# June 22, 2016 – Deadlocks and Devices

Deadlocks

- Resource allocation
- Detecting deadlocks
- Resolving deadlocks

Devices

- Hosts and controllers
- I/O
- Drivers

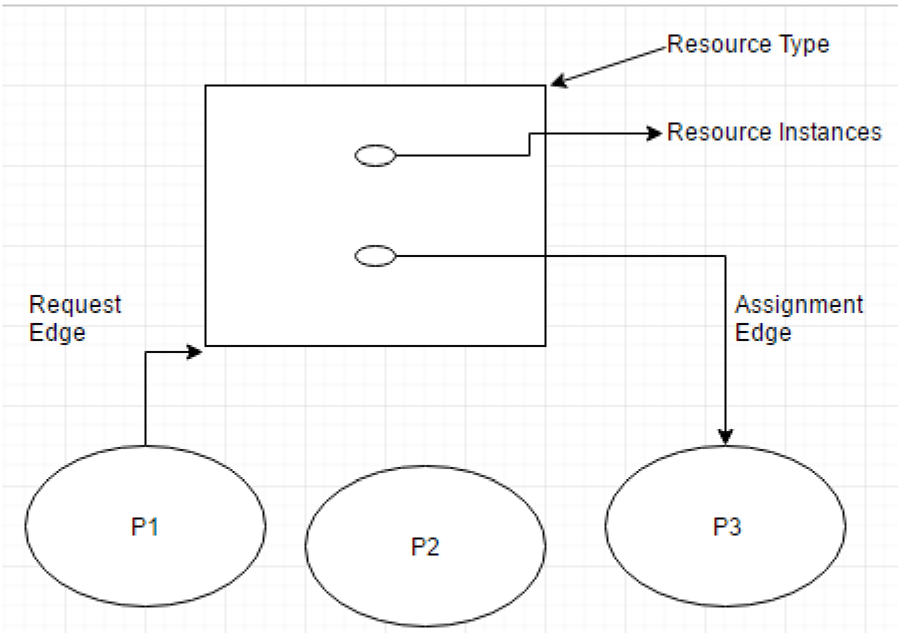
System Model

- Process
- Resources
- Requests
    - Request
    - Use
    - Release

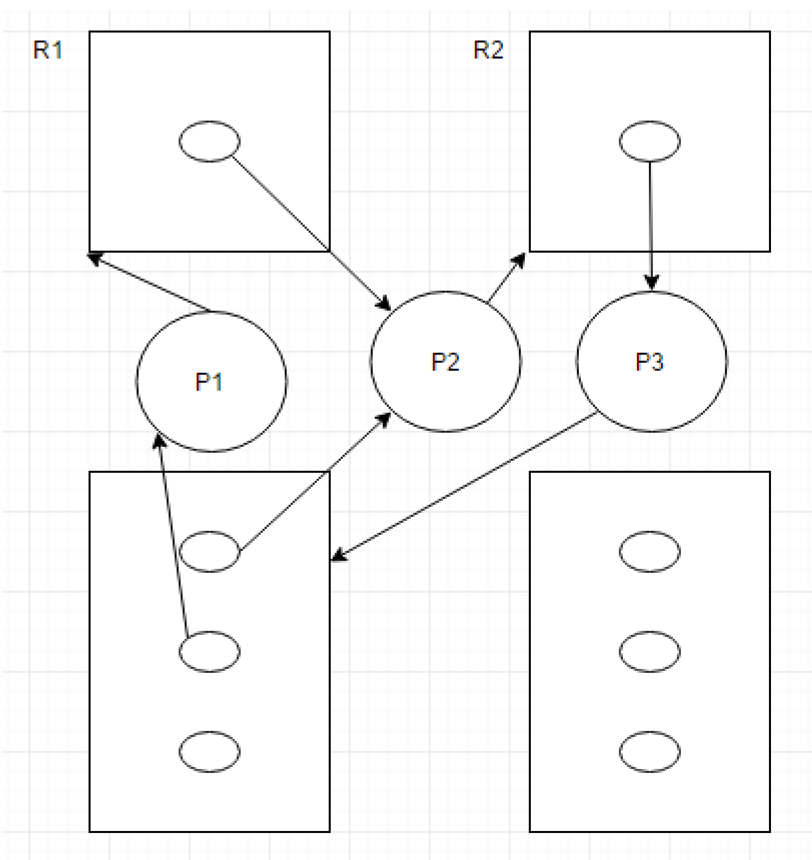
Deadlock Characterization

1. Mutual Exclusion – resources cannot be shared, a second request must be delayed
2. Hold and wait – a process must be holding a resource and waiting for another resource
3. No preemption – resources cannot be taken from a process once it has them
4. Circular waiting – for a set of process {p1…pn} then p1 must wait for p2 which must for p3 … which must wait for Pn which must wait for P1

Resource Allocation Graphs



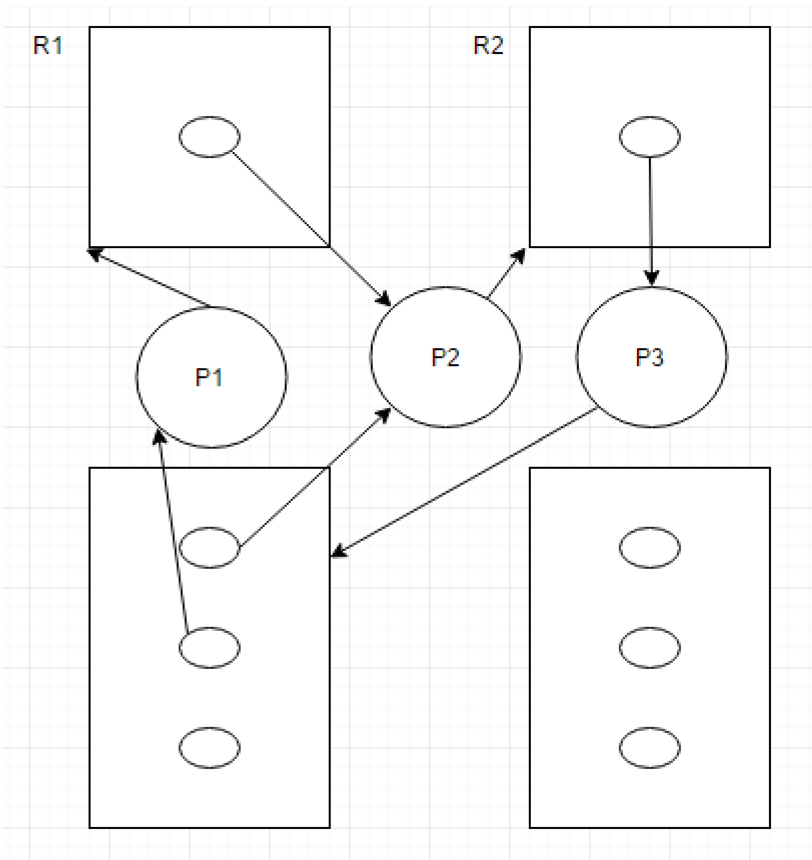

When request edge success = assignment edge (when is done remove it)



Problem: every process is waiting

Handling Deadlocks:

- Let be deadlocks -> let's fix them
- We don't care (not terrible way)
    - Developers manage resources
    - Not worth time




Prevent Deadlocks:

- Break one of the four characterization
- Mutual Exclusion
    - Keep this one
- Hold and Wait (a little wasteful of resources)
    - Require a process request and be allocated all of its resource out the same time
        - If it gets one resource. It gets them all

- No Preemption
  - If a process requests a resource and must wait it must give up all resource it holds
    - Good for easily, saved resources
      - Like a CPU
      - Harder for more complex resources
        - Like a mutex
- Circular Waiting
  - Create a total order of all resources
  - Require process to request resources in increasing order
    - {R1…Rn} F:R -> N
      - F(tape) = 1
      - F(disk) = 2
      - F (printers) = 12
- Protocol
  - A process can initially request any resource but it ran only request resource with higher F(R) value then last received after the first

Deadlock Avoidance

- If we have more information about the behavior of a process (like, which resources a process will require)
- We can determine at runtime which allocations are safe and which are unsafe
- Safe state
  - If there is a safe sequence of resource assignment to process, a process P, can wait and receive the resource it needs from those available and those that will be freed by other process
  - The system is in a safe state, if there is a safe sequence but not all sequences are safe but not all unsafe sequences lead to deadlocks
  - 12drivers / 3 process
- Bankers Algorithm
  - New process declare max # of resources instances they may need

- When actually requested, system checks to see, if that request can be fulfilled, if not the process must wait
- For n processes and m resources

  Available – vector (m) of current, available resources

  Max – matrix [n][m] – max demand

    Max[i][j] is the most instance

    P(i) may request of R(j)

  Allocation – matrix [n][m] – All[i][j] # of R(j) allocated to P1

  Need – matrix[n][m] – need[i][j] - # of R(j) that P1 needs

  Need(i) – row for P(i) from need

    Vector a < vector b

    If for all i Va[i] < Vb[i]

    (1,2,3) < (1,2,4)

- Safety Algorithm
  - Is this system in a safe state?
  - Initialize: work [m] – work = available

      Finish [n] – finish [i] = false for all i

  - 4. find an index: such that finish [i] = false

      And need [i] <= work

      If no such i, go to step 3

  - 2. Work = set work + allocation and finish [i] = true

      Go to step 1

  - 3. If finish [i] = true for all, then the system is safe

- Example:

| ALL | MAX | NEED |
|---|---|---|
| ABC | ABC | ABC |
| 010 | 753 | 743 |
| 200 | 322 | 122 |
| 302 | 902 | 600 |
| 211 | 222 | 011 |
| 002 | 433 | 431 |

Available

ABC

332

0 = work [3,2,2] finish = [F,F,F,F,F]

Choose P1

Work [5,3,2] – finish = [F, T, F, F, F,]

1 = Choose P3

Work [7,4,3,] – finish = [F, T, F, T, F]

2= P0

Work [7,5,3] – finish = [T, T, F, T, F]

3 = P2

Work [9,6,4] – finish = [T, T, T, T, F]

P4

Work [9,6,6] – finish = [T, T, T, T, T]

- Resource Request Allocation
  - A) request is the request vector for Pi
  - B) if request > max – error condition
  - C) if request > A variable – P1 must wait
  - D) otherwise make a copy of the tables such that
    - Available = available – request(i)
    - Allocation(i) = allocation(i) + request(i)
    - Need = need(i) – request(i)
  - If the resulting sate is safe (see safely), request i can be allocated and set the tables to the copies
  - Request(i) = (1,0,2)
  - Available = (2,3,0), Allocation = (3,0,2) Need = (0,2,0)

Deadlock Detection

- Want an algorithm to examine the state of our system and an algorithm to help recover
- Bankers algorithm for deadlock detection
  - Available – vector [m] of the resources available
  - Allocation – matrix [n][m] current requests of each process
  - A) let work be a vector of m and finish be a vector of n
    - For all j if allocation(i) != 0 then finish [i] = false
    - Otherwise finish [i] = true
  - B) find an index I such that finish [i] = false
    - And request(i) <= work
  - C) set work = work + allocation(i), finish [i] = true, go to B
  - D) if finish [i] == false
    - For some I, the system is deadlocked

Example:

| | Allocation | Request |
|---|---|---|
| P0 | 2 4 1 | 0 0 2 |
| P1 | 5 1 3 | 1 1 1 |
| P2 | 3 3 1 | 0 1 6 |

Available

0 0 1

0: work (0,0,1), finish [F,F,F]

1: go to a)

2: system is deadlocked

Example:

Only available changes to 002

0: work = (0,0,2) finish [F,F,F]

1: P0 – work = [2,4,3]

2: P1 – work = [7,5,6]     finish = [T,T,F]

3: P2 – work = [10,8,7]   finish = [T,T,T]

System is not deadlocked