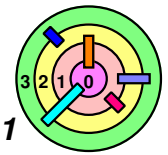
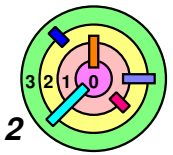
# Housekeeping (Lecture 10 - 9/30/2013)

- Warmup #2 due at 11:45pm on Friday, 10/4/2013
  - if you have code from a previous semester, be very careful and *not copy any code from it*
    - it's best if you just get rid of it
  - get started soon
- *Grading guidelines* is the *ONLY* way we will grade and we will grade on `nunki.usc.edu` in our grading account (which you don't have access to)
  - it's a good idea to run your code against the grading guidelines
- After submission, make sure you *Verify Your Submission*
- Have you installed *Ubuntu 11.10* on your laptop/desktop?
- Do you have partners for kernel assignments?
  - work with your potential partners for warmup 2
    - again, work at high level and must *not* share code



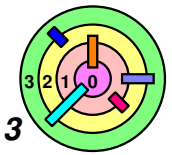
# Housekeeping (Lecture 10 - 9/30/2013)

- Warmup #2 note
  - for the *average number of packets in Q1* statistics, only consider packets that have been served by the server
    - I contradicted myself when I said that you should only consider packets that have made it into Q2
    - that's inconsistent with the spec, I apologize
    - please stick to the spec



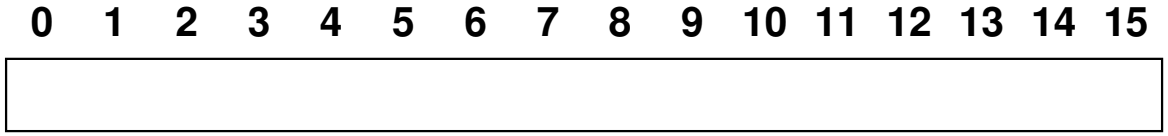
# Buddy Systems

- Data Structure
  1) doubly-linked list (not circular) `FREE` list indexed by *k*
     - links stored in actual blocks
     - `FREE`[*k*] points to first available block of size $2^k$
  2) each block contains
     - `in-use` bit
     - `size`
     - `NEXT` and `PREV` links for `FREE` list
- Can get greater variety in block sizes using Fibonacci sequence of block sizes so $b_j = b_{j-1}+b_{j-2}$
  - ratio of successive block sizes is *2/3* instead of *1/2*



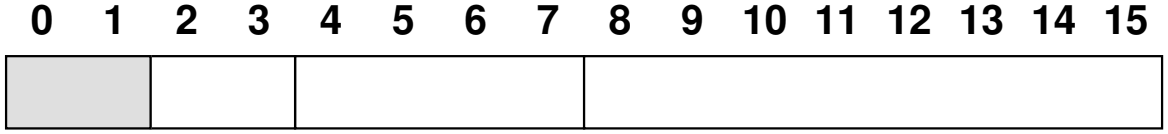
# Example of Buddy Algorithm

- Ex: 16 locations



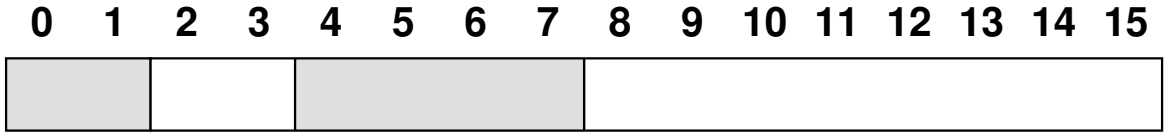

| k | free[k] |
|---|---|
| 0 | Ω |
| 1 | Ω |
| 2 | Ω |
| 3 | Ω |
| 4 | 0 |

1) allocate a block of size 2

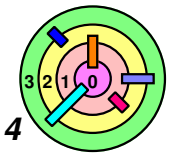



| k | free[k] |
|---|---|
| 0 | Ω |
| 1 | ~~0~~ 2 |
| 2 | ~~0~~ 4 |
| 3 | ~~0~~ 8 |
| 4 | ~~0~~ Ω |

2) allocate a block of size 4

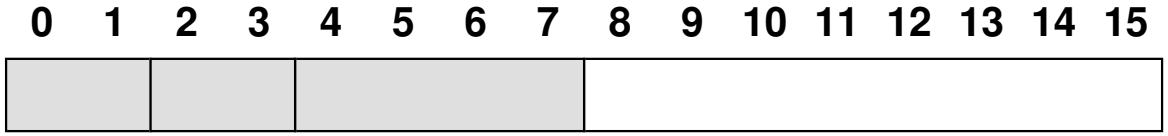



| k | free[k] |
|---|---|
| 0 | Ω |
| 1 | ~~0~~ 2 |
| 2 | ~~0~~ ~~4~~ Ω |
| 3 | ~~0~~ 8 |
| 4 | ~~0~~ Ω |



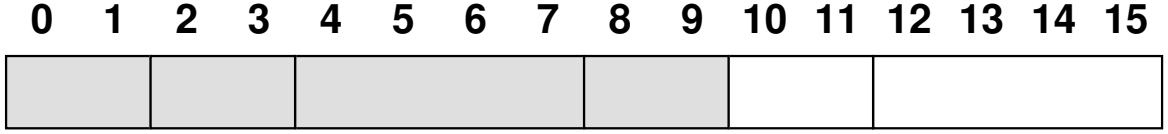
# Example of Buddy Algorithm

## 3) allocate a block of size 2

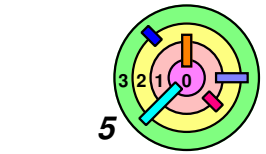



| k | free[k] |
|---|---|
| 0 | Ω |
| 1 | ~~Ω~~ ~~2~~ Ω |
| 2 | ~~Ω~~ ~~4~~ Ω |
| 3 | ~~Ω~~ 8 |
| 4 | ~~0~~ Ω |

## 4) allocate a block of size 2

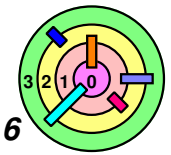



| k | free[k] |
|---|---|
| 0 | Ω |
| 1 | ~~Ω~~ ~~2~~ 10 |
| 2 | ~~Ω~~ ~~4~~ 12 |
| 3 | ~~Ω~~ ~~8~~ Ω |
| 4 | ~~0~~ Ω |

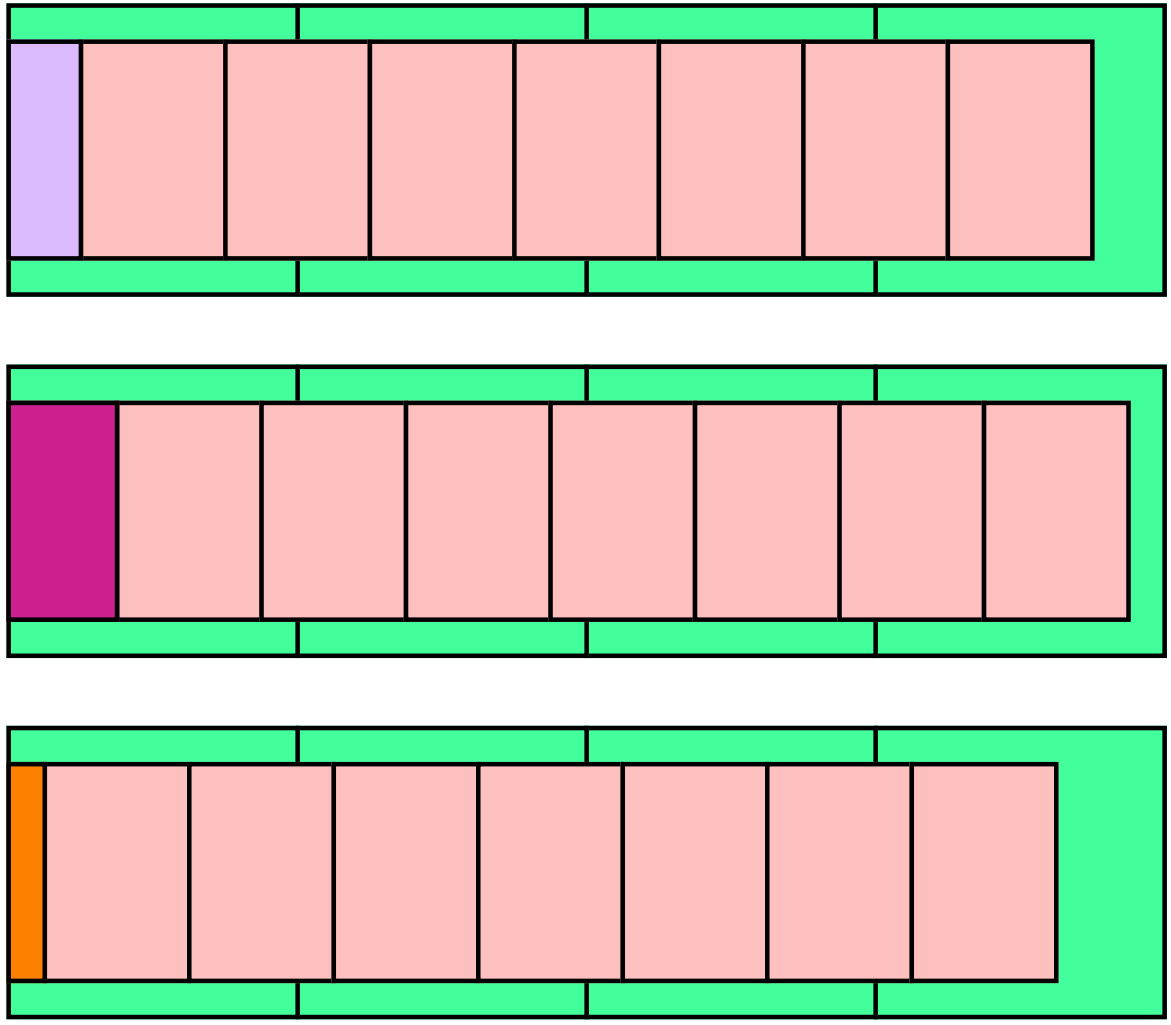


# 3.3 Dynamic Storage Allocation

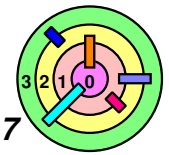
- Best-fit & First-fit Algorithms
- Buddy System
- *Slab Allocation*



# Slab Allocation

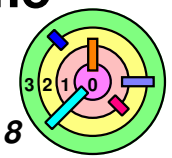


- see weenix kernel code!



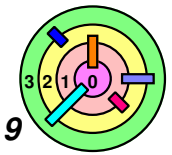
# Slab Allocation

- Objects are allocated and freed frequently
  - allocation involves
    - finding an appropriate-sized storage
    - initialize it
      - pointers need to point at the right places
      - may even need to initialize synchronization data structures
  - deallocation involves
    - tearing down the data structures
    - freeing the storage
  - lots of "overhead"
- Difficulties with dynamic storage allocation
  - you cannot predict what an application will ask for
  - but it's not true for the kernel
    - e.g., can allocate a slab of process control blocks at a time
      - return one of them from a slab



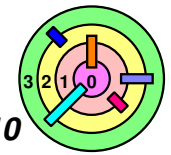
# Slab Allocation

- ***Slab Allocation***
  - sets up a separate cache for each type of object to be managed
  - contiguous sets of pages called ***slabs***, allocated to hold objects
    - we will cover "pages" later, won't get into too much detail now
- Whenever a ***slab*** is allocated, a constructor is called to initialize all the objects it hold
  - this is where you pay for initialization, but it's done in a ***batch***
- As ***objects*** are being allocated, they are taken from the set of existing slabs in the cache
  - objects are considered "preallocated" since they have all been initialized already
- As ***objects*** are being freed, they are simply marked as free
  - don't have to free up storage
  - when appropriate can free up an entire slab



# 3.4 Linking & Loading

- *Static Linking & Loading*
- Shared Libraries



# Remember This?

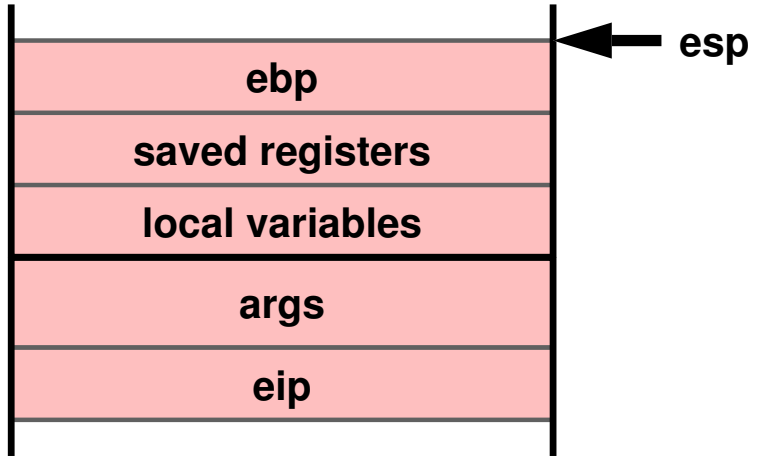
```
main:
   pushl %ebp
   movl %esp, %ebp
   pushl %esi
   pushl %edi
   subl $8, %esp
   ...
   pushl $1
   movl -12(%ebp), %eax
   pushl %eax
   call sub
   addl $8, %esp
   movl %eax, -16(%ebp)
   ...
   addl $8, %esp
   movl $0, %eax
   popl %edi
   popl %esi
   movl %ebp, %esp
   popl %ebp
   ret
```

set up stack frame

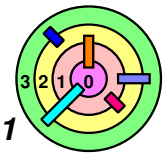
push args

pop args; get result

set return value and restore frame

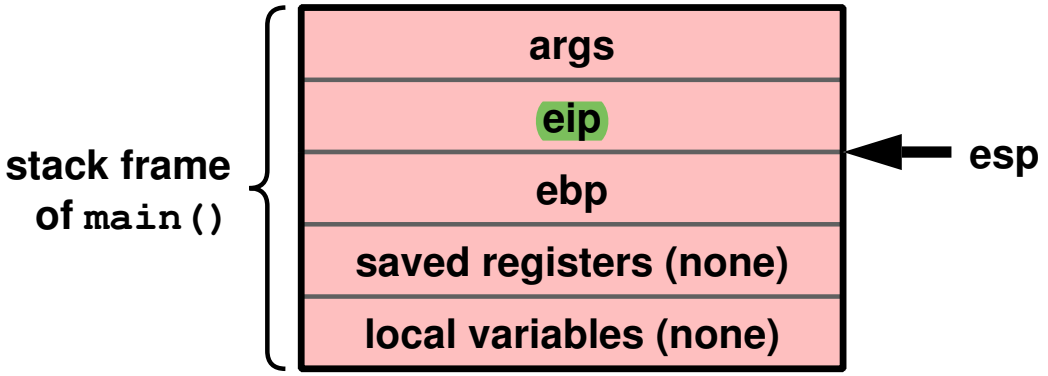



```
int main() {
   int i;
   int a;
   ...
   i = sub(a, 1);
   ...
   return(0);
}
```



# Something Simpler

```
int main(int argc,
         char *[]) {
  return(argc);
}
```

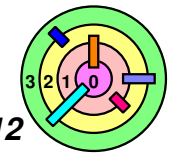



```
main:
  pushl %ebp
  movl %esp, %ebp
  movl 8(%ebp), %eax
  movl %ebp, %esp
  popl %ebp
  ret
```

set up stack frame

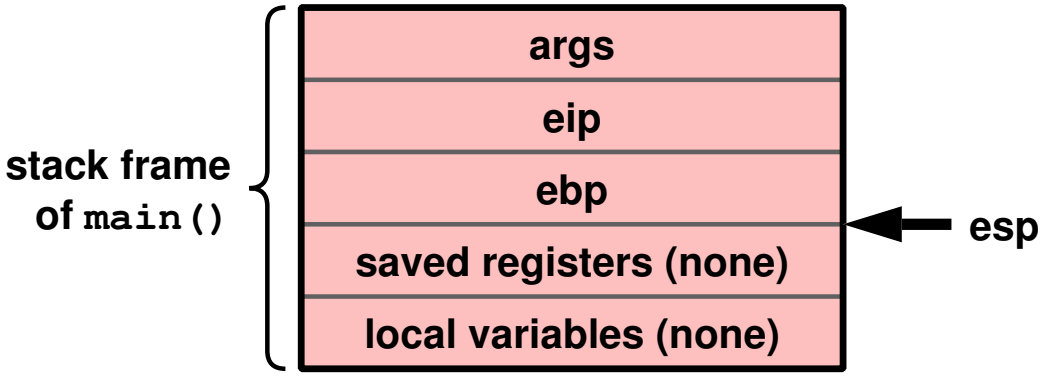
set return value and restore frame

⇨ **Does location matter?**



# Something Simpler

```
int main(int argc,
         char *[]) {
  return(argc);
}
```

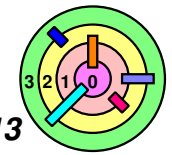



```
main:
  pushl %ebp
  movl %esp, %ebp
  movl 8(%ebp), %eax
  movl %ebp, %esp
  popl %ebp
  ret
```

set up stack frame

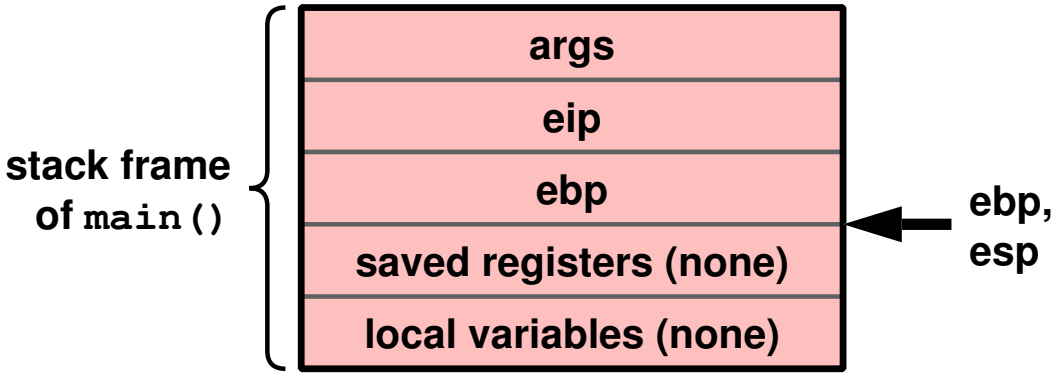
set return value and restore frame

⇨ **Does location matter?**



# Something Simpler

```
int main(int argc,
         char *[]) {
  return(argc);
}
```

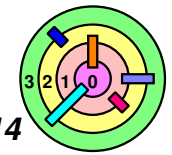



```
main:
  pushl %ebp
  movl %esp, %ebp
  movl 8(%ebp), %eax
  movl %ebp, %esp
  popl %ebp
  ret
```

set up stack frame

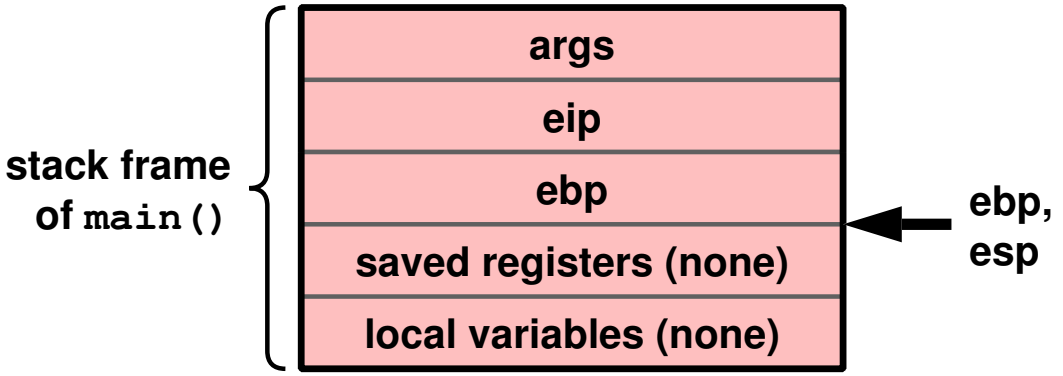
set return value and restore frame

⇨ **Does location matter?**



# Something Simpler

```
int main(int argc,
         char *[]) {
  return(argc);
}
```

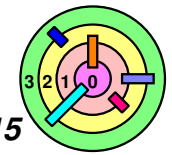



```
main:
  pushl %ebp
  movl %esp, %ebp
  movl 8(%ebp), %eax
  movl %ebp, %esp
  popl %ebp
  ret
```

set up stack frame

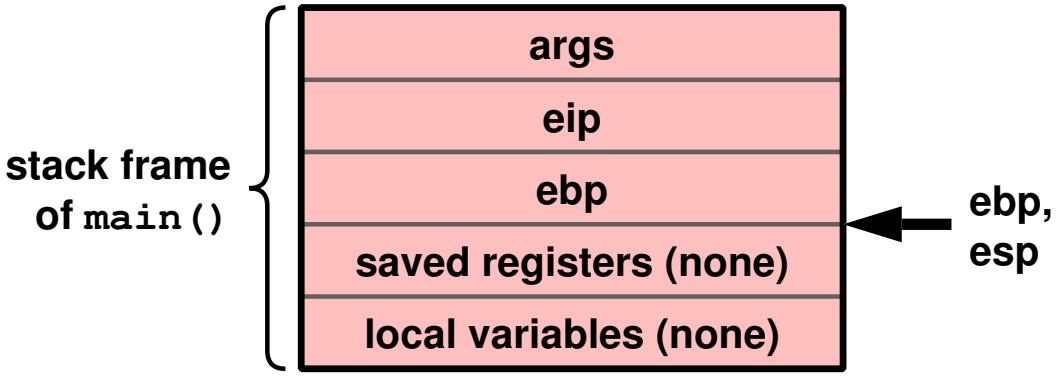
set return value and restore frame

⇨ **Does location matter?**



# Something Simpler

```
int main(int argc,
         char *[]) {
  return(argc);
}
```

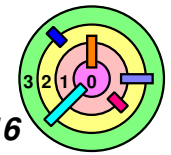



```
main:
  pushl %ebp
  movl %esp, %ebp
  movl 8(%ebp), %eax
  movl %ebp, %esp
  popl %ebp
  ret
```

set up stack frame

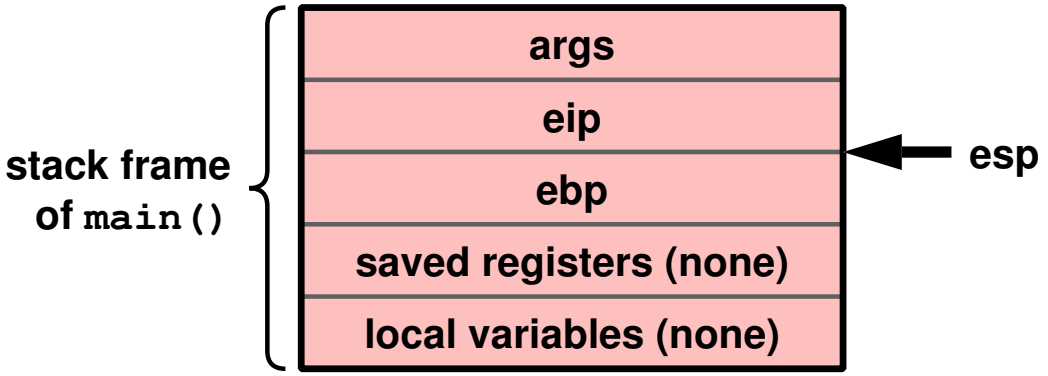
set return value and restore frame

⇨ **Does location matter?**



# Something Simpler

```
int main(int argc,
         char *[]) {
  return(argc);
}
```

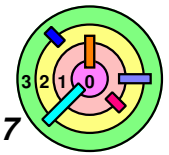



```
main:
  pushl %ebp
  movl %esp, %ebp
  movl 8(%ebp), %eax
  movl %ebp, %esp
  popl %ebp
  ret
```

set up stack frame

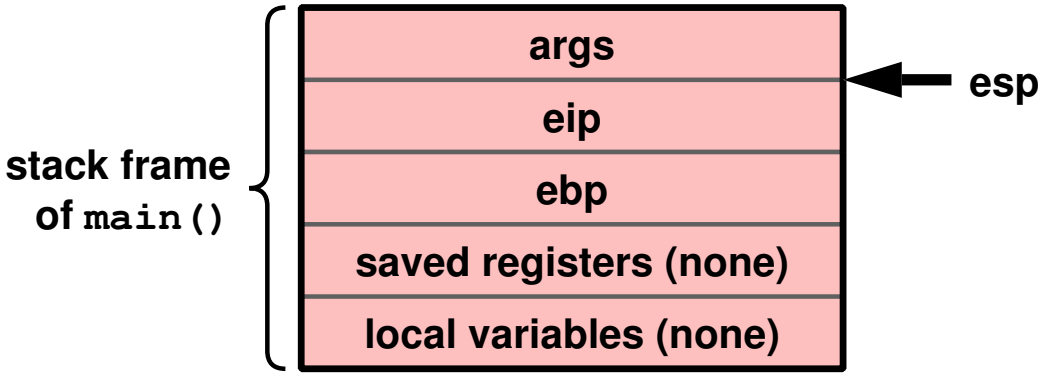
set return value and restore frame

Does location matter?



# Something Simpler

```
int main(int argc,
         char *[]) {
  return(argc);
}
```

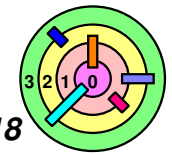



```
main:
  pushl %ebp
  movl %esp, %ebp
  movl 8(%ebp), %eax
  movl %ebp, %esp
  popl %ebp
  ret
```

set up stack frame

set return value and restore frame

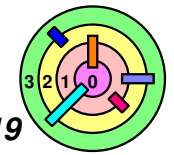
- Does location matter?
  - if everything can be accessed relative to the ***frame pointer***, then you don't need to know the actual address of an object
    - just use relative-addresses



# Location Matters ...

```
int X = 6;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = X;
  subr(y);
  return(0);
}

void subr(int i) {
  printf("i = %d\n", i);
}
```

- Why does it matter here?
  - need to put the address of `X` into `aX`
    - what's the address of `X`?
    - remember, both `X` and `aX` are in the *data segment*
    - *who* would put the actual value into `aX`?

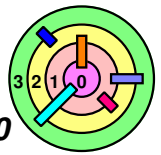


# Coping

- Relocation
  - modify internal references in memory depending on where module is expected to be *loaded*
    - one of the *exec* system calls loads a program into memory
      - everything is laid out carefully in memory
  - modules requiring relocation are said to be *relocatable*
  - the act of modifying such a module to *resolve these references* is called *relocation*
  - the program that performs relocation is called a *linker*

textbook is wrong

- Two main functions of a *linker*
  1) relocation
  2) symbol resolution
- A *loader* loads a program into memory
  - a "relocating loader" may perform additional relocation



# A Slight Revision

```
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```

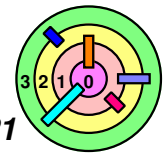
main.c

```
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```

subr.c

```
% gcc -o prog main.c subr.c
```

- `main.c` is compiled into `main.o`
- `subr.c` is compiled into `subr.o`
- `ld` is then invoked to combine them into `prog`
  - `ld` knows where to find `printf()`
  - `prog` can be loaded into memory through one of the *exec* system calls



# A Slight Revision

```
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```

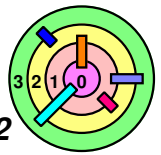
main.c

```
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```

subr.c

```
% gcc -o prog main.c subr.c
```

- how does `ld` decides what needs to be done?
- `main.c` contains undefined references to `X` and `subr()`
  - instructions for doing this are provided in `main.o`
- later on, when the actual locations for these are determined, `ld` will modify them when `main.o` is *copied* into `prog`



# A Slight Revision

```
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```

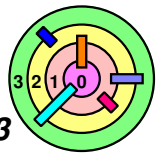
main.c

```
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```

subr.c

```
% gcc -o prog main.c subr.c
```

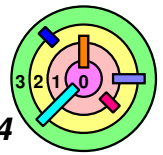
- `main.o` must contains a list of external symbols, along with their types, and instructions for updating this code



# main.s

```
Offset    Op      Arg
0:        .data ; what follows is initialized data
0:        .globl aX ; aX is global: it may be used
                    ; by others
0:  aX:
0:        .long  X
4:
0:        .text ; offset restarts; what follows is
                ; text (read-only code)
0:        .globl main
0:  main:
0:        pushl %ebp ; save the frame pointer
1:        movl  %esp,%ebp ; point to
3:        subl  $4,%esp ; make space
6:        movl  aX,%eax ; put conten
11:       movl  (%eax),%eax ; put *X
13:       movl  %eax,-4(%ebp) ; stor
16:       pushl -4(%ebp) ; push y on
19:       call  subr
24:       addl  $4,%esp ; remove y f
27:       movl  $0,%eax ; set return
31:       movl  %ebp, %esp ; restore
33:       popl  %ebp  ; pop frame pointer
35:       ret
```

```
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```

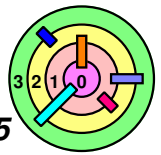


# main.s

```
Offset    Op      Arg
0:        .data ; what follows is initialized data
0:        .globl aX ; aX is global: it may be used
                    ; by others
0:  aX:
0:        .long  X
4:
0:        .text ; offset restarts; w
                ; text (read-only co
0:        .globl main
0:  main:
0:        pushl %ebp ; save the frame pointer
1:        movl  %esp,%ebp ; point to
3:        subl  $4,%esp ; make space
6:        movl  aX,%eax ; put conten
11:       movl  (%eax),%eax ; put *X
13:       movl  %eax,-4(%ebp) ; stor
16:       pushl -4(%ebp) ; push y on
19:       call  subr
24:       addl  $4,%esp ; remove y f
27:       movl  $0,%eax ; set return
31:       movl  %ebp, %esp ; restore
33:       popl  %ebp  ; pop frame pointer
35:       ret
```

What follows goes into the *data* segment

```
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```

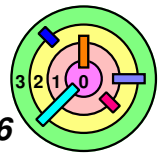


# main.s

```
Offset    Op      Arg
0:        .data ; what follows is initialized data
0:        .globl aX ; aX is global: it may be used
                    ; by others
0:  aX:
0:        .long  X
4:
0:        .text ; offset restarts; w
                ; text (read-only co
0:        .globl main
0:  main:
0:        pushl %ebp ; save the frame pointer
1:        movl  %esp,%ebp ; point to
3:        subl  $4,%esp ; make space
6:        movl  aX,%eax ; put conten
11:       movl  (%eax),%eax ; put *X
13:       movl  %eax,-4(%ebp) ; stor
16:       pushl -4(%ebp) ; push y on
19:       call  subr
24:       addl  $4,%esp ; remove y f
27:       movl  $0,%eax ; set return
31:       movl  %ebp, %esp ; restore
33:       popl  %ebp  ; pop frame pointer
35:       ret
```

What follows goes into the *text* segment

```
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```

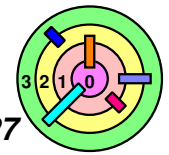


# main.s

```
Offset    Op      Arg
0:        .data ; what follows is initialized data
0:        .globl aX ; aX is global: it may be used
                    ; by others
0:  aX:
0:        .long  X
4:
0:        .text ; offset restarts; w
                ; text (read-only co
0:        .globl main
0:  main:
0:        pushl %ebp ; save the frame pointer
1:        movl  %esp,%ebp ; point to
3:        subl  $4,%esp ; make space
6:        movl  aX,%eax ; put conten
11:       movl  (%eax),%eax ; put *X
13:       movl  %eax,-4(%ebp) ; stor
16:       pushl -4(%ebp) ; push y on
19:       call  subr
24:       addl  $4,%esp ; remove y f
27:       movl  $0,%eax ; set return
31:       movl  %ebp, %esp ; restore
33:       popl  %ebp  ; pop frame pointer
35:       ret
```

offset got restarted because segments are relocatable

```
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```



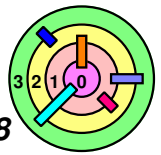
# main.s

```
Offset    Op      Arg
0:        .data ; what follows is initialized data
0:        .globl aX ; aX is global: it may be used
                    ; by others
0:  aX:
0:        .long  X
4:
0:        .text ; offset restarts; w
                ; text (read-only co
0:        .globl main
0:  main:
0:        pushl %ebp ; save the frame pointer
1:        movl  %esp,%ebp ; point to
3:        subl  $4,%esp ; make space
6:        movl  aX,%eax ; put conten
11:       movl  (%eax),%eax ; put *X
13:       movl  %eax,-4(%ebp) ; stor
16:       pushl -4(%ebp) ; push y on
19:       call  subr
24:       addl  $4,%esp ; remove y f
27:       movl  $0,%eax ; set return
31:       movl  %ebp, %esp ; restore
33:       popl  %ebp  ; pop frame pointer
35:       ret
```

`.global` directive means that the symbol mentioned is defined *here* and is *exported*

- i.e., can be referenced by other modules
- `aX` and `main` are global

```
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```



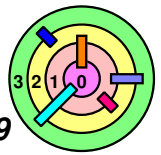
# main.s

```
Offset    Op      Arg
0:        .data ; what follows is initialized data
0:        .globl aX ; aX is global: it may be used
                    ; by others
0:  aX:
0:        .long  X
4:
0:        .text ; offset restarts; w
                ; text (read-only co
0:        .globl main
0:  main:
0:        pushl %ebp ; save the frame pointer
1:        movl  %esp,%ebp ; point to
3:        subl  $4,%esp ; make space
6:        movl  aX,%eax ; put conten
11:       movl  (%eax),%eax ; put *X
13:       movl  %eax,-4(%ebp) ; stor
16:       pushl -4(%ebp) ; push y on
19:       call  subr
24:       addl  $4,%esp ; remove y f
27:       movl  $0,%eax ; set return
31:       movl  %ebp, %esp ; restore
33:       popl  %ebp  ; pop frame pointer
35:       ret
```

aX is 4 bytes long and put the value of x here

- x will remain unresolved

```
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```

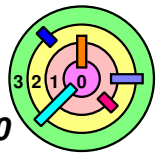


# main.s

```
Offset    Op      Arg
0:        .data ; what follows is initialized data
0:        .globl aX ; aX is global: it may be used
                    ; by others
0:  aX:
0:        .long  X
4:
0:        .text ; offset restarts; w
                ; text (read-only co
0:        .globl main
0:  main:
0:        pushl %ebp ; save the frame pointer
1:        movl  %esp,%ebp ; point to
3:        subl  $4,%esp ; make space
6:        movl  aX,%eax ; put conten
11:       movl  (%eax),%eax ; put *X
13:       movl  %eax,-4(%ebp) ; stor
16:       pushl -4(%ebp) ; push y on
19:       call  subr
24:       addl  $4,%esp ; remove y f
27:       movl  $0,%eax ; set return
31:       movl  %ebp, %esp ; restore
33:       popl  %ebp  ; pop frame pointer
35:       ret
```

these 3 places require relocation

```
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```



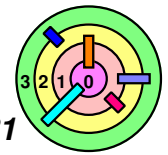
# main.s

```
Offset    Op      Arg
0:        .data ; what follows is initialized data
0:        .globl aX ; aX is global: it may be used
                    ; by others
0:  aX:
0:        .long  X
4:
0:        .text ; offset restarts; w
                ; text (read-only co
0:        .globl main
0:  main:
0:        pushl %ebp ; save the frame pointer
1:        movl  %esp,%ebp ; point to
3:        subl  $4,%esp ; make space
6:        movl  aX,%eax ; put conten
11:       movl  (%eax),%eax ; put *X
13:       movl  %eax,-4(%ebp) ; stor
16:       pushl -4(%ebp) ; push y on
19:       call  subr
24:       addl  $4,%esp ; remove y f
27:       movl  $0,%eax ; set return
31:       movl  %ebp, %esp ; restore
33:       popl  %ebp  ; pop frame pointer
35:       ret
```

this call is a PC-relative call

- what's stored at offset 20 is not the absolute address of `subr`, but a relative address

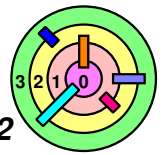
```
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```



# subr.s

```
Offset    Op      Arg
0:        .data ; what follows is initialized data
0:  printfarg:
0:        .string "i = %d\n"
8:
0:        .comm X,4 ; 4 bytes in BSS is required
                    ; for global X
4:
0:        .text ; offset restarts; what follows is
                ; text (read-only code)
0:        .globl subr
0:  subr:
0:        pushl %ebp  ; save the fra
1:        movl  %esp, %ebp ; point t
3:        pushl 8(%ebp)  ; push i on
6:        pushl $printfarg ; push ad
                           ; onto st
11:       call  printf
16:       addl  $8, %esp ; pop argum
19:       movl  %ebp, %esp ; restore stack pointer
21:       popl  %ebp  ; pop frame pointer
23:       ret
```

```c
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```



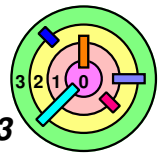
# subr.s

```
Offset    Op      Arg
0:        .data ; what follows is initialized data
0:  printfarg:
0:        .string "i = %d\n"
8:
0:        .comm X,4 ; 4 bytes in BSS
                    ; for global X
4:
0:        .text ; offset restarts; w
                ; text (read-only co
0:        .globl subr
0:  subr:
0:        pushl %ebp  ; save the fra
1:        movl  %esp, %ebp ; point t
3:        pushl 8(%ebp)  ; push i on
6:        pushl $printfarg ; push ad
                           ; onto st
11:       call  printf
16:       addl  $8, %esp ; pop argum
19:       movl  %ebp, %esp ; restore stack pointer
21:       popl  %ebp  ; pop frame pointer
23:       ret
```

this is how you create a string constant

- this one is 8 bytes long
- and local to this module (since it's not global)

```
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```



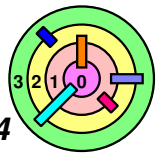
# subr.s

```
Offset    Op      Arg
0:        .data ; what follows is initialized data
0:  printfarg:
0:        .string "i = %d\n"
8:
0:        .comm X,4 ; 4 bytes in BSS
                    ; for global X
4:
0:        .text ; offset restarts; w
                ; text (read-only co
0:        .globl subr
0:  subr:
0:        pushl %ebp  ; save the fra
1:        movl  %esp, %ebp ; point t
3:        pushl 8(%ebp)  ; push i on
6:        pushl $printfarg ; push ad
                           ; onto st
11:       call  printf
16:       addl  $8, %esp ; pop argum
19:       movl  %ebp, %esp ; restore stack pointer
21:       popl  %ebp  ; pop frame pointer
23:       ret
```

this is how you create a string constant

- this one is 8 bytes long
- and local to this module (since it's not global)
- it is used here

```
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```

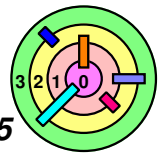


# subr.s

```
Offset    Op      Arg
0:        .data ; what follows is initialized data
0:  printfarg:
0:        .string "i = %d\n"
8:
0:        .comm X,4 ; 4 bytes in BSS
                    ; for global X
4:
0:        .text ; offset restarts; w
                ; text (read-only co
0:        .globl subr
0:  subr:
0:        pushl %ebp  ; save the fra
1:        movl  %esp, %ebp ; point t
3:        pushl 8(%ebp)  ; push i on
6:        pushl $printfarg ; push ad
                           ; onto st
11:       call  printf
16:       addl  $8, %esp ; pop argum
19:       movl  %ebp, %esp ; restore stack pointer
21:       popl  %ebp  ; pop frame pointer
23:       ret
```

4 bytes is required in the ***bss*** segment for this global variable

```
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```

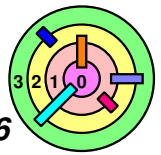


# subr.s

```
Offset    Op      Arg
0:        .data ; what follows is initialized data
0:  printfarg:
0:        .string "i = %d\n"
8:
0:        .comm X,4 ; 4 bytes in BSS
                    ; for global X
4:
0:        .text ; offset restarts; w
                ; text (read-only co
0:        .globl subr
0:  subr:
0:        pushl %ebp  ; save the fra
1:        movl  %esp, %ebp ; point t
3:        pushl 8(%ebp)  ; push i on
6:        pushl $printfarg ; push ad
                           ; onto st
11:       call  printf
16:       addl  $8, %esp ; pop argum
19:       movl  %ebp, %esp ; restore stack pointer
21:       popl  %ebp  ; pop frame pointer
23:       ret
```

`subr` is a global symbol exported from here

```
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```

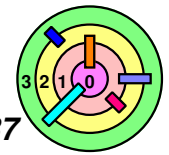


# subr.s

```
Offset    Op      Arg
0:        .data ; what follows is initialized data
0:  printfarg:
0:        .string "i = %d\n"
8:
0:        .comm X,4 ; 4 bytes in BSS
                    ; for global X
4:
0:        .text ; offset restarts; w
                ; text (read-only co
0:        .globl subr
0:  subr:
0:        pushl %ebp   ; save the fra
1:        movl  %esp, %ebp ; point t
3:        pushl 8(%ebp)   ; push i on
6:        pushl $printfarg ; push ad
                           ; onto st
11:       call  printf
16:       addl  $8, %esp ; pop argum
19:       movl  %ebp, %esp ; restore stack pointer
21:       popl  %ebp   ; pop frame pointer
23:       ret
```

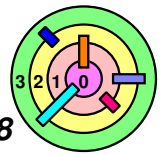
relocation is required for `printf` and `printfarg`

```
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```



# Object Files

- An object file describes what's in the data, bss, and text segments in separate sections
- Along with each section is a list of:
  - global symbols
  - undefined symbols
  - instructions for relocation
    - these instructions indicate
      - which locations within the section must be modified
      - which symbol's value is used to modify the location
    - a symbol's value is the address that is ultimately determined for it
    - typically, this address is added to the location being modified
- To inspect an object file on Unix
  - `nm` - list symbols from object files
  - `objdump` - display information from object files



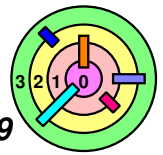
# subr.s

```
Offset    Op      Arg
0:        .data ; what follows is initialized data
0:  printfarg:
0:        .string "i = %d\n"
8:
0:        .comm X,4 ; 4 bytes in BSS
                    ; for global X
4:
0:        .text ; offset restarts; w
                ; text (read-only co
0:        .globl subr
0:  subr:
0:        pushl %ebp   ; save the fra
1:        movl  %esp, %ebp ; point t
3:        pushl 8(%ebp)   ; push i onto stack
6:        pushl $printfarg ; push address of string
                           ; onto stack
11:       call  printf
16:       addl  $8, %esp ; pop arguments from stack
19:       movl  %ebp, %esp ; restore stack pointer
21:       popl  %ebp   ; pop frame pointer
23:       ret
```

**relocation is required for:**

- `printfarg` at offset 7
- `printf` at offset 12

```
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```



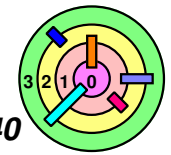
# subr.o

```
Data:
   Size:  8
   Contents: "i = %d\n"

bss:
   Size:  4
   Global: X, offset 0

Text:
   Size:  24
   Global: subr, offset 0
   Undefined: printf
   Relocation:
      offset 7, size 4, value: addr of printfarg
      offset 12, size 4, value: PC-relative addr of
                                printf
   Contents: [machine instructions]
```

**relocation is required for:**

- `printfarg` at offset 7
- `printf` at offset 12

```
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```

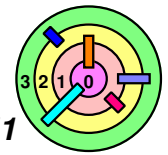


# subr.o

```
Data:
  Size:  8
  Contents: "i = %d\n"

bss:
  Size:  4
  Global: X, offset 0

Text:
  Size:  24
  Global: subr, offset 0
  Undefined: printf
  Relocation:
    offset 7, size 4, value: addr of printfarg
    offset 12, size 4, value: PC-relative addr of
                              printf
  Contents: [machine instructions]
```

X and subr are exported
- needed in main.o

```
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```

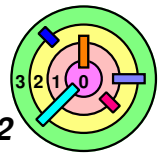


# main.s

```
Offset    Op      Arg
0:        .data ; what follows is initialized data
0:        .globl aX ; aX is global:
                    ; by others
0:  aX:
0:        .long  X
4:
0:        .text ; offset restarts; w
                ; text (read-only co
0:        .globl main
0:  main:
0:        pushl %ebp ; save the fram
1:        movl  %esp,%ebp ; point to
3:        subl  $4,%esp ; make space
6:        movl  aX,%eax ; put conten
11:       movl  (%eax),%eax ; put *X
13:       movl  %eax,-4(%ebp) ; stor
16:       pushl -4(%ebp) ; push y onto stack
19:       call  subr
24:       addl  $4,%esp ; remove y from stack
27:       movl  $0,%eax ; set return value to 0
31:       movl  %ebp, %esp ; restore stack pointer
33:       popl  %ebp  ; pop frame pointer
35:       ret
```

these 2 places remained unresolved

```
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```



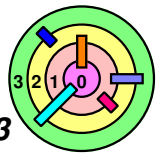
# main.o

```
Data:
  Size:  4
  Global: aX, offset 0
  Undefined: X
  Relocation: offset 0, size 4, v
  Contents: 0x00000000

bss:
  Size:  0

Text:
  Size:  36
  Global: main, offset 0
  Undefined: subr
  Relocation:
    offset 7, size 4, value: addr of aX
    offset 20, size 4, value: PC-relative
                              addr of subr
  Contents: [machine instructions]
```

these 2 places remained unresolved

- they are noted in `main.o`

```
extern int X;
int *aX = &X;

int main( ) {
  void subr(int);
  int y = *aX;
  subr(y);
  return(0);
}
```

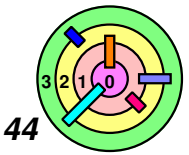


# subr.o

```
Data:
  Size:  8
  Contents: "i = %d\n"

bss:
  Size:  4
  Global: X, offset 0

Text:
  Size:  24
  Global: subr, offset 0
  Undefined: printf
  Relocation:
    offset 7, size 4, value: addr of printfarg
    offset 12, size 4, value: PC-relative addr of
                              printf
  Contents: [machine instructions]
```

`printf` remained unresolved

```
#include <stdio.h>
int X;

void subr(int i) {
  printf("i = %d\n", i);
}
```

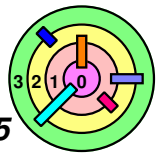


# printf.o

```
Data:
  Size:  1024
  Global: StandardFiles
  Contents: ...

bss:
  Size:  256

Text:
  Size:  12000
  Global: printf, offset 100
  ...
  Undefined: write
  Relocation:
    offset 211, value: addr of StandardFiles
    offset 723, value: PC-relative addr of write
  Contents: [machine instructions]
```

assume that `printf.o` looks like this

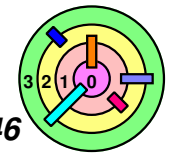
- `write` is unresolved



# write.o

```
Data:
  Size:  0

bss:
  Size:  4
  Global: errno, offset 0

Text:
  Size:  16
  Contents: [machine
             instructions]
```

and `write.o` looks like this

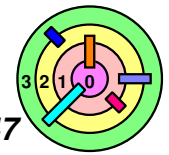


# startup function

```
Data:
  Size:  0

bss:
  Size:  0

Text:
  Size:  36
  Undefined: main
  Relocation:
    offset 21, value: main
  Contents: [machine
             instructions]
```

every C program contains a startup routine that is called first

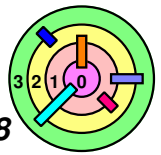
- it calls `main()`
- if `main()` returns, it calls `exit()`
- our example is incomplete



# prog

```
Text
   main              4096
   subr              4132
   printf            4156
   write            16156
   startup          16172

Data
   aX               16384
   printfargs       16388
   StandardFiles    16396

BSS
   X                17420
   errno            17680
```

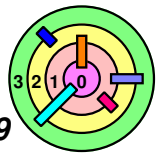
- this is how `ld` might set things up
- main does not start at location 0
  - first "page" is typically made inaccessible so that references to null pointers will fail (get SIGSEG)



# prog

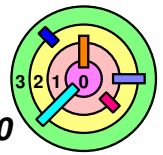
```
Text
  main              4096
  subr              4132
  printf            4156
  write            16156
  startup          16172

Data
  aX               16384
  printfargs       16388
  StandardFiles    16396

BSS
  X                17420
  errno            17680
```

- due to the use of *"pages"*, the data segment needs to start at a page boundary (i.e., multiple of page size)
  - this way, the text segment can be made read-only while the data and bss segments read-write
  - here we assume that pages start at 4096, 8192, 12288, 16384, etc.



# Virtual Memory Basics

- A process has, say, a 32-bit address space
  - that's 4GB of memory
- Our `prog` process, when it starts, only needs about 16KB for text+data+bss
  - plus more for stack
- Allocating 4GB of memory will be a huge waste
- Solution: *page table* in *virtual memory*
  - OS allocate *pages* of *physical memory* at a time
  - a page is 4KB in many systems
  - a page corresponds to physical memory that can be located *(or "mapped")* anywhere in virtual memory
  - one level of *indirection* to get to the physical memory
    - the hardware makes this transparent
- We will spend a lot of time talking about virtual memory (Ch 7)

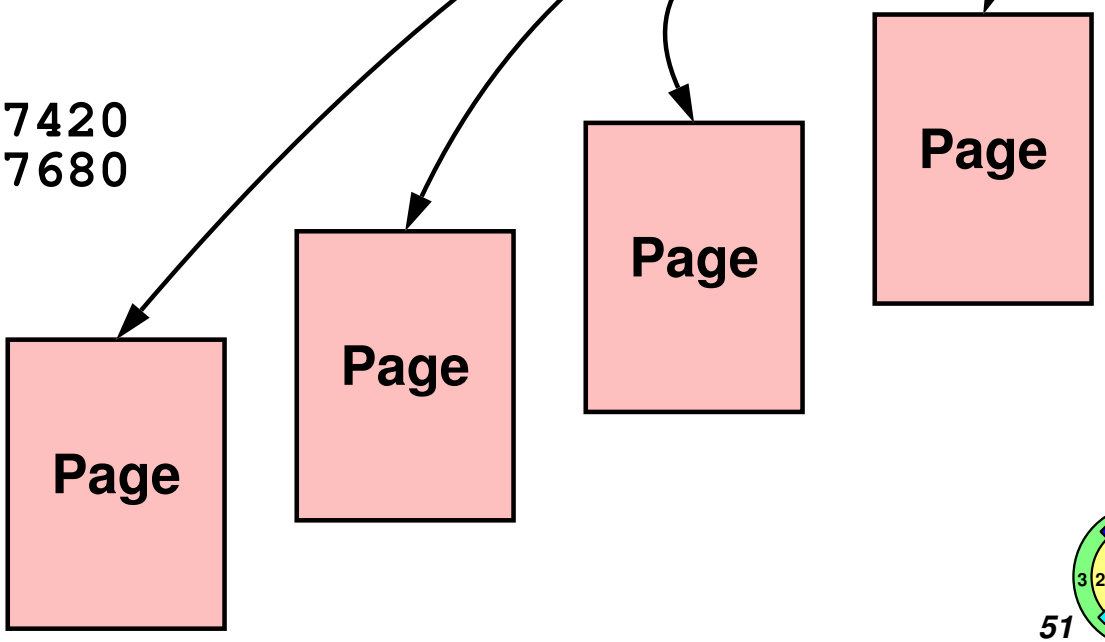


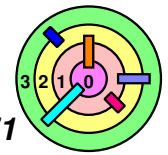
# Virtual Memory Basics

```
Text
   main              4096
   subr              4132
   printf            4156
   write            16156
   startup          16172

Data
   aX               16384
   printfargs       16388
   StandardFiles    16396

BSS
   X                17420
   errno            17680
```

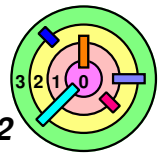
**Page Table**

| *Start* | *Access* | *Physical Addr* |
|---|---|---|
| 0 | - | - |
| 4096 | R | |
| 8192 | R | |
| 12288 | R | |
| 16384 | R/W | |



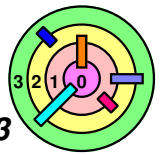



# 3.5 Booting



# Boot

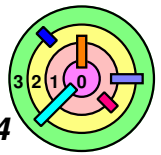
- Came from the idiomatic expression, "to pull yourself up by your bootstraps"
  - without the help of others
  - it's a difficult situation
- In OS
  - load its OS into memory
    - which kind of means that you need an OS in memory to do it
- Solution
  - load a tiny OS into memory
    - known as the *bootstrap loader*
    - then again, who loads this tiny OS into memory?
      - how about first loading a tiny bootstrap loader?



# PDP-8

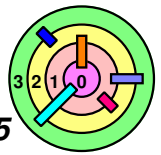

toggle switches

- How about manually put into memory a simple bootstrap loader?
  - approach taken by PDP-8
    - "toggles in" the program
  - read OS from paper tape

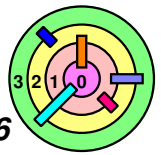


# PDP-8 Boot Code

```
07756 6032 KCC
07757 6031 KSF
07760 5357 JMP .-1
07761 6036 KRB
07762 7106 CLL RTL
07763 7006 RTL
07764 7510 SPA
07765 5357 JMP 7757
07766 7006 RTL
07767 6031 KSF
07770 5367 JMP .-1
07771 6034 KRS
07772 7420 SNL
07773 3776 DCA I 7776
07774 3376 DCA 7776
07775 5356 JMP 7756
07776 0000 AND 0
07777 5301 JMP 7701
```

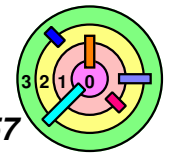


# VAX-11/780

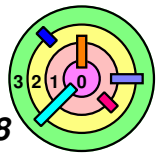
# VAX-11/780 Boot

- Separate "console computer"
  - LSI-11
  - hard-wired to always run the code contained in its on-board read-only memory
  - then read boot code (i.e., the bootstrap loader) from floppy disk
  - then load OS from root directory of first file system on primary disk
- Code on floppy disk (the bootstrap loader) would handle:
  - *disk device*
  - *on-disk file system*
  - it needs the right *device driver*
  - it needs to know how the disk is setup
    - what sort of *file system* is on the disk
    - how the disk is *partitioned*
      - a disk may hold multiple and different file systems, each in a separate partition



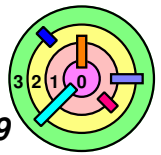
# Configuring the OS

- Early Unix
  - OS statically linked to contain all needed device drivers
    - device drivers were statically linked to the OS
  - all device-specific info included with drivers
  - disk drivers contained partitioning description
  - therefore, the following actions may all require compiling a new version of the OS:
    - adding a new device
    - replacing a device
    - modifying disk-partitioning information

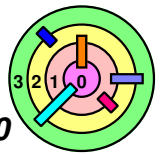


# Configuring the OS

- Later Unix
  - OS statically linked to contain all needed device drivers
  - at boot time, OS would probe to see which devices were present and discover device-specific info
  - partition table in first sector of each disk
- Even later Unix
  - allowed device drivers to be dynamically loaded into a running system

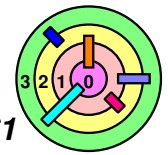


# IBM PC

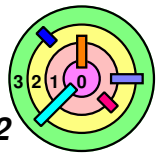
# Issues

- Open architecture
  - although MS-DOS was distributed in binary form only
  - large market for peripherals, most requiring special drivers
  - how to access boot device?
  - how does OS get drivers for new devices?



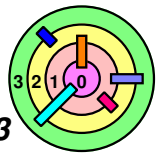
# The Answer: BIOS

- **Basic Input-Output System** ***(BIOS)***
  - **code stored in read-only memory** (ROM)
  - configuration data in non-volatile RAM (NVRAM)
    - such as ***CMOS***
  - including set of boot-device names
  - the BIOS provides three primary functions
    - **power-on self test** ***(POST)***
      - so it knows ***where*** to load the boot program, etc. into
    - **load and transfer control to boot program**
    - **provide** ***drivers*** **for all devices**
- **Main BIOS on motherboard**
  - supplied as a chip on the "motherboard"
  - contains everything necessary to perfrom the above 3 functions
  - additional BIOSes on other boards
    - provide access to additional devices

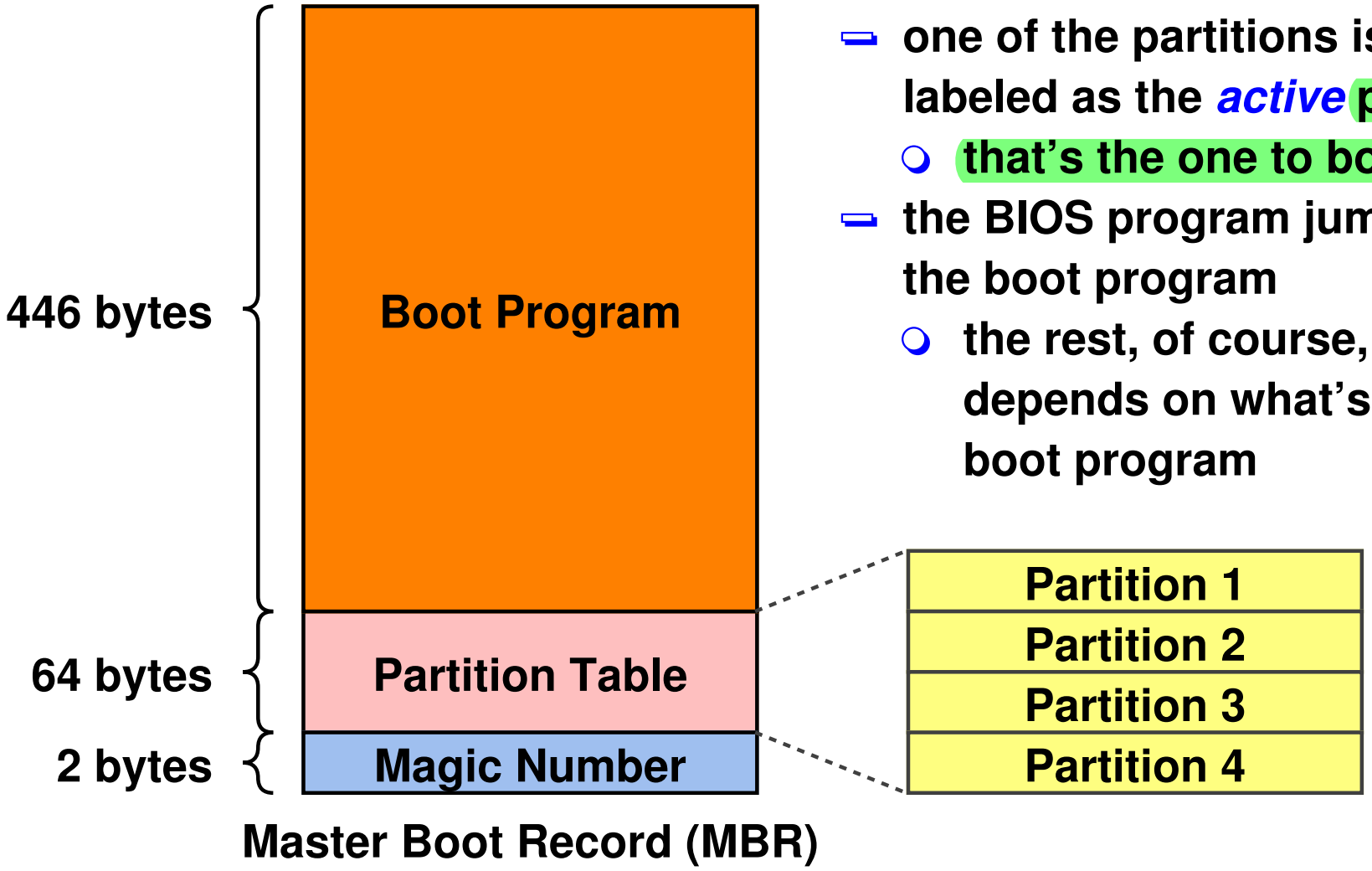


# POST

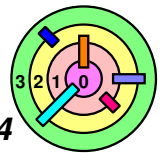
- On power-on, CPU executes BIOS code
  - located in last 64KB of first megabyte of address space
    - starting at location 0xf0000
    - CPU is hard-wired to start executing at 0xffff0 on startup
      - the last 16 bytes of this region
      - jump to POST
- POST
  - initializes hardware
  - counts memory locations
    - by testing for working memory
- Next step is to find a boot device
  - the CMOS is configured with a boot order
- Next step is to load the Master Boot Record *(MBR)* from the first sector of the boot device, if it's a floppy/diskette
  - or cylinder 0, head 0, section 1 of a hard disk (Ch 6)



# Getting the Boot Program

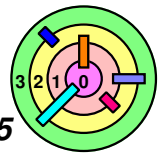



- one of the partitions is labeled as the *active* partition
  - that's the one to boot from
- the BIOS program jumps to the boot program
  - the rest, of course, depends on what's in the boot program



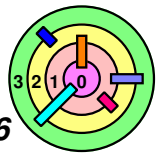
# MS-DOS Boot Program

- Find the active partition
  - load the first sector from it
    - which contains the "volume boot program"
  - pass control to that program
    - which then load the OS from that partition



# Linux Booting (1)

- Two stages of booting provided by one of:
  - *lilo* (Linux Loader)
    - uses sector numbers of kernel image
    - therefore, must be modified if a kernel image moves
  - *grub* (Grand Unified Boot Manager)
    - understands various file systems
    - can find a kernel image given a file system *path name*
  - both allow dual (or greater) booting
  - select which system to boot from menu
    - perhaps choice of Linux or Windows
- The next step is for the kernel to *configure* itself



# Linux Booting (2)

**assembler code (startup_32)**

- Kernel image is compressed
  - step 1: set up stack, clear BSS, uncompress kernel, then transfer control to it

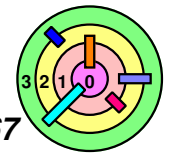
**assembler code (different startup_32)**

- *Process 0* is created
  - step 2: set up initial page tables, turn on *address translation* (Ch 7)
  - process 0 knows how to handle some aspects of paging

**C code (start_kernel)**

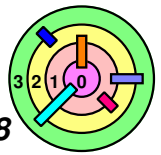
- Do further initialization
  - step 3: initialize rest of kernel, create the "init" process (i.e., *process 1*, which is the ancestor of all other user processes)
  - invoke the *scheduler*

Note: weenix is not exactly Linux



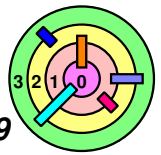
# BIOS Device Drivers

- Originally, the BIO provided drivers for all devices
  - OS would call BIOS-provided code whenever it required services of a device driver
- These drivers sat in low memory and provided minimal functionality
  - later systems would copy them into primary memory
  - even later systems would provide their own drivers
  - nevertheless, BIO drivers are still used for booting
    - how else can you do it?



# Beyond BIOS

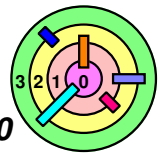
- BIOS
  - designed for 16-bit x86 of mid 1980s
  - not readily extensible to other architectures
- Open Firmware
  - designed by Sun
  - portable
  - drivers, boot code in Forth
    - compiled into bytecode
- Intel developed a replacement for BIOS called *EFI (Extensible Firmware Interface)*
  - also uses bytecode



# Ch 4: Operating-System Design

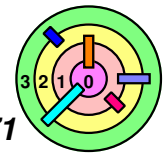
Bill Cheng

*http://merlot.usc.edu/cs402-f13*



# OS Design

- We will now look at how OSes are constructed
  - what goes into an OS
  - how they interact with each other
  - how is the software structured
  - how performance concerns are factoered in
- We will introduce new components in this chapter
  - scheduling (Ch 5)
  - file systems (Ch 6)
  - virtual memory (Ch 7)
- We will start with a simple hardware configuration
  - what OS is needed to support this
- Applications views the OS as the "computer"
  - the OS needs to provide a ***consistent*** and ***usable interface***
    - while being ***secure*** and ***efficient***
  - that's a pretty tall order!



# OS Design

- Our goal is to build a general-purpose OS
  - can run a variety of applications
    - some are interactive
    - many use network communication
    - all read/write to a file system
  - it's like most general-purpose OSes
    - Linux
    - Solaris
    - FreeBSD
    - Mac OS X
    - Chromium OS (has a Linux kernel)
    - Windows (the only one that's not directly based on Unix)
  - all these OSes are quite similar, functionally!
    they all provide:
    - processes
    - threads
    - file systems
    - network protocols with similar APIs
    - user interface with display, mouse, keyboard
    - access control based on file ownership and that file owers can control