Signals

- Signals provide a simple method to transmit software interrupts to processes. They occur asynchronously when:
  - There is an error during the execution of a job.
  - Events created with the help if input devices (cntrl-z, cntrl-c, cntrl-\ etc.).
  - A process notifies another one about an event.
  - Issuing of a `kill` command to a job.
- Signals are identified with integer number.
  - a unique number represent a different type of signal.
- Signals provide a way to handle asynchronous events:
  a user at a terminal typing the interrupt key to suspend a program in execution.

Signals

- Signals take place at what appears to be "random time" to the process.
- We can ask the kernel to do one of the following things when a signal occurs:
  - Ignore the signal (two signals though can never be ignored: `SIGKILL` & `SIGSTOP`).
  - Catch the signal (we do that by informing the kernel to call a function of ours whenever a signal occurs).
  - Let the default action apply (every signal has a default action)

Some of the POSIX Signals

		Action	
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGBUS	7	Core	Bus error (bad memory access)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers
SIGALRM	14	Term	Timer signal from alarm(2)
SIGTERM	15	Term	Termination signal
SIGUSR1	10	Term	User-defined signal 1
SIGUSR2	12	Term	User-defined signal 2
SIGCHLD	17	Ign	Child stopped or terminated
SIGCONT	18	Cont	Continue if stopped
SIGSTOP	19	Stop	Stop process
SIGTSTP	20	Stop	Stop typed at tty
SIGTTIN	21	Stop	tty input for background process
SIGTTOU	22	Stop	tty output for background process

Actions

The "*Action*" column above specifies the *default disposition* for each (how the process behaves when it is delivered the signal):

- `Term`: Default action is to **terminate** the process.
- `Ign`: Default action is to **ignore** the signal.
- `Core`: Default action is to terminate the process & **dump-core**.
- `Stop`: Default action is to **stop** the process.
- `Cont`: Default action is to **continue** the process if it is currently stopped.

• If any of the signals is used, the header file `<signal.h>` must be included.

Sending a signal with `kill`

- `kill [ -signal ] pid ...`

  `kill [ -s signal ] pid ...`

  send a specific signal to process(es)

```
kill -USR1 3424
kill -s USR1 3424
kill -9 3424
```

- `kill -l [signal]`: lists all available signals

```
antoulas@sazerac:~/Set004$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
antoulas@sazerac:~/Set004$
```

Sending a signal to a process through the `kill` call

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

- Signal `sig` is sent to process with `pid`
- `#include <sys/types.h>`
  `#include <signal.h>`
- Should the receiving and dispatching processes belong to the same user or the dispatching process is the superuser the signal can be successfully sent.
- If `sig` is 0 then no signal is dispatched.
- On success (at least one signal was sent), zero is returned. On error, -1 is returned, and `errno` is set appropriately.

The `signal()` system call

- ```
  #include <signal.h>
  typedef void (*sighandler_t)(int);
  sighandler_t  signal(int signum, sighandler_t handler);
  ```
- The `signal()` call installs a new signal handler for the signal with number `signum`.
  - The signal handler is set to `handler`.
  - `handler` may be user-specified function, or `SIG_IGN`, or `SIG_DFL`.
- `signal()` returns the previous value of the signal handler, or `SIG_ERR` on error.
- This call is the traditional way of handling signals.
  ```

Example

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

void f(int);

int main(){
  int i;

  signal(SIGINT, f);
  for(i=0;i<5;i++){
    printf("hello\n");
    sleep(1);
    }
}

void f(int signum){   /* no explicit call to function f          */
  signal(SIGINT, f);  /* re-establish disposition of the signal SIGINT  */
  printf("OUCH!\n");
}
```

```
antoulas@sazerac:~/src$ ./a.out
hello
hello
^COUCH!
hello
hello
^COUCH!
hello
^COUCH!
antoulas@sazerac:~/src$
```

Ignoring a Signal

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

int main(){
  int i;
  signal(SIGINT, SIG_IGN);
  printf("you can't stop me here! \n");
  while(1){
    sleep(1);
    printf("haha \n");
    }
}   /* use cntrl-\ to get rid of this process */
```

```
antoulas@sazerac:~/Desktop/Set004/src$ ./a.out
you can't stop me here!
haha
haha
haha
^Chaha
haha
haha
^Ghaha
haha
^Chaha
haha
haha
^\Quit
antoulas@sazerac:~/Desktop/Set004/src$
```

The pause(), raise() calls

```
int pause(void);
```

- #include <unistd.h>
- causes the *invoking process or thread* to sleep until a signal is received that either terminates it (i.e., process of thread) or causes it to call a signal-handler.
- returns when a signal was caught and the signal-handling function returned. In this case pause returns -1, and errno is set to EINTR.

```
int raise(int sig);
```

- #include <signal.h>
- sends sig to the invoking process; it is equivalent to: kill(getpid(), sig);
- returns 0 on success, non-zero for failure.

The alarm call

```
unsigned int alarm(unsigned int seconds);
```

- `#include <unistd.h>`
- delivers a `SIGALRM` to invoking process in *seconds*.
- any previously set `alarm()` is cancelled.
- returns the number of seconds remaining until any previously scheduled alarm was due to be delivered; otherwise, 0.

```c
#include <stdio.h>
#include <unistd.h>

main(){
    alarm(3); // schedule an alarm signal
    printf("Looping for good!\n"); fflush(stdout);
    while (1) ;
    printf("This line should be never part of the output\n"); fflush(stdout);
    }
```

```
antoulas@sazerac:~/src$ date; ./a.out ; date
Mon Apr 12 22:20:41 EEST 2010
Looping for good!
Alarm clock
Mon Apr 12 22:20:44 EEST 2010
antoulas@sazerac:~/src$
```

Example with signal, alarm and pause

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

void  wakeup(int);

main(){
  printf("about to sleep for 5 seconds \n");
  signal(SIGALRM, wakeup);

  alarm(5);
  pause();  /* pauses the process until a sig arrives */
  printf("Hola Amigo! Un abrazo!\n");
  }

void wakeup(int signum){
  printf("Alarm received from kernel\n");
  }
```

```
antoulas@sazerac:~/src-set004$ ./signal-alarmpause
about to sleep for 5 seconds
Alarm received from kernel
Hola Amigo! Un abrazo!
antoulas@sazerac:~/src-set004$
```

Unreliable Signals – a headache in "older" UNIX

```
    int sig_int();
    ....
    signal(SIGINT, sig_int());
    ...

sig_int(){
    /* this is the point of possible problems */
    signal(SIGINT,sig_int);
    ...
    }
```

1. After a signal has occurred but before the call to `sig_int` is in the signal handler body, another signal may occur!
2. The second signal would cause the default action: this may force the process to *terminate*.
3. A unsuccessful effort is to (re-)state the signal's expected disposition as the *1st line* of the handler..
4. Although this may occassionally appear to work correctly, the mechanism is not "bullet–proof" as we may "lose" a signal along the way.

Unreliable Signals

- A process could “ignore” signals (with a trick):

```
int my_sig_flag=0;
    ...
main(){
    int my_sig_int();
    ...
    signal(SIGINT, my_sig_int);
    ...
    while (my_sig_flag == 0){
                /* point with possible problem in here */
                pause();
                }
    ...
}

my_sig_int(){
    signal(SIGINT, my_sig_int);
    my_sig_flag=1;
    }
```

◇ Under “regular” circumstances the process would “pause” until it received a `SIGINT` and then, it continue on to other actions past the `while` statement; the `while` predicate would disqualify.

Unreliable Signals

There is a small chance that things would go wrong...

1. If the signal takes place **after** the predicate evaluation **but before** the call to *pause*, the process could go on to sleep for ever! (provided that another signal is not generated)
2. In the above scenario the signal is *lost*!
3. Such code is not correct yet it works most of the times...

Unreliable Signal-ing

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <signal.h>

void foohandler(int);
int flag=0;

int main(){
  int lpid=0;
  printf("The process ID of this program is %d \n",getpid());
  lpid=getpid();
  signal(SIGINT, foohandler);
  while (flag==0){
        kill(lpid, SIGINT);     /* lost signal?? */
        printf("flag is %d\n", flag); fflush(stdout);
        pause();
        printf("Hello! \n");
        }
}

void foohandler(int signum){   /* no explicit call to handler foo */
  signal(SIGINT, foohandler);  /* re-establish handler for next time */
  flag=1;
}
```

Unreliable signal-ing

• Running the program, we get into the *pause* (the first signal does not appear to get into handler):

```
antoulas@sazerac:~/src-set004$ ./signal-exampleA
The process ID of this program is 22792
flag is 1
```

The (first) signal *seems* to be "lost" for the time being..

• Forcing now an interrupt with *control-C*, we terminate the program (by getting out of the loop):

```
antoulas@sazerac:~/src-set004$ ./signal-exampleA
The process ID of this program is 22792
flag is 1
^CHello!
antoulas@sazerac:~/src-set004$
```

⊙ *Signal Sets* provide a (*POSIX*) reliable way to deal with signals.

POSIX Signal Sets

- Signal sets are defined using the type `sigset_t`.
- Sets are large enough to hold a representation of *all* signals in the system.
- We may indicate interest in specific signals by empty-ing a set and then add-ing signals or by using a full set and then by selectively delete-ing certain signals.
- Initialization of signals happens through:
  - `int sigemptyset(sigset_t *set);`
  - `int sigfillset(sigset_t *set);`
- Manipulation of signals sets happens via:
  - `int sigaddset(sigset_ *set, int signo);`
  - `int sigdelset(sigset_ *set, int signo);`
- Membership in a signal set:
  - `int sigismember(sigset_t *set, int signo)`

Example in creating different Signal sets

```
#include <signal.h>

sigset_t mask1, mask2;
...
...
sigemptyset(&mask1);                    // create an empty mask
...
sigaddset(&mask1, SIGINT);              // add signal SIGINT
sigaddset(&mask1, SIGQUIT);             // add signal SIGQUIT
...
sigfillset(&mask2);                     // create a full mask
...
sigdelset(&mask2, SIGCHLD); // remove signal SIGCHLD
....
...
```

- mask1 is created entirely empty.
- mask2 is created entirely full.

sigaction() call

- Once a set has been defined, we can elect a specific method to handle a signal with the help of sigaction().

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

- The sigaction structure is:

```
struct sigaction{
      // action to be taken
  void (*sa_handler)(int);
      // additional signals to be blocked
      // during the handling of the signal
  sigset_t sa_mask;
      // flags controlling handler invocation
  int   sa_flags;
      // pointer to a signal handler in applications;
  void (*sa_sigaction)(int, siginfo_t *, void *);
  }
```

Elements of the `sigaction` structure (a)

- `sa_handler` field: identifies the action to be taken when the signal `signo` is received (previous slide)
  1. `SIG_DFL`: restores the system's default action
  2. `SIG_IGN`: ignores the signal
  3. The address of a function which takes an int as argument. The function will be executed when a signal of type `signo` is received and the value of `signo` is passed as parameter. Control is passed to function as soon as signal is received and when function returns, control is passed back to the point at which the process was interrupted.
- `sa_mask` field: the signals specified here will be blocked during the execution of the `sa_handler`.

Elements of the `sigaction` structure (b)

- `sa_flags` field: used to modify the behavior of `signo` – the originally specified signal.
  1. A signal's action is reset to `SIG_DFL` on return from the handler by `sa_flags=SA_RESETHAND`
  2. Extra information will be passed to signal handler, if `sa_flags=SIG_INFO`. Here, `sa_handler` is redundant and the final field `sa_sigaction` is used.
- Use either `sa_handler` or `sa_sigaction` but not both!

Use of sigaction

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

void catchinterrupt(int signo){
    printf("\nCatching: signo=%d\n",signo);
    printf("Catching: returning\n");
    }

main(){
    static struct sigaction act;

    act.sa_handler=catchinterrupt;
    sigfillset(&(act.sa_mask));

    sigaction(SIGINT, &act, NULL);

    printf("sleep call #1\n");
    sleep(1);
    printf("sleep call #2\n");
    sleep(1);
    printf("sleep call #3\n");
    sleep(1);
    printf("sleep call #4\n");
    sleep(1);
    printf("Exiting \n");
    exit(0);
    }
```

Regardless of where the program is interrupted, it resumes execution & carries on

```
antoulas@sazerac:~/Set004/src$ ./a.out
sleep call #1
sleep call #2
^C
Catching: signo=2
Catching: returning
sleep call #3
^C
Catching: signo=2
Catching: returning
sleep call #4
^C
Catching: signo=2
Catching: returning
Exiting
antoulas@sazerac:~/Set004/src$
```

```
antoulas@sazerac:~/Set004/src$ ./a.out
sleep call #1
sleep call #2
^C
Catching: signo=2
Catching: returning
sleep call #3
sleep call #4
Exiting
antoulas@sazerac:~/Set004/src$
```

Changing the behavior of program in interrupt

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

main(){
    static struct sigaction act;

    act.sa_handler=SIG_IGN;   // the handler is set to IGNORE
    sigfillset(&(act.sa_mask));

    sigaction(SIGINT, &act, NULL);   // control-c
    sigaction(SIGTSTP, &act, NULL);  // control-z

    printf("sleep call #1\n"); sleep(1);
    printf("sleep call #2\n"); sleep(1);
    printf("sleep call #3\n"); sleep(1);

    act.sa_handler=SIG_DFL;  // reestablish the DEFAULT behavior
    sigaction(SIGINT, &act, NULL); // default for control-c

    printf("sleep call #4\n"); sleep(1);
    printf("sleep call #5\n"); sleep(1);
    printf("sleep call #6\n"); sleep(1);

    sigaction(SIGTSTP, &act, NULL);  // default for control-z
    printf("Exiting \n");
    exit(0);
    }
```

Running the Program...

```
antoulas@sazerac:~/Set004/src$ ./a.out
./a.out
sleep call #1
^Csleep call #2
^Z^Csleep call #3
sleep call #4
sleep call #5
^Zsleep call #6
Exiting
antoulas@sazerac:~/Set004/src$ ./a.out
sleep call #1
sleep call #2
sleep call #3
sleep call #4
sleep call #5
^C
antoulas@sazerac:~/Set004/src$ ./a.out
sleep call #1
^Csleep call #2
^C^Z^Zsleep call #3
^Z^Zsleep call #4
^Z^Zsleep call #5
^Z^Zsleep call #6
^ZExiting
antoulas@sazerac:~/Set004/src$
```

Restoring a *previous* action

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

main(){
    static struct sigaction act, oldact;

    printf("Saving the default way of handling the control-c\n");
    sigaction(SIGINT, NULL, &oldact);
    printf("sleep call #1\n"); sleep(4);

    printf("Changing (Ignoring) the way of handling\n");
    act.sa_handler=SIG_IGN;   // the handler is set to IGNORE
    sigfillset(&(act.sa_mask));
    sigaction(SIGINT, &act, NULL);

    printf("sleep call #2\n"); sleep(4);

    printf("Reestablishing to old way of handling\n");
    sigaction(SIGINT, &oldact, NULL);
    printf("sleep call #3\n"); sleep(4);

    printf("Exiting \n");
    exit(0);
    }
```

Example in restoring a *previous* action

```
antoulas@sazerac:~/Set004/src$ ./a.out
Saving the default way of handling the control=c
sleep call #1
^C
antoulas@sazerac:~/Set004/src$
antoulas@sazerac:~/Set004/src$
antoulas@sazerac:~/Set004/src$
antoulas@sazerac:~/Set004/src$
antoulas@sazerac:~/Set004/src$ ./a.out
Saving the default way of handling the control=c
sleep call #1
Changing (Ignoring) the way of handling
sleep call #2
^C^C^C^C^C^C^C^C^C^C^CRestablishing to old way of handling
sleep call #3
^C
antoulas@sazerac:~/Set004/src$
```

Blocking Signals

- Occasionally, a program wants to *block* all together (rather than ignore) incoming signals
  - for instance, when updating a data segment in a database.

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

- `how` indicates what specific action `sigprocmask` should take:
  1. `SIG_SETMASK`: group of blocked signals is set to `set`
  2. `SIG_BLOCK`: set of blocked signals is the union of the current set and the `set` argument.
  3. `SIG_UNBLOCK`: signals in set are removed from the current set of blocked signals.
- If `oldset` is non-null, the previous value of signal mask is stored in `oldset`.
- If `set` is NULL, the signal mask is unchanged and current value of mask is returned in `oldset` (if it is not NULL);

Code snippet using sigprocmask()

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

main(){
    sigset_t set1, set2;

    sigfillset(&set1); // completely full set

    sigfillset(&set2);
    sigdelset(&set2, SIGINT);
    sigdelset(&set2, SIGTSTP);  // a set minus INT & TSTP

    printf("This is simple code... \n");
    sleep(5);
    sigprocmask(SIG_SETMASK, &set1, NULL);  // disallow everything here!

    printf("This is CRITICAL code... \n"); sleep(10);

    sigprocmask(SIG_UNBLOCK, &set2, NULL);  // allow all but INT & TSTP
    printf("This is less CRITICAL code... \n"); sleep(5);
    sigprocmask(SIG_UNBLOCK, &set1, NULL);  // unblock all signals in set1
    printf("All signals are welcome!\n");
    exit(0);
}
```

Working with the sigprocmask()

```
antoulas@sazerac:~/Set004/src$ ./a.out
This is simple code...
^C
antoulas@sazerac:~/Set004/src$ ./a.out
This is simple code...
This is CRITICAL code...
^Z^Z^C^C^X^X^C^C^Z^Z
This is less CRITICAL code...
^C
antoulas@sazerac:~/Set004/src$ ./a.out
This is simple code...
This is CRITICAL code...
^Z^C^Z^C^Z^C^Z
This is less CRITICAL code...
^\Quit
antoulas@sazerac:~/Set004/src$ fg
bash: fg: current: no such job
antoulas@sazerac:~/Set004/src$ ./a.out
This is simple code...
This is CRITICAL code...
This is less CRITICAL code...
All signals are welcome!
antoulas@sazerac:~/Set004/src$
antoulas@sazerac:~/Set004/src$
```

About Signals..

- When a signal is dispatched and the receiving process executes a sys-call, the signal has no effect until the sys-call compeletes.
  • *Exception*: a few calls such as `read/write/open` on slow devices could be interrupted by a signal.
- In general, signals cannot be stacked (ie, you can never have more than one signal of each type outstanding at any moment).
  • In this context, when it comes to reliability there are always some questions as a process can never be sure that a signal has not been "lost".
- The effectiveness of signals with respect to IPC is somewhat limited.