Lecture 19. Interprocess Communication

Multiple processes can be created with `fork()` and `exec()`. One way to exchange information across processes was to give them access to an open file. Another way is through interprocess communication (IPC).

IPC supports the following implementations:

- pipes
- message queues
- semaphores
- shared memory
- sockets
- streams

The following table shows a bullet • where basic functionality is supported, and UDS where the feature is provided through UNIX domain sockets. The only forms of IPC supported for IPC between processes on different hosts are sockets and streams.

IPC type	**SUS**	**FreeBSD 5.2.1**	**Linux 2.4.22**	**Mac OS X 10.3**	**Solaris 9**
half-duplex pipes	•	(full)	•	•	(full)
FIFOs	•	•	•	•	•
full-duplex pipes	allowed	•,UDS	opt, UDS	UDS	•, UDS
named full-duplex pipes	XSI option	UDS	opt, UDS	UDS	•, UDS
message queues	XSI	•	•		•
semaphores	XSI	•	•	•	•
shared memory	XSI	•	•	•	•
sockets	•	•	•	•	•
STREAMS	XSI option		opt		•

Pipes

Pipes are the oldest form of IPC. They have two limitations:

1. They have been historically mostly half-duplex (data flows in one direction only)

2. They can only be used b/w processes with a common ancestor. Usually b/w a parent and a child. FIFOs don't have this limitation. Sockets don't have either.

A pipe is created by calling the `pipe()` function:

```
#include <unistd.h>

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

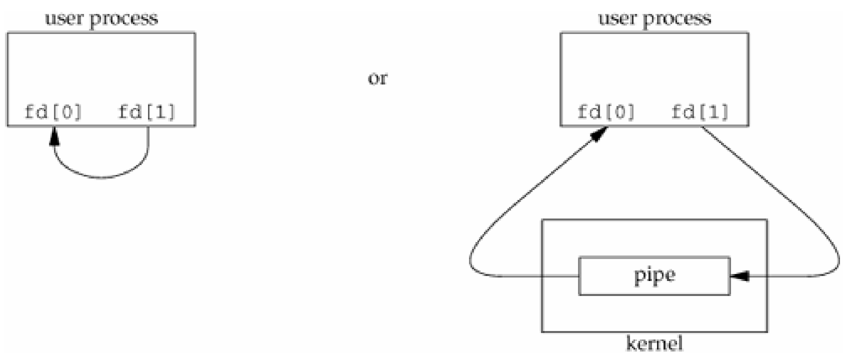
Returns 0 if OK, -1 if error.

The reference argument *`fd[2]`* returns two file descriptors:

- *`fd[0]`*—is open for reading from the pipe
- *`fd[1]`*—is open for writing to the pipe

The output of *`fd[1]`*, which writes to the pipe, is the input of *`fd[0]`*, which reads from the pipe.

The following diagram shows the two ends of the pipe in case of a single process (left) and of a process and data flow to the kernel (right).



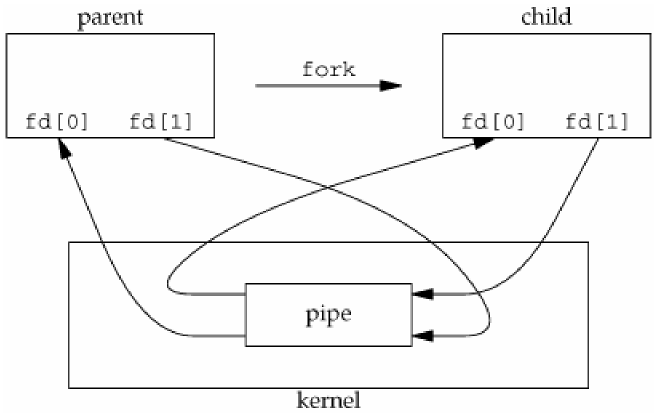
The `fstat()` function returns a FIFO file type for the file descriptor of either end of a pipe. A pipe can be tested with the S_ISFIFO macro.

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

int fstat(int fd, struct stat *buf);
```

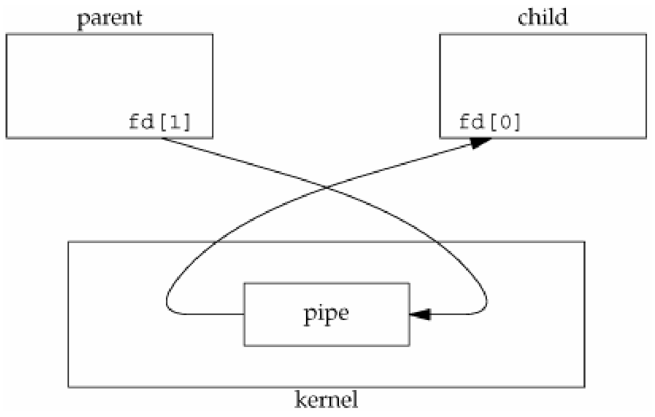
Returns 0 if OK, -1 if error.

A pipe is used between a parent process and its child. The parent usually calls `pipe()`, then calls `fork()`, establishing an IPC channel from itself to the child.

The following diagram shows an IPC channel between a parent and a child.



If the pipe is only one-way, from the parent to the child, the diagram becomes:



The following program shows the creation of a one-way pipe between the parent and the child. The parent writes to the pipe (to its *`fd[1]`*), closing its incoming channel (its *`fd[0]`*). The child read from the pipe (from its *`fd[0]`*), closing its outgoing channel (its *`fd[1]`*).

The data on the pipe in this example is just a simple string. To show the contents of this pipe, the "incoming" data on the pipe end in the child was captured in a type `char *` and displayed at the command line with `write()` to STDOUT_FILENO:

```
#include <unistd.h> /* for pipe(), fork() */
#include <stdlib.h> /* for malloc() */
#include <stdio.h>

#define MAXLINE 1024

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

     pipe(fd);
     pid=fork();
     if (pid>0)     /* parent */
     {
          close(fd[0]); /* close incoming */
          char *line_out=malloc(20*sizeof(char));
          line_out="Adriana Wise";
          write(fd[1], line_out, 20);
          printf("Outgoing: %s\n", line_out);
     }
     else      /* child */
     {
          close(fd[1]); /* close outgoing */
          char *line_in=malloc(MAXLINE);
          n=read(fd[0], line_in, MAXLINE);
          printf("Incoming: %s\n", line_in);
          //write(STDOUT_FILENO, line_in, n);
     }
return 0;
}
```

The output of this program shows the outgoing data (from the parent) and the same incoming data (to the child):

```
$ ./mypipe
Outgoing: Adriana Wise
Incoming: Adriana Wise
```

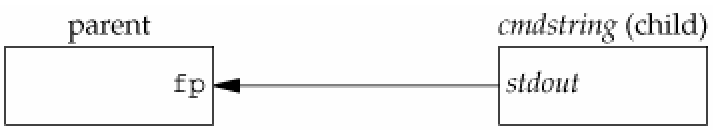
The `popen()` and `pclose()` Functions

These functions handle all the work that was done in the previous example explicitly:

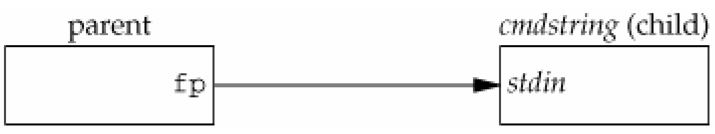
- creating a pipe
- forking a child
- closing the unused ends of the pipe
- executing a shell to run the command
- waiting for the command to terminate

```
#include <stdio.h>

FILE *popen(const char *cmdstring, const char *type);
                                    Returns a file pointer if OK, NULL if error.
int pclose(FILE *fp);
                                    Returns exit status of cmdstring, -1 if error.
```

`popen()` does a `fork()` and `exec()` to execute ***`cmdstring`***, and returns a `FILE *`. If type is `r`, `FILE *` is connected to the standard output of ***`cmdstring`*** (read from child), if it's `w`, `FILE *` is connected to the standard input of ***`cmdstring`*** (write to child). The return value of type `FILE *` is readable if type is `r`, and writeable if type is `w`.



In the first diagram, the data is read from the child and written into the parent.



In the second diagram, the data is read from the parent and written into the child.

The following program is an example of a call to `popen()` with an `r` argument, i.e. the parent process will read the data from the standard output of the command cmdstring executed in the child. In this case, the child executes the command `ls`, and the output in the parent is captured with `fgets()` from the `FILE *` stream:

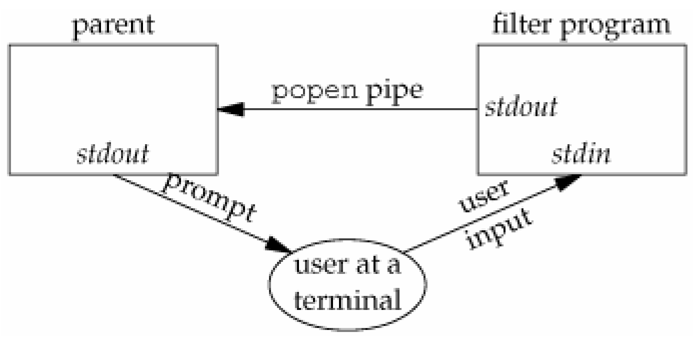
```
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>

#define COMMAND_LEN 20
#define DATA_SIZE 512

int main(void)
{
     FILE *STREAM;
     char command[COMMAND_LEN];
     char data[DATA_SIZE];

     //Execute a file listing
     sprintf(command, "ls");

     //Setup our pipe for reading and execute our command.
     STREAM=popen(command, "r");

     if (!STREAM)
     {
               fprintf(stderr, "Could not open pipe for output.
\n");
               return -1;
     }

     //Grab data from process execution
     while (fgets(data, DATA_SIZE, STREAM)!=NULL)
     {
          //Print grabbed data to the screen.
          fputs(data, stdout);
     }

     if (pclose(STREAM)!=0)
          fprintf(stderr,"Error: Failed to close command stream
\n");
return 0;
}
```

The output of this program is a listing of files in the CWD:

```
$ ./mypopen
mypipe
mypipe.c
mypipe2
mypipe2.c
```

```
mypipe3
mypipe3.c
mypopen
mypopen.c
test
test.c
```

Another example of `popen()` usage is in transforming the input of an application. The pipe arrangement is shown in the diagram below:



The filter program could `exec()` a program converting to uppercase, `myuppercase.c`:

```
#include <ctype.h>
#include <stdio.h>
#include <stdlib.h> /* for malloc() */
#include <string.h> /* for strcat() */

#define MAXLINE 1024

int main(void)
{
     char c;
     char *line=malloc(20*sizeof(MAXLINE));
     char *temp=malloc(sizeof(char));
     line="";

     while ((c=getchar())!=EOF)
     {
          if (islower(c))
          {
               c=toupper(c);
          }
          if (c=='\n')
               fflush(stdout);
          putchar(c);
     }
return 0;
}
```

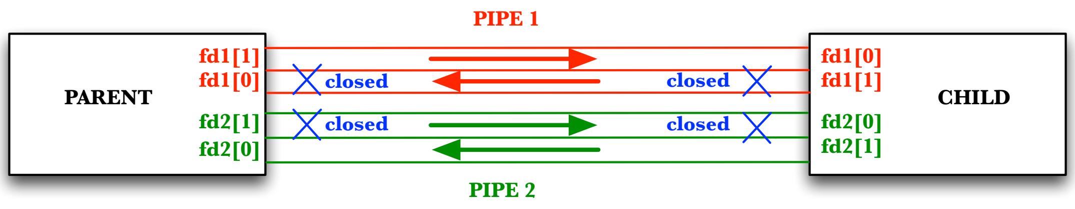
The filter program itself would call `popen()` with *`cmdstring`* set to `./myuppercase`:

```
#include <stdio.h>

#define MAXLINE 1024

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

     STREAM=popen("./myuppercase", "r");
     if (STREAM==NULL)
          printf("Null stream.\n");
     fgets(line, MAXLINE, STREAM);
     fputs(line, stdout);
     pclose(STREAM);
return 0;
}
```

Coprocesses

A UNIX system filter reads from stdin and writes to stdout. Filters are connected linearly in shell pipelines. A filter becomes a coprocess when the same program generates the filter's input and reads the filter's output. It is a coprocess of the user program.

`popen()` gives a one-way pipe to the `stdin`/from the stdout of another process. A coprocess has two one-way (half) pipes to the other process, one to its stdin, and one from its stdout. However, each process must close the other half pipe explicitly. Typically, a parent process would open two half pipes to provide input/collect output from a child process. The next diagram shows the two pipes and the necessary closures.



The following program implements two coprocesses. `coprocess1` calculates the sum of two numbers supplied to it from `stdin` and outputs the result on `stdout`. `coprocess2` provides the numbers through `pipe1` and collects the result through `pipe2`. These coprocesses are implemented as a parent process (the number supplier and result collector), and a child process (which execs the sum calculator, specified as the executable of a separate program).

The sum calculator program is shown first:

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>

#define MAXLINE 1024

int main(void)
{
     int n;
     int *a=malloc(sizeof(int));
     int *b=malloc(sizeof(int));
     char line[MAXLINE];

     while ((n=read(STDIN_FILENO, line, MAXLINE))>0)
     {
          line[n]=0;
          if (sscanf(line, "%d%d", a, b)==2)
          {
               sprintf(line, "%d\n", (*a)+(*b));
               n=strlen(line);
               write(STDOUT_FILENO, line, n);
          }
     }
return 0;
}
```

The program creating the two processes and the two pipes between them:

```
#include <fcntl.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <string.h>

#define MAXLINE 1024

void handler(int);

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

     signal(SIGPIPE, handler);
     if (pipe(fd1)<0 || pipe(fd2)<0)
          printf("Pipe error.\n");

     pid=fork();
     if (pid>0) /* parent */
     {
          printf("Parent: pid=%d\n", pid);
          close(fd1[0]); /* close pipe1 incoming */
          close(fd2[1]); /* close pipe2 outgoing */

          while (fgets(line, MAXLINE, stdin)!=NULL)
          {
               n=strlen(line);
               write(fd1[1], line, n); /* write the 2 numbers to
pipe1 outgoing */
               n=read(fd2[0], line, MAXLINE); /* read result
from pipe2 incoming */
               line[n]=0; /* NULL terminate */
               if (fputs(line, stdout)==EOF)
                    printf("fputs() error.\n");
               printf("fputs() puts %s\n", line);
          }
          exit(0);
     }
     else /* child */
     {
          printf("Child: pid=%d\n", pid);
          close(fd1[1]); /* close pipe1 outgoing */
          close(fd2[0]); /* close pipe2 incoming */

          int n=read(fd1[0], line, n); /* read from pipe1 
incoming */
          printf("Reading from pipe1 incoming: %d\n", n);
          if (fd1[0]!=STDIN_FILENO)
          {
               if (dup2(fd1[0], STDIN_FILENO)!=STDIN_FILENO)
                    printf("dup2() error.\n");
               close(fd1[0]); /* close pipe1 incoming */
          }

          if (fd2[1]!=STDOUT_FILENO)
          {
               if (dup2(fd2[1], STDOUT_FILENO)!=STDOUT_FILENO)
                    printf("dup2() error.\n");
               close(fd2[1]); /* close pipe2 outgoing */
          }

          int m=execl("./mycoprocess1", "mycoprocess1", (char 
*)0);
          //int m=execv("/Users/awise/Stevens/Lecture19/
mycoprocess1", NULL);
          printf("m=%d\n", m);
     }
return 0;
}

void handler(int signo)
{
     printf("SIGPIPE caught.\n");
exit(1);
}
```

The output:

```
$ ./mycoprocess2
Parent: pid=5911
Child: pid=0
3 5
n=2
8
fputs() puts 8
^C
```

FIFOs

FIFOs are also called **named pipes**. Creating a FIFO is similar to creating a file. FIFOs have pathnames in the file system:

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

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

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

Both return 0 if OK, -1 if error.

Arguments:

- *path*—filepath string;
- *mode*—same argument as for the `open()` function;
- *fd*—file descriptor.

There are three cases:

1. If *path* specifies an absolute pathname, *fd* is ignored, and `mkfifoat()` behaves like `mkfifo()`;
2. If *path* specifies a relative pathname, and *fd* is a valid file descriptor for an open directory, the pathname is evaluated relative to the directory specified by *fd*;
3. If *path* specifies a relative pathname, and *fd*=AT_FDCWD, the pathname is evaluated relative to the CWD, and `mkfifoat()` behaves like `mkfifo()`.

Once a FIFO is created, it must be opened with `open()`. All other normal file I/O functions (`close()`, `read()`, `write()`, `unlink()`) also work with FIFOs.

The nonblocking flag to `open()` (O_NONBLOCK) affects other processes' access to the contents of the FIFO:

- without O_NONBLOCK, an `open()` with read-only will be blocked until another process opens the FIFO for writing. An `open()` for write-only will be blocked until another process opens the FIFO for reading.
- with O_NONBLOCK, an `open()` with read-only returns immediately. An open for write-only returns `-1` with errno set to ENXIO if no process reads the FIFO.

FIFOs are used for:

1. Shell commands to pass data from one shell pipeline to another;

2. As rendez-vous points in client-server applications to pass data between clients and servers.

The following program shows a fifo comm link between a parent and a child process:

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

#define MAX_BUF 1024

int main()
{
     int fd;
     char *myfifo="/Users/awise/Stevens/Lecture19/FIFO3.txt";
     pid_t pid;
     char *buf=malloc(sizeof(char *));

     /* create the FIFO (named pipe) */
     mkfifo(myfifo, 0666);

     pid=fork();

     if (pid>0)
     {
          /* write "Hi" to the FIFO */
          fd=open(myfifo, O_WRONLY);
          write(fd, "Hi", sizeof("Hi"));
          close(fd);

          /* remove the FIFO */
          //unlink(myfifo);
     }
     else
     {
             /* open, read, and display the message from the
FIFO */
          fd=open(myfifo, O_RDONLY);
          read(fd, buf, MAX_BUF);
          printf("Received: %s\n", buf);
          close(fd);
     }
return 0;
}
```

The output shows the contents of the FIFO:

```
$ ./myfifo3
Received: Hi
```

XSI IPC

The next three types of IPC:
- message queues
- semaphores
- shared memory

are called XSI IPC. Each IPC structure is identified by a non-negative integer identifier, as opposed to a file descriptor, as pipes and FIFOs. These identifiers are large integers: they are associated with the particular IPC structure and their size continues to increase until it reaches the maximum positive value for an integer, then wraps around to 0.

In addition to the identifier, which is an internal name, each IPC object also has a key, which is an external name under which the IPC object is known across processes. The data type for the key is `key_t`, a long integer.

There are three ways for a client and a server to rendez-vous at the same IPC structure:
1. The server can create a new IPC structure by specifying a key of IPC_PRIVATE and store the identifier for the client to obtain.
2. The server and client can agree on a key by specifying the key in a common header.
3. The server and client can agree on a pathname and project ID (a character value between 0 and 255) and call `ftok()` to convert these values into a key.

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

key_t ftok(const char *path, int id);
```

Returns key if OK, (key_t)-1 if error.

Arguments:

- `path`—pathname to an existing file;
- `id`—the IPC structure ID, of which only the lowest 8 bits are used when generating the key.

The key generated by `ftok()` is formed by taking parts of the `st_dev` and `st_ino` fields of the stat structure corresponding to the pathname in path, and by combining them with the project ID. The mapping is mostly surjective, but it does sometimes happen that two different sets of input values generate identical keys (because both i-node numbers and keys are stored in long integers).

Permission Structure

Each IPC structure has an `ipc_perm` structure, which defines the permissions and owner of the structure:

```
struct ipc_perm
{
     uid_t uid;      /* owner's effective user ID */
     gid_t gid;      /* owner's effective group ID */
     uid_t cuid;     /* creator's effective user ID */
     gid_t cgid;     /* creator's effective group ID */
     mode_t mode;    /* access modes */
     …
};
```

The structure is initialized when the IPC structure is created. Its members can be modified later by calling `msgctl()`, `semctl()`, and `shmctl()`.

The following table summarizes the permission bits for an XSI IPC structure:

Permission	**Bit**
user-read	0400
user-write (alter)	0200
group-read	0040
group-write (alter)	0020
other-read	0004
other-write (alter)	0002

Message Queues

A message queue is a linked list of messages stored within the kernel and identified by a message queue identifier. These are called in short queue, and queue ID.

A new queue is created or an existing queue is opened with `msgget()`. New messages are added to the end of the queue with `msgsnd()`. Every message has:

- a type field—a positive long integer
- a length—a non-negative integer
- data bytes.

Messages are retrieved from the queue with `msgrcv()`. They do not have to be reached for in a FIFO order, rather they can be referenced through their type field.

A queue has the following structure:

```
struct msqid_ds
{
     struct ipc_perm     msg_perm;
     msgqnum_t           msg_qnum; /* # of messages in queue */
     msglen_t            msg_qbytes; /* max # bytes in 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 */
     …
}
```

The function creating a queue or opening an existing queue is `msgget()`:

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

int msgget(key_t key, int flag);
```

Returns message queue ID if OK, -1 if error.

When a new queue is created, the following members of the `msqid_ds` structure are initialized:

- `ipc_perm` structure—permission structure initialized as on page 15
- `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime`, `msg_rtime`—all set to 0
- `msg_ctime`—set to current time
- `msg_qbytes`—set to system limit.

The non-negative queue ID return value is used with the remaining three message queue functions.

The `msgctl()` function (and its counterparts `semctl()` and `shmctl()` for semaphores, and shared memory, respectively) are the `ioctl()`-like functions for XSI IPC.

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

int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```
Returns 0 if OK, -1 if error.

Arguments:
`msqid`—message queue ID, identifying the queue
`cmd`—the command to be performed on the queue specified by *`msqid`*
`buf`—fill in the `msqid_ds` structure

The cmd can be one of the following:

IPC_STAT	Fetch the `msqid_ds` structure, storing it in *`buf`*
IPC_SET	Copy from *`buf`* to `msqid_ds` `msg_perm.uid`, `msg_perm.gid`, `msg_perm.mode`, and `msg_qbytes`
IPC_RMID	Remove the message queue and its data from the system.

These three commands are also provided for semaphores and shared memory.

Data is added to the queue with `msgsnd()`:

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

int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
```
Returns 0 if OK, -1 if error.

Arguments:
`msqid`—message queue ID
`ptr`—pointer to a long integer that contains the >0 int message type, followed by the message data
`nbytes`—size of message in bytes
`flag`—if the value of IPC_NOWAIT is specified, and the queue is full, `msgsnd()` returns immediately with EAGAIN. If 0 is specified, and the queue is full,

`msgsnd()` is blocked until there is room for the message in the queue, the queue is removed from the system (EIDRM is returned), or a signal is caught and its handler returns (EINTR is returned).

The ptr argument is a pointer to a mymesg user-defined structure:

```
struct mymesg
{
     long mtype; /* positive message type */
     char mtext[512]; /* message data, of length nbtyes */
};
```

Messages are retrieved from the queue with `msgrcv()`:

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

ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type,
               int flag);
```
Returns size of data part of message if OK, -1 if error.

The following program sends and receives a message using the message queue functions:

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

void MessageStat(struct msqid_ds buff);

int main(void)
{
     pid_t pid;
     key_t key;
     int msqid;
     int err;
     struct msqid_ds buf;
     typedef struct
     {
          long mtype;     /* positive message type */
          char *mtext; /* actual message data */
     } mymesg;
     mymesg messageSent, messageRcvd;
     ssize_t data;

     messageSent.mtype=10;
     messageSent.mtext="Adriana Wise";

     key=ftok("/Users/awise/Stevens/Lecture19/keyfile.txt", 1);
     printf("Generated key=%zd.\n", key);

     msqid=msgget(key, IPC_CREAT);
     printf("Generated message=%d.\n", msqid);

     err=msgctl(msqid, IPC_STAT, &buf);
     printf("msgctl() returned with err=%d.\n", err);
     printf("msgctl() returned with %d: %s.\n", errno,
strerror(errno));

     MessageStat(buf);

     err=msgsnd(msqid, &messageSent, 20, 0);

     data=msgrcv(msqid, &messageRcvd, 20, 10, 0);
     printf("Message received: %s\n", messageRcvd.mtext);
return 0;
}

void MessageStat(struct msqid_ds buff)
{
     printf("Owner's effective user ID: %d\n",
buff.msg_perm.uid);
     printf("Owner's effective group ID: %d\n",
buff.msg_perm.gid);
     printf("Creator's effective user ID: %d\n",
buff.msg_perm.cuid);
     printf("Creator's effective group ID: %d\n",
buff.msg_perm.cgid);
     printf("Access modes: %d\n", buff.msg_perm.mode);
}
```

The output shows that the message was received:

```
$ sudo ./mymessagequeue
Password:
Generated key=16886957.
Generated message=65536.
msgctl() returned with err=0.
msgctl() returned with 0: Undefined error: 0.
Owner's effective user ID: 501
Owner's effective group ID: 20
Creator's effective user ID: 501
Creator's effective group ID: 20
Access modes: 0
Message received: Adriana Wise
```

Semaphores

Semaphores are not a form of IPC like the others: pipes, FIFOs, and message queues. A semaphore is a counter used to provide access to a shared resource for multiple processes.

To obtain a shared resource (memory, file, data structure), a process does the following:

- tests the semaphore that controls the structure
- if the value of the semaphore is positive, uses the resource, and decrements the value of the semaphore by 1
- if the value of the semaphore is 0, goes to sleep until it gets >0
- once the process is done using the resource, the semaphore counter increments by 1. Any asleep process waiting for it is awakened

The testing for >0 value and decrementing are an atomic operation, implemented in the kernel.

Semaphores are implemented as sets:

1. A semaphore is defined as a set of multiple semaphore values. The number of semaphore values in the set is specified when the semaphore is created;
2. A semaphore is first created (with `semget()`), and then initialized (with `semctl()`). Thus, creating a semaphore and initializing its values are not an atomic operation;
3. A program that terminates must release its semaphores.

The structure maintained in the kernel for semaphore sets is `semid_ds`:

```
struct semid_ds
```

```
{
     struct ipc_perm     sem_perm; /* semaphore permissions */
     unsigned short      sem_nsems;/* # semaphores in set */
     time_t              sem_otime;/* last semop time */
     time_t              sem_ctime;/* last change time */
     …
};
```

Each semaphore is represented by a user-defined structure:

```
struct
{
     unsigned short semval; /* semaphore value, always >=0 */
     pid_t          sempid; /* pid of last process using it */
     unsigned short semncnt;/* # processes awaiting
                              semval>current value */
     unsigned short semzcnt;/* # processes awaiting semval==0 */
     …
};
```

A semaphore is created with `semget()`:

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

int semget(key_t key, int nsems, int flag);
```
Returns semaphore ID if OK, -1 if error.

When a semaphore set is created, members of structure `semid_ds` are initialized:

- `ipc_perm`—the mode member of this structure is initialized to the permission bits of flag
- `sem_otime`—is set to 0
- `sem_ctime`—is set to the current time
- `sem_nsems`—is set to `nsems` (# of semaphore values in the set)

The following function performs a series of commands on the semaphore set:

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

int semctl(int semid, int semnum, int cmd,
               /* union semun arg */);
```

Arguments:

semid—semaphore ID
semnum—# of semaphore values in the semaphore set
cmd—command from a list of 10 possibilities:

IPC_STAT	Retrieve the `semid_ds` structure for the semaphore set, storing it in the structure pointed to by `arg.buf`
IPC_SET	Set the `sem_perm.uid`, `sem_perm.gid`, `sem_perm.mode` from the `arg.buf` structure into the `semid_ds` structure
IPC_RMID	Remove the semaphore set from the system
GETVAL	Return the value of semval for the member *semnum*
SETVAL	Set the value of *semval*, specified by `arg.val`
GETPID	Return the value of *sempid* for the member *semnum*
GETNCNT	Return the value of *semncnt* for the member *semnum*
GETZCNT	Return the value of *semzcnt* for the member *semnum*
GETALL	Retrieve all the semaphore values in the set. They are stored in the array pointed to by `arg.array`
SETALL	Set all the semaphore values in the set to the values pointed to by `arg.array`

Return values:
GET*—the corresponding value
all others—0 if OK, -1 if failure, and errno set

The function `semop()` performs an array of operations on a semaphore set:

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

int semop(int semid, struct sembuf semoparray[], size_t nops);
```
Returns 0 if OK, -1 if error.