Operating systems fundamentals - B05

David Kendall

Northumbria University

Linux Inter-Process Communication (IPC)

- This lecture looks at some mechanisms that are used by processes to communicate with each other.
- These mechanisms come under the heading of *Inter-Process Communication*, abbreviated to *IPC*, and include:
  - Signals
  - Pipes
  - Named pipes

Signals

- A primitive form of communication between processes can be achieved by using *signals*
- A *signal* is used to notify a program about the occurrence of an event, e.g.
  - a hardware exception
  - user hits interrupt or quit at control terminal
  - an alarm timer expires
  - a call to kill()
  - termination of a child process
- Events may occur *asynchronously*
  - when the program is not expecting them

Signal status

- A signal is said to be:
  - *generated* when the event that causes the signal occurs
  - *delivered* when the action for a signal is taken
  - *pending* during the time between the generation of the signal and its delivery
  - *blocked* if unable to deliver due to a signal mask bit being set for the signal

Signals are software interrupts

- Each signal has a name
  - A signal is identified by a named constant (symbolic constant)
  - A set of predefined numbers: 1..MAXSIG
  - Details in **signal.h**
  - e.g. `SIGKILL` is signal number 9
  - `$ kill -l` displays a list of signals and their numbers on your system
- Signals may or may not be queued
  - Implementation-dependent to recognize multiple instances of a signal
- Order of service is not defined when different signals are pending on a process

Signal disposition

- Response to a signal, known as the disposition of the signal, can be one of the following:
  - Ignored (**SIG_IGN**)
    - Never posted to the process
  - Default action (**SIG_DFL**)
    - Termination in general
  - Catch
    - Needs a user-defined signal handler, or signal-catching function
- Most signals can be caught, or ignored except **SIGKILL** and **SIGSTOP**

Some important signals

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal, or death of controlling process
SIGINT	2	Term	Interrupt char (Control-C)
SIGQUIT	3	Term	Quit char (Control-\)
SIGKILL	9	Term	Kill signal
SIGPIPE	13	Term	Attempt to write to closed pipe (or socket)
SIGALRM	14	Term	Expiration of an alarm timer

Some important signals

Signal	Value	Action	Comment
SIGTERM	15	Term	default signal sent by `kill` command; can be caught by application allowing it to clean up and terminate gracefully
SIGCHLD	17	Ign	Child process exit
SIGCONT	18	Cont	Continue if stopped
SIGSTOP	19	Stop	Stop process

Installling your own signal handler

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *act,
              struct sigaction *oact);
```

- **sig** specifies the signal for which the action is being changed
- **act** is points to a **sigaction** structure that defines the new behaviour
- **oact**, if it is non-null, is assumed to point to a **sigaction** structure that will be used to store the definition of the old signal handler

Signal action definition

```
#include <signal.h>

struct sigaction {
  void (*sa_handler)(int);
  sigset_t sa_mask;
  int sa_flags;
}
```

- **sa_handler** can be **SIG_IGN**, **SIG_DFL** or the address of a user-defined function (taking an **int** parameter and returning **void**) that defines the behaviour when the signal is delivered
- **sa_mask** is a set of signals to be blocked during execution of the signal handler
- **sa_flags** allows the default behaviour to be modified (set to **0** for standard behaviour)

Signal handler example - ticker

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

#define TIMEOUT_SECS 2

int tick = 0;

void give_up(char *msg) {perror(msg); exit(1);}
void catchAlarm(int ignored) {tick += 1;}

int main() {

  struct sigaction act;

  act.sa_handler = catchAlarm;
  if (sigfillset(&act.sa_mask) < 0) {
    give_up("sigfillset");
  }
  act.sa_flags = 0;
```

Signal handler example - ticker

```
  if (sigaction(SIGALRM, &act, 0) < 0) {
    give_up("sigaction");
  }

  do {
    alarm(TIMEOUT_SECS);
    pause();
    printf("Tick %i\n", tick);
  } while (1);
}
```

Signal handlers in a shell script

- You can add signal handlers to your shell scripts too
- This can help to make the scripts more robust
- To add a signal handler, use the `trap` keyword
- It is good practice to write a shell function to act as the handler

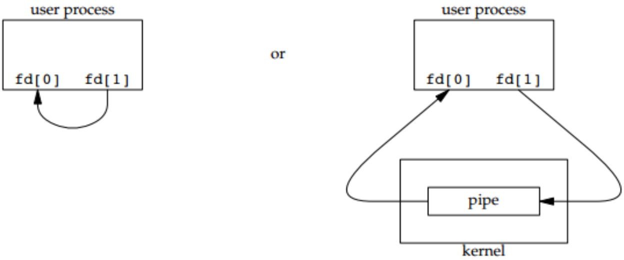
```
SIGINT_handler() {
  echo "This is the SIGINT handler"
}

trap SIGINT_handler SIGINT
```

- Notice that `trap` is followed by the name of the handler, then the name of the signal
- You can use the same `trap` statement to handle multiple signals

Signal handlers in a shell script - example

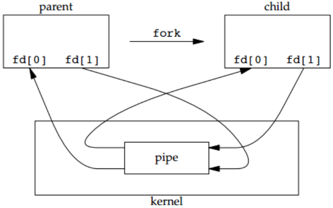
```
#!/bin/bash

SIG_handler() {
  echo "This is a signal handler for SIGQUIT and SIGTERM"
  exit 0
}

SIGINT_handler() {
  echo ""
  echo "This is the SIGINT handler"
  read -p "Press ENTER ..."
}

EXIT_handler() {
  echo "This is the EXIT handler"
  exit 0
}

trap SIG_handler SIGQUIT SIGTERM
trap SIGINT_handler SIGINT
trap EXIT_handler EXIT

while true
do
  echo Hello
  sleep 1
done
```

Pipes

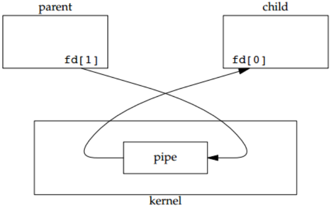


- Pipes are the oldest form of UNIX IPC
- They are a simple way to allow data to flow from one process to another
- They have two limitations
  1. Historically, they are half-duplex (data flows in one direction only)
  2. Processes must have a common ancestor in order to share a pipe - normally a pipe is created by a process, that process calls `fork()`, then the parent and child share the pipe

Pipes



Pipe after `fork()`



Pipe from parent to child

- Data can flow either from parent to child or from child to parent
- Example above shows data flow from parent to child
  - Parent closes the read end of its pipe (`fd[0]`)
  - Child closes the write end of its pipe (`fd[1]`)
  - Now the parent *writes* to, and the child *reads* from, the pipe

Pipes

- We have seen many examples in the shell of using pipes, e.g.

```
$ ls -l | wc -l
```

- The pipe is indicated using the `|` symbol and causes the standard output from the first command to be piped to the standard input of the second command
- These shell pipes are implemented using the techniques that we have just seen
- The shell creates a new pipe, forks a process for the first command, this process closes the read end of the pipe, maps its `stdout` to the write end and `exec`s the first command; the shell forks a process for the second command; this process closes the write end of its pipe, maps its `stdin` to the read end and `exec`s the second command

Named pipes

- We can get around the need for processes to have a common ancestor in order to share a pipe by using *named pipes*
- For example, in the shell we can use the `mkfifo` command

```
$ mkfifo mypipe
$ ls -l mypipe
prw-rw-r-- 1 cgdk2 cgdk2 0 Feb 13 08:31 mypipe
$ cat /etc/init.d/apport >mypipe
```

- In a *different terminal*, execute

```
$ cat <mypipe
```

  and you'll see the output from the command that's running in the first terminal