Parallel access to linked data structures

[Solihin Ch. 4] Answer the questions below.

Name some linked data structures.

What operations can be performed on *all* of these structures?

Why is it hard to parallelize these operations?

Explain how the following code illustrates such a dependence.

```
void addValue(pIntList pList, int key, int x) {
   pIntListNode p = pList->head;
   while (p != NULL) {
     if (p->key == key)
       S1: p->data = p->data + x;
     S2: p = p->next;
 }
}
```

In the notation introduced in Lecture 9, how would the dependence be written?

If we just look at the loops in an “LDS” program, we won’t find any parallelism to be exploited.

So, where can we find the opportunity to execute anything in parallel?

Conceptually, we can allow several operations to be performed in parallel. What kind of operations?

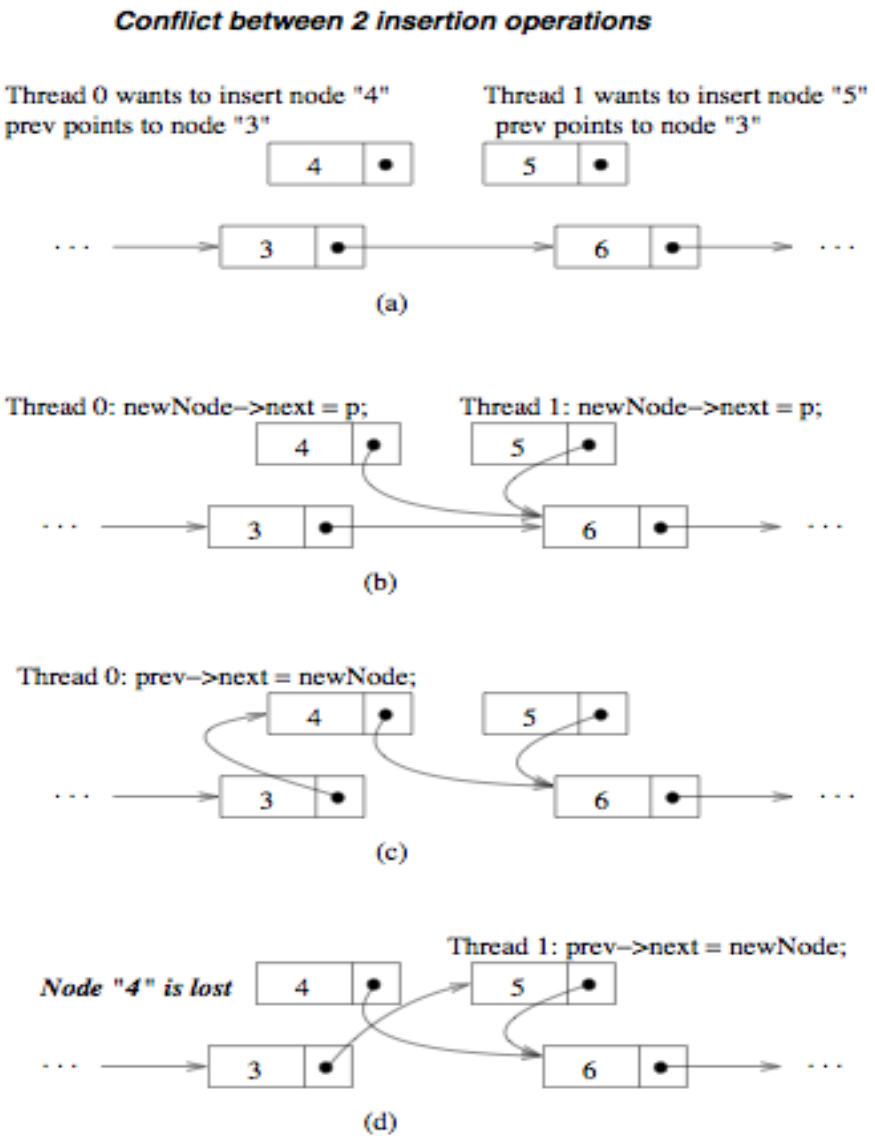
But how do we decide *which* operations can be performed in parallel?

Correctness of parallel LDS operations

Serializability: A parallel execution of a group of operations (or primitives) is said to be *serializable* if there is some sequence of operations (or primitives) that produce an identical result.

Suppose a node insertion i_1 and a node deletion d_1 are performed in parallel. The outcome must be equivalent to either

Conflict between two insertions



Let's look at the simple case of a singly-linked list.

Suppose two items are inserted in parallel: insert both 4 and 5.

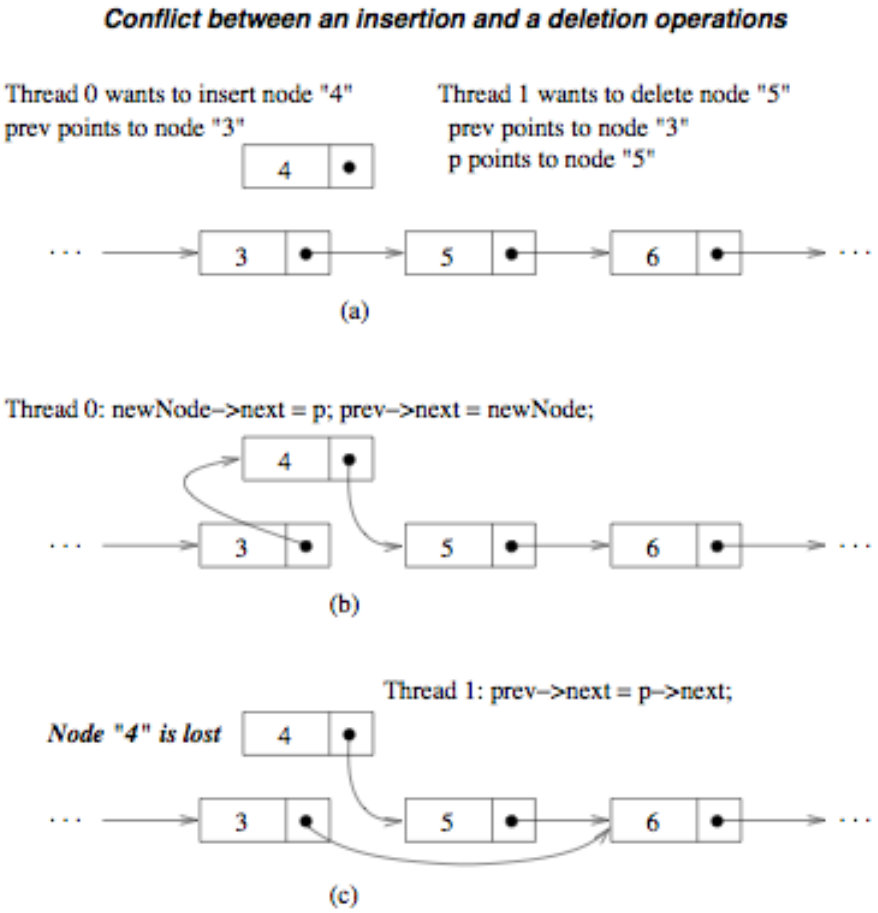
Serializable outcomes:

In any case,

must be in the list at the end of execution

What could happen if the operations are not parallelized correctly?

Conflict between an insertion and a deletion

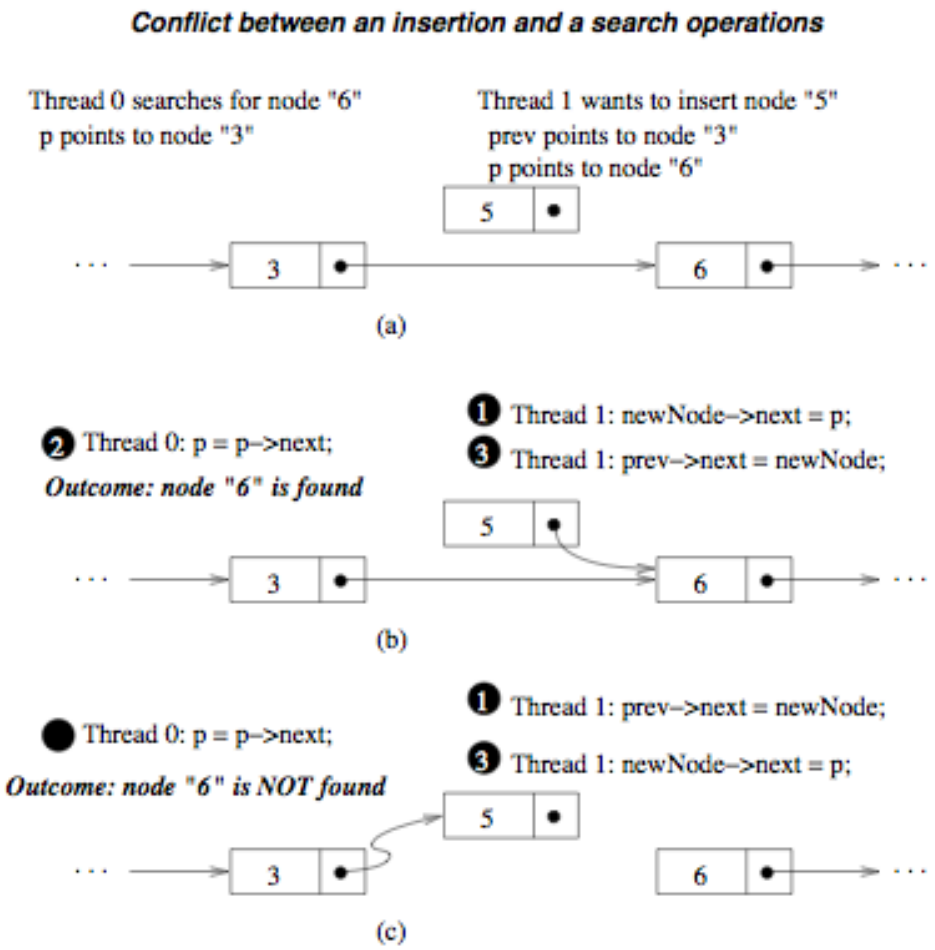


Serializable outcome:

in both cases, at the end of execution, node 4 is in the list, but node 5 is not in the list

In the case shown, node 4 is lost. What would be a sequence that produces *another* incorrect result? What would happen with this sequence? (You may use this worksheet.)

Conflict between an insertion and a search



Suppose we attempt

insert 5, then search 6

or search 6, then insert 5

in both cases, at the end of execution,

- 5 must be in the list, and
- 6 must be found

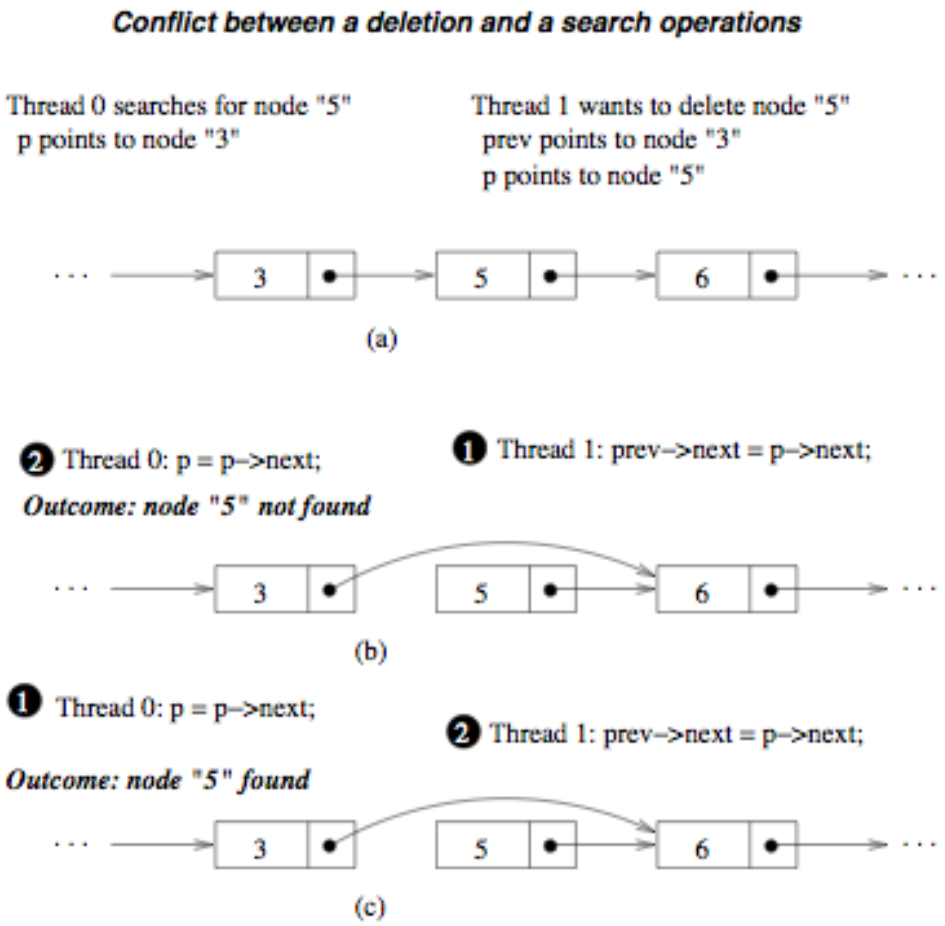
Depending on when the insertion code is executed,

- node 6 will be found, or
- node 6 may not be found, and an uninitialized link may be followed.

Conflict between a deletion and a search

- Deletion and search
  - delete 5, then search for 5
  - search for 5, then delete 5
- Possible outcomes
  - Node 5 may be found or not found
  - Node 5 is deleted from the list

What, if anything, is the problem with these outcomes?



Main Observations

- Parallel execution of two operations that affect a *common node*, in which at least one operation *involves writing* to the node, can produce conflicts that lead to *non-serializable outcome*.
- Under some circumstances, a serializable outcome may still be achieved, despite the conflicts mentioned above.
- Conflicts can also occur between LDS operations and memory-management functions such as allocation and deallocation.

Parallelization strategies

- Parallelization among readers
  - Very simple
  - Works well if structure is modified infrequently

- Global lock approach
  - Relatively simple
  - Parallel traversal, followed by sequential list modifications
- Fine-grain lock approach
  - A lock is associated with each node.
  - Each operation locks only nodes that need to be accessed exclusively.
  - Complex: Deadlock can occur; memory allocation and deallocation become more complex

Parallelization among readers

- Basic idea
  - (Read-only) operations that do not modify the list can execute in parallel.
  - (Write) operations that modify the list execute sequentially
- How to enforce
  - A read-only operation acquires a read lock
  - A write operation acquires a write lock
- Construct a lock-compatibility table

Already-granted lock	Read lock requested	Write lock requested
Read lock	Yes	No
Write lock	No	No

Example

```
IntListNode_Search(int x)
{
  acq_read_lock();
  …
  …
  …
  rel_read_lock();
}
```

```
IntListNode_Insert(node *p)
{
  acq_write_lock();
 …
 …
 …
  rel_write_lock();
}
```

Global-lock approach

- Each operation logically has two steps
  - Traversal
    - Node insertion: Find the correct location for the node
    - Node deletion: Find the node to delete
    - Node search: Find the sought-for node
  - List modification
- Basic idea: perform the traversal in parallel, but modify the list in a critical section,

- Pitfall
  - The list may have changed by the time the write-lock is acquired,
  - so the assumptions must be re-validated.

Example

```
IntListNode_Insert(node *p)
{
  …
  /* perform traversal */
  …
  acq_write_lock();
  /* then check validity:
       nodes still there?
       link still valid? */
  /* if not valid, repeat traversal */
  /* if valid, modify list */
  …
  rel_write_lock();
}
```

Fine-grain locking approach

- Associate each node with a lock (read, write).
- Each operation locks only needed nodes.
- (Read and write) operations execute in parallel except when they conflict on some nodes. Fill in the blanks below.
  - Nodes that will be modified are _____ .

- Nodes that are read and must remain unchanged are _____ .

- Pitfall: Deadlock becomes possible.
  - Suppose one operation locks node 1 and then needs to lock node 2, while another operation locks node 2 and then needs to lock node 1.
  - Then neither operation can complete before the other operation frees the lock it is holding.
- Deadlocks can be prevented by imposing a _____ _____.

Example

```
void insert(pIntList pList, int x){
  int succeed;
  ... /* traversal code to find where to insert */

  /* insert the node at head or between prev & p */
  succeed = 0;
  do {
    acq_write_lock(prev);
    acq_read_lock(p);
    if (prev->next != p || prev->deleted || p->deleted)
    {
      rel_write_lock(prev);
      rel_read_lock(p);
      ... /* repeat traversal */
    }
    else
      succeed = 1;
  } while (!succeed);

  /* prev and p are now valid, so insert node */
  newNode->next = p;
  if (prev != NULL)
    prev->next = newNode;
  else
    pList->head = newNode;
  rel_write_lock(prev);
  rel_read_lock(p);
```

```
}
```

Questions

What do the tests **prev->deleted** and **p->deleted** mean?

Why is garbage collection used, rather than explicit deletion?

The delete operation is similar; code that is the same is shown in green.

```
void delete(pIntList pList, int x){
  int succeed;
  … /* traversal code to find node to delete */

  /* node has been found; perform the deletion */
  succeed = 0;
  do {
    acq_write_lock(prev);
    acq_write_lock(p);
    if (prev->next != p || prev->deleted || p->deleted)
    {
      rel_write_lock(prev);
      rel_write_lock(p);
      … /* repeat traversal; return if not found */
    }
    else
      succeed = 1;
  } while (!succeed);

  /* prev and p are now valid, so delete node */
  if (prev == NULL) { /* delete head node */
    acq_write_lock(pList);
    pList->head = p->next;
    rel_write_lock(pList);
  }
  else /* delete non-head node */
    prev->next = p->next;
```

```
  p->deleted = 1; /*don't deallocate; mark deleted*/
  rel_write_lock(prev);
  rel_write_lock(p);
}
-
```