Programming -- UNIX and POSIX Inter-process Communication (Signals, Pipes & Message Passing)

A/Prof Jian Zhang
School of Computing and Communication
University of Technology, Sydney

Signals

Signals under Unix and POSIX

Signals

- *Signals are not to be confused with the `signal()` system call applied to semaphores!*
- A signal is the software analogy of a hardware generated interrupt

  sources external to the receiving process and arrive asynchronously to be dealt with by special handling functions
- Signals are used for ...
  - Exception handling (divide by zero, page fault etc.)
  - Process notification of asynchronous events (I/O completion, timer expiry etc.)
  - Process termination in abnormal circumstances
  - Inter-process communication (in a limited sense)
  - To emulate multitasking (in a very clumsy and limited sense)

There are POSIX.1 Signals and POSIX.4 compliant Signals

- POSIX compliant signals
  - manipulate signal set -- data structures: `sigset_t`

```
int sigemptyset(sigset_t, *set); //set Empty
int sigfillset(sigset_t, *set);  // set Full
int sigaddset(sigset_t, *set, int set); // add & del signal +
int sigdelset(sigset_t, *set, int sig); //signum from set
int sigismember(sigset_t, *set, int sig); // test if signum is
  a member of set return --  1: a member, 0: not a member and
  -1: error
```

  - set the processes signal blockage mask

```
int sigprocmask(int op, const sigset_t *set, sigset_t
  *oldset); //examine & change blocked signals
int sigaction(int op,const struct sigaction *sa,
  struct sigaction *old_response); // examine & change
  a signal action
```

POSIX.1 and POSIX.4 compliant Signals (Cont)

- Wait for a signal to arrive, setting the given mask

```
int sigsuspend(const sigset_t *new_mask);
```

- **Send a signal to a process** (terminate a process)

```
int kill(pid_t victim_id, int this_signal);
```

– The remainder of the functions are part of the POSIX.4 standard and conditional on **_POSIX_REALTIME_SIGNALS**

```
int sigqueue(pid_t victim_id, int this_signal, union sigval
  extra_info);// Queue a signal and data to a process
int sigwaitinfo(const sigset_t *one_of_these_signals, siginfo_t
  *addl_info); // synchronously wait for a queued signals
 int sigtimedwait(const sigset_t *one_of_these_signals,
  siginfo_t *addl_info, const struct timespec *timeout); // same
  as sigwaitinfo with timeout argument
```

Signal Management

- A signal is the simplest form of inter-process communication

- UNIXes have 32 signals and POSIX has 64

- A signal is like a software interrupt

- A signal is an asynchronous notification sent to a process or to a specific thread within the same process in order to notify it of an event that occurred

- The kernel uses a signal in the case of an exception

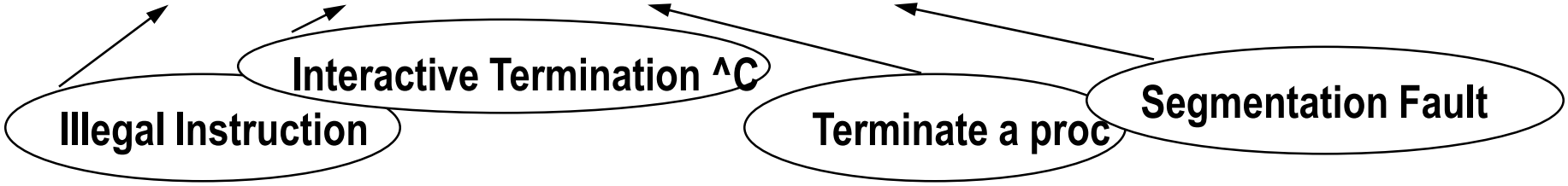
- A process may choose to block or ignore given signals

- A blocked signal is queued until further use. An ignored signal is irretrievably lost

Signal Management (Cont)

- A process may define a special routine (handler) for each of the signals it needs to manage
- Upon signal receipt, the process will execute the whole handler before resuming normal execution
- If a signal is neither blocked, nor ignored, nor handled, it is processed by a default routine, which can ignore the signal or halt the process, with or without a *core dump*
- Signal name macros are defined in `<signal.h>`

  `SIGILL`, `SIGINT`, `SIGKILL`, `SIGSEGV`, etc. etc.



From a perspective of robustness, a real time embedded application needs to process ALL possible signals.

Signal Management (Cont)

The **SIG** macros are used to represent a signal number (**signum**) in the following conditions

S.N.	Macro & Description
1	**SIGABRT**Abnormal program termination.
2	**SIGFPE**Floating-point error like division by zero.
3	**SIGILL**Illegal operation.
4	**SIGINT**Interrupt signal such as ctrl-C.
5	**SIGSEGV**Invalid access to storage like segment violation.
6	**SIGTERM**Termination request. and more!

Signal Sets and Masks

Example: Initialise set `twosigs` to contain `SIGINT` and `SIGUSR1`

```
#include <signal.h>
sigset_t twosigs;

sigemptyset(&twosigs);
sigaddset(&twosigs,SIGINT);// add signum "SIGINT" to
twosigs to set up interrupt
sigaddset(&twosigs,SIGUSR1); add signum "SIGUSR1" indicate
user-defined condition
```

- Each process has his own signal mask

- The `sigprocmask()` function changes the process mask according to 3 different behaviours ... (POSIX ONLY)

`int sigprocmask(int` *`how`*`, const sigset_t *`*`set`*`, sigset_t *`*`oldset`*`)`

1. `SIG_BLOCK`: add signal(s) to those currently unblocked (union of current and new!)
2. `SIG_UNBLOCK`: delete specified signal(s) from those currently blocked
3. `SIG_SETMASK`: set the collection of blocked signals to those given (overwrite!)

- If the *`oldset`* is non-NULL, the previous value of the signal mask is stored in *`oldest`*

Signal Masks (Cont)

Syntax:

```
#include <signal.h>

int sigprocmask(int how, sigset_t *set, sigset_t *oset);
int sigaddset(sigset_t *set, int sig);
int sigemptyset(sigset_t *set);
int sigaction(int sig, const struct sigaction * acti, struct sigaction
*oact);
…..
```

int sigaction(int *signum*, **const struct sigaction *** *act*, **struct sigaction *** *oldact*);

- The sigaction() system call is used to change the action taken by a process on receipt of a specific signal.
- *signum* specifies the signal and can be any valid signal except **SIGKILL** and **SIGSTOP.**
- If *act* is non-NULL, the new action for signal *signum* is installed from *act*. If *oldact* is non-NULL, the previous action is saved in *oldact*.

The *sigaction* structure is defined as something like:

```
struct sigaction {
void     (*sa_handler)(int);
void     (*sa_sigaction)(int, siginfo_t *, void *);
sigset_t   sa_mask;
int        sa_flags;
void     (*sa_restorer)(void);        };
```

Sending Signals

• A signal is sent to a process by using `kill()` function. It requires two arguments, whom (receiver process ID) and what (signal identification)
If what == 0, the receiver's PID is checked but no signal is sent
If whom > 0, the what signal is sent to the whom process
If whom < 0, the signal is sent to all processes belonging to the group process whose group ID == |whom|

• The `raise()` call permits a process to send a signal to itself (ANSI C)

Syntax:

```
#include <sys/types.h>
#include <signal.h>
```

Example:

```
int kill(pid_t whom, int what);

pid_t pid;
/* terminate a process with core dump */
    if (kill(pid, SIGQUIT) == -1)
    {// https://en.wikipedia.org/wiki/Unix_signal
        perror("kill");
        exit(EXIT_FAILURE);
        }
```

Receiving Signals

• A process may associate a *handler* with a signal. The kernel must be informed of this association using the `sigaction()` system call

• The `sigaction()` function allows us to associate a given routine with a particular signal, and to (optionally) save the previously associated routine

• Two default handlers, `SIG_IGN` and `SIG_DFL` allow us to respectively ignore a signal, or to call a default handler

• A handler is executed as soon as signal is received. The signal number is available as a handler parameter. When the handler reaches it's end, the interrupted process resumes

Syntax:

```
#include <signal.h>
int sigaction (int signum, const struct sigaction *act, struct
sigaction *oact);
signum specifies the signal and can be any valid signal except
SIGKILL and SIGSTOP.
```

Example: Set up a handler for `SIGINT`, which is usually `^C`

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

char msg[] = "I caught ^C\n";
void ControlCHandler(int signum)
{
    printf("%s", msg); return;
}
...
struct sigaction  act, oact;
...
act.sa_handler = ControlCHandler;
sigemptyset(&act.sa_mask);
act.sa_flags = 0;
...
if (sigaction (SIGINT, &act, &oact) < 0) {
    /* Attend to any handler registration error here */
}
...
do_my_processing();    /* The handler is active now */
...
if (sigaction (SIGINT, &oact, NULL) < 0) { /* Restore previous handler */
    /* Attend to any handler removal error here */
}
```

Blocking Signals

- **sigprocmask**() is used to fetch and/or change the signal mask of the calling thread. The signal mask is the set of signals whose delivery is currently blocked for the caller.

- The behavior of the call is dependent on the value of *how*, as follows.
  - **SIG_BLOCK** The set of blocked signals is the union of the current set and the *set* argument.
  - **SIG_UNBLOCK** The signals in *set* are removed from the current set of blocked signals. It is permissible to attempt to unblock a signal which is not blocked.
  - **SIG_SETMASK** The set of blocked signals is set to the argument *set*.

- **Blocking a signal results in it being held until the mask indicates that it is no longer blocked at which point it is delivered to the process. Ignoring a signal means that it is permanently discarded.**

- A process can use the `sigprocmask()` call, with `SIG_BLOCK` as first parameter to block specific signals

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

- Blocked signals are queued up and will be delivered when unblocked

- A process can examine the blocked signal queue using the `sigpending()` function. This function is POSIX only.

Syntax:

```
#include <signal.h>

int sigpending (sigset_t *set);
```

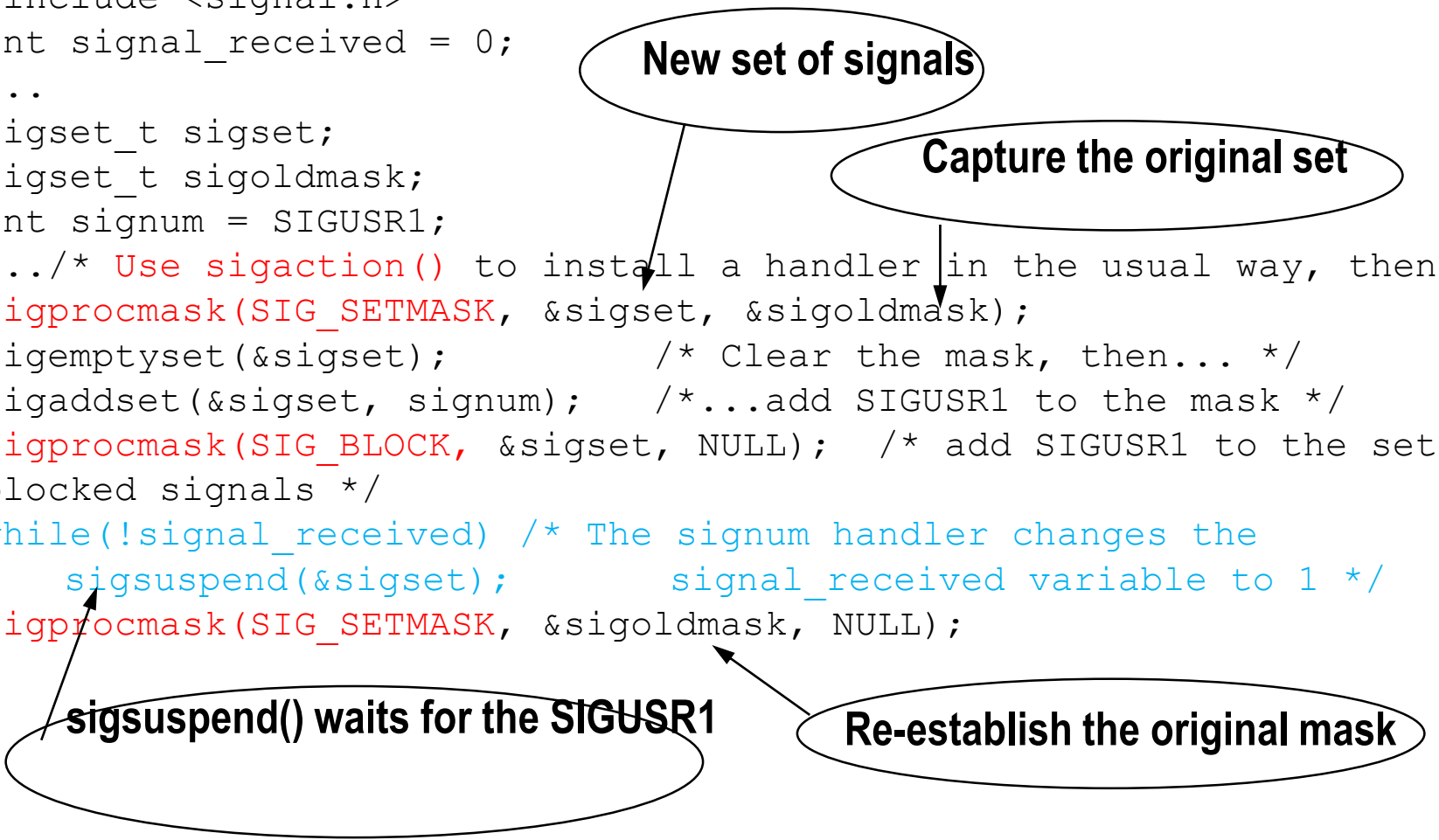
Waiting for a Signal

- Synchronous wait for one or more signals is performed by `sigsuspend()`

Temporarily replaces the signal mask of the calling process with the mask given by *mask* and then suspends the process until delivery of a signal whose action is to invoke a signal handler or to terminate a process

Example: Suspend a process until `signum` occurs and then restore the original signal mask

```
#include <signal.h>
int signal_received = 0;
...
sigset_t sigset;
sigset_t sigoldmask;
int signum = SIGUSR1;
.../* Use sigaction() to install a handler in the usual way, then... */
sigprocmask(SIG_SETMASK, &sigset, &sigoldmask);
sigemptyset(&sigset);          /* Clear the mask, then... */
sigaddset(&sigset, signum);    /*...add SIGUSR1 to the mask */
sigprocmask(SIG_BLOCK, &sigset, NULL);  /* add SIGUSR1 to the set of
blocked signals */
while(!signal_received) /* The signum handler changes the
    sigsuspend(&sigset);        signal_received variable to 1 */
sigprocmask(SIG_SETMASK, &sigoldmask, NULL);
```



Alarms Management

- UNIX offers the ability to set counters which, when reach a given value, will send a `SIGALRM` signal to initiating process
- The counter is set by an `alarm()` function associated with a parameter representative of the specified time (in seconds)
- The default handler associated to `SIGALRM` is to terminate the process!
- A proper `sigaction()` call must be performed *before* calling `alarm()` in order to associate the requested handler if termination isn't wanted
- Other POSIX functions are designed to obtain a greater timer precision
  - `setitimer()`
  - `getittimer()`
- See also the functions `usleep()`, `nanosleep()`…

Example: This program runs for ten seconds of real time

```
#include <unistd.h>

void main(void)
{
    alarm(10);
    while(1);
}
```

If Alarm goes off -- The special signal SIGALRM is sent to the receiving process.

Problems with Signals

Problems with POSIX.1 Signals are numerous ...

- Lack of signals for application use - `SIGUSR1` and `SIGUSR2` alone aren't sufficient
- No signal queuing is provided, so signals can get lost
- No signal delivery order is guaranteed
- Poor information content - signals are only 32-bit numbers
- Signals are slow because of the administration of handlers, masks and system calls

http://cs.bc.edu/~donaldja/362/Signals.html

```
SIGUSR1, SIGUSR2 user-defined signal
```

POSIX.4 tries to counter these problems with …

- More signals being available. See `SIGRTMIN` and `SIGRTMAX` in <signal.h> and `RT_SIGMAX` in `<limits.h>`
- Signal queuing
- Parameters can be passed to handlers - more information transfer

Questions on POIX programming part 2

10.Write down two (2) of the more appropriate uses of POSIX.1 signals.
Discussion: Exception handling, asynchronous event notification (particularly in respect of interaction with I/O devices)
11. Write down three deficiencies of POSIX.1 signals that POSIX.4 attempts to fix.
Discussion: Any three of: only a limited number of POSIX.1 signals, they cannot be queued, non-prioritorised, cannot carry additional data, slow in comparison, can't identify the sender (anonymous).

Questions on POIX programming part 2

12. “Signals are like a software interrupt” – explain what this means.

Discussion:

They are generated by sources external to the receiving process and arrive asynchronously to be dealt with by special handling functions.

13. What happens when an alarm (set by the **alarm()** function) goes off?

Discussion:

The special signal SIGALRM is sent to the receiving process.

14. What is a signal handler? Why is it necessary to set one up for each expected signal?

Discussion: It’s a function which is automatically invoked as the signal with which it’s associated arrives at the receiving process. The default behaviour is to terminate the receiving process, if this is not desired then a handler must be implemented to override the default behavior.

Questions on POIX programming part 2

15. What is the **kill()** function used to do?

Discussion: Send signals – not necessarily to result in processes being killed.

16. How do you send a signal to a process?

Discussion: You use the **kill()** function (either interactively from a shell, or as a system call from a program).

17. What does **sigsuspend()** do?

Discussion: Temporarily replaces the current signal mask with a mask that indicates signals that a process is to receive and then suspends the process until one of the said signals arrives.

Questions on POIX programming part 2

18. What is (most commonly) the result of sending a signal to a process that hasn't been set up explicitly to deal with it?

Discussion: The default behaviour is to terminate the process – with or without a core dump.

19. Explain the difference between blocking a signal and ignoring a signal

Discussion: Blocking a signal results in it being held until the mask indicates that it is no longer blocked at which point it is delivered to the process. Ignoring a signal means that it is permanently discarded.

Pipes, FIFOs and Message Queues

Improved forms of interprocess communication

Inter-process Communication

- Limitations of Signals …
  - Minimal information contained within them.
  - Asynchronous nature ⇒ indeterminate speed.

A comparison of the mechanisms:

Name	Reliable ?	Flexible ?	Fast ?
Signals	Sometimes	Very	No
Messages	Yes	Not very	Not particularly
Semaphores	Yes	Very	Very (+shared memory)
Shared Memory	Yes	Very	Very (+semaphores)

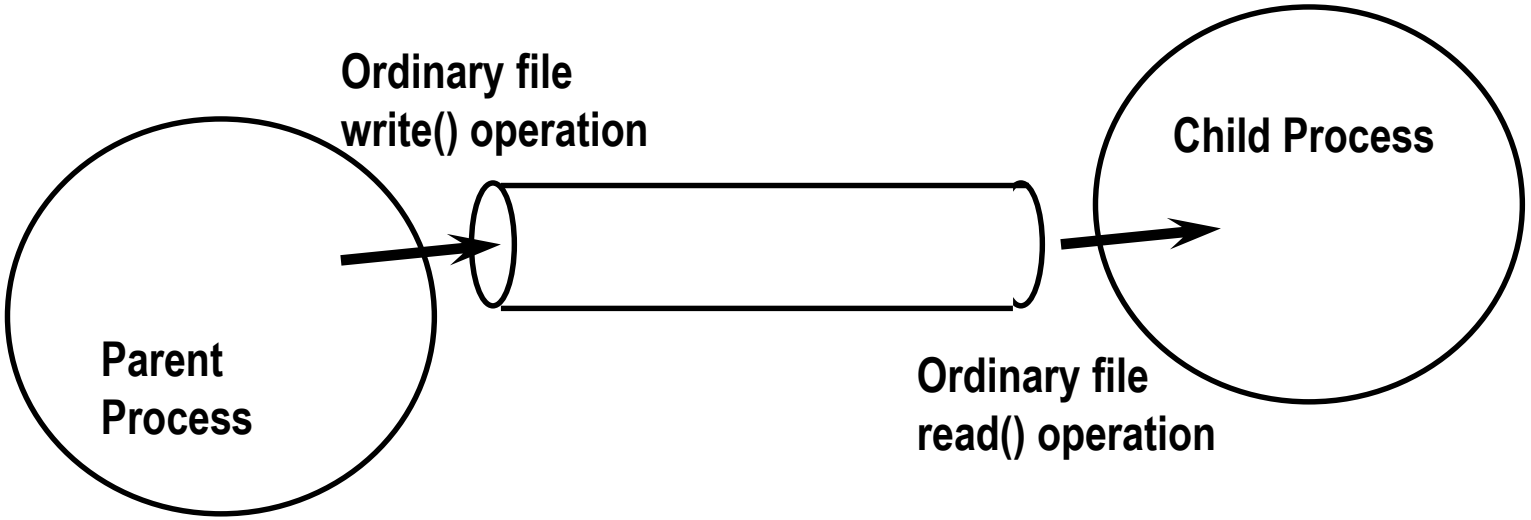
Comparing Signals and Messages

- Signals
  - Convenient for small amounts of information
  - Interface contrasts with other communications mechanisms
  - More suited to asynchronous event notification, such as for I/O operations

- Message-passing Approaches
  - More general and abstract
  - More reflective of the way humans converse
  - Pipes and FIFOs (POSIX.1)
  - Various forms of message queues (POSIX.4)

Pipes and FIFOs - POSIX.1

- Pipes
  - Specifically included in the API to support information transfer
  - Compared to Signals ...
    - No asynchrony (unless specifically requested)
    - No handler functions
    - No masks
  - Queue data internally - the kernel safeguards data in transit
  - Possess some deficiencies that FIFOs attempt to correct



– Points to note about using pipes … *****

- call `int pipe(int pipefd[2])` to create the pipe itself
- give the `pipe()` call an integer array in which the two file descriptors can be stored
- we must read from the first file descriptor - `array[0]` and write to the second file descriptor `array[1]`
- file descriptors with *numbers* `0`, `1` and `2` are normally reserved for `stdin`, `stdout` and `stderr`
- call `fork()` to permit parent and child to have access to the pipe
- treat the pipe as a logical file, using `read()` and `write()` functions
- `read()` will *block* by default until data appears in the pipe - however, the standard file control function `fcntl()` can be used to set the `O_NONBLOCK` flag. This will cause the recipient to fail (rather than block) with an `errno` of `EAGAIN`
- if we `read()` from a pipe whose writing end has been `close()`ed, then `read()` returns `0` to signify end-of-file
- if we `write()` to a pipe that has had its `read()`ing end `close()`ed, the signal `SIGPIPE` is generated for calling a process (e.g. exit). If the calling process is ignoring this signal. The `write()` fails with `EPIPE`
- http://man7.org/linux/man-pages/man7/pipe.7.html

- Limitations of Pipes
  - Pipes are uni-directional - so we need to create two of them if we want bi-directional transfer
  - Pipes require that they be created and set-up by the parent before forking to a child. This hierarchical relationship might not suit some system designs
  - They don't extend to processes on different nodes in an networked environment
  - The file descriptors pertaining to the pipe must be inherited by the child process, after the parent creates the pipe.

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

#define MESSLENGTH 80

char pourIntoPipe[] = "This has been through the pipe!\n";
char collectFromPipe[MESSLENGTH];

int main(void)
{
    int   n, fd[2];// file descriptors: fd[0]- read file, fd[1]- write file
    pid_t pid;

    if ( pipe(fd) < 0 )
      perror("pipe error");
    if ( (pid = fork()) < 0 )
      perror("fork error");
    else if ( pid > 0 )
    {                       /* parent will do the writing this time */
      close(fd[0]);         /* we don't want the reading end at all */
      write(fd[1], pourIntoPipe, strlen(pourIntoPipe));
    }  else {
      close(fd[1]);         /* child will do the reading. */
      n = read(fd[0], collectFromPipe, MESSLENGTH);
      printf("%s", collectFromPipe);
    }
    exit(0);
}
```

- **FIFOs - otherwise known as 'Named Pipes'**
  - A pipe that has a name is a FIFO! -- first-in first-out special file
    - The name of the pipe is handled by the file system
    - Any process that possesses appropriate permissions can get access to this type of pipe
  - Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa.
  - FIFOs cannot be opened for both reading and writing by the same process - POSIX says the results would then be undefined
  - How do we create FIFOs under POSIX.1 ?

```
#include        <sys/types.h>
#include        <sys/stat.h>
int  mkfifo (const char *fifo_name, mode_t mode);
```

permission bits

name

older non-POSIX systems use the mknod() call

- Permission bits are defined in `<sys/stat.h>`, some examples are ...
  - `S_IRUSR` - read permission for processes with the same user id
  - `S_IROTH` - read permission for others
  - `S_IWGRP` - write permission for other processes in the same group
  - `S_IXOTH` - execute permission for other processes

http://pubs.opengroup.org/onlinepubs/007908799/xsh/sysstat.h.html

- The special macro `PIPE_BUF` specifies the pipe capacity - it's also the **atomic** unit for writing/reading operations. Writing less than or equal to this amount guarantees that multiple processes won't mix their data in a shared pipe
- The `unlink()` call should be used to dispose of a FIFO

***Atomic** = non-interruptible

- **Check point! --Pipes require that they be created and set-up by the parent before forking to a child. While FIFO is accessible through the file systems**
- Some limitations of both pipes and FIFOs
  - No prioritisation
    - Messages pass through the pipe in the order in which they entered - no provision is made for an important message to "overtake"
  - Capacity
    - A limited (and *unknown*) amount of kernel-space memory is dedicated to implementing the pipe/FIFO itself. If asynchronous piping is being used (via `O_NONBLOCK`). In this case, the sender *will* be blocked until the recipient removes some of the messages - this can lead to unexpected behaviour
  - No control over structure
    - There's no way to influence pipe length
  - Limited numbers of file descriptors are available

Questions on POIX programming part 2

20. When a standard UNIX pipe is used to transfer information between a parent and a child process, the pipe must be created before the **fork()** system call – explain why this is the case, and also explain why POSIX FIFOs don't have this requirement.

Discussion: The file descriptors pertaining to the pipe must be inherited by the child process, after the parent creates the pipe. Named pipes (FIFOs) don't have this requirement since they're accessed through the file system – their name makes them accessible.

21. Which two system calls are used to put information into pipes and to read information from them?

Disucssion: The filesystem **read()**and **write()**calls..

22. Can traditional UNIX pipes be used to transfer information between processes that are on different machines in a network? What about named pipes?

Discussion: Pipes certainly don't have the ability to transfer over a network and FIFOs don't either (unless the file system is visible over a network using something like NFS).

Message Queues - POSIX.4

- Message Queues attempt to solve the problems (No prioritization!) inherent in Pipes and FIFOs

- They operate on much the same essential principle - a named object is created permitting readers and writers to exchange

- Message Queues possess structure
  - The queue is priority-based, in consideration of priorities attached to individual messages.
- They have the ability to prioritise messages and to specify the size of the message type as well as the capacity of the messaging channel

23. What extra features do POSIX.4 message queues offer over both pipes and FIFOs?

Discussion: They have the ability to prioritise messages and to specify the size of the message type as well as the capacity of the messaging channel.

- Is it supported ?
  - Check for the presence of the constant `_POSIX_MESSAGE_PASSING` in `<unistd.h>`

- A separate class of functions are present that deal with the operations, some examples are ...

mq_open(**onst char ***_name_**, int** _oflag_**, mode_t** _mode_**, struct mq_attr ***_attr_)

mq_open creates a new POSIX message queue or opens an existing queue. The queue is identified by _name_. The _oflag_ argument specifies flags that control the operation of the call. The _create_mode_ is the permission bits and The _attr_ argument specifies attributes for the queue.

Oflag:

O_RDONLY:Open the queue to receive message only

O_WRONLY:Open the queue to send message only

O_RDWR:Open the queue to both send and receive message

. . . .

```
int mq_unlink(const char *mq_name);// Remove a message
   queue.
```

- Unlike pipes and FIFOs, standard file system operations like `open()`, `read()`, `write()` are *not* used. A Separate API implemented as a library is offered.
  - This is potentially faster than a collection of file system calls

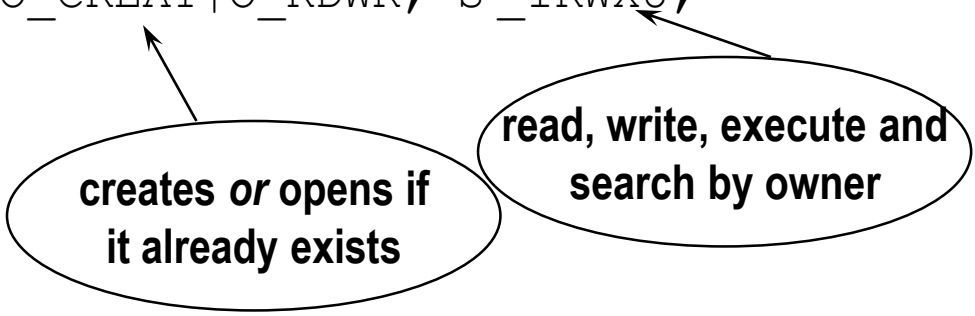
- Checking for message queue support …

```
#include          <unistd.h>
#ifdef _POSIX_MESSAGE_PASSING
#include          <mqueue.h>
```

```
/* Create an example message queue */
struct mq_attr       attributes;
mqd_t                mymq;

attributes.mq_maxmsg = 100;   /* number of messages     */
attributes.mq_msgsize = 128;  /* message size (bytes)   */
attributes.mq_flags = 0;      /* or e.g. O_NONBLOCK     */

mymq = mq_open("/queue_name", O_CREAT|O_RDWR, S_IRWXU,
                &attributes);
if (mymq == (mqd_t)-1)
      perror("mq_open");
```



- Extra flexibility and information is available to the processes using messages to communicate that isn't present when using Pipes and FIFOs

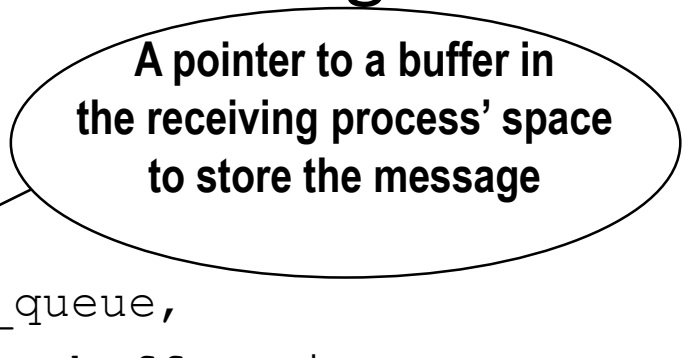
- Send a message to a message queue

```
#include    <mqueue.h>

int    mq_send(mqd_t message_queue, const char
                *message_data, size_t message_length,
                 unsigned int priority);
```

  - adds the message pointed to by *msg_ptr* to the message queue referred to by the message queue
  - If the *message_length* parameter is larger than *mq_attr.mq_msgsize*, then mq_send will fail
  - The *msg_prio* argument is a nonnegative integer that specifies the priority of this message. Messages are placed on the queue in decreasing order of priority, with newer messages of the same priority being placed after older messages with the same priority.

- Receiving a message from a message queue




```
#include      <mqueue.h>
size_t  mq_receive(mqd_t message_queue,
const char *message_buffer, size_t buffer_size,
unsigned int *priority);
```

`mq_receive()` removes the oldest message with the highest priority from the message queue referred to by the descriptor *`message_queue,`* and places it in the buffer pointed to by *`*priority`*

- If the queue is empty, then, by default, `mq_receive()` blocks until a message becomes available, or the call is interrupted by a signal handler unless the `O_NONBLOCK` flag is enable.
- If the receiving buffer hasn't the capacity to hold the message, an error is generated and a message will not be removed from the queue

Signal handler: It's a function which is automatically invoked as the signal with which it's associated arrives at the receiving process. The default behaviour is to terminate the receiving process, if this is not desired then a handler must be implemented to override the default behavior

- Points to remember on using queues
  - Cleaning up after use: *Queues might not exist in file-system namespace - so they might be left unattended after the using processes have terminated. Also, any messages enclosed will remain there!*
  - Always create them: *Queues are not persistent as files are, they don't survive abnormal terminations*
  - Program defensively: *Verify that the operations completed without returning error codes*

- Drawback with Message Queues
  - POSIX.4 requires that the actual message data be copied from sending process to kernel space, and from there again copied to the recipient. *This naturally incurs overhead!*