SUPSI

Inter-Process Communication

Operating Systems

Amos Brocco, Lecturer & Researcher

14 settembre 2018

Objectives

- Study common communication mechanisms

Browsing
- Get a rapid overview.

Reading
- Read it and try to understand the concepts.

Studying
- Read in depth, understand the concepts as well as the principles behind the concepts.

You are also encouraged to try out (compile and run) code examples!

Process communication: an example with POSIX signals

- POSIX signals are simple messages which can be exchanged between processes or sent by the operating system to a process.

- Signals are **asynchronous** events that interrupt the “normal” flow of execution:
  - Each signal has an **name** (ex. SIGKILL) and a **value** (ex. 9):
    - lower signal values have higher priority
  - A program can define a procedure (**signal handler**) which gets called when a particular signal is received
    - If there is no handler, a default action is executed

Examples of signals (man 7 signal)

Signal	Value	Action	Comment
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)
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	30,10,16	Term	User-defined signal 1
SIGUSR2	31,12,17	Term	User-defined signal 2
SIGCHLD	20,17,18	Ign	Child stopped or terminated
SIGCONT	19,18,25	Cont	Continue if stopped
SIGSTOP	17,19,23	Stop	Stop process
SIGTSTP	18,20,24	Stop	Stop typed at terminal
SIGTTIN	21,21,26	Stop	Terminal input for background process
SIGTTOU	22,22,27	Stop	Terminal output for background process
...			
SIGRTMIN	SIGRTMIN	Term	User-defined realtime signal
...			
SIGRTMAX	SIGRTMAX	Term	User-defined realtime signal

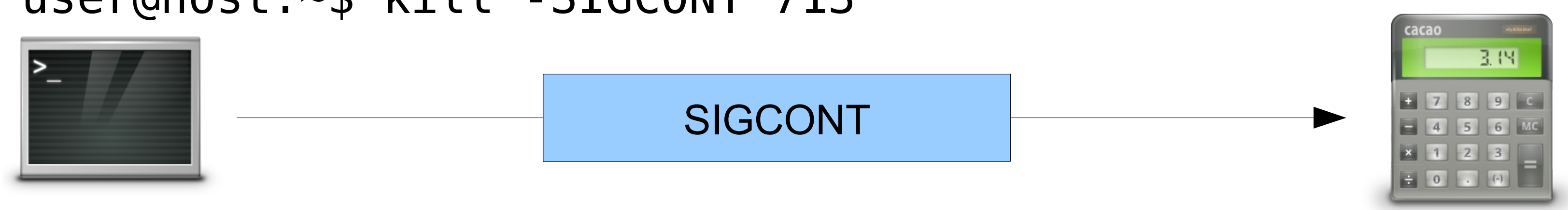
Example

With the shell command kill we can send signals to processes by PID

```
user@host:~$ kill -SIGSTOP 713
```

The process gnome-calculator (PID = 713) is stopped

```
user@host:~$ kill -SIGCONT 713
```



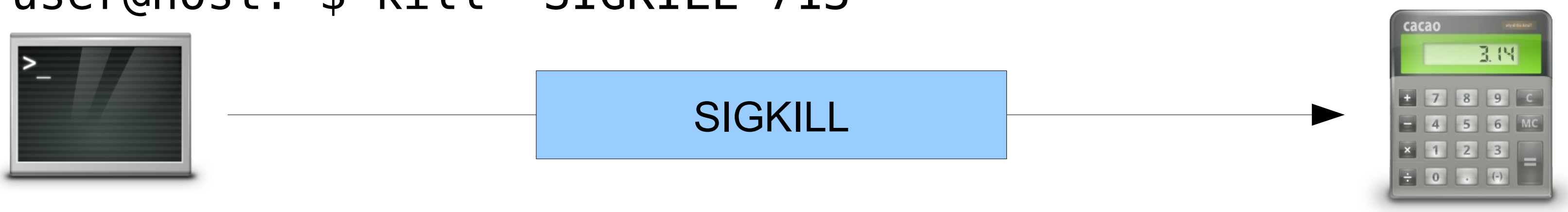
The process gnome-calculator (PID = 713) is resumed

Example

```
user@host:~$ kill -9 713
```

or

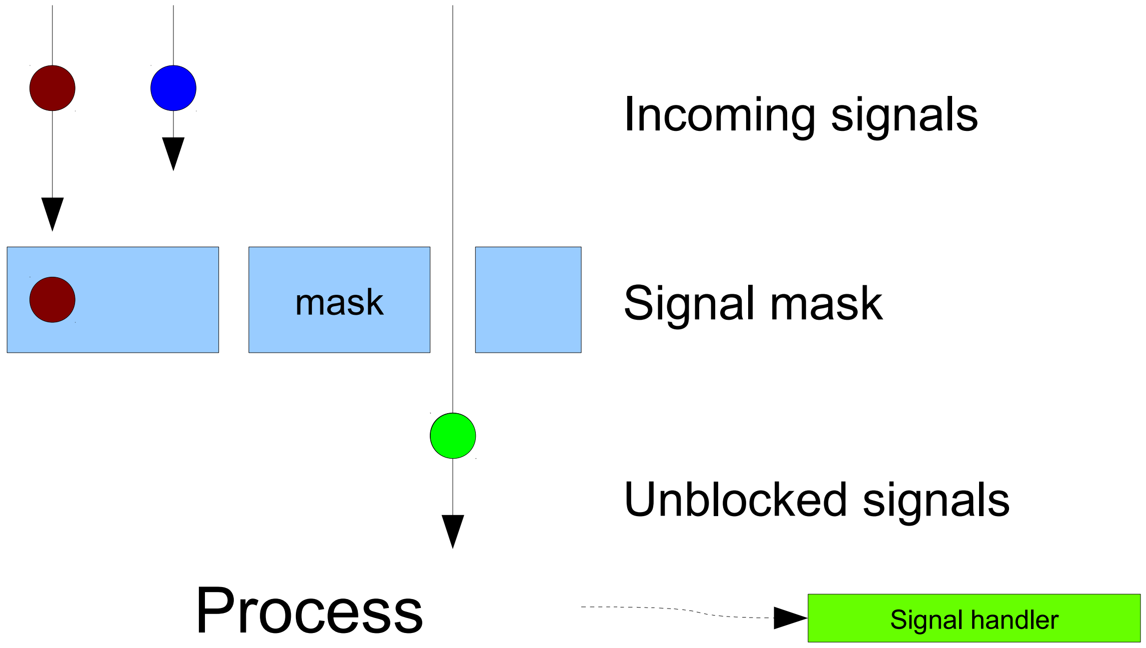
```
user@host:~$ kill -SIGKILL 713
```



The process gnome-calculator (PID = 713) is terminated

Block signals: the signal mask

- Each process has a **signal mask**
  - A process can add signals to the mask and prevent their handler from being executed (**blocked signals**), or remove signals from the mask to unblock them.



Signals which cannot be blocked

- **SIGKILL** and **SIGSTOP** can be neither intercepted (ignored or handled by the process) nor blocked.

Signal state

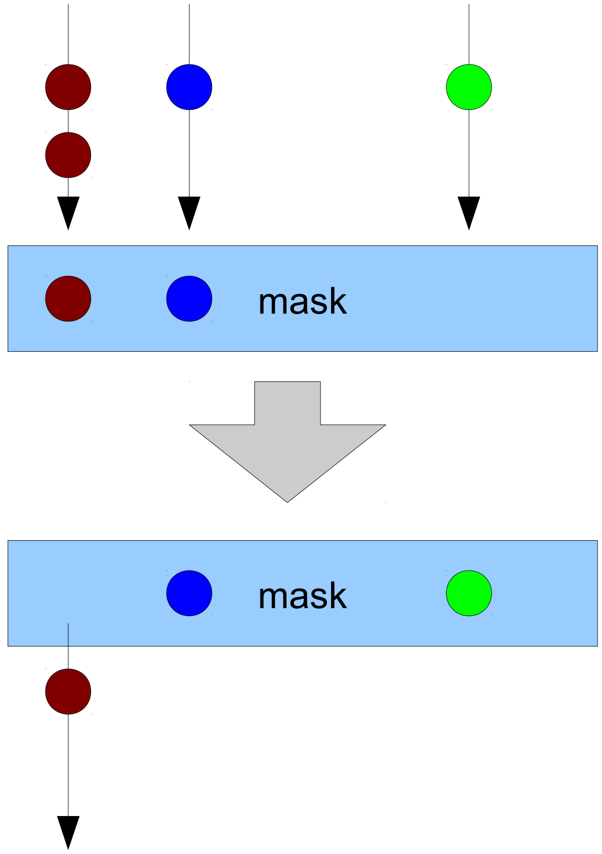
- **Pending**
  - Between generation and delivery
- **Blocked**
  - When delivery is prevented by the signal mask
    - *A blocked signal is still pending!*
- **Delivered**
  - The process executes the signal handler procedure

Signal types

- **Normal signals**
  - Simple numerical values
  - Cannot be stored
  - Delivery order not guaranteed

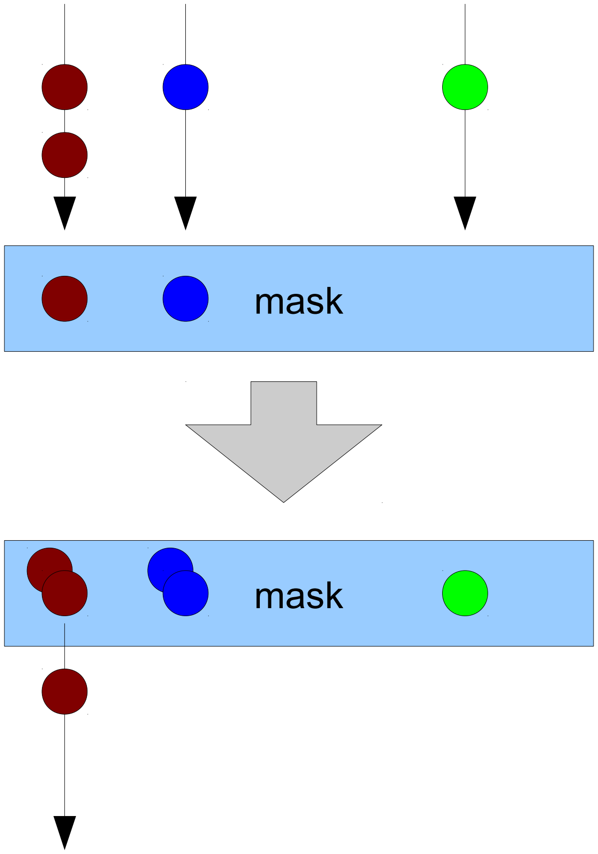
- **Realtime signals**
  - SIGRTMIN to SIGRTMAX
  - Priority is lower than normal signals
  - Can be queued (ordered queue)
  - Can have a small payload

Normal signals semantic



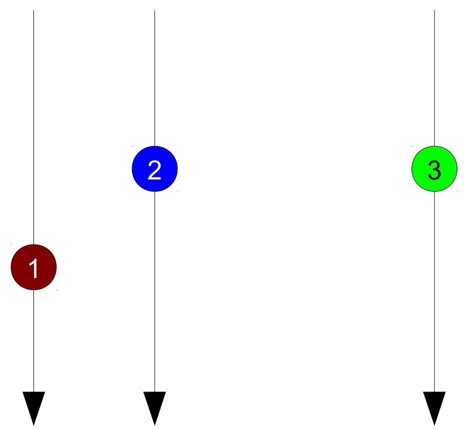
When a signal arrives it sets a flag on the mask: only one signal occurrence for each type is registered (pending or blocked). If the process receives more than one signal of the same type before the handler is executed, only the first occurrence is considered.

Realtime signals semantic



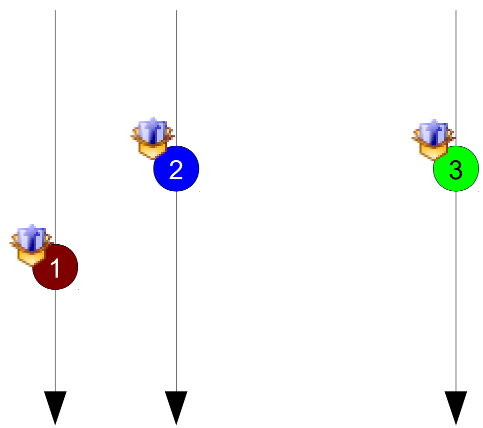
If the process receives multiple signals of the same type **they are queued and handled individually**.

Normal signals semantic



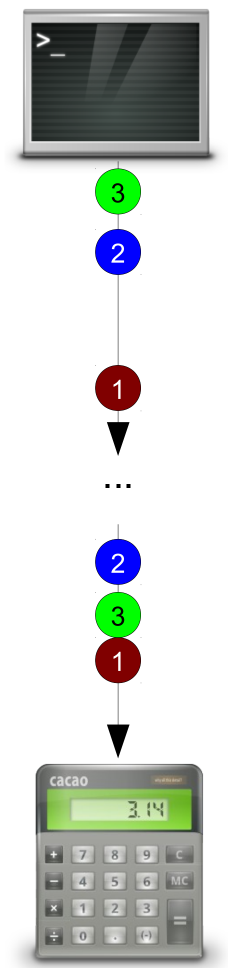
Normal signals are simple events (with a numerical value): no additional payload is available.

Realtime signals semantic



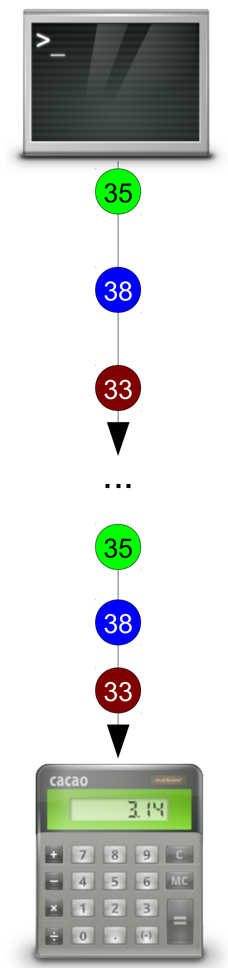
Realtime signals **can carry a small additional payload (information)**.

Normal signals semantic



Normal signals delivery order **is not guaranteed**.

Realtime signals semantic



Delivery order matches dispatch order. If more than one realtime signal is queued, **the delivery order is determined by the priority of the signal (SIGRTMIN being the highest realtime priority)**. If the process is handling a signal, only another signal of higher priority can interrupt it. *

* Normal signals all have the same priority, which is higher than that of realtime signals.

Sending a signal

Send a signal within the same process

```
#include <signal.h>

int raise(int sig);
```

Send a signal to another process

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

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

Send a signal to a process group

```
#include <signal.h>

int killpg(int pgrp, int sig);
```

Send a signal to a process from a shell

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

Sending a realtime signal

Send a realtime signal (with payload)

```
#include <signal.h>

int sigqueue(pid_t pid, int sig, const
             union sigval value);


union sigval {
  int   sival_int;
  void *sival_ptr;
};
```

Payload

Block and unblock signals (set signal mask)

```
#include <signal.h>

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

SIG_BLOCK: Adds the specified signals to the set of blocked signals

SIG_UNBLOCK: Removes the specified signals from the mask

SIG_SETMASK: Sets the mask as in the specified set

Signal set (to define the mask)

Used to retrieve the current mask, NULL if it is not needed

Manipulating a signal set

Initialize the signal set (empty set, no signal defined)

```
#include <signal.h>

int sigemptyset(sigset_t *set);
```

Fill the set (add all signals)

```
int sigfillset(sigset_t *set);
```

Add a signal to the set

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

Remove a signal from the set

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

Check whether a signal is in the set

```
int sigismember(const sigset_t *set, int signum);
```

Example

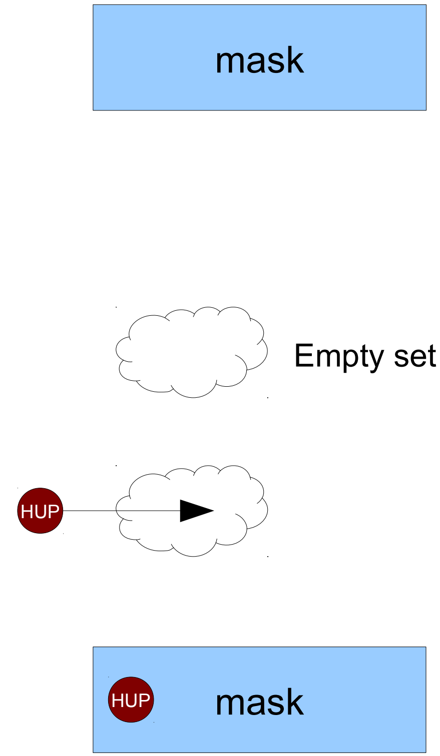
```c
#include <sys/types.h>
#include <signal.h>

void main() {
   sigset_t sigsetNew, sigsetBefore;


   sigemptyset(&sigsetNew);


   sigaddset(&sigsetNew, SIGHUP);


   sigprocmask(SIG_BLOCK, &sigsetNew, &sigsetBefore);
}
```



Handling a signal

- By default a process does not ignore an incoming signal
  - Most signals can be intercepted, and a special purpose routine (**signal handler**) can be defined to respond to the event.
  - A special signal handler enables the process to ignore signals.
  - Some signals cannot be intercepted or ignore (for example, SIGSTOP or SIGKILL), and only the *default* action is possible.

Specifying the signal handler

- With the **sigaction** function we can associate an action to be executed when the specified signal is received by the process

```
#include <signal.h>

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

Struct which contains a pointer to the handler

Pointer to the old sigaction (can be NULL)

struct sigaction

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

Obsolete, not part of the POSIX standard

Pointer to the signal handler: it is possible to use <u>either</u> `sa_handler` <u>or</u> `sa_sigaction` (which gives more information). If set to **SIG_IGN** the signal is ignored, if set to **SIG_DFL** the default action is used.

Signal handler behavior (flags)

Set of signals which will be blocked while the handler is running. The signal itself is automatically blocked (unless **SA_NODEFER** is used)

Flags

- **SA_NOCLDSTOP**
  - The SIGCHLD is not generated when a child process is stopped (i.e. when it receives SIGSTOP, SIGTSTP, SIGTTIN or SIGTTOU) o when it restarts execution (SIGCONT). Useful only when declaring a signal handler for SIGCHLD.
- **SA_NOCLDWAIT**
  - If the signal is SIGCHLD does not zombify child processes when they terminate (i.e. it is not necessary that the parent process uses **`wait`** or **`waitpid`**).
- **SA_NODEFER**
  - Does not block the signal that is currently being handled (allows for recursive calls to the handler).
- **SA_RESTART**
  - Allows for some system calls to restart if they get interrupted by the signal.
- **SA_SIGINFO**
  - If this flag is set use **`sa_sigaction`** instead of **`sa_handler`**.

What does sa_sigaction receive: siginfo_t (detail)

```
siginfo_t {
  int      si_signo;    /* Signal number */
  int      si_errno;    /* An errno value */
  int      si_code;     /* Signal code */
  int      si_trapno;   /* Trap number that caused hardware-generated signal (unused on most
                 architectures) */
  pid_t    si_pid;      /* Sending process ID */
  uid_t    si_uid;      /* Real user ID of sending process */
  int      si_status;   /* Exit value or signal */
  clock_t  si_utime;    /* User time consumed */
  clock_t  si_stime;    /* System time consumed */
  sigval_t si_value;    /* Signal value */
  int      si_int;      /* POSIX.1b signal */
  void    *si_ptr;      /* POSIX.1b signal */
  int      si_overrun;  /* Timer overrun count; POSIX.1b timers */
  int      si_timerid;  /* Timer ID; POSIX.1b timers */
  void    *si_addr;     /* Memory location which caused fault */
  long     si_band;     /* Band event (was int in glibc 2.3.2 and earlier) */
  int      si_fd;       /* File descriptor */
  short    si_addr_lsb; /* Least significant bit of address (since kernel 2.6.32) */
}
```

Example (sa_handler)

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

void myhandler(int sig) {
        write(1, "Signal received!\n", 19);
}

int main(void) {
        struct sigaction saction;
        saction.sa_handler = myhandler;
        sigemptyset(&saction.sa_mask);
        sigaction(SIGINT, &saction, NULL);
        pause();
}
```

CTRL+C

Example (sa_sigaction)

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

void myhandler(int sig, siginfo_t *si, void* p) {
        write(1, "Signal received!\n", 19);
}

int main(void) {
        struct sigaction saction;
        saction.sa_sigaction = myhandler;
        saction.sa_flags = SA_SIGINFO;
        sigemptyset(&saction.sa_mask);
        sigaction(SIGINT, &saction, NULL);
        pause();
}
```

CTRL+C

Ignoring a signal

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

int main(void) {
        struct sigaction saction;
        saction.sa_handler = SIG_IGN;
        sigemptyset(&saction.sa_mask);
        sigaction(SIGHUP, &saction, NULL);
}
```

Simplified nohup

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

void main(void) {
    struct sigaction saction;
    saction.sa_handler = SIG_IGN;
    sigemptyset(&saction.sa_mask);
    sigaction(SIGHUP, &saction, NULL);
    execl("/usr/bin/gedit", "gedit", 0, NULL);
}
```

Signal mask is inherited by the child process

Notes about asynchronous signal handlers

- Because signal handlers execute asynchronously they can arbitrarily interrupt the execution flow, making some library function calls unsafe:
  - If the interrupted thread holds some kind of lock (→ IPC) which is requested by the signal handler problems may occur (→ Deadlocks)

- POSIX has the concept of **async safe functions**, which can be safely called from signal handlers
  - See **`man 7 signal`**

Synchronous signal handling: waiting for a signal

```
#include <signal.h>

int sigwait(const sigset_t *set, int *sig);
```

- Waits until a signal in the specified set is pending, removes it from the pending list, and returns it through the **`sig`** pointer
  - Signals to be retrieved using **`sigwait`** <u>must be blocked</u>
  - The signal handler for the retrieved signal is not called
  - For realtime signals other instances of the same signal might remain queued

sigwait example

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

int main(void) {
    int signal;
    sigset_t signalset;

    sigemptyset(&signalset);
    sigaddset(&signalset,SIGINT);
    sigprocmask(SIG_BLOCK, &signalset, NULL);
    sigwait(&signalset, &signal);
    printf("Received signal %d\n", signal);
    sigprocmask(SIG_UNBLOCK, &signalset, NULL);
}
```

Signals must be blocked before calling sigwait

Synchronous signal handling: waiting for a signal

```
#include <signal.h>

int sigwaitinfo(const sigset_t *set, siginfo_t *info);
```

- Like **`sigwait`**, but returns additional information through the **`siginfo_t`** structure.

Synchronous signal handling: waiting for a signal

```
#include <signal.h>

int sigtimedwait(const sigset_t *set, siginfo_t *info,
    const struct timespec *timeout);
```

- Like sigwaitinfo, but it is possible to specify a timeout

```
struct timespec {
  long tv_sec; /* seconds */
  long tv_nsec; /* nanoseconds */
};
```

sigtimedwait example

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

int main(void) {
    sigset_t signalset;
    struct timespec timeout;
    siginfo_t info;
    timeout.tv_sec = 5;
    timeout.tv_nsec = 0;
    sigemptyset(&signalset);
    sigaddset(&signalset,SIGINT);
    sigprocmask(SIG_BLOCK, &signalset, NULL);
    if (sigtimedwait(&signalset, &info, &timeout) < 0) {
        if (errno == EAGAIN)
            printf("Timeout!\n");
        else
            perror("Error!\n");
        exit(-1);
    }
    printf("Received signal %d\n", info.si_signo);
}
```

Synchronous signal handling: blocking and waiting for a signal

```
#include <signal.h>

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

- Temporarily replaces the signal mask with the given one (blocks signals in the mask), then suspends the process until delivery of a (unblocked) signal whose action is to invoke a signal handler or to terminate a process.
  - After the signal handler is finished, sigsuspend() ends (execution resumes) and the signal mask is restored.

sigsuspend example

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

void myhandler(int sig) {
    write(1, "Signal received!\n", 19);
}

int main(void) {
    sigset_t signalset;

    struct sigaction saction;
    saction.sa_handler = myhandler;
    sigemptyset(&saction.sa_mask);
    sigaction(SIGUSR1, &saction, NULL);

    sigemptyset(&signalset);
    sigaddset(&signalset, SIGINT);
    sigprocmask(SIG_BLOCK, &signalset, NULL); // blocks SIGINT
    sigsuspend(&signalset); // Resumes when a signal other than SIGINT is received
    printf("Received signal\n");
}
```

Synchronous signal handling: check pending signals

```
#include <signal.h>

int sigpending(sigset_t *set);
```

- Returns (using the **`sig`** pointer) the set of (blocked) signals that are currently pending

sigpending example

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

void main() {
    sigset_t sig;

    sigset_t signalset;
    sigemptyset(&signalset);
    sigaddset(&signalset, SIGHUP);
    sigprocmask(SIG_BLOCK, &signalset, NULL);
       raise(SIGHUP); // send signal to ourselves
    if (sigpending( &sig ) != 0 ) {
        perror("Error\n");
    } else if (sigismember(&sig, SIGHUP )) {
        printf("SIGHUP is pending\n");
    } else {
        printf("SIGHUP is not pending\n");
    }
}
```

Synchronous signal handling: wait for any delivered signal

```
#include <unistd.h>

int pause(void);
```

- Waits until a signal is delivered
  - Only non-blocked signals will resume execution

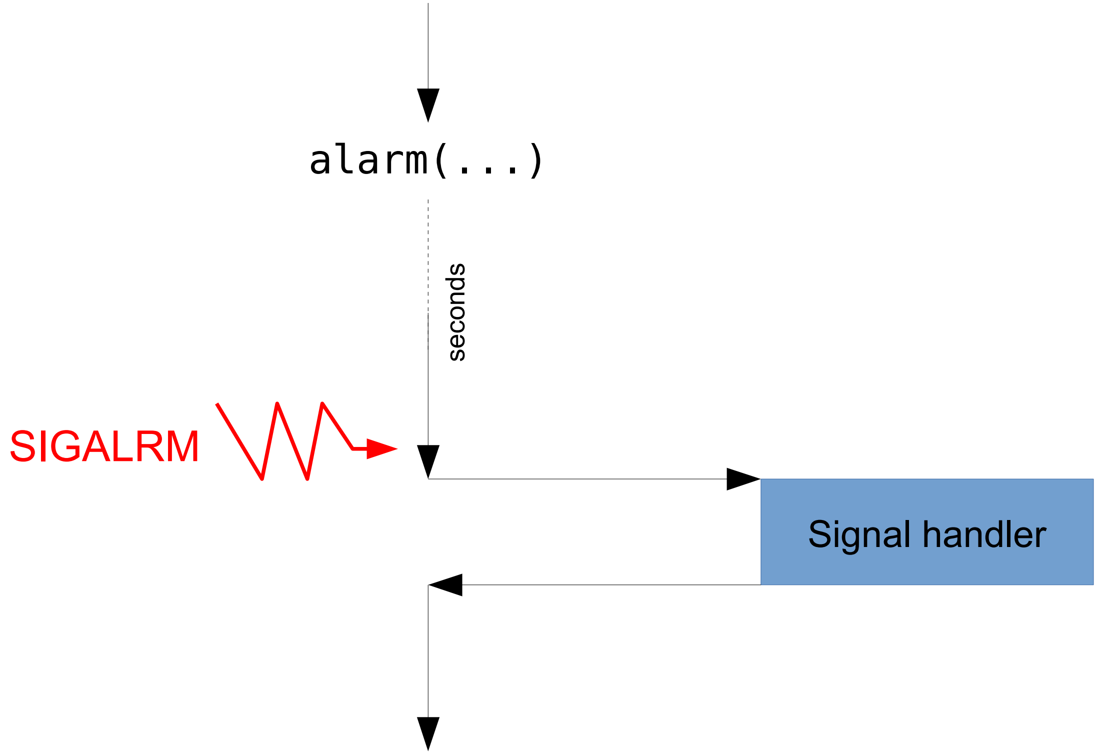
pause example

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
void myhandler(int sig) {
  write(1, "Signal received!\n", 19);
}
int main(void) {
  int segnale;
  sigset_t sigset;
  struct sigaction saction;

  saction.sa_handler = myhandler;
  sigemptyset(&saction.sa_mask);
  sigaction(SIGUSR1, &saction, NULL);

  sigemptyset(&sigset);
  sigaddset(&sigset,SIGINT);
  sigprocmask(SIG_BLOCK, &sigset, NULL);
  pause(); /* SIGINT does not resume  */
  printf("Exiting!\n");
}
```

Using signals, setting an alarm




```
#include <unistd.h>

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

alarm example

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

void myhandler(int sig) {
    write(1, "Alarm received!\n", 19);
}

int main(void) {
    sigset_t signalset;

    struct sigaction saction;
    saction.sa_handler = myhandler;
    sigemptyset(&saction.sa_mask);
    sigaction(SIGALRM, &saction, NULL);

    alarm(5);

    sigsuspend(&signalset);
}
```

A more precise timer

```
#include <sys/time.h>

int getitimer(int which, struct itimerval *curr_value);
int setitimer(int which, const struct itimerval *new_value,
              struct itimerval *old_value);
```

- Initializes an interval timer; three types are available depending on **`which`**:
  - **`ITIMER_REAL`** (to measure realtime intervals, delivers **`SIGALRM`**)
  - **`ITIMER_VIRTUAL`** (to measure only when the process is executing, delivers **`SIGVTALRM`**)
  - **`ITIMER_PROF`** (to measure when executing both in userspace and in the kernel, delivers **`SIGPROF`**)
- Timer intervals are specified using a **`itimerval`** structure
  - Details **`man setitimer`**

setitimer example

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
#include <sys/time.h>

void myhandler(int sig) {
    write(1, "Alarm received!\n", 19);
}

int main(void) {
    sigset_t signalset;
    struct itimerval when;
    struct sigaction saction;
    saction.sa_handler = myhandler;
    sigemptyset(&saction.sa_mask);
    sigaction(SIGALRM, &saction, NULL);
    when.it_interval.tv_sec = 5;
    when.it_interval.tv_usec = 0;
    when.it_value.tv_sec = 5; // First time
    when.it_value.tv_usec = 0; // First time
    setitimer(ITIMER_REAL, &when, NULL);
    while(1) {
        sigsuspend(&signalset);
    }
}
```

Side note: signals and threads

- Threads have their own signal mask (set using **`pthread_sigmask`**)
  - The signal mask of the process is inherited when a thread is created
  - Threads can query their own queue using **`sigwait`**, **`sigpending`**,...
- ... but signal handlers are per-process

- **Process-directed signals** (sent to a PID using **`kill`**):
  - Signal is not delivered to a thread that has this signal blocked.
  - If all threads have the signal blocked, it's queued in the per-process queue.
  - If more than one thread has the signal unblocked, one of the threads will get it.

- **Thread-directed signals** (sent using **`pthread_kill`**):
  - Signal is delivered or queued for the specific thread.

Other uses of POSIX signals... asynchronous I/O

- POSIX define an API for asynchronous input/output without explicitly using threads
  - the input/output request returns without waiting for the actual data transfer to be completed
  - the completion of the data transfer is notified using a signal

- To compile the following examples append **`-lrt`**

```
gcc -o program program.c -lrt
```

  - For more information: **`man aio`**

AIO functions

Asynchronous read

```
#include <aio.h>

int aio_read(struct aiocb *aiocbp);
```

Asynchronous write

```
int aio_write(struct aiocb *aiocbp);
```

Canceling an asynchronous operation

```
int aio_cancel(int fd, struct aiocb *aiocbp);
```

Pointer to the asynchronous operation to be cancelled. If NULL, all operations for the given fd will be cancelled.

Error status of an asynchronous I/O operation

```
int aio_error(const struct aiocb *aiocbp);
```

Return status of an asynchronous I/O operation

```
ssize_t aio_return(struct aiocb *aiocbp);
```

aiocb structure

```
struct aiocb {
/* The order of these fields is implementation-dependent */

   int              aio_fildes;     /* File descriptor */
   off_t            aio_offset;     /* File offset */
   volatile void   *aio_buf;        /* Location of buffer */
   size_t           aio_nbytes;     /* Length of transfer */
   int              aio_reqprio;    /* Request priority */
   struct sigevent aio_sigevent;   /* Notification method */
   int              aio_lio_opcode; /* Operation to be performed;
};
```

AIO with polling

```c
#include <aio.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#define NBYTES 10000

struct aiocb callback;

void main(void)
{
  char* buffer[NBYTES];
  FILE* file = fopen("/var/log/syslog", "r");
  callback.aio_buf = buffer;
  callback.aio_fildes = fileno(file);
  callback.aio_nbytes = NBYTES;
  callback.aio_offset = 0;
  aio_read(&callback);
  while(aio_error(&callback) == EINPROGRESS) {
    sleep(1);
  }
  printf("Bytes read %d.\n",
             (int) aio_return(&callback));
  close(fileno(file));
}
```

AIO with callback signal

```c
#include <aio.h>
#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>
#define NBYTES 10000
struct sigaction action;
struct aiocb callback;
void done_reading(int signal, siginfo_t *info, void *uap) {
  printf("Read completed, operation id %d, bytes %d. CTRL+C to exit.\n", (int) info->si_value.sival_int, (int) aio_return(&callback));
  close(callback.aio_fildes);
}
void main(void) {
  char* buffer[NBYTES];
  action.sa_sigaction = done_reading;
  action.sa_flags = SA_SIGINFO;
  sigemptyset(&action.sa_mask);
  sigaction(SIGRTMIN+7, &action, NULL);
  FILE* file = fopen("/var/log/syslog", "r");
  callback.aio_buf = buffer;
  callback.aio_fildes = fileno(file);
  callback.aio_nbytes = NBYTES;
  callback.aio_offset = 0;
  callback.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
  callback.aio_sigevent.sigev_signo = SIGRTMIN+7;
  callback.aio_sigevent.sigev_value.sival_int = 13; /* operation id */
  aio_read(&callback);
  while(1) sleep(1);
}
```

Other IPC mechanisms: POSIX message queues

- POSIX provides message passing functionalities
- We create a queue using **mq_open**
  - `mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);`
    - The queue has a name (like “/myqueue”) and access permissions can be set
- We can post a message to the queue using **mq_send**
  - `int mq_send(mqd_t mqdes, const char *msg_ptr, size_t msg_len, unsigned msg_prio);`
    - We can also set a timeout (if the queue is full) using **mq_timedsend**
- We can wait for a message using **mq_receive**
  - `ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned *msg_prio);`
    - We can set a timeout (if the queue is empty) using **mq_timedreceive**

- To close and remove a queue we use **mq_close** and **mq_unlink** respectively

POSIX message queues: example

```c
#include <stdio.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <string.h>

// Put message in the queue
int main(int argc, char *argv[]) {
    mqd_t queue_id;
    char* msg = "Hello world";
    struct mq_attr queue_attr;
    queue_id = mq_open("/myqueue",
        O_RDWR | O_CREAT | O_EXCL, S_IRWXU | S_IRWXG, NULL);
    printf("Queue id: %d\n", queue_id);
    mq_getattr(queue_id, &queue_attr);
    printf("Queue capacity: %ld messages; Message size limit:  \
        %ld bytes; Waiting messages: %ld\n",
        queue_attr.mq_maxmsg,
        queue_attr.mq_msgsize,
        queue_attr.mq_curmsgs);
    mq_send(queue_id, msg, strlen(msg) + 1, 0);
    printf("Press enter to destroy queue\n");
    getchar();
    mq_close(queue_id);
    mq_unlink("/myqueue");
}
```

```c
#include <stdio.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <string.h>

// Get message from the queue
int main(int argc, char *argv[]) {
    unsigned int sender;
    int msg_size;
    mqd_t queue_id;
    char msg[10000]; // Receive buffer
    struct mq_attr queue_attr;
    queue_id = mq_open("/myqueue", O_RDWR);
    printf("Queue id: %d\n", queue_id);
    mq_getattr(queue_id, &queue_attr);
    printf("Queue capacity: %ld messages; Message size limit:  \
        %ld bytes; Waiting messages: %ld\n",
        queue_attr.mq_maxmsg,
        queue_attr.mq_msgsize,
        queue_attr.mq_curmsgs);
    msg_size = mq_receive(queue_id, msg, 10000, &sender);
    printf("Got: %d bytes ('%s') from %d\n", msg_size, msg, sender);
    mq_close(queue_id);
}
```

Other IPC mechanisms: Pipes (unnamed pipes)

- A pipe is a communication channel
  - a process can **write** to the pipe
  - another process can **read** from the pipe

- The **`pipe`** function creates a pipe and returns two file descriptors in an array:
  - one for the write end
  - one for the read end

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

int main(void)
{
    int pipe_descriptor[2]; // [0]: read end, [1] write end
    char buffer[12];

    assert(pipe(pipe_descriptor) == 0);
    write(pipe_descriptor[1], "hello world", 12);
    read(pipe_descriptor[0], buffer, 12);
    printf("got %s\n", buffer);

    return 0;
}
```

Other IPC mechanisms: Pipes (unnamed pipes)

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

int main(void)
{
    int pipe_descriptor[2]; // [0]: read end, [1] write end
    char buffer[12];

    assert(pipe(pipe_descriptor) == 0);
    if (fork()) {
        // Parent
        printf("Parent pid: %d\n", getpid());
        write(pipe_descriptor[1], "hello world", 12);
        wait(NULL);
    } else {
        // Child
        read(pipe_descriptor[0], buffer, 12);
        printf("Child %d, got %s\n", getpid(), buffer);
        exit(0);
    }
    return 0;
}
```

```c
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
// ls / | grep e
int main(void)
{
    int pipe_descriptor[2]; // [0]: read end, [1] write end
    char buffer[12];

    assert(pipe(pipe_descriptor) == 0);
    if (fork()) {
        close(0); // close stdin
        dup(pipe_descriptor[0]); // stdin now is on the pipe
        close(pipe_descriptor[1]);
        execl("/bin/grep", "grep", "e", NULL);
    } else {
        close(1); // close stdout
        dup(pipe_descriptor[1]); // stdout now is on the pipe
        close(pipe_descriptor[0]);
        execl("/bin/ls", "ls", "/", NULL);
    }
    return 0;
}
```

Other IPC mechanisms: FIFOs (named pipes)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>


int main(void)
{
    int fd, i;
    char buffer[7];

    mknod("deposit", S_IFIFO | 0666, 0);
    fd = open("deposit", O_WRONLY); // blocks until the
                                    // consumer opens the fifo
    for(;;) {
        for (i=0; i<10; i++) {
            // not very safe, but we know
            // what we are doing
            printf("Writing data...");
            sprintf(buffer, "hello %d", i);
            write(fd, buffer, 7);
            printf("done\n");
        }
    }

    return 0;
}
```

Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>


int main(void)
{
    int fd, i, b;
    char buffer[8];

    mknod("deposit", S_IFIFO | 0666, 0);
    fd = open("deposit", O_RDONLY); // blocks until the
                                    // producer opens the fifo

    buffer[7] = '\0';
    do {
        printf("Reading...");
        b = read(fd, buffer, 7);
        printf("got: %s\n", buffer);
    } while (b > 0);

    return 0;
}
```

Consumer