Chapitre II Signal

Goals of Chapter

- Overview of signals

Notifications sent to a process

UNIX signal names and numbers

Ways to generate signals

2. Signal handling

Installing signal handlers

Ignoring vs. blocking signals

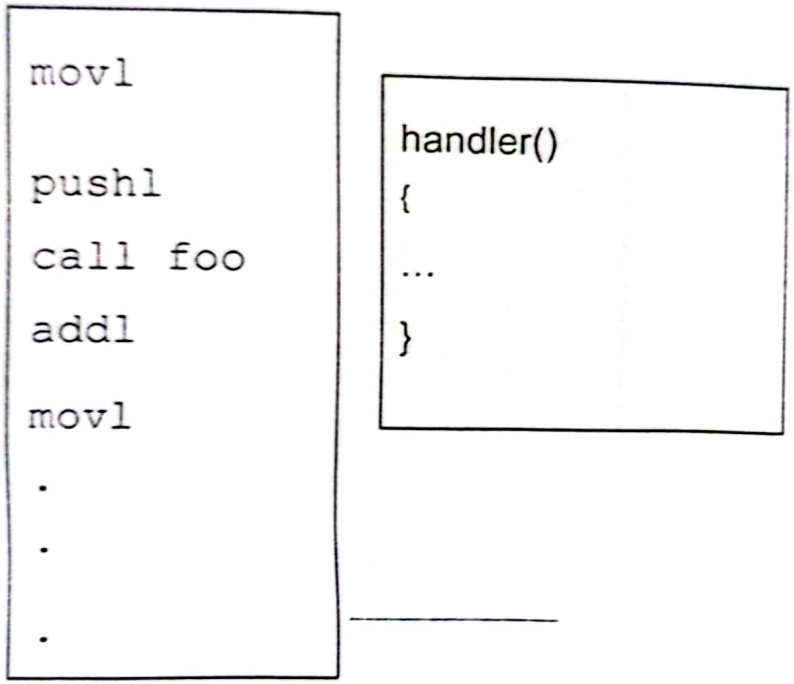
Avoiding race conditions

Signals

- Event notification sent to a process at *any* time

2. An event generates a signal
3. OS stops the process immediately
4. Signal handler executes and completes
5. The process resumes where it left off

Process



The kernel can generate signals to notify processes of events. For example SIGPIPE will be generated when a process writes to a pipe which has been closed by the reader; by default, this causes the process to terminate, which is convenient when constructing Shell pipelines.

Examples of Signals

- User types control-C

    1. 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

Three ways of sending signals exists: 1 from keyboard 2 from shell and 3 from the program.

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);
```

o 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`

```
 1) HUP  2) INT  3) QUIT  4) ILL  5) TRAP  6) ABRT  7) BUS  8) FPE
 9) KILL 10) USR1 11) SEGV 12) USR2 13) PIPE 14) ALRM 15) TERM 16) STKFLT
17) CHLD 18) CONT 19) STOP 20) TSTP 21) TTIN 22) TTOU 23) URG 24) XCPU
25) XFSZ 26) VTALRM 27) PROF 28) WINCH 29) POLL 30) PWR 31) SYS 34) RTMIN
35) RTMIN+1 36) RTMIN+2 37) RTMIN+3 38) RTMAX-3 39) RTMAX-2
40) RTMAX-1 41) 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).  */      9
```

Signal Handling

Signal handler is a programmer defined function which is automatically invoked when a signal is delivered.

- Signals have default handlers

  Usually, terminate the process and generate core image

2. Programs can over-ride default for most signals

   Define their own handlers

   Ignore certain signals, or temporarily block them

3. Two signals are not "catchable" (cannot be overwriten) 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

Signal handlers can be installed with the Signal (2) or Sigaction () system call. If a signal handler is not installed for a particular signal, the default handler is used. Otherwise the signal is intercepted and the signal handler is invoked. The process can also specify two default behaviours without creating a handler SIG_DFL and SIG_IGN.

\endash To install a handler, use

```
#include <signal.h>

typedef void (*sighandler_t)(int);

sighandler_t signal(int sig, sighandler_t handler);
```

\endash Handler will be invoked, when signal `sig` occurs

\endash Return the old handler on success; `SIG_ERR` on error

\endash 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

\endash Problem: the temporary file is not removed

- Process dies before `unlink(tmpfile)` is performed
- Can lead to lots of temporary files lying around

\endash Challenge in solving the problem

Control-C could happen at any time

Which line of code will be interrupted???

\endash Solution: signal handler

\endash Define a “clean-up” function to remove the file

o Install the function as a signal handler

Solution: Clean-Up Signal Handler

```
#include <stdio.h>

#include <signal.h>

char *tmpfile = "temp.xxx";

void cleanup() {

    unlink(tmpfile);

    exit(1);

}

int main() {
```

```
    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

\endash 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>

char *tmpfile = "temp.xxx";

void cleanup() {
   unlink(tmpfile);

   exit(1);
}

int main() {
   if (signal(SIGINT, cleanup) == SIG_ERR)
```

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

```
        fprintf(stderr, "Cannot set up signal\n");

    ...

    return(0);
}
```

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`

\endash 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>

char *tmpfile = "temp.xxx";

void cleanup() {

    unlink(tmpfile);

    exit(1);

}

int main() {
```

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

```
    if (signal(SIGINT, SIG_IGN) != SIG_IGN)
        signal(SIGINT, cleanup);
    ...
    return(0);
}
```

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

\endash Example: testing a global variable set by a handler

```
int myflag = 0;

void myhandler() { myflag = 1;
}
int main() {
  if (myflag == 0)
                              /* do something */
}
```

Problem: `myflag`

might become `1` just *after* the comparison!

Race conditions

1. In Operating Systems, threads are used to achieve concurrency.
2. Concurrency is the ability to perform multiple operations at the same time.
3. If multiple threads access shared data and are not synchronized with each other, we can face situations wherein the threads processing the shared data generate different results every time.

When writing multithreaded applications, one of the most common problems experienced is race conditions.

A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.

A race condition is a semantic error. It is a flaw that occurs in the timing or the ordering of events that leads to erroneous program behavior.

A race condition occurs when two threads access a shared variable at the same time. The first thread reads the variable, and the second thread reads the same value from the

variable. Then the first thread and second thread perform their operations on the value, and they race to see which thread can write the value last to the shared variable.

The value of the thread that writes its value last is preserved, because the thread is writing over the value that the previous thread wrote.

Problems often occur when one thread does a "check-then-act" (e.g. "check" if the value is X, then "act" to do something that depends on the value being X) and another thread does something to the value in between the "check" and the "act". E.g:

```
if (x == 5) // The "Check"
{
   y = x * 2; // The "Act"

   // If another thread changed x in between "if (x == 5)" and "y = x * 2" above,
   // y will not be equal to 10.
```

}

The point being, y could be 10, or it could be anything, depending on whether another thread changed x in between the check and act. You have no real way of knowing.

In order to prevent race conditions from occurring, you would typically put a lock around the shared data to ensure only one thread can access the data at a time. This would mean something like this:

```
// Obtain lock for x
if (x == 5)
{
   y = x * 2; // Now, nothing can change x until the lock is released.
              // Therefore y = 10
}
// release lock for x
```

The other thread will have to wait until the lock is released before it can proceed. This makes it very important that the

lock is released by the holding thread when it is finished with it. If it never releases it, then the other thread will wait indefinitely.

A "race condition" exists when multithreaded (or otherwise parallel) code that would access a shared resource could do so in such a way as to cause unexpected results.

Another example:

```
for ( int i = 0; i < 10000000; i++ )
{
  x = x + 1;
}
```

If you had 5 threads executing this code at once, the value of x WOULD NOT end up being 50,000,000. It would in fact vary with each run.

This is because, in order for each thread to increment the value of x, they have to do the following: (simplified, obviously)

Retrieve the value of x
Add 1 to this value
Store this value to x
Any thread can be at any step in this process at any time, and they can step on each other when a shared resource is involved. The state of x can be changed by another thread during the time between x is being read and when it is written back.

Let's say a thread retrieves the value of x, but hasn't stored it yet. Another thread can also retrieve the **same** value of x (because no thread has changed it yet) and then they would both be storing the **same** value (x+1) back in x!
Example:

Thread 1: reads x, value is 7

Thread 1: add 1 to x, value is now 8
Thread 2: reads x, **value is 7**
Thread 1: stores 8 in x
Thread 2: adds 1 to x, value is now 8
Thread 2: **stores 8 in x**

Race conditions can be avoided by employing some sort of **locking** mechanism before the code that accesses the shared resource:

```
for ( int i = 0; i < 10000000; i++ )
{
   //lock x
   x = x + 1;
   //unlock x
}
```

Here, the answer comes out as 50,000,000 every time.

How do you handle them?

Race condition can be handled by **Mutex** or **Semaphores**. They act as a lock allows a process to acquire a resource based on certain requirements to prevent race condition.

How do you prevent them from occurring?

There are various ways to prevent race condition, such as **Critical Section Avoidance**.

1. No two processes simultaneously inside their critical regions. (**Mutual Exclusion)**
2. No assumptions are made about speeds or the number of CPUs.
3. No process running outside its critical region which blocks other processes.
4. No process has to wait forever to enter its critical region. (A waits for B resources, B waits for C resources, C waits for A resources)

5. Problem with race condition can be solved by adding an 'assurance' that no other process can access the shared resource while a process is using it (read or write). The period of time for the assurance is called the 'critical section'.

6. This assurance can be provided by creating a lock. E.g. If a process need to use a shared resource, it can lock the resource and release it when it is done, as the steps shown below. The lock may use the mechanism called Semaphore or Mutex. Meanwhile other process that need to use the shared resource will do the same steps.

```
wait until Mutex is unlocked
 set Mutex=lock
 begin read/write value in shared resource
  ..... do something
finish read/write value in shared resource
set Mutex=unlock
```

This way a process A can ensure no other process will update the shared resource while A is using the resource. The same issue will apply for thread.

Predefined Signals

List of the predefined signals:

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1
36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4  39) SIGRTMIN+5
40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8  43) SIGRTMIN+9
44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13
52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9
56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6  59) SIGRTMAX-5
60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2  63) SIGRTMAX-1
64) SIGRTMAX
```

Applications can define their own signals

- An application can define signals with unused values

Summary

Signals

– A **signal** is an asynchronous event mechanism

– C90 `raise()` or POSIX `kill()` sends a signal

– C90 `signal()` or POSIX `sigaction()` installs a signal handler

- Most predefined signals are "catchable"

– Beware of race conditions

– Signals of type x automatically are blocked while handler for type x signals is running

– POSIX `sigprocmask()` blocks signals in any critical section of code.