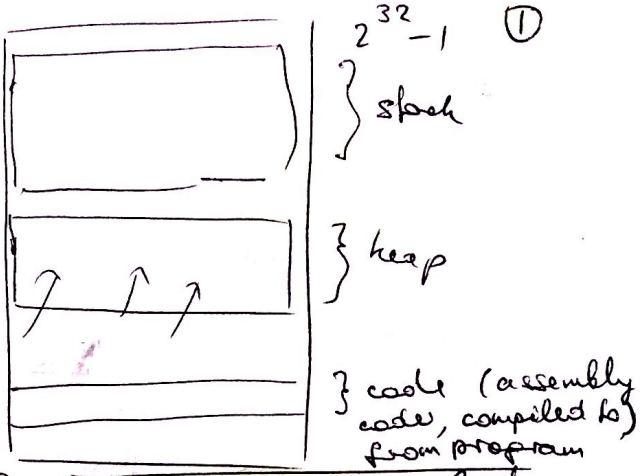
Lecture 8 CS107

Heap management

```
int * arr = malloc(40 * sizeof(int));
```



①

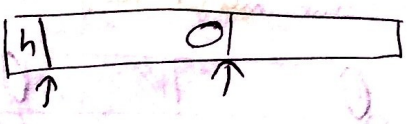
when free or realloc receives a pointer it assumes that the pointer was previously returned by malloc or realloc

heap is managed by malloc, realloc, and free

In the above case, > 160 Bytes are allocated.
- header with information about size of node
- header can be 4, 8 ... Bytes
- pointer is not to the header but to [redacted] after header

ex: allocated more than needed

```
int * arr = malloc (100 * sizeof (int));
```




```
free (arr + 60 * sizeof (int));
```

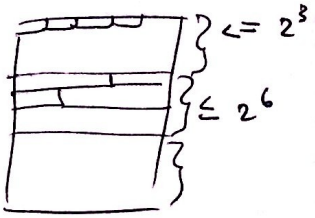
cannot do

will look at 4,8.. Bytes prior and interpret as header

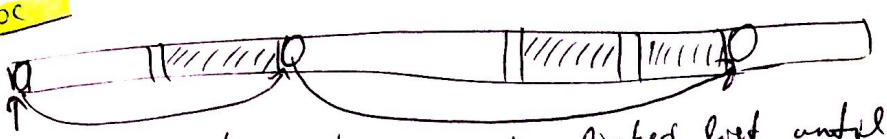
```
int array[160];
free (array);
```

There a different implementations of malloc that use different heuristics.

e.g. map into size categories and allocate within size category-subdivided heap



malloc



one heuristic: traverse the linked list until the first block is found that satisfies an allocation request

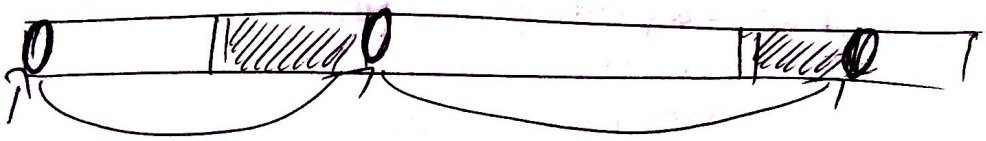
another heuristic: exhaustive search to find a block that is close to an allocation request

third heuristic: find the biggest block that satisfies an allocation request

free

- some implementations leave the header space and use it to point to next free node

- some merge adjacent free nodes

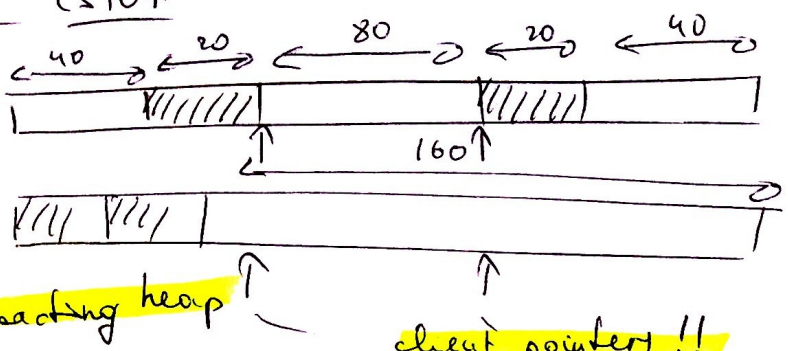


bit patterns stay after freeing.

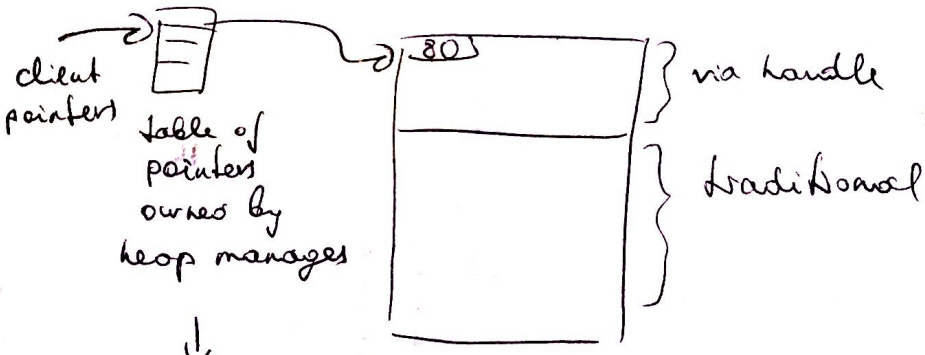
other heuristics that bind free actions to subsequent malloc or realloc calls

lecture 8 CS107

Problem



solution: two hops away from data



↓

heap compacting

```
void **handle = NewHandle(40);
{
HandleLock(handle);   <- so that pointers are locked during use
{
HandleUnlock(handle);
```

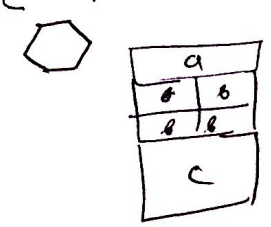
Stack management

the active segment is roughly proportional to the stack depth of currently executed functions

```
void A()
{
  int a;
  short b[4];
  double c;

  B();
  C();
}
```

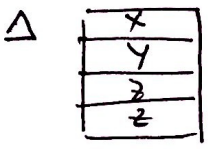
for variables
↓
← space taken from stack



activation record or stack frame for A function

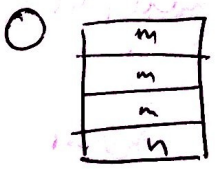
when A is called the stack pointer is decremented by the space of the ⬡ figure

```
void B()
{
  int x;
  char *y;
  char *z[2]
}
```

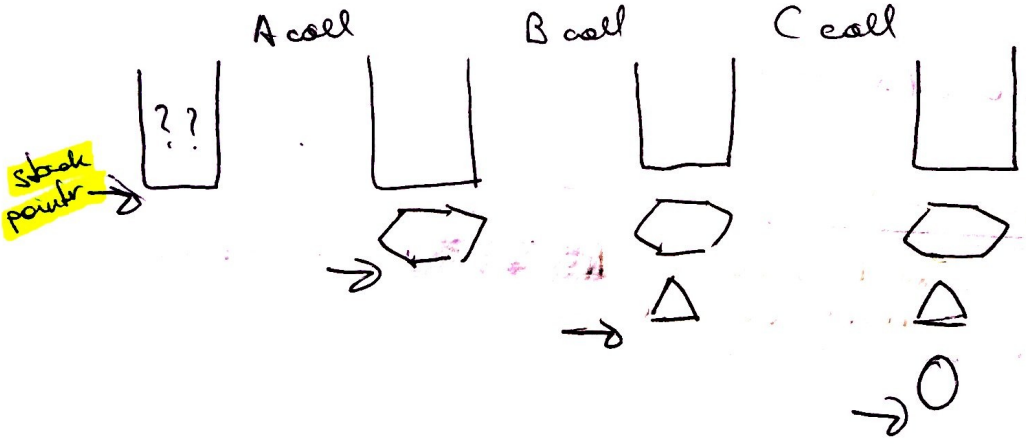


activation record of B

```
void C()
{
  double m[3];
  int n;
}
```

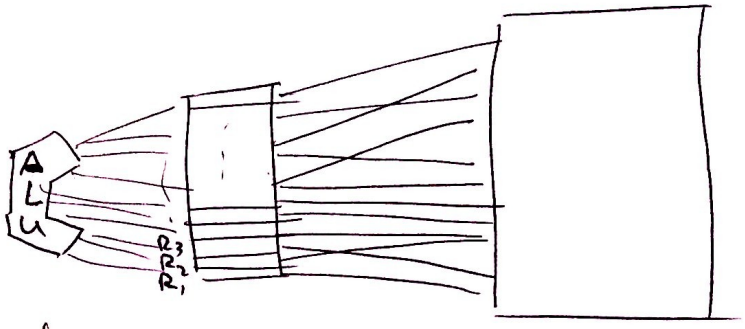


activation record of C



lecture 8 CS107

Processor organization



arithmetic logic unit: addition, multiplication, bit shifting, masking

fast access registers

slow access RAM

- get data from RAM (heap,)
- do arithmetic with registers & stack
- put results in RAM from registers

assembly: - move bytes to/from RAM
- arithmetic

my comment

see cache oblivious algs for asymptotics