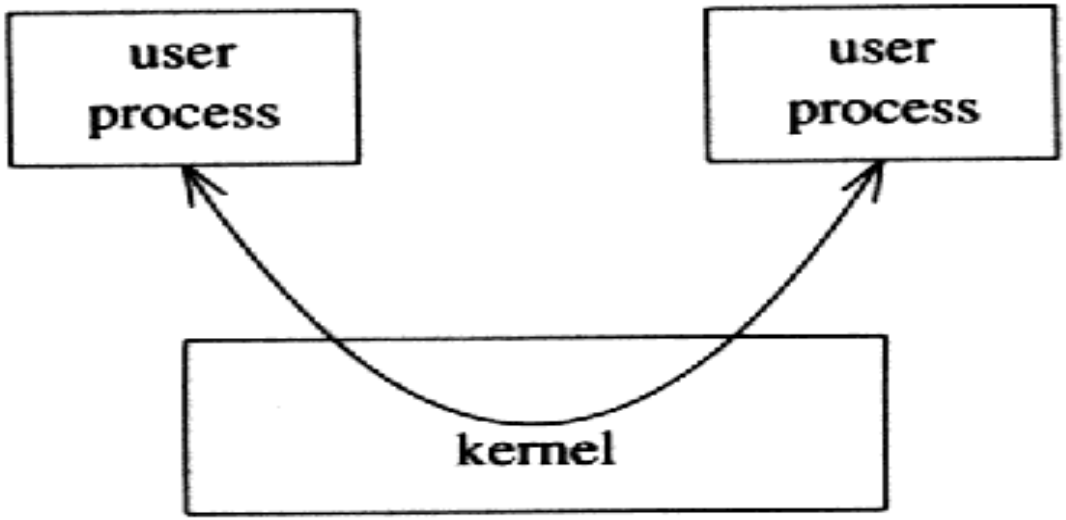
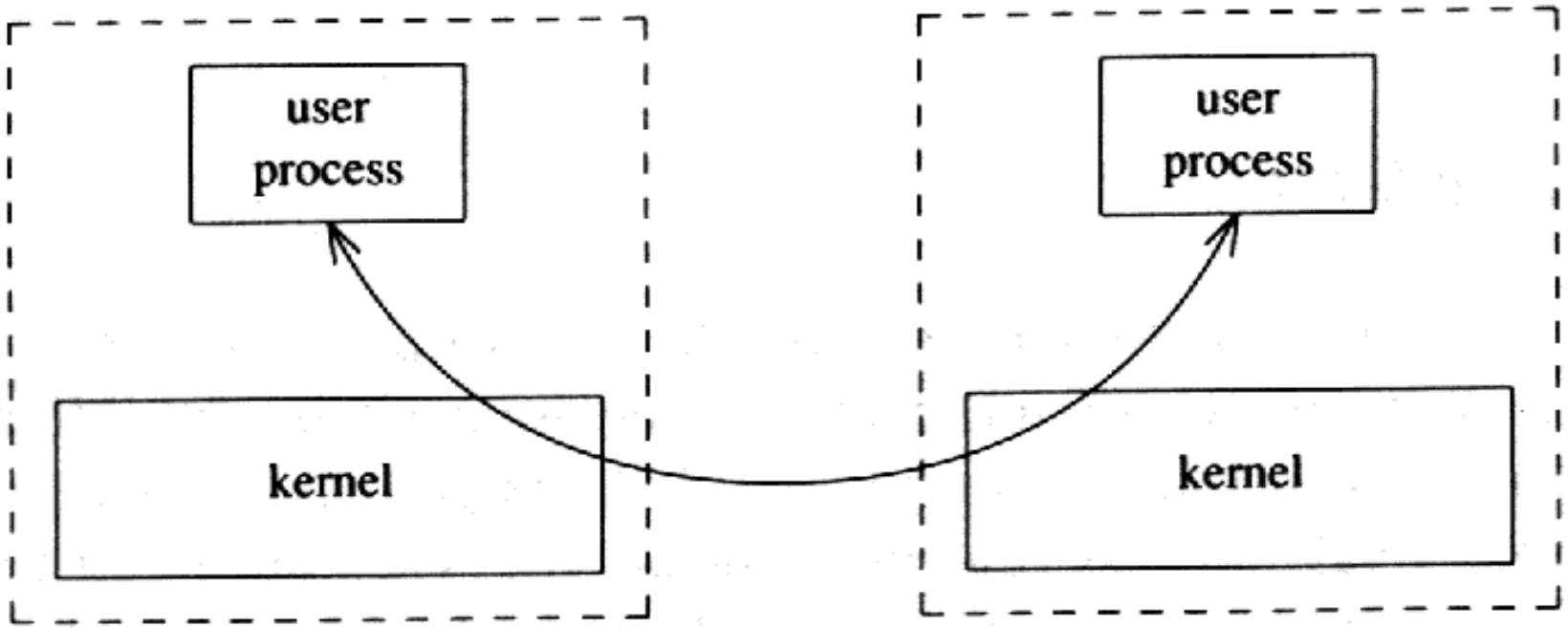
# Interprocess Communication (IPC)

- IPC methods:
  - File locking, pipes, FIFOs, message queues(system V),semaphores, shared memory.
- On a single system



# Interprocess Communication (IPC)

- On different system



# File and Record Locking

- Consider a line printer.
- (lock it)
- it reads the sequence number file
- it uses the number
- it increments the number and writes it back
- (unlock it)

# Program

(lock it)

it reads the sequence

it increments the num

writes it back

(unlock it)

```
#define SEQFILE "seqno"         /* filename */
#define MAXBUFF    100

main()
{
        int     fd, i, n, pid, seqno;
        char    buff[MAXBUFF + 1];

        pid = getpid();
        if ( (fd = open(SEQFILE, 2)) < 0)
                err_sys("can't open %s", SEQFILE);

        for (i = 0; i < 20; i++) {
                my_lock(fd);                    /* lock the file */

                lseek(fd, 0L, 0);               /* rewind before read */
                if ( (n = read(fd, buff, MAXBUFF)) <= 0)
                        err_sys("read error");
                buff[n] = '\0';         /* null terminate for sscanf */

                if ( (n = sscanf(buff, "%d\n", &seqno)) != 1)
                        err_sys("sscanf error");
                printf("pid = %d, seq# = %d\n", pid, seqno);

                seqno++;                /* increment the sequence number */

                sprintf(buff, "%03d\n", seqno);
                n = strlen(buff);
                lseek(fd, 0L, 0);               /* rewind before write */
                if (write(fd, buff, n) != n)

                        err_sys("write error");

                my_unlock(fd);                  /* unlock the file */
        }
}
```

# No Locking

```
/*
 * Locking routines that do nothing.
 */

my_lock(fd)
int     fd;
{
        return;
}

my_unlock(fd)
int     fd;
{
        return;
}
```

# Results for No Locking

```
pid = 186,  seq# = 1
                          (1)
pid = 187,  seq# = 1
pid = 187,  seq# = 2
pid = 187,  seq# = 3
pid = 187,  seq# = 4
pid = 187,  seq# = 5
pid = 187,  seq# = 6
pid = 187,  seq# = 7
pid = 187,  seq# = 8
pid = 187,  seq# = 9
pid = 187,  seq# = 10
                          (2)
pid = 186,  seq# = 2
pid = 186,  seq# = 3
pid = 186,  seq# = 4
pid = 186,  seq# = 5
                          (3)
pid = 187,  seq# = 5
                          (4)
pid = 186,  seq# = 6
pid = 186,  seq# = 7
pid = 186,  seq# = 8
pid = 186,  seq# = 9
```

```
pid = 186,  seq# = 10
pid = 186,  seq# = 11
                          (5)
pid = 187,  seq# = 6
pid = 187,  seq# = 7
pid = 187,  seq# = 8
pid = 187,  seq# = 9
pid = 187,  seq# = 10
                          (6)
pid = 186,  seq# = 12
pid = 186,  seq# = 13
pid = 186,  seq# = 14
pid = 186,  seq# = 15
pid = 186,  seq# = 16
pid = 186,  seq# = 17
pid = 186,  seq# = 18
pid = 186,  seq# = 19
pid = 186,  seq# = 20
                          (7)
pid = 187,  seq# = 11
pid = 187,  seq# = 12
pid = 187,  seq# = 13
pid = 187,  seq# = 14
```

# BSD Locking

```
/*
 * Locking routines for 4.3BSD.
 */

#include        <sys/file.h>

my_lock(fd)
int     fd;
{
        if (flock(fd, LOCK_EX) == -1)
                err_sys("can't LOCK_EX");
}

my_unlock(fd)
int     fd;
{
        if (flock(fd, LOCK_UN) == -1)
                err_sys("can't LOCK_UN");
}
```

# Results for BSD Locking

```
pid = 308, seq# = 1

pid = 307, seq# = 2
pid = 307, seq# = 3
pid = 307, seq# = 4
pid = 307, seq# = 5
pid = 307, seq# = 6
pid = 307, seq# = 7
pid = 307, seq# = 8
pid = 307, seq# = 9
pid = 307, seq# = 10
pid = 307, seq# = 11
pid = 307, seq# = 12
pid = 307, seq# = 13
pid = 307, seq# = 14
pid = 307, seq# = 15

pid = 308, seq# = 16
pid = 308, seq# = 17
pid = 308, seq# = 18
pid = 308, seq# = 19
pid = 308, seq# = 20
pid = 308, seq# = 21
```

```
pid = 308, seq# = 22
pid = 308, seq# = 23
pid = 308, seq# = 24
pid = 308, seq# = 25
pid = 308, seq# = 26
pid = 308, seq# = 27

pid = 307, seq# = 28
pid = 307, seq# = 29
pid = 307, seq# = 30

pid = 308, seq# = 31
pid = 308, seq# = 32
pid = 308, seq# = 33
pid = 308, seq# = 34

pid = 307, seq# = 35
pid = 307, seq# = 36
pid = 307, seq# = 37

pid = 308, seq# = 38
pid = 308, seq# = 39
pid = 308, seq# = 40
```

# Pipes

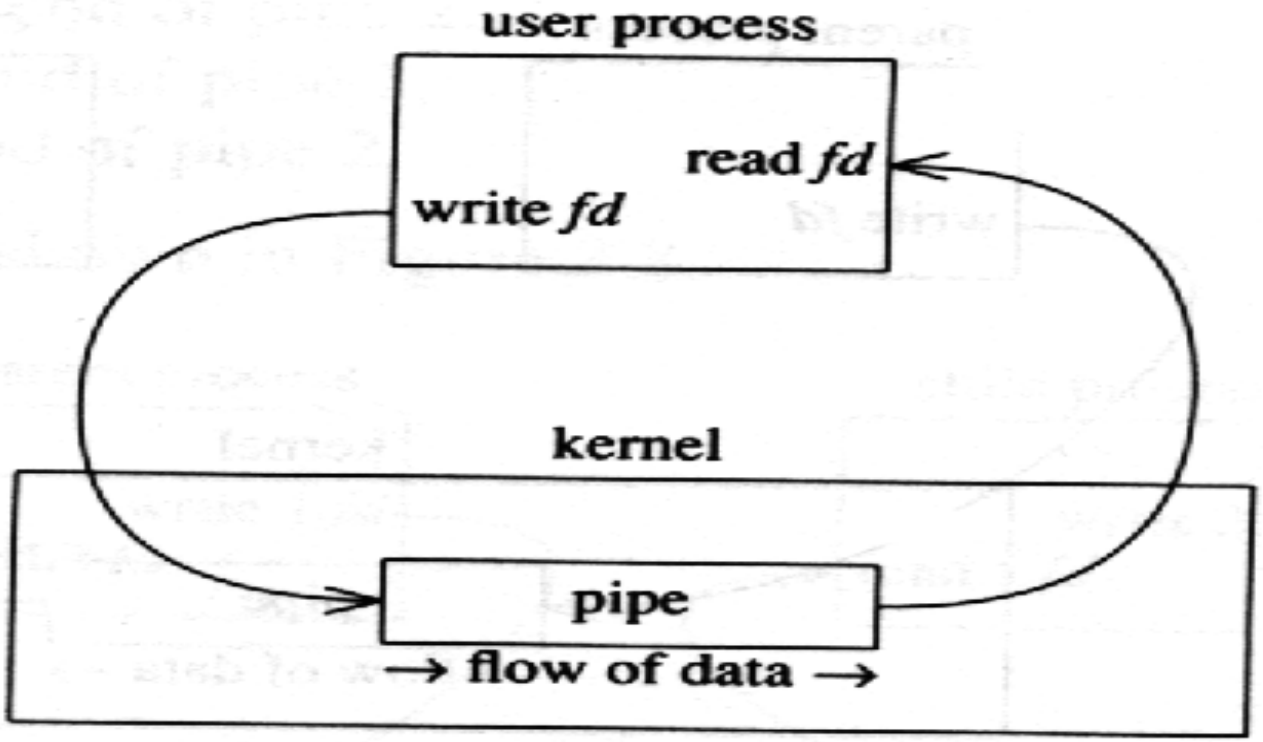



**Figure 3.4** Pipe in a single process.

```
main()
{
        int     pipefd[2], n;
        char    buff[100];

        if (pipe(pipefd) < 0)
                err_sys("pipe error");

        printf("read fd = %d, write fd = %d\n", pipefd[0], pipefd[1]);
        if (write(pipefd[1], "hello world\n", 12) != 12)
                err_sys("write error");

        if ( (n = read(pipefd[0], buff, sizeof(buff))) <= 0)
                err_sys("read error");

        write(1, buff, n);          /* fd 1 = stdout */

        exit(0);
}
```

**The output of this program is**

```
hello world
read fd = 3, write fd = 4
```

# Buffering for printf

- Save data into buffer.
- Output to device when
  - The buffer is full.
  - fflush() is called.
  - exit() is called.
  - For console output,
    - An linefeed is output.
    - The program attempts to read from the terminal.
- Good guidelines for socket programming.
  - Do not mix printf() and write().
  - Do not use printf() for heavy interaction.
    - If you really want, add “flush”.
    - It is ok for CGI-like operations.

# Process-to-Process Piping

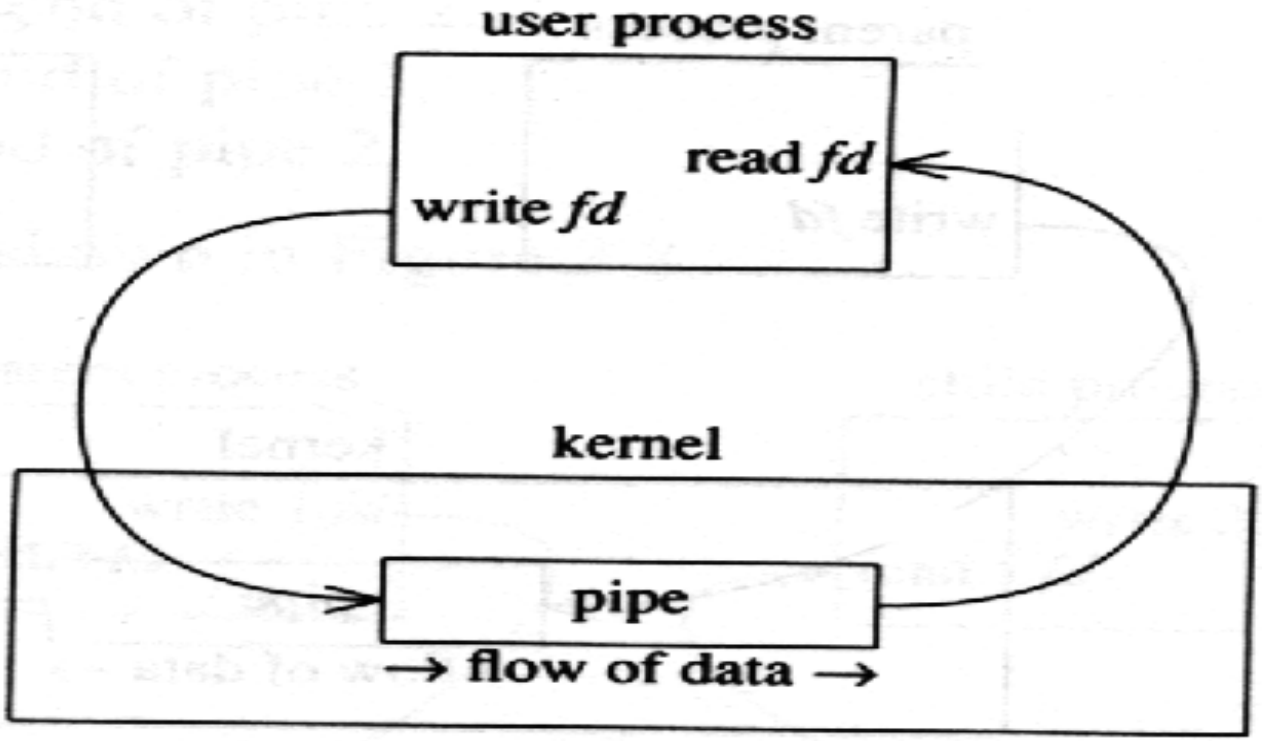
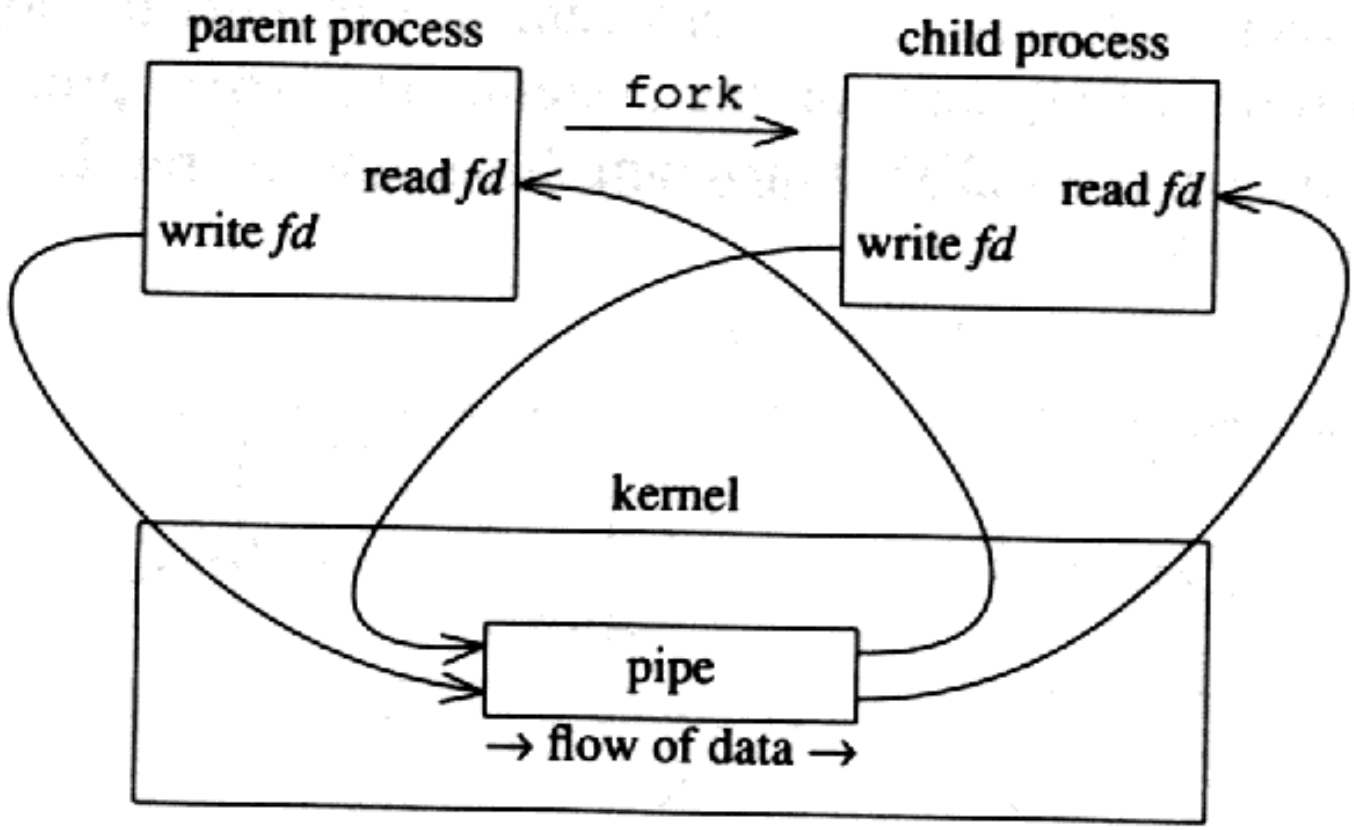
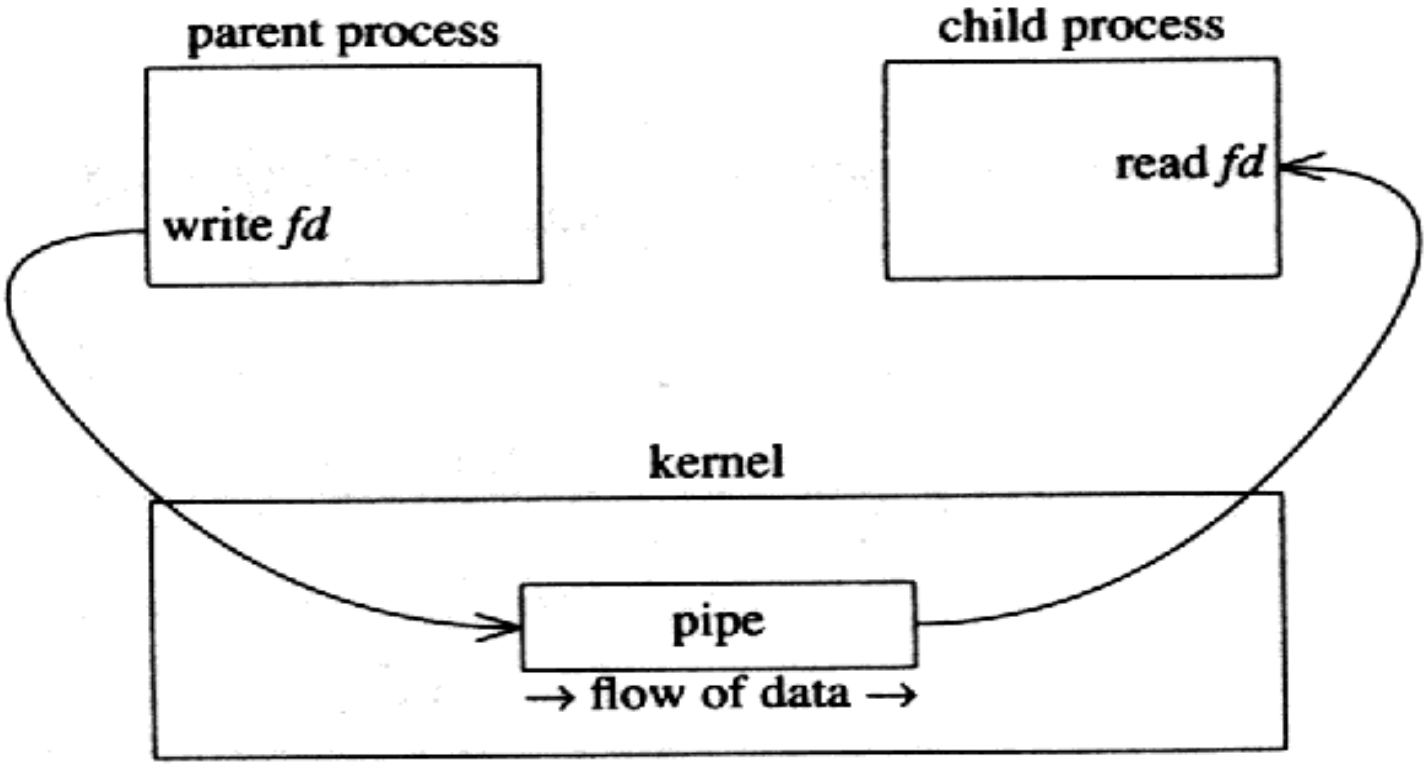
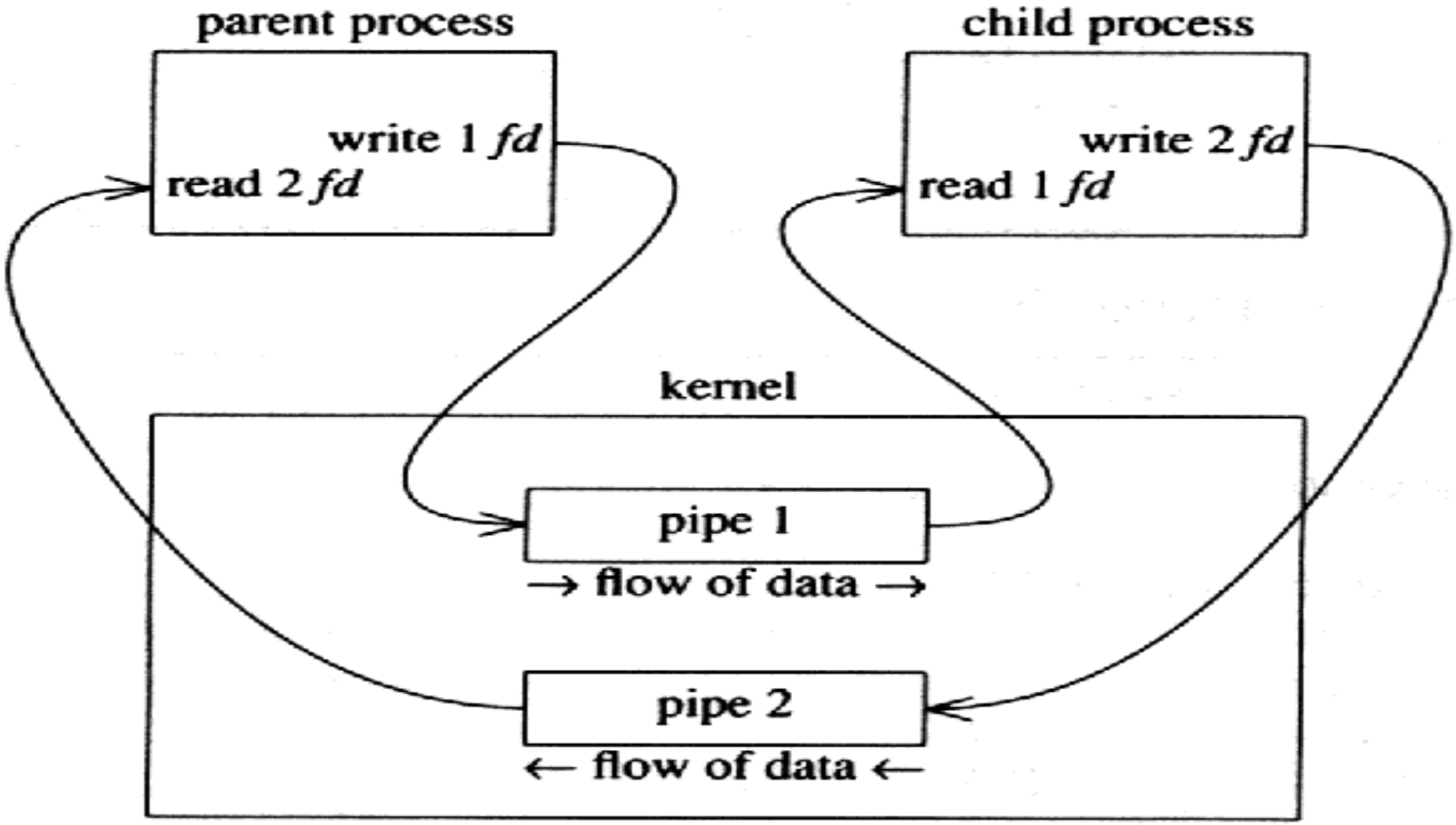



**Figure 3.4** Pipe in a single process.

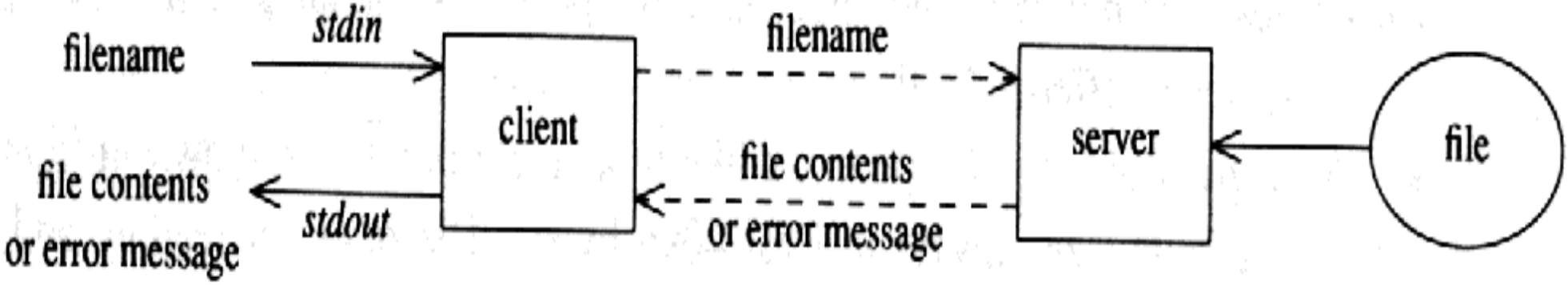

parent process
child process
fork
read fd
write fd
read fd
write fd
kernel
pipe
→ flow of data →


parent process
child process
write fd
read fd
kernel
pipe
→ flow of data →

# Two-Way Pipes



# Pipes with Fork (I)




```
main() {
    int     childpid, pipe1[2], pipe2[2];
    if (pipe(pipe1) < 0 || pipe(pipe2) < 0)  err_sys("can't create pipes");
    if ( (childpid = fork()) < 0) {
            err_sys("can't fork");
```

# Pipes with Fork (II)

```
    } else if (childpid > 0) {                          /* parent */
            close(pipe1[0]);  close(pipe2[1]);
            client(pipe2[0], pipe1[1]);
            while (wait((int *) 0) != childpid)         /* wait for child */
                    ;
            close(pipe1[1]);  close(pipe2[0]);
            exit(0);
    } else {                                                    /* child */
            close(pipe1[1]); close(pipe2[0]);
            server(pipe1[0], pipe2[1]);
            close(pipe1[0]);  close(pipe2[1]);
            exit(0);
    }
}
```

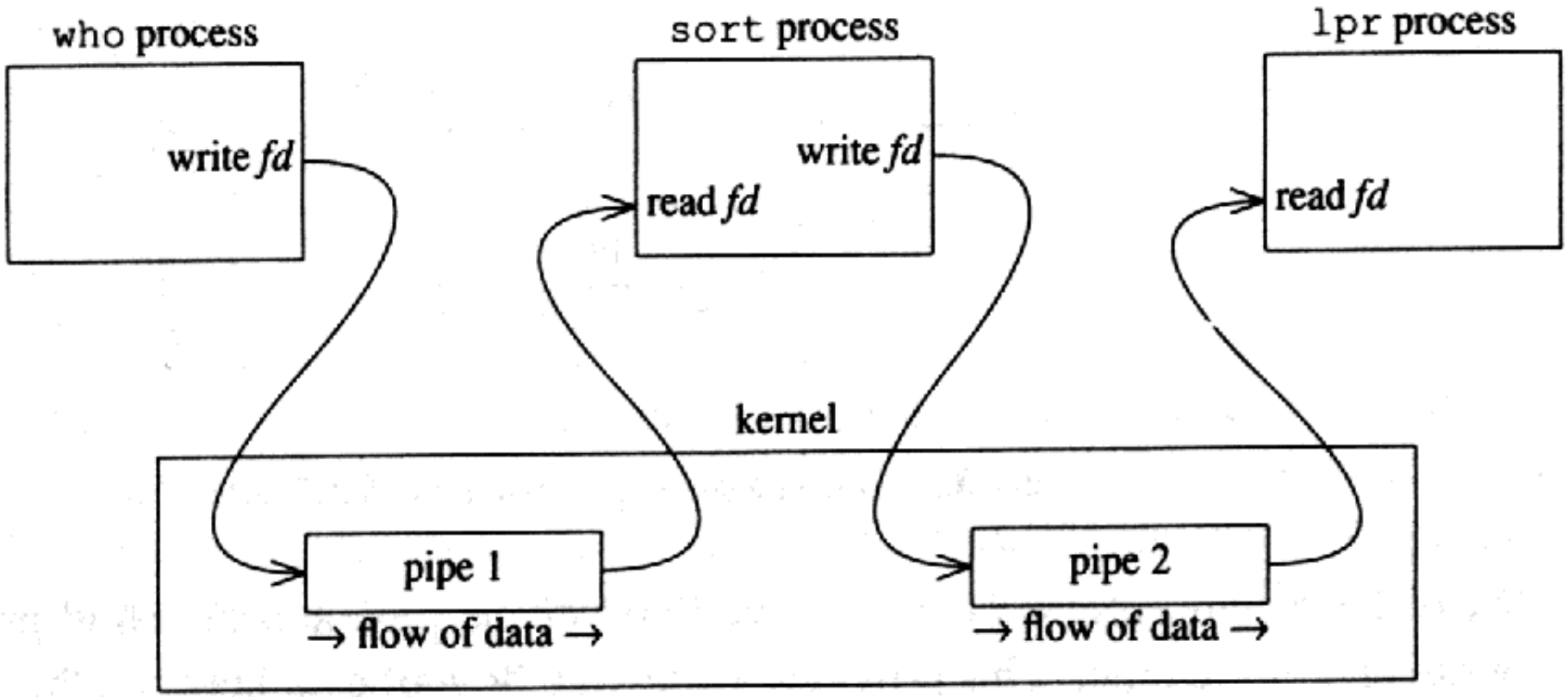
# Client

```
client(int readfd, int writefd)
{
    char   buff[MAXBUFF];
    int    n;

    // Read the filename from standard input, write it to the IPC descriptor.
    if (fgets(buff, MAXBUFF, stdin) == NULL) err_sys( “client: filename read error” );

    n = strlen(buff);
    if (buff[n-1] ==  ‘\n’ ) n--; /* ignore newline from fgets() */
    if (write(writefd, buff, n) != n)  err_sys("client: filename write error");

    // Read the data from the IPC descriptor and write to standard output.
    while ( (n = read(readfd, buff, MAXBUFF)) > 0)
        if (write(1 /* stdout*/, buff, n) != n)    err_sys("client: data write error");
    if (n < 0) err_sys("client: data read error");
}
```

# Server

```
server(int readfd, int writefd)
{
        char    buff[MAXBUFF],  errmesg[256], *sys_err_str();
        int     n, fd;

        // Read the filename from the IPC descriptor.
        if ( (n = read(readfd, buff, MAXBUFF)) <= 0) err_sys("server: filename read error");
        buff[n] = '\0';                         /* null terminate filename */

        if ( (fd = open(buff, 0)) < 0) {
                // Error.  Format an error message and send it back to the client.
                sprintf(errmesg, ": can't open, %s\n", sys_err_str());
                strcat(buff, errmesg);
                n = strlen(buff);
                if (write(writefd, buff, n) != n) err_sys("server: errmesg write error");
        } else {
                // Read the data from the file and write to the IPC descriptor.
                while ( (n = read(fd, buff, MAXBUFF)) > 0)
                    if (write(writefd, buff, n) != n) err_sys("server: data write error");
                if (n < 0) err_sys("server: read error");
        }
}
```

# Unix Shell Pipes



# SIGPIPE

What if none of processes read or write a pipe?

- When trying to read from a pipe that has no write end,
  - the **read()** returns 0.
- When you try to write to a pipe that has no read end,
  - a **SIGPIPE** signal is generated.
  - If the signal isn't handled, the program exits silently.
  - A nice way for the example

    UNIX> cat exec1.c | head -5 | tail -1
    - **head** exits when receiving 5 lines,
    - **tail** will have **read()** return 0, and will exit, and
    - **cat** will try to write to an empty pipe, and thus will generate **SIGPIPE** and exit.

  See http://www.cs.utk.edu/~plank/plank/classes/cs360/360/notes/Pipe/lecture.html

# FIFOs

Stands for first-in-first-out.(System V only)

Main problem of pipes:

- can only be used between processes that have a parent process in common.

- Server

  mknod(pathname, mode, dev)

- Client

  open(pathname, flag)

# Example: FIFO Server

```
#define     FIFO1       "/tmp/fifo.1"
#define     FIFO2       "/tmp/fifo.2"
main() {
    int     readfd, writefd;
    // Create the FIFOs, then open them - one for reading and one for writing.
    if ( (mknod(FIFO1, S_IFIFO | PERMS, 0) < 0) && (errno != EEXIST))
            err_sys("can't create fifo: %s", FIFO1);
    if ( (mknod(FIFO2, S_IFIFO | PERMS, 0) < 0) && (errno != EEXIST)) {
            unlink(FIFO1);
            err_sys("can't create fifo: %s", FIFO2);
    }
    if ( (readfd = open(FIFO1, 0)) < 0)
            err_sys("server: can't open read fifo: %s", FIFO1);
    if ( (writefd = open(FIFO2, 1)) < 0)
            err_sys("server: can't open write fifo: %s", FIFO2);
    server(readfd, writefd);
    close(readfd);
    close(writefd);
}
```

# Example: FIFO Client

```
main() {
    int     readfd, writefd;
    // Open the FIFOs.  We assume the server has already created them.
    if ( (writefd = open(FIFO1, 1)) < 0)
            err_sys("client: can't open write fifo: %s", FIFO1);
    if ( (readfd = open(FIFO2, 0)) < 0)
            err_sys("client: can't open read fifo: %s", FIFO2);
    client(readfd, writefd);
    close(readfd);
    close(writefd);
    // Delete the FIFOs, now that we're finished.
    if (unlink(FIFO1) < 0)   err_sys("client: can't unlink %s", FIFO1);
    if (unlink(FIFO2) < 0)   err_sys("client: can't unlink %s", FIFO2);
}
```

# FIFOs over Two Hosts

What if Client (Host) A makes file mounting on Server B?

- Create a FIFO on B.
- Both processes on A can communicate via it correctly.
- One on A and the other one B cannot communicate.
- For different Clients:

| Processes A&B run on | Does the FIFO work? |
|---|---|
| same client | Yes |
| different client | No |

- References:
  - http://hissa.nist.gov/rbac/titleissues/node24.html
  - https://mail.rtai.org/pipermail/rtai/2006-July/015566.html

# Message Queues

- Message queues: (supported by System V)

  Message v.s. Stream:

  – Message:has boundary.

  – Stream: no boundary for two messages.

# Semaphores

- This does not really send data as read/write/pipe/fifo, simply does synchronous operations to protect shared memory.
- Semaphores are stored in kernel.

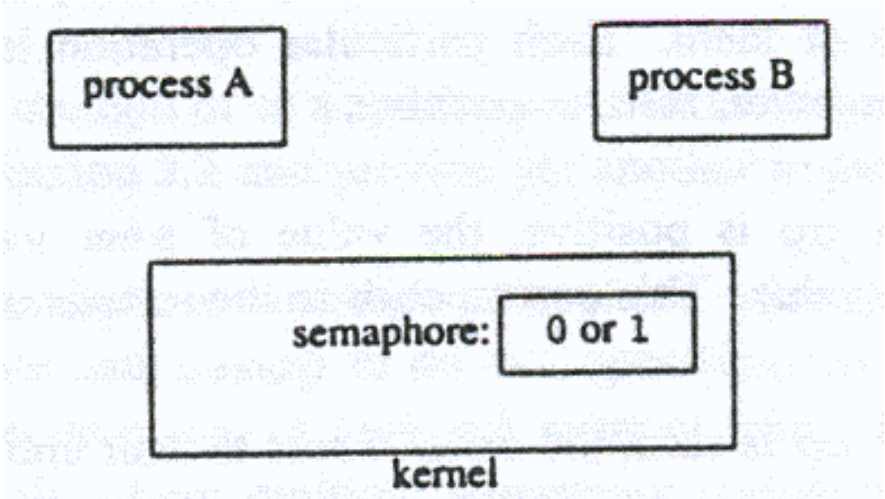



**Figure 3.20** Semaphore value stored in kernel.

# Kernel Data Structures for Semaphores

- Usually, a set of semaphores (in arrays), not just one.

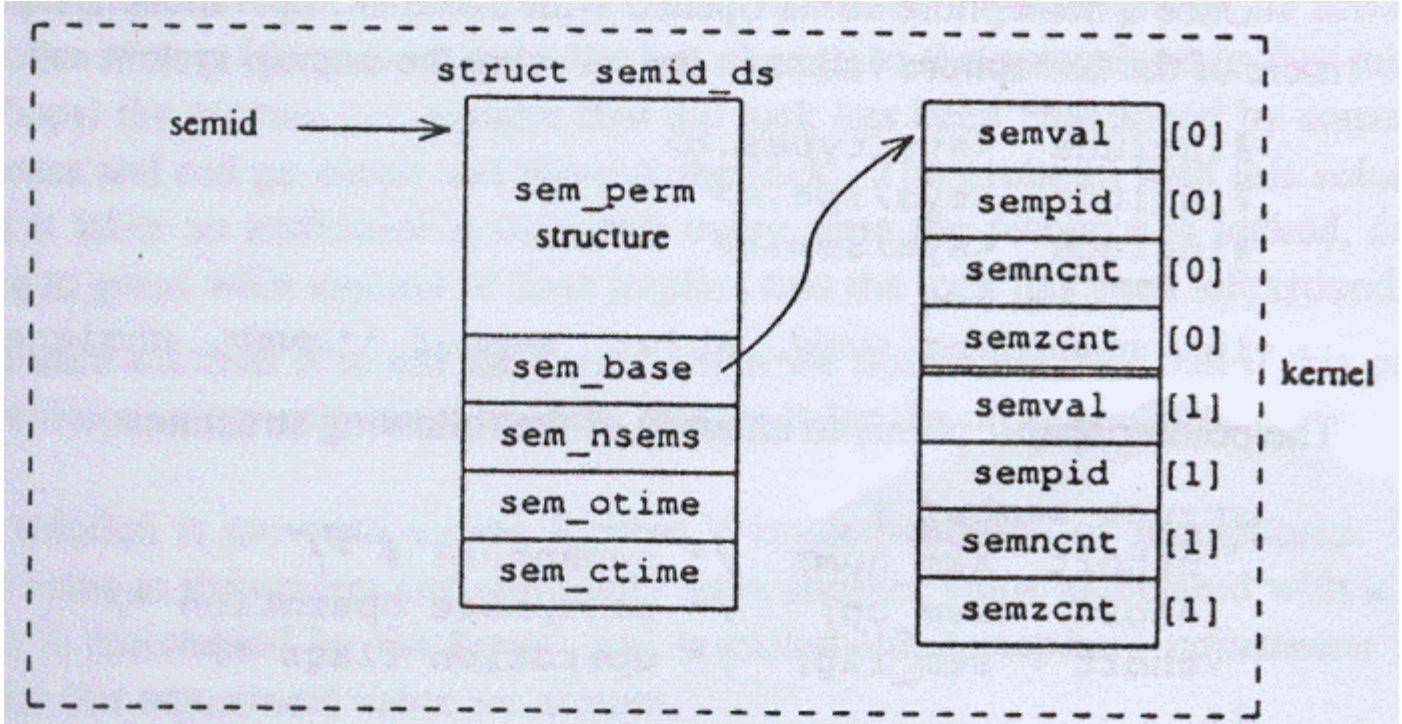



Figure 3.21 Kernel data structures for a semaphore set.

# Allocating Semaphores

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

int semget(key_t key, int nsems, int semflag);
```

| Numeric | Symbolic | Description |
|---|---|---|
| 0400 | SEM_R | Read by owner |
| 0200 | SEM_A | Alter by owner |
| 0040 | SEM_R >> 3 | Read by group |
| 0020 | SEM_A >> 3 | Alter by group |
| 0004 | SEM_R >> 6 | Read by world |
| 0002 | SEM_A >> 6 | Alter by world |
| | IPC_CREAT | (See Section 3.8) |
| | IPC_EXCL | (See Section 3.8) |

# semop

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

int semop(int semid, struct sembuf * opsptr, unsigned int nops);
```

pointer *opsptr* points to an array of the following structures:

```
struct sembuf {
  ushort  sem_num;   /* semaphore # */
  short   sem_op;    /* semaphore operation */
  short   sem_flg;   /* operation flags */
};
```

Note: more like ID.

# Semaphore Operations

- If sem_op is positive, the value of sem_val is added.
- If sem_op is zero, the caller wants to wait until the semaphore becomes zero.
- If sem_op is negative, the caller wants to wait until the semaphore becomes greater than or equal to the absolute value of sem_op. Then, add the value into sem_val.
  (Just like subtract)

Returning value:

- 0, if ok.
- -1, for an error.

# File Locking with Semaphore

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

#define    SEMKEY 123456L  /* key value for semget() */
#define    PERMS   0666

static struct sembuf op_lock[2] = {
    0, 0, 0,                    /* wait for sem#0 to become 0 */
    0, 1, 0                     /* then increment sem#0 by 1 */
};

static struct sembuf op_unlock[1] = {
    0, -1, IPC_NOWAIT      /* decrement sem#0 by 1 (sets it to 0) */
};

int  semid = -1;        /* semaphore id */
```

# File Locking with Semaphore (cont.)

```
my_lock(int fd)
{
    if (semid < 0) {
        if ( (semid = semget(SEMKEY, 1, IPC_CREAT | PERMS)) < 0)
            err_sys("semget error");
    }
    if (semop(semid, &op_lock[0], 2) < 0)
        err_sys("semop lock error");
}

my_unlock(fd)
int  fd;
{
    if (semop(semid, &op_unlock[0], 1) < 0)
        err_sys("semop unlock error");
}
```

# Problems

- If a process aborts while locking,
  - the semaphore value remains one!!!

Possible solutions:

- Catch all signals and use signal handlers to unlock it.
  - But, SIGKILL cannot be caught.
- Let the first sem_op not wait, get the sem_ctime to check if timeout happens.
  - Too sophisticated and does not solve the problem really.
- Let the kernel undo it, while the process aborts.
  - Use “SEM_UNDO” flag to tell kernel to undo.

# File Locking with Semaphore Undo

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

#define    SEMKEY 123456L  /* key value for semget() */
#define    PERMS   0666

static struct sembuf op_lock[2] = {
    0, 0, 0,                    /* wait for sem#0 to become 0 */
    0, 1, SEM_UNDO              /* then increment sem#0 by 1 */
};

static struct sembuf op_unlock[1] = {
    0, -1, (IPC_NOWAIT | SEM_UNDO)
                                /* decrement sem#0 by 1 (sets it to 0) */
};
```

# Problems Still

- The semaphore is never removed.
  - Though we can use semctl() to remove it, a process that aborts may not have chance to call this.
  - The my_lock() code between semget() and semop() is not atomic.

# semctl

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

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

union semun {
  int              val;    /* used for SETVAL only */
  struct semid_ds  *buff;  /* used for IPC_STAT and IPC_SET */
  ushort           *array; /* used for IPC_GETALL & IPC_SETALL */
} arg;
```

- Remove a lock.
  semctl(semid, 0, IPC_RMID, (struct semun *) 0)
- Get and Set semaphore values.
  semval = semctl(id, 1, GETVAL, 0)
  semctl(id, 2, SETVAL, semctl_arg)

# A Robust Semaphore

- Use 3 semaphore values
  1. The real semaphore value
  2. The counter of the number of processes using this semaphore.
  3. A lock variable for the semaphore. (Used for a critical section in code.)
- Provide a simpler and easier to understand interface: 7 routines

```
id = sem_create(key, initval);      # create with initial value or open
id = sem_open(key);                 # open (must already exist)
sem_wait(id);                       # wait = P = down by 1
sem_signal(id);                     # signal = V = up by 1
sem_op(id, amount);                 # wait   if (amount < 0)
                                    # signal if (amount > 0)
sem_close(id);                      # close
sem_rm(id);                         # remove (delete)
```

# sem_create() — (1)

```
int sem_create(key_t key, int initval)
{
    register int                id, semval;
    if (key == IPC_PRIVATE) return(-1);  /* not intended for private semaphores */
    else if (key == (key_t) -1)   return(-1);      /* probably an ftok() error by caller */

again:
    if ( (id = semget(key, 3, 0666 | IPC_CREAT)) < 0) return(-1);
    if (semop(id, &op_lock[0], 2) < 0) {
      if (errno == EINVAL) goto again;
      err_sys("can't lock");
    }
```

```
static struct sembuf op_lock[2] = {
          2, 0, 0,              /* wait for [2] (lock) to equal 0 */
          2, 1, SEM_UNDO        /* then increment [2] to 1 - this locks it */
                                /* UNDO to release the lock if processes exits
                                   before explicitly unlocking */
};
```

# sem_create() — (2)

```
if ( (semval = semctl(id, 1, GETVAL, 0)) < 0) err_sys("can't GETVAL");
if (semval == 0) {
  semctl_arg.val = initval;
  if (semctl(id, 0, SETVAL, semctl_arg) < 0)
        err_sys("can't SETVAL[0]");
  semctl_arg.val = BIGCOUNT;
  if (semctl(id, 1, SETVAL, semctl_arg) < 0)
        err_sys("can't SETVAL[1]");
}
if (semop(id, &op_endcreate[0], 2) < 0)
  err_sys("can't end create");

return(id);
}
```

```
static struct sembuf op_endcreate[2] = {
        1, -1, SEM_UNDO,/* decrement [1] (proc counter) with undo on exit */
                        /* UNDO to adjust proc counter if process exits
                           before explicitly calling sem_close() */
        2, -1, SEM_UNDO     /* decrement [2] (lock) back to 0 -> unlock */
};
```

# sem_rm()

```
sem_rm(int id)
{
    if (semctl(id, 0, IPC_RMID, 0) < 0)
      err_sys("can't IPC_RMID");
}
```

# sem_open()

```
int sem_open(key_t key)
{
    register int  id;
    if (key == IPC_PRIVATE) return(-1);
    else if (key == (key_t) -1) return(-1);

    if ( (id = semget(key, 3, 0)) < 0) return(-1);    /* doesn't exist, or tables full */

    // Decrement the process counter.  We don't need a lock to do this.
    if (semop(id, &op_open[0], 1) < 0) err_sys("can't open");

    return(id);
}
```

```
static struct sembuf op_open[1] = {
        1, -1, SEM_UNDO      /* decrement [1] (proc counter) with undo on exit */
};
```

# sem_close()

```
static struct sembuf op_close[3] = {
        2, 0, 0,   /* wait for [2] (lock) to equal 0 */
        2, 1, SEM_UNDO,/* then increment [2] to 1 - this locks it */
        1, 1, SEM_UNDO  /* then increment [1] (proc counter) */
};
```

```
sem_close(int id)
{
    register int  semval;
    // The following semop() first gets a lock on the semaphore,
    // then increments [1] - the process counter.
    if (semop(id, &op_close[0], 3) < 0) err_sys("can't semop");

    // if this is the last reference to the semaphore, remove this.
    if ( (semval = semctl(id, 1, GETVAL, 0)) < 0) err_sys("can't GETVAL");
    if (semval > BIGCOUNT) err_dump("sem[1] > BIGCOUNT");
    else if (semval == BIGCOUNT) sem_rm(id);
    else
      if (semop(id, &op_unlock[0], 1) < 0) err_sys("can't unlock");  /* unlock */
}
```

```
static struct sembuf op_unlock[1] = {
        2, -1, SEM_UNDO     /* decrement [2] (lock) back to 0 */
};
```

# P/V Operations

```
sem_op(int id, int value)
{
    if ( (op_op[0].sem_op = value) == 0)  err_sys("can't have value == 0");
    if (semop(id, &op_op[0], 1) < 0)   err_sys("sem_op error");
}

sem_wait(int id)
{
    sem_op(id, -1);
}

sem_signal(int id)
{
    sem_op(id, 1);
}
```

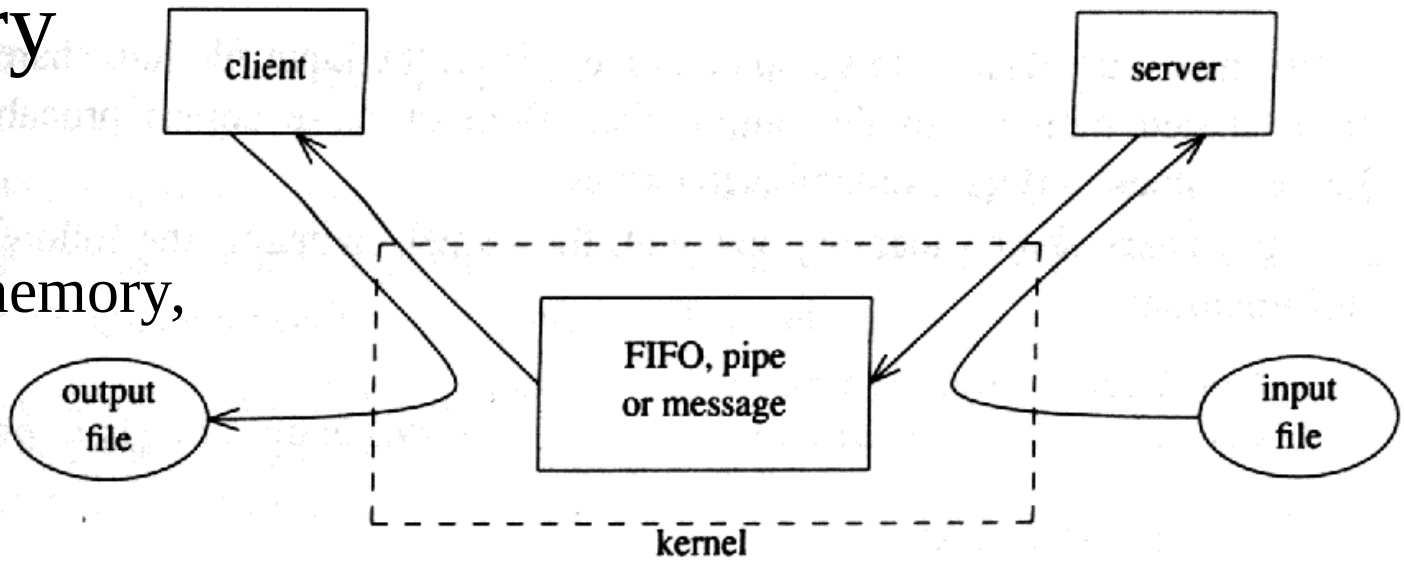
```
static struct sembuf op_op[1] = {
     0, 99, SEM_UNDO /* decrement or increment [0] with undo on exit */
                     /* the 99 is set to the actual amount to add
                                or subtract (positive or negative) */
};
```

# Locking with Semaphores

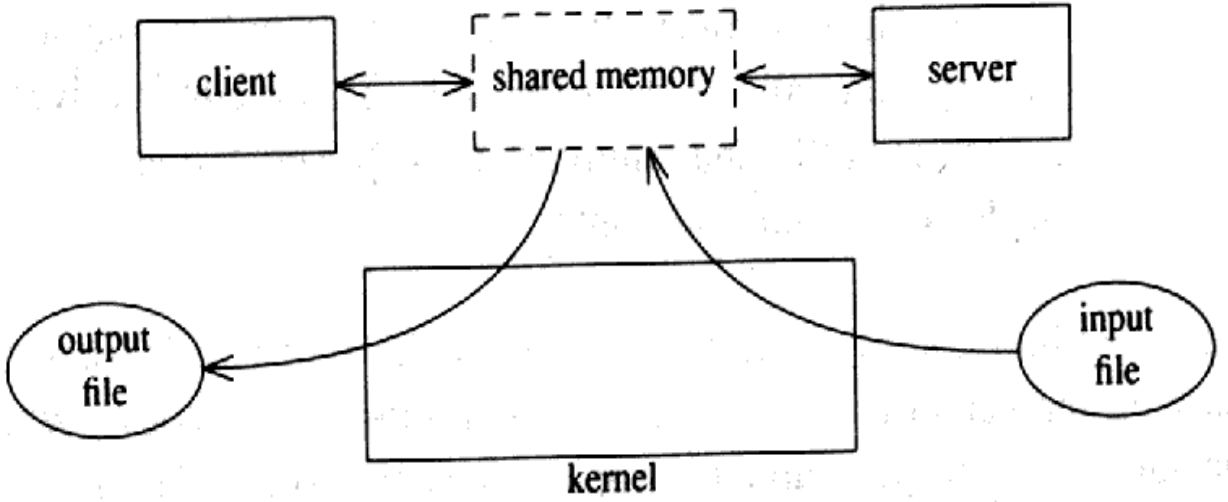
```
#define         SEMKEY          123456L /* key value for sem_create() */
int semid = -1; /* semaphore id */

my_lock(int fd)
{
    if (semid < 0) {
      if ( (semid = sem_create(SEMKEY, 1)) < 0) err_sys("sem_create error");
    }
    sem_wait(semid);
}

my_unlock(int fd)
{
    sem_signal(semid);
}
```

# Shared Memory

- Without shared memory, too many copies.




- Our next example:



# System Call – shmget

- Create a shared memory segment or access an existing one.

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

int shmget(key_t key, int size, int shmflag);
```

- Return *shmid*, or –1 for error.
- *size*: the size of the segment
- *shmflag*: the flags listed right.

| Numeric | Symbolic | Description |
|---|---|---|
| 0400 | SHM_R | Read by owner |
| 0200 | SHM_W | Write by owner |
| 0040 | SHM_R >> 3 | Read by group |
| 0020 | SHM_W >> 3 | Write by group |
| 0004 | SHM_R >> 6 | Read by world |
| 0002 | SHM_W >> 6 | Write by world |
| | IPC_CREAT | (See Section 3.8) |
| | IPC_EXCL | (See Section 3.8) |

# System Call – shmat

- Attach the shared memory segment.

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

char *shmat(int shmid, char *shmaddr, int shmflag);
```

- *shmaddr*:
  - 0: the system selects the address for the caller.
  - Nonzero:
    - SHM_RND value is not specified:
      - attached at the specified address, *shmaddr*.
    - SHM_RND value is specified:
      - attached at the specified address, *shmaddr*, but rounded down by SHMLBA (in *shmflag*).
- *shmflag*:
  - SHM_RDONLY: “read-only” access.
  - SHMLBA (See above).

# System Call – shmdt

- Detach the shared memory segment.

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

int shmdt(char *shmaddr);
```

- This does not really delete the shared memory segment.

# System Call – shmctl

- To remove a shared memory, use this with cmd “IPC_RMID”.

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

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

# Header File – shm.h

```
#define     SHMKEY      ((key_t) 7890) /* base value for shmem key */
#define     SEMKEY1     ((key_t) 7891) /* client semaphore key */
#define     SEMKEY2     ((key_t) 7892) /* server semaphore key */
#define     PERMS   0666
int shmid, clisem, servsem;     /* shared memory and semaphore IDs */

#define     MAXMESGDATA (4096-16)
#define     MESGHDRSIZE (sizeof(Mesg) - MAXMESGDATA)
typedef struct {
  int  mesg_len;                /* #bytes in mesg_data, can be 0 or > 0 */
  long        mesg_type;        /* message type, must be > 0 */
  char        mesg_data[MAXMESGDATA];
} Mesg;
Mesg        *mesgptr;           /* ptr to message structure, which is
                                   in the shared memory segment */
```

# Client – main()

```
main() {
    // Get the shared memory segment and attach it.
    // The server must have already created it.
    if ( (shmid = shmget(SHMKEY, sizeof(Mesg), 0)) < 0) err_sys("…");
    if ( (mesgptr = (Mesg *) shmat(shmid, (char *) 0, 0)) == -1) err_sys("…");
    // Open the two semaphores.  The server must have created them already.
    if ( (clisem = sem_open(SEMKEY1)) < 0) err_sys("…");
    if ( (servsem = sem_open(SEMKEY2)) < 0) err_sys("…");
    client();
    // Detach and remove the shared memory segment and close the semaphores.
    if (shmdt(mesgptr) < 0) err_sys("…");
    if (shmctl(shmid, IPC_RMID, (struct shmid_ds *) 0) < 0) err_sys("…");
    sem_close(clisem);          /* will remove the semaphore */
    sem_close(servsem);         /* will remove the semaphore */
    exit(0);
}
```

# Server – main()

```
main() {
    // Create the shared memory segment, if required, then attach it.
    if ((shmid=shmget(SHMKEY, sizeof(Mesg), PERMS|IPC_CREAT))<0)
      err_sys("server: can't get shared memory");
    if ( (mesgptr = (Mesg *) shmat(shmid, (char *) 0, 0)) == -1) err_sys("…");
    // Create two semaphores.  The client semaphore starts out at 1
    // since the client process starts things going.
    if ( (clisem = sem_create(SEMKEY1, 1)) < 0) err_sys("…");
    if ( (servsem = sem_create(SEMKEY2, 0)) < 0) err_sys("…");
    server();
    // Detach the shared memory segment and close the semaphores.
    // The client is the last one to use the shared memory, so it'll remove it at last.
    if (shmdt(mesgptr) < 0) err_sys("server: can't detach shared memory");
    sem_close(clisem);
    sem_close(servsem);
}
```

# Client – client()

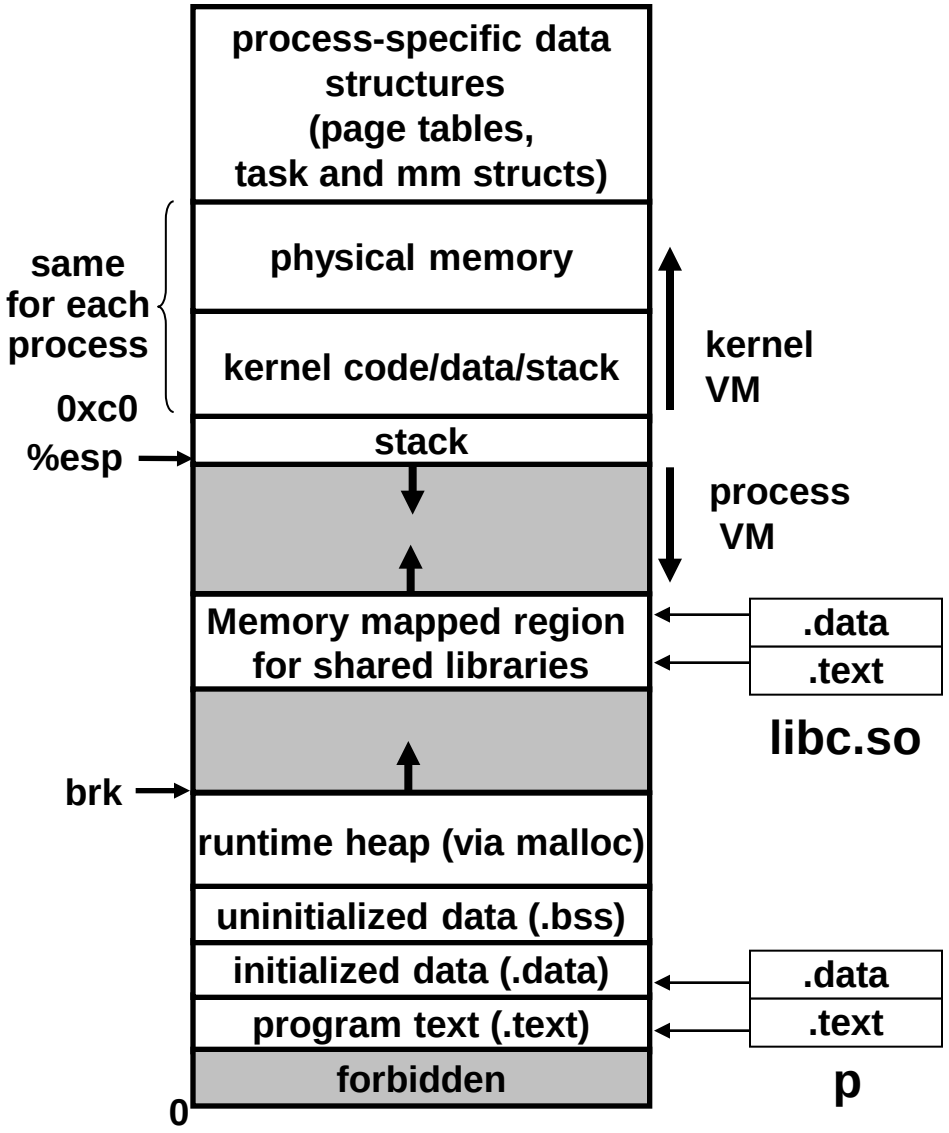
```
client() {
    // Read the filename from standard input, write it to shared memory.
    sem_wait(clisem);              /* get control of shared memory */
    if (fgets(mesgptr->mesg_data, MAXMESGDATA, stdin) == 0) err_sys("…");
    n = strlen(mesgptr->mesg_data);
    if (mesgptr->mesg_data[n-1] == '\n') n--; /* ignore newline from fgets() */
    mesgptr->mesg_len = n;
    sem_signal(servsem);                    /* wake up server */
    // Wait for the server to place something in shared memory.
    sem_wait(clisem);              /* wait for server to process */
    while( (n = mesgptr->mesg_len) > 0) {
      if (write(1, mesgptr->mesg_data, n) != n) err_sys("data write error");
      sem_signal(servsem);         /* wake up server */
      sem_wait(clisem);            /* wait for server to process */
    }
    if (n < 0) err_sys("data read error");
}
```

# Server – server()

```
server() {
    int         n, filefd;
    char        errmesg[256], *sys_err_str();

    // Wait for the client to write the filename into shared memory.
    sem_wait(servsem);          /* we'll wait here for client to start things */
    mesgptr->mesg_data[mesgptr->mesg_len] = '\0';

    if ( (filefd = open(mesgptr->mesg_data, 0)) < 0) {
      // Error.  Format an error message and send it back to the client.
      sprintf(errmesg, ": can't open, %s\n", sys_err_str());
      strcat(mesgptr->mesg_data, errmesg);
      mesgptr->mesg_len = strlen(mesgptr->mesg_data);
      sem_signal(clisem);                   /* send to client */
      sem_wait(servsem);                    /* wait for client to process */
```

# Server – server() (cont.)

```
    } else {
      // Read the data from the file right into shared memory.
      while ( (n = read(filefd, mesgptr->mesg_data, MAXMESGDATA-1)) >
    0) {
                mesgptr->mesg_len = n;
                sem_signal(clisem);        /* send to client */
                sem_wait(servsem);         /* wait for client to process */
      }
      close(filefd);
      if (n < 0) err_sys("server: read error");
    }
    // Send a message with a length of 0 to signify the end.
    mesgptr->mesg_len = 0;
    sem_signal(clisem);
}
```


process-specific data structures (page tables, task and mm structs)
physical memory
kernel code/data/stack
same for each process
0xc0
kernel VM
stack
%esp
process VM
Memory mapped region for shared libraries
.data
.text
libc.so
brk
runtime heap (via malloc)
uninitialized data (.bss)
initialized data (.data)
program text (.text)
forbidden
.data
.text
p
0