Lecture 14. Signals

Signal Sets

Signals sets are data structures used to represent multiple signals. They are necessary in function calls such as `sigprocmask()` to tell the kernel not to allow any of the signals in the set to occur.

POSIX.1 defines the data type `sigset_t` to contain a signal set, for which the following system calls are implemented:

```
#include <signal.h>

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(sigset_t *set, int signo);
```

All 4 return 0 if OK, -1 if error.

Returns 1 if true, 0 if false.

The arguments are the signal set to modify, and the signal number of the specific signal (to add, delete, or check membership in the set for). The actions of these 5 functions are the following:

- `sigemptyset()`—initializes the signal set with ∅ (so all signals are excluded);
- `sigfillset()`—initializes the signal set with all signals;
- `sigaddset()`—adds signal with *`signo`* to the signal set;
- `sigdelset()`—removes signal with *`signo`* from the signal set;
- `sigismember()`—returns TRUE is *`signo`* signal is in the set, FALSE otherwise.

Most implementations have 32 signals, each being represented as a 1-bit in a 32-bit signal mask. Therefore, `sigemptyset()` zeroes all the bits in this mask, while `sigfillset()` turns on all the bits to 1.

The following program empties the signal set, then adds 3 signals to the signal set and prints them:

```
#include <stdio.h>
#include <signal.h>
#include <string.h>

void PrintSigset(sigset_t *set);

int main(void)
{
     sigset_t set;
     sigemptyset(&set);

     sigaddset(&set, SIGINT);
     sigaddset(&set, SIGHUP);
     sigaddset(&set, SIGALRM);

     PrintSigset(&set);
}

void PrintSigset(sigset_t *set)
{
     if (sigismember(set, SIGINT))
          printf("Signal SIGINT added to the set.\n");
     if (sigismember(set, SIGHUP))
          printf("Signal SIGHUP added to the set.\n");
     if (sigismember(set, SIGALRM))
          printf("Signal SIGALRM added to the set.\n");

     for (int i=1; i<32; i++)
     {
          if (sigismember(set, i))
               printf("Signal %d: %s\n", i, strsignal(i));
     }
}
```

This program produces the following output:

```
$ ./mysignalset
Signal SIGINT added to the set.
Signal SIGHUP added to the set.
Signal SIGALRM added to the set.
Signal 1: Hangup: 1
Signal 2: Interrupt: 2
Signal 14: Alarm clock: 14
```

The following program fills the signal set, and uses the `PrintSigset()` function to print out all the signals in the complete set:

```
#include <stdio.h>
#include <signal.h>
#include <string.h>

void PrintSigset(sigset_t *set);

int main(void)
{
     sigset_t set;
     sigfillset(&set);

     PrintSigset(&set);
}

void PrintSigset(sigset_t *set)
{
     for (int i=1; i<32; i++)
     {
          if (sigismember(set, i))
               printf("Signal %d: %s\n", i, strsignal(i));
     }
}
```

The `sigprocmask()` Function

The signal mask of a process is the set of signals currently blocked from delivery to that process. A process can examine/change its signal mask by using the following function:

```
#include <signal.h>

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

Returns 0 if OK, -1 if error.

Arguments:

- *`how`*—if the 2nd argument set is a non-null pointer, *`how`* specifies how the current signal mask is modified (see table below);
- *`set`*—if this is a non-null pointer, it represents a set acting as a modifier for the current signal set, through the *`how`* action;

- `oset`—if this is a non-null pointer, it returns by reference the current signal mask of the process.

how	Description
SIG_BLOCK	The new signal mask for the process is the union of its current signal mask and the signal set pointed to by set. That is, set contains the additional signals that we want to block.
SIG_UNBLOCK	The new signal mask for the process is the intersection of its current signal mask and the complement of the signal set pointed to by set. That is, set contains the signals that we want to unblock.
SIG_SETMASK	The new signal mask for the process is replaced by the value of the signal set pointed to by set.

The following program shows how to modify the signal mask of a process using `sigprocmask()`:

```
#include <stdio.h> /* for printf() */
#include <signal.h> /* for sigemptyset()... */
#include <string.h> /* for strsignal() */
#include <stdlib.h> /* for malloc() */

void PrintSigset(sigset_t *set);

int main(void)
{
     sigset_t *set=malloc(sizeof(sigset_t));
     sigset_t *oset=malloc(sizeof(sigset_t));
     sigemptyset(set);

     sigaddset(set, SIGINT);
     sigaddset(set, SIGHUP);
     PrintSigset(set);

     sigaddset(set, SIGALRM);
     sigprocmask(SIG_SETMASK, set, oset);
     PrintSigset(set);
}

void PrintSigset(sigset_t *set)
{
     printf("-----------------------------\n");
     for (int i=1; i<32; i++)
     {
          if (sigismember(set, i))
               printf("Signal %d: %s\n", i, strsignal(i));
     }
     printf("-----------------------------\n");
}
```

The output shows how the newly added signal will reflect into the signal set of the process:

```
$ ./mysigprocmask
-----------------------------
Signal 1: Hangup: 1
Signal 2: Interrupt: 2
-----------------------------
-----------------------------
Signal 1: Hangup: 1
Signal 2: Interrupt: 2
Signal 14: Alarm clock: 14
-----------------------------
```

The `sigpending()` Function

`sigpending()` returns the set of signals blocked from delivery and currently pending for the calling process:

```
#include <signal.h>

int sigpending(sigset_t *set);
```

Returns 0 if OK, -1 if error.

To showcase this function, we use the `sigprocmask()` call with the SIG_BLOCK *how* argument set, and then display the blocked signal with `sigpending()`:

```
#include <stdio.h> /* for printf() */
#include <signal.h> /* for sigemptyset()... */
#include <string.h> /* for strsignal() */
#include <stdlib.h> /* for malloc() */
#include <unistd.h> /* for alarm() */

void PrintSigset(sigset_t *set);
void handler(int signo);

int main(void)
{
     sigset_t *newset=malloc(sizeof(sigset_t));
     sigset_t *oldset=malloc(sizeof(sigset_t));
     sigset_t *pending=malloc(sizeof(sigset_t));
     sigemptyset(newset);

     sigaddset(newset, SIGINT);
     sigaddset(newset, SIGHUP);
     PrintSigset(newset);

     signal(SIGALRM, handler); /* 1. signal is set here */
     sigaddset(newset, SIGALRM); /* 2. signal is added to the
set */
     sigprocmask(SIG_BLOCK, newset, oldset); /* 3. signal is
blocked here */
     PrintSigset(newset);
     raise(SIGALRM); /* signal is triggered here */
     sleep(3); /* signal is delayed here */
     sigpending(pending); /* 4. signal fills in reference arg
"pending" */
     if (sigismember(pending, SIGALRM))
          printf("SIGALRM pending\n");
     PrintSigset(pending); /* 5. signal shown as set member */
     sigprocmask(SIG_SETMASK, oldset, NULL); /* 6. signal is
unblocked here */
     PrintSigset(pending);
}

void PrintSigset(sigset_t *set)
{
     printf("-----------------------------\n");
     for (int i=1; i<32; i++)
     {
          if (sigismember(set, i))
               printf("Signal %d: %s\n", i, strsignal(i));
     }
     printf("-----------------------------\n");
}

void handler(int signo)
{
        switch (signo)
        {
        case SIGALRM:
                printf("Received SIGALRM.\n");
          break;
        }
}
```

The output shows, in order: the original signal set, the signal set with the new signal SIGALRM added to it, the printout of the message right after the triggering of the signal, the contents of the pending signal set (just SIGALRM), the output from the handler, the newset still containing SIGALRM, and the set with SIGALRM deleted explicitly:

```
$ ./mysigpending
------------------------------
Signal 1: Hangup: 1
Signal 2: Interrupt: 2
------------------------------
------------------------------
Signal 1: Hangup: 1
Signal 2: Interrupt: 2
Signal 14: Alarm clock: 14
------------------------------
SIGALRM pending
------------------------------
Signal 14: Alarm clock: 14
------------------------------
Received SIGALRM.
------------------------------
Signal 1: Hangup: 1
Signal 2: Interrupt: 2
Signal 14: Alarm clock: 14
------------------------------
------------------------------
Signal 1: Hangup: 1
Signal 2: Interrupt: 2
------------------------------
```

The `sigaction()` Function

`sigaction()` allows the examination/modification of an action associated with a signal:

```
#include <signal.h>

int sigaction (int signo, const struct sigaction *act, struct
               sigaction *oact);
```

Returns 0 if OK, -1 if error.

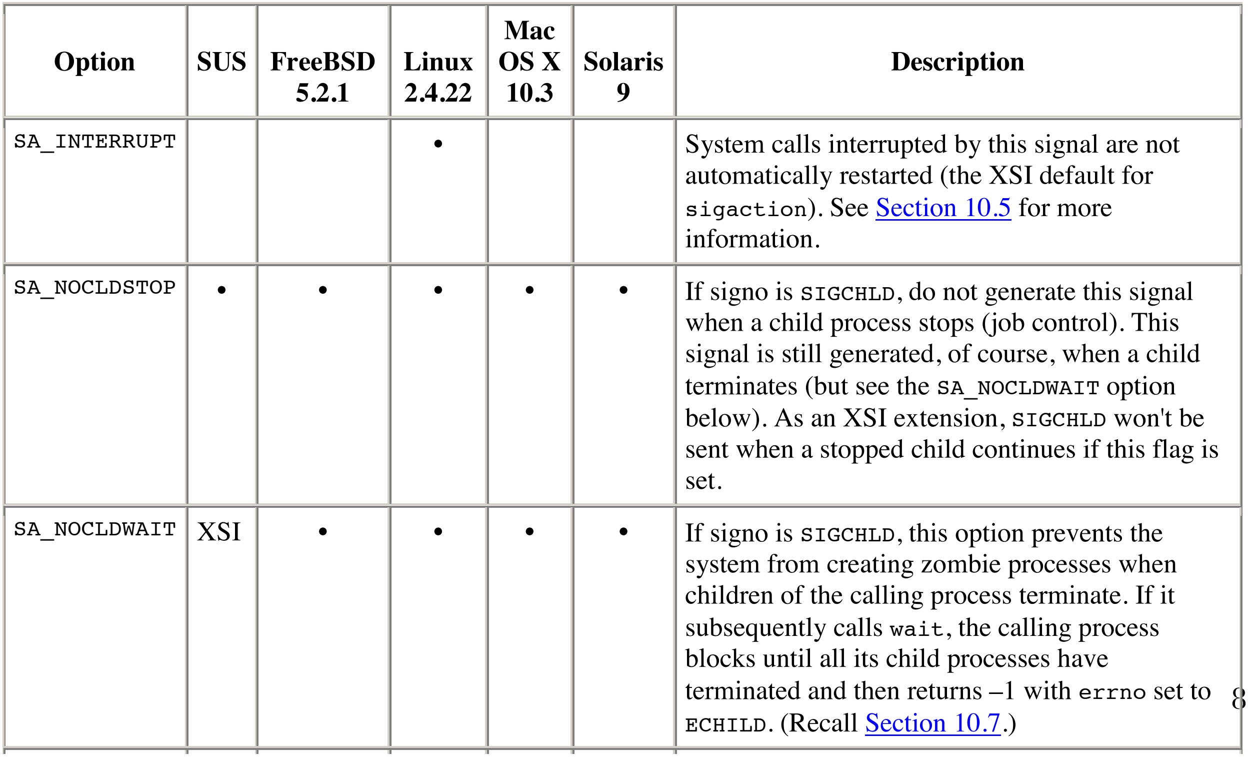
Arguments:
`signo`—the signal number whose action needs examination/modification;
`act`—the new action for the signal, if modified (i.e., if argument is non-null);
`oact`—the old action for the signal (gets old action by reference).

The function uses the following structure:

```
struct sigaction
{
     void (*sa_handler)();     /* pointer to handler(), SIG_IGN
                               (ignore), or SIG_DFL (default) */
     sigset_t sa_mask;         /* additional signals to block */
     int sa_flags;             /* signal options, tab. follows */
};
```

The structure contains additional signals to add to the signal mask of the process before the handler is called. When the handler exits, the signal mask is returned to the previous value. The additional signals associated with the signal action modification are thus available to blocking when calling this particular handler modifier. In particular, when calling `sigaction()` with a certain signal already handled by the primary handler, this signal can be blocked from further occurrences until the primary handler exits.

The 3rd argument signal options are listed below:



Option	SUS	FreeBSD 5.2.1	Linux 2.4.22	Mac OS X 10.3	Solaris 9	Description
SA_INTERRUPT			•			System calls interrupted by this signal are not automatically restarted (the XSI default for sigaction). See Section 10.5 for more information.
SA_NOCLDSTOP	•	•	•	•	•	If signo is SIGCHLD, do not generate this signal when a child process stops (job control). This signal is still generated, of course, when a child terminates (but see the SA_NOCLDWAIT option below). As an XSI extension, SIGCHLD won't be sent when a stopped child continues if this flag is set.
SA_NOCLDWAIT	XSI	•	•	•	•	If signo is SIGCHLD, this option prevents the system from creating zombie processes when children of the calling process terminate. If it subsequently calls wait, the calling process blocks until all its child processes have terminated and then returns –1 with errno set to ECHILD. (Recall Section 10.7.)

SA_NODEFER	XSI	•	•	•	•	When this signal is caught, the signal is not automatically blocked by the system while the signal-catching function executes (unless the signal is also included in sa_mask). Note that this type of operation corresponds to the earlier unreliable signals.
SA_ONSTACK	XSI	•	•	•	•	If an alternate stack has been declared with sigaltstack(2), this signal is delivered to the process on the alternate stack.
SA_RESETHAND	XSI	•	•	•	•	The disposition for this signal is reset to SIG_DFL, and the SA_SIGINFO flag is cleared on entry to the signal-catching function. Note that this type of operation corresponds to the earlier unreliable signals. The disposition for the two signals SIGILL and SIGTRAP can't be reset automatically, however. Setting this flag causes sigaction to behave as if SA_NODEFER is also set.
SA_RESTART	XSI	•	•	•	•	System calls interrupted by this signal are automatically restarted. (Refer to Section 10.5.)
SA_SIGINFO	•	•	•	•	•	This option provides additional information to a signal handler: a pointer to a siginfo structure and a pointer to an identifier for the process context.

The following program shows how SIGALRM can be added to the list of signals in the signal set of a process and its action modified according to sigaction():

```
#include <stdio.h> /* for printf() */
#include <signal.h> /* for sigemptyset()... */
#include <string.h> /* for strsignal() */
#include <stdlib.h> /* for malloc() */
#include <unistd.h> /* for alarm() */

void PrintSigset(sigset_t *set);
void handler(int signo);

sigset_t *newset, *oldset, *pending, *blocked;

int main(void)
{
     blocked=malloc(sizeof(sigset_t));
     sigemptyset(blocked);
     sigaddset(blocked, SIGINT);
     sigaddset(blocked, SIGHUP);
     struct sigaction *act=malloc(sizeof(sigaction));
     struct sigaction *oact=malloc(sizeof(sigaction));
     act->sa_handler=handler;
     act->sa_mask=*blocked;
     act->sa_flags=SA_RESTART;

     newset=malloc(sizeof(sigset_t));
     oldset=malloc(sizeof(sigset_t));
     pending=malloc(sizeof(sigset_t));
     sigemptyset(newset);

     sigaddset(newset, SIGINT);
     sigaddset(newset, SIGHUP);

     sigaction(SIGALRM, act, oact); /* signal is set here */
     sigaddset(newset, SIGALRM); /* signal is added here */
     printf("After adding SIGALRM...\n");
     PrintSigset(newset);
     raise(SIGALRM); /* signal is triggered here */
     raise(SIGINT);
     //sleep(3);
     sigpending(pending); /* SIGALRM fills in reference arg
"pending" w. blocked signals from handler() */
     if (sigismember(pending, SIGINT) && sigismember(pending,
SIGHUP))
          printf("SIGINT and SIGHUP pending.\n");
     printf("Signals pending:\n");
     PrintSigset(pending); /* pending signals are shown as
members of set "pending" */
     alarm(0);
/*
     sigprocmask(SIG_SETMASK, oldset, NULL);
     PrintSigset(newset);

     sigdelset(newset, SIGALRM);
     PrintSigset(newset);
*/
}

void PrintSigset(sigset_t *set)
{
     printf("----------------------------\n");
     for (int i=1; i<32; i++)
     {
          if (sigismember(set, i))
               printf("Signal %d: %s\n", i, strsignal(i));
     }
     printf("-----------------------------\n");
}

void handler(int signo)
{
        switch (signo)
        {
        case SIGALRM:
          printf("In handler()...\n");
                printf("Received SIGALRM.\n");
          sigprocmask(SIG_BLOCK, blocked, oldset); /* all other
signals are blocked here */
          printf("Signals blocked by SIGALRM's handler():\n");
          PrintSigset(blocked);
          sleep(3);
          break;
     }
}
```

The output shows the signals added to the blocked signal set passed to `sigprocmask()`, which is called from within the `handler()` function. Notice that the `handler()` is not called explicitly, rather, it is assigned to the `sa_handler` member of structure `act`, passed to `sigaction()`. System call `sigaction()` is, therefore, capable of adding a new signal set to be blocked while SIGALRM is being processed, while establishing the `handler()` as the disposition of this signal, just like `signal()`. In addition to passing the blocked signal set and the handler, it uses the `sigaction` structure to also pass one of a few built-in signal dispositions (table pp.[8-9]).

```
$ ./mysigaction
After adding SIGALRM...
-----------------------------
Signal 1: Hangup: 1
Signal 2: Interrupt: 2
Signal 14: Alarm clock: 14
-----------------------------
In handler()...
Received SIGALRM.
Signals blocked by SIGALRM's handler():
-----------------------------
Signal 1: Hangup: 1
Signal 2: Interrupt: 2
-----------------------------
```

The `sigsuspend()` Function

The `sigaction()` function was used to change the signal mask (the signal set) for a process to block (and unblock) selected signals. This technique can be also used to protect critical regions of code that we don't want interrupted by a signal.

To block a signal, while waiting for a previously blocked signal to occur, we could use the following code:

```
sigset_t newmask, oldmask;

sigemptyset(&newmask); /* initialize newmask to empty set */
sigaddset(&newmask, SIGINT); /* add SIGINT to newmask */

sigprocmask(SIG_BLOCK, &newmask, &oldmask); /* newmask consists
               now of SIGINT, and has all blocked signals */

/* CRITICAL REGION OF THE CODE */

sigprocmask(SIG_SETMASK, &oldmask, NULL); /* resets mask to old
                              value, before SIGINT was added */

pause(); /* wait for signal to occur */

/* continue program */
```

If the signal is sent while blocked, the delivery will be deferred until the signal is unblocked, that is, after the 2nd call to `sigprocmask()`. Thus, the signal appears to the program as if the signal occurred between the 2nd `sigprocmask()` and `pause()`. However, any occurrence of the signal in this window is lost: if the signal doesn't reoccur again during the call to `pause()`, `pause()` will block indefinitely, and the program will hang.

To solve this problem, the call to `sigprocmask()` and the call to `pause()` should be make a single atomic operation. This feature is provided by the `sigsuspend()` function:

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

Returns -1 with `errno` set to `EINTR`.

The signal set is passed in as `sigmask`. The process is suspended until a signal is caught or until a signal occurs that terminates the process. If the signal is caught and the handler returns, `sigsuspend()` returns with `-1`, and `sigmask` is set to its value before the call to `sigsuspend()`.

The `abort()` Function

This function causes abnormal program termination:

```
#include <stdlib.h>

void abort(void);
```

Never returns.

Homework (due Apr-5-2016): Write a simple program showing the system call `sigsuspend()`. Your program should initialize three sets of signals (oldset, newset, waitset) to the empty set, using `sigemptyset()`. It should then add a few signals to this set, using `sigaddset()`. It should contain a signal handler, which you can design to be shared across the few signals you have added to your set. You should then specify a set of blocked signals, using `sigprocmask()`. The signals you want triggered and handled, you should set, and establish handler(s) for, using either `signal()` or `sigaction()`. Then, you want the signals in the blocked mask to suspend until the signals from the wait mask are triggered and return from their handler, using `sigsuspend()`. After the signals in the wait mask have been delivered and handled, unblock the suspended signals using `sigprocmask()` again. Insert printouts of every point in your program where you can show the signal sets, where the signals are raised, and where the suspended signals get unblocked.