

# Sistemas Operacionais

## Gestão de tarefas - implementação de tarefas

Prof. Carlos Maziero

DInf UFPR, Curitiba PR

Fevereiro de 2019

# Conteúdo

- 1 Contextos
- 2 Trocas de Contexto
- 3 Processos
- 4 Threads
- 5 Programando com *threads*
- 6 Usar *threads* ou processos?

# Contexto de execução

Cada tarefa possui um estado interno:

- Posição atual da execução (*program counter, stack pointer*)
- Valores das variáveis e áreas de memória
- Arquivos abertos, conexões de rede, ...

## Contexto

É o estado interno da tarefa, que se modifica conforme a execução da tarefa evolui.

# Informações do contexto

- Registradores do processador:
  - PC - Contador de programa, indica a posição da execução
  - SP - Apontador de pilha, topo da pilha de execução
  - Flags (nível usuário/núcleo, status, etc).
  - Demais registradores (acumulador, etc)
- Áreas de memória usadas pela tarefa
- Recursos em uso:
  - Arquivos abertos
  - Conexões de rede
  - Semáforos
  - ...

# Task Control Block

Descritor uma tarefa:

- Identificador da tarefa (geralmente inteiro)
- Estado da tarefa (nova, pronta, executando, ...)
- Informações de contexto (registradores, etc)
- Recursos usados pela tarefa (arquivos abertos, ...)
- Informações de contabilização (data de início, tempo de processamento, volume de dados lidos/escritos, etc.).
- Outras informações (prioridade, proprietário, etc.).

Geralmente implementado como um struct

## TCB no Linux 1.0 (include/linux/sched.h)

```

1 struct task_struct {
2     volatile long state;          /* -1 unrunnable, 0 runnable, >0 stopped */
3     long counter;
4     long priority;
5     unsigned long signal;
6     unsigned long blocked;       /* bitmap of masked signals */
7     unsigned long flags;        /* per process flags, defined below */
8     int errno;
9     int debugreg[8]; /* Hardware debugging registers */
10    ...
11    struct task_struct *next_task, *prev_task;
12
13    struct sigaction sigaction[32];
14    int exit_code, exit_signal;
15    ...
16    unsigned long start_code, end_code, end_data, start_brk, brk,
17        start_stack, start_mmap;
18    unsigned long arg_start, arg_end, env_start, env_end;
19    ...
20    struct wait_queue *wait_chldexit; /* for wait4() */
21    unsigned short uid, euid, suid;
22    unsigned short gid, egid, sgid;
23    ...
24    int tty;                       /* -1 if no tty, so it must be signed */
25    unsigned short umask;
26    struct inode * pwd;
27    ...
28    struct vm_area_struct *stk_vma;
29 };
  
```

# Troca de contexto

Consiste em:

- 1 Salvar o contexto da tarefa atualmente em execução
- 2 Escolher a próxima tarefa a receber a CPU
- 3 Restaurar o contexto da outra tarefa, para retomá-la

Entidades envolvidas:

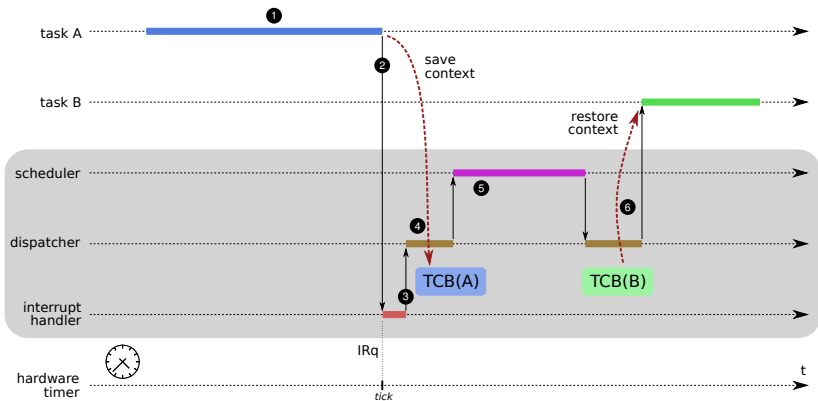
## Escalonador

Avalia as tarefas prontas e define a sua ordem de execução

## Despachante

Implementa a troca de contextos em baixo nível

# Trocas de contexto





# Processos

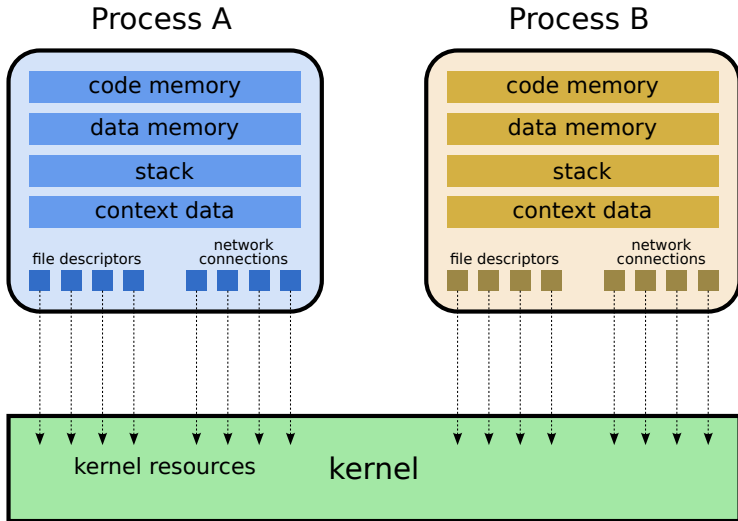
## Processo

Contêiner de recursos utilizados para executar tarefas

Os processos são isolados entre si pelo hardware:

- Isolamento de áreas de memória
- Níveis de operação da CPU
- Chamadas de sistema

# Processos



# Criação de processos

Processos são criados através de chamadas de sistema.

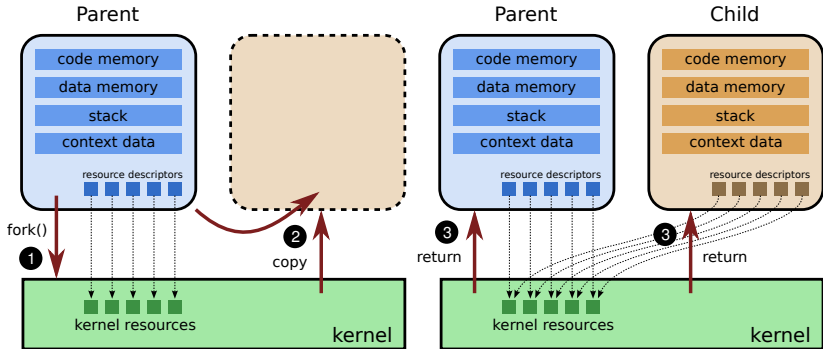
Em Windows:

- `CreateProcess`: cria um novo processo, informando o nome do arquivo contendo o código a executar

Em UNIX:

- `fork()`: duplica o processo atual
- `execve(file)`: substitui o código executável do processo por outro código.

# Criação de processos em UNIX



# Criação de processos em UNIX

```

1  int main (int argc, char *argv[], char *envp[])
2  {
3      int ret ;
4
5      ret = fork () ;                // replicação do processo
6      if ( ret < 0 ) {              // fork não funcionou
7          perror ("Erro: ") ;
8          exit (-1) ;
9      }
10     else
11         if ( ret > 0 )             // sou o processo pai
12             wait (0) ;            // esperar filho concluir
13         else {                    // sou o processo filho
14             execve ("/bin/date", argv, envp) ; // carrega binário
15             perror ("Erro: ") ;    // execve não funcionou
16         }
17     printf ("Tchau !\n") ;
18     exit(0) ;
19 }
  
```

# Hierarquia de processos

```

1  init--cron
2      |-dhcpcd
3      |-getty
4      |-getty
5      |-ifplugd
6      |-ifplugd
7      |-lighttpd---php-cgi--php-cgi
8      |           '-php-cgi
9      |-logsave
10     |-logsave
11     |-ntpd
12     |-openvpn
13     |-p910nd
14     |-rsyslogd--{rsyslogd}
15     |           '-{rsyslogd}
16     |-sshd---sshd---sshd---pstree
17     |-thd
18     '-udevd--udevd
19         '-udevd
  
```

# Threads

## Thread

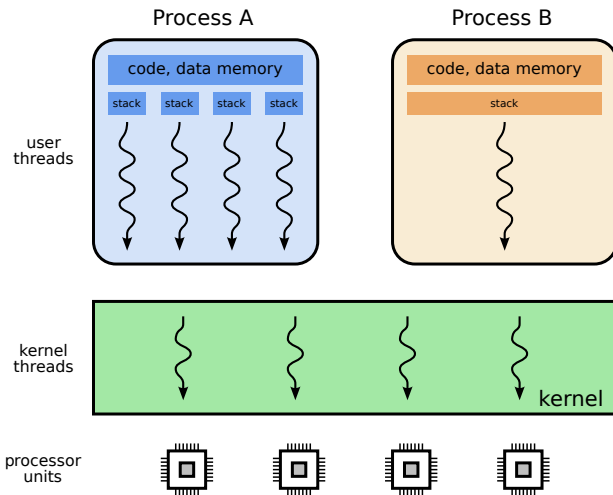
Cada um dos fluxos de execução operando dentro de um processo ou dentro do núcleo do sistema operacional.

Tipos de *threads*:

- **Threads de usuário:** fluxos de execução dentro de um processo, associados à execução da aplicação.
- **Threads de núcleo:** fluxos de execução dentro do núcleo; representam threads de usuário ou atividades do núcleo.

Vários modelos de implementação: N:1, 1:1 ou N:M

# Threads





# Modelo de threads N:1

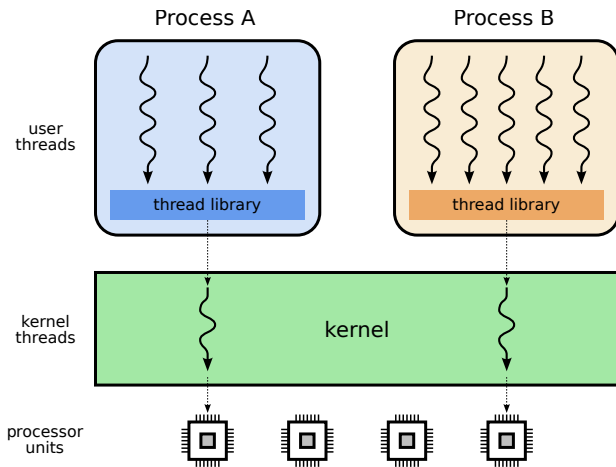
## Características:

- Sistemas antigos não suportavam *threads* de usuário.
- Bibliotecas gerenciam *threads* dentro de cada processo, sem ajuda do núcleo.
- Modelo utilizado por ser leve e de fácil implementação.
- Bom para aplicações com milhares/milhões de *threads*.

## Problemas:

- O núcleo não “vê” as *threads* de usuário, apenas um fluxo de execução para cada processo.
- Se uma *thread* do processo bloquear (E/S), todas param.

# Modelo de threads N:1



# Modelo de threads 1:1

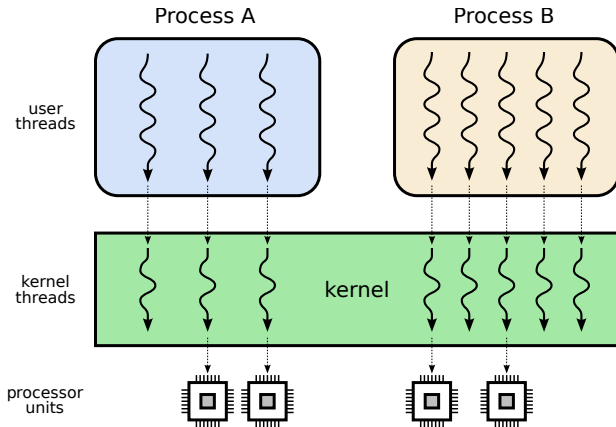
## Características:

- Gerência de *threads* incorporada ao núcleo.
- Cada *thread* de usuário é mapeada em uma *thread* de núcleo.
- As *threads* são escalonadas de forma independente, podendo usar vários processadores.
- É a implementação mais frequente nos sistemas atuais.

## Problemas:

- Pouco escalável (alguns milhares de threads)

# Modelo de threads 1:1



# Modelo de threads N:M

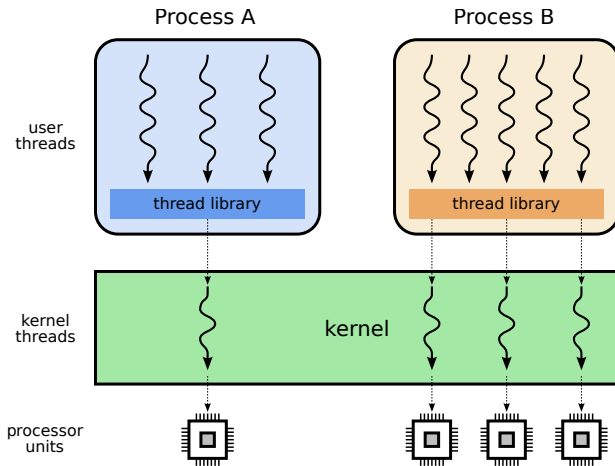
## Características:

- modelo híbrido entre N:1 e 1:1
- Uma biblioteca gerencia um conjunto de *threads* de usuário (dentro do processo)
- *Threads* de usuário mapeadas *threads* do núcleo
- O número de *threads* de núcleo é ajustado dinamicamente conforme a demanda da aplicação
- Modelo implementado no Solaris e FreeBSD

## Problemas:

- complexidade de implementação
- maior custo de gerência dos *threads* de núcleo

# Modelo de threads N:M



# Comparação entre modelos de threads

<b>Modelo</b>	<b>N:1</b>	<b>1:1</b>	<b>N:M</b>
Implementação	biblioteca	núcleo	ambos
Complexidade	baixa	média	alta
Custo de gerência	nulo	médio	alto
Escalabilidade	alta	baixa	alta
Vários processadores	não	sim	sim
Trocas de contexto	rápida	lenta	ambos
Divisão de recursos	injusta	justa	variável
Exemplos	GNU Threads	Windows Linux	Solaris FreeBSD

# Programando com *threads* em C

```

1  #include <pthread.h>
2  ...
3  #define NUM_THREADS 5
4
5  void *print_hello (void *threadid)      // cada thread executa esta função
6  {
7      printf ("%ld: Hello World!\n", (long) threadid);
8      sleep (5);
9      printf ("%ld: Bye bye World!\n", (long) threadid);
10     pthread_exit (NULL);                // encerra esta thread
11 }
12
13 int main (int argc, char *argv[])      // thread "main"
14 {
15     pthread_t thread[NUM_THREADS];
16     long status, i;
17
18     for(i = 0; i < NUM_THREADS; i++)    // cria as demais threads
19     {
20         printf ("Creating thread %ld\n", i);
21         status = pthread_create (&thread[i], NULL, print_hello, (void *) i);
22
23         if (status) { // ocorreu um erro
24             perror ("pthread_create");
25             exit (-1);
26         }
27     }
28
29     pthread_exit (NULL);                // encerra a thread "main"
30 }
  
```



# Programando com *threads* em Java

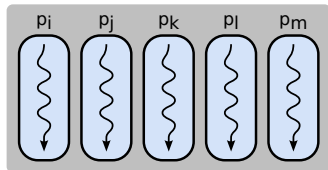
```

1  public class MyThread extends Thread {
2      int threadID;
3      private static final int NUM_THREADS = 5 ;
4
5      MyThread (int ID) {
6          threadID = ID;
7      }
8
9      public void run () {                               // corpo de cada thread
10         System.out.println (threadID + ": Hello World!") ;
11         try {
12             Thread.sleep(5000);
13         }
14         catch (InterruptedException e) {
15             e.printStackTrace();
16         }
17         System.out.println (threadID + ": Bye bye World!") ;
18     }
19
20     public static void main (String args[]) {
21         MyThread[] t = new MyThread[NUM_THREADS] ;
22
23         for (int i = 0; i < NUM_THREADS; i++) {         // cria as threads
24             t[i] = new MyThread (i);
25         }
26         for (int i = 0; i < NUM_THREADS; i++) {         // inicia a execução das threads
27             t[i].start () ;
28         }
29     }
30 }
  
```

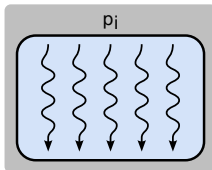
# Usar *threads* ou processos?

Opções:

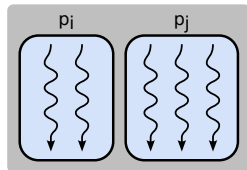
- Tarefas são processos distintos
- Tarefas são threads no mesmo processo
- Híbrido



com processos



com threads



híbrido

# Usar *threads* ou processos?

Característica	Com processos	Com <i>threads</i> (1:1)	Híbrido
Custo de criação de tarefas	alto	baixo	médio
Troca de contexto	lenta	rápida	variável
Uso de memória	alto	baixo	médio
Compartilhamento de dados entre tarefas	canais de comunicação e áreas de memória compartilhada.	variáveis globais e dinâmicas.	ambos.
Robustez	um erro fica contido no processo.	um erro pode afetar todas as <i>threads</i> .	um erro pode afetar as <i>threads</i> no mesmo processo.
Segurança	cada processo pode executar com usuários e permissões distintas.	todas as <i>threads</i> herdam as permissões do processo onde executam.	<i>threads</i> com as mesmas permissões podem ser agrupadas em um mesmo processo.
Exemplos	Apache 1.*, PostGRES	Apache 2.*, MySQL	Chrome, Firefox, Oracle