Programação Concorrente(7)

José C. Cunha, DI/FCT/UNL

- Programação concorrente usando monitores

Aviso

Os *slides* seguintes estão em Inglês!

Cooperation → Communication

(1) Logically Shared Memory:

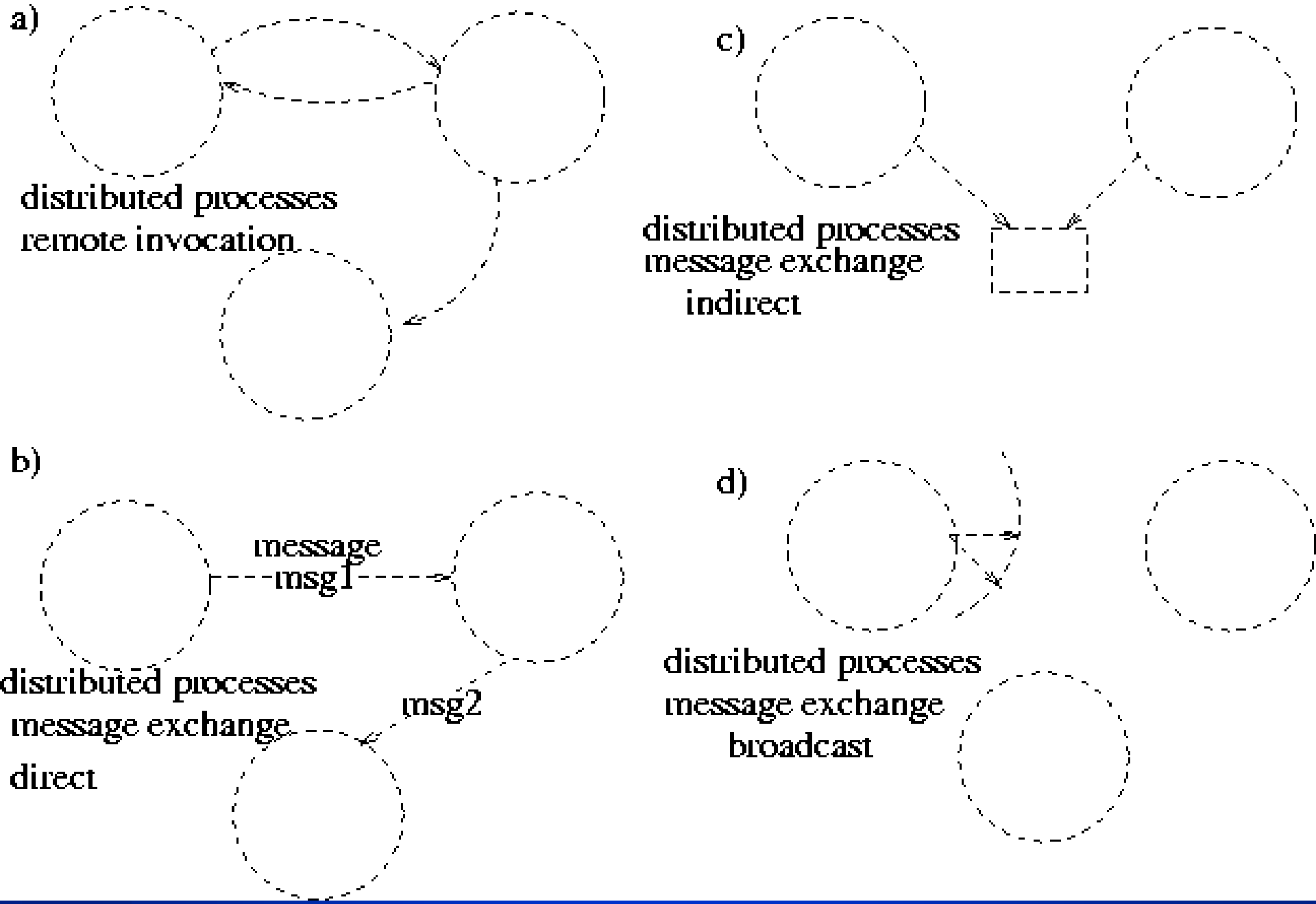
Access to Shared Variables

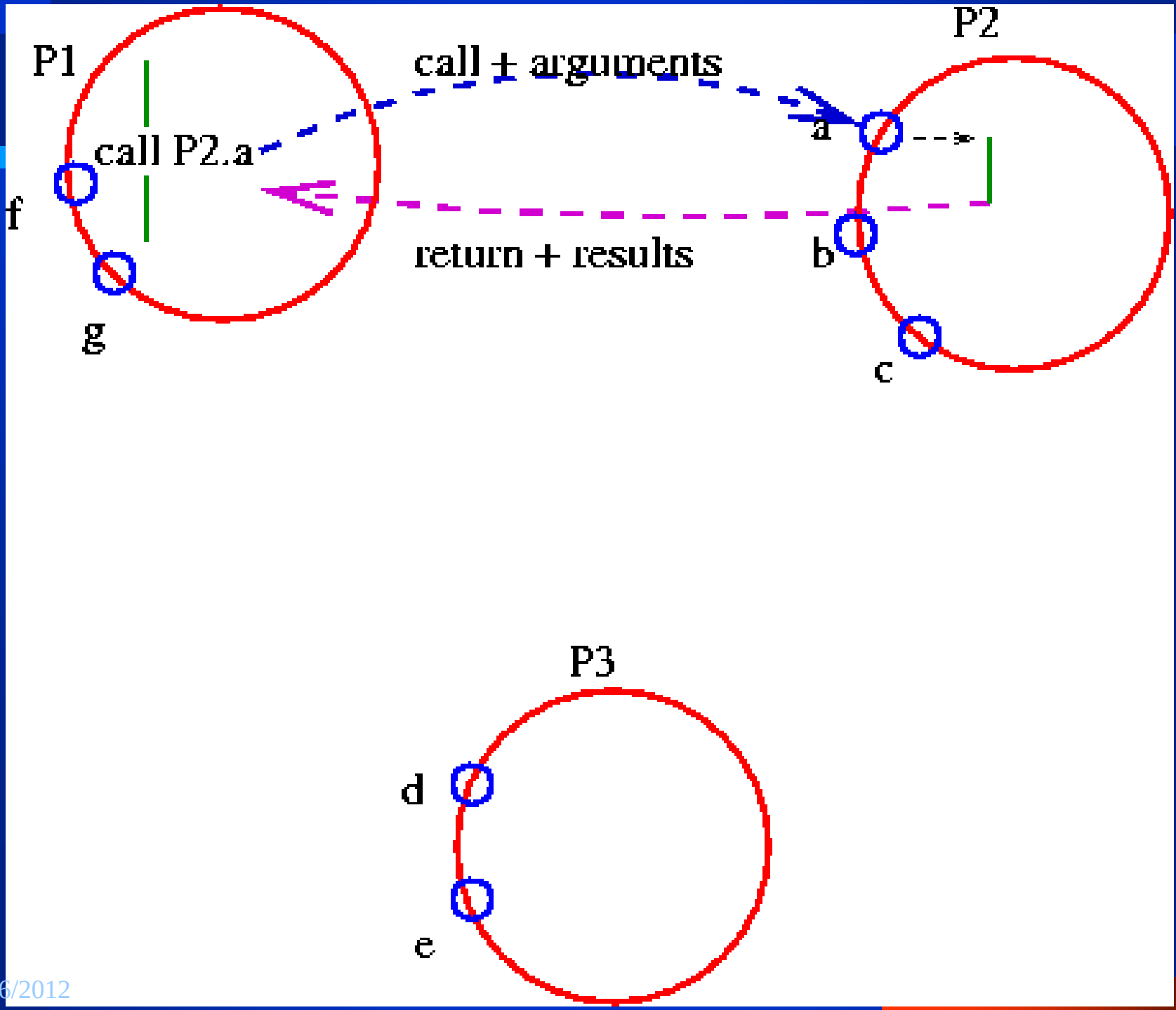
(2) Distributed Memory:

Copying bytes/messages through communication channels or message queues / mailboxes

Or

Invoking Remote Methods in other Objects


a)
distributed processes
remote invocation
c)
distributed processes
message exchange
indirect
b)
message
msg1
distributed processes
message exchange
direct
msg2
d)
distributed processes
message exchange
broadcast


P1
call + arguments
P2
call P2.a
a
f
return + results
b
g
c
P3
d
e

(1) Shared Memory

- **Supported by physically-shared memory**

specific hardware architectures (mono- or multi-processors)

(also supported by logically-shared memory

on physically-distributed memory architectures,

more on this later…!)

Shared-Memory Models

Concurrency control models:

-- **mutual exclusion / atomic operations**

-- **readers and writers / shared readings**

-- **clients and servers**

Mutex locks; Semaphores;

Monitors; conditions

Shared structures appear in different forms:

-- **Shared variables**

-- **Shared objects**

-- **Blackboards**

Semáforo

- **Mecanismo ´´universal´´: resolve quase todos os problemas de sincronização**

É um mecanismo muito elementar e não estruturado:

-- um programa concorrente baseado em semáforos pode-se tornar difícil de compreender

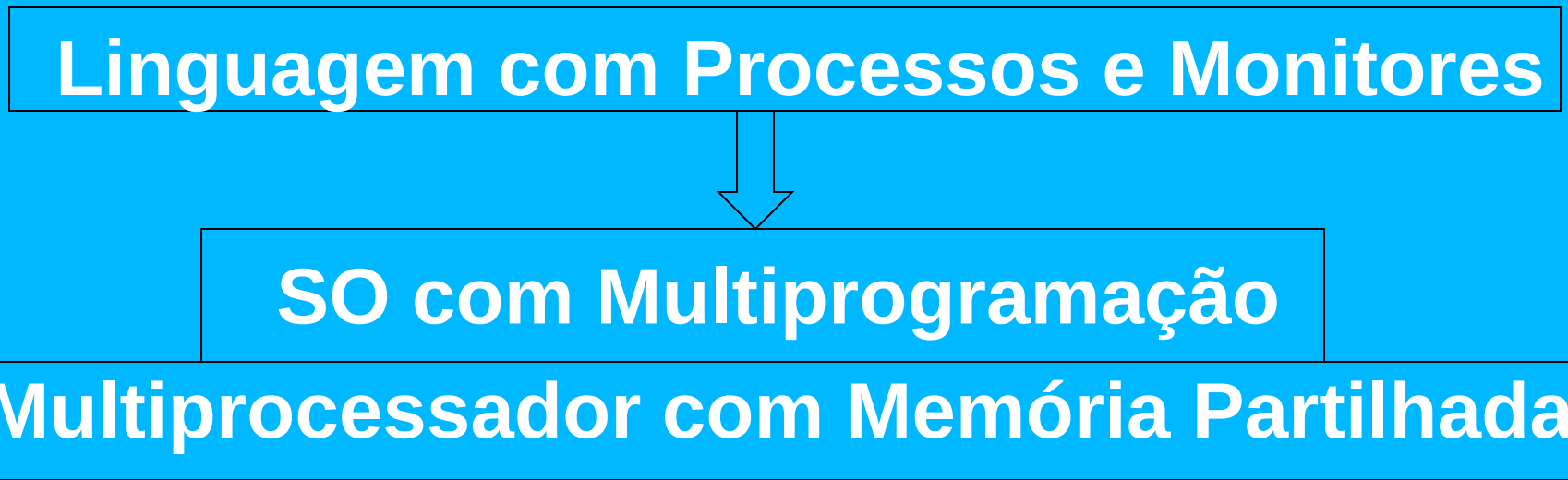
-- a omissão de um P() ou de um V() torna o programa incorrecto e o erro pode ser difícil de detectar, pois pode haver, dispersos nos programas dos processos, múltiplos P() e V() sobre um mesmo semáforo

-- a utilização conjunta de fork/exec/semáforos torna os programas concorrentes difíceis de compreender

Monitores de sincronização

Uma abordagem estruturada para sincronizar processos concorrentes no acesso a memória partilhada

Ao nível de uma Linguagem de Programação (acima do nível do SO)



Language constructs for process synchronisation

Goal: structured programming constructs in high-level concurrent programming languages for shared-memory systems

Main benefits:

Allow the compiler to verify erroneous concurrent accesses

Hide low-level/SO synchronisation code

Monitors are

Programming Language constructs

Language level: Processes and Monitors

Compiler / RuntimeSystem

OS level: Processes and Semaphores

Synchronisation using Monitors

- **Modules → Passive Objects**

control/centralise access to shared memory

First appeared in Concurrent Pascal (1974-5)

(Processes, Classes and Monitors)

-- Per Brinch Hansen

-- Also proposed by C.A.R. Hoare

- **(Re-)appeared in JAVA**

A first attempt

Conditional Critical Regions

- **A 1st attempt to help writing structured concurrent programs with high-level constructs:**

-- concurrent processes and shared variables

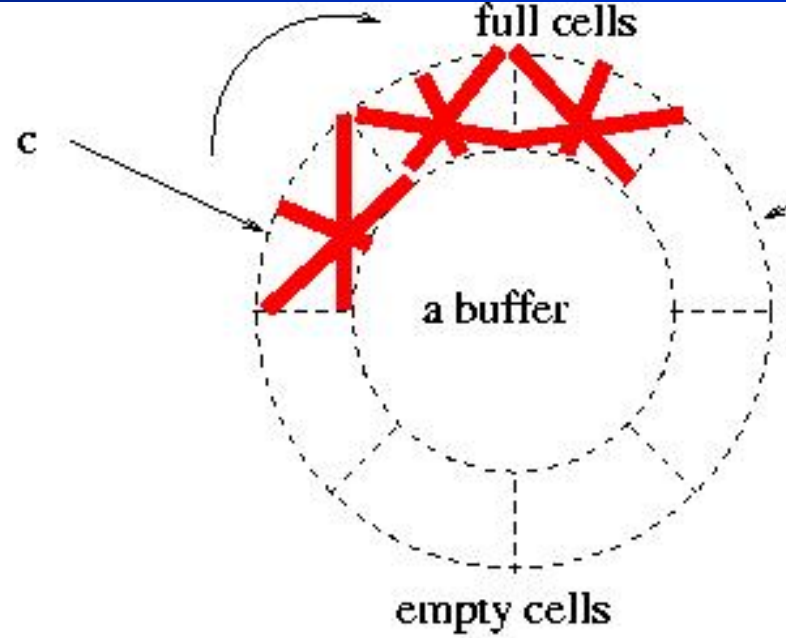
-- Declaration of Critical Regions:

REGION sharedVar DO BEGIN…END;

-- Conditional synchronisation: AWAIT condition;

-- the invoker blocks until condition is TRUE

-- the Runtime System automatically unblocks processes… no need to invoke P or V operations



Conditional Critical Regions

Example:
the EDISON language
for shared–memory multiprocessors

```
type B = shared record
           buffer: array 0..max–1 of T
           p, c: 0..max–1
           full: 0..max
         end

procedure Put (m: T; var b: B);
        region b do
          begin
            AWAIT full < max;
            buffer[p] := m;
            p := (p+1) mod max;
            full := full + 1
          end;
```

```
procedure Get (var m: T; var b: B);
        region b do
          begin
            AWAIT full > 0;
            m := buffer[c];
            c := (c+1) mod max;;
            full := full – 1
          end;
```

Comments

- Expressive and clear programming model
- Implementation is complex:

Whenever any process changes any shared variable, the runtime support must trigger the evaluation of all conditions in all currently blocked processes

Conditions may include shared variables and local variables, private to each process

Implemented in an experimental language (EDISON) for multiprocessors

A second attempt

A compromise:

-- a structured high-level programming style

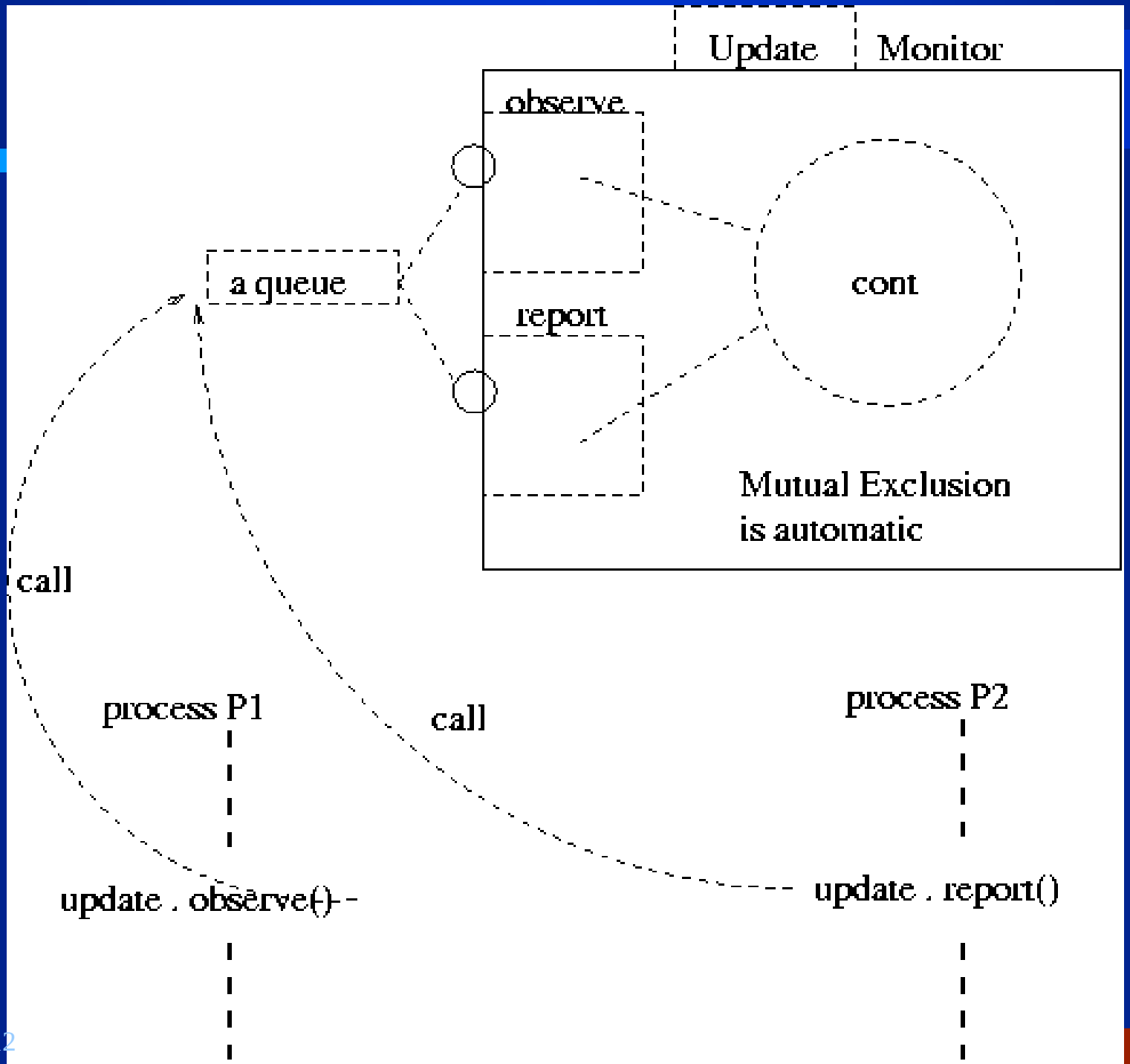
but

-- requires some explicit information by the programmer to unblock only some of the processes in the modified conditions

Synchronisation Monitors

```
Monitor_name: monitor
  begin  …. Declarations of shared variables
    → procedure entry Proc_name(…);
        begin …. end;
    → … other procedure entries….
        … other internal functions …
    … initialisation of monitor variables….
  end;
```

Monitor = a module that encapsulates a shared data structure and its operations


Update
Monitor
observe
report
a queue
cont
Mutual Exclusion
is automatic
call
process P1
call
process P2
update . observe()
update . report()

Update: monitor

```
{ cont: integer;                    /*shared */
  → procedure entry observe;
        cont = cont + 1;
  → procedure entry report;
    {  print(cont);
       cont = 0;
    }
  { /*monitor body, executed initially*/
     cont = 0;
  }
}
```

Acesso exclusivo a um recurso

Resource Manager

- A critical resource (R): mutual exclusion

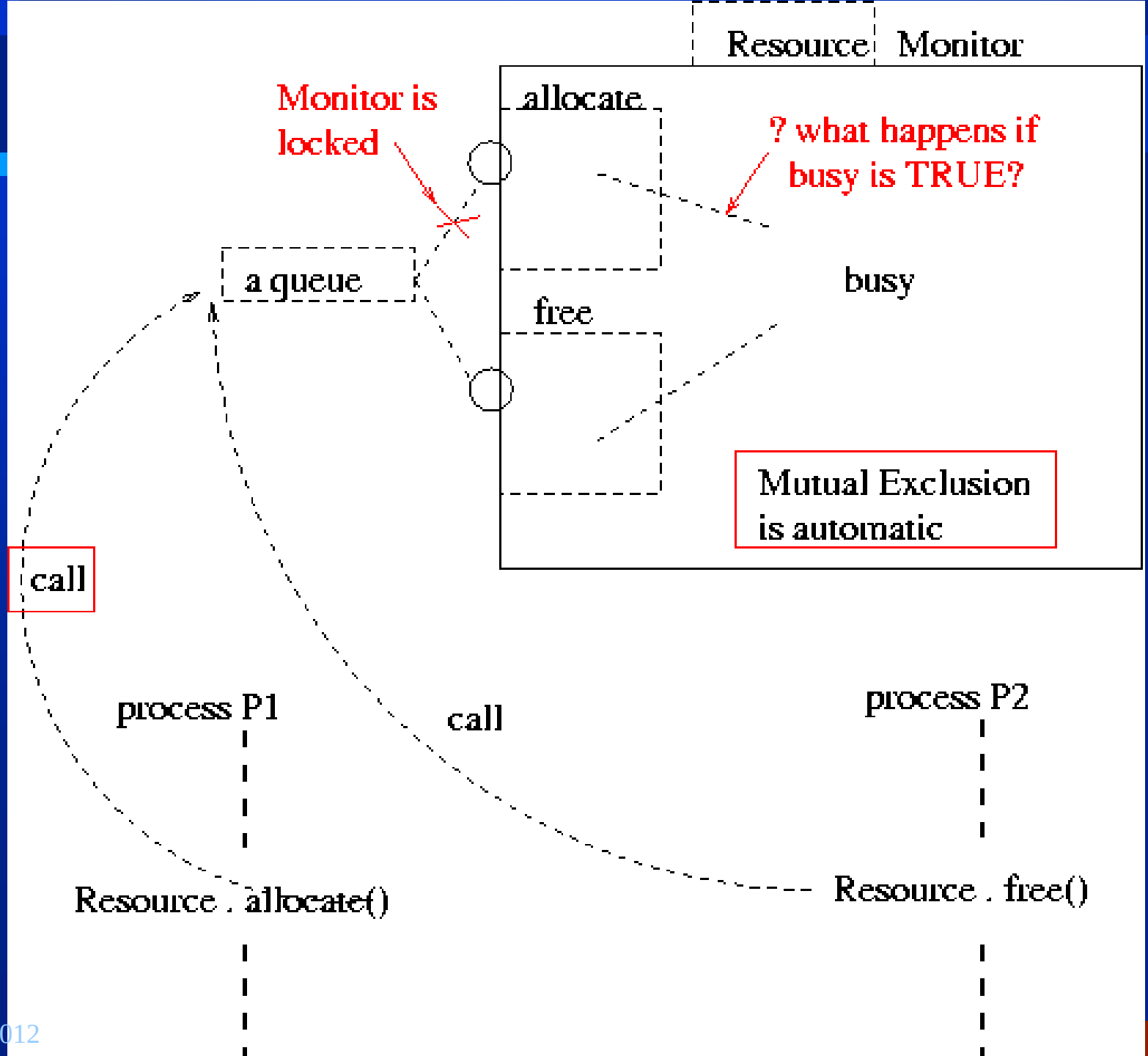
Primitives Allocate () / Free ()

- User Processes:

```
.....
Allocate();
access resource;
Free();
```

- How to implement this, using Monitors?

```
Resource: monitor;
{ busy: boolean;                    /*shared */
  → procedure entry allocate;
    { if busy then …..??…
      else busy = true;
    }
  → procedure entry free;
    { busy = false;}
{/*monitor body, executed initially*/
  busy = false;   / * the resource is free*/
}
```


Resource Monitor
Monitor is locked
allocate
? what happens if busy is TRUE?
a queue
busy
free
Mutual Exclusion is automatic
call
process P1
call
process P2
Resource . allocate()
Resource . free()

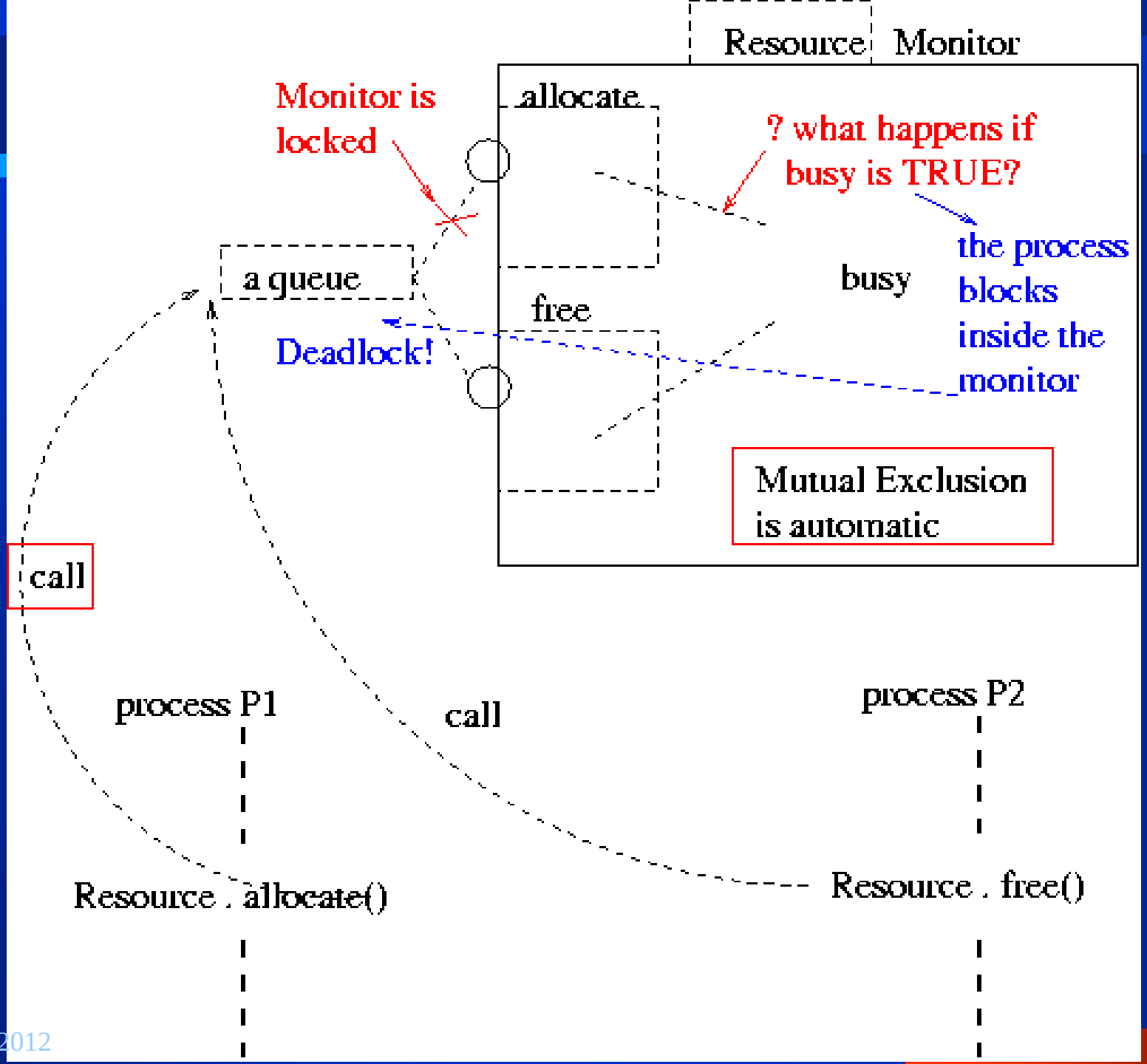
Alternative Solutions?

a) **Allocate returns immediately** even if the resource is busy:

- the invoker must enter a busy waiting loop

→ NOT a good solution.

b) The invoker **blocks** inside **Allocate** until the resource is released….


Resource Monitor
allocate
Monitor is locked
? what happens if busy is TRUE?
the process blocks inside the monitor
busy
a queue
free
Deadlock!
Mutual Exclusion is automatic
call
process P1
call
process P2
Resource . allocate()
Resource . free()

b) The process must release the mutual exclusion to the monitor, before blocking.

A new type of synchronisation variables:

CONDITION VARIABLES

Declared as: var free: condition;

--- defines a process queue (initially empty)

Accessed by two operations:

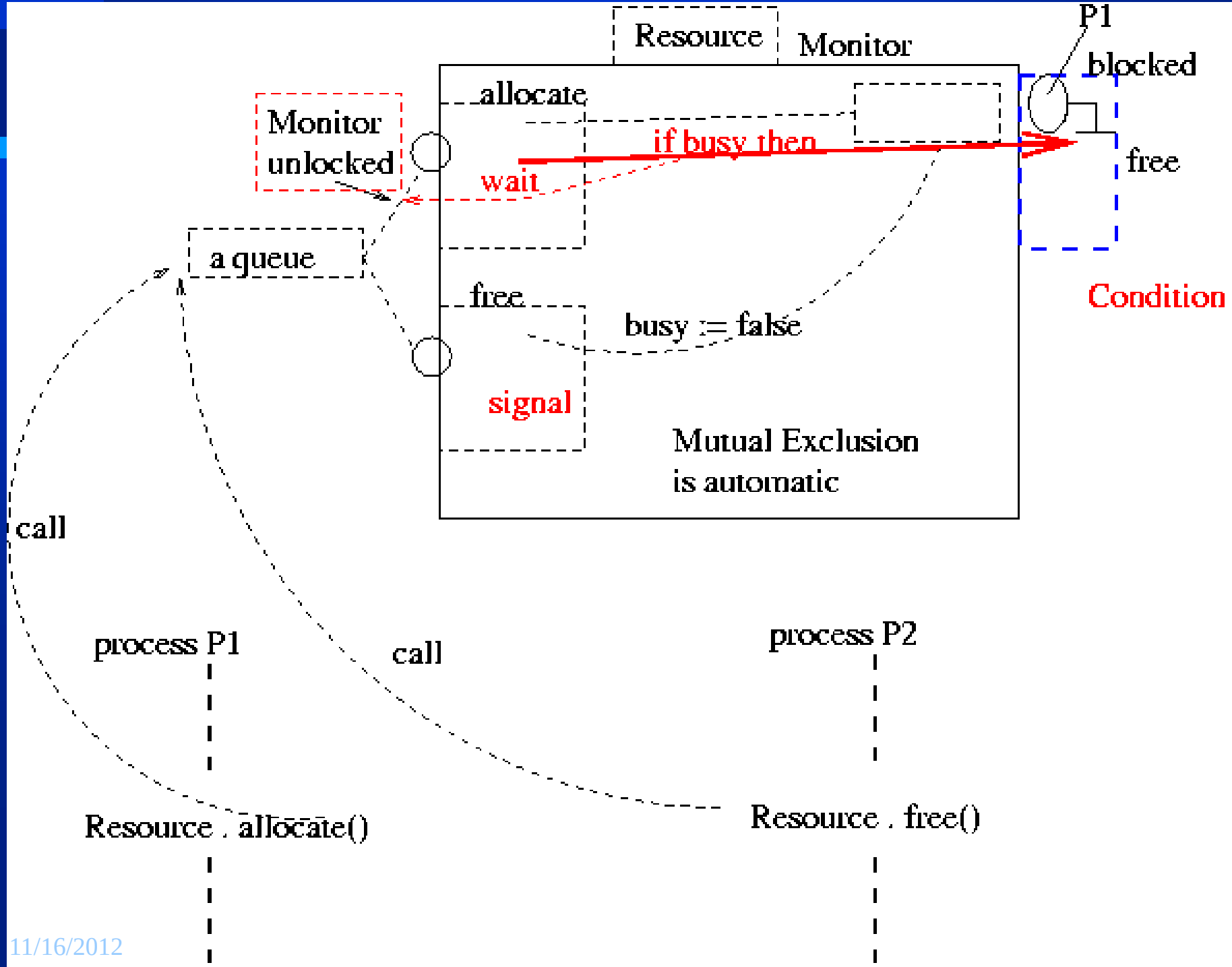
- **To block inside the monitor: WAIT(free);**
  automatically releases the monitor lock, then the process blocks → done atomically
- **To wake up some blocked process: SIGNAL(free) (null if no process blocked)**

Aviso

As primitivas dos monitores WAIT e SiGNAL sobre condições <u>não têm relação nenhuma</u> com as chamadas ao SO Unix com os mesmos nomes

```
Resource: monitor;
{ busy: boolean;
  free: condition;
  → procedure entry allocate;
    { if busy then WAIT(free);      ← blocks
      busy = true;}

  → procedure entry free;
    {   busy = false;
        SIGNAL(free);}              ← awakes one
 { busy = false; /*monitor body*/ }
}
```


Resource
Monitor
P1
blocked
allocate
Monitor
unlocked
if busy then
wait
free
a queue
Condition
free
busy := false
signal
Mutual Exclusion
is automatic
call
process P1
call
process P2
Resource . allocate()
Resource . free()

Barreira de sincronização

Synchronisation Barrier

- N processes synchronise in a collective ´´rendez-vous´´:

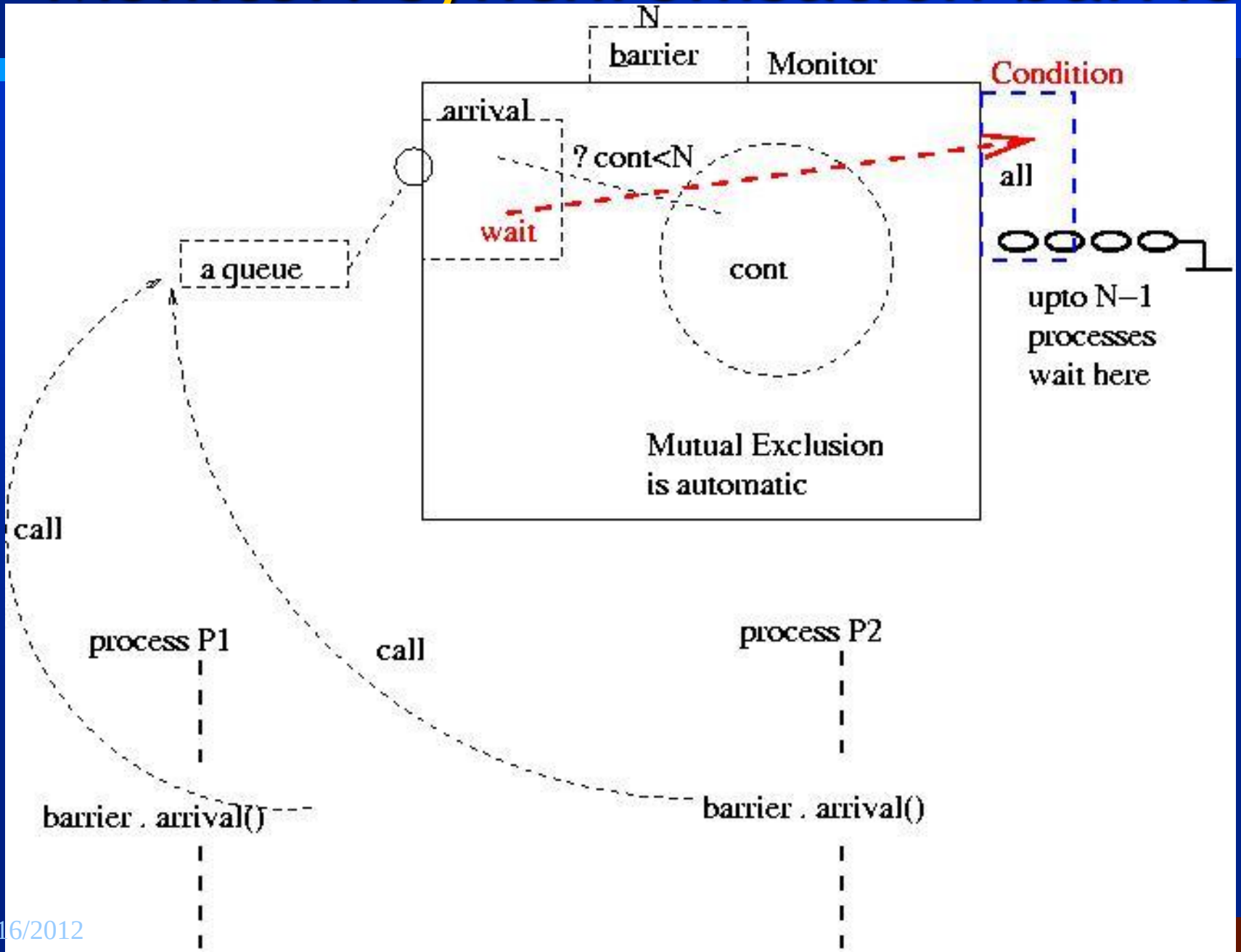
Each process:

```
....
arrival()   /* waits for all the others*/
....        /* proceeds, after all have arrived*/
```

N-Barrier: based on semaphores

```
var count: integer;   /* inicialmente 0*/
    exmut: semaphore; /* inicial/ 1 */
    block: semaphore; /*inicial/ 0*/
arrival()
{  P(exmut); count+ +;
 if (count<N) then {V(exmut); P(block);
   P(exmut)};
 count- -; if (count > 0) then V(block);
 V(exmut);
}
```

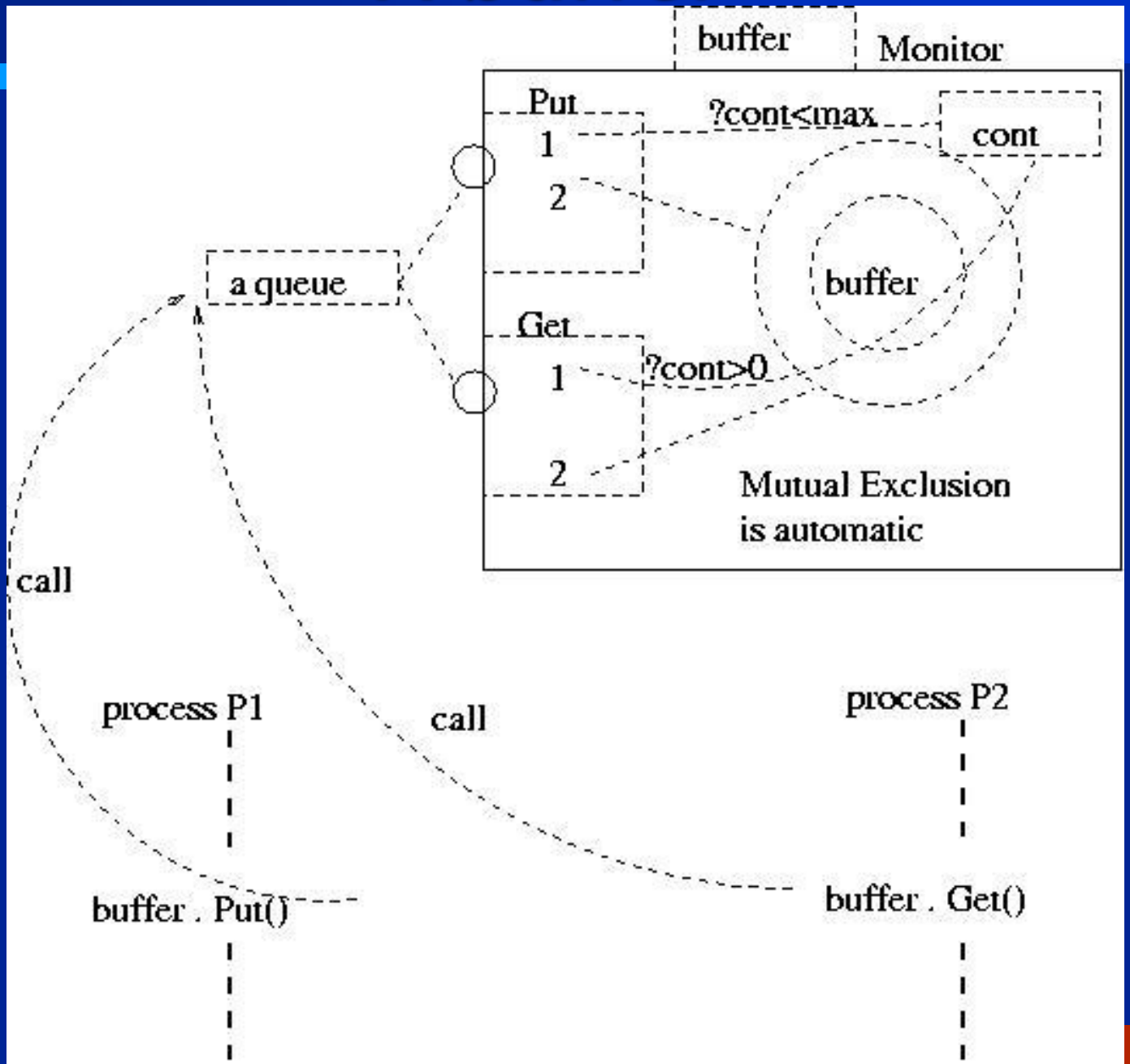
Monitor: synchronisation barrier



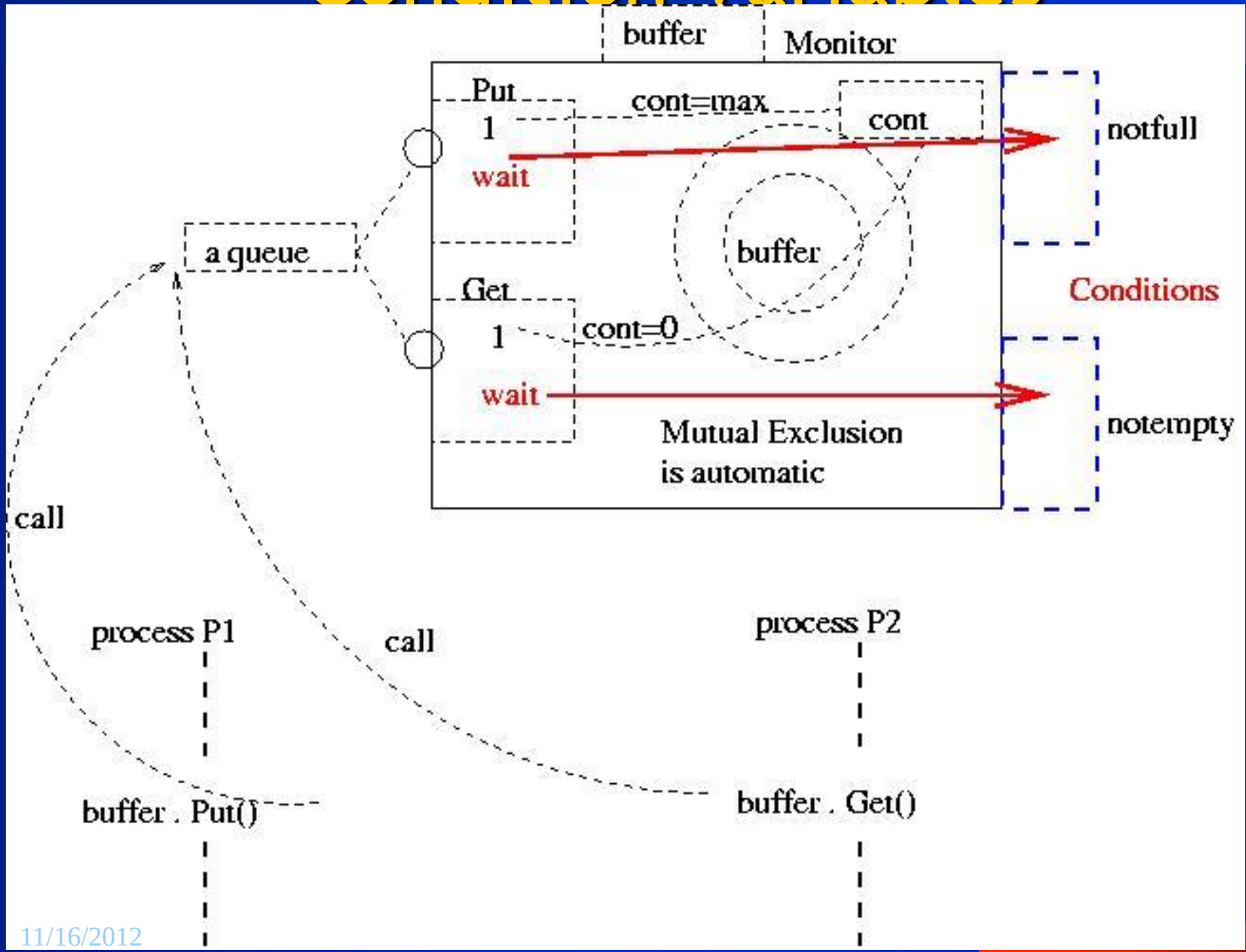
```
barrier: monitor;
{       cont: integer;
        all: condition;
→ procedure entry arrival;
  { cont = cont + 1;
    if cont < N then WAIT(all);
    cont = cont – 1;
    SIGNAL(all);
  }
  { cont = 0; }
}
```

Buffer em memória partilhada

A buffer

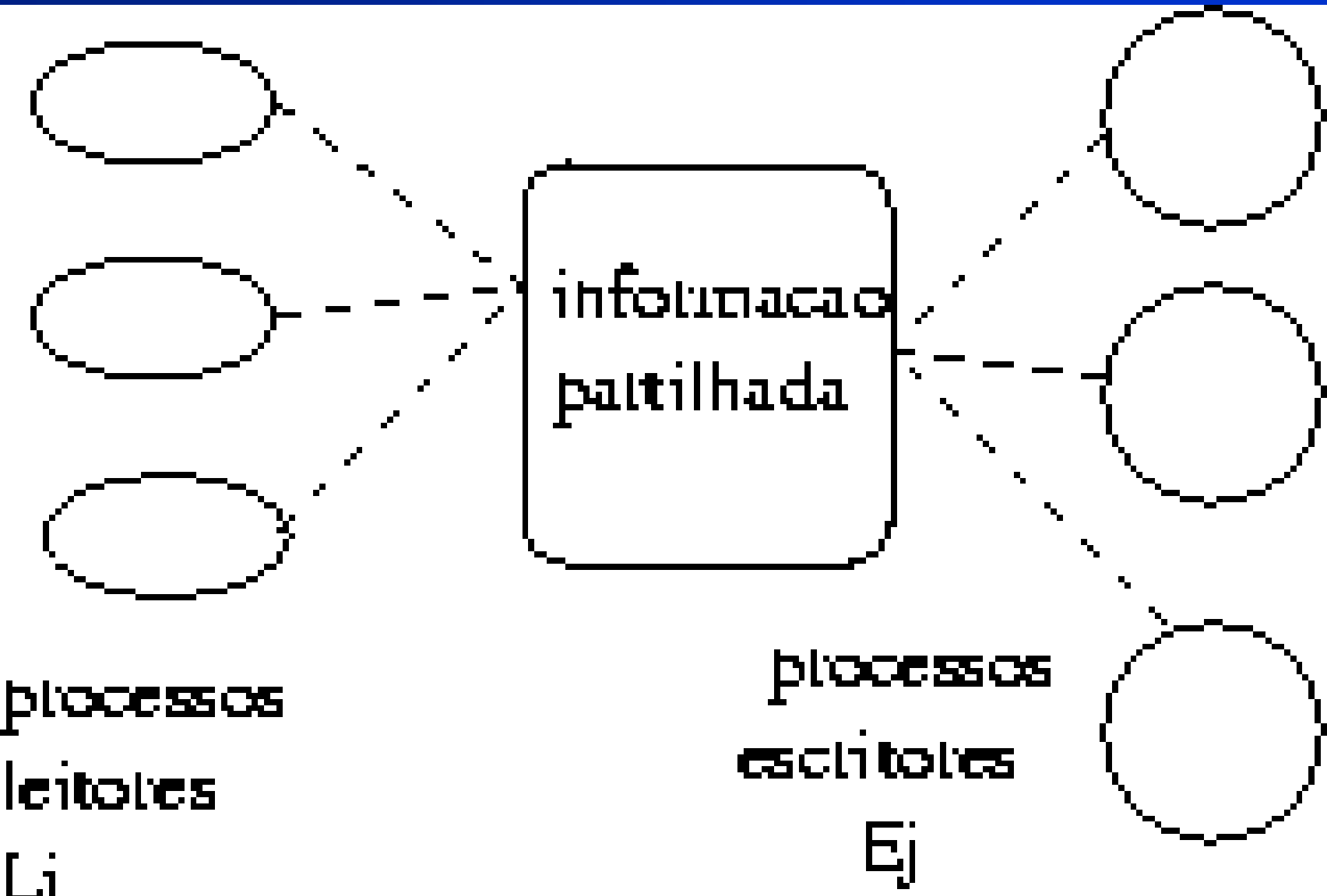

buffer
Monitor
Put
1
2
?cont<max
cont
buffer
a queue
Get
1
?cont>0
2
Mutual Exclusion
is automatic
call
process P1
call
process P2
buffer . Put()
buffer . Get()

Condition Variables



```
buffer: monitor;
{buffer... cont...; notfull, notempty: condition;
→ procedure entry Put(...);
  {   if cont  = max then WAIT(notfull);
      ... Put item into buffer …
      SIGNAL(notempty);}
→ procedure entry Get(...);
  {   if cont = 0 then WAIT(notempty);
      … Get item from buffer …
      SIGNAL(notfull); }
  { cont = 0;}
}
```

Leitores/Escritores acedem a uma BD


informacao
partilhada
processos
leitores
Li
processos
escritores
Ej

Leitor:

```
Pede-para-Ler();  (1)
  Leitura;
Termina-de-Ler(); (2)
```

Escritor:

```
Pede-para-Escrever(); (3)
  Escrita;
Termina-de-Escrever(); (4)
```

- **Como programar as acções 1, 2, 3 e 4?**

variaveis globais partilhadas

```
var R: integer; inicialmente 0
    exmut, Wsem: semaphore. inicial 1
```

Processo Leitor Li

```
WHILE true DO
BEGIN
    P(exmut);
    R := R + 1;
    IF R=1 THEN P(Wsem);
    V(exmut);
    leitura;
    P(exmut);
    R := R – 1;
    IF R=0 THEN V(Wsem);
    V(exmut);
END;
```

Processo Escritor Ei

```
WHILE true DO
BEGIN
    P(Wsem);
    escrita;
    V(Wsem);
END;
```

– Solução baseada num Monitor

```
Leitores_Escritores: monitor;
{       R: integer; W: boolean;
        OK_to_R, OK_to_W: condition; /*filas*/
 → procedure entry Start_R; {…}
 → procedure entry End_R; {…}
 → procedure entry Start_W; {…}
 → procedure entry End_W; {…}
 { R = 0;
   W = false; } /*inicialização*/
}
```

Um Leitor

```
Start_R;
lendo();
End_R;
```

Um Escritor

```
Start_W;
escrevendo();
End_W;
```

- R integer contador de Leitores
- W boolean indica 1 Escritor dentro
- condition OK_to_W

  onde aguardam os escritores
- condition OK_to_R

  onde aguardam os leitores

Start_R:

```
{ if (W or non_empty(OK_to_W))
                then WAIT(OK_to_R);
  R = R + 1;
 SIGNAL(OK_to_R);
}
```

End_R:

```
{ R = R – 1;
  if (R == 0) then SIGNAL(OK_to_W);
}
```

```
Start_W:
{ if ((R > 0) or W)
                then WAIT(OK_to_W);
  W = true;
}

End_W:
{ W = false;
  if (non_empty(OK_to_R))
          then SIGNAL(OK_to_R);
          else  SIGNAL(OK_to_W);
}
```

Uma ´´ponte´´ com uma única faixa mas com tráfego nos dois sentidos

Quando um carro chega a uma entrada:

-- só pode entrar se uma das condições for satisfeita:

- A ponte está desocupada

ou

- O tráfego corrente desloca-se no mesmo sentido que o do carro que acabou de chegar

Hipótese simplificadora: ponte ilimitada!

```
E:  P(exE);
    contE++;
    if (contE==1)
       then P(block);
    V(exE);
    na ponte();
    P(exE);
    contE--;
    if (contE==0)
      then V(block);
    V(exE);
```

```
D:  P(exD);
      contD++;
       if (contD==1)
         then P(block);
      V(exD);
     na ponte();
      P(exD);
       contD--;
         if (contD==0)
         then V(block);
      V(exD);
```

Solução baseada num Monitor

```
ponte: monitor;
{       contE, contD: integer;
        blockE, blockD: condition;
 → procedure entry InE; {…}
 → procedure entry OutE; {…}
 → procedure entry InD; {…}
 → procedure entry OutD; {…}
 { contE = contD = 0;  } /*inicialização*/
}
```

Carro entrando pelo lado esquerdo

```
InE;
na ponte();
OutE;
```

Carro entrando pelo lado direito

```
InD;
na ponte();
OutD;
```

InE:

```
{ if (contD>0)
    then WAIT(blockE);
  contE++;
  SIGNAL(blockE);
}
```

InD:

```
{ if (contE>0)
    then WAIT(blockD);
  contD++;
  SIGNAL(blockD);
}
```

OutE:

```
{ contE--;
  if (contE==0)
  then SIGNAL(blockD);
}
```

OutD:

```
{ contD--;
  if (contD==0)
  then SIGNAL(blockE);
}
```

Invariantes dos monitores

Monitor invariants: expressions that are always true: before and after invoking monitor entries

Example: Buffer

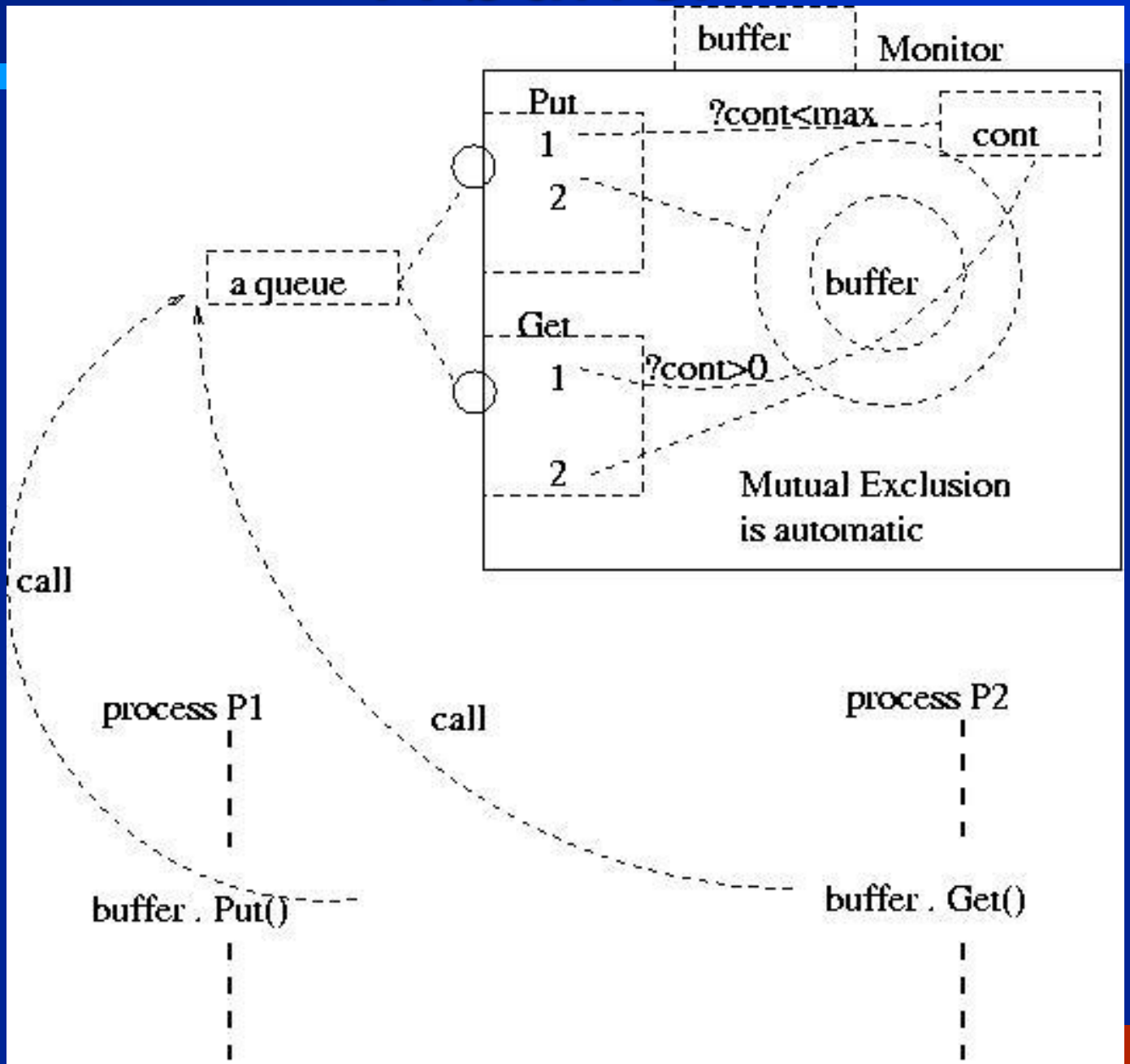
```
var next_in, next_out: 0..BUFSIZE-1;
    cont:0..BUFSIZE;
```

- Initially: next_in = next_out = 0
  cont = 0;
- next_in = (next_out + cont) mod BUFSIZE
- (cont >= 0) AND (cont <= BUFZISE)

notfull → associated to cont < BUFSIZE

notempy → associated to cont > 0

A buffer


buffer
Monitor
Put
1
2
?cont<max
cont
buffer
a queue
Get
1
?cont>0
2
Mutual Exclusion
is automatic
call
process P1
call
process P2
buffer . Put()
buffer . Get()

Invariants help the programmer reasoning about correctness of the monitor implementation

Pre-condition { ..operation... } **Post-condition**

Invariants help the programmer reasoning about correctness of the monitor implementation

Pre-condition { ..operation... } Post-condition

TRUE {monitor initialisation} Invariant

Invariants help the programmer reasoning about correctness of the monitor implementation

Pre-condition { ..operation... } Post-condition

TRUE {monitor initialisation} Invariant

- **On a WAIT(c) for expression B to become TRUE:**

  Invariant {WAIT(c) operation} ?

Invariants help the programmer reasoning about correctness of the monitor implementation

Pre-condition { ..operation... } Post-condition

TRUE {monitor initialisation} Invariant

- On a WAIT(c) for expression B to become TRUE:

  Invariant {WAIT(c) operation} (Invariant AND B)

Invariants help the programmer reasoning about correctness of the monitor implementation

Pre-condition { ..operation... } Post-condition

TRUE {monitor initialisation} Invariant

- On a WAIT(c) for expression B to become TRUE:

  Invariant {WAIT(c) operation} (Invariant AND B)

- Before performing a SIGNAL(c), expression B must be guaranteed TRUE by the signaller:

  (Invariant AND B) {SIGNAL(c)} ?

Invariants help the programmer reasoning about correctness of the monitor implementation

Pre-condition { ..operation... } Post-condition

TRUE {monitor initialisation} Invariant

- On a WAIT(c) for expression B to become TRUE:

  Invariant {WAIT(c) operation} (Invariant AND B)

- Before performing a SIGNAL(c), expression B must be guaranteed TRUE by the signaller:

  (Invariant AND B) {SIGNAL(c)} Invariant

Invariants help the programmer reasoning about correctness of the monitor implementation

Pre-condition { ..operation... } Post-condition

TRUE {monitor initialisation} Invariant

- **On a WAIT(c) for expression B to become TRUE:**

  Invariant {WAIT(c) operation} (Invariant AND B)

- **Before performing a SIGNAL(c), expression B must be guaranteed TRUE by the signaller:**

  (Invariant AND B) {SIGNAL(c)} Invariant

Immediately AFTER the SIGNAL operation executes:

-- a blocked process in c that completes WAIT, MUST find expression B still TRUE….

→>>>>>> How can this be guaranteed?

Immediate reactivation requirement

After a SIGNAL(c) is executed by process S:

Two active processes are inside the monitor:

the signaller (S) and the signalled process (A)

Immediate reactivation requirement

After a SIGNAL(c) is executed by process S:

Two active processes are inside the monitor:

the signaller (S) and the signalled process (A)

-- **One of them (S or A) must exit the monitor immediately**

Immediate reactivation requirement

After a SIGNAL(c) is executed by process S:

Two active processes are inside the monitor:

the signaller (S) and the signalled process (A)

-- One of them (S or A) must exit the monitor immediately.

-- No other process (C) should enter before A resuming execution inside the monitor:

Immediate reactivation requirement

After a SIGNAL(c) is executed by process S:

Two active processes are inside the monitor:

the signaller (S) and the signalled process (A)

-- One of them (S or A) must exit the monitor immediately.

-- No other process (C) should enter before A resuming execution inside the monitor:

otherwise, C may change the condition that S has signalled and then A will not find it TRUE anymore!

Distinct monitor semantics

Preemptive: the signalled (A) is immediately reactivated: A is the next process inside the monitor

- **Signal & Exit:** the signaller exits the monitor
  (**Concurrent Pascal**, Brinch Hansen)
- **Signal & Wait:** the signaller suspends inside the monitor, but waits its turn in the entry queue
  (**MODULA-1**; **Concurrent Euclid**)
- **Signal & Urgent Wait:** the signaller suspends inside the monitor → has higher priority than others in the monitor entry queue
  (**Pascal-Plus**; C.A.R.Hoare)

Non-preemptive: the signaller (S) continues execution inside the monitor and the signalled (A) is forced to (temporarily) exit the monitor and retry the monitor call:

- **Signal & Continue:** the signalled (A) exits the monitor and is forced to retry the monitor call

  (**MESA**, Xerox PARC)

A scheduling strategy based on priorities tries to preserve fairness….

Different programming models

- **Signal & Exit: SIGNAL is always the last operation inside a monitor procedure**

Different programming models

- **Signal & Exit: SIGNAL is always the last operation inside a monitor procedure**
- **Signal & Wait: a signaller may invoke multiple SIGNAL operations inside the monitor**

Different programming models

- **Signal & Exit: SIGNAL is always the last operation inside a monitor procedure**
- **Signal & Wait: a signaller may invoke multiple SIGNAL operations inside the monitor**
- **Signal & Continue: a signalled process must always be programmed to allow retrying the monitor call**

Different programming models

- **Signal & Exit: SIGNAL is always the last operation inside a monitor procedure**
- **Signal & Wait: a signaller may invoke multiple SIGNAL operations inside the monitor**
- **Signal & Continue: a signalled process must always be programmed to allow retrying the monitor call**

→ example of monitors in the MESA language; use priorities to decide which process enters the monitor → example:

Allocate(n) block to get n resource units from a pool

Free(n) releases the n resource units to the pool

```
resource_manager: monitor;  -- in MESA
{   total: integer;   free: condition;
 → procedure entry allocate(n: integer);
 {  if n > total then WAIT(free); -- exit and retry automatically!
    else total = total – n;          -- get the requested n units
 }
 → procedure entry free(n: integer);
 {  total = total + n;                -- release the n units
    BROADCAST(free);  -- wake ALL blocked processes in free
 }
   { total = MAX;}
}
```

Implementation of monitors

- **Monitors are Programming Language Constructs, NOT Operating System calls…**
- **The language compiler and the runtime support must provide for code generation and implementation of 4 basic primitives:**

-- enter_monitor

-- exit_monitor

-- wait_condition

-- signal_condition

Semaphore-based implementation

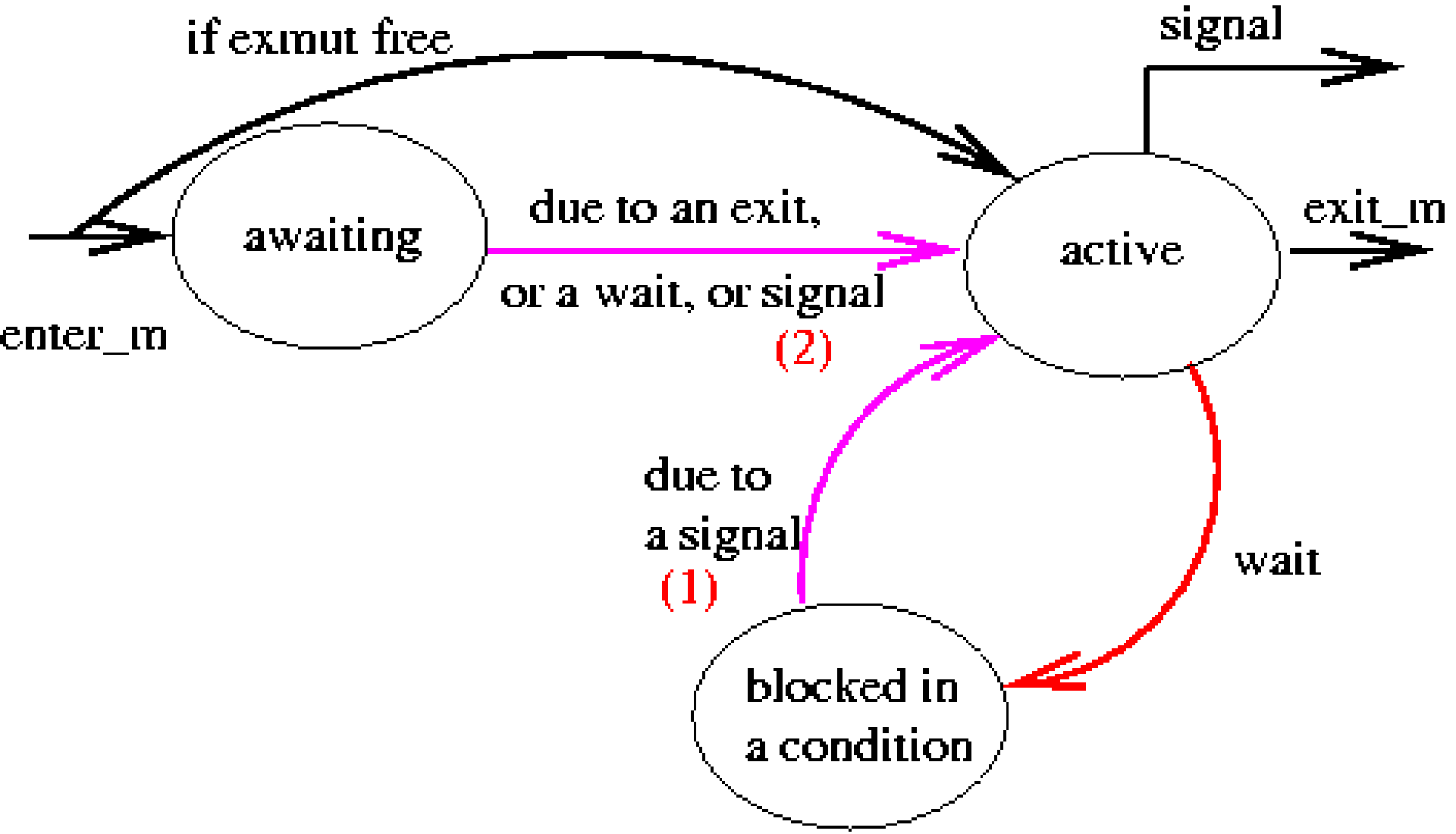
Monitors can be implemented on top of Semaphores:

Language level: Processes and Monitors

Compiler / RuntimeSystem

OS level: Processes and Semaphores

Signal&Exit Monitors



Process states and transitions for a Signal&Exit monitor type

Semaphore-based implementation

Monitors can be implemented on top of semaphores

exmut: semaphore; /*initially 1*/

Conditions

→ **represented by a record / structure with two fields:**

```
-- sem_cond: semaphore;
                    /*initially 0, for blocking processes*/
-- cont: integer;
                    /*initially 0, for counting blocked
                      processes*/
```

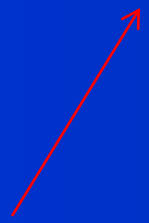
Signal&Exit Monitors – generated code

For a Waiter...

```
enter_monitor;
  ...
wait_condition(C);
  ...
exit_monitor;
```

For a Signaller ...

```
enter_monitor;
        ...
signal_condition(C);
& exit_monitor;
```



```
enter_monitor: P(exmut);
```

```
wait(c);
 {  c.cont ++;
      V(exmut);
      P(c.sem_cond);
    c.cont --;
  }
```

```
signal&exit(c);
  { if c.cont>0 then
      V(c.sem_cond);
    else
      V(exmut);
  }
```

```
exit_monitor: V(exmut);
```

Signal&UrgentWait Monitors(1)

```
enter_monitor;
    ...
wait_condition( );
    ...
    ...
exit_monitor;
```

```
enter_monitor;
    ...
signal_condition( );
    ...
    ...
exit_monitor( );
```

Signal&UrgentWait Monitors(2)

```
urgent: int;            /*initial 0*/ -- # signallers waiting
urgent_sem: semaphore;  /*   signallers to wait   */
```

In Wait() and Exit(), we must test
if urgent>0: a signaller must be unblocked
before any new process may enter

```
Enter_monitor: ---  P(exmut);
```

```
Wait(c):
   c.cont ++;
   if urgent > 0 then V(urgent_sem); else V(exmut);
   P(c.sem_cond);
   c.cont --;
```

Signal&UrgentWait Monitors(3)

Signal(c):

```
urgent ++;
if  c.cont>0 then
      {   V(c.sem_cond);
          P(urgent_sem);
      };
 urgent --;
```

Exit_monitor():

```
if urgent > 0 then V(urgent_sem); else V(exmut);
```

Implementing Semaphores using Monitors!

A Monitor to implement semaphores(1)

```
semaphore: monitor;                    --Signal&Exit Monitor
{ sem_value: int; sem_queue: condition;
  → procedure entry P;
    { sem_value = sem_value – 1;
      if sem_value <0 then wait(sem_queue);
    }
  → procedure entry V;
    { sem_value = sem_value + 1;
      if sem_value <= 0 then signal(sem_queue);
    }
  { /*initially*
    sem_value = …./*semaphore initial value*/
  }}
```

Another Monitor to implement semaphores(2)

```
semaphore: monitor;                    --Signal&Exit Monitor
{  sem_value: int; sem_queue: condition;
   → procedure entry P;
    {  if sem_value == 0 then wait(sem_queue);
       sem_value = sem_value – 1;
    }
   → procedure entry V;
    { sem_value = sem_value + 1;
      signal(sem_queue);
    }
  { /*initially*
     sem_value = …./*semaphore initial value*/
  }}
```

Conclusões – sobre Monitores

Vantagens?

- **Permite uma programação mais clara**
- **Auxílio do compilador evitando mais erros do que no uso directo das funções do SO**

Desvantagens?

- **Maiores zonas de exclusão mútua**
- **Exclusão mútua nem sempre necessária**

Disponibilidade:

- **Exige suporte a nível de uma linguagem, eg *Java***
- **Suporte a nível de biblioteca de *PThreads* também existe, mas a programação situa-se a um mais baixo nível ...**

Condições no modelo de *PThreads*

Exclusão mútua com Mutex: **pthread_mutex_t**

Variáveis de tipo *Condition*: ***pthread_cond_t***

Principais operações sobre *Conditions*:

```
int pthread_cond_init(pthread_cond_t  *cond,
                      pthread_condattr_t *cond_attr;

int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);

int pthread_cond_signal(pthread_cond_t  *cond);
int pthread_cond_broadcast(
                      pthread_cond_t *cond);
```

Conditions in Pthreads -- POSIX manual

```
int pthread_cond_init( pthread_cond_t       *cond,
                       pthread_condattr_t *cond_attr);

int pthread_cond_wait( pthread_cond_t       *cond,
                       pthread_mutex_t      *mutex);

int pthread_cond_timedwait(…);

int pthread_cond_signal( pthread_cond_t      *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);

int pthread_cond_destroy(pthread_cond_t      *cond);
```

Conditions in PThreads

A condition variable must always be associated with a mutex:

to avoid the race condition where a thread prepares to wait on a condition variable and another thread signals the condition just before the first thread actually waits on it.

Conditions in Pthreads – POSIX manual

pthread_cond_init initializes the condition variable

pthread_cond_destroy destroys a condition variable, freeing the resources it might hold. No threads must be waiting on the condition variable on entrance to pthread_cond_destroy.

An error is returned if it is called when there are such threads.

Conditions in Pthreads – POSIX manual

pthread_cond_wait atomically unlocks the mutex (as per pthread_unlock_mutex) and waits for the condition variable cond to be signalled.

The thread execution is suspended and does not consume any CPU time until the condition variable is signalled. The mutex must be explictly locked by the calling thread on entrance to pthread_cond_wait.

Before returning to the calling thread, pthread_cond_wait automatically re-acquires the mutex (as per pthread_lock_mutex).

Conditions in Pthreads – POSIX manual

Unlocking the mutex and suspending on the condition variable is done atomically by the implementation.

Thus, if all threads always acquire the mutex before signalling the condition,

this guarantees that the condition cannot be signalled (and thus ignored) between the time a thread locks the mutex and the time it waits on the condition variable

Conditions in Pthreads – POSIX manual

pthread_cond_signal restarts one of the threads that are waiting on the condition variable cond.

If no threads are waiting on cond, nothing happens.

If several threads are waiting on cond, exactly one is restarted, but it is not specified which.

pthread_cond_broadcast restarts all the threads that are waiting on the condition variable cond.

Nothing happens if no threads are waiting on cond.

Conditions in PThreads

pthread_cond_timedwait atomically unlocks mutex and waits on cond, as pthread_cond_wait does, but it also bounds the duration of the wait.

If cond has not been signalled within the amount of time specified by abstime, the mutex is re-acquired and pthread_cond_timedwait returns the error ETIMEDOUT.

The abstime parameter specifies an absolute time, with the same origin as time(2) and gettimeofday(2).

O exemplo Produtor/Consumidor em *PThreads*

```
… buffer[MAX];                int cont = 0;
pthread_cond_t NotFull, NotEmpty;
pthread_mutex_t bufMutex;
```

```
Put( item )
{ pthread_mutex_lock(bufMutex);

  while (!(cont<MAX))
      pthread_cond_wait(NotFull, bufMutex);
  putinbuffer( item );

  pthread_cond_signal( NotEmpty );
  pthread_mutex_unlock(bufMutex);
}
```

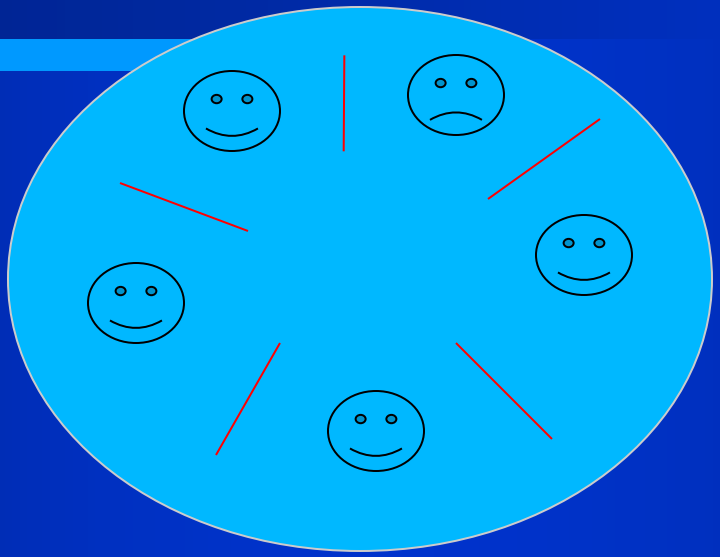
```
Get( Item )
{ pthread_mutex_lock(bufMutex);

  while (!(cont>0))
      pthread_cond_wait( NotEmpty, bufMutex);
  getfrombuffer( Item );

  pthread_cond_signal( NotFull );
  pthread_mutex_unlock(bufMutex);
}
```

Um exercício clássico da programação concorrente

The Dining Philosophers



Philosopher (i):

Think;

take 2 forks (the left and right)

Eat;

release 2 forks (the left and right)

The Dining Philosophers-1

Correctness properties:

- *safety:*

  if a Philosopher is eating → has two forks

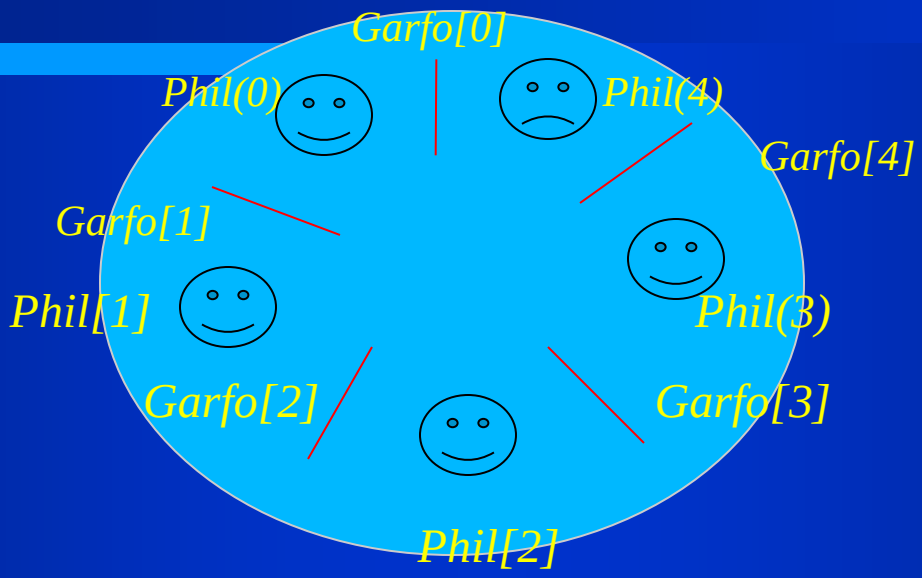
  if a Philosopher is thinking → has no forks

- *mutual exclusion to access the forks*
- *no deadlock*
- *no lockout → no starvation!*

The Dining Philosophers-1

```
var garfo: array[0..4] of semaphore; var i: integer;
Philosopher (i: integer);
{       repeat
            Think;
            P(garfo[i]); P(garfo[(i+1) mod 5]);
            Eat;
            V(garfo[i]); V(garfo[(i+1) mod 5]);
        forever }
{   for i = 0 to 4 do init_sem(garfo[i] = 1);
   cobegin Philosopher(0); Philosopher(1);
                Philosopher(2); Philosopher(3); Philosopher(4);
   coend
 }
```

The Dining Philosophers-1



The Dining Philosophers-1

Possible deadlock:

-- all Phil get their left forks → deadlock!

The Dining Philosophers-2

```
Fork_resource: monitor;
{ var required_forks[0..4] of integer;  i: integer;
   var   OK_to_EAT: array[0..4] of condition;
 →procedure entry Take_fork(i: integer);
   { if required_forks[i] <> 2 then WAIT(OK_to_EAT[i]);
        required_forks[(i+1) mod 5] --;                -- right neighbor
        required_forks[(i+4) mod 5] --;}                -- left neighbor
  → procedure entre Release_fork(i: integer);
   {   required_forks[(i+1) mod 5] ++;
       required_forks[(i+4) mod 5] ++;
        if required_forks[(i+1) mod 5]  == 2 then
                              SIGNAL(OK_to_EAT[(i+1) mod 5]);
        if required_forks[(i+4) mod 5]  == 2 then
                              SIGNAL(OK_to_EAT[(i-1) mod 5]);}
  /* monitor main body */
{   for i = 0 to 4 do required_forks[i]  = 2; }}
```

The Dining Philosophers-2

```
Philosopher (i: integer);
{       repeat
            Think;
            Fork_resource.Take_fork(i);
            Eat;
            Fork_resource.Release_fork(i);
        forever
}
```

The Dining Philosophers-2b

```
Program Dining Philosophers;
{ var required_forks[0..4] of integer;  var i: integer;      ← r
procedure Philosopher(i: integer);
 {  repeat
      Think,
    region   r  begin                          - Conditional Critical Regions
     AWAIT required_forks[i] ==  2;
       required_forks[(i+1) mod 5]  --;              -- right neighbor
       required_forks[(i+4) mod 5]  --;}             -- left neighbor
    end;
      Eat;
    region   r  begin
       required_forks[(i+1) mod 5]  ++;                  -- right neighbor
       required_forks[(i+4) mod 5]  ++;}                 -- left neighbor
    end;
  /* main body */
{   for i = 0 to 4 do required_forks[i]  = 2;  cobegin Phil(0)// Phil(1)//... coend
}
```

The Dining Philosophers-2

Lockout can occur!

	rfork[0]	rfork[1]	rfork[2]	rfork[3]	rfork[4]
Init	2	2	2	2	2
Take(1)	1	2	1	2	2 → Eat
Take(3)	1	2	0	2	1 → Eat
Take(2)	1	2	0	2	1 → Wait
Free(1)	2	2	1	2	1
Take(1)	1	2	0	2	1 → Eat
Free(3)	1	2	1	2	2
Take(3)	1	2	0	2	1 → Eat

Phil 2 never gets a chance!

The Dining Philosophers-3

```
var garfo: array[0..4] of semaphore; var i: integer;
var table: semaphore;
Philosopher (i: integer);
{       repeat
            Think;  P(table);              --- ONLY 4 can compete!
            P(garfo[i]); P(garfo[(i+1) mod 5]);
            Eat;
            V(garfo[i]); V(garfo[(i+1) mod 5]);
                    V(table);
        forever }
{   for i = 0 to 4 do init_sem(garfo[i] = 1); init_sem(table=4);
   cobegin      Philosopher(0); Philosopher(1); Philosopher(2);
                Philosopher(3); Philosopher(4);                coend}
```