What are the issues with Fork in a multithreaded program

The problem with forking when you do have some threads running is that the fork only copies the CPU state of the one thread that called it. It's as if all of the other threads just died, instantly, wherever they may be.

The result of this is locks aren't released, and shared data (such as the malloc heap) may be corrupted.

pthread does offer a [pthread\_atfork](http://www.opengroup.org/onlinepubs/009695399/functions/pthread_atfork.html) function - in theory, you could take every lock in the program before forking, release them after, and maybe make it out alive - but it's risky, because you could always miss one. And, of course, the stacks of the other threads won't be freed.

The fork clones all locks while they're still in the locked state

Look here

<http://www.linuxprogrammingblog.com/threads-and-fork-think-twice-before-using-them>

for detailed explanation

It appears that the threads do not follow. In my case, I am trying to create a daemon and I use fork() with the parent exiting to deamonize it. However, in a new path through the code, I create some threads before the fork and some after. Is there an easy way to change ownership of the threads over to the new forked process rather than moving all my thread creation after the fork?

Nothing. Only the thread calling fork() gets duplicate. The child process has to start any new threads. The parents threads are left alone.

In POSIX when a multithreaded process forks, the child process looks exactly like a copy of the parent, but in which all the threads stopped dead in their tracks and disappeared.

This is very bad if the threads are holding locks.

For this reason, there is a crude mechanism called pthread\_atfork in which you can register handlers for this situation.

Any properly written program module (and especially reusable middleware) which uses mutexes must callpthread\_atfork to register some handlers, so that it does not misbehave if the process happens to call fork.

Besides mutex locks, threads could have other resources, such as thread-specific data squirreled away with pthread\_setspecific which is only accessible to the thread (and the thread is responsible for cleaning it up via a destructor).

In the child process, no such destructor runs. The address space is copied, but the thread and its thread specific value is not there, so the memory is leaked in the child. This can and should be handled withpthread\_atfork handlers also.

Suppose I have a process which spawns exactly one child process. Now when the parent process exits for whatever reason (normally or abnormally, by kill, ^C, assert failure or anything else) I want the child process to die. How to do that correctly?

If you're unable to modify the child process, you can try something like the following:

int pipes[2];

pipe(pipes)

if (fork() == 0) {

close(pipes[1]); /\* Close the writer end in the child\*/

dup2(0, pipes[0]); /\* Use reader end as stdin \*/

exec("sh -c 'set -o monitor; child\_process & read dummy; kill %1'")

}

close(pipes[0]); /\* Close the reader end in the parent \*/

This runs the child from within a shell process with job control enabled. The child process is spawned in the background. The shell waits for a newline (or an EOF) then kills the child.

When the parent dies--no matter what the reason--it will close its end of the pipe. The child shell will get an EOF from the read and proceed to kill the backgrounded child process.

Child can ask kernel to deliver SIGHUP (or other signal) when parent dies by specifying optionPR\_SET\_PDEATHSIG in prctl() syscall like this:

prctl(PR\_SET\_PDEATHSIG, SIGHUP);

But this is only in Linux

Fork in a multithreaded process

The basic thumb rules, according to various internet articles like (<http://www.linuxprogrammingblog.com/threads-and-fork-think-twice-before-using-them> , [fork in multi-threaded program](http://stackoverflow.com/questions/1235516/fork-in-multi-threaded-program) ) are:

1. *(Main) Process[0] Monothread --> fork() --> (Child) Process[1] Multithreaded*: **OK!**  
   If *Process[1]* crashes or messes around with memory it *won't* touch address space of *Process[0]*(unless you use shared R/W memory... but this is another topic of its own).  
   In Linux by default all *fork()ed* memory is [Copy On Write](http://en.wikipedia.org/wiki/Copy-on-write). Given that *Process[0]* is monothreaded, when we invoke *fork()* all possible mutual exclusion primitives should be *generally* in an *unlocked*state.
2. *(Main) Process[0] Multithreaded --> fork() --> (Child) Process[1] Mono/Multithread*: **BAD!**  
   If you *fork()* a Multithreaded process your mutexes and many other thread synchronization primitives will likely be in an *undefined state* in Process[1]. You can work around with *pthread\_atfork()* but if you use libraries you might as well roll a dice and hope to be lucky. Because generally you don't (want to) know the implementation details of libraries.

The advantages of *fork()* into a multithreaded process are that you could manipulate/read/aggregate your data quicker (in the Child process), without having to care about stability of the process you *fork()*from (Main). This is useful if your main process has a dataset of *a lot* of memory and you don't want to duplicate/reload it to safely process the data in another process (Child). This way the original process is stable and independent from the data aggregation/manipulation process (*fork()ed*).

Of course this means that the original process *will generally* be slower than it might be if developed in multithreaded fashion. But again, this is the price you might want to be paying for more stability.

If instead your main process is multithreaded, *refrain* from using *fork()*. It's going to be a *proper mess* to implement it in a stable way.

How is the implementation of threads done in a system? I know that child processes are created using the fork() call and a thread is a light weight. How does the creation of a thread differ from that of a child process?

Threads are created using the clone() system call that can make a new process that shares memory space and some of the kernel control structures with its parent. These processes are called LWPs (light-weight processes) and are also known as kernel-level threads.

fork() creates a new process that *initially* shares memory with its parent but pages are copy-on-write, which means that separate memory pages are created when the content of the original one is altered. Thus both parent and child processes can no longer change each other's memory and effectively they run as separate processes. Also the newely forked child is a full-blown processes with its separate kernel control structures.

***Each process has its own address space*** aka range of virtual addresses that the process can access. When a new process is *forked* a duplicate copy of all the resources involved has to be made. After the forking is complete the child and the parent have their own distinct address space and all the resources involved within it.Naturally, this is an performance intensive operation.

While ***all threads in the same process share the same address space***, So when a new thread is*spawned* each thread only needs its own stack and there is no duplication of all resources as in case of processes.Hence spawning of an thread is considerably less performance intensive.

What are the main advantages of using a model for concurrency based on processes over one based on threads and in what contexts is the latter appropriate?

Fault-tolerance and scalability is the main advantages of using processes vs. threads.

A system that relies on shared memory or some other kind of technology only available when using threads, will be useless you want to run the system on multiple machines. Sooner or later you need to communicate between different processes.

Using processes you are forced to deal with communication through messages, which is the Erlang way of doing communication. Data is not shared, so there is no risk of data corruption.

Another advantage of processes is that they can crash and you are perfectly ok with that, because you just restart them (even across network hosts). If thread crashes, it may crash the entire process, which may bring down your entire application. If an Erlang process crashes, you will only lose that phone call, or that webrequest, etc.

OS processes have many drawbacks that make them harder to use, like the fact that it takes forever to spawn a new process. However, Erlang has it's own notion of processes, which are extremely lightweight.

The disadvantage of using a process-based model is that it will be slower. You will have to copy data between the concurrent parts of your program.

The disadvantage of using a thread-based model is that you will probably get it wrong. It may sound mean, but it's true-- show me code based on threads and I'll show you a bug. I've found bugs in threaded code that has run "correctly" for 10 years.

The advantages of using a process-based model are numerous. The separation forces you to think in terms of protocols and formal communication patterns, which means its far more likely that you will get it right. Processes communicating with each other are easier to scale out across multiple machines. Multiple concurrent processes allows one process to crash without necessarily crashing the others.

The advantage of using a thread-based model is that it is fast.

It may be obvious which of the two I prefer, but in case it isn't: processes, every day of the week and twice on Sunday. Threads are too hard: I haven't ever met anybody who could write correct multi-threaded code; those that claim to be able to usually don't know enough about the space yet.

First and foremost, processes differ from threads mostly in the way their memory is handled:

Process = n\*Thread + memory region (n>=1)

Processes have their own isolated memory. Processes can have multiple threads.

Processes are isolated from each other on the operating system level. Threads share their memory with their peers in the process. (This is often undesirable. There are libraries and methods out there to remedy this, but that is usually an artificial layer over operating system threads.)

The memory thing is the most important discerning factor, as it has certain implications:

1. Exchanging data between processes is slower than between threads. Breaking the process isolation always requires some involvement of kernel calls and memory remapping.
2. Threads are more lightweight than processes. The operating system has to allocate resources and do memory management for each process.
3. Using processes gives you memory isolation and synchronization. Common problems with access to memory shared between threads do not concern you. Since you have to make a special effort to share data between processes, you will most likely sync automatically with that.

Using processes gives you good (or ultimate) encapsulation. Since inter process communication needs special effort, you will be forced to define a clean interface. It is a good idea to break certain parts of your application out of the main executable. Maybe you can split dependencies like that. e.g.Process\_RobotAi <-> Process\_RobotControl The AI will have vastly different dependencies compared to the control component. The interface might be simple: Process\_RobotAI --DriveXY--> Process\_RobotControl. Maybe you change the robot platform. You only have to implement a newRobotControl executable with that simple interface. You don't have to touch or even recompile anything in your AI component.

It will also, for the same reasons, speed up compilation in most cases.

Edit: Just for completeness I will shamelessly add what the others have reminded me of : A crashing process does not (necessarily) crash your whole application.

In General:

1. Want to create something highly concurrent or synchronuous, like an algorithm with n>>1 instances running in parallel and sharing data, use threads.
2. Have a system with multiple components that do not need to share data or algorithms, nor do they exchange data too often, use processes. If you use a RPC library for the inter process communication, you get a network-distributable solution at no extra cost.

I've recently heard a few people say that in Linux, it is almost always better to use processes instead of threads, since Linux is very efficient in handling processes, and because there are so many problems (such as locking) associated with threads. However, I am suspicious, because it seems like threads could give a pretty big performance gain in some situations.

So my question is, when faced with a situation that threads and processes could both handle pretty well, should I use processes or threads? For example, if I were writing a web server, should I use processes or threads (or a combination)?

Linux uses a 1-1 threading model, with (to the kernel) no distinction between processes and threads -- everything is simply a runnable task. \*

On Linux, the system call clone clones a task, with a configurable level of sharing, among which are:

* CLONE\_FILES: share the same file descriptor table (instead of creating a copy)
* CLONE\_PARENT: don't set up a parent-child relationship between the new task and the old (otherwise, child's getppid() = parent's getpid())
* CLONE\_VM: share the same memory space (instead of creating a COW copy)

fork() calls clone(least sharing) and pthread\_create() calls clone(most sharing). \*\*

forking costs a tiny bit more than pthread\_createing because of copying tables and creating COW mappings for memory, but the Linux kernel developers have tried (and succeeded) at minimizing those costs.

Switching between tasks, if they share the same memory space and various tables, will be a tiny bit cheaper than if they aren't shared, because the data may already be loaded in cache. However, switching tasks is still very fast even if nothing is shared -- this is something else that Linux kernel developers try to ensure (and succeed at ensuring).

In fact, if you are on a multi-processor system, **not** sharing may actually be beneficial to performance: if each task is running on a different processor, synchronizing shared memory is expensive.

\* Simplified. CLONE\_THREAD causes signals delivery to be shared (which needs CLONE\_SIGHAND, which shares the signal handler table).

\*\* Simplified. There exist both SYS\_fork and SYS\_clone syscalls, but in the kernel, the sys\_forkand sys\_clone are both very thin wrappers around the same do\_fork function, which itself is a thin wrapper around copy\_process. Yes, the terms process, thread, and task are used rather interchangeably in the Linux kernel...

On a linux system, does the child process view the existing threads the same way as the parent process

Threads on linux nowadays tries to stay posix complient. Only the calling thread is replicated, not other threads (note that e.g. on solaris you can chose what fork does depending on what library you link to)

From <http://www.opengroup.org/onlinepubs/000095399/functions/fork.html>:

A process shall be created with a single thread. If a multi-threaded process calls fork(), the new process shall contain a replica of the calling thread and its entire address space, possibly including the states of mutexes and other resources. Consequently, to avoid errors, the child process may only execute async-signal-safe operations until such time as one of the exec functions is called. Fork handlers may be established by means of the pthread\_atfork() function in order to maintain application invariants across fork() calls.

is there a relationship between kernel/user thread? some OS textbook said that "**maps** one(many) user thread to one(many) kernel thread",what does **map** means here?

When they say map, they mean that each kernel thread is assigned to a certain number of user mode threads.

Kernel threads are used to provide privileged services to applications (such as system calls ). The are also used by the kernel to keep track of what all is running on the system, how much of which resources are allocated to what process, and to do scheduling.

If your applications make a heavy use of system calls, the more user threads per kernel thread, the slower your applications will run, because the kernel thread will become a bottleneck, since all system calls will pass through it.

On the flip side though, if you're programs rarely use system calls (or other kernel services), you can assign a large number of user threads to a kernel thread without much performance penalty, other than overhead.

You can increase the number of kernel threads, but this adds overhead to the kernel in general, so while individual threads will be more responsive with respect to system calls, the system as a whole will become slower.

That is why it is important to find a good balance between the number of kernel threads and the number of user threads per kernel thread.

<http://www.linuxforu.com/2011/08/light-weight-processes-dissecting-linux-threads/>

How efficient is locking an unlocked mutex. In a low level language (C, C++ or whatever): I have the choice in between either having a bunch of mutexes (like what pthread gives me or whatever the native system library provides) or a single one for an object.

How efficient is it to lock a mutex? I.e. how much assembler instructions are there likely and how much time do they take (in the case that the mutex is unlocked)?

How much does a mutex cost? Is it a problem to have really **a lot** of mutexes? Or can I just throw as much mutex variables in my code as I have int variables and it doesn't really matter?

(I am not sure how much differences there are between different hardware. If there is, I would also like to know about them. But mostly, I am interested about common hardware.)

The point is, by using many mutex which each cover only a part of the object instead of a single mutex for the whole object, I could safe many blocks. And I am wondering how far I should go about this. I.e. should I try to safe any possible block really as far as possible, no matter how much more complicated and how much more mutexes this means?

I have the choice in between either having a bunch of mutexes or a single one for an object.

If you have many threads and the access to the object happens often, then multiple locks would increase parallelism. At the cost of maintainability, since more locking means more debugging of the locking.

How efficient is it to lock a mutex? I.e. how much assembler instructions are there likely and how much time do they take (in the case that the mutex is unlocked)?

The precise assembler instructions are the least overhead of [a mutex](http://en.wikipedia.org/wiki/Lock_%28computer_science%29) - [the memory/cache coherency](http://en.wikipedia.org/wiki/Cache_coherence)guarantees are the main overhead. And less often a particular lock is taken - better.

Mutex is made of two major parts (oversimplifying): (1) a flag indicating whether the mutex is locked or not and (2) wait queue.

Change of the flag is just few instructions and normally done without system call. If mutex is locked, syscall will happen to add the calling thread into wait queue and start the waiting. Unlocking, if the wait queue is empty, is cheap but otherwise needs a syscall to wake up one of the waiting processes. (On some systems cheap/fast syscalls are used to implement the mutexes, they become slow (normal) system calls only in case of contention.)

Locking unlocked mutex is really cheap. Unlocking mutex w/o contention is cheap too.

How much does a mutex cost? Is it a problem to have really a lot of mutexes? Or can I just throw as much mutex variables in my code as I have int variables and it doesn't really matter?

You can throw as much mutex variables into your code as you wish. You are only limited by the amount of memory you application can allocate.

Summary. User-space locks (and the mutexes in particular) are cheap and not subjected to any system limit. But too many of them spells nightmare for debugging. Simple table:

1. Less locks means more contentions (slow syscalls, CPU stalls) and lesser parallelism
2. Less locks means less problems debugging multi-threading problems.
3. More locks means less contentions and higher parallelism
4. More locks means more chances of running into undebugable deadlocks.

A balanced locking scheme for application should be found and maintained, generally balancing the #2 and the #3.

(\*) The problem with less very often locked mutexes is that if you have too much locking in your application, it causes to much of the inter-CPU/core traffic to flush the mutex memory from the data cache of other CPUs to guarantee the cache coherency. The cache flushes are like light-weight interrupts and handled by CPUs transparently - but they do introduce so called [stalls](http://en.wikipedia.org/wiki/Central_processing_unit) (search for "stall").

And the stalls are what makes the locking code to run slowly, often without any apparent indication why application is slow. (Some arch provide the inter-CPU/core traffic stats, some not.)

To avoid the problem, people generally resort to large number of locks to decrease the probability of lock contentions and to avoid the stall. That is the reason why the cheap user space locking, not subjected to the system limits, exists.

Processes vs threads

What is the technical difference between a process and a thread? I get the feeling a word like 'process' is over used and there is also hardware and software threads. How about light-weight processes in languages like Erlang? Is there a definitive reason to use one term over the other?

Both processes and threads are independent sequences of execution. The typical difference is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces.

I'm not sure what "hardware" vs "software" threads might be referring to. Threads are an operating environment feature, rather than a CPU feature (though the CPU typically has operations that make threads efficient).

Erlang uses the term "process" because it does not expose a shared-memory multiprogramming model. Calling them "threads" would imply that they have shared memory.

**Process**  
Each process provides the resources needed to execute a program. A process has a virtual address space, executable code, open handles to system objects, a security context, a unique process identifier, environment variables, a priority class, minimum and maximum working set sizes, and at least one thread of execution. Each process is started with a single thread, often called the primary thread, but can create additional threads from any of its threads.

**Thread**  
A thread is the entity within a process that can be scheduled for execution. All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, thread local storage, a unique thread identifier, and a set of structures the system will use to save the thread context until it is scheduled. The thread context includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process. Threads can also have their own security context, which can be used for impersonating clients.

Why are threads light weight processes

Process creation is "expensive", because it has to set up a complete new virtual memory space for the process with it's own address space. "expensive" means takes a lot of CPU time.

Threads don't need to do this, just change a few pointers around, so it's much "cheaper" than creating a process. The reason threads don't need this is because they run in the address space, and virtual memory of the parent process.

Every process must have at least one thread. So if you think about it, creating a process means creating the process AND creating a thread. Obviously, creating only a thread will take less time and work by the computer.

In addition, threads are "lightweight" because threads can interact without the need of inter-process communication. Switching between threads is "cheaper" than switching between processes (again, just moving some pointers around). And inter-process communication requires more expensive communication than threads.