Unit - 2

Process Environment

Outline

- File and Record Locking
- Device File APIs
- The Environment of a UNIX Process - Introduction
- main function
- Process Termination
- Command-Line Arguments
- Environment List & Environment Variables
- Memory Layout of a C Program
- Shared Libraries
- Memory Allocation
- setjmp and longjmp Functions
- UNIX Kernel Support for Processes

main Function

- A C program starts execution with a function called main.
- The prototype for the main function is

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

where

- *argc is the number of command-line arguments*
- *argv is an array of pointers to the* arguments

- When a C program is executed by the kernel by one of the exec functions, a special start-up routine is called before the main function is called.
- The executable program file specifies this routine as the starting address for the program; this is set up by the link editor when it is invoked by the C compiler.
- This start-up routine takes values from the kernel the command-line arguments and the environment and sets things up so that the main function is called

Process Termination

- There are eight ways for a process to terminate.
- Normal termination occurs in five ways:
  1. **Return from main**
  2. **Calling exit**
  3. **Calling _exit or _Exit**
  4. **Return of the last thread from its start routine**
  5. **Calling pthread_exit from the last thread**
- Abnormal termination occurs in three ways:
  6. **Calling abort**
  7. **Receipt of a signal**
  8. **Response of the last thread to a cancellation request**

Process Termination

The start-up routine is also written so that if the main function returns, the exit function is called.

If the start-up routine were coded in C (it is often coded in assembler) the call to main could look like

exit(main(argc, argv));

Exit Functions

- Three functions terminate a program normally: _exit and _Exit, which return to the kernel immediately, and exit, which performs certain cleanup processing and then returns to the kernel.

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

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

Note: The reason for the different headers is that exit and _Exit are specified by ISO C, whereas _exit is specified by POSIX.1.

Exit Functions

- The exit function performs a clean shutdown of the standard I/O library: the fclose function is called for all open streams -- this causes all buffered output data to be flushed (written to the file).
- All three exit functions expect a single integer argument, which we call the *exit status*.
- *Most UNIX* System shells provide a way to examine the exit status of a process.

If (a) any of these functions is called without an exit status, or

(b) main does a return without a return value, or

(c) the main function is not declared to return an integer,

the exit status of the process is undefined.

Exit Functions

Note: if the return type of main is an integer and main "falls off the end" (an implicit return), the exit status of the process is 0.

- This behavior is new with the 1999 version of the ISO C standard. Earlier, the exit status was undefined if the end of the main function was reached without an explicit return statement or call to the exit function

- Returning an integer value from the main function is equivalent to calling exit with the same value.

  Thus exit(0);

  is the same as

  return(0);

  from the main function.

Exit Function

Classic C program

```
#include <stdio.h>
main()
{
    printf("hello, world\n");
}
```

- When we compile and run the program, we see that the exit code is random.
- If we compile the same program on different systems, we are likely to get different exit codes, depending on the contents of the stack and register contents at the time that the main function returns:

```
$ cc hello.c ;  ./a.out
hello, world
$ echo $? print the exit status
13
```

Exit Function

Now if we enable the 1999 ISO C compiler extensions, we see that the exit code changes:

$ cc -std=c99 hello.c *enable gcc's 1999 ISO C extensions*

hello.c:4: warning: return type defaults to 'int'

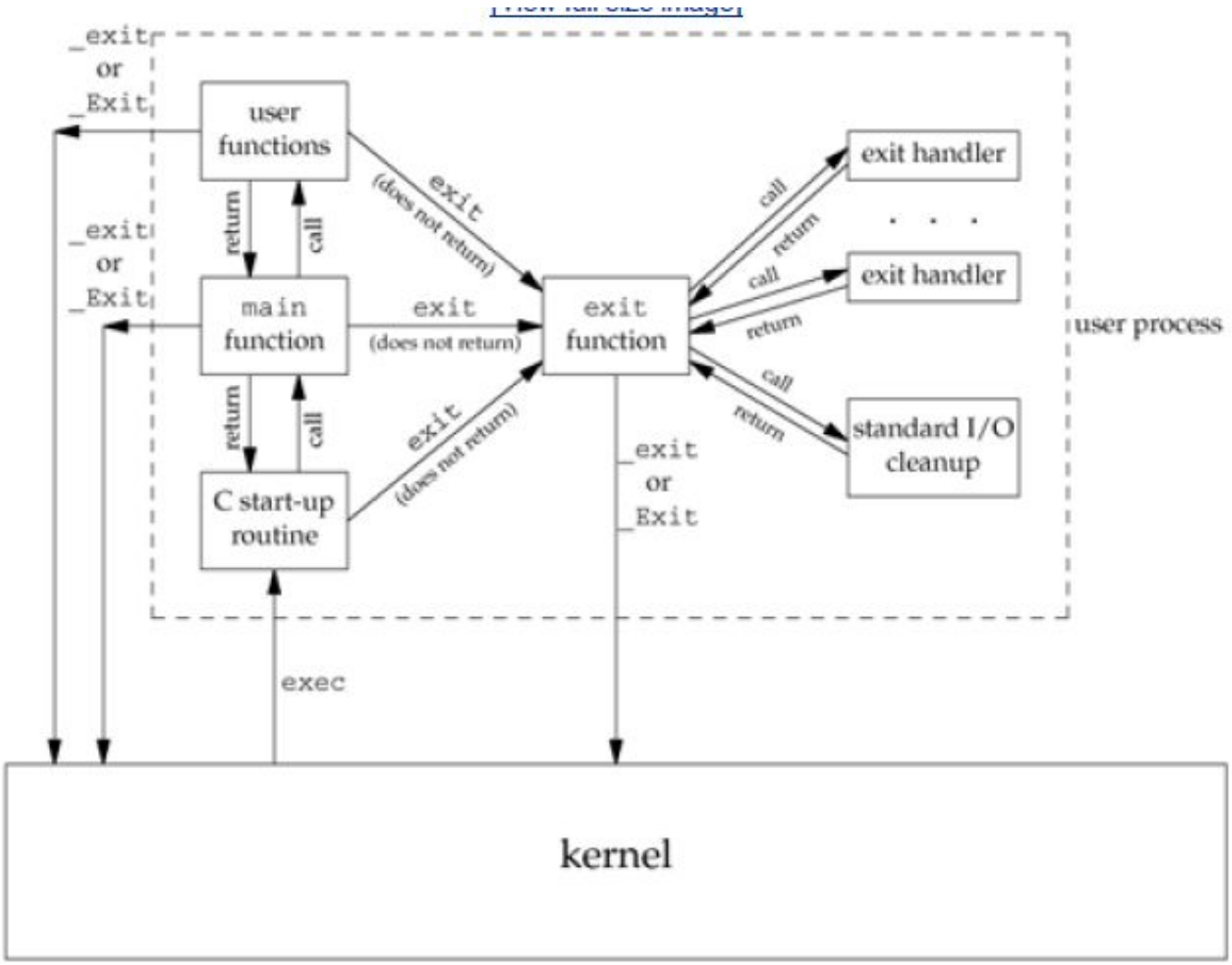
$ ./a.out

hello, world

$ echo $?

0

Lifetime of a UNIX Process



atexit Function

With ISO C, a process can register up to 32 functions that are automatically called by exit. These are called *exit handlers and are registered by calling the atexit function.*

#include <stdlib.h>

int atexit(void (**func)(void));*

Returns: 0 if OK, nonzero on error

Note:

- We have to pass the address of a function as the argument to atexit.
- When this function is called, it is not passed any arguments and is not expected to return a value.
- The exit function calls these functions in reverse order of their registration. Each function is called as many times as it was registered

Example of exit handlers

```
static void my_exit1(void);
static void my_exit2(void);

int
main(void)
{
     if (atexit(my_exit2) != 0)
          err_sys("can't register my_exit2");

     if (atexit(my_exit1) != 0)
          err_sys("can't register my_exit1");

     if (atexit(my_exit1) != 0)
          err_sys("can't register my_exit1");

     printf("main is done\n");
     return(0);
}

static void
my_exit1(void)
{
   printf("first exit handler\n");
}

static void
my_exit2(void)
{
   printf("second exit handler\n");
}
```

Command-Line Arguments

Echo all command-line arguments to standard output

```
int main(int argc, char *argv[])
{
   int i;
   for (i = 0; i < argc; i++) /* echo all command-line args */
    printf("argv[%d]: %s\n", i, argv[i]);
   exit(0);
}
```

Environment List

- Each program is also passed an *environment list.*
- *Like the argument list, the environment list is an* array of character pointers, with each pointer containing the address of a null-terminated C string.
- The address of the array of pointers is contained in the global variable environ:

```
extern char **environ;
```

By convention, the environment consists of *"name=value"* strings

Environment List

- Historically, most UNIX systems have provided a third argument to the main function that is the address of the environment list:

  int main(int *argc, char *argv[], char *envp[]);*

Access to specific environment variables is normally through the getenv and putenv functions

Environment Variables

#include <stdlib.h>

char *getenv(const char **name);*

Returns: pointer to *value associated with name, NULL if not found*

int putenv(char *str);

int setenv(const char *name, const char *value,
int rewrite);

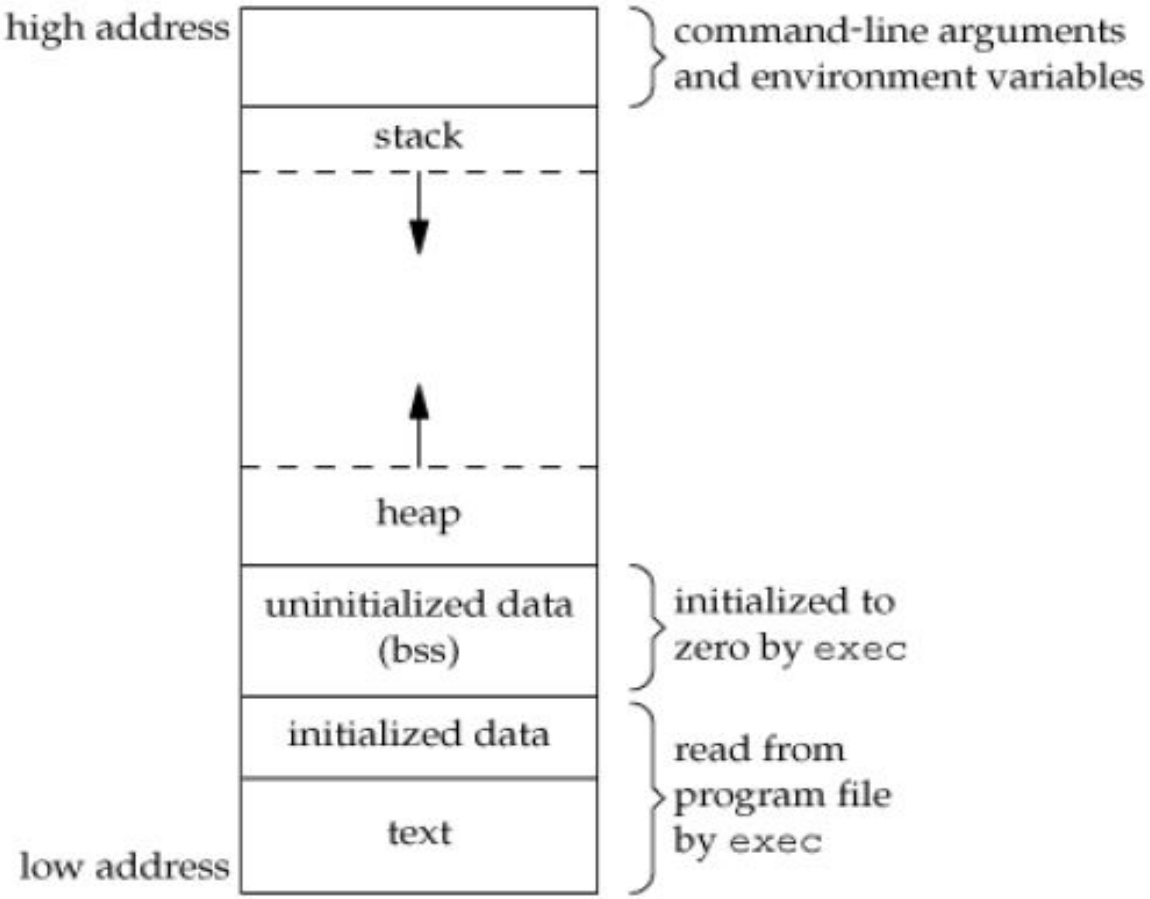
int unsetenv(const char *name);

All return: 0 if OK, nonzero on error

Environment Variables

- The putenv function takes a string of the form *name=value and places it in the environment list.*
  - If *name already exists, its old definition is first removed.*
- The setenv function sets *name to value.*
  - *If name already exists in the environment, then (a) if rewrite is nonzero, the existing definition for name is first removed; (b) if rewrite is 0, an* existing definition for *name is not removed, name is not set to the new value, and no error* occurs.
- The unsetenv function removes any definition of *name.*
  - *It is not an error if such a definition does not exist.*

Memory Layout of a C Program



Memory Layout of a C Program

- Text segment, the machine instructions that the CPU executes.
- Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on.
- Also, the text segment is often readonly, to prevent a program from accidentally modifying its instructions.

- Initialized data segment, usually called simply the data segment, containing variables that are specifically initialized in the program. For example, the C declaration
  int maxcount = 99;
  appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.

Memory Layout of a C Program

Uninitialized data segment, often called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol." Data in this segment is initialized by the kernel to arithmetic 0 or null pointers before the program starts executing.
The C declaration
long sum[1000];

appearing outside any function causes this variable to be stored in the uninitialized data segment.

Memory Layout of a C Program

- Stack, where automatic variables are stored, along with information that is saved each time a function is called.
- Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack.
- The newly called function then allocates room on the stack for its automatic and temporary variables.
- This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function.
- Heap, where dynamic memory allocation usually takes place -- located between the uninitialized data and the stack.

Memory Layout of a C Program

The `size`(1) command reports the sizes (in bytes) of the text, data, and bss segments. For example:

```
$ size /usr/bin/cc /bin/sh
   text     data   bss     dec     hex   filename
  79606     1536   916   82058   1408a   /usr/bin/cc
 619234    21120 18260  658614   a0cb6   /bin/sh
```

Shared Libraries

- Shared libraries remove the common library routines from the executable file
- They maintain a single copy of the library routine somewhere in memory that all processes reference.
- This reduces the size of each executable file but may add some runtime overhead, either when the program is first executed or the first time each shared library function is called.
- Another advantage of shared libraries is that library functions can be replaced with new versions without having to relink edit every program that uses the library.

Shared Libraries

```
$ cc -static hello1.c                    prevent gcc from using shared libraries
$ ls -l a.out
-rwxrwxr-x 1 sar         475570 Feb 18 23:17 a.out
$ size a.out
   text     data     bss     dec     hex    filename
 375657     3780    3220  382657   5d6c1    a.out
```

If we compile this program to use shared libraries, the text and data sizes of the executable file are greatly decreased:

```
$ cc hello1.c                        gcc defaults to use shared libraries
$ ls -l a.out
-rwxrwxr-x 1 sar          11410 Feb 18 23:19 a.out
$ size a.out
   text     data     bss    dec       hex    filename
    872      256       4   1132       46c    a.out
```

Memory Allocation

Three functions for memory allocation:

1. malloc - which allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.
2. calloc - which allocates space for a specified number of objects of a specified size. The space is initialized to all 0 bits.
3. realloc - which increases or decreases the size of a previously allocated area.

When the size increases, it may involve moving the previously allocated area somewhere else, to provide the additional room at the end.

When the size increases, the initial value of the space between the old contents and the end of the new area is indeterminate.

Memory Allocation

```
#include <stdlib.h>

void *malloc(size_t size);

void *calloc(size_t nobj, size_t size);

void *realloc(void *ptr, size_t newsize);
```

All three return: non-null pointer if OK, NULL on error

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

The function free causes the space pointed to by *ptr to be deallocated. This freed space is usually put* into a pool of available memory and can be allocated in a later call to one of the three alloc functions.

setjmp and longjmp Functions

- In C, we can't goto a label that's in another function.
- Instead, we must use the setjmp and longjmp functions to perform this type of branching.
- Application:
  - These two functions are useful for handling error conditions that occur in a deeply nested function call.

Device special files

- Every file system is known by its major and minor device numbers, which are encoded in the primitive system data type dev_t.
- The major number identifies the device driver and sometimes encodes which peripheral board to communicate with
- The minor number identifies the specific subdevice.
- Example, a disk drive often contains several file systems. Each file system on the same disk drive would usually have the same major number, but a different minor number.

Device special files

- We can usually access the major and minor device numbers through two macros defined by most implementations: major and minor.
- They can be found in <sys/types.h> on BSD-based systems.
- Solaris defines them in <sys/mkdev.h>.
- Linux defines these macros in <sys/sysmacros.h>, which is included by <sys/types.h>.
- Need not bother how the two numbers are stored in a dev_t object.

Device special files

- The st_dev value for every filename on a system is the device number of the file system containing that filename and its corresponding i-node.
- Only character special files and block special files have an st_rdev value.
- This value contains the device number for the actual device.

Print st_dev and st_rdev values

```
int main(int argc, char *argv[])
{
    int i;
    struct stat buf;
    for (i = 1; i < argc; i++) {
      printf("%s: ", argv[i]);
      if (stat(argv[i], &buf) < 0) {
            err_ret("stat error");
            continue;
      }
```

Print st_dev and st_rdev values

```
printf("dev = %d/%d", major(buf.st_dev),
    minor(buf.st_dev));
if (S_ISCHR(buf.st_mode) || S_ISBLK(buf.st_mode)) {
    printf(" (%s) rdev = %d/%d",
    (S_ISCHR(buf.st_mode)) ? "character" : "block",
     major(buf.st_rdev), minor(buf.st_rdev));
    }
printf("\n");
}
exit(0);
}
```

Print st_dev and st_rdev values

```
$ ./a.out / /home/sar /dev/tty[01]
/: dev = 3/3
/home/sar: dev = 3/4
/dev/tty0: dev = 0/7 (character) rdev = 4/0
/dev/tty1: dev = 0/7 (character) rdev = 4/1
$ mount                    which directories are mounted on which devices?
/dev/hda3 on / type ext2 (rw,noatime)
/dev/hda4 on /home type ext2 (rw,noatime)
$ ls -lL /dev/tty[01] /dev/hda[34]
brw-------  1 root      3,   3 Dec 31  1969 /dev/hda3
brw-------  1 root      3,   4 Dec 31  1969 /dev/hda4
crw-------  1 root      4,   0 Dec 31  1969 /dev/tty0
crw-------  1 root      4,   1 Jan 18 15:36 /dev/tty1
```

Print st_dev and st_rdev values

- The first two arguments to the program are directories (/ and /home/sar), and the next two are the device names /dev/tty[01].
- Note: We use the shell's regular expression language to shorten the amount of typing we need to do. The shell will expand the string /dev/tty[01] to /dev/tty0 /dev/tty1.
- We expect the devices to be character special files.
- The output from the program shows that the root directory has a different device number than does the /home/sar directory.
- This indicates that they are on different file systems.
- Running the mount(1) command verifies this.

Print st_dev and st_rdev values

- We then use ls to look at the two disk devices reported by mount and the two terminal devices.
- The two disk devices are block special files, and the two terminal devices are character special files.
- Normally, the only types of devices that are block special files are those that can contain random access file systems: disk drives, floppy disk drives, and CD-ROMs, for example.
- Some older versions of the UNIX System supported magnetic tapes for file systems, but this was never widely used.
- Note that the filenames and i-nodes for the two terminal devices (st_dev) are on device 0/7, which is the devfs pseudo file system, which implements the /dev but that their actual device numbers are 4/0 and 4/1

Record Locking

- What happens when two or more processes edit the same file at the same time?
- In most UNIX systems, the final state of the file corresponds to the last process that wrote the file. In some applications, however, such as a database system, a process needs to be certain that it alone is writing to a file.
- To provide this capability for processes that need it, commercial UNIX systems provide record locking

Record Locking

- *Record locking is the term normally used to describe the ability of a process to prevent other* processes from modifying a region of a file while the first process is reading or modifying that portion of the file.
- Under the UNIX System, the adjective "record" is a misnomer, since the UNIX kernel does not have a notion of records in a file. A better term is *byte-range locking, since it is a range of a file* (possibly the entire file) that is locked.

Record Locking

- Early UNIX systems couldn't be used to run database systems, because there was no support for locking portions of files.
- As UNIX systems found their way into business computing environments, various groups added support record locking (differently, of course).
- Early Berkeley releases supported only the flock function. This function locks only entire files, not regions of a file.
- Record locking was added to System V Release 3 through the fcntl function.
- The lockf function was built on top of this, providing a simplified interface.
- These functions allowed callers to lock arbitrary byte ranges in a file, from the entire file down to a single byte within the file.

fcntl Record Locking

#include <fcntl.h>

int fcntl(int *filedes, int cmd, ... /*struct* flock **flockptr* */);*

Returns: depends on *cmd if OK (see following), 1 on error*

fcntl Record Locking

For record locking, *cmd is F_GETLK, F_SETLK, or F_SETLKW.*

The third argument (which we'll call flockptr) is a pointer to an flock structure.

```
struct flock {
    short l_type; /* F_RDLCK, F_WRLCK, or F_UNLCK */
    off_t l_start; /* offset in bytes, relative to l_whence */
    short l_whence; /* SEEK_SET, SEEK_CUR, or SEEK_END */
    off_t l_len; /* length, in bytes; 0 means lock to EOF */
    pid_t l_pid; /* returned with F_GETLK */
};
```

fcntl Record Locking

- This structure describes - The type of lock desired: F_RDLCK (a shared read lock), F_WRLCK (an exclusive write lock), or F_UNLCK (unlocking a region)
- The starting byte offset of the region being locked or unlocked (l_start and l_whence)
- The size of the region in bytes (l_len)
- The ID (l_pid) of the process holding the lock that can block the current process (returned by F_GETLK only)

fcntl Record Locking

Rules about the specification of the region to be locked or unlocked

- The two elements that specify the starting offset of the region are similar to the last two arguments of the lseek function. Indeed, the l_whence member is specified as SEEK_SET, SEEK_CUR, or SEEK_END.
- Locks can start and extend beyond the current end of file, but cannot start or extend before the beginning of the file.
- If l_len is 0, it means that the lock extends to the largest possible offset of the file. This allows us to lock a region starting anywhere in the file, up through and including any data that is appended to the file
- To lock the entire file, we set l_start and l_whence to point to the beginning of the file and specify a length (l_len) of 0. (There are several ways to specify the beginning of the file, but most applications specify l_start as 0 and l_whence as SEEK_SET.)

fcntl Record Locking

Note: Two types of locks exist: a shared read lock (l_type of F_RDLCK) and an exclusive write lock (F_WRLCK).

- The basic rule is that any number of processes can have a shared read lock on a given byte, but only one process can have an exclusive write lock on a given byte.
- if there are one or more read locks on a byte, there can't be any write locks on that byte;
- if there is an exclusive write lock on a byte, there can't be any read locks on that byte

fcntl Record Locking

Region currently has	Request for read lock	Request for write lock
no locks	OK	OK
one or more read locks	OK	denied
one write lock	denied	denied

The compatibility rule applies to lock requests made from different processes, not to multiple lock requests made by a single process. If a process has an existing lock on a range of a file, a subsequent attempt to place a lock on the same range by the same process will replace the existing lock with the new one.

Thus, if a process has a write lock on bytes 1632 of a file and then tries to place a read lock on bytes 1632, the request will succeed (assuming that we're not racing with any other processes trying to lock the same portion of the file), and the write lock will be replaced by a read lock.

fcntl Record Locking

- To obtain a read lock, the descriptor must be open for reading; to obtain a write lock, the descriptor must be open for writing.

Commands:

- F_GETLK - Determine whether the lock described by *flockptr is blocked by some other* lock.
- If a lock exists that would prevent ours from being created, the information on that existing lock overwrites the information pointed to by *flockptr.*
- *If no lock exists that would prevent ours from being created, the* structure pointed to by *flockptr is left unchanged except for the l_type* member, which is set to F_UNLCK.

fcntl Record Locking

- F_SETLK - Set the lock described by *flockptr.*
- *If we are trying to obtain a read lock (l_type* of F_RDLCK) or a write lock (l_type of F_WRLCK) and the compatibility rule prevents the system from giving us the lock, fcntl returns immediately with errno set to either EACCES or EAGAIN
- This command is also used to clear the lock described by *flockptr (l_type of* F_UNLCK).

fcntl Record Locking

- F_SETLKW - This command is a blocking version of F_SETLK. (The W in the command name means *wait.) If the requested read lock or write lock cannot be granted* because another process currently has some part of the requested region locked, the calling process is put to sleep. The process wakes up either when the lock becomes available or when interrupted by a signal.

fcntl Record Locking

Consider the last 100 bytes as a region. Write a C/C++ program to check whether the region is locked or not. If the region is locked, print pid of the process which has locked. Ifthe region is not locked, lock the region with an exclusive lock, read the last 50 bytes and unlock the region.

fcntl Record Locking

```
#include <stdio.h>
 #include <stdlib.h>
 #include <unistd.h>
 #include <fcntl.h>
 #include <errno.h>
 int main(int argc,char *argv[])
 {
        int fd;
        char buffer[255];
        struct flock fvar;
        if(argc==1)
        {
                printf("usage: %s filename\n",argv[0]);
                return -1;
        }
```

fcntl Record Locking

```
if((fd=open(argv[1],O_RDWR))==-1)
        {
                perror("open");
                exit(1);
        }
        fvar.l_type=F_WRLCK;
        fvar.l_whence=SEEK_END;
        fvar.l_start=SEEK_END-100;
        fvar.l_len=100;
        printf("press enter to set lock\n");
        getchar();
```

fcntl Record Locking

```
printf("trying to get lock..\n");
        if((fcntl(fd,F_SETLK,&fvar))==-1)
        {     fcntl(fd,F_GETLK,&fvar);
              printf("\nFile already locked by process (pid):
   \t%d\n",fvar.l_pid);
              return -1;
        }
        printf("locked\n");
if((lseek(fd,SEEK_END-50,SEEK_END))==-1)
        {
              perror("lseek");
              exit(1);
        }
```

fcntl Record Locking

```
if((read(fd,buffer,100))==-1)
        {
                perror("read");
                exit(1);
        }
        printf("data read from file..\n");
        puts(buffer);
        printf("press enter to release
    lock\n");
        getchar();
```

```
fvar.l_type = F_UNLCK;
 fvar.l_whence = SEEK_SET;
 fvar.l_start = 0;
 fvar.l_len = 0;

if((fcntl(fd,F_UNLCK,&fvar))=
=-1)
  {
        perror("fcntl");
        exit(0);
  }
  printf("Unlocked\n");
  close(fd);
  return 0;
 }
```