FOLHA DE PROBLEMAS Nº 4

Sinais

---

1. – "Acordado" por um sinal … ou os efeitos dos sinais durante o "sono" … !

Compile e execute o seguinte programa:

```
// PROGRAMA p01a.c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
#include <stdlib.h>

void sigint_handler(int signo)
{
  printf("In SIGINT handler ...\n");
}

int main(void)
{
  if (signal(SIGINT,sigint_handler) < 0)
  {
    fprintf(stderr,"Unable to install SIGINT handler\n");
    exit(1);
  }

  printf("Sleeping for 30 seconds ...\n");
  sleep(30);
  printf("Waking up ...\n");

  exit(0);
}
```

a) Verifique o que acontece se for enviado o sinal SIGINT ao programa em execução. E se for enviado outro sinal, por exemplo, SIGUSR1. Verifique qual o código de terminação do programa (`echo $?`) nas duas situações.

b) Analise a descrição da função `sleep()`, no manual, e altere o programa por forma a garantir que o "sono" dura sempre 30 segundos, aproximadamente.

c) Substitua a chamada `signal()` por uma chamada `sigaction()`.

2. – Instalação de um *signal handler*. Sinais recebidos durante a execução de um *handler*.

Compile e execute o seguinte programa:

```
// PROGRAMA p02a.c
#include ...

void sigint_handler(int signo)
{
  printf("Entering SIGINT handler ...\n");
  sleep(10);
  printf("Exiting SIGINT handler ...\n");
}
```

```
int main(void)
{
 struct sigaction action;

 action.sa_handler = sigint_handler;
 sigemptyset(&action.sa_mask);
 action.sa_flags = 0;

 if (sigaction(SIGINT,&action,NULL) < 0)
 {
       fprintf(stderr,"Unable to install SIGINT handler\n");
       exit(1);
 }

 printf("Try me with CTRL-C ...\n");
 while(1) pause();

 exit(0);
}
```

a) Tecle várias vezes CTRL-C durante a execução. Quantas ocorrências do sinal SIGINT foram processadas? Interprete o resultado.

b) Usando o comando `kill` da *shell*, envie, a partir de outra janela de terminal, o sinal SIGTERM ao processo, durante o ciclo de espera no interior do *handler*. Verifique o que acontece e interprete o resultado.

c) Instale também um *handler* para o sinal SIGTERM (este *handler* deve apenas escrever mensagens de entrada e de saída do *handler*). Tecle várias vezes em CTRL-C durante a execução e envie o sinal SIGTERM ao processo durante o período em que o *handler* de SIGINT está a ser executado. Interprete os resultados.

3. – Instalação de *signal handlers* e tratamento de sinais. Envio de sinais com diversas origens.

a) Escreva um programa que mostre no écran, com intervalos de 1 segundo, o valor de uma variável `v`. A variável terá o valor inicial zero e, o seu valor deverá começar por crescer. O sentido (crescente / decrescente) de variação do valor da variável poderá ser alterado enviando ao processo os sinais SIGUSR1 (sentido crescente) e SIGUSR2 (sentido decrescente). Use o mesmo *handler* para os dois sinais. Execute o programa e envie os sinais usando o comando `kill` da *shell*, a partir de outra janela de terminal. Termine o programa enviando-lhe o sinal SIGTERM.

b) Altere o programa da alínea anterior por forma criar dois processos, pai e filho. O processo-filho mostra o valor da variável, sendo o sentido de variação desta controlado pelo processo-pai, que envia ao filho, de forma aleatória, os sinais SIGUSR1 e SIGUSR2. Ambos os processos terminam depois de serem mostrados 50 valores.

4. – wait() e waitpid() revisitadas. SIGINT com origens diferentes e seus efeitos.

Compile o seguinte programa:

```
// PROGRAMA p04a.c
#include ...

int main(void)
{
  pid_t pid;
  int i, n, status;

  for (i=1; i<=3; i++) {
    pid=fork();
```

```
      if (pid == 0){
        printf("CHILD no. %d (PID=%d) working ... \n",i,getpid());
        sleep(i*7); // child working ...
        printf("CHILD no. %d (PID=%d) exiting ... \n",i,getpid());
        exit(0);
      }
    }

    for (i=1 ;i<=4; i++ ) {
      printf("PARENT: working hard (task no. %d) ...\n",i);
      n=20; while((n=sleep(n))!=0);
      printf("PARENT: end of task no. %d\n",i);
      printf("PARENT: waiting for child no. %d ...\n",i);
      pid=wait(&status);
      if (pid != -1)
        printf("PARENT: child with PID=%d terminated
                with exit code %d\n",pid,WEXITSTATUS(status));
    }

    exit(0);
  }
```

a) Execute o programa. Verifique, usando o comando `ps u`, a partir de outra janela de terminal, que os processos-filhos ficam apenas temporariamente no estado *zombie*. No entanto, esta solução tem alguns inconvenientes. Identifique-os, tendo em conta que as tarefas executadas por pai e filhos podem ter durações muito diversas e o número de processos que são libertados do estado *zombie* em cada chamada `wait()`.

b) Note que a chamada `wait()` bloqueia o processo-pai durante alguns momentos, se, quando for executada, nenhum dos filhos estiver no estado *zombie*. Existe uma forma alternativa de um processo-pai esperar que os seus filhos terminem, sem bloquear à espera que isso aconteça: usar a chamada `waitpid(-1,&status,WNOHANG)`. Consulte um manual ou os apontamentos desta unidade curricular sobre o significado desta chamada e altere o programa por forma a usá-la, em substituição de `wait()`, evitando que os processos-filhos fiquem no estado *zombie*. Em cada iteração do ciclo `for()` do processo-pai liberte do estado *zombie* todos os processos-filhos que estiverem nesse estado.

c) Execute o programa e, depois de todos os processos-filhos terem sido criados, tecle CTRL-C. Verifique o que aconteceu aos processos que estavam em execução. Repita a execução e verifique o que acontece se enviar o sinal SIGINT ao processo-pai, a partir de outra janela de terminal. Interprete os resultados.

5. – Tratando o sinal SIGCHLD para evitar *zombie*'s.

a) Uma forma de evitar que os processos-filhos fiquem no estado *zombie*, consiste em informar o sistema operativo que se pretende ignorar o sinal SIGCHLD que é enviado a um processo-pai sempre que algum dos seus filhos termina (qual a acção por omissão associada a este sinal ?). Modifique o programa do problema 4.a por forma a o processo-pai ignore os sinais SIGCHLD. Note que, nesta situação, não faz sentido que o processo-pai execute qualquer chamada `wait()`/`waitpid()`.

b) Para recolher o estado de terminação dos processos-filhos, o processo-pai poderia instalar um *handler* para o sinal SIGCHLD e executar uma das chamadas chamada `wait()`/`waitpid()`no interior do *handler*. Tendo em conta as conclusões retiradas no problema 2.a, comente a eficácia desta solução no sentido de evitar que os processos-filhos fiquem no estado *zombie*. Modifique o programa fornecido por forma a criar 10 processos-filhos e altere o tempo de "trabalho" dos filhos por forma a ser igual para todos (ex: `sleep(15)`).

6. – Uso de SIGALRM e de SIGKILL.

Escreva um programa (`limit`) que lance outro programa (`prog`) em execução, impondo um tempo limite (`t`) para a execução deste (ex: `limit t prog prog_arg1 prog_arg2` … , em que os `prog_argn`

são argumentos do programa `prog`). Caso o tempo limite seja excedido, deve ser enviado o sinal SIGKILL ao processo `prog`. Para testar o programa `limit` escreva um programa (`prog`) que apresente no écran, de 5 em 5 segundos, a mensagem indicada por `prog_arg1` (ex: `I'm alive!`), durante um tempo máximo de 30 segundos. O programa `limit` deverá indicar no écran a situação em que se deu a terminação do programa `prog`, natural ou forçada pelo limite temporal imposto. Justifique o envio do sinal SIGKILL para este efeito. Use um alarme para controlar o tempo limite de execução.

7. – Envio de sinais aos processos-filhos.

Escreva um programa que:

- lance em execução os programas cujos nomes dos ficheiros executáveis lhe sejam passados como argumentos da linha de comandos, ficheiros estes que devem estar num dos directórios do *path*;
- fique a aguardar a terminação dos processos criados, apresentando no écran a *PID* e o código de terminação de cada um desses processos, à medida que eles forem terminando;
- se algum dos processos retornar um código de terminação com um valor diferente de zero, envie um sinal a todos os processos que ainda não terminaram, forçando a sua terminação.