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.

<u>**Notes**</u>

- **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()

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

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

<u>**References**</u>

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

<u>**Notes**</u>

- **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 memoery 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));
```

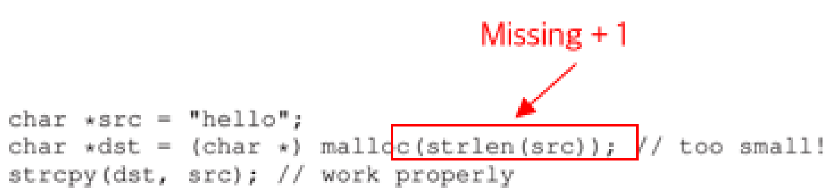
- `free()`
  * Is a C library call
  * Frees heap memory that is no longer in use

  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



c) If the access by two threads are both about reading the stored value (as opposed to write), then concurrency error will not occur.

d) 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
  - Perfoms poorly when contention (the need for performance due to load) is high
    * No concurrent access

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

```
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 strutures
  - Allows more threads to be in locked code at once

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

```c
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
}
```

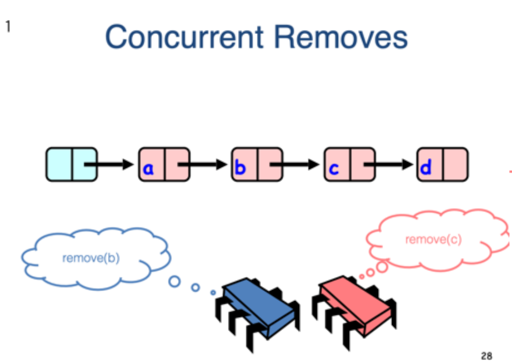
Fine-grained lock here :)

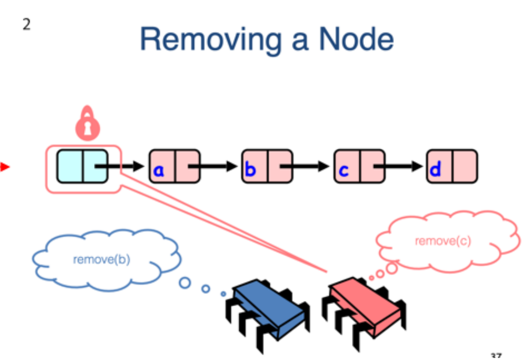
Notice lock is on a struct L

More threads can be locked at once

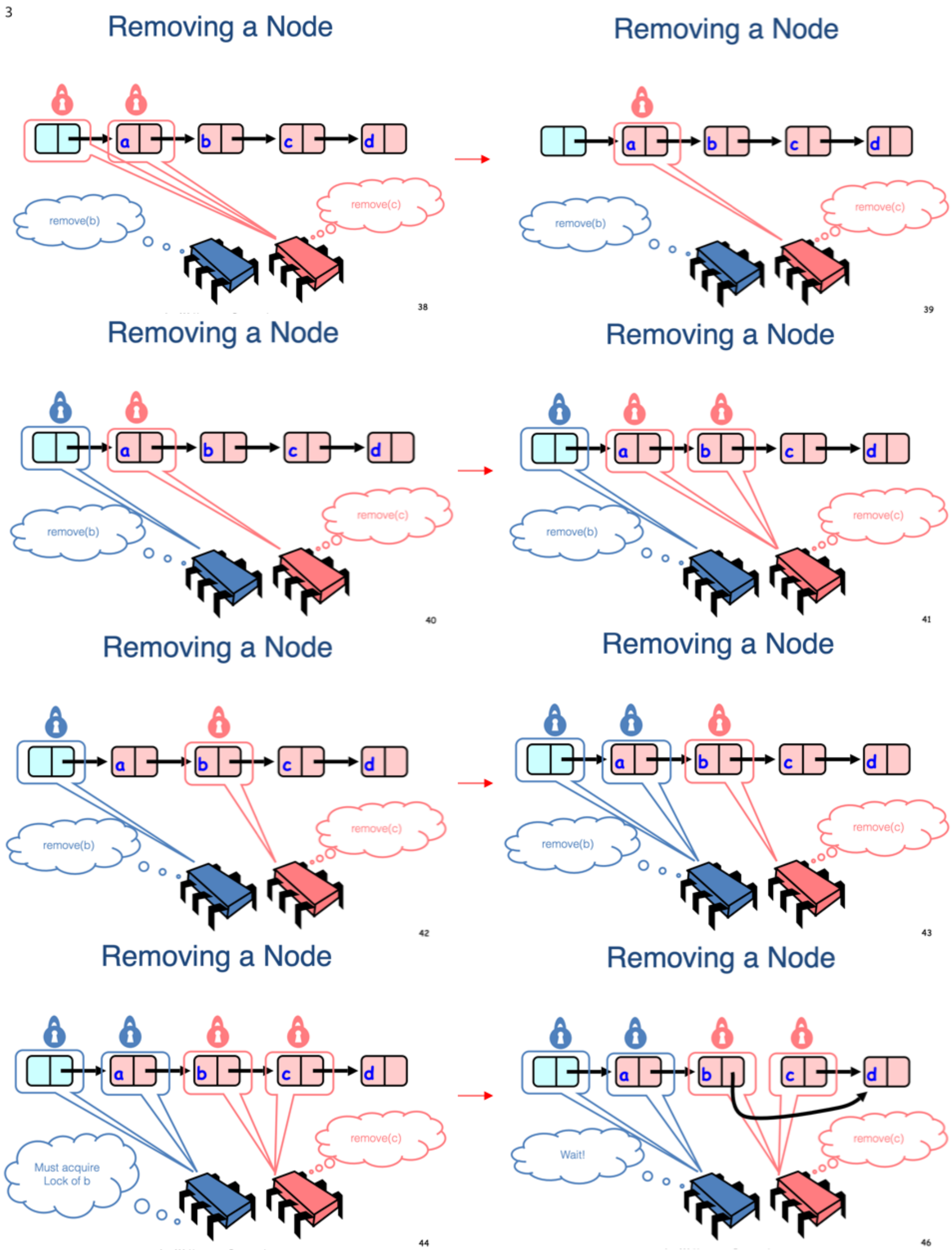
- **Hand-over-hand locking**
  - Idea: instead of having a single lock for the entire list, a lock per node of the list is added; when traversing the list, the list grabs the next node's lock, and releases the current node's lock
  - Is a fine-grained-locking
  - Holds at most 2 locks at a time

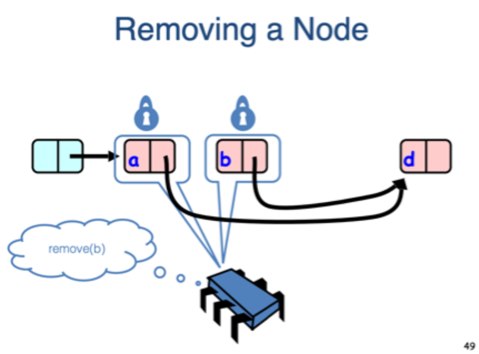
  <u>**Example**</u>

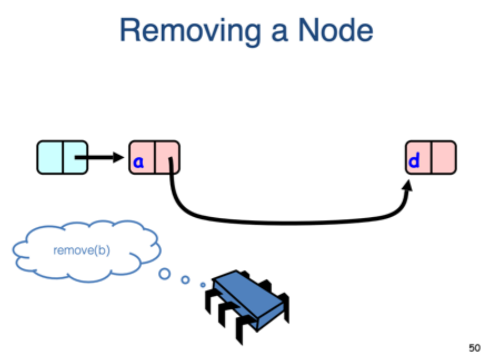


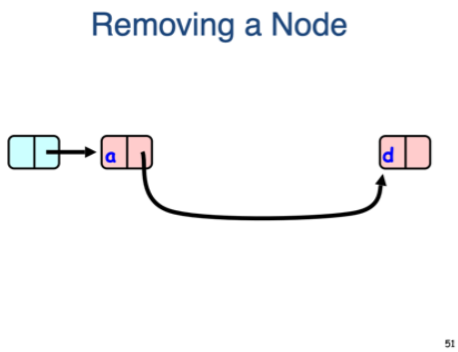


3


Removing a Node
a
b
c
d
remove(b)
remove(c)
38
Removing a Node
a
b
c
d
remove(b)
remove(c)
39
Removing a Node
a
b
c
d
remove(b)
remove(c)
40
Removing a Node
a
b
c
d
remove(b)
remove(c)
41
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







References

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

e) Interactive systems emphasizes quick response time, and for non-preemtive scheduling algorithms, the shorter tasks must wait for a process to finish, which means poor response time.

Notes

- **Preemtive Scheduling Algorihtm**
  - Are designed so that different processes can be executed in the middle of any current process execution.
  - Today, all of the modern scheduling algorithms are **preemptive**

  Example

  Shortest-Time-To-Completion (STCF) Scheduler

- **Non-preemtive Scheduling Algorithm**
  - Are designed so that once a process enters the running state, it cannot be preempted (forestalled) until it completes its allotted time

  Example

  Shortest Job First (SJF) scheduler