Programaç o Paralela e Distribuída

Multiprogramaç o Leve

Gerson Geraldo H Cavalheiro

Bacharelado em Ci ncia da Computaç o
Universidade do Vale do Rio dos Sinos
CC / UNISINOS

Sumário

Multiprogramaç o Leve

Escopo do tópico

Descrever a concorr ncia de uma aplicaç o

Mecanismos para obtenç o de uma execuç o eficiente dependem unicamente de como o programador faz uso dos recursos disponíveis

P e x:

- **Seleç o da arquitetura e da(s) ferramenta(s)**
- **Distribuiç o das tarefas entre os processadores**
- **Critérios na definiç o das sincronizaç es entre tarefas**

Multiprogramaç o Leve

Motivaç o ao uso de *threads*

S o mais leves que os processos;

Permitem descrever a concorr ncia da aplicaç o;

Aumenta o uso do processador por um processo;

Reflete as arquiteturas multiprocessadas (os SMPs);

Possui uma estrutura que permite adaptaç o aos recursos de processamento disponíveis;

E, devido permitir o compartilhamento de memória, é considerada simples em relaç o a outras ferramentas

Sumário

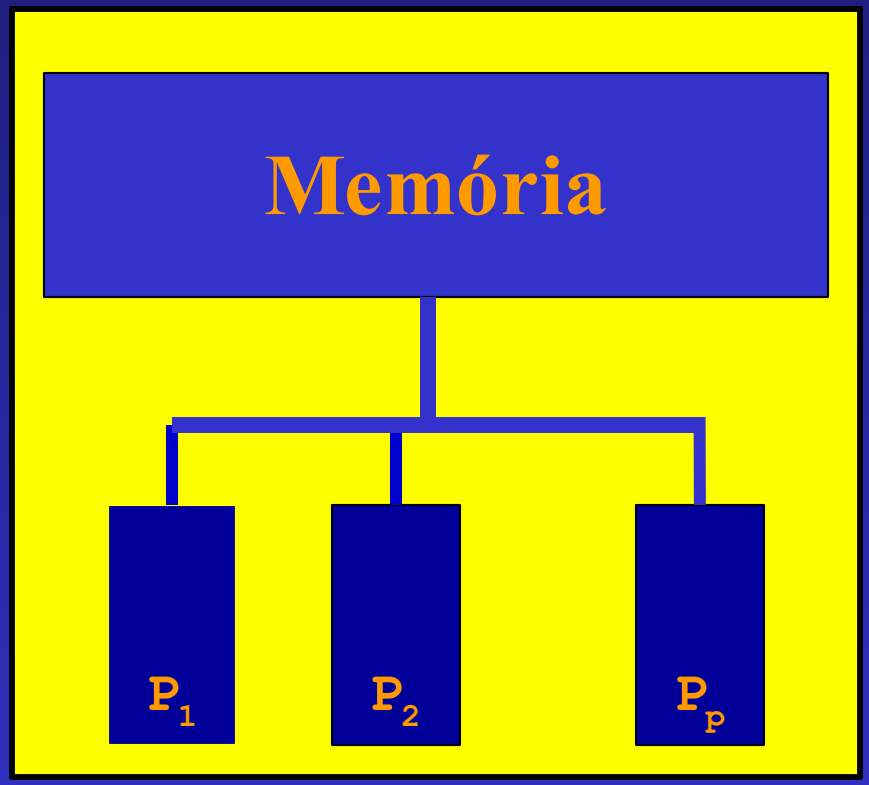
Introduç o

Arquitetura SMP: *Symmetric Multi Processor*

Uma arquitetura SMP dotada de P processadores id nticos;

Uma área de memória compartilhada entre os processadores;

Normalmente a natureza paralela da arquitetura n o é vista pelo usuário;



Modelo de arquitetura para a multiprogramaç o leve

Introduç o

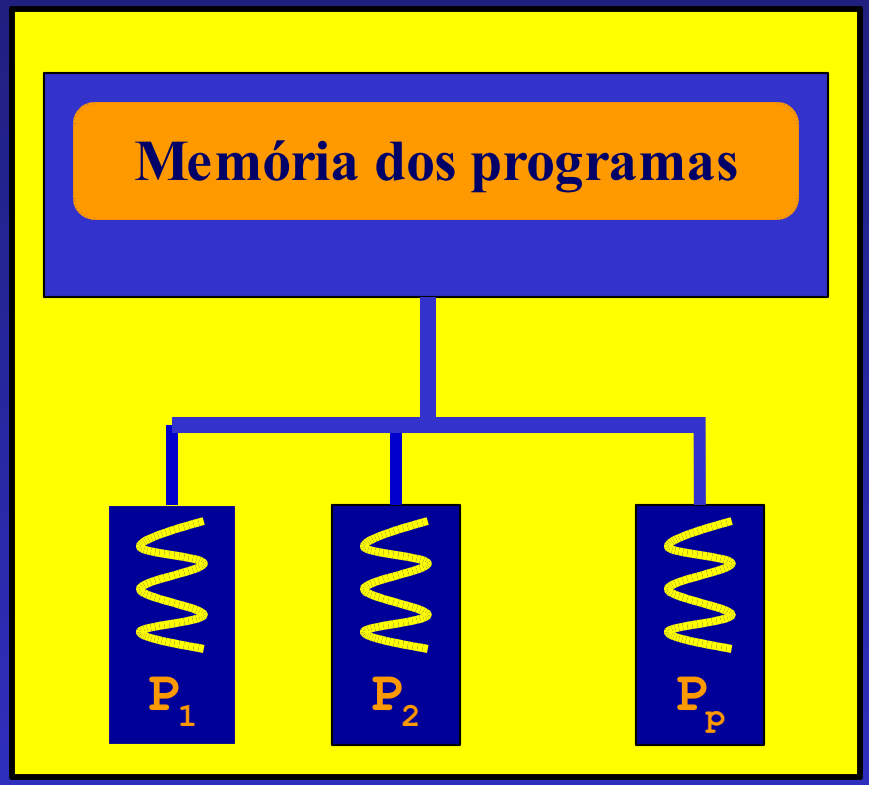
Arquitetura SMP: *Symmetric Multi Processor*

Uma arquitetura SMP dotada de P processadores

Uma área de memória compartilhada entre os processadores

Cada processador é capaz de executar um fluxo de execuç o independente

A memória armazena os dados dos programas



Fluxo de execuç o: representa a atividade do processo

Introduç o

Arquitetura SMP: *Symmetric Multi Processor*

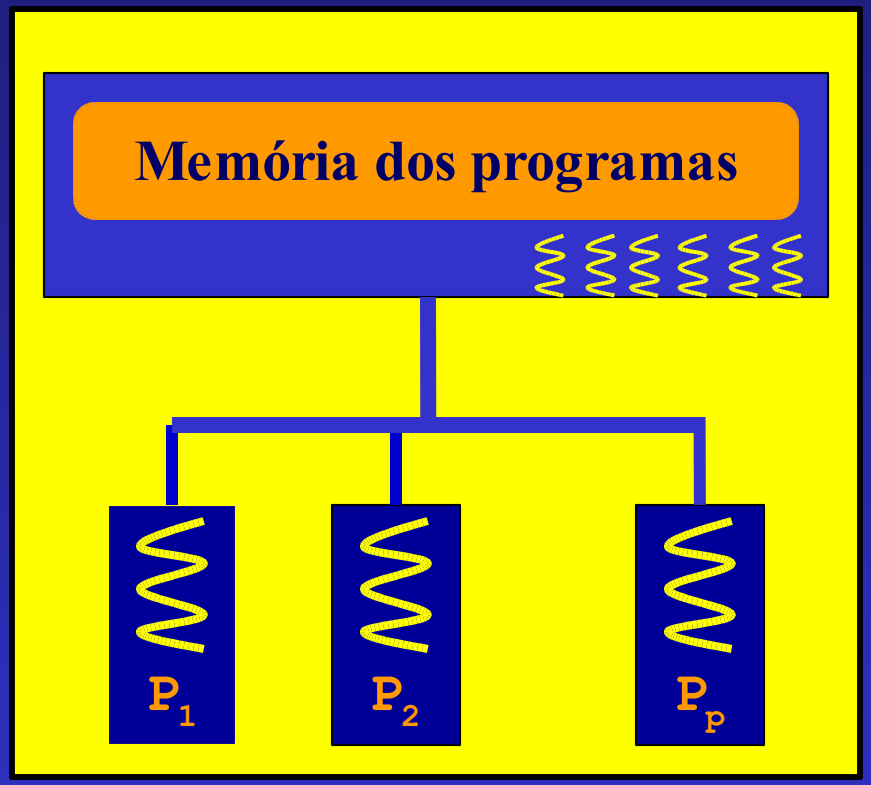
Uma arquitetura SMP dotada de `P` processadores

Uma área de memória compartilhada entre os processadores

Cada processador é capaz de executar um fluxo de execuç o independente

A memória armazena os dados dos programas

Número de fluxos > número de processadores



Introduç o

Processos *vs* Threads

Noç o de processo

Um programa define um conjunto de instruç s

Programa

Multiprogramaç o Leve: Threads POSIX

Processos *vs* Threads

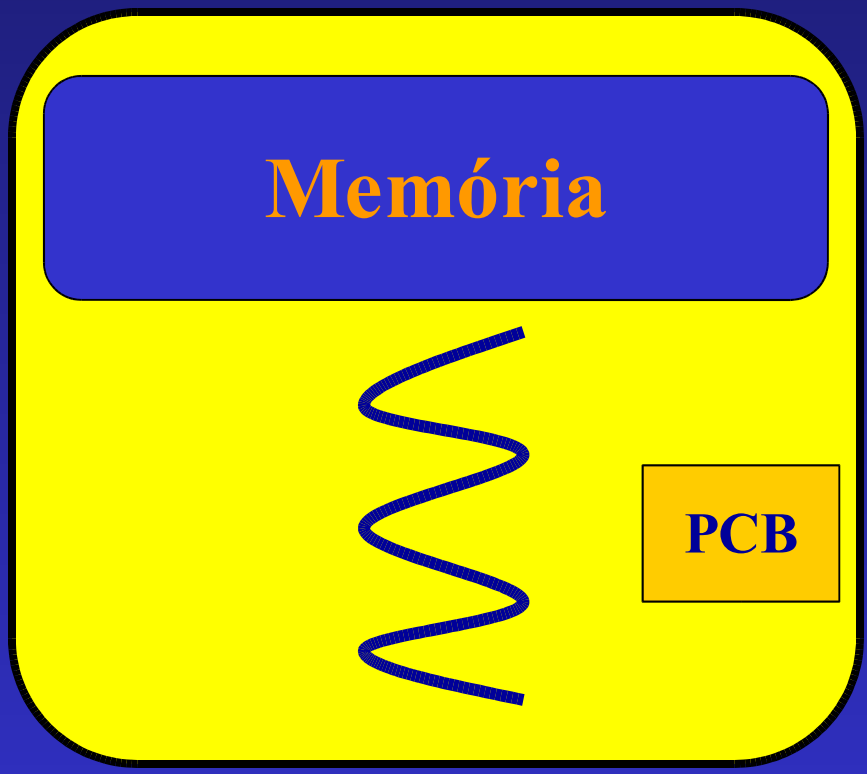
Noç o de processo

Um programa define um conjunto de instruç s

Programa

Quanto submetido execuç o, com um conjunto de dados de entrada, é instanciado sob a forma de um processo

O processo possui um fluxo de



Um processo é uma instância de um programa

Introduç o

Processos *vs* Threads

Em SOs multitarefa (como Unix), cada tarefa (conceito do SO) executa como um processo independente, com com uma área de endereçamento privada;

Em uma arquitetura SMP o SO pode explorar os múltiplos processadores para realizar balanceamento de carga

Introduç o

Processos *vs* Threads

Semelhanças

- o Assim como ocorre com processos, duas ou mais threads n o podem estar ativas em uma mesma CPU em um determinado instante de tempo;
- o Tanto threads como processos executam seq encialmente um conjunto de instruç es e acessam uma área de memória privada;
- o Threads como os processos podem criar filhos

Diferenças

- o Threads n o s o independentes entre si;
- o Threads compartilham um espaço de endereçamento comum;
- o Threads s o projetadas para auxiliar outras threads na realizaç o da computaç o de um programa processos normalmente s o concebidos por usuários distintos;
- o Uma thread filha n o tem acesso ao id da thread m e, ao contrário de processos através de *getppid*

Processos *vs* Threads

Processo Multithread

Execuç o simultânea de diferentes seq ncias de instruç es;

Os recursos do processo s o compartilhados;

Enquanto executam, os fluxos s o independentes entre si;

Trocas de dados (comunicaç es) entre estes fluxos se d o através de leituras e escritas em uma memória compartilhada;

Os fluxos de execuç o concorrem pelos recursos da arquitetura;

Os fluxos de execuç o concorrem pelo acesso aos dados na memória compartilhada (aos dados)

Introduç o

Processos *vs* Threads

Thread: Processo Leve

S o mais leves para criaç o pois necessitam apenas uma pilha (para dados locais) e imagens dos registradores;

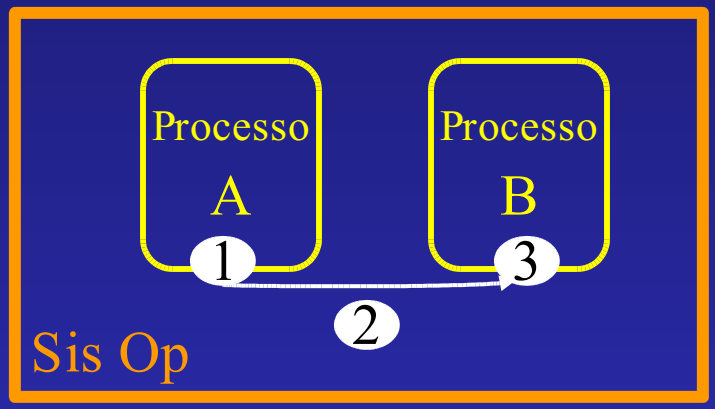
S o mais leves porque consomem menos recursos associados ao suporte execuç o, tais como espaço de endereçamento, dados globais, código do programa;

S o mais leves porque permitem uma troca de contexto mais eficiente entre threads: é necessário apenas salvar/restaurar o program counter, o stack pointer e os registradores

O barato pode sair caro: n o há um encapsulamento entre threads

Processos *vs* Threads

Troca 1:
Espaço usuário
para kernel



Troca 3:
Espaço kernel
para usuário

Troca 2:
Processo A para Processo B

Troca de Contexto de Processo

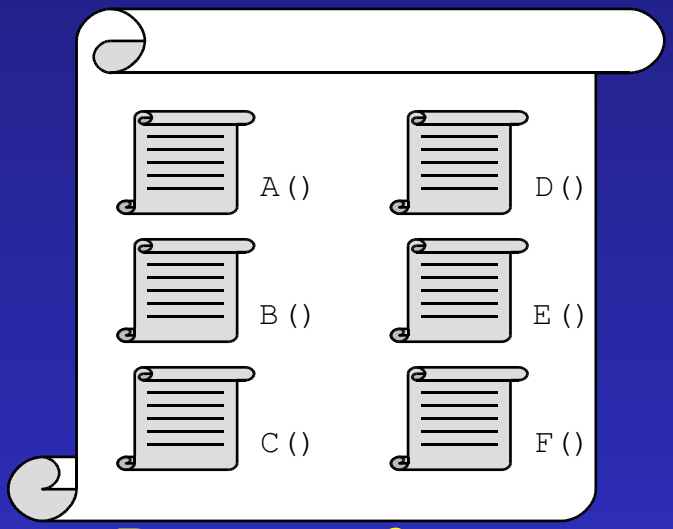
2	Salva contexto corrente da CPU
3	memória
4	Carrega próximo contexto da CPU
5	memória
6	Carrega páginas do swap (se for ocaso)

Troca de Contexto de Therads

2	Salva contexto corrente da CPU
3	Carrega próximo contexto da CPU

Introduç o

Processos *vs* Threads

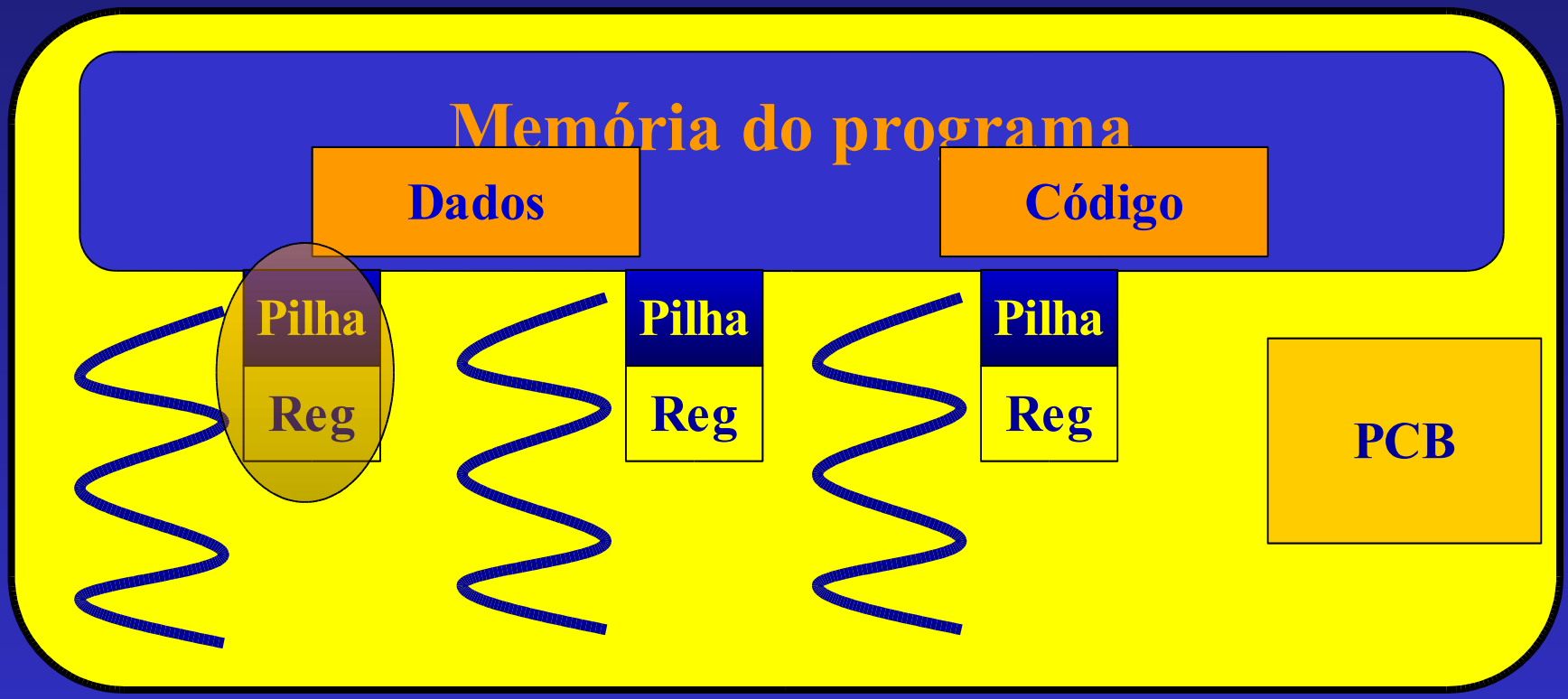


Programa fonte

Processo com múltiplas *threads*

Introduç o

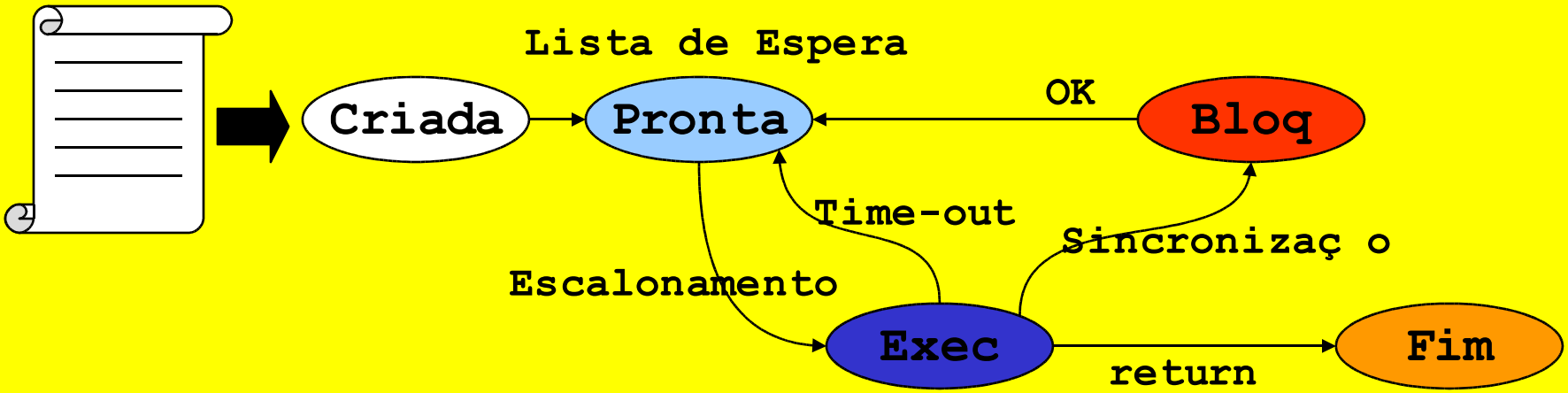
Processos *vs* Threads



Processo Pesado *vs* Processo Leve

Processos *vs* Threads

Ciclo de vida de uma *thread*



Introduç o

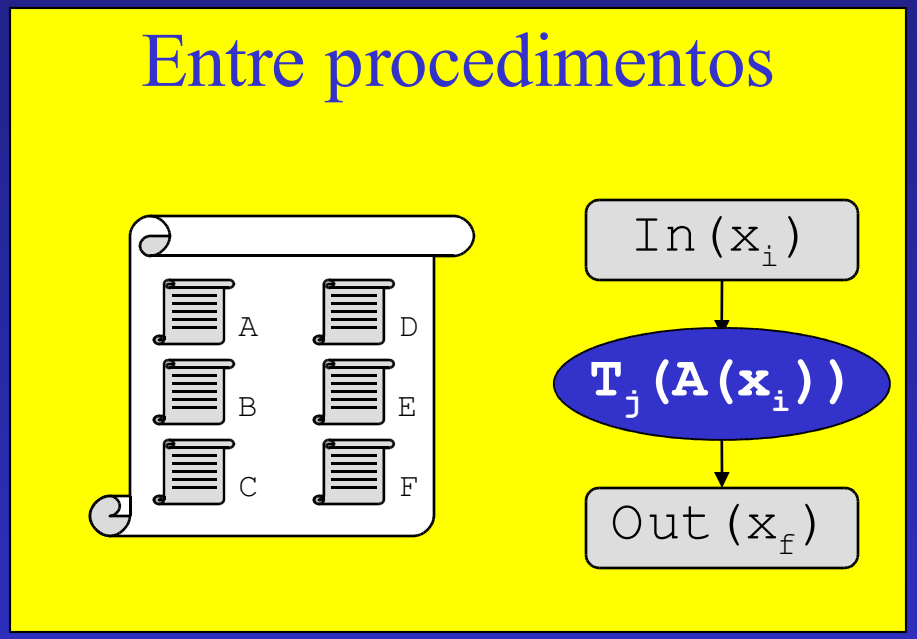
Programaç o Concorrente

Intra-instruç es
- CISC

Entre-instruç es
- Super-Escalar
- Vetorial

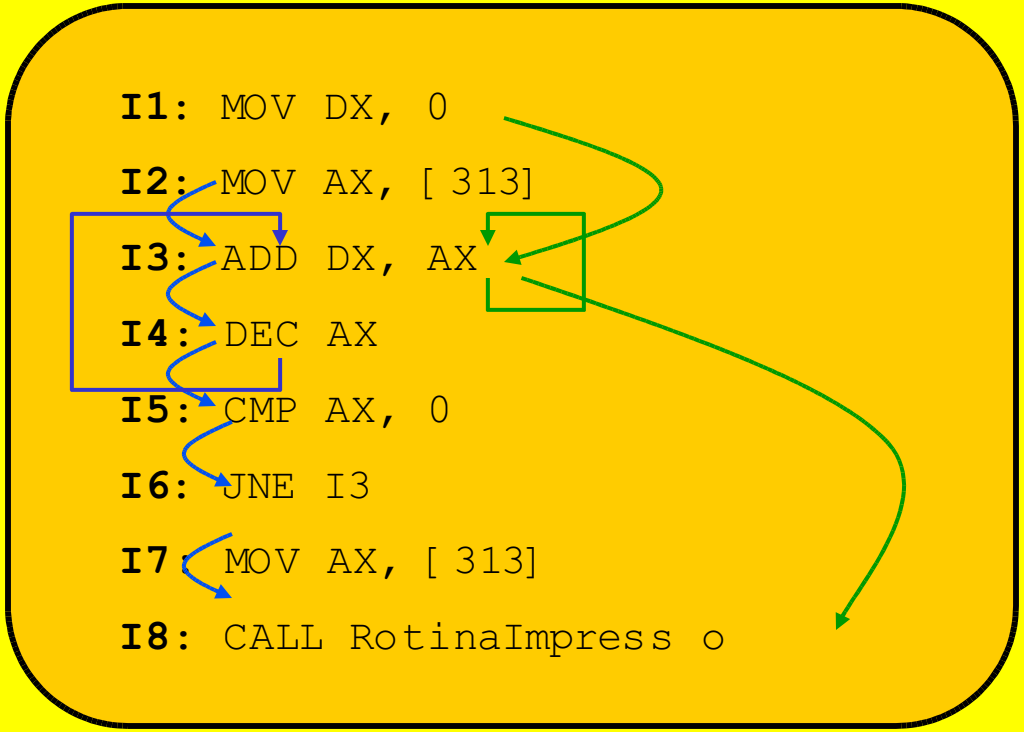
Entre procedimentos
- Tarefas

Entre processos
- Programas



Programaç o Concorrente

Execuç o Seq encial



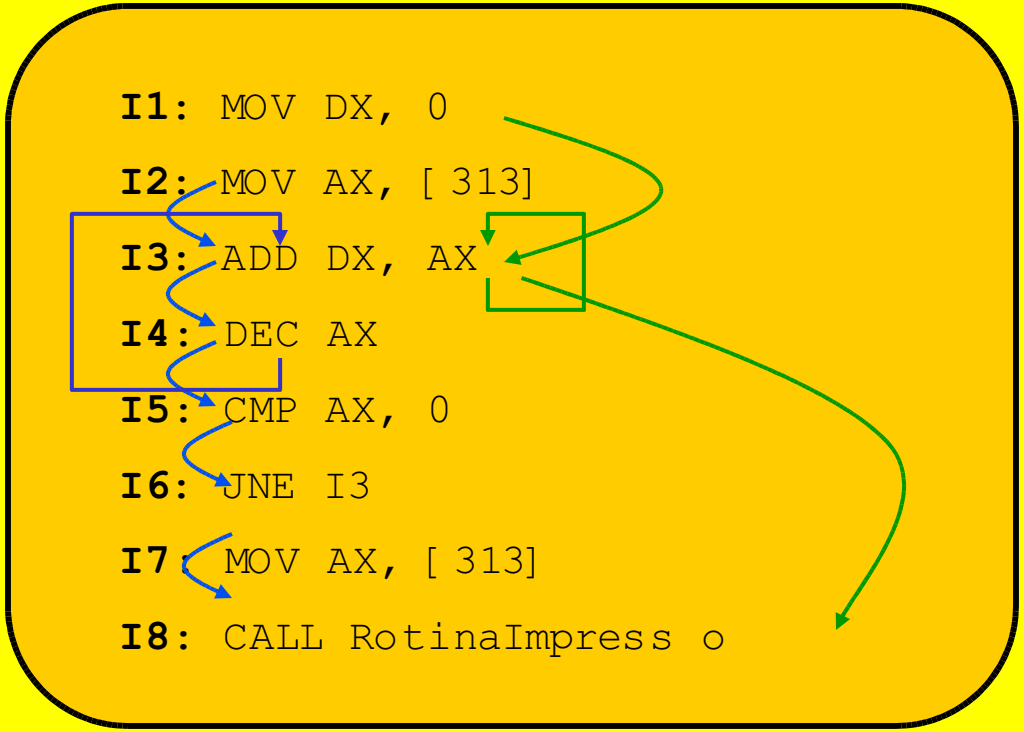
Efeito da execuç o de uma instruç o:

Escrita em memória

Comunicaç o

Programaç o Concorrente

Execuç o Seq encial



Premissa para executar uma instruç o:

Dados de entrada em memória

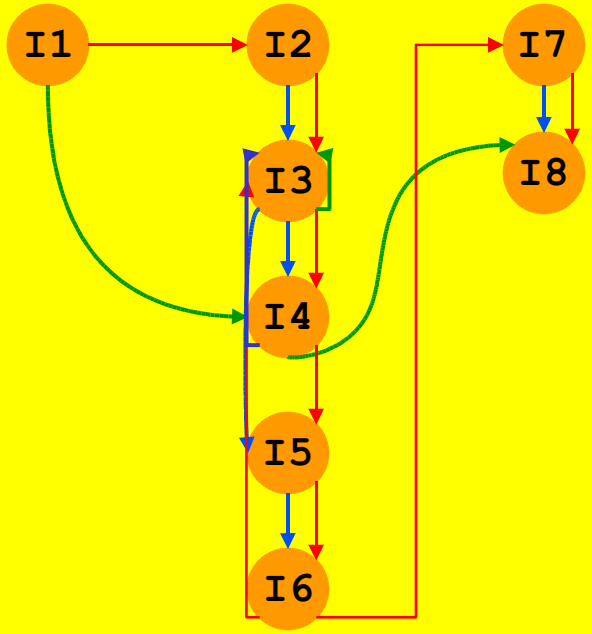
Sincronizaç o

Introduç o

Programaç o Concorrente

Execuç o Seq encial

```
I1: MOV DX, 0
I2: MOV AX, [313]
I3: ADD DX, AX
I4: DEC AX
I5: CMP AX, 0
I6: JNE I3
I7: MOV AX, [313]
I8: CALL RotinaImpress o
```



Introduç o

Programaç o Concorrente

Execuç o Seq encial

Dados em memória;

Unidade de execuç o: instruç es;

Parâmetros e retorno (comunicaç o): implícitos;

Controle da sincronizaç o: implícito

Efeito Colateral

Introduç o

Programaç o Concorrente

Execuç o Seq encial

THREADS

Dados em memória;

Unidade de execuç o: insruç es; FUNÇ ES

Parâmetros e retorno (comunicaç o): implícitos; EXPLÍCITO

Controle da sincronizaç o: implícito EXPLÍCITO

Efeito Colateral

controlado pelo programador

Introduç o

Programaç o Concorrente

Técnica de programaç o que explora a **independ ncia temporal** de *atividades* definidas por uma aplicaç o

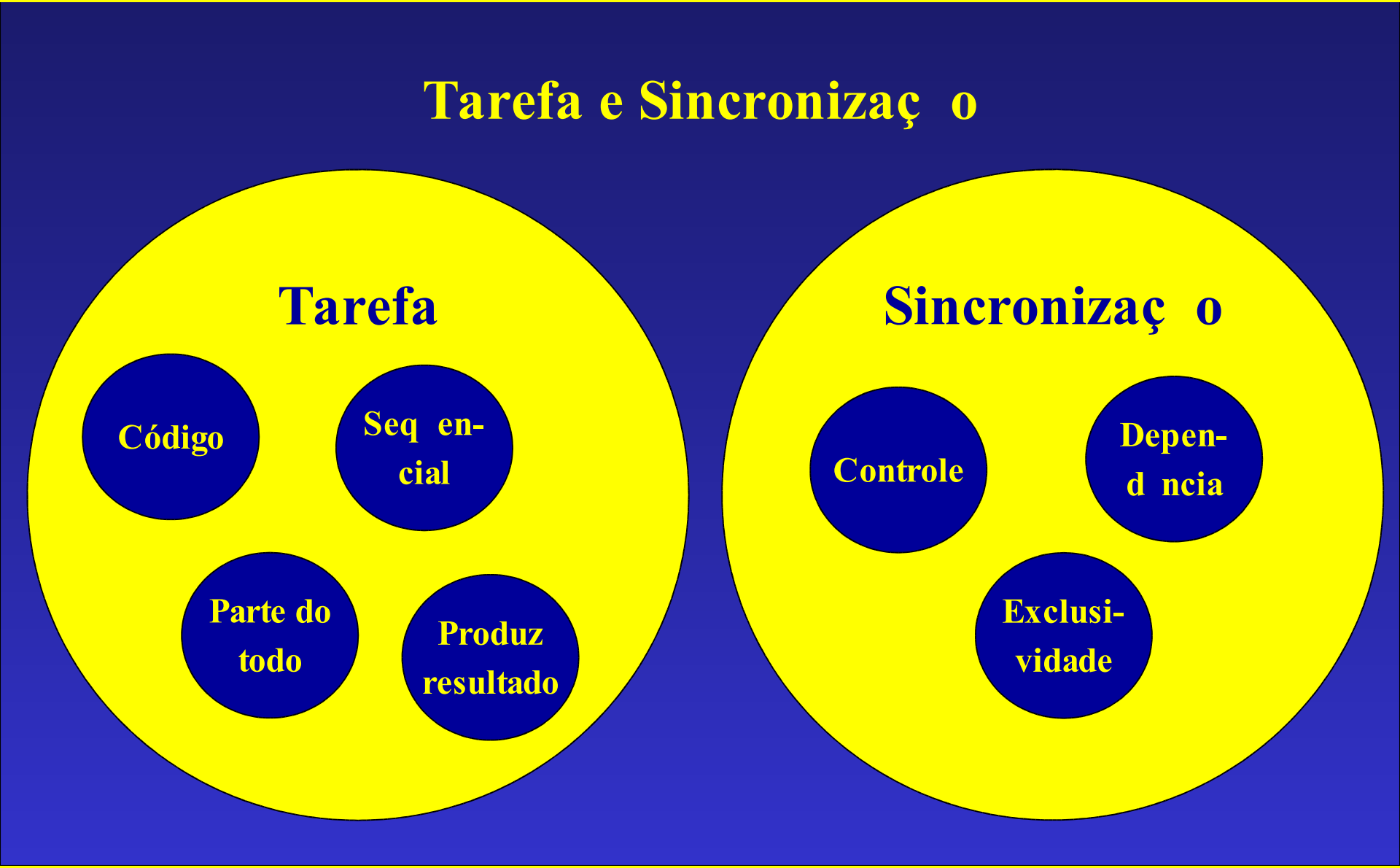
Compartilhamento
de dados

Leitura/escrita
em memória

Cooperaç o
para evoluç o da execuç o

Sincronizaç o
entre atividades

Introduç o


Tarefa e Sincronizaç o
Tarefa
Código
Seq en-
cial
Parte do
todo
Produz
resultado
Sincronizaç o
Controle
Depen-
d ncia
Exclusi-
vidade

Introduç o

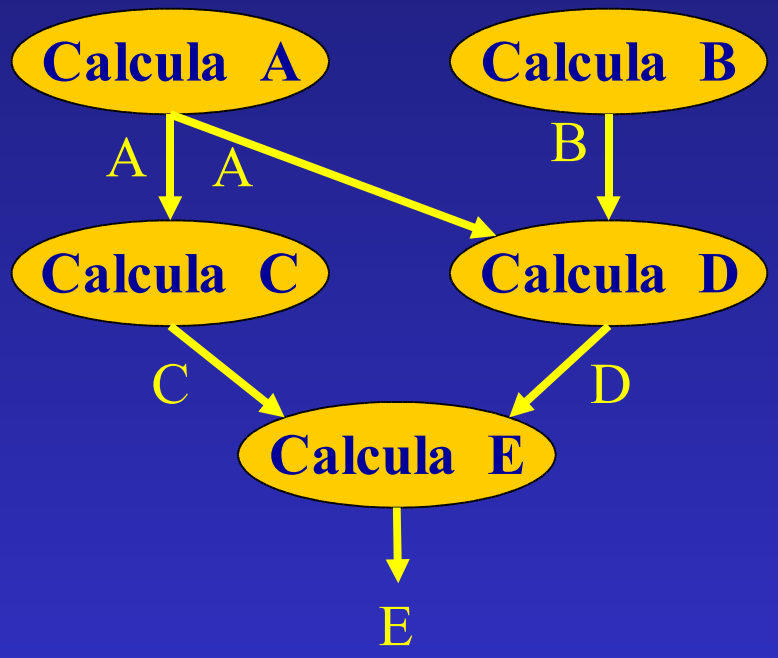

Tarefa e Sincronizaç o
Tarefa
Sincronizaç o
Códig
n-
a
Comunicaç o
Troca de dados entre tarefas
seç es críticas e depend ncias entre tarefas

Tarefa e Sincronizaç o

Exemplo

Dados: A, B, C, D, E

A = Calcula A (void);
B = Calcula B (void);
C = Calcula C (A);
D = Calcula D (A, B);
E = Calcula E (C, D);



Sumário

Threads POSIX Pthreads

GPL

```
/* Linuxthreads - a simple clone()-based implementation of Posix      */
/* threads for Linux                                                  */
/* Copyright (C) 1996 Xavier Leroy (Xavier Leroy@inria fr)            */
/*                                                                    */
/* This program is free software; you can redistribute it and/or      */
/* modify it under the terms of the GNU Library General Public License*/
/* as published by the Free Software Foundation; either version 2     */
/* of the License, or (at your option) any later version              */
/*                                                                    */
 /* This program is distributed in the hope that it will be useful,   */
/* but WITHOUT ANY WARRANTY; without even the implied warranty of     */
/* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE   See the      */
/* GNU Library General Public License for more details                */
```

Concorr ncia de Execuç o

Corpo de uma *Threads*

É uma funç o C/C++ convencional;

Recebe e retorna endereços de memória;

Variáveis locais uma *thread* s o visíveis apenas no escopo

```
void *foo(void *args)
{

 // Código C/C++

}
```

Duas *threads* podem ser criadas a partir da mesma funç o, no entanto, s o instâncias diferentes !!!

Concorr ncia de Execuç o

Manipulaç o de *Threads*

Criaç o:

```
int pthread create(    , void *(*foo)(void *), void *arg );
```

Término:

```
void pthread exit(void *retval );
return (void *)retval;
```

Sincronizaç o:

```
void pthread join(pthread thid, void **ret);
```

Identificaç o:

```
void pthread self(void *retval );
```

Threads POSIX Pthreads

Concorr ncia de Execuç o

Manipulaç o de *Threads*

Criaç o:

```
pthreadc reate( pthread t *tid,
                pthread attrt  *atrib,
                void *(*func)(void *),
                void *args );
```

Cria um novo fluxo de execuç o (uma nova *thread*) O novo fluxo executa de forma concorrente com a *thread* criadora

Threads POSIX Pthreads

Concorr ncia de Execuç o

Manipulaç o de *Threads*

Criaç o:

```
pthreadc reate( pthread t *tid,
                pthread attrt  *atrib,
                void *(*func)(void *),
                void *args );
```

`func`: nome da funç o que contém o código a ser executado pela *thread*

Threads POSIX Pthreads

Concorrência de Execução

Manipulação de *Threads*

Criação:

```
pthreadcreate( pthread_t *tid,
               pthread_attrt  *atrib,
               void *(*func)(void *),
               void *args );
```

`args`: ponteiro para uma região de memória com dados de entrada para a função

Concorr ncia de Execuç o

Manipulaç o de *Threads*

Criaç o:

```
pthreadc reate( pthread t *tid,
                pthread attrt  *atrib,
                void *(*func)(void *),
                void *args );
```

`tid`: identificador (único) da nova *thread*

Concorr ncia de Execuç o

Manipulaç o de *Threads*

Criaç o:

```
pthreadc reate( pthread t *tid,
                pthread attrt  *atrib,
                void *(*func)(void *),
                void *args );
```

`atrib`: atributos para a nova *thread*

Threads POSIX Pthreads

Concorr ncia de Execuç o

Estrutura Opaca

Manipulada apenas através de funç es

```
pthreada ttr init(     );
pthreada ttr setscope(     );
pthreada ttr setdetachedstate(     );
```

`atrib`: atributos para a nova *thread*

Concorr ncia de Execuç o

Manipulaç o de *Threads*

Criaç o:

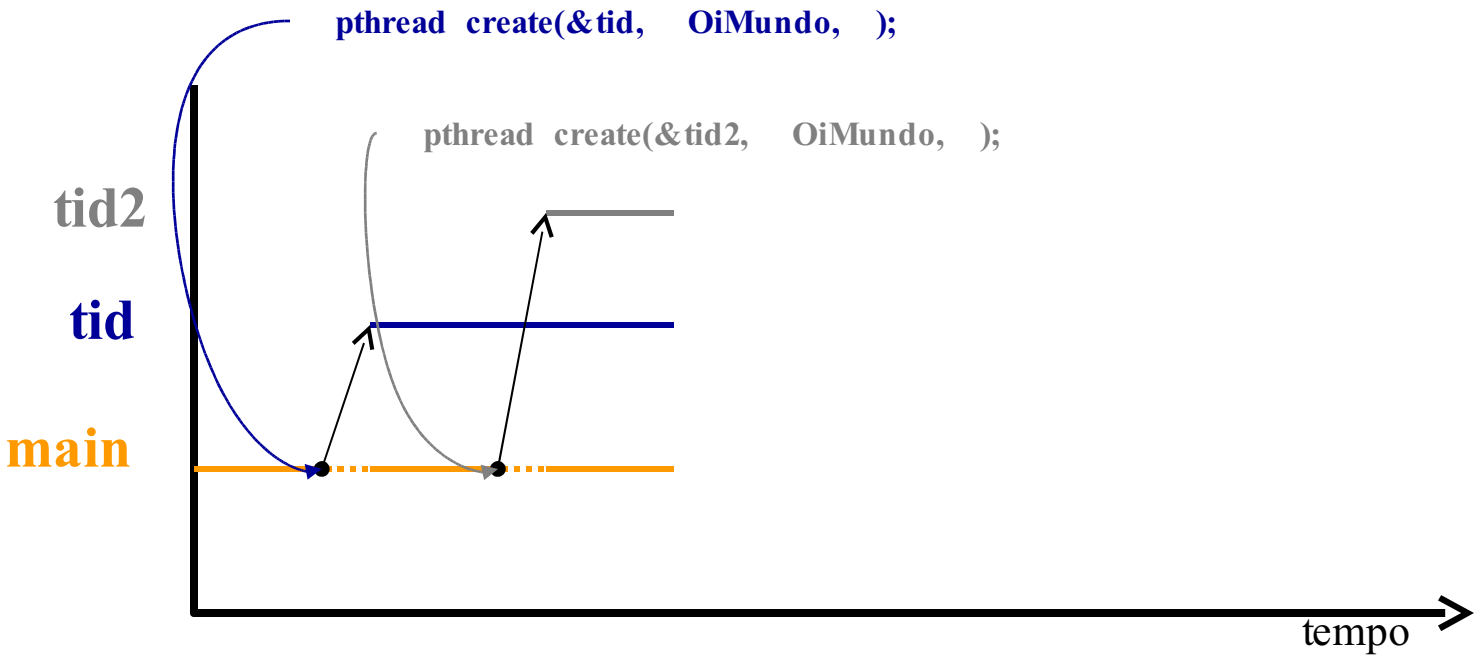
```
main() {
 pthread t tid;
 char *str =  Oi mundo ;
 pthread create( &tid, NULL,
                 OiMundo, str );

}
```

```
void *OiMundo(void *in){
 char *str = (char *)in;
 printf( %s\n , str);
}
```

Concorr ncia de Execuç o

Manipulaç o de *Threads*



Concorr ncia de Execuç o

Clone

Cria um processo filho que compartilha parte de seu contexto de execuç o com o processo original (espaço de memória, tabela de descritores de arquivos e tabela de handlers de sinais);

Difere do fork, uma vez que a chamada clone inicia a execuç o de um novo fluxo em uma funç o informada como parâmetro;

```
int clone(int (*fn)(void *), void *child stack, int flags, void *arg);
```

Threads POSIX Pthreads

Concorr ncia de Execuç o

Manipulaç o de *Threads*

Término:

```
void pthreade xit(void *retval );
```

ou

```
return (void *)retval;
```

Termina a execuç o da *thread* que executou a chamada

Joinable ou Detached

Concorr ncia de Execuç o

Manipulaç o de *Threads*

Término:

```
void pthreade xit(void *retval );
```

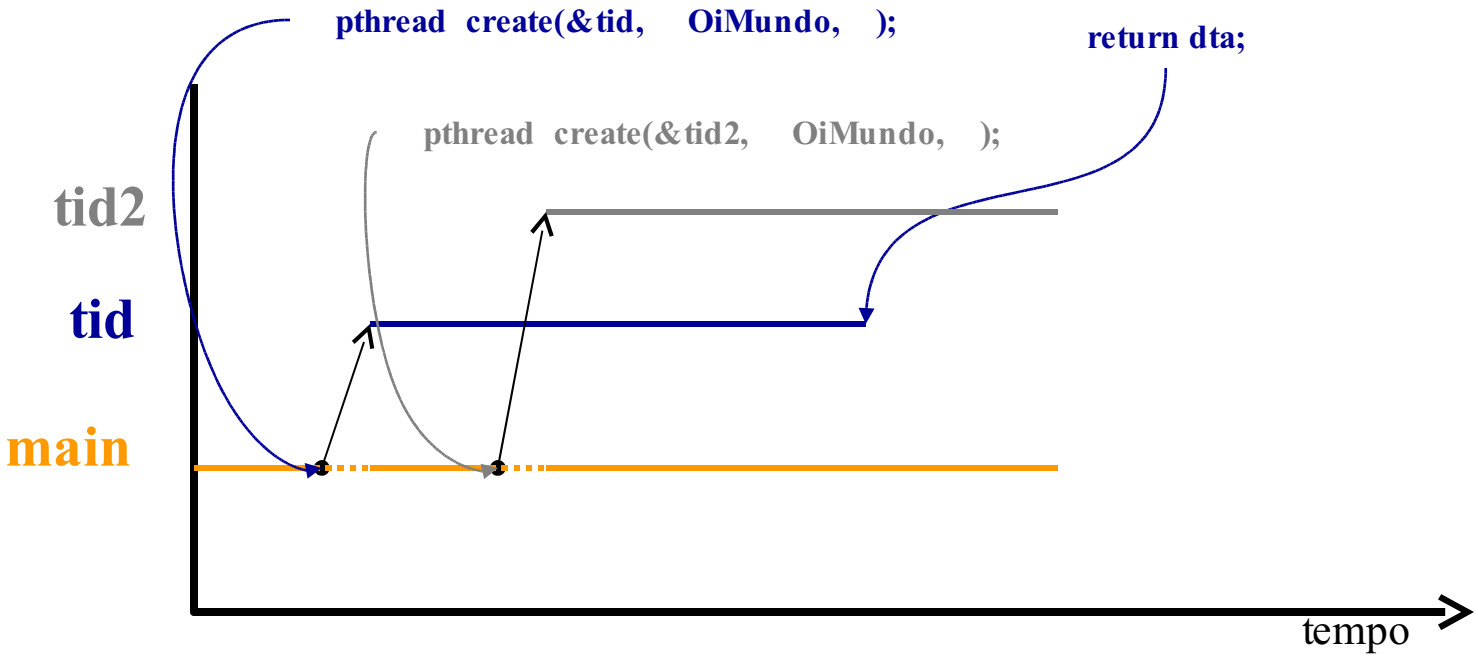
ou

```
return (void *)retval;
```

`retval`: endereço de uma posiç o de memória (dado retornado pela *thread*)

Concorr ncia de Execuç o

Manipulaç o de *Threads*



Threads POSIX Pthreads

Concorr ncia de Execuç o

Manipulaç o de *Threads*

Sincronizaç o:

```
void pthreadj oin(pthreadt  tid, void **ret);
```

Aguarda o término de uma *thread* (se ela n o terminou) e recupera o resultado produzido

Threads POSIX Pthreads

Concorr ncia de Execuç o

Manipulaç o de *Threads*

Sincronizaç o:

```
void pthreadj oin(pthreadt  tid, void **ret);
```

`tid`: identificador da *thread* a ter seu término sincronizado

Threads POSIX Pthreads

Concorr ncia de Execuç o

Manipulaç o de *Threads*

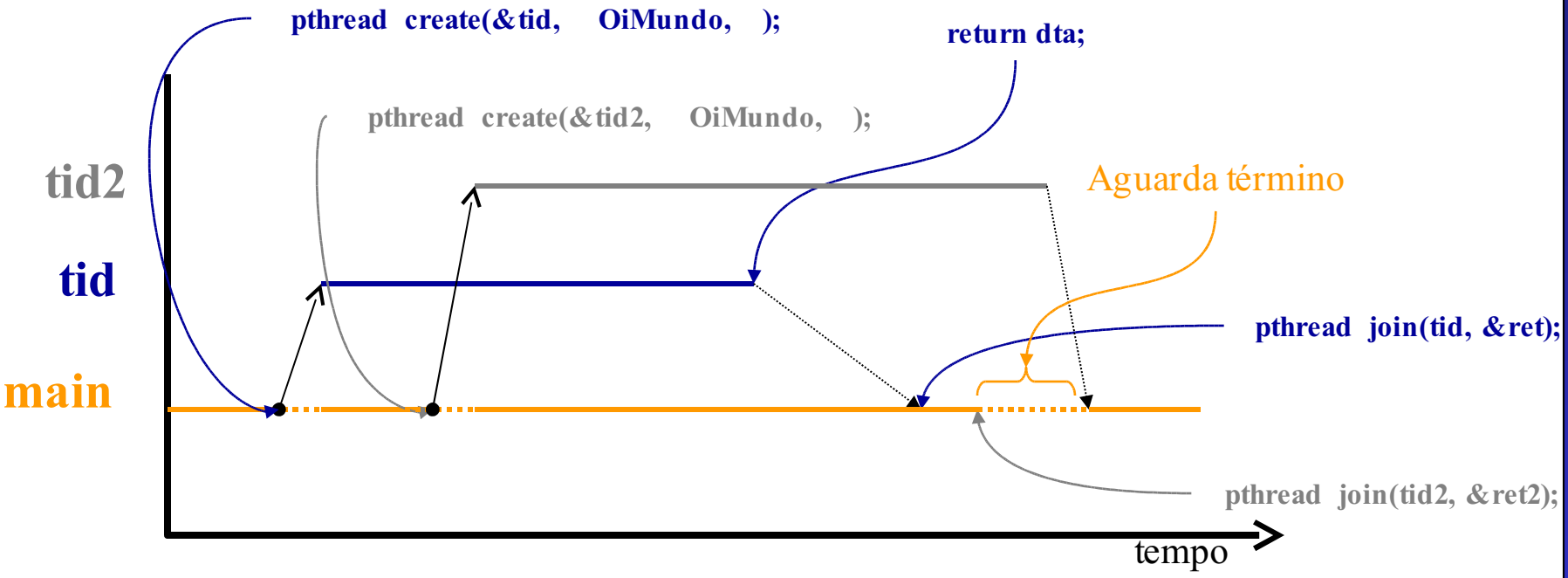
Sincronizaç o:

```
void pthreadj oin(pthreadt  thid, void **ret);
```

`ret`: endereço de um ponteiro que será atualizado com a posiç o de memória que contém os dados retornados pela *thread*

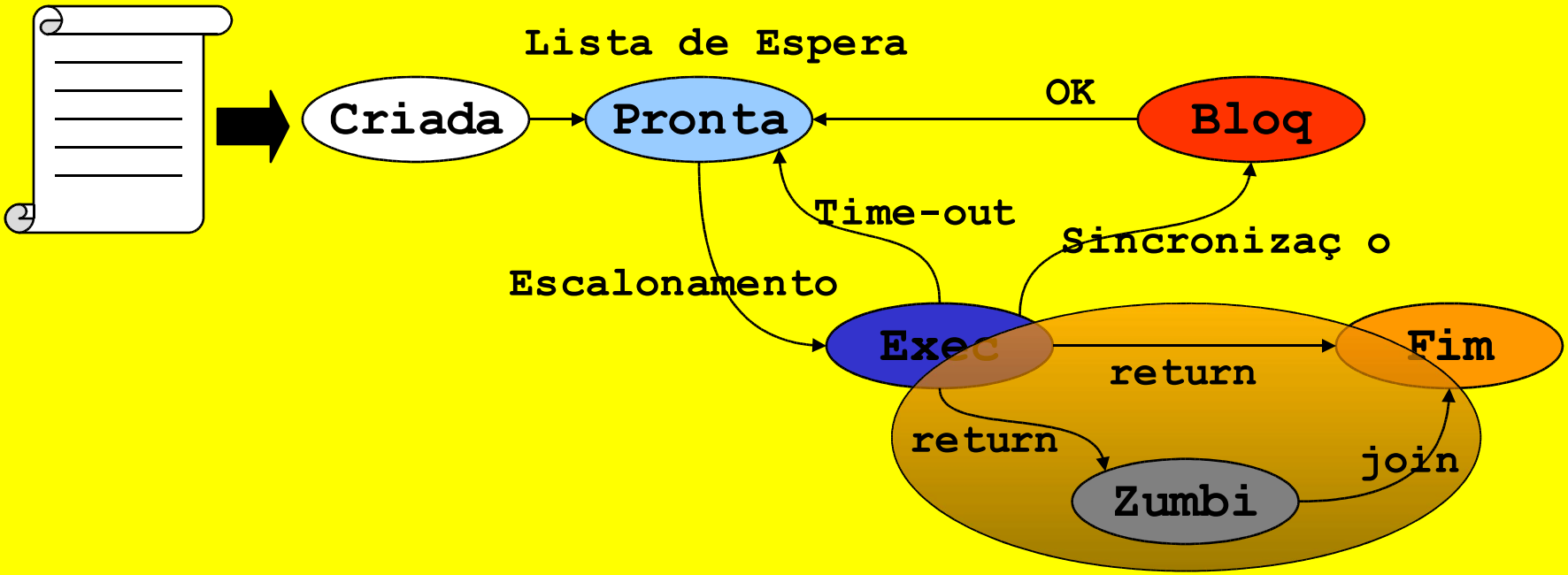
Concorr ncia de Execuç o

Manipulaç o de *Threads*



Concorr ncia de Execuç o

Ciclo de vida de uma *thread*: joinable ou detached



Threads POSIX Pthreads

Concorr ncia de Execuç o

Exemplo: Produtor / Consumidor

```
main(){
 pthread t p[5], c[5];
 for(i = 0 ; i < 5 ; i++ ) {
  pthread create(&(p[i]),   , prod, NULL );
  pthread create(&(c[i]),   , cons, &(p[i]));
 }
 for(i = 0 ; i < 5 ; i++ )
  pthread join(c[i], NULL);
}
```

```
void *prod(void *args){
 Buf *buf = malloc;
 produz item
 return buf;
}
```

```
void *cons(void *args){
 Buf *buf;
 pthread t p;
 p = (pthread t)*args;
 pthread join(p, &buf);
 consome item
 return NULL;
}
```

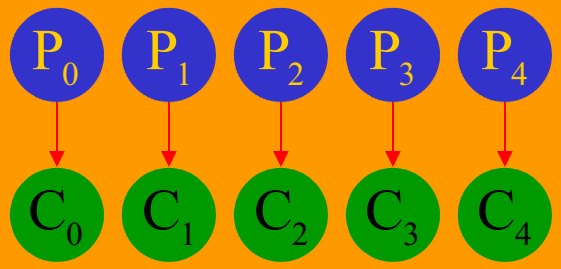
Concorr ncia de Execuç o

Exemplo: Produtor / Consumidor

```
main(){
  pthread join(c[i], NULL);
}
```

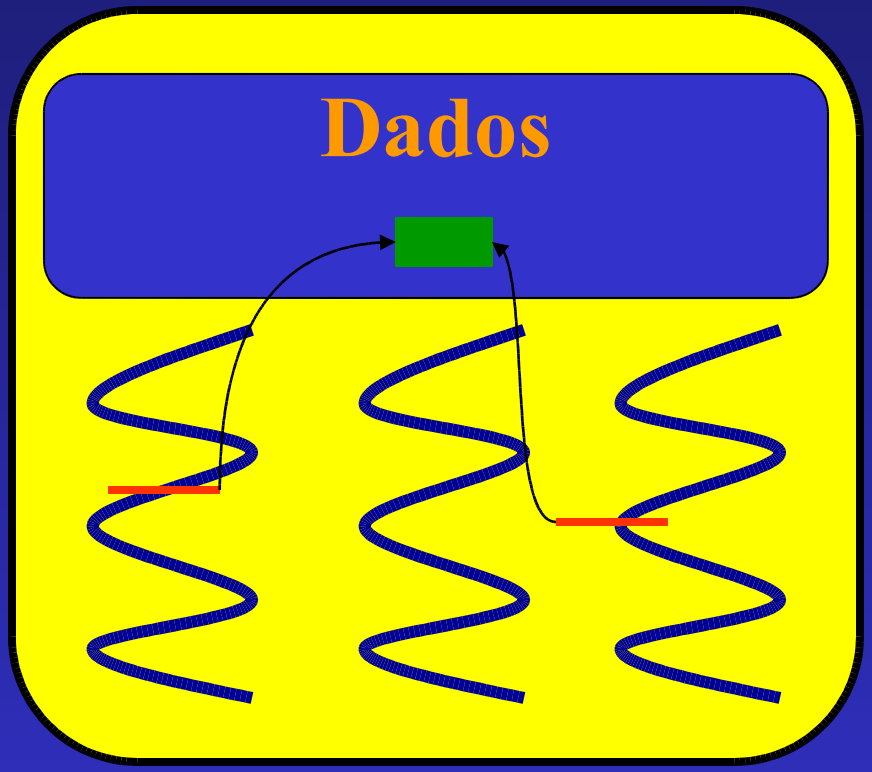
```
void *prod(void *args){
 Buf *buf = malloc;
 p = (pthread t)*args;
 pthread join(p, &buf);
 consome item
 return NULL;
}
```

A sincronizaç o por `join` garante a correta comunicaç o entre as tarefas executadas (controle da comunicaç o)



Threads POSIX Pthreads

Concorr ncia em Acessos Memória



O controle do acesso aos dados é de responsabilidade do programador

Concorr ncia em Acessos Memória

Acessos Memória Compartilhada

os acessos memória se d o através de operaç es de leitura e escrita convencionais, ou seja:

Escrita: DadoCompartilhado = valor;

Leitura: variável = DadoCompartilhado;

As instruç es que acessam os dados compartilhados s o considerados

Seç es Críticas

Concorr ncia em Acessos Memória

Acessos Memória Compartilhada

```
int x = 313;
```

// thread A	// thread B
a = x;	b = x;
a = a + 1;	b = b - 1;
x = a;	x = b;

A	a = x	313
A	a = a + 1	313
B	b = x	313
A	x = a	314
B	b = b - 1	314
B	x = b	312

Concorr ncia em Acessos Memória

Exclus o Mútua

Mutual exclusion
MUTEX

Garantia de que apenas uma *thread* terá acesso a um dado na memória compartilhada em um determinado instante de tempo;

Mecanismo oferecido por POSIX, responsabilidade do programador

Concorr ncia em Acessos Memória

Exclus o Mútua

Tipo de dado: `pthread mutex t`

Primitivas:

```
int pthread mutex lock(pthread mutex t *m);
int pthread mutex unlock(pthread mutex t *m);
int pthread mutex init(pthread mutex t *m,
                       pthread mutexattrib t *atrib);
```

Concorr ncia em Acessos Memória

Exclus o Mútua

Tipo de dado: `pthread mutex t`

Primitivas:

```
int pthread mutex lock(pthread mutex t *m);
int pthread mutex unlock(pthread mutex t *m);
int pthread mutex init(pthread mutex t *m,
                       pthread mutexattrib t *atrib);
```

Uso: `init`, para inicializar o mutex (`NULL` == aberto), `lock` para adquirir o passe e `unlock` para liberar

Concorr ncia em Acessos Memória

Exclus o Mútua

```
int x = 313;
pthread mutex t m;
pthread mutex init(&m,NULL); // executado no main
```

// thread A	// thread B
pthread mutex lock(&m);	pthread mutex lock(&m);
a = x;	b = x;
a = a + 1;	b = b - 1;
x = a;	x = b;
pthread mutex unlock(&m);	pthread mutex unlock(&m);

Concorr ncia em Acessos Memória

Variável de Condiç o

Permite o acesso a uma seç o crítica quando uma determinada condiç o for satisfeita

Leva em conta o estado da sincronizaç o

Uso típico no compartilhamento de um *buffer* por produtores e consumidores

Concorr ncia em Acessos Memória

Variável de Condiç o

Tipo de dado: `pthread cond t`

Primitivas:

```
int pthread cond wait(pthread cond t *c,
                      pthread mutex t *m);
int pthread cond signal(pthread cond t *c);
int pthread cond broadcast(pthread cond t *c);
int pthread cond init(pthread cond t *c,
                      pthread condattrib t *atrib);
```

Concorr ncia em Acessos Memória

Variável de Condiç o

Tipo de dado: `pthread cond t`

Primitivas:

```
int pthread cond wait(pthread cond t *c,
                      pthread mutex t *m);
int pthread cond signal(pthread cond t *c);
int pthread cond broadcast(pthread cond t *c);
```

Uma variável de condiç o n o garante acesso em exclus o mútua, apenas informa se uma condiç o foi ou n o satisfeita Portanto, variáveis de condiç o devem ser utilizadas em conjunto a um mutex

Concorr ncia em Acessos Memória

Variável de Condiç o

Uso:

`init` : Inicializa a variável de condiç o (`NULL` == satisfeita);

`wait` : Bloqueia a thread, aguardando a condiç o;

`signal` : Sinaliza uma das *threads* que est o aguardando que a condiç o foi satisfeita (acorda uma *thread*);

`broadcast` : Sinaliza todas as *threads* que est o aguardando que a condiç o foi satisfeita (acorda todas as *threads*)

Concorr ncia em Acessos Memória

Variável de Condiç o

Uso:

```
pthreadm utex t m;
pthreadc ond t  c;

pthreadm utex lock( &m );
while( teste )
  pthread cond wait( &c, &m );
seço  crítica
pthreadm utex unlock( &m );
```

Concorr ncia em Acessos Memória

Variável de Condiç o

Uso:

```
for(;;) { // Produtor
 it =    ;
 pthread mutex lock(&m);
 WrBuffer(it);
 nb itens++;
 pthread cond signal(&c);
 pthread mutex unlock(&m);
}
```

```
for(;;) { // Consumidor
 pthread mutex lock(&m);
 while( nb itens <= 0 )
  pthread cond wait(&m,&c);
 it = InBuffer();
 nb itens--;
 pthread mutex unlock(&m);
}
```

Sumário

Modelos de Threads

Definiç o do Padr o POSIX

O padr o POSIX define apenas a interface dos serviços

A execuç o das *threads* depende de como a biblioteca que disponibiliza os serviços foi implementada

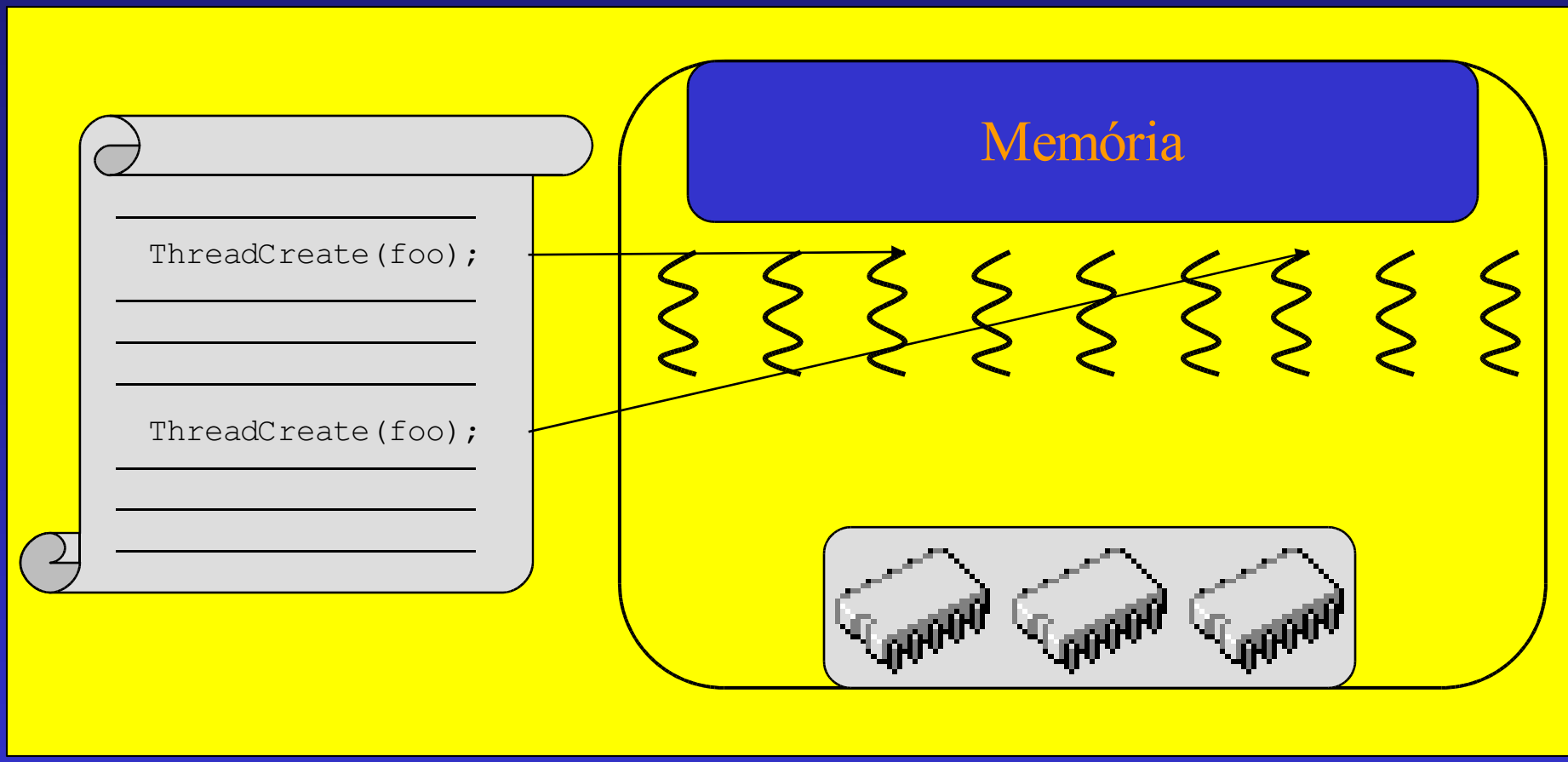
Modelos de Threads

Implementaç es

Pode ser definido no momento da criaç o da thread (atributo)
Caso a implementaç o disponibilize mais de um modelo

Modelos de Threads

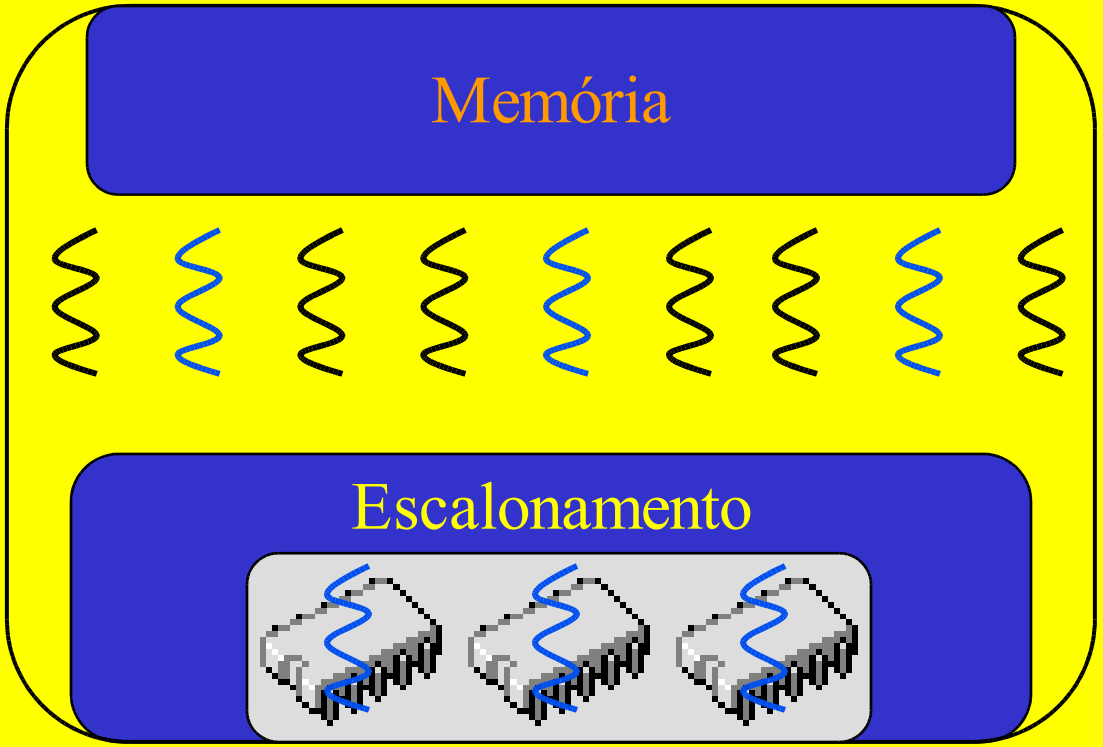
Modelos de *Threads - 1 : 1*



Modelos de Threads

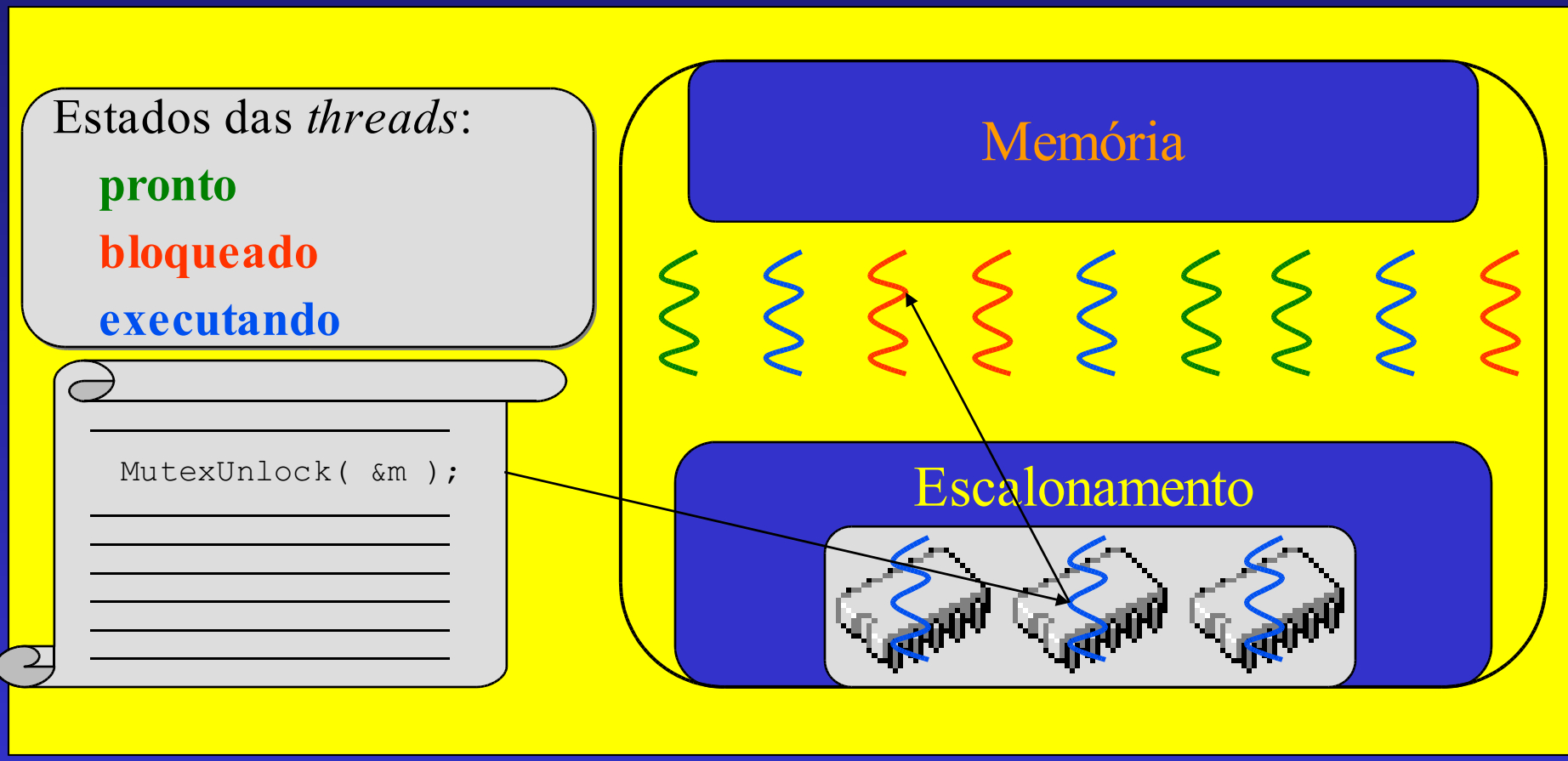
Modelos de *Threads - 1 : 1*

Mecanismo de escalonamento garantido pelo sistema



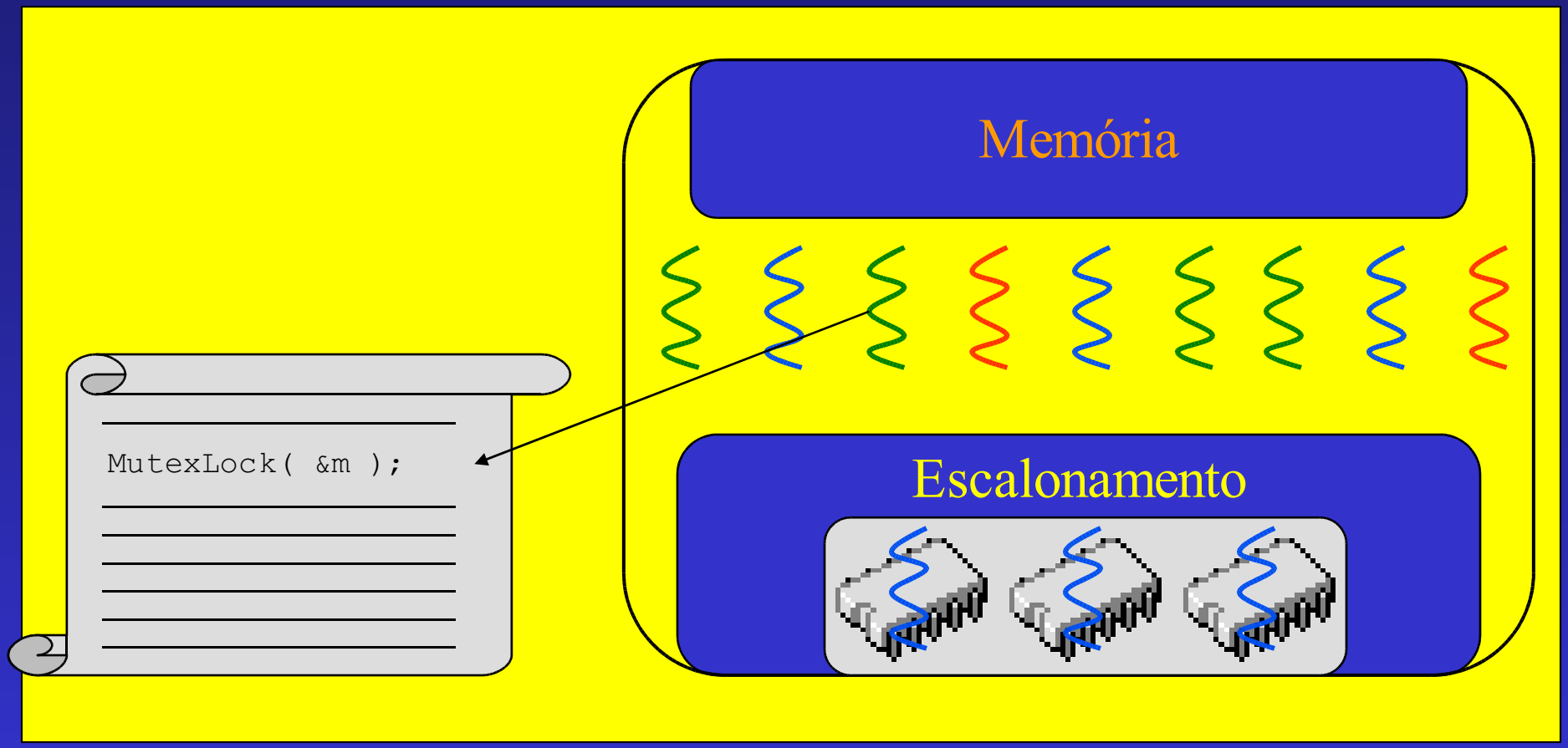
Modelos de Threads

Modelos de *Threads - 1 : 1*



Modelos de Threads

Modelos de *Threads - 1 : 1*



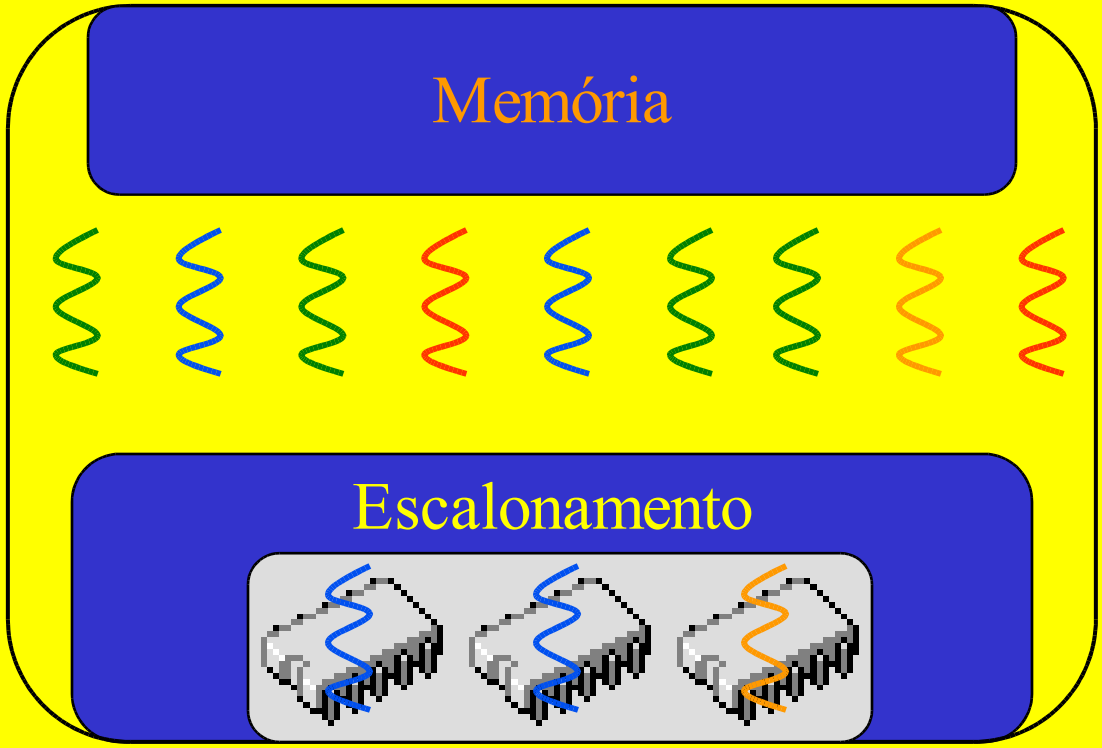
Modelos de Threads

Modelos de *Threads - 1 : 1*

Perda de processador:

Situaç o 1:
- decis o do escalonador
  (término do quantum)

A *thread* é interrompida

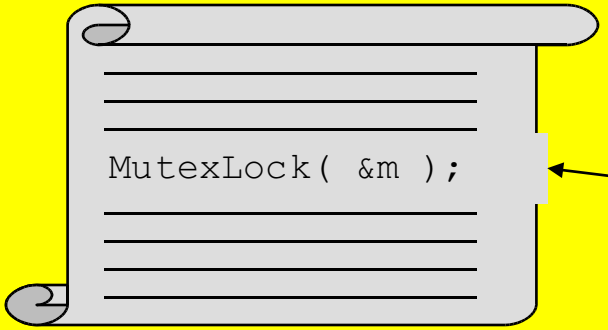


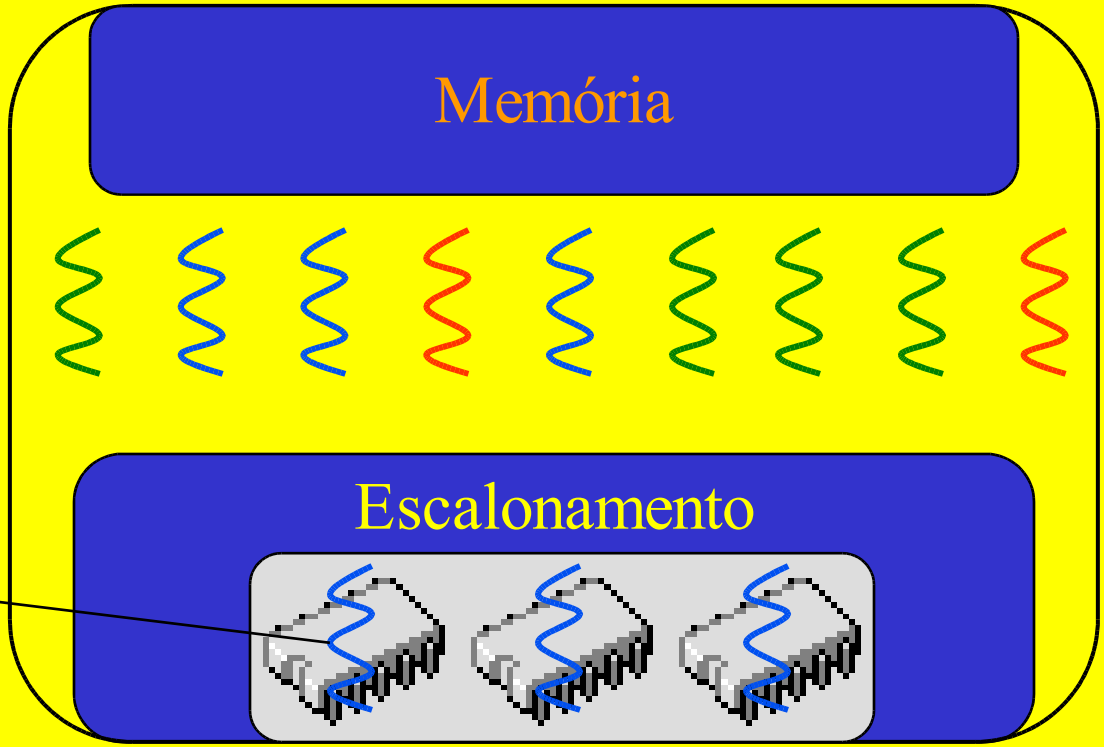
Modelos de Threads

Modelos de *Threads - 1 : 1*

Perda de processador:

Situaç o 2:
- sincronizaç o da *thread*





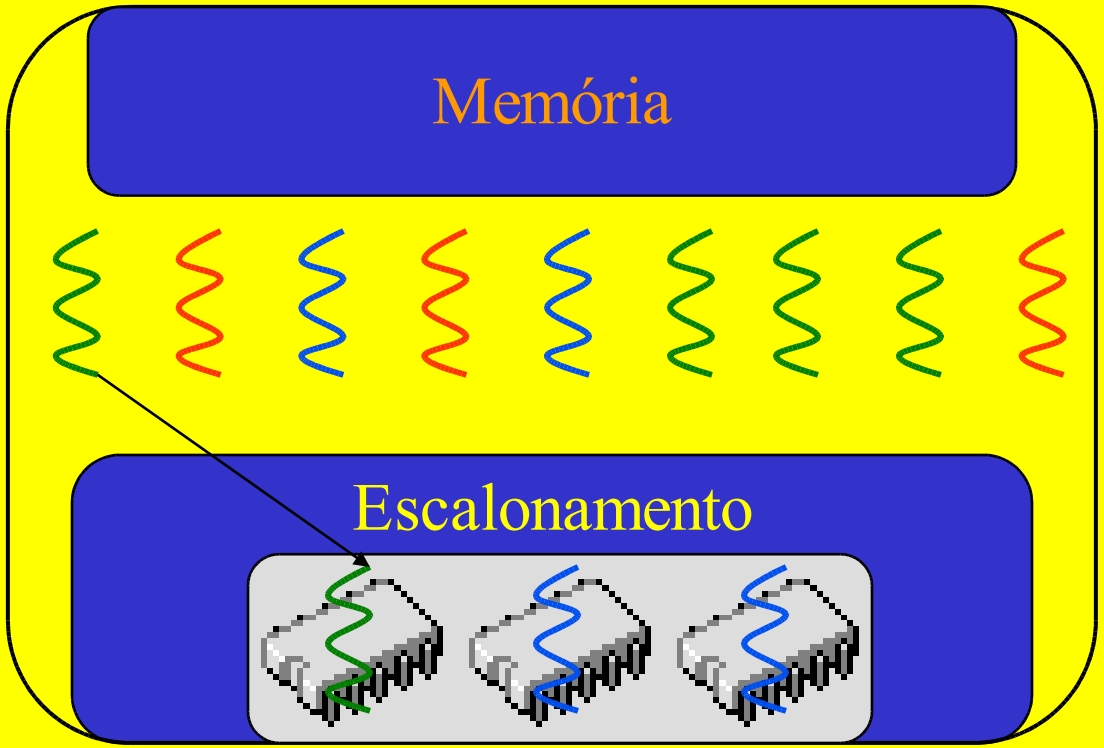
Modelos de Threads

Modelos de *Threads - 1 : 1*

Perda de processador:

Situaç o 2:
- sincronizaç o da *thread*

A *thread* é retirada do
do processador



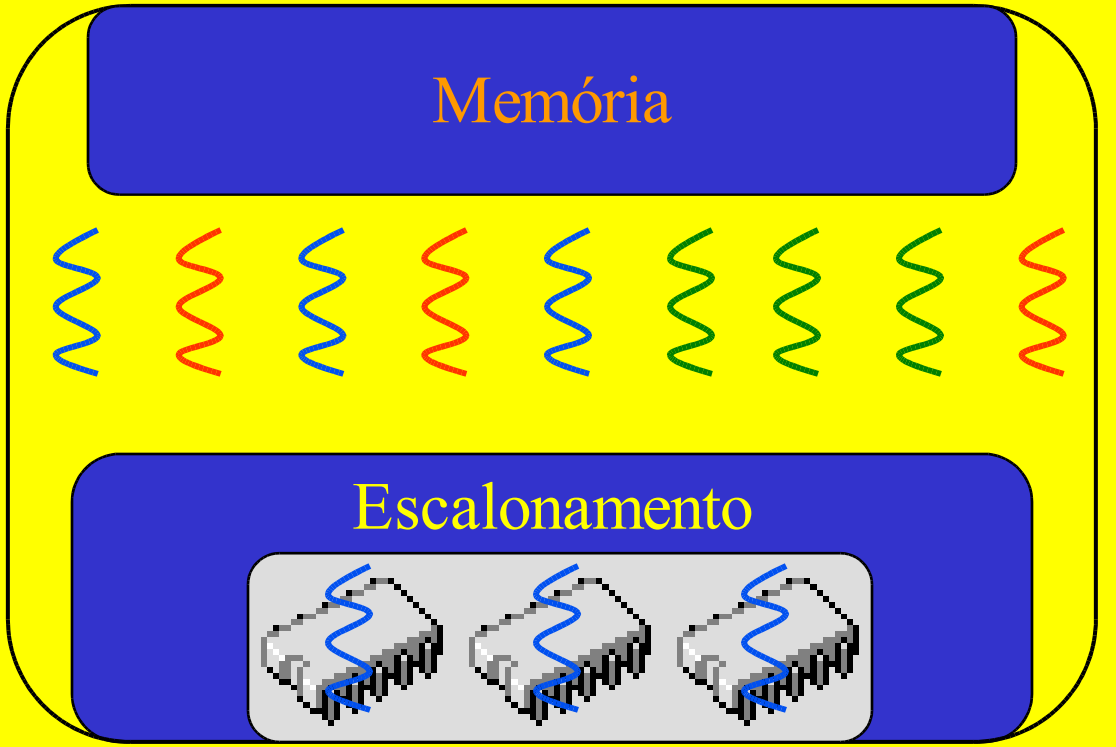
Modelos de Threads

Modelos de *Threads - 1 : 1*

Vantagens

Aumento do nível de paralelismo real na execuç o da aplicaç o

Exploraç o das arquiteturas SMP



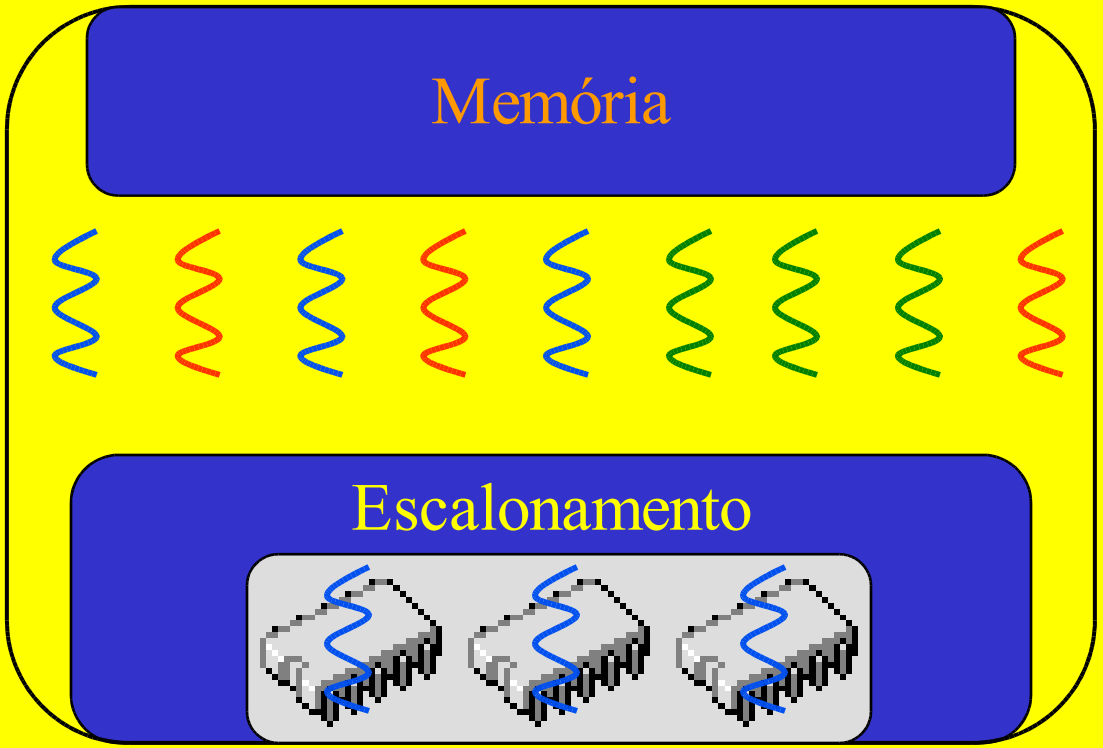
Modelos de Threads

Modelos de *Threads - 1 : 1*

Porém

As *threads* n o s o muito *leves*

Número limitado de atividades concorrentes



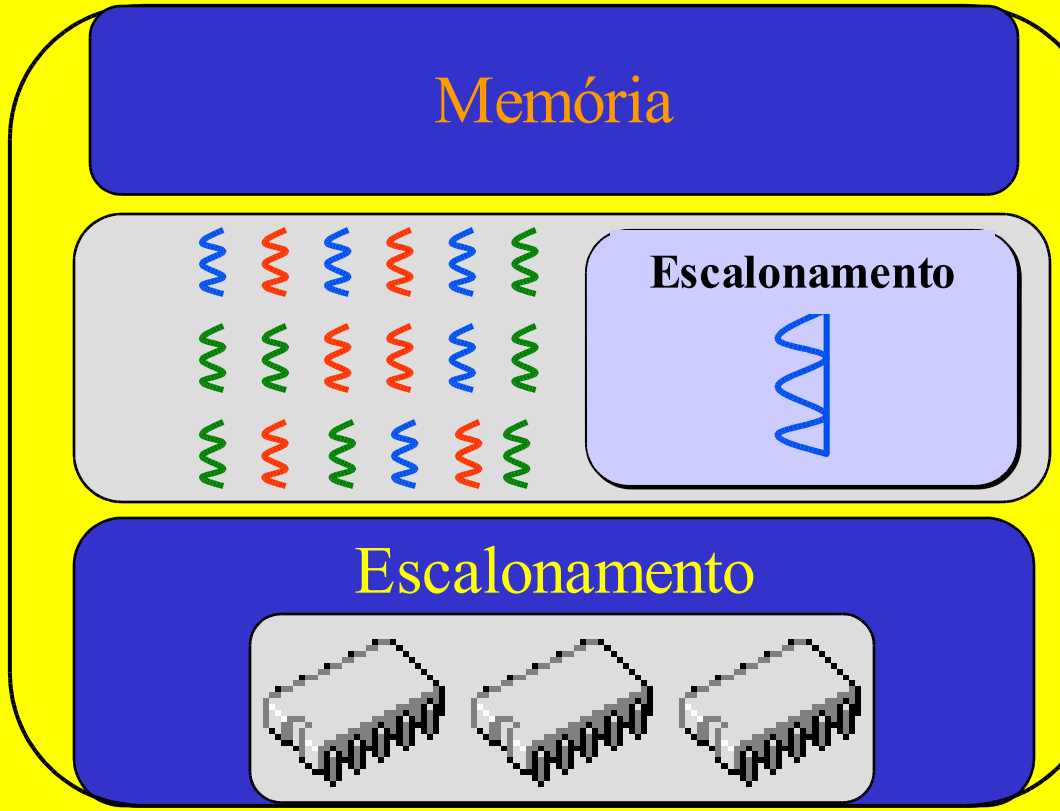
Modelos de Threads

Modelos de *Threads* - N : 1

Threads gerenciadas a nível aplicativo

espaço usuário

O processo fornece um único suporte execuç o



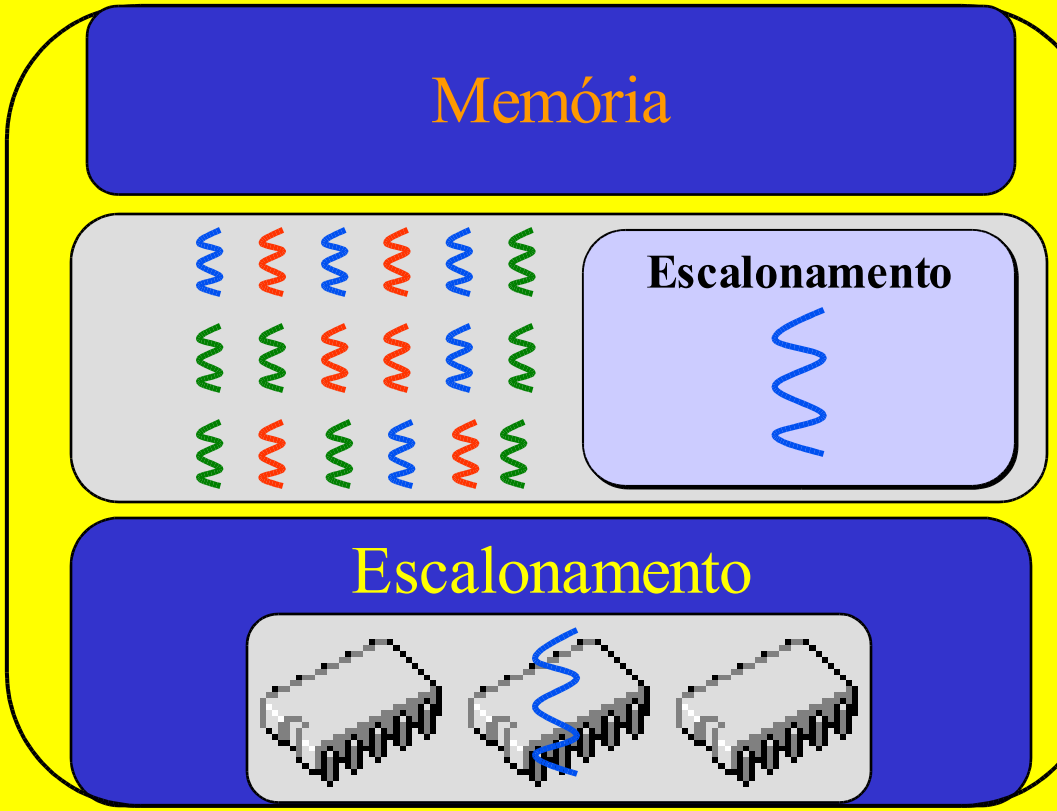
Modelos de Threads

Modelos de *Threads - N : 1*

Threads gerenciadas a nível aplicativo

espaço usuário

O processo é que sofre o escalonamento do sistema operacional



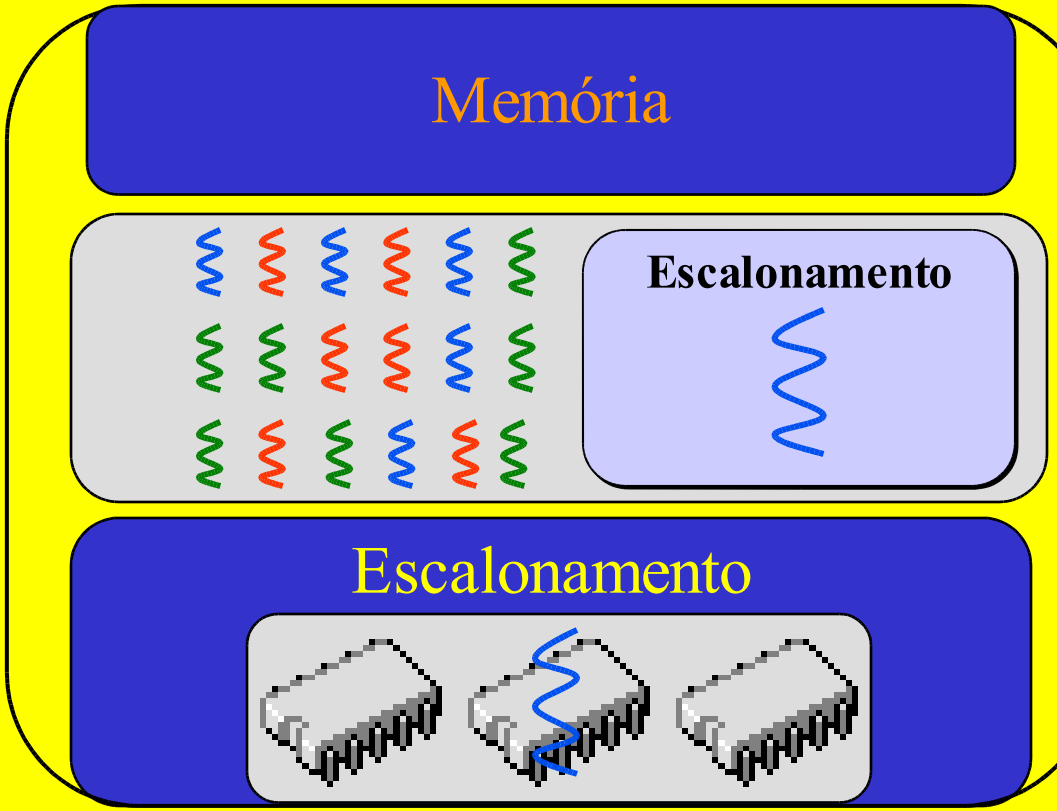
Modelos de Threads

Modelos de *Threads* - N : 1

Vantagens

Aumento do nível de concorr ncia que pode ser expresso para a aplicaç o

As *threads* s o bastante *leves*



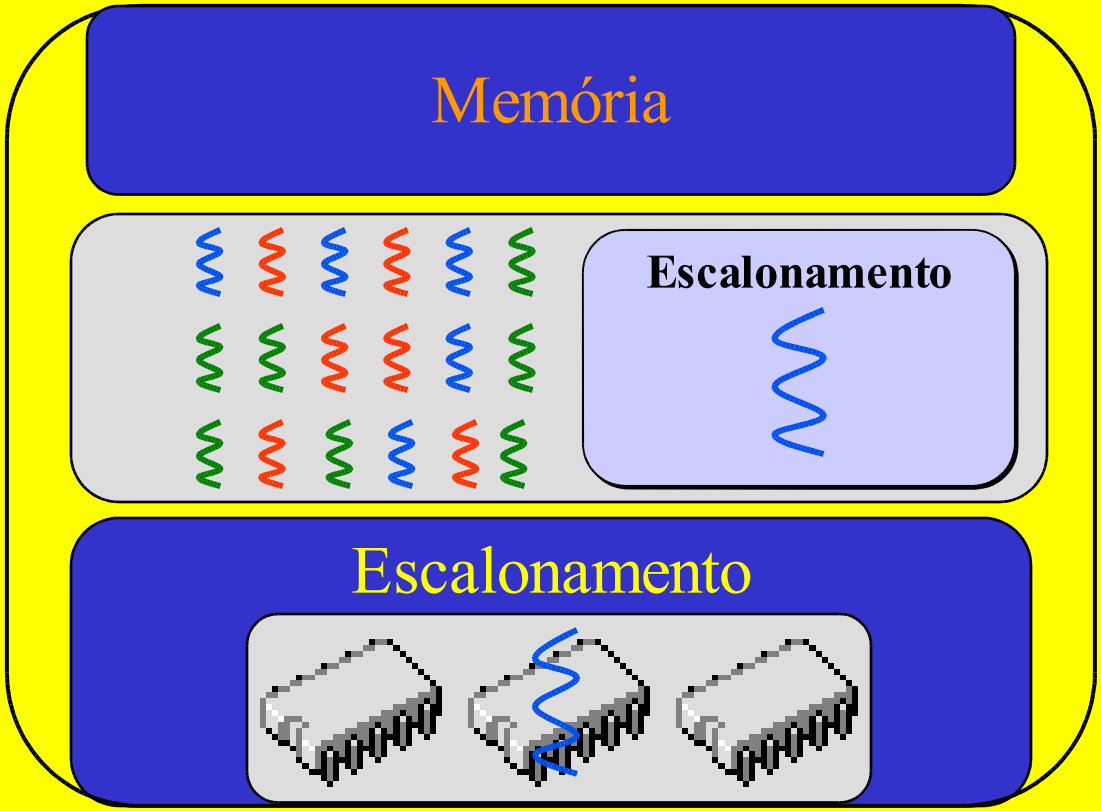
Modelos de Threads

Modelos de *Threads* - N : 1

Porém

N o explora o paralelismo que pode vir a existir em uma arquitetura SMP

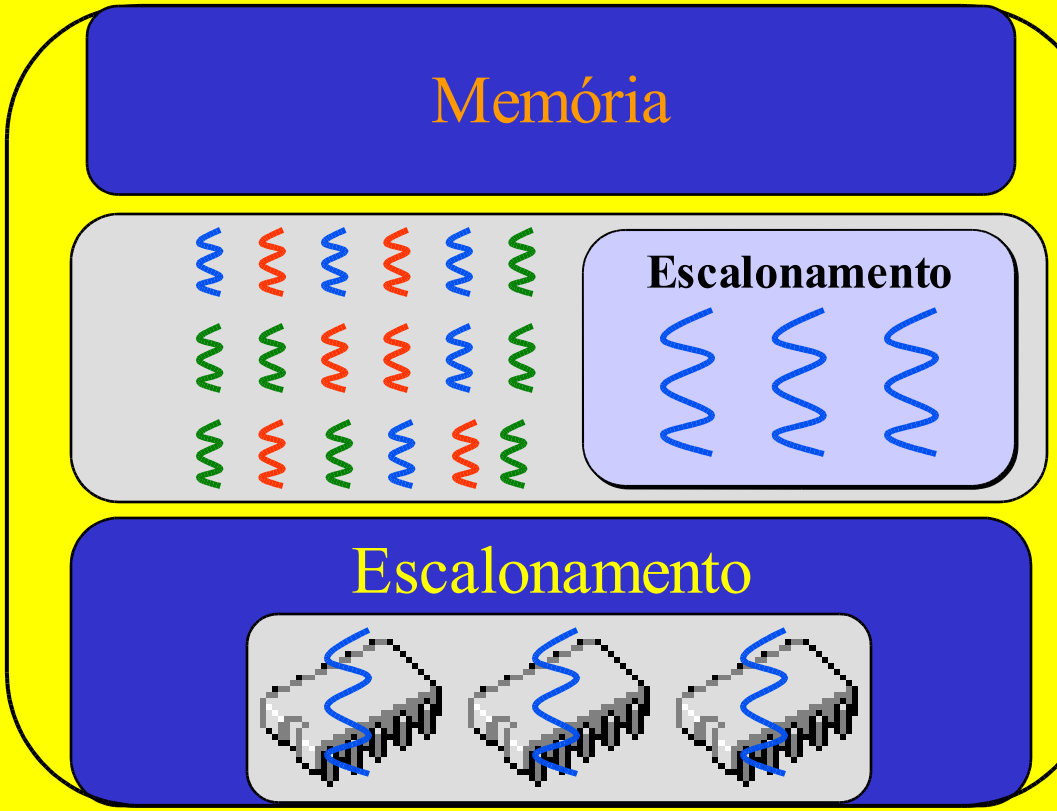
Uma *thread*, ao realizar uma chamada bloqueante (p ex recv, scanf) ao sistema, o processo é bloqueado



Modelos de Threads

Modelos de *Threads* - N : M

Modelo misto
gerenciamento tanto no espaço usuário quanto pelo sistema operacional

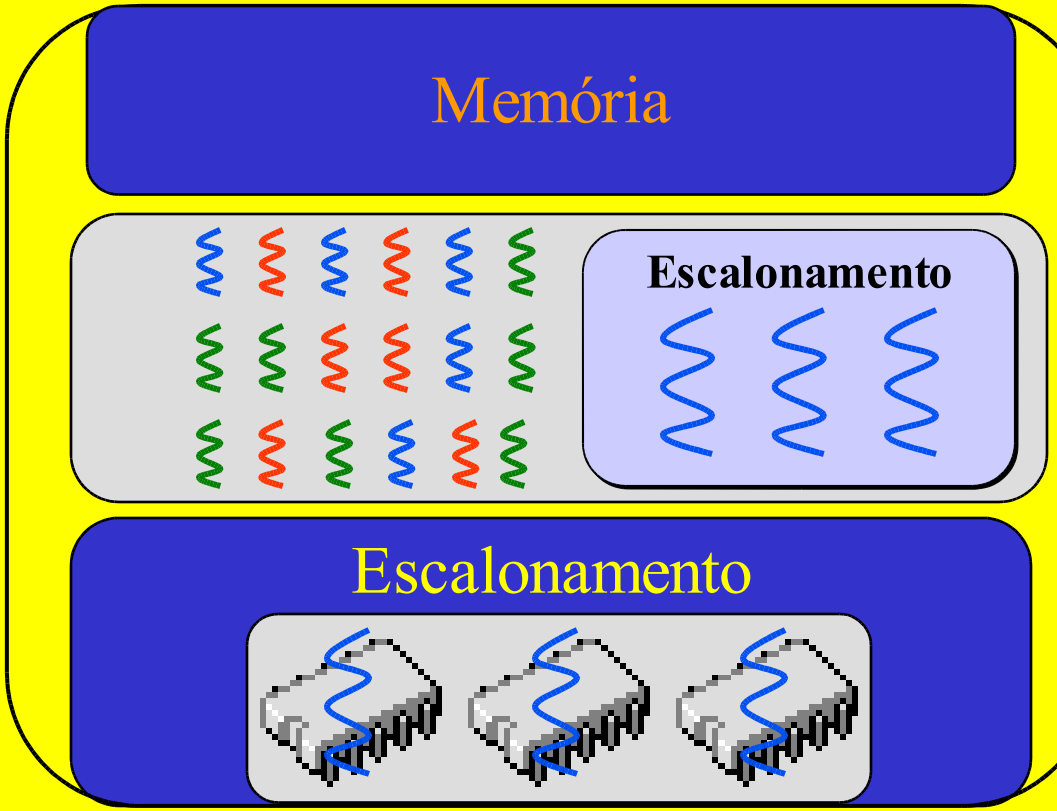


Modelos de Threads

Modelos de *Threads - N : M*

Vantagens

Separaç o entre a descriç o da concorr ncia existente na aplicaç o do paralelismo real existente na arquitetura

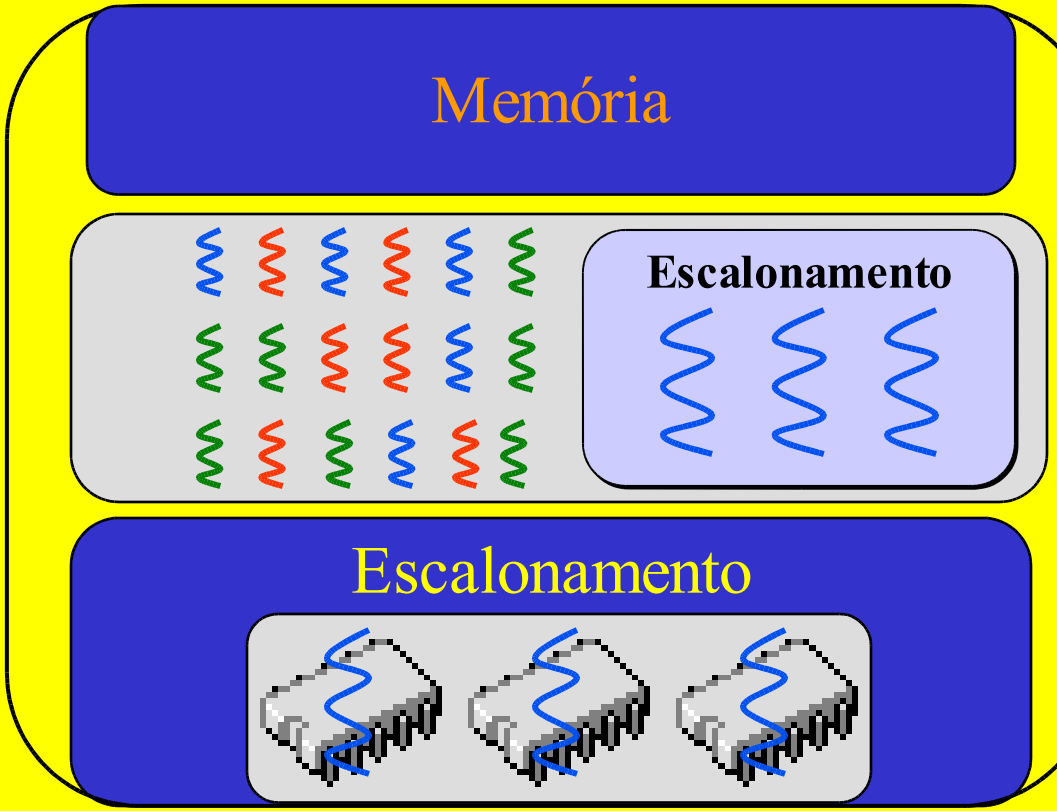


Modelos de Threads

Modelos de *Threads - N : M*

Porém

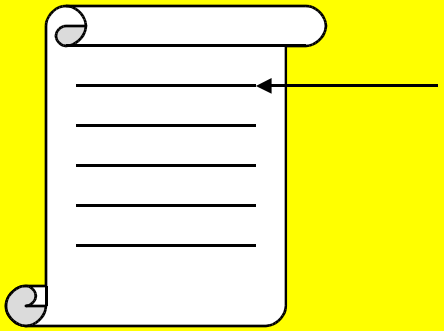
Custo de implementaç o ?



Sumário

Utilizaç o Prática

Arquivo Fonte



prog c

```
#include <pthread h>
```

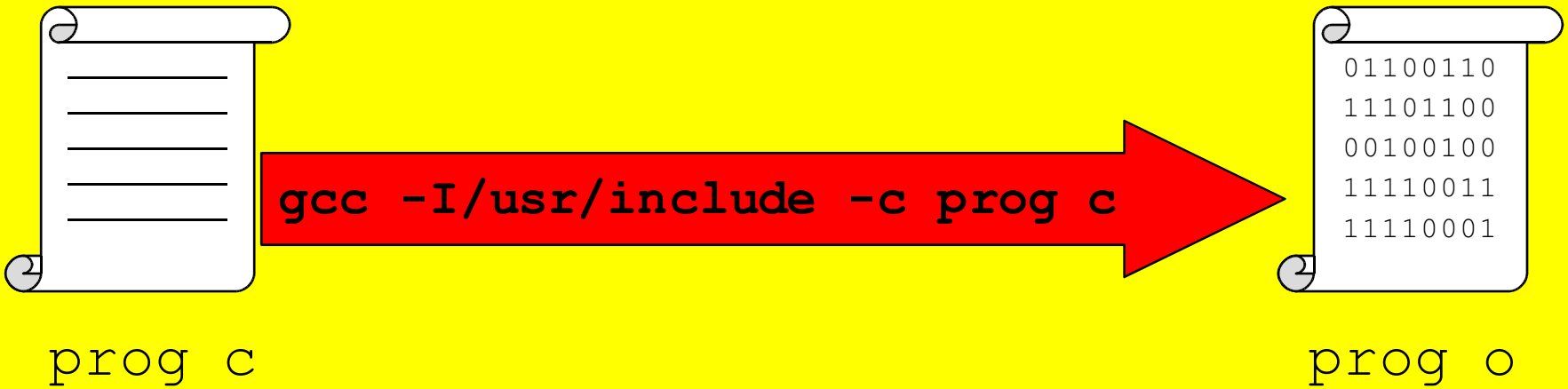
Define:

- Tipos: `pthread t`
- Serviços: `pthread`

Utilizaç o Prática

Uso no Linux

Compilaç o



Utilizaç o Prática

Uso no Linux

Linkediç o

Execuç o:

```
$> prog
```

Utilizaç o Prática

Exemplo de programa

```
Buffer b;                    int nb itens = 0;
pthread mutex t mb;          pthread cond t c;
```

```
void* Produtor( void* in ) {
 Item it;
 int i = *(int*)in;
 for( ; i > 0 ; i-- ) {
  it = ProduzItem();
  pthread mutex lock(&mb);
  Armazena(b, it);
  nb itens++;
  pthread cond signal(&c);
  pthread mutex unlock(&mb);
 }
 return 0;
}
```

```
void* Consumidor(void* in ) {
 Item it;
 int i = *(int*)in;
 for( ; i > 0 ; i-- ) {
  pthread mutex lock(&mb);
  while( nb item <= 0 )
   pthread cond wait(&c,&mb);
  it = LeBuffer(b);
  nb itens--;
  pthread mutex unlock(&mb);
  ConsomeItem(it);
 }
 return 0;
}
```

Utilizaç o Prática

Exemplo de programa

```
Buffer b;                    int nb itens = 0;
pthread m
```

```
void* Pro
 Item it;
 int i =
 for(; i
  it = Pr
  pthread
  Armazen
  nb iten
  pthread
  pthread
 }
 return 0
}
```

```
#include <pthread h>

int main() {
 pthread t prod, cons;
 int quant = 10;

 pthread create( &prod, NULL, Produtor, &quant );
 pthread create( &cons, NULL, Produtor, &quant );

 pthread join( prod, NULL );
 pthread join( cons, NULL );

 return 0;
}
```

```
}
```

Utilizaç o Prática

Exemplo de programa

```
void* Produtor( void* in) {
 Item it;
 it = ProduzItem();
 return it;
}
```

```
void* Consumidor(void* prod) {
 Item it;
 pthread t prod = (produto t*)dta->prod;
 pthread join( prod, &it; );
 ConsomeItem( it );
 return 0;
}
```

Utilizaç o Prática

Exemplo de programa

```
#include <pthread h>

int main() {
 pthread t prod[10], cons[10];
 int i, quant = 10;

 for( i = 0 ; i < 10 ; i++ ) {
  pthread create( &(prod[i]), NULL, Produtor, NULL );
  pthread create( &(cons[i]), NULL, Produtor, &(prod[i]) );
 }
 for( i = 0 ; i < 10 ; i++ )
   pthread join( cons, NULL );

 return 0;
}
```

Sumário

Quest es de Projeto

1:1, 1:N, N:M

Compromisso

Depende dos recursos disponíveis (programaç o e hardware);

N:M apresenta um melhor índice teórico, contudo gera custos adicionais;

Opç o: *Pool de Execuç o* (caso existam *threads* 1:1)

Quest es de Projeto

1:1, 1:N, N:M

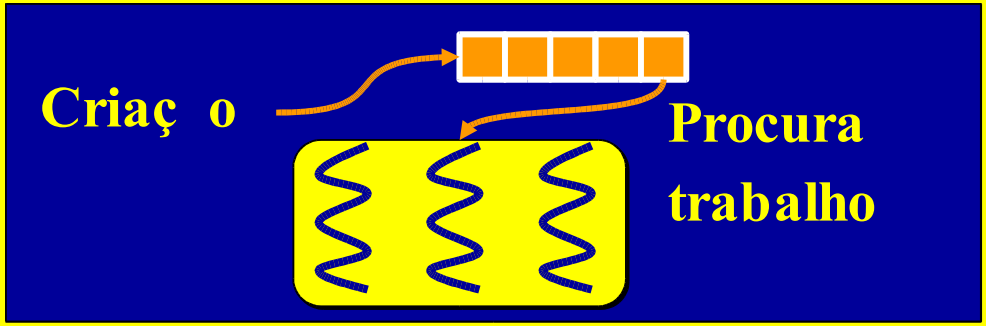
Compromisso

Depende dos recursos disponíveis (programaç o e hardware);

N:M apresenta um melhor índice teórico, contudo gera custos adicionais;

Opç o: *Pool de Execuç o* (caso existam *threads* 1:1)

Evitar sincronizaç es



Quest es de Projeto

Fraç o Serial

Limite da Aplicaç o

Speedup: $S_p = \frac{T_1}{T_p}$

Efici ncia: $E_p = \frac{S_p}{p}$

Desejado: $S_p = p$

Quest es de Projeto

Fraç o Serial

Limite da Aplicaç o

Speedup: $S_p = \frac{T_1}{T_p}$

Efici ncia: $E_p = \frac{S_p}{p}$

Fraç o Serial: $S_m = \frac{1}{(1 \quad F_p)}$

Lei de Amdhal: o limite do desempenho de um programa é determinado por uma fraç o serial

Quest es de Projeto

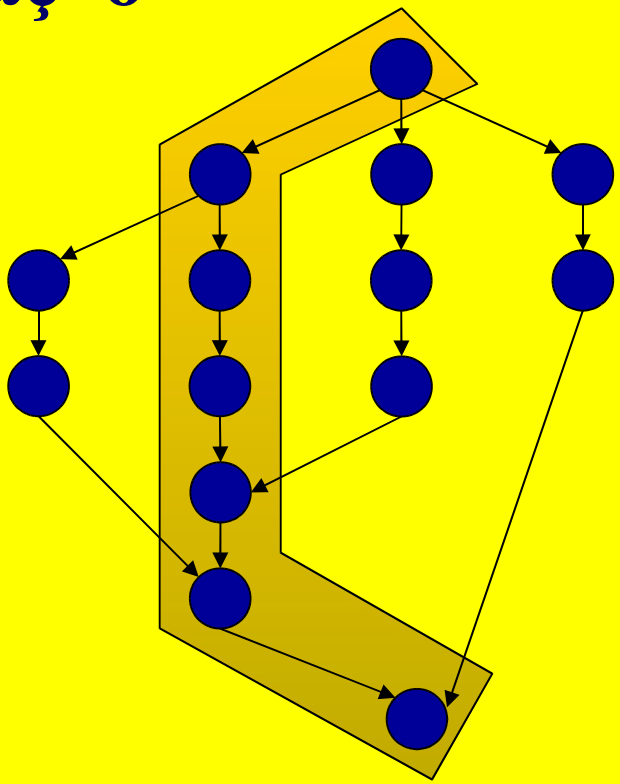
Fraç o Serial

Limite da Aplicaç o

Caminho Crítico

Tempo mínimo de execuç o:

Somatório de todas as tarefas no caminho crítico



Quest es de Projeto

Granulosidade

Sincronizaç es entre tarefas

Relaç o entre as tarefas e as trocas de dados

Mais sincronizaç es, menos concorr ncia potencial

Reduç o das seç es críticas

Grossa *vs* Fina
depende dos recursos utilizados

Quest es de Projeto

Pool de Execuç o

Threads de Serviço

Atividades concorrentes: tarefas

Lista de tarefas

Consumo de tarefas por threads dedicadas ao cálculo

Problema: compartilhamento de dados (*deadlock*)

Vantagem: aproxima o modelo N:M

Sumário

Threads em Outros Ambientes para o PAD

Java

Interface definida na linguagem

Herança de classe virtual

Sincronizaç o: monitores

Passagem de parâmetros e resultados: via métodos

Implementado na máquina virtual

Threads em Outros Ambientes para o PAD

Cilk

Extens o a C

Spawn / Sync : definem tarefas

Comunicaç o via memória (entrada e saída de tarefas)

Pool de Execuç o

Escalonamento eficiente

Threads em Outros Ambientes para o PAD

Anahy

Subconjunto de POSIX *threads*

athread_create / athread_join

Comunicação via memória (entrada e saída de tarefas)

Pool de Execução

Escalonamento eficiente
focando SMP e agregados de computadores

Anahy

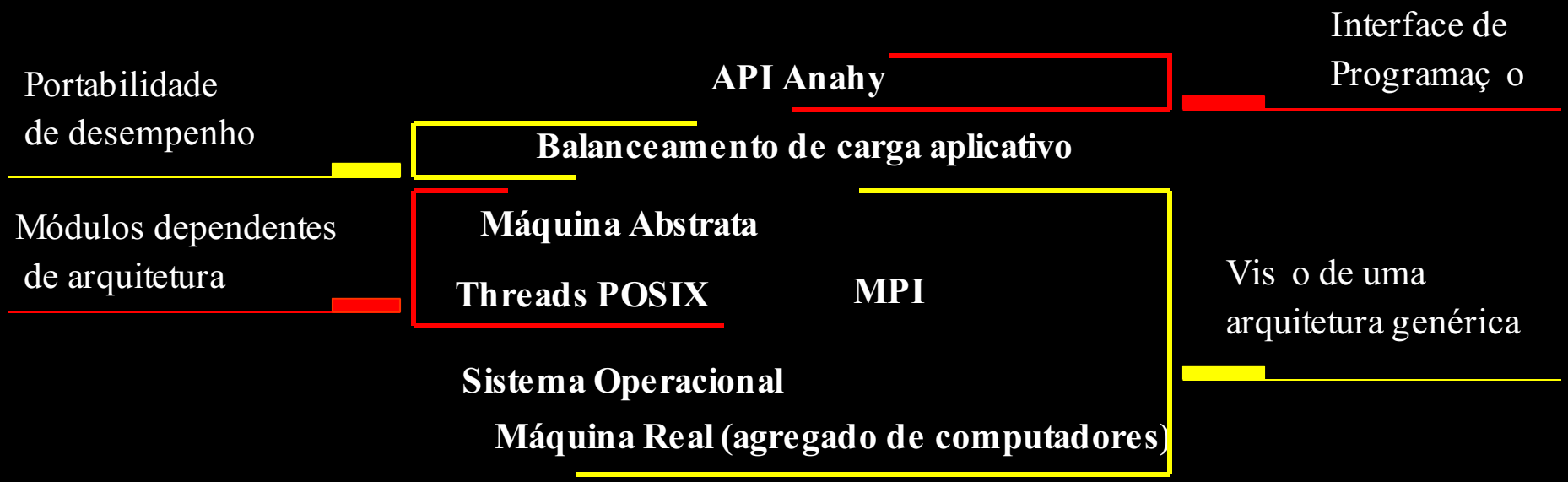
Ambiente de Processamento de Alto Desempenho

CNPq, FAPERGS, UNISINOS

PIPCA - Programa Interdisciplinar de Pós Graduaç o em Computaç o Aplicada
Centro de Ci ncias Exatas e Tecnológicas - UNISINOS

Anahy

Ambiente de Processamento de Alto Desempenho



PIPCA Programa Interdisciplinar de Pós Graduaç o em Computaç o Aplicada

UNISINOS Universidade do Vale do Rio dos Sinos

Desempenho

Avaliaç o do número de *threads*

Algoritmo de Fibonacci

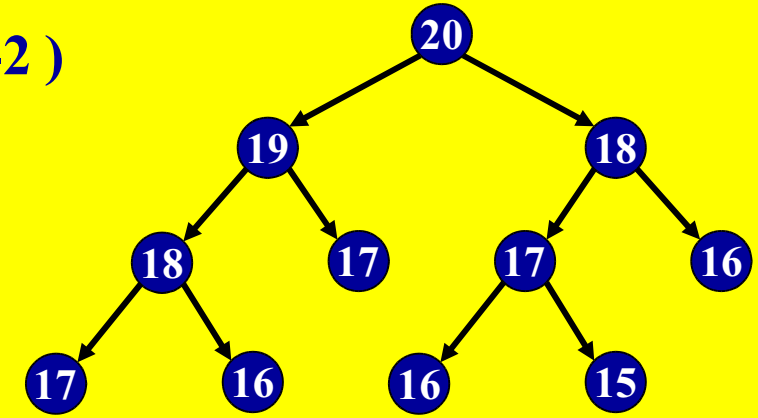
Fibo(n), n > 2, = Fibo(n-1) Fibo(n-2)

Fibo(2), n 2, = 1

Grande número de tarefas

Fibo(15) = 1218

Fibo(20) = 13528



Aplicaç o altamente concorrente
Distribuiç o irregular de carga computacional

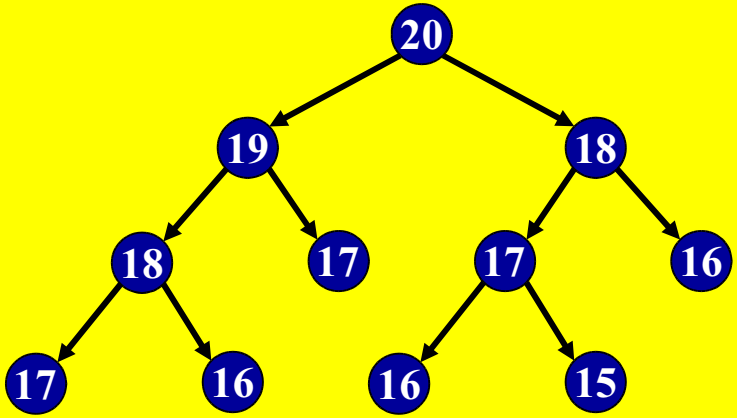
Desempenho

Avaliaç o do número de *threads*

Algoritmo de Fibonacci

Pthreads

	Mono	Bi
15	**6,4**	**3,2**
16	**17,2**	**10,1**



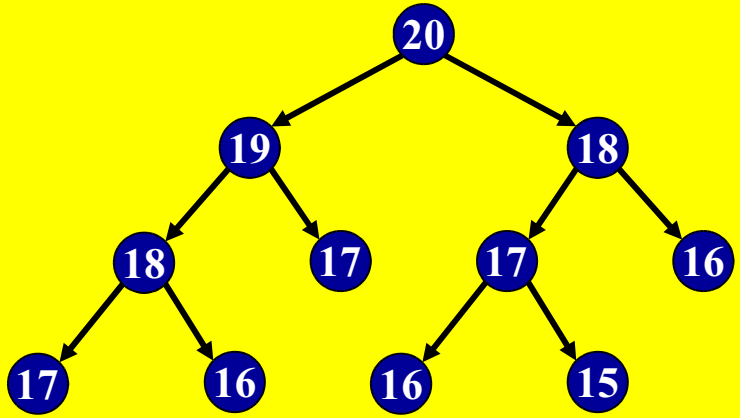
Limite do número de *threads*

Desempenho

Avaliaç o do número de *threads*

Algoritmo de Fibonacci

	Pool de Execuç o		Pthreads	
	Mono	Bi	Mono	Bi
15	0,06	0,03	6,4	3,2
16	0,10	0,08	17,2	10,1
17	0,16	0,09	--	--
20	3,6	2,8	--	--



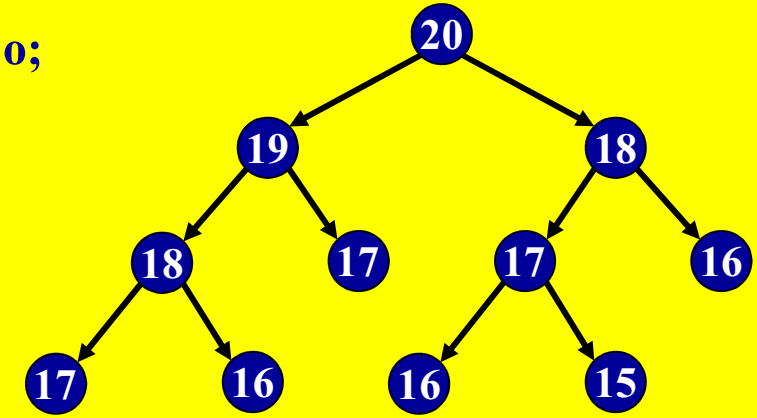
Limite do número de *threads*

Desempenho

Balanceamento de carga

Algoritmo de Fibonacci

Número limitado de *threads* no pool de execuç o;



Tempo de execuç o *vs* Consumo de memória

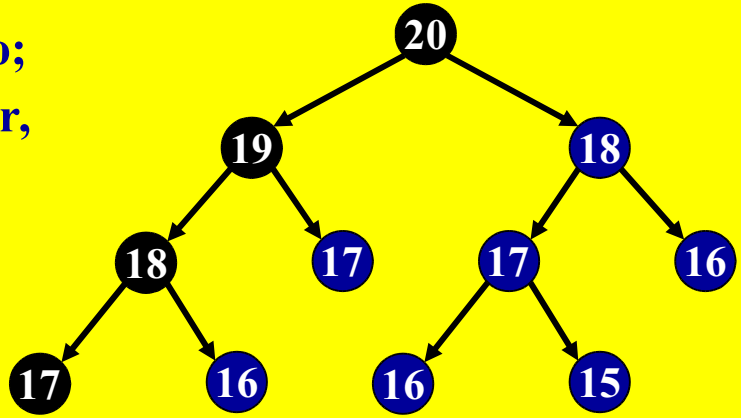
Desempenho

Balanceamento de carga

Algoritmo de Fibonacci

Número limitado de *threads* no pool de execuç o;

Caso um processador PRETO comece a executar, ele ficará responsável por um caminho ;



Ordem Lexicográfica

Desempenho

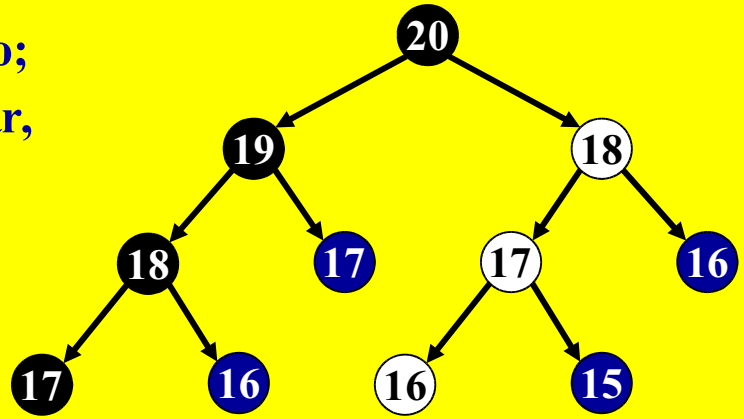
Balanceamento de carga

Algoritmo de Fibonacci

Número limitado de *threads* no pool de execuç o;

Caso um processador PRETO comece a executar, ele ficará responsável por um caminho ;

Um processador BRANCO cará responsável por outro caminho



Tempo de execuç o *vs* Consumo de memória

Desempenho

Balanceamento de carga

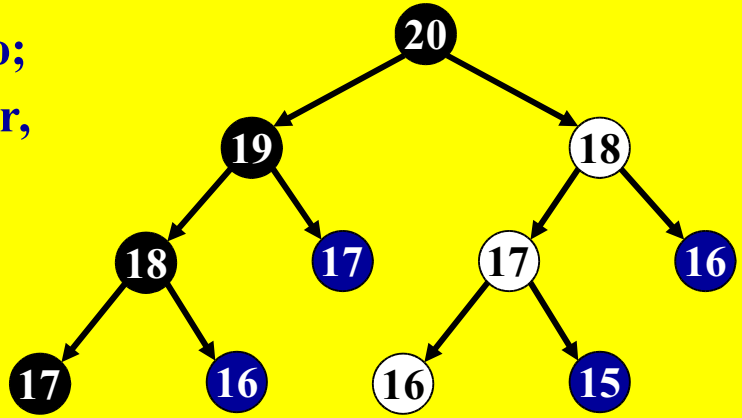
Algoritmo de Fibonacci

Número limitado de *threads* no pool de execuç o;

Caso um processador PRETO comece a executar, ele ficará responsável por um caminho ;

Um processador BRANCO cará responsável por outro caminho

E um VERMELHO ?



Tempo de execuç o *vs* Consumo de memória

Desempenho

Balanceamento de carga

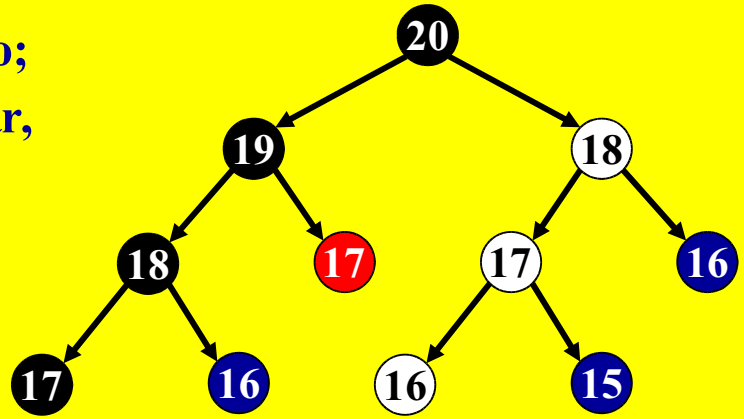
Algoritmo de Fibonacci

Número limitado de *threads* no pool de execuç o;

Caso um processador PRETO comece a executar, ele ficará responsável por um caminho ;

Um processador BRANCO cará responsável por outro caminho

E um VERMELHO ?



Diferentes estratégias fazem com que o tempo de execuç o varie de 20 minutos para 4 segundos, Fibo(20)