Chapter 15: Interprocess Communication

- Half-duplex pipes
- FIFOs
- Full-duplex pipes
- Named full duplex pipes
- Semaphores
- Shared memory
- Sockets
- STREAMS
Pipes

- Pipe limitations
  - Historically half-duplex
  - Can only be used between processes with a common ancestor

- `pipe()` - Postfix delivery to external command
  - `int pipe(int fd[2]);`

- Reading the closed end of a pipe returns 0 for EOF

- Writing the closed end of a pipe returns -1
  - Sends SIGPIPE signal
Figure 15.2 Two ways to view a half-duplex pipe
Figure 15.3  Half-duplex pipe after a fork
Figure 15.4  Pipe from parent to child
#include "apue.h"

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

    if (pipe(fd) < 0)
        err_sys("pipe error");
    if ((pid = fork()) < 0)
        err_sys("fork error");
    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(STDOUT_FILENO, line, n);
    }
    exit(0);
}
Figure 15.8  Using two pipes for parent–child synchronization
popen() and pclose()

- `popen()` - "opens" a process by creating a bidirectional pipe, forking, and invoking the shell.
  - `FILE * popen(const char *command, const char *mode);`

- `pclose()` – waits for the associated process to terminate; it returns the exit status of the command
  - `int pclose(FILE *stream);`
Figure 15.9  Result of \texttt{fp = popen(cmdstring, "r")}

Figure 15.10  Result of \texttt{fp = popen(cmdstring, "w")}
#include "apue.h"
#include <sys/wait.h>

#define PAGER   "${PAGER:-more}" /* environment variable, or default */

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

  if (argc != 2)
    err_quit("usage: a.out <pathname>");
  if ((fpin = fopen(argv[1], "r")) == NULL)
    err_sys("can’t open %s", argv[1]);
  if ((fpout = fopen(PAGER, "w")) == NULL)
    err_sys("popen error");

  /* copy argv[1] to pager */
  while (fgets(line, MAXLINE, fpin) != NULL) {
    if (fputs(line, fpout) == EOF)
      err_sys("fputs error to pipe");
  }

  if (ferror(fpin))
    err_sys("fgets error");
  if (pclose(fpout) == -1)
    err_sys("pclose error");

  exit(0);
}

**Figure 15.11**  Copy file to pager program using popen
Figure 15.13  Transforming input using `popen`
```c
#include "apue.h"
#include <ctype.h>

int
main(void)
{
    int c;

    while ((c = getchar()) != EOF) {
        if (isupper(c))
            c = tolower(c);
        if (putchar(c) == EOF)
            err_sys("output error");
        if (c == '\n')
            fflush(stdout);
    }
    exit(0);
}
```

**Figure 15.14** Filter to convert uppercase characters to lowercase
#include "apue.h"
#include <sys/wait.h>

int main(void)
{
    char line[MAXLINE];
    FILE *fpin;

    if ((fpin = popen("myuclc", "r")) == NULL)
        err_sys("popen error");
    for (; ; ) {
        fputs("prompt> ", stdout);
        fflush(stdout);
        if (fgets(line, MAXLINE, fpin) == NULL) /* read from pipe */
            break;
        if (fputs(line, stdout) == EOF)
            err_sys("fputs error to pipe");
    }
    if (pclose(fpin) == -1)
        err_sys("pclose error");
    putchar('
');
    exit(0);
}

Figure 15.15 Invoke uppercase/lowercase filter to read commands
Coprocesses

- Coprocesses are two way pipes
  - popen() gives us a one way pipe

Figure 15.16  Driving a coprocess by writing its standard input and reading its standard output
FIFOs: mkfifo() and mkfifoat()

- Two uses of FIFO
  - Used to pass data between shells
  - Used to pass data between client and server

- mkfifo - make fifos
  - int mkfifo(const char* path, mode_t mode);

- mkfifoat – make fifo in location relative to fd
  - int mkfifoat(int fd, const char* path, mode_t mode);
  - If path is absolute pathname fd parameter is ignored
  - If path is relative it is evaluated relative to fd
  - If fd parameter is AT_FDCWD, the path is relative to current working directory
FIFO Example: Duplicate Output Stream

Figure 15.20  Procedure that processes a filtered input stream twice
**FIFO Example: Client-Server**

**Figure 15.21** Using a FIFO and `tee` to send a stream to two different processes
FIFO Example: Client-Server

Figure 15.22 Clients sending requests to a server using a FIFO
Figure 15.23  Client–server communication using FIFOs
XSI IPC

- XSI IPC
  - Message Queues
  - Semaphores
  - Shared Memory
Identifiers and Keys

- IPC structures are identified using keys

- Client/server have different ways to rendezvous at the same IPC
  - Server creates a new IPC structure and stores it in a file for the client
  - Define the key in a common header
  - Specify a pathname and project id and use ftok()

- ftok() - create IPC identifier from path name
  - key_t ftok(const char* path, int id);
struct ipc_perm
{
    key_t key;
    uid_t uid;   /* owner effective uid */
    gid_t gid;  /* owner effective gid */
    uid_t cuid; /* creator effective uid*/
    gid_t cgid; /* creator effective gid*/
    mode_t mode; /* access modes */
    ushort seq;  /* slot usage sequence number */
};
Disadvantages of IPC

- Disadvantages
  - IPC structures are system wide but have not reference count
    - Don’t terminate with associated process
  - Not known by name in the file system
    - Need key for reference
  - Don’t use file descriptors
    - Can’t use multiplexed I/O
### Advantages of IPC

<table>
<thead>
<tr>
<th>IPC type</th>
<th>Connectionless?</th>
<th>Reliable?</th>
<th>Flow control?</th>
<th>Records?</th>
<th>Message types or priorities?</th>
</tr>
</thead>
<tbody>
<tr>
<td>message queues</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>STREAMS</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>UNIX domain stream socket</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>UNIX domain datagram socket</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>FIFOs (non-STREAMS)</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
</tr>
</tbody>
</table>

**Figure 15.25** Comparison of features of various forms of IPC
Message Queues

- Message queues are linked lists of messages stored in the kernel
  - Identified by a message queue ID

```c
struct msqid_ds {
    struct ipc_perm msg_perm;
    msgqnum_t *msg_qnum; /* # of messages on queue */
    msglen_t msg_qbytes; /* max # of bytes on queue */
    pid_t msg_lspid; /* pid of last msgsnd() */
    pid_t msg_lrpid; /* pid of last msgrcv() */
    time_t msg_stime; /* last msgsnd() time */
    time_t msg_rtime; /* last msgrcv() time */
    time_t msg_ctime; /* last change time */
    ...
};
```
Message Queues: msgget() and msgctl()

- msgget() - get a message queue identifier
  - int msgget(key_t key, int flag);

- msgctl() - message control operations
  - int msgctl(int msqid, int cmd, struct msqid_ds* buf);
  - IPC_STAT
  - IPC_SET
  - IPC_RMID
Message Control: msgsnd() and msgrcv()

- **msgsnd()** - send message
  - int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);

- **msgrcv()** – receive message
  - ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg);
  - type == 0: First message on queue is returned
  - type > 0: Return first message of type specified
  - type < 0: Message of lowest value less than or equal to the absolute value of type is returned
Semaphores

- To obtain a resource guarded by a semaphore
  - Test the semaphore that controls the resource
  - If the value of the semaphore is positive, decrement it and use resource
  - Else (value is 0), Process goes to sleep and starts process over
  - When finished, increment the value, if other processes are waiting they will be awakened
Semaphores cont.

- Semaphore (test, inc, dec) actions should be atomic

- Semaphore problems
  - Semaphore values are really a set of one or more values
  - Semaphore creation is not atomic
  - A semaphore remain on system if the process using them does not terminate it
Semaphores: semget(), semctl(), and semopt()

- semget() - get a semaphore set identifier
  - int semget(key_t key, int nsems, int semflg);

- semctl() - semaphore control operations
  - int semctl(int semid, int semnum, int cmd, ...);

- semopt() – semaphore operation
  - int semopt(int semid, struct sembuf semoparray[], size_t nops);
Shared Memory: shmget() and shmctl()

- Shared memory is the fastest IPC
- shmget() - allocates a shared memory segment
  - int shmget(key_t key, size_t size, int flag);
- shmctl() - shared memory control
  - int shmctl(int shmid, int cmd, struct shmid_ds* buf);
Shared Memory: shmat() and shmdt()

- shmat() – attach shared memory segment to calling process
  - void *shmat(int shmid, const void *shmaddr, int shmflg);

- shmdt() – detach shared memory from process
  - int shmdt(const void *shmaddr);
```c
#include "apue.h"
#include <sys/shm.h>

#define ARRAY_SIZE 40000
#define MALLOC_SIZE 100000
#define SHM_SIZE 100000
#define SHM_MODE 0600    /* user read/write */

char    array[ARRAY_SIZE];    /* uninitialized data = bss */

int main(void)
{
    int    shmid;
    char   *ptr, *shmptr;

    printf("array[] from %p to %p\n", (void *)&array[0],
                       (void *)&array[ARRAY_SIZE]);
    printf("stack around %p\n", (void *)&shmid);

    if ((ptr = malloc(MALLOC_SIZE)) == NULL)
        err_sys("malloc error");
    printf("allocated from %p to %p\n", (void *)ptr,
                       (void *)ptr+MALLOC_SIZE);

    if ((shmid = shmget(IPC_PRIVATE, SHM_SIZE, SHM_MODE)) < 0)
        err_sys("shmget error");
    if ((shmptr = shmat(shmid, 0, 0)) == (void *)-1)
        err_sys("shmat error");
    printf("shared memory attached from %p to %p\n", (void *)shmptr,
                       (void *)shmptr+SHM_SIZE);

    if (shmctl(shmid, IPC_RMID, 0) < 0)
        err_sys("shmctl error");

    exit(0);
}
```

Figure 15.31  Print where various types of data are stored
Figure 15.32  Memory layout on an Intel-based Linux system
POSIX semaphores

- POSIX semaphores vs XSI semaphores
  - POSIX allows for higher-performance implementation
  - POSIX interface is simpler
    - No semaphore set
    - Interfaces are patterned after file system operations
  - POSIX semaphores can be removed more gracefully
  - POSIX has named option
POSIX Semaphores: `sem_open()`, `sem_close()`, and `sem_unlink()`

- **sem_open()** - initialize and open a named semaphore
  - `sem_t *sem_open(const char *name, int oflag, ...)``

- **sem_close()** - close a named semaphore
  - `int sem_close(sem_t *sem)`

- **sem_unlink()** - remove a named semaphore
  - `int sem_unlink(const char* name)`
POSIX Semaphores: sem_trywait(), sem_wait(), and sem_timedwait()

- sem_wait() – lock semaphore or block
  - int sem_wait(sem_t *sem);

- sem_trywait() – lock semaphore or return error
  - int sem_trywait(sem_t *sem);

- sem_timedwait() – lock semaphore or block for a specified time
  - int sem_timedwait(sem_t* restrict sem, const struct timespec* restrict tsptr);
... sem_post(), sem_init(), sem_destroy() and sem_getvalue()

- **sem_post()** - unlock a semaphore
  - int sem_post(sem_t *sem);

- **sem_init()** - initialize an unnamed semaphore
  - int sem_init(sem_t *sem, int pshared, unsigned int value);

- **sem_destroy()** - destroy an unnamed semaphore
  - int sem_destroy(sem_t *sem);

- **sem_getvalue()** - get the value of a semaphore
  - int sem_getvalue(sem_t *sem, int *sval);