What is a Signal?
- A software interrupt generated by the OS or processes
- In response to an asynchronous event
  - Exception or error. E.g.: Bad memory access, divide-by-zero
  - OS notification of an event. E.g.: Alarm, child process termination, async-I/O completion
  - Generated by the user/another process. E.g.: Ctrl-C, process stop, user-specified signal

Signal Names and Types
- Each signal has a name
  - Starts with SIG
  - E.g.: SIGALRM, SIGSEGV, SIGKILL
- Each signal name corresponds to a specific event type. Examples:
  - SIGCHLD: Child terminated
  - SIGSEGV: Segmentation fault
  - SIGKILL: Kill signal
  - SIGPIPE: Writing to a pipe with no readers
  - SIGINT: Interrupt key input (typically Ctrl-C)
Signal Generation and Delivery

- A signal is generated when an event occurs
  - E.g.: When user presses Ctrl-C
  - Could be generated by the OS or by another process
- Signal is delivered when the process takes action
  - Process must be running to get signal delivery
- Signal is pending between its generation and delivery
  - Only one instance of a signal remains pending
    - Multiple undelivered instances are combined

Generating Signals: `kill`

```c
int kill(pid_t pid, int sig);
```

- Sends a signal to another process
- Parameters
  - `pid`: Signal-receiving process
    - Sender should have appropriate permissions
  - `sig`: Signal name
  - `kill` command
    - `kill -s signal_name pid`
    - E.g.: `kill -s USR1 1234`

Handling Signals

- A process can take one of multiple actions
- Ignore:
  - Throw away the signal
  - Cannot ignore SIGKILL and SIGSTOP
- Block:
  - Delay the delivery of the signal
    - Unblocking delivers a pending signal
    - Cannot block SIGKILL and SIGSTOP
- Catch:
  - Set up a user function to be called when the signal is delivered

Default Action

- Depends on signal type
- Most signals result in process termination
- Some are ignored by default (e.g.: SIGCHLD)
**Blocking Signals**

- A signal can be blocked
  - Is not delivered to process
  - Remains in pending state
  - Delivered to process upon unblocking
- Why block signals?
  - To avoid being interrupted by unexpected signals
  - E.g.: in the middle of a critical section

**Blocking Signals: Signal Masks and Sets**

- Signal mask: Set of signals that are currently blocked by a process
- Manipulated using signal set (type sigset_t)
- Operations:
  - sigaddset: Add a signal to the set
  - sigdelset: Remove a signal from the set
  - sigismember: Check if a signal is in the set
  - sigemptyset: Clear all signals in the set
  - sigfillset: Set all signals in the set

**Manipulating Signal Masks**

```c
int sigprocmask(int how, sigset_t *set, sigset_t *old_set);
```

- Manipulates the signal mask (set of blocked signals)
- Parameters
  - how: How the signal mask would be modified
    - SIG_BLOCK: Add a set of signals for blocking
    - SIG_UNBLOCK: Unblock a set of signals
    - SIG_SETMASK: Set current signal mask to given signal set
  - set: Signal set to be used for manipulation
  - old_set: The old signal mask before modification

**Signal Blocking Example**

```c
sigset_t newsigset, oldsigset;
/* Add SIGINT to the new signal set */
sigemptyset(&newsigset);
sigaddset(&newsigset, SIGINT);
/* Add SIGINT to the set of blocked signals */
sigprocmask(SIG_BLOCK, &newsigset, &oldsigset);
/* Check if SIGQUIT is in the old signal mask */
if (sigismember(&oldsigset, SIGQUIT))
  printf("SIGQUIT already blocked\n");
```
Catching Signals

- A signal can be caught
  - A user-defined function is called upon signal delivery
- To catch a signal:
  - Define a signal handler function
  - Set up the signal to be caught and signal handler to be called
- Why catch signals?
  - May want user to take meaningful action for specific events
  - E.g.: do something when an alarm goes off

Signal Handler

- User-defined function called when signal is delivered
- Syntax: `void (*sa_handler)(int)`
  - Argument is set to the signal number being delivered
  - Does not allow passing other parameters
  - Realtime signal handler is more versatile
- Can also specify:
  - `SIG_DFL`: Default action
  - `SIG_IGN`: Ignore action

Catching Signals: `sigaction`

```c
int sigaction(int sig, struct sigaction *action, struct sigaction *old_action);
```

- Specifies the action to take for a signal
- Parameters
  - `sig`: Name of the signal
  - `action`: Action to be taken
  - `old_action`: The previous action associated with the signal

Specifying the Action

- `struct sigaction`
  - `sa_handler`: Signal handler
  - `sa_mask`: Additional signals to be blocked in the signal handler
  - `sa_flags`: Special flags and options
  - `sa_sigaction`: Realtime signal handler
- Only one of `sa_handler` and `sa_sigaction` can be specified
```c
/* Signal handler */
void myhandler(int signo) {
    printf("Received signal: %d\n", signo);
}
struct sigaction newact;
newact.sa_handler = myhandler; /* Set sig handler */
sigemptyset(&newact.sa_mask); /* No other signals to be blocked */
newact.sa_flags = 0; /* No special options */
/* Install the signal handler for SIGINT */
sigaction(SIGINT, &newact, NULL);
```

**Waiting for Signals**

- Signals can be used to wait for a specific event without busy waiting
- Approach 1: Set a signal handler for a specific signal and wait until the signal is caught
  - `pause`
  - `sigsuspend`
- Approach 2: Block a signal and wait for signal to be generated
  - `sigwait`

**Waiting for Signals: `pause`**

- Wait until a signal arrives
- Returns after the return of signal handler
- Have to check which signal arrived
- No way to ensure that signal does not arrive before calling `pause`
- Would like to:
  - Block the desired signal until `pause`
  - Unblock it immediately before calling `pause`

**Waiting for Signals: `sigsuspend`**

- Change signal mask and wait until a signal arrives
- Operations done atomically
- Signals in sigmask are blocked
- Signal mask used to atomically unblock the signal to be caught
- Remove desired signal from the signal mask
Waiting for Signals: `sigwait`

```c
int sigwait(sigset_t *set, int *signo);
```

- Waits for a set of signals
  - Returns when one of the signals becomes pending
  - Removes it from set of pending signals

**Parameters**
- `set`: Set of signals to wait for
- `signo`: Contains the signal number on return

**Main Differences:**
- No signal handler used
- Desired signals should be blocked before calling `sigwait`

Signal Safety Problems

- Signals are asynchronous
  - Can arrive in the middle of a function
  - Signal handler can be executed concurrently
- Some library functions are not async-signal safe
  - Use static/global data structures
  - Can result in data access conflicts
  - E.g.: `strtok`, `perror`
- Some blocking calls return prematurely when interrupted by signals
  - E.g.: `read`, `write`, etc.

Ensuring Signal Safety

- May need to restart interrupted library calls
- Use async-signal safe library calls inside signal handlers
- Block a signal before entering critical sections that may conflict with the signal handler
  - Similar to locking/unlocking
- Save and restore `errno` inside the signal handler

Signals Summary

- Signals: Software interrupts
- Signal generation: By OS or `kill` function
- Signal handling:
  - Default
  - Ignore
  - Catch with user-defined signal handler
  - Block/unblock
- Signal waiting
- Signal safety issues