CS 33 – Discussion 1A
Computer System Organization

Week 8

Questions before we start?

Logistics

- Lab 4 – parallel lab – OpenMP multithreading\
    - Due 11:59pm Friday June 5th
- HW 4 and HW5 have been released
    - Hw 4 due: Wednesday, 27 May 2020 11:59pm PDT
    - Hw 5 due: Wednesday, 3 June 2020, 11:59 PM PDT
- Please submit suggestions on CCLE TA-site on how to improve OH/Disc.
    - Please fill out LA survey on CCLE if you haven't already

Agenda

10am-11am PST:

- Exceptions
- Linking
- Virtual Memory

11am-11:50am PST:

- LA worksheet

Exceptions

Exception:

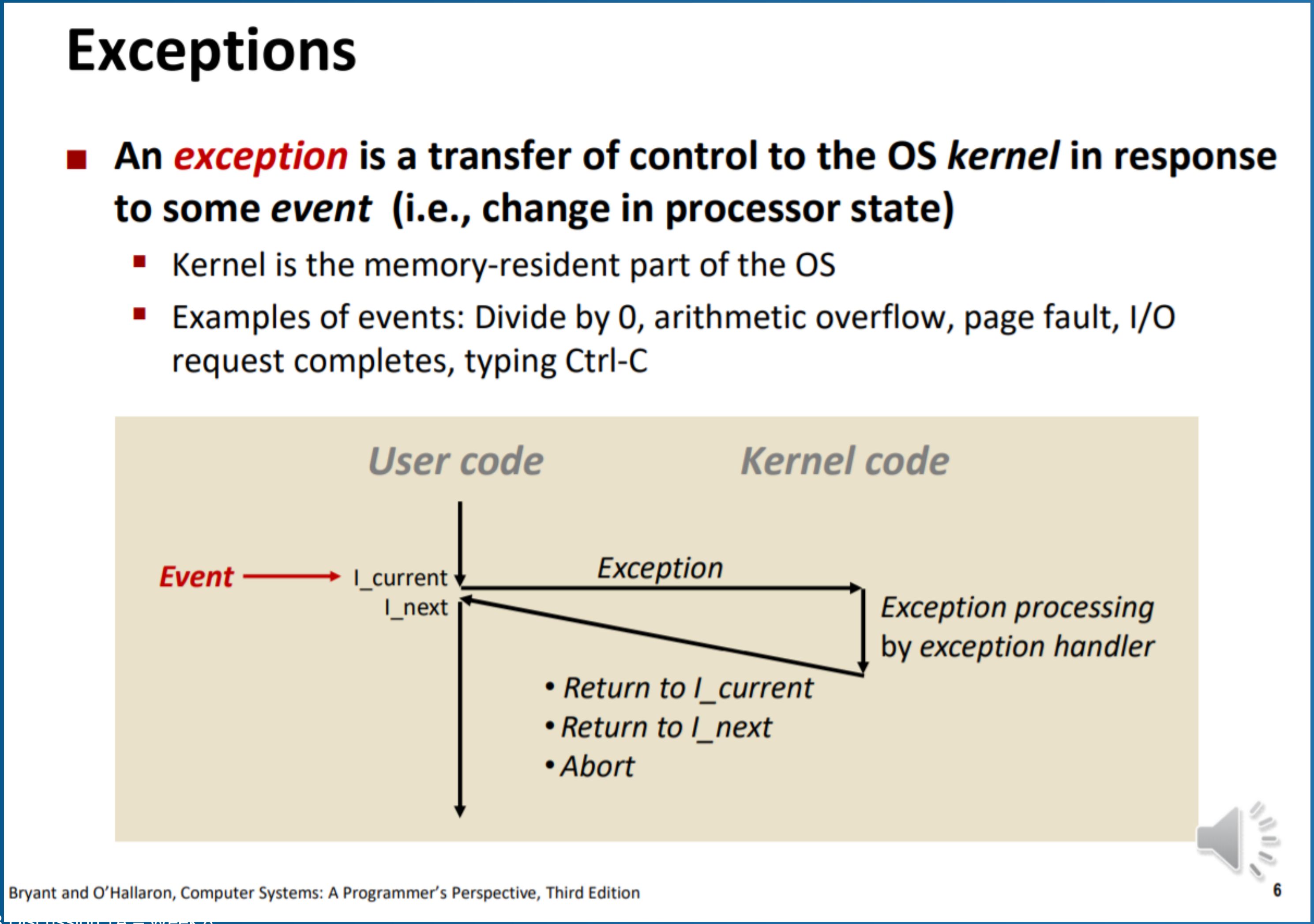
- An event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions. During an exception, control is transferred to the OS.

- An **exception table** stores pointers to exception-handler code, where the exception number is used as an index into this table.

3 things can happen after exception-handling code is executed:

1. The current instruction gets executed again – recoverable, restart
2. The next instruction gets executed – recoverable, onto the next
3. The program aborts – unrecoverable

Exceptions

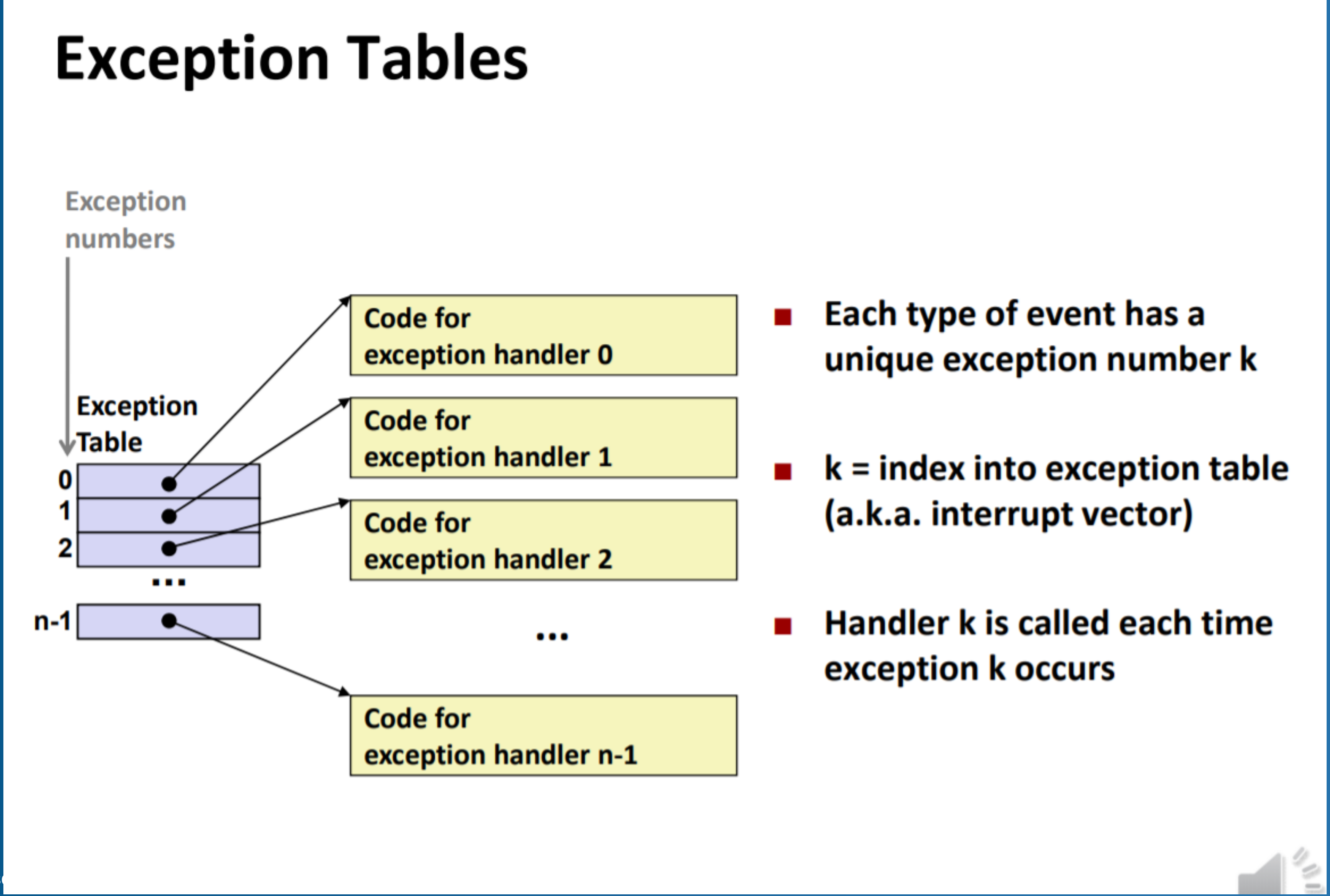


Exceptions

- An *exception* is a transfer of control to the OS *kernel* in response to some *event* (i.e., change in processor state)
  - Kernel is the memory-resident part of the OS
  - Examples of events: Divide by 0, arithmetic overflow, page fault, I/O request completes, typing Ctrl-C

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Exceptions



Exception Tables

Exception numbers

Exception Table

0
1
2
...
n-1

Code for exception handler 0

Code for exception handler 1

Code for exception handler 2

...

Code for exception handler n-1

- Each type of event has a unique exception number k
- k = index into exception table (a.k.a. interrupt vector)
- Handler k is called each time exception k occurs

Exceptions

Types of exceptions :

1) Asynchronous:

- Caused by events or devices external to the processor and memory, e.g. network delivers packet, disk delivers page, keyboard interrupt.
- Control returns to the next instruction

2) Synchronous:

Caused by the execution of an instruction

- **Traps** - Intentional, control returns to next instruction e.g. syscall to fork thread
- **Faults -** Unintentional, possibly recoverable, re-executes current instruction or aborts e.g. pagefault (recoverable), division by zer0
- **Aborts -** Unintentional, unrecoverable, e.g. access bad memory (seg fault)

Exceptions

Types of exceptions :

1) Asynchronous:

- Caused by events or devices external to the processor and memory, e.g. network delivers packet, disk delivers page, keyboard interrupt.
- Control returns to the next instruction

2) Synchronous:

Caused by the execution of an instruction

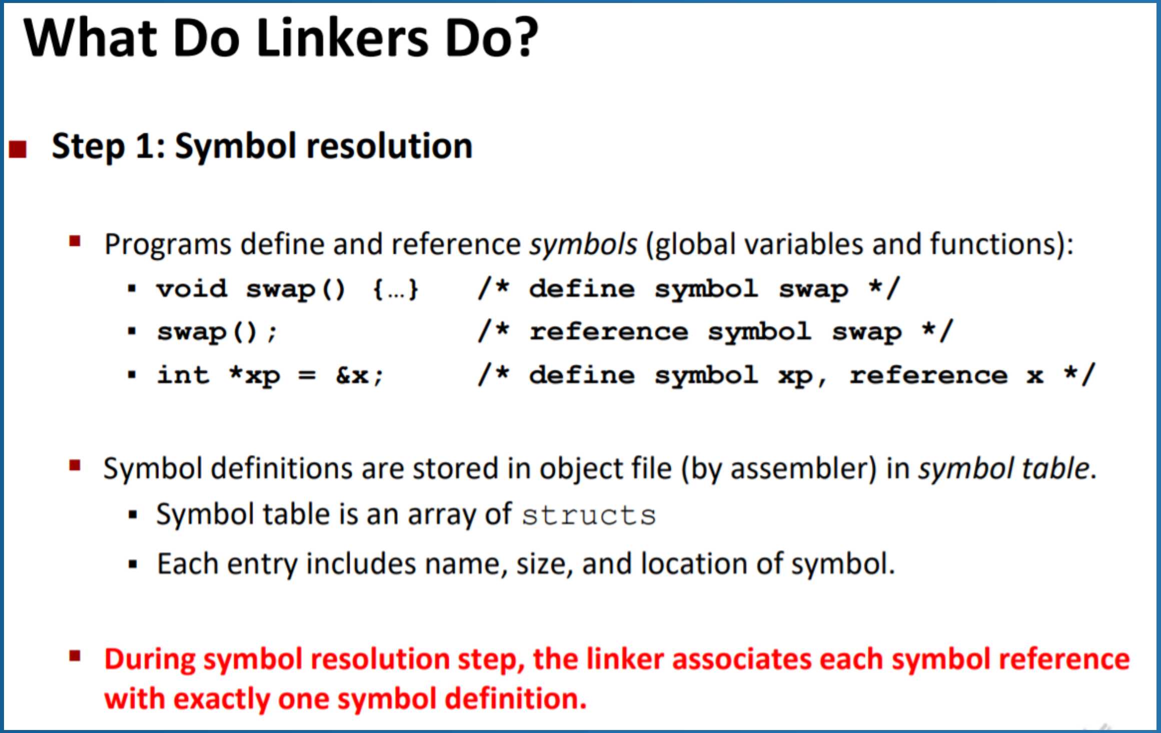
- **Traps** - Intentional, control returns to next instruction e.g. syscall to fork thread
- **Faults -** Unintentional, possibly recoverable, re-executes current instruction or aborts e.g. pagefault (recoverable), division by zer0
- **Aborts -** Unintentional, unrecoverable, e.g. access bad memory (seg fault)

Linking

Steps :

Symbol Resolution:

- Associating variable and function references to variable and function definitions.
- Exactly one definition paired to all declarations – no multiple definitions allowed
- Stores information in symbol table of relocatable object file
- This has implications in terms of the order files try to be compiled in on the command line.



What Do Linkers Do?

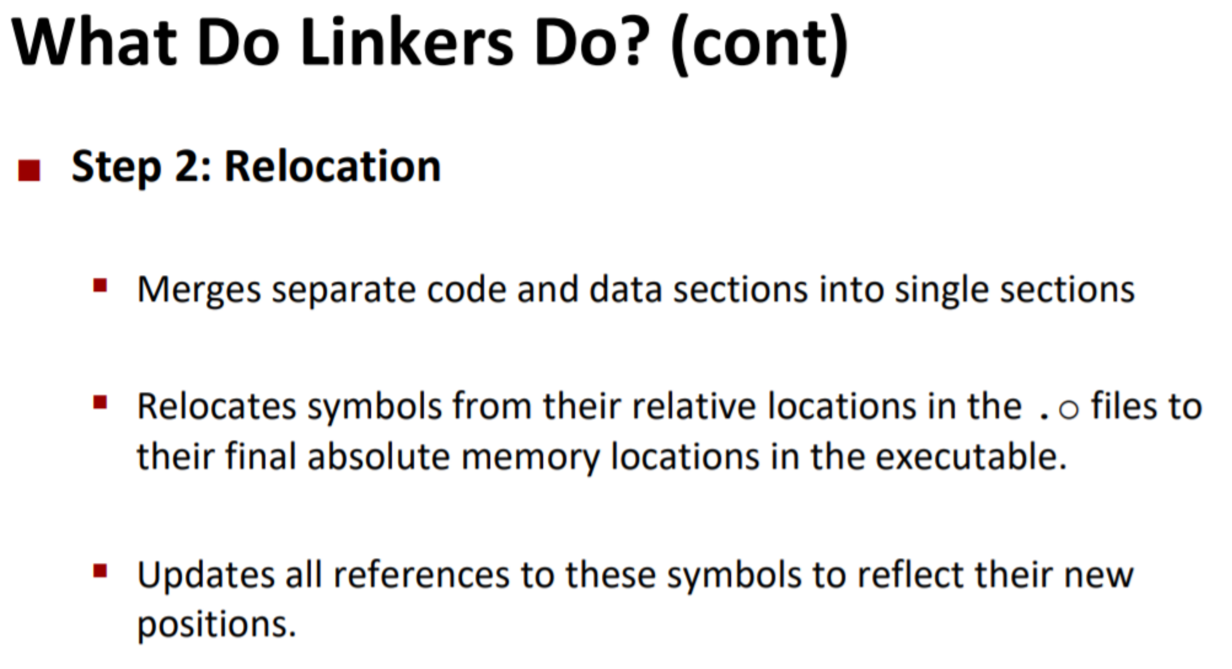
- **Step 1: Symbol resolution**
  - Programs define and reference *symbols* (global variables and functions):
    - `void swap() {…}    /* define symbol swap */`
    - `swap();            /* reference symbol swap */`
    - `int *xp = &x;      /* define symbol xp, reference x */`
  - Symbol definitions are stored in object file (by assembler) in *symbol table*.
    - Symbol table is an array of `structs`
    - Each entry includes name, size, and location of symbol.
  - **During symbol resolution step, the linker associates each symbol reference with exactly one symbol definition.**

Linking

Steps :

ObjectCode Relocation;

- Combining data and code sections of different object files
- Relocating variables and function from locations in their own object files to final memory locations of the executable
- Updating references to these variables and functions with the new memory locations



What Do Linkers Do? (cont)

- **Step 2: Relocation**
  - Merges separate code and data sections into single sections
  - Relocates symbols from their relative locations in the `.o` files to their final absolute memory locations in the executable.
  - Updates all references to these symbols to reflect their new positions.

Linking

Relevant file types :

- **Relocatable object files (.o)** - Combinable with other .o files to produce executable, contains symbtab with definitions in this file and unresolved refs

- **Executable object file (a.out)** - Ready to be copied to memory and executed – next step call loader

- **Shared object file (.so)** - Object file that can be linked dynamically at load-time or run-time

- **Archive files/Static libraries** (.a) - Collection of related .o files. When linked with, .o files - resolve references are pulled into the final executable

Linking

Relevant file types :

- **Relocatable object files (.o)** - Combinable with other .o files to produce executable, contains symbtab with definitions in this file and unresolved refs
- **Executable object file (a.out)** - Ready to be copied to memory and executed – next step call loader
- **Shared object file (.so)** - Object file that can be linked dynamically at load-time or run-time
- **Archive files/Static libraries** (.a) - Collection of related .o files. When linked with, .o files - resolve references are pulled into the final executable

Linking

Relevant file types :

- **Relocatable object files (.o)** - Combinable with other .o files to produce executable, contains symbtab with definitions in this file and unresolved refs
- **Executable object file (a.out)** - Ready to be copied to memory and executed – next step call loader
- **Shared object file (.so)** - Object file that can be linked dynamically at load-time or run-time – used for dynamic linking (DLLs in windows)
- **Archive files/Static libraries (.a)** - Collection of related .o files. When linked with, .o files - resolve references are pulled into the final executable – used with static linking

Linking

Types of symbols :

- **Global symbols -** Symbols that can be referenced by other files, except static symbols in C
- **External symbols -** Global symbols defined by other files, declared in this file
- **Local symbols -** Symbols that are defined within the file and can only be referenced within the file, like static symbols in C or local variables to a function
  - **Local vars are stored where? (hint: function call)**

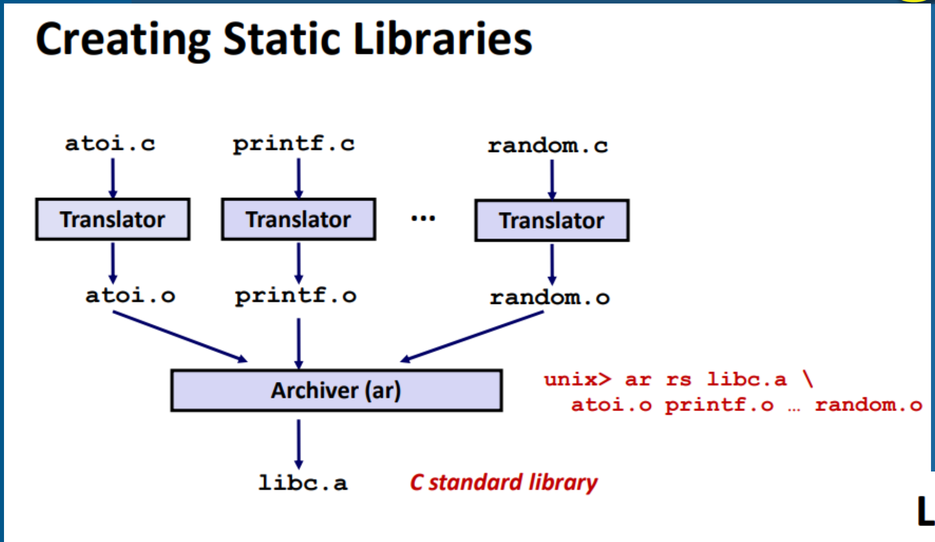
Linking

Types of linking

1) Static linking:

- Copy all modules used by the program into final executable
- Uses .a or .o files
- Executables are not compact/space-efficient
- Entire program needs to be re-compiled and re-linked if a single file changes
- Usually facilitates faster programs
- Programs have constant load-time into memory
- Very portable – only need to ship exe

Linking - Static

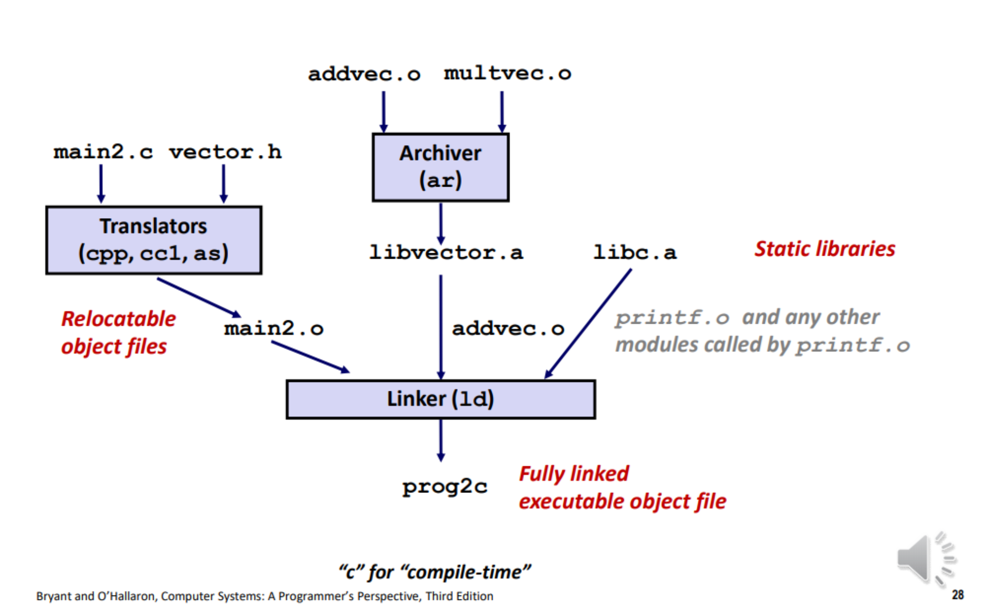


Creating Static Libraries

```
unix> ar rs libc.a \
  atoi.o printf.o … random.o
```

- Archiver allows incremental updates
- Recompile function that changes and replace .o file in archive.

Linking with Static Libraries



"c" for "compile-time"

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

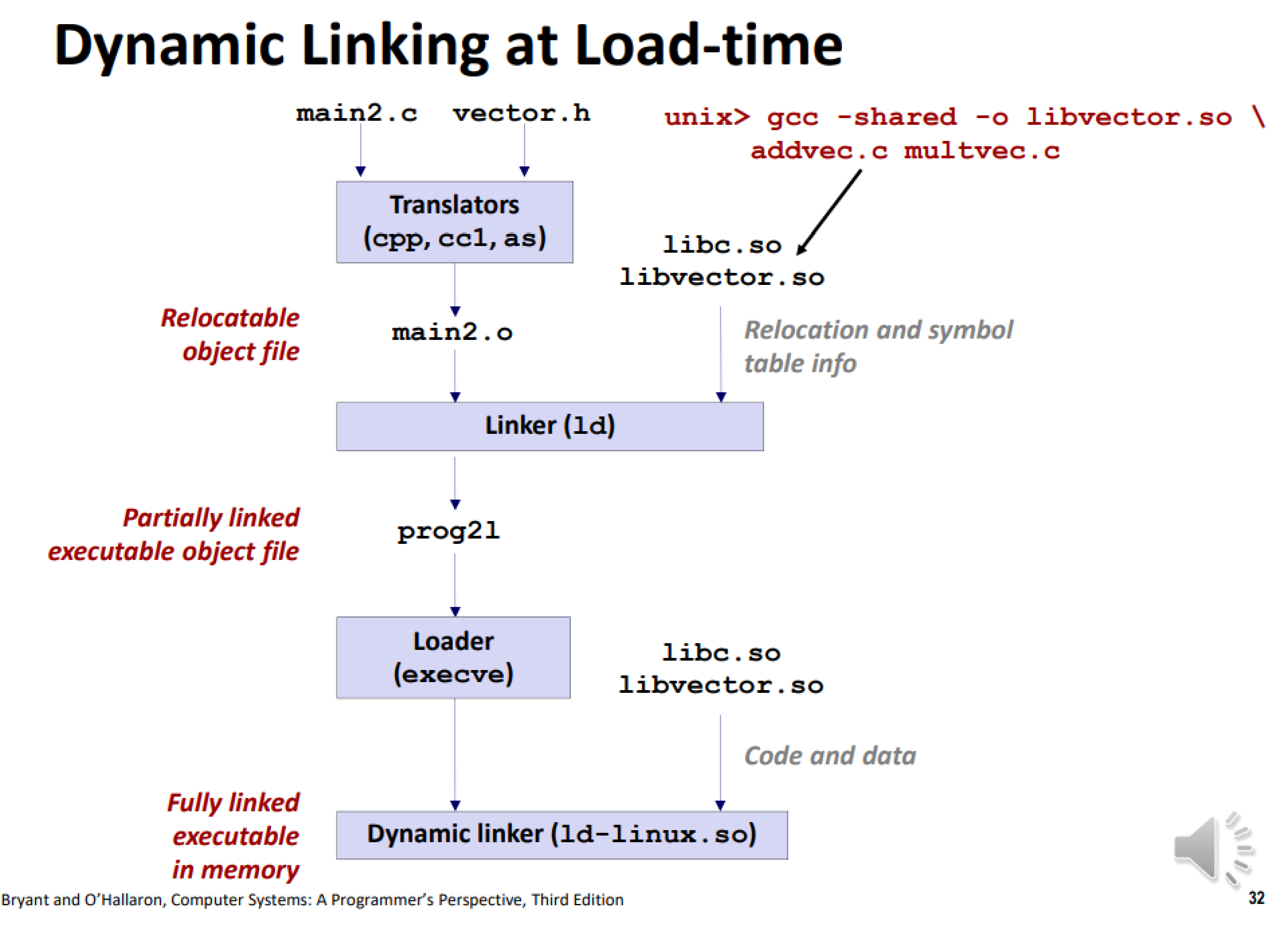
Linking

Types of linking

2) Dynamic linking:

- References are resolved dynamically when the program is run because location of the library is not known until run-time
- Only the addresses of the code, not the actual code itself, makes it into the final executable (using a jump table, and a dynamic loader that fills the jump table)
- Upon program startup, the dynamic load library is invoked to fill in the jump table with the addresses of each function in the shared library
- Can occur at load-time or run-time
- Uses .so files
- Different programs can share the same library in memory
- Only files that have been changed need to be recompiled
- Programs may not run as fast because things are performed dynamically
- Programs can have variable load-time as some shared libraries are already present in memory
- Not as portable if shared libraries don't exist on different machine

Linking – Dynamic



Dynamic Linking at Load-time

```
unix> gcc -shared -o libvector.so \
        addvec.c multvec.c
```

main2.c
vector.h
Translators
(cpp, cc1, as)
libc.so
libvector.so
Relocatable object file
main2.o
Relocation and symbol table info
Linker (ld)
Partially linked executable object file
prog2l
Loader
(execve)
libc.so
libvector.so
Code and data
Fully linked executable in memory
Dynamic linker (ld-linux.so)

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Virtual Memory

See other slides

Questions before we start worksheet?

Works Cited

- Professor Reinman’s slides (CCLE)
- LAs – Sidharth Ramanan, Julia Baylon, Sana Shrikant et al.

- **Credit for compilation: Attiano Purpura-Pontoniere**