NOVA FACULDADE DE CIÊNCIAS E TECNOLOGIA DEPARTAMENTO DE INFORMÁTICA

Fundamentos de Sistemas de Operação

LEI - 2023/2024

Vitor Duarte
Mª. Cecília Gomes

1

Aula 8

- Regiões críticas
- Exclusão mútua

- OSTEP: cap. 26, 27, 28.1, 28.2

2

Ações concorrentes

- Threads/processos concorrentes: dados dois threads P e Q:
  - uma vez iniciado P, pode-se iniciar Q, sem ter de esperar pelo fim de P, ou vice-versa.
  - A execução concorrente de P com Q define um conjunto de **várias sequências possíveis**. Exemplo considerando apenas um CPU:

P: { **p1; p2**} e Q: {**q1; q2**}

p1; p2; q1; q2 ← equivale a P; Q

p1; q1; p2; q2

p1; q1; q2; p2

q1; q2; p1; p2 ← equivale a Q; P

q1; p1; q2; p2

q1; p1; p2; q2

Correção de um programa concorrente

- Programa concorrente: especifica múltiplos processos ou threads concorrentes
- A sua execução tem de produzir resultados corretos em TODAS as execuções possíveis.
  - Fácil: se threads INDEPENDENTES.
  - Difícil: se os threads DEPENDEM uns dos outros
    - Comunicam ou sincronizam-se
  - *Ao repetir a execução de um MESMO programa com os MESMOS dados, os resultados podem ser DIFERENTES?*
  - *Se nunca vir um resultado diferente não significa que o programa esteja correto!*

Regiões Críticas

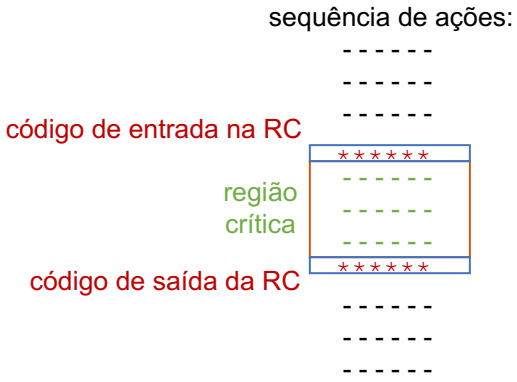
- Regiões Críticas: Regiões de código que envolvem recursos partilhados executadas concorrentemente
- É necessário sincronizar as ações concorrentes que interferem
  - Impor uma ordem ou impedir certas ordens
- Recorre-se à programação de protocolos de acesso à região crítica
  - Recorrendo ao auxílio de entidades externas (p.e. SO)
  - Recorrendo a algoritmos e/ou instruções específicas
  - Exemplo: optar pela exclusão mútua no uso do recurso, criando operações indivisíveis ou atómicas

NOVA FACULDADE DE CIÊNCIAS E TECNOLOGIA DEPARTAMENTO DE INFORMÁTICA

5

Exclusão Mútua

- Se o código de entrada e de saída da RC garantirem que só um processo/thread está na RC, deixa de existir concorrência na RC

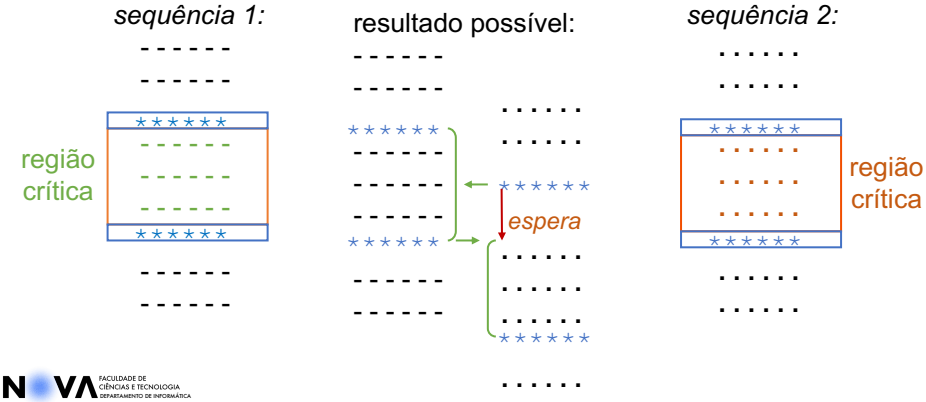


NOVA FACULDADE DE CIÊNCIAS E TECNOLOGIA DEPARTAMENTO DE INFORMÁTICA

6

Exclusão Mútua (exemplo)

- O objectivo é que cada região crítica seja indivisível ou seja, não haja reentradas na região
- Obriga a ordem entre RCs, não ocorrendo data races (race conditions)



Exemplo de protocolo

- Inspirado nos semáforos de trânsito:
  - Variável com dois estados que indica se se pode entrar ou não
  1. cada thread aguarda até semáforo estar verde;
  2. depois muda semáforo para vermelho e entra na RC;
  3. à saída muda semáforo para verde

- Implementação de semáforo por software? Será que funciona?

```
bool ocupado = FALSE;

entrar() {
  while (ocupado)
     ; // espera
  ocupado = TRUE;
}

sair() {
  ocupado = FALSE;
}
```

- Não! Estas funções são RC. Só se estas funções fossem atómicas!

Uma solução: Mutex (ou lock)

- Mecanismo para exclusão mútua entre *threads* do mesmo processo
- Variáveis do tipo de dados **`pthread_mutex_t`**
  - Possui dois estados: *fechado* ou *aberto*
- Operações:

```
pthread_mutex_init( *mutex, attr )
```
inicia, sempre no estado aberto

```
pthread_mutex_destroy( *mutex )
```

```
pthread_mutex_lock( *mutex )   //fechar
pthread_mutex_unlock( *mutex )   //abrir
```
para os threads estas funções são atómicas

Uso de um mutex

- Garantir o acesso exclusivo a uma região crítica:

```
pthread_mutex_t exmut;
pthread_mutex_init( &exmut, NULL);
```

  - em cada thread:

```
pthread_mutex_lock( &exmut );
    Região Crítica…
pthread_mutex_unlock( &exmut );
```

  - Como garantir a utilização correta?
    - **O programador** certifica-se que coloca os locks/unlocks corretos

Corrigindo o exemplo do contador

```
int max;
int counter=0;
pthread_mutex_t rc = PTHREAD_MUTEX_INITIALIZER;
      // eq. a pthread_mutex_init(&rc, NULL);

void * mythread(void *arg){
 for (int i=0; i<max; i++){
   pthread_mutex_lock(&rc);
   counter = counter+1;
   pthread_mutex_unlock(&rc);
 }
 return NULL;
}
```

```
void * mythread(void *arg){
 int c = 0;
 for (int i=0; i<max; i++)
     c = c+1;
 pthread_mutex_lock(&rc);
 counter = counter + c;
 pthread_mutex_unlock(&rc);
 return NULL;
}                  // melhor!
```


Funções reentrantes e thread-safe

- Sempre que uma função não possa ser re-chamada antes de terminar uma chamada em curso, diz-se:
  - NÃO REENTRANTE – não é seguro reentrar a meio de outra chamada, por diferentes threads, por um *handler* de um sinal ou por recursividade
  - problema típico quando se manipulam variáveis estáticas (como var. globais) ou variáveis partilhadas entre "chamadas"
    - Como a mythread original
- Propriedades das funções na terminologia POSIX:
  - MT-Safe (MultiThread-safe) e AS-Safe (AsyncSignal-safe)
  - nota: uma função pode ser tornada *thread-safe* com um lock à entrada, serializando o acesso por multiplos *threads*
- As funções das bibliotecas são reentrantes?
- Os métodos das classes *standard* do Java são reentrantes?

Exemplos libc

- Algumas funções podem existir com duas implementações:
  - normal vs *multi-thread*
- Versão *multi-thread* só é usada se compilarmos definindo **`_REENTRANT`** (compilando com **`-pthread`**).
  - as chamadas ao SO passam a usar um errno local a cada *thread*
- Outras funções existem em diferentes versões com outras interfaces. Exemplos:

```
char *strtok(char *str, char *delim)
char *strtok_r(char *str, char *delim, char *ptr)

int rand()
int rand_r(unsigned *seed)
```


Escrever funções Thread-safe

- Funções puras: só dependem dos argumentos e não têm efeitos colaterais
  - não alteram vars. globais ou static, parâmetros ou outros recursos;
  - só chamam funções reentrantes
- Usam algoritmos que garantem não haver sequências inválidas (sem data races)
- Usam mecanismos de controlo da concorrência para evitar sequências inválidas (ex. *mutex* ou outros)
  - garantir que as variáveis partilhadas são acedidas de forma controlada
  - ou garantir **serialização** das chamadas à função (só existe uma instância da função em execução)

Exemplo: implementando fputs

```
/* not thread-safe */
fputs(char *s, FILE *stream){
    char *p;
    for (p=s; *p; p++)
        putc((int)*p, stream);
}
```

só para programas sequênciais

para programas multi-thread mas pouco eficiente

para programas multi-thread e permitindo concorrência para ficheiros diferentes

```
/* serializable */
fputs(char *s, FILE *stream){
    static mutex_t m=PTHREAD_MUTEX_INICIAL
    mutex_lock(&m);
    char *p;
    for (p=s; *p; p++)
        putc((int)*p, stream);
    mutex_unlock(&m);
}

/* MT-Safe */
mutex_t m[NFILE];  //per file desc

fputs(char *s, FILE *stream){
    char *p;
    mutex_lock(&m[fileno(stream)]);
    for (p=s; *p; p++)
        putc((int)*p, stream);
    mutex_unlock(&m[fileno(stream)]);
}
```

Exemplo de:
Multithreaded Programming Guide, Oracle/Sun Solaris

Correção dos programas

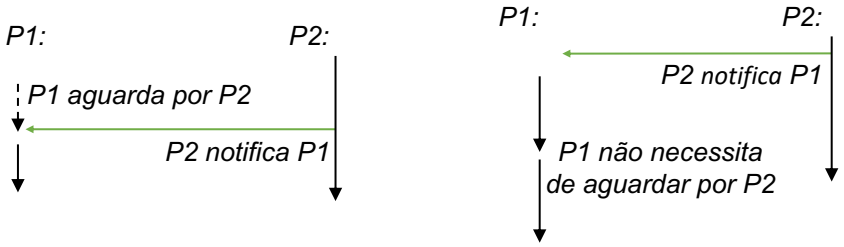
- A correção dum programa multi-threaded não depende só de usar funções thread-safe
- Sequências de chamadas de funções thread-safe podem não ser thread-safe e levar na mesma a erros
- Exemplo: assuma *List* com implementação de *contains()* e *add() thread-safe*
  - queremos implementar inserção sem repetições (como num Set)

```
insertUniq(L, x) {
    if ( ! L.contains(x) )
        L.add(x);
}
```

  - problemas?

Outras interações: sincronização

- Threads necessitam de aguardar por determinada condição
  - exemplo:
  - aguardar que uma variável partilhada seja alterada
  - avisar um thread ou um de entre vários



Tentativa de solução

Inicio: notificado = FALSE

P1:

. . .

while (!notificado)

;

notificado = FALSE

. . .

P2:

. . .

notificado = TRUE

. . .

Grande desperdício de CPU!

Não funciona para todos os casos

Tentativa de solução (2)

```
Inicio:  notificado = FALSE
         mutex  // já locked
P1:
. . .
if ( !notificado)
    lock(mutex)
notificado = FALSE
. . .
```

```
P2:
. . .
notificado = TRUE
unlock(mutex)
 . . .
```

Não funciona!

Tentativa de solução (2 - b)

```
Inicio:
         mutex  // já locked
```

```
P1:
. . .
lock(mutex)
. . .
```

```
P2:
. . .
unlock(mutex)
 . . .
```

Quando não há condição a testar.

Só funciona uma vez.

Variáveis de condição

- Permitem controlar o teste concorrente de condições e aguardar
- Usada junto com um *mutex* para exclusão mútua enquanto se testa a condição e para se bloquear
  - Início:

```
pthread_cond_t cond;
pthread_cond_init( &cond, NULL);
pthread_mutex_t mutex;
pthread_mutex_init( &mutex, NULL);
```

Bloquear aguardando alteração da condição:

```
pthread_cond_wait(&cond, &mutex)
```

liberta mutex

Desbloquear um dos bloqueados na condição ou todos:

```
pthread_cond_signal(&cond)
pthread_cond_broadcast(&cond)
```


Exemplo de uso

```
pthread_mutex_lock(&lock);
while (!notificado)  // condição a testar
   pthread_cond_wait(&cond, &lock);
notificado = FALSE;
pthread_mutex_unlock(&lock);
```

```
pthread_mutex_lock(&lock);
notificado = TRUE;
pthread_cond_signal(&cond);
pthread_mutex_unlock(&lock);
```