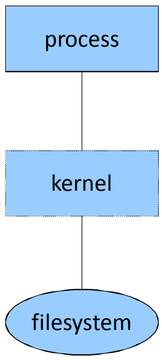
Operating Systems

Interprocess Communication (IPC)

IPC

- Unix
  - pipes, fifos (named pipes)
- System V
  - signals
  - message queues
  - semaphores
  - shared memory
- Posix
  - signals
  - message queues
  - semaphores
  - shared memory

Persistence of IPC Objects

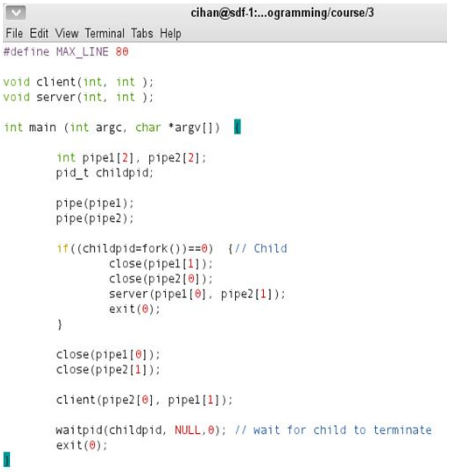


- process-persistent IPC:
  - exists until last process with IPC object closes the object
- kernel-persistent IPC
  - exists until kernel reboots or IPC object is explicitly deleted
- filesystem-persistent IPC
  - exists until IPC object is explicitly deleted

Pipes

- A pipe provides a one-way flow of data
  - example: `who | sort| lpr`
- The difference between a file and a pipe:
  - pipe is a data structure in the kernel.
- A pipe is created by using the pipe system call
  - `int pipe(int* filedes);`
  - Two file descriptors are returned
    - filedes[0] is open for reading
    - filedes[1] is open for writing
- Typical size is 512 bytes (Minimum limit defined by POSIX)

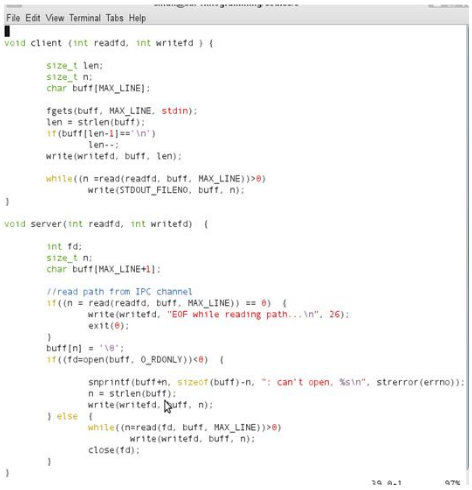
Pipe Example - II



cihan@sdf-1:...ogramming/course/3

File Edit View Terminal Tabs Help

```c
#define MAX_LINE 80

void client(int, int );
void server(int, int );

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

        int pipe1[2], pipe2[2];
        pid_t childpid;

        pipe(pipe1);
        pipe(pipe2);

        if((childpid=fork())==0)  {// Child
                close(pipe1[1]);
                close(pipe2[0]);
                server(pipe1[0], pipe2[1]);
                exit(0);
        }

        close(pipe1[0]);
        close(pipe2[1]);

        client(pipe2[0], pipe1[1]);

        waitpid(childpid, NULL,0); // wait for child to terminate
        exit(0);
}
```



File Edit View Terminal Tabs Help

```c
void client (int readfd, int writefd ) {

        size_t len;
        size_t n;
        char buff[MAX_LINE];

        fgets(buff, MAX_LINE, stdin);
        len = strlen(buff);
        if(buff[len-1]=='\n')
                len--;
        write(writefd, buff, len);

        while((n =read(readfd, buff, MAX_LINE))>0)
                write(STDOUT_FILENO, buff, n);
}

void server(int readfd, int writefd)  {

        int fd;
        size_t n;
        char buff[MAX_LINE+1];

        //read path from IPC channel
        if((n = read(readfd, buff, MAX_LINE)) == 0)  {
                write(writefd, "EOF while reading path...\n", 26);
                exit(0);
        }
        buff[n] = '\0';
        if((fd=open(buff, O_RDONLY))<0)  {

                snprintf(buff+n, sizeof(buff)-n, ": can't open, %s\n", strerror(errno));
                n = strlen(buff);
                write(writefd, buff, n);
        } else  {
                while((n=read(fd, buff, MAX_LINE))>0)
                        write(writefd, buff, n);
                close(fd);
        }
}
```

More pipe functions

- FILE *popen (const char * command, const char *type)
  - Type is r, the calling process reads the standart output of the command,
  - Type is w, the calling process writes to the standart input of the command
  - Return file * if OK, NULL on error
- int pclose (FILE *stream);
  - Closes a standard I/O stream that was created by popen

FIFOs

- Pipes have no names, they can only be used between processes that have a parent process in common.
- FIFO stands for first-in, first-out
- Similar to a pipe, it is a one-way (half duplex) flow of data
- A FIFO has a pathname associated with it, allowing unrelated processes to access a single FIFO
- FIFOs are also called *named pipes*

FIFOs

- #include <sys/types.h>
- #include <sys/stat.h>

- int mkfifo(const char *pathname, mode_t mode)
  - returns 0 if OK, -1 on error

FIFO example

File Edit View Terminal Tabs Help

```c
#define MAX_LINE 80
#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"

void client(int, int );

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

        int readfd, writefd;

        writefd = open(FIFO1, O_WRONLY);
        readfd  = open(FIFO2, O_RDONLY);

        client(readfd, writefd);

        close(readfd);
        close(writefd);

        unlink(FIFO1);
        unlink(FIFO2);

        exit(0);
}
```

File Edit View Terminal Tabs Help

```c
#define MAX_LINE 80

#define FIFO1 "/tmp/fifo.1"
#define FIFO2 "/tmp/fifo.2"
#define FIFO_MODE (S_IRWXU | S_IRWXG | S_IRWXO)

void server(int, int );

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

        int readfd, writefd;

        if((mkfifo(FIFO1, FIFO_MODE )<0)&&(errno != EEXIST))  {
                printf("can not open %s\n",FIFO1);
                exit(-1);
        }

        if((mkfifo(FIFO2, FIFO_MODE )<0)&&(errno!= EEXIST))  {
                printf("can not open %s\n",FIFO2);
                exit(-1);
        }

        readfd  = open(FIFO1, O_RDONLY);
        writefd = open(FIFO2, O_WRONLY);

        server(readfd, writefd);

        exit(0);
}
```

Signals

- Definition
- Signal Types
- Generating Signals
- Responding to a Signal
- POSIX Signal Functions
- Signals & System Calls

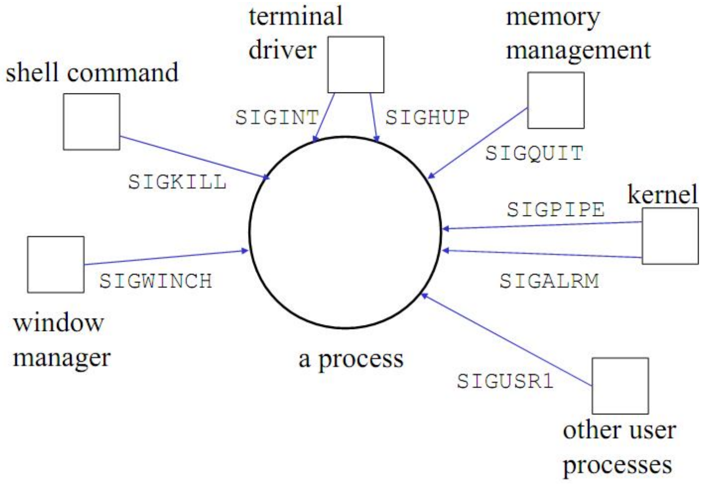
Definition

- A signal is an *asynchronous* event which is delivered to a process
- Asynchronous means that the event can occur at any time
  - may be unrelated to the execution of the process
  - e.g. user types `ctrl-C`, or the modem hangs

Common use of Signals

- Ignore a Signal
- Clean up and Terminate
- Dynamic Reconfiguration
- Report Status
- Turn Debugging on/off
- Restore Previous Handler
- Signals & System Calls

Signal Sources



Generating a Signal

- Use the Unix command
  - $> kill -KILL 4481
    - Send a SIGKILL signal to pid 4481
  - ps -l
    - To make sure process died
- kill function

```
#include <sys/types.h>
#include <signal.h>
    int kill(pid_t pid, int sig);
```

Responding to a Signal

- A process can;
  - Ignore/discard the signal (not possible for SIGKILL & SIGSTOP)
  - Execute a signal handler function, and then possibly resume execution or terminate
  - Carry out default action for that signal
- The choice is called the process' *signal disposition*

POSIX Signal System

- The POSIX signal system, uses signal sets, to deal with pending signals that might otherwise be missed while a signal is being processed
- The signal set stores collection of signal types
- Sets are used by signal functions to define which signal types are to be processed
- POSIX contains several functions for creating, changing and examining signal sets

POSIX Functions

```
#include <signal.h>

    int sigemptyset( sigset_t *set );
    int sigfillset( sigset_t *set );
    int sigismember( const sigset_t *set,
     int signo );
    int sigaddset( sigset_t *set, int signo );
    int sigdelset( sigset_t *set, int signo );
    int sigprocmask ( int how, const sigset_t *set,
sigset_t *oldset);
```

A Critical Code Region

- `sigset_t newmask, oldmask;`
- `sigemptyset( &newmask );`
- `sigaddset( &newmask, SIGINT );`
- `/* block SIGINT; save old mask */`
- `sigprocmask( SIG_BLOCK, &newmask, &oldmask );`
- `/* critical region of code */`
- `/* reset mask which unblocks SIGINT */`
- `sigprocmask( SIG_SETMASK, &oldmask, NULL );`

Example

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>

void ouch( int );

int main (void)  {

        struct sigaction act;

        act.sa_handler = ouch;

        sigemptyset(&act.sa_mask);
        act.sa_flags = 0;

        sigaction(SIGINT, &act, 0);

        while(1)  {
                printf("Hello world\n");
                sleep(1);
        }
        exit 0;
}

void ouch( int sigNo ) {

        printf("received SIGINT...\n");
}
```

- This function will continually capture the ctrl-C (SIGINT) signal.
- Default behavior is **not** restored after signal is caught.
- To terminate the program, must type ctrl-\, the SIGQUIT signal.

Interrupted System Calls

- When a system call is interrupted by a signal, a signal handler is called, returns, and then what ?
- Slow system function calls do not resume, Instead they return an error and errno is assigned EINTR.
- Some UNIXs resume non-slow system functions after the handler has finished.
- Some UNIXs only call the handler after non-slow system function call has finished.

System Calls Inside Handlers

- If a system function is called inside a signal handler then it may interact with an interrupted call to the same function in the main code.
  - e.g. malloc()
- Not a problem if the function is reentrant
  - A process can contain multiple calls to these functions at the same time. e.g. read(), write()
- A function may be non-reentrant for a number of reasons
  - It uses a static data structure
  - It manipulates the heap. e.g. malloc(), free()
  - It uses standart I/O library. e.g. printf()

Message Queues

- Unlike pipes and FIFOs, message queues support messages that have structure.
- Like FIFOs, message queues are persistent objects that must be initially created and eventually deleted when no longer required.
- Message queues are created with a specified maximum message size and maximum number of messages.
- Message queues are created and opened using a special version of the open system call, mq_open.

POSIX Message Queue Functions

- mq_open()
- mq_close()
- mq_unlink()
- mq_send()
- mq_receive()
- mq_setattr()
- mq_getattr()
- mq_notify()

mq_open(const char *name, int oflag,...)

- name
  - Must start with a slash and contain no other slashes
  - QNX puts these in the /dev/mqueue directory
- oflag
  - O_CREAT – to create a new message queue
  - O_EXCL – causes creation to fail if queue exists
  - O_NONBLOCK – usual interpretation
- mode – usual interpretation
- &mqattr – address of structure used during creation

Message Queue Persistence - I

- As noted, a message queue is persistent.
- Unlike a FIFO, however, the contents of a message queue are also persistent.
- It is not necessary for a reader and a writer to have the message queue open at the same time. A writer can open (or create) a queue and write messages to it, then close it and terminate.
- Later a reader can open the queue and read the messages.

Message Queue Persistence - II

- mkdir /dev/mqueue
- mount -t mqueue none /dev/mqueue
- ls -la /dev/mqueue

```
cihan@sdf-1:~/Desktop> ls -la /dev/mqueue/
total 0
drwxrwxrwt  2 root  root    80 2009-03-18 20:59 .
drwxr-xr-x 14 root  root  4600 2009-03-18 20:52 ..
-rwxr-x---  1 cihan users   80 2009-03-18 19:40 myqueue123
-rwxr-x---  1 cihan users   80 2009-03-18 20:59 test
cihan@sdf-1:~/Desktop>
```

Message Queue Example

- ./DropOne -q -p 11
- ./DropOne -p 101
- ./DropOne -p 99

```
File Edit View Terminal Tabs Help
-rwxr-x---  1 cihan users   80 2009-03-18 21:56 test
cihan@sdf-1:/media/KINGSTON__________/networkProgramming/course/4> ./TakeOne
Queue "/test":
        - stores at most 10 messages
        - large at most 8192 bytes each
        - currently holds 3 messages
Received message (57 bytes) from 101: Hello from process 14198 (at Wed Mar 18 21:56:26 2009
).
cihan@sdf-1:/media/KINGSTON__________/networkProgramming/course/4> ./TakeOne
Queue "/test":
        - stores at most 10 messages
        - large at most 8192 bytes each
        - currently holds 2 messages
Received message (57 bytes) from 99: Hello from process 14194 (at Wed Mar 18 21:56:24 2009
).
cihan@sdf-1:/media/KINGSTON__________/networkProgramming/course/4> ./TakeOne
Queue "/test":
        - stores at most 10 messages
        - large at most 8192 bytes each
        - currently holds 1 messages
Received message (57 bytes) from 11: Hello from process 14187 (at Wed Mar 18 21:56:21 2009
).
cihan@sdf-1:/media/KINGSTON__________/networkProgramming/course/4>
```

The effect of fork on a message queue

- Message queue descriptors are not (in general) treated as file descriptors; the unique open, close, and unlink calls should already suggest this.
- Open message queue descriptors are not inherited by child processes created by fork.
- Instead, a child process must explicitly open (using mq_open) the message queue itself to obtain a message queue descriptor

Detecting non-empty queues

- mq_receive on an empty queue normally causes a process to block, and this may not be desirable.
- Of course, O_NONBLOCK could be applied to the queue to prevent this behavior, but in that case the mq_receive call will return -1, and our only recourse is to try mq_receive again later.
- With the mq_notify call we can associate a single process with a message queue so that it (the process) will be notified when the message queue changes state from empty to non-empty

mq_notify(mqd_t mqdes, const struct sigevent *notification)

- queuefd – as usual, to identify the message queue
- sigev – a struct sigevent object that identifies the signal to be sent to the process to notify it of the queue state change.
- Once notification has been sent, the notification mechanism is removed. That is, to be notified of the next state change (from empty to non-empty), the notification must be reasserted.

Changing the process to be notified

- Only one process can be registered (at a time) to receive notification when a message is added to a previously-empty queue.
- If you wish to change the process that is to be notified, you must remove the notification from the process which is currently associated (call mq_notify with NULL for the sigev argument), and then associate the notification with a different process.

mq_getattr(queuefd,&mqstat)

- This function retrieves the set of attributes for a message queue to the struct mq_attr object named mqstat.
- Recall that the mq_flags member of the attributes is not significant during mq_open, but it can be set later (using mq_setattr).

mq_setattr(queuefd,&mqstat,&old)

- This function is used to
  - Set (or clear) to O_NONBLOCK flag in the mqattr structure for the identified message queue
  - Retrieve (if old is not NULL) the previously existing message queue attributes
- Making changes to any other members of the mqattr structure is ineffective.

Timed send and receive

- Two additional functions, mq_timedsend and mq_timedreceive, are like mq_send and mq_receive except they have an additional argument, a pointer to a struct timespec.
- This provides the absolute time at which the send or receive will be aborted if it cannot be completed (because the queue is full or empty, respectively).