# Signals and Terminal I/O

- Signals and how to control the program behavior in handling signals.
- Terminal I/O

- Readings
  - APUE 10.2-10.5, 10.9 - 10.14

# Signals

- A form of inter-process communication
- tells a process that some event occurs
  - the kill command
    - “kill –l”
    - “kill –s INT pid”
    - “kill –INT pid”
  - Type “ctrl-C” when a program is running (SIGINT)
  - Memory violation (SIGSEGV), etc
  - Divided by 0 (SIGFPE).
  - A process dies (SIGHUP and SIGCHLD).
  - a packet arrives
  - ……

# Some Commonly Used Signals

- SIGABRT, SIGALRM, SIGCHLD, SIGHUP, SIGINT, SIGUSR1, SIGUSR2, SIGTERM, SIGKILL, SIGSTOP, SIGSEGV, SIGILL
- All defined in signal.h
- `man -s 7 signal` on linprog

# Signal

- When a process receives a signal, it performs one of the following three options
  - Ignore the signal
    - Two signals cannot be ignored
    - SIGKILL and SIGSTOP
  - Perform the default operation
    - Ignore the signal
    - exit
  - Catch the signal
    - Informs kernel to call user-defined function when signal occurs
- We can also block a signal from happening
  - Kernel remembers if a signal occurs and deliver it when we unblock the signal

# Catch a Signal

- Similar to interrupt (software interrupt)
- When a process receives a signal:
  - stop execution
  - call the signal handler routine
  - continue
- Signal can be received at any point in the program.

- Most default signal handlers will exit the program.

# ANSI C signal Function

- syntax:
  - #include <signal.h>
  - void (*signal(int signo, void (*func)(int)))(int);
- semantic:
  - signo -- signal number (defined in signal.h)
  - func: SIG_IGN, SIG_DFL or the address of a signal handler
    - SIG_IGN: signore signal
    - SIG_DFL: perform default action
  - Handler may be erased after one invocation.
    - How to get continuous coverage?
    - Still have problems – may lose signals
- See example1.c (lingprog and program): not well defined

# Block/unblock Signal: sigprocmask

- Manipulate signal sets
  - #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(const sigset_t *set, int signo);
- Manipulate signal mask of a process
  - int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
    - How: SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK
  - See example2.c, example2a.c, example3.c

# Critical Region

- For a critical region where you don't want certain signal to come, the program will look like:

```
sigprocmask(SIG_BLOCK, &newmask,
  &oldmask);
……. /* critical region */
sigprocmask(SIG_SETMASK, &oldmask, NULL);
```

# The sigaction Function

- Supersedes the signal function + signal blocking
- #include <signal.h>

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

```
struct sigaction {
   void (*sa_handler)(); /* signal handler */
   sigset_t sa_mask; /*additional signal to be block  during
                       execution of signal handler*/
   int sa_flags;        /* various options for handling signal */
};
```

- See example4.c and example4a.c (noted the program behavior with multiple signals)

# The kill Function

- Send a signal to a process
- #include <signal.h>
- #include <sys/types.h>
- int kill(pid_t pid, int signo);
  - pid > 0, normal
  - pid == 0, all processes whose group ID is the current process's group ID.
  - pid == -1, all processes for which sender has permission to send
  - pid <-1, all processes whose group ID = |pid|
- See example5.c
- See example6.c for the use of alarm.

# Impact of Signals on System Calls

- A system call may return prematurely
- See example7.c
- How to deal with this problem?
  - Check the return value of the system call and act accordingly
  - Check the errno variable.

  - See example7a.c

# Terminal I/O

- The semantics of an output operation is relatively simple. Input is rather messy.
- Two input modes:
  - Canonical mode:  the default mode, input line by line
  - Noncanonical mode: input characters are not assembled.
- We will focus on noncanonical mode
  - When do we use it?
  - What should we expect when using this mode for input?
  - Example: Which input mode does vi use?
  - How about an “old” shell such as Bourne Shell?
  - How about a new shell such as bash that supports command-line editing?

# The termios Structure

- In POSIX.1, all the characteristics of a terminal device that we can examine and change are in a termios structure (termios.h)

```
struct termios {
    tcflag_t c_iflag;    /* input flag */
    tcflag_t c_oflag;   /* output flag */
    tcflag_t c_cflag;    /* control flags */
    tcflag_t c_lflag;     /* local flags */
    cc_t     c_cc[NCCS];  /* control characters */
}
```

# Manipulating termios Structure

- Functions to get and set the fields in the termios structure
  - tcgetattr and tcsetattr;
  - #include <termios.h>
  - int tcgetattr(int fildes, struct termios *termios_p)
  - int tcsetattr(int fildes, int optional_actions, const struct termios *termios_p)
    - optional_actions: TCSANOW, TCSADRAIN, TCSAFLUSH

# Noncanonical Mode

- Turn on the noncanonical mode:
  - Unset the ICANON flag in c_lflag
    - myterm.c_lflag & = ~ICANON
  - When will a read return using the noncanonical mode for input?
    - Number of characters(VMIN)
    - Time (VTIME)
    - Specified in the c_cc field
      - c_cc[VMIN] = ???, c_cc[VTIME] = ???
      - VMIN > 0, VTIME > 0
      - VMIN = 0, VTIME > 0
      - VMIN > 0, VTIME = 0
      - VMIN = 0, VTIME = 0
- See example8.c and example9.c