■■ Tema 3: sandbox (Process Monitoring & Signals)

O objetivo é executar uma função (`f`) em um processo filho e monitorá-la. O pai deve determinar se a função é "boa" (termina com `exit(0)`) ou "ruim" (termina com sinal, timeout, ou exit code != 0).

Funções-Chave: `fork`, `waitpid`, `exit` (ou `_exit`), `alarm`, `sigaction`, `kill`, `strsignal`, `errno`.

Revisão: `fork` e `waitpid`

- **Objetivo:** Obter o `status` de um filho.
- **Explicação:** `waitpid(pid_t pid, int *status, int options)` é mais preciso que `wait`. Usar `waitpid(pid, &status, 0)` espera *especificamente* pelo `pid` dado.
- **Dica:** Crie um `fork()`. O filho chama `exit(42)`. O pai chama `waitpid()` e armazena o `status`.

Analisando o Status (Foco: `WIFEXITED`, `WEXITSTATUS`)

- **Objetivo:** Verificar uma terminação normal.
- **Dica:** Após `waitpid`, use `if (WIFEXITED(status))`. Se for verdade, o filho terminou *normalmente*. Em seguida, use `int code = WEXITSTATUS(status);` para obter o código de saída (que seria `42` no exercício 1).
- **Teste:** Faça o filho sair com `0` e `1`. Verifique se o pai consegue ler esses valores.

Analisando o Status (Foco: `WIFSIGNALED`, `WTERMSIG`)

- **Objetivo:** Verificar uma terminação por sinal (ex: SEGFAULT).
- **Dica:** Use `if (WIFSIGNALED(status))`. Se for verdade, o filho foi morto por um sinal. Use `int sig = WTERMSIG(status);` para obter o número do sinal (ex: `SIGSEGV`).
- **Teste:** Crie um filho que causa um SEGFAULT (`int *p = NULL; *p = 1;`). Verifique se o pai detecta `WIFSIGNALED`.

Formatando Sinais (Foco: `strsignal`)

- **Objetivo:** Converter o número do sinal em uma string legível.
- **Explicação:** `strsignal(int sig)` (de `<string.h>`) retorna uma string como "Segmentation fault".
- **Dica:** Combine com o Exercício 3. Se `WIFSIGNALED`, obtenha o `sig` e imprima `strsignal(sig)`.
- **Teste:** Verifique se o pai imprime "Segmentation fault" (ou similar).

A Bomba Relógio (Foco: `alarm`)

- **Objetivo:** Enviar um sinal `SIGALRM` após `N` segundos.
- **Explicação:** `alarm(unsigned int seconds)` agenda um `SIGALRM` para o processo atual. A ação padrão do `SIGALRM` é terminar o processo.
- **Dica:** Escreva um programa que chama `alarm(2)` e depois entra em `while(1);`.
- **Teste:** O programa deve rodar por 2 segundos e depois ser terminado com "Alarm clock".

Desarmando a Bomba (Foco: `sigaction`)

- **Objetivo:** Capturar `SIGALRM` em vez de morrer.
- **Explicação:** `sigaction(int signum, const struct sigaction *act, ...)` permite definir um "handler" (uma função) para um sinal.
- **Dica:** Defina um handler simples `static void handler(int sig) { (void)sig; }`. Na `main`, configure uma `struct sigaction` para usar este handler para `SIGALRM`. Chame `alarm(2)` e entre em `while(1);`.
- **Teste:** O programa agora deve rodar para sempre. O `SIGALRM` é enviado, mas o handler o captura e não faz nada, então o processo continua.

O Problema do `waitpid` Bloqueado

- **Objetivo:** Entender o desafio do timeout.

- **Dica:** No `sandbox`, o pai `fork`a e chama `waitpid(pid, ...)` (que bloqueia). Se o filho entrar em loop infinito, o pai fica bloqueado para sempre. Se o pai chamar `alarm(timeout)` *antes* de `waitpid`, o `SIGALRM` atingirá o *pai* e o matará (a menos que seja tratado).
- **Provocação:** E se o pai tratar o `SIGALRM` (Exercício 6)? A `sigaction` interrompe a `waitpid`?

A Interrupção de Chamada de Sistema (Foco: `EINTR`)

- **Objetivo:** Usar o `SIGALRM` para interromper `waitpid`.
- **Dica:** Quando um handler de sinal é executado, chamadas de sistema bloqueantes (como `waitpid`) são interrompidas. Elas retornam `-1` e definem a variável global `errno` como `EINTR`.
- **Dica de Implementação:** No pai, configure o handler (Exercício 6), chame `alarm(timeout)`, e então `waitpid()`. Se `waitpid()` retornar `-1` e `errno == EINTR`, significa que o *seu* alarme disparou. O filho ainda está rodando (timeout!).

Lidando com o Timeout (Foco: `kill`)

- **Objetivo:** Matar o filho que estourou o tempo.
- **Explicação:** `kill(pid_t pid, int sig)` envia um sinal para um processo específico.

Dica: No bloco `if (errno == EINTR)` do Exercício 8, o pai agora sabe que o filho `pid` excedeu o tempo. O pai deve:

1. `kill(pid, SIGKILL)` (para forçar a morte do filho).
2. `waitpid(pid, NULL, 0)` (para recolher o filho agora morto e evitar zumbis).
3. Imprimir "timed out" e retornar 0 (função ruim).

Exercício Composto: `sandbox` Completo

- **Objetivo:** Juntar tudo.

Dica:

1. `fork()`.

2. **Filho:** Chama `alarm(timeout)` (como um seguro caso o pai falhe) e então `f()`. Se `f()` retornar, o filho chama `_exit(0)` (sucesso).

3. **Pai:** Configura `sigaction` para `SIGALRM`. Chama `alarm(timeout)`.

4. **Pai:** Chama `waitpid()`.

Pai: Verifica o retorno de `waitpid`:

- Se `r < 0 && errno == EINTR`: É timeout. Chame `kill` e `waitpid` de novo (Exercício 9). Retorne `0` (ruim).
- Se `r > 0` (filho terminou): Cancele o alarme (`alarm(0)`).
- Verifique o `status` (Exercícios 2 e 3).
- `WIFEXITED` com `0`: Retorne `1` (bom).
- `WIFEXITED` com `!= 0`: Retorne `0` (ruim).
- `WIFSIGNALED`: Retorne `0` (ruim).

- **Teste:** Use todas as funções de teste (`ok_f`, `inf_f`, `segfault_f`, etc.) fornecidas no arquivo para verificar todos os caminhos.
