

# *Fundamentos de Sistemas de Operação*

Unix Windows NT Netware Mac OS DOS/V/S Vax/VMS  
Linux Solaris HP/UX AIX Mach Chorus

*Serviços do SO:  
Como são implementados os Locks*

# *Locks: algumas interrogações...*

## □ *Locks como primitivas de nível utilizador?*

- *Premissas:*
  - Sistema com um só CPU
  - Escalonador com preempção
- *É possível implementar locks sem recorrer ao SO?*
- *E a solução será aceitável, i.e., garante quando usada*
  - exclusão mútua?
  - justeza (logo, não há hipótese de starvation)?
  - desempenho?

# *Locks: implementação naive* (1)

## □ Locks como primitivas de nível utilizador: tentativa 1

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    mutex->flag = 0;                                // 0, lock is available, 1, is held
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)                      // TEST the flag
        ; // spin-wait
    mutex->flag = 1;                                // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

# *Locks: implementação naïve* (2)

- Traço de execução da tentativa 1 (excl. mútua falhada)

## Thread 1

```
call lock()  
while (flag == 1)  
interrupt: switch to Thread 2
```

```
flag = 1; // set flag to 1 (too!)
```

## Thread 2

```
call lock()  
while (flag == 1)  
flag = 1;  
interrupt: switch to Thread 1
```

- No traço acima, a exclusão mútua não funciona: as duas threads acabam por avançar com a flag a 1 (de OSTEP)

# *Locks: implementação CLI/STI (1)*

## □ Locks como primitivas de nível utilizador: tentativa 2

- Como deve recordar (AC ☺)
  - CLI é a instrução Intel que desactiva a recepção de interrupções
  - e STI volta a activar...
- Uma possível implementação seria:

```
void lock() {  
    asm("CLI"); // invocar a instrução assembly CLI  
}  
  
void unlock() {  
    asm("STI"); // invocar a instrução assembly STI  
}
```

# *Locks: implementação CLI/STI (2)*

## □ Locks usando CLI/STI: crítica

- *Funcionaria, garantiria a exclusão mútua, mas:*
  - Só mesmo em sistemas com um só CPU (porquê? ☺)
  - Se um programa executasse o CLI e “nunca mais” fizesse STI
    - Nunca mais seria possível libertar o CPU
    - Punha em risco o funcionamento do sistema (CPU deixava de atender interrupções)
- *E, pelas razões acima, CLI e STI são instruções privilegiadas*
- *No fundo, nem vale a pena analisar a “justeza” e “performance”*

# *Locks: instruções especializadas do CPU*

## □ Locks recorrendo a **instruções especializadas** do CPU

- Os fabricantes de CPUs incluiram instruções especializadas que se mostram essenciais para suporte de operações específicas. Uma destas é designada por “test and set” ou “compare and exchange”; no Intel, **xCHG**.
- Em “pseudo-C” a seguinte função descreve a semântica da instrução, que é executada **atomicamente!** (num só ciclo)

```
int TestAndSet(int *old, int newValue) {  
    int oldValue= *old;  
    *old= newValue;  
    return oldValue;  
}
```

# *Locks: implementação T&S (1)*

- Locks recorrendo à instrução “test-and-set”

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    mutex->flag = 0;                                // 0, lock is available, 1, held
}

void lock(lock_t *mutex) {
    while (TestAndSet(mutex->flag, 1) == 1)
        ; // spin-wait
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

# *Locks: implementação T&S (2)*

- *Funcionamento do lock com “test-and-set”*
- *Como só a função lock é diferente do caso anterior, (slide 4) basta compreender como esta funciona...*
  - Se o mutex está *livre (unlocked)*, então *flag == 0*
  - *Uma thread que tente trancar o mutex, vai executar a função TestAndSet, que vai retornar 0 (valor “antigo” da flag) ao mesmo tempo que muda a flag para 1 – logo não vai permanecer no while (o retorno foi zero!) e conseguiu trancar o lock.*
  - Se o mutex estiver já trancado, a função *TestAndSet*, vai retornar 1 (valor “antigo” da flag) ao mesmo tempo que muda a flag para 1 – logo vai permanecer no while (o retorno foi um!) em *busy-waiting*.

# *Locks: implementação T&S (3)*

## □ Locks usando T&S: crítica

- Exclusão mútua: funciona, garante
- Justeza: não garante
  - Se houver apenas duas threads, e uma trancar o lock, a outra não consegue trancá-lo e fica em spin... mas se a primeira destrancar o lock, para logo a seguir o trancar de novo, e isso acontecer durante o seu timeslice, então a outra não consegue “entrar”. Será preciso “muito azar”, mas este padrão pode repetir-se indefinidamente...
  - Já se houver mais CPUs, não acontecerá (quase certamente... ☺)
- Desempenho: pode ser mau
  - Pense ☺

# *Locks: outras instruções especializadas*

- Os fabricantes (os seus projectistas!) introduziram outras instruções que podem ser usadas para o mesmo efeito
  - Compare-and-Swap: processadores Sparc (Sun/Oracle) e Intel (onde é chamada Compare-and-Exchange)
  - Load-Linked e Store-Conditional: é necessário usar as duas para implementar a EM. Existe nos processadores PowerPC (IBM) e ARM (Intel)
  - Fetch-And-Add: processadores Intel

*Nota: apenas por razões de completude; não é importante para o estudo ☺*

# *Locks sem spin-waiting*

- *Do exposto, concluímos que*
  - É possível implementar locks usando instruções “especializadas” dos CPUs, que não são privilegiadas – ou seja, locks de nível utilizador, que não requerem intervenção do SO
- *Será que conseguimos uma solução sem spinning?*
  - E será que esta requer ajuda do SO?

# *Locks sem spin-waiting: solução naive* (1)

- Adicionar um pedido para sermos re-escalonados...

```
void lock(lock_t *mutex) {  
    while (TestAndSet(mutex->flag, 1) == 1)  
        sched_yield(); // tirem-nos do CPU ☺  
}
```

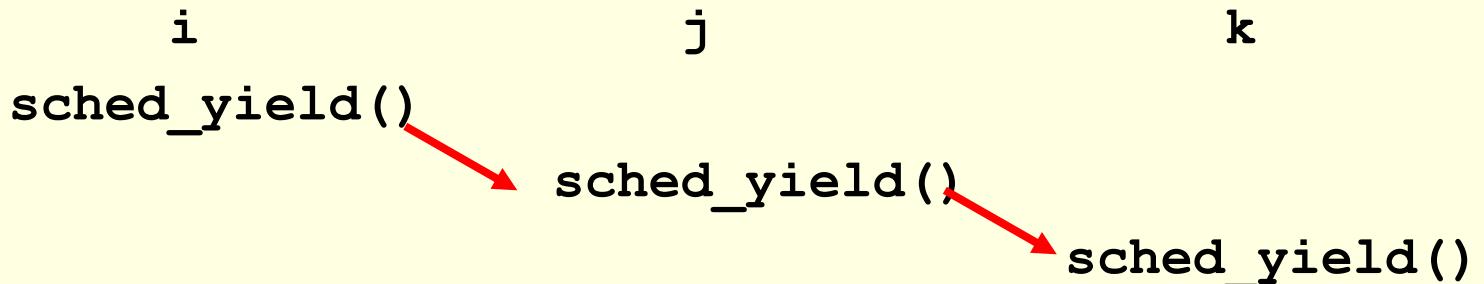
- *sched\_yield()* [POSIX standard]

- Pede ao escalonador para a entidade (*thread*, *processo*) ser retirada do estado “execução” (*running*) e passada para “pronto” (*ready*)
- Algumas implementações de *Pthreads* têm uma função de biblioteca, *pthread\_yield()* [*não standard*]

# *Locks sem spin-waiting: solução naive (2)*

## □ A solução é satisfatória?

- À primeira vista... sim, porque acabou-se o busy-waiting
- Mas... imagine-se várias (muitas) threads à espera de um lock: cada uma, quando recebe CPU, larga-o imediatamente via `sched_yield()`. Sejam  $i, j, k$  threads à espera:



- As repetidas (e muito rápidas) chamadas a `sched_yield()` acarretam **context switches** (entre “user” e “kernel”), o que representa um grande overhead...

# *Locks sem spin-waiting: solução naïve* (3)

## □ A solução é satisfatória?

(continuação)

- *Finalmente, a solução, tal como as anteriores, não é isenta de “starvation” ainda que, como as anteriores, esta seja pouco provável*

# *Locks sem spin: solução com o SO* (1)

- *Locks com filas de espera e ajuda do SO*
  - Para conseguir uma solução justa (sem starvation) e sem busy-waiting vamos buscar suporte ao SO, na forma de funções para adormecer e acordar threads (usamos como exemplo as funções **park()** e **unpark()** disponíveis no SO Solaris (Sun/Oracle), e uma fila de espera
- **park()**
  - Coloca a thread que a invoca no estado de espera (wait)
- **unpark(tid x)**
  - Coloca a thread com ID x no estado running

# *Locks sem spin: solução com o SO (2)*

## □ Estrutura de dados e inicialização

```
typedef struct __lock_t {  
    int flag;  
    int guard;  
    queue_t *q;  
} lock_t;  
  
void lock_init(lock_t *m) {  
    m->flag = 0;  
    m->guard = 0;  
    queue_init(m->q);  
}
```

- A **flag** é usada para guardar o estado locked (1) e unlocked (0)
- A **guard** é usada para garantir EM no acesso à fila **q**

# *Locks sem spin: solução com o SO (3)*

## □ Função lock

```
void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning

    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }

}
```

# *Locks sem spin: solução com o SO* (4)

## □ Algoritmo de lock

- Adquirir um trinco sobre **guard** para operar na fila em EM
- Se o *lock* está *livre* (**flag == 0**) mudar para *trancado* e libertar a guarda (**guard= 0**)
- Senão, guardar o *ID* da *thread* numa nova entrada na fila, libertar a guarda (**guard= 0**) e adormecer a *thread*

# *Locks sem spin: solução com o SO (5)*

## □ Função unlock

```
void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; // adquirir a guarda de EM via spinning

    if (queue_empty(m->q))
        m->flag = 0;                                // ninguém quer o lock, libertar
    else
        unpark(queue_remove(m->q));   // manter o lock, pode haver mais threads interessadas

    m->guard = 0;                                  // libertar a guarda de EM
}
```

- Os comentários são suficientes para compreender o algoritmo ☺