Inter-Process Communication (IPC)

When two or more processes would like to co-ordinate or exchange data the mechanism for doing so is called *inter-process communication*, usually abbreviated as IPC. 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 and we do not need to waste time with long descriptions of scenarios where IPC is desirable. Every time you send an e-mail, for example, there's IPC going on. In the context of your own program, you could consider breaking a large task into smaller subtasks, allowing multiple users to edit the same data, and system modularity.

Before proceeding, we need to define some idea about what communication is: 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* and the process receiving it will be the *receiver*. This terminology is brutally, painfully obvious, but completeness is key.

The processes involved must have some agreement on what data a message should contain, and the way the data is formatted. Though there may be defined standards, e.g., transferring data formatted in XML, for the message content, the processes themselves would have to be aware of the fact that the message is in XML format. How this agreement is reached tends to fall outside the purview of the operating system; perhaps the authors agree in a meeting or the sender publishes its format online and the author of the receiver program writes her code to accommodate that.

Sending and receiving of messages may be either synchronous or asynchronous. If sending is synchronous, the sender sends the message and then is blocked from proceeding until the message is received. If sending is asynchronous, the sender can post the message and then carry on execution. If receiving is synchronous, the receiver is blocked until it receives the message. If receiving is asynchronous, the receiver is notified there is no message available and continues execution. Thus there are four combinations to consider, three of which are common [HZMG15]:

1. **Synchronous send, synchronous receive:** the sender is blocked until the receiver collects the message; the receiver waits for a message and is blocked until it arrives.

2. **Synchronous send, asynchronous receive:** the sender is blocked when the message is sent, but the receiver will continue whether or not a message is available. This is a very uncommon configuration.

3. **Asynchronous send, synchronous receive:** the sender continues execution when the message is sent, but the receiver will wait until the message is received before it can continue. This is the most common configuration; usually the receiver needs a message to continue.

4. **Asynchronous send, asynchronous receive:** the sender can continue as soon as it sends the message, and the receiver will check for a message but will continue whether or not the message is available.

It is also common in the case of asynchronous receive for the receiver to send back another message confirming receipt of the message. When that happens, just reverse the labels: the receiver of the initial message is the sender of the acknowledgement.

A general paradigm for understanding IPC is known as the *producer-consumer* problem. The *producer* creates some information which is later used by the *consumer*. For example, the database may produce the data (some records from the database) to be consumed by the shell to be displayed to the user. This is a general problem and applicable to client-server situations (e.g., web servers sending out web pages to web browsers).

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

1. The file system.

2. Message passing.

3. Shared memory.

All of these methods are quite common and a system can easily implement them all. There is no single option that is optimal in every situation, but each method has some areas of strength and weakness.

File System

One way for two processes to communicate is through the file system. Messages stored in the file system will be persistent and survive a reboot. It can also be used when the sender and receiver know nothing about one another (and the programmer knows nothing about any other IPC mechanisms).

The producer may write to a file in an agreed upon location and the consumer may read from that same location. The operating system is still involved because of its role in file creation and manipulation (as well as permissions for who may read and write a file).

If one file is being used then we have the problem of co-ordination: making sure one process does not overwrite the changes of another. We can get around this, however, by using multiple files with unique IDs. Consider an 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 when it finds files, reads the file and imports the data contained therein. The imported data is then shown in the program. In this case, since one process writes files and another reads them, there is no possibility that one process overwrites the data of another. As long as 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 is a service provided by the operating system where 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.

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. And in the simplest case, we will send the message directly to the recipient process. This requires us to know the ID of that process, however, which may be considered a limitation. But let's get into it.

Using Signals. We got onto the topic of inter-process communication by talking about signals, so it makes sense to start with then. Signals are, as previously introduced, an interrupt with a specified ID. They don't contain any "message" within them, so they are somewhat limited. It's a little bit like how pagers work (strange as those are to think about, doctors still use them!). When someone is paged their pager vibrates and they can see only the number who paged them. So it's like a text message, but the body is blank. Nevertheless, the poor sleep-deprived overworked medical resident is alerted and will react appropriately, dealing with the situation before going back to whatever else they were doing before being paged.

As you can imagine, the fact that a signal contains no message is a limitation that means signals can't be used for every single interprocess communication scenario. That is true, but it is sufficient for some of them. When the fire alarm sounds in a building, you don't need an accompanying voice announcement to say "this indicates a fire alarm; please calmly exit the building" – you will figure this out just fine on your own (one hopes!). Some of the reason it works out, though, is that you have previously been informed that when the fire alarm sounds it means you need to exit the building, and the same is true for signals: you need to know what to listen for and what's supposed to happen if you want to react accordingly.

The appropriate header for including signals is `signal.h` and it contains the definitions that let you write `SIGKILL` instead of having to put an explicit int `9` in your program. Earlier, we introduced a small table of some of the common UNIX signals. They are by no means exhaustive, but unfortunately there is not always 100% agreement between different implementations about what the higher signal numbers are. For this reason it's helpful to use the names, so if you use `SIGABC` then it abstracts away the fact that on one system that's implemented as being `X` and on another it's `Y`.

We already learned how to send a signal from the command line. But 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 (for whatever reason, such as no such process existing). The `raise` function sends the signal to the current process, so a nice little shortcut when you need it.

We need to know the process ID of the recipient. This is fine, but may require a little bit of negotiation for how processes find out about each other. A common convention is for services to register themselves in some way, which might be as simple as putting a file on disk in a specific location that contains the process ID. For example, mysql (a database) server will put its process ID in the file `/var/run/mysqld/mysqld.pid` and in that file is just the number of its process ID (e.g., `1494`). So if you know where to look, you can find the recipient's address. But this could also be communicated in any other way.

You can use `kill` to do some interesting things, like signal all your processes. It depends on the value chosen for your `pid` argument [SR13]:

- `pid > 0` – send the signal to the process with process ID `pid`.

- `pid == 0` – send the signal to all processes in the same process group as the caller.

- `pid == -1` – send the signal to all processes for which the calling process has permission to send a signal, except `init` and the calling process. Broadcast, basically.

- `pid < -1` – send the signal to all processes whose process group ID is equal to the absolute value of `pid`.

In addition to the signals we know, 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. If not, the `errno` return value of `ESRCH` tells you that it no longer exists [SR13]. This is of limited utility, however, because (1) the process might exit between the time you check for its existence and the time you do something with that information; and (2) process IDs are only relatively unique so a given ID could be reused for a different process, and the recipient is no longer whom you expect it to be...

A signal can be sent to a given process, but that process can only actually deal with it 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*. The pending signal is typically delivered at the first opportunity, which might be immediately if the recipient is currently executing [Ker10].

Interestingly, 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 signals is supposed to be temporary, although a badly-behaved program could ignore them indefinitely.

If the same signal is sent more than once when that signal is blocked by the recipient, it might be delivered only once, depending on your particular operating system implementation [SR13].

As introduced earlier, signals have a default action. The action that is taken when the signal is delivered is called 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. There are three options (1) ignore it, (2) run a signal handler, and (3) run the default action. The third option is used to undo an earlier change (such as if we said we wanted to ignore the signal but now no longer do). We'll restrict our examination to the "run a signal handler" option in this lecture.

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

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

Yikes! That is difficult to read. The good news is that using this in practice is a lot easier. What we actually do is call a function to say "for signal X, run function foo". There are restrictions on the function signature. It must have a void return type and take one parameter of an int. It returns a pointer to the old handler (if there was one). You could use that to set it back if you wanted.

So a sample signal handler would be:

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

The integer parameter is set to the number of the signal received. This allows us to have one signal handler that handles many signals, if we wish, but differentiate between them at the time of invocation. To register the signal handler, the simplest way is to ignore the return value of signal:

```
signal( SIGINT, sig_handler );
```

Alright, with your signal handler set up, then the workflow looks like the diagram below:
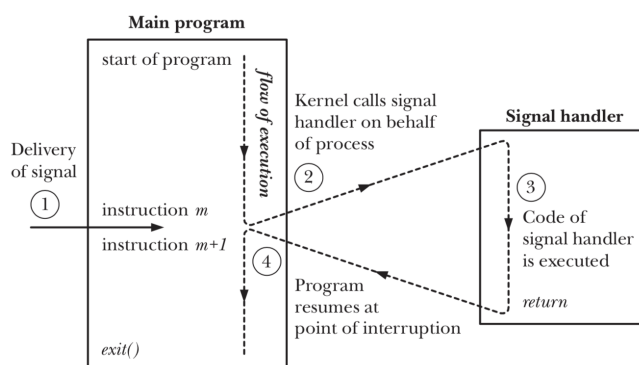
Signal delivery, signal handler execution and return [Ker10].

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 [SR13].

We can only use functions that are *reentrant*. That is to say, a function that can be interrupted during execution, have another complete call to that same function execute, and then resume (and have everything be okay). In particular the big ones to avoid are: `printf`, `malloc`, `free`, and anything that could possibly block the process (e.g., a read of a file!). The subject of reentrancy is a little bit more complicated than can be covered in a few sentences, and future courses should cover reentrancy in much more detail.

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 is what we would like to do here: if `SIG_BLOCK`, the signals pointed to by `set` are added to the block list; if `SIG_UNBLOCK` then the ones in `set` are removed from the block list; if `SIG_SETMASK` then `set` is assigned to the signal mask (overwrite all current values)[Ker10].

The third argument is optional, and if a pointer is provided then upon a change to the signal mask, `old_set` is updated to contain the values from before the change.

There is also the ability to manage signal disposition in a more advanced way using the function `sigaction`, but we will consider this beyond the scope of the course.

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 */
```

A quick example based on [Ker10]:

```
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 */
```

Finally, 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.   Earlier, it was mentioned that signals require you to know the recipient process ID, and also that they contain no message. We can now look at something that overcomes both of these limitations.

To deal with the process ID problem, what we would like is *indirect communication* where the messages are sent to mailboxes (queues). The queue is owned by the operating system, so it is persistent and independent of any particular process. The diagram below shows a message queue for communication between processes A and B:
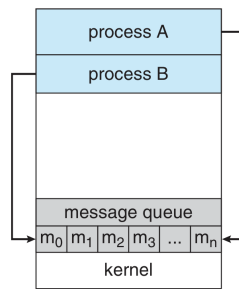
A view of memory in a message-passing system [SGG13].

UNIX is kind enough to provide us with some message queues that sort of resemble this. The total number of queues, the maximum number of messages, and the maximum size of each message is dependant on the implementation of the operating system. It is unfortunately not possible to give you an exact answer to how many messages can fit. In a real implementation, limits are something you need to consider; you have three choices:

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).

The first step in message-passing is to obtain a *key* that identifies a specific IPC structure (the queue that we will use). Keys are just integer values, so we would like them to be unique (or at least close to it).

One method is to generate the key with the "file to key" function found in `sys/ipc.h` [Ker10]:

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

The key is generated from the given file name (`pathname`) and the value `proj`. The file does have to exist, because the function uses its inode (file structure on disk that contains the metadata). And the integer argument allows generating multiple IPC objects based off the same file. There is a very small risk of duplicate numbers if we are unlucky, but the risk is small enough that we consider it acceptable.

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 );
```

The first parameter is the key that we have previously generated and is straightforward. The `flag` parameter starts with the UNIX permissions and can be modified with additional creation options. The permissions follow the UNIX permission standards, e.g. `0600`.If the queue is being created for the first time, use `IPC_CREAT`. If you want also be sure that this is being created anew, use bitwise-OR to combine `IPC_CREAT` with `IPC_EXCL` so that the call will fail if the queue already exists.

The return value is the queue ID of the queue we will use. Then we can send and receive messages. Alright, we're getting somewhere. But 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];
};
```

Wait... Does this mean a message can be only one character at a time? No, all it really means is that whatever message type you want to send has to have the first part be a long value; anything is fine after that. Such as [Hal15]:

```
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;
};
```

Right, so let's say that we want to send some data then:

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

The first parameter is the queue we want to send it to; the second is the message; the third is the data that we want to send. The last parameter is for what happens if the queue is full: normally we just want to wait, so this would be a blocking send, and you can provide 0 as the argument. But if you would prefer, you can use IPC_NOWAIT, in which case an attempt to add to the full queue will just return with an error instead of blocking.

And to receive:

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

The first argument is the queue to receive from as per usual; the second argument is the destination where the message will be copied to; the third argument is the number of bytes. The last parameter, again, can be used to specify that you don't want to wait if there is no message. The fourth parameter, `type`, is used to specify what kind of message you want. It corresponds to the `mtype` field in the message [SR13]:

- `type == 0` – Return the first message on the queue (any type)

- `type > 0` – Return the first message on the queue with the specified type

- `type < 0` – Return the first message on the queue whose type is the smallest value less than or equal to the absolute value of `type`.

When we are finished, we clean up the queue using [SR13]:

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

To clean it up, the first argument is the queue ID we got back from `msgget` and the command we provide is `IPC_RMID` and we can give `NULL` as the last parameter. This deletes the queue as well as any data in the queue immediately. This function can do a number of other things, but we will consider the advanced configuration beyond the scope of this course.

Let's put it together in a simple example where the parent process sends a message to the child process:

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/types.h>
#include <sys/msg.h>
#include <unistd.h>

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

```c
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;
}
```

References

[Hal15]   Brian "Beej Jorgensen" Hall. Beej's guide to unix ipc, 2015. Online; accessed 21-August-2019. URL: https://beej.us/guide/bgipc/html/multi/index.html.

[HZMG15] Douglas Wilhelm Harder, Jeff Zarnett, Vajih Montaghami, and Allyson Giannikouris. *A Practical Introduction to Real-Time Systems for Undergraduate Engineering*. 2015. Online; version 0.15.08.17.

[Ker10]   Michael Kerrisk. *The Linux Programming Interface*. No Starch Press Inc, 2010.

[SGG13]   Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts (9th Edition)*. John Wiley & Sons, 2013.

[SR13]   W. Richard Stevens and Stephen A. Rago. *Advanced Programming in the UNIX Environment, Third Edition*. Addison-Wesley, 2013.