

# Monitores

Carlos Baquero (Slides: Paulo Sérgio Almeida)

Grupo de Sistemas Distribuídos  
Departamento de Informática  
Universidade do Minho

# Problemas com semáforos

- ▶ Semáforo é uma primitiva de baixo nível.
- ▶ Não é estruturada (analogia com “goto”).
- ▶ É fácil cometer erros com semáforos:
  - ▶ esquecer de um release que emparelhe com um acquire;
  - ▶ trocar ordem de acquires;
- ▶ Para sistemas grandes, a associação entre recursos e semáforos que os protegem pode não ser clara.
- ▶ Envolvem alguma “arte” na construção de soluções.
- ▶ Seria desejável uma primitiva que permita a construção sistemática de soluções.

# Monitores

- ▶ Primitiva estruturada de controlo de concorrência.
- ▶ Tipo de dados com operações, que encapsula estado.
  - ▶ Semelhança com objectos.
- ▶ Acesso concorrente é controlado internamente.
- ▶ Clientes podem simplesmente invocar operações.
- ▶ Apenas um processo pode estar “dentro” num dado momento.
  - ▶ Exclusão mútua é obtida implicitamente.
- ▶ Disponibiliza **variáveis de condição**.
  - ▶ Permitem processos bloquearem-se voluntariamente.
  - ▶ Usadas em problemas de ordem-de-execução (mas não só).

## Exemplo: bounded-buffer como monitor

- ▶ Tipo abstracto de dados com controlo de concorrência.
- ▶ Exclusão mútua é garantida internamente pelo buffer.
- ▶ Buffer contém mutex implícito (não declarado).
- ▶ Cada operação adquire e liberta o mutex, implicitamente.

```
monitor Buffer {  
    // mutex(mut) implicito  
    int a[N];  
    ...  
    int take() {  
        // lock(mut) feito implicitamente  
        ...  
        // unlock(mut) feito implicitamente  
        return x;  
    }  
    put(int x) {  
        // lock(mut) feito implicitamente  
        ...  
        // unlock(mut) feito implicitamente  
    }  
}
```

# Notas

- ▶ Monitor é uma entidade passiva, que é usada por processos.
- ▶ Exclusão mútua é obtida trivialmente.
- ▶ Tal como em semáforos, pode haver starvation na entrada.
- ▶ Monitores são usados em linguagens modernas via objectos.
- ▶ Monitores servem de base a primitivas em bibliotecas de concorrência para linguagens procedimentais; e.g. pthreads.

# Variáveis de condição – condition variables

- ▶ Para além de exclusão mútua podemos ter outros requisitos.
- ▶ Exemplo: consumidor não pode prosseguir se buffer vazio.
- ▶ Variáveis de condição permitem a um processo bloquear-se voluntariamente.
- ▶ Variáveis de condição são declaradas explicitamente.
- ▶ Por tradição, o nome deverá sugerir uma condição (predicado) que se verdadeira permite ao processo prosseguir; e.g.:
  - ▶ `condition notEmpty;`
  - ▶ `condition notFull;`
- ▶ Processos testam predicado sobre variáveis de estado do monitor e decidem se bloqueiam.
- ▶ As variáveis de condição não têm valor que se leia ou escreva; o termo “variável” vem do aspecto sintáctico da declaração.
- ▶ Também chamadas *condition queues*.

# Operações sobre variáveis de condição (monitores clássicos)

- ▶ A cada v.c. é associada uma fila  $f$  de processos bloqueados.
- ▶ Sendo  $p$  o processo actual que executa num monitor  $mon$ :
- ▶ Primitiva **wait** bloqueia processo na v.c.

```
wait(cond):  
    cond.f.append(p)  
    unlock(mon.mut)  
    suspend()  
    ...
```

- ▶ wait liberta mutex antes de bloquear processo.
- ▶ Primitiva **signal** liberta processo bloqueado na v.c.

```
signal(cond):  
    if cond.f != []:  
        q = cond.f.pop(0)  
        ready(p)
```

- ▶ Se não existir processo bloqueado, o signal “perde-se” (ao contrário dos semáforos).

# Exemplo: implementar semáforos com monitores

- ▶ Com monitores é possível implementar semáforos (fortes).
- ▶ Implementação com monitores **clássicos**:

```
monitor Semaphore {  
    int v;  
    condition notZero;  
  
    acquire() {  
        if (v == 0)  
            wait(notZero);  
        v = v - 1;  
    }  
  
    release() {  
        v = v + 1;  
        signal(notZero);  
    }  
}
```

- ▶ A seguir ao signal a execução continua no wait (se existir processo bloqueado).



## Exemplo: bounded-buffer como monitor

- ▶ Exclusão mútua é garantida implicitamente.
- ▶ Pode ser necessário bloquear operação se buffer vazio ou cheio.

```
monitor Buffer {
    condition notEmpty;
    condition notFull;
    int a[N], nitems, ...;

    int take() {
        if (nitems == 0)
            wait(notEmpty);
        x = ...
        nitems--;
        signal(notFull);
        return x;
    }

    put(int x) {
        if (nitems == N)
            wait(notFull);
        ...
        nitems++;
        signal(notEmpty);
    }
}
```

# Produtor-consumidor com bounded-buffer como monitor

- ▶ O buffer trata da exclusão mútua e ordem-de-execução.
- ▶ O código do produtor e consumidor fica trivial.
- ▶ Sendo buffer um monitor do tipo Buffer atrás:

Consumidor:

```
while (...) {  
    x = buffer.take();  
    consume(x);  
}
```

Produtor

```
while (...) {  
    x = produce();  
    buffer.put(x);  
}
```

# Continuação da execução de signal e wait

- ▶ Como apenas uma operação pode estar a executar num monitor num dado momento . . .
- ▶ Quando um processo  $s$  faz signal, estando outro processo  $w$  bloqueado num wait, qual prossegue em seguida?
  - ▶ continua  $s$  imediatamente até acabar a operação (ou wait)?
  - ▶ continua  $w$  imediatamente, retomando  $s$  mais tarde?
  - ▶ pode executar um terceiro processo à espera de obter o lock?
  - ▶ há alguma garantia ou pode ser indeterminado?
- ▶ Conforme as garantias teremos que ter diferentes algoritmos.

# Semáforos via monitores; revisitado

- ▶ Reparemos no fragmento da implementação de semáforos:

```
acquire() {  
    if (v == 0)  
        wait(notZero);  
    v = v - 1;  
}  
release() {  
    v = v + 1;  
    signal(notZero);  
}
```

- ▶ O que acontece se depois de signal, wait não executar imediatamente e se intrometer um terceiro processo?
  - ▶ suponhamos que o terceiro processo faz acquire.

# Bounded-buffer com monitores; revisitado

- Reparemos no fragmento da implementação:

```
int take() {  
    if (nitems == 0)  
        wait(notEmpty);  
    ...  
}  
put(int x) {  
    ...  
    nitems++;  
    signal(notEmpty);  
}
```

- O que acontece se o wait de um take não executar imediatamente a seguir ao signal do put, tendo ficando 1 item no buffer, e se intrometer um terceiro processo a fazer take?

# Execução a seguir a um signal

- ▶ Os exemplos anteriores mostram que, se um wait não prosseguir imediatamente a seguir ao signal:
  - ▶ um outro processo pode alterar o estado do monitor;
  - ▶ o predicado que o wait esperava pode ficar outra vez falso;
  - ▶ os algoritmos anteriores falham.
- ▶ Os algoritmos anteriores são para **monitores clássicos**.

# Monitores clássicos

- ▶ Nestes monitores é garantido que:
  - ▶ se estiver algum processo bloqueado num wait, a seguir a um signal, prossegue o processo bloqueado;
  - ▶ assim, se predicado é verdadeiro quando é feito signal ...
  - ▶ ... o predicado permanece verdadeiro depois do wait;
  - ▶ mais tarde prossegue quem fez signal;
  - ▶ finalmente podem entrar no monitor outros processos.
- ▶ Este comportamento é chamado de **immediate resumption requirement** ou **signal and urgent wait**.
- ▶ Podemos então usar testes de predicados com **if**:

```
if (!predicado())  
    wait(cond);
```

# Diferentes variantes de monitores

- ▶ O caso anterior é apenas uma de diferentes possibilidades.
- ▶ Dadas as filas de candidatos a prosseguir aquando um signal:
  - ▶ processos que fizeram signal (S) (caso este não prossiga logo);
  - ▶ processos desbloqueados, à espera de retornar do wait (W);
  - ▶ processos à espera de entrar (E), adquirindo o lock;
- ▶ Quando é feito signal, quem continua a execução?
- ▶ Designando por E, W, S, as prioridades destas classes de processos, podemos ter várias hipóteses:
  - ▶ monitores clássicos:  $E < S < W$
  - ▶ Pthreads e Java:  $E = W < S$
  - ▶ ...



# SignalAll

Disponibilização de variante de signal, **signalAll**:

- ▶ acorda todos os processos bloqueados na variável de condição;
- ▶ encontra-se em monitores modernos; e.g em Java.

# Monitores modernos

- ▶ Os mais usados actualmente, e.g., em Java e Pthreads têm:

$$E = W < S$$

- ▶ Ou seja:
  - ▶ primeiro continua o processo que faz signal;
  - ▶ depois pode correr o processo acordado ou
  - ▶ pode correr um terceiro processo que estivesse a querer entrar;
- ▶ Como um terceiro processo pode ter mudado o estado do monitor, o predicado pode já não ser verdadeiro depois do wait.
- ▶ Conclusão: temos que usar testes de predicados com **while**:

```
while (!predicado())  
    wait(cond);
```

# Monitores modernos e spurious wakeups

- ▶ Às vezes, poderíamos ser tentados a não usar while:
  - ▶ se não mudássemos o estado depois do signal e
  - ▶ soubéssemos que mais nenhum processo pudesse estar a tentar entrar no monitor, não havendo perigo de ultrapassagem.
- ▶ Um outro fenómeno vai, no entanto, obrigar ao uso de ciclos de espera: os **spurious wakeups**.
- ▶ Para obter implementações eficientes de monitores em multiprocessadores, **um wait pode, embora muito raramente, desbloquear mesmo sem ter sido feito signal**.
- ▶ Conclusão: temos que fazer **sempre** espera em ciclo.

## Exemplo: leitores e escritores

- ▶ Caso mais geral de exclusão mútua.
- ▶ Suponhamos duas classes de processos:
  - ▶ readers: querem fazer operações de leitura sobre um recurso;
  - ▶ writers: querem fazer operações de escrita sobre um recurso;
- ▶ Um bloco de operações de leitura ou escrita é rodeado de código de sincronização; assim existem 4 operações:
  - ▶ readLock e readUnlock para rodear bloco de leitura;
  - ▶ writeLock e writeUnlock para rodear bloco de escrita.
- ▶ Requisitos de segurança:
  - ▶ podem estar vários processos a ler;
  - ▶ se um processo estiver a escrever, mais nenhum pode estar a ler ou escrever.
- ▶ Problema: implementar as 4 operações de sincronização.

# Leitores e escritores com monitores modernos (starvation de escritores)

```
monitor RWLock { //  $E = W < S$ 
    int readers = 0, writers = 0;
    condition OKread, OKwrite;
    readLock() {
        while (writers != 0) wait(OKread);
        readers++;
        signal(OKread);
    }
    readUnlock() {
        readers--;
        if (readers == 0) signal(OKwrite);
    }
    writeLock() {
        while (writers != 0 || readers != 0) wait(OKwrite);
        writers++;
    }
    writeUnlock() {
        writers--;
        signal(OKread);
        signal(OKwrite);
    }
}
```



# Leitores e escritores com monitores modernos (starvation de leitores)

```
monitor RWLock { //  $E = W < S$ 
    int readers = 0, writers = 0, wantWrite = 0;
    condition OKread, OKwrite;
    readLock() {
        while (writers != 0 || wantWrite > 0) wait(OKread);
        readers++;
        signal(OKread);
    }
    readUnlock() {
        readers--;
        if (readers == 0) signal(OKwrite);
    }
    writeLock() {
        wantWrite++;
        while (writers != 0 || readers != 0) wait(OKwrite);
        wantWrite--; writers++;
    }
    writeUnlock() {
        writers--;
        signal(OKread);
        signal(OKwrite);
    }
}
```