INE5412 – Sistemas Operacionais I
Profs. Giovani Gracioli e Márcio Bastos Castro
2023/01

Projeto de um Sistema Operacional em Nível de Usuário
Trabalho 2: Gerenciamento Básico de Threads

1. Descrição

Este trabalho consiste na implementação de algumas funções básicas para gerenciamento de threads dentro do SO em nível de usuário, usando como base a classe CPU implementada no trabalho anterior para fazer as trocas de contexto entre as threads.

O gerenciamento das threads do SO será realizado por métodos implementados dentro da classe **Thread**. Para isso, dois novos arquivos são necessários: **thread.h** e **thread.cc**. O arquivo thread.h conterá a declaração da classe **Thread**, seus atributos e métodos são disponibilizados pelo professor. O arquivo thread.cc terá a implementação dos métodos e a declaração dos atributos estáticos da classe Thread e deverá ser criado pela dupla.

Os seguintes métodos devem ser implementados dentro da classe Thread:

- **template<typename ... Tn>**
  Thread(void (* entry)(Tn ...), Tn ... an);

  É o construtor da classe. Cria uma Thread passando um ponteiro para a função a ser executada e os parâmetros passados para a função, que podem variar (por isso o uso de variadic template). O construtor também deve criar o contexto da Thread. Devido ao template, este método deve ser implementado no mesmo arquivo thread.h (entretanto, **fora** do escopo da classe).

- **static Thread * running();**

  Retorna um ponteiro para a Thread que está em execução.

- **static int switch_context(Thread * prev, Thread * next);**

  Método para trocar o contexto entre duas thread, a anterior (prev) e a próxima (next). Deve encapsular a chamada para a troca de contexto realizada pela classe CPU. Valor de retorno é negativo se houve erro, ou zero.

- **void thread_exit (int exit_code);**

  Termina a Thread e “limpa” a memória associada a ela. exit_code é o código de término devolvido pela tarefa (ignorar agora, vai ser usado nos próximos trabalhos). Quando a thread encerra, o controle deve retornar à main.

- **int id();**

Retorna o ID da Thread.

Além desses métodos obrigatórios da classe Thread, cada grupo pode criar métodos que achar necessário para a conclusão do trabalho.

A classe Thread possui os seguintes atributos:

- **int _id:** contém o ID da Thread.
- **Context * volatile _context:** contém o contexto da Thread. Note que há um typedef dentro da classe Thread para redefinicação do Context (typedef CPU::Context Context).
- **static Thread * _running:** ponteiro para a Thread que estiver em execução. Toda vez que uma nova Thread for executada, este ponteiro deve ser atualizado.

Cada grupo pode adicionar outros atributos necessários para a solução do trabalho.

Além do gerenciamento de Threads, este trabalho irá adicionar outras duas classes ao SO. A primeira será a classe **System** (arquivo system.h disponibilizado), que terá inicialmente um único método **init()** que fará a inicialização de variáveis e/ou estruturas de dados internas do SO. Para este trabalho, o método **init()** deve apenas desativar o buffer de saída padrão usado pelo cout (**setvbuf (stdout, 0, _IONBF, 0)**). Isso evita condições de corrida que podem ocorrer no buffer quando threads são usadas. System::init() deve ser sempre chamado no início da função main(). O arquivo system.cc deve ser criado pelo grupo e conter a implementação de System::init().

A outra classe adicionada ao sistema é a classe Debug (arquivos debug.h e debug.cc disponibilizados pelo professor). Essa classe adiciona 4 níveis de debug ao sistema: error (ERR), warning (WRN), info (INF) e trace (TRC). Os níveis de debug devem ser declarados e habilitados no arquivo traits.h, como no exemplo abaixo que trace está habilitado:

```
template<> struct Traits<Debug>: public Traits<void>
{
   static const bool error   = false;
   static const bool warning = false;
   static const bool info    = false;
   static const bool trace   = true;
};
```

O trace habilitado conjuntamente com a habilitação do debug para cada classe do sistema, torna possível a impressão de diferentes mensagens para depuração, como no exemplo abaixo:

```
template<typename T>
struct Traits {
   static const bool debugged = false;
};

template<> struct Traits<System> : public Traits<void>
{
```

```
    static const bool debugged = true;
};
```

Debugged, no exemplo, está habilitado para a classe System. Desta forma, dentro dos métodos da classe System, podemos usar os níveis de depuração da seguinte forma:

db<System>(TRC) << "System::init() chamado\n";

Assim, como debugged na classe Traits<System> está habilitado e trace está habilitado para Debug (também no traits), a mensagem acima será impressa na tela.

Para habilitar os níveis de depuração para uma classe deve-se fazer:

db<CLASSE>(NÍVEL) << “mensagem\n”;

Sendo CLASSE substituída pelo nome da classe e NÍVEL por ERR, WRN, INF ou TRC. Deve-se também habilitar debugged no Traits da respectiva classe e o nível de depuração dentro do Traits da classe Debug.

Está incluído neste trabalho fazer todas as alterações necessários dentro do arquivo traits.h (declaração das novas classes, declaração do traits para as novas classes, declaração do traits de Debug como acima e a variável booleana debugged para cada uma das novas classes). O uso de Debug corretamente no código será avaliado.

Os seguintes arquivos também são disponibilizados pelo professor: **main.cc** (contém a função main) e **main_class.h e main_class.cc** (exemplo de uso das Threads através de uma aplicação exemplo) A saída esperada para esse exemplo é a seguinte:

```
main: inicio
    Ping: inicio
    Ping0
        Pong: inicio
        Pong0
    Ping1
        Pong1
    Ping2
        Pong2
    Ping3
        Pong3
    Ping4
        Pong4
    Ping5
        Pong5
    Ping6
        Pong6
    Ping7
        Pong7
    Ping8
        Pong8
```

```
   Ping9
      Pong9
   Ping: fim
      Pong: fim
main: fim
```

É também sugerido que os grupos implementem uma outra aplicação de teste para ter maior prática e entendimento na criação das Threads.

Lembrem-se de sempre usar __BEGIN_API e __END_API no início e no final dos arquivos .h e .cc. A falta disso poderá acarretar em penalidades devido a avaliação automática de vazamento de memória. Faz parte do trabalho pensar em todas o uso das funções implementadas, não apenas os cenários descritos/apresentados neste documento e no código de aplicação exemplo.

2. Formato de Entrega

Todos os arquivos utilizados na implementação do trabalho devem ser entregues em um único arquivo .zip ou .tar.gz na atividade do moodle. Deve ser anexado um arquivo Makefile para compilar o código.

3. Data de Entrega:

Conforme dia e horário na tarefa do Moodle.

4. Avaliação

A avaliação se dará em 4 fases

1. Avaliação de compilação: compilar o código enviado. Caso haja erros de compilação, a nota do trabalho será automaticamente zerada.

2. Avaliação de execução: para validar que a solução executa corretamente sem falhas de segmentação. Caso haja falhas de segmentação, a nota é zerada. Será também avaliado o uso de variáveis globais (-5 pontos) e vazamentos de memória (-20%).

3. Avaliação da organização do código: busca-se nesta fase avaliar a organização do código orientado a objetos. Deve-se usar classes e objetos e não estilo de programação baseado em procedimentos (como na linguagem C).

4. A quarta fase consiste na apresentação do trabalho em dia e horário agendado pelo professor. Durante as apresentações, o professor irá avaliar o **conhecimento individual dos alunos sobre os conteúdos teóricos e práticos vistos em aula e sobre a solução adotada no trabalho**. A nota atribuída à cada aluno *i* no trabalho ($NotaTrabalho_i$) será calculada da seguinte forma, onde A_i é a nota referente à apresentação do aluno *i* e *S* é a nota atribuída à solução do trabalho:

$$NotaTrabalho_i = \frac{A_i x S}{10}$$

Plágio não será tolerado em nenhuma hipótese ao longo dos trabalhos, acarretando em nota 0 a todos os envolvidos.