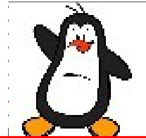
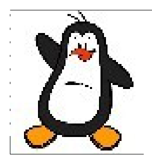


Video Lecture # 25
Design and Code of Signal Handlers

Course: SYSTEM PROGRAMMING

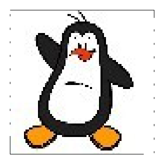
Instructor: Arif Butt

Punjab University College of Information Technology (PUCIT)
University of the Punjab



Today's Agenda

- Sending signals using **`kill()`, `raise()`, `alarm()`, `abort()`, `pause()`**
- Ignoring and writing our own Signal Handlers using `signal()`
- Introduction to Signal Sets and their related system calls
- Masking signals using `sigprocmask()`
- Limitations of `signal()` call
- Ignoring and writing Signal Handlers using `sigaction()`



Sending Signals to Processes in a C Program

kill() System call

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

- One process can send a signal to another process using the `kill()` system call
- The `pid` argument identifies one or more processes to which the signal specified by `sig` is to be sent
- If `sig` is zero then normal error checking is performed but no signal is sent. Used to determine if a specified process still exists. If it doesn't exist, a -1 is returned & `errno` is set to ESRCH
- If no process matches the specified `pid`, `kill()` fails and sets `errno` to ESRCH

`kill()` System call (cont...)

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

Four different cases determine how `pid` is interpreted:

- If **`pid` > 0**, the signal is sent to the process with the process ID specified by first argument
- If **`pid` == 0**, the signal is sent to every process in the same process group as the calling process, including the calling process itself
- If **`pid` < –1**, the signal is sent to every process in the process group whose PGID equals the absolute value of `pid`
- If **`pid` == –1**, the signal is sent to every process for which the calling process has permission to send a signal, except `init` and the calling process. If a privileged process makes this call, then all processes on the system will be signaled, except for these last two

`raise()` Library call

```
int raise(int sig);
```

- Sometimes, it is useful for a process to send a signal to itself. The `raise()` function performs this task
- In a single-threaded program, a call to `raise()` is equivalent to the following call to `kill()`:

```
kill(getpid(), sig);
```

- When a process sends itself a signal using `raise()` or `kill()`, the signal is delivered immediately (i.e., before `raise()` returns to the caller)
- Note that `raise()` returns a nonzero value (not necessarily –1) on error. The only error that can occur with `raise()` is EINVAL, because `sig` was invalid

abort() Library call

```
void abort();
```

- The abort() function terminates the calling process by raising a SIGABRT signal. The default action for SIGABRT is to produce a core dump file and terminate the process. The core dump file can then be used within a debugger to examine the state of the program at the time of the abort() call
- abort() function never returns

pause() System call

```
int pause();
```

- The `pause()` system call causes the invoking process/thread to sleep until a signal is received that either terminates it or causes it to call a signal catching function
- The `pause()` function only returns when a signal was caught and the signal-catching function returned. In this case `pause()` returns -1, and errno is set to EINTR

`alarm()` System call

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

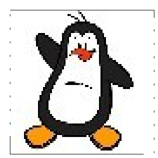
- The **`alarm()`** system call is used to ask the OS to send calling process a special signal SIGALARM(14) after a given number of seconds. If seconds is zero no new alarm is scheduled
- This function returns the previously registered alarm clock for the process that has not yet expired, i.e., the number of seconds left for that alarm clock is returned as the value of this function. Previously registered alarm clock is replaced by new value
- UNIX like systems do not operate as real-time systems, so your process might receive this signal after a longer time than requested. Moreover, there is only one alarm clock per process. Can be used for following purposes:
  - To check timeouts (e.g., wait for user input up to 30 seconds, else exits)
  - To check some conditions on a regular basis (e.g., if a server has not responded in last 30 seconds, notify the user and exits)

Adding a Delay: using `sleep()`

```
int sleep(unsigned int secs);
int usleep(useconds_t usec);
int nanosleep(const struct timespec* req,
              struct timespec* rem);
```

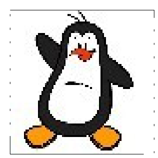
- These calls causes the calling thread to sleep (suspend execution) either until the number of specified in seconds specified in the argument have elapsed or until a signal arrives which is not ignored
- Returns zero if the requested time has elapsed, or the number of seconds left to sleep, if the call was interrupted by a signal handler

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

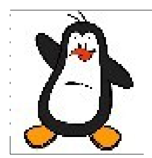


Sending Signals Proof of Concept

`sig1.c – sig5.c`



Ignoring Signals and Writing SHs using `signal()`



signal() System call

```
sighandler_t signal(int signum, void (*sh)(int));
```

- To change the disposition of a particular signal a programmer can use the `signal()` system call, which installs a new signal handler for the signal with number `signum`
- The second parameter can have three values

  i) `SIG_IGN`: the signal is ignored

  ii) `SIG_DFL`: the default action associated with signal occur

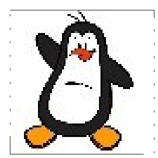
  iii) A user specified function address, which is passed an integer argument and returns nothing
- The `signal()` system call returns the previous signal handler, or SIG_ERR on error
- The signals `SIGKILL` and `SIGSTOP` cannot be caught. Moreover, the behavior of a process is undefined after it ignores `SIGFPE`, `SIGILL` or `SIGSEGV` signal that was not generated by `kill()` or `raise()` functions

Handling Signals

```
void (*oldhandler)(int);
oldhandler = signal(SIGINT, newhandler);
---
---
---
---
if (signal(SIGINT,oldhandler) == SIG_ERR){---}
---
---
---
```

If SIGINT is delivered, newhandler will be used to handle signal

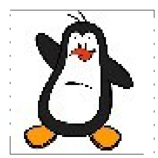
If SIGINT is delivered, oldhandler will be used to handle signal



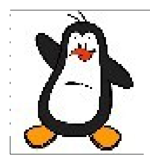
Proof of Concept

`usingsignal/ignoringsig/ignoringsig.c`

`usingsignal/handlingsig/handler1.c - handler5.c`



Masking Signals
using
`sigprocmask()`



Avoiding Race Conditions Using Signal Mask

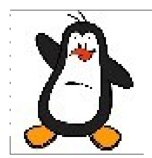
- One of the problems that might occur when handling a signal, is the occurrence of a second signal while the signal handler function is executing
- A process can temporarily prevent signals from being delivered, by blocking/masking it, while it is doing some thing critical, or while it is executing inside a signal handler
- Every process has a signal mask that defines the set of signals currently blocked for that process. One bit for each possible signal. If a bit is ON, that signal is currently blocked
- Since it is possible for the number of signals to exceed to number of bits in an integer, therefore, POSIX.1 defines a data type called `sigset_t` that holds a signal set of a process
- When a process blocks a signal, the OS doesn't deliver signal until the process unblocks the signal. However, when a process ignores a signal, signal is delivered and the process handles it by throwing it away
- Remember, after a `fork()`, child process inherits its parent mask

Functions related to Signal Sets

```
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int sig);
int sigdelset(sigset_t *set, int sig);
```

To create a process signal mask, you need to create a variable of type `sigset_t`. The `sigemptyset()` function initializes a signal set to contain no members, while the `sigfillset()` function initializes a set to contain all signals. After initialization, individual signals can be added to a set using `sigaddset()` and removed using `sigdelset()`. There are two ways to initialize a signal set:

- You can initially specify it to be empty with `sigemptyset()` and then add specified signals individually using `sigaddset()`
- You can initially specify it to be full with `sigfillset()` and then delete specified signals individually using `sigdelset()`

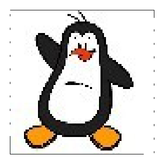


Setting the Process Signal Mask

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

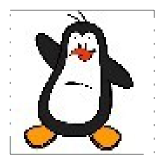
- The **`sigprocmask()`** allows us to get the existing signal mask or set a new signal mask of a process
- The second argument specifies the new signal mask. It it is NULL, then the signal mask is unchanged
- The third argument will store the old mask of the process. This is useful when we want to restore the previous masking state once we're done with our critical section
- The first argument `how` actually determines how the process signal mask will be changed. It can have following three values:

SIG_BLOCK	The set of blocked signals is the union of **nset** and the current signal set **oset**
SIG_UNBLOCK	The signals in the **nset** are removed from the current set of blocked signals. It is legal to attempt to unblock a signal which is not blocked
SIG_SETMASK	The set of blocked signals is set to the argument **nset**

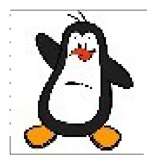


Proof of Concept

`usingsignal/maskingsig/sigprocmask.c`



Ignoring Signals, Masking Signals and Writing SHs using `sigaction()`



Limitations of `signal()` System call

- Using the `signal()` call, we cannot determine the current disposition of a signal without changing the disposition. Example: If we want to determine the current disposition of SIGINT, we can't do it without changing the current disposition

```
sighandler_t oldHandler = signal(SIGINT, &newHandler);
```

- If we use `signal()`, to register a handler for a signal, it is possible that after we entered the signal handler, but before we managed to mask all the signals using `sigprocmask()`, we receive another signal, which WILL be called

- There are a lot of variations in the behavior of `signal()` call across UNIX implementations

sigaction() System call

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

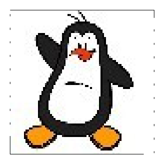
- Although `sigaction()` is somewhat more complex to use than `signal()`, it gives following advantages over `signal()`:
  - `sigaction()` allows us to retrieve the disposition of a signal without changing it, and to set various attributes controlling precisely what happens when a signal handler is invoked
  - `sigaction()` is more portable than `signal()`
- The first argument `signum` identifies the signal whose disposition we want to retrieve or change
- The second argument `newact` is a pointer to a structure specifying a new disposition for the signal. If we are interested only in finding the existing disposition of the signal, then we can specify NULL for this argument
- The third argument `oldact` is used to return information about the signal's previous disposition. If we are not interested in this information, then we can specify NULL for this argument

sigaction() System call (cont...)

- The structures pointed to by third and fourth argument to `sigaction` is of following type:

```
struct sigaction {
  void (*sa_handler)(int);
  sigset_t sa_mask;
  int sa_flags;
};
```

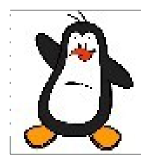
- The `sa_handler` field specifies the pointer to the handler function
- The `sa_mask` field specifies the process signal mask to be set while this signal is being handled
- The `sa_flags` field contains flags that effect signal behavior, normally it is set to zero



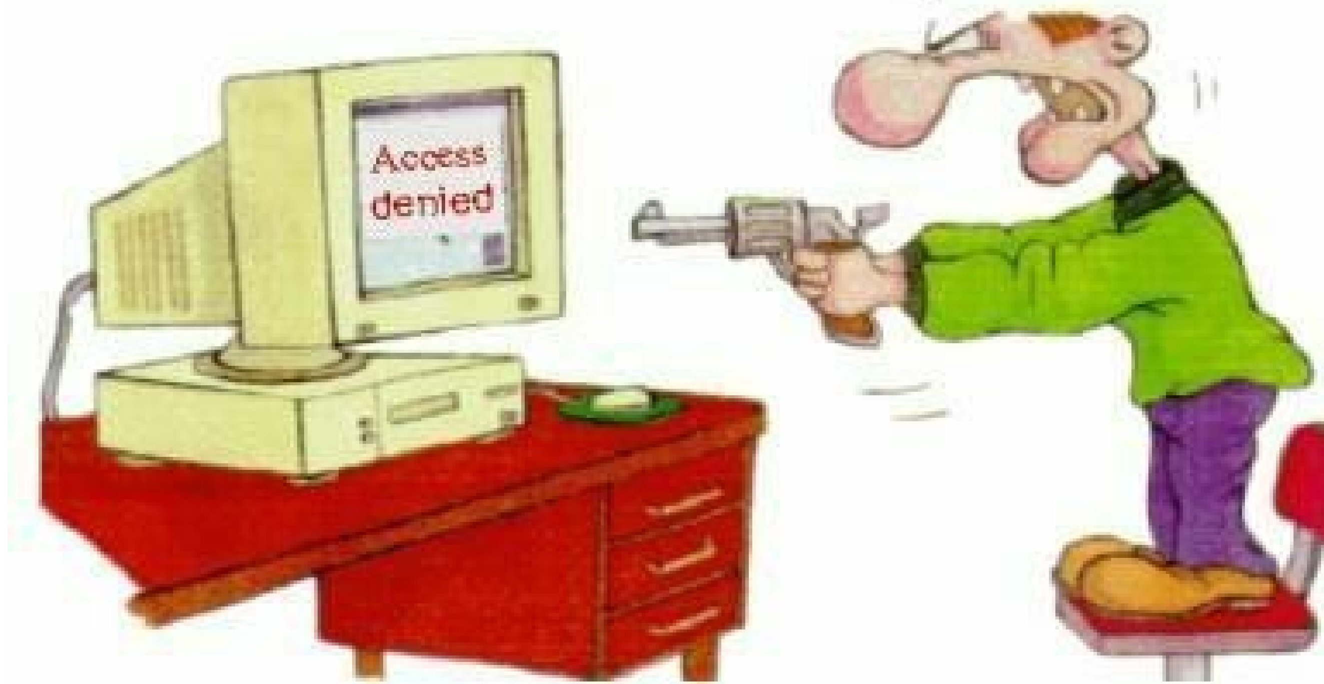
Proof of Concept

usingsigaction/ignoringsig.c

usingsigaction/handler1.c



Things To Do



If you have problems visit me in counseling hours. . . .