- אני מכיר את הפקודות printf, scanf, pow, sqrt, rand, strcpy וכו׳ (הפקודות שכולם מכירים)
- אני עושה כל מאמץ שבתרגילים תצטרכו לקרוא מדריכים(!)
- אני לא מתכוון להקריא לכם מדריכים בשיעור או לעבור על הגדרות של עשרות פונקציות וקריאות מערכת. זה שיטה איומה ונוראה ללמוד. לא זוכרים ככה כלום (כמו שאי אפשר ללמוד שחייה בהתכתבות) וזה בזבוז זמן נוראי של הזמן שלי. אתם יודעים לקרוא.
- אתם תדרשו לקרוא מדריכים (או לצפות במדריכים ביוטיוב) גם במטלות הבאות. גם בלימודים גבוהים. גם בעבודה. גם אני. גם היום. אני עדיין קורא מדריכים. אם אתם לא בנויים ללימוד עצמי תחליפו מקצוע.
- אם במקרה לא תצטרכו לקרוא מדריכים לצורך מטלה מסוימת, אנא דווחו לי. המטלה תרד מהקורס במהדורות הבאות ותוחלף במטלה שתחייב קריאת מדריכים.

הדרך ללמוד למבחן

- זה לתכנת.
- ולתכנת.
- ולתכנות.
- אני אתן מטלות ומשימות אקסטרה אם תרצו.
- אני לא מתכוון לעזור לכם ללמוד לקורס בצורה תיאורטית (זה פשוט שגוי)

Operating Systems
Courtesy of BGU-CSE and Dr. Itamar Cohen

Tutorial 2 - Signals

Some remarks in red

Ben-Gurion University of the Negev
Communication Systems Engineering Department

Outline

- **Motivation & basics**
  - **Motivation**
  - Signals types
- Handling signals
- Sending signals
- Concluding question

Motivation

- Inter-Processes Communication
  - We will see some much better ways later
- Software interrupt
- Notifications of important events
  - Error – eg, division by 0
  - User request to terminate the process
    - E.g., pressing ^-C
  - Stop on breakpoints (debugging)

- A signal can be sent to a process by another *process* or by the *kernel*

Outline

- **Motivation & basics**
  - Motivation
  - **Signals types**
- Handling signals
- Sending signals
- Concluding question

(A)Synchronous signals

- Programs are ***synchronous***: executed line by line
- Signals can be
  - **Synchronous** – eg, dividing by zero
  - **Asynchronous** – eg by clock, key stroke

Signals - Examples

- SIGSEGV – SEGmentation Violation
- SIGFPE – Floating point error, eg division by 0
- SIGILL – Illegal instruction
- SIGINT – Interrupt, eg by user pressing ctrl+C. By default causes the process to terminate.
- SIGABRT – Abnormal termination, eg by user pressing ctrl+Q.
- SIGTSTP – Suspension of a process, eg by user pressing ctrl+Z
- SIGCONT – Causes suspended process to resume execution
- Which are synchronous?
- More POSIX signals

Signals 1,2,3 are synchronous, since they may arrive only as a response to a command that has been executed

Outline

- Motivation & basics
- **Handling signals**
  - **When are Signals Processed?**
  - Default actions
  - Signal handlers
- Sending signals
- Concluding question

When are Signals Processed?

- Signals are processed ***after*** a process returns from an interrupt or a system call

  … and ***before*** returning / switching to the user code

Signal – Blocking and Ignoring

- Blocking
  - The signal is received but handling it is delayed
  - Useful for protecting sensitive operations, eg
    - Prevent (some) other signals from interrupting a currently-running signal handler
    - Prevent the signal handler from modifying global variables, which the process currently uses
    - A mask can block a set of signals for a process.
- Ignoring
  - The signal is received and discarded without any action being taken

Outline

- Motivation & basics
- **Handling signals**
  - When are Signals Processed?
  - **Default actions**
  - Signal handlers
- Sending signals
- Examples & questions

Default actions

- Each signal has a ***default*** action, for example:
  - SIGTERM – Termination signal: terminate process
  - SIGFPE - floating point exception: dump core and exit
    - Dump core: produce a file, which contains the process' memory at the time of termination, and may be used for debugging
  - SIGCHILD – Child stopped or terminated: ignore
- Same default action is used for that signal for all processes

5 possible default actions

- Exit
  - Forces the process to exit
- Core
  - Forces the process to exit and create a core file
  - ulimit –c unlimited
  - If you do not see core on new ubuntu (there are other ways)
  - sudo sysctl -w kernel.core_pattern=/tmp/cores/core.%e.%p.%h.%t
  - By default on /cores in OSX
- Stop
  - Stops (pauses) the process
- Ignore
  - Ignores the signal; no action taken and **won't** be taken
- Continue
  - Resume execution of a stopped process

Outline

- Motivation & basics
- **Handling signals**
  - When are Signals Processed?
  - Default actions
  - **Signal handlers**
- Sending signals
- Examples & questions

Signal Table

- Each process has a signal table
- Each signal is presented as an entry in the table

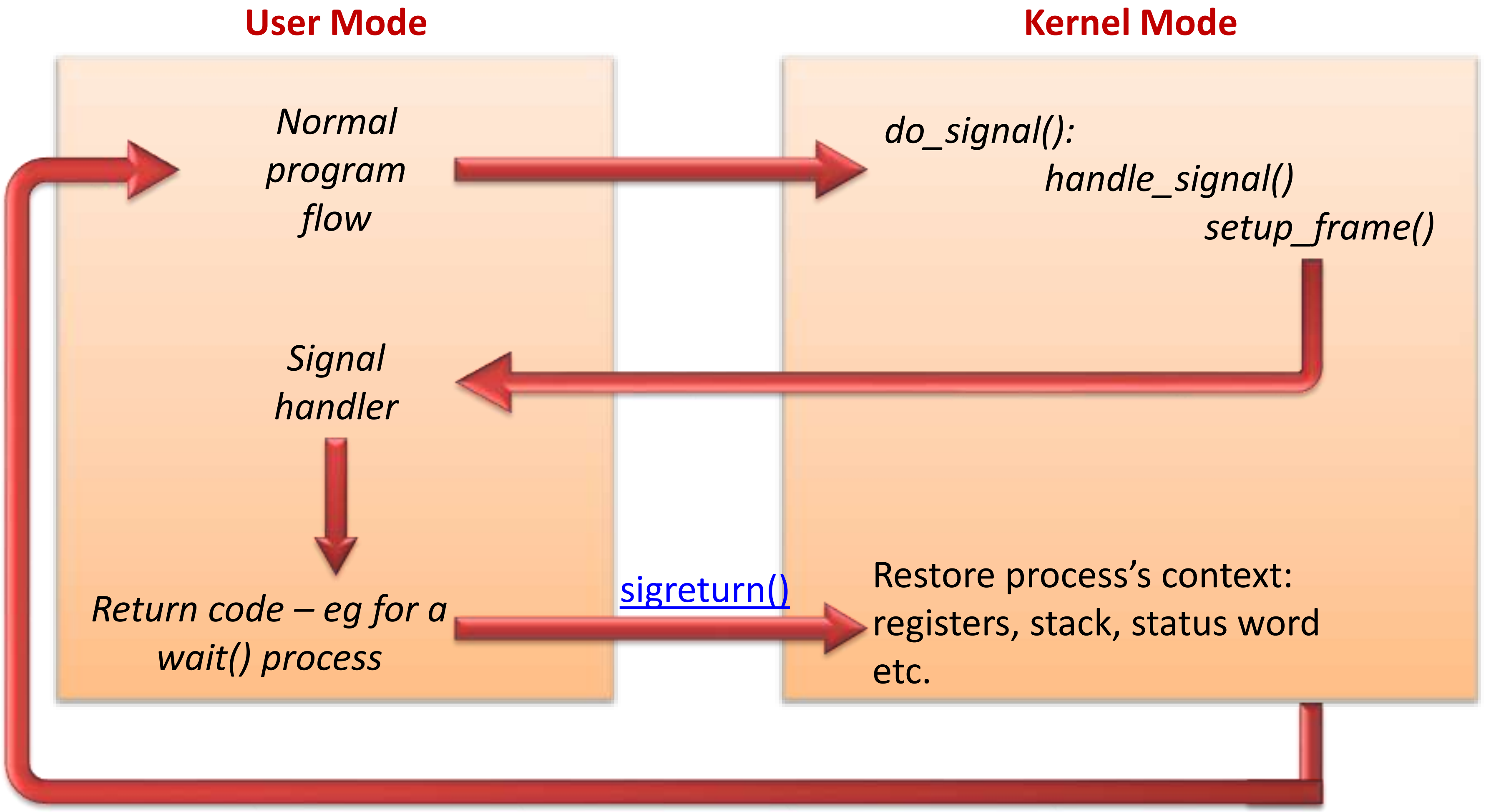
Sig_Num	SIG_IGN	Action
1		
2		

- Column SIG_IGN
  - Whether to ignore the signal or not
- Column ACTION
  - What to do on receiving the signal (if not ignoring it)
    - Pointer to a function

Signal Handlers

- A process should either
  - Ignore a signal
  - Use the default signal’s action
  - Have a *signal handler* function, which is called when the specified signal happens for that process
    - Note: the signal handler is per-process per-signal
    - In that case, we say that the process *catches* the signal

Signal Processing Scheme



Signal Handlers limitations

- Two special signals cannot be caught, blocked or ignored
  - SIGKILL – which kills the process
  - SIGSTOP – stops a process. Used for breakpoints while debugging
    - Note: this is NOT *SIGTSTP* described earlier
- When calling **`execvp()`**, any signals set to be caught by the calling process are reset to their default behaviour.
  - However, for most signals, the *ignore* bit in the process' signal table is preserved
    - Namely, the process will ignore the signal also after execvp
    - For details, see execvp() manual.

Signals, syscalls and interrupts

- Signal handlers are executed in user mode, and therefore may be preempted by another thread, just like any other user level thread.
- Signal handler may call syscalls.
- However, there exist a list of Async-signal-safe functions, namely, functions, which are NOT interrupted once called from the signal handler.

Signal Handlers (Cont')

- A process may define, that once receiving a specific signal, instead of performing the default action, its *signal handler* will be called.
- This is done using the system calls signal()and sigaction()
  - It's recommended to be consistent: always use the same function out of the two
  - signal() is simpler
    - And historically, widely used (inc. in previous years' questions)
  - However, sigaction() is more flexible and stable

signal()

sighandler_t **signal** (int signum, sighandler_t handler)

- Installs a new signal handler for the signal with number **`signum`**.
- The signal handler is set to **`sighandler`** which may be either
  - A user specified function
  - **`SIG_IGN`** (ignore the signal)
  - **`SIG_DFL`** (use the default signal's actions)
- ~~signal() is **one-shot**~~ - **incorrect**
  - ~~Should be called again after every signal caught~~
  - Sigaction allows the signal to be reset automatically
- ~~Just as bad as one-time dishes~~

Signal and one time – from man page

- The only portable use of **signal**() is to set a signal's disposition to **SIG_DFL** or **SIG_IGN**. The semantics when using **signal**() to establish a signal handler vary across systems (and POSIX.1 explicitly permits this variation); **do not use it for this purpose.**
- POSIX.1 solved the portability mess by specifying sigaction(2), which provides explicit control of the semantics when a signal handler is invoked; use that interface instead of **signal**(). In the original UNIX systems, when a handler that was established using **signal**() was invoked by the delivery of a signal, the disposition of the signal would be reset to **SIG_DFL**, and the system did not block delivery of further instances of the signal.
- This is equivalent to calling sigaction(2) with the following flags: sa.sa_flags = SA_RESETHAND | SA_NODEFER; System V also provides these semantics for **signal**(). This was bad because the signal might be delivered again before the handler had a chance to reestablish itself. Furthermore, rapid deliveries of the same signal could result in recursive invocations of the handler. BSD improved on this situation, but unfortunately also changed the semantics of the existing **signal**() interface while doing so.
- On BSD, when a signal handler is invoked, the signal disposition is not reset, and further instances of the signal are blocked from being delivered while the handler is executing. Furthermore, certain blocking system calls are automatically restarted if interrupted by a signal handler (see signal(7)). The BSD semantics are equivalent to calling sigaction(2) with the following flags: sa.sa_flags = SA_RESTART;

Signal(2) from man page

- The situation on Linux is as follows:
- The kernel's **signal**() system call provides System V semantics.
- By default, in glibc 2 and later, the **signal**() wrapper function does not invoke the kernel system call. Instead, it calls sigaction(2) using flags that supply BSD semantics.
- This default behavior is provided as long as a suitable feature test macro is defined: **_BSD_SOURCE** on glibc 2.19 and earlier or **_DEFAULT_SOURCE** in glibc 2.19 and later. (By default, these macros are defined; see feature_test_macros(7) for details.)
- If such a feature test macro is not defined, then **signal**() provides System V semantics.

What does it mean

- The POSIX standard does not define UNIX signal behaviour
- On BSD (OSX) UNIXes the default behaviour is not to reset the signal
- On Linux the behaviour is set by the library by default it is translated using sigaction(2) to BSD behaviour

sicgaction()

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

- `signum:` the signal's number
- `act:` a pointer to a *struct* containing much information including possibly a pointer to the new signal handler function
- `oldact` if not `null`, the old signal handler will be saved into it
- See documentation. Example file: **sig_actioner.c**

Sigprocmask()

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

- Changes the list of currently *blocked* signals.
- **`how`** is either any of the followings:
  - **`SIG_SETMASK`**
    - The set of blocked signals is set to the argument **`set`**.
  - **`SIG_BLOCK`**
    - The set of blocked signals is the ***union*** of the current set and **`set`**.
  - **`SIG_UNBLOCK`**
    - The signals in **`set`** are removed from the current set of blocked signals.
    - It is legal to attempt to unblock a signal which is not blocked.

Sigprocmask()

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

- **`oldset:`** if non-Null, oldset will hold the previous value of the signal mask

Sigprocmask()

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

- **`sigset_t:`** a basic data structure which stores signals using an array of bits, one for each signal type:

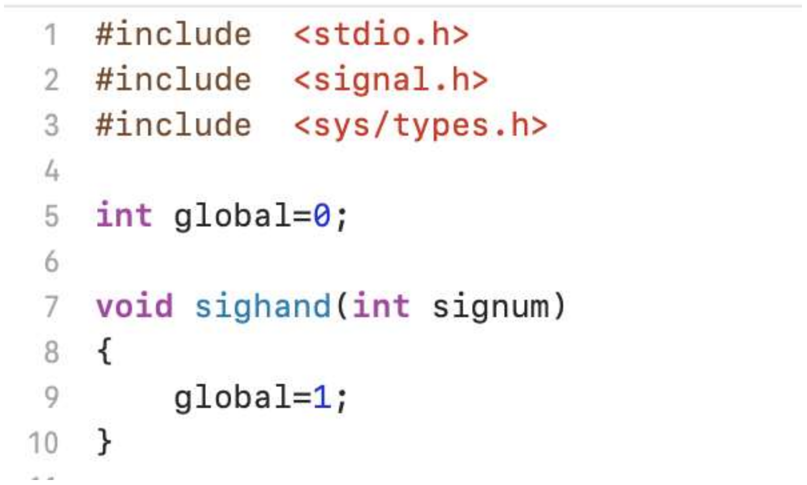
```
typedef struct {
    unsigned long sig[2];
} sigset_t;
```

- The structure should be initialized and edited using functions such as sigemptyset(), sigfillset() etc.

Signal Handlers limitations (Cont')

- A signal handler receives only a single parameter – the number of the received signal
    - As the same signal handler may catch a few different signals
- ~~A signal handler does **not** see the process' variables~~

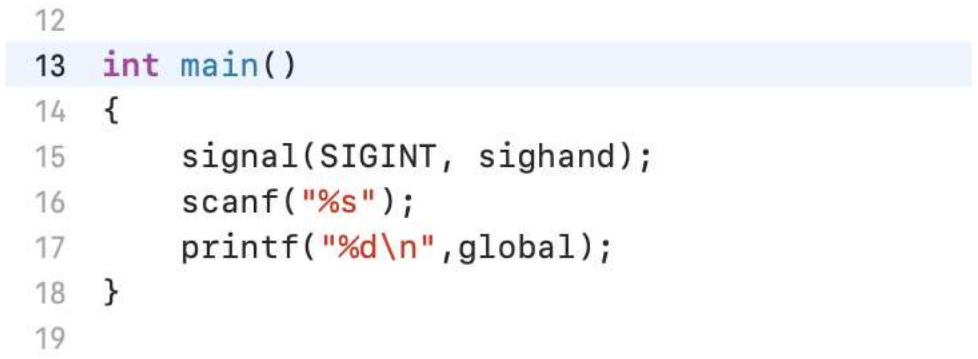
Counterexample



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

int global=0;

void sighand(int signum)
{
    global=1;
}
```



```c

int main()
{
    signal(SIGINT, sighand);
    scanf("%s");
    printf("%d\n",global);
}

```

Example 2 (see file: children_killer.c)

-
  ```
  I'm child number 3. My pid is 2621
  I'm child number 2. My pid is 2620
  I'm child number 1. My pid is 2619
  I'm child number 0. My pid is 2618
  Child number 3 caught one. My pid is 2621
  Child number 2 caught one. My pid is 2620
  Child number 1 caught one. My pid is 2619
  Child number 0 caught one. My pid is 2618
  2618 is dead
  2619 is dead
  2620 is dead
  2621 is dead
  ```
- Documentation of wait()

Is this the only possible output?

Real-Time signals

- Have no pre-defined meaning, and can be defined by the application
  - We are not discussing real time yet so this is out of scope

- POSIX defines a subset of 32 Real-Time signals which are more sophisticated
  - Multiple instances may be queued
  - Provide richer information
  - Delivered in guaranteed order
- Further details

Outline

- Motivation & basics
- Handling signals
- **Sending signals**
- Concluding question

Sending Signals

- Signals can be sent (*generated*)
  - From ***keyboard***
  - From ***command line*** via the shell
  - Using ***system calls***

Sending Signals – Keyboard

- Ctrl-C
  - Sends a ***SIGINT*** (signal-interrupt)
  - By default, this causes the process to terminate
- Ctrl-Q
  - Sends a ***SIGABRT*** signal
  - Causes an abnormal termination (abort)
- Ctrl-Z
  - Sends a ***SIGTSTP*** signal
  - By default, this causes the process to suspend execution

Sending Signal: raise() and kill()

- raise(sig) – send signal to the current running thread
- kill -<signal> <PID>
  - Sends the specified signal to the specified PID
  - e.g. `Kill -9` 1024 sends signal 9 (SIGKILL) to process 1024
  - If no signal is specified, the TERM signal is sent
- killall -<signal> <PNAME>
  - can be used to send multiple signals to processes running specific commands, owned by a specified user, have a certain age etc
    - See examples in the comments part.
- fg <PID>
  - Resumes the execution of a suspended process by sending a SIGCONT signal. This will cause the resumed process to run in the **foreg**round

Security Issues

- Not all processes can send signals to all other processes.
- Only the kernel and super-user can send signals to all processes.
- Normal processes can only send signals to processes owned by the same user.

Process Group ID

- A ***process group*** is a collection of related processes
- Each process has an ***ID*** (PID) and a ***group ID*** (PGID).
- All processes in a process group are assigned the same ***process-group identifier*** (PGID).
- A signal can be sent to a single process or to a group.
- Used by the shell to control different tasks executed by it.

Process ID

- `int getpid()`
  - Returns the process's PID
- `int getpgrp()`
  - Return the process's PGID
- `setpgrp()`
  - Sets this process's PGID to be equal to its PID
- `setpgrp(int pid1, int pid2)`
  - Sets process's pid1 PGID to be equal to pid2's PID

Outline

- Motivation & basics
- Handling signals
- Sending signals
- **Concluding question**

Question (from 2004)

- תלמיד קיבל משימה לכתוב תכנית שמטרתה להריץ תכנית נתונה (כשברשותו רק הקובץ הבינארי) בשם prompt, ע"י שימוש ב-fork ו-execvp.
- בנוסף נדרש התלמיד למנוע מן המשתמש "להרוג" את התכנית ע"י הקשת ctrl-c.
- שים לב כי התכנית prompt אינה מסתיימת לעולם.
- מצורף פתרון שהוצע ע"י התלמיד וכן התכנית prompt בקבצים – prompter.c, T2_concluding_qstn.c

Question – Cont.

- Describe the exact output, when the input is:

```
Good luck [enter] in the [^c] midterm exam.
```

- Solution:

```
Type something: Good luck
You typed: Good luck
Type something: in the ^c
My son 139 has terminated
```

Question – Cont.

- Does the suggested solution answer demands?

- Solution:
    - `execvp()` doesn't save signal handlers
    - Therefore `prompt.c` doesn't ignore `^c`
    - This means that the process can be terminated

Question – Cont.

- Fix the student’s solution, changing at most 2 code lines.

- Solution:
  - Replace

```
signal (SIGINT, cntl_c_handler);
```

With

```
signal (SIGINT, SIG_IGN);
```

```
Note, however, that this only makes the process to
ignore ^C; the output is still not as required.
```