Concurrent Programming

In Linux/Unix

Identifiers

pid – Process Indentifier

- usually from 0 to 32K
- given by the system to each new process
- we can get it by *int getpid()*
- 0 is a number of special kernel process to exchange processes - scheduler
- **1** boot kernel special process number *init*
- 2 in Unix implementations designed to work with virtual memory, page demon.

Identifiers

ppid – parent process identification number

- each process has its own parent number (except init)
- it can be obtained by int getppid()
- Example:

```
main()
 {
 printf(„pid = %d, ppid = %d\
  n”,getpid(),getppid());
 exit(0);
 }
```

Identifiers

- **Identifier** group of process
  - each process is a member of a process group
  - the group is identified by a positive integer
  - many of processes can have the same number
  - the process group leader has the same id as the group
  - the value is checked by: *int getpgrp(int pid);*
  - if pid = 0 then the process group id is for the current process
  - if pid > 0 it's for the given pid
  - you can set the group id: *int setpgrp(int pid, int pgrp);*
  - you must of course have rights to such a process.

Identifiers

- **an identifier of the terminal group and the controlling terminal**
  - positive integer
  - any process can be a member of a terminal group
  - process group leader has id = terminal group id
  - this is the process that opened the terminal
  - the terminal has only one controlling process
  - it is a controlling process
  - a terminal group identifier identifies the controlling terminal

Identifiers

- the controlling terminal sends signals that are received when certain keys on the terminal are pressed and at the and the reported shell.
- the control terminal can be referenced automatically via the device /dev/tty*
- The *ioctl* function can be used with the appropriate option to query or establish a terminal group identifier for your controlling terminal (*TIOCGPGRP* or *TIOCSPGRP*)
- can be disconnected by the function *setpgrp*

Identifiers

- **real user ID**
  - positive integer assigned to each user
  - can be obtained by unsigned short getuid();
  - the mapping of uids and usernames is in the /etc/passwd file

Identifiers

- **real group ID**
  - a positive integer to which all users are assigned
  - can be obtained by unsigned short getgid();
  - the mapping of **gid**s and group names is in the /etc/group file

Identifiers

- ***effective user ID***
  - *an efective user ID is assigned to each process*
  - *can be obtained by **unsigned short geteuid();***
  - *usually the same value as the real ID.*
  - *a special 1-bit marker can be specified*
    - *if it is **1** then during the execution of this program the valid user ID of the process becomes the user ID assigned to the owner of the program file.*
    - *eg root files set with **S** are executed with root privileges.*

Access rights

- *each process is assigned four types of identification numbers*
  - *real user ID*
  - *real group ID*
  - *effective user ID*
  - *effective group ID*
- *In addition, in Linux we use separate permissions for the user, group and others*
- *RWXRW-R-- 764*

Process

- ***fork*** *– the only way to create a new process in Unix is by calling a function* **int fork();** *(this does not apply to the init process).*
- *It is called in two cases:*
  - *When a process wants to create a copy of itself so that one of them can perform some other task*
  - *When a process wants to execute a second program then the copy is executed by exec, which is usually what shell programs do.*

Process

- *makes a copy of the calling process*
- *the process that triggers the **fork** is called the parent or ancestor*
- *the resulting process is called a child process*
- *One call to fork returns twice the value*
  - *pid of the new process to the ancestor*
  - *0 to new process*
- *when the error occurred -1 is returned*

Process

```
main ()
{
int childpid;
if ((childpid = fork()) == -1)
    {
    perror(„can't fork");
    exit(1);
    }
else
    if (childpid == 0) /* child process */
        {
        printf(„child: child pid = %d, parent pid = %d\n",getpid(),
getppid());
        exit(0);
        }
    else   /* parent process */
        {
        printf(„parent: child pid = %d, parent pid = %d\n",childpid,
getpid());
        exit(0);
        }
}
```

Process

The child process copies the following values from the parent process:

- real user ID,
- real group ID,
- effective user ID,
- effective group ID,
- process group identifier,
- terminal group identifier,
- main directory,
- current working directory,
- settings for signal handling,
- file access mode mask.

Process

A child process differs from a parent process in that:

- has a new unique process identifier
- has a different parent process ID
- the child process has its own copies of the parent process's file descriptors
- the time remaining until the alarm is triggered is reset to zero for the child process.

exit

- ***exit*** – is used to end the process.
- once executed, it never returns to the process that triggered it.
- the process calling en exit passes an integer to the kernel representing the end state of the process
- the parent process can read it using the function ***wait***
- You have to distinguish the **exit** system function from the standard C function where the I / O buffers are flushed and the **exit** system is called.
- If you want to call exit immediately, use _exit

exit

- when the parent called wait, it is notified of the end of the child
- When called in a child, it returns from the parent's wait function.
- if the parent process did not call wait, then the process to be terminated becomes a ghost-process (zombie). Resources are freed by the kernel, but the state must be maintained until parent ask for it.
- if it is an orphan, the parent id becomes 1(init) or user init context
- if all fields of ids: process, process group and terminal group of the terminated process are identical, the kernel sends a suspend signal, SIGHUP, to each process that has the process group identifier the same as the terminating process.

exec

- the only way for Unix to execute any program is by calling exec*
- it replaces the program of the current process with a new program.
- the process that called the exec system call is called the calling process
- the program to be executed is called new program (not process)
- the exec function can only return to the program it was called from when it turns out that an error has occurred.

exec

There are 6 different versions of exec

- `int execlp(char *filename, char *arg0, char *arg1, ..., char *argn, (char *) 0);`
- `int execl(char *pathname, char *arg0, char *arg1, ..., char *argn, (char *) 0);`
- `int execle(char *pathname, char *arg0, char *arg1, ..., char *argn, (char *) 0, char **envp);`
- `int execvp(char *filename, char **argv);`
- `int execv(char *pathname, char **argv);`
- `int execve(char *pathname, char **argv, char **envp));`

exec

- some of these functions have arguments passed by the argument list, and some have a pointer to an array of strings with arguments
- some of them have pathname and some have filename which is transformed into pathname by the PATH environment variable
- some of them have an envp and some do not. When not present, the default value of the external variable environ is used.

exec

The new program will inherit the following characteristics

- process ID
- parent process ID,
- group process ID,
- terminal group ID,
- time left until the alarm signal appears,
- main directory
- current working directory,
- file access mode mask,
- real user ID
- effective user ID,
- files occupied.

exec

- The new program may differ in:
  - valid user id
  - valid group id
- If the user bit for the program to be executed with exec is set to 1, the valid user id becomes the user id belonging to the owner of the program file.
- similar with the group
- Signals
  - those that were ignored will be
  - those who completed the process will also react the same
  - And those that were handled will not be any more, because the address of the signal handling function is different.

wait

It causes the process to wait until one of its children has finished running

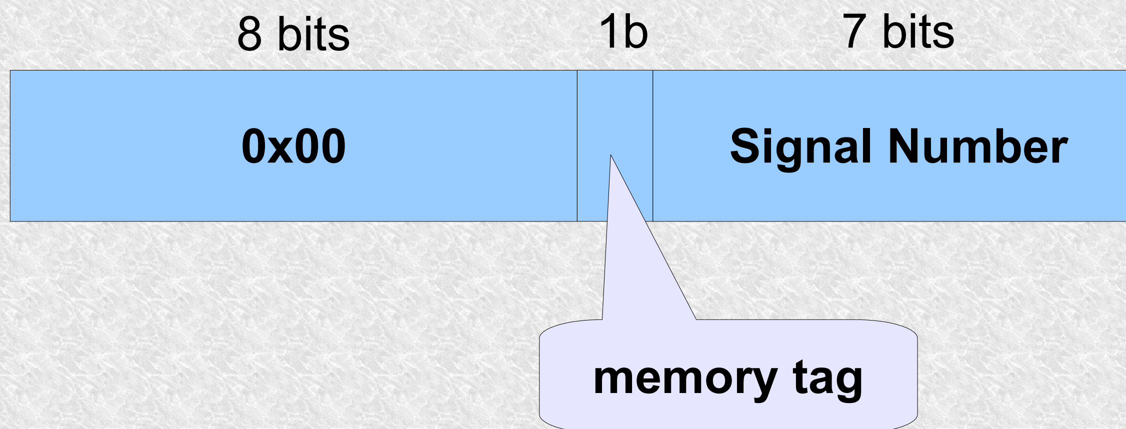
int wait(int *status);

wait returns the id of the child process that exits as a result:

1. child called *exit* the variable status has

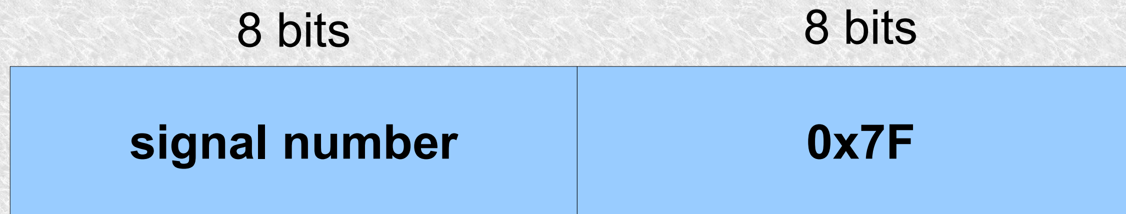
wait

2. the child terminated upon incoming signal the variable status has:



wait

3 the child was exited while executing it in tracking mode. The variable status has:



wait

- if the process ending generated core then the memory tag bit is 1
- if the process has no children, the function returns immediately –1
- if the process has children, it is suspended until one of them terminates
- *if the status is not NULL then it will get the value returned by the child by the exit function.*
- when the child process exits, the parent gets the SIGCLD signal, you can write signal handlers where we call the functions wait
- you can also ignore the SIGCLD signal (in the parent, of course)

```
signal(SIGCLD, SIG_IGN);
```

in this way, the kernel will be informed that the parent does not care about their children's status, so the zombie processes will be removed immediately.

wait3

it differs from wait in that it has the ability not to wait for the child process to finish

```
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
int wait3(union wait *status, int options, struct
rusage *rusage);
```

- if options = WNOHANG then wait3 does not wait for the end of the child but returns 0.
- *rusage* informs the parent process about the child's CPU usage time

Signals

- A signal tells the process that an event has occurred. They are also called software interrupts.
- They are sent asynchronously
- Each signal has a name as described in ***<signal.h>***
- You can send from one process to another or from kernel to process.
- A system function ***kill*** is used to send signals

Signals

int kill (int idproc, int sig);

- Only its owner or supervisor can send a signal to the process.
- If pid = 0, the signal is sent to everyone in the sender's process group.
- If pid = -1 and the supervisor is not root, the signal will be sent to all processes with the same owner as the valid uid of the sending process.
- If pid = - 1 and supervisor is root, the signal is sent to all processes except system processes (usually 0 or 1).
- If pid <-1, the signal is sent to all processes whose group id is = absolute value.
- If sig = 0 it is a test signal, e.g. to check pid.
- The shell kill command does the same but takes arguments from the command line.

Signals

- Certain characters from the terminals cause sending signals, e.g.
  - *Ctrl*+C or *Delete SIGINT*,
  - *Ctrl*+\ *SIGQUIT*.
  - *Ctrl+Z SIGTSTP*
- You can match almost any character from the terminal to the break and exit character.
- Certain hardware situations also generate signals
  - When the floating point calculation fails, we have *SIGFPE*
  - A reference to an out-of-process address space *SIGSEGV*
- Certain situations detected by the system software such as high priority data appearing in a socket.

Signals

What the process can do with the signal?

- can provide a function that will be called whenever a special type of signal occurs. (**signal handler**)
- He can ignore it, all but SIGKILL and SIGSTOP
- The process may allow the default behavior to take place.

Signals

To determine how the signal is to be handled, the process calls the system function:

```
   #include <signal.h>
   int (*signal (int sig, void (*func)(int)))
(int);
```

- This means that the signal function passes a pointer to the function that passes an integer.
- *func* specifies the address of a function that passes nothing.
- It can have two fixed parameters
  - *SIG_DFL* means the signal will do default actions
  - *SIG_IGN* means to ignore the signal
- Signal always returns the previous value of func for a given signal.

Signals

Example: We want SIGUSR1 to be ignored, we write this

```
signal (SIGUSR1, SIG_IGN);
```

We want SIGINT to call my_interrupt () functions, then we write it

```
#include <signal.h>
extern void my_interrupt();
...
signal(SIGINT, my_interrupt);
```

When the function is called to handle a signal, the interrupt number is taken as the first parameter, so one function can handle many signals and recognize it.

Signals

- **SIGALRM** - Alarm clock - The process can set an alarm clock for a certain number of seconds. Default action, process termination
  unsigned int alarm(unsigned int sec);*After sec of seconds, the kernel will pass the SIGALRM signal to the process that caused the alarm*.
  unsigned int sleep(unsigned int sec); The process puts the process to sleep for seconds.
- **SIGCLD** - Sent to the parent process when the child ends..
- **SIGHUP** - Suspend - When the terminal is closed, a SIGHUP is sent to the processes for which it is the controlling terminal. It is also sent to group processes when the leadership process ends. Process termination by default.
- **SIGINT** - Interrupt character - usually when the user presses the interrupt key on the terminal. By default, termination
- **SIGKILL** – Absolute process termination, it cannot be ignored or captured.

Signals

- **SIGPIPE** - Data is not received from the communication link - the writer gets it when it sends data to a link or queue where there is no one to receive. By default, termination
- **SIGQUIT -** Quit character - when the user presses the break character. It is similar to SIGINT but here the memory image is generated.
- **SIGSEGV** - Violation of segmentation. (Internal 11)
- **SIGSTOP** - absolute stop, cannot be ignored or intercepted, then the process can be reactivated via SIGCONT
- **SIGTERM** - Programmatic process termination - Sent by another process to kill the process. Default ending.
- **SIGUSR1** i **SIGUSR2** - User Defined - It can be used for inter-process communication, but it doesn't carry much information except that it has appeared.

Signals

```
#include <signal.h>

void handling2()
{
    printf("Program received signal ctrl+c\n");
}
void handling11()
{
    printf("Something override memory\n");
    exit(-1);
}
void handling28()
{
    printf("Change terminal size \n");
}
main()
{
char word[5];
signal(2,handling2);
signal(11,handling11);
signal(28,handling28);
strcpy(word,"ala ma kota jednak kot jej nie lubi");
printf("%s\n",word);
sleep(20);
printf("Normal ending\n");
}
```

Reliable signals

Early implementations of signals were unreliable, when there were many of them there was a so-called racing situation and it could get lost.
The following properties have been added to obtain reliable signals:

- The signal handler is still installed after a signal has occurred. Previously, it was removed, and before the process called signal functions again, some signals could be lost
- The process should be able to suspend the incoming signal, it is not about ignoring it, but temporarily suspending it until it is ready to serve it.
- While the signal is handled by the process, the second signal is stored and handled when the first signal is handled.

Signal mask

we can define the signal mask with macroinstructions

```
#include <sygnal.h>
sigmask(int sig);
```

e.g. for SIGQUIT and SIGINT we create a mask like this:

```
int mask;
mask = sigmask(SIGQUIT) | sigmask(SIGINT);
```

The implementation of the mask is a 32-bit number, for each bit there is one signal.

Singal block

To block one or more signals, a system function must be called

```
int sigblock(int mask);
```

As an argument we give a mask mask, these signals will be added to the set of blocked signals.
The return value is the mask in effect before calling this function.

Signal block

The signal is unlocked using:

```
int sigsetmask(int mask);
```

The mask argument does not contain the signal we want to unblock.

Signal block

Example:

If we do not want any signal to appear while executing some part of the program, we do it like this:

```
int oldmask;
oldmask = sigblock(sigmask(SIGQUIT) |
sigmask(SIGINT));
/* protected program fragment */
sigsetmask(odlmask); /* restore the old mask */
```

Signal block

In the V system there is no concept of a signal mask but it can be blocked by the function:

```
int sighold(int sig);
```

And it is used to unlock

```
int sigrelse(int sig);
```