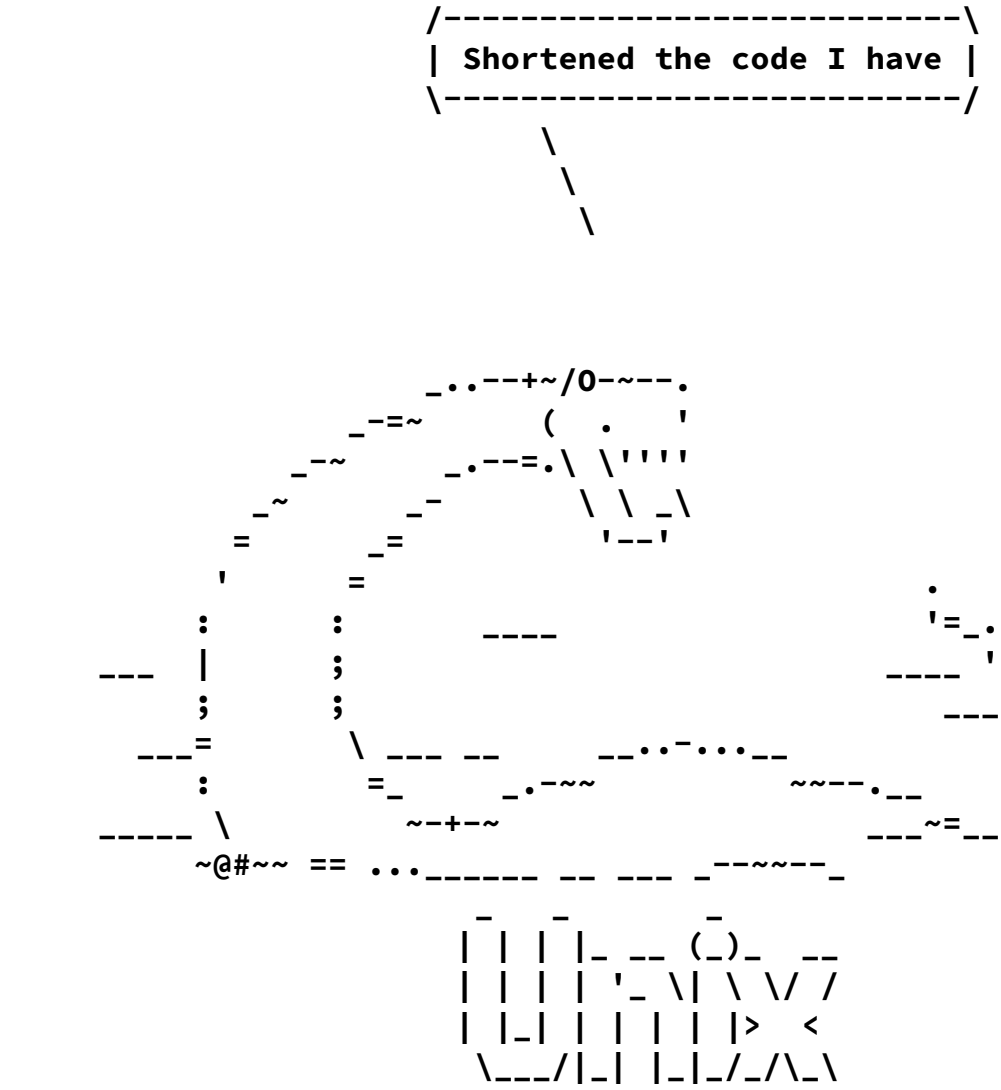


SOHAN JAIN

UNIX PROGRAMMING [18CS56]

SOHAN JAIN

February 17, 2022

Part I

Module-4

1. What is process accounting? Write a program to illustrate the generating of accounting data.

Most Unix systems provide an option to do process accounting. When enabled the kernel writes an accounting record each time a process terminates.

Program to generate accounting data.

```c
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

int
main(void)
{
  pid_t pid;

  if ((pid = fork()) < 0)
    perror("fork error");
  else if (pid != 0) {
    sleep(2);
    exit(2);
  }

  if ((pid = fork()) < 0)
    perror("fork error");
  else if (pid != 0) {
    sleep(4);
    abort();
  }

  if ((pid = fork()) < 0)
    perror("fork error");
  else if (pid != 0) {
    execl("/usr/bin/dd", "dd", "if=/boot", "of=/dev/null", NULL);
    exit(7);
  }

  if ((pid = fork()) < 0)
    perror("fork error");
  else if (pid != 0) {
    sleep(8);
    exit(0);
  }

  sleep(6);
  kill(getpid(), SIGKILL);
  exit(6);
}
```

2. Explain "system" function with its prototype.

`system` Function is used to execute a command string from within a program.

SYNOPSIS:

```c
#include <stdlib.h>

int system(const char* command);
```

Since `system` is implemented by calling `fork`, `exec`, and `waitpid`, there are different types of return values.

- If command is NULL, then a nonzero value if a shell is available, or 0 if no shell is available.
- If a child process could not be created, or its status could not be retrieved, the return value is -1 and errno[1] is set to indicate the error.
- If the `exec` fails the return value is as if the shell had executed `exit (127)`.
- If all system calls succeed, then the return value is the termination status of the child shell used to execute command.

The `system()` library function uses `fork` to create a child process that executes the shell command specified in command using `execl` as follows:

```c
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <unistd.h>

int
system(const char* command)
{
  pid_t pid;
  int status;

  if (command == NULL)
    return 1;

  if ((pid = fork()) < 0) {
    status = -1;
  } else if (pid == 0) {
    execl("/bin/sh", "sh", "-c", command, (char *) NULL);
    _exit(127);
  } else {
    while (waitpid(pid, &status, 0) < 0)
      if (errno != EINTR) {
        status = -1;
        break;
      }
  }

  return status;
}
```

Note: we call `_exit` instead of `exit`. This is to prevent any standard I/O buffers from being flushed in the child.

3. Explain popen and pclose function with prototypes and demonstrate its usage with a simple C program.

These two functions handles:

- creation of a pipe
- the fork of a child

[1] number of last error

- closing the unused ends of the pipe
- execing a shell to execute the command
- waiting for the command to terminate

SYNOPSIS:

```
#include <stdio.h>

FILE *popen(const char* command, const char* type);
```

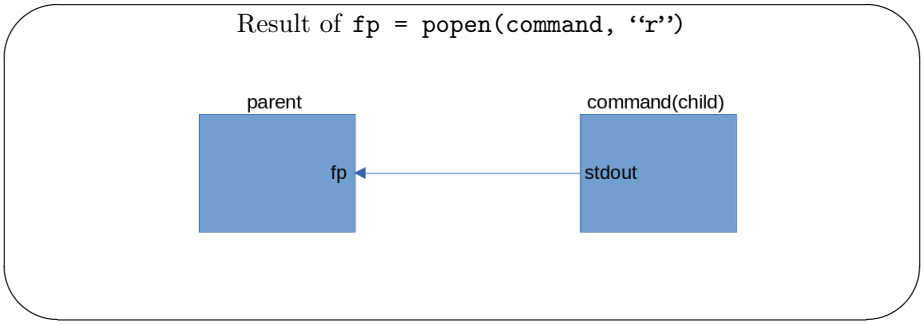
Returns: file pointer if OK, NULL on error

```
int pclose(FILE *fp);
```

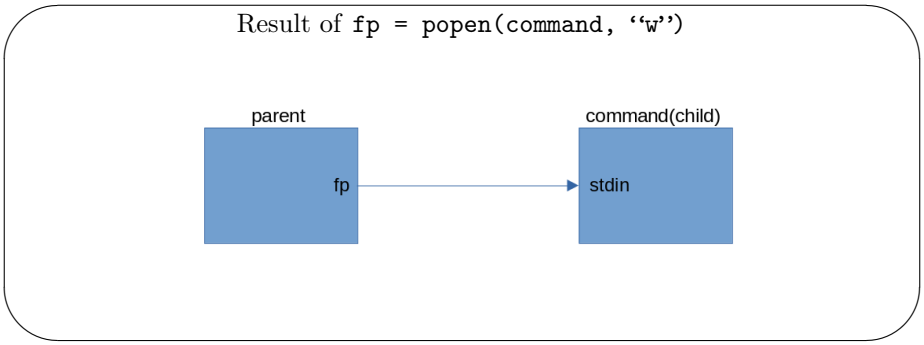
Returns: termination status of command, or -1 on error

The function popen does a fork and exec to execute the command, and returns a standard I/O file pointer.

If type "r", the file pointer is connected to the standard output of command.



If type is "w", the file pointer is connected to the standard input of command.



The `pclose` function closes the standard I/O stream, waits for the command to terminate, and returns the termination status of the shell. If the shell cannot

be executed, the termination status returned by `pclose` is as if the shell had executed `exit (127)`.

Program: Copy file to pager program using popen

```c
#include <sys/types.h>
#include <sys/wait.h>
#include <errno.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXLINE 1024
#define PAGER "${PAGER:-more}"

int
main(int argc, char* argv[])
{
  char line[MAXLINE];
  FILE *fpin, *fpout;

  if (argc != 2)
    fprintf(stderr, "usage: a.out <pathname>\n");

  if ((fpin = fopen(argv[1], "r")) == NULL)
    fprintf(stderr, "can't open %s\n", argv[1]);

  if ((fpout = popen(PAGER, "w")) == NULL)
    perror("popen error");

  while (fgets(line, MAXLINE, fpin) != NULL) {
    if (fputs(line, fpout) == EOF)
      perror("fputs error to pipe");
  }
  if (ferror(fpin))
    perror("fgets error");
  if (pclose(fpout) == -1)
    perror("pclose error");

  exit(0);
}
```

4. Write a simple C program to illustrate the concept of a co-process.

□

5. Discuss with an example, the client-server communication using FIFO.

Part II

Module-5

1. What are signals? List any four signals along with brief explanation. Write a program to setup signals handler for SIGALRM and SIGINT signals.

Signals are software interrupts.Signals provide a way of handling asynchronous events.

List of signals:

- SIGALRM - This signal is generated when a timer set with the alarm function expires. This signal is also generated when an interval timer set by the setitimer(2) function expires.
- SIGINT - This signal is generated by the terminal driver when we press the interrupt key. This signal is sent to all processes in the foreground process group.
- SIGKILL - The SIGKILL signal is sent to a process to cause it to terminate immediately.
- SIGPIPE - The SIGPIPE signal is sent to a process when it attempts to write to a pipe without a process connected to the other end.

Program to handle SIGALRM and SIGINT

```c
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>

static void sigint_handler(int);

int
main()
{
  alarm(4);
  struct sigaction sigact;
  sigact.sa_handler = &sigint_handler;

  sigaction(SIGINT, &sigact, NULL);
  sigaction(SIGALRM, &sigact, NULL);

  pause();
}

static void
sigint_handler(int signo)
{
  if (signo == SIGINT) {
    printf("received SIGINT\n");
    exit(0);
  } else if (signo == SIGALRM) {
    printf("received SIGALRM\n");
  } else {
    fprintf(stderr, "received signal %d\n", signo);
    exit(1);
  }}
```

2. What are signals? Write a program to setup signal handler for the SIGINT signal using sigaction API.

Signals are software interrupts.Signals provide a way of handling asynchronous events.

signal handler for SIGINT using `sigaction`[2]

```c
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>

static void sigint_handler(int);

int
main()
{
  struct sigaction sigact;
  sigact.sa_handler = &sigint_handler;

  sigaction(SIGINT, &sigact, NULL);
  pause();
}

static void
sigint_handler(int signo)
{
  if (signo == SIGINT) {
    printf("received SIGINT\n");
    exit(0);
  } else {
    fprintf(stderr, "received signal %d\n", signo);
    exit(1);
  }
}
```

3. What is signal? Discuss any five POSIX-defined signals. Explain how to setup a signal handler.

Signals are software interrupts.Signals provide a way of handling asynchronous events.

POSIX signals:

- SIGABRT - The SIGABRT signal is sent to a process to tell it to abort, i.e. to terminate.
- SIGINT - This signal is generated by the terminal driver when we press the interrupt key. This signal is sent to all processes in the foreground process group.
- SIGILL - This signal indicates that the process has executed an illegal hardware instruction.

[2]The sigaction() function allows the calling process to examine and/or specify the action to be associated with a specific signal. The argument sig specifies the signal; acceptable values are defined in <signal.h>.

- SIGPIPE - The SIGPIPE signal is sent to a process when it attempts to write to a pipe without a process connected to the other end.
- SIGKILL - The SIGKILL signal is sent to a process to cause it to terminate immediately.

Setup a Signal handler:

If a process receives a signal, the process has a choice of action for that kind of signal. The process can ignore the signal, can specify a handler function, or accept the default action for that kind of signal.

We can handle the signal using `signal` or `sigaction` function.

Here we used `sigaction` to setup a signal handler for SIGINT signal.

```c
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>

static void sigint_handler(int);

int
main()
{
  struct sigaction sigact;
  sigact.sa_handler = &sigint_handler;

  sigaction(SIGINT, &sigact, NULL);
  pause();
}

static void
sigint_handler(int signo)
{
  if (signo == SIGINT) {
    printf("received SIGINT\n");
    exit(0);
  } else {
    fprintf(stderr, "received signal %d\n", signo);
    exit(1);
  }
}
```

4. What is the use of setjmp and longjmp functions with examples?

In C, we can't goto a label that's in another function. Instead, we must use the setjmp and longjmp functions to perform this type of branching. As we'll see, these two functions are useful for handling error conditions that occur in a deeply nested function call.

Syntax is NOT NECESSARY

```c
#include <setjmp.h>

int setjmp(jmp_buf env);
```

Returns: 0 if called directly, nonzero if returning from a call to `longjmp`

```c
void longjmp(jmp_buf env, int val);
```

Example of `setjmp` and `longjmp`

```c
#include <stdio.h>
#include <setjmp.h>

jmp_buf buf;

int
main() {
   int x = 1;
   //////////
   setjmp(buf);
   //////////
   printf("5");
   x++;
   if (x <= 10)
      longjmp(buf, 1);

   printf("\nAll finish\n");
   return 0;
}
```

Output:

```
$ ./setjmp
5555555555
All finish
```

5. Write the timeline or program sequence of execution for sigsetjmp and siglongjmp handling.

[BETTER NOT TO ATTEND THIS QUE]

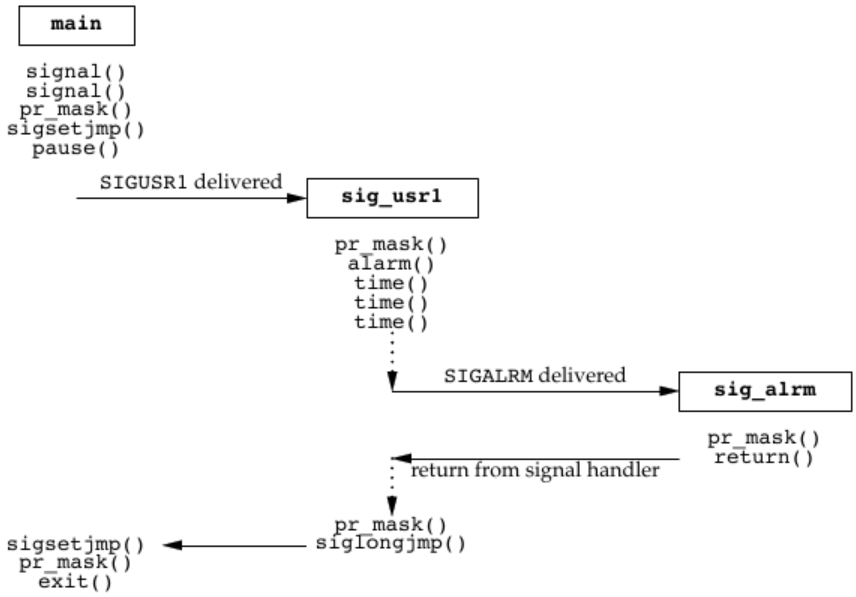


Figure 10.21 Timeline for example program handling two signals

6. Explain the following APIs along with their prototypes with respect to signals: kill

The kill function sends a signal to a process or a group of processes.

SYNOPSIS:

```
#include <signal.h>

int kill(pid_t pid, int signo);
```

Returns: 0 if OK, -1 on error

There are four different conditions for the pid argument to kill.

- pid $>$ 0 - The signal is sent to the process whose process ID is pid.
- pid $==$ 0 - The signal is sent to all processes whose process group ID equals the process group ID of the sender and for which the sender has permission to send the signal.

- pid < 0 - The signal is sent to all processes whose process group ID equals the absolute value of pid and for which the sender has permission to send the signal.
- pid $== -1$ - The signal is sent to all processes on the system for which the sender has permission to send the signal.

If the `signo` argument is 0, then the normal error checking is performed by kill, but no signal is sent. If we send the process the null signal and it doesn't exist, kill returns -1 and errno is set to ESRCH.

7. Write the prototype of ALARM and PAUSE function and explain how they operate.

The *alarm* function allows us to set a timer that will expire at a specified time in the future. When the timer expir

es, the SIGALRM signal is generated. If we ignore or don't catch this signal, its default action is to terminate the process.

```
#include <unistd.h>

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

Returns: 0 or number of seconds until previously set alarm

The seconds value is the number of clock seconds in the future when the signal should be generated.

The pause function suspends the calling process until a signal is caught.

```
#include <unistd.h>

int pause(void);
```

Returns: -1 with errno set to EINTR

The only time pause returns is if a signal handler is executed and that handler returns. In that case, pause returns -1 with errno set to EINTR.

Example program to illustrate the working of `alarm()` and `pause()`.

```c
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>

static void sigint_handler(int);

int
main()
{
  alarm(4);
  struct sigaction sigact;
  sigact.sa_handler = &sigint_handler;

  sigaction(SIGINT, &sigact, NULL);
  sigaction(SIGALRM, &sigact, NULL);

  pause();
}

static void
sigint_handler(int signo)
{
  if (signo == SIGINT) {
    printf("received SIGINT\n");
    exit(0);
  } else if (signo == SIGALRM) {
    printf("received SIGALRM\n");
  } else {
    fprintf(stderr, "received signal %d\n", signo);
    exit(1);
  }}
```

8. Explain the following APIs along with their prototypes w.r.t. Signals: alarm. [SAME AS 7]

9. What are daemon process? Explain the BSD facility adapted by daemon processes for error handling.

Daemons are processes that live for a long time. They are often started when the system is bootstrapped and terminate only when the system is shut down.

[Refer Que 12]

10. Define daemon process. Discuss the basic coding rules of daemon process.

Daemons are processes that live for a long time. They are often started when the system is bootstrapped and terminate only when the system is shut down.

Coding Rules:

1. Call umask to set the file mode creation mask to a known value, usually 0. The inherited file mode creation mask could be set to deny certain permissions. If the daemon process creates files, it may want to set specific permissions. For example, if it creates files with group-read and group-write enabled, a file mode creation mask that turns off either of these permissions would undo its efforts. On the other hand, if the daemon

calls library functions that result in files being created, then it might make sense to set the file mode create mask to a more restrictive value (such as 007), since the library functions might not allow the caller to specify the permissions through an explicit argument.

2. Call fork and have the parent exit. This does several things. First, if the daemon was started as a simple shell command, having the parent terminate makes the shell think that the command is done. Second, the child inherits the process group ID of the parent but gets a new process ID, so we're guaranteed that the child is not a process group leader. This is a prerequisite for the call to setsid that is done next.

3. Call setsid to create a new session. The three steps listed in Section 9.5 occur. The process (a) becomes the leader of a new session, (b) becomes the leader of a new process group, and (c) is disassociated from its controlling terminal.

4. Change the current working directory to the root directory. The current working directory inherited from the parent could be on a mounted file system. Since daemons normally exist until the system is rebooted, if the daemon stays on a mounted file system, that file system cannot be unmounted.

5. Unneeded file descriptors should be closed. This prevents the daemon from holding open any descriptors that it may have inherited from its parent (which could be a shell or some other process). We can use our open_max function (Figure 2.17) or the getrlimit function (Section 7.11) to determine the highest descriptor and close all descriptors up to that value.

6. Some daemons open file descriptors 0, 1, and 2 to /dev/null so that any library routines that try to read from standard input or write to standard output or standard error will have no effect. Since the daemon is not associated with a terminal device, there is nowhere for output to be displayed, nor is there anywhere to receive input from an interactive user. Even if the daemon was started from an interactive session, the daemon runs in the background, and the login session can terminate without affecting the daemon. If other users log in on the same terminal device, we wouldn't want output from the daemon showing up on the terminal, and the users wouldn't expect their input to be read by the daemon.

12. Explain error handling for daemon process with a neat block diagram. Write the system library functions associated with error logging.

The BSD syslog facility has been widely used since 4.2BSD. Most daemons use this facility. Figure 13.2 illustrates its structure.

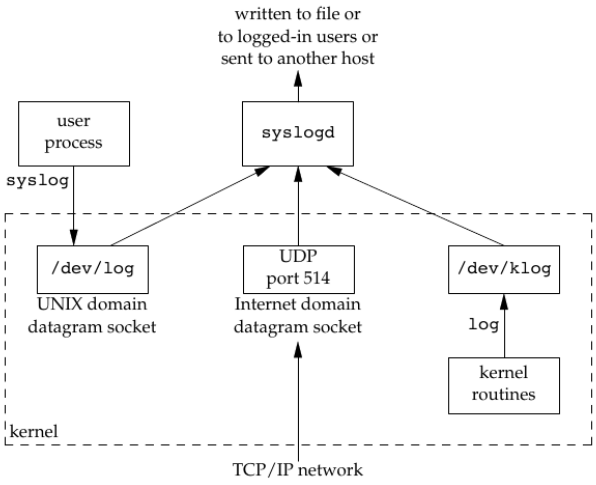


Figure 13.2 The BSD `syslog` facility

There are three ways to generate log messages:

1. Kernel routines can call the log function. These messages can be read by any user process that opens and reads the /dev/klog device. We won't describe this function any further, since we're not interested in writing kernel routines.

2. Most user processes (daemons) call the syslog(3) function to generate log messages. We describe its calling sequence later. This causes the message to be sent to the UNIX domain datagram socket /dev/log.

3. A user process on this host, or on some other host that is connected to this host by a TCP/IP network, can send log messages to UDP port 514. Note that the syslog function never generates these UDP datagrams: they require explicit network programming by the process generating the log message.

system library functions associated with error logging:

```
#include <syslog.h>

void openlog(const char* ident, int option, int facility);

void syslog(int priority, const char* format, ...);

void closelog(void);

int setlogmark(int maskpri);
```

Returns: previous log priority mask value