Operating Systems Lab (C+Unix)

Enrico Bini

University of Turin

Outline

Signals

- **Signals** are software interrupts delivered to processes.
- Signals can be generated by user, software, or hardware events
- Example of signals are:
  - `SIGFPE` "Floating Point Exceptions" such as division by zero
  - `SIGILL` trying to execute an "Illegal instruction"
  - `SIGINT` used to cause program interrupt (`Ctrl+C`)
  - `SIGKILL` causes immediate program termination
  - `SIGTERM` polite version of terminating a program (`SIGTERM` can be handled by the user)
  - `SIGALRM` received when a timer (set by `alarm(int seconds)`, `timer_create(...)`, or other calls) has expired
  - `SIGCHLD` sent to a parent when a child terminates
  - `SIGSTOP`/`SIGCONT` stop/continue a process
  - `SIGUSR1`/`SIGUSR2` user-defined signals

es.

`man 7 signal` for a full list

Outline

Sending a signal to any process

1. From a C program

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

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

   - `signum`, the ID of the signal
     - ★ by sending the signal 0 (*null signal*) we can test the existence of a `pid`
       1. if `(errno==ESRCH)`, `pid` doesn't exist
       2. if `(errno==EPERM)`, `pid` exists but no permission to send signals
       3. if successful, `pid` exists and we can send signals
   - `pid`, the target process
     - ★ if -1 the signal is sent to all (allowed) processes

2. From command line: `kill`, signal, and PID

```
kill -INT <PID>
kill -SIGINT <PID>
kill -2 <PID>
```

   - If no signal is specified, then `SIGTERM` is the default
   - `sudo kill -9 -1` is an interesting experience...

Sending a signal to myself

1. `raise(signum)`

```
#include <signal.h>

int raise(int signum);
```

to send signal `signum` to myself now.

2. `alarm(int sec)`

```
#include <unistd.h>

unsigned int alarm(unsigned int sec);
```

asks the OS to send me `SIGALRM` after `sec` seconds

- returns the seconds remaining until any previously scheduled alarm was due to be delivered, or zero if there was no previously scheduled alarm

3. we can also set a periodic timer

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

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

`man setitimer`

Outline

Signal handler: default action

Each signal has a default handler (`man 7 signal`)

- **`Term`**, terminate the process.
- **`Ign`**, ignore the signal.
- **`Core`**, terminate the process and dump "core". The core dump file contains the image of the process memory at the time of termination and can be used by a debugger (such as `gdb`) to inspect the causes of termination
- **`Stop`**, stop the process
- **`Cont`**, continue the process if it is currently stopped.

Signal handler: user-defined action

- The user-defined signal handler is a function that is called **asynchronously** at the time of signal delivery, wherever in the code this happens
- At the time of signal delivery
  1. the state of the process is saved (registers, etc.)
  2. the function of the handler is executed and then returned
  3. the state of the process is restored
- The signal handler is very similar to an interrupt handler
- Some signals (example: `SIGKILL`) **cannot** be handled by the user. Otherwise, an immortal process may be allowed by handling `SIGKILL`
- **synchronous** signal delivery is possible
  - the process may want to know the precise moment of signal delivery and wait for it, if needed
    - ★ `sigwaitinfo(...)`, `sigtimedwait(...)`, and `sigwait(...)` suspends the execution until a signal is received

  out of the scope of this course

Signal handler: definition of sigaction

- The user may set a signal handler by the sigaction() system call

```
#define _GNU_SOURCE /* necessary from now on */
#include <signal.h>

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

1. signum, the number of the signal to be handled
2. act, new handler of the signal, if NULL handler unchanged
3. oldact, pointer to the old handler, if NULL no handler returned

- WARNING: sigaction is both a sys call and a struct

```
sigaction(signum,&new,NULL); /*set new handler*/
sigaction(signum,NULL,&old); /*get cur handler*/
sigaction(signum,&new,&old); /* do both */
```

Format of the sigaction structure

- Signal handlers are specified by a sigaction data structure

  man 2 sigaction

```
struct sigaction {
   void     (*sa_handler)(int signum);
   sigset_t sa_mask; /* illustrated later */
   int sa_flags;     /* illustrated later */
   /* plus others (for advanced users) */
};
```

- sa_handler is a pointer to a function declared as

```
void signal_handler(int signum);
```

- If standard behavior is required, all bytes of "other fields" must be set to 0

  test-signal-handle.c

User-defined signal handlers 1/2

1. user-defined signal handlers must be attached to the corresponding signal **before** the signal may be released (for example, if `SIGALRM` is going to be handled, first attach the handler to the signal, then invoke `alarm(...)`)
2. often a single function handles many signal and a `switch(signum)`/`case` selects the proper action for the signal

```
void handle_signal(int signum) {
/* signal signum triggered the handler */
  switch (signum) {
  case SIGINT:
    /* handle SIGINT */
    break;
  case SIGALRM:
    /* handle SIGALRM */
    break;
  /* other signals */ } }
```

User-defined signal handlers 2/2

1. user-defined signal handlers of parents are inherited by child processes
2. global variables are visible:
   1. both in the handler code (executed asynchronously upon the reception of a signal)
   2. and in the rest of the code (executed according to the "normal" flow of the program)
   - good: global variables offer a way to inform the "`main`" program of the occurrence of signals
   - bad: special care must be taken when invoking functions that use global variables

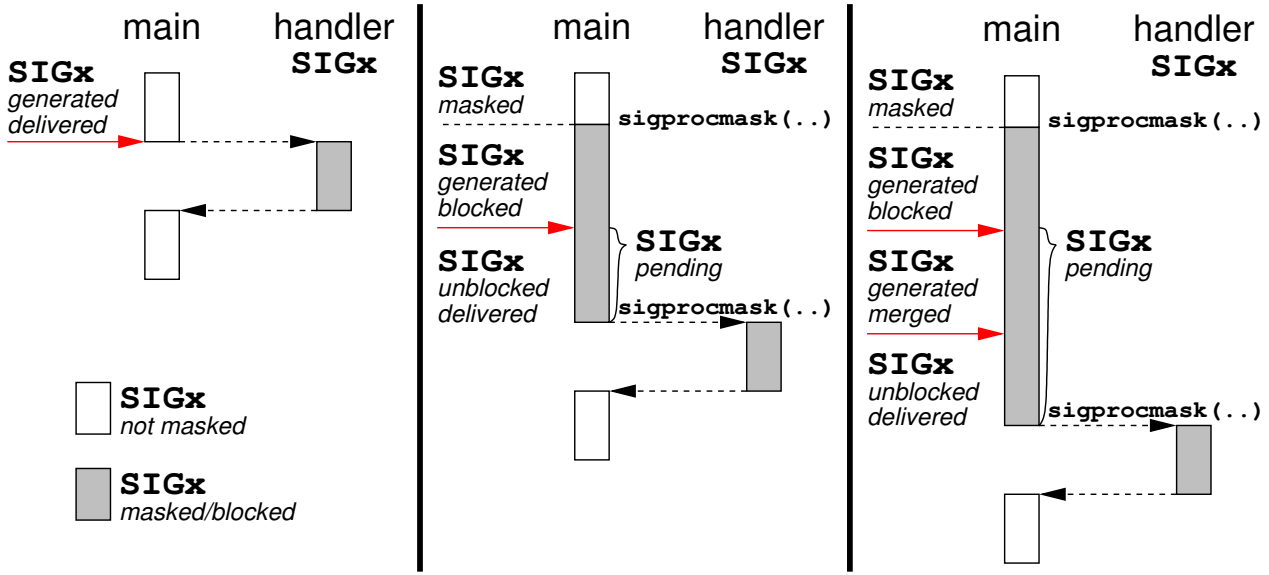
Global variables and signal handlers

- some functions (including `printf(...)`) use global data structure. The asynchronous arrival of signals may corrupt this data and produce unexpected behavior
  - `man signal-safety`
    "If a signal interrupts the execution of an unsafe function, and handler calls an unsafe function, then the behavior of the program is undefined"
- functions that can be safely used within a handler are marked by "AS-Safe" (Asynchronous Signal-Safe) in the GNU libc documentation:
  - `write(...)` is AS-Safe,
  - `printf(...)` is **AS-Unsafe**, because it uses global variables (the output buffer)
- *`test-printf-handler.c`*
- Every time you use any library function in a signal handler **check** that it is **AS-safe** at GNU libc documentation
- `errno` is a global variable, which may be overwritten within the signal (if any function writing to `errno` is used). It is recommended to save it and restore it at the end of the handler

Outline

Signals lifecycle

- Signals are *generated* by hardware or software events and are sent to a process
  1. Any process may set a *signal mask* to postpone signal handling
  2. if a signal arrives to a process when it is masked, the the signal is *blocked*
  3. if the signal is blocked, then it remains *pending*, until it is *unblocked*. As soon as unblocked, it is immediately *delivered* to the process,
  4. if a signal is generated again while it is already pending, then it is *merged*: after unblocked, it will be delivered only **once**!!
  5. if the signal is not blocked, it is immediately *delivered* to the process.



Setting signal masks

- During its execution, each process has its own *signal mask*
- a child process inherits the parent's signal mask
- The signal mask is the collection of signals that are currently *blocked*
- The signal mask of a process can be updated by the system call `sigprocmask(...)` (details later)
- Signal masks are of type `sigset_t`. Functions to manipulate sets:

```
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);
```

`man sigsetops` for more details

Signal mask of a process

- `sigprocmask(...)` is used to set the signal mask of a process

```
#include <signal.h>

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

- `oldset` is the old mask
- the new mask is set according to:
  - if (`how==SIG_BLOCK`), then signals in `set` are added to the current mask
  - if (`how==SIG_UNBLOCK`), then signals in `set` are removed from the current mask
  - if (`how==SIG_SETMASK`), then `set` becomes the new signal mask
- *test-signal-mask.c*

Signal masks: why

- Signals arrive asynchronously and may interrupt the program execution at any time
- The programmer must take special care in preventing signals to interrupt the execution in places that leave the status in an inconsistent status

  test-signal-non-atomic.c
- type `sig_atomic_t` is an integer and it is guaranteed by the compiler to be accessed atomically (by a single assembly instruction)
- In practice, we can assume that
  - `int` and
  - pointers

  are atomic
- To mask or not to mask signals?
  1. to mask to avoid inconsistent data
  2. not to mask to increase responsiveness and reduce the risk of signal merge

Signal mask during a handler

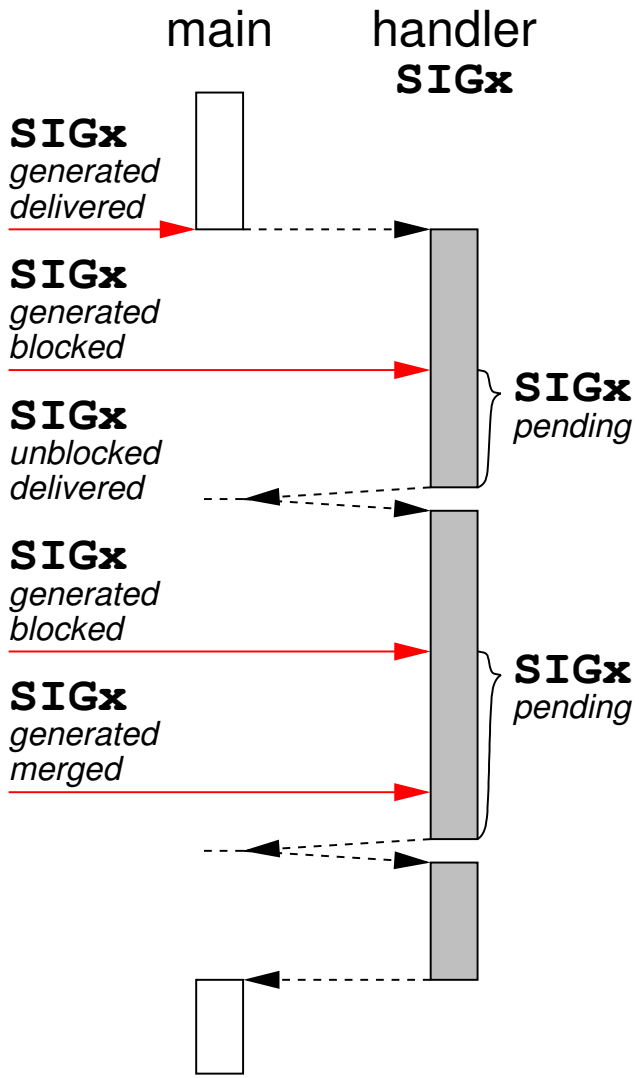
- The signal that triggered the handler is masked during the handler
  - unless the flag `SA_NODEFER` is set (to be explained later)
- `sa_mask` field of `sigaction` sets the mask during the handler
  - when the handler returns, the set of blocked signals is restored to the value before its execution, regardless of any manipulation of the blocked signals made in the handler

```
/* How to mask a signal during SIGINT handler */
struct sigaction sa;
sigset_t  my_mask;

bzero(&sa, sizeof(sa));          /* clean sa struct */
sa.sa_handler = handle_signal;    /* set handler */
sigemptyset(&my_mask);         /* Set an empty mask */
/* Add a signal to the sa_mask field struct sa  */
sigaddset(&my_mask, signal_to_mask_in_handler);
sa.sa_mask = my_mask;
/* Set the handler */
sigaction(SIGINT, &sa, NULL);
```

Merged signals in handler

- If a signal is generated while it is still pending for being handled, the newly generated and the pending one are *merged* into one
  - the presence of a pending signal is stored by a flag only, not by a number (of pending signals)
  - a signal handler cannot be reliably used to count the number of collected signals!
- *test-signal-merge.c*

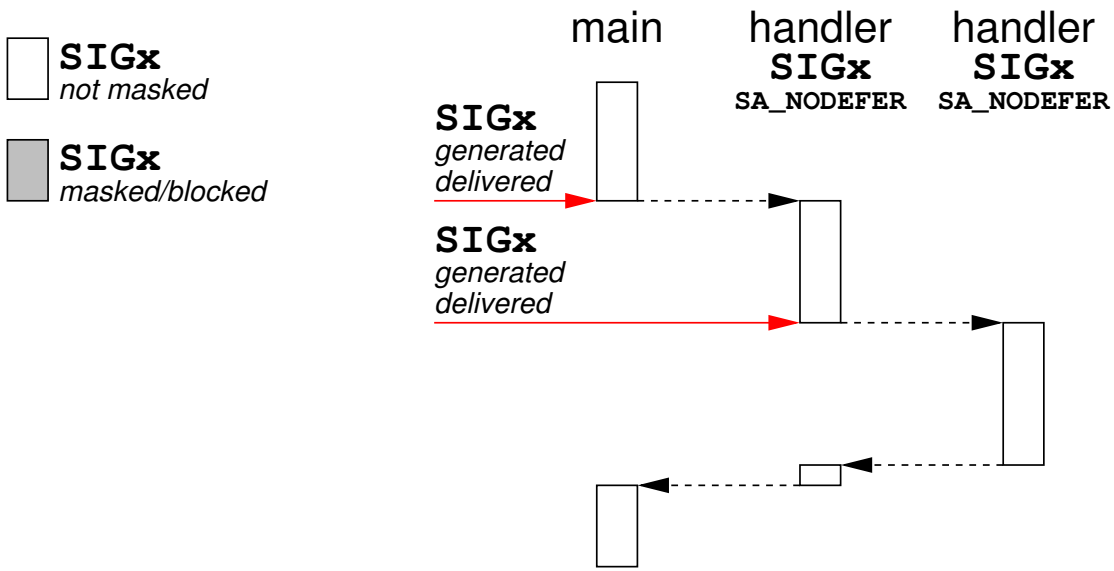


Unblocking the delivered signal during its own handler

- When a signal `signum` is delivered to a process, during its handler, the signal `signum` is automatically blocked until the handler returns
- The default behavior may be changed by setting the `SA_NODEFER` flag of the `struct sigaction`

```
bzero(&sa, sizeof(sa));
sa.sa_handler = handle_signal;
sa.sa_flags = SA_NODEFER; /* nested signals */
sigaction(SIGUSR1, &sa, NULL);
```

- *test-signal-nodefer.c*



Outline

Entering the waiting state: pausing, sleeping

1. `pause()`

```
#include <unistd.h>

int pause(void);
```

the process stays in *waiting* state until a signal is caught

2. `sleep(sec)` **(deprecated)**

```
#include <unistd.h>

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

the process sits in *waiting* state for `sec` seconds. If the process caught a signal while sleeping, then `sleep` returns the remaining second to sleep

Deprecated because from `man 3 sleep`: "`sleep()` may be implemented using `SIGALRM`; mixing calls to `alarm()` and `sleep()` is a **bad idea**."

Sleeping by nanosleep

- if sleeping for some time is needed, nanosleep is the right alternative to sleep

```
#include <time.h>

struct timespec my_time;

my_time.tv_sec = 1;
my_time.tv_nsec = 234567000;
nanosleep(&my_time, NULL);
```

sleeps for 1.234567 seconds

```
man nanosleep
```

"Synchronization" by `sleep`/`nanosleep`: **WRONG**

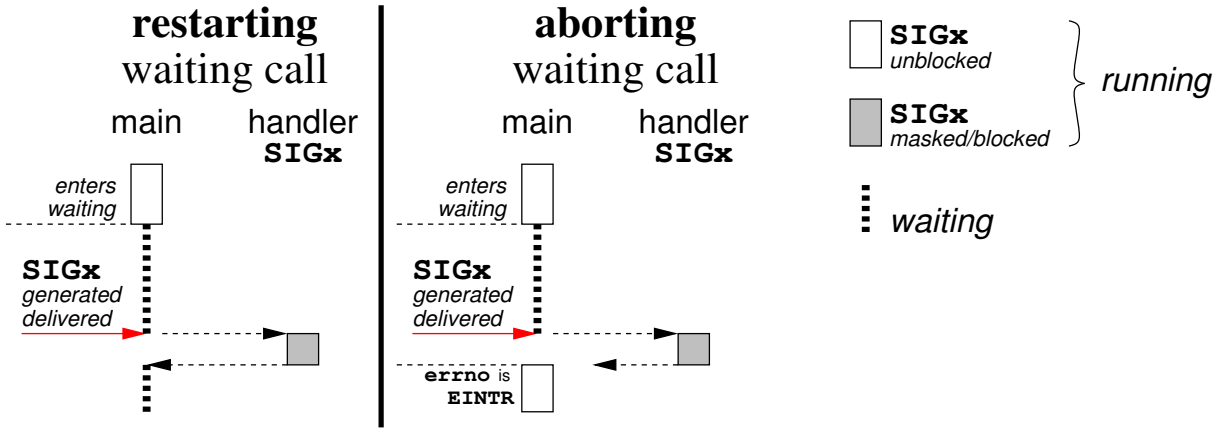
- Scenario: parent process must wait that child completes some work
- Here's a temptating (wrong) code to synchronize the two processes

```
...
if (fork()) {
    /* PARENT */
    sleep(10);
    /* the parent thinks the child has finished */
} else {
    /* CHILD */
    /* do my work before the parent checks */
}
```

- why wrong?
  1. we don't know if the child will take less than 10 seconds
  2. the OS may decide not to schedule the child for more than 10 seconds
- `sleep(sec)` only guarantees that the process sleeps for `sec`
- `sleep(...)`/`nanosleep(...)` are used mostly to show output slowly to the user
- **Never** use `sleep(...)`/`nanosleep(...)` to wait for another process

Delivery of signals to a *waiting* process

- Signal handler executes asynchronously:
  1. the state of the process is saved (registers, etc.)
  2. the function of the handler is executed
  3. the state of the process is restored
- What happens when a signal is delivered to a *waiting* process?
  - "waiting process": process waiting on `wait()`, on `pause()`, or `sleep()`, etc... Counted as "sleeping" in `top`
  1. the process is not executing, since it is waiting on some system call
     - ★ its state is already saved
  2. the function of the handler is normally executed
  3. upon the return of the handler, **two** possible behaviors:
     - ★ **restarting**: the system call is restarted when the handler returns, **OR**
     - ★ **aborting**: the system call aborts, `errno` is set to `EINTR`



Signals when waiting: selecting the desired behavior

- Which behaviour among "**restarting**" or "**aborting**"?
- The default is "**aborting**"
  1. the waiting system call returns an erroneous value: `-1`, etc. depending on the call
  2. `errno` is set equal to `EINTR`
- If the "**restarting**" behavior is desired, then consider
  1. setting the flag `SA_RESTART` in the `sa_flags` field of the `struct sigaction`
  2. checking if `(errno == EINTR)` after the waiting call and possibly re-invoke the call
- Unfortunately, different calls have different behavior
- It is then recommended to check the full documentation at

  `man 7 signal`

  Section "Interruption of system calls and library functions by signal handlers"
- *test-signal-when-wait.c*