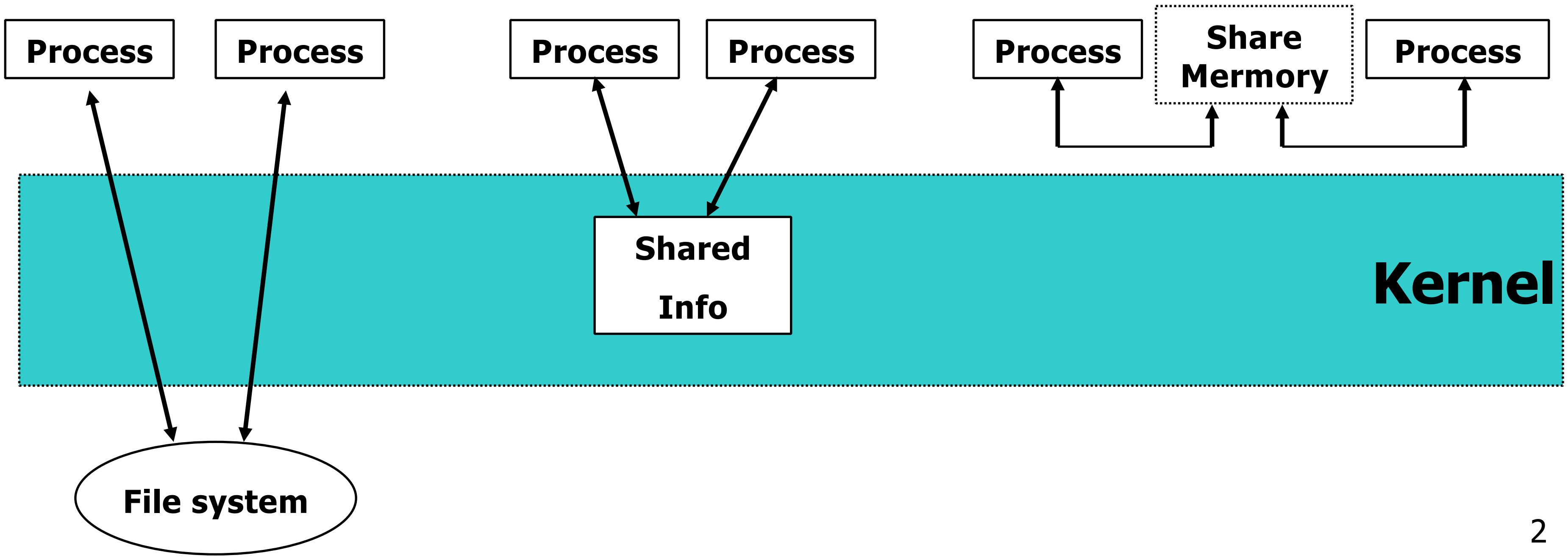
Chapter 15. Interprocess Communication

朱金辉

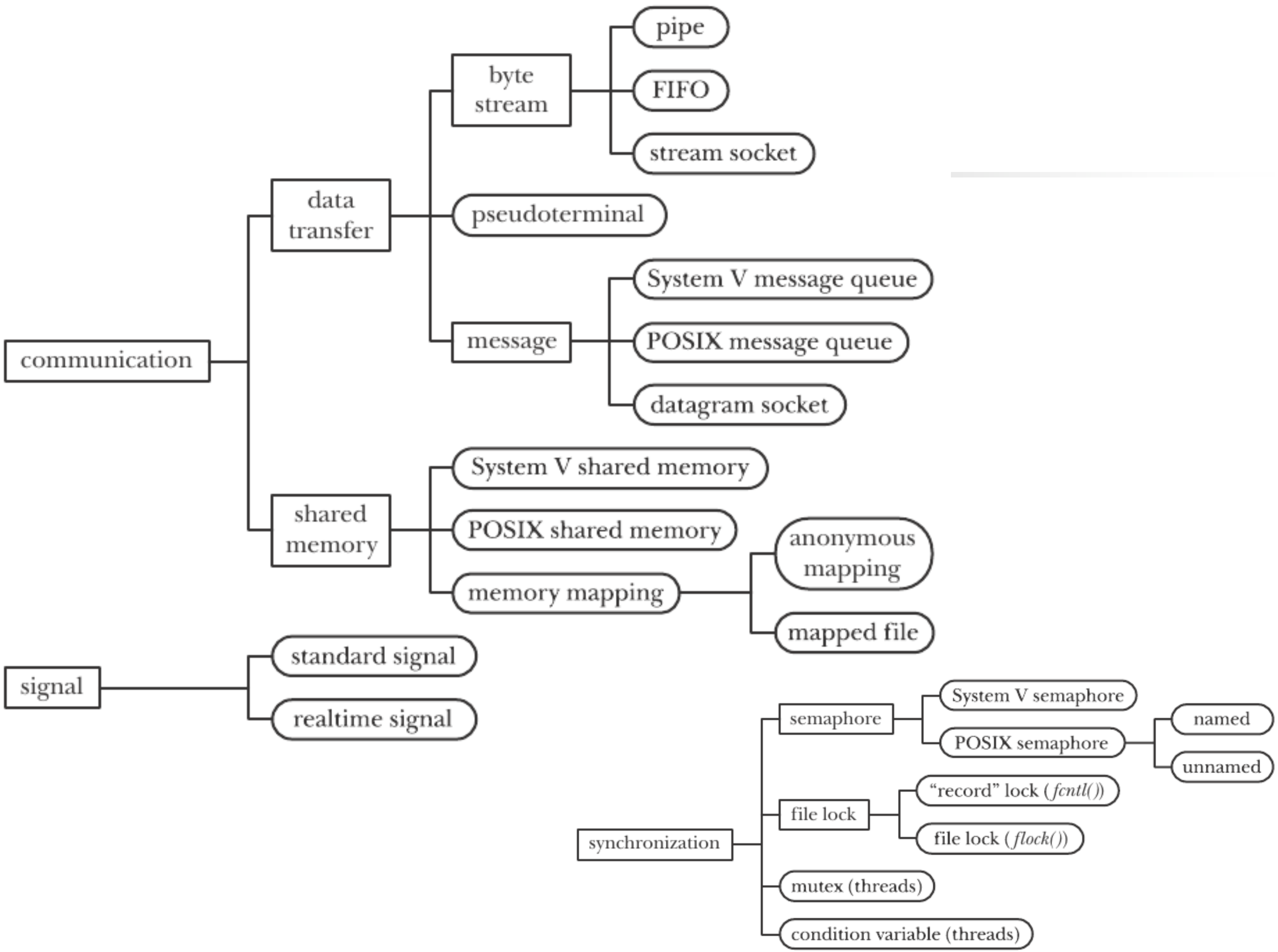
华南理工大学软件学院

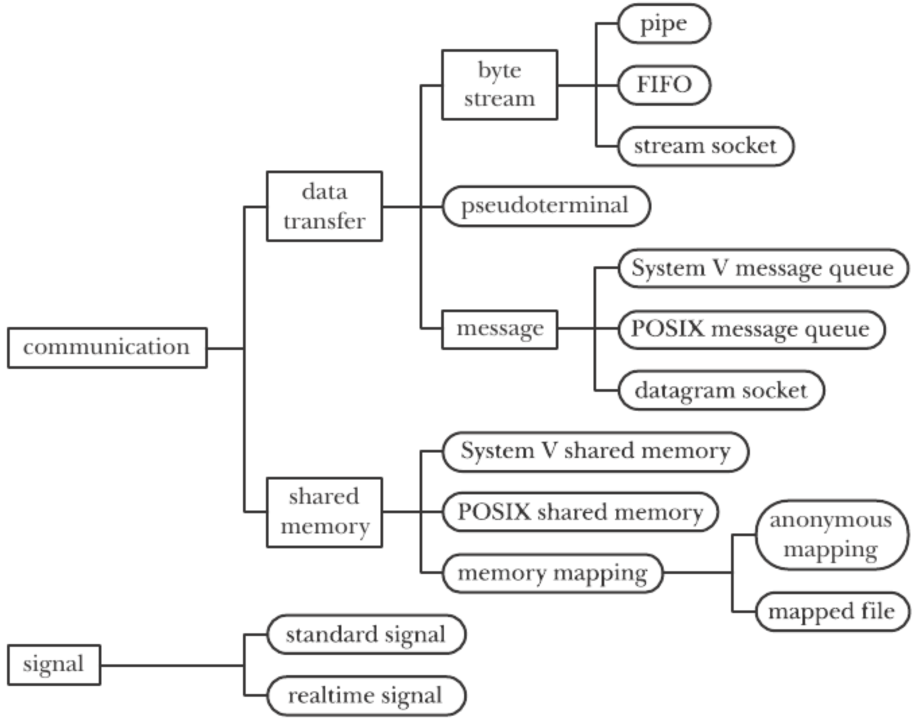
UNIX IPC

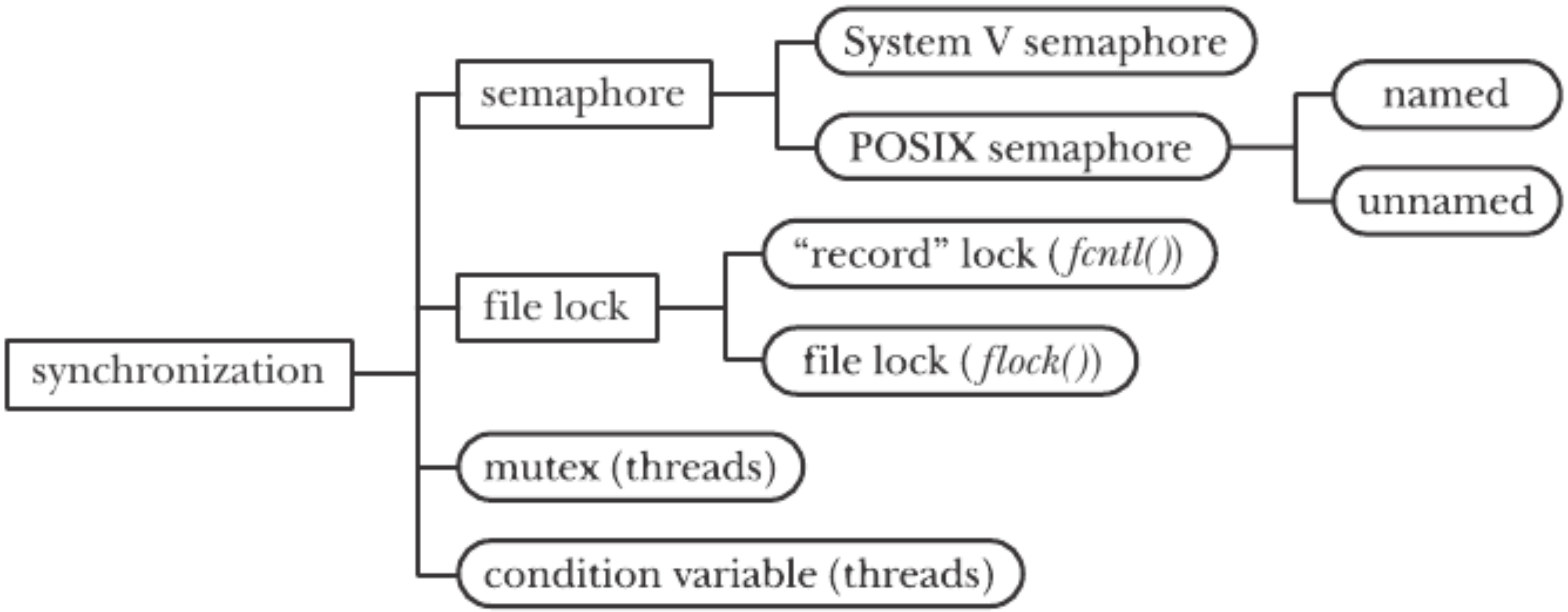
- IPC stands for interprocess communication
- Message passing between different process that are running on some operating system
- Need some forms of synchronization



A taxonomy of UNIX IPC facilities



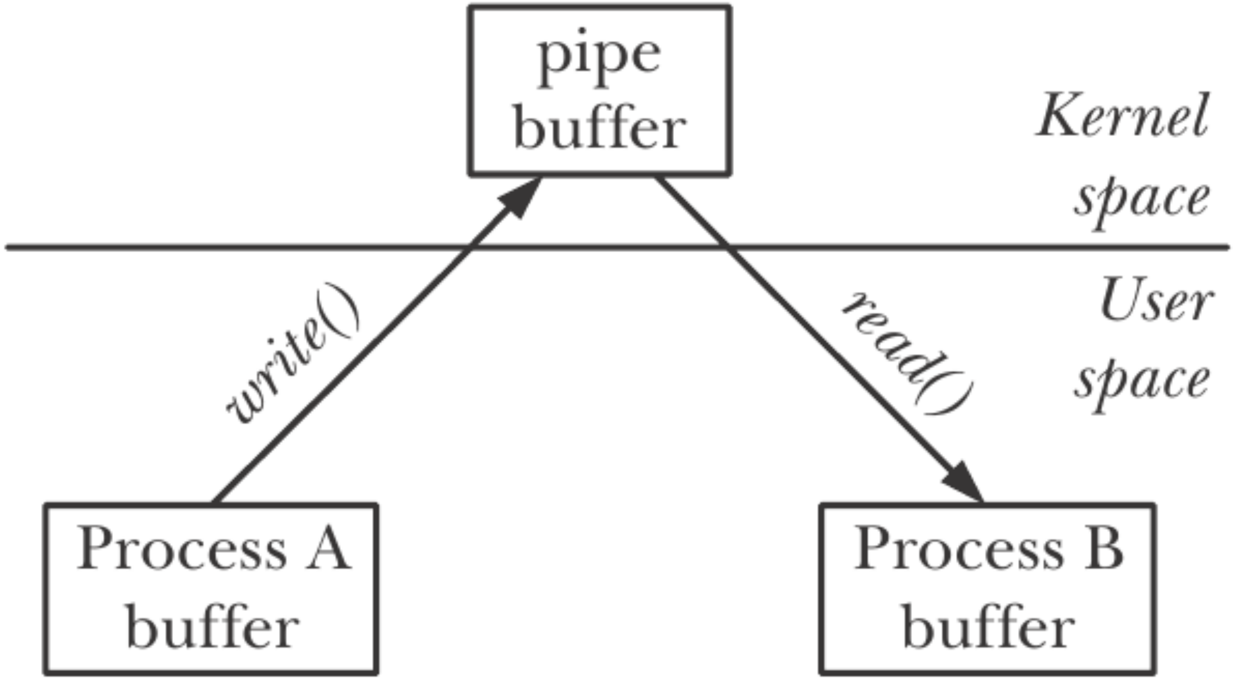

communication
data transfer
byte stream
pipe
FIFO
stream socket
pseudoterminal
message
System V message queue
POSIX message queue
datagram socket
shared memory
System V shared memory
POSIX shared memory
memory mapping
anonymous mapping
mapped file
signal
standard signal
realtime signal


synchronization
semaphore
System V semaphore
POSIX semaphore
named
unnamed
file lock
"record" lock (fcntl())
file lock (flock())
mutex (threads)
condition variable (threads)

Contents

- Pipes
- FIFO: name pipes
- Message Queues
- Semaphores
- Shared Memory
- Memory mapping

1. Pipes



Exchanging data between two processes using a pipe

Pipes

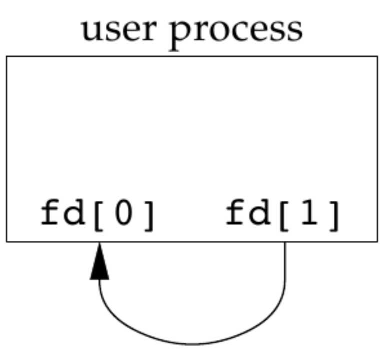
- Oldest form
- half-duplex (only 1-direction data flow)
- no names
- process creates a pipe, forks, and uses the pipe to communicate with child

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

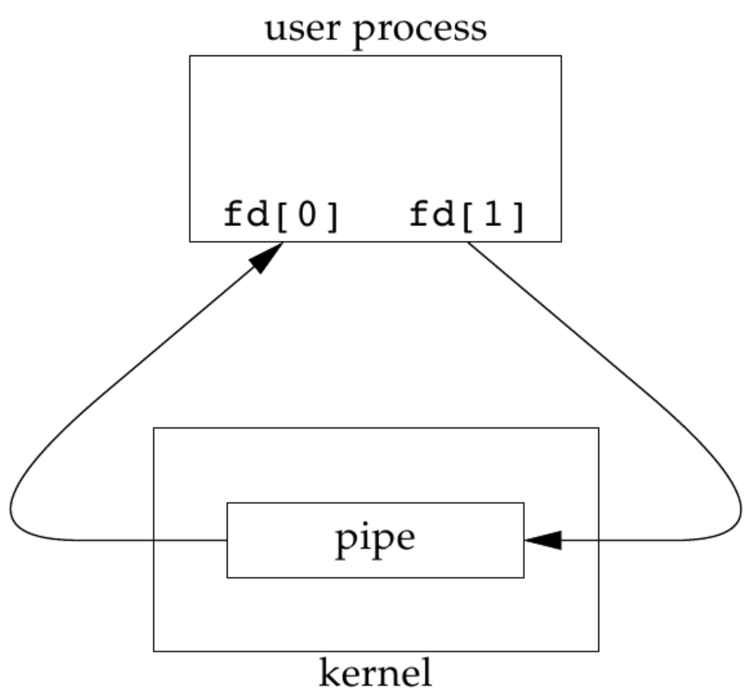
- Returns: 0 if OK, -1 on error

Pipes

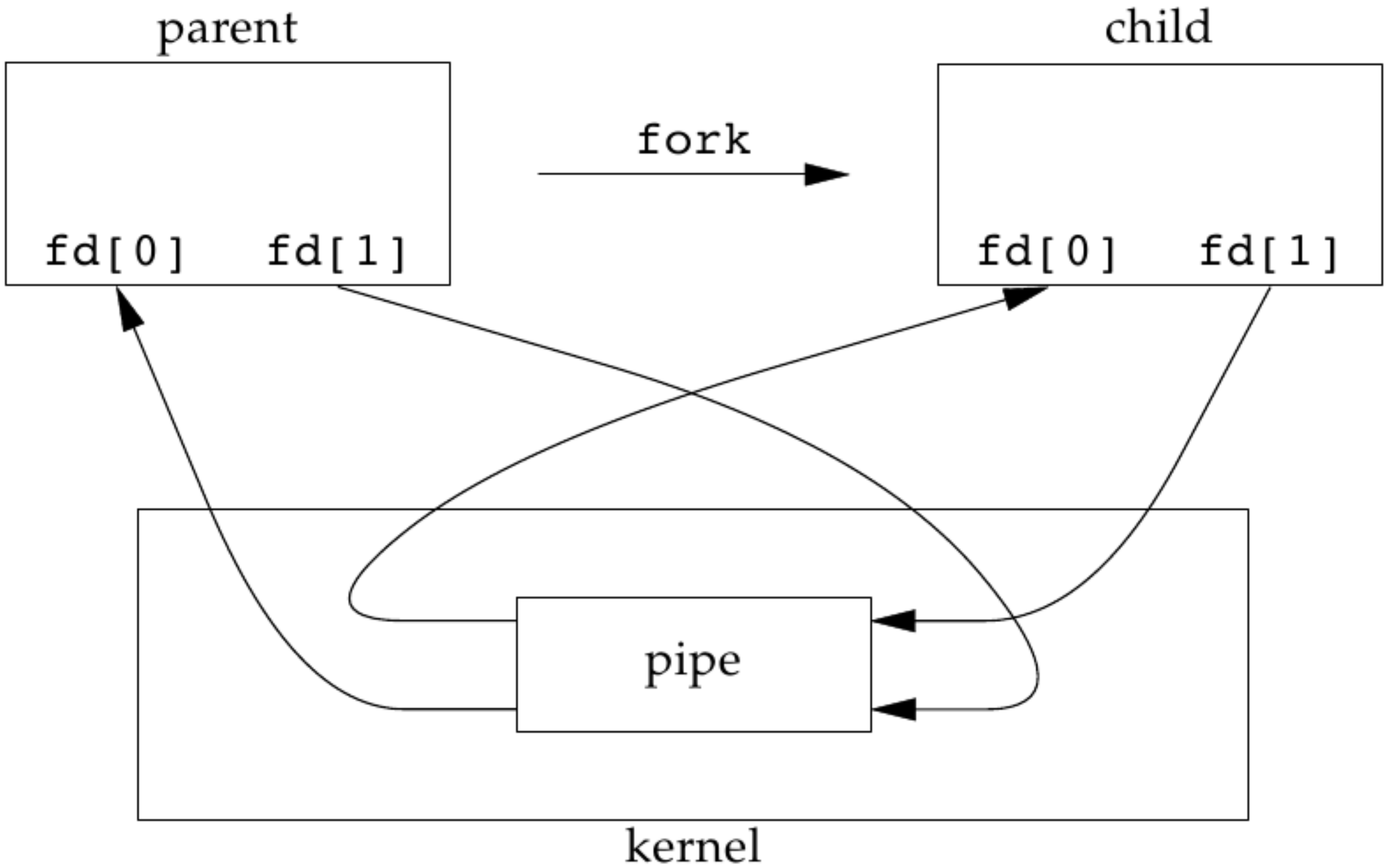
- filedes[0]: open for reading
- filedes[1]: open for writing
- Output of filedes[1] is input of filedes[0]

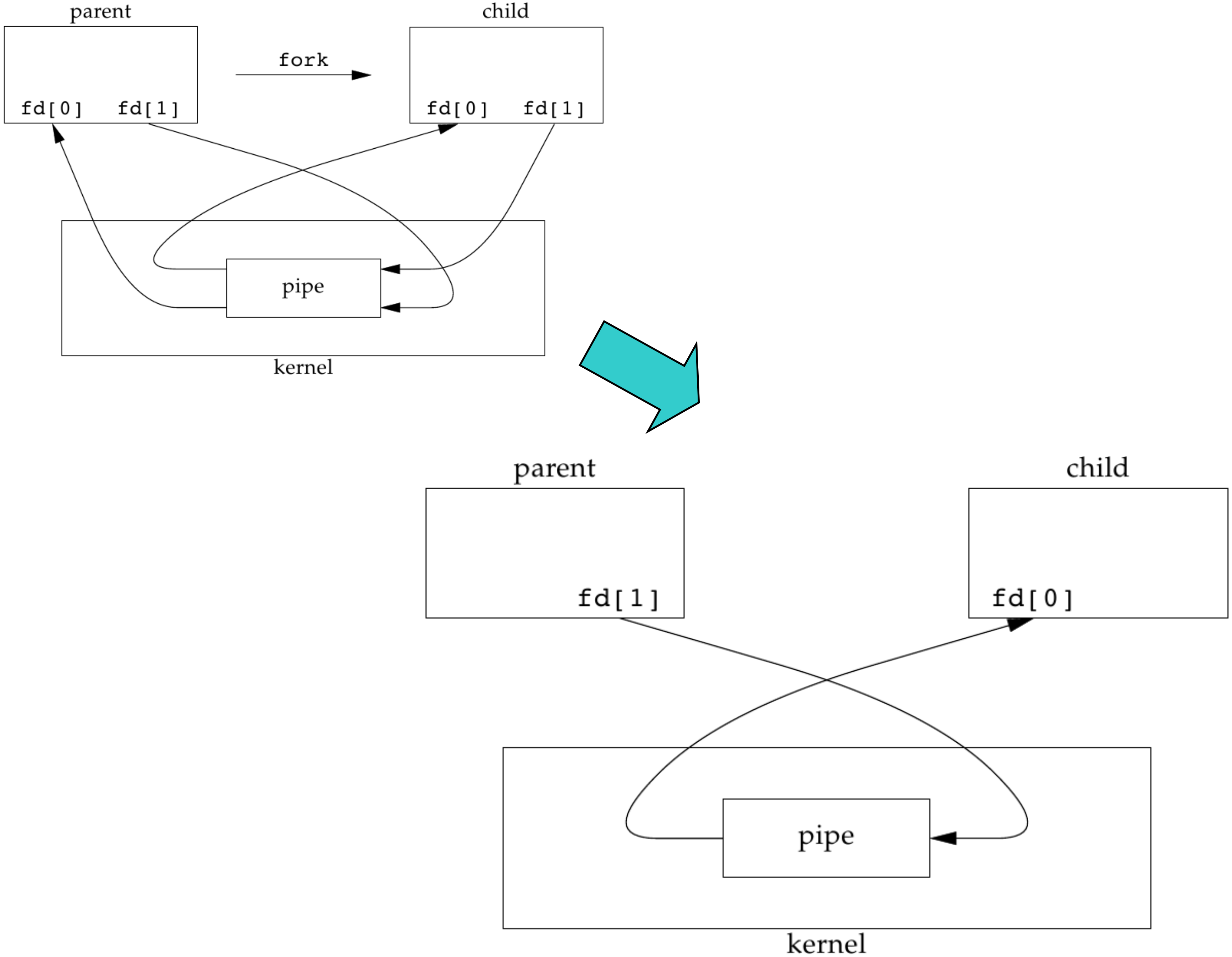


or



Half-duplex pipe after a fork




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

Program 15.5: Parent→Child Pipe

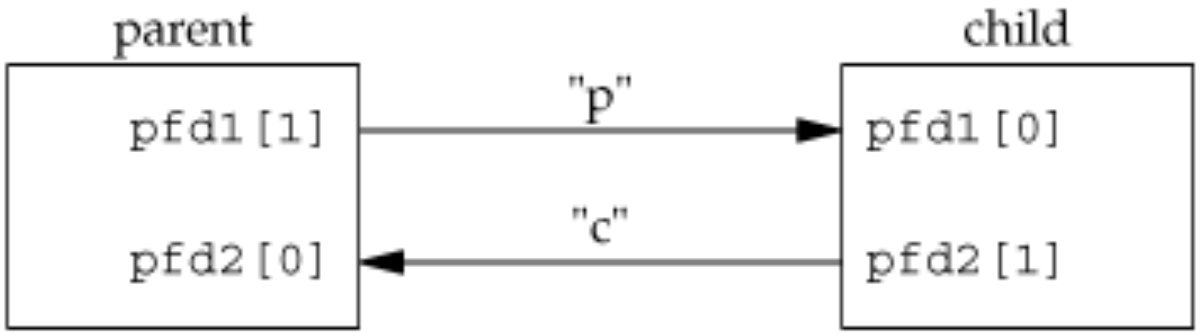
```
int main(void) {
   int         n, fd[2];
   pid_t       pid;
   charline[MAXLINE];
   if (pipe(fd) < 0)
       err_sys("pipe error");
   if ( (pid = fork()) < 0) err_sys("fork error");
   else if (pid > 0) {              /* parent */
       close(fd[0]);
       write(fd[1], "hello world\n", 12);
   } else {                         /* child */
       close(fd[1]);
       n = read(fd[0], line, MAXLINE);
       write(STDOUT_FILENO, line, n);
   }
   exit(0);
}
```

Pipes

- Read from a pipe if write end closed:
  - if data, data is read, and
  - when no data, returns 0 (→ EOF)
- Write to a pipe if read end closed:
  - SIGPIPE generated,
  - write() returns errno = EPIPE
- PIPE_BUF: #bytes in kernel's pipe buffer size

Using 2 pipes for parent/child synchronization

Figure 15.8. Using two pipes for parentchild synchronization



Program 15.7: Pipe sync

```
#include    "apue.h"

static int  pfd1[2], pfd2[2];

void
TELL_WAIT() {
  if (pipe(pfd1) < 0 || pipe(pfd2) < 0)
      err_sys("pipe error");
}
Void TELL_PARENT(pid_t pid) {
  if (write(pfd2[1], "c", 1) != 1)
      err_sys("write error");
}
```

Program 15.7: Pipe sync

```
void WAIT_PARENT(void) {
   char      c;
    if (read(pfd1[0], &c, 1) != 1) err_sys("read error");
    if (c != 'p') err_quit("WAIT_PARENT: incorrect data");
}
void TELL_CHILD(pid_t pid) {
  if (write(pfd1[1], "p", 1) != 1) err_sys("write error");
}
void WAIT_CHILD(void) {
   char      c;
   if (read(pfd2[0], &c, 1) != 1) err_sys("read error");
   if (c != 'c')
       err_quit("WAIT_CHILD: incorrect data");
}
```

popen & pclose Functions

- All dirty work:
  - create a pipe
  - fork a child
  - close unused ends of pipe
  - exec a shell to execute a command
  - wait for command to terminate
- #include <stdio.h>

  FILE *popen(const char *cmdstring, const char *type);
- Returns: file pointer if OK, NULL on error

  int pclose(FILE *fp);
- Returns: termination status of cmdstring, or -1 on error

popen: type = r or w

Figure 15.9 Result of `fp = popen(`*cmdstring*`, "r")`

Figure 15.10 Result of `fp = popen(`*cmdstring*`, "w")`

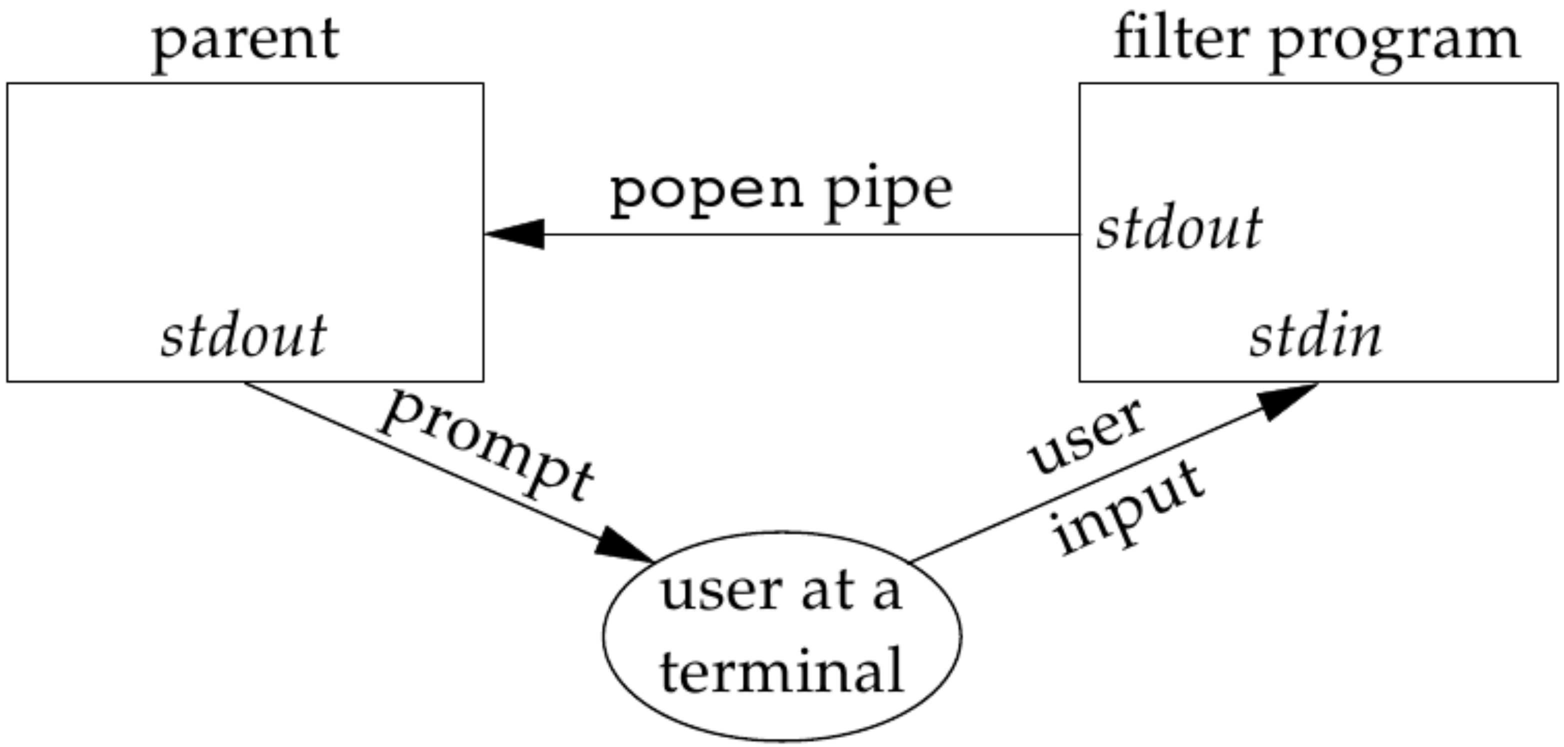
Program 15.11: pager (popen)

```
#include    <sys/wait.h>
#include    "apue.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]);
```

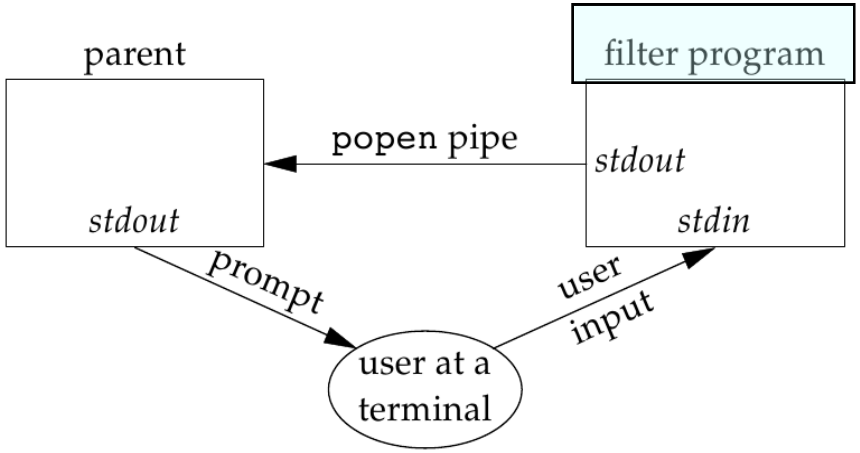
Program 15.11: pager (popen)

```
if ( (fpout = popen(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);
}
```

Filter Program (with popen)

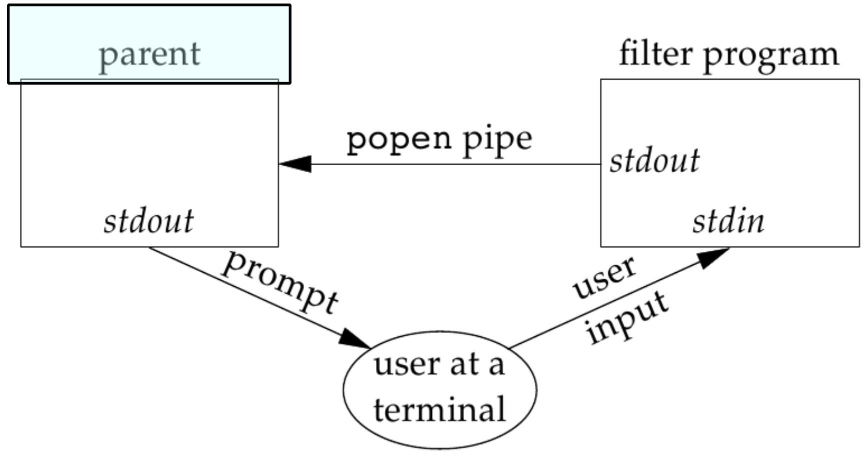


Program 15.14 (filter)




```c
#include      <ctype.h>
#include      "apue.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);
```

Program 15.15 (using filter)




```
#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('\n');
	exit(0);
}
```

2. FIFOs

- Also called "named pipes"
- Can be used by unrelated processes
- File type: stat.st_mode = FIFO
- Test with S_ISFIFO macro
- Similar to creating a file : mkfifo
- Pathname exists
- open, close, read, write, unlink, etc

FIFOs

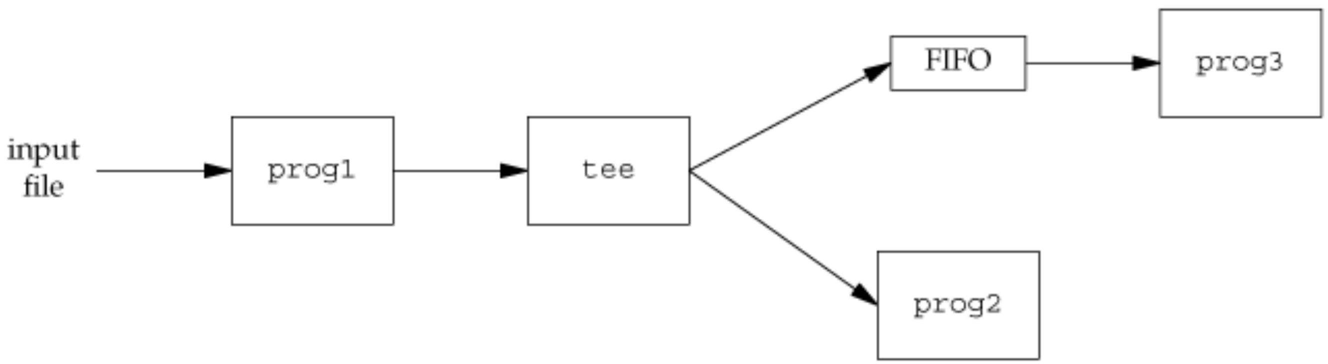
```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo (const char *pathname, mode_t mode);
```

- Returns: 0 if OK, -1 on error
- mode: same as for open function

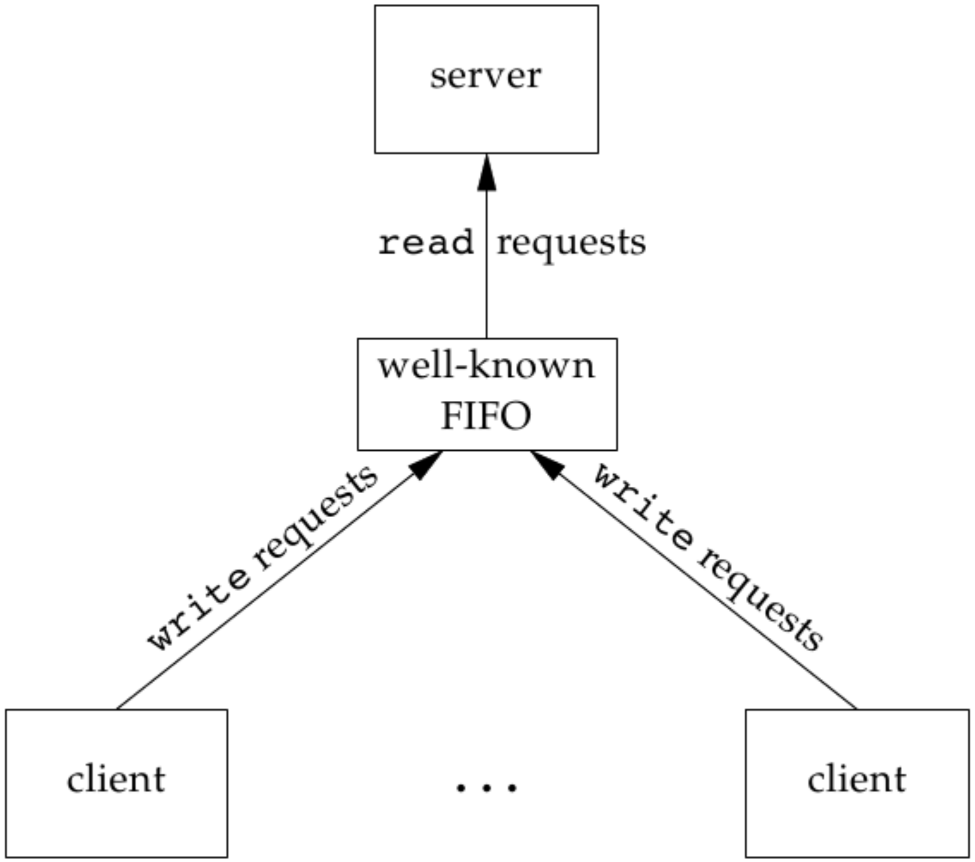
Using FIFOs to Duplicate Output Streams

- mkfifo fifo1
- prog3 < fifo1 &
- prog1 < infile | tee fifo1 | prog2

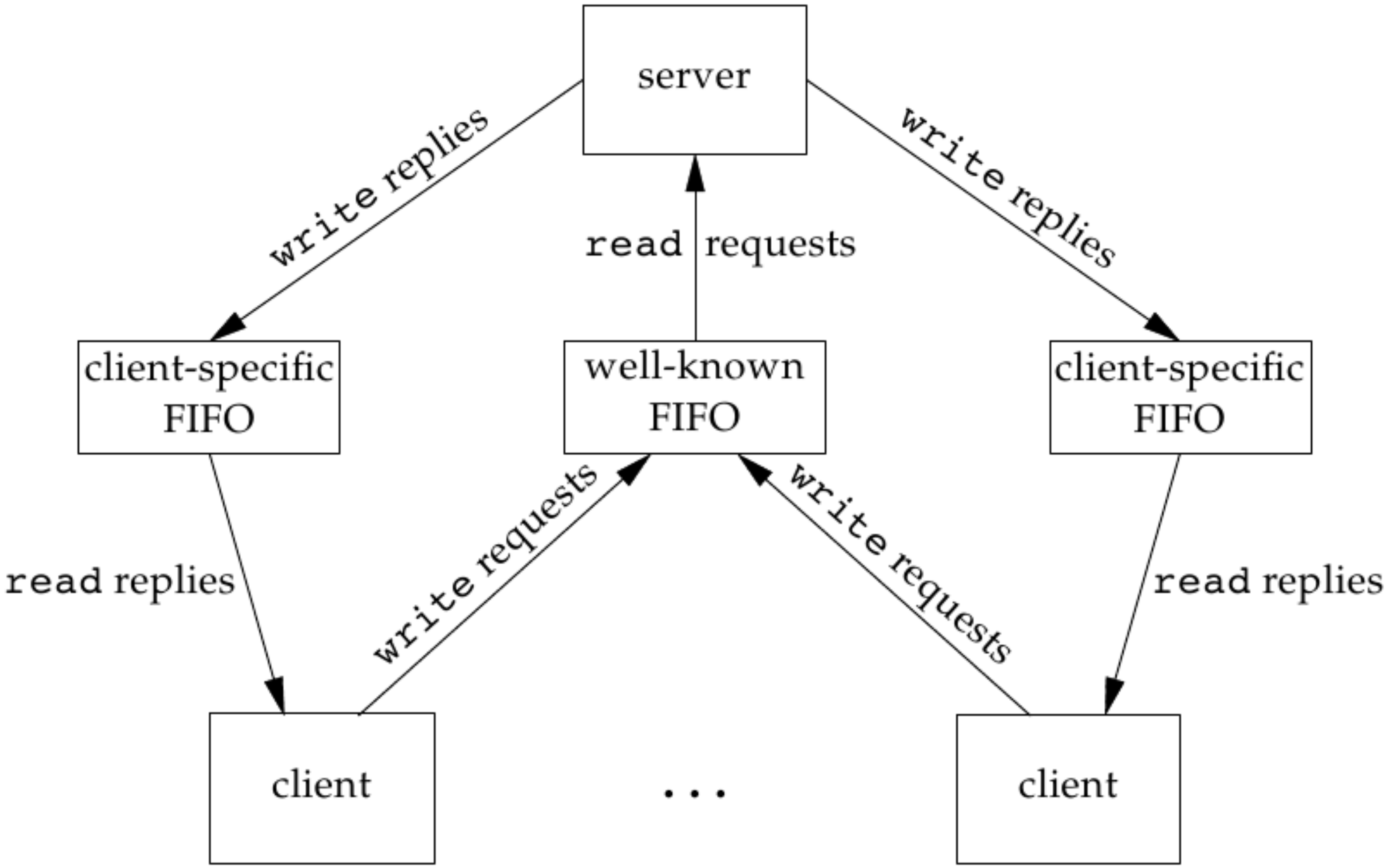
Copies stdin to stdout and file



Client-Server Communication Using a FIFO



Client-Server Communication Using a FIFO



Client-Server Communication Using a FIFO

- Solution: Create a FIFO for each client such that server can reply using the client-specific FIFO
  - **E.g.** /tmp/serv1.XXXX, where XXXXX is client's process ID
  - Impossible for server to know if a client has crashed, FIFOs left in system,
  - server must catch SIGPIPE (FIFO with 1 writer, no reader)

3. XSI IPC

XSI : X/Open System Interface extension

The XSI IPC functions are based closely on the System V IPC functions.

- ✓ Message Queues
- ✓ Semaphores
- ✓ Shared Memory

- When creating an IPC structure, a key must be specified (type: key_t)
- Each IPC structure has a nonnegative integer identifier (large!)
- Command: ipcs, ipcrm

Client-Server Rendezvous at same IPC structure (1)

- Server creates a new IPC structure using key = IPC_PRIVATE
- Server stores returned identifier in some file for client to obtain
- Disadvantage: file I/O!

Client-Server Rendezvous at same IPC structure (2)

- Define a key in a common header
- Client and server agree to use that key
- Server creates a new IPC structure using that key
- Problem: key exists? (msgget, semget, shmget returns error)
- Solution: delete existing key, create a new one again!

Client-Server Rendezvous at same IPC structure (3)

- Client and server agree on
  - a pathname
  - a project ID (char between 0 ~ 255)
- ftok() converts the 2 values into a key
- Client and server use that key
- Disadvantage: ftok → a function call!

Permission Structure for IPC

XSI IPC associates an ipc_perm structure with each IPC structure.

```
struct ipc_perm {
    uid_t uid;   /* owner's EUID */
    gid_t gid;   /* owner's EGID */
    uid_t cuid;        /* creator's EUID */
    gid_t cgid;        /* creator's EGID */
    mode_t mode; /* access mode */
    ulong seq;         /* slot usage seq number */
    key_t key;         /* key */
};
```

Summary of XSI IPC functions

	Message queues	Semaphores	Shared Memory
Header	<sys/msg.h>	<sys/sem.h>	<sys/shm.h>
create or open	msgget	semget	shmget
Control operations	msgctl	semctl	shmctl
IPC operations	msgsnd msgrcv	semop	shmat shmdt

(1) Message Queues

- Linked list of messages
- Stored in kernel
- Identified by message queue identifier
- msgget: create new or open existing q
- msgsnd: add new msg to a queue
- msgrcv: receive msg from a queue
- Fetching order: based on type of msg

Message Queues

- creates a new message queue or obtains the identifier of an existing queue
- #include <sys/types.h>

  #include <sys/ipc.h>

  #include <sys/msg.h>

  int msgget(key_t key, int flag);
- Returns: msg queue ID if OK, -1 on error

```
id = msgget(key, IPC_CREAT | S_IRUSR | S_IWUSR);
if (id == -1)
    errExit("msgget");
```

Operations on queue

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

IPC_STAT: fetch into buf
IPC_SET: set from buf
IPC_RMID: remove from queue

- Returns: 0 if OK, -1 on error

Message Queues data structure

```
struct msqid_ds {
  struct ipc_perm  msg_perm;     /* see Section 15.6.2 */
  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 Structure

```
struct mymesg {
    long mtype;          /* positive msg type */
    char mtext[512];     /* message data */
}
```

Place data on message queue

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

Pointer to:
type + data (nbytes data)

- Returns: 0 if OK, -1 on error

IPC_NOWAIT or 0

Retrieve Message from Queue

type == 0: 1st msg

type>0: 1st msg of type

type < 0: first msg with type <= |type| and smallest

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

- Returns: data size in message if OK, -1 on error

Example(1/4)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#define MSGSZ 128
struct msgbuf {
   long mtype;
    char mtext[MSGSZ];
};

int main () {
        int msgid;
        key_t key;
        struct msgbuf sbuf,rbuf;
        key=4587;
```

Example(2/4)

```
if ((msgid=msgget(key,IPC_CREAT|0666))<0) {
        printf("msgget");
        exit(1);
}
if (msgrcv(msgid,&rbuf,MSGSZ,0,0)<0) {
        printf("msgrcv");
        exit(1);
}
sbuf.mtype=2;
sprintf(sbuf.mtext,"I received the message.");
if (msgsnd(msgid,&sbuf,strlen(sbuf.mtext)+1,0)<0) {
        printf("msgsnd");
        exit(1);
}
exit(0);
}
```

Example(3/4)

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdio.h>
#define MSGSZ 128
struct msgbuf {
   long mtype;
   char mtext[MSGSZ];
};
int main () {
        int msgid;
        key_t key;
        struct msgbuf sbuf,rbuf;
        key=4587;
```

Example(4/4)

```
    if ((msgid=msgget(key,0666))<0) {
            printf("msgget");
            exit(1);
     }
     sbuf.mtype=1;
     sprintf(sbuf.mtext,"Did you get the Message ?");
     if (msgsnd(msgid,&sbuf,strlen(sbuf.mtext)+1,0) {
            printf("msgsnd");
            exit(1);
     }
     if (msgrcv(msgid,&rbuf,MSGSZ,2,0)<0) {
            printf("msgrcv");
            exit(1);
     }
     printf("%s\n",rbuf.mtext);
     exit(0);
}
```

(2) Semaphores

- Semaphores are intended to let multiple processes synchronize their operations.
- A counter to provide access to shared data object for multiple processes
- To object a shared resource:
  - Test semaphore controlling resource
  - If value > 0, value--, grant use
  - If value == 0, sleep until value > 0
  - Release resource, value ++

Need to be ATOMIC

Complication

- Three features contribute to this unnecessary complication.

1. A semaphore is defined as a set of one or more semaphore values
2. Creation (semget) is independent of initialization (semctl). Cannot atomically create a new semaphore set and initialize all the values in the set.
3. All IPCs exist even if no process is using them. Need worry about process terminating without releasing semaphore.

Semaphore structure

```
struct semid_ds {
  struct ipc_perm  sem_perm;  /* see Section 15.6.2 */
  unsigned short   sem_nsems; /* # of semaphores in set */
  time_t           sem_otime; /* last-semop() time */
  time_t           sem_ctime; /* last-change time */
  :
};
```

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

```
struct {
  unsigned short  semval;   /* semaphore value, always >= 0 */
  pid_t           sempid;   /* pid for last operation */
  unsigned short  semncnt;  /* # processes awaiting semval>curval */
  unsigned short  semzcnt;  /* # processes awaiting semval==0 */
  :
};
```

Obtain a semaphore

- #include <sys/types.h>
- #include <sys/ipc.h>
- #include <sys/sem.h>
- int semget (key_t key, int nsems, int flag);
- Returns: sem ID if OK, -1 on error

Semaphore control

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/sem.h>

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

- cmd: stat, set, rmid, getval, setval, getall, setall …

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

Semaphore operation(1)

- #include <sys/types.h>
- #include <sys/ipc.h>
- #include <sys/sem.h>
- int semop(int semid,struct sembuf semoparray[],size_t nops);
- Return: 0 if ok, -1 on error. This is an atomic operation

```
struct sembuf {
  unsigned short  sem_num;  /* member # in set (0, 1, ..., nsems-1) */
  short           sem_op;   /* operation (negative, 0, or positive) */
  short           sem_flg;  /* IPC_NOWAIT, SEM_UNDO */
};
```

Struct of sembuf

```
struct sembuf {
    ushort sem_num;
 /*member # in set (0,1,…,nsems-1)*/
    short sem_op;
/*operation (negative, 0, or positive) */
    short sem_flg;
 /*IPC_NOWAIT,SEM_UNDO */
}
```

Semaphore operation(2)

- sem_op is positive: release the resource hold by process, value of sem_op added to semaphore’s value
- sem_op is 0: the calling process wants to wait until the semaphore’s value becomes 0.
- sem_op is negative: process want to obtain resources

Example(1/5)

```
int main {
    int sem_id,i,creat=0,pause_time;
    char *cp;
    srand((unsigned int )getpid());
    sem_id=open_semaphore_set((key_t)1234,1);
    if (argc>1&&strcmp(argv[1],"1")) {
        init_a_semaphore(sem_id,0,1);
        creat=1;
        sleep(2);
    }
```

Example(2/5)

```
for(i=0;i<argc;i++) {
        cp=argv[i];
        if (!semaphore_P(sem_id)) exit(-1);
        printf("process %d:",getpid());
        fflush(stdout);
        while(*cp) {
                printf("%c",*cp);  fflush(stdout);
                sleep(rand()%3);
                cp++;
        }
        printf("\n");
        if (semaphore_V(sem_id)) exit(-1);
        sleep(rand()%2);
}
```

Example(3/5)

```
    printf("\n%d –finished\n",getpid());
    if (creat==1) {
        sleep(10);
        rm_semaphore(sem_id);
    }
    exit(0);
}
int open_semaphore_set(key_t keyval,int numsems) {
    int sid;
    if (!numsems)  return(-1);
    if ((sid=semget(keyval,numsems,IPC_CREAT|0660))==-1) return (-1);
    else return(sid);
}
```

Example(4/5)

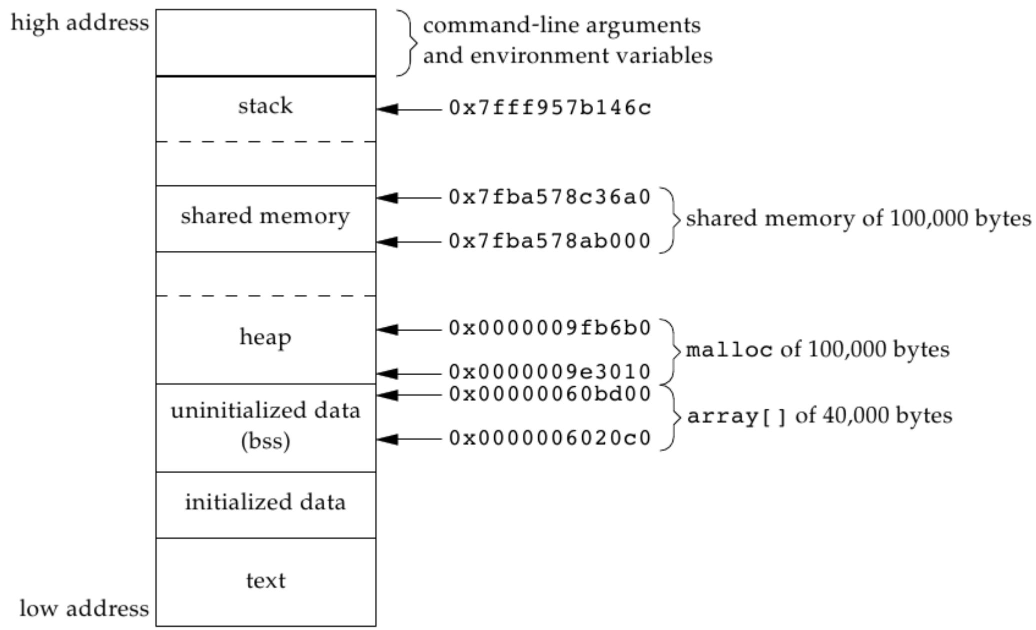
```
int semaphore_P(int sem_id) {
    struct sembuf sb; sb.sem_num=0;
    sb.sem_op=-1; sb.sem_flag=SEM_UNDO;
    if (semop(sem_id,&sb,1)==-1 {
        fprintf(stderr,"semaphore_P failed\n"); return(0);
    }
    return(1);
}
int semaphore_V(int sem_id) {
    struct sembuf sb; sb.sem_num=0;
    sb.sem_op=1; sb.sem_flag=SEM_UNDO;
    if (semop(sem_id,&sb,1)==-1 {
        fprintf(stderr,"semaphore_V failed\n"); return(0);
    }
    return(1);
}
```

Example(5/5)

```
union semun {
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};
void init_a_semaphore(int sid,int semnum,int initval) {
    union semun semopts;
    semopts.val=initval;
    semctl(sid,semnum,SETVAL,semopts);
}
int rm_semaphore(int sid) {
    return(semctl(sid,0,IPC_RMID,0));
}
```

(3) Shared Memory

- Fastest form of IPC: no need of data copying between client & server
- Must synchronize access to a shared memory segment
- Semaphores are used



Obtain a shared memory id

- #include <sys/types.h>
- #include <sys/ipc.h>
- #include <sys/shm.h>
- int shmget (key_t key, int size, int flag);
- Returns: shared memory ID if OK, -1 on error

Shared Memory Operations

- #include <sys/types.h>
- #include <sys/ipc.h>
- #include <sys/shm.h>
- int shmctl (int shmid, int cmd, struct shmid_ds *buf);
- Returns: 0 if OK, -1 on error

Shared Memory Segment Structure

```
struct shmid_ds {
  struct ipc_perm  shm_perm;    /* see Section 15.6.2 */
  size_t           shm_segsz;   /* size of segment in bytes */
  pid_t            shm_lpid;    /* pid of last shmop() */
  pid_t            shm_cpid;    /* pid of creator */
  shmatt_t         shm_nattch;  /* number of current attaches */
  time_t           shm_atime;   /* last-attach time */
  time_t           shm_dtime;   /* last-detach time */
  time_t           shm_ctime;   /* last-change time */
  :
};
```

Attaching to a process

- #include <sys/types.h>
- #include <sys/ipc.h>
- #include <sys/shm.h>
- void *shmat (int shmid, void *addr, int flag);
- Returns: pointer to shared memory segment if OK, -1 on error

- addr is NULL, the system chooses a suitable (unused) address at which to attach the segment.
- addr isn't NULL and SHM_RND is specified in flag, the attach occurs at the address equal to addr rounded down to the nearest multiple of SHMLBA.

Detaching from a process

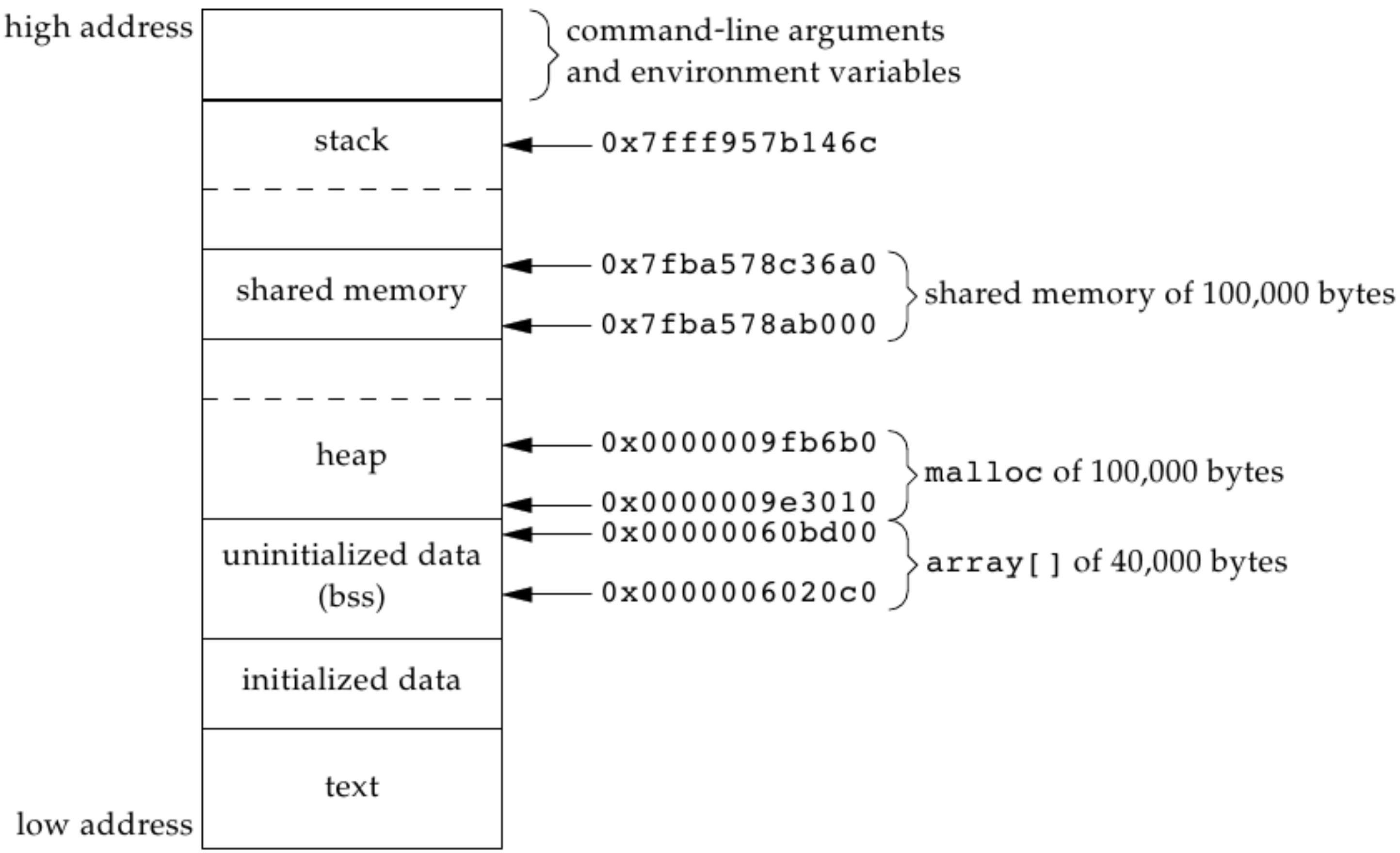
- #include <sys/types.h>
- #include <sys/ipc.h>
- #include <sys/shm.h>
- int shmdt (void *addr);
- Returns: 0 if OK, -1 on error

Program 15.31: data storage

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include "apue.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);
}
```

Memory Layout (Program 15.31)



4. memory mapped

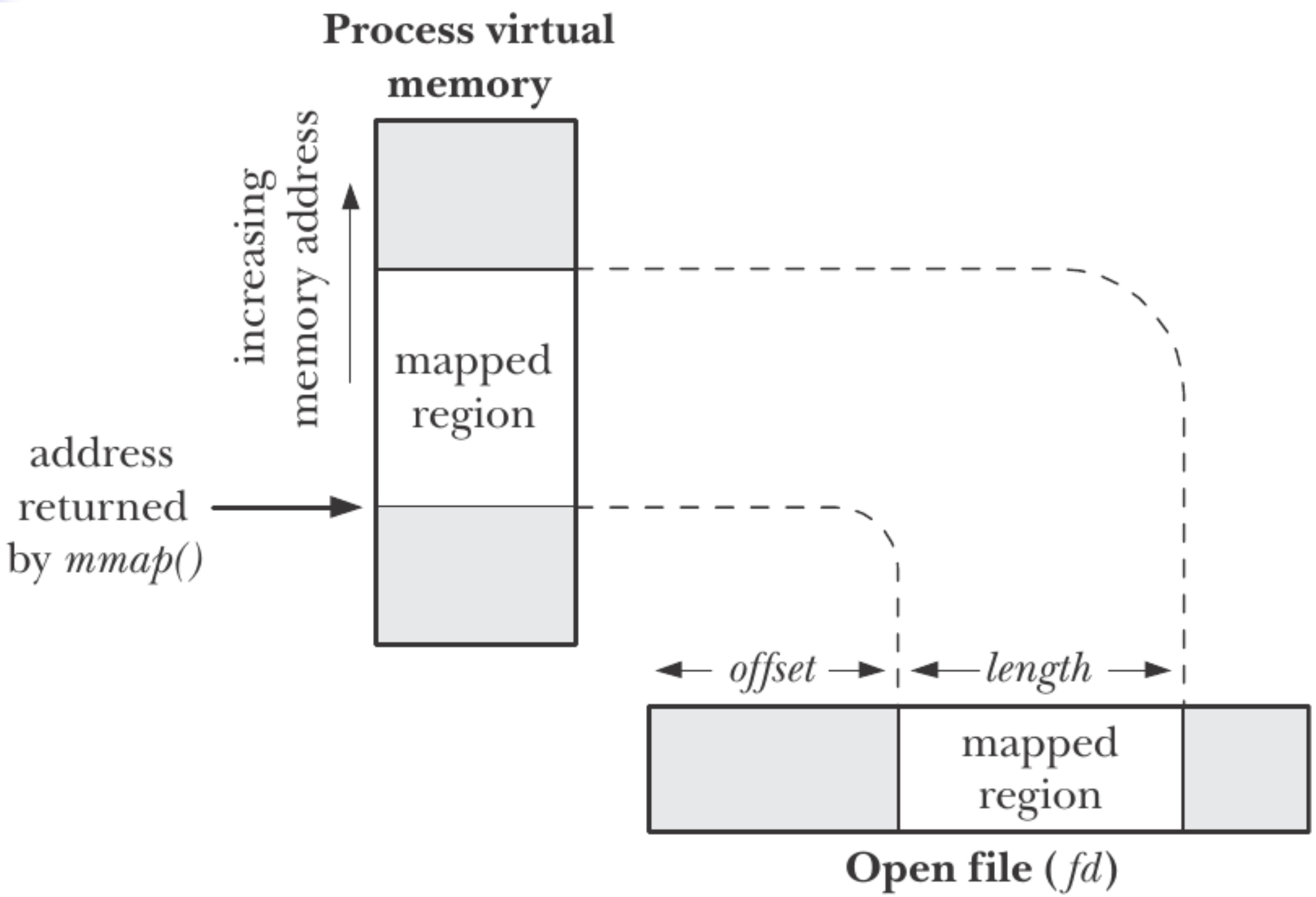
- The mmap() system call creates a new memory mapping in the calling process's virtual address space.
  - File mapping: A file mapping maps a region of a file directly into the calling process's virtual memory.
  - Anonymous mapping: An anonymous mapping doesn't have a corresponding file. Instead, the pages of the mapping are initialized to 0.

mmap

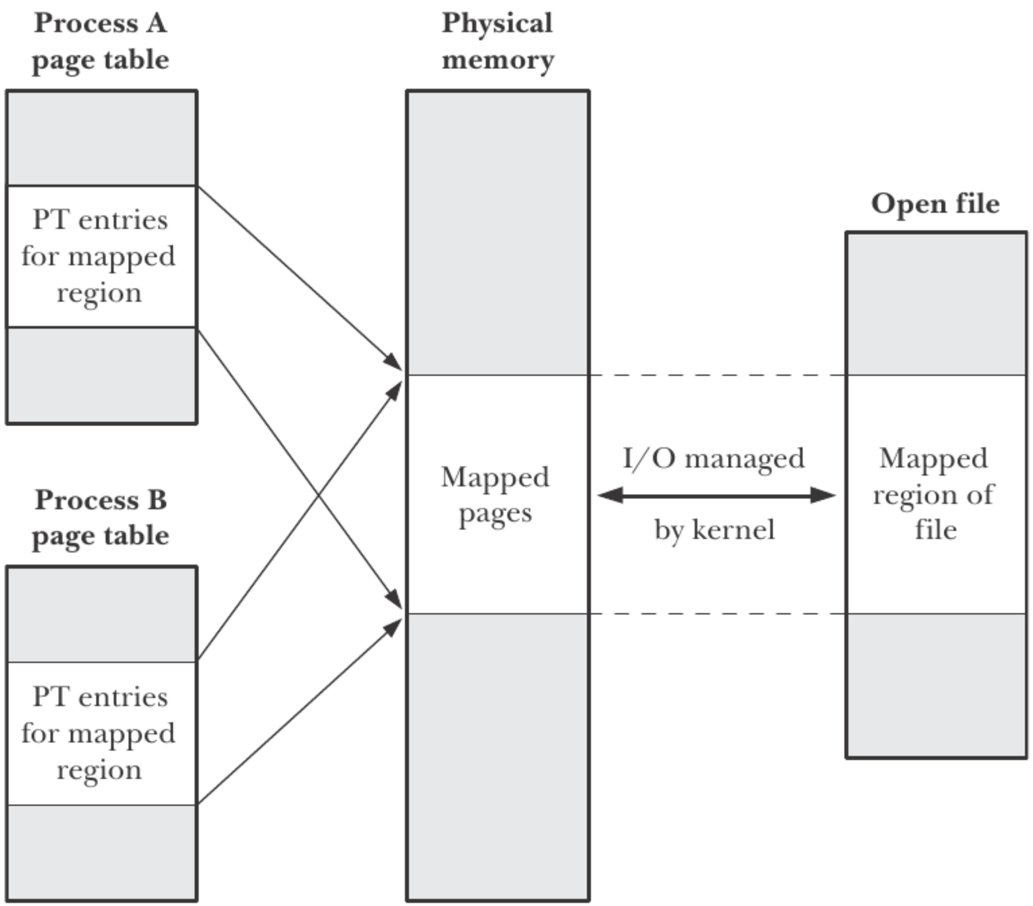
- include <sys/mman.h>
- void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
- Returns starting address of mapping on success, or MAP_FAILED on error

- int munmap(void *addr, size_t length);

(1) File mapping



Two processes with a shared mapping of the same region of a file




```
addr = mmap(NULL, MEM_SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
```

(2) Anonymous mapping

- An anonymous mapping is one that doesn’t have a corresponding file.
  - Specify MAP_ANONYMOUS in *flags* and specify *fd* as –1.
  - Open the /dev/zero device file and pass the resulting file descriptor to *mmap()*.

15.33 IPC between parent and child using memory mapped I/O of /dev/zero

```
#include "apue.h"
#include <fcntl.h>
#include <sys/mman.h>

#define NLOOPS      1000
#define SIZE        sizeof(long)    /* size of shared memory area */

static int
update(long *ptr)
{
    return((*ptr)++);   /* return value before increment */
}
```

```
int
main(void)
{
    int       fd, i, counter;
    pid_t     pid;
    void      *area;

    if ((fd = open("/dev/zero", O_RDWR)) < 0)
        err_sys("open error");
    if ((area = mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,
      fd, 0)) == MAP_FAILED)
        err_sys("mmap error");
    close(fd);      /* can close /dev/zero now that it's mapped */
```

```
    TELL_WAIT();

    if ((pid = fork()) < 0) {
        err_sys("fork error");
    } else if (pid > 0) {             /* parent */
        for (i = 0; i < NLOOPS; i += 2) {
            if ((counter = update((long *)area)) != i)
                err_quit("parent: expected %d, got %d", i, counter);

            TELL_CHILD(pid);
            WAIT_CHILD();
        }
    } else {                          /* child */
        for (i = 1; i < NLOOPS + 1; i += 2) {
            WAIT_PARENT();

            if ((counter = update((long *)area)) != i)
                err_quit("child: expected %d, got %d", i, counter);

            TELL_PARENT(getppid());
        }
    }

    exit(0);
}
```

Anonymous Memory Mapping

- To modify the program in Figure 15.33 to use this facility, we make three changes:
- (a) remove the open of /dev/zero
- (b) remove the close of fd
- (c) change the call to mmap to the following:

```
if ((area = mmap(0, SIZE, PROT_READ | PROT_WRITE,
  MAP_ANON | MAP_SHARED, -1, 0)) == MAP_FAILED)
```

5. Posix IPC

Interface	Message queues	Semaphores	Shared memory
Header file	`<mqueue.h>`	`<semaphore.h>`	`<sys/mman.h>`
Object handle	*mqd_t*	*sem_t* *	*int* (file descriptor)
Create/open	*mq_open()*	*sem_open()*	*shm_open()* + *mmap()*
Close	*mq_close()*	*sem_close()*	*munmap()*
Unlink	*mq_unlink()*	*sem_unlink()*	*shm_unlink()*
Perform IPC	*mq_send()*, *mq_receive()*	*sem_post()*, *sem_wait()*, *sem_getvalue()*	operate on locations in shared region
Miscellaneous operations	*mq_setattr()*—set attributes *mq_getattr()*—get attributes *mq_notify()*—request notification	*sem_init()*—initialize unnamed semaphore *sem_destroy()*—destroy unnamed semaphore	(none)

Comparison of System V IPC and POSIX IPC

- POSIX IPC has the following general advantages when compared to System V IPC:

✓ interface is simpler

✓ the use of names instead of keys, and the open, close, and unlink functions—is more consistent with the traditional UNIX file model

✓ POSIX IPC objects are reference counted. This simplifies object deletion,because we can unlink a POSIX IPC object, knowing that it will be destroyed only when all processes have closed it.

Comparison of System V IPC and POSIX IPC

- However, there is one notable advantage in favor of System V IPC: portability.
- System V IPC is specified in SUSv3 and supported on nearly every UNIX implementation. By contrast, each of the POSIX IPC mechanisms is an optional component in SUSv3. Some UNIX implementations don't support (all of) the POSIX IPC mechanisms.