Lecture #10 – Interprocess Communication (Chapter 15)

- Introduction

  Now that we have studied the creation of processes, we need to show methods for communicating among various processes.

  Some examples of IPC (interprocess communication) include pipes, FIFO queues, message queues, semaphores, shared memory, sockets, and streams.

  Note: See page 427 for a list of these IPC mechanisms and the O/S versions they support

  The only IPC mechanism that is supported on every O/S/ version is the half-duplex pipe.

  All but sockets and streams require that the processes involved in the communication reside on the same host (machine).

- Pipes

  Pipes are the oldest form of IPC, and have two distinct limitations:

  1. They are half-duplex (data flow only in one direction).
  2. They can be used only between processes that have a common ancestor. For example, a process normally creates a pipe, then calls fork, and uses the pipe to communicate between parent and child.

  Syntax: int pipe(int filedes[2]);

  Two file descriptors are returned through "filedes":

  filedes[0] is opened for reading
  filedes[1] is opened for writing
  The output of filedes[1] is the input for filedes[0].

  Note: See figures on top of page 497 for a visual representation.

  After we create the pipe and call fork, then we need to decide which way we need data to flow:

  If from parent to child, the parent closes filedes[0], and child closes filedes[1].
  If from child to parent, the child closes filedes[0], and parent closes filedes[1].

  Note: See picture on page 498 for parent to child flowing pipe.

If we try and read from a pipe whose write end has been closed, after all data has been read, then "read" returns a 0 to indicate EOF.

If we try and write to a pipe whose read end has been closed, the signal SIGPIPE is generated. If we ignore this signal, or return from our handler, then write returns error with errno set to EPIPE.

Example:

```
int main(void)
{
        int     n, fd[2];
        pid_t   pid;
        char    line[MAXLINE];

        if (pipe(fd) < 0)
        {
                perror("pipe error");
                exit(1);
        }

        if ((pid = fork()) < 0)
                perror("fork failed");
        else if (pid > 0)
        {
                /* parent */

                close(fd[0]);
                write(fd[1], "hello world\n", 12);
        }
        else
        {
                /* child */

                close(fd[1]);
                n = read(fd[0], line, MAXLINE);
                write(0, line, n);
        }

        exit(0);
}
```

Example: (pager_one.c)

```
#define DEF_PAGER       "/bin/more"             /* default pager program */
#define MAXLINE         132

int main(int argc, char *argv[])
{
        int     n, fd[2];
        pid_t   pid;
        char    line[MAXLINE], *pager, *argv0;
        FILE    *fp;

        if (argc != 2)
        {
                fprintf(stderr, "usage: %s <pathname>\n", argv[0]);
                return(1);
        }

        if ( (fp = fopen(argv[1], "r")) == NULL)
        {
                fprintf(stderr, "can't open %s", argv[1]);
                return(1);
        }

        if (pipe(fd) < 0)
        {
                perror("pipe");
                return(1);
        }

        if ( (pid = fork()) < 0)
        {
                perror("fork");
                return(1);
        }
        else if (pid > 0)
        {
                close(fd[0]);

                while (fgets(line, MAXLINE, fp) != NULL)
                {
                        n = strlen(line);

                        if (write(fd[1], line, n) != n)
                        {
                                perror("write");
```

```
                return(1);
            }
        }

        if (ferror(fp))
        {
            perror("fgets");
            return(1);
        }

        close(fd[1]);

        if (waitpid(pid, NULL, 0) < 0)
        {
            perror("waitpid");
            return(1);
        }

        return(0);

    }
    else
    {
        close(fd[1]);

        if (fd[0] != 0)
        {
            if (dup2(fd[0], 0) != 0)
            {
                perror("dup2");
                return(1);
            }
            close(fd[0]);
        }

        if ( (pager = getenv("PAGER")) == NULL)
            pager = DEF_PAGER;

        if ( (argv0 = strrchr(pager, '/')) != NULL)
            argv0++;        /* step past rightmost slash */
        else
            argv0 = pager;  /* no slash in pager */

        if (execl(pager, argv0, (char *) 0) < 0)
        {
            perror("execl");
```

```
                return(1);
            }
        }
    }
```

- Popen and pclose

```
FILE   *popen(const char *cmdstring, const char *type);
int    pclose(FILE *fp);
```

Popen does a fork and exec to execute the "cmdstring", and returns a stdio FILE pointer.
If "type" is "r", then the file pointer is connected to stdout of "cmdstring".
If "type" is "w", then the file pointer is connected to stdin of "cmdstring".

Pclose closes the stdio streams, and waits for the command to terminate, and returns the termination status of the shell.

Example:

```
fp = popen("ls *.c", "r");
```

Example:

```
int main(int argc, char **argv)
{
        char    line[MAXLINE];
        FILE    *fpin, *fpout;

        if (argc != 2)
        {
                fprintf(stderr, "usage: %s <pathname>", argv[0]);    exit(1);
        }

        if ((fpin = fopen(argv[1], "r")) == NULL)
        {
                fprintf(stderr, "Cannot open file %s\n", argv[1]);    exit(1);
        }

        if ((fpout = popen("more", "w")) == NULL)
        {
                perror("popen more");                                exit(1);
        }

        while (fgets(line, MAXLINE, fpin) != NULL)
                fputs(line, fpout);
```

```
        if (pclose(fpout) == -1)
                perror("pclose");

        fclose(fpin);
}
```

Note: See sample implementation of popen on pages 505-507. Basic structure is as follows:

```
validate arguments
create pipe
fork
        C:
                if type is "r"
                        close(fd[0]);
                        dup2(pfd[1], 1);
                        close(fd[1]);

                if type is "w"
                        close(fd[1]);
                        dup2(fd[0], 0);
                        close(fd[0]);

                excecl("/bin/sh", "sh", "-c", cmdstring, (char *) 0);
                _exit(127);

        P:
                if type is "r"
                        close(fd[1]);
                        fp = fdopen(fd[0], type);

                if type is "w"
                        close(fd[0]);
                        fp = fdopen(fd[1], type);
```

- Co-processes

A UNIX filter is a program that:

1. Reads from stdin
2. Performs some processing
3. Writes to its stdout

Filters are very commonly used in pipelines.

```
$  who | awk '{print $1}' | sort | uniq
```

In this example, awk and sort are operating as filters.

A filter becomes a co-process when the same process generates its input and reads its output (see picture on page 510).

Example:

Suppose we had a program called "add" that reads two numbers from stdin, computes their sum, and writes to result to stdout.

```
int main(void)
{
        int     n, fd1[2], fd2[2];
        pid_t   pid;
        char    line[MAXLINE];

        if (signal(SIGPIPE, SIG_IGN) == SIG_ERR)
                perror("signal error");

        if (pipe(fd1) < 0 || pipe(fd2) < 0)
                perror("pipe");

        if ((pid = fork()) < 0)
                perror("fork");
        else if (pid > 0)                       /* parent */
        {
                close(fd1[0]);
                close(fd2[1]);

                while (fgets(line, MAXLINE, stdin) != NULL)
                {
                        n = strlen(line);

                        if (write(fd1[1], line, n) != n)
                                perror("write to pipe");

                        if ((n = read(fd2[0], line, MAXLINE)) < 0)
                                perror("read from pipe");

                        if (n == 0)
                                break;

                        line[n] = 0;
                        fputs(line, stdout) ;
                }
```

```
                exit(0);
        }
        else                            /* child */
        {
                close(fd1[1]);
                close(fd2[0]);

                dup2(fd1[0], 0);
                dup2(fd2[1], 1);

                if (execl("./add", "add", (char *) 0) < 0)
                        perror("execl");
        }
}
```

Note: see "add" programs on pages 511 (unbuffered I/O) and 513 (stdio). The stdio program does not work, why?

- FIFO's

  FIFO (first in, first out) queues are also called named pipes.

  Pipes can be only used when the processes shared a common ancestor (through fork), but FIFO queues allow processes to exchange data regardless of their process ancestry.

  Creating a FIFO is similar to creating a file:

  int mkfifo(const char *pathname, mode_t mode);

  After creation, we may open it using "open", and operate on it with the normal file functions (close, read, write, etc).

  Example:

```
mkfifo fifo1
prog3 < fifo1 &
prog1 < infile | tee fifo1 | prog2
```

- System V IPC Background

System V IPC is basically three related but different IPC mechanisms - semaphores, shared memory, and message queues.

Identifiers and Keys:

Each IPC structure is referred to by a non-negative integer called an "identifier".
Whenever an IPC structure is created, a "key" must be specified.
The "key" is converted into an "identifier" by the kernel.

Techniques for sharing IPC structures:

1. The server can create a new IPC structure by specifying a key of IPC_PRIVATE, and either store the "identifier" somewhere for the client to obtain, or call fork to create a child who inherently knows the "id".
2. The client and server can agree on a key by defining the key in a common header or configuration file. It is possible for the IPC structure to already exist, and this situation must be handled.
3. The client and server can agree on a pathname and a project ID (a char between 0 and 255), and call the function "ftok". This function takes these two values, and generates a "key". The "key" can then be used to generate the "id".

The IPC_CREAT flag is used to indicate that we are trying to create the structure. If the flag is omitted, then we are trying to attach to a previously created IPC structure.

If we want to generate an error if the structure already exists, and we are trying to create it, then we use the IPC_EXCL flag.

Permission Structure:

```
struct ipc_perm {
        uid_t       uid;        /*  owner */
        gid_t       gid;        /*  group owner */
        uid_t       cuid;       /*  creator's UID */
        gid_t       cgid;       /*  creator's GID */
        mode_t      mode;       /*  access permissions */
        ulong       seq;        /*  slot usage sequence number */
        key_t       key;        /*  key */
};
```

All fields except "seq" are initialized when the IPC is created.

Note: see page 520 for description of mode; notice no execute permissions.

Configuration limits:

All three forms of IPC have limits built into the kernel.
Most of these limits can be tuned by rebuilding the kernel (root only).

Advantages / Disadvantages:

System V IPC structures are system wide with no reference count.
They are no automatically deleted or reclaimed, and must be explicitly removed.
They do not share file system semantics (no open, read, write, etc).