Topics: IPC and Unix Special Files
(USP Chapters 6 and 7)

CS 3733 Operating Systems

Instructor: Dr. Turgay Korkmaz

Department Computer Science

The University of Texas at San Antonio

Office: NPB 3.330
Phone: (210) 458-7346
Fax: (210) 458-4437
e-mail: korkmaz@cs.utsa.edu
web: www.cs.utsa.edu/~korkmaz

These slides are prepared based on the materials provided by Drs. Robbins, Zhu, and Liu. Thanks to all.

Outline

- Inter-Process communication (IPC)
- Pipe and its operations
  - Allow **related** processes (parent-child) to communicate
- FIFOs: named pipes
  - Allow **un-related** processes to communicate
- Ring of communicating processes
  - Steps for Ring Creation with Pipes
  - A Ring of n Processes
- Other issues in token ring
  - Which process to write/read: token management

EXAMPLE

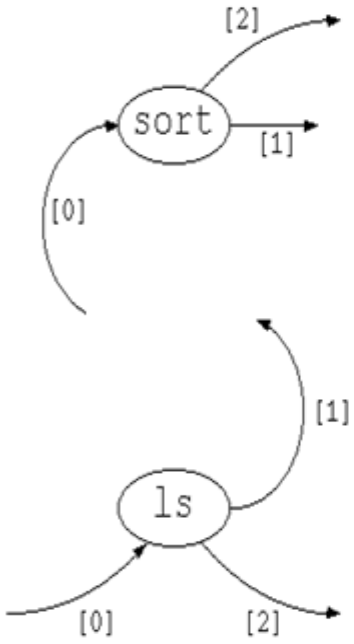
How Can Processes Communicate?

> ls -l > tmp

> sort -nr +4 < tmp

> rm tmp

vs. > ls -l | sort -nr +4



sort

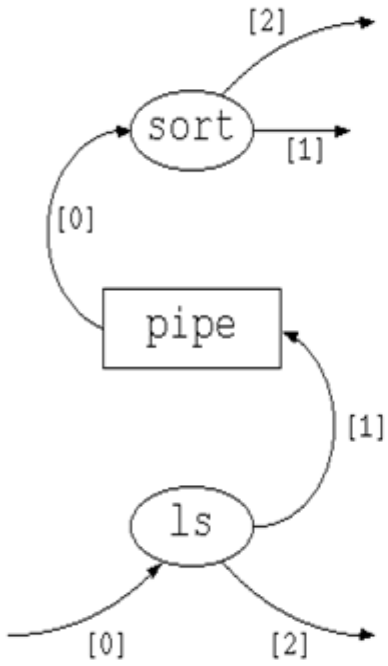
file descriptor table

[0]	fd-tmp
[1]	*standard output*
[2]	*standard error*

ls

file descriptor table

[0]	*standard input*
[1]	fd-tmp
[2]	*standard error*



`sort` (parent)

file descriptor table

[0]	`pipe` *read*
[1]	*standard output*
[2]	*standard error*

`ls` (child)

file descriptor table

[0]	*standard input*
[1]	`pipe` *write*
[2]	*standard error*

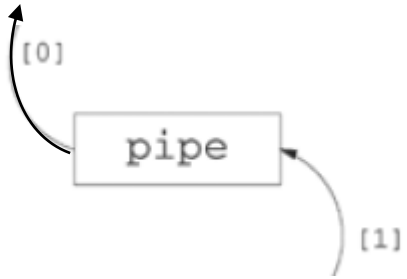
Interprocess Communication (IPC)

- IPC is a set of data structures and methods for allowing **cooperating** processes to exchange data

- Why do processes cooperate?
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience

> ls -l | sort -nr +4

Pipe: Communication Buffer



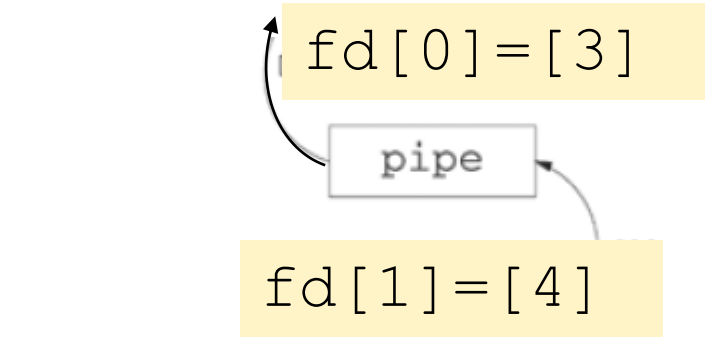
- Create a pipe: system call pipe()

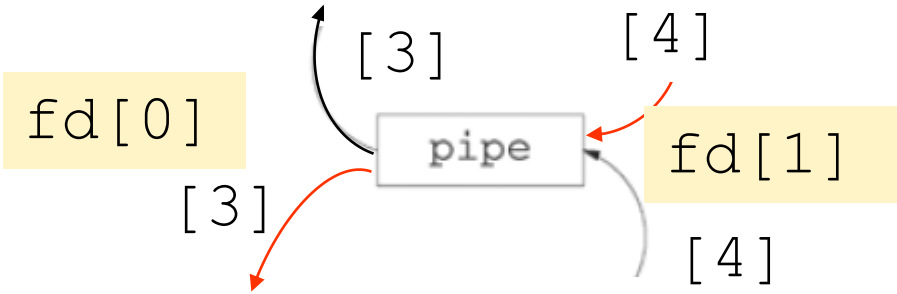
```
#include <unistd.h>
int pipe(int fildes[2]);
```

  - Create a ***unidirectional*** communication buffer with two file descriptors: **fildes[0]** for **read** and **fildes[1]** for **write**
  - Data write and read on a **first-in-first-out** base
  - **No external or permanent name,** and can only be accessed through two file descriptors
  - The pipe can only be used by the process that created it and its **descendants (i.e., child & grand-child processes)**

Program 6.1, parentwritepipe.c, page 185

```
int main(void) {
   char bufin[BUFSIZE] = "empty";
   char bufout[] = "hello";
   int bytesin;
   pid_t childpid;
   int fd[2];

   if (pipe(fd) == -1) {
      perror("Failed to create the pipe");
      return 1;
   }
   bytesin = strlen(bufin);
   childpid = fork();
   if (childpid == -1) {
      perror("Failed to fork");
      return 1;
   }
   if (childpid)                                   /* parent code */
      write(fd[1], bufout, strlen(bufout)+1);
   else                                             /* child code */
      bytesin = read(fd[0], bufin, BUFSIZE);
   fprintf(stderr, "[%ld]:my bufin is {%.*s}, my bufout is {%s}\n",
           (long)getpid(), bytesin, bufin, bufout);
   return 0;
}
```



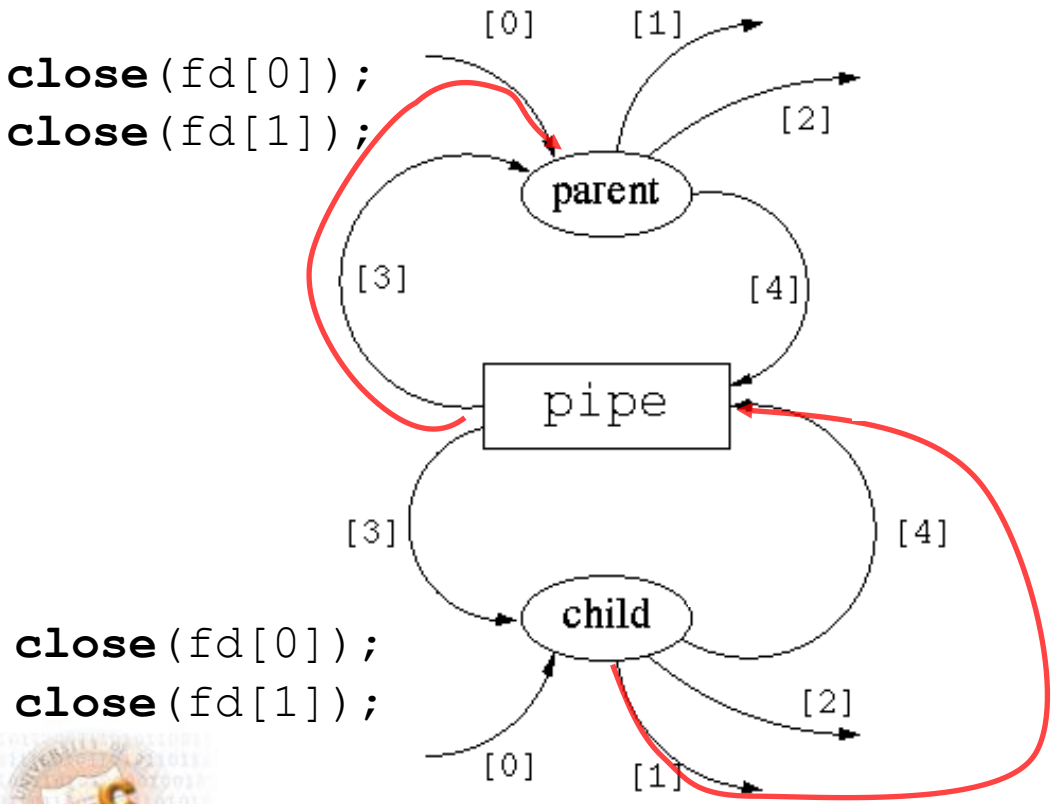


Which process write? Which process read?

Parent/Child File Descriptor Tables After `pipe()` and `fork()`

what should we do if we want
parent to read from STDIN_FILENO
what child writes to STDOUT_FILENO?

dup2`(fd[0],STDIN_FILENO);`



close`(fd[0]);`
close`(fd[1]);`

close`(fd[0]);`
close`(fd[1]);`

dup2`(fd[1],STDOUT_FILENO);`

parent

file descriptor table

[0]	*standard input*
[1]	*standard output*
[2]	*standard error*
[3]	pipe *read*
[4]	pipe *write*

child

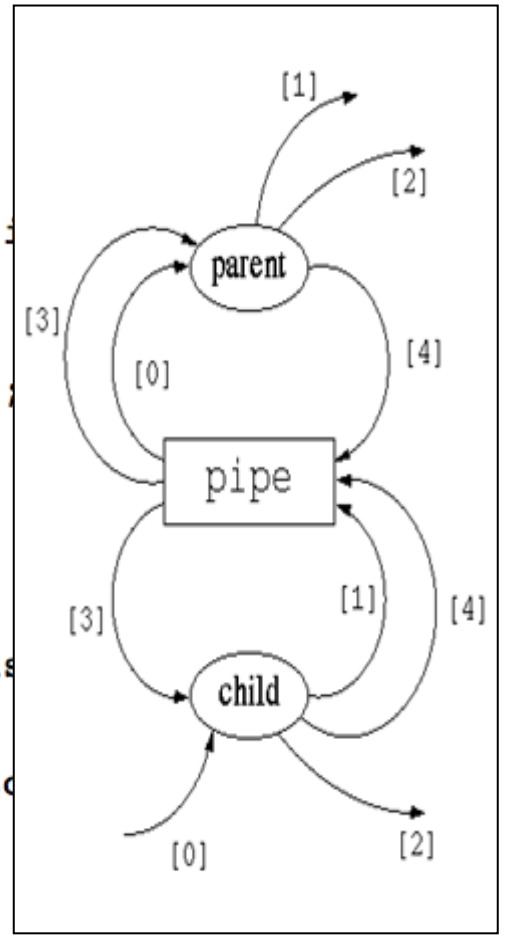
file descriptor table

[0]	*standard input*
[1]	*standard output*
[2]	*standard error*
[3]	pipe *read*
[4]	pipe *write*

Pipes and Simple Redirection

Example 6.5, page 188:
ls -l | sort -n +4

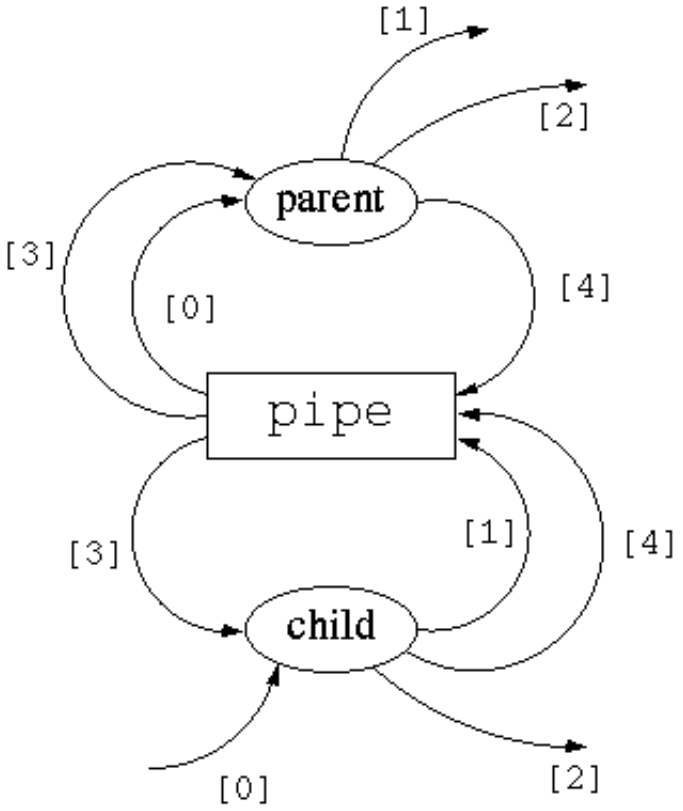
```
int main(void) {
   pid_t childpid;
   int fd[2];

   if ((pipe(fd) == -1) || ((childpid = fork()) == -1)) {
      perror("Failed to setup pipeline");
      return 1;
   }

   if (childpid == 0) {                                   /* ls i
      if (dup2(fd[1], STDOUT_FILENO) == -1)
         perror("Failed to redirect stdout of ls");
      else if ((close(fd[0]) == -1) || (close(fd[1]) == -1))
         perror("Failed to close extra pipe descriptors on ls");
      else {
         execl("/bin/ls", "ls", "-l", NULL);
         perror("Failed to exec ls");
      }
      return 1;
   }
   if (dup2(fd[0], STDIN_FILENO) == -1)                     /* sort is
      perror("Failed to redirect stdin of sort");
   else if ((close(fd[0]) == -1) || (close(fd[1]) == -1))
      perror("Failed to close extra pipe file descriptors on so
   else {
      execl("/bin/sort", "sort", "-n", "+4", NULL);
      perror("Failed to exec sort");
   }
   return 1;
}
```



After both dup2 functions

Which file descriptors are unnecessary?

Close them!



parent: sort -n +4

file descriptor table

[0]	pipe *read*
[1]	*standard output*
[2]	*standard error*
[3]	pipe *read*
[4]	pipe *write*

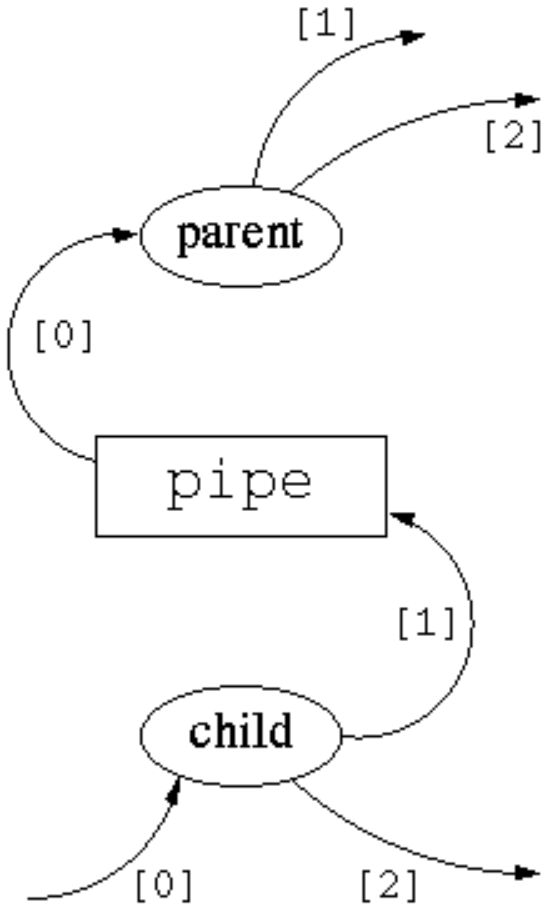
child: ls -l

file descriptor table

[0]	*standard input*
[1]	pipe *write*
[2]	*standard error*
[3]	pipe *read*
[4]	pipe *write*

)

After Close Unnecessary File Descriptors



parent :sort -n +4

file descriptor table

[0]	pipe *read*
[1]	*standard output*
[2]	*standard error*

child: ls -l

file descriptor table

[0]	*standard input*
[1]	pipe *write*
[2]	*standard error*

Pipe: Read and Write Semantic

- Read
  - Not necessarily **atomic**: may read less bytes
  - **Blocking**: if no data, but write file descriptor still opens
  - **If empty and all file descriptors for the write end are closed → read sees end-of-file and returns 0**

Close unnecessary file descriptors after dup2!
Otherwise, other program never gets end-of-file!

- Write
  - **Atomic** for at most PIPE_BUF bytes (512, 4K, or 64K)
  - **Blocking**: if buffer is full, and read file descriptors open
  - When all file descriptors referring to the read end of a pipe are closed → cause a **SIGPIPE** signal for the calling process

Pros/Cons of Pipe

- Pros
  - simple
  - flexible
  - efficient communication
- Cons:
  - it impossible for two arbitrary processes to share the same pipe, unless the pipe was created by a common ancestor process.

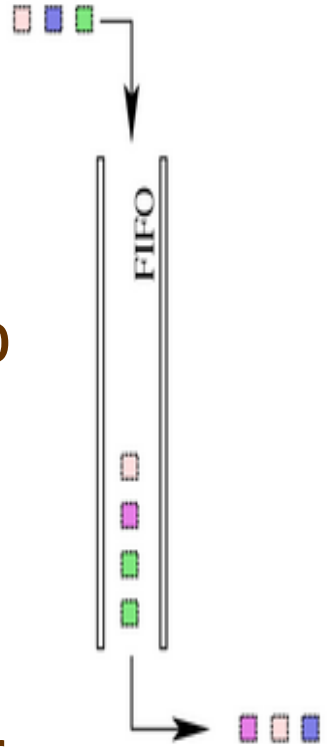
Outline

- Inter-Process communication (IPC)
- Pipe and its operations
  - Allow related processes to communicate
- FIFOs: named pipes
  - Allow un-related processes to communicate
- Ring of communicating processes
  - Steps for Ring Creation with Pipes
  - A Ring of n Processes
- Other issues in token ring
  - Which process to write/read: token management

EXAMPLE

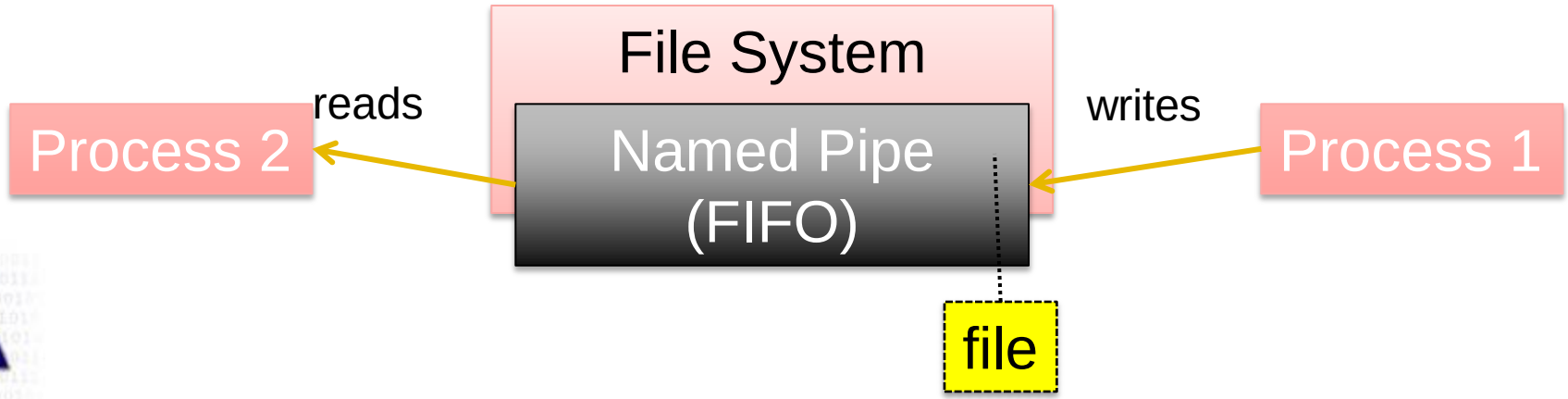
Named Pipes (FIFO)

- Named Pipes are more powerful than ordinary pipes *[also known as FIFOs (first-in, first-out)]*
- Named pipes allow two unrelated processes to communicate with each other.
  - No parent-child/sibling relationship is necessary between the communicating processes
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems



Named Pipes Cont.

- How does it work?
  - Uses an access point (A file on the file system)
  - Two unrelated processes opens this file to communicate
  - One process writes and the other reads, thus it is a half-duplex communication



FIFOs: Named Pipes

- Create a FIFO

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

A named pipe is a special instance of a file that has no contents on the filesystem.

- `path` specific the name of the pipe, which appears in the same directory as ordinary file
- Can be read/write by others depending on the permissions (mode)
- Remove a FIFO: same as remove a file
  - `rm or unlink`

Parent/Child Processes with a FIFO

```
int dofifochild(const char *fifoname, const char *idstring);
int dofifoparent(const char *fifoname);

int main (int argc, char *argv[]) {
   pid_t childpid;

   if (argc != 2) {                                  /* command line has pipe name */
      fprintf(stderr, "Usage: %s pipename\n", argv[0]);
      return 1;
   }
   if (mkfifo(argv[1], FIFO_PERM) == -1) {           /* create a named pipe */
      if (errno != EEXIST) {
         fprintf(stderr, "[%ld]:failed to create named pipe %s: %s\n",
               (long)getpid(), argv[1], strerror(errno));
         return 1;
      }
   }
   if ((childpid = fork()) == -1){
      perror("Failed to fork");
      return 1;
   }
   if (childpid == 0)                                /* The child writes */
      return dofifochild(argv[1], "this was written by the child");
   else
      return dofifoparent(argv[1]);
}
```

Program 6.4 on page 194 [USP]

Only the name of the pipe is passed to the parent and child routines.

Parent/Child Processes with a FIFO (cont.)

Program 6.5 on page 195 [USP]

```
int dofifochild(const char *fifoname, const char *idstring) {
   char buf[BUFSIZE];
   int fd;
   int rval;
   ssize_t strsize;

   fprintf(stderr, "[%ld]:(child) about to open FIFO %s...\n",
           (long)getpid(), fifoname);
   while (((fd = open(fifoname, O_WRONLY)) == -1) && (errno == EINTR)) ;
   if (fd == -1) {
      fprintf(stderr, "[%ld]:failed to open named pipe %s for write: %s\n",
              (long)getpid(), fifoname, strerror(errno));
      return 1;
   }
   rval = snprintf(buf, BUFSIZE, "[%ld]:%s\n", (long)getpid(), idstring);
   if (rval < 0) {
      fprintf(stderr, "[%ld]:failed to make the string:\n", (long)getpid());
      return 1;
   }
   strsize = strlen(buf) + 1;
   fprintf(stderr, "[%ld]:about to write...\n", (long)getpid());
   rval = r_write(fd, buf, strsize);
   if (rval != strsize) {
      fprintf(stderr, "[%ld]:failed to write to pipe: %s\n",
              (long)getpid(), strerror(errno));
      return 1;
   }
   fprintf(stderr, "[%ld]:finishing...\n", (long)getpid());
   return 0;
}
```

Parent/Child Processes with a FIFO (cont.)

Program 6.6 on page 196 [USP]

```
int dofifoparent(const char *fifoname) {
   char buf[BUFSIZE];
   int fd;
   int rval;

   fprintf(stderr, "[%ld]:(parent) about to open FIFO %s...\n",
                        (long)getpid(), fifoname);
   while (((fd = open(fifoname, FIFO_MODES)) == -1) && (errno == EINTR))  ;
   if (fd == -1) {
      fprintf(stderr, "[%ld]:failed to open named pipe %s for read: %s\n",
             (long)getpid(), fifoname, strerror(errno));
      return 1;
   }
   fprintf(stderr, "[%ld]:about to read...\n", (long)getpid());
   rval = r_read(fd, buf, BUFSIZE);
   if (rval == -1) {
      fprintf(stderr, "[%ld]:failed to read from pipe: %s\n",
             (long)getpid(), strerror(errno));
      return 1;
   }
   fprintf(stderr, "[%ld]:read %.*s\n", (long)getpid(), rval, buf);
   return 0;
}
```

Using FIFO in Client-Server Model

- Client-Server Communication
  - Client processes request service from a server.
  - A server process waits for requests from clients.
- A process can act as both a client and a server

- When processes are exchanging data via the FIFO, the kernel passes all data internally without writing it to the filesystem. Thus, it is much faster than sockets, and not using network resources.

Server: Receive and Output

Program 6.7 on page 197 [USP]

```
#define FIFOARG 1
#define FIFO_PERMS (S_IRWXU | S_IWGRP| S_IWOTH)

int main (int argc, char *argv[]) {
   int requestfd;

   if (argc != 2) {    /* name of server fifo is passed on the command line */
      fprintf(stderr, "Usage: %s fifoname > logfile\n", argv[0]);
      return 1;
   }
                          /* create a named pipe to handle incoming requests */
   if ((mkfifo(argv[FIFOARG], FIFO_PERMS) == -1) && (errno != EEXIST)) {
      perror("Server failed to create a FIFO");
      return 1;
   }
                       /* open a read/write communication endpoint to the pipe */
   if ((requestfd = open(argv[FIFOARG], O_RDWR)) == -1) {
      perror("Server failed to open its FIFO");
      return 1;
   }
   copyfile(requestfd, STDOUT_FILENO);
   return 1;
}
```

Server process that creates a pipe (if necessary) and opens the FIFO. Whatever it receives from the FIFO it outputs to standard output.

Client: Write to a Named FIFO from Server

Program 6.8 on page 198 [USP]

```
int main (int argc, char *argv[]) {
   time_t curtime;
   int len;
   char requestbuf[PIPE_BUF];
   int requestfd;

   if (argc != 2) {  /* name of server fifo is passed on the command line */
      fprintf(stderr, "Usage: %s fifoname\n", argv[0]);
      return 1;
   }

   if ((requestfd = open(argv[FIFOARG], O_WRONLY)) == -1) {
       perror("Client failed to open log fifo for writing");
       return 1;
   }

   curtime = time(NULL);
   snprintf(requestbuf, PIPE_BUF, "%d: %s", (int)getpid(), ctime(&curtime));
   len = strlen(requestbuf);
   if (r_write(requestfd, requestbuf, len) != len) {
      perror("Client failed to write");
      return 1;
   }
   r_close(requestfd);
   return 0;
}
```

Can server and client be on two different machines?

client process that opens the named pipe for writing and writes the string to the pipe.

Benefits of Named Pipes

- Named pipes are very simple to use.
- Unrelated process on the same system can use it to communicate
- Named pipes have permissions (read and write) associated with them, unlike anonymous pipes.
- mkfifo is a thread-safe function.
- No synchronization mechanism is needed when using named pipes.
- write (using write function call) to a named pipe is guaranteed to be atomic. It is atomic even if the named pipe is opened in non-blocking mode.

Limitations of Named Pipes

- Named pipes can only be used for communication among processes on the same host machine.
- Careful programming is required for the client and server, in order to avoid deadlocks.
- Named pipe data is a **byte stream**, and no record identification exists.
- Named pipes can be created only in the local file system of the host, (i.e. cannot create a named pipe on the NFS file system.)

Outline

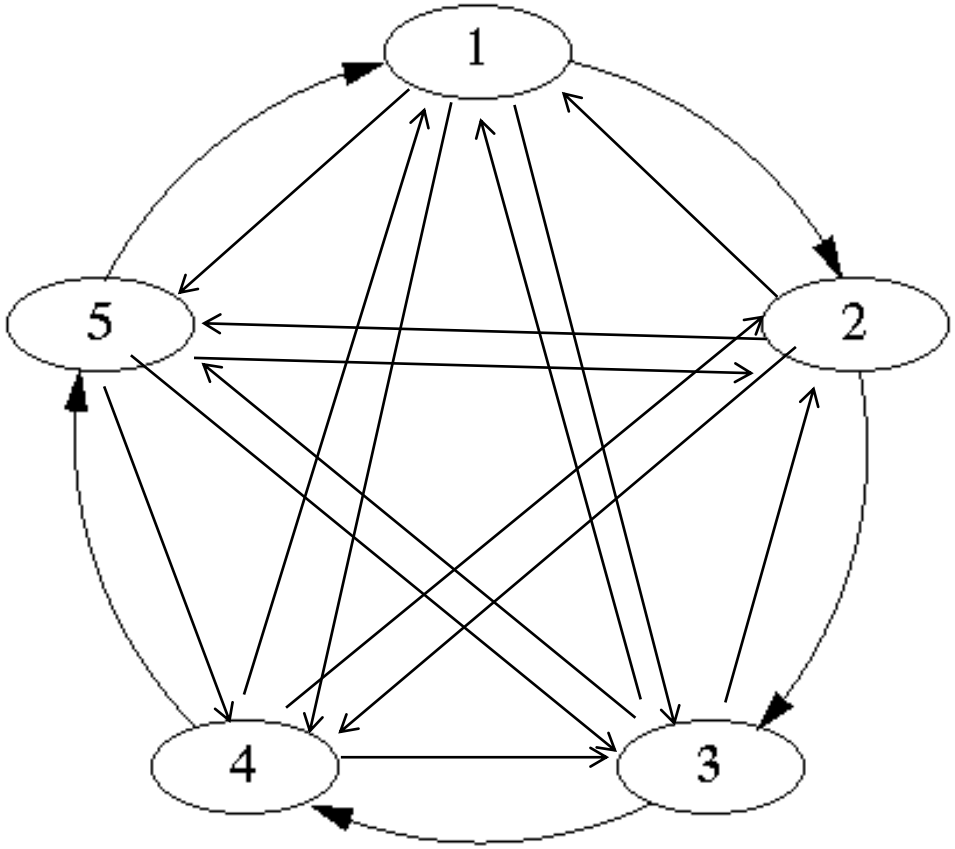
- Inter-Process communication (IPC)
- Pipe and its operations
- FIFOs: named pipes
  - Allow un-related processes to communicate
- Ring of communicating processes
  - Steps for Ring Creation with Pipes
  - A Ring of n Processes
- Other issues in token ring
  - Which process to write/read: token management

EXAMPLE

How do multiple processes communicate?

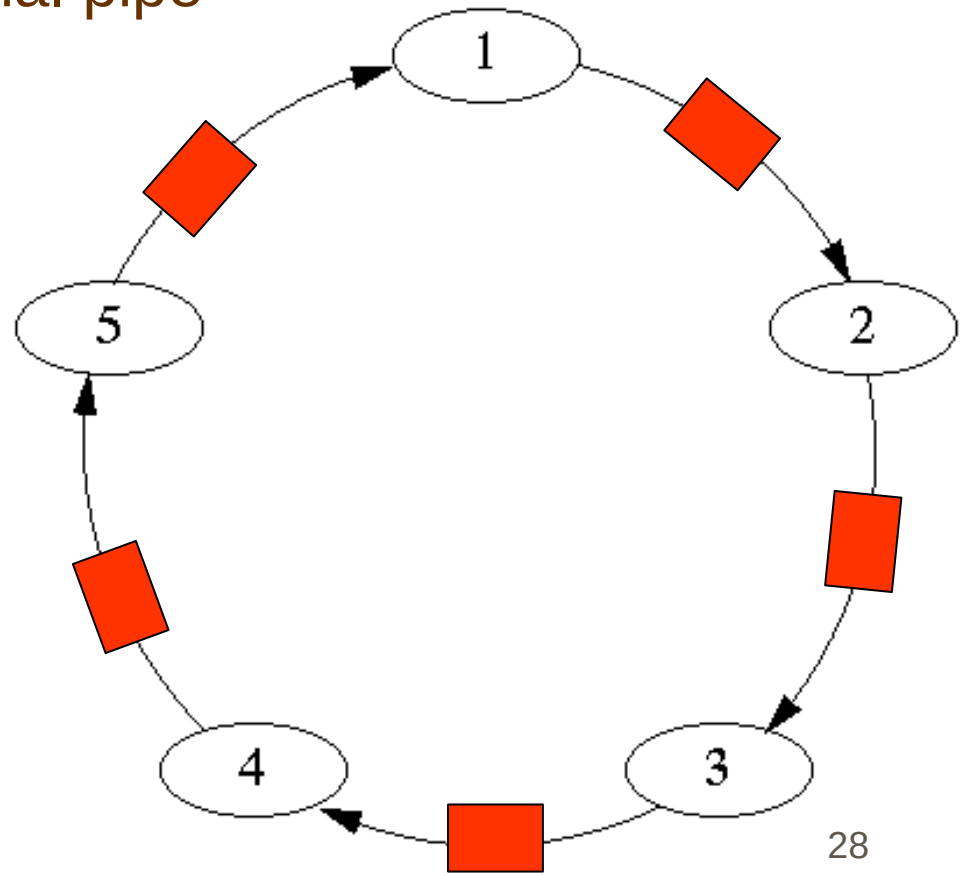
- Nodes represent processes
- Links represent a unidirectional pipe

N^2 problem!



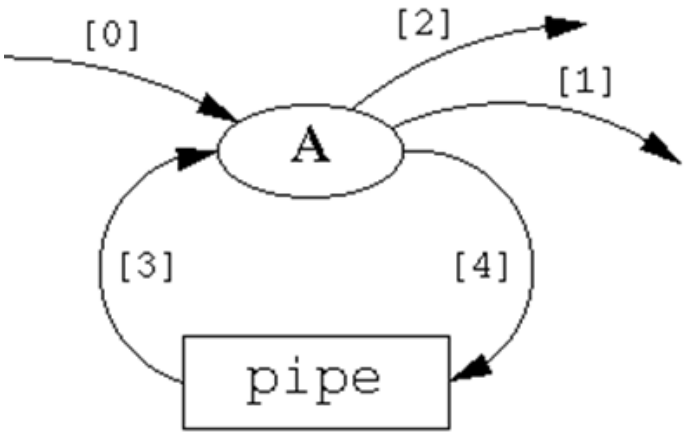
An Example of Unidirectional Ring

- A ring of 5 nodes
  - Nodes represent processes
  - Links represent a unidirectional pipe
- How do the processes communicate ?
  - Message with target PID
  - Receive targeted message
  - Forward other messages

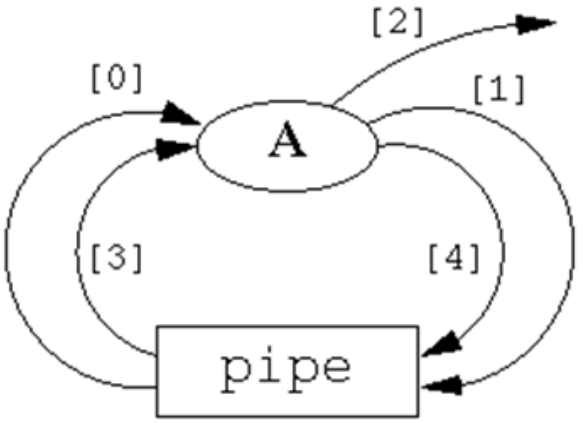


Ring with A Single Process

- Code section for a single process ring without error check

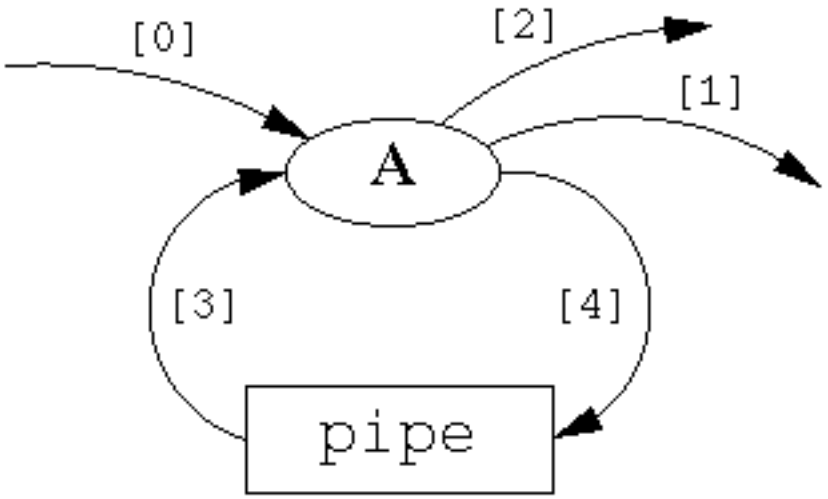



```
int fd[2];

pipe(fd);

dup2(fd[0], STDIN_FILENO);
dup2(fd[1], STDOUT_FILENO);

        close(fd[0]);
        close(fd[1]);
```



Ring with A Single Process (cont.)

- Status after **`pipe(fd)`**



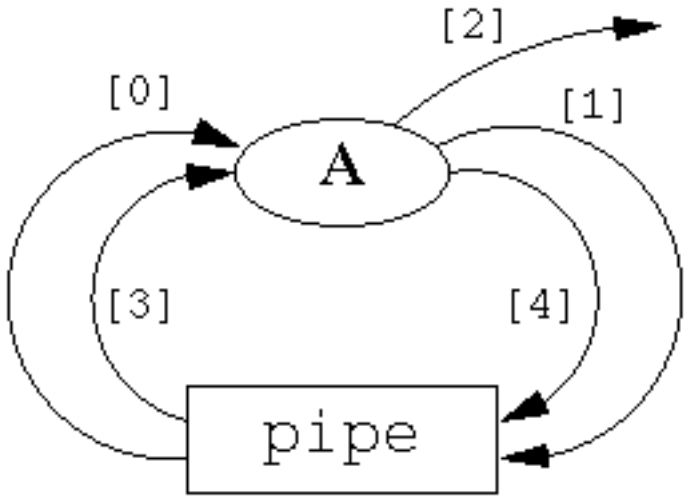
Process A

file descriptor table

[0]	*standard input*
[1]	*standard output*
[2]	*standard error*
[3]	pipe *read*
[4]	pipe *write*

Ring with A Single Process (cont.)

- Status after both **`dup2(..., ...)`**



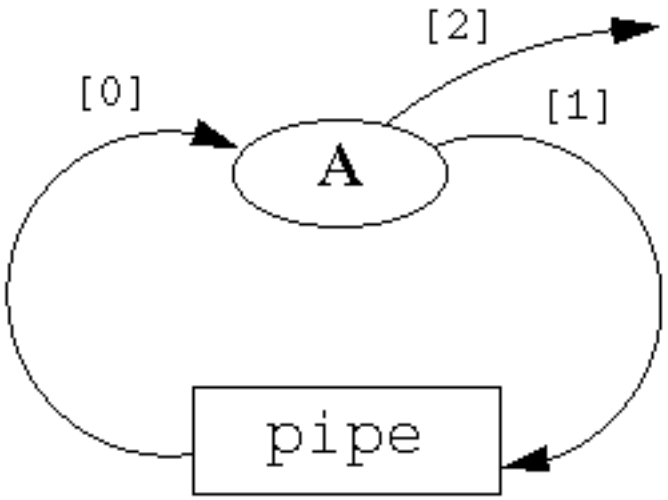
Process A

file descriptor table

[0]	pipe *read*
[1]	pipe *write*
[2]	*standard error*
[3]	pipe *read*
[4]	pipe *write*

Ring with A Single Process (cont.)

- Status after both `close(...)`



Process A

file descriptor table

[0]	pipe *read*
[1]	pipe *write*
[2]	*standard error*

Examples of Single Process Ring

- What will be the output of the following code?
  - Write first, then read

Exercise 7.2, page 228

```
int i, myint;

for (i = 0; i < 10; i++) {
   write(STDOUT_FILENO, &i, sizeof(i));
   read(STDIN_FILENO, &myint, sizeof(myint));
   fprintf(stderr, "%d\n", myint);
}
```

It prints out {0,1,…,9} on the screen!

Are you sure! What if read does not read() whole integer (say 4-8 bytes)?

Examples of Single Process Ring (cont.)

- What will be the output of the following code?
  - Read first, then write

Exercise 7.3, page 229

```
int i, myint;

for (i = 0; i < 10; i++) {
   read(STDIN_FILENO, &myint, sizeof(myint));
   write(STDOUT_FILENO, &i, sizeof(i));
   fprintf(stderr, "%d\n", myint);
}
```

The program hangs on the first read because no writes on the pipe

Examples of Single Process Ring (cont.)

- What will be the output of the following code?
  - Using standard file pointers that has buffer

Exercise 7.4, page 229

```
int i, myint;

for (i = 0; i < 10; i++) {
   printf("%d ", i);
   scanf("%d", &myint);
   fprintf(stderr, "%d\n", myint);
}
```

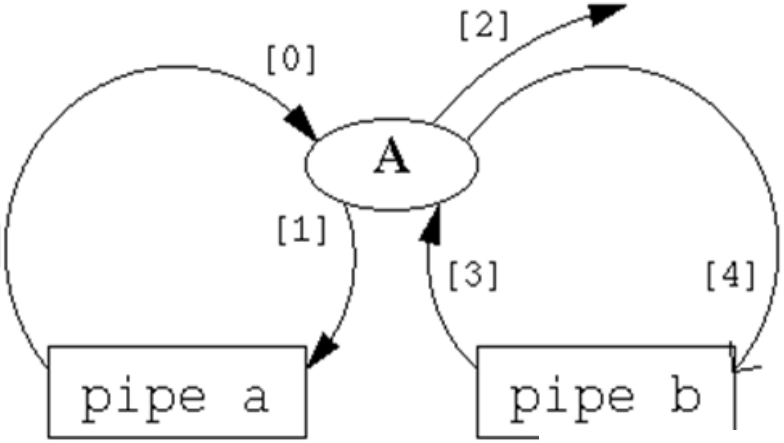
scanf may flush stdout, so the program might print 1, 2, …
OR
The program may hang on the scanf if the printf buffers its output. Put a fflush(stdout) after the printf to get output.

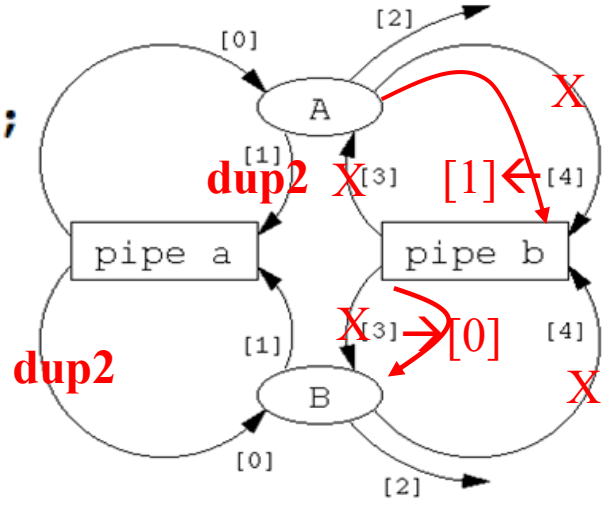
A Ring of Two Processes

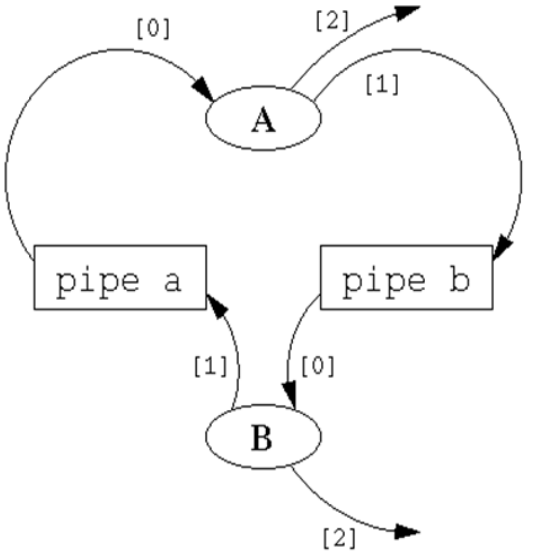
- Code section without error check

Example 7.5, page 230

```
int fd[2];
pid_t haschild;

pipe(fd);
dup2(fd[0], STDIN_FILENO);
dup2(fd[1], STDOUT_FILENO);
close(fd[0]);
close(fd[1]);
pipe(fd);
haschild = fork();
if (haschild > 0)
   dup2(fd[1], STDOUT_FILENO);
else if (!haschild)
   dup2(fd[0], STDIN_FILENO);
close(fd[0]);
close(fd[1]);
```

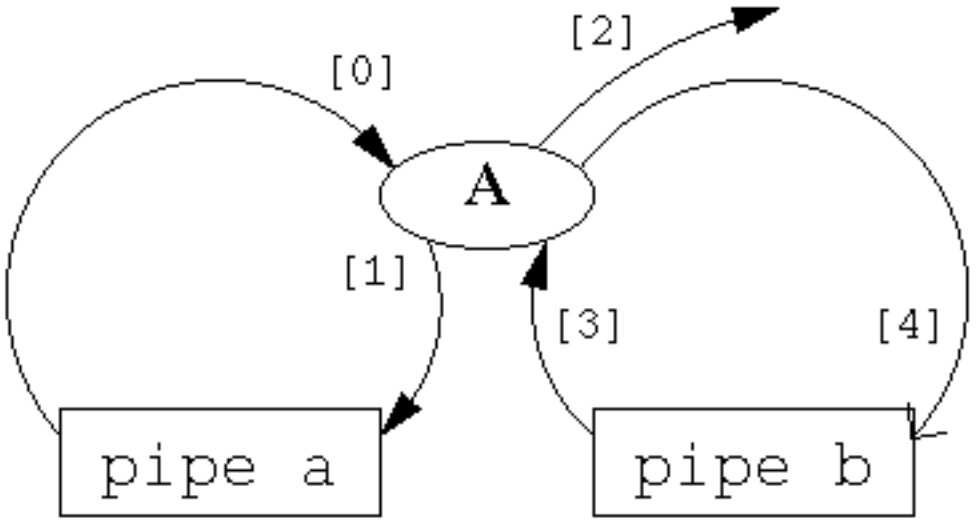






A Ring of Two Processes (cont.)

- Status after the 2nd pipe ()



Process A

file descriptor table

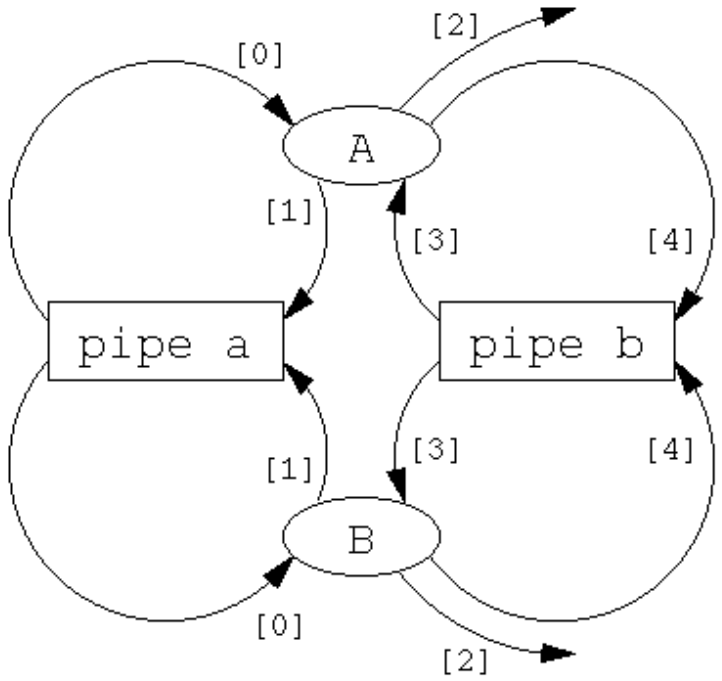
[0]	pipe a *read*
[1]	pipe a *write*
[2]	*standard error*
[3]	pipe b *read*
[4]	pipe b *write*

```
pipe(fd);
dup2(fd[0], STDIN_FILENO);
dup2(fd[1], STDOUT_FILENO);
close(fd[0]);
close(fd[1]);
pipe(fd);
```

A Ring of Two Processes (cont.)

```
pipe(fd);
dup2(fd[0], STDIN_FILENO);
dup2(fd[1], STDOUT_FILENO);
close(fd[0]);
close(fd[1]);
pipe(fd);
haschild = fork();
```

- Status after fork()



Process A

file descriptor table

[0]	pipe a *read*
[1]	pipe a *write*
[2]	*standard error*
[3]	pipe b *read*
[4]	pipe b *write*

Process B

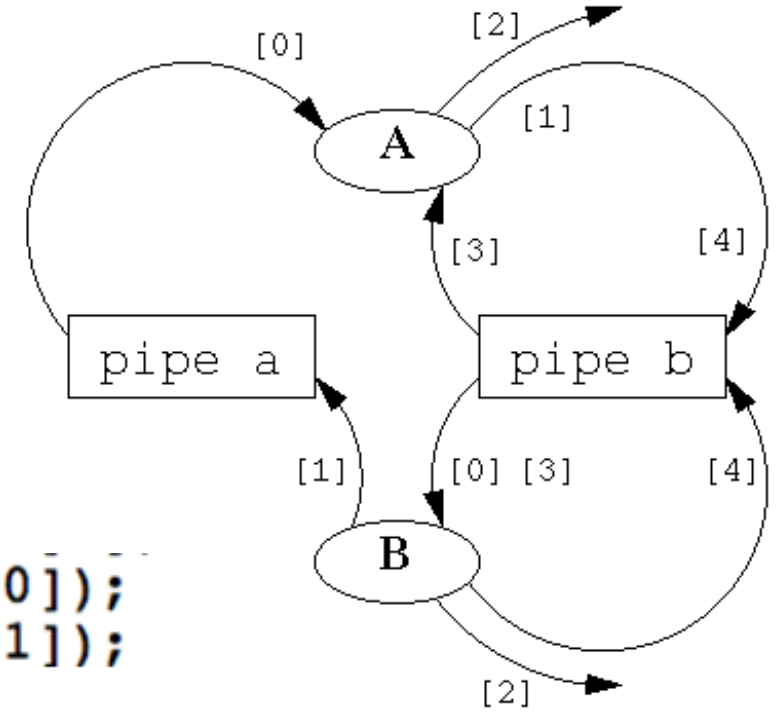
file descriptor table

[0]	pipe a *read*
[1]	pipe a *write*
[2]	*standard error*
[3]	pipe b *read*
[4]	pipe b *write*

A Ring of Two Processes (cont.)

- After dup2()

```
if (haschild > 0)
   dup2(fd[1], STDOUT_FILENO);
else if (!haschild)
   dup2(fd[0], STDIN_FILENO);
```




```
close(fd[0]);
close(fd[1]);
```

Process A

file descriptor table

[0]	pipe a *read*
[1]	pipe b *write*
[2]	*standard error*
[3]	pipe b *read*
[4]	pipe b *write*

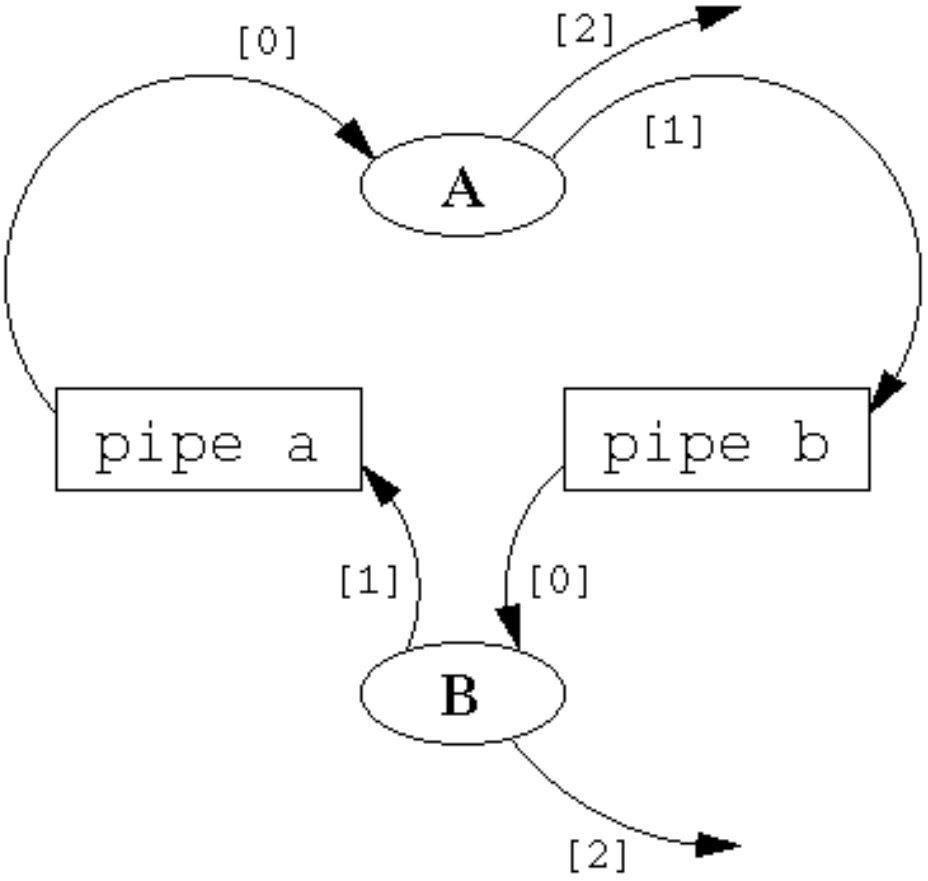
Process B

file descriptor table

[0]	pipe b *read*
[1]	pipe a *write*
[2]	*standard error*
[3]	pipe b *read*
[4]	pipe b *write*

A Ring of Two Processes (cont.)

- After close()



Process A

file descriptor table

[0]	pipe a *read*
[1]	pipe b *write*
[2]	*standard error*

Process B

file descriptor table

[0]	pipe b *read*
[1]	pipe a *write*
[2]	*standard error*

A Ring of n Processes (no error check)

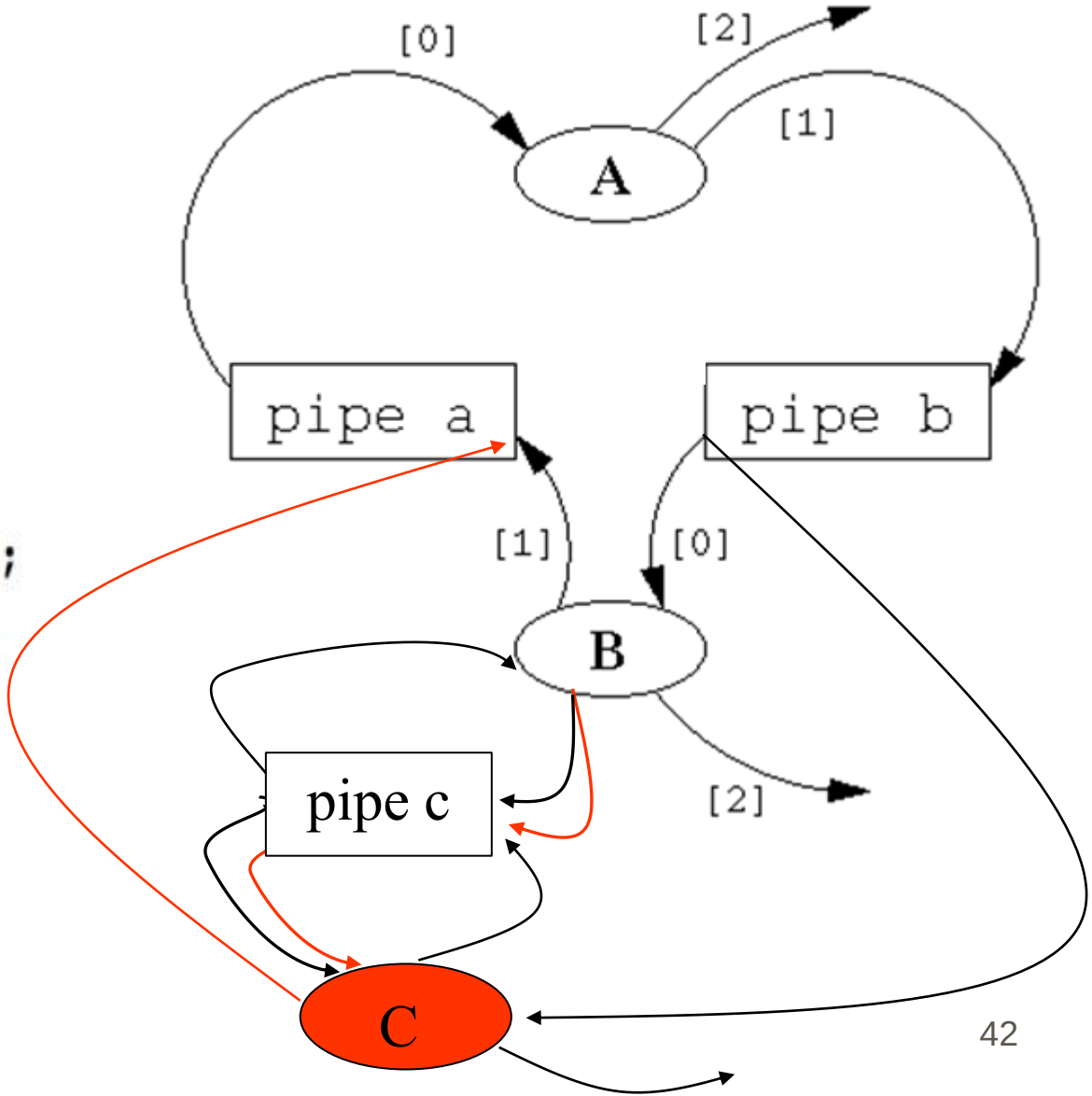
Program 7.1 on page 223 [USP]

```
int main(int argc,  char *argv[ ]) {
   pid_t childpid;                /* indicates process should spawn another     */
   int error;                     /* return value from dup2 call                */
   int fd[2];                     /* file descriptors returned by pipe          */
   int i;                         /* number of this process (starting with 1)   */
   int nprocs;                    /* total number of processes in ring          */
   pipe (fd);
   dup2(fd[0], STDIN_FILENO);
   dup2(fd[1], STDOUT_FILENO);
   close(fd[0]);
   close(fd[1]);
   for (i = 1; i < nprocs;  i++) {          /* create the remaining processes */
      pipe(fd);
      childpid = fork();
      if (childpid > 0)                  /* for parent process, reassign stdout */
          dup2(fd[1], STDOUT_FILENO);
      else                                 /* for child process, reassign stdin */
          dup2(fd[0], STDIN_FILENO);
      close(fd[0]);
      close(fd[1]);
      if (childpid)
         break;
   }                                                  /* say hello to the world */
   fprintf(stderr, "This is process %d with ID %ld and parent id %ld\n",
           i, (long)getpid(), (long)getppid());
   return 0;
}
```

Parent process break;

child process create the next pipe and new process;

```
pipe (fd);
dup2(fd[0], STDIN_FILENO);
dup2(fd[1], STDOUT_FILENO);
close(fd[0]);
close(fd[1]);
for (i = 1; i < nprocs;  i++) {
    pipe(fd);
    childpid = fork();
    if (childpid > 0)
        dup2(fd[1], STDOUT_FILENO);
    else
        dup2(fd[0], STDIN_FILENO);
    close(fd[0]);
    close(fd[1]);
    if (childpid)
       break;
}
```



Token Management

- What is a token?
- Where does the token come from?
- How does the token is used?
- What if the token is lost (or can the token be lost)?

Summary

- Inter-Process communication (IPC)
- Pipe and its operations
- FIFOs: named pipes
  - Allow un-related processes to communicate
- Ring of communicating processes
  - Steps for Ring Creation with Pipes
  - A Ring of n Processes
- Other issues in token ring
  - Which process to write/read: token management