CS631 - Advanced Programming in the UNIX Environment
Interprocess Communication I

Department of Computer Science
Stevens Institute of Technology
Jan Schaumann
jschauma@cs.stevens.edu
https://stevens.netmeister.org/631/

System V IPC

Three types of *asynchronous* IPC originating from System V:

- Semaphores
- Shared Memory
- Message Queues

All three use *IPC structures*, referred to by an *identifier* and a *key*; all three are (necessarily) limited to communication between processes on one and the same host.

Since these structures are not known by name, special system calls (`msgget(2)`, `semop(2)`, `shmat(2)`, etc.) and special userland commands (`ipcrm(1)`, `ipcs(1)`, etc.) are necessary.

System V IPC: Semaphores

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:

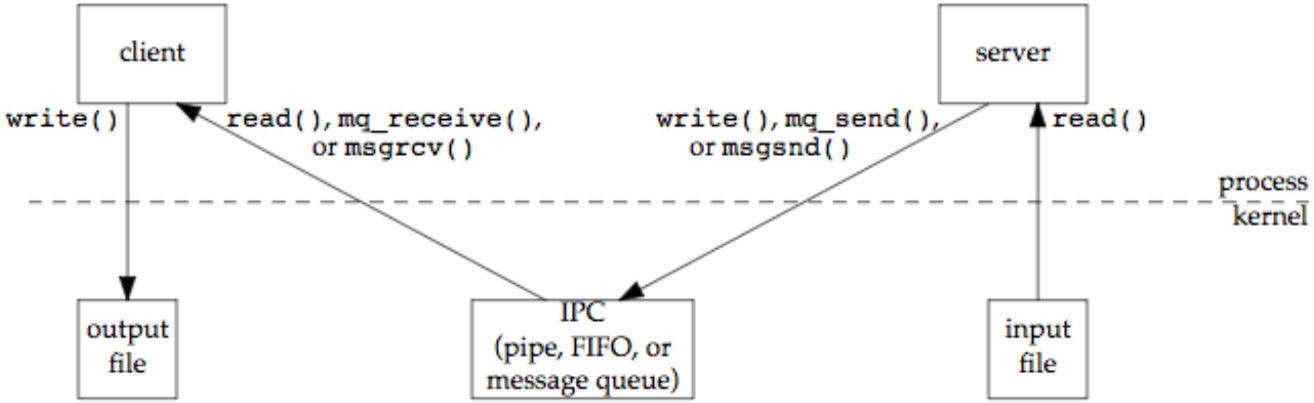
1. Test semaphore that controls the resource.
2. If value of semaphore $>$ 0, decrement semaphore and use resource; increment semaphore when done
3. If value == 0 sleep until value $>$ 0

Semaphores are obtained using `semget(2)`, properties controlled using `semctl(2)`, operations on a semaphore performed using `semop(2)`.

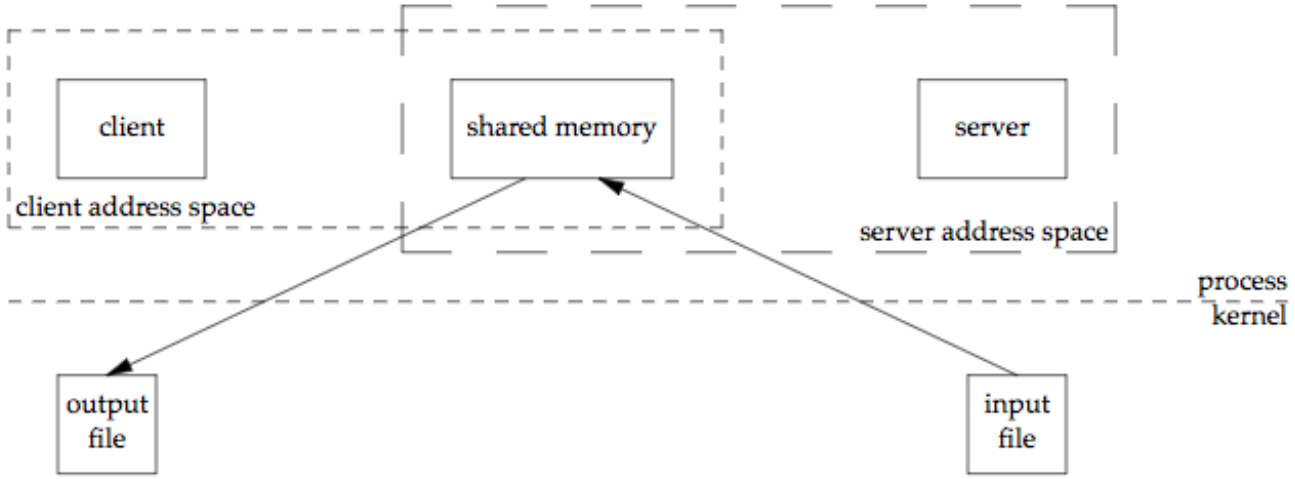
System V IPC: Semaphores

```
$ cc -Wall semdemo.c
1$ ./a.out

2$ ./a.out

$ ipcs -s

$ ipcrm -s 1234

1$ ./a.out
^Z

2$ ./a.out
```

IPC data flow



System V IPC: Shared Memory



System V IPC: Shared Memory

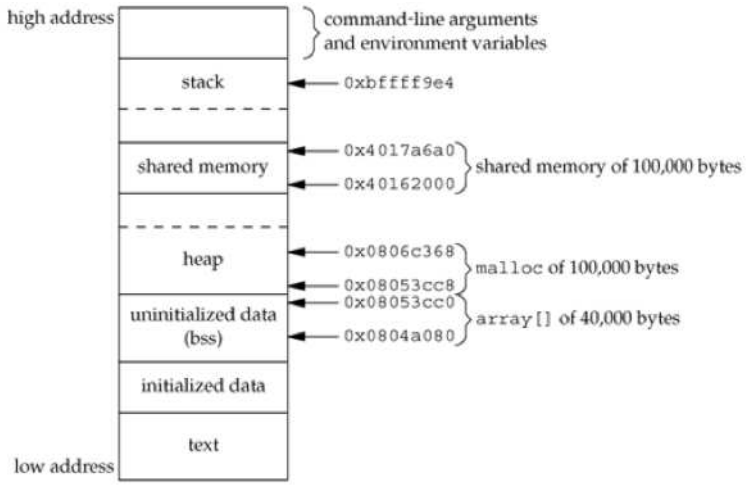
- fastest form of IPC
- access to shared region of memory often controlled using semaphores
- obtain a shared memory identifier using `shmget(2)`
- catchall for shared memory operations: `shmctl(2)`
- attach shared memory segment to a processes address space by calling `shmat(2)`
- detach it using `shmdt(2)`

System V IPC: Shared Memory

```
$ cc -Wall shmdemo.c
$ ./a.out "Cow says: 'Moo!'"
$ ./a.out
$ ipcs -m
```

System V IPC: Shared Memory

```
$ cc -Wall memory-layout.c
$ ./a.out
array[] from 804a080 to 8053cc0
stack around bffff9e4
malloced from 8053cc8 to 806c368
shared memory attached from 40162000 to 4017a6a0
```



System V IPC: Message Queues

- linked list of messages stored in the kernel
- create or open existing queue using `msgget(2)`
- add message at end of queue using `msgsnd(2)`
- control queue properties using `msgctl(2)`
- receive messages from queue using `msgrcv(2)`

The message itself is contained in a user-defined structure such as

```
struct mymsg {
    long mtype;       /* message type */
    char mtext[512]; /* body of message */
};
```

System V IPC: Message Queues

```
$ cc -Wall msgsend.c -o msgsend
$ cc -Wall msgrecv.c -o msgrecv
$ ipcs -qo
$ ./msgsend 1 'Hello!'
$ ipcs -qo
$ ./msgsend 1 'How are you?'
$ ipcs -qo
$ ./msgrecv 1
$ ipcs -qo
$ ./msgrecv 1
$ ipcs -qo
$ ./msgrecv 1
^C
$ ipcs -q
$ ./msgsend 2
$ ipcrm -q <msqid>
```

POSIX Message Queues

`mq(3)` provides an real-time IPC interface similar to System V message queues. Notably:

- message queues are identified by a named identifier (no `ftok(3)` needed); may or may not be exposed in the file system (e.g. `/dev/mqueue`)
- `mq_send(3)` and `mq_receive(3)` allow both blocking and non-blocking calls
- `mq_send(3)` lets you specify a priority; equal priority messages are queued as a FIFO, but higher priority messages are inserted before those of a lower priority
- `mq(3)` provides an asynchronous notification mechanism: `mq_notify(3)`

POSIX Message Queues

```
$ cc $CFLAGS mqrecv.c -o mqrecv -lrt
$ cc $CFLAGS -DWAIT mqsend.c -o mqsend
$ ./mqrecv

$ ./mqsend bacon avocado

$ ./mqrecv wait

$ ./mqsend bacon avocado

$ cc $CFLAGS mqsend.c -o mqsend

$ ./mqsend bacon avocado
```

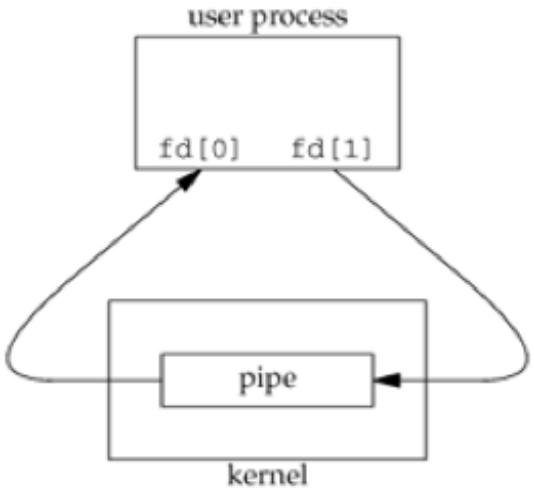
Pipes: `pipe(2)`

```
#include <unistd.h>

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

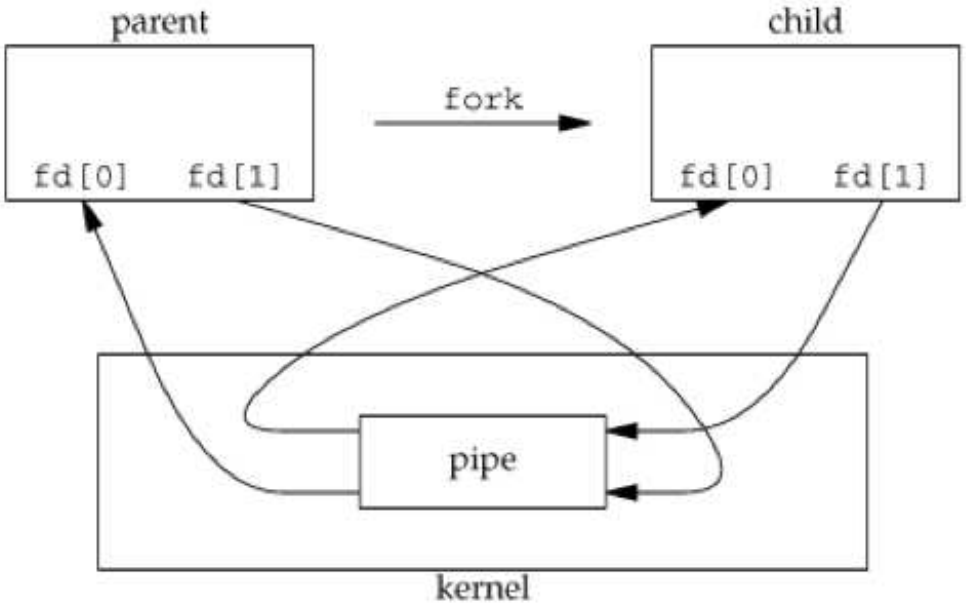
Returns: 0 if OK, -1 otherwise

- oldest and most common form of UNIX IPC
- half-duplex (on some versions full-duplex)

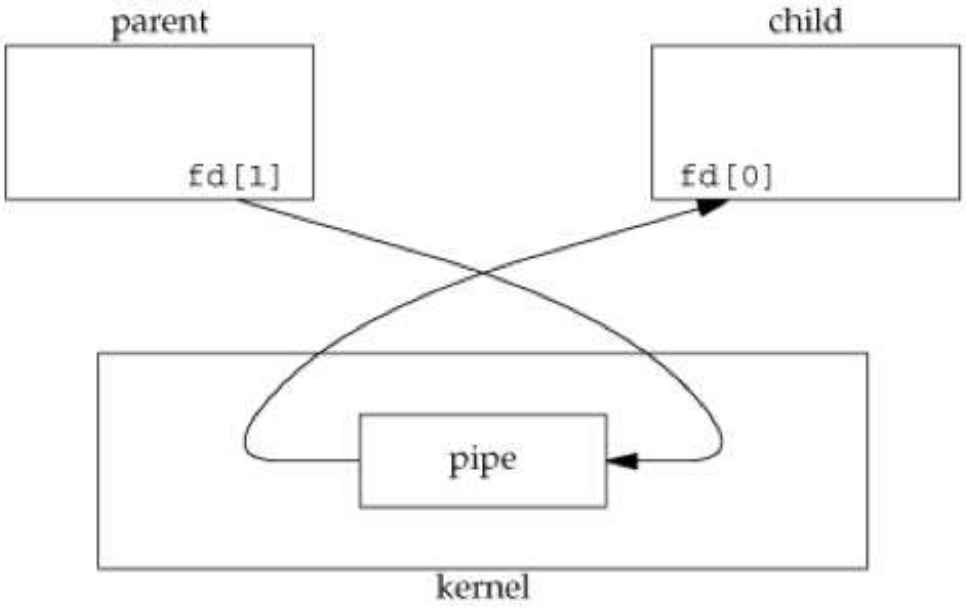
Pipes: `pipe(2)`



Pipes: `pipe(2)`



Pipes: `pipe(2)`



Pipes: `pipe(2)`

```
$ cc -Wall pipe1.c
$ ./a.out
P=> Parent process with pid 23988 (and its ppid 7474).
P=> Sending a message to the child process (pid 23989):
C=> Child process with pid 23989 (and its ppid 23988).
C=> Reading a message from the parent (pid 23988):
Hello child!  I'm your parent pid 23988!
$
```

Pipes: `pipe(2)`

```
$ cc -Wall pipe1.c
$ ./a.out
P=> Parent process with pid 23984 (and its ppid 7474).
P=> Sending a message to the child process (pid 23985):
C=> Child process with pid 23985 (and its ppid 1).
C=> Reading a message from the parent (pid 1):
Hello child!  I'm your parent pid 23984!
$
```

Pipes: `pipe(2)`

```
$ cc -Wall pipe1.c
$ ./a.out
P=> Parent process with pid 23986 (and its ppid 7474).
P=> Sending a message to the child process (pid 23987):
C=> Child process with pid 23987 (and its ppid 23986).
C=> Reading a message from the parent (pid 1):
Hello child!  I'm your parent pid 23986!
$
```

Pipes: `pipe(2)`

A more useful example: displaying some content using the user's preferred pager. (Look, Ma, no temporary files!)

```
$ cat pipe2.c | ${PAGER:-/usr/bin/more}
$ cc -Wall pipe2.c
$ echo $PAGER

$ ./a.out pipe2.c
[...]
^Z
$ ps -o pid,ppid,comm
  PID  PPID COMMAND
22306 26650 ./a.out pipe2.c
22307 22306 more
23198 26650 ps -o pid,ppid,comm
26650 26641 -ksh
$ fg
$ env PAGER=/bin/cat ./a.out pipe2.c
```

Pipes: `pipe(2)`

```
#include <unistd.h>

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

Returns: 0 if OK, -1 otherwise

- oldest and most common form of UNIX IPC
- half-duplex (on some versions full-duplex)
- can only be used between processes that have a common ancestor
- can have multiple readers/writers (`PIPE_BUF` bytes are guaranteed to not be interleaved)

Behavior after closing one end:

- `read(2)` from a pipe whose write end has been closed returns 0 after all data has been read
- `write(2)` to a pipe whose read end has been closed generates `SIGPIPE` signal. If caught or ignored, `write(2)` returns an error and sets `errno` to `EPIPE`.

Pipes: `popen(3)` and `pclose(3)`

```
#include <stdio.h>

FILE *popen(const char *cmd, const char *type);
```

Returns: file pointer if OK, `NULL` otherwise

```
int pclose(FILE *fp);
```

Returns: termination status *cmd* or -1 on error

- historically implemented using unidirectional pipe (nowadays frequently implemented using sockets or full-duplex pipes)
- *type* one of “r” or “w” (or “r+” for bi-directional communication, if available)
- *cmd* passed to `/bin/sh -c`

Pipes: `popen(3)` and `pclose(3)`

```
$ cc -Wall popen.c
$ echo $PAGER

$ ./a.out popen.c
[...]
$ env PAGER=/bin/cat ./a.out popen.c
[...]
$
```

Pipes: `popen(3)` and `pclose(3)`

```
$ cc -Wall popen.c
$ echo $PAGER

$ ./a.out popen.c
[...]
$ env PAGER=/bin/cat ./a.out popen.c
[...]
$ env PAGER=/bin/cat/foo ./a.out popen.c
sh: /bin/cat/foo: Not a directory
$ env PAGER="more; touch /tmp/boo" ./a.out popen.c
$ env PAGER="more; rm /etc/passwd 2>/dev/null" ./a.out popen.c
```

FIFOs: `mkfifo(2)`

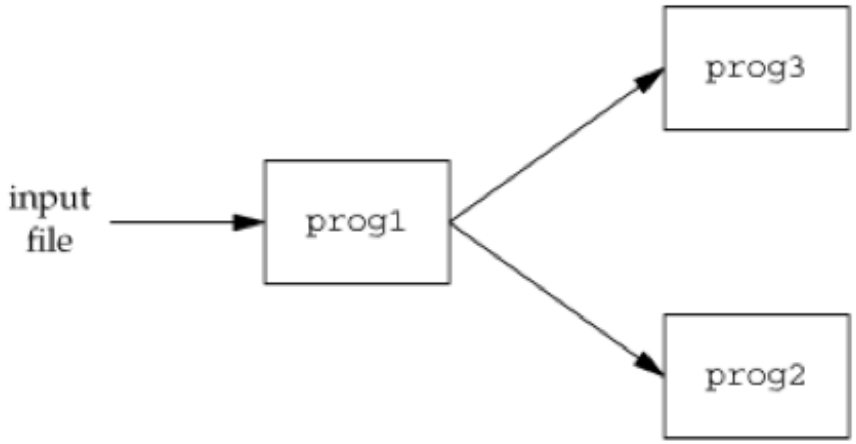
```
#include <sys/stat.h>

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

Returns: 0 if OK, -1 otherwise

- aka “named pipes”
- allows unrelated processes to communicate
- just a type of file – test for using `S_ISFIFO(st_mode)`
- *mode* same as for `open(2)`
- use regular I/O operations (ie `open(2)`, `read(2)`, `write(2)`, `unlink(2)` etc.)
- used by shell commands to pass data from one shell pipeline to another without creating intermediate temporary files

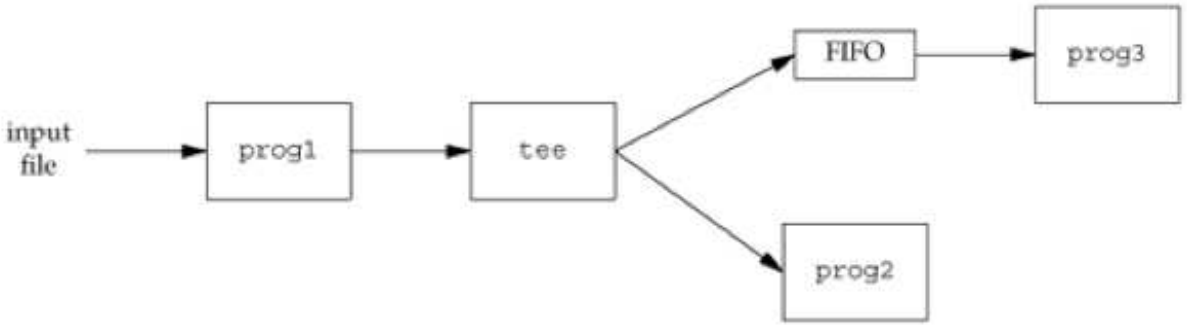
FIFOs: `mkfifo(2)`

Example: split input into sets



FIFOs: `mkfifo(2)`

Example: split input into sets




```
$ mkfifo fifo
$ grep pattern fifo > match &
$ gzcat file.gz | tee fifo | grep -v pattern > nomatch
```

In-class coding exercise

Implement the 'command' function:

`https://stevens.netmeister.org/631/f18-hw2.html`

Reading

- https://stevens.netmeister.org/631/ipctut.pdf
- https://stevens.netmeister.org/ipc.pdf
- https://beej.us/guide/bgipc/html/single/bgipc.html
- https://is.gd/M2dkju