

1. a) Yes, they are part of system call's Application Programming Interface, and they are the only way to interact between computer program and OS kernel.

### Notes

- **System Calls**

- Is issued by a client
- Is the only entry points into the kernel system
- Provides services via API or Application Program Interface
- Has five different types of calls

Types of System Calls	Windows	Linux
Process Control	CreateProcess() ExitProcess() WaitForSingleObject()	fork() exit() wait()
File Management	CreateFile() ReadFile() WriteFile() CloseHandle()	open() read() write() close()
Device Management	SetConsoleMode() ReadConsole() WriteConsole()	ioctl() read() write()
Information Maintenance	GetCurrentProcessID() SetTimer() Sleep()	getpid() alarm() sleep()
Communication	CreatePipe() CreateFileMapping() MapViewOfFile()	pipe() shmget() mmap()

### Example

`open()`, `read()`, `write()`, `close()`, `mkdir()` are other examples of system calls

### References

- 1) Tutorials Point, Types of System Calls, link
- b) It is user's responsibility to keep track of allocated blocks of heap memory, and memory leak occurs if user fails to deallocate allocated blocks of heap memory

### Notes

- **Memory API**

- Has two types of memory

1. **Stack**

- \* Is also called **automatic memory**
- \* Allocations and deallocations are managed by compiler

- \* Deallocates memory by the end of function call

## 2. Heap

- \* Is long-lived
- \* Allocation and deallocation are managed by user
- \* Creates **memory leak** if memory not freed
- \* **valgrind** is a useful heap memory debugging tool link

### – malloc()

- \* Is a C library call
- \* **Syntax:** void \*malloc(size\_t size)
- \* Allocates a block of size bytes to **heap memory** and if successful, returns a pointer to it
- \* Returns NULL if memory allocation is unsuccessful

### Example

```
int *x = malloc(10 * sizeof(int));
```

### – free()

- \* Is a C library call
- \* Frees heap memory that is no longer in use

### Example

```
int *x = malloc(10 * sizeof(int));
```

```
...
```

```
free(x);
```

### – brk(), sbrk(), mmap()


- \* Are system calls for memory management

## • Buffer overflow

- is an error that occurs when not enough heap memory is allocated

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src)); // too small!
strcpy(dst, src); // work properly
```

Missing + 1



- If the access by two threads are both about reading the stored value (as opposed to write), then concurrency error will not occur.
- Hand-over-hand locking is a fine-grained-locking, which allows more threads to be locked at once than single lock, and this means it will perform better than single lock when there is a lot of contention,

## Notes

- **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
- Performs 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

- **Fine-grained-locking**

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

**Example**

```

1 void List_Init(list_t *L) {
2     L->head = NULL;
3     pthread_mutex_init(&L->lock, NULL);
4 }
5
6 void List_Insert(list_t *L, int key) {
7     // synchronization not needed
8     node_t *new = malloc(sizeof(node_t));
9     if (new == NULL) {
10         perror("malloc");
11         return;
12     }
13     new->key = key;
14
15     // just lock critical section
16     pthread_mutex_lock(&L->lock);
17     new->next = L->head;
18     L->head = new;
19     pthread_mutex_unlock(&L->lock);
20 }
21
22 int List_Lookup(list_t *L, int key) {
23     int rv = -1;
24     pthread_mutex_lock(&L->lock);
25     node_t *curr = L->head;
26     while (curr) {
27         if (curr->key == key) {
28             rv = 0;
29             break;
30         }
31         curr = curr->next;
32     }
33     pthread_mutex_unlock(&L->lock);
34     return rv; // now both success and failure
35 }

```

Fine-grained lock here :)

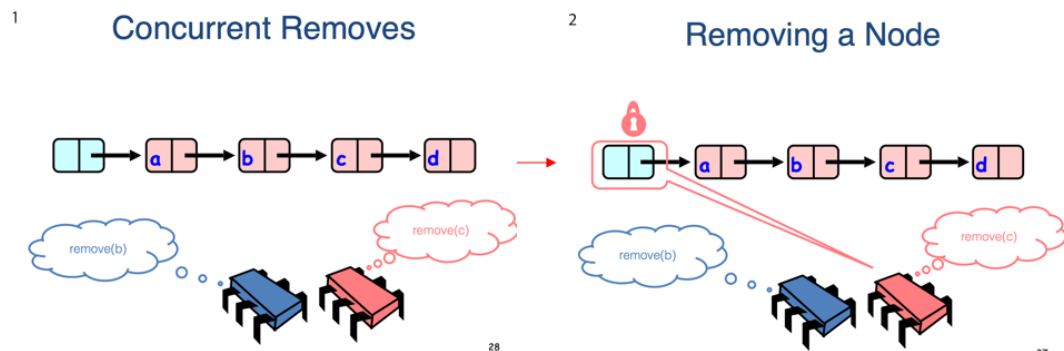
Notice lock is on a struct L

More threads can be locked at once

### • Hand-over-hand locking

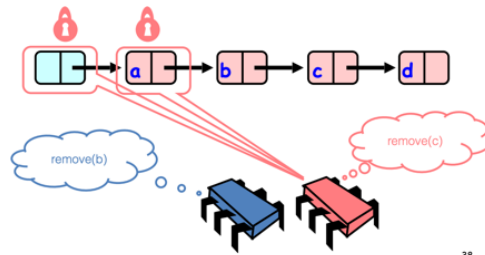
- Idea: instead of having a single lock for the entire list, a lock per node of the list is added; when traversing the list, the list grabs the next node's lock, and releases the current node's lock
- Is a fine-grained-locking
- Holds at most 2 locks at a time

### Example



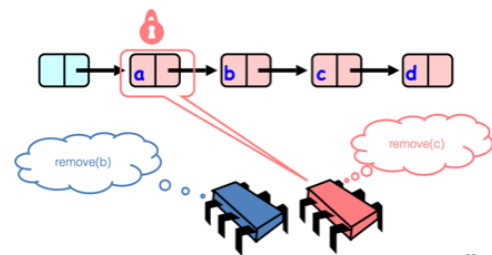
3

## Removing a Node



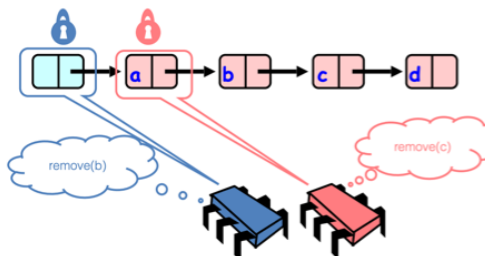
38

## Removing a Node



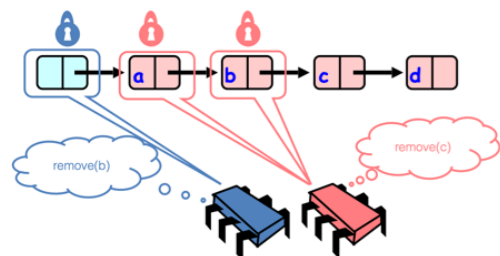
39

## Removing a Node



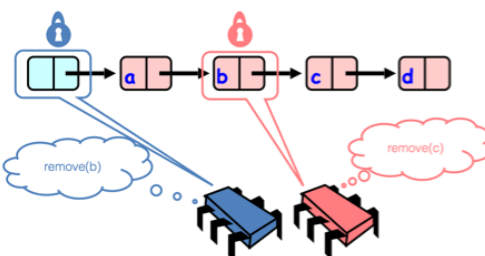
40

## Removing a Node



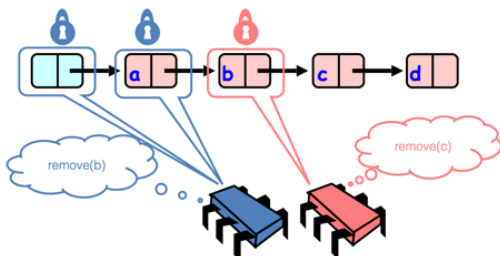
41

## Removing a Node



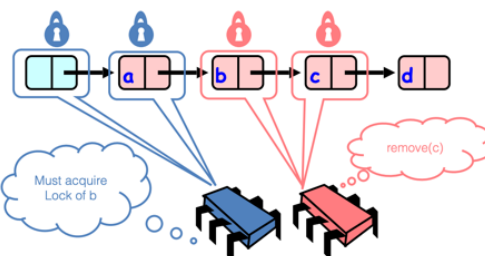
42

## Removing a Node



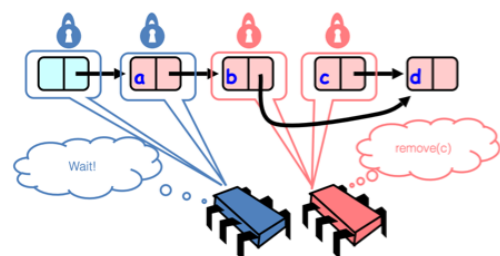
43

## Removing a Node

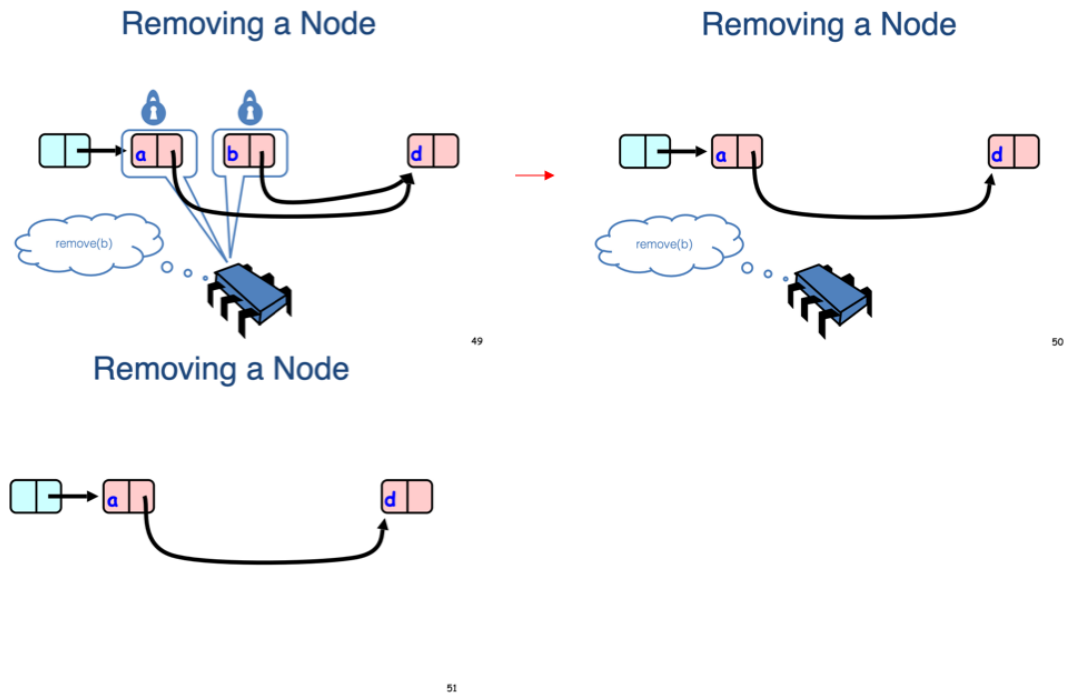


44

## Removing a Node



46



### References

- 1) Techion, Linked Lists: The Role of Locking, link