Lecture 6 — Inter-Process Communication

Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

September 4, 2022

IPC Motivation

When 2+ processes would like to co-ordinate/exchange data the mechanism is called inter-process communication.

If a process shares data with another process in the system, the operating system will provide some facilities to make this possible.

The motivations for inter-process communication are fairly obvious.

Before proceeding, we need to define some things.

It is the transfer of data from one process to another.

The data being transferred is typically referred to as the *message*.

The process sending that message is the *sender*.

The process receiving it will be the *receiver*.

This terminology may seem (painfully) obvious.

Thanks!

IPC: What to Send

The processes involved must have some agreement on:

- What data a message should contain; and
- The way the data is formatted.

There may be defined standards, e.g., XML.

The processes themselves must be aware the message is in XML format.

How this agreement is made falls outside the purview of the OS.

Messages: (A)synchronous

Sending and receiving of messages may be either synchronous or asynchronous.

Synchronous Send: the sender sends the message and then is blocked from proceeding until the message is received.

Asynchronous Send: the sender can post the message and then carry on.

Synchronous Receive: the receiver is blocked until it receives a message.

Asynchronous Receive: the receiver is notified there is no message available and continues execution.

Messages: (A)synchronous

Thus there are four combinations to consider, three of which are common:

1. **Synchronous send, synchronous receive**
2. **Synchronous send, asynchronous receive**
3. **Asynchronous send, synchronous receive**
4. **Asynchronous send, asynchronous receive**

We may also have “acknowledgement” messages.

Producer-Consumer Problem

A general paradigm for understanding IPC is known as the *producer-consumer* problem.

The producer creates some information.

The information is later used by the consumer.

Example: the database produces data to be consumed by the shell.

This is a general problem and applicable to client-server situations.

IPC Implementation Strategies

There are three approaches we will consider on how we can accomplish IPC:

1. Shared memory.
2. The file system.
3. Message passing.

All are quite common.

Shared Memory

Shared Memory

Conceptually, the idea of shared memory is very simple.

A region of memory is designated as being shared with some processes.

Those processes may read and write to that location.

To share an area of memory, the OS must be notified.

Shared Memory

Normally, a region of memory is associated with exactly one process (its owner).

That process may read and write that location.
- Other processes may not.

If a second process attempts to do so, the operating system will intervene and that will be an error.

If a process wants to designate memory as shared, it needs to tell the operating system it is okay.

Shared Memory

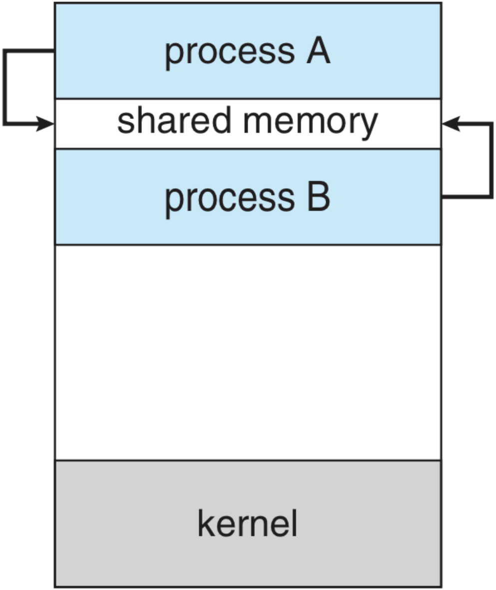
The OS needs to know that the memory is referenced by two processes.

If the first one terminates and is reaped, the memory may still be in use by the second process.

The previously-shared region should not be considered free as long as the second process is still using it.

Once the area of memory is shared, when either process attempts to access it, it is just a normal memory access.

The kernel is only involved in the setup and cleanup of that shared area.


process A
shared memory
process B
kernel

Shared Memory: Risk

When a section of memory is shared, there is the possibility that one process overwrites another's changes.

To prevent this, we need a system of co-ordination.

...A subject we will return to later.

File System

File System

Another way for 2 processes to communicate is through the file system.

Unlike shared memory, messages stored in the file system are persistent.

Can be used if the sender & receiver know nothing about one another.

File System

The producer may write to a file in an agreed upon location.

The consumer may read from that same location.

The operating system is still involved because of its role in file creation and manipulation.

File System: Co-ordination

If one file is being used then we still have the problem of co-ordination.

We can get around this, however, by using multiple files with unique IDs.

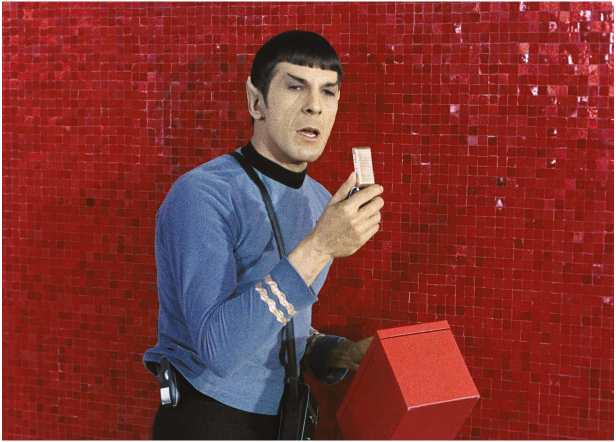
Example from a co-op work term: if the producer is generating XML data, it can write in a file in a designated `import/` directory.

The consumer program scans the directory, and imports files.

In this case, since one process writes files and another reads them, there is no possibility that one process overwrites the data of another.

If the sender chooses distinct file names, it will not overwrite a message if a second message is created before the receiver picks up the first.

Message Passing



Message Passing

Message passing is a service provided by the operating system.

The sender will give the message to the OS and ask that it be delivered to a recipient.

There are two basic operations: sending and receiving.

Messages can be of fixed or variable size.

Message Passing

Our experience with postal mail, or e-mail, suggests that to send a message successfully, the sender needs to indicate where the message should go.

Under *direct communication*, each process that wants to communicate needs to explicitly name the recipient or sender of the communication.

We have to know some identifier for the other processes; not very flexible.

Signals

Signals are interrupts with a specified ID.

Image Credit: Steven Puetzer/Getty Images

They don't really contain a message.

Signal: No Message

The fact that a signal contains no message is a limitation that means signals can't be used for every single interprocess communication scenario.

When the fire alarm sounds in a building, you don't need an accompanying voice announcement!

Why?

Signal: Preparation

You have previously been informed that when the fire alarm sounds it means you need to exit the building.

Signals: you need to know what to listen for and what's supposed to happen if you want to react accordingly.

Programmatic Signals

The appropriate header for including signals is `signal.h`.

It contains the definitions that let you write `SIGKILL` instead of having to put an explicit int 9.

Unfortunately there is not always 100% agreement between different implementations about what the higher signal numbers mean.

Programmatic Signals

There are two functions for sending a signal programmatically:

```
int kill( int pid, int signo );
int raise( int signo );
```

Both functions return 0 if they were successful and -1 if they were unsuccessful.

The `raise` function sends the signal to the current process.

Finding the Identity of a Stranger

We need to know the process ID of the recipient.

But how do processes find one another's IDs? Registration!

mysql (a database) server will put its process ID in the file `/var/run/mysqld/mysqld.pid`.

In that file is just the number of its process ID (e.g., `1494`).

Any other communication method will work!

Why not Function Overload?!

The `kill` function does different things depending on its first argument.

- `pid > 0`
- `pid == 0`
- `pid == -1`
- `pid < -1`

The Null Signal

You can also invoke the `kill` function with a `0` argument for the signal.

This is called the null signal.

It does not actually send any signal, but can be used to check if the recipient process exists.

But beware: process IDs are only relatively unique!

Did You Get My Text?!

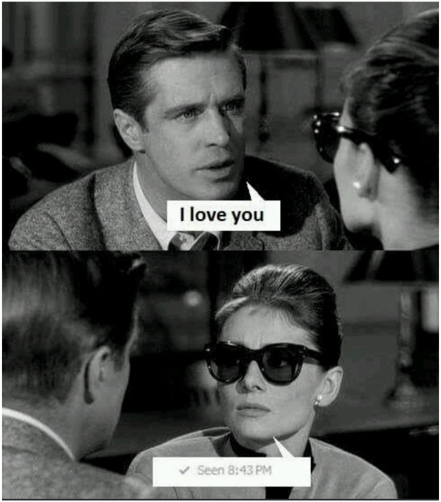
A process can only actually deal with a signal when that process is running.

A signal is generated by something, and it is later delivered to the recipient.

But during the time between generation and delivery, we say the signal is pending.

It will be delivered at the first opportunity.

Awk-ward!



Refuse to Listen

For most (but not all) signals, your process can choose to refuse to listen.

This is called blocking signals, and can be done to any with with the exception of `SIGKILL` and `SIGSTOP`.

When a signal is blocked, it just remains in the pending state until signals of that type are unblocked.

Blocking is meant to be temporary.

Signal Default Actions

Signals have a default action.

The action taken when the signal is delivered is the disposition of the signal.

If you don’t explicitly change what happens when the signal arrives, the default (see the table) happens.

But we can change it.

Option 1: Ignore it.

Option 2: Run a signal handler.

Option 3: Run the default option.

We will focus on Option 2 here.

MANY WHELPS! HANDLE IT!

If we decide to register a signal handler, the function is:

```
void (*signal( int signo, void (*handler)(int))) (int);
```

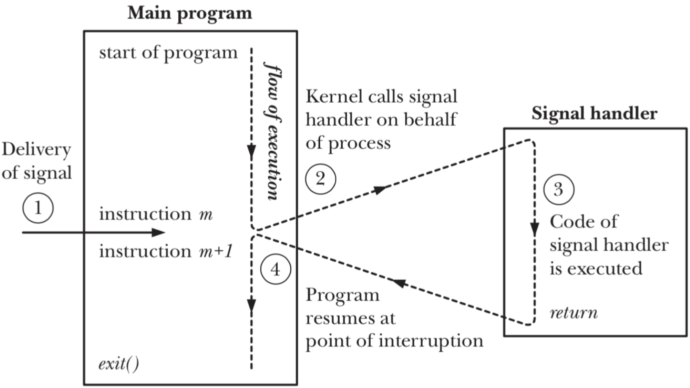
`signo`: Signal number to watch for

`handler`: Function to run to handle the signal.

So a sample signal handler would be:

```
void sig_handler( int signal_num ) {
  /* Handle the signal in some way */
}
```

Signal Handler Workflow



Tread Carefully!

The content of your signal handler, however, is restricted.

Because the handler deals with an interrupt and runs between two instructions it is important to make sure that the signal handler doesn't mess anything up.

If the signal handler runs in the middle of `malloc` and the signal handler itself calls `malloc` it could put the memory management in an invalid state!

We can only use functions that are reentrant.

Do Some Research

There are tables of what functions are safe to invoke from within a signal handler.

In general what you are looking for is a designation of async-signal safe.

To block a signal, unblock one, or just find out what the current state is, the function is:

```
int sigprocmask( int how, const sigset_t * set, sigset_t * old_set );
```

The first argument says what we're trying to do here: `SIG_BLOCK`, `SIG_UNBLOCK`, `SIG_SETMASK`.

Third argument: updated to the old values (if provided).

I am the Mask you wear...

There are some helper functions to fill in the mask:

```
int sigemptyset( sigset_t *set ); /* Initialize an empty sigset_t */
int sigaddset( sigset_t *set, int signal ); /* Add specified signal to set */
int sigfillset( sigset_t *set ); /* Add ALL signals to set */
int sigdelset( sigset_t *set, int signal ); /* Remove specified signal from set */
int sigismember( sigset_t *set, int signal ); /* Returns 1 if true, 0 if false */
```

Signal Blocking Example

```
sigset_t set;
sigset_t previous;

sigemptyset( &set ); /* Initialize set */
sigaddset( &set, SIGINT ); /* Add SIGINT to it */

sigprocmask( SIG_BLOCK, &set, &previous ); /* Add SIGINT to the mask */
/* SIGINT is blocked in this section */
sigprocmask( SIG_SETMASK, &previous, NULL ); /* Restore previous mask */
```

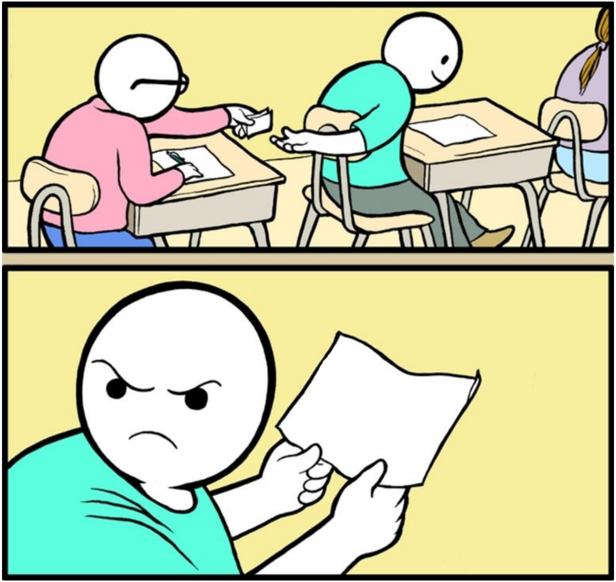
Waiting for a Page

If you want to pause your program for a bit until the call is interrupted by a signal, there is the function `int pause( )`.

This function always returns -1 and it suspends your program until the signal handler runs.

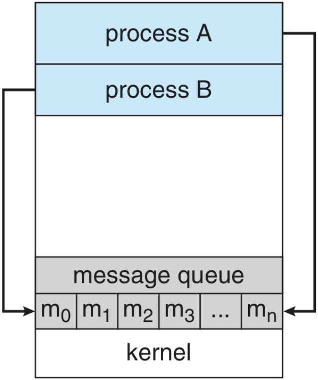
This can be useful if we really do need to wait for something...

Pass Your Message.



That... Did Not Help

To deal with the process ID problem, what we would like is indirect communication where the messages are sent to mailboxes (queues).



UNIX gives us this: message queues.

The max sizes and numbers of queues are implementation-specific.

In a real implementation, limits are something you need to consider:

1. Wait for the space to be available (block).
2. Overwrite older messages (sometimes this is what you want).
3. Discard the current message (leave the old ones as they are).

This is Key

The first step in message-passing is to get a key that identifies a specific queue.

Keys are just integer values, so we would like them to be unique (or close to it).

One method is to generate the key with the “file to key” function found in `sys/ipc.h`:

```
key_t ftok( char *pathname, int proj );
```

The file does have to exist.

Keep it Secret, Keep it Safe

Another way we can get a key is using the constant `IPC_PRIVATE`.

If we give the constant in where a `key_t` is expected then a guaranteed unique key is returned.

This method is used when there is a parent and child relationship between the processes that want to communicate.

Regardless of how we generate the key, we use it to get the queue with the function:

```
int msgget( key_t key, int flag );
```

`key`: the key we have generated or `IPC_PRIVATE`.

`flag`: UNIX permissions, optionally `IPC_CREAT` with `IPC_EXCL`.

The permissions follow the UNIX permission standards, e.g. `0600`.

`IPC_CREAT`: create queue (if does not exist).

`IPC_EXCL`: fail if trying to create and queue already exists.

Returns: the queue ID.

What does a message look like?

Unlike in a lot of other contexts, here, the message has a defined structure:

```
struct msgbuf {
    long mtype;
    char mtext[1];
};
```

Whatever message type you want to send has to have the first part be a long value; anything is fine after that.

```
struct pirate_msgbuf {
    long mtype;  /* must be positive */
    struct pirate_info {
        char name[30];
        char ship_type;
        int notoriety;
        int cruelty;
        int booty_value;
    } info;
};
```

Sending Data

```
int msgsnd( int msqid, const void *ptr, size_t nbytes, int flag );
```

`msgid`: queue ID.

`ptr`: message to send.

`nbytes`: size of the message.

`flag`: 0 for blocking, `IPC_NOWAIT` for return with error if queue full.

Receiving Data

```
ssize_t msgrcv( int msqid, void *ptr, size_t nbytes, long type, int flag );
```

`msgid`: queue ID.

`ptr`: where the message will go.

`nbytes`: size of the message.

`type`: kind of message you want.

`flag`: 0 for blocking, `IPC_NOWAIT` for return with error if queue empty.

Type Options

- `type == 0`
- `type > 0`
- `type < 0`

Destroy!

```
int msgctl( int msqid, int command, struct msqid_ds * buf );
```

`msgid`: queue ID.

`command`: `IPC_RMID` to delete the queue.

`buf`: use NULL.

This immediately deletes the queue and all messages inside!

Example of Message Passing

```c
struct msg {
  long mtype;
  int data;
};

int main( int argc, char** argv ) {
    int msgqid = msgget( IPC_PRIVATE, 0666 | IPC_CREAT );

    int pid = fork();
    if ( pid > 0 ) { /* Parent */
        struct msg m;
        m.mtype = 42;
        m.data = 252;
        msgsnd( msgqid, &m, sizeof( struct msg ), 0 );
    } else if ( pid == 0 ) { /* Child */
       struct msg m2;
       msgrcv( msgqid, &m2, sizeof( struct msg ), 42, 0 );
       printf("Received %d!\n", m2.data );
       msgctl( msgqid, IPC_RMID, NULL );
    }
    return 0;
}
```