

Computação Paralela

LCC/LERSI

Programação com Threads

Computação Paralela 2006/2007

Ricardo Rocha DCC-FCUP

Concorrência ou Paralelismo Potencial

- Concorrência ou paralelismo potencial diz-se quando um programa possui tarefas que podem ser executadas em qualquer ordem sem alterar o resultado final.

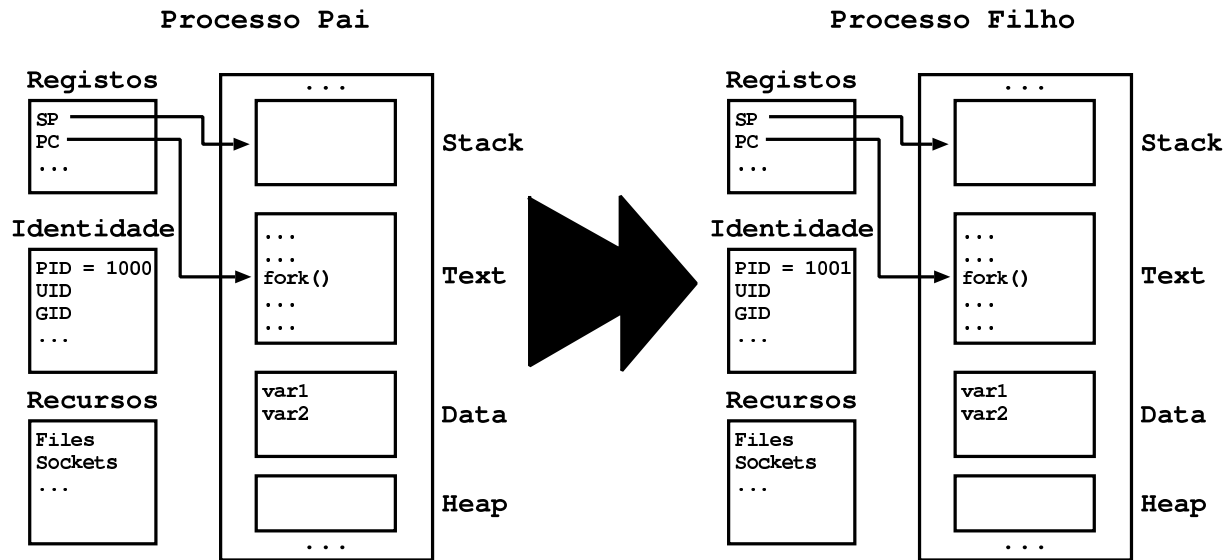
<code>começar()</code>	<code>uma_tarefa()</code>	<code>outra_tarefa()</code>	<code>terminar()</code>
------------------------	---------------------------	-----------------------------	-------------------------

<code>começar()</code>	<code>outra_tarefa()</code>	<code>uma_tarefa()</code>	<code>terminar()</code>
------------------------	-----------------------------	---------------------------	-------------------------

<code>começar()</code>	<code>uma_</code>	<code>outra_</code>	<code>tarefa()</code>	<code>tarefa()</code>	<code>terminar()</code>
------------------------	-------------------	---------------------	-----------------------	-----------------------	-------------------------

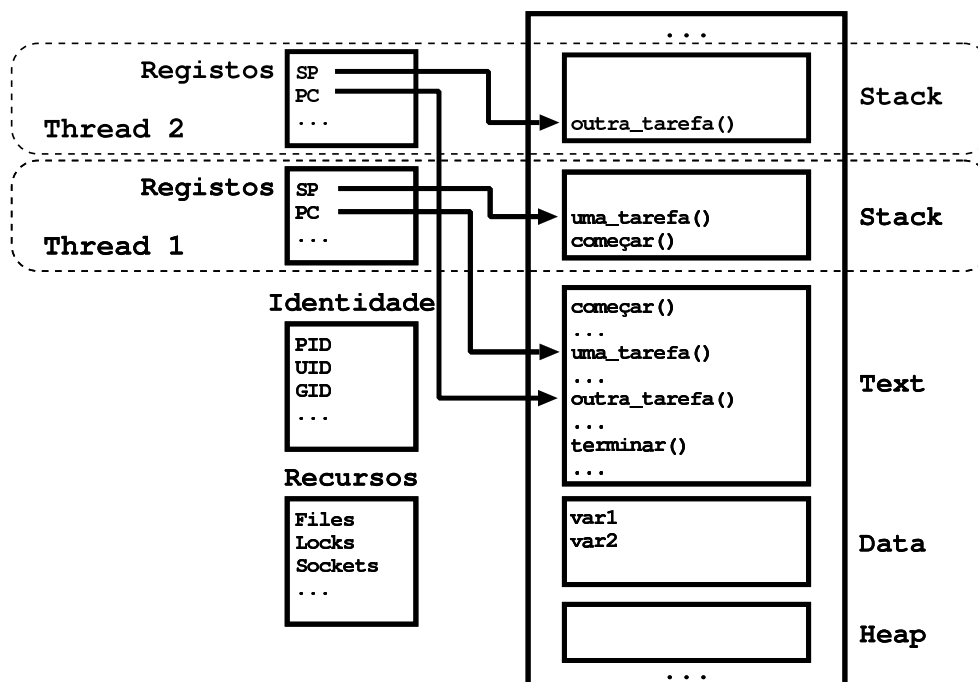
- Uma razão óbvia para explorar concorrência é conseguir reduzir o tempo de execução dos programas em máquinas multiprocessador.
- Existem, no entanto, outras situações em que o paralelismo potencial de um programa pode ser explorado: operações de I/O, ocorrência assíncrona de eventos, escalonamento de tarefas em tempo-real, etc.

Concorrência com Processos



2

Concorrência com Processos Multithreaded



3

Processos Multithreaded

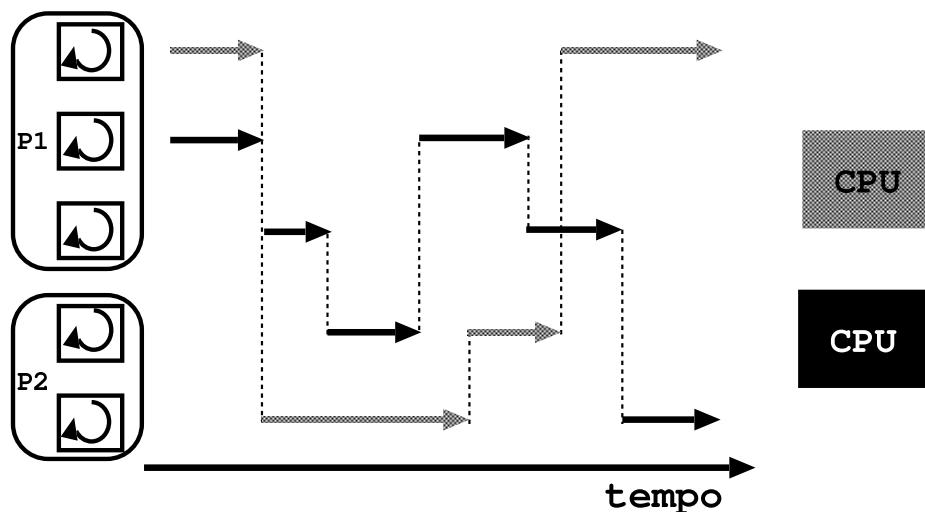
Processo = Conjunto de *Threads* + Conjunto de Recursos

- Um *thread* representa um fluxo de execução sequencial dentro do processo.
- A cada *thread* está associado uma pilha de execução (*stack*) e um conjunto de registos de contexto, tais como o *stack pointer* e o *program counter*.
- Os restantes recursos do processo são partilhados pelo conjunto dos *threads*: espaço de endereçamento, ficheiros abertos, identidade do utilizador, etc.

4

Execução de Processos Multithreaded

- Todos os *threads* de um processo podem ser executados concorrentemente e em diferentes processadores, caso existam.



5

Multithreading: Vantagens e Inconvenientes

- (+) Facilita a estruturação dos programas. Grande parte dos programas são intrinsecamente estruturados em múltiplas unidades de execução.
- (+) Elimina espaços de endereçamento múltiplos, permitindo reduzir a carga de memória do sistema e melhorar o tempo de resposta dos programas.
- (+) A partilha do espaço de endereçamento permite utilizar mecanismos de sincronização mais eficientes e trocas de contexto entre *threads* mais rápidas do que entre processos.
- (–) A partilha transparente de recursos exige do programador cuidados redobrados de sincronização.

O Modelo POSIX Threads (Pthreads)

- Como é que um programa pode ser desenhado para executar múltiplos *threads* dentro de um processo?
- É necessário um modelo que suporte a criação e manipulação de tarefas cuja execução possa ser intercalada ou executada em paralelo. Pthreads, Mach Threads e NT Threads são exemplos desses modelos.
- O modelo Pthreads pertence à família POSIX (*Portable Operating System Interface*) e define um conjunto de rotinas (biblioteca) para manipulação de *threads*.
- As definições da biblioteca Pthreads encontram-se em 'pthread.h' e a sua implementação em 'libpthread.so'. Sendo assim, para compilar um programa com *threads* é necessário incluir o cabeçalho '#include <pthread.h>' no início do programa e compilá-lo com a opção '-lpthread'.

Criação de Threads

- Quando se inicia um programa, um *thread* é desde logo criado (**main thread**). Outros *threads* podem ser criados através de:

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

- `pthread_create()` cria um novo *thread* que inicia a sua execução na função indicada por `start_routine` com o argumento indicado em `arg`. Em caso de sucesso instancia `th` com o identificador do novo *thread* e retorna 0, senão retorna um código de erro.

8

Criação de Threads

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

- `th` é o identificador do novo *thread*.
- `attr` permite especificar atributos de como o novo *thread* deve interagir com o resto do programa. Se `NULL` o novo *thread* é criado com os atributos por defeito. Os *threads* possuem atributos como política de escalonamento, prioridade, tipo de estado, etc, que podem ser definidos por invocação de funções adequadas, como por exemplo `pthread_attr_setdetachstate()`.
- `start_routine` é a função inicial que o novo *thread* deve executar.
- `arg` é o argumento único a passar à função `start_routine`. Múltiplos argumentos podem ser passados recorrendo a uma estrutura de dados e utilizando o endereço da estrutura como único argumento.

9

Junção de Threads (ou Sincronização Bloqueante)

- Tal como com os processos, por vezes é necessário esperar que um dado *thread* termine antes de continuar a execução. Com processos essa sincronização é conseguida pelas funções `wait()` ou `waitpid()`. Com *threads* a função é `pthread_join()`.

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

- `pthread_join()` suspende a execução até que o *thread* `th` termine. Assim que termine, `thread_return` é instanciado com o valor de retorno de `th` e `pthread_join()` retorna 0, senão retorna um código de erro.
- `th` é o identificador do *thread* a esperar que termine.
- `thread_return` é o valor de retorno do *thread* `th`. Se o valor de retorno for desprezável indique `NULL`.

10

Terminar Threads

- Por defeito, existem 2 formas de um *thread* terminar:
 - ◆ A função que iniciou o *thread* retorna.
 - ◆ A função `main()` retorna ou algum *thread* chama a função `exit()`. Nestes dois casos todos os *threads* terminam.
- Um outro modo de um *thread* terminar é este invocar directamente a função `pthread_exit()`.

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

- `pthread_exit()` termina o *thread* corrente.
- `retval` é o valor de retorno do *thread*.

11

Joinable Threads x Detached Threads

- Um *thread* pode estar num dos seguintes estados: **joinable** ou **detached**. O estado de um *thread* apenas condiciona o modo como este termina.
- Quando um *joinable thread* termina, parte do seu estado é mantido pelo sistema (identificador do *thread* e *stack*) até que um outro *thread* chame `pthread_join()` para obter o seu valor de retorno. Só então os recursos do *thread* são totalmente libertados.
- Os recursos de um *detached thread* são totalmente libertados logo que este termina. Qualquer chamada posterior a `pthread_join()` retorna um erro.
- Um *thread* pode ser criado como *joinable* ou como *detached* (ver atributos da função `pthread_create()`). Por defeito, os *threads* são criados como *joinable*. É também possível mudar dinamicamente o estado do *thread* para *detached*.

12

Joinable Threads x Detached Threads

Mudar o estado de um *thread* para *detached*:

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

- `pthread_detach()` retorna 0 se OK, valor positivo se erro.
- `th` é o identificador do *thread* a colocar *detached*.

Obter o identificador do *thread* corrente:

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

- `pthread_self()` retorna o identificador do *thread* corrente.

13

Integração Numérica com Threads (integra_threads.c)

```
float somas[NTHREADS];

main() {
    pthread_t thread[NTHREADS];
    float soma;
    int i;

    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); // main thread sincroniza com os restantes
        soma += somas[i];
    }
    printf("Area total= %f\n", H * (soma + (f(B) - f(A)) / 2));
}

void *integral(void *region_ptr) {
    int region = (int) region_ptr;
    ...
    somas[region] = soma_parcial;
    return NULL;
}
```

14

Cuidados na Utilização de Threads

- Passar endereços de variáveis em `pthread_create()` pode ser perigoso!

```
main () {
    ...
    for (i = 0; i < NTHREADS; i++)
        pthread_create(&thread[i], NULL, integral, (void *)&i);
    ...
}

void *integral(void *region_ptr) {
    int region = *((int *) region_ptr);
    ...
}
```

- A variável `i` pode ser alterada no *main thread* antes de ser lida pelo novo *thread*!

15

Cuidados na Utilização de Threads

- Retornar directamente o valor calculado nem sempre é possível!

```
main () {  
    ...  
    for (i = 0; i < NTHREADS; i++) {  
        float soma_parcial;  
        pthread_join(thread[i], (void *)&soma_parcial);  
        soma += soma_parcial;  
    }  
    ...  
}  
  
void *integral(void *region_ptr) {  
    float soma_parcial;  
    ...  
    // gcc error: cannot convert to a pointer type  
    return ((void *) soma_parcial);  
}
```

- Se o valor a retornar for por exemplo do tipo *float*, o compilador não consegue converter para um apontador.

16

Cuidados na Utilização de Threads

- Retornar endereços de variáveis locais pode originar erros ou *segmentation fault*!

```
main () {  
    ...  
    for (i = 0; i < NTHREADS; i++) {  
        float *soma_parcial;  
        pthread_join(thread[i], (void *)&soma_parcial);  
        soma += *soma_parcial;  
    }  
    ...  
}  
  
void *integral(void *region_ptr) {  
    float soma_parcial;  
    ...  
    // gcc warning: function returns address of local variable  
    return ((void *) &soma_parcial);  
}
```

- Como a função termina, o endereço de memória da variável local *soma_parcial* fica fora de âmbito.

17

Cuidados na Utilização de Threads

- Mesmo utilizando a função `pthread_exit()` o problema acontece!

```
void *integral(void *region_ptr) {  
    float soma_parcial;  
    ...  
    pthread_exit((void *) &soma_parcial);  
}
```

- O endereço de memória da variável local `soma_parcial` fica igualmente fora de âmbito quando o *main thread* sincroniza através da chamada `pthread_join()`.

18

Sincronização e Regiões Críticas

- A principal causa da ocorrência de erros na programação de *threads* está relacionada com o facto dos dados serem todos partilhados. Apesar de este ser um aspectos mais poderosos da utilização de *threads*, também pode ser um dos mais problemáticos.
- O problema existe quando dois ou mais *threads* tentam aceder/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).

19

Mutexs

- 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 seja libertado.

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

- `pthread_mutex_init()` 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 defeito.

```
pthread_mutex_t mutex;  
pthread_mutex_init(&mutex, NULL);
```

- Outra forma de iniciar um mutex (se estaticamente alocado) com os atributos por defeito é a seguinte:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

20

Operações sobre Mutexs

Obter o *lock* no mutex:

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

Libertar o *lock*:

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

Faz tentativa de obter o *lock* mas não bloqueia caso não seja possível:

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

- `mutex` é a variável que representa o mutex.
- Todas as funções retornam 0 se OK, valor positivo se erro.

21

Integração Numérica com Mutexs (integra_mutex.c)

```
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;
}
```

22

Fila de Tarefas com Mutexs

```
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);
}
```

23

Variáveis de Condição

- 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 tome 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.

24

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. O *thread* B ao sincronizar mais tarde nessa variável, deverá ficar à espera que um outro *thread* volte a sinalizar a variável de condição.

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

- `pthread_cond_init()` inicia uma variável de condição. Retorna 0 se OK, valor positivo se erro.

25

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. Se `NULL` é iniciada com os atributos por defeito.

```
pthread_cond_t cond;
pthread_cond_init(&cond, NULL);
```

- Outra forma de iniciar uma variável de condição (se estaticamente alocada) com os atributos por defeito é a seguinte:

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

26

Sinalizar uma Variável de Condição

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

- `pthread_cond_signal()` acorda um dos *threads* bloqueados na variável `cond`. Caso existam vários *threads* bloqueados, apenas um é acordado (não é especificado qual).

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

- `pthread_cond_broadcast()` acorda todos os *threads* que possam estar bloqueados na variável `cond`.
- Em ambos os casos, se nenhum *thread* estiver bloqueado na variável especificada, nada acontece.
- Ambas as funções retornam 0 se OK, valor positivo se erro.

27

Bloquear numa 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 perda desse mesmo sinal.

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

- `pthread_cond_wait()` bloqueia o *thread* na variável de condição `cond`.
- `pthread_cond_wait()` de um modo atómico liberta o mutex (tal como se executasse `pthread_unlock_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 o *thread* é acordado, `pthread_cond_wait()` readquire o *lock* no mutex (tal como se executasse `pthread_lock_mutex()`) antes de regressar à execução.

28

Fila de Tarefas com Variáveis de Condição

```
pthread_cond_t flag_cond = PTHREAD_COND_INITIALIZER;
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);
        while (flag_is_set == FALSE) // faz loop enquanto flag_is_set é 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);
}
```

29

Thread-Specific Data

- Existem duas soluções básicas para associar dados a um *thread* durante toda a execução:
 - ◆ Guardar os dados numa estrutura global associada com o *thread*.
 - ◆ Passar os dados como argumento em todas as funções que o *thread* invoque.
- No entanto, em algumas circunstâncias, nenhuma das soluções funciona. Vejamos o que acontece se pretendermos reescrever uma biblioteca/módulo de funções para suportar *multithreading* (**thread-safe functions**).

30

Thread-Specific Data

```
int fd;

init_module( ... ) {
    ...
    fd = open("module.log", O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
    ...
}

use_module( ... ) {
    ...
    write(fd, ...);
    ...
}
```

- Por um lado não queremos redefinir os argumentos das funções e por outro lado não sabemos o número adequado de estruturas globais a utilizar.

31

Chaves

- Para resolver a situação o Pthreads introduz o conceito de chave (**key**), um tipo de apontador que associa dados com *threads*. Apesar de durante a execução todos os *threads* referirem uma mesma chave, cada um acede a dados diferentes.

```
int pthread_key_create(pthread_key_t *key, void
                      (*destr_function) (void *));
```

- `pthread_key_create()` aloca uma nova chave em todos os *threads* em execução e inicia-as com NULL. Se novos *threads* forem entretanto criados, todas as chaves do sistema são igualmente alocadas para os novos *threads* e iniciadas com NULL.
- `key` é o identificador da chave a alocar.
- `destr_function`, se não NULL, especifica uma função a ser executada no caso de o *thread* corrente terminar, em que o argumento da função é o valor associado com `key`.

32

Iniciação Única

```
int pthread_once(pthread_once_t *once_control, void
                 (*init_routine) (void));
```

- `pthread_once()` é um mecanismo para garantir que um determinado código de iniciação não é executado mais do que uma vez. A primeira vez que `pthread_once()` é executado para um dado argumento de controle `once_control`, a função `init_routine` é executada. Subsequentes chamadas a `pthread_once()` com o mesmo argumento de controle não fazem nada.
- `once_control` é o argumento de controle. Deve ser estaticamente iniciado com `PTHREAD_ONCE_INIT`.

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;
```
- `init_routine` é a função a executar não mais do que uma vez.

33

Operações com Chaves

Instanciar uma chave:

```
int pthread_setspecific(pthread_key_t key, const void *pointer);
```

- `pthread_setspecific()` instancia a chave `key` do *thread* corrente com o valor `pointer`.

Ler o valor de uma chave:

```
void * pthread_getspecific(pthread_key_t key);
```

- `pthread_getspecific()` retorna o valor associado com a chave `key` do *thread* corrente.

34

Thread-Specific Data (module_key.c)

```
pthread_once_t fd_once = PTHREAD_ONCE_INIT;
pthread_key_t fd_key;

init_module( ... ) {
    int fd;
    ...
    pthread_once(&fd_once, module_once);
    sprintf(filename, "module_t%d.log", (int) pthread_self());
    fd = open(filename, O_RDWR | O_CREAT | O_TRUNC, S_IRUSR | S_IWUSR);
    pthread_setspecific(fd_key, (void *) fd);
    ...
}

use_module( ... ) {
    int fd = (int) pthread_getspecific(fd_key);
    ...
    write(fd, ...);
    ...
}

void module_once (void) { pthread_key_create(&fd_key, module_destructor); }

void module_destructor (void *fd) { close((int)fd); }
```

35

Implementações do Pthreads

- As implementações do Pthreads podem dividir-se em 3 categorias:
 - ◆ **User Threads:** implementações geridas ao nível do espaço do utilizador.
 - ◆ **Kernel Threads:** implementações geridas ao nível do *kernel*.
 - ◆ **Two-Level Scheduler Threads ou Lightweight Processes:** implementações híbridas.
- Cada uma das implementações condiciona de modo diferente o modo de escalonamento e performance dos *threads* de um programa.
- No entanto, todas as implementações disponibilizam o objectivo base do paradigma: concorrência.

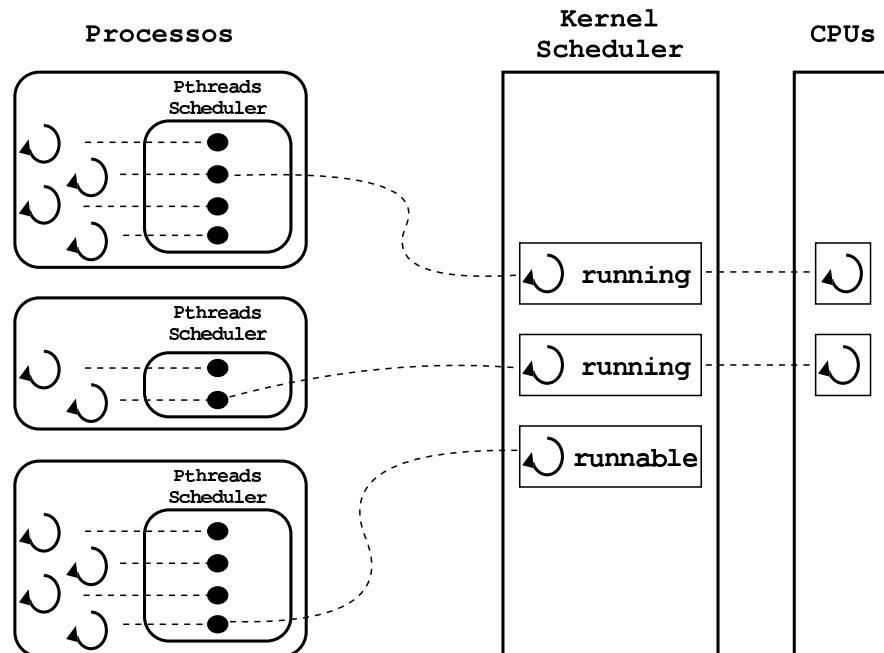
36

User Threads

- São executados e geridos no espaço do utilizador, no âmbito de um processo, sem estarem visíveis para o *kernel* do sistema.
- A biblioteca Pthreads implementa toda a política de escalonamento e multiplexagem dos contextos de execução. O *kernel* não tem a noção de *threads*, continua apenas a escalonar processos.
- Implementações:
 - ◆ Digital OpenVMS (anteriores à versão 7.0)

37

User Threads



38

User Threads: Vantagens e Inconvenientes

- (+) São mais fáceis de implementar pois não interferem com o *kernel*.
- (+) São bastante eficientes, pois não envolvem o *kernel* para sincronização.
- (+) São escalares, pois a criação de mais e mais *threads* não sobrecarrega o sistema.
- (–) *Threads* de um mesmo processo competem entre si pelo tempo de CPU e não entre todos os *threads*/processos existentes no sistema.
- (–) Em máquinas multiprocessador não é possível ter *threads* de um mesmo processo em paralelo.

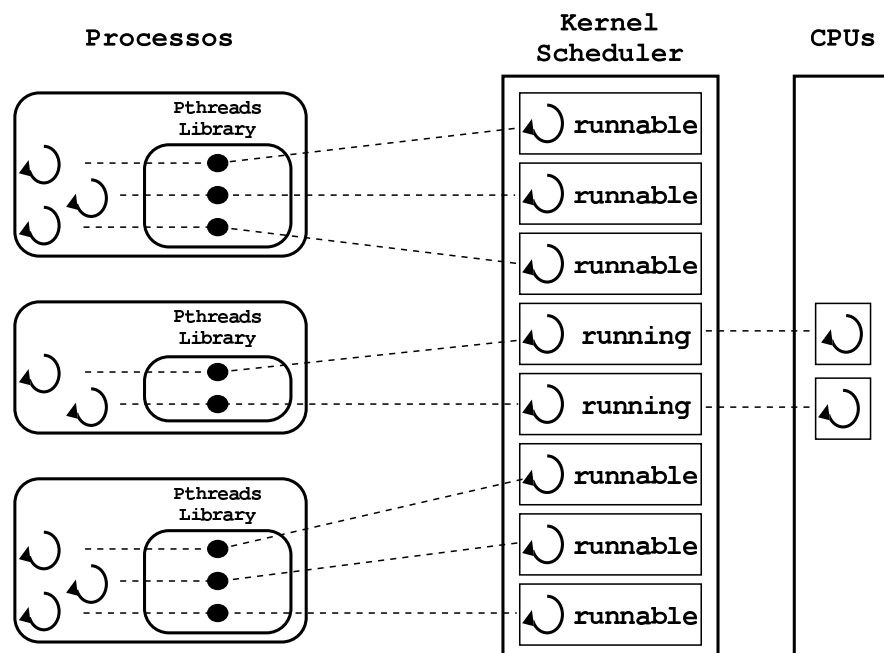
39

Kernel Threads

- São executados e geridos pelo *kernel* como se fossem processos.
- O *kernel* é o responsável pela multiplexagem do contexto de execução de cada *thread*. Como tal, alguma informação tradicionalmente associada a processos, como sejam a prioridade de escalonamento, atribuição de CPU para execução, conjunto de registos guardados, é igualmente necessária para gerir *threads*.
- Implementações:
 - ◆ GNU/Linux
 - ◆ Digital UNIX pre-Version 4.0

40

Kernel Threads



41

Kernel Threads: Vantagens e Inconvenientes

- (+) Todos os *threads* do sistema competem entre si pelo tempo de CPU e não entre os *threads* do mesmo processo.
- (+) Em máquinas multiprocessador é possível executar *threads* de um mesmo processo em paralelo.
- (–) Requer um maior esforço de implementação.
- (–) Apesar de menos dispendiosa que a criação de um processo, a criação e manutenção de novos *threads* acarreta algum custo ao sistema. Este custo pode ser desnecessário se o nosso programa não correr em máquinas multiprocessador (provavelmente *user threads* seria suficiente).
- (–) São menos escalares, já que o manuseamento de bastantes *threads* pode degradar consideravelmente a performance do sistema.

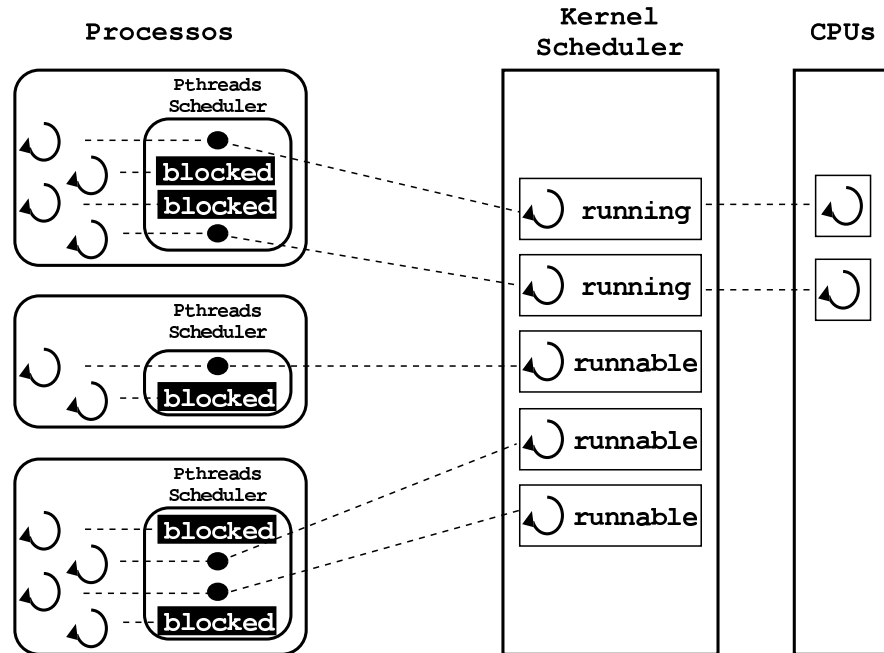
42

Two-Level Scheduler Threads

- São executados e geridos em cooperação entre a biblioteca Pthreads e o *kernel*. Ambos mantêm estruturas de dados para representar *user* e *kernel threads* respectivamente.
- Utiliza dois níveis de controlo em que conjuntos de *user threads* são mapeados sobre *kernel threads* que por sua vez são escalados para execução nos processadores do sistema.
- O programador escreve o seu programa em termos de *user threads* e pode especificar quantos *kernel threads* vão estar associados ao processo.
- Implementações:
 - ◆ Solaris
 - ◆ Digital OpenVMS 7.0
 - ◆ Digital UNIX 4.0

43

Two-Level Scheduler Threads



44

Two-Level Scheduler Threads: Vantagens e Inconvenientes

- (+) O melhor dos dois mundos: boa performance e baixos custos de criação/manutenção.
- (–) Complexidade do sistema que se reflecte quando o programador necessita de fazer *debugging*.

45

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* vagabundo pode corromper as estruturas de dados de todos os outros *threads*. Um processo vagabundo não consegue fazer isso porque o seu espaço de endereçamento é privado.
- A memória a copiar 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 o mais possível e apenas é efectuada se a memória for alterada. Isto diminui, de certa forma, o custo a pagar 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.
- Partilhar dados entre *threads* é trivial porque estes partilham a mesma memória. Partilhar dados entre processos requer a utilização de uma das técnicas de comunicação entre processos (IPC). Apesar de mais penoso, isto torna a utilização de múltiplos processos menos susceptível de erros de concorrência.

46

Bibliografia

- ***Advanced Linux Programming.*** M. Mitchell, J. Oldham, A. Samuel. New Riders. **(Capítulo 4)**
- ***Pthreads Programming.*** B. Nichols, D. Buttler, J.P. Farrell. O'Reilly. **(Capítulos 1, 3, 4 (Págs 109–128) e 6 (Págs 193–202))**

47