Advanced Programming in the UNIX Environment — *Inter Process Communication*

Hop Lee
hoplee@bupt.edu.cn

School of Information and Communication Engineering

Table of Contents I

Table of Contents II

Foreword

- Pipe(half duplex)
- FIFOs(named pipes)
- Message Queue
- Semaphore
- Shared Memory

Pipe I

- **Pipe** is the oldest form of UNIX IPC, and almost all the UNIX support it.
- Pipe has two limits
  1. half-duplex, the data can only transfer along one direction
  2. a pipe can only be used between processes that have the same ancestor.
- We will find that **FIFO** break the second limitation, and that **UNIX domain sockets** get around both.
- A pipe is created by calling the `pipe` function:

```
1 #include <unistd.h>
2 int pipe(int fd[2]);
```

Pipe II

- Two file descriptors are returned through the *fd* parameter: *fd*[0] is open for reading and *fd*[1] is open for writing. The output of *fd*[1] is the input of *fd*[0].

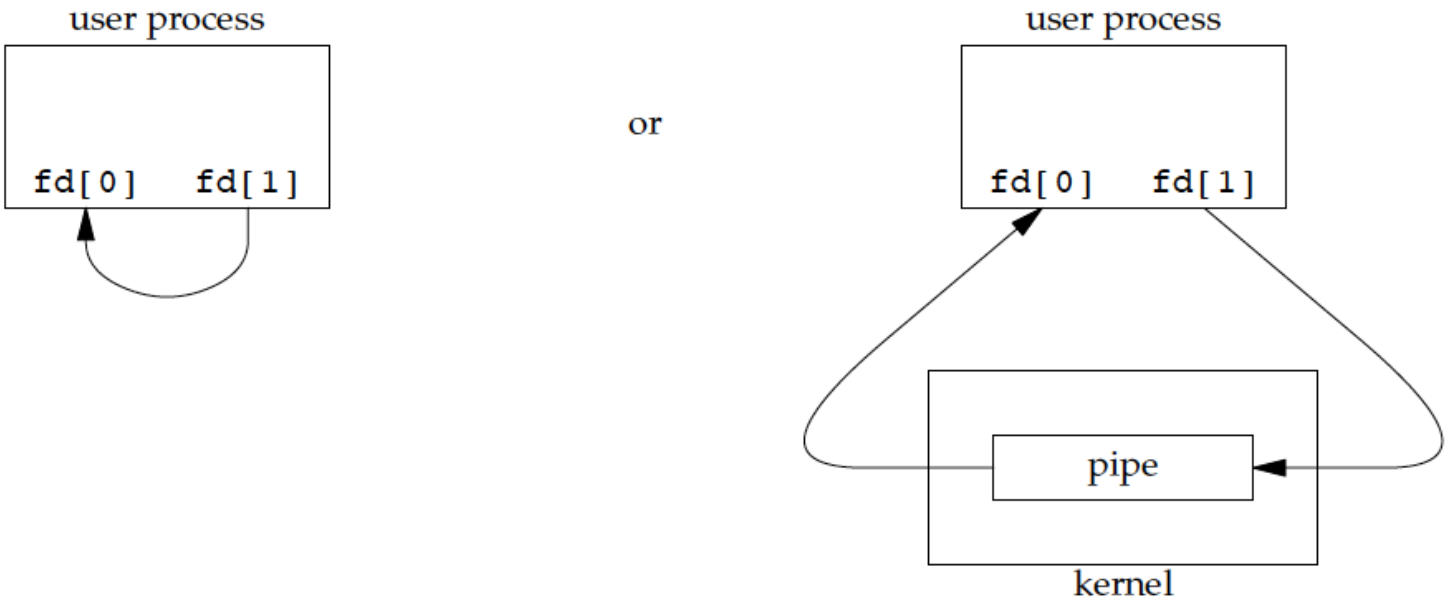

Figure: Two ways to view a half-duplex pipe

Pipe III

- The `fstat` function returns a file type of FIFO for the file descriptor of either end of a pipe. We can test for a pipe with the `S_ISFIFO` macro.
- Normally the process that calls `pipe` then calls `fork`. For a pipe from the parent to the child, the parent closes the read end of the pipe and the child closes the write end.

Pipe IV

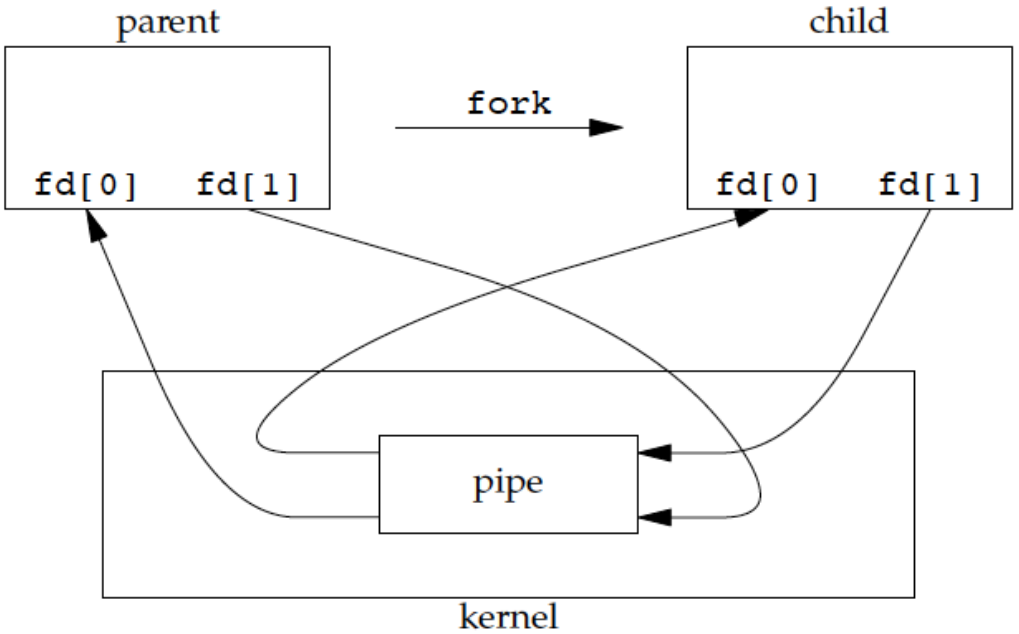

Figure: Half-duplex pipe after a `fork`

Pipe V

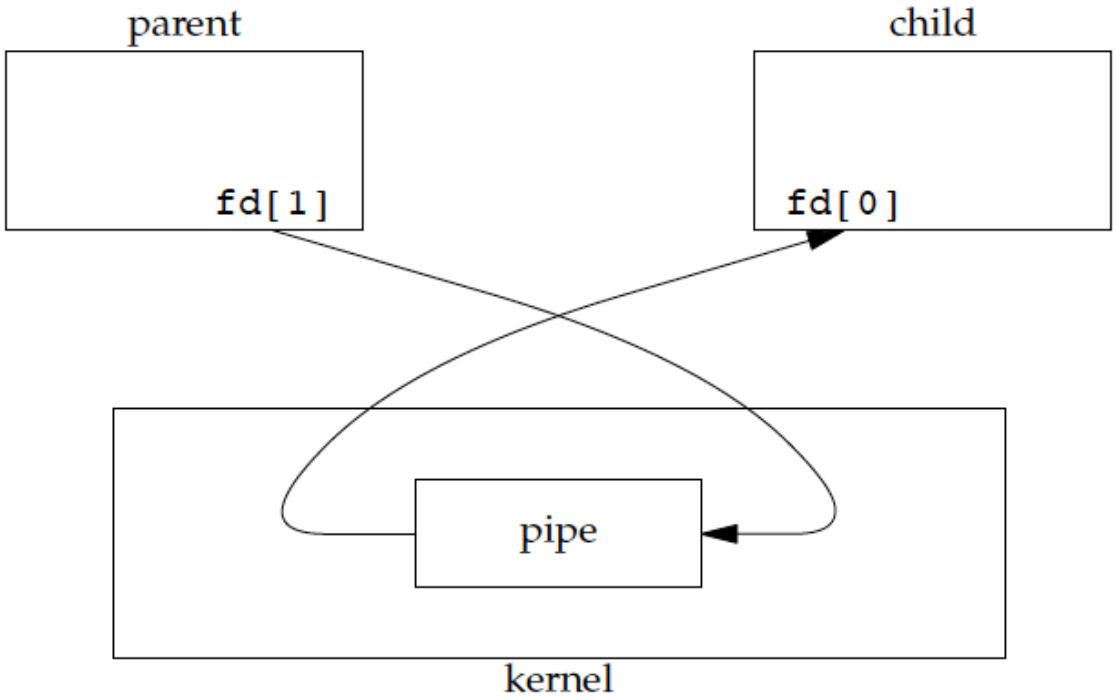

Figure: Pipe from parent to child

- When one end of a pipe is closed, two rules apply:

Pipe VI

1. If we `read` from a pipe whose write end has been closed, after all the data has been read, `read` returns 0 to indicate an end of file.
2. If we `write` to a pipe whose read end has been closed, the signal `SIGPIPE` is generated. If we either ignore the signal or catch it and return from the signal handler, `write` returns an error with `errno` set to `EPIPE`.

- Example (Figure 15.5, `ipc1/pipe1.c`).
- Example (Figure 15.6, `ipc1/pipe2.c`).
- Example (Figure 15.7, `ipc1/tellwait.c`) shows an implementation to let a parent and child synchronize using pipes.

Pipe VII

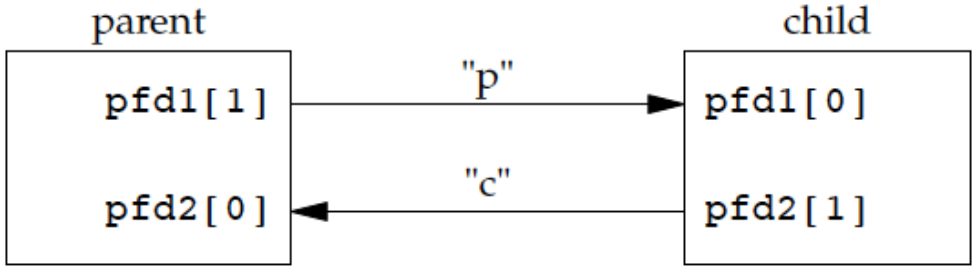

Figure: Using two pipes for parent–child synchronization

popen and pclose Functions I

- There are two functions can do the pipe creation and discard routine:

```
1 #include <stdio.h>
2 FILE *popen(const char *cmdstr, const char *type);
3 int pclose(FILE *fp);
```

- The popen does a fork and exec to execute the *cmdstr*, and returns a standard I/O file pointer.
- If the *type* is "r", the file pointer is connecter to the standard output of *cmdstr*.

popen and pclose Functions II

Figure: Result of `fp = popen(cmdstring, "r")`

- If the *type* is "`w`", the file pointer is connecter to the standard input of *cmdstr*.

Figure: Result of `fp = popen(cmdstring, "w")`

popen and pclose Functions III

- The pclose function closes the standard I/O stream, waits for the command to terminate, and returns the termination status of the shell. If the shell cannot be executed, the termination status returned by pclose is as if the shell had executed exit(127).
- The *cmdstr* is executed as:

```
1 sh -c cmdstr
```

  This means that the shell expands any of its special characters in *cmdstr*.
- Example (Figure 15.11, ipc1/popen2.c) redo the program from Figure 15.6, using popen.

popen and pclose Functions IV

- Example (Figure 15.12, ipc1/popen.c) shows our version of popen and pclose.
- One thing that popen is especially well suited for is executing simple filters to transform the input or output of the running command. Such is the case when a command wants to build its own pipeline.
- With the popen function, we can interpose a program between the application and its input to transform the input.

popen and pclose Functions V

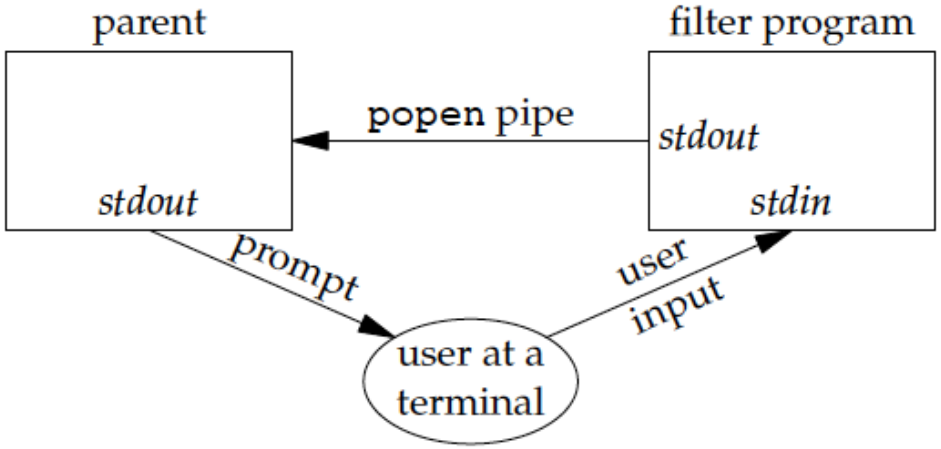

Figure: Transforming input using popen

- Example (Figure 15.14, ipc1/myuclc.c) shows a simple filter to demonstrate this operation which we then invoke from the program in Figure 15.15 (ipc1/popen1.c) using popen.

Coprocesses I

- A UNIX system **filter** is a program that reads from standard input and writes to standard output. Filters are normally 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.
- The program in Figure 15.17 (`ipc1/add2.c`) is a simple coprocess that reads two numbers from its standard input, computes their sum, and writes the sum to its standard output. We compile this program into `add2`.
- The example in Figure 15.18 (`ipc1/pipe4.c`) invokes the `add2` coprocess after reading two numbers from its standard input. The value from the coprocess is written to its standard output.

Coprocesses II

- If we replace the low-level I/O routines in `add2` with standard I/O (`ipc1/add2stdio.c`), Figure 15.18 won't work any more because the I/O buffering mechanism (p552).

FIFOs I

- **FIFO**s are also called **named pipe**. Pipes can be used only between related processes when a common ancestor has created the pipe. With FIFOs, however, unrelated process can exchange data.
- Creating a FIFO is similar to creating a file.

```
1 #include <sys/types.h>
2 #include <sys/stat.h>
3 int mkfifo(const char *pathname, mode_t mode);
4 int mkfifoat(int fd, const char *path, mode_t mode);
```

- Once we have created a FIFO, we can open it using `open`.
- There are two uses for FIFOs:

FIFOs II

1. FIFOs are used by shell command to pass data from one shell pipeline to another, without creating intermediate temporary files.
2. FIFOs are used in a client-server application to pass data between the clients and server.

- Example — Using FIFOs to Duplicate Output Streams
  - Whereas pipes can be used only for linear connections between processes, a FIFO has a name, so it can be used for nonlinear connections.

FIFOs III

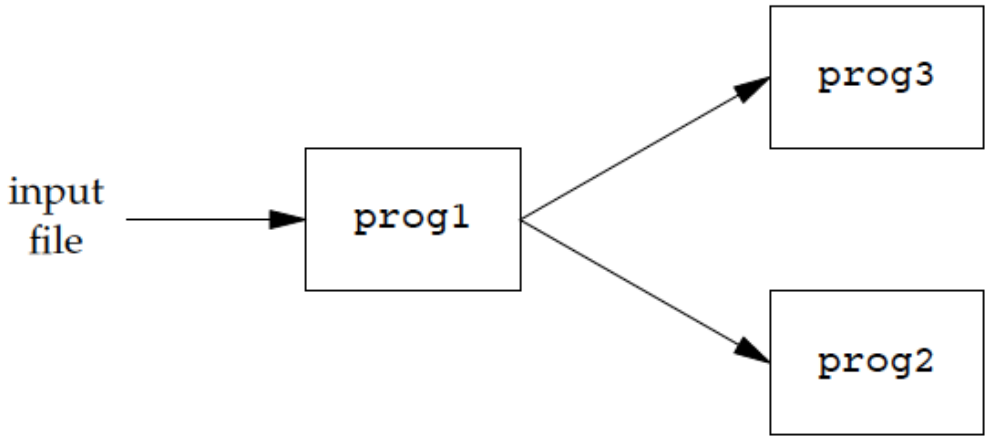

Figure: Transforming input using popen

- With a FIFO and the UNIX program `tee`(1), we can accomplish this procedure without using a temporary file.

```
1 mkfifo fifo1
2 prog3 < fifo1 &
3 prog1 < infile | tee fifo1 | prog2
```

FIFOs IV

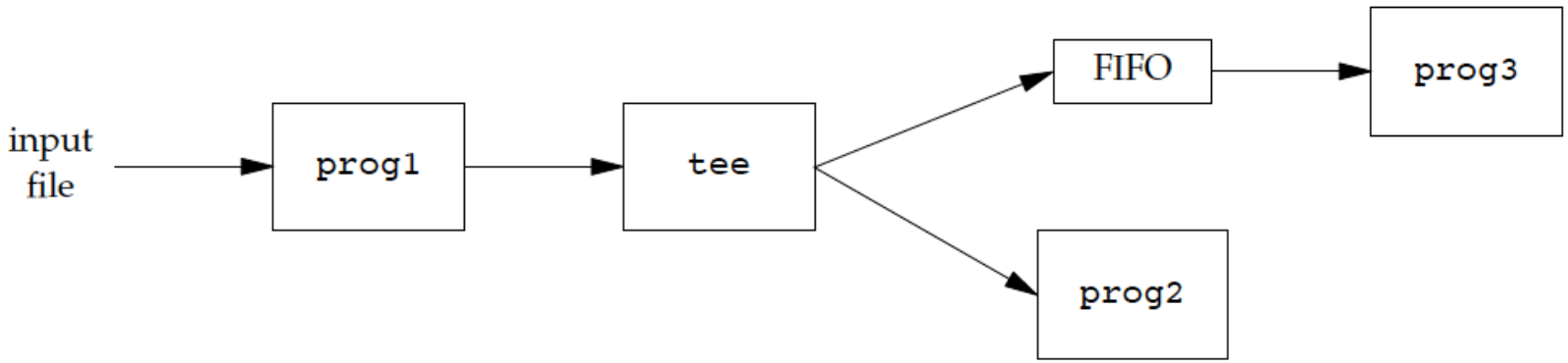

Figure: Using a FIFO and `tee` to send a stream to two different processes

- Example — Client-Server Communication Using a FIFO
  - If we have a server that is contacted by numerous clients, each client can write its request to a well-known FIFO that the server creates.

FIFOs V

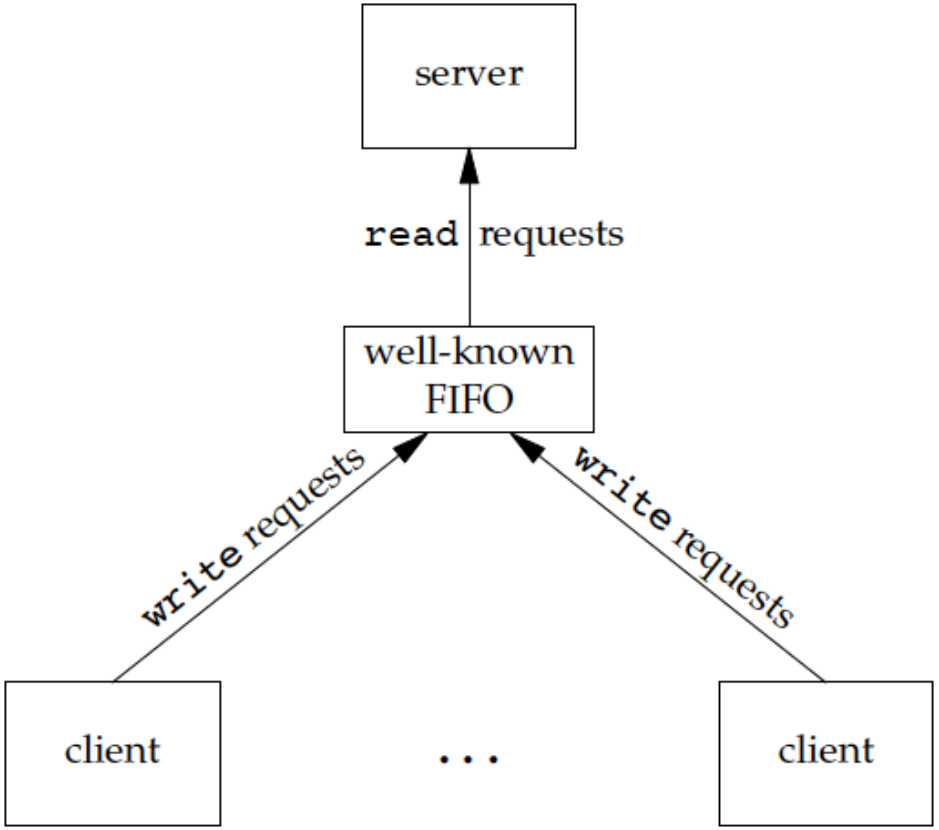

Figure: Clients sending requests to a server using a FIFO

FIFOs VI

- The problem in using FIFOs for this type of client-server communication is how to send replies back from the server to each client.
- One solution is for each client to send its pid with the request. The server then creates a unique FIFO for each client based on these pids.
- This arrangement works with sevaral flaws.

FIFOs VII

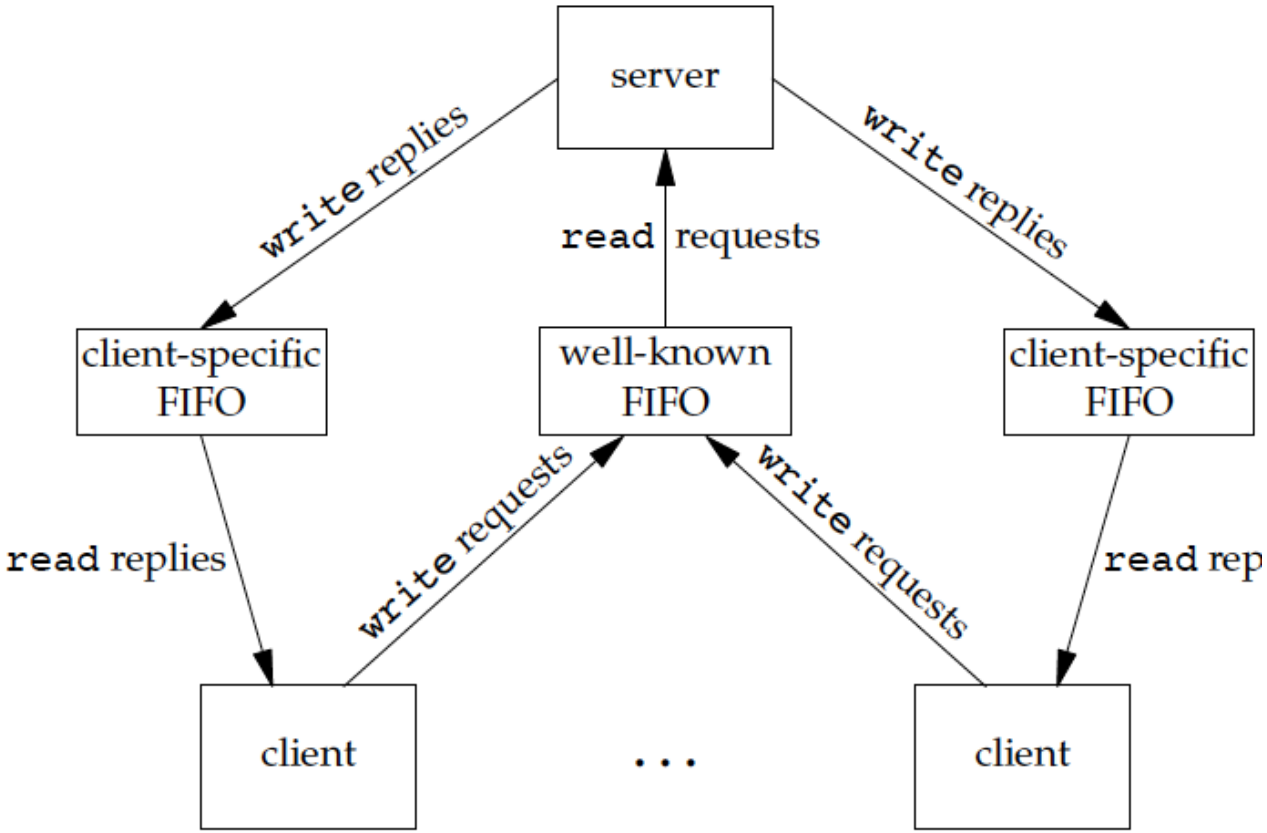

Figure: Client-server communication using FIFOs

XSI IPC

- The three types of IPC that we call **XSI IPC** — message queues, semaphores, and shared memory — have many similarities.
- In this section we cover these similar features, before looking at the specific functions for each of the three IPC types.

Identifiers and Keys I

- Each **IPC structure** in the kernel is referred to by a nonnegative integer **identifier**.
- To send or fetch a message to or from a message queue, for example, all we need know is the identifier for the queue.
- Unlike the file descriptor, IPC identifiers are not small integers.
- Whenever an IPC structure is being created, a **key** acts as an external name must be specified. The data type of this key is the primitive system data type `key_t`, which is often defined as a long integer in the header `sys/types.h`. This key is converted into an identifier by the kernel.
- There are various ways for a client and a server to rendezvous at the same IPC structure.

Identifiers and Keys II

1. The server can create a new IPC structure by specifying a key of `IPC_PRIVATE` and store the returned identifier somewhere (such as a file) for the client to obtain. The key `IPC_PRIVATE` guarantees that the server creates a new IPC structure. The disadvantage of this technique is that additional file system operations are required.
2. The client and the server can agree on a key by defining the key in a common header, for example. The server then creates a new IPC structure specifying this key. The problem with this approach is that it's possible for the key to already be associated with an IPC structure, in which case the `get` function (`msgget`, `semget`, or `shmget`) returns an error. The server must handle this error, deleting the existing IPC structure, and try to create it again.

Identifiers and Keys III

3. The client and the server can agree on a pathname and **project ID** (the project ID is a character value between 0 and 255) and call the function `ftok` to convert these two values into a key. This key is then used in step 2. The only service provided by `ftok` is a way of generating a key from a pathname and project ID.

```
1 #include <sys/ipc.h>
2 key_t ftok(const char *path, int id);
```

- The *path* argument must refer to an existing file. Only the lower 8 bits of *id* are used when generating the key.

Identifiers and Keys IV

- The key created by `ftok` is usually formed by taking parts of the `st_dev` and `st_ino` fields in the `stat` structure corresponding to the given pathname and combining them with the project ID.

Permission Structure I

- XSI IPC associates an `ipc_perm` structure with each IPC structure. This structure defines the permissions and owner and includes at least the following members:

```
1 struct ipc_perm {
2   uid_t  uid;  /* owner's effective user ID */
3   gid_t  gid;  /* owner's effective group ID */
4   uid_t  cuid; /* creator's effective user ID */
5   gid_t  cgid; /* creator's effective group ID */
6   mode_t mode; /* access modes */
7   ...
8 };
```

- All the fields are initialized when the IPC structure is created. At a later time, we can modify the `uid`, `gid`, and `mode` fields by calling `msgctl`, `semctl`, or `shmctl`.

Configuration Limits

- All three forms of XSI IPC have build-in limits that we may encounter.
- Most of these limits can be changed by reconfiguring the kernel.

Advantages and Disadvantages I

- A fundamental problem with XSI IPC is that the IPC structures are systemwide and do not have a reference count.
- Another problem with XSI IPC is that these IPC structures are not known by names in the file system. We can't access them and modify their properties with the functions we described in Chapters 3 and 4.
- Since these forms of IPC don't use file descriptors, we can't use the multiplexed I/O functions (`select` and `poll`) with them. This makes it harder to use more than one of these IPC structures at a time or to use any of these IPC structures with file or device I/O.

Advantages and Disadvantages II

- Using identifiers allows a process to send a message to a message queue with a single function call (`msgsnd`), whereas other forms of IPC normally require an `open`, `write`, and `close`.
- These XSI IPC are reliable, flow controlled, and record oriented, and that they can be processed in other than fifo order.

Message Queue I

- A message queues is a linked list of messages stored within the kernel and identified by a message queue identifier.
- Each queue has the following `msqid_ds` structure associated with it:

```
1  struct msqid_ds {
2    struct ipc_perm msg_perm;   /* see Section 15.6.2 */
3    msgqnum_t       msg_qnum;   /* # of messages on queue */
4    msglen_t        msg_qbytes; /* max # of bytes on queue */
5    pid_t           msg_lspid;  /* pid of last msgsnd() */
6    pid_t           msg_lrpid;  /* pid of last msgrcv() */
7    time_t          msg_stime;  /* last-msgsnd() time */
8    time_t          msg_rtime;  /* last-msgrcv() time */
9    time_t          msg_ctime;  /* last-change time */
10   ...
11 };
```

This structure defines the current status of the queue.

Message Queue II

- A queue is created, or an existing queue is opened by `msgget`.

```
1 #include <sys/msg.h>
2 int msgget(key_t key, int flag);
```

- When a new queue is created, the following members of the `msqid_ds` structure are initialized.
  - The `ipc_perm` structure is initialized as described in Section 15.6.2. The `mode` member of this structure is set to the corresponding permission bits of *flag*. These permissions are specified with the values from Figure 15.24.
  - `msg_qnum`, `msg_lspid`, `msg_lrpid`, `msg_stime`, and `msg_rtime` are all set to 0.
  - `msg_ctime` is set to the current time.
  - `msg_qbytes` is set to the system limit.

Message Queue III

- The `msgctl` function performs various operations on a queue. This function and the related functions for semaphores and shared memory (`semctl` and `shmctl`) are the `ioctl`-like functions for XSI IPC (i.e., the garbage-can functions).

```
1 #include <sys/msg.h>
2 int msgctl(int msqid, int cmd, struct msqid_ds *buf)
```

- The *cmd* argument specifies the command to be performed on the queue specified by *msqid*.

  `IPC_STAT` Fetch the `msqid_ds` structure for this queue, storing it in the structure pointed to by *buf*.

Message Queue IV

`IPC_SET` Copy the following fields from the structure pointed to by *buf* to the `msqid_ds` structure associated with this queue: `msg_perm.uid`, `msg_perm.gid`, `msg_perm.mode`, and `msg_qbytes`. Only the superuser can increase the value of `msg_qbytes`.

`IPC_RMID` Remove the message queue from the system and any data still on the queue. This removal is immediate. Any other process still using the message queue will get an error of `EIDRM` on its next attempted operation on the queue.

We'll see that these three commands are also provided for semaphores and shared memory.

Message Queue V

- New messages are added to the end of a queue by `msgsnd`.

```
1 #include <sys/msg.h>
2 int msgsnd(int msqid, const void *ptr,
3            size_t nbytes, int flag);
```

- Each message is composed of a positive long integer type field, a non-negative length (*nbytes*), and the actual data bytes.
- Messages are always placed at the end of the queue.
- The *ptr* points to a long integer that contains the positive integer message type, and it is immediately followed by the message data.

Message Queue VI

- The *flag* value of `IPC_NOWAIT` is similar to the nonblocking I/O flag for file I/O. If the message queue is full, specifying `IPC_NOWAIT` causes msgsnd to return immediately with an error of `EAGAIN`; otherwise, we are blocked.
- Messages are retrieved from a queue by `msgrcv`.

```
1 #include <sys/msg.h>
2 ssize_t msgrcv(int msqid, void *ptr,
3                size_t nbytes, long type, int flag);
```

- The *ptr* argument points to a long integer (where the message type of the returned message is stored) followed by a data buffer for the actual message data.
- *nbytes* specifies the size of the data buffer.

Message Queue VII

- If the returned message is larger than *nbytes* and the `MSG_NOERROR` bit in flag is set, the message is truncated. Otherwise, an error of `E2BIG` is returned instead.
- The *type* argument lets us specify which message we want.

  $type == 0$ The first message on the queue is returned.

  $type > 0$ The first message on the queue whose message type equals *type* is returned.

  $type < 0$ The first message on the queue whose message type is the lowest value less than or equal to the absolute value of *type* is returned.
- We can specify a *flag* value of `IPC_NOWAIT` to make the operation nonblocking,
- Example — Timing Comparison of Message Queues and Full-Duplex Pipes, p565.

Semaphore I

- A **semaphore** is a counter used to provide access to a shared data object for multiple processes.
- To obtain a shared resource a process needs to do the following:
  1. Test the semaphore that controls the resource,
  2. If the value of the semaphore is positive, the process can use the resource. The process decrements the semaphore value by 1.
  3. If the value of the semaphore is 0, the process goes to sleep until the semaphore value is greater than 0. When the process wakes up it returns to step 1.

Semaphore II

- When a process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1. If any other processes are asleep, waiting for the semaphore, they are awakened.
- To implement semaphore correctly, the test of a semaphore's value and the decrementing of this value must be an atomic operation.
- In general, however, a semaphore can be initialized to any positive value, with the value indicating how many units of the shared resource are available for sharing.
- A common form of semaphore is called a **binary semaphore**. It controls a single resource, and its value is initialized to 1.
- Three features contribute to the unnecessary complication of XSI semaphores:

Semaphore III

1. A semaphore is not simply a single non-negative value.
2. The creation of a semaphore (`semget`) is independent of its initialization (`semctl`).
3. Since all forms of XSI IPC remain in existence even when no process is using them, how and when will a semaphore be released?

- The kernel maintains a `semid_ds` structure for each semaphore set:

```
1 struct semid_ds {
2   struct ipc_perm sem_perm;  /* see Section 15.6.2 */
3   unsigned short  sem_nsems; /* # of semaphores in set */
4   time_t          sem_otime; /* last-semop() time */
5   time_t          sem_ctime; /* last-change time */
6   ...
7 };
```

Semaphore IV

- Each semaphore is represented by an anonymous structure containing at least the following members:

```
1 struct {
2   unsigned short semval;  /* semaphore value, always >= 0 */
3   pid_t          sempid;  /* pid for last operation */
4   unsigned short semncnt; /* # processes awaiting semval>curval */
5   unsigned short semzcnt; /* # processes awaiting semval==0 */
6   ...
7 };
```

- When we want to use XSI semaphores, we first need to obtain a semaphore ID by calling the `semget` function.

```
1 #include <sys/sem.h>
2 int semget(key_t key, int nsems, int flag);
```

- When a new set is created, the following members of the `semid_ds` structure are initialized:

Semaphore V

- The `ipc_perm` structure is initialized as described before. The `mode` member of this structure is set to the corresponding permission bits of *flag*.
- `sem_otime` is set to 0.
- `sem_ctime` is set to the current time.
- `sem_nsems` is set to *nsems*.

- The number of semaphores in the set is *nsems*. If a new set is being created (typically by the server), we must specify *nsems*. If we are referencing an existing set (a client), we can specify *nsems* as 0.
- The `semctl` function is the catchall for various semaphore operations.

Semaphore VI

```
1 #include <sys/sem.h>
2 int semctl(int semid, int semnum, int cmd,
3               ... /* union semun arg */);
```

- The fourth argument is optional, depending on the command requested, and if present, is of type semun, a union of various command-specific arguments:

```
1 union semun {
2   int              val;    /* for SETVAL */
3   struct semid_ds *buf;    /* for IPC_STAT and IPC_SET */
4   unsigned short  *array; /* for GETALL and SETALL */
5 };
```

Semaphore VII

- The *cmd* argument specifies one of the ten commands (p568) to be performed on the set specified by *semid*. The five commands that refer to one particular semaphore value use *semnum* to specify one member of the set. The value of *semnum* is between 0 and $nsems - 1$, inclusive.
- The function `semop` atomically performs an array of operations on a semaphore set.

```
1 #include <sys/sem.h>
2 int semop(int semid, struct sembuf semoparray[],
3           size_t nops);
```

- The semoparray argument is a pointer to an array of semaphore operations, represented by `sembuf` structures:

Semaphore VIII

```
1 struct sembuf {
2   unsigned short sem_num; /* member # in set */
3   short          sem_op;  /* operation (<0, 0, or >0) */
4   short          sem_flg; /* IPC_NOWAIT, SEM_UNDO */
5 };
```

- The *nops* argument specifies the number of operations (elements) in the array.
- There are detailed discussion about `sem_op` values in p569-570.
- Example — Timing Comparison of Semaphores, Record Locking, and Mutexes, p570-571.

Shared Memory I

- Shared memory allows two or more processes to share a given region of memory. This is the *fastest* form of IPC because the data does not need to be copied between the client and server.
- The only trick in using shared memory is synchronizing access to a given region among multiple processes.
- Often semaphore, record locking, and mutexes are used to synchronize shared memory access.
- The XSI shared memory differs from memorymapped files in that there is no associated file, they are anonymous segments of memory.
- The kernel maintains a structure with at least the following members for each shared memory segment:

Shared Memory II

```
1  struct shmid_ds {
2    struct ipc_perm shm_perm;    /* see Section 15.6.2 */
3    size_t          shm_segsz;   /* size of segment in bytes */
4    pid_t           shm_lpid;    /* pid of last shmop() */
5    pid_t           shm_cpid;    /* pid of creator */
6    shmatt_t        shm_nattch;  /* # of current attaches */
7    time_t          shm_atime;   /* last-attach time */
8    time_t          shm_dtime;   /* last-detach time */
9    time_t          shm_ctime;   /* last-change time */
10   ...
11 };
```

- The first function called is usually shmget, to obtain a shared memory identifier.

```
1 #include <sys/shm.h>
2 int shmget(key_t key, size_t size, int flag);
```

Shared Memory III

- When a new segment is created, the following members of the `shmid_ds` structure are initialized.
  - The `ipc_perm` structure is initialized as described before. The `mode` member of this structure is set to the corresponding permission bits of *flag*.
  - `shm_lpid`, `shm_nattch`, `shm_atime`, and `shm_dtime` are all set to 0.
  - `shm_ctime` is set to the current time.
  - `shm_segsz` is set to the *size* requested.
- When a new segment is created, the contents of the segment are initialized with zeros.
- The `shmctl` function is the catchall for various shared memory operations:

Shared Memory IV

```
1 #include <sys/shm.h>
2 int shmctl(int shmid, int cmd,
3             struct shmid_ds *buf );
```

- The *cmd* argument specifies one of the five commands (p573) to be performed, on the segment specified by *shmid*.
- Once a shared memory segment has been created, a process attaches it to its address space by calling `shmat`:

```
1 #include <sys/shm.h>
2 void *shmat(int shmid, const void *addr, int flag);
```

Shared Memory V

- The value returned by `shmat` is the address at which the segment is attached, or -1 on error. If `shmat` succeeds, the kernel will increment the `shm_nattch` counter in the `shmid_ds` structure.
- When we're done with a shared memory segment, we call `shmdt` to detach it. The identifier remains in existence until some process (often a server) specifically removes it by calling `shmctl` with a command of `IPC_RMID`.

```
1 #include <sys/shm.h>
2 int shmdt(const void *addr);
```

- Figure 15.31 (`ipc1/tshm.c`) prints some information on where one particular system places various types of data.

Shared Memory VI

- Note that the shared memory segment is placed well below the stack.
- Example — Memory Mapping of `/dev/zero`. Figure 15.33, `ipc1/devzero.c`.
- Example — Anonymous Memory Mapping, p578.

Shared Memory VII

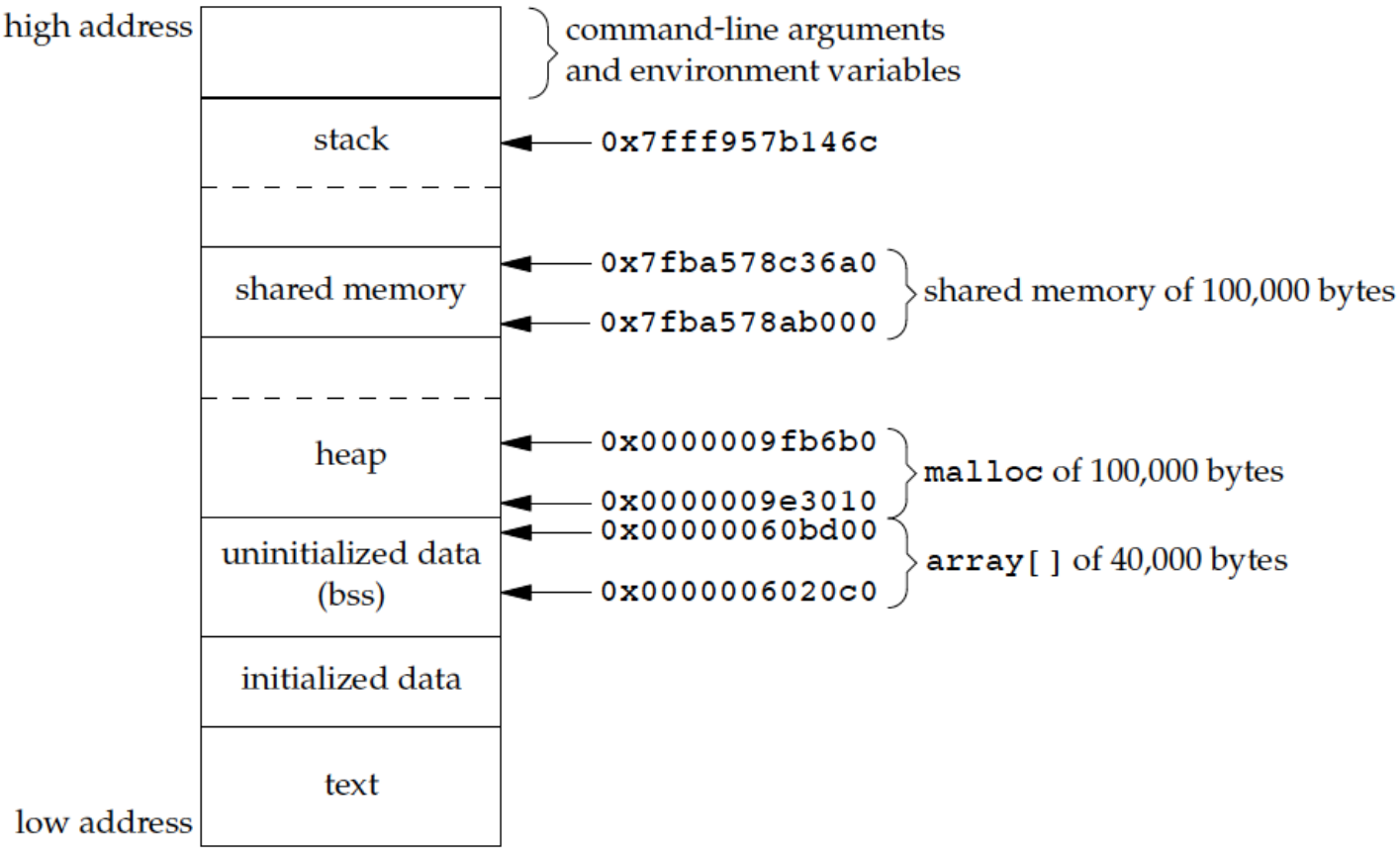

Figure: Memory layout on an Intel-based Linux system

POSIX Semaphores I

- The POSIX semaphore interfaces were meant to address several deficiencies with the XSI semaphore interfaces:
  - The POSIX semaphore interfaces allow for higher-performance implementations compared to XSI semaphores.
  - The POSIX semaphore interfaces are simpler to use: there are no semaphore sets, and several of the interfaces are patterned after familiar file system operations.
  - The POSIX semaphores behave more gracefully when removed. With POSIX semaphores, operations continue to work normally until the last reference to the semaphore is released.
- POSIX semaphores are available in two flavors: named and unnamed. They differ in how they are created and destroyed, but otherwise work the same.

POSIX Semaphores II

- To create a new named semaphore or use an existing one, we call the `sem_open` function:

```
1 #include <semaphore.h>
2 sem_t *sem_open(const char *name, int oflag,
3                 ... /* mode_t mode,
4                 unsigned int value */ );
```

- To promote portability, we must follow certain conventions when selecting a semaphore name:
  - The first character in the name should be a slash (/).
  - The name should contain no other slashes to avoid implementation-defined behavior.
  - The maximum length of the semaphore name is implementation defined.

POSIX Semaphores III

- The `sem_open` function returns a semaphore pointer that we can pass to other semaphore functions when we want to operate on the semaphore. When we are done with the semaphore, we can call the `sem_close` function to release any resources associated with the semaphore.

```
1 #include <semaphore.h>
2 int sem_close(sem_t *sem);
```

- If our process exits without having first called `sem_close`, the kernel will close any open semaphores automatically.
- If we call `sem_close`, the semaphore value is unaffected.
- To destroy a named semaphore, we can use the `sem_unlink` function:

POSIX Semaphores IV

```
1 #include <semaphore.h>
2 int sem_unlink(const char *name);
```

- It removes the name of the semaphore. If there are no open references to the semaphore, then it is destroyed. Otherwise, destruction is deferred until the last open reference is closed.
- Unlike with XSI semaphores, we can only adjust the value of a POSIX semaphore by one with a single function call.
- When a binary semaphore has a value of 1, we say that it is **unlocked**; when it has a value of 0, we say that it is **locked**.
- To decrement the value of a semaphore, we can use either the `sem_wait` or `sem_trywait` function:

POSIX Semaphores V

```
1 #include <semaphore.h>
2 int sem_trywait(sem_t *sem);
3 int sem_wait(sem_t *sem);
```

- With the `sem_wait` function, we will block if the semaphore count is 0. We won't return until we have successfully decremented the semaphore count or are interrupted by a signal. We can use the `sem_trywait` function to avoid blocking.
- A third alternative is to block for a bounded amount of time. We can use the `sem_timedwait` function for this purpose:

POSIX Semaphores VI

```
1 #include <semaphore.h>
2 #include <time.h>
3 int sem_timedwait(sem_t *restrict sem,
4 const struct timespec *restrict tsptr);
```

- The *tsptr* argument specifies the absolute time when we want to give up waiting for the semaphore.
- To increment the value of a semaphore, we call the `sem_post` function:

```
1 #include <semaphore.h>
2 int sem_post(sem_t *sem);
```

POSIX Semaphores VII

- If a process is blocked in a call to `sem_wait` (or `sem_timedwait`) when we call `sem_post`, the process is awakened
- When we want to use POSIX semaphores within a single process, it is easier to use unnamed semaphores. To create an unnamed semaphore, we call the `sem_init` function:

```
1 #include <semaphore.h>
2 int sem_init(sem_t *sem, int psh,
3               unsigned int value);
```

- The *psh* argument indicates if we plan to use the semaphore with multiple processes. The *value* argument specifies the initial value of the semaphore.

POSIX Semaphores VIII

- We need to declare a variable of type `sem_t` and pass its address to `sem_init` for initialization.
- When we are done using the unnamed semaphore, we can discard it by calling the `sem_destroy` function:

```
1 #include <semaphore.h>
2 int sem_destroy(sem_t *sem);
```

- After calling `sem_destroy`, we can't use any of the semaphore functions with *sem* unless we reinitialize it by calling `sem_init` again.
- One other function is available to allow us to retrieve the value of a semaphore, `sem_getvalue` function:

POSIX Semaphores IX

```
1 #include <semaphore.h>
2 int sem_getvalue(sem_t *restrict sem,
3                  int *restrict valp);
```

- On success, the integer pointed to by the *valp* argument will contain the value of the semaphore.
- The example in Figure 15.35 (`ipc1/slock.c`) shows an implementation of a semaphore-based mutual exclusion primitive.

Client-Server Properties I

- The simplest type of relationship is to have the client `fork` and `exec` the desired server. Two half-duplex pipes can be created before the `fork` to allow data to be transferred in both directions.
- With FIFOs, an individual per-client FIFO is also required if the server is to send data back to the client. If the client-server application sends data only from the client to the server, a single well-known FIFO suffices.
- Multiple possibilities exist with message queues.
  1. A single queue can be used between the server and all the clients, using the type field of each message to indicate the message recipient.

Client-Server Properties II

2. Alternatively, an individual message queue can be used for each client.
   One problem with this technique is that each client-specific queue usually has only a single message on it, this seems wasteful of a limited systemwide resource. Another problem is that the server has to read messages from multiple queues. Neither `select` nor `poll` works with message queues.

- Either of these two techniques using message queues can be implemented using shared memory segments and a synchronization method (a semaphore or record locking).
- The problem with this type of client-server relationship is for the server to identify the client accurately.
- With message queues, there is no portable way to obtain the effective user ID by giving the process ID.

Client-Server Properties III

- We'll see that using a the socket subsystem can make the kernel provide the effective user ID and effective group ID of the client

The End

The End of Chapter 12.