

CSCB09 Software Tools and Systems Programming

Inter-Process Communication – Signals

Marcelo Ponce

Winter 2023

Department of Computer and Mathematical Sciences - UTSC

Today's class

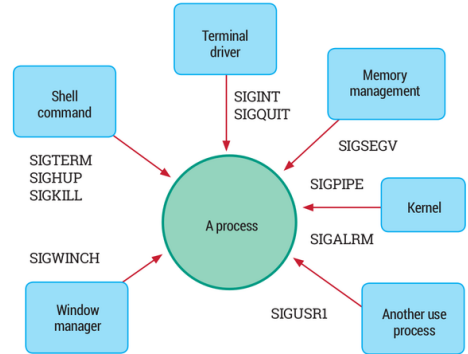
Today we will discuss the following topics:

- Signals
- Signals Handling
- Examples

Signals

Signals

- are used to notify a program of a given “event”
- standardized messages sent to a running program to trigger specific behavior, such as quitting or error handling
- a limited form of inter-process communication (IPC)
- asynchronous notification sent to a process or thread within a process to notify of an event
- Common uses of signals are to interrupt, suspend, terminate or kill a process.



- Unexpected/unpredictable asynchronous events
 - floating point error
 - segmentation fault
 - control-C (termination request)
 - control-Z (suspend request)
- Events are called interrupts
- When the OS kernel recognizes an event, it sends a signal to the process.
- Normal processes may send signals.

What are signals for?

- Synchronization between processes (e.g. parent and forked children)
- OS communicates hardware events to a process
- Signals are generated by
 - machine interrupts
 - the program itself, other programs or the user.

Signals Table

`man 7 signal` \rightsquigarrow description of signals with default actions

Signal	Default Action	Comment
SIGINT	Terminate	Interrupt from keyboard
SIGSEGV	Terminate/Dump core	Invalid memory reference
SIGKILL	Terminate (cannot ignore)	Kill
SIGCHLD	Ignore	Child stopped or terminated
SIGSTOP	Stop (cannot ignore)	Stop process
SIGCONT		Continue if stopped

...			
Signal	Standard	Action	Comment
SIGABRT	P1990	Core	Abort signal from abort(3)
SIGALRM	P1990	Term	Timer signal from alarm(2)
SIGBUS	P2001	Core	Bus error (bad memory access)
SIGCHLD	P1990	Ign	Child stopped or terminated
SIGCLD	-	Ign	A synonym for SIGCHLD
SIGCONT	P1990	Cont	Continue if stopped
SIGEMT	-	Term	Emulator trap
SIGFPE	P1990	Core	Floating-point exception
SIGHUP	P1990	Term	Hangup detected on controlling terminal or death of controlling process
SIGILL	P1990	Core	Illegal Instruction
SIGINFO	-		A synonym for SIGPWR
SIGINT	P1990	Term	Interrupt from keyboard
SIGIO	-	Term	I/O now possible (4.2BSD)
SIGIOT	-	Core	IOT trap. A synonym for SIGABRT
SIGKILL	P1990	Term	Kill signal
SIGLOST	-	Term	File lock lost (unused)
SIGPIPE	P1990	Term	Broken pipe: write to pipe with no readers; see pipe(7)
SIGPOLL	P2001	Term	Pollable event (Sys V); synonym for SIGIO
SIGPROF	P2001	Term	Profiling timer expired
SIGPWR	-	Term	Power failure (System V)
SIGQUIT	P1990	Core	Quit from keyboard
SIGSEGV	P1990	Core	Invalid memory reference
SIGSTKFLT	-	Term	Stack fault on coprocessor (unused)
SIGSTOP	P1990	Stop	Stop process
SIGTSTP	P1990	Stop	Stop typed at terminal
SIGSYS	P2001	Core	Bad system call (SVr4); see also seccomp(2)
SIGTERM	P1990	Term	Termination signal
SIGTRAP	P2001	Core	Trace/breakpoint trap
SIGTTIN	P1990	Stop	Terminal input for background process
SIGTTOU	P1990	Stop	Terminal output for background process
SIGUNUSED	-	Core	Synonymous with SIGSYS
SIGURG	P2001	Ign	Urgent condition on socket (4.2BSD)
SIGUSR1	P1990	Term	User-defined signal 1
SIGUSR2	P1990	Term	User-defined signal 2
SIGVTALRM	P2001	Term	Virtual alarm clock (4.2BSD)
SIGXCPU	P2001	Core	CPU time limit exceeded (4.2BSD); see setrlimit(2)
SIGXFSZ	P2001	Core	File size limit exceeded (4.2BSD); see setrlimit(2)
SIGWINCH	-	Ign	Window resize signal (4.3BSD, Sun)
The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored.			
...			

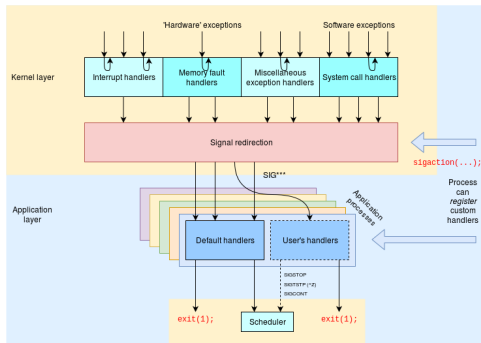
SIGINT	2	Interrupt a process (used by Ctrl-C)
SIGHUP	1	Hang up or shut down and restart process
SIGKILL	9	Kill the process (cannot be ignored or caught elsewhere)
SIGTERM	15	Terminate signal, (can be ignored or caught)
SIGTSTP	20	Stop the process (used by Ctrl-z)
SIGSTOP	19	Stop execution (cannot be caught or ignored)

- There are 31 standard signals, numbered 1-31.
Each signal is named as "SIG" followed by a suffix.
- Starting from version 2.2, the Linux kernel supports 33 different real-time signals. These have numbers 32-64 but one should instead use SIGRTMIN+n notation.
- Linux implementation of signals is fully POSIX compliant.
- Just as hardware subsystems can interrupt the processor, signals interrupt process execution. They are therefore seen as *software interrupts*.
- Some signals are mapped to specific key inputs:
SIGINT for ctrl+c, SIGSTOP for ctrl+z, SIGQUIT for ctrl+\.

- When a signal is sent, the operating system interrupts the target process' normal flow of execution to deliver the signal.
- If the process has previously registered a *signal handler*, that routine is executed. Otherwise, the default signal handler is executed.
- Signals are similar to *interrupts*, the difference being that interrupts are mediated by the CPU and handled by the kernel while signals are mediated by the kernel (possibly via system calls) and handled by individual processes.
- The kernel may pass an interrupt as a signal to the process that caused it (typical examples are SIGSEGV, SIGBUS, SIGILL and SIGFPE).

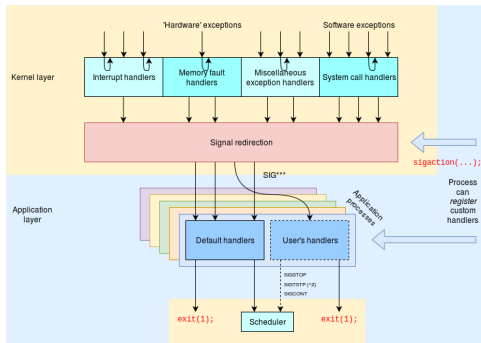
OS Structures for Signals

- For each process, the OS maintains 2 integers with the bits corresponding to a signal numbers.
- The two integers keep track of: pending signals and blocked signals
- With 32 bit integers, up to 32 different signals can be represented.



OS Structures for Signals

- For each process, the OS maintains 2 integers with the bits corresponding to a signal numbers.
- The two integers keep track of: pending signals and blocked signals
- With 32 bit integers, up to 32 different signals can be represented.



Example: SIGINT (= 2) signal is blocked and no other signals are pending

Pending Signals

31	30	29	28	...	3	2	1	0
0	0	0	0	...	0	0	0	0

Blocked Signals

31	30	29	28	...	3	2	1	0
0	0	0	0	...	0	1	0	0

- A signal is sent to a process setting the corresponding bit in the pending signals integer for the process.
- Each time the OS selects a process to be run on a processor, the pending and blocked integers are checked.
 - If no signals are pending, the process is restarted normally and continues executing at its next instruction.
 - If \exists pending signals:
 1. "blocked" \rightarrow restart normally, but with the signals still marked as pending
 2. "pending" and *not* blocked \rightarrow OS executes the process' routines to handle the signals

Sending a signal

- From the command line use
`kill [-signal] pid [pid] ...`
- If no signal is specified, kill sends the TERM signal to the process.
- signal can be specified by the number or name of the signal.
- Examples:
`kill -QUIT 8883`
`kill -STOP 78911`
`kill -9 76433 - (9 == KILL)`
- Try `kill -l` to see the supported signals

Sending a signal

- From the command line use
`kill [-signal] pid [pid] ...`
- If no signal is specified, kill sends the TERM signal to the process.
- signal can be specified by the number or name of the signal.
- Examples:
`kill -QUIT 8883`
`kill -STOP 78911`
`kill -9 76433 - (9 == KILL)`
- Try `kill -l` to see the supported signals

Programmatically

- A signal can be generated by calling `raise()` or `kill()` system calls
- `raise()` sends a signal to the current process
- `kill()` sends a signal to a specific process

Signalling between processes

- One process can send a signal to another process using the “misleadingly” named function call: `kill(int pid, int sig);`
- This call sends the signal `sig` to the process `pid`
- Signalling between processes can be used for many purposes:
 - kill errant processes
 - temporarily suspend execution of a process
 - make a process aware of the passage of time
 - synchronize the actions of processes.

Default Actions

SIGINT	2	Interrupt a process (used by Ctrl-C)
SIGHUP	1	Hang up or shut down and restart process
SIGKILL	9	Kill the process (cannot be ignored or caught elsewhere)
SIGTERM	15	Terminate signal, (can be ignored or caught)
SIGTSTP	20	Stop the process (used by Ctrl-z)
SIGSTOP	19	Stop execution (cannot be caught or ignored)

Each signal has a default actions:

- Terminate
- Stop
- Ignore

Default Actions

SIGINT	2	Interrupt a process (used by Ctrl-C)
SIGHUP	1	Hang up or shut down and restart process
SIGKILL	9	Kill the process (cannot be ignored or caught elsewhere)
SIGTERM	15	Terminate signal, (can be ignored or caught)
SIGTSTP	20	Stop the process (used by Ctrl-z)
SIGSTOP	19	Stop execution (cannot be caught or ignored)

Each signal has a default actions:

- Terminate
- Stop
- Ignore

- The default action can be changed by installing a *signal handler* using the `sigaction()` function. The exceptions are SIGKILL and SIGSTOP.

Default Actions

SIGINT	2	Interrupt a process (used by Ctrl-C)
SIGHUP	1	Hang up or shut down and restart process
SIGKILL	9	Kill the process (cannot be ignored or caught elsewhere)
SIGTERM	15	Terminate signal, (can be ignored or caught)
SIGTSTP	20	Stop the process (used by Ctrl-z)
SIGSTOP	19	Stop execution (cannot be caught or ignored)

Each signal has a default actions:

- Terminate
- Stop
- Ignore

- The default action can be changed by installing a *signal handler* using the `sigaction()` function. The exceptions are SIGKILL and SIGSTOP.
- Note: You don't explicitly call the signal handler – OS will transfer control to this function when signal occurs.

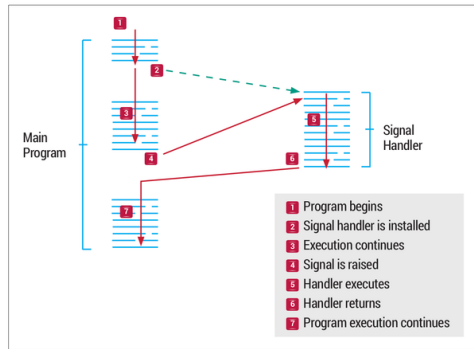
Signals Handlers

- `void handler (int signum) { ... }`
- Defines action(s) to be taken when receiving a particular signal.
- Called with the number of the signal that triggered it as an argument

Signals Handlers

- ```
void handler (int signum) { ... }
```
- Defines action(s) to be taken when receiving a particular signal.
- Called with the number of the signal that triggered it as an argument
- signal handling is vulnerable to *race conditions*.

As signals are asynchronous, another signal (even of the same type) can be delivered to the process during execution of the signal handling routine.



- Install a signal handler, act, for the signal sig

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

- Struct defined in <signal.h> to pass in for act

```
struct sigaction {
 /* SIG_DFL, SIG_IGN, or pointer to function */
 void (*sa_handler)(int);
 sigset_t sa_mask; /* Signals to block during handler */
 int sa_flags; /* flags and options */
};
```

- You may come across various extensions, including another field in the sigaction struct for a function to catch signals.

# Groups of Signals

- Signal masks are used to store the set of signals that are currently blocked.
- Operations on sets of signals:

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signo);
int sigdelset(sigset_t *set, int signo);
int sigismember(const sigset_t *set, int signo);
```

```
/* define a new mask set */
sigset_t mask_set;
/* first clear the set (i.e. make it
 contain no signal numbers) */
sigemptyset(&mask_set);
/* lets add the TSTP and INT signals to
 our mask set */
sigaddset(&mask_set, SIGTSTP);
sigaddset(&mask_set, SIGINT);
/* and just for fun, lets remove the TSTP
 signal from the set. */
sigdelset(&mask_set, SIGTSTP);
/* finally, lets check if the INT signal
 is defined in our set */
if (sigismember(&mask_set, SIGINT))
 printf("signal INT is in our set\n");
else
 printf("signal INT is not in our set -
 how strange...\n");
/* finally, lets make the set contain ALL
 signals available on our system */
sigfillset(&mask_set)
```

# Blocking Signals

- Signals can arrive at any time.
- To temporarily prevent a signal from being delivered we **block** it.
- The signal is held until the process unblocks the signal.
- When a process **ignores** a signal, it is thrown away.



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

- how indicates how the signal will be modified
  - SIG\_BLOCK: add to those currently blocked
  - SIG\_UNBLOCK: delete from those currently blocked
  - SIG\_SETMASK: set the collection of signals being blocked
- set points to the set of signals to be used for modifying the mask
- oset on return holds the set of signals that were blocked before the call.

# Timer Signals

- Three interval timers are maintained for each process:
  - SIGALRM (real-time alarm, like a stopwatch)
  - SIGVTALRM (virtual-time alarm, measuring CPU time)
  - SIGPROF (used for profilers)
- Useful functions to set and get timer info:
  - `sleep()` – cause calling process to suspend.
  - `usleep()` – like `sleep()` but at a finer granularity.
  - `alarm()` – sets SIGALRM
  - `pause()` – suspend until next signal arrives
  - `setitimer()`, `getitimer()`
- `sleep()` and `usleep()` are interruptible by other signals.

# Signals Handling

---

| Function            | Description                                                       |
|---------------------|-------------------------------------------------------------------|
| <code>raise</code>  | artificially sends a signal to the calling process                |
| <code>kill</code>   | artificially sends a signal to a specified process                |
| <code>signal</code> | sets the action taken when the program receives a specific signal |

# Examples

```
1 #include <signal.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4
5 static void catch_function(int signo) {
6 puts("Interactive␣attention␣signal␣caught.");
7 }
8
9 int main(void) {
10 // Set above function as signal handler for the SIGINT signal:
11 if (signal(SIGINT, catch_function) == SIG_ERR) {
12 fputs("An␣error␣occurred␣while␣setting␣a␣signal␣handler.\n", stderr);
13 return EXIT_FAILURE;
14 }
15 puts("Raising␣the␣interactive␣attention␣signal.");
16 if (raise(SIGINT) != 0) {
17 fputs("Error␣raising␣the␣signal.\n", stderr);
18 return EXIT_FAILURE;
19 }
20 puts("Exiting.");
21 return EXIT_SUCCESS;
22 // exiting after raising signal
23 }
```

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 /* signal handling function */
5 void quit(int code) {
6 fprintf(stderr, "\nInterrupt (code=%d)\n", code);
7 exit(1);
8 }
9
10 int main() {
11 int i = 0;
12 struct sigaction newact;
13
14 /* fill in newact */
15 newact.sa_handler = quit;
16 newact.sa_flags = 0;
17
18 if (sigaction(SIGINT, &newact, NULL) == -1) exit(1);
19
20 /* compute for a while */
21 for(;;)
22 if ((i++ % 50000000) == 0) fprintf(stderr, ".");
23 }

```

```

1 // C program to implement sighup(),
 sigint()
2 // and sigquit() signal functions
3 #include <signal.h>
4 #include <stdio.h>
5 #include <stdlib.h>
6 #include <sys/types.h>
7 #include <unistd.h>
8
9 // function declaration
10 void sighup();
11 void sigint();
12 void sigquit();
13
14 // driver code
15 void main()
16 {
17 int pid;
18
19 /* get child process */
20 if ((pid = fork()) < 0) {
21 perror("fork");
22 exit(1);
23 }

```

```

1 if (pid == 0) { /* child */
2 signal(SIGHUP, sighup);
3 signal(SIGINT, sigint);
4 signal(SIGQUIT, sigquit);
5 for (;;)
6 ; /* loop for ever */
7 }
8
9 else /* parent */
10 { /* pid hold id of child */
11 printf("\nPARENT: sending SIGHUP\n\n");
12 kill(pid, SIGHUP);
13
14 sleep(3); /* pause for 3 secs */
15 printf("\nPARENT: sending SIGINT\n\n");
16 kill(pid, SIGINT);
17
18 sleep(3); /* pause for 3 secs */
19 printf("\nPARENT: sending SIGQUIT\n\n");
20 kill(pid, SIGQUIT);
21 sleep(3);
22 }
23 }

```

```

1 // sighup() function definition
2 void sighup()
3 {
4 signal(SIGHUP, sighup); /* reset signal */
5 printf("CHILD: I have received a SIGHUP\n");
6 }
7
8 // sigint() function definition
9 void sigint()
10 {
11 signal(SIGINT, sigint); /* reset signal */
12 printf("CHILD: I have received a SIGINT\n");
13 }
14
15 // sigquit() function definition
16 void sigquit()
17 {
18 printf("My Parent has Killed me!!!\n");
19 exit(0);
20 }

```



## Further Resources

- <https://man7.org/linux/man-pages/man2/signal.2.html>
- <https://man7.org/linux/man-pages/man7/signal.7.html>