Linking, Loading, and Shared Libraries

Martin Kellogg

Course Announcements

- PA4 leaderboard is up
  - current leader is (unsurprisingly) `reference --opt`
  - but things are starting to get competitive

Course Announcements

- PA4 leaderboard is up
  - current leader is (unsurprisingly) `reference --opt`
  - but things are starting to get competitive
- PA4c1 due in one week
  - this checkpoint is mostly optional
  - but you're required to include a dataflow analysis for DCE in your eventual PA4 submission, so we recommend using PA4c1 as an excuse to do it

Course Announcements

- PA4 leaderboard is up
  - current leader is (unsurprisingly) `reference --opt`
  - but things are starting to get competitive
- PA4c1 due in one week
  - this checkpoint is mostly optional
  - but you're required to include a dataflow analysis for DCE in your eventual PA4 submission, so we recommend using PA4c1 as an excuse to do it
- I will be around this afternoon ~4:30pm if you want to see a test case
  - PA3 test case views can still be used

Agenda

- Review + finish global register allocation
- And then...
  - Object Files
  - Linking
  - Relocations
  - Shared Libraries
  - Separate Typechecking

Review: Register Allocation

Review: Register Allocation

- A **register allocator** creates a mapping from IR's abstract registers to physical registers
  - or, if that's not possible, to memory locations

Review: Register Allocation

- A **register allocator** creates a mapping from IR's abstract registers to physical registers
  - or, if that's not possible, to memory locations
- We saw two **local** allocators:

Review: Register Allocation

- A **register allocator** creates a mapping from IR's abstract registers to physical registers
  - or, if that's not possible, to memory locations
- We saw two **local** allocators:
  - one simple one based on **frequency count**

Review: Register Allocation

- A **register allocator** creates a mapping from IR's abstract registers to physical registers
    - or, if that's not possible, to memory locations
- We saw two **local** allocators:
    - one simple one based on **frequency count**
    - a more complex **greedy algorithm** that does much better

Review: Register Allocation

- A **register allocator** creates a mapping from IR's abstract registers to physical registers
  - or, if that's not possible, to memory locations
- We saw two **local** allocators:
  - one simple one based on **frequency count**
  - a more complex **greedy algorithm** that does much better
- We noted that extending a local allocator into a **regional** register allocation is not a good idea

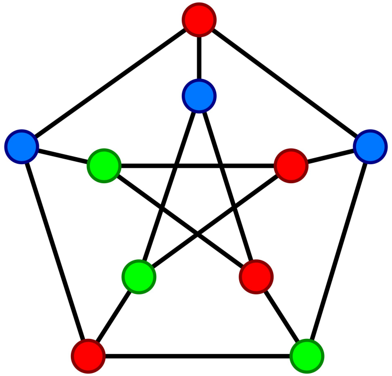
Review: Register Allocation

- A **register allocator** creates a mapping from IR's abstract registers to physical registers
  - or, if that's not possible, to memory locations
- We saw two **local** allocators:
  - one simple one based on **frequency count**
  - a more complex **greedy algorithm** that does much better
- We noted that extending a local allocator into a **regional** register allocation is not a good idea
  - critical edges are one complication on basic block boundaries

Review: Register Allocation

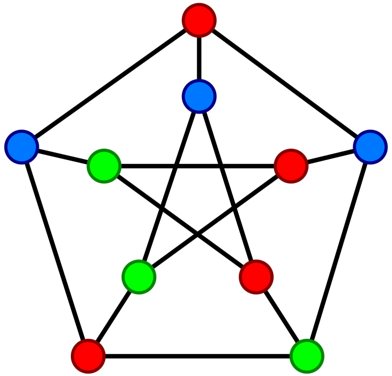
- A **register allocator** creates a mapping from IR's abstract registers to physical registers
  - or, if that's not possible, to memory locations
- We saw two **local** allocators:
  - one simple one based on **frequency count**
  - a more complex **greedy algorithm** that does much better
- We noted that extending a local allocator into a **regional** register allocation is not a good idea
  - critical edges are one complication on basic block boundaries
  - instead, a **global** allocator can coordinate assignments across basic blocks

Review: Global Register Allocation

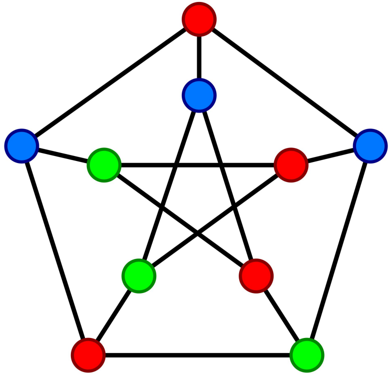


Review: Global Register Allocation

- We're covering one global register allocation algorithm: **reduction to vertex coloring**

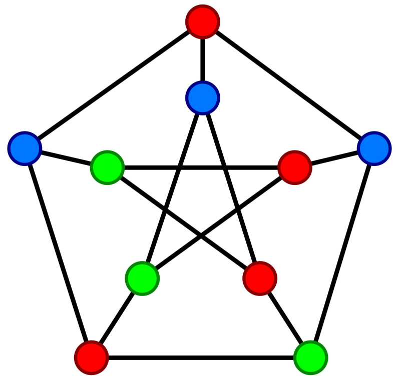


Review: Global Register Allocation



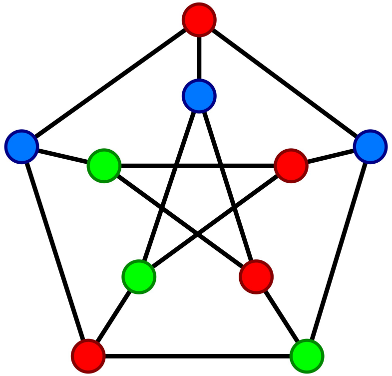
- We're covering one global register allocation algorithm: **reduction to vertex coloring**
- How do we **map** the global register allocation problem to a graph coloring problem?

Review: Global Register Allocation



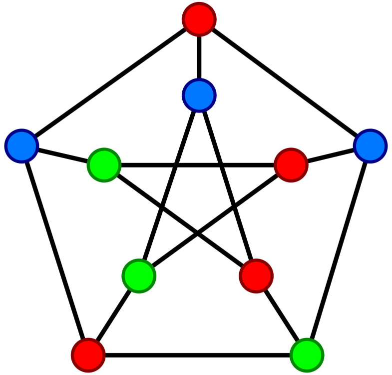
- We're covering one global register allocation algorithm: **reduction to vertex coloring**
- How do we **map** the global register allocation problem to a graph coloring problem?
    - We extended our notion of liveness to ***live ranges***, each of which is a def-use closure

Review: Global Register Allocation



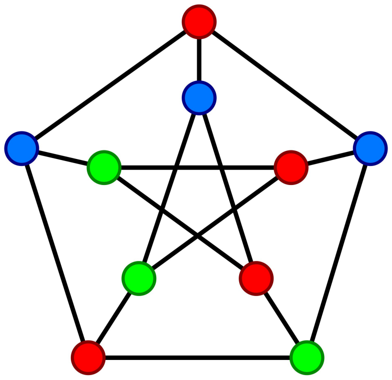
- We're covering one global register allocation algorithm: **reduction to vertex coloring**
- How do we **map** the global register allocation problem to a graph coloring problem?
  - We extended our notion of liveness to ***live ranges***, each of which is a def-use closure
  - We used those live ranges to construct an ***interference graph***

Review: Global Register Allocation



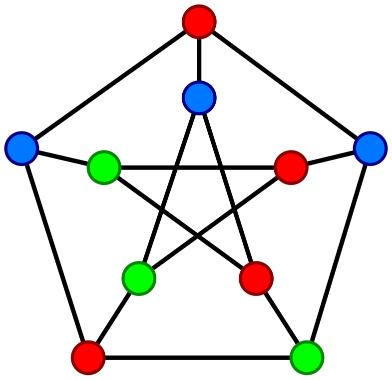
- We're covering one global register allocation algorithm: **reduction to vertex coloring**
- How do we **map** the global register allocation problem to a graph coloring problem?
  - We extended our notion of liveness to ***live ranges***, each of which is a def-use closure
  - We used those live ranges to construct an ***interference graph***
    - nodes in this graph are values, edges represent values that are simultaneously live

Review: Global Register Allocation



- We're covering one global register allocation algorithm: **reduction to vertex coloring**
- How do we **map** the global register allocation problem to a graph coloring problem?
  - We extended our notion of liveness to ***live ranges***, each of which is a def-use closure
  - We used those live ranges to construct an ***interference graph***
    - nodes in this graph are values, edges represent values that are simultaneously live
    - it is easy to construct this graph from live ranges

Review: Global Register Allocation



- We're covering one global register allocation algorithm: **reduction to vertex coloring**
- How do we **map** the global register allocation problem to a graph coloring problem?
  - We extended our notion of liveness to ***live ranges***, each of which is a def-use closure
  - We used those live ranges to construct an ***interference graph***
    - nodes in this graph are values, edges represent values that are simultaneously live
    - it is easy to construct this graph from live ranges
- Next, we will **actually color** this graph

Global Reg. Alloc.: Coloring the Graph

- Graph coloring can be used to allocate registers to values by trying to color the interference graph with **as many colors as there are registers** in the target machine

Global Reg. Alloc.: Coloring the Graph

- Graph coloring can be used to allocate registers to values by trying to color the interference graph with **as many colors as there are registers** in the target machine
  - This is **not always possible**, in which case some values must be spilled (i.e. stored in memory)

Global Reg. Alloc.: Coloring the Graph

- Graph coloring can be used to allocate registers to values by trying to color the interference graph with **as many colors as there are registers** in the target machine
  - This is **not always possible**, in which case some values must be spilled (i.e. stored in memory)
  - We can map "spilling a value" back into the graph to **simplify** the interference graph (and therefore the graph coloring problem)

Global Reg. Alloc.: Coloring the Graph

- Graph coloring can be used to allocate registers to values by trying to color the interference graph with **as many colors as there are registers** in the target machine
  - This is **not always possible**, in which case some values must be spilled (i.e. stored in memory)
  - We can map “spilling a value” back into the graph to **simplify** the interference graph (and therefore the graph coloring problem)
    - To be clear: this approach is **heuristic** (graph coloring is NP-complete). You could solve the graph coloring problem here using any graph coloring algorithm.

Global Reg. Alloc.: Coloring by Simplification

- Assume we are coloring a graph G with K colors

Global Reg. Alloc.: Coloring by Simplification

- Assume we are coloring a graph G with K colors
- Coloring by simplification works as follows:

Global Reg. Alloc.: Coloring by Simplification

- Assume we are coloring a graph G with K colors
- Coloring by simplification works as follows:
  - as long as the graph G has at least one node n with **less than K neighbors**, n is removed from G, and coloring proceeds with that simplified graph

Global Reg. Alloc.: Coloring by Simplification

- Assume we are coloring a graph G with K colors
- Coloring by simplification works as follows:
  - as long as the graph G has at least one node n with **less than K neighbors**, n is removed from G, and coloring proceeds with that simplified graph
    - we assign n a *specific* color later; we remove it here because we know that coloring it is *possible*

Global Reg. Alloc.: Coloring by Simplification

- Assume we are coloring a graph G with K colors
- Coloring by simplification works as follows:
  - as long as the graph G has at least one node n with **less than K neighbors**, n is removed from G, and coloring proceeds with that simplified graph
    - we assign n a *specific* color later; we remove it here because we know that coloring it is *possible*
- More formally, if the simplified graph is ***K-colorable***, then so is G: since n has less than K neighbors, those use at most K-1 colors, and there is therefore at least one color available for n.

Global Reg. Alloc.: Coloring by Simplification

- During simplification, it is possible to reach a point where **all nodes have at least K neighbors**

Global Reg. Alloc.: Coloring by Simplification

- During simplification, it is possible to reach a point where **all nodes have at least K neighbors**
  - When this occurs, a node must be chosen and its value must be **spilled**

Global Reg. Alloc.: Coloring by Simplification

- During simplification, it is possible to reach a point where **all nodes have at least K neighbors**
  - When this occurs, a node must be chosen and its value must be **spilled**
  - Then, we can remove its node from the graph

Global Reg. Alloc.: Coloring by Simplification

- During simplification, it is possible to reach a point where **all nodes have at least K neighbors**
    - When this occurs, a node must be chosen and its value must be **spilled**
    - Then, we can remove its node from the graph
- When colors are assigned to nodes after simplification is complete, it can happen that a node initially designated as spilled can be colored because its neighbors do not use all available colors

Global Reg. Alloc.: Coloring by Simplification

- During simplification, it is possible to reach a point where **all nodes have at least K neighbors**
  - When this occurs, a node must be chosen and its value must be **spilled**
  - Then, we can remove its node from the graph
- When colors are assigned to nodes after simplification is complete, it can happen that a node initially designated as spilled can be colored because its neighbors do not use all available colors
  - When this happens, the potential spill is not turned into an actual spill

Global Reg. Alloc.: Coloring by Simplification

- During simplification, it is possible to reach a point where **all nodes have at least K neighbors**
    - When this occurs, a node must be chosen and its value must be **spilled**
    - Then, we can remove its node from the graph
- When colors are assigned to nodes after simplification is complete, it can happen that a node initially designated as spilled can be colored because its neighbors do not use all available colors
    - When this happens, the potential spill is not turned into an actual spill
    - This technique is known as ***optimistic coloring***

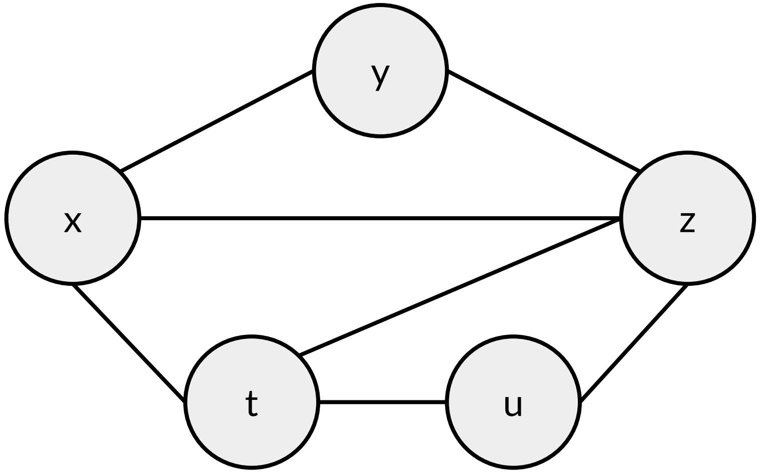
Global Reg. Alloc.: Coloring by Simplification

- During simplification, it is possible to reach a point where **all nodes have at leas**
  - When t ... e **spilled**
  - Then, w
- When colo ... e, it can happen that a node initially designated as spilled can be colored because its neighbors do not use all available colors
  - When this happens, the potential spill is not turned into an actual spill
  - This technique is known as ***optimistic coloring***

When a node is really spilled, the program has to be rewritten, which changes the interference graph. When this happens, the allocation process must be **restarted completely**. In practice, it converges in one or two iterations in most cases.

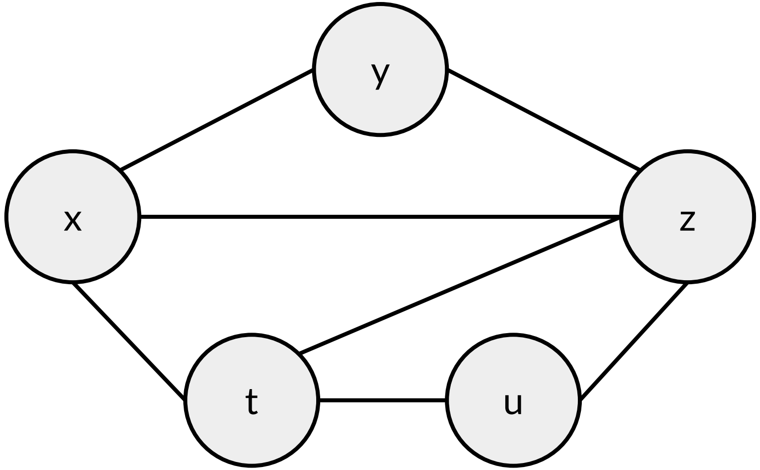
Coloring by Simplification Example

Consider the graph from earlier:



Coloring by Simplification Example

Consider the graph from earlier:

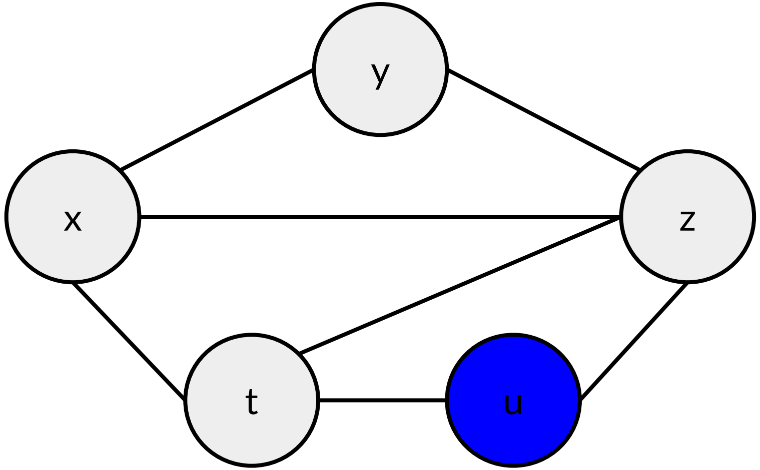


Let's try to fit these values into 3 registers:

- r1
- r2
- r3

Coloring by Simplification Example

Consider the graph from earlier:

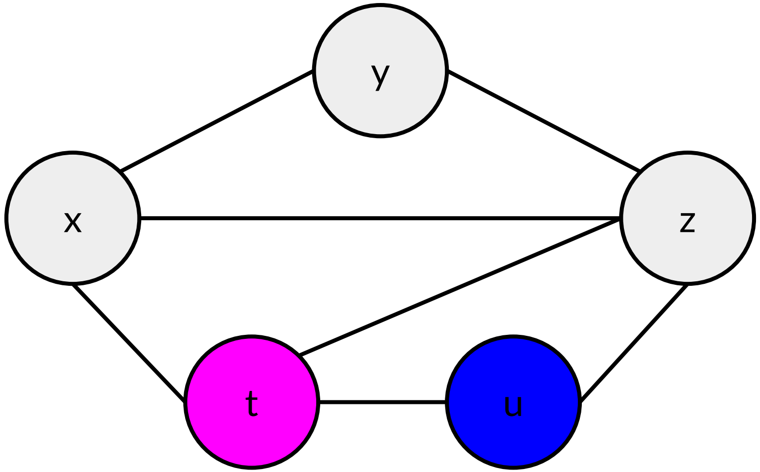


Let's try to fit these values into 3 registers:

- **r1**
- **r2**
- **r3**

Coloring by Simplification Example

Consider the graph from earlier:

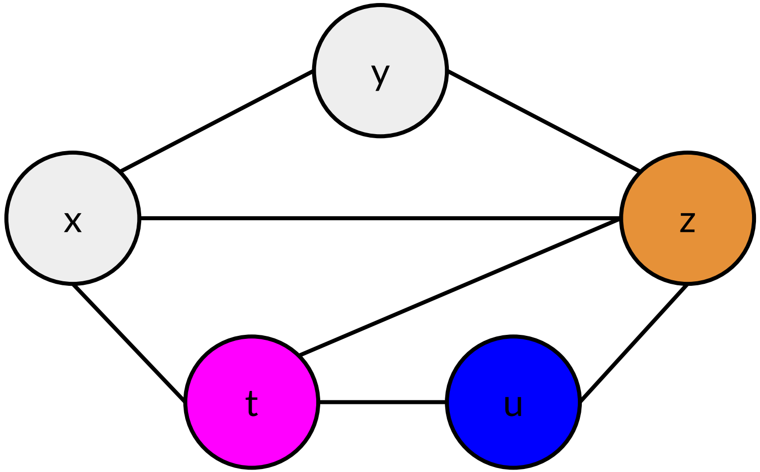


Let's try to fit these values into 3 registers:

- r1
- r2
- r3

Coloring by Simplification Example

Consider the graph from earlier:

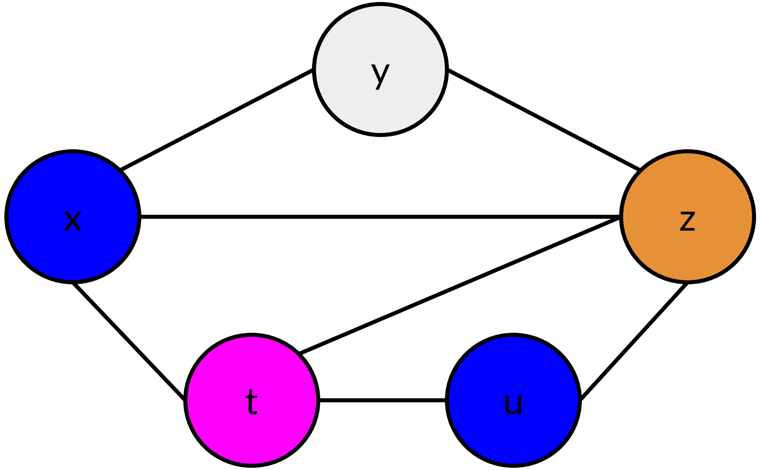


Let's try to fit these values into 3 registers:

- r1
- r2
- r3

Coloring by Simplification Example

Consider the graph from earlier:

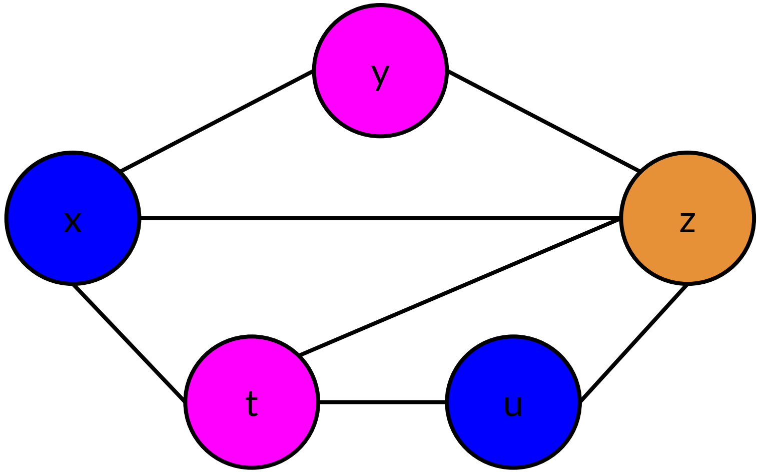


Let's try to fit these values into 3 registers:

- r1
- r2
- r3

Coloring by Simplification Example

Consider the graph from earlier:



Let's try to fit these values into 3 registers:

- **r1**
- **r2**
- **r3**

Global Reg. Alloc.: Coloring Complications

Global Reg. Alloc.: Coloring Complications

- It is sometimes useful to ***coalesce*** two SSA names that do not share an edge into a single name (and then put them in the same register)

Global Reg. Alloc.: Coloring Complications

- It is sometimes useful to ***coalesce*** two SSA names that do not share an edge into a single name (and then put them in the same register)
  - unfortunately, proving that this is safe is tough

Global Reg. Alloc.: Coloring Complications

- It is sometimes useful to ***coalesce*** two SSA names that do not share an edge into a single name (and then put them in the same register)
  - unfortunately, proving that this is safe is tough
- Similarly, there are some situations where ***splitting a live range*** can take a graph that is not K-colorable and make it K-colorable

Global Reg. Alloc.: Coloring Complications

- It is sometimes useful to ***coalesce*** two SSA names that do not share an edge into a single name (and then put them in the same register)
  - unfortunately, proving that this is safe is tough
- Similarly, there are some situations where ***splitting a live range*** can take a graph that is not K-colorable and make it K-colorable
  - “splitting a live range” = storing the value to memory and the retrieving it later, effectively creating two new live ranges

Global Reg. Alloc.: Coloring Complications

- It is sometimes useful to ***coalesce*** two SSA names that do not share an edge into a single name (and then put them in the same register)
    - unfortunately, proving that this is safe is tough
- Similarly, there are some situations where ***splitting a live range*** can take a graph that is not K-colorable and make it K-colorable
    - “splitting a live range” = storing the value to memory and the retrieving it later, effectively creating two new live ranges
    - again, no good, general-case heuristics :(

Global Reg. Alloc.: Coloring Complications

- It is sometimes useful to ***coalesce*** two SSA names that do not share an edge into a single name (and then put them in the same register)
  - unfortunately, proving that this is safe is tough
- Similarly, there are some situations where ***splitting a live range*** can take a graph that is not K-colorable and make it K-colorable
  - “splitting a live range” = storing the value to memory and the retrieving it later, effectively creating two new live ranges
  - again, no good, general-case heuristics :(
- It’s common that you may need to put some values in specific registers, e.g. to adhere to calling conventions

Global Reg. Alloc.: Coloring Complications

- It is sometimes useful to *coalesce* two SSA names that do not share an edge into a single name (and then put them in the same register)
    - unfortunately, proving that this is safe is tough
- Similarly, there are some situations where ***splitting a live range*** can take a graph that is not K-colorable and make it K-colorable
    - “splitting a live range” = storing the value to memory and the retrieving it later, effectively creating two new live ranges
    - again, no good, general-case heuristics :(
- It’s common that you may need to put some values in specific registers, e.g. to adhere to calling conventions
    - this is easy to handle: just **pre-color** those nodes in the graph

Global Register Allocation: Summary

Global Register Allocation: Summary

- A global register allocator can do **much better** than a local allocator

Global Register Allocation: Summary

- A global register allocator can do **much better** than a local allocator
  - This justifies significant investment into solving the hard problem of global register allocation

Global Register Allocation: Summary

- A global register allocator can do **much better** than a local allocator
  - This justifies significant investment into solving the hard problem of global register allocation
- The traditional technique is **reduction to graph coloring**:

Global Register Allocation: Summary

- A global register allocator can do **much better** than a local allocator
  - This justifies significant investment into solving the hard problem of global register allocation
- The traditional technique is **reduction to graph coloring**:
  - compute **live ranges**

Global Register Allocation: Summary

- A global register allocator can do **much better** than a local allocator
  - This justifies significant investment into solving the hard problem of global register allocation
- The traditional technique is **reduction to graph coloring**:
  - compute **live ranges**
  - construct an **interference graph**

Global Register Allocation: Summary

- A global register allocator can do **much better** than a local allocator
  - This justifies significant investment into solving the hard problem of global register allocation
- The traditional technique is **reduction to graph coloring**:
  - compute **live ranges**
  - construct an **interference graph**
  - use **simplification** to color the graph, decide what to spill, and then assign physical registers

Global Register Allocation: Summary

- A global register allocator can do **much better** than a local allocator
  - This justifies significant investment into solving the hard problem of global register allocation
- The traditional technique is **reduction to graph coloring**:
  - compute **live ranges**
  - construct an **interference graph**
  - use **simplification** to color the graph, decide what to spill, and then assign physical registers
- Implementing a global register allocator correctly is a **challenge**

Global Register Allocation: Summary

- A global register allocator can do **much better** than a local allocator
  - This justifies significant investment into solving the hard problem of global register allocation
- The traditional technique is **reduction to graph coloring**:
  - compute **live ranges**
  - construct an **interference graph**
  - use **simplification** to color the graph, decide what to spill, and then assign physical registers
- Implementing a global register allocator correctly is a **challenge**
  - I don't expect all (or even most) of you to succeed at this, and it is not required for PA4

Agenda

- Review + finish global register allocation
- **And then…**
  - Object Files
  - Linking
  - Relocations
  - Shared Libraries
  - Separate Typechecking

Motivation: Separate Compilation

Motivation: Separate Compilation

- ***Separate compilation*** is the ability to compile different parts of your program at different times
    - And then link them together later

Motivation: Separate Compilation

- ***Separate compilation*** is the ability to compile different parts of your program at different times
    - And then link them together later
- This is a **big win**. Why?

Motivation: Separate Compilation

- ***Separate compilation*** is the ability to compile different parts of your program at different times
  - And then link them together later
- This is a **big win**. Why?
  - Faster compile times on small changes
  - For software engineering reasons (modularity, team organization, etc - take 490 with me in the fall!)
  - Independently develop different parts (libraries)

Motivation: Separate Compilation

- ***Separate compilation*** is the ability to compile different parts of your program at different times
  - And then link them together later
- This is a **big win**. Why?
  - Faster compile times on small changes
  - For software engineering reasons (modularity, team organization, etc - take 490 with me in the fall!)
  - Independently develop different parts (libraries)
- **All** major languages and big projects use it

Motivation: Separate Compilation

- *Separate compilation* is the ability to compile different parts of your program at different times
  - And then link them together later
- This is a **big win**. Why?
  - Faster compile times on small changes
  - For software engineering reasons (modularity, team organization, etc - take 490 with me in the fall!)
  - Independently develop different parts (libraries)
- All major languages and big projects use it
  - Compilers/languages that do not support separate compilation are **not useful** in practice (RIP Cool)

Pieces of Programs

- A compiled program fragment is called an ***object file***. It contains:

Pieces of Programs

- A compiled program fragment is called an ***object file***. It contains:
    - Code (for methods, etc.)

Pieces of Programs

- A compiled program fragment is called an ***object file***. It contains:
    - Code (for methods, etc.)
    - Variables (e.g., values for global variables)

Pieces of Programs

- A compiled program fragment is called an ***object file***. It contains:
    - Code (for methods, etc.)
    - Variables (e.g., values for global variables)
    - Debugging information

Pieces of Programs

- A compiled program fragment is called an ***object file***. It contains:
  - Code (for methods, etc.)
  - Variables (e.g., values for global variables)
  - Debugging information
  - References to code/data that appears elsewhere (e.g., printf)

Pieces of Programs

- A compiled program fragment is called an ***object file***. It contains:
  - Code (for methods, etc.)
  - Variables (e.g., values for global variables)
  - Debugging information
  - References to code/data that appears elsewhere (e.g., printf)
  - Tables for organizing the above

Pieces of Programs

- A compiled program fragment is called an ***object file***. It contains:
  - Code (for methods, etc.)
  - Variables (e.g., values for global variables)
  - Debugging information
  - References to code/data that appears elsewhere (e.g., printf)
  - Tables for organizing the above
- The job of the ***linker*** is to combine one or more object files into a single executable

Pieces of Programs

- A compiled program fragment is called an ***object file***. It contains:
  - Code (for methods, etc.)
  - Variables (e.g., values for global variables)
  - Debugging information
  - References to code/data that appears elsewhere (e.g., printf)
  - Tables for organizing the above
- The job of the ***linker*** is to combine one or more object files into a single executable
  - compiler : source code -> assembly
  - assembler : assembly -> object file
  - linker : object files -> binary

Linking: Two Big Tasks

Linking: Two Big Tasks

- Recall that the operating system uses **virtual memory** so every program starts at a standard [virtual] address (e.g., address 0)

Linking: Two Big Tasks

- Recall that the operating system uses **virtual memory** so every program starts at a standard [virtual] address (e.g., address 0)
  - As a result, each program has its own **virtual address space**

Linking: Two Big Tasks

- Recall that the operating system uses **virtual memory** so every program starts at a standard [virtual] address (e.g., address 0)
  - As a result, each program has its own **virtual address space**
- Linking involves two tasks:

Linking: Two Big Tasks

- Recall that the operating system uses **virtual memory** so every program starts at a standard [virtual] address (e.g., address 0)
  - As a result, each program has its own **virtual address space**
- Linking involves two tasks:
  - **Relocating** the code and data from each object file to a particular fixed virtual address

Linking: Two Big Tasks

- Recall that the operating system uses **virtual memory** so every program starts at a standard [virtual] address (e.g., address 0)
  - As a result, each program has its own **virtual address space**
- Linking involves two tasks:
  - **Relocating** the code and data from each object file to a particular fixed virtual address
  - **Resolving references** (e.g., to variable locations or jump-target labels) so that they point to concrete and correct virtual addresses in the ~~New World Order~~ program's virtual address space

Relocatable Object Files

For this to work, a ***relocatable*** object file must have three tables:

Relocatable Object Files

For this to work, a ***relocatable*** object file must have three tables:

- ***Import Table***: points to places in the code where an external symbol (variable or method) is references

Relocatable Object Files

For this to work, a ***relocatable*** object file must have three tables:

- ***Import Table***: points to places in the code where an external symbol (variable or method) is references
    - List of (external_symbol_name, where_in_code) pairs

Relocatable Object Files

For this to work, a ***relocatable*** object file must have three tables:

- ***Import Table***: points to places in the code where an external symbol (variable or method) is references
  - List of (external_symbol_name, where_in_code) pairs
  - One external_symbol_name may come up **many times**!

Relocatable Object Files

For this to work, a ***relocatable*** object file must have three tables:

- ***Import Table***: points to places in the code where an external symbol (variable or method) is references
    - List of (external_symbol_name, where_in_code) pairs
    - One external_symbol_name may come up **many times**!
- ***Export Table***: points to symbol definitions in the code that are exported for use by others

Relocatable Object Files

For this to work, a ***relocatable*** object file must have three tables:

- ***Import Table***: points to places in the code where an external symbol (variable or method) is references
  - List of (external_symbol_name, where_in_code) pairs
  - One external_symbol_name may come up **many times**!
- ***Export Table***: points to symbol definitions in the code that are exported for use by others
  - List of (internal_symbol_name, where_in_code) pairs

Relocatable Object Files

For this to work, a ***relocatable*** object file must have three tables:

- ***Import Table***: points to places in the code where an external symbol (variable or method) is references
    - List of (external_symbol_name, where_in_code) pairs
    - One external_symbol_name may come up **many times**!
- ***Export Table***: points to symbol definitions in the code that are exported for use by others
    - List of (internal_symbol_name, where_in_code) pairs
- ***Relocation Table***: points to places in the code where local symbols are referenced

Relocatable Object Files

For this to work, a ***relocatable*** object file must have three tables:

- ***Import Table***: points to places in the code where an external symbol (variable or method) is references
  - List of (external_symbol_name, where_in_code) pairs
  - One external_symbol_name may come up **many times**!
- ***Export Table***: points to symbol definitions in the code that are exported for use by others
  - List of (internal_symbol_name, where_in_code) pairs
- ***Relocation Table***: points to places in the code where local symbols are referenced
  - List of (internal_symbol_name, where_in_code) pairs

Relocatable Object Files

For this to work, a ***relocatable*** object file must have three tables:

- ***Import Table***: points to places in the code where an external symbol (variable or method) is references
    - List of (external_symbol_name, where_in_code) pairs
    - One external_symbol_name may come up **many times**!
- ***Export Table***: points to symbol definitions in the code that are exported for use by others
    - List of (internal_symbol_name, where_in_code) pairs
- ***Relocation Table***: points to places in the code where local symbols are referenced
    - List of (internal_symbol_name, where_in_code) pairs
    - One internal_symbol may come up **many times**!

Relocatable Object Files

For this to work, a ***relocatable*** object file must have three tables:

- ***Import Table***: points to places in the code where an external symbol (v
  - List o
  - One e
- ***Export Tal***
  exported
  - List of (internal_symbol_name, where_in_code) pairs
- ***Relocation Table***: points to places in the code where local symbols are referenced
  - List of (internal_symbol_name, where_in_code) pairs
  - One internal_symbol may come up **many times**!

So Many Tables!

- these tables contain **a lot** of information…
- tables also must be easy to understand…
- maybe an example will help

Relocatable Object Files: Example

- Consider this C program:

```
extern double sqrt(double x);

static double temp = 0.0;

double quadratic(double a, b, c) {
        temp = b*b – 4.0*a*c;
        if (temp >= 0.0) { goto has_roots; }
        throw Invalid_Argument;
has_roots:
        return (–b + sqrt(temp)) / (2.0*a);
}
```

Relocatable Object Files: Example

- Consider this C program:

```
extern double sqrt(double x);

static double temp = 0.0;

double quadratic(double a, b, c) {
	temp = b*b – 4.0*a*c;
	if (temp >=
	throw Inv
has_roots:
	return (–b + sqrt(temp)) / (2.0*a);
}
```

0x1000	…
0x1004	push r1
0x1008	call loc_{sqrt}

Relocatable Object Files: Example

- Consider this C program:

```
extern double sqrt(double x);

static doubl

double qua
    temp =
    if (temp >=
    throw Inv
has_roots:
    return (-b + sqrt(temp)) / (2.0*a);
}
```

Import Table:
Replace address used at 0x1008 with the final location of **sqrt**

0x1000	...
0x1004	push r1
0x1008	call loc_{sqrt}

Relocatable Object Files: Example

- Consider this C program:

```
extern double sqrt(double x);

static double temp = 0.0;

double quadratic(double a, b, c) {
        temp = b*b – 4.0*a*c;
        if (temp >= 0.0) { goto has_roots; }
        throw Invalid_Argument;
has_roots:
        return (–b + sqrt(temp)) / (2.0*a);
}
```

0x0200	r1 = b
0x0204	r1 = r1 * r1
0x0208	r2 = 4.0
0x020c	r2 = r2 * a

Relocatable Object Files: Example

- Consider this C program:

```
extern double sqrt(double x);

static double temp = 0.0;

double quadratic(double a, b, c) {
      temp = b*b – 4.0*a*c;
      if (temp >= 0.0) { goto
      throw Invalid_Argum
has_roots:
      return (–b + sqrt(temp
}
```

```
0x0200    r1 = b
0x0204    r1 = r1 * r1
0x0208    r2 = 4.0
0x020c    r2 = r2 * a
```

Export Table:
We provide **quadratic**. If others want it, they can figure out where 0x0200 is finally relocated to. Call that new location R. They then replace all of their references to $loc_{quadratic}$ with R.

Relocatable Object Files: Example

- Consider this C program:

```
extern double sqrt(double x);

static double temp = 0.0;

double quadratic(double a, b, c) {
        temp = b*b – 4.0*a*c;
        if (temp >= 0.0) { goto has_roots; }
        throw Invalid_Argument;
has_roots:
        return (–b + sqrt(temp)) / (2.0*a);
}
```

0x0600 r1 = ld loc_{temp}
0x0604 jgz r1 loc_{has_roots}

Relocatable Object Files: Example

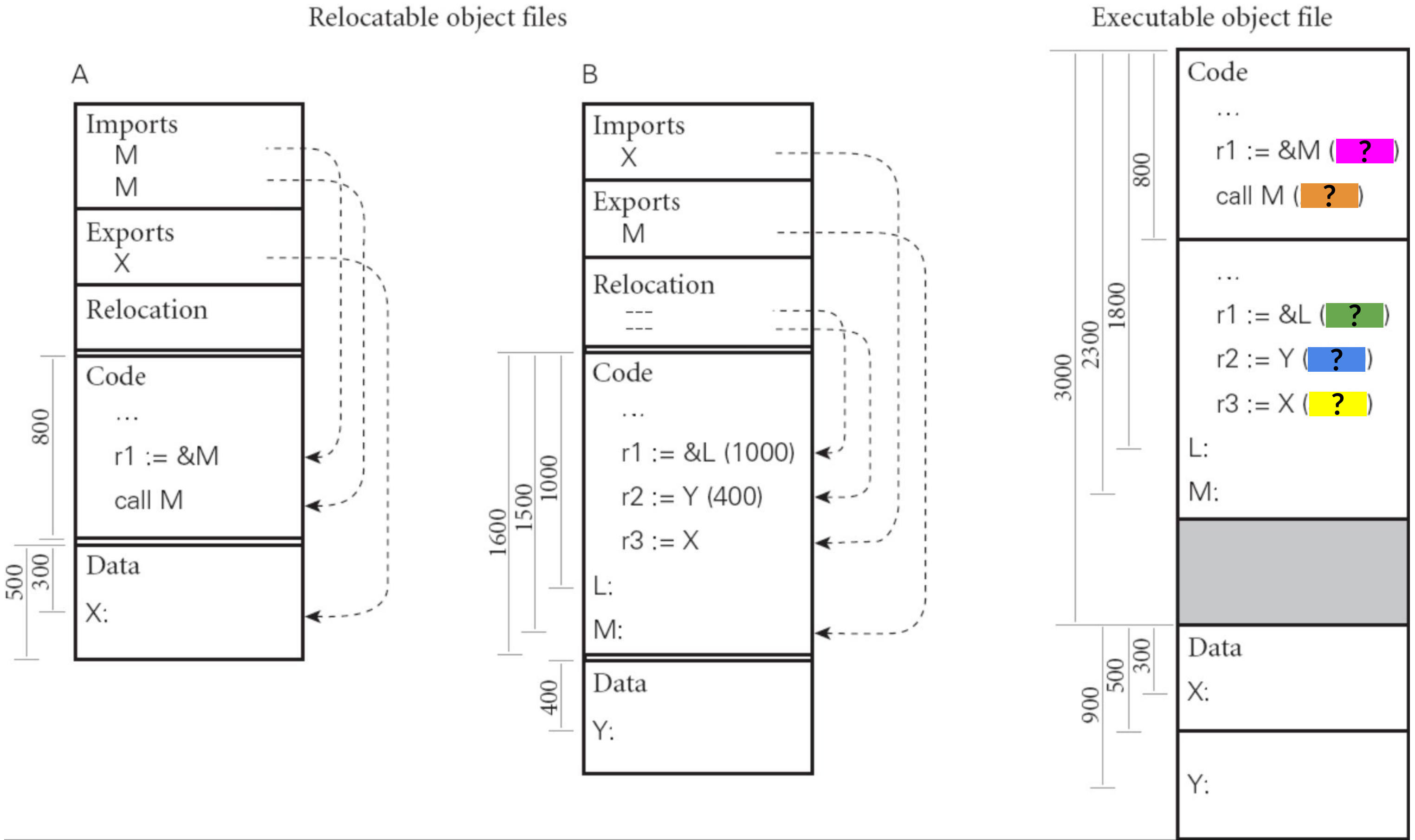
- Consider this C program:

```
extern double sqrt(double x);

static double temp = 0.0;

double quadratic(double a, b, c) {
        temp = b*b – 4.0*a*c;
        if (temp >= 0.0) { goto has_roots; }
        throw Invalid_Argument;
has_roots:
        return (–b + sqrt(temp)) / (2.0*a);
}
```

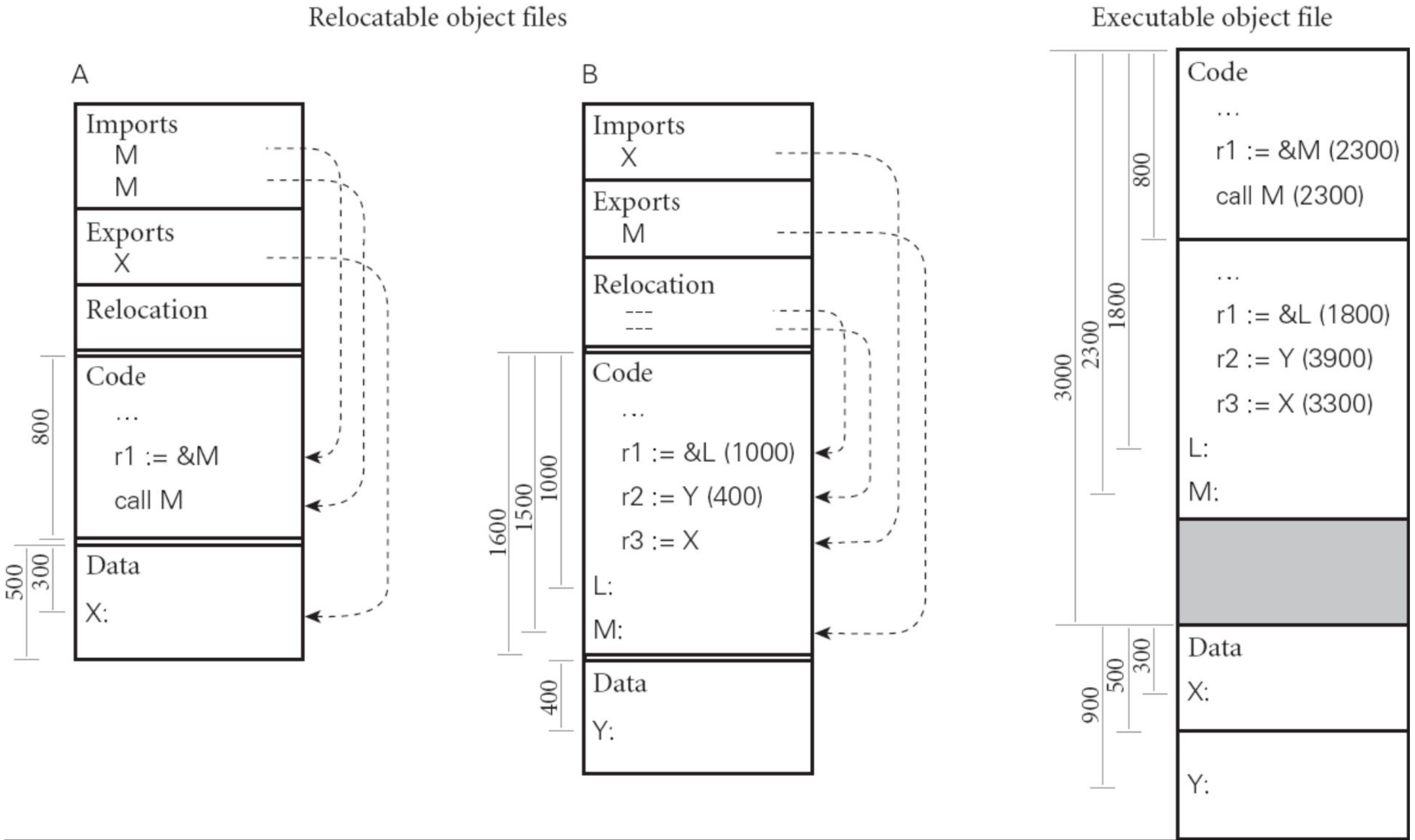
0x0600	r1 =	ld loc_{temp}
0x0604	jgz	r1 loc_{has_roots}

Relocation Table:
Find final relocated address of temp. Call that R_{temp}. Find final relocated address of 0x0600. Call that R_{0x0600}. Replace address referenced at R_{0x0600} with R_{temp}.

Big Linking Example (On Paper w/ a Friend)



Big Linking Example (Answers)



Relocatable Object Files: Summary

Relocatable Object Files: Summary

- Your relocatable object file: main.o
  - Exports main(), imports sqrt(), relocations ...

Relocatable Object Files: Summary

- Your relocatable object file: main.o
  - Exports main(), imports sqrt(), relocations ...
- Your math library: math.o
  - Exports sqrt(), relocations

Relocatable Object Files: Summary

- Your relocatable object file: main.o
  - Exports main(), imports sqrt(), relocations ...
- Your math library: math.o
  - Exports sqrt(), relocations
  - Libraries themselves **can have imports**: example?

Relocatable Object Files: Summary

- Your relocatable object file: main.o
  - Exports main(), imports sqrt(), relocations ...
- Your math library: math.o
  - Exports sqrt(), relocations
  - Libraries themselves **can have imports**: example?
  - In Unix, math.o lives in libmath.a and –lmath on the command line will find it

Relocatable Object Files: Summary

- Your relocatable object file: main.o
  - Exports main(), imports sqrt(), relocations ...
- Your math library: math.o
  - Exports sqrt(), relocations
  - Libraries themselves **can have imports**: example?
  - In Unix, math.o lives in libmath.a and –lmath on the command line will find it
- The linker reads them in, picks a fixed final relocation address for all code and data **(1st pass)** and then goes through and modifies every instruction with a symbol reference **(2nd pass)**

Trivia Break: Philosophy

This Athenian philosopher authored no texts and is known mainly through the posthumous accounts of classical writers, particularly his students Plato and Xenophon. These accounts are written as dialogues, in which this philosopher and his interlocutors examine a subject in the style of question and answer. He was a polarizing figure in the Athenian society of the day; in 399 BC, he was accused of impiety and corrupting the youth. After a trial that lasted a day, he was sentenced to death. He spent his last day in prison, refusing offers to help him escape. Despite this, he has exerted a strong influence on philosophers in later antiquity and has continued to do so in the modern era.

Trivia Break: Philosophy

This family of normative ethical theories prescribes actions that maximize happiness and well-being for the affected individuals. In other words, these theories encourage actions that lead to the greatest good for the greatest number. The seeds of the theory can be found in the hedonists Aristippus and Epicurus (who viewed happiness as the only good), the state consequentialism of the ancient Chinese philosopher Mozi (who developed a theory to maximize benefit and minimize harm), and in the work of the medieval Indian philosopher Shantideva. Modern proponents include Jeremy Bentham, John Stuart Mill, Henry Sidgwick, R. M. Hare, and Peter Singer.

Sharing is Caring

- Relocatable object files are fine, but if two programs **both** use math.o they will **each** get a copy of it

Sharing is Caring

- Relocatable object files are fine, but if two programs **both** use math.o they will **each** get a copy of it
    - You can optimize this a bit by only linking and copying in the parts of a library that you really need (**transitive closure of dependencies**)

Sharing is Caring

- Relocatable object files are fine, but if two programs **both** use math.o they will **each** get a copy of it
  - You can optimize this a bit by only linking and copying in the parts of a library that you really need (**transitive closure of dependencies**)
    - But that's just a band-aid over the real problem

Sharing is Caring

- Relocatable object files are fine, but if two programs **both** use math.o they will **each** get a copy of it
  - You can optimize this a bit by only linking and copying in the parts of a library that you really need (**transitive closure of dependencies**)
    - But that's just a band-aid over the real problem
- If we run both programs, we will load both copies of math.o into memory – **wasting memory** (recall: they're identical)!

Sharing is Caring

- Relocatable object files are fine, but if two programs **both** use math.o they will **each** get a copy of it
    - You can optimize this a bit by only linking and copying in the parts of a library that you really need (**transitive closure of dependencies**)
        - But that's just a band-aid over the real problem
- If we run both programs, we will load both copies of math.o into memory – **wasting memory** (recall: they're identical)!
- How could we go about **sharing** math.o?

Dynamic Linking

- Idea: ***shared libraries*** (.so) or ***dynamically linked libraries*** (.dll, .dylib) use virtual memory so that multiple programs can share the same libraries in main memory

Dynamic Linking

- Idea: ***shared libraries*** (.so) or ***dynamically linked libraries*** (.dll, .dylib) use virtual memory so that multiple programs can share the same libraries in main memory
  - Load the library into physical memory once

Dynamic Linking

- Idea: ***shared libraries*** (.so) or ***dynamically linked libraries*** (.dll, .dylib) use virtual memory so that multiple programs can share the same libraries in main memory
    - Load the library into physical memory once
    - Each program using it has a virtual address V that points to it

Dynamic Linking

- Idea: ***shared libraries*** (.so) or ***dynamically linked libraries*** (.dll, .dylib) use virtual memory so that multiple programs can share the same libraries in main memory
    - Load the library into physical memory once
    - Each program using it has a virtual address V that points to it
    - During dynamic linking, resolve references to library symbols using that virtual address V

Dynamic Linking

- Idea: *shared libraries* (.so) or *dynamically linked libraries* (.dll, .dylib) use virtual memory so that multiple programs can share the same libraries in main memory
    - Load the library into physical memory once
    - Each program using it has a virtual address V that points to it
    - During dynamic linking, resolve references to library symbols using that virtual address V
- This is another example of the old adage: every problem in computer science can be solved by either adding a cache or **another layer of abstraction**

Dynamic Linking

- Idea: ***shared libraries*** (.so) or ***dynamically linked libraries*** (.dll, .dylib) use virtual memory so that multiple programs can share the same libraries in main memory
  - Load the library into physical memory once
  - Each program using it has a virtual address V that points to it
  - During dynamic linking, resolve references to library symbols using that virtual address V
- This is another example of the old adage: every problem in computer science can be solved by either adding a cache or **another layer of abstraction**
- What could **go wrong** with this plan? Code? Security?

Relocations in the DLL

- Since we are sharing the code to math.dll, we **cannot** set its relocations separately for each client

Relocations in the DLL

- Since we are sharing the code to math.dll, we **cannot** set its relocations separately for each client
- So if math.dll has a jump to loc_{math_label}, that must be resolved to the **same location** (e.g., 0x1234) for **all clients**

Relocations in the DLL

- Since we are sharing the code to math.dll, we **cannot** set its relocations separately for each client
- So if math.dll has a jump to loc_{math_label}, that must be resolved to the **same location** (e.g., 0x1234) for **all clients**
    - Because we can only patch the instruction once!

Relocations in the DLL

- Since we are sharing the code to math.dll, we **cannot** set its relocations separately for each client
- So if math.dll has a jump to loc_{math_label}, that must be resolved to the **same location** (e.g., 0x1234) for **all clients**
  - Because we can only patch the instruction once!
  - And every thread/program shares that patched code!

Relocations in the DLL

- Since we are sharing the code to math.dll, we **cannot** set its relocations separately for each client
- So if math.dll has a jump to loc_{math_label}, that must be resolved to the **same location** (e.g., 0x1234) for **all clients**
    - Because we can only patch the instruction once!
    - And every thread/program shares that patched code!
- So either:

Relocations in the DLL

- Since we are sharing the code to math.dll, we **cannot** set its relocations separately for each client
- So if math.dll has a jump to loc_{math_label}, that must be resolved to the **same location** (e.g., 0x1234) for **all clients**
  - Because we can only patch the instruction once!
  - And every thread/program shares that patched code!
- So either:
  - Every program using math.dll agrees to put it at virtual address location 0x1000 (what might **go wrong**?)

Relocations in the DLL

- Since we are sharing the code to math.dll, we **cannot** set its relocations separately for each client
- So if math.dll has a jump to loc_{math_label}, that must be resolved to the **same location** (e.g., 0x1234) for **all clients**
  - Because we can only patch the instruction once!
  - And every thread/program shares that patched code!
- So either:
  - Every program using math.dll agrees to put it at virtual address location 0x1000 (what might **go wrong**?)
  - math.dll uses **no relocations** in its **code segment** (how?)

Position-Independent Code

- Rather than “0x1000: jump to 0x1060”, use “jump to PC+0x60”

Position-Independent Code

- Rather than “0x1000: jump to 0x1060”, use “jump to PC+0x60”
  - This code can be relocated to any address

Position-Independent Code

- Rather than “0x1000: jump to 0x1060”, use “jump to PC+0x60”
  - This code can be relocated to any address
  - This is called ***position-independent code*** (“***PIC***”)

Position-Independent Code

- Rather than "0x1000: jump to 0x1060", use "jump to PC+0x60"
    - This code can be relocated to any address
    - This is called ***position-independent code*** ("***PIC***")
        - By default, Gradescope's Ubuntu environment expects **all code** to be position independent (for security reasons)
            - This is why you compile with `gcc -no-pie`

Position-Independent Code

- Rather than “0x1000: jump to 0x1060”, use “jump to PC+0x60”
    - This code can be relocated to any address
    - This is called ***position-independent code*** (“***PIC***”)
        - By default, Gradescope’s Ubuntu environment expects **all code** to be position independent (for security reasons)
            - This is why you compile with `gcc -no-pie`
- OK, that works for branches.

Position-Independent Code

- Rather than "0x1000: jump to 0x1060", use "jump to PC+0x60"
  - This code can be relocated to any address
  - This is called ***position-independent code*** ("***PIC***")
    - By default, Gradescope's Ubuntu environment expects **all code** to be position independent (for security reasons)
      - This is why you compile with `gcc -no-pie`
- OK, that works for branches.
- But what about global variables?

Position-Independent Code

- Rather than “0x1000: jump to 0x1060”, use “jump to PC+0x60”
  - This code can be relocated to any address
  - This is called ***position-independent code*** (“***PIC***”)
    - By default, Gradescope’s Ubuntu environment expects **all code** to be position independent (for security reasons)
      - This is why you compile with `gcc -no-pie`
- OK, that works for branches.
- But what about global variables?
  - You tell me:
    - Where should they live?
    - Should they be shared?

Data Linkage Table

- Store shared-library global variable addresses starting at some virtual address *B*

Data Linkage Table

- Store shared-library global variable addresses starting at some virtual address *B*
  - This table of addresses is the ***linkage table***

Data Linkage Table

- Store shared-library global variable addresses starting at some virtual address *B*
  - This table of addresses is the ***linkage table***
- Compile the PIC assuming that some distinguished register **GP** (or R5 or ...) will hold the current value of *B*

Data Linkage Table

- Store shared-library global variable addresses starting at some virtual address *B*
  - This table of addresses is the ***linkage table***
- Compile the PIC assuming that some distinguished register **GP** (or R5 or ...) will hold the current value of *B*
  - **Problems**? What might go wrong with this scheme?

Data Linkage Table

- Store shared-library global variable addresses starting at some virtual address *B*
  - This table of addresses is the ***linkage table***
- Compile the PIC assuming that some distinguished register **GP** (or R5 or ...) will hold the current value of *B*
  - **Problems**? What might go wrong with this scheme?
- The entry point to a shared library (or the caller) sets register GP to hold *B*

Data Linkage Table

- Store shared-library global variable addresses starting at some virtual address *B*
  - This table of addresses is the ***linkage table***
- Compile the PIC assuming that some distinguished register **GP** (or R5 or ...) will hold the current value of *B*
  - **Problems**? What might go wrong with this scheme?
- The entry point to a shared library (or the caller) sets register GP to hold *B*
  - Optimization: if the code and data live at fixed offsets, can do e.g. GP = ((PC & 0xFF00)+0x0100)

Shared Library = Shared Data?

- Typically each client of a shared library X wants its **own copies** of X's globals

Shared Library = Shared Data?

- Typically each client of a shared library X wants its **own copies** of X's globals
  - Example: **errno** variable in libc (cf. Exceptions lecture later)

Shared Library = Shared Data?

- Typically each client of a shared library X wants its **own copies** of X's globals
  - Example: **errno** variable in libc (cf. Exceptions lecture later)
- When dynamically linking, you share the code segment but get your own copy of the data segment

Shared Library = Shared Data?

- Typically each client of a shared library X wants its **own copies** of X's globals
  - Example: **errno** variable in libc (cf. Exceptions lecture later)
- When dynamically linking, you share the code segment but get your own copy of the data segment
  - And thus your own base address *B* to put in GP

Shared Library = Shared Data?

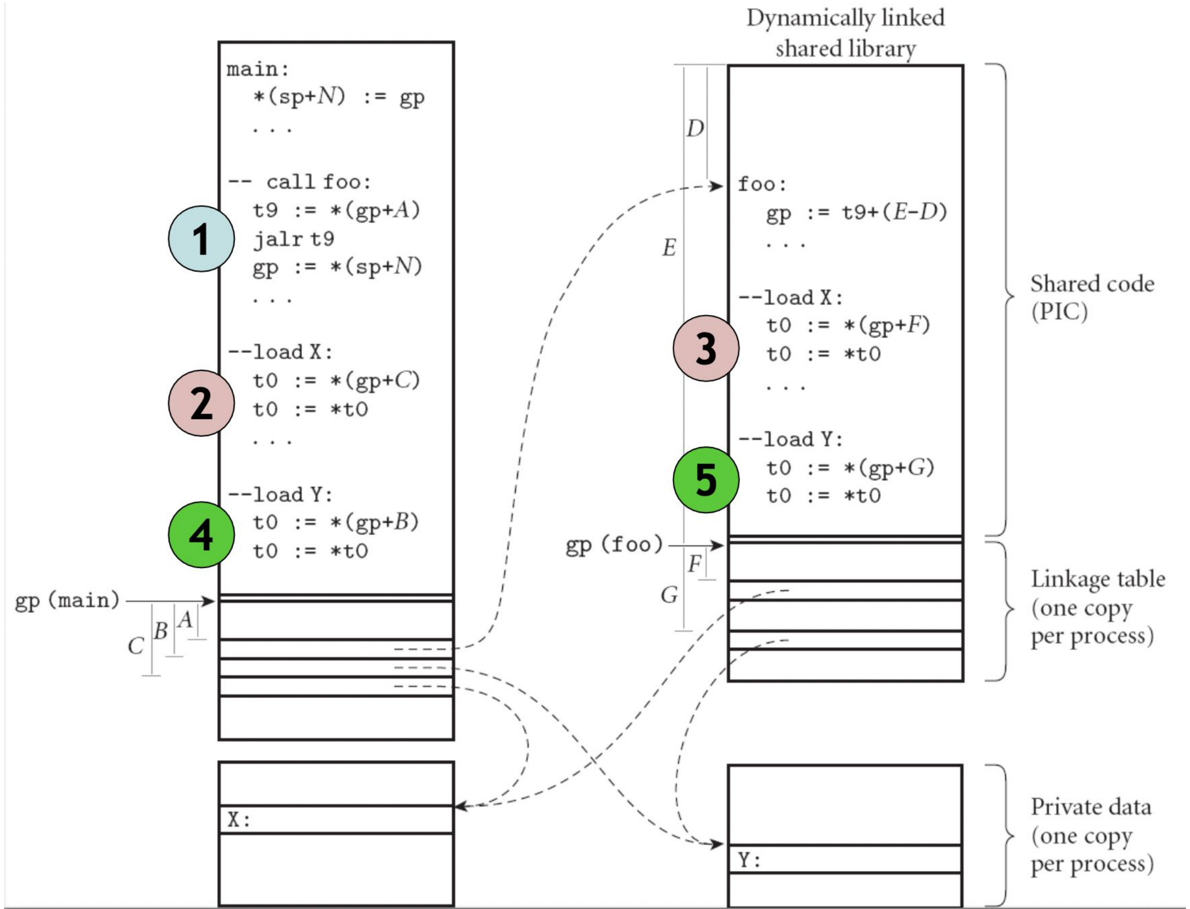
- Typically each client of a shared library X wants its **own copies** of X's globals
  - Example: **errno** variable in libc (cf. Exceptions lecture later)
- When dynamically linking, you share the code segment but get your own copy of the data segment
  - And thus your own base address *B* to put in GP
  - Optimization: use ***copy-on-write*** virtual memory

Shared Library = Shared Data?

- Typically each client of a shared library X wants its **own copies** of X's globals
  - Example: **errno** variable in libc (cf. Exceptions lecture later)
- When dynamically linking, you share the code segment but get your own copy of the data segment
  - And thus your own base address *B* to put in GP
  - Optimization: use ***copy-on-write*** virtual memory
- Detail: use an **extra level of indirection** when the PIC shared library code does **callbacks** to unshared main() or references global variables from unshared main()
  - Allows the unshared non-PIC target address to be kept in the data segment, which is private to each program

Example (Not As Bad As It Looks!)

Example (Not As Bad As It Looks!)



Fully Dynamic Linking

- So far this is all happening at load time when you start the program

Fully Dynamic Linking

- So far this is all happening at load time when you start the program
- Could we do it at run-time **on demand**?
  - Decrease load times with many libraries
  - Support dynamically-loaded code (e.g., Java)
  - Important for scripting languages

Fully Dynamic Linking

- So far this is all happening at load time when you start the program
- Could we do it at run-time **on demand**?
    - Decrease load times with many libraries
    - Support dynamically-loaded code (e.g., Java)
    - Important for scripting languages
- Use linkage table as before

Fully Dynamic Linking

- So far this is all happening at load time when you start the program
- Could we do it at run-time **on demand**?
  - Decrease load times with many libraries
  - Support dynamically-loaded code (e.g., Java)
  - Important for scripting languages
- Use linkage table as before
  - But instead loading the code for foo(), point to a special **stub procedure** that loads foo() and all variables from the library and then updates the linkage table to point to the newly-loaded foo()

Typechecking

- So we have separate compilation and shared libraries

Typechecking

- So we have separate compilation and shared libraries
- But, do we have them **safely**?

Typechecking

- So we have separate compilation and shared libraries
- But, do we have them **safely**? Consider the following:

```
(* Main *)

extern string sqrt();
void main() {
  string str = sqrt();
  printf("%s\n",str);
  return;
}
```

```
(* math *)

export double sqrt(double a) {
     return ...;
}
```

Header/Interface Files

- When we typecheck a piece of code we generate an ***interface file*** (or ***header file***)

Header/Interface Files

- When we typecheck a piece of code we generate an ***interface file*** (or ***header file***)
  - Listing all exported methods ***and their types***

Header/Interface Files

- When we typecheck a piece of code we generate an ***interface file*** (or ***header file***)
    - Listing all exported methods ***and their types***
    - Listing all exported globals ***and their types***

Header/Interface Files

- When we typecheck a piece of code we generate an ***interface file*** (or ***header file***)
  - Listing all exported methods ***and their types***
  - Listing all exported globals ***and their types***
  - The imp map and class map from PA2 suffice perfectly: just throw away the expression information

Header/Interface Files

- When we typecheck a piece of code we generate an ***interface file*** (or ***header file***)
  - Listing all exported methods ***and their types***
  - Listing all exported globals ***and their types***
  - The imp map and class map from PA2 suffice perfectly: just throw away the expression information
- When we compile a client of a library we check the interface file for the types of external symbols

Header/Interface Files

- When we typecheck a piece of code we generate an ***interface file*** (or ***header file***)
    - Listing all exported methods ***and their types***
    - Listing all exported globals ***and their types***
    - The imp map and class map from PA2 suffice perfectly: just throw away the expression information
- When we compile a client of a library we check the interface file for the types of external symbols
    - Can anything **go wrong** with this plan?

Header/Interface Files: Bait and Switch

- Cunning evil plan to deceive the user:

Header/Interface Files: Bait and Switch

- Cunning evil plan to deceive the user:
  - Write math.cl where sqrt() returns a **string**
  - Generate interface file
  - Give interface file to user

Header/Interface Files: Bait and Switch

- Cunning evil plan to deceive the user:
    - Write math.cl where sqrt() returns a **string**
    - Generate interface file
    - Give interface file to user
    - Write new math.cl: sqrt() returns a **double**
    - Compile source to relocatable object file
    - Give object file to user

Header/Interface Files: Bait and Switch

- Cunning evil plan to deceive the user:
    - Write math.cl where sqrt() returns a **string**
    - Generate interface file
    - Give interface file to user
    - Write new math.cl: sqrt() returns a **double**
    - Compile source to relocatable object file
    - Give object file to user
    - ...
    - Profit?

Header/Interface Files: Bait and Switch

- Cunning evil plan to deceive the user:
  - Write math.cl where sqrt() returns a **string**
  - Generate interface file
  - Give interface file to user
  - Write new math.cl: sqrt() returns a **double**
  - Compile source to relocatable object file
  - Give object file to user
  - ...
  - Profit?

How might we **prevent** this from happening (even accidentally)?

Checksums and Name Mangling

- From the interface file, take all of the exported symbols and all of their types and write them into a list, then **hash** (or "***checksum***") it

Checksums and Name Mangling

- From the interface file, take all of the exported symbols and all of their types and write them into a list, then **hash** (or "***checksum***") it
  - Include this hash value in the relocatable object

Checksums and Name Mangling

- From the interface file, take all of the exported symbols and all of their types and write them into a list, then **hash** (or "***checksum***") it
  - Include this hash value in the relocatable object
- Each library client also computes the hash value based on the interface it was given

Checksums and Name Mangling

- From the interface file, take all of the exported symbols and all of their types and write them into a list, then **hash** (or "***checksum***") it
    - Include this hash value in the relocatable object
- Each library client also computes the hash value based on the interface it was given
- At link time, **check to make sure** the hash values are the same

Checksums and Name Mangling

- From the interface file, take all of the exported symbols and all of their types and write them into a list, then **hash** (or "***checksum***") it
  - Include this hash value in the relocatable object
- Each library client also computes the hash value based on the interface it was given
- At link time, **check to make sure** the hash values are the same
  - C++ ***name mangling*** is the same idea, but done on a per symbol basis (rather than a per-interface basis)

Shared Libraries: Summary

- We wanted **separate compilation** for program pieces.

Shared Libraries: Summary

- We wanted **separate compilation** for program pieces.
  - So we must **link** those compiled pieces together later.

Shared Libraries: Summary

- We wanted **separate compilation** for program pieces.
    - So we must **link** those compiled pieces together later.
    - We must **resolve** references from one object to another.
        - This involves a copious number of tables

Shared Libraries: Summary

- We wanted **separate compilation** for program pieces.
  - So we must **link** those compiled pieces together later.
  - We must **resolve** references from one object to another.
    - This involves a copious number of tables
- We also wanted to **share libraries** between programs.

Shared Libraries: Summary

- We wanted **separate compilation** for program pieces.
  - So we must **link** those compiled pieces together later.
  - We must **resolve** references from one object to another.
    - This involves a copious number of tables
- We also wanted to **share libraries** between programs.
  - To do so, we used indirection to create position-independent code

Shared Libraries: Summary

- We wanted **separate compilation** for program pieces.
  - So we must **link** those compiled pieces together later.
  - We must **resolve** references from one object to another.
    - This involves a copious number of tables
- We also wanted to **share libraries** between programs.
  - To do so, we used indirection to create position-independent code
- We also wanted to **typecheck** separately-compiled modules.

Shared Libraries: Summary

- We wanted **separate compilation** for program pieces.
  - So we must **link** those compiled pieces together later.
  - We must **resolve** references from one object to another.
    - This involves a copious number of tables
- We also wanted to **share libraries** between programs.
  - To do so, we used indirection to create position-independent code
- We also wanted to **typecheck** separately-compiled modules.
  - We distribute header files with typing information and a checksum that ensures integrity