15-213 Recitation
VM + Malloc Lab (Checkpoint)

Your TAs

Friday, October 10th

Reminders

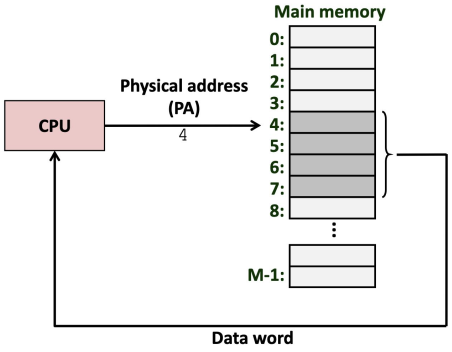
- `cachelab` was due ***yesterday***.
- `malloclab` was released yesterday:
  - Checkpoint: ***October 28th***
  - Final: ***November 4th***
- In Class Midterm: ***October 21st***

Agenda

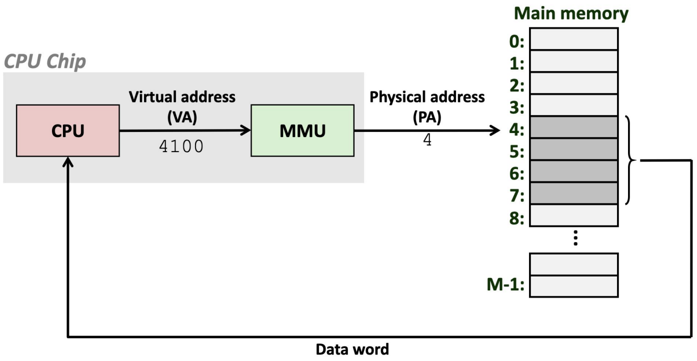
- Virtual Memory
- Activity: Analyzing TLBs with real-world examples
- Review: Programming in C
- `malloc` concepts
- Strategy Guide
  - Debugging and Suggested Roadmap

Virtual Memory - Review

Physical Addressing



Virtual Addressing



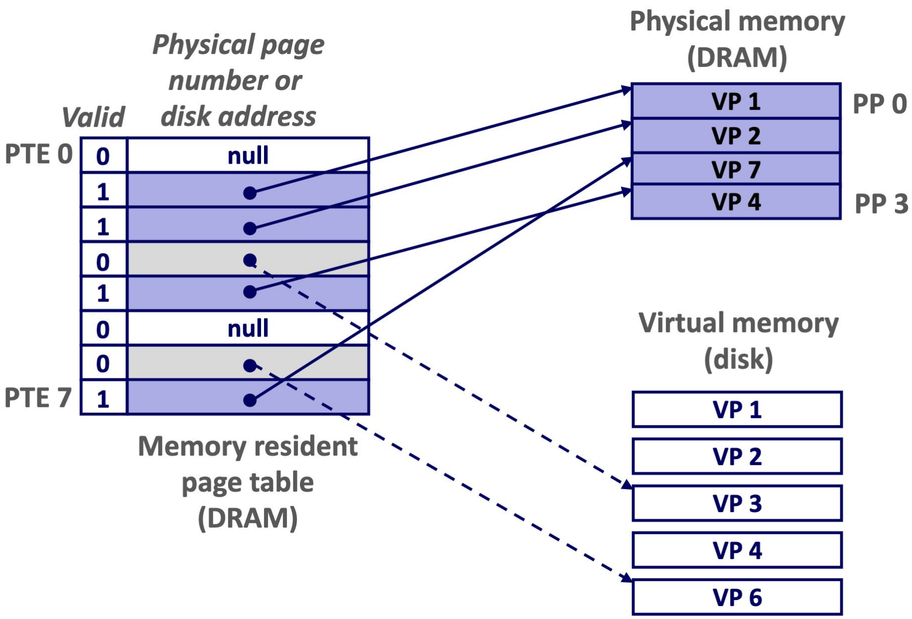
Memory address refers to an exact location in memory—only used in simple systems

Memory address refers to a process-specific address, mapped to physical memory via the hardware memory management unit.

One of the Great Ideas Of Computer Science™

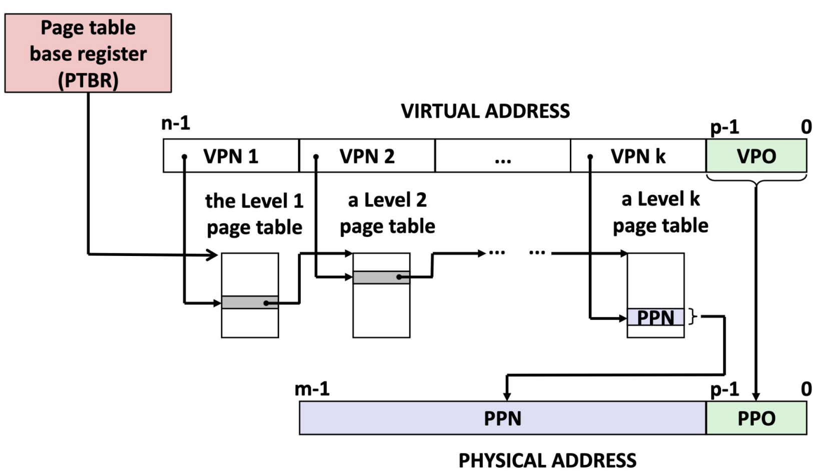
Virtual Memory - Page Table

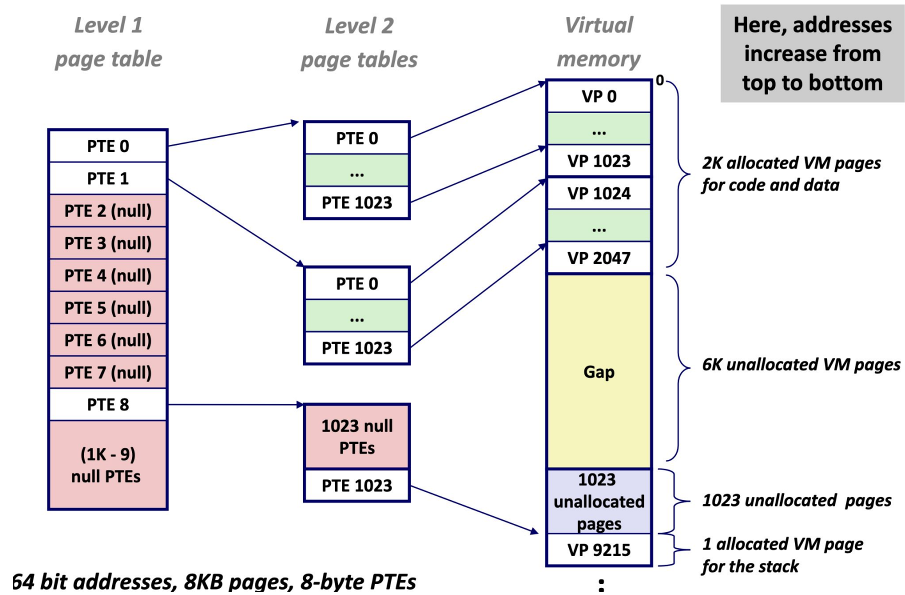
- Virtual addresses are mapped to physical addresses in the page table. Each entry is called a page table entry.
- Pages are in memory, like a cache. If they are not available in memory, we have a page miss.
- A page miss causes a page fault, which causes the OS to fetch the page from disk and evict a page from DRAM.



Virtual Memory - Multi-Level Page Tables

- The size of a page table quickly gets out of control when we have to address large addresses space.
- The solution is to nest page tables. The VPO/PPO acts as the pseudo-”block offset”





Example - Multi-Level Page Table

- Consider a system with 32 bit virtual address space and a 24 bit physical address space. Page Size is 4KB. Assume the size of entries in the Page Table is 4 bytes.
- **Question of interest**: How would we map the virtual address space? Is a single-level page table enough? Do we need more levels? Let's dive into it….

Example (Address Decomp.)

- Setup: 32 bit VA, 24 bit PA, Page Size = 4KB, PTE Size = 4 bytes
- Question 1: How many bits in the virtual/physical address for page offset?
- VPO = PPO = $\log_2$(page size) = 12 bits

20 bits	12 bits
to be discussed in later slides	offset (VPO = PPO)

Example (Mapping PTEs to VA)

- Setup: 32 bit VA, 24 bit PA, Page Size = 4KB, PTE Size = 4 bytes

- Question 2: How many pages are required to map the entire VA space?

- # of pages for VA space = size of VA space/size of a page
  - 2^32/2^12 = 2^20 PTEs

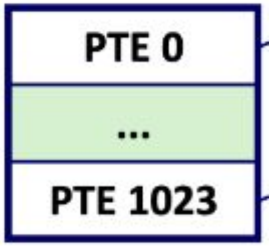
- Note that # of pages for VA space = # of PTEs for VA space
  - There is an one-to-one mapping between PTEs and virtual pages!

Example (Multi-Level Storage)

- Setup: 32 bit VA, 24 bit PA, Page Size = 4KB, PTE Size = 4 bytes
- So far, we've discussed preliminary values that tell us how to map onto the entire VA space.
  - General/"Single-Level" Ideas
- Now let's talk about how we can extend this to a multi-level page table

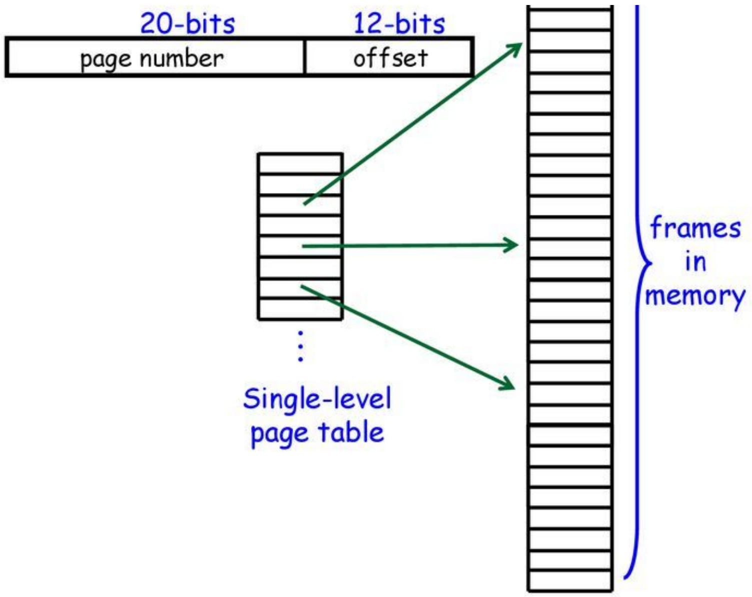
Example (PTEs in Pages)

- Setup: 32 bit VA, 24 bit PA, Page Size = 4KB, PTE Size = 4 bytes
- Question 3: How many PTEs (page table entries) fit inside a single page?
- # of PTEs in a page = size of a page / size of a PTE
  - 4KB/4B = 2^12/2^2 = 2^10 = 1024



Example (Multi-Level Storage)

- Setup: 32 bit VA, 24 bit PA, Page Size = 4KB, PTE Size = 4 bytes
- Question 4: How many pages do we need to cover the single level page table?
- # of pages for Single Level = # of PTEs to map VA space/# of PTEs in a page
  - 2^20/2^10 = 2^10 pages

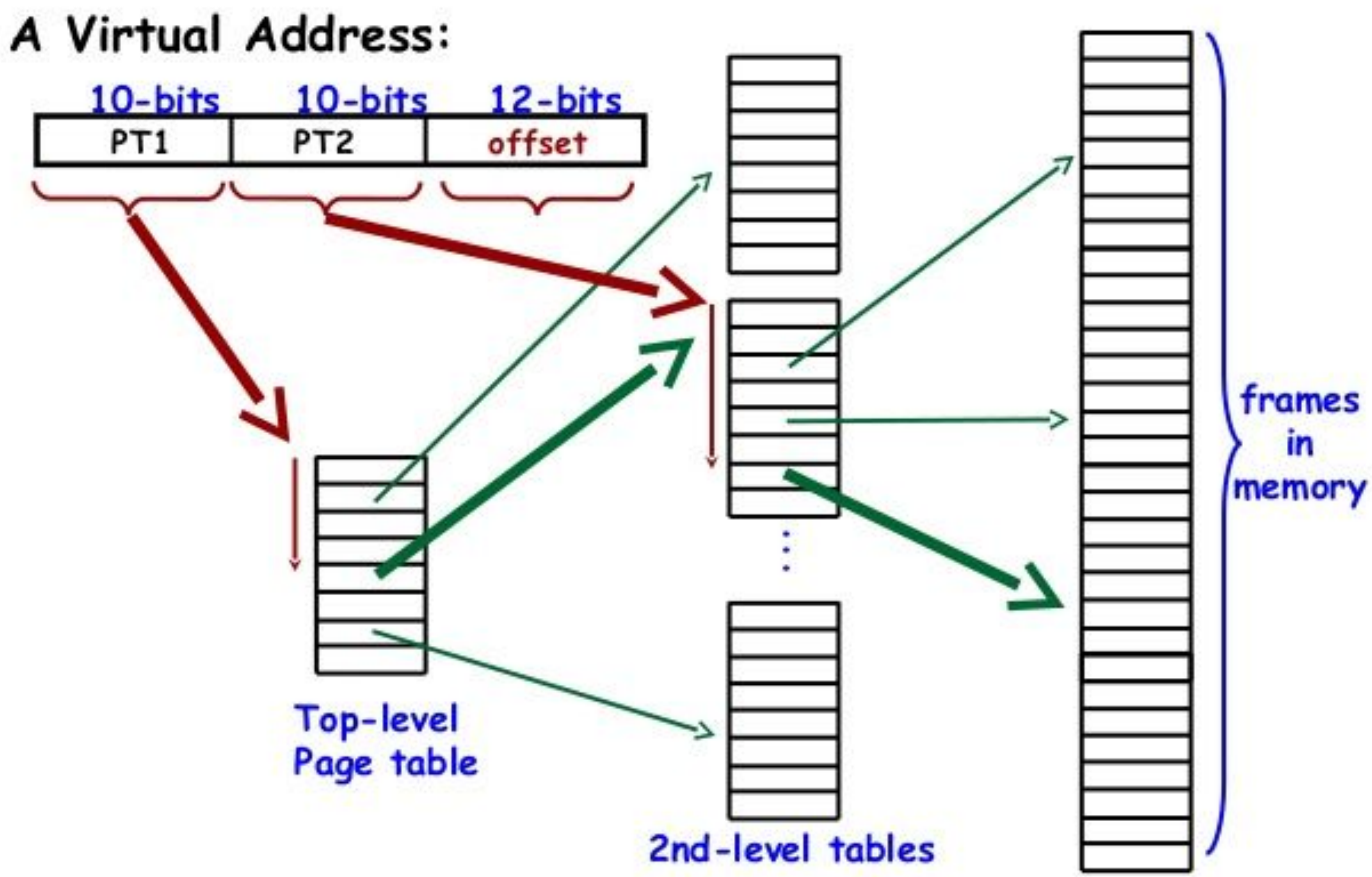


Example (Multi-Level Storage)

- Setup: 32 bit VA, 24 bit PA, Page Size = 4KB, PTE Size = 4 bytes
- Question 5: How many pages do we need to represent the outer level page table?
- # of pages for Outer Level = # of pages for Single Level / # PTEs in a page
  - 2^10/2^10 = 1 page

Example (Multi-Level Storage)

- This is what our final multi-level page table would look like



Example (Multi-Level Storage)

- Great, now we've setup a 2-level page table, let's talk about the benefits we get.
- Without the outer level, we would have to store the entirety of the single-level page table.
  - Oops that's (2^20 PTEs x 4 bytes) = 2^22 bytes = 4096 KB
  - Can also think of as (2^10 Pages x 4 KB)

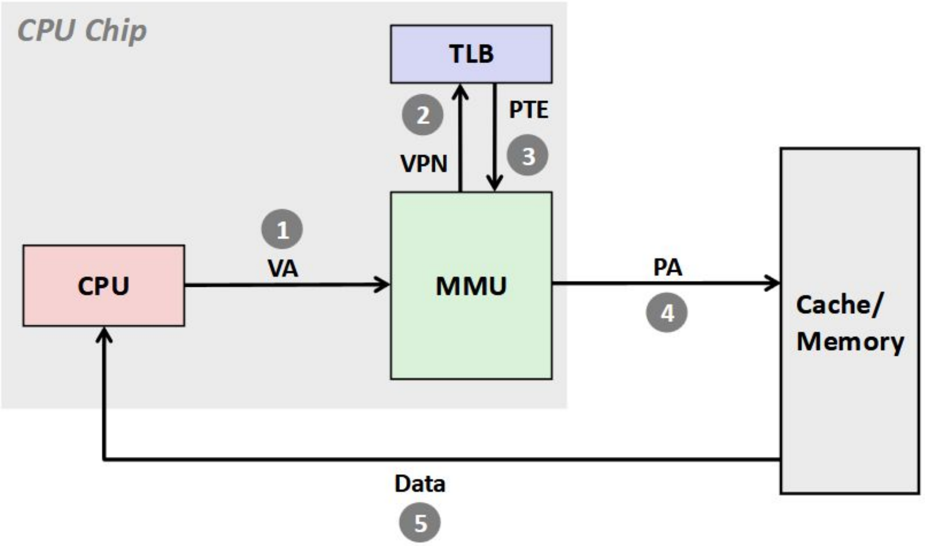
Example (Multi-Level Storage)

- Now we have two-levels. Suppose we have a single memory access (assuming the page table was empty at first). How many pages would be required?
- Entire outer level (there is only one page)
- 1 PTE needed from outer level => 1 page in inner level
- Total 2 pages! We saved a huge chunk of space.
  - 2 pages = 8 KB <<<<<< 4096 KB

Activity: Analyzing TLBs with Real World Examples

Review: What is a TLB?

- The TLB (or Translation Lookaside Buffer) is a cache that stores translations from virtual to physical addresses.
- Upon a TLB hit, we do not have to perform a page walk to perform translations!



TLB is a Cache!

- We can make similar analysis of TLBs as we did with caches
- TLBs are usually set associative
- Accesses to memory blocks –> Accesses to pages
- This changes how we think about locality and misses
  - But the general ideas still carry over from cachelab!

Analyzing TLB Benefits

- We focus on 2 main levels of analysis:
  1. **Locality of Access**
  2. **Size of Working Set**
- Before we move onto the activity, let's quickly introduce each, drawing parallels to cache analysis tools!

Locality of Access

- Suppose a workload has good locality, what are the benefits we get from a TLB?
- Good locality indicates reuse in memory in the same contiguous region in memory, or the same page
- Memory accesses to the same page benefit from previously stored translations!

Size of Working Set

- The working set of a program is the set of accessed, active virtual pages.
- What can happen if our working set is too large?
- A large working set results in **thrashing**, or the constant swapping of pages.
- For the TLB, this means a previously stored translation for a page will likely be invalid as the page has been swapped out.
  - Similar to capacity miss?

Activity

- In this activity, we'll be using real world scenarios, along with the two analysis tools, to reason about TLB benefits!
- Split into groups of 3-4 people!
- Please download the student handout from the course website!

Activity

- **Scenario**: We are running a large-data computation task, processing data on the magnitude of terabytes. Suppose we have a reasonably good, regular access pattern to data, as well as a reasonable page size (eg. 4KB)

- More information in the student handout!

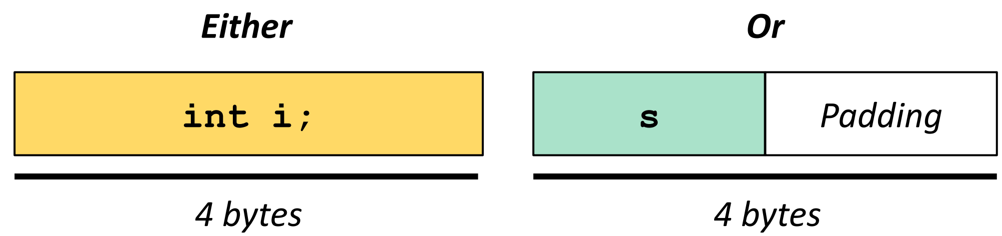
Activity

- Here are some main questions to answer:
  1. Given the features of the workload, what implications does it have on the TLB? (use the 2 analysis tools)
  2. Given these implications, what are some design changes that might help to gain the benefits from TLB or avoid the pitfalls of the TLB?
     - eg) cache features, page sizes, ect…

Review: Programming in C

Programming in C: Unions

```
union temp {
   int i;
   short s;
};
```

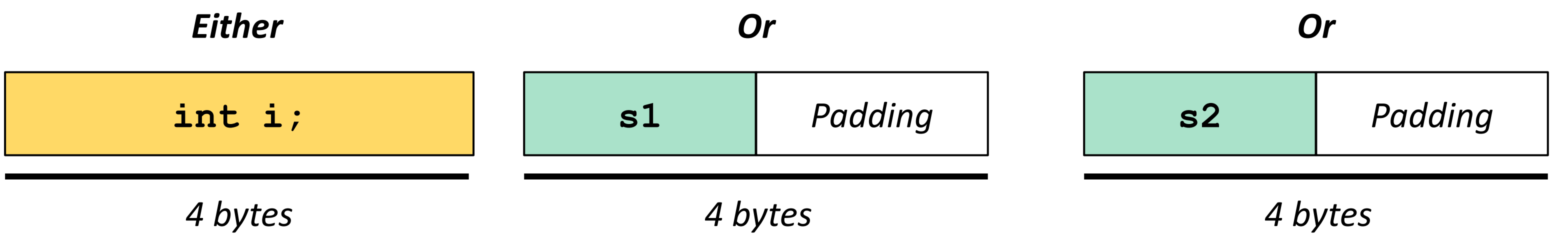


- Store potentially different data types in the same region of memory.
- Specifies multiple ways to interpret data at the same memory location.

Unions

How would the union be represented in memory?

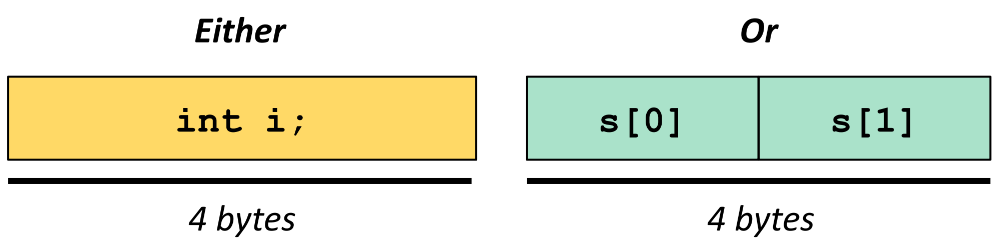
```
union temp {
    int i;
    short s1;
    short s2;
};
```



Unions

How would the union be represented in memory?

```
union temp {
    int i;
    short s[2];
};
```

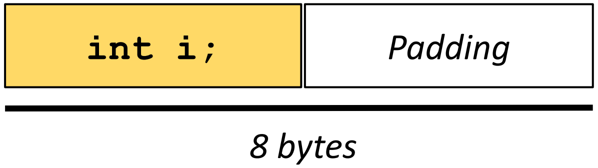


Unions

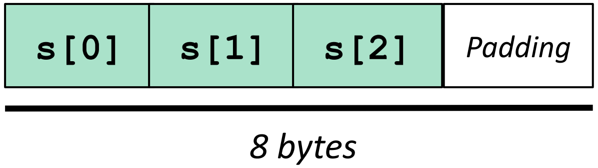
How would the union be represented in memory?

```
union temp {
    int i;
    short s[3];
};
```

Either



Or



Programming in C: Zero-Length Arrays

```
typedef uint64_t word_t;
```

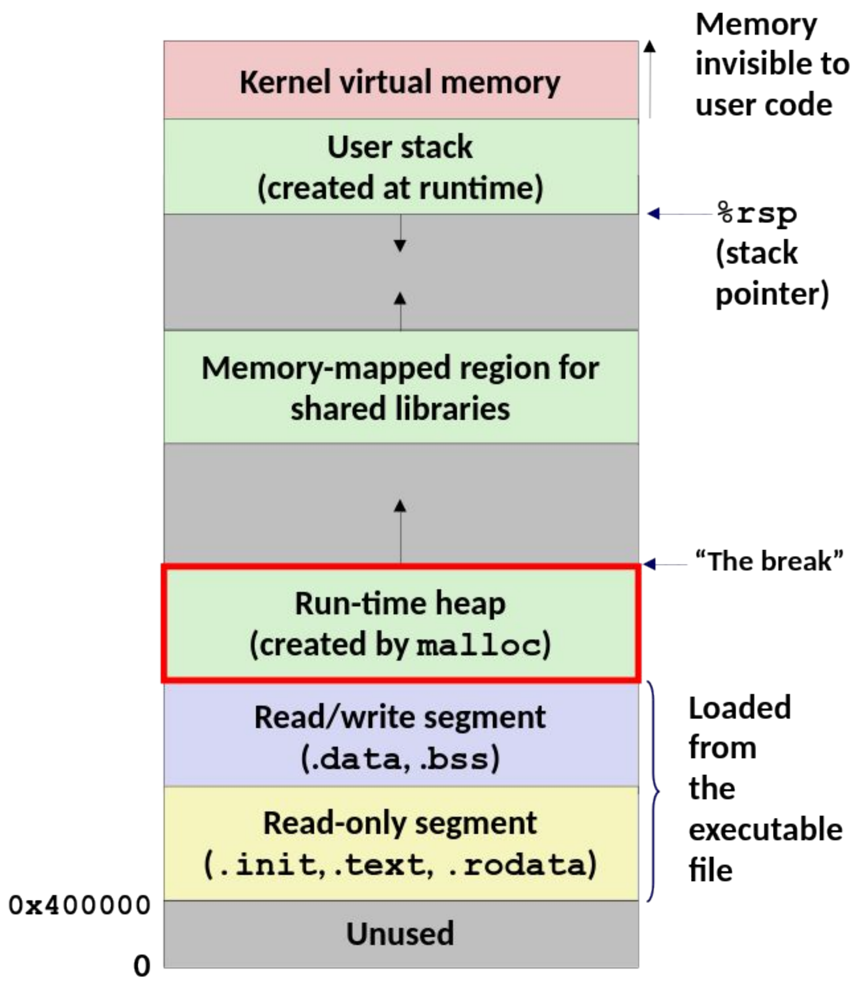
```
typedef struct block
{
    word_t header;
    unsigned char payload[0];        // Zero length array
} block_t;
```

- Allowed in GNU C as an extension.
- A zero-length array must be the last element in a struct.
- **`sizeof(payload)`** always returns 0
- But, the payload itself can have variable length

`malloc` Concepts

What does `malloc` do?

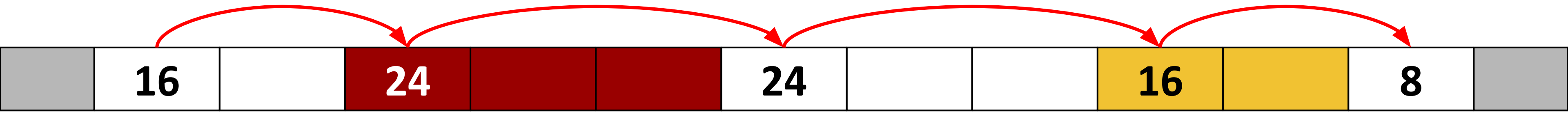
- Given a bunch of heap space, manage it effectively:
  1. Use heap space to organize blocks and information we store about blocks in a *structured way*.
  2. Using that structure, *decide where to allocate new blocks*.
  3. *Update structure correctly* when we allocate or free, *maintaining heap invariants*.
- …and do so in a way that maximizes throughput and utilization!



Throughput/Utilization

- What is throughput and utilization?
- **Throughput** is the average number of operations per second
- **Utilization** is peak ratio between the total amount of memory requested and the total amount of heap space allocated

Implicit Lists

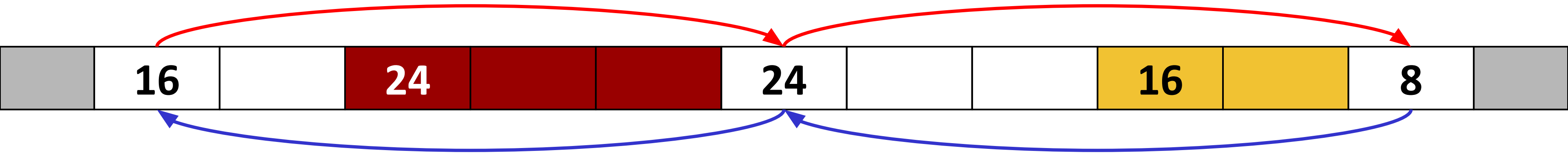


- Implicit lists traverse the heap through block lengths.
- What implication does this have on throughput/utilization?
- Since we have to iterate through all blocks, it results in terrible **throughput**

Coalescing

- Coalescing handles the case of consecutive free blocks - merging them to create a larger free block.
- What implication does this have on throughput/utilization?
- We get better utilization because we reduce **external fragmentation**
    - Recall external fragmentation occurs when there is enough aggregate heap memory, but no single free block is large enough!

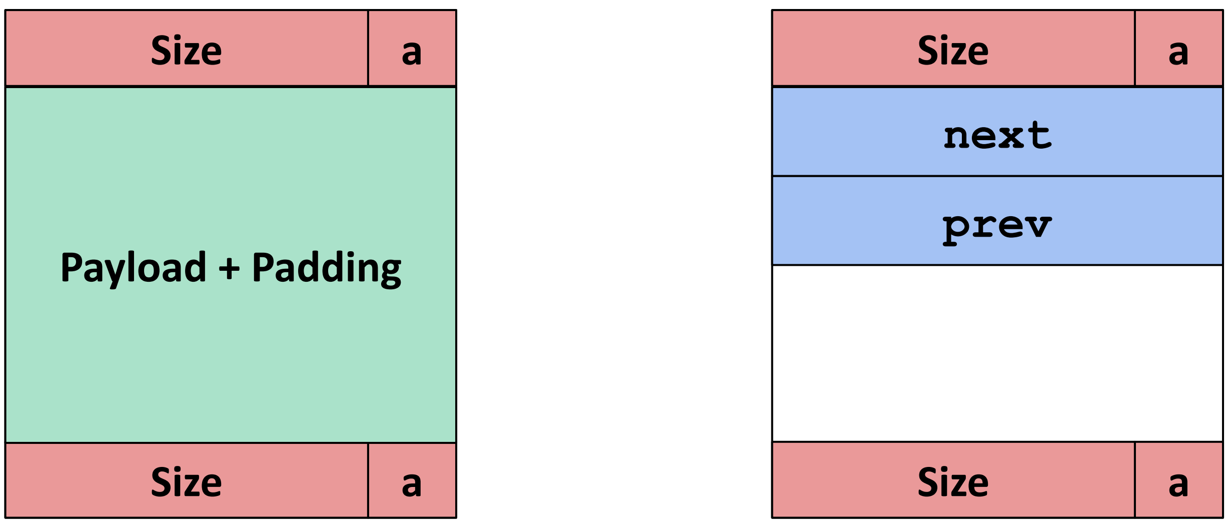
Explicit Lists



- Explicit lists traverse free blocks using pointers
- What implication does this have on throughput/utilization?
- We should see a great improvement in **throughput**, as we no longer have to iterate through ALL blocks to find a free block.
- However, pointers take space…

Explicit Lists

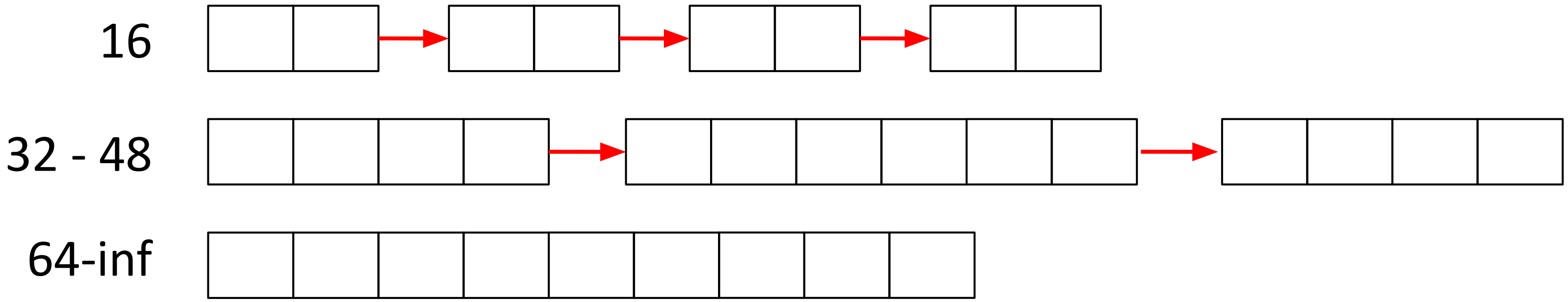
- How does explicit lists affect utilization/fragmentation?
  - Hint: Think about varying size requests



Allocated (as before)

Free

Segregated Lists



- We maintain *multiple* free blocks, based on sizes
  - Note that the size classes used above are just an example
- What implication does this have on throughput/utilization?
- Improves throughput, as we are guaranteed to find a large enough block faster!

`malloc` Starter Code

```
static block_t *coalesce_block(block_t *block) {
    // TODO: delete or replace this comment once you're done.
    return block;
}
```

- Starter code: **working** implementation of implicit free list with boundary tags.
- However, it does not implement coalescing!
- You will need to implement the features mentioned previously

`malloc` Starter Code

```
[dalud@angelshark:~/.../15213/s17/malloclabcheckpoint-handout] $ ./mdriver -p
Found benchmark throughput 13090 for cpu type Intel(R)Xeon(R)CPUE5520@2.27GHz, benchmark checkpoint

Throughput targets: min=2618, max=11781, benchmark=13090
.......................
Results for mm malloc:
  valid    util      ops    msecs     Kops  trace
   yes    78.4%       20     0.002    9632 ./traces/syn-array-short.rep
   yes    13.4%       20     0.001   25777 ./traces/syn-struct-short.rep
   yes    15.2%       20     0.001   24783 ./traces/syn-string-short.rep
   yes    73.1%       20     0.001   19277 ./traces/syn-mix-short.rep
   yes    16.0%       36     0.001   31192 ./traces/ngram-fox1.rep
   yes    73.6%      757     0.145    5237 ./traces/syn-mix-realloc.rep
 * yes    62.0%     5748     3.925    1464 ./traces/bdd-aa4.rep
 * yes    58.3%    87830  1682.766      52 ./traces/bdd-aa32.rep
 * yes    58.0%    41080   410.385     100 ./traces/bdd-ma4.rep
 * yes    58.1%   115380  4636.711      25 ./traces/bdd-nq7.rep
 * yes    56.6%    20547    26.677     770 ./traces/cbit-abs.rep
 * yes    55.8%    95276   675.303     141 ./traces/cbit-parity.rep
 * yes    58.0%    89623   611.511     147 ./traces/cbit-satadd.rep
 * yes    49.6%    50583   185.382     273 ./traces/cbit-xyz.rep
 * yes    40.6%    32540    76.919     423 ./traces/ngram-gulliver1.rep
 * yes    42.4%   127912  1284.959     100 ./traces/ngram-gulliver2.rep
 * yes    39.4%    67012   338.591     198 ./traces/ngram-moby1.rep
 * yes    38.6%    94828   701.305     135 ./traces/ngram-shake1.rep
 * yes    90.9%    80000  1455.891      55 ./traces/syn-array.rep
 * yes    88.0%    80000   915.167      87 ./traces/syn-mix.rep
 * yes    74.3%    80000   914.366      87 ./traces/syn-string.rep
 * yes    75.2%    80000   812.748      98 ./traces/syn-struct.rep
16 16     59.1% 1148359 14732.604      78

Average utilization = 59.1%. Average throughput = 78 Kops/sec
Checkpoint Perf index = 20.0 (util) + 0.0 (thru) = 20.0/100
```

Very **slow!**

Checkpoint Targets: Performance

Optimization	Utilization	Throughput
Implicit List (Starter Code)	59%	10–100
Explicit Free List[a]	mid-50s	1000–2500
Segregated Free Lists	–	6000

- We have motivated explicit lists and seg lists as a throughput optimization
- Could there be utilization improvements too?
  - Segregated lists size classes?
  - Fit Algorithms?

Design Choices

Design Choices

- Though we'll recommend a strategy later, there are many ways to optimize your allocator.
- What kind of implementation to use?
  - Implicit list, explicit, segregated, binary tree, etc.
- What fit algorithm to use?
  - Best Fit?
  - First Fit? Next Fit?
  - Which is faster? Which gets better utilization?
- There are many different ways to get a full score!

Strategy Guide: Debugging

In a perfect world…

- Setting up blocks, metadata, lists, etc. (500 LoC)
- Finding and allocating the right blocks (500 LoC)
- Updating heap structure on frees (500 LoC)

=

```
[dalud@angelshark:~/.../15213/s17/malloclabcheckpoint-handout] $ ./mdriver
Found benchmark throughput 13056 for cpu type Intel(R)Xeon(R)CPUE5520@2.27

Throughput targets: min=6528, max=11750, benchmark=13056
......................
Results for mm malloc:
  valid    util      ops    msecs    Kops  trace
   yes    78.1%       20    0.004    5595 ./traces/syn-array-short.rep
   yes     3.2%       20    0.004    5273 ./traces/syn-struct-short.rep
 * yes    96.0%    80000   17.176    4658 ./traces/syn-array.rep
 * yes    93.2%    80000    6.154   12999 ./traces/syn-mix.rep
 * yes    86.4%    80000    3.717   21521 ./traces/syn-string.rep
 * yes    85.6%    80000    3.649   21924 ./traces/syn-struct.rep
16 16     74.2% 1148359   55.949   20525

Average utilization = 74.2%. Average throughput = 20525 Kops/sec
Perf index = 60.0 (util) + 40.0 (thru) = 100.0/100
```

In reality…

- Setting up blocks, metadata, lists, etc. (500 LoC)
- Finding and allocating the right blocks (500 LoC)
- Updating heap structure on frees (500 LoC)
- **+ Some bug hiding in those 1500 LoC…**

=

```
[dalud@angelshark:~/.../15213/s17/malloclabcheckpoint-handout] $ ./mdriver
Found benchmark throughput 13056 for cpu type Intel(R)Xeon(R)CPUE5520@2.27

Throughput targets: min=6528, max=11750, benchmark=13056
.....Segmentation fault
[dalud@angelshark:~/.../15213/s17/malloclabcheckpoint-handout] $
```

Debugging Strategies

- Use **`gdb`**!
- Write a heap checker!
  - Checks heap invariants
  - Call around major operations to make sure heap invariants aren't violated.
- Assertions (like 122!):
  - **`dbg_assert(...)`**

Common Errors

- ***Garbled Bytes***
  - This means you're overwriting data in an allocated block.
- ***Overlapping Payloads***
  - This means you have unique blocks whose payloads overlap in memory
- **`segfault`**!
  - This means something is accessing invalid memory.
- For all of the above, step through with **`gdb`** to see where things start to break!
  - Note: to run assert statements, you'll need to run **`./mdriver-dbg`** rather than **`./mdriver`**.

Using gdb: Breakpoints and Watchpoints

- *Breakpoints:*
  - `break coalesce_block`
  - `break mm.c:213`
  - `break find_fit if size == 24`
- *Watchpoints:*
  - `w block = 0x8000010`
  - `w *0x15213`
  - `rwatch <thing>` – stop on *reading* a memory location
  - `awatch <thing>` – stop on *any* access to the location

Using gdb: Inspecting Frames

```
(gdb) backtrace  #0  find_fit (...)
#1  mm_malloc (...)
#2  0x0000000000403352 in eval_mm_valid (...)  #3  run_tests (...)
#4  0x0000000000403c39 in main (...)
```

- **`backtrace`** - print call stack up until current function
- **`frame 1`**: switch to mm_malloc's stack frame
  - Can then inspect local variables.

Writing a Heap Checker

- Heap checker: a function that loops over your heap/data structures and makes sure *invariants* are satisfied.
  - Returns **`true`** *if and only if* heap is well-formed.
- Critical for debugging!
  - Update when your implementation changes.
- Worry about *correctness*, not efficiency.
  - But *do* avoid printing excessively.
- For Checkpoint, *you will be graded on the quality of your heap checker*. View the writeup for more details!

Strategy Guide: Suggested Roadmap

Suggested Roadmap

- First: read the write-up!
  - "Roadmap to Success" section

0. Start writing your heap checker!
1. Implement `coalesce_block()` *first*.
2. Implement an *explicit free list*.
3. Implement *segregated lists*!
4. Further optimizations (in this order)
   - Footer Removal in allocated blocks
   - Decrease minimum block size
   - Compress Headers (hard)

Checkpoint

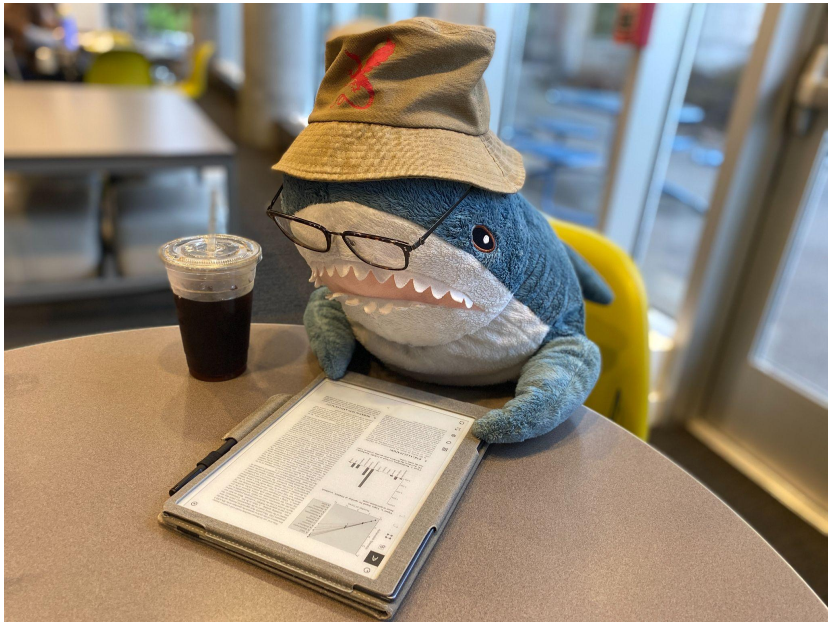
Final

Note: Using `git`

- As we have seen:
  - This is a difficult lab.
  - You will experiment with different optimizations, with varying effects on performance and thus, your score.
- Make sure to regularly checkpoint your code with commits, and push it to GitHub!
  - Don't want to lose your progress.
  - It will be helpful to include performance metrics in your commit messages.

Wrapping Up

- `malloc` due dates:
  - Checkpoint: ***October 28th***
  - Final: ***November 4th***
  - Start early!
- In class midterm: ***October 21st***
- **`cachelab`**: Watch your inbox for an email from your code review TA!
- Have a good Fall Break :-)



The End