Computação Paralela

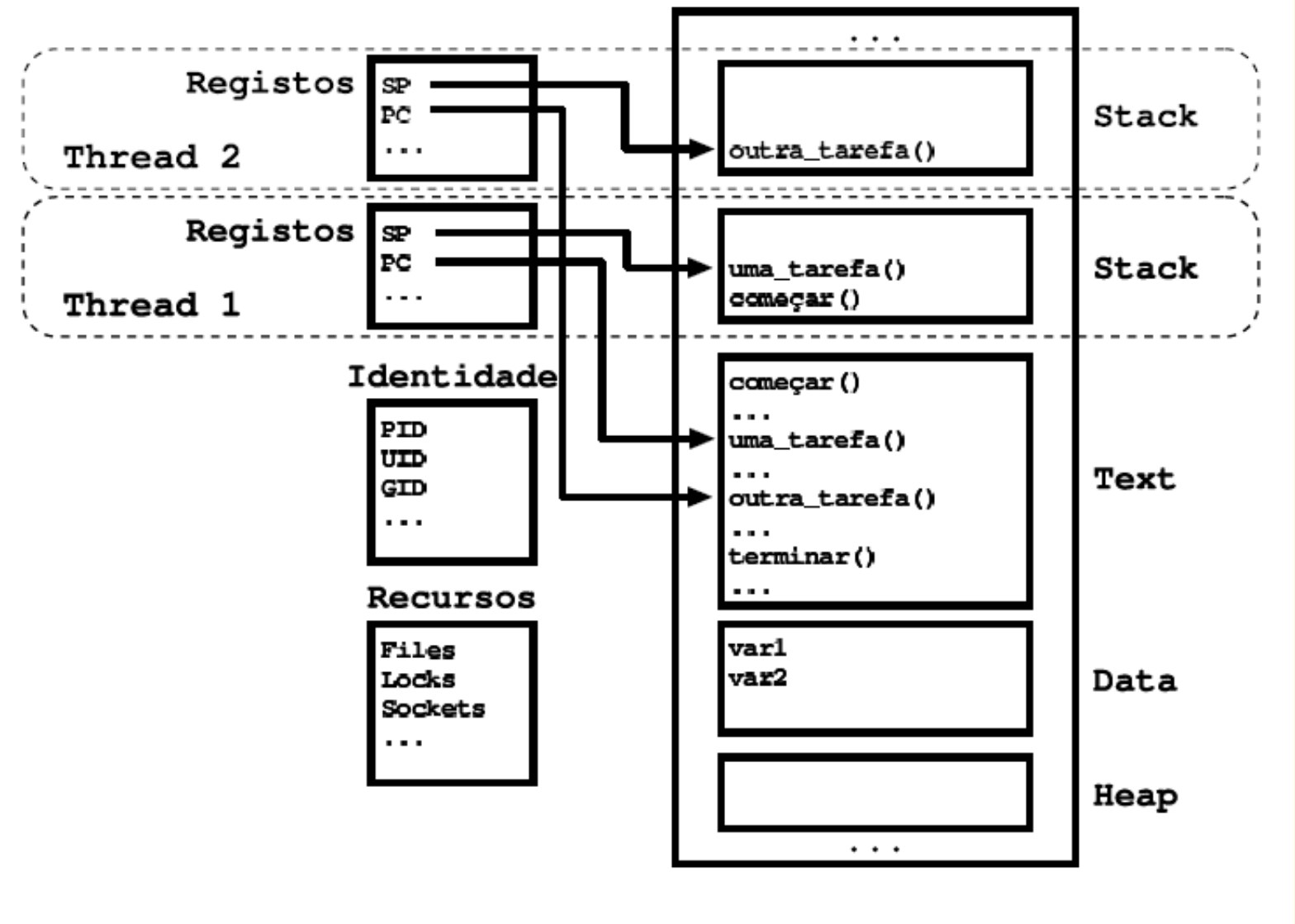
Rafael Sachetto Oliveira
sachetto@ufsj.edu.br

2 de maio de 2012

Concorrência ou paralelismo potencial

- Concorrência ou paralelismo potencial é quando um programa possui tarefas que podem ser executadas em qualquer ordem sem alterar o resultado final.
- Uma razão óbvia para explorar concorrência é conseguir reduzir o tempo de execução dos programas em máquinas multiprocessadas.
- Existem, no entanto, outras situações em que o paralelismo potencial de um programa pode ser explorado: operações de I/O, ocorrência assíncrona de eventos, escalonamento de tarefas em tempo-real, etc.

Concorrência com processos multithread



Processos multithread

```
Processo = Threads +  Recursos
```

- Uma thread representa um fluxo de execução sequencial dentro do processo.
- A cada thread está associado uma pilha de execução e um conjunto de registradores de contexto, tais como o *stack pointer* e o *program counter*.
- Os recursos restantes do processo são compartilhados pelas threads.

Multithreading: vantagens e inconvenientes

- (+) Facilita a estruturação dos programas. Vários programas podem ser estruturados em múltiplas unidades de execução.
- (+) Elimina múltiplos espaços de endereçamento, reduzindo a carga de memória dos sistema.
- (+) O compartilhamento do espaço de endereçamento permite a utilização de mecanismos de sincronização mais eficientes e trocas de contexto mais rápidas
- (-) O compartilhamento transparente de recursos exige do programador cuidados redobrados de sincronização.

O modelo POSIX Threads (pthreads)

- Como um programa pode ser desenvolvido para executar múltiplos threads dentro de um processo?
- Para que isso seja possível, é necessário um modelo que suporte a criação e manipulação dessas threads. PThreas, Mach Threads e NT Threads são exemplos desse modelo.
- O modelo pthreads pertence à família POSIX (*Portable Operating System Interface* e define um conjunto de rotinas para a manipulação de threads.
- As definições da biblioteca pthreads encontram-se em 'pthread.h' e a sua implementação em 'libpthread.so'. Para compilar um programa com suporte a threads é necessários incluir o cabeçalho 'pthread.h' e compilá-lo com '-lpthread'

Criação de threads

- Quando se inicia um programa, uma thread é criada (**main thread**). Outras threads podem ser criadas usando a função:

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

- cria uma nova thread que inicia sua execução na função indicada por start_routine com o argumento indicado em arg. Em caso de sucesso, instancia th com o identificador da nova thread e retorna 0, senão retorna código de erro.

Criação de threads

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

- th é o identificador da nova thread.
- attr permite especificar como a thread deve interagir com o resto do programa.
- start_routine é a função inicial que a nova thread deve executar.
- arg é o argumento a ser passado para a função start_routine.

Junção de threads

- Assim como os processos, as vezes é necessário esperar que uma dada thread termine antes da execução terminar. Com processos, essa sincronização é conseguida pelas funções wait ou waitpid. Com thread a função é pthread_join.

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

- Suspende a execução até que a thread th termine.
- th é o identificador da thread a esperar.
- thread_return é o valor de retorno da thread th.

Terminar threads

- Por default, existem 2 formas de uma thread terminal:
  - A função que iniciou a thread retorna.
  - A função main retorna ou alguma thread chama a função exit. Nestes dois casos **todas** as threads terminam.
- Outro modo de uma thread terminar é este invocar diretamente a função pthread_exit

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

Joinable Threads x Detached Threads

- Uma thread pode estar em um dos seguintes estados: **joinable** ou **detached**. O estado de uma thread apenas condiciona como ela termina.
- Quando uma *joinable thread* termina, parte do seu estado é mantido pelo sistema (indentificador e pilha) até que uma outra thread chame pthread_join para obter seu valor de retorno. Só então os recursos da thread são totalmente liberados.
- Os recursos de uma *detached thread são totalmente liberados logo que ela termina*.
- Uma thread pode ser criada como *joinable* ou como *detached*. Por padrão, todas as thread são criadas como *joinable*.

Joinable Threads x Detached Threads

Mudar o estado de uma thread para *detached* :

```
int pthread_detach(pthread_t th);
```

- Retorna 0 se OK, valor positivo se erro;
- th é o identificador da thread que ficará *detached*

Obter o identificador da thread corrente:

```
pthread_t pthread_self();
```

- Retorna o identificador da thread corrente.

Integração Numérica com Threads

```
float somas[NTHREADS];

main() {
    pthread_t thread[NTHREADS];
    int i;
    float soma;
    for (i = 0; i < NTHREADS; i++) {
        pthread_create(&thread[i], NULL, integral, (void *)i);
    }
    soma = 0;
    for (i = 0; i < NTHREADS; i++) {
        pthread_join(thread[i], NULL);
        soma += somas[i];
    }
    printf("Area total= %f\n", H * (soma + (f(A) + f(B)) / 2));
}
void *integral(void *region_ptr) {
    int region = (int) region_ptr;
    ...
    somas[region] = soma_parcial;
    return NULL;
}
```

Sincronização e Regiões Críticas

- A principal causa da ocorrência de erros na programação de threads está relacionada com o fato dos dados serem todos compartilhados. Apesar de este ser um aspectos mais poderosos da utilização de threads, também pode ser um dos mais problemáticos.
- O problema existe quando duas ou mais threads tentam alterar as mesmas estruturas de dados (race conditions).
- Existem dois tipos principais de sincronização:
  - Mutexs: para situações de curta duração.
  - Variáveis de Condição: para situações em que o tempo de espera não é previsível (pode depender da ocorrência de um evento).

Mutexs

- Um mutex (MUTual EXclusion) é um lock que apenas pode estar na posse de uma thread de cada vez, garantindo exclusão mútua.

```
int pthread_mutex_init(pthread_mutex_t *mutex,
                       pthread_mutexattr_t *mutexattr);
```

- Inicia um mutex. Retorna 0 se OK, valor positivo se erro.
- mutex é a variável que representa o mutex.
- mutexattr permite especificar atributos do mutex. Se NULL o mutex é iniciado com os atributos por padrão.
- Outra foram de iniciar um mutex (se estaticamente alocado) com os atributos por padrão é a seguinte:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Operações sobre Mutexs

Obter o lock no mutex:

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
```

Liberaar o lock:

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Tenta obter o lock mas não bloqueia caso não seja possível:

```
int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Integral com mutex

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
float soma;
main() {
    pthread_t thread[NTHREADS];
    int i;
    soma = 0;
    for (i = 0; i < NTHREADS; i++)
        pthread_create(thread[i], NULL, integral, (void *)i);
    for (i = 0; i < NTHREADS; i++)
        pthread_join(thread[i], NULL);

    printf("Area total= %f\n", H * (soma + (f(B) - f(A)) / 2));
}

void *integral(void *region_ptr)  {
    ...
    pthread_mutex_lock(&mutex);
    soma += soma_parcial;
    pthread_mutex_unlock(&mutex);
    return NULL;
}
```

Fila de tarefas com mutex

```
pthread_mutex_t flag_mutex = PTHREAD_MUTEX_INITIALIZER;
int flag_is_set = FALSE;
void *thread_function (void *thread_arg) {
    while (TRUE) {
        pthread_mutex_lock(&flag_mutex);
        if (flag_is_set) {
            get_task();
            if (no_more_tasks())
                flag_is_set = FALSE;
            pthread_mutex_unlock(&flag_mutex);
            do_work();
        }
        else
            pthread_mutex_unlock(&flag_mutex);
    }
}
void new_task() {
    pthread_mutex_lock(&flag_mutex);
    put_task();
    flag_is_set = TRUE;
    pthread_mutex_unlock(&flag_mutex);
}
```

Fila de tarefas com mutex 2

```
pthread_mutex_t flag_mutex = PTHREAD_MUTEX_INITIALIZER;
int flag_is_set = FALSE;
void *thread_function (void *thread_arg) {
    while (TRUE) {
        while (flag_is_set == FALSE);
        pthread_mutex_lock(&flag_mutex);
        if (flag_is_set) {
            get_task();
            if (no_more_tasks())
                flag_is_set = FALSE;
            pthread_mutex_unlock(&flag_mutex);
            do_work();
        }
        else
            pthread_mutex_unlock(&flag_mutex);
    }
}
void new_task() {
    pthread_mutex_lock(&flag_mutex);
    put_task();
    flag_is_set = TRUE;
    pthread_mutex_unlock(&flag_mutex);
}
```

Variáveis de Condição

- Os mutexs permitem prevenir acessos simultâneos a variáveis compartilhadas. 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 tome o valor pretendido.
- Em lugar de testar exaustivamente uma variável, o ideal era adormecer o thread enquanto a condição pretendida não ocorre.
- As variáveis de condição permitem adormecer threads até que uma dada condição aconteça.

Variáveis de Condição

- Ao contrário dos semáforos, as variáveis de condição não têm contadores. Se um thread A sinalizar uma variável de condição antes de um outro thread B estar à espera, o sinal perde-se.

```
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t cond_attr);
```

inicia uma variável de condição. Retorna 0 se OK, valor positivo se erro.

Variáveis de Condição

```
int pthread_cond_init(pthread_cond_t *cond,
                      pthread_condattr_t*cond_attr);
```

- cond representa a variável de condição.
- cond_attr permite especificar atributos da variável de condição.
- Pode-se inicializar utilizando:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

Sinalizar uma variável de condição

```
int pthread_cond_signal(pthread_cond_t *cond);
```

- acorda um dos threads bloqueados na variável cond. Caso existam vários threads bloqueados, apenas um é acordado.

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- acorda todos os threads que possam estar bloqueados na variável cond.
- Em ambos os casos, se nenhum thread estiver bloqueado na variável especificada, os sinais são perdidos.
- Ambas as funções retornam 0 se OK, valor positivo se erro.

Bloquear numa Variável de Condição

- A uma variável de condição está sempre associado um mutex. Isto acontece para garantir que entre teste de uma dada condição e a ativação da espera sobre uma variável de condição, nenhum outro thread sinaliza a variável de condição, o que poderia originar a perda do sinal.

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

- bloqueia o thread na variável de condição cond.
- de um modo atómico libera 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.

Fila de tarefas variáveis de condição

```
pthread_mutex_t flag_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t flag_cond = PTHREAD_COND_INITIALIZER;
int flag_is_set = FALSE;
void *thread_function (void *thread_arg) {
    while (TRUE) {
        pthread_mutex_lock(&flag_mutex);
        while (flag_is_set == FALSE)
            pthread_cond_wait(&flag_cond, &flag_mutex);
        get_task();
        if (no_more_tasks())
            flag_is_set = FALSE;
        pthread_mutex_unlock(&flag_mutex);
        do_work();
    }
}
void new_task() {
    pthread_mutex_lock(&flag_mutex);
    put_task();
    flag_is_set = TRUE;
    pthread_cond_signal(&flag_cond);
    pthread_mutex_unlock(&flag_mutex);
}
```

Processos x Threads

- Todos os threads num programa executam o mesmo executável. Um processo filho pode executar um programa diferente se invocar a função exec.
- Um thread mal-programado pode corromper as estruturas de dados de todos os outros threads. Um processo não consegue fazer isso porque o seu espaço de endereçamento é privado.
- A memória a ser copiada na criação de um novo processo acrescenta um peso maior ao sistema do que na criação de um novo thread. No entanto, a cópia é retardada e apenas é efetuada se a memória for alterada. Isto diminui, de certa forma, o custo nos casos em que o processo filho apenas lê dados.
- A utilização de processos é mais indicada em problemas de granularidade grossa/média, enquanto que os threads são indicados para problemas de granularidade fina.
- Compartilhar dados entre threads é trivial porque eles compartilham a mesma memória. Compartilhar dados entre processos requer a utilização de uma das técnicas de comunicação entre processos (IPC). Apesar de mais difícil, isto torna a utilização de múltiplos processos menos suscetível de erros de concorrência.