Lecture # 1.6
Process Heap
Behind the Curtain

Course: Advanced Operating System

Instructor: Arif Butt

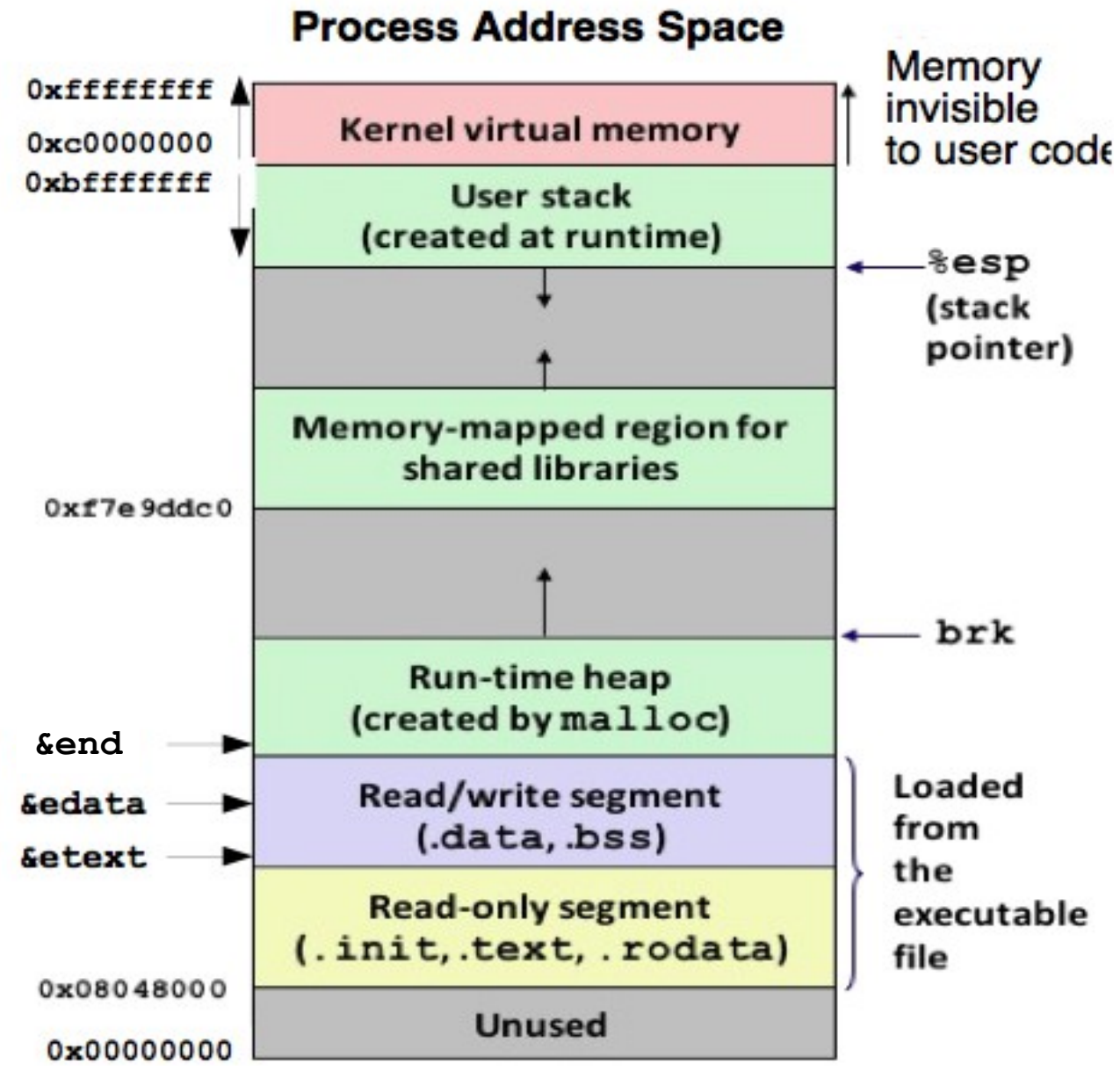
Punjab University College of Information Technology (PUCIT)
University of the Punjab

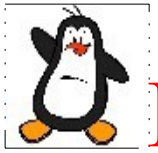
Agenda

- Recap: Allocating, using and freeing memory on heap
- Layout of heap and heap allocators
- System Calls `brk()` and `sbrk()`
- Common programming errors
- Tools and Libraries for malloc debugging
  - Splint
  - Electric Fence
  - Valgrind

RECAP
Allocating, using and freeing Memory on Heap

Process Logical Address Space





Heap Allocators

Allocators come in two basic styles. Both styles require the application to explicitly allocate blocks. They differ about which entity is responsible for freeing allocated blocks

- **Explicit allocators** require the application to explicitly free any allocated blocks. For example, the C standard library provides an explicit allocator called the `malloc` package. C programs allocate a block by calling the `malloc` function and free a block by calling the `free` function. In C++, we normally use the `new` and `delete` operators
- **Implicit allocators** require the allocator to detect when an allocated block is no longer being used by the program and then free the block. Implicit allocators are also known as garbage collectors, and the process of automatically freeing unused allocated blocks is known as garbage collection. For example, higher-level languages such as Lisp, ML and Java rely on garbage collection to free allocated blocks

The malloc family in C

```
void *malloc (size_t size);
void*calloc(size_t noOfObjects, size_t size);
void *realloc (void* ptr, size_t newsize );
void free ( void* ptr );
```

- **malloc()** allocates size bytes from the heap and returns a pointer to the start of the newly allocated block of memory. On failure returns NULL and sets **errno** to indicate error
- **calloc()** allocates space for specific number of objects, each of specified size. Returns a pointer to the start of the newly allocated block of memory. Unlike **malloc(), calloc()** initializes the allocated memory to zero. On failure returns NULL and sets **errno** to indicate error
- **realloc()** is used to resize a block of memory previously allocated by one of the functions in **malloc()** package. Ptr argument is the pointer to the block of memory that is to be resized. On success **realloc()** returns a pointer to the location of the resized block, which may be different from its location before the call. On failure, returns NULL and leaves the previous block pointed to by pointer untouched.
- **free()** deallocates the block of memory pointed to by its pointer argument, which should be an address previously returned by functions of **malloc** package

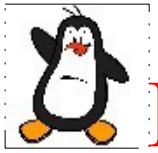
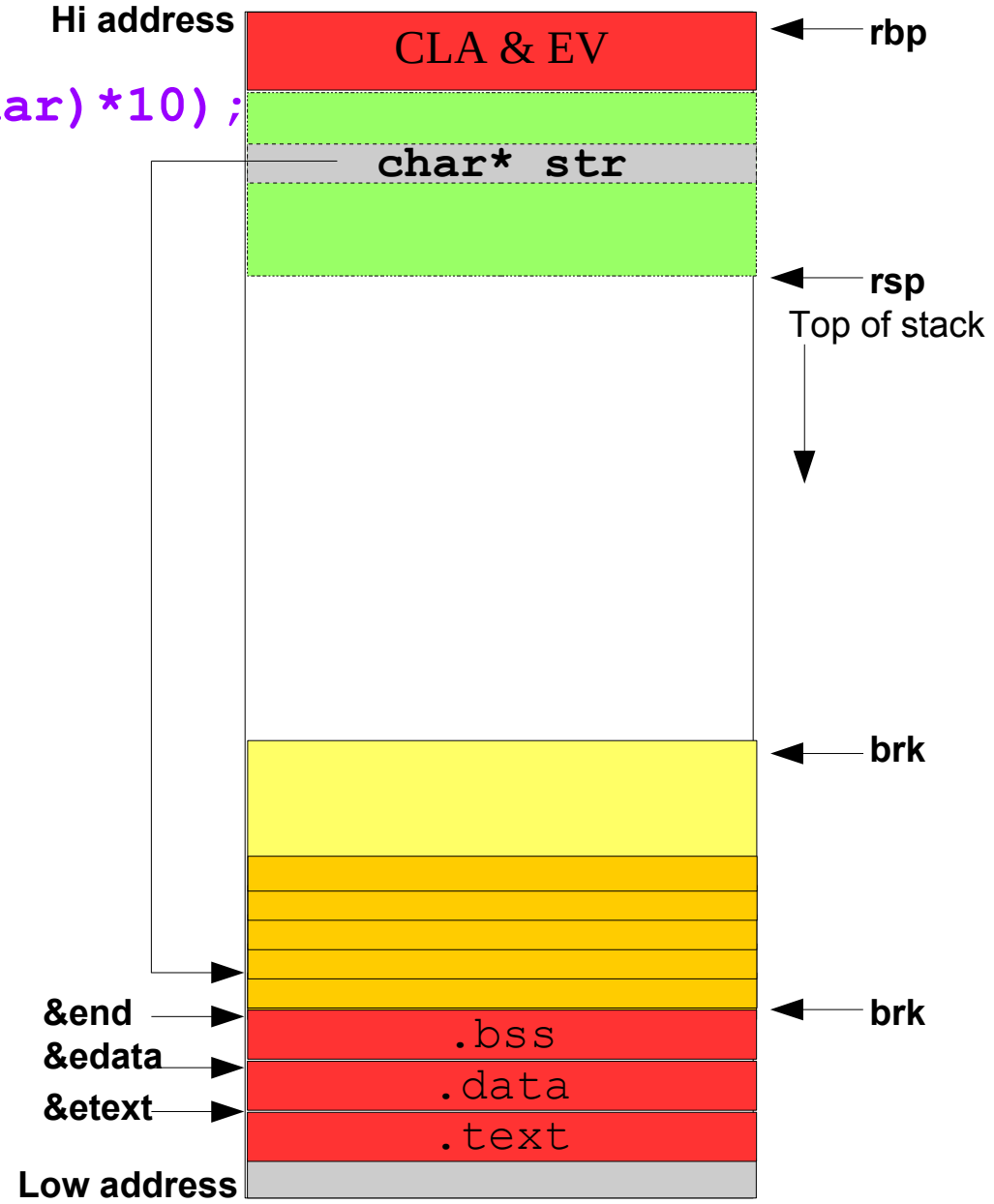


Illustration: 1D Array on Heap

```
char*str=(char*)malloc(sizeof(char)*10);
    ...
    ...
    ...
free(str);
```



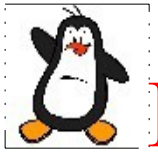
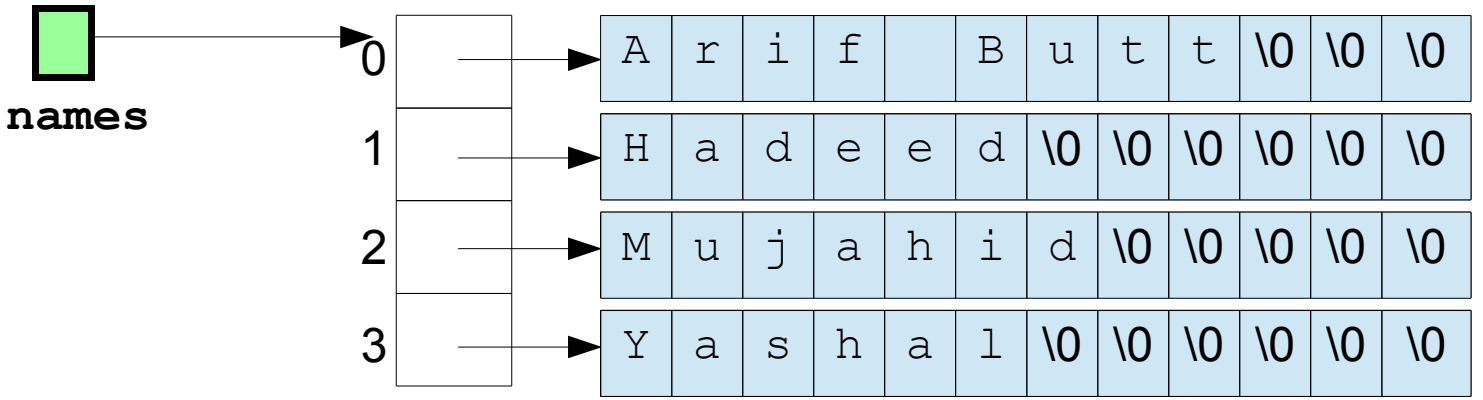


Illustration: 2D Array on Heap

```
int i, int rows = 4, cols = 12;
char ** names = (char**)malloc(sizeof(char*) * rows);
for(i = 0; i < rows; i++)
   names[i] = (char*)malloc(sizeof(char) * cols);
…
…
…
for(i = 0; i < rows; i++)
   free(names[i]);
free(names);
```



$100 Question

If a process continuously calls **`malloc()`**, without calling **`free()`**, what happens and why?

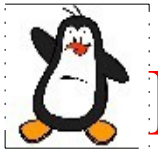
Example: `heapfiller.c`

Heap: Behind the curtain

System Call: `brk()`

```
int brk(void* end_data_segment);
```

- Resizing the heap is actually telling the kernel to adjust the process's **program break**, which lies initially just above the end of the uninitialized data segment (i.e `end` variable)
- **`brk()`** is a system call that sets the **program break** to location specified by **end_data_segment**. Since virtual memory is allocated in pages, this request is rounded up to the page boundary. Any attempt to lower the program break than **`end`** results in segmentation fault
- The upper limit to which the program break can be set depends on range of factors like:
  - Process resource limit for size of data segment
  - Location of memory mappings, shared memory segment and shared libraries



Library Call: `sbrk()`

```
void *sbrk (intptr_t increment);
```

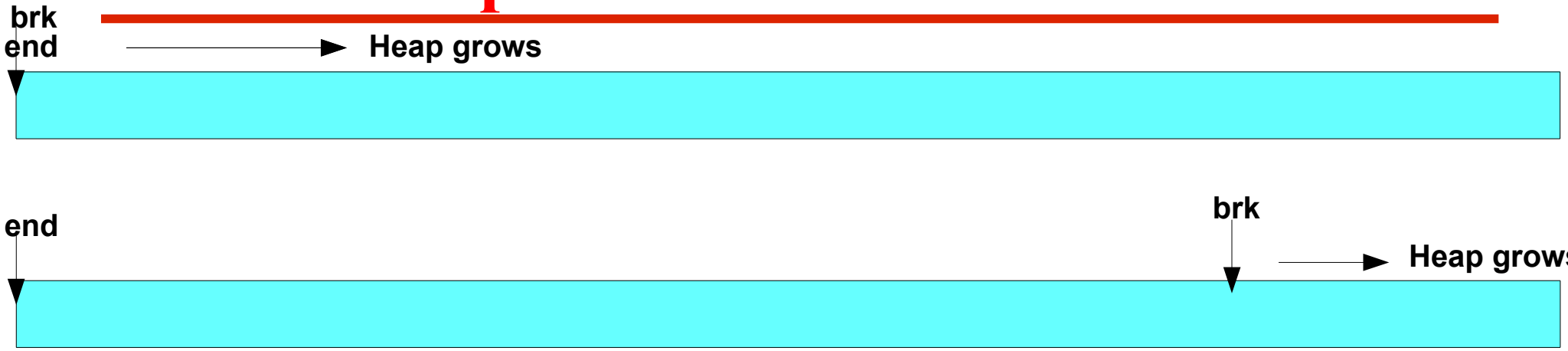
- **`sbrk()`** is a C library wrapper implemented on top of **`brk()`**. It increments the program break by **increment** bytes
- On success, **`sbrk()`** returns a pointer to the end of the heap before **`sbrk()`** was called, i.e., a pointer to the start of new area
- So calling **`sbrk(0)`** returns the current setting of the program break without changing it
- On failure -1 is returned with **`errno ENOMEM`**

$100 Question

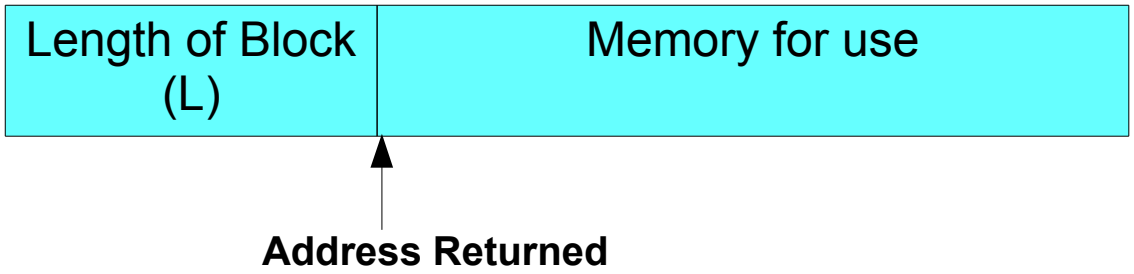
After a process calls **`malloc()`**, which in turn calls **`brk()`**, what is the new location of program break **`brk`**?

Example: `brk.c`

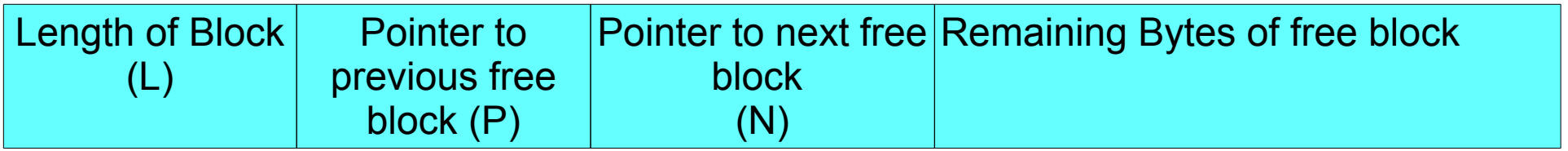
A Basic Heap Allocator



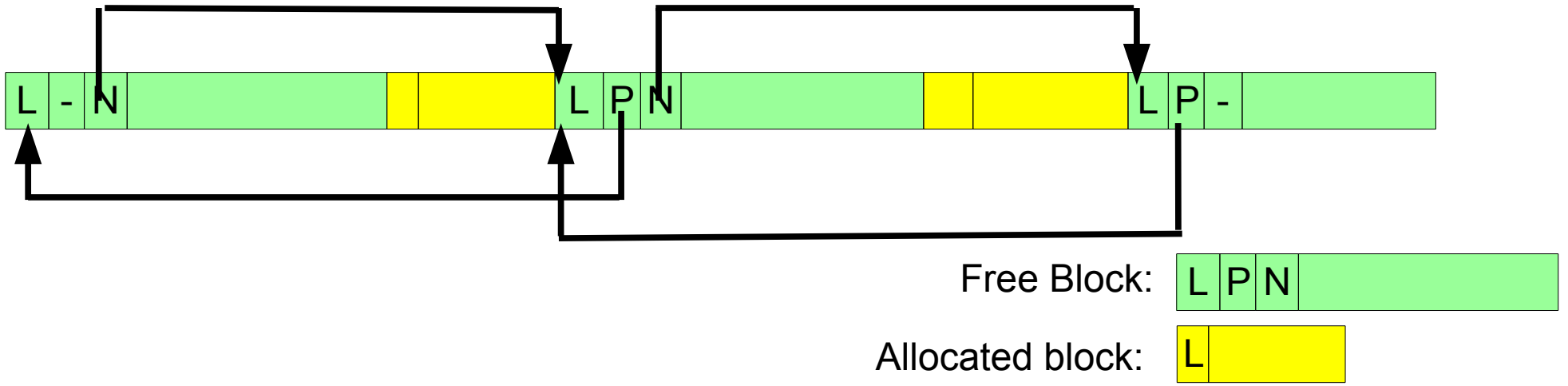
Structure of Allocated Block on Heap:



Structure of Free Block on Heap:

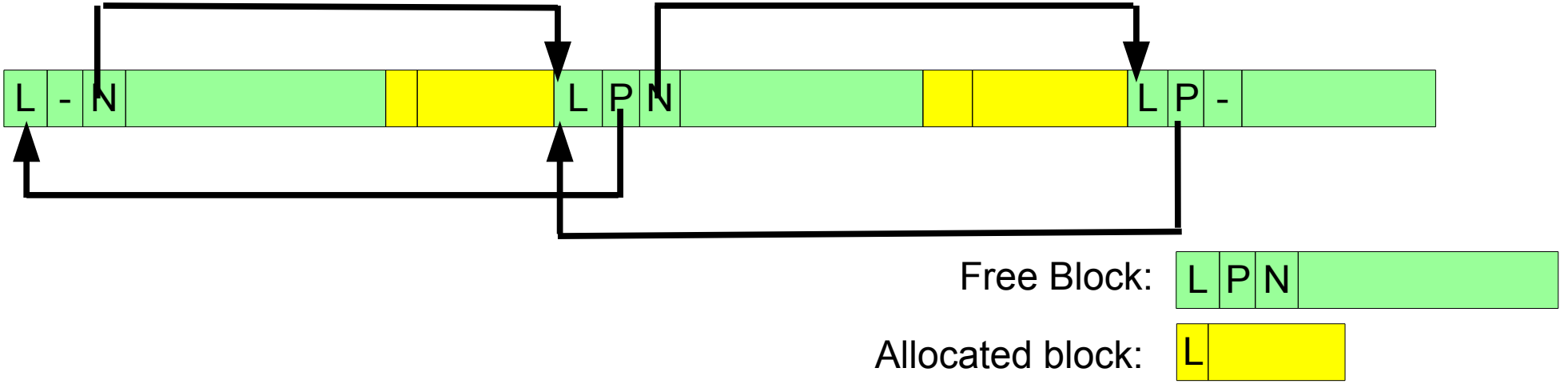


A Basic Heap Allocator (cont...)



- When a program calls **`malloc`** the allocator scans the link list of free memory blocks as per one of the contiguous memory allocation algorithm's (**first fit, best fit, next fit**), assigns the block and update the data structures
- If no block on the free list is large enough then **`malloc()`** calls **`sbrk()`** to allocate more memory
- To reduce the number of calls to **`sbrk()`**, **`malloc()`** do not allocate exact number of bytes required rather increase the **`program break`** in large units (some multiples of virtual memory page size) and putting the excess memory onto the free list

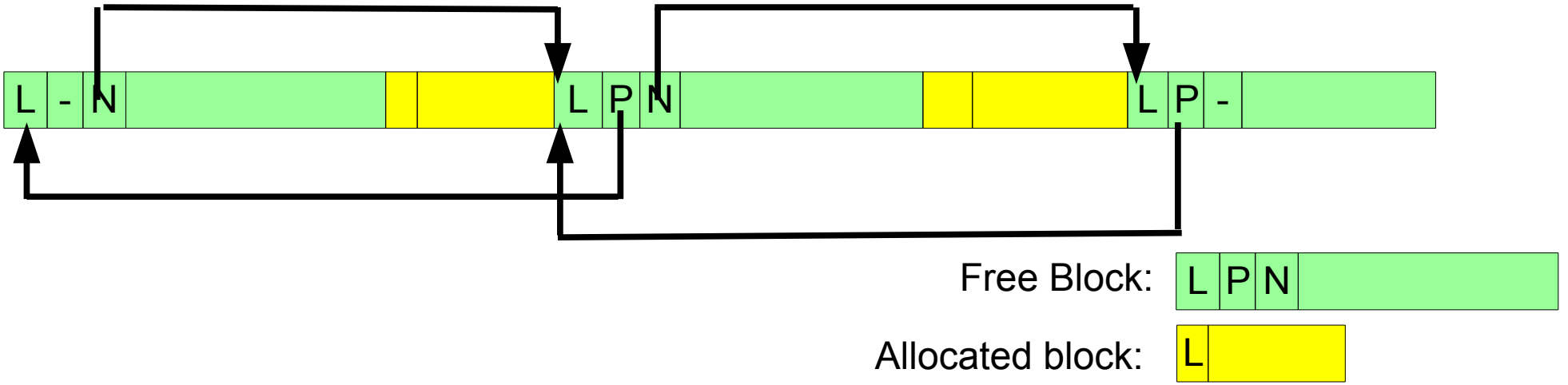
A Basic Heap Allocator (cont...)



$100 Question

When a process calls **`free()`**, how does it know as to how much memory it needs to free? Does it has any effect on program break **`brk`**?

A Basic Heap Allocator (cont...)



Coalescing Freed Memory

- When you call `free()`, you put a chunk of memory back on the free list
- There may be times when the chunk immediately before it in memory, and/or the chunk immediately after it in memory are also free
- If so, it make sense to try to merge the free chunks into one free chunk, rather than having three contiguous free chunks on the free list
- This is called "**coalescing**" free chunks

Why not use `brk()` and `sbrk()`

C program use **`malloc`** family of functions to allocate & deallocate memory on the heap instead of **`brk()` & `sbrk()`**, because:

- **`malloc`** functions are standardized as part of C language
- **`malloc`** functions are easier to use in threaded programs
- **`malloc`** functions provide a simple interface that allows memory to be allocated in small units
- **`malloc`** functions allow us to deallocate blocks of memory

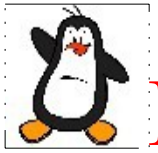
Why **free()** doesn't lower the program break? rather adds the block of memory to a lists of free blocks to be used by future calls to **`malloc()`**. This is done for following **reasons**:

- Block of memory being freed is somewhere in the middle of the heap, rather than at the end, so lowering the program break is not possible
- It minimizes the numbers of **sbrk()** calls that the program must perform

$100 Question

When a process requests 1-24 B on heap, why the memory allocated is 32 B?
When a process requests 25-40 B on heap, why the memory allocated is 48 B?
When a process requests 41-56 B on heap, why the memory allocated is 64 B?
When a process requests 57-72 B on heap, why the memory allocated is 80 B?

Example: `allocated_block.c`



Points to Ponder (Heap)

Common programming errors while using heap:

1. Reading/writing freed memory areas
2. Reading/writing memory addresses before or after the allocated memory using faulty pointer arithmetic
3. Freeing the same piece of allocated memory more than once
4. Freeing heap memory by a pointer, that wasn't obtained by a call to **`malloc`** package
5. Memory leaks, i.e., not freeing memory and keep just allocating it

Static checking using `splint` Example: `hellobug.c`

Tools and Libraries for `malloc` debugging

malloc Debugging libraries

- Tasks of finding dynamic memory allocation bugs can be eased considerably by using the **malloc debugging libraries** that are designed for this purpose
- In order to use these libraries we need to link our program against a particular library instead of **malloc** package in the standard C library
- These libraries operates at the cost of slower **run-time** operation , increased memory consumption or both. So we should use them only for debugging purpose, and then return to linking with the standard **malloc** package for the production version of an application. Some example libraries are
  1. Electric Fence
  2. Valgrind
  3. Insure++
  4. IBM Rational Purify

Electric Fence Example: `hellobug.c`

valgrind Debugging Library

Valgrind is a suite of command line tools for debugging and profiling

- **Memcheck:** It is the default, which is a memory error detector
- **Cachegrind:** It is a cache and branch-prediction profiler. It help you make your program run faster
- **Callgrind:** It is a call graph generating cache profiler
- **Helgrind:** It is a thread error detector
- **Drd:** It is also a thread error detector

While using valgrind, it is recommended to compile with options like no optimization (-O0), and including debugging info (-g)

```
$ gcc -Wall -std=c99 -O0 -g prog1.c
```

```
$ valgrind [valgrind-options] yourprogram [yourprogram options]
```

```
$ valgrind ./a.out
```

Valgrind Example: `faultyprogram.c`

Things To Do

If you have problems visit me in counseling hours. . . .