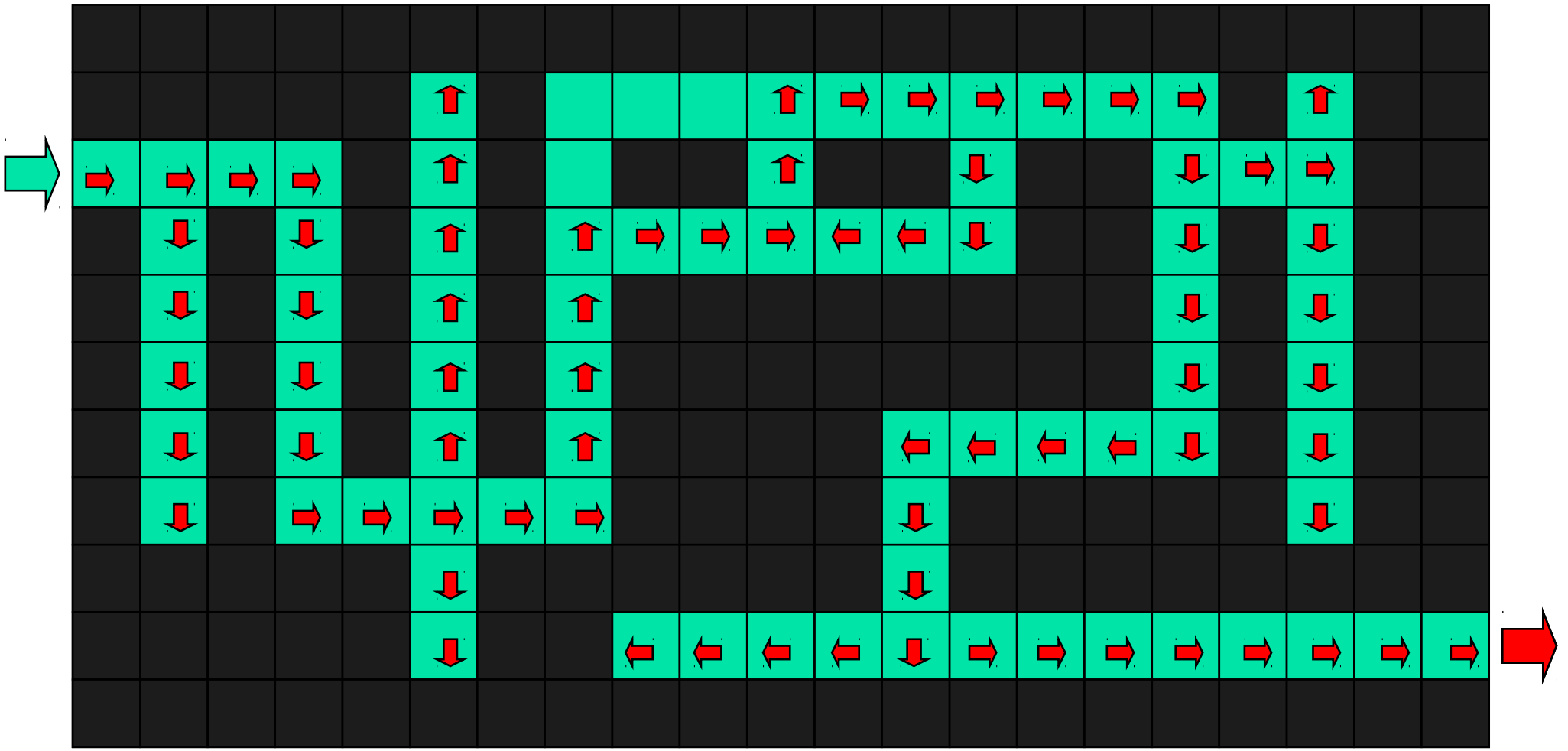
Programação concorrente (processos e *threads*)

Programação concorrente

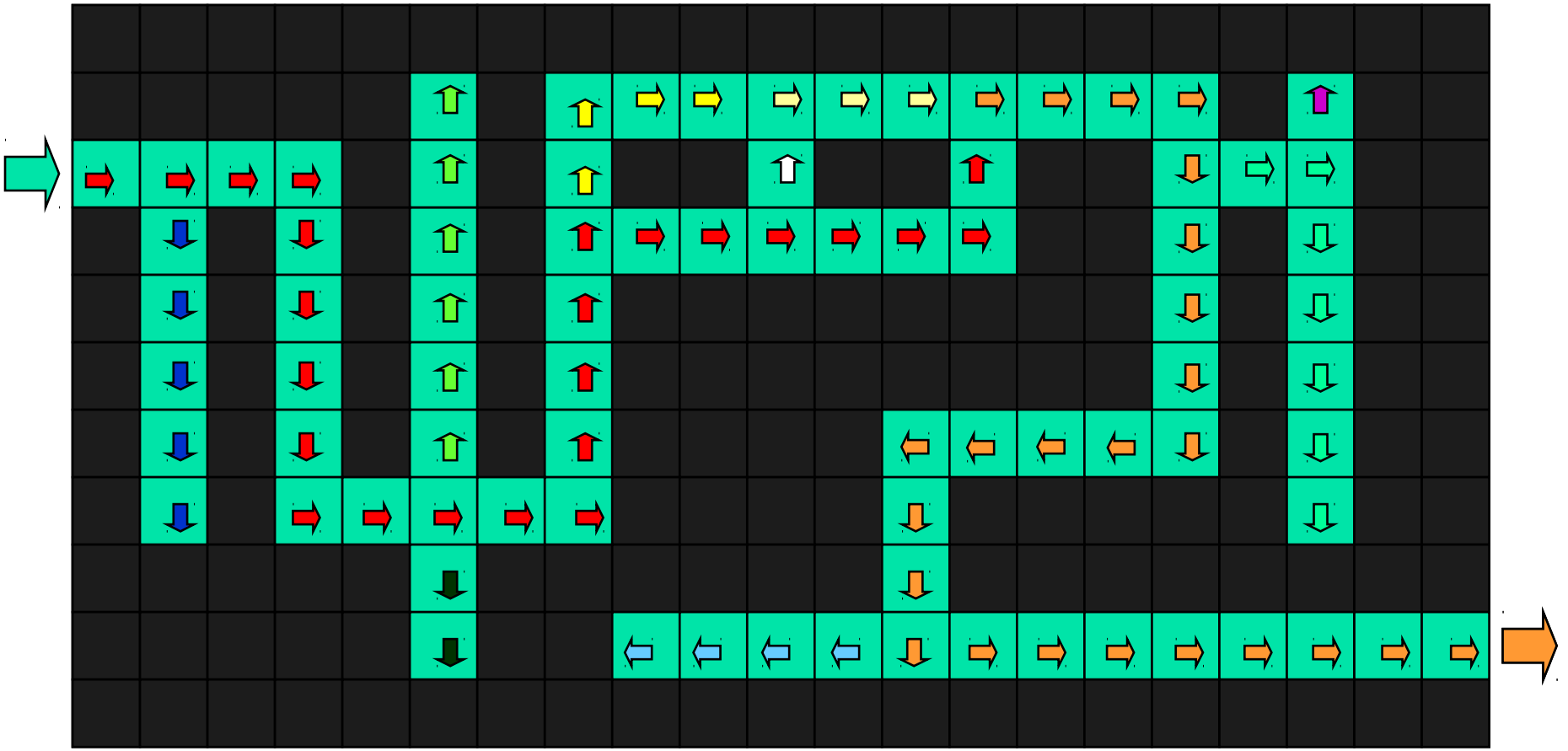
Por que precisamos dela?

- Para utilizar o processador completamente
  - Paralelismo entre CPU e dispositivos de I/O
- Para modelar o paralelismo do mundo real
- Para que mais de um computador/processador possa ser utilizado para resolver o problema
  - Considere como exemplo encontrar o caminho através de um labirinto

Busca Sequencial no Labirinto



Busca Concorrente no Labirinto



Concorrência na linguagem ou no SO?

Há um longo debate sobre se a concorrência deve ser definida na linguagem de programação ou deixada para o sistema operacional

- Ada, Java e C# fornecem concorrência
- C e C++ não

Implementação de tarefas concorrentes

Formas de implementar uma coleção de tarefas executando concorrentemente

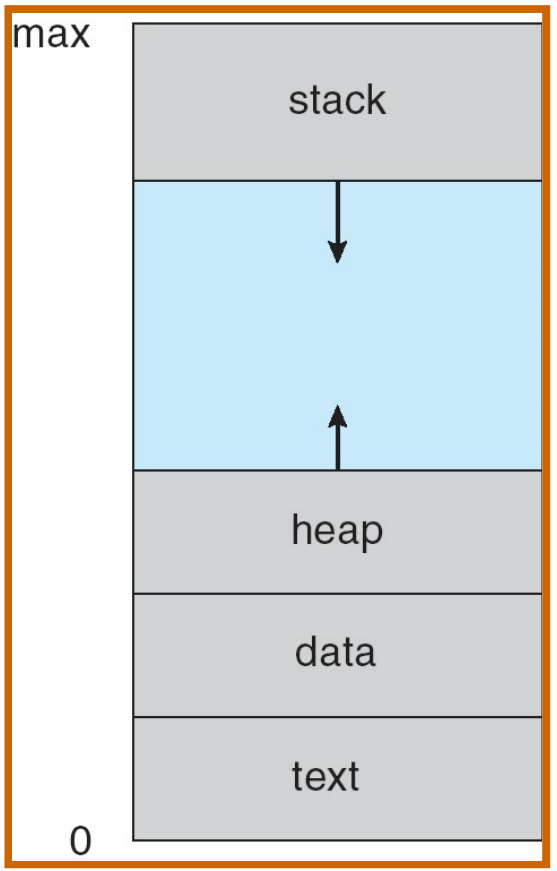
- Multiprogramação
  as tarefas multiplexam suas execuções num único processador (pseudo-paralelismo)

- Multiprocessamento
  as tarefas multiplexam suas execuções num sistema multiprocessador onde há acesso a uma memória compartilhada (acoplamento forte. e.g.: multicore)

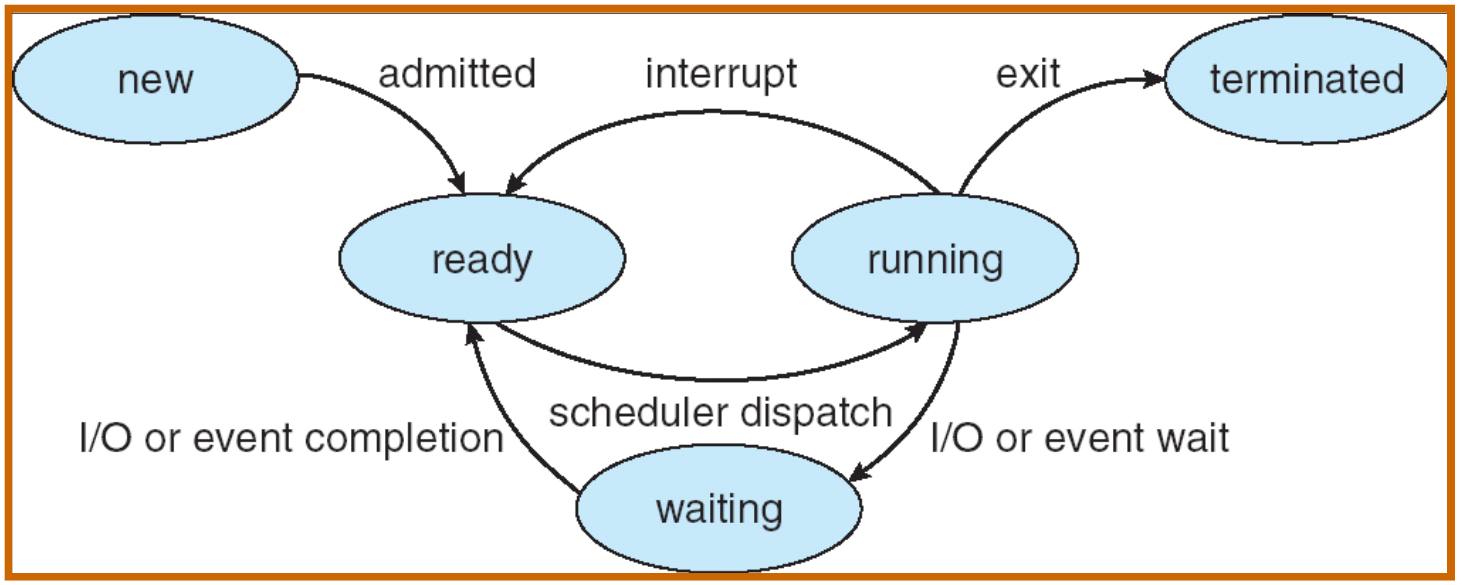
- Processamento distribuído
  as tarefas multiplexam suas execuções em vários processadores que não compartilham memória (acoplamento fraco. e.g. LAN´s, WAN,s)

Conceito de processo

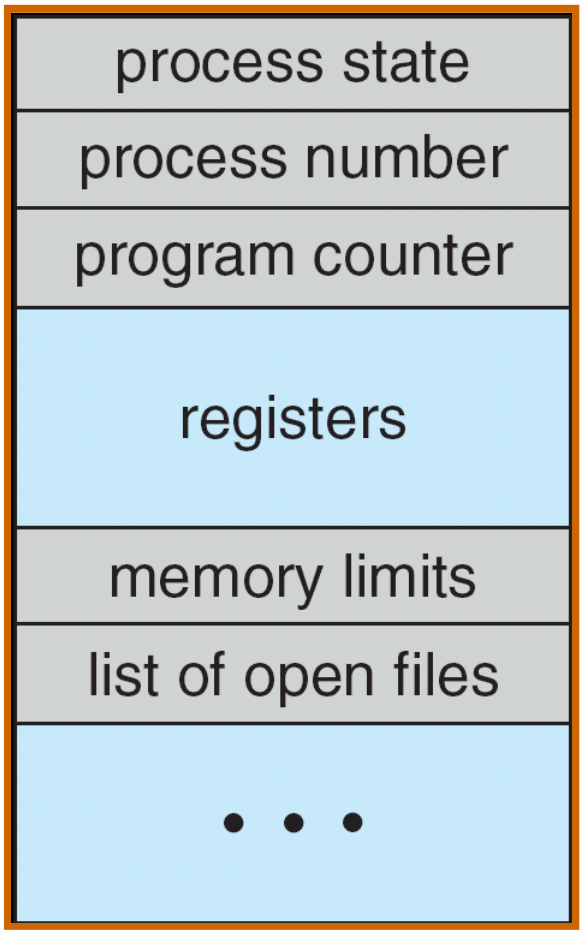
Inclui: contador de programa (PC), pilha, dados



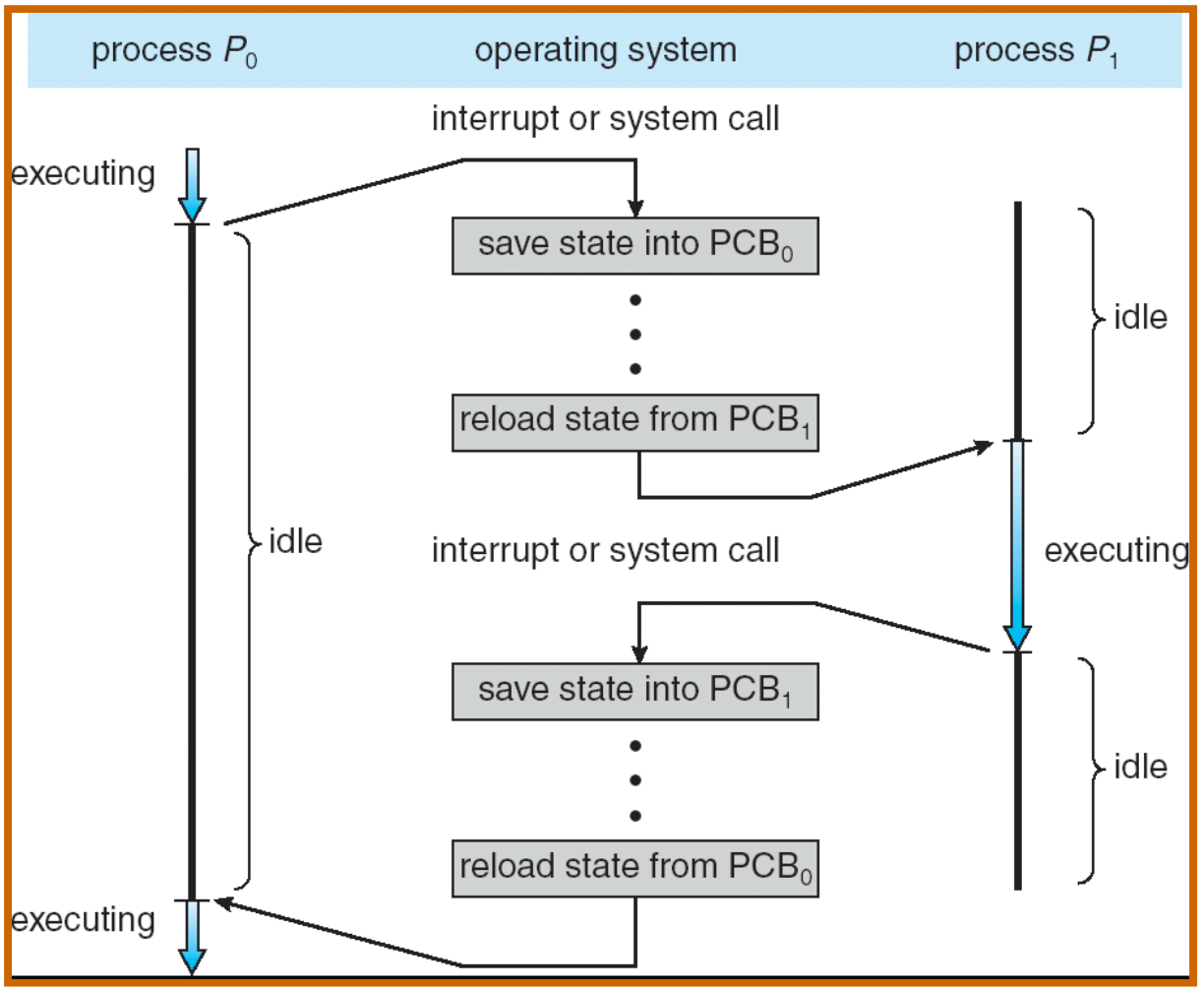
Estados de um processo (tarefa)



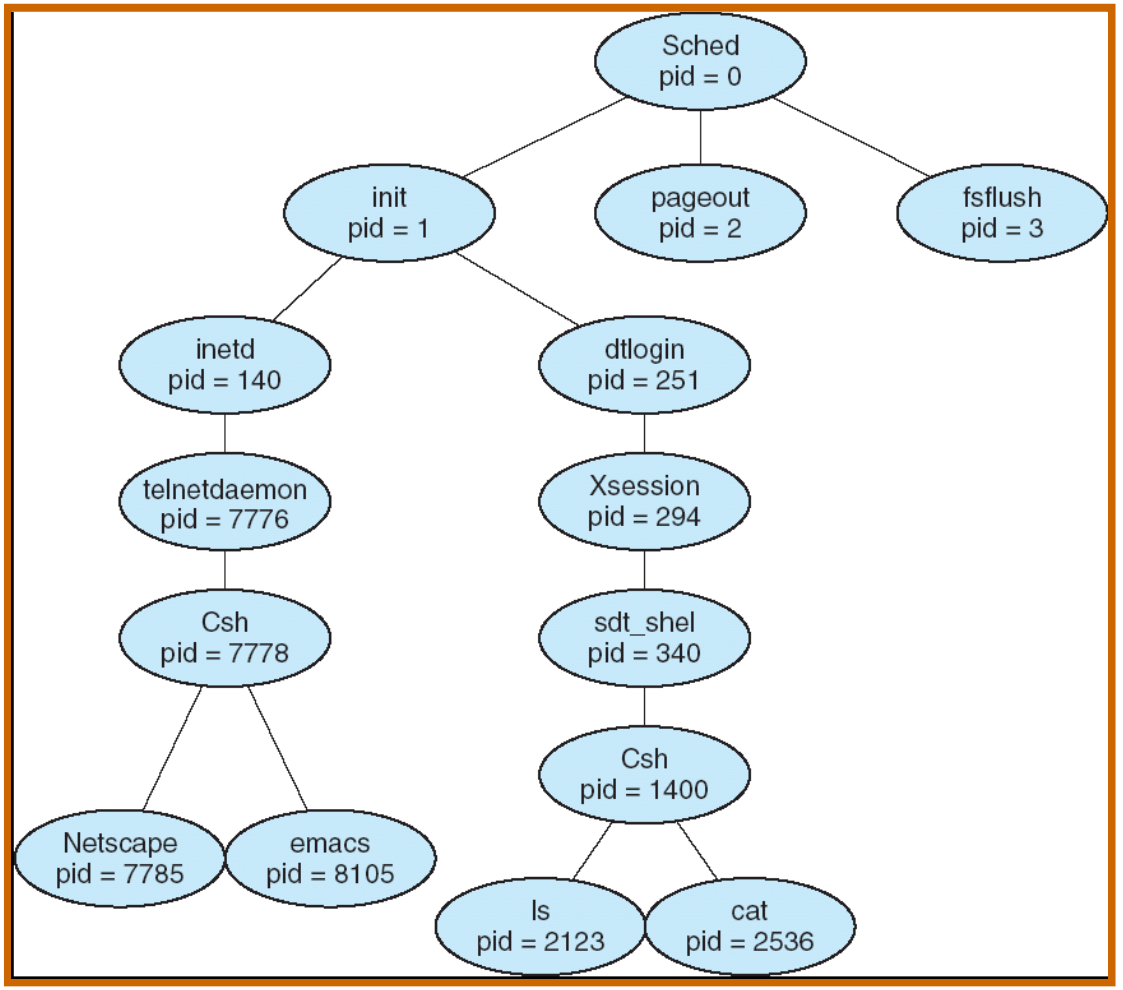
Process Control Block (PCB)



Chaveamento de contexto

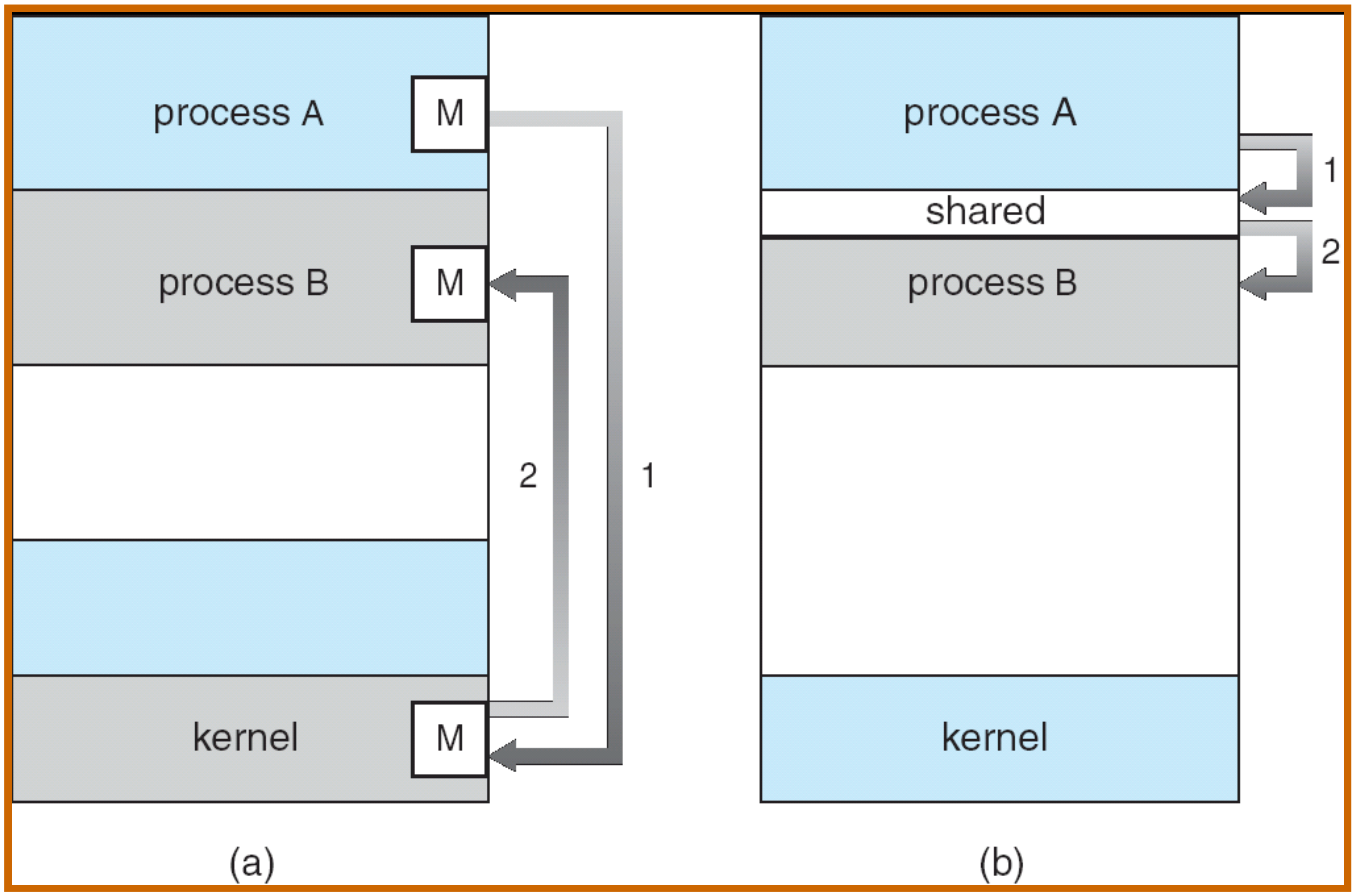


Árvore de Processos



IPC (comunicação entre processos)

- Memória compatilhada
- Troca de mensagens
- Pipes
- Semáforos
- Monitores
- *Sockets*
- RMI (*remote method invotation*)
- RPC (*remote procedure calls*)

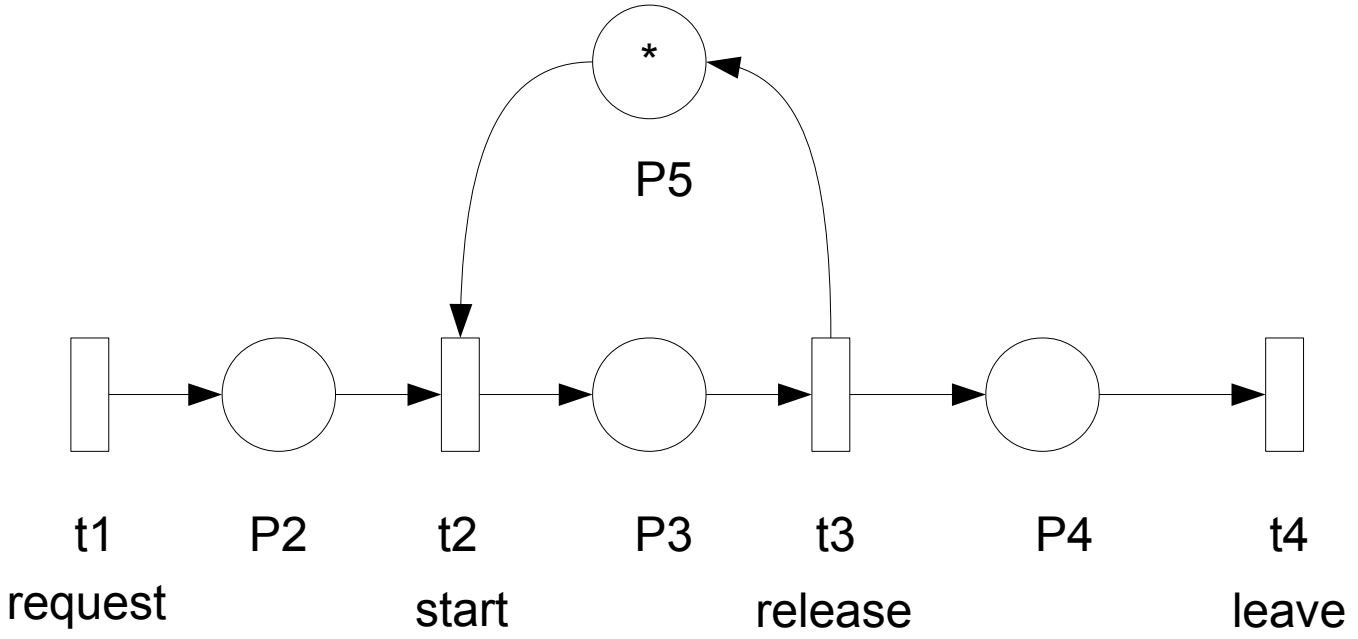


Problemas de concorrência

- Condições de disputa: o resultado depende da ordem de execução (imprevisível)
- Regiões críticas: partes do código onde ocorrem as condições de disputa. Solução: exclusão mútua e.g. **Memória compartilhada**: x=x+1; carrega o valor de x num registrador, incrementa, armazena o valor do registrador em x.

Princípios para boa solução de exclusão mútua

1) Nunca dois processos simultaneamente em suas regiões críticas
2) Nada pode ser afirmado sobre velocidade ou número de CPUs
3) Nenhum processo fora da sua região crítica pode bloquear outros
4) Nenhum processo pode esperar eternamente para entrar em sua região crítica



Semáforos

- São números inteiros: *typedef int Semaphore;*
- Seu valor pode apenas ser inicializado: *Semaphore x = 0;*
  - Não pode ser manipulado diretamente

    ~~*x++;*~~

    ~~*x--;*~~

    ~~*x=10;*~~
- A manipulação tem que ser feita através de duas primitivas: *down* e *up*
  - *down*:diminui o valor do semáforo (bloqueia se <0)

    down(&x);
  - *up*: aumenta o valor do semáforo (desbloqueia se <0)

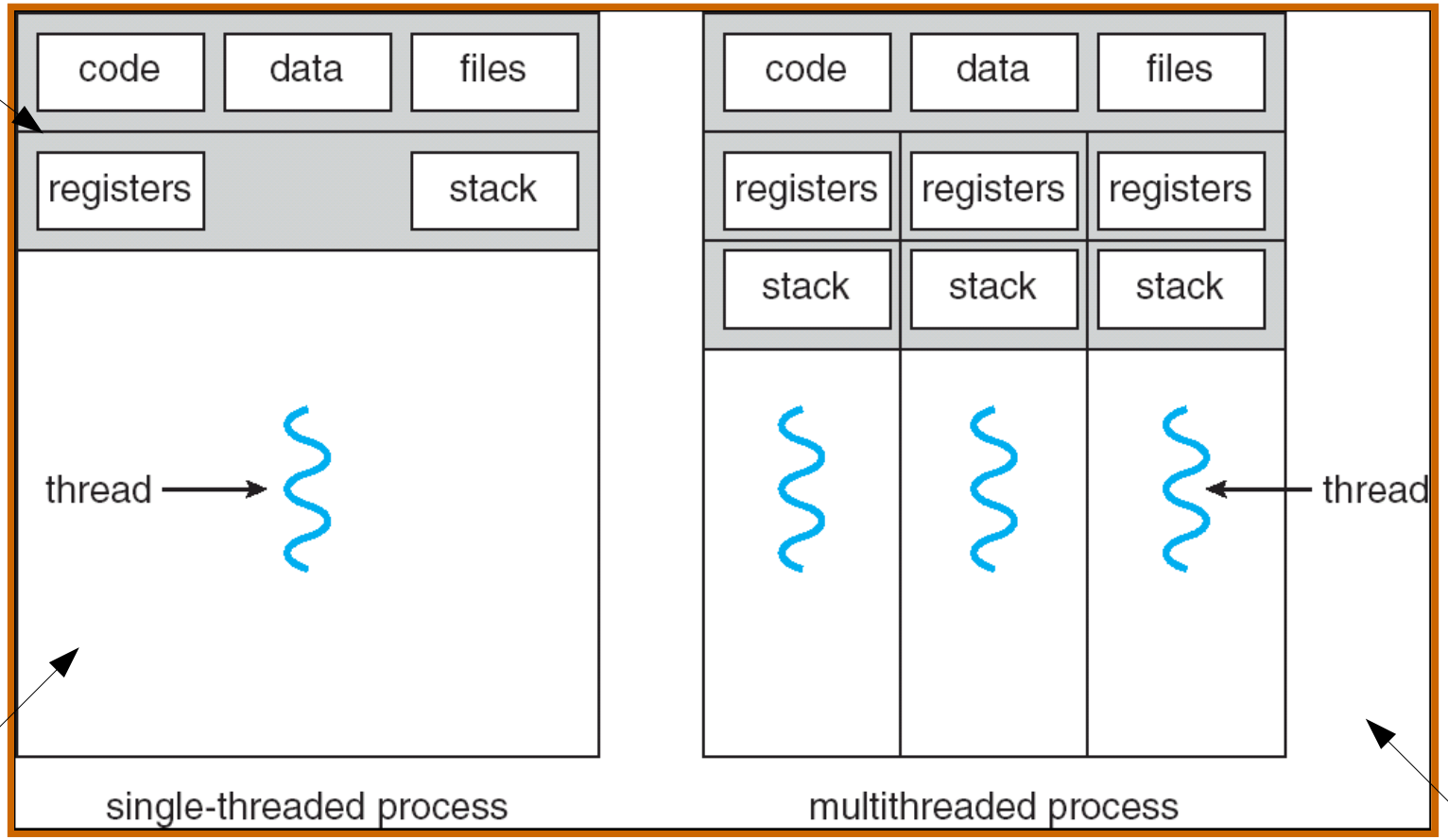
    up(&x);
- Nomes alternativos:

down	=	*wait*	=	*P*
up	=	*signal*	=	*V*

Threads (fluxos de execução)

Dados que devem ser mantidos para cada *thread*

de usuário e de núcleo

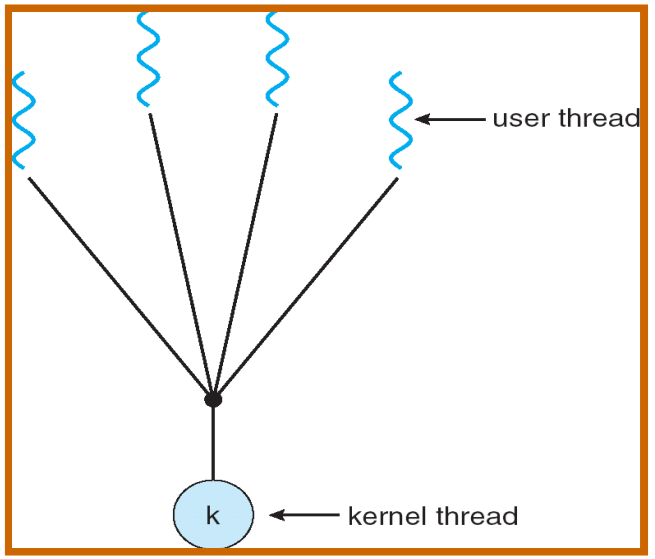


Modelo antigo: uma *thread* por processo

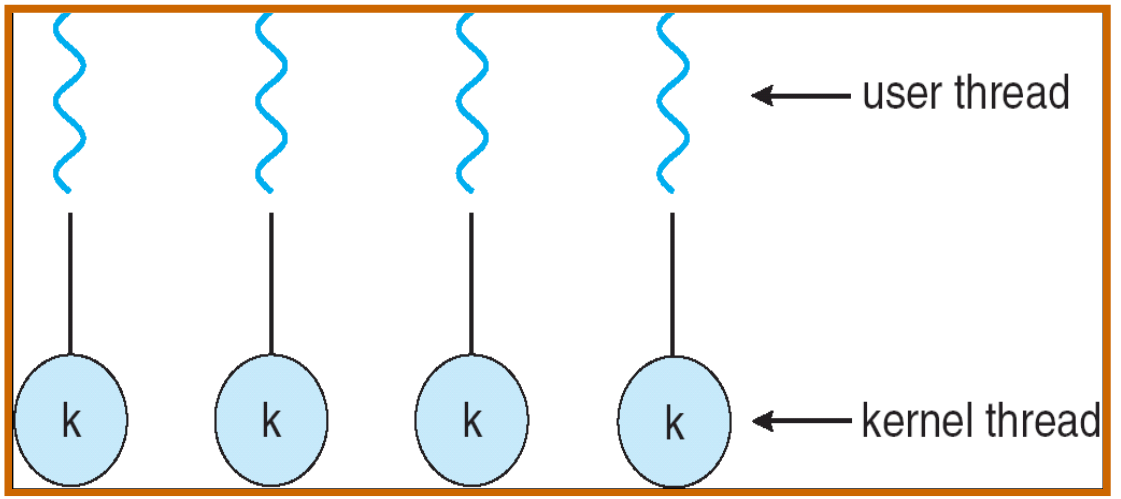
Modelo novo: várias *threads* por processo

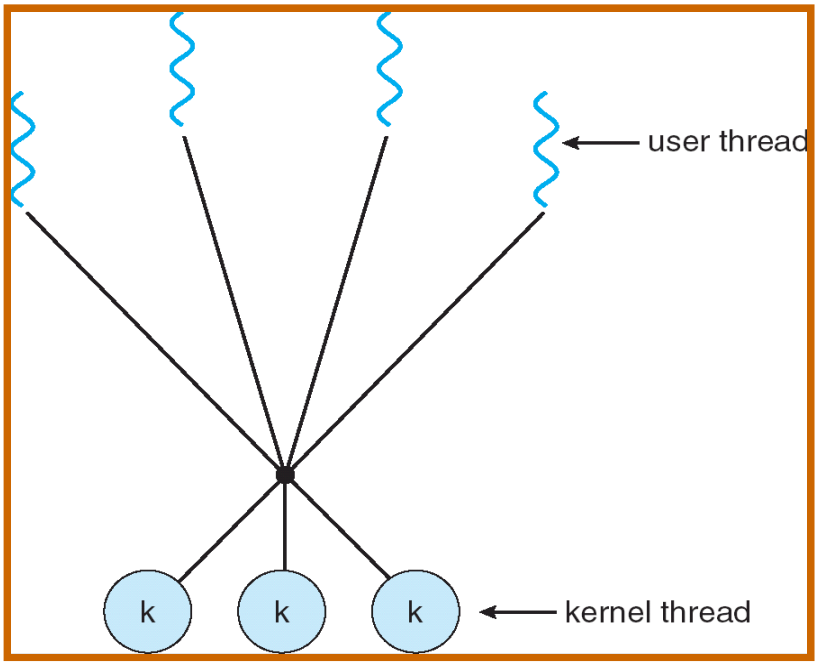
Modelos *multithreading*

Muitos para um



um para um



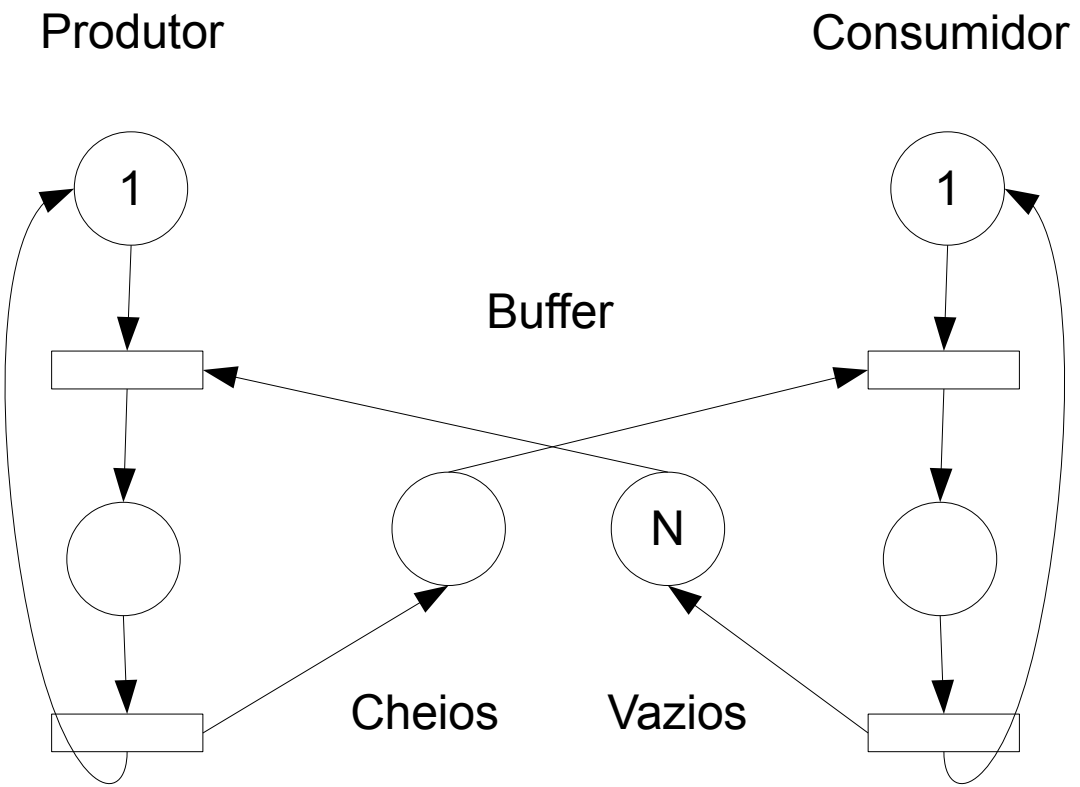


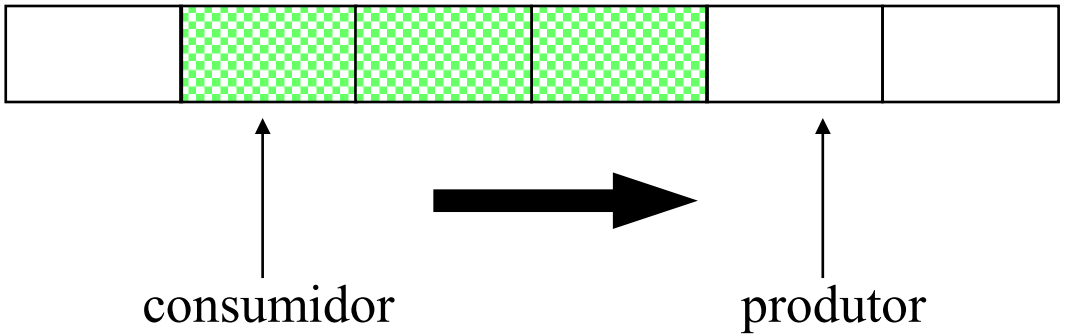
muitos para muitos

Problemas clássicos de sincronização

Produtor-consumidor (buffer limitado)

```
Semaphore mutex=1;
Semaphore vazios=N;
Semaphore cheios=0;

void produtor (void)
{
      while(true){
            // produz item
            down(&vazios);
            down(&mutex);
            // adiciona item
            up(&mutex);
            up(&cheios);
      }
}

void consumidor (void)
{
      while(true){
            down(&cheios);
            down(&mutex);
            // remove item
            up(&mutex);
            up(&vazios);
            // consome item
      }
}
```





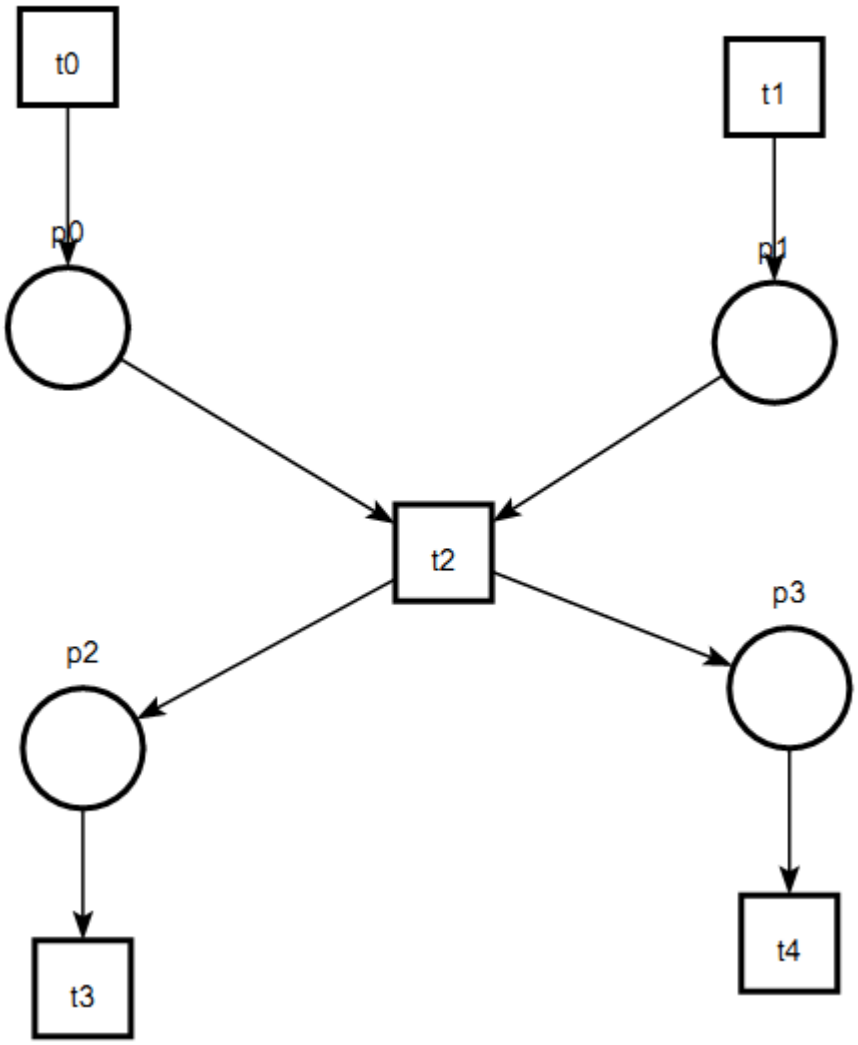
Problemas clássicos de sincronização

Rendez-vous

Considere o código abaixo e suponha que se deseje que a Instrução *a1* seja executada antes de *b2* e *b1* antes de *a2*

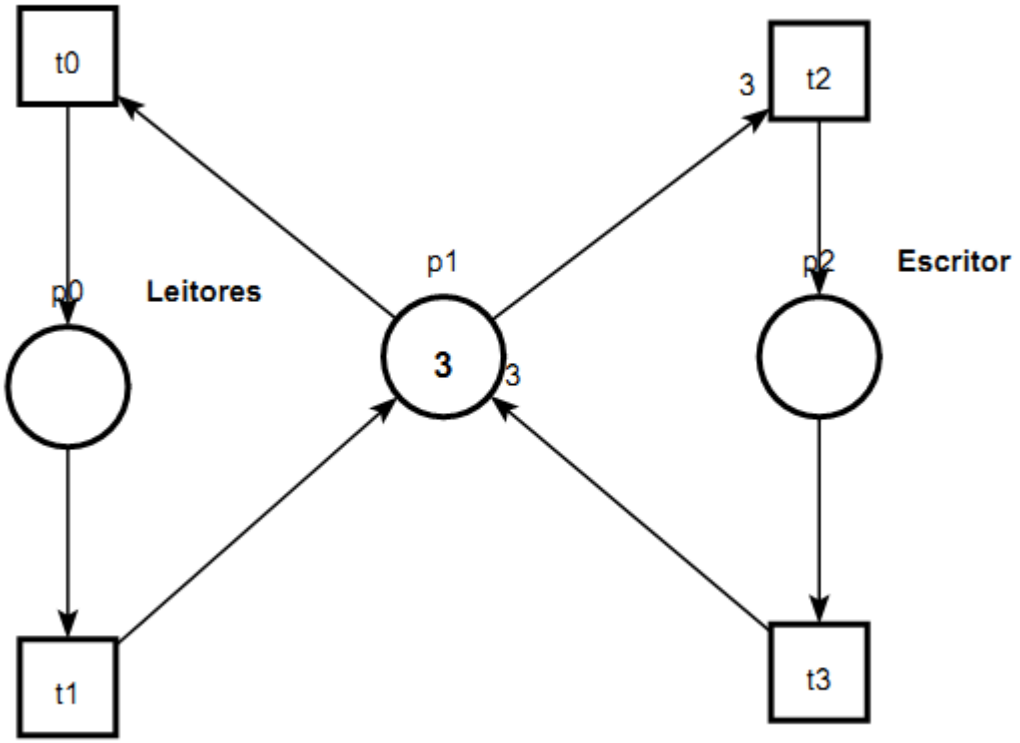
```
void thread1 (void)
{
      // a1
      // rendez-vous
      // a2
}

Void thread2 (void)
{
      // b1
      // rendez-vous
      // b2
}
```

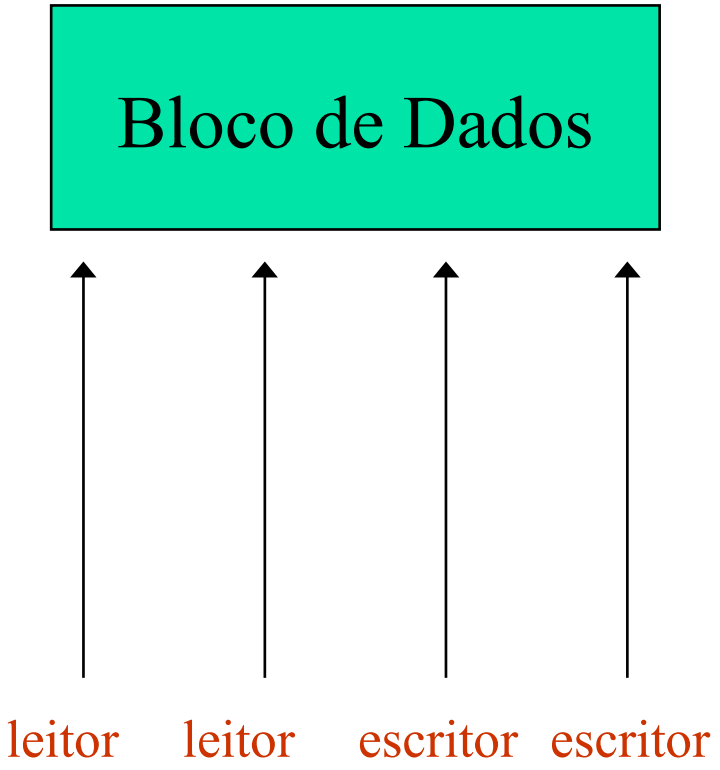
```
Semaphore x=0;
Semaphore y=0;

void thread1 (void)
{
      // a1
      up(&x);
      down(&y);
      // a2
}

void thread2 (void)
{
      // b1
      up(&y);
      down(&x);
      // b2
}
```



Problemas clássicos de sincronização

Leitores e Escritores



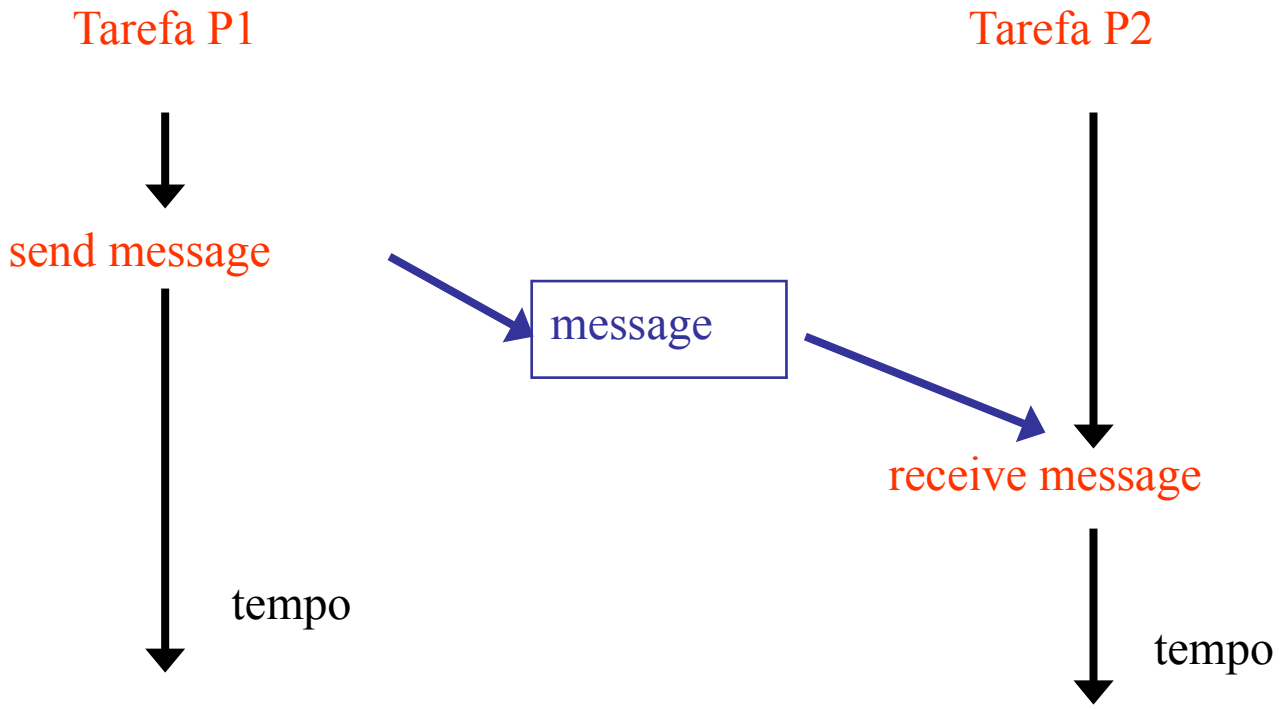


Sincronização e comunicação baseadas em mensagens

Assíncrona (sem espera)

- Requer armazenamento de mensagens (*buffer*) → *buffers* potencialmente infinitos
  - O que fazer quando o buffer estiver cheio?

void send (Task destino, Msg msg);
*void receive (Task origem, Msg *msg);*

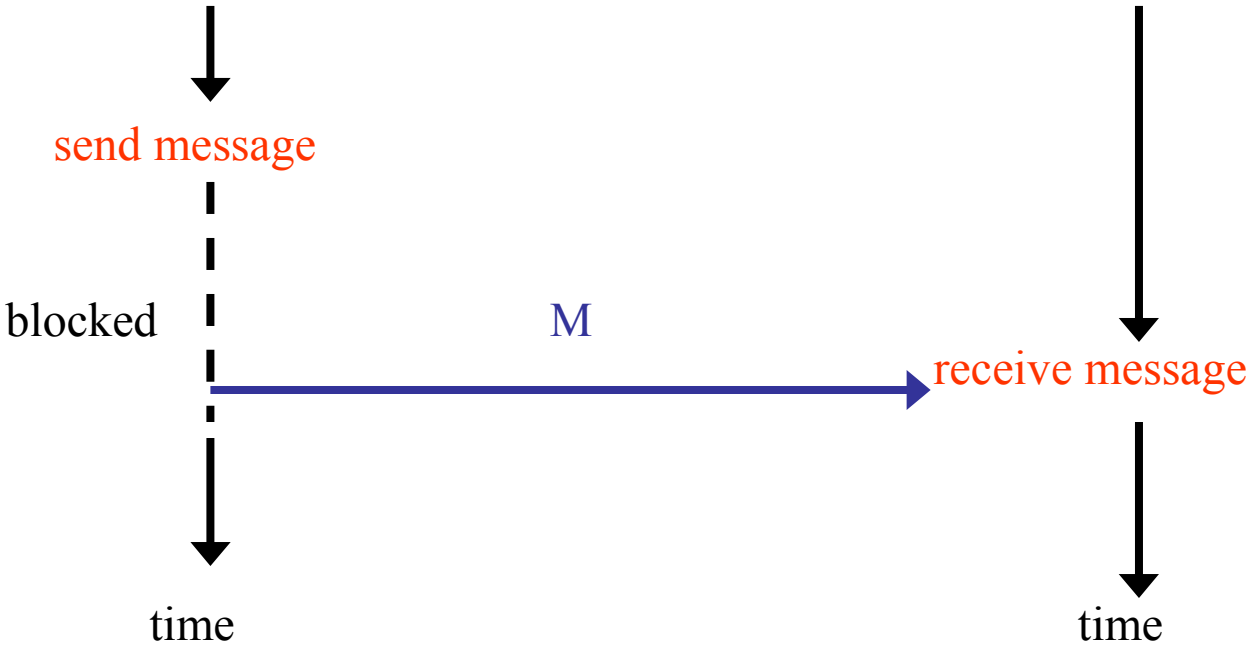


Sincronização e comunicação baseadas em mensagens

Síncrona

- Não requer *buffer*
  - Facilita a implementação
  - Conhecida como *Rendezvous*

void send (Task destino, Msg msg);
*void receive (Task origem, Msg *msg);*

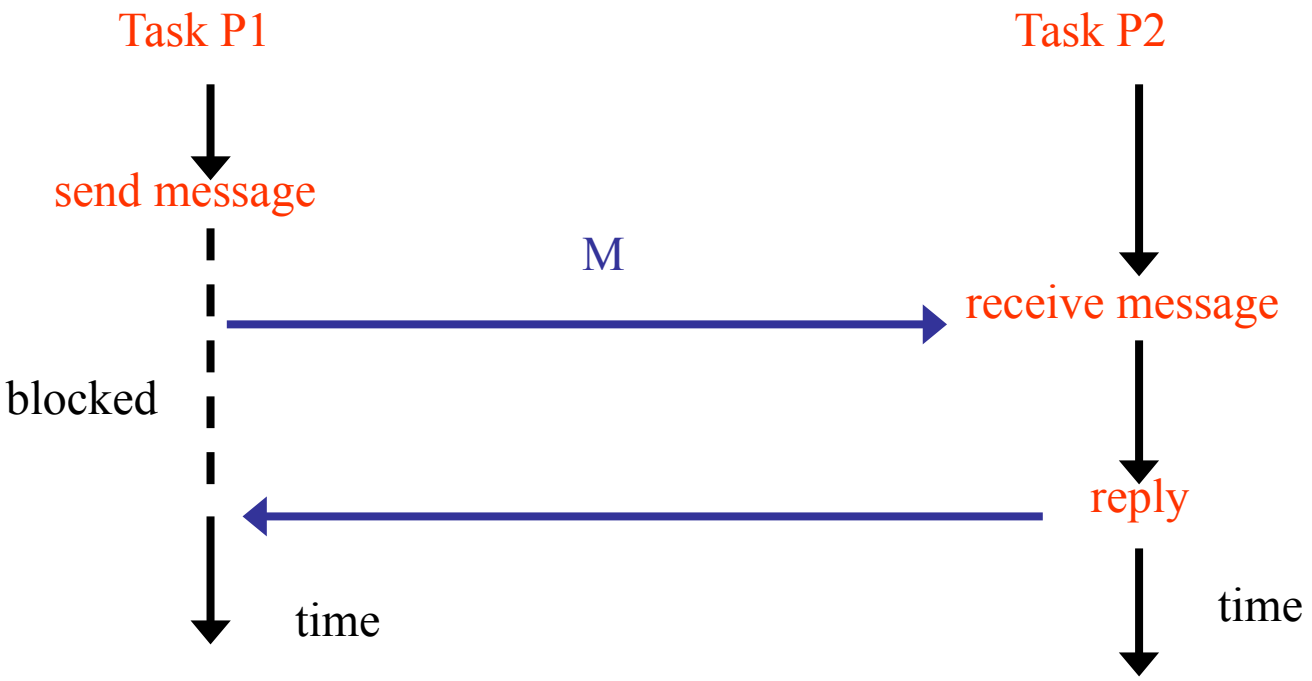


Sincronização e comunicação baseadas em mensagens

Remote invocation (e.g. Ada)

- Conhecida como rendezvous estendida

Status send (Task destino, Msg msg);
*Status receive (Task origem, Msg *msg);*



Sincronização e comunicação baseadas em mensagens

Nomeação das tarefas

- **Direta** → simplicidade

  O transmissor nomeia o receptor explicitamente

  void send(Task t, Msg m);

- **Indireta** → auxilia a decomposição do *software* (a *mailbox* pode ser vista como uma interface entre as várias partes do programa)

  o transmissor nomeia uma entidade intermediária (canal, *mailbox*, *link* ou *pipe*)

  void send(MailBox mb, Msg m);

Sincronização e comunicação baseadas em mensagens

Nomeação das tarefas

- **Simética**

  Ambos, TX e RX, nomeiam um ao outro (ou à caixa de correio).

  void send(Task t, Msg m);
  void receive(Task t, Msg m);

  void send(Mailbox mb, Msg m);
  *void receive(Mailbox mb, Msg *m);*

- **Assimétrica**

  o receptor não nomeia a origem e recebe mensagens de todas as tarefas (ou de todas as caixas de correio)

*void receive (Msg *m);*

Bibliografia

[1] *Real-Time Systems and Programming Languages*. Burns A., Wellings A. 2nd edition

[2] Análise de Sistemas Operacionais de Tempo Real Para Applicações de Robótica e Automação. Aroca R. V. Dissertação de Mestrado.

[3] *Operating System Concepts*. Silberschatz, Galvin, Gagne. 8th edition

[4] Sistemas Operacionais Modernos. Tanenbaum 2a edição