Insper

POSIX Lab

Sistemas Hardware-Software

Entrega: **16/06**

A segunda parte do curso apresenta (e exercita) conceitos importantes de sistemas operacionais, focando em sistemas compatíveis com o padrão POSIX. Neste lab iremos melhorar um framework de testes usando as chamadas de sistemas vistas em aula para deixá-lo mais robusto a erros e mais rápido.

Parte 0 - testes unitários em *C*

Nesta seção vamos revisar a implementação do framework de testes que já usamos na atividade do vetor dinâmico. O arquivo *mintest/macros.h* contém *macros* usada dentro das funções de teste para dar prints e checar se uma condição que deveria ser verdade.

O arquivo *mintest/runner.h* contém uma função `main` que chama cada um dos testes e mostra seus resultados. Um arquivo de testes segue o modelo abaixo.

```
#include "mintest/macros.h"

int test1() {
    test_printf("Hello! %d %f\n", 3, 3.14);
    return 0;
}

int test2() {
    test_assert(1 == 0, "This always fails!");
    printf("This never runs!\n");
    test_assert(1 == 1, "Neither this.");
    return 0;
}

int test3() {
    test_printf("<-- Name of the function before the printf!\n");
    test_assert(1 == 1, "This always succeeds");
    return 0;
}

test_list = { TEST(test1), TEST(test2), TEST(test3) };

#include "mintest/runner.h"
```

Basta compilar o arquivo acima, rodar e você deve obter a seguinte saída.

```
Running 3 tests:
====================

test1: Hello! 3 3.140000
test1: [PASS]
test2: [FAIL] This always fails! in example.c:9
test3: <-- Name of the function before the printf!
test3: [PASS]


====================
2/3 tests passed
```

Esta implementação, apesar de funcional, tem diversos problemas:

1. falhas em um teste podem afetam a execução de outros (um teste pode, acidentalmente, apontar para dados de outros testes)
2. um teste que dê erro impede a execução dos testes seguintes
3. um teste que entre em loop infinito impede a execução dos testes seguintes
4. não é possível rodar somente um teste específico.

Seria interessante, também, que nosso programa mostrasse uma mensagem de confirmação quando o usuário pressiona `Ctrl+C` para terminar o programa e que fosse possível rodar os testes em paralelo para que eles terminassem mais rápido.

Importante: estamos supondo aqui que a ordem de execução dos testes não importa.

Parte 1 - requisitos de implementação

Este laboratório exercitará os seguintes conceitos vistos em sala de aula:

1. Criação e gerenciamento de processos
2. Sinais;
3. Tratamento de arquivos e redirecionamento de saída;
4. Compilação de programas e *C* avançado;

Nosso objetivo será resolver os problemas listados na seção anterior. Mais especificamente, iremos implementar as seguintes novas *features*:

Execução de um teste específico

Seu programa permite passar como argumento na linha de comando o nome de um teste a ser rodado. Somente este teste será executado.

```
$> teste_vetor test_creation_destroy
```

Quando não for passado um nome de teste como argumento todos os testes deverão ser executados.

Execução dos testes de maneira isolada

Atualmente um erro em um teste impede a execução dos testes seguintes. É possível isolar a execução de cada teste fazendo com que cada teste rode em um processo separado. Você pode adotar a seguinte convenção:

1. o processo do teste retorna 0 se todos `test_assert` passarem.
2. o processo do teste retorna 1 se um dos `test_assert` falhar.
3. se o processo do teste der erro mostrar `[ERRO]` como status.

Relatório de erros dos testes que falharem

Quando um teste retornar `[ERRO]`, mostrar ao lado uma descrição do erro ocorrido.

Execução dos testes em paralelo

Além de executarem em cada processo, seus testes deverão ser executados em paralelo. Isto melhorará significativamente o tempo de execução total do conjunto de testes, mas exige um gerenciamento de processos criados um pouco mais complexo.

Tempo limite de execução para os testes

Seu programa deverá impor um tempo limite de *2s* para a execução de cada teste. Se um teste passar deste limite deverá ser cancelado e uma mensagem de erro deve ser apresentada. O status do teste cancelado deverá `[TIME]`.

Este tempo limite pode ser implementado tanto no pai como nos filhos. Pense na forma mais inteligente e sucinta de fazê-lo.

Organizar saída padrão e de erros para cada teste

Com diversos testes rodando ao mesmo tempo é possível que a saída padrão mostre seus resultados todos misturados. Cada processo deverá ter sua saída redirecionada para um arquivo temporário (ou armazenado em memória) e a saída de cada teste deve ser mostrada apenas após o seu fim.

Da mesma maneira, a saída de erros dos processos filhos também deverá ser redirecionada.

Confirmação de saída ao apertar Ctrl+C

Ao apertar `Ctrl+C` o programa mostra a seguinte mensagem:

```
Você deseja mesmo sair [s/n]?
```

Se o usuário digitar `s` o programa parará todos os testes em execução. Cada teste parado fica com status `[STOP]` e é mostrado o resumo de quantos testes passaram e quantos não passaram.

Contagem de tempo

Ao finalizar a execução de um teste o tempo que ele levou para rodar aparece ao lado de seu status:

```
[PASS] (0.5s)
[FAIL] (0.1s)
```

Parte 3 - Implementação

Você precisará modificar os arquivos dentro da pasta *mintest* para implementar os itens acima. Não deverão ser necessárias mudanças no *teste_exemplo.c*.

Importante: em diversos momentos será necessário decidir se uma funcionalidade é implementada no processo original (que cria todos os processos filhos) ou nos filhos (que efetivamente rodam as funções de teste).

Parte 4 - Entrega

Você deverá entregar:

1. seus arquivos modificados *macros.h* e *runner.h*
2. um ou mais arquivos contendo testes escritos por você mesmo. Faça pelo menos um teste de cada tipo abaixo
   1. cause erro e termine com falha de segmentação
   2. cause erro e termine com divisão por zero
   3. fique em loop infinito
   4. faça muito trabalho, mas eventualmente acabe (sem usar `sleep`)
   5. tenha *assert*s que falham e passem no mesmo teste
   6. tenha testes que façam muitos prints
   7. tenha testes que sejam rápidos e testes que sejam lentos (pode usar `sleep` para simular).
3. um arquivo *README.txt* explicando sua implementação.

Sua implementação deverá ser compatível com a original, mas oferecer as extensões e melhorias descritas neste enunciado.

Conceitos

A rubrica adotada neste projeto é incremental. Para obter um conceito é necessário realizar todas as tarefas de todos os conceitos anteriores.

Conceito I

Não entregou ou entregou o exemplo disponibilizado sem modificações

Conceito D

Implementou algum item, mas não todos, do conceito *C*. Os arquivos de testes entregues não contém testes para todos os casos descritos.

Conceito C

- **Feature**: Execução dos testes de maneira isolada
- **Feature**: Execução dos testes em paralelo
- **Feature**: Execução de um teste específico

Conceito B

- **Feature**: Tempo limite de execução para os testes
- **Feature**: Relatório de erros dos testes que falharem
- **Feature**: Organizar saída padrão e de erros para cada teste

Conceitos B+, A, A+

A partir do conceito **B** cada uma das seguintes características adiciona um ponto na nota:

- **Feature**: Contagem de tempo
- **Feature**: Confirmação de saída ao apertar Ctrl+C
- **Feature**: A biblioteca disponibiliza a função `void set_time_limit(float f)` para que cada teste possa modificar seu tempo limite. Essa função só pode ser chamada uma vez por função e deverá retornar um erro caso seja chamada mais de uma vez.

A partir do conceito **C** cada uma das seguintes funcionalidades adiciona 0.25 na nota final:

- os status dos testes são impressos em cores.
- (adicional ao de cima) se a saída de errros do `main` for redirecionada para um arquivo imprima tudo sem cores.