CSC 369

Week 12: Concurrency Bugs,

Deadlocks and Transactions

University of Toronto, Department of Computer Science

Plan for This Topic

- Concurrency bugs:
- Non-deadlock bugs
- Deadlocks
  - Defining deadlock
  - Conditions for deadlock to occur
  - How to deal with deadlocks:
    - Deadlock Prevention
    - Deadlock Avoidance
    - Deadlock Detection (and recovery)
    - The OS approach
- Transactions (briefly..)

Non-deadlock bugs

- Most common occurrence
- Two types
  - Atomicity violation bugs
    - When a code region is intended to be atomic, but the atomicity is not enforced during execution
  - Order violation bugs
    - When the desired order between memory accesses is flipped (i.e., when a certain order is assumed, but not specifically enforced)

Atomicity violation bugs

- Example:

```
Thread 1:
if (inodes[10].links_count == 1) {
    inodes[10].links_count--;
    remove_inode();
}

Thread 2:
    inodes[10].links_count ++;
```

- Notice the issue? How do we fix it?
  - Atomicity assumption! If it gets broken, we have a problem.
  - Must use locks around the critical sections

Order violation bugs

- Example (recall A1):

```
Say we initialize the pidlist for SYS_read when
intercepting it (in my_syscall)
  INIT_LIST_HEAD (&(table[SYS_read].my_list));
  ...

In my_exit_group:
  del_pid(pid);
```

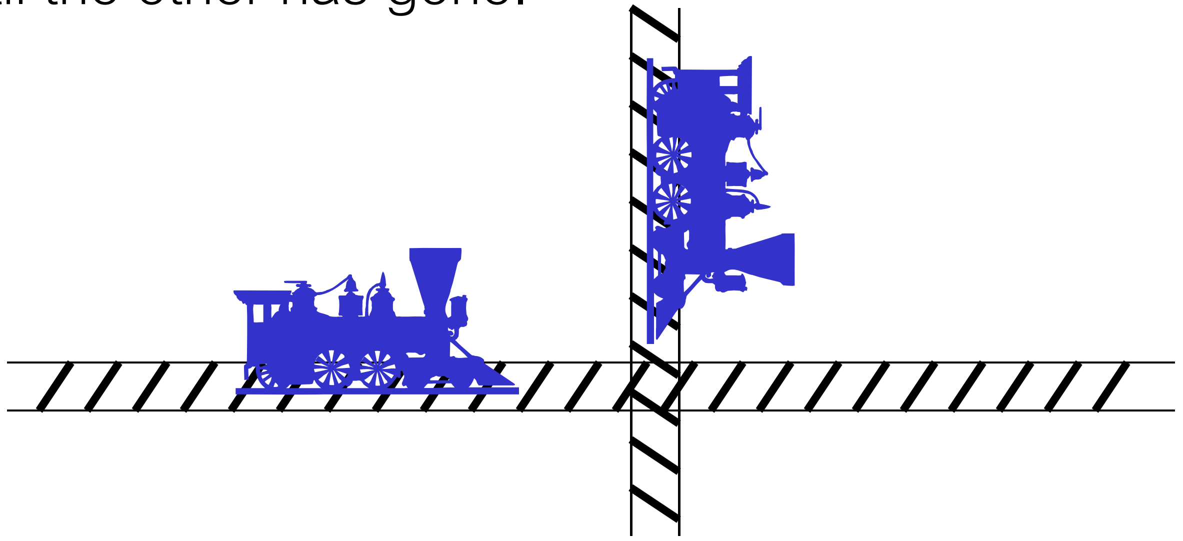
- Recall the problem?
  - In del_pid, all the pidlists are assumed to have been initialized!
  - Fix by enforcing ordering, or extra checks for my_list
  - Of course, we could just initialize everything in the init_function

Deadlocks

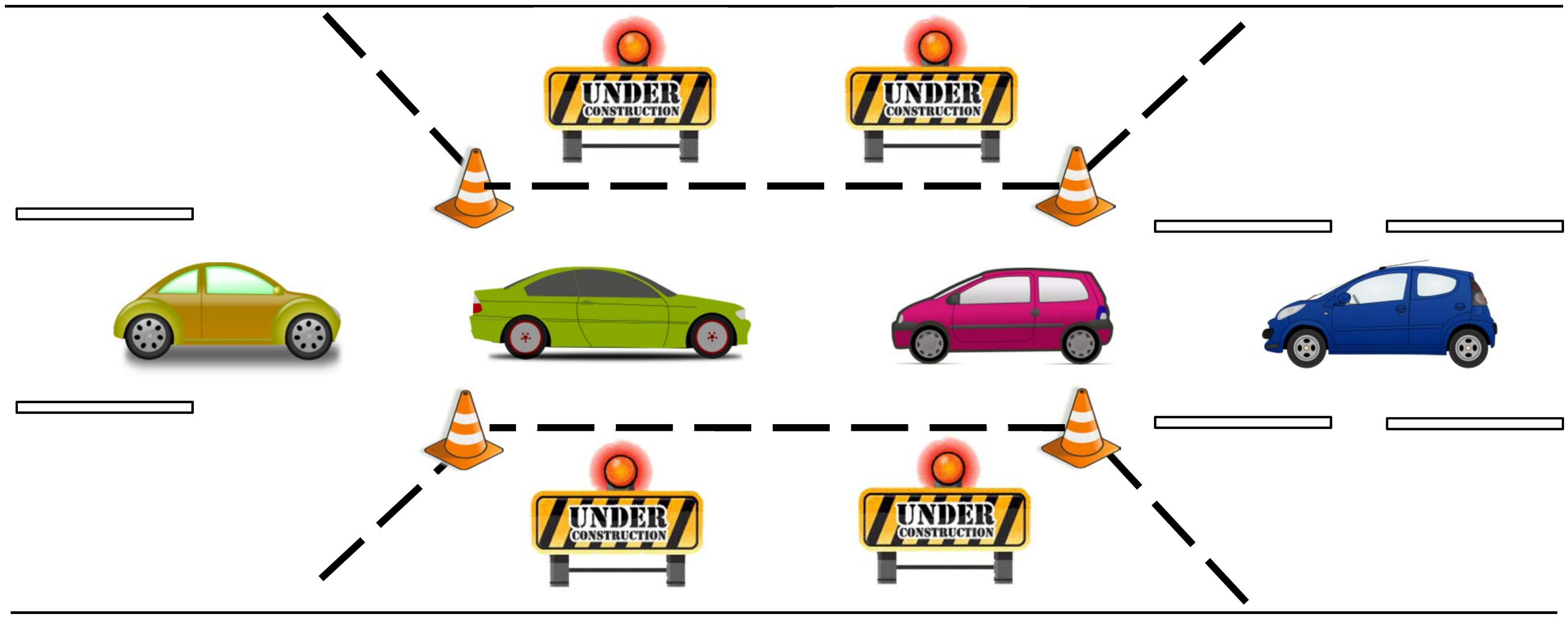
- The *mutual* blocking of a set of processes or threads
- Each process in the set is blocked, waiting for an event which can only be caused by another process in the set
- You've seen this informally in A2!

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.”



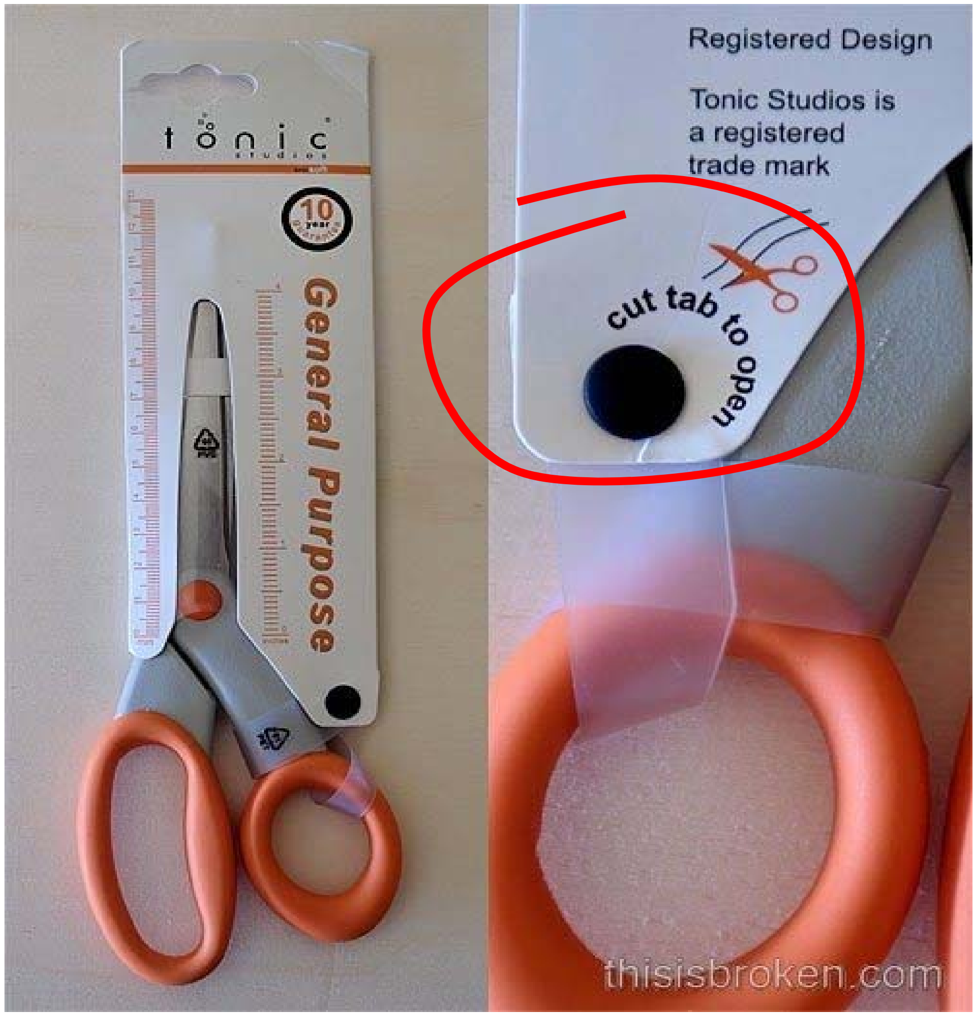
Deadlocks in real life ...



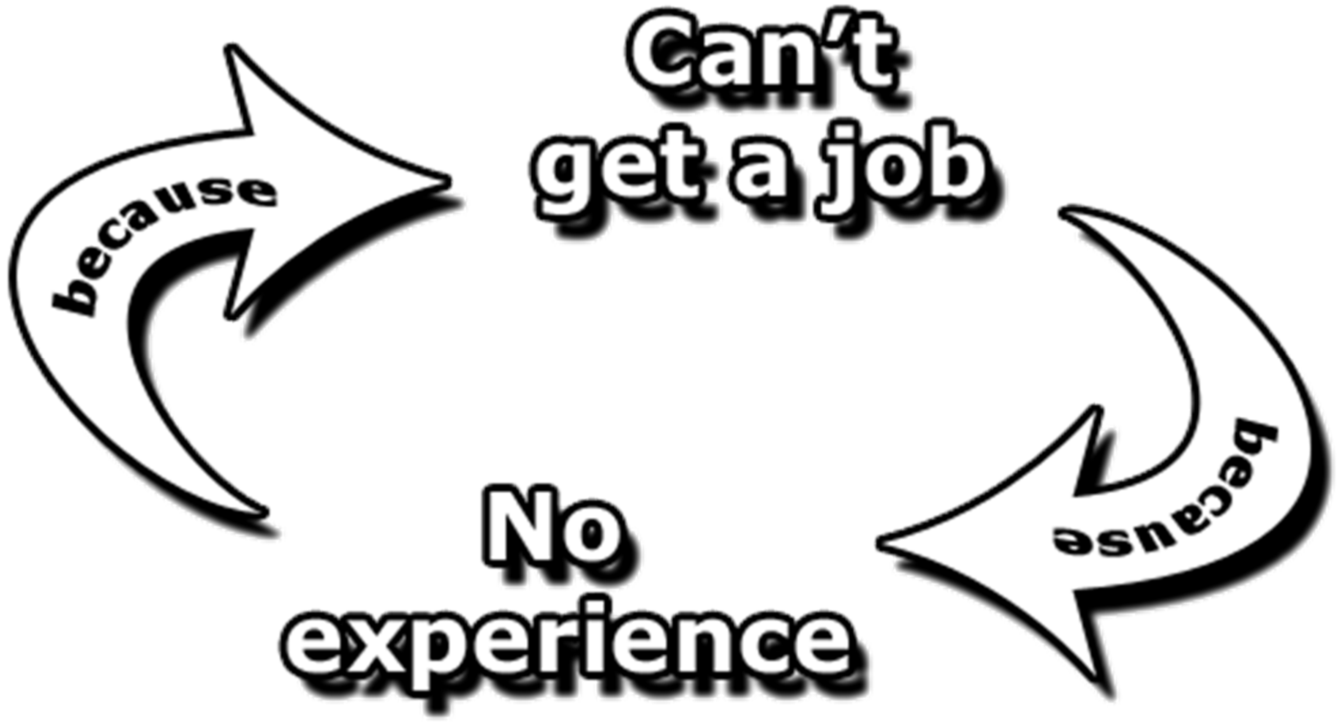
Deadlock

- Traffic jam

Deadlock


tonic studios
10 year guarantee
General Purpose
Registered Design
Tonic Studios is a registered trade mark
cut tab to open
thisisbroken.com

Sounds familiar .. ?



Deadlocks

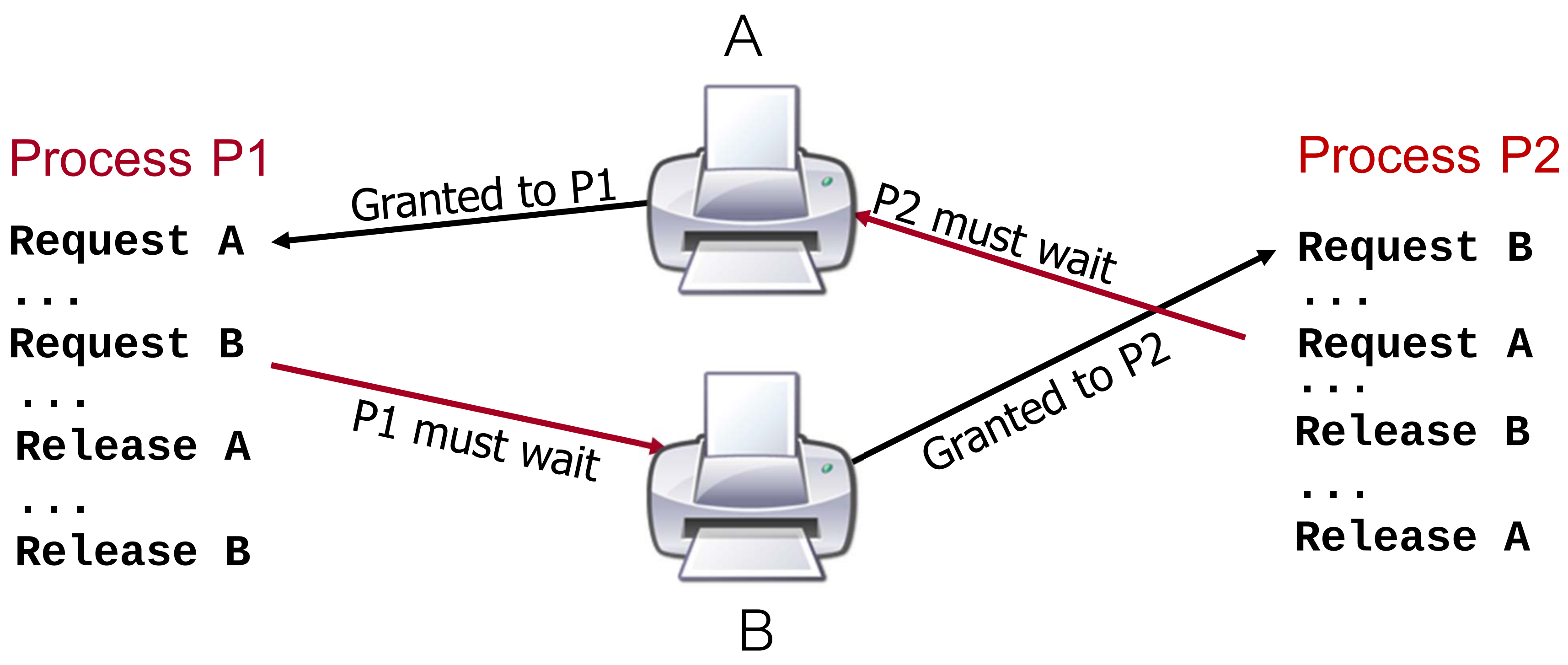
- Situations:
  - Communicate with each other => communication deadlocks
  - Compete for system resources => resource deadlocks
- We will focus on *resource deadlocks.*
- *Root causes:*
  - Resources are *finite*
  - Processes *wait* if a resource they need is unavailable
  - Resources may be *held* by other waiting processes

What do we mean by “Resource”?

- Any object that might be needed by a process to do its work
  - Hardware
    - printers, memory, processors, disk drive
  - Data
    - Shared variables, record in a database, files
  - Synchronization objects (or equivalently, the critical regions they protect)
    - Locks, semaphores, monitors
- We are concerned with *reusable resources*
  - Can be used by one process at a time, released

Example of Deadlock

- Suppose processes *P1* and *P2* request access (locks!) to printers A and B as follows:



Process P1

```
Request A
...
Request B
...
Release A
...
Release B
```

Process P2

```
Request B
...
Request A
...
Release B
...
Release A
```

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
- Circular wait implies hold and wait, but the former results from a sequence of events, while the latter is a policy decision

Deadlock prevention

Mutual Exclusion

Hold and wait

No preemption

Circular wait

Idea: break one => Deadlock cannot occur!

Deadlock prevention

Mutual Exclusion?

Hold and wait

No preemption

Circular wait

Idea: break one => Deadlock cannot occur!

Lock-free data structures

- Maurice Herlihy: use architectural support to create lock-free data structures
- Recall Compare-And-Swap (CAS) instruction
  - `int CAS(int *address, int expected, int new);`
- Blocking vs. non-blocking atomic add (of amount a to value pointed to by val):

```
void AtomicAdd(int *val, int a) {
  lock(L);
  *val += a;
  unlock(L);
}
void AtomicAdd(int *val, int a) {
  do {
    int old = *val;
  } while(CAS(val, old, old + a) == 0);
}
```

No locking => no deadlock can arise!

- Complex operations may not be as trivial to convert to a lock-free version!

Deadlock prevention

Mutual Exclusion

not feasible, or very complex to get it right

Hold and wait?

No preemption

Circular wait

Idea: break one => Deadlock cannot occur!

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
- All or nothing approach: either get all resources from the get go or none of them
- See any problems though?

Deadlock prevention

Mutual Exclusion

not feasible, or very complex to get it right

<u>Hold and wait?</u>

possible efficiency issues, possibly unrealistic assumptions, etc.

No preemption

Circular wait

Idea: break one => Deadlock cannot occur!

Alternative

- Some thread libraries offer `trylock()` function: grab a lock if it's available, otherwise try later.
- For example:

```
ready = false;
while(!ready){
   lock(L1);
   if (trylock(L2) == -1) {
      unlock(L1);
      ready = false;
   }
   else
      ready = true;
}
```

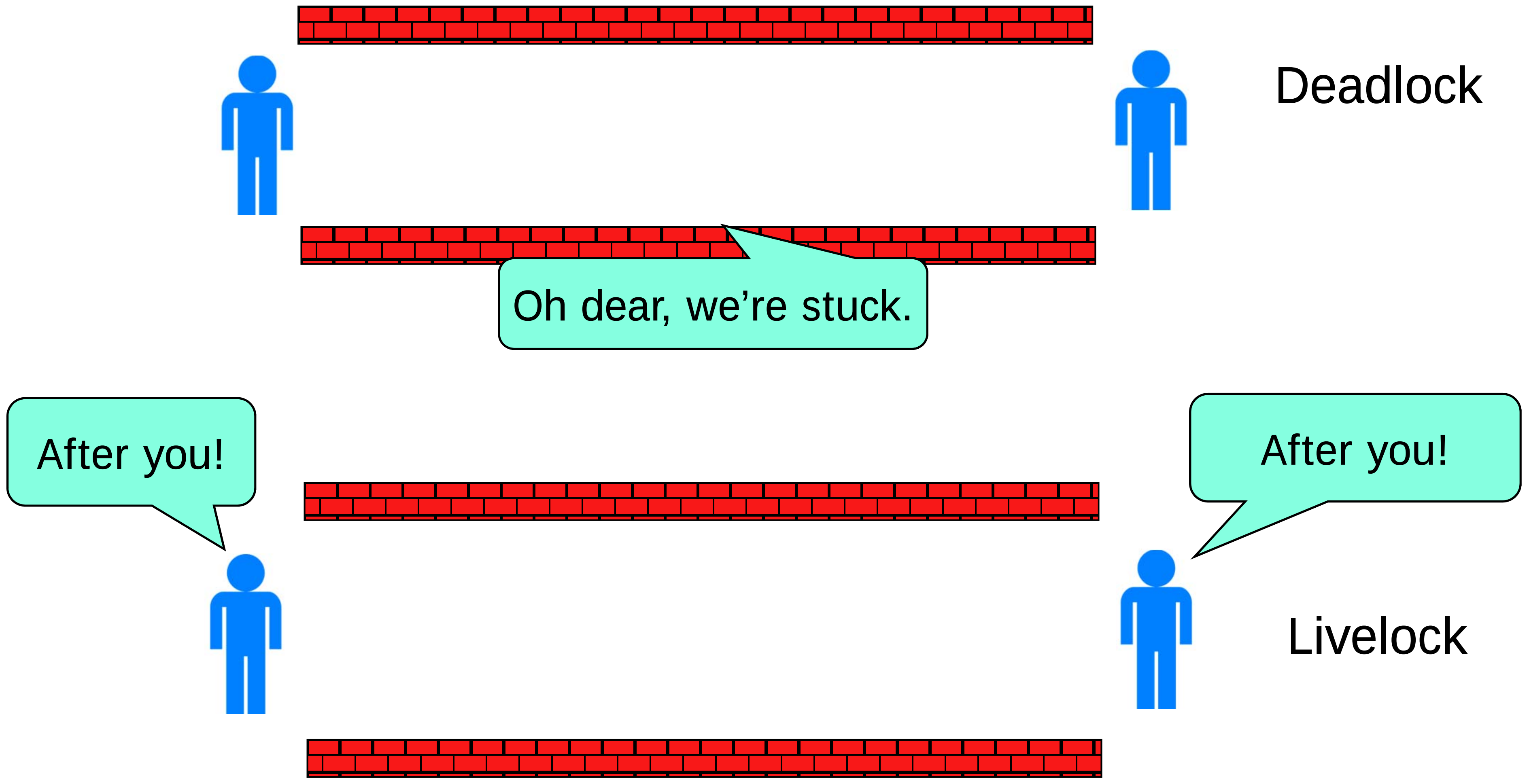
Deadlock avoided!

What could happen here though?

Possible livelock

Solutions to livelock exist too..

Quick reminder: deadlock vs livelock



Deadlock prevention

Mutual Exclusion

not feasible, or very complex to get it right

Hold and wait

possible efficiency issues, possibly unrealistic assumptions, etc.

No preemption?

Circular wait

Idea: break one => Deadlock cannot occur!

Deadlock prevention

Mutual Exclusion

not feasible, or very complex to get it right

Hold and wait

possible efficiency issues, possibly unrealistic assumptions, etc.

No preemption?

not feasible, or highly complex to safely achieve

Circular wait

Idea: break one => Deadlock cannot occur!

Deadlock prevention

Mutual Exclusion

not feasible, or very complex to get it right

Hold and wait

efficiency problems, possibly unrealistic assumptions

No preemption

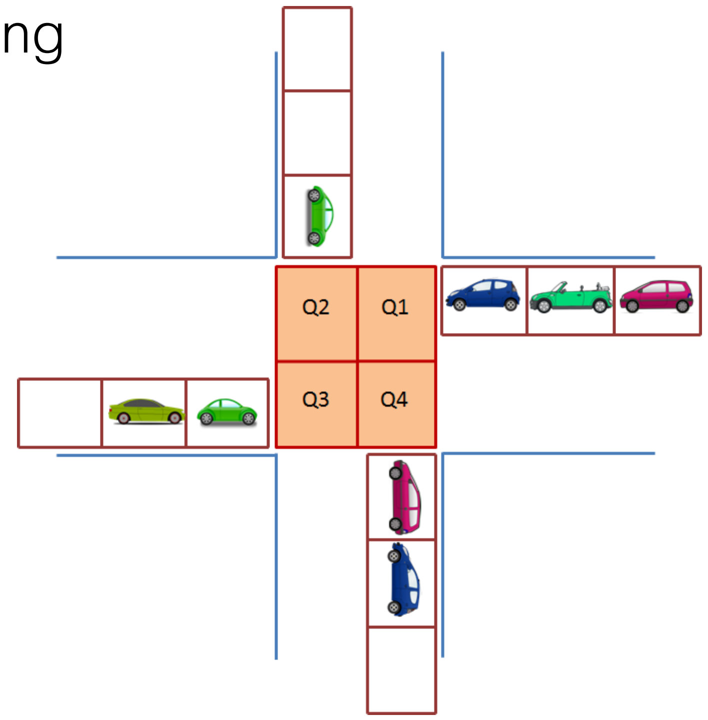
not feasible, or at best problematic

Circular wait?

Idea: break one => Deadlock cannot occur!

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
    - Sounds familiar?



- Can you think of a simple criteria of how to order locks?
- Hard to come up with **total order** when there are lots of resource types
    - **Partial order**, groups of locks with internal ordering, etc.

Deadlock prevention

Mutual Exclusion

not feasible, or very complex to get it right

Hold and wait

efficiency problems, possibly unrealistic assumptions

No preemption

not feasible, or at best problematic

Circular wait

lock ordering – user must take special care of this

Idea: break one => Deadlock cannot occur!

Next up...

- How to deal with deadlocks:
  - Deadlock Prevention
  - **Deadlock Avoidance**
  - Deadlock Detection and Recovery
  - The OS approach

Deadlock Avoidance

- All prevention strategies are unsatisfactory in some situations
- *Avoidance* allows the first three conditions, but ensures that circular wait cannot possibly occur, should a given request be met
  - 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 any future resource allocation "path" leads to deadlock

Processes must declare maximum resource needs up front

Restrictions on Avoidance

- A. Maximum resource requirements for each process must be known in advance
- B. Processes must be independent
  - If order of execution is constrained by synchronization requirements, system is not free to choose a safe sequence
- C. There must be a fixed number of resources to allocate
  - Tough luck if a printer goes offline!

Banker's algorithm

- Due to Dijkstra
- Each thread
  - States its maximum resource requirements
  - Acquires and releases resources incrementally
- Runtime system delays granting some requests to ensure the system never deadlocks
- System can be in one of three states:
  - Safe: for any possible sequence of resource requests, there is at least one safe sequence that eventually succeeds in granting all pending and future requests
  - Unsafe: if all threads request their maximum resources at this point, the system would deadlock (i.e., there is no safe sequence)
  - Deadlocked

Algorithm – Basic idea

- Deadlock avoidance algorithm:
- For every resource request
  - 1. Can the request be granted?
    - If not, request is impossible at this point => block the process until we can grant the request
  - 2. Assume that the request is granted
    - Update state assuming request is granted
  - 3. 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

Example

- Suppose there are a total of 10 resources

	Has	Max
A	3	9
B	2	4
C	2	7

Num free: 3

This is a safe state because there exists a chain of requests which allows all processes to acquire their necessary resources and complete.

Example

- Suppose there are a total of 10 resources (same type) and we transition to a new state

	Has	Max
A	3	9
B	2	4
C	2	7

Num free: 3

Say it goes to either state s1) or s2)

s1)

	Has	Max
A	4	9
B	2	4
C	2	7

Num free: 2

Is this second state safe?

s2)

	Has	Max
A	3	9
B	4	4
C	2	7

Num free: 1

Is this second state safe?

Example: go to state s1

	Has	Max
A	3	9
B	2	4
C	2	7

Num free: 3

⇨

	Has	Max
A	4	9
B	2	4
C	2	7

Num free: 2

B requests 2

⇨

B acquires all 4

	Has	Max
A	4	9
B	4	4
C	2	7

Num free: 0

⇨

B completes

	Has	Max
A	4	9
B	-	-
C	2	7

Num free: 4

If A or C request all of their remaining resources, then deadlocks

Example: go to state s2

	Has	Max
A	3	9
B	2	4
C	2	7

Num free: 3

B requests 2 ⇨

B acquires all 4

	Has	Max
A	3	9
B	4	4
C	2	7

Num free: 1

⇨

B completes

	Has	Max
A	3	9
B	-	-
C	2	7

Num free: 5

C req 5 ⇨

	Has	Max
A	3	9
B	-	-
C	7	7

Num free: 0

Now C completes and then A can also get all of its resources.

Announcements

- A4 extra office hours - TBA
- Please keep checking Piazza as well, or post questions
- Tutorial this week – Deadlocks and Banker's algorithm
- Course evaluations:
  - Important to give feedback!
  - http://www.uoft.me/course-evals

Deadlock Detection & Recovery

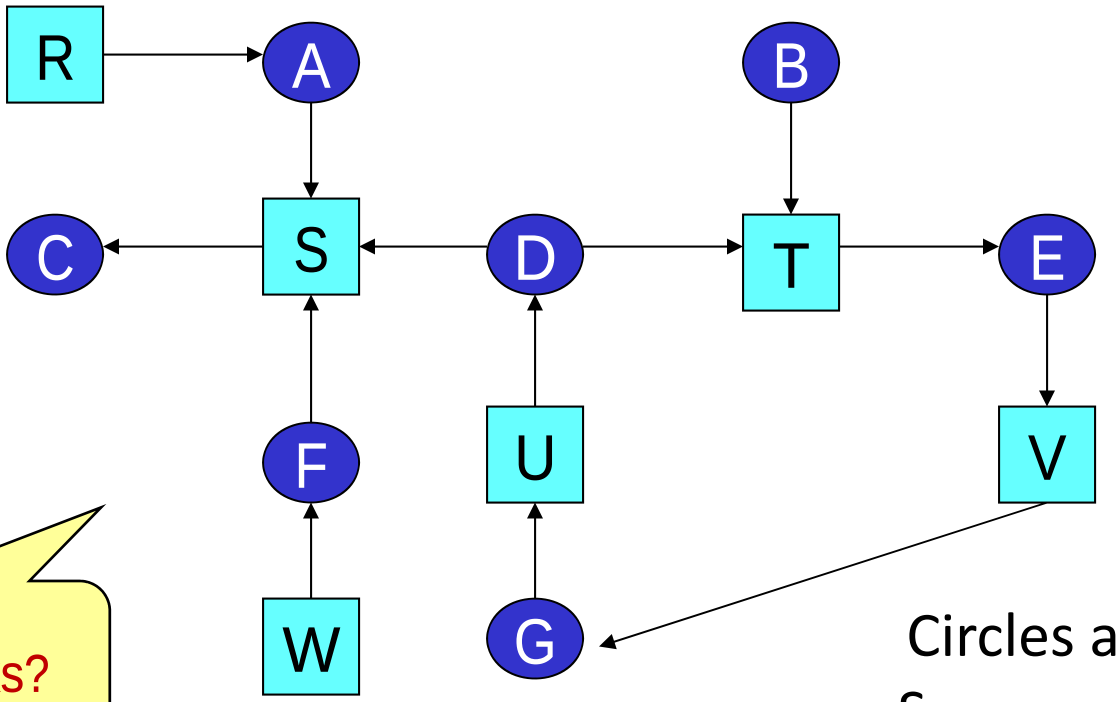
- Prevention and avoidance are awkward and costly
  - Need to be cautious leads to low utilization
- Instead, allow deadlocks to occur, but detect when this happens and find a way to break it
  - Check for circular wait condition periodically
- When should the system check for deadlocks?
  - 1. On every allocation request
  - 2. Fixed periods
  - 3. When system utilization starts to drop below a threshold

Detection continued

- Finding circular waits is equivalent to finding a cycle in the *resource allocation graph*
  - Nodes are processes (drawn as circles in the next slide) 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 for our goal
  - Note: with multiple instances of a type of resource, cycles may exist without deadlock

Example Resource Alloc Graph

Note again: with multiple instances of a type of resource, cycles may exist without deadlock


$$\begin{tikzcd}
R \arrow[r] & A \arrow[d] & & B \arrow[d] & \\
C & S \arrow[l] & D \arrow[l] \arrow[r] & T \arrow[r] & E \arrow[d] \\
& F \arrow[u] & U \arrow[u] & & V \arrow[dlll] \\
& W \arrow[u] & G \arrow[u] & &
\end{tikzcd}$$

Any deadlocks?

Circles are processes, Squares are resources

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 and OS approach

- 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 that it doesn’t happen often

Source: picturespider.com

Why does the “Ostrich Algorithm” Work?

- Recall the 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

Deadlock and Starvation

- a set of threads is in a *deadlocked* state when every process in the set is waiting for a event that can be caused only by another process in the set

$$\neq$$

- 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, resend message and use protocols to detect duplicate messages (why need the latter?)

Transactions and Atomicity

Atomic Transactions

- Recall ATM banking example:
  - Concurrent deposit/withdrawal operation
  - Need to protect shared account balance
- What about transferring funds between accounts?
  - step1. Withdraw funds from account A
  - step2. Deposit funds into account B

Properties of funds transfer

- Should appear as a single operation
  - Another process reading the account balances should see either both updates, or none
- Either both operations complete, or neither does
  - Need to recover from crashes that leave transaction partially complete

Definitions for Transactions

- Defn: Transaction
  - A collection of operations that performs a single logical function
  - We will consider a sequence of **read** and **write** operations, terminated by a **commit** or **abort**
- Defn: Committed
  - A transaction that has completed successfully; once committed, a transaction cannot be undone
- Defn: Aborted
  - A transaction that did not complete normally; typically rollback and start again

Write-ahead logging

- Before performing any operations on the data, write the intended operations to a *log* on stable storage.
  - Sound familiar?
- 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

Checkpoints

- Limitations of basic log strategy:
  - Time-consuming to process <u>entire log</u> after failure
  - <u>Large amount of space</u> required by log
  - <u>Performance penalty</u> – each write requires a log update before the actual data update
- Checkpoints help with first two problems
  - Write all updates to log and data to stable storage; periodically write a *checkpoint* entry to the log
  - Recovery only needs to look at log since last checkpoint

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 T_0 and T_1 ?

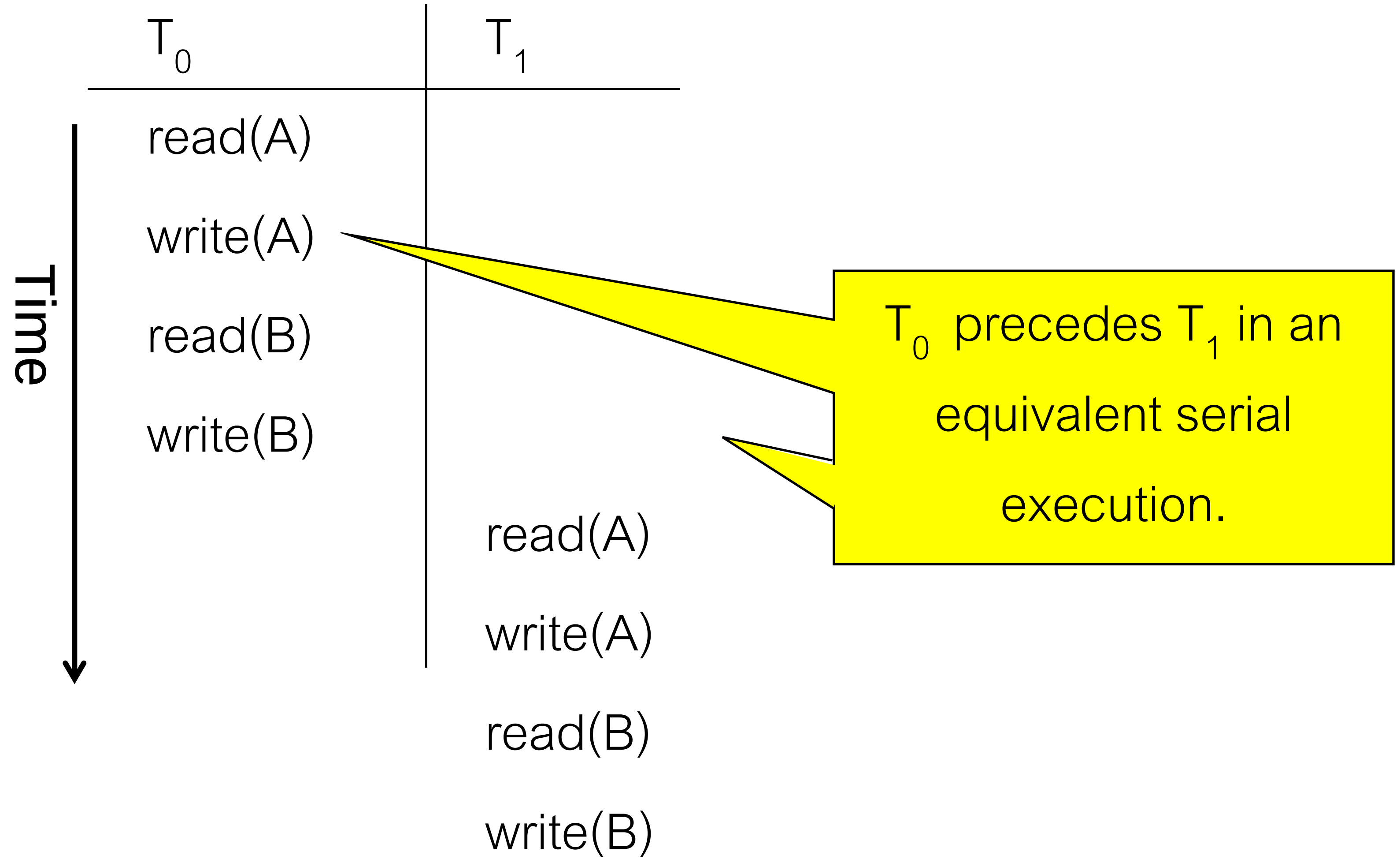
Time

T_0	T_1
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Accesses to different data items do not conflict.

Conflict Serializability

- Swap read/write of B in T_0 with read/write of A in T_1 to get schedule shown below:



T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

T_0 precedes T_1 in an equivalent serial execution.

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. Why?
    - Fix: prevent hold-and-wait by aborting and retrying transaction if any lock is unavailable

Example of 2 phase locking

Transaction_start

$$\left.\begin{array}{l}\text{Lock(A)}\\ \text{Read(A)}\\ \text{Lock(B)}\\ \text{Read(B)}\\ \text{Lock(C)}\end{array}\right\}\text{ Growing – Phase 1}$$

$$\left.\begin{array}{l}\text{Unlock(A)}\\ \text{Unlock(B)}\\ \text{Write(C)}\\ \text{Unlock(C)}\end{array}\right\}\text{ Shrinking – Phase 2}$$

Transaction_end

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 on data X:
  - If transaction has "earlier" timestamp than W-TS on data X, then transaction needs to read a value that was already overwritten
    - Abort transaction, restart with new timestamp
- Writes on data X:
  - 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)

Announcements

- Make sure to check Piazza regularly!
  - Assignment details on how to handle various corner cases
    - You are primarily responsible for reading the ext2 docs and thinking about how to handle various scenarios, then writing solid/robust code!
  - We'll provide clarifications that might help more than just the OP...
  - When in doubt, ask and you will get help, as usual!
- Next time
  - Monday: Course recap, exam review, logistics, etc.
  - Wednesday: exam prep exercises (BA1190 and SS1070)
- Reminder - course evals: http://uoft.me/course-evals
  - Important to give feedback!