Computação em Larga Escala

Concurrency 2

António Rui Borges

Summary

- *General principles of concurrency*
  - *Critical regions*
  - *Racing conditions*
  - *Deadlock and indefinite postponement*
- *Synchronization devices*
  - *Monitors*
- *Threads and monitors in Unix*
- *Suggested reading*

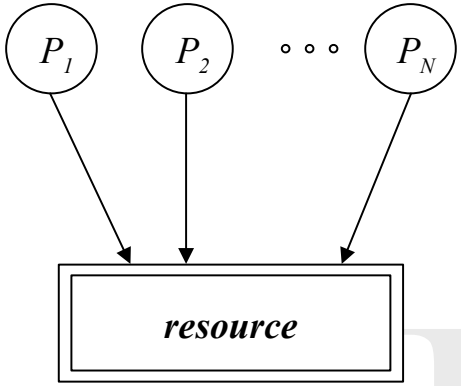
General principles of concurrency - 1

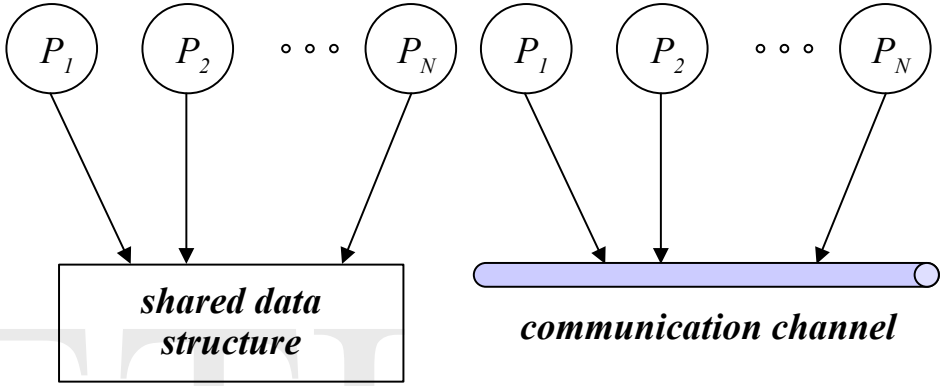
In a multiprogrammed environment, coexisting processes may present different behaviors interaction wise.

They may act as

- *independent processes* – when they are created, live and die without explicitly interacting among themselves; the underlying interaction is implicit and has its roots in the *competition* for the computational system resources; they are typically processes created by different users, or by the same user for different purposes, in an interactive environment, or processes which result from *job* processing in a *batch* environment
- *cooperating processes* – when they share information or communicate in an explicit manner; *sharing* presupposes a common addressing space, while *communication* can be carried out either by sharing the addressing space, or through a communication channel that connects the intervening processes.

General principles of concurrency - 2



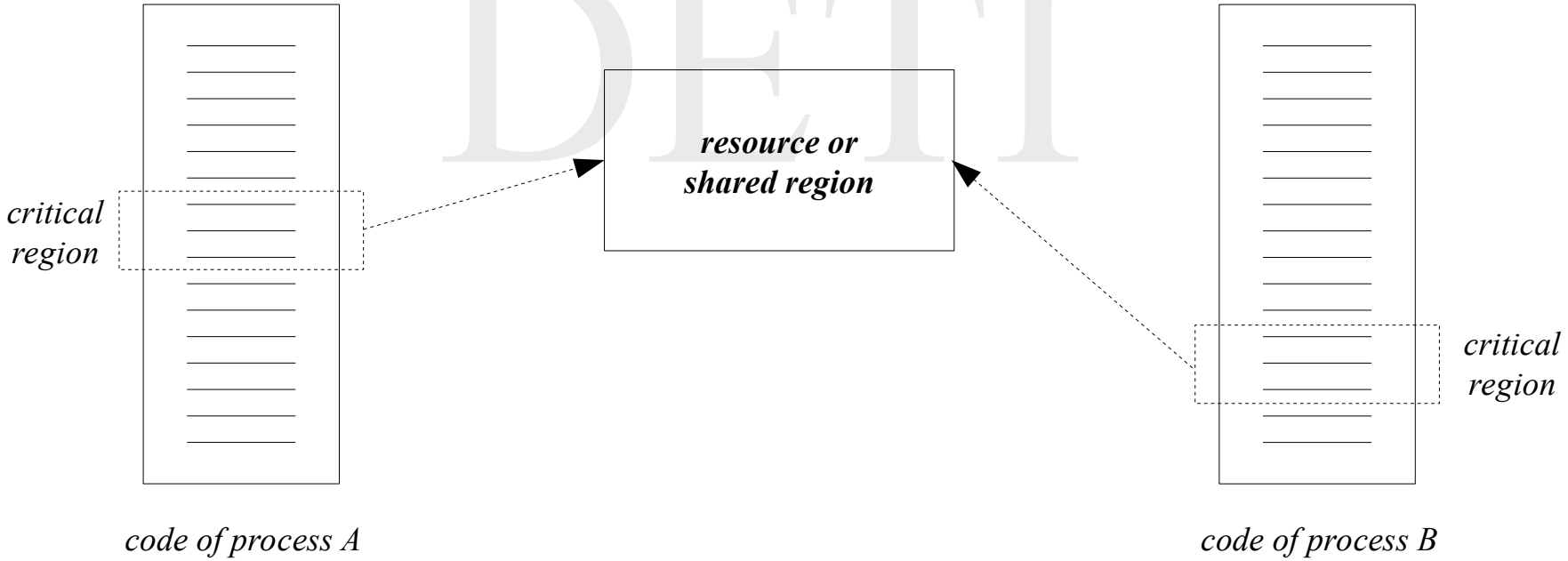


- *independent processes* that compete for access to a common resource of the computational system
- it is the *OS* responsibility to ensure that the resource assignment is carried out in a controlled fashion so that there is no loss of information
- this requires in general that only a single process may have access to the resource at a time (*mutual exclusion*)

- *cooperating processes* which share information or communicate among themselves
- it is the responsibility of the involved processes to ensure that access to the shared region is carried out in a controlled fashion so that there is no loss of information
- this requires in general that only a single process may have access at a time to the resource (*mutual exclusion*)
- the communication channel is typically a resource of the computational system; hence, access to it should be seen as *competition* for access to a common resource

General principles of concurrency - 3

Making the language precise, whenever one talks about *access by a process to a resource, or a shared region*, one is in reality talking about the processor execution of the corresponding access code. This code, because it must be executed in a way that prevents the occurrence of *racing conditions*, which inevitably lead to loss of information, is usually called *critical region*.



General principles of concurrency - 4

Imposing mutual exclusion on access to a resource, or to a shared region, can have, by its restrictive character, two undesirable consequences

- *deadlock / livelock* – it happens when two or more processes are waiting forever (blocked / in *busy waiting*) for the access to the respective critical regions, holding back for events which, one may prove, will never occur; the result is, therefore, the blocking of operations
- *indefinite postponement* – it happens when one or more processes compete for the access to a critical region and, due to a conjunction of circumstances where new processes come up continuously and compete with the former for this goal, access is successively denied; one is here, therefore, facing a real obstacle to their continuation.

One aims, when designing a multithreaded application, to prevent these pathological consequences to occur and to produce code which has a *liveness* property.

Problem of access to a critical region with mutual exclusion

Desirable properties that a general solution to the problem must assume

- *effective assurance of mutual exclusion imposition* – access to the critical region associated to a given resource, or shared region, can only be allowed to a single process at a time, among all that are competing to the access concurrently
- *independency on the relative speed of execution of the intervening processes, or of its number* – nothing should be presumed about these factors
- *a process outside the critical region can not prevent another to enter*
- *the possibility of access to the critical region of any process that wishes to can not be postponed indefinitely*
- *the time a process is inside a critical region is necessarily finite*.

Resources

Generally speaking, a *resource* is something a process needs to execute. Resources may either be *physical components of the computational system* (processors, regions of the main or mass memory, specific input / output devices, etc), or *common data structures* defined at the operating system level (process control table, communication channels, etc) or among processes of an application.

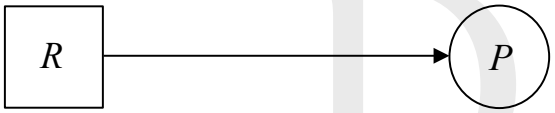
An essential property of resources is the kind of appropriation processes make of them. In this sense, resources are divided in

- *preemptable resources* – when they can be taken away from the processes that hold them, without any malfunction resulting from the fact; the processor and regions of the main memory where a process addressing space is stored, are examples of this class in multiprogrammed environments
- *non-preemptable resources* – when it is not possible; the printer or a shared data structure, requiring mutual exclusion for its manipulation, are examples of this class.

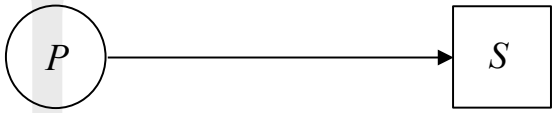
Schematic deadlock characterization

In a *deadlock* situation, only *non-preemptable* resources are relevant. The remaining can always be taken away, if necessary, from the processes that hold them and assigned to others to ensure that the latter may progress.

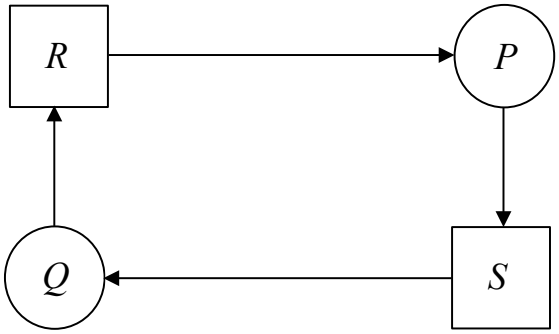
Thus, using this kind of framework, one may develop a schematic notation which represents a *deadlock* situation graphically.



process P holds the resource R



process P requires the resource S



typical deadlock situation
(*the most simplest one*)

Necessary conditions to the occurrence of deadlock

It can be shown that, whenever *deadlock* occurs, there are four conditions which are necessarily present. They are the

- *condition of mutual exclusion* – each existing resource is either free, or was assigned to a single process (its holding can not be shared)
- *condition of waiting with retention* – each process, upon requesting a new resource, holds all other previously requested and assigned resources
- *condition of non-liberation* – nobody, but the process itself, can decide when a previously assigned resource is freed
- *condition of circular waiting* (or *vicious circle*) – a circular chain of processes and resources, where each process requests a resource which is being held by the next process in the chain, is formed.

Deadlock prevention - 1

The necessary conditions to the occurrence of *deadlock* lead to the statement

deadlock occurrence ⇒ *mutual exclusion on access to a resource* **and**
waiting with retention **and**
no liberation of resources **and**
circular waiting

which is equivalent to

no mutual exclusion on access to a resource **or**
no waiting with retention **or**
liberation of resources **or**
no circular waiting ⇒ *no deadlock occurrence* .

Thus, in order to make deadlock *impossible to happen*, one has only to deny one of the necessary conditions to the occurrence of deadlock. Policies which follow this strategy are called *deadlock prevention policies*.

Deadlock prevention - 2

The first, *mutual exclusion on access to a resource*, is too restrictive because it can only be denied for non-preemptable resources. Otherwise, *racing conditions* are introduced which lead, or may lead, to information inconsistency.

Reading access by multiple processes to a file is a typical example of denying this condition. One should point out that, in this case, it is also common to allow at the same time a single writing access. When this happens, however, *racing conditions*, with the consequent loss of information, can not be completely discarded. *Why*?

Therefore, only the last three conditions are usually object of denial.

Denying the condition of waiting with retention

It means that a *process must request at once all the resources it needs for continuation*. If it can get hold of them, the completion of the associated activity is ensured. Otherwise, it must wait.

One should notice that *indefinite postponement* is not precluded. The procedure must also ensure that sooner or later the necessary resources will always be assigned to any process which will be requesting them. The introduction of *aging* policies to increase the priority of a process is a very popular method used in this situation.

Imposing the condition of liberation of resources

It means that a *process, when it can not get hold of all the resources it requires for continuation, must release all the resources in its possession and start later on the whole request procedure from the very beginning*. Alternatively, it also means that a *process can only hold a resource at a time* (this, however, is a particular solution and is not applicable in most cases).

Care should be taken for the process not to enter a *busy waiting* procedure of request / acquire resources. In principle, the process must block after freeing the resources it holds and be waken up only when the resources it was requesting are released.

Nevertheless, *indefinite postponement* is not precluded. The procedure must also ensure that sooner or later the necessary resources will always be assigned to any process which will be requesting them. The introduction of *aging* policies to increase the priority of a process is a very popular method used in this situation.

Denying the condition of circular waiting

It means *to establish a linear ordering of the resources* and *to make the process, when it tries to get hold of the resources it needs for continuation, to request them in increasing order of the number associated to each of them*.

In this way, the possibility of formation of a circular chain of processes holding resources and requesting others is prevented.

One should notice that *indefinite postponement* is not precluded. The procedure must also ensure that sooner or later the necessary resources will always be assigned to any process which will be requesting them. The introduction of *aging* policies to increase the priority of a process is a very popular method used in this situation.

Monitors - 1

A *monitor* is a synchronization device, proposed independently by Hoare and Brinch Hansen, which can be thought of as a special module defined within the [concurrent] programming language and consisting of an internal data structure, initialization code and a set of access primitives.

```
monitor example
  (* internal data structure
     only accessible from the outside through the access primitives *)
  var
    val: DATA;                                  (* shared region *)
    c: condition;          (* condition variable for synchronization *)
  (* access primitives *)
  procedure pa1 (...);
  end (* pa1 *)
  function pa2 (...): real;
  end (* pa2 *)
  (* initialization *)
  begin
    ...
  end
end monitor;
```

Monitors - 2

An application written in a concurrent language, implementing the *shared variables paradigm*, is seen as a set of *threads* that compete for access to shared data structures. When the data structures are implemented as *monitors*, the programming language ensures that the execution of a *monitor* primitive is carried out following a mutual exclusion discipline. Thus, the compiler, on processing a *monitor*, generates the required code to impose this condition in a manner totally transparent to the applications programmer.

A *thread* enters a *monitor* by calling one of its primitives, which constitutes the only way to access the internal data structure. As primitive execution entails mutual exclusion, when another *thread* is presently inside the monitor, the *thread* is blocked at the entrance, waiting for its turn.

Monitors - 3

Synchronization among *threads* using monitors is managed by *condition variables*. *Condition variables* are special devices, defined inside a monitor, where a thread may be blocked, while waiting for an event that allows its continuation to occur. There are two atomic operations which can be executed on a *condition variable*

wait – the calling *thread* is blocked at the *condition variable* passed as argument and is placed *outside the monitor* to allow another *thread*, wanting to enter, to proceed

signal – if there are blocked *threads* in the *condition variable* passed as argument, one of then is waken up; otherwise, nothing happens.

Monitors - 4

To prevent the coexistence of multiple *threads* inside a *monitor*, a rule is needed which states how the contention arisen by a *signal* execution is resolved

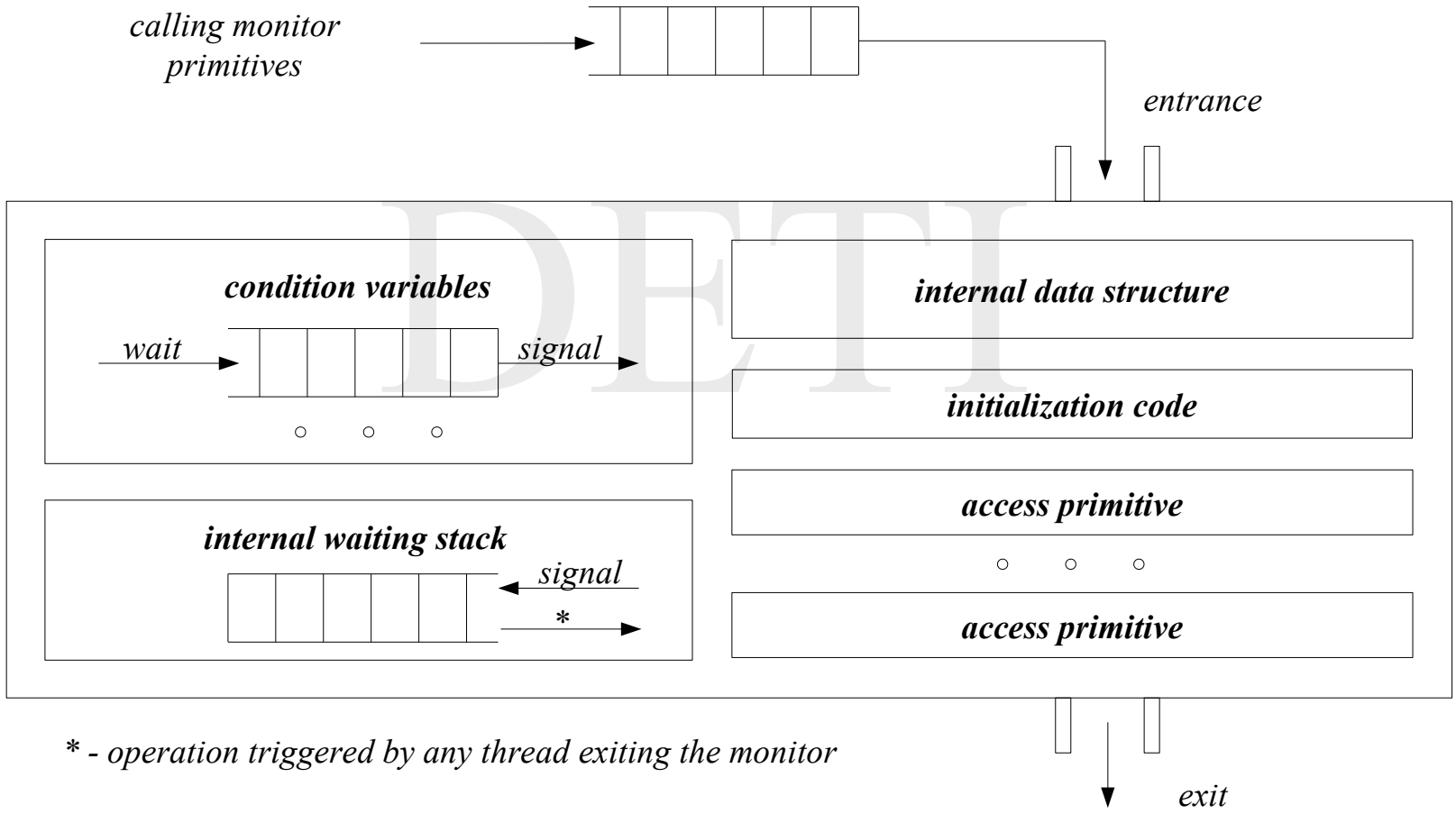
Hoare monitor – the *thread* which calls *signal* is placed *outside the monitor* so that the waken up *thread* may proceed; it is a very general solution, but its implementation requires a *stack*, where the *signal* calling *threads* are stored

Brinch Hansen monitor – the *thread* which calls *signal* must immediately exit the *monitor* (*signal* should be the very last executed instruction in any access primitive, except for a possible *return*); it is quite simple to implement, but it may become rather restrictive because it reduces the number of *signal* calls to one

Lampson / Redell monitor – the *thread* which calls *signal* proceeds, the waken up *thread* is kept *outside the monitor* and must compete for access to it again; it is still simple to implement, but it may give rise to *indefinite postponement* of some of the involved *threads*.

Monitors - 5

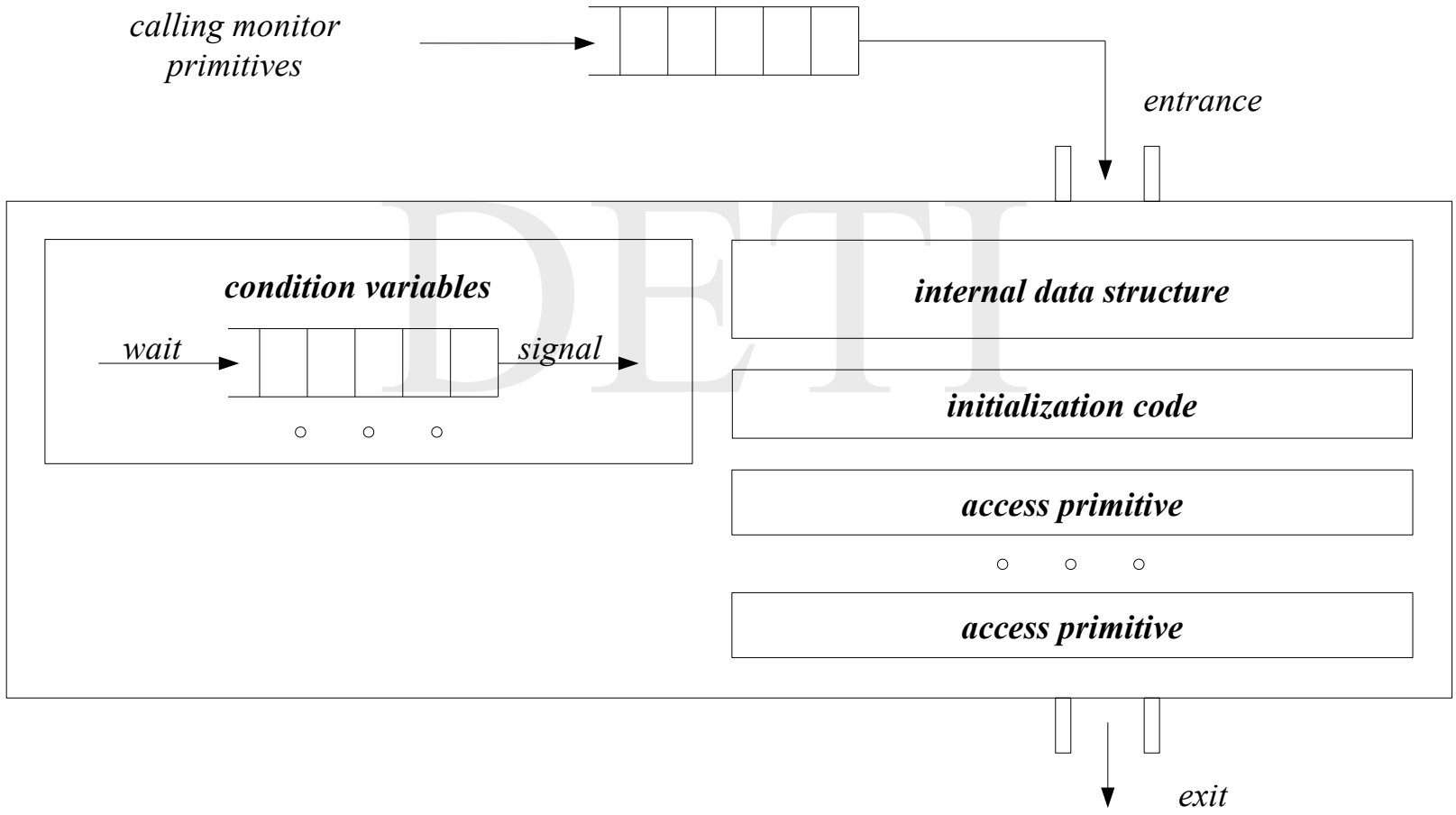
Hoare monitor



* - operation triggered by any thread exiting the monitor

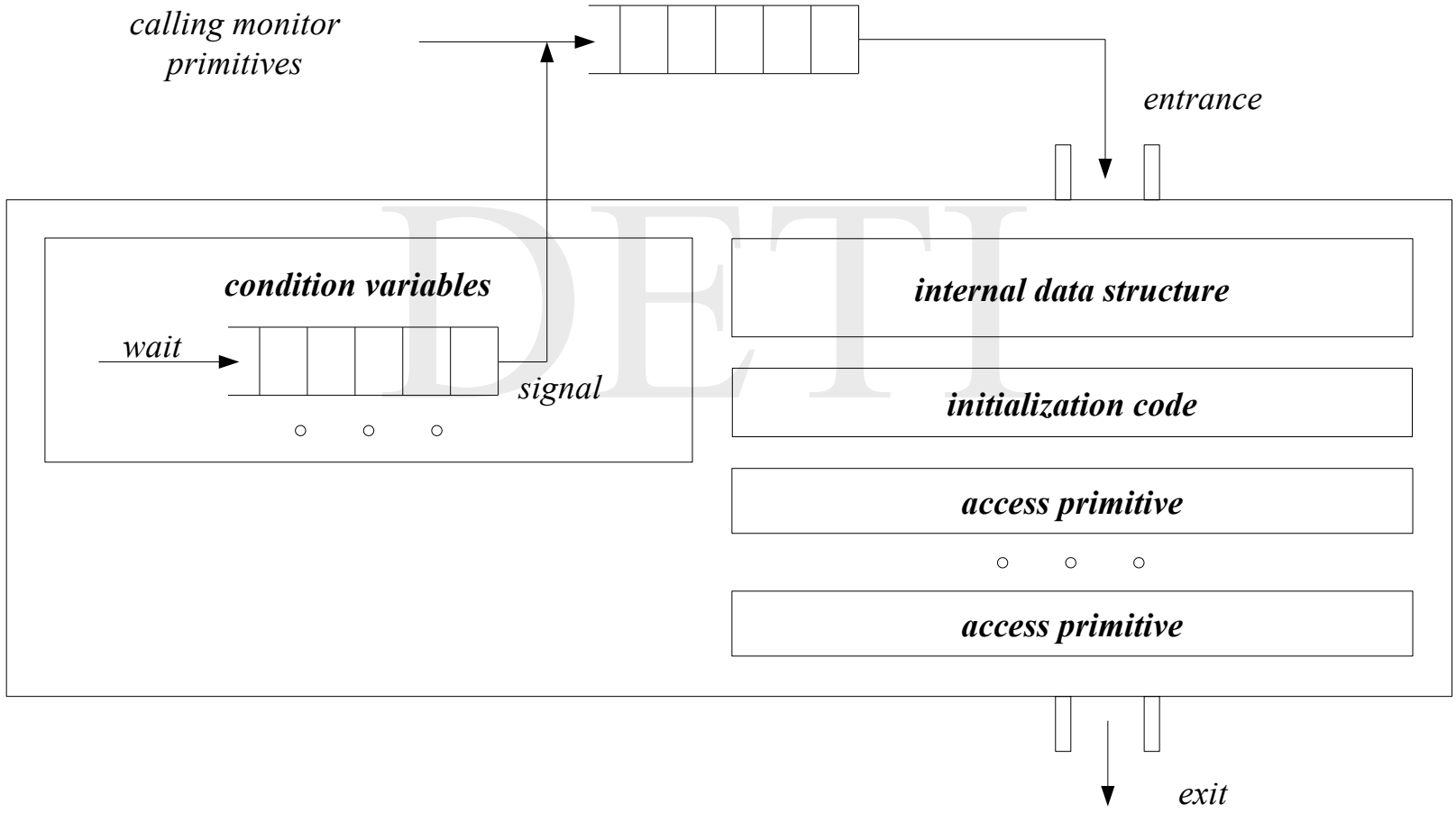
Monitors - 6

Brinch Hansen monitor



Monitors - 7

Lampson / Redell monitor



Threads and monitors in Unix - 1

The standard *POSIX*, *IEEE 1003.1c* defines an application programming interface (*API*) for creating and synchronizing *threads*. Unix-like operating systems usually have a library that implements it – this library is called *pthread*.

Its use is very important for C Language programmers since it enables the development of a full fledged monitor-based multithread environment in C. As C is not a concurrent language, the concept of *monitor* is not supported, but it can be implemented through the use of *mutexes* and *condition variables* supplied by the library. Bear in mind that the *pthread* monitors are Lampson / Redell.

For programs using functions of this library, one is required to name it explicitely in the link processing command

```
gcc -lpthread ...
```

Threads and monitors in Unix - 2

The most important library functions are

- `pthread_create` – *thread* start, a new execution thread will be created and will be assigned to the calling of the function passed in argument
- `pthread_exit` – equivalent to `exit` in the case of a process
- `pthread_once`,`pthread_mutex_*`,`pthread_cond_*` – required for monitor implementation
- `pthread_join` – almost equivalent to `wait` in the case of a process, although one must specify the *thread* id whose termination one is waiting for
- `pthread_self` – equivalent to `getpid` in the case of a process.

Suggested reading

- *POSIX Threads Programming: Tutorial*,
  https://docs.oracle.com/cd/E26502_01/html/E35303/tlib-1.html
  http://www.cs.kent.edu/~ruttan/sysprog/lectures/multi-thread/multi-thread.html