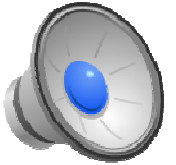




Linking

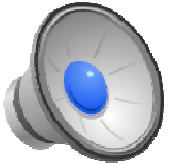
- Topics
 - How do you transform a collection of object files into an executable?
 - How is an executable structured?
 - Why is an executable structured as it is?
- Learning Objectives:
 - Explain what ELF format is.
 - Explain what an executable is and how it got that way

With huge thanks to Steve Chong for his notes from CS61.



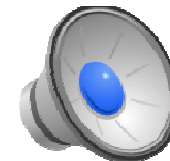
What is Linking?

- The process of merging all the code and data from multiple object files into a single file that is ready to be loaded into memory and executed.
- Two forms of linking:
 - Static: compile/link time – executable contains all the code and data needed to execute.
 - Dynamic: load time or run time – parts of the code and/or data are integrated after link time.
 - Think shared libraries.



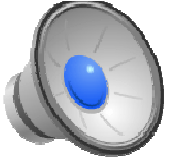
What does the Linker do?

- **Copy** code and data from each object file into the executable.
- **Resolve** references between object files
- **Relocate** symbols to use absolute addresses rather than relative addresses.



Symbols

- Symbols are a convenience for the programmer, but the compiler, assembler, linker and loader have to deal with transforming symbols into something meaningful.
- Local variables: easy – compiler translates those into references into the stack.
- Local functions: relatively straight forward – place the functions somewhere and then translate references to the functions into addresses.
- Global variables: slightly more work
 - When a module allocates space for them, the compiler has to allocate space for them and translate references to the appropriate address.
 - When a module uses them, the compiler has to leave a note indicating that at link time, we will need to figure out where those global variables live and fix up references to them.
- External functions: similar to global variables
 - Defining module: place the code and remember that you have the definition of the symbol corresponding to the function name.
 - Using module: remember that you have to fill in calls to it later.



Example

- Consider the two files: main.c and swap.c:

main.c

```
int buf[2] = {1,2};

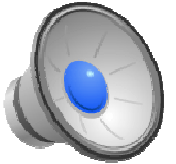
int main()
{
    swap();
    return 0;
}
```

swap.c

```
extern int buf[];

void swap()
{
    int temp;

    temp = buf[1];
    buf[1] = buf[0];
    buf[0] = temp;
}
```



Example: Local Variable

- Consider the two files: main.c and swap.c:

main.c

```
int buf[2] = {1,2};

int main()
{
    swap();
    return 0;
}
```

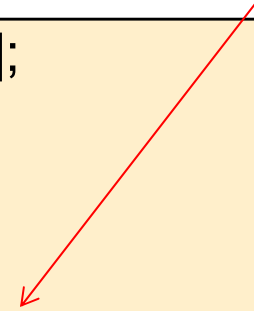
swap.c

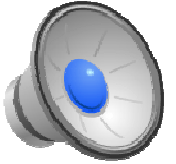
```
extern int buf[];

void swap()
{
    int temp;

    temp = buf[1];
    buf[1] = buf[0];
    buf[0] = temp;
}
```

Local variable





Example: Define Global Variable

- Consider the two files: main.c and swap.c:

main.c

```
int buf[2] = {1,2};
```

```
int main()
```

```
{
```

```
    swap();
```

```
    return 0;
```

```
}
```

Global variable declared

swap.c

```
extern int buf[];
```

```
void swap()
```

```
{
```

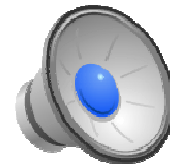
```
    int temp;
```

```
    temp = buf[1];
```

```
    buf[1] = buf[0];
```

```
    buf[0] = temp;
```

```
}
```



Example: Use of Global Variable

- Consider the two files: main.c and swap.c:

main.c

```
int buf[2] = {1,2};

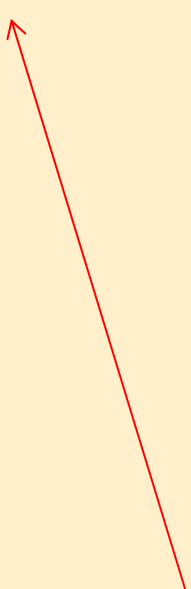
int main()
{
    swap();
    return 0;
}
```

swap.c

```
extern int buf[];

void swap()
{
    int temp;

    temp = buf[1];
    buf[1] = buf[0];
    buf[0] = temp;
}
```



Global variable referenced



Example: Define Function

- Consider the two files: main.c and swap.c:

main.c

```
int buf[2] = {1,2};

int main()
{
    swap();
    return 0;
}
```

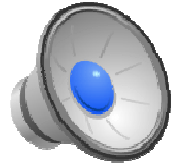
swap.c

```
extern int buf[];

void swap()
{
    int temp;

    temp = buf[1];
    buf[1] = buf[0];
    buf[0] = temp;
}
```

External function defined



Example: Use External Function

- Consider the two files: main.c and swap.c:

main.c

```
int buf[2] = {1,2};

int main()
{
    swap();
    return 0;
}
```

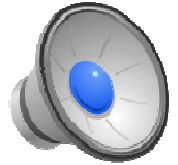
External function used

swap.c

```
extern int buf[];

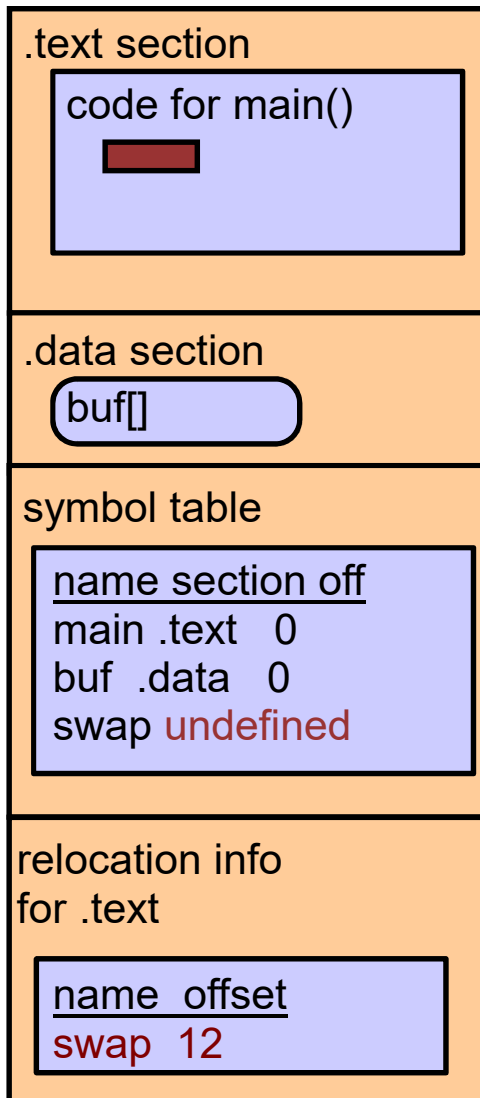
void swap()
{
    int temp;

    temp = buf[1];
    buf[1] = buf[0];
    buf[0] = temp;
}
```

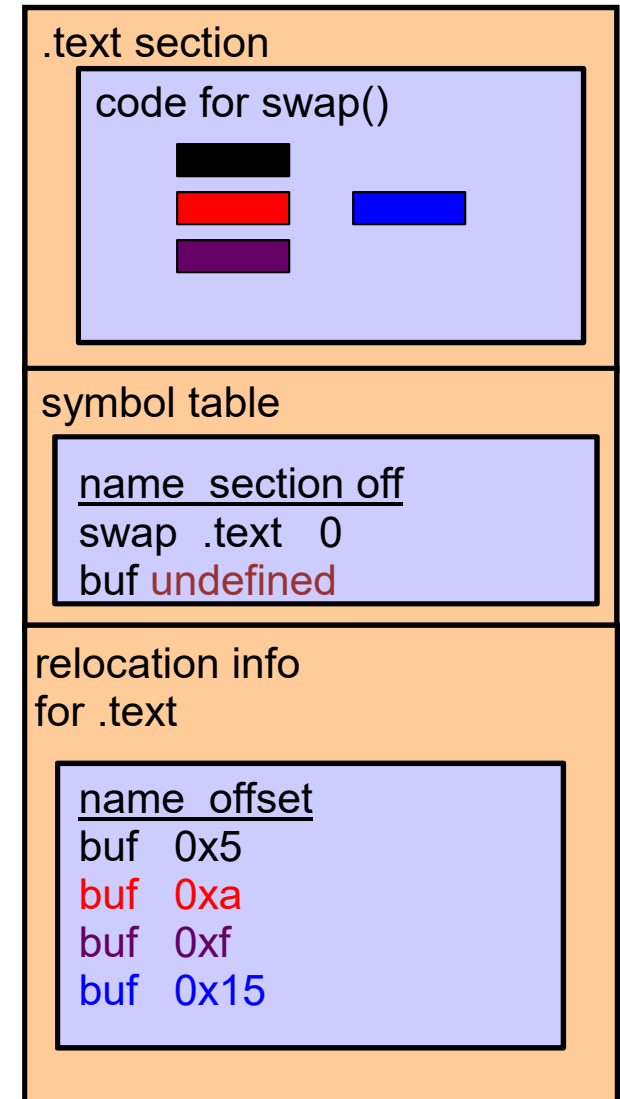


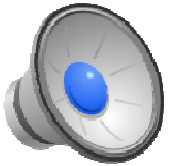
.o files

main.o

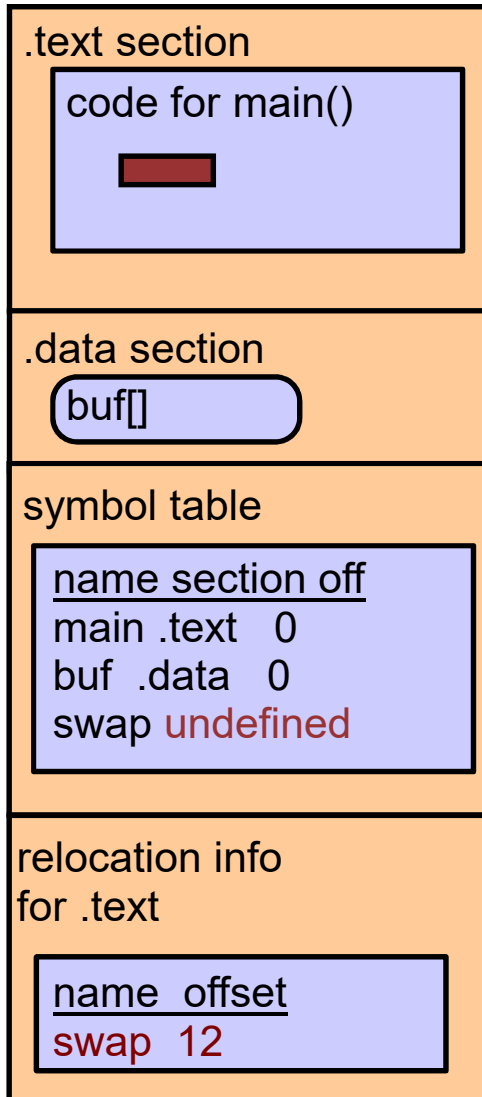


swap.o



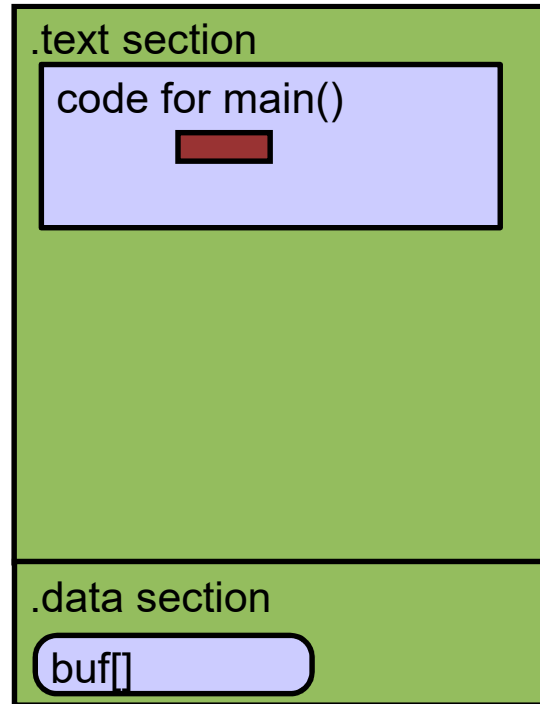


main.o

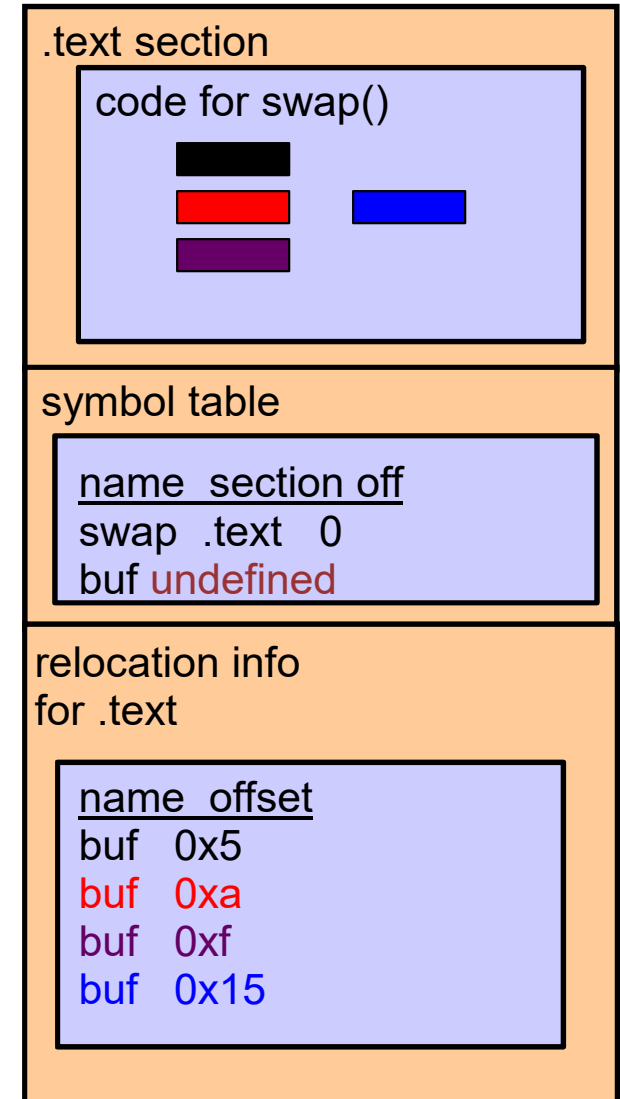


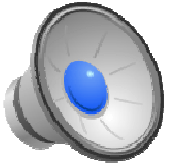
.o -> executable

myprog

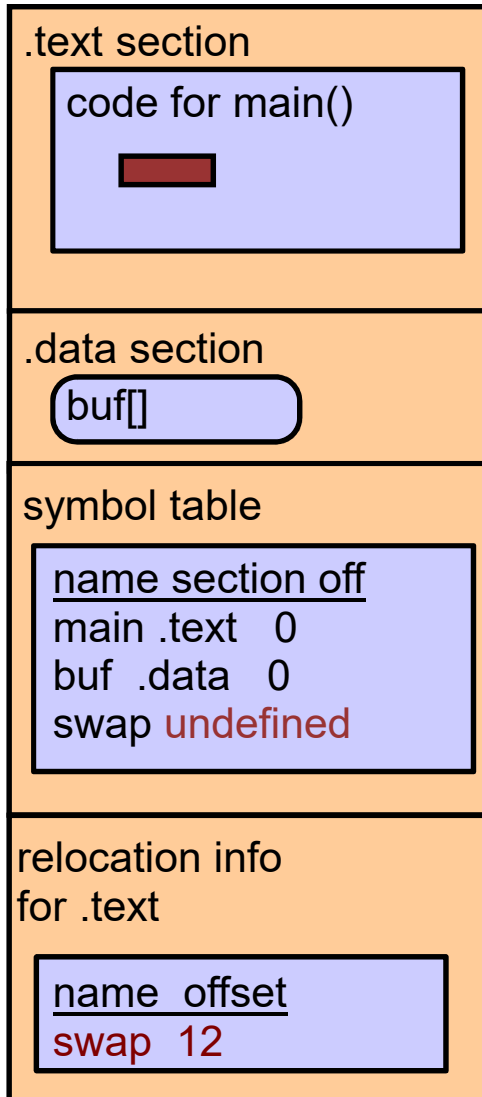


swap.o



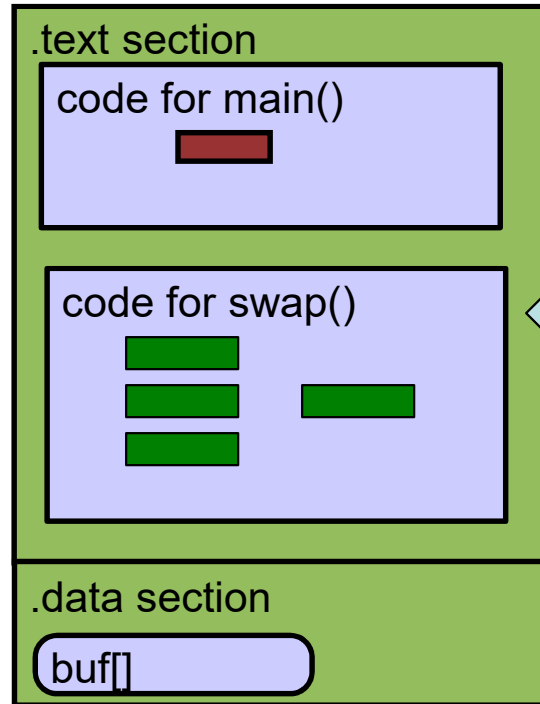


main.o

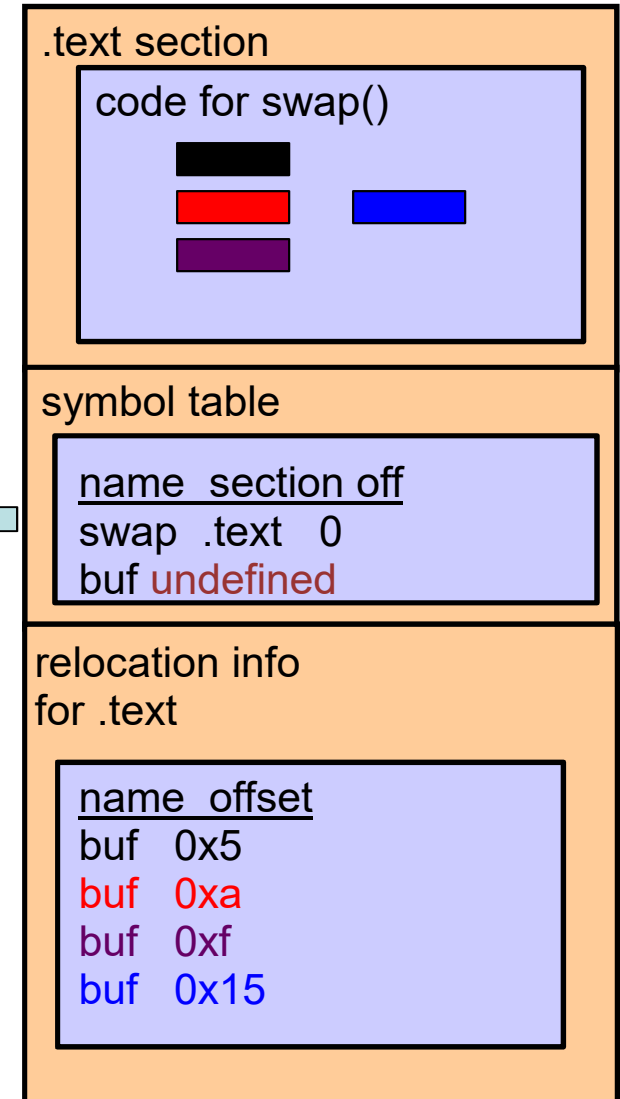


.o -> executable

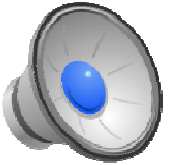
myprog



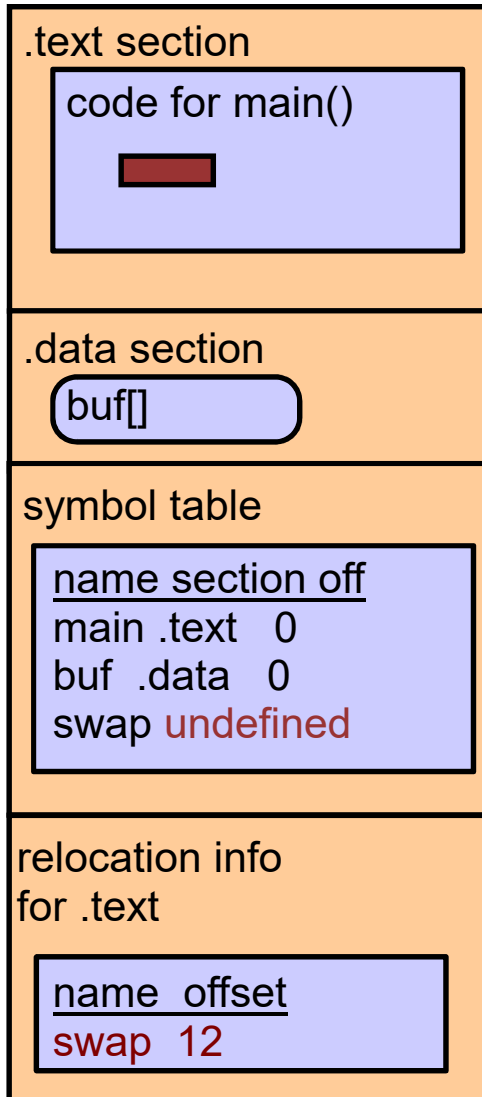
swap.o



Can now resolve references to buf

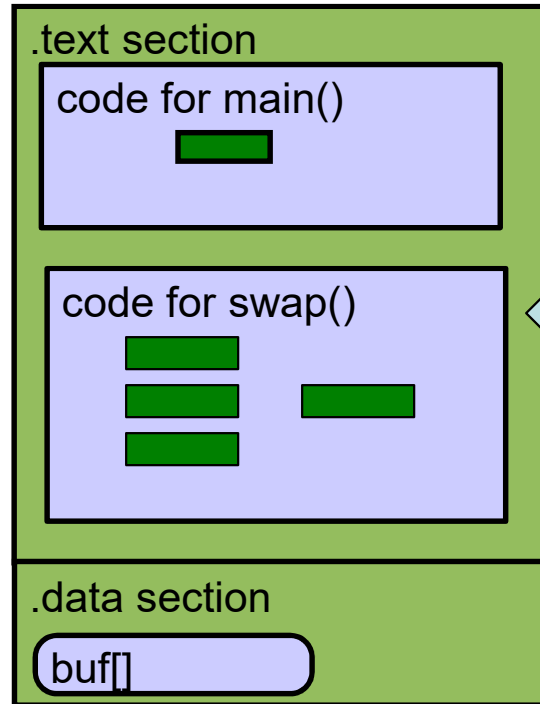


main.o

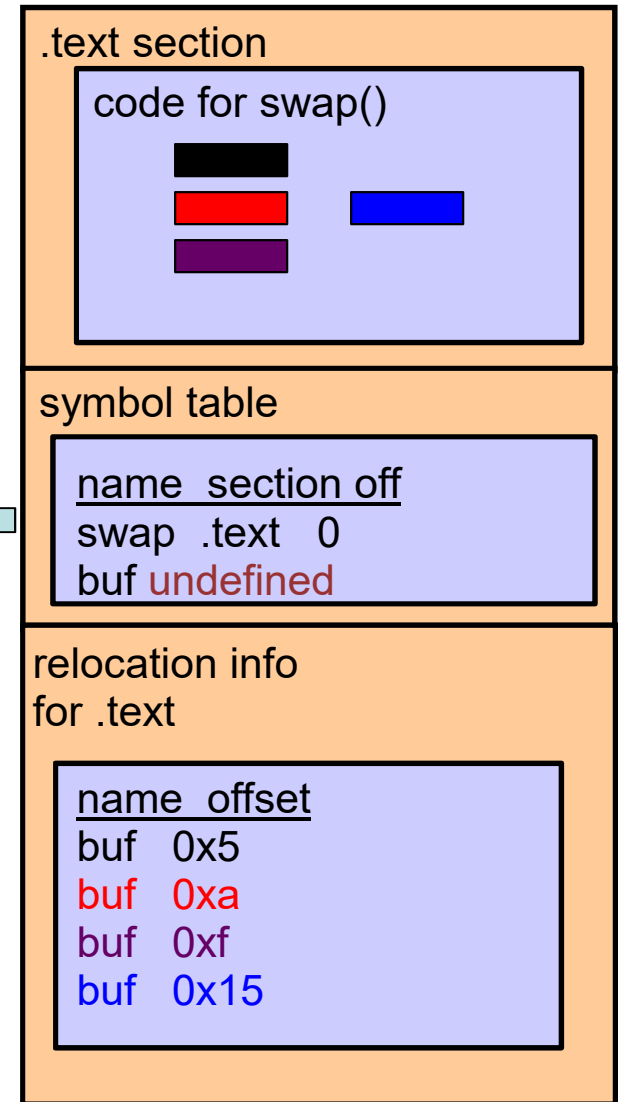


.o -> executable

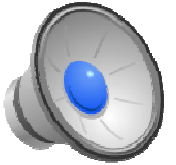
myprog



swap.o

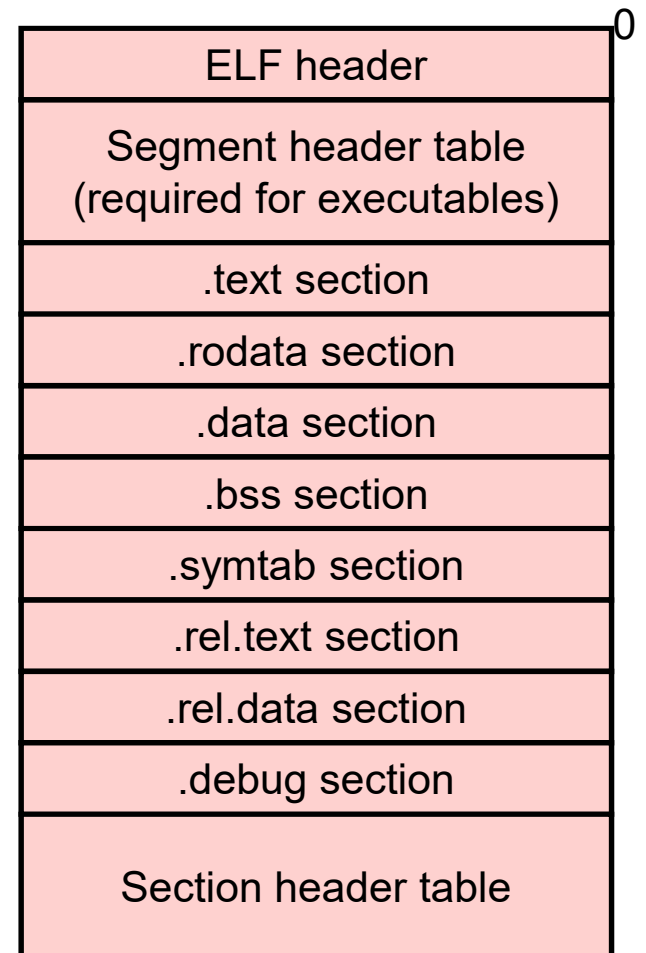


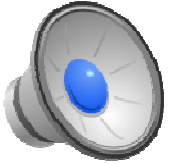
Can also resolve reference to swap



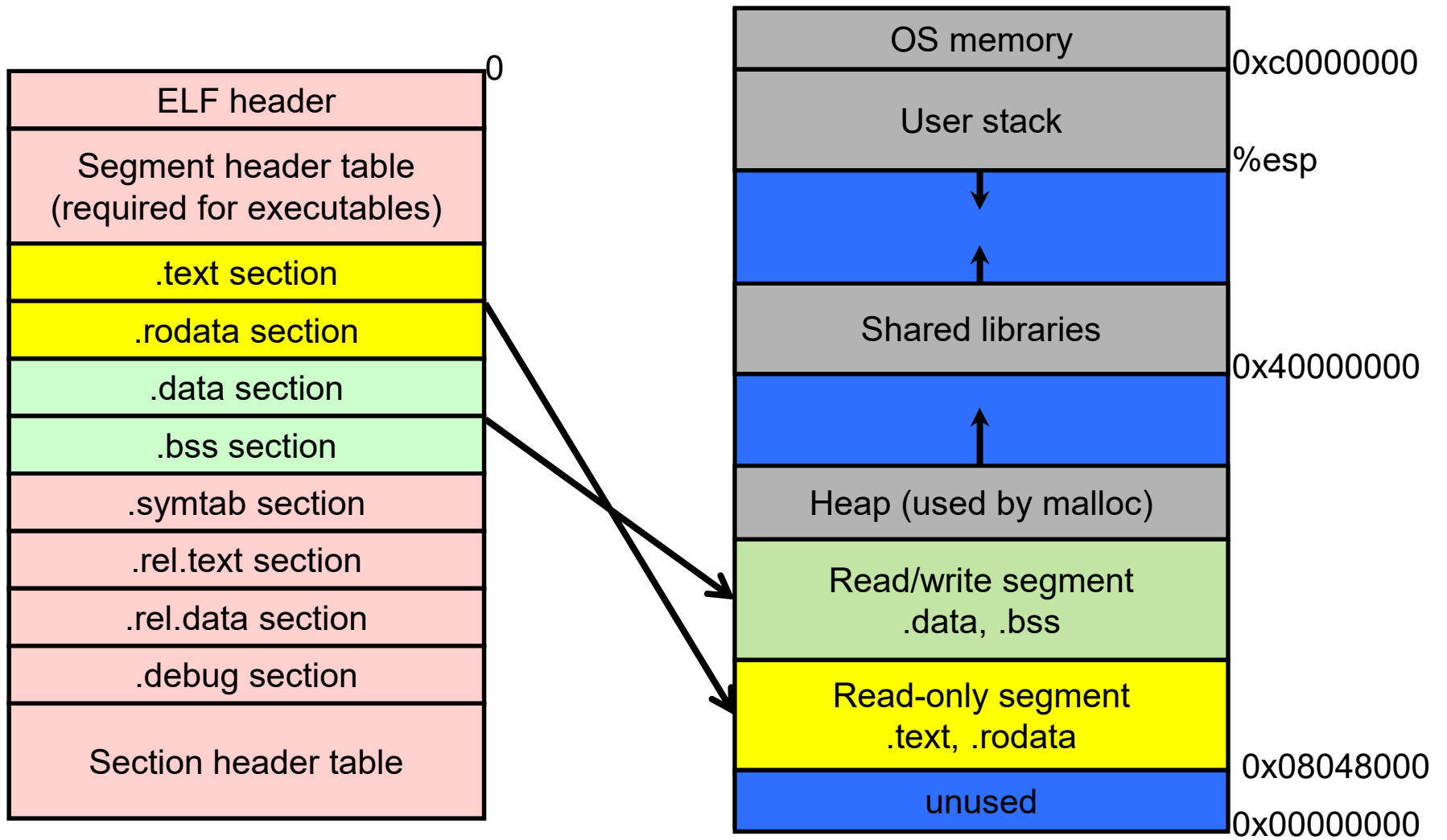
ELF: Executable and Linking Format

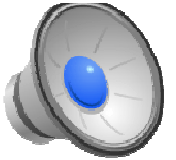
- A single format to represent object files, shared libraries (.so), and executables.
- Header: magic number, type, etc.
- Segment header: maps segments in file to (runtime) segments in memory.
- Text section: code
- Rodata: Read-only data
- Bss: Uninitialized global variables
- Symtab: Symbol Table
- Rel.Text: Relocation for text
- Rel.Data: Relocation for data
- Debug: Debugging info (gcc -g)
- Section header table: offsets and sizes of each section





From ELF to Process





Populating the Address Space

- Until you have virtual memory, you will need to copy all the code and data from the executable into the address space.
- The heap and stack are not present in the executable, so what do you do?
 - When you need pages in the heap or stack, you produce zero-filled pages.