



MAC422 Sistemas Operacionais

Prof. Marcel P. Jackowski

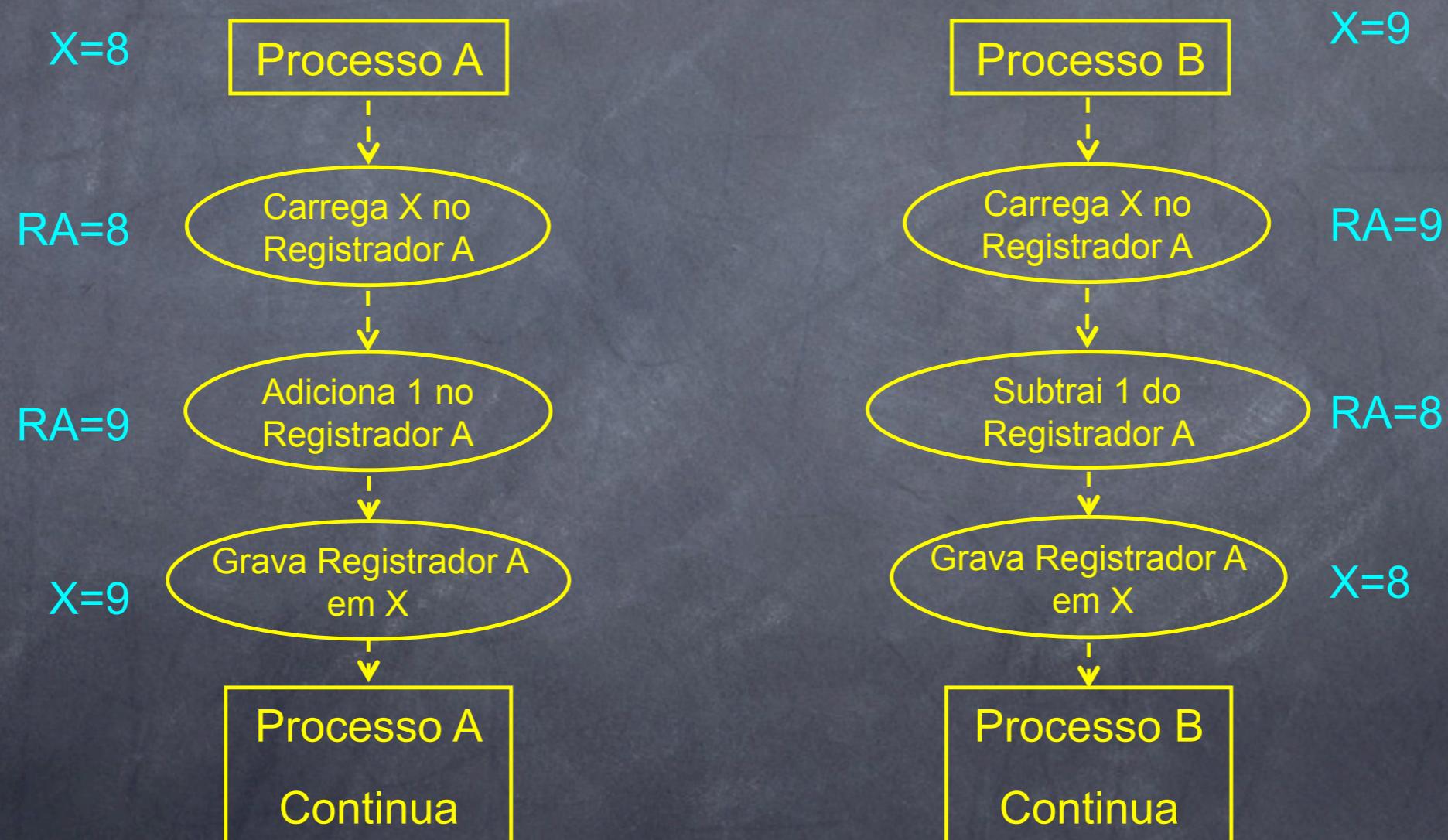
Aula #9

Introdução à sincronização

Sincronização entre produtor e consumidor

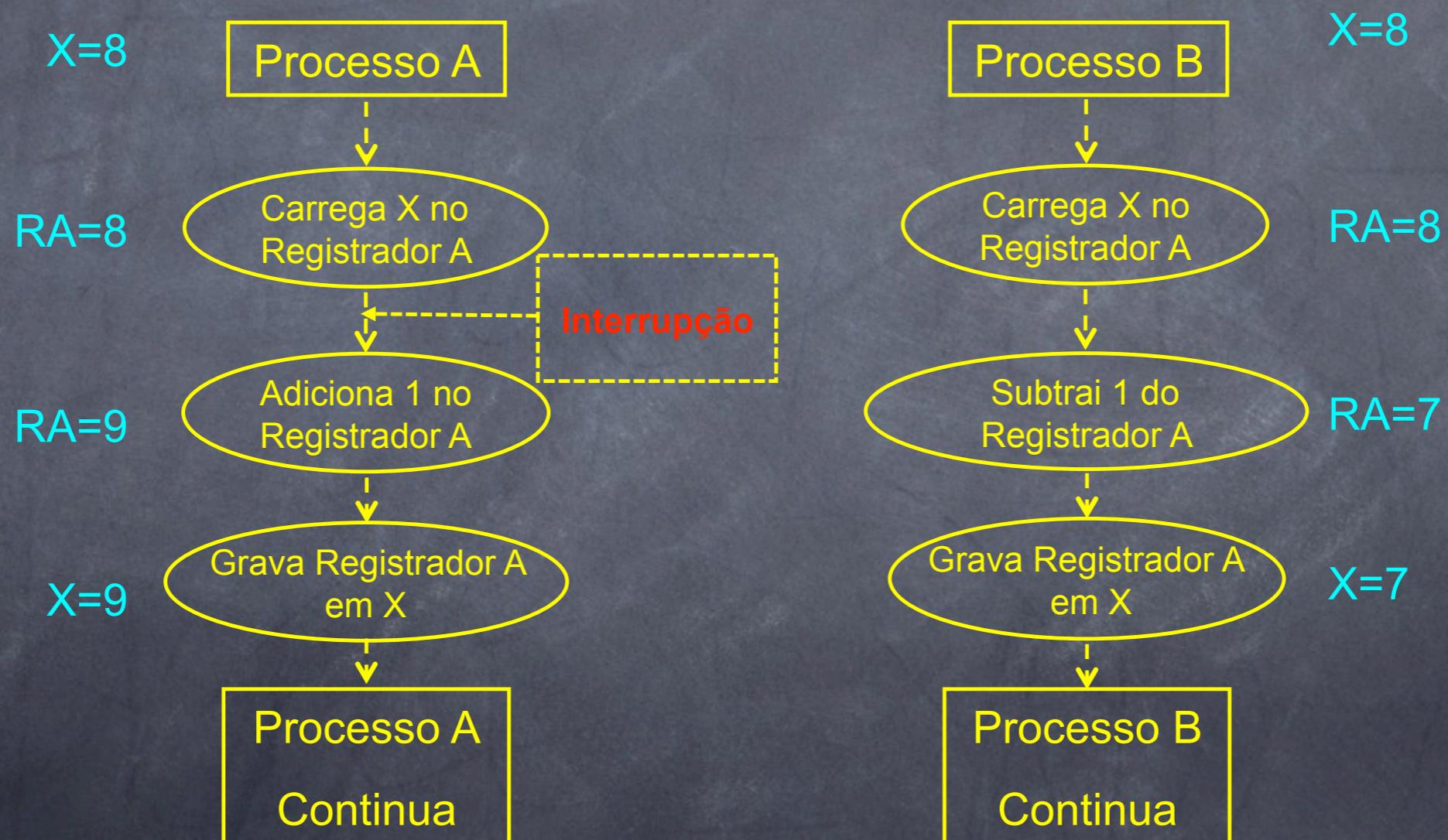
- Dois processos (ou threads), P_A e P_B , utilizam um buffer compartilhado para se comunicar da seguinte forma:
 - P_A insere um registro no buffer
 - Incrementa o contador: $X \leftarrow X + 1$
 - P_B retira um registro do buffer
 - Decrementa o contador: $X \leftarrow X - 1$
- Suponha que já existam 8 registros neste buffer
- Qual o valor final de X ?

Sincronização



$P1;P2$

Possíveis escalonamentos



P1;<INT>;P2;P1

“Race condition”

- Também chamada de **condição de disputa**, ela é um erro de “timing” envolvendo um estado compartilhado:
 - O resultado é dependente da sequência das instruções de máquina
 - Não-determinístico
 - Difícil de detectar e depurar
- Tais operações devem ser tratadas de forma *atômica*
- **Região crítica:** um segmento de código que acessa uma variável (ou recurso) compartilhada que não deve ser executado concorrentemente por mais que um thread.

Atomicidade

- Garantia de atomicidade em operações:
 - Instruções que lêem e gravam palavras inteiras de forma alinhada
 - e.g. endereço de um *int* deve ser múltiplo de 4 bytes
 - Instruções *inc* e *dec*
 - $a=a+1$, $a++$, $++a$, $a=a-1$, $a--$, $--a$
 - Prefixo *lock*
 - “Trava” o barramento de memória até o término da próxima instrução

Níveis de sincronização

- Primitivas para sincronização de baixo nível
 - Em máquinas monoprocessadas, pode-se habilitar e desabilitar interrupções: *cli* e *sti*
 - Leitura e gravação de palavras inteiras alinhadas
 - Instruções especiais: *test-and-set*, *compare-and-swap*
- Primitivas de sincronização de alto nível
 - Travas (“Locks”)
 - Semáforos (“Semaphores”)
 - Monitores

Seções críticas

```
void main()
{
    for(;;) {
        /* início das operações */
        seção_inicial();

        /* verifica se existe outro thread executando na SC */
        seção_de_entrada();

        /* altera dados compartilhados */
        seção_crítica();

        /* sinaliza à outros threads que estamos prestes a deixar
           a SC */
        seção_de_saída();

        /* restante das instruções */
        seção_final();
    }
}
```

Propriedades

- **Segurança** (“Safety”, exclusão mútua):
 - Não mais que um thread por vez executando na seção crítica
- **Vivacidade** (“Liveness”, progresso):
 - Se múltiplos threads simultaneamente pedem para entrar na seção crítica, um deles deve conseguir
 - Esta decisão não deve envolver threads fora da seção crítica
- **Espera limitada** (“starvation-free”)
 - Eventualmente, threads em espera devem conseguir entrar na seção crítica
- Não depender da velocidade e/ou número de CPUs

Propriedades desejáveis

- **Eficiência:** não consumam muitos recursos durante a espera
 - Sem espera ociosa (“busy wait” ou “spin wait”)
 - Preferível largar a CPU e deixar outros threads ou processos rodarem
- **Justiça:** não deixar threads esperararem mais que outros
- **Simplicidade:** fáceis de usar.

Travas (“locks”)

```
void main()
{
    for(;;) {

        /* adquire trava de forma exclusiva, espera se não
         disponível */
        lock();

        /* altera dados compartilhados */
        seção_crítica();

        /* libera acesso exclusivo à trava */
        unlock();

        /* restante das instruções */
        seção_final();
    }
}
```

i.e. variáveis de impedimento

Exemplo

```
pthread_mutex_t l = PTHREAD_MUTEX_INITIALIZER

void* deposit(void *arg)
{
    int i;
    for(i=0; i<1e7; ++i) {
        pthread_mutex_lock(&l);
        ++ balance;
        pthread_mutex_unlock(&l);
    }
}

void* withdraw(void *arg)
{
    int i;
    for(i=0; i<1e7; ++i) {
        pthread_mutex_lock(&l);
        -- balance;
        pthread_mutex_unlock(&l);
    }
}
```

Como implementar as primitivas
lock e unlock ?

Desabilitar e habilitar interrupções

```
lock()  
{  
    disable_interrupt();  
}  
  
unlock()  
{  
    enable_interrupt();  
}
```

- Somente utilizado em sistemas com uma única CPU
- Simples, porém:
 - São comandos privilegiados (ring 0)
 - Não funcionam em sistemas com vários cores ou CPUs.

Implementação via software (i)

```
// 0: lock is available, 1: lock is held by a thread
int flag = 0;

lock()
{
    while (flag == 1)
        ; // spin wait
    flag = 1;
}

unlock()
{
    flag = 0;
}
```

- Idea: use one flag, test then set; if unavailable, spin-wait (or busy-wait)
- Problem?

Implementação via software (ii)

```
// 1: a thread wants to enter critical section, 0: it doesn't
int flag[2] = {0, 0};

lock()
{
    flag[self] = 1; // I need lock
    while (flag[1-self] == 1)
        ; // spin wait
}

unlock()
{
    // not any more
    flag[self] = 0;
}
```

- ❑ Idea: use per thread flags, set then test, to achieve mutual exclusion

Implementação via software (iii)

```
// whose turn is it?  
int turn = 0;  
  
lock()  
{  
    // wait for my turn  
    while (turn == 1 - self)  
        ; // spin wait  
}  
  
unlock()  
{  
    // I'm done. your turn  
    turn = 1 - self;  
}
```

- ❑ Idea: strict alternation to achieve mutual exclusion

P1 entra, muda turn; P2 não está interessado na SC,
porém P1 está!

Algoritmo de Peterson

```
// whose turn is it?  
int turn = 0;  
// 1: a thread wants to enter critical section, 0: it doesn't  
int flag[2] = {0, 0};  
  
lock()  
{  
    flag[self] = 1; // I need lock  
    turn = 1 - self;  
    // wait for my turn  
    while (flag[1-self] == 1  
        && turn == 1 - self)  
        ; // spin wait while the  
        // other thread has intent  
        // AND it is the other  
        // thread's turn  
}  
  
unlock()  
{  
    // not any more  
    flag[self] = 0;  
}
```

□ Why works?

- Safe?
- Live?
- Bounded wait?

Algoritmo de Peterson

- Implementação de locks via software que satisfaz todas as propriedades
- Não requer ajuda de hardware
- Suposições:
 - Loads e stores são atômicos
 - Não requerem instruções especiais
- Complicações:
 - $N > 2$ (Algoritmo do padeiro, “Bakery algorithm”)
 - CPUs que executam instruções fora de ordem

Implementação: “test and set”

```
// 0: lock is available, 1: lock is held by a thread
int flag = 0;

lock()
{
    while(test_and_set(&flag))
        ;
}

unlock()
{
    flag = 0;
}
```

- Problem with the test-then-set approach: **test and set are not atomic**
- Fix: **special atomic operation**
 - **int test_and_set (int *lock)**
 - Atomic: returns ***lock** and sets ***lock to 1**

“test and set” (x86)

```
long test_and_set(volatile long* lock)
{
    int old;
    asm("xchgl %0, %1"
        : "=r"(old), "+m"(*lock) // output
        : "0"(1)                // input
        : "memory"              // can clobber anything in memory
        );
    return old;
}
```

- **xchg reg, addr**: atomically swaps ***addr** and **reg**
- Some version of Linux **spin_lock** is implemented using this instruction ([include/asm-i386/spin_lock.h](#))

“Spin lock” ou bloqueio ?

- Problem: waste CPU cycles
 - Worst case: prev thread holding a busy-wait lock gets preempted, other threads try to acquire the same lock
- On uniprocessor: should not use spin-lock
 - Yield CPU when lock not available (need OS support)
- On multi-processor

Problemas adicionais

- Pipelined CPUs
 - CPU pode reordenar as instruções (“out of order execution”)
 - Disponibilizar meios para desligar, pois podem causar problemas de “timing”
 - e.g. *test and set*
- Multiprocessamento simétrico (SMP)
 - Kernel usa “spin locks”
 - Hardware deve manter o cache coerente com alterações em variáveis compartilhadas.

Passando o bastão: yield()

```
lock()
{
    while(test_and_set(&flag))
        yield();
}
```

- Problem:
 - Still a lot of context switches: **thundering herd**
 - Starvation possible
- Why? **No control** over who gets the lock next
- **Need explicit control** over who gets the lock

Implementação de locks

```
lock() {  
    while (test_and_set(&flag))  
        add myself to wait queue  
        yield  
    ...  
}
```

```
unlock() {  
    flag = 0  
    if(any thread in wait queue)  
        wake up one wait thread  
    ...  
}
```

Prob II: Lock from
a third thread?

- The idea: **add thread to queue when lock unavailable; in unlock(), wake up one thread in queue**

possível intervenção de um terceiro thread

Implementação

```
typedef struct __mutex_t {
    int flag;      // 0: mutex is available, 1: mutex is not available
    int guard;     // guard lock to avoid losing wakeups
    queue_t *q;   // queue of waiting threads
} mutex_t;

void lock(mutex_t *m) {
    while (test_and_set(m->guard))
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // acquire mutex
        m->guard = 0;
    } else {
        enqueue(m->q, self);
        m->guard = 0;
        yield();
    }
}

void unlock(mutex_t *m) {
    while (test_and_set(m->guard))
        ;
    if (queue_empty(m->q))
        // release mutex; no one wants mutex
        m->flag = 0;
    else
        // direct transfer mutex to next thread
        wakeup(dequeue(m->q));
    m->guard = 0;
}
```

Semáforos: motivação

- **Problem with lock:** mutual exclusion, but no ordering
- **Producer-consumer problem:** need order
 - \$ cat 1.txt | sort | uniq | wc
 - **Producer:** creates a resource
 - **Consumer:** uses a resource
 - **bounded buffer between them**
 - **Scheduling order:** producer waits if buffer full, consumer waits if buffer empty

Tarefa de casa

- Quais as instruções que são atômicas no x86 e que podem auxiliar na confecção de travas ?
- Quais os system calls disponíveis para a sincronização de processos em Linux ?
 - Em Windows ?
 - No xv6 ?