03_06_Deadlock Detection

1. Definition

1.1. General Definition

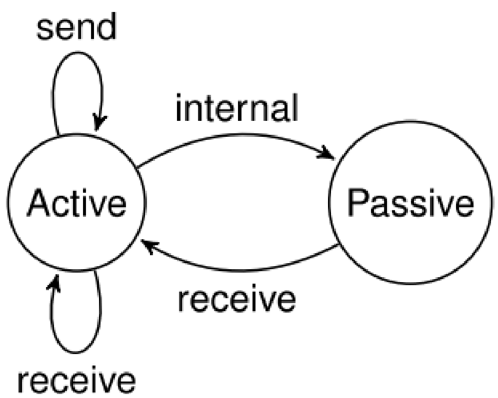
In a transition system a configuration is a **deadlocked or terminating** if there is **no outgoing transition** from that configuration

1. **terminating configuration** is reached when the algorithm **successfully completes** its task

1.2. LTS perspective

A node in isolation, can remain in one of the two states:

1. A process is active, if it has something to execute.
2. Otherwise, it is passive. A passive process will not necessarily always remain passive.



LTS_perspective

1.2.1. Conditions of Deadlock

1. every process is in a passive state
2. all channels are empty

2. Background

If the initiator does not hear from any one for some time, how can it determine whether:

1. the computation is still making progress;
2. it is deadlocks, or
3. it has terminated?

2.1. Why Deadlock Detection

If deadlock is detected one can use a **rollback mechanism**

2.1.1. Three ways to deal with Deadlock

2.2. Types of Request

- **OR request**: one positive reply is sufficient
- **AND request:** all processes must reply positively
- **N-out-of-M request:** (at least) N positive replies needed

3. Wait-For Graph

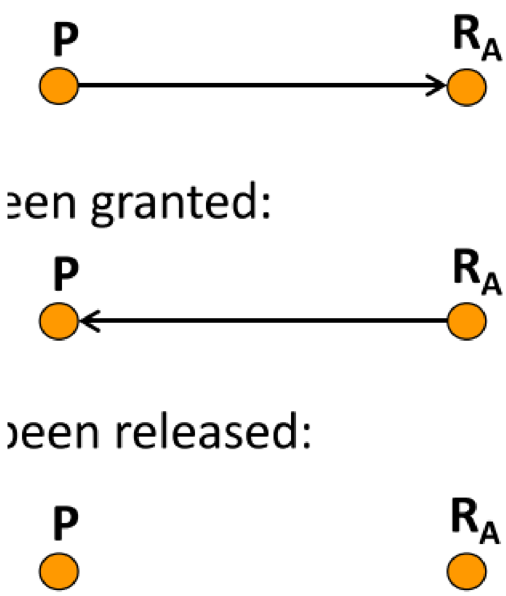
3.1. Resource Mode

Representation

- arrow from process P to resource Q (process R_Q) if P has a request for Q that has not been granted
    - arrow from resource Q to process P if P has acquired Q

Deadlock

deadlock indicated by a **cycle** in the WFG graph

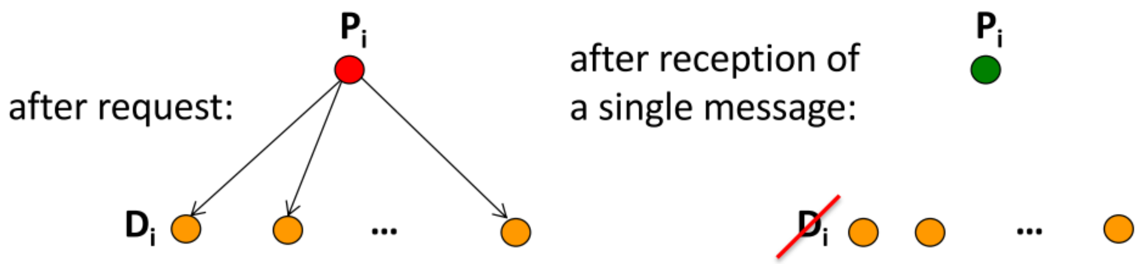


3.2. Communication Model

Representation

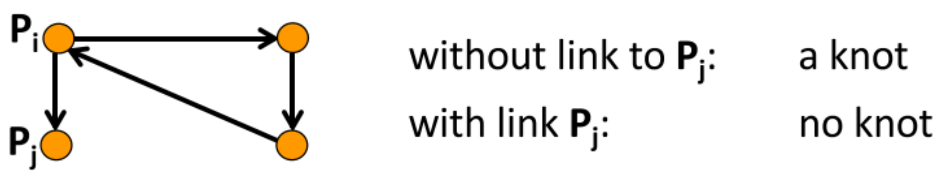
- arrows from process P to all processes from which P expects a single message
- So every blocked process P_i has a **dependent set** D_i of processes on which it waits

- A process turns active when it receives a message **from any** process in its dependent set



Deadlock

- a cycle does not necessarily indicate a deadlock
- There is a deadlock iff **there is a knot** in the WFG
- A **knot**: a set S of nodes with:
  - a path from every node in S to every other node in S
  - no edge from a node in S to a node not in S



4. Algorithms-1

4.1. Candy, Misra and Haas AND model (Resource model)

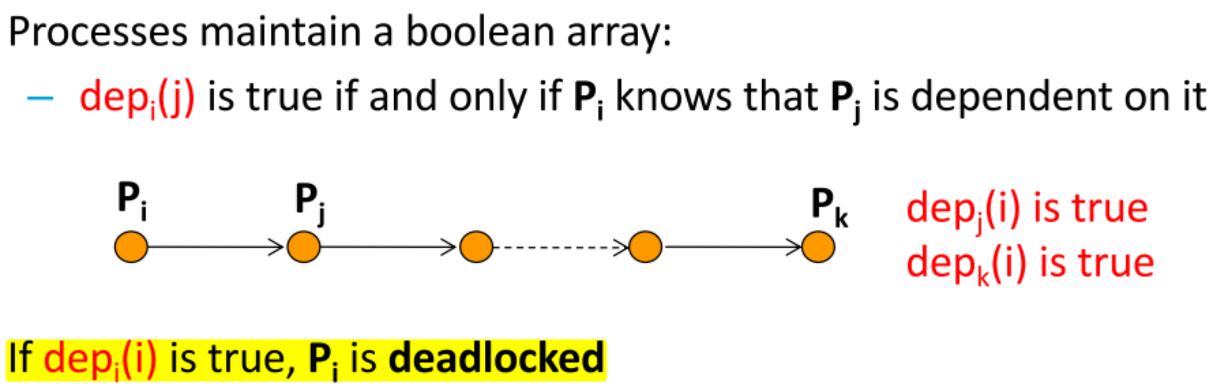
Basic Idea

- when a process **suspects deadlock**, it sends a **special message** to all processes it is waiting for
- those processes **recursively propagate** these messages
- Deadlock if a procss receive a message initially from itselft

Understanding and correctness

- A process waits for all resources it has requested

- Process P_i is dependent on process P_j if **there is a path** in the WFG from P_i to P_j



Attention: This knowing is dynamically, that means the link can always exist, but if Pi has executed, then the value will be false (not "bottleneck")

Process

- A process that suspects it is deadlocked initiates a **probe** by sending out probe messages to the processes it is waiting for
- Processes receiving a probe message **propagate it further** to the processes they are waiting for
- A process is deadlocked when it receives a probe message corresponding to a probe initiated by itself
- A probe message initiated by process P_i and sent by process P_j to process P_k has parameters (i, j, k)
- P_j sends probe (i, j, k) if
  - P_j is blocked
  - P_j is waiting for P_k
  - P_j knows that P_i is dependent on it (except when i=j, when the probe is initiated

Implementation

I. Initiating a probe

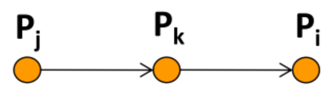
for all (P_j for which P_i is waiting) **do**
send(probe(i,i,j)) /* initiator, source, destination */

II. Process P_i becomes executing

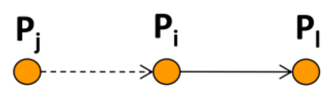
for all j **do** $dep_i(j)$:= false /* reset dependencies */

III. Receiving a probe message by P_i

upon receipt of (probe(j,k,i)) **do**
if ((P_i is blocked) **and** ($dep_i(j)$=false) **and**
(P_i has not replied to all requests of P_k)) **then**
$dep_i(j)$:= true /* record dependency */



if (i=j) **then** P_j is deadlocked /* receive own probe */
else
for all (P_l for which P_i is waiting) **do**
send(probe(j,i,l)) /* propagate probe */



Note:

if P_i has replied to requests of P_k, and still receive probe (j,k,i), that means the waiting status of P_j is not caused by P_i, and because $dep()$ represents the current dependent situation

4.2. Candy, Misra and Haas OR model (Communication Model)

Main Idea

- When a process **suspects deadlock**, it sends a **special message (a "query")** to all processses it is waiting for
- those processes **recursively propagate these messages**
- when a process has already received the message, **it sends back a reply imediately**
- when a process has exhausted all its links, **it sends back a reply**
- **deadlock** if the process receives a reply from all processes it sent a query to

Understanding

A knot is a "closed universe", we can always find a tree and so for all queries a reply will be received. If there is a link go out and break the universe, from the implementation, this query will not return a **REPLY** because in segement 2, the first block will be executed, no reply will be send

Process

- A process that suspects deadlock initiates a **query** in the system
- Queries of the same process **are numbered consecutively**
- **Reply** messages may be sent in reply to query messages
- **Parameters** of queries and replies are (i, m, j, k):
  - i is the id of the initiator of the query
  - m is the query sequence number of P_i
  - the message is sent by P_j to P_k

- Structure of the algorithm:
  - wave of query messages along the “dependent-set” links
  - wave of reply messages back along these links (perhaps)
- A process that hears about a certain query for the first time, is **“engaged”** and **propagates** the query further
- A process that receives a query while it has **already been engaged**, sends **a reply back** (avoid cycles, create a tree)
- So a tree with **engager links** is created **until the dependent-set links** have been exhausted
- Aprocess is **deadlocked** if and only if for every QUERY message it has received a corresponding REPLY message

Preparation (Structure)

Data structures in every process $\mathbf{P_i}$:

- **latest(j)**: the **largest query sequence number** of process $\mathbf{P_j}$ it knows about
- **engager(j)**: the id of the process that notifies it about **latest(j)**
- **num(j)**: (# of query messages sent) **minus** (# of reply messages received) for the current query of process $\mathbf{P_j}$
- **wait(j)**: is true if it has been idle ever since it was notified of **latest(j)**

Implementation

I. Initiating a query (on suspicion of deadlock)

latest(i) = latest(i)+1 /* increase query number */

wait(i) = true

for all $j \in D_i$ **do** send(query(i,latest(i),i,j)) /* send out query */

num(i) = $|D_i|$ /* number of replies needed */

II. Receiving a query message by P_i from P_k

upon receipt of query(j,m,k,i) **do** /* process P_j is initiator */

  if (m > latest(j)) **then** /* new query number */

    latest(j) = m /* record it */

    engager(j) = k /* and record id of “engager” */

    wait(j) = true

    for all $l \in D_i$ **do** /* propagate query */

      send(query(j,m,i,l)) /* to own dependent set */

      num(j) = $|D_i|$

  else if ((m=latest(j)) **and** wait(j)) **then** /* if engaged before */

    send(reply(j,m,i,k)) /* send reply back */

A message of the same initiator and the same query number may come back, so the first if is designed

III. Receiving a reply message by P_i from P_k

```
upon receipt of reply(j,m,k,i) do          /* process P_j is initiator */
if ( (m = latest(j)) and wait(j) ) then    /* same query, still in wait */
    num(j) := num(j) - 1
    if ( num(j) = 0 ) then                 /* all replies received */
        if (j = i) then                    /* if initiator, */
            P_i is deadlocked              /* then deadlocked, */
        else                               /* else */
            l := engager(j)                /* send reply back */
            send(reply(j,m,i,l))           /* only to engager */
```

IV. On becoming executing

```
for all j do wait(j) := false              /* stop all messaging */
```

If someone in the "chain" is executing, no message will propagate by it.

5. Bracha and Toueg

5.1. Bracha and Toueg with instaneous message

Main Idea

- Any process (the initiator) that **suspects a deadlock** of the basic computation **can initiate** the algorithm
- Two phases:
  - **notify**: notify nodes that the algorithm has started (create a spanning tree)
  - **simulate**: simulate the granting of resources
    - may unblock processes, which then also grant resources
    - simulates a continuation of the basic computation
- When the **initiator remains blocked** after the execution of the algorithm, it is **deadlocked**

Model

- Processes have two states:
  - **active**: not waiting for another process
  - **blocked:** waiting for a request to be satisfied (process cannot do any further requests)
- Channels are FIFO

- A **transformation** on the WFG corresponds to an action in a single process (e.g., doing a request adds edges to G)
- Let **σ** be a **schedule** (sequence of transformations)
- A node v is **active** if **nv=0**
- **Transformations of a configuration** G: 1
  - **adding k outgoing edges** to v and setting **nv=r**
    - result of doing an r-out-of-k **REQUEST**
  - **deleting an edge (v,u)** and decreasing **nv by 1**. If then **nv=0**, all remaining outgoing edges from v are **deleted** •
    - result of **sending a REPLY to v**

Preparation (Structure)

- Variables in a node:
  - **nv**: number of **REPLIES** v still needs
  - **OUTv**: set of nodes u such that **(v,u)** in WFG
  - **INv**: set of nodes u such that **(u,v)** in WFG
- Messages:
  - **NOTIFY**: wave of outgoing messages at the start involving other processes in the algorithm
  - **DONE**: finish of **NOTIFY** process (also end of grant if able)
  - **GRANT**: grant of resources, means sender is not blocked
  - **ACK**: reply to NOTIFY

Process

Implementation

I. **Initialization** in node **v**:

OUT := {u|(v,u) in E} /* edges from v in WFG */
IN := {u|(u,v) in E} /* edges towards v in WFG */
notified := false /* not yet participated in notify */
free := false /* deadlocked or not */
#granted := 0 /* number of requests granted */

II. **notify()** procedure: /* start of algorithm: */

notified := true /* initiator executes notify() */
for all (w in OUT) send(w,NOTIFY) /* spread NOTIFYs */
if (n=0) **then** grant() /* if active, simulate REPLY */
for all (w in OUT) await(w,DONE) /* end of algorithm: */
/* initiator receives all DONEs */

III. Receiving a NOTIFY in **v** from **u**:

if (not notified) **then** notify() /* NOTIFYs spread outwards */
send(u,DONE) /* DONEs go back */

IV. Procedure **grant()**:

free := true /* execute grant() only once */
for all (w in IN) send(w,GRANT) /* so simulate REPLY */
for all (w in IN) await(w,ACK) /* and wait for ACK */

V. Receiving a GRANT by **v** from **u**:

#granted := #granted+1
if ((not free) **and** (#granted≥n)) **then** grant() /* apply grant() */
send(u,ACK) /* only once */

Attention:

- **wait** operations are **non-blocking**

- the initiator is **not deadlocke**d if and only if at the end of the algorithm, **free=true** in the initiator
- sending **GRANT**s starts at nodes with **n=0**

Correctness

See Slides

understanding

- only a node with n=0 can start **GRANT**
- **DONE** message means the over of the algorithm
- **GRANT** messages are used to **simulate REPLY** messages.

5.2. Bracha and Toueg without instaneous message

Assumption

There maybe message in transit

Main Idea

- 4 kind of Color of edge

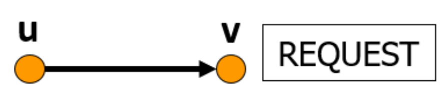
  - **grey edges will become black**
  - **white and translucent edges will disappear**

- Transmation

- Transformations of a configuration **G**:
  1. adding **k grey** outgoing edges to **v** and setting n_v**=r**
     - doing an **r-out-of-k REQUEST**
  2. changing a **grey** edge into a **black** edge
     - receiving a **REQUEST**
  3. changing a **black** edge **(u,v)** into a **white** edge
     - sending a **REPLY** to **u**
  4. removing a **white** edge **(u,v)** and decreasing n_u. If n_u**=0**, then all outgoing edges of **u** are made **translucent**
     - receiving a **REPLY** and possibly sending **RELINQUISH**es
  5. removing a **translucent** edge
     - receiving a **RELINQUISH**

- Color Information
  - Every process sends a **COLOR message** to all processes in its IN and OUT sets, making those processes aware of their membership of these sets.

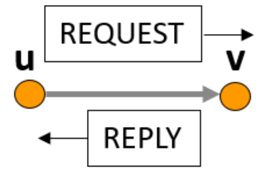
1. **u** in **IN**$_v$ and **v** in **OUT**$_u$:
   - **v** has received **u**'s REQUEST
   - **(u,v)** is **black**



2. **u** in **IN**$_v$ and **v** not in **OUT**$_u$:
   - **v** does not yet know that **u**'s request has been satisfied
   - **(u,v)** is **translucent**

3. **u** not in **IN_v** and **v** in **OUT_u**:

- REQUEST still on its way (**u** not yet in **IN_v**), **or**
- REPLY has been sent but is still on its way (**u** already removed from in **IN_v**)
- **(u,v)** is **grey** or **white** (is all that's needed)



- Objective

- Assume (**optimistically**) that:
  - the grey, white, and translucent edges are **non-existent**
  - so all REQUESTs corresponding to **grey and white edges** will be granted
- Let **#greywhite** be the total number of outgoing grey and white edges
- Then: a node **v** is **active** if $n_v \leq \#greywhite_v$

Remark

- If a deadlock is detectedm then there was indedd one
- There may be a deadlock without detecting it at current time