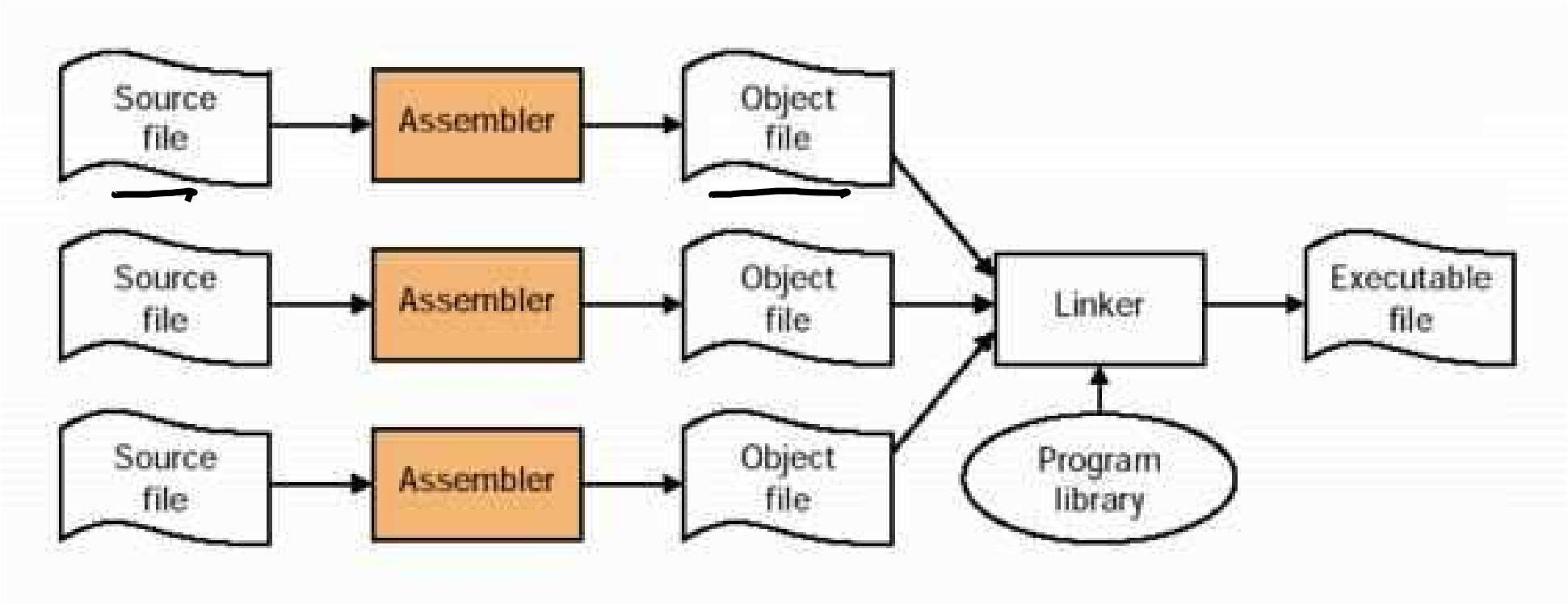
The compilation process

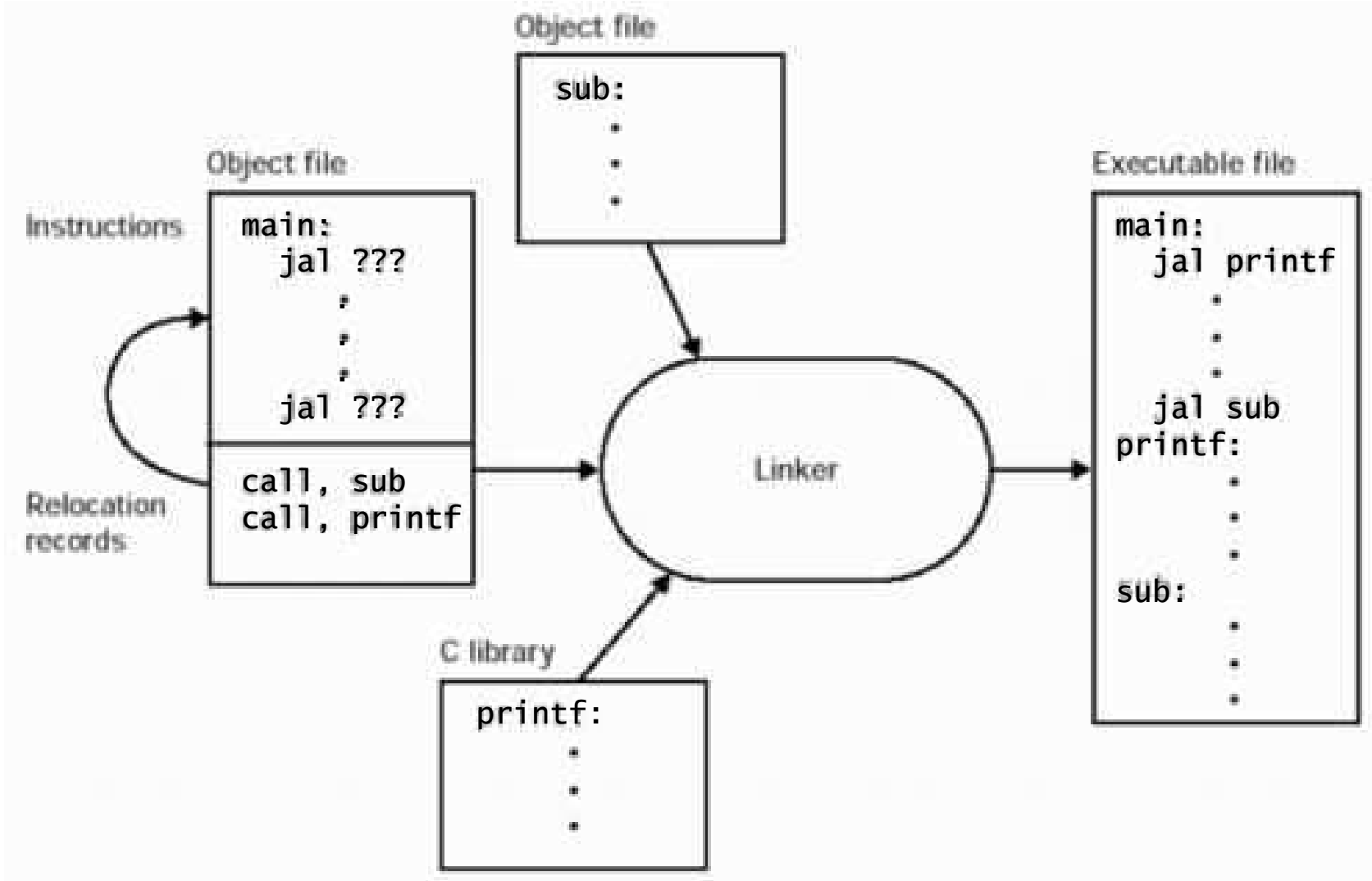
- To produce assembly code: `gcc -S test.c`
  - produces `test.s`
- To produce object code: `gcc -c test.c`
  - produces `test.o`
- To produce executable code: `gcc test.c`
  - produces `a.out`

The purpose of a linker



- The **linker** is a program that takes one or more object files and assembles them into a single executable program.
- The linker resolves references to undefined symbols by finding out which other object defines the symbol in question, and replaces placeholders with the symbol's address.

What the linker does



Loader

- Before we can start executing a program, the O/S must **load** it:

- Loading involves 5 steps:
  1. Allocates memory for the program's execution
  2. Copies the text and data segments from the executable into memory
  3. Copies program arguments (command line arguments) onto the stack
  4. Initializes registers: sets `$sp` to point to top of stack, clears the rest
  5. Jumps to start routine, which: 1) copies `main`'s arguments off of the stack, and 2) jumps to `main`.

Compiler

- Purpose: convert high-level code into low-level assembly
- Four key steps:

  lexing ⟶ parsing ⟶ code optimizations ⟶ code generation

  ⟵ CS 421, 426 ⟶

- Code generation:
  - instruction selection (depends on ISA)
  - instruction scheduling (later in the course)
  - register allocation (today's topic)

- Question: is instruction selection easier with a RISC or a CISC ISA?
- Answer: No clear answer. With a RISC ISA, there are fewer assembly instructions to choose from, which limits the search for appropriate instructions. On the other hand, a CISC ISA is more "high-level", therefore closer to a programming language, which may simplify the translation of high-level code into low-level code.

Register allocation

- The compiler initially produces “intermediate” code that assumes an infinite number of registers $t0, $t1, … and maps each variable to a unique register

- To get actual code, variables must share registers

- Suppose there are only 4 real registers `$t1`, `$t2`, `$t3`, `$t4`

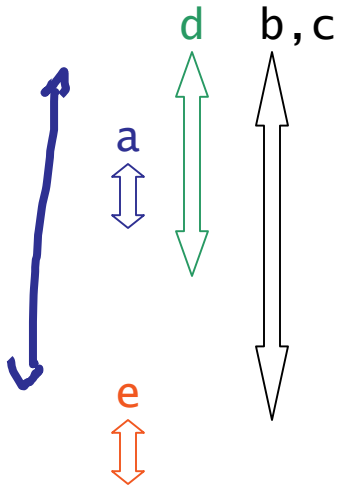
- An easy case: only 4 variables are active in any given scope

```
int b, c, d, e;
{
  int a = b + c;
  if(a < d)
    d = d + 1;
}
e = b + c;
```

Live variable analysis

- Scope defines "live range" of variable
  - but it's an *over-estimate*, e.g. `d`'s "live range" is smaller than its scope, since it isn't used again after the statement `d = d + 1`

```
int b, c, d, e;
{
  int a = b + c;
  if(a < d)
    d = d + 1;
}
e = b + c;
```



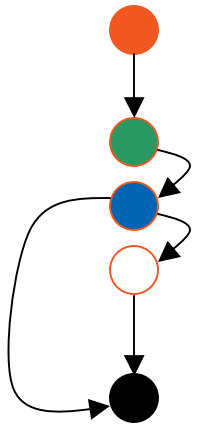
- A variable *x* is live after statement *s* if:
  - *x* is defined at or before *s*
  - there is a statement *t* after *s* that uses *x*, and
  - *x* is not re-defined before *t*

Graph representation

- A *graph* is a collection of vertices and edges
  - vertices represent entities ("things")
  - edges represent interactions

- A program can be represented as a graph (Control Flow Graph)
  - each statement is a vertex
  - edge $s \longrightarrow t$ whenever control can jump from *s* to *t*

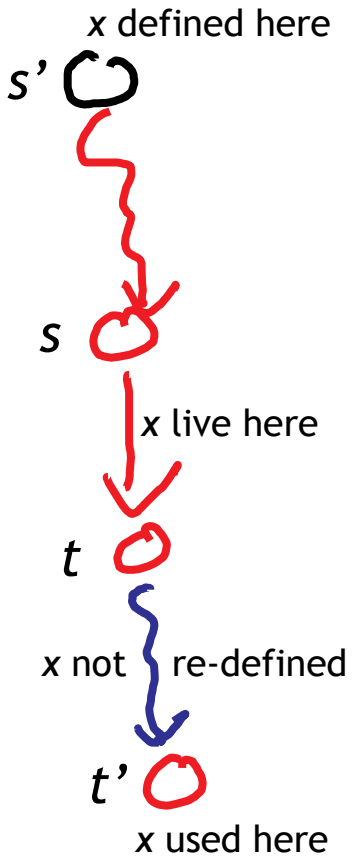
```
int b, c, d, e;
{
  int a = b + c;
  if(a < d)
    d = d + 1;
}
e = b + c;
```



Liveness

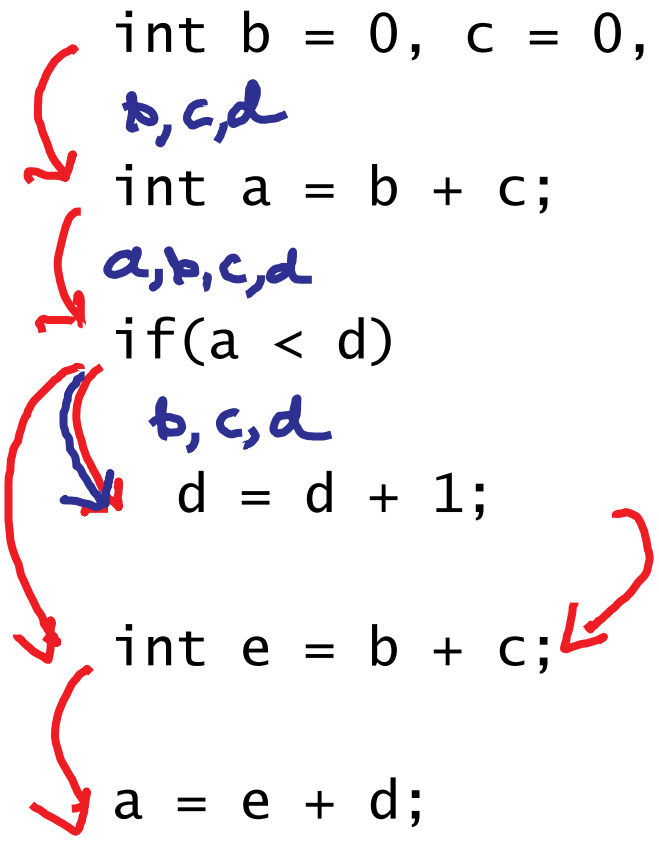
- For statement *s*, define
  use(*s*) = set of variables used (read) by *s*
  def(*s*) = set of variables defined (written) by *s*

- Variable *x* is live on edge $s \longrightarrow t$ if
  - *x* is in *def*(*s'*) for some *s'* that can reach *s*
  - *x* is in *use*(*t'*) for some *t'* reachable from *t*
  - there is a path from *t* to *t'* along which *x* is not defined



Example

- Compute the set of variables that are live at every edge in:



Accommodating live variables

- What if there are *k* registers, and > *k* variables are live on some edge?
  - Must spill some registers (e.g. to the stack)

- *Register Allocation Problem* asks two key questions:
  - are we *forced* to spill registers?
  - if so, *which* registers should we spill?

- Key idea: model this as (another) graph!

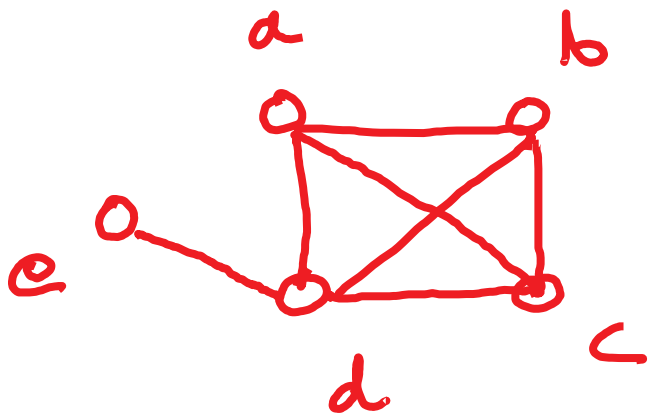
- Relying on graph-theory results, we know that this problem is hard
  - hard problems have no known *efficient* solutions (algorithms)
  - many good *heuristics* work well in practice

Interference Graph

- Entities = vertices = variables

- Interactions = edges = "live together at some point"
  - there is an edge between *u* and *v* if *u* and *v* are both live on some edge of the Control Flow Graph

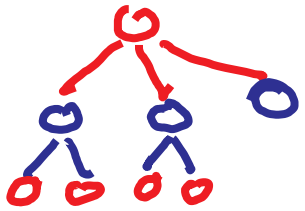
```
    int a, b, c, d, e;
b, c, d
    a = b + c;
a, b, c, d
    if(a < d)
b, c, d
      d = d + 1;
b, c, d
    e = b + c;
e, d
    a = e + d;
```



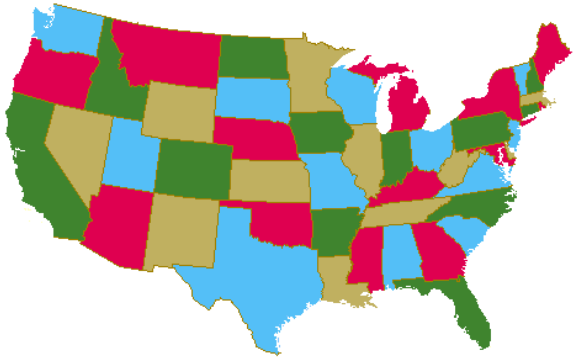
If *x* is allocated register *r*, no neighbor of *x* gets allocated register *r*

Graph Coloring

- Color the vertices of a graph with *k* colors so that no two neighboring vertices get the same color
- A *tree* is always 2-colorable



- A *map* is always 4-colorable



- There isn't an efficient way to decide if a graph is 3-colorable
  - unless "P = NP" (the biggest open problem in CS!)

Register Allocation

- Given *k* registers, decide if the interference graph can be *k*-colored
  - if so, it is always possible to assign registers to variables so that no spilling occurs
  - if not, registers must be spilled
- A brute-force solution to the graph coloring problem tries all possible combinations of vertex colorings to see if a "legal" one exists – this is a very computationally expensive process
- Since graph coloring is hard, it was *wrongly assumed* that the hard question was "do registers need to be spilled?"
  - led to belief that register allocation is harder for RISC architectures
- Interference graphs have a special structure
  - in fact, *k*-coloring for such graphs is an easy problem
  - the hard problem is "which registers to spill?"
  - this is more critical for CISC architectures, since there are very few registers, so each holds data that is potentially needed very often

Midterm 1 next Wednesday!

- Exam time: 2pm to 2:50pm
- Where?
  - Last name A – G in 1310 DCL
  - Last name H – Z in 1320 DCL
- Major topics:
  - C to MIPS, correcting MIPS [calling conventions]
  - CPU performance
- Minor topics:
  - Amdahl's law (problem-solving)
  - Interrupt handling (conceptual)
  - RISC vs. CISC (conceptual)
  - Register Allocation (conceptual)
- Practice exam released on Friday
  - Monday section is exam review (no attendance), solutions released
  - Tuesday is "all day" office hours

] 80%

Midterm 1 typically has the highest average, so make it count!