Signals contd..

(Signal Masks & Using sigaction)

POSIX-Defined Signals (1)

* SIGALRM: Alarm timer time-out. Generated by ***alarm( )*** API.
* SIGABRT: Abort process execution. Generated by ***abort( )*** API.
* SIGFPE: Illegal mathematical operation.
* SIGHUP: Controlling terminal hang-up.
* SIGILL: Execution of an illegal machine instruction.
* SIGINT: Process interruption. Can be generated by *<Delete> or <ctrl_C>* keys.
* SIGKILL: Sure kill a process. Can be generated by

  "***kill*** *-9 <process_id>*" command.
* SIGPIPE: Illegal write to a pipe.
* SIGQUIT: Process quit. Generated by *<crtl_\>* keys.
* SIGSEGV: Segmentation fault. generated by de-referencing a NULL pointer.

POSIX-Defined Signals (2)

* SIGTERM: process termination. Can be generated by
  *"**kill** <process_id>"* command.
* SIGUSR1: Reserved to be defined by user.
* SIGUSR2: Reserved to be defined by user.
* SIGCHLD: Sent to a parent process when its child process has terminated.
* SIGCONT: Resume execution of a stopped process.
* SIGSTOP: Stop a process execution.
* SIGTTIN: Stop a background process when it tries to read from from its controlling terminal.
* SIGTSTP: Stop a process execution by the control_Z keys.
* SIGTTOUT: Stop a background process when it tries to write to its controlling terminal.

Signaling Processes

* Signal

  A signal is a notification to a process that an event has occurred. Signals are sometimes called “software interrupts”.

* Features of Signal

  - Signal usually occur asynchronously.

  - The process does not know ahead of time exactly when a signal will occur.

  - Signal can be sent by one process to another process (or to itself) or by the kernel to a process.

Sources for Generating Signals

* Hardware
  - A process attempts to access addresses outside its own address space.
  - Divides by zero.
* Kernel
  - Notifying the process that an I/O device for which it has been waiting is available.
* Other Processes
  - A child process notifying its parent process that it has terminated.
* User
  - Pressing keyboard sequences that generate a quit, interrupt or stop signal.

Three Courses of Action

Process that receives a signal can take one of three action:

* Perform the system-specified default for the signal

  - notify the parent process that it is terminating;

  - generate a core file;

    (a file containing the current memory image of the process)

  - terminate.

* Ignore the signal

  A process can do ignoring with all signal but two special signals: SIGSTOP and SIGKILL.

* Catch the Signal

  When a process catches a signal, except SIGSTOP and SIGKILL, it invokes a special signal handing routine.

Signal Masks

- Process can temporarily prevent signal from being delivered by *blocking* it.
- *Signal Mask* contains a set of signals currently blocked.
- Signal mask is of type *sigset_t*
- **Important!** Blocking a signal is different from ignoring signal. Why?

- When a process blocks a signal, the OS does not deliver signal until the process unblocks the signal
- A *blocked* signal is not delivered to a process until it is unblocked.
- When a process ignores signal, signal is delivered and the process handles it by throwing it away.

Blocking Signals

Blocking signals

- To **block** a signal is to **queue** it for delivery at a later time
- Differs from **ignoring** a signal

Each process has a **signal mask** in the kernel

- OS uses the mask to decide which signals to deliver
- User program can modify mask with **`sigprocmask()`**

sigprocmask API - Signal Mask (1)

* Function

  A process can query or set its signal mask via the ***sigprocmask*** API.

* Include: <signal.h>

* Summary: *int* ***sigprocmask*** *(int cmd,*

  *cost sigset_t *new_mask, sigset_t *old_mask);*

* Return:

  Success: 0

  Failure: -1

  Sets errno: Yes

Arguments of ***sigprocmask*** API - Signal Mask (2)

* *new_mask*: defines a set of signals to be set or reset in a calling process signal mask.

  new_mask = NULL, current process signal mask unaltered.

* *cmd*: specifies how the new_mask value is to be used:

  - SIG_BLOCK: Adds the signals specified in the new_mask argument to the calling process signal mask.

  - SIG_UNBLOCK: Removes the signals specified in the new_mask argument from the calling process signal mask

  - SIG_SETMASK: Overrides the calling process signal mask with the value specified in the new_mask argument.

* *old_mask*: Address of a *sigset_t* variable that will be assigned the calling process's original signal mask.

  old_mask = NULL, no previous signal mask will be returned.

sigsetops APIs - Signal Mask (3)

* *int* ***sigemptyset*** *(sigset_t* sigmask);*

  Clears all signal flags in the *sigmask* argument.

* *int* ***sigaddset*** *(sigset_t* sigmask, const int signal_num);*

  Sets the flag corresponding to the *signal_num* signal in the *sigmask* argument.

* *int* ***sigdelset*** *(sigset_t* sigmask, const int signal_num);*

  Clears the flag corresponding to the *signal_num* signal in the *sigmask* argument.

* *int* ***sigfillset****(sigset_t* sigmask);*

  Sets all the signal flags in the *sigmask* argument.

* *int* ***sigismember****(const sigset_t* sigmask,const int signal_num);*

  Returns 1 if the flag corresponding to the *signal_num* signal in the *sigmask* argument is set; zero: not set; -1: the call fails.

sigprocmask Example1 -Signal Mask (4)

```
/* The example checks whether the SIGINT signal is present in a
   process signal mask and adds it to the mask if it is not there.
   It clears the SIGSEGV signal from the process signal mask. */
int main( ){
sigset_t     sigmask;
sigemptyset(&sigmask);  /*initialize set */
if(sigprocmask(0,0,&sigmask)==-1)/*get current signal mask*/
   {     perror("sigprocmask");  exit(1); }
else      sigaddset(&sigmask, SIGINT); /* set SIGINT flag*/
sigdelset(&sigmask, SIGSEGV); /* clear SIGSEGV flag */
if (sigprocmask(SIG_SETMASK,&sigmask,0) == -1)
     perror("sigprocmask"); /* set a new signal mask */
}
```

sigprocmask Example2 -Signal Mask (5)

```
/* The example blocks the SIGINT signal while doing some work, unblocks the
    signal, and does more work. This is repeated continually in a loop. */
void main( ) {
sigset_t  sigmask;
sigemptyset(&sigmask);    /*initialize set */
sigaddset(&sigmask, SIGINT); /*adds SIGINT to the set */
for( ; ; ) {
    sigprocmask(SIG_BLOCK, &sigmask, NULL);
    printf("SIGINT is blocked\n");
    /* Do some work */
   printf("Work is finished\n");
   sigprocmask(SIG_UNBLOCK, &sigmask, NULL);
   printf("SIGINT is unblocked\n");
    /* Do some more work */
   printf("All work finished\n"); }
}
```

sigpending API- Signal Mask (6)

* Function

  The sigpending API can find out whether one or more signals are pending for a process and set up special signal handing methods for those signals before the process calls the sigprocmask API to unblock them.

* Include: <signal.h>

* Summary: *int* ***sigpending*** *(sigset_t* sigmask);*

* Return: Success: 0; Failure: -1; Sets errno: Yes

* Argument

  sigmask argument, to the ***sigpending*** API, is the address of a *sigset_t*-type variable and is assigned the set of signals pending for the calling process by the API.

sigpending Example - Signal Mask (7)

```
/* The example reports to the console whether the SIGTERM
   signal is pending for the process. */
int main ( ) {
sigset_t     sigmask;
sigemptyset(&sigmask); /* initialize set */
if (sigpending(&sigmask) == -1) /* Any signal is pending */
    perror("sigpending");
else
    printf("SIGTERM signal is: ", (sigismember
    (&sigmask,SIGTERM) ? "Set" : "Not Set"));
/* whether SIGTERM signal  in the sigmask argument is set? */
```

sigaction API - sigaction (1)

* Function

  .The ***sigaction*** API setups a signal handling method for each signal it wants to deal with and passes back the previous signal handling method for a given signal. The ***sigaction*** API blocks the signal it is catching allowing a process to specify additional signals to be blocked when the API is handling a signal.

* Include: <signal.h>

* Summary: *int* ***sigaction*** *(int signal_num,*

  struct sigaction action, struct sigaction* old_action);*

* Return: Success: 0; Failure: -1; Sets errno: Yes

struct ***sigaction*** - sigaction (2)

* *struct sigaction*

{

    *void   (*sa_handler) (int);*

    sigset_t sa_mask;

    int   sa_flag;

}

* *sa_handler* is the function point of a user-defined signal handler function, SIG_IGN (ignores a signal), or SIG_DFL (accepts the default action of a signal).
* *sa_mask* specifies additional signals that a process wishes to block when it is handling the signal_num signal.
* *sa_flag* specifies special handling for certain signals (refer man page of sigaction).

Arguments of ***sigaction*** - sigaction (3)

* signal_num

  The signal_num argument designates which signal handling action is defined in the action argument.

* old_action

  The previous signal handling method for signal_num will be returned via the old_action argument if it is not a NULL pointer.

* action

  action argument sets up a signal handling method for the signal_num argument.

  If the action argument is a NULL pointer, the calling process's existing signal handling method for signal_num will be unchanged.

sigaction Example

This code segment sets the signal handler for SIGINT to `mySignalHandler()`

```
struct sigaction newAction;
int status;

newAction.sa_handler = mySignalHandler;  // Set the signal handler
newAction.sa_flags = 0;   // No special actions

status = sigemptyset(&newAction.sa_mask);
if (status == -1)
   perror("Failed to initialize signal set");
else
   {
   status = sigaction(SIGINT, &newAction, NULL);
   if (status == -1)
      perror("Failed to install signal handler for SIGINT");
   } // End else
```

Signal Handler

A signal handler is an ordinary function that returns `void` and has one integer parameter

When the operating system delivers the signal, it sets the parameter to the number of the signal that was delivered

Most signal handlers ignore this value, but it is possible to use the same signal handler for many signals

The usefulness of signal handlers is limited by the inability to pass values to them (this has been corrected for the sa_sigaction handler)

Two special values of the `sa_handler` member of the `struct sigaction` structure are `SIG_DFL` and `SIG_IGN`

- `SIG_DFL` specifies that the `sigaction()` function should restore the default action for the signal
- `SIG_IGN` specifies that the process should handle the signal by ignoring it (throwing it away)

Example use of SIG_IGN

The following code segment causes the process to change its new action to ignore the SIGINT signal if the default action is currently in effect for this signal

```
struct sigaction action;
int status;

status = sigaction(SIGINT, NULL, &action);
if (status == -1)
   perror("Failed to get old handler information for SIGINT");
else if (action.sa_handler == SIG_DFL)
   {
   action.sa_handler = SIG_IGN;
   status = sigaction(SIGINT, &action, NULL);
   if (status == -1)
      perror(
         "Failed to set SIGINT signal handler to ignore signal");
   } // End else
```

Example SIGINT Signal Handler

```
#include <signal.h>
#include <stdio.h>
#include <unistd.h>
#define MAX_PRESSES 5

static int pressCount = 0;

void catchCtrlC(int signalNbr);

int main(void)
{
// See next slide
}

// This function is the signal handler
void catchCtrlC(int signalNbr)
{
char message[] = "Ctrl-C was pressed\n";

write(STDERR_FILENO, message, strlen(message) );
pressCount++; // Global variable
} // End catchCtrlC
```

The `catchCtrlC()` function is defined here as a signal handler for the SIGINT signal generated by Ctrl-C.

The `write()` function is used instead of `fprintf()` because POSIX guarantees that it is async-signal safe, meaning that the function can be called safely from within a signal handler.

(More on next slide)

Example SIGINT Signal Handler (continued)

```
// ****************************************
int main(void)
{
struct sigaction action;
int status;

action.sa_handler = catchCtrlC;
action.sa_flags = 0;
status = sigemptyset(&action.sa_mask);
if (status == -1)
   {
   perror("Failed to initialize signal set");
   exit(1);
   } // End if
status = sigaction(SIGINT, &action, NULL);
if (status == -1)
   {
   perror("Failed to set signal handler for SIGINT");
   exit(1);
   } // End if

while (pressCount < MAX_PRESSES); // Loop has no statements
return 0;
} // End main
```

Sample Output

```
uxb3% a.out
Ctrl-C was pressed
Ctrl-C was pressed
Ctrl-C was pressed
Ctrl-C was pressed
Ctrl-C was pressed

uxb3%
```

kill API - kill and sigaction(1)

* Function

  - A process can send a signal to a related process via the ***kill*** API.

  - This is a simple means of interprocess communication or control.

  - The sender and recipient processes must be related such that either the sender process real or effective user ID matches that of the recipient process, or the sender process has superuser privileges.

* Include: *<signal.h>*

* Summary: *int **kill** (pid_t pid, int signal_num);*

* Return: Success: 0; Failure: -1; Sets errno: Yes

Arguments of ***kill*** - kill and sigaction (2)

* int signal_num: the integer value of a signal to be sent to one or more processes designated by pid.
* pid_t pid value
  - positive value: pid is process ID. Sends signal_num to that process.
  - 0: sends signal_num to all process whose group ID is the same as the calling process.
  - -1: Sends signal_num to all processes whose real user ID is the same as the effective user ID of the calling process.

Example: ***killpipe.c***

alarm API - alarm

* Function

  The ***alarm*** API requests the kernel to send the SIGALRM signal after a certain number of real clock seconds.

* Include: <*signal.h*>

* Summary: unsigned *int* ***kill*** *(unsigned int time_interval);*

* Return:

  Success: the number of CPU seconds left in the process timer; Failure: -1; Sets errno: Yes

* Argument

  time_interval: the number of CPU seconds elapsed time, after which the kernel will send the SIGALRM signal to the calling process.