

INE5410 – Programação Concorrente

Unidade II – Fundamentos de Programação Concorrente

Prof. Frank Siqueira
frank.siqueira@ufsc.br



Conteúdo

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

Threads

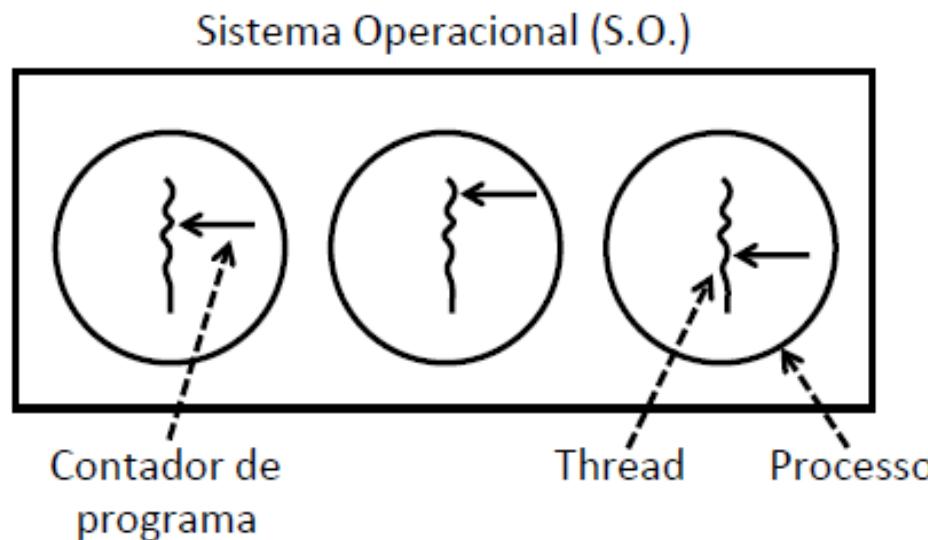
- Definição
 - Threads são atividades concorrentes executadas por um processo
 - Ao ser criado, um processo possui apenas uma thread
 - Novas threads podem ser criadas no código do programa
 - Threads pertencentes a um mesmo processo compartilham recursos e memória
 - Threads permitem que um mesmo programa use simultaneamente vários núcleos do processador (ou seja, permite paralelismo real)

Threads

- Cada thread possui:
 - Um contador de programa (PC)
 - Registradores
 - Pilha de execução
 - Estado
- Threads do mesmo processo compartilham:
 - Espaço de endereçamento
 - Variáveis globais do programa
 - Arquivos abertos, alarmes e sinais
 - Informações de contabilidade do processo

Threads

- Processos X Threads



Três processos, cada um com uma thread



Um único processo com três threads

© Tanenbaum

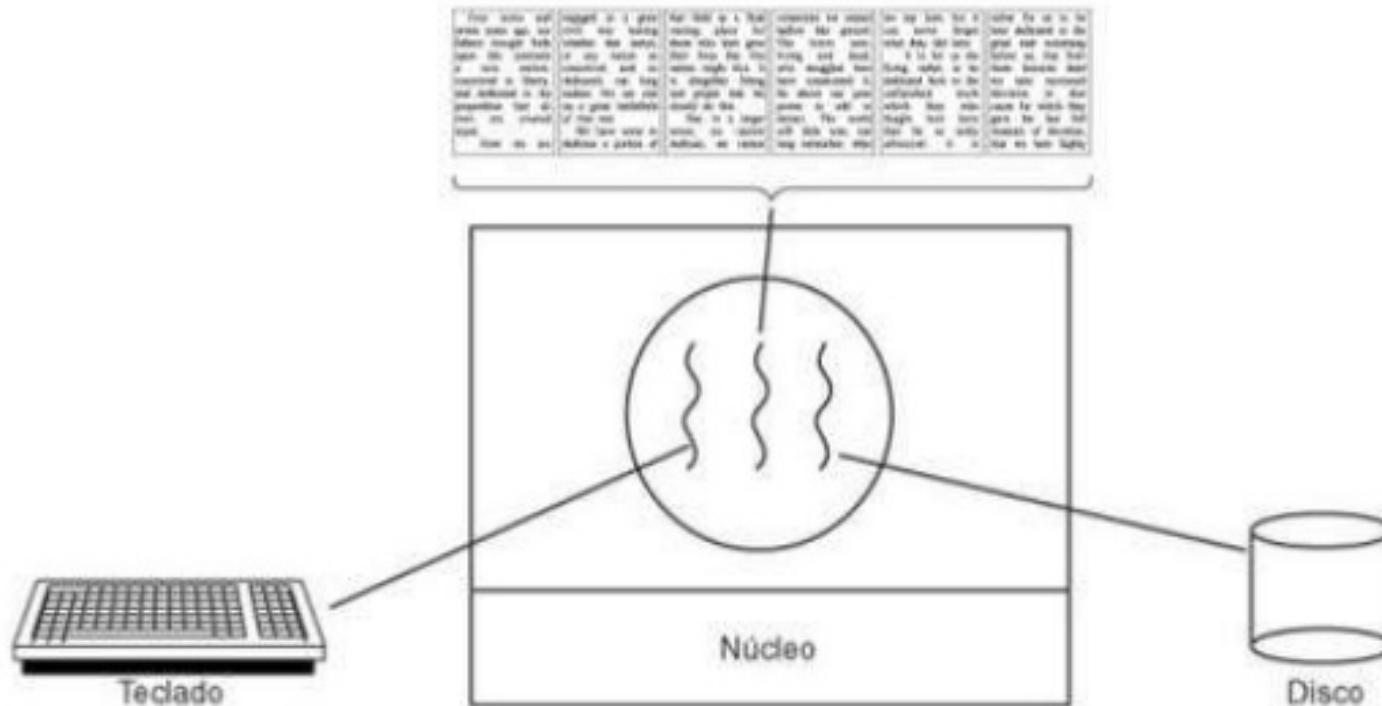
Threads

- Processos X Threads

	Processos	Threads
Criação	Cara	Barata (+ rápida)
Uso de Recursos	Maior	Menor
Troca de Contexto	Completa	Parcial (+rápida)
Área de Memória	Independente	Compartilhada
Comunicação	Inter-Processo	Intra-Processo
Código-fonte	Independente	Mesmo código
Suporte em SOs	Quase todos	Os mais completos
Suporte em Ling. Prog.	Quase todas	As mais modernas

Threads

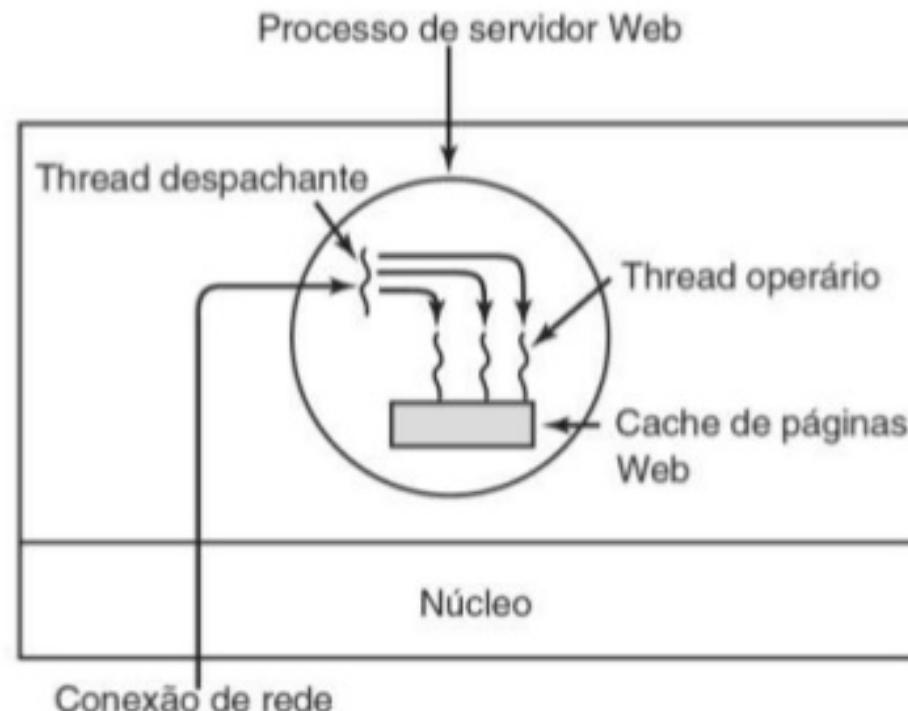
- Exemplos de Aplicações Multithread
 - Um processador de texto pode ter que executar várias atividades simultaneamente



© Tanenbaum

Threads

- Exemplos de Aplicações Multithread
 - Um servidor Web pode ter uma thread para ouvir a rede e outras para processar requisições



© Tanenbaum

Threads

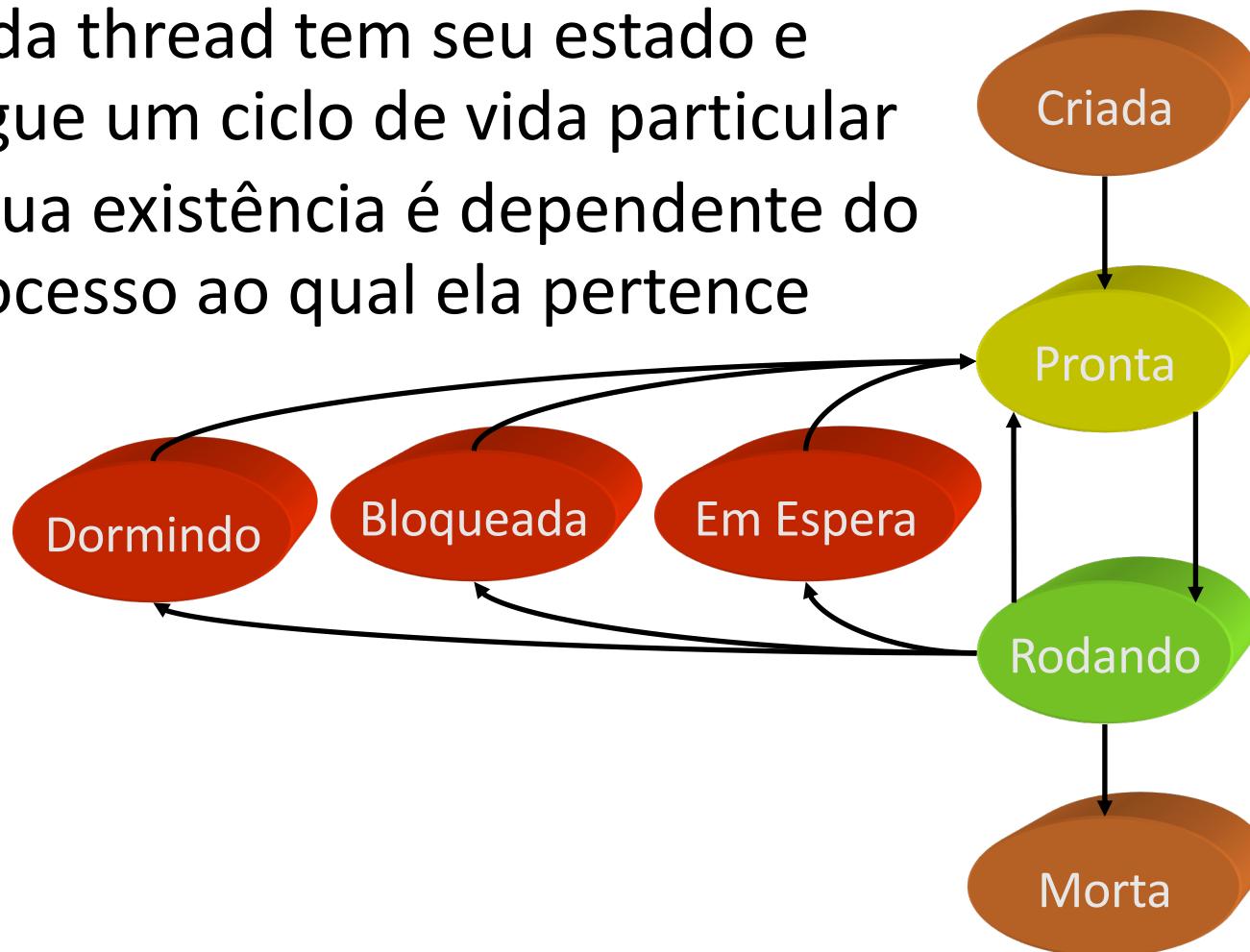
- Compartilhamento de Dados
 - Threads de um mesmo processo compartilham variáveis globais
 - Permite que threads trabalhem conjuntamente em um mesmo problema a ser resolvido
 - Útil nas seguintes situações:
 - Threads trabalham sobre os mesmos dados;
 - Threads precisam trocar informações entre si.
 - Acesso concorrente aos dados compartilhados pode ocasionar inconsistências, que podem ser evitadas realizando controle de concorrência

Threads

- Suporte a Threads
 - Threads nativas de sistema: threads são criadas através de chamadas de sistema.
Ex.: Linux (kernel 2.6 ou +), Windows (95 ou +)
 - Bibliotecas/APIs para programação multithread: threads são fornecidas por bibliotecas ou APIs externas.
Ex.: POSIX threads
 - Linguagem de programação multithreaded: a linguagem fornece nativamente mecanismos para criação de threads.
Ex.: Python, Java, C#, C++11, etc.

Threads

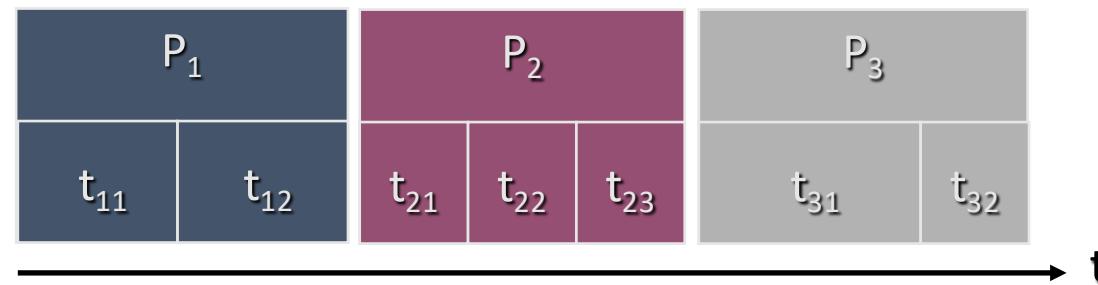
- Ciclo de Vida
 - Cada thread tem seu estado e segue um ciclo de vida particular
 - A sua existência é dependente do processo ao qual ela pertence



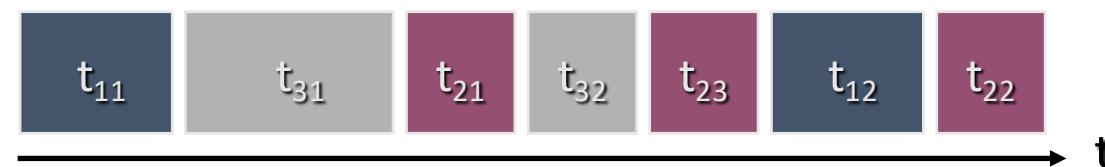
Threads

- Escalonamento

- Por Processo: escalonador aloca tempo para execução dos processos, que definem como usar este tempo para executar suas threads



- Por Thread: escalonador define a ordem na qual as threads serão executadas



Threads

- Troca de Contexto
 - Quando duas threads de um mesmo processo se alternam no uso do processador, ocorre uma troca de contexto parcial
 - Numa troca parcial, o contador de programa, os registradores e a pilha devem ser salvos
 - Uma troca de contexto parcial é mais rápida que uma troca de contexto entre processos
 - Uma troca de contexto completa é necessária quando uma thread de um processo que não estava em execução assume o processador

Threads

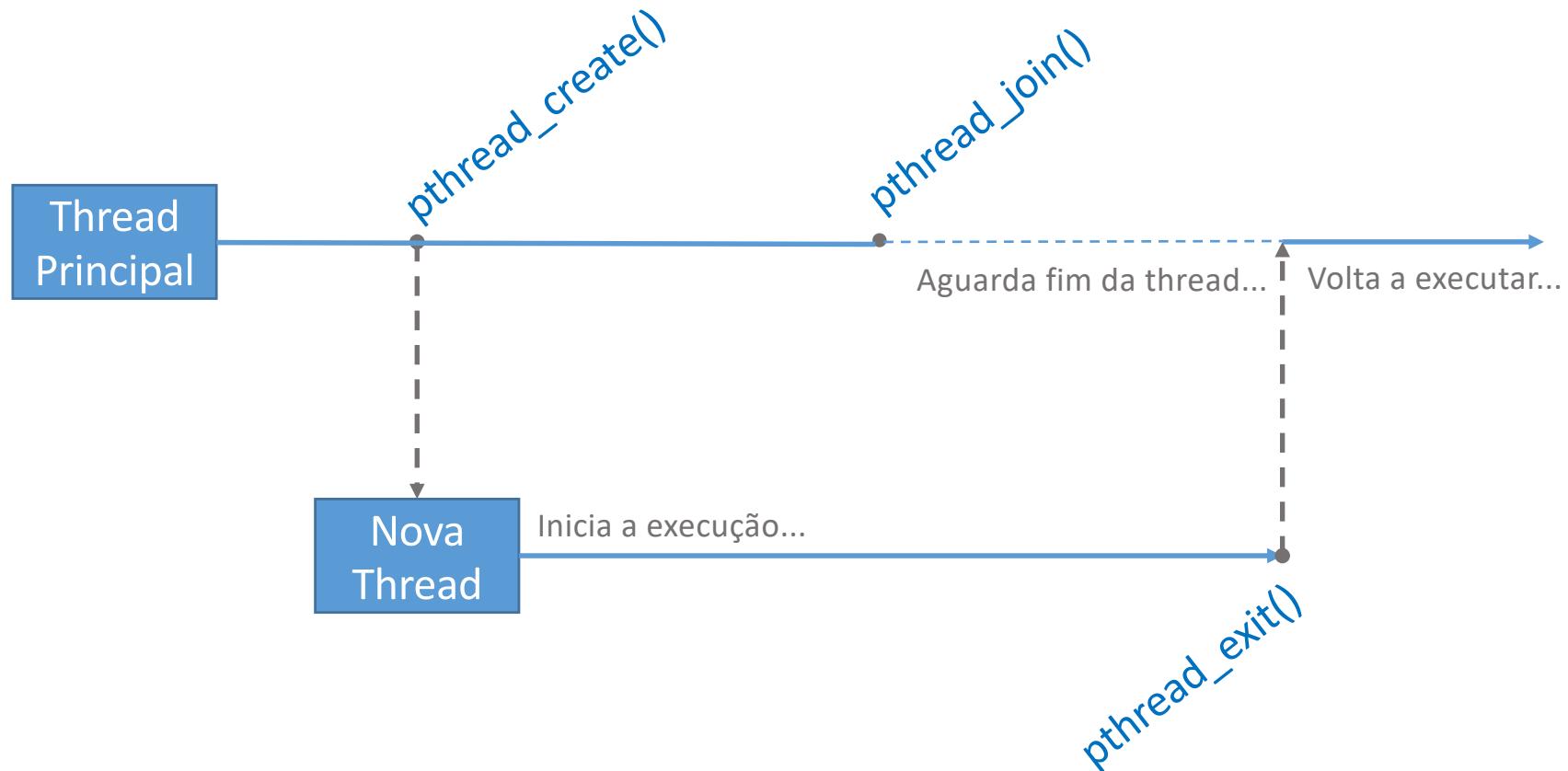
- POSIX (*Portable Operating System Interface*)
 - Conjunto de padrões abertos que busca definir interfaces compatíveis para sist. operacionais
 - Mantido pela IEEE e referendado por ISO e ANSI
 - Objetivo: garantir portabilidade de Código
- POSIX Threads
 - Interface padrão para uso de threads
 - Suportada em diversos sistemas operacionais
 - Existem diferentes implementações deste padrão
 - Implementação em C: biblioteca *pthread*
 - Incluir o *header file*: #include <pthread.h>
 - Compilar com *flag*: -lpthread

Threads

- Principais funções de POSIX Threads
 - Criar uma Thread:
`pthread_create(<thread>, <attrs>, <rotina>, <args>);`
 - Obter identificador da Thread:
`pthread_self();`
 - Suspender a execução da Thread:
`pthread_delay_np(<tempo>);`
 - Aguardar o término de uma Thread:
`pthread_join (<thread>, <retorno>);`
 - Finalizar a Thread:
`pthread_exit (<retorno>);`

Threads

- Sincronização entre POSIX Threads
 - Uso de `pthread_join()` e `pthread_exit()`



Threads

- A rotina a ser executada pela Thread POSIX precisa ter a seguinte assinatura:

```
void * <nome_funcao> ( void * <nome_arg> );
```

- Para enviar argumentos não-void, deve-se fazer a conversão (*cast*) para void * ao criar a thread
- Para obter o valor original do argumento, faça o *cast* para o tipo correspondente ao acessá-lo
- É possível enviar vários argumentos colocando-os em uma estrutura de dados (*struct*)
- O mesmo se aplica ao valor retornado pela função (obtido ao fazer o *join*)

Threads

- POSIX Threads
 - Exemplo: criando várias Threads, aguardando que terminem de executar e obtendo retorno

```
...
pthread_t threads[NUM_THREADS]; // Array com identificadores das threads
void * retorno;                // Ponteiro para retorno das threads

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

// Loop que aguarda o término de todas as threads e obtém valor retornado
for(int i=0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], &retorno);
    printf("Thread %d encerrada. Retorno = %d\n",
          threads[i], *((int *) retorno));
}

...
```

Threads

- POSIX Threads

- Exemplo: função com código da thread que retorna um valor ao final da execução

```
...
int valor = 0;      // Valor que será retornado pela thread

// Função com o código da thread
void *func_thread(void *arg) {
    // Imprime o identificador da thread
    printf("Thread %d criada.\n", pthread_self());

    // Encerra a thread e retorna um valor
    pthread_exit(&valor);
}

...
```

Threads

- POSIX Threads
 - Exemplo: criando várias Threads e enviando um inteiro como argumento

```
...
pthread_t threads[NUM_THREADS]; // Array com identificadores das threads
int valores[NUM_THREADS];      // Array com valores enviados às threads

// Loop que cria as threads e envia valores
for(int i=0; i < NUM_THREADS; i++) {
    valores[i] = i;
    pthread_create(&threads[i], NULL, func_thr2, (void *) &valores[i]);
}

// Loop que aguarda o término de todas as threads
for(int i=0; i < NUM_THREADS; i++) {
    pthread_join(threads[i], NULL);
    printf("Thread %d encerrada.\n", threads[i]);
}
...
```

Threads

- POSIX Threads

- Exemplo: função com código da thread que recebe um valor como argumento

```
...
// Função com o código da thread
void *func_thr2(void *arg) {
    // Obtém o valor recebido; converte void* em int* e derreferencia
    int valor = *((int *)arg);

    // Imprime o identificador da thread e o valor recebido
    printf("Thread %d criada. Valor recebido = %d\n",
           pthread_self(), valor);

    // Encerra a thread
    pthread_exit(NULL);
}
...
```

Threads

- Escalonamento de Threads no Linux
 - CFS (*Completely Fair Scheduler*)
 - Escalonador padrão do Linux
 - Cria filas separadas para cada núcleo
 - Threads de um processo são colocadas na mesma fila
 - Processo pode trocar de fila (afinidade)
 - As filas são balanceadas periodicamente
 - Prioridade Estática
 - Real-time: 0 (mais alta) a 99
 - Normal: 100 a 139 (mais baixa) -> Nice: -20 a 19, 0 default.
 - Alterada com a função `nice()`
(Obs: só o *root* pode aumentar prioridade)
 - *Quantum*
 - Tempo máximo que cada tarefa pode ficar com a CPU
 - Processos mais prioritários têm *quantum* maior

Threads

- Escalonamento de Threads no Linux (cont.)
 - Prioridade Dinâmica
 - Altera a prioridade estática em ± 5 níveis com base no tempo médio que a thread fica dormindo
 - Threads que dormem muito são favorecidas
 - Threads muito ativas perdem prioridade
 - Políticas de Escalonamento – Threads *Real-time*
 - FIFO: thread com mais alta prioridade executa até liberar a CPU (não considera o *quantum* de tempo)
 - *Round Robin*: threads da fila de mais alta prioridade se alternam na CPU (cada thread usa o seu *quantum*)
 - *Deadline*: introduzida no *kernel* 3.14; dá prioridade a threads com menor prazo para terminar

Threads

- Escalonamento de Threads no Linux (cont.)
 - Políticas de Escalonamento – Threads Normais
 - *Other*: política padrão (*Round Robin*); threads da fila de mais alta prioridade se alternam na CPU
 - *Batch*: usado para execução de tarefas em lote (com uso intenso da CPU e sem interação com o usuário)
 - *Idle*: próprio para execução de tarefas com baixa prioridade em *background*

Threads

- Threads no Windows
 - Criar uma Thread:
`CreateThread (<atribs>, <tam_stack>, <rotina>, <params>, <flags>,<thread_id>);`
 - Obter Identificador da Thread:
`GetCurrentThreadId();`
 - Suspender Execução:
`Sleep(<tempo>) ou SuspendThread (<thread>)`
 - Finalizar a Thread:
`ExitThread(<retorno>)`
 - Destruir uma Thread:
`TerminateThread (<thread>,<retorno>)`

Threads

- Escalonamento de Threads no Windows
 - Cada processo é associado a uma classe
 - Existem 6 classes: *Idle*, *Below Normal*, *Normal (default)*, *Above Normal*, *High Priority*, *Realtime*.
 - Pode ser definida na criação do processo e alterada chamando `SetPriorityClass()`
 - Cada thread tem uma prioridade
 - Existem 7 prioridades: *Idle*, *Lowest*, *Below Normal*, *Normal (default)*, *Above Normal*, *Highest*, *Time critical*.
 - Pode ser alterada chamando `SetThreadPriority()`
 - A prioridade base da thread é determinada pela classe do processo e por sua prioridade

Threads

- Escalonamento de Threads no Windows (cont.)
 - Prioridade Base de uma Thread
 - Valor de 0 (mais baixa) a 31 (mais alta)
 - Prioridade 0 é usada somente pelo sistema
 - Classe real-time é de uso exclusivo do administrador

Prioridade da thread	Classe do processo					
	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

Threads

- Escalonamento de Threads no Windows (cont.)
 - A prioridade dinâmica é determinada pela prioridade base + *boost* definido pelo sistema
 - Somente ganham *boost* threads com prior. base < 16
 - Evita que algumas threads nunca sejam executadas
 - Privilegia threads de processos em primeiro plano, que recebem dados do teclado ou mouse, respostas de operações de I/O, etc.
 - Escalonamento *Round Robin*
 - Threads com prioridade dinâmica mais alta são executadas, alternando-se caso haja mais threads de mesma prioridade do que núcleos disponíveis
 - Uma thread só é executada se não houver outra thread pronta com prioridade dinâmica maior

Threads

- Escalonamento de Threads no Windows (cont.)

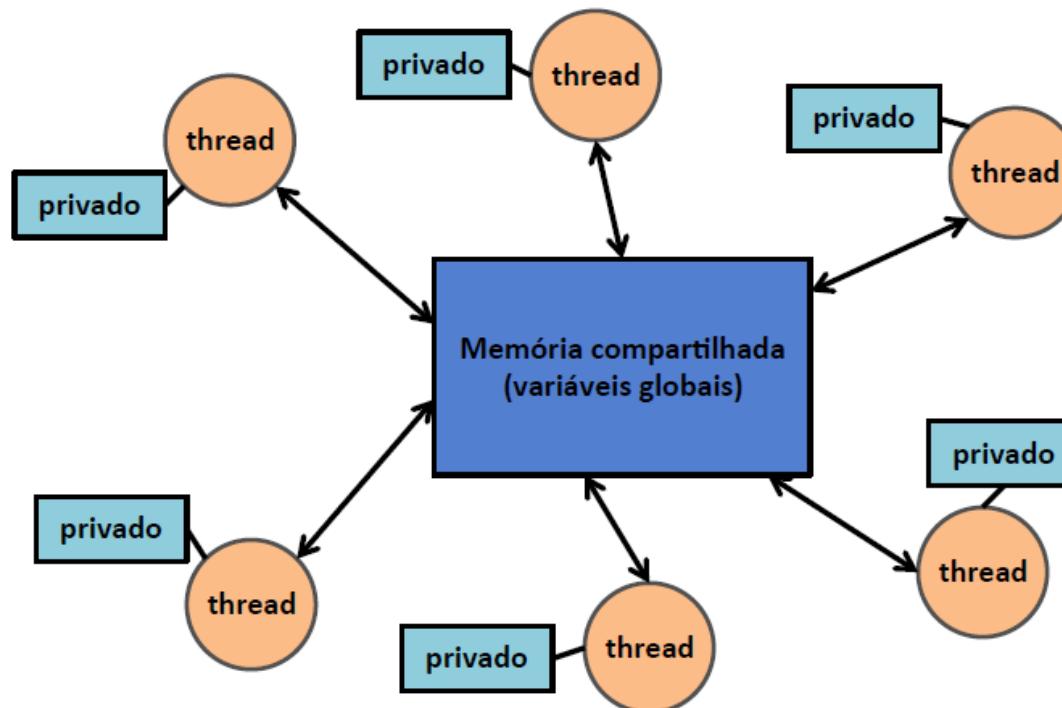
Prior. Dinâmica	Threads
31	
30	
...	
17	
16	A
15	→ B → C → D ━
14	→ E → F ━
13	
12	→ G → H → I ━
11	
10	→ J → K ━
9	L
8	→ M → N → O ━
...	
1	R
0	S

Threads

- Comunicação entre Processos e Threads
 - É comum que processos/threads precisem trocar dados entre si durante a sua execução
 - Podem precisar de um dado produzido por outro processo/thread para continuar sua execução
 - Para isso, processos precisam usar mecanismos de comunicação, como Pipes, Sockets, etc.
 - Ex.: `ps -ef | grep program`
`ls -al | more`
 - Threads de um mesmo processo podem trocar dados por meio de memória compartilhada
 - Compartilham o mesmo espaço de endereçamento
 - Acesso à memória é muito mais rápido que os mecanismos de comunicação entre processos

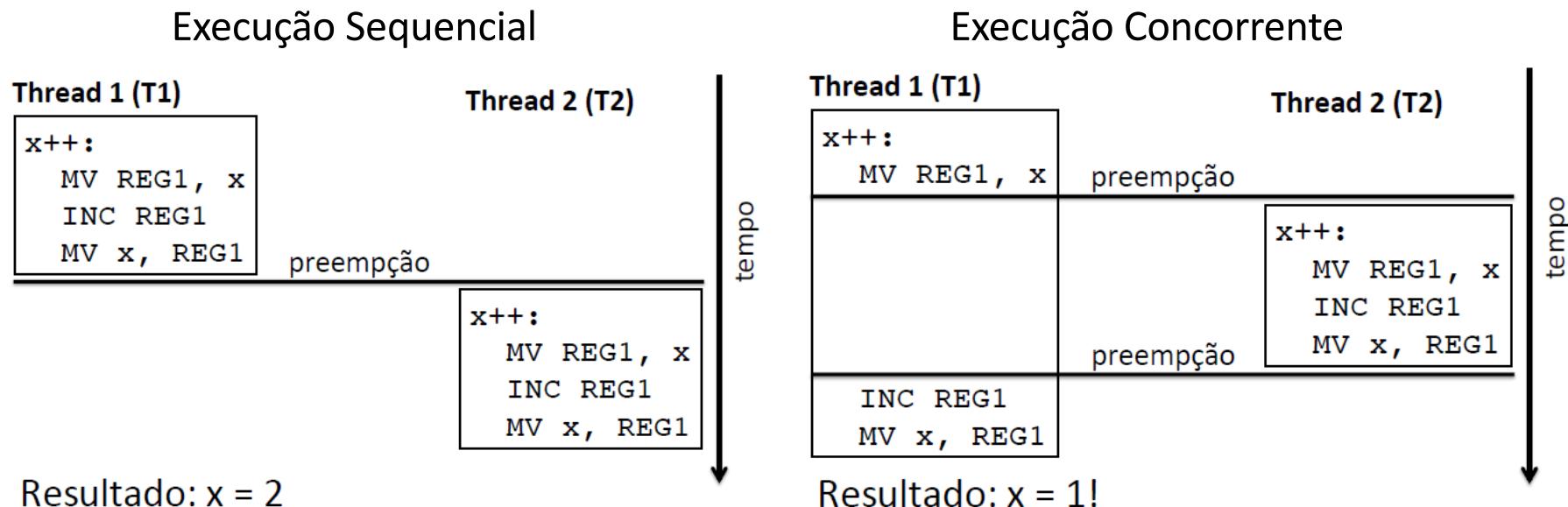
Threads

- Comunicação entre Threads
 - As variáveis globais são compartilhadas por todas as threads do processo
 - Cada thread pode ter suas variáveis locais



Threads

- O acesso concorrente a dados pode resultar em **inconsistências**
 - Exemplo: duas threads incrementam uma variável global x, com valor inicial 0



Threads

- O exemplo anterior mostra o que se chama na literatura de **condição de corrida**
 - Situação em que, dependendo da ordem de execução das operações, podemos ter um resultado diferente
- Isso mostra a necessidade de que seja feito **controle de concorrência**
 - Ou seja, em determinadas situações temos que impedir que haja concorrência entre threads que acessam os mesmos dados!
 - Há diversos mecanismos para isso, que serão descritos em seguida