UNIX FILE-RELATED SYSTEM CALLS:

- There are many times when it would be handy to create a file and to delete that file when the program no longer needs it. Two UNIX system calls allow this: *creat* and *unlink*. Creat behaves like *touch* and unlink behaves like *rm*. These are handy to use when to files need to access a common data file (we then use creat and unlink to make a semaphore).

- In certain cases, it is possible that another process will attempt to grab our newly created file (and possible rendering it useless to us). To reduce this possibility, we use the UNIX system call *tmpnam*, *tempnam*, or *mkstemp)* to automatically create (unique?) file names.

- **CREAT**

  The creat() function establishes a connection between the file named by the path parameter and a file descriptor. The opened file descriptor is used by subsequent I/O functions, such as read() and write(), to access that file. The calling process is suspended until the request is completed.

  SYNOPSIS

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

  int creat (const char *path, mode_t mode);
  ```

  PARAMETERS

  path - Specifies the file to be opened or created.

mode - Specifies the read, write, and execute permissions of the file to be created. If the file already exists, this parameter is ignored.

NOTES:

1. Files created with creat are opened for write only by default.

- EXAMPLE (create a file):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>          // CREAT
#include <sys/stat.h>       // CREAT
#include <sys/types.h>      // CREAT
#include <errno.h>
#include <unistd.h>         // UNLINK and SLEEP

int main(void){
int fd;

    fd = creat("/tmp/junk",0666);
    unlink(“/tmp/junk”);
    return 0;
}
```

- EXAMPLE (semaphores via CREAT - will not if superuser!):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>           // CREAT
#include <sys/stat.h>        // CREAT
#include <sys/types.h>       // CREAT
#include <errno.h>
#include <unistd.h>          // UNLINK and SLEEP

int LOCK(char *name){
int fd, tries;
extern int errno;

    tries = 0;
    while (((fd = creat(name, 0)) == -1) &&
            (errno == EACCES)) {
        if (++tries == 5) return 0;
        sleep(1);
    }
    if ((fd == -1) || (close(fd) == -1)) {
       printf("System lock error\n");
       return 0;
    }
    return 1;
}

void UNLOCK(char *name){

   if (unlink(name) == -1) {
      printf("Unlock error\n");
   }
}

int main(void){
char filename[256];

    strcpy(filename, "/tmp/junk"));

    if (LOCK(filename)) {
       printf("Account open and locked.\n");

       // manipulate account.
```

```
            sleep(1);
            UNLOCK(filename);
        } else {
            printf("Account locked.\n");
        }
        return 0;
    }
```

- **OPEN**

The open() function establishes a connection between the file named by the path parameter and a file descriptor. The opened file descriptor is used by subsequent I/O functions, such as read() and write(), to access that file. The calling process is suspended until the request is completed.

SYNOPSIS

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

int open (const char *path, int oflag , mode_t mode);
```

PARAMETERS

path - Specifies the file to be opened or created.

oflag - Specifies the file access permissions (read, write, both, create, truncate an existing file, append, test for file existence, etc).

mode - [Optional in most cases] Specifies the read, write, and execute permissions of the file to be created. If the file already exists, this parameter is ignored.

NOTES:

1. Open is a fancier more powerful version of creat.

2. The truncate option will destroy the contents of a file.

3. Traditionally open is only used to access existing files. However, there are ways around this.

- **UNLINK**

  The unlink() function removes the directory entry specified by the path parameter and, if the entry is a hard link, decrements the link count of the file referenced by the link.

  When all links to a file are removed and no process has the file open or mapped, all resources associated with the file are reclaimed, and the file is no longer accessible. If one or more processes have the file open or mapped when the last link is removed, the link is removed before the unlink() function returns, but the removal of the file contents is postponed until all open or map references to the file are removed. If the path parameter names a symbolic link, the symbolic link itself is removed.

  SYNOPSIS

  ```
  #include <unistd.h>

  int unlink (const char *path);
  ```

  PARAMETERS

  path - Specifies the file to be opened or created.

  NOTES:

  1. A hard link to a directory cannot be unlinked.

  2. Upon successful completion, a value of 0 (zero) is returned. If the unlink() function fails, a value of -1 is returned, the named file is not changed, and errno is set to indicate the error.

3. The unlink command cannot be used to unlink a directory.

4. A process must have write access to the parent directory of the file to be unlinked with respect to all access policies.

- **TMPNAM / TEMPNAM**

  The tmpnam() and tempnam() functions generate filenames for temporary files.

  BOTH OF THESE COMMANDS ARE CONSIDERED TO BE SECURITY WEAK POINTS – DO NOT USE THEM.

- **MKSTEMP**

  The mkstemp() function generates a unique temporary file name from template. The last six characters of template must be XXXXXX and these are replaced with a string that makes the filename unique.

  SYNOPSIS

  ```
  #include <stdlib.h>

  int mkstemp (char *template);
  ```

  PARAMETERS

  template – The string from which to make the filename.

  NOTES:

  1. Since it will be modified, template must not be a string constant, but should be declared as a character array.

- EXAMPLE (create a file with a unique name):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>          // CREAT
#include <sys/stat.h>       // CREAT
#include <sys/types.h>      // CREAT
#include <errno.h>
#include <unistd.h>         // UNLINK and SLEEP

void main(void){
char tmp[10];
int  fd;

    strcpy(tmp, "RAM_XXXXXX");
    mkstemp(tmp);
//    unlink(tmp);
}
```

- **OUTPUT:**

```
UNIX system calls test (with the file RAM_2vaWX1 created
in the current directory)
```

- **READ**

  The read() function attempts to read nbytes of data from the file associated with the fd parameter to the buffer pointed to by the buffer parameter.

  SYNOPSIS

  ```
  #include <unistd.h>

  ssize_t read(int fd, void *buffer, size_t nbytes);
  ```

  PARAMETERS

  fd - Identifies the file from which the data is to be read.

  buffer - Points to the buffer to contain the data to be read.

  nbytes - Specifies the number of bytes to read from the file associated with the fd parameter.

  NOTES:

  1. Read returns the number of bytes read.

- **WRITE**

  The write() function attempts to write nbytes of data to the file associated with the fd parameter from the buffer pointed to by the buffer parameter.

  SYNOPSIS

  ```
  #include <unistd.h>

  ssize_t write(int fd, const void *buffer, size_t nbytes);
  ```

  PARAMETERS

  fd - Identifies the file to which the data is to be written.

  buffer - Points to the buffer containing the data to be written.

  nbytes - Specifies the number of bytes to write to the file associated with the fd parameter.

  NOTES:

  1. Write returns the number of bytes written.

  2. Write does not guarantee that the data was successfully written to the disk, only to the disk buffer.

- **CLOSE**

The close() function closes the file associated with the fd parameter.

When all file descriptors associated with a pipe or FIFO special file have been closed, any data remaining in the pipe or FIFO is discarded. When all file descriptors associated with an open file descriptor are closed, the open file descriptor is freed. If the link count of the file is 0 (zero) when all file descriptors associated with the file have been closed, the space occupied by the file is freed and the file is no longer accessible.

SYNOPSIS

```
 #include <unistd.h>

 int close (int fd);
```

PARAMETERS

fd - Identifies the file to be closed.

NOTES:

1. Close does not flush the buffer to the disk. All it does is make the file descripter available for reuse.

- EXAMPLE (open a file or create it if it does not exist):

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>            // CREAT/OPEN
#include <sys/stat.h>         // CREAT/OPEN
#include <sys/types.h>        // CREAT/OPEN
#include <errno.h>
#include <unistd.h>           // UNLINK
#include <errno.h>            // errno

int main(void){
int fd;
char file [256];

    strcpy(file, “/tmp/junk”));

   if ((fd = open(file, O_WRONLY)) == -1){

      if (errno == ENOENT)  {
         printf("File does not exist.\n");

         if ((fd = creat((file,0666)) == -1) {
            printf("File creation error.\n");
         } else {
            printf("File created.\n");
         }

      } else {
         printf("File open error.\n");
      }

    } else {
       printf("File opened.\n");
    }

//    unlink((“/tmp/junk”);
    return 0;
}
```

- EXAMPLE (open a file or create it if it does not exist):

```
#include <stdlib.h>
#include <fcntl.h>          // CREAT/OPEN
#include <sys/stat.h>       // CREAT/OPEN
#include <sys/types.h>      // CREAT/OPEN
#include <errno.h>
#include <unistd.h>         // UNLINK
#include <errno.h>          // errno

int main(void){
int  fd;
char file [256];

    strcpy(file, “/tmp/junk”));

    if ((fd = open(file, O_WRONLY|O_CREAT, 0666)) == -1){
       printf("File open error.\n");
    } else {
       printf("File opened.\n");
    }
    return 0;
}
```

- EXAMPLE (semaphores via OPEN - will work even if superuser!):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>           // CREAT
#include <sys/stat.h>        // CREAT
#include <sys/types.h>       // CREAT
#include <unistd.h>          // UNLINK and SLEEP
#include <errno.h>

int LOCK(char *name){
int fd, tries;
extern int errno;
    tries = 0;
    while ((fd = open(name, O_WRONLY|O_CREAT|O_EXCL,
                      0606)) == -1 &&
                      errno = EEXIST) {
        if (++tries == 5) return 0;
        sleep(1);
    }
    if ((fd == -1) || (close(fd) == -1)) {
       printf("System lock error\n");
       return 0;
    }
    return 1;
}

void UNLOCK(char *name){
   if (unlink(name) == -1) printf("Unlock error\n");
}

int main(void){
char file[L_tmpnam];

    strcpy(file, "/tmp/ram"));
    if (LOCK(file)) {
       printf("Account open and locked.\n");

       // manipulate account.

       sleep(1);
       UNLOCK(file);
    } else {
        printf("Account locked.\n");
    }
    return 0;
}
```

- EXAMPLE (writing to a temporary file):

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>          // CREAT
#include <sys/stat.h>       // CREAT
#include <sys/types.h>      // CREAT
#include <unistd.h>         // UNLINK

void main(void){
int  fd;
int x;
char buffer[] = "Hello";
char buffer2[] = ".....";

    // Create a temporary file and write to it.
    fd = creat("/tmp/junk", 0666);
    write(fd, buffer, strlen(buffer));
    close(fd);

    // Read from temporary file.
    fd = open(("/tmp/junk", O_RDWR, 0666);
    read(fd, buffer2, strlen(buffer));
    close(fd);

    printf("%s\n", buffer2);

    // Destroy temporary file.
    unlink(("/tmp/junk");
}
```

- **LINK**

  The link() function creates an additional hard link (directory entry) for an existing file. The old and the new link share equal access rights to the underlying object. The link() function atomically creates a new link for the existing file and increments the link count of the file by one.

  Both the path1 and path2 parameters must reside on the same file system. A hard link to a directory cannot be created.

  SYNOPSIS

  ```
  #include <unistd.h>

  int link (const char *path1, const char *path2 );
  ```

  PARAMETERS

  path1 - Points to the pathname of an existing file.

  path2 - Points to the pathname for the directory entry to be created. If the path2 parameter names a symbolic link, an error is returned.

  NOTES:

1. A process must have write permission in the target directory with respect to all access control policies configured on the system.

- EXAMPLE (moving a file):

```
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>               // CREAT
#include <sys/stat.h>            // CREAT
#include <sys/types.h>           // CREAT
#include <unistd.h>              // UNLINK

void move(char *from, char *to)
{
int fd;
int isdir;
extern int errno;

   isdir = ((fd = open(to, O_WRONLY)) == -1 &&
             errno == EISDIR);

   if (fd != -1 && close(fd) == -1) {
      printf("Target is invalid.\n");
      return;
   if (isdir != 1) {
      if (unlink(to) == -1 && errno != ENOENT) {
         printf("Target can not be written.\n");
         return;
      }
   }
   if (isdir == 1) {
      strcat(to, "/");
      strcat(to, from);
   }
   if (link(from, to) == -1) {
      printf("Error linking file.\n");
      return;
   }
   if (unlink(from) == -1) {
      printf("Error unlinking file.\n");
      return;
   }
}
```

- **MKNOD**

  The mknod() function creates a special file or FIFO (which creat or open cannot do). Using the mknod() function to create file types other than FIFO special requires superuser privilege.

  For the mknod() function to complete successfully, a process must have search permission and write permission in the parent directory of the path parameter.

  SYNOPSIS

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

  int mknod (const char *path, int mode, dev_t device);
  ```

  PARAMETERS

    path - Names the new file. If the final component of the path parameter names a symbolic link, the link will be traversed and pathname resolution will continue.

    mode - Specifies the file type, attributes, and access permissions.

    device - Depends upon the configuration and is used only if the mode parameter specifies a block or character special file. If the file you specify is a remote file, the value of the device parameter must be meaningful on the node where the file resides.

  NOTES:

  1. A mistake(?) in UNIX only allows the superuser to create directories with mknod.

2. When FIFOs were first introduced, how to make one was a mystery (not even mentioned in the UNIX text at the time).

- EXAMPLE #1 (making a FIFO):

```
#include <stdio.h>
#include <sys/stat.h>
#include <sys/mode.h>

void main(void){
   printf("%d\n", mknod("tmp", S_IFIFO | 0666, 0));
   //printf("%d\n", mknod("tmp", S_IFDIR | 0775, 0));
   // Only the superuser can execute the above line.
}
```

- EXAMPLE #2 (making a directory):

```
#include <stdio.h>
#include <stdlib.h>

void main(void){
char cmd[256];
   sprintf(cmd, "mkdir %s", "./TMP");
   printf("%d\n", system(cmd));
}
```

- **FCNTL**

  The fcntl() function performs controlling operations on the open file specified by the fd parameter.

  SYNOPSIS

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

  int fcntl (int fd,
            int request[
            int argument | struct flock *argument]);
  ```

  PARAMETERS

  fd - Specifies an open file descriptor obtained from a successful open(), fcntl(), or pipe() function.

  request - Specifies the operation to be performed.

  argument - Specifies a variable that depends on the value of the request parameter.

  NOTES:

1. The file locks set by the fcntl() and lockf() functions do not interact in any way with the file locks set by the flock()function. If a process sets an exclusive lock on a file using the fcntl() or lockf() function, the lock will not affect any process that is setting or clearing locks on the same file using the flock() function. It is therefore possible for an inconsistency to arise if a file is locked by different processes

using flock() and fcntl(). (The fcntl() and lockf() functions use the same mechanism for record locking.)

- EXAMPLE #1 (modifying a file):

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>              // CREAT
#include <sys/stat.h>           // CREAT
#include <sys/types.h>          // CREAT
#include <unistd.h>             // UNLINK

void main(void){
int  kbd, flags, fd;
char buffer1[256] = " hello1 ";
char buffer2[256] = " hello2 ";
char buffer3[256] = " hello3 ";
    // Create a file ( hello1 ).
    fd = open("/tmp/junk",O_WRONLY|O_CREAT, 0666);
    write(fd, buffer1, strlen(buffer1));
    close(fd);

    // Open the file for writing.
    fd = open("/tmp/junk",O_WRONLY|O_CREAT, 0666);

    // Get/record file status flags.
    flags = fcntl(fd, F_GETFL, 0);

    // Modify the file to allow appending.
    fcntl(fd, F_SETFL, flags | O_APPEND);

    // Append to the file ( hello1  hello2 ).
    write(fd, buffer2, strlen(buffer2));
    close(fd);

    // Open the file for writing.
    fd = open("/tmp/junk",O_WRONLY|O_CREAT, 0666);

    // Get/record file status flags.
    flags = fcntl(fd, F_GETFL, 0);

    // Modify the file to prohibit appending.
    fcntl(fd, F_SETFL, flags & ~O_APPEND);

    // Write to the file ( hello3  hello2 ).
    write(fd, buffer3, strlen(buffer3));
    close(fd);
}
```

- **ORDINARY FILES:**

  1. Most files on UNIX are "ordinary files". Ordinary files contain bytes of data organized into a linear array. Any byte or sequence of bytes can be read or written.

  2. It is not possible to insert bytes into the middle of an ordinary file (spreading it out). Bytes can only be added (appended) to the end of the file increasing its length.

  3. Nor is it possible to remove bytes from the middle (shrinking it). It is only possible to truncate the file to a length of 0 bytes. Ordinary files cannot be shrunk to any intermediate size.

     - Notice in the above example (modifying a file). Once we append on to the file (hello1 hello2) the file length remains fixed, even when we reopen the file for write with the append turned off (hello3 hello2).

     - The same characteristics holds true for directories (which are also ordinary files). Once a very large directory has been formed it will continue to gobble up disk space indefinitely. The only way to shrink the directory is to create a new directory, copy all of the files/links to it and destroy the old directory.

  4. Generally, when we want to change (shrink) a file's size, we just write a new version of the file and destroy the old one.

- **SPECIAL FILES:**

  1. "Special files" are device drivers and FIFOs. Special files only have a few rules (if any) to follow.

UNIX PROCESS-RELATED SYSTEM CALLS:

- **FORK**

  The fork()function creates a new (heavyweight) process (child process) that is identical to the calling process (parent process).

  The new process's instruction, user-data, and system-data segments are exact copies (almost) of the old process's.

  After fork returns, both processes (parent and child) receive a different return "signal". The child receives a 0, while the parent receives the PID of the child.

  SYNOPSIS

  ```
  #include <unistd.h>

  pid_t fork (void);
  ```

  PARAMETERS

  NOTES:

  1. Application developers may want to specify an #include statement for <sys/types.h> before the one for <unistd.h> if programs are being developed for multiple platforms.

  2. The child gets a copy of the parent's open file descriptors. Each is opened to the same file and the file pointer has the same value and is shared. So, if the child increments the file pointer (with lseek, for example) the parent will also be affected. However, the file descriptor is distinct. If either process closes the file, the other process still has access to it.

3. Even though the child's instruction, user-data, and system-data segments are copies of the parent's. They're just that copies and independent of each other.

4. Without EXE, fork is of little, if any, use.

- **EXEC**

  The exec functions replace the current process image with a new process image. The system constructs the new image from an executable file. Successful calls to the exec functions do not return because the system overlays the calling process with the new process.

  To run an executable file using one of the exec functions, applications include a function call such as the following:

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

  For forms of the exec functions that do not include the envp parameter, applications also define the environ variable to be a pointer to an array of character strings. The character strings define the environment in which the new process image runs. For example, the following shows how an application defines the environment variable:

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

  The environ array is terminated by a null pointer.

  SYNOPSIS

  ```
    #include <unistd.h>

    extern char **environ;
    int execl  (const char *path, const char *arg, ... );

    int execv  (const char *path, char * const argv[ ] );

    int execle (const char *path, const char *arg,
  ```

```
           ...
           char * const envp[ ] );

int execve (const char *path, char * const argv[ ],
               char * const envp[ ] );

int execlp (const char *file, const char *arg, ... );

int execvp (const char *file, char * const argv[ ] );
```

PARAMETERS

path - Points to a pathname identifying the new process image file.

arg... - Specifies a pointer to a null-terminated string, which is one argument available to the new process image. The first of these parameters points to the filename that is associated with the process being started by execl(), execle(), or execlp(). The last element in the list of arg parameters must be a null pointer.

argv - Specifies an array of character pointers to null-terminated strings, which are the arguments available to the new process image. The value in the argv[0] parameter points to the filename of the process being started by execv(), execve(), or execvp(). The last member of this array must be a null pointer.

envp - Specifies an array of character pointers to null-terminated strings, constituting the environment for the new process. This array must be terminated by a null pointer.

file - Identifies the new process image file. If this parameter points to a string containing a slash character, its contents are used as the absolute or relative pathname to the process image file. Otherwise, the system searches the directories specified in the PATH environment variable definition associated with the new process image to obtain a path prefix for the file.

NOTES:

1. The variations on exec provide different methods of passing arguments. The 3 primary differences are:

   a) Passing arguments via an array (handy when the number of arguments is not known at compile time) vs as a list.

   b) Searching for the program using a path (handy when the exact location of the file may not be known).

   c) Manually passing the environment (handy if the environment requires changing for the program) vs using the automatically passed environment.

   This is summarized in the following table:

	Argument format	Environment passing	path search
execl	list	auto	no
execv	array	auto	no
execle	list	manual	no
execve	array	manual	no
execlp	list	auto	yes
execvp	array	auto	yes

2. Since path searching and automatic environment passing are almost always wanted, execlp and execvp are the 2 most commonly used.

3. execlp and execvp can handle shell command files.

- **EXIT**

  The exit() function terminates the calling process after calling the _cleanup() function to flush any buffered output. Then it calls any functions registered previously for the process by the atexit() function, in the reverse order to that in which they were registered. In addition, the exit() function flushes all open output streams, closes all open streams, and removes all files created by the tmpfile() function. Finally, it calls the _exit() function, which completes process termination and does not return.

  The atexit() function registers functions to be called at normal process termination for cleanup processing. The function adds a single exit handler to a list of handlers to be called at process termination. The system calls the functions in reverse order, calling the function at the top of the list first. Any function that is registered more than once will be repeated.

  SYNOPSIS #1

  ```
  #include <stdlib.h>

  void exit(int status);

  int atexit(void (*function) (void));
  ```

  SYNOPSIS #2

  ```
  #include <unistd.h>

  void _exit(int status);
  ```

  PARAMETERS

status - Indicates the status of the process.

function - Points to a function that is called at normal process termination for cleanup processing. The number of exit handlers that can be specified with the atexit() function is limited by the amount of available virtual memory.

NOTES:

1. The exit call exit() is a misuse of the exit system call.

- **WAITPID**

  The wait() function suspends execution of the calling process until status information for one of its terminated child processes is available, or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process. If status information is available prior to the call to wait(), return is immediate.

  The waitpid() function behaves identically to wait(), if the process_id parameter has a value of (pid_t)-1 and the options parameter specifies a value of zero (0).

  SYNOPSIS

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

    pid_t wait(int *status_location);

    pid_t waitpid(
          pid_t process_id,
          int *status_location,
          int options);
  ```

  PARAMETERS

    process_id - Specifies the child process or set of child processes.

    status_location - Points to a location that contains the termination status (the exitcode) of the child process as defined in the <sys/wait.h> header file.

options - Modifies the behavior of the function. The flags for the options parameter are defined in the DESCRIPTION section.

NOTES:

1. Application developers may want to specify an #include statement for <sys/types.h> before the one for <sys/wait.h> if programs are being developed for multiple platforms.

- EXAMPLE #1 (forking a process):

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

void main(void){
int   exitstatus;
char  chr;
char  text[7];
char  message[24] = "This is a test.";
pid_t pid;
FILE  *fd;
   fd = fopen("tmp", "w");
   fprintf(fd, "HELLO\n");
   fclose(fd);
   printf("First message: %s\n", message);
   fd = fopen("tmp", "r");
   pid = fork();
   if (pid == 0) {
      printf("Child message #1: %s\n", message);
      strcpy(message, "Time for a change.");
      printf("Child message #2: %s\n", message);
      fscanf(fd, "%c", &chr);
      printf("Read by child: %c\n", chr);
      exit(0);
   } else {
      waitpid(pid, &exitstatus, 0);
      printf("Parent message #1: %s\n", message);
      strcpy(message, "Time for a nap.");
      printf("Parent message #2: %s\n", message);
      fscanf(fd, "%s", text);
      printf("Read by parent: %s\n", text);
   }
   fclose(fd);
}
```

```
First message: This is a test.
Child message #1: This is a test.
Child message #2: Time for a change.
Read by child: H                       <-- Read by child.
Parent message #1: This is a test.
Parent message #2: Time for a nap.
```

```
Read by parent: ELLO                    <-- Read by parent.
```

- **PTHREAD_CREATE**

  The pthread_create() function creates a new (lightweight) process (thread) that executes concurrently with the calling process (or thread). The new thread terminates either explicitly, by calling pthread_exit, or implicitly, by returning from the start_routine function.

  SYNOPSIS

  ```
  #include <pthread.h>

  int pthread_create(
                    pthread_t * threadID,
                    pthread_attr_t *attr,
                    void *(*start_routine)(void *),
                    void *arg);
  ```

  PARAMETERS

  threadID – Thread identifier.

  attr – The attributes to assign to the thread. The attributes include:

  - detachstate - Controls whether the thread is created in the joinable state (value PTHREAD_CREATE_JOINABLE) or in the detached state (PTHREAD_CREATE_DETACHED). The default value is PTHREAD_CREATE_JOINABLE.

  - schedpolicy - Selects the scheduling policy for the thread: one of SCHED_OTHER (regular, nonrealtime scheduling), SCHED_RR (realtime, roundrobin) or

SCHED_FIFO (realtime, firstin firstout). The default value is SCHED_OTHER.

- schedparam - Contains the scheduling parameters (essentially, the scheduling priority) for the thread. The default priority is 0.

- inheritsched - Indicates whether the scheduling policy and scheduling parameters for the newly created thread are determined by the values of the schedpolicy and schedparam attributes (value PTHREAD_EXPLICIT_SCHED) or are inherited from the parent thread (value PTHREAD_INHERIT_SCHED). The default value is PTHREAD_EXPLICIT_SCHED.

- scope - Defines the scheduling contention scope for the created thread. The only value supported in the LinuxThreads implementation is PTHREAD_SCOPE_SYSTEM, meaning that the threads contend for CPU time with all processes running on the machine. In particular, thread priorities are interpreted relative to the priorities of all other processes on the machine.

  The other value specified by the standard, PTHREAD_SCOPE_PROCESS, means that scheduling contention occurs only between the threads of the running process: thread priorities are interpreted relative to the priorities of the other threads of the process, regardless of the priorities of other processes. PTHREAD_SCOPE_PROCESS is not supported in LinuxThreads.

The default value is PTHREAD_SCOPE_SYSTEM.

(*start_routine)(void *) – The function to execute as a thread.

arg – First argument to pass to start_routine.

NOTES:

1. Attr can be NULL for which the created thread is joinable (not detached) and has default (non realtime) scheduling policy.

2. A thread can terminate either by calling pthread_exit, or implicitly, by returning from the start_routine function.

- **PTHREAD_EXIT**

  The pthread_exit() function terminates a thread.

  SYNOPSIS

  ```
  #include <pthread.h>

  void pthread_exit(void *retval);
  ```

  PARAMETERS

  retval – The return value of the thread. It can be consulted from another thread using pthread_join.

NOTES:

- **PTHREAD_JOIN**

  The pthread_join() suspends the execution of the calling thread until the thread identified (threadID) terminates, either by calling pthread_exit or by being cancelled.

  * Think of this sort of like a waitpid(), only for threads.

  SYNOPSIS

  ```
    #include <pthread.h>

    int pthread_join(
              pthread_t threadID,
              void **thread_return);
  ```

  PARAMETERS

  threadID – The thread identifier.

  thread_return – If thread_return is not NULL then the return value of threadID is stored in thread_return. The return value of threadID is either the argument it gave to pthread_exit, or PTHREAD_CANCELED if threadID was cancelled.

NOTES:

1. The joined thread threadID must be in the joinable state: it must not have been detached using pthread_detach or the PTHREAD_CREATE_DETACHED attribute to pthread_create.

2. When a joinable thread terminates its memory resources (thread descriptor and stack) are not deallocated until another thread performs pthread_join on it. Therefore, pthread_join

must be called once for each joinable thread created to avoid memory leaks.

3. At most one thread can wait for the termination of a given thread. Calling pthread_join on a thread threadID on which another thread is already waiting for termination returns an error.

- **PTHREAD_DETACH**

  The pthread_detach() function puts the thread threadID in the detached state. This guarantees that the memory resources consumed by threadID will be freed immediately when threadID terminates. However, this prevents other threads from synchronizing on the termination of threadID using pthread_join.

  SYNOPSIS

  ```
  #include <pthread.h>

  int pthread_detach(pthread_t threadID);
  ```

  PARAMETERS

  threadID – The thread identifier.

NOTES:

1. A thread can be created initially in the detached state, using the detachstate attribute to pthread_create. In contrast, pthread_detach applies to threads created in the joinable state, and which need to be put in the detached state later.

- **NICE**

  The nice() function adds the value specified in the increment parameter to the nice value of the calling process. The nice value is a nonnegative number; a higher nice value gives the process a lower CPU priority.

  SYNOPSIS

  ```
  #include <unistd.h>

  int nice(int increment);
  ```

  PARAMETERS

  increment - Specifies a value that is added to the current process priority. You can specify a negative value.

  NOTES:

1. When you are using the Standard C Library version of the nice() function, the maximum nice value for a process is 39 (2 * {NZERO} -1) and the minimum is 0 (zero). Requests for values outside these limits result in the nice value being set to the corresponding limit.

2. Any process can lower its priority (numerically raise its nice value). A process must have superuser privileges to raise its priority (numerically lower its nice value).

UNIX IPC-RELATED SYSTEM CALLS:

- **PIPE**

The pipe() function creates a unidirectional interprocess channel called a pipe, and returns two file descriptors, fd[0] and fd[1]. The file descriptor specified by the fd[0] parameter is opened for reading and the file descriptor specified by the fd[1] parameter is opened for writing. Their integer values will be the two lowest available at the time of the call to the pipe() function.

A process has the pipe open for reading if it has a file descriptor open that refers to the read end, fd[0]. A process has the pipe open for writing if it has a file descriptor open that refers to the write end, fd[1]. A read on file descriptor fd[0] accesses the data written to filedes[1] on a first-in, first-out (FIFO) basis.

SYNOPSIS

```
#include <unistd.h>

int pipe (int fd[2]);
```

PARAMETERS

fd - Specifies the address of an array of two integers into which the new file descriptors are placed.

NOTES:

Commands used with pipe include:

1. write - writes data to a pipe. If the pipe becomes full, write is blocked until data is removed with a read. All writes are atomic and there are no EOL or EOF markers.

2. read - reads data from a pipe. Data ordering is maintained. If the pipe is empty read is blocked until data is written.

3. close - closes the pipe (must close both ends). When the write end is closed it also acts as an EOF marker. You should always close the write end first.

- EXAMPLE #1 (a piping process):

```
#include <unistd.h>
#include <stdio.h>

void main(void) {
int pfd[2];
char text[100];

   // Create a pipe.
   pipe(pfd);

   // Write to the pipe.
   write(pfd[1], "hello", 6);

   // Close write end.
   close(pfd[1]);

   // Read from the pipe.
   read(pfd[0], text, sizeof(text));
   printf("%s\n", text);

   // Close read end.
   close(pfd[0]);
}
```

- EXAMPLE #2 (piping to a process):

```
// PARENT PROCESS.
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

void main(void) {
int  PID1;
int  exitCode1;
int  pfd[2];
char fdstr[10];
   pipe(pfd);
   PID1 = fork();
   if (PID1 == 0) {
      sprintf(fdstr, "%d", pfd[0]);
      execlp("File1", fdstr, NULL);
   }
   if (PID1 != 0) {
      wait(&exitCode1);
   }
}

// CHILD PROCESS.
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void main(int argc, char *argv[]) {
int  i;
int  fd;
char chr[150];
   fd = atoi(argv[0]);
   read(fd, chr, 150);
   close(fd);
}
```

- In the above example, I piped a block of data. It is also possible to pipe 1 data item at a time.

- **DUP**

  The dup() function duplicates an existing file descriptor, giving a new file descriptor that is open to the same file or pipe. The two file descriptors share the same file pointer.

  The dup() function always uses the lowest numbered file descriptor avialable.

  SYNOPSIS

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

    int dup(int fd );
  ```

  PARAMETERS

  fd - Specifies an open file descriptor obtained from a successful open(), fcntl(), or pipe() function.

  NOTES:

  1. The only advantage to having another file descriptor is that its number may be better suited to the purpose. For example, it is possible to dup to redirect the standard input to instead read from a pipe.

  2. In some versions of UNIX, the fcntl() function could be used to force the use of a specific file descriptor:

     ```
             close(6);
             newfd = fcntl(fd, F_DUPFD, 6);
     ```

The above would close 6 (just in case it was open) and force newfd to be a duplicate of fd using file descriptor 6.

- EXAMPLE #1 (piping cat via dup):

```
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

void main(void) {
int  newfd;
int  PID1;
int  pfd[2];
   pipe(pfd);                 <-- uses fd 3 and fd 4.
   PID1 = fork();
   if (PID1 == 0) {
      close(0);               <-- closes std-input
      newfd = dup(pfd[0]);    <-- dups fd 3 as fd 0.
      close(pfd[0]);          <-- closes read side.
      close(pfd[1]);          <-- closes write side.
     execlp("cat", "cat", NULL);
   }
   if (PID1 != 0) {
      close(pfd[0]);          <-- closes read side.
      write(pfd[1], "hello", 6);
      close(pfd[1]);          <-- closes write side.
   }
}
```

- Generally, the process would be:

  1. Call dup() repeatedly until the desired file descriptor is returned.

  2. Close the file descriptors that were opened, but not needed.

- **BI-DIRECTIONAL PIPES**

  Earlier I stated that "the pipe() function creates a unidirectional interprocess channel called a pipe". I lied. Actually, the pipe() function creates a bi-directional interprocess channel.

  There are many times when a bi-directional pipe is needed, or at least would be handy. For example, assume that we want to sort some data that we have generated. We could:

  1. Write our own sort routine. But, this "goes against the grain" when it come to UNIX (after all, the idea behind UNIX was to be able to use existing programs to produce a larger program. Not to start all over each time.).

  2. Call the UNIX sort routine with a system call like: system("sort < data > output") and then read in the output file.

  3. Use a pipe to pipe the data to sort and then back into our program.

- EXAMPLE #1 (a bi-directional piping process):

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

void main(void) {
int  fd, nread;
int  PID;
int  pfd[2];
char buf[512];
   pipe(pfd);
   PID1 = fork();

   if (PID1 == 0) {
      close(0);
      dup(pfd[0]);
      close(1);
      dup(pfd[1]);
      close(pfd[0]);
      close(pfd[1]);
      execlp("sort", "sort", NULL);
   }

   if (PID1 != 0) {
     fd = open("/tmp/junk", O_RDONLY);
      while ((nread = read(fd, buf, 512)) != 0) {
         write(pfd[1], buf, nread);
     }
     close(fd);
     close(pfd[1]);
     while (nread = read(pfd[0], buf, 512)) != 0) {
         write(1, buf, nread);
     }
     close(pfd[0]);
   }
}
```

- The above program compiles and runs (for a while) then it hangs. It also does not produce the expected results. What's wrong?

- The previous example has several problems, including:

  1. What's to prevent the parent from reading its own data back from the pipe (before the sort routine could process the data)? Remember <u>any</u> read will remove data from the pipe.

  2. How does each process know when to close their end of the pipe? If either process closes its end too soon we end up with a deadlock situation. (Basically, any process-pair that tries to use one bi-directional pipe for two-way communication will deadlock.)

- On possible solution would be to use semaphores. However:

  1. Semaphores were not implemented in many early versions of UNIX. So your code may not be very portable.

  2. Semaphores implemented via file locking would work on any version of UNIX, but they're inefficient.

- Another possible solution is to use 2 one-way pipes. One pipe for > traffic and one pipe for < traffic. This solution will work with any version of UNIX and it the most efficient. However, deadlock can still occur:

  1. The most likely is that the parent can fill up the ">" pipe and be forced to wait (for the child to read data from the pipe). However, at the same time it is possible for the child to fill up the "<" pipe and be forced to wait (for the parent to read data from the pipe). Both processes are now in a state of deadlock.

  2. Other causes of deadlock are also possible. In each case (program) the programmer has to consider all possible

interactions between the 2 processes in an effort to locate and prevent any deadlock situation.

- EXAMPLE #1 (a bi-directional piping process):

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

void main(void) {
int  fd, nread;
int  PID;
int  pfdIn[2], pfdOut[2];
char buf[512];
   pipe(pfdIn);
   pipe(pfdOut);
   PID1 = fork();

   if (PID1 == 0) {
      close(0);
      dup(pfdOut[0]);
      close(1);
      dup(pfdIn[1]);
      close(pfdOut[0]);
      close(pfdOut[1]);
      close(pfdIn[0]);
      close(pfdIn[1]);
      execlp("sort", “sort”, NULL);
   }

   if (PID1 != 0) {
      close(pfdOut[0]);
      close(pfdIn[1]);
      fd = open(“/tmp/junk”, O_RDONLY);
      while ((nread = read(fd, buf, 512)) != 0) {
         write(pfdOut[1], buf, nread);
     }
     close(fd);
     close(pfdOut[1]);
     while (nread = read(pfdIn[0], buf, 512))!=0) {
         write(1, buf, nread);
     }
     close(pfdIn[0]);
   }
}
```

- **FIFOS**

  - A FIFO has features of a file and a pipe. Like a file (We use mknod to create a FIFO.) it has a name, can be opened for read or write, and <u>any</u> process can use the FIFO (not just the processes that inherited the file handles). However, once opened the FIFO acts like a pipe.

  - An obvious use for a FIFO is a replacement for a pipe. However, this may not be as good an idea as one might expect. FIFOs are disk files, which require more overhead than a pipe and the FIFO name may not be unique (name clashing) unless one uses the system function tmpnam (or tempnam).

    - ❖ FIFOs were added to replace messages (esp. messages between multiple (> 2) processes – like in databases) in UNIX not to replace pipes.

- **SEMAPHORES**

  - A semaphore is a mechanism that prevents multiple processes from accessing the same resource simultaneously. Semaphores can be implemented using file locking, FIFOs, pipes, and in some cases built-in functions. The consensus is that semaphores are not easy to use.

  - Semaphore system functions include: semget, semop, and semctl.

- **SEMGET**

  The semget() function returns (and possibly creates) a semaphore ID. The system defines sets of semaphores in a system-wide table, with each set being an entry in the table. The semget() function returns an ID that identifies the semaphore set's entry in the table. You determine which semaphore set's ID is returned by specifying the key parameter.

SYNOPSIS

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

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

PARAMETERS

key - Specifies the key that identifies the semaphore set. The value for the key parameter can be IPC_PRIVATE or a random number other than 0 (zero). To ensure that you receive a new, unused entry in the semaphore table, specify IPC_PRIVATE as the value of key.

nsems - Specifies the number of semaphores to create in the semaphore set.

semflg - Specifies the creation flags. Possible values are as follows:

IPC_CREAT If the key specified does not exist, the semget function creates a semaphore ID using the specified key.

IPC_EXCL Specifies that you want exclusive access to the semaphore set.

- **SEMOP**

The semop() function performs operations on the semaphores in the specified semaphore set. The semaphore operations are defined in the sops array. The sops array contains nsops elements, each of which is represented by a sembuf structure.

SYNOPSIS

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

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

PARAMETERS

semid - Specifies the ID of the semaphore set.

sops - Points to the user-defined array of sembuf structures that contain the semaphore operations.

nsops - The number of sembuf structures in the array.

NOTES:

1. The sembuf structure (from sys/sem.h) is shown here:

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

   sem_num - Specifies an individual semaphore within the semaphore set.

sem_op - Specifies the operation to perform on the semaphore. The sem_op operation is specified as a negative integer, a positive integer, or 0 (zero).

sem_flg - Specifies various flags for the operations.

- **SEMCTL**

The semctl() function allows a process to perform various operations on an individual semaphore within a semaphore set, on all semaphores within a semaphore set, and on the semid_ds structure associated with the semaphore set.

SYNOPSIS

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

  int semctl(int semid, int semnum, int cmd, ...);
```

PARAMETERS

semid - Specifies the ID of the semaphore set.

semnum - Specifies the number of the semaphore to be processed.

cmd - Specifies the type of command.

The fourth argument is optional and depends on the operation requested. If required, it is of the type union semun, which the application program must explicitly declare as follows:

```
    union semun {
        int val;
        struct semid_ds *buf;
        u_short *array;
    } arg );
```

NOTES:

The cmd value determines which operation is performed. The following commands operate on the specified semaphore within the semaphore set identified by semid: GETVAL, SETVAL, GETNCNT, GETZCNT, GETALL, SETALL, and various IPC commands.

- **EXAMPLE – SIMPLE SEMAPHORE USE:**

```
// Seminit - Semaphore initialization.

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#include <stdio.h>
/*
KEY is an arbitrary long integer which serves as an
external identifier for any program that wishes to use it.
*/
#define KEY (1492)

void main(){
   int id;
   union semun {
     int val;
     struct semid_ds *buf;
     ushort * array;
   } argument;
   argument.val = 0;

   id = semget(KEY, 1, 0666 | IPC_CREAT);
   if (id < 0) {
      fprintf(stderr, "Unable to obtain semaphore.\n");
      exit(0);
   }
   if (semctl(id, 0, SETVAL, argument) < 0) {
      fprintf(stderr, "Cannot set semaphore value.\n");
   } else {
      fprintf(stderr, "Semaphore %d initialized.\n", KEY);
   }
}

// SemA - Semaphore program A.

// SemA and SemB may be initiated at any time, but SemB
// will be forced to wait until SemA is executed. SemA and
// SemB do not have to be executed by the same user!

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

```
#define KEY (1492)

void main(){
   int id;
   struct sembuf operations[1];
   int retval;

   id = semget(KEY, 1, 0666);
   if (id < 0) {
      fprintf(stderr, "Cannot find semaphore, exiting.\n");
      exit(0);
   }

   /* Do a semaphore V-operation (a release operation). */
   operations[0].sem_num = 0;
   operations[0].sem_op = 1;
   operations[0].sem_flg = 0;
   retval = semop(id, operations, 1);
   if (retval == 0) {
      printf("Successful V-operation by program SemA.\n");
   } else {
      printf("SemA: V-operation did not succeed.\n");
      perror("REASON");
   }
}

/*
Think carefully about what the V-operation does. If sema is
executed twice, then semb can execute twice.
*/

// SemB - Semaphore program B.

// SemA and SemB may be initiated at any time, but SemB
// will be forced to wait until SemA is executed. SemA and
// SemB do not have to be executed by the same user!

// HOW TO TEST:
//  Execute SemB & - The & is important - otherwise you
//  would have have to move to a different terminal to
//  execute SemA. Then execute SemA.

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

```
#define KEY (1492)

void main()
{
   int id;
   struct sembuf operations[1];
   int retval;

   id = semget(KEY, 1, 0666);
   if (id < 0) {
      fprintf(stderr, "Cannot find semaphore, exiting.\n");
      exit(0);
   }

   /* Do a semaphore P-operation (an acquire operation). */
   operations[0].sem_num = 0;
   operations[0].sem_op = -1;
   operations[0].sem_flg = 0;
   retval = semop(id, operations, 1);
   if (retval == 0) {
      printf("Successful P-operation by program SemB.\n");
      printf("Process id is %d\n", getpid());
   } else {
      printf("SemB: P-operation did not succeed.\n");
   }
}
```

- **SHARED MEMORY (System V):**

  The fastest way to send information. Memory (called segments) can be shared between any number of processes. Any process can access multiple shared memory segments. Typically, semaphores are used in conjunction with shared memory to prevent one process from over writing data being written by another process.

  A shared memory segment is first created outside of any process's space. Then any process that wants to use the shared memory segment makes a system call to map the shared segment into its own address space.

  On modern computers there are several memory segmentation registers. Using these registers make shared memory accesses very fast, just as fast as normal memory accesses.

  Because of hardware differences, it is best to only map one shared segment at a time (should be portable this way). There are also memory segment size restrictions that vary from platform to platform.

  Shared memory system functions include: shmget, *shmat, shmdt, and shmctl.

- **SHMGET**

The shmget() function returns (and possibly creates) the ID for the shared memory region identified by the key parameter. If IPC_PRIVATE is used for the key parameter, the shmget() function returns the ID for a private (that is, newly created) shared memory region. The flags parameter supplies creation options for the shmget() function. If the key parameter does not already exist, the IPC_CREAT flag instructs the shmget() function to create a new shared memory region for the key and return the kernel-assigned ID

SYNOPSIS

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

    int shmget(key_t key, size_t size, int flags);
```

PARAMETERS

key - Specifies the key that identifies the shared memory region. The value for the key parameter can be IPC_PRIVATE or a random number other than zero (0). If the value of key is IPC_PRIVATE, it can be used to assure the return of a new, unused shared memory region.

size - Specifies the minimum number of bytes to allocate for the region.

flags - Specifies the creation flags. Possible values are: IPC_CREAT and IPC_EXCL.

- **SHMAT**

The shmat() function attaches the shared memory region identified by the shmid parameter to the virtual address space of the calling process. For the addr parameter, the process can specify either an explicit address or 0 (zero), to have the kernel select the address. If an explicit address is used, the process can set the SHM_RND flag to have the kernel round off the address, if necessary.

SYNOPSIS

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

    void *shmat(int shmid, const void *shmaddr,  int shmflg);
```

PARAMETERS

shmid - Specifies the ID for the shared memory region. The ID is typically returned by a previous shmget() function.

addr - Specifies the virtual address at which the process wants to attach the shared memory region. The process can also specify 0 (zero) to have the kernel select an appropriate address.

flags - Specifies the attach flags. Possible values are:

SHM_RND - If the addr parameter is not 0 (zero), the kernel rounds off the address, if necessary.

SHM_RDONLY - If the calling process has read permission, the kernel attaches the region for reading only.

- **SHMDT**

The shmdt() function detaches the shared memory region at the address specified by the addr parameter.

SYNOPSIS

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

int shmdt(const void *addr);
```

PARAMETERS

addr - Specifies the starting virtual address for the shared memory region to be detached. This is the address returned by a previous shmat() call.

- **SHMCTL**

The shmctl() function allows a process to query or set the contents of the shmid_ds structure associated with the specified shared memory region ID. It also allows a process to remove the shared memory region's ID and its
associated shmid_ds structure.

SYNOPSIS

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

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

PARAMETERS

shmid - Specifies the ID of the shared memory region.

cmd - Specifies the type of command. The possible commands and the operations they perform are as follows:

IPC_STAT Queries the shared memory region ID by copying the contents of its associated shmid_ds data structure into the buf structure.

IPC_SET - Sets the shared memory region ID by copying values found in the buf structure into corresponding fields in the shmid_ds structure associated with the shared memory region ID.

IPC_RMID Removes the shared memory region ID and deallocates its associated shmid_ds structure.

buf - Specifies the address of a shmid_ds structure. This structure is used only with the IPC_STAT and IPC_SET commands.

- **EXAMPLE - Sharing memory:**

```
// Program one (source).

#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define KEY 1959

void main(void){
int  i;
int  memID;
char *data = "Hello";
void *shmem;

   // Create a block of shared memory.
   memID = shmget( KEY, 10, 0666 | IPC_CREAT );

   // Attach shared memory to shmem pointer.
   shmem = shmat( memID, 0, 0 );

   // Write data to shared memory.
   bcopy(data, (char*)shmem, strlen(data));

   // Detach shmem from shared memory region.
   shmdt(shmem);

   // Delay.
   scanf("%d", &i);

   // Delete shared memory block.
   shmctl( memID, IPC_RMID, NULL );

}


// Program one (destination).

#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
```

```
#include <sys/shm.h>
#include <string.h>

#define KEY 1959

void main(void){
int  memID;
char *data;
void *shmem;

   // Create a block of shared memory.
   memID = shmget( KEY, 10, 0666);

   // Attach shared memory to shmem pointer.
   shmem = shmat( memID, 0, 0 );

   // Read data from shared memory.
   data = new char;
   bcopy((char *)shmem, data, 6);

   // Display shared memory contents.
   printf("%s\n", data);

   // Detach shmem from shared memory region.
   shmdt(shmem);

   // Delete shared memory block.
   shmctl( memID, IPC_RMID, NULL );
}
```

- **SHARED MEMORY (Posix):**

  The fastest way to send information. Memory (called segments) can be shared between any number of processes. Any process can access multiple shared memory segments. Typically, semaphores are used in conjunction with shared memory to prevent one process from over writing data being written by another process.

  A shared memory segment is first created outside of any process's space. Then any process that wants to use the shared memory segment makes a system call to map the shared segment into its own address space.

  On modern computers there are several memory segmentation registers. Using these registers make shared memory accesses very fast, just as fast as normal memory accesses.

  Because of hardware differences, it is best to only map one shared segment at a time (should be portable this way). There are also memory segment size restrictions that vary from platform to platform.

  Shared memory system functions include: shm_open, shm_unlink,

- **SHM_OPEN**

shm_open() creates and opens a new, or opens an existing, POSIX shared memory object. A POSIX shared memory object is in effect a handle which can be used by unrelated processes to mmap the same region of shared memory. The shm_unlink() function performs the converse operation, removing an object previously created by shm_open(). Close() should be used when the file descriptor is no longer needed.

SYNOPSIS

```
 #include <sys/mman.h>
#include <sys/stat.h>             /* For mode constants */
#include <fcntl.h>                /* For O_* constants */

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

int shm_unlink(const char *name);

Link with -lrt.
```

PARAMETERS

name - The operation of shm_open() is analogous to that of open) and name specifies the shared memory object to be created or opened (i.e. filename).

oflag - A bit mask created by ORing together exactly one of O_RDONLY or O_RDWR and any of the other flags listed allowed (see man page).

mode - A bit mask created by ORing together exactly one of O_RDONLY or O_RDWR.

- **FTRUNCATE**

The ftruncate() function cause the regular file named by path or referenced by fd to be truncated to a size of precisely length bytes. If the file previously was larger than this size, the extra data is lost. If the file previously was shorter, it is extended, and the extended part reads as null bytes ('\0').

SYNOPSIS

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

int ftruncate(int *fd*, off_t *length*);

PARAMETERS

fd - Specifies the ID (i.e. filename) for the shared memory region.

length - Specifies the length (in bytes) of the memory region.

- **MMAP**

mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in addr. The length argument specifies the length of the mapping. Munmap unmaps the shared memory object from the virtual address space of the calling process.

SYNOPSIS

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

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

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

PARAMETERS

addr - If addr is NULL, then the kernel chooses the address at which to create the mapping; this is the most portable method of creating a new mapping.

length - Specifies the length (in bytes) of the memory region.

prot - Describes the desired memory protection of the mapping (and must not conflict with the open mode of the file).

flags - Determines whether updates to the mapping are visible to other processes mapping the same region, and whether updates are carried through to the underlying file.

fd - Specifies the ID (i.e. filename) for the shared memory region.

Offset - Must be a multiple of the page size as returned by sysconf(_SC_PAGE_SIZE). But 0 seems to work too ☺

- **EXAMPLE - Sharing memory:**

```
// Program one.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types>
#include <fcntl.h>

int main(void) {
int i;
int data = 1492;
int memID;
void *memAD;
mode_t oldMask = umask(0);
   // Acquire shared memory (Posix style).
   memID = shm_open("/myMemory", O_CREAT | O_RDWR,
                    S_IRUSR | S_IWUSR);
   if (memID < 0) {
      printf("Error - cannot acquire shared memory
              [%d].\n", errno);
      exit(1);
   }
   if (ftruncate(memID, sizeof(data)) == -1) {
      printf("Error - cannot set shared memory size
              [%d].\n", errno);
      exit(1);
   }
   // Map memory to file.
   memAD = mmap(NULL, sizeof(data), PROT_READ |
                   PROT_WRITE, MAP_SHARED, memID, 0);

   if (memAD  == MAP_FAILED) {
      printf("Error - cannot map shared memory size
              [%d].\n", errno);
      exit(1);
   }
   // Map data to memory.
   memcpy(memAD, &data, sizeof(data));

   printf("Enter 0 to continue ");
   scanf("%d", &i);
   // Map memory to data.
   memcpy(&data, memAD, sizeof(data));
   for (i = 0; i < 10; i++) {
       printf("%d\n", data);
   }

   // Clean up.
   munmap(memAD, sizeof(data));
   close(memID);
   umask(oldMask);
   return 1;
}

// Program two.

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types>
#include <fcntl.h>

int main(void) {
int i;
int data;
int memID;
void *memAD;
mode_t oldMask = umask(0);
   // Acquire shared memory (Posix style).
   memID = shm_open("/myMemory", O_CREAT | O_RDWR,
                    S_IRUSR | S_IWUSR);
   if (memID < 0) {
      printf("Error - cannot acquire shared memory
             [%d].\n", errno);
      exit(1);
   }
   if (ftruncate(memID, sizeof(data)) == -1) {
      printf("Error - cannot set shared memory size
             [%d].\n", errno);
      exit(1);
   }
   // Map memory to file.
   memAD = mmap(NULL, sizeof(data), PROT_READ |
                 PROT_WRITE, MAP_SHARED, memID, 0);
   if (memAD  == MAP_FAILED) {
      printf("Error - cannot map shared memory size
             [%d].\n", errno);
      exit(1);
   }
   // Map memory to data.
   memcpy(&data, memAD, sizeof(data));
   for (i = 0; i < 10; i++) {
       printf("%d\n", data);
   }
   printf("Enter 0 to continue ");
   scanf("%d", &i);

   // Map data to memory.
   data = 42;
   memcpy(memAD, &data, sizeof(data));

   // Clean up.
   munmap(memAD, sizeof(data));
   close(memID);
   umask(oldMask);
   return 1;
}
```

- **SOCKETS (example):**

```
/* A simple server in the internet domain using TCP
   The port number is passed as an argument */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(int argc, char *argv[]) {
int    i, credits;
int    SocketOut, SocketIn, PortNo, n;
char   buffer[256];
socklen_t Client;
struct sockaddr_in ServerAddress, ClientAddress;
     // Trap missing command line argument.
     if (argc < 2) {
        printf("ERROR, no port provided.\n");
   exit(1);
}
// Open socket.
SocketOut = socket(AF_INET, SOCK_STREAM, 0);
if (SocketOut < 0) {
   printf("ERROR opening socket.\n");
}
// Define connection.
bzero((char *) &ServerAddress,
      sizeof(ServerAddress));
PortNo = atoi(argv[1]);
ServerAddress.sin_family = AF_INET;
ServerAddress.sin_addr.s_addr = INADDR_ANY;
ServerAddress.sin_port = htons(PortNo);
if (bind(SocketOut, (struct sockaddr *)
    &ServerAddress, sizeof(ServerAddress)) < 0) {
     printf("ERROR on binding.\n");
}
// Listen.
listen(SocketOut, 5);
Client = sizeof(ClientAddress);
SocketIn = accept(SocketOut, (struct sockaddr *)
                  &ClientAddress, &Client);
if (SocketIn < 0) {
     printf("ERROR on accept.\n");
}
// Get credits from client.
bzero(buffer, 256);
n = read(SocketIn, buffer, 255);
if (n < 0) printf("ERROR reading from socket.\n");
credits = atoi(buffer);
printf("[Server] Client requests %d credits.\n",
        credits);
// Send reply to client.
for (i = 0; i < credits; i++) {
     n = write(SocketIn, "Got your message.",
               strlen("Got your message."));
    if (n < 0) printf("ERROR writing to socket.\n");
}
// Send termination reply to client.
n = write(SocketIn, "Terminate.",
          strlen("Terminate."));
if (n < 0) {
    printf("ERROR writing to socket.\n");
}
// Close sockets.
close(SocketIn);
```

```
     close(SocketOut);
     return 0;
}



// Client code

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

int main(int argc, char *argv[]) {
int i;
int    Socket, PortNo, cnt;
char   buffer[256];
struct sockaddr_in ServerAddress;
struct hostent *Server;
    // Trap missing command line argument.
    if (argc < 3) {
       printf("ERROR, usage %s hostname port.\n",
              argv[0]);
       exit(0);
    }
    // Open socket.
    PortNo = atoi(argv[2]);
    Socket = socket(AF_INET, SOCK_STREAM, 0);
    if (Socket < 0) {
       printf("ERROR opening socket.\n");
    }
    // Determine host.
    Server = gethostbyname(argv[1]);
    if (Server == NULL) {
        printf("ERROR, no such host.\n");
        exit(0);
    }
    // Connect to socket.
    bzero((char *) &ServerAddress,
          sizeof(ServerAddress));
    ServerAddress.sin_family = AF_INET;
    bcopy((char *)Server->h_addr, (char
          *)&ServerAddress.sin_addr.s_addr, Server-
          >h_length);
    ServerAddress.sin_port = htons(PortNo);
   if (connect(Socket,(struct sockaddr *)
       &ServerAddress, sizeof(ServerAddress)) < 0) {
        printf("ERROR connecting");
    }

    // Send 50 credits.
    strcpy(buffer, "50");
    cnt = write(Socket, buffer, strlen(buffer));
    if (cnt < 0) printf("ERROR writing to socket.\n");
    // Wait for replied data.
   for (i = 0; i < 55; i++) {
       bzero(buffer, 256);
       cnt = read(Socket, buffer, 255);
       if (cnt < 0) printf("ERROR reading from
                            socket.\n");
       printf("From server: %s\n", buffer);
       if (!strcmp(buffer, "Terminate.")) break;
       // Send ack.
       strcpy(buffer, "ack");
       cnt = write(Socket, buffer, strlen(buffer));
       if (cnt < 0) printf("ERROR writing to socket.\n");
    }
    // Close sockets.
    close(Socket);
    return 0;
}
```