Sistemas Operativos
4ª parte - Threads

Prof. José Rogado

jrogado@ulusofona.pt

Prof. Pedro Gama

pedrogama@gmail.com

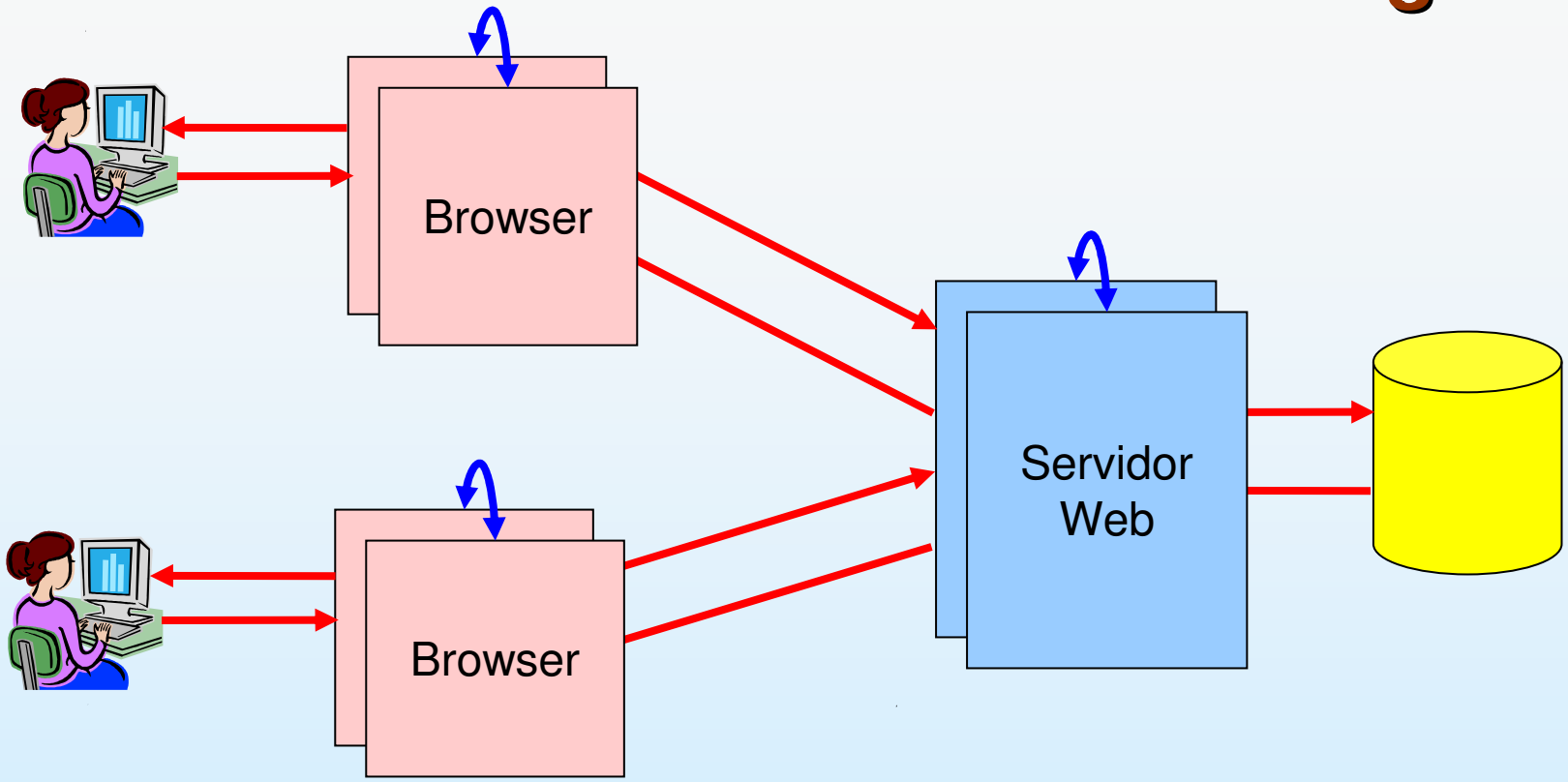
Universidade Lusófona

Adaptação LIG e Notas por Dr. Adriano Couto

Threads

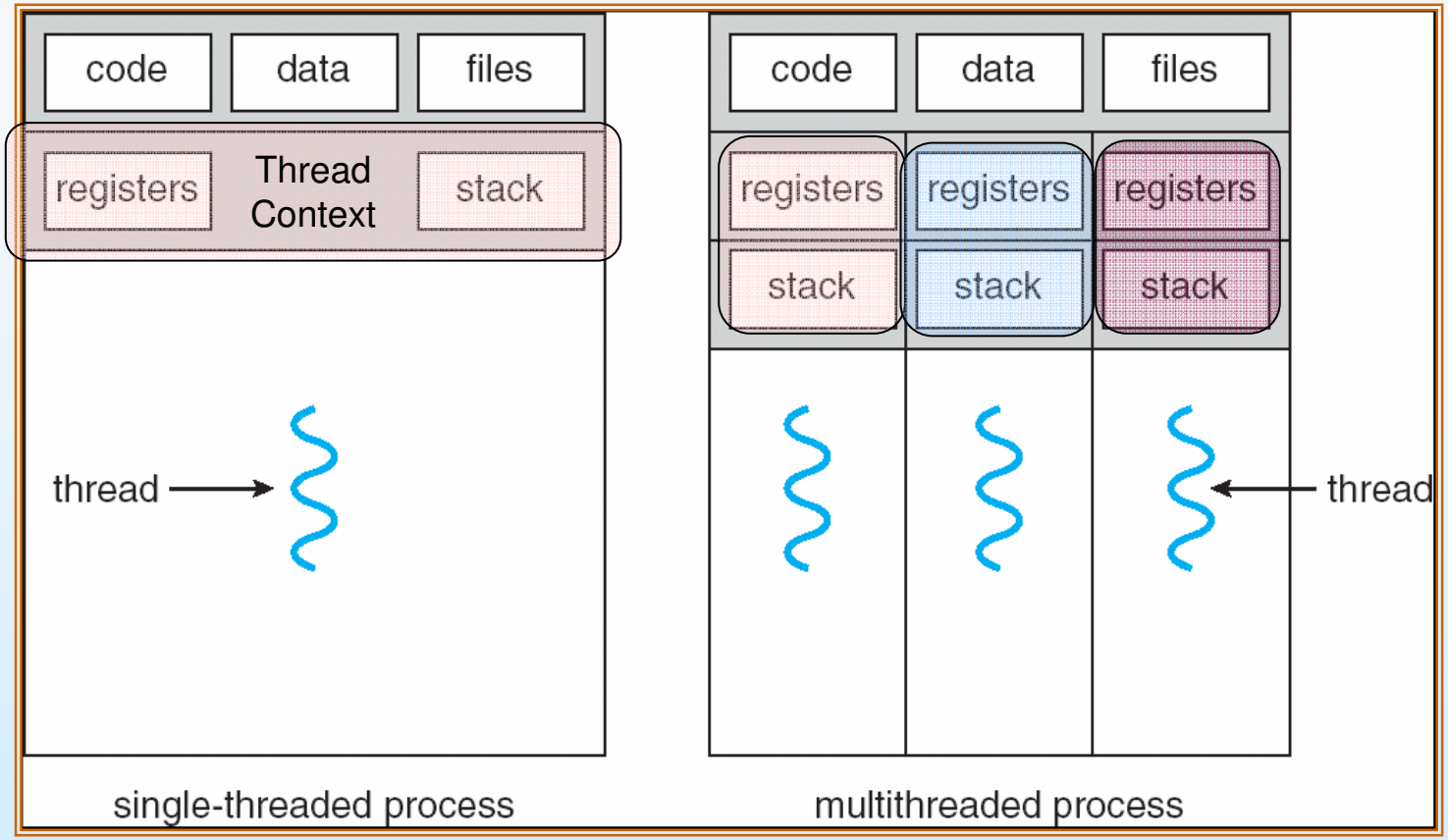
- Objectivos do Capítulo:
  - Introduzir o conceito como pedra basilar paradigma moderno programação concorrente.
  - Referir o interesses dos fabricantes de CPU no paradigma concorrente que os CPU's já suportam.
- Necessidades e Conceito
- Modelos de Multithreading
- Problemas de Threading
- As Pthreads
- Windows XP Threads
- Linux Threads
- Java Threads

Necessidades de Multithreading



- Grande parte das aplicações interactivas e distribuídas precisa de gerir diversos fluxos de execução simultâneos
- Interactividade com utilizador (Browser) ou gestão de várias conecções simultâneas (Web server)
- A utilização de vários processos distintos não é óptima pois implica *overhead* na comunicação de dados e comutação de contexto

Single and Multithreaded Processes

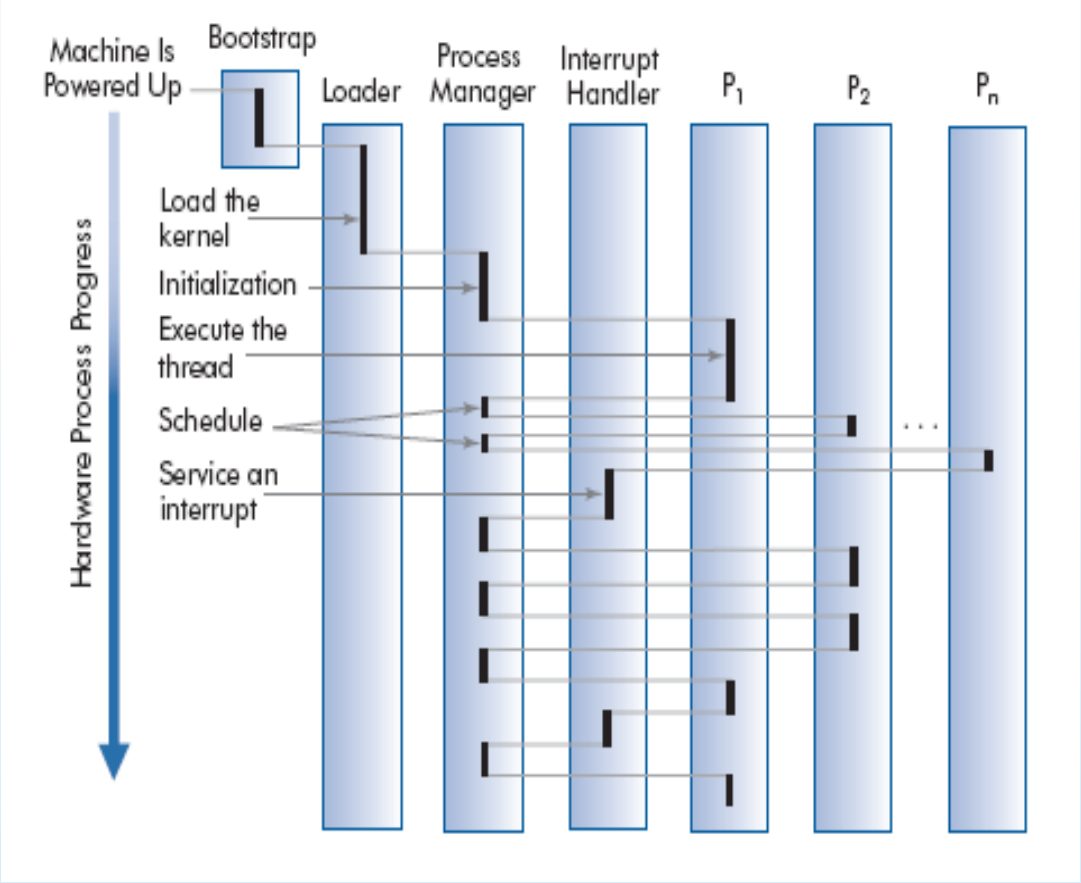


- No modelo multithreaded, em cada processo existem vários fluxos de execução ou *threads*
- As várias *threads* de um processo partilham o espaço de endereçamento, mas têm contextos de execução distintos

Benefícios

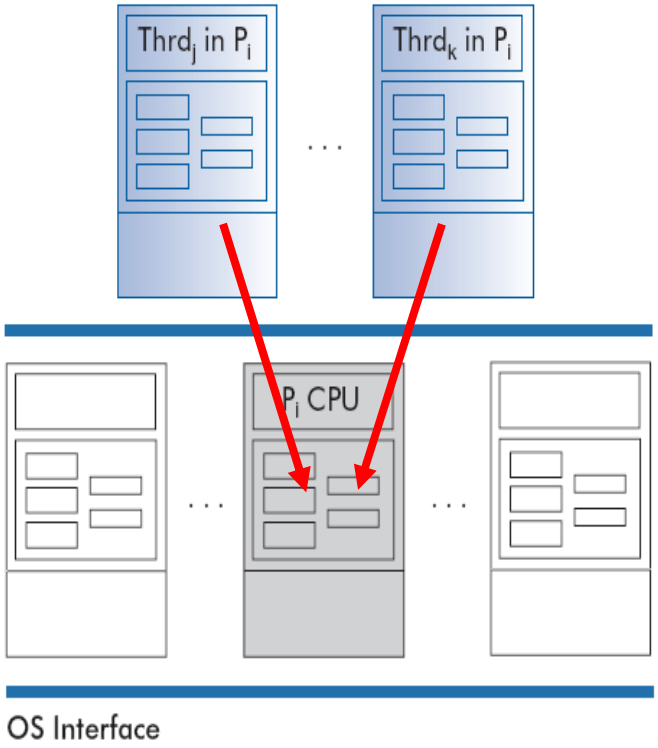
- Maior interactividade
- Menor tempo de resposta
- Partilha de Recursos
- Economia
- Utilização de Arquitecturas Multiprocessador/Hyperthreading

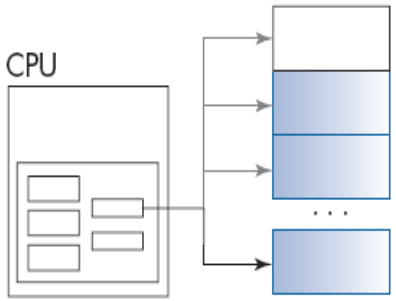
Conceito de *Thread*



- O Scheduler atribui o CPU a cada processo, criando um fluxo de execução, ou *thread*
- No modelo clássico, em cada processo existe um único fluxo de execução

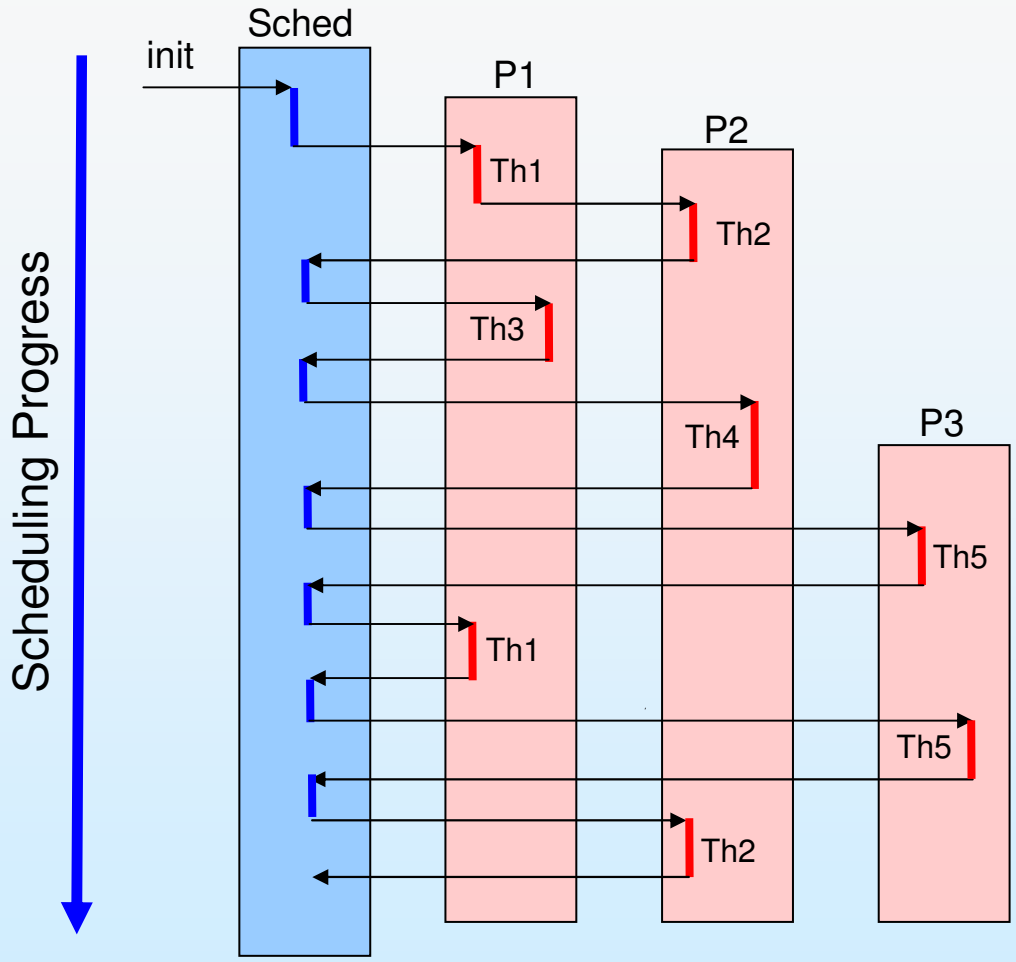
Modelo de Execução de Threads





Source: Operating Systems, Gary Nutt
Copyright © 2004 Pearson Education, Inc.

Multi-threaded Scheduling



- A unidade de *scheduling* deixa de ser o processo e passa a ser a *thread*

Threads em Modo User

- Gestão de Threads é feita numa biblioteca em modo utilizador, incluída na aplicação
- Três principais bibliotecas de threads:
  - threads POSIX (pthreads)
  - threads Win32 (suportadas pelo Kernel)
  - threads Java

Threads em Modo Kernel

- Implementadas no Kernel
- Exemplos
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

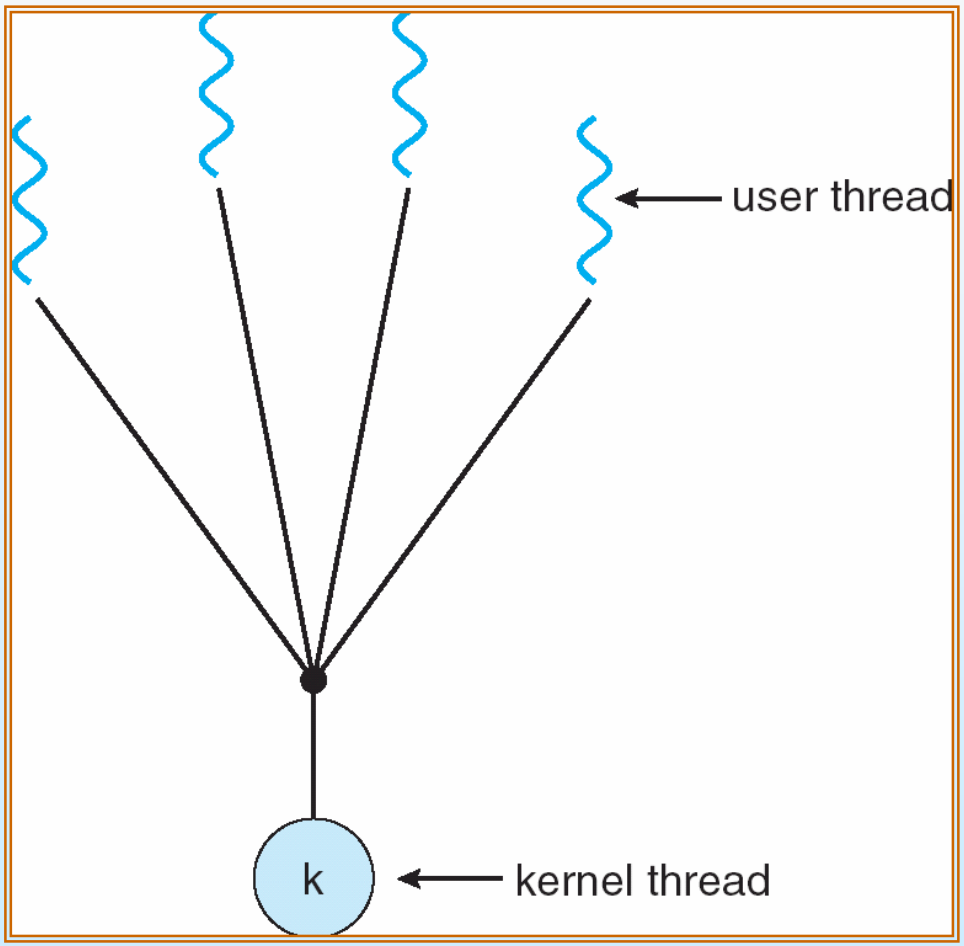
Modelos de Multithreading

- Many-to-One
- One-to-One
- Many-to-Many

Many-to-One

- Várias threads utilizador mapeadas numa única thread do kernel
- Quando a thread bloqueia no kernel, todas as outras threads também bloqueiam
- Para nunca bloquear, o scheduler de threads deve evitar a realização de operações de I/O que possam bloquear sem tomar as devidas precauções
  - Utilização do system call select()
- Exemplos:
  - Solaris Green Threads
  - GNU Portable Threads

Modelo Many-to-One

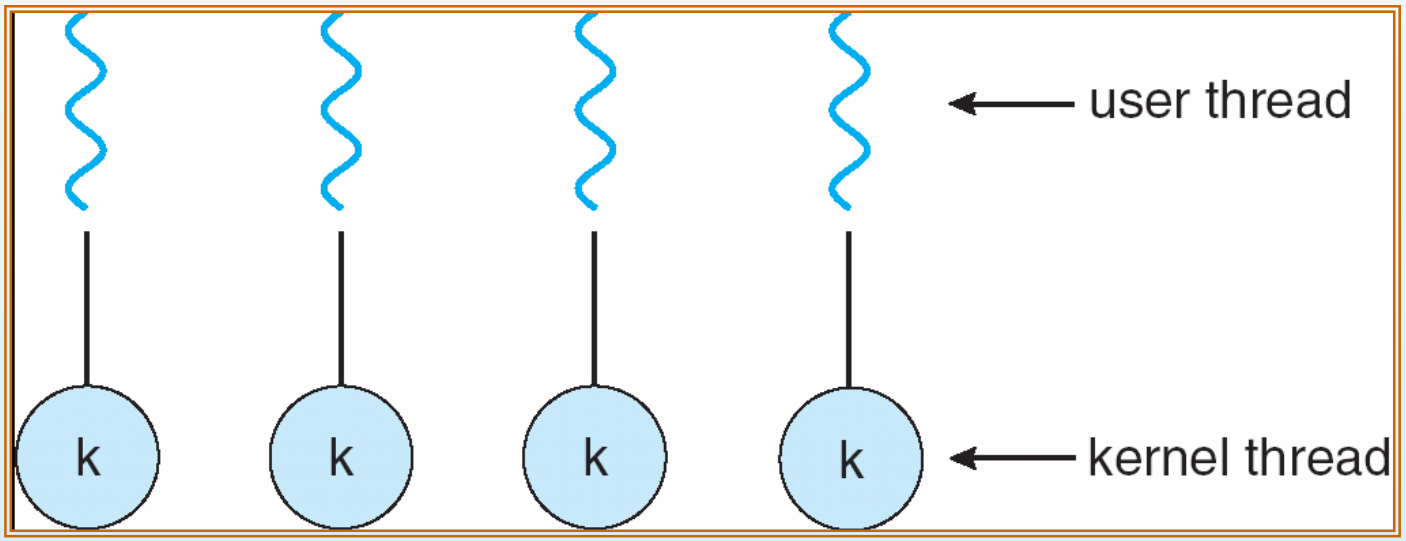


One-to-One

- Cada thread utilizador mapeada para uma thread do kernel
- O *scheduler* do kernel é responsável pela gestão das threads

- Exemplos
  - Windows NT/XP/2000
  - Linux
  - Solaris a partir da versão 9

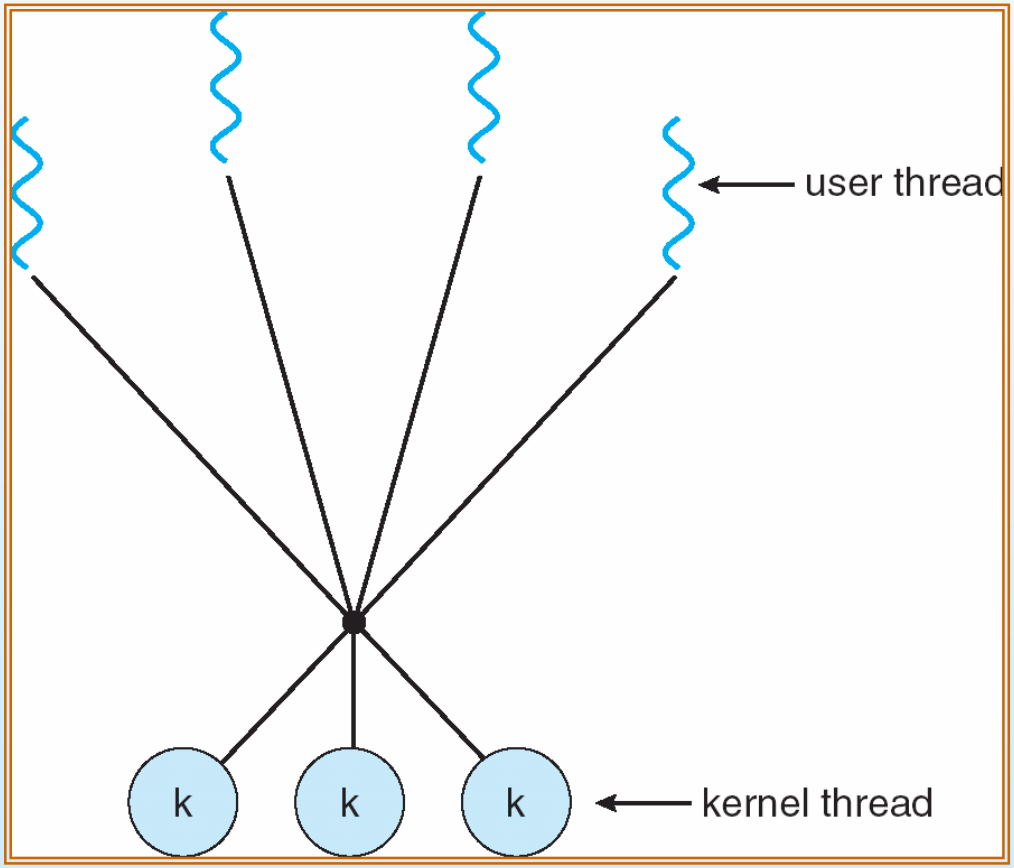
One-to-one Model



Many-to-Many Model

- Permite que um conjunto de threads utilizado sejam mapeadas para várias threads do kernel
- Permite ao Sistema Operativo dosear o número de kernel threads necessárias em cada momento
- Implica uma comunicação específica entre o kernel e o package de threads
  - Scheduler activations
- Exemplos
  - Solaris antes da versão 9
  - Windows NT/2000 com o package *ThreadFiber*

Many-to-Many Model



Problemas de Threading

- Semântica dos system calls **fork()** e **exec()**
- Cancelamento de Threads
- Gestão de Sinais
- Pools de Threads
- Dados específicos das threads
- Activação do scheduler

Semantica de fork() e exec()

- Será que **fork()** deve duplicar todas as *threads* de um processo, ou só aquela que invocou o system call ?
  - Se o processo invoca **exec()**, não é necessário duplicar as *threads* pois o novo processo vai carregar código diferente
  - Se o processo continua a sua execução normal, deve ser idêntico ao processo pai, incluindo no número de *threads*
- Alguns sistemas (ex. Solaris) têm duas versões de fork() uma que duplica as threads, outra que não duplica (fork1)
  - No caso de **fork()** seguido de **exec()** deve ser utilizado **fork1()**
- Em Windows o syscall **CreateProcess()**, utilizado para criar e executar um novo processo, instancia sempre uma só thread (*primary thread*).
  - Não existe equivalente de **fork()**

Cancelamento de Threads

- Cancelar uma thread antes do código ter finalizado
  - Ex: caso de um browser que carrega uma página usando uma thread por frame e é interrompido pelo utilizador no meio do carregamento da página
- Duas aproximações possíveis:
  - **Terminação síncrona** finaliza a thread imediatamente
    - Pode ser problemático se uma thread se encontra no decurso de uma actualização de um recurso e pode criar problemas de coerência
  - **Terminação deferida** deixa a thread alvo verificar periodicamente se deve ser terminada
    - Permite que a thread detecte o facto de ter sido cancelada, e realizar as operações necessárias para o fazer de forma correcta
    - A thread examina a ordem de terminar nos chamados ***cancellation points*** (pthreads)

Gestão de Sinais

- Os Sinais são utilizados em sistemas Unix para notificar um processo de que um dado evento ocorreu
  - Versão utilizador dos interrupts hardware
  - Syscall *signal* **(**int *signum***,** sighandler_t *handler*)
- Um processo declara um ***signal handler*** para gerir sinais
  1. Um sinal é gerado por um evento específico
  2. O sinal é entregue ao processo sob forma da invocação do *signal handler*
  3. O sinal é gerido pelo processo
- Opções em processos com threads: em que thread é executado o handler ?
  - Na thread na qual o sinal faz sentido (caso das excepções)
  - Em todas as threads de um processo – confuso e difícil de gerir
  - Estabelecer de forma explícita que threads recebem que sinais
    - pthread_sigmask() estabelece os sinais para cada thread
    - sigwait() espera pelos sinais (*cancellation point*)

Pools de Threads

- Criação de um número de threads por antecipação
  - As threads são guardadas numa *pool* esperando pelo necessidade de actuarem
- Vantagens:
  - É mais rápido servir um pedido com uma thread existente do que criar uma nova
  - Permitir que o numero de threads numa aplicação seja limitado a um número pré determinado
  - Em caso de necessidade a aplicação pode alocar mais threads, caso tenha direitos para o efeito.
  - Pode também diminuir o número de threads na pool em função da carga.
- Exemplos
  - Windows: *QueueUserWorkItem (Function)*
    - Permite executar uma função por uma thread da pool
  - Java: package *ThreadPool.java*

Scheduler Activations

- Os modelos M:M e Two-level implicam a existência de mecanismos de comunicação entre o *kernel* e o *package* de threads
  - O *kernel* previne o *package* de que uma *thread* vai esperar por um evento
  - O *kernel* previne o *package* de que ocorreu o evento pelo qual uma *thread* esperava
  - O *scheduler* do *package* de threads pode assim gerir a alocação de threads utilizado as threads do *kernel*.
- Este conjunto de comunicações do *kernel* para o gerenciador de threads do *package* é designado por ***Scheduler Activations Upcalls***
- Esta comunicação permite a uma aplicação manter um número correcto de threads *kernel* para executar as threads utilizador

As pthreads

- Uma API standard POSIX (IEEE 1003.1c) para a criação de threads e sua sincronização
- A API especifica o comportamento da biblioteca de threads, a implementação é da responsabilidade da biblioteca
- Comum aos sistemas UNIX (Solaris, Linux, Mac OS X)
- Algunas funções:
  - **`pthread_create`**: criação de uma thread
  - **`pthread_join`**: espera que uma thread acabe
  - **`pthread_exit`**: finalização da thread
  - **`pthread_mutex_init`**: criação de um mutex
  - **`pthread_mutex_lock`**: aquisição do mutex
  - **`pthread_mutex_try_lock`**: aquisição não bloqueante
  - **`pthread_mutex_unlock`**: libertação do mutex

Exemplo:

```c
#include <stdio.h>
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <pthread.h>

int file, mythread(int );

const char *msg[] = {
  "Thread #1 ",
  "Thread #2 "
};
#define MAXTHREADS 2

main(int argc, char *argv[])
{
   int ret, i;
   char buffer[64] = {"0"};
   pthread_t thread[MAXTHREADS];

   file = open (argv[1], O_RDWR);
   if (file < 0) {
     perror("file");
     exit(1);
   }
   for (i = 0; i < MAXTHREADS; i++) {
     ret = pthread_create(&thread[i], NULL, (void *)(&mythread), (void *)i);
     if (ret) {
       printf("ret %d\n", ret);
       perror("thread create");
       exit(1);
     }
   }
   printf("Main Thread\n");
   for (i = 0; i < MAXTHREADS; i++)
     pthread_join(thread[i], NULL);

   lseek(file, 0, SEEK_SET); // Reset the file I/O pointer !!

   ret = read(file, buffer, sizeof(buffer));
   if (ret < 0) {
     perror("read");
     exit(1);
   }
   printf("File Content:\n%s\n", buffer);
   printf("%s\n", buffer+strlen(buffer)+1);
}
```

```c
mythread(int i)
{
  int ret;

  printf("I am %s\n", msg[i]);
  ret = write(file, msg[i], strlen(msg[i]));
  if (ret < 0)
     perror("write");
}
```

Criação das threads

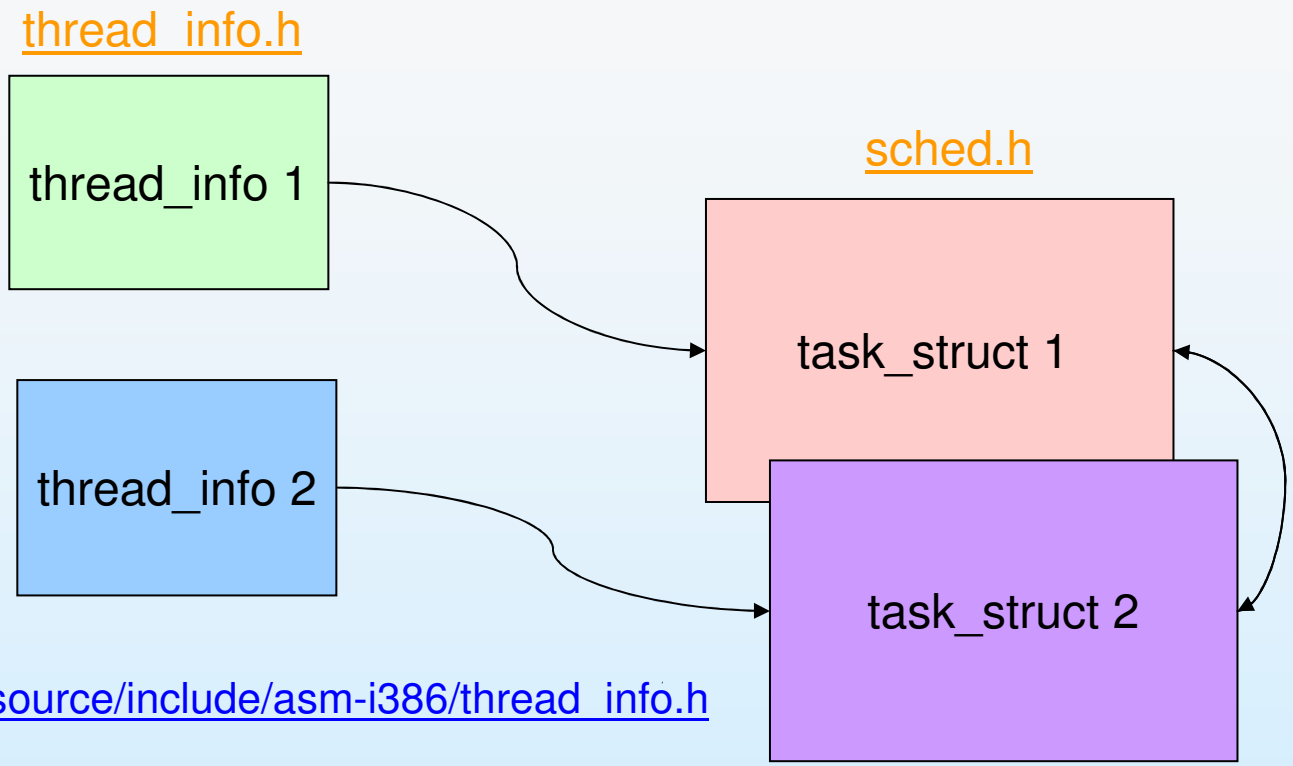
Espera

Código da thread

Linux Threads

- Em Linux, uma nova thread cria um novo fluxo de execução que partilha todos os recursos
  - Internamente não há distinção entre thread e processo
  - Implementação one-to-one
- A criação de threads é feita através do syscall **clone()** que permite implementar fork e thread_create
  - Cria um novo fluxo de execução que pode ou não partilhar todos os recursos com o precedente
- No caso de partilha total, obtém-se uma nova thread no "mesmo" processo com uma nova pilha
  - `clone(SHARE_VM|SHARE_FILES 0)=> thread_create`
- No caso de duplicação, obtém-se um novo processo com um espaço de endereçamento copiado mas distinto
  - `clone(COPY_VM, 0)` => fork
- Ambas as funcionalidades são implementadas por uma rotina comum:
  - `do_fork()`
  - http://lxr.linux.no/source/kernel/fork.c#L1356

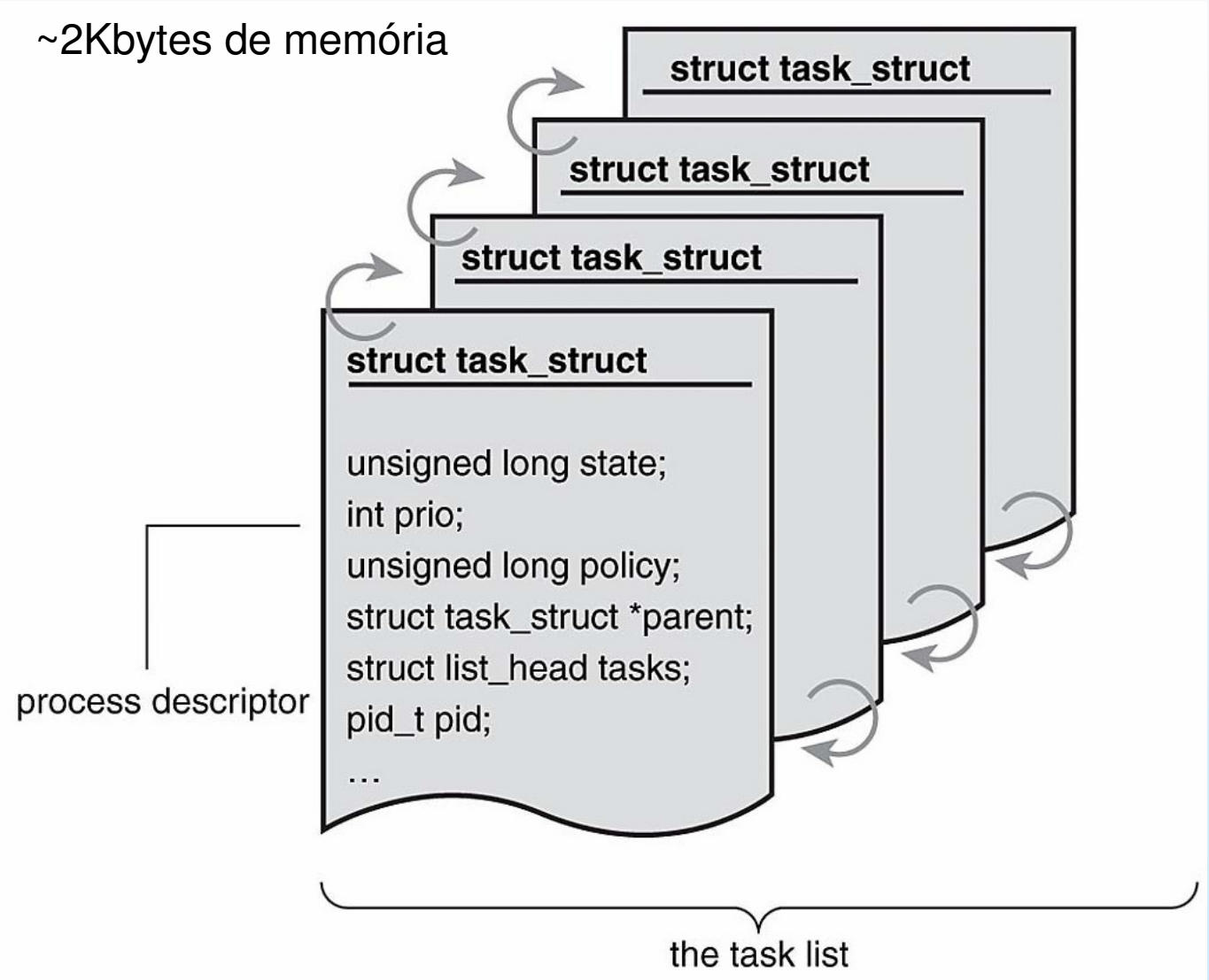
Estruturas de Dados em Linux



http://lxr.linux.no/source/include/asm-i386/thread_info.h

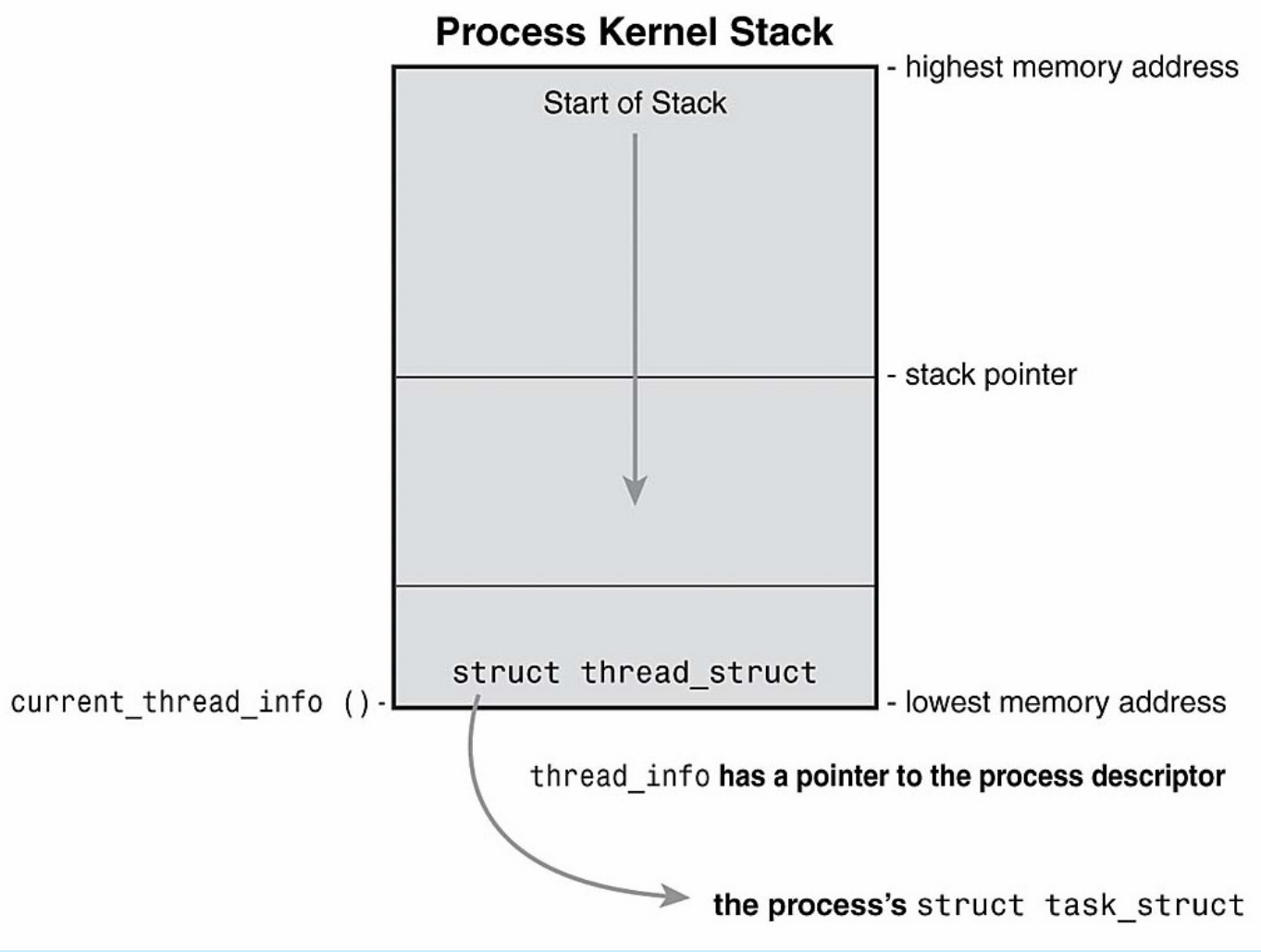
http://lxr.linux.no/source/include/linux/sched.h#L821

A Task List



Source:
Linux kernel Development, Robert Love
Novell Press, 2005.

Localização da thread_struct



Source:
Linux kernel Development, Robert Love
Novell Press, 2005.

Threads Windows XP

- Implementação one-to-one
- Cada descritor de thread contém:
  - Um identificador: thread id
  - Cópia dos registos
  - 2 pilhas: utilizador e supervisor
  - Dados privados
- Os registos, as pilhas e os dados privados são designados pelo contexto da thread
- As estruturas de dados de uma *thread* incluem:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

API Threads do Windows

Function	Description
AttachThreadInput	Attaches the input processing mechanism of one thread to that of another thread.
CreateRemoteThread	Creates a thread that runs in the virtual address space of another process.
CreateThread	Creates a thread to execute within the virtual address space of the calling process.
ExitThread	Ends the calling thread.
GetCurrentThread	Retrieves a pseudo handle for the current thread.
GetCurrentThreadId	Retrieves the thread identifier of the calling thread.
GetExitCodeThread	Retrieves the termination status of the specified thread.
GetThreadId	Retrieves the thread identifier of the specified thread.
GetThreadIOPendingFlag	Determines whether a specified thread has any I/O requests pending.
GetThreadPriority	Retrieves the priority value for the specified thread.
GetThreadPriorityBoost	Retrieves the priority boost control state of the specified thread.
GetThreadTimes	Retrieves timing information for the specified thread.
OpenThread	Opens an existing thread object.
QueryIdleProcessorCycleTime	Retrieves the cycle time for the idle thread of each processor in the system.
QueryThreadCycleTime	Retrieves the cycle time for the specified thread.
ResumeThread	Decrements a thread's suspend count.
SetThreadAffinityMask	Sets a processor affinity mask for the specified thread.
SetThreadIdealProcessor	Specifies a preferred processor for a thread.
SetThreadPriority	Sets the priority value for the specified thread.

http://msdn2.microsoft.com/en-us/library/ms684847.aspx

Exemplo:

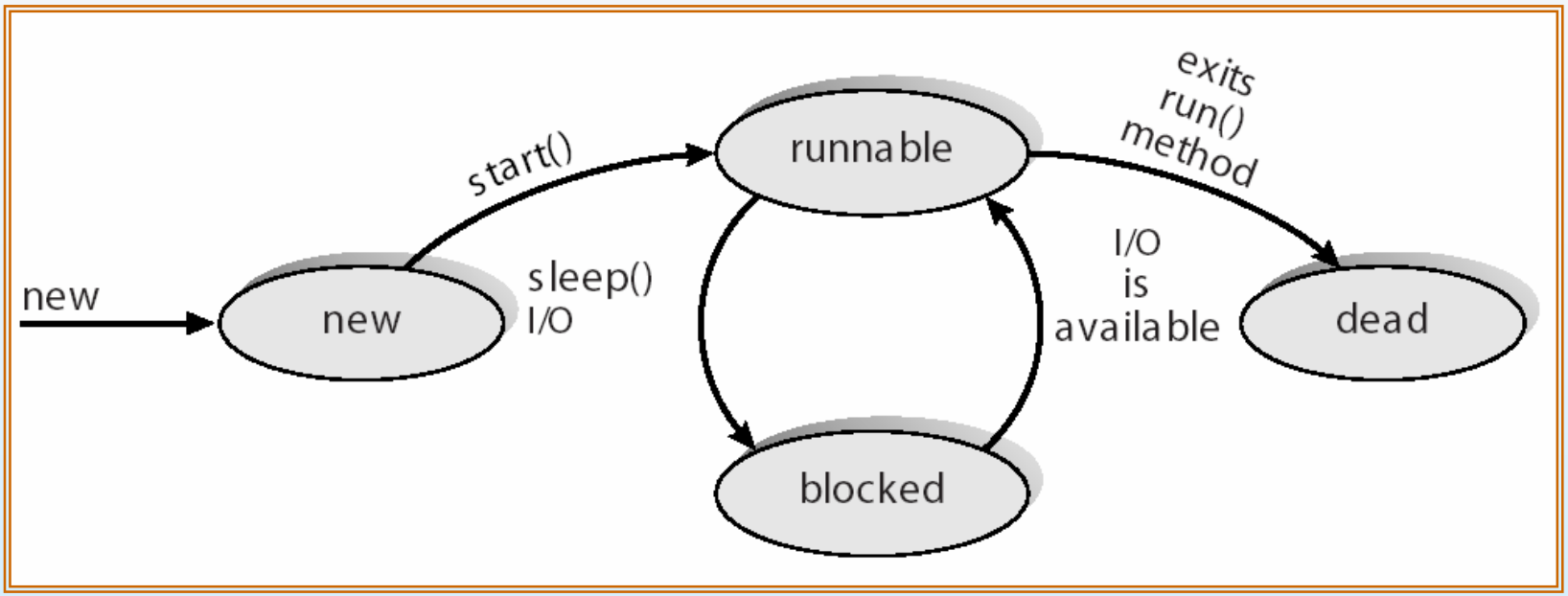
```c
#include <windows.h>
#include <tchar.h>
#include <string.h>

#define MAX_THREADS 3
#define BUF_SIZE 256

typedef struct MyData {
    int val1;
    int val2;
} MYDATA, *PMYDATA;

DWORD WINAPI MyThread( LPVOID lpParam )
{
    HANDLE hStdout;
    PMYDATA pData;

    TCHAR msgBuf[BUF_SIZE];
    size_t cchStringSize;
    DWORD dwChars;

    hStdout = GetStdHandle(STD_OUTPUT_HANDLE);
    if( hStdout == INVALID_HANDLE_VALUE )
        return 1;

    // Cast the parameter to the correct data type.

    pData = (PMYDATA)lpParam;

    // Print the parameter values.

    snprintf(msgBuf,
        BUF_SIZE, TEXT("Parameters = %d, %d\n"),
        pData->val1, pData->val2);

    cchStringSize = strnlen(msgBuf, BUF_SIZE);

    WriteConsole(hStdout, msgBuf, cchStringSize,
                                  &dwChars, NULL);

    return 0;
}
```

```c
int _tmain()
{
    PMYDATA pData;
    DWORD dwThreadId[MAX_THREADS];
    HANDLE hThread[MAX_THREADS];
    int i;

    // Create MAX_THREADS worker threads.

    for( i=0; i<MAX_THREADS; i++ ) {
        // Allocate memory for thread data.
        pData = (PMYDATA) HeapAlloc(GetProcessHeap(),
                            HEAP_ZERO_MEMORY, sizeof(MYDATA));
        if( pData == NULL )
            ExitProcess(2);

        // Generate unique data for each thread.
        pData->val1 = i;
        pData->val2 = i+100;

        hThread[i] = CreateThread(
            NULL,               // default security attributes
            0,                  // use default stack size
            MyThread,           // thread function
            pData,              // argument to thread function
            0,                  // use default creation flags
            &dwThreadId[i]);    // returns the thread identifier

        // If failure, close existing thread handles,
        // free memory allocation, and exit.
        if (hThread[i] == NULL) {
           for(i=0; i<MAX_THREADS; i++) {
              if (hThread[i] != NULL)
                 CloseHandle(hThread[i]);
           }
           HeapFree(GetProcessHeap(), 0, pData);
           ExitProcess(i);
         }
    }
    // Wait until all threads have terminated.
    WaitForMultipleObjects(MAX_THREADS, hThread, TRUE, INFINITE);

    // Close all thread handles and free memory allocation.
    for(i=0; i<MAX_THREADS; i++) {
        CloseHandle(hThread[i]);
    }
    HeapFree(GetProcessHeap(), 0, pData);
    return 0;
}
```

Java Threads

- As threads Java são geridas pela JVM
- Podem ser criadas:
  - Extendendo a classe Thread
  - Implementando a interface Runnable
  - Ver em java.sun.com/j2se/1.5.0/docs/api/

Java Thread States



Fim das Threads