Modelos para Computação Concorrente ou Sistemas Operacionais

Memória Compartilhada - Monitores

(com slides de Ben-Ari)

Fernando Luís Dotti

PUCRS | ESCOLA POLITÉCNICA

- Semáforos
  - Construções de baixo nível de abstração
  - Depende da construção e uso correto nos processos que usam a estrutura compartilhada

- Monitores
  - Provêem estrutura que concentra responsabilidade pelo acesso concorrente correto junto à estrutura representada
  - Encapsulamento

- Monitores
  - Encapsulamento + Sincronização
  - Processos (threads) usuárias do Monitor não se envolvem no problema de sincronização das operações

Monitor

(Brinch Hansen, Hoare) 73, 74

- É um modelo que permite o compartilhamento de dados
- possui valores que representam o estado do objeto e as procedures que manipulam os valores
- as procedures são executadas de forma mutuamente exclusiva
- variáveis especiais (condição) permitem a um processo se bloquear a espera de uma condição (wait)
- a condição é sinalizada por um outro processo (operação signal)

Monitor

(Brinch Hansen, Hoare) 73, 74

- Variáveis Condição:
  - ex.: var x, y : condition ;
  - x.wait:
    - o processo que executa essa operação é suspenso até que um outro processo execute a operação x.signal
  - x.signal:
    - acorda um único processo
    - se não existem processos bloqueados, a operação não produz efeitos

Algorithm 7.3: Producer-consumer (finite buffer, monitor)

```
monitor PC
    bufferType buffer ← empty
    condition notEmpty
    condition notFull
    operation append(datatype V)
        if buffer is full
            waitC(notFull)
        append(V, buffer)
        signalC(notEmpty)
    operation take()
        datatype W
        if buffer is empty
            waitC(notEmpty)
        W ← head(buffer)
        signalC(notFull)
        return W
```

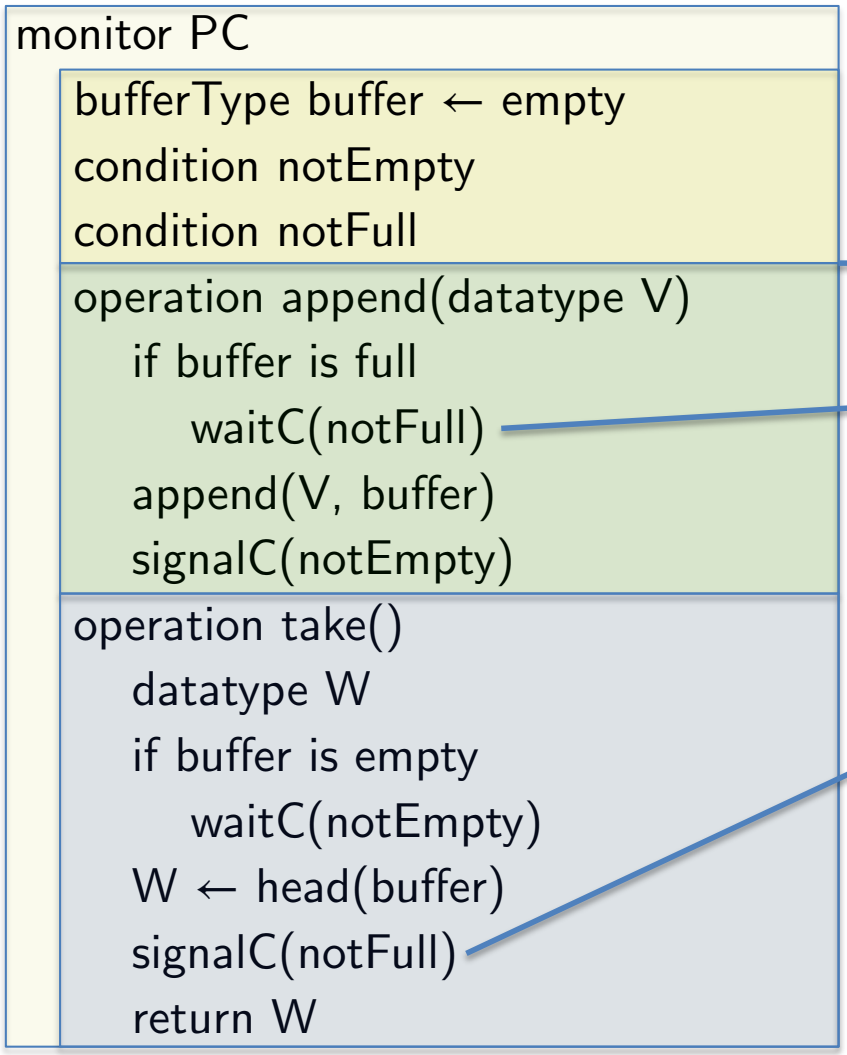
Entrada do monitor
operação atômica

Entrada do monitor
operação atômica

```
monitor PC
    bufferType buffer ← empty
    condition notEmpty
    condition notFull

    operation append(datatype V)
        if buffer is full
            waitC(notFull)
        append(V, buffer)
        signalC(notEmpty)

    operation take()
        datatype W
        if buffer is empty
            waitC(notEmpty)
        W ← head(buffer)
        signalC(notFull)
        return W
```

Algorithm 7.3: Producer-consumer (finite buffer, monitor) (continued)

	producer		**consumer**
	datatype D		datatype D
	loop forever		loop forever
p1:	D ← produce	q1:	D ← PC.take
p2:	PC.append(D)	q2:	consume(D)



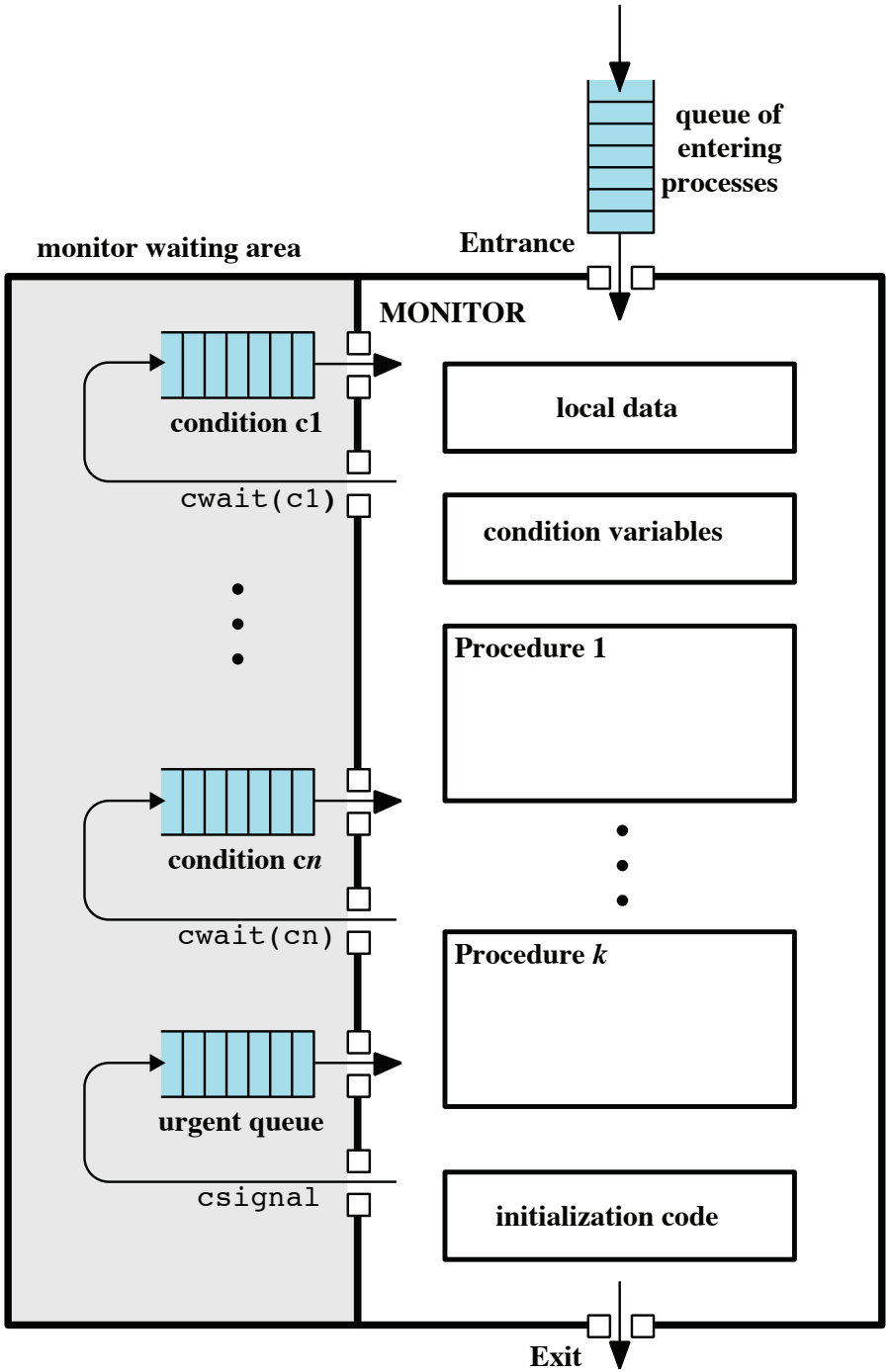
Como atomicidade é mantida ?

Como um processo pode entrar em wait dentro do monitor ? Deixa monitor bloqueado ?

Como um processo sinaliza outro para prosseguir ? Então teremos dois processos no monitor ?

Algorithm 7.3: Producer-consumer (finite buffer, monitor) (continued)

	producer		**consumer**
	datatype D		datatype D
	loop forever		loop forever
p1:	D ← produce	q1:	D ← PC.take
p2:	PC.append(D)	q2:	consume(D)


queue of
entering
processes
Entrance
monitor waiting area
MONITOR
condition c1
cwait(c1)
local data
condition variables
Procedure 1
condition cn
cwait(cn)
Procedure k
urgent queue
csignal
initialization code
Exit

Monitor

(Brinch Hansen, Hoare) 73, 74

- Implementação de Monitor com Semáforo:
  - cada monitor é representado por um semáforo **mutex** inicializado com 1.
    wait (mutex): entrar no monitor
    signal (mutex): liberar monitor
  - semáforo **next** inicializado com 0
    para um processo sinalizador se bloquear
  - next count: contém o No. de processos bloqueados em next

MonitorEntry: ***wait (mutex)***

" corpo da procedure entry "

MonitorExit: ***if next_count > 0***
then signal (next)
else signal (mutex)

Monitor

(Brinch Hansen, Hoare) 73, 74

– Para cada variável condition X, associar
  um semáforo **x_sem**
  uma variável inteira **x_count**

e as operações:

```
X.waitC: x_count ++                  /* vou me bloquear */
         if  next_count > 0          /* se algum proc estava no monitor e foi bloq, libera */
         then  signal ( next )
         else signal ( mutex )       /* senão libera proc querendo entrar no monitor */
         wait ( x_sem )              /* se bloqueia */
         x_count --                  /* depois de desbloquear, diminui nro de bloqueados */
```

```
X.signalC:if x_count > 0             /* se há processo bloqueado */
          then  next_count ++        /* vou me bloquear para liberar o processo bloqueado! */
                signal ( x-sem )     /* desbloqueia processo */
                wait ( next )        /* se bloqueia em next com preferencia sobre procs fora */
                next-count  --       /* quando desbloqueado de next, decrementa */
          else vazio
```

Monitor (Brinch Hansen, Hoare) 73, 74

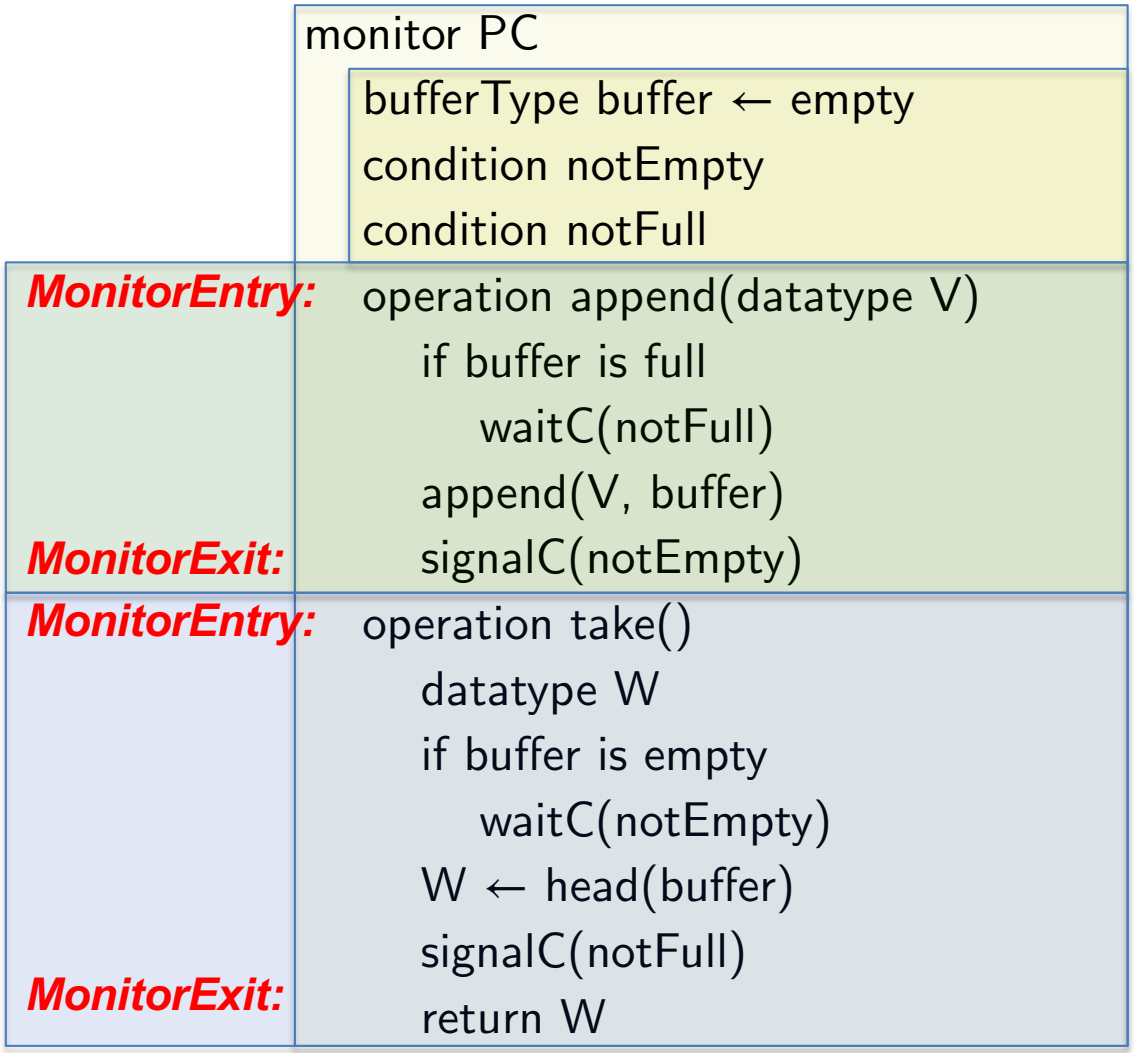
```
MonitorEntry :
    wait(mutex)

MonitorExit :
    if next_count > 0          /* se algum processo bloqueado em next
    then  signal ( next )      /* libera de next (maior prioridade) */
    else  signal ( mutex )     /* senao libera mutex */
```

mutex : semaforo para exclusao mutua
next : semáforo de bloqueados devido a signal em condicao
next_count: numero de bloqueados

para cada variável condition X, x_sem int x_count

```
X.waitC: x_count ++                /* vou me bloquear */
         if  next_count > 0        /* se algum proc estava no monitor e foi bloq, libera */
         then  signal ( next )
         else signal ( mutex )     /* senão libera proc querendo entrar no monitor */
         wait ( x_sem )            /* se bloqueia */
         x_count - -               /* depois de desbloquear, diminui nro de bloqueados */
```

```
X.signalC:if x_count > 0           /* se há processo bloqueado */
            then  next_count ++    /* vou me bloquear para liberar o processo bloqueado! */
                  signal ( x-sem ) /* desbloqueia processo */
                  wait ( next )    /* se bloqueia em next com preferencia sobre procs fora */
                  next-count  --   /* quando desbloqueado de next, decrementa */
            else vazio
```


monitor PC
bufferType buffer ← empty
condition notEmpty
condition notFull
MonitorEntry:
operation append(datatype V)
if buffer is full
waitC(notFull)
append(V, buffer)
MonitorExit:
signalC(notEmpty)
MonitorEntry:
operation take()
datatype W
if buffer is empty
waitC(notEmpty)
W ← head(buffer)
signalC(notFull)
MonitorExit:
return W