# 6. Process Event Flow

- Monitoring Processes
- Signals
- Typical Applications
- Advanced Synchronization With Signals

# Motivating Example

## Shell Job Control

Monitoring stop/resume cycles of a child process

```
$ sleep 60
Ctrl-Z                          // Deliver SIGTSTP

// Shell notified of a state change in a child process (stopped)
[1]+  Stopped          sleep    // Recieved terminal stop signal
$ kill -CONT %1                 // Equivalent to fg
sleep                           // Resume process
Ctrl-C                          // Deliver SIGINT
                                // Terminate process calling _exit(0)

// Shell notified of a state change in a child process (exited)
$
```

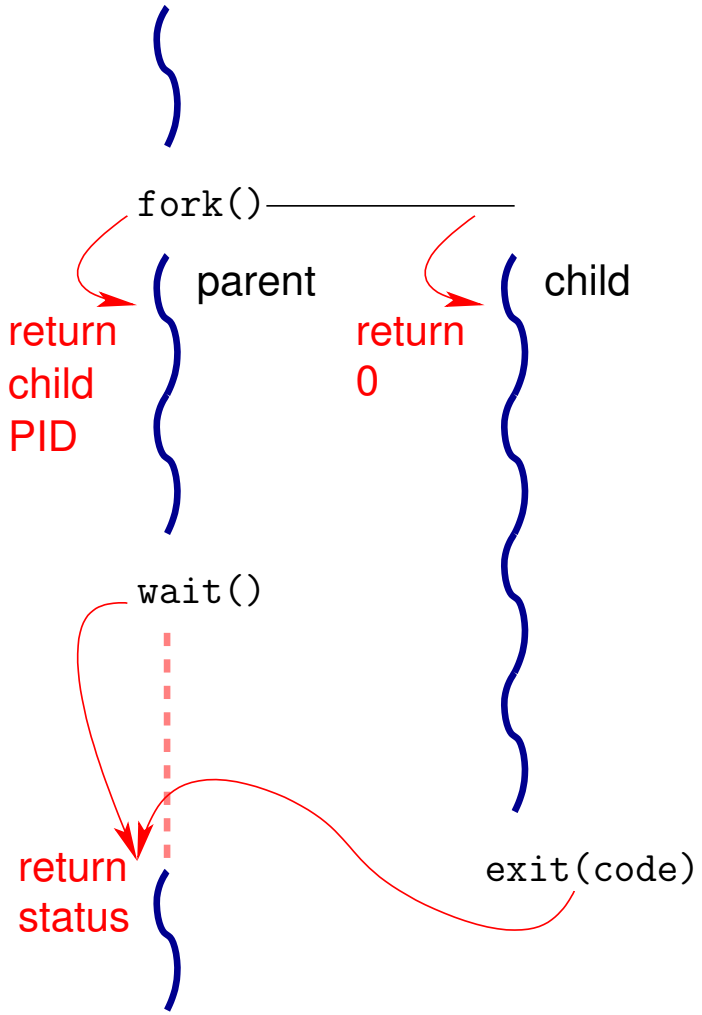
*How does this work?*

*Signal: most primitive form of communication (presence/absence)*

# 6. Process Event Flow

- Monitoring Processes
- Signals
- Typical Applications
- Advanced Synchronization With Signals

# Monitoring Processes



# System Call: `wait()` and `waitpid()`

## Wait For Child Process to Change State

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status_pointer);
pid_t waitpid(pid_t pid, int *status_pointer, int options);
```

## Description

- Monitor state changes and return PID of
  - Terminated child
  - Child stopped or resumed by a signal
- If a child terminates, it remains in a *zombie* state until `wait()` is performed to retrieve its state (and free the associated process descriptor)
  - Zombie processes do not have children: they are adopted by `init` process (**1**)
  - The `init` process always waits for its children
  - Hence, a zombie is removed when its parent terminates

# System Call: `wait()` and `waitpid()`

## Wait For Child Process to Change State

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status_pointer);
pid_t waitpid(pid_t pid, int *status_pointer, int options);
```

## Whom to Wait For

- `pid` $> 0$ : `waitpid()` suspends process execution until child specified by `pid` changes state, or returns immediately if it already did
- `pid` $= 0$ : wait for any child in the same process group
- `pid` $< -1$: wait for any child in process group `-pid`
- `pid` $= -1$: wait for any child process

## Short Cut

`wait(&status)` is equivalent to `waitpid(-1, &status, 0)`

# System Call: `wait()` and `waitpid()`

## Wait For Child Process to Change State

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status_pointer);
pid_t waitpid(pid_t pid, int *status_pointer, int options);
```

## How to Wait

- Option `WNOHANG`: do not block if no child changed state
  Return **0** in this case
- Option `WUNTRACED`: report stopped child
  (due to `SIGSTOP`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU` signals)
- Option `WCONTINUED`: report resumed child
  (due to `SIGCONT` signal)

# System Call: `wait()` and `waitpid()`

## Wait For Child Process to Change State

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status_pointer);
pid_t waitpid(pid_t pid, int *status_pointer, int options);
```

## State Change Status

- If non-NULL `status_pointer`, store information into the `int` it points to

  `WIFEXITED(status)`: true if child terminated normally (i.e., `_exit()`)

  `WEXITSTATUS(status)`: if the former is true, child exit status (lower **8** bits of `status`)

  `WIFSIGNALED(status)`: true if child terminated by signal

  `WTERMSIG(status)`: if the former is true, signal that caused termination

  `WIFSTOPPED(status)`: true if child stopped by signal

  `WSTOPSIG(status)`: if the former is true, signal that caused it to stop

  `WIFCONTINUED(status)`: true if child was resumed by delivery of `SIGCONT`

# System Call: `wait()` and `waitpid()`

## Wait For Child Process to Change State

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status_pointer);
pid_t waitpid(pid_t pid, int *status_pointer, int options);
```

## Error Conditions

- Return $-1$ if an error occurred
- Typical error code

  `ECHILD`, **calling** `wait()`: if all children were configured to be *unattended* (a.k.a. *un-waited for*, i.e., not becoming zombie when terminating, *see* `sigaction()`)

  `ECHILD`, **calling** `waitpid()`: `pid` is not a child or is *unattended*

# Process State Changes and Signals

## Process State Monitoring Example

```
int status;
pid_t cpid = fork();
if (cpid == -1) { perror("fork"); exit(1); }
if (cpid == 0) {                    // Code executed by child
  printf("Child PID is %ld\n", (long)getpid());
  pause();                          // Wait for signals
} else                              // Code executed by parent
  do {
    pid_t w = waitpid(cpid, &status, WUNTRACED | WCONTINUED);
    if (w == -1) { perror("waitpid"); exit(1); }
    if (WIFEXITED(status))          // Control never reaches this point
      printf("exited, status=%d\n", WEXITSTATUS(status));
    else if (WIFSIGNALED(status))
      printf("killed by signal %d\n", WTERMSIG(status));
    else if (WIFSTOPPED(status))
      printf("stopped by signal %d\n", WSTOPSIG(status));
    else if (WIFCONTINUED(status)) printf("continued\n");
  } while (!WIFEXITED(status) && !WIFSIGNALED(status));
```

# Process State Changes and Signals

## Running the Process State Monitoring Example

```
$ ./a.out &
Child PID is 32360
[1] 32359
$ kill -STOP 32360
stopped by signal 19
$ kill -CONT 32360
continued
$ kill -TERM 32360
killed by signal 15
[1]+  Done                    ./a.out
$
```

# 6. Process Event Flow

# Process Synchronization With Signals

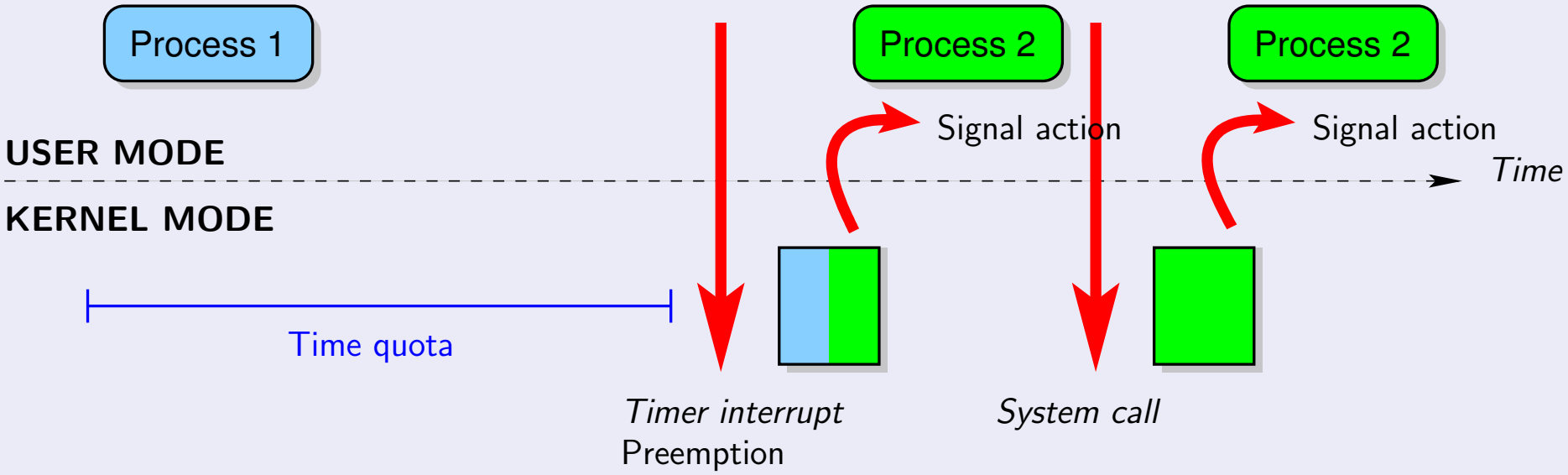
## Principles

- Signal *delivery* is *asynchronous*
  - Both sending and recieving are asynchronous
  - Sending may occur during the signaled process execution or not
  - Recieving a signal may interrupt process execution at an arbitrary point
- A signal *handler* may be called upon signal delivery
  - It runs in *user mode* (sharing the user mode stack)
  - It is called "*catching the signal*"
- A signal is *pending* if it has been delivered but not yet handled
  - Because it is currently *blocked*
  - Or because the kernel did not yet check for its delivery status
- *No queueing* of pending signals

# Process Synchronization With Signals

## Catching Signals

- Signal caught when the process *switches from kernel to user mode*
  - Upon context switch
  - Upon return from system call



# System Call: `kill()`

## Send a Signal to a Process or Probe for a Process

```
#include <sys/types.h>
#include <signal.h>

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

## Whom to Deliver the Signal

- `pid` $> 0$ : to `pid`
- `pid` $= 0$ : to all processes in the group of the current process
- `pid` $< -1$: to all processes in group $-$`pid`
- `pid` $= -1$: to all processes the current process has permitssion to send signals to, except himself and `init` (**1**)

# System Call: `kill()`

## Send a Signal to a Process or Probe for a Process

```
#include <sys/types.h>
#include <signal.h>

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

## Existence and Permission

- No signal sent if `sig` is **0**, but error checks are performed
- UID or EUID of the sender must match the UID or EUID of the reciever

## Error Conditions

- Return **0** on success, **−1** on error
- Possible `errno` codes

  `EINVAL`: an invalid signal was specified
  `EPERM`: no permission to send signal to any of the target processes
  `ESRCH`: the process or process group does not exist

# List of The Main Signals

`SIGHUP`[0]: terminal hang up

`SIGINT`[0]: keyboard interrupt (**Ctrl-C**)

`SIGQUIT`[0,1]: keyboard quit (**Ctrl-**)

`SIGKILL`[0,3]: unblockable kill signal, terminate the process

`SIGBUS/SIGSEGV`[0,1]: memory bus error / segmentation violation

`SIGPIPE`[0]: broken pipe (writing to a pipe with no reader)

`SIGALRM`[0]: alarm signal

`SIGTERM`[0]: termination signal (`kill` command default)

`SIGSTOP`[3,4]: suspend process execution,

`SIGTSTP`[4]: terminal suspend (**Ctrl-Z**)

`SIGTTIN/SIGTTOU`[4]: terminal input/output for background process

`SIGCONT`[2]: resume after (any) suspend

`SIGCHLD`[2]: child stopped or terminated

`SIGUSR1/SIGUSR2`[0]: user defined signal 1/2

**More signals:** `$ man 7 signal`

[0] terminate process

[1] dump a core

[2] ignored by default

[3] non-maskable, non-catchable

[4] suspend process

# System Call: signal()

## ISO C Signal Handling (pseudo UNIX V7)

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
// Alternate, "all-in-one" prototype
void (*signal(int signum, void (*handler)(int)))(int);
```

## Description

- Install a new handler for signal `signum`
  - `SIG_DFL`: default action
  - `SIG_IGN`: signal is ignored
  - Custom handler: function pointer of type `sighandler_t`
- Return the previous handler or `SIG_ERR`
- Warning: deprecated in multi-threaded or real-time code compiled with `-pthread` or linked with `-lrt`
  *Some of the labs need threads, use* `sigaction`

# System Call: signal()

## ISO C Signal Handling (pseudo UNIX V7)

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
// Alternate, "all-in-one" prototype
void (*signal(int signum, void (*handler)(int)))(int);
```

## When Executing the Signal Handler

- The `signum` argument is the caught signal number
- Blocks (defers) nested delivery of the signal being caught
- Asynchronous execution w.r.t. the process's main program flow
  - Careful access to global variables (much like threads)
  - Limited opportunities for system calls
    Explicit list of "safe" functions: `$ man 2 signal`

# System Call: `pause()`

## Wait For Signal

```
#include <unistd.h>

int pause();
```

## Description

- Suspends the process until it is delivered a signal
  - That terminate the process (`pause()` does not return...)
  - That causes a signal handler to be called
- Ignored signals (`SIG_IGN`) do *not* resume execution
  In fact, they never interrupt any system call
- Always return $-1$ with error code `EINTR`

# 6. Process Event Flow

# System Call: alarm()

## Set an Alarm Clock for Delivery of a SIGALRM

```
#include <unistd.h>

int alarm(unsigned int seconds);
```

## Description

- Deliver SIGALRM to the calling process after a delay (non-guaranteed to react immediately)
- Warning: the default action is to terminate the process

# System Call: alarm()

## C library function: sleep

```
unsigned int sleep(unsigned int seconds)
```

- Combines signal(), alarm() and pause()
- Uses the same timer as alarm() (hence, do not mix)
- See also setitimer()

## Putting the Process to Sleep

```
void do_nothing(int signum)
{
  return;
}
void my_sleep(unsigned int seconds)
{
  signal(SIGALRM, do_nothing); // Note: SIG_IGN would block for ever!
  alarm(seconds);
  pause();
  signal(SIGALRM, SIG_DFL);    // Restore default action
}
```

# More Complex Event Flow Example

## Shell Job Control

Monitoring stop/resume cycles of a child process

```
$ top
Ctrl-Z                          // Deliver SIGTSTP

[1]+  Stopped                 top   // Stop process
$ kill -CONT %1                     // Resume (equivalent to fg)
                                    // Recieve SIGTTOU and stop
[1]+  Stopped                 top   // Because of background terminal I/O
$ kill -INT %1
                                    // SIGINT is pending, i.e.
[1]+  Stopped                 top   //   did not trigger an action yet
$ fg
top
                                    // Terminate process calling exit(0)
$
```

# 6. Process Event Flow

- Monitoring Processes
- Signals
- Typical Applications
- Advanced Synchronization With Signals

# Advanced Synchronization With Signals

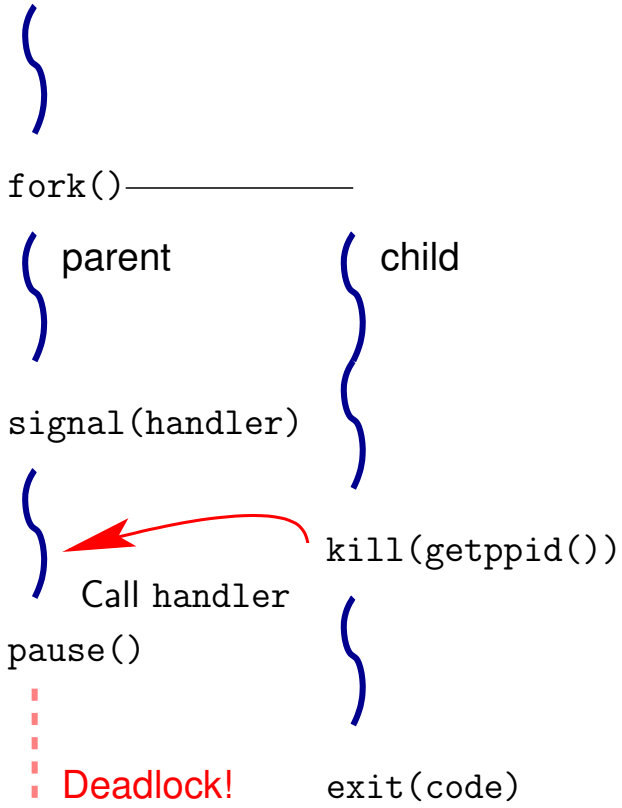
## Determinism and Atomicity

- ISO C (pseudo UNIX V7) signals are error-prone and may lead to uncontrollable run-time behavior: historical *design flaw*
  - Example: install a signal handler (`signal()`) before suspension (`pause()`)
  - What happens if the signal is delivered in between?

# Advanced Synchronization With Signals

## Determinism and Atomicity

- ISO C (pseudo UNIX V7) signals are error-prone and may lead to uncontrollable run-time behavior: historical *design flaw*
  - Example: install a signal handler (`signal()`) before suspension (`pause()`)
  - What happens if the signal is delivered in between?



# Advanced Synchronization With Signals

## Determinism and Atomicity

- ISO C (pseudo UNIX V7) signals are error-prone and may lead to uncontrollable run-time behavior: historical *design flaw*
  - Example: install a signal handler (`signal()`) before suspension (`pause()`)
  - What happens if the signal is delivered in between?
    Asynchronous signal delivery
    → Possible deadlock
    → Hard to fix the bug

# Advanced Synchronization With Signals

## Determinism and Atomicity

- ISO C (pseudo UNIX V7) signals are error-prone and may lead to uncontrollable run-time behavior: historical *design flaw*
  - Example: install a signal handler (`signal()`) before suspension (`pause()`)
  - What happens if the signal is delivered in between?
    Asynchronous signal delivery
    → Possible deadlock
    → Hard to fix the bug
- Solution: atomic (un)masking (a.k.a. (un)blocking) and suspension

# Advanced Synchronization With Signals

## Determinism and Atomicity

- ISO C (pseudo UNIX V7) signals are error-prone and may lead to uncontrollable run-time behavior: historical *design flaw*
  - Example: install a signal handler (`signal()`) before suspension (`pause()`)
  - What happens if the signal is delivered in between?
    Asynchronous signal delivery
    $\rightarrow$ Possible deadlock
    $\rightarrow$ Hard to fix the bug
- Solution: atomic (un)masking (a.k.a. (un)blocking) and suspension
- Lessons learned
  - Difficult to tame low-level concurrency mechanisms
  - Look for *deterministic* synchronization/communication primitives (enforce functional semantics)

# System Call: `sigaction()`

## POSIX Signal Handling

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
              struct sigaction *oldact);
```

## Description

- Examine and change the action taken by a process on signal delivery
- If `act` is not `NULL`, it is the new action for signal `signum`
- If `oldact` is not `NULL`, store the current action into the `struct sigaction` pointed to by `oldact`
- Return **0** on success, **−1** on error

# System Call: sigaction()

## POSIX Signal Handling

```
#include <signal.h>

int sigaction(int signum, const struct sigaction *act,
                          struct sigaction *oldact);
```

## Error Conditions

- Typical error code

  `EINVAL`: an invalid signal was specified, or attempting to change the action for `SIGKILL` or `SIGSTOP`
  Calling `sigaction()` with `NULL` second and third arguments and checking for the `EINVAL` error allows to check whether a given signal is supported on a given platform

# System Call: sigaction()

## POSIX Signal Action Structure

```
struct sigaction {
  void (*sa_handler)(int);
  void (*sa_sigaction)(int, siginfo_t*, void*);
  sigset_t sa_mask;
  int sa_flags;
}
```

## Description

**sa_handler:** same function pointer as the argument of signal() (it may also be set to SIG_DFL or SIG_IGN)

**sa_sigaction:** handler with information about the context of signal delivery (excusive with sa_handler, should not even be initialized if sa_handler is set)

**sa_mask:** mask of blocked signals when executing the signal handler

**sa_flags:** bitwise or of handler behavior options

# System Call: sigaction()

## POSIX Signal Action Structure

```
struct sigaction {
  void (*sa_handler)(int);
  void (*sa_sigaction)(int, siginfo_t*, void*);
  sigset_t sa_mask;
  int sa_flags;
}
```

SA_NOCLDSTOP: if signum is SIGCHLD, no notification when child processes stop (SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU) or resume (SIGCONT)

SA_NOCLDWAIT: if signum is SIGCHLD, "leave children *unattended*", i.e., do not transform terminating children processes into zombies

SA_SIGINFO: use sa_sigaction field *instead* of sa_handler

- siginfo_t parameter carries signal delivery context
- $ man 2 sigaction for (lengthy) details

**A few others:** reset handler after action, restart interrupted system call, etc.

# The sigsetops Family of Signal-Set Operations

```
$ man 3 sigsetops

#include <signal.h>

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);
```

## Description

- Respectively: empty set, full set, add, remove, and test whether a signal belong to the `sigset_t` pointed to by `set`
- The first four return **0** on success and **−1** on error
- `sigismember()` returns **1** if `signum` is in the set, **0** if not, and **−1** on error
- See also the non-portable `sigisemptyset()`, `sigorset()`, `sigandset()`

# Simple sigaction Example

```
int count_signal = 0;

void count(int signum) {
  count_signal++;
}

// ...

{
  struct sigaction sa;

  sa.sa_handler = count;                    // Signal handler
  sigemptyset(&sa.sa_mask);                 // Pass field address directly
  sa.sa_flags = 0;
  sigaction(SIGUSR1, &sa, NULL);

  while (true) {
    printf("count_signal = %d\n", count_signal);
    pause();
  }
}
```

# System Call: sigprocmask()

## Examine and Change Blocked Signals

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

## Semantics

- If `set` is not `NULL`, `how` describes the behavior of the call

  `SIG_BLOCK`: blocked $\leftarrow$ blocked $\cup$ `*set`
  `SIG_UNBLOCK`: blocked $\leftarrow$ blocked $-$ `*set`
  `SIG_SETMASK`: blocked $\leftarrow$ `*set`

- If `oldset` is not `NULL`, store the current mask of blocked signals into the `sigset_t` pointed to by `oldset`
- Return **0** on success, $-\mathbf{1}$ on error

# System Call: sigprocmask()

## Examine and Change Blocked Signals

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
```

## Remarks

- Unblockable signals: SIGKILL, SIGSTOP
  (attempts to mask them are silently ignored)
- Use sigsuspend() to unmask signals before suspending execution

# System Call: `sigpending()`

## Examine Pending Signals

```
#include <signal.h>

int sigpending(sigset_t *set);
```

## Semantics

- A signal is *pending* if it has been delivered but not yet handled, because it is currently blocked
  (or because the kernel did not yet check for its delivery status)
- Stores the set of pending signals into the `sigset_t` pointed to by `set`
- Return **0** on success, **−1** on error

# System Call: `sigsuspend()`

## Wait For a Signal

```
#include <signal.h>

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

## Semantics

- Perform the two following operations *atomically* w.r.t. signal delivery
  1. Set `mask` as the temporary set of masked signals
  2. Suspend the process until delivery of an *unmasked*, *non-ignored* signal
- When recieving a non-terminating, non-ignored signal, execute its handler, *and then*, *atomically* restore the previous set of masked signals and resume execution
- Always return $-\mathbf{1}$, typically with error code `EINTR`

# System Call: `sigsuspend()`

## Wait For a Signal

```
#include <signal.h>

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

## Typical Usage

- *Prevent early signal delivery between unmasking and suspension*
  1. Call `sigprocmask()` to disable a set of signals
  2. Perform some critical operation
  3. Call `sigsuspend()` to atomically enable some of them and suspend execution
- Without this atomic operation (i.e., with `signal()` and `pause()`)
  1. A signal may be delivered *between* the installation of the signal handler (the call to `signal()`) and the suspension (the call to `pause()`)
  2. Its handler (installed by `signal()`) may be triggered *before the suspension* (the call to `pause()`)
  3. Handler execution *clears the signal* from the process's pending set
  4. The suspended process deadlocks, waiting for an already-delivered signal

# Example With Signals and Memory Management

```
#include <stdio.h>
#include <signal.h>

struct sigaction sa;
char *p;

void catch(int signum) {        // Catch a segmentation violation
  static int *save_p = NULL;
  if (save_p == NULL) { save_p = p; brk(p+1); }
  else { printf("Page size: %d\n", p - save_p); exit(0); }
}

int main(int argc, char *argv[]) {
  sa.sa_handler = catch; sigemptyset(&sa.sa_mask); sa.sa_flags = 0;
  sigaction(SIGSEGV, &sa, NULL);
  p = (char*)sbrk(0);
  while (1) *p++ = 42;
}

$ page
Page size: 4096
```

# Command-Line Operations on Processes

- Cloning and executing
  `$ program arguments &`
- Joining (waiting for completion)
  `$ wait [PID]`
- Signaling events
  `$ kill [-signal] PID`
  `$ killall [-signal] process_name`
  Default signal `TERM` terminates the process
- `$ nohup`: run a command immune to *hang-up* (`HUP` signal)