Signals 4: Synchronization

1. Overview
2. System Calls
3. Signal Handlers
4. **Synchronization**
   - **IPC vs. synchronization**
   - **synchronization: pause() and sigsuspend()**
   - **synchronous handling: sigwait^{+}()**
5. Signals and fork, exec, Pthreads
6. Advanced Topics
7. Signal Handler Issues

IPC and Process Synchronization

A signal informs a process that some event has occurred, so if one process sends a signal to another process, information is transmitted, allowing a very simple form of IPC.

Often, this is all that is needed, as when one process needs to inform another that it has completed certain operations.

The `SIGUSR?` signals are particularly appropriate for this purpose.

It is common for such IPC to be used for **synchronization**—to have one process wait to proceed beyond a certain point in its code until the other informs it that certain (precondition) code has been completed.

Signals are useful for synchronization because there are calls that will cause a process to *suspend itself* until an appropriate signal is received.

Synchronization: pause and sigsuspend

`pause` suspends a process/thread until a signal is received:

```
int pause(void)
```

`sigsuspend` does the same thing, but temporarily replaces the process' *signal mask* before suspending the process:

```
int sigsuspend(const sigset_t *mask)
```

A key requirement for a signal to unpause/unsuspend a process is that *the signal's disposition must be one of two things*:

1. terminate (the process)
2. catch (invoke a handler function)

Because of this requirement, it is not uncommon to define and register an "*empty*" handler function (i.e., function does nothing).

This allows the signals to be used with `pause()`/`sigsuspend()` to unsuspend a process and continue running.

Synchronization: pause and sigsuspend (contd.)

The added functionality of `sigsuspend()` in changing the signal mask allows control over what signals might unsuspend the program, by changing the set of signals that are blocked while suspended.

(It also automatically restores the mask in effect prior to the call when the process is unsuspended.)

Thus, `sigsuspend()` has similar functionality to doing:

```
sigprocmask(SIG_SETMASK,&mask,&savemask);
pause();
sigprocmask(SIG_SETMASK,&savemask,NULL);
```

The problem with the above sequence of three calls is that they are not carried out as an **atomic operation**—like `sigsuspend()`.

This can lead to problems with **race conditions**.

Synchronization: pause and sigsuspend (contd.)

With the three-call sequence, you could unblock a signal that you want to unsuspend the process, but then have that signal get delivered before `pause()` gets executed.

This would result in the program *hanging*, because it would be suspended waiting for a signal that has already been delivered.

This is a *race condition*, because whether the problem occurs or not will depend on the exact relative timing of the instructions.

Race conditions are a serious problem that programmers must consider when writing programs that involve *asychronous events* like signals.

Synchronous Handling: sigwait+

Asynchronously interrupting a process' normal execution to run a signal handler function has drawbacks, such as requiring handler code be reentrant, potentially interupting syscalls, and so forth.

We just saw how `sigsuspend()` can be used to have a process simply stop running and wait for a signal to be received, effectively meaning that normal execution does not get interrupted.

This approach to signal interaction with a program is often described as **synchronous signal handling**.

While `sigsuspend()` can be used to implement synchronous signal handling, we saw that this requires defining a handler function, which is often empty.

Because of this, three new system calls were added to SUS to make it easier to use signals to *synchronously* accept signals.

Synchronous Handling: sigwait$^+$ (contd.)

The three calls all allow a process or a thread to suspend itself and wait for any of a *set* of signals before proceeding.

This is similar to what can be done with `sigsuspend()` and *empty handlers*.

The three calls are:

- `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)`

Synchronous Handling: sigwait$^+$ (contd.)

These functions are typically used by *blocking all signals* (using `sigprocmask()`) and then *waiting for particular signals*.

The functions all return when one of the waited-for signals is *generated and becomes pending* (but not delivered since blocked).

(The pending signal is cleared before the function returns.)

The functions provide similar functionality, differing as follows:

- `sigwait` – signal is placed in `sig` parameter
- `sigwaitinfo` – signal is return value, stores info in `siginfo_t` (struct) parameter
- `sigtimedwait` – like `sigwaitinfo` but can limit time to wait

Synchronous Handling: sigwait$^+$ (contd.)

Example of waiting for the SIGUSR signals:

```
//Block all signals:
sigset_t blocked;
sigfillset(&blocked);
sigprocmask(SIG_SETMASK,&blocked,NULL);

//Set up signals to wait for:
sigset_t waiting;
sigemptyset(&waiting);
sigaddset(&waiting,SIGUSR1);
sigaddset(&waiting,SIGUSR2);

//Wait for and dispatch on SIGUSR? signal:
while(1) {
  switch(sigwaitinfo(&waiting,NULL) {
    case SIGUSR1:
      ...
    case SIGUSR2:
      ...
  }
}
```

Synchronous Handling: sigwait$^+$ (contd.)

The sigwait$^+$ calls are particularly useful with *threads:*

- they suspend only the *thread* they are called in
- signal masks can be set on a *per-thread* basis

(Signals and thread interactions are discussed further below.)

The calls also work with the **real-time signals** (see below).

Because they are "newish," the following feature test macro may be required to be able to use the sigwait$^+$ calls:

```
#define _POSIX_C_SOURCE 199309
```