Synchronously Handling Asynchronous Posix Signals

...or...

Handling Asynchronous Posix Signals Synchronously

A Short Tutorial

A quick overview

- Signals are a limited form of inter-process communication used in Unix, Unix-like, and other POSIX-compliant operating systems. 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. Signals have been around since the 1970's Bell Labs Unix and have been more recently specified in the POSIX standard.
- When a signal is sent, the operating system interrupts the target process's normal flow of execution to deliver the signal. Execution can be interrupted during any non-atomic instruction. If the process has previously registered a signal handler, that routine is executed. Otherwise, the default signal handler is executed.

--Wikipedia

Rules of Posix Signals

- A signal mask exists for each thread and is controlled with pthread_sigmask().
  - pthread_sigmask(SIG_BLOCK, sigset_t*); // Add signals in set to current thread's mask
  - pthread_sigmask(SIG_UNBLOCK, sigset_t*); // Remove signals in set from thread's mask
  - pthread_sigmask(SIG_SETMASK, sigset_t*); // Set thread's mask to set
- Signal masks (sigset_t) are manipulated with:
  - int sigemptyset(sigset_t *set);
  - int sigfillset(sigset_t *set);
  - int sigaddset(sigset_t *set, int signum);
  - int sigdelset(sigset_t *set, int signum);
  - int sigismember(const sigset_t *set, int signum);
- An application begins running with a empty signal mask.
- All threads inherit their parent's signal mask.

Signal Delivery

- sigaction() is used to set function callbacks for each signal at the application level.
- When a posix signal is delivered to an application, the kernel selects a thread whose signal mask does not block that signal, interrupts that process at any non-atomic operation and runs the signal callback in the thread's context.
- The signal callback's stack is literally placed on top of the host thread's stack wherever the thread was interrupted. Thus, signal handlers are REENTRANT.
- It is unsafe to call any non-reentrant system call (almost all of them) from within a signal handler callback.

Signal safety and EINTR

- Any system call may return EINTR if it is interrupted by a signal.
- To safely handle signal interruption, every system call **should** be wrapped as so:

  ```
  int rc;

  do { rc = system_call(...); } while (rc == EINTR);
  ```

- Do you wrap every system call like such?
- Do your vendor libraries?

Solutions

- Block signals for all threads but one.
  - This prevents system calls from being interrupted.
- That solves the EINTR problem, but how about the restriction on system calls in the signal handler callback?
  - sigwaitinfo()

sigwaitinfo()

- int sigwaitinfo(const sigset_t *set, siginfo_t *info);
- sigwaitinfo() suspends execution of the calling thread until one of the signals in set is pending (If one of the signals in set is already pending for the calling thread, sigwaitinfo() will return immediately.)
- sigwaitinfo() removes the signal from the set of pending signals and returns the signal number as its function result. If the info argument is not NULL, then the buffer that it points to is used to return a structure of type siginfo_t (see sigaction(2)) containing information about the signal.
- If multiple signals in set are pending for the caller, the signal that is retrieved by sigwaitinfo() is determined according to the usual ordering rules; see signal(7) for further details.

Psignals

```
extern "C"

int _main(int argc, char *argv[])

{
    namespace kps = ktl::psignals;
    kps::this_thread::fill_mask();

    // Spawn threads here

    kps::sigset signals{ SIGTERM, SIGINT, SIGHUP };
    const kps::signum_t signum = kps::wait(signals);

    std::cout << "Received signal: " << signum << std::endl;

    kps::this_thread::clear_mask();
    return 0;
}
```