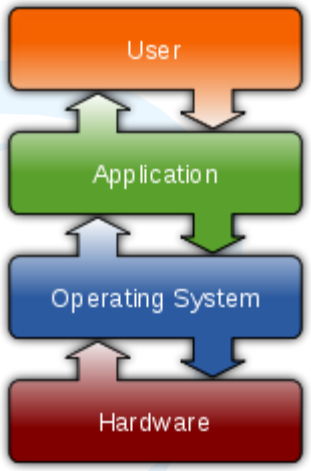
ENCE360 Operating Systems

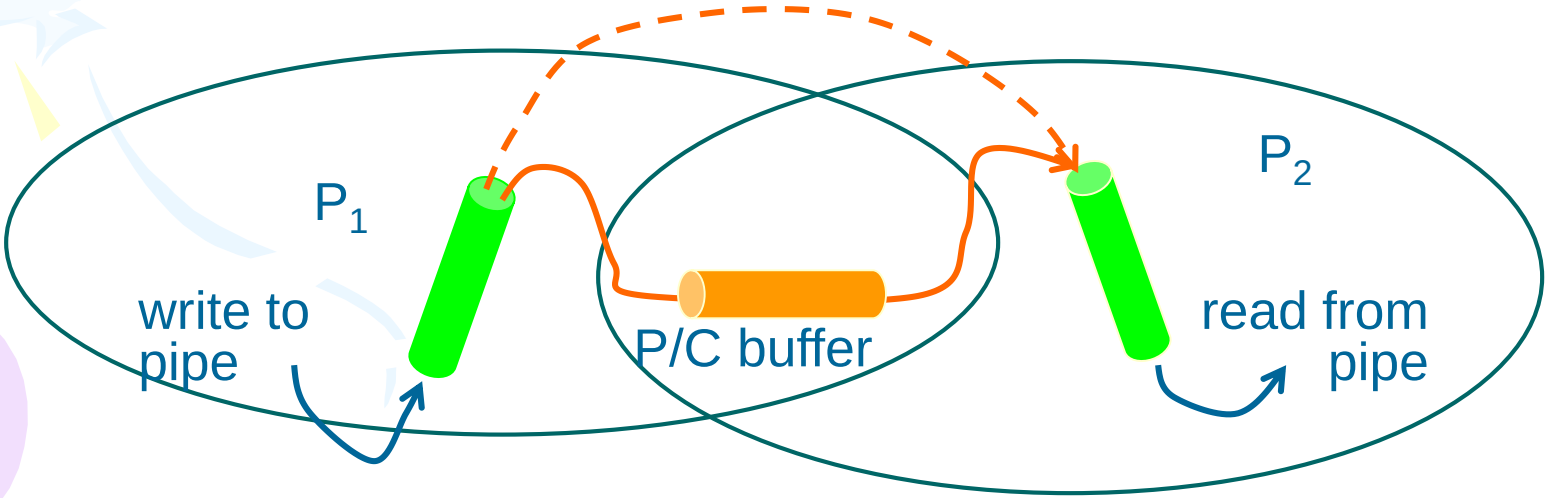


- Operating Systems -

Pipes

Stream-based IPC Using Pipes

- Functionally like using a file but more efficient.
- One way synchronised byte stream.
- Kernel buffers the data.
- For example, ls | sort | more.



Stream-based IPC Using Pipes

- Pipe Properties:
  - Synchronised byte stream.
  - Operated as a bounded buffer with blocking.
  - Each pipe is a one-way stream.
  - One to one mapping.
- No way to test a pipe for data.
- Is it possible to do two-way communication using pipes?
- Yes! But, need 2 pipes to support 2-way communication.

Linux pipes

- Major usage is combining several programs to perform multiple steps of a single task:

a | b | c

  - Standard output of each process in pipe is forwarded to standard input of next process in pipe:

stdout of a goes to stdin of b

stdout of b goes to stdin of c

Linux pipes

- created by the **mknod** or **mkfifo**

  mkfifo pipe1

- in one virtual console1, type:

  ls -l > pipe1

  and in another type:

  cat < pipe1

  output of the command run on the first console shows up on the second console
  (Note that the order in which you run the commands doesn't matter.)

Linux pipes

- To allow one process to pass information to another, they maybe linked by a pipe:
- A pipe:
  - provides one way communication
  - is guaranteed to provide FIFO delivery of information
- There are two types of pipes:
  - **Unnamed pipes** can only be used between two related processes (e.g. child/parent or child/child). They disappear when the processes have finished.
  - **Named pipes** are persistent since each pipe has an entry in the file system. Providing the **permissions** on the entry in the file system are set appropriately, any processes may communicate via this pipe.
- An important aspect to the linking of processes by unnamed pipe is that parent and child processes share file descriptors. Thus, the parent uses one of the file descriptors created by the pipe system call while the child uses the other.
- **Named** and **unnamed** pipes are **both one-way**/unidirectional (not bidirectional) queues that the OS maintains **in memory** (not on disk).

Creating pipes

- Construction of an **unnamed pipe**:
  - is achieved with the **pipe** system call
- The syntax is:

  *int **pipe**(int fdes[2])*
- The two file descriptors refer to each end of the pipe:
  - *fdes[1]*

    is used for writing to the pipe:

    write(**feds[1]**,buff,strlen(buff))
  - *fdes[0]*

    is used for reading from the pipe:

    read(**feds[0]**,buff,strlen(buff))

Construction of a **named pipe**:

- is achieved with the **mknod/fopen** system calls
- The syntax is:

mknod("./named_pipe_filename",
S_IFIFO | 0666, 0)

fopen("./named_pipe_filename", "**w**")
fwrite(buffer, 1, strlen(buffer), fp);

fopen("./named_pipe_filename", "**r**")
fread(buffer, 1, BUFSIZE, fp)

Unnamed Pipe	Named Pipe
only between parent/child processes	between any processes
doesn't need mknod() because the OS knows this is an unnamed pipe from pipe()	needs to tell the OS that this is a pipe (of file type S_FIFO) using mknod()
pipe() does not use a filename and so this pipe does not appear in the file system	open() uses a filename and so this pipe exists as a filename in the file system
disappears when processes have finished	persistent (=> access though shell)
... so don't need to close	... so need to close when finished with pipe
pipe() creates a pipe	mknod() (or mkfif0()) creates a pipe
pipe() also opens both ends of a pipe returning 2 file descriptors in an int array (usually before fork() in the parent so that the child inherits the file descriptors)	open() opens only one end of a pipe returning one file descriptor as an integer so only one file descriptor is available in each process
because pipe() opens both ends, need to close one in each process	open() only opens one end of the pipe
sending process write() to the pipe using input pid[1] – so need to close the output pid[0]	sending process opens pipe with O_WRONLY, so just use file descriptor from open() to write() to pipe
receiving process read() from pipe using output pid[0] - soneed to close the input pid[1]	receiving process opens pipe with O_RDONLY, so just use file descriptor from open()to read() from pipe
bytestream: can only use read()/write()	filestream: can use read()/write() or fgets()/fputs() or fread()/fwrite()

Establishing unnamed pipes between processes

- Generating a pipe using the pipe system call:
  1. Forking a child process using the **fork()** system call,
  2. Organizing the file descriptors, and
  3. Passing command execution information from one process to another via the pipe.
  - This process is quite common the standard I/O library provides ***popen()*** and ***pclose()*** calls.

Create an unnamed pipe
(between parent and child)

```
int main(int argc, char* argv[])
{
   int data_pipe[2];// the file desc. of the pipe
   int pid;              // pid of child process, or 0
   int rc;               // stores return values

   // first, create a pipe:
   rc = pipe(data_pipe);

   // fork a child process:
   pid = fork();

   switch (pid) {

          case 0:      // inside child process
             do_child(data_pipe);

          default:  // inside parent process
             do_parent(data_pipe);
           }

   return 0;
}
```

```
/* example of the child processing. */
void do_child(int data_pipe[])
{
   int c;     /*data received from the parent  */
   int rc;    /* return status of read().          */

   /*close the un-needed write-part of pipe: */
   close(data_pipe[1]);

  /* loop reading data from pipe: */
   while ((rc = read(data_pipe[0], &c, 1)) > 0)
             putchar(c);

   /* got EOF via the pipe. */
   exit(0);
}
```

Establishing named pipes

- Once the named pipe has been created (**mkfifo()**), the named file can be opened and closed using the standard **fopen** and **fclose** system calls.
- Problems:
  - A process that tries to read an empty pipe is blocked;
  - likewise a process that tries to write to a full pipe is blocked.

- Notes:
- I/O operations on a FIFO are essentially the same for normal pipes as for a file:
  - Although a **pipe resides in the kernel and not on a physical file system**, we can treat the pipe as a stream:
    - opening it up with fopen(),
    - and closing it with fclose().

Server code for a named pipe

```
/*************************************************************************
 Simple server for named pipes
USAGE: fserver & - it blocks on input
 *************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <sys/stat.h>
#include <unistd.h>

#define FIFO_FILE       "MY_NAMED_PIPE"

int main(void)
{
        FILE *fp;
        char buffer[80];
                                        /* Create the FIFO if it does not exist */
        umask(0);
        mknod(FIFO_FILE, S_IFIFO|0666, 0);

        while(1) // loop reading from the buffer
        {
                fp = fopen(FIFO_FILE, "r");
                fgets(buffer, 80, fp); //read from the client
                printf("Your message is here: %s\n", buffer);
                fclose(fp);
        }

        return(0);
}
```

Client code for a named pipe

```
/*************************************************************************
***********
 Simple client to demonstrate usage of named pipes
Type on command line fclient somemessagehere
*************************************************************************
***********/

#include <stdio.h>
#include <stdlib.h>
#define FIFO_FILE "MY_NAMED_PIPE"

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

        fp = fopen(FIFO_FILE, "w")

        fputs(argv[1], fp);     //send the message

        fclose(fp);
        return(0);
}
```

Named pipe

fgets() fputs() vs fread() fwrite() vs read() write()

```
#include <fcntl.h> // for S_IFIFO , O_RDONLY, O_WRONLY

//create a named pipe:
mknod("./named_pipe_filename", S_IFIFO | 0666, 0);
```

```
FILE* fp1 = fopen( "./named_pipe_filename" , "w");
FILE* fp0 = fopen( "./named_pipe_filename" , "r");
fputs(buffer, fp1);
fgets(buffer, BUFSIZE, fp0);
//or
fwrite(buffer, 1, strlen(buffer), fp1);
fread(buffer, 1, BUFSIZE, fp0);
```

//or

```
int fp0 = open("./named_pipe_filename", O_RDONLY);
int fp1 = open( "./named_pipe_filename", O_WRONLY);
write(fp1, buffer, strlen(buffer));
read(fp0, buffer, BUFSIZE);
```

unnamed vs named pipes

unnamed pipe:

```
// create and open an unnamed pipe:
int pid[2];
pipe(pid);

write(pid[1], buffer, strlen(buffer));
read(pid[0], buffer, BUFSIZE);
```

named pipe:

```
// create and open a named pipe:
int pid0, pid1;
mknod("./named_pipe_filename", S_IFIFO | 0666, 0);
pid1 = open("./named_pipe_filename", O_WRONLY);
pid0 = open("./named_pipe_filename", O_RDONLY);

write(pid1, buffer, strlen(buffer));
read(pid0, buffer, BUFSIZE);
```

Even the read() and write() could be identical for both named and unnamed pipes if both used int pid[2]

```
int pid[2];
```

```
//unnamed
pipe(pid);
```

```
//named:
 mknod("./named_pipe_filename",S_IFIFO | 0666, 0);
pid[0] = open("./named_pipe_filename",O_RDONLY);
pid[1] = open("./named_pipe_filename",O_WRONLY);
```

```
// exactly the same read() and write() function calls
// for both the named pipe and unnamed pipe:

read(pid[0], buffer, BUFSIZE);

write(pid[1], buffer, strlen(buffer));
```

Named pipe

fgets() fputs() vs fread() fwrite() vs read() write()

- fopen is ANSI C, open is a system call
- fopen is higher level, supports quite a few stdio calls (fprintf, fscanf etc.)
- fopen is buffered, generally higher performance but need to be careful to fflush()
- fopen does end of line translation in text mode (e.g. useful for using windows text files in linux/mac)

Obviously you cannot use fopen for an anonymous/unamed pipe however!

Python named pipe

os.mkfifo(*path*, *mode=0o666*, *, *dir_fd=None*)

Create a FIFO (a named pipe) named *path* with numeric mode *mode*.

FIFOs are pipes that can be accessed like regular files.

FIFOs exist until they are deleted

Generally, FIFOs are used as rendezvous between "client" and "server" type processes.
E.g. the server opens the FIFO for reading, and the client opens it for writing.

Note that mkfifo() doesn't open the FIFO — it just creates the rendezvous point.

Python unnamed pipes

```
import os, sys

print "The child will write text to a pipe and "
print "the parent will read the text written by child..."

# file descriptors r, w for reading and writing
r, w = os.pipe()

processid = os.fork()
if processid:
    # This is the parent process
    # Closes file descriptor w
    os.close(w)
    r = os.fdopen(r)
    print "Parent reading"
    str = r.read()
    print "text =", str
    sys.exit(0)
else:
    # This is the child process
    os.close(r)
    w = os.fdopen(w, 'w')
    print "Child writing"
    w.write("Text written by child...")
    w.close()
    print "Child closing"
    sys.exit(0)
```

Communication with pipes

pipe = kernel supplied object for message passing

send to pipe (read & write ends open):

- writing to a full pipe blocks
- if receiver process waiting, then receiver is given message and released
- if no receiver waiting, message is queued

Communication with pipes

receive from pipe (read & write ends open):

- reading from an empty pipe blocks
- if message ready, obtain message from front of queue and leave
- may have multiple queued receivers

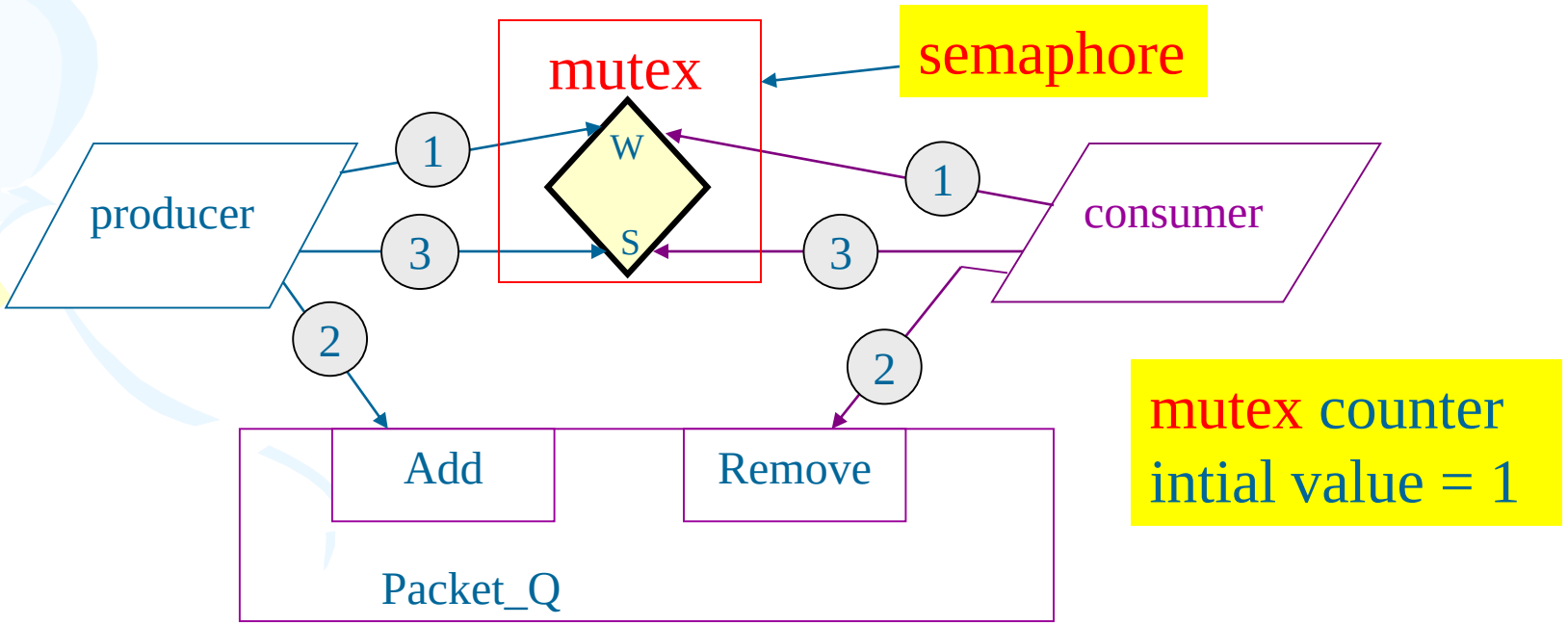
Communication with pipes

read or write ends closed:

- reading from a pipe with the write end closed returns EOF
- writing to a pipe (or FIFO) with the read end closed raises SIGPIPE (since pipes and FIFOs have the same semantics)

Mutual Exclusion (mutex) in Pipe Process

- Stream-2-Pipe example: want mutually exclusive access to packet_Q



Note: **s** = signal() = sem_post() = up() = pthread_mutex_unlock()
w = wait() = sem_wait() = down() = pthread_mutex_lock()

Adding to Packet_Q

```
Protected_Add(  P :  packet_buffer )
{ mutex.Wait;   // gain exclusive access
  Packet_Q.Add( P );    // add to Q
  mutex.Signal;//release exclusive access
}
```

Removing from Packet_Q

```
Protected_Remove(var P:  packet_buffer)
1 { mutex.Wait;  // gain exclusive access
2   Packet_Q.Remove( P );//remove from Q
3   mutex.Signal;//release exclusive access
  }
```

Synchronization

- Stream-2-Pipe – only want to allow:

  Remove : when there is a packet available

  Add : when there is space for the packet

- need more semaphores to synchronize!
  - will introduce 2 more

Additional Semaphores

packets_in_Q : semaphore = 0;

// used to block Removers until a packet ready

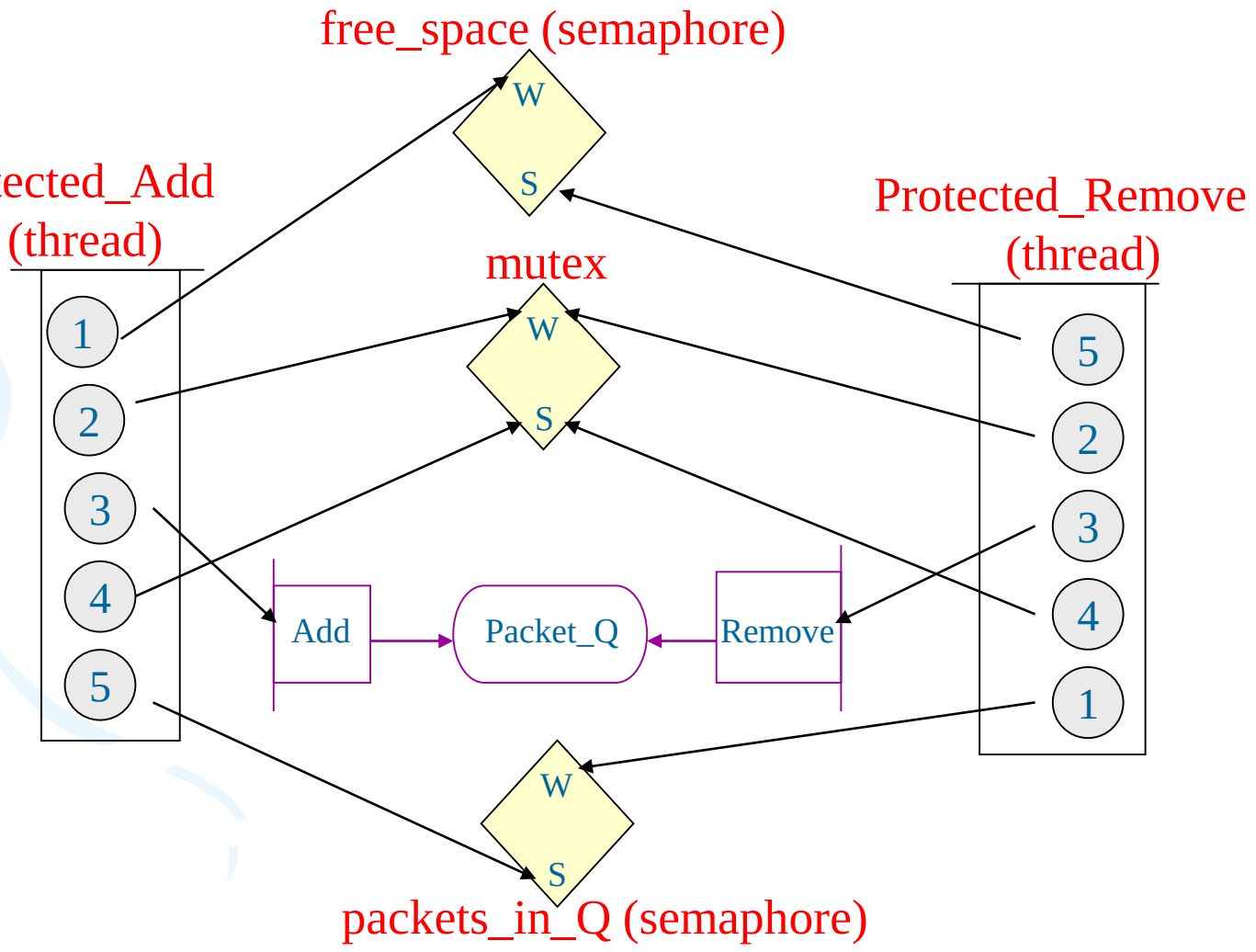
// initially no packets ready – gate closed!

free_space : semaphore = Q_Size;

// used to block Adders until space is available

// initially all spaces in Packet_Q are available

Mutex & Synchronization in Pipe Process (with 2 threads)



Note: **s** = signal() = sem_post() = up() = pthread_mutex_unlock()
w = wait() = sem_wait() = down() = pthread_mutex_lock()

Revised **Add** to Packet_Q

```
Protected_Add(  P :  packet_buffer )
{
1  free_space.Wait;      //get space in Packet_Q
2  mutex.Wait;           // gain exclusive access
3  Packet_Q.Add( P );    // add to Q
4  mutex.Signal;         // release exclusive access
5  packets_in_Q.Signal;// packet now ready!
}
```

Revised **Remove** From Packet_Q

```
Protected_Remove( var P :  packet_buffer)
{
1  packets_in_Q.Wait;      // wait for packet
2  mutex.Wait;             // gain exclusive access
3  Packet_Q.Remove( P );// remove from Q
4  mutex.Signal; // release exclusive access
5  free_space.Signal;      //1 more freed space!
}
```

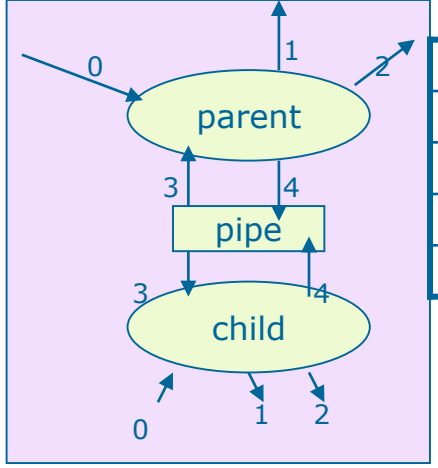
Duplicating file descriptors

- Duplicating file descriptors
  - file descriptor is a small positive integer that is used to refer to a fie that is being manipulated.
  - **There are three standard file descriptors:**
    - **0** standard input (STDIN_FILENO)
    - **1** standard output (STDOUT_FILENO), and
    - **2** standard error
- A pipe between two processes is regarded as being a file (albeit transient and stored in memory)
  - **Pipe descriptors:**
    - a read only file descriptor and
    - a write only file descriptor
  - **Often there is a need to set the read end of a pipe to be the same as standard input, or the write end of a pipe to be the same as standard output.**
  - This is achieved using the **dup2()** system call.

Simple example

```
/*Duplicating file fds*/
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(void) {
int fd;

    fd = open("my.file", O_WRONLY | O_CREAT, S_IRUSR | S_IWUSR)

    dup2(fd ,STDOUT_FILENO)

    execl("/bin/ls","ls","-l",NULL);
            ……
}
```

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
#include<fcntl.h>

void main(void)
{
      int fd[2];
      pid_t childpid;
     pipe(fd);
     if (( childpid=fork())==0) { // in child:
            dup2(fd[1],STDOUT_FILENO);
            close(fd[0]);
            close(fd[1]);
            execl("/bin/ls","ls","-l",NULL);
            perror("The exec of ls failed");
     } else { // in parent:
            dup2(fd[0], STDIN_FILENO);
           close(fd[0]);
           close(fd[1]);
           execl("/bin/sort", "sort","-n","+4",NULL);
           perror("the exec of sort failed");
     }
     exit(0);
}
```

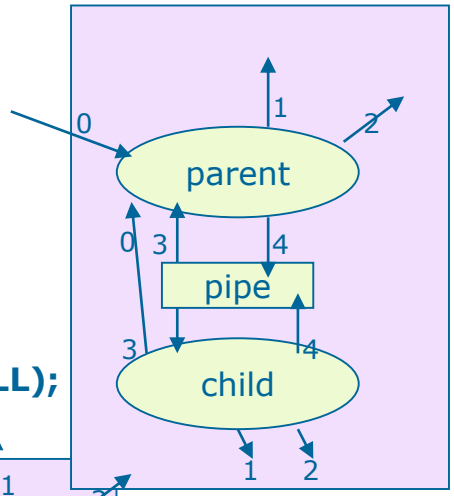


0 stdin
1 stdout
2 stderror
3 pipe read
4 pipe write

File Descriptor Table (FDT) after fork parent

0 stdin
1 stdout
2 stderror
3 pipe read
4 pipe write

FDT after fork child

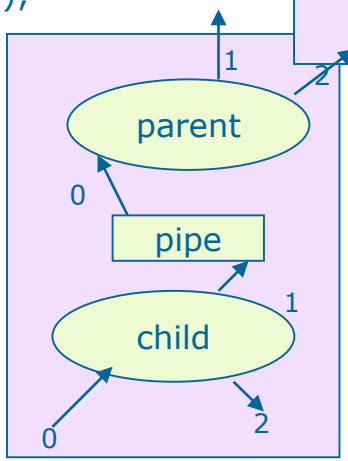


0 read pipe
1 stdout
2 stderror
3 pipe read
4 pipe write

FDT after dup2 parent

0 stdin
1 write pipe
2 stderror
3 pipe read
4 pipe write

FDT after dup2 child



0 stdin
1 write pipe
2 stderror

0 read pipe
1 stdout
2 stderror

FDTs after execl

Message passing - Signals

- Sending a signal to a process indicates the occurrence of an exceptional condition.
- Signals may be sent by
  - other processes (including those constituting the operating system), or
  - by the user from the keyboard.
- There are a variety of signals that maybe sent to a process, each used to indicate a reasonably generic, yet exceptional, condition. For example (Linux):
  - **SIGINT** indicates that the process is being **interrupted** with the typical response being that the process prematurely ends in a graceful manner;
  - **SIGQUIT** indicates that the process is to end and to place a copy of it's address space in a file called core;
  - **SIGILL** indicates an attempt to execute an illegal instruction;
  - **SIGSEGV** indicates an attempt to access memory beyond that to which the process has permission.

Each signal has its own default signal handler, the behaviour of which is defined in your local man pages

SIGABRT	Process abort signal.
SIGALRM	Alarm clock.
SIGFPE	Erroneous arithmetic operation.
SIGHUP	Hangup.
SIGILL	Illegal instruction.
SIGINT	Terminal interrupt signal.
SIGKILL	Kill (cannot be caught or ignored).
SIGPIPE	Write on a pipe with no one to read it.
SIGQUIT	Terminal quit signal.
SIGSEGV	Invalid memory reference.
SIGTERM	Termination signal.
SIGUSR1	User-defined signal 1.
SIGUSR2	User-defined signal 2.
SIGCHLD	Child process terminated or stopped.
SIGCONT	Continue executing, if stopped.
SIGSTOP	Stop executing (cannot be caught or ignored).
SIGTSTP	Terminal stop signal.
SIGTTIN	Background process attempting read.
SIGTTOU	Background process attempting write.
SIGBUS	Bus error.
SIGPOLL	Pollable event.
SIGPROF	Profiling timer expired.
SIGSYS	Bad system call.
SIGTRAP	Trace/breakpoint trap.
SIGURG	High bandwidth data is available at a socket.
SIGVTALRM	Virtual timer expired.
SIGXCPU	CPU time limit exceeded.
SIGXFSZ	File size limit exceeded.

Generating signals

- Signals may be generated from various sources:
  - Hardware such as divide by zero
  - Operating System such as notifying that file size limit is exceeded.
  - User by entering keystrokes such as ctrl-Z (SIGQUIT), ctrl-C (the SIGINT signal), or using the kill command.
  - Other processes such as a child process notifying its parent that it has terminated (***SIGCHLD***).

- Signals may be sent from processes using a variety of system calls:
  - ***raise()*** is used to send signals to **yourself**
    - **it uses *kill(getpid(); sig);***
  - ***kill()*** is used to send signals to a **specified process**
  - ***alarm()*** sends the ***SIGALRM*** system to itself after a specified number of real seconds
  - **signal numbers**: 0-31
  - syntax:
    - *kill(int pid, int signal);*
    - if pid = 0 -> signal goes to all processes (except OS processes)

Actions on signals

- The receiving process can do one of three things upon receiving a signal:
  - **default**
    - by default, most signals terminate the process.
  - **catch**
    - Catch the signal. This also cannot be done for **SIGSTOP** and **SIGKILL**.
      - If process catches the signal it executes a special signal handling routine.
        - » this treatment corresponds to java *try – catch* pair and eg IOException(e){}
  - **ignore**
    - All but two of the signals can be ignored. The two exceptions are:
      - **SIGSTOP** which stops the process from executing
      - **SIGKILL** which kills the process

Example

```
#include <signal.h>

void sighup(); // routines child calls when signal from parent
void sigint();
void sigquit();

int main (int argc, const char * argv[])
{
    int child_pid;
    if((child_pid = fork()) < 0 ) exit(1); // fork() failed */
    if(child_pid == 0) {//child: fork() succeeded, now in the child
        signal(SIGHUP,  sighup);// name functions to receive signals
        signal(SIGINT,  sigint);
        signal(SIGQUIT, sigquit);
        for(; ;); //loop forever
    }
else {//parent
    printf("\nPARENT: sending SIGHUP signal to child\n\n");
    kill(child_pid,SIGHUP);
    printf("\nPARENT: sending SIGINT signal to child\n\n");
    kill(child_pid,SIGINT);
    printf("\nPARENT: sending SIGQUIT signal to child\n\n");
    kill(child_pid,SIGQUIT);
    }
}
```

Example (continued)

```
void sighup() {
  signal(SIGHUP,sighup); /* reset signal
When a signal handler is called in a C program, its corresponding signal action is usually set to SIG_DFL.
You must reset the signal action if you want to handle the same signal again.
If you do not, the default action is taken on subsequent exceptions.
The handling of the signal can be reset from inside or outside the handler by calling signal()*/
   printf("CHILD: I have received a SIGHUP\n");
}
void sigint() {
  signal(SIGINT,sigint); /* reset signal */
To ignore a ctrl-c command from the command line. we could do: signal(SIGINT, SIG_IGN);
TO reset system so that SIGINT will causes a termination anywhere, we would signal:(SIGINT, SIG_DFL);
   printf("CHILD: I have received a SIGINT\n");
}
void sigquit() {
  sleep(1);
  printf("CHILD: My Parent process has killed me!!!\n");
  printf("CHILD: cleaning up...\n");
  sleep(5);
  exit(0);
}

//If need to tell calling program you exited with the "I-exited-on-SIGINT" status,
//then you cannot "fake" the proper exit status with an exit(3). So do this:
void sigint_handler(int sig)
{
        [do some cleanup]
        signal(SIGINT, SIG_DFL);
        kill(getpid(), SIGINT);
}
```

Signal notes

- **Multiple types of signal can be handled from one signal handler**. For example:

```
void handler(int sigNum) {
        if      (sigNum == SIGCHLD) printf("got a SIGCLHD\n");
        else if (sigNum == SIGQUIT) printf("got a SIGQUIT\n");
}

signal(SIGCHLD, &handler);
signal(SIGQUIT, &handler);
```

- **SIGCHLD signal is sent to a parent process when one of it's child processes exits**.

  So it is not necessary to trigger it explicitly with “kill”.

  SIGCHLD signal also enables the OS to clean up resources used by a child process after its termination.

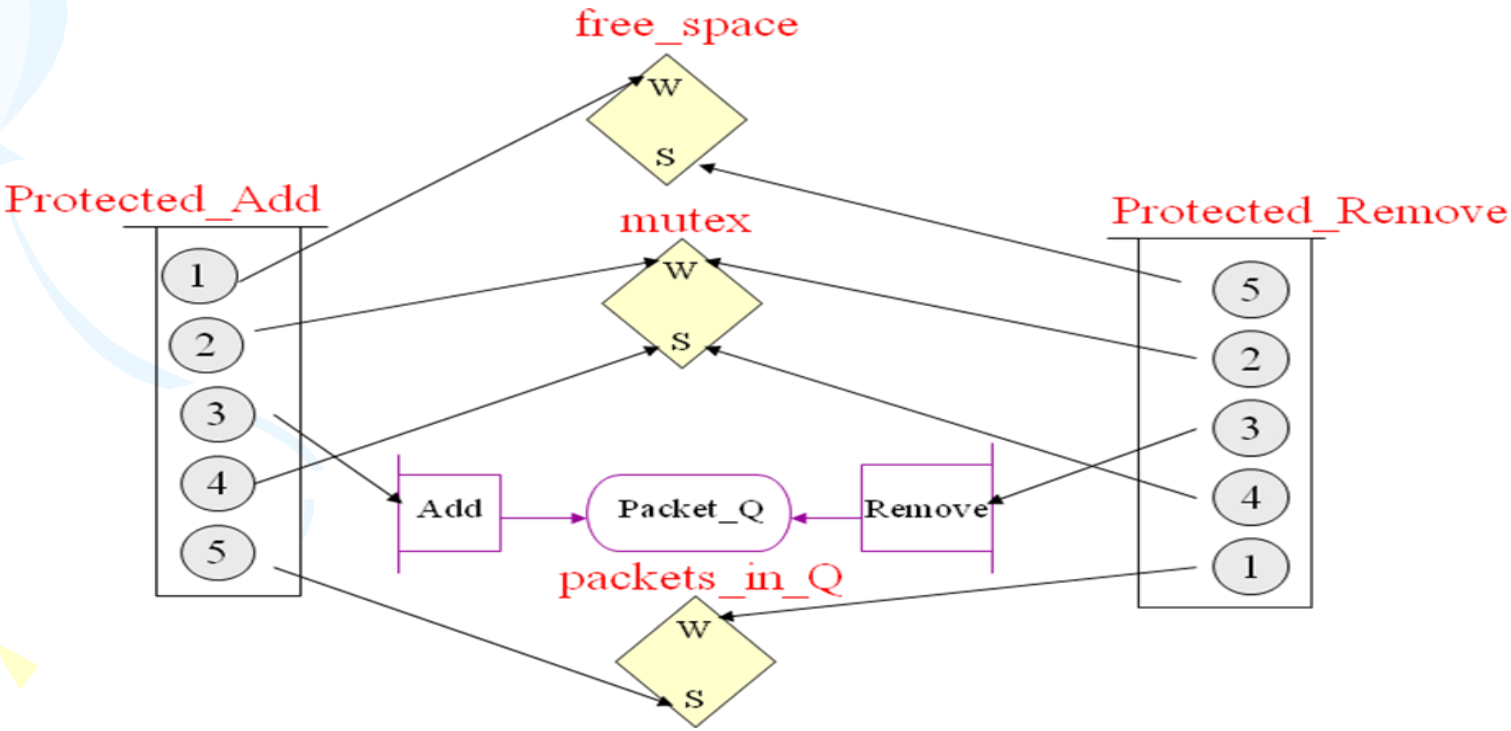
Signal summary

- Signals provide a basic communication technique:
  - They do not carry any information
  - Participating processes must know each others process IDs
  - The number of signals is limited
  - Cooperating processes must agree on the meaning of each signal
  - No easy way for the sending process to know if its signal was received
  - Signal manipulation can be tricky

Sample Exam Question

- In terms of *semaphores*, *mutexes*, *threads* and *processes*, **explain the operation of a pipe using the following diagram** (which represents a pipe).

  Ensure your explanation encompasses *semaphores*, *mutexes*, *threads, processes and the initial state*.)



Sample Exam Answer

Protected_Add and ***Protected_Remove*** **are two threads in a process** connected by the queue, *Packet_Q* – which is shared memory in this process.

(This is a producer consumer configuration synchronised by two semaphores (*free_space* and *packets_in_Q*) and a *mutex*.)

In the application programs, this (one way) **pipe appears like a file that has write-only permission** in one process (where file write calls *Protected_Add*) **and as a read-only file** in the other process (where file read calls *Protected_Remove*).

* reading from a pipe with the **write end closed returns EOF**
* writing to a pipe (or FIFO) with the **read end closed raises SIGPIPE**

free_space is used to used to block *Add* until space is available in *Packet_Q*
packets_in_Q is used to block *Remove* until a packet is ready in *Packet_Q*.
mutex ensures that *Add* and *Remove* cannot happen at the same time.

Sample Exam Answer

Initially when the pipe is created, Packet_Q is empty, *packets_in_Q*=0, *free_space*=*Q_Size*, mutex=1.

Protected_Add runs its steps 1 to 5 as follows:

1. free_space.Wait: get access to space in Packet_Q
2. mutex.Wait: gain exclusive access to Add/Remove
3. Packet_Q.Add: add packet to Packet_Q
4. mutex.Signal: release exclusive access
5. packets_in_Q.Signal: packet is now ready to Remove

Protected_Remove runs its steps 1 to 5 as follows:

1. packets_in_Q.Wait: wait for packet
2. mutex.Wait: gain exclusive access to Add/Remove
3. Packet_Q.Remove: remove packet from Packet_Q
4. mutex.Signal: release exclusive access
5. free_space.Signal: one more freed space for Add

Sample Exam Question

Describe the differences between unnamed pipes and named pipes using the code below.

unnamed pipe:

```
// create and open an unnamed pipe:
int pid[2];
pipe(pid);

write(pid[1], buffer, strlen(buffer));
read(pid[0], buffer, BUFSIZE);
```

named pipe:

```
// create and open a named pipe:
int pid0, pid1;
mknod("./named_pipe_filename", S_IFIFO | 0666, 0);
pid1 = open("./named_pipe_filename", O_WRONLY);
pid0 = open("./named_pipe_filename", O_RDONLY);

write(pid1, buffer, strlen(buffer));
read(pid0, buffer, BUFSIZE);
```

Sample Exam Answer

Unnamed Pipe	Named Pipe
only between parent/child processes	between any processes
doesn't need mknod() because the OS knows this is an unnamed pipe from pipe()	needs to tell the OS that this is a pipe (of file type S_FIFO) using mknod()
pipe() does not use a filename and so this pipe does not appear in the file system	open() uses a filename and so this pipe exists as a filename in the file system
disappears when processes have finished	persistent (=> access though shell)
... so don't need to close	... so need to close when finished with pipe
pipe() creates a pipe	mknod() (or mkfif0()) creates a pipe
pipe() also opens both ends of a pipe returning 2 file descriptors in an int array (usually before fork() in the parent so that the child inherits the file descriptors)	open() opens only one end of a pipe returning one file descriptor as an integer so only one file descriptor is available in each process
because pipe() opens both ends, need to close one in each process	open() only opens one end of the pipe
sending process write() to the pipe using input pid[1] – so need to close the output pid[0]	sending process opens pipe with O_WRONLY, so just use file descriptor from open() to write() to pipe
receiving process read() from pipe using output pid[0] - soneed to close the input pid[1]	receiving process opens pipe with O_RDONLY, so just use file descriptor from open()to read() from pipe
bytestream: can only use read()/write()	filestream: can use read()/write() or fgets()/fputs() or fread()/fwrite()

Note: Not part of above answer: named and unnamed pipes are both one-way/unidirectional (not bidirectional) queues that the OS maintains in memory (not on disk).

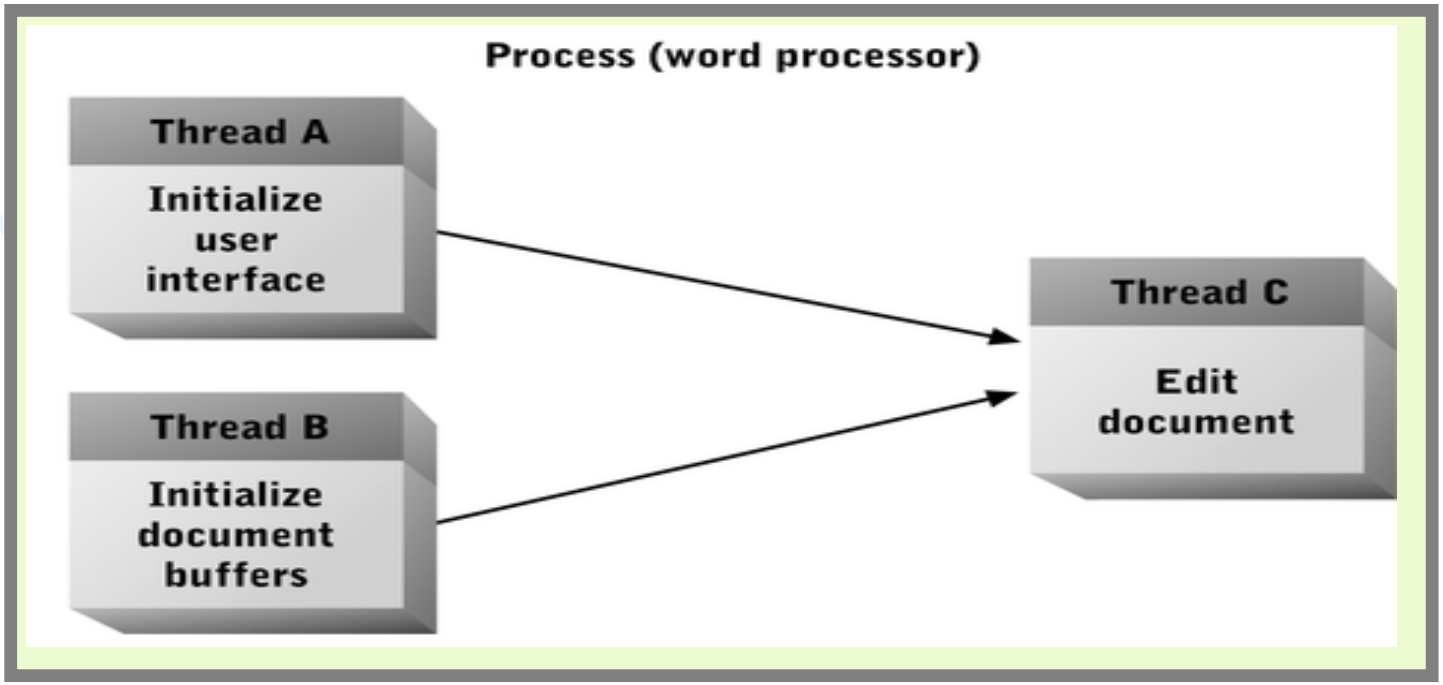
Sample Exam Question

Signals provide a very basic communication technique between processes. Describe signals, their purpose and limitations.

- They do not carry any information
- Participating processes must know each other's process IDs
- The number of signals is limited (examples)
- Cooperating processes must agree on the meaning of each signal
- No easy way for sending process to know if its signal was received
- Most signals terminate the process
- A signal handling routine can catch a signal instead of terminating the process - except SIGKILL, SIGSTOP
- if signal is not reset in signal handling routine, it cannot process more signals
- may come from hardware (/0), OS (file limit), user (ctrl-C), or other process
- only two user defined signals, SIGUSR1, SIGUSR2

Interprocess Communication using Threads

- Several threads enable concurrent execution within a single process
- All threads within a process have access to the same data



Interprocess Communication Using Pipes

- Performed by writing to and reading from memory shared by processes
- Shared memory can be thought of as a pipe that supports the flow of data from one process to another process

Interprocess Communication Using Sockets