Polling for Completion

If a parent process uses **`wait`** for a child process, then the parent process can't do other things

```
#include "csapp.h"

int main() {
  pid_t pid;
  pid = Fork();
  if (pid == 0) {
    Sleep(3);
  } else {
    int status;
    while (Wait(&status) != pid) {
      printf("Tick...\n");
      Sleep(1);
    }
  }
  return 0;
}
```

Copy

Polling for Completion

The **`WNOHANG`** option causes **`waitpid`** to always return immediately, but with `0` if a child process hasn't finished

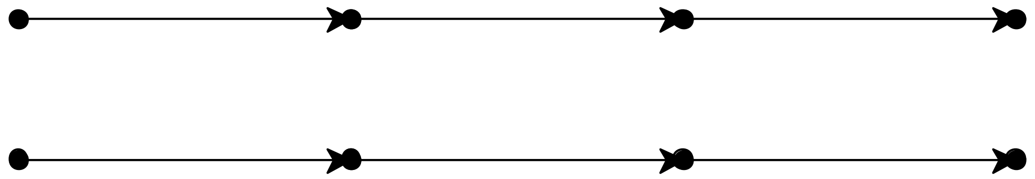
```
#include "csapp.h"

int main() {
  pid_t pid;
  pid = Fork();
  if (pid == 0) {
    Sleep(3);
  } else {
    int status;
    while (Waitpid(pid, &status, WNOHANG) != pid) {
      printf("Tick...\n");
      Sleep(1);
    }
  }
  return 0;
}
```

Copy

Signals

A ***signal*** is a general mechanism to push information to a process

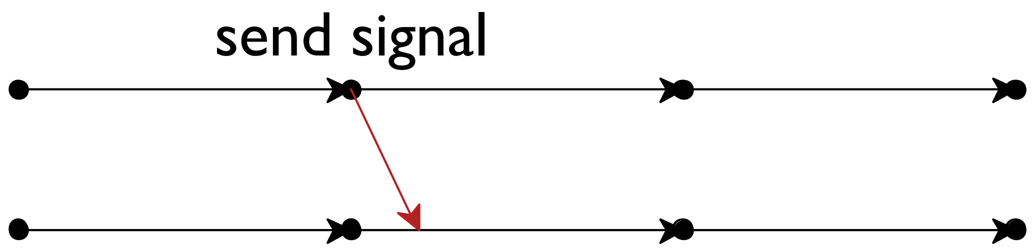
as opposed to *pulling* via syscall



Signals

A ***signal*** is a general mechanism to push information to a process

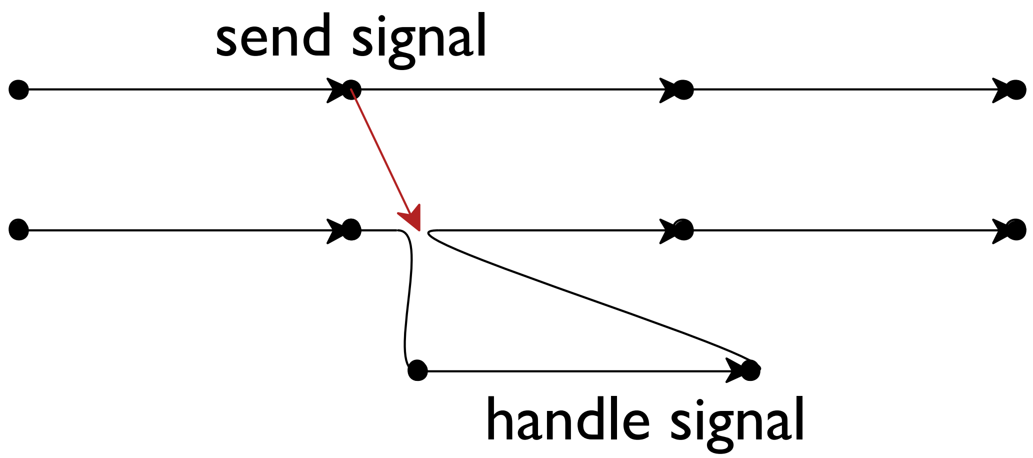
as opposed to *pulling* via syscall



Signals

A ***signal*** is a general mechanism to push information to a process

as opposed to *pulling* via syscall



`SIGINT`	=	`2`	Ctl-C
`SIGKILL`	=	`9`	sent by `kill -9`
`SIGALRM`	=	`14`	timer expired
`SIGTERM`	=	`15`	sent by `kill`
`SIGCHLD`	=	`20`	child state changed

Receiving Signals

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
```

Sets **`handler`** as the current process's handler for **`signum`**

Predefined handlers:

- **`SIG_IGN`** — ignore
- **`SIG_DFL`** — default, which is specific to **`signum`**

can't change handler for **`SIGKILL`**

Sending Signals

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

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

Sends **`signum`** to **`pid`**

Child process terminates	⇒	kernel sends **`SIGCHLD`**
Ctl-C in a shell	⇒	shell sends **`SIGINT`**
`$ kill` *pid*	⇒	**`kill`** sends **`SIGTERM`**
`$ kill -9` *pid*	⇒	**`kill`** sends **`SIGKILL`**

SIGCHLD Example

```
#include "csapp.h"

void done(int sigchld) {
  int status;
  Wait(&status);
  sio_puts("done\n");
}

int main() {
  Signal(SIGCHLD, done);

  if (Fork() == 0) {
    Sleep(3);
    exit(0);
  } else {
    while (1) {
      printf("Tick...\n");
      Sleep(1);
    }
  }
}
```

Copy

Signal Properties

- No extra data with a signal

  only info is that it happened

- A signal that is sent but not delivered is ***pending***

  delivered means handler is called; takes some time

- No queue: a signal is pending or not

  multiple sends before delivery ⇒ one pending

- A signal can be ***blocked*** to delay delivery

  send to blocked ⇒ stays pending until unblocked

- Each signal has a handler for delivery

Process State

signal	pending	blocked	handler
SIGHUP	0	0	**SIG_DFL**
SIGINT	1	1	**handle_ctl_c**
SIGQUIT	0	1	**handle_quit**
SIGILL	1	0	**give_up**
SIGTRAP	0	0	**SIG_DFL**
…			

- Kernel makes process call **give_up** with:

signal	pending	blocked	handler
SIGHUP	0	0	**SIG_DFL**
SIGINT	1	1	**handle_ctl_c**
SIGQUIT	0	1	**handle_quit**
SIGILL	0	1	**give_up**
SIGTRAP	0	0	**SIG_DFL**
…			

Process State

signal	pending	blocked	handler
SIGHUP	0	0	**SIG_DFL**
SIGINT	1	1	**handle_ctl_c**
SIGQUIT	0	1	**handle_quit**
SIGILL	1	0	**give_up**
SIGTRAP	0	0	**SIG_DFL**
…			

- Kernel makes process call **give_up** with:

signal	pending	blocked	handler
SIGHUP	0	0	[illegible]
SIGINT	1	1	[illegible]
SIGQUIT	0	1	**handle_quit**
SIGILL	0	1	**give_up**
SIGTRAP	0	0	**SIG_DFL**
…			

a delivered signal is blocked until handler returns

Process State

signal	pending	blocked	handler
`SIGHUP`	`0`	`0`	`SIG_DFL`
`SIGINT`	`1`	`1`	`handle_ctl_c`
`SIGQUIT`	`0`	`1`	`handle_quit`
`SIGILL`	`1`	`0`	`give_up`
`SIGTRAP`	`0`	`0`	`SIG_DFL`
...			

- Kernel makes process ca

other signals not blocked and may trigger nested handlers

signal	pending	b	handler
`SIGHUP`	`0`	`0`	`SIG_DFL`
`SIGINT`	`1`	`1`	`handle_ctl_c`
`SIGQUIT`	`0`	`1`	`handle_quit`
`SIGILL`	`0`	`1`	`give_up`
`SIGTRAP`	`0`	`0`	`SIG_DFL`
...			

Signal Mask

```
#include <signal.h>

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

A process's ***signal mask*** is a set of signals that are blocked

- **`how`** = **`SIG_BLOCK`**: add to mask
- **`how`** = **`SIG_UNBLOCK`**: remove from mask
- **`how`** = **`SIG_SETMASK`**: set mask

```
int sigemptyset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
```

Ctl-C Example

```
#include "csapp.h"

static void hit(int sigchld) {
  sio_puts("got Ctl-c\n");
}

int main() {
  sigset_t sigs;
  Sigemptyset(&sigs);
  Sigaddset(&sigs, SIGINT);

  Signal(SIGINT, hit);

  while (1) {
    // Sigprocmask(SIG_BLOCK, &sigs, NULL);
    Sleep(1);
    // Sigprocmask(SIG_UNBLOCK, &sigs, NULL);
    printf("Tick\n");
  }
}
```

Copy

Uncomment ⇒ Ctl-c only at tick

Multiple Ctl-C between ticks ⇒ only one printout

`sleep` in handler ⇒ cannot interrupt

Comment ⇒ signal interrupts `sleep`

Reacting to a Signal

```
#include "csapp.h"

static int child_running = 0;

void done(int sigchld) {
  int status;
  Wait(&status);
  child_running = 0;
}

int main() {
  Signal(SIGCHLD, done);
  while (1) {
    if (!child_running) {
      child_running = 1;
      if (Fork() == 0) {
        Sleep(3);
        printf("done\n");
        exit(0);
      }
    }
    printf("Tick...\n");
    Sleep(1);
  }
}
```

Copy

Signal handlers often set a global variable to communicate with the rest of the program

Reacting to a Signal

```c
#include "csapp.h"

static int child_running = 0;

void done(int sigchld) {
  int status;
  Wait(&status);
  child_running = 0;
}

int main() {
  Signal(SIGCHLD, done);
  while (1) {
    if (!child_running) {
      child_running = 1;
      if (Fork() == 0) {
        Sleep(3);
        printf("done\n");
        exit(0);
      }
    }
    printf("Tick...\n");
    Sleep(1);
  }
}
```

Copy

Signal handlers often set a global variable to communicate with the rest of the program

Bug: try removing parent **`printf`** and **`Sleep`**, and compile with **`-O2`**

Reacting to a Signal

```c
#include "csapp.h"

static int child_running = 0;

void done(int sigchld) {
  int status;
  Wait(&status);
  child_running = 0;
}

int main() {
  Signal(SIGCHLD, done);
  while (1) {
    if (!child_running) {
      child_running = 1;
      if (Fork() == 0) {
        Sleep(3);
        printf("done\n");
        exit(0);
      }
    }
    printf("Tick...\n");
    Sleep(1);
  }
}
```

Copy

Signal handlers often set a global variable to communicate with the rest of the program

Bug: try removing parent **`printf`** and **`Sleep`**, and compile with **`-O2`**

⇒ need **`volatile`** on **`child_running`**

Interacting with a signal handler is almost the only valid use for **`volatile`**

Signal Handlers are Concurrent

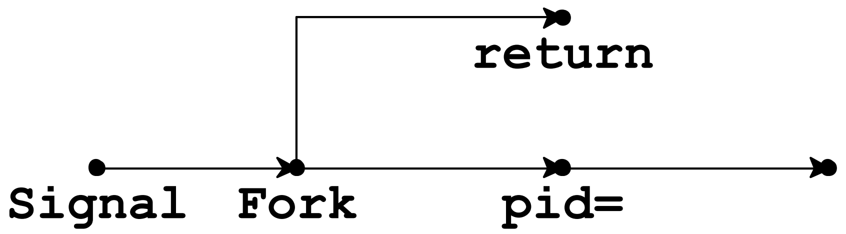
```
#include "csapp.h"

static volatile int child_running = 0;
static volatile pid_t pid = 0;

void done(int sigchld) {
  int status;
  Waitpid(pid, &status, 0);
  child_running = 0;
}

int main() {
  Signal(SIGCHLD, done);
  while (1) {
    if (!child_running) {
      child_running = 1;
      pid = Fork();
      if (pid == 0)
        return 0;
    }
  }
}
```

Copy

Signal Handlers are Concurrent

```c
#include "csapp.h"

static volatile int child_running = 0;
static volatile pid_t pid = 0;

void done(int sigchld) {
  int status;
  Waitpid(pid, &status, 0);
  child_running = 0;
}

int main() {
  Signal(SIGCHLD, done);
  while (1) {
    if (!child_running) {
      child_running = 1;
      pid = Fork();
      if (pid == 0)
        return 0;
    }
  }
}
```

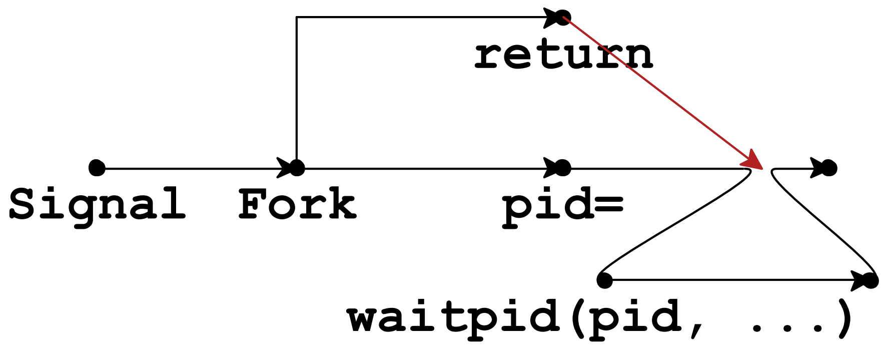
Copy



Signal Handlers are Concurrent

```
#include "csapp.h"

static volatile int child_running = 0;
static volatile pid_t pid = 0;

void done(int sigchld) {
  int status;
  Waitpid(pid, &status, 0);
  child_running = 0;
}

int main() {
  Signal(SIGCHLD, done);
  while (1) {
    if (!child_running) {
      child_running = 1;
      pid = Fork();
      if (pid == 0)
        return 0;
    }
  }
}
```

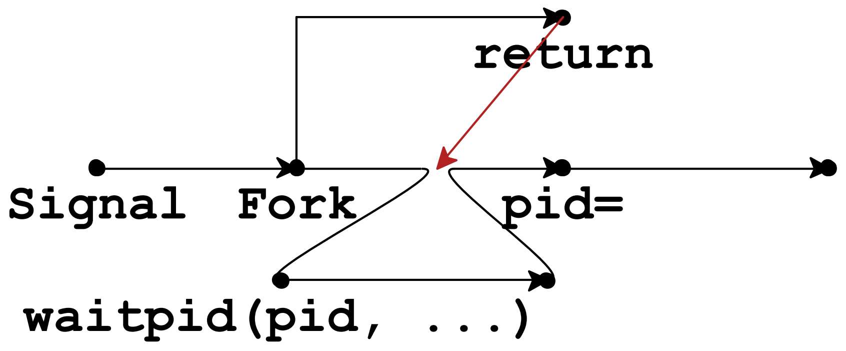
Copy



Signal Handlers are Concurrent

```
#include "csapp.h"

static volatile int child_running = 0;
static volatile pid_t pid = 0;

void done(int sigchld) {
  int status;
  Waitpid(pid, &status, 0);
  child_running = 0;
}

int main() {
  Signal(SIGCHLD, done);
  while (1) {
    if (!child_running) {
      child_running = 1;
      pid = Fork();
      if (pid == 0)
        return 0;
    }
  }
}
```

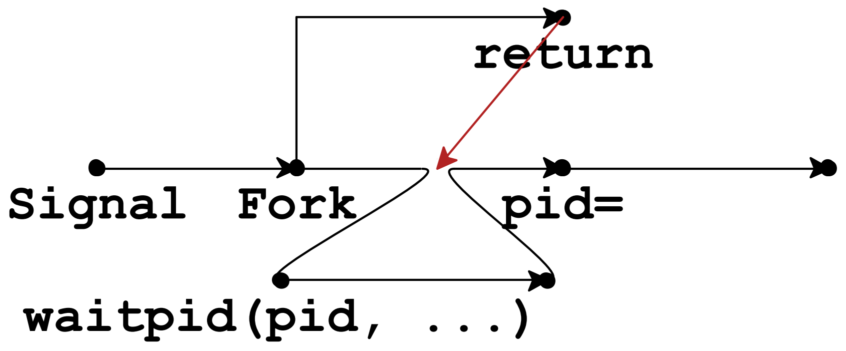
Copy



Signal Handlers are Concurrent

```c
#include "csapp.h"

static volatile int child_running = 0;
static volatile pid_t pid = 0;

void done(int sigchld) {
  int status;
  Waitpid(pid, &status, 0);
  child_running = 0;
}

int main() {
  Signal(SIGCHLD, done);
  while (1) {
    if (!child_running) {
      child_running = 1;
      pid = Fork();
      if (pid == 0)
        return 0;
    }
  }
}
```

Copy



Solution: ***block*** signals during **`pid = Fork()`**

Reliable `SIGCHLD` Handling

```
int main() {
  sigset_t sigs;
  Sigemptyset(&sigs);
  Sigaddset(&sigs, SIGCHLD);

  Signal(SIGCHLD, done);

  while (1) {
    if (!child_running) {
      child_running = 1;
      Sigprocmask(SIG_BLOCK, &sigs, NULL);
      pid = Fork();
      Sigprocmask(SIG_UNBLOCK, &sigs, NULL);
      if (pid == 0)
        return 0;
    }
  }
}
```

Copy

Async-Signal-Safe

```
 #include "csapp.h"

 static void ack(int sigchld) {
   printf("got alarm\n");
 }

 int main() {
   Signal(SIGALRM, ack);

   if (Fork() == 0) {
     while (1)
       Kill(getppid(), SIGALRM);
   } else {
     double a = 1.0;
     while (1) {
       printf("%f ", a);
       a = a + 1.0;
     }
   }
}
```

Copy

Async-Signal-Safe

```
#include "csapp.h"

static void ack(int sigchld) {
  printf("got alarm\n");
}

int main() {
  Signal(SIGALRM, ack);

  if (Fork() == 0) {
    while (1)
      Kill(getppid(), SIGALRM);
  } else {
    double a = 1.0;
    while (1) {
      printf("%f ", a);
      a = a + 1.0;
    }
  }
}
```

Copy

Eventually freezes

`printf` is not ***async-signal-safe***

Async-Signal-Safe

```
 #include "csapp.h"

 static void ack(int sigchld) {
   sio_puts("got alarm\n");
 }

 int main() {
   Signal(SIGALRM, ack);

   if (Fork() == 0) {
     while (1)
       Kill(getppid(), SIGALRM);
   } else {
     double a = 1.0;
     while (1) {
       printf("%f ", a);
       a = a + 1.0;
     }
   }
}
```

Copy

`sio_puts` from `csapp.c` uses only async-signal-safe functions

Signal Handlers and `errno`

```
#include "csapp.h"

static void ack(int sigchld) {
  /* broken; sets errno to ECHILD: */
  waitpid(getpid(), NULL, 0);
}

int main() {
  Signal(SIGALRM, ack);
  if (Fork() == 0) {
   while (1)
      Kill(getppid(), SIGALRM);
  } else {
    while (1) {
      /* broken; should set errno to ENOENT */
      open("not_there.txt", O_RDONLY);
      if (errno == ECHILD)
        printf("ECHILD from open?!\n");
    }
  }
}
```

Copy

Handler that makes syscalls implicitly shares **`errno`**

⇒ save **`errno`** in entry and restore **`errno`** on exit

Guidelines for Writing Safe Handlers

- Keep your handlers as simple as possible
- Call only async-signal-safe funcions in a handler
- Save and restore **`errno`** on entry and exit
- Declare shared variables as **`volatile`**
- Protect shared data by temporarily blocking all signals

Waiting for Signals

If you just need to wait for a child:

```
Waitpid(pid, &status, 0);
```

Wait for a Ctl-C? Or child, whichever happens first?

```
/* ... install handlers to set ctl_c_hit
       and child_finished ... */

while (!ctl_c_hit && !child_finished) { }
```

Busy waiting like that is too wasteful

`sigsuspend`

```
#include <signal.h>

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

Atomically sets the signal mask to **`mask`** and waits for a signal to be delivered

useful if **`mask`** unblocks some signals

Restores the signal mask before returning

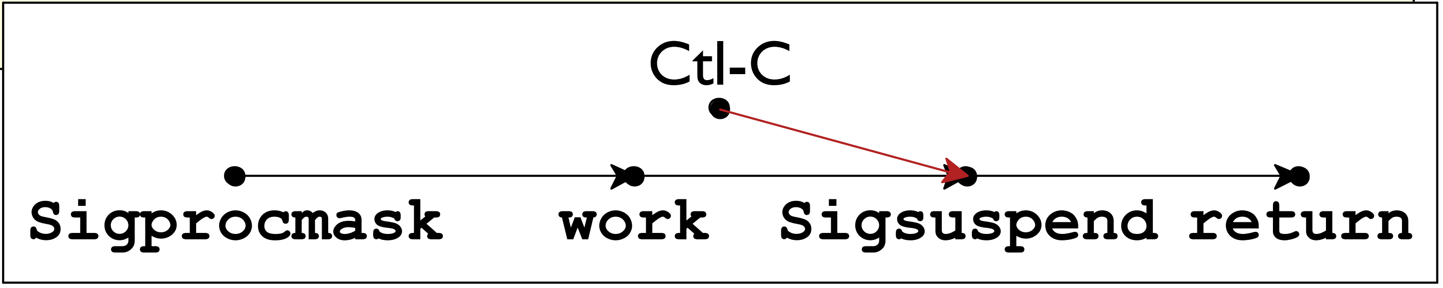
Using `sigsuspend`

```
#include "csapp.h"

static void hit(int sigchld) { sio_puts("got Ctl-c\n"); }
static void work() { Sleep(1); }  /* simulate useful work */

int main() {
  sigset_t sigs, empty_mask;
  Sigemptyset(&sigs); Sigemptyset(&empty_mask);
  Signal(SIGINT, hit);

  Sigaddset(&sigs, SIGINT);
  Sigprocmask(SIG_BLOCK, &sigs, NULL);

  work();
  Sigsuspend(&empty_mask);

  return 0;
}
```

Copy

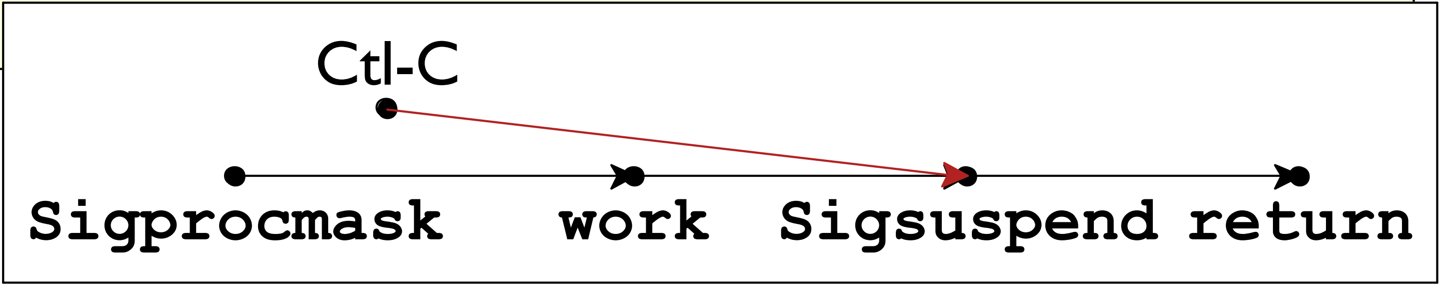
Using sigsuspend

```
#include "csapp.h"

static void hit(int sigchld) { sio_puts("got Ctl-c\n"); }
static void work() { Sleep(1); }  /* simulate useful work */

int main() {
  sigset_t sigs, empty_mask;
  Sigemptyset(&sigs); Sigemptyset(&empty_mask);
  Signal(SIGINT, hit);

  Sigaddset(&sigs, SIGINT);
  Sigprocmask(SIG_BLOCK, &sigs, NULL);

  work();
  Sigsuspend(&empty_mask);

  return 0;
}
```



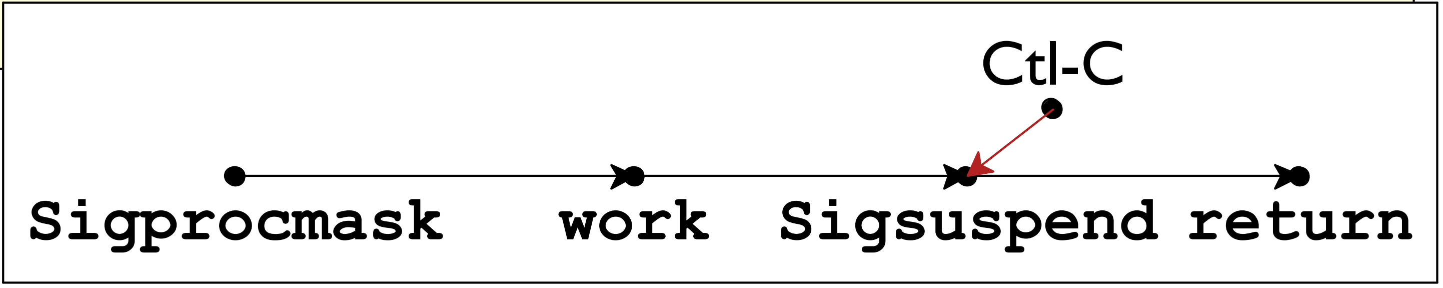
Using `sigsuspend`

```
#include "csapp.h"

static void hit(int sigchld) { sio_puts("got Ctl-c\n"); }
static void work() { Sleep(1); }  /* simulate useful work */

int main() {
  sigset_t sigs, empty_mask;
  Sigemptyset(&sigs); Sigemptyset(&empty_mask);
  Signal(SIGINT, hit);

  Sigaddset(&sigs, SIGINT);
  Sigprocmask(SIG_BLOCK, &sigs, NULL);

  work();
  Sigsuspend(&empty_mask);

  return 0;
}
```



Using `sigsuspend`

```
#include "csapp.h"

static void hit(int sigchld) { sio_puts("got Ctl-c\n"); }
static void work() { Sleep(1); }  /* simulate useful work */

int main() {
  sigset_t sigs, empty_mask;
  Sigemptyset(&sigs); Sigemptyset(&empty_mask);
  Signal(SIGINT, hit);

  Sigaddset(&sigs, SIGINT);
  Sigprocmask(SIG_BLOCK, &sigs, NULL);

  work();
  Sigsuspend(&empty_mask);

  return 0;
}
```



Misusing `sleep` as `sigsuspend`

Since **`sleep`** also returns on a signal:

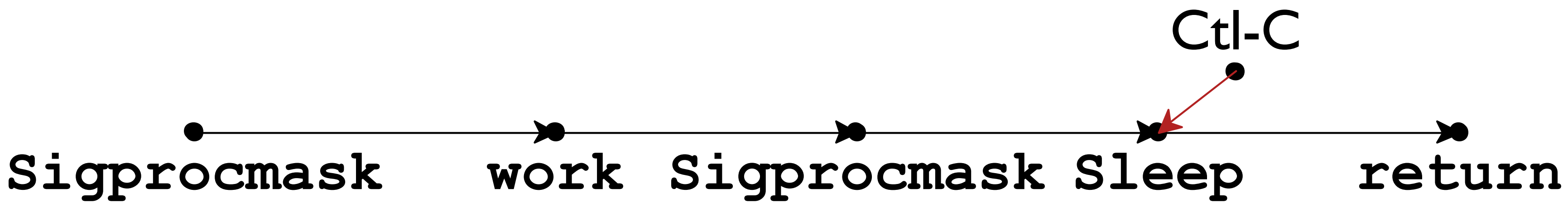
```
Sigaddset(&sigs, SIGINT);
Sigprocmask(SIG_BLOCK, &sigs, NULL);

work();

Sigprocmask(SIG_SETMASK, &empty_mask, NULL);
while (Sleep(1000) == 0) { }
```
Copy

Misusing `sleep` as `sigsuspend`

Since **`sleep`** also returns on a signal:

```
Sigaddset(&sigs, SIGINT);
Sigprocmask(SIG_BLOCK, &sigs, NULL);

work();

Sigprocmask(SIG_SETMASK, &empty_mask, NULL);
while (Sleep(1000) == 0) { }
```

Copy

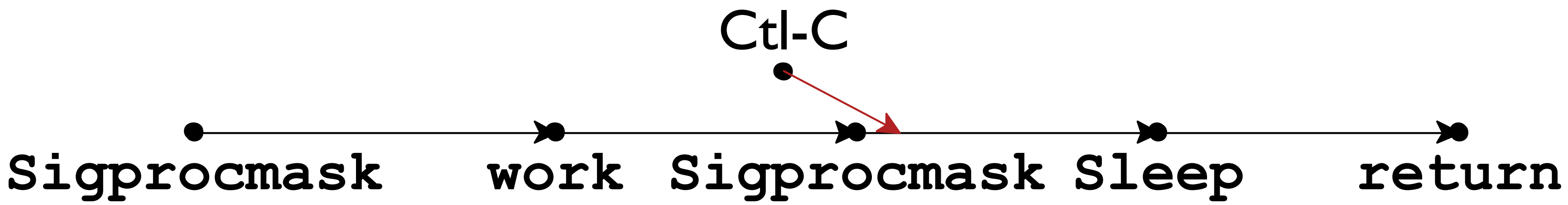


Misusing `sleep` as `sigsuspend`

Since **`sleep`** also returns on a signal:

```
Sigaddset(&sigs, SIGINT);
Sigprocmask(SIG_BLOCK, &sigs, NULL);

work();

Sigprocmask(SIG_SETMASK, &empty_mask, NULL);
while (Sleep(1000) == 0) { }
```

Copy

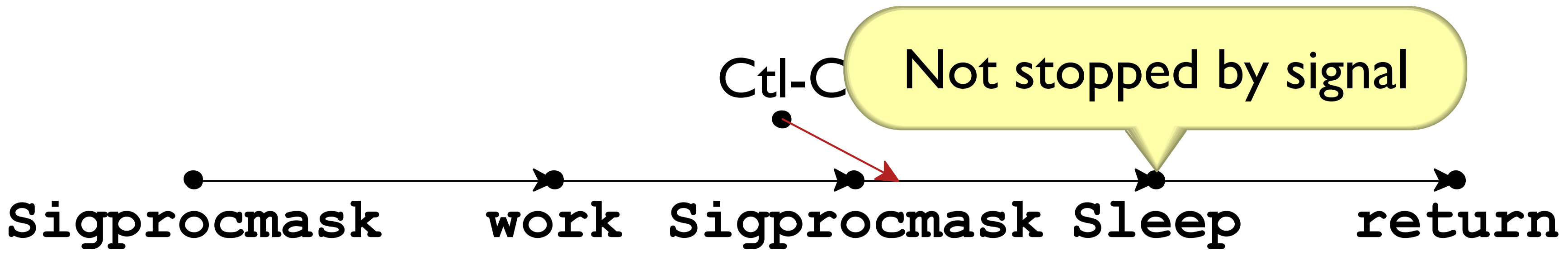


Misusing `sleep` as `sigsuspend`

Since **`sleep`** also returns on a signal:

```
Sigaddset(&sigs, SIGINT);
Sigprocmask(SIG_BLOCK, &sigs, NULL);

work();

Sigprocmask(SIG_SETMASK, &empty_mask, NULL);
while (Sleep(1000) == 0) { }
```

Copy



`sigprocmask` plus **`sleep`** is not ***atomic***

`pause` has the same problem

Guideline for Waiting for Signals

- Use **`sigsuspend`** to wait for signals
- Keep in mind that multiple signals may have happened
- Don't busy-wait

  because it takes CPU from useful work
- Don't use **`sleep`** or **`pause`** to wait for a signal

  because it doesn't work

Stopped Processes

In addition to

- ***running*** or
- ***terminated/zombie***

there's one more possible state for a process:

- ***stopped***

The special signals **`SIGSTOP`** and **`SIGCONT`** stop and continue a process, respectively

`signal` can't change **`SIGSTOP`** handler

A shell typically reacts to Ctl-Z by stopping a process

Technically, uses **`SIGTSTP`** instead of **`SIGSTOP`**

Stop and Continue in a Shell

```
$ /bin/cat
hi
hi
^Z
[1]+  Stopped                 /bin/cat
$ ps ax | grep /bin/cat
13388 pts/0    T      0:00 /bin/cat
13393 pts/0    S+     0:00 grep --color=auto /bin/cat
$ fg %1
/bin/cat
hi again
hi again
```

Stop and Continue in a Shell

```
$ /bi
hi
hi
^Z
[1]+  Stopped                        /bin/cat
$ ps ax | grep /bin/cat
13388 pts/0    T      0:00 /bin/cat
13393 pts/0    S+     0:00 grep --color=auto /bin/cat
$ fg %1
/bin/cat
hi again
hi again
```

Ctl-Z sends **`SIGTSTP`**

Stop and Continue in a Shell

```
$ /bin/cat
hi
hi
^Z
[1]+  Stopped                              t
$ ps ax | grep /b   /cat
13388 pts/0    T      0:00 /bin/cat
13393 pts/0    S+     0:00 grep --color=auto /bin/cat
$ fg %1
/bin/cat
hi again
hi again
```

T mean “stopped”

Stop and Continue in a Shell

```
$ /bin/cat
hi
hi
^Z
[1]+  Stopped                    /bin/cat
$ ps ax |
13388 pts                            at
13393 pt  /0     S+      0:00 grep --color=auto /bin/cat
$ fg %1
/bin/cat
hi again
hi again
```

fg %1 sends **SIGCONT**

Stop and Continue in a Shell

```
$ /bin/cat
hi
hi
^Z
[1]+  Stopped                 /bin/cat
$ ps ax | grep /bin/cat
13388 pts/0    T      0:00 /bin/cat
13393 pts/                        olor=auto /bin/cat
$ fg %1
/bin/cat
hi again
hi again
```

process continues running

Child Stop and Continue Signals

`SIGCHLD` is sent when a child is stopped or continued

By default, **`waitpid`** does not report stop or continue

- Use **`WUNTRACED`** option to get “stopped” reports

  detect with **`WIFSTOPPED(status)`**

- Use **`WCONTINUED`** option to get “continued” reports

  detect with **`WIFCONTINUED(status)`**

Shells and Process Groups

When a shell sends a signal, it sends it to a ***process group***

```
#include "csapp.h"

void hit(int sigchld) {
  static int hit_once = 0;
  if (hit_once) _exit(0);
  hit_once = 1;
  sio_puts("ignoring first Ctl-C\n");
}

int main() {
  Signal(SIGINT, hit);

  Fork();
  Fork();

  while (1) Pause();
}
```

Copy

Ctl-C ⇒ four messages

Negated group ID to **`kill`** sends to all processes in group

Setting a Process Group

```
#include <unistd.h>

int setpgid(pid_t pid, pid_t pgid);
```

Sets the process group of **`pid`** to **`gid`**

subject to many constraints

A 0 for **`pid`** or **`gid`** uses the current process's ID

`setpgid(0, 0)` ⇒ current process in a new group

Hiding Ctl-C from a Child Process

```
#include "csapp.h"

void hit(int sigchld) {
  static int hits = 0;
  if (++hits == 9) _exit(0);
}

static char *const argv[] = { "/bin/cat", NULL };

int main() {
  pid_t pid;
  Signal(SIGINT, hit);

  pid = Fork();
  if (pid == 0) {
    // Setpgid(0, 0);
    Execve(argv[0], argv, NULL);
  }

  Waitpid(pid, NULL, 0);
  return 0;
}
```

Copy

`Setpgid`

⇒ **`/bin/cat`**

survives Ctl-C