IPC - Inter Process Communication

1.	**1. what is Signal ? how to use signals to communicate between processes ?** **2. is a signal also an interrupt ?**	1. Signals can be viewed as a mean of communication between the OS kernel and processes. to communicate between processes we need to use the kill function (probably communicates the processes via the OS). 2. no. because no communication with cpu is involved.

2. **give at least 5 examples for IPC methods**

1. Pipes (use pipe before the fork)
2. Signals - with the use of the kill() syscall.
3. Message Queues - consumer producer, shared queue created in main and the passed to both.
4. memory mapped file - mmap()
5. Shared Memory (posix, sys V - mapping of two process virtual addresses to same address)
6. Sockets

----
http://stackoverflow.com/questions/404604/comparing-unix-linux-ipc
----
Here are the big seven:

1. Pipe

Useful only among processes related as parent/child. Call pipe(2) and fork(2). Unidirectional.

2. FIFO, or named pipe

Two unrelated processes can use FIFO unlike plain pipe. Call mkfifo(3). Unidirectional.

3. Socket and Unix Domain Socket

Bidirectional. Meant for network communication, but can be used locally too. Can be used for different protocol. There's no message boundary for TCP. Call socket(2).

4. Message Queue

OS maintains discrete message. See sys/msg.h.

5. Signal

Signal sends an integer to another process. Doesn't mesh well with multi-threads. Call kill(2).

6. Semaphore

A synchronization mechanism for multi processes or threads, similar to a queue of people waiting for bathroom. See sys/sem.h.

7. Shared memory

Do your own concurrency control. Call shmget(2).
Message Boundary issue

One determining factor when choosing one method over the other is the message boundary issue. You may expect "messages" to be discrete from each other, but it's not for byte streams like TCP or Pipe.

Consider a pair of echo client and server. The client sends string, the server receives it and sends it right back. Suppose the client sends "Hello", "Hello", and "How about an answer?".

With byte stream protocols, the server can receive as "Hell", "oHelloHow", and " about an answer?"; or more realistically "HelloHelloHow about an answer?". The server has no clue where the message boundary is.

An age old trick is to limit the message length to CHAR_MAX or UINT_MAX and agree to send the message length first in char or uint. So, if you are at the receiving side, you have to read the message length first. This also implies that only one thread should be doing the message reading at a time.

With discrete protocols like UDP or message queues, you don't have to worry about this issue, but programmatically byte streams are easier to deal with because they behave like files and stdin/out.

3. **int open(const char *path, int oflag, ...);**

   The open() function shall establish the connection between a file and a file descriptor. It shall create an open file description that refers to a file and a file descriptor that refers to that open file description. The file descriptor is used by other I/O functions to refer to that file. The path argument points to a pathname naming the file.

   O_RDWR - Open for reading and writing.
   example 1.6.16

4. **int pthread_cancel(pthread_t thread);**

   The pthread_cancel() function sends a cancellation request to the thread thread. Whether and when the target thread reacts to the cancellation request depends on two attributes that are under the control of that thread: its cancelability state and type. A thread's cancelability state, determined by pthread_setcancelstate(3), can be enabled (the default for new threads) or disabled. If a thread has disabled cancellation, then a cancellation request remains queued until the thread enables cancellation. If a thread has enabled cancellation, then its cancelability type determines when cancellation occurs.

5. **int pthread_key_create(pthread_key_t key, void (destructor)(void*));**

   The pthread_key_create() function shall create a thread-specific data key visible to all threads in the process. Key values provided by pthread_key_create() are opaque objects used to locate thread-specific data. Although the same key value may be used by different threads, the values bound to the key by pthread_setspecific() are maintained on a per-thread basis and persist for the life of the calling thread.

   Upon key creation, the value NULL shall be associated with the new key in all active threads. Upon thread creation, the value NULL shall be associated with all defined keys in the new thread.

6. **int pthread_setspecific(pthread_key_t key, const void *value);**

   The pthread_getspecific() function shall return the value currently bound to the specified key on behalf of the calling thread.

   The pthread_setspecific() function shall associate a thread-specific value with a key obtained via a previous call to pthread_key_create(). Different threads may bind different values to the same key. These values are typically pointers to blocks of dynamically allocated memory that have been reserved for use by the calling thread.

   The effect of calling pthread_getspecific() or pthread_setspecific() with a key value not obtained from pthread_key_create() or after key has been deleted with pthread_key_delete() is undefined.

   Both pthread_getspecific() and pthread_setspecific() may be called from a thread-specific data destructor function. A call to pthread_getspecific() for the thread-specific data key being destroyed shall return the value NULL, unless the value is changed (after the destructor starts) by a call to pthread_setspecific(). Calling pthread_setspecific() from a thread-specific data destructor routine may result either in lost storage (after at least PTHREAD_DESTRUCTOR_ITERATIONS attempts at destruction) or in an infinite loop.

7. **void mmap(void addr, size_t lengthint " prot ", int " flags ,int fd, off_t offset);int munmap(void *addr, size_t length);**

   mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in addr. The length argument specifies the length of the mapping.
   If addr is NULL, then the kernel chooses the address at which to create the mapping; this is the most portable method of creating a new mapping. If addr is not NULL, then the kernel takes it as a hint about where to place the mapping; on Linux, the mapping will be created at a nearby page boundary. The address of the new mapping is returned as the result of the call.

   address = mmap(NULL, / **NULL = dont care which address**/
   len, / **100 - how much bytes** /
   PROT_READ | PROT_WRITE, / **permission** /
   MAP_PRIVATE, / **shared or not** /
   fildes, / **pointer to the file, use "open" function** /
   0); / **offset - in pages (4096 bytes in a page)** /

8. **What are 3 differences between a semaphore and a mutex ?**

   1. Mutex has ownership (although default is without) while semaphore doesnt.
   2. You can create a semaphore initiated to more than 1, mutex initiated to one only
   3. The purpose of semaphore is synchronization between different threads and processes ()consumer - producer), whilst with mutex the purpose is mutual exclusion for a critical area.

   point 3 addresses the fact that a mutex has ownership and semaphores doesnt, it allows that with semaphores after some number of processes have taken some resources, it will be locked, and some other totally unrelated process will work on creating more resources (producer-consumer).

9. **what are 3 differences between sys V sem and posix sem ?**

   1. with posix we make a syscall only when changing the state of the semaphore, whilst with sys V all use of the sem' requires a syscall.
   2. the ownership of a posix semaphore is of the creating process\thread, with sys V it is of the operating sys.
   3. with posix semaphore value can be changed only by 1 incremendt\decrement at a time whilst with sys V sem' it can be changed by more than.

10. **What are the differences between a process and a thread ?**

   1. each process lives in a separate virtual memory space while a thread lives inside its process virtual space.
   2. faster context switch with threads , because no need to change virtual space mapping - not necessary perform Translation lookaside buffer flush.
   3. creation of a new process is heavier than that of a thread , need to create a new virtual space.
   4. threads a more prone to deadlocks and race conditions because they share memory samespace.
   5. IPC is easier with threads because they share virtual memory space, processes require OS to communicate
   5. when a process dies all of its threads also die.
   6. a process threads share a lot of resources - like open files

11. **what are the three ways to react to a signal ?**

   1. default
   2. catch - can override catched signals with sigaction (writing our own function - "signal handler").
   3. ignore

12. **what does the following function does ?**
    int kill(pid_t pid, int sig);

   The kill() system call can be used to send any signal to any process group or process.
   e.g
   void childPong(int _num, siginfo_t **_info, void** _context)
   {
   write(1, "PING\n", 6);
   kill(_info->si_pid, SIGUSR1); / **_info->si_pid = pid of calling process** /
   }

13. **what is the difference between a signal and an interrupt ?**

   Signals can be viewed as a mean of communication between the OS kernel and processes.
   Interrupts can be viewed as a mean of communication between the OS kernel and the CPU.

14. **when should you use the pipe function so both father and child will receive the file descriptors ?**

   need to do pipe() before the fork().