Preview

- lseek() System call
- pread() pwrite()
- sync(), fsync()
- File copy
- Command Line Argument
- File Sharing
- dup() and dup2() System Cal
- sat(), fsat(), lsat() system Call
- ID's for a process
- File Access permission
- access() System Call
- umask() System Call

lseek System Calls

- Every open file has an associated with **current file offset** which is saved in a file table.
- It is non negative integer that measures the number of bytes from the beginning of the file.
- When a file is opened, offset is set to 0.
- Read/write operation start at current file offset and cause the offset to be incremented by the number of bytes read or write.

lseek System Calls

- An opened file offset can be explicitly positioned by calling lseek system call.

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

- Returns the offset of the pointer (in bytes) from the *beginning* of the file. If the return value is -1, then there was an error moving the pointer.

lseek System Calls

- The interpretation of the *offset* depends on the value of the *whence* argument.
  - **SEEK_SET**: offset is set to offset bytes from the beginning of the file
  - **SEEK_CUR**: offset is set to its current value plus the offset
  - **SEEK_END**: set to the size of the file plus the offset


```
/*  program creatHole.c creat a file with a hole in it */
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

char buf1[]="abcdefghij";
char buf2[]="ABCDEFGHIJ";
/* err_sys display error message and exit  */
void err_sys(char *str)
{
   printf ("%s",str);
   exit (1);
}
#define FILE_MODE (S_IRUSR | S_IWUSR|S_IRGRP|S_IROTH)
int main()
{
   int filedes;

   if ((filedes = creat("filehole.txt", FILE_MODE)) <0)
      err_sys("Creat File Open Error");
   if (write (filedes, buf1, 10) != 10)
      err_sys("Creat Write Error"); /*now offset = 10*/
   if (lseek (filedes, 40, SEEK_SET)== -1)
      err_sys("Creat seek1 Error");/*now offset =40 */
   if (write(filedes, buf2, 10) != 10)
      err_sys("Creat write Error"); /*now offset = 50 */
   return 0;
}
```

od –c filename

pread() and pwrite() system call

```
#include <unistd.h>
ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);
                                  Returns: size of byte read, 0 if end of file, -1 on error
ssize_t pwriteint fildes, void *buf, size_t nbyte, off_t offset);
                                  Returns: size of byte read, 0 if end of file, -1 on error
```

- Calling pread() (pwrite()) is equivalent to calling lseek() followed by a call to read() (write()) with following exceptions.
  - There is no way to interrupt the two operations that occur when we call pread() (pwrite())
  - The current file offset is not updated

pread() and pwrite() system call

```
//pwrite.c shows example for pread() and pwrite()
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/stat.h>
#include <sys/types.h>

int main()
{
        int fd, nr, nr2, nw, nw2;
        char buf_wr[]={"This is First message to write"};
        char buf_wr2[]={"This is second message to write"};
        char buf_rd[120];

        //open file
        fd = open("out.txt", O_RDWR|O_CREAT, 0666);
        // write first message to the file
        nw = pwrite(fd, &buf_wr, strlen(buf_wr), 0);
        // write the second message just after the first message to the file
        nw2= pwrite(fd, &buf_wr2, strlen(buf_wr2), strlen(buf_wr));
        // read the second message from the file
        nr = pread(fd, &buf_rd, sizeof(buf_rd), strlen(buf_wr));
        printf("read from file: %s",buf_rd);

        close(fd);

        return 0;
```


sync(), fsync() and fdatasync() System calls

- Traditional Unix system maintains a buffer cache (or page cache) in the kernel's space.
- When we write data to a file, the data is normally copied into the buffer cache and queued for writing to disk at some later time. (delayed write). The kernel eventually writes all the delayed-write blocks to disk, normally when it needs to reuse the buffer for some other disk block.
- sync(), fsync(), and fdatasync() system calls are provided to ensure consistence of file system on disk with the contents of buffer cache.
- The function sync()is normally called periodically (usually every 30 seconds) from a system daemon, often called update. This guarantees regular flushing of the kernel's block buffers.

sync(), fsync() and fdatasync() System calls

```
#include <unistd.h>
int fsync(int fildes);
int fdatasync (int filedes);
void sync (void);
```

Returns: 0 if ok, -1 on error

- The fsync() refers only to a single file, specified by the file descriptor filedes, and waits for the disk writes to complete before returning.
- This function is used when an application, such as a database, needs to be sure that the modified blocks have been written to the disk.
- The fdatasync() is similar to fsync(), but it affects only the data portions of a file. With fsync(), the file's attributes are also updated synchronously

sync(), fsync() and fdatasync() System calls

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

int main()
{
  char wbuffer[] = "1234567890";
  char rbuffer[100];
  int fd;

  /* Open the file */
  fd = open ("test.txt", O_RDWR | O_CREAT | O_TRUNC);
  /* Write 10 bytes of data */
  write (fd, (void *) wbuffer, 10);
/* and make sure it's written */
  fsync (fd);
  /* Seek the beginning of the file */
  lseek (fd, 0, SEEK_SET);
  /* Read 10 bytes of data */
  read (fd, (void *) rbuffer, 10);
  /* Terminate the data we've read with a null character */
  rbuffer[10] = '\0';
  printf ("String read = %s.\n", rbuffer);
  close (fd);
  return 0;
}
```


A File Copy

- Ways to copy a file
  - We can read contents of a file a byte base and write to a file, or
  - We can read contents of a file and save into a buffer and write to a file

A File Copy by a Byte by Byte

```
/*copyfile.c: read a character by character from a copyfile.c
  and write to out.txt file */
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>

int main()
{
   int InFileDes, OutFileDes;
   char Ach;
   /* open file for read only */
   InFileDes = open("copyfile.c", O_RDONLY);
   /* open file for write only or create only, created file mode will rw_ _ _ _ _ _
_*/
   OutFileDes = open ("output.txt", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
   while (read(InFileDes, &Ach, 1) == 1)
      write(OutFileDes, &Ach, 1);
   close (InFileDes);
   close (OutFileDes);
   exit (0);
}
```

```c
/*copyfile2.c: read a character by character from two
files and write to updown.txt */

#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
int main()
{
   int InFileDes, InFileDes1,OutFileDes;
   char Ach,Ach1;
   /* open file for read only */
   InFileDes = open("upper.txt", O_RDONLY);
   InFileDes1 = open("lower.txt", O_RDONLY);
  /* open file for write only or create only, created file mode will rw_ _ _ _ _ _
_*/
   OutFileDes = open ("updown.txt", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
   while  ( (read(InFileDes, &Ach, 1) == 1)&& (read(InFileDes1, &Ach1, 1)))
   {
      write(OutFileDes, &Ach, 1);
      write(OutFileDes, &Ach1, 1);
      lseek(OutFilesDes, 1, SEEK_CUR); // skip one space
   }
   close (InFileDes);
   close (InFileDes1);
   close (OutFileDes);
   exit (0);
}
```



```c
/*copyBfile.c: read data and save into a buffer
  and write to updown.txt */
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#define BUFFER_SIZE 4096
void err_sys(char *str)
{
   printf ("%s",str);
   exit (1);
}
int main()
{
   int InFileDes,OutFileDes;
   int nbyte;
   char buffer[BUFFER_SIZE];
   /* open file for read only */
   InFileDes = open("copyBfile.c", O_RDONLY);
  /* open file for write only or create only, created file mode will rw_ _ _ _ _ _ _*/
   OutFileDes = open  ("copyBfile.txt", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);
   while ((nbyte = read(InFileDes, buffer, BUFFER_SIZE)) >0)
      if (write (OutFileDes, buffer, nbyte) !=nbyte)
         err_sys("Write Error");
    if (nbyte <0)
      err_sys("Read Error");

   close (InFileDes);
   close (OutFileDes);
   exit (0);
}
```


Command Line Argument

- It is possible to pass arguments to C programs when they are executed.
- The brackets which follow main are used for this purpose.

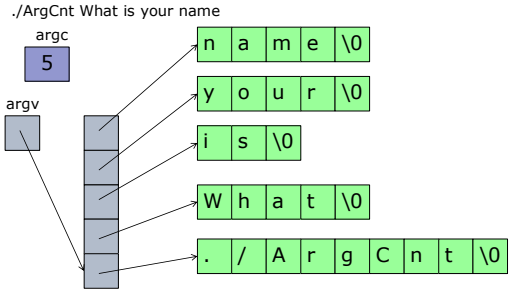
```
int main (int argc, char*argv[])
```

- *argc* :refers to the number of arguments passed, *argv[]:* a pointer array to c-string which points to each argument which is passed to main.

Command Line Argument

```c
/*   argcnt.c :count the number of argument passed and display
*/
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] )
{
   int num;
   printf ("You pass following %d arguments %s\n", argc);
   for(num =0; num <argc; num++)
   {
      printf ("The argument %d: %s\n", num, argv[num]);
   }
   if (argc == 1)
      printf("No arguments is passed !\n");

   exit (0);
}
```

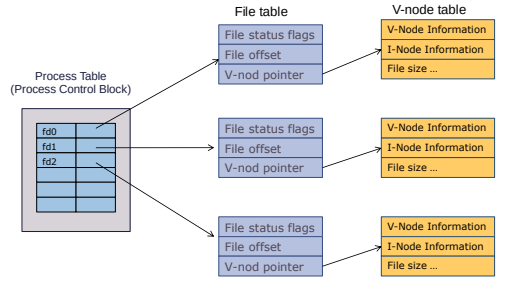

Command Line Argument



File Sharing

- Linux or Unix supports the sharing of open file between different processes.
- Kernel use three data structure for managing processes.
  - **Every process has an entry in the process table (process control block)**
    - The file descriptor flags
    - A pointer to a file table entry
  - **File table for open files for each processes**
    - File status flags for the file (read, write, append,…0
    - The current file offset
    - A pointer to the v-node table for the file
  - V-node table for each open files
    - Information about type of file
    - Pointer to the functions that operate on the file
    - i-node information

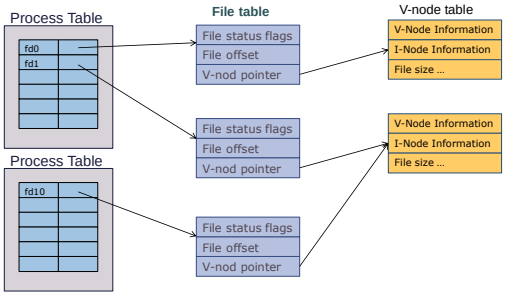
File Sharing



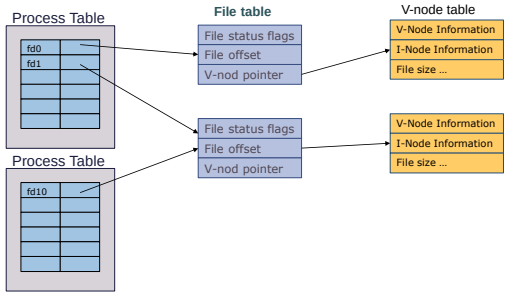
File Sharing

- How kernel take care of following situation when two processes sharing a file!
  - A process write to, a process read from a file
  - Both process append to a file
  - Both processes need do lseek()
    - Since each process has file table, based on the file status flag, before read, write, or lseek, check file size which is saved in v-table

File Sharing



File Sharing



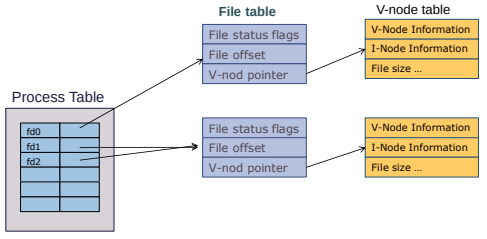
dup and dup2 System Calls

- A existing file descriptor can be duplicated by using dup() or dup2() system calls

```
#include <unistd.h>
int dup(int filedes);
int dup2(int filedes, int filedes2)

Return new file descriptor if OK, -1 on error
```

- The new file descriptor returned by `dup()` is guaranteed to be the lowest-numbered available file descriptor.
- With `dup2()`, we specify the value of the new descriptor with the filedes2 argument.
- If filedes2 is already open, it is first closed. If filedes equals filedes2, then dup2() returns filedes2 without closing it.

dup and dup2 System Calls

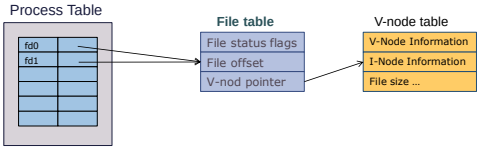


```c
/* This program dupexample.c shows example for dup() system call */
#include <unistd.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#define FILE_MODE (S_IRUSR | S_IWUSR|S_IRGRP|S_IROTH)
char buf[] ="Programmer:Sang-Eon Park";
char buf1[]="Date:09/25/08";

void err_sys(char *str)
{
   printf ("%s",str);
   exit (1);
}

int main()
{
   int filedes,filedes1;

   if ((filedes = creat("Share_file.txt", FILE_MODE)   ) <0)
      err_sys("Creat File Open Error");
   if (write (filedes, buf, sizeof(buf)) != sizeof(buf))
      err_sys("Creat Write Error");
   if ((filedes1= dup(filedes)) <0)
      err_sys("Creat Descriptor duplication error");
   if (lseek (filedes1, sizeof(buf)+1 , SEEK_SET)== -1)
      err_sys("Creat seek1 Error");//now offset =40 */
   if (write (filedes1, buf1, sizeof(buf1)) != sizeof(buf1))
      err_sys("Creat Write Error");
   close(filedes);
   exit (0);
}
```

dup and dup2 System Calls

filedes = creat("Share_file.txt", FILE_MODE)

filedes1=dup(filedes)




```c
/* dup2example.c: shows output redirection with dup2()*/
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
void err_sys(char *);
int main(int argc, char **argv)
{
          int pid, status;
          int newfd;  /* new file descriptor */

          if (argc != 2)
                    err_sys("usage: %s output file name\n");
          if ((newfd = open(argv[1], O_CREAT|O_TRUNC|O_WRONLY, 0644)) < 0)
                    err_sys("usage: open file error.\n");
          printf("This goes to the standard output.\n");
          printf("Now the standard output will go to \"%s\".\n", argv[1]);

          /* standard output is file descriptor 1, so we use dup2 to */
          /* to copy the new file descriptor onto file descriptor 1 */
          /* dup2 will close the current standard output */
          /* after dup2, standard output is closed and new file will become the standard output*/
          dup2(newfd, 1);

          printf("Now this goes to the new file, since new file becomes std output\n");
          exit(0);
}

void err_sys(char *str)
{
   printf ("%s",str);
   exit (1);
}
```



```c
/* dup2ex.c: shows output redirection with dup2()*/
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
char buf1[]="abcdefghij";
char buf2[]="ABCDEFGHIJ";
void err_sys(char *);
int main(int argc, char **argv)
{
          int pid, status;
          int fd1, fd2;          /* file descripters */

          if (argc != 3)
              err_sys("usage: %s output file name\n");
          if ((fd1 = open(argv[1], O_CREAT|O_TRUNC|O_WRONLY, 0644)) < 0)
              err_sys("usage: open file error.\n");
          if ((fd2 = open(argv[2], O_CREAT|O_TRUNC|O_WRONLY, 0644)) < 0)
              err_sys("usage: open file error.\n");

          if (write (fd1, buf1, 10) != 10)
              err_sys("usage: Write Error");
          printf("This goes to %s.\n", argv[2]);
          dup2(fd2, fd1); //output redirected to second file
          if (write(fd1, buf2, 10) != 10)
              err_sys("usage: write Error");
          exit(0);
}
void err_sys(char *str)
{
   printf ("%s",str);
   exit (1);
}
```



```c
/* dup2ex1.c: shows output redirection to standard output*/
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
char buf1[]="abcdefghij";
char buf2[]="ABCDEFGHIJ";

void err_sys(char *);
int main(int argc, char **argv)
{
          int pid, status;
          int fd;    /* file descriptor */

          if (argc != 2)
              err_sys("usage: %s output file name\n");
          if ((fd = open(argv[1], O_CREAT|O_TRUNC|O_WRONLY, 0644)) < 0)
              err_sys("usage: open file error.\n");

          if (write (fd, buf1, 10) != 10) //write to a file
              err_sys("usage: Write Error");

          printf("This goes to standard output");
          dup2(1, fd); //redirect to standard output
          if (write(fd, buf2, 10) != 10)
              err_sys("usage: write Error");
          exit(0);
}
void err_sys(char *str)
{
   printf ("%s",str);
   exit (1);
}
```



```c
/* dup2example1.c shows output redirection with dup2() send the output of a command to a file of the user's choice. */
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
void err_sys(char *);

int main(int argc, char **argv)
{
          int pid, status;
          int newfd;   /* new file descriptor */
          char *cmd[] = { "/bin/ls", "-al", "/", 0 };

          if (argc != 2)
              err_sys("usage: %s output file name\n");

          if ((newfd = open(argv[1], O_CREAT|O_TRUNC|O_WRONLY, 0644)) < 0)
              err_sys("usage: open file error.\n");

          printf("writing output of the command %s to \"%s\"\n", cmd[0], argv[1]);

          /* This new file will become the standard output after dup2.*/
          dup2(newfd, 1);

          /* now we run the command. It runs in this process and will have */
          /* this process' standard input, output, and error */
          if ((execvp(cmd[0], cmd))<0)
              err_sys("usage: execvp failed.\n");
          return 0;

}
void err_sys(char *str)
{
   printf ("%s",str);
   exit (1);
}
```

sat, fsat and lsat System Calls

- A file information (attributes) can get by calling `sat()`, `fsat()`, or `lstat()` system call.
- The `stat()` returns a structure of information about the named file.
- The `fstat()` obtains information about the file that is already open on the descriptor filedes.
- The `lstat()` function is similar to stat, but when the named file is a symbolic link, `lstat()` returns information about the symbolic link,

sat, fsat and lsat System Calls

- A file information (attributes) can get by calling `sat()`, `fsat()`, or `lstat()` system call.

```
#include <sys/types.h>
#include<sys/sat.h>
int stat(const char *pathname, struct stat *buf);
int fsat(int filedes, struct stat *buf);
int lstat(cons char *pathname , struct stat *buf);

                                    Returns: 0 if OK, -1 on error
```

- The stat() returns a structure of information about the named file.
- The fstat() obtains information about the file that is already open on the descriptor filedes.
- The lstat() function is similar to stat, but when the named file is a symbolic link, lstat() returns information about the symbolic link,

sat, fsat and lsat System Calls

```
struct stat {
        mode_t st_mode;   /*file type & mode (permissions) */
        ino_t st_ino;     /* i-node number */
        dev_t st_dev;     /* device number (file system) */
        dev_t st_rdev;    /* device number for special files */
        nlink_t st_nlink; /* number of links */
        uid_t st_uid;     /* user ID of owner */
        gid_t st_gid;     /* group ID of owner */
        off_t st_size;    /* size in bytes, for regular files */
        time_t st_atime;  /* time of last access */
        time_t st_mtime;  /* time of last modification */
        time_t st_ctime;  /* time of last file status change */
        blksize_t st_blksize; /* best I/O block size */
        blkcnt_t st_blocks; /*number of 512 byte blocks allocated */
        mode_t st_attr;   /* The DOS-style attributes for this file */
};
```


sat, fsat and lsat System Calls

- File Types
  - **Regular file**: contains data of some form (text or binary)
  - **Directory file**: contains the name of other files and pointers to the information on these file. Only kernel can write to the directory file
  - **Character special file**: A type of file providing unbuffered I/O access in variable-sized units to devices.
  - **Block special file**: A type of file used for disk devices.
  - **FIFO**: A type of file used for inter-process communication
  - **Socket**: A type of file used for network communication
  - **Symbolic link**: A type of file that points to another file.

sat, fsat and lsat System Calls

- The type of a file is encoded in the **st_mode** member of the **stat** structure.
- We can determine the file type by using macros In <sys/stat.h>
  - S_ISREG(st_mode): Regular file
  - S_ISDIR(st_mode): Directory
  - S_ISCHR(st_mode): Character special
  - S_ISBLK(st_mode): Block special
  - S_ISFIFO(st_mode) : FIFO
  - S_ISLNK(.st_mode): Symbolic link
  - S_ISSOCK(st_mode): Socket


```
* typeoffile.c display type of files*/
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
void err_ret(char *str)
{
   printf ("%s",str);
   exit (1);
}
int main (int argc, char argv[])
{
   int i;
   struct stat buf;
   char *ptr;

   if (argc != 2)
            err_ret("Argument number error.\n");
   /*get for a file infomation */
   if (stat(argv[1], &buf) < 0)
            err_ret("lstat Error");
   if (S_ISREG(buf.st_mode)) ptr ="regular";
   else if (S_ISDIR(buf.st_mode)) ptr ="directory";
   else if (S_ISCHR(buf.st_mode)) ptr ="character special";
   else if (S_ISBLK(buf.st_mode)) ptr = "block special";
   else if (S_ISFIFO(buf.st_mode)) ptr="fifo";
   else if (S_ISLNK(buf.st_mode)) ptr = "symbolic link";
   else if (S_ISSOCK(buf.st_mode)) ptr ="socket";
   else ptr ="unknown mode ";

   printf ("%s \n", ptr);
   exit (0);
}
```

```c
// filetype.c check a file type by passing path as a parameter
int main(int argc, char *argv[])
{
  struct stat sb;
  if (argc != 2) {
      fprintf(stderr, "Usage: %s <pathname>\n", argv[0]);
      exit(1);
  }
  // get a file infomation and save in stat structure type
  if (stat(argv[1], &sb) == -1) {
      perror("stat");
      exit(1);
  }
  printf("File type:                ");
  // check type of file with macros
  switch (sb.st_mode & S_IFMT) {
   case S_IFBLK:  printf("block device\n");            break;
   case S_IFCHR:  printf("character device\n");        break;
   case S_IFDIR:  printf("directory\n");               break;
   case S_IFIFO:  printf("FIFO/pipe\n");               break;
   case S_IFLNK:  printf("symlink\n");                 break;
   case S_IFREG:  printf("regular file\n");            break;
   case S_IFSOCK: printf("socket\n");                  break;
   default:       printf("unknown?\n");                break;
   }
   // print file informations
   printf("I-node number:            %ld\n", (long) sb.st_ino);

   printf("Mode:                     %lo (octal)\n",(unsigned long) sb.st_mode);

   printf("Link count:               %ld\n", (long) sb.st_nlink);
   printf("Ownership:                UID=%ld   GID=%ld\n",
           (long) sb.st_uid, (long) sb.st_gid);

   printf("Preferred I/O block size: %ld bytes\n", (long) sb.st_blksize);
   printf("File size:                %lld bytes\n",(long) sb.st_size);
   printf("Blocks allocated:         %lld\n",(long) sb.st_blocks);

   printf("Last status change:       %s", ctime(&sb.st_ctime));
   printf("Last file access:         %s", ctime(&sb.st_atime));
   printf("Last file modification:   %s", ctime(&sb.st_mtime));

   exit(0);
}
```

ID's for a Process

- A process has more than six IDs
  - **Real user ID** and **group ID** – These two fields are taken from our entry in the password file when we log in. Normally, these values don't change during a login session but it can be changed by superuser.
  - **Effective user ID**, **group ID** –used for file access permission.
  - **Saved set user ID and group ID** –contain copies of the effective user ID and the effective group ID, respectively, when a program is executed. These are saved by exec function
- Normally, the effective user ID equals the real user ID, and the effective group ID equals the real group ID.

ID's for a Process

- Every file has an owner and group owner are specified by by st_uid and st_gid in **stat** structure.
- When a program is executed, usually its effective user ID is the real user ID and effective group ID is usually the real group ID.
- However, by setting special flag in st_mode a process can be treated different way.
  - When a file is executed, set the effective user ID of the process to be the owner of the file (ex. even though owner of the file is superuser).

File Access Permission

- File access permission bits from <sys/sat.h>

st_mode mask	Meaning
S_IRUSR	user-read
S_IWUSR	user-write
S_IXUSR	user-execute
S_IRGRP	group-read
S_IWGRP	group -write
S_IXGRP	group -execute
S_IROTH	other-read
S_IWOTH	other-write
S_IXOTH	other-execute

File Access Permission

- Kernel perform the file access test each time a process opens, creates or modifies.
- It is depends on the owner of the file (st_uid, st_gid) and **effective IDs**.
  - If the effective user ID of the process is 0 (superuser) access allows.
  - If user ID is same as effective ID, access allows depends on the permission bit.
  - If the effective group ID of the process equals the group ID of the file (even user ID does not match), access is allowed if the appropriate group access permission bit is set.

Ownership of New Files and Directories

- When a new file or directory is created, the rule for the ownerships are followings.
  - The user ID of a new file (or directory) is set to the effective user ID of the process.
  - The group ID is set by implementing one of the following options.
    1. The group ID of new file (or directory) can be the effective group ID of the process.
    2. The group ID of new file can be the group ID of the directory in which the file (or directory) is being created.

access() System Call

- When accessing a file with the open system call, the kernel performs its access test based on the effective IDs.
- But, if a process is running inside a process, some case, we need check its access test based on the real ID's. (ex. If a process create a child which runs a program with exec())
- The **access** function can performs the test on real user ID and group IDs.

access() System Call

```
#include <unistd.h>
  int access(const char *pathname, int mode)

                                Returns: 0 if OK, -1 on error
```

Mode in <unistd.h>

- F_OK: test for existence of a file
- R_OK: test for read permission
- W_OK: test for write permission
- X_OK: test for execute permission


```
/* the accessex.c show a example to use access system call */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>

void err_ret(char *str)
{
   printf ("Access Error for %s\n",str);
   exit (1);
}
int main (int argc, char *argv[])
{
   if (argc != 2)
      err_ret ("More than two argument Error");
   if (access (argv[1], R_OK) <0)
      err_ret ( argv[1]);
   else
      printf ("read access OK\n");
   if (access (argv[1], W_OK) <0)
      err_ret ( argv[1]);
   else
      printf ("write access OK\n");
   if (access (argv[1], X_OK) <0)
      err_ret ( argv[1]);
   else
      printf ("execute access OK\n");
   exit(0);
}
```


access() System Call

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
   pid_t pid;

   if((pid = fork()) < 0 ) /* create a child process*/
   {
      perror("fork failure");
      exit(1);
   }
   if(child_pid == 0) /* a child process*/
   {
      if ((access ("/bin/xprogram" , X_OK) == 0)) //check accessable
      {
            execl("/bin/xprogram", "xprogram", 0, 0);
            _exit(1);
      }
      else
      {
            perror("access error");
            exit(1);
      }
 * parent process*/
   else
   {
      ...
   }

   return 0;
}
```


A System Call umask

- Nine permission bits are associated with a file.
- A umask() system call set the file mode creation mask for the process and return the previous value
- Prototype

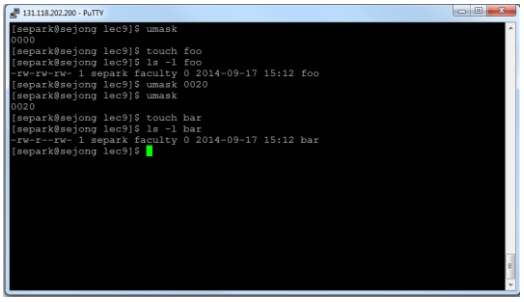
```
#include <sys/types.h>
#include <sys/stat.h>
mode_t umask(mode_t cmask);
                Returns: previous file mode creation
```

- Most user does not deal with umask value.
- When writing a programs that create new files, if we want to assure that specific access permission bits are not enabled, we must modify the umask value while the process is running.

A System Call `umask()`

- We can check current mask value with shell command `umask`. It shows current mask value for file creation.
- If the mask is cleared (0000) then we can create a file with any mode.
- But if mask is (0020), write protected for group. A file will be created without group write permission.
- `touch` shell command create a file with `rw-rw-rw` with cleared mask. But if mask is 0020, a file will be created with `rw-r--rw-`.
- When writing a programs that create new files, if we want to assure that specific access permission, you must clear the file mode creation mask by umask() system call before creating a new file.

A System Call `umask()`




```c
/*******************************************************
  umask.c : change a file permission inside a process
*******************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>

void err_sys(char *str)
{
   printf ("%s\n",str);
   exit (1);
}
int main ()
{
   umask(0); /* clear mask */
   if (creat("foo",S_IRUSR |S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH) <0)
     err_sys ("creat Error for foo ");
   umask(S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH); */ same as 0033 */
   if (creat ("bar",S_IRUSR |S_IWUSR|S_IRGRP|S_IWGRP|S_IROTH|S_IWOTH) <0)
     err_sys ("creat Error for bar ");
   exit(0);
}
```