Chapter 15
Classical Inter-Process Communication

Cheng-Hsin Hsu
National Tsing Hua University
Department of Computer Science

Parts of the course materials are courtesy of Prof. Chun-Ying Huang

Introduction

- We have described the process control primitives and seen how to invoke multiple processes
- How does a process communicate with other processes?
- The inter-process communication (IPC)

Common IPC Mechanisms

- (Half-duplex) pipes
- FIFOs
- Message queues
- Semaphores
- Shared memory
- Sockets ← not today…

Pipes

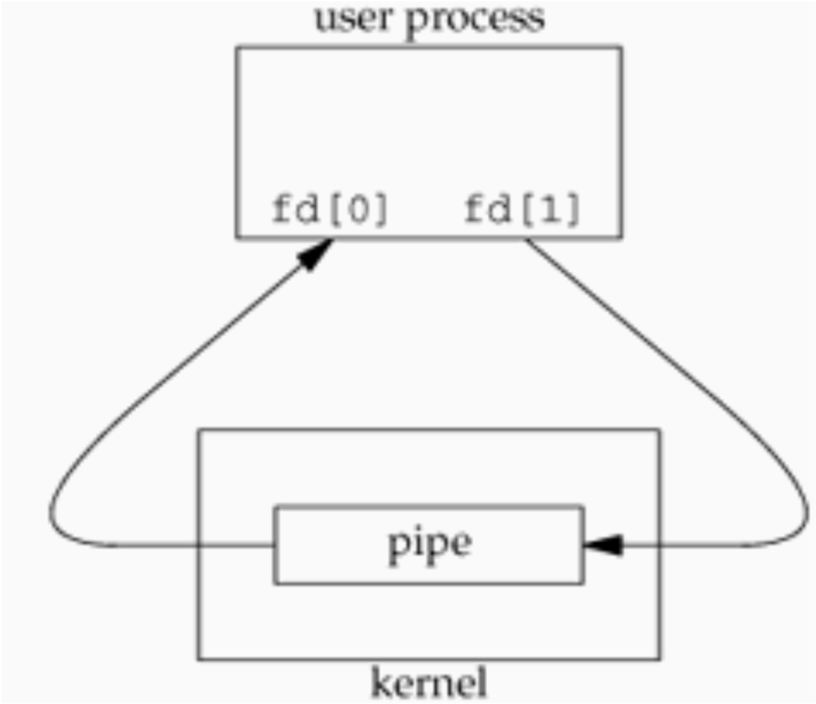
- The oldest form of UNIX System IPC
- Historically, they have been half duplex
  - Some modern system has full duplex pipe, but for program portability, it is not suggested to use full duplex pipe.
- Pipes can be used only between processes that have a common ancestor
  - Normally, a pipe is created by a process
  - The process then calls fork
  - The pipe is then used between the parent and the child

Creating a Pipe

- Synopsis

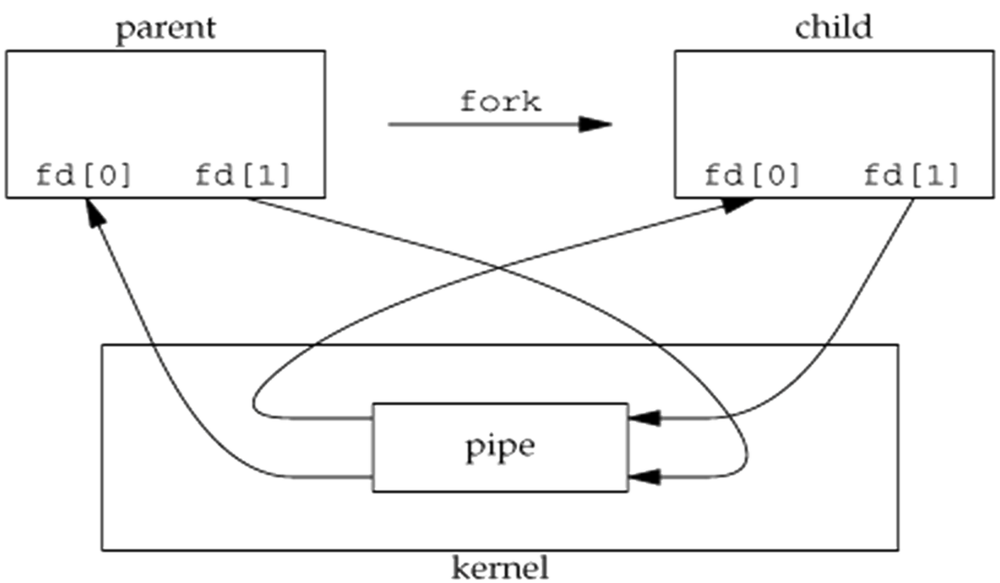
```
int pipe(int filedes[2]);
```

  - Returns: 0 if OK, -1 on error
- Two descriptors are created
  - filedes[0] is opened for reading
  - filedes[1] is opened for writing



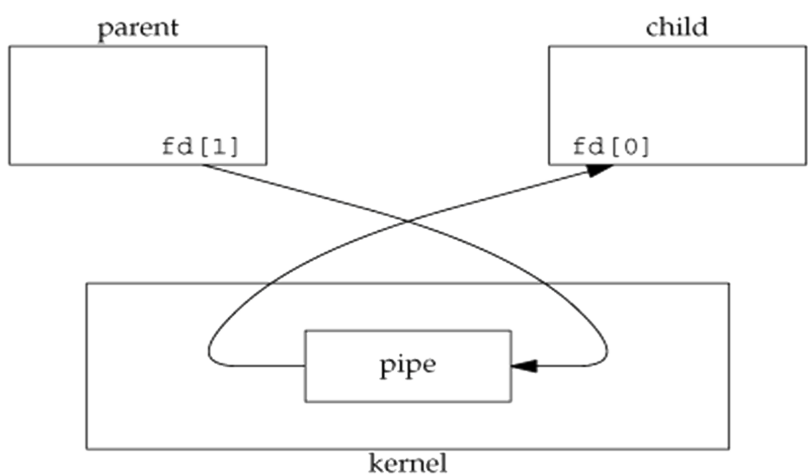
Sharing a Pipe

- A pipe in a single process is useless
- Normally, the process that calls pipe then calls fork
  - This creates an IPC channel from the parent to the child or vice versa



Sharing a Pipe (Cont'd)

- As the pipe is half duplex, the following actions may apply depending on the scenario
  - If the pipe is used for a child to send data to its parent
    - The parent closes fd[1] and the child closes fd[0]
  - If the pipe is used for a parent to send data to its child
    - The parent closes fd[0] and the child closes fd[1], see the figure



An Example of Creating a Pipe

```
int main(void) {
     int n;
     int 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);
}
```

Process Synchronization: Using a Pipe

- Recall: In Chapter 8
  - Race Conditions between the Parent and the Child

```
    int main(void) {
        pid_t   pid;
+       TELL_WAIT();
        if ((pid = fork()) < 0)     {
            err_sys("fork error");
        } else if (pid == 0) {
+           WAIT_PARENT();          /* parent goes first */
            charatatime("output from child\n");
        } else {
            charatatime("output from parent\n");
+           TELL_CHILD(pid);
        }
        exit(0);
    }
```

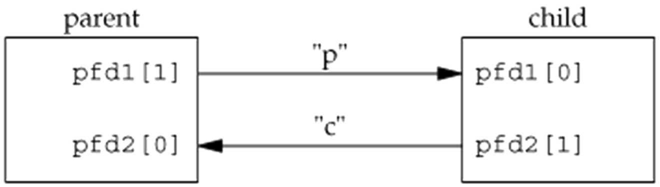
Process Synchronization: Using a Pipe (Cont'd)

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

void TELL_WAIT(void) {
     if (pipe(pfd1) < 0 || pipe(pfd2) < 0)
         err_sys("pipe error");
}

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");
}
```



Only part of the implementation…

popen and pclose Functions

- Execute a command and access its standard I/O
  - Read from its standard output, or
  - Write to its standard input
  - As we are using half-duplex pipe, we cannot read/ write at the same time
- Synopsis
  - FILE *popen(const char *cmdstring, const char *type);
  - Returns: file pointer if OK, NULL on error
  - int pclose(FILE *fp);
  - termination status of *cmdstring*, or -1 on error

popen and pclose Functions

- Operations
  - create a pipe (pipe)
  - fork a child (fork)
  - close the unused ends of the pipe (close)
  - configure the descriptor (dup2)
  - execute a shell to run the command (exec), and
  - wait for the command to terminate (wait)

- popen with a type of “r”

- popen with a type of “w”

Implementation of popen and pclose

- See Figure 15.12 of the textbook
- popen
  - Make sure that type is “r” or “w”
  - Create a buffer for popen children PIDs
  - Create a pipe and fork a child process
  - For the child:
    - If type is “r”, close fd[0], otherwise close fd[1]
    - execl("/bin/sh", "sh", "-c", cmdstring, (char *)0);
  - For the parent
    - If type is “r”, close fd[1], otherwise close fd[0]
    - If type is “r”, FILE *fp = fdopen(fd[0], type)
    - Otherwise, FILE *fp = fdopen(fd[1], type)
    - Save child PID (indexed by pipe fd) and return fp

Implementation of popen and pclose (Cont'd)

- pclose
  - Get descriptor number by fd = fileno(fp);
  - Retrieve the pid (indexed by pipe fd)
  - Reset the corresponding pid on the children's pid buffer to zero
  - fclose(fp)
  - waitpid(pid, &stat, 0)
  - return(stat)

A popen Example: Filter

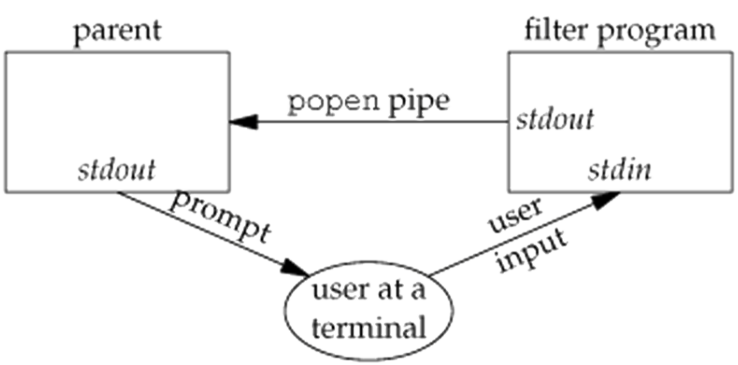
- A filter that converts uppercases into lowercases

```
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);
}
```

popen Example: Filters (Cont'd)

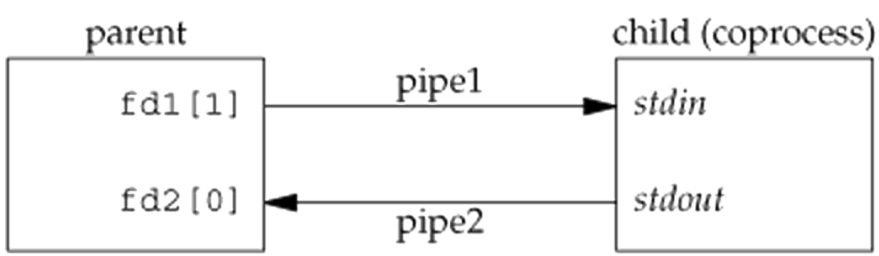
- A program that run the filter using popen, and show the filtered content

```
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);
}
```



Coprocess

- Definition of an UNIX system filter
  - A process that reads from standard input and writes to standard output
- Coprocess
  - An UNIX system filter becomes a coprocess if the filter's input and output are both associated with the same program
  - We need two pipe() calls to setup the communication channel between a program and its coprocess



Coprocess, an Example

- A filter that read from STDIN, adds two numbers, and write to STDOUT
  - Implemented using file I/O

```
int main(void) {
    int     n, int1, int2;
    char    line[MAXLINE];
    while ((n = read(STDIN_FILENO, line, MAXLINE)) > 0) {
    line[n] = 0  /* null terminated */
    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);
}
```

Coprocess, an Example (Cont'd)

The **sig_pipe** function just print a message and then exit(1);

```
int main(void) {
  int n, fd1[2], fd2[2];
  pid_t pid;
  char line[MAXLINE];
  if (signal(SIGPIPE, sig_pipe) == SIG_ERR)
    err_sys("signal error");
  if (pipe(fd1) < 0 || pipe(fd2) < 0)
    err_sys("pipe error");
  if ((pid = fork()) < 0) err_sys("fork error");
  else if (pid > 0) {      /* parent */
    close(fd1[0]);
    close(fd2[1]);
    while (fgets(line, MAXLINE, stdin) != NULL) {
      n = strlen(line);
      if (write(fd1[1], line, n) != n)
        err_sys("write error to pipe");
      if ((n = read(fd2[0], line, MAXLINE)) < 0)
        err_sys("read error from pipe");
      if (n == 0) {
        err_msg("child closed pipe");
        break;
      }
      line[n] = 0;         /* null terminate */
      if (fputs(line, stdout) == EOF)
        err_sys("fputs error");
    }
    if (ferror(stdin))
      err_sys("fgets error on stdin");
    exit(0);
  } else {                 /* child */
    close(fd1[1]);
    close(fd2[0]);
    if (fd1[0] != STDIN_FILENO) {
      if (dup2(fd1[0], STDIN_FILENO) != STDIN_FILENO)
        err_sys("dup2 error to stdin");
      close(fd1[0]);
    }
    if (fd2[1] != STDOUT_FILENO) {
      if (dup2(fd2[1], STDOUT_FILENO) !=
STDOUT_FILENO)
        err_sys("dup2 error to stdout");
      close(fd2[1]);
    }
    if (execl("./add2", "add2", (char *)0) < 0)
      err_sys("execl error");
  }
  return 0;
}
```

Coprocess and Standard I/O

- What happens if the coprocess is implemented using standard I/O?
  - The filter no longer works!
- It is because the I/O buffering mode
  - When standard input/output are not terminal devices, they are fully buffered
  - Solution: We need pseudo-terminals devices to emulate the line buffer or unbuffered channel (not discussed in this Chapter)

FIFOs

- First in, first out
- FIFOs are sometimes called named pipes
- Pipes can be only used between processes of a common ancestor
- With FIFOs, unrelated processes can exchange data
- Creating a FIFO, synopsis
  - int mkfifo(const char *pathname, mode_t mode);
  - Returns: 0 if OK, -1 on error
- Once we have used mkfifo to create a FIFO, we open it using open

Open an FIFO

- When we open a FIFO, the non-blocking flag (O_NONBLOCK) affects what happens
- In the normal case (O_NONBLOCK not specified)
  - An open for read-only blocks until another process opens the FIFO for writing
  - Similarly, an open for write-only blocks until some other process opens the FIFO for reading
- If O_NONBLOCK is specified
  - An open for read-only returns immediately
  - But an open for write-only returns -1 with errno set to ENXIO if no process has the FIFO open for reading

Share an FIFO

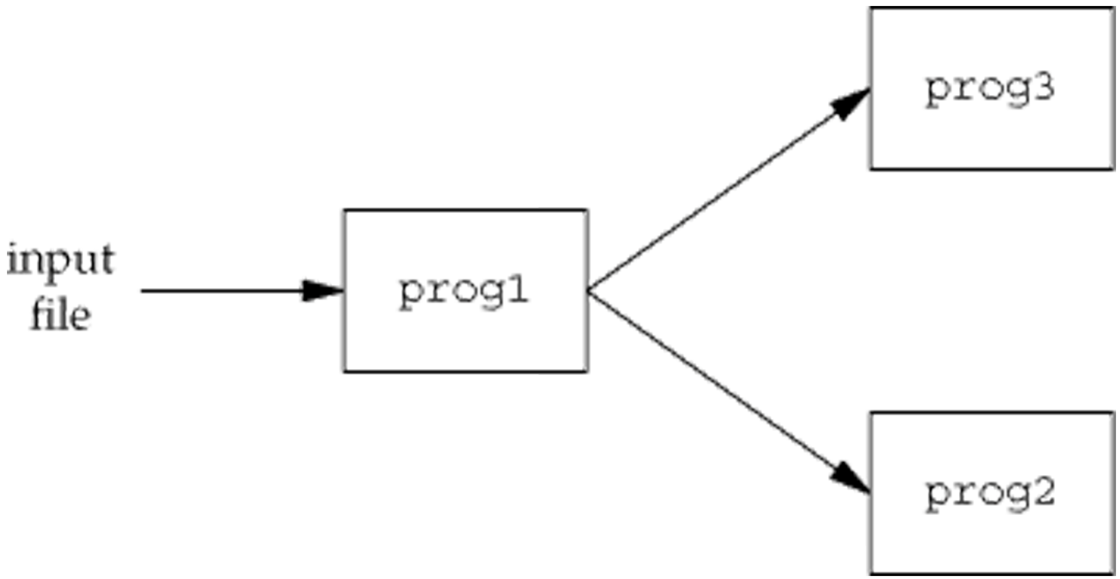
- It is common to have multiple writers for a given FIFO
- We have to worry about atomic writes if we don't want the writes from multiple processes to be interleaved

Applications of FIFOs

- Data passing
  - Pass data without creating intermediate temporary files
- Client-server communication
  - Used as rendezvous points in client-server applications

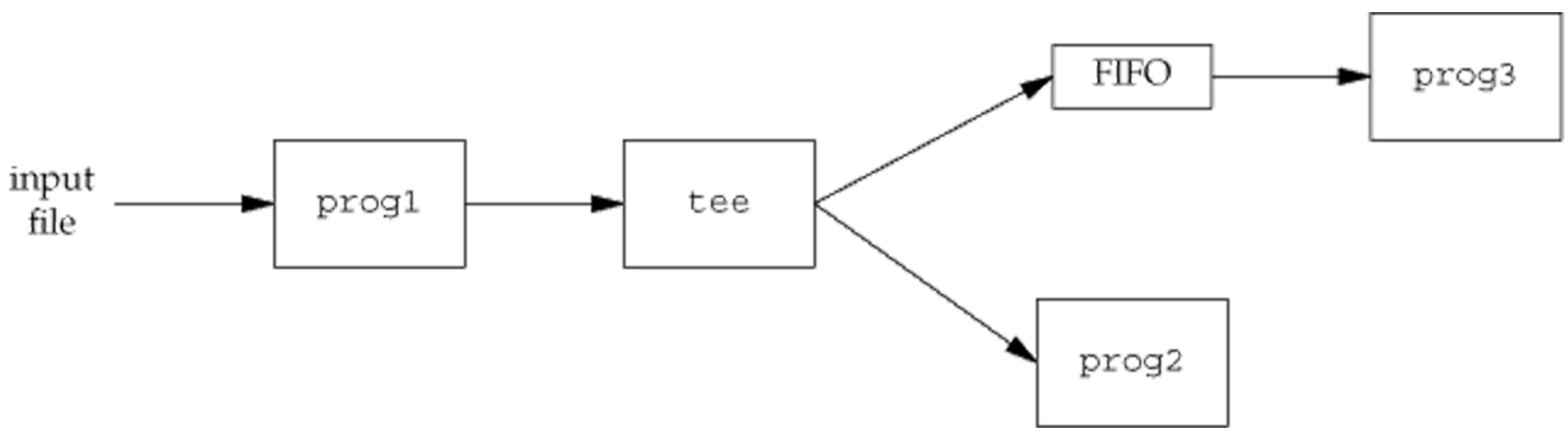
FIFO Applications – Data Passing

- Scenario
  - Process a filtered input stream twice



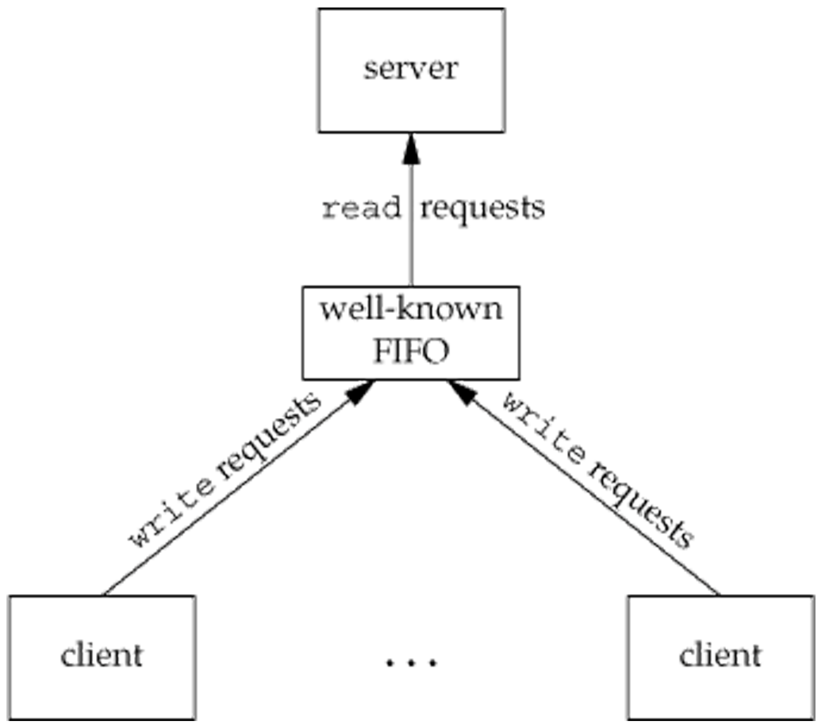
FIFO Applications – Data Passing (Cont'd)

- Solutions with FIFO
  - ```
    $ mkfifo fifo1
    $ prog3 < fifo1 &
    $ prog1 < infile | tee fifo1 | prog2
    ```


    ```

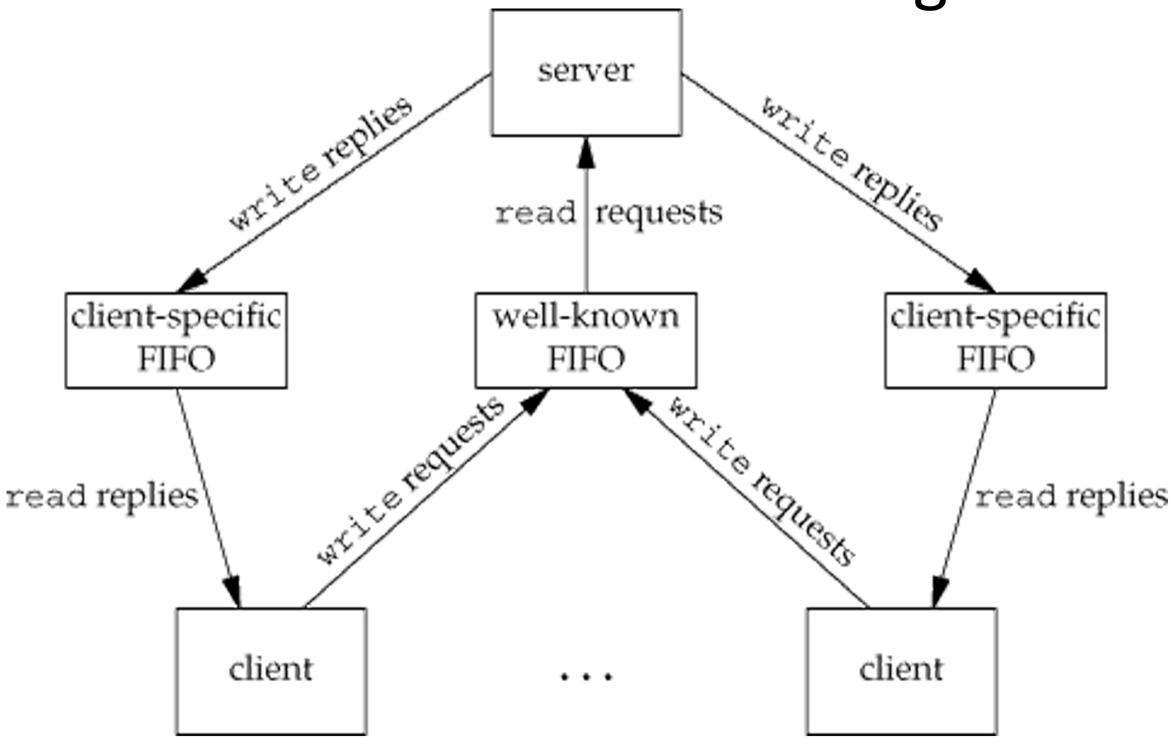
FIFO Applications – Client-Server Communication

- Scenario #1: One way communication
  - Clients send requests to a server



FIFO Applications – Client-Server Communication (Cont'd)

- Scenario #2: Two-way communications
  - Client-server communication using FIFOs



XSI (SysV) IPC

- XSI – X/Open System Interface
- Three types of XSI IPC
  - Message queue
  - Semaphore
  - Shared memory

- Common user commands
  - ipcs – list IPC objects
  - ipcrm – remove IPC objects

XSI (SysV) IPC (Cont'd)

- IPC identifiers
  - Each IPC structure in the kernel is referred to by a non-negative integer identifier
  - We need to know the identifier to access the IPC object
- However, the identifier is an internal name for an IPC object
  - We need a naming scheme to refer the same IPC object – the IPC keys
- IPC keys
  - Whenever an IPC structure is being created , a key must be specified
  - Keys are of data type key_t
  - Then, the identifier of the referred IPC object is returned

Sharing of IPC Objects

- A server can create an IPC object with a key of IPC_PRIVATE
  - The identifier of the created IPC object can be passed by storing in a file, or
  - Fork a child, which inherits the identifier directly
- A server and a client can agree on a key by defining the key in a common header
- A server and a client can agree on a pathname and a project ID
  - The key can be generated by the ftok function
  - key_t ftok(const char *path, int id);
  - path must be an existing file, and
  - id is a 8-bit non-zero number (you can not use more than 8 bits!)

XSI IPC – Advantages and Disadvantages *

- Advantages
  - Reliable
  - Supports flow control
  - Record based
  - Can be processed in other than first-in, first-out order
- Disadvantages
  - IPC data may left in the system even if no one refers to it
  - They are different from file system objects, i.e. no descriptors
  - Therefore, we need a different set of system calls to manipulate them

Message Queues

- A message queue is a linked list of messages stored within the kernel
- Each queue has a message queue identifier
- Creating or opening a message queue
  - int msgget(key_t key, int flag);
  - Returns: 0 if OK, -1 on error
  - Upon creating, the least significant 9 bits of *flag* define the permissions for the message queue
  - flag can be OR'ed with IPC_CREAT and/or IPC_EXCL

Message Queue – System Limitations

- The limitations may vary on different platforms
  - “ipcs -l” command on Linux
  - “ipcs -Q” on BSD and Mac OS X

```
$ ipcs -l

...

------ Messages Limits --------
max queues system wide = 32768
max size of message (bytes) = 8192
default max size of queue (bytes) = 16384
```

Controlling a Message Queue

- The internal data structure associated with a message queue

```
struct msqid_ds {
    struct ipc_perm msg_perm;     /* Ownership and permissions */
    time_t          msg_stime;    /* Time of last msgsnd(2) */
    time_t          msg_rtime;    /* Time of last msgrcv(2) */
    time_t          msg_ctime;    /* Time of last change */
    unsigned long   __msg_cbytes; /* Current number of bytes in queue (non-standard) */
    msgqnum_t       msg_qnum;     /* Current number of messages in queue */
    msglen_t        msg_qbytes;   /* Maximum number of bytes allowed in queue */
    pid_t           msg_lspid;    /* PID of last msgsnd(2) */
    pid_t           msg_lrpid;    /* PID of last msgrcv(2) */
};
```

Controlling a Message Queue (Cont’d)

- Synopsis
  - int msgctl(int msqid, int cmd, struct msqid_ds *buf);
  - Returns: 0 if OK, -1 on error
- The *cmd* can be
  - IPC_STAT: Retrieve the internal msqid_ds data structure
  - IPC_SET: Set the msqid_ds
    - msg_perm.uid, msg_perm.gid, msg_perm.mode, and msg_qbytes
    - Only superuser is able to increase msg_qbytes
  - IPC_RMID: Remove the queue (immediately)

Send a Message into Queue

- Synopsis
  - int msgsnd(int msqid, const void *ptr, size_t nbytes, int flag);
- The message, which is pointed to by *ptr*
  - It must be started with an long integer (the type of the message)
  - A *nbytes* message follows the long integer

```
struct msgbuf {
    long type;   /* message type, must be > 0 */
    char mtext[1];   /* message data */
};
```

  - The flag
    - IPC_NOWAIT: non-blocking access to the queue
    - If the queue is full and IPC_NOWAIT is specified
      - It returns a error with errno set to EAGAIN

Receive a Message from Queue

- Synopsis
  - ssize_t msgrcv(int msqid, void *ptr, size_t nbytes, long type, int flag);
  - Returns: size of data portion of message if OK, -1 on error
- The message type
  - If type == 0, the first message on the queue is returned
  - If type > 0, the first message on the queue whose message type *equals* type is returned
  - If 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

Receive a Message from Queue (Cont'd)

- The flags
  - IPC_NOWAIT: non-blocking access to the queue
  - MSG_EXCEPT
    - If type > 0, the first message on the queue whose message type *has a non-equal* type is returned
  - MSG_NOERROR
    - If the received message has a longer size than *nbytes*, it is *truncated* and then returned

Message Queue: Hello, World! Example

```
struct msgbuf {
  long mtype;       /* message type, must be > 0 */
  char mtext[0];    /* message data */
};

int main() {
  int qid = -1, rlen, wlen;
  char buf[1024];
  pid_t pid;
  struct msgbuf *msg = (struct msgbuf*) buf;
  //
  if((qid = msgget(IPC_PRIVATE, IPC_CREAT|IPC_EXCL|0660)) < 0)
    err_sys("msgget");
  if((pid = fork()) < 0)
    err_sys("fork");
```

Message Queue: Hello, World! Example (Cont'd)

```
  if(pid == 0) {  /* child */
    msg->mtype = 0;
    if((rlen = msgrcv(qid, msg, sizeof(buf)-sizeof(*msg), 0, 0)) < 0)
      err_sys("msgrcv");
    printf("[%ld] %s (%u bytes)\n", msg->mtype, msg->mtext, rlen);
  } else {         /* parent */
    msg->mtype = 1024;
    wlen = snprintf(msg->mtext, sizeof(buf)-sizeof(*msg),
        "%s", MESSAGE);
    if(msgsnd(qid, msg, wlen+1, 0) < 0)
      perror("msgsnd");
    else if(wait(&wlen) < 1)
      perror("wait");
    if(qid >= 0)
      if(msgctl(qid, IPC_RMID, NULL) < 0)
        err_sys("msgctl(RMID)");
  }
  return 0;
}
```

Semaphore (1/3)

- A semaphore is a shared counter
- It is used to provide access to a shared data object for multiple processes
- Procedures for a process to obtain a shared resource
  - Test the semaphore that controls the resource
  - If the value of the semaphore is positive, the process can use the resource
    - The process decrements the semaphore value by 1
  - If the value of the semaphore is 0
    - The process goes to sleep until the semaphore value is greater than 0

Semaphore (2/3)

- Features
  - A semaphore is a set of one or more semaphore values
    - It is not simply a single non-negative value
  - Semaphore creation (semget) and initialization (semctl) are independent
    - It may be a problem as we cannot *atomically* create a new semaphore set and initialize all the values in the set
  - All XSI IPC objects are not released automatically
    - They remain in existence even when no process is using them
    - We have to worry about a program's termination without releasing semaphores
    - This can be solved by the semaphore UNDO feature

Semaphore (3/3)

- Creating or opening a set of semaphore
  - int semget(key_t key, int nsems, int semflg);
  - Returns: semaphore ID if OK, -1 on error
  - Creates a new set of *nsems* semaphores
    - If opening an existing semaphores, this value can be 0
  - Upon creating, the least significant 9 bits of *semflg* define the permissions for the semaphore set
  - *semflg* can be OR'ed with IPC_CREAT and/or IPC_EXCL

Semaphore – System Limitations

- The limitations may vary on different platforms
  - "ipcs -l" command on Linux
  - "ipcs -S" on BSD and Mac OS X

```
$ ipcs -l

...

------ Semaphore Limits --------
max number of arrays = 128
max semaphores per array = 250
max semaphores system wide = 32000
max ops per semop call = 32
semaphore max value = 32767
```

Controlling Semaphores (1/3)

- The internal data structure associated with a semaphore set

```
struct semid_ds {
    struct ipc_perm sem_perm;   /* Ownership and permissions */
    time_t          sem_otime;  /* Last semop time */
    time_t          sem_ctime;  /* Last change time */
    unsigned short  sem_nsems;  /* No. of semaphores in set */
};
```

- Each member of the semaphore set has at least these attributes maintained by the kernel:
  - semval: semaphore value, always >= 0
  - sempid: pid for last operation
  - semncnt: # of processes waiting for the *semval* to increase
  - semzcnt: # of processes waiting for the *semval* to be zero

Controlling Semaphores (2/3)

- Synopsis
  - int semctl(int semid, int semnum, int cmd, /* union semun arg */);
  - Returns: it depends on commands
  - This function may be called with 3 or 4 arguments, depends on *cmd*
  - The 4th argument

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

Controlling Semaphores (3/3)

- Available *cmds*

cmds	Description
IPC_STAT	Retrieve the internal semid_ds data structure and stores in *arg.buf*
IPC_SET	Set the internal semid_ds data structure by *arg.buf* ▪ *sem_perm.uid*, *sem_perm.gid*, and *sem_perm.mode*
IPC_RMID	Remove the semaphore (immediately)
GETVAL	Return the value of *semnum*-th member
SETVAL	Set the value of *semnum*-th member by arg.val
GETPID	Return the value of *sempid* for the *semnum*-th member
GETNCNT	Return the value of *semncnt* for the *semnum*-th member
GETZCNT	Return the value of *semzcnt* for the *semnum*-th member
GETALL	Retrieve all semaphore values, returned by *arg.array*
SETALL	Set all semaphore values by *arg.array*

Semaphore Operations

- Synopsis
  - int semop(int semid, struct sembuf semoparray[], size_t nops);
  - Returns: 0 if OK, -1 on error
  - The *semoparray* argument is a pointer to an array of semaphore operations
  - Please see the next slide for the details of operations

```
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 */
};
```

Semaphore Operations – Return Resources

- *sem_op* is positive: *sem_op* is added to the semaphore's value
- If SEM_UNDO is specified, *sem_op* is *subtracted* from the semaphore's *adjustment value* for this process

Semaphore Operations – Obtain Resources

- *sem_op* is negative
- If resources are available (|*sem_op*| >= *sem_val*)
  - |*sem_op*| is substracted from the semaphore's value
  - If SEM_UNDO is specified, |*sem_op*| is *added* to the semaphore's *adjustment value* for this process
- If resources are not available (|*sem_op*| < *sem_val*)
  - If IPC_NOWAIT is specified, *semop* returns an error of EAGAIN
  - If IPC_NOWAIT is not specified
    - The *semncnt* value for this semaphore is increased
    - The process is suspended until …
      - The semaphore's value becomes greater than or equal to the |*sem_op*|, the *semncnt* should be increased
      - The semaphore is removed from the system: *semop* returns an error of EIDRM
      - It is interrupted by a signal: *semop* returns an error of EINTR

Semaphore Operations – Wait until Zero

- *sem_op* is zero
- The calling process wants to wait until the semaphore's value becomes 0
- If the semaphore's value is currently 0, the function returns immediately
- Otherwise,
  - If IPC_NOWAIT is specified, return is made with an error of EAGAIN
  - If IPC_NOWAIT is not specified
    - The *semzcnt* value for this semaphore is incremented
    - The calling process is suspended until …
      - The semaphore's value becomes 0 , the *semzcnt* should be increased
      - The semaphore is removed from the system: *semop* returns an error of EIDRM
      - It is interrupted by a signal: *semop* returns an error of EINTR

Semaphore Adjustment on Terminating a Process

- We have mentioned the problem
  - A program's termination without releasing semaphores may block future access to the resource
- The problem can be solved by the UNDO feature
  - When we specify the SEM_UNDO flag for a semaphore operation
  - The kernel remembers how many resources we allocated from that particular semaphore
  - When the process terminates, the kernel checks whether the process has any *outstanding semaphore adjustments*, i.e., the value is > 0
  - If so, applies the adjustment to the corresponding semaphore
    - *semval* is increased by the adjustments

Shared Memory

- Allows two or more processes to share a given region of memory
- This is the fastest form of IPC
  - The data does not need to be copied between the client and the server, but
  - We have to synchronize access to a given region among multiple processes
    - If the server is placing data into a shared memory region, the client should not try to access the data
  - Synchronizing can be done by semaphores

Shared Memory (Cont'd)

- Creating or opening a shared memory
- Synopsis
  - int shmget(key_t key, size_t size, int flag);
  - Returns: shared memory ID if OK, -1 on error
  - Upon creating, the least significant 9 bits of *semflg* define the permissions for the shared memory
  - *flag* can be OR'ed with IPC_CREAT and/or IPC_EXCL
  - The actual size of the created shared memory is round up to multiples of the PAGE_SIZE (4096 bytes)
  - When a shared memory is created, it's content initialized to all zero

Shared Memory – System Limitations

- The limitations may vary on different platforms
  - "ipcs -l" command on Linux
  - "ipcs -M" on BSD and Mac OS X

```
$ ipcs -l

...

------ Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 18014398509465599
max total shared memory (kbytes) = 18446744073642442748
min seg size (bytes) = 1
```

Controlling Shared Memory

- The internal data structure associated with a shared memory

```
struct shmid_ds {
    struct ipc_perm shm_perm;     /* Ownership and permissions */
    size_t          shm_segsz;    /* Size of segment (bytes) */
    time_t          shm_atime;    /* Last attach time */
    time_t          shm_dtime;    /* Last detach time */
    time_t          shm_ctime;    /* Last change time */
    pid_t           shm_cpid;     /* PID of creator */
    pid_t           shm_lpid;     /* PID of last shmat(2)/shmdt(2) */
    shmatt_t        shm_nattch;   /* No. of current attaches */
    ...
};
```

Controlling Shared Memory (Cont’d)

- Synopsis
  - int shmctl(int shmid, int cmd, struct shmid_ds *buf);
  - Returns: 0 if OK, -1 on error
  - Commands
    - IPC_STAT: Retrieve the internal shmid_ds data structure
    - IPC_SET: Set the internal shmid_ds data structure
      - *shm_perm.uid*, *shm_perm.gid*, and *shm_perm.mode*
    - IPC_RMID: Remove the shared memory, but *it is actually removed until the last process using the segment terminates or detaches it*
    - SHM_LOCK: Make the shared memory not swappable
    - SHM_UNLOCK: Make the shared memory swappable
      - The last two commands can be only used by superuser

Attach a Shared Memory

- Synopsis
  - void *shmat(int shmid, const void *addr, int flag);
  - Returns: pointer to shared memory segment if OK, -1 on error
  - The *addr* argument
    - If *addr* is NULL, the segment is attached at the first available address selected by the kernel (**RECOMMENDED*)
    - If *addr* is not NULL and SHM_RND is not specified, the segment is attached at the address given by *addr*
    - If *addr* is not NULL and SHM_RND is specified, the segment is attached at the address given by (*addr - (addr modulus SHMLBA*))
      - Round down to the multiples of SHMLBA
  - The *flag* argument
    - If the SHM_RDONLY bit is specified in *flag*, the segment is attached read-only

Detach a Shared Memory

- Synopsis
  - int shmdt(void *addr);
  - Returns: 0 if OK, -1 on error

Message Queue versus Pipe versus UNIX Socket

- Difference is not clear
- Message queues share disadvantages of XSI IPC, see textbook 15.6.4

Operation	User	System	Clock
message queue	0.58	4.16	5.09
full-duplex pipe	0.61	4.30	5.24
UNIX domain socket	0.59	5.58	7.49

Figure 15.27 Timing comparison of IPC alternatives on Solaris

Semaphores versus Record Locking versus Mutex

- Observation 1: Semaphore may be overcomplicated
- Observation 2: Record locking may be preferred because: (i) easier to handle the case of process termination and (ii) process-shared mutex may not be supported

Operation	User	System	Clock
semaphores with undo	0.50	6.08	7.55
advisory record locking	0.51	9.06	4.38
mutex in shared memory	0.21	0.40	0.25

Figure 15.29 Timing comparison of locking alternatives on Linux


Homework?
NOW
LATER