Basics of Deadlock Theory

Deadlocks

- deadlock occurs when there is a set of processes which have outstanding requests for resources that can never be satisfied
- starvation occurs when there are processes waiting for resources that become available and never get assigned to them
- the wait-for-graph (WFG) is a graph with
  - nodes representing processes
  - edges p --> q if process p waits for a resource held by process q
- in studying deadlocks both processes, resources, resource types and accesses, and types of requests need to be considered
Necessary conditions for deadlocks

- Four general conditions are necessary for deadlock to occur
  - exclusive access resources
  - hold and wait
  - no preemption
  - circular wait
- these conditions are not sufficient

Deadlock Handling Strategies

- prevention
  - grant resource requests so that one of the necessary conditions does not hold
- detection & recovery
  - examine resource allocation and pending requests and test for deadlock; if present, recover by aborting some deadlocked processes
- avoidance
  - grant resource requests as long as the system remains in a safe state after resources are allocated
Deadlock Models

- Single-unit requests
- AND-requests
  - process requests multiple resources and stays blocked until all are satisfied
  - cycles in WFG are sufficient for deadlock
- OR-requests
  - process requests multiple resources and stays blocked until any one of them is satisfied
  - cycles in WFG are not sufficient for deadlock; knots are
- AND-OR requests
- P-out-of-Q requests

Resource Types & Accesses

- Reusable resources
  - resource does not “change” when assigned/released to/by processes
  - a resource allocated to one processes P can be allocated to another process after P releases that resource
  - typical reusable resources: CPU, disk, etc
- Consumable resources
  - resource changes (ceases to exist) after is assigned to a process
  - typical consumable resources: messages, signals, semaphore operations
- Resources can be accessed in exclusive or shared mode
**Graph-Model for System State**

- General resource system consists of
  - a set of processes $P_1, P_2, \ldots, P_n$
  - a set of resources (Reusable & Consumable) $R_1, R_2, \ldots, R_m$ with
    - $t_i$, available units for reusable resource $R_i$
    - $Q_i$, a set of producer processes for consumable resource $R_i$

- General resource graph $G$ models system state
  - Bipartite graph with nodes the processes and resources
  - An available units vector $r = (r_1, r_2, \ldots, r_m)$ for all resources
  - Request edges: $(P, R)$ if $P$ requests 1 unit of $R$
  - Assignment edges: $(R, P)$ if 1 unit of $R$ (reusable) is assigned to $P$
  - Producer edges: $(R, P)$ if $P$ is producer of consumable $R$

---

**Example General Resource Graph**

![Example General Resource Graph](image)
**General resource graph**

- **Must satisfy the following conditions**
  - for each reusable resource $R_i$
    - total assigned units of $R_i \leq$ initial number of units of $R_i$
    - available units of $R_i = \text{initial units of } R_i - \text{total assigned units of } R_i$
    - for each process $P_j$
      - assigned units of $R_i$ to $P_j$ + requested units of $R_i$ by $P_j \leq \text{ri}$
  - for each consumable resource
    - the producer edges are proper
    - available units $\geq 0$

- **Process operations and effects on general resource graph**
  - request
  - acquisition
  - release

---

**Example graph**

![Example graph](image-url)
P1 requests two units of R1

P1 acquires one unit of R2
**P2 acquires one unit of R1**

**P2 releases three units of R2**
Graph Reductions

- Reducing general resource graph $G$ by an unblocked process $P_i$
  - for each reusable resource:
    - delete all assignment and request edges
    - for each assignment edge for a reusable resource increment the number of available units for that resource
  - for each consumable resource:
    - delete all assignment, request, and producer edges
    - set available units for that resource to infinity
- A process is *blocked* iff for some resource(s) the number of requested units is more than the available units of that resource

P2 releases one unit of R1

![Diagram showing resource allocation](image)
Graph Reductions & Deadlocks

- **G is completely reducible** iff there exists a sequence of reductions that removes all edges from G.

  **Theorem.** Pi is not deadlocked if there exists a sequence of reductions that takes the system in state where Pi is not blocked.

  **Theorem.** if G is completely reducible then G is deadlock free.

- Graph G is **expedient** if all processes with outstanding requests are blocked.

Cycles, Knots, & Deadlocks

- A **knot** in a graph G is a set K with at least two nodes such that
  - the restriction of G to K is strongly connected
  - there are no nodes in G-K reachable from K

  **Useful fact:** If a graph does not have a knot then there exists a path from every node to a sink node.
**Cycles, Knots, & Deadlocks**

- **Theorem.** In a general resource graph $G$
  - a cycle is necessary for deadlocks
  - a knot is sufficient for a deadlock provided $G$ is expeditent
  
  _Prove it!!_

- **Theorem.** For any process $P_i$ in an expeditent $G$, if there is no path from $P_i$ to a sink then $P_i$ belongs to a knot in $G$ and $P_i$ is deadlocked

---

**Knots are not necessary for deadlock**

![Diagram showing cycles and knots in a resource graph]

- Diagram illustrating cycles and knots in a resource graph.
Single-Unit Systems

Consider a general resource graph $G$ for

- a system where processes can request 1 unit of a resource at a time
  - **Theorem.** If $G$ is expedient then a knot is necessary and sufficient condition for a deadlock.

- a system with single-unit reusable resources only
  - **Theorem.** If $G$ is expedient then a cycle is necessary and sufficient for a deadlock.

Efficient deadlock detection algorithms are possible

System with only Reusable Resources

Consider resource graph $G$

- **Theorem.** All reduction sequences applied to $G$ result in the same state (graph) $G'$.

- **Corollary.** $G$ is deadlock free if and only if $G$ is completely reducible.

Efficient deadlock detection algorithms are possible
**System with only Consumable Resources**

- **Theorem.** If G’s claim-limited graph is completely reducible then G is deadlock-free

- Note that a system state may be deadlock free even though its claim-limited graph is not completely reducible

---

**System with only Consumable Resources**

- Difficult to efficiently detect deadlocks
  - a knot is not necessary for a deadlock to occur
  - different reduction sequences lead into different states

- Can test whether a system is deadlock free using the
  - **claim-limited graph**
    - A general resource graph which corresponds to the worst-case system state
    - the claim-limited graph for a system is constructed by
      - making each consumable resource have zero available units
      - having a request edge (P,R) iff P is a consumer of R
**Deadlock Prevention Methods**

- Grant resource requests in such a way such that one or more of the four necessary conditions for deadlock do not hold
  - process begins only if all its requested resources can be granted
  - blocked processes release resources they hold to other active (higher priority) processes requesting them
  - processes request resources according to a resource priority ordering

**Deadlock Avoidance**

- Assumption: Maximum resource requirements of all processes are known at all times
  - A state is *safe* if there exists a serial process execution sequence where all processes complete
  - Banker’s algorithm for deadlock avoidance
Banker’s Algorithm

Maintain
- **A**: maximum available units (row) vector
- **B**: maximum claim matrix
  one row vector Bi per process Pi denoting maximum resource units ever to be requested by Pi
- **C**: allocation matrix
  one row vector Ci per process Pi denoting resource units allocated to Pi
- **D**: available matrix
  \[ D = A - \text{sum}(C_i, i=1..n) \]
- **E**: need matrix
  \[ E = B - C \]

Correctness conditions
- **Bi <= A** (claim units <= available units)
- **C <= B** (allocated units <= claim units)
- **D >= 0** (total allocated units <= available units)

Pi requests/releases resources with a vector \( \text{Fi <= Ei} \)
- if Pi releases resources \( \text{Fi} \) then
  \[ D = D + \text{Fi} \]
  \[ C_i = C_i + \text{Fi} \]
  \[ E_i = E_i - \text{Fi} \]
- if Pi requests resources \( \text{Fi} \) then
  if \( \text{Fi} > D \) then block Pi
  else test safety of \( \text{Fi} \) and grant or deny its request depending on whether the resulting system state will be safe or not
Testing Safety of a System State given a request

- Initially, label all processes unfinished
- Conditionally grant request $F_i$ of $P_i$
  
  $D = D - F_i$
  $C_i = C_i + F_i$
  $E_i = E_i - F_i$

  While there exists an unfinished $P_i$ such that $E_i \leq D$
  - $D = D + C_i$
  - label $P_i$ as finished
- If all processes are labeled finished then state is safe and the request is granted
- else state is not safe
  - undo all changes to $D$ and process labels due to the while loop
  - undo changes due to conditional granting of $F_i$
  - deny request and block $P_i$

Pros and Cons of Approaches

- Prevention (conservative)
  - unnecessary pre-emptions, restricts concurrency, limits resource utilization
- avoidance (eager pessimistic)
  - unnecessary denials, a priori knowledge of resource needs
  - no process aborts
- detection (lazy optimistic)
  - overhead for detection algorithm
  - process aborts and rollbacks
  - maximum concurrency, flexibility, no prior knowledge is needed
Reading

- Chapter 3 of Singhal & Shivaratri
- R.C. Holt, “Some deadlock properties of computer systems”,