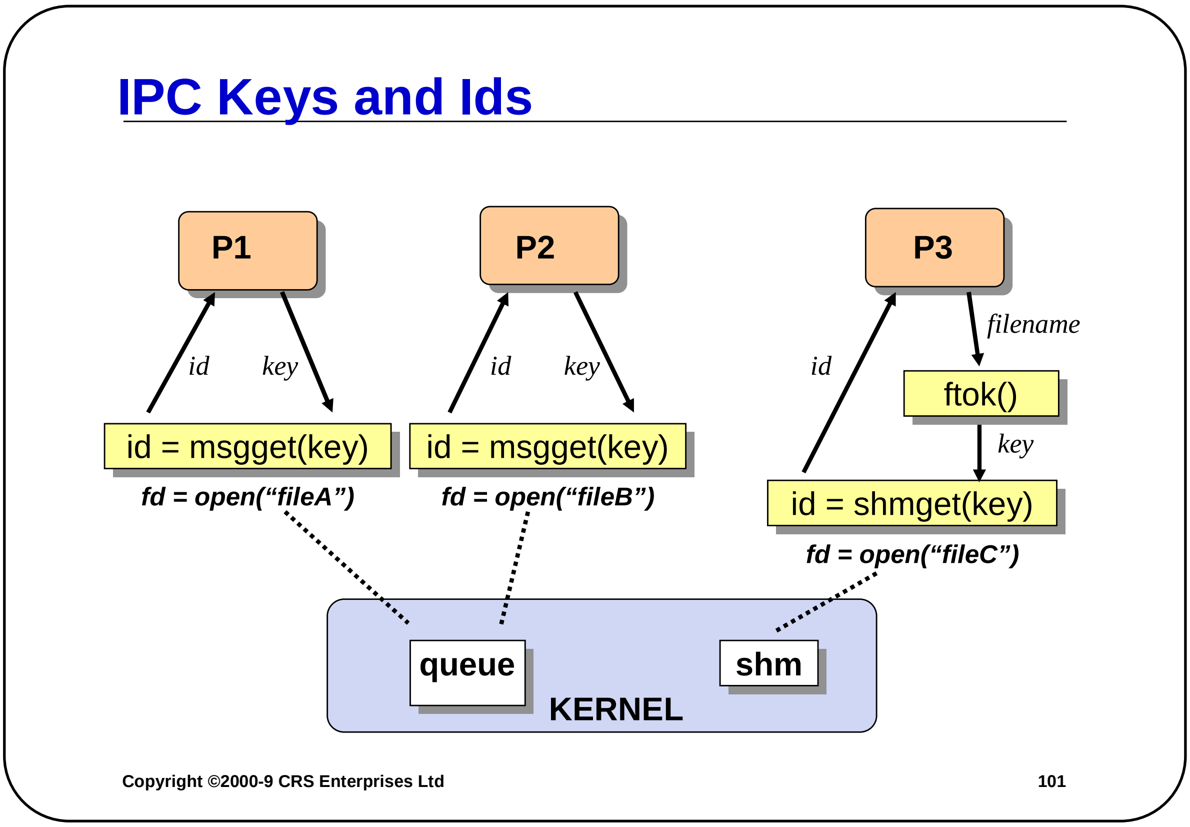
The "create/open" functionality is achieved using routines called "...get()".

Control functions are applied using routines called "...ctl()".

Message queues do not use the usual read() and write() system calls to perform I/O. The special msgrcv() and msgsnd() system calls are used instead.

Semaphores can be modified using the semop() system call. Use semctl() to read the value of a semaphore.

Shared memory performs I/O by direct memory access provided a kernel page has been attached to the process address space by shmat().



IPC Keys and Ids

101

System V IPC resources are identified by integer key values. Keys are of the type key_t as defined in <sys/types.h>. In System V IPC, the key is used analogously to how the pathname is used to identify a named pipe; it provides a means whereby unconnected processes can "rendezvous" on an IPC resource; the key can be considered as an IPC filename.

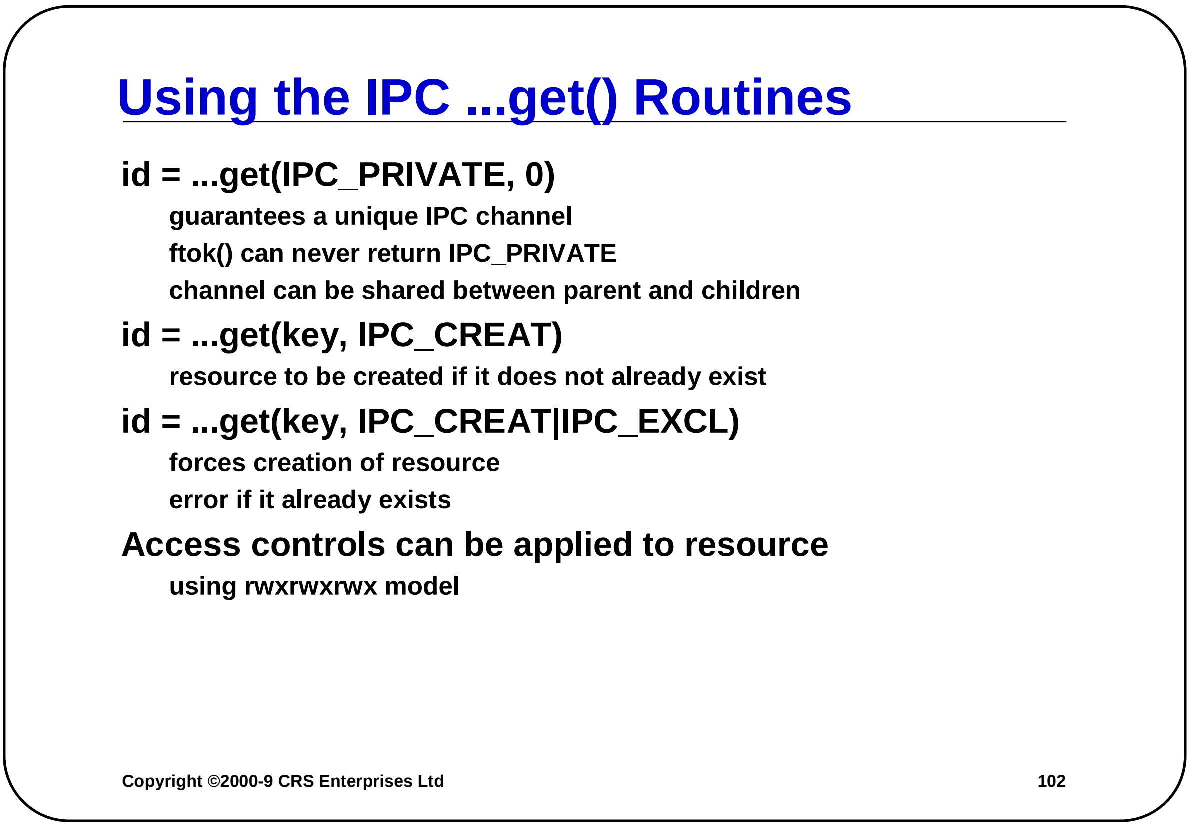
For example, two processes wishing to share a message queue would each call the msgget() routine to access a queue; using the same key value will ensure that both processes get access to the same queue.

After successful calls to the ...get() routines, the IPC resources are identified by integer id values. This is again similar to files, which are accessed through file descriptors after having been opened. It is important to remember that IPC descriptors and file descriptors are distinct from each other, and it is not possible to use the standard Unix I/O routines such as open(), read() and write() with IPC structures.

To generate an (almost) unique key use ftok()

```
key_t ftok (char *pathname, char proj)
```

ftok() will return a key value based on a "hashing" of its two parameters (the integer inode for the pathname and the integer ASCII code for proj). Processes wishing to use the key value must agree on the parameters beforehand, in much the same way as they agree on a key value.



Using the IPC ...get() Routines

id = ...get(IPC_PRIVATE, 0)

- guarantees a unique IPC channel
- ftok() can never return IPC_PRIVATE
- channel can be shared between parent and children

id = ...get(key, IPC_CREAT)

- resource to be created if it does not already exist

id = ...get(key, IPC_CREAT|IPC_EXCL)

- forces creation of resource
- error if it already exists

Access controls can be applied to resource

- using rwxrwxrwx model

102

The details of the individual routines will be covered at a later point in the course. There are, however, a number of common features across all such calls.

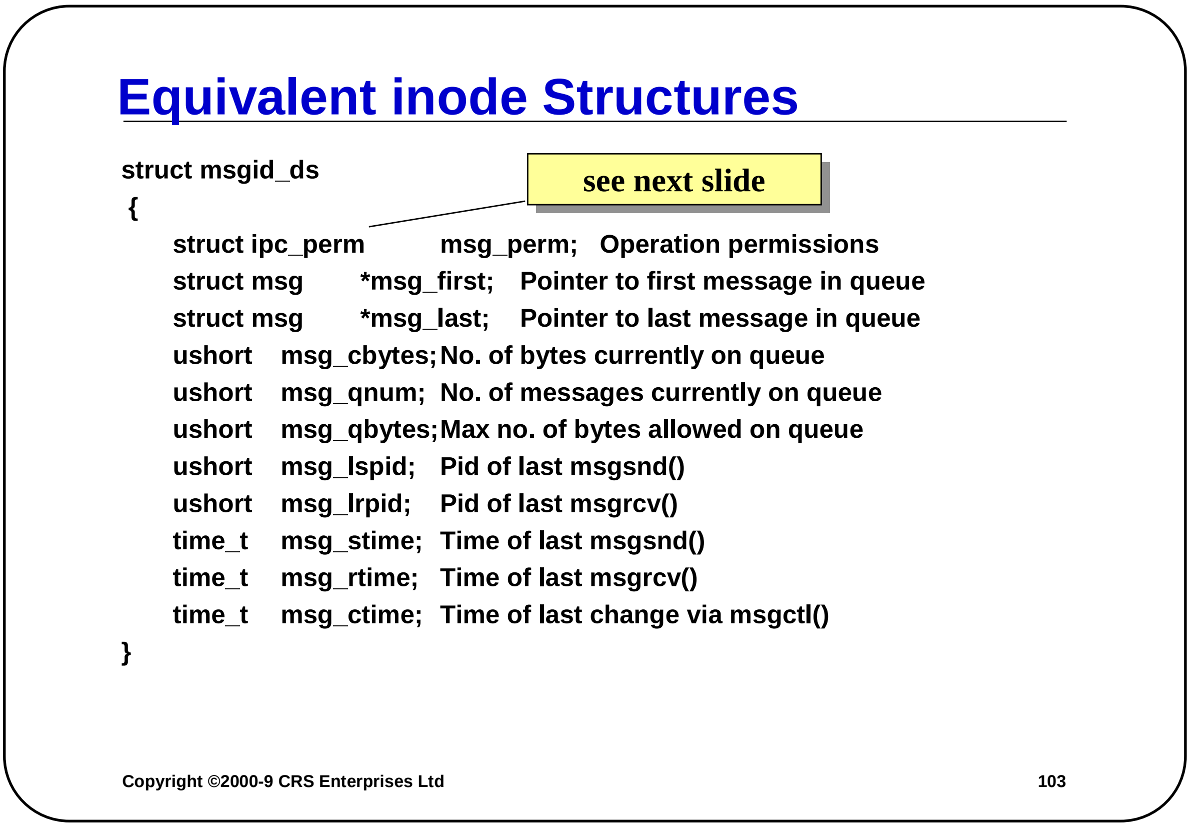
The name by which a process refers to a particular IPC resource is a key value of type key_t. To avoid possible conflicts, processes must agree on key values if they wish to share an IPC resource.

A special key value of IPC_PRIVATE is provided, which will ensure that a unique IPC resource is allocated. Although unrelated processes may not use IPC_PRIVATE channels to communicate with each other, resources allocated in this way are retained and are still visible across fork() calls. This facility provides a useful means of establishing a private IPC between parent and child processes.

The flag values IPC_CREAT and IPC_EXCL behave in the same way as the flags O_CREAT and O_EXCL behave with regard to files:

flags	if resource already exists	if resource does not exist
0	error	attach to resource
IPC_CREAT	attach to resource	create resource
IPC_CREAT|IPC_EXCL	error	create resource

An IPC resource can have protection applied to it (in the same way as files) using the rwxrwxrwx model. This is done using the ...ctl() routines; there is no equivalent Unix command.



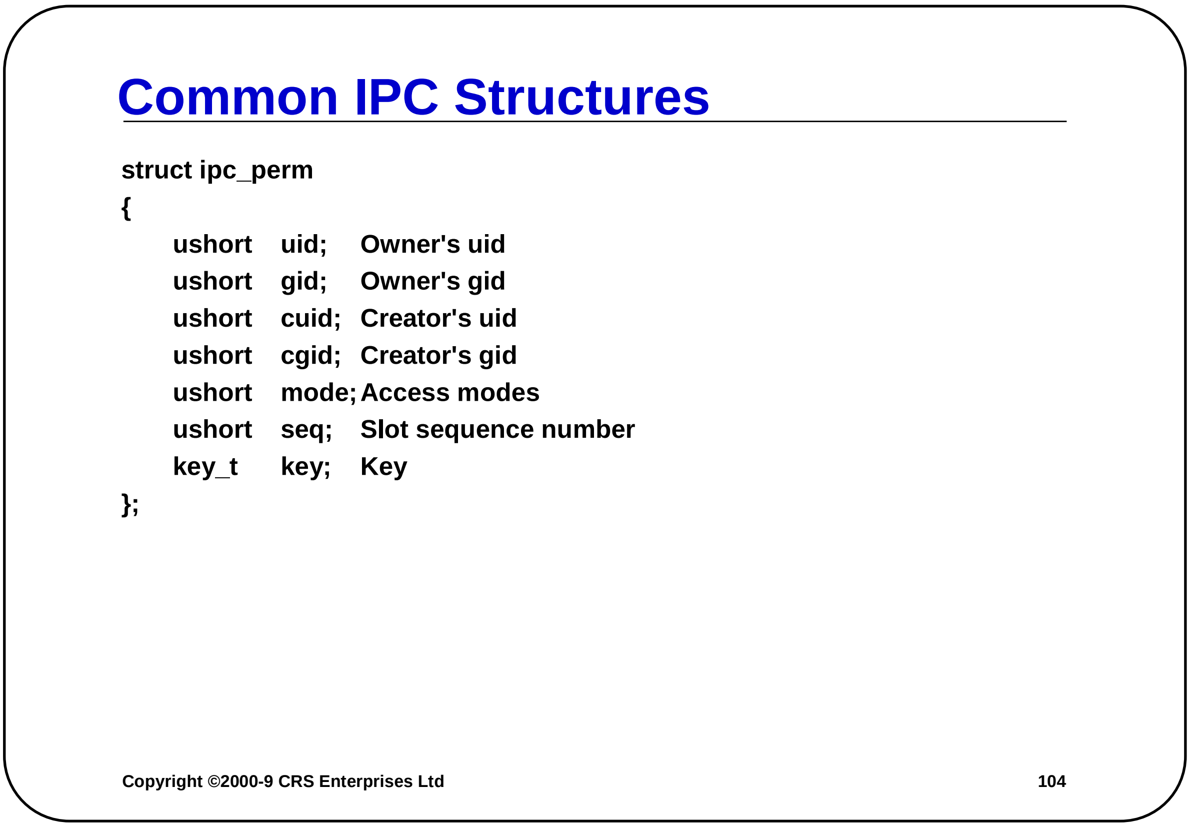
Equivalent inode Structures

see next slide

```
struct msgid_ds
 {
       struct ipc_perm         msg_perm;   Operation permissions
       struct msg      *msg_first;   Pointer to first message in queue
       struct msg      *msg_last;    Pointer to last message in queue
       ushort   msg_cbytes;No. of bytes currently on queue
       ushort   msg_qnum;  No. of messages currently on queue
       ushort   msg_qbytes;Max no. of bytes allowed on queue
       ushort   msg_lspid;  Pid of last msgsnd()
       ushort   msg_lrpid;  Pid of last msgrcv()
       time_t   msg_stime;  Time of last msgsnd()
       time_t   msg_rtime;  Time of last msgrcv()
       time_t   msg_ctime;  Time of last change via msgctl()
}
```

Copyright ©2000-9 CRS Enterprises Ltd 103

The kernel maintains an instance of this data structure for every message queue currently in the system. Similar structures are used for semaphores and shared memory. The fields of this structure may be examined using the IPC_STAT flag with *msgctl()*. Certain fields may be changed using the IPC_SET flag with *msgctl()*.

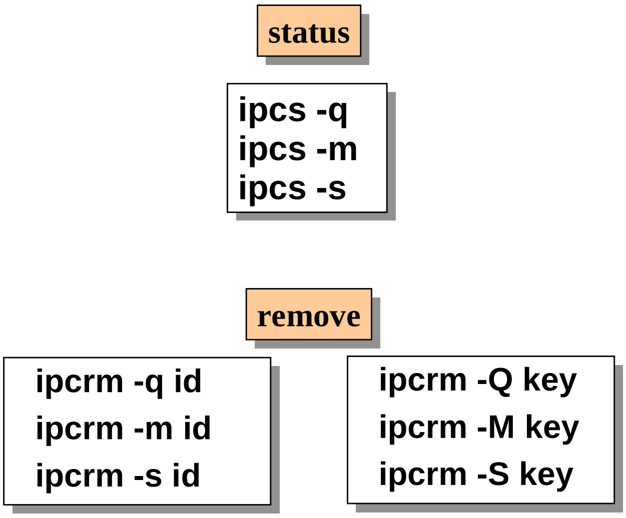


Common IPC Structures

```
struct ipc_perm
{
    ushort  uid;   Owner's uid
    ushort  gid;   Owner's gid
    ushort  cuid;  Creator's uid
    ushort  cgid;  Creator's gid
    ushort  mode;  Access modes
    ushort  seq;   Slot sequence number
    key_t   key;   Key
};
```

Every IPC resource (message queue, shared memory segment or semaphore) has a structure of this type associated with it. Normally, the ipc_perm structure will be included as part of a more specific data structure relating to the type of IPC resource. For example, the message queue data structure is of type struct msgq_ds.

IPC Commands



status

```
ipcs -q
ipcs -m
ipcs -s
```

remove

```
ipcrm -q id
ipcrm -m id
ipcrm -s id
```

```
ipcrm -Q key
ipcrm -M key
ipcrm -S key
```

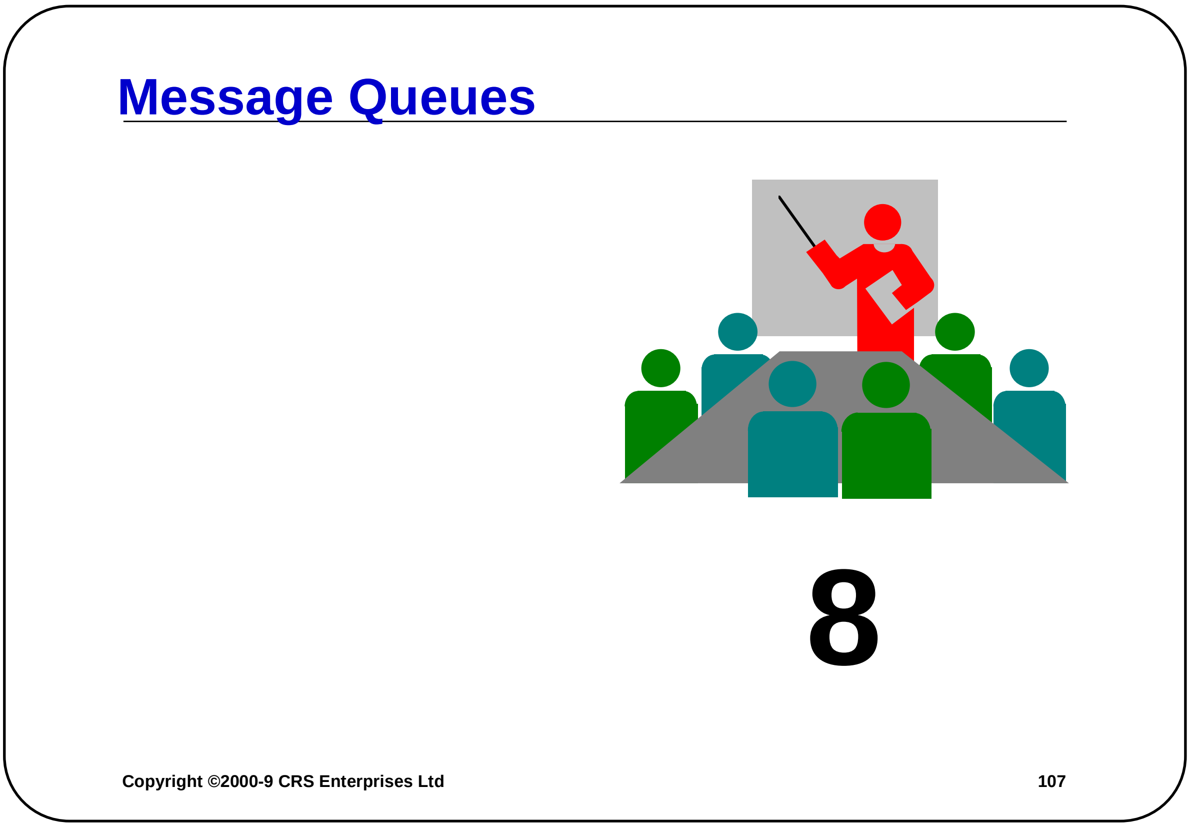
All IPC structures are kept always in the kernel's memory area and can persist even after all processes using the structures have exited. The kernel does not maintain a reference count for IPC resources and therefore does not know when the IPC structure is no longer in use. The Unix commands shown above are provided to determine which IPC structures exist and to allow the removal of unwanted structures.

Use ipcs to obtain status information:

-q	message queues
-m	shared memory
-s	semaphores
-a	all three

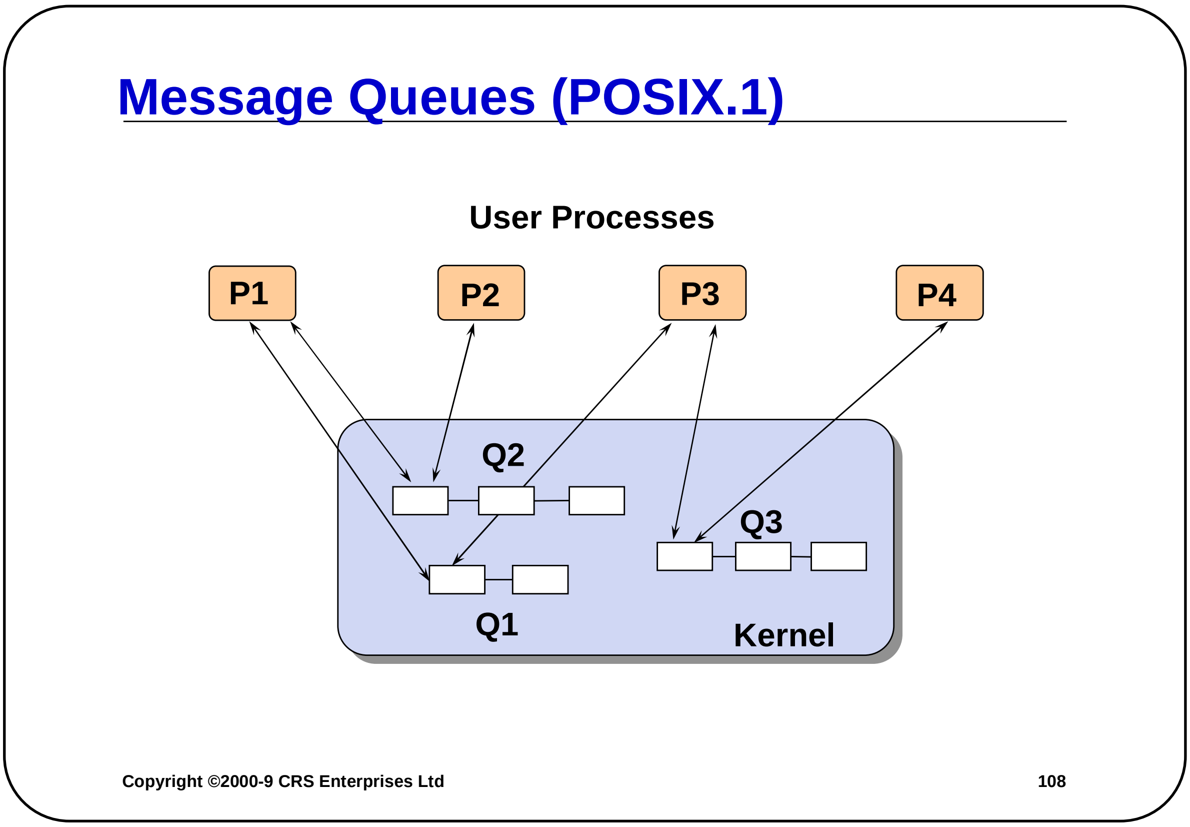
Use ipcrm to remove the structures. Upper case options are used to identify the structure by key and lower case options identify by id.

8



Message Queues

8



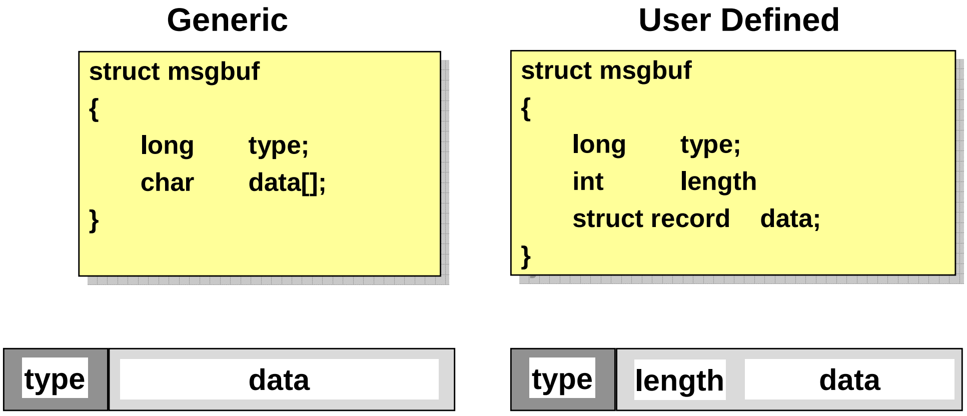
Message Queues (POSIX.1)

Message queues allow processes to exchange tagged "messages", i.e. discrete blocks of data.

As the name suggests, a channel of this kind is maintained as a queue, with new messages added to the end and messages (generally) taken off the front. A process may write to and read from the same message queue, subject to the access controls described earlier. Several processes may share the same message queue, which provides a highly flexible means of exchanging information among processes.

The slide shows a situation where there are three message queues: Q1, Q2 and Q3. Processes P1 and P3 share queue Q1, P1 and P2 share Q2, and P3 and P4 share Q3.

Message Format



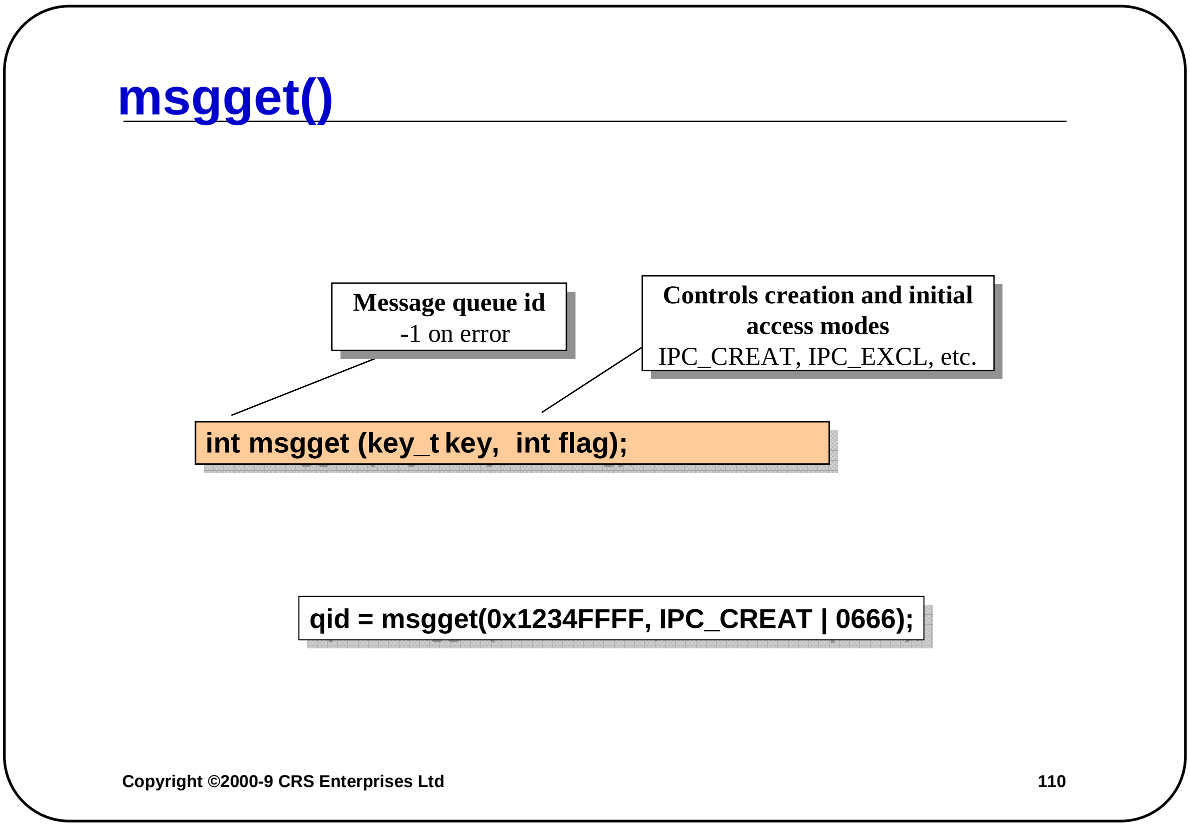
Generic

```
struct msgbuf
{
      long      type;
      char      data[];
}
```

User Defined

```
struct msgbuf
{
      long      type;
      int       length
      struct record    data;
}
```

Messages for use with message queues have a simple format, namely a long value interpreted as a type or tag for the message and a number of bytes comprising the message itself. The message may contain textual or binary data. It may be any length from 0 bytes up to a system-defined maximum, which is normally preset to 8KB, but may be altered by reconfiguring the kernel. The length of the message data is maintained internally; the system call msgsnd(), which sends a message, passes this information into the kernel. Applications may exchange more structured data on message queues, by creating user defined structures.



msgget()

Message queue id
-1 on error

Controls creation and initial access modes
IPC_CREAT, IPC_EXCL, etc.

```
int msgget (key_t key,  int flag);
```

```
qid = msgget(0x1234FFFF, IPC_CREAT | 0666);
```

Copyright ©2000-9 CRS Enterprises Ltd 110

msgget() is called by a process to create a message queue, identified by the value of the key parameter. The IPC_CREATE flag must be used and the initial permissions specified. For example, to create a message queue with hex key 1234FFFF and initial permissions rw-rw-rw- use

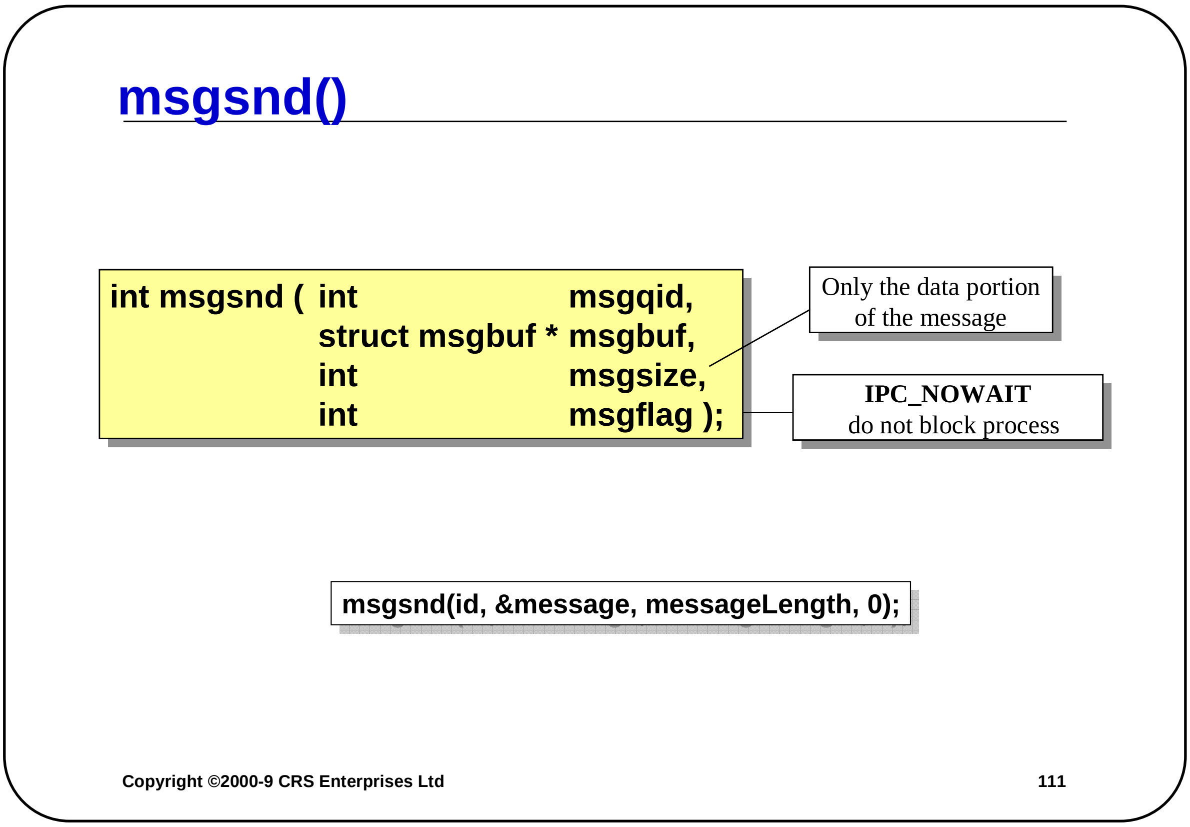
```
id = msgget(0x1234FFFF, IPC_CREAT| 0666);
```

If a process attempts to create a message queue that already exists, the kernel ignores the IPC_CREATE flag and grants access to the shared resource; if IPC_EXCL is also specified and error is returned.

msgget() is called by a process to obtain access to an existing message queue; the flags should be set to zero.

Processes that wish to use message queues to communicate will normally agree on a protocol with regard to which process is responsible for creating or deleting the queue, as well as the key value to identify the queue.

As described previously, the ftok() routine can be used to generate a key value from a pathname plus character-identifier pair. Alternatively, the IPC_PRIVATE special value can be used to guarantee the creation of a unique message queue.



msgsnd()

```
int msgsnd ( int              msgqid,
             struct msgbuf *  msgbuf,
             int              msgsize,
             int              msgflag );
```

Only the data portion of the message

IPC_NOWAIT
do not block process

```
msgsnd(id, &message, messageLength, 0);
```

msgsnd() is used to send a message on a message queue. The queue is identified by the msgqid parameter as returned from a previous call to msgget().

The message itself is referenced by passing a pointer to a structure of type struct msgbuf, as described earlier. The length parameter, msgsize, describes the length of the data portion of the message not including the type field.

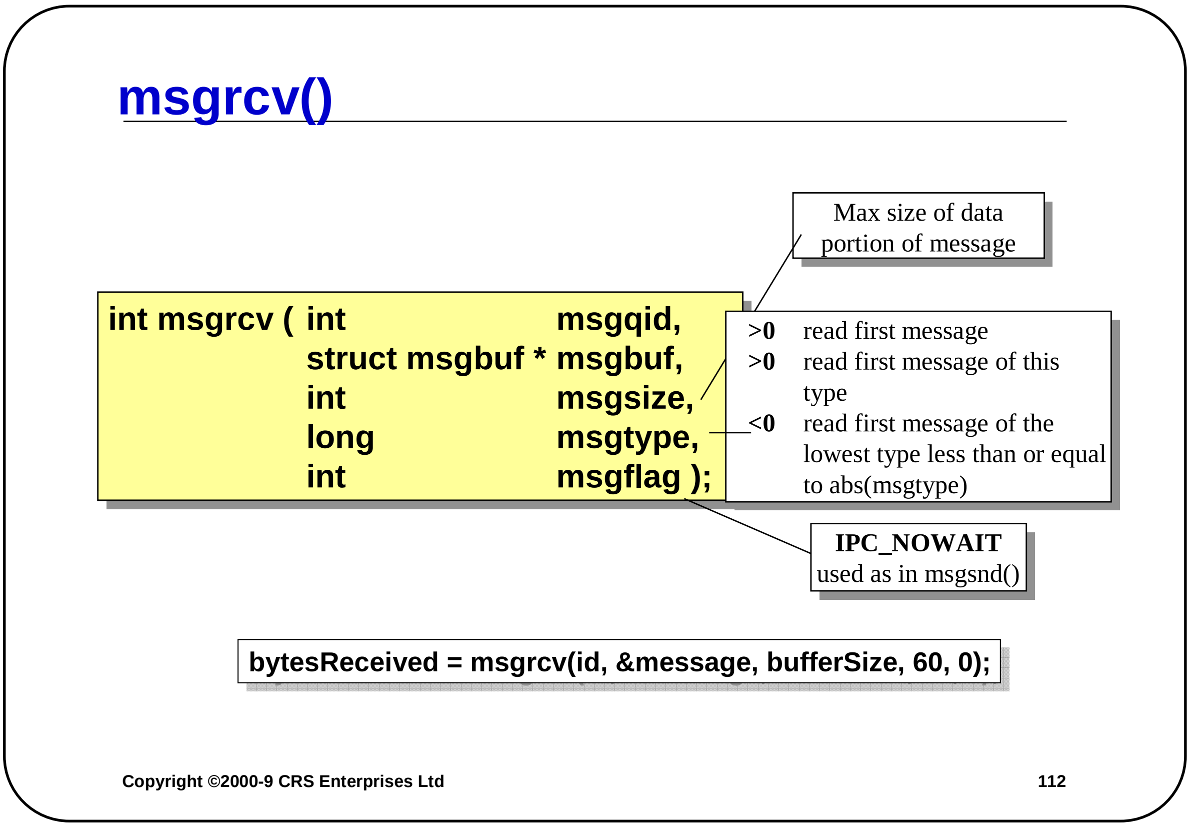
If the message cannot be sent (for example if the queue is full), the value of the msgflag parameter is examined. If this flag has the IPC_NOWAIT bit set, the call will return immediately, indicating an error. Otherwise, the process will block until either:

the message can be sent.

the message queue is removed from the system.

the process is signaled.

msgsnd() returns 0 if successful, or -1 on error.



msgrcv()

```
int msgrcv ( int              msgqid,
             struct msgbuf *  msgbuf,
             int              msgsize,
             long             msgtype,
             int              msgflag );
```

Max size of data portion of message

>0 read first message
>0 read first message of this type
<0 read first message of the lowest type less than or equal to abs(msgtype)

IPC_NOWAIT
used as in msgsnd()

```
bytesReceived = msgrcv(id, &message, bufferSize, 60, 0);
```

Copyright ©2000-9 CRS Enterprises Ltd 112

msgrcv() is used to read a message from the message queue specified by msgqid. The message is read into the buffer area pointed to by the msgbuf parameter. The size of the received message is returned. Set the msgsize parameter to the maximum data portion you have space to store (not including the type field) to avoid overflows.

If there are no messages of the requested type on the queue, msgflag is checked for IPC_NOWAIT. If this flag is set, the call returns an error immediately. If not, the calling process will block until:

a message of the appropriate type appears on the queue.

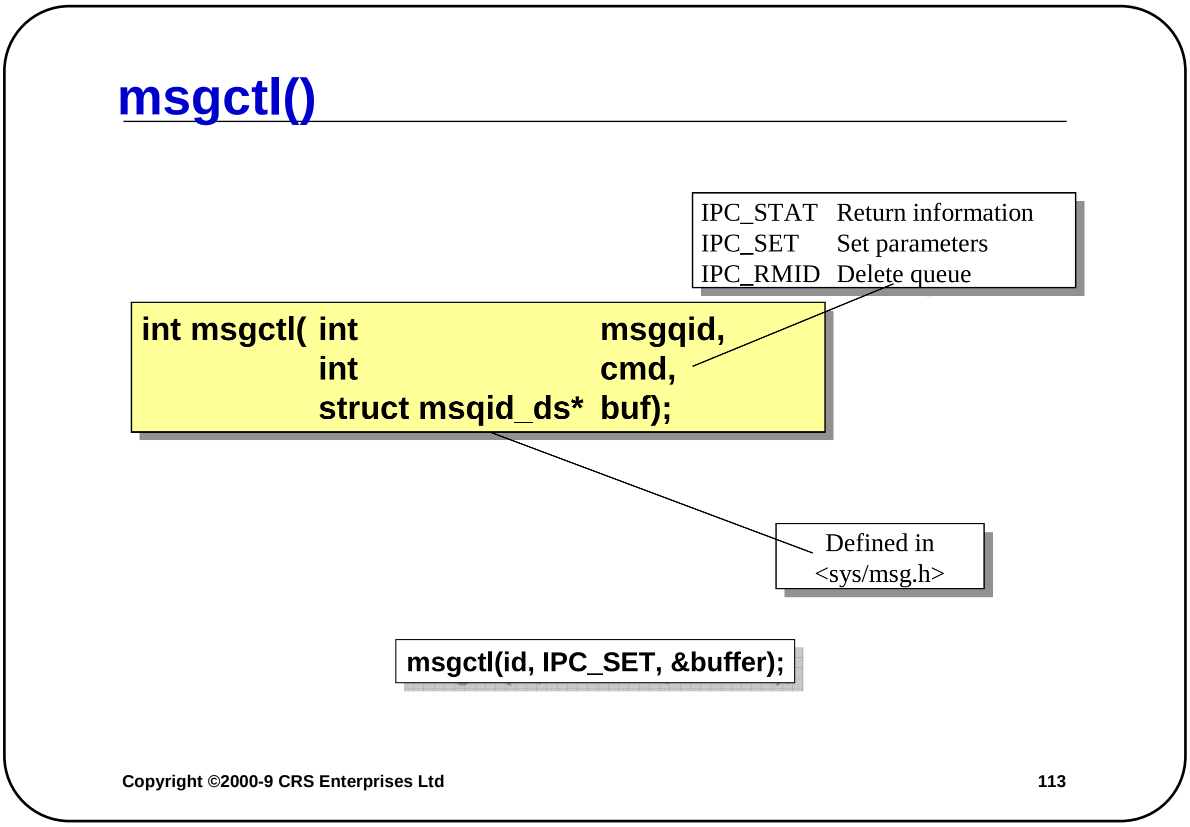
the message queue is removed.

the calling process is signaled.

The msgtype parameter allows the calling process to select particular messages from the queue:

msgtype=0	The first message on the queue is read, regardless of its type.
msgtype>0	The first message of type msgtype is read.
msgtype<0	The first message whose type is the lowest type less than or equal to abs(msgtype) is read. Examples:

For example, if msgtype = -30 and there are messages of types 15, 20 and 25 on the queue, msgrcv() will read the first message of type 15. Alternatively, if msgtype = -30 and there are messages of types 35, 40 and 45 on the queue, msgrcv() will block.



msgctl()

IPC_STAT Return information
IPC_SET Set parameters
IPC_RMID Delete queue

```
int msgctl( int                msgqid,
            int                cmd,
            struct msqid_ds*   buf);
```

Defined in
<sys/msg.h>

```
msgctl(id, IPC_SET, &buffer);
```

msgctl() is used to carry out various control functions on a message queue. Three values for the cmd parameter are available for message queues:

IPC_STAT Return information about the message queue. It fills in the fields of the struct msqid_ds pointed to by buf. This data structure is defined in the include file <sys/msg.h>.

IPC_SET Set the values of certain attributes of the queue using the third parameter to msgctl(); only certain values may be set:

msg_perm.uid

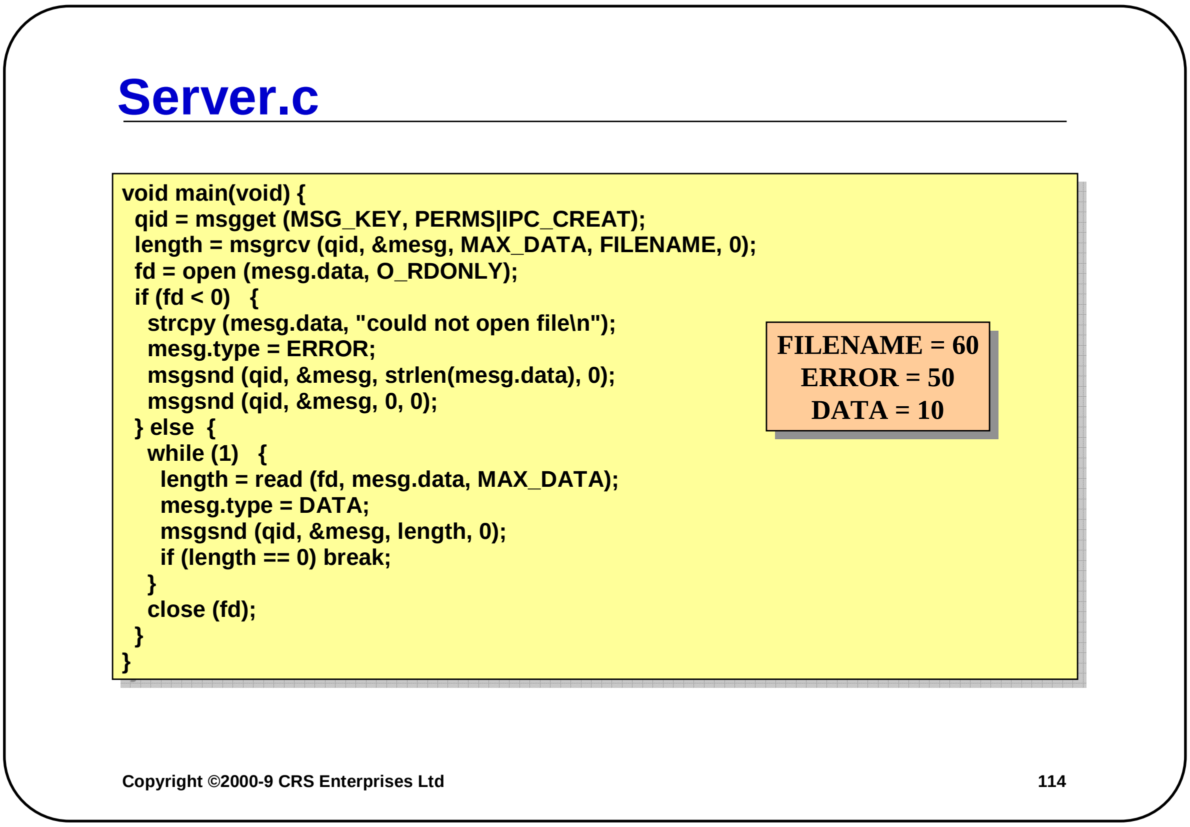
msg_perm.gid

msg_perm.mode (only the low-order 9 bits)

msg_qbytes

These fields can be changed only by the superuser or by a process whose effective uid is equal to the owner or creator of the message queue. msg_qbytes can be increased only by the superuser.

IPC_RMID Remove the message queue from the system. A queue may be removed only by the superuser, by its creator, or by a process whose effective uid is equal to the creator or owner of the queue.



Server.c

```
void main(void) {
  qid = msgget (MSG_KEY, PERMS|IPC_CREAT);
  length = msgrcv (qid, &mesg, MAX_DATA, FILENAME, 0);
  fd = open (mesg.data, O_RDONLY);
  if (fd < 0)   {
    strcpy (mesg.data, "could not open file\n");
    mesg.type = ERROR;
    msgsnd (qid, &mesg, strlen(mesg.data), 0);
    msgsnd (qid, &mesg, 0, 0);
  } else  {
    while (1)   {
      length = read (fd, mesg.data, MAX_DATA);
      mesg.type = DATA;
      msgsnd (qid, &mesg, length, 0);
      if (length == 0) break;
    }
    close (fd);
  }
}
```

FILENAME = 60
ERROR = 50
DATA = 10

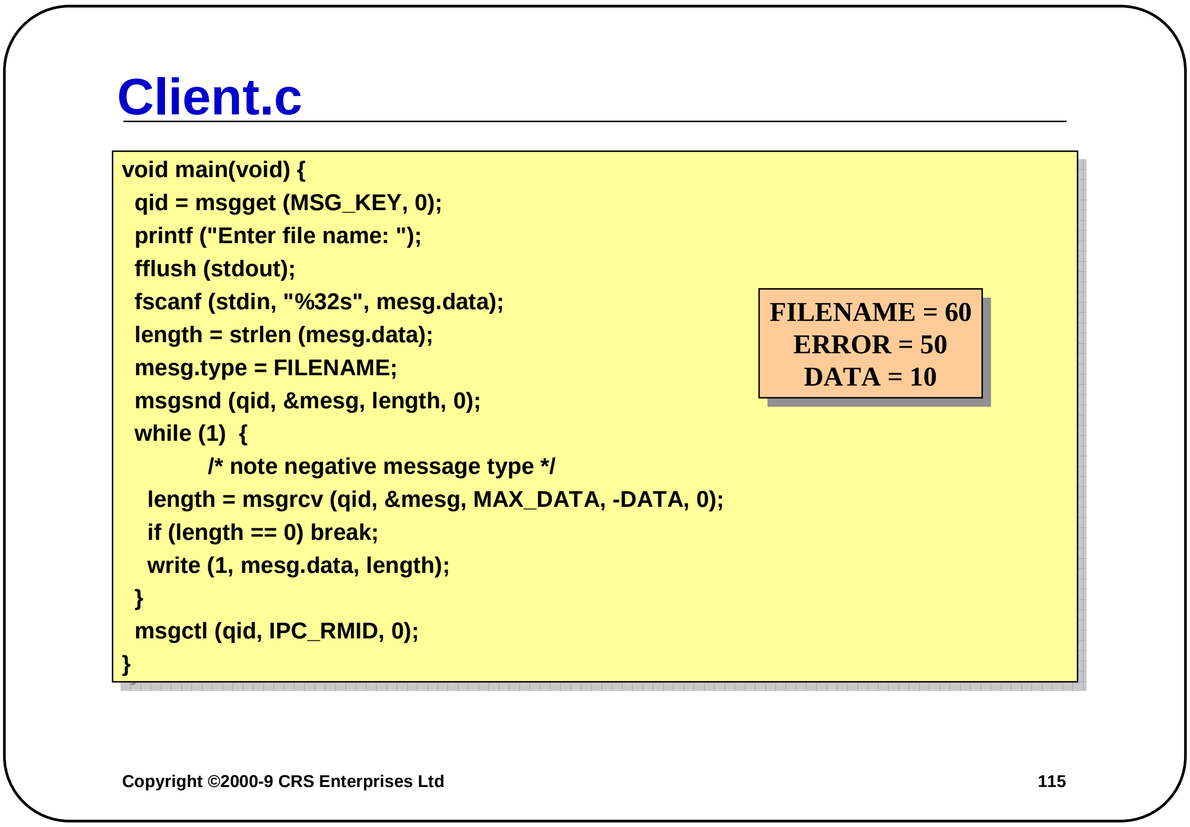
114

This example uses the client/server model for communication between processes. The server must be executed before any client programs, because here the server is responsible for creating the message queues.

1. client sends the filename to the server and the server attempts to open the named file.
2. server sends a data buffer to the client
3. The client receives the buffer of data or error message
4. This continues until the server has no more data to send. The server sends a zero-length message to the client to signal end of transmission
5. The client deletes the message queue using the system call *msgctl()*, then exits when it reads the zero-length message.

The following points should be noted:

1. The header file for this example is given overleaf.
2. Three types of message are used: FILENAME, DATA and ERROR. These are defined in the header file.
3. Although a template for the message structure is defined in <sys/mesg.h>, it is best to define your own structure.
4. The client waits for messages of type -DATA after sending the filename to the server. This is done so that DATA or ERROR messages can be received. Remember that the -DATA message type (-50) means "*get the smallest type <= DATA (50)*". This is done to exclude FILENAME (60) message types; otherwise the client might read and consume its own messages.



Client.c

```
void main(void) {
  qid = msgget (MSG_KEY, 0);
  printf ("Enter file name: ");
  fflush (stdout);
  fscanf (stdin, "%32s", mesg.data);
  length = strlen (mesg.data);
  mesg.type = FILENAME;
  msgsnd (qid, &mesg, length, 0);
  while (1)  {
            /* note negative message type */
    length = msgrcv (qid, &mesg, MAX_DATA, -DATA, 0);
    if (length == 0) break;
    write (1, mesg.data, length);
  }
  msgctl (qid, IPC_RMID, 0);
}
```

FILENAME = 60
ERROR = 50
DATA = 10

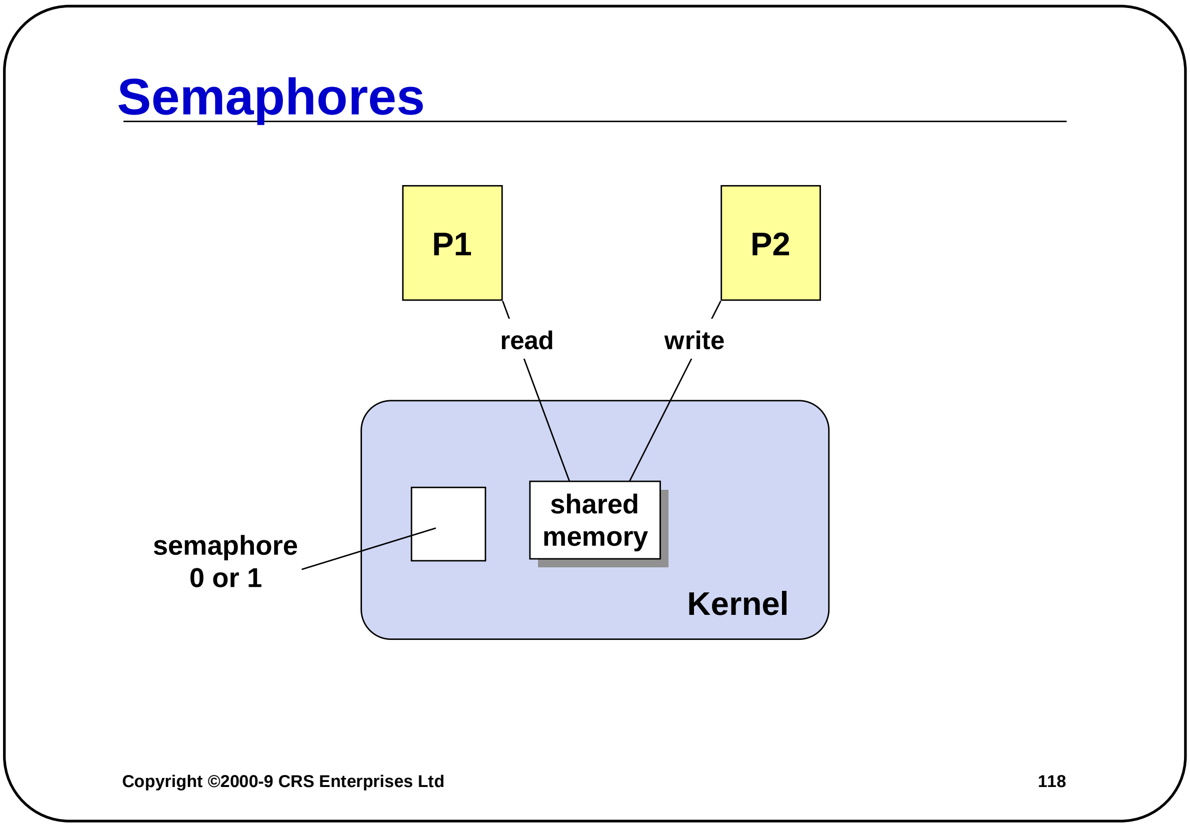
The header file for this example is given below:

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define MAX_DATA  (1024)
typedef struct
{
        long type;                  /* Message type, must be > 0 */
        char data[MAX_DATA];              /* Data being transferred */
} Message;
#define PERMS   0666
#define MSG_KEY  20
#define FILENAME 60
#define DATA   50
#define ERROR   10
```

9

Semaphores

9



Semaphores

118

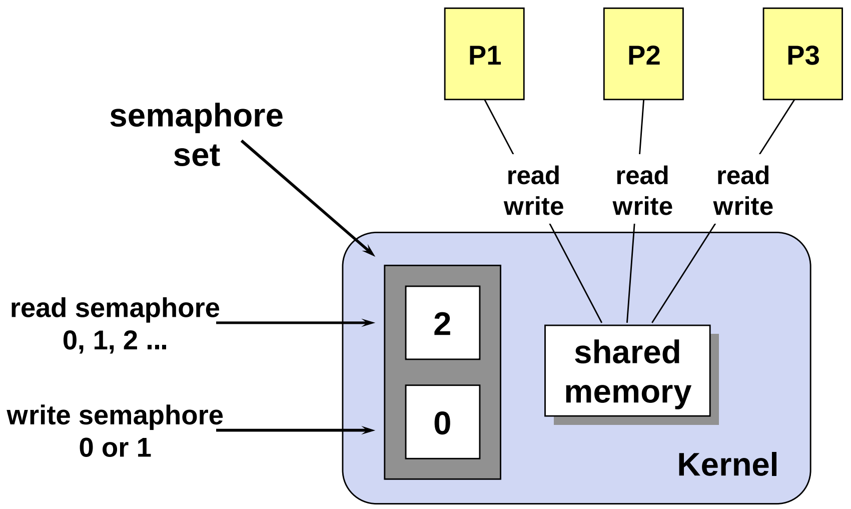
Semaphores are used in the synchronization of multiple processes that require access to some shared resource; for example, a shared memory segment. System V uses counting semaphores.

The most common use of semaphores is for providing a means of locking a resource, so that only one process may access it at a time. The semaphore may be thought of as a "counter" that indicates how many processes have locked the resource.

The crucial aspect of this facility is the ability to guarantee that between the "test" operation and the "set" operation, the process will not be interrupted. Otherwise, it would be possible for another process to alter the value of the semaphore after the first had tested it.

The semaphores described here, as part of the System V IPC mechanisms, provide this assurance. This is because they are implemented wholly within the kernel and are manipulated via system calls that are guaranteed to be atomic with respect to other user processes.

Counting Semaphores

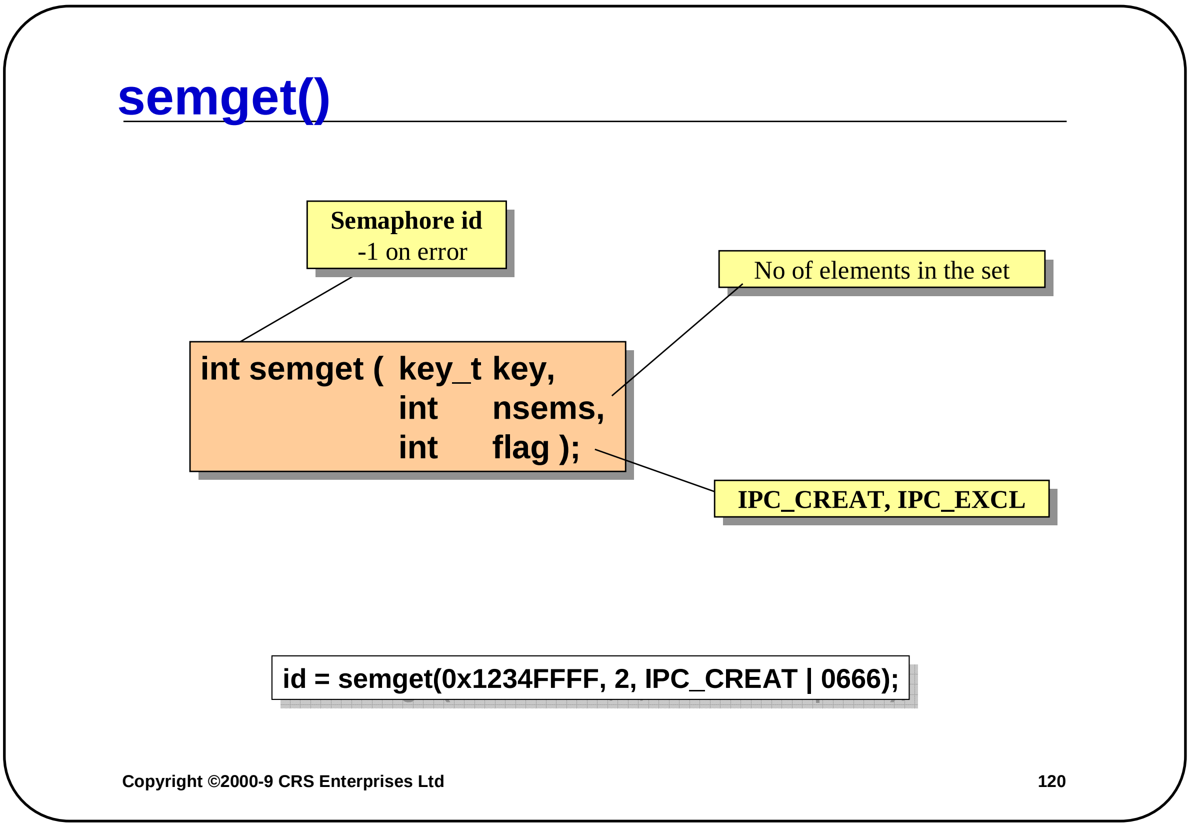


Copyright ©2000-9 CRS Enterprises Ltd 119

System V semaphores provide a very flexible (but also more complicated) set of facilities. A semaphore set is a group of semaphores that work together to synchronize processes. Each semaphore may have any non-negative integer value up to 64KB. This value is never allowed to become negative; any attempt to negate one or more semaphores in a set will cause that process to block.

Binary semaphores can be implemented easily by restricting the semaphore set to a single semaphore and controlling how the semaphore's values are changed, so that they are only ever 0 and 1.

Read and write semaphores require two semaphores in the set; one to control read operations and one to control write operations. These are discussed later in this chapter.



semget()

Semaphore id
-1 on error

No of elements in the set

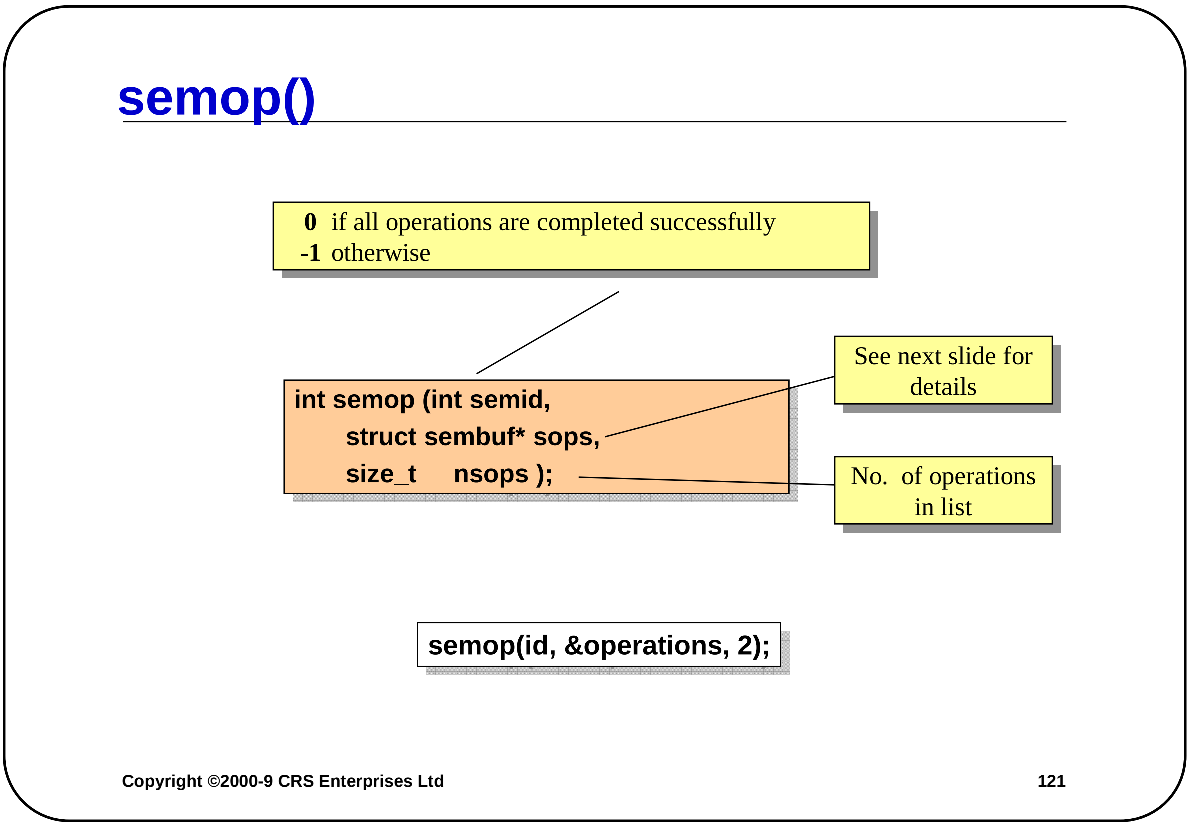
```
int semget ( key_t key,
             int   nsems,
             int   flag );
```

IPC_CREAT, IPC_EXCL

```
id = semget(0x1234FFFF, 2, IPC_CREAT | 0666);
```

semget() is used to access a semaphore, and to create it if necessary. The routine returns a descriptor for the semaphore or -1 on error. The nsems parameter indicates how many semaphores there should be in the semaphore set.

The value of the flag parameter determines the behaviour regarding creation of semaphores, in exactly the same way as for message queues with msgget() and shared memory segments with shmget().



semop()

0 if all operations are completed successfully
-1 otherwise

```
int semop (int semid,
      struct sembuf* sops,
      size_t    nsops );
```

See next slide for details

No. of operations in list

```
semop(id, &operations, 2);
```

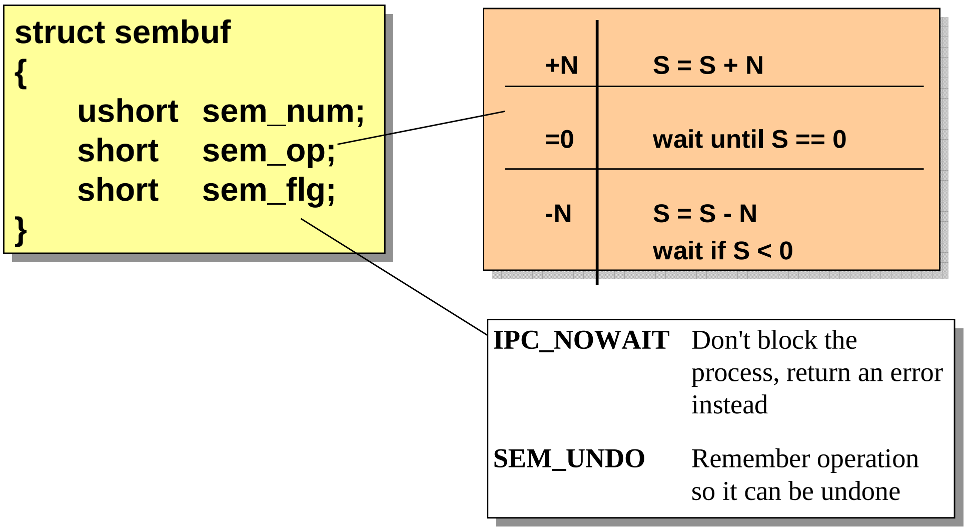
121

All semaphore operations, with the exception of setting an initial value, are carried out using the semop() system call. semop() operates on a semaphore set obtained from a call to semget(), via its semaphore id semid.

The programmer specifies a number of "operations" that are to be applied to the semaphore set. An individual operation is described by an object of type struct sembuf. A pointer to an array of these data structures (sops) is passed to the routine, together with the number of elements in the array (nsops).

semop() will execute all of the operations in the array atomically, i.e. all or none of the operations will be applied.

Semaphore Operations



```
struct sembuf
{
        ushort  sem_num;
        short   sem_op;
        short   sem_flg;
}
```

+N	S = S + N
=0	wait until S == 0
-N	S = S - N wait if S < 0

IPC_NOWAIT	Don't block the process, return an error instead
SEM_UNDO	Remember operation so it can be undone

122

Each operation to be applied to the semaphore is described by a structure of type struct sembuf. The fields of the structure indicate which of the values in the set is to be operated on, the nature of the operation and any flags that are to apply to the operation.

Semaphore values are indexed from 0 to (nsems-1).

The effect of the operation depends on the value of the sem_op field:

sem_op > 0   The value of sem_op is added to the semaphore value.

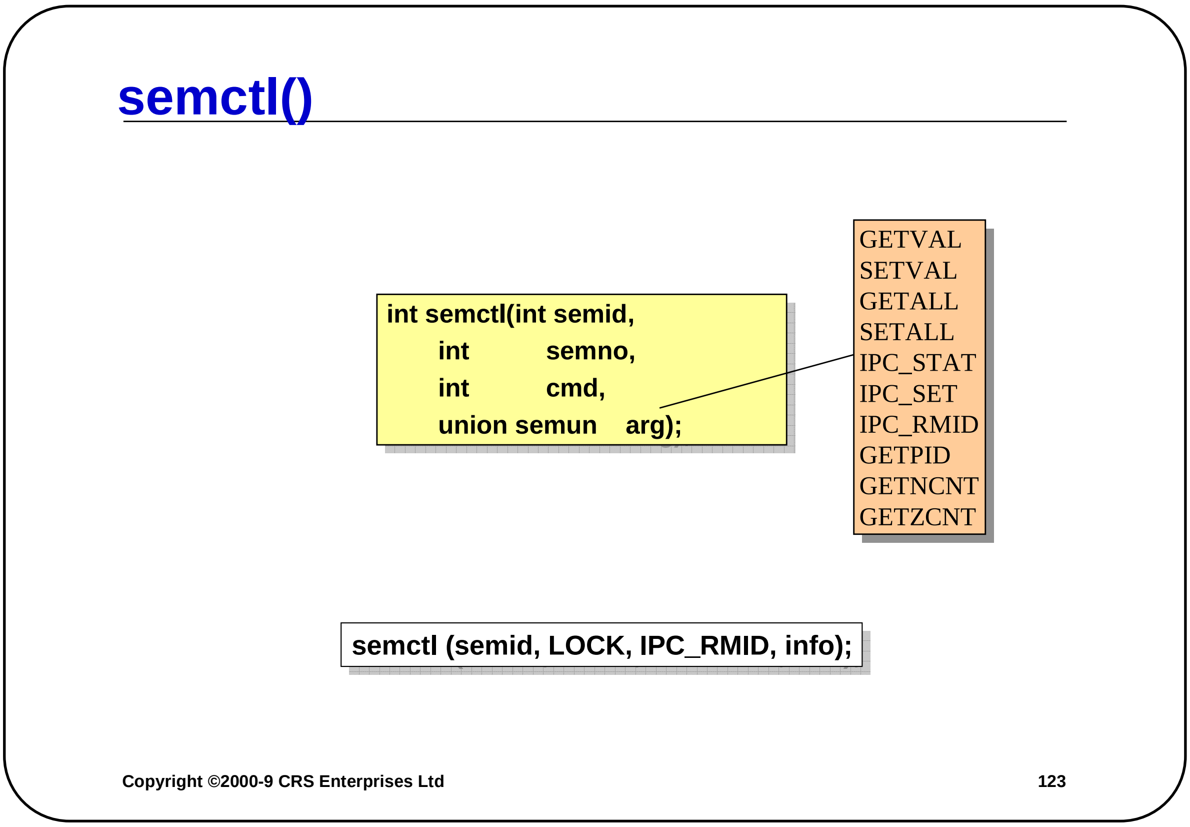
sem_op = 0   The calling process will be suspended until the value of the semaphore reaches 0. If the value is already 0, the call returns immediately.

sem_op < 0   The calling process is suspended until the semaphore value is greater than or equal to abs(sem_op). At this time, abs(sem_op) is subtracted from the value. This is the "test and set" operation.

In the cases where the process blocks, a signal may cause the routine to return an error with an errno of EINTR (interrupted system call). If the semaphore is removed, the call will also return immediately with an error.

Blocking may be prevented using the IPC_NOWAIT flag. If the flag is specified, the routine will return immediately with an error (EAGAIN).

The SEM_UNDO flag causes the sem_op value to be remembered in an "adjust" value, which is maintained on a per-process basis throughout the life of the semaphore. When a process exits, its "adjust" value is applied to the semaphore to clear any locks that the process may have been holding. The SEM_UNDO flag should be used consistently in "lock" and "release" operations; the kernel does not enforce this - it is up to the programmer.



semctl()

```
int semctl(int semid,
      int         semno,
      int         cmd,
      union semun    arg);
```

```
semctl (semid, LOCK, IPC_RMID, info);
```

123

semctl() is used to carry out various control functions on a semaphore, identified by its semid id. The following commands are available for semaphores:

IPC_STAT Return information about the semaphore.

IPC_SET Set the values of certain attributes of the semaphore

IPC_RMID Remove the semaphore from the system. A segment may be removed only by the superuser, by its creator or by a process whose effective uid is equal to the creator or owner of the semaphore.

GETPID Return the value of sempid.

GETNCNT Return the value of semncnt.

GETZCNT Return the value of semzcnt.

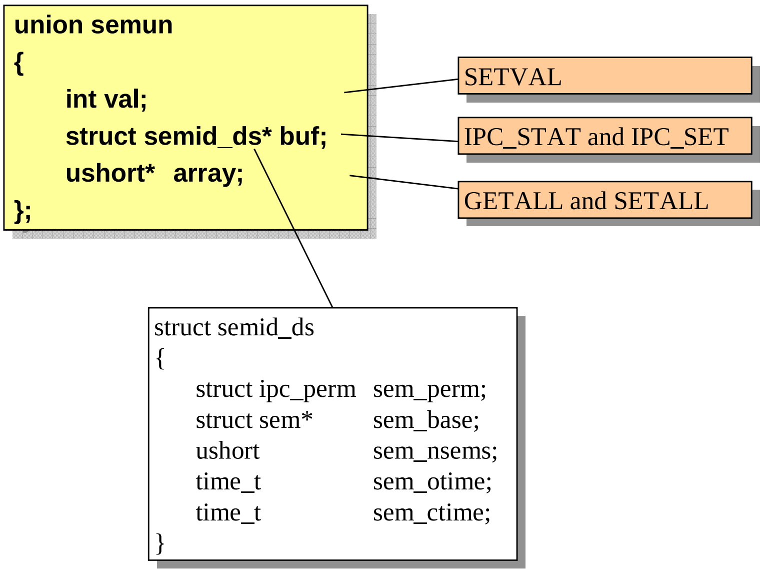
GETVAL Return the value of the specified semaphore.

SETVAL Set the value of the specified semaphore and clear all "adjust" values for the semaphore. Note that this is not a "normal" semaphore operation; the value will be set absolutely, regardless of its current value. It should only be used to initialize the semaphore immediately after creation.

GETALL Return the values of all the elements in the set.

SETALL Set the values of all elements in the set and clear all their adjust values.

Arguments to semctl()



```
union semun
{
      int val;
      struct semid_ds* buf;
      ushort*   array;
};
```

SETVAL

IPC_STAT and IPC_SET

GETALL and SETALL

```
struct semid_ds
{
      struct ipc_perm   sem_perm;
      struct sem*       sem_base;
      ushort            sem_nsems;
      time_t            sem_otime;
      time_t            sem_ctime;
}
```

semctl() takes an argument of different types, depending on the command being applied. The different types are represented in the semun union.

semun.val

> Treat the union as an integer when an individual value is being changed with SETVAL.

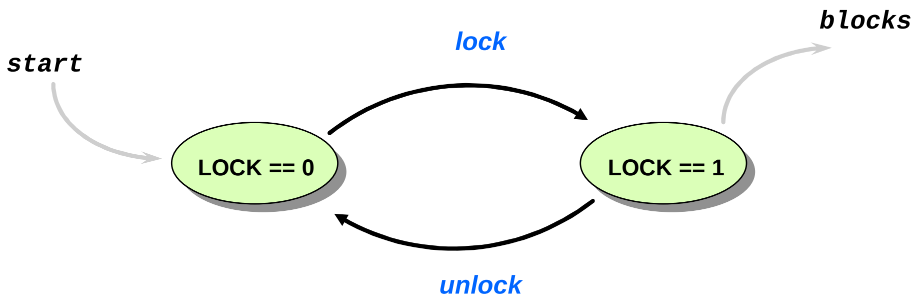
semun.buf

> Treat the union as a pointer to an object of type struct semid_ds when the semaphore attributes are being interrogated.

semun.array

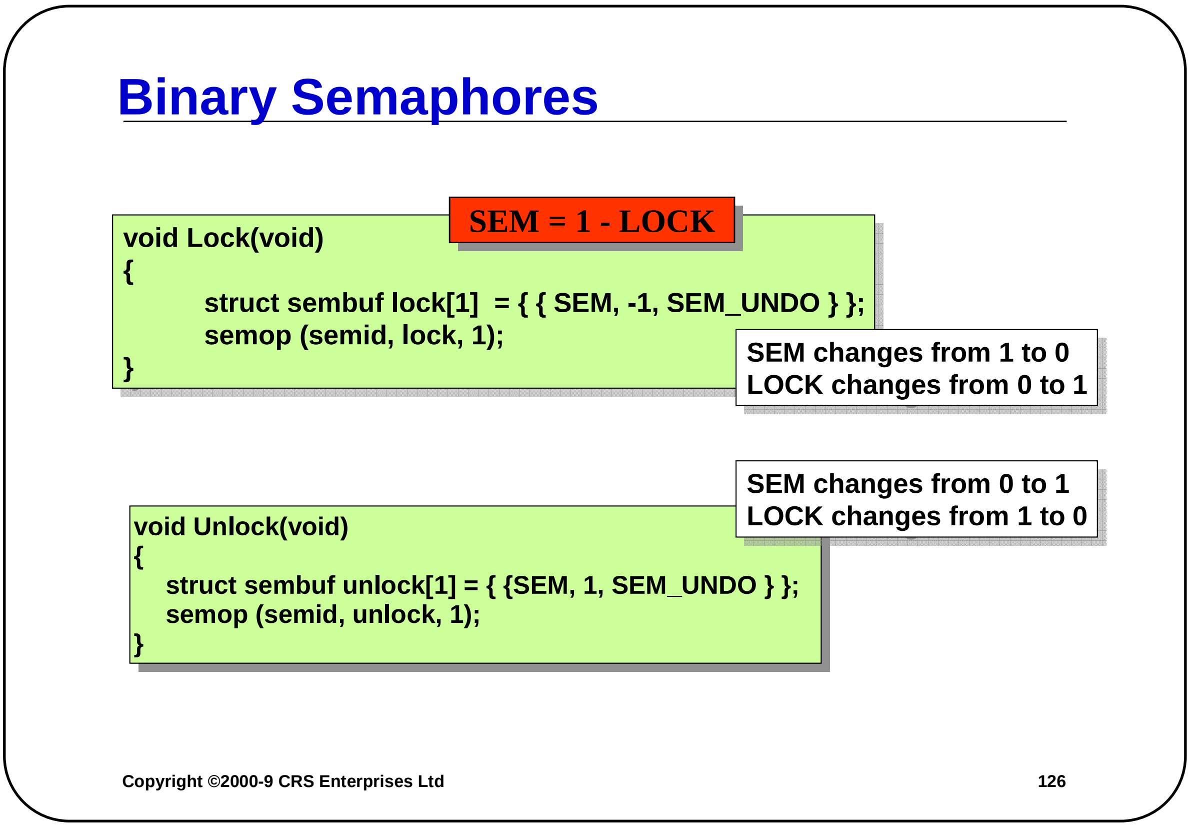
> Treat the union as a an array when the entire value set is being modified or interrogated. The array should consist of unsigned short values and be large enough to contain a value for every element in the semaphore set.

Binary Semaphore



Binary semaphores are the simplest type of semaphore. Only one resource is involved and a single semaphore is used. It is called a binary semaphore because the semaphore has two states: locked and unlocked. Processes can perform operations on the semaphore to change its state. The lock operation will change the semaphore's state to locked, provided it was previously in the unlocked state. Similarly, the unlock operation changes the semaphore's state to unlocked.

If any process attempts to lock the semaphore when it is already in the locked state the process will be suspended (blocked) immediately by the kernel. The kernel will automatically reawaken the process when the semaphore is unlocked by another process. The unlock operation is only used by the process that has locked the semaphore; this operation never blocks.



Binary Semaphores

SEM = 1 - LOCK

```
void Lock(void)
{
        struct sembuf lock[1]  = { { SEM, -1, SEM_UNDO } };
        semop (semid, lock, 1);
}
```

SEM changes from 1 to 0
LOCK changes from 0 to 1

SEM changes from 0 to 1
LOCK changes from 1 to 0

```
void Unlock(void)
{
    struct sembuf unlock[1] = { {SEM, 1, SEM_UNDO } };
    semop (semid, unlock, 1);
}
```

This example illustrates the setting up and use of a binary semaphore.

There is only one semaphore in the semaphore set and the two operations defined restrict its value to be 0 or 1. In the design on the previous slide we defined states in terms of the semaphore LOCK. To simplify coding it is good idea to introduce a semaphore SEM that is related to LOCK by:

SEM = 1 - LOCK

Thus SEM has the values:

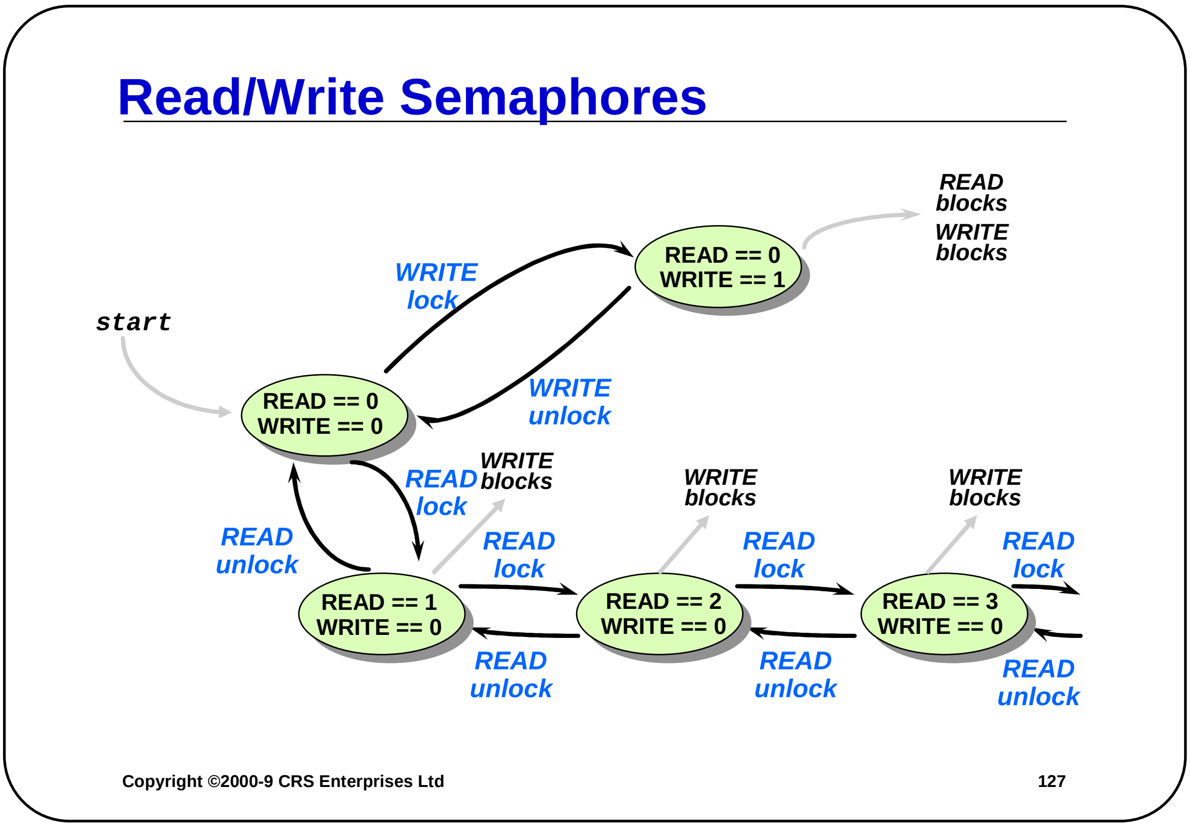
SEM = 1	when LOCK = 0
SEM = 0	when LOCK = 1

We use SEM in the code rather than LOCK. The code for the binary semaphore is relatively straightforward; only the code fragments for locking and unlocking the semaphore are shown.

During the lock operation, we attempt to decrement the value of SEM by 1 using the data structure represented by the array lock[1]. By using a sem_op value of -1 in this operation, we ensure that the operation will complete only when the value of the SEM is 1 already. When the SEM is 0, the process will block.

Notice the importance of the SEM_UNDO flag; if we terminate prematurely without having time to release the semaphore ourselves, the kernel will "undo" the lock.

To unlock the resource, we increment the semaphore value by 1 using the unlock[1] data structure. The SEM_UNDO flag is used consistently in both the lock and unlock operations. This operation cannot possibly block.



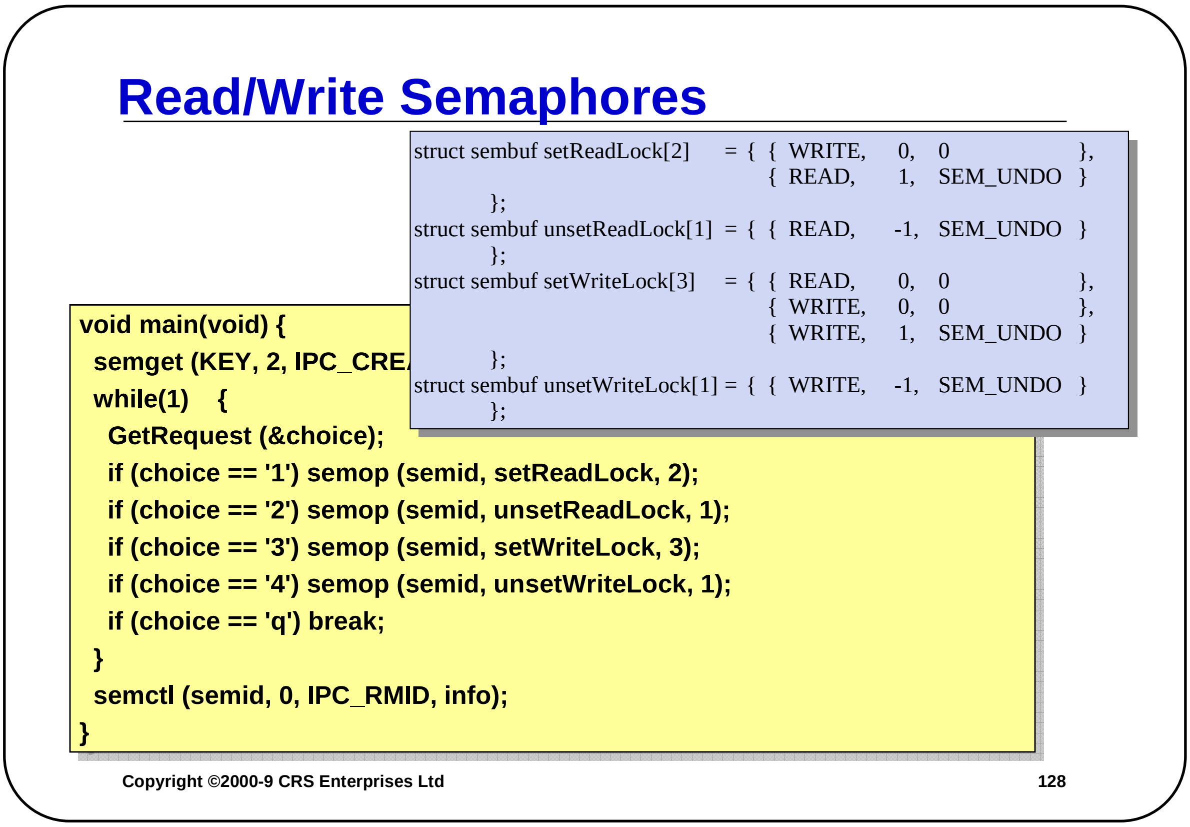
Read/Write Semaphores

127

Semaphores can be used for more complex situations than the simple Binary Semaphore. Consider the typical database access problem where several independent records in a file must be maintained so that multiple processes can read records simultaneously (provided that the records in question are not being updated). If a record is being updated, no processes are allowed to read the record.

To solve this synchronization problem, two semaphores must be used: a READ semaphore and a WRITE semaphore. Several processes can lock the READ semaphore, but only one process can lock the WRITE semaphore.

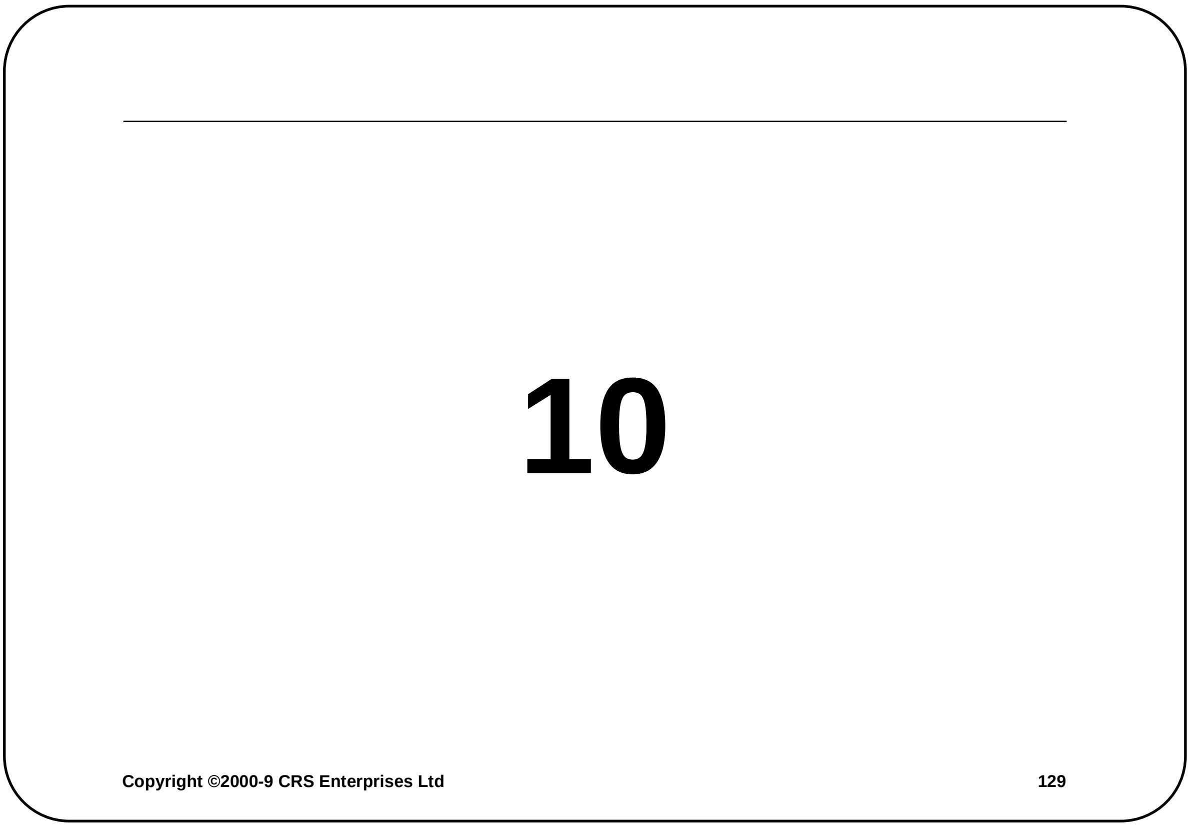
The state diagram of the Read/Write Semaphore problem is shown opposite. Initially, no process has a lock on any record. In the unlocked state, any process can obtain a write lock on a record (we enter the write-lock state) or a read lock on a record (we enter the first read-lock state). In the write-lock state, any process attempting to read lock or write lock the semaphore will block (be suspended). In the read-lock state, any process attempting to write lock the semaphore will block, but processes are allowed to perform further read locks. Each time a process establishes a READ lock, we move further away from the unlocked state. Each of the read lock states is characterized by the number of processes that currently hold a read lock.



Read/Write Semaphores

```
struct sembuf setReadLock[2]    = { {  WRITE,    0,   0              },
                                    {  READ,     1,   SEM_UNDO  }
        };
struct sembuf unsetReadLock[1]  = { {  READ,    -1,   SEM_UNDO  }
        };
struct sembuf setWriteLock[3]   = { {  READ,     0,   0              },
                                    {  WRITE,    0,   0              },
                                    {  WRITE,    1,   SEM_UNDO  }
        };
struct sembuf unsetWriteLock[1] = { {  WRITE,   -1,   SEM_UNDO  }
        };
```

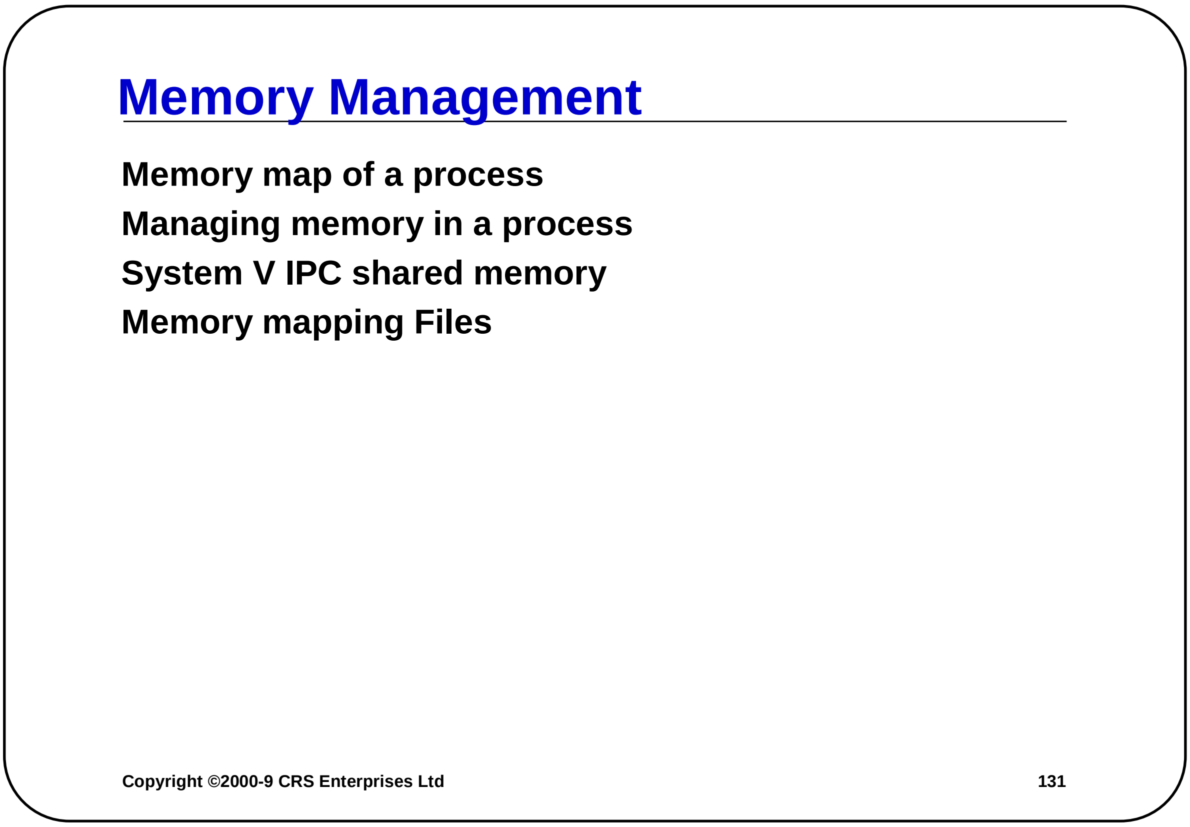
```
void main(void) {
  semget (KEY, 2, IPC_CRE
  while(1)    {
    GetRequest (&choice);
    if (choice == '1') semop (semid, setReadLock, 2);
    if (choice == '2') semop (semid, unsetReadLock, 1);
    if (choice == '3') semop (semid, setWriteLock, 3);
    if (choice == '4') semop (semid, unsetWriteLock, 1);
    if (choice == 'q') break;
  }
  semctl (semid, 0, IPC_RMID, info);
}
```



10

Memory Management

10



Memory Management

Memory map of a process

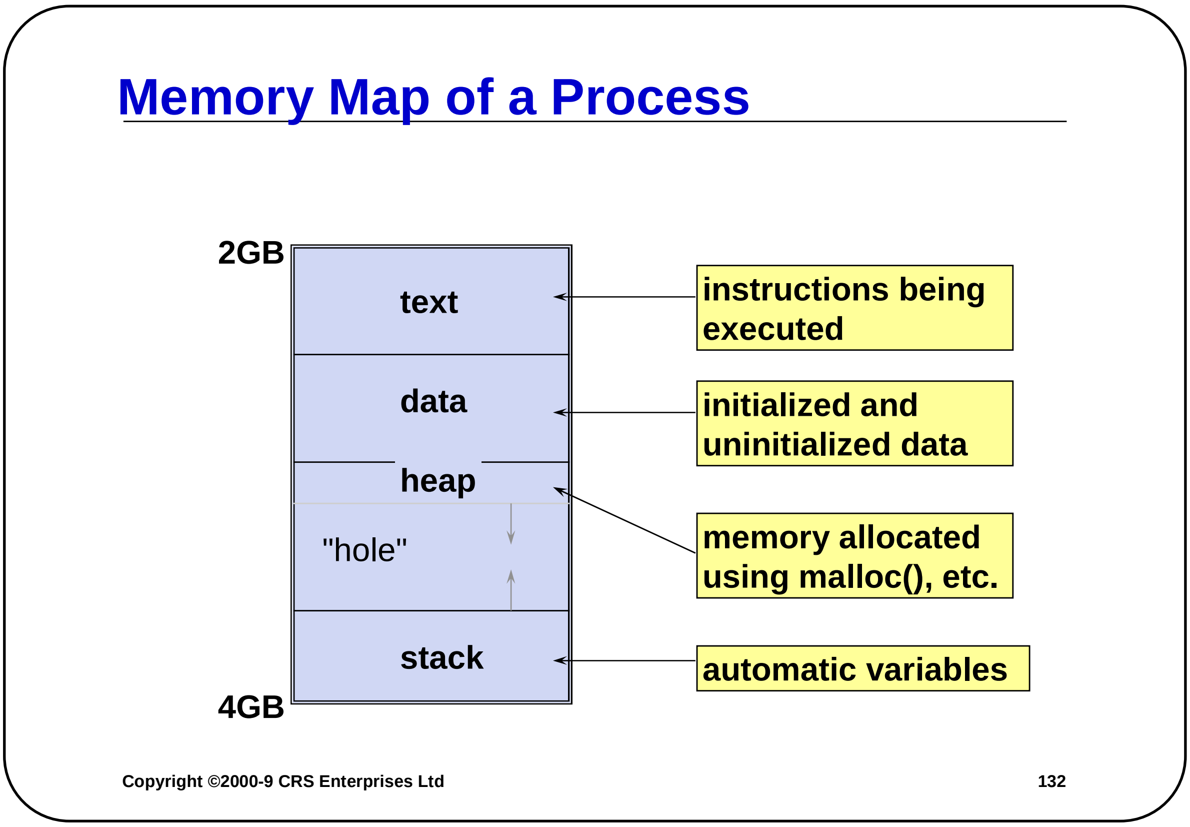
Managing memory in a process

System V IPC shared memory

Memory mapping Files

131

This section deals with the way in which memory is managed on behalf of a Unix process. First, we look at the "standard" memory map of a process, showing the various sections or "segments" of the process. Shared memory is discussed and compared with memory mapping.



Memory Map of a Process

2GB

text

data

heap

"hole"

stack

4GB

instructions being executed

initialized and uninitialized data

memory allocated using malloc(), etc.

automatic variables

Copyright ©2000-9 CRS Enterprises Ltd 132

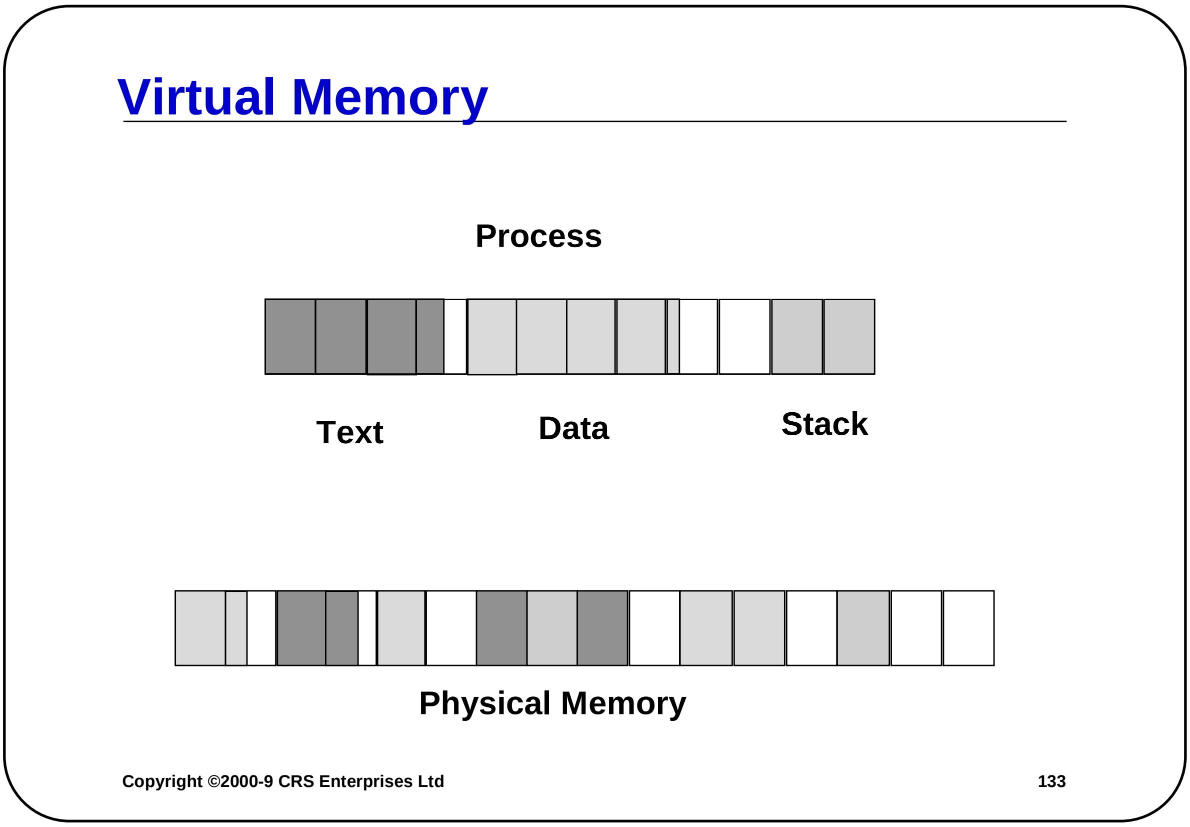
A Unix process executes in its own virtual memory or virtual address space. The virtual memory is a linear address space, (i.e. addressed from 0 upwards). Its size will depend on the implementation, but modern systems allow processes to use 32 bits to specify a virtual address, giving a virtual address space of some 4GB (the first 2GB is usually reserved by the kernel).

The virtual memory of a process is organised into sections or "segments", as shown above. This layout, and the addressing scheme to achieve it, is constructed by the link editor (ld), and the correct memory mappings are established when a program is executed (more about this later).

The text portion of a process is the area of memory containing the instructions being executed. The virtual-memory system arranges for this memory to be read-only, and any attempt by a process to change the contents of a location in this area will result in a trap (usually a segmentation violation or bus error).

The data segment contains all static and global variables, initialized and uninitialized, of the program. The heap area is used when more memory is allocated to the process dynamically (using malloc() and related routines). The stack area is used to hold the stack frames of the process, which may contain return addresses from routines and automatic variables of the routines being called.

There will normally be a (very large) hole in the memory map between the heap and the stack. However, given that both areas may grow during execution, it is possible, although unlikely, that they may collide. In this case, the process will terminate with an appropriate error message. You may normally assign an upper limit to the sizes of the stack and data segments of a process (as well as some other resource limits such as cpu time). This is done using the setrlimit() system call (the limits can be examined using getrlimit()).



Virtual Memory

The memory of a process can be thought of as a series of pages. The size of a page depends on the system being used (it is a property of the memory-management hardware of a machine), but will normally be 4KB or 8KB.

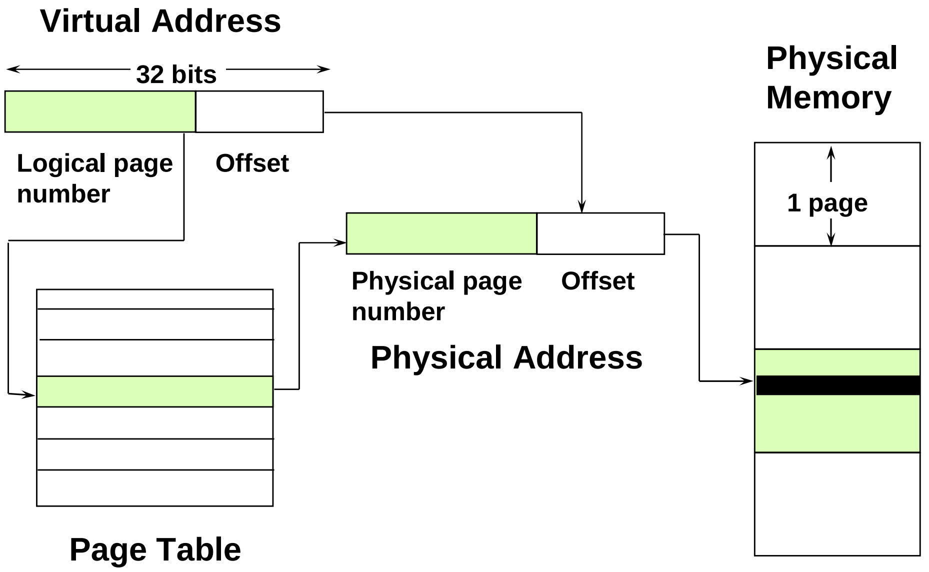
Groups of related pages are known as segments. Earlier versions of Unix treated segments as entities in themselves, but more recent versions deal with pages in preference to segments. In common with the "traditional" Unix model, a process has three main segments associated with it, namely text, data and stack. This has already been described.

Processes access their address space through virtual addresses. The virtual-memory system maps the virtual addresses, a page at a time, onto the machine's physical memory pages, so that accesses to locations in the process's memory will reference the correct data in physical storage.

The physical pages corresponding to a process's virtual pages need not be contiguous; indeed, they do not even have to appear on the same order, and may be scattered throughout the physical memory.

In order to maximise the use of physical memory amongst processes, the system will also copy infrequently-used pages out to a special area of disk called the "swap area". If a process tries to read or write such a page, then a "page fault" occurs and the virtual memory system will copy the page back into memory, setting up the virtual to physical mapping so that the data is accessible again.

Virtual and Physical Addresses



Processes use virtual (or logical) addresses to access locations in their virtual memory. Basically, a virtual address can be thought of as comprising two parts: a logical page number and an offset within the page. We can do this because the logical and physical pages for a given system will be the same size and we want to translate a logical page number into the corresponding physical page number.

The number of bits used to represent the offset and page number is dependent on the implementation. For example, there must be enough bits in the "offset" portion to represent all possible locations in a page, so this will depend on the system's page size. For a system with pages of 8KB bytes, the "offset" portion of the virtual address would be 13 bits.

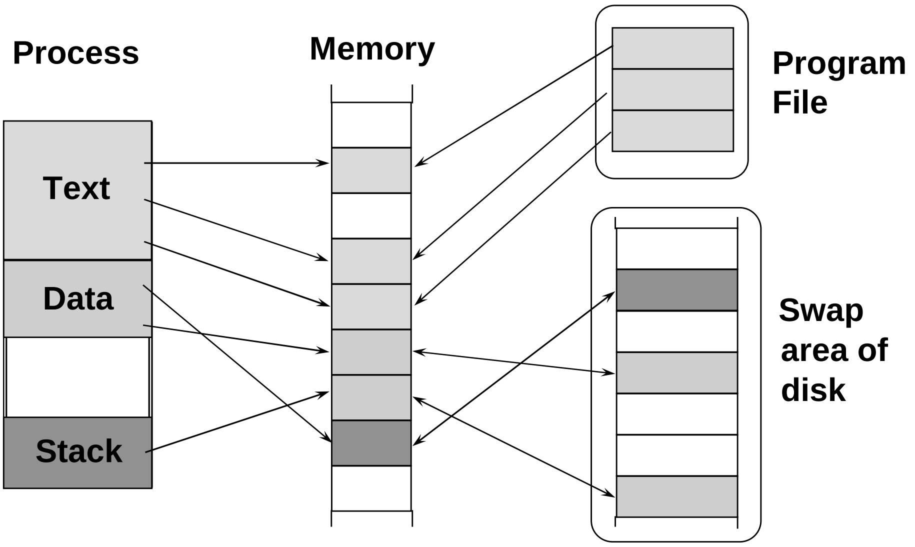
The logical page number is actually an index into a table known as the page table, which contains details of all the logical pages for a process. The information held in the page table includes (obviously) the address of the start of the physical page corresponding to the logical page, and also whether that page of memory can be written to (it may be part of the process text segment, in which case it is read only).

The translation from logical to physical address involves obtaining the physical page number (read from the correct page table entry) and combining it with the offset portion of the logical address to specify the location within the physical page. As part of this procedure, the access controls for the page are also checked, and if necessary a write-protect exception can be raised.

The mapping of logical to physical address is normally carried out by the memory-management-unit hardware in a system, as it clearly must be an extremely fast operation.

The page tables are part of a process's context, and the operating system arranges for the correct page tables to be loaded into the MMU whenever a process is scheduled to run.

Backing Virtual Memory



Under normal circumstances, there will be many processes, each of which has up to 4GB of virtual memory. Clearly, there cannot be a one-to-one mapping between virtual and physical pages. In other words, it is not possible for all pages from all processes to be resident in memory at all times.

The system will attempt to use the available physical memory as efficiently as possible, so that the performance of individual processes is not impacted and each has a fair share of memory. Pages of physical memory are periodically copied out to a special area of disk (called the swap area) so that the memory pages can be used by other processes.

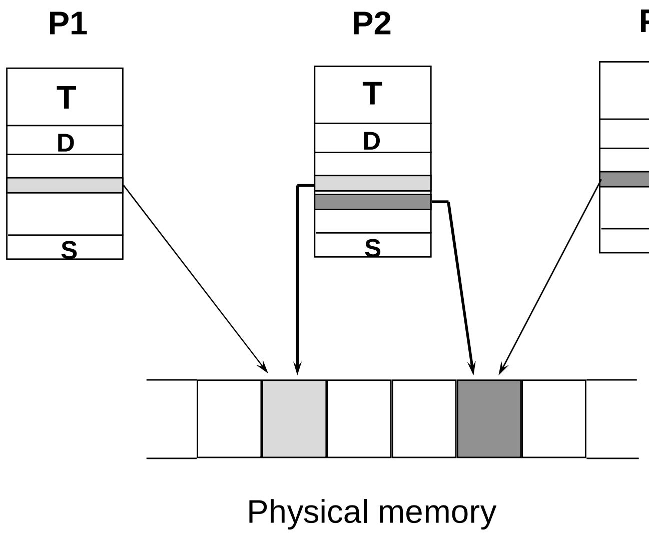
At this point, the page table entry for that page in the process will be marked to indicate that the page is on disk and not in memory. Any access to virtual addresses in the page will result in a "page fault", at which time the relevant page is copied back into memory. The page may be read in to a different physical page before the copy out to disk (known as "paging out"), but the page table entry will be updated to note the new mapping.

Since the text segment of a process is read only, many processes can share access to the program. The swap area is not used in this case; the memory pages are "backed" by the pages of the program file itself. When a text page is being "reused", the data is not copied to the swap area; it can always be reread from the program file when it is paged in again.

Every time the virtual memory of the process is increased, for example by calling sbrk(), or even just when the stack segment is increased by entering a new program block, any new pages assigned to the process must be backed by some amount of disk storage. This will normally be in the swap area of the disk. If there is not enough swap space to back the memory, then the allocation will fail and the error ENOMEM (out of memory) will be returned.

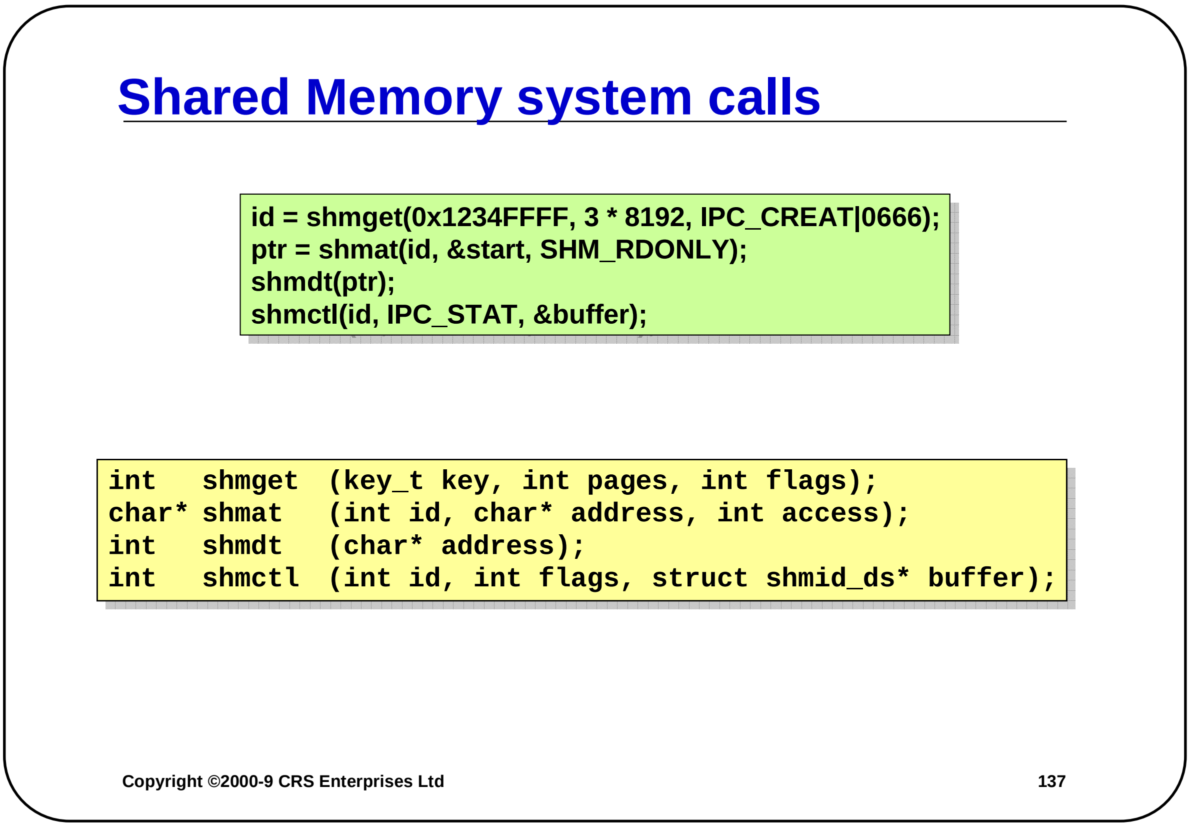
Normally, Unix systems will require at least 2 x physical memory in swap area.

Shared Memory



Shared memory provides the ability for multiple processes to map the same portion of physical memory into their individual virtual-address space. Once this has been done, the memory can be accessed from the processes as if it was normal memory.

A process can map more than one area of shared memory into its address space. In the slide, we see that processes P1 and P2 share the same piece of memory, and P2 and P3 share a different memory area.



Shared Memory system calls

```
id = shmget(0x1234FFFF, 3 * 8192, IPC_CREAT|0666);
ptr = shmat(id, &start, SHM_RDONLY);
shmdt(ptr);
shmctl(id, IPC_STAT, &buffer);
```

```
int   shmget  (key_t key, int pages, int flags);
char* shmat   (int id, char* address, int access);
int   shmdt   (char* address);
int   shmctl  (int id, int flags, struct shmid_ds* buffer);
```

Copyright ©2000-9 CRS Enterprises Ltd 137

shmget() is used to access a shared memory segment. The size parameter indicates the size in bytes of the requested segment rounded to the nearest page size. The value of the flag parameter will determine the behaviour regarding the creation of shared memory segments, in exactly the same way as for message queues with msgget().

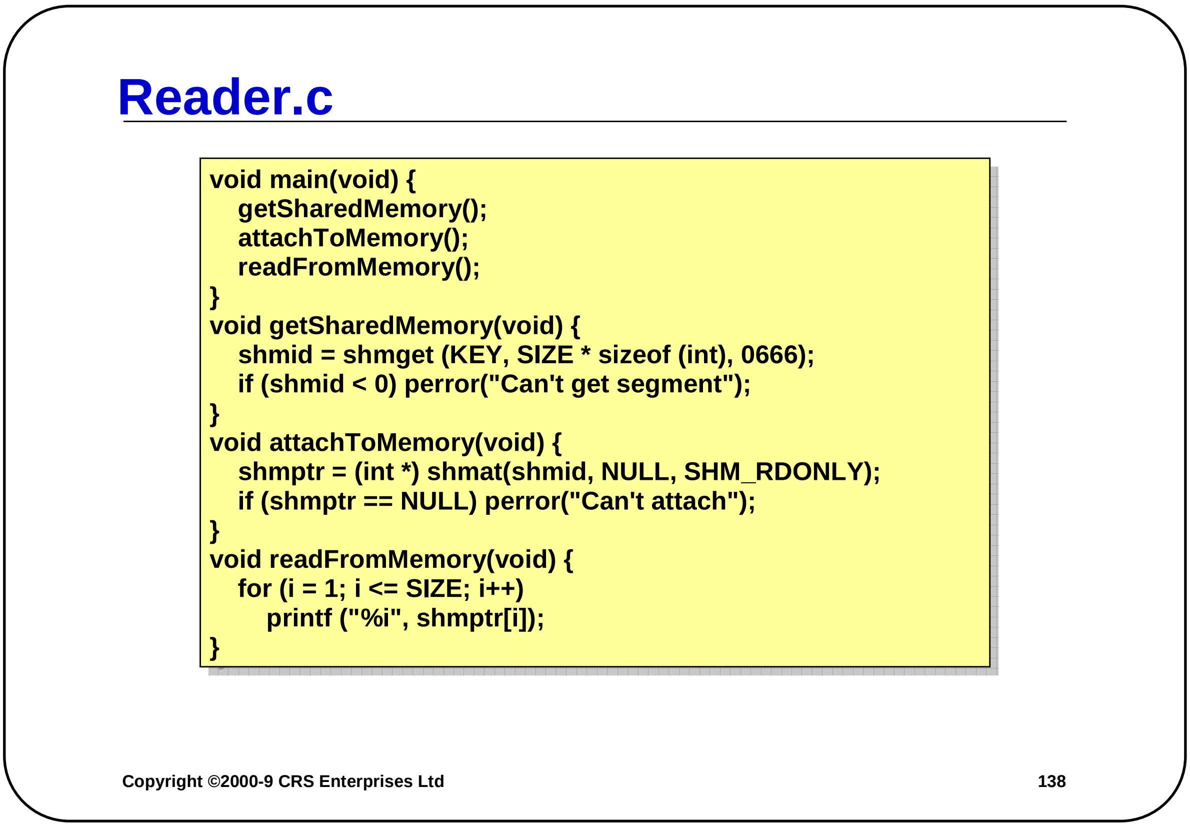
shmat() attaches a segment to the process's address space. This is achieved using the shmat() call. The address for attaching the segment may be specified in the shmaddr parameter; if specified as 0 the system will select an optimum address. shmat() will return the address where the memory has been attached. Once shmat() has returned successfully, the shared-memory segment is accessible to the process in exactly the same way as normal memory; there are no special access routines.

If a specific address is to be used, then the shmflag parameter can be set to SHM_RND, which will round the address down by the value of the symbolic constant SHMLBA (Lower Boundary Address). The segment may be mapped "read-only" by setting SHM_RDONLY.

shmdt() detaches the shared memory segment from the process's address space. shmdt() does not remove the shared-memory segment from the system.

shmctl() is used to carry out various control functions on a shared-memory segment. The following commands are available for shared-memory segments:

IPC_STAT	Return information about the segment.
IPC_SET	Set the values of certain attributes of the segment.
IPC_RMID	Remove the shared segment from the system.
SHM_LOCK	Lock the segment into memory.
SHM_UNLOCK	Unlock a segment previously locked with SHM_LOCK.



Reader.c

```
void main(void) {
    getSharedMemory();
    attachToMemory();
    readFromMemory();
}
void getSharedMemory(void) {
    shmid = shmget (KEY, SIZE * sizeof (int), 0666);
    if (shmid < 0) perror("Can't get segment");
}
void attachToMemory(void) {
    shmptr = (int *) shmat(shmid, NULL, SHM_RDONLY);
    if (shmptr == NULL) perror("Can't attach");
}
void readFromMemory(void) {
    for (i = 1; i <= SIZE; i++)
        printf ("%i", shmptr[i]);
}
```

138

Consider the example code for a reader and writer of shared memory.

The writer must be run first to create the shared-memory segment and fill the segment with data. The reader program can be run at any time after the segment has been created. Note that there is no requirement for the two processes to be run concurrently and that the shared memory segment persists, even after each process completes (no one deletes it!). Thus, the reader program can be rerun as many times as desired and will still work. You will need to use ipcrm to remove the segment from memory.

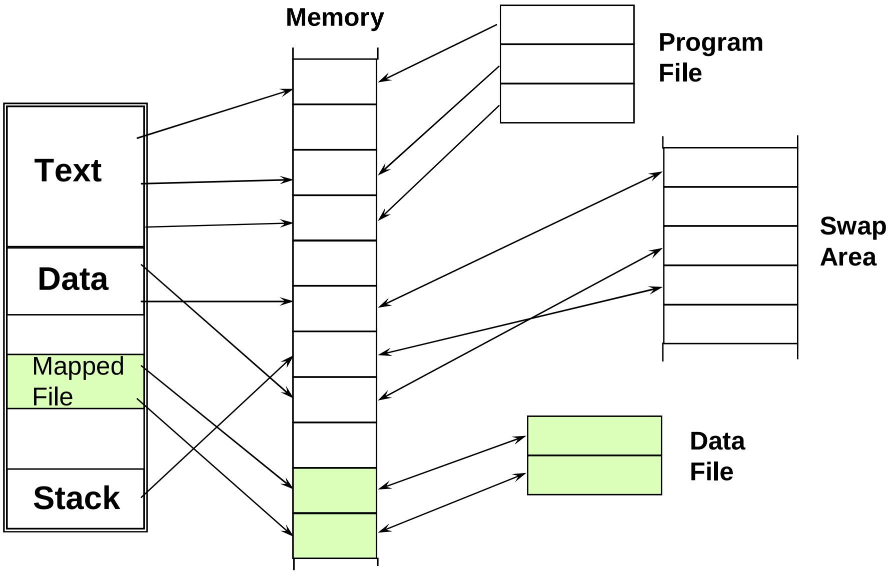
The writer program creates the segment, attaches to it and then treats it as a large array of integers. The array is filled with the numbers 1 to 8192.

The reader program uses the shared-memory key to get the id of the segment so that it can attach to the segment. The reader then prints out the array.

Writer.c

```
void main(void) {
    createSharedMemory();
    attachToMemory();
    writeToMemory();
}
void createSharedMemory(void) {
    shmid = shmget (KEY, SIZE * sizeof(int), IPC_CREAT | 0666);
    if (shmid < 0) perror("Can't create segment");
}
void attachToMemory(void) {
    shmptr = (int *) shmat(shmid, NULL, 0);
    if (shmptr == NULL) perror("Can't attach");
}
void writeToMemory(void) {
    for (i = 0; i < SIZE; i++)
        shmptr[i] = i;
}
```

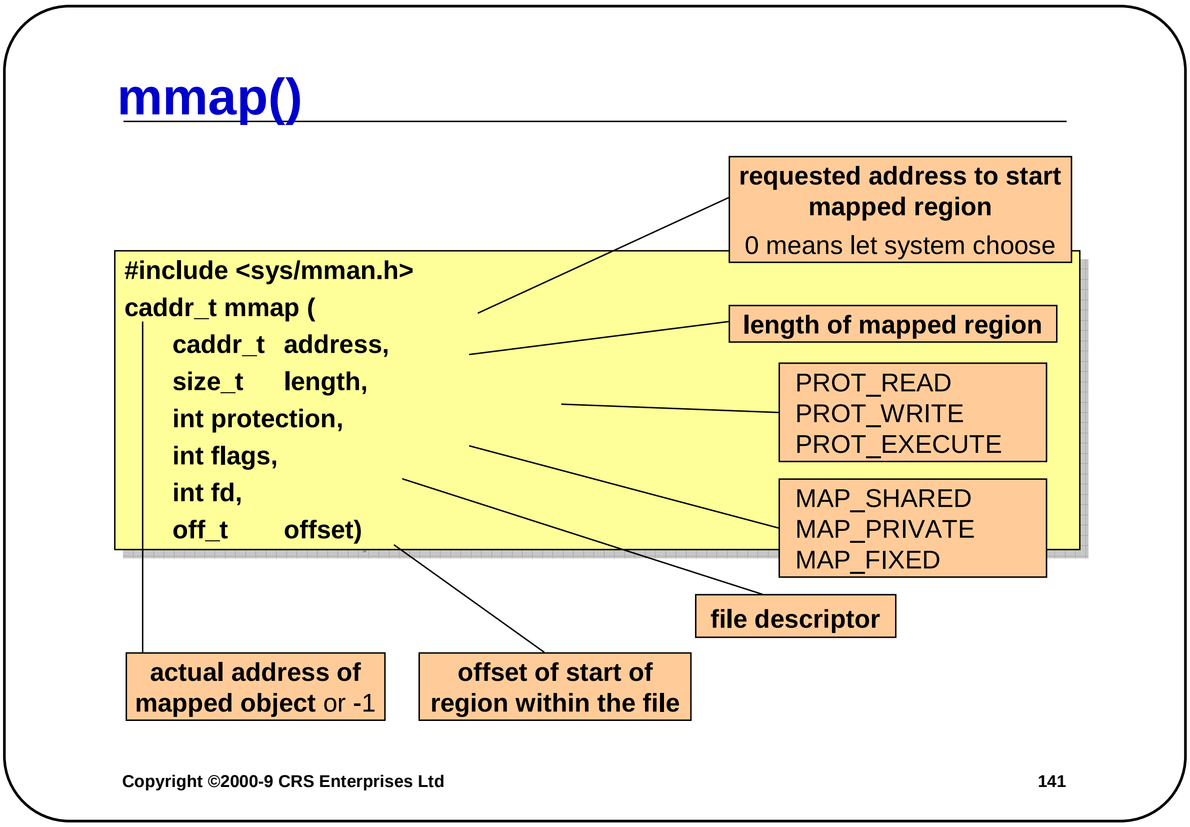
Memory Mapped Files



From the programmer's point of view, the most visible aspect of the new VM system is the ability to incorporate files into a process's virtual memory, so that their contents become immediately available through virtual addresses. A file is "mapped" into an unused part of the virtual address space, and then its contents are accessible to the process without the need for read() and write() calls. When a virtual address in this "region" is accessed, the appropriate page from the file is copied into memory in exactly the same way as other memory is copied in from the swap area on a page fault.

This method has been extended as the main means of transferring data and instructions to and from files. When a program is executed, using the exec() call, it is only necessary to set up the mapping from the program file into the process's address space. When the instructions are executed, the page-fault mechanism ensures that the pages are read into memory, so only those pages required by the program are copied in.

The new VM system also allows for sophisticated memory sharing between processes at a page level. Sharing can be on a true "shared" basis, with both processes able to write to the memory. Alternatively, processes can have read-only access or copy-on-write access, where a process that updates the memory receives its own private copy.



mmap()

```
#include <sys/mman.h>
caddr_t mmap (
    caddr_t  address,
    size_t   length,
    int protection,
    int flags,
    int fd,
    off_t    offset)
```

requested address to start mapped region
0 means let system choose
length of mapped region
PROT_READ
PROT_WRITE
PROT_EXECUTE
MAP_SHARED
MAP_PRIVATE
MAP_FIXED
file descriptor
actual address of mapped object or -1
offset of start of region within the file

The mmap() system call provides the programmer's basic interface to the VM system. mmap() establishes a mapping from a "memory object", such as a file or device, to a process's address space. After mmap() returns, the memory object is accessible through virtual addresses in the same way as "main" memory.

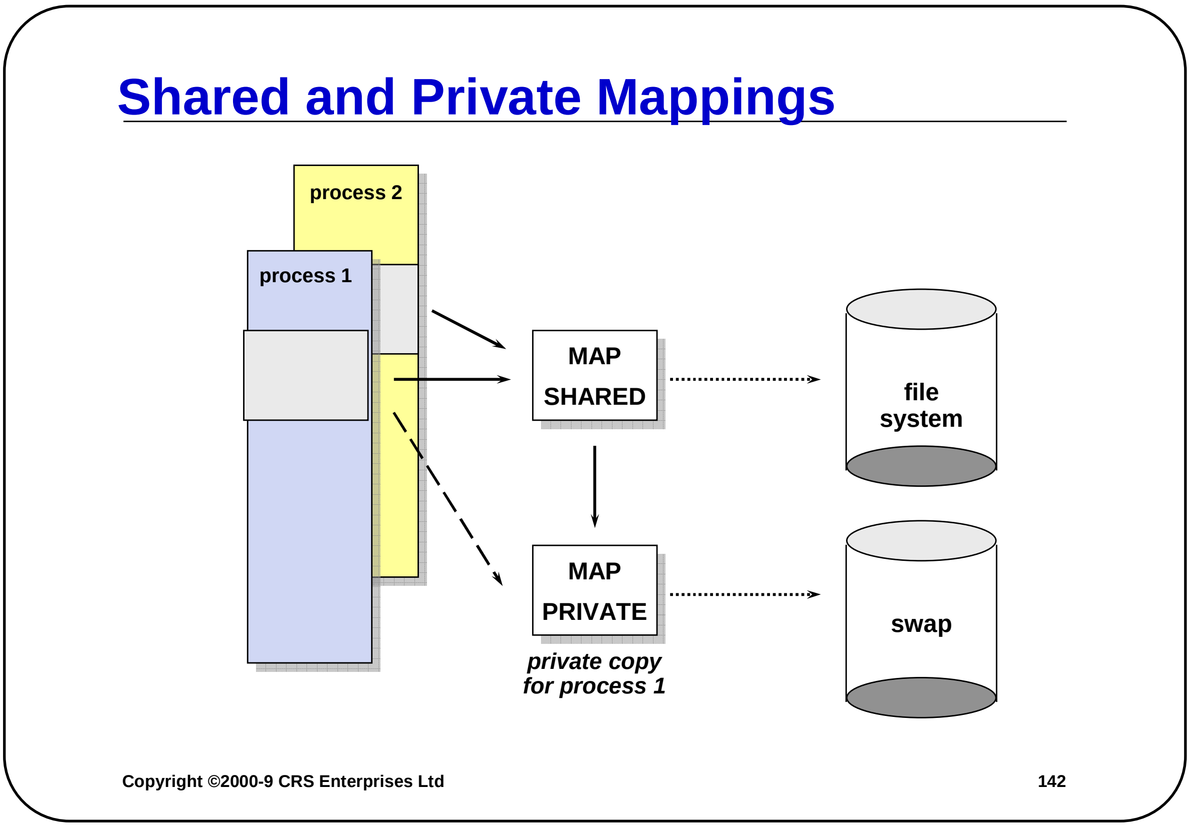
The virtual address at which the memory object is mapped is returned by mmap() on success; otherwise a value of -1 (cast to type caddr_t) is returned.

The programmer may specify a preferred virtual address from the process's virtual-address space at which to start the mapping, through the addr parameter. If so, the system makes every effort to map at that address. Most often, however, programmers are content to let the system choose the best address for the mapping and specify an addr parameter of 0. This is the preferred practice, as it allows the virtual-memory system to make best use of the available virtual-address space, ensuring that enough contiguous virtual memory is available for the mapping.

The starting point of the mapped area within the memory object is represented by the off parameter. This must be aligned on a page boundary. The size of the mapped area is specified by the len parameter. This need not be a multiple of the system page size, but any residual partial pages will be mapped.

Memory beyond the end of the mapped object, up to the page boundary, is zero filled.

The object being mapped must be accessible through a descriptor, so it must previously have been opened using open(). Once the mapping has been established, the object can be closed. This prevents access through the descriptor (for example with read() and write()), but allows access through the virtual addresses.



Shared and Private Mappings

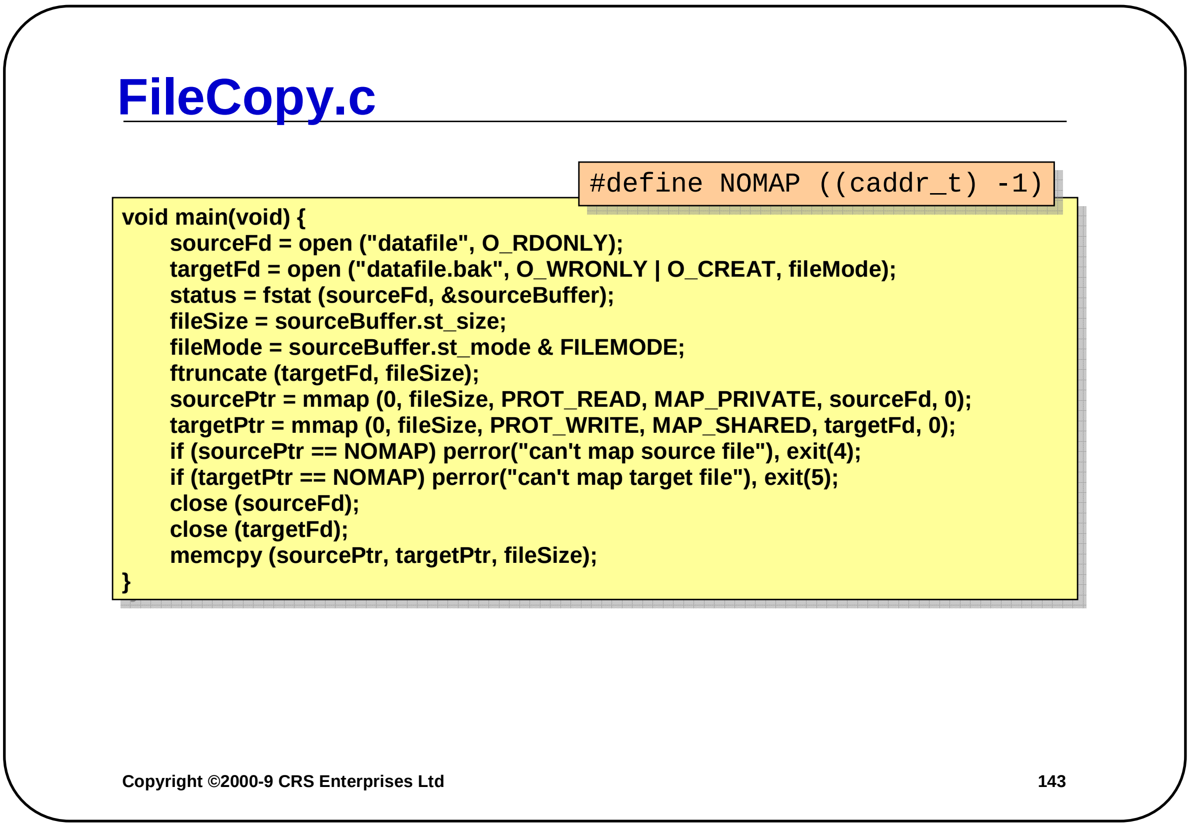
private copy for process 1

Copyright ©2000-9 CRS Enterprises Ltd 142

The mmap() system provides the MAP_PRIVATE and MAP_SHARED options for choosing a memory mapping. It is important that you understand the difference between these options.

Use MAP_SHARED if you want to share the changes you make to the mapped object with other processes. Objects mapped in this way are backed up on the files themselves. If multiple processes map the same file and wish to share changes made amongst themselves, then they must all use MAP_SHARED.

If you do not want to share the changes made to the pages in your mapping, you can use MAP_PRIVATE. Any pages that you change, having mapped them using MAP_PRIVATE, are backed up on the swap area. The changes are visible to your process only, and will not survive beyond the end of the process.



FileCopy.c

```
#define NOMAP ((caddr_t) -1)
```

```
void main(void) {
        sourceFd = open ("datafile", O_RDONLY);
        targetFd = open ("datafile.bak", O_WRONLY | O_CREAT, fileMode);
        status = fstat (sourceFd, &sourceBuffer);
        fileSize = sourceBuffer.st_size;
        fileMode = sourceBuffer.st_mode & FILEMODE;
        ftruncate (targetFd, fileSize);
        sourcePtr = mmap (0, fileSize, PROT_READ, MAP_PRIVATE, sourceFd, 0);
        targetPtr = mmap (0, fileSize, PROT_WRITE, MAP_SHARED, targetFd, 0);
        if (sourcePtr == NOMAP) perror("can't map source file"), exit(4);
        if (targetPtr == NOMAP) perror("can't map target file"), exit(5);
        close (sourceFd);
        close (targetFd);
        memcpy (sourcePtr, targetPtr, fileSize);
}
```

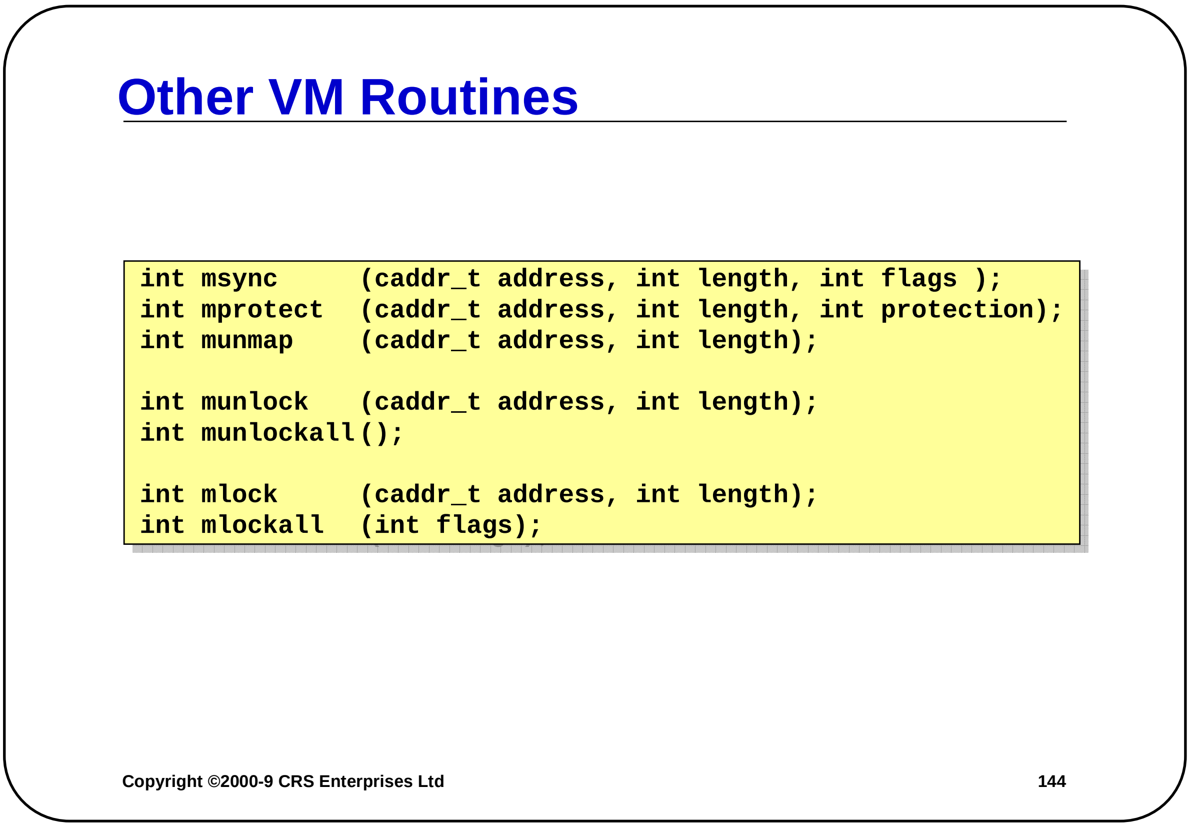
This example shows how to copy a binary file using memory mapping. The source file (datafile) and destination file (datafile.bak) are opened as normal and then both files are memory mapped. Once the files have been successfully mapped, there is no need for them to remain open, so we explicitly close them before copying any data. This makes sure we do not try to use read() and write().

Note that the MAP_SHARED option is used for the target mapping, even though we do not intend to share this mapping with other processes. This is because the changes to our mapped memory will be written back to file if we map using this option. PROT_READ and PROT_WRITE give read and write access to the mappings. The memcpy() function is used to copy the binary data (with embedded null characters).

ftruncate() ensures that the source and destination mapped regions are the same size, and that the files backing them are also the same size.

fstat() is used to extract the size and mode information from the datafile.

memcpy() is used to perform the memory to memory transfer of data. The MAP_SHARED semantics ensure that the changes made to the destination file will be recorded on disk.



Other VM Routines

```
int msync     (caddr_t address, int length, int flags );
int mprotect  (caddr_t address, int length, int protection);
int munmap    (caddr_t address, int length);

int munlock   (caddr_t address, int length);
int munlockall();

int mlock     (caddr_t address, int length);
int mlockall  (int flags);
```

msync(addr, len, flags) is used in applications that need to be sure of their data's integrity. It forces the data in the pages from addr to be flushed out for len bytes to the disk according to the value in flags:

MS_SYNC returns after the I/O operations are complete.

MS_ASYNC returns after the I/O operations are scheduled.

MS_INVALIDATE invalidates any cached copies of the data, forcing references to reread the data from backing store.

mprotect(addr, len, prot) allows an area of virtual memory to be protected using the PROT_READ, PROT_WRITE and PROT_EXECUTE flags, as described under mmap(). The protection specified must be "included" in the protections of the object backing the memory. For example, a read-only file mapped into memory cannot be set writeable using this method.

munmap(addr, len) removes the specified pages from the process's virtual address space.

mlock(addr, len) is used to lock the pages referenced by addresses in the range from addr to addr+len in memory.

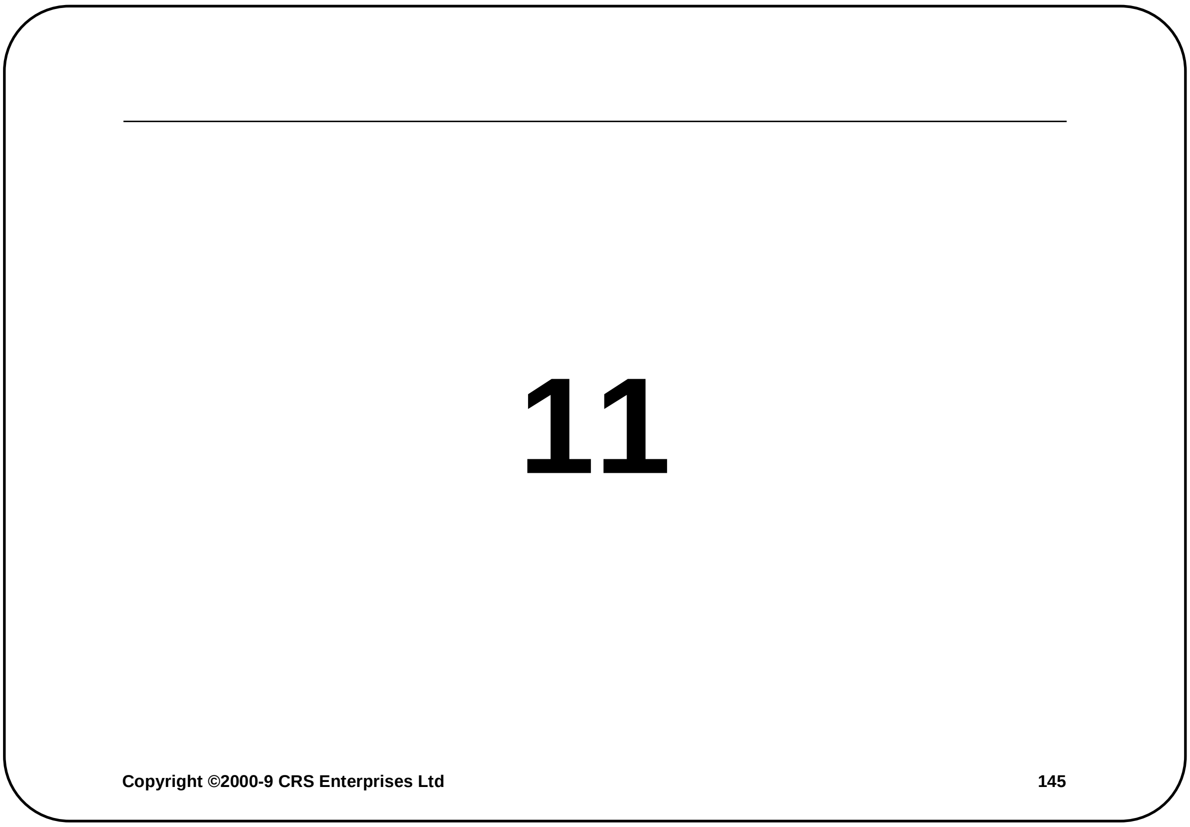
mlockall(flags) specifies that all pages in the process's address space should be locked in memory. The flags parameter is one or both of the following:

MCL_CURRENT locks all current mappings in memory.

MCL_FUTURE locks all future mappings in memory.

munlock(addr, len) removes the locks on the specified pages.

munlockall() removes all locks on pages in the process's address space.

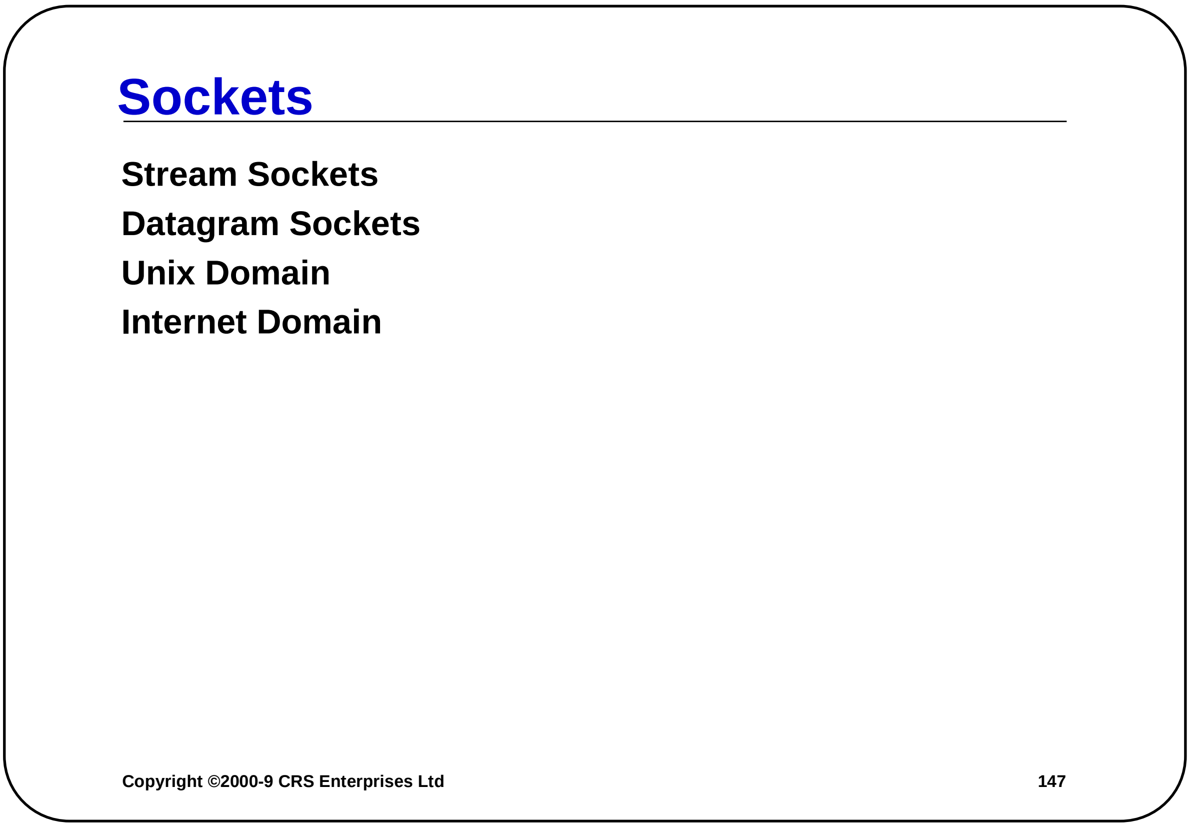


11

Introduction To Sockets

11

146



Sockets

Stream Sockets

Datagram Sockets

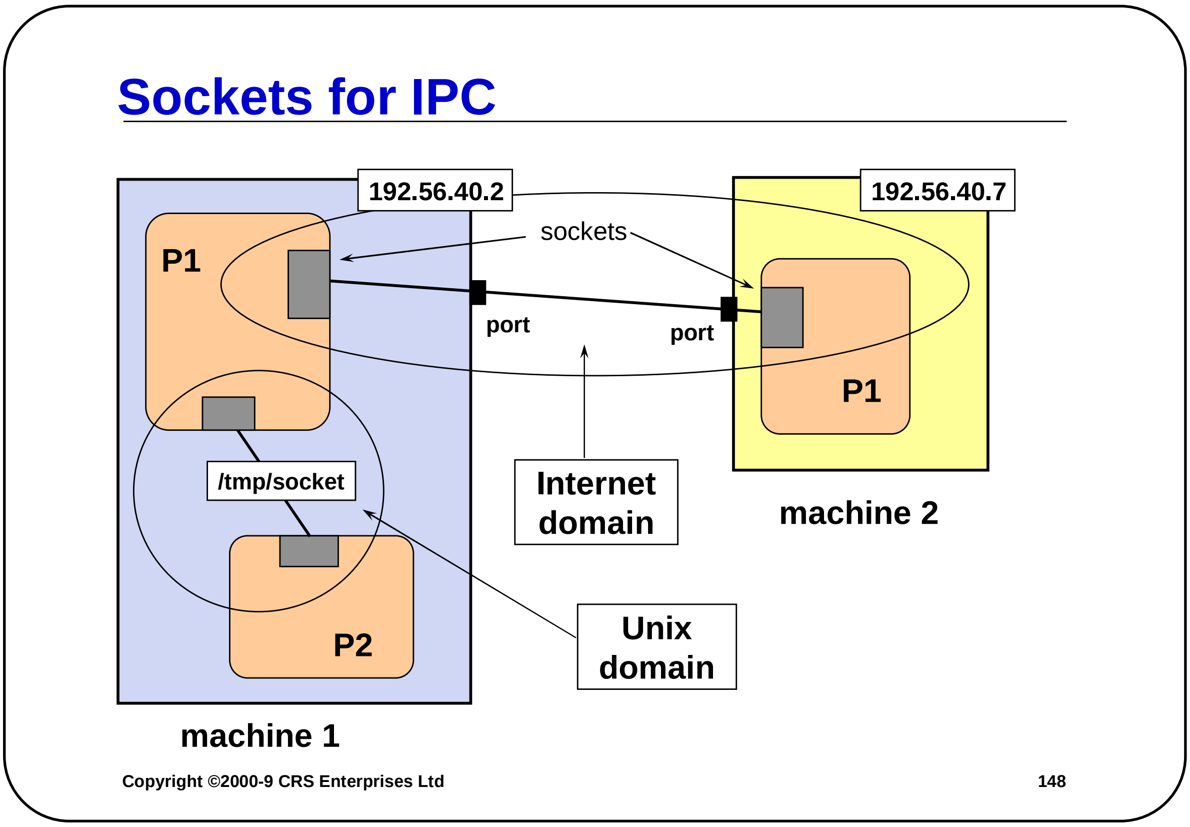
Unix Domain

Internet Domain

This chapter provides an introduction to sockets as a mechanism for IPC. There are two types of socket - Stream Sockets and Datagram Sockets. Stream Sockets are similar to named pipes; Datagram Sockets are similar to message queues.

Sockets are unique as an IPC method in that they support communications between processes that have the same API. The processes can be on the same machine (Unix domain) or on different machines (Internet domain).

The basics of how sockets work will be described, but it is beyond the scope of the course to cover inter-machine communication fully, so examples will relate to communication between local processes.



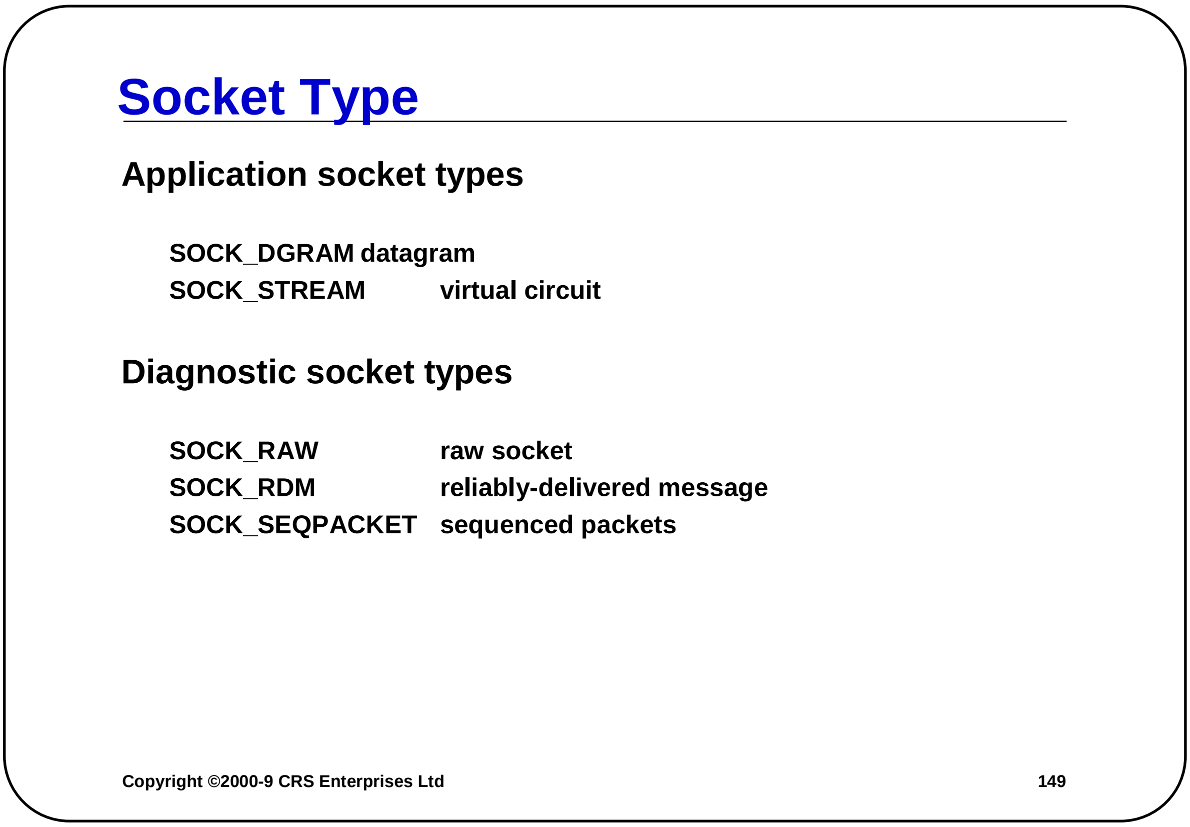
Sockets for IPC

A socket is an endpoint of a communications link between processes. When a process wishes to exchange information with another process, it sends or receives data through a socket.

Sockets in themselves do not hold enough information to allow the communication to be specified fully; they must operate within a particular communications domain. Different communications domains are used to allow communications between processes on the same machine or on different machines.

To transfer data in the Unix domain, processes must use a common filename (as with named pipes). In the Internet domain it is not possible to use filenames because two separate Unix systems are involved; instead we use a port number and an Internet number.

The diagram shows a particular setup, where process P1 on machine 1 has established links through sockets with process P2 on machine 1 and process P1 on machine 2. The first of these uses the Unix domain; the second uses the Internet domain.



Socket Type

Application socket types

SOCK_DGRAM	**datagram**
SOCK_STREAM	**virtual circuit**

Diagnostic socket types

SOCK_RAW	**raw socket**
SOCK_RDM	**reliably-delivered message**
SOCK_SEQPACKET	**sequenced packets**

A socket has a type, which is used to describe how data is transferred. Several "standard" types have been defined, although not all types will always be available.

SOCK_DGRAM Datagram sockets. Data is sent in packets, and there is no guarantee of delivery. Packets may be lost, duplicated or arrive out of order.

SOCK_STREAM Stream sockets. These implement a virtual circuit, i.e. a full duplex connection, where data is transferred reliably without record boundaries. Stream sockets require a connection to be established before data may be transferred. If the connection is broken, the communicating processes will be informed.

SOCK_RAW Raw sockets. These are a special case, designed to provide direct access to the internal networking software. They are nearly always used for debugging protocol implementations.

SOCK_RDM Reliably-delivered messages. Data is sent in packets, but is guaranteed to arrive safely and in order.

SOCK_SEQPACKET Sequenced packet. A connection-oriented reliable link, where packets are sent and delivered in sequence (with no duplicates) and the sender is notified if any messages are lost.

The socket subsystem allows new socket types to be added, but this is a complex operation.

Communications Domain

Unix	**AF_UNIX**
Internet	**AF_INET**
Xerox NS	**AF_NS**
IBM SNA	**AF_SNA**
CCITT X.25	**AF_X25**
AppleTalk	**AF_APPLETALK**

Sockets operate within communications domains. A communications domain defines how the socket is named (so that unconnected processes may communicate through a socket) and which communications protocols are used to effect the transfer of the data.

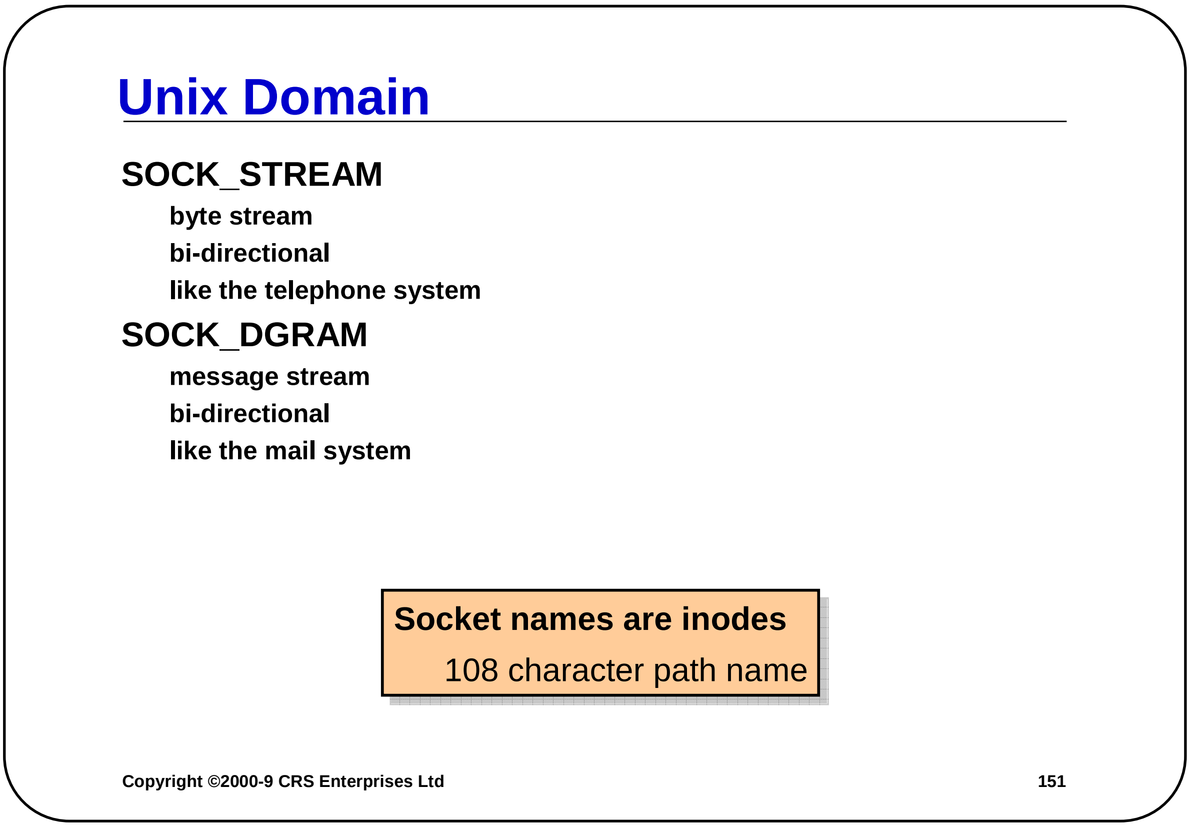
Most implementations of Unix that include sockets support two standard domains:

Unix used with processes residing on the same machine.

Internet used with processes residing on different machines

Other domains have been implemented, most notably to support XNS (Xerox) protocols.

It is possible, though difficult, to add support for new communications domains.



Unix Domain

SOCK_STREAM

- byte stream
- bi-directional
- like the telephone system

SOCK_DGRAM

- message stream
- bi-directional
- like the mail system

Socket names are inodes

108 character path name

151

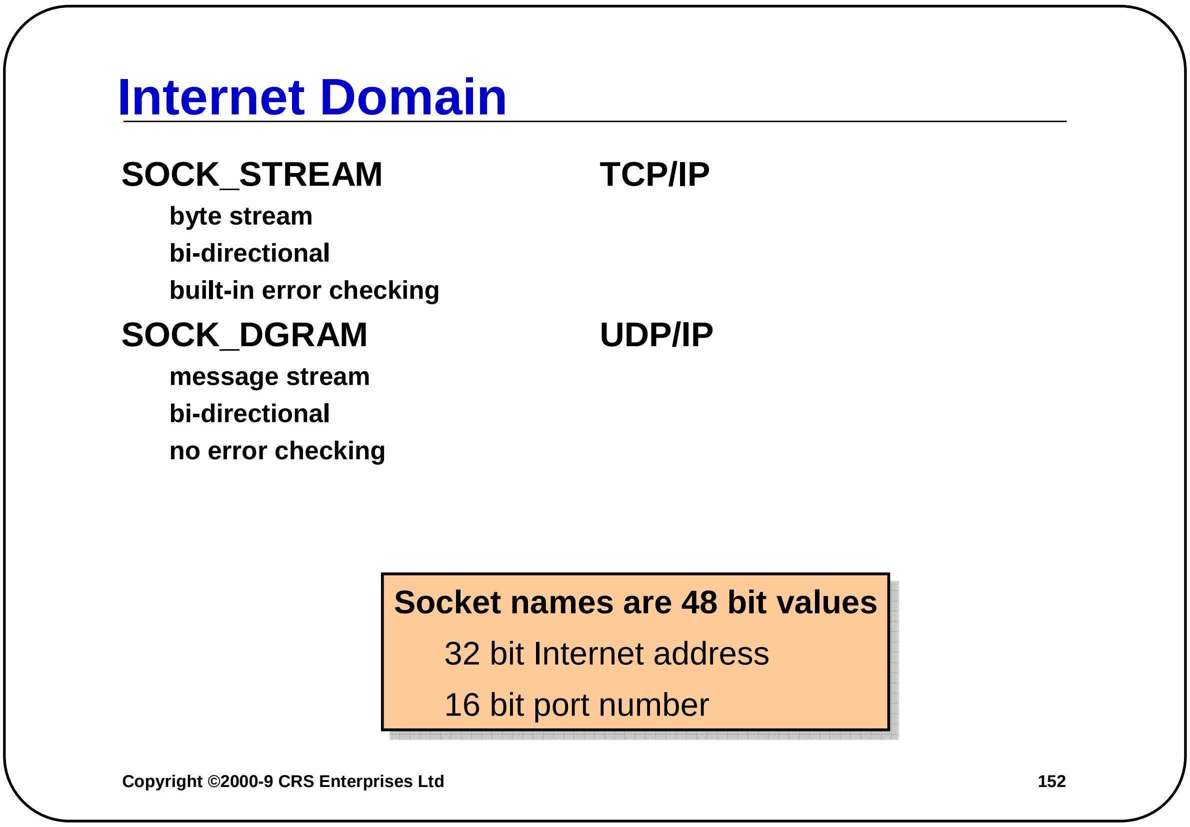
In the Unix domain, SOCK_STREAM and SOCK_DGRAM type sockets are supported.

SOCK_STREAM provides a bi-directional, byte-stream-oriented link between processes. This is similar to the way traditional pipes have worked, except that stream sockets support full-duplex links. In fact, pipes are often implemented using the sockets facility on systems where sockets are available.

SOCK_DGRAM provides a bi-directional, datagram link where data is exchanged with message boundaries preserved.

Socket names in the Unix domains are strings and are interpreted in the file-system name space. It is possible to see information on a socket using ls -l; the first character of the output will be "s".

```
$ ls -l /tmp
srw-rw-r-- ... /tmp/mysocket
```



Internet Domain

SOCK_STREAM **TCP/IP**

- **byte stream**
- **bi-directional**
- **built-in error checking**

SOCK_DGRAM **UDP/IP**

- **message stream**
- **bi-directional**
- **no error checking**

Socket names are 48 bit values

- 32 bit Internet address
- 16 bit port number

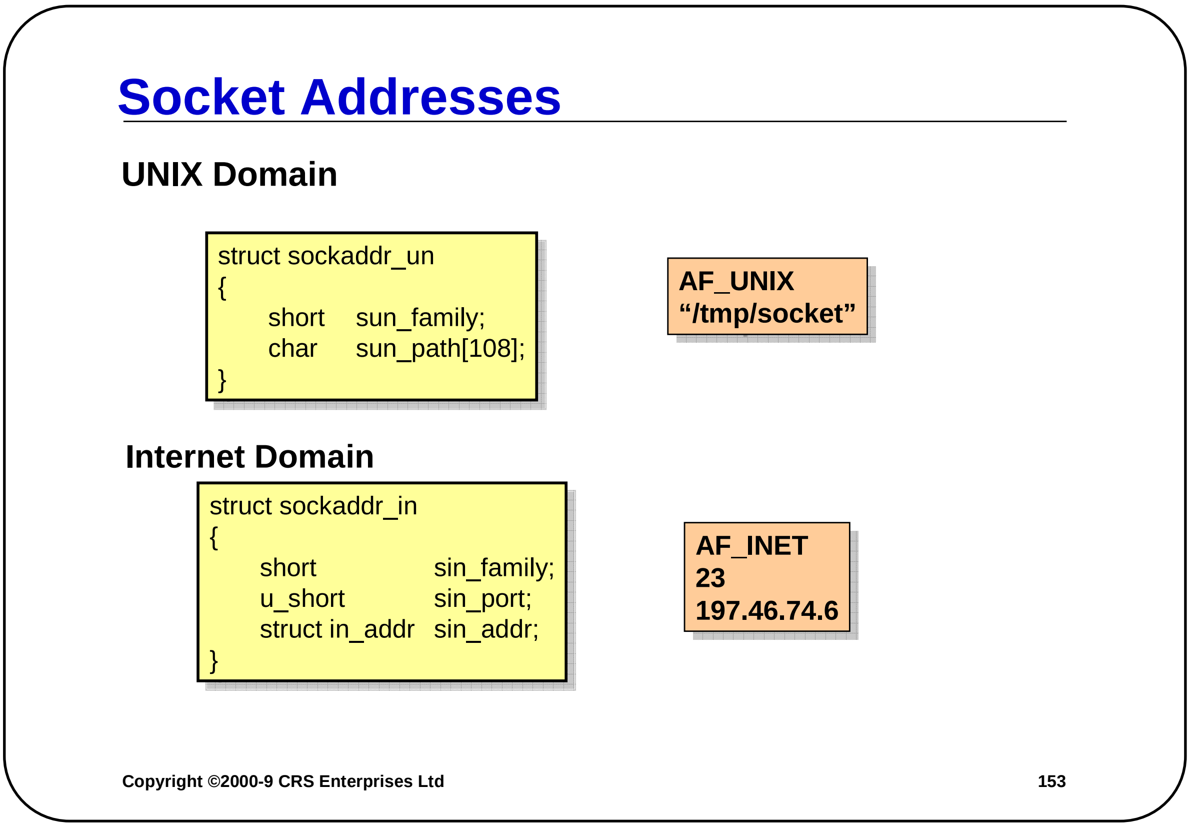
The Internet domain supports SOCK_STREAM and SOCK_DGRAM socket types. SOCK_STREAM provides communications between processes using TCP. SOCK_DGRAM provides communications using UDP. Both of the above use the facilities of IP.

Socket names in the Internet domain have two components:

A 32-bit Internet address, which identifies the machine.

A 16-bit port number, which identifies the socket on the host.

Many well-known TCP/IP services use specific port numbers. These numbers are reserved. A list of available port numbers is normally available, either as part of a system's documentation or as part of the official TCP/IP documentation.



Socket Addresses

UNIX Domain

```
struct sockaddr_un
{
        short    sun_family;
        char     sun_path[108];
}
```

AF_UNIX
“/tmp/socket”

Internet Domain

```
struct sockaddr_in
{
        short              sin_family;
        u_short            sin_port;
        struct in_addr     sin_addr;
}
```

AF_INET
23
197.46.74.6

Before sockets can be used as a form of inter-process communication, they must be given a name. In the Unix domain this consists of a pathname, but in the Internet domain a port number and Internet number are used. This information is packaged into a structure before it is used in the system calls. The Unix domain structure is called

struct sockaddr_un

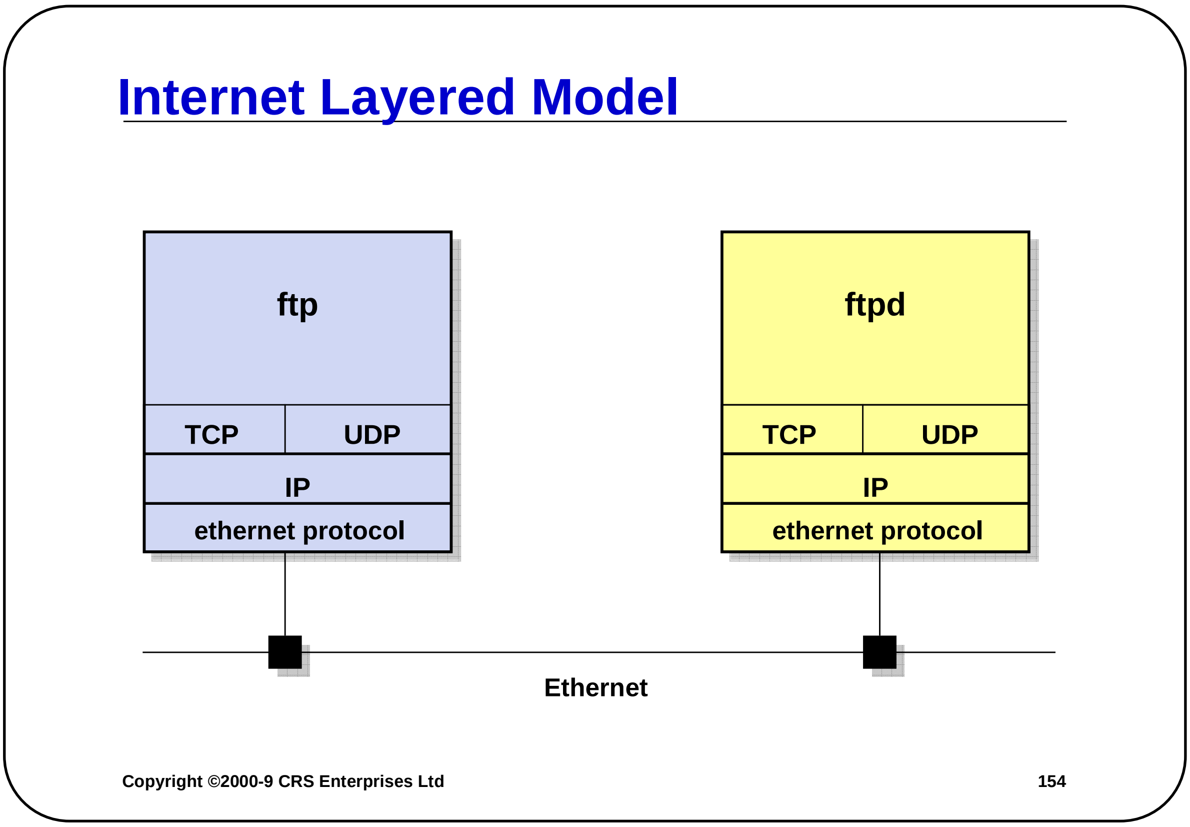
and the Internet domain structure is called

struct sockaddr_in

Sometimes

struct sockaddr

is used as a generic address structure.



Internet Layered Model

ftp

TCP UDP

IP

ethernet protocol

ftpd

TCP UDP

IP

ethernet protocol

Ethernet

Data transmission across the network uses layered protocols. At the top layer, an application program uses the socket family of system calls to communicate with the layer below (TCP or UDP). The lower levels are implemented as device drivers inside the kernel.

As an example, consider the ftp file transfer application. It uses the socket layer to pass large blocks of data to the TCP protocol (or in some implementations ftp uses the UDP protocol).

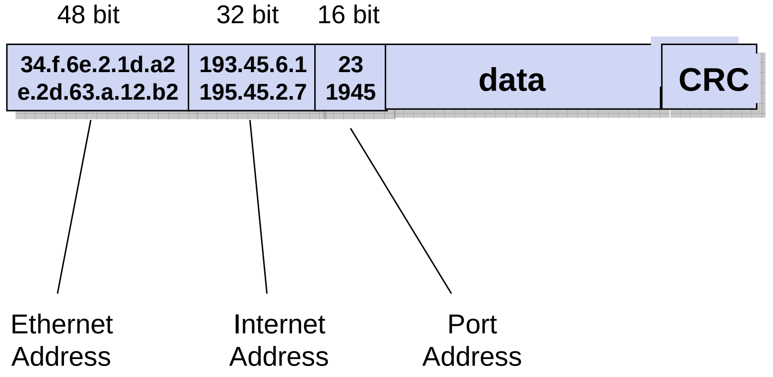
The TCP (or UDP) layer breaks these blocks up into packets of more manageable size. TCP also provides reliable data transmission by adding error checking at the receiver (not available with UDP). This layer is responsible for prepending the sender and receiver’s port numbers to the packets sent across the Internet.

The IP layer prepends the sender and receiver’s Internet addresses to the packets.

The Ethernet layer adds cyclic redundancy checking (CRC) to each packet and prepends the sender and receiver’s ethernet addresses to the packets.

Finally the hardware layer carries the digitised packet. The Ethernet uses the CSMA/CD protocol. CS is carrier sense. MA is multiple access. CD is collision detect.

Internet Packets



48 bit 32 bit 16 bit

34.f.6e.2.1d.a2 e.2d.63.a.12.b2	193.45.6.1 195.45.2.7	23 1945	data	CRC

Ethernet Address

Internet Address

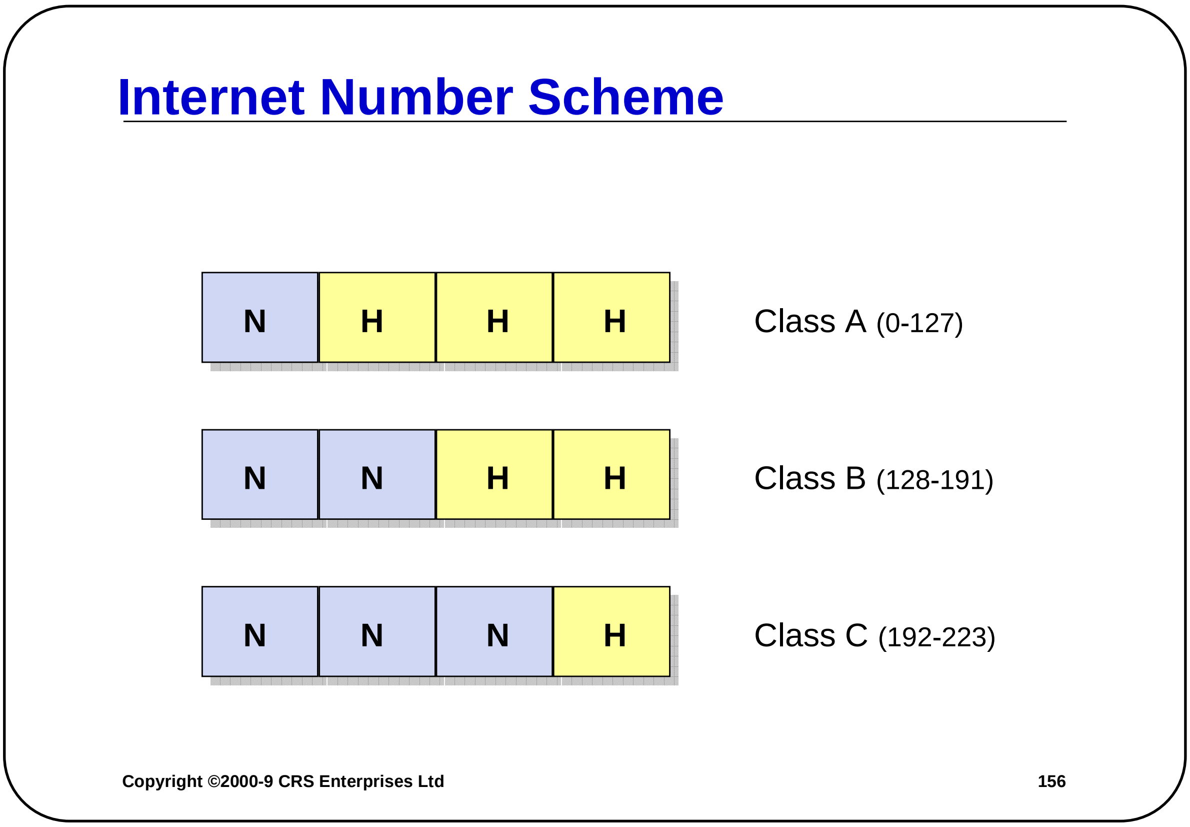
Port Address

By the time the digitised packets are sent out onto the Ethernet several addresses have been prepended to the data portion of the message. The addresses are prepended in pairs - one address for the sender and one for the receiver.

The Ethernet address can be decoded by the transceivers on the ethernet bus. This allows machines to communicate on a single network.

The Port number directs the message to a specific application on a machine.

The Internet address can be decoded by a router machine and the packet passed on across the Internet.



Internet Number Scheme

N	H	H	H	Class A (0-127)
N	N	H	H	Class B (128-191)
N	N	N	H	Class C (192-223)

156

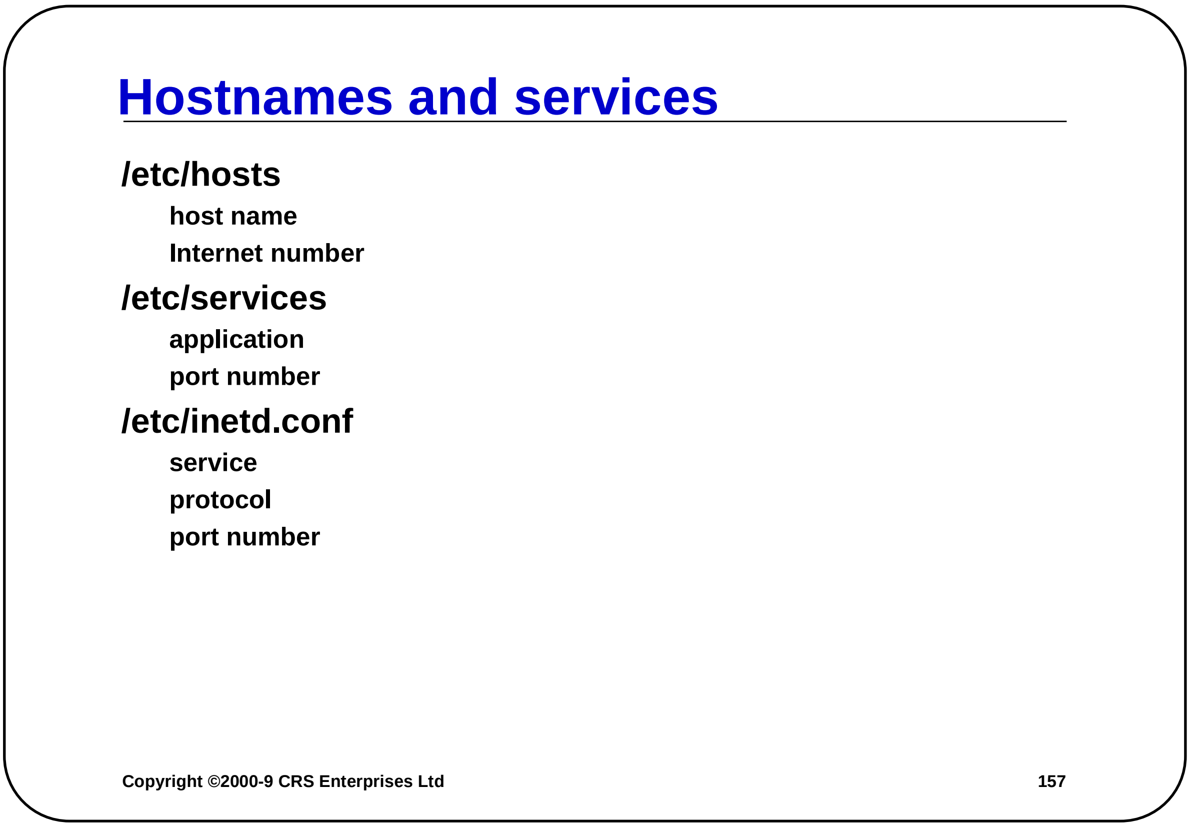
An Internet addresses is categorised as a class A, B or C addresses. An Internet address contains both a machine and a network address.

Each Internet address is 32 bits long, but each of the classes subdivide the 32 bit Internet address between network address and machine address in different ways.

Class A addresses use 8 bits to specify the network and 24 bits to identify the machine.

Class B addresses use 16 bits to specify the network and 16 bits to identify the machine.

Class A addresses use 24 bits to specify the network and 8 bits to identify the machine.



Hostnames and services

/etc/hosts

- host name
- Internet number

/etc/services

- application
- port number

/etc/inetd.conf

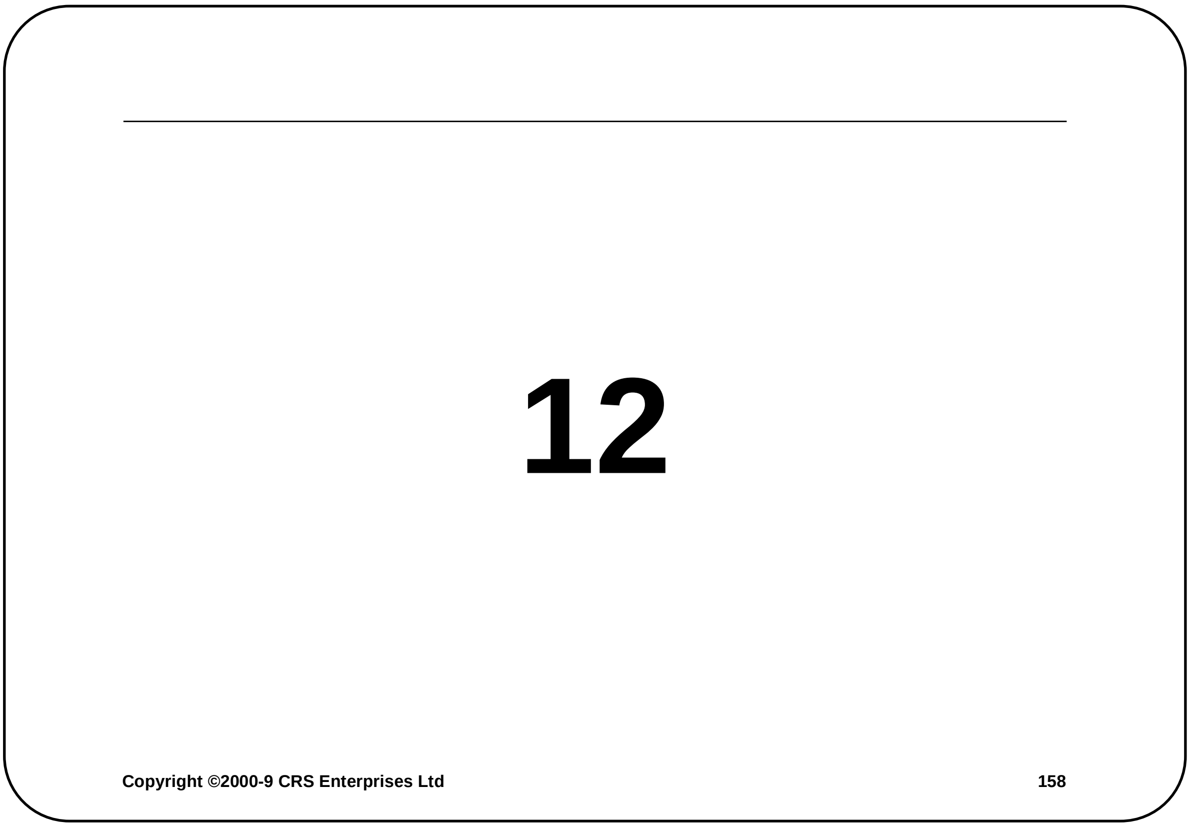
- service
- protocol
- port number

There are three important system administration files that are used in Internet communications.

The etc/hosts file provides a lookup table mapping machine host names to Internet numbers.

The /etc/services file provides a lookup table mapping Internet services (applications) to port numbers.

The /etc/inetd.conf file is a configuration file of services provided by the Internet daemon.



12

Unix Domain Sockets

12

Copyright ©2000-9 CRS Enterprises Ltd

159

Unix Domain Sockets

Stream Sockets

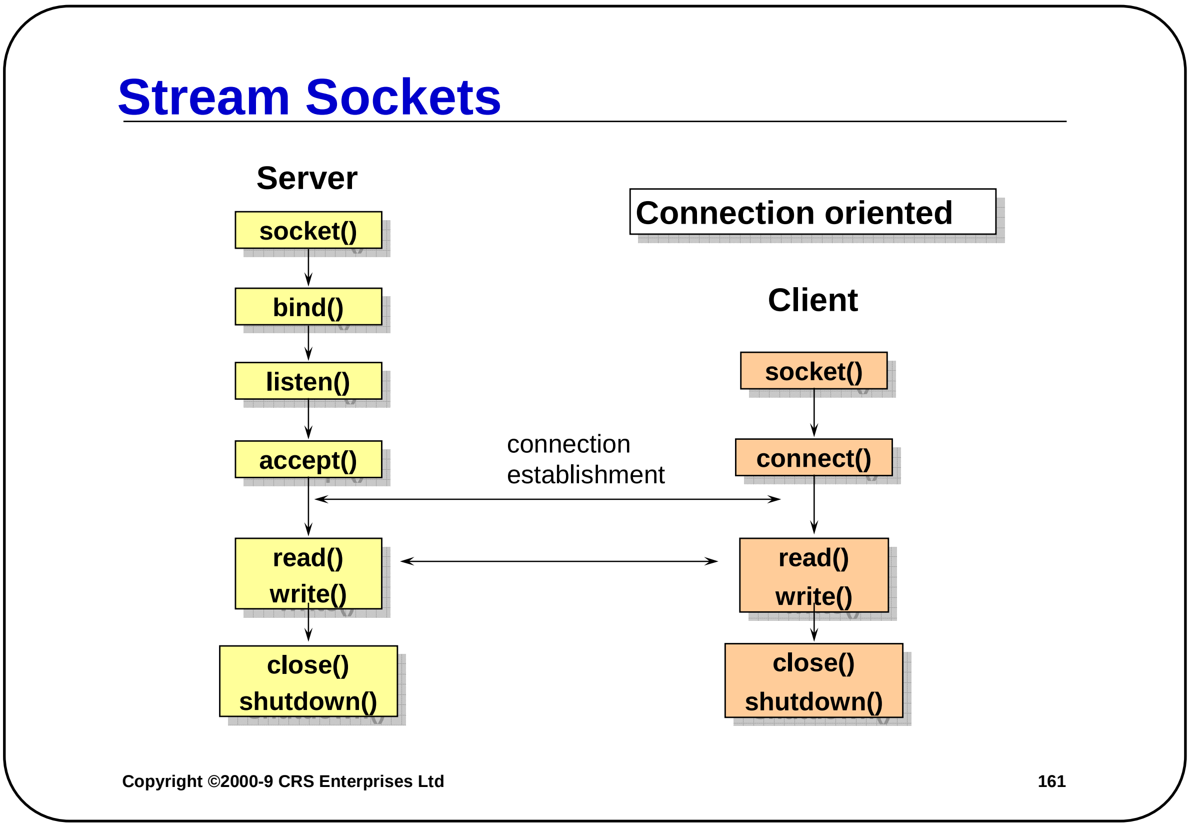
Datagram Sockets

System Calls

Kernel Data Structures

Examples

In this chapter we will look specifically at Unix domain sockets (Stream Sockets and Datagram Sockets). System calls and kernel data structures will be discussed and simple client/server examples presented.



Stream Sockets

Server

socket()

bind()

listen()

accept()

read()
write()

close()
shutdown()

Connection oriented

Client

socket()

connect()

connection
establishment

read()
write()

close()
shutdown()

Copyright ©2000-9 CRS Enterprises Ltd 161

Sockets are designed around the client/server model of IPC. Communication in this way is asymmetric; each end of the link must know its role. The stages involved in a conversation between a client and a server process are shown in the diagram.

The server must first create a socket and bind it to an address. The socket is created using the socket() system call, at which time the socket type and communications domain are specified. A socket is referenced using a file descriptor; bind() will associate the socket with a filename.

The next stage is to have the socket listen for connections from other processes. This is done with the listen() system call. listen() allows us to queue pending connection requests.

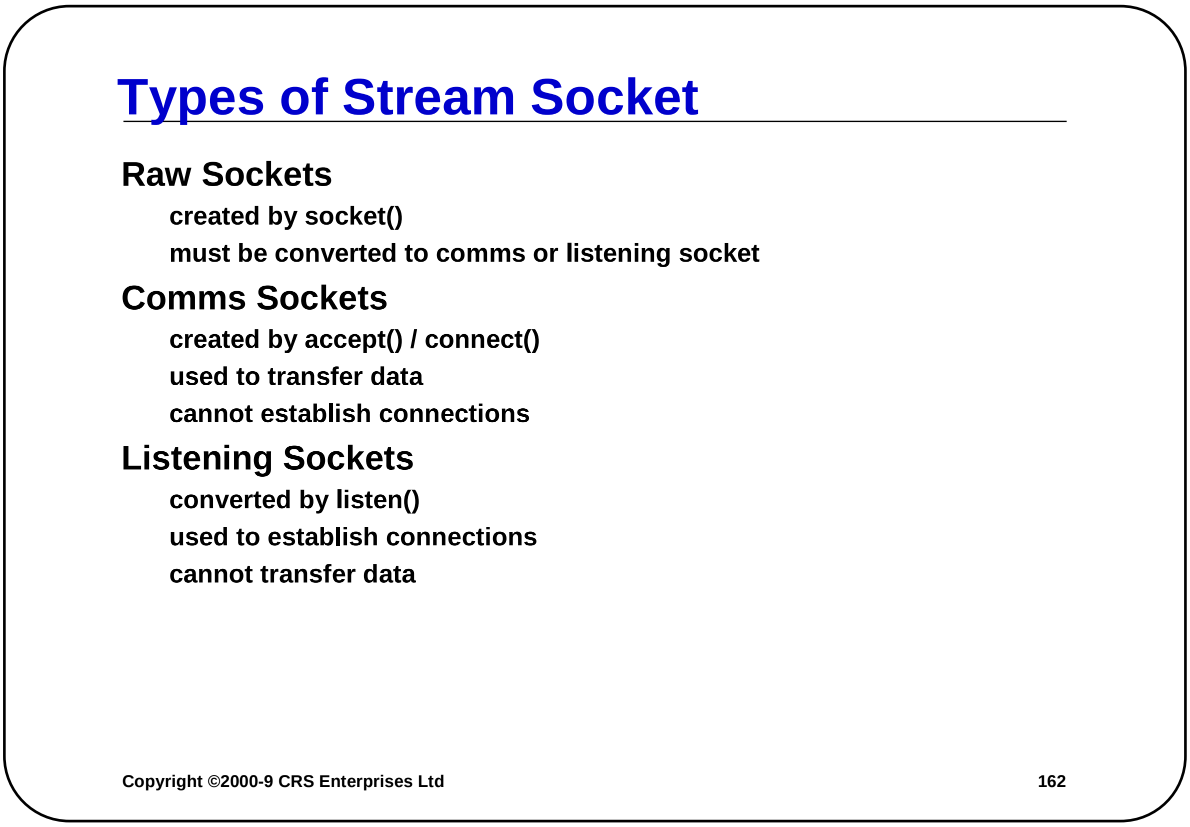
To make a connection, the server calls accept(). If there are no connection requests to be serviced, this call will normally cause the process to block, waiting for a request. If there are requests pending, a connection is made.

accept() returns a new socket descriptor, which is then used to transfer the data using read() and write(). The new descriptor is used for data transfers; the old descriptor is used to listen for further requests from other clients.

The client process also creates a socket using socket(). This socket is to be used to communicate with the server, but the client does not bind() to the socket.

To request a connection with the server process, the client process calls connect(), specifying the address of the socket to which the connection is required. If connect() returns successfully, a rendezvous has been made with the server's accept() call and the connection is established. Data may be transferred using read() and write().

To terminate a connection, either side calls close() or shutdown().



Types of Stream Socket

Raw Sockets

- **created by socket()**
- **must be converted to comms or listening socket**

Comms Sockets

- **created by accept() / connect()**
- **used to transfer data**
- **cannot establish connections**

Listening Sockets

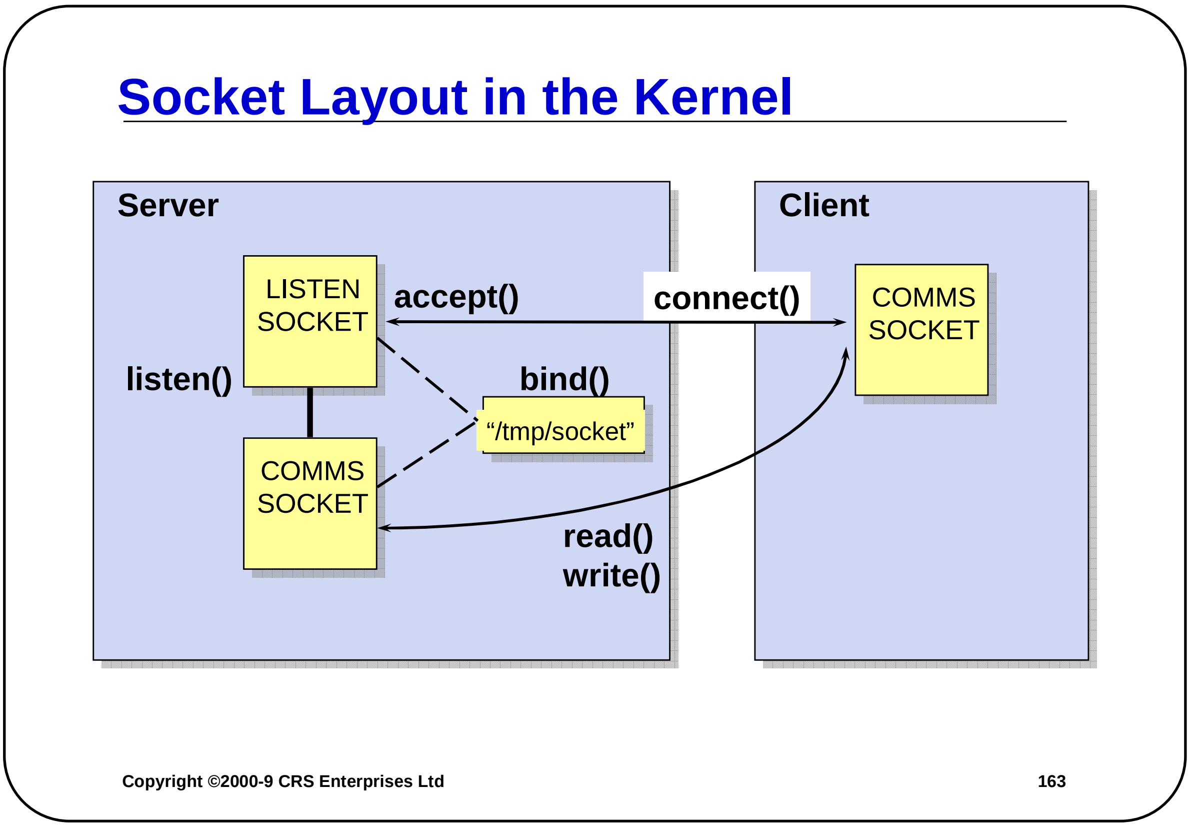
- **converted by listen()**
- **used to establish connections**
- **cannot transfer data**

162

On the server side a socket must be converted to a listening socket before it is useable. A listening socket is used to accept connections from a client process. The listen() system call converts a raw socket into a listening socket.

The listening socket on the server provides a rendezvous for client processes wishing to exchange messages with the server. However a listening socket is not itself capable of transmitting and receiving data. Before data transfers can take place a communications socket must be created at the server. This is achieved by the accept() system call.

On the client side a socket must be converted to a communications socket before it is useable. This is achieved by the connect() system call.



Socket Layout in the Kernel

Server

LISTEN SOCKET

accept()

connect()

COMMS SOCKET

listen()

bind()

"/tmp/socket"

COMMS SOCKET

read()
write()

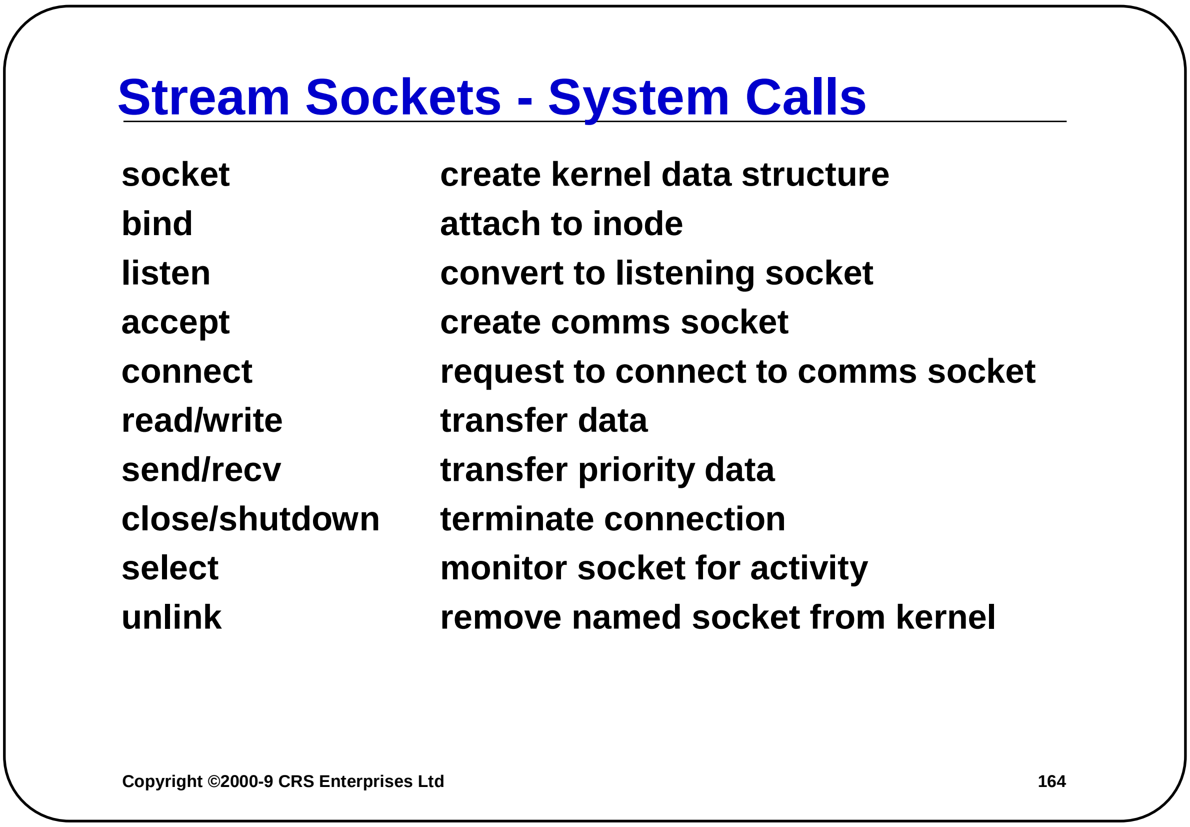
Client

This slide shows the various data structures constructed in the kernel as a result of the system calls.

On the server side the kernel maintains two sockets. The listening socket is named (associated with an inode by the bind() system call) to allow connections from clients. The comms socket is used for data transmissions. This socket does not have a name; it is referenced by a file descriptor.

The client has one comms socket and it is unnamed. The socket is referenced in all system calls by its file descriptor.

The client and server comms sockets are connected inside the kernel's address space by the connect() and accept() system calls. The comms sockets once connected, behave as a bi-directional pipe.



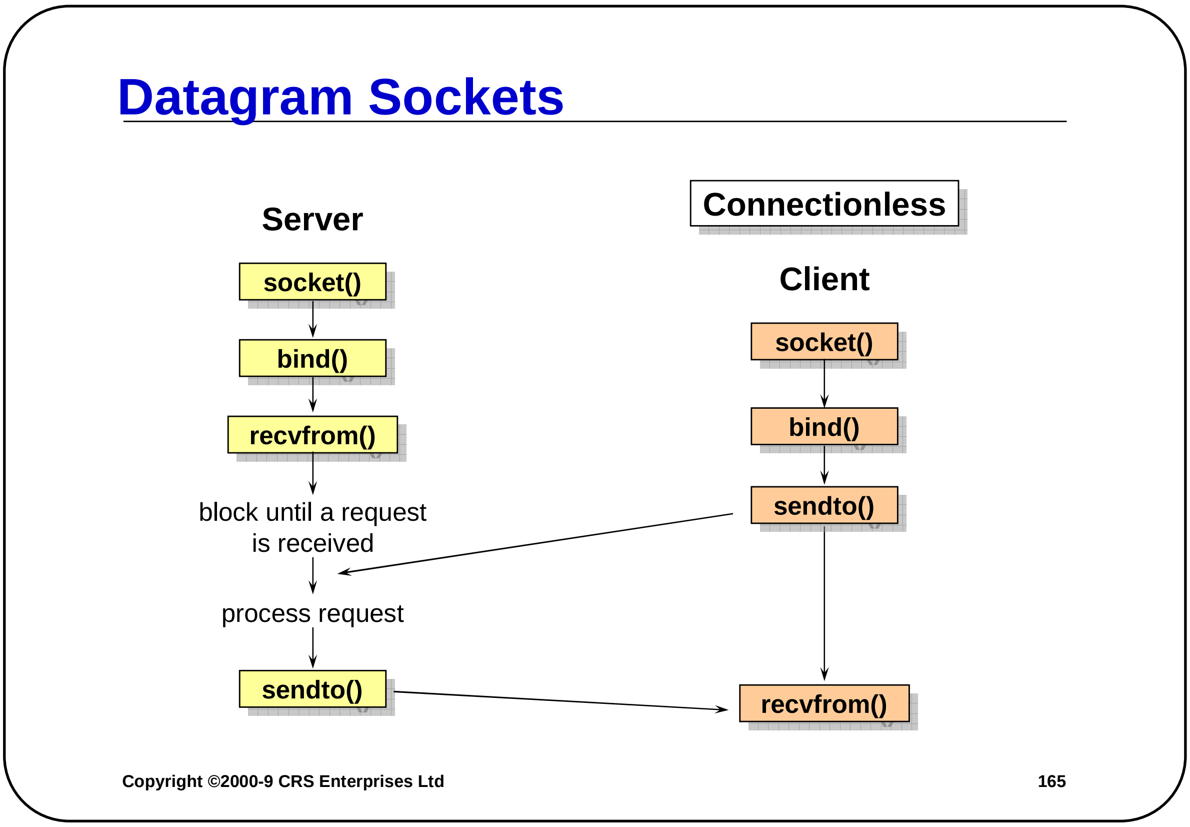
Stream Sockets - System Calls

socket	**create kernel data structure**
bind	**attach to inode**
listen	**convert to listening socket**
accept	**create comms socket**
connect	**request to connect to comms socket**
read/write	**transfer data**
send/recv	**transfer priority data**
close/shutdown	**terminate connection**
select	**monitor socket for activity**
unlink	**remove named socket from kernel**

There are numerous system calls used for Stream Sockets; we have discussed the most important in the preceding pages. The other calls are described below.

Stream Sockets behave very similarly to bi-directional pipes, but there is one important difference. Stream sockets can be programmed to send and receive priority data (sometimes referred to as out of band data). The send() and recv() system calls are provide this facility.

Communication on a sockets is normally terminated by the close system call. However, close() waits until all data transmission has been completed before removing the file descriptor from a process's open file table. The shutdown() system call can be used to terminate communication immediately. Both system calls are used to remove the comms socket structure from the kernel; the listening socket is removed by the standard unlink() system call.



Datagram Sockets

Connectionless

Server

socket()

bind()

recvfrom()

block until a request
is received

process request

sendto()

Client

socket()

bind()

sendto()

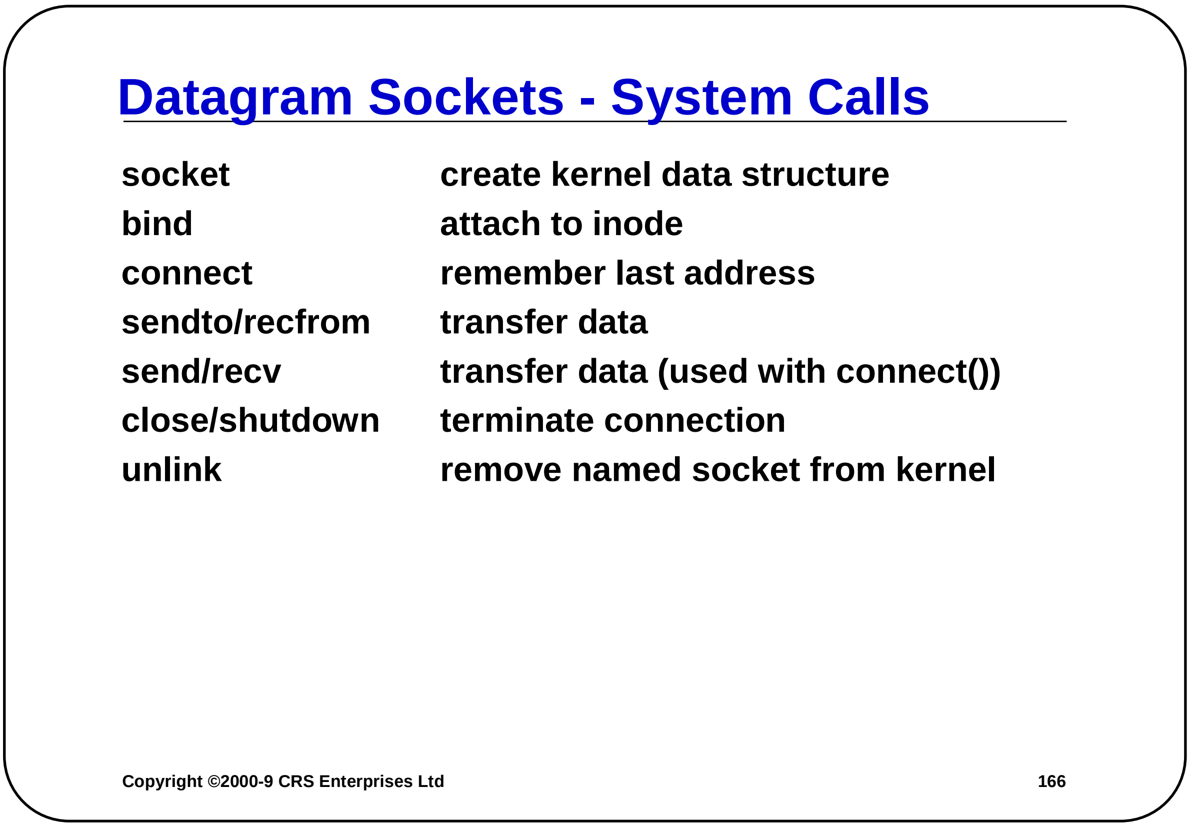
recvfrom()

Datagram sockets lead to a slightly different pattern of interaction between the client and server processes. With datagrams, the interaction is completely symmetric.

Both the client and server must create there own sockets and bind them to an address (a filename in the Unix domain). However, no firm connection is made in this case (the kernel does not maintain a link between the client and server socket structures); the server waits for a request by calling the recvfrom() system call. This blocks the process until a request comes in from some client. The request will contain the socket address of the client to let the server know where the datagram originated. A reply can be sent to the client using the sendto() call. Since there is no connection between the processes, there is no need to shut down the link.

Note:

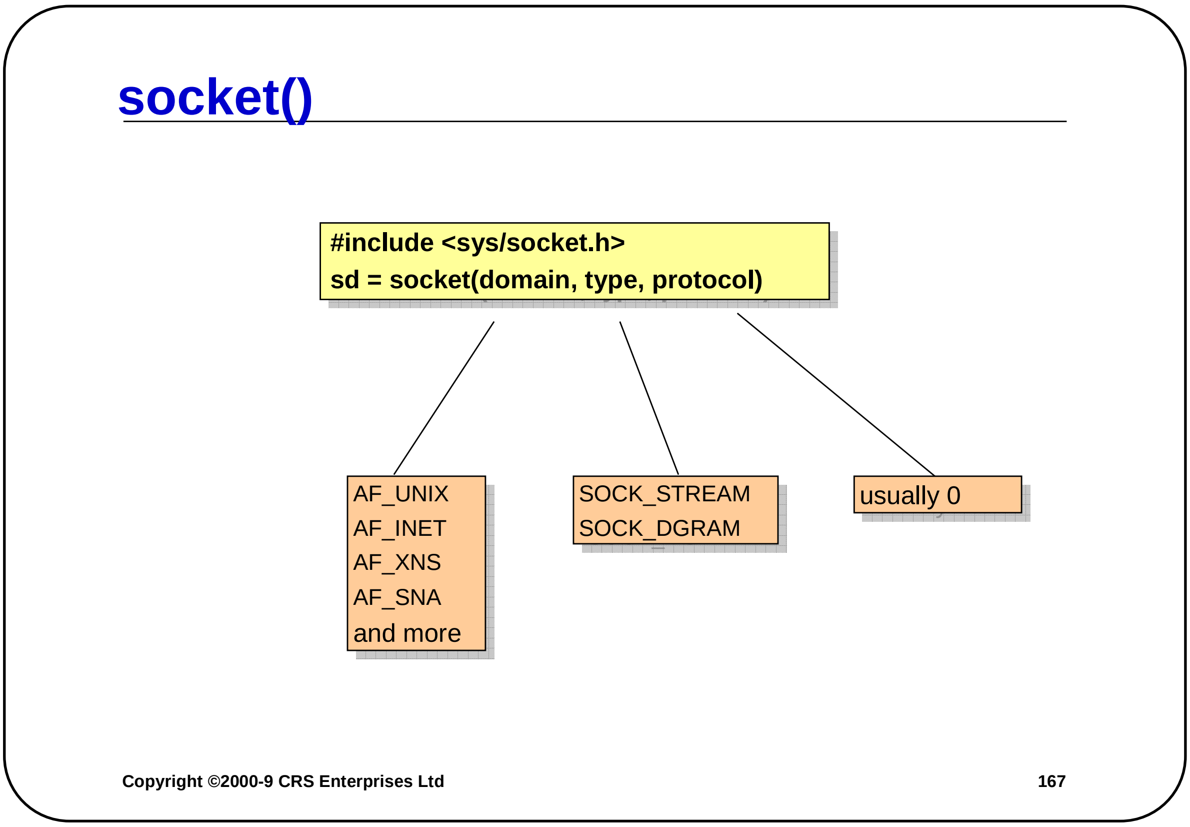
Do not use the read() and write() system calls with datagrams. Each datagram is processed once only by system calls. If you attempt to use read() on a datagram, the call will read the first part of the datagram and discard the remainder. The discarded data will be lost forever.



Datagram Sockets - System Calls

socket	create kernel data structure
bind	attach to inode
connect	remember last address
sendto/recfrom	transfer data
send/recv	transfer data (used with connect())
close/shutdown	terminate connection
unlink	remove named socket from kernel

There are fewer system calls used for Datagram Sockets because the communications mechanism is much simpler.



socket()

```
#include <sys/socket.h>
sd = socket(domain, type, protocol)
```

AF_UNIX
AF_INET
AF_XNS
AF_SNA
and more

SOCK_STREAM
SOCK_DGRAM

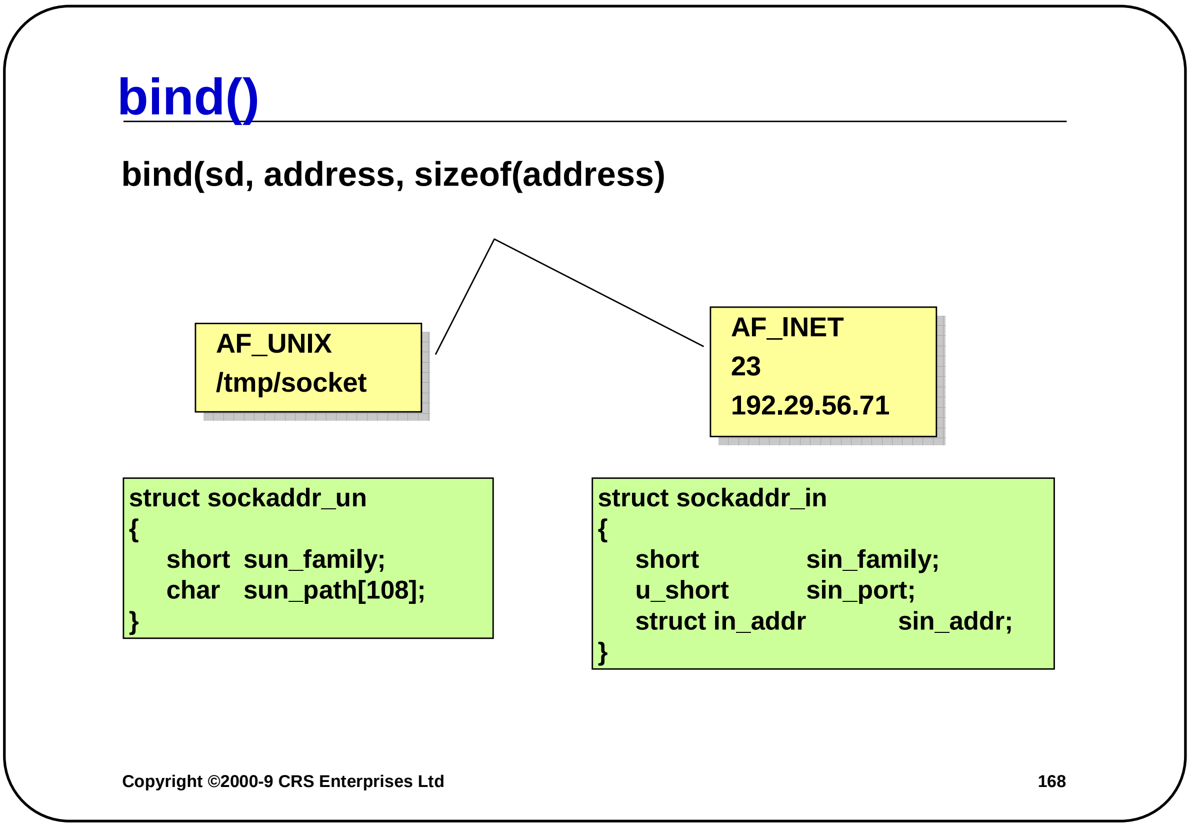
usually 0

Copyright ©2000-9 CRS Enterprises Ltd 167

The socket() system call creates a socket. The call simply creates the endpoint; it is not at this stage bound to any address or connected to any other socket.

It is necessary to specify the type of the socket and the domain in which the socket is to operate. Both of these can be specified as constants defined in the include file <sys/socket.h>.

In most cases, there will be a single protocol used when a particular type of socket is used in a particular domain (e.g. TCP used for a stream socket in the Internet domain). There may, however, be cases when there is a choice of protocols, and then a particular protocol should be selected when the socket is created. There are support routines to help with this, which are beyond the scope of this course. In most cases, the protocol parameter may safely be set to 0.



bind()

bind(sd, address, sizeof(address))

AF_UNIX
/tmp/socket

AF_INET
23
192.29.56.71

```
struct sockaddr_un
{
    short  sun_family;
    char   sun_path[108];
}
```

```
struct sockaddr_in
{
    short          sin_family;
    u_short        sin_port;
    struct in_addr         sin_addr;
}
```

168

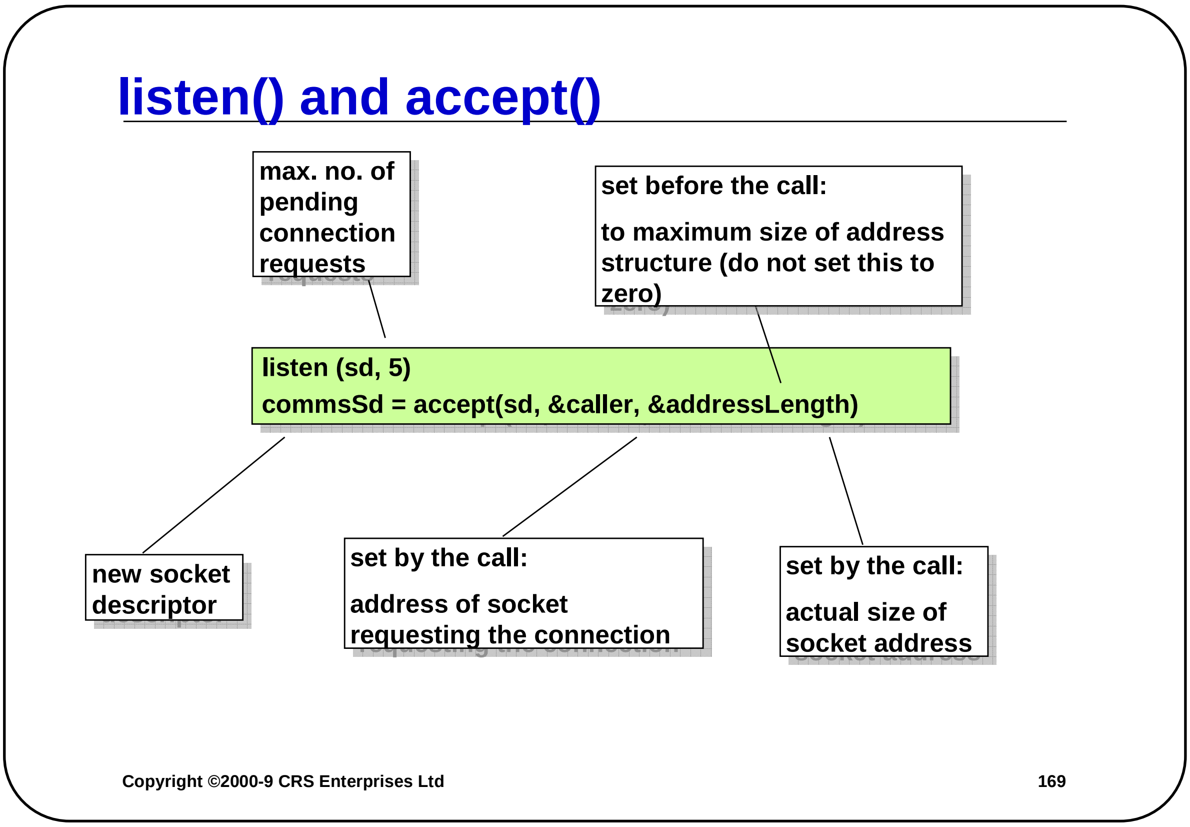
bind() is used to associate a name with a socket. The name will depend on the domain of the socket.

The name or address of the socket is specified using a generic structure of type struct sockaddr. The details of this structure is dependent on the addressing structure of the domain. For example, the Internet domain uses the 32-bit Internet address, plus the 16-bit port number. The Unix domain uses a pathname.

Sockets in the Unix domain are named with a structure of type struct sockaddr_un, as described in the slide. Pathnames of up to 108 characters may be specified.

Sockets in the Internet domain are named with a structure of type struct sockaddr_in, as described in the slide. This structure contains a Port number and an Internet address.

The bind() routine requires the name details to be passed by a pointer to the appropriate structure together with the actual length in bytes of the structure. This is a common way of passing information into (and around) the kernel.



listen() and accept()

max. no. of pending connection requests

set before the call:

to maximum size of address structure (do not set this to zero)

```
listen (sd, 5)
commsSd = accept(sd, &caller, &addressLength)
```

new socket descriptor

set by the call:

address of socket requesting the connection

set by the call:

actual size of socket address

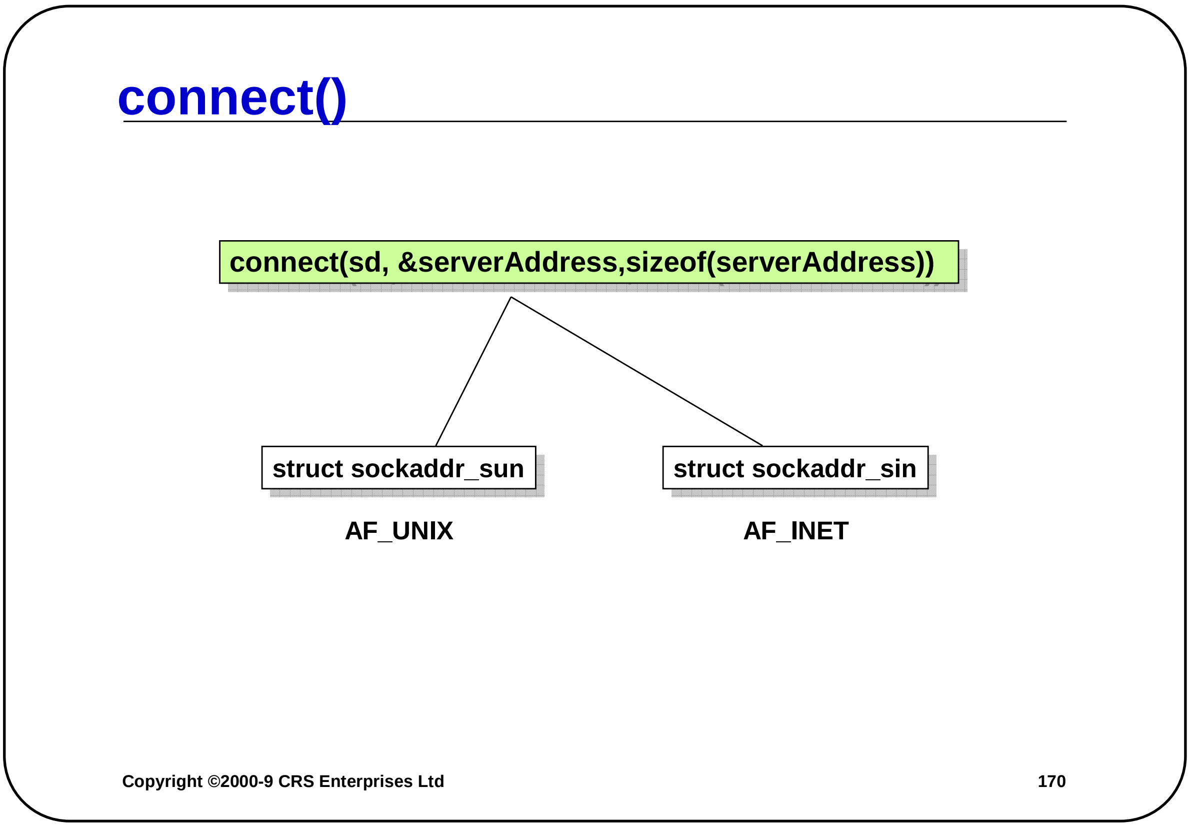
listen() is used to prepare a connection-oriented socket for accepting connections. The call sets up a queue onto which connection requests are placed. The maximum size of the queue is set here. Any connection requests arriving when the queue is full will be refused.

Establishing a connection between a client and a server process is done by a form of rendezvous. After the server has set up the characteristics of its socket, it waits for a connection request to come in. The accept() system call achieves this.

accept() causes the calling process to block until a connection request is received. If there are any pending requests in the backlog specified in the listen() call earlier, then the call returns immediately.

When a request is received, and the connection is made, accept() returns a new socket descriptor. The new socket is intended to be used for the communication between the two processes. The read() and write() calls can safely be used on the descriptor.

When a connection is made, accept() fills in the address of the "peer" socket in the address structure and returns its length in the addrlen parameter. This is so that the server knows which client it is dealing with.



connect()

connect(sd, &serverAddress,sizeof(serverAddress))

struct sockaddr_sun

AF_UNIX

struct sockaddr_sin

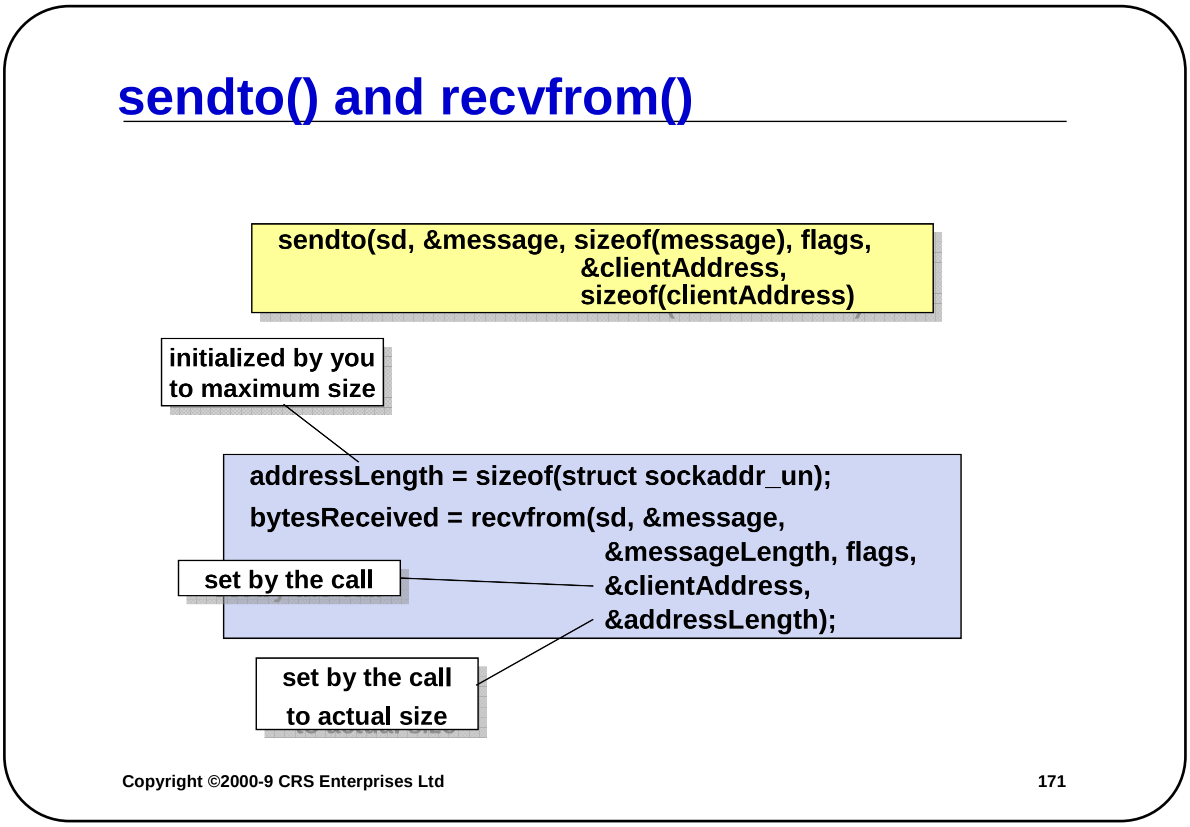
AF_INET

connect() is used by the client end of the rendezvous procedure to make a connection between processes on a connection-oriented socket. The server has made (or will make) a call to accept(), and when the client calls connect(), the link is made.

The socket that is to be used from the client must be specified (by file descriptor), together with the address of the server process to which we are trying to connect and the length of the address structure.

connect() will block the calling process until the connection is made or refused (for example, if the backlog is full or some protocol-specific timeout occurs). If the connection is made, connect() returns 0. If the connection is not made, connect() returns -1.

connect() can also be used with connectionless protocols, but as discussed earlier in the chapter, this does not establish a connection between sockets.



sendto() and recvfrom()

```
sendto(sd, &message, sizeof(message), flags,
                            &clientAddress,
                            sizeof(clientAddress)
```

initialized by you to maximum size

```
addressLength = sizeof(struct sockaddr_un);
bytesReceived = recvfrom(sd, &message,
                              &messageLength, flags,
                              &clientAddress,
                              &addressLength);
```

set by the call

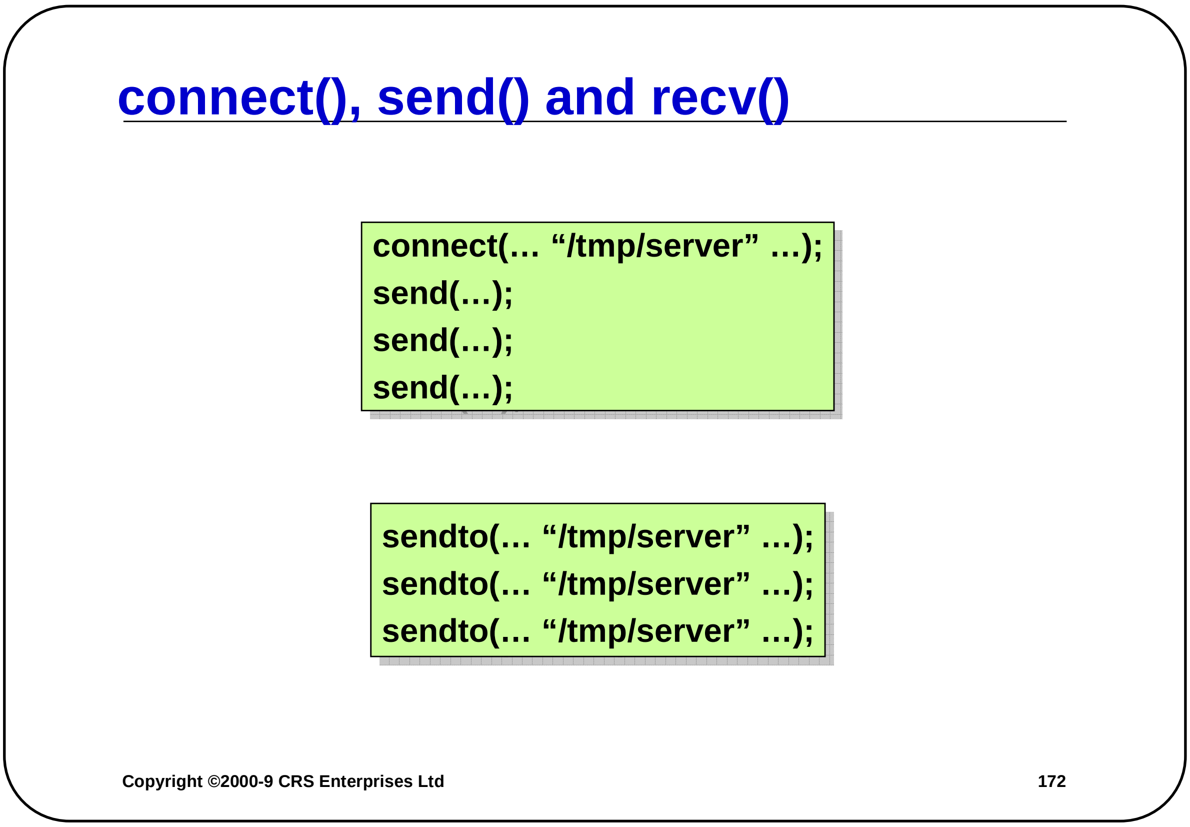
set by the call to actual size

171

recvfrom() and sendto() are used to transmit data through a datagram socket. They are similar to the standard read() and write() calls, but more flexible because they provide a flags field that can be used to send and receive priority data (out of band data).

Note:

As stated earlier, read() and write() should not be used with datagrams.



connect(), send() and recv()

```
connect(… "/tmp/server" …);
send(…);
send(…);
send(…);
```

```
sendto(… "/tmp/server" …);
sendto(… "/tmp/server" …);
sendto(… "/tmp/server" …);
```

recv() and send() are used to read and write data on sockets. They are similar to the standard read() and write() calls, but more flexible because they provide a flags field that can be used to receive priority data (out of band data).

recv() and send() are normally used with connection-oriented sockets, but may also be used on connectionless sockets, where connect() has been executed to mark the destination for packets.

Note:

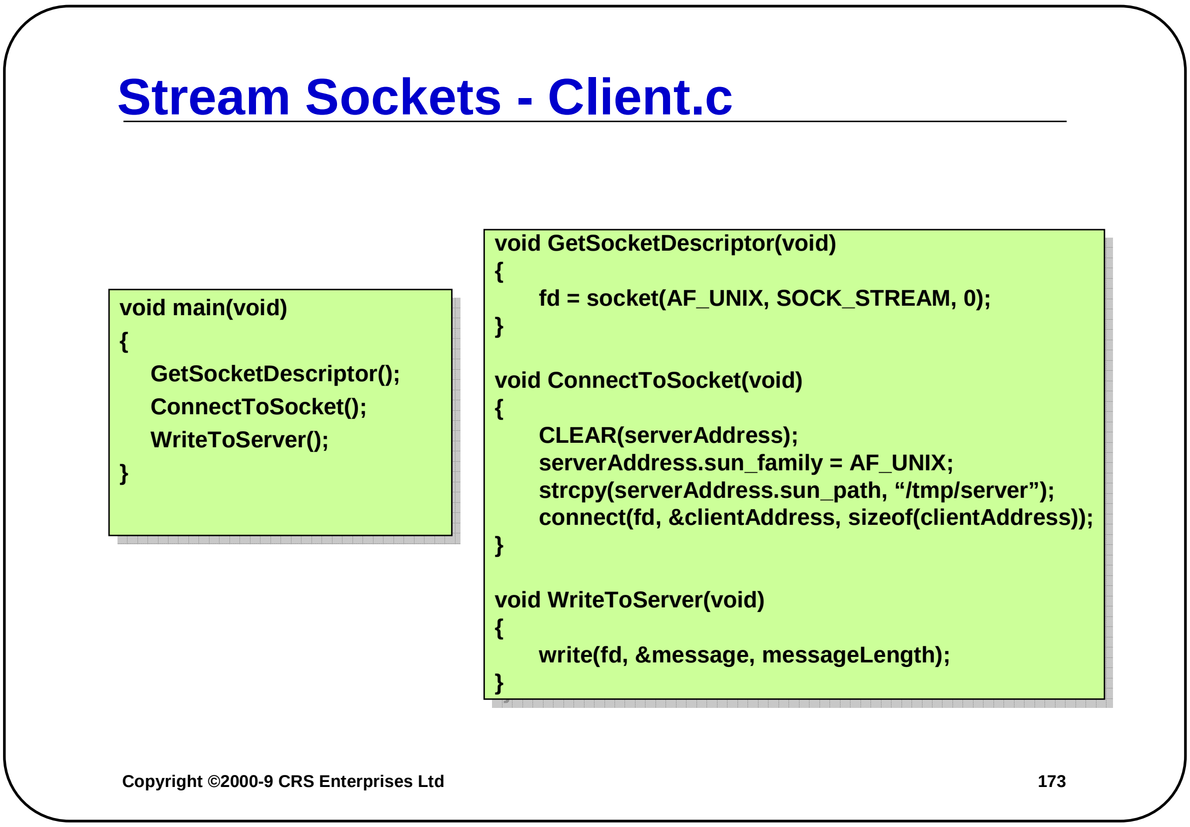
The connect() system call is very confusing when used with datagrams. Since the kernel does not maintain a connection between datagram sockets this call is a misnomer. It is used to remember the last socket address used with a send() or recv() system call. The send() and recv() calls are entirely equivalent to sendto() and recvfrom(), but do not specify the address to/from which the datagram is being sent/received. Instead, connect() is used to remember the address.

Thus the following two sequences of system calls that send 3 datagrams

```
connect("/tmp/server")
send()
send()
send()
```

are equivalent to

```
sendto("/tmp/server")
sendto("/tmp/server")
sendto("/tmp/server")
```



Stream Sockets - Client.c

```
void main(void)
{
    GetSocketDescriptor();
    ConnectToSocket();
    WriteToServer();
}
```

```
void GetSocketDescriptor(void)
{
      fd = socket(AF_UNIX, SOCK_STREAM, 0);
}

void ConnectToSocket(void)
{
      CLEAR(serverAddress);
      serverAddress.sun_family = AF_UNIX;
      strcpy(serverAddress.sun_path, “/tmp/server”);
      connect(fd, &clientAddress, sizeof(clientAddress));
}

void WriteToServer(void)
{
      write(fd, &message, messageLength);
}
```

This example shows the client code required to send a message to the server (error handling not shown).

The client creates a comms socket with the socket() system call. Before it can be used it must be connected to the server socket by the kernel. The connect() system call is used to do this. connect() takes the server’s associated filename as a parameter. This filename must be agreed in advance between the client and server applications.

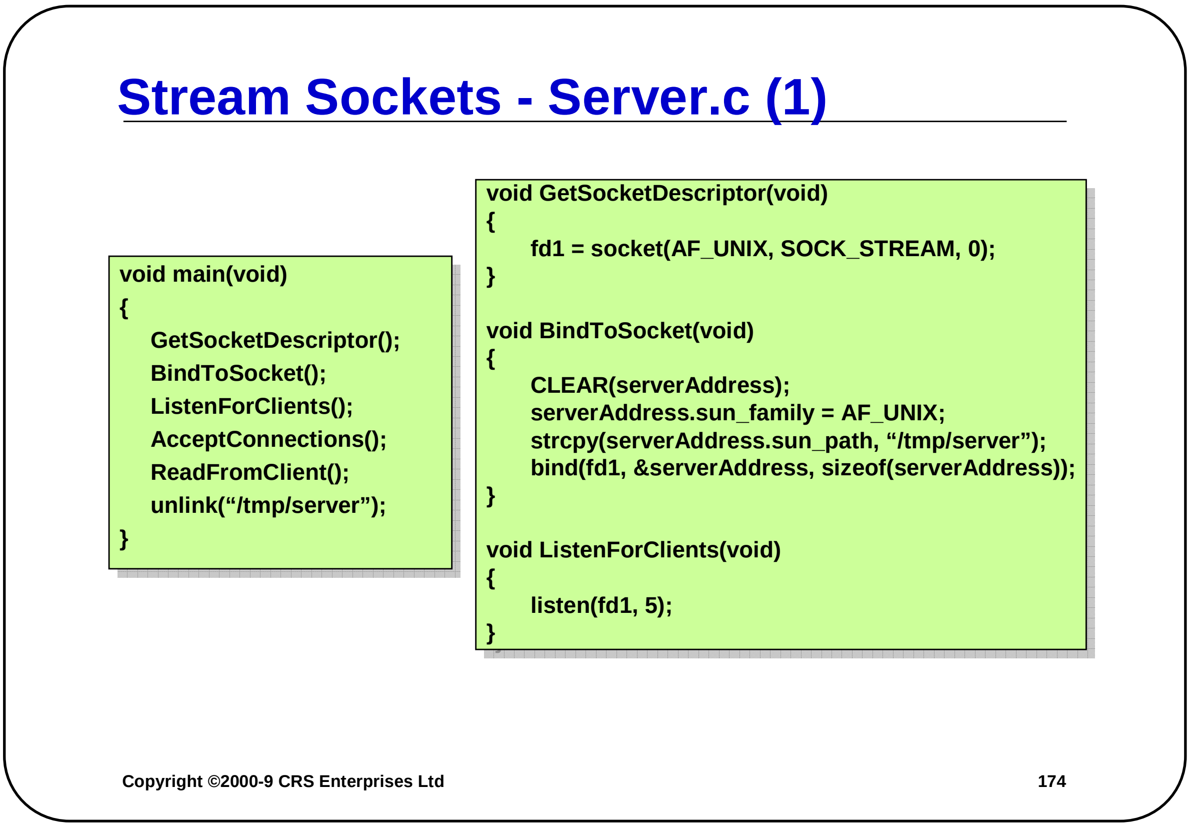
The connect() call will block until the server is ready to take part in the conversation with the client.

Notice that it is a good idea to clear out the address structure (fill with nulls) before adding the server's address; some implementations fail if there are stray characters after the null terminator of “/tmp/server”.

The CLEAR macro is given below

```
#define CLEAR(STRUCTURE)
memset((void*)&STRUCTURE,‘\0’,sizeof(STRUCTURE))
```

The client can use the standard write() system call to send a message once the connection has been established.



Stream Sockets - Server.c (1)

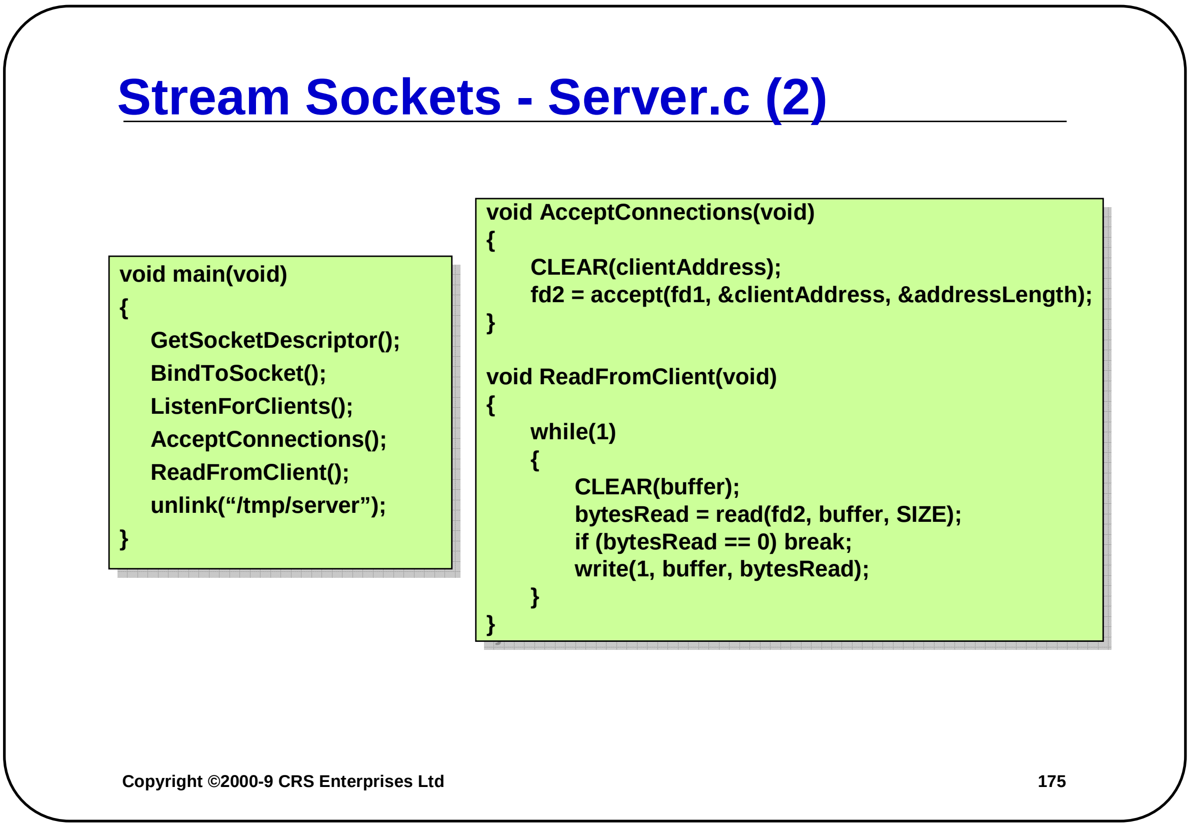
```
void main(void)
{
    GetSocketDescriptor();
    BindToSocket();
    ListenForClients();
    AcceptConnections();
    ReadFromClient();
    unlink("/tmp/server");
}
```

```
void GetSocketDescriptor(void)
{
      fd1 = socket(AF_UNIX, SOCK_STREAM, 0);
}

void BindToSocket(void)
{
      CLEAR(serverAddress);
      serverAddress.sun_family = AF_UNIX;
      strcpy(serverAddress.sun_path, "/tmp/server");
      bind(fd1, &serverAddress, sizeof(serverAddress));
}

void ListenForClients(void)
{
      listen(fd1, 5);
}
```

174

This example shows the server code required to receive a message from the client (error handling not shown).

The server creates a socket with the socket() system call. Before it can be used it must be given a name and converted to a listening socket. bind() is used to give the socket a name and listen() to convert it to a listening socket.



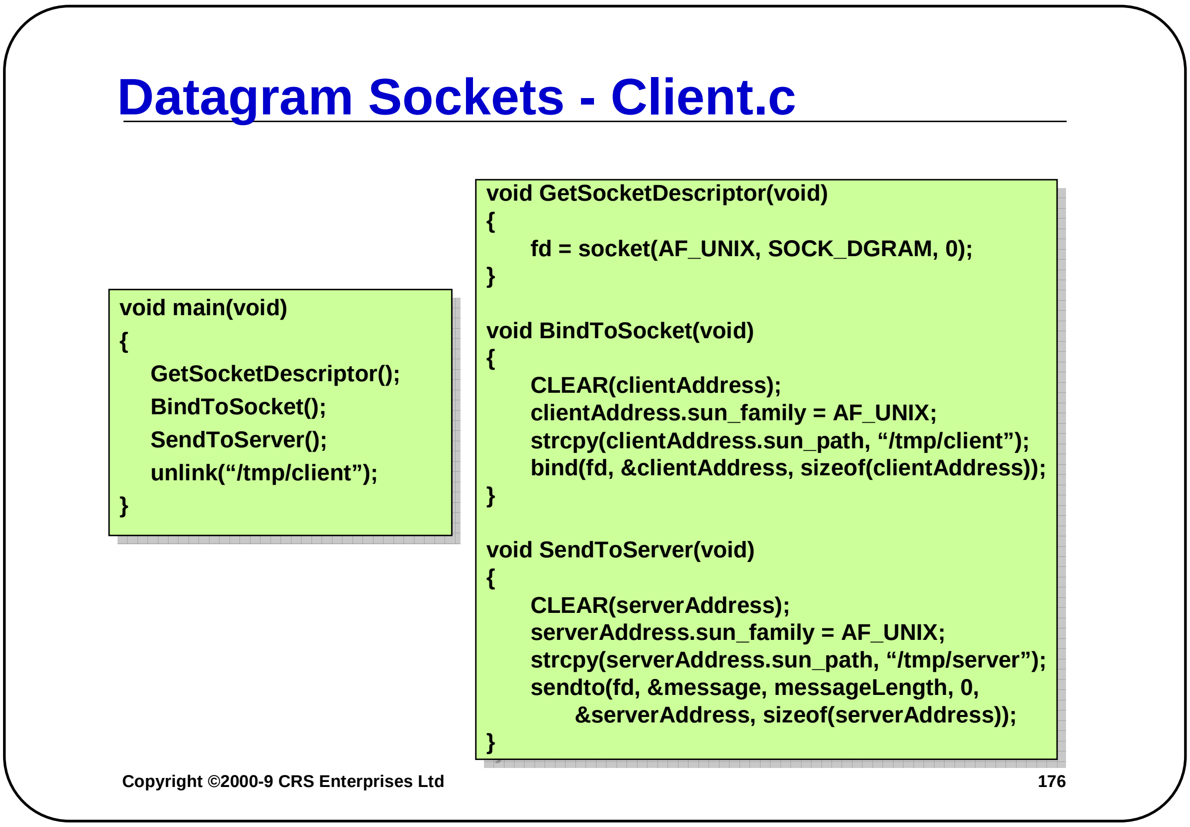
Stream Sockets - Server.c (2)

```
void main(void)
{
    GetSocketDescriptor();
    BindToSocket();
    ListenForClients();
    AcceptConnections();
    ReadFromClient();
    unlink("/tmp/server");
}
```

```
void AcceptConnections(void)
{
    CLEAR(clientAddress);
    fd2 = accept(fd1, &clientAddress, &addressLength);
}

void ReadFromClient(void)
{
    while(1)
    {
        CLEAR(buffer);
        bytesRead = read(fd2, buffer, SIZE);
        if (bytesRead == 0) break;
        write(1, buffer, bytesRead);
    }
}
```

The accept() system call creates a second socket that is connected to the client's socket by the kernel. The file descriptor returned by accept() is the only reference to this socket and should be used in all future system calls.

The server can use the standard read() system call to receive a message once the connection has been established. Note that the read() call does not guarantee to deliver a complete message from the kernel's internal buffers. Several reads may be required before a complete message is extracted from the socket. read() returns zero when the message is complete.



Datagram Sockets - Client.c

```
void main(void)
{
    GetSocketDescriptor();
    BindToSocket();
    SendToServer();
    unlink(“/tmp/client”);
}
```

```
void GetSocketDescriptor(void)
{
      fd = socket(AF_UNIX, SOCK_DGRAM, 0);
}

void BindToSocket(void)
{
      CLEAR(clientAddress);
      clientAddress.sun_family = AF_UNIX;
      strcpy(clientAddress.sun_path, “/tmp/client”);
      bind(fd, &clientAddress, sizeof(clientAddress));
}

void SendToServer(void)
{
      CLEAR(serverAddress);
      serverAddress.sun_family = AF_UNIX;
      strcpy(serverAddress.sun_path, “/tmp/server”);
      sendto(fd, &message, messageLength, 0,
            &serverAddress, sizeof(serverAddress));
}
```

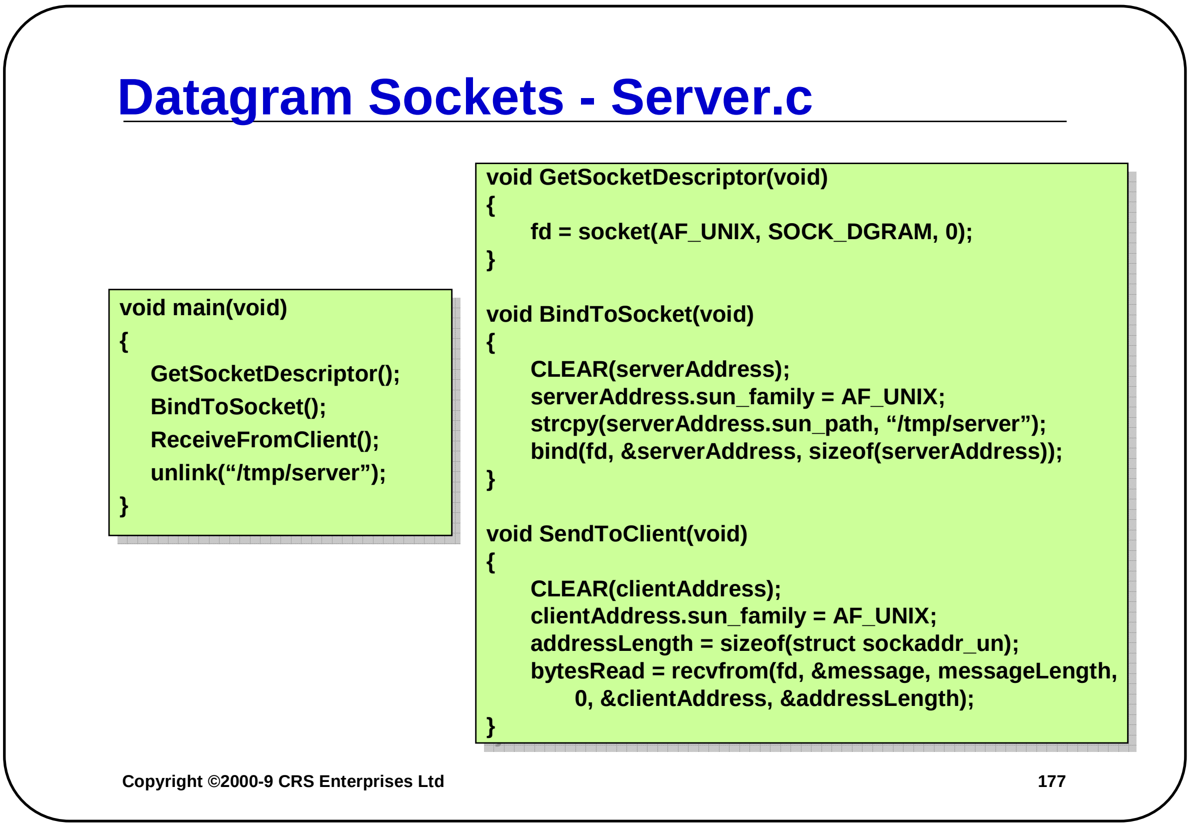
This example shows the client code required to send a single datagram to a server.

The client creates a socket with the socket() system call. Before it can be used it must be given a name with bind(). Notice that it is a good idea to clear out the address structure (fill with nulls) before adding the client address; some implementations fail if there are stray characters after the null terminator of “/tmp/client”.

The CLEAR macro is given below

```
#define CLEAR(STRUCTURE) memset((void*)&STRUCTURE, ‘\0’, sizeof(STRUCTURE))
```

The sendto() system call is used to send the datagram to the server. Note that the server’s address must be include in the call since no connection has been established.



Datagram Sockets - Server.c

```
void main(void)
{
    GetSocketDescriptor();
    BindToSocket();
    ReceiveFromClient();
    unlink("/tmp/server");
}
```

```
void GetSocketDescriptor(void)
{
      fd = socket(AF_UNIX, SOCK_DGRAM, 0);
}

void BindToSocket(void)
{
      CLEAR(serverAddress);
      serverAddress.sun_family = AF_UNIX;
      strcpy(serverAddress.sun_path, "/tmp/server");
      bind(fd, &serverAddress, sizeof(serverAddress));
}

void SendToClient(void)
{
      CLEAR(clientAddress);
      clientAddress.sun_family = AF_UNIX;
      addressLength = sizeof(struct sockaddr_un);
      bytesRead = recvfrom(fd, &message, messageLength,
            0, &clientAddress, &addressLength);
}
```

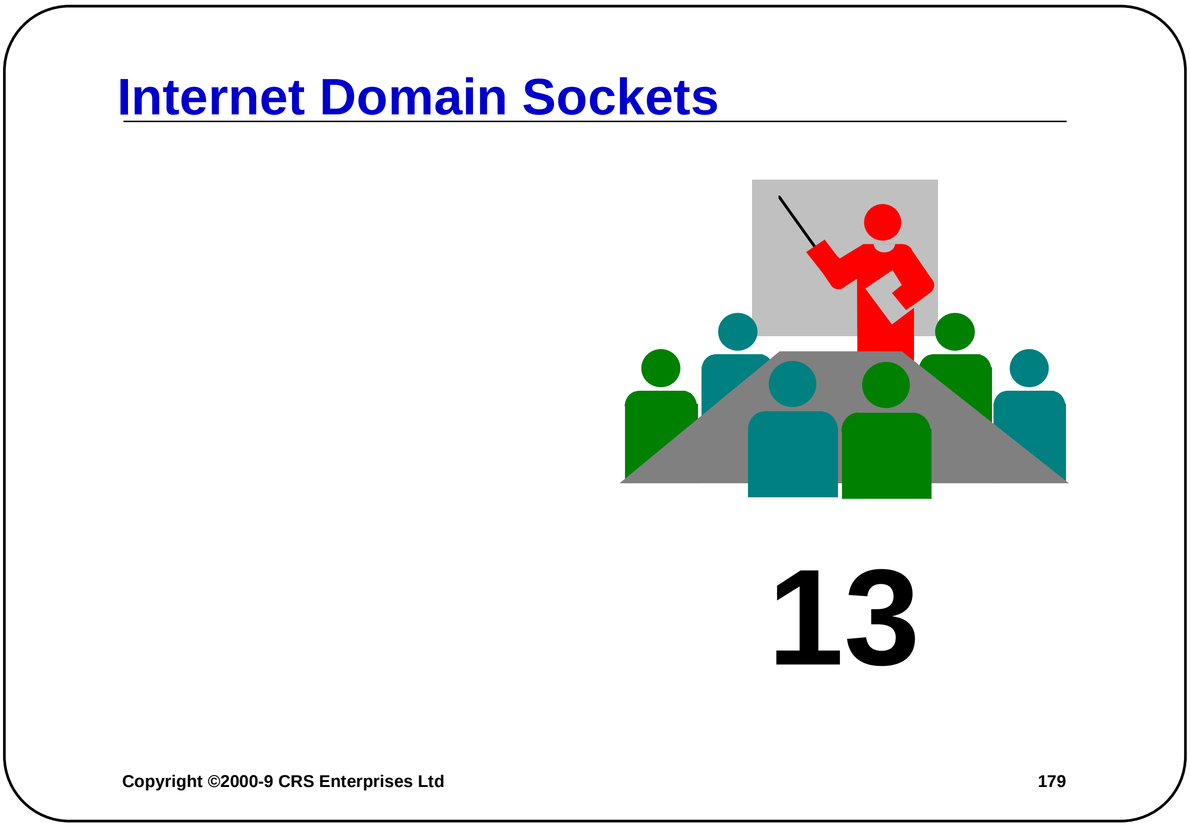
This example shows the server code required to receive a single datagram from a client (error handling not shown). The code is similar to the client code (see earlier) .

Notes:

When the server receives the datagram, the recvfrom() system call will fill in the sender's address in the clientAddress structure. Before calling recvfrom() you must initialize addressLength to the size of the buffer which will receive the sender's address.

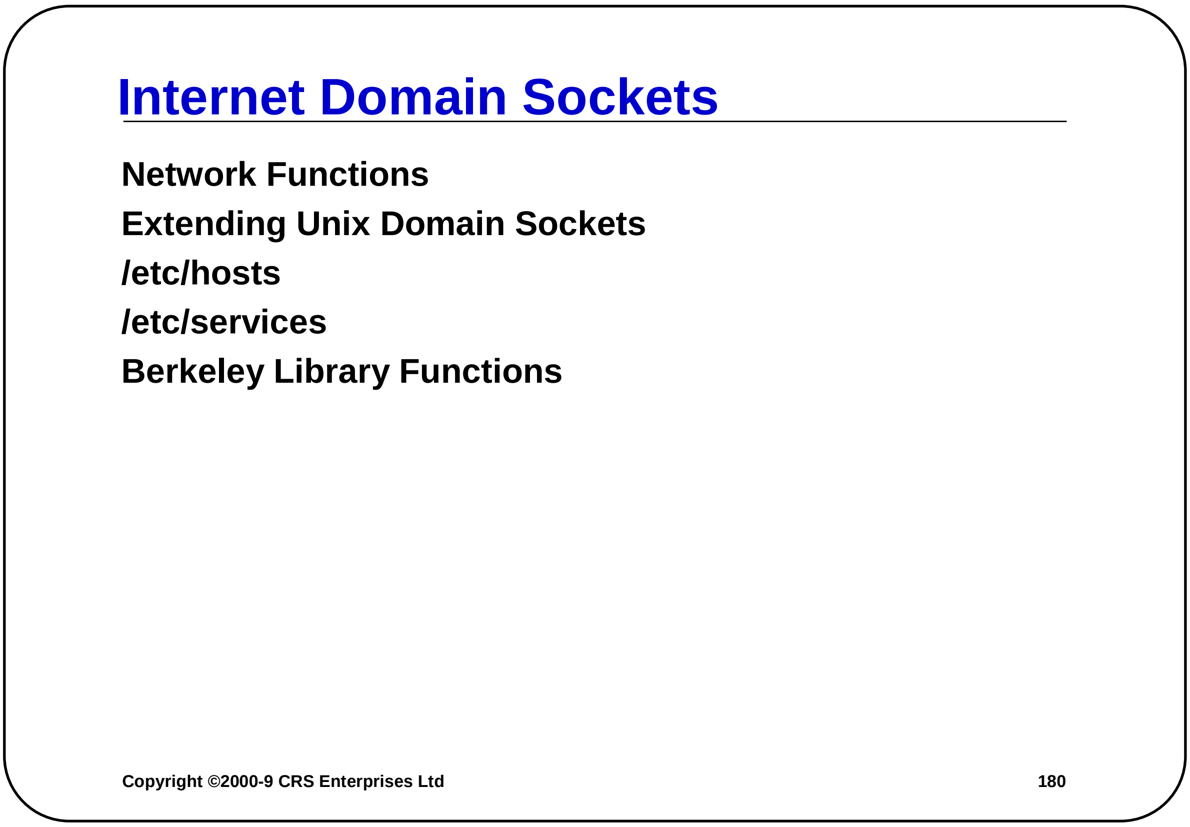
The length of the datagram is returned in bytesRead..

13



Internet Domain Sockets

13



Internet Domain Sockets

Network Functions

Extending Unix Domain Sockets

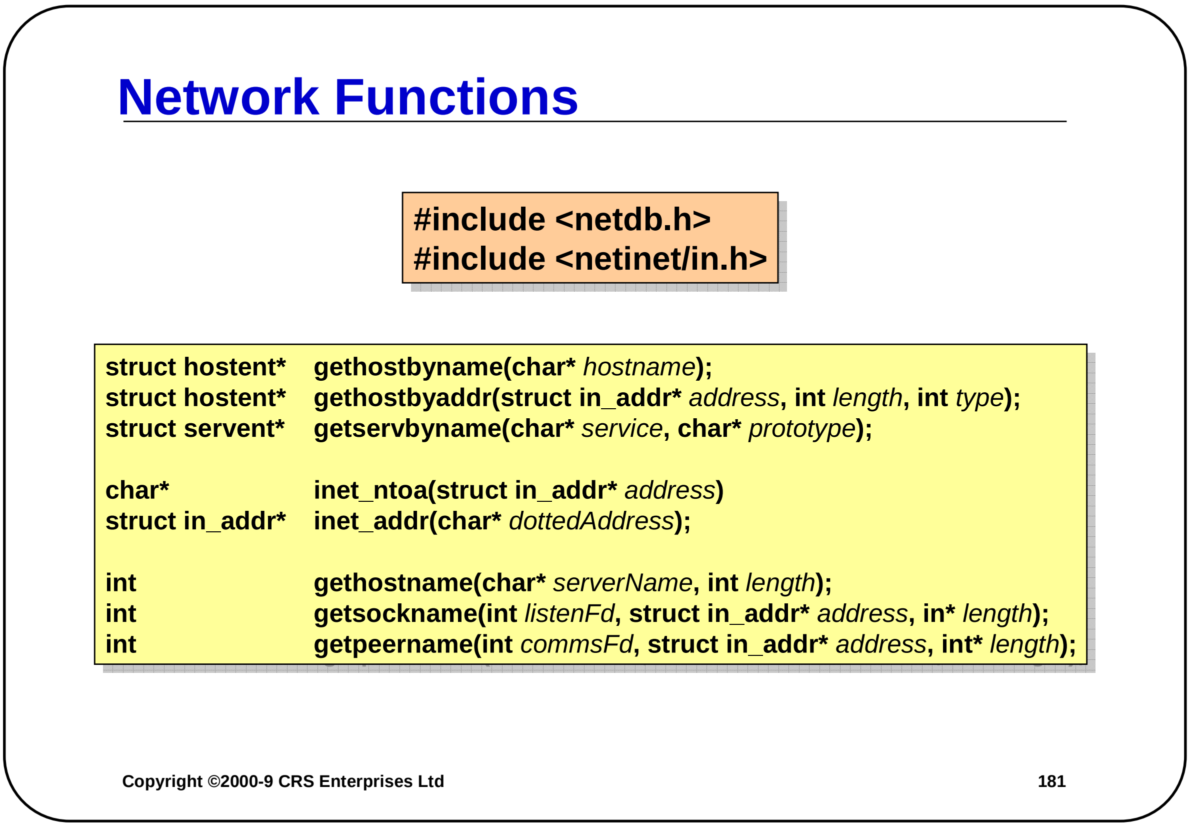
/etc/hosts

/etc/services

Berkeley Library Functions

This chapter provides an introduction to Internet Domain sockets. Stream and Datagram sockets for the Unix Domain are easily extended to the Internet Domain, simply by changing the socket filename into a 32 bit internet address and 16 bit port number.

Internet Domain socket programming also involves using strings for the hostname instead of a numeric internet address and a string for the service instead of a numeric port number. Lookup functions are used to extract this information from the Domain Name Server if one is being used or else from the local administration files /etc/hosts and /etc/services. We discuss these lookup routines and the data structures associated with them.



Network Functions

```
#include <netdb.h>
#include <netinet/in.h>
```

```
struct hostent*   gethostbyname(char* hostname);
struct hostent*   gethostbyaddr(struct in_addr* address, int length, int type);
struct servent*   getservbyname(char* service, char* prototype);

char*             inet_ntoa(struct in_addr* address)
struct in_addr*   inet_addr(char* dottedAddress);

int               gethostname(char* serverName, int length);
int               getsockname(int listenFd, struct in_addr* address, in* length);
int               getpeername(int commsFd, struct in_addr* address, int* length);
```

181

Function prototypes of the network functions discussed in this chapter are presented above. Brief descriptions of these routines follow:

gethostbyname

> use string to lookup host in /etc/hosts or name server and return a structure describing the host.

gethostbyaddr

> reverse lookup of gethostbyname.

getservbyname

> use string to lookup service in /etc/services or name server and return a structure describing the service (port number and protocol).

inet_ntoa

> convert dotted address to 32 bit internet address.

inet_addr

> convert 32 bit internet address to dotted address.

gethostname

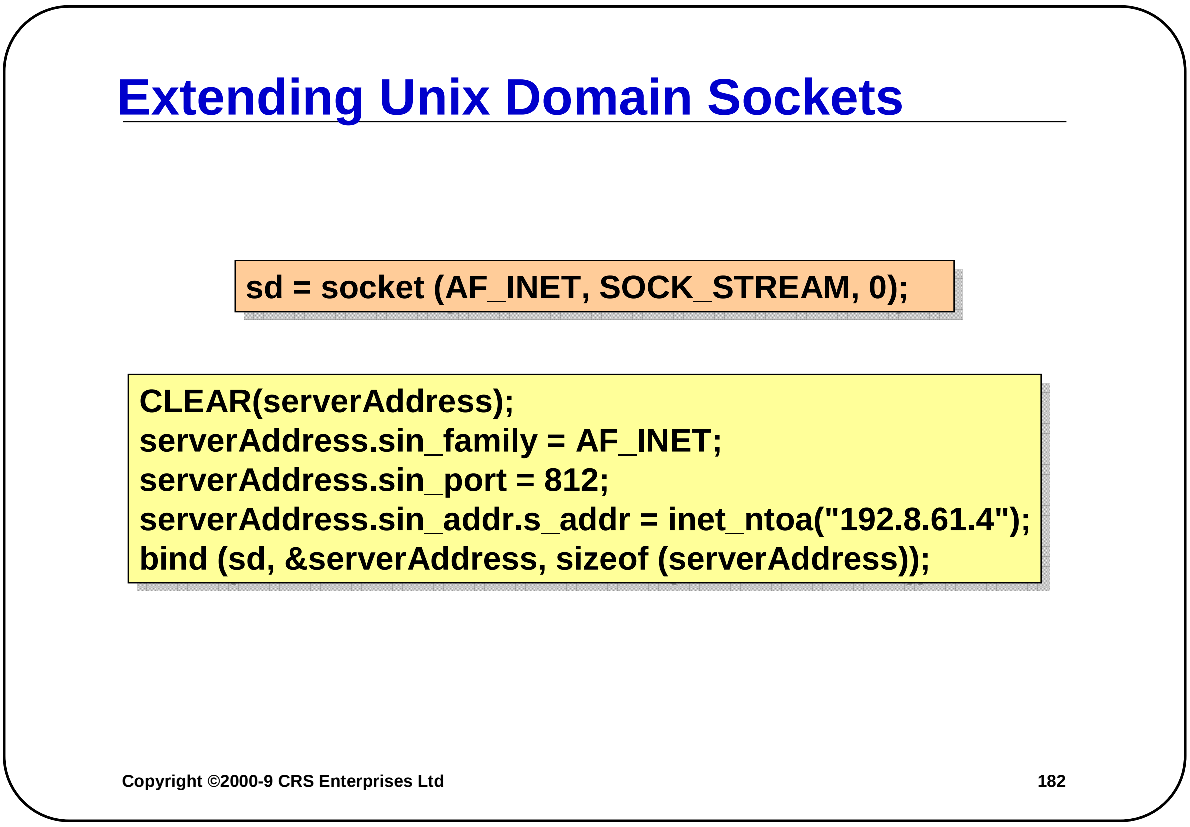
> determine name of host

getsockname

> determine 32 bit internet address of host

getpeername

> determine 32 bit internet address of connected host (tcp only)



Extending Unix Domain Sockets

```
sd = socket (AF_INET, SOCK_STREAM, 0);
```

```
CLEAR(serverAddress);
serverAddress.sin_family = AF_INET;
serverAddress.sin_port = 812;
serverAddress.sin_addr.s_addr = inet_ntoa("192.8.61.4");
bind (sd, &serverAddress, sizeof (serverAddress));
```

Extending Unix Domain Sockets to the Internet Domain is very simple:

The domain name used in the socket() system call is changed from AF_UNIX to AF_INET.

The server address specified in the bind() system call is changed from a Unix filename to a 32 bit internet number and a 16 bit port number. This information is packaged together in the internet socket address structure:

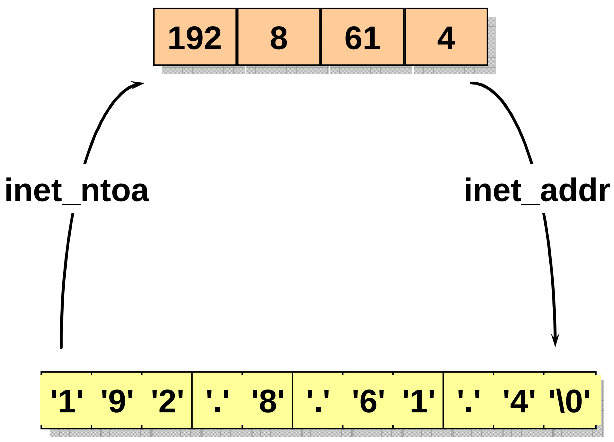
```
struct sockaddr_in
{
        short           sin_family;
        unsigned short          sin_port;
        struct in_addr          sin_addr;
};
```

where the 32 bit internet address is defined by:

```
struct in_addr
{
        unsigned long           s_addr;
};
```

The internet address is often specified in dotted form and converted to 32 bits by the conversion routine inet_ntoa().

Address Conversion Functions



192 8 61 4

inet_ntoa

inet_addr

'1' '9' '2' '.' '8' '.' '6' '1' '.' '4' '\0'

Conversions between 32 bit internet address and the dotted form are common place. Conversion routines exist in both directions.

Dotted address to 32 bit address:

```
struct in_addr address;
char dottedAddress[] = "192.8.61.4";
address = inet_ntoa(dottedAddress);
```

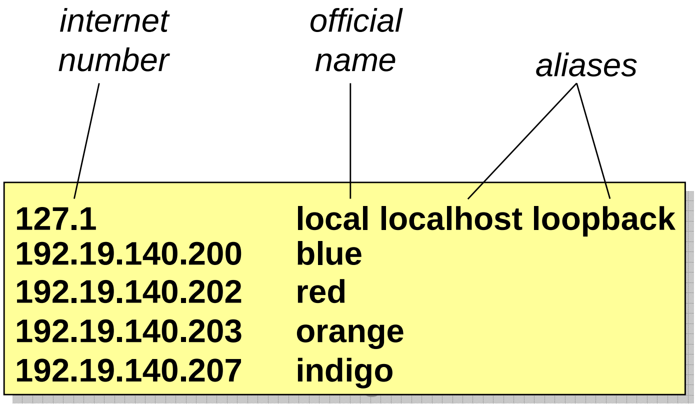
32 bit address to dotted address:

```
struct in_addr address = { 192, 8, 61, 4 };
char dottedAddress[20];
dottedAddress = inet_addr(address);
```

In some applications it is useful to use a wildcard address (for the server to accept connections from any client):

```
struct in_addr = INADDR_ANY;
```

/etc/hosts

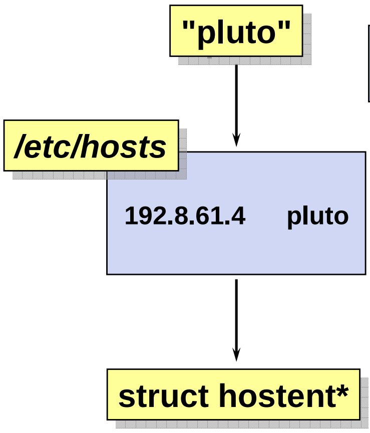


```
127.1             local localhost loopback
192.19.140.200    blue
192.19.140.202    red
192.19.140.203    orange
192.19.140.207    indigo
```

The /etc/hosts file contains a lookup table of machine names and machine internet numbers. Nowadays, it is commonplace to replace this file with a name server to simplify administration in large networks. Either way, our application programs often require this information.

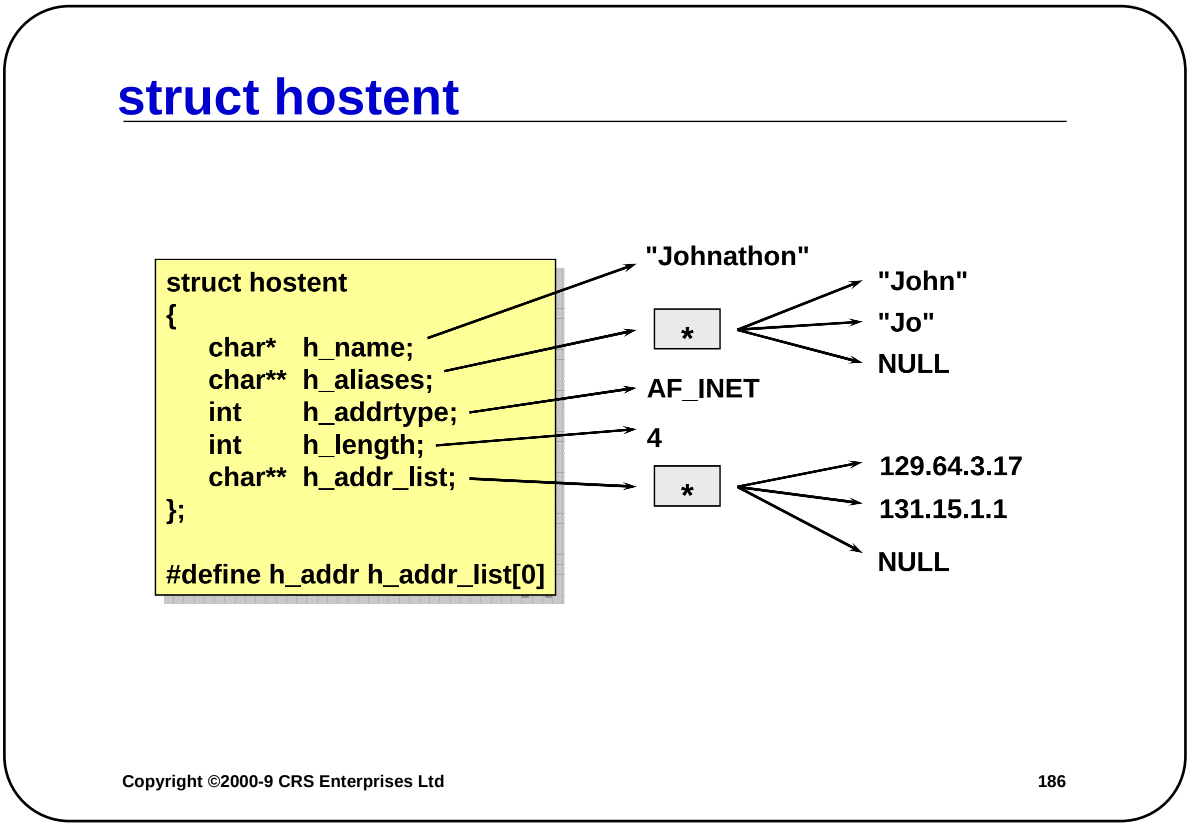
Notice that each machine has an official name plus optional aliases. Any given machine can also have more than one internet number if it is connected to more than network. The special internet number 127.0.0.1 is called the local host. The local host is not a real machine in its own right; it is implemented as a loopback device driver inside the kernel. This allows internet socket programming to be tested on a stand alone machine.

gethostbyname()



```
struct hostent* pHost;
pHost = gethostbyname(serverName);
```

Use the gethostbyname() function to extract information on a machine from the Domain Name Server or /etc/hosts. The name of the machine is used as the input to gethostbyname(); it returns a pointer to a struct hostent structure (see overleaf). This structure contains the official name and aliases of the machine together with a list of the 32 internet numbers for the machine.



struct hostent

```
struct hostent
{
    char*   h_name;
    char**  h_aliases;
    int     h_addrtype;
    int     h_length;
    char**  h_addr_list;
};

#define h_addr h_addr_list[0]
```

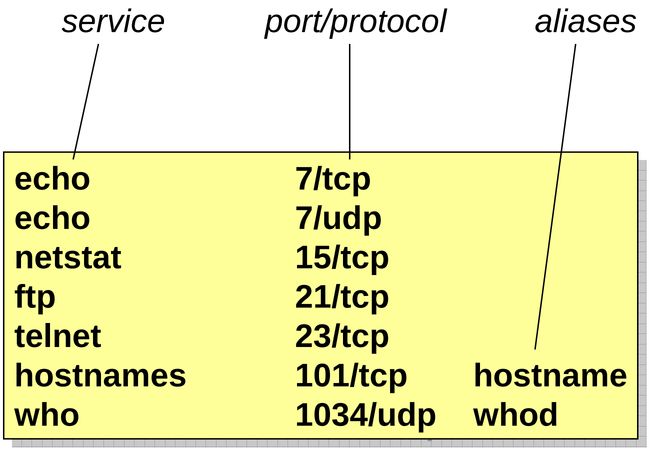
186

The gethostbyname() function returns a pointer to a struct hostent:

```
struct hostent
{
        char*       h_name;
        char**      h_aliases;
        int         h_addrtype;
        int         h_length;
        char**      h_addr_list;
};
```

Information on aliases and internet numbers can be easily extracted from this structure (see examples at the end of the chapter).

/etc/services



service port/protocol aliases

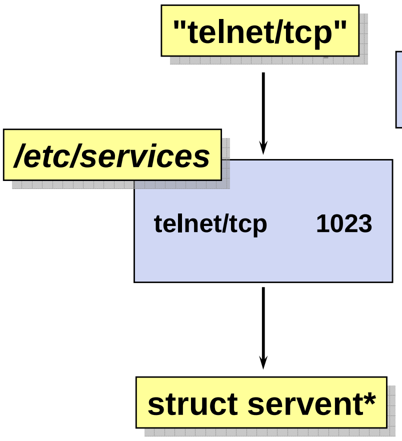
```
echo        7/tcp
echo        7/udp
netstat     15/tcp
ftp         21/tcp
telnet      23/tcp
hostnames   101/tcp    hostname
who         1034/udp   whod
```

The /etc/services file contains a lookup table of services available on a machine and their corresponding protocols and port numbers. As with the host information, this file is often replaced with a name server to simplify administration in large networks.

Notice that each service has an official name plus optional aliases. Any given service will be assigned a port number and protocol. Note that the set of port numbers for TCP and UDP are distinct. Thus TCP port 23 is not the same as UDP port 23.

Note: When making a bind() system call, a port number of zero indicates that you want the kernel to choose a free port for this protocol.

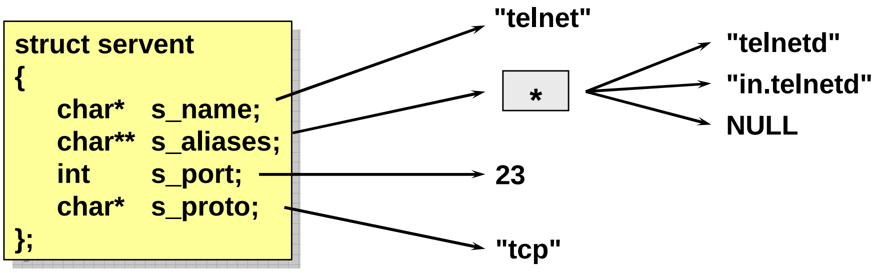
getservbyname()



```
struct servent* pService;
pService = getservbyname(service, "tcp");
```

Use the getservbyname() function to extract information on a machine from the Domain Name Server or /etc/services. The name of the service and protocol to be used are input to getservbyname() and a pointer to a struct hostent is returned (see overleaf). This structure contains the name of the service and any aliases defined for the service together with a the port number and protocol.

struct servent



```
struct servent
{
    char*   s_name;
    char**  s_aliases;
    int     s_port;
    char*   s_proto;
};
```

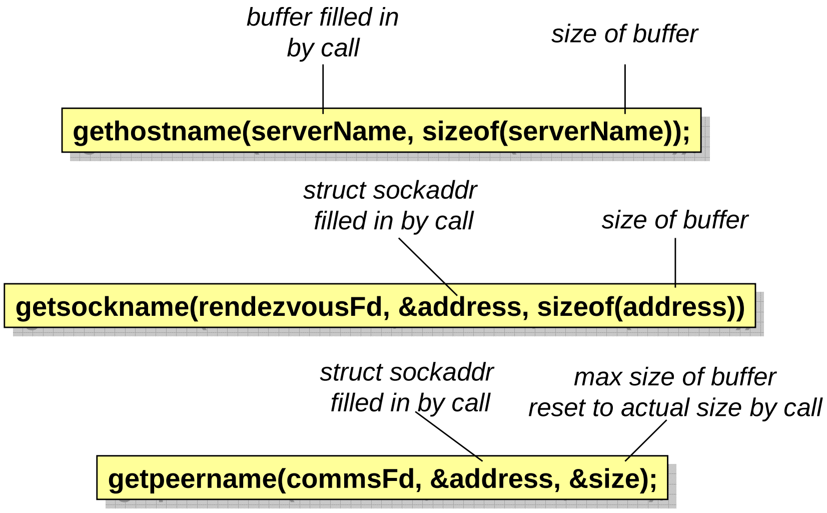
The getservbyname() function returns a pointer to a struct servent:

```
struct servent
{
        char*       s_name;
        char**      s_aliases;
        int         s_port;
        char*       s_proto;
};
```

Information on aliases and port number can be easily extracted from this structure:

```
struct servent* pService;
int i;

pService = getservbyname("telnet", "tcp");
printf("Telnet(TCP) is on port %i", pAddress->s_port);
for(i = 0; ; i++)
{
        if (pAddress->s_aliases[i] != NULL) break;
        printf("Alias name: %s \n", s_aliases[i]);
}
```

Berkeley Library Functions



buffer filled in by call

size of buffer

```
gethostname(serverName, sizeof(serverName));
```

struct sockaddr filled in by call

size of buffer

```
getsockname(rendezvousFd, &address, sizeof(address))
```

struct sockaddr filled in by call

max size of buffer reset to actual size by call

```
getpeername(commsFd, &address, &size);
```

The 3 utility functions above are part of the Berkeley compatibility library:

int gethostname(char* serverName, int size)

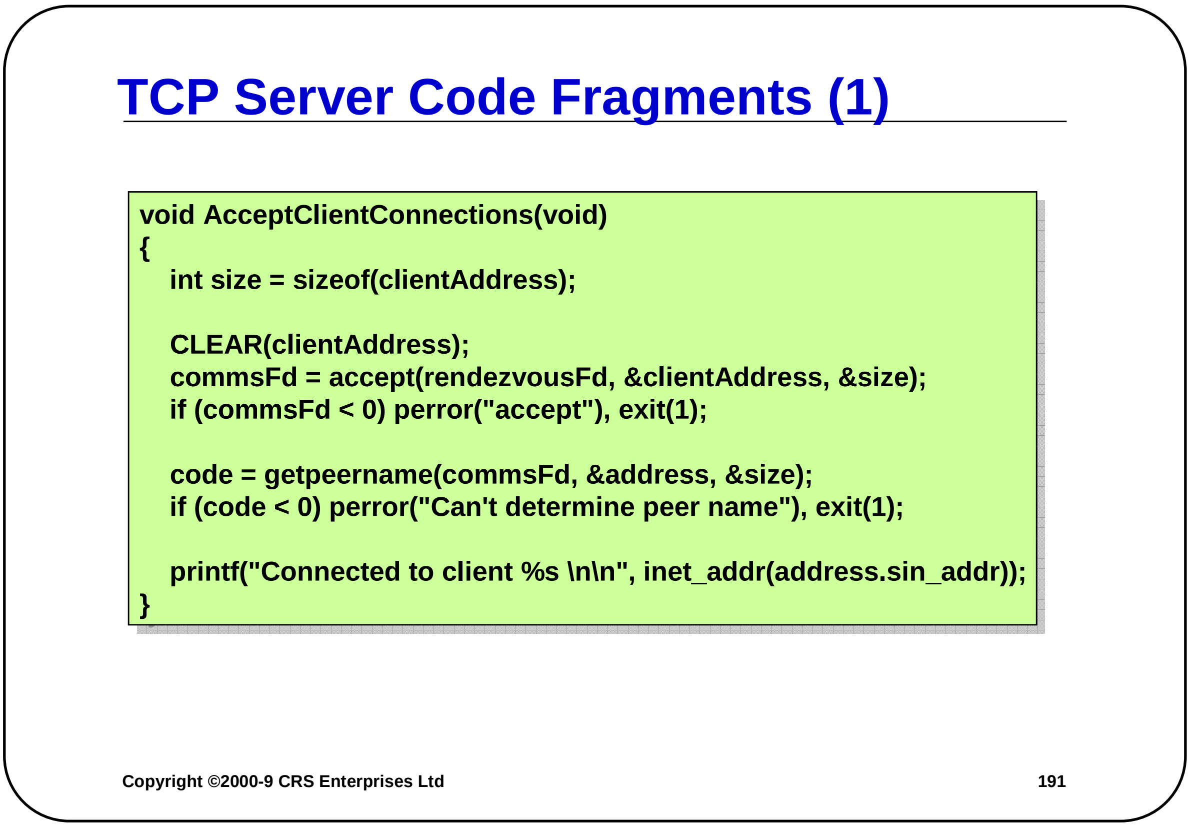
> Fills in the name of the server into the character array serverName. The second parameter specifies the size of the character array.

int getsockname(int rendezvousFd, struct in_addr* address, int size)

> Fills in the internet address of the server given the file descriptor of the socket. The file descriptor is the one returned from the original socket() call. The last parameter specifies the size of the address structure.

int getpeername(int commsFd, struct in_addr* address, int* size)

> Used by a server to determine the internet address of a connected client. The file descriptor used is that returned in the server by the accept() system call. The last parameter is a pointer to an integer that specifies the size of the address structure. This must be a pointer, because it is updated on return from the call with the actual size of address structure.

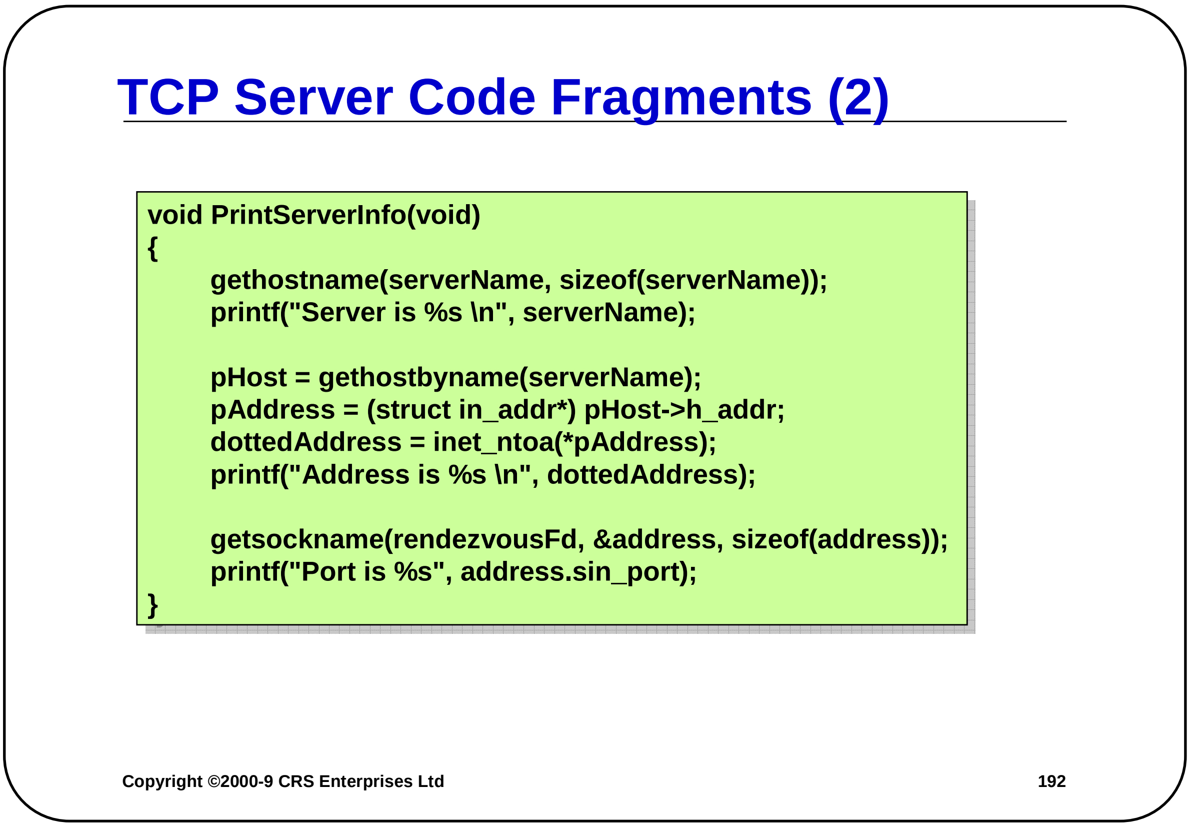


TCP Server Code Fragments (1)

```
void AcceptClientConnections(void)
{
   int size = sizeof(clientAddress);

   CLEAR(clientAddress);
   commsFd = accept(rendezvousFd, &clientAddress, &size);
   if (commsFd < 0) perror("accept"), exit(1);

   code = getpeername(commsFd, &address, &size);
   if (code < 0) perror("Can't determine peer name"), exit(1);

   printf("Connected to client %s \n\n", inet_addr(address.sin_addr));
}
```

191

This code fragment shows a TCP server establishing a connection to a client. After the connection is established, the internet address of the client is determined using getpeername(). The 32 bit internet address is printed in dotted decimal form with the help of the inet_addr() function.

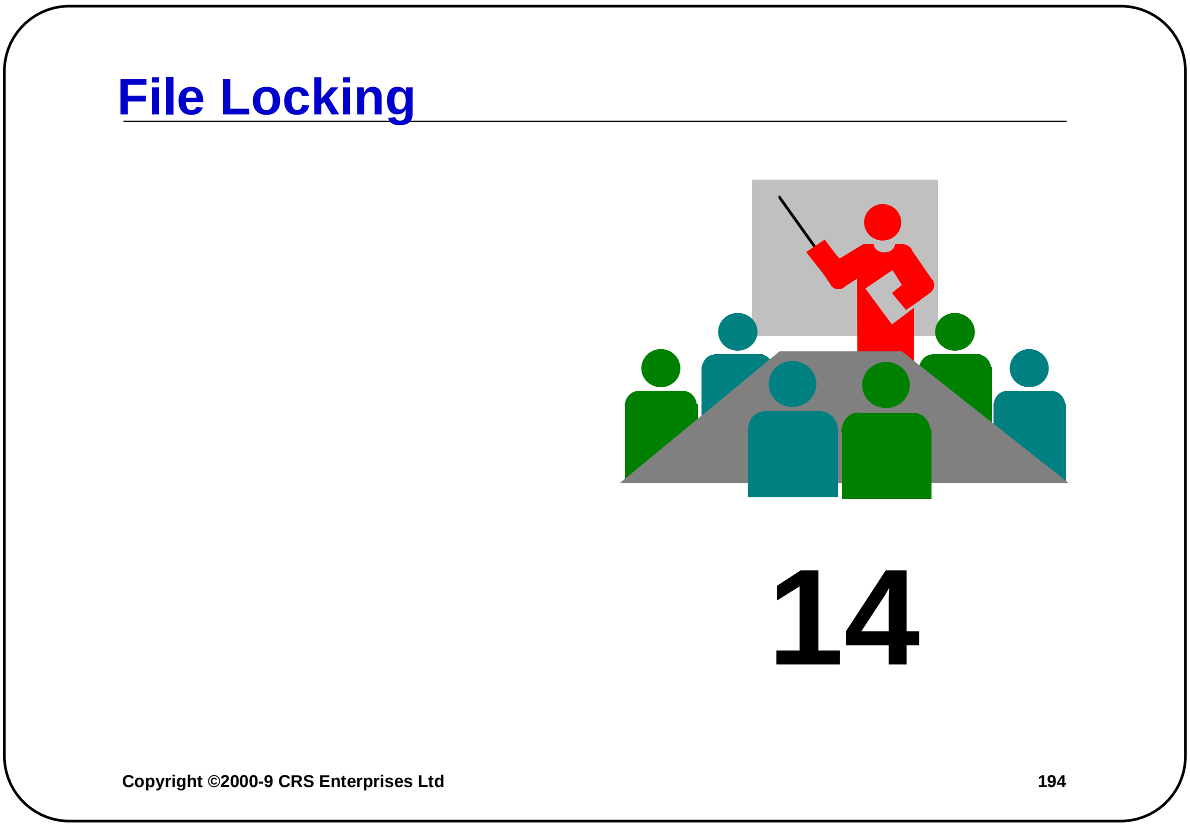


TCP Server Code Fragments (2)

```
void PrintServerInfo(void)
{
        gethostname(serverName, sizeof(serverName));
        printf("Server is %s \n", serverName);

        pHost = gethostbyname(serverName);
        pAddress = (struct in_addr*) pHost->h_addr;
        dottedAddress = inet_ntoa(*pAddress);
        printf("Address is %s \n", dottedAddress);

        getsockname(rendezvousFd, &address, sizeof(address));
        printf("Port is %s", address.sin_port);
}
```

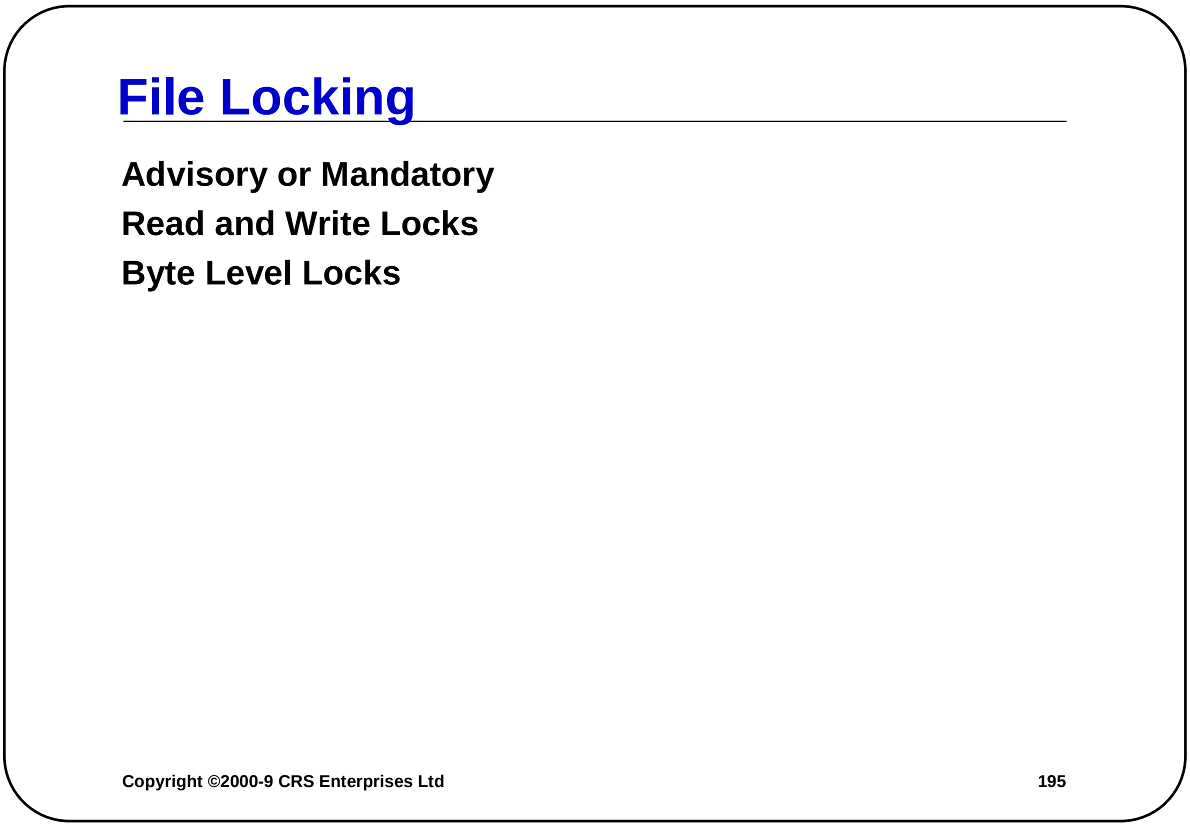
This code fragment shows how to utilise some of the functions presented earlier to extract and print information about a server using the TCP protocol. It is assumed that the server has already returned from accept() and therefore a connection with a client has been established.

14



File Locking

14



File Locking

Advisory or Mandatory

Read and Write Locks

Byte Level Locks

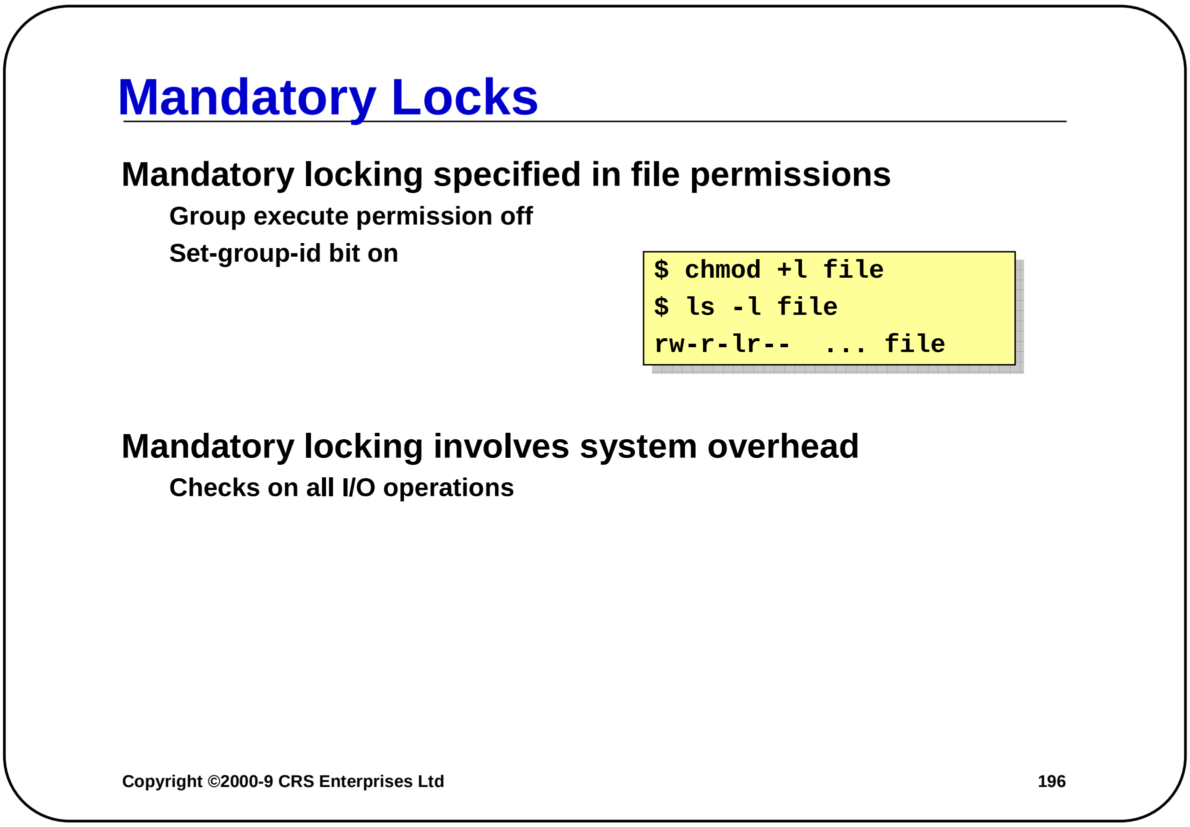
Unix allows processes to lock files or portions of files, so that access to shared data can be coordinated. In Unix terminology, a record is simply a contiguous sequence of bytes within a file. There is no record structure built in to files on Unix, but many applications implement their own structure on top of the standard Unix byte-stream model. The notion of a record in Unix is therefore very flexible.

Two forms of locking are available:

> An advisory lock is noted by the system, but not checked by the kernel during I/O operations. Processes are expected to cooperate when using this form of locking. It is assumed that a process will not access a record without checking for locks first. This means that it is possible for a process to override the locking, so files that are to be used this way are normally given extra protection through the access permissions and set-group-id mechanism.
>
> With mandatory locking on a file, all I/O operations (read() and write()) involve a check on the lock status of the file by the kernel. These locks are strictly enforced.

Two types of lock are possible:

> Read locks are often called shared locks, since many processes may hold a read lock on a portion of a file. No process may lock the file for writing while there is a read lock in place.
>
> Write locks are also called exclusive locks. Only one process may hold a write lock on a portion of a file. Read locks are prohibited if there is a write lock in place.



Mandatory Locks

Mandatory locking specified in file permissions

Group execute permission off

Set-group-id bit on

```
$ chmod +l file
$ ls -l file
rw-r-lr--  ... file
```

Mandatory locking involves system overhead

Checks on all I/O operations

Copyright ©2000-9 CRS Enterprises Ltd 196

To specify that mandatory locking is to be applied to a file, the access permission bits are used. Group-execute permission is turned off and the set-group-id bit is turned on. This combination has no other meaning to Unix, so it can be used in this way.

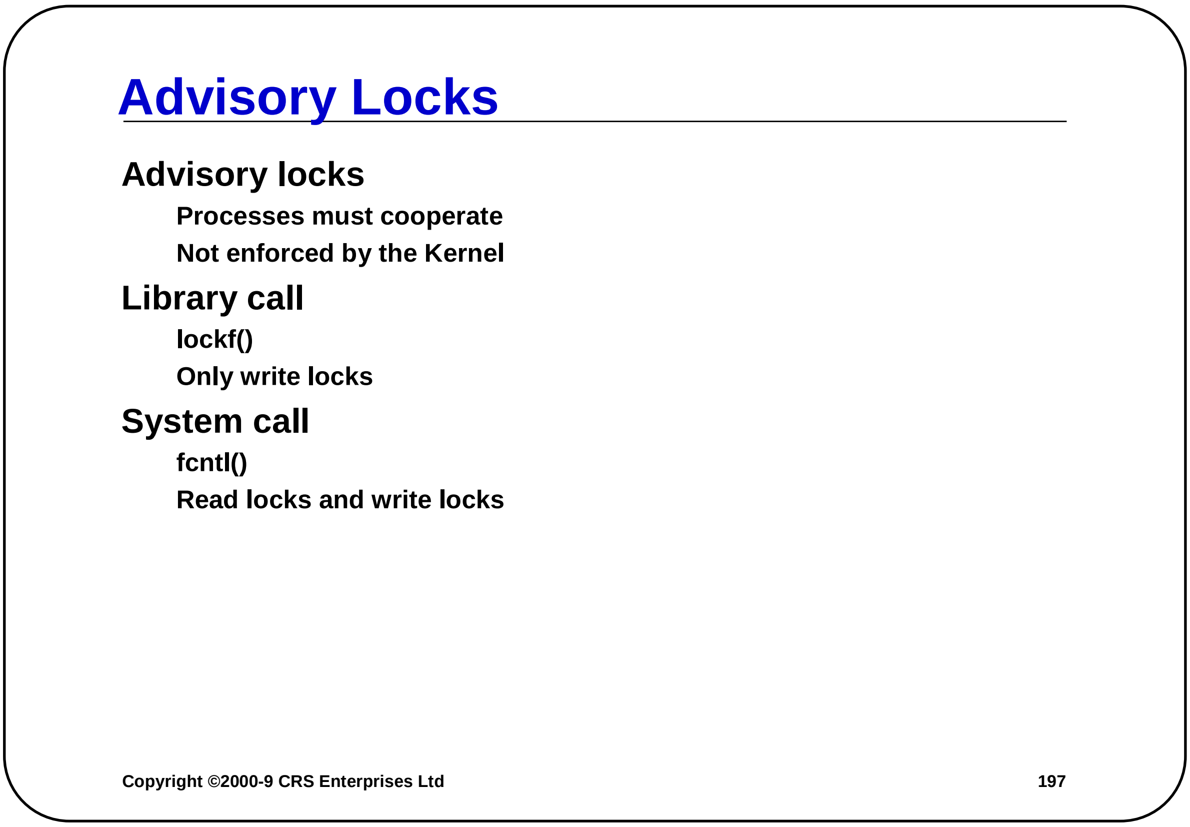
The chmod command can be used to achieve this:

```
$ chmod +l file
```

When ls -l is used on the file, the display will be:

```
-rw-r-lr-- ... file
```

Although it may seem more powerful, mandatory locking is not always superior to advisory locking. Mandatory locking involves extra system activity on all I/O operations to a file, and this may have a performance impact. Only the locked portions of a file are protected; the rest is accessible as normal via the standard Unix access controls. It is dangerous, therefore, to rely on the locking system to protect data.



Advisory Locks

Advisory locks

- Processes must cooperate
- Not enforced by the Kernel

Library call

- lockf()
- Only write locks

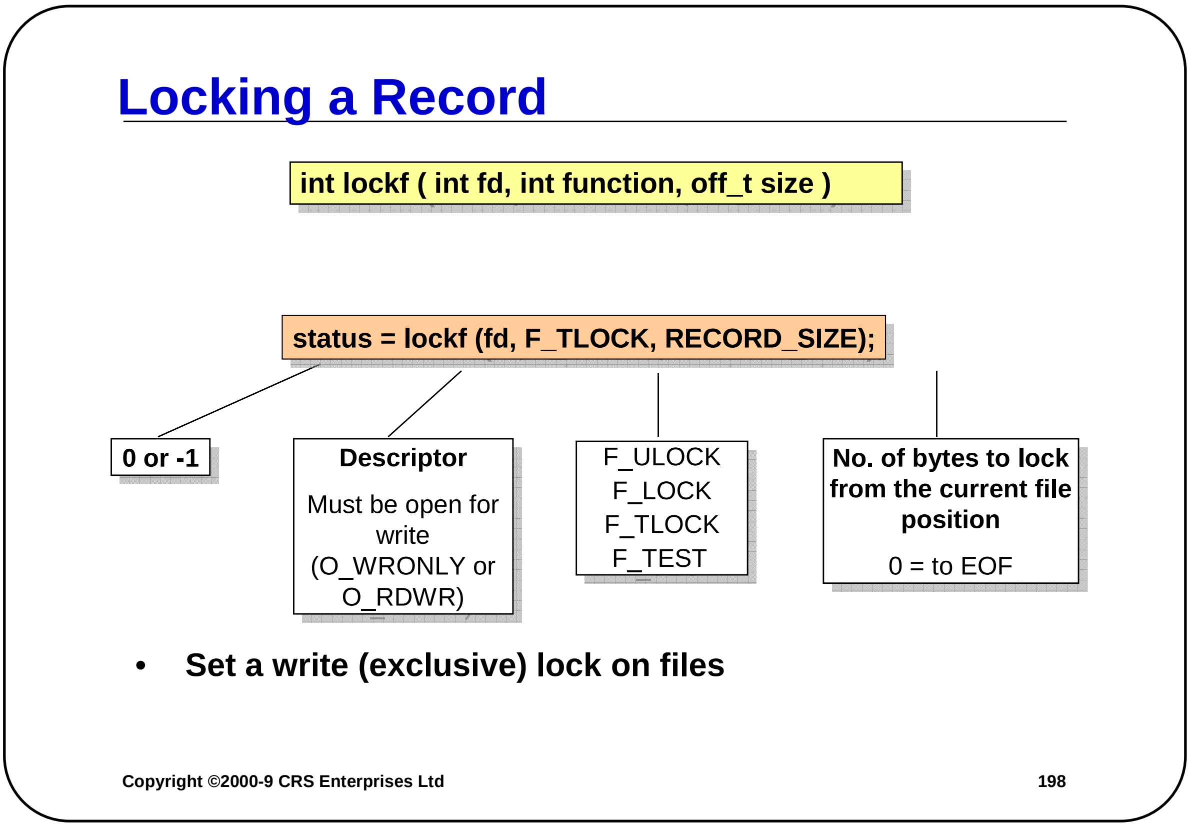
System call

- fcntl()
- Read locks and write locks

Advisory locking provides an efficient method for cooperating processes to lock files. Unlike mandatory locking, the kernel does not check locks during read() and write() system calls for advisory locks. It is incumbent on a process to check it has the appropriate lock before attempting I/O.

The library call lockf() provides a simplified interface to the fcntl() system call. lockf() can only be used for maintaining write (exclusive) locks. lockf() can be used to lock an entire file or any range of bytes within a file.

The system call fcntl() provides both read (shared) and write (exclusive) locks and can be used to lock an entire file or any range of bytes within a file. The parameters passed to fcntl() are a little complicated, so use lockf() if you need only write locks.



Locking a Record

int lockf (int fd, int function, off_t size)

status = lockf (fd, F_TLOCK, RECORD_SIZE);

0 or -1

Descriptor

Must be open for write (O_WRONLY or O_RDWR)

F_ULOCK
F_LOCK
F_TLOCK
F_TEST

No. of bytes to lock from the current file position

0 = to EOF

- **Set a write (exclusive) lock on files**

Copyright ©2000-9 CRS Enterprises Ltd 198

In order to use lockf() on a file, the file must have been opened with write permission (i.e. with the O_WRONLY or O_RDWR flag). The descriptor fd of the file is passed into the function. lockf() operates on a portion of a file from the current file position, either forwards or backwards according to the value of the size parameter. If size is 0, then the locked portion is taken to stretch to the end of the file.

The details of the operation to be performed by lockf() are contained in the function parameter:

F_ULOCK Unlock a section that had previously been locked.

F_LOCK Place an exclusive lock on the specified section. If the section (or part of it) is already locked, the process will block until the lock is freed. However, before the process sleeps, a check is made to see whether there is the potential for a deadlock situation that would be caused by this process sleeping. lockf() returns -1with errno set to EDEADLK if this is the case.

F_TLOCK This causes an atomic "test and set" operation for acquiring a lock on the specified portion. It is very similar to F_LOCK, but does not cause the process to block if the record is already locked. lockf() returns -1 with errno set to EAGAIN if the area is locked.

F_TEST Test to see whether there are any locks held on the required record. lockf() returns -1 with errno set to EAGAIN if the area is locked.

Locking.c

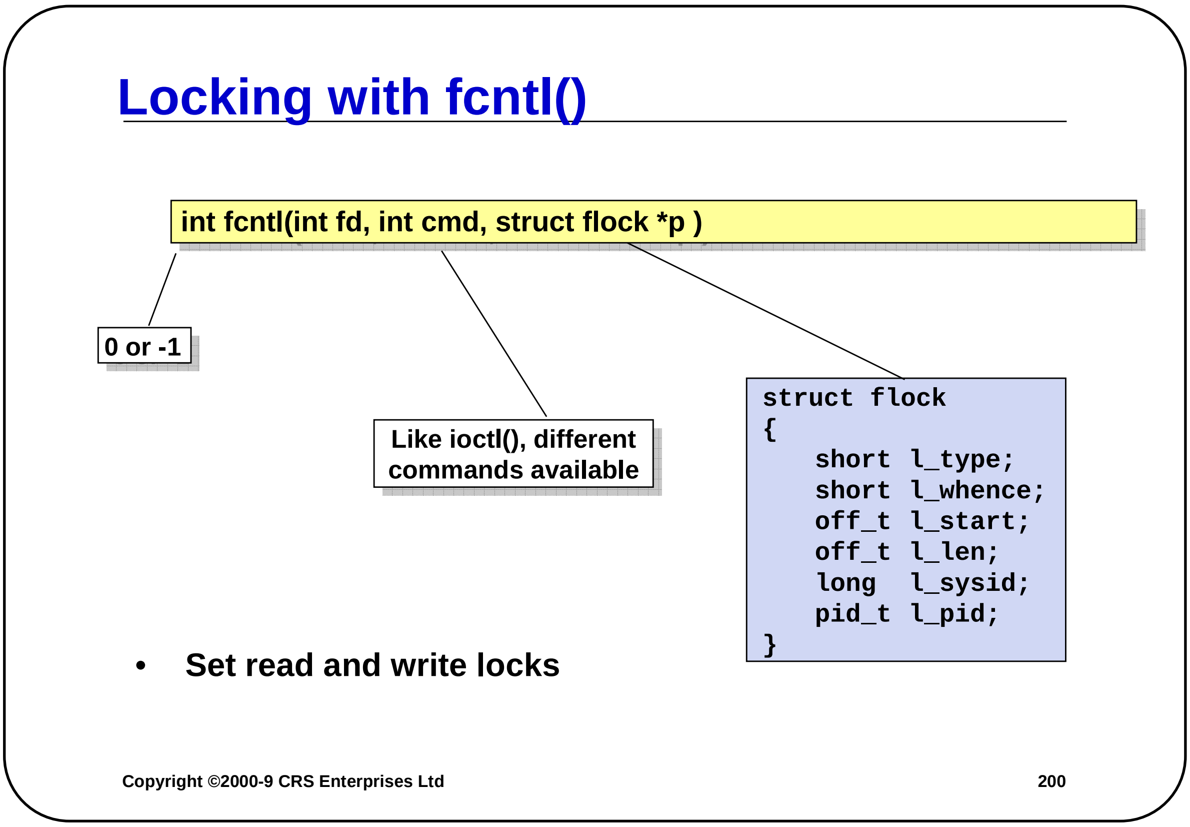
```
void OpenFile(void) {
   fd = open ("database", O_WRONLY);
}
void MoveFilePointer(void) {
   status = lseek (fd, record * RECORD_SIZE, SEEK_SET);
}
void Lock(void) {
   status = lockf (fd, F_TLOCK, RECORD_SIZE);
   if (status < 0) {
      printf("not safe to access record");
      read(fd, buffer, RECORD_SIZE);
   } else
      read(fd, buffer, RECORD_SIZE);
   }
void Unlock(void) {
   status = lockf (fd, F_ULOCK, RECORD_SIZE);
}
```

Blocks with mandatory locking

This example shows a program that locks and unlocks records in a file called "database". The file is assumed to contain fixed length records each of size RECORD_SIZE.

If this program is executed in several different windows, you can investigate the advisory locking mechanism. For example, process one locks records 0, 2 and 4. Process two can now only lock records 1 and 3. Attempting to lock one of the other records is refused by lockf(), unless process one unlocks the record first.

Notice the use of the F_TLOCK option when attempting to lock a record. This option is does not block; if a lock can't be set then the system call returns an error. If you fail to grab the lock and are using mandatory locking, the kernel will enforce the lock and any read() call will block. If you are using advisory locking, the kernel does not enforce the lock; any read() call will succeed! This later case is regarded as a programming error.



Locking with fcntl()

int fcntl(int fd, int cmd, struct flock *p)

0 or -1

Like ioctl(), different commands available

```
struct flock
{
    short l_type;
    short l_whence;
    off_t l_start;
    off_t l_len;
    long  l_sysid;
    pid_t l_pid;
}
```

- Set read and write locks

Copyright ©2000-9 CRS Enterprises Ltd 200

fcntl() is a system call that allows operations to be performed on open files. Among the possible operations is the manipulation of file and record locks.

Like the similar ioctl() function, flexibility in fcntl() is achieved at the price of a rather complex interface. fcntl() operates on an open file by its descriptor, and the action taken is determined by the cmd parameter. The third parameter is used depending on the nature of cmd; some commands require no parameters, others require one, and the type of the argument will depend on the command.

The locking commands use an argument of type struct flock*, which has the following fields:

l_type The type of the lock:

F_RDLCK for a read (or shared) lock

F_WRLCK for a write (or exclusive) lock

F_UNLCK for an unlock operation

l_whence Flag for the starting offset, relative to:

SEEK_SET = start of file

SEEK_CUR = current file position

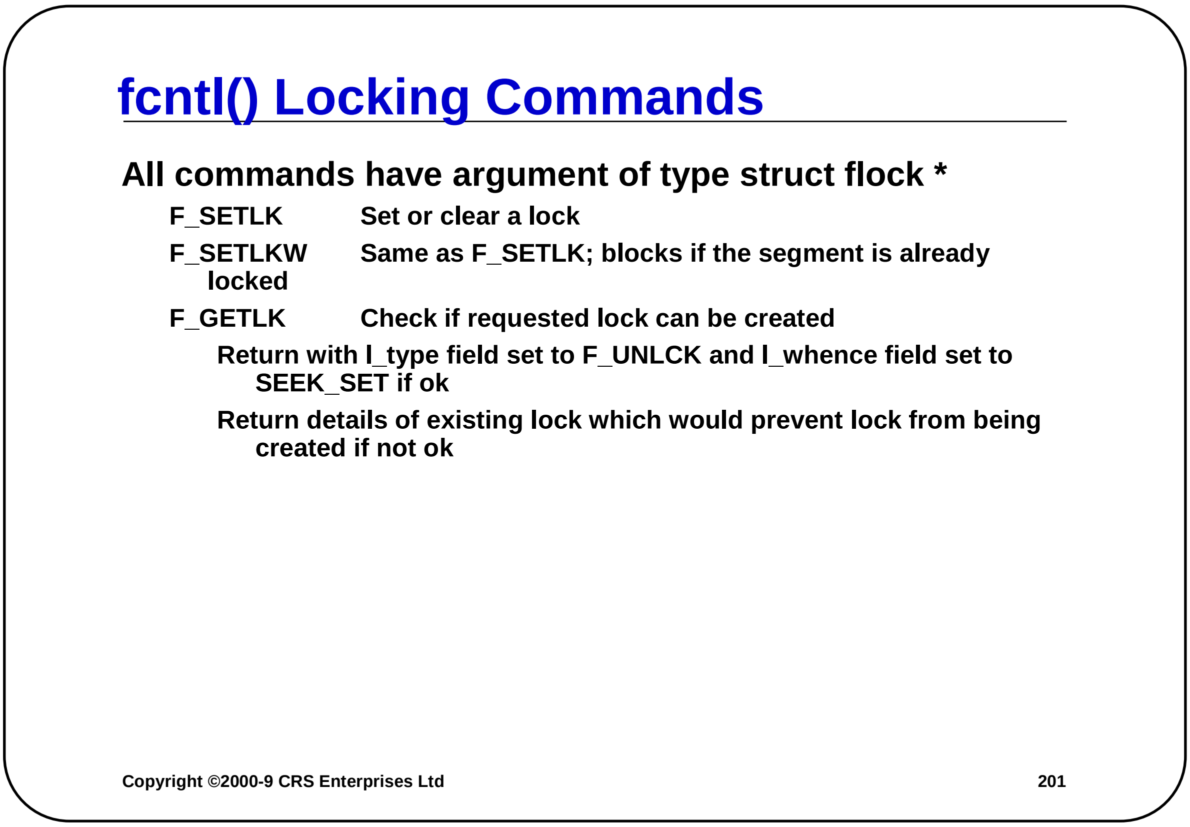
SEEK_END = end of file

l_start Relative offset of the start of the portion to be locked

l_len Size of portion; 0 means to end of file

l_sysid System id of locked portion

l_pid Pid of process holding lock; only returned by F_GETLK command



fcntl() Locking Commands

All commands have argument of type struct flock *

- **F_SETLK** Set or clear a lock
- **F_SETLKW** Same as F_SETLK; blocks if the segment is already locked
- **F_GETLK** Check if requested lock can be created
  - **Return with l_type field set to F_UNLCK and l_whence field set to SEEK_SET if ok**
  - **Return details of existing lock which would prevent lock from being created if not ok**

fcntl() has a wider use than just file locking, but here we look at the file-locking commands. All three commands take an argument which is a pointer to a struct flock. The fields of this structure are described on the previous slide.

F_SETLK This command is used to set or clear a lock on a portion of the file. The type of lock (read, write or a request to unlock) and details of the portion of the file are contained in the structure.

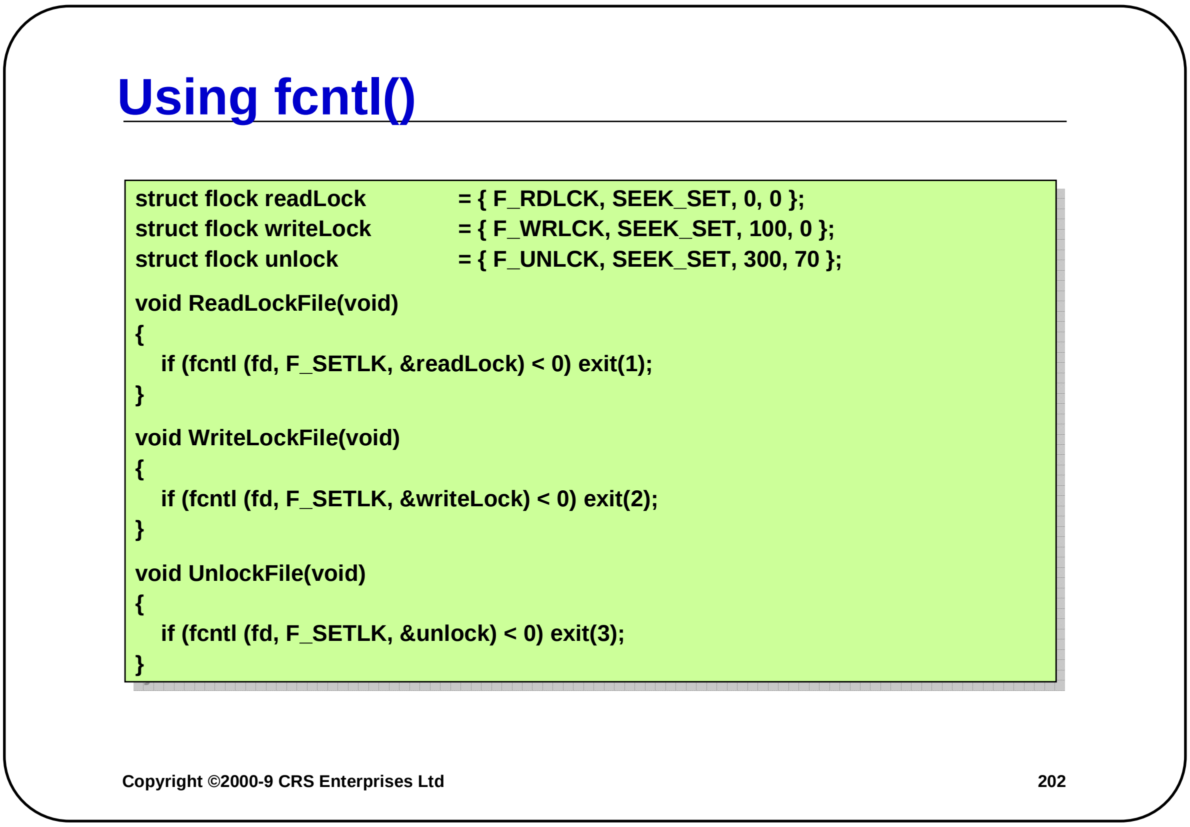
If the lock cannot be set for whatever reason (including another process already holding a lock, the call returns -1, with errno set to the correct value.

F_SETLKW This command carries out the same actions as F_SETLK, but will block if the requested portion of the file is already locked. The block continues until the file is free and the lock can be placed.

F_GETLK This command is used to find out if a lock can be placed on the required portion of the file, without actually making the lock. The lock request is made using the argument in the same way.

If the lock could be placed, the call will return the argument with the l_type field set to F_UNLCK and the l_whence field set to SEEK_SET.

If there is an existing lock that would prevent the lock from being placed, the argument on return will contain the details of the existing lock, including the process and system ids of the process holding the lock.



Using fcntl()

```
struct flock readLock          = { F_RDLCK, SEEK_SET, 0, 0 };
struct flock writeLock         = { F_WRLCK, SEEK_SET, 100, 0 };
struct flock unlock            = { F_UNLCK, SEEK_SET, 300, 70 };

void ReadLockFile(void)
{
   if (fcntl (fd, F_SETLK, &readLock) < 0) exit(1);
}

void WriteLockFile(void)
{
   if (fcntl (fd, F_SETLK, &writeLock) < 0) exit(2);
}

void UnlockFile(void)
{
   if (fcntl (fd, F_SETLK, &unlock) < 0) exit(3);
}
```

These routines show how `fcntl()` can be used to lock and unlock a file.

fcntl (fd, F_SETLK, &readLock) uses the information in the structure *readLock* to set a lock. By examining *readlock* we can see that this call will set a read lock.

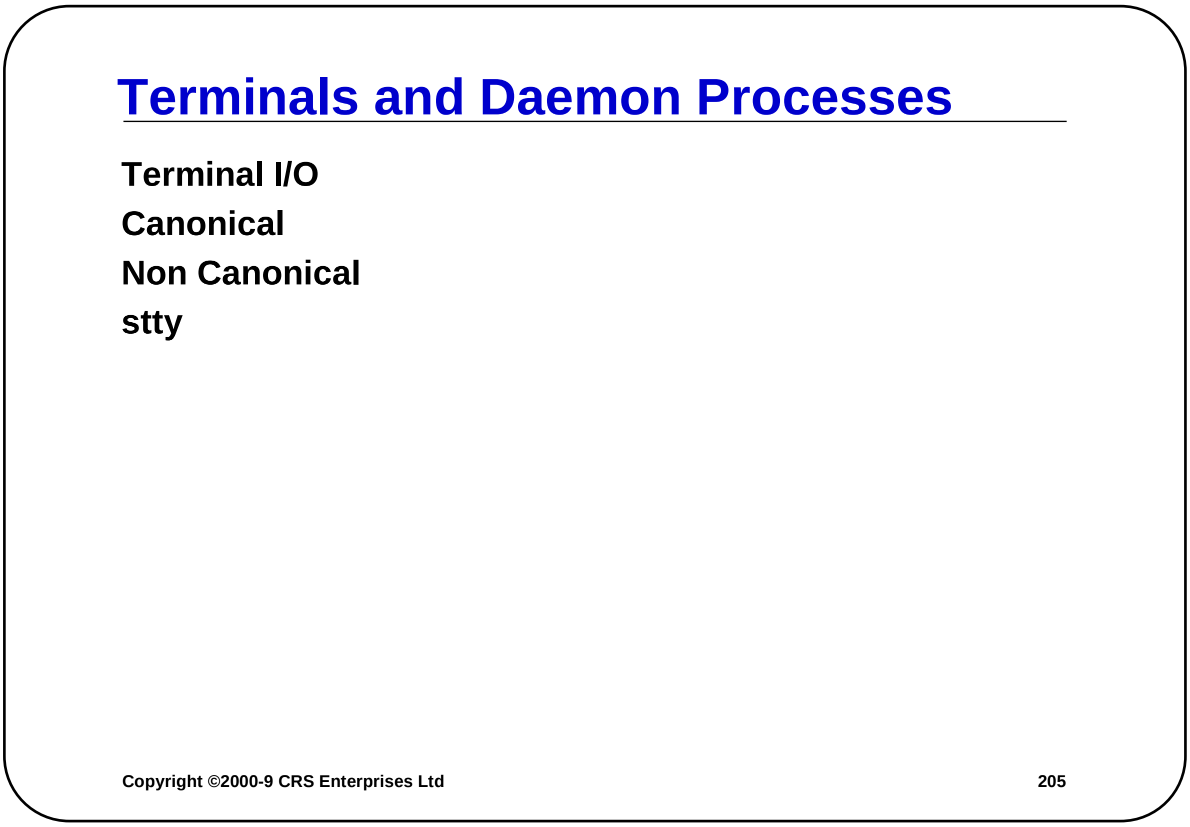
fcntl (fd, F_SETLK, &writeLock) uses the information in the structure *writeLock* to set a lock. This time the call will set a write lock. The call

fcntl (fd, F_SETLK, &unlock) uses the information in the structure *unlock* to set a lock. This call does not set a lock at all; it unlocks the file instead!

15

Terminals

15



Terminals and Daemon Processes

Terminal I/O

Canonical

Non Canonical

stty

205

The chapter investigates terminal devices as used through the POSIX termios interface. Terminal devices include terminals, printers, modems and psuedo terminals.

Terminals can be driven in Canonical (line by line) or Non Canonical (raw) mode and have numerous attributes. Attributes can be set using system calls or by the Unix command stty.

Overview of Terminal I/O

Applies to terminals and pseudo terminals

Also devices connected over serial lines, e.g. modems, printers

Canonical mode

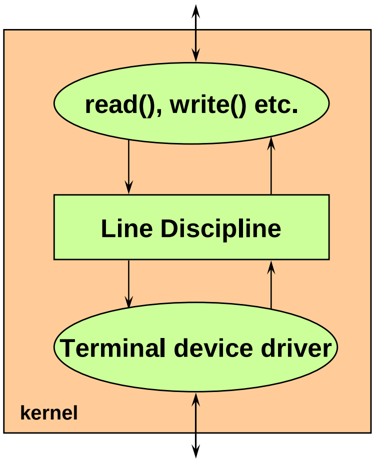
Characters processed as lines

Simple character conversions

Non Canonical mode

No grouping into lines

No character conversions



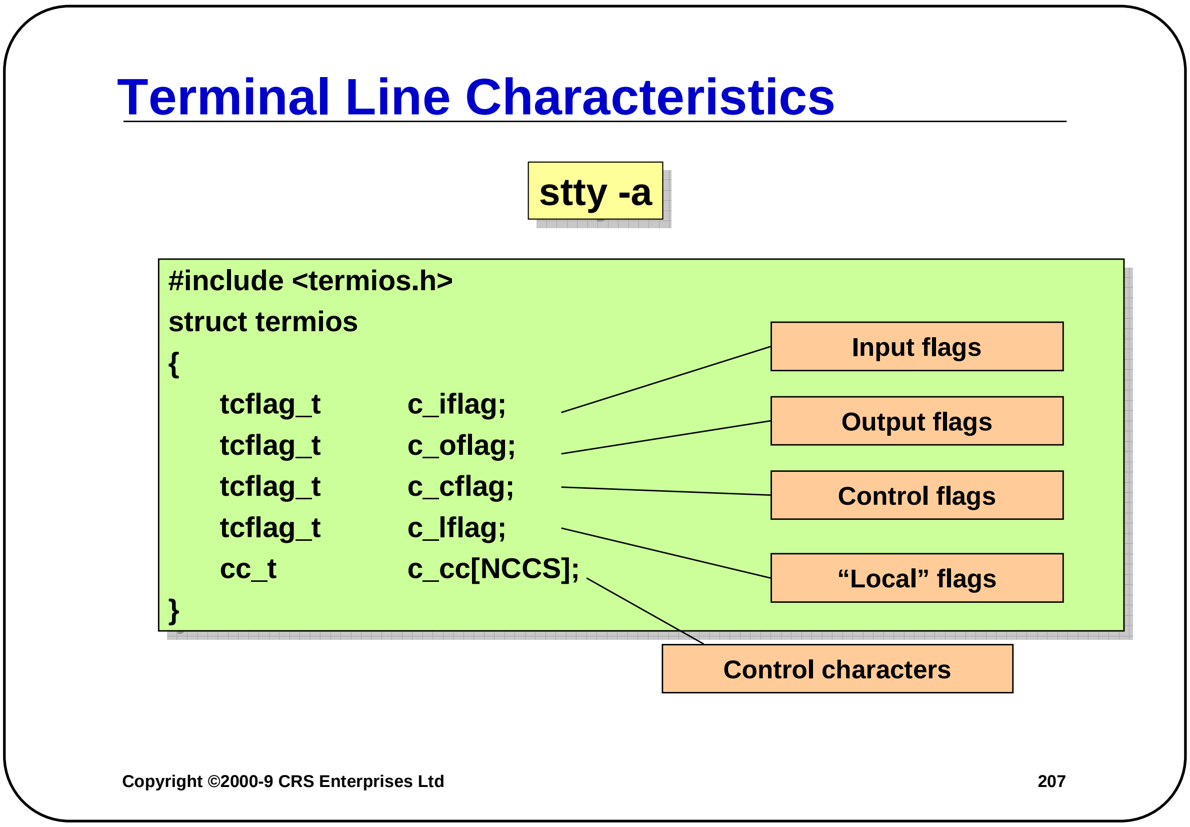
In Unix, terminal I/O processing is used to provide an interface to “terminal” devices such as physical terminals and pseudo terminals, but also to control more general devices accessed across serial lines, such as modems and printers. There are generally two modes of operation in terminal I/O:

Canonical mode

> Characters read from the terminal device driver are grouped into lines (i.e. buffered until a newline character is read). This is usually handled by a module called the Line Discipline. Similarly, output characters are handled as lines.
>
> Within lines, certain control characters are recognised and cause processing on the characters before they are passed to the reading process, for example ^u to kill a line, ^c to generate an interrupt signal. Carriage returns are always converted to linefeeds ('\n').

Non Canonical mode

> Characters are not grouped into lines and no character conversions take place. All I/O is raw. This allows a more flexible form of input from the terminal, for example that used by programs like vi.



Terminal Line Characteristics

stty -a

```
#include <termios.h>
struct termios
{
    tcflag_t        c_iflag;
    tcflag_t        c_oflag;
    tcflag_t        c_cflag;
    tcflag_t        c_lflag;
    cc_t            c_cc[NCCS];
}
```

Input flags

Output flags

Control flags

“Local” flags

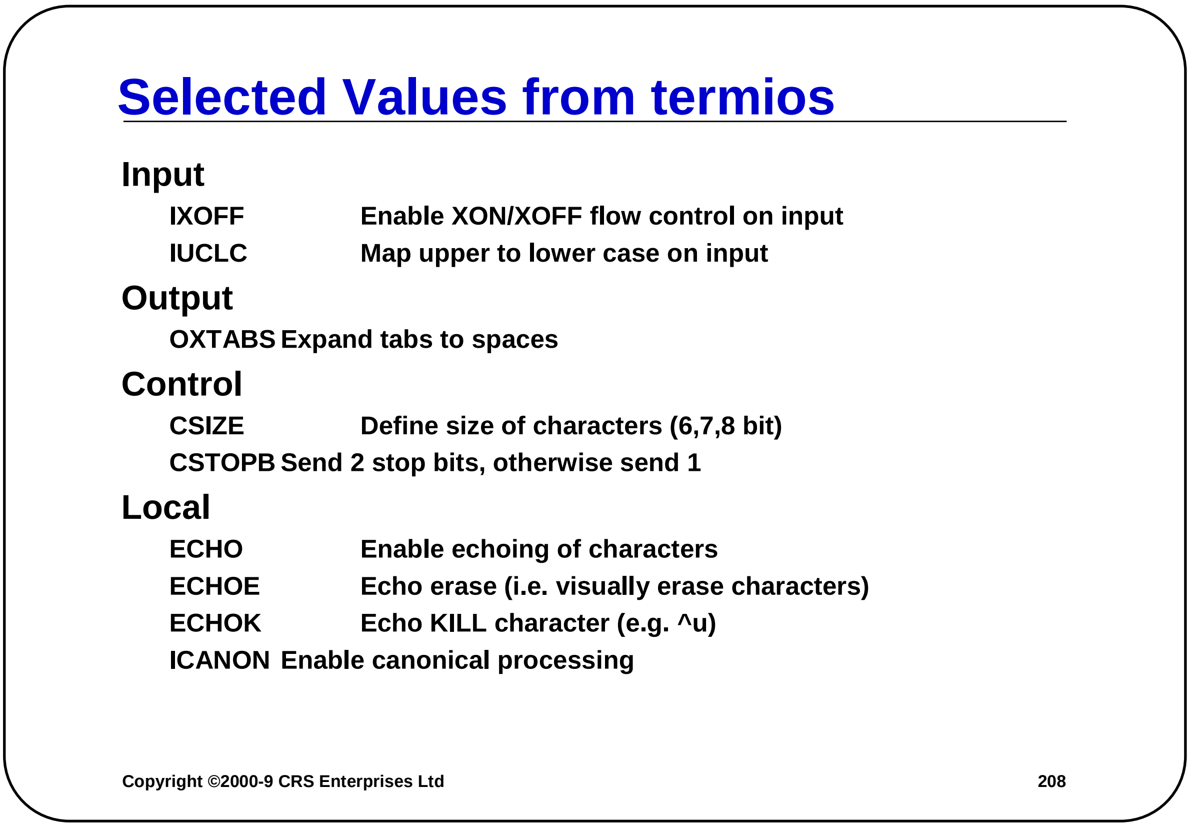
Control characters

The termios interface to terminals is defined by the termios structure. The various fields in the data structure control different aspects of the terminal driver's behaviour. Each of the flag fields contains bits that are set or cleared to enable or disable certain functionality. The c_cc array contains the special control characters used for erase, interrupt, quit and so on.

The Unix command

```
stty -a
```

can be used to print out the contents of the termios structure.



Selected Values from termios

Input

IXOFF	Enable XON/XOFF flow control on input
IUCLC	Map upper to lower case on input

Output

OXTABS	Expand tabs to spaces

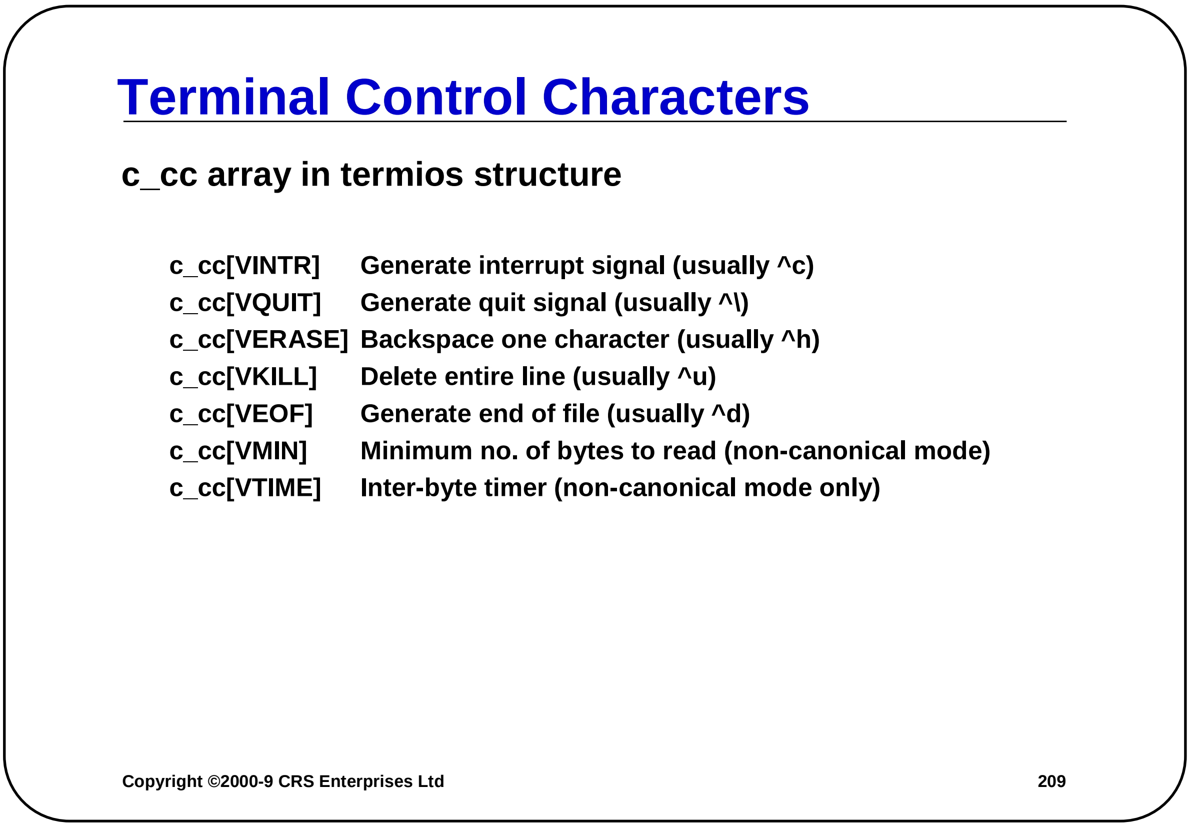
Control

CSIZE	Define size of characters (6,7,8 bit)
CSTOPB	Send 2 stop bits, otherwise send 1

Local

ECHO	Enable echoing of characters
ECHOE	Echo erase (i.e. visually erase characters)
ECHOK	Echo KILL character (e.g. ^u)
ICANON	Enable canonical processing

This slide shows some of the more common flags from the termios structure. There are many more as detailed on the manual page.



Terminal Control Characters

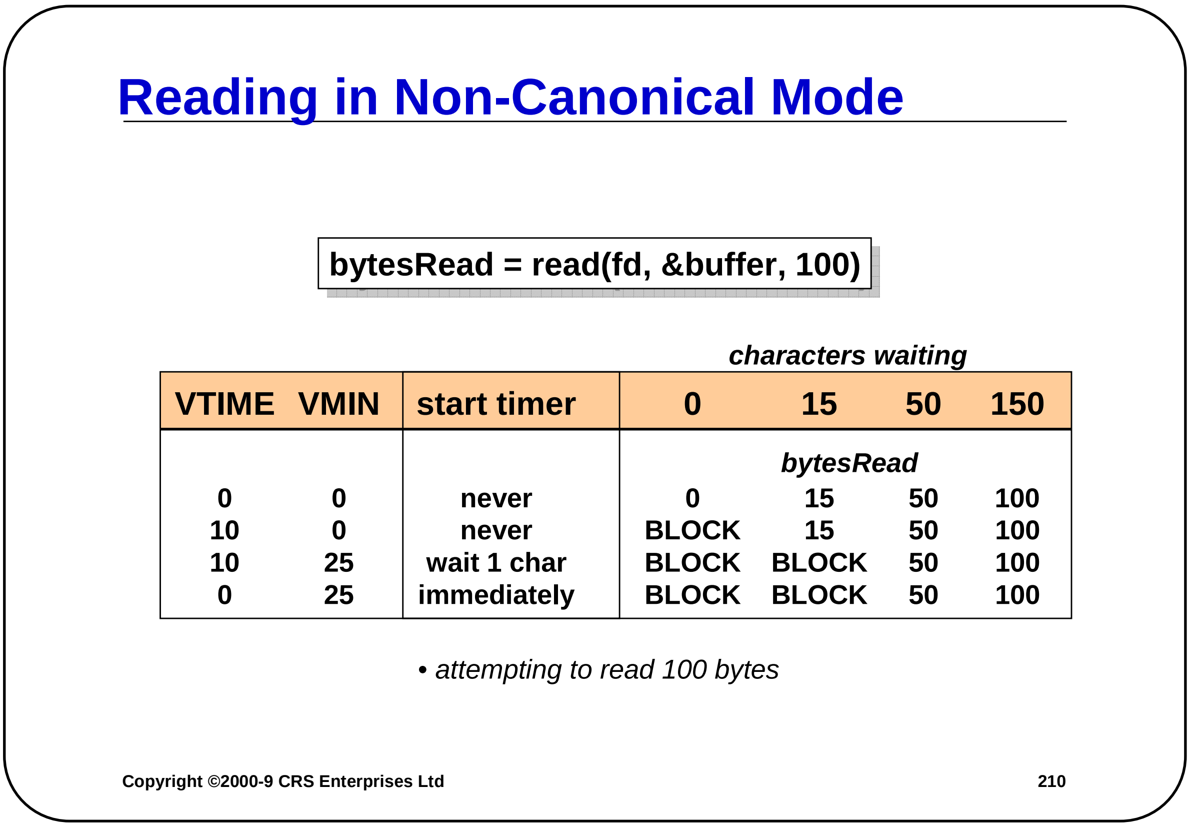
c_cc array in termios structure

c_cc[VINTR]	Generate interrupt signal (usually ^c)
c_cc[VQUIT]	Generate quit signal (usually ^\)
c_cc[VERASE]	Backspace one character (usually ^h)
c_cc[VKILL]	Delete entire line (usually ^u)
c_cc[VEOF]	Generate end of file (usually ^d)
c_cc[VMIN]	Minimum no. of bytes to read (non-canonical mode)
c_cc[VTIME]	Inter-byte timer (non-canonical mode only)

When operating in canonical mode, characters are grouped into lines (delimited by CR) before being passed to the reading process. Before this, the line discipline module recognises several control characters in the input stream; these cause certain actions to be taken. The list of control characters and their actions is held in the c_cc array of the termios structure.

You can change these characters, just as you can change other aspects of the terminal line's functionality. Some of the common elements of the array are shown on the slide, together with two additional elements which are only used in non-canonical mode processing. These are discussed in detail later in the chapter.

You can examine and change the values of these and other terminal characteristics from the command line using the stty command.



Reading in Non-Canonical Mode

```
bytesRead = read(fd, &buffer, 100)
```

			characters waiting			
VTIME	VMIN	start timer	0	15	50	150
			bytesRead			
0	0	never	0	15	50	100
10	0	never	BLOCK	15	50	100
10	25	wait 1 char	BLOCK	BLOCK	50	100
0	25	immediately	BLOCK	BLOCK	50	100

- *attempting to read 100 bytes*

When reading in canonical mode, characters are returned to the process when CR is pressed. In non-canonical mode we need another way of signalling when characters are to be returned. We use the VMIN and VTIME special characters for this. There are four possibilities:

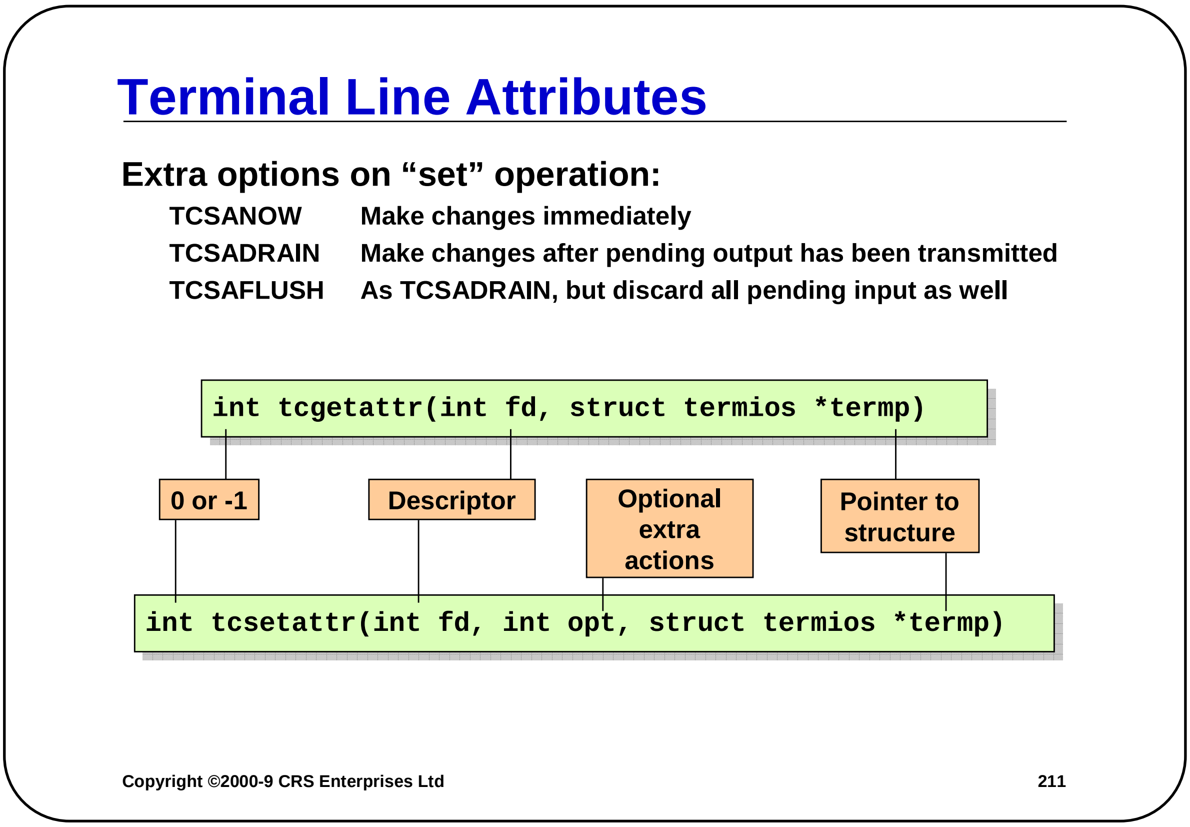
1. VMIN > 0 and VTIME > 0: here the read blocks until the first character is read. Then we start an inter-byte timer, specified to expire in VTIME/10 seconds. The read will return when either we have read the required number of characters (i.e. VMIN), or when the timer expires, in which case we return the number of characters that have been read up until now.

2. VMIN > 0 and VTIME == 0: here the read will block until the required number of characters (VMIN) is read. Note that this may block the process indefinitely.

3. VMIN == 0 and VTIME > 0: the interbyte timer is started as soon as the read is issued. It expires in VTIME/10 seconds. The read returns when the first byte arrives, or when the timer expires (in which case it returns 0).

4. VMIN ==0 and VTIME == 0: here the read will return immediately. If there are characters to be read, then they are returned (up to the number requested). Otherwise read returns 0 at once.

Note that no processing of special characters takes place in non-canonical mode. If you enter ^c, ^u etc. they will appear in the input stream.



Terminal Line Attributes

Extra options on “set” operation:

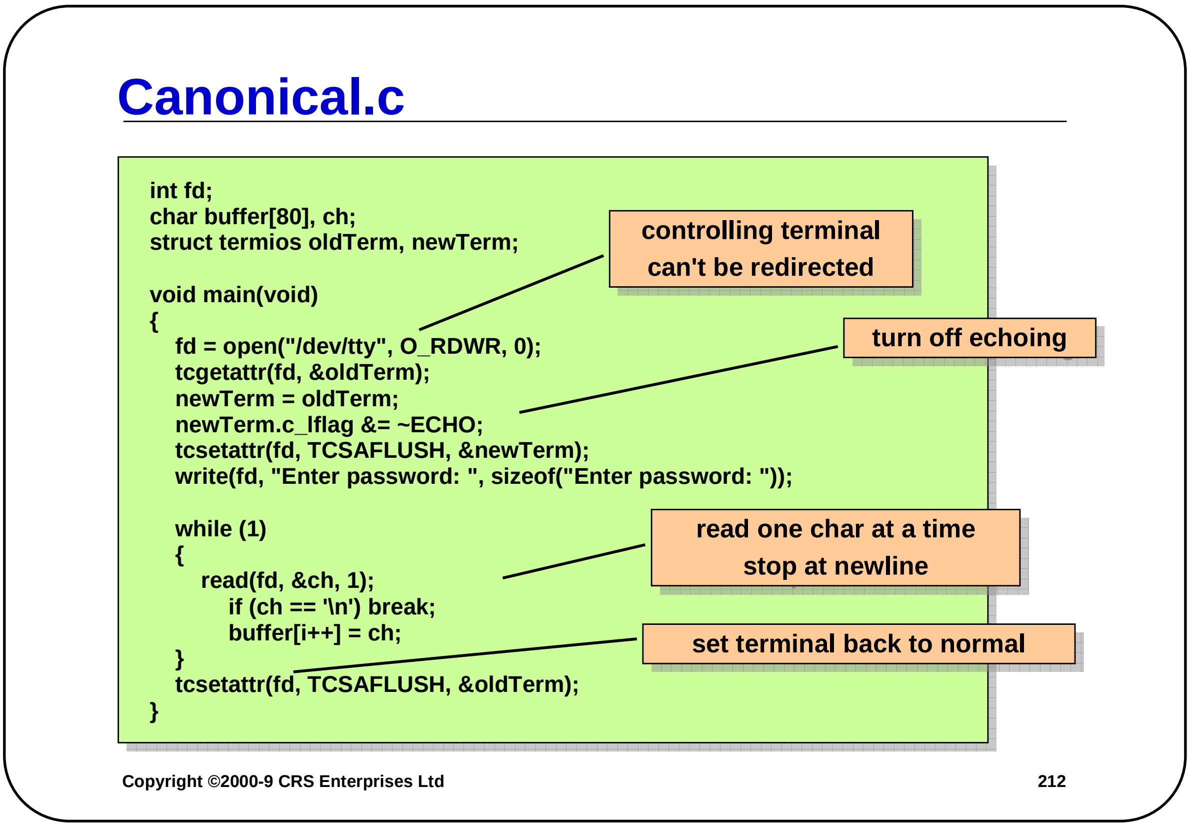
TCSANOW	Make changes immediately
TCSADRAIN	Make changes after pending output has been transmitted
TCSAFLUSH	As TCSADRAIN, but discard all pending input as well

```
int tcgetattr(int fd, struct termios *termp)
```

```
int tcsetattr(int fd, int opt, struct termios *termp)
```

To examine the current terminal characteristics, use the tcgetattr() function. This is a high-level wrapper function around ioctl() functionality and fills in the fields of the termios structure (pointed to by its termp parameter) to contain the current settings.

To change settings, use the tcsetattr() function. This has an interface similar to tcgetattr(), except that you can specify additional information through the opt argument. This allows you to delay changing values until all pending output has been sent, or additionally to discard any pending input.



Canonical.c

```
int fd;
char buffer[80], ch;
struct termios oldTerm, newTerm;

void main(void)
{
    fd = open("/dev/tty", O_RDWR, 0);
    tcgetattr(fd, &oldTerm);
    newTerm = oldTerm;
    newTerm.c_lflag &= ~ECHO;
    tcsetattr(fd, TCSAFLUSH, &newTerm);
    write(fd, "Enter password: ", sizeof("Enter password: "));

    while (1)
    {
        read(fd, &ch, 1);
            if (ch == '\n') break;
            buffer[i++] = ch;
    }
    tcsetattr(fd, TCSAFLUSH, &oldTerm);
}
```

controlling terminal can't be redirected

turn off echoing

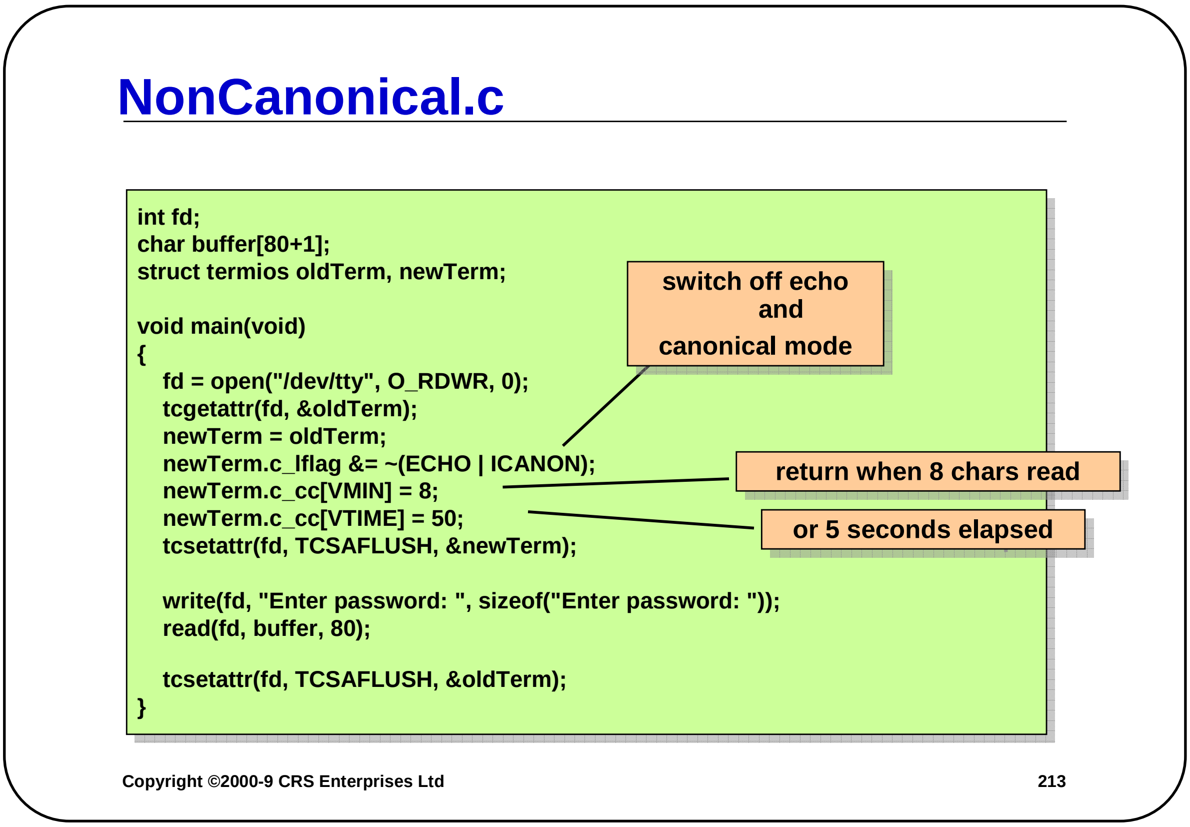
read one char at a time stop at newline

set terminal back to normal

This code fragment demonstrates how to turn off the (normally automatic) echoing of characters read, but still retaining other facilities of canonical-mode processing.

The code prompts for the user to enter a "password", then reads it while echoing is turned off. The code examines the characters one at a time, stopping when newline is read.

Note that because we are working in Canonical mode, the line discipline buffers the terminal input until the newline character is pressed. This means that the first read() call blocks until the user presses newline.



NonCanonical.c

```
int fd;
char buffer[80+1];
struct termios oldTerm, newTerm;

void main(void)
{
   fd = open("/dev/tty", O_RDWR, 0);
   tcgetattr(fd, &oldTerm);
   newTerm = oldTerm;
   newTerm.c_lflag &= ~(ECHO | ICANON);
   newTerm.c_cc[VMIN] = 8;
   newTerm.c_cc[VTIME] = 50;
   tcsetattr(fd, TCSAFLUSH, &newTerm);

   write(fd, "Enter password: ", sizeof("Enter password: "));
   read(fd, buffer, 80);

   tcsetattr(fd, TCSAFLUSH, &oldTerm);
}
```

switch off echo and canonical mode

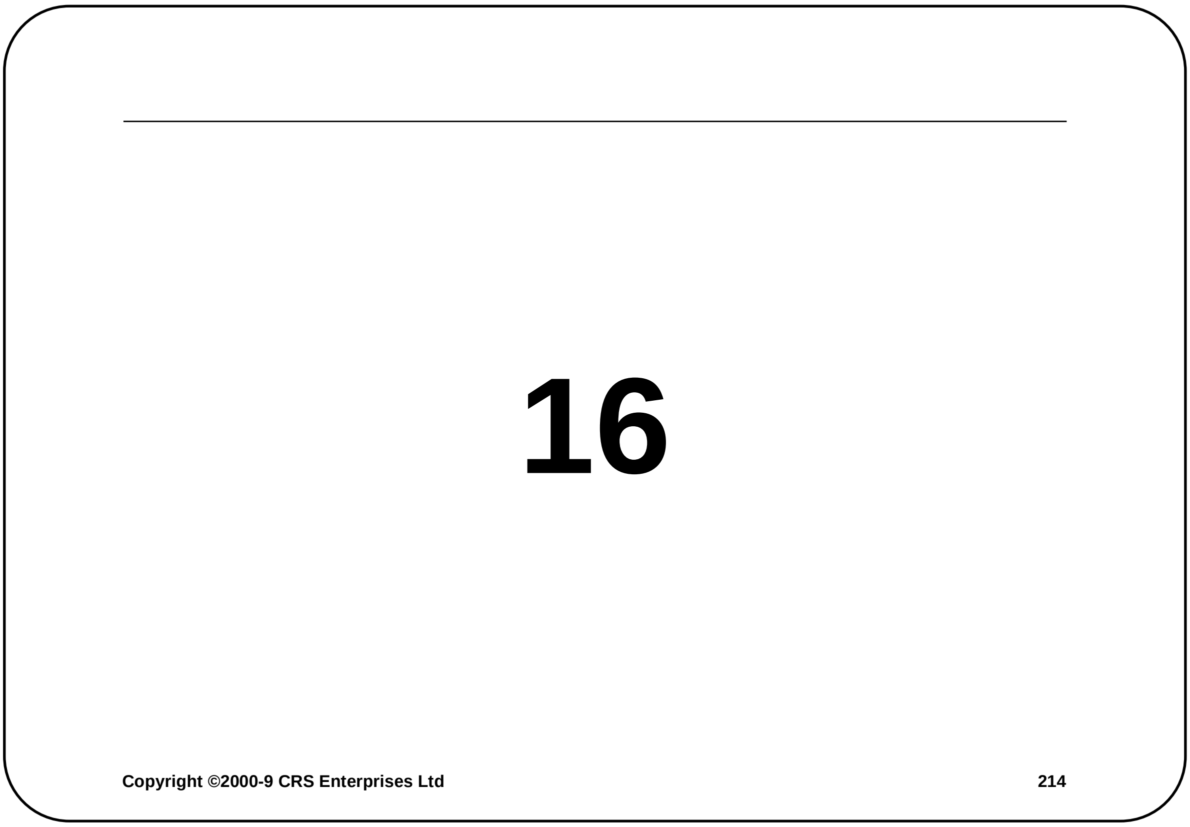
return when 8 chars read

or 5 seconds elapsed

In this example, we disable canonical-mode processing as well as the echo. The VMIN and VTIME control characters are set so that the read() will return after 8 characters have been read or when 5 seconds have elapsed from the start of the timeout (the timeout starts after the 1st character is entered).

Note that read() only returns at most VMIN characters and not return the 80 characters we requested unless we type ahead (i.e. there are more than VMIN characters in the line discipline buffer before we issue the read() call).

As before, the user is prompted to enter a password which is then displayed. Remember that, because canonical processing is disabled, you cannot use ^h (backspace) to correct typing errors. Each ^h character you type counts as one more character read. If you wish to deal with such features, you must add the necessary processing yourself.



16

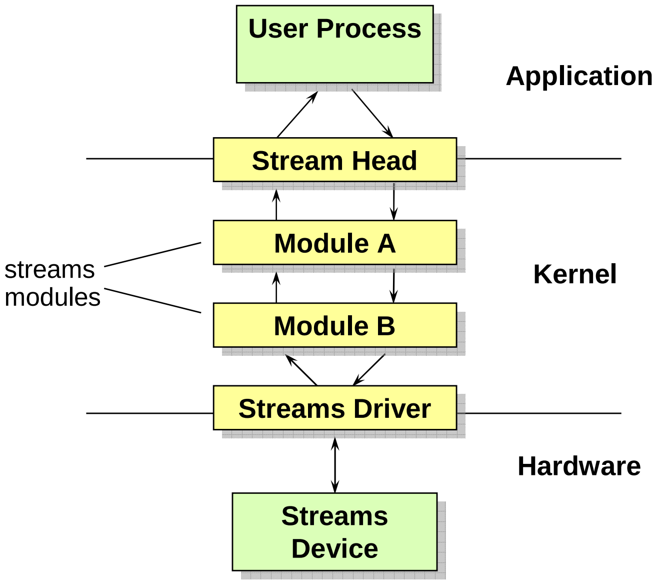
214

STREAMS

16

215

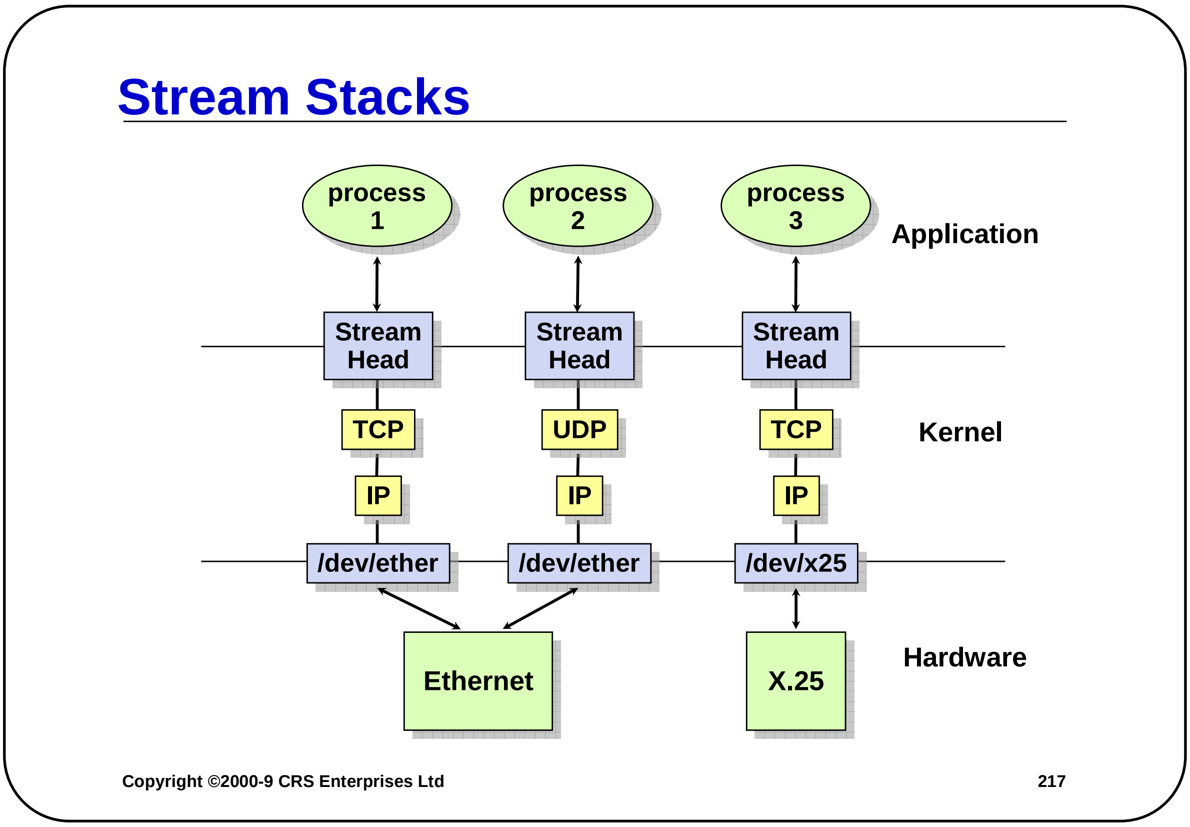
Streams



Unix STREAMS I/O were introduced into System V to modularise the implementation of certain device drivers. We write STREAMS in upper case to avoid confusion with the stream I/O model used by the run time library.

STREAMS device drivers are written as a series of STREAMS modules. These modules form a pipeline inside the kernel. User processes can use the read() and write() system calls to interface with the stream head. Data written to the stream head is passed down a succession of STREAMS modules until it reaches the STREAMS driver module. This module communicates directly with the hardware. Input data enters the kernel from the hardware via the STREAMS driver and is passed up through each module until it appears at the STREAMS head. The read() system call passes this data to the user process.

Since STREAMS modules are device drivers, we will not be concerned with writing these modules. However, we will be interested in interfacing with STREAMS devices. In particular we will investigate the poll() system call (only available on STREAMS devices) and the pseudo terminal STREAMS device. The pseudo terminal device is utilised in many applications nowadays, including network logins, the Unix script command and emulating terminals under X Windows.



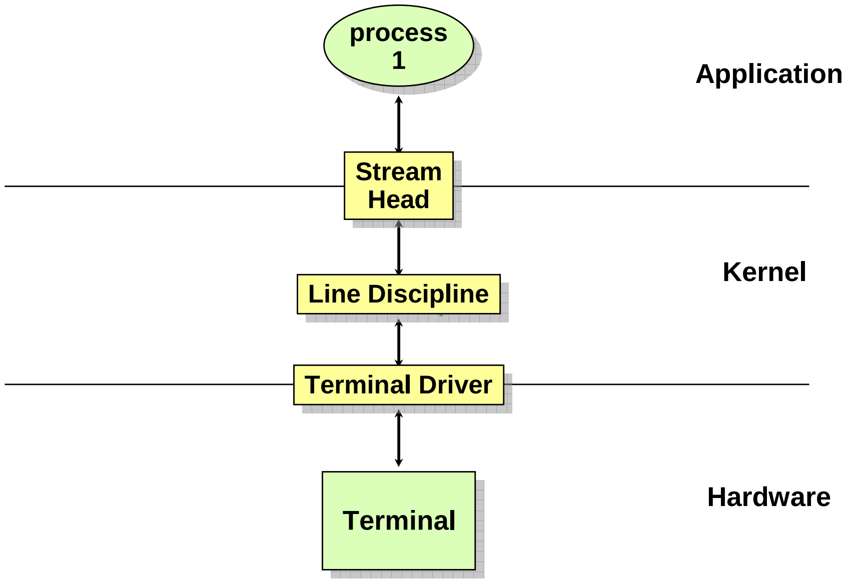
Stream Stacks

STREAMS modules are designed to allow device driver software to modularised in order to simplify their reuse in other device drivers. The streams modules are added to the kernel using the ioctl() system call (see later). Modules can also be removed from the kernel using ioctl(). Since modules are added and removed as last-in first-out, streams modules are often referred to as a stream stack.

STREAMS were originally designed to modularise network device drivers. In the above example processes 1 and 2 use an ethernet connection to communicate across the network. Process 1 pushes an IP and a TCP module onto its stream stack because it intends to use the TCP/IP protocol. Process 2 pushes an IP and a UDP module onto its stream stack because it intends to use the UDP/IP protocol. Process 3 also intends to use TCP/IP, but across an X.25 interface; the process simply opens a different device driver and pushes IP and TCP modules as before.

Using streams modules makes setting up TCP/IP and UDP/IP much simpler than in earlier versions of Unix.

Terminals



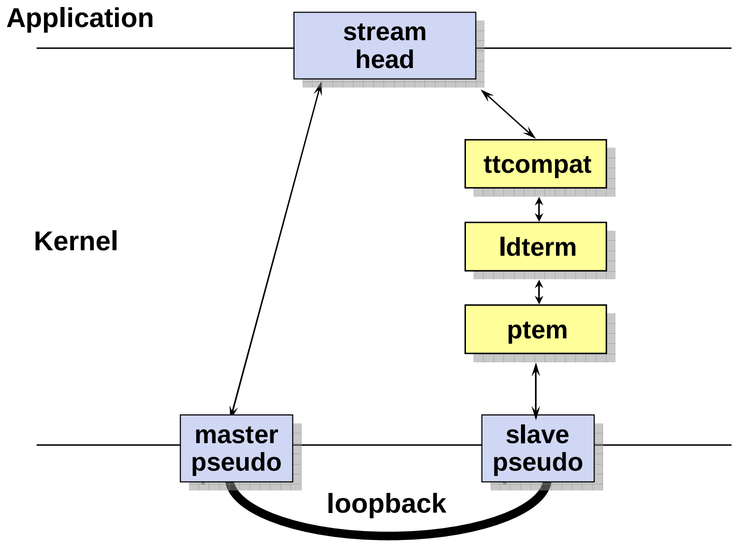
Terminal device drivers are usually written as STREAMS devices in SVR4; the terminal line discipline is implemented as STREAMS modules.

Recall that the line discipline can be set to canonical or non canonical mode:

In canonical (cooked) mode, input and output are processed as complete lines, simple errors in the input data can be corrected using backspace and other editing keys, carriage returns are translated to newline characters.

In non-canonical (raw) mode, input and output are not treated as complete lines. Characters are input as groups (using the c_cc[VMIN] flag) with timeouts (using the c_cc[VMIN] flag).

Pseudo Terminals



A pseudo terminal STREAMS device behaves in a similar manner to a real terminal device, but is implemented as a loopback driver within the kernel. The loopback driver consists of two STREAMS devices: a master pseudo and a slave pseudo. The slave pseudo is supported by 3 STREAMS modules.

The line disciple (ldterm) on the slave side is sandwiched by 2 other STREAMS modules. The ptem module required if the terminal is to emulate a real terminal and the ttcompat module is required for compatibility between System V and BSD ioctl() calls.

The device name of the master side of the pseudo terminal will be something like:

/dev/pty1

/dev/pty4

/dev/pty8

The device name of the slave side of the pseudo terminal will be something like:

/dev/tty1

/dev/tty4

/dev/tty8

Pseudo Terminals

```
void OpenMasterAndSlave (void)
{
   masterFd = open("/dev/ptmx", O_RDWR);
   grant(masterFd);
   unlockpt(masterFd);

   slaveDeviceName = ptsname(masterFd);
   slaveFd = open(slaveDeviceName, O_RDWR);

   ioctl(slaveFd, I_PUSH, "ttcompat");
   ioctl(slaveFd, I_PUSH, "ptem");
   ioctl(slaveFd, I_PUSH, "ldterm");
}
```

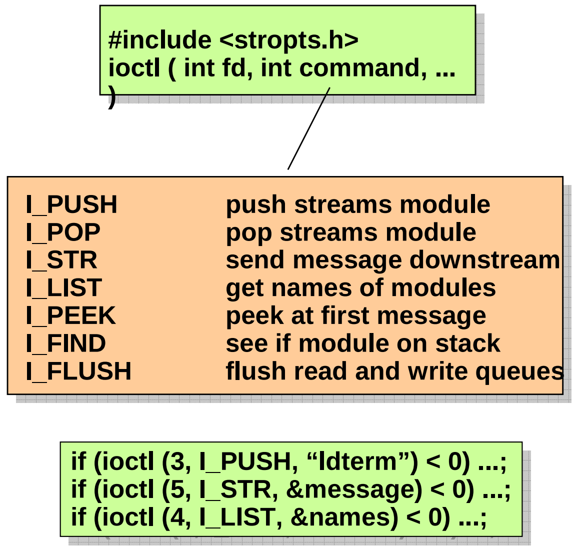
The master pseudo terminal is opened in the parent process using the clone device /dev/ptmx. Opening this device creates a master-slave pair and returns the file descriptor of the master. The clone device only opens the master side of the pseudo terminal, it does not open the slave. Use the ptsname() call to extract the device name of the slave (e.g. /dev/pts5) so that it can be opened later.

Once the master side of the pseudo is opened, the slave side must be unlocked (unlockpt()) and have read-write access granted (grantpt()). Note that these operations are performed before the slave pseudo is opened.

After the slave device is opened, it must be configured by pushing 3 streams modules onto the STREAMS device:

ttcompat	compatibility module for BSD and SVR4 ioctls
ptem	pseudo terminal emulation mode
ldterm	terminal line discipline

ioctl



```
#include <stropts.h>
ioctl ( int fd, int command, ...
)
```

I_PUSH	push streams module
I_POP	pop streams module
I_STR	send message downstream
I_LIST	get names of modules
I_PEEK	peek at first message
I_FIND	see if module on stack
I_FLUSH	flush read and write queues

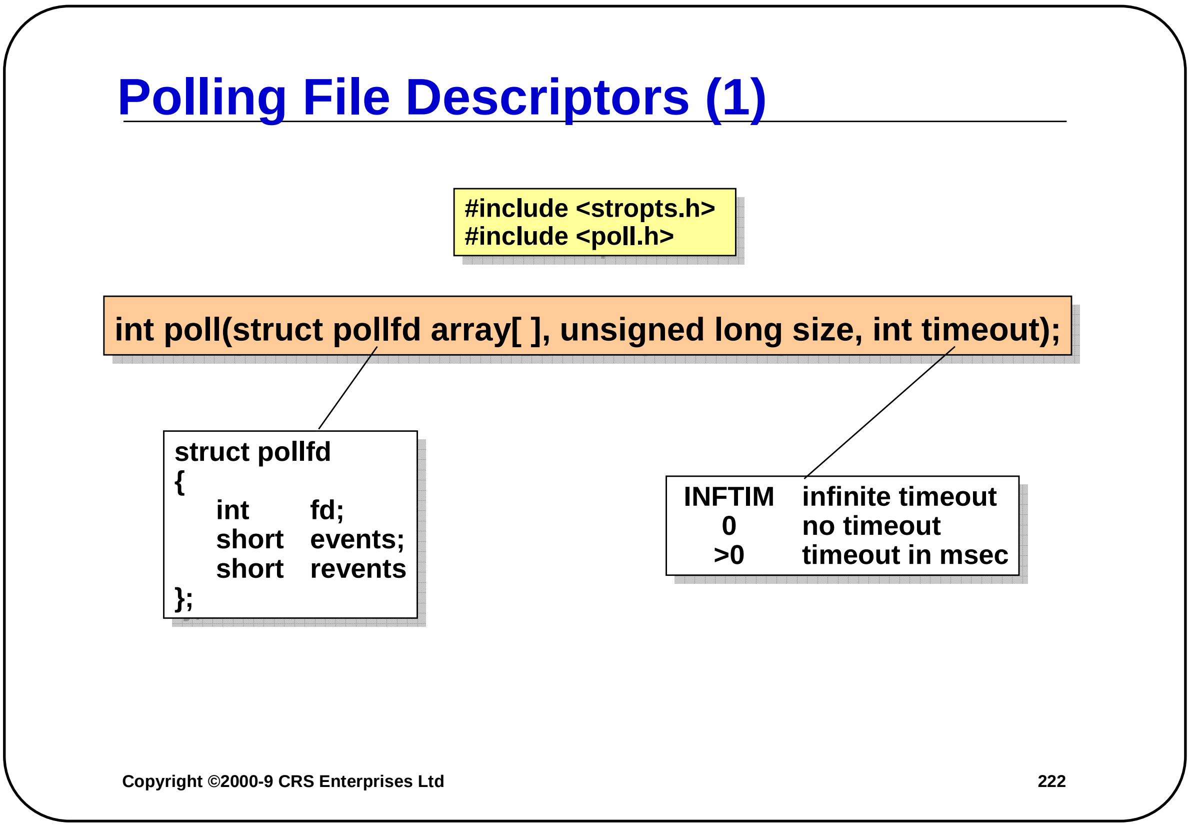
```
if (ioctl (3, I_PUSH, “ldterm”) < 0) ...;
if (ioctl (5, I_STR, &message) < 0) ...;
if (ioctl (4, I_LIST, &names) < 0) ...;
```

STREAMS devices are configured and managed using ioctl() system calls. Each ioctl() call takes a STREAMS file descriptor as its first parameter. The second parameter to ioctl() is a an integer representing a command. The third parameter can have different meaning, dependent on the second parameter. Many commands are available (see streamio in the manual pages); a selection of commands is shown below:

ioctl (3, I_PUSH, "ldterm") Push the “ldterm” streams module onto the stream stack associated with file descriptor 3.

ioctl (5, I_STR, &message) Send a stream’s message downstream towards the device driver. This is equivalent to a write on a non streams device.

ioctl (4, I_LIST, &names) Extract the names of all modules on the stream stack.



Polling File Descriptors (1)

```
#include <stropts.h>
#include <poll.h>
```

```
int poll(struct pollfd array[ ], unsigned long size, int timeout);
```

```
struct pollfd
{
    int     fd;
    short   events;
    short   revents
};
```

INFTIM	infinite timeout
0	no timeout
>0	timeout in msec

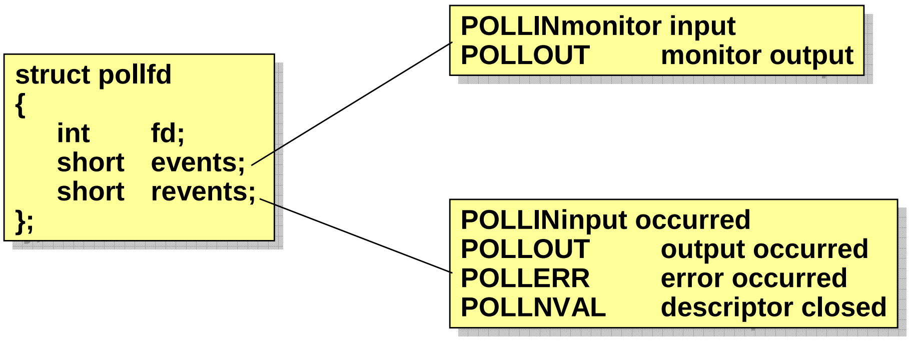
The poll() system call allows us to monitor several file descriptors simultaneously. poll() originally worked only with STREAMS devices, but now has been extended to cover most SVR4 devices .

To use poll() you must first set up an array of descriptors in which you have interest. Each element of the array is a structure:

```
struct pollfd
{
        int         fd;
        short       events;
        short       revents
};
```

The events field specifies which events you want to monitor. poll() will block until one of these events occurs; when poll() does return, the revents field is filled in with information about which event occurred.

Polling File Descriptors (2)



```
struct pollfd
{
     int      fd;
     short    events;
     short    revents;
};
```

POLLIN	monitor input
POLLOUT	monitor output

POLLIN	input occurred
POLLOUT	output occurred
POLLERR	error occurred
POLLNVAL	descriptor closed

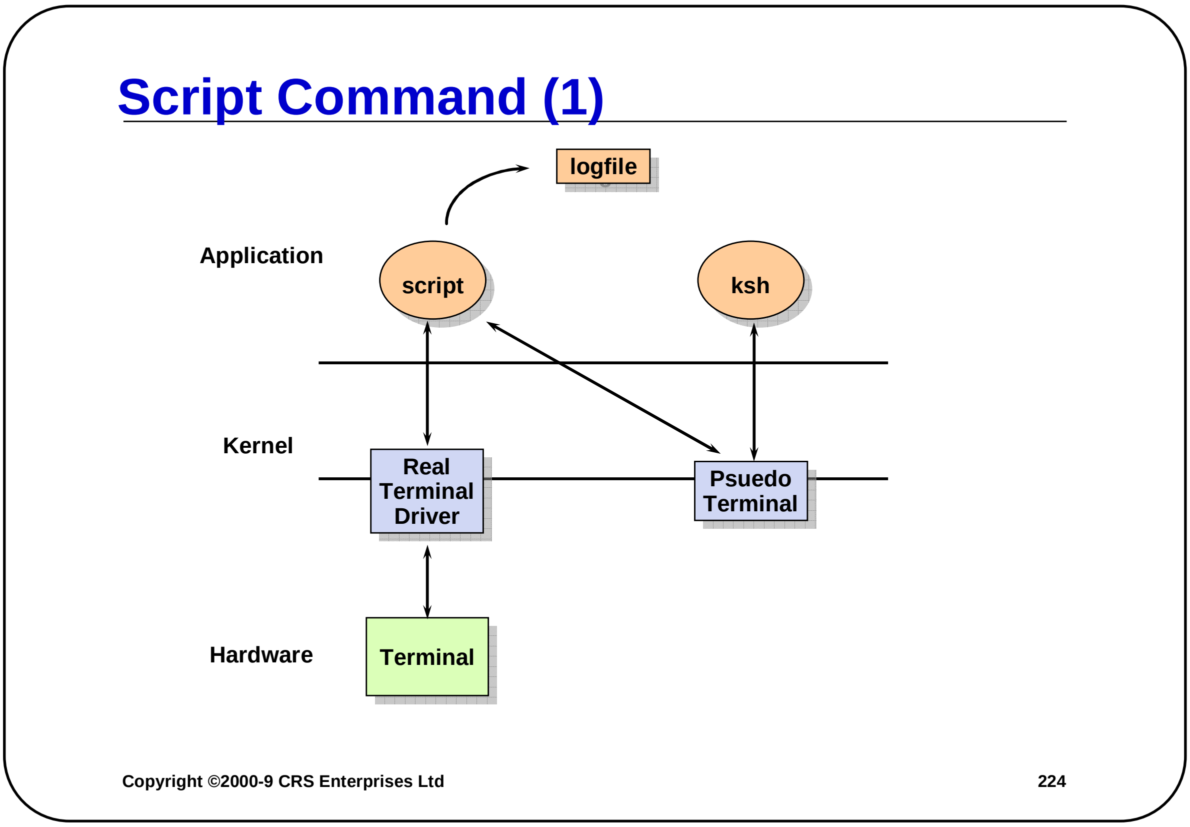
The events and revents fields of the struct pollfd can take on several values:

events

POLLIN	monitor input on given file descriptor.
POLLOUT	monitor output on given file descriptor.

revents

POLLIN	input has occurred on given file descriptor.
POLLOUT	output has occurred on given file descriptor.
POLLERR	error occurred on given file descriptor.
POLLNVAL	file descriptor not valid (probably closed).



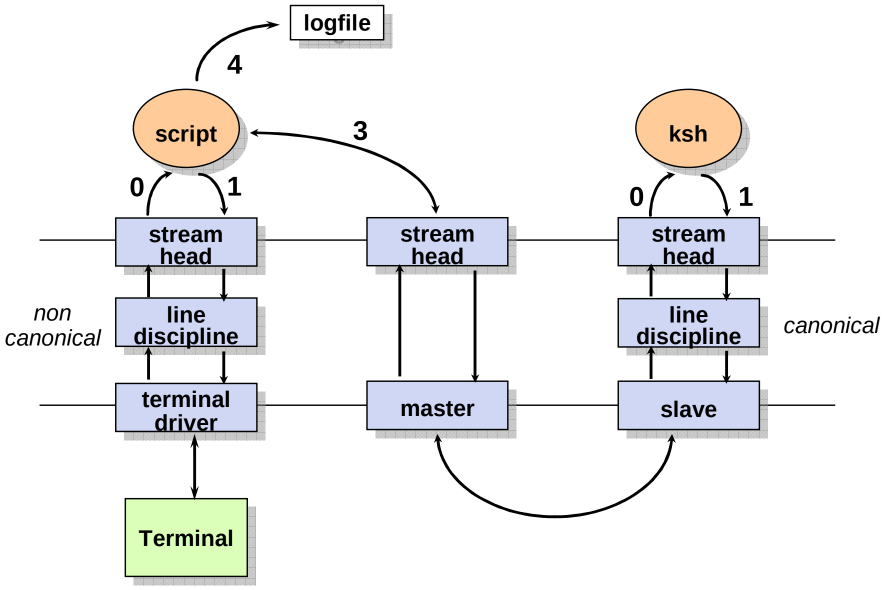
Script Command (1)

As an example of using the pseudo terminal device, consider the Unix script command. This command logs all input and output at a terminal session to a disk file.

It is tempting to think that all the script command need do is to redirect all activity on file descriptors 0, 1 and 2 to the log file. In practice this approach fails because the kernel buffering for disk and terminal devices is different. I/O to disk files is buffered in pages and does not support canonical and non-canonical terminal buffering. Enter the pseudo terminal device. This device behaves like a terminal and does support canonical and non-canonical buffering.

The script program first opens a pseudo terminal and then uses fork() and exec() to run a ksh that talks to the pseudo terminal. Data from the pseudo terminal is routed to the to the real terminal via script program. This allows the script program to monitor all I/O between the ksh and the real terminal. It is then a simple matter for the script program to copy all I/O to a log file. With this arrangement, the ksh is fooled into thinking it is talking to a real terminal and hence buffering is no longer a problem.

Script Command (2)



Take a closer look at the arrangement of processes, STREAMS modules and device drivers in the script program.

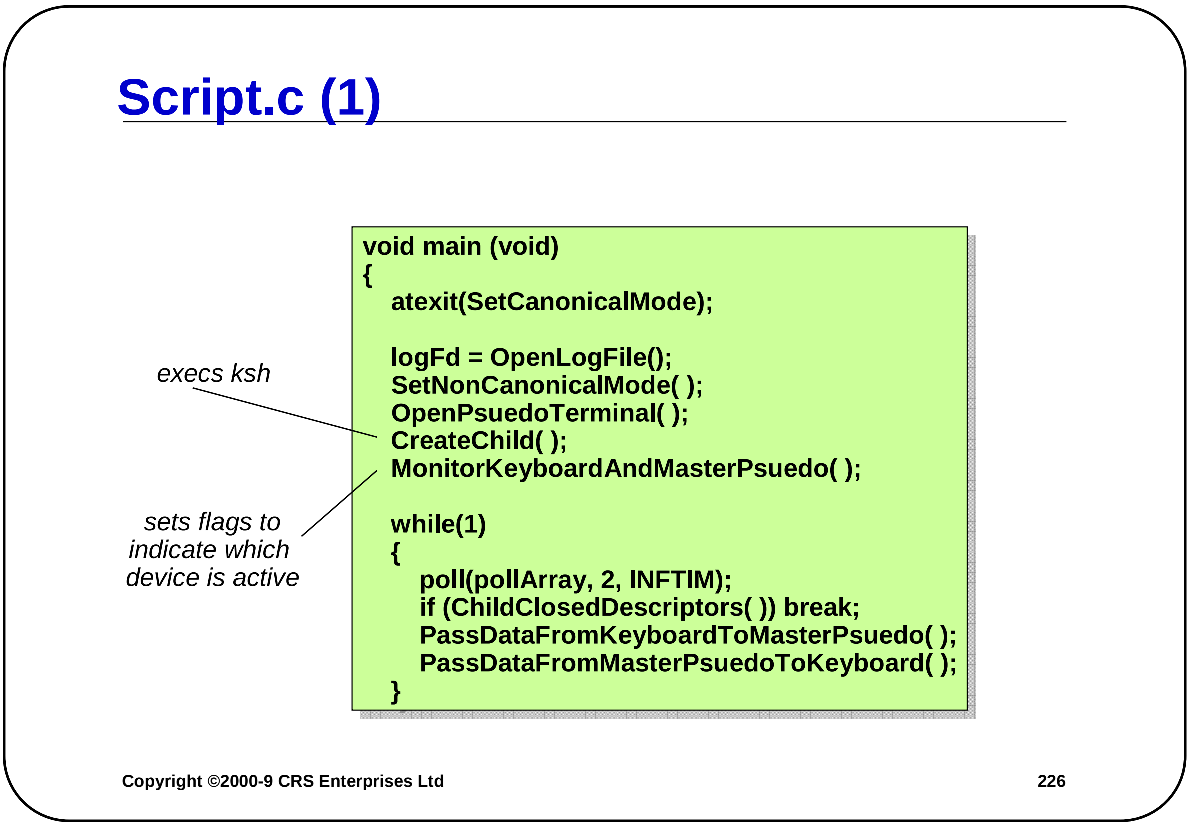
The ksh process descriptors are arranged as:

0	READ ONLY	Slave Pseudo (stdin)
1	WRITE ONLY	Slave Pseudo (stdout)
2	WRITE ONLY	Real Terminal (stderr) - not shown

The script process descriptors are arranged as:

0	READ ONLY	Slave Pseudo (stdin)
1	WRITE ONLY	Slave Pseudo (stdout)
2	WRITE ONLY	Real Terminal (stderr) - not shown
3	READ/WRITE	Master Pseudo
4	WRITE ONLY	log file

The line discipline on the pseudo terminal is set to canonical to make it easy to log output from the ksh. This means that the line discipline on the real terminal must be set to non-canonical (otherwise data would get cooked twice!).



Script.c (1)

```
void main (void)
{
   atexit(SetCanonicalMode);

   logFd = OpenLogFile();
   SetNonCanonicalMode( );
   OpenPsuedoTerminal( );
   CreateChild( );
   MonitorKeyboardAndMasterPsuedo( );

   while(1)
   {
      poll(pollArray, 2, INFTIM);
      if (ChildClosedDescriptors( )) break;
      PassDataFromKeyboardToMasterPsuedo( );
      PassDataFromMasterPsuedoToKeyboard( );
   }
```

execs ksh

sets flags to indicate which device is active

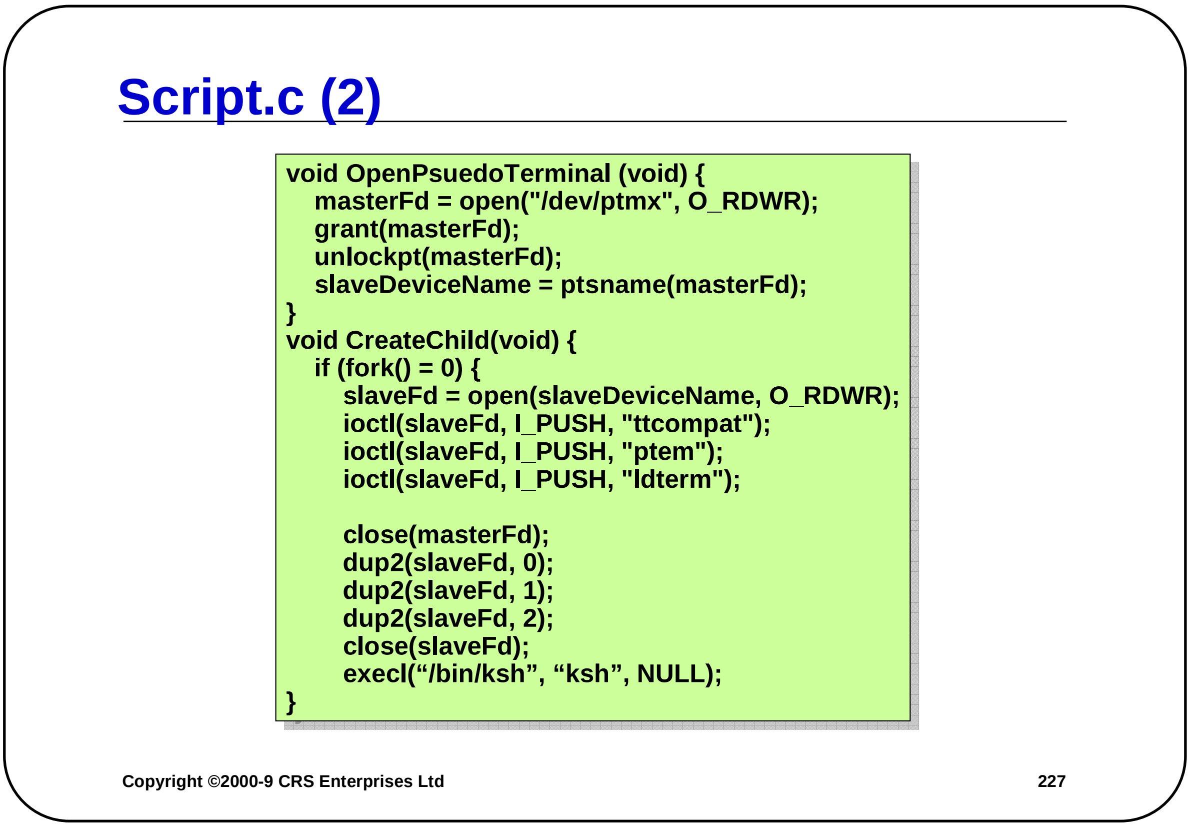
Copyright ©2000-9 CRS Enterprises Ltd 226

The script.c program can be split into several functions as shown.

The first task is to set the terminal driver to non canonical (raw) mode. The atexit() function ensures the terminal is reset when the program exits.

Next the log file and the master side of the pseudo terminal is opened. The script process then creates a child process (details shown later) with CreateChild(). The slave pseudo is opened by the child, which then execs the ksh program.

Data passing through the script program is monitored. The poll() system call is used to check the appropriate file descriptors for activity (note that INFTIM indicates infinite timeouts). The poll() call will only return when either the ksh outputs data or there is input on the real terminal. The data is logged and then copied to its intended destination. The script program loops until the ksh closes its file descriptors.



Script.c (2)

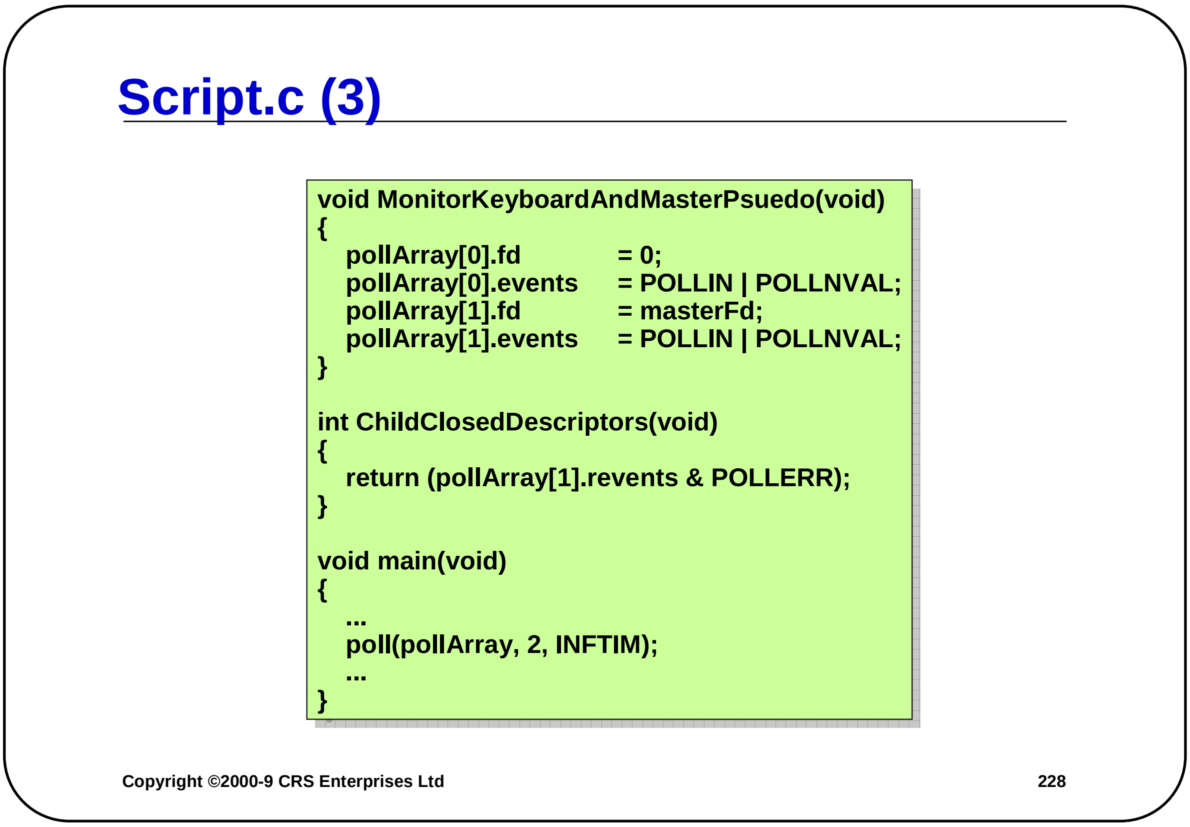
```
void OpenPsuedoTerminal (void) {
   masterFd = open("/dev/ptmx", O_RDWR);
   grant(masterFd);
   unlockpt(masterFd);
   slaveDeviceName = ptsname(masterFd);
}
void CreateChild(void) {
   if (fork() = 0) {
      slaveFd = open(slaveDeviceName, O_RDWR);
      ioctl(slaveFd, I_PUSH, "ttcompat");
      ioctl(slaveFd, I_PUSH, "ptem");
      ioctl(slaveFd, I_PUSH, "ldterm");

      close(masterFd);
      dup2(slaveFd, 0);
      dup2(slaveFd, 1);
      dup2(slaveFd, 2);
      close(slaveFd);
      execl("/bin/ksh", "ksh", NULL);
}
```

The master pseudo terminal is opened in the parent process using the clone device /dev/ptmx. Opening this device creates a master-slave pair and returns the file descriptor of the master. The ptsname call extracts the device name of the slave (e.g. /dev/pts5) so that it can be opened later by the child process.

Once the master side of the pseudo is opened, the slave side must be unlocked (unlockpt) and have read-write access (grantpt). The child process handles the slave side of the pseudo terminal. After the fork(), the child pushes 3 streams modules:

ttcompat	compatibility module for BSD and SVR4 ioctls
ptem	pseudo terminal emulation mode
ldterm	terminal line discipline

The child process must now arrange for file descriptors 0, 1 and 2 to be redirected to the slave pseudo, set canonical mode on the slave and finally exec the ksh program. The ksh program will now pass all its input and output through the slave pseudo.



Script.c (3)

```
void MonitorKeyboardAndMasterPsuedo(void)
{
    pollArray[0].fd          = 0;
    pollArray[0].events      = POLLIN | POLLNVAL;
    pollArray[1].fd          = masterFd;
    pollArray[1].events      = POLLIN | POLLNVAL;
}

int ChildClosedDescriptors(void)
{
    return (pollArray[1].revents & POLLERR);
}

void main(void)
{
    ...
    poll(pollArray, 2, INFTIM);
    ...
}
```

Once the pseudo terminal is in place and the child process is running the ksh program, the parent reads data from the terminal (file descriptor 0) and from the ksh via the master pseudo (file descriptor masterFd).

This parent can't simply use read() system calls in case they block; the parent can't be sure where the next data is coming from (terminal or ksh). Instead, the poll system call allows the parent process to monitor these two file descriptors so that it can determine which descriptor has data available.

The poll() system call requires and array to be set up indicating which file descriptors are of interest and what events are to be monitored. In this example the parent monitors both file descriptors for:

POLLIN input data present

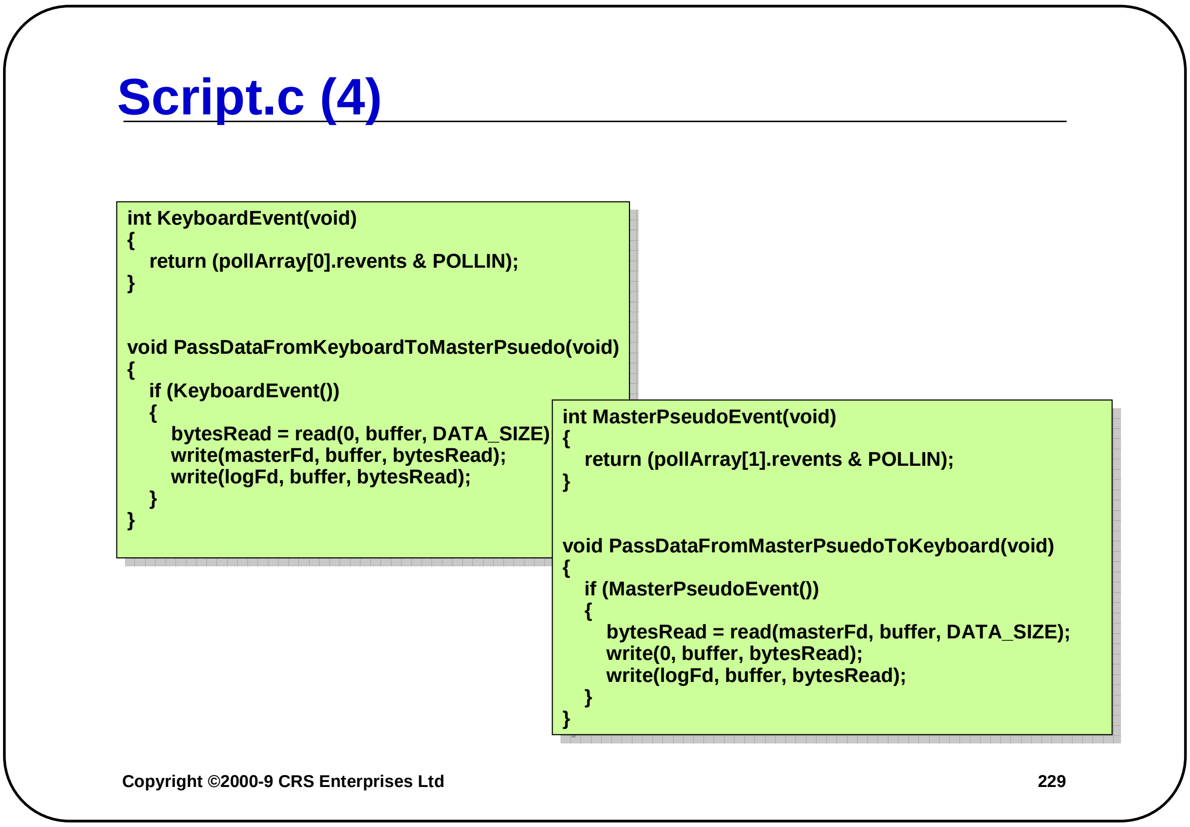
POLLNVAL file descriptor not valid

The master pseudo is also checked for a POLLERR. This state will occur when the ksh exits.

POLLERR error on file descriptor (i.e. closed)

poll() can use timeouts if desired; here we have chosen to poll forever:

INFTIM infinite timeout



Script.c (4)

```
int KeyboardEvent(void)
{
   return (pollArray[0].events & POLLIN);
}


void PassDataFromKeyboardToMasterPsuedo(void)
{
   if (KeyboardEvent())
   {
      bytesRead = read(0, buffer, DATA_SIZE)
      write(masterFd, buffer, bytesRead);
      write(logFd, buffer, bytesRead);
   }
}
```

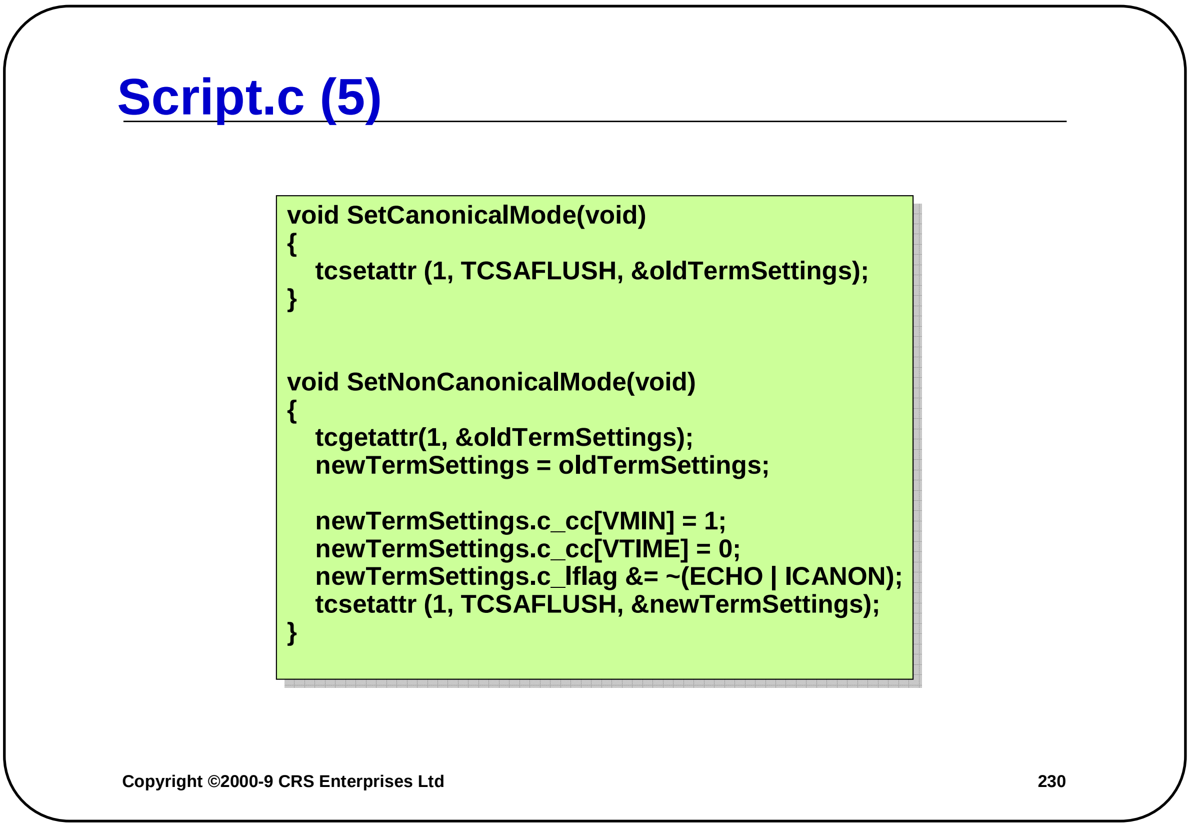
```
int MasterPseudoEvent(void)
{
   return (pollArray[1].revents & POLLIN);
}


void PassDataFromMasterPsuedoToKeyboard(void)
{
   if (MasterPseudoEvent())
   {
      bytesRead = read(masterFd, buffer, DATA_SIZE);
      write(0, buffer, bytesRead);
      write(logFd, buffer, bytesRead);
   }
}
```

229

Once the poll system call returns we can be sure one of our file descriptors has data available. To determine which one we look at the revents component of the poll array. Since we are only interested in input events, all other events are masked out using:

pollArray[ ].revents & POLLIN

If the keyboard has data, it is logged and passed to the ksh via the master pseudo. If the master pseudo has data sent from the ksh, it is logged and passed to the terminal.

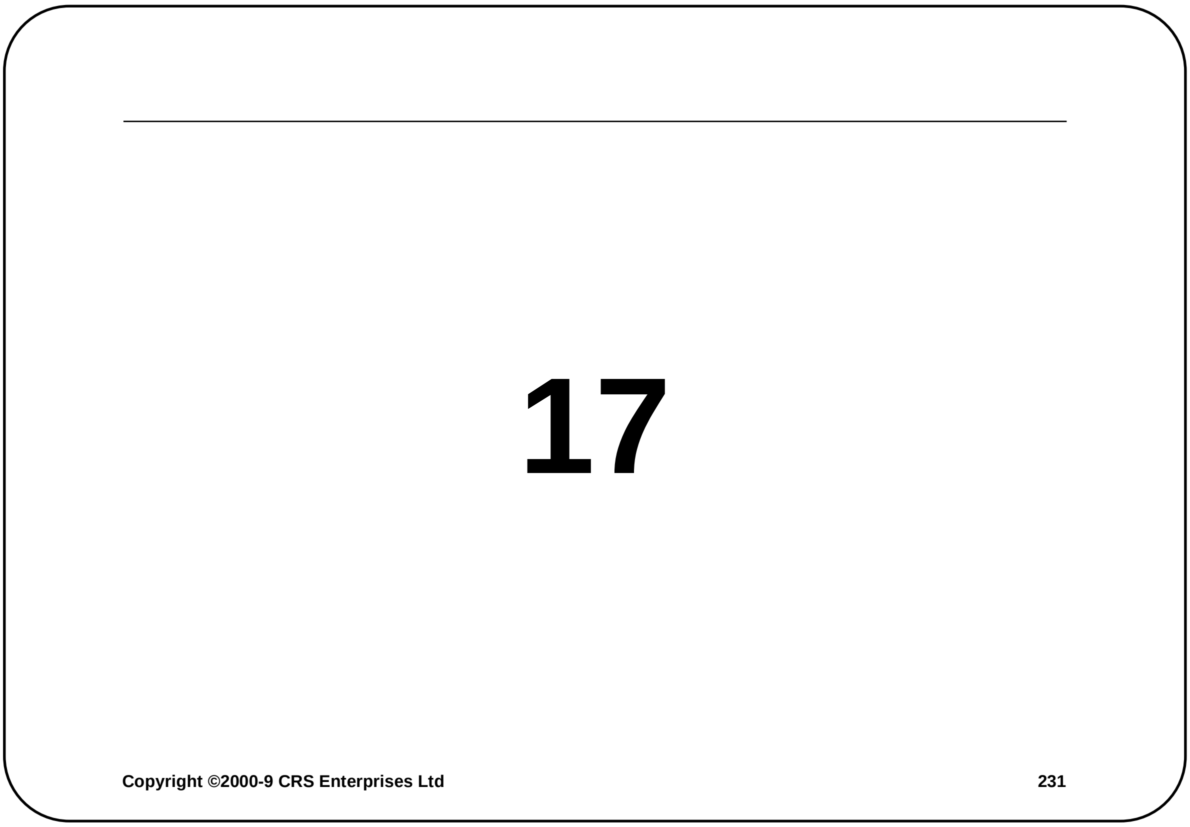


Script.c (5)

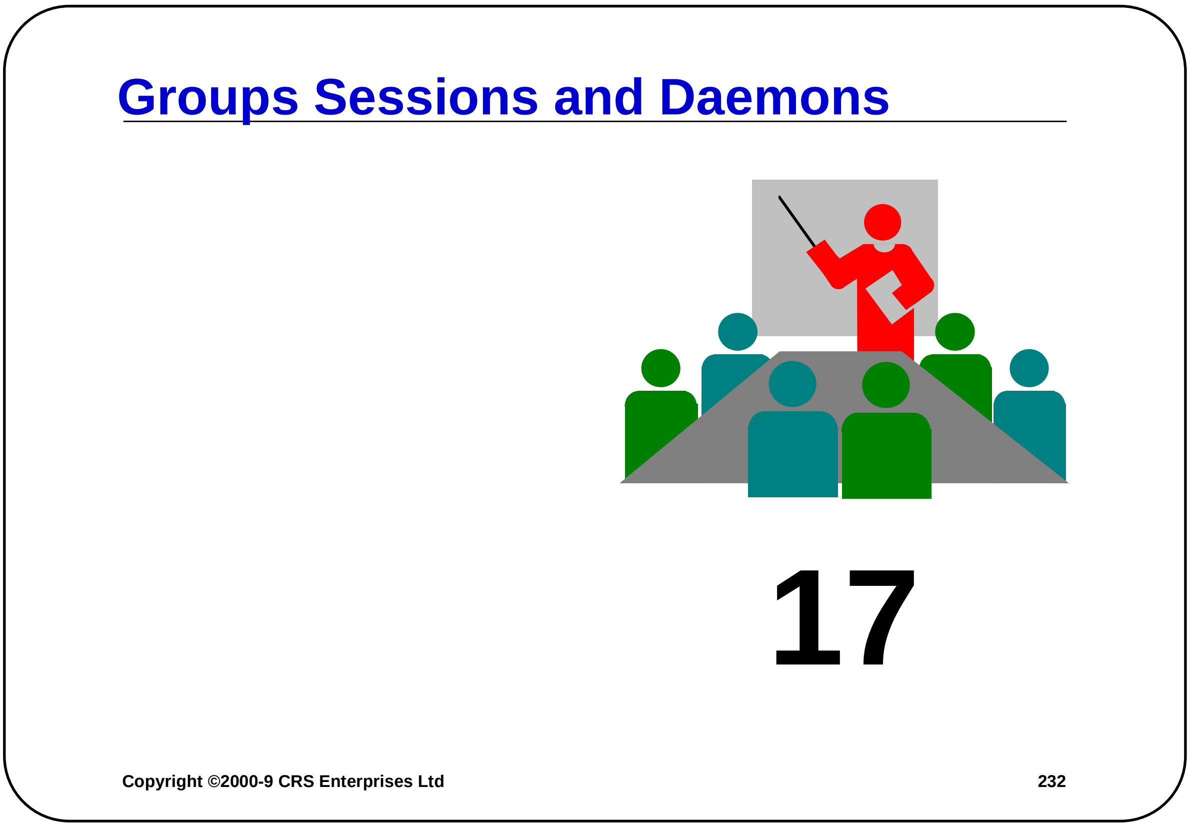
```
void SetCanonicalMode(void)
{
    tcsetattr (1, TCSAFLUSH, &oldTermSettings);
}


void SetNonCanonicalMode(void)
{
    tcgetattr(1, &oldTermSettings);
    newTermSettings = oldTermSettings;

    newTermSettings.c_cc[VMIN] = 1;
    newTermSettings.c_cc[VTIME] = 0;
    newTermSettings.c_lflag &= ~(ECHO | ICANON);
    tcsetattr (1, TCSAFLUSH, &newTermSettings);
}
```

To complete the program we must set Canonical mode on the pseudo device, but Non Canonical mode for the real terminal. This ensures the ksh behaves properly.

Note that on the terminal device we read raw data from the device one character at a time with no timeout. We switch off the loopback echoing of characters because the ksh will echo characters for us.

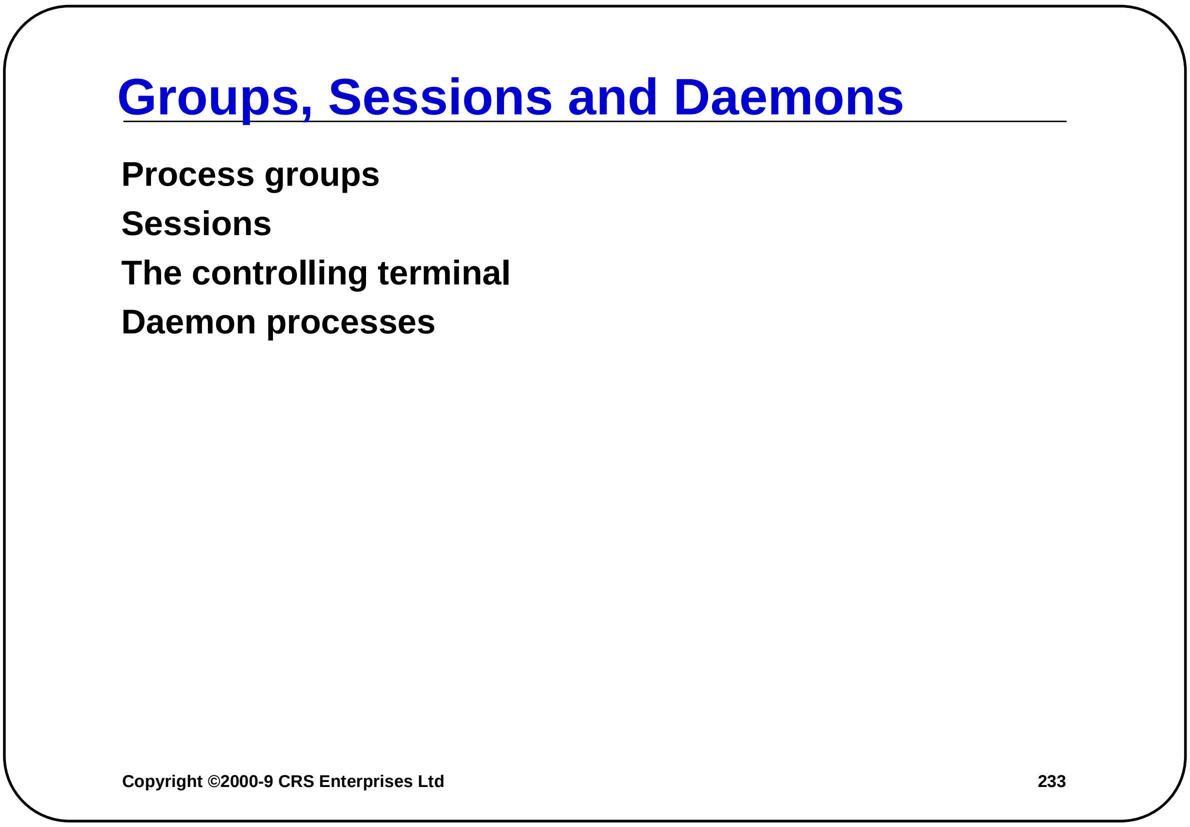


17



Groups Sessions and Daemons

17



Groups, Sessions and Daemons

Process groups

Sessions

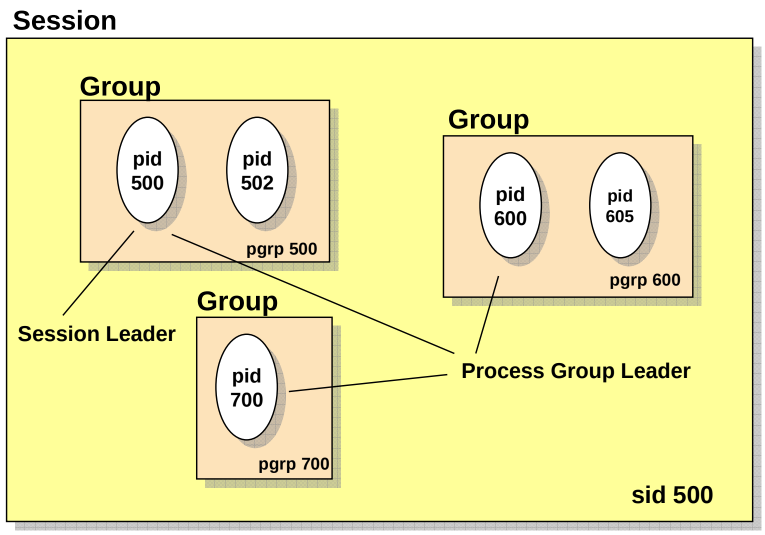
The controlling terminal

Daemon processes

We begin by considering how processes are grouped together into process groups, largely for the purposes of dealing with certain types of signal. Process groups are collected together into sessions. The interactions amongst process groups, sessions and terminals are subtle, but need to be understood in order to allow you to write effective daemon, or server, processes.

Each session has a controlling terminal. This terminal is always available to processes for I/O even if file descriptors 0, 1 and 2 have been redirected. Daemon processes are processes that have detached from their controlling terminal.

Process Groups and Sessions



A session is a collection of process groups. The easiest way to visualise a session is to think of a login session. All jobs (commands) started from your login shell will belong to the same session.

All processes belong to a process group. Process groups correspond roughly to “shell” commands. The shell defines a process group for each command you type, and all processes resulting from that command are placed into the process group. Each process group has a “leader”, denoted by the fact that the process group id is the same as its process id. When a new process is created, it inherits its parent’s process group id.

Session and Group System Calls

Process Groups

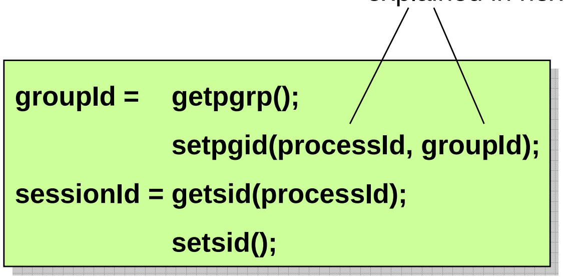
getpgid()

setpgid()

Sessions

getsid()

setsid()



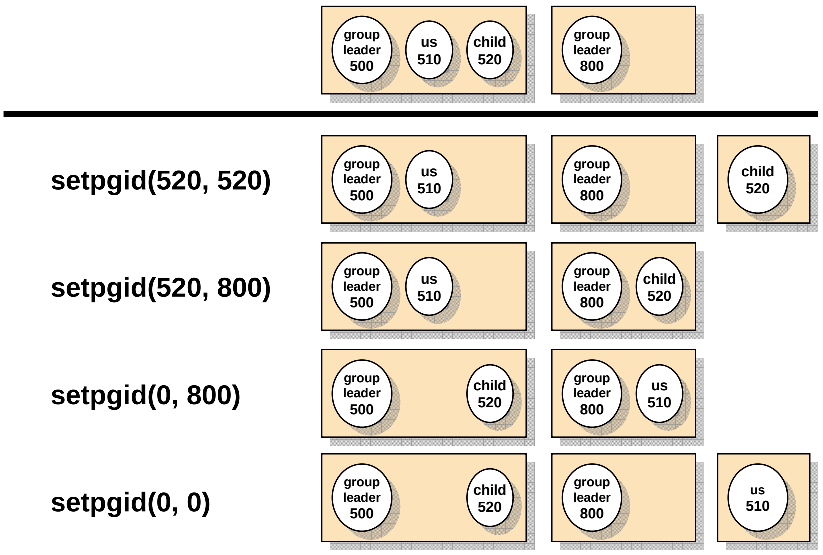
```
groupId =   getpgrp();
            setpgid(processId, groupId);
sessionId = getsid(processId);
            setsid();
```

various combinations
explained in next slide

You can find out the process group of the current process by calling the getprgrp() function. You can also change the process group using setpgid(). The parameters to setpgid() are a little tricky; examples are given overleaf. Note that you can only change the process group of the current process, or one if its children.

You can find out which session your process belongs to by calling the getsid(). A process can create a new session (it becomes the session leader) setsid(). Note that group leaders are not allowed to leave a session.

Changing Process Groups



setpgid(520, 520)

setpgid(520, 800)

setpgid(0, 800)

setpgid(0, 0)

You can change the process group using setpgid(). You can only change the process group of the current process, or one if its children as follows:

setpgid(pid, pgid)

pid	current process or one of its children
pgid	process group number

There are 4 cases to consider

pid = pgid	process pid becomes group leader
pid <> pgid	process pid joins group pgid
pid = 0	current process joins group pgid
pgid = pid = 0	current process becomes group lead

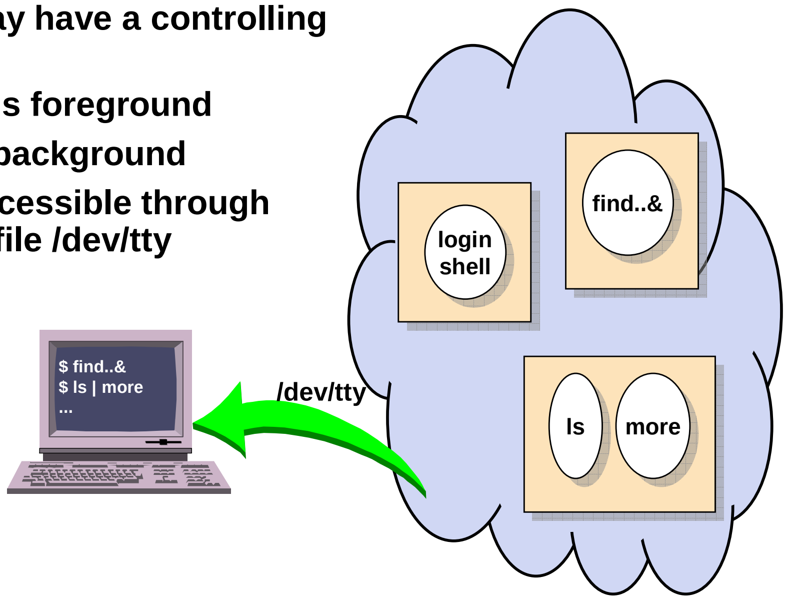
The Controlling Terminal

Session may have a controlling terminal

one group is foreground

others are background

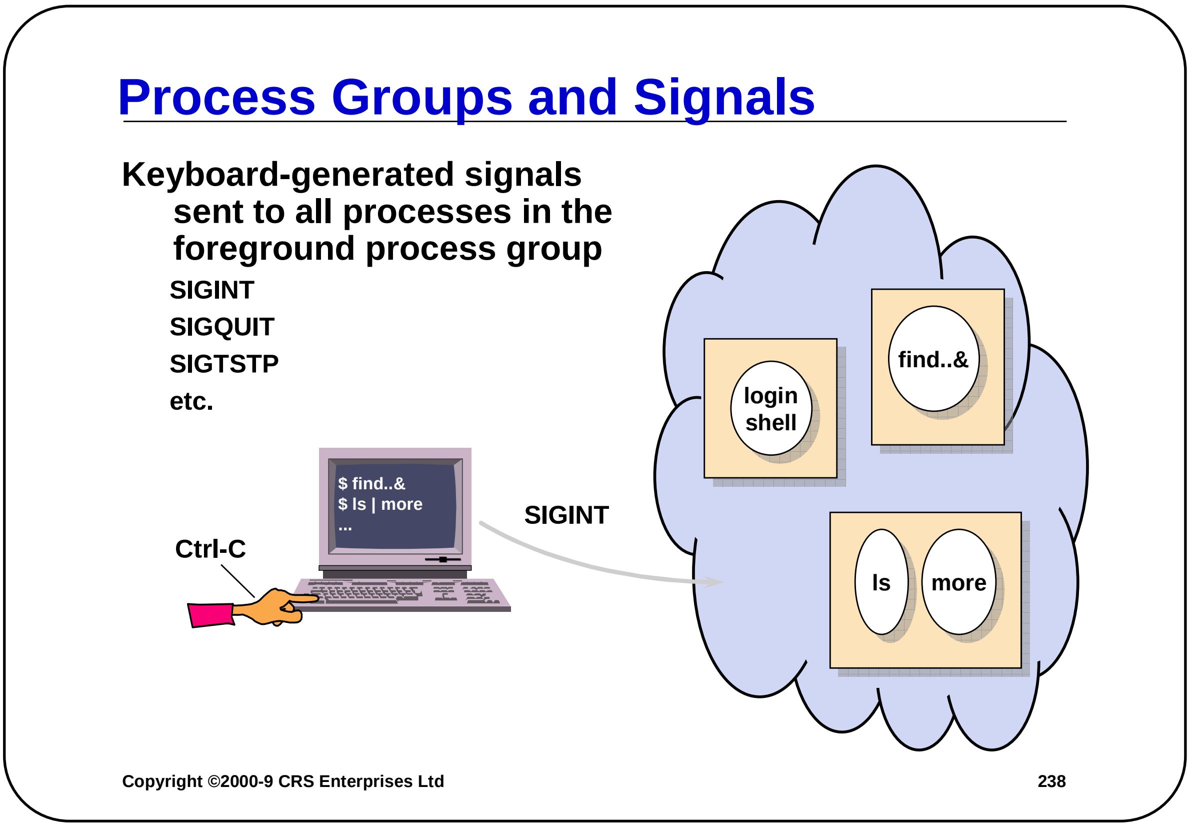
terminal accessible through special file /dev/tty



```
$ find..&
$ ls | more
...
```

Normally, a session is associated with a terminal device. This terminal is known as the controlling terminal. In such a situation, the session is said to have one foreground process group and a number of background process groups.

In all cases, the controlling terminal can be accessed from a process through the special file /dev/tty. This is a useful facility, as it allows a process to output to and read from its terminal, even when stdin, stdout and stderr have been redirected.



Process Groups and Signals

Keyboard-generated signals sent to all processes in the foreground process group

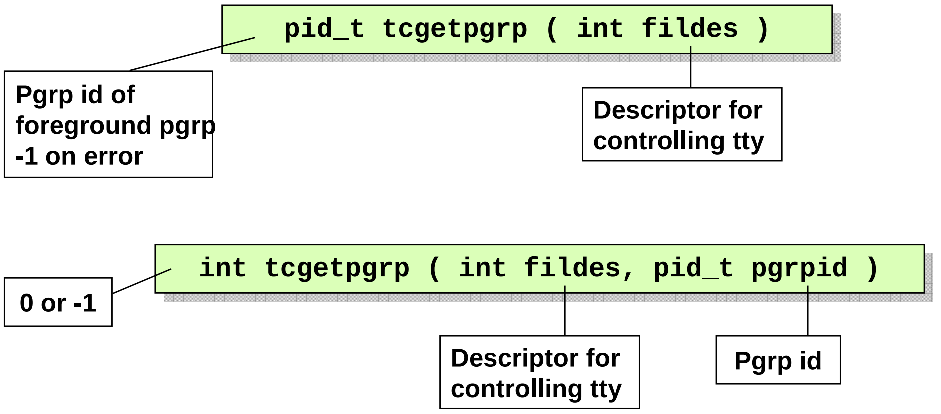
- **SIGINT**
- **SIGQUIT**
- **SIGTSTP**
- **etc.**

238

The main reason that we have process groups is to allow the distribution of signals generated from the keyboard of the controlling terminal device. For example: in the slide, when the user types ^c, which jobs do we interrupt? The signal is sent to all processes in the foreground process group.

Getting/Setting Foreground Pgrp

Usually called only by the shell



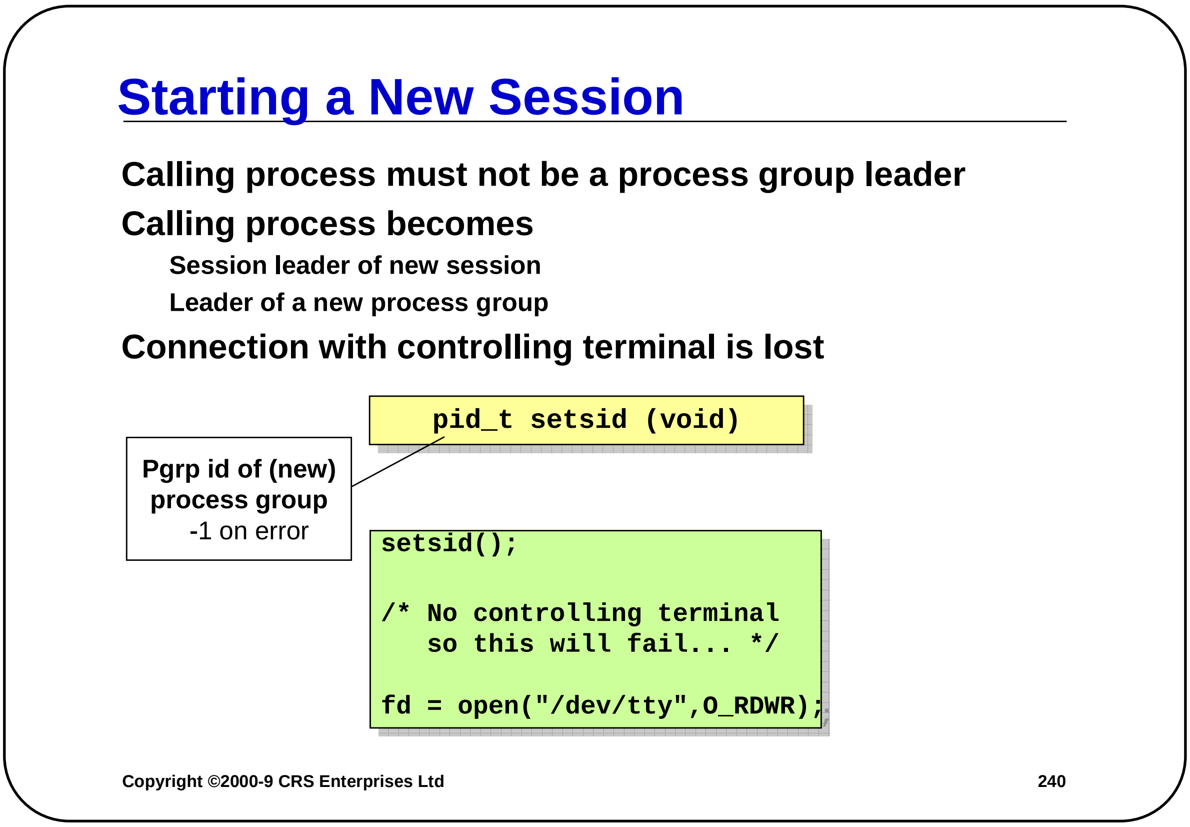
```
pid_t tcgetpgrp ( int fildes )
```

Pgrp id of foreground pgrp
-1 on error
Descriptor for controlling tty

```
int tcgetpgrp ( int fildes, pid_t pgrpid )
```

0 or -1
Descriptor for controlling tty
Pgrp id

It is possible to find out the pgrp id of the current foreground process group and also to change foreground process group using these two functions from the termios group. However, it is unusual to see these functions used anywhere other than inside a shell that provides job-control functionality.



Starting a New Session

Calling process must not be a process group leader

Calling process becomes

- Session leader of new session
- Leader of a new process group

Connection with controlling terminal is lost

```
pid_t setsid (void)
```

Pgrp id of (new) process group
-1 on error

```
setsid();

/* No controlling terminal
   so this will fail... */

fd = open("/dev/tty",O_RDWR);
```

Copyright ©2000-9 CRS Enterprises Ltd 240

It is often desirable, particularly when writing a daemon server process (see later), to move processes into a new session that is not connected with a controlling terminal. You can create a new session with the setsid() function. This causes a number of things to happen:

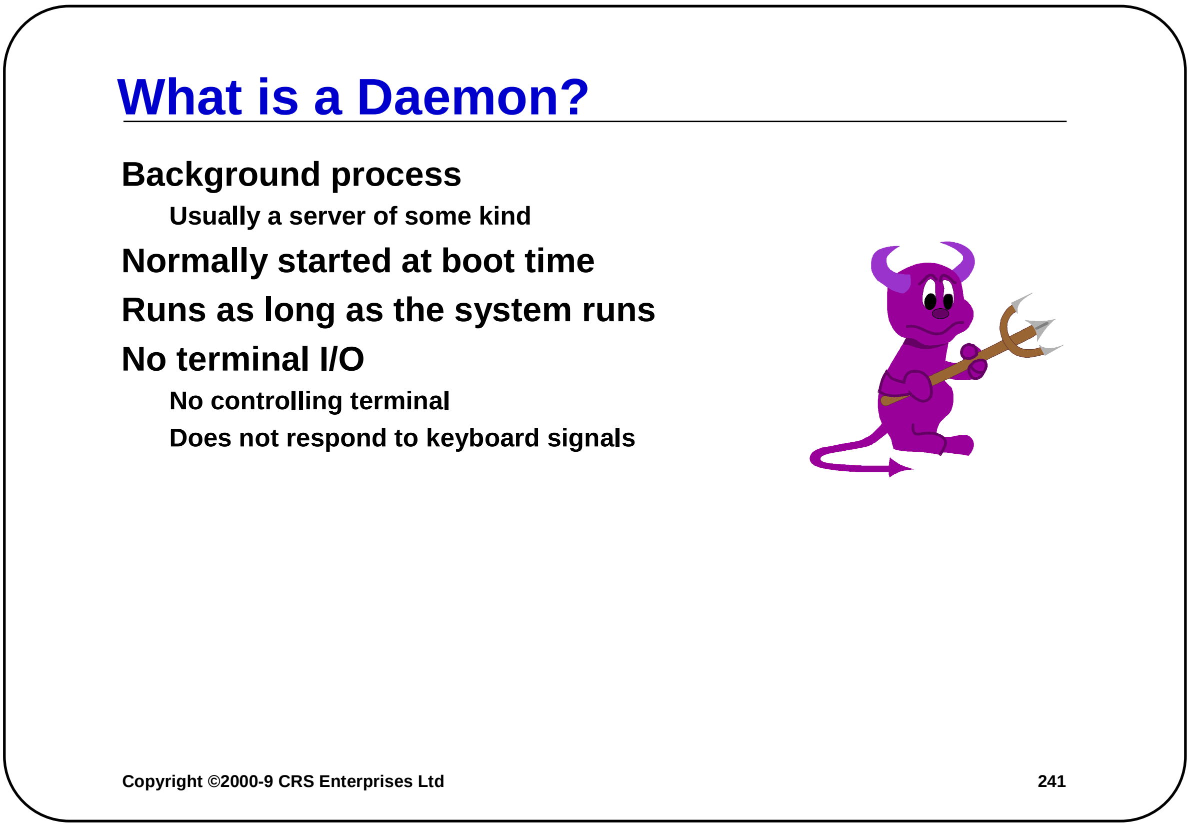
1. The calling process becomes the leader of a new session (and the only member of the new session). It also becomes the leader of a new process group.

   Note that the process must not be a process group leader already, otherwise the call will fail.

2. The connection to the controlling terminal is broken (if there was one). Attempts to open the file /dev/tty will fail after this.

   Note that if the new session leader opens a terminal (or pseudo-terminal) device (such as /dev/ttya) then that terminal becomes the controlling terminal for the new session.

The new session now operates independently of the initial session.



What is a Daemon?

- Background process
  - Usually a server of some kind
- Normally started at boot time
- Runs as long as the system runs
- No terminal I/O
  - No controlling terminal
  - Does not respond to keyboard signals

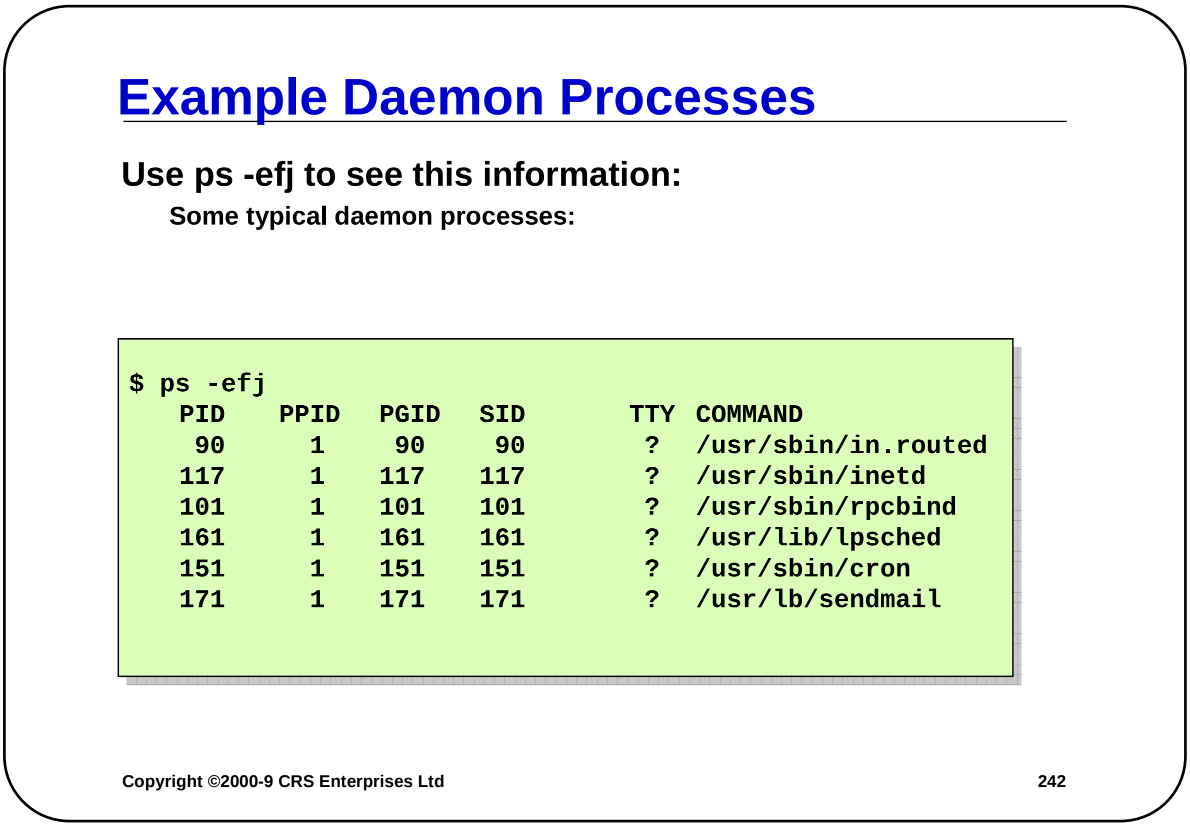
Copyright ©2000-9 CRS Enterprises Ltd 241

In Unix, a daemon is a special type of process, normally implementing a server of some kind. A daemon is similar to a background process, but it has some important differences from a normal process started like

```
$ cmd &
```

The daemon is not part of the login session. It has no controlling terminal, indeed a daemon will not use a terminal for any type of I/O. This means it cannot respond to any keyboard generated signals. Daemon processes will normally have init as their parent.

A daemon has a long lifespan. It will typically be started when the system boots, and run until the system is shut down. Its job is to sit quietly most of the time, but respond to requests from client processes when required.



Example Daemon Processes

Use ps -efj to see this information:

Some typical daemon processes:

```
$ ps -efj
   PID    PPID   PGID   SID       TTY  COMMAND
    90      1     90     90        ?   /usr/sbin/in.routed
   117      1    117    117        ?   /usr/sbin/inetd
   101      1    101    101        ?   /usr/sbin/rpcbind
   161      1    161    161        ?   /usr/lib/lpsched
   151      1    151    151        ?   /usr/sbin/cron
   171      1    171    171        ?   /usr/lb/sendmail
```

242

Normal operation of a Unix system involves many daemons. To see them, you can use the various options of the ps command. The information shown on the slide is obtained from the command:

```
$ ps -efj
```

Most of the daemons control the operation of the networking applications. For example: in the slide, inetd (pid 117) is the master control program for TCP/IP servers such as ftp and telnet; rpcbind (pid 101) is used for remote procedure calls.

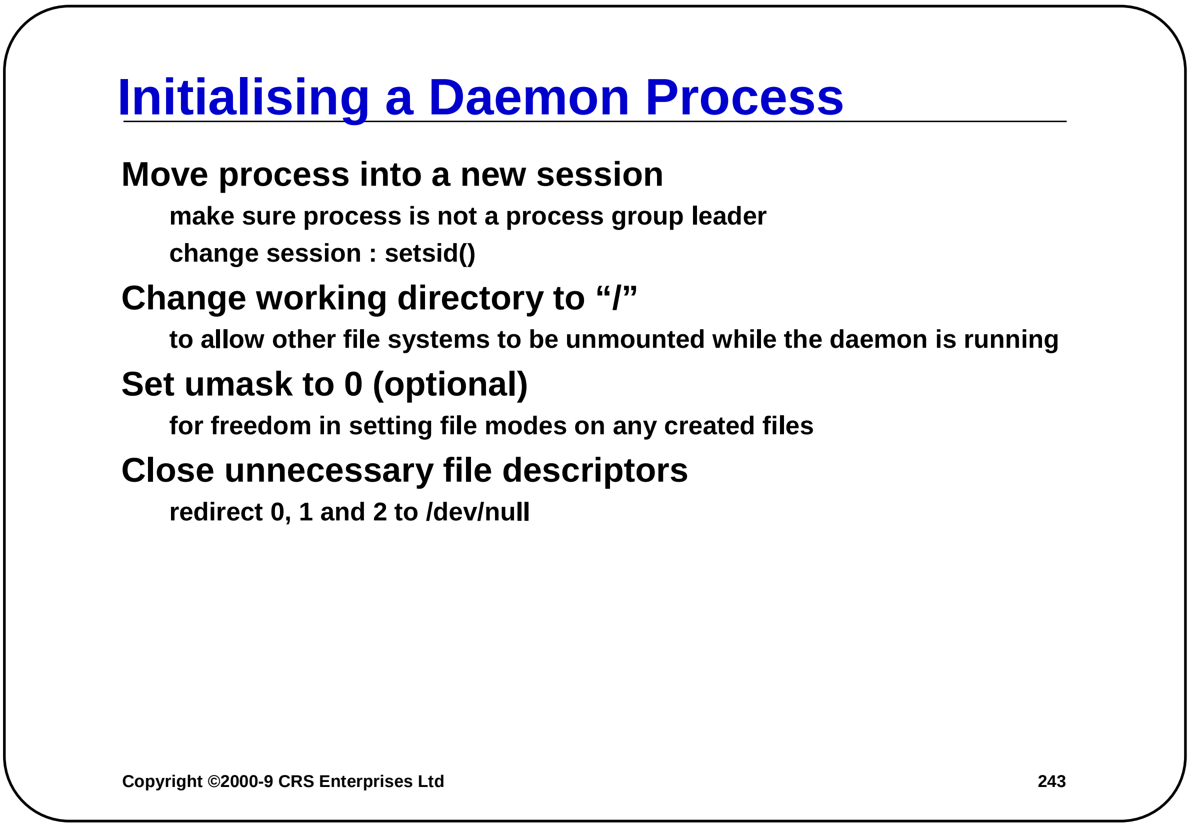
Other daemons are also shown:

lpsched is the print spooler.

cron is the clock daemon, arranging for jobs to be run at specific times. This is used by commands such as at.

sendmail is the mail delivery daemon. It delivers electronic mail to users locally, or forwards it to remote systems.

You can recognise the characteristics of daemon processes from this output. Each daemon is its own process group leader, and belongs in its own session. The '?' in the TTY column indicates that there is no controlling terminal.



Initialising a Daemon Process

Move process into a new session

- make sure process is not a process group leader
- change session : setsid()

Change working directory to “/”

- to allow other file systems to be unmounted while the daemon is running

Set umask to 0 (optional)

- for freedom in setting file modes on any created files

Close unnecessary file descriptors

- redirect 0, 1 and 2 to /dev/null

Copyright ©2000-9 CRS Enterprises Ltd 243

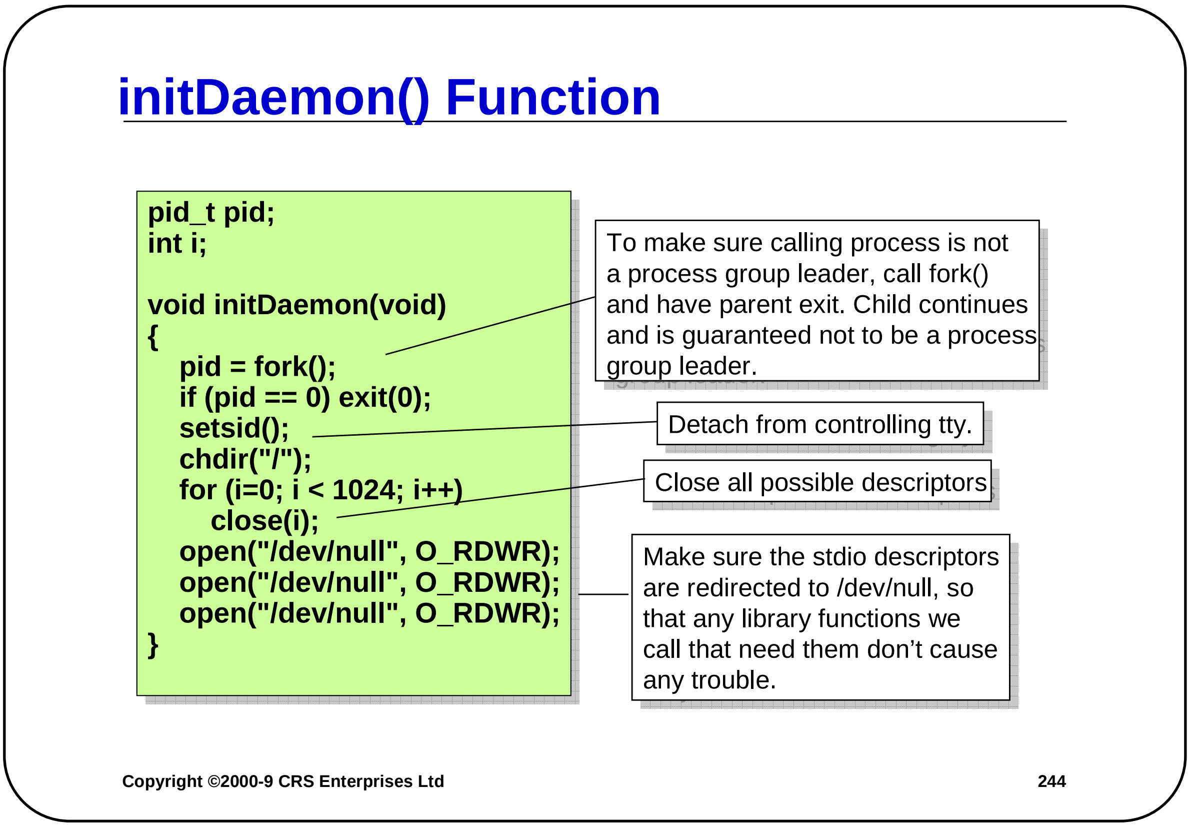
This slide illustrates the steps necessary to initialize a daemon process. The main requirements are:

1. Create a new session, with this process as the session group leader. To do this, we call setsid(). However we must ensure that the process is not a process group leader beforehand, otherwise the call to setsid() fails. Once in a new session, we must also ensure that we do not acquire a controlling terminal. To do this, we must take care if the process opens a terminal device, since the first terminal device opened (that is not already a controlling terminal) will be allocated as our controlling terminal. The safest way to do this is to make sure that the O_NOCTTY flag is passed to any open() call to a terminal.

2. Change the working directory to “/”. This means that it will be possible to unmount other file systems while the daemon is running. If we did not do this, then the daemon would be holding open the current directory from when it was started, and thus prevent the file system containing that directory from being unmounted. You can, of course, use any directory as a working directory, but the “unmounting” problem should be remembered.

3. Set the file-mode creation mask (umask) to 0. This gives the daemon maximum control over the permissions when it creates files.

4. Close all unnecessary file descriptors. This can be rather tricky (i.e. working out how many descriptors were open), but should be done nevertheless. As regards the standard I/O descriptors, it is safest to have them open but redirected to /dev/null. Some library functions that the daemon may call might use these descriptors. If we did not have them open at all, this may cause errors. Worse, we may have opened other files that have been given these descriptors and this could cause corruption of the files.



initDaemon() Function

```
pid_t pid;
int i;

void initDaemon(void)
{
   pid = fork();
   if (pid == 0) exit(0);
   setsid();
   chdir("/");
   for (i=0; i < 1024; i++)
      close(i);
   open("/dev/null", O_RDWR);
   open("/dev/null", O_RDWR);
   open("/dev/null", O_RDWR);
}
```

To make sure calling process is not a process group leader, call fork() and have parent exit. Child continues and is guaranteed not to be a process group leader.

Detach from controlling tty.

Close all possible descriptors

Make sure the stdio descriptors are redirected to /dev/null, so that any library functions we call that need them don't cause any trouble.

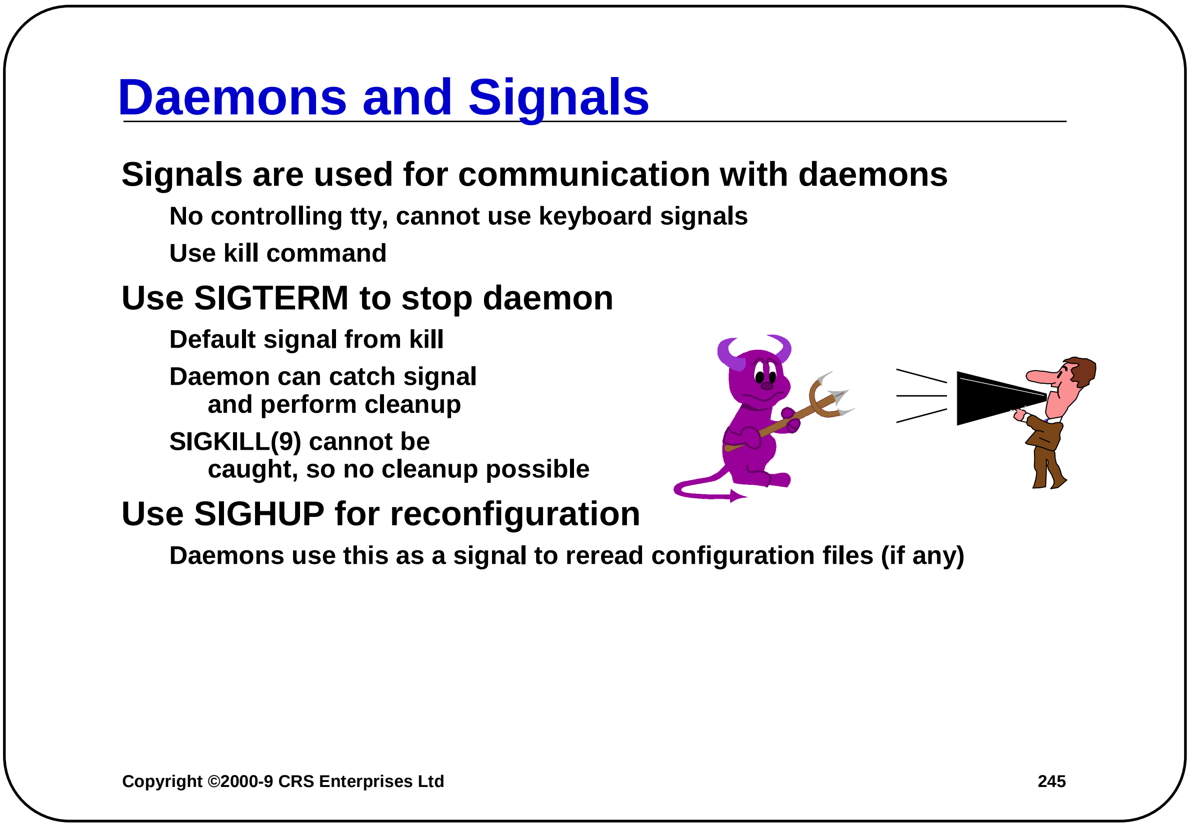
This is an example of a function that can be used to set up a daemon process. Each of the stages described on the previous slide can be identified.

First we call setsid() to create a new session for the daemon, thus losing the controlling tty. To ensure the success of setsid(), we must not be a process group leader when calling it. This is achieved by fork()ing, and having the parent process exit. By doing this, the child process (which now continues as the daemon proper) will have inherited the parent's process group id but have its own pid (i.e. it is not a process group leader).

Then we reset the umask to 0, and change directory to "/".

Now we must close all unnecessary file descriptors. In this example we assume there are 1024 file descriptors; alternatively we could use the function getrlimit() to tell us how many descriptors there are.

Finally, we must set up file descriptors 0, 1 and 2 to point to /dev/null, to safeguard the operation of any library functions we may call that have need of these descriptors.



Daemons and Signals

Signals are used for communication with daemons

- No controlling tty, cannot use keyboard signals
- Use kill command

Use SIGTERM to stop daemon

- Default signal from kill
- Daemon can catch signal and perform cleanup
- SIGKILL(9) cannot be caught, so no cleanup possible

Use SIGHUP for reconfiguration

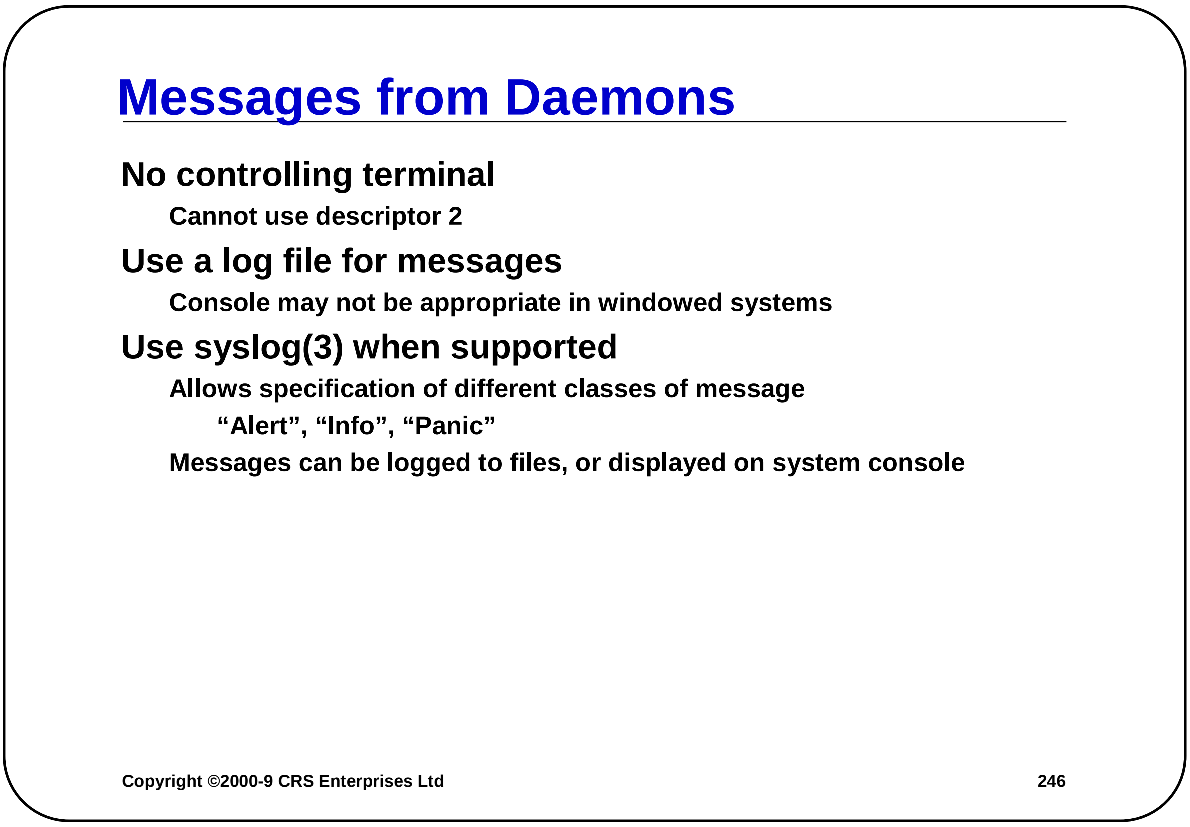
- Daemons use this as a signal to reread configuration files (if any)

Since the daemon has no controlling terminal, it is usual to use signals to communicate with it. In particular, when you want to stop the daemon, it is necessary to send it a signal using the kill command.

The normal termination signal, SIGTERM (signal no.15), is the default signal sent by kill. This signal can be caught by the daemon, and allows it to perform cleanup before terminating. You should always use this signal to stop a daemon and likewise, when writing a daemon, you should install a handler to allow cleanup to be performed before closing down.

Do not use SIGKILL (signal no. 9) unless the daemon has stopped responding to SIGTERM. SIGKILL cannot be caught and so there is no chance of any cleanup being done before the daemon exits. SIGKILL is a last resort.

If your daemon reads a configuration file for information on startup, (like inetd reads inetd.conf), then it is useful to allow reconfiguration without stopping and restarting the daemon. Most system daemons that have this property use SIGHUP (signal no. 1) as a means of requesting that they reread their configuration file. This can be a useful mechanism to include in your daemon.



Messages from Daemons

No controlling terminal

- **Cannot use descriptor 2**

Use a log file for messages

- **Console may not be appropriate in windowed systems**

Use syslog(3) when supported

- **Allows specification of different classes of message**
  - **“Alert”, “Info”, “Panic”**
- **Messages can be logged to files, or displayed on system console**

Since daemons have no controlling terminal, you cannot use stderr for displaying messages and diagnostics. Normally, messages should be logged to a file. It is not safe to rely on the system console (/dev/console) as in some windowed interfaces there is no way of monitoring such messages.

Many systems now support a facility known as syslog, a flexible and powerful mechanism for logging and displaying error messages and diagnostic messages from system processes, daemons and the kernel itself. Messages can be grouped into various types and logged to various destinations including the console, files and through electronic mail to the system administrator.

If your system has syslog, you should investigate it and use it with daemon processes.