Sistemas Operativos

Licenciatura Engenharia Informática
Licenciatura Engenharia Computacional

Ano letivo 2023/2024

Nuno Lau (nunolau@ua.pt)

Monitores

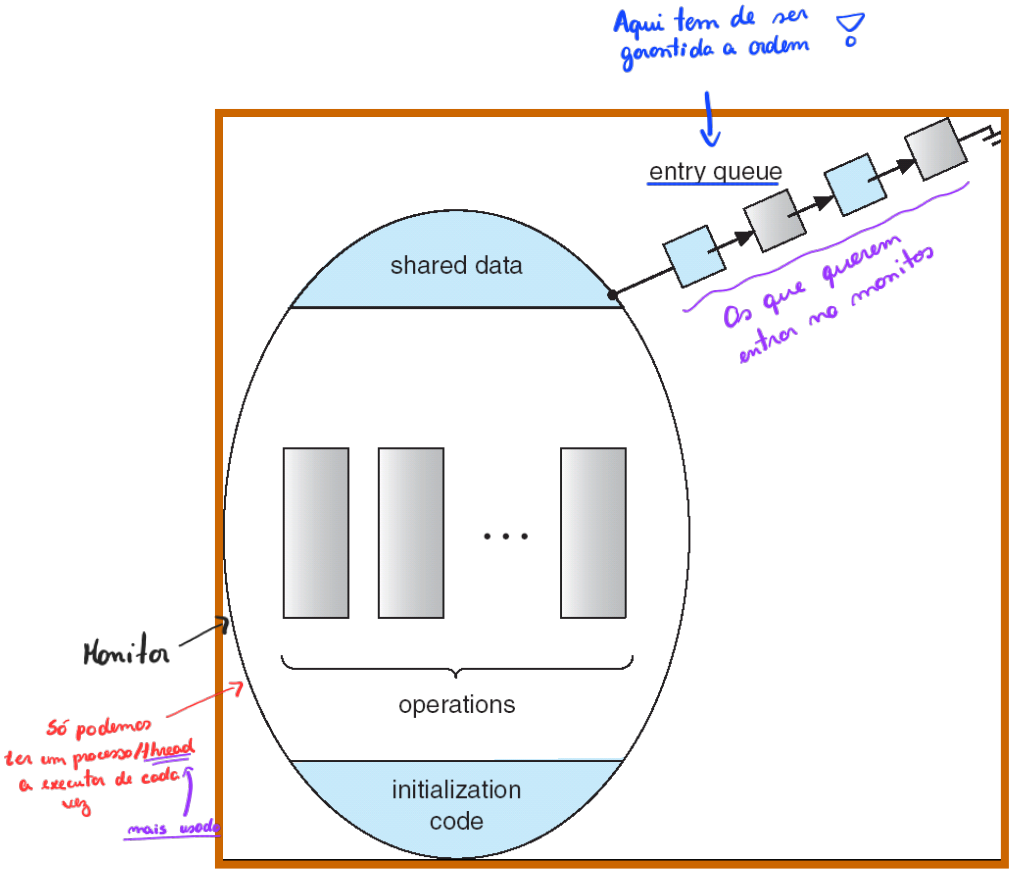
~> Hoje em dia mais utilizado que os semáforos!

- Abstração de alto nível usada para sincronização de processos
- Apenas um processo pode estar ativo no monitor de cada vez
- Proposto independentemente por Hoare e Brinch Hansen
- Constituído por:
  - Estrutura de dados interna
  - Código de inicialização
  - Primitivas de acesso

```
monitor monitor name
{
   // shared variable declarations

   initialization code ( . . . ) {
      . . .
   }

   public P1 ( . . . ) {
      . . .
   }

   public P2 ( . . . ) {
      . . .
   }
         .
         .
         .
   public Pn ( . . . ) {
      . . .
   }
}
```

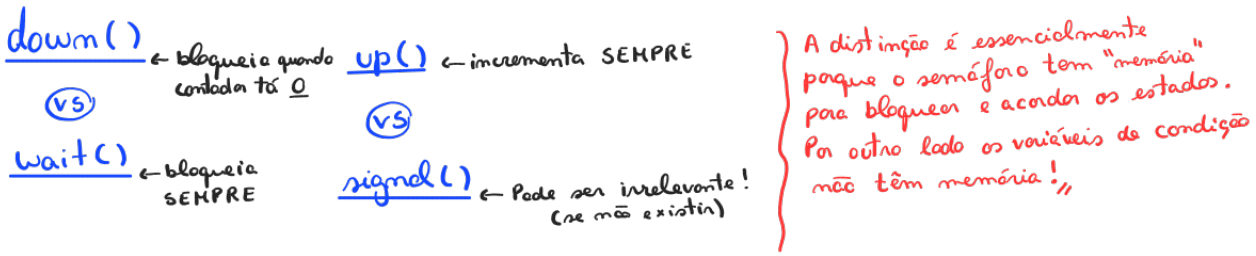
No máximo 1 processo pode estar a executar P2

Monitores

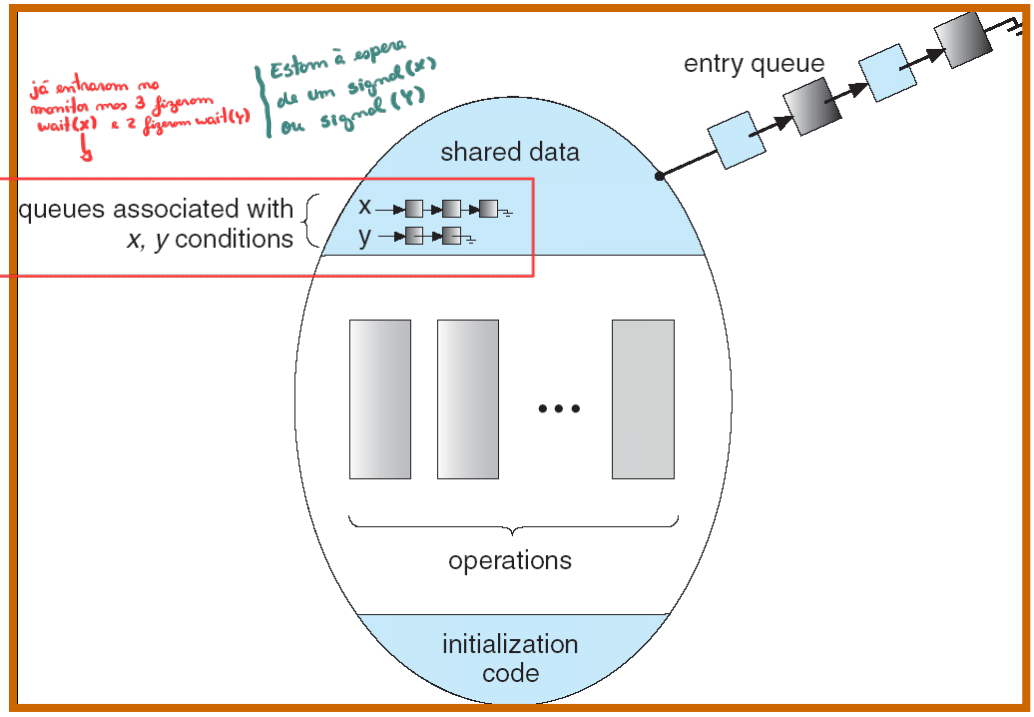

Aqui tem de ser garantida a ordem
entry queue
shared data
Os que querem entrar no monitor
operations
initialization code
Monitor
Só podemos ter um processo/thread a executar de cada vez
mais usado

Variáveis de condição

- Permitem bloquear um processo até que determinada condição se verifique
- 2 operações
  - **Wait()** – bloqueia o processo/thread e liberta o monitor, permitindo que outro processo/thread execute primitivas do monitor
  - **Signal()** – acorda um dos processos (se existir) bloqueado nesta variável de condição; se não existir processo bloqueado nada acontece.



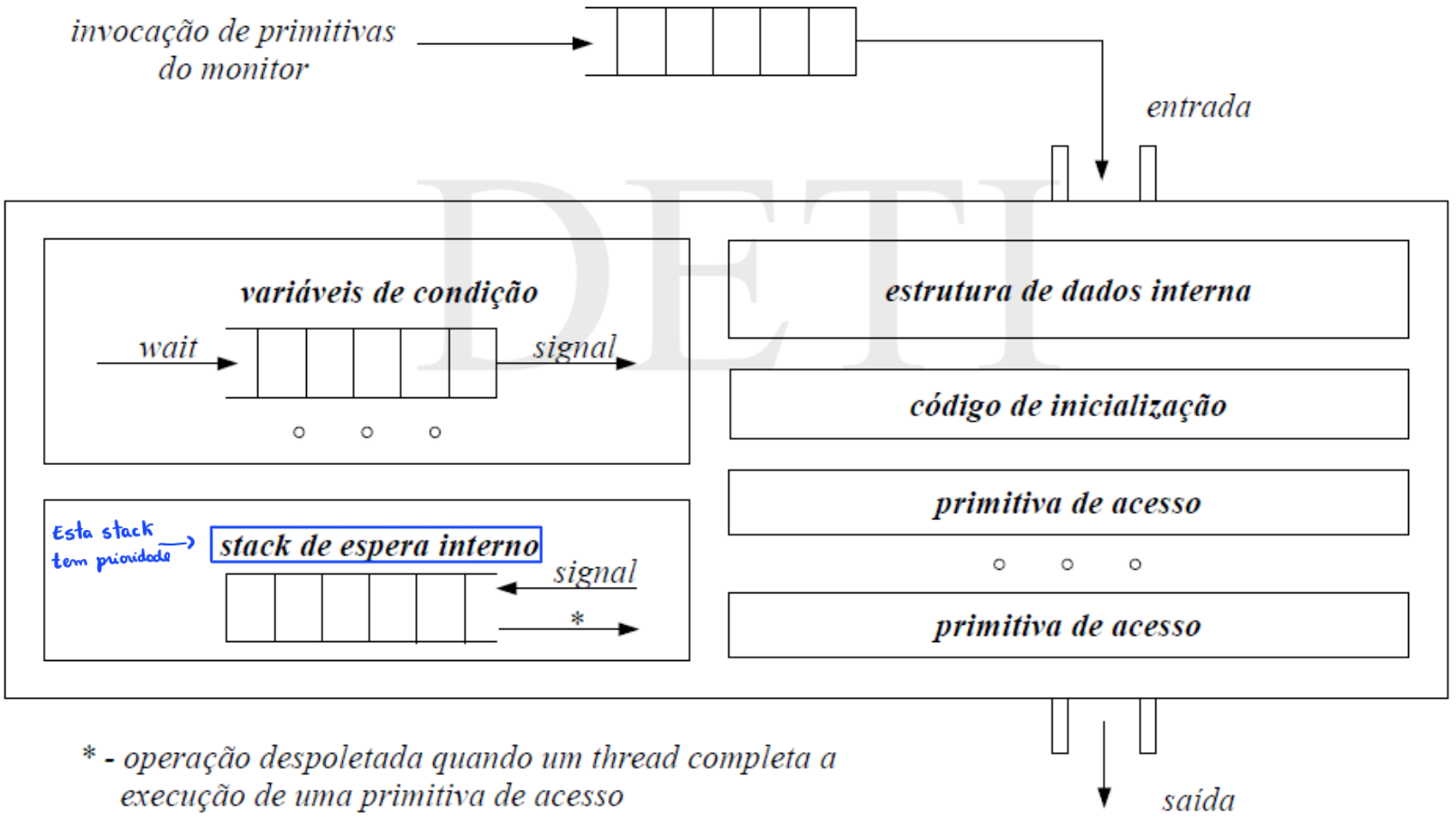
Monitor com 2 Variáveis de condição



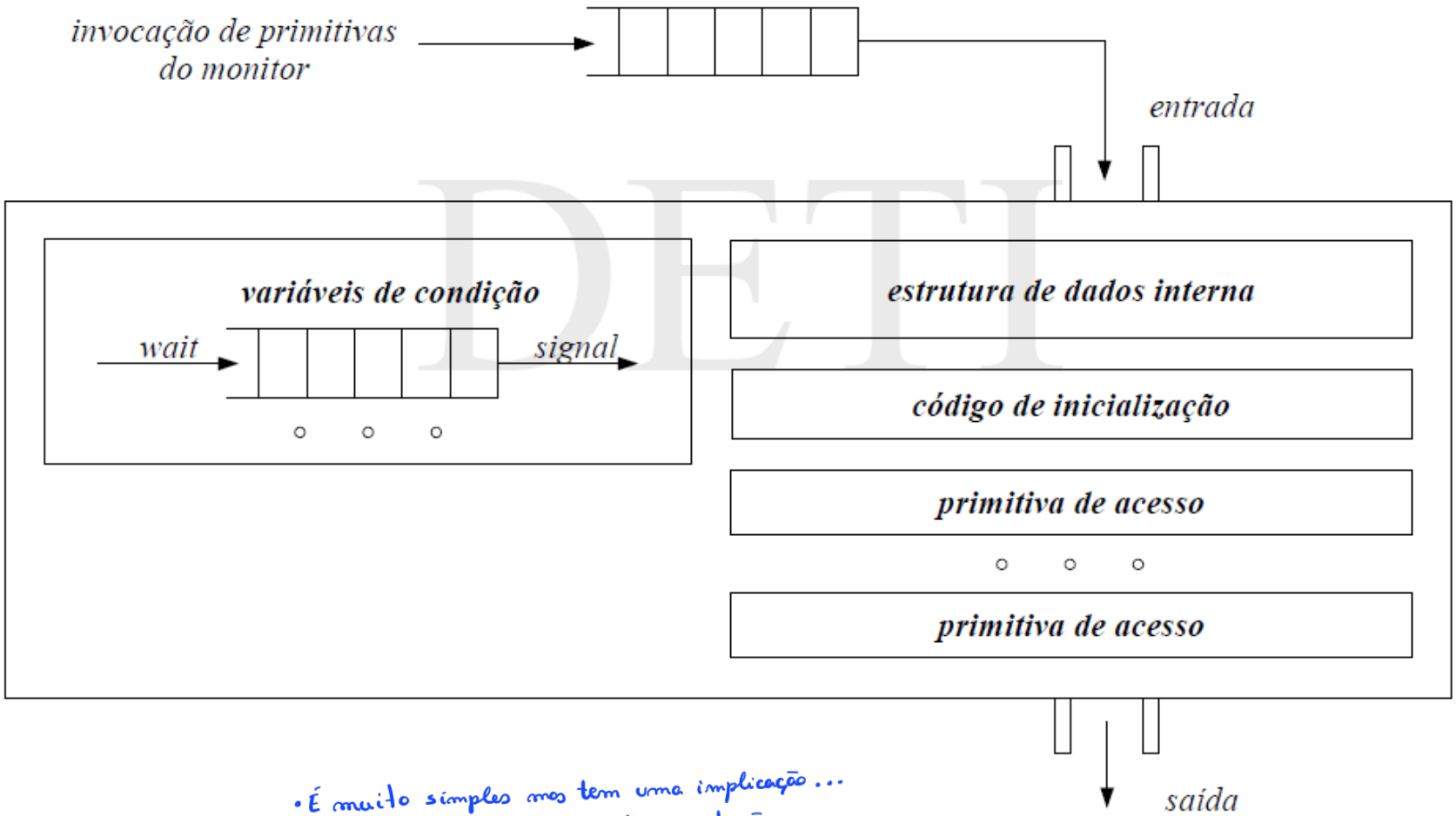
Resolução de *signal*

- Modelos de resolução após a execução de *signal:*
  - *Monitor de Hoare*
    - ***thread* que invoca *signal* é colocada *fora do monitor*** para que a *thread* acordada possa prosseguir;
    - muito geral, mas a sua implementação exige uma *stack*, onde são colocadas as *threads* postas *fora do monitor* por invocação de *signal*;
  - *Monitor de Brinch Hansen*
    - ***thread* que invoca *signal* liberta imediatamente o *monitor*** (*signal* é a última instrução executada);
    - simples de implementar, mas pode tornar-se bastante restritivo porque permite apenas a execução de um *signal* em cada invocação de uma primitiva de acesso;
  - *Monitor de Lampson / Redell*
    - ***thread* que invoca *signal* prossegue a sua execução**, a *thread* acordada mantém-se *fora do monitor* e compete pelo acesso a ele
    - simples de implementar, mas pode originar situações em que algumas *threads* são colocadas em *adiamento indefinido*.

Monitor de Hoare



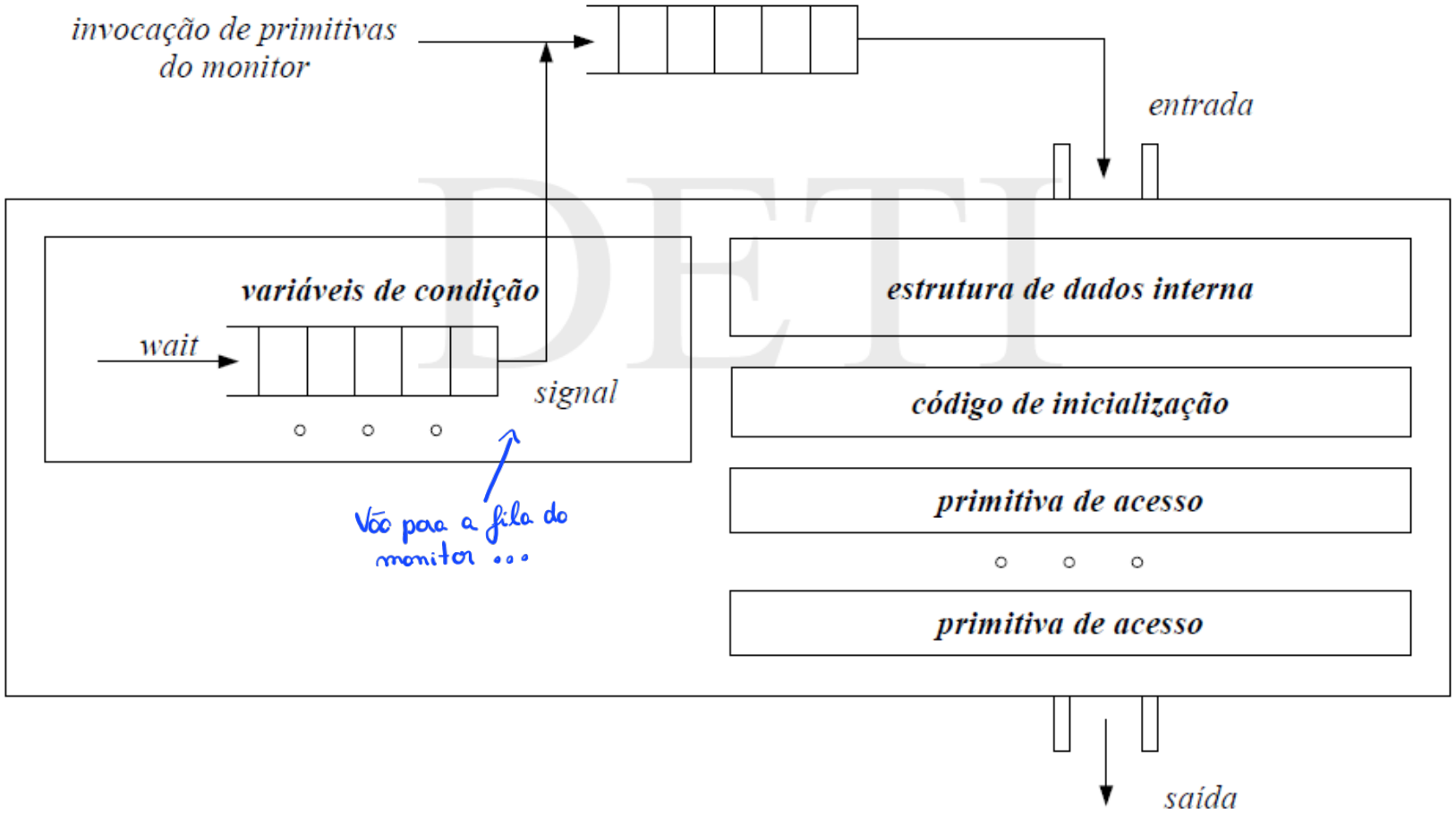
Monitor de Brinch Hansen



- É muito simples mas tem uma implicação ...
  → "signal" tem de ser a última instrução

Monitor de Lampson / Redell

Mais utilizada!



Produtores / Consumidores

```
monitor transf;                                   (* monitor de Hoare / Brinch Hansen *)
  var
    fifo: FIFO;                                   (* memória de tipo FIFO de tamanho K *)
    n_pos_vazias: integer;                        (* n. de posições vazias *)
    fifo_vazio, fifo_cheio: condition;            (* sinal. de FIFO vazio e FIFO cheio *)
  (* introdução de um valor *)
  procedure put_val (val: DATA);
  begin
    if n_pos_vazias = 0 then wait (fifo_cheio);   (* verifica se há espaço *)
    fifo_in (fifo, val);                          (* armazenamento *)
    n_pos_vazias := n_pos_vazias - 1;             (* actualiza o estado interno *)
    signal (fifo_vazio)                           (* testa a eventualidade *)
  end; (* put_val *)                             (* de haver threads consumidores à espera *)
  (* retirada de um valor *)
  procedure get_val (var val: DATA);
  begin
    if n_pos_vazias = K then wait (fifo_vazio);   (* verifica se há informação *)
    fifo_out (fifo, val);                         (* recolha *)
    n_pos_vazias := n_pos_vazias + 1;             (* actualiza o estado interno *)
    signal (fifo_cheio)                           (* testa a eventualidade *)
  end; (* get_val *)                             (* de haver threads produtores à espera *)
  begin
    n_pos_vazias := K;                            (* situação inicial *)
  end;
end monitor; (* transf *)
```

Estamos no Monitor !!!

→ No caso de alguém estar à espera para remover ...

Produtores / Consumidores (Pode sair no teste)

```
monitor transf;                                   (* monitor de Lampson / Redell *)
  var
    fifo: FIFO;                          (* memória de tipo FIFO de tamanho K *)
    n_pos_vazias: integer;                          (* n. de posições vazias *)
    fifo_vazio, fifo_cheio: condition;   (* sinal. de FIFO vazio e FIFO cheio *)
  (* introdução de um valor *)
  procedure put_val (val: DATA);
  begin
    while n_pos_vazias = 0 do wait (fifo_cheio);      (* verifica se há espaço *)
    fifo_in (fifo, val);                                    (* armazenamento *)
    n_pos_vazias := n_pos_vazias - 1;           (* actualiza o estado interno *)
    signal (fifo_vazio)                             (* testa a eventualidade *)
  end; (* put_val *)                    (* de haver threads consumidores à espera *)
  (* retirada de um valor *)
  procedure get_val (var val: DATA);
  begin
    while n_pos_vazias = K do wait (fifo_vazio);  (* verifica se há informação *)
    fifo_out (fifo, val);                                        (* recolha *)
    n_pos_vazias := n_pos_vazias + 1;           (* actualiza o estado interno *)
    signal (fifo_cheio)                             (* testa a eventualidade *)
  end; (* get_val *)                     (* de haver threads produtores à espera *)
  begin
    n_pos_vazias := K;                                   (* situação inicial *)
  end;
end monitor; (* transf *)
```

Precisamos de o colocar na fila →

Temos de fazer isto

`wait()` deve estar sempre num ciclo **`while`** que verifica condições de continuação

Jantar de filósofos

Solução de Tanenbaum

```
monitor DiningPhilosophers
{
    enum State {THINKING, HUNGRY, EATING};
    State[] states = new State[5];
    Condition[] self = new Condition[5];

    public DiningPhilosophers {
       for (int i = 0; i < 5; i++)
          state[i] = State.THINKING;
    }

    public void takeForks(int i) {
       state[i] = State.HUNGRY;
       test(i);
       if (state[i] != State.EATING)
          self[i].wait;
    }

    public void returnForks(int i) {
       state[i] = State.THINKING;
       // test left and right neighbors
       test((i + 4) % 5);
       test((i + 1) % 5);
    }

    private void test(int i) {
       if ( (state[(i + 4) % 5] != State.EATING) &&
          (state[i] == State.HUNGRY) &&
          (state[(i + 1) % 5] != State.EATING) ) {
             state[i] = State.EATING;
             self[i].signal;
       }
    }
}
```

while

Se não puder vai esperar...

Usamos sempre monitores do tipo Lampson / Redell
→ O thread acordado vai para a fila de fora do monitor

Programando com monitores

- Identificar objetos partilhados
  - Definir a sua interface
  - Identificar estado interno e invariantes
  - Implementar métodos de manipulação
- Passos para cada objeto partilhado
  - Criar um *lock*
  - Adicionar código para adquirir e libertar *lock*
  - Identificar e adicionar variáveis de condição
  - Adicionar *loops* nos *waits* das variáveis de condição
  - Adicionar *signal* e *broadcast*

Pode não ser preciso em algumas linguagens

Os métodos vão precisar de um lock() no início e um unlock() no fim (existem linguagens em que não é preciso ...)

Vamos utilizar monitores do tipo Lampson/Redell

Signal all (acorda todos)

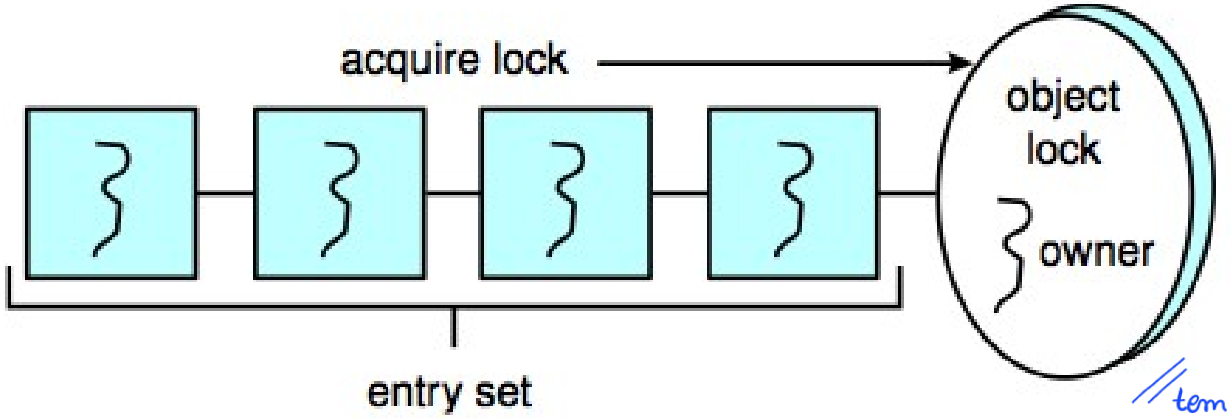
Programando com monitores

- Estrutura consistente
- Usar apenas locks e variáveis de condição para a sincronização
- Adquirir lock sempre no início do método e libertar sempre no fim
- Ter sempre o lock quando se opera sobre variáveis de condição
- Esperar sempre num ciclo while quando o wait é invocado
- Não usar sleep() para esperar por outras threads

Sincronização em Java

- Primitivas de sincronização estão incluídas na própria linguagem Java
- Cada objecto Java tem associado um *lock*
- O *lock* é adquirido ao entrar num método *synchronized*
- O *lock* é libertado ao sair desse método
- *Threads* que têm de esperar são colocadas no *entry set*

Sincronização em Java



Bounded Buffer em Java

• Nunca estão os 2 no monitor ao mesmo tempo

```
public synchronized void insert(Object item) {
    while (count == BUFFER_SIZE)
        Thread.yield();

    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
}
```

lock() — Não são precisos porque "synchronized" faz isso — unlock()

⇐ yield() deverá libertar o lock!

⇒ Não existem Raceconditions, pois no monitor só podemos ter uma thread/processo a executar

```
public synchronized Object remove() {
    Object item;

    while (count == 0)
        Thread.yield();

    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    return item;
}
```

→ Aqui não são utilizados variáveis de condição! Podia ser melhor...

diz ao CPU que pode parar de ser executado

⇐ yield() deverá libertar o lock!

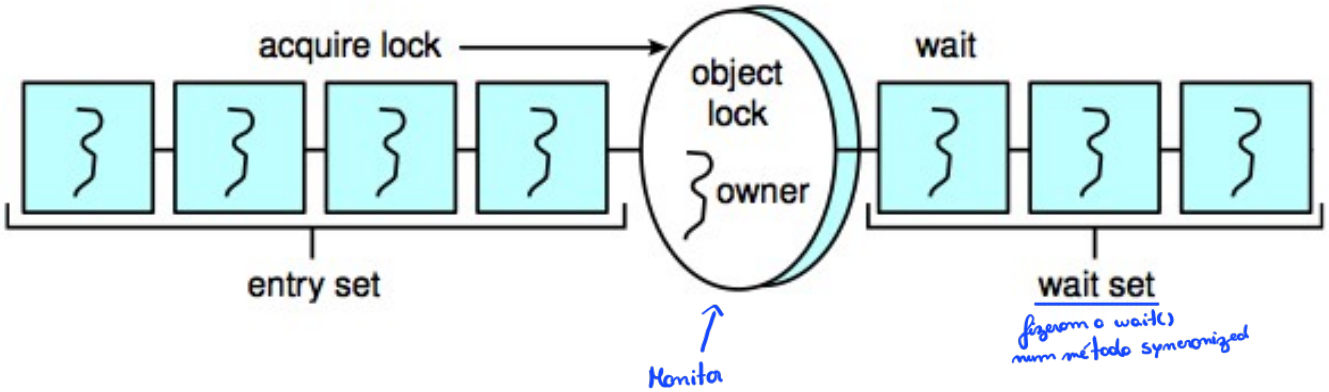
Sincronização em Java

- Cada objecto tem um ***wait set***
- Quando uma *thread* entra num método ***synchronized*** e verifica que não pode prosseguir então pode executar **wait()** → Parecido às variáveis de condição !
  - *Thread* liberta o ***lock*** do objecto
  - É bloqueada
  - É colocada no ***wait set*** do objecto
- Uma outra *thread* pode invocar **notify()** (ou **notifyAll()**) para retirar *threads* do *wait set*
  - Uma *thread* T é retirada do ***wait set*** e colocada no ***entry set***
  - T é colocada no estado *Ready*

signal

broadcast()

Sincronização em Java



Bounded Buffer em Java

```
public synchronized void insert(Object item) {
    while (count == BUFFER_SIZE) {
        try {
            wait();
        }
        catch (InterruptedException e) { }
    }

    ++count;
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;

    notify();
}
```

wait() → ela liberta o lock()!

Aqui não temos mais de uma variável de condição

```
public synchronized Object remove() {
    Object item;

    while (count == 0) {
        try {
            wait();
        }
        catch (InterruptedException e) { }
    }

    --count;
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    notify();

    return item;
}
```

Sincronização de blocos

- Também é possível sincronizar apenas uma secção de código interna a um método

```
Object mutexLock = new Object();
. . .
public void someMethod() {
    nonCriticalSection();

    synchronized(mutexLock) {
       criticalSection();
    }

    remainderSection();
}
```

Envolve apenas uma parte do método →

Concorrência em Java

- Semáforos:

Isto seria um semáforo ...

```
Semaphore sem = new Semaphore(1);

try {
   sem.acquire();
   // critical section
}
catch (InterruptedException ie) { }
finally {
   sem.release();
}
```

Concorrência em Java

- *Locks*
  - Semelhantes a mutex
  - Métodos **lock()** e **unlock()**
- Variáveis de Condição
  - Associadas a *locks*
  - Métodos **await()** e **signal()**

```
Lock key = new ReentrantLock();
Condition condVar = key.newCondition();
```

Concorrência em C

- Semáforos:

Operation	
Initialize a semaphore	**`sem_init()`**
Increment a semaphore	**`sem_post`**`()`
Block on a semaphore count	**`sem_wait()`**
Decrement a semaphore count	**`sem_trywait()`**
Destroy the semaphore state	**`sem_destroy()`**

Concorrência em C

- Mutex/Locks:

Operation	
Initialize a mutex	`pthread_mutex_init()`
Make mutex consistent	`pthread_mutex_consistent_np()`
Lock a mutex	`pthread_mutex_lock()`
Unlock a mutex	`pthread_mutex_unlock()`
Lock with a nonblocking mutex	`pthread_mutex_trylock()`
Destroy a mutex	`pthread_mutex_destroy()`

Concorrência em C

- Variáveis de condição:

Operation	
Initialize a condition variable	`pthread_cond_init()`
Block on a condition variable	`pthread_cond_wait()`
Unblock a specific thread	`pthread_cond_signal()`
Block until a specified event	`pthread_cond_timedwait()`
Unblock all threads	`pthread_cond_broadcast()`
Destroy condition variable state	`pthread_cond_destroy()`

Concorrência em Python

- **Lock** objects
  - Semelhantes a mutex
  - Métodos **acquire()** e **release()**
- **RLock** objects
  - Reentrant Locks
  - Locks associados a *thread*
  - Métodos **acquire()** e **release()**
- **Condition** objects
  - Tem um **Lock** ou **RLock** associado
  - Métodos **wait()**, **notify()**, **notify_all()**, **acquire()** e **release()**

≠ ⇒ Ele verifica se o dono da Thread é quem está a fazer lock() e permite um segundo lock (não bloqueia)

o lock é libertado

Concorrência em Python

- **Semaphore** objects
  - Métodos **acquire()** e **release()**
- **Event** objects
  - Métodos **set()**, **clear()** e **wait()**
- **Timer** objects
  - Método **start()**
- **Barrier** objects
  - Método **wait()**