Atividade Prática 7

Paula Raissa Silva

1 Pthreads e Concorrência

Um dos conceitos mais importantes relacionados a sistemas operacionais é denominado processo. De modo geral, um **processo** é uma abstração de um programa em execução. Tal programa possui um espaço de endereçamento e, em sistemas tradicionais, apenas um **thread** (segmento ou fluxo de controle) de execução. Além disso, o processo contém toda informação relacionada ao seu contexto de execução, por exemplo, o contador de programa, apontador de pilha e demais registradores. Dito de outra maneira, é um programa com com sua função principal, denominada main, sendo executado sequencialmente, instrução por instrução.

No entanto, em alguns sistemas é possível criar mais de um thread no mesmo processo, isto é, no **mesmo espaço de endereçamento**. Nesse caso, mais de um fluxo de execução ocorre dentro do mesmo processo. Diante disso, é importante destacar algumas aplicações:

- Tratar atividades que ocorrem "simultaneamente";
- Dividir a aplicação em tarefas que acessam recursos compartilhados;
- Reduzir o tamanho de uma aplicação, uma vez que threads ocupam menos espaço em relação aos processos;
- São mais fáceis de criar e destruir;
- A sobreposição de tarefas pode acelerar a aplicação;
- Possibilitam paralelismo real em sistemas multicore.

É importante destacar que **threads** e **processos** são conceitos diferentes. Como dito anteriormente, o **processo** é basicamente um agrupador de recursos (código e dados) e possui uma identidade, enquanto os **threads** são criados no contexto de um processo e compartilham o mesmo espaço de endereçamento. À vista disso, *threads* não são independentes como os processos. Pois, embora compartilhem o mesmo espaço de endereçamento dentro de um processo, cada *thread* possui os mecanismos para gerenciar seu contexto de execução. Assim, *threads* possuem seu próprio contador de programa, apontador de pilha e registradores.

Os *threads* criados ocupam a CPU do mesmo modo que o processo criador, e também são escalonadas pelo próprio processo. Nesse contexto, quando uma aplicação *multithread* é executada, esses *threads* podem estar em qualquer um dos seguintes estados: em execução, bloqueado (aguardando), pronto para ser executado ou concluído (finalizado). Isso é ilustrado na Figura 1.

1.1 Pthreads

Para padronizar a utilização de *threads* em diversos sistemas o IEEE estabeleceu o padrão POSIX threads, ou *Pthreads*. Esse padrão define mais de 60 funções para criar e gerenciar threads. Tais funções são definidos na biblioteca **pthreads.h**. Além disso, a biblioteca define estruturas de dados e atributos para configurar os threads. De modo geral, esses atributos são passados como argumentos para os parâmetros das funções, por exemplo:

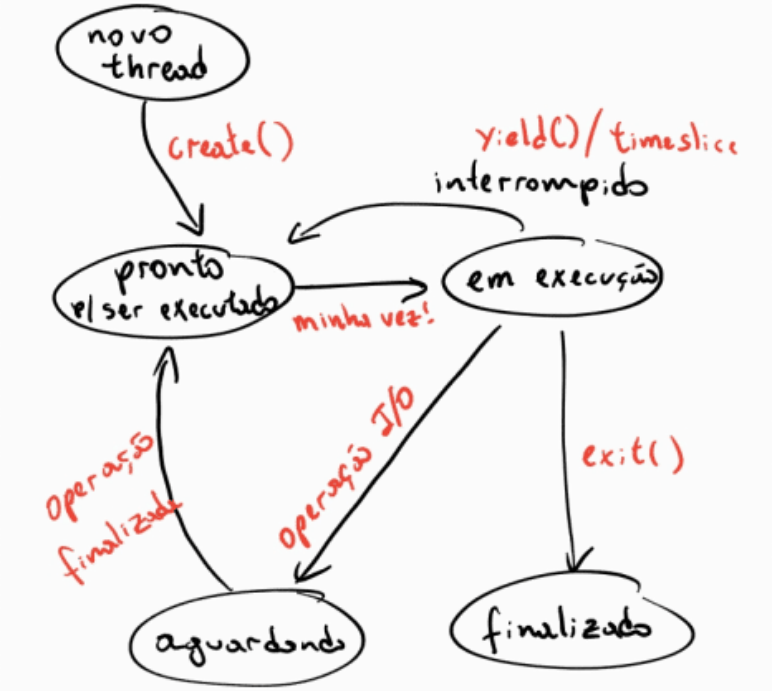

Figura 1: Estados de um thread

- pthread_t: Handle para pthread, isto é, um valor que permite identificar o thread;
- pthread_attr_t: Atributos para configuração de thread.

Esses recursos são utilizados nas principais funções para criação e gerenciamento de *threads*. Tais funções são apresentadas a seguir. Cabe ressaltar que as funções que retornam valor, tem como padrão o inteiro 0, indicando sucesso. Assim, qualquer inteiro diferente de 0 é um código de erro.

1.2 Criação de Threads

A função **pthread_create** é utilizada para inicializar um thread. Para isso, a função recebe como argumento o endereço de um dado *pthread_t* que será inicializado. Além disso, o endereço da função que será executada é informado no parâmetro *start_routine*. Tal função tem como retorno um valor void * e recebe apenas um argumento a partir do seu parâmetro void *. Esse argumento é passado para função *start_routine* especificando o último parâmetro da função *pthread_create*, denominado *arg*. Cabe ressaltar que essa função pode receber um atributo para configuração do thread. No entanto, esse argumento pode ser NULL, indicando que o thread será configurada conforme o padrão.

```
int pthread_create (pthread_t *thread, pthread_attr_t *attr,
    void *(*start_routine) (void *), void *arg);
```

Um thread pode ser finalizado a partir da função **pthread_exit**. Essa função recebe como argumento um endereço que é utilizado para armazenar o valor de retorno do thread.

```
void pthread_exit (void *retval);
```

1.3 Gestão de Threads

Considerando um ambiente em que mais de um thread está sendo executado, pode ser necessário, em algum momento, aguardar a finalização de um procedimento. Isso pode ser realizado com a função **pthread_join**. A função **pthread_join** recebe como argumentos a estrutura de controle do thread, do tipo **pthread_t**, que será finalizado e o endereço de um apontador **(void **)** para o valor de retorno do thread.

```
int pthread_join (pthread_t thread, void **thread_return);
```

No contexto de execução do thread é possível obter o identificador do thread, denominado **thread ID**. Isso é realizado com a chamada **pthread_self** que tem como retorno um valor do tipo **pthread_t**.

```
pthread_t pthread_self (void);
```

Como dito anteriormente, um thread pode estar em diversos estados. Em sua execução o thread pode indicar para gerenciador que o mesmo pode ser bloqueado. Isso é realizado pela função **sched_yield**. Nesse caso, um outro thread entrará em execução.

```
int sched_yield (void);
```

1.4 Sincronização de Threads

Mutexes são usados para evitar inconsistências de dados devido a condições de corrida. Uma condição de corrida geralmente ocorre quando dois ou mais threads precisam executar operações na mesma área de memória, mas os resultados dos cálculos dependem da ordem em que essas operações são executadas. Mutexes são usados para serializar recursos compartilhados. Sempre que um recurso global é acessado por mais de um thread, o recurso deve ter um Mutex associado a ele. Pode-se aplicar um mutex para proteger um segmento de memória ("região crítica") de outros threads. Mutexes podem ser aplicados aos threads em um único processo e não funcionam entre processos como os semáforos.

Um mutex (Mutual exclusion) é um lock que apenas pode estar na posse de um thread de cada vez, garantindo exclusão mútua. Os restantes threads que tentem aceder ao lock ficam bloqueados até que este esteja libertado.

1.5 Variáveis de Condição

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t *cond,
    pthread_condattr_t *cond attr);

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
```

Os mutexs permitem prevenir acessos simultâneos a variáveis partilhadas. No entanto, por vezes o uso de mutexs pode ser bastante ineficiente. Se pretendermos realizar uma dada tarefa apenas quando uma dada variável tome um certo valor, temos que consultar sucessivamente a variável até que esta tenha o valor pretendido. Em lugar de testar exaustivamente uma variável, o ideal era adormecer o thread enquanto a condição pretendida não sucede. As variáveis de condição permitem adormecer threads até que uma dada condição suceda.

Ao contrário dos semáforos, as variáveis de condição não têm contadores. Se um thread A sinalizar uma variável de condição antes de um outro threas B estar à espera, o sinal perde-se. O thread B ao sincronizar mais tarde nessa variável, deverá ficar à espera que outro thread volte a sinalizar a variável de condição.

A uma variável de condição está sempre associado um mutex. Isto acontece de modo a garantir que entre o testar de uma dada condição e o activar da espera sobre uma variável de condição, nenhum outro thread sinaliza a variável de condição, o que poderia originar a perca desse mesmo sinal. *pthread_cond_wait*() bloqueia o thread na variável de condição *cond*. De um modo atómico, essa função liberta o mutex e bloqueia na variável de condição *cond* até que esta seja sinalizada. Isto requer, obviamente, que se obtenha o *lock* sobre o mutex antes de invocar a função. Quando a variável é sinalizada e thread é acordado, *pthread_cond_wait*() readquire o *lock* no mutex (tal como se executasse *pthread_lock_mutex*()) antes de regressar à execução.

2 Resolução dos Exercícios

Threads compartilham espaços de endereço, o que implica que as modificações nos dados compartilhados, como variáveis globais, devem ser sincronizadas; caso contrário, haverá um comportamento incorreto do programa. Observe que o código do exercício 1 cria 2 threads com a chamada *pthread_ create* e passa *inc* como ponto de partida de sua execução. *inc* modifica a variável global compartilhada em um loop for de 10000 iterações. Portanto, se os dois threads incrementarem o valor de compartilhado em 10000 cada, o programa deve gerar 20000 como resultado final. Ao executar o código, o resultado será algum número inteiro aleatório, mas não 20000. Esse comportamento é geralmente classificado como uma condição de corrida, o que implica que determinados threads acessam a variável compartilhada sem consultar uns aos outros, ou seja, sem sincronização. Frequentemente quando a execução do loop intercala, a inconsistência é alcançada nos acessos e armazenamentos da variável compartilhada e um é produzido um cálculo incorreto.

A seção de código onde vários threads modificam o mesmo objeto na memória é chamada de seção crítica. Geralmente, a seção crítica deve ser protegida com algum tipo de bloqueio que forçaria outros encadeamentos a esperar até que o encadeamento atual termine a execução e garanta que todos obtenham o valor incrementado correto. Mutex é um dos tipos de bloqueio que podem ser utilizados para proteger a seção crítica como no loop for em inc (Ver exercício 2).

No exercício 2, utilizaremos a biblioteca de threads POSIX e seu tipo integrado **pthread_mutex_t**. A variável do tipo **pthread_mutex_t** é geralmente declarada como duração de armazenamento static. Mutex deve ser inicializado apenas uma vez antes de ser usado. Quando o mutex é declarado como *static*, para inicializá-lo deve-se usar a macro **PTHREAD_MUTEX_INITIALIZER** . Uma vez que o mutex é inicializado, os threads podem usar as funções *pthread_ mutex_ lock* e *pthread_ mutex_ unlock* correspondentemente. *pthread_ mutex_ lock* bloqueia o objeto mutex passado como o único argumento. Se o mutex já estiver bloqueado, o thread de chamada será bloqueado até que o mutex se torne disponível. *pthread_ mutex_ unlock* deve ser chamado para desbloquear o mutex. Se houver threads esperando no mesmo mutex, a política de agendamento determina qual delas obtém o bloqueio do objeto. Finalmente, chamamos *pthread_ join* em cada uma das duas threads e imprimimos o inteiro - *count*, que neste caso deve ter o valor correto armazenado.