INE5410 – Programação Concorrente

Unidade II – Fundamentos de Programação Concorrente

Prof. Frank Siqueira
frank.siqueira@ufsc.br

Conteúdo

- Processos
- Threads
- **Exclusão Mútua**
- Semáforos
- Deadlocks

Controle de Concorrência

- Conforme vimos anteriormente, quando duas ou mais threads acessam dados compartilhados em memória concorrentemente podem surgir inconsistências no funcionamento do programa
- Para evitar que isso ocorra, temos que utilizar mecanismos de controle de concorrência, que impedem o acesso concorrente aos dados

Controle de Concorrência

- Região Crítica
  - É a parte de um programa onde a memória compartilhada está sendo acessada
- Mecanismos de Controle de Concorrência
  - Limitam o acesso concorrente a regiões críticas
  - Garantem o isolamento entre processos e threads concorrentes
  - Buscam evitar inconsistências nos dados
  - Threads que não são autorizadas a acessar a região crítica são suspensas, deixando de ser executadas até que chegue sua vez de entrar na região crítica

Controle de Concorrência

- Exemplo de Região Crítica

```
...
int contador_global = 0;    // Contador que será incrementado pelas threads

// Função com o código das threads
void *incrementa_contador(void *arg) {
        // Obtém o número de vezes que o contador será incrementado
        int num_incr = *((int *)arg);

        // Loop que incrementa o contador
        for (int i = 0; I < num_incr; i++) {
                contador_global++;     // Região crítica!!!
                printf("Thread %d incrementou o contador pela %dª vez\n",
                        pthread_self(), i);
        }

        pthread_exit(NULL);
}
...
```

Controle de Concorrência

- Exemplo de Região Crítica
  - Se criarmos várias threads *incrementa_contador* teremos uma condição de corrida que poderá tornar o contador inconsistente
  - Para evitar que isso ocorra, temos que proteger a região crítica do código para não permitir que mais de uma thread a execute simultaneamente
  - Isso limitará a concorrência, pois quando uma thread estiver dentro da região crítica e perder o direito de usar a CPU, as demais threads não poderão entrar na região crítica
  - As demais threads aguardarão em uma fila até que chegue sua vez de entrar na região crítica

Controle de Concorrência

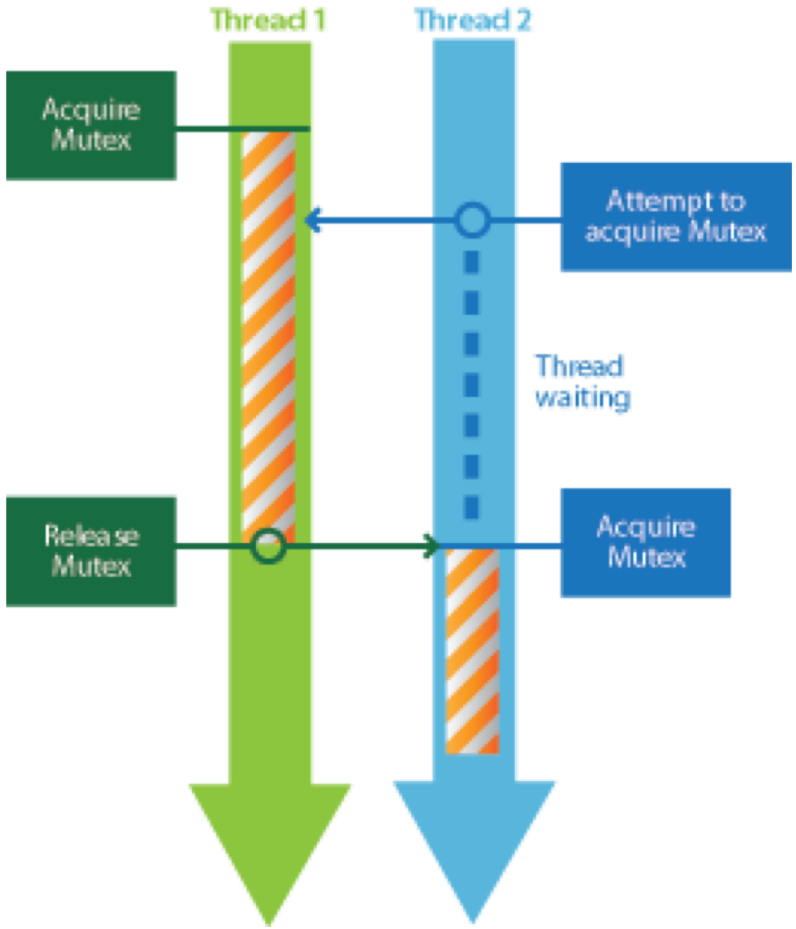
- Mecanismos de Controle de Concorrência:
  - Mutex (ou Lock): mecanismo usado para impedir que mais de uma tarefa entre simultaneamente em uma região crítica, criando uma fila de acesso
  - Semáforo: limita o número de tarefas que acessam simultaneamente uma região crítica, criando filas de acesso se este número for excedido
  - Monitor: delimita regiões críticas do código, impedindo o acesso concorrente a elas

Exclusão Mútua

- Para garantir a exclusão mútua no acesso a uma região crítica podemos usar o mecanismo chamado Mutex ou Lock
  - Só 1 thread é autorizada a usar o Mutex por vez
- O Mutex é um tipo abstrato de dados que contém:
  - Um estado, que indica se está livre ou travado
  - Uma fila de threads que aguardam para usá-lo
- Desempenho: o Mutex deve ser usado com cuidado, pois ele limita a concorrência entre threads, tornando o programa mais lento

Exclusão Mútua

- Uso de Mutex para proteger uma região crítica



Exclusão Mútua

- Operação de travamento de um Mutex

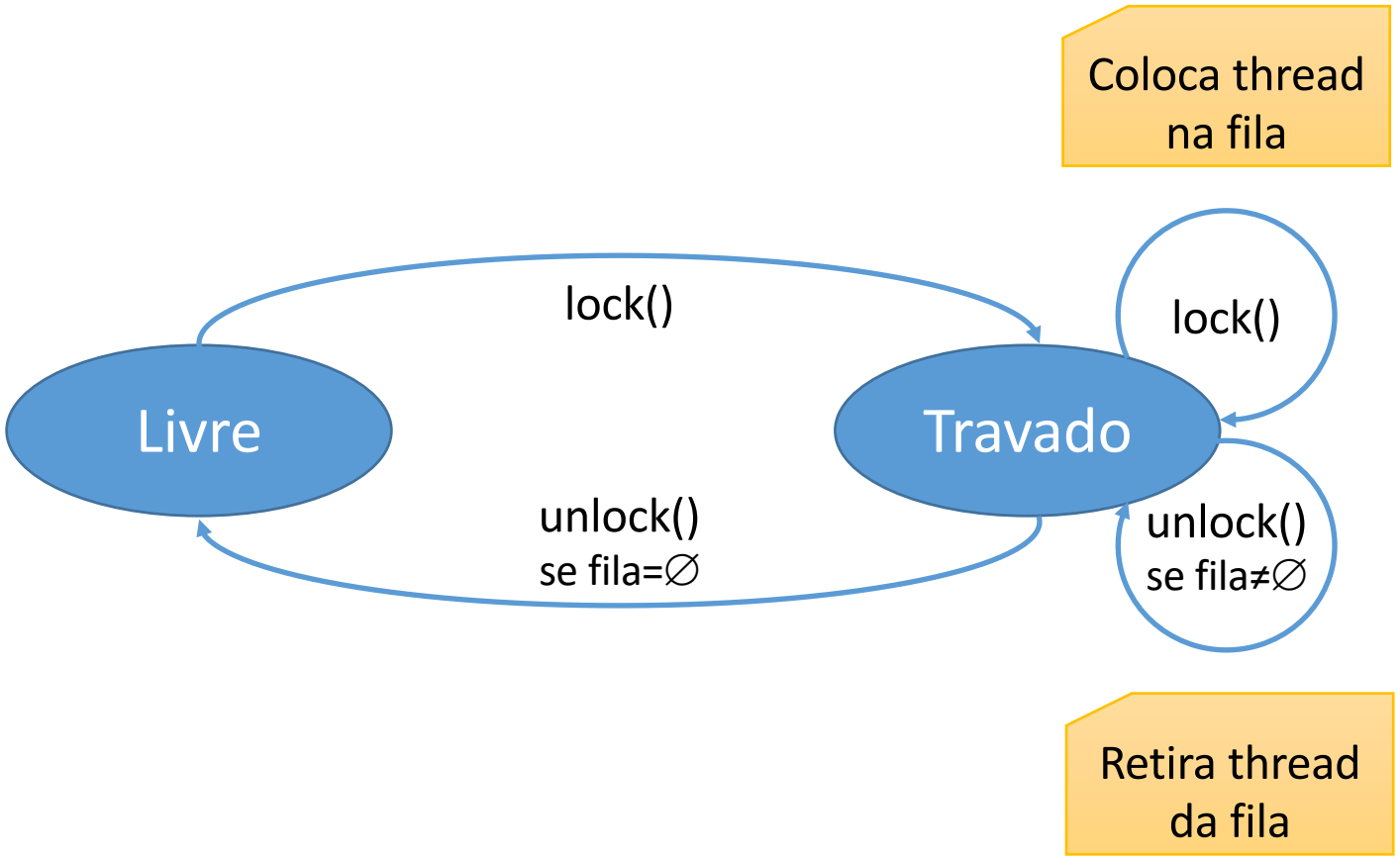
```
lock(mutex) :
   SE o mutex está livre ENTÃO
         Marca o mutex como travado
   SENÃO
         Bloqueia a thread que fez a chamada
         Insere a thread no fim da fila do mutex
```

- Operação de liberação de um Mutex

```
unlock(mutex) :
   SE a fila do mutex está vazia ENTÃO
         Marca o mutex como livre
   SENÃO
         Libera a thread do início da fila do mutex
```

Exclusão Mútua

- Estados de um Mutex



Exclusão Mútua

- Mutex de POSIX Threads
  - Um mutex deve ser declarado como variável global:
    pthread_mutex_t mutex;
  - O mutex deve ser inicializado antes de ser usado:
    pthread_mutex_init(&mutex, attrs);
  - Ao entrar na região crítica, o mutex deve ser travado:
    pthread_mutex_lock(&mutex);
  - Tenta travar o mutex, sem bloquear a thread:
    pthread_mutex_trylock(&mutex);
  - Ao sair da região crítica, o mutex deve ser liberado:
    pthread_mutex_unlock(&mutex);
  - Se não for mais necessário, o mutex deve ser destruído:
    pthread_mutex_destroy(&mutex);

Exclusão Mútua

- Exemplo de Região Crítica com Mutex

```
...
int contador_global = 0;     // Contador que será incrementado pelas threads
pthread_mutex_t mutex;  // Mutex para proteger o acesso ao contador

// Função com o código das threads
void *incrementa_com_mutex(void *arg) {
   // Obtém o número de vezes que o contador será incrementado
   int num_incr = *((int *)arg);

   // Loop que incrementa o contador
   for (int i = 0; I < num_incr; i++) {
            pthread_mutex_lock(&mutex);        // Trava o mutex
            contador_global++;                 // Região crítica!
            pthread_mutex_unlock(&mutex);      // Libera o mutex
            printf("Thread %d incrementou o contador\n", pthread_self());
   }

   pthread_exit(NULL);
}
...
```

Exclusão Mútua

- Exemplo de Região Crítica com Mutex (cont.)

```
...
int main(int argc, char ** argv) {
   int num_threads = argc >= 2? atoi(argv[1]) : 10; // Núm. de threads (def. 10)
   pthread_t threads[num_threads];   // Identificadores das threads
   int num_incr= 1000*num_threads; // Número de incrementos do contador

   pthread_mutex_init(&mutex, NULL); // Inicializa o mutex destravado

   for(int i=0; i < NUM_THREADS; i++)     // Loop que cria as threads
            pthread_create(&threads[i], NULL,
                incrementa_com_mutex, (void *) &num_incr);

   for(int i=0; i < NUM_THREADS; i++)    // Aguarda o término das threads
            pthread_join(threads[i], NULL);

   pthread_mutex_destroy(&mutex);   // Destrói o mutex
   return 0;
}
```

Exclusão Mútua

- Considerações sobre o exemplo:
  - Se a varíavel global for usada em outra parte do código, devemos também protegê-la com o Mutex
  - Cada thread deve permanecer o menor tempo possível com o Mutex bloqueado
  - Para cada varíavel global que precisa ser protegida, devemos criar um Mutex, pois podemos permitir que enquanto uma thread altera uma variável global, outra thread altere outra variável