Signals

What is the difference between sigaction and signal?

I was about to add an extra signal handler to an app we have here and I noticed that the author had used sigaction to set up the other signal handlers. I was going to use signal. To follow convention I should use sigaction but if I was writing from scratch, which should I choose?

Use sigaction() unless you've got very compelling reasons not to do so.

The signal() interface has antiquity (and hence availability) in its favour, and it is defined in the C standard. Nevertheless, it has a number of undesirable characteristics that sigaction() avoids - unless you use the flags explicitly added to sigaction() to allow it to faithfully simulate the old signal() behaviour.

1. The signal() function does not block other signals from arriving while the current handler is executing; sigaction() can block other signals until the current handler returns.
2. The signal() function resets the signal action back to SIG\_DFL (default) for almost all signals. This means that the signal() handler must reinstall itself as its first action. It also opens up a window of vulnerability between the time when the signal is detected and the handler is reinstalled during which if a second instance of the signal arrives, the default behaviour (usually terminate, sometimes with prejudice - aka core dump) occurs.

These are generally good reasons for using sigaction() instead of signal(). However, the interface of sigaction() is undeniably more fiddly.

Whichever of the two you use, do not be tempted by the alternative signal interfaces such as sighold(), sigpause(), sigignore() and sigrelse(). They are nominally alternatives to sigaction(), but they are only barely standardized and are present in POSIX for backwards compatibility rather than for serious use. Note that the POSIX standard says their behaviour in multi-threaded programs is undefined.

(Multi-threaded programs and signals is a whole other complicated story. AFAIK, both signal() and sigaction() are OK in multi-threaded applications.)

Lots of IPCs are offered by unix/linux : Pipes/sockets/shared memory/dbus/message-queues. I would like to know what are the respective most suitable applications , e.g. what are the places where shared memory are preferred over message queue and vice versa.

Also looking for data comparing their performances.

Here are the big seven:

1. [Pipe](http://en.wikipedia.org/wiki/Pipe_%28Unix%29)
2. [FIFO](http://en.wikipedia.org/wiki/Named_pipe), or named pipe
3. [Socket](http://en.wikipedia.org/wiki/Internet_socket) and [Unix Domain Socket](http://en.wikipedia.org/wiki/Unix_domain_sockets)
4. [Message Queue](http://en.wikipedia.org/wiki/Message_queue)
5. [Signal](http://en.wikipedia.org/wiki/Signal_%28computing%29)
6. [Semaphore](http://en.wikipedia.org/wiki/Semaphore_%28programming%29)
7. [Shared memory](http://en.wikipedia.org/wiki/Shared_memory)
   1. Pipe is useful only among processes related as parent/child. Call [pipe(2)](http://linux.die.net/man/2/pipe) and [fork(2)](http://linux.die.net/man/2/fork). Unidirectional.
   2. Two unrelated processes can use FIFO unlike plain pipe. Call [mkfifo(3)](http://linux.die.net/man/3/mkfifo). Unidirectional.
   3. 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)](http://linux.die.net/man/2/socket).
   4. Message Queue. OS maintains discrete message. See [sys/msg.h](http://opengroup.org/onlinepubs/007908799/xsh/sysmsg.h.html).
   5. Signal sends an integer to another process. Doesn't mesh well with multi-threads. Call[kill(2)](http://linux.die.net/man/2/kill).
   6. Semaphore is a synchronization mechanism for multi processes or threads, similar to a queue of people waiting for bathroom. See [sys/sem.h](http://www.opengroup.org/onlinepubs/7990989775/xsh/syssem.h.html).
   7. Shared memory is a shared memory. Do your own concurrency control. Call [shmget(2)](http://linux.die.net/man/2/shmget).

**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.

t's worth noting that lots of libraries implement one type of thing on top of another.

Shared memory doesn't need to use the horrible sysv shared memory functions - it's much more elegant to use mmap() (mmap a file in on a tmpfs /dev/shm if you want it named; mmap /dev/zero if you want forked not exec'd processes to inherit it anonymously). Having said that, it still leaves your processes with some need for synchronisation to avoid problems - typically by using some of the other IPC mechanisms to do synchronisation of access to a shared memory area.

Here is a webpage with a simple benchmark: <http://www.rikkus.info/sysv-ipc-vs-unix-pipes-vs-unix-sockets>

As far as I can tell, each has their advantages:

* Pipe I/O is the fastest but needs a parent/child relationship to work.
* Sysv IPC has a defined message boundary and can connect disparate processes locally.
* UNIX sockets can connect disparate processes locally and has higher bandwidth but no inherent message boundaries.
* TCP/IP sockets can connect any processes, even over the network but has higher overhead and no inherent message boundaries.

Posix threads and signals

I've been trying to understand the intricacies of how POSIX threads and POSIX signals interact. In particular, I'm interested in:

* What's the best way to control which thread a signal is delivered to (assuming it isn't fatal in the first place)?
* What is the best way to tell another thread (that might actually be busy) that the signal has arrived? (I already know that it's a bad idea to be using pthread condition variables from a signal handler.)
* How can I safely handle passing the information that a signal has occurred to other threads? Does this need to happen in the signal handler? (I do not in general want to kill the other threads; I need a far subtler approach.)

For reference about why I want this, I'm researching how to convert the [TclX](http://sourceforge.net/projects/tclx/) package to support threads, or to split it up and at least make some useful parts support threads. Signals are one of those parts that is of particular interest.

* What's the best way to control which thread a signal is delivered to?

As @zoli2k indicated, explicitly nominating a single thread to handle all signals you want handled (or a set of threads each with specific signal responsibilities), is a good technique.

* What is the best way to tell another thread (that might actually be busy) that the signal has arrived?[...]
* How can I safely handle passing the information that a signal has occurred to other threads? Does this need to happen in the signal handler?

I won't say "best," but here's my recommendation:

Fashion the signal receiving thread as a signal-driven event loop, dispatching newly arrived signals as**some other intra-thread communication**. This thread will block all signals except when waiting for new ones to arrive.

The receiver's signal handler is simple and uncontroversial. It might, for instance:

* set sig\_atomic\_t flags
* call sigaddset(&signals\_i\_have\_seen\_recently, latest\_sig)
* write() a byte to a non-blocking [self-pipe](http://cr.yp.to/docs/selfpipe.html) to break itself out of a select or pselect  
  (or to awaken another thread directly)

all of which are signal handler safe.

In the loop body, again with most/all signals blocked, the thread then converts the newly arrived signals, perhaps broadcasting a pthread\_cond\_t, waking up other threads with more I/O, enqueuing a command in an application-specific thread-safe queue, whatever.

According to the POSIX standard all threads should appear with the same PID on the system and usingpthread\_sigmask you can define the signal blocking mask for every thread.

Since it is allowed to define only one signal handler per PID, I prefer to handle all signals in one thread and send pthreads\_cancel if a running thread need to be cancelled. It is the preferred way againstpthreads\_kill since it allows to define cleanup functions for the threads.

On some older systems, because of the lack of proper kernel support, the running threads may have different PID from the parent thread's PID. See FAQ for signal handling with [linuxThreads on Linux 2.4](http://pauillac.inria.fr/~xleroy/linuxthreads/faq.html" \l "J).

1. In unix, If a multi-threaded process was sent a signal, which thread will be the one to execute the handling function?
2. if it is a multi-cpu machine, more than 1 thread is running at the same time. which thread will be the on to run the signal handling function?

According to [man 7 signal](http://linux.die.net/man/7/signal), all threads in the process share the same signal handler, and if a signal is delivered to a process with multiple threads that have not blocked the signal, one of them is arbitrarily chosen to receive it.

Having a multi-CPU machine will not change these semantics.

I was under the impression that signal handlers were called by the kernel when it encountered an exceptional condition (e.g. divide by 0). Also, that they're only called if you specifically register them.

This would seem to imply (to me) that they aren't called through your normal code.

Moving on with that thought... setjmp and longjmp as I understand them are for collapsing up the stack to a previous point and state. I don't understand how you can collapse up a stack when a signal handler is called since its called from the Kernel as a one-off circumstance rather than from your own code. What's the next thing up the stack from a signal handler!?

longjmp [does not perform normal stack unwinding](http://en.wikipedia.org/wiki/Setjmp.h#Caveats_and_limitations). Instead, the stack pointer is simply restored from the context saved by setjmp.

[Here is an illustration](http://www.gnu.org/s/hello/manual/libc/Longjmp-in-Handler.html) on how this can bite you with non-async-safe critical parts in your code. It is advisable to e.g. mask the offending signal during critical code.

The way the kernel "calls" a signal handler is by interrupting the thread, saving the signal mask and processor state in a ucontext\_t structure on the stack just beyond (below, on grows-down implementations) the interrupted code's stack pointer, and restarting execution at the address of the signal handler. The kernel does not need to keep track of any "this process is in a signal handler" state; that's entirely a consequence of the new call frame that was created.

If the interrupted thread was in the middle of a system call, the kernel will back out of the kernelspace code and adjust the return address to repeat the system call (if SA\_RESTART is set for the signal and the system call is a restartable one) or put EINTR in the return code (if not restartable).

It should be noted that longjmp is async-signal-unsafe. This means it invokes undefined behavior if you call it from a signal handler if the signal interrupted another async-signal-unsafe function. But as long as the interrupted code is not using library functions, or only using library functions that are marked async-signal-safe, it's legal to call longjmp from a signal handler.

Finally, my answer is based on POSIX since the question is tagged unix. If the question were just about pure C, I suspect the answer is somewhat different, but signals are rather useless without POSIX anyway...

I need to find the number of signals pending in the signal queue of a thread in linux. Is there any API that is provided by Linux ?

This API needs to be called from thread, other than the thread we are querying.

sigpending gives the API for the calling thread. Is there any API, which takes thread id as arg, and provides some information about the signals pending in the queue.

sigpending() returns the set of signals pending delivery for a thread. You can iterate over that sigset\_t and use sigismember() to test for membership.

Traditionally, UNIX does not queue more than one instance of a signal (in this sense, they continue to be "unreliable"). Sometimes you can get this effect with realtime signals.

On Linux, read()ing from a signalfd will allow more than one instance of a signal to be read at once, should they be queued.

What's the difference between kernel stack and user stack? Why kernel stack is used? If a local variable is declared in an ISR, where it will be stored? Does each process has its own kernel stack ? Then how the process coordinates between both these stacks?

1. *What's the difference between kernel stack and user stack ?*  
   In short, nothing - apart from using a different location in memory (and hence a different value for the stackpointer register), and usually different memory access protections. I.e. when executing in user mode, kernel memory (part of which is the kernel stack) will not be accessible even if mapped. Vice versa, without explicitly being requested by the kernel code (in Linux, through functions likecopy\_from\_user()), user memory (including the user stack) is not usually directly accessible.
2. *Why is [ a separate ] kernel stack used ?*  
   Separation of privileges and security. For one, userspace programs can make their stack(pointer) anything they want, and there is usually no architectural requirement to even have a valid one. The kernel therefore cannot *trust* the userspace stackpointer to be valid nor usable, and therefore will require one set under its own control. Different CPU architectures implement this in different ways; x86 CPUs automatically switch stackpointers when privilege mode switches occur, and the values to be used for different privilege levels are configurable - by privileged code (i.e. only the kernel).
3. *If a local variable is declared in an ISR, where will it be stored ?*  
   On the kernel stack. The kernel (Linux kernel, that is) does *not* hook ISRs directly to the x86 architecture's *interrupt gates* but instead delegates the interrupt dispatch to a common kernel interrupt entry/exit mechanism which saves pre-interrupt register state before calling the registered handler(s). The CPU itself when dispatching an interrupt might execute a privilege and/or stack switch, and this is used/set up by the kernel so that the common interrupt entry code can already rely on a kernel stack being present.  
   That said, interrupts that occur while executing kernel code will simply (continue to) use the kernel stack in place at that point. This can, if interrupt handlers have deeply nested call paths, lead to stack overflows (if a deep kernel call path is interrupted and the handler causes another deep path; in Linux, filesystem / software RAID code being interrupted by network code with iptables active is known to trigger such in untuned older kernels ... solution is to increase kernel stack sizes for such workloads).
4. *Does each process have its own kernel stack ?*  
   Not just each process - each *thread* has its own kernel stack (and, in fact, its own user stack as well). Remember the only difference between processes and threads (to Linux) is the fact that multiple threads can share an address space (forming a process).
5. *How does the process coordinate between both these stacks ?*  
   Not at all - it doesn't need to. Scheduling (how / when different threads are being run, how their state is saved and restored) is the operating system's task and processes don't need to concern themselves with this. As threads are created (and each process must have at least one thread), the kernel creates kernel stacks for them, while userspace stacks are either explicitly created/provided by whichever mechanism is used to create a thread (functions like makecontext() orpthread\_create() allow the caller to specify a memory region to be used for the "child" thread's stack), or inherited (by on-access memory cloning, usually called "copy on write" / COW, when creating a new process).  
   That said, the process *can* influence scheduling of its threads and/or influence the *context* (state, amongst that is the thread's stackpointer). There are multiple ways for this: UNIX signals,setcontext(), pthread\_yield() / pthread\_cancel(), ... - but this is disgressing a bit from the original question.

My answer is collected from other SO questions with my stuffs.

What's the difference between kernel stack and user stack?

As a kernel programmer, you know that the kernel should be restricted from erroneous user programs. Suppose you keep same stack for both kernel & user space, then simple segfault in user application crashes kernel and needs restart.

There is one "kernel stack" per CPU like ISR Stack and one "kernel stack" per Process. There is one "user stack" for each process, though each thread has its own stack, including both user and kernel threads.

<http://linux.derkeiler.com/Mailing-Lists/Kernel/2004-10/3194.html>

Why kernel stack is used?

So when we are in kernel mode, stack kind of mechanism is necessary for dealing with function calls, local variables similar to user space.

<http://www.kernel.org/doc/Documentation/x86/x86_64/kernel-stacks>

If a local variable is declared in an ISR, where it will be stored?

It will be stored in ISR stack(IRQSTACKSIZE). The ISR runs on a separate interrupt stack only if the hardware supports it. Otherwise, the ISR stack frames are pushed onto the stack of the interrupted thread.

The user space does not know and frankly does not care about whether the interrupt is served in current process's kernel stack or a separate ISR stack. As interrupts comes per cpu, therefore ISR stack has to be per cpu.

Does each process has its own kernel stack ?

Yes. Each process has its own kernel stack.

Then how the process coordinates between both these stacks?

@FrankH's answer looks great to me.

Pipe as inter process communication

It seems like the receiving end will block waiting for input, which I would expect

You expect correctly an actual 'read' call will block until something is there. However, I believe there are some C functions that will allow you to 'peek' at what (and how much) is waiting in the pipe. Unfortunately, I don't remember if this blocks as well.

will the sending end block sometimes waiting for someone to read from the stream

No, sending should never block. Think of the ramifications if this were a pipe across the network to another computer. Would you want to wait (through possibly high latency) for the other computer to respond that it received it? Now this is a different case if the reader handle of the destination has been closed. In this case, you should have some error checking to handle that.

If I write an eof to the stream can I keep continue writing to that stream until I close it

I would think this depends on what language you're using and its implementation of pipes. In C, I'd say no. In a linux shell, I'd say yes. Someone else with more experience would have to answer that.

Are there differences in the behaviour named and unnamed pipes? As far as I know, yes. However, I don't have much experience with named vs unnamed. I believe the difference is:

* Single direction vs Bidirectional communication
* Reading AND writing to the "in" and "out" streams of a thread

Does it matter which end of the pipe I open first with named pipes?

Generally no, but you could run into problems on initialization trying to create and link the threads with each other. You'd need to have one main thread that creates all the sub-threads and syncs their respective pipes with each other.

Is the behaviour of pipes consistent between different linux systems?

Again, this depends on what language, but generally yes. Ever heard of POSIX? That's the standard (at least for linux, Windows does it's own thing).

Does the behaviour of the pipes depend on the shell I'm using or the way I've configured it?

This is getting into a little more of a gray area. The answer *should* be no since the shell should essentially be making system calls. However, everything up until that point is up for grabs.

Are there any other questions I should be asking

The questions you've asked shows that you have a decent understanding of the system. Keep researching and focus on what level you're going to be working on (shell, C, so on). You'll learn a lot more by just trying it though.

Communicating between a parent and its children

On Unix, in a program with a parent and some children:   
- How can the parent alert the children **efficiently** to do some work.. ?   
- Or how can the children wait for parent signal to start doing some work?

EDIT:  
This program tries to do a complex computation in parallel, I have already used shared memory as a common workspaces for all children to update results and for data transfer.  
What I need now is the parent say "start" efficiently to all its children...(called many times)

Your ipc tag says it all. You need to look into inter-process communuication:

* Shared memory.
* Semaphores.
* Pipes.
* Signals.
* Memory-mapped files.
* Sockets.

No doubt there are other possibilities but that's a good start.

How *efficient* each is depends pretty much on your use case. If you only need to notify a child to do something, signals is probably what I'd use. If you need to transfer some more information between processes, it's probably a good idea to fully specify the requirements.

One thing you *may* want to consider is to bypass all the inter-process stuff altogether and just use threads. At least in Linux, threads are first class citizens to the scheduler. Older UNIXes may have made a distinction (with user-mode threads) but that is not the case with Linux.

I've found that it's simpler to do it that way and your information is automatically shared (keeping in mind you still need to protect shared stuff with mutexes and such).

Another possibility if you're already committed to shared memory is to use signals. Assuming you've set aside sections of that memory for each child (and they know where they are), signals are probably the quickest way to notify a bunch of children for parallel work.

If your children just wait around in a select loop with (for example) a 30-second timeout for doing periodic work, a signal will cause it to exit immediately with EINTR. That gives you your efficient CPU usage while still giving immediate response.

[Is fork (supposed to be) safe from signal handlers in a threaded program?](http://stackoverflow.com/questions/4453822/is-fork-supposed-to-be-safe-from-signal-handlers-in-a-threaded-program)

'm really uncertain about the requirements POSIX places on the safety of fork in the presence of threads and signals. fork is listed as one of the async-signal-safe functions, but if there is a possibility that library code has registered pthread\_atfork handlers which are not async-signal-safe, does this negate the safety of fork? Does the answer depend on whether the thread in which the signal handler is running could be in the middle of using a resource that the atfork handlers need? Or said a different way, if the atfork handlers make use of synchronization resources (mutexes, etc.) but fork is being called from a signal handler which executed in a thread that never accesses these resources, is the program conforming?

Building on this question, if "thread-safe" forking is implemented internally in the system library using the idioms suggested by pthread\_atfork (obtain all locks in the prefork handler and release all locks in both the parent and child postfork handlers), then is fork ever safe to use from signal handlers in a threaded program? Isn't it possible that the thread handling the signal could be in the middle of a call tomalloc or fopen/fclose and holding a global lock, resulting in deadlock during fork?

Finally, even if fork is safe in signal handlers, is it safe to fork in a signal handler and then return from the signal handler, or does a call to fork in a signal handler always necessitate a subsequent call to \_exit or one of the exec family of functions before the signal handler returns?

Trying my best to answer all the sub-questions; I apologise that some of this is vaguer than it ideally should be:

If there is a possibility that library code has registered pthread\_atfork handlers which are not async-signal-safe, does this negate the safety of fork?

Yes. The [fork documentation](http://pubs.opengroup.org/onlinepubs/009695399/functions/fork.html) explicitly mentions this:

When the application calls fork() from a signal handler and any of the

fork handlers registered by pthread\_atfork() calls a function that is

not asynch-signal-safe, the behavior is undefined.

Of course, this means you can't actually use pthread\_atfork() for its intended purpose of making multi-threaded libraries transparent to processes that believe they are single-threaded, because none of the pthread synchronisation functions are async-signal-safe; this is noted as a defect in the spec, see<http://www.opengroup.org/austin/aardvark/latest/xshbug3.txt> (search for "L16723").

Does the answer depend on whether the thread in which the signal handler is running could be in the middle of using a resource that the atfork handlers need? Or said a different way, if the atfork handlers make use of synchronization resources (mutexes, etc.) but fork is being called from a signal handler which executed in a thread that never accesses these resources, is the program conforming?

Strictly speaking the answer is no, because the according to the spec, functions are either async-signal-safe or they're not; there's no concept of "safe under certain circumstances". In practice you might well get away with it, but you would be vulnerable to a clunky-but-correct implementation that didn't partition its resources in the way you were expecting.

Building on this question, if "thread-safe" forking is implemented internally in the system library using the idioms suggested by pthread\_atfork (obtain all locks in the prefork handler and release all locks in both the parent and child postfork handlers), then is fork ever safe to use from signal handlers in a threaded program? Isn't it possible that the thread handling the signal could be in the middle of a call to malloc or fopen/fclose and holding a global lock, resulting in deadlock during fork?

If it were implemented in that way, then you're right, fork() from a signal handler would never be safe, because attempting to obtain a lock might deadlock if the calling thread already held it. But this implies that an implementation using such a method would not be conforming.

Looking at glibc as one example, it doesn't do that - rather, it takes two approaches: firstly, the locks that it does obtain are recursive (so if the current thread already has them, their lock count will simply be increased); further, in the child process, it simply unilaterally overwrites all the locks - see this extract from nptl/sysdeps/unix/sysv/linux/fork.c:

/\* Reset the file list. These are recursive mutexes. \*/

fresetlockfiles ();

/\* Reset locks in the I/O code. \*/

\_IO\_list\_resetlock ();

/\* Reset the lock the dynamic loader uses to protect its data. \*/

\_\_rtld\_lock\_initialize (GL(dl\_load\_lock));

where each of the resetlock and lock\_initialize functions ultimately call glibc's internal equivalent of pthread\_mutex\_init(), effectively resetting the mutex regardless of any waiters.

I think the theory is that, by obtaining the (recursive) lock it's guaranteed that no other threads will be touching the data structures (at least in a way that might cause a crash), and then resetting the individual locks ensures the resources aren't permanently blocked. (Resetting the current thread's lock is safe since there are now no other threads to contend for the data structure, and indeed won't be until whatever function is using the lock has returned).

I'm not 100% convinced that this covers all eventualities (not least because if/when the signal handler returns, the function that's just had its lock stolen will try to unlock it, and the internal recursive unlock function doesn't protect against unlocking too many times!) - but it seems that a workable scheme *could*be built on top of async-signal-safe recursive locks.

Finally, even if fork is safe in signal handlers, is it safe to fork in a signal handler and then return from the signal handler, or does a call to fork in a signal handler always necessitate a subsequent call to \_exit or one of the exec family of functions before the signal handler returns?

I assume you're talking about the child process? (If fork() being async-signal-safe means anything then it should be possible to return in the parent!)

Not having found anything in the spec that states otherwise (though I may have missed it) I believe it*should be* safe - at least, 'safe' in the sense that returning from the signal handler in the child doesn't imply undefined behaviour in and of itself, though the fact that a multi-threaded process has just forked may imply that an exec\*() or \_exit() is probably the safest course of action.