Signals 5: fork, exec, Pthreads

1. Overview
2. System Calls
3. Signal Handlers
4. Synchronization
5. **Signals and fork, exec, Pthreads**
   - **signals across fork**
   - **signals across exec**
   - **signals and Pthreads**
6. Advanced Topics
7. Signal Handler Issues

Signals Across fork/exec

Since signals are often used in **multi-process** programs, it is critical to understand what happens to signal disposition across calls to `fork()` and `exec()`.

Consider, for example, that it is possible for a signal to be able to be delivered to a new process before the first statement in the new process can be executed.

Thus, the only way we could be guaranteed to have signal handling in place in time would be by setting it up *prior* to making the `fork()/exec()` call.

This would obviously also require that signal disposition remained unchanged across `fork()/exec()`.

Is this the case?

Signals Across fork

Here is the situation with signals and `fork()`:

- signal disposition (including handlers) is inherited
- signal mask is inherited
- pending signals are not inherited
- pending timers are not inherited

So signal disposition and blocking is unaffected by `fork()`.

The child will have the same settings as were in affect in the parent at the time of the `fork()`.

Signals Across exec

Here is the situation with signals and `exec()`:

- signals whose disposition is *catch* are reset to *default*
- signals whose disposition is default/ignore remain unchanged
- unspecified if this is true for ignored `SIGCHLD`,
  however Linux leaves it unchanged
- signal mask remains unchanged
- pending signals remain unchanged
- all pending timers are cleared

(Man page for `execve()` makes somewhat different claims!)

Signals and Pthreads

The signal model was developed long before **POSIX Threads**.

The interaction of signals with Pthreads can make it somewhat complex to use signals in **multithreaded programs**.

Nonetheless, it is possible to use signals in multithreaded programs.

In fact, the ability to have thread(s) dedicated to handling signals can be very useful.

When using signals in multithreaded programs , it is critical to understand that some aspects of signals apply **process-wide** and some are **thread-specific**.

E.g., while most older signal syscalls apply **process-wide**, several signal-related `pthread_` syscalls are **thread-specific**.

Signals and Pthreads (contd.)

Here are key points about signals and threads:

- **process-wide** vs. **thread-specific**:
  - most signals are *process-wide* by default
  - **synchronous signals** (e.g., `SIGFPE`) are *thread-specific*
  - `kill()` delivers a signal to an entire *process*
  - `raise()` delivers a signal to the calling *thread*
  - `pthread_kill()` and `pthread_sigqueue` will deliver a signal to a specific Pthread
  - target thread must be in same process as sender
  - there is no way to signal a single thread in another process
  - **timers** are *process-wide* resources (so shared by all threads)

Signals and Pthreads (contd.)

key points about signals and threads (contd.):

- signal **blocking**:
  - **signal masks** (signals being **blocked**) are a *per-thread* attribute
  - this means signal *blocking* is *thread-specific*
  - use `pthread_sigmask()` to set *signal masks* in multithreaded programs
  - in multithreaded programs, the behavior of `sigprocmask()` is *undefined*
  - a newly created thread *inherits a copy* of its creator's *signal mask*
  - the set of **pending signals** for the new thread is *empty*
  - `sigpending()` gives the set of pending signals for the calling *thread* only
  - this is the *union* of pending thread-specific and process-directed signals

Signals and Pthreads (contd.)

key points about signals and threads (contd.):

- signal **disposition**:
  - signal **disposition** is a *per-process* attribute
  - this means all threads will must have the *same disposition* for each signal
  - if a delivered signal's disposition is *termination*, the *entire process* (all threads) is terminated
- signal **handling**:
  - if a *process-wide* signal is *caught*, one thread is *randomly chosen* to run the handler
  - **synchronous signal handling** is better with Pthreads
  - approach: block signal(s) in all threads, then call `sigwait()` in *"handler thread"*

Pthreads Signal Syscalls

Signal-related syscalls that are specifically for use with *Pthreads*:

- `pthread_sigmask` – set signal mask for a thread (`sigprocmask()`'s behavior is undefined in multithreaded program)
- `pthread_kill` – send signal to one thread (in same process) (`kill()` sends signal to entire process)
- `pthread_sigqueue` – queue a signal and data to a thread in process (`sigqueue()` sends signal to entire process)

Synchronous Signal Handling

In multithreaded programs that must respond to particular signals, *asynchronous* signal handlers are generally *not* the best approach.

It is better to use **synchronous signal handling**:

- *block* signals to be handled in all threads
- create a thread for signal handling
  (or can create separate thread for each signal)
- have the handling thread(s) use `sigwait()` (or related) to set the thread up to respond to its particular signal(s)
- when `sigwait()` returns, the handling thread carries out the actions that would traditionally have been in handler functions

Example: Asynchronous Signal Handler

Classic *asynchronous* SIGINT handler example:

```
int main(void)
{
  struct sigaction act;
  memset(&act,0,sizeof(act));
  act.sa_handler = sigint_handler;
  sigemptyset(&act.sa_mask);
  sigaction(SIGINT,&act,NULL);  //Set SIGINT to be caught

  ...main program code...

  return EXIT_SUCCESS;
}

void sigint_handler(int sig)
{
  ...handler actions...
  return;
}
```

Example: Synchronous Signal Handling

Thread-based *synchronous* SIGINT handler example:

```
int main(void)
{
  sigset_t mask;
  sigemptysetset(&mask);
  sigaddset(&mask, SIGINT);  //Block SIGINT for entire process
  sigprocmask(SIG_SETMASK, &mask, NULL);

  pthread_t tid;
  pthread_create(&tid, NULL, sigint_handler_thread, NULL);

  ...main program code...
  return EXIT_SUCCESS;
}

void *sigint_handler_thread(void *ignore)
{
  sigset_t catching;
  sigemptyset(&catching);
  sigaddset(&catching, SIGINT);
  int sig;
  while(1) {
    sigwait(&catching, &sig);  //Wait for SIGINT (to be pending)
    ...handler actions...
  }
  return NULL;
}
```

sigwait() and related

Three syscalls for synchronous signal handling with threads:

- `int sigwait(const sigset_t *set, int *sig)`
- `int sigwaitinfo(const sigset_t *set, siginfo_t *info)`
- `int sigtimedwait(const sigset_t *set, siginfo_t *info, const struct timespec *timeout)`

Operation:

- suspend execution of the calling thread until one of the signals specified in `set` becomes *pending*
- *accept* the signal (removing from pending) and return
- if a signal in `set` is pending when called, return immediately
- if multiple signals in `set` are pending, retrieved signal is determined by usual ordering rules (see "`man 7 signal()`")

sigwait() and related (contd.)

`sigwait()`:

- passes signal number back in `sig`
- returns 0 on success, else a positive error number

`sigwaitinfo()`:

- passes `siginfo_t` structure describing the signal back in `info`
- returns signal number on success, else -1 with `errno` set

`sigtimedwait()` is just like `sigwaitinfo()` except:

- `timeout` argument specifies max time for which the thread can be suspended waiting for a signal