COMP 2355 Introduction to Systems Programming

Christian Grothoff
christian@grothoff.org

http://grothoff.org/christian/
Today

- Long jumps
- What are Signals?
- Using Signals
- Signals and gdb
Funky Control Flow

- `int setjmp(jmp_buf env)`
- `void longjmp(jmp_buf env, int val)`

> 99.999% of the time it is a very bad idea to use these functions!
Signals

• Signals are **software interrupts**

• Examples: illegal instruction, division by zero, segmentation violation, terminal closed, CTRL-C, etc.

• Possible actions: ignore, block (delay until unblocked), catch (call a signal handler) or die

• Not all actions are possible for all signals, each signal has a default action
Signal Generation

Signals can be generated by:

- Errors
- External events
- Explicit requests

You will see plenty of examples.
Signal Delivery

Once a signal is generated, it becomes pending.

- Delivery of blocked signals will be delayed until they are unblocked
- For certain signals, the action is fixed (example: SIGKILL, SIGSTOP)
- For other signals, the programmer can specify an action, including ignoring the signal
Running the Action

- Interrupt the currently executing code (may happen at any time!)
- Save the current registers (on the stack)
- Use the current stack to run the signal handler
- Return from signal handler restores registers and resumes original execution
Default Action

• Some signals have default actions

• Signals representing errors usually terminate the application, some with a core dump (if enabled)

• Some signals are ignored by default
#include <signal.h>
#include <setjmp.h>
sigjmp_buf jbuf;  int i;

static void handler(int sig) {
  printf("Oops!\n");
  i = 5;
  longjmp(jbuf, 1);
}

int main(int argc, char** argv) {
  if (0 != setjmp(jbuf)) return 1;
  signal(SIGFPE, &handler);
  return 1 / i;
}
static void handler(int sig) {
    printf("Oops!\n");
    exit(0);
}

int main(int argc, char** argv) {
    char * ptr;

    ptr = NULL;
    signal(SIGSEGV, &handler);
    ptr[3] = 4;
    return 0;
}
SIGBUS

int main(int argc, char** argv) {
    char a[20];
    long * l = &a[3];

    signal(SIGBUS, &handler);
    *l = 0; /* on certain processors only! */
    return 0;
}
SIGALRM

static void handler(int sig) {
    exit(0);
}
int main(int argc, char** argv) {
    signal(SIGALRM, &handler);
    alarm(4); // read the man-page!
    while (1) fprintf(stderr, ".");
    return 0;
}
SIGCHLD

pid_t child;
static void handler(int sig) {
    int status; waitpid(child, &status, 0); exit(0); }
int main(int argc, char** argv) {
    int i = 10;
    signal(SIGCHLD, &handler);
    child = fork();
    if (child == 0)
        while (--i) fprintf(stderr, "C");
    else
        while (1) fprintf(stderr, "P");
    return 0; }
SIGPIPE

static void handler(int sig) {
    printf("SIGPIPE!\n");
}

int main(int argc, char** argv) {
    int p[2];
    signal(SIGPIPE, &handler);
    pipe(p);
    close(p[0]);
    write(p[1], "Hello", 5);
    return 0;
}

External Signals

- SIGHUP, SIGINT, SIGQUIT, SIGTERM, SIGKILL
- SIGTRAP, SIGPROF
- SIGUSR1, SIGUSR2
- SIGSTOP, SIGCONT
Signal Handling

• `pid_t getpid()`

• `int kill(pid_t pid, int sig)`

• `int pause(void)` – usually `select` is better!

• `typedef void (*sighandler_t)(int)`

• `sighandler_t signal(int signum, sighandler_t handler)`
Modern Signal Handling

- `int sigaction(int signum, const struct sigaction * act, struct sigaction * old)`

  ```c
  struct sigaction {
    void (*sa_handler)(int)    // ...
    int sa_flags;
  }
  ```
Signals during Signal Handling

- Signals can arrive during the signal handler
- The signal that is currently handled is automatically blocked (and unblocked upon completion of the handler)
- `struct sigaction` can be used to specify additional signals that should be blocked
- If a signal arrives again before the previous signal was handled, the two signals maybe merged into one
Signal Handler Code

Because a signal handler maybe called at any time handler code must be careful with mutable state of the application – it maybe in an inconsistent state.

- Do not use malloc or free (they internally use global state!
- If you change global variables, consider declaring them volatile
Signals and System Calls

Signals may happen during system calls!

- Most system calls for IO will return an error code (-1)
- `errno` will be set to `EINTR`
- Always check return codes of all system calls!
- Often, `EINTR` should be handled by trying again
Exercise (Teams of two students are ok)

Write a program that interprets receiving the signals SIGUSR1 and SIGUSR2 as morse code (USR1 being dot, USR2 being dash) and prints the decoded text.

Write a second program that takes a PID as the first argument and sends a sequence of signals to the process with the given PID which corresponds to the morse code of the text read from stdin.

Test your code. The experience will be vital to answering quiz questions next time.
Questions