*

Load Balancing and Termination Detection

*Material based on B. Wilkinson et al., "PARALLEL PROGRAMMING. Techniques and Applications Using Networked Workstations and Parallel Computers"

©2002-2004 R. Leduc

Intro to Load Balancing

Originally, we divide the problem into a fixed number of processes.

Each process has an equal portion of the tasks with no consideration for different CPU types or speeds.

Could result in some processors completing tasks early and becoming idle.

Cause: Work load unevenly divided (ie, some portions take longer to process) or faster processor or combination.

Goal: want all processors working continuously on tasks such that we get minimal runtime.

Achieving this goal by distributing tasks uniformally amongst the processors is referred to as *load balancing.*

©2002-2004 R. Leduc

Load Balancing

In Mandelbrot example, we dealt with special case without interprocess communication (except master with slave processes).

Will now extend it to include communication between processes.

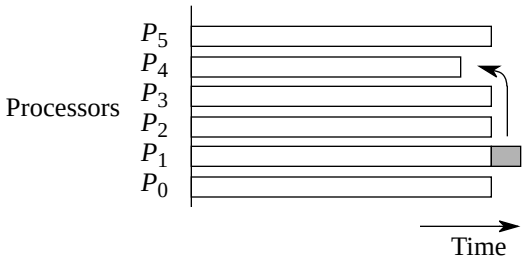
Figure 7.1 shows an example of perfect load balancing.

Assume sequential run time took k clock cycles to process a problem, thus $t_s = k$.

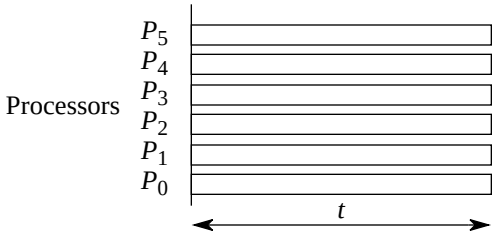
We achieve perfect load balancing when our parallel execution time is:

$$t_p = k/p$$

©2002-2004 R. Leduc



(a) Imperfect load balancing leading to increased execution time



(b) Perfect load balancing

Figure 7.1 Load balancing.

Static Load Balancing

Static approaches are when task division is performed before program is executed.

Often referred to as the *mapping problem* or *scheduling problem.*

Theory developed using optimization methods, knowledge of interdependencies of parts of the problem, and estimates of their runtimes.

Proposed techniques:

- *Round Robin Algorithm:* gives out tasks in sequential order, then returning to first process and repeats.

- *Randomized Algorithm:* chooses process at random to perform tasks.

©2002-2004 R. Leduc

Static Load Balancing Cont.

- *Recursive Bisection:* divides problem up recursively into tasks of equal effort, while minimizing communication.

- *Simulated Annealing, and Genetic Algorithm:* two optimization techniques.

©2002-2004 R. Leduc

Mapping Problem

If processors are connected by static links, want to map communicating processes to processors with direct links to reduce communication delay.

Key aspect of "mapping" processes to the physical system.

Likely require different mapping for different networks.

In general, problem is NP-complete.

Normally, heuristics are used to choose which processes are to run on which processors.

©2002-2004 R. Leduc

Problems with Static Approach

1) Hard to create acurate estimates of execution time of parts of a program without actually executing the code.

Practically guarantees that scheduling and mapping will be suboptimal.

2) Systems may have communication delays that change depending on circumstances. Hard to incorporate in a static approach

3) Some problems have an indeterminate number of steps to arrive at solution. ie search algorithms.

4) Mapping of processes to processors is system dependent. Not very portable.

5) Even with perfect knowledge, choosing the correct mapping is usually complicated.

©2002-2004 R. Leduc

Dynamic Load Balancing

To address the problems with static approach, and to make one's program more adaptable, people often use *dynamic load balancing (DLB).*

In DLB, division of load is decided based upon the execution of individual parts, as they are being executed.

Increases overhead while executing, but much more adaptive.

For DLB, individual tasks are allocated to processors while program is running.

Two main DLB classifications:

- *Centralized DLB.*

- *Decentralized DLB.*

©2002-2004 R. Leduc

Centralized DLB

In centralized approach, tasks are distributed from centralized location.

Have a master process that has the group of tasks to be performed and sends them to slave processes.

When a slave has completed its task, it requests another.

Basic *work pool* or *processor farm* approach.

Can be applied to problems with uniform tasks sizes or problems with tasks of different sizes and difficulty levels.

In latter case, best to hand out large/difficult tasks first.

Allows the smaller tasks to even out the work load once the large ones have been completed.

©2002-2004 R. Leduc

Varying Size Work Pools

For some problems, the size of work pool is not known in advance.

For eg, a search algorithm. Processing a task may generate new tasks to process, adding to the work pool.

Eventually, there must be no new tasks generated, so that the pool of tasks can be emptied, completing the problem.

©2002-2004 R. Leduc

Ordering Tasks

List of tasks to be processed is often stored in a queue, as in Figure 7.2.

If tasks are of equal size/importance, a FIFO would probably work.

If some tasks are considered more important, they can be put at front of queue.

The master process might keep additional information, such as the current best solution.

©2002-2004 R. Leduc

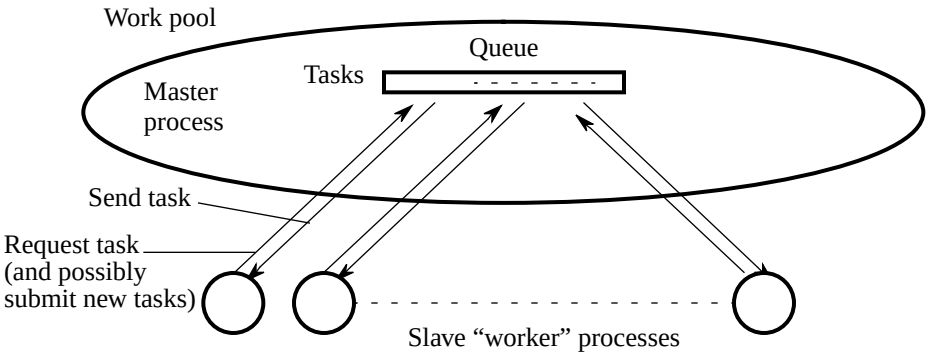


Figure 7.2 Centralized work pool.

Centralized Termination

One of the difficult aspects of dynamic load balancing is detecting when the computation is complete. Called *termination detection.*

Advantage of centralized approach is often easy for master to detect termination.

When tasks taken from a queue, computation is complete when:

1) Task Queue is empty.

2) All Process have requested a new task and no new tasks have been generated.

Can't just quite when task queue is empty, because running process might generate new tasks for the queue.

Once termination detected by master, each slave sent termination signal.

©2002-2004 R. Leduc

Slaves Detecting Termination

In some cases, termination might be detected by slave processes which reach a local condition.

ie. search algorithm, and slave finds desired item.

Slave would announce this to master, who would then send signal to other slaves.

For some applications, each slave might need to reach local condition.

ie. convergence of local solution such as in Heat Distribution Problem.

In this situation, master must receive termination signal from all slaves.

©2002-2004 R. Leduc

Problem with Centralized DLB

Limitation of centralized approach is that you have single process dispersing tasks one at a time.

Potential for bottleneck and loss of parellization when multiple slaves request tasks simultaneously.

Works well if few slaves, and tasks are computationally intensive.

If many slaves, and tasks have finer granularity, then may get better performance by distributing the work pool across multiple processes.

©2002-2004 R. Leduc

Decentralized DLB

In *decentralized load balancing*, groups of processes work on the problem, exchanging information with each other, ultimately reporting to a single process.

A worker might receive tasks from other workers, and may send tasks to other workers which may perform the task, or pass it on as they see fit.

Typically we have multiple work queues distributed amongst the processes, with some mechanism to disperse tasks throughout the system, as well as detect termination.

©2002-2004 R. Leduc

Mini-master Approach

In this approach, the master process divides it's initial work pool into sections, and sends each to a set of processes that function as "mini-masters (M_0 to M_{n-1}."

Each mini-master would have its own group of slave processes to dole tasks out to. See Figure 7.3.

If doing an optimization problem, could have each mini-master find local optimum value, then pass results to master to determine global value.

©2002-2004 R. Leduc

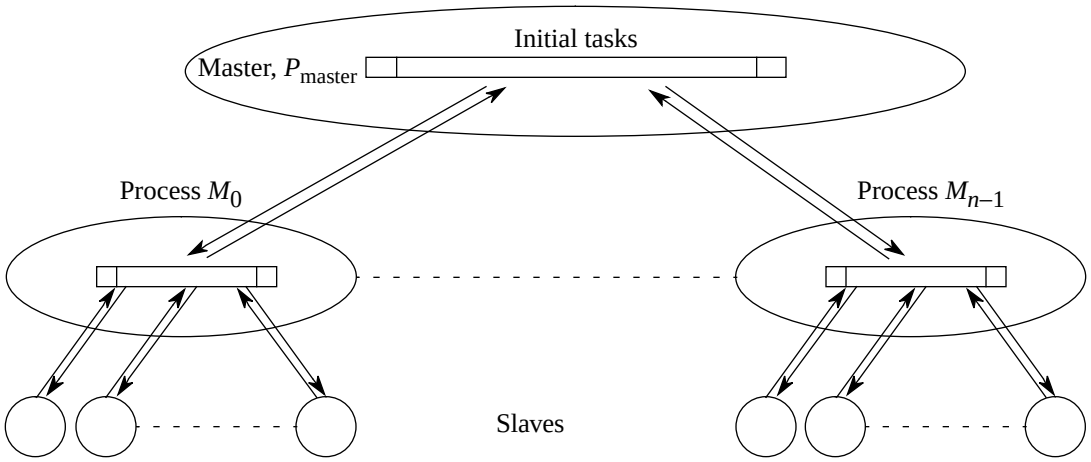


Figure 7.3 A distributed work pool.

Fully Distributed Work Pool

Particularly when processes not only execute tasks, but generate new ones, it may be worthwhile to have each process have its own task queue.

In this approach, a process needing a task could request one from any other process. See Figure 7.4.

Two main transfer methods:

1) The *receiver-initiated*

2) The *sender-initiated*

©2002-2004 R. Leduc

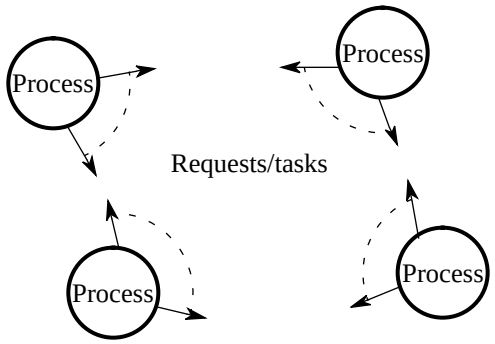


Figure 7.4 Decentralized work pool.

Task Transfer Methods

In the receiver-initiated approach, a process chooses which processes to request tasks from.

Typically, when a process is out of tasks (or close to it), it will request more from other processes.

Works well under high load.

In the sender-initiated method, the sending process selects which processes to send tasks to.

In general, a process that is heavily loaded will pass on some of its tasks to other processes than are willing to accept them.

Works well under light load.

Could do a mixture of both, but doing load calculations is costly.

©2002-2004 R. Leduc

Load balancing

Consider load balancing for receiver-initiated method.

Strategies usually depend on network interconnect.

If using ring interconnect, processes then request tasks from its neighbours.

For hypercube, can request from any process with a direct interconnect.

Must make sure you don't just pass on the same task you just received.

©2002-2004 R. Leduc

Process Selection

Without specific topology, all process are equivalent, thus could select any other process to send/request tasks to/from.

Each process would have local selection algorithm (See Figure 7.5), which could be applied to all possible processes or specific subsets.

For *round robin algorithm,* process requests a task from each other process in turn, then repeats.

In *random polling algorithm,* process p_i requests a task from process p_x, where $x \in \{0, 1, \ldots, n-1\}$ is selected randomly (n processes, and $x \neq i$).

©2002-2004 R. Leduc

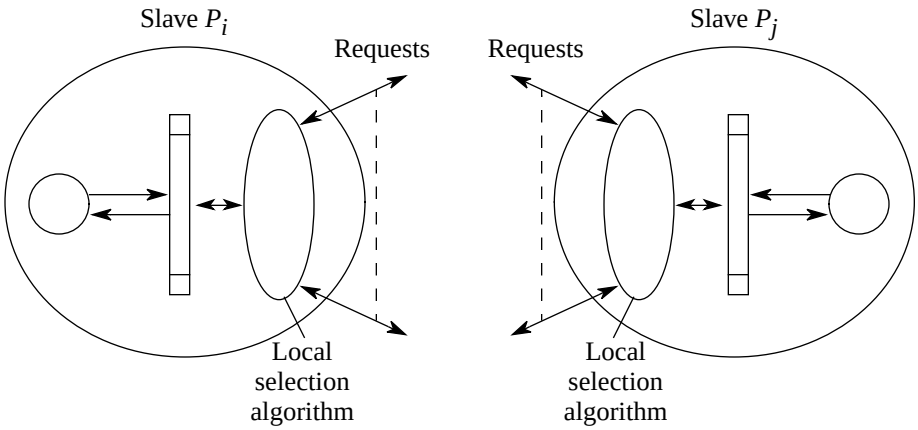


Figure 7.5 Decentralized selection algorithm requesting tasks between slaves.

Load balancing using Line Structure

Technique designed to be used when processors are constructed in line structure.

Creates a task queue, with individual processes accessing specific locations. See Figure 7.6.

Tasks are fed in from the left, and are shifted down the queue.

When a process detects a task at its queue location and it's idle, it takes the task to process.

Tasks then shuffle down from the left until the space is filled, and a new task is inserted on the far left.

Can put larger/higher priority tasks into queue first.

©2002-2004 R. Leduc

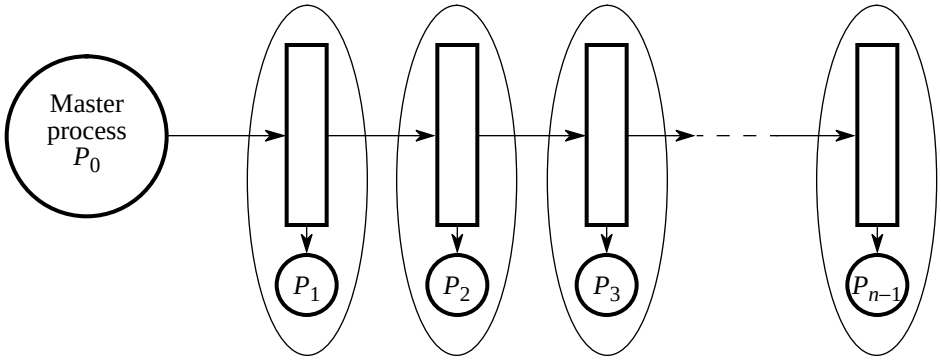


Figure 7.6 Load balancing using a pipeline structure.

Shifting Tasks

Shifting performed by exchanging messages between adjacent processes.

Use two subprocesses (see Figure 7.7):

- Left and right communication (P_{comm}).
- Performs current tasks (P_{task}).

To implement these two processes on single processor, use threads.

If not available, see time sharing example at end of Section 7.2.3.

Could create similar structure using binary tress. See Figure 7.8.

©2002-2004 R. Leduc

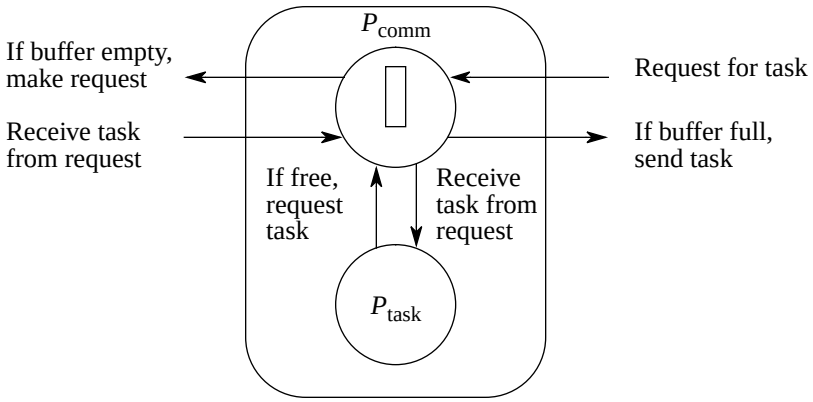


Figure 7.7 Using a communication process in line load balancing.

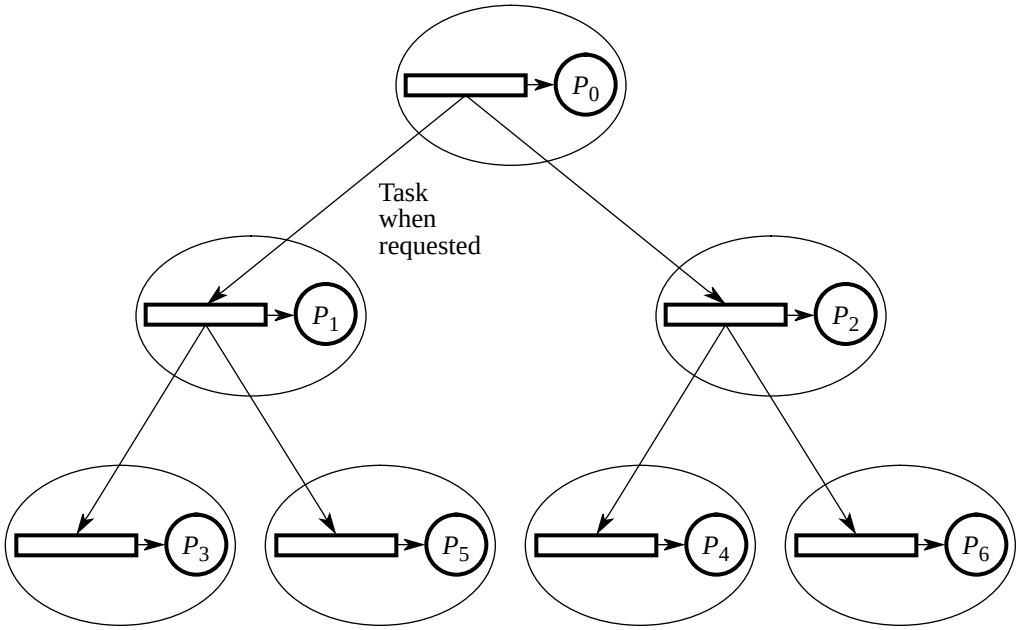


Figure 7.8 Load balancing using a tree.

Decentralized Termination Conditions

For distributed approaches, it is usually hard to know when to terminate unless the problem type is such that one process reaches a solution.

Distributed termination at time t needs to satisfy these two conditions:

1) At time t, local (application specific) termination conditions have been satisfied at each process.

2) There are no in transit messages between process at time t.

©2002-2004 R. Leduc

Decentralized Termination Conditions Cont.

Condition 1 is easy to determine. Each process could send message to Master when local condition met.

Second difficult as time to send message between processes not known in advance.

Could wait for a "long" period to allow all messages to be received, but code wouldn't be portable.

©2002-2004 R. Leduc

Acknowledgement Messages

Method to detect distributed termination using request and acknowledgement messages. See Figure 7.9.

Every process can be in two possible states: *inactive* or *active.*

A given process (say P_i) is initially in the inactive state. When it is sent a task from another process (say P_j), it goes to the active state, and P_j becomes its parent.

If process P_i passess a task to an inactive process (say P_k), it becomes P_k's parent.

Creates a tree of processes, each with a unique parent.

An active process could receive tasks from processes other than its parent, but these processes do NOT become its parent.

©2002-2004 R. Leduc

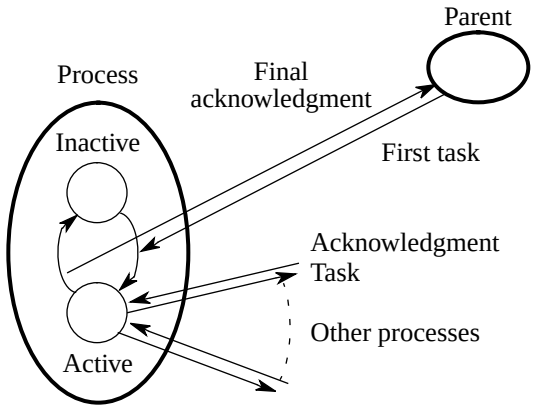


Figure 7.9 Termination using message acknowledgments.

Sending Acknowledgements

Whenever a process sends a task to a processor, it expects an acknowledgment back.

When a process (say P_i) receives a task from a process that is not its parent, P_i immediately sends an acknowledgment.

When P_i receives a task from its parent, P_i only sends an acknowledgement when it becomes inactive.

A process becomes inactive when:

- Process has reached its local termination condition (all tasks finished).
- It has sent all its acknowledgements for tasks it received.
- It has received all acknowledgements for tasks that it sent out.

©2002-2004 R. Leduc

Acknowledgement Messages: Final Condition

Conditions for inactive, means that a processes' children must become inactive before it can.

Computation terminates when root process becomes inactive.

Best method to use as very general and has been proven to be sound.

©2002-2004 R. Leduc

Ring Termination Algorithms

Processes are organized in ring structure for termination. See Figure 7.10.

First consider a single-pass ring algorithm. Here, a process can not be reactivated after reaching local termination condition.

Algorithm works as follows:

- When process P_0 terminates, it creates a token that it passes to process P_1.

- When process P_i $(1 \leq i < n)$ gets the token, it passes it on to P_{i+1} if it has already reached its termination condition. Otherwise, it waits until it satisfies its local conditions, then passes the token on.

©2002-2004 R. Leduc

Ring Termination Algorithms Cont.

- Process P_{n-1} passes the token to P_0.

- When P_0 gets the token, it knows everyone has terminated and can send a message (if necessary) to each process to notify them of global termination.

©2002-2004 R. Leduc

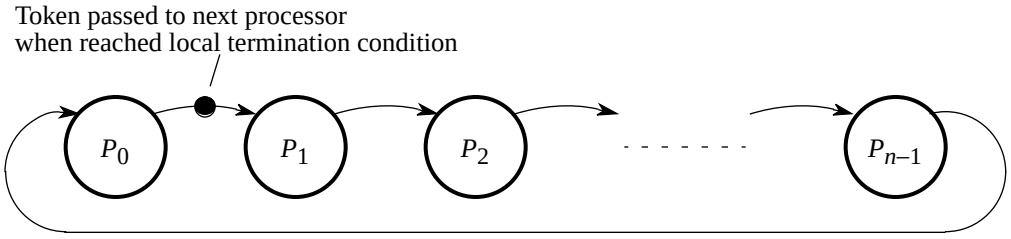


Figure 7.10 Ring termination detection algorithm.

Multi-pass Ring Algorithm

this algorithm can handle processes being re-activated, but may require two or more passes around the ring.

A reactivation occurs when a process P_i passes a task to process P_j, with $j < i$ and the token has already passed P_j. See Figure 7.12.

This means the token must make another pass to handle this.

To differentiate, we have a black or a white token, and can have black or white processes.

Algorithm works as follows:

- When process P_0 terminates, it becomes white and passes a white token to P_1.

©2002-2004 R. Leduc

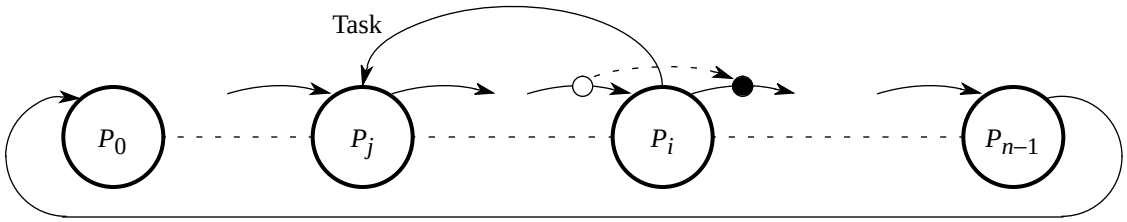


Figure 7.12 Passing task to previous processes.

Multi-pass Algorithm Cont.

- The token is passed along the ring as each process, in turn, terminates.

- If P_i passes a task to process P_j, with $j < i$, process P_i becomes black otherwise it is a white process.

- A black process colors a token black, passes it on, and then becomes a white process. A white process passes on the token without changing the token's color.

- If P_0 receives a black token it passes on a white one. When it receives a white token, then all processes have terminated.

©2002-2004 R. Leduc

Tree Algorithm

Can use a similar concept for a tree interconnection network.

When a leaf node has met its local termination conditions, it passes a token to its parent node.

When a non leaf node has reached its termination conditions, and has received a token from all of its children nodes, it sends a token to its parent. See Figure 7.13.

When the root node has received a token from all its children nodes and has itself terminated, global termination has occurred.

The root node then informs the other processes.

©2002-2004 R. Leduc

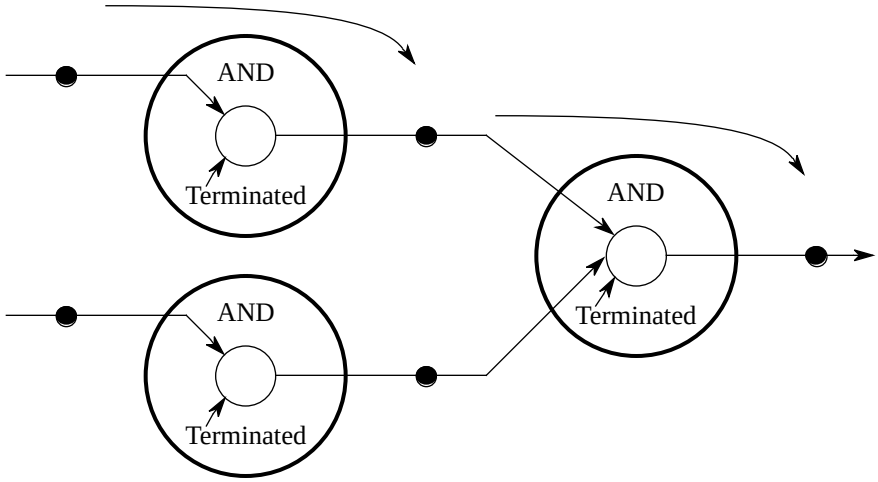


Figure 7.13 Tree termination.