Signals

경희대학교 컴퓨터공학과

조 진 성

Signal Concepts

- Signals
  - ✓ are a sort of software interrupts for application processes
  - ✓ have names for each
  - ✓ provide a way of handling asynchronous events

- Conditions that generate a signal
  - ✓ Signals from terminal
    - e.g.) **`SIGINT, SIGQUIT`**
  - ✓ Signals from hardware exceptions
    - e.g.) **`SIGBUS, SIGFPE, SIGSEGV`**
  - ✓ Signals from software conditions
    - e.g.) **`SIGALRM, SIGCHLD, SIGPIPE`**
  - ✓ Signals from processes using **`kill()`** function
  - ✓ Signals from shells using **`kill`** command
    - Default signal: **`SIGTERM`**

Signal Concepts (Cont'd)

- Actions associated with a signal
  - ✓ Catch the signal
    - Register a signal handler
  - ✓ Let the default action apply if no signal handler is registered
    - Default action for most signals: termination of the process
    - SIGCHLD: ignore the signal
  - ✓ When a process calls **`fork()`**, the child inherits the parent's signal actions
- Major signals
  - ✓ SIGABRT
  - ✓ SIGALRM
  - ✓ SIGBUS
  - ✓ SIGCHLD
  - ✓ SIGCONT
  - ✓ SIGFPE
  - ✓ SIGHUP

Signal Concepts (Cont'd)

- Major Signals (Cont'd)
  - ✓ SIGILL
  - ✓ SIGINT
  - ✓ SIGIO
  - ✓ SIGKILL
  - ✓ SIGPIPE
  - ✓ SIGPOLL
  - ✓ SIGQUIT
  - ✓ SIGSEGV
  - ✓ SIGSTOP
  - ✓ SIGSYS
  - ✓ SIGTERM
  - ✓ SIGUSR1
  - ✓ SIGUSR2

Register a Signal Handler

- signal
  - ✓ `#include <signal.h>`
  - ✓ `void (*signal(int signo, void (*func)(int)))(int);`
  - ✓ return: previous signal handler if OK, **`SIG_ERR(-1)`** on error

```
#include <stdio.h>
#include <signal.h>
void SigIntHandler(int signo)
{
    printf("Received a SIGINT signal\n");
    /* Process the signal */
    exit(0);
}
main()
{
    signal(SIGINT, SigIntHandler);
    /* Do something */
}
```

Exercise

- Simple example for signal

```
$ gcc –o sig1 sig1.c (or make sig1)
$ ./sig1
^C
```

- Another example

```
$ gcc –o sig2 sig2.c (or make sig2)
$ ./sig2

$ ps –ef | grep cjs
$ kill –USR1 13418
$ kill –USR2 13418
$ kill –TERM 13418
```

Unreliable Signals

- Problems
  - ✓ The action for a signal may be reset to its default, each time the signal occurred
    - In the following example, the process may be terminated
    - When another signal occurs before the call to signal in signal handler
    - True in Solaris, False in Linux
  - ✓ Signals are stateless, i.e., UNIX don't remember if they do occur
    - In the following example, the process may sleep forever
    - When the signal occurs after the test of SigIntFlag, but before pause()

```
#include <signal.h>
int SigIntFlag;
int SigIntHandler(int signo)
{
    signal(SIGINT, SigIntHandler);
    SigIntFlag = 1;
}
main()
{
    signal(SIGINT, SigIntHandler);
    while (SigIntFlag == 0)  pause();
    .....
}
```


Exercise

- An example for unreliable signals

```
$ gcc –o sig2 sig2.c (or make sig2)
$ ./sig2

$ ps –ef | grep cjs
$ kill –USR1 13418
$ kill –USR1 13418
$ (What happens in Linux? How about in Solaris?)
```

Send a Signal

- Send a signal to a process or a group of processes
  - ✓ `#include <sys/types.h>`
  - ✓ `#include <signal.h>`
  - ✓ `int kill(pid_t pid, int signo);`
  - ✓ return: 0 if OK, –1 on error
  - ✓ The first argument, **`pid`**
    - if **`pid > 0`**, send to the process of which the process ID is pid
    - if **`pid == 0`**, send to all processes of which the group ID equals the sender's
    - if **`pid < 0`**, send to all processes of which the group ID is the absolute value of pid

- Send a signal to itself
  - ✓ `#include <sys/types.h>`
  - ✓ `#include <signal.h>`
  - ✓ `int raise(int signo);`
  - ✓ return: 0 if OK, –1 on error

Other System Calls related with Signals

- Sleep for the specified number of seconds
  - ✓ `#include <unistd.h>`
  - ✓ `unsigned int sleep(unsigned int seconds);`
  - ✓ return: 0 or number of seconds until previously set alarm(i.e. unslept time)

- Set an alarm clock for delivery of a signal
  - ✓ `#include <unistd.h>`
  - ✓ `unsigned int alarm(unsigned int seconds);`
  - ✓ return: 0 or number of seconds until previously set alarm(i.e. unslept time)

- Wait for signal
  - ✓ `#include <unistd.h>`
  - ✓ `int pause(void);`
  - ✓ return: –1 with **`errno`** set to **`EINTR`**

Exercise

- Make my own **sleep** system call using **signal** & **pause** system calls

```
$ gcc -o mysleep mysleep.c (or make mysleep)
$ ./mysleep
```

- An example for periodic alarms

```
$ gcc -o alarm alarm.c (or make alarm)
$ ./alarm
```


Interrupted System Calls

- If a process catch a signal while it has been blocked in a system call,
  - ✓ the system call returns an error and **errno** is set to **EINTR**
  - ✓ it's a good chance to wake up the blocked system call
  - ✓ so, you have to handle it as follows:

```
.....
while (1)  {
    if ((n = read(fd, buf, MAX_BUF)) < 0)  {
        if (errno == EINTR)
            continue;
        /* handle other errors */
    }
    .....
}
```

Reentrant Functions

- Reentrant function
  - ✓ A function which multiple processes are allowed to enter concurrently
  - ✓ A function which a process is allowed to enter nestedly

  - ✓ Reentrant functions don't handle sharable data
  - ✓ For example,
    - **`strcpy()`** : reentrant function
    - **`getpwnam()`** : non-reentrant function

- When writing multi-threaded applications,
  - ✓ you MUST NOT call non-reentrant functions

- In asynchronous event handlers,
  - ✓ (i.e., interrupt handlers, signal handlers, or exit handlers)
  - ✓ you MUST NOT call non-reentrant functions

Exercise

- An example for calling a non-reentrant function from a signal handler

```
$ gcc –o nonreent nonreent.c (or make nonreent)
$ ./nonreent
```

Signal Handling in Threads

- Which thread should handle signals?
  - ✓ To the thread to which the signal applies
  - ✓ To every thread in the process
  - ✓ To certain threads in the process
  - ✓ Assign a specific thread to receive all signals for the process
    - Solaris' implementation: main thread receives signals

Exercise

- Signals in threads

```
$ gcc –o sig_thread sig_thread.c –lpthread (or make sig_thread)
$ ./sig_thread

$ (How about in Linux?)
```


Kill Another Thread (Thread Cancellation)

- Send a cancellation request
  - ✓ `#include <pthread.h>`
  - ✓ `int pthread_cancel(pthread_t tid);`
  - ✓ return: 0 if OK, non-zero on error

- Set the type and state of cancellation request
  - ✓ `#include <pthread.h>`
  - ✓ `int pthread_setcancelstate(int state, int *oldstate);`
  - ✓ `int pthread_setcanceltype(int type, int *oldtype);`
  - ✓ return: 0 if OK, non-zero on error
  - ✓ The first argument in **`pthread_setcancelstate()`**, **`state`**
    - **`PTHREAD_CANCEL_DISABLE, PTHREAD_CANCEL_ENABLE`**(by default)
  - ✓ The first argument in **`pthread_setcanceltype()`**, **`type`**
    - **`PTHREAD_CANCEL_ASYNCHRONOUS, PTHREAD_CANCEL_DEFERRED`**(by default)

Exercise

- Cancel thread executions

```
$ gcc -o cancel cancel.c -lpthread (or make cancel)
$ ./cancel
```

Summary

- Signals
  - ✓ are a sort of software interrupts for application processes
  - ✓ provide a way of handling asynchronous events
  - ✓ **`SIGINT, SIGALRM, SIGCHLD, SIGUSR1, …`**

- System calls in Linux for signals
  - ✓ **`signal`**
  - ✓ **`kill, raise`**
  - ✓ **`sleep, alarm, pause`**
  - ✓ **`pthread_cancel`**
  - ✓ **`pthread_setcancelstate, pthread_setcanceltype`**