**March 5, 2008**
**Operating Systems**
**LIACS**
**Spring Semester 2008**
**Assignment #3**

Deadline: **Tuesday, March 18, 2008**
You are encouraged to work in teams of 2 persons.

## Goal
Learn what signals and interval timers are and how they are used in a Linux environment.

## Introduction
Systems must be able to react to changes in system state that are not captured by internal program variables and are not necessarily related to the execution of the program. For example, a hardware timer goes off at regular time intervals and must be dealt with. Packets arrive at the network adapter and must be stored in memory. Programs request data from a disk and then sleep until they are notified that the data is ready. Parent processes that create child processes must be notified when their children terminate.

Modern systems react to these changes by making abrupt changes in the control flow. These abrupt changes in control flow occur at all levels of a computer system. For example, at the hardware level, events detected by the hardware trigger abrupt control transfers to exception handlers. At the operating systems level, the kernel transfers control from one user process to another via context switches. At the application level, a process can send a signal to another process that abruptly transfers control to a signal handler in the recipient.

In this assignment we will look at the application level. We will see how a process can send signals to other processes and how a process handles such a signal. We will manually send signals, set up timers that automatically send signals and write our own signal handler.

## Remark 1
Upon termination of a child process the kernel datastructures for the child process are still kept around (until the parent reaps (does a wait or waitpid) on the child). A terminated child which has not been reaped is called a *zombie*.
In general there are three cases to consider:

1) Parent is alive and does not reap the child (does not do a wait or waitpid), the child will stay a zombie upon termination for as long as the parent lives.
2) Parent is alive and reaps the child.
3) Parent dies before the child terminates. In this case eventually the `init` process will reap the terminated child.

## Remark 2
There are 4 ways to send signals to processes:

1) With the kill (`/bin/kill`) program.

2) A process can send signals to any process (including itself) by calling the kill function:
   ```
   int kill(pid_t pid, int sig);
   ```
   Include `<sys/types.h>` and `<signal.h>`.
   Its name is misleading because any of the 30 or so signals can be sent with it.

3) From the keyboard: Typing `ctrl-c` causes a `SIGINT` signal to be sent to the shell. The shell then invokes a signal handler (officially this is called: catches the signal). This signal handler sends a `SIGINT` to every process in the foreground process group. If a process, does not have a signal handler for this signal the default action will be termination of the process. Typing `ctrl-z` sends a `SIGTSTP` signal to the shell, which catches it and sends `SIGTSTP` to every process in the foreground process group. In the default case (the receiving process does not have a signal handler installed), the receiving process is stopped (suspended).

4) A process can send `SIGALRM` signal to itself by calling the alarm function:
(`unsigned int alarm (unsigned int secs);`
This function returns the remaining number of seconds of the previous alarm, or 0 if no previous alarm. The alarm function arranges for the kernel to send a `SIGALRM` signal to the calling process in secs seconds. If `secs` is zero, then no new alarm is scheduled. In any event, the call to alarm cancels any pending alarms and returns the number of remaining seconds remaining until any pending alarm was due to be delivered (had not this call to alarm cancelled it), of 0 if there were no pending alarms.

## Assignment

The first part of this assignment introduces you to signals. The second part introduces interval timers and leads to a program that determines a program's execution time.

## Part a)

Download, compile and run the file `assignment3.c` from the site. Run the program a few times for each question as the output can slightly vary in some cases.

- Describe what this program does.
- Change line 59 so it also sends `SIGUSR1`. What happens to the program's output? Can you come up with a reason why this doesn't always work as you'd expect? Afterwards change it back to `SIGUSR2`.
- Try removing the `sleep` statement from line 51, 61 or both. Are both these `sleep` statements required? Note that `sleep` puts the program to sleep for a given number of seconds (or indefinitely), but that signals can wake the program.

## Part b)

Do the Lab Exercise on pages 232-239 of the book (Gary Nutt, Operating Systems, 3rd Ed.)

**Deliverables:**

i A README file which contains a list of items you turned in + in what fashion they are to be used. Also mention the platform(s)/kernel version(s) on which you have tested your programs.
ii A file containing the answers to part a.
iii The C program for part b.
iv A one-page lab report detailing what you have done in the lab, what problems you encountered and more importantly what you have learned from this and what you have learned from your lab partner.

Make sure all files contains the names of the authors, student IDs, assignment number and date turned in!

**Due date**: as stated before March 18.

Put all files you want to deliver into a separate directory (e.g., *assignment3*), free of object files and/or binaries, and create a gzipped tar file of this directory:

tar -cvzf assignment3.tgz assignment3/

**Mail** your gzipped tar file to Sjoerd Henstra (e-mail: shenstra at liacs dot nl) and *let the subject field of your e-mail contain the string* "OS Assignment 3".