EXPERIMENT 9

Linux/Unix Signals

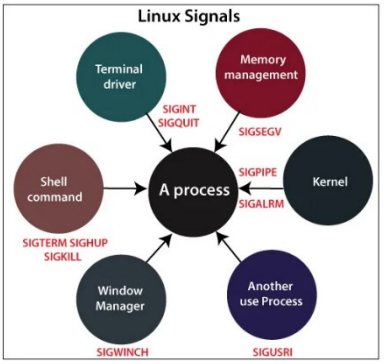
What are Signals?

A signal is a software interrupt delivered to a process. The operating system uses signals to report exceptional situations to an executing program. Some signals report errors such as references to invalid memory addresses; others report asynchronous events, such as disconnection of a phone line. They are a way for the kernel, other processes, or the system itself to notify a process about events or conditions that have occurred.

Understanding signals is essential for understanding the behavior of Unix-like operating systems and writing robust, reliable, and responsive software. Students can explore signals through programming labs by implementing signal handlers, creating signal-driven applications, and experimenting with signal handling strategies to gain practical experience with this important aspect of operating system programming.

- Each signal is identified by a unique integer number.
- Signals can be generated by the kernel in response to events such as errors, exceptions, or user actions (e.g., pressing Ctrl+C).
- Processes can also send signals to other processes using the `kill` command or system calls like `kill` and `raise`.
- Some common signals include `SIGINT` (generated by Ctrl+C), `SIGSEGV` (segmentation violation), `SIGTERM` (termination request), `SIGILL` (illegal instruction), and `SIGALRM` (alarm clock).
- Each signal has a default action associated with it, such as terminating the process, ignoring the signal, or stopping the process.
- Signal handlers should be carefully designed to be signal-safe, meaning they can be safely executed asynchronously without causing unexpected behavior or corruption.
- Signal-safe functions are those that can be safely called from within a signal handler. Examples include `write`, `read`, and `exit`.

#	Signal Name	Default Action	Comment	POSIX
1	SIGHUP	Terminate	Hang up controlling terminal or process	Yes
2	SIGINT	Terminate	Interrupt from keyboard, Control-C	Yes
3	SIGQUIT	Dump	Quit from keyboard, Control-\	Yes
4	SIGILL	Dump	Illegal instruction	Yes
5	SIGTRAP	Dump	Breakpoint for debugging	No
6	SIGABRT	Dump	Abnormal termination	Yes
6	SIGIOT	Dump	Equivalent to SIGABRT	No
7	SIGBUS	Dump	Bus error	No
8	SIGFPE	Dump	Floating-point exception	Yes
9	SIGKILL	Terminate	Forced-process termination	Yes
10	SIGUSR1	Terminate	Available to processes	Yes
11	SIGSEGV	Dump	Invalid memory reference	Yes
12	SIGUSR2	Terminate	Available to processes	Yes
13	SIGPIPE	Terminate	Write to pipe with no readers	Yes
14	SIGALRM	Terminate	Real-timer clock	Yes
15	SIGTERM	Terminate	Process termination	Yes
16	SIGSTKFLT	Terminate	Coprocessor stack error	No
17	SIGCHLD	Ignore	Child process stopped or terminated or got a signal if traced	Yes
18	SIGCONT	Continue	Resume execution, if stopped	Yes
19	SIGSTOP	Stop	Stop process execution, Ctrl-Z	Yes
20	SIGTSTP	Stop	Stop process issued from tty	Yes
21	SIGTTIN	Stop	Background process requires input	Yes
22	SIGTTOU	Stop	Background process requires output	Yes
23	SIGURG	Ignore	Urgent condition on socket	No
24	SIGXCPU	Dump	CPU time limit exceeded	No
25	SIGXFSZ	Dump	File size limit exceeded	No
26	SIGVTALRM	Terminate	Virtual timer clock	No
27	SIGPROF	Terminate	Profile timer clock	No
28	SIGWINCH	Ignore	Window resizing	No
29	SIGIO	Terminate	I/O now possible	No
29	SIGPOLL	Terminate	Equivalent to SIGIO	No
30	SIGPWR	Terminate	Power supply failure	No
31	SIGSYS	Dump	Bad system call	No
31	SIGUNUSED	Dump	Equivalent to SIGSYS	No



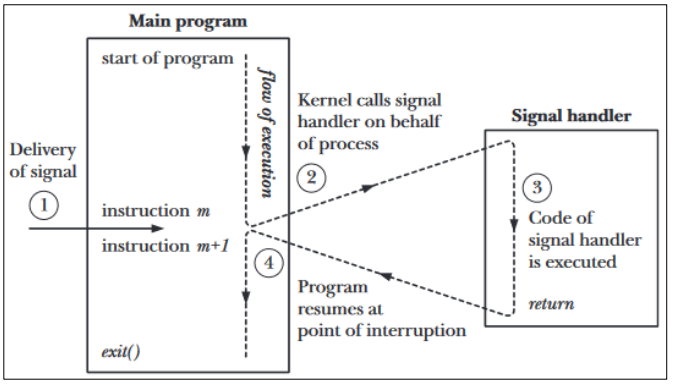


Figure 1: List of Linux signals.

Signal Handling and Delivery

- Processes can install signal handlers to specify custom actions to be taken when a signal is received. This is done using the `signal` function or more modern alternatives like `sigaction`.
- Signal handlers are functions that are executed asynchronously when a signal is delivered to the process. They can perform tasks such as cleanup, error handling, or responding to user input.
- Signals can be delivered synchronously or asynchronously. Synchronous signals are generated because of the program's actions (e.g., dividing by zero), while asynchronous signals can arrive at any time (e.g., due to user input). By default, signals are delivered to the thread that caused them. However, signal handling can be configured to deliver signals to specific threads or to the entire process.

Blocking Signals (https://www.gnu.org/software/libc/manual/html_node/Blocking-Signals.html**)**

Blocking a signal means telling the operating system to hold it and deliver it later. Generally, a program does not block signals indefinitely—it might as well ignore them by setting their actions to SIG_IGN. But it is useful to block signals briefly, to prevent them from interrupting sensitive operations. For instance:

- You can use the **sigprocmask** function to block signals while you modify global variables that are also modified by the handlers for these signals.
- You can set **sa_mask** in your **sigaction** call to block certain signals while a particular signal handler runs. This way, the signal handler can run without being interrupted by signals.
- The **pthread_sigmask()** function is just like sigprocmask(2), with the difference that its use in multithreaded programs (https://linux.die.net/man/3/pthread_sigmask).

Signals and Threads

Signals behave differently when used between threads compared to when they are used between processes in Unix-like operating systems. Understanding these differences is important for designing and implementing multi-threaded applications with robust signal handling mechanisms. Here are the key differences:

Signal Delivery
- Between Processes:
  - Signals are delivered to entire processes, not individual threads.
  - When a signal is sent to a process, it is delivered to one of the threads within the process. The specific thread that receives the signal may vary depending on the operating system and threading model.
- Between Threads:
  - Signals can be directed to specific threads within a process.
  - The `pthread_kill` function can be used to send a signal to a specific thread identified by its thread ID.

Signal Handling
- Between Processes:
  - Each process has its own signal handlers, which are independent of other processes.
  - Signal handlers are installed using the `signal` function or similar mechanisms within each process.
- Between Threads:
  - By default, threads within a process share the same signal handlers.
  - Signal handlers installed by one thread are inherited by all other threads within the same process.
  - This means that if one thread installs a signal handler for a specific signal, all threads within the process will use the same handler for that signal.

Signal Masking
- Between Processes:
  - Signal masking affects the entire process, blocking or unblocking signals for all threads within the process.
  - Signal masks are managed using system calls like `sigprocmask` or their pthreads equivalents.
- Between Threads:
  - Signal masking can be performed independently by each thread.
  - Each thread has its own signal mask, allowing it to block or unblock signals without affecting other threads in the process.
  - The `pthread_sigmask` function is used to manipulate the signal mask for individual threads.

Signal Safety
- Between Processes:
  - Signal handlers must be implemented carefully to ensure they are signal-safe, as they may be executed asynchronously in response to signals.
  - Signal-safe functions are those that can be safely called from within a signal handler without causing unexpected behavior.
- Between Threads:
  - The same considerations for signal safety apply to signal handlers within threads.
  - Functions called from signal handlers in one thread should be signal-safe to avoid potential issues.

Code workout # 1:

A C program that demonstrates the use of a signal handler for the SIGINT signal (which is sent when the user presses CTRL+C). The program will print a message when the signal is received.

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

// Signal handler function
void sigint_handler(int signum) {
    fprintf(stdout,"Caught SIGINT signal (%d)\n", signum);
}

int main() {
    // Registering signal handler for SIGINT
    if (signal(SIGINT, sigint_handler) == SIG_ERR) {
        printf("Error setting up signal handler for SIGINT\n");
        return 1;
    }

    printf("Press Ctrl+C to send a SIGINT signal\n");

    // Infinite loop to keep the program running
    while(1) {
        sleep(1);
    }

    return 0;
}
```

In this above code:

- We include the necessary header files: <stdio.h>, <signal.h>, and <unistd.h>.
- We define a signal handler function sigint_handler, which simply prints a message indicating that the SIGINT signal has been caught. printf in not safe when called in function exposed to multiple threads (technically-> printf is not reentrant), use fprintf instead.
- In the main function, we register the sigint_handler function to handle the SIGINT signal using the signal function.
- We print a message indicating that the program is running and waiting for the user to press Ctrl+C.
- The program then enters an infinite loop to keep running indefinitely.
- When the user presses Ctrl+C, the SIGINT signal is sent to the program, and the sigint_handler function is called to handle the signal, printing a message indicating that the signal has been caught.

Observations and DIY code modifications (as per questions In-lab)

a) Compile and run this code from bash command-line.

b) You can use **signal(SIGINT, SIG_IGN);** to ignore the SIGINT signal when received as well. However, we are ignoring it by assigning a signal handler to catch the signal and just displaying the message.

c) Change the sigint_handler to terminate the program after printing the message.

d) Read signal.h manual page https://manpages.ubuntu.com/manpages/trusty/man7/signal.h.7posix.html

e) You cannot terminate this program by pressing Ctrl+C. Openup another bash window and query the process and kill it as per the commands in given in the diagram below.

```
$ ps -ef | grep signal2
   38665  0 10:22 pts/4    00:00:00 ./signal2
   38760  0 10:25 pts/5    00:00:00 grep --color=auto signal2
$ kill -9 38665
$ ps -ef | grep signal2
   38760  0 10:26 pts/5    00:00:00 grep --color=auto signal2
$
```

f) Now place **raise (SIGKILL);** call as the last statement in the signal handler to kill the process when Ctrl+C is pressed. SIGKILL cannot be ignored or handled.

g) Now, use **signal (SIGUSR1, my_handler);** with appropriate my_handler function to place a user defined signal and its handler in your code. Raise the signal with **kill (pid, SIGUSR1);**

Code workout # 1(a):

Same requirements as of Code workout # 1, however using the **sigaction()** system call this is used now-a-days in all production code instead of **signal()** system call.

```c
#define _XOPEN_SOURCE 700
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>

// Signal handler function
void sigint_handler(int signum) {
    printf("Ctrl+C (SIGINT) received. Exiting...\n");
    // Perform cleanup or other necessary tasks before exiting
    exit(signum);
}

int main() {
    // Set up the signal action structure
    struct sigaction sa;
    sa.sa_handler = sigint_handler; // Specify the signal handler function
    sigemptyset(&sa.sa_mask);       // Clear the signal mask
    sa.sa_flags = 0;                // No special flags

    // Register the signal handler for SIGINT using sigaction
    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("sigaction");
        return EXIT_FAILURE;
    }

    // Loop indefinitely to keep the program running
    while(1) {
        // Do some work or wait for further input
    }

    return EXIT_SUCCESS;
}
```

In this code:

- We define a signal handler function sigint_handler to handle the SIGINT signal (Ctrl+C).
- We set up a struct sigaction variable sa with the signal handler function specified.
- We register the signal handler using the sigaction system call for the SIGINT signal.
- Inside the main function, we have a loop to keep the program running indefinitely.
- When the user presses Ctrl+C, the sigint_handler function will be called, printing a message and exiting the program.

Observations

- Why do we have **#define _XOPEN_SOURCE 700** statement at line 1?
  Answer: The problem is that the signal() function can have two different forms of behaviour when installing a signal handling function:
    - System V semantics, where the signal handler is "one-shot" - that is, after the signal handling function is called, the signal's disposition is reset to SIG_DFL - and system calls that are interrupted by the signal are not restarted; **or**
    - BSD semantics, where the signal handler is not reset when the signal fires, the signal is blocked whilst the signal handler is executing, and most interrupted system calls are automatically restarted.

  On Linux with glibc, you get the BSD semantics if _BSD_SOURCE is defined, and the System V semantics if it is not. The _BSD_SOURCE macro is defined by default, but this default definition is suppressed if you define _XOPEN_SOURCE (or a few other macros too, like _POSIX_SOURCE and _SVID_SOURCE).
- Install your signal handlers with sigaction() instead of signal() , and **set the SA_RESTART flag, which will cause system calls to automatically restart in case it got aborted by a signal**.

Code workout # 2:

When a child process stops or terminates, SIGCHLD is sent to the parent process. The default response to the signal is to ignore it. The signal can be caught and the exit status from the child process can be obtained by immediately calling wait (2) and wait3 (3C). This allows zombie process entries to be removed as quickly as possible. The following example demonstrates installing a handler that catches SIGCHLD. The wait3() system call uses WNOHANG parameter to return immediately if no child process has terminated.

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

void proc_exit() {
    int wstat;
    pid_t pid;

    while (1) {
        // Get info about child process.
        // WNOHANG returns immediately if there is no child to wait
        pid = wait3(&wstat, WNOHANG, NULL);
        if (pid == 0 || pid == -1) {
            fprintf(stdout,"return value of wait3() is %d\n",pid);
            return;
        }
        fprintf(stdout,"Return code: %d\n", wstat);
    }
}

int main() {
    signal(SIGCHLD, proc_exit);
    switch (fork()) {
    case -1:
        perror("main: fork");
        exit(0);
    case 0:
        printf("I'm alive (temporarily)\n"); // chaild only executes this and exits
        int ret_code = rand();
        printf("Return code is %d\n",ret_code);
        exit(ret_code);
    default:
        pause(); // suspends main process execution untill a signal arrives
    }
   exit (0);
}
```

Observations and DIY code modifications (as per questions In-lab)

a) Run the above code and dry run it.

b) Now, modify the main () to create 3 child processes **without** the wait () system calls to wait for each process to terminate. However, insert a sleep () call in the parent so that children terminate and become zombies. Keep line # 23 above intact so that when SIGCHLD is raised (by three processes in this case) the signal handler proc_exit is called once (observe that termination of multiple children results in only one invocation of signal handler).

```c
void sigchld_handler(int signum) {
    pid_t pid;
    int status;

    // Reap terminated child processes
    while ((pid = waitpid(-1, &status, WNOHANG)) > 0) {
        printf("Child process with PID %d terminated\n", pid);
        // Handle child process termination, if necessary
    }
}
```

Code workout # 3:

The following multithreaded C code contains a user defined signal SIGUSR1 handler and two ways to call it: i) using kill() system call that sends SIGUSR1 signal to a process, and ii) pthread_kill () library call that sends SIGUSR1 signal to a specific thread.

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

#define NUM_THREADS 4
pthread_t threads[NUM_THREADS]; // Global variable to store thread IDs

void sigusr1_handler(int signum) { // signal handler
    int i = gettid(), j = getppid(), k = getpid();
    fprintf(stdout,"Thread %u received SIGUSR1 signal (parent=%u) \
    [pid=%u] {tid=%u} \n", pthread_self(), j,k,i);
}

void *thread_function(void *arg) { // thread function
    //signal(SIGUSR1, sigusr1_handler); // register signal handler
    while (1) sleep(1);  // Keep the thread alive
    return NULL;
}

int main() {
    signal(SIGUSR1, sigusr1_handler); // register signal handler
    for (int i = 0; i < NUM_THREADS; ++i) {
        if (pthread_create(&threads[i], NULL, thread_function, NULL) != 0) {
            perror("pthread_create"); exit(EXIT_FAILURE);
        }
    }
    int i = gettid(), j = getpid(), k = getppid();
    fprintf(stdout,"Parent Process ID: %u, Process ID:%u, main() thread ID: %u\n", \
    k,j,i);
    fprintf(stdout,"Thread IDs: 0=%u, 1=%u, 2=%u and 3=%lu\n", \
    threads[0],threads[1],threads[2],threads[3] );
    kill(j,SIGUSR1); // send signal to process
    pthread_kill(threads[2], SIGUSR1); // send signal to 3rd thread

    for (int i = 0; i < NUM_THREADS; ++i) {
        if (pthread_join(threads[i], NULL) != 0) {
            perror("pthread_join"); exit(EXIT_FAILURE);
        }
    }
    return 0;
}
```

Observations and DIY code modifications (as per questions In-lab)

a) Compile and run the above code.

b) The signal handler is registered at line 23 inside the main (). Note that at line # 34 kill () is used to send signal to process and at line # 35 pthread_kill() is used to send the signal to the 3rd thread.

c) Now, comment line # 23 and uncomment line # 17. This changed signal registration from main () to the thread worker function. Observe that still both kill () and pthread_kill() are working. This means that when signal send to the process using kill (), the runtime looked up a thread where the signal handler SIGUSR1 is register and process the signal. The output of the debugger while running the program gives some insights. *Note: You may or may not reproduce the debugger output like this in the lab.*

```
Breakpoint 1, main () at /home/nakafi/Downloads/C-CPP_work/signal4.c:22
22      int main() {
Loaded '/lib/x86_64-linux-gnu/libpthread.so.0'. Symbols loaded.
Loaded '/lib/x86_64-linux-gnu/libc.so.6'. Symbols loaded.
[New Thread 0x7ffff7d9a700 (LWP 54836)]
[New Thread 0x7ffff7599700 (LWP 54837)]
[New Thread 0x7ffff6d98700 (LWP 54838)]
[New Thread 0x7ffff6597700 (LWP 54839)]

Thread 1 "signal4" received signal SIGUSR1, User defined signal 1.
0x00007ffff7de13db in kill () at ../sysdeps/unix/syscall-template.S:78
Execute debugger commands using "-exec <command>", for example "-exec info registers" will list registers in use (when GDB is the debugger)
[Switching to thread 5 (Thread 0x7ffff6597700 (LWP 54839))](running)
=thread-selected,id="5"

Thread 4 "signal4" received signal SIGUSR1, User defined signal 1.
[Switching to Thread 0x7ffff6d98700 (LWP 54838)]
0x00007ffff7e7b23f in __GI___clock_nanosleep (clock_id=clock_id@entry=0, flags=flags@entry=0, req=req@entry=0x7ffff6d97ea0, rem=rem@entry=0x7ffff6d97ea0) at
../sysdeps/unix/sysv/linux/clock_nanosleep.c:78
[Switching to thread 5 (Thread 0x7ffff6597700 (LWP 54839))](running)
=thread-selected,id="5"
```

----------(X)----------