NEU CS 3650 Computer Systems

Instructor: Dr. Ziming Zhao

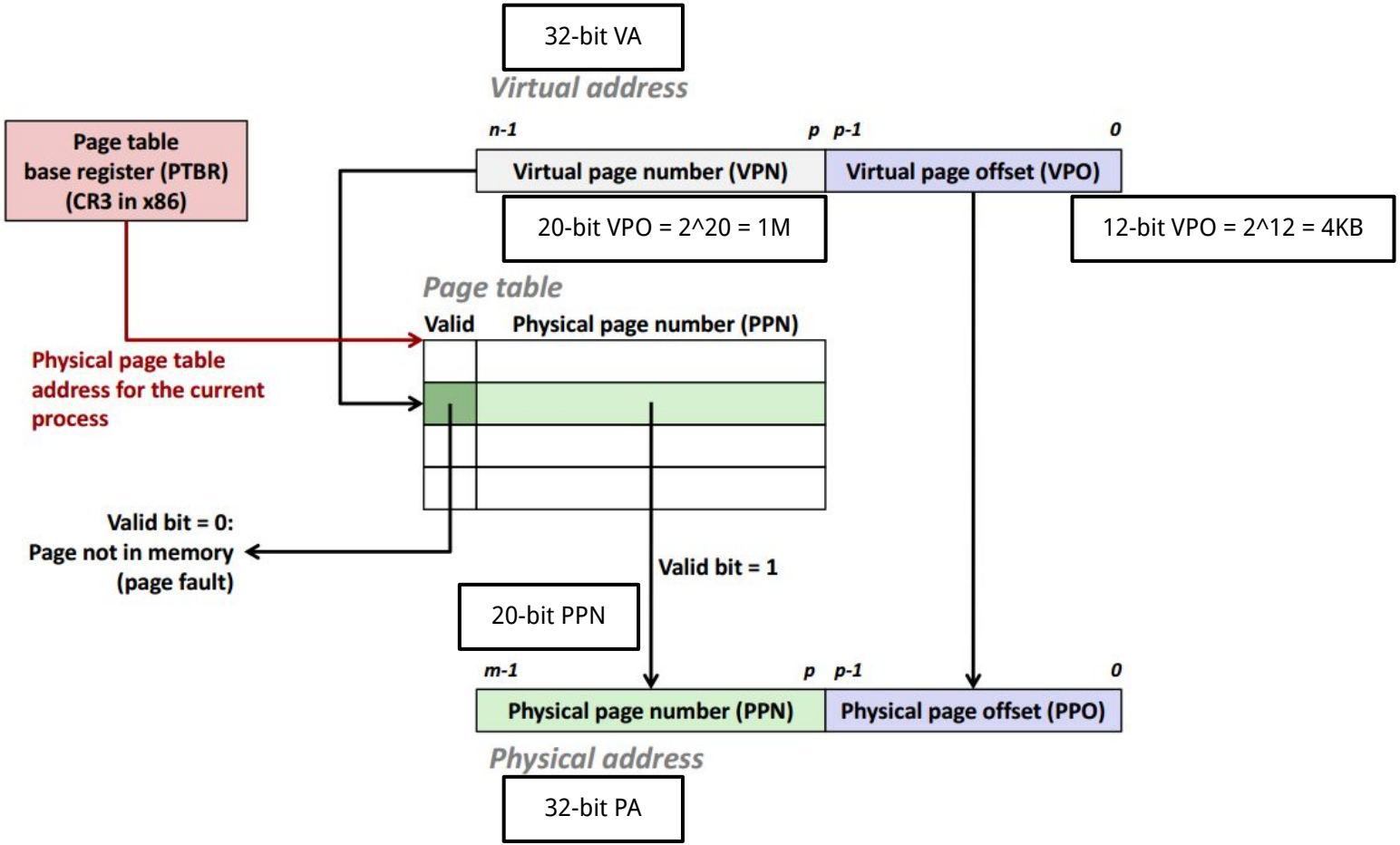
* Acknowledgements: created based on Christo Wilson, Ferdinand Vesely, Alden Jackson, Ben Weintraub, Gene Cooperman, Peter Desnoyers' lecture slides for the same course.

x86 32-bit as an Example

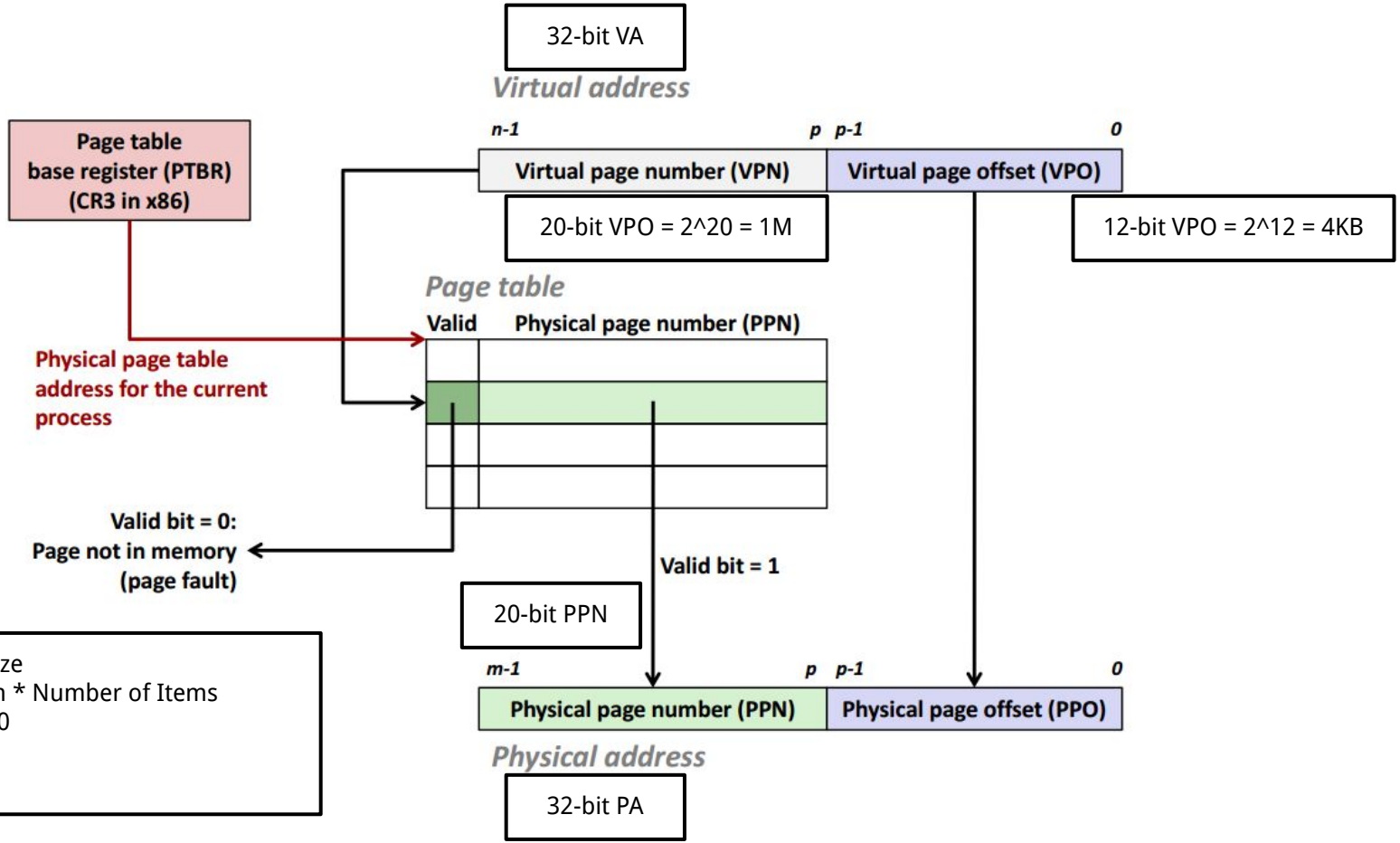
x86 32-bit as an Example: Goal

- Translate 32-bit virtual address to 32-bit physical address
- A page is 4096 (4KB).

How Big is the Page Table?



How Big is the Page Table?



Page Table Size
= Size of Item * Number of Items
= 2^20 * 2^20
= 2^40
= 1TB

Observations

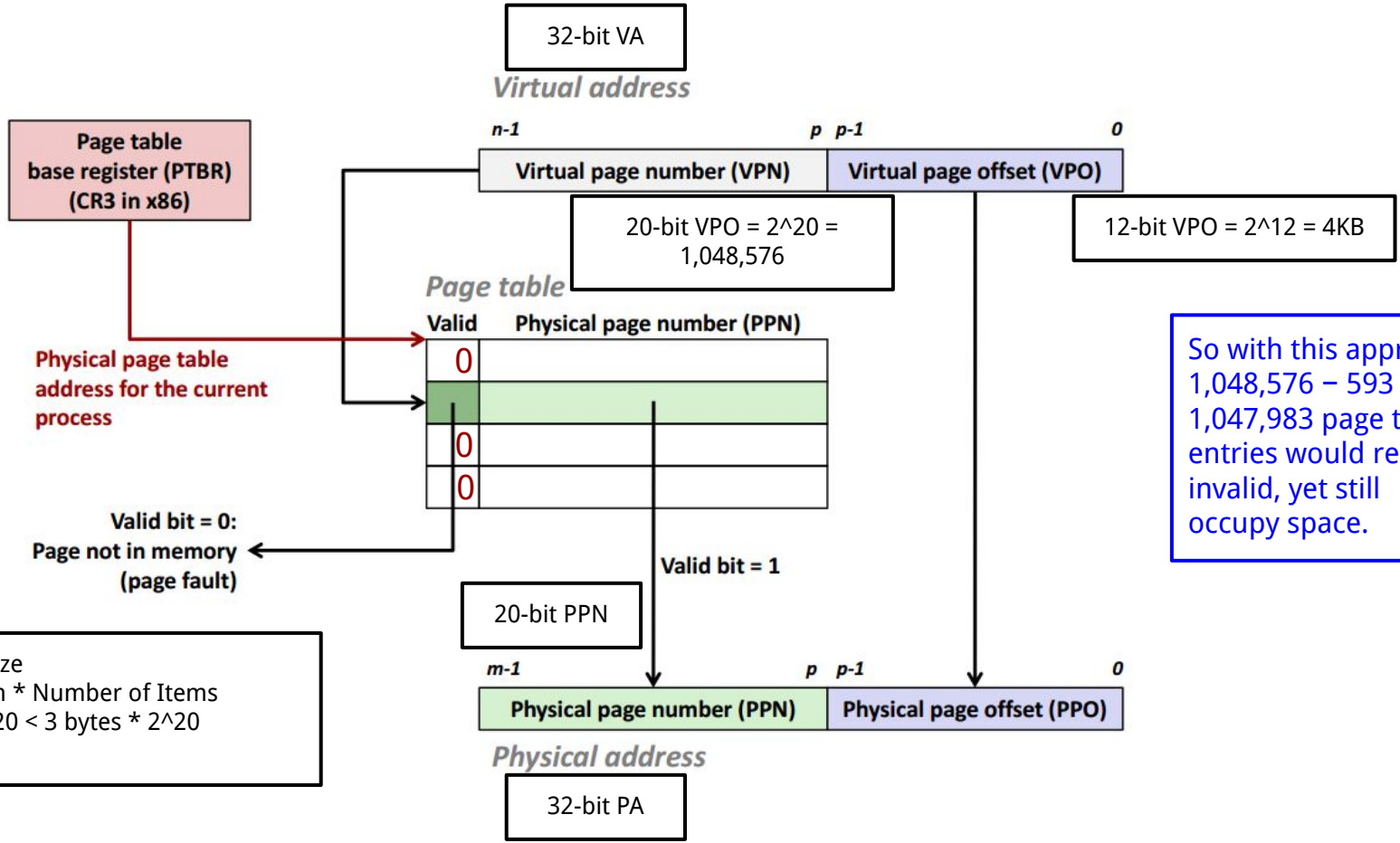
- A large portion of the Virtual Address Space is not used. So not every virtual page number (VPN) needs to be translated. (In the example below: only 2372/4 = 593 pages are used)

1 GB = 0x40000000

2 GB = 0x80000000

```
ziming@ziming-ThinkPad:~/Dropbox/myTeaching/System Security - Attack and Defense for Binaries UB 2020/code/processmap$ pmap -X 21732
21732:   ./pm
 Address Perm   Offset Device    Inode Size Rss Pss Referenced Anonymous LazyFree ShmemPmdMapped Shared_Hugetlb Private_Hugetlb Swap SwapPss Locked Mapping
56569000 r-xp 00000000 103:02 28575310    4   4   4          4         0        0             0              0               0    0       0      0 pm
5656a000 r--p 00000000 103:02 28575310    4   4   4          4         4        0             0              0               0    0       0      0 pm
5656b000 rw-p 00001000 103:02 28575310    4   4   4          4         4        0             0              0               0    0       0      0 pm
57cf2000 rw-p 00000000  00:00        0  136   4   4          4         4        0             0              0               0    0       0      0 [heap]
f7d73000 r-xp 00000000 103:02  2883591 1876 772 772        772         0        0             0              0               0    0       0      0 libc-2.27.so
f7f48000 ---p 001d5000 103:02  2883591    4   0   0          0         0        0             0              0               0    0       0      0 libc-2.27.so
f7f49000 r--p 001d5000 103:02  2883591    8   8   8          8         8        0             0              0               0    0       0      0 libc-2.27.so
f7f4b000 rw-p 001d7000 103:02  2883591    4   4   4          4         4        0             0              0               0    0       0      0 libc-2.27.so
f7f4c000 rw-p 00000000  00:00        0   12   8   8          8         8        0             0              0               0    0       0      0
f7f75000 rw-p 00000000  00:00        0    8   8   8          8         8        0             0              0               0    0       0      0
f7f77000 r--p 00000000  00:00        0   12   0   0          0         0        0             0              0               0    0       0      0 [vvar]
f7f7a000 r-xp 00000000  00:00        0    8   8   8          8         0        0             0              0               0    0       0      0 [vdso]
f7f7c000 r-xp 00000000 103:02  2883587  152 144 144        144         0        0             0              0               0    0       0      0 ld-2.27.so
f7fa2000 r--p 00025000 103:02  2883587    4   4   4          4         4        0             0              0               0    0       0      0 ld-2.27.so
f7fa3000 rw-p 00026000 103:02  2883587    4   4   4          4         4        0             0              0               0    0       0      0 ld-2.27.so
ffef3000 rw-p 00000000  00:00        0  132  12  12         12        12        0             0              0               0    0       0      0 [stack]
                                        ==== === === ========== ========= ======== ============= ============== =============== ==== ======= ======
                                        2372 988 988        988        60        0             0              0               0    0       0      0 KB
```

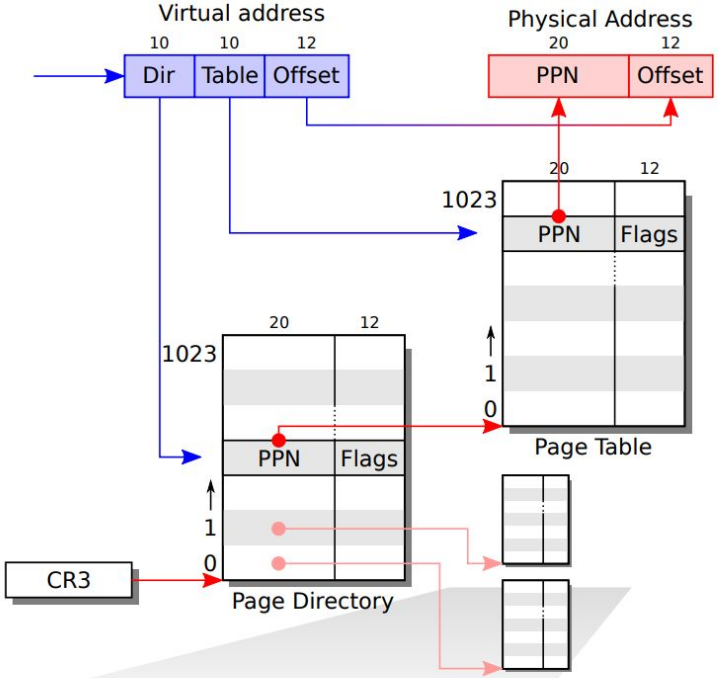
How Big is the Page Table?

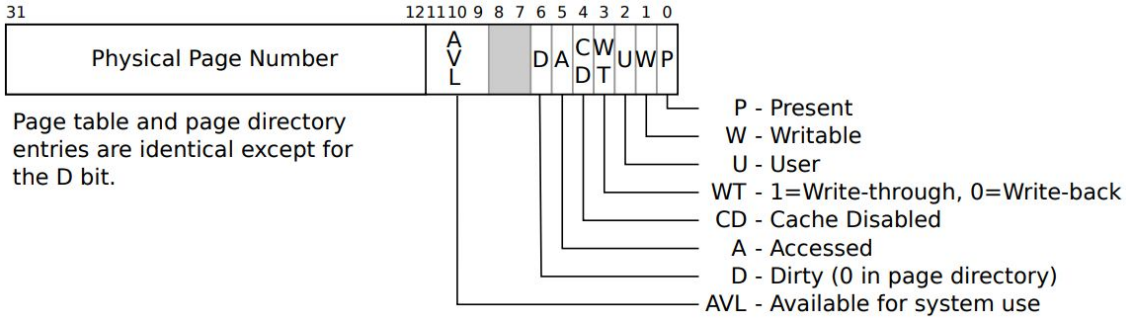


So with this approach, 1,048,576 − 593 = 1,047,983 page table entries would remain invalid, yet still occupy space.

Page Table Size
= Size of Item * Number of Items
= 20 bit * 2^20 < 3 bytes * 2^20
= 3MB

The solution





- A page table is stored in physical memory as a two-level tree.
- The root of the tree is a 4096-byte page directory that contains 1024 PTE-like references to page table pages.
- Each page table page is an array of 1024 32-bit PTEs.
- This two-level structure allows a page table to omit entire page table pages.

xv6 definitions

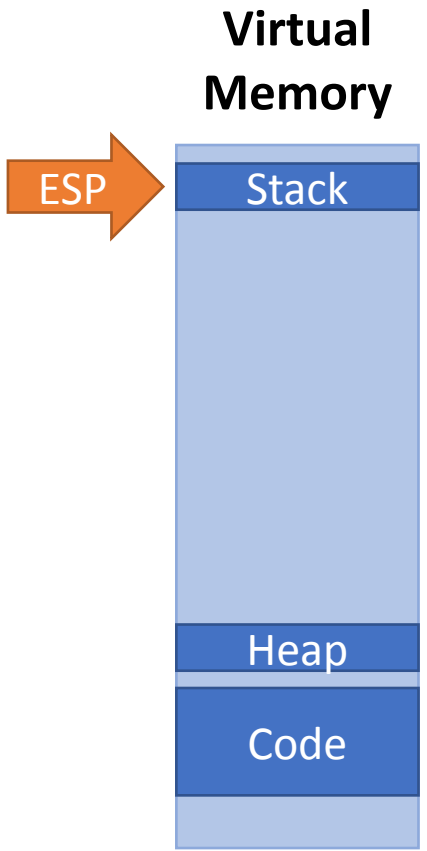
```
// page directory index
#define PDX(va)         (((uint)(va) >> PDXSHIFT) & 0x3FF)

// page table index
#define PTX(va)         (((uint)(va) >> PTXSHIFT) & 0x3FF)

// construct virtual address from indexes and offset
#define PGADDR(d, t, o) ((uint)((d) << PDXSHIFT | (t) << PTXSHIFT | (o)))

// Page directory and page table constants.
#define NPDENTRIES      1024    // # directory entries per page directory
#define NPTENTRIES      1024    // # PTEs per page table
#define PGSIZE          4096    // bytes mapped by a page

#define PTXSHIFT        12      // offset of PTX in a linear address
#define PDXSHIFT        22      // offset of PDX in a linear address

#define PGROUNDUP(sz)  (((sz)+PGSIZE-1) & ~(PGSIZE-1))
#define PGROUNDDOWN(a) (((a)) & ~(PGSIZE-1))

// Page table/directory entry flags.
#define PTE_P           0x001   // Present
#define PTE_W           0x002   // Writeable
#define PTE_U           0x004   // User
#define PTE_PS          0x080   // Page Size

// Address in page table or page directory entry
#define PTE_ADDR(pte)   ((uint)(pte) & ~0xFFF)
#define PTE_FLAGS(pte)  ((uint)(pte) &  0xFFF)
```

https://github.com/mit-pdos/xv6-public/blob/master/mmu.h

Memory allocators

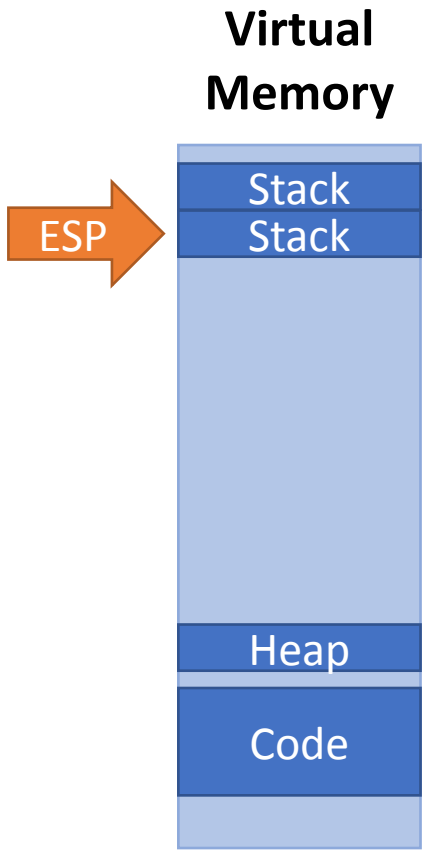
Dynamic Allocation of Pages

- Page tables allow the OS to dynamically assign physical frames to processes on-demand
  - E.g., if the stack grows, the OS can map in an additional page



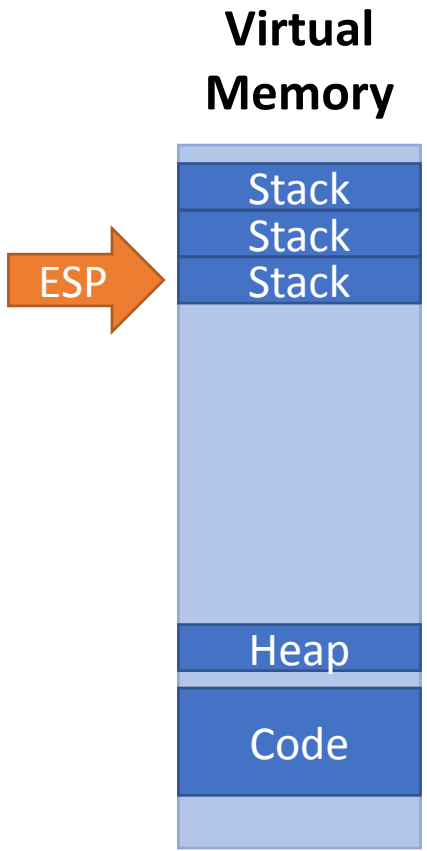
Dynamic Allocation of Pages

- Page tables allow the OS to dynamically assign physical frames to processes on-demand
  - E.g., if the stack grows, the OS can map in an additional page



Dynamic Allocation of Pages

- Page tables allow the OS to dynamically assign physical frames to processes on-demand
  - E.g., if the stack grows, the OS can map in an additional page

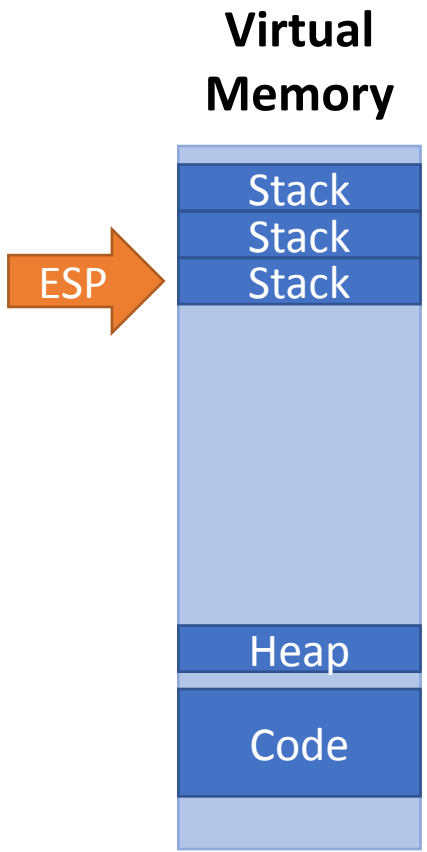


Dynamic Allocation of Pages

- Page tables allow the OS to dynamically assign physical frames to processes on-demand
  - E.g., if the stack grows, the OS can map in an additional page

- On Linux, processes use sbrk()/brk()/mmap() to request additional heap pages
  - But these syscalls only allocates memory in multiples of 4KB

  - **Why 4KB?**

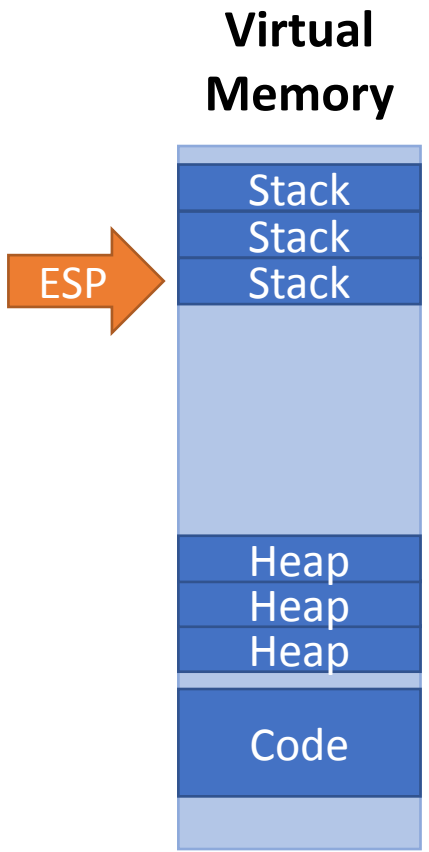


Dynamic Allocation of Pages

- Page tables allow the OS to dynamically assign physical frames to processes on-demand
  - E.g., if the stack grows, the OS can map in an additional page

- On Linux, processes use sbrk()/brk()/mmap() to request additional heap pages
  - But these syscalls only allocates memory in multiples of 4KB

  - **Why 4KB?**



What About malloc() and free()?

- The OS only allocates and frees memory in units of 4KB pages
  - What if you want to allocate <4KB of memory?
  - E.g. char * string = (char *) malloc(100);

What About malloc() and free()?

- The OS only allocates and frees memory in units of 4KB pages
  - What if you want to allocate <4KB of memory?
  - E.g. char * string = (char *) malloc(100);

- Each process manages its own heap memory
  - On Linux, glibc implements *malloc()* and *free()*, manages objects on the heap
  - The JVM uses a garbage collector to manage the heap

What About malloc() and free()?

- The OS only allocates and frees memory in units of 4KB pages
  - What if you want to allocate <4KB of memory?
  - E.g. char * string = (char *) malloc(100);
- Each process manages its own heap memory
  - On Linux, glibc implements *malloc()* and *free()*, manages objects on the heap
  - The JVM uses a garbage collector to manage the heap
- There are many different strategies for managing free memory

Free Space Management

- How do processes manage free memory?
  1. Explicit memory management
     - Languages like C, C++; programmers control memory allocation and deallocation
  2. Implicit memory management
     - Languages like Java, Javascript, Python; runtime takes care of freeing useless objects from memory

- In both cases, software must keep track of the memory that is in use or available

Why Should You Care?

- Regardless of language, all of our code uses dynamic memory
- However, there is a performance cost associated with using dynamic memory
- Understanding how the heap is managed leads to:
  - More performant applications
  - The ability to diagnose difficult memory related errors and performance bottlenecks

Setting the Stage

- Many languages allow programmers to explicitly allocate and deallocate memory
  - C, C++
  - *malloc()* and *free()*
- Programmers can *malloc()* any size of memory
  - Not limited to 4KB pages
- *free()* takes a pointer, but not a size
  - How does *free()* know how many bytes to deallocate?
- Pointers to allocated memory are returned to the programmer
  - As opposed to Java or C# where pointers are “managed”
  - Code may modify these pointers

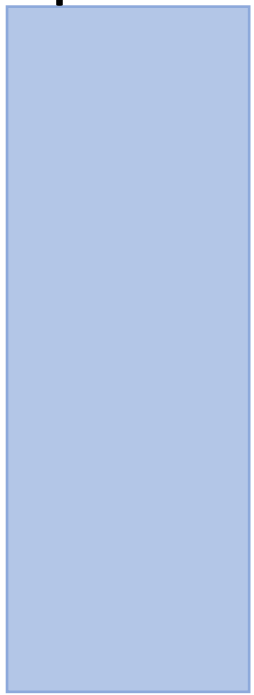
Requirements and Goals

- Keep track of memory usage
  - What bytes of the heap are currently allocated/unallocated?
- Store the size of each allocation
  - So that *free()* will work with just a pointer
- Minimize fragmentation
  - … without doing compaction or relocation
  - More on this later
- Maintain higher performance
  - O(1) operations are obviously faster than O(n), etc.
  - We won't cover this in class; you may refer to the textbook

Heap Fragmentation

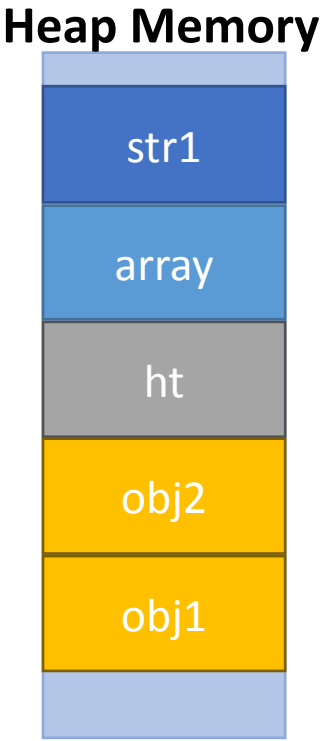
```
obj * obj1, * obj2;
hash_tbl * ht;
int array[];
char * str1, * str2;
… // allocation of objects
```

Heap Memory



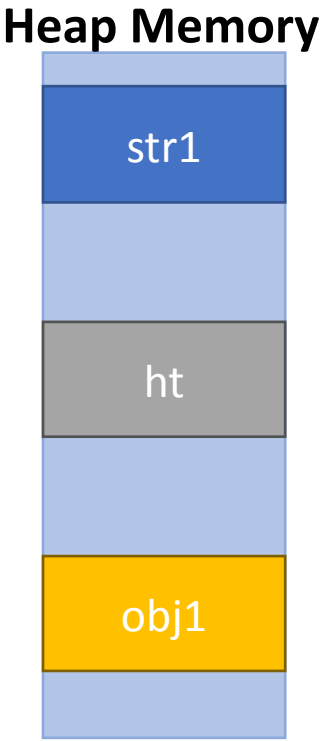
Heap Fragmentation

```
obj * obj1, * obj2;
hash_tbl * ht;
int array[];
char * str1, * str2;
… // allocation of objects
…
free(obj2);
free(array);
```



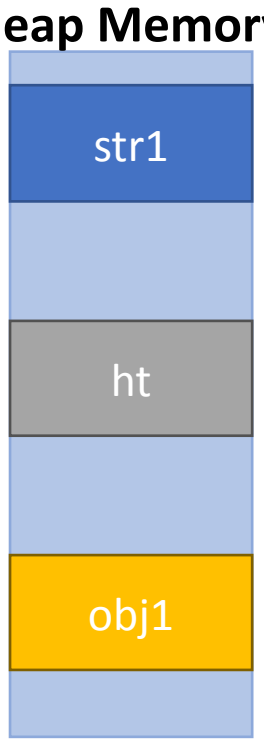
Heap Fragmentation

```
obj * obj1, * obj2;
hash_tbl * ht;
int array[];
char * str1, * str2;
… // allocation of objects
…
free(obj2);
free(array);
```



Heap Fragmentation

```
obj * obj1, * obj2;
hash_tbl * ht;
int array[];
char * str1, * str2;
… // allocation of objects
…
free(obj2);
free(array);
…
str2 = (char *) malloc(300);
```



- This is an example of **external** fragmentation
  - There is enough empty space for str2, but the space isn't usable
- As we will see, internal fragmentation may also be an issue

The Free List

- A free list is a simple data structure for managing heap memory
- Three key components
  1. A linked-list that records free regions of memory
     - Free regions get split when memory is allocated
     - Free list is kept in sorted order by memory address
  2. Each allocated block of memory has a header that records the size of the block
  3. An algorithm that selects which free region of memory to use for each allocation request

The Free List

- A free list is a simple data structure for managing heap memory
- Three key components
  1. A linked-list that records free regions of memory
     - Free regions get split when memory is allocated
     - Free kept in sorted order by memory address
  2. Each al ck of memory has a header that records the size of
  3

- Design challenge: linked lists are dynamic data structures
- Dynamic data structures go on the heap
- But in this case, we are implementing the heap?!

Free List Data Structures

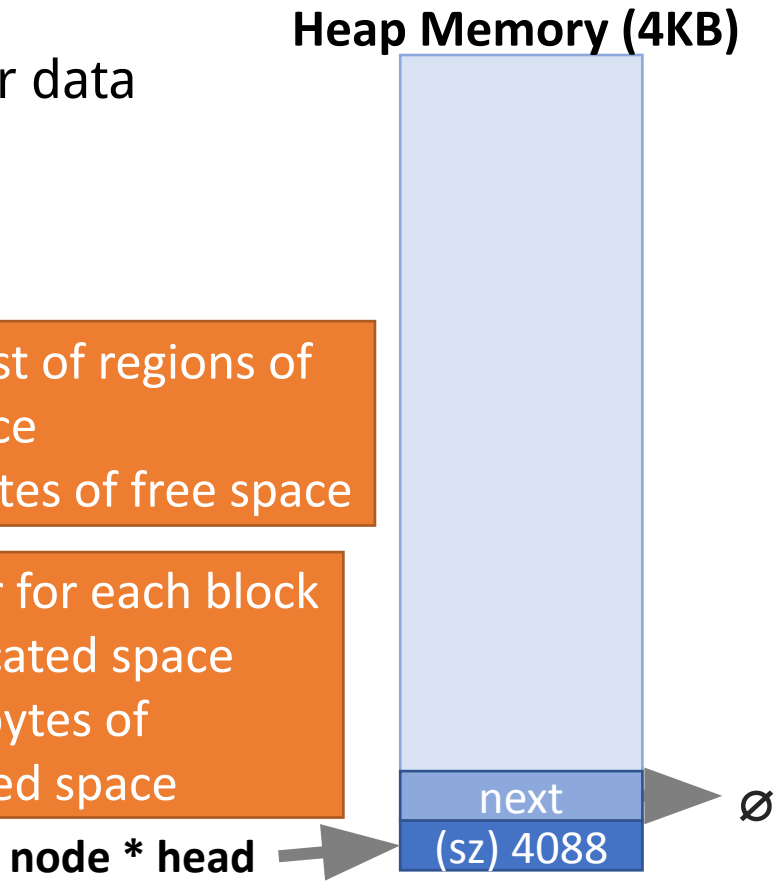
- The free list is a linked list
- Stored in heap memory, alongside other data
- For *malloc(n)*:

  num_bytes = *n* + sizeof(header)

```
typedef struct node_t {
    int size;
    struct node_t * next;
} node;

typedef struct header_t {
    int size;
} header;
```

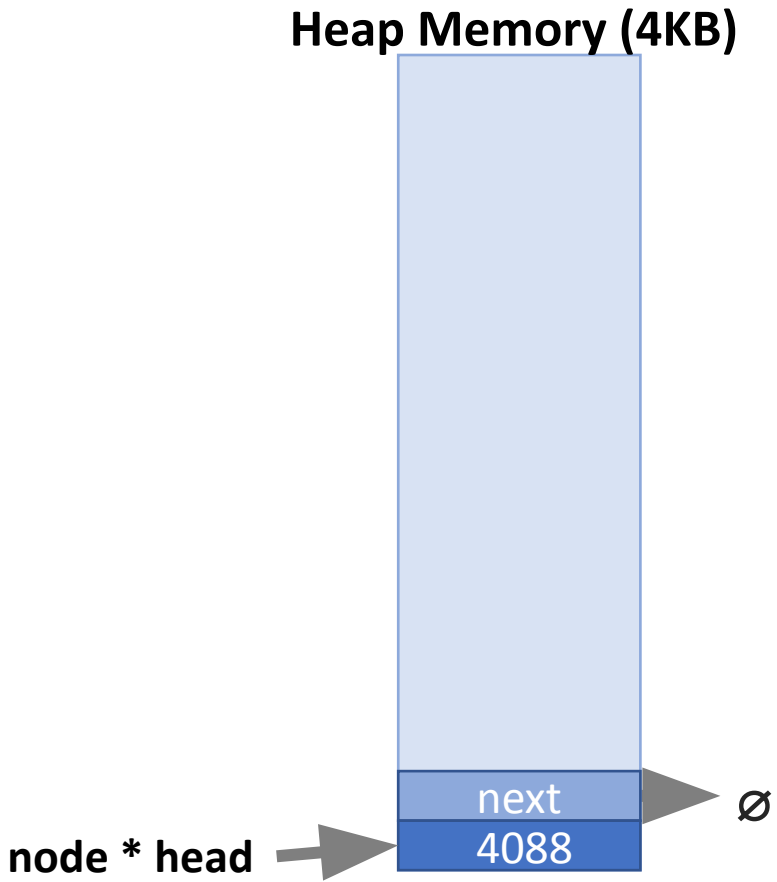
- Linked list of regions of free space
- size = bytes of free space

- Header for each block of allocated space
- size = bytes of allocated space



Allocating Memory (Splitting)

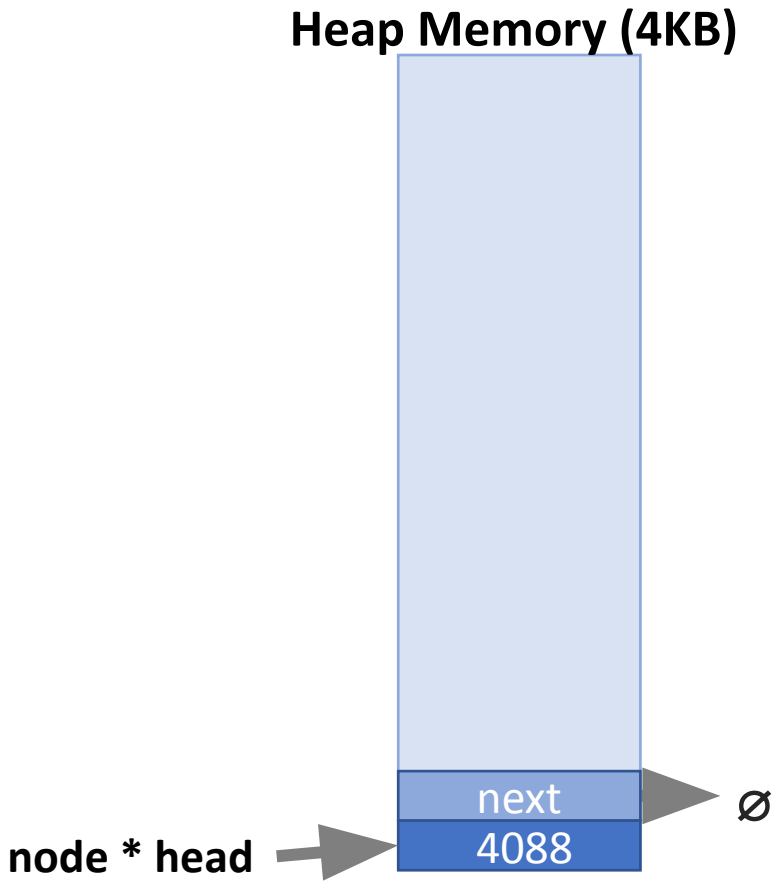
```
typedef struct node_t {
        int size;
        struct node_t * next;
} node;

typedef struct header_t {
        int size;
} header;
```



Allocating Memory (Splitting)

```
char * s1 = (char *) malloc(100); // 104 bytes
```

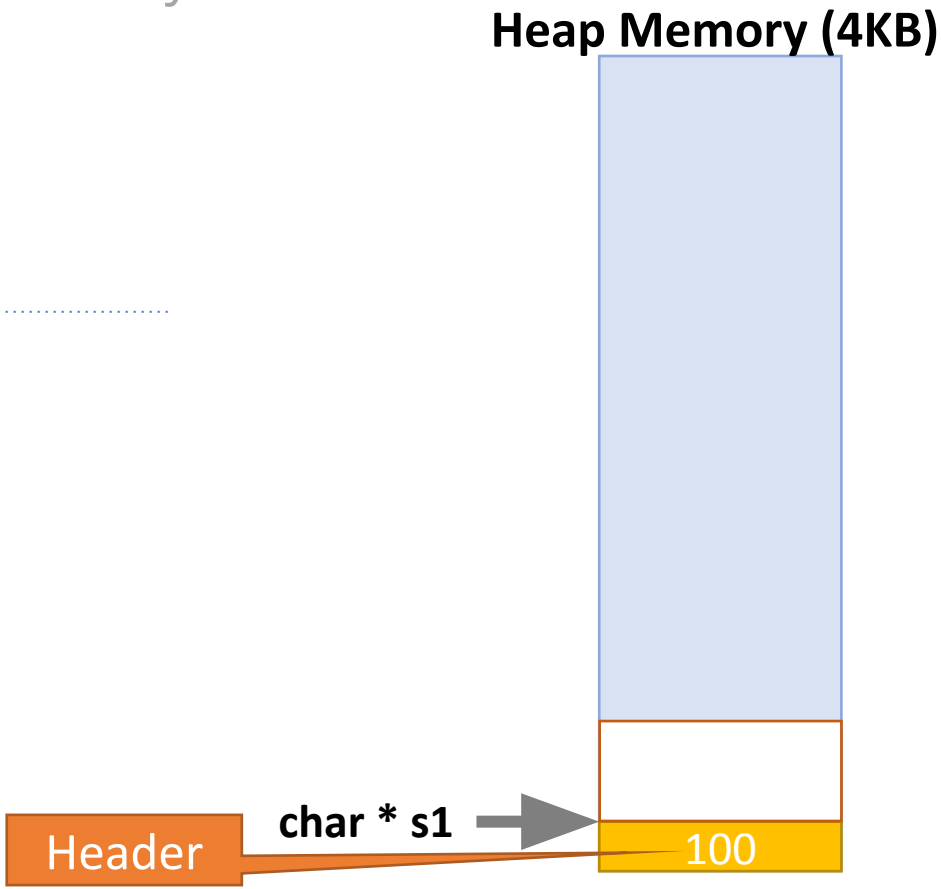
```
typedef struct node_t {
        int size;
        struct node_t * next;
} node;

typedef struct header_t {
        int size;
} header;
```



Allocating Memory (Splitting)

```
char * s1 = (char *) malloc(100); // 104 bytes
```

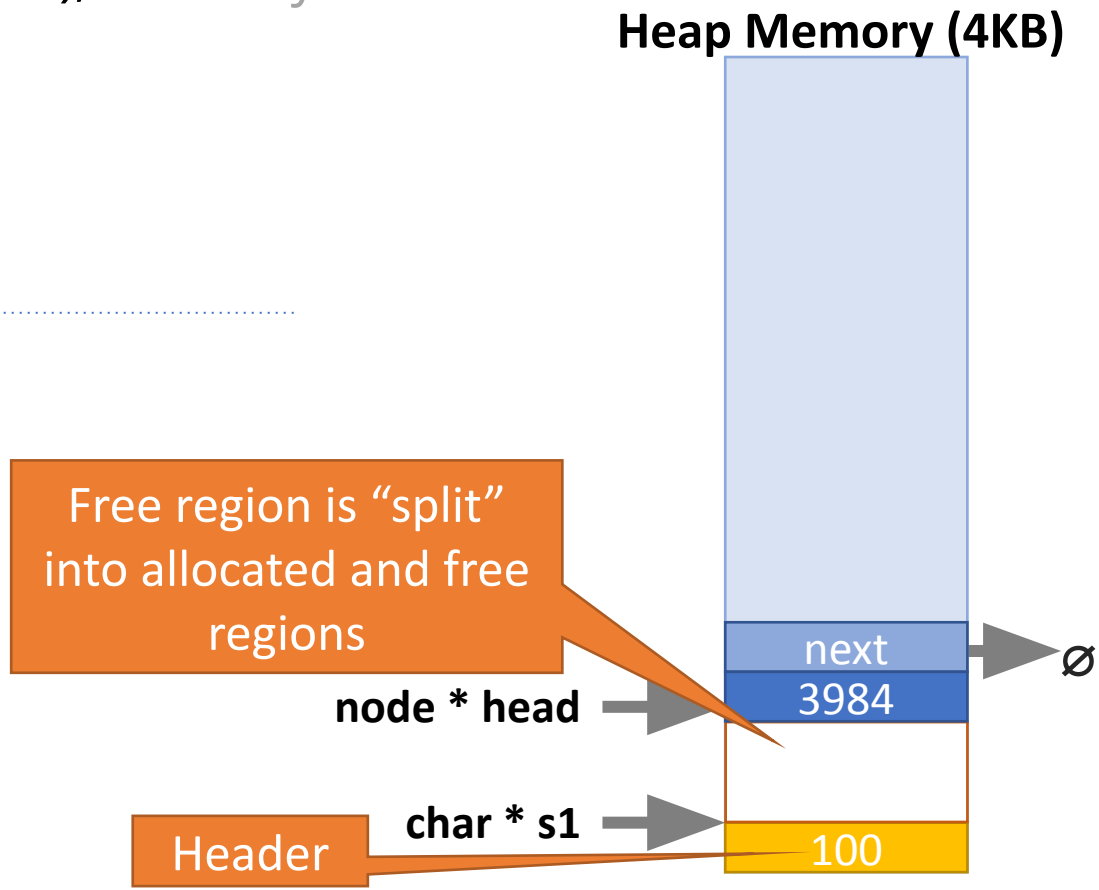
```
typedef struct node_t {
        int size;
        struct node_t * next;
} node;

typedef struct header_t {
        int size;
} header;
```



Allocating Memory (Splitting)

```
char * s1 = (char *) malloc(100); // 104 bytes
```

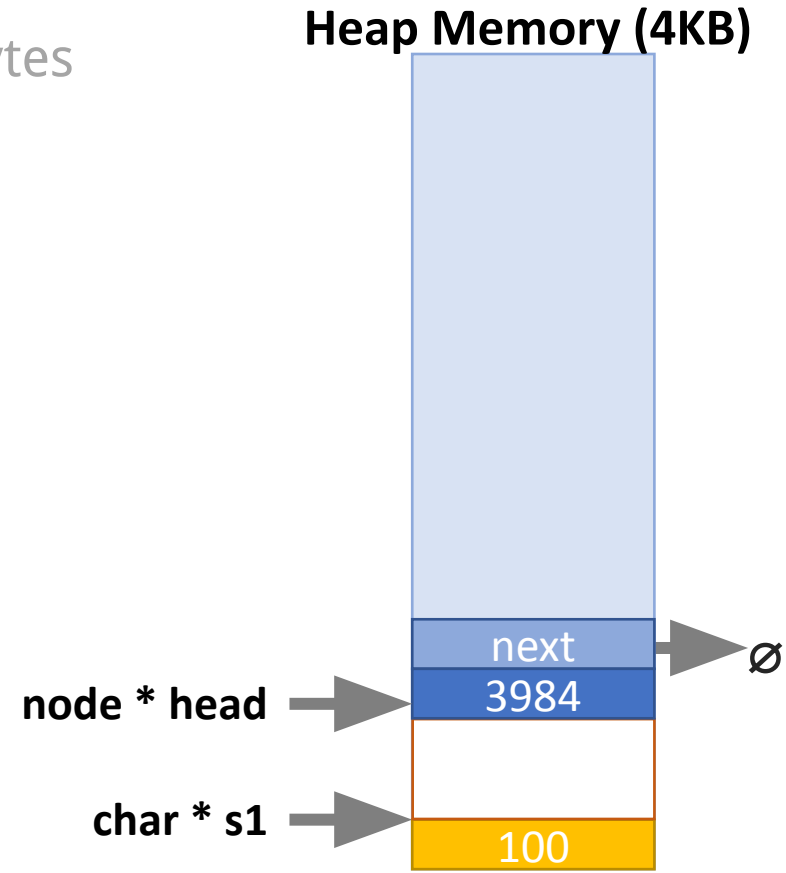
```
typedef struct node_t {
        int size;
        struct node_t * next;
} node;

typedef struct header_t {
        int size;
} header;
```



Allocating Memory (Splitting)

```
char * s1 = (char *) malloc(100); // 104 bytes
char * s2 = (char *) malloc(100); // 104 bytes
```

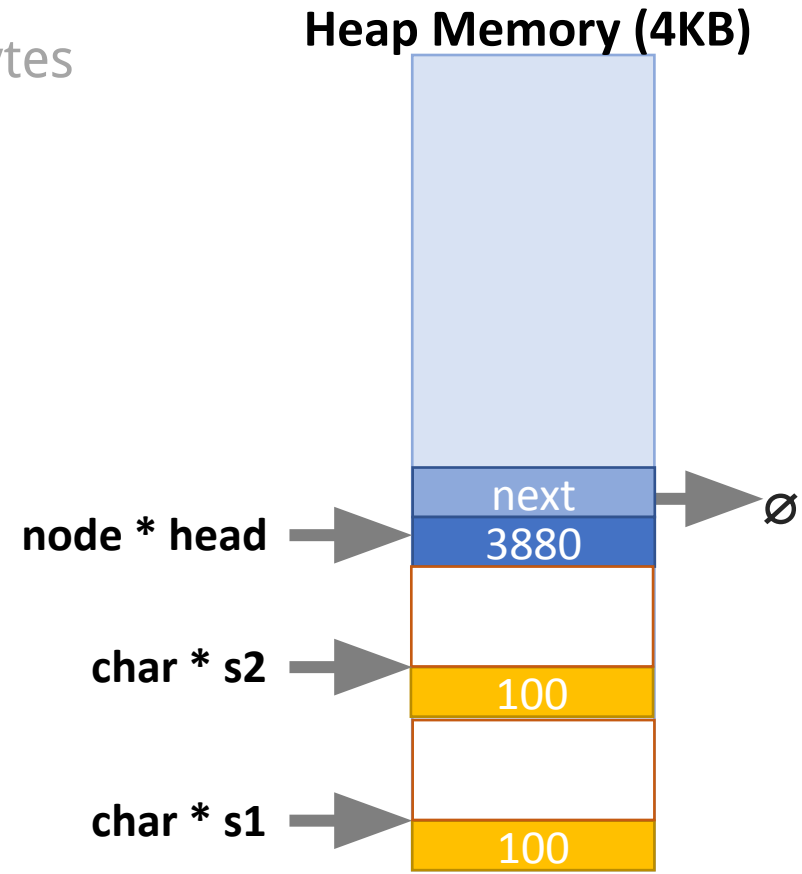
```
typedef struct node_t {
    int size;
    struct node_t * next;
} node;

typedef struct header_t {
    int size;
} header;
```



Allocating Memory (Splitting)

```
char * s1 = (char *) malloc(100); // 104 bytes
char * s2 = (char *) malloc(100); // 104 bytes
```

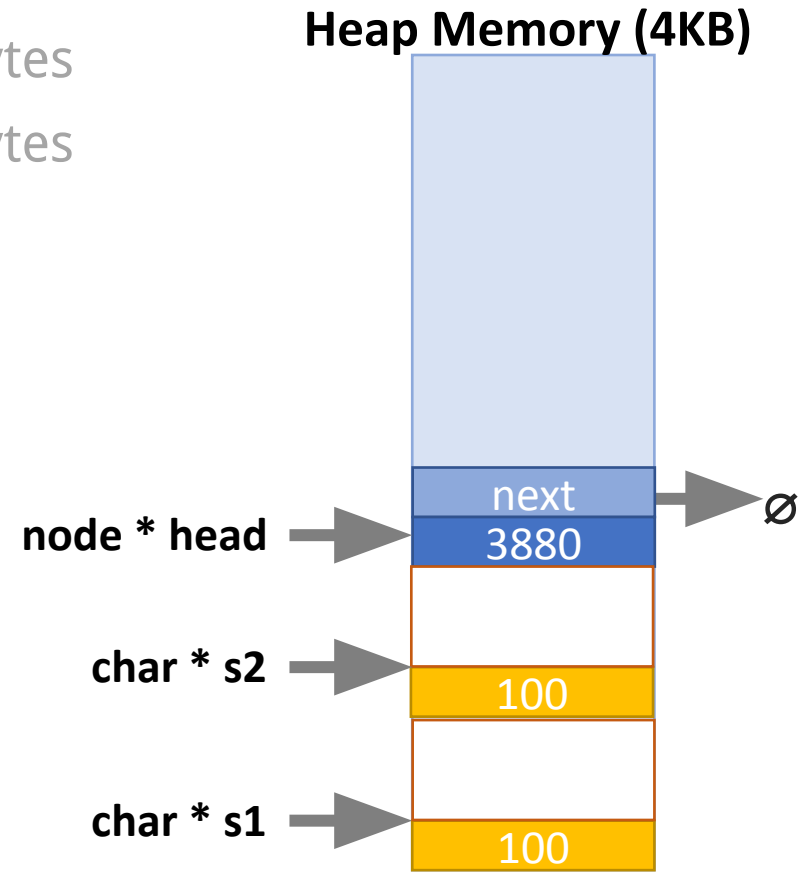
```
typedef struct node_t {
    int size;
    struct node_t * next;
} node;

typedef struct header_t {
    int size;
} header;
```



Allocating Memory (Splitting)

```
char * s1 = (char *) malloc(100); // 104 bytes
char * s2 = (char *) malloc(100); // 104 bytes
char * s3 = (char *) malloc(100); // 104 bytes
```

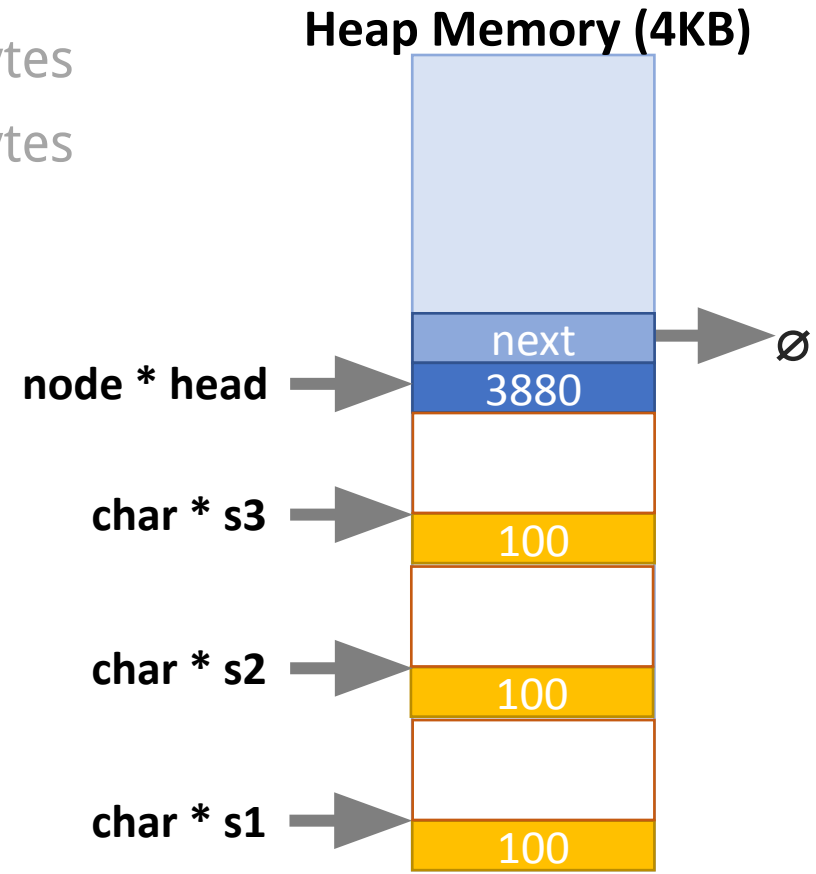
```
typedef struct node_t {
    int size;
    struct node_t * next;
} node;

typedef struct header_t {
    int size;
} header;
```



Allocating Memory (Splitting)

```
char * s1 = (char *) malloc(100); // 104 bytes
char * s2 = (char *) malloc(100); // 104 bytes
char * s3 = (char *) malloc(100); // 104 bytes
```

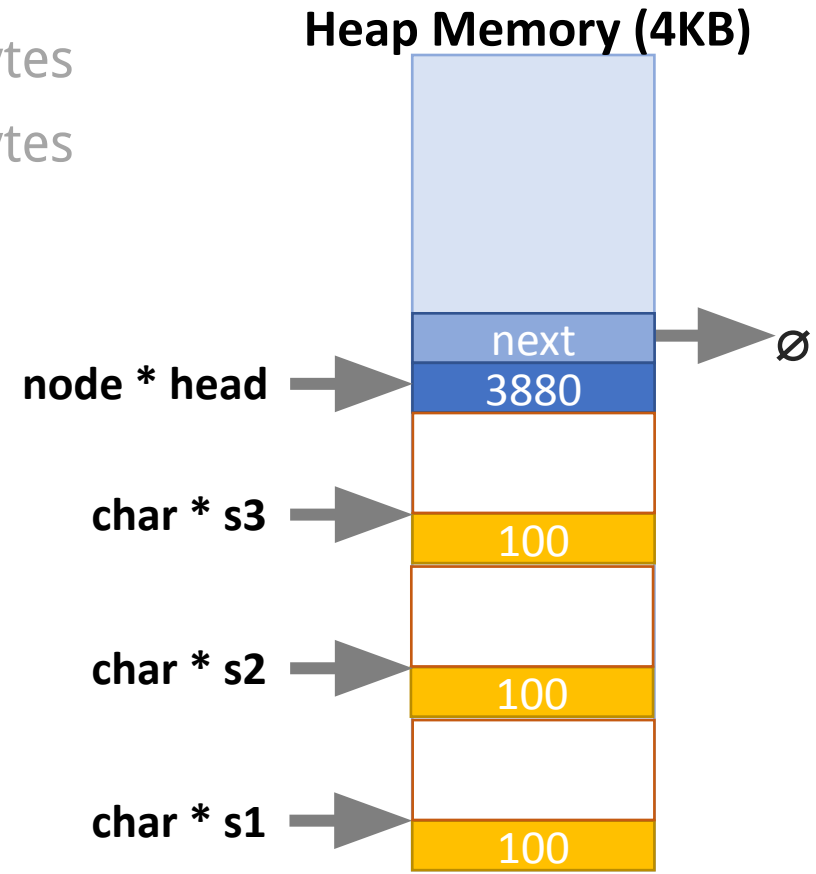
```
typedef struct node_t {
    int size;
    struct node_t * next;
} node;

typedef struct header_t {
    int size;
} header;
```



Allocating Memory (Splitting)

```
char * s1 = (char *) malloc(100); // 104 bytes
char * s2 = (char *) malloc(100); // 104 bytes
char * s3 = (char *) malloc(100); // 104 bytes
```

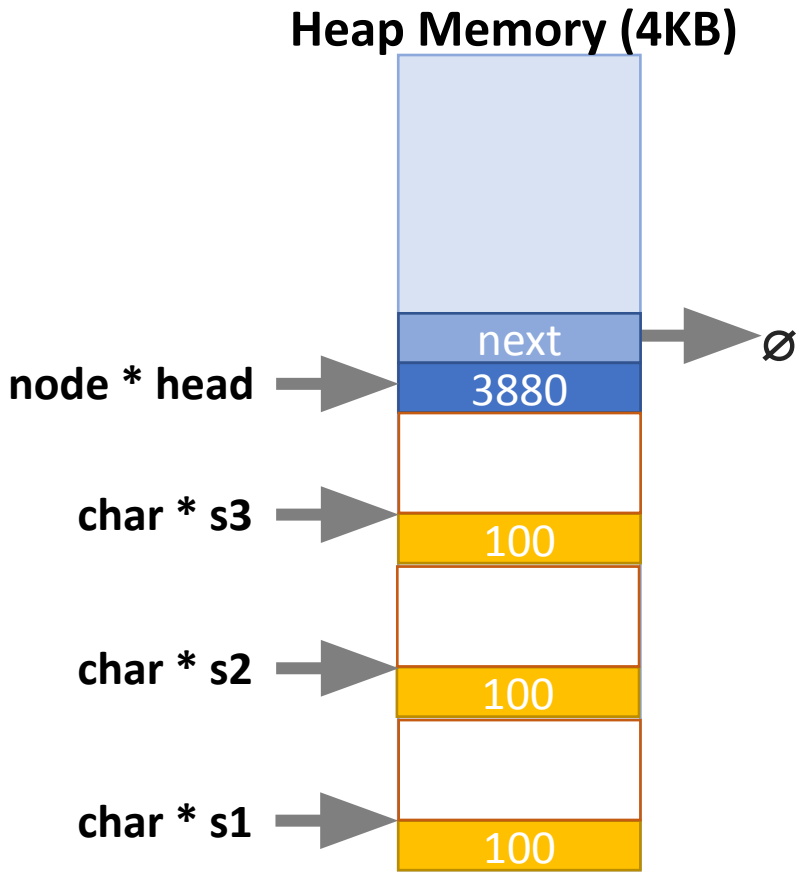
```
typedef struct node_t {
    int size;
    struct node_t * next;
} node;

typedef struct header_t {
    int size;
} header;
```



Freeing Memory

- The free list is kept in sorted order
  - free() is an O(n) operation

```
typedef struct node_t {
        int size;
        struct node_t * next;
} node;

typedef struct header_t {
        int size;
} header;
```

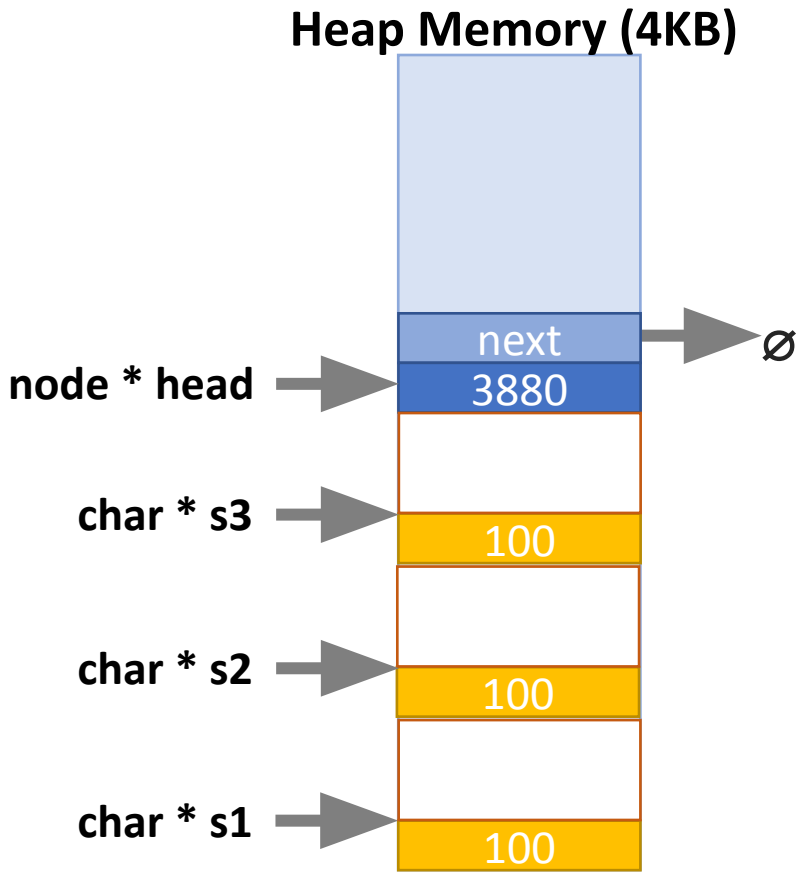


Freeing Memory

- The free list is kept in sorted order
  - free() is an O(n) operation

```
free(s2); // returns 100 + 4 – 8 bytes
```

```
typedef struct node_t {
        int size;
        struct node_t * next;
} node;

typedef struct header_t {
        int size;
} header;
```

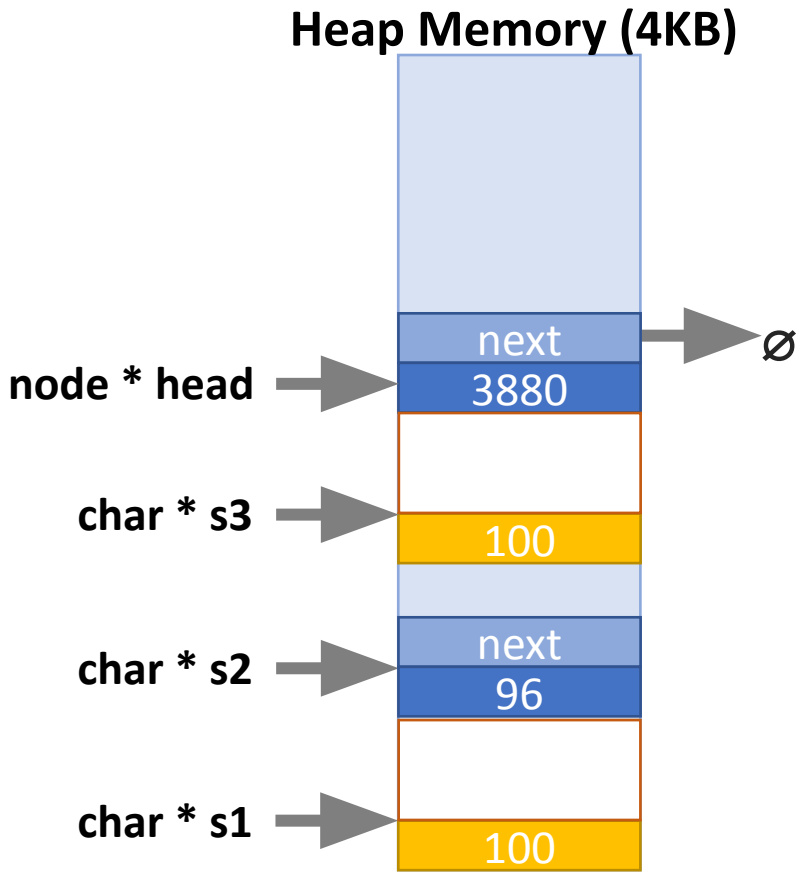


Freeing Memory

- The free list is kept in sorted order
  - free() is an O(n) operation

```
free(s2); // returns 100 + 4 – 8 bytes
```

```
typedef struct node_t {
        int size;
        struct node_t * next;
} node;

typedef struct header_t {
        int size;
} header;
```

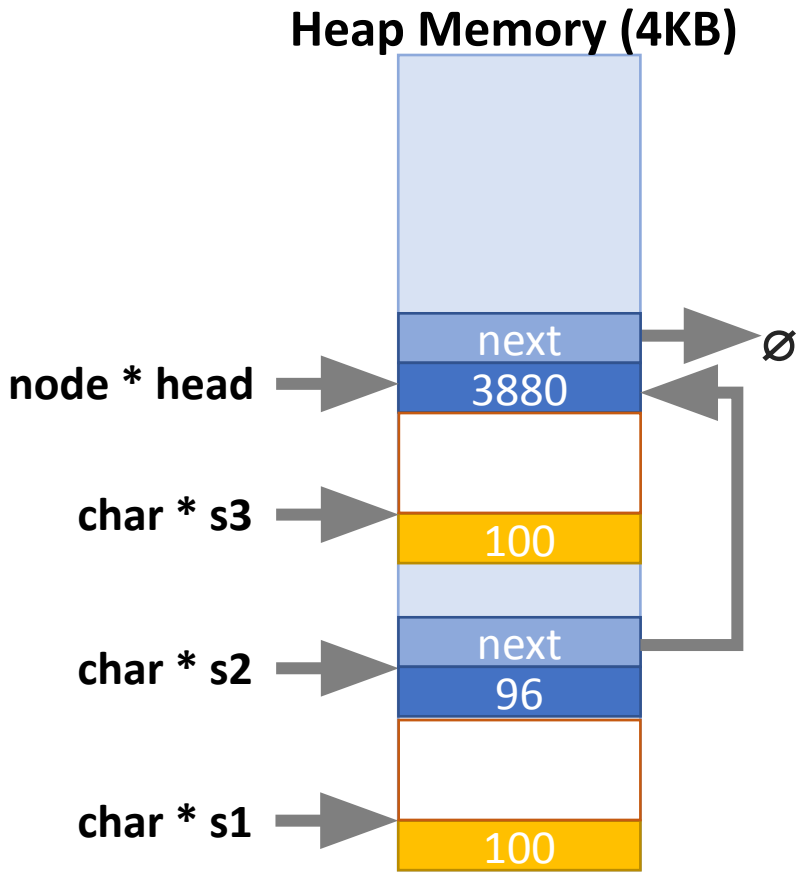


Freeing Memory

- The free list is kept in sorted order
  - free() is an O(n) operation

```
free(s2); // returns 100 + 4 – 8 bytes
```

```
typedef struct node_t {
        int size;
        struct node_t * next;
} node;

typedef struct header_t {
        int size;
} header;
```

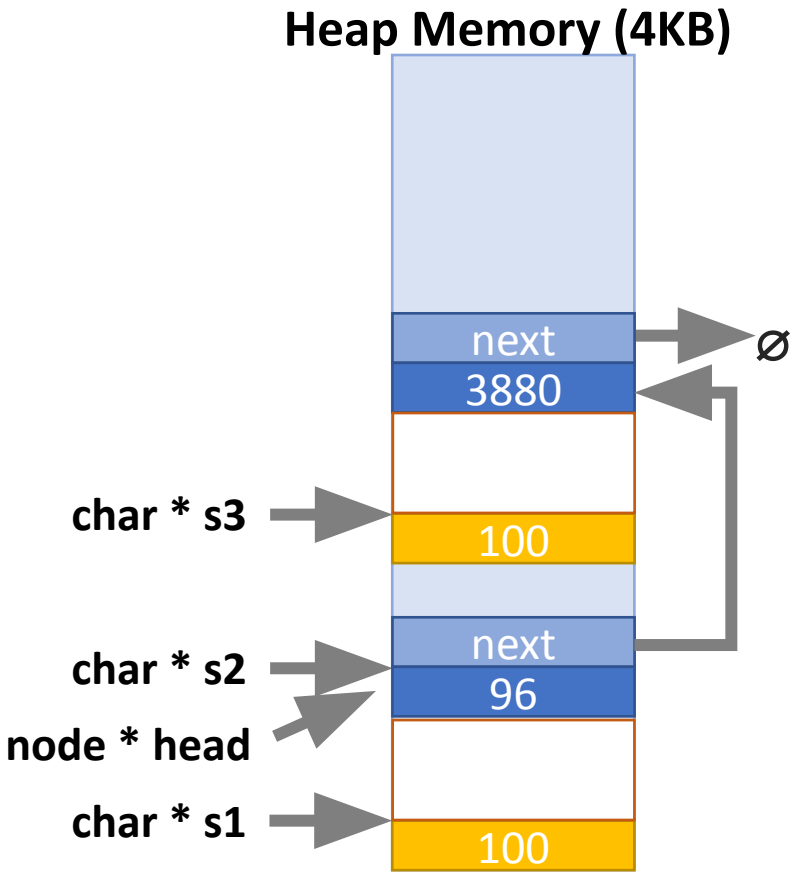


Freeing Memory

- The free list is kept in sorted order
  - free() is an O(n) operation

free(s2); // returns 100 + 4 – 8 bytes

```
typedef struct node_t {
        int size;
        struct node_t * next;
} node;

typedef struct header_t {
        int size;
} header;
```

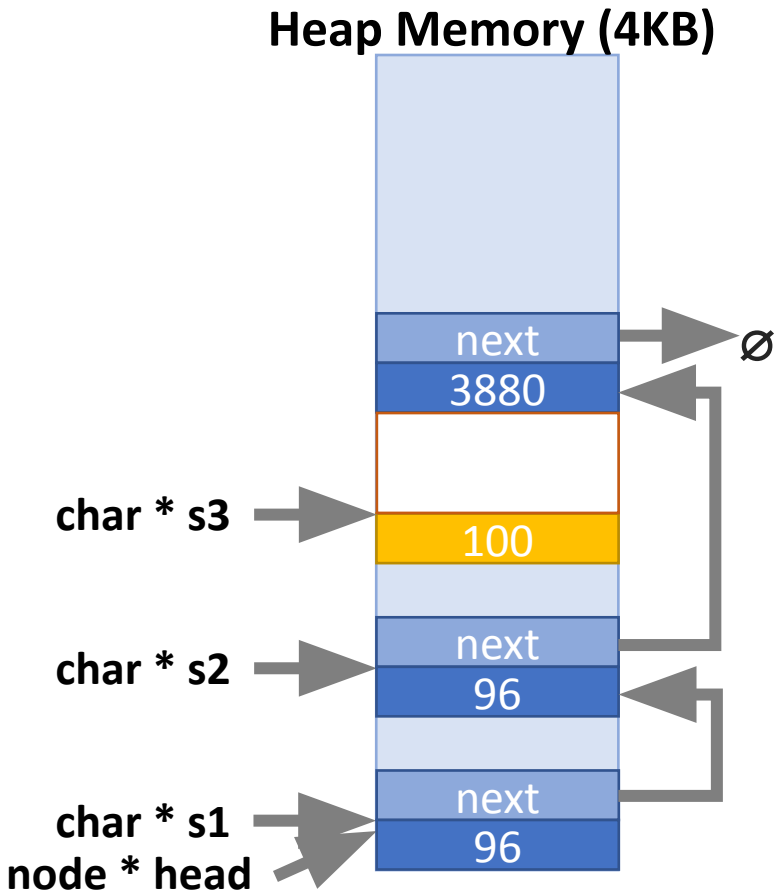


Freeing Memory

- The free list is kept in sorted order
  - free() is an O(n) operation

```
free(s2); // returns 100 + 4 – 8 bytes
free(s1); // returns 100 + 4 - 8 bytes
```

```
typedef struct node_t {
        int size;
        struct node_t * next;
} node;

typedef struct header_t {
        int size;
} header;
```

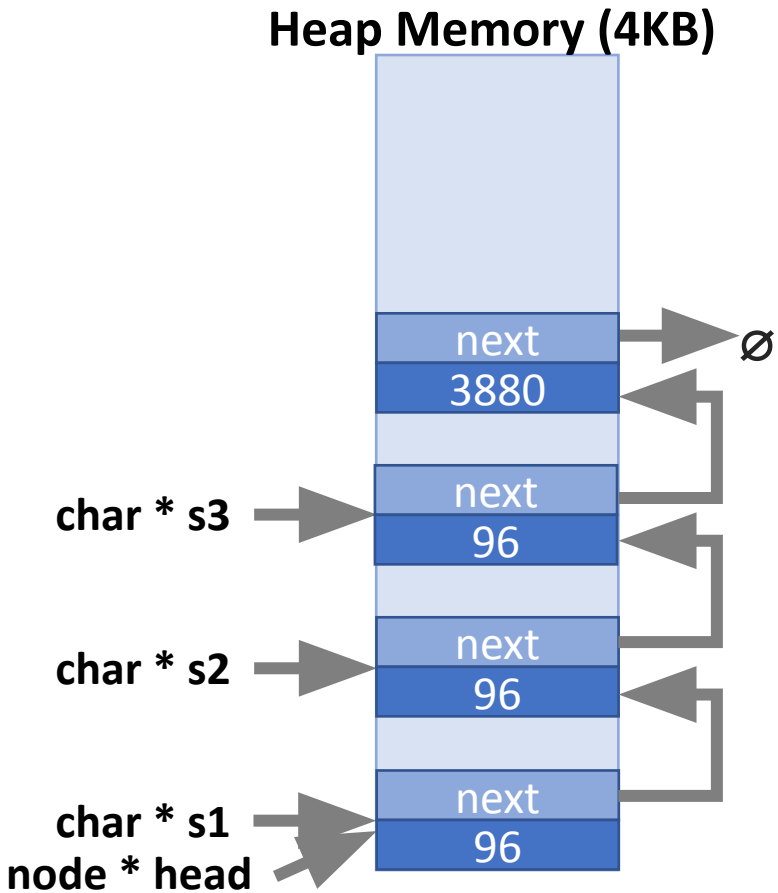


Freeing Memory

- The free list is kept in sorted order
  - free() is an O(n) operation

```
free(s2); // returns 100 + 4 – 8 bytes
free(s1); // returns 100 + 4 - 8 bytes
free(s3); // returns 100 + 4 - 8 bytes
```

```
typedef struct node_t {
        int size;
        struct node_t * next;
} node;

typedef struct header_t {
        int size;
} header;
```



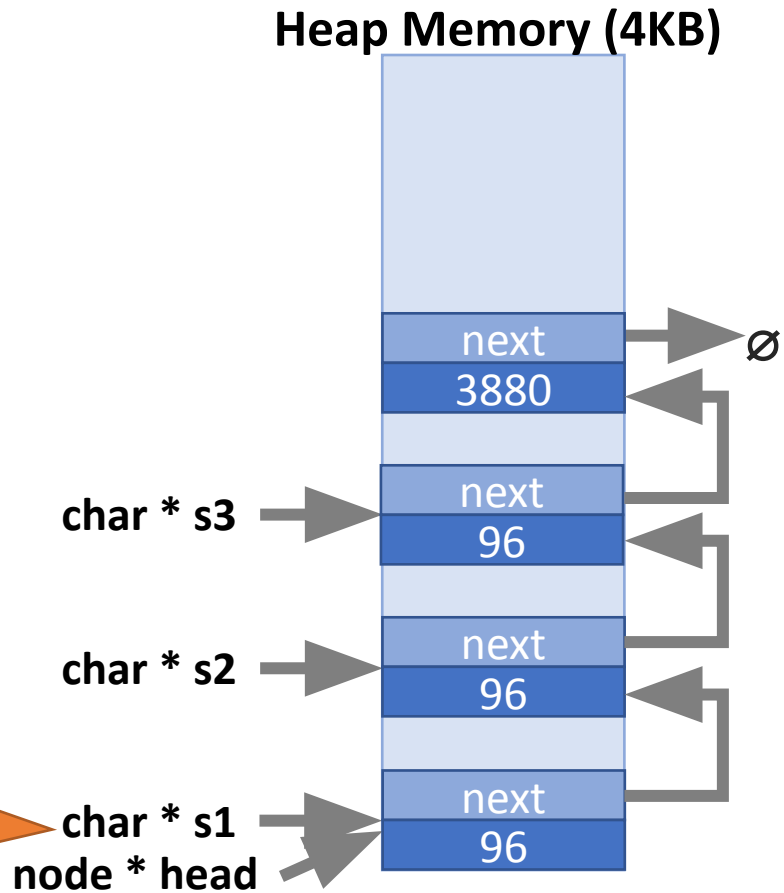
Freeing Memory

- The free list is kept in sorted order
  - free() is an O(n) operation

```
free(s2); // returns 100 + 4 – 8 bytes
free(s1); // returns 100 + 4 - 8 bytes
free(s3); // returns 100 + 4 - 8 bytes
```

```
typedef struct node_t {
        int size;
        struct node_t * next;
} node;

typedef struct h
        int size;
} header;
```

These pointers are “dangling”: they still point to heap memory, but the pointers are invalid



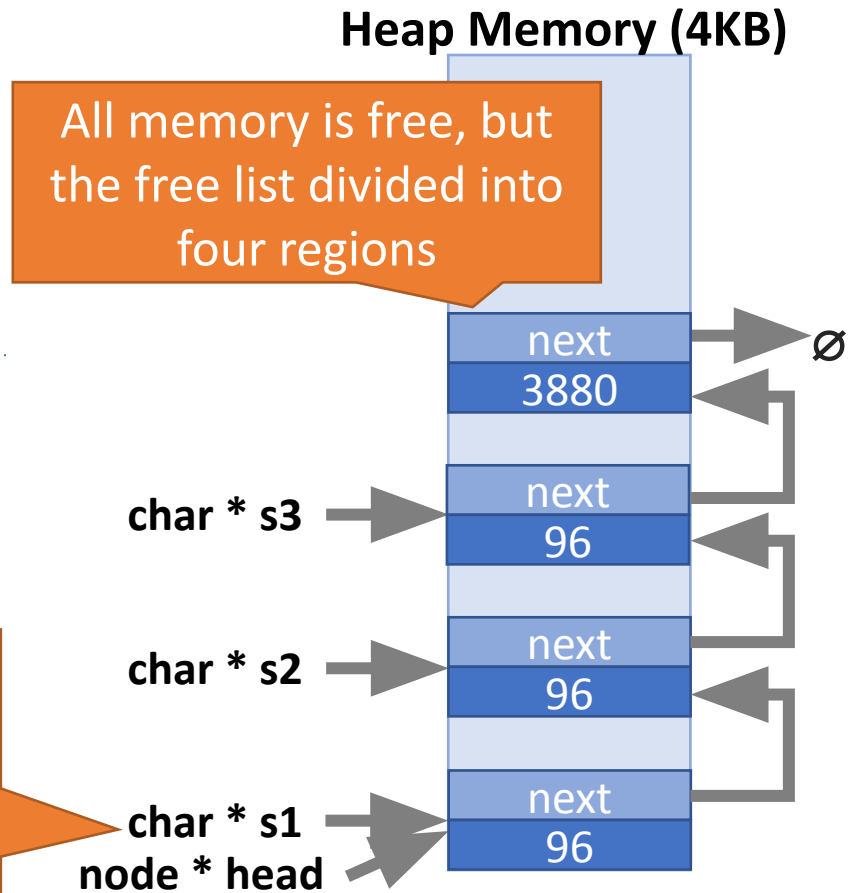
Freeing Memory

- The free list is kept in sorted order
  - free() is an O(n) operation

```
free(s2); // returns 100 + 4 – 8 bytes
free(s1); // returns 100 + 4 - 8 bytes
free(s3); // returns 100 + 4 - 8 bytes
```

```
typedef struct node_t {
        int size;
        struct node_t * next;
} node;

typedef struct h
        int size;
} header;
```

These pointers are "dangling": they still point to heap memory, but the pointers are invalid



Freeing Memory

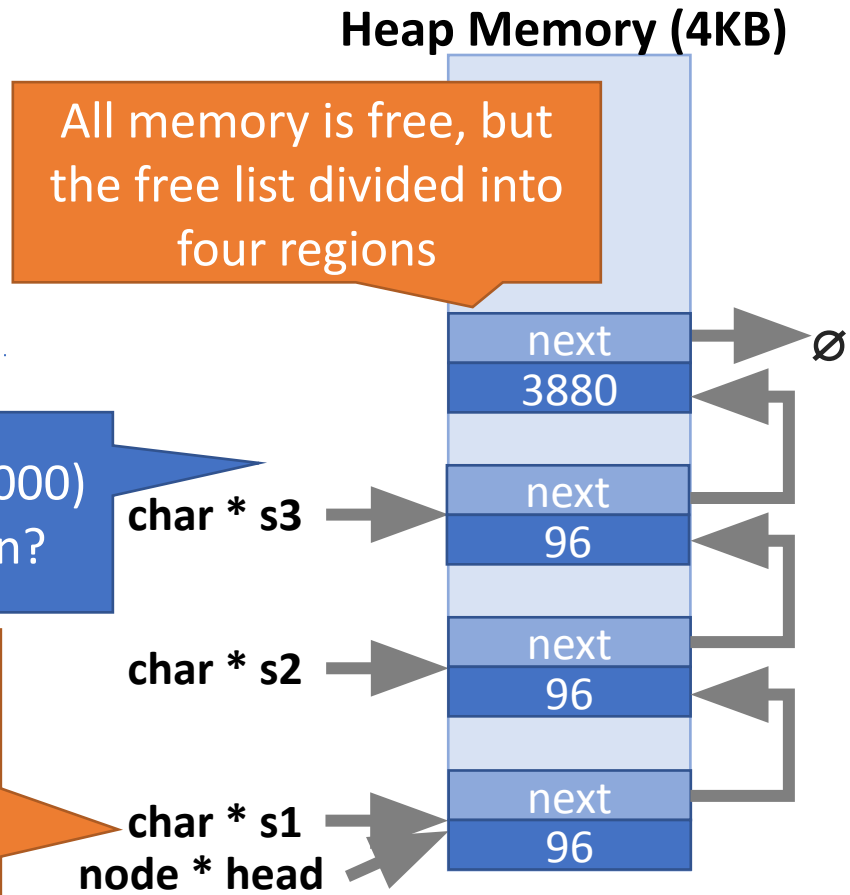
- The free list is kept in sorted order
  - free() is an O(n) operation

```
free(s2); // returns 100 + 4 – 8 bytes
free(s1); // returns 100 + 4 - 8 bytes
free(s3); // returns 100 + 4 - 8 bytes
```

```
typedef struct node_t {
        int size;
        struct node_t * nex
} node;

typedef struct h
        int size;
} header;
```

If user calls malloc(4000) what would happen?

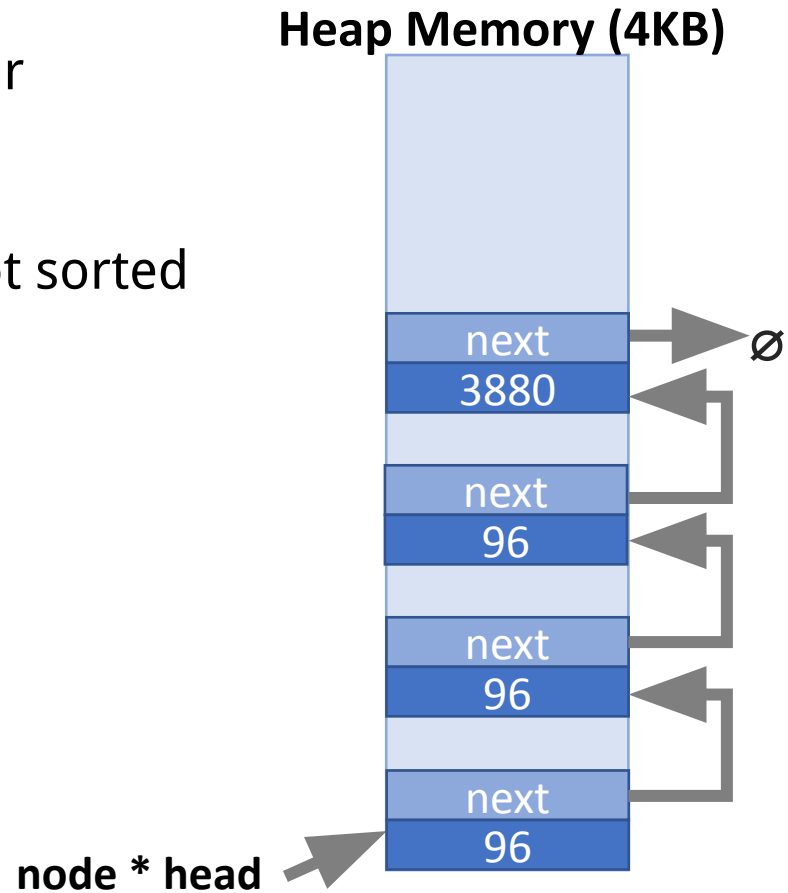
These pointers are "dangling": they still point to heap memory, but the pointers are invalid



Coalescing

- Free regions should be merged with their neighbors
  - Helps to minimize fragmentation
  - This would be $O(n^2)$ if the list was not sorted

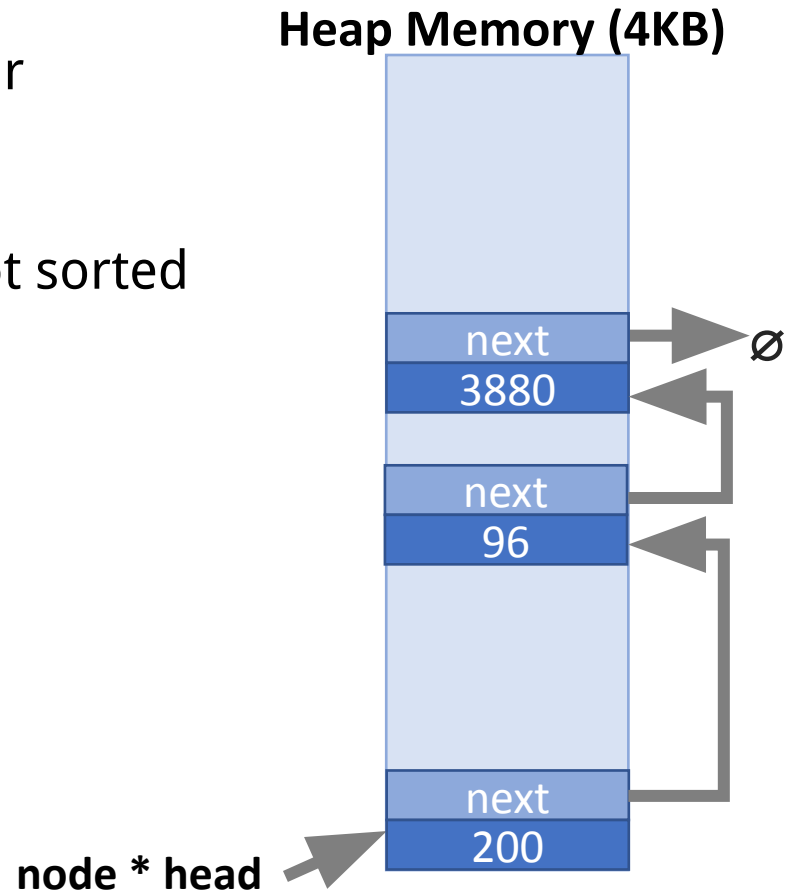
```
typedef struct node_t {
        int size;
        struct node_t * next;
} node;

typedef struct header_t {
        int size;
} header;
```



Coalescing

- Free regions should be merged with their neighbors
  - Helps to minimize fragmentation
  - This would be $O(n^2)$ if the list was not sorted

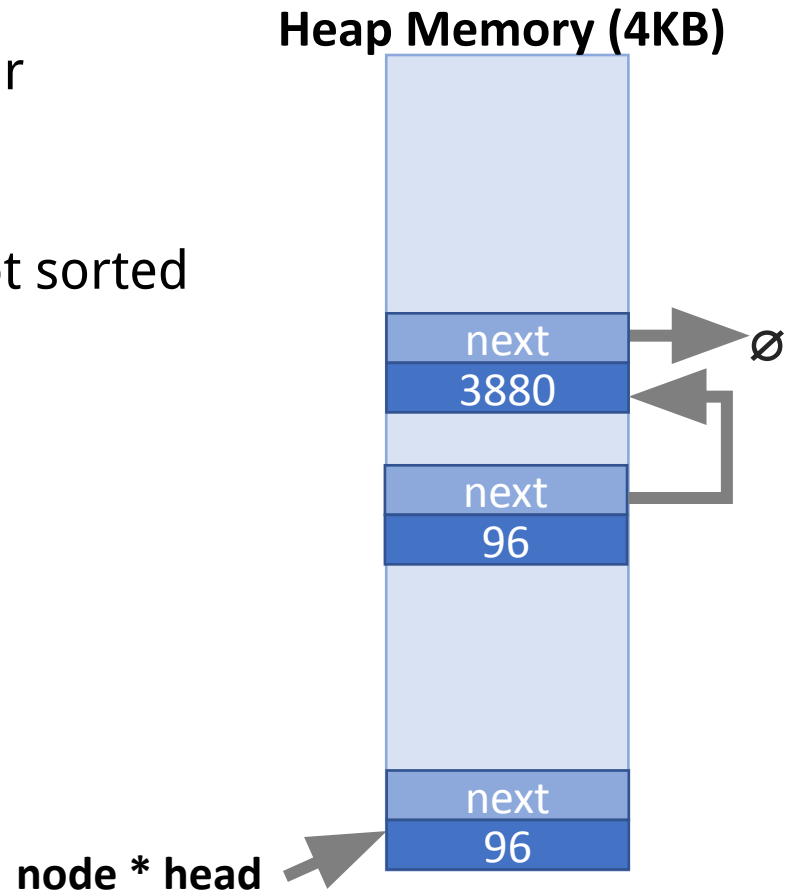
```
typedef struct node_t {
        int size;
        struct node_t * next;
} node;

typedef struct header_t {
        int size;
} header;
```



Coalescing

- Free regions should be merged with their neighbors
  - Helps to minimize fragmentation
  - This would be $O(n^2)$ if the list was not sorted

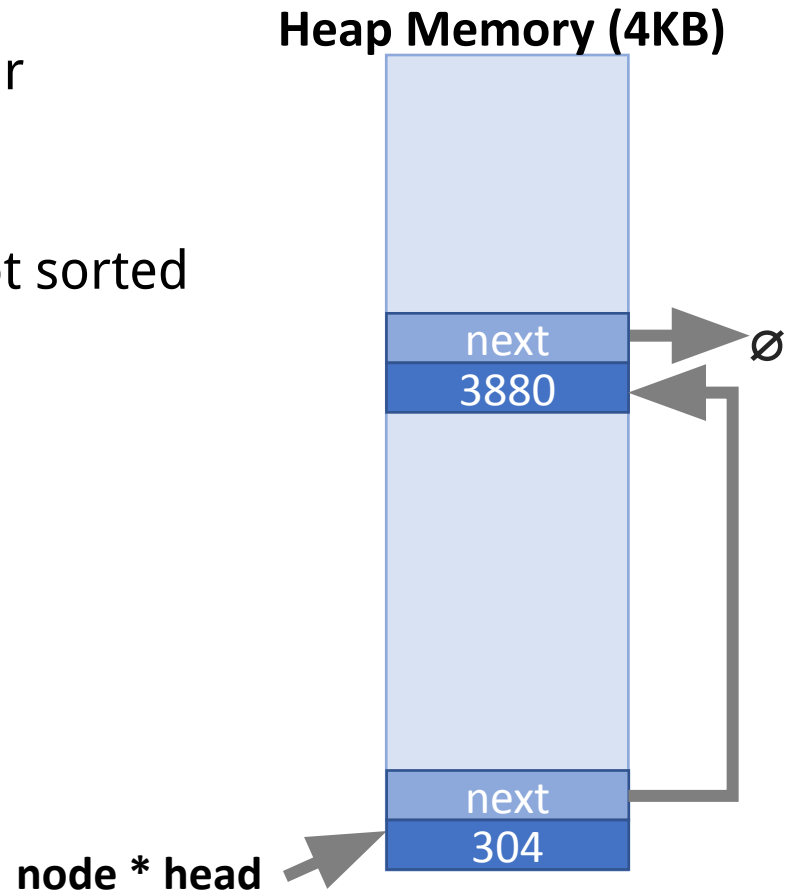
```
typedef struct node_t {
        int size;
        struct node_t * next;
} node;

typedef struct header_t {
        int size;
} header;
```



Coalescing

- Free regions should be merged with their neighbors
  - Helps to minimize fragmentation
  - This would be $O(n^2)$ if the list was not sorted

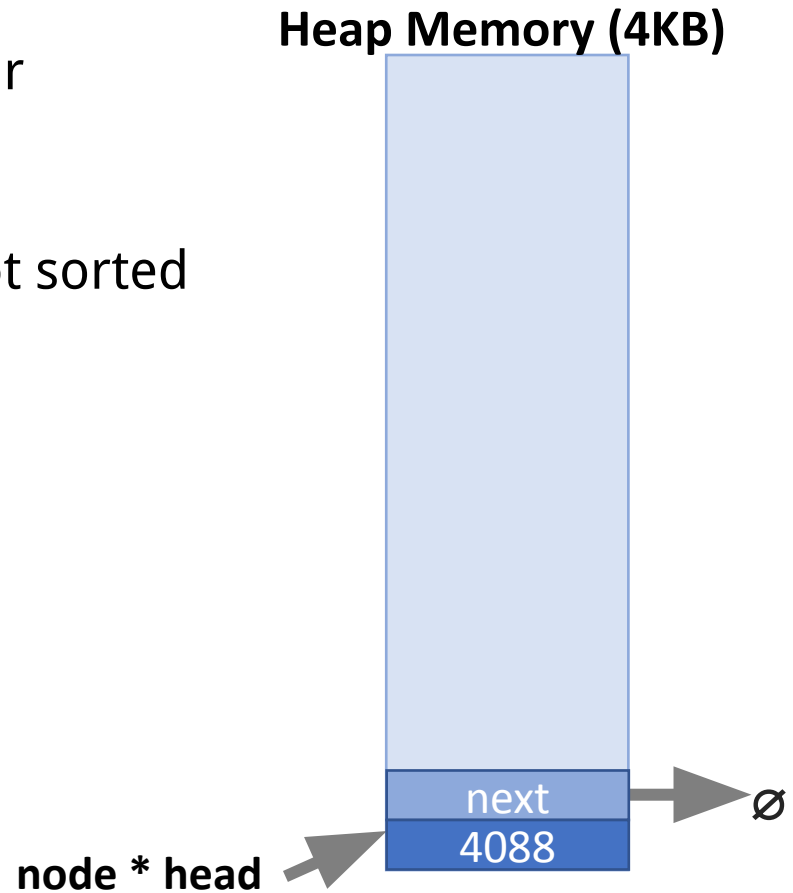
```
typedef struct node_t {
        int size;
        struct node_t * next;
} node;

typedef struct header_t {
        int size;
} header;
```



Coalescing

- Free regions should be merged with their neighbors
  - Helps to minimize fragmentation
  - This would be $O(n^2)$ if the list was not sorted

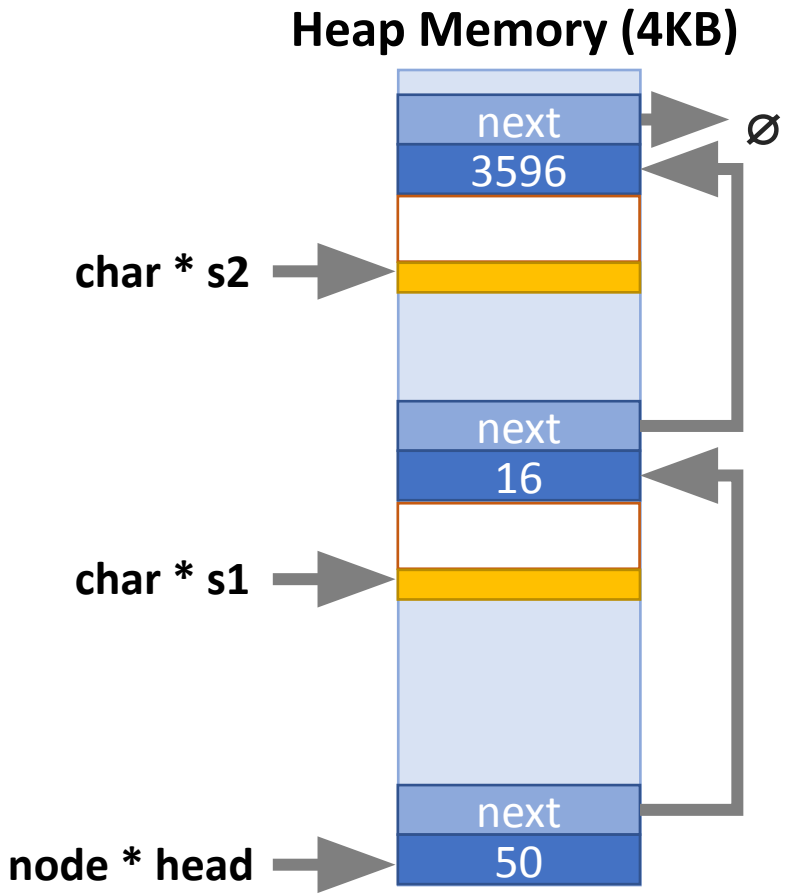
```
typedef struct node_t {
        int size;
        struct node_t * next;
} node;

typedef struct header_t {
        int size;
} header;
```



Choosing Free Regions (1)

```
int i[] = (int*) malloc(8);
// 8 + 4 = 12 total bytes
```

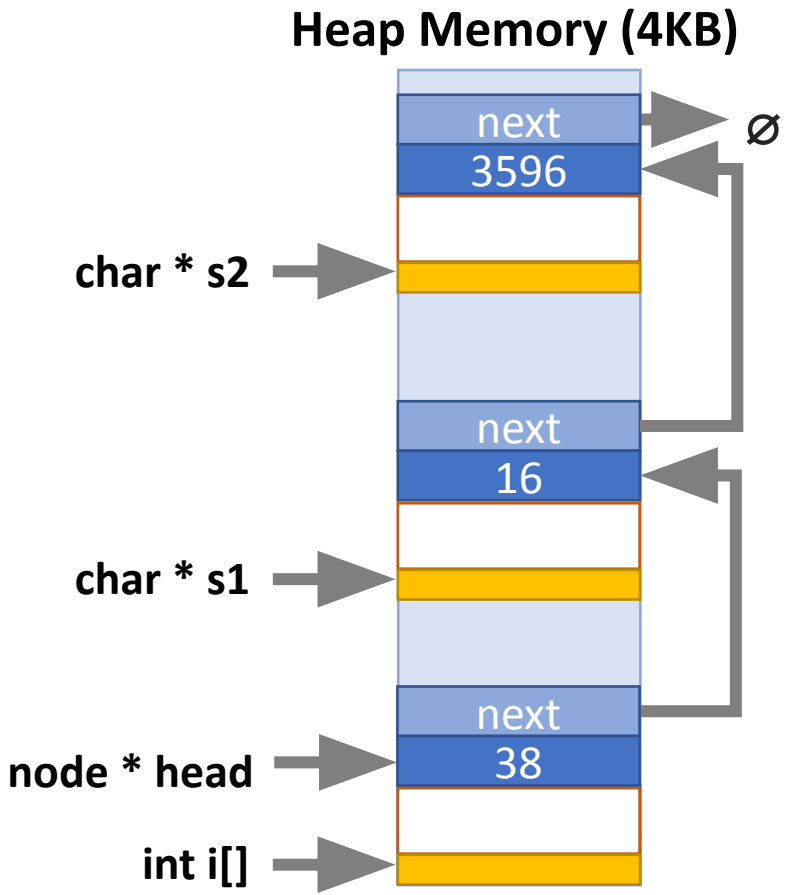
- Which free region should be chosen?



Choosing Free Regions (1)

```
int i[] = (int*) malloc(8);
// 8 + 4 = 12 total bytes
```

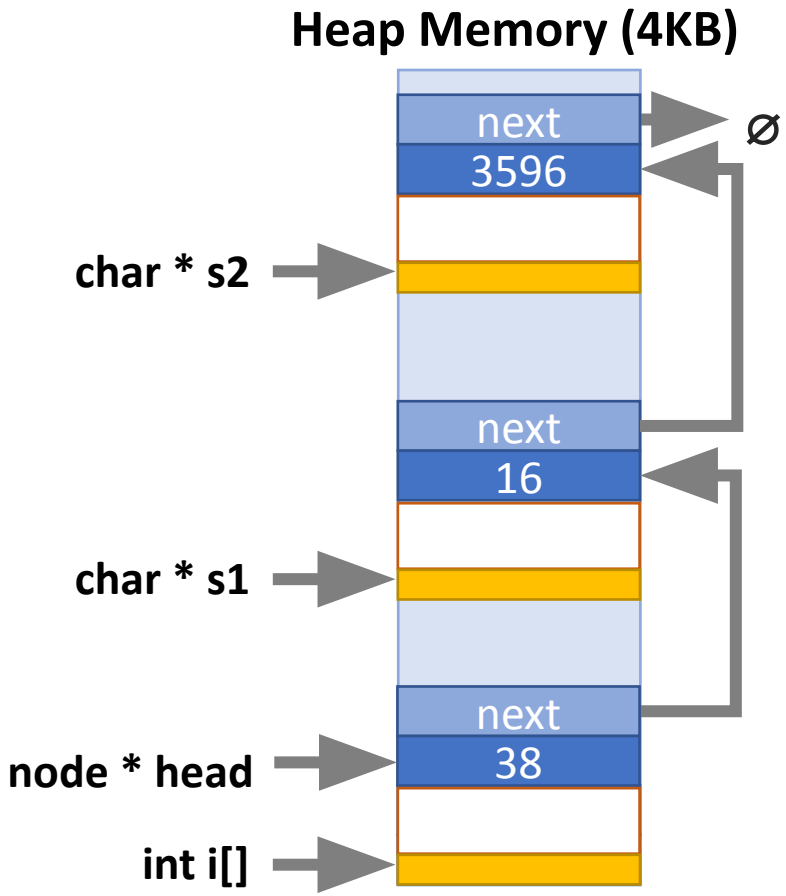
- Which free region should be chosen?
- Fastest option is **First-Fit**
  - Split the first free region with >=8 bytes available



Choosing Free Regions (1)

```
int i[] = (int*) malloc(8);
// 8 + 4 = 12 total bytes
```

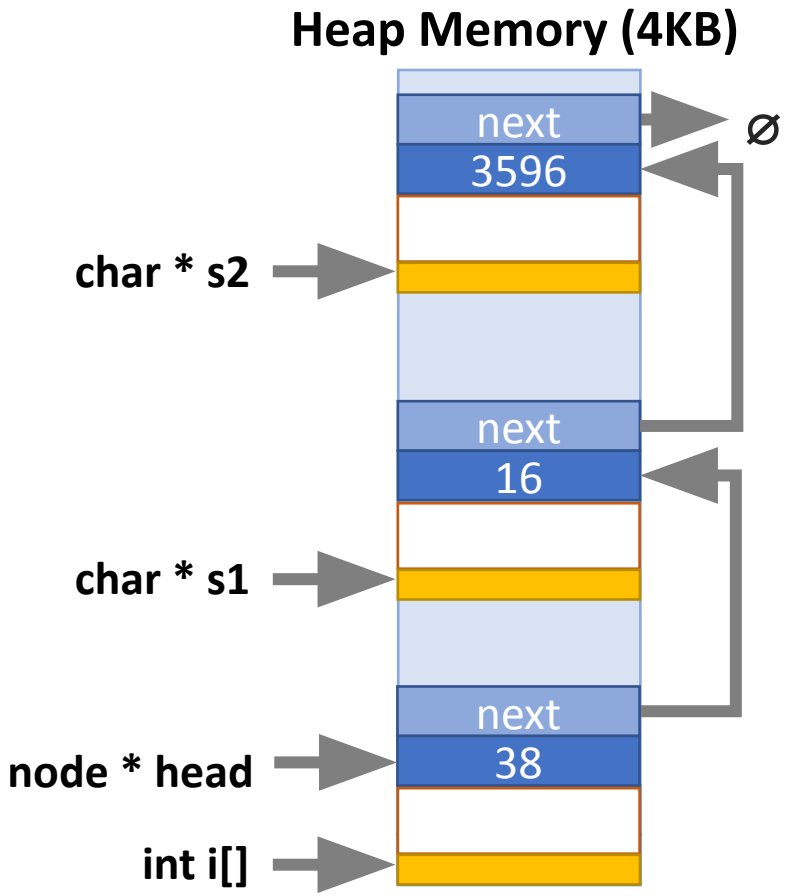
- Which free region should be chosen?
- Fastest option is **First-Fit**
  - Split the first free region with >=8 bytes available
- Problem with First-Fit?
  -



Choosing Free Regions (1)

```
int i[] = (int*) malloc(8);
// 8 + 4 = 12 total bytes
```

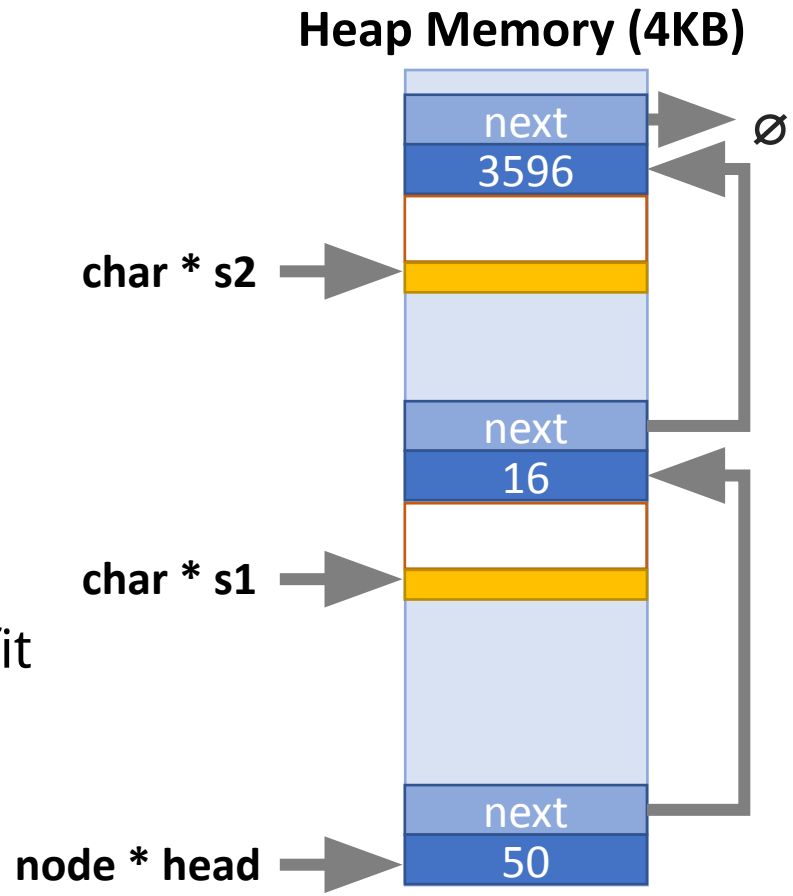
- Which free region should be chosen?
- Fastest option is **First-Fit**
  - Split the first free region with >=8 bytes available
- Problem with First-Fit?
  - Leads to external fragmentation



Choosing Free Regions (2)

```
int i[] = (int*) malloc(8);
// 8 + 4 = 12 total bytes
```

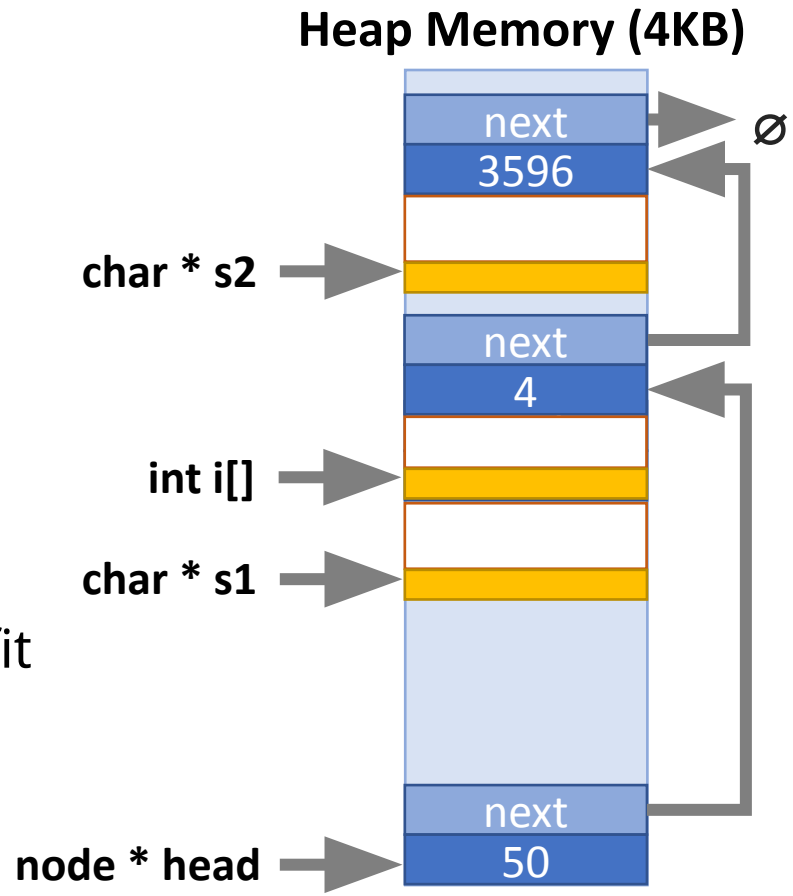
- Which free region should be chosen?
- Second option: **Best-Fit**
  - Locate the free region with size closest to (and >=) 8 bytes
- Less external fragmentation than First-fit



Choosing Free Regions (2)

```
int i[] = (int*) malloc(8);
// 8 + 4 = 12 total bytes
```

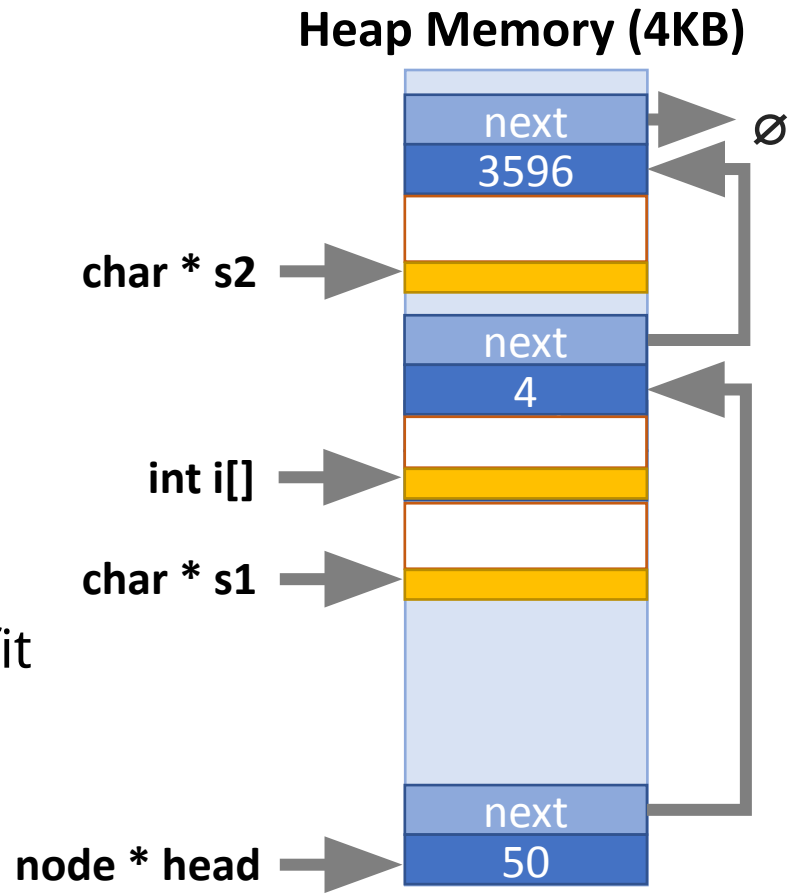
- Which free region should be chosen?
- Second option: **Best-Fit**
  - Locate the free region with size closest to (and >=) 8 bytes
- Less external fragmentation than First-fit



Choosing Free Regions (2)

```
int i[] = (int*) malloc(8);
// 8 + 4 = 12 total bytes
```

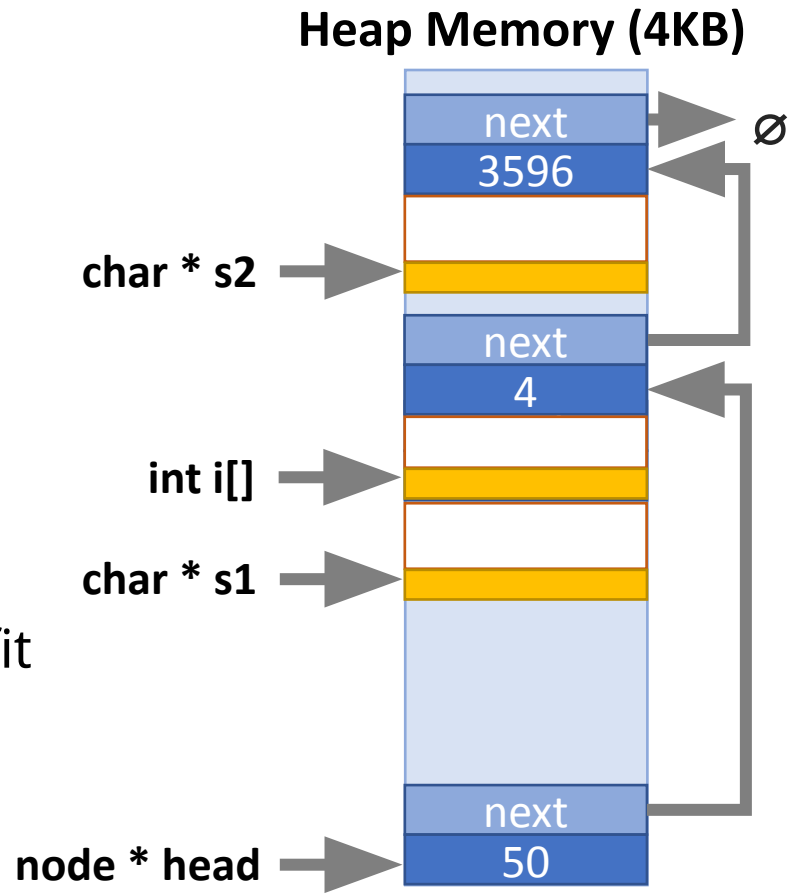
- Which free region should be chosen?
- Second option: **Best-Fit**
  - Locate the free region with size closest to (and >=) 8 bytes
- Less external fragmentation than First-fit
- Problem with Best-Fit?



Choosing Free Regions (2)

```
int i[] = (int*) malloc(8);
// 8 + 4 = 12 total bytes
```

- Which free region should be chosen?
- Second option: **Best-Fit**
  - Locate the free region with size closest to (and >=) 8 bytes
- Less external fragmentation than First-fit
- Problem with Best-Fit?
  - Requires O(n) time



Basic Free List Review

- Singly-linked free list

- List is kept in sorted order
  - *free()* is an O(n) operation
  - Adjacent free regions are coalesced

- Various strategies for selecting which free region to use
  - First-fit: use the first free region with >=*n* bytes available
    - Worst-case is O(n), but typically much faster
    - Tends to lead to external fragmentation at the head of the list
  - Best-fit: use the region with size closest (and >=) to *n*
    - Less external fragments than first-fit, but O(n) time

Some clarification for the assignment

- You do not have to know about 4KB granularity allocation happening behind the scene
- You may treat as if sbrk(X) newly allocates X bytes of memory block
- You do not have to split the memory block when reusing it for allocation
- You do not have to merge the memory block during freeing
- You simply need to treat the allocated block from sbrk call as a single memory block and reuse the blocks as they are

Concurrency

Concurrent thinking

- Humans tend to think sequentially
- Thinking about all the potential sequences of events is difficult for humans.
  - https://www.psychologicalscience.org/news/why-humans-are-bad-at-multitasking.html
- Computers on the other hand, can multi-task quite well.

Parallelism vs Concurrency (programming context)

- What are parallelism and concurrency?
- What is the difference?

Parallelism vs Concurrency (programming context)

- Concurrency:

  Happening at at the same time, interleaving, sharing resources
  - Multiple tasks **in progress** at the same time
  - **Dealing** with multiple things at once

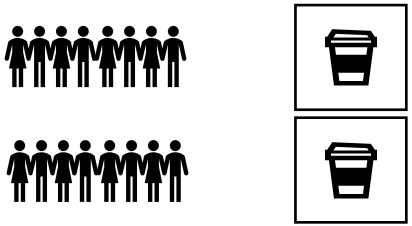
- Parallelism:

  Happening at the same time, progressing independently
  - Multiple tasks **executing** at the same time
  - **Doing** multiple things at once
  - Simultaneous execution

Parallelism vs Concurrency (programming context)

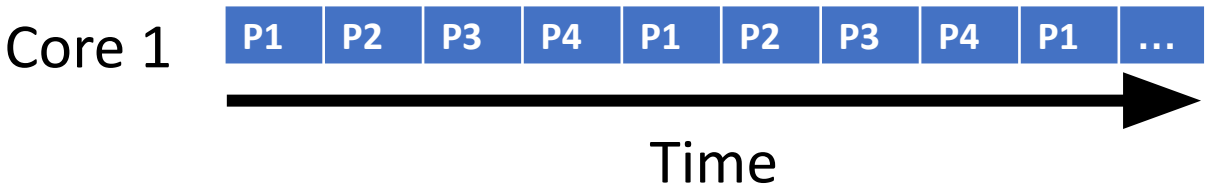
- Concurrency
  - Two queues for one vending machine

- Parallelism
  - Two queues for two vending machines

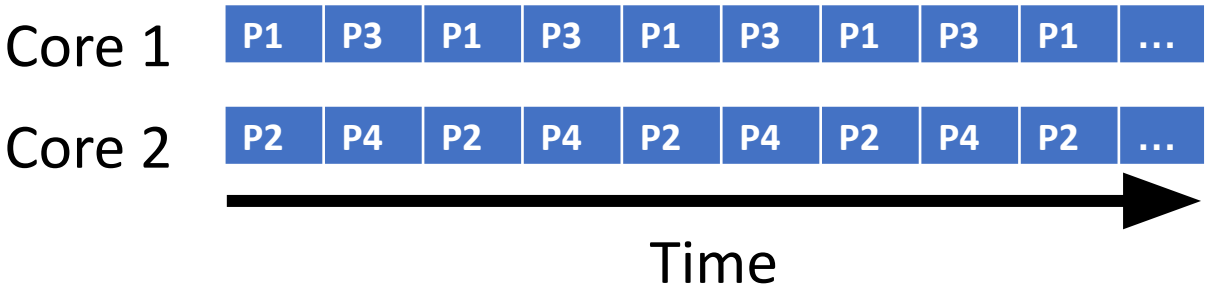


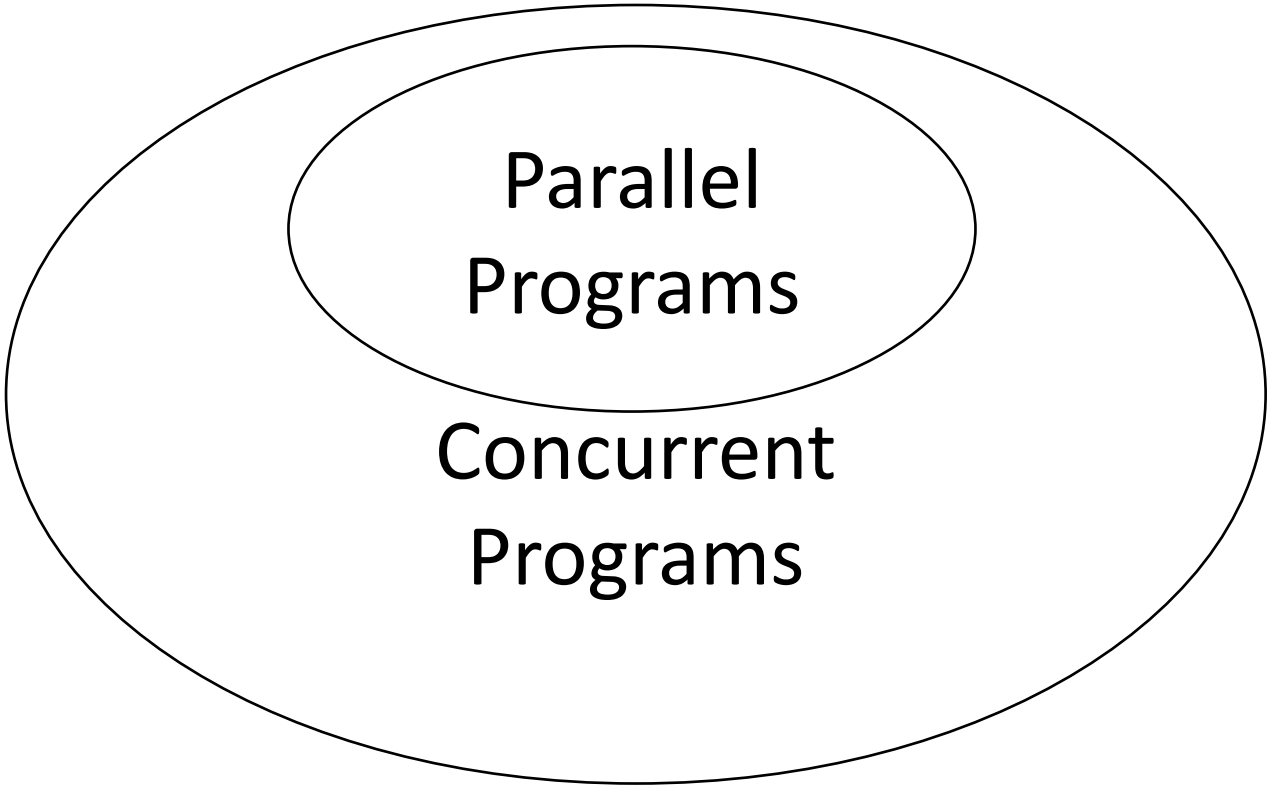
Parallelism vs Concurrency (programming context)

- Concurrent execution on a single-core system:



- Parallel execution on a dual-core system:



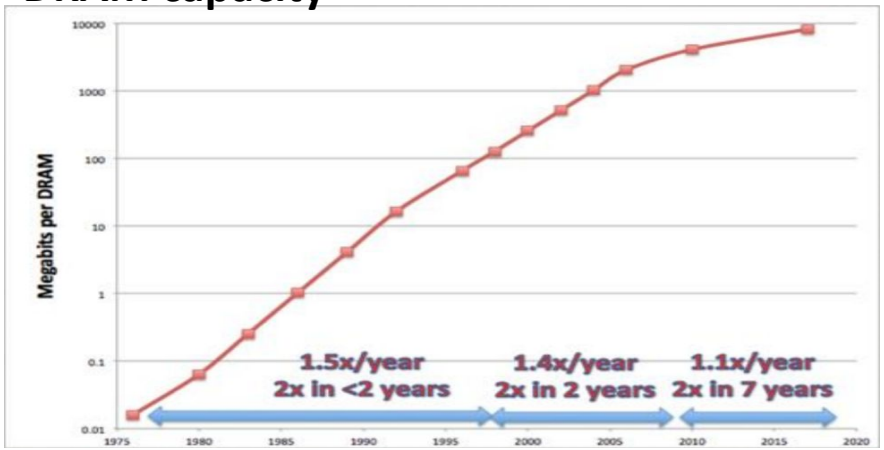

Parallel
Programs
Concurrent
Programs

Why is concurrency so important?

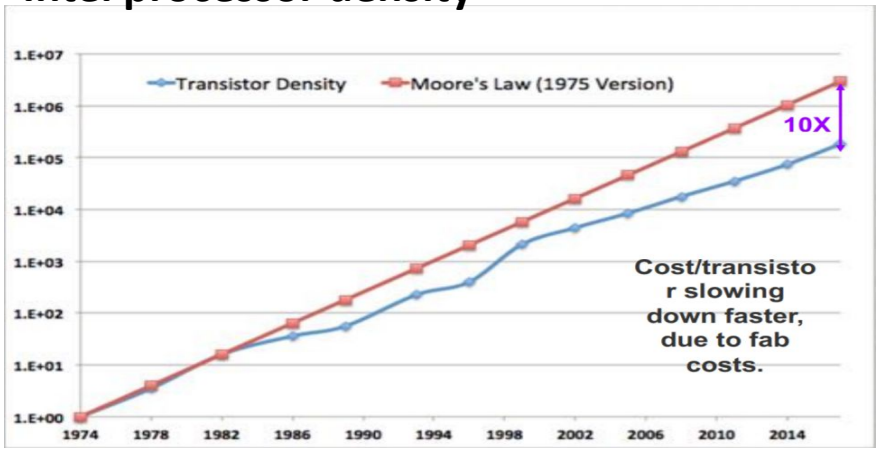
Moore's law is slowing down

- The number of transistors on IC chips doubles approx. every 2 years

DRAM capacity



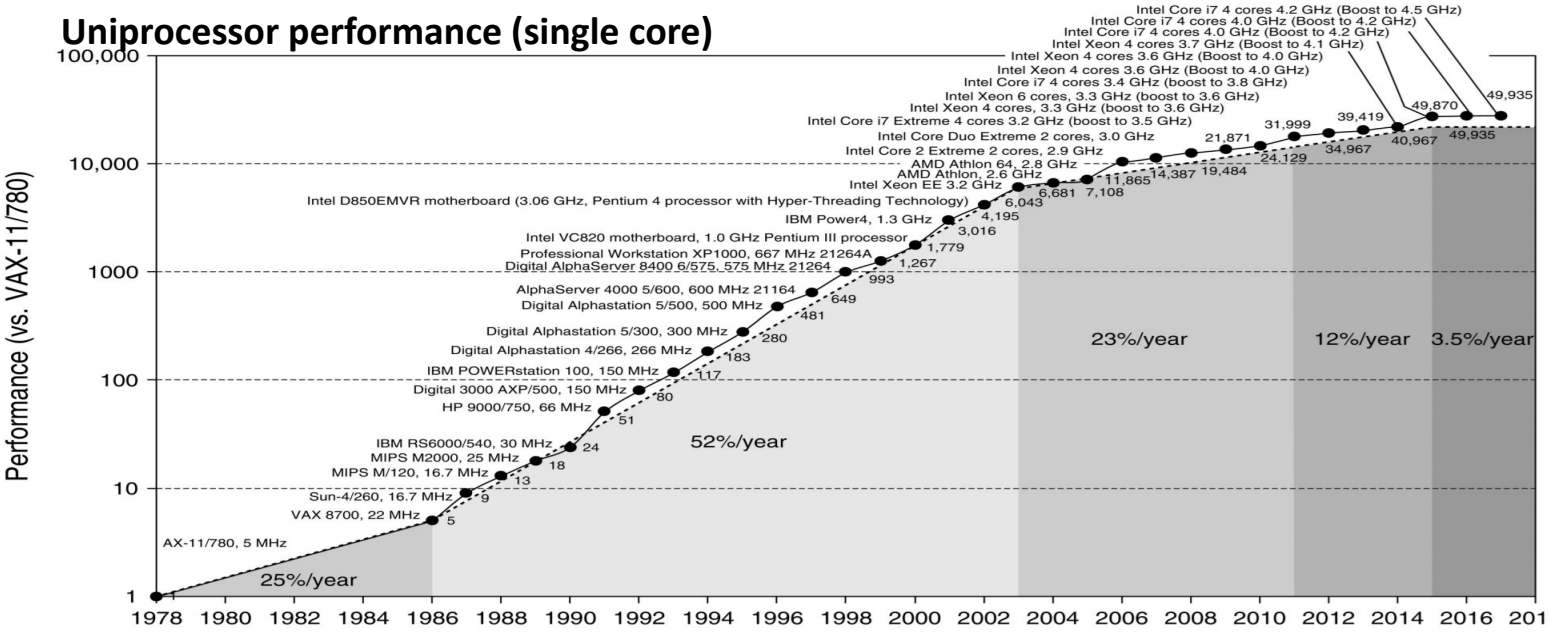
Intel processor density



Source:The end of moore's law & faster general purpose computing, and a road forward, John Hennessy. https://p4.org/assets/P4WS_2019/Speaker_Slides/9_2.05pm_John_Hennessey.pdf

End of Dennard scaling

- Processor frequency increased for free as transistors became smaller
- Transistors became so small that current leakage overheats the chip



Source:The end of moore's law & faster general purpose computing, and a road forward, John Hennessy. https://web.stanford.edu/~hennessy/Future%20of%20Computing.pdf

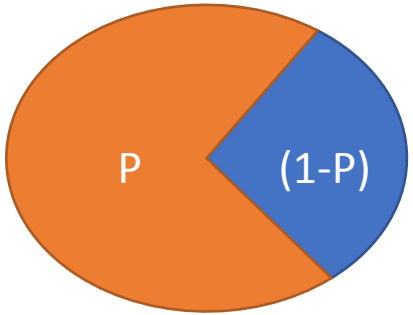
Implications of CPU Evolution

- Increasing transistor count/clock speed
  - Greater number of tasks can be executed concurrently

- Clock speed increases have almost stopped in the past few years
  - Instead, more transistors = more CPU cores
  - More cores = increased opportunity for parallelism

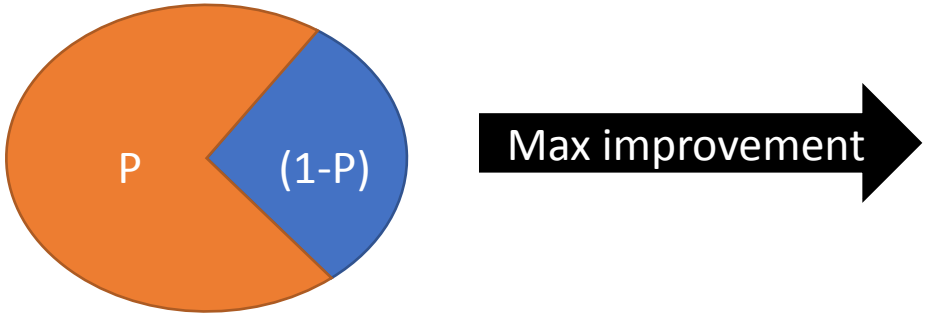
Amdahl's Law

- Speed up does not necessarily apply to the entire system
- Speed up indicates a relative performance improvement
  - Originally spent time to improved time ratio
- $\text{Speed up} = \frac{1}{(1-P)+P/S}$
  - (1 – P) = the part that was not enhanced
  - P = the part that was enhanced
  - S = speed up of the part that was enhanced



Amdahl's Law

- Speed up does not necessarily apply to the entire system
- Speed up indicates a relative performance improvement
  - Originally spent time to improved time ratio
- $\text{Speed up} = \frac{1}{(1-P)+P/S}$
  - (1 – P) = the part that was not enhanced
  - P = the part that was enhanced
  - S = speed up of the part that was enhanced



Amdahl's Law

- Speed up does not necessarily apply to the entire system
- Speed up indicates a relative performance improvement
  - Originally spent time to improved time ratio
- $\text{Speed up} = \frac{1}{(1-P)+P/S}$
  - (1 – P) = the part that was not enhanced
  - P = the part that was enhanced
  - S = speed up of the part that was enhanced

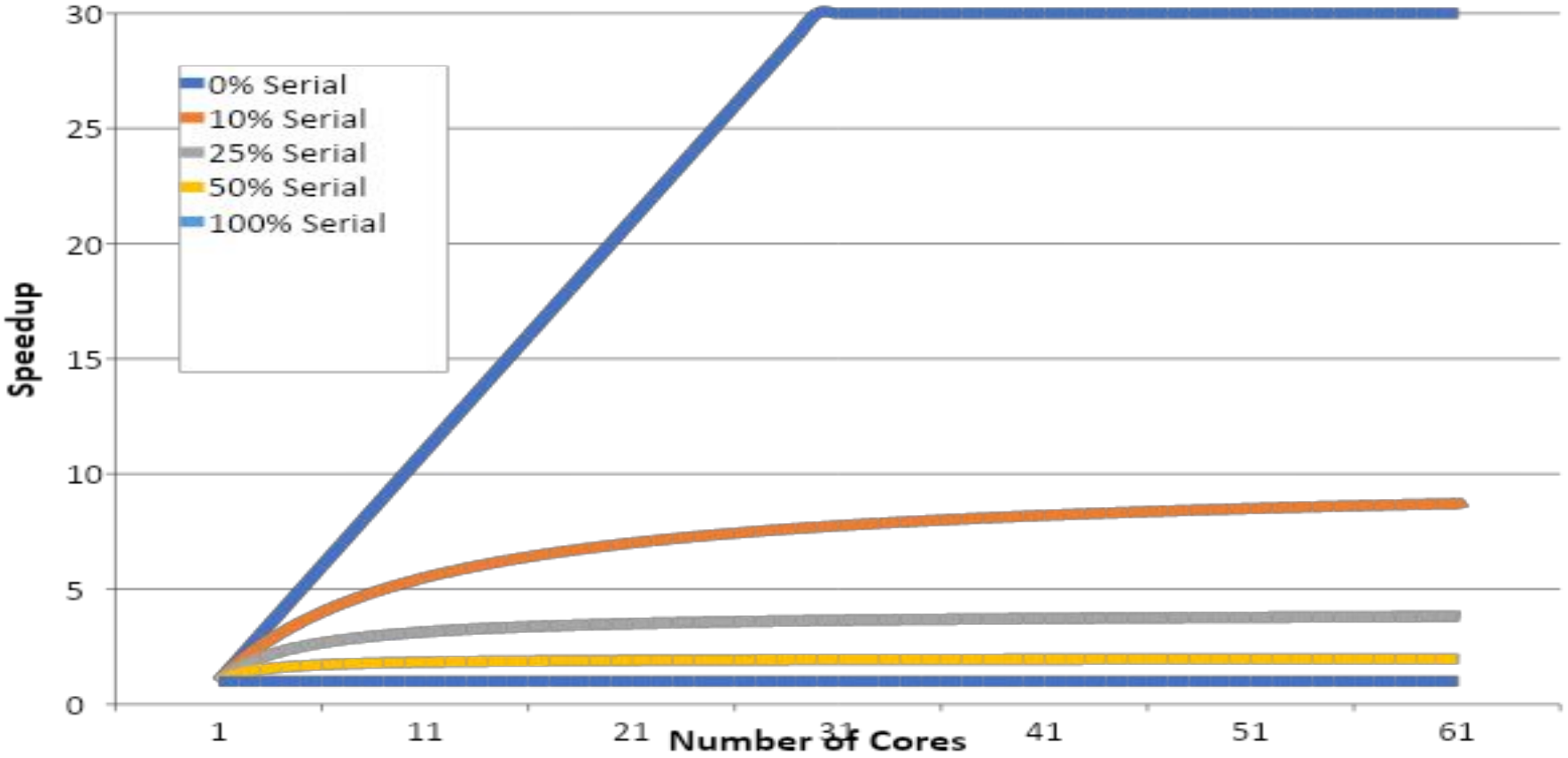
Amdahl's Law

- Upper bound on performance gains from parallelism
  - If I take a single-threaded task and parallelize it over *N* CPUs, how much more quickly will my task complete?

- Definition:
  - *seq* is the fraction of processing time that is processed sequentially
  - *par* is the fraction of processing time that can be parallelized
  - *seq+par = 1*
  - *N* is the number of CPU cores

$$\text{Speedup} = \frac{1}{seq + \frac{par}{N}}$$

Amdahl's Law


0% Serial
10% Serial
25% Serial
50% Serial
100% Serial
Speedup
30
25
20
15
10
5
0
1
11
21
31
41
51
61
Number of Cores

Concurrency

- In general, concurrency (like parallelism) is used because it is necessary for a system to function.

- It is also largely motivated by increased performance
  - The potential for more tasks to happen at once can thus increases performance (especially, if we have multiple cores on our machine)

Concurrency

- In general, concurrency (like parallelism) is used because it is necessary for a system to function.

- It is also largely motivated by increased performance
  - The potential for more tasks to happen at once can thus increases performance (especially, if we have multiple cores on our machine)

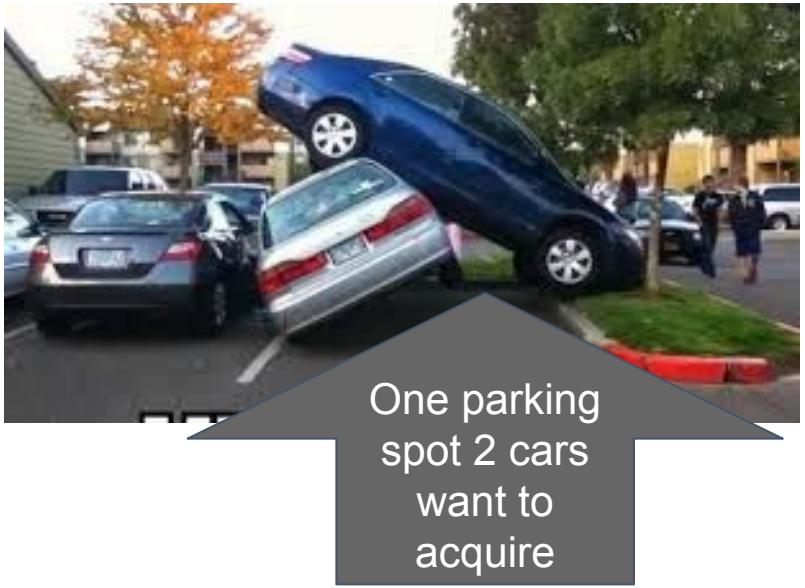
Concurrency comes with some caveats however (next slide!)

Bad Concurrency = Data Race

- When two (or more) processes contending for one shared resource.

Bad Concurrency = Data Race

- When two (or more) processes contending for one shared resource.



Data race is not always as obvious...(1/4)

- Imagine you check your fridge and find there is no milk
    - So you run to the store

Data race is not always as obvious...(2/4)

- Imagine you check your fridge and find there is no milk
  - So you run to the store
- Then moments later your roommate checks the fridge and finds it is empty
  - So they run to the store

Data race is not always as obvious...(3/4)

- Imagine you check your fridge and find there is no milk
  - So you run to the store
- Then moments later your roommate checks the fridge and finds it is empty
  - So they run to the store
- Roommate # 3 comes and notices the same
  - ....

Data race is not always as obvious...(4/4)

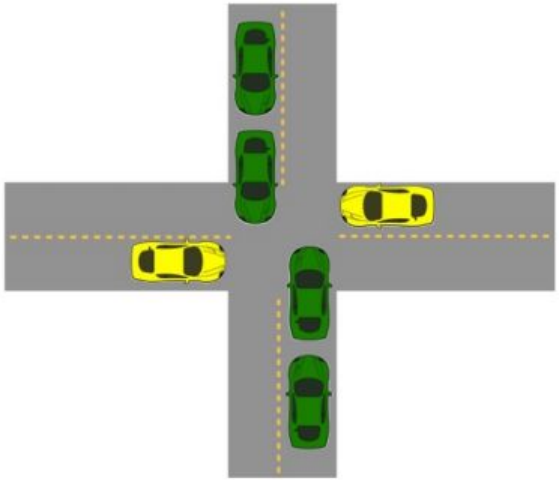
- You get the idea when you then find out you have 3 times as much milk as your house needs when everyone returns.

Bad Concurrency = Deadlock

- Grid lock in a traffic jam
- Each car prevents others from going through a shared resource (the intersection).
- (One car needs a piece of the intersection in order to move forward)

Bad Concurrency = Starvation

- Imagine a constant stream of green cars
- Progress is still being made by the green cars
- The yellow cars can never make progress to get across the street.
  - They are resource starved of a shared resource (again, they cannot cross the intersection)



Concurrent Programming needs some extra care

- Races
  - Outcome depends on the arbitrary scheduling decisions
  - e.g. Who gets the last seat on the airplane.
- Deadlock: Improper resource allocation prevents forward progress
  - e.g. traffic gridlock
- Starvation/Fairness: External events and/or scheduling decisions can prevent sub-task progress
  - e.g. Someone jumping in front of you in line

- But regardless, concurrent programming is important and necessary to get the most out of current processor architectures!

A Few Approaches to Concurrency

- Process-Based
  - Fork() different processes
  - Each process has its own private address space
- Event-Based
  - Programmer manually interleaves multiple logical flows and polls for events
  - All flows share the same address space
  - Uses technique called I/O multiplexing
- **Thread-based (Today's focus)**
  - **Kernel automatically interleaves multiple logical flows**
  - **Each flow shares the same address space**
  - **Hybrid of process-based and event-based.**

Threads

Problems with Processes

- Process creation is heavyweight (i.e. slow)

- IPC mechanisms are cumbersome

Problems with Processes

- Process creation is heavyweight (i.e. slow)
  - Space must be allocated for the new process
  - fork() copies all state of the parent to the child
- IPC mechanisms are cumbersome

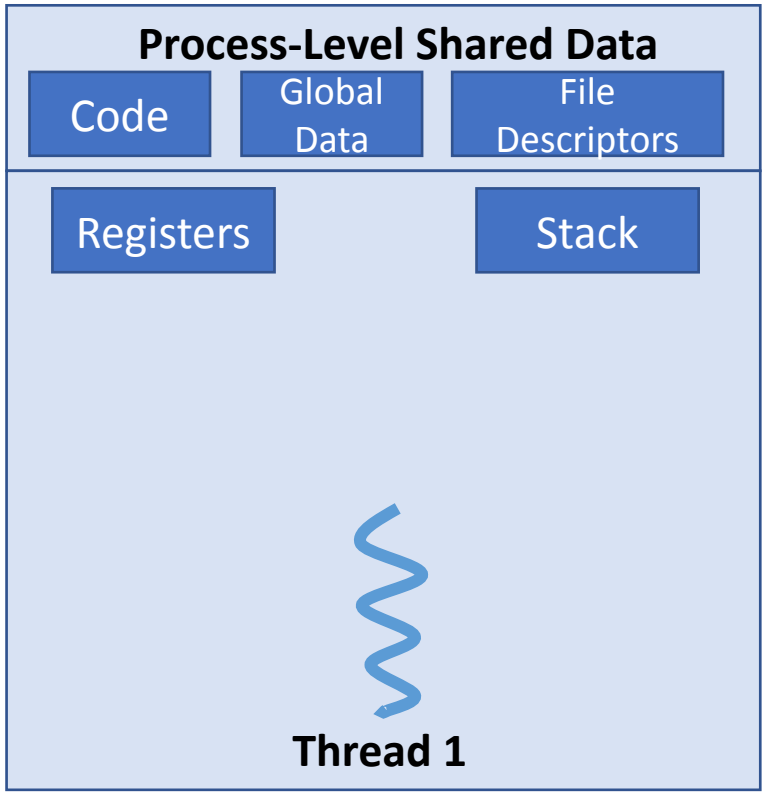
Problems with Processes

- Process creation is heavyweight (i.e. slow)
  - Space must be allocated for the new process
  - fork() copies all state of the parent to the child
- IPC mechanisms are cumbersome
  - Difficult to use fine-grained synchronization
  - Message passing is slow
    - Each message may have to go through the kernel

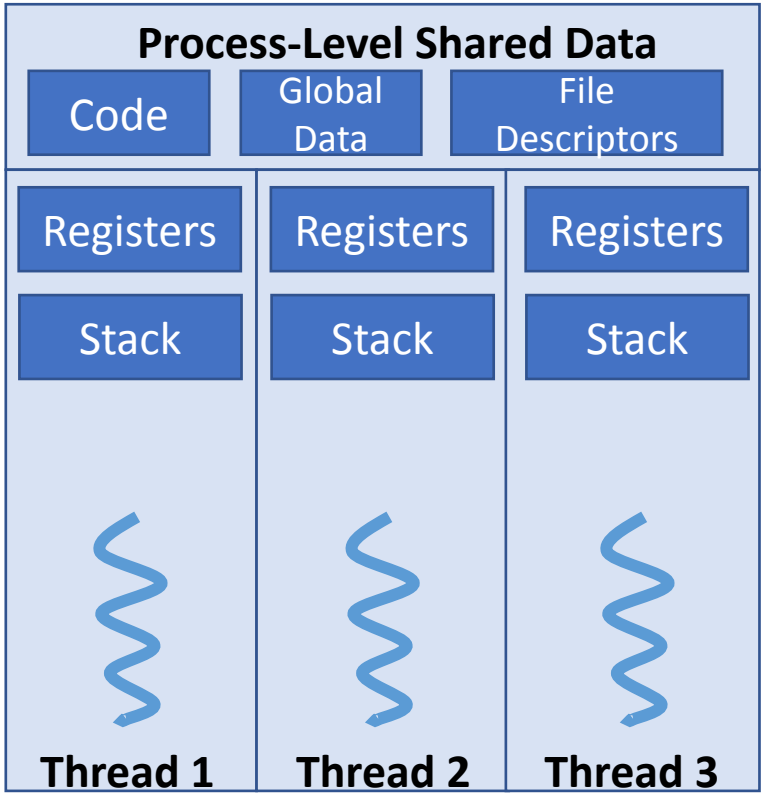
Threads

- Light-weight processes that share the same memory and state
- Every process has at least one thread
- Benefits:
  - Resource sharing, no need for IPC
  - Economy: faster to create, faster to context switch
  - Scalability: simple to take advantage of multi-core CPUs

Single-Threaded Process

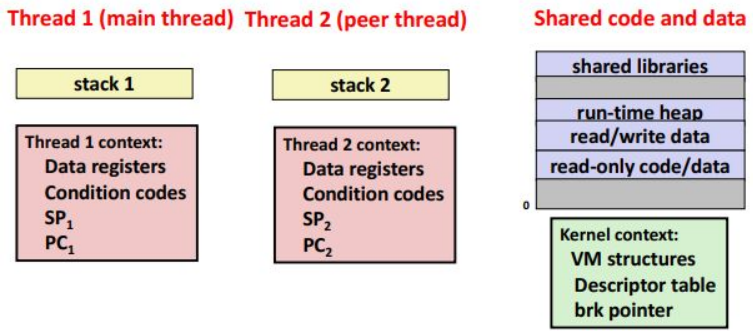


Multi-Threaded Process



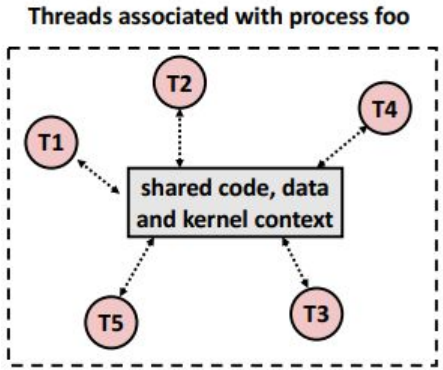
A Process can have Multiple Threads

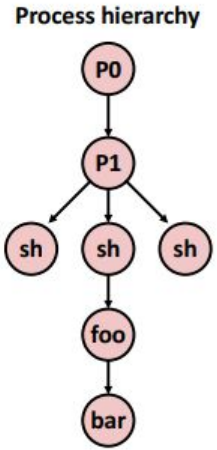
- Each thread shares the same code, data, and kernel context
- A thread has its own thread id (TID)
- A thread has its own logical control flow (no need to exec)
- A thread has its own stack for local variables



View of Threads

- Threads associated with a process form a “pool” of peers
  - Unlike processes (on the right) which form a tree hierarchy (i.e. parent/child relationship)

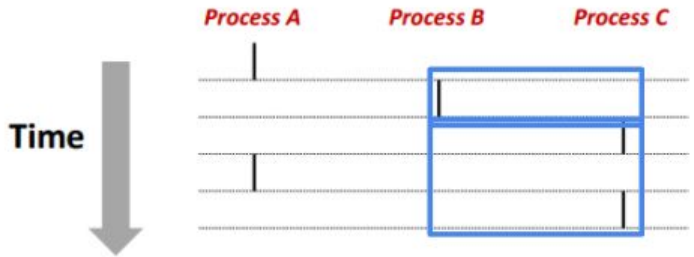




Remember this diagram on Concurrent Processes?

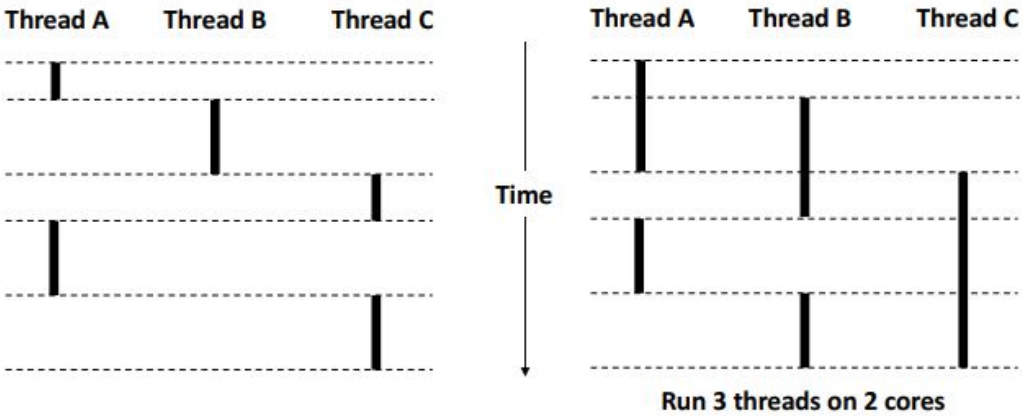
- We looked at multiple processes running on a single core (next slide for multiple cores)

  - On a single core, which processes here are concurrent relative to each other?
    - Concurrent: A&B, A&C
  - Which are sequential?
    - Sequential: B &C



Concurrent Thread (or Process) Execution

- Single Core Process
  - Simulate parallelism by time slicing
- Multi-Core Processor
  - Can have true parallelism



Threads vs Processes

- Similarities
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores if available)
  - Each is context switched
- Differences
  - Threads share all code and data (except local stacks)
    - Processes (typically) do not (i.e. fork makes a copy)
  - Threads are usually less expensive than managing processes
    - Process control is twice as expensive as thread control
      - Linux estimates
        - ~20k cycles to create and reap a process
        - ~10k cycles to create and reap a thread

Pthreads
(POSIX Threads)

POSIX Pthreads

- POSIX
  - Portable Operating System Interface

- POSIX standard API for thread creation
  - IEEE 1003.1c
  - *Specification*, not *implementation*
    - Defines the API and the expected behavior
    - … but not how it should be implemented

- Implementation is system dependent
  - On some platforms, user-level threads
  - On others, maps to kernel-level threads

Posix Threads API (PThreads Interface)

- Sample functions
  - Creating and reaping threads
    - `pthread_create()`
    - `pthread_join()`
  - Determining thread ID
    - `pthread_self()`
  - Terminating threads
    - `pthread_cancel()`
    - `pthread_exit()`
      - `exit()` - Terminates all threads
      - `return` - terminates current thread
  - Synchronizing access to shared variables
    - `pthread_mutex_init` (mutual exclusion lock)
      - `pthread_mutex_lock` and `pthread_mutex_unlock`

Pthread examples

Hello Thread

- The thread that is “launched” is a function in the program
  - This is done when the thread is created
  - Different attributes can be sent to threads (in this case the first NULL)
  - Arguments can also be passed to the function (second NULL)

```
1  // Compile with:
2  //
3  // clang -lpthread thread1.c -o thread1
4  //
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <pthread.h>
8
9  // Thread with variable arguments
10 void *thread(void *vargp){
11         printf("Hello from thread\n");
12         return NULL;
13 }
14
15 int main(){
16         // Store our Pthread ID
17         pthread_t tid;
18         // Create and execute the thread
19         pthread_create(&tid, NULL, thread, NULL);
20         // Wait in 'main' thread until thread executes
21         pthread_join(tid,NULL);
22         // end program
23         return 0;
24 }
```

Hello Thread

- The thread that is “launched” is a function in the program
  - This is done when the thread is created
  - Different attributes can be sent to threads (in this case the first NULL)
  - Arguments can also be passed to the function (second NULL)

```
1  // Compile with:
2  //
3  // clang -lpthread thread1.c -o thread1
4  //
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <pthread.h>
8
9  // Thread with variable arguments
10 void *thread(void *vargp){
11         printf("Hello from thread\n");
12         return NULL;
13 }
14
15 int main(){
16         // Store our Pthread ID
17         pthread_t tid;
18         // Create and execute the thread
19         pthread_create(&tid, NULL, thread, NULL);
20         // Wait in 'main' thread until thread executes
21         pthread_join(tid,NULL);
22         // end program
23         return 0;
24 }
```

Hello Thread

- The thread that is “launched” is a function in the program
  - This is done when the thread is created
  - Different attributes can be sent to threads (in this case the first NULL)
  - Arguments can also be passed to the function (second NULL)
- `pthread_join` is the equivalent to “`wait`” for threads

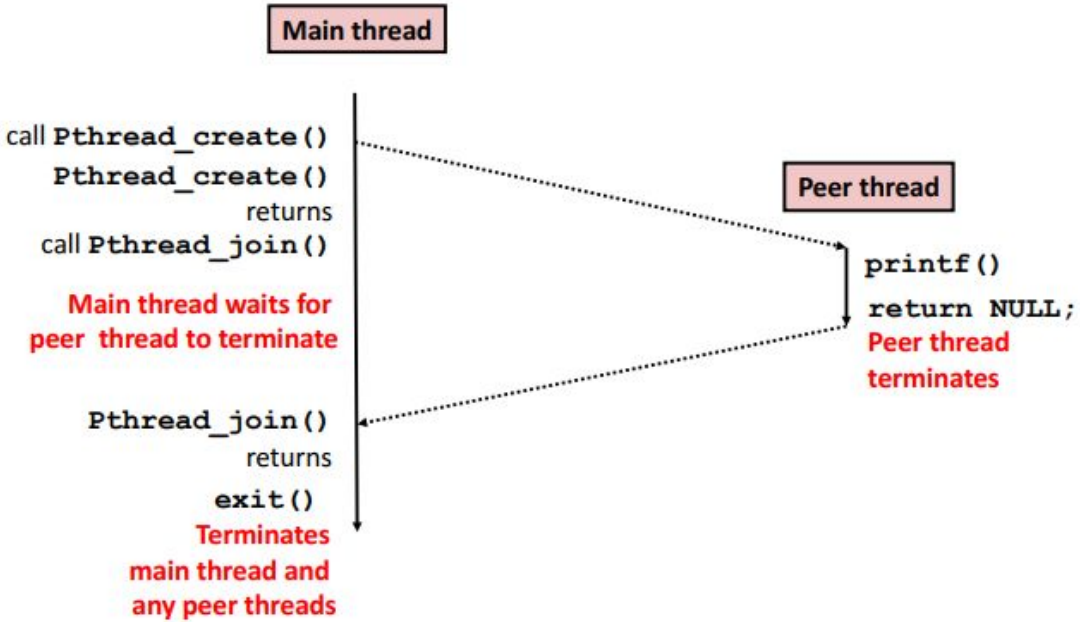
```
1  // Compile with:
2  //
3  // clang -lpthread thread1.c -o thread1
4  //
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <pthread.h>
8
9  // Thread with variable arguments
10 void *thread(void *vargp){
11         printf("Hello from thread\n");
12         return NULL;
13 }
14
15 int main(){
16         // Store our Pthread ID
17         pthread_t tid;
18         // Create and execute the thread
19         pthread_create(&tid, NULL, thread, NULL);
20         // Wait in 'main' thread until thread executes
21         pthread_join(tid,NULL);
22         // end program
23         return 0;
24 }
```

Hello Thread

- The thread that is “launched” is a function in the program
  - This is done when the thread is created
  - Different attributes can be sent to threads (in this case the first NULL)
  - Arguments can also be passed to the function (second NULL)
- `pthread_join` is the equivalent to “`wait`” for threads
- What if we don’t call join?

```
1  // Compile with:
2  //
3  // clang -lpthread thread1.c -o thread1
4  //
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <pthread.h>
8
9  // Thread with variable arguments
10 void *thread(void *vargp){
11         printf("Hello from thread\n");
12         return NULL;
13 }
14
15 int main(){
16         // Store our Pthread ID
17         pthread_t tid;
18         // Create and execute the thread
19         pthread_create(&tid, NULL, thread, NULL);
20         // Wait in 'main' thread until thread executes
21         pthread_join(tid,NULL);
22         // end program
23         return 0;
24 }
```

Visual execution of “Hello Thread”



Launching multiple threads

- Store 10 thread ids.

```c
// Compile with:
//
// clang -lpthread thread2.c -o thread2
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10

// Thread with variable arguments
void *thread(void *vargp){
        printf("Hello from thread %ld\n", pthread_self());
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Main thread id: %ld\n",pthread_self());
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }
        // Make main wait for each thread
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }

        printf("Main thread returns: %ld\n",pthread_self());
        // end program
        return 0;
}
```

Launching multiple threads

- Store 10 thread ids.
- Launch 10 threads

```c
// Compile with:
//
// clang -lpthread thread2.c -o thread2
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10

// Thread with variable arguments
void *thread(void *vargp){
        printf("Hello from thread %ld\n", pthread_self());
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Main thread id: %ld\n",pthread_self());
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }
        // Make main wait for each thread
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }

        printf("Main thread returns: %ld\n",pthread_self());
        // end program
        return 0;
}
```

Launching multiple threads

- Store 10 thread ids.
- Launch 10 threads
- Print out their thread ids to show which thread is executing.

```
1  // Compile with:
2  //
3  // clang -lpthread thread2.c -o thread2
4  //
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <pthread.h>
8
9  #define NTHREADS 10
10
11 // Thread with variable arguments
12 void *thread(void *vargp){
13         printf("Hello from thread %ld\n", pthread_self());
14         return NULL;
15 }
16
17 int main(){
18         // Store our Pthread ID
19         pthread_t tids[NTHREADS];
20         printf("Main thread id: %ld\n",pthread_self());
21         // Create and execute multiple threads
22         for(int i=0; i < NTHREADS; ++i){
23                 pthread_create(&tids[i], NULL, thread, NULL);
24         }
25         // Make main wait for each thread
26         for(int i=0; i < NTHREADS; ++i){
27                 pthread_join(tids[i], NULL);
28         }
29
30         printf("Main thread returns: %ld\n",pthread_self());
31         // end program
32         return 0;
33 }
```

Launching multiple threads

- Store 10 thread ids.
- Launch 10 threads
- Print out their thread ids to show which thread is executing.
- Join all of our threads with the main thread
  - (i.e. make the main thread wait until all 10 threads have executed.)

```c
// Compile with:
//
// clang -lpthread thread2.c -o thread2
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10

// Thread with variable arguments
void *thread(void *vargp){
        printf("Hello from thread %ld\n", pthread_self());
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Main thread id: %ld\n",pthread_self());
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }
        // Make main wait for each thread
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }

        printf("Main thread returns: %ld\n",pthread_self());
        // end program
        return 0;
}
```

Launching multiple threads

- *New Program*

```
1  // Compile with:
2  //
3  // clang -lpthread thread3.c -o thread3
4  //
5  #include <stdio.h>
6  #include <stdlib.h>
7  #include <pthread.h>
8
9  #define NTHREADS 10000
10
11 int counter = 0;
12
13 // Thread with variable arguments
14 void *thread(void *vargp){
15         counter = counter +1;
16         return NULL;
17 }
18
19 int main(){
20         // Store our Pthread ID
21         pthread_t tids[NTHREADS];
22         printf("Counter starts at: %d\n",counter);
23         // Create and execute multiple threads
24         for(int i=0; i < NTHREADS; ++i){
25                 pthread_create(&tids[i], NULL, thread, NULL);
26         }
27         // Create and execute multiple threads
28         for(int i=0; i < NTHREADS; ++i){
29                 pthread_join(tids[i], NULL);
30         }
31
32         printf("Final Counter value: %d\n",counter);
33         // end program
34         return 0;
35 }
```

Launching multiple threads

- *New Program*
- This time launch 10000 threads

```c
// Compile with:
//
// clang -lpthread thread3.c -o thread3
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10000

int counter = 0;

// Thread with variable arguments
void *thread(void *vargp){
        counter = counter +1;
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Counter starts at: %d\n",counter);
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }

        printf("Final Counter value: %d\n",counter);
        // end program
        return 0;
}
```

Launching multiple threads

- This time launch 10000 threads
- counter is shared between threads
- What is wrong with this program?

```
1 // Compile with:
2 //
3 // clang -lpthread thread3.c -o thread3
4 //
5 #include <stdio.h>
6 #include <stdlib.h>
7 #include <pthread.h>
8
9 #define NTHREADS 10000
10
11 int counter = 0;
12
13 // Thread with variable arguments
14 void *thread(void *vargp){
15         counter = counter +1;
16         return NULL;
17 }
18
19 int main(){
20         // Store our Pthread ID
21         pthread_t tids[NTHREADS];
22         printf("Counter starts at: %d\n",counter);
23         // Create and execute multiple threads
24         for(int i=0; i < NTHREADS; ++i){
25                 pthread_create(&tids[i], NULL, thread, NULL);
26         }
27         // Create and execute multiple threads
28         for(int i=0; i < NTHREADS; ++i){
29                 pthread_join(tids[i], NULL);
30         }
31
32         printf("Final Counter value: %d\n",counter);
33         // end program
34         return 0;
35 }
```

Launching multiple threads

- This time launch 10000 threads
- counter is shared between threads
- What is wrong with this program?
- What is the final output?

```c
// Compile with:
//
// clang -lpthread thread3.c -o thread3
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10000

int counter = 0;

// Thread with variable arguments
void *thread(void *vargp){
        counter = counter +1;
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Counter starts at: %d\n",counter);
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }

        printf("Final Counter value: %d\n",counter);
        // end program
        return 0;
}
```

Launching multiple threads

- This time launch 10000 threads
- counter is shared between threads
- What is wrong with this program?
- What is the final output?

```
Counter starts at: 0
Final Counter value: 9998
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9998
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9997
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9999
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9997
```

```c
// Compile with:
//
// clang -lpthread thread3.c -o thread3
//
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10000

int counter = 0;

// Thread with variable arguments
void *thread(void *vargp){
        counter = counter +1;
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Counter starts at: %d\n",counter);
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }

        printf("Final Counter value: %d\n",counter);
        // end program
        return 0;
}
```

What was happening?

Thread 1 (counter = counter + 1)	Thread 2 (counter = counter + 1)
Read “counter”: 10	Read “counter”: 10
Add 1 to “counter”: 11	Add 1 to “counter”: 11
Write to “counter”: 11	Write to “counter”: 11

What was happening?

Thread 1 (counter = counter + 1)

Read “counter”: 11

Add 1 to “counter”: 12

Write to “counter”: 12

Thread 3 (counter = counter + 1)

Read “counter”: 12

Add 1 to “counter”: 13

Write to “counter”: 13

Thread 2 (counter = counter + 1)

Read “counter”: 11

Add 1 to “counter”: 12

Write to “counter”: 12

Synchronization of Threads

- Shared variables are thus handy for moving around data
- If we do not share properly, we can have synchronization errors!
  - There is a solution however!
  - (recap below)

=

```
Counter starts at: 0
Final Counter value: 9998
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9998
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9997
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9999
-bash-4.2$ ./thread3
Counter starts at: 0
Final Counter value: 9997
```

Locks

- If multiple entities tries to acquire the lock, only one will succeed
  - Lock cannot be shared
- If someone is holding the lock others have to wait until the lock holder unlocks

- Lock can project shared variables or code regions that should not be executed concurrently

Example with lock

- Included a pthread_mutex_lock

```c
// Compile with:
// clang -lpthread thread4.c -o thread4
// This program fixes a problem with thread3.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10000

int counter = 0;
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

// Thread with variable arguments
void *thread(void *vargp){
        pthread_mutex_lock(&mutex1);
                counter = counter +1;
        pthread_mutex_unlock(&mutex1);
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Counter starts at: %d\n",counter);
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }

        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }
        printf("Final Counter value: %d\n",counter);
        // end program
        return 0;
}
```

Example with lock

- Included a pthread_mutex_lock
- lock and unlock protects
- Locks in other words enforce, that we have exclusive access to a region of code.

```
1  // Compile with:
2  // clang -lpthread thread4.c -o thread4
3  // This program fixes a problem with thread3.c
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include <pthread.h>
7
8  #define NTHREADS 10000
9
10 int counter = 0;
11 pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
12
13 // Thread with variable arguments
14 void *thread(void *vargp){
15         pthread_mutex_lock(&mutex1);
16                 counter = counter +1;
17         pthread_mutex_unlock(&mutex1);
18         return NULL;
19 }
20
21 int main(){
22         // Store our Pthread ID
23         pthread_t tids[NTHREADS];
24         printf("Counter starts at: %d\n",counter);
25         // Create and execute multiple threads
26         for(int i=0; i < NTHREADS; ++i){
27                 pthread_create(&tids[i], NULL, thread, NULL);
28         }
29
30         // Create and execute multiple threads
31         for(int i=0; i < NTHREADS; ++i){
32                 pthread_join(tids[i], NULL);
33         }
34         printf("Final Counter value: %d\n",counter);
35         // end program
36         return 0;
37 }
```

What was happening?

Thread 1 (counter = counter + 1)	Thread 2 (counter = counter + 1)
pthread_mutex_lock	pthread_mutex_lock

Read “counter”: 10	// Lock is held by thread 1 so
	// thread 2 has to wait until
	// thread 1 unlocks
	BLOCKS EXECUTION

Add 1 to “counter”: 11	// Now acquires the lock and runs
Write to “counter”: 11	Read “counter”: 10
pthread_mutex_unlock	Add 1 to “counter”: 11
	Write to “counter”: 11
	pthred_mutex_unlock

Example with lock

- Included a pthread_mutex_lock
- lock and unlock protect
- Locks in other words enforce, that we have exclusive access to a region of code.

```
mike:8$ gcc thread4.c -o thread4 -lpthread
mike:8$ ./thread4
Counter starts at: 0
Final Counter value: 10000
mike:8$ ./thread4
Counter starts at: 0
Final Counter value: 10000
mike:8$ ./thread4
Counter starts at: 0
Final Counter value: 10000
mike:8$ ./thread4
Counter starts at: 0
^[[AFinal Counter value: 10000
```

```c
// Compile with:
// clang -lpthread thread4.c -o thread4
// This program fixes a problem with thread3.c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

#define NTHREADS 10000

int counter = 0;
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;

// Thread with variable arguments
void *thread(void *vargp){
        pthread_mutex_lock(&mutex1);
                counter = counter +1;
        pthread_mutex_unlock(&mutex1);
        return NULL;
}

int main(){
        // Store our Pthread ID
        pthread_t tids[NTHREADS];
        printf("Counter starts at: %d\n",counter);
        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_create(&tids[i], NULL, thread, NULL);
        }

        // Create and execute multiple threads
        for(int i=0; i < NTHREADS; ++i){
                pthread_join(tids[i], NULL);
        }
        printf("Final Counter value: %d\n",counter);
        // end program
        return 0;
}
```

Posix Threads API (PThreads Interface)

- Sample functions
  - Creating and reaping threads
    - `pthread_create()`
    - `pthread_join()`
  - Determining thread ID
    - `pthread_self()`
  - Terminating threads
    - `pthread_cancel()`
    - `pthread_exit()`
      - `exit()` - Terminates all threads
      - `return` - terminates current thread
  - Synchronizing access to shared variables
    - `pthread_mutex_init`
      - `pthread_mutex_lock` and `pthread_mutex_unlock`