Operating Systems Notes - Chapter 8

July 27, 2025

Contents

1 System model **2**
- 1.1 Introduction . . . 2
- 1.2 Things to learn . . . 2
- 1.3 System model . . . 2
- 1.4 Section glossary . . . 2

2 Deadlock in multithreaded applications **3**
- 2.1 Deadlock Example Code . . . 3
- 2.2 Livelock . . . 3
- 2.3 Livelock Example Code . . . 3
- 2.4 Section glossary . . . 4

3 Deadlock characterization **5**
- 3.1 Necessary conditions . . . 5
- 3.2 Resource-allocation graph . . . 5

4 Methods for handling deadlocks **6**

5 Deadlock prevention **7**

6 Deadlock avoidance **8**
- 6.1 Deadlock detection . . . 9
  - 6.1.1 Single instance of each resource type . . . 9
  - 6.1.2 Several instances of a resource type . . . 9
  - 6.1.3 Detection-algorithm usage . . . 10

7 Recovery from deadlock **11**
- 7.1 Process and thread termination . . . 11
- 7.2 Resource preemption . . . 11

8 Summary **12**

1 System model

1.1 Introduction

- **Multiprogramming Environment**: Multiple threads compete for finite resources.
- **Resource Request**: Threads request resources. If unavailable, thread enters a waiting state.
- **Deadlock**: A situation where a waiting thread can never change state because its requested resources are held by other waiting threads.
- **Formal Definition**: Every process in a set is waiting for an event that can only be caused by another process in the set.

1.2 Things to learn

- Illustrate how deadlock can occur when mutex locks are used.
- Define the four necessary conditions that characterize deadlock.
- Identify a deadlock situation in a resource allocation graph.
- Evaluate the four different approaches for preventing deadlocks.
- Apply the banker's algorithm for deadlock avoidance.
- Apply the deadlock detection algorithm.
- Evaluate approaches for recovering from deadlock.

1.3 System model

- **System Composition**: Finite number of resources distributed among competing threads.
- **Resource Types**:
  - Resources partitioned into types (classes), each with identical instances.
  - Examples: CPU cycles, files, I/O devices.
  - If a system has 4 CPUs, resource type *CPU* has 4 instances.
  - Any instance of a resource type should satisfy a request.
- **Synchronization Tools as Resources**:
  - Mutex locks and semaphores are common sources of deadlock.
  - Each lock instance is typically its own resource class (e.g., one lock for a queue, another for a linked list).
- **Resource Utilization Sequence**:
  1. **Request**: Thread requests a resource. Waits if not immediately available.
  2. **Use**: Thread operates on the resource.
  3. **Release**: Thread releases the resource.
- **System Management**:
  - Request/release can be system calls (`request()`, `release()`, `open()`, `close()`, `allocate()`, `free()`).
  - Or via semaphore operations (`wait()`, `signal()`) and mutex locks (`acquire()`, `release()`).
  - OS checks for resource allocation via a system table.
  - Table tracks free/allocated resources and the owning thread.
  - Waiting threads are queued for requested resources.
- **Deadlocked State**:
  - Every thread in a set is waiting for an event that can only be caused by another thread in the set.
  - Main events: resource acquisition and release.
  - Example: Dining-philosophers problem. Each philosopher holds one chopstick and waits for another, creating a circular wait.
- **Developer Responsibility**:
  - Must be aware of deadlock possibilities.
  - Locking tools prevent race conditions but require careful management of lock acquisition/release to avoid deadlocks.

1.4 Section glossary

Term	Definition
deadlock	The state in which two processes or threads are stuck waiting for an event that can only be caused by one of the processes or threads.

2 Deadlock in multithreaded applications

- **Pthreads Example**: Illustrates deadlock with POSIX mutex locks.
  - `pthread_mutex_init()`: Initializes an unlocked mutex.
  - `pthread_mutex_lock()`: Acquires a lock; blocks if the lock is held.
  - `pthread_mutex_unlock()`: Releases a lock.
- **Deadlock Scenario**:
  - Two mutex locks created: `first_mutex`, `second_mutex`.
  - Two threads, `thread_one` and `thread_two`, access both locks.
  - `thread_one` locks in order: (1) `first_mutex`, (2) `second_mutex`.
  - `thread_two` locks in order: (1) `second_mutex`, (2) `first_mutex`.
  - **Deadlock possible**: If `thread_one` acquires `first_mutex` and `thread_two` acquires `second_mutex`, both threads will block waiting for the other's lock.

2.1 Deadlock Example Code

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   pthread_mutex_lock(&first_mutex);
   pthread_mutex_lock(&second_mutex);
   /**
     * Do some work
     */
   pthread_mutex_unlock(&second_mutex);
   pthread_mutex_unlock(&first_mutex);

   pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
   pthread_mutex_lock(&second_mutex);
   pthread_mutex_lock(&first_mutex);
   /**
    * Do some work
    */
   pthread_mutex_unlock(&first_mutex);
   pthread_mutex_unlock(&second_mutex);

   pthread_exit(0);
}
```

- **Intermittent Nature**:
  - Deadlock might not occur if one thread acquires and releases both locks before the other thread starts.
  - Occurrence depends on the CPU scheduler.
  - Makes identifying and testing for deadlocks difficult.

2.2 Livelock

- **Liveness Failure**: Similar to deadlock, but threads are not blocked.
- **Condition**: A thread continuously attempts an action that fails.
- **Analogy**: Two people trying to pass in a hallway, repeatedly moving into each other's way. They are active but make no progress.
- **Pthreads Example with** `pthread_mutex_trylock()`:
  - `pthread_mutex_trylock()` attempts to acquire a lock without blocking.
  - **Livelock Scenario**: `thread_one` acquires `first_mutex`, `thread_two` acquires `second_mutex`. Both then call `trylock` on the other mutex, which fails. They release their locks and repeat indefinitely.

2.3 Livelock Example Code

```
/* thread_one runs in this function */
void *do_work_one(void *param)
{
   int done = 0;

   while (!done) {
      pthread_mutex_lock(&first_mutex);
      if (pthread_mutex_trylock(&second_mutex)) {
         /**
          * Do some work
          */
```

```
        pthread_mutex_unlock(&second_mutex);
        pthread_mutex_unlock(&first_mutex);
        done = 1;
    }
    else
        pthread_mutex_unlock(&first_mutex);
  }
  pthread_exit(0);
}

/* thread_two runs in this function */
void *do_work_two(void *param)
{
  int done = 0;

  while (!done) {
    pthread_mutex_lock(&second_mutex);
    if (pthread_mutex_trylock(&first_mutex)) {
        /**
         * Do some work
         */
        pthread_mutex_unlock(&first_mutex);
        pthread_mutex_unlock(&second_mutex);
        done = 1;
    }
    else
        pthread_mutex_unlock(&second_mutex);
  }

  pthread_exit(0);
}
```

- **Avoidance**:
  - Livelock often occurs when threads retry failing operations at the same time.
  - Can be avoided by having threads retry at random times.
  - **Ethernet Example**: Hosts involved in a network collision **backoff** for a random period before retransmitting.
- **Rarity**: Less common than deadlock, but still a challenge in concurrent application design.

2.4 Section glossary

Term	Definition
livelock	A condition in which a thread continuously attempts an action that fails.

3 Deadlock characterization

3.1 Necessary conditions

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

- **Mutual exclusion:** At least one resource must be held in a nonsharable mode.
- **Hold and wait:** A thread must be holding at least one resource and waiting to acquire additional resources held by other threads.
- **No preemption:** A resource can be released only voluntarily by the thread holding it.
- **Circular wait:** A set of waiting threads $\{T_0, T_1, \ldots, T_n\}$ must exist such that T_0 is waiting for a resource held by T_1, T_1 is waiting for a resource held by T_2, ..., T_{n-1} is waiting for a resource held by T_n, and T_n is waiting for a resource held by T_0.

All four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition.

3.2 Resource-allocation graph

Deadlocks can be described using a directed graph called a **system resource-allocation graph**.

- The graph consists of a set of vertices V and a set of edges E.
- Vertices V are partitioned into two types:
  - T $= \{T_1, T_2, \ldots, T_n\}$, the set of all active threads.
  - R $= \{R_1, R_2, \ldots, R_m\}$, the set of all resource types.
- A directed edge from thread T_i to resource type R_j ($T_i \rightarrow R_j$) is a **request edge**; it signifies that T_i has requested an instance of R_j.
- A directed edge from resource type R_j to thread T_i ($R_j \rightarrow T_i$) is an **assignment edge**; it signifies that an instance of R_j has been allocated to T_i.
- If the graph contains no cycles, no thread is deadlocked.
- If the graph contains a cycle, a deadlock may exist.
  - If each resource type has exactly one instance, a cycle implies a deadlock has occurred.
  - If each resource type has several instances, a cycle is a necessary but not a sufficient condition for deadlock.

Section glossary

Term	Definition
system resource-allocation graph	A directed graph for precise description of deadlocks.
request edge	In a system resource-allocation graph, an edge (arrow) indicating a resource request.
assignment edge	In a system resource-allocation graph, an edge (arrow) indicating a resource assignment.

4 Methods for handling deadlocks

- Three ways to deal with deadlock:
  - Ignore problem: pretend deadlocks never occur (most OS, e.g., Linux, Windows).
  - Prevent/Avoid: use protocol to ensure system **never** enters deadlocked state.
  - Detect/Recover: allow system to enter deadlocked state, then detect and recover (some systems, e.g., databases).
- First solution (ignoring) common due to infrequency of deadlocks and cost of other methods.
- Basic approaches can be combined for optimal solution per resource class.

Deadlock Prevention

- Provides methods to ensure at least one necessary condition for deadlock cannot hold.
- Prevents deadlocks by constraining resource request methods.

Deadlock Avoidance

- OS given advance info on resources a thread will request/use.
- OS decides if request can be satisfied or delayed based on:
  - Currently available resources.
  - Resources allocated to each thread.
  - Future requests/releases of each thread.

Deadlock Detection and Recovery

- If no prevention/avoidance, deadlock may arise.
- System provides algorithms to:
  - Examine system state to determine if deadlock occurred.
  - Recover from deadlock.
- Undetected deadlock: system performance deteriorates, resources held, more threads deadlock. Eventually, manual restart needed.
- Manual recovery for other liveness failures (e.g., livelock) may be used for deadlock recovery.

Section glossary

Term	Definition
deadlock prevention	Methods to ensure at least one necessary condition for deadlock cannot hold.
deadlock avoidance	OS method where processes inform OS of resource use; system approves/denies requests to avoid deadlock.

5 Deadlock prevention

- Deadlock occurs if all four necessary conditions hold.
- Prevent deadlock by ensuring at least one condition cannot hold.

Mutual exclusion

- Mutual-exclusion condition must hold for deadlock.
- Sharable resources (e.g., read-only files) do not require mutual exclusion, thus cannot be involved in deadlock.
- Cannot generally prevent deadlocks by denying mutual-exclusion, as some resources are intrinsically nonsharable (e.g., mutex locks).

Hold and wait

- To prevent hold-and-wait:
  - Protocol 1: Thread requests/allocates all resources before execution. (Impractical for dynamic resources).
  - Protocol 2: Thread requests resources only when holding none. Must release all current resources before requesting more.
- Disadvantages of both protocols:
  - Low resource utilization: resources allocated but unused for long periods.
  - Starvation possible: thread waits indefinitely for popular resources.

No preemption

- To prevent no-preemption:
  - Protocol 1: If thread requests resource and must wait, all currently held resources are preempted (implicitly released). Thread restarts when old and new resources are available.
  - Protocol 2: If resources are not available, check if held by waiting thread. If so, preempt from waiting thread and allocate to requesting thread. If not, requesting thread waits. Its resources may be preempted by other requests. Thread restarts when new resources allocated and preempted resources recovered.
- Often applied to resources whose state can be saved/restored (e.g., CPU registers, database transactions).
- Cannot generally apply to mutex locks and semaphores (where deadlocks commonly occur).

Circular wait

- Practical solution: impose total ordering of all resource types.
- Require threads to request resources in increasing order of enumeration.
- Example: $F(first\_mutex) = 1$, $F(second\_mutex) = 5$. Thread must request $first\_mutex$ then $second\_mutex$.
- Alternatively, thread requesting R_j must have released any R_i such that $F(R_i) \geq F(R_j)$.
- If multiple instances of same resource type needed, single request for all must be issued.
- This protocol prevents circular wait (proof by contradiction).
- Developing ordering can be difficult for many locks. Java uses 'System.identityHashCode(Object)' for lock acquisition ordering.
- Lock ordering does not guarantee deadlock prevention if locks acquired dynamically (e.g., 'transaction()' function example).

Section glossary

Term	Definition
deadlock prevention	A set of methods intended to ensure that at least one of the necessary conditions for deadlock cannot hold.
deadlock avoidance	An operating system method in which processes inform the operating system of which resources they will use during their lifetimes so the system can approve or deny requests to avoid deadlock.

6 Deadlock avoidance

- Deadlock-prevention limits how requests are made, ensuring no necessary condition occurs.
- Side effects: low device utilization, reduced system throughput.
- Alternative: require additional info on resource requests.
- System needs to know max resources each thread may need (a priori information).
- Deadlock-avoidance algorithm dynamically examines resource-allocation state to prevent circular-wait.
- Resource-allocation **state**: defined by available/allocated resources and max demands.

Safe state

- A state is **safe** if system can allocate resources to each thread (up to max) in some order and avoid deadlock.
- System is safe if a **safe sequence** exists.
- Safe sequence $< T_1, T_2, \ldots, T_n >$: for each T_i, its resource requests can be met by currently available resources + resources held by all T_j $(j < i)$.
- Safe state is not deadlocked; deadlocked state is unsafe.
- Not all unsafe states are deadlocks, but unsafe states **may** lead to deadlock.
- OS avoids unsafe states as long as state is safe.
- In unsafe state, OS cannot prevent deadlocks; thread behavior controls unsafe states.

Resource-allocation-graph algorithm

- For systems with only one instance of each resource type.
- Introduces **claim edge** ($T_i \to R_j$ dashed line): indicates T_i may request R_j in future.
- Claim edge $T_i \to R_j$ converted to request edge when T_i requests R_j.
- Request edge $T_i \to R_j$ converted to assignment edge when R_j allocated to T_i.
- Assignment edge $R_j \to T_i$ converted to claim edge when R_j released by T_i.
- Resources must be claimed a priori.
- Request granted only if no cycle formed in graph (checked by cycle-detection algorithm).
- Cycle indicates unsafe state.
- Algorithm complexity: $O(n^2)$ operations, where n is number of threads.

Banker's algorithm

- Applicable to systems with multiple instances of each resource type.
- Less efficient than resource-allocation graph scheme.
- When new thread enters, declares max instances of each resource type needed (cannot exceed total system resources).
- Request granted only if allocation leaves system in safe state.
- Data structures:
  - **Available**: vector of length m, number of available resources of each type.
  - **Max**: $n \times m$ matrix, max demand of each thread.
  - **Allocation**: $n \times m$ matrix, resources currently allocated to each thread.
  - **Need**: $n \times m$ matrix, remaining resource need of each thread ($Need[i][j] = Max[i][j] - Allocation[i][j]$).

Safety algorithm

1. Initialize $Work = Available$, $Finish[i] = false$ for $i = 0, 1, \ldots, n-1$.
2. Find index i such that $Finish[i] == false$ and $Need_i \leq Work$. If no such i, go to step 4.
3. $Work = Work + Allocation_i$; $Finish[i] = true$. Go to step 2.
4. If $Finish[i] == true$ for all i, system is in safe state.

- Algorithm complexity: $O(m \times n^2)$ operations.

Resource-request algorithm

- Let $Request_i$ be request vector for thread T_i.
  1. If $Request_i \leq Need_i$, go to step 2. Else, error (thread exceeded max claim).
  2. If $Request_i \leq Available$, go to step 3. Else, T_i must wait (resources unavailable).
  3. Pretend allocation:
     - $Available = Available - Request_i$
     - $Allocation_i = Allocation_i + Request_i$
     - $Need_i = Need_i - Request_i$
  4. If resulting state is safe (using Safety Algorithm), grant request. Else, T_i waits, restore old state.

Section glossary

Term	Definition
safe sequence	Sequence of processes $< P_1, P_2, \ldots, P_n >$ where each P_i's requests can be met by available resources + resources held by P_j $(j < i)$.
claim edge	In resource-allocation graph, an edge indicating a process might claim a resource in the future.
banker's algorithm	Deadlock avoidance algorithm for multiple resource instances, less efficient than resource allocation graph.

6.1 Deadlock detection

- If system doesn't use deadlock-prevention or deadlock-avoidance:
  - Algorithm to determine if deadlock occurred.
  - Algorithm to recover from deadlock.
- Detection-and-recovery overhead: run-time costs, potential losses from recovery.

6.1.1 Single instance of each resource type

- Use **wait-for graph**: variant of resource-allocation graph.
- Obtained by removing resource nodes and collapsing edges.
- Edge $T_i \rightarrow T_j$ in wait-for graph implies thread T_i waiting for thread T_j to release a resource R_q that T_i needs.
- Deadlock exists if wait-for graph contains a cycle.
- Detection: maintain wait-for graph; periodically invoke algorithm to search for cycles.
- Cycle detection: $O(n^2)$ operations, where n is number of vertices.
- BCC toolkit: 'deadlock_detector' tool for Pthreads mutex locks on Linux.
  - Inserts probes to trace 'pthread_mutex_lock()' and 'pthread_mutex_unlock()'.
  - Constructs wait-for graph, reports deadlock if cycle detected.

6.1.2 Several instances of a resource type

- Wait-for graph scheme not applicable.
- Algorithm similar to banker's algorithm, uses:
  - **Available**: vector of length m (number of available resources of each type).
  - **Allocation**: $n \times m$ matrix (resources currently allocated to each thread).
  - **Request**: $n \times m$ matrix (current request of each thread).
    - 'Request[i][j] = k': thread T_i requests k more instances of resource type R_j.
- Detection algorithm steps:
  1. Initialize **Work** = **Available**. For $i = 0, 1, \ldots, n-1$:
     - If **Allocation**$_i \neq 0$, then **Finish**$[i]$ = **false**.
     - Otherwise, **Finish**$[i]$ = **true**.
  2. Find an index i such that both:
     - **Finish**$[i]$ == **false**
     - **Request**$_i \leq$ **Work**

     If no such i exists, go to step 4.
  3. **Work** = **Work** + **Allocation**$_i$; **Finish**$[i]$ = **true**. Go to step 2.
  4. If **Finish**$[i]$ == **false** for some i, $0 \leq i < n$, then thread T_i is deadlocked.
- Algorithm complexity: $O(m \times n^2)$ operations.
- Optimistic attitude: if **Request**$_i \leq$ **Work**, assume T_i will complete and return resources.
- Example: 5 threads (T_0 to T_4), 3 resource types (A, B, C).
  - A: 7 instances, B: 2 instances, C: 6 instances.
  - Initial state:

	Allocation *A B C*	**Request** *A B C*	**Availabl** *A B C*
T_0	0 1 0	0 0 0	0 0 0
T_1	2 0 0	2 0 2	
T_2	3 0 3	0 0 0	
T_3	2 1 1	1 0 0	
T_4	0 0 2	0 0 2	

  - Initial claim: system not deadlocked. Sequence $< T_0, T_2, T_3, T_1, T_4 >$ results in **Finish**$[i]$ == **true** for all i.
  - If T_2 requests 1 additional instance of C:

	Request *A B C*
T_0	0 0 0
T_1	2 0 2
T_2	0 0 1
T_3	1 0 0
T_4	0 0 2

  - New claim: system is deadlocked. Can reclaim T_0's resources, but not enough for others. Deadlock involves T_1, T_2, T_3, T_4.

6.1.3 Detection-algorithm usage

- When to invoke depends on:
  1. How often is a deadlock likely to occur?
  2. How many threads will be affected by deadlock when it happens?
- Frequent deadlocks $\implies$ frequent invocation.
- Resources allocated to deadlocked threads become idle; number of threads in deadlock cycle may grow.
- Extreme: invoke every time a resource request cannot be granted immediately.
  - Identifies deadlocked threads and the specific thread that "caused" the deadlock.
  - High computational overhead.
- Less expensive: invoke at defined intervals (e.g., hourly, when CPU utilization drops below 40%).
  - May not identify the "causing" thread.

Managing deadlock in databases

- Database systems manage deadlock using detection and recovery.
- Updates as **transactions**; locks used for data integrity.
- Deadlocks possible with multiple concurrent transactions.
- Database server periodically searches for cycles in the wait-for graph.
- When deadlock detected:
  - A victim transaction is selected.
  - Victim transaction is aborted and rolled back, releasing its locks.
  - Remaining transactions are freed from deadlock.
  - Aborted transaction is reissued.
- Victim choice: e.g., MySQL minimizes rows inserted, updated, or deleted.

Section glossary

Term	Definition
wait-for graph	In deadlock detection, a variant of the resource-allocation graph with resource nodes re moved; indicates a deadlock if the graph contains a cycle.
thread dump	In Java, a snapshot of the state of all threads in an application; a useful debugging tool fo deadlocks.

7 Recovery from deadlock

- When deadlock detected, options:
  - Inform operator (manual recovery).
  - System recovers automatically.
- Two options for breaking deadlock:
  - Abort one or more threads (break circular wait).
  - Preempt resources from deadlocked threads.

7.1 Process and thread termination

- Eliminate deadlocks by aborting process/thread.
- System reclaims all resources.
- Methods:
  - **Abort all deadlocked processes**: Breaks cycle, but expensive (discarded computations, recomputation needed).
  - **Abort one process at a time until deadlock eliminated**: High overhead (deadlock-detection after each abort).
- Aborting process issues:
  - File in incorrect state if updating.
  - Shared data integrity issues if updating while holding mutex lock (must restore lock status).
- If partial termination, determine which process to terminate (policy decision, economic).
- Factors for choosing victim (minimum cost):
  - Process priority.
  - Computation time (how long computed, how much longer).
  - Resources used (types, ease of preemption).
  - Resources needed to complete.
  - Number of processes to terminate.

7.2 Resource preemption

- Successively preempt resources from processes, give to others until deadlock broken.
- Three issues:
  - **Selecting a victim**: Which resources/processes to preempt? Minimize cost (e.g., resources held, time consumed).
  - **Rollback**: What to do with preempted process?
    - Cannot continue normal execution (missing resource).
    - Roll back to safe state, restart.
    - Simplest: total rollback (abort, restart).
    - More effective: roll back only as necessary (requires more state info).
  - **Starvation**: How to ensure resources not always preempted from same process?
    - Process never completes.
    - Ensure process picked as victim finite number of times.
    - Common solution: include number of rollbacks in cost factor.

Section glossary

Term	Definition
recovery mode	A system boot state providing limited services and designed to enable the system admin to repair system problems and debug system startup.

8 Summary

- Deadlock: set of processes, each waiting for event caused by another process in set.
- Four necessary conditions for deadlock:
  - Mutual exclusion.
  - Hold and wait.
  - No preemption.
  - Circular wait.
- Deadlock only possible if all four conditions present.
- Deadlocks modeled with resource-allocation graphs; cycle indicates deadlock.
- Deadlock prevention: ensure one of four conditions cannot occur.
- Practical prevention: eliminate circular wait.
- Deadlock avoidance: use banker's algorithm; don't grant resources if leads to unsafe state.
- Deadlock detection: algorithm evaluates processes/resources on running system to find deadlocked state.
- Deadlock recovery:
  - Abort one process in circular wait.
  - Preempt resources assigned to deadlocked process.