Interprocess Communication

Prof. Michele Loreti

Laboratorio di Sistemi Operativi
Corso di Laurea in Informatica (L31)
Scuola di Scienze e Tecnologie

Interprocess Communication

Multiple processes may be running on a machine and maybe be controlled (spawned by `fork()`) by one of our programs.

Interprocess Communication

Multiple processes may be running on a machine and maybe be controlled (spawned by `fork()`) by one of our programs.

In numerous applications there is clearly a need for these processes to communicate with each exchanging data or control information. There are a few methods which can accomplish this task.

Interprocess Communication

Multiple processes may be running on a machine and maybe be controlled (spawned by `fork()`) by one of our programs.

In numerous applications there is clearly a need for these processes to communicate with each exchanging data or control information. There are a few methods which can accomplish this task.

We will consider some of the constructs that can be used to coordinate computations of multiple process and let them exchange data.

Piping in a C program

Piping is a process where the output of one process is made the input of another.

Piping in a C program

Piping is a process where the output of one process is made the input of another.

Pipes are used when we have two (or more) forked processes and we want to communicate between them.

Piping in a C program

Piping is a process where the output of one process is made the input of another.

Pipes are used when we have two (or more) forked processes and we want to communicate between them.

UNIX allows two ways of opening a pipe:

- formatted pipes;
- low-level pipes.

Formatted Piping

Function popen can be used to initiate pipe streams to or from a process.

Formatted Piping

Function popen can be used to initiate pipe streams to or from a process.

```
#include <stdio.h>

FILE *popen(const char *command, const char *mode);
```

Formatted Piping

Function popen can be used to initiate pipe streams to or from a process.

```
#include <stdio.h>

FILE *popen(const char *command, const char *mode);
```

The popen() function shall execute the command specified by the string command.

Formatted Piping

Function popen can be used to initiate pipe streams to or from a process.

```
#include <stdio.h>

FILE *popen(const char *command, const char *mode);
```

The popen() function shall execute the command specified by the string command.

It shall create a pipe between the calling program and the executed command, and shall return a pointer to a stream that can be used to either read from or write to the pipe.

Formatted Piping

```c
FILE *fp;
int status;
char path[PATH_MAX];

fp = popen("ls *", "r");
if (fp == NULL)
    /* Handle error */;

while (fgets(path, PATH_MAX, fp) != NULL)
    printf("%s", path);

status = pclose(fp);
if (status == -1) {
    /* Error reported by pclose() */
    ...
} else {
    ...
}
```

Low level Piping

To create a low level pipe function `pipe` can be used.

Low level Piping

To create a low level pipe function pipe can be used.

```
#include <unistd.h>

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

Low level Piping

To create a low level pipe function pipe can be used.

```
#include <unistd.h>

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

This function shall create a pipe and place two file descriptors, one each into the arguments fd[0] and fd[1], that refer to the open file descriptions for the read and write ends of the pipe.

Low level Piping

To create a low level pipe function pipe can be used.

```
#include <unistd.h>

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

This function shall create a pipe and place two file descriptors, one each into the arguments fd[0] and fd[1], that refer to the open file descriptions for the read and write ends of the pipe.

Data can be written to the file descriptor fd[1] and read from the file descriptor fd[0].

Low level Piping

```c
int pdes[2];

pipe(pdes);
if ( fork() == 0 ) {
  /* child */
  close(pdes[1]);
  read( pdes[0]); /* read from parent */
  .....
} else {
  close(pdes[0]);
  write( pdes[1]); /* write to child */
  .....
}
```

Interrupts and Signals

When a process terminates abnormally it usually tries to send a signal indicating what went wrong. C programs (and UNIX) can trap these for diagnostics. Also user specified communication can take place in this way.

Interrupts and Signals

When a process terminates abnormally it usually tries to send a signal indicating what went wrong. C programs (and UNIX) can trap these for diagnostics. Also user specified communication can take place in this way.

Signals are software generated interrupts that are sent to a process when a event happens.

Interrupts and Signals

When a process terminates abnormally it usually tries to send a signal indicating what went wrong. C programs (and UNIX) can trap these for diagnostics. Also user specified communication can take place in this way.

Signals are software generated interrupts that are sent to a process when a event happens.

Signals can be posted to a process when the system detects a software event, such as a user entering an interrupt or stop or a kill request from another process.

Interrupts and Signals

When a process terminates abnormally it usually tries to send a signal indicating what went wrong. C programs (and UNIX) can trap these for diagnostics. Also user specified communication can take place in this way.

Signals are software generated interrupts that are sent to a process when a event happens.

Signals can be posted to a process when the system detects a software event, such as a user entering an interrupt or stop or a kill request from another process.

Signals can also be come directly from the OS kernel when a hardware event such as a bus error or an illegal instruction is encountered.

Interrupts and Signals

The system defines a set of signals that can be posted to a process. Signal delivery is analogous to hardware interrupts in that a signal can be blocked from being delivered in the future.

Interrupts and Signals

The system defines a set of signals that can be posted to a process. Signal delivery is analogous to hardware interrupts in that a signal can be blocked from being delivered in the future.

Most signals cause termination of the receiving process if no action is taken by the process in response to the signal. Some signals stop the receiving process and other signals can be ignored.

Interrupts and Signals

The system defines a set of signals that can be posted to a process. Signal delivery is analogous to hardware interrupts in that a signal can be blocked from being delivered in the future.

Most signals cause termination of the receiving process if no action is taken by the process in response to the signal. Some signals stop the receiving process and other signals can be ignored.

Each signal has a default action which is one of the following:

- The signal is discarded after being received
- The process is terminated after the signal is received
- A core file is written, then the process is terminated
- Stop the process after the signal is received

Interrupts and Signals

Each signal defined by the system falls into one of five classes:

- Hardware conditions
- Software conditions
- Input/output notification
- Process control
- Resource control

Interrupts and Signals

Each signal defined by the system falls into one of five classes:

- Hardware conditions
- Software conditions
- Input/output notification
- Process control
- Resource control

Macros are defined in `signal.h` header file for common signals.

Interrupts and Signals

Each signal defined by the system falls into one of five classes:

- Hardware conditions
- Software conditions
- Input/output notification
- Process control
- Resource control

Macros are defined in signal.h header file for common signals.

Examples: SIGHUP, SIGINT, SIGQUIT, SIGILL, SIGKILL,...

Interrupts and Signals

There are two common functions used to send signals: `kill` and `raise`:

Interrupts and Signals

There are two common functions used to send signals: `kill` and `raise`:

```
#include <signal.h>

int kill(pid_t pid, int sig);

int raise(int sig);
```

Interrupts and Signals

There are two common functions used to send signals: kill and raise:

```
#include <signal.h>

int kill(pid_t pid, int sig);

int raise(int sig);
```

The kill() function shall send a signal to a process or a group of processes specified by pid.

Interrupts and Signals

There are two common functions used to send signals: kill and raise:

```
#include <signal.h>

int kill(pid_t pid, int sig);

int raise(int sig);
```

The kill() function shall send a signal to a process or a group of processes specified by pid.

If pid is greater than 0, sig shall be sent to the process whose process ID is equal to pid.

Interrupts and Signals

There are two common functions used to send signals: kill and raise:

```
#include <signal.h>

int kill(pid_t pid, int sig);

int raise(int sig);
```

The kill() function shall send a signal to a process or a group of processes specified by pid.

If pid is greater than 0, sig shall be sent to the process whose process ID is equal to pid.

If pid is 0, sig shall be sent to all processes (excluding an unspecified set of system processes) whose process group ID is equal to the process group ID of the sender.

Interrupts and Signals

There are two common functions used to send signals: kill and raise:

```
#include <signal.h>

int kill(pid_t pid, int sig);

int raise(int sig);
```

The raise() function shall send the signal sig to the executing thread or process.

Interrupts and Signals

There are two common functions used to send signals: kill and raise:

```
#include <signal.h>

int kill(pid_t pid, int sig);

int raise(int sig);
```

The raise() function shall send the signal sig to the executing thread or process.

If a signal handler is called, the raise() function shall not return until after the signal handler does.

Signal Handling

An application program can specify a function called a signal handler to be invoked when a specific signal is received.

Signal Handling

An application program can specify a function called a signal handler to be invoked when a specific signal is received.

When a signal handler is invoked on receipt of a signal, it is said to catch the signal.

Signal Handling

An application program can specify a function called a signal handler to be invoked when a specific signal is received.

When a signal handler is invoked on receipt of a signal, it is said to catch the signal.

A process can deal with a signal in one of the following ways:

- let the default action happen;
- block the signal (some signals cannot be ignored);
- catch the signal with a handler.

Signal Handling

Signal handlers usually execute on the current stack of the process.

Signal Handling

Signal handlers usually execute on the current stack of the process.

This lets the signal handler return to the point that execution was interrupted in the process.

Signal Handling

Signal handlers usually execute on the current stack of the process.

This lets the signal handler return to the point that execution was interrupted in the process.

This can be changed on a per-signal basis so that a signal handler executes on a special stack.

Signal Handling

Signal handlers usually execute on the current stack of the process.

This lets the signal handler return to the point that execution was interrupted in the process.

This can be changed on a per-signal basis so that a signal handler executes on a special stack.

If a process must resume in a different context than the interrupted one, it must restore the previous context itself

Signal Handling

Signal management is done via function signal:

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

Signal Handling

Signal management is done via function signal:

```
#include <signal.h>
```

```
void (*signal(int sig, void (*func)(int)))(int);
```

The function signal() will call the func functions if the process receives a signal sig.

Signal Handling

Signal management is done via function signal:

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

The function signal() will call the func functions if the process receives a signal sig.

Parameter func can have three values:

- SIG_DFL, a pointer to a system default function which will terminate the process upon receipt of sig.
- SIG_IGN, a pointer to system ignore function, which will disregard the sig action.
- A pointer to a user specified function.

Signal Handling

Example (Part 1)

```c
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>

void sigproc(int);

void quitproc(int);

main() {
     signal(SIGINT, sigproc);
     signal(SIGQUIT, quitproc);
     printf("ctrl-c disabled use ctrl-\\ to quit\n");
     for(;;); /* infinite loop */
}

...
```

Signal Handling

Example (Part 2)

```
...

void sigproc(int s)
{      signal(SIGINT, sigproc); /*  */
    /* NOTE some versions of UNIX will reset signal to
   default
    after each call. So for portability reset signal each
   time */

    printf("you have pressed ctrl-c \n");
}

void quitproc(int s)
{      printf("ctrl-\\ pressed to quit\n");
    exit(0); /* normal exit status */
}
```

Signal Handling

Parent-child interaction...

Let us now write a program that communicates between child and parent processes using `kill ()` and `signal ()`.

Signal Handling

Parent-child interaction. . .

Let us now write a program that communicates between child and parent processes using kill () and signal ().

fork () creates the child process from the parent.

Signal Handling

Parent-child interaction. . .

Let us now write a program that communicates between child and parent processes using `kill ()` and `signal ()`.

`fork ()` creates the child process from the parent.

The pid can be checked to decide whether it is the child $(== 0)$ or the parent $(!= 0)$.

Signal Handling

Parent-child interaction...

Let us now write a program that communicates between child and parent processes using `kill ()` and `signal ()`.

`fork ()` creates the child process from the parent.

The pid can be checked to decide whether it is the child ($== 0$) or the parent ($!= 0$).

The parent can then send messages to child using the `pid` and `kill ()`.

Signal Handling

Parent-child interaction. . .

Let us now write a program that communicates between child and parent processes using kill () and signal ().

fork () creates the child process from the parent.

The pid can be checked to decide whether it is the child ($== 0$) or the parent ($!= 0$).

The parent can then send messages to child using the pid and kill ().

The child picks up these signals with signal () and calls appropriate functions.

Signal Handling

Parent-child interaction (Part 1)

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

void sighup(int); /* routines child will call upon sigtrap */
void sigint(int);
void sigquit(int);

int main()
{
  int pid;

  /* get child process */

   if ((pid = fork()) < 0) {
        perror("fork");
        exit(1);
    }
```

Signal Handling

Parent-child interaction (Part 2)

```
if (pid == 0)
   { /* child */
     printf(”\nCHILD: Setting Signal Handlers!\n\n”);
     signal(SIGHUP,sighup); /* set function calls */
     signal(SIGINT,sigint);
     signal(SIGQUIT, sigquit);
     printf(”\nCHILD: DONE!\n\n”);
     for(;;); /* loop for ever */
   }
```

Signal Handling

Parent-child interaction (Part 3)

```
else /* parent */
     { /* pid hold id of child */
       sleep(3); /* pause for 3 secs */
       printf("\nPARENT: sending SIGHUP\n\n");
       kill(pid,SIGHUP);
       sleep(3); /* pause for 3 secs */
       printf("\nPARENT: sending SIGINT\n\n");
       kill(pid,SIGINT);
       sleep(3); /* pause for 3 secs */
       printf("\nPARENT: sending SIGQUIT\n\n");
       kill(pid,SIGQUIT);
       sleep(3);
       for(;;); /* loop for ever */
     }
}
```

Signal Handling

Parent-child interaction (Part 4)

```
void sighup(int i)
{  signal(SIGHUP,sighup); /* reset signal */
   printf("CHILD: I have received a SIGHUP\n");
}

void sigint(int i)
{  signal(SIGINT,sigint); /* reset signal */
   printf("CHILD: I have received a SIGINT\n");
}

void sigquit(int i)
{ printf("My DADDY has Killed me!!!\n");
  exit(0);
}
```

Other signal functions

Process Signal Mask: The collection of signals that are currently blocked is called the signal mask.

There are a few other functions defined in `signal.h`:

Other signal functions

Process Signal Mask: The collection of signals that are currently blocked is called the signal mask.

There are a few other functions defined in `signal.h`:

`int sighold(int sig)`: adds `sig` to the calling process's signal mask.

Other signal functions

Process Signal Mask: The collection of signals that are currently blocked is called the signal mask.
There are a few other functions defined in `signal.h`:

`int sighold(int sig)`: adds `sig` to the calling process's signal mask.

`int sigrelse(int sig)`: removes sig from the calling process's signal mask.

Other signal functions

Process Signal Mask: The collection of signals that are currently blocked is called the signal mask.
There are a few other functions defined in `signal.h`:

`int sighold(int sig)`: adds `sig` to the calling process's signal mask.

`int sigrelse(int sig)`: removes sig from the calling process's signal mask.

`int sigignore(int sig)`: sets the disposition of `sig` to SIG_IGN.

Other signal functions

Process Signal Mask: The collection of signals that are currently blocked is called the signal mask.
There are a few other functions defined in `signal.h`:

`int sighold(int sig)`: adds `sig` to the calling process's signal mask.

`int sigrelse(int sig)`: removes sig from the calling process's signal mask.

`int sigignore(int sig)`: sets the disposition of `sig` to SIG_IGN.

`int sigpause(int sig)`: removes `sig` from the calling process's signal mask and suspends the calling process until a signal is received.

Sockets

Sockets provide point-to-point, two-way communication between two processes.

Sockets

Sockets provide point-to-point, two-way communication between two processes.

Sockets are very versatile and are a basic component of interprocess and intersystem communication.

Sockets

Sockets provide point-to-point, two-way communication between two processes.

Sockets are very versatile and are a basic component of interprocess and intersystem communication.

A socket is an endpoint of communication to which a name can be bound.

Sockets

Sockets provide point-to-point, two-way communication between two processes.

Sockets are very versatile and are a basic component of interprocess and intersystem communication.

A socket is an endpoint of communication to which a name can be bound.

A socket has a type and one or more associated processes.

Socket types

Socket types define the communication properties visible to the application. Processes communicate only between sockets of the same type.

Socket types

Socket types define the communication properties visible to the application. Processes communicate only between sockets of the same type.

Stream socket: provides two-way, sequenced, reliable, and unduplicated flow of data with no record boundaries (SOCK_STREAM).

Socket types

Socket types define the communication properties visible to the application. Processes communicate only between sockets of the same type.

Stream socket: provides two-way, sequenced, reliable, and unduplicated flow of data with no record boundaries (SOCK_STREAM).

Datagram socket: supports a two-way flow of messages. A on a datagram socket may receive messages in a different order from the sequence in which the messages were sent (SOCK_DGRAM).

Socket types

Socket types define the communication properties visible to the application. Processes communicate only between sockets of the same type.

Stream socket: provides two-way, sequenced, reliable, and unduplicated flow of data with no record boundaries (SOCK_STREAM).

Datagram socket: supports a two-way flow of messages. A on a datagram socket may receive messages in a different order from the sequence in which the messages were sent (SOCK_DGRAM).

Sequential packet socket: provides a two-way, sequenced, reliable, connection, for datagrams of a fixed maximum length (SOCK_SEQPACKET).

Socket types

Socket types define the communication properties visible to the application. Processes communicate only between sockets of the same type.

Stream socket: provides two-way, sequenced, reliable, and unduplicated flow of data with no record boundaries (SOCK_STREAM).

Datagram socket: supports a two-way flow of messages. A on a datagram socket may receive messages in a different order from the sequence in which the messages were sent (SOCK_DGRAM).

Sequential packet socket: provides a two-way, sequenced, reliable, connection, for datagrams of a fixed maximum length (SOCK_SEQPACKET).No protocol for this type has been implemented for any protocol family!.

Socket types

Socket types define the communication properties visible to the application. Processes communicate only between sockets of the same type.

Stream socket: provides two-way, sequenced, reliable, and unduplicated flow of data with no record boundaries (SOCK_STREAM).

Datagram socket: supports a two-way flow of messages. A on a datagram socket may receive messages in a different order from the sequence in which the messages were sent (SOCK_DGRAM).

Sequential packet socket: provides a two-way, sequenced, reliable, connection, for datagrams of a fixed maximum length (SOCK_SEQPACKET).No protocol for this type has been implemented for any protocol family!.

Raw socket: provides access to the underlying communication protocols.

Socket Creation and Naming

To create a socket function socket can be used:

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

Socket Creation and Naming

To create a socket function socket can be used:

```
#include <sys/socket.h>

int socket(int domain, int type, int protocol);
```

The socket() function shall create an unbound socket in a communications domain, and return a file descriptor that can be used in later function calls that operate on sockets.

Socket Binding

A remote process has no way to identify a socket until an address is bound to it.

Socket Binding

A remote process has no way to identify a socket until an address is bound to it.

Communicating processes connect through addresses:

- In the UNIX domain, a connection is usually composed of one or two path names.
- In the Internet domain, a connection is composed of local and remote addresses and local and remote ports.

Socket Binding

```
#include <sys/socket.h>

int bind(int socket, const struct sockaddr *address,
      socklen_t address_len);
```

Socket Binding

```
#include <sys/socket.h>

int bind(int socket, const struct sockaddr *address,
      socklen_t address_len);
```

The bind() function shall assign a local socket address address to a socket identified by descriptor socket that has no local socket address assigned.

Socket Binding

```c
#include <sys/socket.h>

int bind(int socket, const struct sockaddr *address,
      socklen_t address_len);
```

The bind() function shall assign a local socket address address to a socket identified by descriptor socket that has no local socket address assigned.

Sockets created with the socket() function are initially unnamed; they are identified only by their address family.

Connecting Stream Sockets

Connecting sockets is usually not symmetric. One process usually acts as a server and the other process is the client.

Connecting Stream Sockets

Connecting sockets is usually not symmetric. One process usually acts as a server and the other process is the client.

The server binds its socket to a previously agreed path or address. It then blocks on the socket. For a SOCK_STREAM socket, the server calls:

```
int listen(int s, int backlog)
```

Connecting Stream Sockets

Connecting sockets is usually not symmetric. One process usually acts as a server and the other process is the client.

The server binds its socket to a previously agreed path or address. It then blocks on the socket. For a SOCK_STREAM socket, the server calls:

```
int listen(int s, int backlog)
```

A client initiates a connection to the server's socket by a call to:

```
int connect(int s, struct sockaddr *name, int namelen)
```

Stream Data Transfer and Closing

Several functions to send and receive data from a SOCK_STREAM socket.

Stream Data Transfer and Closing

Several functions to send and receive data from a SOCK_STREAM socket.

Besides read and write functions that can be used to read and write data from/to a stream, we can also use:

```
int send(int s, const char *msg, int len, int flags),
int recv(int s, char *buf, int len, int flags)
```

Stream Data Transfer and Closing

Several functions to send and receive data from a SOCK_STREAM socket.

Besides read and write functions that can be used to read and write data from/to a stream, we can also use:

```
int send(int s, const char *msg, int len, int flags),

int recv(int s, char *buf, int len, int flags)
```

These functions have some additional operational flags (see documentation).

Stream Data Transfer and Closing

Several functions to send and receive data from a SOCK_STREAM socket.

Besides read and write functions that can be used to read and write data from/to a stream, we can also use:

```
int send(int s, const char *msg, int len, int flags),
int recv(int s, char *buf, int len, int flags)
```

These functions have some additional operational flags (see documentation).

A SOCK_STREAM socket is discarded by calling close().

Datagram sockets

A datagram socket does not require that a connection be established.

Datagram sockets

A datagram socket does not require that a connection be established.

Each message carries the destination address.

Datagram sockets

A datagram socket does not require that a connection be established.

Each message carries the destination address.

If a particular local address is needed, a call to `bind()` must precede any data transfer.

Datagram sockets

A datagram socket does not require that a connection be established.

Each message carries the destination address.

If a particular local address is needed, a call to `bind()` must precede any data transfer.

Data is sent through calls to `sendto()` or `sendmsg()`.

Datagram sockets

A datagram socket does not require that a connection be established.

Each message carries the destination address.

If a particular local address is needed, a call to bind() must precede any data transfer.

Data is sent through calls to sendto() or sendmsg().

To receive datagram socket messages, call recvfrom() or recvmsg().

Datagram sockets

A datagram socket does not require that a connection be established.

Each message carries the destination address.

If a particular local address is needed, a call to bind() must precede any data transfer.

Data is sent through calls to sendto() or sendmsg().

To receive datagram socket messages, call recvfrom() or recvmsg().

Datagram sockets can also use connect() to connect the socket to a specified destination socket (send() and recv() are used).

Datagram sockets

A datagram socket does not require that a connection be established.

Each message carries the destination address.

If a particular local address is needed, a call to bind() must precede any data transfer.

Data is sent through calls to sendto() or sendmsg().

To receive datagram socket messages, call recvfrom() or recvmsg().

Datagram sockets can also use connect() to connect the socket to a specified destination socket (send() and recv() are used).

accept() and listen () are not used with datagram sockets.

Example Socket Programs

To be continued. . .