INE5412 – Sistemas Operacionais I
Profs. Giovani Gracioli e Márcio Bastos Castro
2022/01

Projeto de um Sistema Operacional em Nível de Usuário
Trabalho 3: Thread Main, Thread Despachante, yield e Inicialização do Sistema

1. Descrição

Este trabalho tem os seguintes objetivos:

1. Criação de uma Thread main, que será a primeira e principal Thread a ser criada no sistema, bem como o conceito de estados das Threads (RUNNING, READY e FINISHING);
2. Criação de uma Thread despachante (*dispatcher*) que será responsável pela escolha e controle de qual Thread do usuário executará;
3. Criação do método yield() dentro da classe Thread que permite que a Thread que o chame volte a fila de prontos, devolvendo assim o processador para a Thread Dispatcher;
4. Mudanças na inicialização do sistema para suportar as Threads main e dispatcher;
5. Criação da fila de tarefas prontas (*ready queue*) para suportar as decisões tomadas pelo dispatcher e o gerenciamento do estado das tarefas;

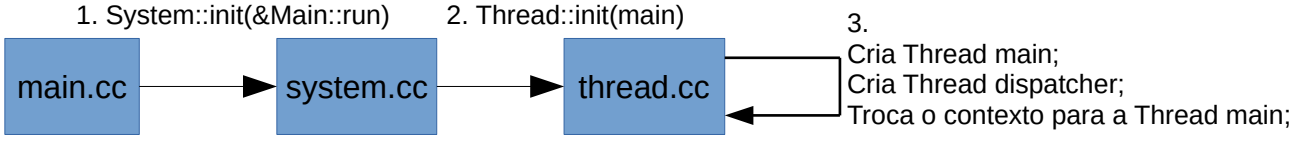
As atividades a serem realizadas para atingir os objetivos são detalhadas a seguir.

2. Inicialização do Sistema

Durante a fase de inicialização de um SO tradicional, existe a execução de diversas atividades para configurar o processador e as estruturas internas do SO. Uma das atividades de inicialização realizada é a inicialização do gerenciamento de Threads.

Neste trabalho iremos alterar o método init() da classe System para suportar a criação da Thread Main. O método init() deve receber um ponteiro para uma função que não tem valor de retorno e que receba um ponteiro void como parâmetro: **void System::init(void (*main)(void *))**.

Desta forma, a classe main passa a não ser mais criada dentro da função main (arquivo main.cc). A função main() do programa simplesmente irá chamar init() da classe System passando o endereço da função/método principal do programa a ser executada, algo parecido com: System::init(&MAIN). Assim, conseguimos simular o procedimento de criação da função principal em nosso SO. A Figura abaixo demonstra os passos a serem realizados na inicialização do sistema, sendo o passo 1 a chamada para System::init(). **Você deverá alterar o método System::init() conforme a descrição.**



3. Thread Main e Estados das Threads

Em SOs tradicionais que suportam a execução de Threads, a função main() nos programas de usuário é transformada em uma Thread, sendo esta a principal e primeira Thread a ser criada no sistema. Em nosso SO de nível de usuário, criaremos uma Thread para executar a função principal (função run da classe Main no exemplo liberado neste trabalho), como demonstrado na Figura acima. O método init() da classe System irá chamar o método init() da classe Thread. Este método irá então criar a Thread main, passando como parâmetro para o contexto o ponteiro da função main recebido como argumento.

O método init() da classe Thread irá também criar a Thread dispatcher (detalhada na próxima seção) e depois irá trocar o contexto para a Thread main. Deste ponto em diante, a função main do programa do usuário passa a ser executada (método run da classe Main no exemplo liberado). A classe Thread possui uma fila ordenada que deverá conter todas as Threads prontas para serem executadas (atributo static Ready_Queue _ready).

Cada Thread possui seu próprio elemento da fila chamado de _link (atributo Ready_Queue::Element _link), que deve ser utilizado para fazer a inserção de Threads na fila de prontos, toda vez que uma nova Thread é criada: _link(PONTEIRO PARA A THREAD, inteiro). O valor inteiro é usado para ordenar a fila de prontos, como se fosse uma prioridade da Thread. Neste trabalho, usaremos o instante de criação da Thread como critério para ordenação. Desta forma, o construtor da Thread deve inicializar _link() passando o instante atual como segundo parâmetro (inteiro). Sugestão é utilizar as bibliotecas ctime e chrono do C++ para retornar o timestamp atual:

_link(PONTEIRO PARA THREAD, (std::chrono::duration_cast<std::chrono::microseconds>
(std::chrono::high_resolution_clock::now().time_since_epoch()).count()))

Além disso, este trabalho adiciona o conceito de estado para as Threads. Cada Thread pode estar em um dos seguintes estados:

- RUNNING: em execução. Somente uma Thread pode estar em estado RUNNING em um determinado instante de tempo. O ponteiro _running da classe Thread deve apontar para esta Thread.
- READY: estado pronto. Threads em estado READY devem estar dentro da fila de prontos (_ready).
- FINISHING: estado de finalização. O método thread_exit, quando chamado, deve alterar o estado da thread para FINISHING, informando que a mesma está sendo finalizada.

Você deverá implementar o método Thread::init conforme a descrição, garantir que a Thread main seja executada e alterar a implementação do construtor da Thread para inicializar _link e adicionar as Threads na fila de prontos.

4. Thread Dispatcher

A Thread despachante (ou dispatcher) será chamada para escolher uma outra thread pronta para que seja executada. A Thread despachante sempre estará pronta para ser executada. O código da Thread despachante deve seguir mais ou menos o modelo abaixo:

```
void Thread::dispatcher() {
imprima informação usando o debug em nível TRC

enquanto existir thread do usuário:
        escolha uma próxima thread a ser executada
        atualiza o status da própria thread dispatacher para READY e reinsire a mesma em _ready
        atualiza o ponteiro _running para apontar para a próxima thread a ser executada
        atualiza o estado da próxima thread a ser executada
        troca o contexto entre as duas threads
        testa se o estado da próxima thread é FINISHNING e caso afirmativo a remova de _ready

muda o estado da thread dispatcher para FINISHING
remova a thread dispatcher da fila de prontos
troque o contexto da thread dispatcher para main
}
```

5. Método yield

O método yield() da classe Thread é chamada para ceder a vez do processador, ou seja, a thread que chama o método cede o processador para outra Thread do sistema. É a base para o correto funcionamento de Threads cooperativas.

O seguinte algoritmo pode ser de base para a implementação do método yield() neste trabalho:

```
void Thread::yield() {
imprima informação usando o debug em nível TRC

escolha uma próxima thread a ser executada

atualiza a prioridade da tarefa que estava sendo executada (aquela que chamou yield) com o timestamp atual, a fim de reinserí-la na fila de prontos atualizada (cuide de casos especiais, como estado ser FINISHING ou Thread main que não devem ter suas prioridades alteradas)

reinsira a thread que estava executando na fila de prontos

atualiza o ponteiro _running

atualiza o estado da próxima thread a ser executada

troque o contexto entre as threads
}
```

Dica para atualizar a prioridade com o timestamp atual:

```
int now =
  std::chrono::duration_cast<std::chrono::microseconds>(std::chrono::high_resolution_clock::now().time_since_epoch()).count();
thread->_link.rank(now);
```

6. Arquivos Disponibilizados

Os seguintes arquivos foram disponibilizados neste trabalho:

- thread.h: atualização do arquivo thread.h contendo a descrição dos novos métodos a serem implementados neste trabalho, bem como novos atributos necessários.
- list.h: implementação de uma lista ordenada em C++.
- main.cc: arquivo da função main().
- main_class.cc e main_class.h: implementação de uma aplicação exemplo, contendo também a função run() que serve como main para o exemplo.

A saída esperada do programa exemplo é:

```
main: inicio
  Pang: inicio
  Pang: 0
    Peng: inicio
    Peng: 0
      Ping: inicio
      Ping: 0
        Pong: inicio
        Pong: 0
          Pung: inicio
          Pung: 0
  Pang: 1
    Peng: 1
      Ping: 1
        Pong: 1
          Pung: 1
  Pang: 2
    Peng: 2
      Ping: 2
        Pong: 2
          Pung: 2
  Pang: 3
    Peng: 3
      Ping: 3
        Pong: 3
          Pung: 3
  Pang: 4
    Peng: 4
      Ping: 4
```

```
            Pong: 4
                Pung: 4
    Pang: 5
        Peng: 5
            Ping: 5
                Pong: 5
                    Pung: 5
    Pang: 6
        Peng: 6
            Ping: 6
                Pong: 6
                    Pung: 6
    Pang: 7
        Peng: 7
            Ping: 7
                Pong: 7
                    Pung: 7
    Pang: 8
        Peng: 8
            Ping: 8
                Pong: 8
                    Pung: 8
    Pang: 9
        Peng: 9
            Ping: 9
                Pong: 9
                    Pung: 9
    Pang: fim
        Peng: fim
            Ping: fim
                Pong: fim
                    Pung: fim
main: fim
```

Os novos atributos da classe Thread são:

- static Thread _main;
- static CPU::Context _main_context;
- static Thread _dispatcher;
- static Ready_Queue _ready;
- Ready_Queue::Element _link;
- volatile State _state;

State é uma enum declarada na própria classe Thread, contendo os 3 estados possíveis:

```
enum State {
    RUNNING,
    READY,
    FINISHING
};
```

Além disso, deve-se reutilizar e alterar os arquivos do trabalho passado seguindo as orientações apresentadas neste documento: system.h, system.cc, thread.cc.

7. 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.

8. Data de Entrega

Conforme tarefa do moodle.

9. Avaliação

A avaliação se dará em 3 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).