

# Processos

**Processo** → Programa em execução

Um processo fornece os subsídios necessários para o gerenciamento de tarefas concorrentes, mas não necessariamente paralelas.

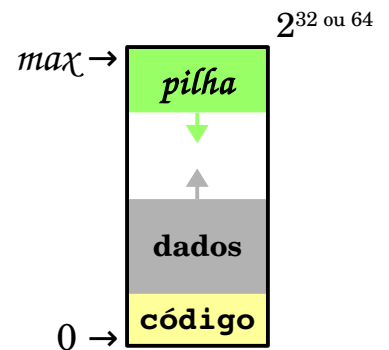
↓  
Competem pelo direito de uso  
da CPU, memória, ...

↓  
Atuam ao mesmo  
tempo!

\* Ligado a um espaço de endereçamento

\* Tabela de processos do SO (recursos alocados)

- Process ID (**PID**)
- User ID (**UID**); Group ID (**GID**)
- Arquivos abertos
- Processos relacionados (pai e filhos)
- Signals (interrupções recebidas: alarmes, erros, etc.)
- Contabilidade e prioridade de uso da CPU
- Diretório de trabalho
- Threads alocadas
  - Estado atual de execução de cada thread (new, ready, running...)
  - Estado atual dos registradores
  - Segmento de pilha
  - Parâmetros de escalonamento (prioridade)
- Status de saída (exit code)



“O processo é a unidade de gerenciamento de recursos”

**UNIXes** → Um registro de entrada na tabela de processos é mantido mesmo após o término deste, pelo menos até que o processo pai requisiite o status de saída do processo que terminou: (**zombie process**).

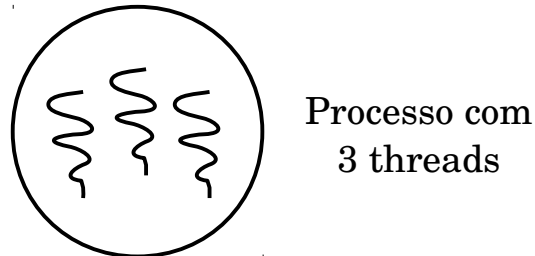
Processos em primeiro plano: **Apps do usuário**

Processos em segundo plano (*Background*): **Serviços (daemons)**

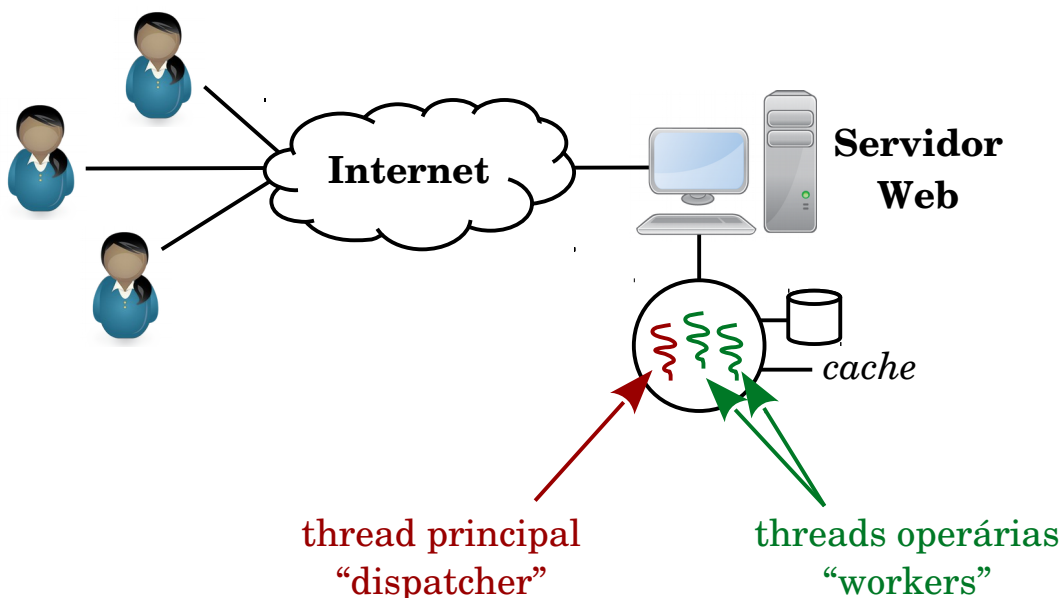
Um processo não interfere no espaço de endereçamento de outro processo: mecanismo de proteção por hardware, porém controlado pelo SO.

# Threads

**Thread** → Linha de execução independente, porém confinada ao espaço de endereçamento de um processo.



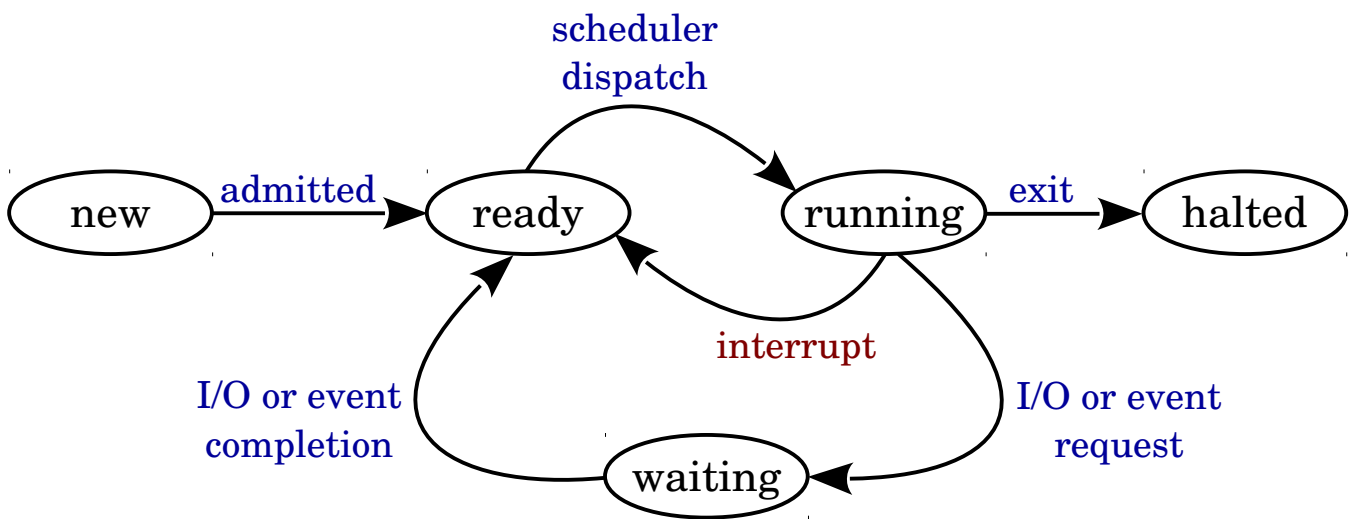
- Threads são as entidades escalonadas para execução na CPU.
- Atuam como “sub-processos” que compartilham de um mesmo espaço de endereçamento.
- Oferecem “paralelismo” mesmo com chamadas bloqueantes, mas também são a fonte de muitos problemas: concorrência / consistência.



A técnica garante um maior número de requisições por segundo  
 Maior vazão → *throughput*

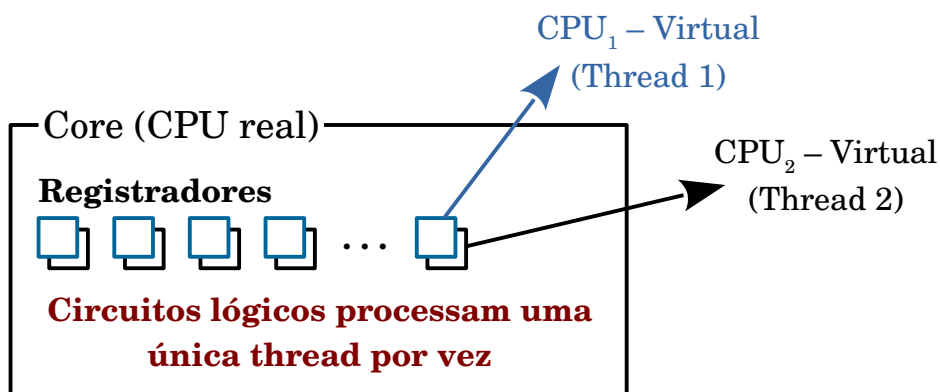
Pense em como seria com uma única thread!

## Estados de uma thread



SO modernos oferecem multitarefa **preemptiva**... Impõe restrições de tempo de execução para cada thread, tipicamente 20..50ms (quantum da CPU; timeslice).

## Multithreading vs Hyperthreading

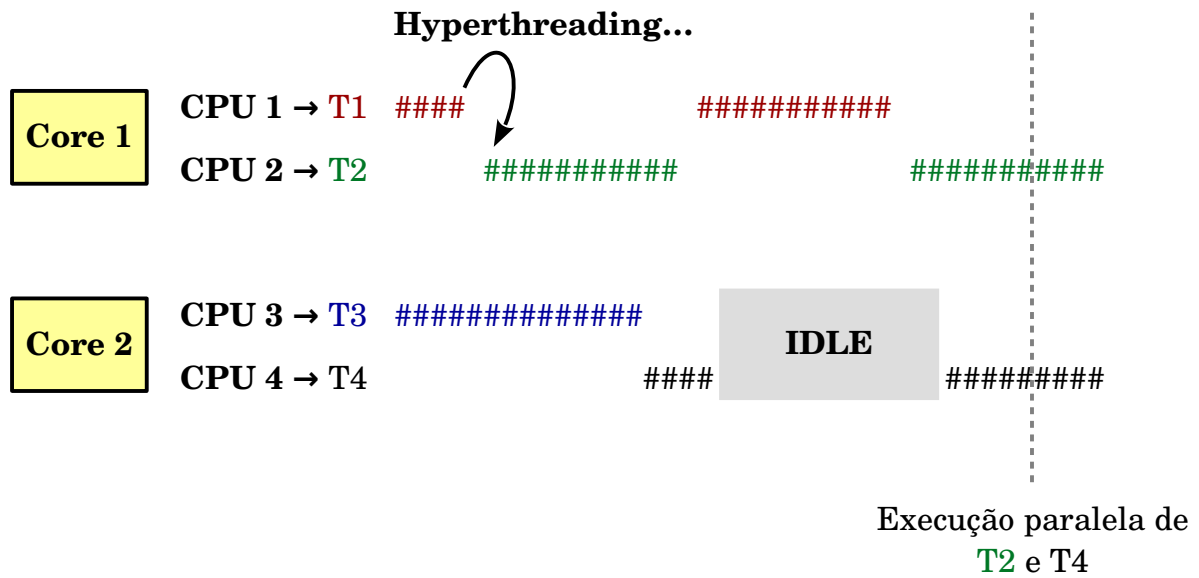


Hyperthreading → Troca rápida de contexto entre threads 1 e 2...

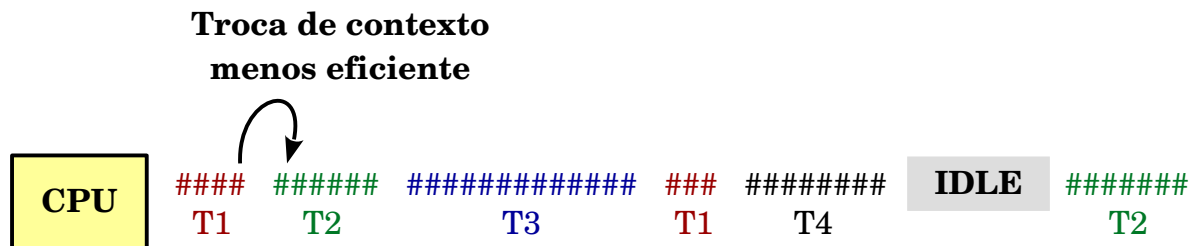
O SO “enxerga” duas CPUs por core. Cabe ao SO escalonar tarefas da melhor forma possível.

Oracle SPARC T4 → Processador Octa-core / Octa-thread (64 CPUs)

## Situação 1: Dual-Core / Dual-Thread



## Situação 2: Sistema monoprocessado



A execução concorrente existe mesmo com uma única CPU  
“Ilusão de paralelismo”

## Scheduler → Escalonador

**Problema:** Mais de uma thread pronta para execução e uma única CPU disponível.

T1:   $t = 22^{+CPU}$

T2:   $t = 27^{+I/O}$

Qual thread executar?

## Escalonamento sequencial...


T1 .. T2 ou T2 .. T1

$$t = 22 + 27 = 27 + 22 = 49$$

## Escalonamento T1 prioritária...

T1:   $t = 22^{+CPU}$

T2:   $t = 27^{+I/O}$

  $t = 39$

## Escalonamento T2 prioritária...

T1:   $t = 22^{+CPU}$

T2:   $t = 27^{+I/O}$

  $t = 28$

**Conclusão:** Processos do tipo I/O-intensive que recebem respostas de requisições de I/O devem entrar em execução o mais rapidamente possível → melhor uso da CPU em conjunto com o hardware de I/O.

# Algoritmos de escalonamento...

## 1. First-Come First-Served (não-preemptivo)

- Processos são alocados em fila para execução.
- Quando bloqueado (I/O), o processo é movido para o fim da fila e outro processo “ganha” o direito de uso da CPU.

## 2. Shortest-Job First (não-preemptivo)

- Tarefas com menor tempo de processamento são escalonadas primeiro.
- Assume que o tempo de execução de cada tarefa é conhecido.

## 3. Shortest Remaining Time Next (preemptivo)

- Se um processo “recém-chegado” apresenta tempo de execução menor que o tempo necessário para o processo atual ser concluído, então o novo processo “ganha” a CPU.
- Também assume que o tempo de execução de cada tarefa é conhecido.

## 4. Round-Robin (preemptivo) → Chaveamento Circular

- Os processos são alocados alternadamente e executam por uma fatia máxima de tempo (*quantum*), tipicamente 20..50ms.
  - *Quantum* muito baixo → processos são alternados mais rapidamente, mas reduz eficiência da CPU porque a própria troca de contexto consome tempo.
  - *Quantum* muito alto → prejudica tempo de resposta de requisições curtas.

SO modernos tipicamente adotam o esquema Round-Robin juntamente com um controle de prioridade.

## Curiosidade!

### Windows 7

Painel de Controle → Configurações avançadas do sistema  
“Ajuste do Escalonador (Scheduler)”

Priorizar...

(o) Programas

( ) Processos em segundo plano (Serviços)

### Linux

Usuário voluntariamente reduz prioridade do processo...

`nice -n <prioridade> comando`  
-20..20

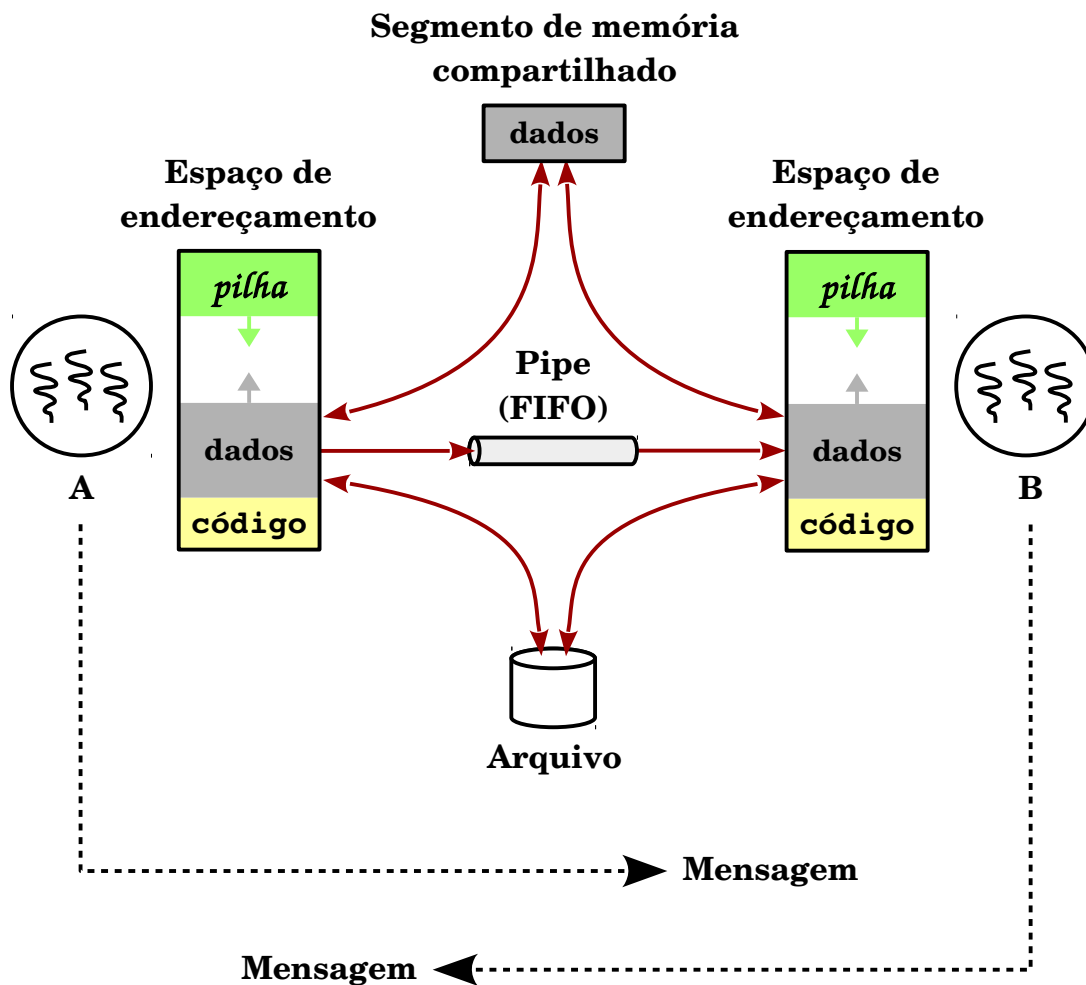
`renice <prioridade> <nr.processo>`



# IPC – Interprocess Communication

Como um processo pode se comunicar com outro?

1. Área compartilhada: segmento de memória, arquivo, pipe.
2. Mensagens (sockets) → Sistemas Distribuídos.



**Atenção!** O desempenho obtido com o uso de memória compartilhada pode degradar em razão da consistência de cache mantida em hardware quando vários cores atuam sobre o compartilhamento.

## IPC → Área compartilhada

**Problema:** Condição de Corrida (Race Condition)

Ocorre quando dois ou mais processos acessam a área compartilhada ao mesmo tempo e pelo menos um deles efetua uma operação de escrita. O resultado final dependerá da ordem em que as ações forem executadas, podendo até mesmo ser inconsistente.

Pipe → Imune

**Solução:** Exclusão mútua

Garantir que apenas um processo tenha acesso à área compartilhada por vez.

O trecho de código que deve ter acesso exclusivo à área compartilhada de modo a manipulá-la de forma consistente é denominado região crítica.



Limita o paralelismo

**Atenção!** Não confundir região crítica (uma parte do código) com área compartilhada (uma parte da memória ou arquivo).

## Como obter exclusão mútua?

Desabilitando interrupções: requer **modo privilegiado** e funciona para apenas uma CPU.


ou...

Mantendo processos concorrentes em estado de espera, porém constantemente verificando pela possibilidade de entrar na região crítica (Trava Giratória “Spin Lock” → Espera Ocupada “Busy Waiting”).

### Técnica 1: Variáveis de trava (lock)

**Busy Waiting...** **Spin Lock**  
**“Trava Giratória”**


```
while (n == 1) ; // processos na região crítica
n = 1;
```




Não funcionam pois também estão sujeitas à condição de corrida.

### Técnica 2: Alternância explícita

```
// Processo: 0
while (TRUE) {
    while (turn != 0) ;
    critical_region();
    turn = 1;
    noncritical_region();
}
```



```
// Processo: 1
while (TRUE) {
    while (turn != 1) ;
    critical_region();
    turn = 0;
    noncritical_region();
}
```



Um processo por vez... A técnica não é adequada quando os processos envolvidos apresentam tempo de processamento muito divergentes.

## Técnica 3: Peterson


Solução em software que combina alternância e trava e permite a continuidade de processos com tempos divergentes.

Algoritmo de Peterson para dois processos: 0 e 1

```
int turn;
int interested[2]; // inicialmente FALSE (0)

void enter_region(int process) {
    interested[process] = TRUE;
    int other = 1 - process;
    turn = other;
    while (interested[other] == TRUE && turn == other) ;
}

void leave_region(int process) {
    interested[process] = FALSE;
}
```



## Técnica 4: Hardware (TSL, XCHG)

Primitivas de sincronização de baixo nível mais eficientes. Bloqueiam o barramento de memória durante a execução (operações atômicas).

### CPUs Intel x86

enter\_region:

```
TSL REG, #lock
CMP REG, 0
JNE enter_region
```

RET

enter\_region:

```
MOV REG, 1
XCHG REG, #lock
CMP REG, 0
JNE enter_region
```

RET

leave\_region:

```
MOV #lock, 0
RET
```

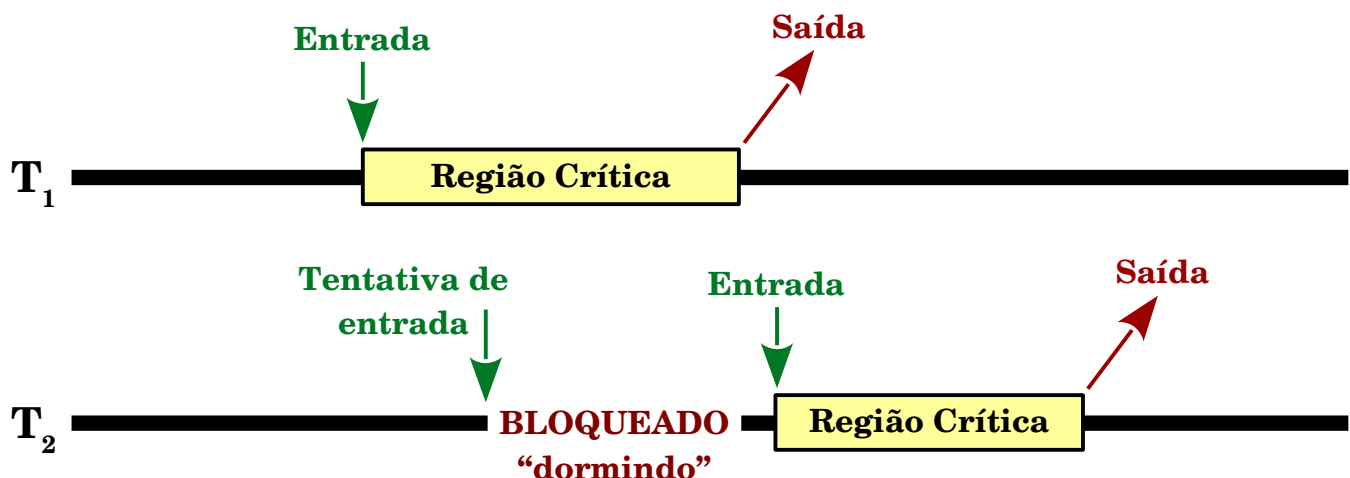
leave\_region:

```
MOV #lock, 0
RET
```

**Problema:** A trava giratória desencadeia uma espera ocupada.

**Solução:** Bloquear ao invés de permanecer em spin lock.

Comportamento desejado...



**Bloqueios** → Colocando uma thread para dormir (sleep) e acordando no momento adequado (wakeup).

Estudo de caso: produtor-consumidor

```

01 #define N 100
02 int count = 0;
03
04 void producer() {
05     int item;
06     while (TRUE) {
07         item = produce_item();
08         if (count == N) sleep();
09         insert_item(item);
10         count++;
11         if (count == 1) wakeup(consumer);
12     }
13 }
14
15 void consumer() {
16     int item;
17     while (TRUE) {
18         if (count == 0) sleep();
19         item = remove_item();
20         count--;
21         if (count == N - 1) wakeup(producer);
22         consume_item(item);
23     }
24 }

```

**Scheduler...**

15, 16, 17, 18 (**no-sleep**), 04 .. 11 (sinal de “wakeup” perdido), 18 (**sleep**)

Resultado: Produtor e consumidor irão dormir para sempre

**Como resolver o problema do sinal perdido?**

**Resposta:** Registrando o número de sinais emitidos → Semáforos

## Implementação típica...

Sim... operações sobre semáforos são rápidas!

```
MOV #lock, 0
CALL thread_sleep
JMP down
```

## Estudo de caso: produtor-consumidor com semáforos

```
#define N 100

typedef int semaphore;

semaphore used = 0;    } utilizados para
semaphore free = N;    } sincronização

void producer() {
    int item;
    while (TRUE) {
        item = produce_item();
        down(&free);
        insert_item(item);
        up(&used);
    }
}

void consumer() {
    int item;
    while (TRUE) {
        down(&used);
        item = remove_item();
        up(&free);
        consume_item(item);
    }
}
```

**Problema:** A inserção e remoção de itens de dados no buffer ainda sofre do problema da condição de corrida (Race Condition).

**Solução:** Semáforos de apenas dois estados (semáforos binários) podem ser empregados para tratar as operações de inserção e remoção como regiões críticas...

**mutex** → semáforo binário utilizado para se obter exclusão mútua



## Estudo de caso: produtor-consumidor com semáforos e mutex

```
#define N 100

typedef int semaphore;

semaphore used = 0;
semaphore free = N;
semaphore mutex = 1; → utilizado para se obter exclusão mútua

void producer() {
    ...
}

void consumer() {
    ...
}

void insert_item(int value) {
    down(&mutex);
    ...
    up(&mutex);
}

int remove_item() {
    down(&mutex);
    int value;
    ...
    up(&mutex);
    return value;
}
```

## Implementação típica de um mutex...

mutex\_lock:

```
MOV REG, 1
XCHG REG, #mutex
CMP REG, 0
JZE ok
CALL thread_sleep
JMP mutex_lock
```

mutex\_unlock:

```
MOV #mutex, 0
CALL thread_wakeup
RET
```

ok:

```
RET
```

## POSIX → Threads e primitivas de sincronização

pthread\_create

pthread\_exit

pthread\_join

pthread\_yield

pthread\_attr\_init

pthread\_attr\_destroy

pthread\_mutex\_init

pthread\_mutex\_destroy

pthread\_mutex\_lock

pthread\_mutex\_trylock

pthread\_mutex\_unlock



**Controle de acesso sobre a  
região crítica do código  
“Exclusão mútua”**

pthread\_cond\_init

pthread\_cond\_destroy

pthread\_cond\_wait

pthread\_cond\_signal

pthread\_cond\_broadcast



**Espera e envio de sinal para  
threads bloqueadas**

## Estudo de caso: produtor-consumidor com threads (POSIX)

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

#define MAX 100000

pthread_mutex_t mutex_buffer;
pthread_cond_t thread_producer, thread_consumer;

int buffer = 0;

void *producer(void *ptr) {
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&mutex_buffer);
        while (buffer != 0) {
            pthread_cond_wait(&thread_producer, &mutex_buffer);
        }
        buffer = i;
        pthread_cond_signal(&thread_consumer);
        pthread_mutex_unlock(&mutex_buffer);
    }
    pthread_exit(0);
}

void *consumer(void *ptr) {
    int i;
    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&mutex_buffer);
        while (buffer == 0) {
            pthread_cond_wait(&thread_consumer, &mutex_buffer);
        }
        printf("%i\n", buffer);
        buffer = 0;
        pthread_cond_signal(&thread_producer);
        pthread_mutex_unlock(&mutex_buffer);
    }
    pthread_exit(0);
}
```

```
int main(int argc, char **argv) {
    pthread_t p, c;
    pthread_mutex_init(&mutex_buffer, 0);
    pthread_cond_init(&thread_producer, 0);
    pthread_cond_init(&thread_consumer, 0);
    pthread_create(&p, 0, producer, 0);
    pthread_create(&c, 0, consumer, 0);
    pthread_join(p, 0);
    pthread_join(c, 0);
    pthread_cond_destroy(&thread_consumer);
    pthread_cond_destroy(&thread_producer);
    pthread_mutex_destroy(&mutex_buffer);
    return 0;
}
```

## Considerações...

Spin locks são adequados quando o tempo de espera é curto. Mas quando há muita contenção (muitas tentativas de aquisição de trava ocorrendo), é melhor bloquear as threads sendo contidas e deixar o kernel do SO desbloqueá-las a medida que a trava é liberada.

Baixa contenção → Spin Locks

Alta contenção → Bloqueios

## Mecanismos de sincronização...

Baixo nível → semaphores (operações up e down)  
mutexes (operações lock e unlock)  
condition variables (operações wait e signal)

Alto nível → monitor

Um “monitor” é um pacote (módulo) com uma coleção de métodos (funções) que garante a seguinte propriedade: somente uma thread por vez tem acesso ao monitor.

Monitores são uma facilidade oferecida pela linguagem de programação ou por uma biblioteca para se definir regiões críticas. Cabe ao compilador ou a biblioteca garantir a exclusão mútua.

Estruturas de dados compartilhadas devem ficar encapsuladas no monitor. A exclusão mútua é obtida transformando-se cada região crítica em uma sub-rotina do monitor.

## IPC → Mensagens

Necessário quando os processos executam em máquinas distintas: sistemas distribuídos.

Primitivas operacionais: send / receive

### **Problemas...**

Todos relacionados a comunicação em redes:

- Perda de pacotes
- Duplicidade
- Reordenação
- Autenticidade
- Integridade