Programming Signals

A. A Review of Interrupts:

1. A *hardware interrupt*:

    a. A signal sent to the processor from a device that is part of the computer itself or from some other external device.

    b. Causes the CPU to stop its current task and transfer execution to the kernel.

    c. The act of initiating a hardware interrupt is referred to as an *interrupt request* (a.k.a. IRQ).

    d. An IRQ has a *type* associated with it that is determined by where and/or how it originated.

    e. Example

    The disk controller indicating that a read or write operation has completed.

    f. Example – External device

    Clicking the mouse or entering a character at the keyboard.

    g. The kernel *services* the IRQ using a table of pointers in memory, where each pointer represents the start address of the *interrupt service routines* associated with each type.

2. A *software interrupt*:

a. Caused by an exceptional condition in the CPU (a.k.a. *trap* or *exception*) or a special instruction in the instruction set that causes an interrupt when it is executed.

b. Could be an error or event during execution that is exceptional or serious enough that it cannot be handled within the currently executing task.

c. Causes the CPU to stop its current task and transfer execution to the kernel.

d. Example

   If the CPU's arithmetic logic unit attempts to divide a number by zero.

e. Example

If an attempt is made to reference memory with an uninitialized pointer.

f. Could be a system call to request services from low-level system software, such as device drivers.

g. Example

   An `open`, `read`, or `write` instruction results in a *system call* that causes the CPU to stop its current task, trap to the *system call interface*, and transfer execution to the kernel.

h. Example

A *signal*.

i. Signals can occur in the middle of executing an instruction in an application program, so care is required when using them (i.e., such as whether the interrupted code is *re-entrant*).

- Re-entrant functions are those that can be called more than once by the same process or simultaneously by multiple processes.

- A re-entrant function is written so that none of its code is modifiable (no values are changed) and it does not keep track of anything.

- The calling programs keep track of their own progress (variables, flags, etc.), thus one copy of the re-entrant code can be shared by any number of users or processes.

j. Functions that are re-entrant do not:

- Use static internal data structures (which could be modified by the signal handler).

- Call `malloc` or `free` (which stores allocated areas in a linked list that could be corrupted by the signal handler).

- Use the standard I/O library (the standard I/O library uses some global data structures which could be modified by the signal handler).

B. Signals

1. A *signal* is software interrupt mechanism that generates a notification indicating to a process that some event has occurred.

2. Every signal has a name and is associated with an integer-valued number.

3. Example – Partial list of Linux signals (could be different than UNIX, other Linuxes, or other OSs as values are system-dependent (can be found in `/usr/include/bits/signum.h` on Linux machines in CL115).

```
#define SIGHUP    1      /* hangup (POSIX) */
```

```
#define SIGINT    2      /* terminal interrupt (ANSI) */
#define SIGQUIT   3      /* terminal quit (POSIX) */
#define SIGILL    4      /* illegal instruction (ANSI) */
#define SIGTRAP   5      /* trace trap  (POSIX) */
#define SIGABRT   6      /* abort (4. 2 BSD) */
#define SIGBUS    7      /* bus error (4.2 BSD) */
#define SIGFPE    8      /* floating point exception (ANSI) */
#define SIGKILL   9      /* kill (can't be caught or ignored) (POSIX) */
#define SIGUSR1   10          /* user defined signal 1 (POSIX) */
#define SIGSEGV   11          /* segmentation violation (ANSI) */
#define SIGUSR2   12          /* user defined signal 2 (POSIX) */
#define SIGPIPE   13          /* write on a pipe with no reader (POSIX) */
#define SIGALRM   14          /* alarm clock (POSIX) */
#define SIGTERM   15          /* termination signal from kill (ANSI) */
#define SIGSTKFLT 16          /* stack fault */
#define SIGCHLD   17          /* child status change */
#define SIGCONT   18          /* if stopped, continue executing (POSIX)
#define SIGSTOP   19          /* stop (can't be caught or ignored) (POSIX) */
#define SIGTSTP   20          /* terminal stop (POSIX) */
#define SIGTTIN   21          /* background process trying to read from terminal (POSIX)
            */
#define SIGTTOU   22          /* background process trying to write to terminal (POSIX)
            */
#define SIGURG    23          /* urgent condition related to socket (4.2 BSD) */
#define SIGXCPU   24          /* cpu limit exceeded (4.2 BSD) */
#define SIGXFSZ   25          /* file size limit execeeded (4.2 BSD) */
#define SIGVTALRM 26          /* virtual alarm clock (4.2 BSD) */
#define SIGPROF   27          /* profiling alarm clock (4.2 BSD) */
#define SIGWINCH  28          /* window size change (4.3 BSD) */
#define SIGIO     29          /* I/O now possible (4.2 BSD) */
```

```
#define SIGPWR    30          /* power failure restart (System V) */
```

4. Several events can cause a signal to be generated:

   a. *Hardware exceptions*: The conditions are detected by the hardware, which notifies the kernel, which generates the appropriate signal, which is sent to the appropriate process. Examples include:

      - Division by zero (i.e., `SIGFPE`).

      - Invalid memory reference (i.e., `SIGSEGV`).

   b. *Software conditions*: When an event happens that a process should know about. Examples include:

      - Writing to a pipe that has no reader (i.e., `SIGPIPE`).

      - When a timer set by a process expires (i.e., `SIGALRM`).

      - When some user-defined condition occurs (i.e., `SIGUSR1`).

   c. *Terminal-generated signals*: When a user presses keys simultaneously in particular combinations. Examples include:

      - `Control/C` to stop a runaway process (i.e., `SIGINT`).

      - `Control/Z` to suspend a process running in foreground (i.e., `SIGTSTP`).

   d. *The* `kill/sigqueue` *system calls* (more on these later): To send any signal from a user-owned process to any other user-owned process.

e. *The* `kill` *command*: This is a command line interface to the kill system call to enable a signal to be sent from the shell to a (typically) runaway background process.

5. There are *two* generations of signals (at least for the purposes of our discussion there is):

    a. *Unreliable*: A throwback to the very early versions of signals in UNIX that have been superseded by the POSIX signals standard.

    b. *Reliable*: A (modern) version of signals adhering to the POSIX signals standard.

6. Unreliable signals suffer from a number of problems and should not be used in new programs:

    a. They can get lost (i.e., a signal could be sent but the intended recipient misses it).

    b. The disposition of a signal set by a process must be reset by the process each time the signal is received.

        - If the disposition is to catch the signal (with a *signal handler*), but the default action is to kill the process, there is a small window of time where the default action would be enabled until the process resets it again.

        - Another example of a *race condition*.

    c. They handling of a signal cannot be deferred, only ignored.

7. Reliable signals solve the problems with unreliable signals.

    a. The disposition of a signal set by a process is not reset to the default each time a signal is received, only when the process specifically changes it.

    b. Processes have the ability to both ignore or temporarily block signals.

- When a signal is blocked by a process, the kernel places it on a queue of pending signals for that process.

- A blocked signal remains pending until the process unblocks it or changes its disposition to ignore it.

- `SIGKILL` and `SIGSTOP` cannot be blocked.

8. From here on, we assume the use of reliable signals.

9. A signal will always be in one of *three* possible states:

   a. A signal is *generated* (i.e., sent to a process) when the event that causes the signal occurs.

   b. A signal is *pending* (i.e., blocked) if it has been generated but not delivered.

   c. A signal is *delivered* when the action associated with the signal is actually invoked.

10. The *lifetime* of a signal is the interval between its generation and delivery.

11. Signals may be generated in *two* ways:

    a. *Synchronously*: When an event occurs that is directly caused by the execution of a process' code (also called a *trap*) (e.g., `SIGFPE`).

    b. *Asynchronously*: When an event occurs at a seemingly random time with respect to the process (e.g., `SIGKILL`).

12. A process can respond to the receipt of a signal (called the signal's *disposition* or *associated action*) in *two* ways when it is delivered:

a. *Catch it*: Call a signal handler, a user-written function contained in a process that describes how the event should be handled. Examples include:

   - Catching `SIGTERM` (the default termination signal sent by the `kill` command) to release memory and delete temporary files.

   - Catching `SIGCHLD` to catch the termination of a child process.

b. *Take one of five possible default actions*:

   - Ignore the signal (it is possible to ignore certain signals generated by a hardware exception (e.g., `SIGFPE`), but process behaviour may become difficult to understand).

   - Terminate the process.

   - Core dump.

   - Stop if the process is currently running.

   - Continue if the process is currently stopped.

13. Generate signals from the shell with the `kill` command.

   a. List the symbolic names of the signals available (POSIX).

```
kill -l
```

   b. Kill a particular process (POSIX).

```
kill -s signal_name pid
```

c. Example

```
kill -s USR1 3423
```

d. Traditional `kill` command (still supported by POSIX, but only because of widespread usage).

```
kill -signal_name pid
kill -signal_number pid
```

e. Example

```
kill -KILL 3423
kill -9 3423
```

14. The `kill` system call is used to send a signal to a process.

```
#include <sys/types.h>
#include <signal.h>

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

a. The `kill` system call sends the signal specified by *sig* to the process specified by *pid*.

b. The first parameter, *pid*, is a valid process identifier (can actually have other values, but we don't discuss them here).

c. The second parameter, *sig*, must be a valid signal name or `0`.

- If `sig` is `0`, (i.e., the `NULL` signal), normal error checking is performed, *but no signal is actually sent*.

- Why would we want to do this? We can use `0` to check whether *pid* is a valid process before we actually try to kill it.

d. If successful, `kill` returns `0`. If unsuccessful, `kill` returns `-1` and sets `errno`.

e. Example – Child killing its parent.

PROGRAM = `childKillsParent.c`

15. The POSIX signal handling interface makes use of *signal sets* rather than individual signals.

a. A signal set is a *bit mask*, one bit for each signal.

b. If a bit is set to `0` (`1`), the corresponding signal is not (is) a member of the set.

c. Since the number of different signals can exceed the number of bits in an `int`, a signal set is of the `sigset_t` data type (defined in `signal.h`).

16. Signal sets can be created and deleted using the following *five* functions:

```
#include <signal.h>

int sigemptyset (sigset_t *signal_set);
int sigfillset (sigset_t *signal_set);  (Won't be discussed further in this course)
int sigaddset (sigset_t *signal_set, int signal_number);
int sigdelset (sigset_t *signal_set, int signal_number);  (Won't be discussed further in this course)
```

All four return `0` on success. Otherwise, they return `-1`.

`int sigismember (sigset_t *`*signal_set*`, int `*signal_number*`);` (Won't be discussed further in this course)

It returns `1` if true. Otherwise, it returns **`0`**.

17. The `sigemptyset` function initializes the signal set pointed to by *signal_set* to exclude all signals (i.e., the empty set).

18. Example

```
sigset_t interruptMask;

sigemptyset (&interruptMask);
```

19. The `sigaddset` function adds the single signal specified by *signal_number* to the signal set pointed to by *signal_set*.

20. Example

```
sigaddset (&interruptMask, SIGRTMIN);
```

21. The action associated with a signal in a signal set can examined and modified using the following *two* functions:

```
int sigaction (int signal_number, const struct sigaction *action, struct sigaction *old_action);
sigprocmask (int what_to_do, const sigset_t *signal_set, sigset_t *old_signal_set);
```

22. The `sigaction` function allows for the examination and modification of the action associated with a particular signal.

a. A `sigaction struct` consists of *four* members:

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

b. If `sa_flags` is set to `SA_SIGINFO`, then *three* arguments can be passed to the signal handler if its function prototype is declared as follows (enables the *receiving* process to determine the identity of the *sending* process):

```
static void signal_handler (int signal_number, siginfo_t *signal_info, void *context);
```

c. The `siginfo_t` data type provides information about why a signal was generated and where it originated.

```
typedef struct
{
    int si_signo;
    int si_errno;
    int si_code;
    pid_t si_pid;
    uid_t si_uid;
    .
    .
    .
} siginfo_t;
```

23. Example

```
struct sigaction act;
.
.
.
act.sa_sigaction = &SignalHandler;
act.sa_flags = SA_SIGINFO;
sigemptyset (&act.sa_mask);
sigaction (SIGRTMIN, &act, NULL);
.
.
.
```

24. The `sigprocmask` function allows for the examination and modification of the signal mask stored in the `sigaction struct`.

25. Example

```
.
.
.
sigprocmask (SIG_BLOCK, &interruptMask, NULL);
.
.
.
```
do something you don't want interrupted
```
.
.
.
```

```
sigprocmask (SIG_UNBLOCK, &interruptMask, NULL);
.
.
.
```

26. The `sigqueue` function is an extension to the `kill` function that put signals in a queue.

```
int sigqueue (pid_t pid, int signal_number, const union sigval value);
```

   a. Returns `0` on success. Othewise, it returns `-1`.

27. Example

```
union sigval dummyValue;

sigqueue (pid, SIGRTMIN, dummyValue);
```

28. The `pause` system call suspends a process until a signal is delivered.

```
#include <unistd.h>

int pause (void);
```

   a. The `pause` system call suspends the calling process until it receives a signal that it is not currently ignoring.

   b. The `pause` system call returns `-1` and sets `errno` when a signal handler is executed.

   c. If the disposition of the received signal is to terminate, `pause` does not return. Otherwise, the process continues executing from where it was suspended.

29. Example – A process catching a signal sent from the command line.

PROGRAM = `catchSignals.c`

30. Example – A process catching a signal sent from a child.

PROGRAM = `parentCatchSignals.c`