Interprocess communication

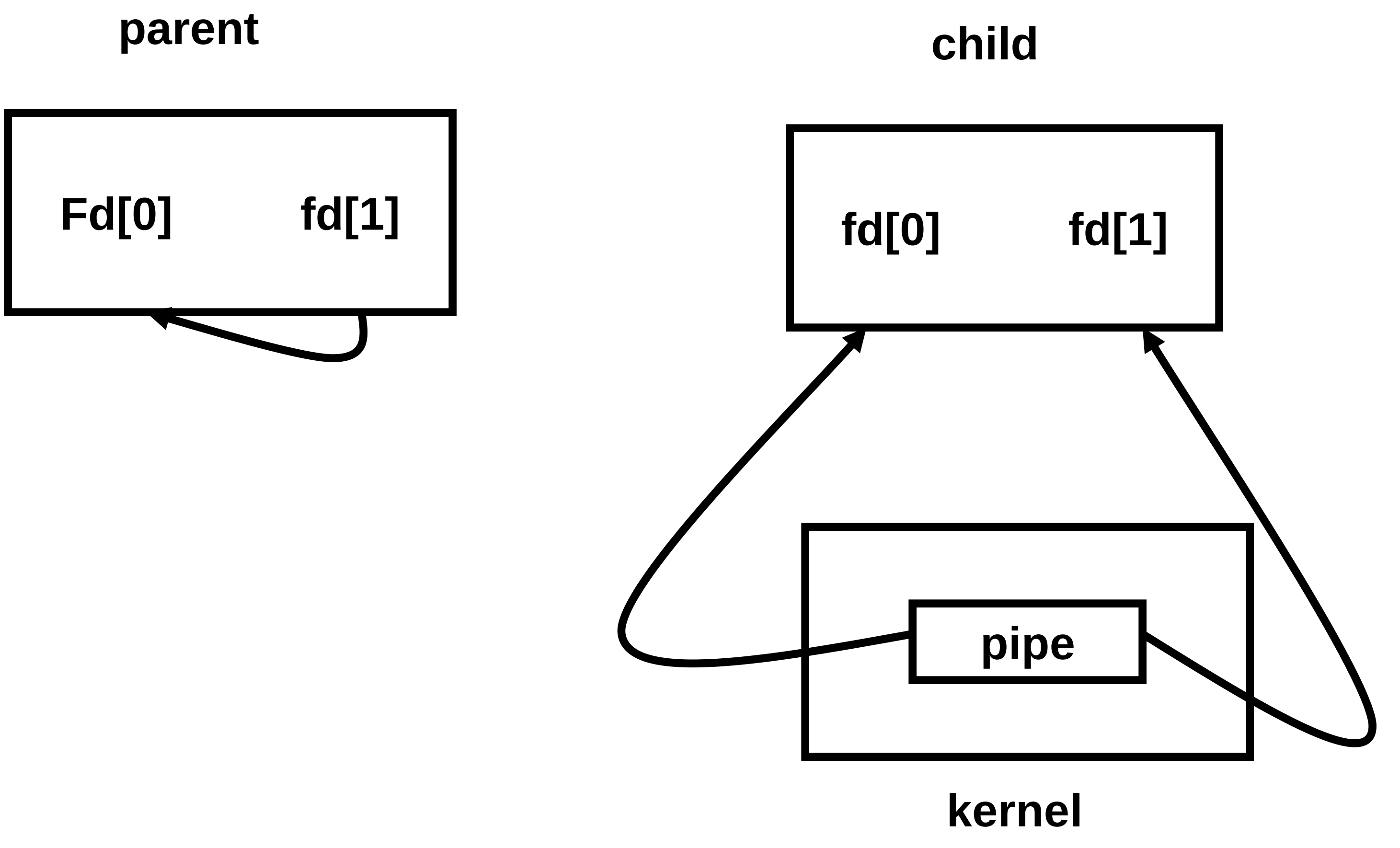
IPC type	POSIX.1	XPG3	V7	SVR2	SVR3.2	SVR4	4.3 BSD	4.3+ BSD
Pipes	•	•	•	•	•	•	•	•
FIFOs	•	•		•	•	•		•
Stream pipes					•	•	•	•
Named stream pipes					•	•	•	•

IPC type	POSIX.1	XPG3	V7	SVR2	SVR3.2	SVR4	4.3 BSD	4.3+ BSD
Message queues		•		•	•	•	•	• •
Semaphores		•		•	•	•		
Shared Memory		•		•	•	•		
Socket						•	•	•
Streams					•			

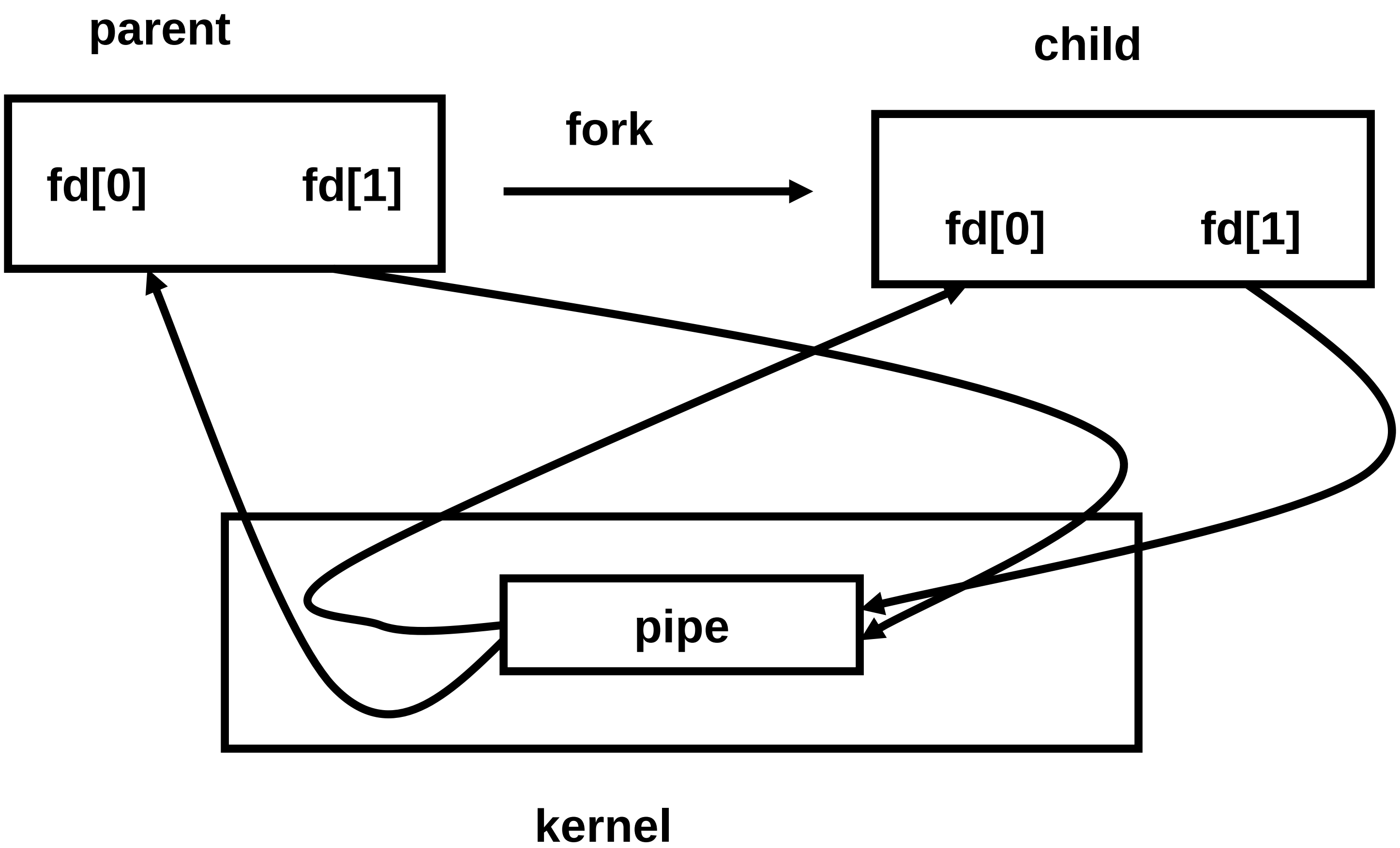
Pipes

- They are oldest form of IPC
- They are half-duplex. Data flows in only one direction
- They can be used only between processes that have a common ancestor

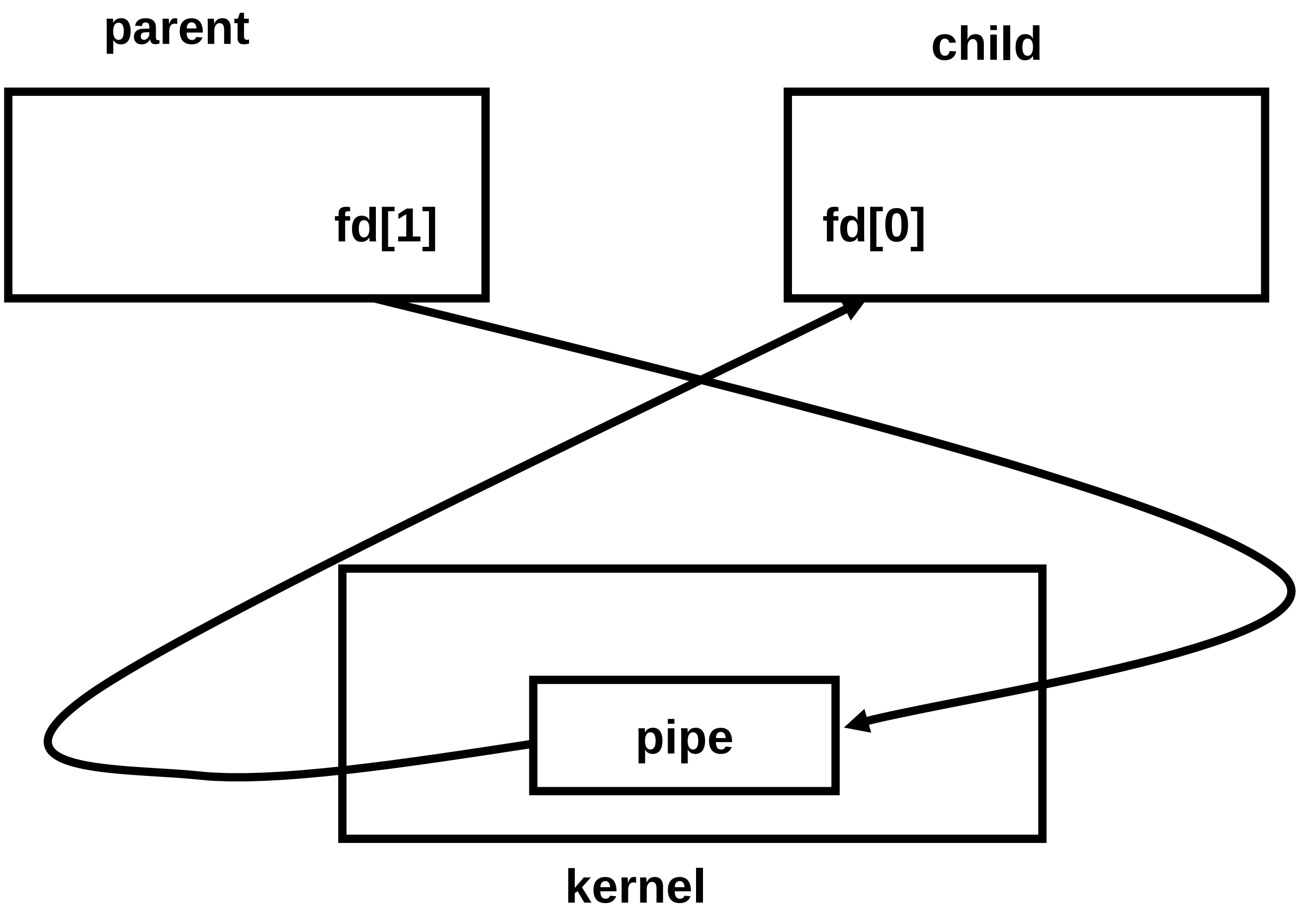
```
#include <unistd.h>
int pipe (int filedes[2]);
```


parent
Fd[0]
fd[1]
child
fd[0]
fd[1]
pipe
kernel

- **Half-duplex pipe after a fork**



Pipe from parent to child



```c
#include   "ourhdr.h"
int main (void)
{
   int n, 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);
}
```

```
#include   <sys/wait.h>
#include   "ourhdr.h"
#define    DEF_PAGER    "/usr/bin/more"
           /* default pager program */
int main(int argc, char *argv[])
{
   int        n, fd[2];
   pid_t      pid;
   char       line[MAXLINE], *pager, *argv0;
   FILE       *fp;
   if (argc != 2)
        err_quit("usage: a.out <pathname>");
```

```
if ( (fp = fopen(argv[1], "r")) == NULL)
   err_sys("can't open %s", argv[1]);
   if (pipe(fd) < 0)
   err_sys("pipe error");

if ( (pid = fork()) < 0)
        err_sys("fork error");
       else if (pid > 0)
      {          /* parent */
        close(fd[0]);              /* close read end */
```

```
while (fgets(line, MAXLINE, fp) != NULL)
        {
                n = strlen(line);
                if (write(fd[1], line, n) != n)
                err_sys("write error to pipe");
        }
        if (ferror(fp))
        err_sys("fgets error");
        close(fd[1]);
        if (waitpid(pid, NULL, 0) < 0)
        err_sys("waitpid error");
        exit(0);
    }
```

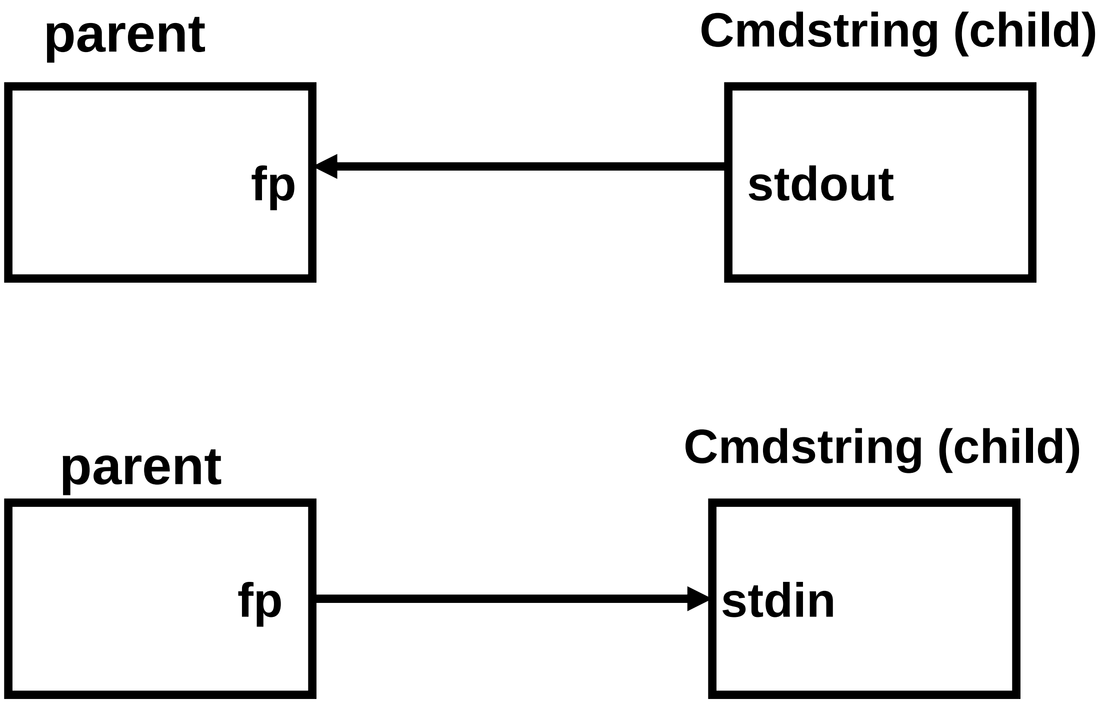
```
else {
        /* child */
        close(fd[1]);     /* close write end */
        if (fd[0] != STDIN_FILENO)
       {
   if (dup2(fd[0], STDIN_FILENO) !=
                                STDIN_FILENO)
        err_sys("dup2 error to stdin");
        close(fd[0]);
     /* don't need this after dup2 */
        }
              /* get arguments for execl() */
```

```
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)
            err_sys("execl error for %s",
   pager);
   }
}
```

popen and pclose functions

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

- **The function popen does a fork and exec to execute the cmdstring, and returns a standard I/O file pointer.**
- **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 us if the shell has executed exit. [127]**


parent
Cmdstring (child)
fp
stdout
parent
Cmdstring (child)
fp
stdin

```
#include    <sys/wait.h>
#include    "ourhdr.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('\n');
    exit(0);
}
```

```
/*mtuclc*/
#include   <ctype.h>
#include   "ourhdr.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);
}
```

Coprocesses

- Filters are normally connected linearly in shell pipelines.
- A filter becomes co-process when the same program generates its input and reads its output.
- The kornshell provides co-processes .
- The Bourne shell and c shell don't provide a way to connect processes together as coprocesses.

- A coprocess normally runs in the background from shell
- Its standard input and standard output are connected to another program using a pipe.

```
#include   "ourhdr.h"
int main(void)
{
   int          n, int1, int2;
   char         line[MAXLINE];

   while ( (n = read(STDIN_FILENO, line,
                              MAXLINE)) > 0)
        {
          line[n] = 0;
         /* null terminate */
```

```
if (sscanf(line, "%d%d", &int1, &int2) == 2)
    {
      sprintf(line, "%d\n", int1 + int2);
      n = strlen(line);

    if (write(STDOUT_FILENO, line, n) != n)
                err_sys("write error");
      }
```

```
else
        {
        if (write(STDOUT_FILENO, "invalid
                                        args\n", 13) != 13)
                          err_sys("write error");
        }
   }
   exit(0);
}
```

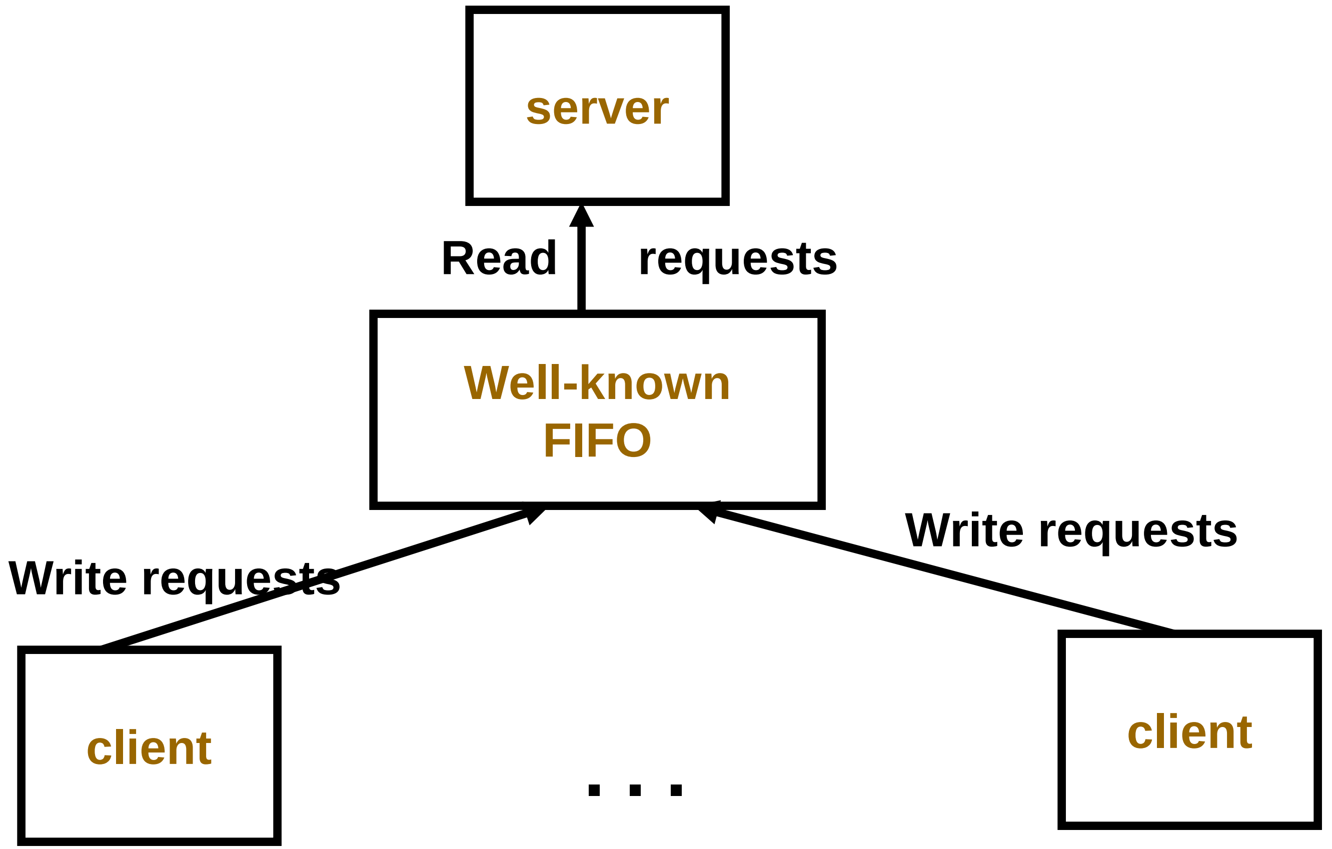
FIFOS

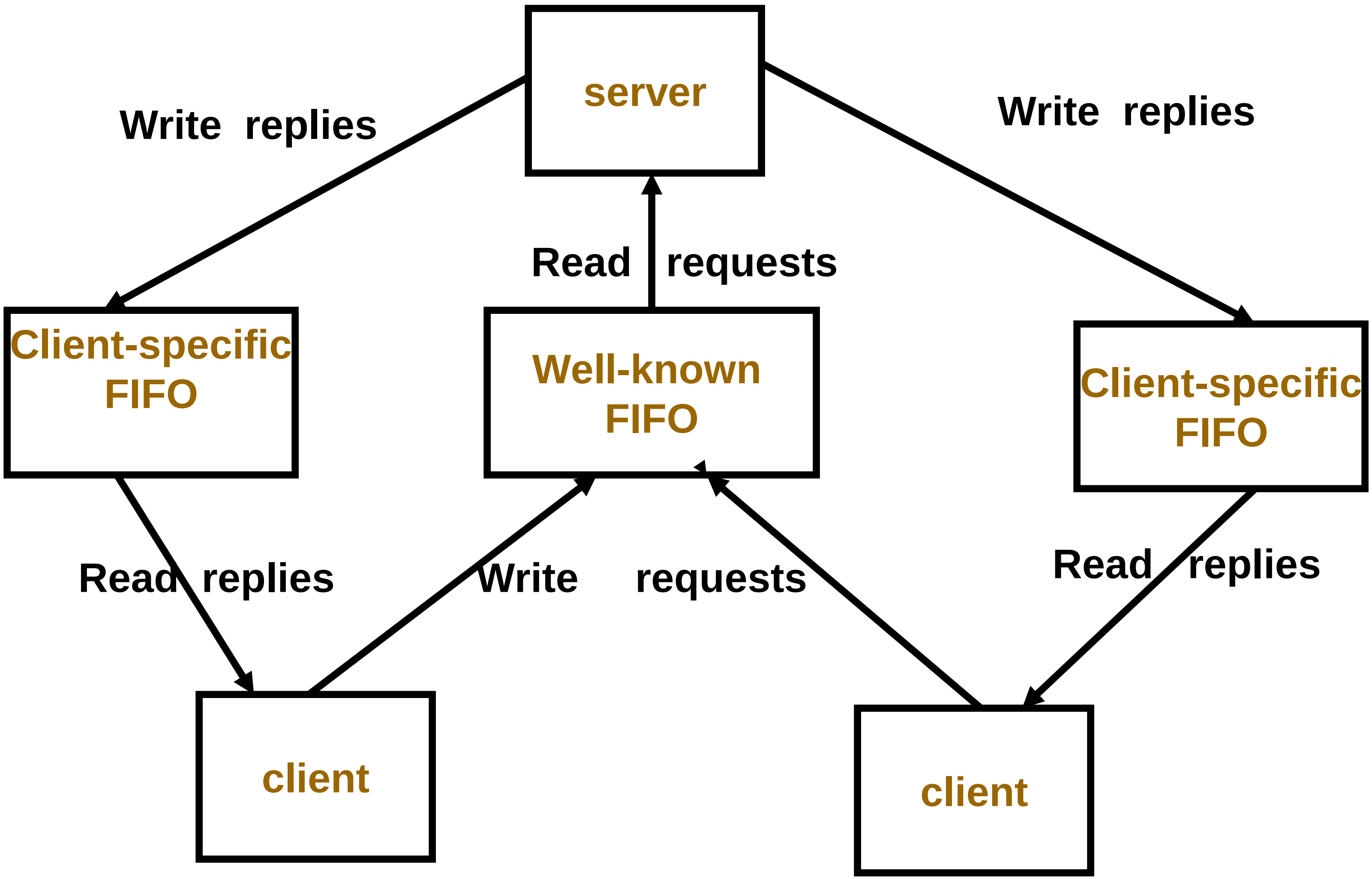
- Also called named pipes
- Can be used only between related processes when a common ancestor has created the pipe

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo (const char *pathname,
                                        mode_t mode);
```

- If O_NONBLOCK is not specified , an open for read-only blocks until some other process opens the FIFO for writing
- if O_NONBLOCK is specified , an open read-only returns an error with an errno of ENXIO if no process if no process has FIFO open for reading

Client server communication using a FIFO




server
Write replies
Write replies
Read requests
Client-specific FIFO
Well-known FIFO
Client-specific FIFO
Read replies
Write requests
Read replies
client
client

System V IPC

- **IPC structures**

1. **Message queues**
2. **Semaphores**
3. **Shared memory**

- **Identifiers and keys**
- **Each IPC structure in the kernel is referred to by a nonnegative integer identifier**
- **The identifier continually increases until it reaches maximum positive value for an integer and then wraps around to 0**
- **This value that is remembered even after an IPC structure is deleted and incremented every time it is used is called slot usage sequence number**

- The various ways for a client and server to rendezvous at the same IPC structure

1. The server can create a new IPC structure by specifying a key of IPC_PRIVATE and store the returned identifier for the client to obtain.
2. The client and server can agree upon a key by defining the key in a common header
3. The client and server can agree on a pathname and project ID and call the function ftok to convert these two values into a key

- **Permission structure**

```
Struct ipc_perm
 {
   uid_t uid;
   gid_t gid;
   uid_t cuid;
   gid_t cgid;
   mode_t mode;
   ulong seq;
   key_t  key;
};
```

permission	Message queue	semaphore	Shared memory
User-read User-write	MSG_R MSG_W	SEM_R SEM_A	SHM_R SHM_W
Group-read Group-write	MSG_R>>3 MSG_W>>3	SEM_R >> 3 SEM_A >> 3	SHM_R>>3 SHM_W>>3
Other-read Other-write	MSG_R>>6 MSG_W>>6	SEM_R >> 6 SEM_A >> 6	SHM_R>>6 SHM_W>>6

ADVANTAGES AND DISADVANTAGES

IPC type	Connectionless?	Reliable?	Flow Control
Message queues	No	Yes	Yes
Streams	No	Yes	Yes
Unix stream socket	No	Yes	Yes
Unix datagram socket	Yes	Yes	No
FIFOs	No	Yes	Yes

IPC type	Records?	Message types or primitives
Message queues	Yes	Yes
Streams	Yes	Yes
Unix stream socket	No	No
Unix datagram socket	Yes	No
FIFOs	No	No

Message queues

- Message queues are linked list of messages stored within the kernel and indentified by a message queue identifier
- The message queue is called “queue”
- Its identifier is called “queue ID”
- New messages are added o the end of a queue by msgnd

```
struct msqid_ds
  {
    struct ipc_perm    msg_perm;
    struct msg    *msg_first;
    struct msg    *msg_last;
    ulong    msg_cbytes;
    ulong    msg_qnum;
    ulong    msg_qbytes;
```

```
    pid_t    msg_lspid;
    pid_t    msg_lrpid;
    time_t   msg_stime;
    time_t   msg_rtime;
    time_t   msg_ctime;

};
```

- First function usually called is msgget

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

- When a queue is created

1. ipc_perm structure is initialized
2. msg_qnum, msg_lspid, msg_lrpid , msg_stime, and msg_rtime are set to 0

3. **msg_ctime is set to current time**
4. **msg_qbytes is set to system limit**

- **Msgget returns nonnegative queue ID**
- **Various other operations are performed by msgctl**

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

- **The cmd argument specifies the command to be performed , on the queue specified by msquid**
- **IPC_STAT – fetch the msqid_ds structure for this queue, storing it in structure pointed to by buf**
- **IPC_SET – set msg_perm.uid, msg_perm.gid, msg_perm.mode, msg_perm.qbytes from structure pointed to by buf in the structure associated with the queue**

- **IPC_RMID** – remove the message queue from the system and any data still on the queue
- Data is placed into queue by calling msgsnd

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgsnd (int msqid, const void *ptr,
     Size_t nbytes, int flags);
```

- **ptr argument is a pointer to a mymesg structure**

```
struct mymesg
 {
   long mtype;
   char mtext[512];
 }
```

- **Messages are retrieved from the queue using msgrcv**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
int msgrcv (int msqid, void *ptr
    size_t nbytes,long type,int flags);
```

- **Type = = 0 the first message in the queue is returned**
- **Type > 0 the first message in 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 absolute value of type is returned**

Semaphores

- A semaphore isn’t really a form of IPC .
- 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 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, indicating that it has used one unit of the resources.
3. If the value of the semaphore value 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.

- **When process is done with a shared resource that is controlled by a semaphore, the semaphore value is incremented by 1.**
- **If any process are asleep, waiting for the semaphore, they are awakened.**
- **To implement semaphores correctly, the test of a semaphore’s value and the decrementing of this value must be an atomic operation.**
- **For this reason, semaphores are normally implemented inside kernel.**

- **A common form of semaphore is called a binary semaphore.**
- **It controls a simple resource and its value is initialized to 1.**
- **In general , however, a semaphore can be initialized to any positive value.**
- **with the value indicating how many of units of the shared resource are available for sharing.**

- **System V semaphores**

- **A semaphore is not just a singal non negative value .it is set of one or more semaphore values**
- **The creation of semaphore is independent of its initialization**
- **The system neednot worry about the process of releasing semaphores before terminating**

```
struct semid_ds
 {
   struct ipc_perm    sem_perm;
   struct sem    *sem_base;
   ushort    sem_nsems;
   time_t    sem_otime;
   time_t    sem_ctime;
 };
```

```
struct sem
  {
    ushort   semval;
    pid_t    sempid;
    ushort   semncnt;
    ushort   semzcnt;
  };
```

- **The first function to call is semget to obtain semaphore ID**

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

- **When a new set is created**

1. ipc_perm structure is initialized

3. sem_otime is set to 0

4. sem_ctime is set to current time

5. sem_qbytes is set to nsems

- **The semctl function is the catchall for various semaphore operations**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl (int semid, int semnum,
        int cmd,union semnum arg);
```

```
union semun   {
int     val;
struct   semid_ds;
ushort    *array;    }
```

- **IPC_STAT – fetch the msqid_ds structure for this set, storing it in structure pointed to by arg.buf**
- **IPC_SET – set sem_perm.uid, sem_perm.gid, sem_perm.mode from structure pointed to by arg.buf in the structure associated with the set**
- **IPC_RMID – remove the semaphore set from the system**
- **GETVAL – return the value semval for the member semnum**

- **SETVAL – set the value semval for the member semnum**
- **GETPID – return the value sempid for the member semnum**
- **GETNCNT – return the value semncnt for the member semnum**
- **GETZCNT – return the value semzcnt for the member semnum**
- **GETALL – fetch all the semaphore values in the set**
- **SETALL – set all the semaphore values in the set to the values pointed to by arg.array**

Shared memory

- **Allows two or more processes to share a given region of memory**

```
struct msqid_ds
 {
   struct ipc_perm    shm_perm;
   struct anon_map    *shm_amp;
   int    shm_segsz;
   ushort    shm_lkcnt;
   pid_t    shm_lpid;
```

```
  pid_t    shm_lpid;
  ulong    shm_nattch;
  ulong    shm_cnattch;
  time_t   shm_atime;
  time_t   shm_dtime;
  time_t   shm_time;
};
```

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

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

- **When a new segment is created**

1. **ipc_perm structure is initialized**
2. **shm_lpid, shm_nattach, shm_atime, and shm_dtime are all set to 0**
3. **shm_ctime is set to current time**

- **The shmctl function is the catchall for various shared memory operations**

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl (int shmid, int cmd,
            struct shmid_ds *buf);
```

- **IPC_STAT – fetch the shmid_ds structure for this set, storing it in structure pointed to by a buf**

- **IPC_SET – set shm_perm.uid, shm_perm.gid, shm_perm.mode from structure pointed to by buf in the structure associated with the segment**
- **IPC_RMID – remove the shared memory segment from the system**
- **SHM_LOCK – lock the shared memory segment in memory**
- **SHM_UNLOCK – unlock the shared memory segment**

- **Once a shared memory segment has been created , a process attaches it to its address address space by calling shmat**

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

```
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/shm.h>
#include   "ourhdr.h"

#define    ARRAY_SIZE    40000
#define    MALLOC_SIZE 100000
#define    SHM_SIZE      100000
#define    SHM_MODE      (SHM_R | SHM_W)
        /* user read/write */

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

```
int main(void)
{
    int         shmid;
    char        *ptr, *shmptr;
printf("array[ ] from %x to %x\n", &array[0],
                                   &array[ARRAY_SIZE]);
    printf("stack around %x\n", &shmid);
if ( (ptr = malloc(MALLOC_SIZE)) ==
                                          NULL)
        err_sys("malloc error");
```

```
printf("malloced from %x to %x\n", ptr,
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 %x to
    %x\n",
                              shmptr, shmptr+SHM_SIZE);
    if (shmctl(shmid, IPC_RMID, 0) < 0)
        err_sys("shmctl error");

    exit(0);
}
```

Client-server properties

- When we are dealing with unrelated processes, a named stream pipe is required. We can take an unnamed stream pipe and attach a pathname in the file system to either end.
- A daemon server would create just one end of a stream pipe and attach a name to the end. This way unrelated clients can rendezvous with the daemon, sending message to the server's end of the pipe.

- **An even better approach is to use a technique whereby the server creates one end of a stream with a well known name, and clients connect to that end . Each time a new client connects to the server ‘s named stream pipe, a brand new stream. Pipe is created between client and server. This way the server is notified each time a new client connects to the server and when any client terminates.**

- **Three functions that can be used by a client server to establish these per-client connection.**

```
#include "ourhdr.h"
int serv_listen(const char * name);
```

- **First server has to announce its willingness to listen for client connections on a well-known name by calling serv_listen name is the well known name of the server.**

- **Client will use this name when they want to connect to the server. The return value is the file descriptor for the server's end of the named stream pipe.**
- **Once a server has called serv_listen , it calls serv_accept to wait for a client connection to arrive.**

```
#include "ourhdr.h"
int serv_accept (int listenfd, vid_t*vidplr);
```

- **Listenfd is a descriptor from serv_listen. This function doesn't return until a client connects the server's well known name .**

- When client does connect to the server's well known name. When client does connect to the server, a brand new stream pipe is automatically created.
- The new descriptor is returned as the value of the function. Additionally, the effective user ID of the client is stored through the pointer vidptr.
- A client just call cli_conn to connect to a server

Questions

- Give an overview of IPC methods **(10)**
- Write a short note on: **(5 each)**

1. Semaphores
2. Client-server interaction:
3. Coprocesses

- Exlpain p open and p close fuctions with prototypes and write a program to demonstrate the p open and p close fuctions **(10)**

- What are pipes? explain their limitations explain how pipes are created and used in ipc with example (10)
- Explain unix kernel support for messages and show related data structure. (10)
- what is message queue ? explain client server communication using a message queue (10)
- write a program to create a pipe from the parent to the child and send the data down the pipe (10)