Signals

Guilherme Ottoni

Prof. David August

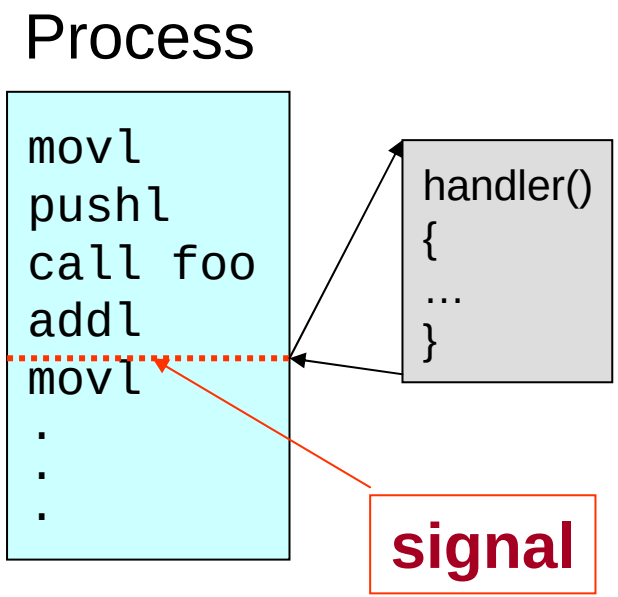
COS 217

Goals of Today's Lecture

- Overview of signals
  - Notifications sent to a process
  - UNIX signal names and numbers
  - Ways to generate signals
- Signal handling
  - Installing signal handlers
  - Ignoring vs. blocking signals
  - Avoiding race conditions
- Keeping track of the passage of time
  - Interval timers
  - Real time vs. CPU time

Signals

- Event notification sent to a process at *any* time
  - An event generates a signal
  - OS stops the process immediately
  - Signal handler executes and completes
  - The process resumes where it left off



Examples of Signals

- User types control-C
  - Event generates the “interrupt” signal (SIGINT)
  - OS stops the process immediately
  - Default handler terminates the process

- Process makes illegal memory reference
  - Event generates “segmentation fault” signal (SIGSEGV)
  - OS stops the process immediately
  - Default handler terminates the process, and dumps core

Sending Signals from Keyboard

- Steps
  - Pressing keys generates interrupts to the OS
  - OS interprets key sequence and sends a signal
- OS sends a signal to the running process
  - Ctrl-C → INT signal
    - By default, process terminates immediately
  - Ctrl-Z → TSTP signal
    - By default, process suspends execution
  - Ctrl-\ → ABRT signal
    - By default, process terminates immediately, and creates a core image

Sending Signals From The Shell

- **`kill -<signal> <PID>`**
  - Example: **`kill -INT 1234`**
    - Send the INT signal to process with PID 1234
    - Same as pressing Ctrl-C if process 1234 is running
  - If no signal name or number is specified, the default is to send an SIGTERM signal to the process,
- **`fg`** (foreground)
  - On UNIX shells, this command sends a **`CONT`** signal
  - Resume execution of the process (that was suspended with Ctrl-Z or a command "**`bg`**")
  - See man pages for **`fg`** and **`bg`**

Sending Signals from a Program

- The kill command is implemented by a system call

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
```

- Example: send a signal to itself

```
if (kill(getpid(), SIGABRT))
   exit(0);
```

  - The equivalent in ANSI C is:

```
int raise(int sig);

if (raise(SIGABRT) > 0)
      exit(1);
```

Predefined and Defined Signals

- Find out the predefined signals

```
% kill -l
% HUP INT QUIT ILL TRAP ABRT BUS FPE KILL
  USR1 SEGV USR2 PIPE ALRM TERM STKFLT CHLD
  CONT STOP TSTP TTIN TTOU URG XCPU XFSZ
  VTALRM PROF WINCH POLL PWR SYS RTMIN
  RTMIN+1 RTMIN+2 RTMIN+3 RTMAX-3 RTMAX-2
  RTMAX-1 RTMAX
```

- Applications can define their own signals
  - An application can define signals with unused values

Some Predefined Signals in UNIX

```
#define SIGHUP          1       /* Hangup (POSIX).  */
#define SIGINT          2       /* Interrupt (ANSI).  */
#define SIGQUIT         3       /* Quit (POSIX).  */
#define SIGILL          4       /* Illegal instruction (ANSI).  */
#define SIGTRAP         5       /* Trace trap (POSIX).  */
#define SIGABRT         6       /* Abort (ANSI).  */
#define SIGFPE          8       /* Floating-point exception (ANSI).  */
#define SIGKILL         9       /* Kill, unblockable (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      /* Broken pipe (POSIX).  */
#define SIGALRM         14      /* Alarm clock (POSIX).  */
#define SIGTERM         15      /* Termination (ANSI).  */
#define SIGCHLD         17      /* Child status has changed (POSIX).  */
#define SIGCONT         18      /* Continue (POSIX).  */
#define SIGSTOP         19      /* Stop, unblockable (POSIX).  */
#define SIGTSTP         20      /* Keyboard stop (POSIX).  */
#define SIGTTIN         21      /* Background read from tty (POSIX).  */
#define SIGTTOU         22      /* Background write to tty (POSIX).  */
#define SIGPROF         27      /* Profiling alarm clock (4.2 BSD).  */
```

Signal Handling

- Signals have default handlers
  - Usually, terminate the process and generate core image
- Programs can over-ride default for most signals
  - Define their own handlers
  - Ignore certain signals, or temporarily block them
- Two signals are not “catchable” in user programs
  - KILL
    - Terminate the process immediately
    - Catchable termination signal is TERM
  - STOP
    - Suspend the process immediately
    - Can resume the process with signal CONT
    - Catchable suspension signal is TSTP

Installing A Signal Handler

- Predefined signal handlers
  - **`SIG_DFL:`** Default handler
  - **`SIG_IGN:`** Ignore the signal
- To install a handler, use

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int sig, sighandler_t handler);
```

  - Handler will be invoked, when signal **`sig`** occurs
  - Return the old handler on success; **`SIG_ERR`** on error
  - On most UNIX systems, after the handler executes, the OS resets the handler to **`SIG_DFL`**

Example: Clean Up Temporary File

- Program generates a lot of intermediate results
  - Store the data in a temporary file (e.g., “temp.xxx”)
  - Remove the file when the program ends (i.e., unlink)

```
#include <stdio.h>

char *tmpfile = “temp.xxx”;
int main() {
    FILE *fp;

    fp = fopen(tmpfile, “rw”);

    …
    fclose(fp);
    unlink(tmpfile);
    return(0);
}
```

Problem: What about Control-C?

- What if user hits control-C to interrupt the process?
  - Generates a SIGINT signal to the process
  - Default handling of SIGINT is to terminate the process
- Problem: the temporary file is not removed
  - Process dies before **`unlink(tmpfile)`** is performed
  - Can lead to lots of temporary files lying around
- Challenge in solving the problem
  - Control-C could happen at any time
  - Which line of code will be interrupted???
- Solution: signal handler
  - Define a “clean-up” function to remove the file
  - Install the function as a signal handler

Solution: Clean-Up Signal Handler

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
char *tmpfile = “temp.xxx”;

void cleanup(void) {
   unlink(tmpfile);
   exit(EXIT_FAILURE);
}
int main(void) {
   if (signal(SIGINT, cleanup) == SIG_ERR)
      fprintf(stderr, “Cannot set up signal\n”);
  …
   return(0);
}
```

Ignoring a Signal

- Completely disregards the signal
  - Signal is delivered and "ignore" handler takes no action
  - E.g., **`signal(SIGINT, SIG_IGN)`** to ignore the ctrl-C
- Example: background processes (e.g., "a.out &")
  - Many processes are invoked from the same terminal
    - And, just one is receiving input from the keyboard
  - Yet, a signal is sent to *all* of these processes
    - Causes *all* processes to receive the control-C
  - Solution: shell arranges to ignore interrupts
    - All *background* processes use the SIG_IGN handler

Example: Clean-Up Signal Handler

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
char *tmpfile = “temp.xxx”;

void cleanup(void) {
   unlink(tmpfile);
   exit(EXIT_FAI
}
int main(void) {
   if (signal(SIGINT, cleanup) == SIG_ERR)
      fprintf(stderr, “Cannot set up signal\n”);
  …
   return 0;
}
```

Problem: What if this is a background process that was *ignoring* SIGINT???

Solution: Check for Ignore Handler

- **`signal()`** system call returns previous handler
  - E.g., **`signal(SIGINT, SIG_IGN)`**
    - Returns **`SIG_IGN`** if signal was being ignored
    - Sets the handler (back) to **`SIG_IGN`**
- Solution: check the value of previous handler
  - If previous handler was “ignore”
    - Continue to ignore the interrupt signal
  - Else
    - Change the handler to “cleanup”

Solution: Modified Signal Call

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
char *tmpfile = “temp.xxx”;

void cleanup(void) {
   unlink(tmpfile);
   exit(EXIT_FAILURE);
}
int main(void) {
   if (signal(SIGINT, SIG_IGN) != SIG_IGN)
      signal(SIGINT, cleanup);
  …
   return(0);
}
```

Solution: If SIGINT was ignored, simply keep on ignoring it!

Blocking or Holding a Signal

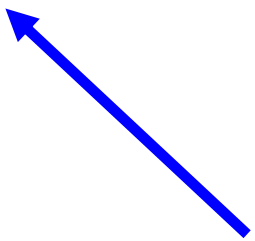
- Temporarily defers handling the signal
  - Process can prevent selected signals from occurring
  - … while ensuring the signal is not forgotten
  - … so the process can handle the signal later
- Example: testing a global variable set by a handler

```
int myflag = 0;

void myhandler(void) {
    myflag = 1;
}

int main(void) {
  if (myflag == 0)
      /* do something */
}
```

Problem: **myflag** might become **1** just *after* the comparison!

Race Condition: Salary Example

```
void add_salary_to_savings(void) {
    int tmp;
    tmp = savingsBalance;
    tmp += monthlySalary;
    savingsBalance = tmp;
}
```



Handler for “Monthly salary signal”

Race Condition: Handler Starts

```
void add_salary_to_savings(void) {
    int tmp;
    tmp = savingsBalance;
    tmp += monthlySalary;
    savingsBalance = tmp;
}
```

2000

Race Condition: Signal Again

```
void add_salary_to_savings(void) {
    int tmp;
    tmp = savingsBalance;
    tmp += monthlySalary;
    savingsBalance = tmp;
}
```

2000

```
void add_salary_to_savings(void) {
    int tmp;
    tmp = savingsBalance;
    tmp += monthlySalary;
    savingsBalance = tmp;
}
```

Race Condition: 2nd Handler Call

```
void add_salary_to_savings(void) {
    int tmp;
    tmp = savingsBalance;
    tmp += monthlySalary;
    savingsBalance = tmp;
}
```

2000

```
void add_salary_to_savings(void) {
    int tmp;
    tmp = savingsBalance;
    tmp += monthlySalary;
    savingsBalance = tmp;
}
```

2000

Race Condition: Back to 1st Handler

```
void add_salary_to_savings(void) {
    int tmp;
    tmp = savingsBalance;
    tmp += monthlySalary;
    savingsBalance = tmp;
}
```

2000

```
void add_salary_to_savings(void) {
    int tmp;
    tmp = savingsBalance;
    tmp += monthlySalary;
    savingsBalance = tmp;
}
```

2050

Race Condition: Lost Money!

```
void add_salary_to_savings(void) {
    int tmp;
    tmp = savingsBalance;
    tmp += monthlySalary;
    savingsBalance = tmp;
}
```

2000

2050

```
void add_salary_to_savings(void) {
    int tmp;
    tmp = savingsBalance;
    tmp += monthlySalary;
    savingsBalance = tmp;
}
```

You just lost a month's worth of salary!

Blocking Signals

- Why block signals?
  - An application wants to ignore certain signals
  - Avoid race conditions when another signal happens in the middle of the signal handler's execution
- Two ways to block signals
  - Affect all signal handlers

    `sigprocmask()`
  - Affect a specific handler

    `sigaction()`

Block Signals

- Each process has a signal mask in the kernel
  - OS uses the mask to decide which signals to deliver
  - User program can modify mask with **`sigprocmask()`**
- **`int sigprocmask()`** with three parameters
  - How to modify the signal mask (int how)
    - **`SIG_BLOCK:`** Add **`set`** to the current mask
    - **`SIG_UNBLOCK:`** Remove **`set`** from the current mask
    - **`SIG_SETMASK:`** Install **`set`** as the signal mask
  - Set of signals to modify **`(const sigset_t *set)`**
  - Old signals that were blocked **`(sigset_t *oset)`**
- **Functions for constructing sets**
  - **`sigemptyset(), sigaddset(), …`**

Example: Block Interrupt Signal

```
#include <stdio.h>
#include <signal.h>

sigset_t newsigset;

int main(void) {
   sigemptyset(&newsigset);
   sigaddset(&newsigset, SIGINT);
   if (sigprocmask(SIG_BLOCK, &newsigset, NULL) < 0)
      fprintf(stderr, “Could not block signal\n”);
   …
}
```

Problem: Blocking Signals in Handler

- Goal: block certain signals within a handler
  - Another signal might arrive at the start of the handler
  - … before calling the system call to block signals
- Solution: install handler and the mask together
  - Sigaction() system call, with three parameters
    - Signal number (int signum)
    - Set new signal action (const struct sigaction *)
    - Examine old signal action (struct sigaction *)
  - Sigaction data structure
    - Handler for the signal
    - Mask of additional signals to block during the handler
    - Flags controlling delivery of the signal

Sigaction() vs. Signal()

- Benefits of **`sigaction()`** over **`signal()`**
  - Set the mask and handler together
  - Examine the existing handler without reassigning it
  - Provide handler with information about process state
- You should not mix the two system calls
  - They interact in strange ways
  - In Assignment #7, we recommend using **`sigaction()`**

Keeping Track of Passage of Time

- Interval timers
  - Generate a signal after a specific time interval
- Real time (or wall-clock time)
  - Elapsed time, independent of activity
  - Not an accurate measure of execution time
  - Timer generates a SIGALRM signal
- Virtual time (or CPU time)
  - Time the process spends running
  - Doesn't include spent by other processes
  - Timer generates a SIGPROF signal

Interval Timer in Real Time

- Sends an SIGALRM signal after n seconds

```
unsigned int alarm(unsigned int seconds);
```

- Example

```
#include <signal.h>    /* signal names and API */
void catch_alarm(int sig) {
   if (signal(SIGALRM, catch_alarm) == SIG_ERR)
      ...;
   ...
}
int main(void) {
   if (signal(SIGALRM, catch_alarm) == SIG_ERR)
      ...;
   alarm(10);

   ...;
}
```

Interval Timer in CPU Time

- Send an signal after an interval timer expires

```
#include <sys/time.h>
int setitimer(int which,
  const struct itimerval *value,
  struct itimerval *ovalue);
```

- Example: SIGPROF signal every 10 milliseconds

```
struct itimerval timer;

timer.it_value.tv_sec = 0;
timer.it_value.tv_usec = 10000;    /* 10ms */
timer.it_interval.tv_sec = 0;
timer.it_interval.tv_usec = 10000; /* reload 10ms */
if (setitimer(ITIMER_PROF, &timer, NULL) == ...)
    ...;
```

- On Linux, the minimal effective granularity is **10ms**.

Conclusions

- Signals
  - An asynchronous event mechanism
  - Use **`sigaction()`** and blocking to avoid race conditions
  - Signal handlers should be simple and short
  - Most predefined signals are “catchable”
- Interval timers
  - Real time (SIGALRM) or CPU time (SIGPROF)
  - Linux imposes 10ms as the minimal granularity