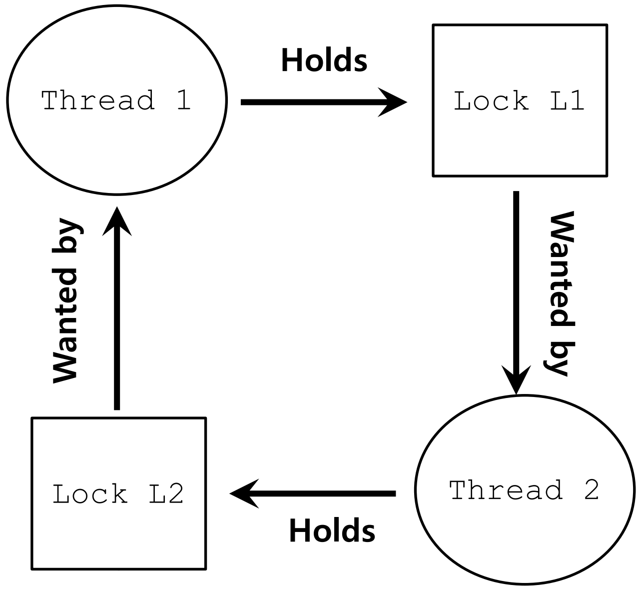
# Operating Systems

## Deadlock

# Deadlock Bugs

```
Thread 1:          Thread 2:
lock(L1);          lock(L2);
lock(L2);          lock(L1);
```

- The presence of a cycle
  - `Thread1` is holding a lock `L1` and waiting for another one, `L2`.
  - `Thread2` that holds lock `L2` is waiting for `L1` to be released.



# Why Do Deadlocks Occur?

- Reason 1:
  - In large code bases, complex dependencies arise between components.
- Reason 2:
  - Due to the nature of encapsulation
    - Hide details of implementations and make software easier to build in a modular way.
    - Such modularity *does not mesh* well with <u>locking</u>.

# Conditions for Deadlock

- <u>Four conditions</u> need to hold for a deadlock to occur.

| Condition | Description |
|---|---|
| Mutual Exclusion | Threads claim exclusive control of resources that they require. |
| Hold-and-wait | Threads hold resources allocated to them while waiting for additional resources |
| No preemption | Resources cannot be forcibly removed from threads that are holding them. |
| Circular wait | There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain |

  - If any of these four conditions are not met, **deadlock cannot occur**.

# Solutions to Deadlock

- Prevention
- Avoidance
- Detect and Recover

# Prevention – Circular Wait

- Provide a total ordering on lock acquisition
  - This approach requires *careful design* of global locking strategies.
- **Example**:
  - There are two locks in the system (L1 and L2)
  - We can prevent deadlock by always acquiring L1 before L2.

# Prevention – Hold-and-wait

- Acquire all locks at once, atomically.

```
lock(prevention);
lock(L1);
lock(L2);
…
unlock(prevention);
```

  - This code guarantees that **no untimely thread switch can occur** *in the midst of* lock acquisition.
  - **Problem**:
    - Require us to know when calling a routine exactly which locks must be held and to acquire them ahead of time.
    - Decrease *concurrency* as all locks must be acquired early on (at once) instead of when they are truly needed.

# Prevention – No Preemption

- **Multiple lock acquisition** often gets us into trouble because when waiting for one lock we are holding another.
- `trylock()`
  - Used to build a *deadlock-free*, *ordering-robust* lock acquisition protocol.
  - Grab the lock (if it is available).
  - Or, return -1: you should try again later.

```
top:
        lock(L1);
        if( tryLock(L2) == -1 ){
                unlock(L1);
                goto top;
        }
```

# Prevention – No Preemption (Cont.)

- livelock
  - Both systems are running through the code sequence *over and over again*.
  - Progress is not being made.
  - Solution:
    - Add **a random delay** before looping back and trying the entire thing over again.

# Prevention – Mutual Exclusion

- wait-free
  - Using powerful hardware instruction.
  - You can build data structures in a manner that *does not require* explicit locking.

```
int CompareAndSwap(int *address, int expected, int new){
    if(*address == expected){
        *address = new;
        return 1; // success
    }
    return 0;
}
```

# Prevention – Mutual Exclusion (Cont.)

## Example: list insertion

- Surrounding this code with a lock acquire and release.

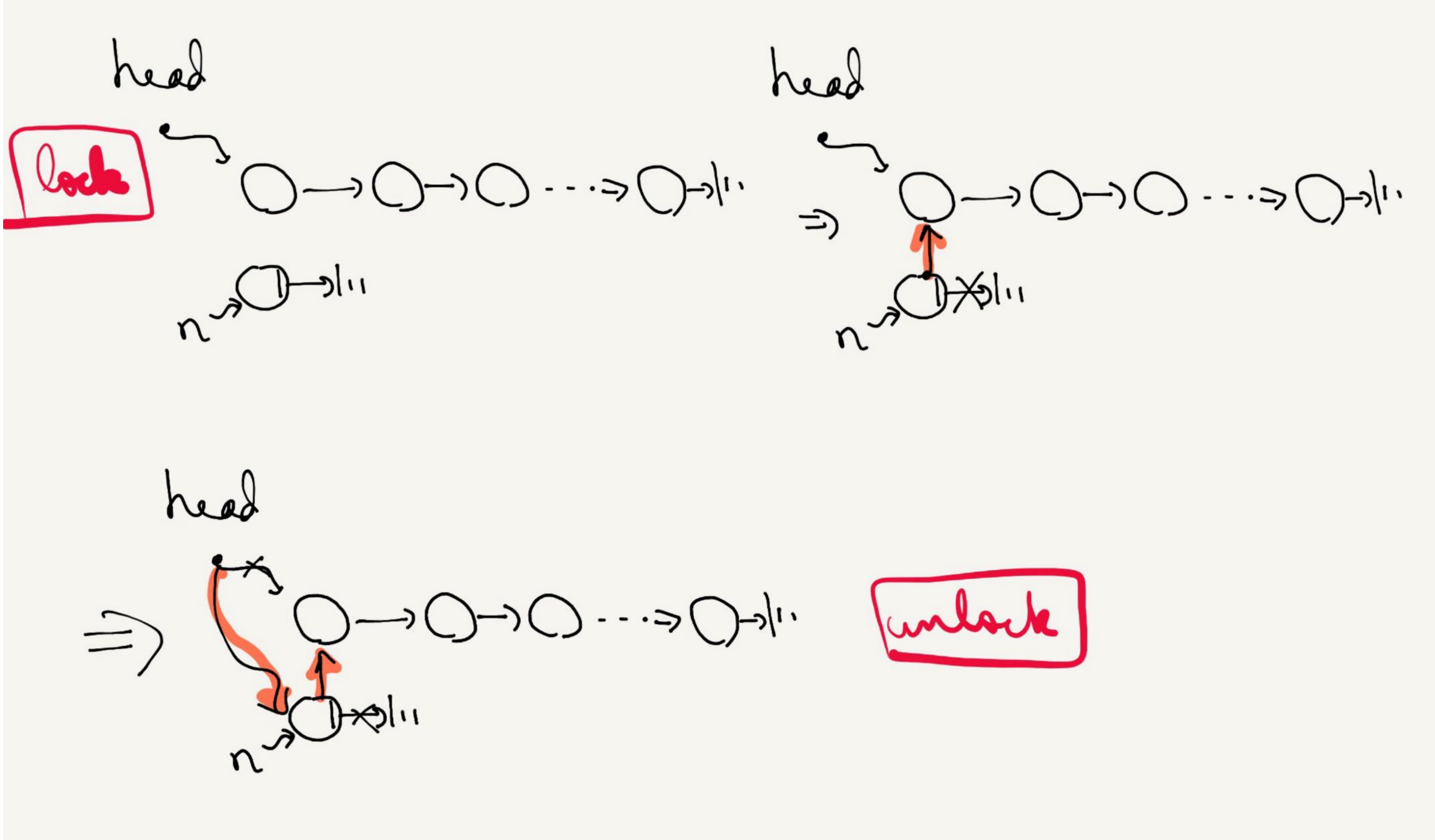
```
void insert(int value){
      node_t * n = malloc(sizeof(node_t));
      assert( n != NULL );
      n->value = value ;
      lock(listlock); // begin critical section
      n->next   = head;
      head      = n;
      unlock(listlock) ;  //end critical section
}
```

- wait-free manner using the `compare-and-swap` instruction

```
int CompareAndSwap(int *address, int expected, int new){
      if(*address == expected){
            *address = new;
            return 1; // success
      }
      return 0;
}
```

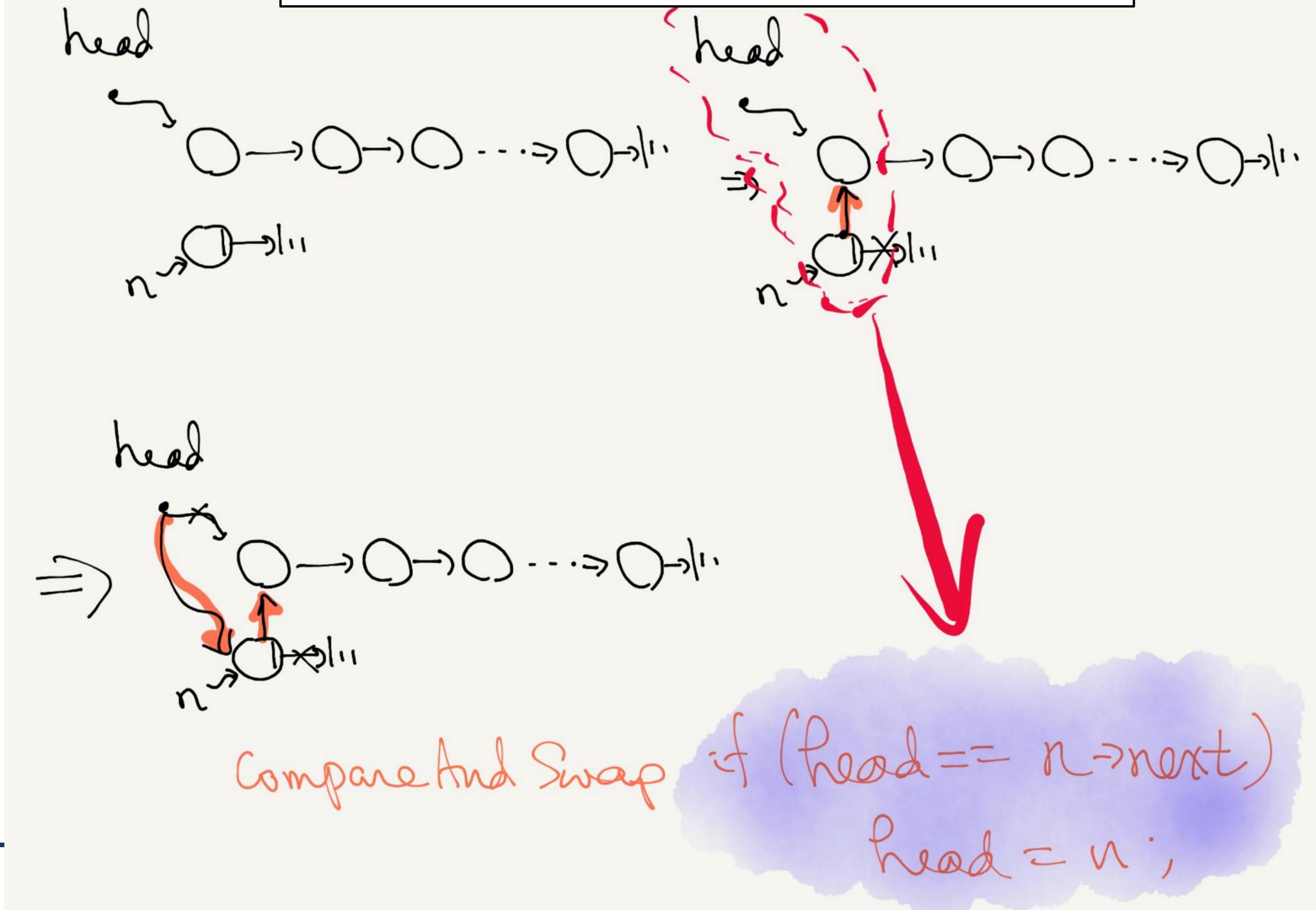
```
void insert(int value) {
      node_t *n = malloc(sizeof(node_t));
      assert(n != NULL);
      n->value = value;
      do {
            n->next = head;
      } while (CompareAndSwap(&head, n->next, n));
}
```

# Insert with lock



# Lock-free insert

```c
void insert(int value) {
        node_t *n = malloc(sizeof(node_t));
        assert(n != NULL);
        n->value = value;
        do {
                n->next = head;
        } while (CompareAndSwap(&head, n->next, n));
}
```



# Deadlock Avoidance via Scheduling

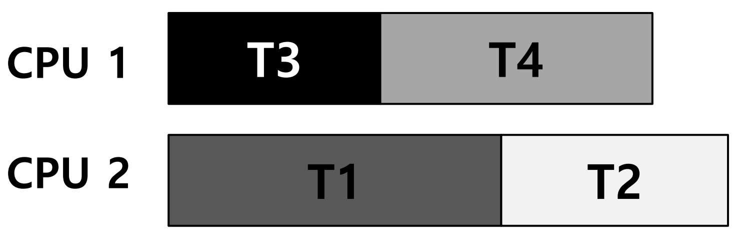
- Deadlock Avoidance
  - Get the information about the locks various threads might grab during their execution.
  - schedule the threads in a way <u>to guarantee</u> no deadlock can occur.
- In some scenarios, deadlock avoidance is preferable.
- Problem: Global knowledge is required.

# Example of Deadlock Avoidance via Scheduling (1)

- We have two processors and four threads.
  - Lock acquisition demands of the threads:

| | T1 | T2 | T3 | T4 |
|---|---|---|---|---|
| L1 | yes | yes | no | no |
| L2 | yes | yes | yes | no |

  - A smart scheduler could compute that as long as <u>T1 and T2 are not run at the same time</u>, no deadlock could ever arise.

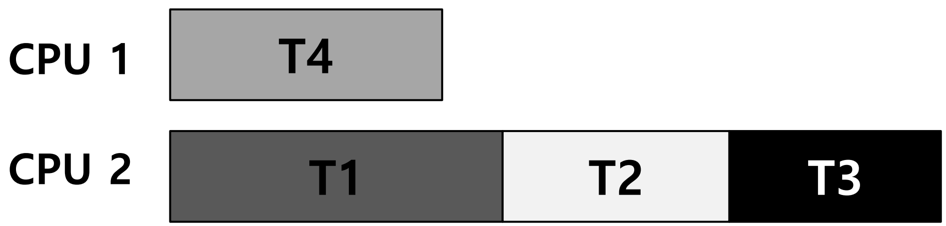


# Example of Deadlock Avoidance via Scheduling (2)

- More contention for the same resources

| | T1 | T2 | T3 | T4 |
|---|---|---|---|---|
| L1 | yes | yes | yes | no |
| L2 | yes | yes | yes | no |

  - A possible schedule that guarantees that *no deadlock* could ever occur.




    - The total time to complete the jobs is lengthened considerably.

# Detect and Recover

- Allow deadlock to occasionally occur and then *take some action*.
  - **Example**: if an OS froze, you would reboot it.
- View system as graph
  - Processes and Resources are nodes
  - Resource Requests and Assignments are edges
- If graph has no cycles $\longrightarrow$ no deadlock
- If graph contains a cycle
  - Definitely deadlock if only one instance per resource
  - Otherwise, maybe deadlock, maybe no
- Many database systems employ *deadlock detection* and *recovery technique*.
  - A deadlock detector **runs periodically**.
  - Building a **resource graph** and checking it for cycles.
  - In deadlock, the system **need to be restarted**.