# Relatório 08

Vinícius de Oliveira Peixoto Rodrigues (245294)

Outubro de 2022

## Item (a)

O programa cria um processo-filho, que "recebe a tarefa"de dormir por 6 segundos. O processo-pai, após o `fork`, dorme por aproximadamente 1 segundo e em seguida termina sua execução sem esperar pela finalização do filho. Enquanto isso, o processo-filho ainda está rodando, e termina sua execução somente após a morte do pai.


## Item (b)

```
❯ ./ex_b.out &
[1] 139159
pid(processo)=139159, pid(processo-pai)=137385

Processo-pai:
pid(processo-filho)=139160

Processo-filho:
pid(processo-filho)=139160, pid(processo-pai)=139159
Processo 139160 terminou!
nuke@elpis ~/faculdade/ea872/08-processes/report/lab08_exercicios main
* ⇡
❯ Processo 139159 terminou!

[1]  + 139159 done       ./ex_b.out
nuke@elpis ~/faculdade/ea872/08-processes/report/lab08_exercicios main
* ⇡
❯
```

```
nuke      139159  0.0  0.0   2484   868 pts/1    SN   04:28   0:00 ./ex_b.out
nuke      139160  0.0  0.0      0     0 pts/1    ZN   04:28   0:00 [ex_b.out] <defunct>
nuke      139164  0.0  0.0  10024  3472 pts/0    R+   04:28   0:00 ps ux
```

O processo-filho se encerra antes do pai e entra no estado `zombie` (i.e., "dead but not reaped"), visto que o processo-pai não usou a syscall `wait` para ler o estado de saída do processo-filho. O processo-filho só é limpo após a morte do processo-pai (que implica na terminação/cleanup de todos os childs).

## Item (c)

A mensagem não aparece porque o primeiro processo-filho termina por meio da chamada `exit` imediatamente após sair do `sleep` (e antes de chegar no `printf` fora do escopo do primeiro condicional).

Além disso, os processos não entram no estado `zombie` porque foi usada a syscall `wait` para esperar pela finalização dos dois processos-filhos (de modo que o processo-pai permanece vivo durante toda a duração dos processos-filhos).

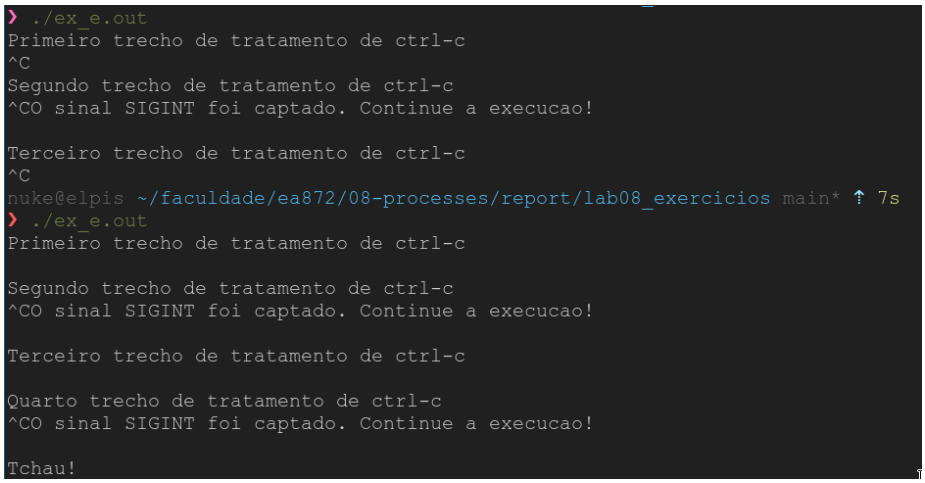
# Item (d)

O programa spawna um child process que chama a syscall `execlp(const char *file, const char *arg, ...)`, que troca a imagem do programa atual pela imagem do programa identificado por `file`, usando `arg` como o `argv[0]` do programa e os argumentos variádicos em `...` como o restante dos argumentos.

Enquanto isso, o processo-pai mede o tempo de execução do processo filho (marcando o timestamp de início, depois esperando pelo processo filho com a syscall `wait`, e depois marcando o timestamp de quando a chamada `wait` termina).

A diferença do tempo de execução se deve ao fato de que o tempo de CPU usado é definido de forma não determinística pelo `scheduler` e varia dependendo da quantidade/uso de CPU dos outros processos rodando no sistema.

# Item (e)

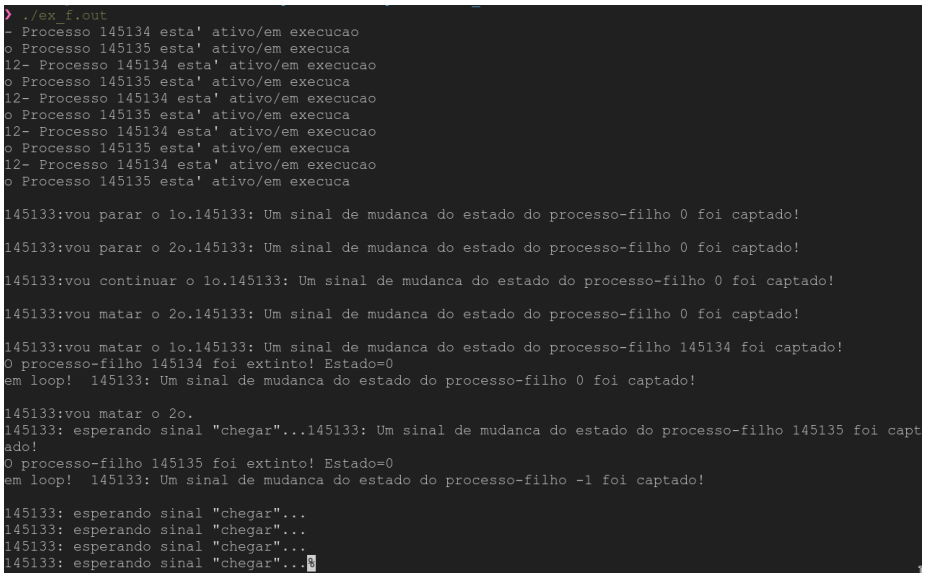



Primeiramente, é criado um ponteiro de função `void (*ger_antiga)()` (isto é, ponteiro para uma função de tipo `void` que não recebe argumentos). Esse tipo é "compatível"com o tipo `void (*sighandler_t)(int)`, que define os handlers de signals registrados por meio da chamada `signal()`.

Em seguida, são registrados, em intervalos de 5s, três handlers diferentes para o sinal `SIGINT`:

1. `SIG_IGN`, que sinaliza que o signal `SIGINT` deve ser ignorado;
2. `ger_nova`, que printa a mensagem `"O sinal SIGINT foi capturado..."`
3. `SIG_DFL`, que é o default handler para o signal `SIGINT` (simplesmente aborta a execução do processo)
4. o endereço do handler `ger_nova` é copiado para o ponteiro `ger_antiga`, de modo que a segunda e quarta execução de handlers são iguais (o código que consta no roteiro guarda o default handler `SIG_DFL` no ponteiro, mas o código fonte fornecido no Moodle guarda o `ger_nova`).

# Item (f)




A chamada `wait3` é uma interface antiga e deprecada que hoje em dia foi substituída interface mais moderna `waitpid`; ela serve para capturar uma mudança de estado em qualquer um dos filhos de um processo. As diferenças principais entre ela e a chamada `wait` são que:

1. `wait3` por padrão espera pela terminação de um processo filho (pode ser alterado por meio do uso de flags)
2. `wait3` pode ser non-blocking por meio do uso de flags

Se eliminarmos a linha com a chamada `wait3`, o valor de `pid` fica inicializado como lixo de memória (provavelmente com um valor maior que 0), de modo que o programa fica preso em um loop infinito no sighandler.