Examples plus Question

Lab 13

Adnan Mustafa

* Can we avoid a deadlock? Can we detect and recover from a deadlock?
* Safe state
* Deadlock avoidance
* Banker's Algorithm
* Deadlock detection
* Deadlock recovery

Deadlock Avoidance

* The **system knows** the complete sequence of requests and releases for each process.
* The **system decides** for each request whether or not the process should wait in order to avoid a deadlock.
* Each **process declares** the maximum number of resources of each type that it may need.
* The system should always be at a **safe state**.

## Safe state → no deadlock

* + the inverse is not always true. 3
* A state is said to be **safe**, if it has a process sequence
  + {P1, P2,…, Pn}, such that for each Pi,
  + the resources that Pi can still request can be satisfied by the currently available resources plus the resources held by all Pj, where j < i.
* State is safe because OS can definitely avoid deadlock
  + by blocking any new requests until safe order is executed
* This avoids **circular wait** condition
  + Process waits until safe state is guaranteed
* Suppose there are 12 tape drives

|  |  |  |
| --- | --- | --- |
|  | **Max Needs** | **Current Needs** |
| p0 | 10 | 5 |
| p1 | 4 | 2 |
| p2 | 9 | 2 |

* + 3 drives remain
* Current state is safe because a safe sequence exists: <p1,p0,p2>
  + p1 can complete with current resources
  + p0 can complete with current+p1
  + p2 can complete with current +p1+p0
* If p2 requests 1 drive, then it must wait to avoid unsafe state.

# Resource-Allocation Graph Algorithm

### Works only if each resource type has **one**



R1

P1

P2

R2

instance.

* Algorithm:
  + Add a **claim edge**, Pi → Rj, if Pi can request Rj in the future
  + Represented by a dashed line in graph
* A request Pi → Rj can be granted only if:
  + Adding an assignment edge Rj → Pi does not introduce cycles



R1

P1

P2

R2

* + - (since cycles imply unsafe state)
* Applicable to resources with **multiple instances**.
* Less efficient than the resource-allocation graph scheme.
* Each process declares its needs (number of resources)
* When a process requests a set of resources:
  + Will the system be at a safe state after the allocation?
    - Yes → Grant the resources to the process.
    - No → Block the process until the resources are released by some other process.

n: integer # of processes

m: integer # of resource-types

available[1..m] available[i] is # of avail resources of type i max[1..n,1..m] max demand of each Pi for each Ri allocation[1..n,1..m] current allocation of resource Rj to Pi need[1..n,1..m] max # resource Rj that Pi may still request

* If request[i] > need[i] then
  + error (asked for too much)
* If request[i] > available[i] then
  + wait (can’t supply it now)
* Resources are available to satisfy the request
  + Let’s assume that we satisfy the request. Then we would have:
    - available = available - request[i]
    - allocation[i] = allocation [i] + request[i]
    - need[i] = need [i] - request [i]
  + Now, check if this would leave us in a safe state:
    - If yes, grant the request,
    - If no, then leave the state as is and cause process to wait.
* **Safety Algorithm**

work[1..m] = available /\* how many resources are available \*/ finish[1..n] = false (for all i) /\* none finished yet \*/

**Step 1:**

Find an i such that finish[i]=false and need[i] <= work /\* find a proc that can complete\*/

/\* its request now \*/

If no such i exists, go to step 3 /\* we’re done \*/

**Step 2:** Found an i:

finish [i] = true /\* done with this process \*/

work = work + allocation [i] /\* assume this process were to finish, \*/

/\*and its allocation back to the available list \*/

go to step 1

**Step 3:** If finish[i] = true for all i, the system is safe. Else Not

#### Allocation Max Available A B C A B C A B C

|  |  |  |  |
| --- | --- | --- | --- |
| **P0** | **0 1 0** | **7 5 3** | **3 3 2** |
| **P1** | **2 0 0** | **3 2 2** |  |
| **P2** | **3 0 2** | **9 0 2** |  |
| **P3** | **2 1 1** | **2 2 2** |  |
| **P4** | **0 0 2** | **4 3 3** |  |

* This is a safe state: safe sequence <P1, P3, P4, P2, P0>
* Suppose that P1 requests (1,0,2)
  + Add it to P1’s allocation and subtract it from Available.

**Allocation Max Available A B C A B C A B C**

|  |  |  |  |
| --- | --- | --- | --- |
| **P0** | **0 1 0** | **7 5 3** | **2 3 0** |
| **P1** | **3 0 2** | **3 2 2** |  |
| **P2** | **3 0 2** | **9 0 2** |  |
| **P3** | **2 1 1** | **2 2 2** |  |
| **P4** | **0 0 2** | **4 3 3** |  |

* This is still safe: safe seq <P1, P3, P4, P0, P2>
* In this new state,P4 requests (3,3,0)
  + Not enough available resources.
* P0 requests (0,2,0)
  + Let’s check resulting state...

**Allocation Max Available A B C A B C A B C**

|  |  |  |  |
| --- | --- | --- | --- |
| **P0** | **0 3 0** | **7 5 3** | **2 1 0** |
| **P1** | **3 0 2** | **3 2 2** |  |
| **P2** | **3 0 2** | **9 0 2** |  |
| **P3** | **2 1 1** | **2 2 2** |  |
| **P4** | **0 0 2** | **4 3 3** |  |

* This is unsafe state (why?).
* So P0’s request will be denied.
* We saw that you can **prevent** deadlocks.
  + By **negating** one of the four necessary conditions.
* We saw that the OS can schedule processes in a careful way so as to **avoid** deadlocks.
  + Using a resource allocation graph.
  + **Banker’s algorithm**.

### What are the downsides to these?

Deadlock Detection

* If neither avoidance or prevention is implemented, deadlocks can (and will) occur.
* Coping with this requires:
  + **Detection**: finding out if deadlock has occurred
    - Keep track of **resource allocation** (who has what)
    - Keep track of **pending requests** (who is waiting for what)
  + **Recovery**: resolve the deadlock

# Using the RAG Algorithm to detect deadlocks

* Suppose there is only one instance of each resource
* Example 1: Is this a deadlock?
  + P1 has R2 and R3, and is requesting R1
  + P2 has R4 and is requesting R3
  + P3 has R1 and is requesting R4
* Example 2: Is this a deadlock?
  + P1 has R2, and is requesting R1 and R3
  + P2 has R4 and is requesting R3
  + P3 has R1 and is requesting R4
* Use a **wait-for graph:**
  + Collapse resources
  + An edge Pi → Pk exists only if RAG has Pi → Rj & Rj → Pk
  + Cycle in wait-for graph → deadlock! 16
* Multiple instances per resource.
* Data structures:

n: integer # of processes

m: integer # of resource-types

available[1..m] available[i] is # of avail resources of type i request[1..n,1..m] current demand of each Pi for each Ri allocation[1..n,1..m] current allocation of resource Rj to Pi finish[1..n] true if Pi’s request can be satisfied

Let request[i] be vector of # instances of each resource Pi wants

* Multiple instances per resource.
* Data structures:

n: integer # of processes

m: integer # of resource-types

available[1..m] available[i] is # of avail resources of type i request[1..n,1..m] current demand of each Pi for each Ri allocation[1..n,1..m] current allocation of resource Rj to Pi finish[1..n] true if Pi’s request can be satisfied

Let request[i] be vector of # instances of each resource Pi wants

* work[]=available[]
* for all i < n, if allocation[i] != 0
  + then finish[i]=false else finish[i]=true
* find an index i such that:
  + finish[i]=false;
  + request[i]<=work
* if no such i exists, go to 7.
* work=work+allocation[i]
* finish[i] = true, go to 3
* if finish[i] = false for some i,
  + then system is deadlocked with Pi in deadlock

**Allocation Request Available**

|  |  |  |
| --- | --- | --- |
| **A B C** | **A B C** | **A B C** |
| **0 1 0** | **0 0 0** | **0 0 0** |
| **2 0 0** | **2 0 2** |  |
| **3 0 3** | **0 0 0** |  |
| **2 1 1** | **1 0 0** |  |
| **0 0 2** | **0 0 2** |  |

**P0 P1 P2 P3 P4**

* The system is not in a deadlocked state.
* What will happen if P2 makes an additional request for a instance of type C?
* **Killing** one/all deadlocked processes
  + Keep killing processes, until deadlock broken
  + Repeat the entire computation
* **Preempt** resource/processes until deadlock broken
  + Selecting a victim (# resources held, how long executed)
  + Rollback (partial or total)
  + Starvation (prevent a process from being executed)

**Lab Questions**

Do your own work. If two solutions are found similar, no credit will be allocated to any of the students. Write to the point answer of the following question.

Consider the following system snapshot using the data structures in the Banker's algorithm, with resources A, B, C, and D, and processes P0 to P4:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **Max** | | | |  | **Allocation** | | | |  | **Need** | | | |  | **Available** | | | |
| **A** | **B** | **C** | **D** | **A** | **B** | **C** | **D** |  | **A** | **B** | **C** | **D** |  | **A** | **B** | **C** | **D** |
| **P0** | 7 | 0 | 2 | 1 | 4 | 0 | 0 | 1 |  |  |  |  |  |  |  |  |
| **P1** | 1 | 6 | 5 | 0 | 1 | 1 | 0 | 0 |  |  |  |  |  |  |  |  |
| **P2** | 3 | 3 | 4 | 6 | 1 | 0 | 4 | 5 |  |  |  |  |  |  |  |  |
| **P3** | 1 | 5 | 6 | 2 | 0 | 4 | 2 | 1 |  |  |  |  |  |  |  |  |
| **P4** | 2 | 4 | 3 | 2 | 0 | 3 | 1 | 2 |  |  |  |  | 3 | 2 | 2 | 1 |

Using the Banker's algorithm, answer the following questions. While executing the Banker's algorithm, if there are multiple processes that may complete on a given cycle, please choose the one with the lowest index.

(b) (3 pts) How many resources of type A, B, C, and D are there?

(c) (5 pts) What are the contents of the Need matrix?

(d) (5 pts) Is the system in a safe state? Why?

(e) (5 pts) If a request from process P2 arrives for additional resources of (0,2,0,0), can the Banker's algorithm grant the request immediately? Why? Show the new system state and other criteria.

(f) (5 pts) Given the original state (from parts a-d), if a request from process P4 arrives for additional resources of (0,2,0,0), can the Banker's algorithm grant the request immediately? Why?