Sistemas Operativos

Licenciatura Engenharia Informática
Licenciatura Engenharia Computacional

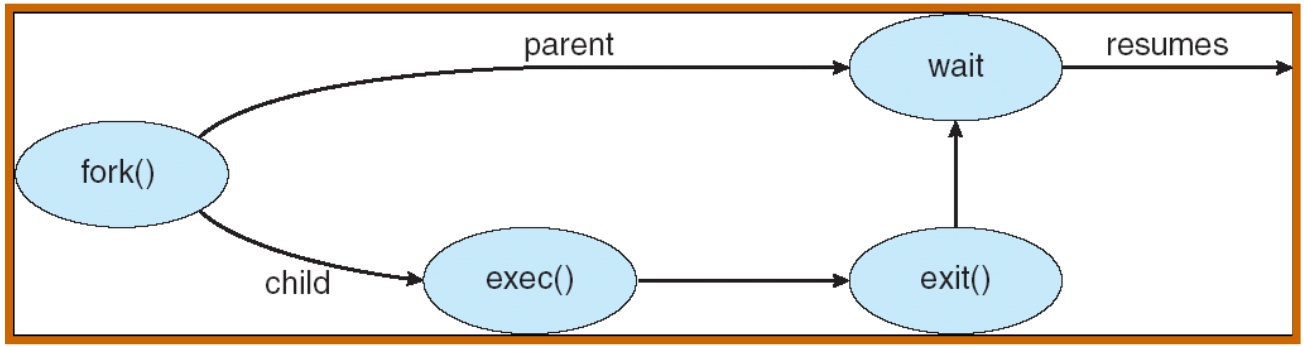
Ano letivo 2023/2024

Nuno Lau (nunolau@ua.pt)

Processo

- Programa em execução
- Criar um processo
  - Inicialização do Sistema
  - Execução de chamada ao sistema por processo em execução
  - Pedido do utilizador para criar novo processo
  - Início de um *batch script*
- Processos podem correr em:
  - *foreground*: interage com utilizador
  - *background*: executa sem interação, *daemon*

Criação de processos



POSIX *Input / Output*

- Abrir e fechar ficheiros

```
int open(const char *path, int oflag, .../*,mode_t mode */);
int close(int filedes);
```

- Ler / Escrever

```
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t nbytes);
```

- Duplicar *file descriptors*

```
int dup (int oldfd);
int dup2 (int oldfd, int newfd);
```

Leitura

```c
#include <sys/types.h>
#include <unistd.h>

#include <stdio.h>

#define BUFSIZE 1024

int main(int argc, char *argv[])
{
    int fd, nr;
    char buf[BUFSIZE];

    nr = read(0, buf, BUFSIZE);

    printf("read bytes=%d buf=%.*s\n", nr, nr, buf);

    return 0;
}
```

Escrita

```c
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

#include <stdio.h>

int main(int argc, char *argv[])
{
    int fd;

    fd = open("write1.txt", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);

    write(fd, "message1\n",9);

    close(fd);

    return 0;
}
```

Redirecionamento *output*

```c
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

#include <stdio.h>

int main(int argc, char *argv[])
{
    int fd;

    fd = open("rdex1.txt", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);

    dup2(fd, 1); // close 1, then make 1 refer to same file as fd
    close(fd);   // close fd

    execlp("ls","ls",NULL);

    return 0;
}
```

Redir *output* no processo filho

```c
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>

#include <stdio.h>

int main(int argc, char *argv[])
{
    int fd;

    switch( fork() ) {
      case  0: // child
        fd = open("redirforkexec1.txt", O_WRONLY|O_CREAT, S_IRUSR|S_IWUSR);

        dup2(fd, 1); // close 1, then make 1 refer to same file as fd
        close(fd);   // close fd

        execlp("ls","ls",NULL); // exec ls

        break;
      default:  //parent
        printf("pid=%d\n",getpid());
        break;
    }

    printf("END.\n");

    return 0;
}
```

Threads

→ A mudança de contexto entre processos é algo "pesado", precisamos de uma solução!
↳ Muda PC
↳ Registos
↳ Comunicação entre processos é chata

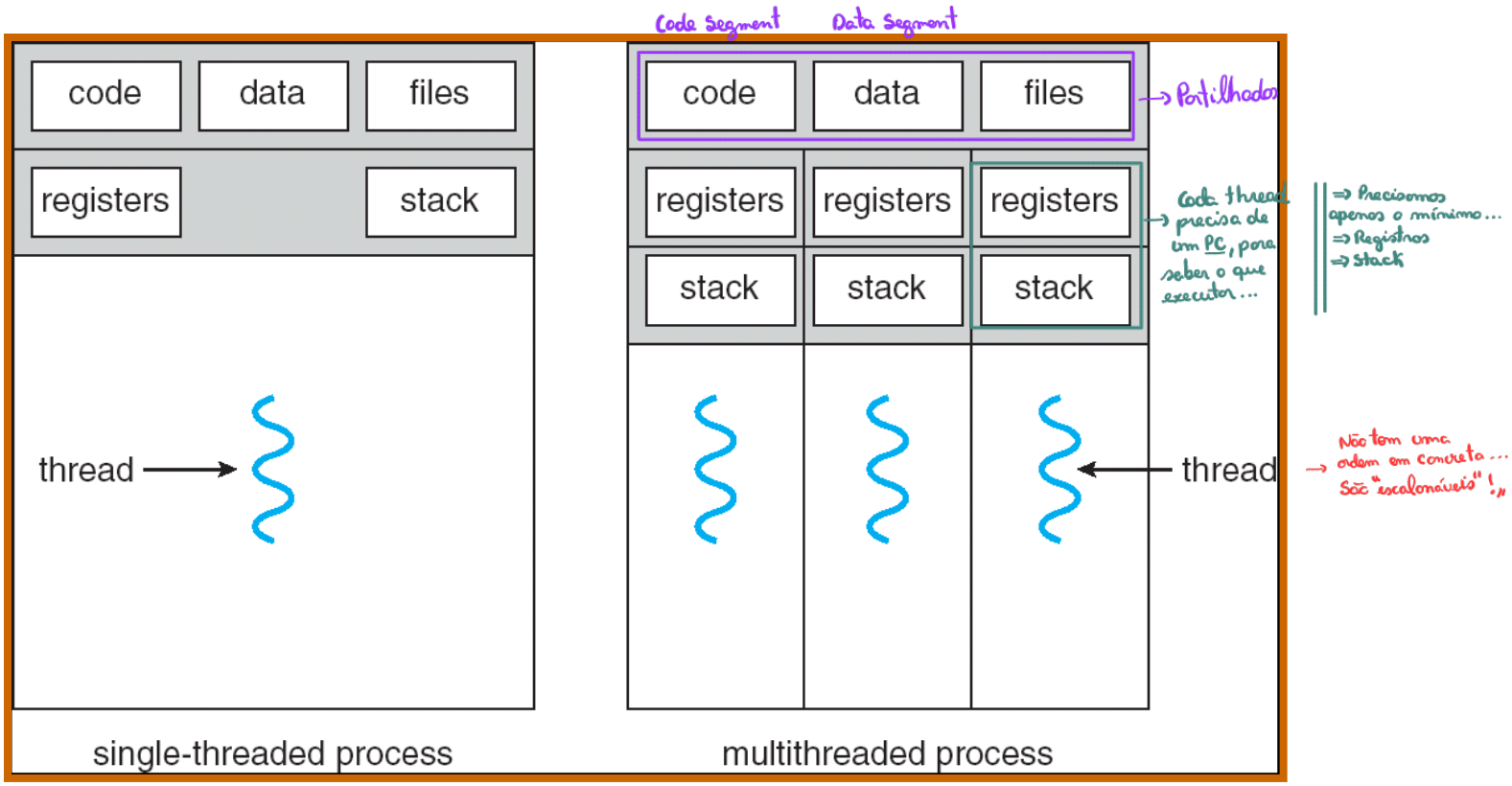
- Programas têm geralmente de executar diversas atividades distintas → Num mesmo processo ter diferentes execuções sequenciais!
- Usando *threads*, o programador pode desenvolver o programa como um conjunto de fluxos de execução sequenciais, um para cada atividade → Em paralelo se tivermos CPU's suficientes ⇒ Estes fluxos de execução são "escalonáveis" pelo CPU
- Cada *thread* comporta-se como tendo o seu processador próprio.
- Todas as *threads* do mesmo processo partilham espaço de endereçamento (memória)

↳ Têm o mesmo código mas fluxos diferentes

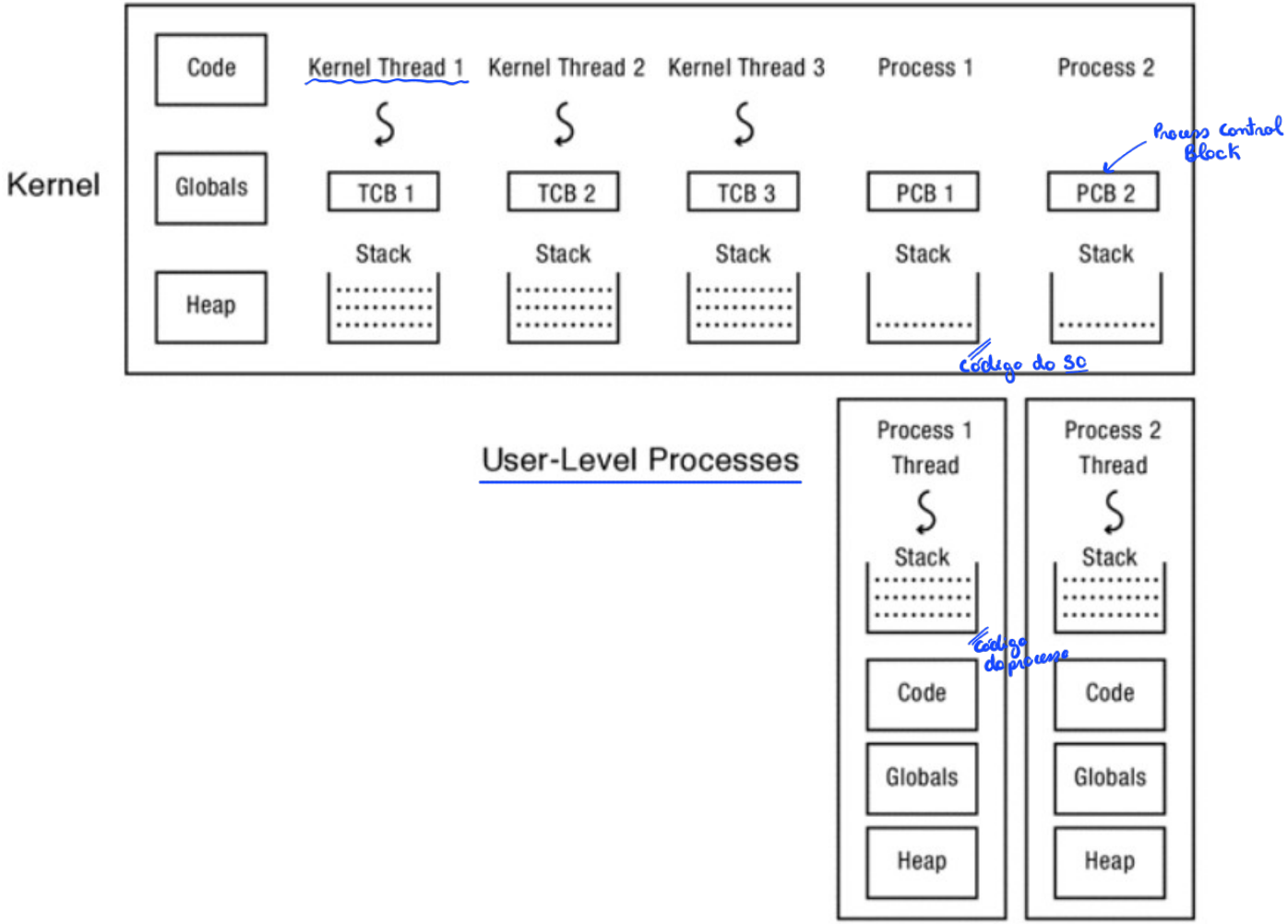
→ e.g.: autocomplite num editor de texto

1 Thread: escrever no editor
2 Thread: lê o texto e procura palavras
} Os 2 querem utilizar var. globabais partilhadas entre os 2 Threads

Processos *Single* e *Multi threaded*



Kernel e User threads



Processos e *Threads*

Per-process items	Per-thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

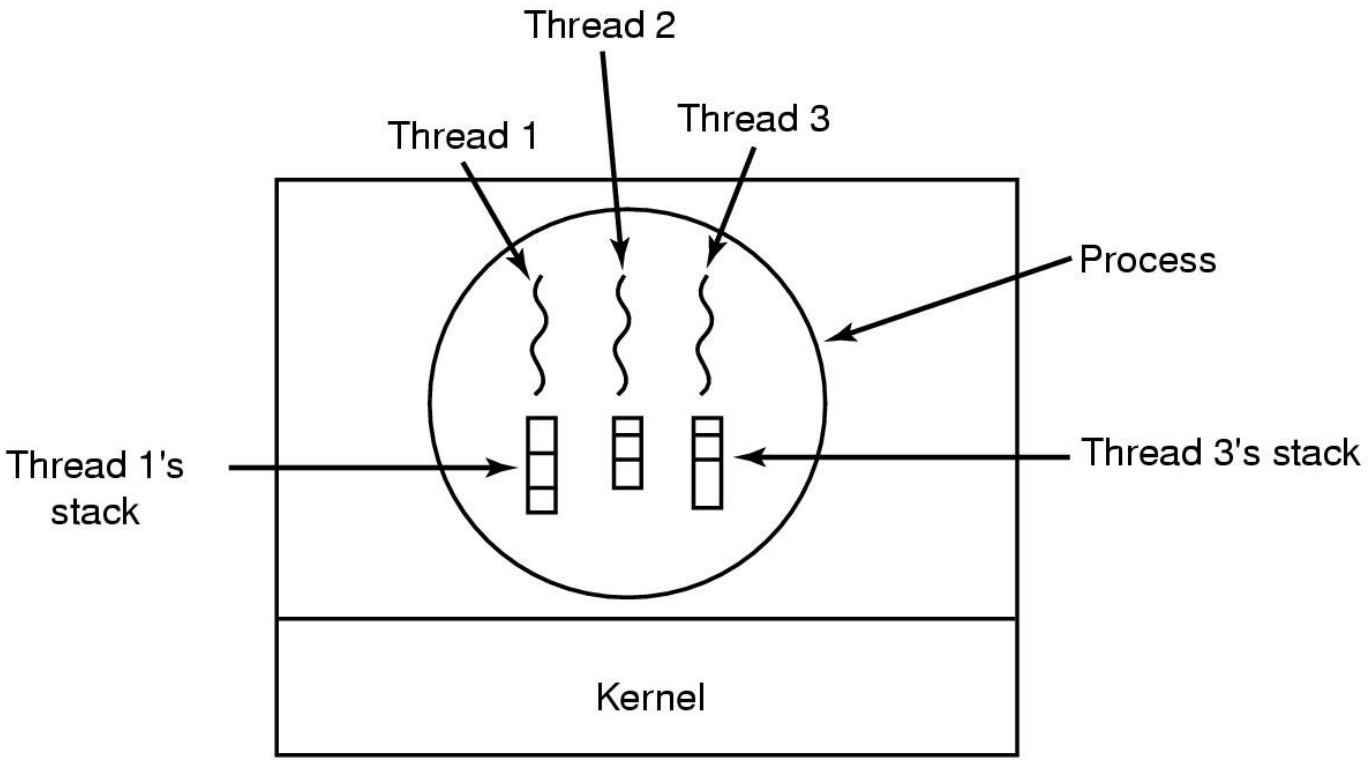
=> Contexto

Cada uma tem o seu próprio fluxo ?

Ready
Waiting
Finish
...

Cada *thread* tem a sua *stack*

↳ Mas partilham o mesmo processo
→ espaço de endereçamento
→ ...



Servidor Web Multithreaded

- Num servidor web, cada pedido de página pode ser processado numa *thread* separada
- Há uma (*dispatcher*) *thread* que recebe todos os pedidos e os distribui pelas (*worker*) *threads*

```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

Dispatcher thread

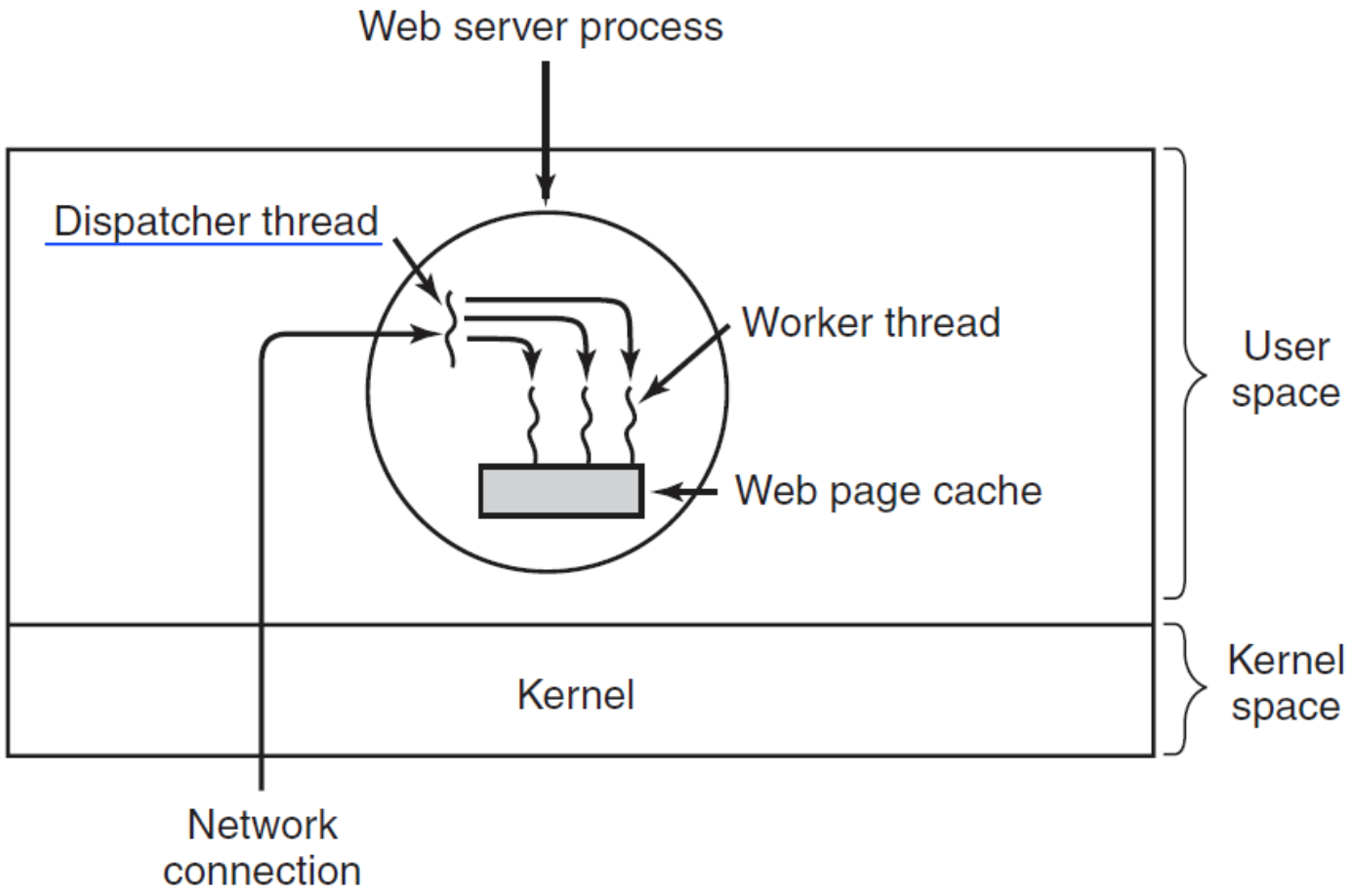
→ vai recolhendo pedidos
→ atribui o pedido a um worker

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

Worker threads

→ esperam por trabalho
→ ... processamento
→ envia a página

Servidor Web Multithreaded



Vantagens das *threads*

- Estrutura do programa/Modularidade
  → Conjuntos de fluxos sequenciais ...
- Responsividade ⇝ e.g.: à espera que escreves no teclado
- Partilha de recursos ⇒ Ficheiros abertos, memória var. globais
- Melhor desempenho ⇝ Mesmo com apenas 1 processador, pois esperar por I/O é muito comum e demora "muito" tempo, e o escalonamento aproveitaria o CPU
- Utilização de arquitecturas multi-processador → Processador com vários cores!

  } Existem até cores que aceitam vários threads...

  1 thread ⟶ 1 core
  Desperdício de recursos

  ⊕ Threads ⟶ ⊕ cores
  Aproveita todos os cores ...

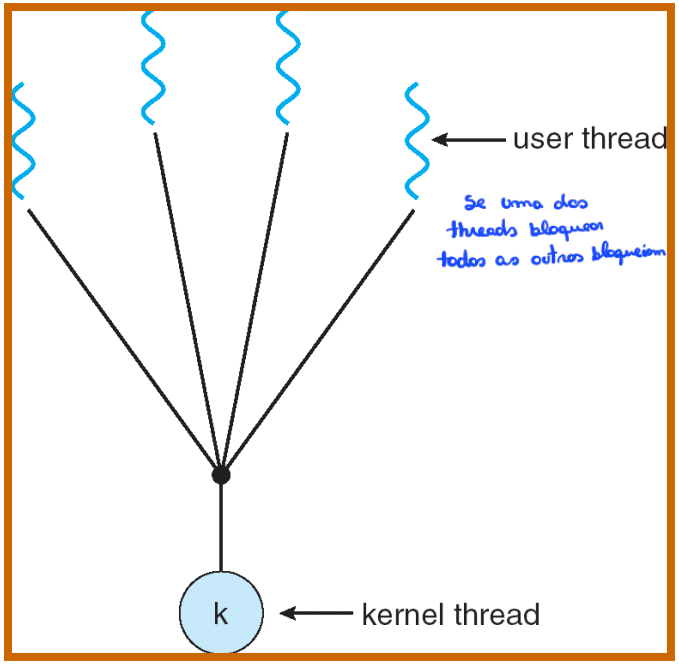
  e.g.: Uma thread a processar a parte de cima da imagem e outra a parte de baixo

Suporte à implementação

- *User threads* → Têm problemas
  - Gestão das *threads* é realizada por uma biblioteca que corre em modo de utilizador
- Kernel *threads*
  - Gestão das *threads* é realizada directamente pelo kernel
  - Windows XP/2000, Solaris, Linux, Tru64 UNIX, Mac OS X

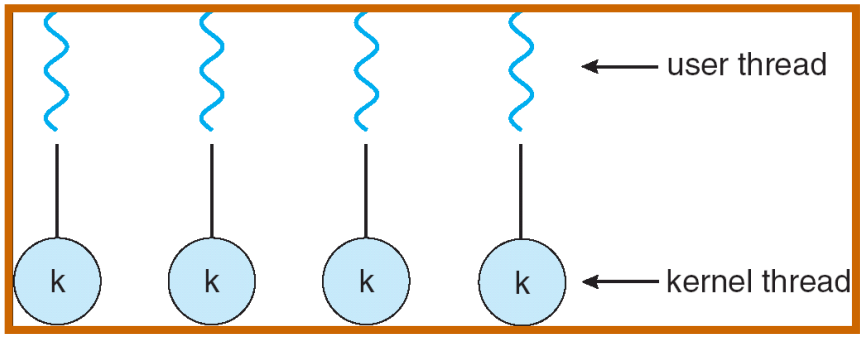
Modelos *Multithreading*

- *Many-to-one*
  - Várias *threads* do utilizador mapeadas numa *thread* do kernel
  - Exemplos
    - Solaris Green Threads
    - Gnu Portable Threads
  - Se uma *thread* bloqueia todas bloqueiam
  - Não tira partido de vários processadores



Modelos *Multithreading*

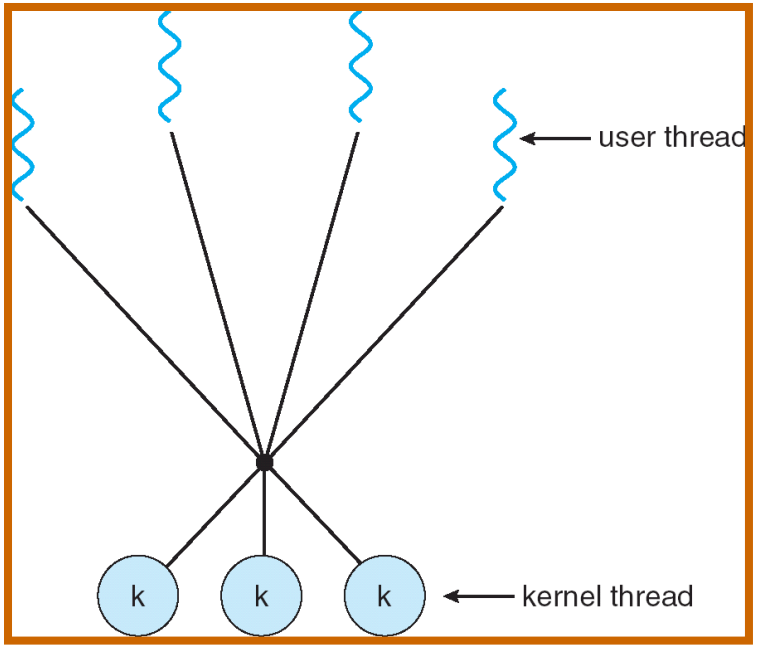
- *One-to-one* (Mais comum)
  - Cada *thread* do utilizador mapeada numa *thread* do kernel
  - Exemplos
    - Windows NT/XP/2000
    - Linux
    - Solaris 9 e post.
  - Número total de *threads* do sistema pode ser limitado



=> Pode ser um problema...

Modelos *Multithreading*

- *Many-to-many*
  - Várias *threads* do utilizador mapeadas em várias *threads* do kernel
  - Exemplos
    - Solaris antes de 9
    - Windows NT/2000 com ThreadFiber
  - Número de *threads* do kernel pode variar com aplicação e com sistema



// Uma solução...

Pthreads

- POSIX standard para a criação e sincronização de *threads*
- API define comportamento, mas não implementação
- Comum em sistemas UNIX (Linux, Mac OS X)

POSIX *Threads*

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit → A thread que cria costuma esperar ...
Pthread_yield	Release the CPU to let another thread run → Desiste do CPU
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Criar POSIX *Threads*

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

```c
#include <stdio.h>
#include <pthread.h>

#define NUM_THREADS 5

void *PrintMsg(void *threadid) {
   long tid;
   tid = (long)threadid;
   printf("Hello World! Thread ID, %d\n", tid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[]) {
   pthread_t threads[NUM_THREADS];
   int rc;
   int i;

   for( i = 0; i < NUM_THREADS; i++ ) {
      printf( "main() : creating thread, %d\n",i);
      rc = pthread_create(&threads[i], NULL, PrintMsg, (void *)i);
      if (rc) {
         printf("Error: unable to create thread, %d\n", rc);
         exit(1);
      }
   }
   pthread_exit(NULL);
}
```