Insper

19 - Sinais II

Igor Montagner

Parte 1 - Capturando sinais

Apesar de muitos recursos mostrarem o uso da chamada `signal` para a captura de sinais, ela é considerada obsoleta e o recomendado é usar `sigaction`, que é um pouco mais complicada de usar mas permite maior flexibilidade ao definir o comportamento do processo.

O exemplo abaixo cria um `struct sigaction` e o seta para executar um handler quando o processo receber `SIGINT` (Ctrl+C).

```
void sig_handler(int num) {
    // faz algo aqui
}

....
struct sigaction s;
s.sa_handler = sig_handler; // aqui vai a função a ser executada
sigemptyset(&s.sa_mask);
s.sa_flags = 0;

sigaction(SIGINT, &s, NULL);
```

Exercício: Usando como exemplo o código acima, modifique o arquivo *sinal1.c* para que o programa só termine após apertar Ctrl+C três vezes. Você pode usar `exit` para sair na terceira vez. Não se esqueça de consultar `man sigaction` para verificar quais `include`s deve ser usados.

Para resetar o comportamento padrão de um sinal atribua a `s.sa_handler` a constante *SIG_DFL* em `s.sa_handler` e chame novamente `sigaction`.

Exercício: Restaure o comportamento original no segundo *Ctrl+C*, fazendo com que o processo realmente termine com o sinal.

Parte 2 - sinais e concorrência

Nesta parte vamos trabalhar com o arquivo `sinais-concorrentes.c`.

Exercício: Leia o conteúdo do arquivo acima e complete as partes faltantes.

Exercício: Teste sua implementação enviando sinais `SIGINT` e `SIGTERM` para seu processo. Os resultados foram os esperados?

Vamos agora examinar o que acontece quando trabalhamos com vários sinais sendo recebidos ao mesmo tempo.

Exercício: Rode o programa e execute a seguinte sequência de comandos. Você precisará de dois terminais abertos e enviará sinais usando o comando `kill`.

1. Envie o sinal SIGINT para o programa. O quê foi printado?

2. Envie o sinal SIGTERM para o programa. O quê foi printado?

3. Envie de novo SIGINT. Foi printado algo? Por quê?

Exercício: Assumindo que cada função roda do começo ao fim sem interrupçãom, os valores da variável `status` foram os esperados? Se não, como você explica o ocorrido?

Importante: Valide sua resposta com o professor antes de prosseguir. Se quiser, poderá esperar pela correção do exercício no fim da aula.

Parte 3 - bloqueio de sinais

A principal vantagem de usarmos `sigaction` é que esta chamada permite configurar sinais a serem **bloqueados** durante a execução da função `sa_handler`. Ou seja, se um sinal bloqueado for recebido durante sua execução ele é colocado "em espera" até que `sa_handler` acabe de rodar!

Exercício: Bloquear sinais evita o problema detectado na parte anterior?

Exercício: Quais sinais deverão ser bloqueados durante a execução do handler `SIGINT`? E durante a execução do handler `SIGTERM`?

Exercício: O campo `sa_mask` permite bloquear sinais enquanto os handlers executam. Consulte `man sigsetops` para ver como preenchê-lo.

Exercício: Modifique `sinais-concorrentes.c` para que *SIGTERM* seja bloqueado enquanto o handler de *SIGINT* roda. Repita então o código acima e veja que não há mais conflito na variável global compartilhada.

Exercício: O que fizemos não permite que `SIGINT` seja interrompido por um `SIGTERM`, mas permite que um `SIGTERM` seja interrompido por um `SIGINT`! Corrija esta situação.

Exercícios para praticar

Exercício: Modifique `sinal1.c` para que, ao ser colocado em background usando Ctrl+Z (SIGTSTP), imprima uma mensagem antes de parar de executar.

Dicas:

- Você precisa retornar o comportamento padrão do sinal depois de dar o print.
- Pesquise como usar `raise` para (re)enviar um sinal para o próprio processo.

Exercício: Complete o programa acima com uma outra função que imprime a mensagem *Continuando!* quando o programa voltar a rodar (sinal `SIGCONT`).

Uma parte importante de sinais em sistemas POSIX é que, ao interromper um processo, eles podem cancelar operações que estavam ocorrendo. Em especial, chamadas de sistema que deixam um processo bloqueado (como `wait` e `sleep`) ou que fazem operações de entrada/saída (como `read` e `write`).

Exercício: compile e rode o programa `sleep_longo.c`. O quê foi mostrado na tela?

Exercício: Rode novamente o programa. Abra um novo terminal e envie um sinal `SIGTERM` para este processo. O quê é mostrado na tela? Você consegue interpretar este resultado?

Exercício: Como checamos que `sleep` realmente parou o processo por todo o tempo?

Exercício: Modifique o programa para que ele chame `sleep` tanto quanto for necessário para que o processo durma o tempo especificado. Salve este arquivo como `sleep_longo_while.c`.

Exercício: Troque o código de `sleep_longo.c` para ignorar o sinal `SIGTERM`. O programa agora funciona como esperado? Por que?