Signals & Shared Memory

Dr. Yingwu Zhu

Topics

- Signals
- Shared memory for IPC

Question

- Q: How does the OS communicate to a process?
- A: Signals

Signals

- What if something unexpected or unpredictable happens?
  - a floating-point error
  - a power failure
  - an alarm clock "ring"
  - the death of a child process
  - a termination request from a user (i.e., a Control-C)
  - a suspend request from a user (i.e., a Control-Z)

Signals

- These kind of events are often called *interrupts*
  - i.e., they interrupt the normal flow of the program to service an interrupt handler
- When Linux recognizes such event it sends corresponding *signal,*
  - e.g., floating point error: kernel sends offending process signal number 8

Who can send signals?

- The kernel
- Any process can send any other process a signal as long as it has permission
  - receiving process suspends its current flow of control
  - executes signal handler
  - resumes original flow when signal handler finishes

Two Signal Types

- Standard signal (traditional unix signals)
  - delivered to a process by setting a bit in a bitmap
  - one for each signal
  - thus there cannot be multiple instances of the same signal; bit can be one (signal) or zero (no-signal)
- real-time signals (or queued signals)
  - defined by POSIX 1003.1b where successive instances of the same signal are significant and need to be properly delivered.
  - In order to use queued signals, you must use the sigaction () system call, rather than signal ()

Defined Signals

- Where are signals defined?
  - Signals are defined in /usr/include/signal.h
  - other platform-specific header files
    - e.g., /usr/include/asm/signal.h
- Programmer may chose that
  - particular signal triggers a user-defined signal handler
  - triggers the default kernel-supplied handler
  - signal is ignored

Default Signal Handler

- Terminates the process and generates a dump of memory in a core file (core)
- Terminates the process without generating a core image file (quit)
- Ignores and discards the signal (ignore)
- Suspends the process (stop)
- Resumes the process

Macro	#	Default action	Description
SIGHUP	1	quit	Hangup or death of controlling process.
SIGINT	2	quit	Keyboard interrupt.
SIGQUIT	3	core	Quit.
SIGILL	4	core	Illegal instruction.
SIGABRT	6	core	Abort.
SIGFPE	8	core	Arithmetic exception.
SIGKILL	9	quit	Kill (cannot be caught, blocked, or ignored).
SIGUSR1	10	quit	User-defined signal.
SIGSEGV	11	core	Segmentation violation (out of range address).
SIGUSR2	12	quit	User-defined signal.
SIGPIPE	13	quit	Write on a pipe or other socket with no one to read it.
SIGALRM	14	quit	Alarm clock.
SIGTERM	15	quit	Software termination signal (default signal sent by *kill*).
SIGCHLD	17	ignore	Status of child process has changed.
SIGCONT	18	none	Continue if stopped.
SIGSTOP	19	stop	Stop (suspend) the process.
SIGTSTP	20	stop	Stop from the keyboard.
SIGTTIN	21	stop	Background read from tty device.
SIGTTOU	22	stop	Background write to tty device.

Terminal Signals

- Easiest way to send signal to foreground process
  - press *Control-C or Control-Z*
  - when terminal driver recognizes a Control-C it sends SIGINT signal to all of the processes in the current foreground job
  - Control-Z causes SIGSTP to be sent
  - by default
    - SIGINT terminates a process
    - SIGTSTP suspends a process

Alarm Signal

- *System Call: unsigned int alarm (unsigned int count)*
- alarm () instructs the kernel to send the SIGALRM signal to the calling process after count seconds. If an alarm had already been scheduled, it is overwritten. If count is 0, any pending alarm requests are cancelled.
- alarm () returns the number of seconds that remain until the alarm signal is sent.
- The default handler for this signal displays the message "Alarm clock" and terminates the process

Example

```
$ cat alarm.c                    ...list the program.
#include <stdio.h>
main ()
{
 alarm (3); /* Schedule an alarm signal in three seconds */
 printf ("Looping forever...\n");
 while (1);
 printf ("This line should never be executed\n");
}
```

```
$ ./alarm                    ...run the program.
Looping forever...
Alarm clock                  ...occurs three seconds later.
$ _
```

Handling Signals

- How do you override the default action in the previous example?
  - the signal() system call may be used
- System Call:
  - **typedef void (*sighandler_t)(int);**
  - **sighandler_t signal(int** *signum*, **sighandler_t** *handler***);**

  - signal () allows a process to specify the action that it will take when a particular signal is received.
  - The parameter signum specifies the number of the signal that is to be reprogrammed

Handling Signals

- func may be one of several values:
  - SIG_IGN indicates that the specified signal should be ignored and discarded.
  - SIG_DFL indicates that the kernel's default handler should be used.
  - an address of a user-defined function, which indicates that the function should be executed when the specified signal arrives.

Handling Signals

- signals SIGKILL and SIGSTP may not be reprogrammed.
- a child process inherits signal settings from its parent during fork (). When process performs exec (), previously ignored signals remain ignored but installed handlers are set back to the default handler.
- with the exception of SIGCHLD, signals are not stacked, e.g., if a process is sleeping and three identical signals are sent to it, only one of the signals is actually processed.
- signal () returns the previous func value associated with *signum if successful; otherwise it returns -1.*

What is the problem?

```
$ cat alarm.c                    ...list the program.
#include <stdio.h>
main ()
{
 alarm (3); /* Schedule an alarm signal in three seconds */
 printf ("Looping forever...\n");
 while (1);
 printf ("This line should never be executed\n");
}
```

System Call: int **pause (void)**

- pause () suspends the calling process and returns when the calling process receives a signal.
- It is most often used to wait efficiently for an alarm signal.
- pause () doesn't return anything useful.
- to enhance efficiency the previous program is modified to wait for a signal.
  - also a custom signal handler is installed

```
$ cat handler.c                          ...list the program.
#include <stdio.h>
#include <signal.h>
int alarmFlag = 0; /* Global alarm flag */
void alarmHandler (); /* Forward declaration of alarm handler */
/****************************************************************/
main ()
{
 signal (SIGALRM, alarmHandler); /* Install signal handler */
 alarm (3); /* Schedule an alarm signal in three seconds */
 printf ("Looping...\n");
 while (!alarmFlag) /* Loop until flag set */
   {
     pause (); /* Wait for a signal */
   }
 printf ("Loop ends due to alarm signal\n");
}
/****************************************************************/
void alarmHandler ()
{
 printf ("An alarm clock signal was received\n");
 alarmFlag = 1;
}
$ ./handler                             ...run the program.
Looping...
An alarm clock signal was received  ...occurs three seconds later.
Loop ends due to alarm signal
```

Handling ctr-c

- Sometimes we want to protect critical pieces of code against Control-C attacks and other such signals
  - save previous value of the handler so that it can be restored after the critical code has executed
  - in the following example SIGINT is “disabled”

```
$ cat critical.c                              ...list the program.
#include <stdio.h>
#include <signal.h>
main ()
{
 void (*oldHandler) (); /* To hold old handler value */
  printf ("I can be Control-C'ed\n");
 sleep (3);
 oldHandler = signal (SIGINT, SIG_IGN); /* Ignore Control-C */
 printf ("I'm protected from Control-C now\n");
 sleep (3);
 signal (SIGINT, oldHandler); /* Restore old handler */
 printf ("I can be Control-C'ed again\n");
 sleep (3);
 printf ("Bye!\n");
}
$ ./critical                           ...run the program.
I can be Control-C'ed
^C                                     ...Control-C works here.
$ ./critical                           ...run the program again.
I can be Control-C'ed
I'm protected from Control-C now
^C                                     ...Control-C is ignored.
I can be Control-C'ed again
Bye!
```

Send Signals to other processes

- Process may send signal to other process by using kill()
  - often misunderstood as "killing another process", but not all kill signals do that
- System Call: int **kill (pid_t pid, int sigCode)**
  - sends the signal with value sigCode to the process with PID pid. kill () succeeds and the signal is sent as long as at least one of the following conditions is satisfied:
    - The sending process and the receiving process have the same owner.
    - The sending process is owned by a super-user.

Kill()

- There are a few variations on the way that kill () works:
  - If pid is 0, the signal is sent to *all of the processes in the* sender's process group.
  - If pid is -1 and the sender is owned by a super-user, the signal is sent to all processes, including the sender.
  - If pid is -1 and the sender is not a super-user, the signal is sent to all of the processes owned by the same owner as the sender, excluding the sending process.
  - If the pid is negative and not -1, the signal is sent to all of the processes in the process group.
  - If kill () manages to send at least one signal successfully, it returns 0; otherwise, it returns -1.

SIGCHLD

- When child terminates
  - child process sends SIGCHLD to parent
  - parent often installs a handler to deal with this signal
  - parent typically executes a wait() to accept the child's termination code (such child is not zombie anymore)
    - Alternatively, the parent can choose to ignore SIGCHLD signals, in which case the child de-zombifies automatically.

```
#include <stdio.h>
#include <signal.h>
int delay;
void childHandler ();
/*********************************************************************/
main (argc, argv)
int argc;
char* argv[];
{
 int pid;
 signal (SIGCHLD, childHandler); /* Install death-of-child handler */
 pid = fork (); /* Duplicate */
 if (pid == 0) /* Child */
   {
     execvp (argv[2], &argv[2]); /* Execute command */
     perror ("limit"); /* Should never execute */
   }
 else /* Parent */
   {
     sscanf (argv[1], "%d", &delay); /* Read delay from command line */
     sleep (delay); /* Sleep for the specified number of seconds */
     printf ("Child %d exceeded limit and is being killed\n", pid);
     kill (pid, SIGINT); /* Kill the child */
   }
}
/*********************************************************************/
void childHandler () /* Executed if the child dies before the parent */
{
 int childPid, childStatus;
 childPid = wait (&childStatus); /* Accept child's termination code */
 printf ("Child %d terminated within %d seconds\n", childPid, delay);
 exit (/* EXITSUCCESS */ 0);
}
```

Suspending/resuming a process

- The SIGSTOP and SIGCONT signals suspend and resume a process, respectively.
- They are used by the Linux shells to support job control to implement built-in commands like *stop, fg, and bg.*
- following example:
  - create two children
  - suspend and resume one child
  - terminate both children

```c
#include <signal.h>
#include <stdio.h>
main ()
{
 int pid1;
 int pid2;
  pid1 = fork();
 if (pid1 == 0) /* First child */
    {
      while (1) /* Infinite loop */
        {
          printf ("pid1 is alive\n");
          sleep (1);
        }
    }
 pid2 = fork (); /* Second child */
 if (pid2 == 0)
    {
      while (1) /* Infinite loop */
        {
          printf ("pid2 is alive\n");
          sleep (1);
        }
    }
 sleep (3);
 kill (pid1, SIGSTOP); /* Suspend first child */
 sleep (3);
 kill (pid1, SIGCONT); /* Resume first child */
 sleep (3);
 kill (pid1, SIGINT); /* Kill first child */
 kill (pid2, SIGINT); /* Kill second child */
}
```

Process Group

- What happens when you Control-C a program that created several children?
  - typically the program and its children terminate
  - why the children?

Process Group

- In addition to having unique ID, process also belongs to a *process group*
  - Several processes can be members of the same process group.
  - When a process forks, the child inherits its process group from its parent.
  - A process may change its process group to a new value by using setpgid ().
  - When a process execs, its process group remains the same.

Shared Memory

Shared Memory

- Simplest, fastest, but local communication
- Allow two or more processes to access the same memory (as if they called malloc and were returned pointer to the same memory address)
  - Let multiple processes attach a segment of physical memory to their virtual address spaces
- One changes the memory, then all others see the change!

Caution on Shared Memory

- Sometimes, you may need to synchronize accesses to the shared memory
  - Synchronization issue!

Overview

- Creation: shmget()
- Access control: shmctl()
- Attached to addr. space: shmat()
- Detached from addr. space : shmdt()
- #include <sys/types.h>
- #include <sys/ipc.h>
- #include <sys/shm.h>

Notes

- A process creates a shared memory segment using shmget().
  - The original owner of a shared memory segment can assign ownership to another user with shmctl(). It can also revoke this assignment.
- Other processes with proper permission can perform various control functions on the shared memory segment using shmctl().
- Once created, a shared segment can be attached to a process address space using shmat(). It can be detached using shmdt().
  - The attaching process must have the appropriate permissions for shmat(). Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation.
  - A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory. The identifier of the segment is called the shmid. The structure definition for the shared memory segment control structures and prototypes can be found in <sys/shm.h>.

Create a Share Memory Segment

- int shmget(key_t key, size_t size, int shmflg)
  - key: integer ID
    - processes can access the same seg. by using the same key; IPC_PRIVATE as the key guarantee a new seg. is created
  - size: number of bytes, actual bytes rounded up to multiple of page size
  - shmflg: access permissions flags and creation control flags
    - specify IPC_CREAT, if a segment for the key does not exist, it is created
    - If you specify IPC_CREAT | IPC_EXCL, the key must not exist, otherwise fails. If the key exists and the ICP_EXCL is not given, the existing seg. is returned!
    - Permission flag to indicate permissions granted to owner, group and world, see <sys/stat.h> for constants, or simply bits
    - #define PERM_UREAD 0400 #define PERM_UWRITE 0200
      #define PERM_GREAD 0040 #define PERM_GWRITE 0020
      #define PERM_OREAD 0004 #define PERM_OWRITE 0002
  - return: shared memory segment ID on success
  - int segment_id = shmget (shm_key, getpagesize (),IPC_CREAT | S_IRUSR | S_IWUSER);
  - int segment_id = shmget (shm_key, 1000,IPC_CREAT | 0666);

Attach a Shared Memory Segment

- Must attach it before using!
- void *shmat (int shmid, const void*shmaddr, int shmflg);
  - returns a pointer to the head of the shared segment associated with a valid shmid
  - #2 para: specifies where in your process's address space you want to map the shared memory; if you specify NULL, Linux will choose an available address
  - #3 para:
    - SHM_RND indicates that the address specified for the second parameter should be rounded down to a multiple of the page size. If you don't specify this flag, you must page-align the second argument to shmat yourself.
    - SHM_RDONLY indicates that the segment will be only read, not written.

Detaching a Share Mem. Seg.

- Never forget to detach it!
- int shmdt(const void* shmaddr);
  - Detaches the shared memory segment located at the address indicated by shmaddr

Controlling a Shared Mem. Seg.

- int shmctl(int shmid, int cmd, struct shmid_ds *buf);
- cmd
  - **SHM_LOCK** -- Lock the specified shared memory segment in memory. The process must have the effective ID of superuser to perform this command.
  - **SHM_UNLOCK** -- Unlock the shared memory segment. The process must have the effective ID of superuser to perform this command.
  - **IPC_STAT** -- Return the status information contained in the control structure and place it in the buffer pointed to by buf. The process must have read permission on the segment to perform this command.
  - **IPC_SET** -- Set the effective user and group identification and access permissions. The process must have an effective ID of owner, creator or superuser to perform this command.
  - **IPC_RMID** -- Remove the shared memory segment.
- The buf is a structure of type struct shmid_ds which is defined in <sys/shm.h>

Deleting Segment

- When you detach from the segment, it isn't destroyed. Nor is it removed when *everyone* detaches from it. You have to specifically destroy it using a call to shmctl(), similar to the control calls for the other System V IPC functions:
  - shmctl(shmid, IPC_RMID, NULL);

Examples: server.c

```
main() {
    char c;
    int shmid;
    key_t key;
    char *shm, *s; /* * We'll name our shared memory segment * "5678". */
    key = 5678;
    /* * Create the segment. */
    if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
      perror("shmget");
      exit(1); }
    /* * Now we attach the segment to our data space. */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
      perror("shmat"); exit(1); }
    /* * Now put some things into the memory for the * other process to read. */
    s = (char*)shm;
    for (c = 'a'; c <= 'z'; c++)
      *s++ = c;
    *s = NULL;
    /* * Finally, we wait until the other process * changes the first character of our
    memory * to '*', indicating that it has read what * we put there. */
    while (*shm != '*')
      sleep(1);
    exit(0);
}
```

client.c

```
main() {
    int shmid;
    key_t key;
    char *shm, *s;
    /* * We need to get the segment named * "5678", created by the server. */
    key = 5678;
    /* * Locate the segment. */
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget"); exit(1); }
    /* * Now we attach the segment to our data space. */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat"); exit(1); }
    /* * Now read what the server put in the memory. */
    for (s = shm; *s != NULL; s++)
        putchar(*s);
    putchar('\n');
    /* * Finally, change the first character of the * segment to '*', indicating we have
    read * the segment. */
    *shm = '*';
    exit(0);
}
```

server.c: does this have problem?

```
main() {
    char c;
    int shmid;
    key_t key;
    char *shm, *s; /* * We'll name our shared memory segment * "5678". */
    key = 5678;
    /* * Create the segment. */
    if (shmid = shmget(key, SHMSZ, IPC_CREAT | 0666) < 0) {
       perror("shmget");
       exit(1); }
    /* * Now we attach the segment to our data space. */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
       perror("shmat"); exit(1); }
    /* * Now put some things into the memory for the * other process to read. */
    s = (char*)shm;
    for (c = 'a'; c <= 'z'; c++)
      *s++ = c;
    *s = NULL;
    /* * Finally, we wait until the other process * changes the first character of our
    memory * to '*', indicating that it has read what * we put there. */
    while (*shm != '*')
       sleep(1);
    exit(0);
}
```

Fork and shmat

- After fork(), the child inherits the attached shared memory segments

Debugging

- ipcs –m
  - Get info about shared memory

------ Shared Memory Segments --------

key	shmid	owner	perms	bytes	nattch	status
0x00000000	1627649	user	640	25600	0	

- ipcrm shm 1627649
  - Remove the shared memory was erroneously left behind by a program