

# MC504/MC514 - Sistemas Operacionais

## Processos e Threads: Condição de Corrida

Islene Calciolari Garcia

Primeiro Semestre de 2016

# Sumário

- 1 Revisão create e join
- 2 Pilhas de execução
- 3 Condição de corrida
- 4 Exclusão mútua

# 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`

# GDB e threads

- `info threads`
- `thread <thread number>`
- `set non-stop on`

In non-stop mode, when one thread stops, other threads can continue to run freely. You'll be able to step each thread independently, leave it stopped or free to run as needed.

- `interrupt`  
Interrupt the execution of the debugged program. If non-stop mode is enabled, interrupt only the current thread, otherwise all the threads in the program are stopped. To interrupt all running threads in non-stop mode, use the `-a` option.

# Pilha de execução

Códigos desse tipo podem causar erro de execução?

```
void f() {  
    int v[5];  
    int i;  
  
    for (i = 0; i < 10; i++)  
        v[i] = i;  
}
```

- Veja o código pilha.c

# Vamos examinar a pilha de execução

- Componentes do frame (não necessariamente nesta ordem):
  - valor de retorno
  - endereço de retorno
  - registradores
  - argumentos para a função
  - variáveis locais da função
- Veja os códigos `pilha2.c` e `pilha3.c`
- No gdb execute comandos do tipo:

```
(gdb) p (int[10]) *v
```

```
(gdb) p (int[10]) *(v-2)
```

```
(gdb) set *(v-2) = ...
```

# 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

- (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

```
volatile 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

```
volatile int s; /* 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`

# 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 */
}
```

# Exclusão Mútua

- Os algoritmo devem 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)

# 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

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

- Veja o código: tentativa\_lock.c

## Thread 1

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

# 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

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

- Veja o código: `alternancia.c`

## Thread 1

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



# 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

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

- Veja o código: interesse.c

## Thread 1

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

# 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`



# 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;
```

# 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 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;
```

# 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;
```