CMSC 216
Introduction to Computer Systems
Lecture 25
C Preprocessor

Section 10.9, Bryant and O'Hallaron

Heap Management

Implementing dynamic allocation

Where does memory for a program's heap come from?

- the program (algorithms of the library functions that implement the heap) makes calls to the OS (system calls) to add memory to the heap of a running process (move the top of heap pointer)
  - on Linux, this is **`void *sbrk(int incr)`**
    - adds at least **`incr`** bytes to the end of the program's data segment
  - the key idea is to get big chunks of memory from the OS and then hand out small regions to the program on demand
    - OS calls can be a hundred times slower than local subroutines
- How is the memory in the heap managed?
  - data structures need to be kept to:
    - keep track of what memory is in use, and
    - keep track of where the free memory regions are
  - calls to **`malloc()`**, **`free()`**, etc., update these data structures

Heap Manager Requirements

- The heap manager must be able to handle arbitrary sequences of requests to allocate and deallocate memory
  - other than the constraint that a **`free()`** on an address can't occur before it was returned by an earlier call to **`malloc()`**, **`calloc()`**, or **`realloc()`**
- It must respond immediately
  - it can't wait for other requests (can’t buffer the requests)
- All data it uses must be stored in the heap itself
  - it can't use any auxiliary data structures
- It must be *aligned* to be able to hold any data
  - it can round up to the next natural alignment if needed
- It can't modify already-allocated blocks by changing their size, location, etc.
  - pointers in the program would be invalidated by doing this
  - it must only modify free blocks of memory

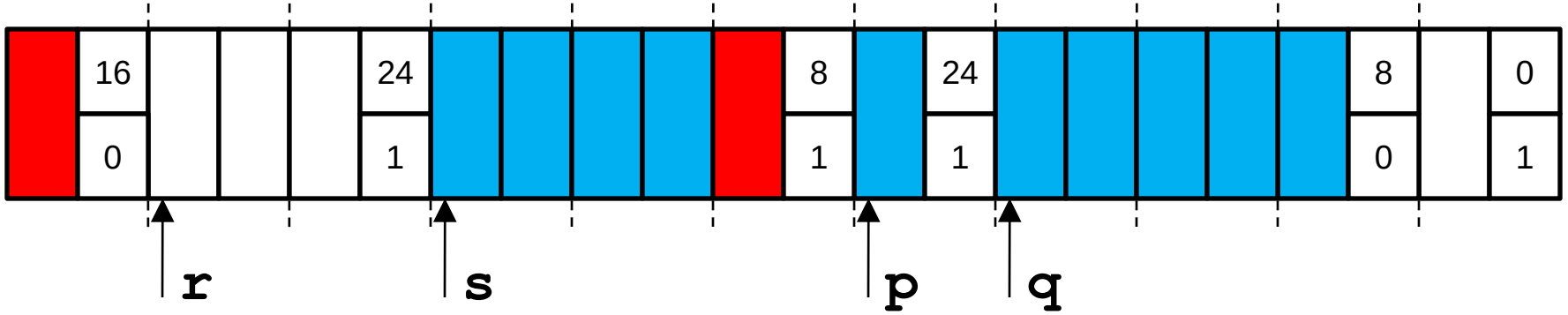
Heap Manager Goals

- Maximize throughput
  - each request needs to be fast
  - but it's often OK if the average is fast, yet some calls might be slow
- Minimize wasted space
  - what is the peak utilization of memory?
  - how much space is wasted at the peak?
  - minimize *fragmentation*
    - *internal fragmentation* - an allocated memory region is larger than the data being stored in it, due to alignment requirements or other reasons
    - *external fragmentation* - the heap has enough free space for an allocation request, but it's split up into small blocks such that no single block is large enough to satisfy the request

A simple heap structure

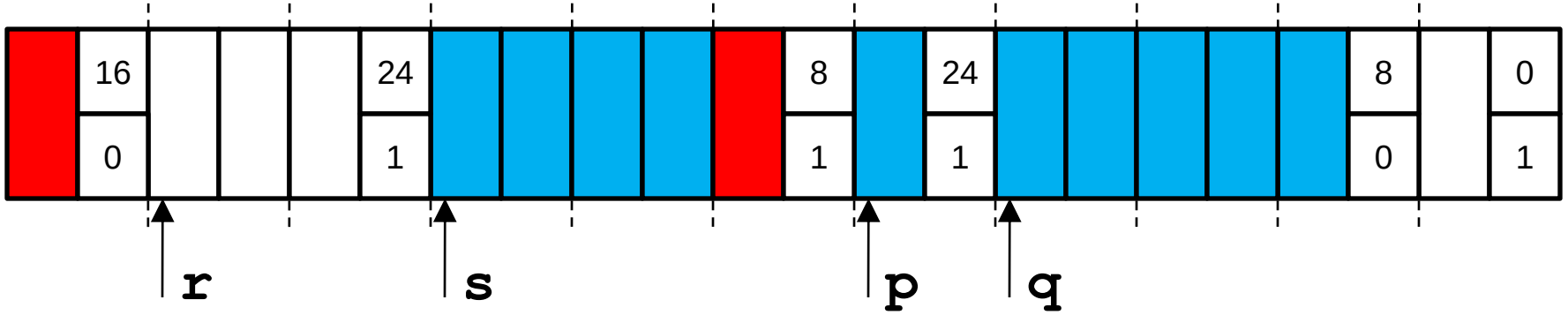
- Divide the heap into blocks
- heap blocks:
  - header (32 bits)
  - allocated block (multiple of 8, double-word aligned)
  - optional padding for alignment or other reasons
- 4-byte header:
  - 29 bits for size of block (why 29?)
  - 1 bit to indicate free/allocated
  - 2 bits available for other purposes; we won't use them
  - size is size of heap block, not allocated block
  - end of heap marked by header w/ size = 0

A simple heap structure, cont.



- each cell above is 4 bytes
- headers: size and used flag
- blue areas: allocated blocks
- red areas: wasted space due to alignment
- How can we find the address of the next block?
- What kinds of things could be stored at the areas pointed to by the pointers?

A simple heap structure, cont.



- Above could be generated by this sequence:

```
r = malloc(8);
s = malloc(16);
p = malloc(4);
q = malloc(20);
free(r);
```

Using our simple heap

- Allocation
  - Find a free block in which our request will fit
  - Placement policy
    - first fit: tends to retain large free blocks at end, but small free blocks at beginning
    - best fit: good space optimization, but performance hit
  - Split the free block?
    - depends on amount of internal fragmentation that would be introduced
- Deallocation:
  - find the header, mark it free

Coalescing free blocks

- After a deallocation, we may find ourselves with consecutive free blocks
  - "false fragmentation"
- How do we make these into one?
  - finding next block is easy
  - finding previous block is not
    - linear search for headers
    - or include footers, but waste space; can we save some space?
- Immediate vs. deferred coalescing

Dangers of dynamic memory

- What if we free something that's not in the heap?
- What happens if we free something that wasn't allocated?
- What happens if you run off the end of a dynamically allocated array? How about an array on the stack?

Nonstandard facilities for heap debugging

- Heap inspection functions
  - need to use **`#include <malloc.h>`**

  `void malloc_stats(void);`
    - prints a summary of information about heap, including current memory available and amount currently allocated

  `struct mallinfo mallinfo(void);`
    - **`uordblks`** field contains total allocated space
    - **`fordblks`** field contains total free space
    - to see the structure definition:

      `less /usr/include/malloc.h`

Checking for memory leaks

- This code checks to see if any extra heap memory is used after a call to a given function:

```
long before, after;
before = mallinfo().uordblks;
function();
after = mallinfo().uordblks;
if (before != after)
  printf("Memory leak?\n");
```

More nonstandard heap debugging

- Automatic checking of heap integrity
  - set environment variable **`MALLOC_CHECK_`** to 1; in tcsh, that's:

    ```
    setenv MALLOC_CHECK_ 1
    ```

  - or compile with **`-lmcheck`**
  - automatic memory checking can detect certain kinds of errors, like writing outside of dynamically allocated regions
  - only performs checking when allocation functions are called - only find out problems exist, not where caused

More nonstandard heap debugging

- Manual checking of heap integrity
  - Need to **`#include <mcheck.h>`**
  - call **`mcheck(NULL)`** before any allocations
    - turns on same consistency checks as before
    - also allows calling **`mprobe()`** on any pointer to the heap, forcing a consistency check on that block; returns something other than **`MCHECK_OK`** for invalid blocks
  - calling **`mcheck_pedantic(NULL)`** instead of **`mcheck(NULL)`** causes entire heap to be checked instead of just those being accessed by allocation functions
  - **`mcheck_check_all(void)`** checks whole heap immediately

Heap debugging

- This was just a summary of the most common operations
  - `gcc` has many other parameters to allow tuning of memory allocation, logging of allocations, etc.
- There are also tools available to check these kinds of issues
  - Linux has open-source **valgrind**
  - Commercial products exist for other platforms (Purify, Insure)