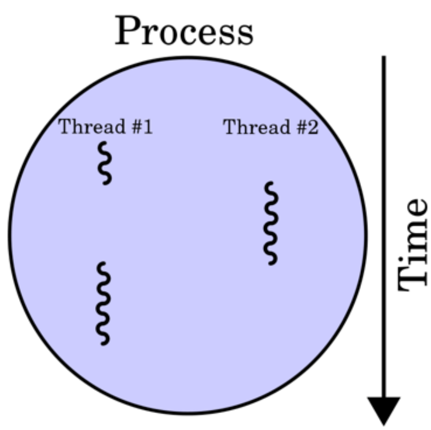
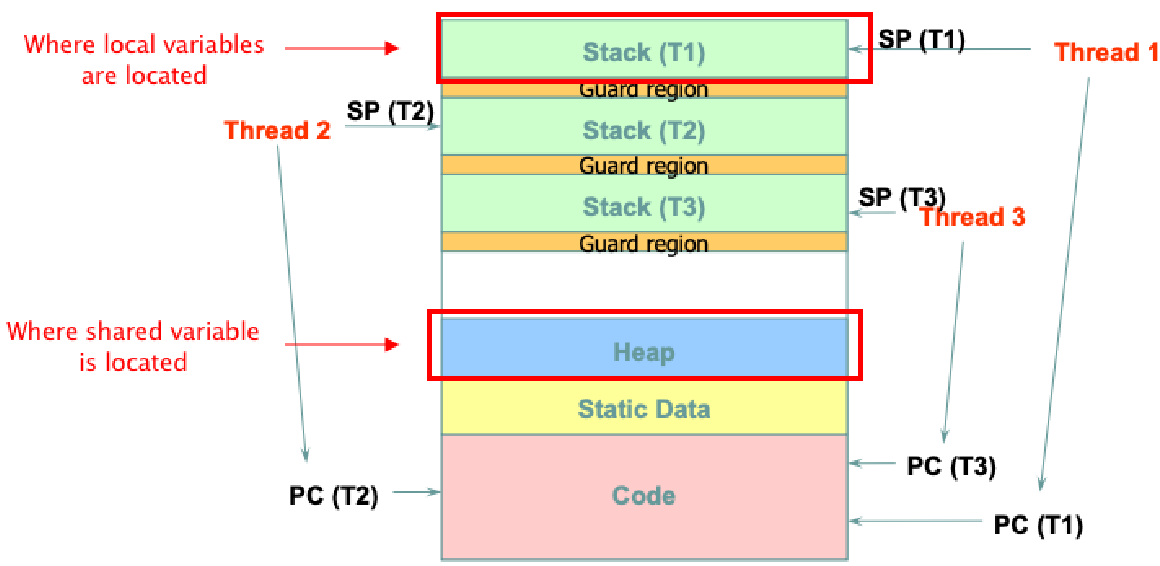
# 1 Critical Section

- Is a piece of code that accesses a *shared* resource, usually a variable or data structure

# 2 Thread

- Is a lightweight process that can be managed independently by a schdeduler [4]
- Improves the application performance using parallelism. (e.g peach)




- A thread is bound to a single process
- A process can have multiple threads
- Has two types
  - **User-level Threads:**

    * Are implemented by users and kernel is not aware of the existence of these threads
    * Are represented by a program counter(PC), stack, registers and a small process control block
    * Are small and much faster than kernel level threads
  - **Kernel-level Threads:**
    * Are handled by the operating system directly
    * Thread management is done by the kernel
    * Are slower than user-level threads

# 3 Thread API

- `pthread_create`
  - **syntax:**

```c
int pthread_create(pthread_t *thread,
                   const pthread_attr_t *attr,
                   void * (*start_routine)(void*),
                   void * arg)
```

    * `thread`
      · is a pointer to a structure of type `pthread_t`
    * `attr`
      · is used to specify any attributes this thread might have
      · is initialized with a separate call `pthread_attr_init()`
      · set default by passing `NULL`
    * `(start_routine)`
      · means which function this thread should start running in?
      · setting void pointer `(void *)` as an argument to function `start_routine` allows us to pass in any type of argument
      · setting void pointer `(void *)` as return type allows us to return any type of result
    * `args`
      · is where to pass the arguments for the function pointer ((start_routine))

    **Example**

```c
#include <stdio.h>
#include <pthread.h>

typedef struct {
    int a;
    int b;
} myarg_t;

void *mythread(void *arg) {
    myarg_t *args = (myarg_t *) arg;
    printf("%d %d\n", args->a, args->b);
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t p;
    myarg_t args = { 10, 20 };

    int rc = pthread_create(&p, NULL, mythread, &args);
    ...
}
```

Struct is copied here

Argument is initialized here

- `pthread_cond_wait`
  - Puts the calling thread to sleep (a blocked state)
  - Waits for some other thread to signal it

**Example**

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;

Pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

Puts calling thread cond to sleep

- `pthread_cond_signal`
  - Is used to unblocks at least one of the threads that are blocked on the specified condition variable cond

**Example**

```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

Wakes a thread that's been put to sleep on cond variable

# 4 Condition Variable

- is an explicit queue that threads can put themselves on when some state of execution is not as desired (so it can be put to sleep)
- when states are changed, one or more of the waiting threads can be awaken and be allowed to continue (done by **signaling** the condition)
- queue is **FIFO**
- `wait()` call is used to put thread to sleep
- `singal()` call is used to awake thread from sleep
- **Syntax (initialization):**

  ```
  pthread_cont_t c = PTHREAD_COND_INITIALIZER
  ```

  **Example**

```c
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Initialized here

- **Syntax (Wait):**

  `Pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m)`

  <u>**Example**</u>

```c
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Put thread to sleep until done

- **Syntax (Signal):**

  `Pthread_cond_signal(pthread_cond_t *c)`

  **Example**

```
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Awake thread here

# 5 Spinlock

- Is the simplest lock to build
- Uses a lock variable
  - 0 - (available/unlock/free)
  - 1 - (acquired/locked/held)
- Has two operations
  1. `acquire()`

```
boolean test_and_set(boolean *lock)
{
      boolean old = *lock;
      *lock = True;
      return old;
}
boolean lock;

void acquire(boolean *lock) {
       while(test_and_set(lock));
}
```

2. `release()`

```
void release(boolean *lock) {
       *lock = false;
}
```

- Allows a single thread to enter critical section at a time
- Spins using CPU cycles until the lock becomes available.
- May spin forever

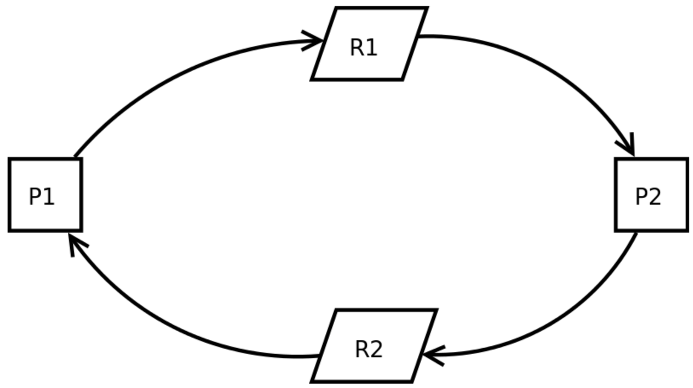
# 6 Livelock

- Two or more threads reapeatedly attempting this code over and over (e.g acquiring lock), but progress is not being made (e.g acquiring lock)
- Solution: Add a random delay before trying again (decrease odd of livelock)

# 7 Mutual Exclusion

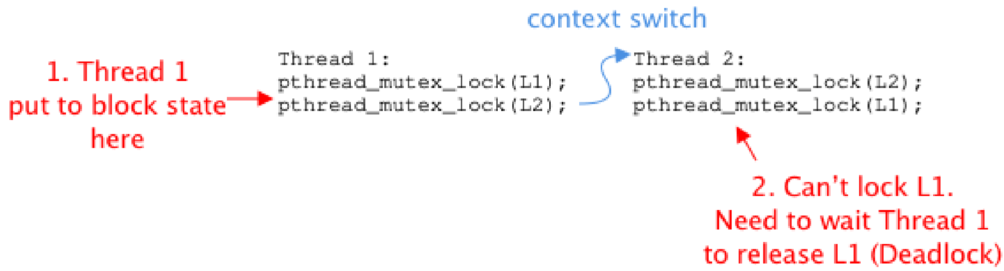
- Is a guarentee that is one thread is executing within the critical section, the others will be prevented from doing so.

# 8 Deadlock




- Is a state in which each member of a group is waiting for another member including itself, to take action (e.g. releasing lock)
- Conditions for Deadlock (All four must be met)
  - **Mutual Exclusion**
    * Occurs when threads claim exclusive control of resources that they require (e.g. thread grabing a lock)
  - **Hold-and-wait**
    * Occurs when threads hold resources allocated to them (e.g locks that they have already acquired) while waiting for additional resources (e.g. locks that they wish to acquire)
  - **No Preemption**
    * Occurs when resource cannot be forcibly removed from threads that are holding them
  - **Circular Wait**
    * Occurs when there exists a circular chain of threads such that each threads hold one or more resources (e.g. locks) that are being requested by the next thread in the chain.

**Example**




- Preventions
  - **Circular Wait**
    - Write code such that circular wait is never induced
    - Is the most practical prevention technique
    - Requires deep understanding of the code base
    - **Total Ordering** (Most starightforward)

      **<u>Example</u>**

      Given two locks in the system (`L1` and `L2`), always acuiqure `L1` before `L2`

    - **Partial Ordering** (Applied to complex systems)

      **<u>Example</u>**

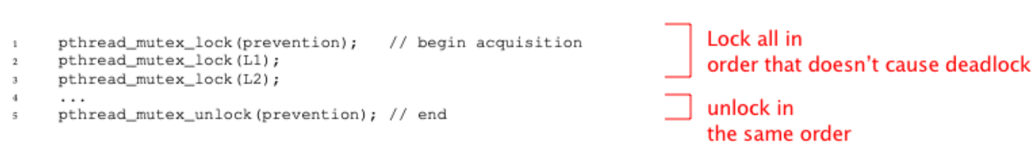
      Memory mapping code in Linux (has then different groups).

      (Simple) `i_mutex` before `i_mmap_mutex`

      (More complex) `i_mmap_mutex` before `private_lock` before `swap_lock` before `mapping->tree_lock`

  - **Hold-and-wait**
    - Can be avoided by acquiring all locks at once
    - Can be problematic
    - Must know which lock must be held and acquire ahead of time
    - Is likely to decrease concurrency (since all need to be acquired over their needs)

      **<u>Example</u>**



- **No Preemption**
  - Can be avoided by adding code that force unlock if not available

    Thread 1

    ```
    top:
      pthread_mutex_lock(L1);
      if (pthread_mutex_trylock(L2) != 0) {
        pthread_mutex_unlock(L1);
        goto top;
      }
    ```

    1. Check if L2 is locked or not available

    2. Unlock L1 forcibly (to avoid deadlock)

    Thread 2

    ```
    top:
      pthread_mutex_lock(L2);
      if (pthread_mutex_trylock(L1) != 0) {
        pthread_mutex_unlock(L2);
        goto top;
      }
    ```

  - `pthread_mutex_trylock` tries to lock the speicied mutex.
  - `pthread_mutex_trylock` returns 0 if lock is available
  - `pthread_mutex_trylock` returns the following error if occupied

    `EBUSY` - Mutex is already locked

    `EINVAL` - Is not initialized mutex

    `EFAULT` - Is in valid pointer
  - May result in **live lock**
- **Mutual Exclusion**
  - Idea: Avoid the mutual exclusion at all
  - Use **lock-free/wait-free** approach: building data structures in a manner that does not require explicit locking using hardware instructions

    **<u>Example</u>**

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

- Avoidance
  - **Banker's Algorithm**

# 9 Process

- Is a program in execution
- Is named by it's process ID or PID
- Can be described by the following states at any point in time
  - Address Space
  - CPU Registers
  - Program Counter
  - Stack Pointer
  - I/O Information

  (wait. this is PCB)
- Exists in one of many different **process states**, including
  1. Running
  2. Ready to Run
  3. Blocked

  - Different events (Getting Scheduled, descheduled, or waiting for I/O) transitions one of these states to the other

# 10 Coarse-grained-locking

- Is one big lock that is used any time any critical section is accessed
- Is easy to write
- Is easy to prove correctness
- No fault-tolerance but deadlock-free
- Perfoms poorly when contention (the need for performance due to load) is high
  - No concurrent access

  **Example**

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;

Pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

One lock for all threads (coarse-grained lock)

Notice that only one thread can pass at a time

# 11 Fine-grained-locking

- Uses different locks to often protect different data and data strutures
- Allows more threads to be in locked code at once

<u>**Example**</u>

```
void List_Init(list_t *L) {
    L->head = NULL;
    pthread_mutex_init(&L->lock, NULL);
}

void List_Insert(list_t *L, int key) {
    // synchronization not needed
    node_t *new = malloc(sizeof(node_t));
    if (new == NULL) {
        perror("malloc");
        return;
    }
    new->key = key;

    // just lock critical section
    pthread_mutex_lock(&L->lock);
    new->next = L->head;
    L->head   = new;
    pthread_mutex_unlock(&L->lock);
}

int List_Lookup(list_t *L, int key) {
    int rv = -1;
    pthread_mutex_lock(&L->lock);
    node_t *curr = L->head;
    while (curr) {
        if (curr->key == key) {
            rv = 0;
            break;
        }
        curr = curr->next;
    }
    pthread_mutex_unlock(&L->lock);
    return rv; // now both success and failure
}
```

Fine-grained lock here :)

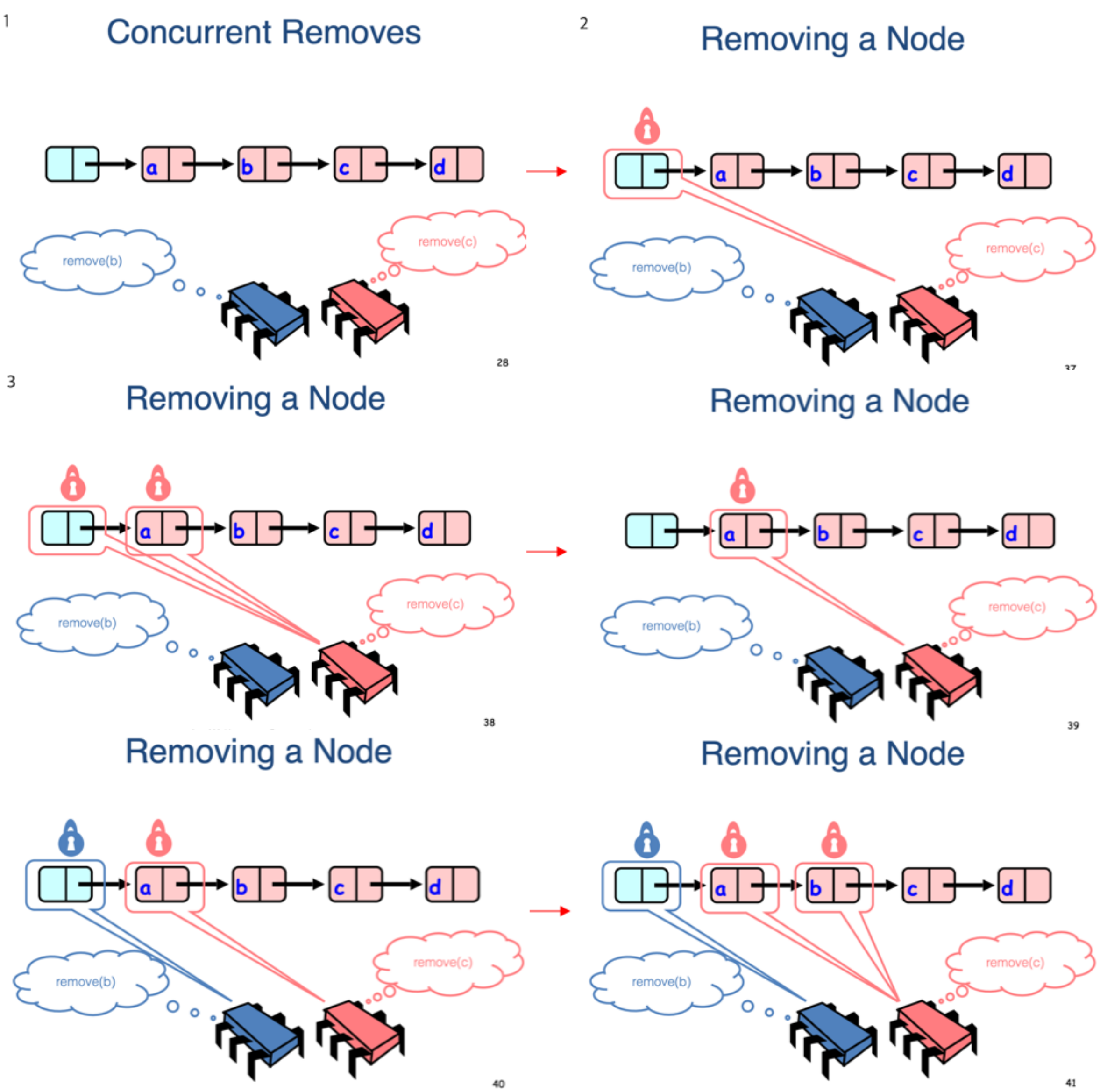
Notice lock is on a struct L

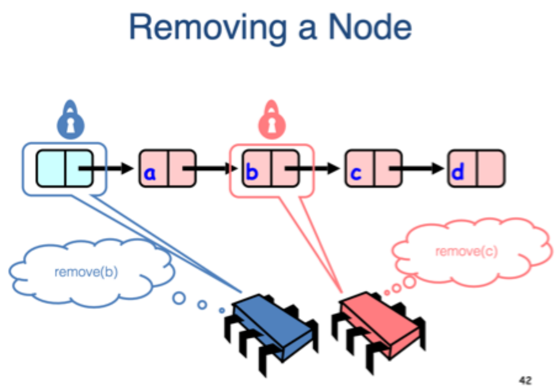
More threads can be locked at once

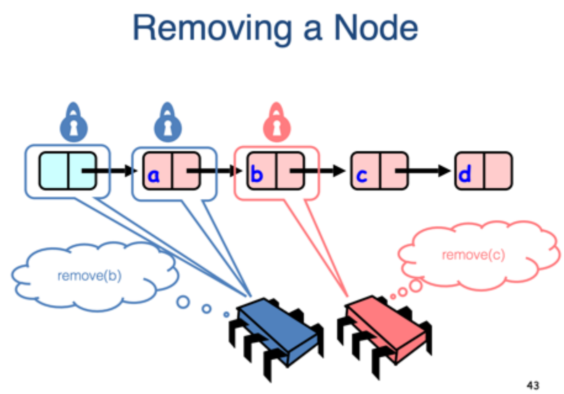
# 12 Hand-over-hand locking

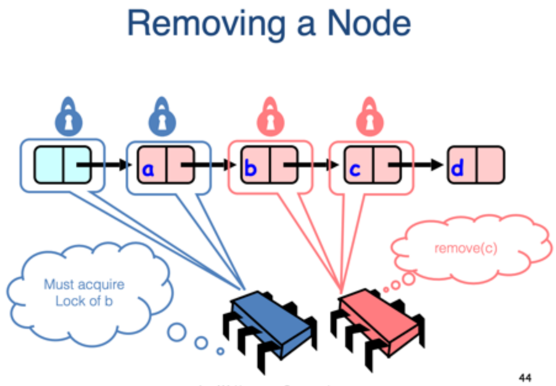
- Idea: instead of having a single lock for the entire list, a lock per node of the list is added; when traversing the list, the list grabs the next node's lock, and releases the current node's lock
- Is a fine-grained-locking
- Holds at most 2 locks at a time

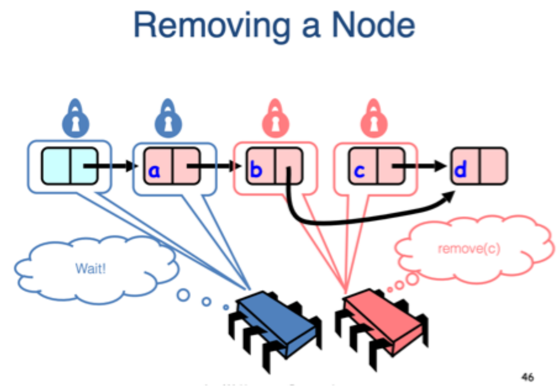
**Example**

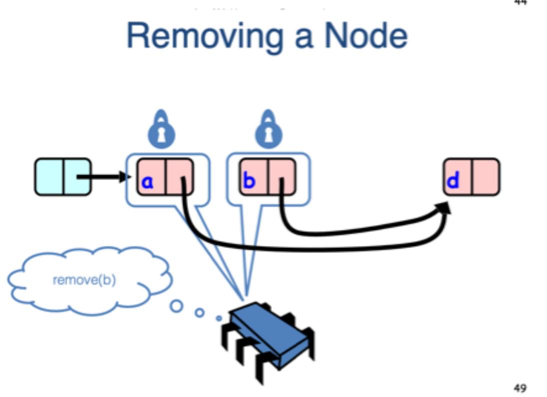


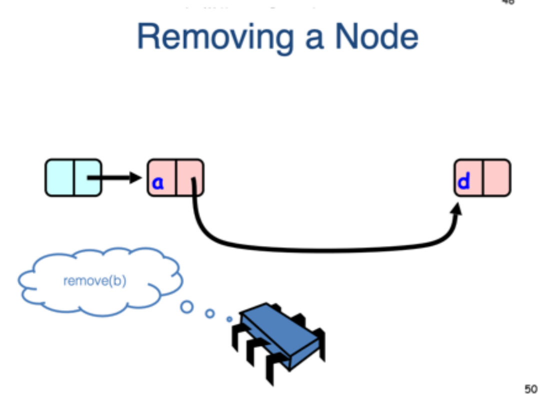

Removing a Node
a
b
c
d
remove(b)
remove(c)
42

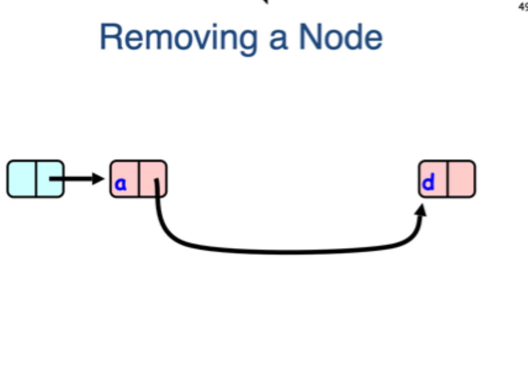

Removing a Node
a
b
c
d
remove(b)
remove(c)
43


Removing a Node
a
b
c
d
Must acquire Lock of b
remove(c)
44


Removing a Node
a
b
c
d
Wait!
remove(c)
46


Removing a Node
a
b
d
remove(b)
49


Removing a Node
a
d
remove(b)
50


Removing a Node
a
d
51