Distributed Computing and Systems
Chalmers university of technology

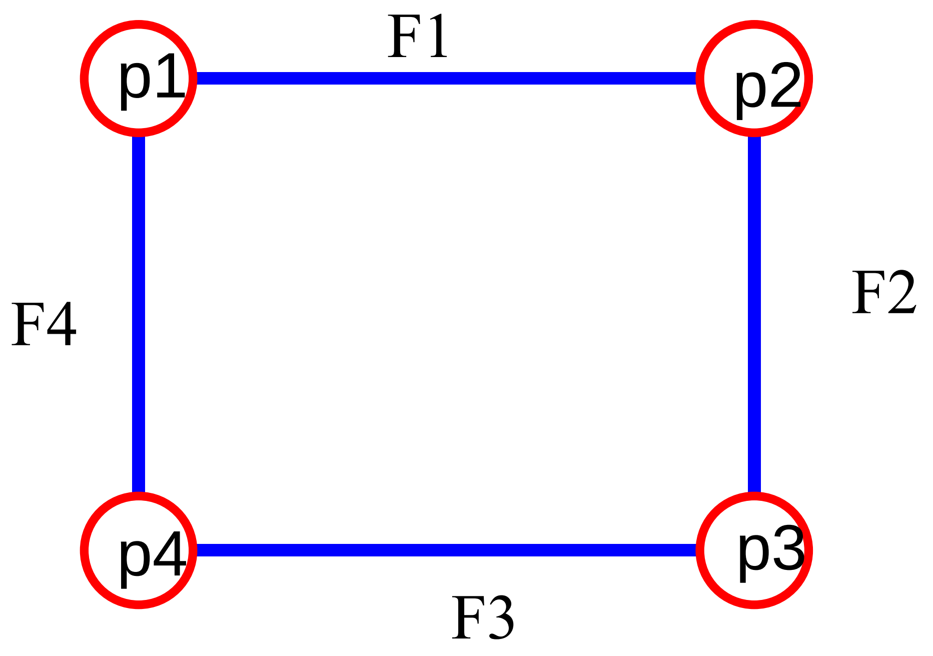
Prof Philippas Tsigas
Distributed Computing and Systems Research Group

DISTRIBUTED SYSTEMS II

RESOURCE ALLOCATION II

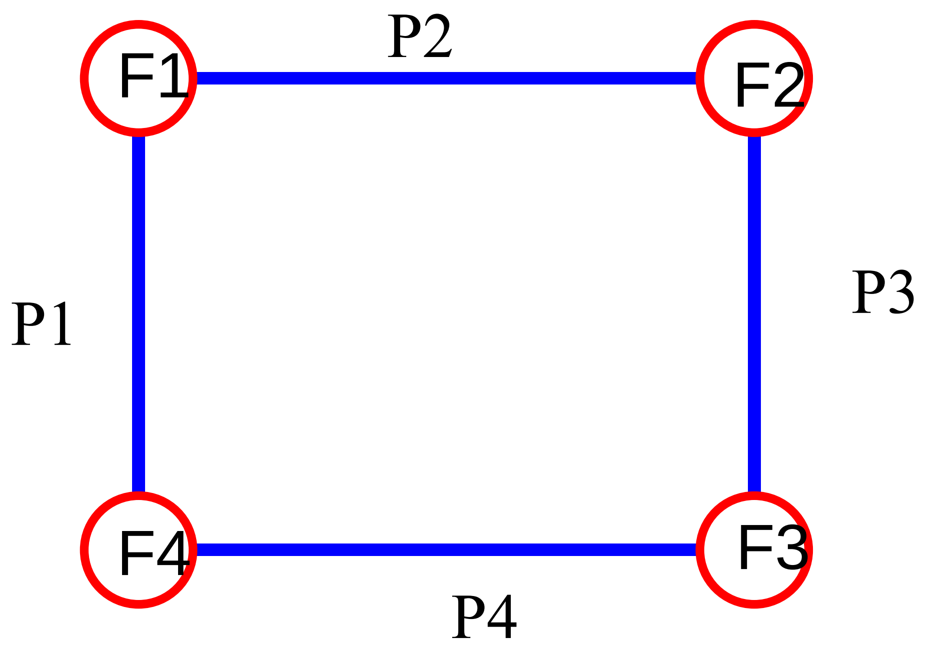
Process Graph: Coloring Philoshophers

Edge: Conflict on a resource by the nodes

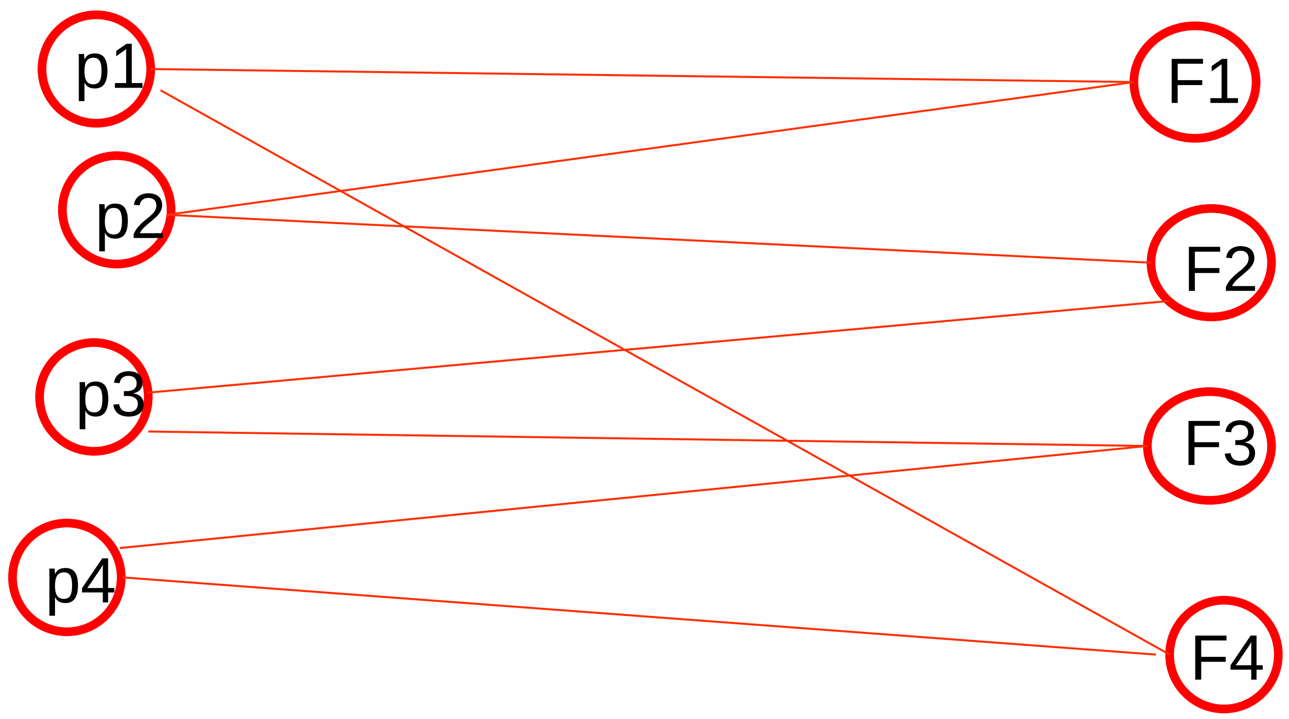


Resource Graph: Coloring Forks

Edge: Resources are needed at the same time by one (or more) processes



Process Graph: Coloring Philoshophers



F3

GENERALIZED DINING PHILOSOPHERS

Resource Allocation

Generalising the Dinning Philosophers

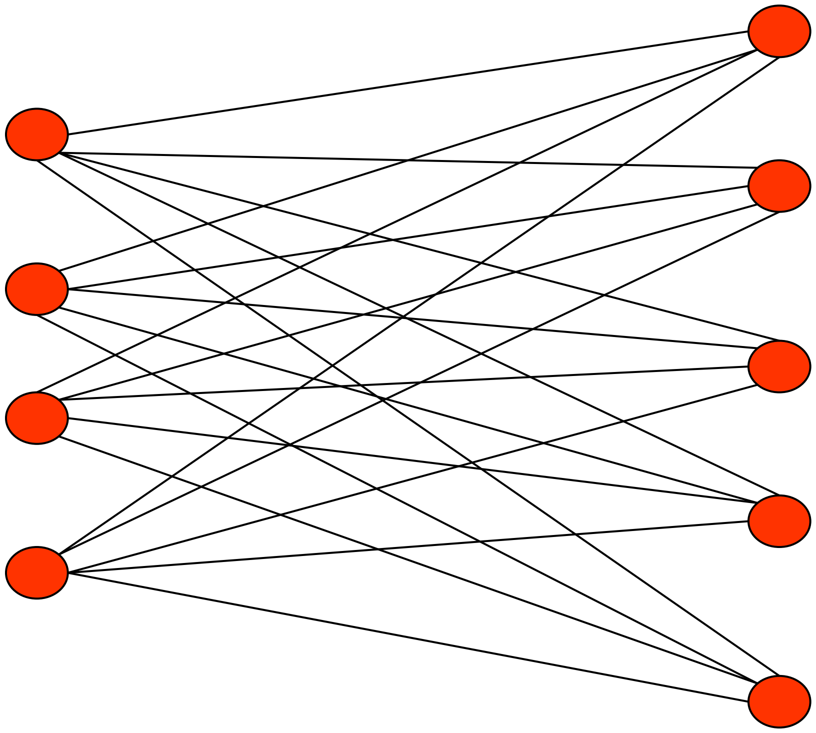
A resource allocation problem consists of a finite set of *resources* and some competing *processes*.

Processes request access to the resources from time to time in order to execute a code segment called *critical section*. Upon being granted all the requested resources, a process proceeds to use them and eventually relinquishes them.

Requirements;

1. **Mutual exclusion** No resource should be accessed by more than one process at the same time.
2. **No Starvation** As long as processes do not fail, no process should wait forever for requested resources.

General Case



Resource Allocation Solution

We generalise in a straightforward way the RightLeft Dinning philosophers algorithm to an arbitrary resource allocation problem.

We first construct the **Resource Graph**:

- The nodes of this graph represent the resources
- There is a node from one node to another if there is some process that uses both the resources.

Coloring again

We node-color the *resource graph*

Each graph is Δ+1-colorable

– Sequential algorithm: trivial
– Greedy algorithm

The Generalisation

Each process seeks its resources in increasing order according to the total ordering constructed by the coloring.

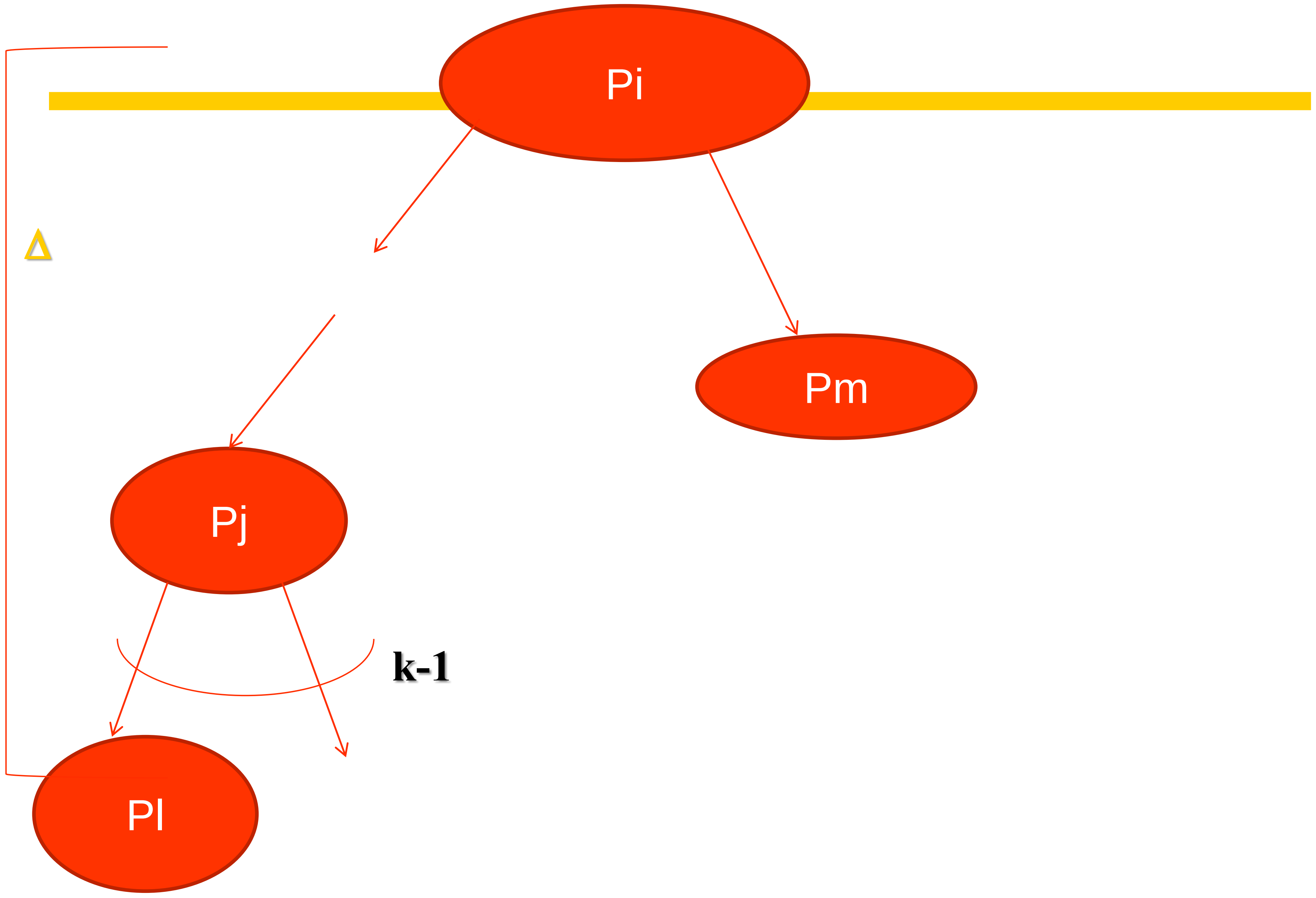
A process seeks a resource by putting its index at the end of that resource's queue.

The process obtains the resource when its index reaches the front of that resource's queue.

When a process exits C, it returns all of its resources by removing its index from their queues.

Let us assume that k is the maximum number of processes that require any single resource.

What is the time complexity of the algorithm?


Pi
Δ
Pm
Pj
k-1
Pl

Conflict -> Precedence Graph

Undirected GRAPH, in which edges represent shared resources between processes we call this graph CONFLICT GRAPH.

The algorithm by Chandy and Misra resolves conflicts by defining for every possible conflict a *precedence* relation:

- When two processes compete for a resource the one with higher precedence may access the resource first.
- In order to receive a solution which is fair these precedences will have to change dynamically.

The directed graph graph that changes dynamically is called *precedence graph*.

For each resource an edge of the precedence graph is directed from processes with lower precedence to processes with higher precedence.

The precedences of the graph are chosen such that it is always possible to distinguish at least one process from all other processes i.e. this process can enter its critical section. (NO DEADLOCK)

This is ensured by the existence of at least one process which has higher precedence for all its shared resources. A process with this property is called *sink*.

Its existence is guaranteed when the precedence graph is always acyclic.

By changing directions of edges it is possible to change the precedences dynamically.

This must happen in a way that the precedence graph stays acyclic, so *progress*, *fairness* and *mutual exclusion* is guaranteed.

Starting with a DAG

- The graph is initialised acyclic for example by a node-colouring algorithm.
- The graph can remain acyclic if after use of the critical section a process reverse all adjacent precedences in one step.
- Need a mechanism to keep the sense of direction:

The mechanism

Forks which have the property to be either *clean* or *dirty*.

- A fork will be cleanedbefore it is send to a neighbour process.
- A clean fork will become dirty when the holder of the resource enters the critical section.
- After use it remains DIRTY until it is sent to a neighbour process.

The dynamic DAG

- The respective precedence graph H can be defined in the following way:
- For all pairs of processes *p* and *q* which share a common resource, <*p*,*q*> one of the following statements is true:

1. *p* holds the *fork* for the resource and the *fork* is CLEAN
2. *q* holds the *fork* for the resource and the *fork* is DIRTY
3. the *fork* for the resource is in transit from *q* to *p*

Requesting Forks

The request of forks is realized by *request tokens*.

For each fork there exist one request token such that only the holder of the request token can request a fork.

A *hungry* process requests a fork by sending the *request* TOKEN to the owner of the desired *fork*.

A process is not interested in accessing its resources when it holds a *request* TOKEN but not a *fork*.

The algorithm

The algorithm is initialised by an acyclic precedence graph H and all processes with lower precedence own dirty forks while processes with higher precedence own request tokens.

All processes are *thinking* i.e they are not interested in their resources.

A process which becomes *hungry* will send all its *request* TOKEN to neighbour processes and wait until it received all *forks*.

- A process which received all forks will change its state to *eating*.
- A process which leaves the CRITICAL SECTION changes the state of all its *forks* to DIRTY. Then for all held *request* TOKEN the respective *fork* is sent to neighbour processes.

The above steps assume following rules:

Receiving a *request* TOKEN for fork f:

1. If processors state is different from *eating* and f is DIRTY then f will be sent to the requesting processor.

2. If processors state was also *hungry* then the *request* TOKEN will also be sent back.

Receiving a fork f: The state of f will be set to clean.

Correctness

Mutual Exclusion:

Proof. The precedence graph H is acyclic. □

No Starvation

Proof. Let the depth in H of any process p be defined as the maximum number of edges along a path from p to another process without predecessor. The proof will show by induction that a process of depth k will eventually eat if predecessors at depth *k-1* can EAT. □

Complexities

Communication Complexity: *O(degree)*

Proof. A process sends at most one *request* TOKEN to each neighbour and receives from each neighbour at most one *fork*. □

Tine Complexity: *O(n)*