Chapter 8: Resource Allocation

* Critical section: a shared resource in a distributed system which needs synchronization to ensure it is properly modified
* **Proving correctness (**Ensuring mutual exclusion in a distributed system**)**:
  + **Safety**: two processes should not have permission to enter the critical section concurrently / simultaneously.
  + **Liveness**: every request for the critical section is eventually granted.
  + **Fairness**: Requests must be granted **in the order they are made**.
* Centralized Algorithm: singular process is designated as the leader, which dictates access to the critical section.
  + Algorithm: **//\*\* Satisfies ONLY safety and liveness.**
    - Var: Boolean haveToken = initially true for current process, false for all other processes
    - Var: pendingQueue = initially empty
    - Receive(request) from Pj:
      * if haveToken (current process) : send “okay” to Pj
      * else: pendingQueue.add(Pj)
    - Receive(release) from Pj:
      * If (!pendingQueue.isEmpty()):
        + Send “okay” to process pendingQueue.dequeue();
      * Else:
        + haveToken = true; // no more processes requesting C.S., so the current process takes ownership of the token
  + Modifying algorithm to have **fairness** property using **vector clocks**:
    - New algorithm:
      * Var: array of N integers, **v** initially all 0 (don’t want v[i] to have highest vector clock value)
      * Var: pendingQueue = initially empty
      * Receive(request) from Pj:
        + Pj undergoes message send of vector clock mechanism.
        + If (findMaxIndexOf(v) == process **j**):

Send “okay” to Pj

* + - * + Else:

pendingQueue.add(Pj)

* + - * Receive(release) from Pj:
        + **Pi** undergoes message receive of vector clock mechanism.
        + If (!pendingQueue.isEmpty()):

Send “okay” to process pendingQueue.dequeue();

* **Lamport’s Algorithm**:
  + Algorithm (Pi):
    - Var: pendingQueue = initially empty of (timestamp, processID)
    - Var: numberOfAcknowledgements = initially 0
    - RequestCS():
      * send “request” message with (logicalClock (timestamp), PID) to all other processes
      * pendingQueue.add(Pi);
      * numAck = 0;
    - On receive(logicalClock(timestamp), j) from Pj:
      * Insert(logicalClock(timestamp), j) into pendingQueue
      * Return (acknowledgement, logicalClock(timestamp))
    - On receive((acknowledgement, logicalClock(timestamp)):
      * numAck++;
      * if (N – 1 == numAcks(of Pi)) && minTimestamp(pendingQueue) == Pi:
        + enterCS
    - ReleaseCS():
      * Remove Pi from pendingQueue
      * Send “okay” message to all other processes
    - On receive(release) from Pj:
      * Remove Pj from pendingQueue
      * If (numAcks(of Pj) == N – 1) && minTimeStamp(pendingQueue) == Pj):
        + enterCS
  + Complexity: **3 \* (N – 1)**
    - N – 1 requests
    - N – 1 acknowledgements
    - N – 1 releases
* **Ricart and Agrawala’s Algorithm**:
  + Complexity: **2 \* (N – 1)**
    - N – 1 requests
    - N – 1 okay messages (combines acknowledgement and release functionalities)
  + Algorithm:
    - Var: pendingQueue = initially empty with (timestamp, PID)
    - Var: myTimestamp = initially Integer.MAX\_VALUE
    - Var: numOkays = initially 0
    - RequestCS() for Pi:
      * myTimestamp = Pi.logicalClock;
      * send request containing myTimestamp to all other processes
      * numOkay = 0;
    - On receive(Request from Pj):
      * If (Pj.logicalClock < Pi.logicalClock) {
        + Send “okay” to process Pj == (Pj.numOkays++)
      * Else:
        + pendingQueue.enqueue(Pj)
    - Receive(“okay” message) for Pi from Pj:
      * numOkay++;
      * if (numOkay == N – 1):
        + enterCS()
    - Release() for Pi:
      * Pi.myTimestamp = Integer.MAX\_VALUE
      * For process Pj : pendingQueue:
        + Send “okay” message to Pj
      * pendingQueue.clear()
  + **Dining Philosopher’s Algorithm:**
    - Complexity: **O(2 \* (N – 1))**
    - Goal: reduce average complexity of sent messages in coordinating request for C.S.
    - Uses conflict graphs to resolve conflict resolution between two or more processes requiring shared resource(s).
      * Process == vertex
      * Edge == shared resource(s)
      * Source: process with no incoming edges
      * Auxiliary resource: **fork** **(associated with each edge)** => process **Pi** owns the fork if **Pi** has precedence over **Pj**.
        + Var: Boolean isDirty

True: philosopher has eaten from the fork

False: ready to hand fork over to philosopher (process) requesting the fork

* + - * Dining Philosopher’s algorithm **ensures the conflict graph stays acyclic.**
    - Rules of Dining Philosopher:
      * **Eating rule:** a process **Pi** can only eat if it has all forks for the edges incident to the process **Pi**.
      * **Edge-reversal rule:** after eating session is completed, a process reverses orientation of all outgoing edges to incoming edges.
    - Edge precedence cases **(u has precedence over v):**
      * **‘u’ holds the fork and fork is clean**
      * **‘v’ holds the fork and fork is dirty (‘v’ has just finished eating)**
        + **\*\*\*** edge reversal occurs as soon as a process starts to eat
        + \*\*\* **Change in conflict graph occurs only when process starts to eat, not when fork is being cleaned**
      * **fork is in transit from ‘v’ to ‘u’**
    - Revisiting complexity:
      * There will be at most N – 1 requests for forks
      * On average, complexity is less:
        + If a philosopher (process) does not desire the critical section, this philosopher simply eventually relinquishes its forks when incident philosophers request the fork. In this case, **this philosopher will not have to send any messages**.
    - Algorithm:
      * Var: hungry, eating, thinking = Boolean
      * Var: fork(f) = boolean // **process Pi holds fork**
      * Var: request(f) = boolean // **process Pi desires request token for fork**
      * Var: dirty(f) = boolean // **fork is dirty**
      * **Init:**
        + Conflict graph is acyclic
        + All forks are dirty
        + Every request token and fork is held by a different process (philosopher)
      * requestFork(Pi):
        + if Pi.hungry && ~Pi.fork(f) && Pi.request(f):

send request token to process Pi

Pi.request(f) = false;

* + - * releaseFork(Pi):
        + if Pi.request(f) && ~Pi.eating && Pi.dirty(f):

Pi.dirty(f) = false; // clean fork

Pi.fork(f) = false; // release ownership of fork

Send fork f;

* + - * On receive request for request token:
        + Pi.request(f) = true;
      * On receive fork f:
        + Pi.fork(f) = true;
  + **Token-based Algorithms:**
    - **// TODO**
  + **Quorum-based Algorithms:**
    - **//TODO**