Process management

EPL222 - Lab1

Summary

- fork/daemon
- signal
- system
- named pipes (FIFOs)

daemon

- Ως daemon process στο Unix ορίζεται ένα process που εκτελείται μόνιμα στο υπόβαθρο (background) του συστήματος, χωρίς να έχει άμεση αλληλεπίδραση με το χρήστη μέσω του standard input/output ή κάποιου γραφικού interface.

- Πολλές υπηρεσίες στο Unix (για παράδειγμα η υπηρεσία που εκτυπώνει έγγραφα ή που αναλαμβάνει την αποστολή/υποδοχή της ηλεκτρονικής αλληλογραφίας) είναι υλοποιημένες ως daemon processes.

daemon

- Basic rules to coding a daemon
  - first call *fork()* and have the parent *exit()*
  - call *setsid()* to create a new session
  - change the current directory to the root directory (call *chdir()*)
  - set the file mode creation mask to 0 (call *umask()*)

fork() examples

- Fork system call is used for creating a new process, which is called child process, which runs concurrently with the process that makes the *fork()* call (parent process)
  - After a new child process is created, both processes will execute the next instruction following the *fork()* system call
  - A child process uses the same pc (program counter), same CPU registers, same open files which are used in the parent process.
  - It takes no parameters and returns an integer value.
  - The values returned by *fork()* can be:
    - **Negative Value**: creation of a child process was unsuccessful
    - **Zero**: Returned to the newly created child process
    - **Positive value**: Returned to parent or caller. The value contains process ID of newly created child process

fork() examples

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
    // make two process which run same
    // program after this instruction
    fork();

    printf("Hello world!\n");
    return 0;
}
```

Output:
Hello world!
Hello world!

fork() examples

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void main() {
    int pid;
    pid = fork();
    printf("%d \n", pid);
}
```

Output:

```
0
<child pid>
or
<child pid>
0
```

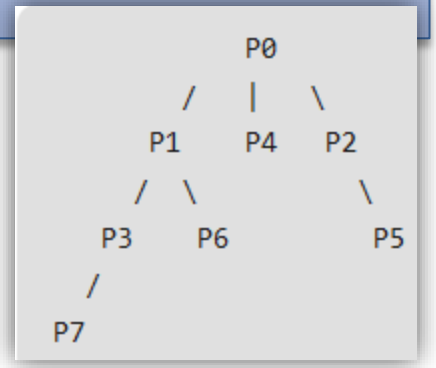
- What are the possible outputs, assuming the fork has succeeded?

fork() examples

```
#include <stdio.h>
#include <sys/types.h>
int main() {
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

Number of times hello printed is equal to number of process created.
Total Number of Processes = 2^n where n is number of fork system calls. So here n = 3, $2^3 = 8$

The main process: P0
Processes created by the 1st fork: P1
Processes created by the 2nd fork: P2, P3
Processes created by the 3rd fork: P4, P5, P6, P7



- Calculate number of times hello is printed.

Signals in C language

- A signal is a software generated interrupt that is sent to a process by the OS because of when user press ctrl-c or another process tell something to this process.
- There is a fix set of signals that can be sent to a process. Signal are identified by integers.
- Signal number have symbolic names.

Signals in C language

```
#define SIGHUP   1    /* Hangup the process */
#define SIGINT   2    /* Interrupt the process */
#define SIGQUIT  3    /* Quit the process */
#define SIGILL   4    /* Illegal instruction */
#define SIGTRAP  5    /* Trace trap. */
#define SIGABRT  6    /* Abort */
```

Linux Signals

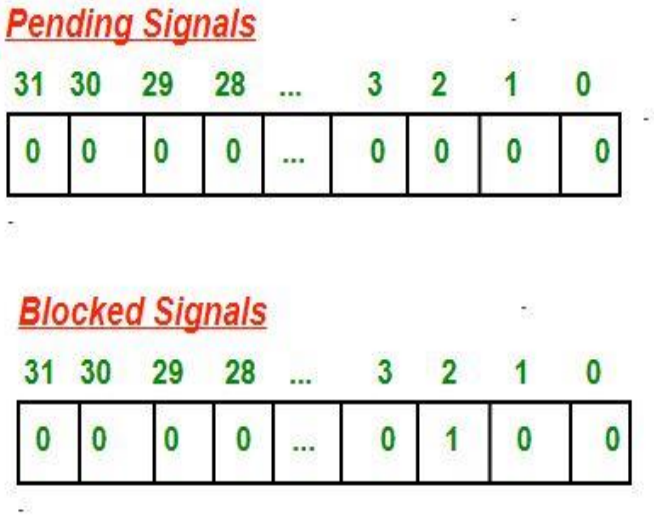
SIGHUP	Terminal hangup	SIGCONT	Continue
SIGQUIT	Keyboard quit	SIGTSTP	Keyboard stop
SIGTRAP	Trace trap	SIGTTOU	Terminal write
SIGBUS	Bus error	SIGXCPU	CPU limit exceeded
SIGKILL	Kill signal	SIGVTALRM	Virtual alarm clock
SIGSEGV	Segmentation violation	SIGWINCH	Window size unchanged
SIGPIPT	Broken pipe	SIGPWR	Power failure
SIGTERM	Termination	SIGRTMIN	First real-time signal
SIGCHLD	Child status unchanged	SIGRTMAX	Last real-time signal

Signals in C language

- For each process, the operating system maintains 2 integers with the bits corresponding to specific signal numbers.
- The two integers keep track of:
  - pending signals
  - blocked signals
- With 32 bit integers, up to 32 different signals can be represented.

Signals Example

- In the example below, the SIGINT (= 2) signal is blocked and no signals are pending.
- A signal is sent to a process setting the corresponding bit in the pending signals integer for the process. Each time the OS selects a process to be run on a processor, the pending and blocked integers are checked. If no signals are pending, the process is restarted normally and continues executing at its next instruction.
- If 1 or more signals are pending, but each one is blocked, the process is also restarted normally but with the signals still marked as pending.
- If 1 or more signals are pending and NOT blocked, the OS executes the routines in the process's code to handle the signals.



Hello world of Signal Handling

- The primary system call for signal handling is *signal()*, which given a signal and function, will execute the function whenever the signal is delivered. This function is called the signal handler because it handles the signal. The *signal()* function has a strange declaration:

```
int signal(int signum, void (*handler)(int))
```

Hello world of Signal Handling

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>
// for signal() and raise()

void hello(int signum){
    printf("Hello World!\n");
}

int main(){
    //execute hello() when receiving
    // signal SIGUSR1
    signal(SIGUSR1, hello);
   //send SIGUSR1 to the calling process
   raise(SIGUSR1);
}
```

- The program first establishes a signal handler for the user signal SIGUSR1.
- The signal handling function *hello()* does as expected: prints "Hello World!" to stdout.
- The program then sends itself the SIGUSR1 signal, which is accomplished via *raise()*,
- and the result of executing the program is the beautiful phrase: Hello World!

Alarm Signals and SIGALRM

- In the previous slide, we explored signal handling and user generated signals, particularly those from the terminal, the SIGKILL, and the user signals, or the SIGUSR1 and SIGUSR2.
- There are also O.S. generated signals like SIGALRM.
- A SIGALRM signal is delivered by the Operating System via a request from the user occurring after some amount of time.
- To request an alarm, use the *alarm()* system call:

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

- After seconds have passed since requesting the *alarm()*, the SIGALRM signal is delivered.
- The default behavior of SIGALRM is to terminate, so we can catch and handle the signal, leading to a nice hello world program

Alarm Signals and SIGALRM

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

void alarm_handler(int signum){
   printf("Buzz Buzz Buzz\n");
}
int main(){
   //set up alarm handler
   signal(SIGALRM, alarm_handler);
   //schedule alarm for 1 second
   alarm(1);
   //do not proceed until signal is handled
   pause();
}
```

- The program looks very much like our other signal handling programs, except this time the signal is delivered via the *alarm()* call.
- The *pause()* call will "pause" the program until a signal is delivered and handled.
- Pausing is an effective way to avoid busy waiting, e.g., while(1);, because the process is suspended during a pause and awoken following the return of the signal handler.

signal example

- Write a program that will continually alarm every 1 second.

signal example

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

void tick(int sig) {
  printf("tick\n");
  alarm(1);  //set a new alarm for 1 second
}

int main() {
  //set up alarm handler
  signal(SIGALRM, tick);

  //schedule the first alarm
  //set an alarm clock for delivery of a signal
  alarm(1);

  //pause in a loop
  while(1){
    pause();
  }
  return 0;
}
```

We would need to reset the alarm once the signal is delivered. The natural place to do that is in the signal handler. After the first SIGALRM is delivered and "tick" is printed, another alarm is schedule via *alarm(1)*. The process will resume after the *pause()*, but since it is in a loop, it will return to a suspended state. The result is an alarm clock buzzing every 1 second.

system()

- *system()* is used to invoke an operating system command from a C/C++ program.

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

- Using *system()*, we can execute any command that can run on terminal if operating system allows.
  - i.e. we can call system("dir") on Windows and system("ls") to list contents of a directory.

system() example

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main () {
   char command[50];
   strcpy(command, "ls -l");
   system(command);
   return(0);
}
```

list down all the files and directories in the current directory under unix machine.

Named Pipe or FIFO

- A named pipe (also known as a FIFO) is one of the methods for inter-process communication
- It is an extension to the traditional pipe concept on Unix. A traditional pipe is “unnamed” and lasts only as long as the process
- A named pipe, however, can last as long as the system is up, beyond the life of the process. It can be deleted if no longer used
- Usually a named pipe appears as a file and generally processes attach to it for inter-process communication. A FIFO file is a special kind of file on the local storage which allows two or more processes to communicate with each other by reading/writing to/from this file.
- A FIFO special file is entered into the filesystem by calling *mkfifo()* in C. Once we have created a FIFO special file in this way, any process can open it for reading or writing, in the same way as an ordinary file
  - However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.

Named Pipe or FIFO

```
int mkfifo(const char *pathname, mode_t mode);
```

- *mkfifo()* makes a FIFO special file with name pathname
  - Here mode specifies the FIFO's permissions.
  - It is modified by the process's umask in the usual way: the permissions of the created file are (mode & ~umask).
- Using FIFO: As named pipe(FIFO) is a kind of file, we can use all the system calls associated with it i.e. open, read, write, close.
- Example Programs to illustrate the named pipe: There are two programs that use the same FIFO. Program 1 writes first, then reads. Program 2 reads first, then writes. They both keep doing it until terminated.

named pipes example

```c
// C program to implement one side of FIFO
// This side writes first, then reads
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    int fd;
    char * myfifo = "/tmp/myfifo"; // FIFO file path
    // Creating the named file(FIFO):
    // mkfifo(<pathname>, <permission>)
    mkfifo(myfifo, 0666);

    char arr1[80], arr2[80];
    while (1)  {
              // Open FIFO for write only
              fd = open(myfifo, O_WRONLY);
              // Take an input arr2ing from user.80 is maximum length
              fgets(arr2, 80, stdin);
              // Write the input arr2ing on FIFO  and close it
              write(fd, arr2, strlen(arr2)+1);
              close(fd);
              // Open FIFO for Read only
              fd = open(myfifo, O_RDONLY);
              // Read from FIFO
              read(fd, arr1, sizeof(arr1));
              // Print the read message
              printf("User2: %s\n", arr1);
              close(fd);
    }
    return 0;
}
```

```c
// C program to implement one side of FIFO
// This side reads first, then writes
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    int fd1;
    char * myfifo = "/tmp/myfifo"; // FIFO file path
    // Creating the named file(FIFO):
    // mkfifo(<pathname>,<permission>)
    mkfifo(myfifo, 0666);

    char str1[80], str2[80];
    while (1)  {
              // First open in read only and read
              fd1 = open(myfifo,O_RDONLY);
              read(fd1, str1, 80);

              // Print the read string and close
              printf("User1: %s\n", str1);
              close(fd1);

              // Now open in write mode and write
              // string taken from user.
              fd1 = open(myfifo,O_WRONLY);
              fgets(str2, 80, stdin);
              write(fd1, str2, strlen(str2)+1);
              close(fd1);
    }
    return 0;
}
```

Filesystem related	
close	Close a file descriptor.
link	Make a new name for a file.
open	Open and possibly create a file or device.
read	Read from file descriptor.
write	Write to file descriptor
Process related	
execve	Execute program.
exit	Terminate the calling process.
getpid	Get process identification.
setuid	Set user identity of the current process.
ptrace	Provides a means by which a parent process my observe and control the execution of another process, and examine and change its core image and registers.
Scheduling related	
sched_getparam	Sets the scheduling parameters associated with the scheduling policy for the process identified by `pid`.
sched_get_priority_max	Returns the maximum priority value that can be used with the scheduling algorithm identified by `policy`.
sched_setscheduler	Sets both the scheduling policy (e.g., FIFO) and the associated parameters for the process `pid`.
sched_rr_get_interval	Writes into the timespec structure pointed to by the parameter `tp` the round robin time quantum for the process `pid`.
sched_yield	A process can relinquish the processor voluntarily without blocking via this system call. The process will then be moved to the end of the queue for its static priority and a new process gets to run.

Interprocess Communication (IPC) related	
msgrcv	A message buffer structure is allocated to receive a message. The system call then reads a message from the message queue specified by msqid into the newly created message buffer.
semctl	Performs the control operation specified by cmd on the semaphore set semid.
semop	Performs operations on selected members of the semaphore set semid.
shmat	Attaches the shared memory segment identified by shmid to the data segment of the calling process.
shmctl	Allows the user to receive information on a shared memory segment, set the owner, group, and permissions of a shared memory segment, or destroy a segment.
Socket (networking) related	
bind	Assigns the local IP address and port for a socket. Returns 0 for success and -1 for error.
connect	Establishes a connection between the given socket and the remote socket associated with sockaddr.
gethostname	Returns local host name.
send	Send the bytes contained in buffer pointed to by *msg over the given socket.
setsockopt	Sets the options on a socket
Miscellaneous	
fsync	Copies all in-core parts of a file to disk, and waits until the device reports that all parts are on stable storage.
time	Returns the time in seconds since January 1, 1970.
vhangup	Simulates a hangup on the current terminal. This call arranges for other users to have a "clean" tty at login time.