Universidade Federal de São Carlos
Departamento de Computação

Sistemas Operacionais

Gerenciamento de Threads no Windows

Gerenciamento de Processo

Um sistema operacional envolve atividades assíncronas e algumas vezes paralelas;

A noção de processo de software é usada em sistemas operacionais para expressar o gerenciamento e controle de suas atividades;

Processo é um dos conceitos fundamentais no projeto de um sistema operacional moderno.

Processos and *Threads*

Uma aplicação consiste de um ou mais processos

– Um processo é um programa em execução

Um ou mais *threads* executam no contexto de um processo;

Um *thread* é a unidade básica ao qual o sistema operacional aloca tempo do processador;

Cada processo é iniciado com um único *thread*, geralmente chamado de *thread* primário (processo não executa);

Pode criar *threads* adicionais a partir de qualquer de seus threads.

Processos

Cada processo provê os recursos necessários para um programa executar:

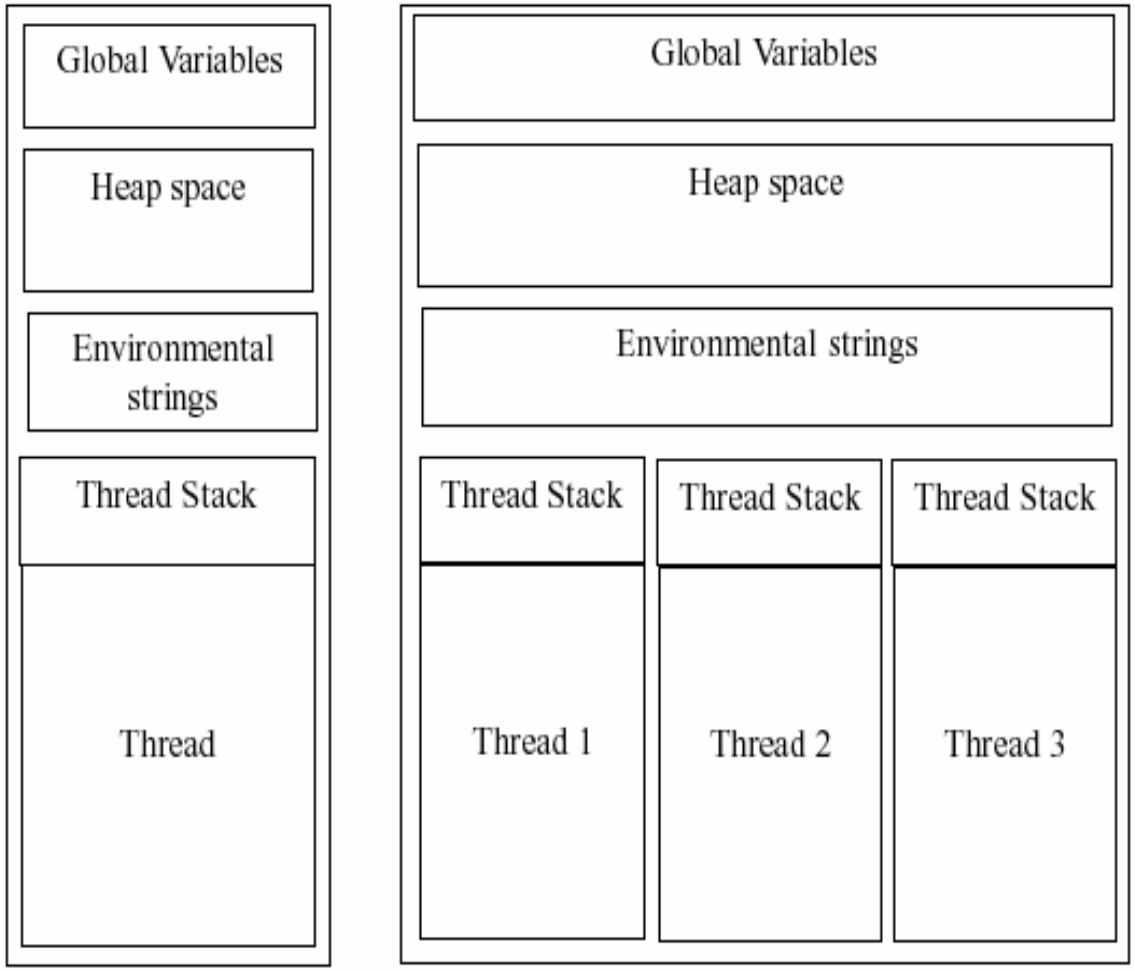
- **Espaço de endereçamento virtual;**
- **Código executável;**
- ***Heaps* do processo;**
- **Manipuladores para objetos do sistema e outros *heaps*;**
- **Contexto de segurança;**
- **Identificador único de processo;**
- **Variáveis de Ambiente;**
- **No mínimo um *thread* de execução.**

Threads

Cada *thread* partilha código, variáveis globais, *strings* de ambientes, e recursos.
Além disso, um thread tem:

- **Uma pilha para chamadas de procedimento;**
- **Armazenamento local do thread (TLS).**
- **São escalonados de forma independente;**
- **Tem uma estrutura de contexto, mantida pelo *kernel*.**

Processos and Threads



Thread Único versus MultiThread

Gerenciamento de Threads

Multithread:

- O programador especifica que aplicações contem *threads* de execução;
- Cada *thread* designa a porção do programa que pode executar concorrentemente com outros *threads*;
- Escrever programas *multithread* pode ser desafiador.

Threads

Definição:

- Processo Leve (LWP - *Lightweight process* – troca de contexto rápida);
- Compartilha espaço de endereçamento e outras informações globais com seu processo;
- Registradores, *stack*, mascara de sinal e outros dados específico do thread são locais a cada *thread*.

Threads podem ser gerenciados pelo sistema operacional ou pela aplicação do usuário.

Motivação Para o Uso de Threads

Atualmente o uso de *Thread* está muito difundido devido a uma tendência em:

- Projeto de Software
  Expressa mais naturalmente o paralelismo inerente às tarefas;
- Performance
  Escala (*Scales*) melhor em sistemas multiprocessadores;
- Cooperação
  O compartilhamento do espaço de endereço incorre em um menor *overhead* que os mecanismos de IPC.

Exemplos:

- Web *servers*
- Processadores de Texto

Threads no Windows

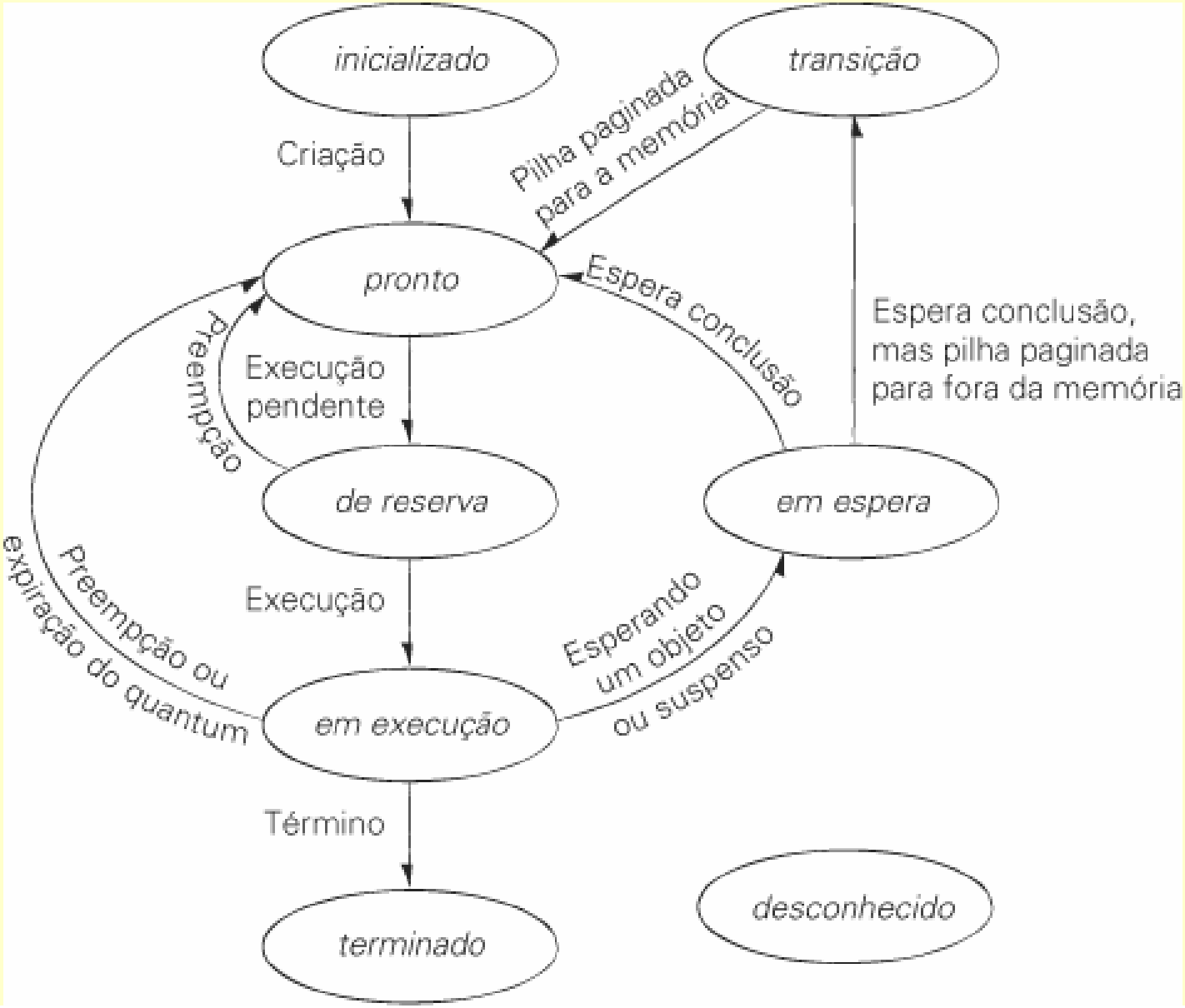
Threads

- A unidade real de execução “despachada” para um processador;

- Executa um pedaço do código do processo no contexto do processo, usando os recursos do processo;

- O contexto de execução contem
  - *Runtime stack*;
  - Estado dos registros do processador;
  - Diversos atributos.

Diagrama de Transição de Estado



API Windows Win32

Funções de gerenciamento de threads:

- *CreateThread*
- *ExitThread*
- *GetExitCodeThread*
- *GetCurrentThread*
- *GetCurrentThreadId*
- *GetThreadId*
- *OpenThread*
- *ResumeThread*
- *SuspendThread*
- *CreateRemoteThread*

CreateThread

Função usada para criar um *thread* para executar com o espaço de endereçamento virtual do processo

- Especifica o endereço de inicio do *thread*;
- Especifica o tamanho da pilha (default é 1MB);
- Especifica um ponteiro para um argumento para o *thread*.

```
HANDLE CreateThread(
        LPSECURITY_ATTRIBUTES lpThreadAttributes,
        DWORD dwStackSize,
        LPTHREAD_START_ROUTINE lpStartAddr,
        LPVOID lpThreadParm,
        DWORD dwCreationFlags,
        LPDWORD lpThreadId
        );
```

Parâmetros: CreateThread

CreatesThread	Cria um novo *thread* no processo pai
security	Atributos de Segurança, 0 = mesmo do pai
stack	Tamanho inicial da pilha thread, 0 = 1M
function	Nome da função que o thread inicia
param	Valor de 4 bytes que a função do thread recebe na inicialização
flags	0 ou CREATE_SUSPENDED
threadID	Um ponteiro para uma DWORD para receber o ID do thread
	Retorna um *handle* para o novo *thread*, ou 0 se erro.

Esperando o Thread Terminar

No Windows quando um processo termina, todos os seus threads também terminam. É necessário que o programa principal (thread primário*) espere pelo termino dos threads que ele criou.*

DWORD WaitForSingleObject (**HANDLE** *hHandle*,
DWORD *dwMilliseconds*);

hHandle – Um manipulador para um objeto;
dwMilliseconds – O intervalo time-out, em milesegundos. Se INFINITE, o intervalo nunca termina.

Esperando Multiplos Threads Terminarem

DWORD WaitForMultipleObjects (DWORD *nCount*,
const HANDLE* *lpHandles*,
BOOL *bWaitAll*,
DWORD *dwMilliseconds*);

nCount - O número de manipuladores de objetos em um array apontado por *lpHandles*;
lpHandles – Um array de manipuladores de objetos;
bWaitAll – Se TRUE, a função retorna quando todos os objetos terminarem, FALSE retorna quando o primeiro deles terminar;
dwMilliseconds – O intervalo time-out, em milesegundos. Se INFINITE, o intervalo nunca termina.

– O número de threads que podem esperar nesta chamada é limitado por MAXIMUM_WAIT_OBJECTS.

– A chamada pode ser usada não somente para processos e threads mas também qualquer objeto do Windows.

(Ver 1_Exemplo_Vetor_Threads)

Passando Parâmetro para o Thread

É possível passar um parâmetro para a função do *thread*, passando um ponteiro para uma variável.

A estrutura deve ser estável. Isto é, ela precisa ser uma variável global ou uma variável local estática.

(Ver 2_Exemplo_Parametro)

Passando Estruturas como Parâmetro para o Thread

Quando precisamos passar mais de um parâmetro para o *thread*, fazemos isso passando uma estrutura que conterá um campo para cada um dos parâmetros que desejamos passar.

A estrutura deve ser estável. Isto é, ela precisa ser uma variável global, uma variável local estática, ou um bloco alocado do *heap*.

A estrutura não pode ser uma variável local apontando para uma função que pode cessar de existir durante a execução do *thread*.

(Ver 3_Exemplo_Multiplos_Parametros)

GetCurrentThread

HANDLE GetCurrentThread (void);
– Retorna o pseudo manipulador não “herdável” do *thread*.

DWORD GetCurrentThreadId (void);
– Obtem o ID do *thread* que executa esta função.

void GetThreadId (HANDLE Thread);
– Obtem o ID de um *thread* específico a partir do seu manipulador.

Suspende e Retoma Execução

DWORD SuspendThread (HANDLE hThread);

– Suspende o *thread* especificado

DWORD ResumeThread (HANDLE hThread);

– Decrementa o contador de "suspenso" do *thread*. Quando o contador é zero, a execução do *thread* é retomada

(Ver 4_Exemplo_Suspend_Resume)

Prioridade e Escalonamento

Threads recebem prioridade relativa à classe de prioridade de seus processos

- IDLE_PRIORITY_CLASS (4)
- BELOW_NORMAL_PRIORITY_CLASS (5)
- NORMAL_PRIORITY_CLASS (9 or 7)
- ABOVE_NORMAL_PRIORITY_CLASS (11)
- HIGH_PRIORITY_CLASS (13)
- REALTIME_PRIORITY_CLASS (24)

Prioridade e Escalonamento

As prioridades dos *threads* são inicializadas relativamente à prioridade básica do processo

- THREAD_PRIORITY_IDLE (-4)
- THREAD_PRIORITY_ABOVE_IDLE (-3)
- THREAD_PRIORITY_LOWEST (-2)
- THREAD_PRIORITY_BELOW_NORMAL (-1)
- THREAD_PRIORITY_NORMAL (0)
- THREAD_PRIORITY_ABOVE_NORMAL (+1)
- THREAD_PRIORITY_HIGHEST (+2)
- THREAD_PRIO RITY_TIME_CRITICAL (+3)

Prioridade e Escalonamento

Processes

– BOOL SetPriorityClass (HANDLE hProcess,
DWORD dwPriorityClass);

– DWORD GetPriorityClass (HANDLE
hProcess);

Threads

– BOOL SetThreadPriority (HANDLE hThread,
int nPriority);

– int GetThreadPriority (HANDLE hThread);

Prioridade e Escalonamento

Exemplo:
Se um processo em *foreground* tem uma prioridade classe normal e um dos seus *threads* "seta" sua prioridade para THREAD_PRIORITY_LOWEST, então o valor de prioridade do thread é 7 (9 – 2 = 7).

O escalonador usa o valor da prioridade de todos os *threads* em existência para determinar qual ganhará o próximo "slice" do tempo da CPU. Os *threads* são escalonados com o round-robin em cada nível de prioridade, e somente quando não existem *threads* em um nível superior é que *threads* de um nível inferior são considerados para execução.

(Ver 5_Exemplo_Prioridade)