Module 3

UNIX File APIs: General File APIs, File and Record Locking, Directory File APIs, Device File APIs, FIFO File APIs, Symbolic Link File APIs.

UNIX Processes and Process Control:

The Environment of a UNIX Process: Introduction, main function, Process Termination, Command-Line Arguments, Environment List, Memory Layout of a C Program, Shared Libraries, Memory Allocation, Environment Variables, setjmp and longjmp Functions, getrlimit, setrlimit Functions, UNIX Kernel Support for Processes.

Process Control: Introduction, Process Identifiers, fork, vfork, exit, wait, waitpid, wait3, wait4 Functions, Race Conditions, exec Functions

UNIX FILE APIs

- What is API?
- An application programming interface (**API**) is a computing interface which defines interactions between multiple software intermediaries. It defines the kinds of calls or requests that can be made, how to make them, the data formats that should be used, the conventions to follow, etc.

UNIX FILE APIs

- **General file API's**
- Files in a UNIX and POSIX system may be any one of the following types:

1. Regular file
2. Directory File
3. FIFO file
4. Block device file
5. character device file
6. Symbolic link file.

- There are special API's to create these types of files. There is a set of Generic API's that can be used to manipulate and create more than one type of files. These API's are:
- open(),creat(),read(),write(),close(),fcntl()

Open()

- This is used to establish a connection between a process and a file i.e. it is used to **open an existing file for data transfer function or else it may be also be used to create a new file.**
- The returned value of the open system call is the **file descriptor (row number of the file table), which contains the inode information**.
- The prototype of open function is

```
#include<sys/types.h>
#include<sys/fcntl.h>
int open(const char *pathname, int accessmode, mode_t permission);
```

open()…

- If **successful**, open returns a nonnegative integer representing the **open file descriptor**.
- If **unsuccessful**, open returns –1.
- The **first argument** is the name of the file to be created or opened. This may be an **absolute pathname or relative pathname.**
- If the given pathname is **symbolic link**, the open function will resolve the symbolic link reference to a non symbolic link file to which it refers.
- The **second argument is access modes**, which is an integer value that specifies how actually the file should be accessed by the calling process.

open()….

- Generally the access modes are specified in <fcntl.h>. Various access modes are:

O_RDONLY	- open for reading file only
O_WRONLY	- open for writing file only
O_RDWR	- opens for reading and writing file.

There are **other access modes, which are termed as access modifier flags**,

O_APPEND	- Append data to the end of file.
O_CREAT	- Create the file if it doesn't exist
O_EXCL	- Generate an error if O_CREAT is also specified and the file already exists.
O_TRUNC	- If file exists discard the file content and set the file size to zero bytes.
O_NONBLOCK	- Specify subsequent read or write on the file should be non-blocking.
O_NOCTTY	- Specify not to use terminal device file as the calling process control terminal.

open()….

- To illustrate the use of the above flags, the following example statement opens a file called /usr/usp for read and write in append mode:

int fd=open(“/usr/usp”,O_RDWR | O_APPEND,0);

- If the file is opened in **read only, then no other modifier flags can be used.**
- If a file is opened in write only or read write, then we are allowed to use any modifier flags along with them.
- The third argument is used only when a new file is being created. The symbolic names for file permission are given in the table in the previous page.

open()…

symbol	meaning
S_IRUSR	read by owner
S_IWUSR	write by owner
S_IXUSR	execute by owner
S_IRWXU	read, write, execute by owner
S_IRGRP	read by group
S_IWGRP	write by group
S_IXGRP	execute by group
S_IRWXG	read, write, execute by group
S_IROTH	read by others
S_IWOTH	write by others
S_IXOTH	execute by others
S_IRWXO	read, write, execute by others

creat(creat)

- This system call is used to create new regular files. The prototype of creat is

```
#include <sys/types.h>
#include<unistd.h>
int creat(const char *pathname, mode_t mode);
```

- Returns: file descriptor opened for write-only if OK, -1 on error.
- The first argument pathname specifies name of the file to be created.
- The second argument mode_t, specifies permission of a file to be accessed by owner group and others.
- The creat function can be implemented using open function as:

creat(path_name, mode)
open (pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);

read()

- The read function **fetches a fixed size of block of data** from a file referenced by a given file descriptor. The prototype of read function is:

```
#include<sys/types.h>
#include<unistd.h>
size_t read(int fdesc, void *buf, size_t nbyte);
```

If **successful**, read **returns the number of bytes actually read**.

- If **unsuccessful**, read **returns –**1.
- The first argument is an integer, fdesc that refers to an opened file.
- The second argument, buf is the address of a buffer holding any data read.
- The third argument specifies how **many bytes of data are to be read from the file.**
- The size_t data type is defined in the <sys/types.h> header and should be the same as unsigned int.

read()..

- **There are several cases in which the number of bytes actually read is less than the amount requested:**
  - When reading from a **regular file**, if the end of file is reached before the requested number of bytes has been read. For example, if 30 bytes remain until the end of file and we try to read 100 bytes, read returns 30. The next time we call read, it will return 0 (end of file).
  - When reading from a **terminal device**. Normally, up to one line is read at a time.
  - When reading from a **network**. Buffering within the network may cause less than the requested amount to be returned.
  - When reading from a **pipe or FIFO**. If the pipe contains fewer bytes than requested, read will return only what is available.

write()

- The write system call is used to write data into a file.
- The write function puts data to a file in the form of **fixed block size referred by a given file descriptor.**
- The prototype of write is

```
#include<sys/types.h>
#include<unistd.h>
ssize_t write(int fdesc, const void *buf, size_t size);
```

If **successful**, write **returns the number of bytes actually written**.

- If **unsuccessful**, write **returns –1**.
- The first argument, fdesc is an integer that refers to an opened file.
- The second argument, buf is the address of a buffer that contains data to be written.
- The third argument, size specifies how many bytes of data are in the buf argument.
- The return value is usually equal to the number of bytes of data successfully written to a file. (*size* value)

close()

- The close system call is used to **terminate the connection** to a file from a process.
- The prototype of the close is

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

If **successful**, close **returns 0**.

- If **unsuccessful**, close **returns –1**.
- The **argument** fdesc refers to an opened file.
- Close function **frees the unused file descriptors** so that they can be reused to reference other files.
- The close function **de-allocates system resources** like file table entry and memory buffer allocated to hold the read/write.

lseek

- The lseek function is used to change the file offset to a different value.
- Thus lseek allows a process to perform **random access of data on any opened file.** The prototype of lseek is

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fdesc, off_t pos, int whence);
```

On **success it returns new file offset, and –1 on error**.

- The **first argument fdesc**, is an integer file descriptor that refer to an opened file.
- The **second argument** pos, specifies a byte offset to be added to a reference location in deriving the new file offset value.
- The third argument whence, is the reference location.

Whence value	Reference location
SEEK_CUR	Current file pointer address
SEEK_SET	The beginning of a file
SEEK_END	The end of a file

They are defined in the <unistd.h> header.

If an lseek call will result in a new file offset that is beyond the current end-of-file, two outcomes possible are:

- If a file is opened for read-only, lseek will fail.
- If a file is opened for write access, lseek will succeed.
- The data between the end-of-file and the new file offset address will be initialized with NULL characters

link()….

- The link function creates a new link for the existing file.
- The prototype of the link function is

```
#include <unistd.h>
int link(const char *cur_link, const char *new_link);
```

If **successful**, the link function **returns 0.**

- If **unsuccessful**, link **returns –1.**
- The first argument **cur_link, is the pathname of existing file**.
- The second argument **new_link is a new pathname to be assigned to the same file.**
- If this call succeeds, the hard link count will be increased by 1.
- The UNIX ln command is implemented using the link API.

unlink

- The unlink function deletes a link of an existing file.
- This function decreases the hard link count attributes of the named file, and removes the file name entry of the link from directory file.
- A file is removed from the file system when its hard link count is zero and no process has any file descriptor referencing that file.
- The prototype of unlink is

```
#include <unistd.h>
int unlink(const char * cur_link);
```

If successful, the unlink function returns 0.

- If unsuccessful, unlink returns –1.
- The argument cur_link is a path name that references an existing file.
- ANSI C defines the rename function which does the similar unlink operation.
- The prototype of the rename function is:

```
#include<stdio.h>
int rename(const char * old_path_name,const char * new_path_name);
```

stat, fstat

- The stat and fstat function retrieves the file attributes of a given file.
- The only difference between stat and fstat is that the first argument of a stat is a file pathname, where as the first argument of fstat is file descriptor.
- The prototypes of these functions are

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

int stat(const char *pathname, struct stat *statv);
int fstat(const int fdesc, struct stat *statv);
```

- The second argument to stat and fstat is the address of a struct stat-typed variable which is defined in the <sys/stat.h> header.
- Its declaration is as follows:
- **struct stat {**

  dev_t st_dev; /* file system ID */

  ino_t st_ino; /* file inode number */

  mode_t st_mode; /* contains file type and permission */ **nlink_t st_nlink;** /* hard link count */

  uid_t st_uid; /* file user ID */

  gid_t st_gid; /* file group ID */

  dev_t st_rdev; /*contains major and minor device#*/

  off_t st_size; /* file size in bytes */

  time_t st_atime; /* last access time */

  time_t st_mtime; /* last modification time */

  time_t st_ctime; /* last status change time */

  };

- The return value of both functions is
  - 0 if they succeed
  - -1 if they fail
  - *errno* contains an error status code
- The lstat function prototype is the same as that of stat:

```
int lstat(const char * path_name, struct stat* statv);
```

✓ We can determine the file type with the macros as shown.

macro	Type of file
S_ISREG()	regular file
S_ISDIR()	directory file
S_ISCHR()	character special file
S_ISBLK()	block special file
S_ISFIFO()	pipe or FIFO
S_ISLNK()	symbolic link
S_ISSOCK()	socket

access

- The access system call checks the existence and access permission of user to a named file.
- The prototype of access function is:

```
#include<unistd.h>
int access(const char *path_name, int flag);
```

On success access returns 0, on failure it returns –1.

- The first argument is the pathname of a file.
- The second argument flag, contains one or more of the following bit flag .

Bit flag	Uses
F_OK	Checks whether a named file exist
R_OK	Test for read permission
W_OK	Test for write permission
X_OK	Test for execute permission

- The flag argument value to an access call is composed by bitwise-ORing one or more of the above bit flags as shown:
- **int rc=access(“/usr/divya/usp.txt”,R_OK | W_OK);**
- example to check whether a file exists:

```
if(access(“/usr/divya/usp.txt”, F_OK)==-1)
        printf(“file does not exists”);
else
        printf(“file exists”);
```

chmod, fchmod

- The chmod and fchmod functions change file access permissions for owner, group & others as well as the set_UID, set_GID and sticky flags.
- A process must have the effective UID of either the super-user/owner of the file.
- The prototypes of these functions are

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

int chmod(const char *pathname, mode_t flag);
int fchmod(int fdesc, mode_t flag);
```

The pathname argument of chmod is the path name of a file whereas the fdesc argument of fchmod is the file descriptor of a file.

- The chmod function operates on the specified file, whereas the fchmod function operates on a file that has already been opened.
- To change the permission bits of a file, the effective user ID of the process must be equal to the owner ID of the file, or the process must have super-user permissions. The mode is specified as the bitwise OR of the constants shown below.

chown, fchown, lchown

- The chown functions changes the user ID and group ID of files.
- The prototypes of these functions are

The path_name argument is the path name of a file.

- The uid argument specifies the new user ID to be assigned to the file.
- The gid argument specifies the new group ID to be assigned to the file.

utime

- The utime function modifies the access time and the modification time stamps of a file.
- The prototype of utime function is

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

int utime(const char *path_name, struct utimbuf *times);
```

On success it returns 0, on failure it returns –1.

- The path_name argument specifies the path name of a file.
- The times argument specifies the new access time and modification time for the file.

- The struct utimbuf is defined in the <utime.h> header as:

  struct utimbuf {

  time_t actime; /* access time */

  time_t modtime; /* modification time */ }
- The time_t datatype is an unsigned long and its data is the number of the seconds elapsed since the birthday of UNIX : 12 AM , Jan 1 of 1970.
- If the times (variable) is specified as NULL, the function will set the named file access and modification time to the current time.
- If the times (variable) is an address of the variable of the type struct utimbuf, the function will set the file access time and modification time to the value specified by the variable.

File and Record Locking

- Multiple processes performs read and write operation on the same file concurrently.
- This provides a means for data sharing among processes, but it also renders difficulty for any process in determining when the other process can override data in a file.

- So, in order to overcome this drawback UNIX and POSIX standard support file locking mechanism.
- File locking is applicable for regular files.
- Only a process can impose a write lock or read lock on either a portion of a file or on the entire file.
- **write lock** prevents the other process from reading and writing on the locked file and setting any over-lapping read or write lock on the locked file.
- **read lock** is set, it prevents other processes from reading and setting any overlapping write locks on the locked region.

- The prototype of fcntl is

```
#include<fcntl.h>
int fcntl(int fdesc, int cmd_flag, ....);
```

The first argument specifies the file descriptor.

- The second argument cmd_flag specifies what operation has to be performed.
- If fcntl is used for file locking then it can values as

F_SETLK	sets a file lock, do not block if this cannot succeed immediately.
F_SETLKW	sets a file lock and blocks the process until the lock is acquired.
F_GETLK	queries as to which process locked a specified region of file.

- For file locking purpose, the third argument to fctnl is an address of a *struct flock* type variable.
- This variable specifies a region of a file where lock is to be set, unset or queried.

```
struct flock {
short l_type; /* what lock to be set or to unlock file */
short l_whence; /* Reference address for the next field */
off_t l_start ; /*offset from the l_whence reference addr*/
off_t l_len ; /*how many bytes in the locked region */
pid_t l_pid ; /*pid of a process which has locked the file */ };
```

- The l_type field specifies the lock type to be set or unset.

The possible values, which are defined in the <fcntl.h> header, and their uses are

l_type value	Use
F_RDLCK	Set a read lock on a specified region
F_WRLCK	Set a write lock on a specified region
F_UNLCK	Unlock a specified region

The l_whence, l_start & l_len define a region of a file to be locked or unlocked. The possible values of l_whence and their uses are

l_whence value	Use
SEEK_CUR	The l_start value is added to current file pointer address
SEEK_SET	The l_start value is added to byte 0 of the file
SEEK_END	The l_start value is added to the end of the file

- "**Exclusive lock**" - The intension of the write lock is to **prevent other processes from both reading and writing the locked region** while the process that sets the lock is modifying the region, so write lock is termed as **Exclusive lock**
- The use of read lock is to prevent other processes from writing to the locked region while the process that sets the lock is reading data from the region.
- Other processes are allowed to lock and read data from the locked regions. Hence a **read lock is also called as "shared lock ".**

- File lock may be **mandatory** if they are enforced by an operating system kernel.
- If a mandatory exclusive lock is set on a file**, no process can use the read or write system calls to access the data on the locked region**.
- These mechanisms can be used to **synchronize** reading and writing of shared files by multiple processes.
- If a process locks up a file, other processes that attempt to write to the locked regions are **blocked** until the former process releases its lock.
- Problem with mandatory lock is – if a **runaway** process sets a mandatory exclusive lock on a file and never unlocks it, then, no other process can access the locked region of the file until the runway process is killed or the system has to be rebooted.

- To make use of **advisory locks**, process that manipulate the same file must **co-operate** such that they follow the given below procedure for every read or write operation to the file.

1. Try to set a lock at the region to be accesses. If this fails, a process can either wait for the lock request to become successful.
2. After a lock is acquired successfully, read or write the locked region.
3. Release the lock.

- If a process sets a read lock on a file, for example from address 0 to 256, then sets a write lock on the file from address 0 to 512, the process will own only one write lock on the file from 0 to 512, the previous read lock from 0 to 256 is now covered by the write lock and the process does not own two locks on the region from 0 to 256. This process is called "**Lock Promotion**".
- Furthermore, if a process now unblocks the file from 128 to 480, it will own two write locks on the file: one from 0 to 127 and the other from 481 to 512. This process is called "**Lock Splitting**".
- UNIX systems provide fcntl function to support file locking to impose read or write locks on either a region or an entire file.

- **Example Program #include <unistd.h> #include<fcntl.h>**
  int main ( )
  {
  int fd;
  struct flock lock;
  fd=open(“Hello.txt”,O_RDONLY);
  lock.l_type=F_RDLCK;
  lock.l_whence=0;
  lock.l_start=10;
  lock.l_len=15;
  fcntl(fd,F_SETLK,&lock);
  }

Directory File API's

- A Directory file is a **record-oriented file**, where each record stores a file name and the inode number of a file that resides in that directory.
- Directories are created with the mkdir API and deleted with the rmdir API.
- The prototype of mkdir is

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

int mkdir(const char *path_name, mode_t mode);
```

The first argument is the path name of a directory file to be created.

- The second argument mode, specifies the **access permission** for the owner, groups and others to be assigned to the file. This function creates a new empty directory.
- The entries for "." and ".." are automatically created.
- To allow a process to scan directories in a file system independent manner, a directory record is defined as ***struct dirent*** in the <dirent.h> header for UNIX.

```
struct dirent {
  ino_t d_ino;                    /* i-node number */
  char  d_name[NAME_MAX + 1];     /* null-terminated filename */
}
```

- Some of the functions that are defined for directory file operations in the above header are

```
#include <dirent.h>

DIR *opendir(const char *pathname);
```

Returns: pointer if OK, NULL on error

```
struct dirent *readdir(DIR *dp);
```

Returns: pointer if OK, NULL at end of directory or error

```
void rewinddir(DIR *dp);

int closedir(DIR *dp);
```

Returns: 0 if OK, 1 on error

- The uses of these functions are

Function	Use
opendir	Opens a directory file for read-only. Returns a file handle dir * for future reference of the file.
readdir	Reads a record from a directory file referenced by dir-fdesc and returns that record information.
rewinddir	Resets the file pointer to the beginning of the directory file referenced by dir-fdesc. The next call to readdir will read the first record from the file.
closedir	closes a directory file referenced by dir-fdesc.

An empty directory is deleted with the rmdir API. The prototype of rmdir is

```
#include<unistd.h>
int rmdir (const char * path_name);
```

If the link count of the directory becomes 0, with the call and no other process has the directory open then the space occupied by the directory is freed.

```
• // C program to create a folder
#include <conio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
 void main()
{
   int check;
   char* dirname = “UNIX";
   clrscr();
    check = mkdir(dirname,0777);
    // check if directory is created or not
   if (!check)
      printf("Directory created\n");
   else {
      printf("Unable to create directory\n");
      exit(1);
   }
    getch();
   system("dir");
   }
```

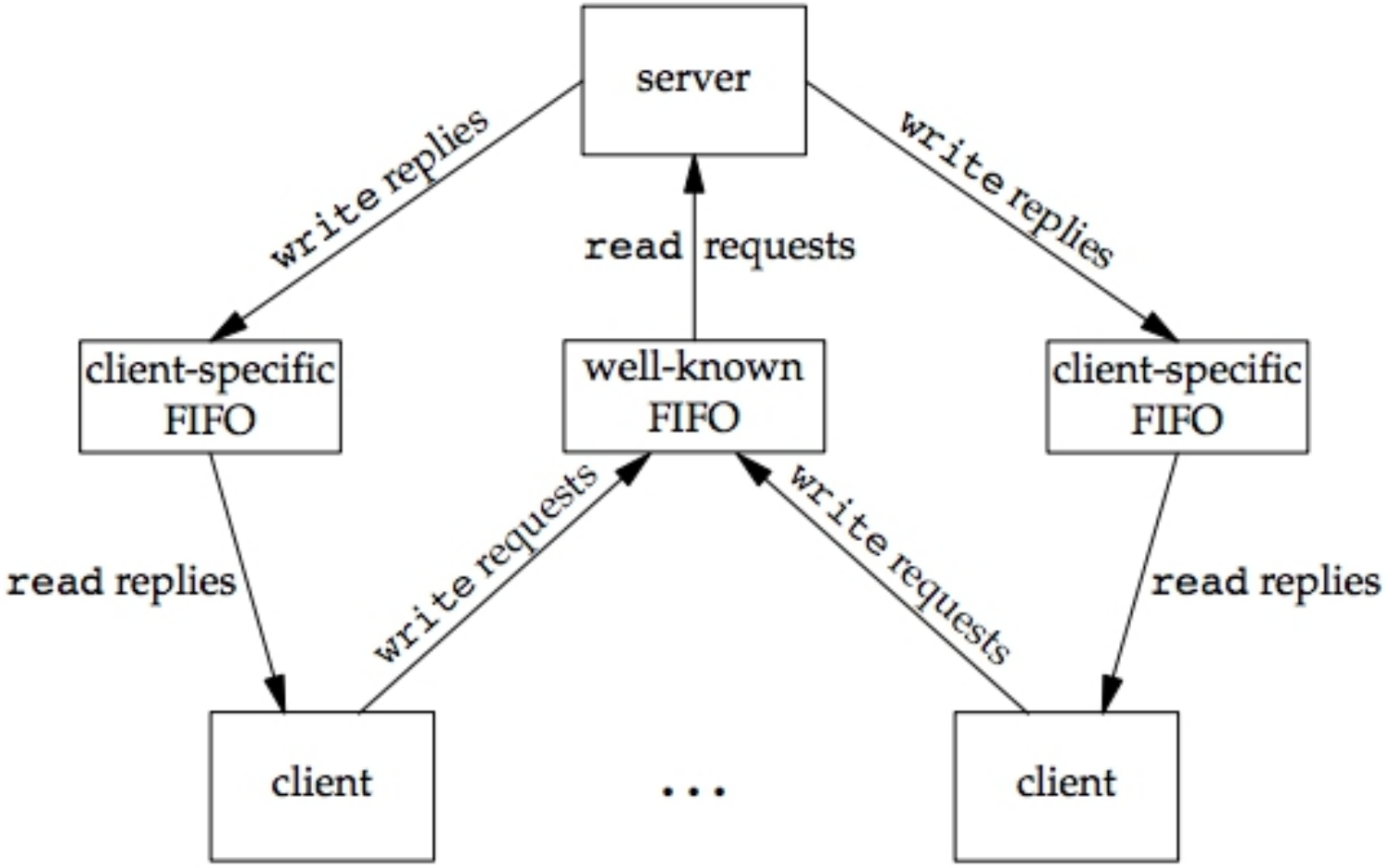
Device file APIs

- Device files are used to **interface physical device with application programs.**
- A process with **superuser privileges** to create a device file must call the **mknod API**.
- The user ID and group ID attributes of a device file are assigned in the same manner as for regular files.
- When a process reads or writes to a device file, the kernel uses the major and minor device numbers of a file to select a device driver function to carry out the actual data transfer.

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

int mknod(const char* path_name, mode_t mode, int device_id);
```

- The first argument pathname is the **pathname** of a device file to be created.
- The second argument mode specifies the **access permission**, , for the owner, group and others, also S_IFCHR or S_IBLK flag to be assigned to the file.
- The third argument device_id contains the major and minor device number.
- **Example**
- **mknod("SCSI5",S_IFBLK | S_IRWXU | S_IRWXG | S_IRWXO,(15<<8) | 3);**
- The above function creates a block device file "SCSI5", to which all the three i.e. read, write and execute permission is granted for user, group and others with major number as 8 and minor number 3.
- On success mknod API returns 0 , else it returns -1

FIFO File API's



FIFO file API's

- FIFO files are sometimes called **named pipes**.
- Pipes can be used only between related processes when a common ancestor has created the pipe.
- Creating a FIFO is similar to creating a file.
- Indeed the pathname for a FIFO exists in the file system.
- The prototype of mkfifo is

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

int mkfifo(const char *path_name, mode_t mode);
```

The first argument pathname is the pathname(filename) of a FIFO file to be created.

- The second argument mode specifies the access permission for user, group and others and as well as the S_IFIFO flag to indicate that it is a FIFO file.
- On success it returns 0 and on failure it returns –1.
- **Example**
- **mkfifo("FIFO5",S_IFIFO | S_IRWXU | S_IRGRP | S_ROTH);**

- Once we have created a FIFO using mkfifo, we open it using open.
- Indeed, the normal file I/O functions (**read, write, unlink etc**) all work with FIFOs.
- When a process opens a **FIFO file for reading**, the kernel will block the process until there is another process that opens the same file for writing.
- Similarly whenever a process **opens a FIFO file write**, the kernel will block the process until another process opens the same FIFO for reading.
- This provides a means for **synchronization** in order to undergo inter-process communication.
- If a particular process tries to write something to a **FIFO file that is full**, then that process will be blocked until another process has read data from the FIFO to make space for the process to write.
- Similarly, if a process attempts to **read data from an empty FIFO**, the process will be blocked until another process writes data to the FIFO.

- Another method to create FIFO files (not exactly) for inter-process communication is to use the pipe system call.
- The prototype of pipe is

```
#include <unistd.h>
int pipe(int fds[2]);
```

on success Returns 0 and –1 on failure.

If the pipe call executes successfully, the process can read from fd[0] and write to fd[1]. A single process with a pipe is not very useful. Usually a parent process uses pipes to communicate with its children.

Difference : hard link and symbolic link

Hard link	Symbolic link
Does not create a new inode	Create a new inode
Cannot link directories unless it is done by root	Can link directories
Cannot link files across file systems	Can link files across file system
Increase hard link count of the linked inode	Does not change hard link count of the linked inode

link()….

- The link function creates a new link for the existing file.
- The prototype of the link function is

```
#include <unistd.h>
int link(const char *cur_link, const char *new_link);
```

If **successful**, the link function **returns 0.**

- If **unsuccessful**, link **returns –1.**
- The first argument **cur_link, is the pathname of existing file**.
- The second argument **new_link is a new pathname to be assigned to the same file.**
- If this call succeeds, the hard link count will be increased by 1.
- The UNIX ln command is implemented using the link API.

Symbolic Link File API's

- A symbolic link is an indirect pointer to a file, unlike the hard links which pointed directly to the inode of the file.
- Symbolic links are developed to get around the limitations of hard links:
- Symbolic links can link files across file systems.
- Symbolic links can link directory files
- Symbolic links always reference the latest version of the files to which they link
- There are no file system limitations on a symbolic link and what it points to and anyone can create a symbolic link to a directory.
- A symbolic link is created with the symlink.

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

int ***symlink***(**const char** *org_link, **const char** *sym_link);

```
/* Program to illustrate symlink function */
#include<unistd.h>
#include<sys/types.h>
#include<string.h>

int main(int argc, char *argv[])
{
      char *buf [256], tname [256];
      if (argc ==4)
      return symlink(argv[2], argv[3]); /* create a symbolic link */
      else
      return link(argv[1], argv[2]); /* creates a hard link */
```

The Environment of a UNIX Process

- **INTRODUCTION**
- A Process is a program under execution in a UNIX or POSIX system.
- **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, and 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:
  - Return from `main`
  - Calling `exit`
  - Calling `_exit` or `_Exit`
  - Return of the last thread from its start routine
  - Calling `pthread_exit` from the last thread
- **Abnormal termination occurs in three ways:**
  - Calling `abort`
  - Receipt of a signal
  - Response of the last thread to a cancellation request

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);
```

All three exit functions expect a single integer argument, called the exit status. 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. In the following situations the exit status of the process is undefined.

- any of these functions is called without an exit status.
- main does a return without a return value

Example:

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

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.

The Environment of a UNIX Process

- **INTRODUCTION**
- A Process is a program under execution in a UNIX or POSIX system.
- **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, and 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.

- 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:
  - Return from `main`
  - Calling `exit`
  - Calling `_exit` or `_Exit`
  - Return of the last thread from its start routine
  - Calling `pthread_exit` from the last thread
- **Abnormal termination occurs in three ways:**
  - Calling `abort`
  - Receipt of a signal
  - Response of the last thread to a cancellation request

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);
```

All three exit functions expect a single integer argument, called the exit status. 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. In the following situations the exit status of the process is undefined.

- any of these functions is called without an exit status.
- main does a return without a return value

Example:

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

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

This declaration says that we 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.

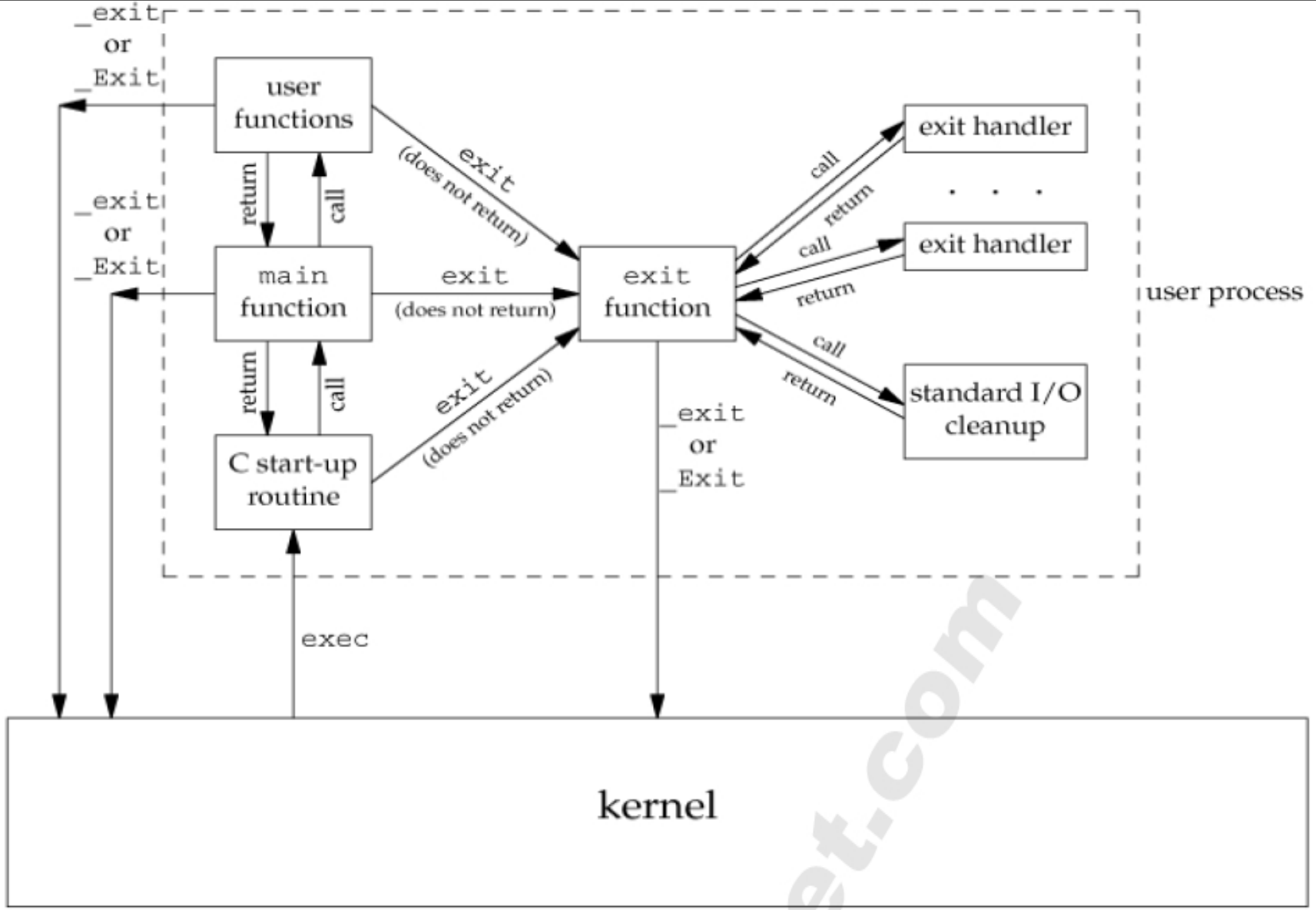
Each function is called as many times as it was registered

Example of exit handlers

```
#include "apue.h"

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");
}
```

- Output:

```
$ ./a.out
main is done
first exit handler
first exit handler
second exit handler
```



The figure summarizes how a C program is started and the various ways it can terminate.

COMMAND-LINE ARGUMENTS

- When a program is executed, the process that does the exec can pass command-line arguments to the new program.
- Example: 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);
}
```

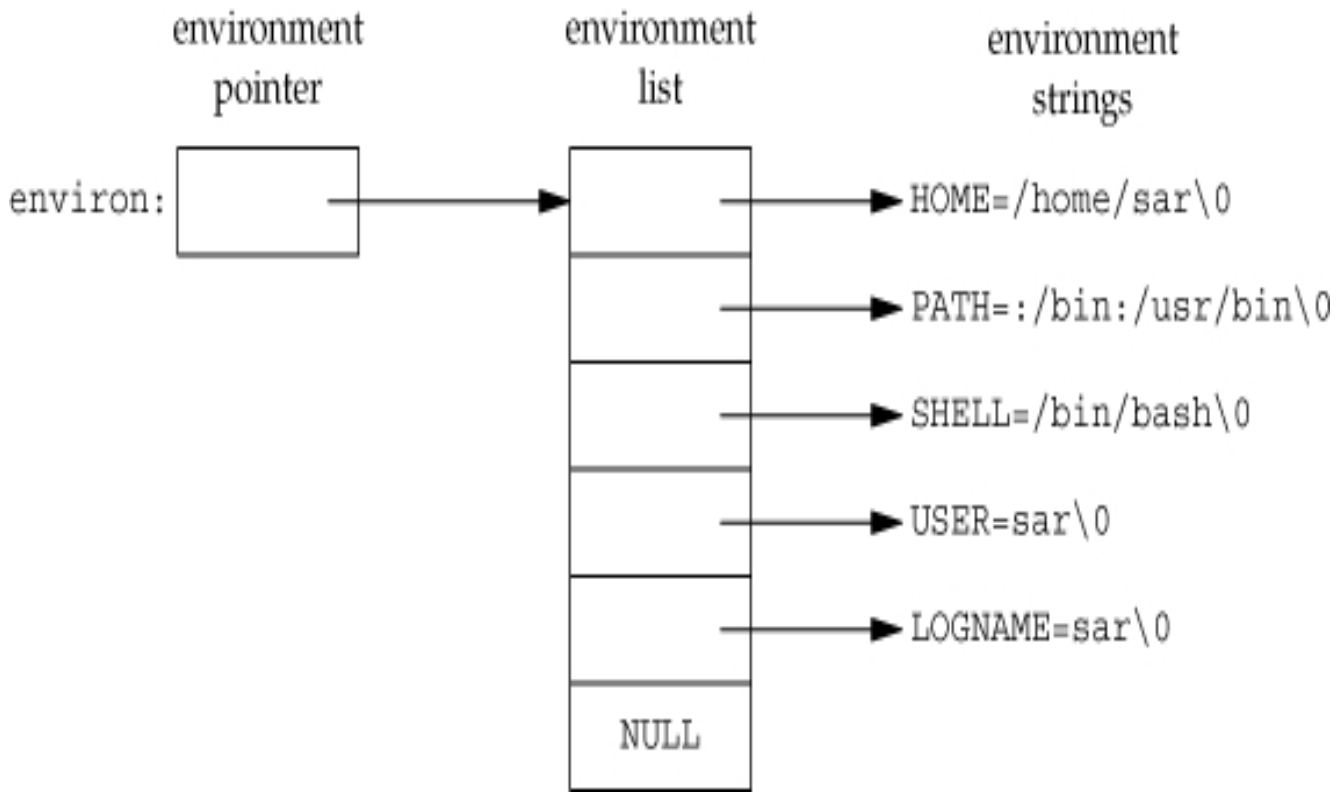
Output:

```
$ ./echoarg arg1 TEST foo
argv[0]: ./echoarg
argv[1]: arg1
argv[2]: TEST
argv[3]: foo
```

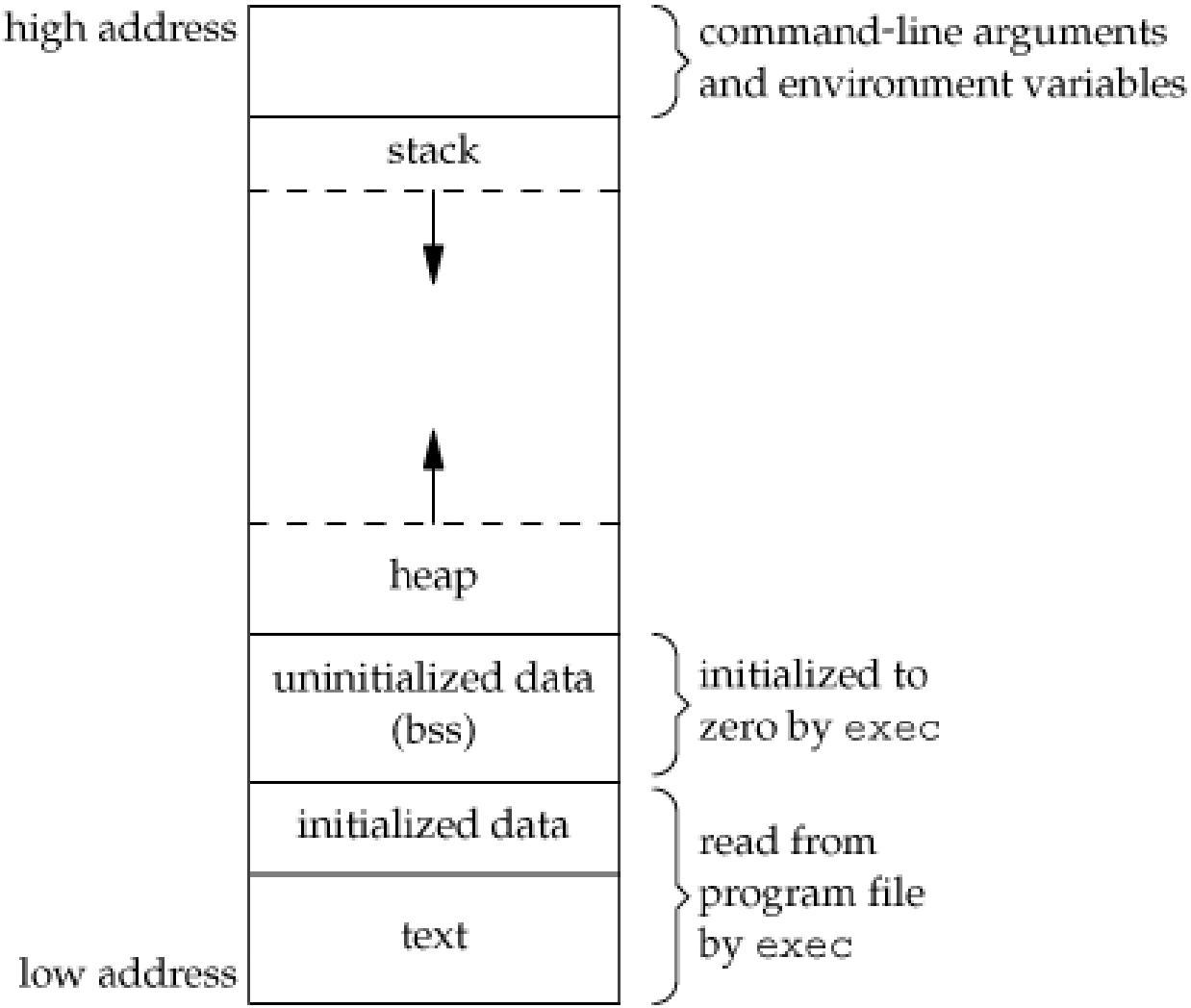
ENVIRONMENT LIST

- 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;**

Generally any environmental variable is of the form: ***name=value.***



MEMORY LAYOUT OF A C PROGRAM



MEMORY LAYOUT OF A C PROGRAM

- Historically, a C program has been composed of the following pieces:
- **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 read-only, 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.

- **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.
- **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. Historically, the heap has been located between the uninitialized data and the stack.

SHARED LIBRARIES

- Nowadays most UNIX systems support shared libraries. Shared libraries remove the **common library routines from the executable file,** instead maintaining 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 re-link,** edit every program that uses the library.

MEMORY ALLOCATION

- ISO C specifies three functions for memory allocation:
- **malloc**, which allocates a specified number of **bytes of memory**. The initial value of the memory is indeterminate.
- **calloc**, which allocates space for a specified **number of objects** of a specified size. The space is initialized to all 0 bits.
- **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. Also, when the size increases, the initial value of the space between the old contents and the end of the new area is indeterminate

```
#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.

- The **realloc** function lets us increase or decrease the size of a previously allocated area.
- For example, if we allocate room for 512 elements in an array that we fill in at runtime ,but find that we need room for more than 512 elements, we can call realloc.
- If there is room beyond the end of the existing region for the requested space, then realloc doesn't have to move anything; it simply allocates the additional area at the end and returns the same pointer that we passed it.
- But if there isn't room at the end of the existing region, realloc allocates another area that is large.
- The allocation routines are usually implemented with the **sbrk(2) system call**. Although sbrk can expand or contract the memory of a process, most versions of malloc and free never decrease their memory size.

Alternate Memory Allocators

- Many replacements for `malloc` and `free` are available.
- **`libmalloc`**
  - SVR4-based systems, such as Solaris, include the `libmalloc` library, provides a set of interfaces matching the ISO C memory allocation functions.
  - The `libmalloc` library includes `mallopt`, a function that allows a process to set certain variables that control the operation of the storage allocator.
  - A function called `mallinfo` is also available to provide statistics on the memory allocator.
- **`vmalloc`**
  - Describes a memory allocator that allows processes to allocate memory using different techniques for different regions of memory.

- **`quick-fit`**
  - malloc algorithm used either a best-fit or a first-fit memory allocation strategy.
  - Quick-fit is faster than either, but tends to use more memory.
- **alloca Function**
  - Same calling sequence as malloc; however, instead of allocating memory from the heap, the memory is allocated from the **stack frame of** the current function.
  - The advantage: free the space goes away automatically when the function returns.
  - The alloca function **increases the size of the stack frame**.

- **ENVIRONMENT VARIABLES**
- The environment strings are usually of the form: ***name=value.***
- Some, such as HOME and USER, are set automatically at login, and others are for us to set.
- Environment variables are set in a shell start-up file to control the shell's actions.
- The functions used to set and fetch values from the variables are setenv, putenv, and getenv functions. The prototype of these functions are

```
#include <stdlib.h>
char *getenv(const char *name);
```

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

- To change the value of an existing variable or add a new variable to the environment. The prototypes of these functions are

```
#include <stdlib.h>
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.

- 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.

Environment variables defined in the Single UNIX Specification

Variable	Description
COLUMNS	terminal width
DATEMSK	`getdate`(3) template file pathname
HOME	home directory
LANG	name of locale
LC_ALL	name of locale
LC_COLLATE	name of locale for collation
LC_CTYPE	name of locale for character classification
LC_MESSAGES	name of locale for messages
LC_MONETARY	name of locale for monetary editing
LC_NUMERIC	name of locale for numeric editing
LC_TIME	name of locale for date/time formatting
LINES	terminal height
LOGNAME	login name
MSGVERB	`fmtmsg`(3) message components to process
NLSPATH	sequence of templates for message catalogs
PATH	list of path prefixes to search for executable file
PWD	absolute pathname of current working directory
SHELL	name of user's preferred shell
TERM	terminal type
TMPDIR	pathname of directory for creating temporary files
TZ	time zone information

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.

```
#include <setjmp.h>
int setjmp(jmp_buf env);
```

Returns: 0 if called directly, nonzero if returning from a call to longjmp

void longjmp(jmp_buf env, int val)

- The setjmp function records or marks a location in a program code so that later when the longjmp function is called from some other function, the execution continues from the location onwards.
- The env variable(the first argument) records the necessary information needed to continue execution.
- The env is of the jmp_buf defined in <setjmp.h> file, it contains the task.

getrlimit AND setrlimit FUNCTIONS

- Every process has a set of resource limits, some of which can be queried and changed by the getrlimit and setrlimit functions.

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

int getrlimit(int resource, struct rlimit *rlptr);
int setrlimit(int resource, const struct rlimit  *rlptr);
```

- Both return: 0 if OK, nonzero on error
- Each call to these two functions specifies a single resource and a pointer to the following structure:
- **struct rlimit {**

  rlim_t rlim_cur; /* soft limit: current limit */

  rlim_t rlim_max; /* hard limit: maximum value for rlim_cur */

};

- Three rules govern the changing of the resource limits.

1. A process can change its soft limit to a value less than or equal to its hard limit.
2. A process can lower its hard limit to a value greater than or equal to its soft limit. This lowering of the hard limit is irreversible for normal users.
3. Only a superuser process can raise a hard limit.

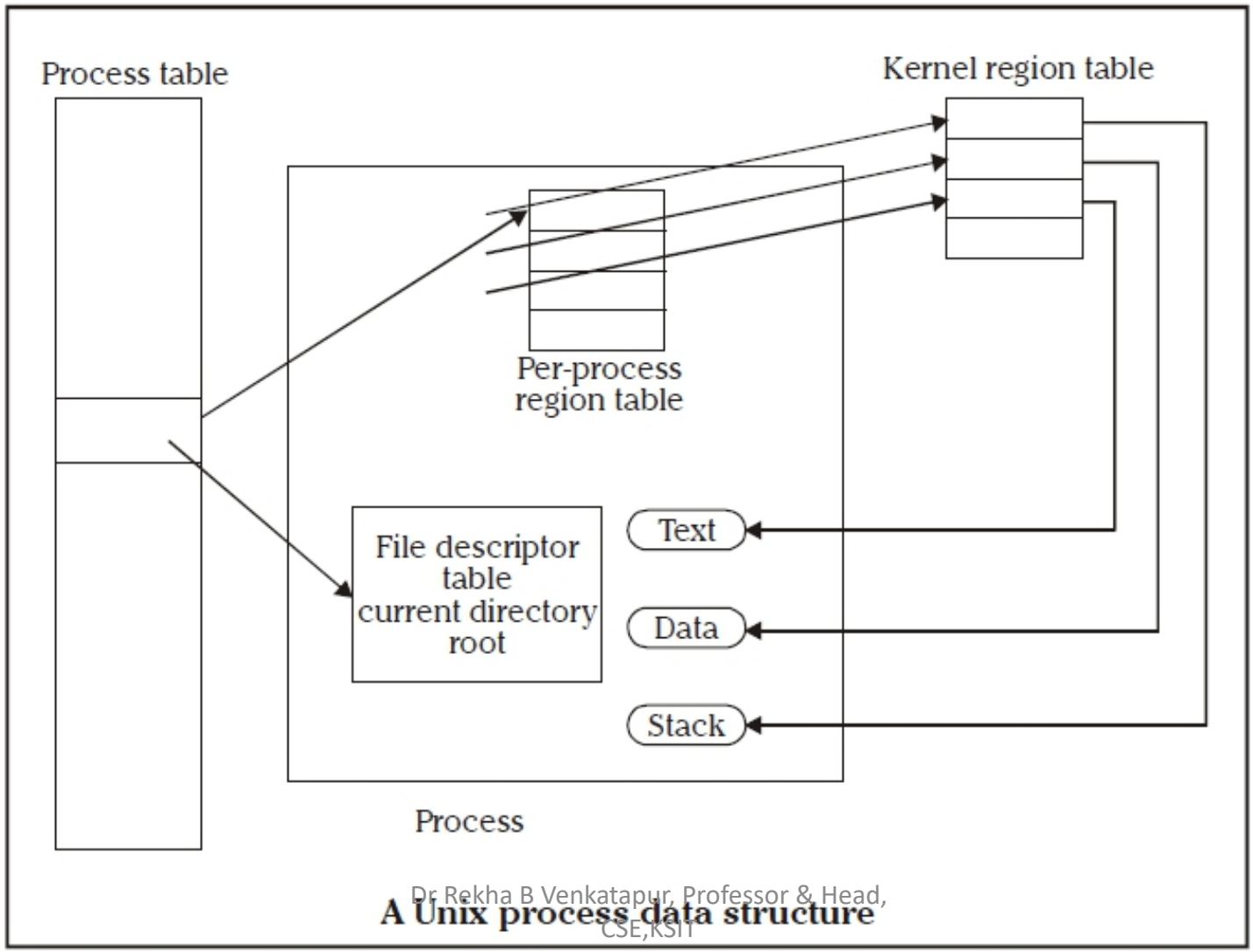
- An infinite limit is specified by the constant RLIM_INFINITY.

RLIMIT_AS	The maximum size in bytes of a process's total available memory.
RLIMIT_CORE	The maximum size in bytes of a core file. A limit of 0 prevents the creation of a core file.
RLIMIT_CPU	The maximum amount of CPU time in seconds. When the soft limit is exceeded, the `SIGXCPU` signal is sent to the process.
RLIMIT_DATA	The maximum size in bytes of the data segment: the sum of the initialized data, uninitialized data, and heap.
RLIMIT_FSIZE	The maximum size in bytes of a file that may be created. When the soft limit is exceeded, the process is sent the `SIGXFSZ` signal.
RLIMIT_LOCKS	The maximum number of file locks a process can hold.
RLIMIT_MEMLOCK	The maximum amount of memory in bytes that a process can lock into memory using `mlock`(2).
RLIMIT_NOFILE	The maximum number of open files per process. Changing this limit affects the value returned by the `sysconf` function for its `SC OPEN MAX` argument
RLIMIT_NPROC	The maximum number of child processes per real user ID. Changing this limit affects the value returned for `SC CHILD MAX` by the `sysconf` function
RLIMIT_RSS	Maximum resident set size (RSS) in bytes. If available physical memory is low, the kernel takes memory from processes that exceed their RSS.
RLIMIT_SBSIZE	The maximum size in bytes of socket buffers that a user can consume at any given time.
RLIMIT_STACK	The maximum size in bytes of the stack.
RLIMIT_VMEM	This is a synonym for `RLIMIT AS`.

UNIX KERNEL SUPPORT FOR PROCESS

- A UNIX process consists minimally of a text segment, a data segment and a stack segment. A segment is an area of memory that is managed by the system as a unit.
  - A text segment consists of the program text in machine executable instruction code format.
  - The data segment contains static and global variables and their corresponding data.
  - A stack segment contains runtime variables and the return addresses of all active functions for a process.

- UNIX kernel has a process table that keeps track of all active process present in the system.
- Some of these processes belongs to the kernel and are called as “system process”.
- Every entry in the process table contains pointers to the text, data and the stack segments and also to U-area of a process.
- U-area of a process is an extension of the process table entry and contains other process specific data such as the file descriptor table, current root and working directory inode numbers and set of system imposed process limits.



A Unix process data structure

- All processes in UNIX system expect the process that is created by the system boot code, are created by the fork system call.
- After the fork system call, once the child process is created, both the parent and child processes resumes execution.
- When a process is created by fork, it contains duplicated copies of the text, data and stack segments of its parent as shown in the Figure below.
- Also it has a file descriptor table, which contains reference to the same opened files as the parent, such that they both share the same file pointer to each opened files.

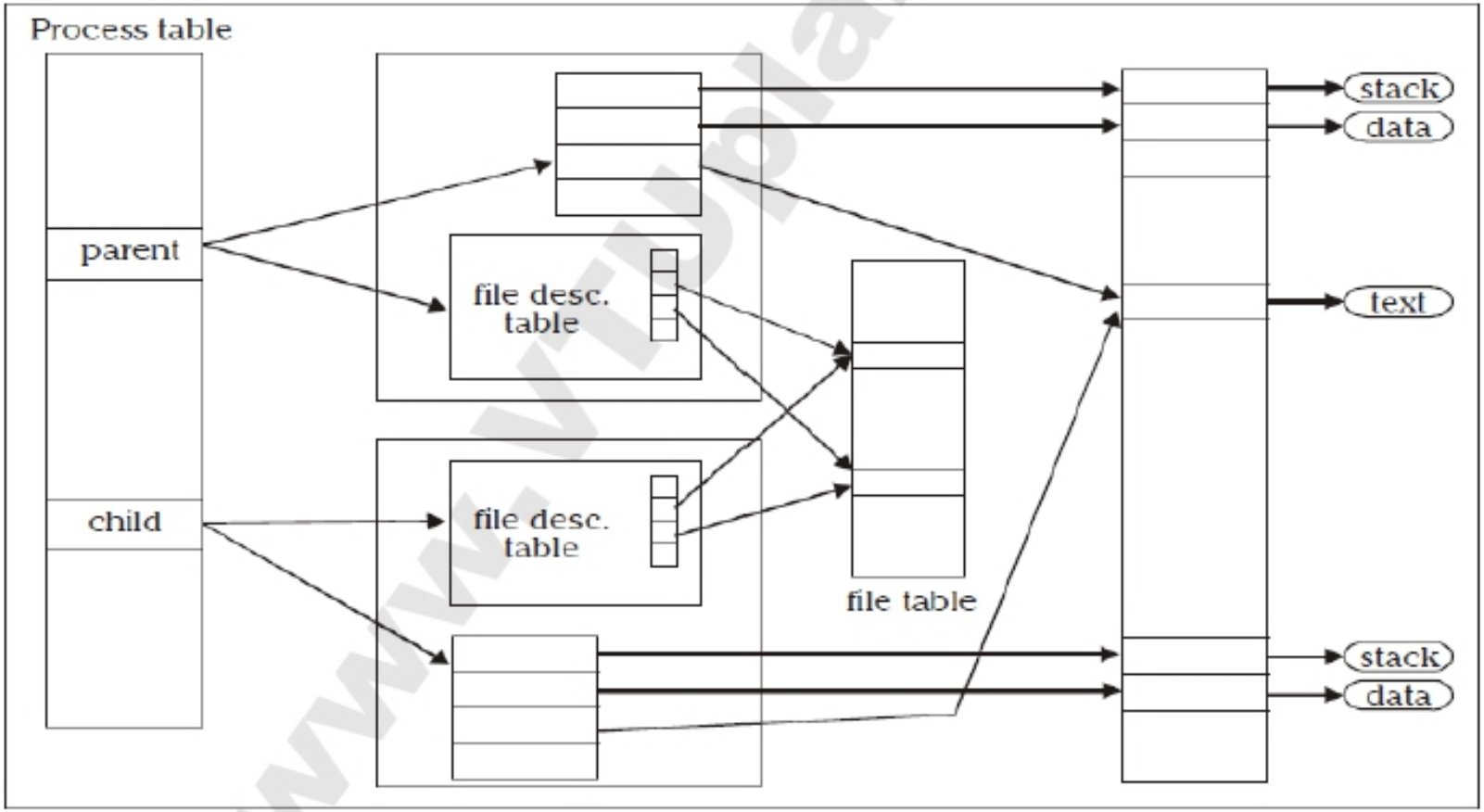


Figure: Parent & child relationship after fork

- The process will be assigned with attributes, which are either inherited from its parent or will be set by the kernel.
- **A real user identification number (rUID)**: the user ID of a user who created the parent process.
- **A real group identification number (rGID)**: the group ID of a user who created that parent process.
- **An effective user identification number (eUID)**: this allows the process to access and create files with the same privileges as the program file owner.
- **An effective group identification number (eGID)**: this allows the process to access and create files with the same privileges as the group to which the program file belongs.
- **Saved set-UID and saved set-GID**: these are the assigned eUID and eGID of the process respectively.

- **Process group identification number (PGID) and session identification number (SID):** these identify the process group and session of which the process is member.
- **Supplementary group identification numbers:** this is a set of additional group IDs for a user who created the process.
- **Current directory:** this is the reference (inode number) to a working directory file.
- **Root directory**: this is the reference to a root directory.
- **Signal handling**: the signal handling settings.
- **Signal mask**: a signal mask that specifies which signals are to be blocked.
- **Unmask**: a file mode mask that is used in creation of files to specify which accession rights should be taken out.
- **Nice value**: the process scheduling priority value.
- **Controlling terminal**: the controlling terminal of the process

- In addition to the above attributes, the following attributes are different between the parent and child processes:
- **Process identification number (PID)**: an integer identification number that is unique per process in an entire operating system.
- **Parent process identification number (PPID)**: the parent process PID.
- **Pending signals**: the set of signals that are pending delivery to the parent process.
- **Alarm clock time**: the process alarm clock time is reset to zero in the child process.
- **File locks**: the set of file locks owned by the parent process is not inherited by the child process.

PROCESS CONTROL

- **INTRODUCTION**
- Process control is concerned about creation of new processes, program execution, and process termination.
- **PROCESS IDENTIFIERS**
- **#include <unistd.h> pid_t getpid(void);**
- Returns: process ID of calling process
- **pid_t getppid(void);**
- Returns: parent process ID of calling process
- **uid_t getuid(void);**
- Returns: real user ID of calling process
- **uid_t geteuid(void);**
- Returns: effective user ID of calling process
- **gid_t getgid(void);**
- Returns: real group ID of calling process
- **gid_t getegid(void);**
- Returns: effective group ID of calling process

fork FUNCTION

- An existing process can create a new one by calling the fork function.

```
#include <unistd.h>
pid_t fork(void);
```

Returns: 0 in child, process ID of child in parent, 1 on error.

- The new process created by fork is called the **child process.**
- This function is called once **but returns twice.**
- The only difference in the returns is that the return value in the **child is 0**, whereas the return value in the **parent is the process ID of the new child.**
- The reason the child's process ID is returned to the parent is that a process can have **more than one child**, and there is no function that allows a process to obtain the process IDs of its children.

- The reason fork returns 0 to the child is that a process can have only a **single parent**, and the child can always call **getppid** to obtain the process ID of its parent. Both the child and the parent continue executing with the instruction that follows the call to fork.
- The child is a **copy of the parent**.
- For example, the child gets a copy of the parent's data space, heap, and stack.
- Note that this is a copy for the child; the parent and the child do not share these portions of memory.

- The parent and the child share the text segment .
- Example programs:
- **Program 1** /* Program to demonstrate fork function Program name – fork1.c */

```
#include<sys/types.h>
#include<unistd.h>
int main( )
{ fork( );
printf("\n hello USP");
}
```

Output :

```
$ cc fork1.c
$ ./a.out
hello USP
hello USP
```

Note : The statement hello USP is executed twice as both the child and parent have executed that instruction.

- **Program 2** /* Program name – fork2.c */

```
#include<sys/types.h>
#include<unistd.h>
int main( )
{ printf("\n 6 sem ");
fork( );
printf("\n hello USP");
}
```

Output :

```
$ cc fork1.c
$ ./a.out
6 sem
hello USP
hello USP
```

File Sharing

- Consider a process that has three different files opened for standard input, standard output, and standard error. On return from fork, we have the arrangement shown in Figure 8.2.

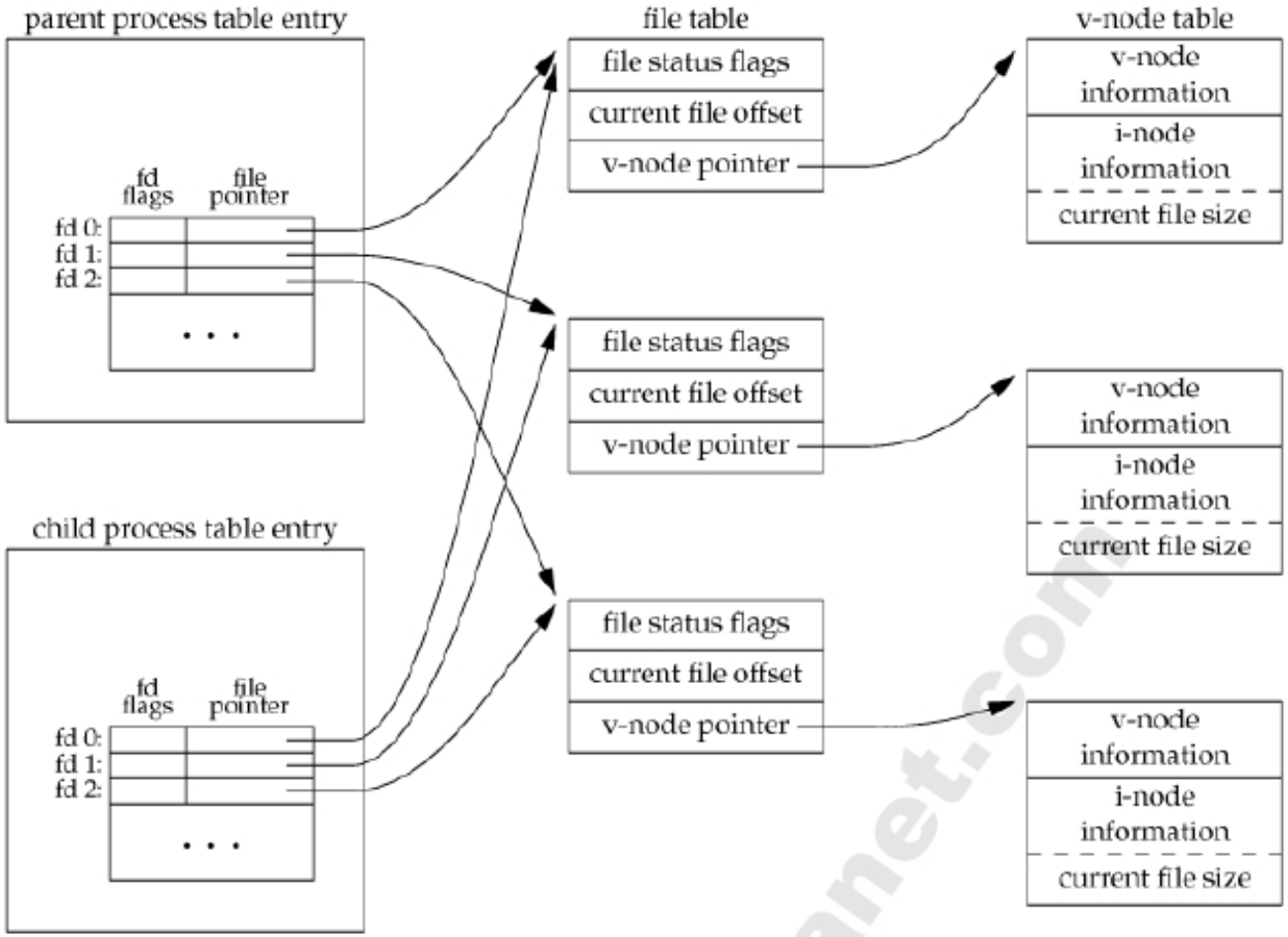


Figure 8.2 Sharing of open files between parent and child after fork

- It is important that the parent and the child share the same file offset.
- Consider a process that forks a child, then waits for the child to complete.
- Assume that both processes write to standard output as part of their normal processing.
- If the parent has its standard output redirected (by a shell, perhaps) it is essential that the parent's file offset be updated by the child when the child writes to standard output.
- In this case, the child can write to standard output while the parent is waiting for it; on completion of the child, the parent can continue writing to standard output, knowing that its output will be appended to whatever the child wrote.
- If the parent and the child did not share the same file offset, this type of interaction would be more difficult to accomplish and would require explicit actions by the parent.

- There are two normal cases for handling the descriptors after a fork.
- The **parent waits for the child to complete**.
- **Both the parent and the child go their own ways**
- There are numerous other properties of the parent that are inherited by the child:

- Real user ID, real group ID, effective user ID, effective group ID
- Supplementary group IDs
- Process group ID
- Session ID
- Controlling terminal
- The set-user-ID and set-group-ID flags
- Current working directory
- Root directory
- File mode creation mask
- Signal mask and dispositions
- The close-on-exec flag for any open file descriptors
- Environment
- Attached shared memory segments
- Memory mappings
- Resource limits

- The differences between the parent and child are
- The return value from fork
- The process IDs are different
- The two processes have different parent process IDs: the parent process ID of the child is the parent; the parent process ID of the parent doesn't change
- The child's tms_utime, tms_stime, tms_cutime, and tms_cstime values are set to 0
- File locks set by the parent are not inherited by the child
- Pending alarms are cleared for the child
- The set of pending signals for the child is set to the empty set

- The two main reasons for fork to fail are

  (a) if **too many processes** are already in the system, which usually means that something else is wrong, or

  (b) if the total number of processes for this real user ID **exceeds the system's limit.**

- There are two uses for fork:
  - When a **process wants to duplicate itself** so that the parent and child can each execute different sections of code at the same time. This is common for network servers, the parent waits for a service request from a client. When the request arrives, the parent calls fork and lets the child handle the request. The parent goes back to waiting for the next service request to arrive.
  - When a process wants to **execute a different program**. This is common for shells. In this case, the child does an exec right after it returns from the fork

vfork FUNCTION

- The function vfork has the **same calling sequence** and same return values as fork.
- The vfork function is intended to create a new process when the purpose of the new process is to **exec a new program**.
- The vfork function creates the new process, just like fork, without copying the address space of the parent into the child, as the child won't reference that address space; the **child simply calls exec (or exit) right after the vfork.**
- Another difference between the two functions is that vfork guarantees that the **child runs first**, until the child calls exec or exit. When the child calls either of these functions, the parent resumes.

Example of vfork function

```
#include "apue.h"
int glob = 6; /* external variable in initialized data */
int main(void)
{ int var; /* automatic variable on the stack */
pid_t pid; var = 88;
printf("before vfork\n"); /* we don't flush stdio */
if ((pid = vfork()) < 0)
{ err_sys("vfork error");
}
else if (pid == 0)
{ /* child */ glob++; /* modify parent's variables */
var++;
_exit(0); /* child terminates */
 } /*  Parent continues here. */
printf("pid = %d, glob = %d, var = %d\n", getpid(), glob, var); exit(0); }
```

Output:

```
$ ./a.out
before vfork
pid = 29039, glob = 7, var = 89
```

exit FUNCTIONS

- **A process can terminate normally in five ways:**

1. Executing a return from the main function.
2. Calling the exit function.
3. Calling the _exit or _Exit function.

   In most UNIX system implementations, exit(3) is a function in the standard C library, whereas _exit(2) is a system call.
4. Executing a return from the start routine of the last thread in the process. When the last thread returns from its start routine, the process exits with a termination status of 0.
5. Calling the pthread_exit function from the last thread in the process.

- The three forms of abnormal termination are as follows:
- Calling abort. This is a special case of the next item, as it generates the SIGABRT signal.
- When the process receives certain signals. Examples of signals generated by the kernel include the process referencing a memory location not within its address space or trying to divide by 0.
- The last thread responds to a cancellation request. By default, cancellation occurs in a deferred manner: one thread requests that another be canceled, and sometime later, the target thread terminates

wait AND waitpid FUNCTIONS

- When a process terminates, either normally or abnormally, the kernel notifies the parent by sending the SIGCHLD signal to the parent. Because the termination of a child is an asynchronous event - it can happen at any time while the parent is running - this signal is the asynchronous notification from the kernel to the parent.
- The parent can choose to ignore this signal, or it can provide a function that is called when the signal occurs: a signal handler.
- A process that calls wait or waitpid can:

1. Block, if all of its children are still running
2. Return immediately with the termination status of a child, if a child has terminated and is waiting for its termination status to be fetched
3. Return immediately with an error, if it doesn't have any child processes

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

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);
```

Both return: process ID if OK, 0 (see later), or 1 on error.

The differences between these two functions are as follows.

- The **wait** function can block the caller until a child process terminates, whereas waitpid has an option that prevents it from blocking.
- The **waitpid** function doesn't wait for the child that terminates first; it has a number of options that control which process it waits for.
- If a child has already terminated and is a zombie, wait returns immediately with that child's status.
- Otherwise, it blocks the caller until a child terminates. If the caller blocks and has multiple children, wait returns when one terminates.

- Print a description of the `exit` status **#include "apue.h"**

```
#include <sys/wait.h>
void pr_exit(int status)
{
if (WIFEXITED(status))
printf("normal termination, exit status = %d\n",
WEXITSTATUS(status));
else if (WIFSIGNALED(status))
printf("abnormal termination, signal number = %d%s\n",
WTERMSIG(status),
#ifdef WCOREDUMP WCOREDUMP(status) ? " (core file
generated)" : "");
#else
       "");
#endif
else if (WIFSTOPPED(status))
printf("child stopped, signal number = %d\n",
WSTOPSIG(status)); }
```

Program to Demonstrate various exit statuses

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

Int main(void)
{
    pid_t   pid;
    int     status;

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)                  /* child */
        exit(7);

    if (wait(&status) != pid)           /* wait for child */
        err_sys("wait error");
    pr_exit(status);                    /* and print its status */
```

```
    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)                  /* child */
        abort();                        /* generates SIGABRT */

    if (wait(&status) != pid)           /* wait for child */
        err_sys("wait error");
    pr_exit(status);                    /* and print its status */

    if ((pid = fork()) < 0)
        err_sys("fork error");
    else if (pid == 0)                  /* child */
        status /= 0;                    /* divide by 0 generates SIGFPE */

    if (wait(&status) != pid)           /* wait for child */
        err_sys("wait error");
    pr_exit(status);                    /* and print its status */

    exit(0);
}
```

- The interpretation of the pid argument for `waitpid` depends on its value: pid == 1 Waits for any child process. In this respect, waitpid is equivalent to wait.
- **pid > 0** Waits for the child whose process ID equals pid.
- **pid == 0** Waits for any child whose process group ID equals that of the calling process.
- **pid < 1** Waits for any child whose process group ID equals the absolute value of pid

Macros to examine the termination status returned by `wait` and `waitpid`	
Macro	**Description**
`WIFEXITED`(status)	True if status was returned for a child that terminated normally. In this case, we can execute `WEXITSTATUS` (status) to fetch the low-order 8 bits of the argument that the child passed to `exit`, `_exit`,or `_Exit`.
`WIFSIGNALED` (status)	True if status was returned for a child that terminated abnormally, by receipt of a signal that it didn't catch. In this case, we can execute `WTERMSIG` (status) to fetch the signal number that caused the termination. Additionally, some implementations (but not the Single UNIX Specification) define the macro `WCOREDUMP` (status) that returns true if a core file of the terminated process was generated.
`WIFSTOPPED` (status)	True if status was returned for a child that is currently stopped. In this case, we can execute `WSTOPSIG` (status) to fetch the signal number that caused the child to stop.
`WIFCONTINUED` (status)	True if status was returned for a child that has been continued after a job control stop

The options constants for `waitpid`	
Constant	**Description**
`WCONTINUED`	If the implementation supports job control, the status of any child specified by pid that has been continued after being stopped, but whose status has not yet been reported, is returned.
`WNOHANG`	The `waitpid` function will not block if a child specified by pid is not immediately available. In this case, the return value is 0.
`WUNTRACED`	If the implementation supports job control, the status of any child specified by pid that has stopped, and whose status has not been reported since it has stopped, is returned. The `WIFSTOPPED` macro determines whether the return value corresponds to a stopped child process.

- The waitpid function provides three features that aren't provided by the wait function.
- The waitpid function lets us wait for one particular process, whereas the wait function returns the status of any terminated child. We'll return to this feature when we discuss the popen function.
- The waitpid function provides a nonblocking version of wait. There are times when we want to fetch a child's status, but we don't want to block.
- The waitpid function provides support for job control with the WUNTRACED and WCONTINUED options.

Program to Avoid zombie processes by calling fork twice

```
#include "apue.h" #include <sys/wait.h>
 int main(void)
{
pid_t       pid;
 if ((pid = fork()) < 0) { err_sys("fork error");
} else if (pid == 0) {  /* first child */ if ((pid = fork()) < 0)
err_sys("fork error"); else if (pid > 0)
exit(0);    /* parent from second fork == first child */
/*
We're the second child; our parent becomes init as soon as our real parent calls exit() in the statement above. Here's where we'd continue executing, knowing that when we're done, init will reap our status. */ sleep(2);
printf("second child, parent pid = %d\n", getppid()); exit(0);
}
 if (waitpid(pid, NULL, 0) != pid)           /* wait for first child */ err_sys("waitpid error");
 /*
We're the parent (the original process); we continue executing, knowing that we're not the parent of the second child. */ exit(0);
}
```

- **Output:**

$ ./a.out

$ second child, parent pid = 1

wait3 AND wait4 FUNCTIONS

- The only feature provided by these two functions that isn't provided by the wait, waitid, and waitpid functions is an additional argument that allows the kernel to return a summary of the resources used by the terminated process and all its child processes. The prototypes of these functions are

```
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
pid_t wait3(int *statloc, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *statloc, int options, struct rusage *rusage);
```

Both return: process ID if OK,-1 on error

The resource information includes such statistics as the amount of user CPU time, the amount of system CPU time, number of page faults, number of signals received etc. the resource information is available only for terminated child process not for the process that were stopped due to job control.

RACE CONDITIONS

- A race condition occurs when multiple processes are trying to do something with shared data and the final outcome depends on the order in which the processes run

Example: The program below outputs two strings: one from the child and one from the parent. The program contains a race condition because the output depends on the order in which the processes are run by the kernel and for how long each process runs.

```
#include "apue.h"
static void charatatime(char *);
int main(void)
{
pid_t       pid;
 if ((pid = fork()) < 0)
{ err_sys("fork error");
}
else if (pid == 0)
{ charatatime("output from child\n");}
else
{charatatime("output from parent\n");}
exit(0);
}
 static void charatatime(char *str)
{
char        *ptr;
int         c;
 setbuf(stdout, NULL);          /* set unbuffered */
for (ptr = str; (c = *ptr++) != 0; )
putc(c, stdout);}
```

Output:

```
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
ooutput from child
utput from parent
$ ./a.out
output from child
output from parent
```

program modification to avoid race condition

```
#include "apue.h"
static void charatatime(char *);
int main(void)
{
pid_t       pid;
+           TELL_WAIT();
+
if ((pid = fork()) < 0) { err_sys("fork error");
} else if (pid == 0) {
+           WAIT_PARENT();     /* parent goes first */
charatatime("output from child\n");
} else {
charatatime("output from parent\n");
+           TELL_CHILD(pid);
}
exit(0);
}
```

```
static void charatatime(char *str)
{
char   *ptr;
int     c;

setbuf(stdout, NULL);   /* set unbuffered */
for (ptr = str; (c = *ptr++) != 0; )
putc(c, stdout);
}
```

exec FUNCTIONS

- When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function.
- The process ID does not change across an exec, because a new process is not created; exec merely replaces the current process - its text, data, heap, and stack segments - with a brand new program from disk. There are 6 exec functions:

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0,... /* (char *)0 */ );
int execv(const char *pathname, char *const argv []);
int execle(const char *pathname, const char *arg0,... /*(char *)0, char
*const envp */ );
int execve(const char *pathname, char *const argv[], char *const envp[]);
int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );
int execvp(const char *filename, char *const argv []);
```

- All six return: -1 on error, no return on success.
- The **first difference** in these functions is that the first four take a pathname argument, whereas the last two take a filename argument. When a filename argument is specified
  - If filename contains a slash, it is taken as a pathname.
  - Otherwise, the executable file is searched for in the directories specified by the PATH environment variable.
- The **next difference** concerns the passing of the argument list (l stands for list and v stands for vector). The functions execl, execlp, and execle require each of the command-line arguments to the new program to be specified as separate arguments. For the other three functions (execv, execvp, and execve), we have to build an array of pointers to the arguments, and the address of this array is the argument to these three functions.
- The **final difference** is the passing of the environment list to the new program. The two functions whose names end in an e (execle and execve) allow us to pass a pointer to an array of pointers to the environment strings. The other four functions, however, use the environ variable in the calling process to copy the existing environment for the new program.

Function	pathname	filename	Arg list	argv[]	environ	envp[]
execl	•		•		•	
execlp		•	•		•	
execle	•		•			•
execv	•			•	•	
execvp		•		•	•	
execve	•			•		•
(letter in name)		p	l	v		e

The above table shows the differences among the 6 exec functions.

- We've mentioned that the process ID does not change after an exec, but the new program inherits additional properties from the calling process:

o Process ID and parent process ID

o Real user ID and real group ID

o Supplementary group IDs

o Process group ID

o Session ID

o Controlling terminal

o Time left until alarm clock

o Current working directory

o Root directory

o File mode creation mask

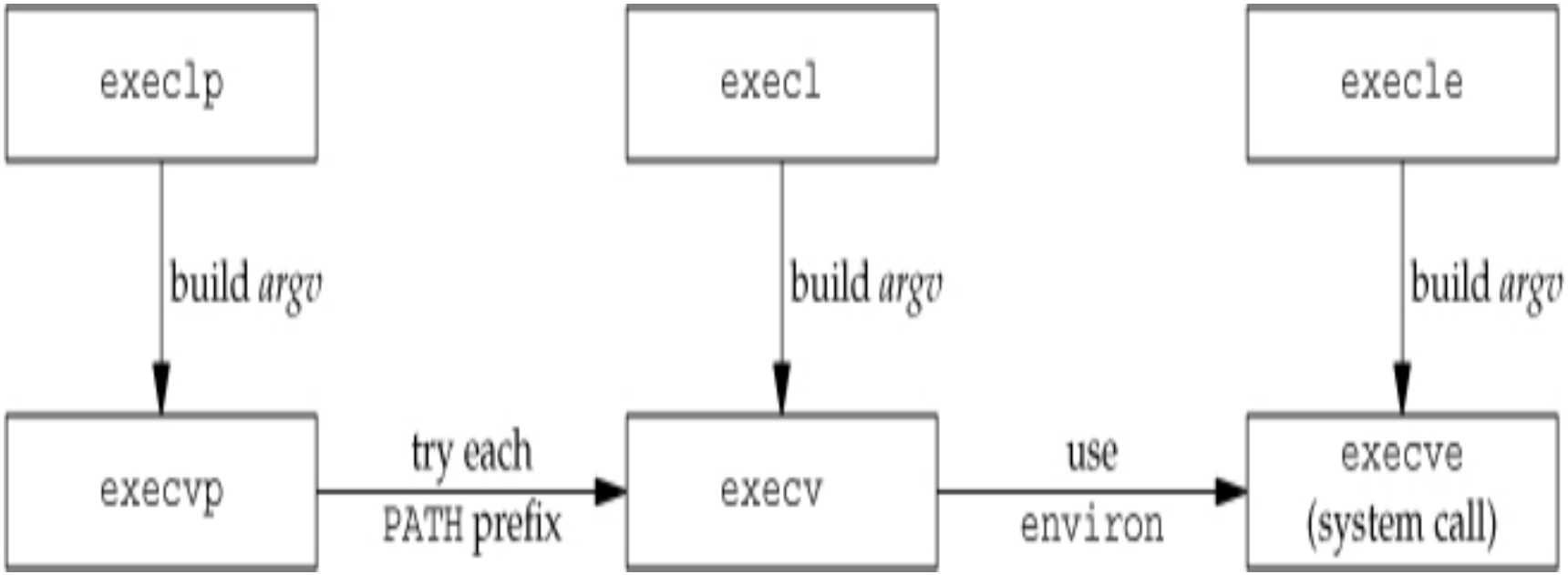
o File locks

o Process signal mask

o Pending signals

o Resource limits

o Values for tms_utime, tms_stime, tms_cutime, and tms_cstime.


execlp
execl
execle
build argv
build argv
build argv
execvp
try each
PATH prefix
execv
use
environ
execve
(system call)

Example of exec functions

```
#include "apue.h"
#include <sys/wait.h>
char *env_init[] = { "USER=unknown", "PATH=/tmp", NULL };
int main(void)
{
pid_t pid;
if ((pid = fork()) < 0)
{ err_sys("fork error");}
else if (pid == 0) {  /* specify pathname, specify environment */ if
(execle("/home/sar/bin/echoall", "echoall", "myarg1",
"MY ARG2", (char *)0, env_init) < 0) err_sys("execle error");
}
if (waitpid(pid, NULL, 0) < 0) err_sys("wait error");
if ((pid = fork()) < 0) { err_sys("fork error");
}
else if (pid == 0) {  /* specify filename, inherit environment */ if (execlp("echoall",
"echoall", "only 1 arg", (char *)0) < 0)
err_sys("execlp error");
}
exit(0);
```

- **Output:**

```
$ ./a.out
argv[0]: echoall
argv[1]: myarg1
argv[2]: MY ARG2
USER=unknown
PATH=/tmp
$ argv[0]: echoall
argv[1]: only 1 arg
USER=sar
LOGNAME=sar
SHELL=/bin/bash
            47 more lines that aren't shown
HOME=/home/sar
```

Echo all command-line arguments and all environment strings

```
#include "apue.h"
 int main(int argc, char *argv[])
{
int       i;
char     **ptr; extern char **environ;
 for (i = 0; i < argc; i++)          /* echo all command-line args */
printf("argv[%d]: %s\n", i, argv[i]);
 for (ptr = environ; *ptr != 0; ptr++)   /* and all env strings */
printf("%s\n", *ptr);
 exit(0);
}
```