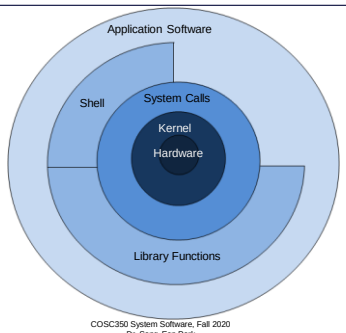
Preview

- System Call
- Library Functions
- File Descriptors for a Process
- System Call for Managing Files
  - write()
  - read()
  - open()
  - close()
  - lseek()
  - pwrite(), pread();

System Call

- A system call is a request for the operating system to do something on behalf of the user's program.
- The system calls are functions used in the kernel itself.
- To the programmer, the system call appears as a normal C function call.
- When a system call, control change from user's mode to kernel's mode.

System Call



System Call

- System calls is the layer of software interface to the kernel.
- It is controlled by kernel (unbuffered I/O)
- Library functions are built on top of system call (Buffered I/O). It is called and controlled by a process.
- Linux system calls are used to process management, file system management, and inter-process communication.
- The Linux system interface consist of about 80 system calls.
- To access and control file, need system call to open, read, write, close file.

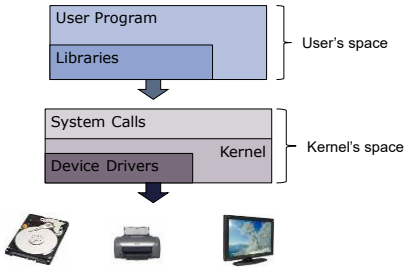
System Call

- A system call is very expensive routine
  - Change a mode from user's mode to kernel's mode.
  - Save all parameters for user's program for later execution (save snapshot of CPU)
  - Load all necessary parameters for system call routine to CPU .
  - Execute the system call routine.
  - After system call routine, load all saved parameters for user's program to CPU.
  - Change mode from user's mode to kernel's mode.
  - Continue execution of user's program.

Library Functions

- To reduce the overhead of system calls, system such as Unix or Linux provide library functions use buffers.
- For example, I/O function library that provide buffered output.
- With these library functions, system call routine need execute when size of buffer becomes full.
- This dramatically reduce the system call overhead.

Library Functions



File Descriptors for a Process

- To the kernel, all open files are referred to by file descriptors.
- A file descriptor is a non-negative integer.
- When we open an existing file or create a new file, the kernel returns a file descriptor to the process.
- We can read or write a file with the file descriptor which was return by system calls open() or create().

File Descriptors for a Process

- A process has a number of the descriptors associated it.
- When a process is created, it has three descriptors which can be used for input, output and error.
  - 0: Standard input
  - 1: Standard output
  - 2: Standard error

System Calls for Managing Files (`write()`)

- The write() system call attempts to write *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the open file descriptor, *fildes*.
- It returns the number of bytes actually write.

```
#include <unistd.h>
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

Returns: number of bytes written, or -1 on error

System Calls for Managing Files (`write()`)

```
//write.c
#include <unistd.h>
#include <stdlib.h>

int main()
{
   if ((write (1, "Hear is some data\n", 18)) != 18)
        write (2, "error on file descriptor 1\n", 46);
   exit (0);
}
```


System Calls for Managing Files (`read()`)

- The read() system call function attempts to read *nbyte* bytes from the buffer pointed to by *buf* to the file associated with the open file descriptor, *fildes*.
- It returns the number of bytes actually read.

```
#include <unistd.h>
ssize_t read(int fildes, const void *buf, size_t nbyte);
```

Returns: size of byte read, 0 if end of file, -1 on error

System Calls for Managing Files (read())

```
/* read_buffer.c
/* prepare 128 bytes buffer and read a string upto 128 bytes*/
#include <stdio.h>
#include <unistd.h>

int main()
{
   char buffer[128];
   int nread;
   nread = read(0, buffer, 128);

   if (nread ==-1)
      write (2, "A read error\n", 12);

   if ((write (1, buffer, nread))!= nread)
      write (2, "A write Error!\n", 14);
   exit (0);
}
```


System Calls for Managing Files (read())

```
/* read_buffer1.c */
/* read and write byte by byte until empty (Ctr-D)*/

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

int main()
{
        char b[1];
        int nread;
        while ((nread =read(0, b, 1) > 0))
          write (1, b, nread);

        return 0;


}
```


System Calls for Managing Files

- Only using read and write system call, we can copy standard input to standard output.
- This assume that these have been set up by the shell before this program is executed.

System Calls for Managing Files

```
/* copyStdIO.c copy standard input to standard output */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h> /*STDIN_FILENO, STDOUT_FILENO*/

/* buffer size effect the efficiency of the program */
#define BUFFER_SIZE 2
void err_sys(char *str)
{
   printf ("%s",str);
   exit (1);
}
int main()
{
   int nbyte;
   char buffer[BUFFER_SIZE];

   while ((nbyte = read(STDIN_FILENO, buffer, BUFFER_SIZE)) >0)
      if (write (STDOUT_FILENO, buffer, nbyte) != nbyte)
         err_sys ("Write Error");
   if (nbyte <0)
      err_sys("read Error");
   exit (0);
}
```


System Calls for Managing Files (open())

- open() lets you open a file for reading, writing, or reading and writing.
- It returns a file descriptor.
- The prototype for the open() system call is:

```
#include <fcntl.h>
int open(const char *fname, int flags )
int open (const char *fname, int flags, mode_t mode);
```

Returns: file descriptor or -1 on error

- The third argument mode is used only when a new file is being created.

System Calls for Managing Files (open())

- The allowable flags as defined in "/usr/include/fcntl.h" are:

- #define O_RDONLY 0 /* Open the file for reading only */
- #define O_WRONLY 1 /* Open the file for writing only */
- #define O_RDWR 2 /* Open the file for both reading and writing*/
- #define O_NDELAY 04 /* Non-blocking I/O */
- #define O_APPEND 010 /* append (writes guaranteed at the end) */
- #define O_CREAT 00400 /*open with file create (uses third open arg) */
- #define O_TRUNC 01000 /* open with truncation */
- #define O_EXCL 02000 /* exclusive open */

System Calls for Managing Files (open())

- The allowable mode_type as defined in a header sys/stat.h" are:
  - S_IRUSR Read permission, owner.
  - S_IWUSR Write permission, owner.
  - S_IXUSR Execute/search permission, owner.
  - S_IRGRP Read permission, group.
  - S_IWGRP Write permission, group.
  - S_IXGRP Execute/search permission, group.
  - S_IROTH Read permission, others.
  - S_IWOTH Write permission, others.
  - S_IXOTH Execute/search permission, others.
- Or we can use octal number numerical form
  - 0777, 0755, 0555, ….


```c
/*  open.c demonstrate open file */
#include <fcntl.h>          /* defines options flags */
#include <sys/types.h>      /* defines types used by sys/stat.h */
#include <sys/stat.h>       /* defines S_IREAD & S_IWRITE  */
static char message[] = "Hello, world";

int main()
{
    int fd;
    char buffer[80];

    /* open for rw and create exclusive open, ceate as rw-rw-rw */
    fd = open("datafile.dat",O_RDWR | O_CREAT | O_EXCL, S_IREAD | S_IWRITE);
    printf ("fd = %d\n",fd);
    if (fd != -1)
    {
        printf("datafile.dat opened for read/write access\n");
        write(fd, message, sizeof(message));
        lseek(fd, 0, SEEK_SET);/* go back to the beginning of the file */
        if (read(fd, buffer, sizeof(message)) == sizeof(message))
            printf("\"%s\" was written to datafile.dat\n", buffer);
        else
            printf("*** error reading datafile.dat ***\n");
        close (fd);
    }
    else
        printf("*** datafile.dat already exists ***\n");
    return 0;
}
```


System Calls for Managing Files (creat())

- A new file can also be created by calling the create system call.
- One deficiency with `creat()` is that the file is opened only for writing.

```c
#include <fcntl.h>
int create (const char *fname, mode_t mode);
```

Return: file descriptor, or -1 on error

System Calls for Managing Files (close())

- Any opened file is closed by a system call close.
- Closing a file releases any record that the process may have on the file

```c
#include <fcntl.h>
int close (int filedes);
```

Returns: 0 if OK, -1 on error

System Calls for Managing Files (lseek())

- Every open file has an associated with current file offset.
- 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.

System Calls for Managing Files (lseek())

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

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

Returns: new file offset or -1 on error

System Calls for Managing Files (lseek())

- 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

System Calls for Managing Files (lseek())

```
/*  program testlseek.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>

#define FILE_MODE (S_IRUSR | S_IWUSR|S_IRGRP|S_IROTH)
/* err_sys display error message and exit  */
void err_sys(char *);

int main()
{
   int filedes, offset;

   if ((filedes = open("testlseek.txt", FILE_MODE)) <0)
      err_sys("Creat File Open Error");

   if ((offset = lseek (filedes, 0, SEEK_END))== -1)
      err_sys("Creat seek1 Error");

   printf("offset = %d", offset);
   return 0;
}
void err_sys(char *str)
{
   printf ("%s",str);
   exit (1);
}
```


System Calls for Managing Files (lseek())

```
/* lseek.c this program test its standard input to see whether
   it is capable of seeking or not */
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
   if (lseek(STDIN_FILENO, 0, SEEK_CUR)== -1)
     printf("cannot seek.\n");
   else
     printf("Seek OK. \n");
   exit (0);
}
```


System Calls for Managing Files (lseek())

```
/* lseek1.c this program display size of input file with input
redirection */
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main()
{
   int offset;
   if ((offset =lseek(STDIN_FILENO, 0, SEEK_END))== -1)
     printf("cannot seek.\n");
   else
     printf("file size is %d bytes.\n", offset);
   exit (0);
}
```



```
/*  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";
void err_sys(char *str)
#define FILE_MODE (S_IRUSR | S_IWUSR|S_IRGRP|S_IROTH)
int main()
{
   int filedes;

   if ((filedes = open("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;
}
/* err_sys display error message and exit  */
void err_sys(char *str)
{
   printf ("%s",str);
   exit (1);
}
```


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 write 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

```c
//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)+1);
        // read the second message from the file
        nr = pread(fd, &buf_rd, sizeof(buf_rd), strlen(buf_wr)+1);
        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

```c
#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

```c
#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;
}
```