Remember example from third week?

My_work(id_t id) { /* id can be 0 or 1 */
    /* entry section */
    flag[id] = true;    /* indicate entering CS */
    while (flag[1-id]) ;/* entry section */
    /* critical section, access protected resource */
    flag[id] = false;   /* exit section */
    /* remainder section */
    ...
}

- What went wrong here?
Types of Resources

- Reusable
  - Can be used by one process at a time, released and used by another process
    - printers, memory, processors, files
    - Locks, semaphores, monitors

- Consumable
  - Dynamically created and destroyed
  - Can only be allocated once
    - e.g. interrupts, signals, messages
Not just an OS Problem!

- Law passed by Kansas Legislature in early 20th Century:
  - “When two trains approach each other at a crossing, both shall come to a full stop and neither shall start upon again until the other has gone.”
Deadlock Defined

- The *permanent* blocking of a set of processes that either:
  - Compete for system resources, or
  - Communicate with each other
- Each process in the set is blocked, waiting for an event which can only be caused by another process in the set
  - Resources are *finite*
  - Processes wait if a resource they need is unavailable
  - Resources may be held by other waiting processes
Example of Deadlock

- Suppose processes $P$ and $Q$ need (reusable) resources $A$ and $B$:
Example: dining philosophers:

- A philosopher needs two forks to eat.
- Idea for protocol:
  - When philosopher gets hungry grab right fork, then grab left fork.
- Is this a good solution?
Deadlock continued …

- What conditions must hold for a deadlock to occur?
  - Necessary conditions
  - Sufficient conditions
Conditions for Deadlock

1. Mutual Exclusion
   - Only one process may use a resource at a time

2. Hold and wait
   - A process may hold allocated resources while awaiting assignment of others

3. No preemption
   - No resource can be forcibly removed from a process holding it

These are necessary conditions
One more condition...

4. Circular wait
   - A closed chain of processes exists, such that each process holds at least one resource needed by the next process in the chain
   - Together, these four conditions are **necessary and sufficient** for deadlock
Solutions

- Prevention
- Avoidance
- Detection and Recovery
- Do Nothing!
Deadlock Prevention

- Ensure one of the four conditions doesn’t occur
  - Break mutual exclusion - not much help here, as it is often required for correctness
Preventing Hold-and-Wait

- Break “hold and wait” - processes must request all resources at once, and will block until entire request can be granted simultaneously
  - May wait a long time for all resources to be available at the same time
  - May hold resources for a long time without using them (blocking other processes)
  - May not know all resource requirements in advance

- An alternative is to release all currently-held resources when a new one is needed, then make a request for the entire set of resources
Preventing No-Preemption

- Break “no preemption” - forcibly remove a resource from one process and assign it to another
  - Need to save the state of the process losing the resource so it can recover later
  - May need to rollback to an earlier state

- Name some resources that this works for…
- Name some resources for which this is hard…
- Impossible for consumable resources
Preventing Circular-wait

- Break “circular wait” - assign a linear ordering to resource types and require that a process holding a resource of one type, $R$, can only request resources that follow $R$ in the ordering.
Preventing Circular-wait

- Break “circular wait” - assign a linear ordering to resource types and require that a process holding a resource of one type, $R$, can only request resources that follow $R$ in the ordering
  - e.g. $R_i$ precedes $R_j$ if $i < j$
  - For deadlock to occur, need P to hold $R_i$ and request $R_j$, while Q holds $R_j$ and requests $R_i$
  - This implies that $i < j$ (for P’s request order) and $j < i$ (for Q’s request order), which is impossible.

- Hard to come up with total order when there are lots of resource types
Deadlock Avoidance

- All prevention strategies are unsatisfactory in some situations.
- *Avoidance* allows the first three conditions, but orders events to ensure circular wait does not occur.
  - How is this different from preventing circular wait?
- Requires knowledge of future resource requests to decide what order to choose.
  - Amount and type of information varies by algorithm.
Two Avoidance Strategies

1. Do not start a process if its maximum resource requirements, together with the maximum needs of all processes already running, exceed the total system resources
   - Pessimistic, assumes all processes will need all their resources at the same time

2. Do not grant an individual resource request if it might lead to deadlock
Safe States

- A state is **safe** if there is at least one sequence of process executions that does not lead to deadlock, *even if every process requests their maximum allocation immediately*.

- **Example:** 3 processes, 1 resource type, 10 instances

<table>
<thead>
<tr>
<th>PID</th>
<th>Alloc</th>
<th>Max Claim</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>B</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>C</td>
<td>2</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>T0: Available = 3</th>
<th>T1: Available = 1</th>
<th>T2: Available = 5</th>
<th>T3: Available = 0</th>
<th>T4: Available = 7</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0: Available = 3</td>
<td>T1: Available = 1</td>
<td>T2: Available = 5</td>
<td>T3: Available = 0</td>
<td>T4: Available = 7</td>
</tr>
</tbody>
</table>
Unsafe States & Algorithm

- An **unsafe** state is one which is not safe
  - Is this the same as a deadlocked state?
- **Deadlock avoidance algorithm**
  - For every resource request
    - Update state assuming request is granted
    - Check if new state is safe
    - If so, continue
    - If not, restore the old state and block the process until it is safe to grant the request
- This is the **banker’s algorithm**
  - Processes must declare maximum needs
  - See text for details of the algorithm
Restrictions on Avoidance

- Maximum resource requirements for each process must be known in advance
- Processes must be independent
  - If order of execution is constrained by synchronization requirements, system is not free to choose a safe sequence
- There must be a fixed number of resources to allocate
Deadlock Detection & Recovery

- Prevention and avoidance is awkward and costly
  - Need to be cautious, thus low utilization
- Instead, allow deadlocks to occur, but detect when this happens and find a way to break it
  - Check for circular wait condition periodically
- When should the system check for deadlocks?
Deadlock Detection & Recovery

- How can you detect a deadlock?
Draw resource alloc graph

- Check for cycles in resource allocation graph
Deadlock Detection

- Finding circular waits is equivalent to finding a cycle in the resource allocation graph
  - Nodes are processes (drawn as circles) and resources (drawn as squares)
  - Arcs from a resource to a process represent allocations
  - Arcs from a process to a resource represent ungranted requests
- Any algorithm for finding a cycle in a directed graph will do
  - Note that with multiple instances of a type of resource, cycles may exist without deadlock
Deadlock Recovery

- Basic idea is to break the cycle
  - Drastic - kill all deadlocked processes
  - Painful - back up and restart deadlocked processes (hopefully, non-determinism will keep deadlock from repeating)
  - Better - selectively kill deadlocked processes until cycle is broken
    - Re-run detection alg. after each kill
  - Tricky - selectively preempt resources until cycle is broken
    - Processes must be rolled back
Reality Check

- No single strategy for dealing with deadlock is appropriate for all resources in all situations
- All strategies are costly in terms of computation overhead, or restricting use of resources
- Most operating systems employ the “Ostrich Algorithm”
  - Ignore the problem and hope it doesn’t happen often
Why does the Ostrich Alg work?

- Recall causes of deadlock:
  - Resources are *finite*
  - Processes wait if a resource they need is unavailable
  - Resources may be held by other waiting processes

- Prevention/Avoidance/Detection mostly deal with last 2 points

- Modern operating systems *virtualize* most physical resources, eliminating the first problem
  - Some logical resources can’t be virtualized (there has to be exactly one), such as bank accounts or the process table
    - These are protected by synchronization objects, which are now the only resources that we can deadlock on
What is atomicity?

- Recall ATM banking example:
  - Concurrent deposit/withdrawal operation
  - Need to protect shared account balance
- What about transferring funds between accounts?
  - Withdraw funds from account A
  - Deposit funds into account B
- Should appear as a single **atomic** operation
  - Another process reading the account balances should see either both updates, or none
  - Either both operations complete, or neither does
Why would atomicity fail?

- Suppose fund transfer is implemented by our known withdraw and deposit functions using locks.

```c
Withdraw(acct, amt) {
    acquire(lock);
    balance = get_balance(acct);
    balance = balance - amt;
    put_balance(acct,balance);
    release(lock);
    return balance;
}

Deposit(acct, amt) {
    acquire(lock);
    balance = get_balance(acct);
    balance = balance + amt;
    put_balance(acct,balance);
    release(lock);
    return balance;
}

Transfer (acctA, acctB, amt) {
    Withdraw (acctA,amt);
    Deposit (acctB,amt;
}
```

- What can go wrong?
Definitions for Transactions

- **Defn: Transaction**
  - A collection of operations that performs a single logical function and are executed **atomically**
  - Here: a sequence of **read** and **write** operations, terminated by a **commit** or **abort**

- **Defn: Committed**
  - A transaction that has completed successfully;
  - **All** operations took effect
  - Once committed, a transaction **cannot be undone**

- **Defn: Aborted**
  - A transaction that did not complete normally
  - **None** of the operations took effect
How to ensure atomicity in the face of failures?

- Write intended operation to a log on **stable** storage
- Then execute the actual operation
- Log can be used to undo/redo any transaction, allowing recovery from arbitrary failures
Write-ahead logging

- Before performing any operations on the data, write the intended operations to a log on stable storage.
- Log records identify the transaction, the data item, the old value, and the new value.
- Special records indicate the start and commit (or abort) of a transaction.
- Log can be used to undo/redo the effect of any transactions, allowing recovery from arbitrary failures.
Problems with logging …

- Limitations of basic log strategy:
  - Time-consuming to process entire log after failure
  - Large amount of space required by log
  - Performance penalty – each write requires a log update before the data update

- **Checkpoints** help with first two problems
  - Periodically write all updates to log and data to stable storage; write a *checkpoint* entry to the log
  - Recovery only needs to look at log since last ckpt.
Concurrent Transactions

- Transactions must appear to execute in some arbitrary but serial order
  - Soln 1: All transactions execute in a critical section, with a single common lock (or mutex semaphore) to protect access to all shared data.
    - But most transactions will access different data
    - Limits concurrency unnecessarily
  - Soln 2: Allow operations from multiple transactions
    - To overlap, as long as they don’t conflict
    - End result of a set of transactions must be indistinguishable from Solution 1
Conflicting Operations

- Operations in two different transactions conflict if both access the same data item and at least one is a write
  - Non-conflicting operations can be reordered (swapped with each other) without changing the outcome
  - If a serial schedule can be obtained by swapping non-conflicting operations, then the original schedule is conflict-serializable
Conflict Serializability

- Is there an equivalent serial execution of T0 and T1?

<table>
<thead>
<tr>
<th>T₀</th>
<th>T₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>write(A)</td>
<td>write(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td>read(B)</td>
</tr>
<tr>
<td>write(B)</td>
<td>write(B)</td>
</tr>
</tbody>
</table>
## Conflict Serializabile?

<table>
<thead>
<tr>
<th>$T_0$</th>
<th>$T_1$</th>
<th>$T_0$</th>
<th>$T_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>read(A)</td>
<td></td>
<td>read(A)</td>
<td>read(A)</td>
</tr>
<tr>
<td>write(A)</td>
<td></td>
<td>write(A)</td>
<td>write(A)</td>
</tr>
<tr>
<td>read(B)</td>
<td></td>
<td>read(B)</td>
<td>read(B)</td>
</tr>
<tr>
<td>write(B)</td>
<td></td>
<td>write(B)</td>
<td>write(B)</td>
</tr>
</tbody>
</table>

Yes

No
Ensuring serializability

- Two-phase locking
  - Individual data items have their own locks
  - Each transaction has a growing phase and shrinking phase:
    - Growing: a transaction may obtain locks, but may not release any lock
    - Shrinking: a transaction may release locks, but may not acquire any new locks.
  - Does not guarantee deadlock-free
Example of 2 phase locking

Transaction_start
Lock(A)  
Read(A)  
**Lock(B)**  
Read(B)  
**Lock(C)**  
Unlock(A)  
Unlock(B)  
Write(C)  
Unlock(C)  
Transaction_end

Growing
Shrinking
Timestamp Protocols

- Each transaction gets unique *timestamp* before it starts executing
  - Transaction with “earlier” timestamp must appear to complete before any later transactions

- Each data item has two timestamps
  - **W-TS**: the largest timestamp of any transaction that successfully wrote the item
  - **R-TS**: the largest timestamp of any transaction that successfully read the item
Timestamp Ordering

● Reads:
  ● If transaction has “earlier” timestamp than $W$-TS on data, then transaction needs to read a value that was already overwritten
    ● Abort transaction, restart with new timestamp

● Writes:
  ● If transaction has “earlier” timestamp than $R$-TS ($W$-TS) on data, then the value produced by this write should have been read (overwritten) already!
    ● Abort & restart

● Some transactions may “starve” (abort & restart repeatedly)
Deadlock and Starvation

- A set of threads is in a **deadlocked** state when every process in the set is waiting for an event that can be caused only by another process in the set.
- A thread is suffering **starvation** (or indefinite postponement) if it is waiting indefinitely because other threads are in some way preferred.
Communication Deadlocks

- Messages between communicating processes are a consumable resource

Example:
- Process B is waiting for a request
- Process A sends a request to B, and waits for reply
- The request message is lost in the network
- B keeps waiting for a request, A keeps waiting for a reply, we have a deadlock

Solution: Use timeouts and protocols to detect duplicate messages
Livelock

- Occurs when a set of processes continually retry some failed operation and prevent other processes in the set from making progress
- Functionally equivalent to deadlock
  - Ex 1: two processes each request the same two spinlocks in the opposite order
    - Each succeeds in first acquire, then spins
    - CPU utilization is high, but no progress
  - Ex 2: A set of processes retries a failed fork()
    - operation when the process table is full
    - No process exits, so fork() keeps failing