Embedded Systems Programming

Signaling
(Module 24)

Yann-Hang Lee
Arizona State University
yhlee@asu.edu
(480) 727-7507

Summer 2014

Signals

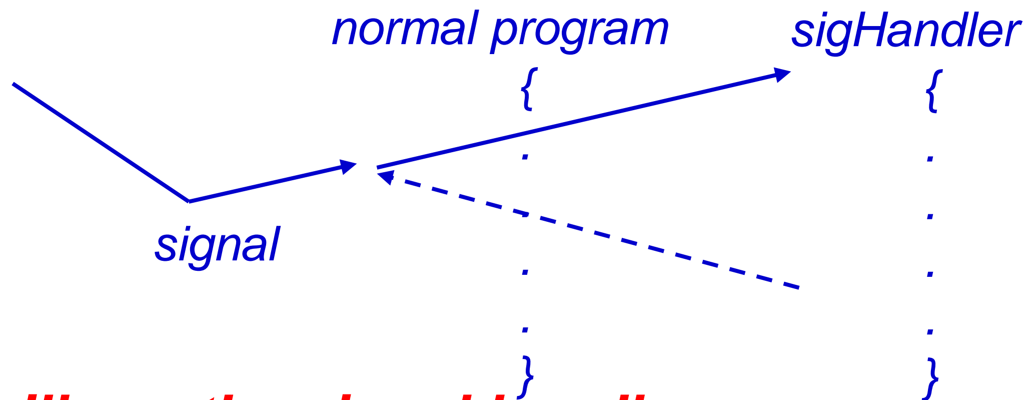
- **A signal is an event generated by OS in response to some condition (from processes or IO).**
- **Upon receipt of a signal a process or thread may take some action.**
- **Signal generation:**
  - by error conditions or external events (memory segment violations, floating point processor errors, illegal instructions)
  - by one process (thread) to another to send information
- **Signals may be generated asynchronously to thread execution**
  - queued? argument?
  - pending or blocked

Signal Handler

- To register a handler -- signal (signo, sigHandler)

  *void sigHandler (int sig, int code, struct sigcontext * pSigCtx);*

- Exception: OS issues a signal to the running task
  - if no signal handler, suspend the task
  - hardware dependent
  - return with *exit(), taskRestart(), longjump(), or return*



- ***Who will run the signal handler***

Example

```
#include <signal.h>      /* signal name macros, and the signal() prototype */

sig_atomic_t sigusr1_count = 0;

void handler (int signal_number)
    {
        ++sigusr1_count;
    }

int main ()
{
    struct sigaction sa;
    memset (&sa, 0, sizeof (sa));
    sa.sa_handler = &handler;
    sigaction (SIGUSR1, &sa, NULL);
    /* ... */
    printf (“SIGUSR1 was raised %d times\n”, sigusr1_count);
    return 0;
}
```

Linux and POSIX Signals

- **POSIX standard and real-time signals, and other signals**
- **Program Error Signals –SIGSEGV, SIGFPE, SIGILL, SIGABRT, etc.**
  - generated when a serious program error is detected
- **Termination Signals – SIGHUP, SIGQUIT, SIGKILL, SIGTERM …**
  - to tell a process to terminate
- **Alarm Signals – SIGALRM, …**
  - to indicate the expiration of timers.
- **Asynchronous I/O Signals – SIGIO, SIGURG.**
- **Job Control Signals**
- **Miscellaneous Signals – SIGPIPE, SIGUSR1, SIGUSR2**
- **If both standard and real-time signals are pending,**
  - POSIX leaves it unspecified which is delivered first.
  - Linux gives priority to standard signals

In User Mode

- **Write a signal handler function, e .g. handle SIGINT**

```
void sigint_handler(int sig) {
        fprintf(stderr,”Interrupted!\n”);
    }
```

- **Install it:**

```
struct sigaction new_action, old_action;
new_action.sa_handler = sigint_handler;
sigaction(SIGINT, &new_action, &old_action);
```

- ***Struct sigaction***

```
struct sigaction {
    void (*sa_handler)(int);
    void (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t sa_mask;
    int sa_flags;
    void (*sa_restorer)(void);
  }
```

Some Issues on Signals

- When the handler will run after the signal is delivered
- Access variables in handler
- Handler may execute at any time
  - Need to be careful of manipulating global state in signal handler
  - Signal delivery may interrupt execution of handler – reentrant
  - may make system calls
- Should block signals if this is not acceptable
- Only one signal handler per signal per process
  - delivered to one thread (arbitrary one)
- Can't use in library code
- In many implementations, no signal queuing

Alarm System Call

unsigned alarm(unsigned seconds);

- **Requests the system to generate a SIGALRM for the process after *seconds* time have elapsed.**
  - If *seconds* is 0, a pending alarm request, if any, is canceled.
  - Alarm requests are not stacked; only one *SIGALRM* generation can be scheduled in this manner.
  - If the *SIGALRM* signal has not yet been generated, the call shall result in rescheduling the time at which the *SIGALRM* signal is generated.
- **Processor scheduling delays may prevent the process from handling the signal as soon as it is generated.**