

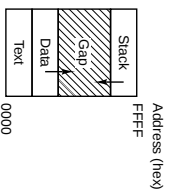
Processos e Threads 1

Objetivos

- Processo
 - Espaço de endereçamento
- Pthreads
 - Operações create e join
 - Envio e recepção de valores para threads

Processo

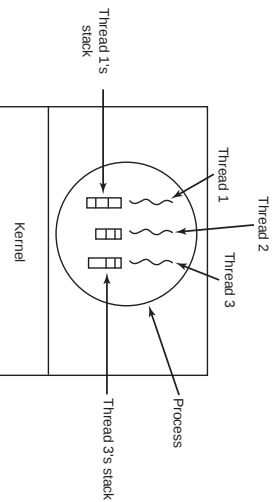
- Programa em execução
- Espaço de endereçamento



Tanenbaum: Figura 1.20

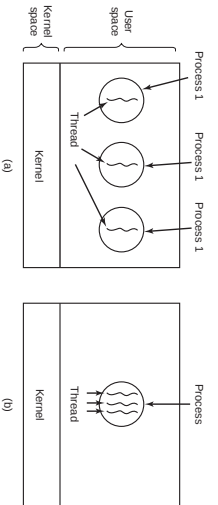
- Veja os códigos: `ender.c` e `ender-malloc.c`

Pilhas independentes



Tanenbaum: Figura 2.8

Modelo de threads



Tanenbaum: Figura 2.6

Como trabalhar com threads

Veja os comandos:

- `pthread_create`
- `pthread_join`
- `pthread_exit`

Para mais informações: `man <comando>`

Como criar uma thread

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

Veja o código: `create0.c`

Como esperar por uma thread

```
int pthread_join(pthread_t thr,  
void **thread_return);
```

Veja os códigos: `join0.c`, `join1.c`, `join2.c`, `join3.c` e `join4.c`

Como passar argumentos para uma thread

- Exemplo: cada thread pode precisar de um identificador único.
- Veja os códigos: `createl.c`, `create2.c`, `create3.c` e `create4.c`

MC514–Sistemas Operacionais: Teoria e Prática

1s2008

Processos e Threads 2

Objetivos

- Pthreads
 - Revisão `create` e `join`
 - Operação `exit`
- Pilha de execução
- Primeiros problemas de condição de corrida

Create e Join

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

int pthread_join(pthread_t thr,
                 void **thread_return);
```

Veja o código: `create-join.c`

Como encerrar a execução de uma thread

- Comando `return` na função principal da thread (passada como parâmetro em `pthread_create`)
- Análogo ao comando `return` na função `main()`

Veja os códigos: `return0.c`, `return1.c`, `pthread_return.c`

Como encerrar a execução de uma thread

- `void pthread_exit(void *retval);`
- Análogo ao comando `exit(status);`

Veja os códigos: `exit0.c`, `exit1.c` e `pthread_exit0.c`

Pilha de execução:

- Espaço para valor de retorno da função
- Argumentos
- Endereço de retorno
- Registradores
- Variáveis locais

Veja o código: `pilha.c`

Exemplo de endereços

0x804971c	:
global	:
0x80483ed	:
main	:
0x80483a4	f

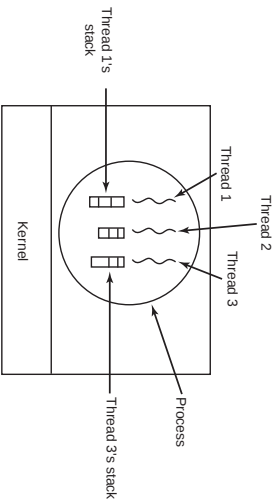
Exemplo de endereços

0xbfb0f1f0	local_main
...	...
0xbfb0f1d0	param_f
...	...
0xbfb0f1c4	v [1]
0xbfb0f1c0	v [0]
...	...

É muito fácil corromper a pilha

- Basta fazer acesso a posições não alocadas de um vetor
- Veja os códigos: corrompe_pilha.c e corrompe_pilha1.c

Pilhas independentes



Veja o código: pilhas.c

Uma thread pode corromper a pilha de outra thread

- Pilhas são independentes, mas não protegidas
- Veja o código: corrompe_thread.c

Acesso a recursos compartilhados

- Estudo de caso:
volatile int s; /* Variável compartilhada */

/* Cada thread tentar executar os seguintes comandos sem interferência. */

s = thr_id;
printf ("Thr %d: %d", thr_id, s);

Condição de disputa

Saída esperada

```
int s; /* Variável compartilhada */  
  
Thread 0  
(i) s = 0;  
(ii) print ("Thr 0: ", s);  
  
Thread 1  
(iii) s = 1;  
(iv) print ("Thr 1: ", s);
```

Saída: Thr 0: 0
Thr 1: 1

Condição de disputa

Saída esperada II

```
int s; /* Variável compartilhada */  
  
Thread 0  
(iii) s = 0;  
(iv) print ("Thr 0: ", s);  
  
Thread 1  
(i) s = 1;  
(ii) print ("Thr 1: ", s);
```

Saída: Thr 1: 1
Thr 0: 0

Condição de disputa

Saída inesperada

```
int s = 0; /* Variável compartilhada */  
  
Thread 0  
(i) s = 0;  
(iii) print ("Thr 0: ", s);  
  
Thread 1  
(ii) s = 1;  
(iv) print ("Thr 1: ", s);
```

Saída: Thr 0: 1
Thr 1: 1

Veja o código: inesperada.c

Processos e Threads 3

Objetivos

- Primeiros problemas de condição de corrida
- Exclusão mútua
- Primeiras tentativas de algoritmos
- Algoritmo de Dekker
- Algoritmo do desempate

Acesso a recursos compartilhados

- Estudo de caso:

```
volatile int s; /* Variável compartilhada */

/* Cada thread tentar executar os seguintes
   comandos sem interferência. */

s = thr_id;
printf ("Thr %d: %d", thr_id, s);
```

Condição de disputa Saída esperada

```
volatile int s; /* Variável compartilhada */

Thread 0                Thread 1
(i) s = 0;                (iii) s = 1;
(ii) print ("Thr 0: ", s); (iv) print ("Thr 1: ", s);

Saída: Thr 0: 0
      Thr 1: 1
```

Condição de disputa Saída esperada II

```
volatile int s; /* Variável compartilhada */

Thread 0                Thread 1
(iii) s = 0;              (i) s = 1;
(iv) print ("Thr 0: ", s); (ii) print ("Thr 1: ", s);

Saída: Thr 1: 1
      Thr 0: 0
```

Condição de disputa Saída inesperada

```
volatile int s; /* Variável compartilhada */

Thread 0                Thread 1
(i) s = 0;                (ii) s = 1;
(iii) print ("Thr 0: ", s); (iv) print ("Thr 1: ", s);

Saída: Thr 0: 1
      Thr 1: 1
```

Veja o código: inesperada.c

Escalonamento de threads

- A execução de uma thread pode ser interrompida a qualquer momento.
- Veja o código preemptivo.c

Exclusão mútua

- Acesso controlado a recursos compartilhados
- Estudo de caso:

```
volatile int s; /* Variável compartilhada */
while (1) {
    /* Região não crítica */
    /* Protocolo de entrada */
    /* Região crítica */
    s = thr_id;
    printf ("Thr %d: %d", thr_id, s);
    /* Protocolo de saída */
}
```

Tentando implementar um lock

- Lock = variável compartilhada com o seguinte significado:
 - lock == 0 $\Rightarrow$ região crítica está livre
 - lock != 0 $\Rightarrow$ região crítica está ocupada
- Protocolo de entrada na região crítica

```
while (lock != 0);
```
- Protocolo de saída da região crítica

```
lock = 0;
```

Tentando implementar um lock

```
volatile int s = 0, lock = 0;

Thread 0          Thread 1
while (lock == 1); while (lock == 1);
lock = 1;         lock = 1;
s = 0;            s = 1;
print ("Thr 0:" , s); print ("Thr 1:" , s);
lock = 0;         lock = 0;
```

- Veja o código: tentativa-lock.c

Solução em hardware

```
entra_RC:
    TSL RX, lock
    CMP RX, #0
    JNE entra_RC
    RET

deixa_RC:
    MOV lock, \#0
    RET

• Não vale para a aula de hoje :-)
```

Abordagem da Alternância

```
int s = 0;
int vez = 1; /* Primeiro a thread 1 */

Thread 0          Thread 1
while (true)      while (true)
while (vez != 0); while (vez != 1);
s = 0;            s = 1;
print ("Thr 0:" , s); print ("Thr 1:" , s);
vez = 1;         vez = 0;
```

- Veja o código: alternancia.c

Abordagem da Alternância

N threads

```
Thread i:
while (true)
while (vez != i);
s = i;
print ("Thr ", i, " : ", s);
vez = (i + 1) % N;

• Veja o código: alternanciaN.c
```

Limitações da Alternância

- Uma thread fora da RC pode impedir outra thread de entrar na RC
- Se uma thread interromper o ciclo a outra não poderá mais entrar na RC

Vetor de Interesse

```
int s = 0;
int interesse[2] = {false, false};

Thread 0          Thread 1
while (true)      while (true)
interesse[0] = true; interesse[1] = true;
while (interesse[1]); while (interesse[0]);
s = 0;            s = 1;
print("Thr 0:" , s); print("Thr 1:" , s);
interesse[0] = false; interesse[1] = false;

• Veja o código: interesse.c
```

Algoritmos de Exclusão Mútua

- Devemos garantir:
 - exclusão mútua
 - ausência de deadlock
 - progresso (uma thread que não esteja interessada na região crítica não pode impedir outra thread de entrar na região crítica)
 - Como escrever provas formais?
- Fonte: Principles of Concurrent and Distributed Programming - M. Ben-Ari

Vetor de Interesse

```
int i[2] = {false, false};
```

Thread 0

```
while (true)
```

```
    a0: nao-critica();
```

```
    b0: i[0] = true;
```

```
    c0: while (i[1]);
```

```
    d0: critica();
```

```
    e0: i[0] = false;
```

Thread 1

```
while (true)
```

```
    a1: nao-critica();
```

```
    b1: i[1] = true;
```

```
    c1: while (i[0]);
```

```
    d1: critica();
```

```
    e1: i[1] = false;
```

- Basta apresentar um escalonamento: $a0 \ a1 \ b0 \ b1$

Prova - deadlock

Prova - exclusão mútua

$$i[0] \equiv at(a0) \vee at(d0) \vee at(e0)$$

$$i[1] \equiv at(c1) \vee at(d1) \vee at(e1)$$

$$\text{Exclusão mútua} \equiv \neg(at(d0) \wedge at(d1))$$

Prova - exclusão mútua

$$i[0] \equiv at(a0) \vee at(d0) \vee at(e0)$$

- $a0$ A fórmula é inicialmente válida
- $a0 \rightarrow b0$ - não altera a fórmula
- $b0 \rightarrow c0$ - altera os dois lados da fórmula
- $c0 \rightarrow d0$, $c0 \rightarrow d0$ e $d0 \rightarrow e0$ - não alteram a validade de nenhum dos dois lados da fórmula
- $e0 \rightarrow a0$ - altera os dois lados da fórmula
- Transições na thread 1 não alteram a fórmula

Prova - exclusão mútua

$$\neg(at(d0) \wedge at(d1))$$

- A fórmula é inicialmente válida
- Considere $at(d0)$ e que a thread 1 vai fazer a transição $c1 \rightarrow d1$
- $at(d0)$ implica $i[0]$ e, portanto, a thread 1 fica presa no loop e não consegue completar a transição
- Cenários simétricos $\Rightarrow$ provas similares

Limitações do Vetor de Interesse

- O algoritmo anterior garante exclusão mútua, mas...
- se as duas threads ficarem interessadas ao mesmo tempo haverá deadlock.
- Podemos tentar sanar este problema da seguinte forma:
Se as duas threads ficarem interessadas ao mesmo tempo, elas irão baixar o interesse, esperar um pouco e tentar novamente.
- Veja o código: interesse2.c

Vetor de Interesse II

```
int s = 0;
```

```
int interesse[2] = {false, false};
```

Thread 0

```
while (true)
```

```
    interesse[0] = true;
```

```
    while (interesse[1])
```

```
        interesse[0] = false;
```

```
    sleep(1);
```

```
    interesse[0] = true;
```

```
    s = 0;
```

```
    print("Thr 0:", s);
```

```
    interesse[0] = false;
```

Thread 1

```
while (true)
```

```
    interesse[1] = true;
```

```
    while (interesse[0])
```

```
        interesse[1] = false;
```

```
    sleep(1);
```

```
    interesse[1] = true;
```

```
    s = 1;
```

```
    print("Thr 1:", s);
```

```
    interesse[1] = false;
```

Limitações do Vetor de Interesse II

- O algoritmo anterior garante exclusão mútua, mas...
- se as duas threads andarem sempre no mesmo passo haverá livelock.
- Podemos tentar outra abordagem que é:
Se as duas threads ficarem interessadas ao mesmo tempo, entrará na região crítica a thread cujo identificador estiver marcado na variável vez.
- Veja o código: interesse-vez.c

Vetor de Interesse e Alternância

```
int s = 0, vez = 0;
```

```
int interesse[2] = {false, false};
```

Thread 0

```
while (true)
```

```
    interesse[0] = true;
```

```
    if (interesse[1])
```

```
        while (vez != 0);
```

```
    s = 0;
```

```
    print("Thr 0:", s);
```

```
    vez = 1;
```

```
    interesse[0] = false;
```

Thread 1

```
while (true)
```

```
    interesse[1] = true;
```

```
    if (interesse[0])
```

```
        while (vez != 1);
```

```
    s = 1;
```

```
    print("Thr 1:", s);
```

```
    vez = 0;
```

```
    interesse[1] = false;
```

Limitações da combinação anterior

- O algoritmo anterior não garante exclusão mútua. Você consegue indicar um cenário?
- Podemos tentar melhorar o algoritmo:
Se as duas threads ficarem interessadas ao mesmo tempo, elas deverão baixar o interesse e esperar por sua vez.
- Veja o código: `quase.dekker.c`

Limitações do algoritmo anterior

- O algoritmo anterior garante exclusão mútua?
- É possível que uma thread ganhe sempre a região crítica enquanto a outra fica só esperando?
- Podemos melhorar o algoritmo:
Se as duas threads ficarem interessadas ao mesmo tempo, a thread da vez não baixa o interesse.
- Veja o código: `dekker.c`

Algoritmo do Desempe (1981)

```
int s = 0, ultimo = 0, interesse[2] = {false, false};
```

Thread 0

```
while (true)
    interesse[0] = true;
    ultimo = 0;
    while (ultimo == 0 &&
           interesse[1]);
    s = 0;
```

Thread 1

```
while (true)
    interesse[1] = true;
    ultimo = 1;
    while (ultimo == 1 &&
           interesse[0]);
    s = 1;
    print ("Thr 1:", s);
    interesse[1] = false;
```

Objetivos

- Revisão de problemas de exclusão mútua
- Tentativas de algoritmos para N threads
- Algoritmo de Dijkstra
- Algoritmos de Hyman e Peterson

Quase o algoritmo de Dekker

```
int s = 0, vez = 0, interesse[2] = {false, false};
```

Thread 0

```
while (true)
    interesse[0] = true;
    while (interesse[1])
        interesse[0] = false;
        while (vez != 0);
        interesse[0] = true;
        s = 0;
```

```
    print ("Thr 0:", s);
    vez = 1;
    interesse[0] = false;
```

Thread 1

```
while (true)
    interesse[1] = true;
    while(interesse[0])
        interesse[1] = false;
        while(vez != 1);
        interesse[1] = true;
        s = 1;
```

```
    print ("Thr 1:", s);
    vez = 0;
    interesse[1] = false;
```

Algoritmo de Dekker

```
int s = 0, vez = 0, interesse[2] = {false, false};
```

Thread 0

```
while (true)
    interesse[0] = true;
    while (interesse[1])
        if (vez != 0)
            interesse[0] = false;
            while (vez != 0);
            interesse[0] = true;
            s = 0;
            print ("Thr 0:", s);
            vez = 1;
            interesse[0] = false;
```

Thread 1

```
while (true)
    interesse[1] = true;
    while(interesse[0])
        if (vez != 1)
            interesse[1] = false;
            while(vez != 1);
            interesse[1] = true;
            s = 1;
            print ("Thr 1:", s);
            vez = 0;
            interesse[1] = false;
```

MC514–Sistemas Operacionais: Teoria e Prática

1s2008

Processos e Threads 4

Exclusão mútua

- Devemos garantir: exclusão mútua, ausência de deadlock e ausência de starvation
- ```
volatile int s; /* Variável compartilhada */
while (1) {
 /* Região não crítica */
 /* Protocolo de entrada */
 /* Região crítica */
 s = thr_id;
 printf ("Thr %d: %d", thr_id, s);
 /* Protocolo de saída */
}
```

## Vetor de Interesse

```
int s = 0;
int interesse[2] = {false, false};
```

### Thread 0

```
while (true)
 interesse[0] = true;
while (interesse[1]);
s = 0;
print("Thr 0: ", s);
interesse[0] = false;
```

### Thread 1

```
while (true)
 interesse[1] = true;
while (interesse[0]);
s = 1;
print("Thr 1: ", s);
interesse[1] = false;
```

- Veja o código: interesse.c

## Interesse para N threads

```
int interesse[N] = {false, ..., false}
while (true) { /* Código da Thread_i */
 interesse[i] = true;
 while (existe j!=i tal que (interesse[j]));
 s = i;
 print ("Thr ", i, ": ", s);
 interesse[i] = false;
}
```

- Veja o código: interessN.c

## Vetor de Interesse e Alternância

```
int s = 0, vez = 0;
int interesse[2] = {false, false};
```

### Thread 0

```
while (true)
 interesse[0] = true;
if (interesse[1])
 while (vez != 0);
s = 0;
print("Thr 0: ", s);
vez = 1;
interesse[0] = false;
```

### Thread 1

```
while (true)
 interesse[1] = true;
if (interesse[0])
 while (vez != 1);
s = 1;
print("Thr 1: ", s);
vez = 0;
interesse[1] = false;
```

- Veja o código: interesse-vez.c

## Interesse e vez para N threads

```
int interesse[N] = {false, ..., false}
int vez = 0;
while (true) { /* Código da Thread_i */
 interesse[i] = true;
 if (existe j!=i tal que (interesse[j]))
 while (vez != i);
 s = i;
 print ("Thr ", i, ": ", s);
 vez = (i + 1) % N;
 interesse[i] = false;
}
```

- Veja o código: interesse-vezN.c

## Algoritmo de Dekker (1965)

```
int s = 0, vez = 0, interesse[2] = {false, false};
```

### Thread 0

```
while (true)
 interesse[0] = true;
while (interesse[1])
 if (vez != 0)
 interesse[0] = false;
 while (vez != 0);
 interesse[0] = true;
s = 0;
print ("Thr 0: ", s);
vez = 1;
interesse[0] = false;
```

### Thread 1

```
while (true)
 interesse[1] = true;
while (interesse[0])
 if (vez != 1)
 interesse[1] = false;
 while (vez != 1);
 interesse[1] = true;
s = 1;
print ("Thr 1: ", s);
vez = 0;
interesse[1] = false;
```

## Sugestão para N threads

```
int vez = 0, interesse = {false, ..., false}
while (true) { /* Código da Thread_i */
 interesse[i] = true;
 while (existe j!=i tal que (interesse[j]))
 if (vez != i)
 interesse[i] = false;
 while (vez != i) ;
 interesse[i] = true;
 s = i;
 print ("Thr ", i, ": ", s);
 vez = (i+1) % N;
 interesse[i] = false;
}
```

## Sugestão para N threads

### Garante exclusão mútua?

- Uma thread só entra na região crítica após percorrer o vetor e verificar que nenhuma outra está interessada.

### Garante ausência de deadlock?

- Se todas estiverem interessadas, pelo menos uma thread (a da vez) consegue entrar na região crítica

### Garante progresso?

- Não. A vez pode ser passada para uma thread desinteressada.
- Veja o código dekkerN.c

## Outra sugestão...

```
int vez = -1, interesse = {false, ..., false}
while (true) { /* Código da Thread_i */
 interesse[i] = true;
 while (existe j!=i tal que (interesse[j]))
 if (vez != -1 && vez != i)
 interesse[i] = false;
 while (vez == -1 || vez != i) ;
 interesse[i] = true;
}
```

## Outra sugestão... (continuação)

```
s = i;
print ("Thr ", i, ": ", s);
vez = alguma interessada ou -1;
interesse[i] = false;
```

## Por que não funciona?

- Porque mais de uma thread pode achar que é a vez dela ao encontrar vez == -1
- Veja o código: outro-dekkerN.c

## Algoritmo de Dijkstra (1965)

```
int vez = -1, interesse = {false, ..., false}
while (true) { /* Código da Thread_i */
 interesse[i] = true;
 while (existe j!=i tal que (interesse[j]))
 if (vez != i)
 interesse[i] = false;
 while (vez != -1);
 vez = i;
 interesse[i] = true;
```

## Algoritmo de Dijkstra (1965)

(continuação)

```
s = i;
print ("Thr ", i, ": ", s);
vez = -1
interesse[i] = false;
```

## Algoritmo de Dijkstra

### Garante exclusão mútua?

- Uma thread só entra na região crítica após percorrer o vetor e verificar que nenhuma outra está interessada.

### Garante ausência de deadlock?

- Entre as interessadas, pelo menos a última a alterar a variável vez consegue entrar na região crítica

### Garante ausência de starvation?

- Não. Uma thread pode nunca conseguir ser a última a alterar vez.
- Veja o código dijkstra.c

## Proposta incorreta de Hyman (1966)

```
int s = 0, vez = 0, interesse[2] = {false, false};
```

### Thread 0

```
while (true)
 interesse[0] = true;
 while (vez != 0)
 while (interesse[1]);
 vez = 0;
```

### Thread 1

```
while (true)
 interesse[1] = true;
 while(vez != 1)
 while(interesse[0]);
 vez = 1;

s = 0;
print ("Thr 0: ", s);
interesse[0] = false;
```

```
s = 1;
print ("Thr 1: ", s);
interesse[1] = false;
```

## Algoritmo do Desempate (1981)

```
int s = 0, ultimo = 0, interesse[2] = {false, false};
```

### Thread 0

```
while (true)
 interesse[0] = true;
 ultimo = 0;
 while (ultimo == 0 &&
 interesse[1]);
 s = 0;
```

### Thread 1

```
while (true)
 interesse[1] = true;
 ultimo = 1;
 while (ultimo == 1 &&
 interesse[0]);
 s = 1;
```

```
print ("Thr 0: ", s);
interesse[0] = false;
```

```
print ("Thr 1: ", s);
interesse[1] = false;
```

## Objetivos

- Algoritmo do desempate (Peterson)
- Extensão para N threads
- Técnica do campeonato

## Algoritmo de Dekker (1965)

```
int s = 0, vez = 0, interesse[2] = {false, false};

Thread 0
while (true)
 interesse[0] = true;
 while (interesse[1])
 if (vez != 0)
 interesse[0] = false;
 while (vez != 0);
 interesse[0] = true;

 s = 0;
 print ("Thr 0:" , s);
 vez = 1;
 interesse[0] = false;

Thread 1
while (true)
 interesse[1] = true;
 while(interesse[0])
 if (vez != 1)
 interesse[1] = false;
 while(vez != 1);
 interesse[1] = true;

 s = 1;
 print ("Thr 1:" , s);
 vez = 0;
 interesse[1] = false;
```

## Proposta incorreta de Hymman (1966)

```
int s = 0, vez = 0, interesse[2] = {false, false};

Thread 0
while (true)
 interesse[0] = true;
 while (vez != 0)
 while (interesse[1]);
 vez = 0;
 s = 0;
 print ("Thr 0:" , s);
 interesse[0] = false;

Thread 1
while (true)
 interesse[1] = true;
 while(vez != 1)
 while(interesse[0]);
 vez = 1;
 s = 1;
 print ("Thr 1:" , s);
 interesse[1] = false;
```

## Algoritmo do Desempate (1981)

```
int s = 0, ultimo = 0, interesse[2] = {false, false};

Thread 0
while (true)
 interesse[0] = true;
 ultimo = 0;
 while (ultimo == 0 &&
 interesse[1]);
 s = 0;
 print ("Thr 0:" , s);
 interesse[0] = false;

Thread 1
while (true)
 interesse[1] = true;
 ultimo = 1;
 while (ultimo == 1 &&
 interesse[0]);
 s = 1;
 print ("Thr 1:" , s);
 interesse[1] = false;
```

## Algoritmo do Desempate 3 Threads (bug!)

```
int s=0, ultimo=0, interesse[3];

Thread 0
while (true)
 interesse[0] = true;
 ultimo = 0;
 while (ultimo == 0 && (interesse[1] || interesse[2]));
 s = 0;
 print ("Thr 0:" , s);
 interesse[0] = false;
```

## Algoritmo do desempate Extensão para 3 threads

- Para 2 threads, podemos estabelecer que a thread de identificador último perde;
- Caso 3 threads alterem a variável ultimo simultaneamente, só poderemos identificar a que fez a última alteração.
- Como indicar que 2 threads perderam?

## Algoritmo do Desempate 3 Threads

```
int s=0, ultimo, penultimo, interesse[3];

Thread 0
while (true)
 interesse[0] = true;
 ultimo = 0;
 while (ultimo == 0 && interesse[1] && interesse[2]);
 penultimo = 0;
 while (penultimo == 0 && (interesse[1] || interesse[2]));
 s = 0;
 print ("Thr 0:" , s);
 interesse[0] = false;
```

## Algoritmo do Desempate

### Características

Região crítica

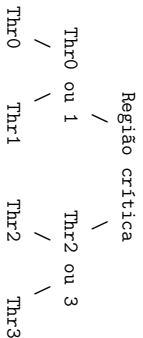
/

\

Thr0    Thr1

- Funciona para 2 threads
- Variável ultimo é acessada pelas 2 threads
- Variável interesse[i] é acessada
  - para escrita pela thread i
  - para leitura pela thread adversária

Campeonato entre 4 threads



- A thread campeã da disputa entre Thr0 e Thr1 disputa a região crítica com a thread campeã da disputa entre Thr2 e Thr3.
- Todas as partidas são instâncias do algoritmo do desempate.

Campeonato entre 4 threads

Variáveis de controle replicadas

```
int ultimo_final = 0;
int interesse_final[2] = {false, false};

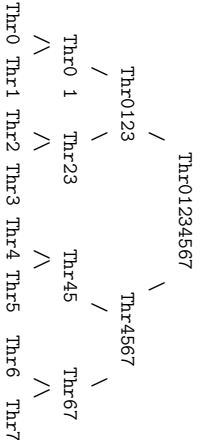
int ultimo01 = 0;
int interesse01[2] = {false, false};

int ultimo23 = 2;
int interesse23[2] = {false, false};

• Veja código: camp4.c
```

Exclusão mútua entre N threads

Abordagem do campeonato



- As threads podem concorrer duas a duas
- Garante ausência de starvation?

Algoritmo do desempate

Extensão para N threads

- Caso *N* threads alterem a variável ultimo simultaneamente, só poderemos identificar a que fez a última alteração.
- Como indicar que *N* – 1 threads perderam?

Algoritmo do desempate

N threads

- Dividimos o problema em N-1 fases (0..N-2)
- A cada fase, conseguimos identificar uma thread perdadora, que fica esperando
- Variáveis de controle:  
int interesse[N] ; /\* -1..N-2 \*/  
int ultimo[N-1] ;

Desempate para N threads

Estado inicial

interesse

|      |      |      |      |      |
|------|------|------|------|------|
| Thr0 | Thr1 | Thr2 | Thr3 | Thr4 |
| -1   | -1   | -1   | -1   | -1   |

ultimo

|       |       |       |       |
|-------|-------|-------|-------|
| Fase0 | Fase1 | Fase2 | Fase3 |
| -     | -     | -     | -     |

Desempate para N threads

Todas as threads interessadas

interesse

|      |      |      |      |      |
|------|------|------|------|------|
| Thr0 | Thr1 | Thr2 | Thr3 | Thr4 |
| 0    | 0    | 0    | 0    | 0    |

ultimo

|       |       |       |       |
|-------|-------|-------|-------|
| Fase0 | Fase1 | Fase2 | Fase3 |
| 2     | -     | -     | -     |

- Thread 2 não poderá mudar de fase

Desempate para N threads

Todas as threads interessadas

interesse

|      |      |      |      |      |
|------|------|------|------|------|
| Thr0 | Thr1 | Thr2 | Thr3 | Thr4 |
| 1    | 1    | 0    | 1    | 1    |

ultimo

|       |       |       |       |
|-------|-------|-------|-------|
| Fase0 | Fase1 | Fase2 | Fase3 |
| 2     | 1     | -     | -     |

- Thread 1 não poderá mudar de fase

## Desempate para N threads

Todas as threads interessadas

| Thr0 | Thr1 | Thr2 | Thr3 | Thr4 |
|------|------|------|------|------|
| 2    | 1    | 0    | 2    | 2    |

interesse

| Fase0 | Fase1 | Fase2 | Fase3 |
|-------|-------|-------|-------|
| 2     | 1     | 0     | -     |

ultimo

- Thread 0 não poderá mudar de fase

## Desempate para N threads

Todas as threads interessadas

| Thr0 | Thr1 | Thr2 | Thr3 | Thr4 |
|------|------|------|------|------|
| 2    | 1    | 0    | 3    | 3    |

interesse

| Fase0 | Fase1 | Fase2 | Fase3 |
|-------|-------|-------|-------|
| 2     | 1     | 0     | 4     |

ultimo

- Thread 3 pode entrar na região crítica

## Desempate para N threads

Algumas threads interessadas

| Thr0 | Thr1 | Thr2 | Thr3 | Thr4 |
|------|------|------|------|------|
| 1    | 0    | -1   | -1   | -1   |

interesse

| Fase0 | Fase1 | Fase2 | Fase3 |
|-------|-------|-------|-------|
| 1     | 0     | -     | -     |

ultimo

- Thread 1 deverá esperar
- Thread 0 pode progredir pois as outras threads não estão interessadas

## Desempate para N threads

```
int interesse[N], ultimo [N-1];
```

**Thread i:**

```
for (f = 0; f < N-1; f++)
 interesse[i] = f;
ultimo[f] = i;
for (k = 0; k < N && ultimo[f] == i; k++)
 if (k == i)
 while (f <= interesse[k] && ultimo[f] == i);
s = i;
print ("Thr ", i, s);
interesse[i] = -1;
```

## Desempate para N Threads

- Garante exclusão mútua
- Garante ausência de deadlock
- Garante ausência de starvation
  - *deve haver um limite no número de vezes que outras threads podem entrar na região crítica (rodadas) a partir do momento que uma thread submete o pedido e o momento em que ela executa a região crítica.*
- espera máxima =  $N(N-1)/2$  rodadas?

## Desempate para N Threads

Pior cenário?

- $Thr_0$  perde de n-1 threads na fase 0
- Estas N-1 threads tentam novamente
- $Thr_0$  é desbloqueada e uma outra thread fica bloqueada na fase 0.
- $Thr_0$  perde de n-2 threads na fase 1
- ...
- Como ilustrar este cenário?

## Algoritmo de Knuth

```
enum estado {passive, requesting, in_cs};
int vez, interesse[N];

Thread i:
do {
 interesse[i] = requesting;
 vez_local = vez;
 while (vez_local != i)
 if (interesse[vez_local] != passive)
 vez_local = vez;
 else vez_local = (vez + 1) % N;
 interesse[i] = in_cs;
 } while (existe j=i tal que interesse[j] == in_cs);
vez = i;
```

## Algoritmo de Knuth (continuação)

- ```
s = i;
print ("Thr ", i, s);
vez = (thr_id + 1) % N;
interesse[i] = passive;
```
- Espera máxima pode parecer linear, mas ...

Algoritmo de Knuth

Pior cenário $2^{N-1} - 1$

- 3 Threads: $T_2 T_1 T_2 T_0$
 - T_0 faz o pedido ($vez = 1$)
 - T_2 pára imediatamente antes de setar in_cs
 - T_1 pára imediatamente antes de setar in_cs
 - T_2 entra na RC e $vez = 0$
 - T_1 entra na RC e $vez = 2$
 - T_2 entra na RC e $vez = 0$
- 4 Threads: $T_3 T_2 T_3 T_1 T_3 T_2 T_3 T_0$

Algoritmo de Knuth

Pior cenário $2^{N-1} - 1$

- 4 Threads: $T_3 T_2 T_3 T_1 T_3 T_2 T_3 T_0$
 - T_0 faz o pedido ($vez = 1$)
 - T_3, T_2 e T_1 param antes de setar in_cs
 - T_3 entra na RC e $vez = 0$
 - T_2 entra na RC e $vez = 3$
 - T_3 entra na RC e $vez = 0$
 - T_1 entra na RC e $vez = 2$
 - T_3 pára imediatamente antes de setar in_cs
 - T_2 entra na RC e $vez = 3$
 - T_3 entra na RC e $vez = 0$

Algoritmo de Bruijn

```
enum estado {passive, requesting, in_cs};
int vez, interesse[N];
```

Thread i:

```
do {
    interesse[i] = requesting;
    vez_local = vez;
    while (vez_local != i)
        if (interesse[vez_local] != passive)
            vez_local = vez;
    else vez_local = (vez + 1) % N;
    interesse[i] = in_cs;
} while (existe j!=i tal que interesse[j] == in_cs);
```

Algoritmo de Bruijn (continuação)

```
s = i;
print ("Thr ", i, s);
if (interesse[vez] == passive || vez == i)
    vez = (thr_id + 1) % N;
interesse[i] = passive;
```

- Espera máxima $n(n-1)/2$

MC514–Sistemas Operacionais: Teoria e Prática

1s2008

Processos e Threads 6

Objetivos

- Abordagem da Thread Gerente
- Algoritmo da Padaria
- Prioridades para Threads

Algoritmos de Exclusão Mútua

- E se tivéssemos uma thread gerente?
 - Os algoritmos seriam mais simples?
 - Qual o grande ponto negativo desta abordagem?
- Veja os programas: gerente?.c

Algoritmo da Padaria

- Análogo a um sistema de distribuição de senhas a clientes em uma loja
- A thread com a senha de menor número é atendida
- A própria thread deve escolher o seu número

Algoritmo da padaria

Primeira tentativa

```
num[N] = { 0, 0, ..., 0 }
```

Thread i:

```
num[i] = max (num[0]...num[N-1]) + 1
```

```
for (j = 0; j < N; j++)  
    while (num[j] != 0 && num[j] < num[i]) ;
```

```
s = i;
```

```
print ("Thr ", i, s);
```

```
num[i] = 0;
```

Algoritmo da padaria

Segunda tentativa

```
num[N] = { 0, 0, ..., 0 }
```

Thread i:

```
num[i] = max (num[0]...num[N-1]) + 1
```

```
for (j = 0; j < N; j++)  
    while (num[j] != 0 &&  
           (num[j] < num[i] || num[i] == num[j] && j < i));
```

```
s = i;
```

```
print ("Thr ", i, s);
```

```
num[i] = 0;
```

Algoritmo da padaria

```
escolhendo[N] = { false, false, ..., false }  
num[N] = { 0, 0, ..., 0 }
```

Thread i:

```
escolhendo[i] = true;
```

```
num[i] = max (num[0]...num[N-1]) + 1
```

```
escolhendo[i] = false;
```

```
for (j = 0; j < N; j++)
```

```
    while (escolhendo[j]) ;
```

```
    while (num[j] != 0 &&
```

```
           (num[j] < num[i] || num[i] == num[j] && j < i));
```

```
s = i;
```

```
print ("Thr ", i, s);
```

```
num[i] = 0;
```

Black-White Bakery

Gadi Taubenfe

- The Black-White Bakery Algorithn. Proceedings of the 18th international symposium on distributed computing, Amsterdam, the Netherlands, October 2004. In: LNCS 3274 Springer Verlag 2004, 56-70
- rodadas de senhas coloridas
- permite senhas de tamanho fixo

Filas de prioridades diferentes

- Suponha que o gerente da padaria está pensando em implantar atendimento especial a idosos e gestantes
- Existem threads prioritárias e outras menos prioritárias;
- Nenhuma thread menos prioritária é atendida se houver uma thread mais prioritária esperando;
- Se uma thread menos prioritária estiver sendo atendida, a mais prioritária deve esperar;

Modificação para duas filas

Duas instâncias do algoritmo da padaria

```
#define N 10  
#define M 5
```

```
esc[N] = { false, false, ..., false }
```

```
num[N] = { 0, 0, ..., 0 }
```

```
esc_pri[M] = { false, false, ..., false }
```

```
num_pri[M] = { 0, 0, ..., 0 }
```

Modificação para duas filas

Uma instância do algoritmo do desempate

```
#define PRI 0  
#define NAO_PRI 1  
int vez;  
int interesse[2];
```

Thread menos prioritária

```
esc[i] = true;  
num[i] = max (num[0]...num[N-1]) + 1  
esc[i] = false;  
for (j = 0; j < N; j++)  
    while (esc[j]) ;  
    while (num[j] != 0 &&  
           (num[j] < num[i] || num[i] == num[j] && j < i));  
    interesse[NAO_PRI] = 1;  
    vez = NAO_PRI;  
    while (vez == NAO_PRI && interesse[PRI]);  
s = i;  
print ("Thr ", i, s);  
interesse[NAO_PRI] = 0;  
num[i] = 0;
```

Thread mais prioritária (?)

```
esc_pri[i] = true;
num_pri[i] = max (num_pri[0] .. num_pri[M-1]) + 1
esc_pri[i] = false;
for (j = 0; j < M; j++)
    while (esc_pri[j]) ;
    while (num_pri[j] != 0 && (num_pri[j] < num_pri[i] ||
        num_pri[i] == num_pri[j] && j < i));
    interesse[PRI] = 1;
vez = PRI;
while (vez == PRI && interesse[MAQ_PRI]);
s = i;
print ("Thr ", i, s);
interesse[PRI] = 0;
num_pri[i] = 0;
```

Thread mais prioritária

```
/* Código da padaria simples */
if (!interesse[PRI]){
    interesse[PRI] = 1;
    vez = PRI;
    while (vez == PRI && interesse[MAQ_PRI]);
}
/* Região crítica */
if (não existe j!= i : num_pri[j] > 0)
    interesse[PRI] = 0;
num_pri[i] = 0;
```

MC514–Sistemas Operacionais: Teoria e Prática

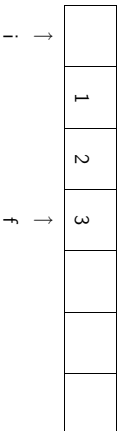
1s2008

Produtores e Consumidores

Problema do Produtor-Consumidor

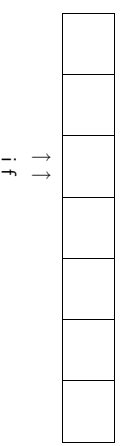
- Dois processos partilham um *buffer* de tamanho fixo
- O produtor insere informação no *buffer*
- O consumidor remove informação do *buffer*

Controle do buffer



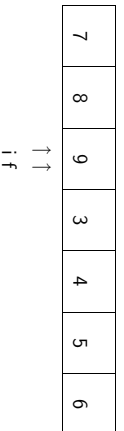
- **i**: aponta para a posição anterior ao primeiro elemento
- **f**: aponta para o último elemento
- **c**: indica o número de elementos presentes
- **N**: indica o número máximo de elementos

Buffer vazio



- **i == f**
- **c == 0**

Buffer cheio



- **i == f**
- **c == N**

Comportamento básico

```
int buffer[N];
int c = 0;
int i = 0, f = 0;

Produtor
while (true)
    f = (f+1)%N;
    buffer[f] = produz();
    c++;

Consumidor
while (true)
    i = (i+1)%N;
    consome(buffer[i]);
    c--;
```

Veja código: prod-cons-basico.c

Problemas

1. produtor insere em posição que *ainda* não foi consumida
2. consumidor remove de posição *já* foi consumida

Veja código: prod-cons-basico-bug.c

Condição de disputa

- | Produtor | Consumidor |
|----------|------------|
| c++; | c--; |
| mov rp,c | mov rc,c |
| inc rp | dec rc |
| mov c,rp | mov c,rc |
- Decremento/incremento não são atômicos
 - Veja código: prod-cons-basico-race.c

Algoritmo com espera ocupada

```
int buffer[N];
int c = 0;
int i = 0, f = 0;
```

Produtor

```
while (true)
    while (c == N);
    f = (f+1)%N;
    buffer[f] = produz();
    c++;
```

Consumidor

```
while (true)
    while (c == 0);
    i = (i+1)%N;
    consume(buffer[i]);
    c--;
```

Veja código: prod-cons-basico-busy-wait.c

Semáforos

- Semáforos são *contadores especiais* para recursos compartilhados.
- Proposto por Dijkstra (1965)
- Operações básicas (atômicas):
 - decremento (down, wait ou P)
 - bloqueia se o contador for nulo
 - incremento (up, signal (post) ou V)
 - nunca bloqueia

Semáforos

Comportamento básico

- sem_init(s, 5)
- wait(s)
 - if (s == 0)
 - bloqueia_processo();
 - else s--;
- signal(s)
 - if (s == 0 && existe processo bloqueado)
 - acorda_processo();
 - else s++;

Vários produtores e consumidores

```
semaforo cheio = 0, vazio = N;
semaforo lock_prod = 1, lock_cons = 1;
```

Produtor:

```
while (true)
    wait(vazio);
    wait(lock_prod);
    f = (f + 1) % N;
    buffer[f] = produz();
    signal(lock_prod);
    signal(cheio);
```

Consumidor:

```
while (true)
    wait(cheio);
    wait(lock_cons);
    i = (i + 1) % N;
    consume(buffer[i]);
    signal(lock_cons);
    signal(vazio);
```

Produtor-Consumidor com Semáforos

```
semaforo cheio = 0;
semaforo vazio = N;
```

Produtor:

```
while (true)
    wait(vazio);
    f = (f+1)%N;
    buffer[f] = produz();
    signal(cheio);
```

Consumidor:

```
while (true)
    wait(cheio);
    i = (i+1)%N;
    consume(buffer[i]);
    signal(vazio);
```

Veja código: prod-cons-sem.c

Vários produtores e consumidores

```
semaforo cheio = 0, vazio = N;
semaforo lock_prod = 1, lock_cons = 1;
```

Produtor:

```
while (true)
    item = produz();
    wait(vazio);
    wait(lock_prod);
    f = (f + 1) % N;
    buffer[f] = item;
    signal(lock_prod);
    signal(cheio);
```

Consumidor:

```
while (true)
    wait(cheio);
    wait(lock_cons);
    i = (i + 1) % N;
    item = buffer[i];
    signal(lock_cons);
    signal(vazio);
    consume(item);
```

Semáforos

- Exclusão mútua
- Sincronização

Mutex locks

⇒ Exclusão mútua

- pthread_mutex_lock
- pthread_mutex_unlock

Variáveis de condição

⇒ Sincronização

- pthread_cond_wait
- pthread_cond_signal
- pthread_cond_broadcast
- precisam ser utilizadas em conjunto com mutex_locks

Thread 0 acorda Thread 1

```
int s;  
/* Veja cond_signal.c */  
Thread 1:  
mutex_lock(&mutex);  
if (preciso_esperar(s))  
    cond_wait(&cond, &mutex);  
mutex_unlock(&mutex);  
  
Thread 0:  
mutex_lock(&mutex);  
if (devo_acordar_thread_1(s))  
    cond_signal(&cond);  
mutex_unlock(&mutex);
```

Produtor-Consumidor

```
int c = 0; /* Contador de posições ocupadas */  
mutex_t lock_c; /* lock para o contador */  
  
cond_t pos_vazia; /* Para o produtor esperar */  
cond_t pos_ocupada; /* Para o consumidor esperar */
```

Produtor-Consumidor

```
int f = 0;  
Produtor:  
mutex_lock(&lock_c);  
if (c == N)  
    cond_wait(&pos_vazia, &lock_c);  
f = (f+1)%N;  
buffer[f] = produz();  
c++;  
if (c == 1)  
    cond_signal(&pos_ocupada);  
mutex_unlock(&lock_c);
```

Produtor-Consumidor

```
int i = 0;  
Consumidor:  
mutex_lock(&lock_c);  
if (c == 0)  
    cond_wait(&pos_ocupada, &lock_c);  
i = (i+1)%N;  
consume(buffer[i]);  
if (c == N-1)  
    cond_signal(&pos_vazia);  
c--;  
mutex_unlock(&lock_c);
```

Produtor-Consumidor

```
cond_t pos_vazia, pos_ocupada; mutex_t lock_v, lock_o;  
int i = 0, f = 0, nv = N, no = 0;  
Produtor:  
mutex_lock(&lock_v);  
if (nv == 0) cond_wait(&pos_vazia, &lock_v);  
nv--;  
mutex_unlock(&lock_v);  
f = (f+1)%N;  
buffer[f] = produz();  
mutex_lock(&lock_o);  
no++;  
cond_signal(&pos_ocupada);  
mutex_unlock(&lock_o);
```

Produtor-Consumidor

Consumidor:

```
mutex_lock(&lock_o);
if (no == 0) cond_wait(&pos_ocupada, &lock_o);
no--;
mutex_unlock(&lock_o);
i = (i+1)%N;
consume(buffer[i]);
mutex_lock(&lock_v);
nv++;
cond_signal(&pos_vazia);
mutex_unlock(&lock_v);
```

Produtores-Consumidores

Será que funciona?

```
cond_t pos_vazia, pos_ocupada;
mutex_t lock_v, lock_o;
int nv = N, no = 0;

mutex_t lock_i, lock_f;
int i = 0, f = 0;
```

Produtores-Consumidores

Consumidor:

```
mutex_lock(&lock_o);
if (no == 0) cond_wait(&pos_ocupada, &lock_o);
no--;
mutex_unlock(&lock_o);
mutex_lock(&lock_i);
i = (i+1)%N;
item = buffer[i];
mutex_unlock(&lock_i);
mutex_lock(&lock_v);
nv++;
cond_signal(&pos_vazia);
mutex_unlock(&lock_v);
consume(item);
```

Thread 0 acorda alguma thread

```
int s;
/* Veja cond_signal_n.c */
```

Thread i:

```
mutex_lock(&mutex);
if (preciso_esperar(s))
    cond_wait(&cond, &mutex);
mutex_unlock(&mutex);
```

Thread 0:

```
mutex_lock(&mutex);
if (devo_acordar_alguna_thread(s))
    cond_signal(&mutex);
mutex_unlock(&mutex);
```

Produtores-Consumidores

Produtor:

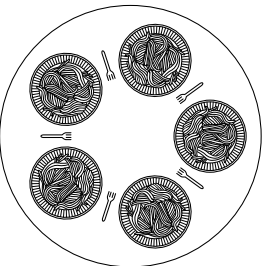
```
item = produz();
mutex_lock(&lock_v);
if (nv == 0) cond_wait(&pos_vazia, &lock_v);
nv++;
mutex_unlock(&lock_v);
mutex_lock(&lock_f);
f = (f+1)%N;
buffer[f] = item;
mutex_unlock(&lock_f);
mutex_lock(&lock_o);
no++;
cond_signal(&pos_ocupada);
mutex_unlock(&lock_o);
```

MC514-Sistemas Operacionais: Teoria e Prática

1s2008

Filósofos Famintos

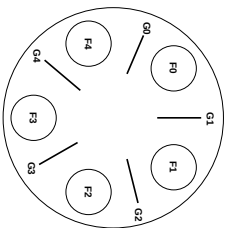
Jantar dos Filósofos



Boas soluções

- ausência de *deadlock*
- ausência de *starvation*
- alto grau de paralelismo

Representação da mesa



```
- T - T - T - T - T -
- T - T - H - T - T -
- T - T | H | T - T -
- T - T | E | T - T -
```

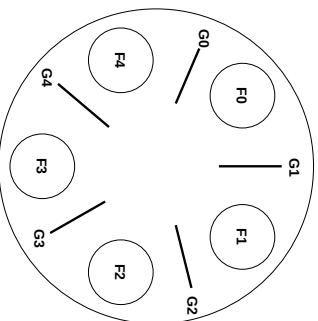
Implementação com Semáforos

Um semáforo por garfo

- sem_init(garfo, 1)
- wait(garfo)
- signal(garfo)

Filósofos famintos

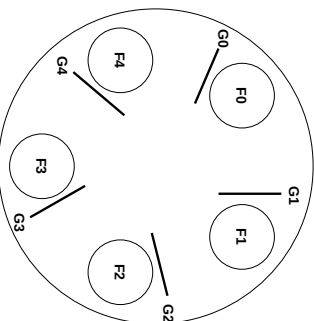
Um semáforo por garfo



Implementação simplista

```
Filósofo i:
while (true)
    pensa();
    wait(garfo[i]);
    wait(garfo[(i+1) % N]);
    come();
    signal(garfo[i]);
    signal(garfo[(i+1) % N]);
```

Deadlock



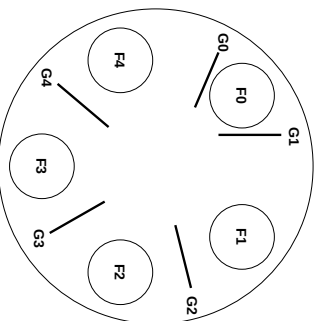
Veja códigos: deadlock.c e deadlock-bug-exibicao.c

Outra tentativa...

```
semáforo lock = 1;

Filósofo i:
while (true)
    pensa();
    wait(lock);
    wait(garfo[i]);
    wait(garfo[(i+1) % N]);
    come();
    signal(garfo[(i+1) % N]);
    signal(garfo[i]);
    signal(lock);
```

Baixíssimo paralelismo



Veja código: sem.central.c

O que acontece se lock == 2?

```
semáforo lock = 2;
```

Filósofo i:

```
while (true)
    pensa();
    wait(lock);
    wait(garfo[i]);
    wait(garfo[(i+1) % N]);
    come();
    signal(garfo[(i+1) % N]);
    signal(garfo[i]);
    signal(lock);
```

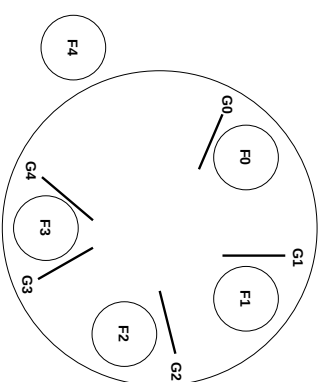
Menos lugares à mesa

semaforo lugar_mesa = 4;

Filósofo i:

```
while (true)
    pensa();
    wait(lugar_mesa);
    wait(garfo[i]);
    wait(garfo[(i+1) % N]);
    come();
    signal(garfo[(i+1) % N]);
    signal(garfo[i]);
    signal(lugar_mesa);
```

Menos lugares à mesa

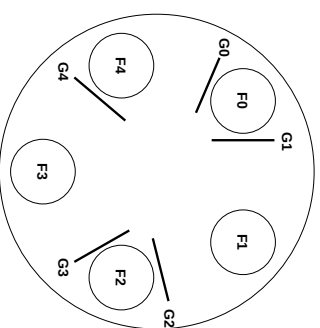


Veja código: limite_lugares.c

Solução assimétrica

```
while (true)
    pensa();
    if (i % 2 == 0)
        wait(garfo[i]);
        wait(garfo[(i+1) % N]);
    else
        wait(garfo[(i+1) % N]);
        wait(garfo[i]);
    come();
    signal(garfo[(i+1) % N]);
    signal(garfo[i]);
```

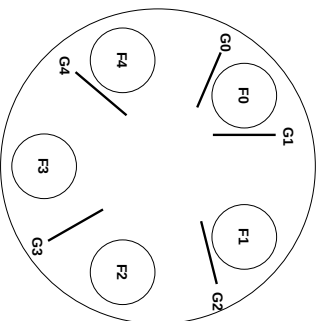
Solução assimétrica



Veja código: assimetrica.c

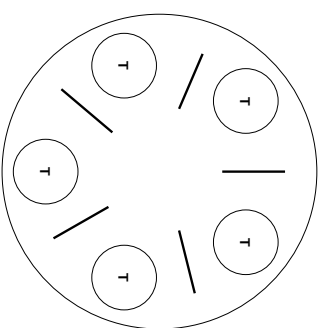
Solução assimétrica

Baixo paralelismo?!



Filósofos famintos

Um semaforo por filósofo



Solução do livro Tanenbaum

```
semaforo lock;
semaforo filosofo[N] = {0, 0, 0, ..., 0}
int estado[N] = {T, T, T, ..., T}
```

Filósofo i:

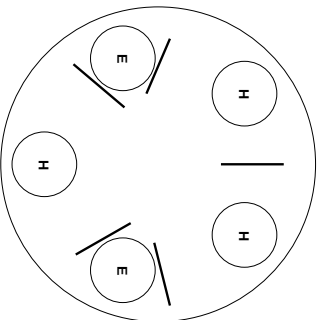
```
while (true)
    pensa();
    pega_garfos();
    come();
    solta_garfos();
```

```
testa_garfos(int i)
    if (estado[i] == H && estado[fil_esq] != E &&
        estado[fil_dir] != E)
        estado[i] = E;
        signal(filosofo[i]);
```

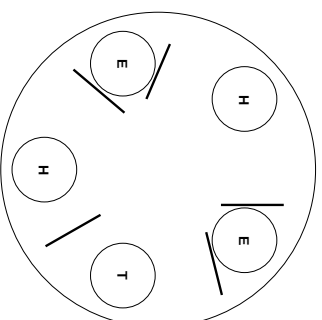
```
pega_garfos()
    wait(lock);
    estado[i] = H;
    testa_garfos(i);
    signal(lock);

    solta_garfos()
    wait(lock);
    estado[i] = T;
    testa_garfos(fil_esq);
    testa_garfos(fil_dir);
    signal(lock);
```

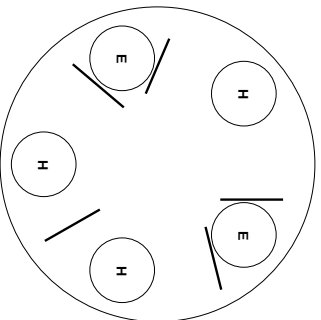
Alto paralelismo



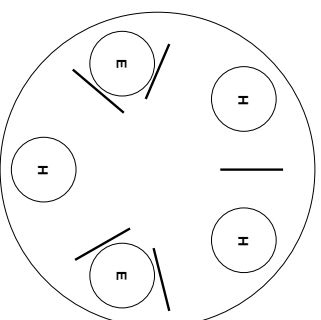
Alto paralelismo



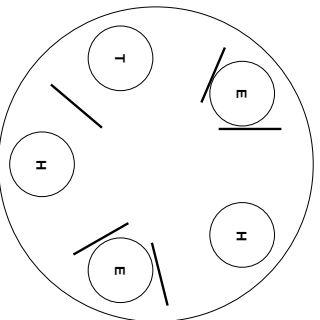
Alto paralelismo



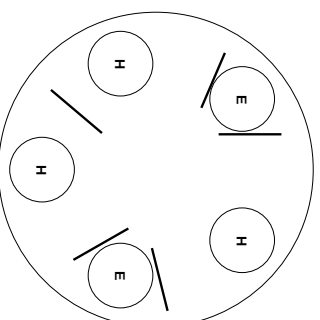
Alto paralelismo



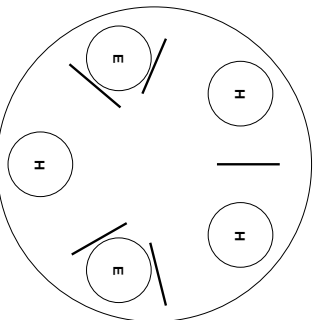
Alto paralelismo



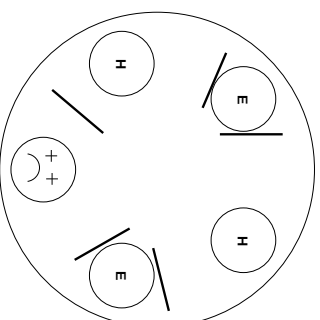
Alto paralelismo



Alto paralelismo



Starvation



Como matar os filósofos de fome?

- É preciso ajustar os tempos.
- Veja o código: `tanen-4-2.c` e `tanen-5-1.c`
- Como implementar `tanen-8-2.c`?

MC514-Sistemas Operacionais: Teoria e Prática

1s2008

Mutex locks simples, recursivos e com verificação de erros

Mutex locks

⇒ Exclusão mútua

- `pthread_mutex_lock`
- `pthread_mutex_unlock`

Variáveis de condição

⇒ Sincronização

- `pthread_cond_wait`
- `pthread_cond_signal`
- `pthread_cond_broadcast`
- precisam ser utilizadas em conjunto com `mutex_locks`

Thread 0 acorda Thread 1

```
int s; /* Veja cond_signal.c */
```

Thread 1:

```
mutex_lock(&mutex);
if (preciso_esperar(s))
    cond_wait(&cond, &mutex);
mutex_unlock(&mutex);
```

Thread 0:

```
mutex_lock(&mutex);
if (devo_acordar_thread_1(s))
    cond_signal(&cond);
mutex_unlock(&mutex);
```

Thread 0 acorda todas as threads

```
int s; /* Veja cond_broadcast.c */
```

Thread i:

```
mutex_lock(&mutex);
if (preciso_esperar(s))
    cond_wait(&cond, &mutex);
mutex_unlock(&mutex);
```

Thread 0:

```
mutex_lock(&mutex);
if (devo_acordar_todas_as_threads(s))
    cond_broadcast(&cond);
mutex_unlock(&mutex);
```

Thread 0 acorda todas as threads

mas algumas delas voltam a dormir

```
int s; /* Veja cond_broadcast2.c */
```

Thread i:

```
mutex_lock(&mutex);
while (preciso_esperar(s)) /* <===== */
    cond_wait(&cond, &mutex);
mutex_unlock(&mutex);
```

Thread 0:

```
mutex_lock(&mutex);
if (devo_acordar_todas_as_threads(s))
    cond_broadcast(&cond);
mutex_unlock(&mutex);
```

Thread 0 acorda 1 (ou +) threads

```
int s; /* Veja cond_signal_n.c */
```

Thread i:

```
mutex_lock(&mutex);
while (preciso_esperar(s))
    cond_wait(&cond, &mutex);
mutex_unlock(&mutex);
```

Thread 0:

```
mutex_lock(&mutex);
if (devo_acordar_pelo_menos_uma_thread(s))
    cond_signal(&mutex);
mutex_unlock(&mutex);
```

Importância do teste com while

- Cenário 1: Implementação não garante que apenas uma thread será acordada
- Cenário 2:
 - Thread *i* vai dormir pois *C* é verdadeira
 - Thread *j* acorda thread *i* pois torna *C* falsa
 - Thread *k* pega o lock e torna *C* verdadeira
 - Thread *i* executa de maneira inconsistente
 - Veja o código teste-cond.wait.c

Locks simples

Estrutura protegida por um mutex lock

```
typedef struct estrutura {
    mutex_t lock;
    Tipo1 campo1;
    Tipo2 campo2;
    Tipo3 campo3;
} Estrutura;
```

- Como escrever as funções que fazem acesso a estes campos?

Locks simples

E se funcao2 invocasse funcao1?

```
void funcao2(Estrutura *e) {
    mutex_lock(&e->lock);
    /* ... */
    if (condicao)
        funcao1(e);
    /* ... */
    mutex_unlock(&e->lock);
}
```

Locks simples

Funções atômicas

```
void funcao1(Estrutura *e) {
    mutex_lock(&e->lock);
    /* ... */
    mutex_unlock(&e->lock);
}
```

```
void funcao2(Estrutura *e) {
    mutex_lock(&e->lock);
    /* ... */
    mutex_unlock(&e->lock);
}
```

Deadlock de uma thread só

```
void f() {
    mutex_lock(&lock);
    mutex_lock(&lock);
}
```

Veja o código: deadlock.c

Locks simples

E se funcao2 invocasse funcao1?

Possíveis soluções:

- Replicação de código
- Função auxiliar não atômica

```
void funcao1(Estrutura *e) {
    mutex_lock(&e->lock);
    aux_funcao1(e);
    mutex_unlock(&e->lock);
}
```

Locks recursivos

```
void f() {
    mutex_lock(&lock);
    /* faz alguma coisa */
    mutex_unlock(&lock);
}
```

```
void g() {
    mutex_lock(&lock);
    f();
    /* faz outra coisa */
    mutex_unlock(&lock);
}
```

Locks recursivos

Implementação a partir de locks simples

e variáveis de condição

```
typedef struct {
    pthread_t thr;
    cond_t cond;
    mutex_t lock;
    int c;
} rec_mutex_t;
```

rec_mutex_lock()

```
int rec_mutex_lock(rec_mutex_t *rec_m) {
    pthread_mutex_lock(&rec_m->lock);
    if (rec_m->c == 0) { /* Lock livre */
        rec_m->c = 1;
        rec_m->thr = pthread_self();
    } else /* Mesma thread */
        if (pthread_equal(rec_m->thr,
                           pthread_self()))
            rec_m->c++;
    else {
        /* Thread deve esperar */
    }
```

rec_mutex_lock()

```
else {
    /* Thread deve esperar */
    while (rec_m->c != 0)
        pthread_cond_wait(&rec_m->cond,
                           &rec_m->lock);
    rec_m->thr = pthread_self();
    rec_m->c = 1;
}
pthread_mutex_unlock(&rec_m->lock);
return 0;
}
```

Verificação de erros

rec_mutex_unlock()

```
int rec_mutex_unlock(rec_mutex_t *rec_m) {
    pthread_mutex_lock(&rec_m->lock);
    if (rec_m->c == 0 ||
        !pthread_equal(rec_m->thr,
                        pthread_self())) {
        pthread_mutex_unlock(&rec_m->lock_var);
        return ERROR;
    }
    else
        /* ... */
}
```

rec_mutex_unlock()

```
int rec_mutex_unlock(rec_mutex_t *rec_m) {
    pthread_mutex_lock(&rec_m->lock);
    rec_m->c--;
    if (rec_m->c == 0)
        pthread_cond_signal(&rec_m->cond);
    pthread_mutex_unlock(&rec_m->lock);
    return 0;
}
```

Locks recursivos

Implementação reduzida

```
typedef struct {
    pthread_t thr;
    mutex_t lock;
    int c;
} rec_mutex_t;
```

rec_mutex_lock()

```
int rec_mutex_lock(rec_mutex_t *rec_m) {
    if (!pthread_equal(rec_m->thr,
                       pthread_self())) {
        pthread_mutex_lock(&rec_m->lock);
        rec_m->thr = pthread_self();
        rec_m->c = 1;
    }
    else
        rec_m->c++;
    return 0;
}
```

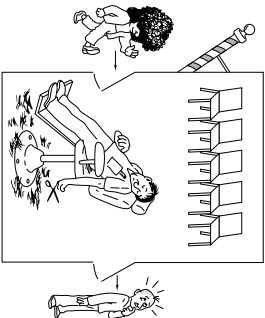
rec_mutex_unlock()

```
int rec_mutex_unlock(rec_mutex_t *rec_m) {
    if (!pthread_equal(rec_m->thr, pthread_self())
        || rec_m->c == 0)
        return ERROR;
    rec_m->c--;
    if (rec_m->c == 0)
        pthread_mutex_unlock(&rec_m->lock);
    return 0;
}
```

- A implementação reduzida tem comportamento equivalente à primeira?

- Veja o código pthread_mutex_lock.c
- Quando que as variáveis de condição são imprescindíveis?

Barbeiro Dorminhoco



Cadeiras da sala de espera

- Se não tiver onde sentar, o cliente vai embora...
 - Esta abordagem funciona?
- ```
semaforo cadeiras = 5;
wait(cadeiras);
```

## Barbeiro Dorminhoco

- Se não há clientes, o barbeiro adormece;
- Se a cadeira do barbeiro estiver livre, um cliente pode ser atendido imediatamente;
- O cliente espera pelo barbeiro se houver uma cadeira de espera vazia.
- Se não tive onde sentar, o cliente vai embora...

### Cadeiras da sala de espera

- Esta abordagem funciona?
- ```
semaforo cadeiras = 5;
if (sem_getvalue(cadeiras) > 0)
    wait(cadeiras);
```

Cadeiras da sala de espera

```
mutex_lock mutex;
int cadeiras = 5;

mutex_lock(mutex);
if (cadeiras > 0)
    cadeiras--;
mutex_unlock(mutex);
entra_na_barbearia();
else
    mutex_unlock(mutex);
desiste_de_cortar_o_cabelo();
```

Clientes só esperam nas cadeiras

```
semaforo cadeiras = 5;

if (trywait(cadeiras) == 0)
    entra_na_barbearia();
else
    desiste_de_cortar_o_cabelo();
```

Disputa pela cadeira do barbeiro

- ```
semaforo cadeiras = 5;
semaforo cad_barbeiro = 1;

if (trywait(cadeiras) == 0)
 wait(cad_barbeiro);
```
- Todo cliente precisa passar pela sala de espera?

### Disputa pela cadeira do barbeiro

- ```
semaforo cadeiras = 5;
semaforo cad_barbeiro = 1;

if (trywait(cad_barbeiro) == 0)
    if (trywait(cadeiras) == 0)
        wait(cad_barbeiro);
```
- Esta abordagem é justa?

Iniciando o corte

```
semaforo cadeiras = 5;  
semaforo cad_barbeiro = 1;
```

```
if (trywait(cadeiras) == 0)  
    wait(cad_barbeiro);
```

- Como avisar o barbeiro que você está esperando?

Iniciando o corte

```
semaforo cadeiras = 5;  
semaforo cad_barbeiro = 1;
```

```
if (trywait(cadeiras) == 0)  
    wait(cad_barbeiro);
```

- Como avisar o barbeiro que você está esperando?

Iniciando o corte

```
semaforo cadeiras = 5;  
semaforo cad_barbeiro = 1;  
semaforo cliente_cadeira = 0;
```

```
if (trywait(cadeiras) == 0)  
    wait(cad_barbeiro);  
    signal(cliente_cadeira);
```

- E os outros clientes?

Cortando o cabelo

```
semaforo cadeiras = 5;  
semaforo cad_barbeiro = 1;  
semaforo cliente_cadeira = 0;
```

```
if (trywait(cadeiras) == 0)  
    wait(cad_barbeiro);  
    signal(cadeiras);  
    signal(cliente_cadeira);
```

- Quem decide que o corte acabou?

Cliente

```
if (trywait(cadeiras) == 0)  
    wait(cad_barbeiro);  
    signal(cadeiras);  
    signal(cliente_cadeira);  
    wait(cabelo_cortado);  
    signal(cad_barbeiro);
```

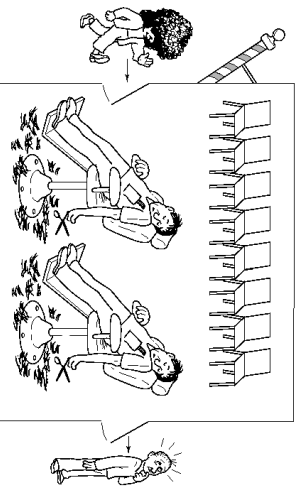
Barbeiro

```
semaforo cabelo_cortado = 0;  
semaforo cliente_cadeira = 0;
```

```
while (true)  
    wait(cliente_cadeira);  
    corta_cabelo();  
    signal(cabelo_cortado);
```

- Veja o código barbeiro.c
- Como implementar um cineminha?

Múltiplos Barbeiros Dorminhocos



Múltiplos Barbeiros Dorminhocos

- Vários semáforos semelhantes ao problema anterior

```
semaforo cadeiras = N_CADEIRAS_ESPERA;  
semaforo cad_barbeiro[N_BARBEIROS] =  
    {0, 0, 0, ..., 0};  
semaforo cabelo_cortado[N_BARBEIROS] =  
    {0, 0, 0, ..., 0};  
semaforo cliente_cadeira[N_BARBEIROS] =  
    {0, 0, 0, ..., 0};
```

Múltiplos Barbeiros Dorminhocos

- O cliente precisa saber qual é o identificador do barbeiro disponível.
- Problema análogo a fila única em bancos com visor para chamar os clientes.

Modelando o visor

- Variável para armazenar identificadores:
`int visor;`
- Barbeiros executam escritas
 - Um barbeiro só pode escrever se o barbeiro anterior já atendeu um cliente;
- Clientes executam leituras
 - Apenas um cliente pode ser atendido de cada vez.

Múltiplos Barbeiros Dorminhocos

- O cliente precisa saber qual é o identificador do barbeiro disponível.
- Problema análogo a fila única em bancos com visor para chamar os clientes.
`semaforo escreve_visor = 1;`
`semaforo le_visor = 0;`
`int visor;`

Barbeiro

```
while (true)
    wait(escreve_visor);
    visor = id_barbeiro;
    signal(le_visor);
    wait(cliente_cadeira[id_barbeiro]);
    corta_cabelo();
    signal(cabelo_cortado[id_barbeiro]);
```

Cliente

```
if (trywait(cadeiras) == 0)
    signal(cadeiras);
    wait(le_visor);
    minha_cadeira = visor;
    signal(escreve_visor);
    wait(cad_barbeiro[minha_cadeira]);
    signal(cliente_cadeira[minha_cadeira]);
    wait(cabelo_cortado[minha_cadeira]);
    signal(cad_barbeiro[minha_cadeira]);
```

- Podemos eliminar `cad_barbeiro`?

Múltiplos Barbeiros Dorminhocos

- Como implementar um cineminha?
- Como você implementaria este problema utilizando locks e variáveis de condição?

Leitores e escritores

```
semaforo sem_dados = 1;
Leitor:
    while(true)
        wait(sem_dados);
        le_dados();
        signal(sem_dados);
Escritor:
    while(true)
        wait(sem_dados);
        escreve_dados();
        signal(sem_dados);
```

Leitores e escritores

- Problema: apenas um leitor pode fazer acesso ao banco de dados por vez
- Veja o código: l-e-sem-concorrenda.c
- Possível solução: permitir o acesso simultâneo a vários leitores

Vários leitores simultâneos

```
semaphore sem_dados = 1, sem_nl = 1;  
int nl; /* Leitores ativos num dado instante */
```

Leitor:

```
while(true)  
    wait(sem_nl);  
    nl++; if (nl == 1) wait(sem_dados);  
    signal(sem_nl);  
    le_dados();  
    wait(sem_nl);  
    nl--; if (nl == 0) signal(sem_dados);  
    signal(sem_nl);
```

Vários leitores simultâneos

- Problema: os escritores podem morrer de fome
- Veja o código: l-e-starvation.c
- Como escrever este código usando locks e variáveis de condição?
- Como implementar variações desta abordagem?

Leitor

```
mutex_lock(&lock_nl);  
nl++;  
if (nl == 1) mutex_lock(&lock_dados);  
mutex_unlock(&lock_nl);  
le_dados();  
mutex_lock(&lock_nl);  
nl--;  
if (nl == 0)  
    mutex_unlock(&lock_dados);  
mutex_unlock(&lock_nl);
```

Leitores simultâneos

- Problema: Uma thread leitora faz o lock e outra faz o unlock
- Veja o código: l-e-lock.c
- Tipos de lock:
 - FAST
 - RECURSIVE
 - ERROR CHECKING

Leitores simultâneos

Locks e variáveis de condição

Segunda tentativa

```
mutex_t lock_dados; /* Controle dos dados */  
boolean bloq_leitura = false;  
  
mutex_t lock_nl; /* Lock para o contador */  
int nl = 0; /* Número de leitores ativos */
```

Leitores simultâneos

Locks e variáveis de condição

Primeira tentativa

```
int nl = 0; /* Número de leitores ativos */  
mutex_t lock_nl; /* Lock para o contador nl */  
  
mutex_t lock_dados; /* Lock para os dados */
```

Leitor

```
mutex_lock(&lock_nl);  
nl++;  
if (nl == 1)  
    mutex_lock(&lock_dados);  
    bloq_leitura = true;  
    mutex_unlock(&lock_dados);  
    mutex_unlock(&lock_nl);  
le_dados();  
/* ... */
```

Leitor

```
/* ... */
le_dados();
mutex_lock(&lock_n1);
n1--;
if (n1 == 0)
    mutex_lock(&lock_dados);
    bloq_leitura = false;
    cond_signal(&cond_dados);
    mutex_unlock(&lock_dados);
mutex_unlock(&lock_n1);
```

Escritor

```
mutex_lock(&lock_dados);
while (bloq_leitura)
    cond_wait(&cond_dados, &lock_dados);
escreve_dados();
cond_signal(&cond_dados);
mutex_unlock(&lock_dados);
```

Leitores simultâneos Locks e variáveis de condição Terceira tentativa

```
cond_t cond_dados; /* Espera pelos dados */

mutex_t lock_cont; /* Lock para os contadores */
int n1 = 0; /* Número de leitores ativos */
int ne = 0; /* Número de escritores ativos */
```

Veja o código: l-e-broadcast.c

Escritor

```
mutex_lock(&lock_cont);
while (n1 > 0 || ne > 0)
    cond_wait(&cond_dados, &lock_cont);
ne++;
mutex_unlock(&lock_cont);
escreve_dados();
mutex_lock(&lock_cont);
ne--;
cond_broadcast(&cond_dados);
mutex_unlock(&lock_cont);
```

Leitor

```
mutex_lock(&lock_cont);
while (ne > 0 || nw > 0)
    cond_wait(&cond_leit, &lock_cont);
n1++;
mutex_unlock(&lock_cont);
/* Leitura */
mutex_lock(&lock_cont);
n1--;
if (n1 == 0 && nw > 0)
    cond_signal(&cond_esc);
mutex_unlock(&lock_cont);
```

Leitor

```
mutex_lock(&lock_cont);
while (ne > 0)
    cond_wait(&cond_dados, &lock_cont);
n1++;
mutex_unlock(&lock_cont);
le_dados();
mutex_lock(&lock_cont);
n1--;
if (n1 == 0)
    cond_signal(&cond_dados);
mutex_unlock(&lock_cont);
```

Leitores e escritores Prioridade para os escritores

```
int n1 = 0; /* Número de leitores */
int ne = 0; /* Número de escritores */
int nw = 0; /* Número de escritores esperando */
mutex_t lock_cont;
cond_t cond_esc, cond_leit;
```

Veja o código: l-e-broadcast2.c

Escritor

```
mutex_lock(&lock_cont);
nw++;
while (n1 > 0 || ne > 0)
    cond_wait(&cond_esc, &lock_cont);
nw--; ne++;
mutex_unlock(&lock_cont);
/* Escrita */
mutex_lock(&lock_cont);
ne--;
if (nw > 0)
    cond_signal(&cond_esc);
else
    cond_broadcast(&cond_leit);
mutex_unlock(&lock_cont);
```

Leitores e escritores

Como implementar um bom compromisso?

- Ausência de starvation
- Leitores simultâneos

Leitores e escritores

RWLock

- pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
- pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
- pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
- Qual é a política implementada?

MC514–Sistemas Operacionais: Teoria e Prática

1s2008

Processos: fork

fork()

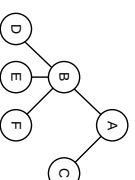
- Cria um novo processo, que executará o mesmo código
- Retorna
PID do processo criado para o pai e
0 para o processo filho

Espaços de endereçamento distintos

```
if (fork() == 0)
    s = 0;
    printf("Filho: %s=%p s=%d\n", &s, s);
} else {
    s = 1;
    printf("Pai: %s=%p s=%d\n", &s, s);
}
```

- Veja o código: fork0.c

Hierarquia de processos



Como implementar uma arquitetura como esta utilizando a chamada fork?

- Veja os códigos: fork1.c fork2.c fork3.c

wait()

```
pid_t wait(int *status);
```

- Aguarda pela morte de um filho.
- Bloqueia o processamento
- Retorna o pid do filho morto
- status indica causa da morte
- Veja os códigos: wait1.c, wait2.c e getppid.c

waitpid()

```
pid_t waitpid(pid_t pid, int *status,
               int options);
```

- Aguarda pela morte de um filho.
 - Específico pid = PID
 - Qualquer pid = -1
- Versão não bloqueante (options = WNOHANG)
- Veja o código waitpid1.c