Lecture 24

CIS 341: COMPILERS

Announcements

- HW 6: Dataflow Analysis and Optimizations
  - Due: Monday , April 20

- HW 7: Optimization & Experiments
  - Due: April 29th

Registers

REGISTER ALLOCATION

Register Allocation

- Once we have the program in SSA form we can do register allocation.

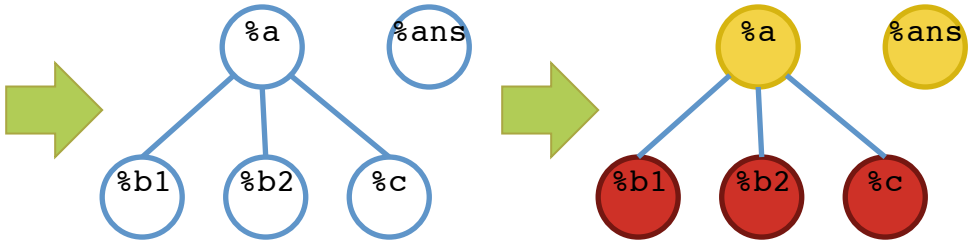
- Basic process:

1. Compute liveness information for each temporary.
2. Create an *interference graph*:
   - Nodes are temporary variables.
   - There is an edge between node n and m if n is live at the same time as m
3. Try to color the graph
   - Each color corresponds to a register
4. In case step 3. fails, "spill" a register to the stack and repeat the whole process.
5. Rewrite the program to use registers

Interference Graphs

- Nodes of the graph are `%uids`
- Edges connect variables that *interfere* with each other
  - Two variables interfere if their live ranges intersect (i.e. there is an edge in the control-flow graph across which they are both live).
- Register assignment is a *graph coloring*.
  - A graph coloring assigns each node in the graph a color (register)
  - Any two nodes connected by an edge must have different colors.
- Example:

```
// live = {%a}
%b1 = add i32 %a, 2
// live = {%a,%b1}
%c = mult i32 %b1, %b1
// live = {%a,%c}
%b2 = add i32 %c, 1
// live = {%a,%b2}
%ans = mult i32 %b2, %a
// live = {%ans}
return %ans;
```



Interference Graph

2-Coloring of the graph
red = EAX
yellow = EBX

Register Allocation Questions

- Can we efficiently find a k-coloring of the graph whenever possible?
  - Answer: in general the problem is NP-complete (it requires search)
  - But, we can do an efficient approximation using heuristics.

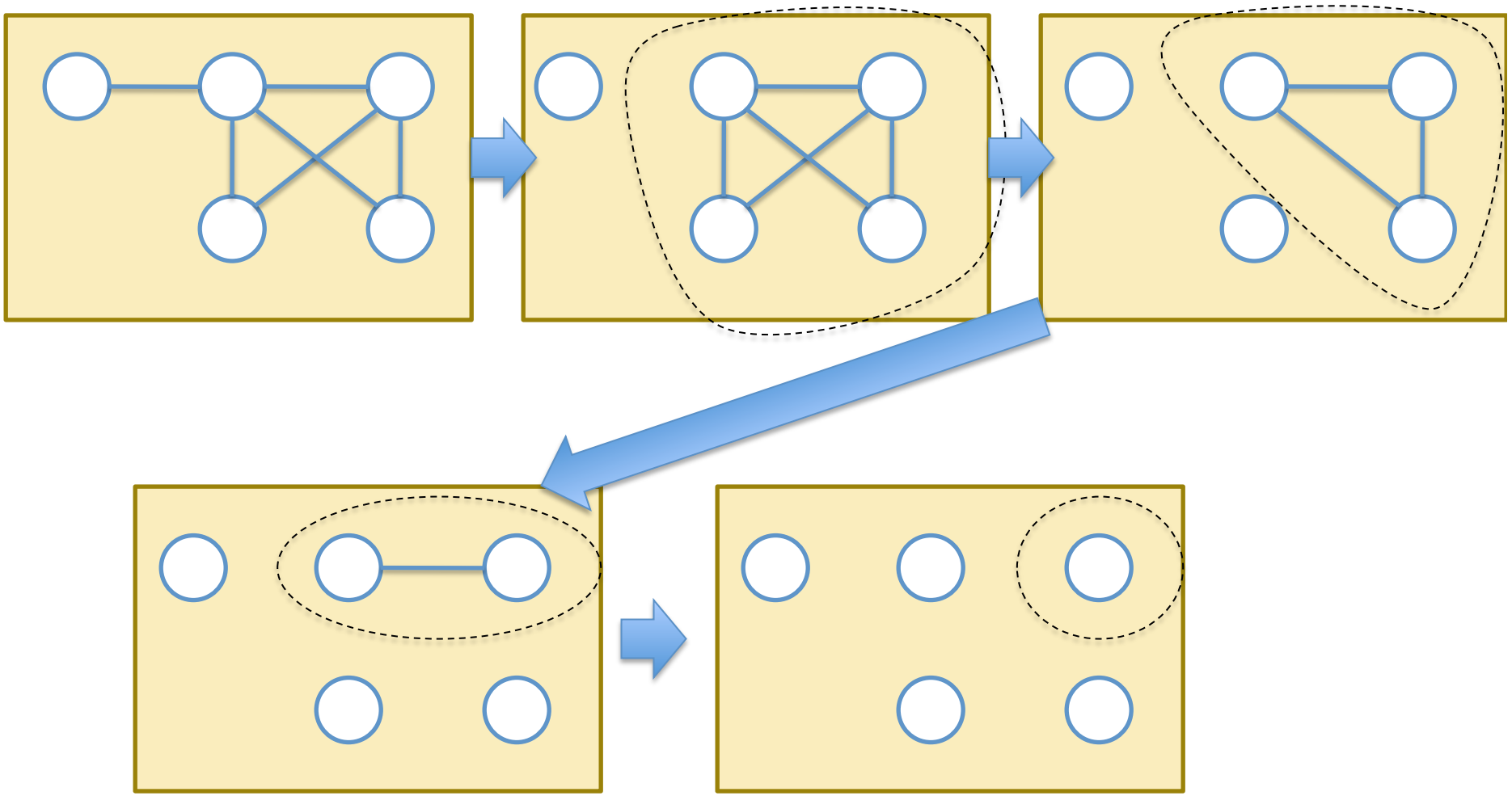
- How do we assign registers to colors?
  - If we do this in a smart way, we can eliminate redundant MOV instructions.

- What do we do when there aren't enough colors/registers?
  - We have to use stack space, but how do we do this effectively?

Coloring a Graph: Kempe's Algorithm

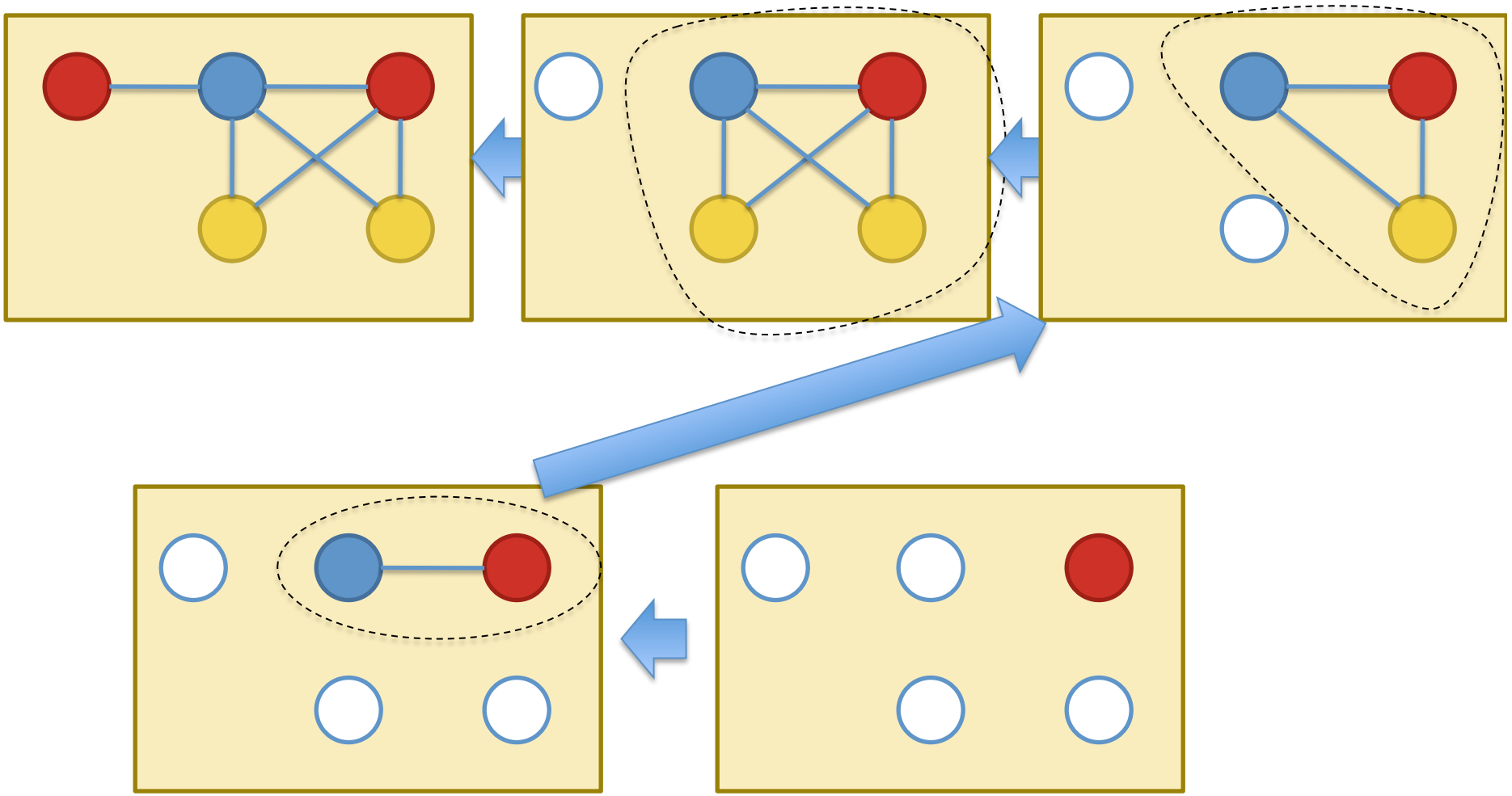
- Kempe [1879] provides this algorithm for K-coloring a graph.
- It's a recursive algorithm that works in three steps:
- Step 1: Find a node with degree < K and cut it out of the graph.
  - Remove the nodes and edges.
  - This is called *simplifying* the graph
- Step 2: Recursively K-color the remaining subgraph
- Step 3: When remaining graph is colored, there must be at least one free color available for the deleted node (since its degree was < K). Pick such a color.

Example: 3-color this Graph



Recursing Down the Simplified Graphs

Example: 3-color this Graph

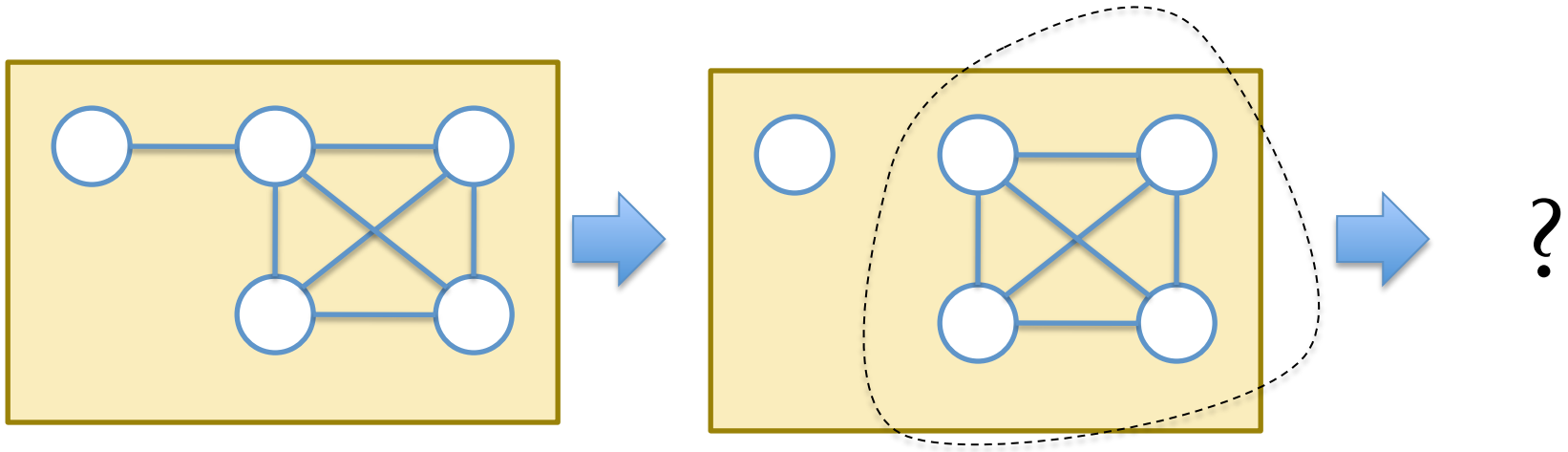


Assigning Colors on the way back up.

Failure of the Algorithm

- If the graph cannot be colored, it will simplify to a graph where every node has at least K neighbors.
  - This can happen even when the graph is K-colorable!
  - This is a symptom of NP-hardness (it requires search)

- Example: When trying to 3-color this graph:



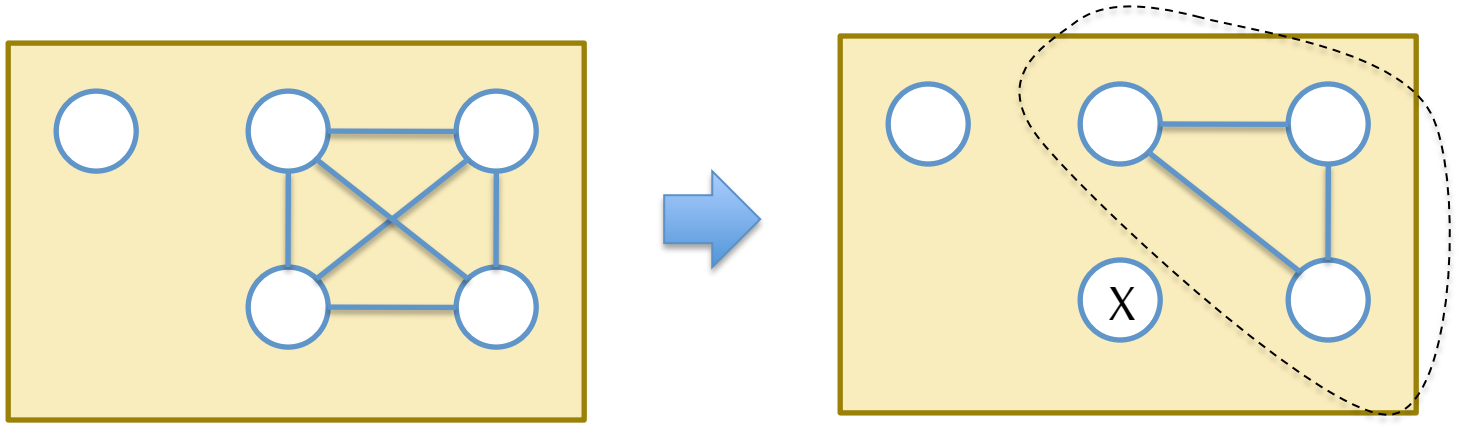
Spilling

- Idea: If we can't K-color the graph, we need to store one temporary variable on the stack.
- Which variable to spill?
  - Pick one that isn't used very frequently
  - Pick one that isn't used in a (deeply nested) loop
  - Pick one that has high interference (since removing it will make the graph easier to color)
- In practice: some weighted combination of these criteria

- When coloring:
  - Mark the node as spilled
  - Remove it from the graph
  - Keep recursively coloring

Spilling, Pictorially

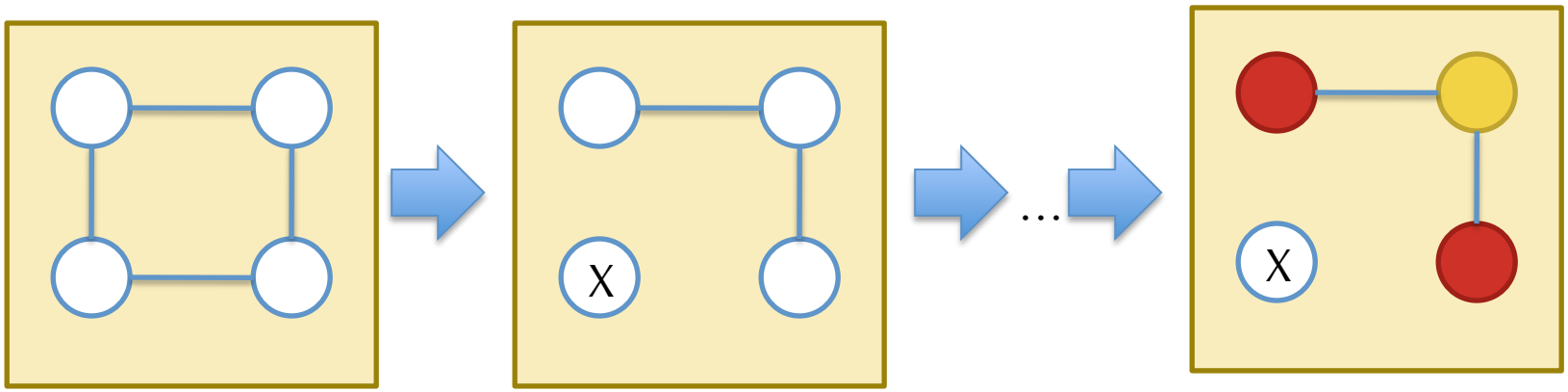
- Select a node to spill
- Mark it and remove it from the graph
- Continue coloring



Optimistic Coloring

- Sometimes it is possible to color a node marked for spilling.
  - If we get "lucky" with the choices of colors made earlier.

- Example: When 2-coloring this graph:



- Even though the node was marked for spilling, we can color it.
- So: on the way down, mark for spilling, but don't actually spill…

Accessing Spilled Registers

- If optimistic coloring fails, we need to generate code to move the spilled temporary to & from memory.
- Option 1: Reserve registers specifically for moving to/from memory.
  - Con: Need at least two registers (one for each source operand of an instruction), so decreases total # of available registers by 2.
  - Pro: Only need to color the graph once.
  - Not good on X86 (especially 32bit) because there are too few registers & too many constraints on how they can be used.

- Option 2: Rewrite the program to use a new temporary variable, with explicit moves to/from memory.
  - Pro: Need to reserve fewer registers.
  - Con: Introducing temporaries changes live ranges, so must recompute liveness & recolor graph
  - This strategy is usually used on X86.

Example Spill Code

- Suppose temporary `t` is marked for spilling to stack slot `[rbp+offs]`
- Rewrite the program like this:

```
%t = a op b;                %t = a op b                  // defn. of t
…                           movq %t, [rbp+offs]
                            …
%x = %t op c                movq [rbp+offs], %t37        // use 1 of t
…                           %x = %t37 op c
                            …
%y = d op %t                movq [rbp+offs], %t38        // use 2 of t
                            %y = d op %t38
```

- Here, `%t37` and `%t38` are freshly generated temporaries that replace `%t` for different uses of `%t`.
- Rewriting the code in this way breaks `t`'s live range up:

  `%t`, `%t37`, `%t38` are only live across one edge

Precolored Nodes

- Some variables must be pre-assigned to registers.
  - E.g. on X86 the multiplication instruction: IMul must define %rax
  - The "Call" instruction should kill the caller-save registers %rax, %rcx, %rdx.
  - Any temporary variable live across a call interferes with the caller-save registers.

- To properly allocate temporaries, we treat registers as nodes in the interference graph with pre-assigned colors.
  - Pre-colored nodes can't be removed during simplification.
  - Trick: Treat pre-colored nodes as having "infinite" degree in the interference graph – this guarantees they won't be simplified.
  - When the graph is empty except the pre-colored nodes, we have reached the point where we start coloring the rest of the nodes.

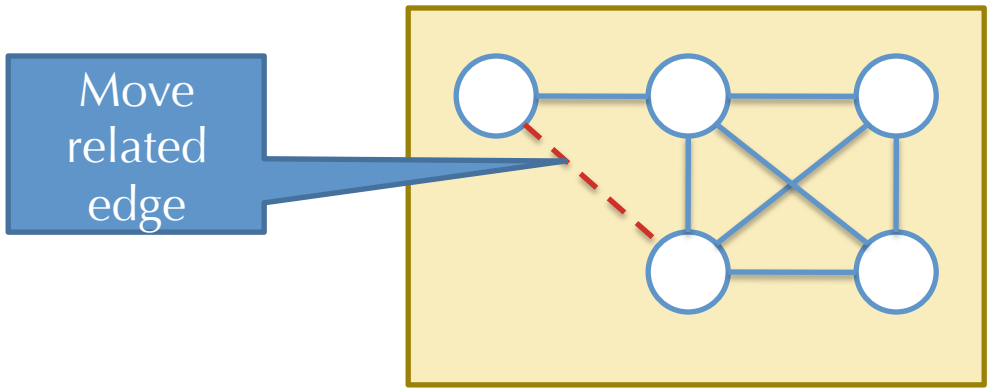
Picking Good Colors

- When choosing colors during the coloring phase, *any* choice is semantically correct, but some choices are better for performance.
- Example:
  `movq %t1, %t2`
  - If t1 and t2 can be assigned the same register (color) then this move is redundant and can be eliminated.

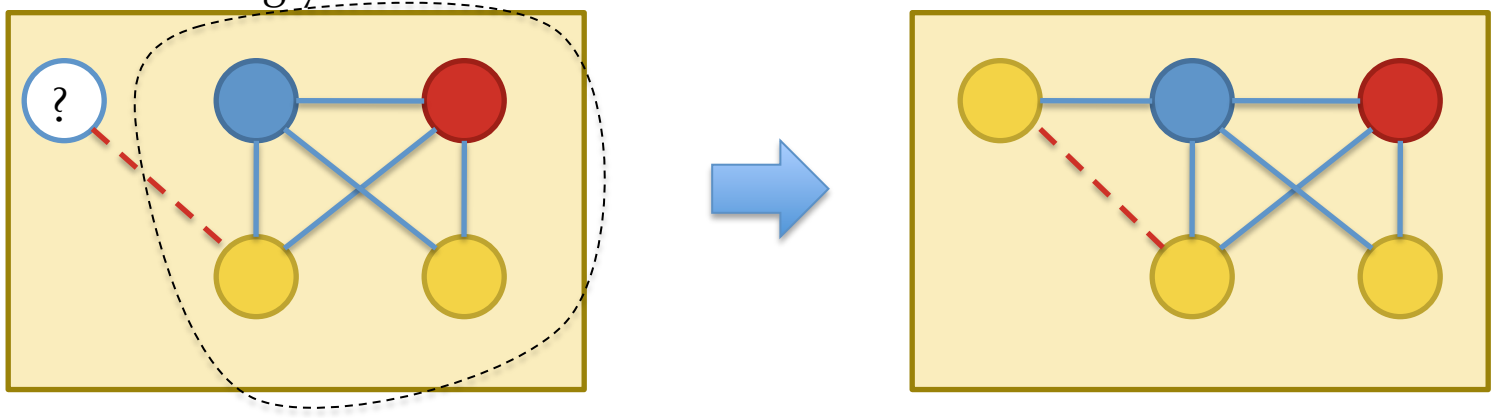
- A simple color choosing strategy that helps eliminate such moves:
  - Add a new kind of "move related" edge between the nodes for t1 and t2 in the interference graph.
  - When choosing a color for t1 (or t2), if possible pick a color of an already colored node reachable by a move-related edge.

Example Color Choice

- Consider 3-coloring this graph, where the dashed edge indicates that there is a Mov from one temporary to another.

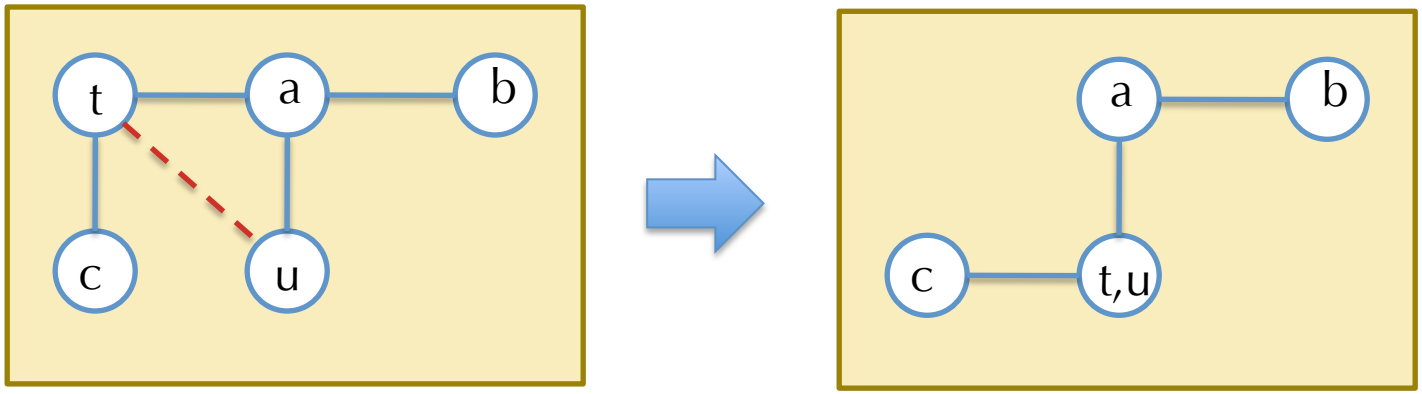


- After coloring the rest, we have a choice:
  - Picking yellow is better than red because it will eliminate a move.

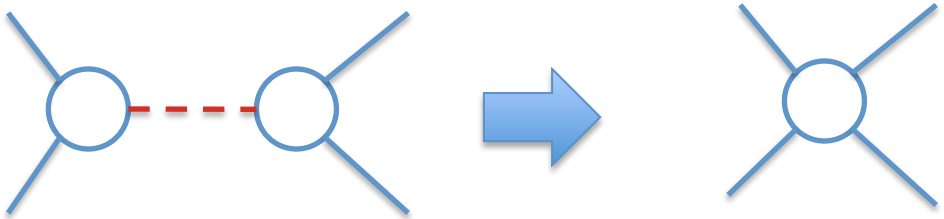


Coalescing Interference Graphs

- A more aggressive strategy is to *coalesce* nodes of the interference graph if they are connected by move-related edges.
  - Coalescing the nodes *forces* the two temporaries to be assigned the same register.



- Idea: interleave simplification and coalescing to maximize the number of moves that can be eliminated.
- Problem: coalescing can sometimes increase the degree of a node.



Conservative Coalescing

- Two strategies are guaranteed to preserve the k-colorability of the interference graph.

- *Brigg's strategy*: It's safe to coalesce x & y if the resulting node will have fewer than k neighbors (with degree ≥ k).

- *George's strategy:* We can safely coalesce x & y if for every neighbor t of x, either t already interferes with y or t has degree < k.

Complete Register Allocation Algorithm

1. Build interference graph (precolor nodes as necessary).
   - Add move related edges
2. Reduce the graph (building a stack of nodes to color).
   1. Simplify the graph as much as possible without removing nodes that are move related (i.e. have a move-related neighbor). Remaining nodes are high degree or move-related.
   2. Coalesce move-related nodes using Brigg's or George's strategy.
   3. Coalescing can reveal more nodes that can be simplified, so repeat 2.1 and 2.2 until no node can be simplified or coalesced.
   4. If no nodes can be coalesced *freeze* (remove) a move-related edge and keep trying to simplify/coalesce.
3. If there are non-precolored nodes left, mark one for spilling, remove it from the graph and continue doing step 2.
4. When only pre-colored node remain, start coloring (popping simplified nodes off the top of the stack).
   1. If a node must be spilled, insert spill code as on slide 14 and rerun the whole register allocation algorithm starting at step 1.

Last details

- After register allocation, the compiler should do a peephole optimization pass to remove redundant moves.
- Some architectures (e.g. x86-64) specify calling conventions that use *registers* to pass function arguments.
  - It's helpful to move such arguments into temporaries in the function prelude so that the compiler has as much freedom as possible during register allocation.
  - When compiling C (or Oat), the default LLVM compilation strategy achieves this by using alloca to create storage space for function parameters. Subsequent alloca promotion turns them into temporaries.

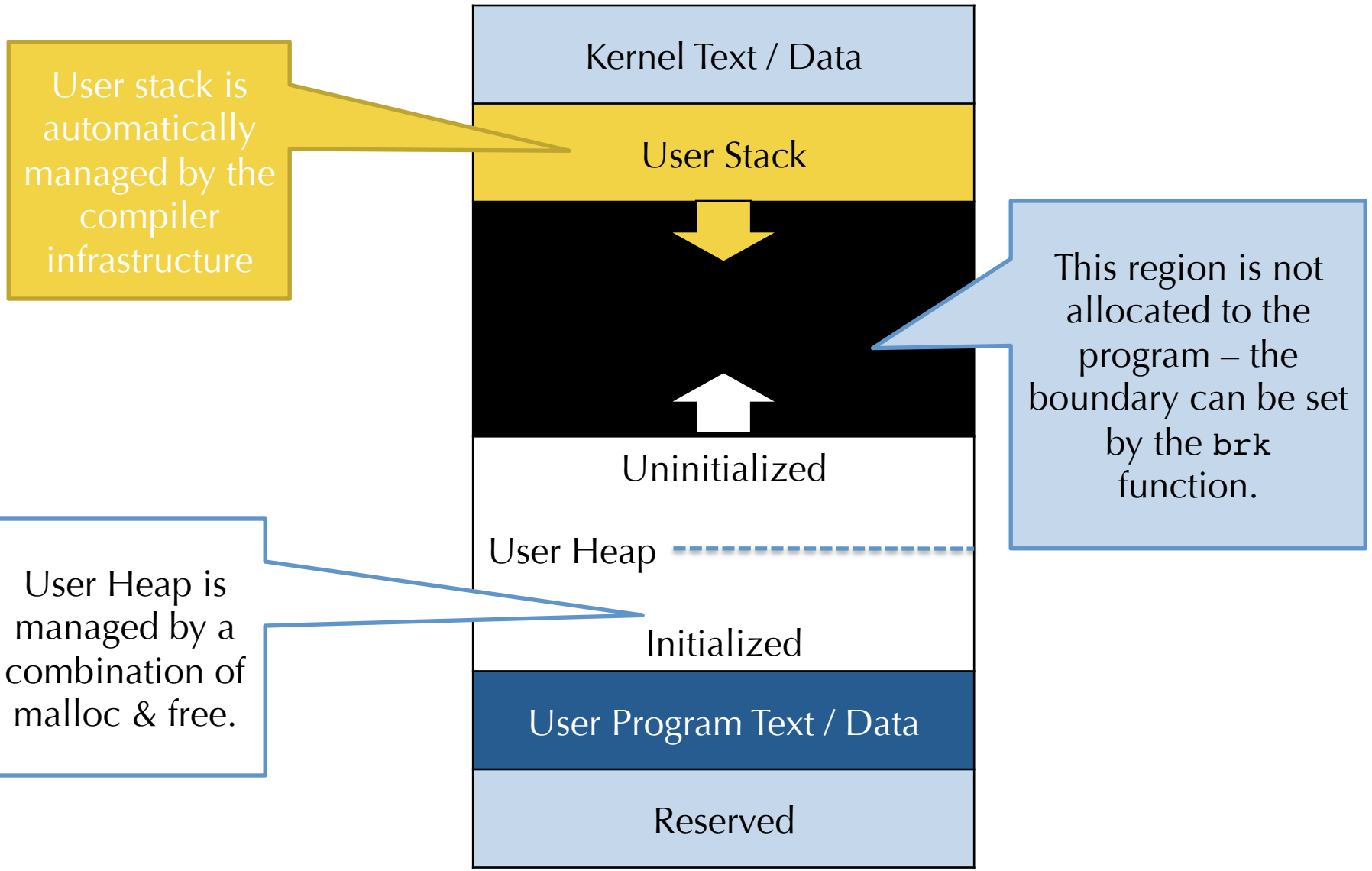
MEMORY MANAGEMENT

Memory Management

- Program data is stored in memory.
  - Memory is a finite resource: programs may need to reuse some of it.
- Most programming languages provide two means of structuring data stored in memory:
- *Stack*: memory space (stack frames) for storing data local to a function body.
  - The programming language provides facilities for automatically managing stack-allocated data. (i.e. compiler emits code for allocating/freeing stack frames)
  - (Aside: Unsafe languages like C/C++ don't enforce the stack invariant, which leads to bugs that can be exploited for code injection attacks…)
- *Heap*: memory space for storing data that is created by a function but needed in a caller. (Its lifetime is unknown at compile time.)
  - Freeing/reusing this memory can be up to the programmer (C/C++)
  - (Aside: Freeing memory twice or never freeing it also leads to many bugs in C/C++ programs…)
  - *Garbage collection* automates memory management for Java/ML/C#/etc.

EXPLICIT MEMORY MANAGEMENT

Unix Memory Layout



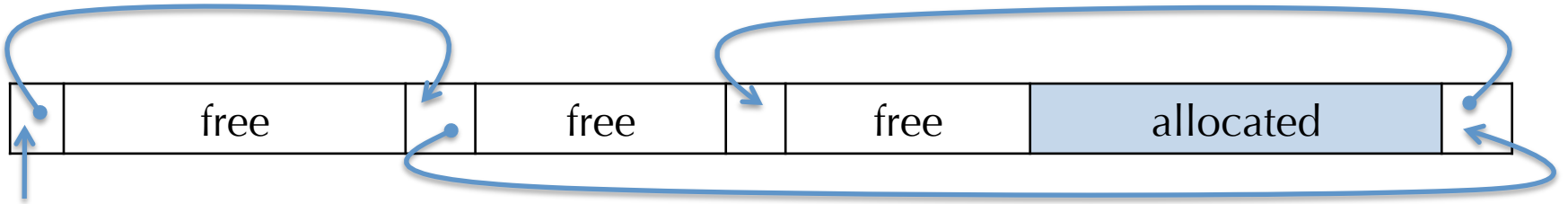
Explicit Memory Management

- On unix, libc provides a library that allows programmers to manage the heap:
- `void * malloc(size_t n)`
  - Allocates `n` bytes of storage on the heap and returns its address.
- `void free(void *addr)`
  - Releases the memory previously allocated by `malloc` address `addr`.

- These are user-level library functions. Internally, `malloc` uses `brk` (or `sbrk`) system calls to have the kernel allocate space to the process.

Simple Implementation: Free Lists

- Arrange the blocks of unused memory in a *free list*.
  - Each block has a pointer to the next free block.
  - Each block keeps track of its size. (Stored before & after data parts.)
  - Each block has a status flag = allocated or unallocated (Kept as a bit in the first size (assuming size is a multiple of 2 so the last bit is unused)



- Malloc: walk down free list, find a block big enough
  - First fit? Best fit?
- Free: insert the freed block into the free list.
  - Perhaps keep list sorted so that adjacent blocks can be merged.
- Problems:
  - Fragmentation ruins the heap
  - Malloc can be slow

Exponential Scaling / Buddy System

- Keep an array of freelists: FreeList[i]
  - FreeList[i] points to a list of blocks of size 2^i

- Malloc: round requested size up to nearest power of 2
  - When FreeList[i] is empty, divide a block from FreeList[i+1] into two halves, put both chunks into FreeList[i]
  - Alternatively, merge together two adjacent nodes from FreeList[i-1]
- Free: puts freed block back into appropriate free list

- Malloc & free take O(1) time
- This approach trades external fragmentation (within the heap as a whole) for internal fragmentation (within each block).
  - Wasted space: ~30%

GARBAGE COLLECTION

Why Garbage Collection?

- Manual memory management is cumbersome & error prone:
  - Freeing the same pointer twice is ill defined (seg fault or other bugs)
  - Calling free on some pointer not created by `malloc` (e.g. to an element of an array) is also ill defined
  - `malloc` and `free` aren't modular: To properly free all allocated memory, the programmer has to know what code "owns" each object. Owner code must ensure free is called just once.
  - Not calling free leads to *space leaks*: memory never reclaimed
    - Many examples of space leaks in long-running programs

- Garbage collection:
  - Have the language runtime system determine when an allocated chunk of memory will no longer be used and free it automatically.
  - But… garbage collector is usually the most complex part of a language's runtime system.
  - Garbage collection does impose costs (performance, predictability)

Memory Use & Reachability

- When is a chunk of memory no longer needed?
  - In general, this problem is undecidable.

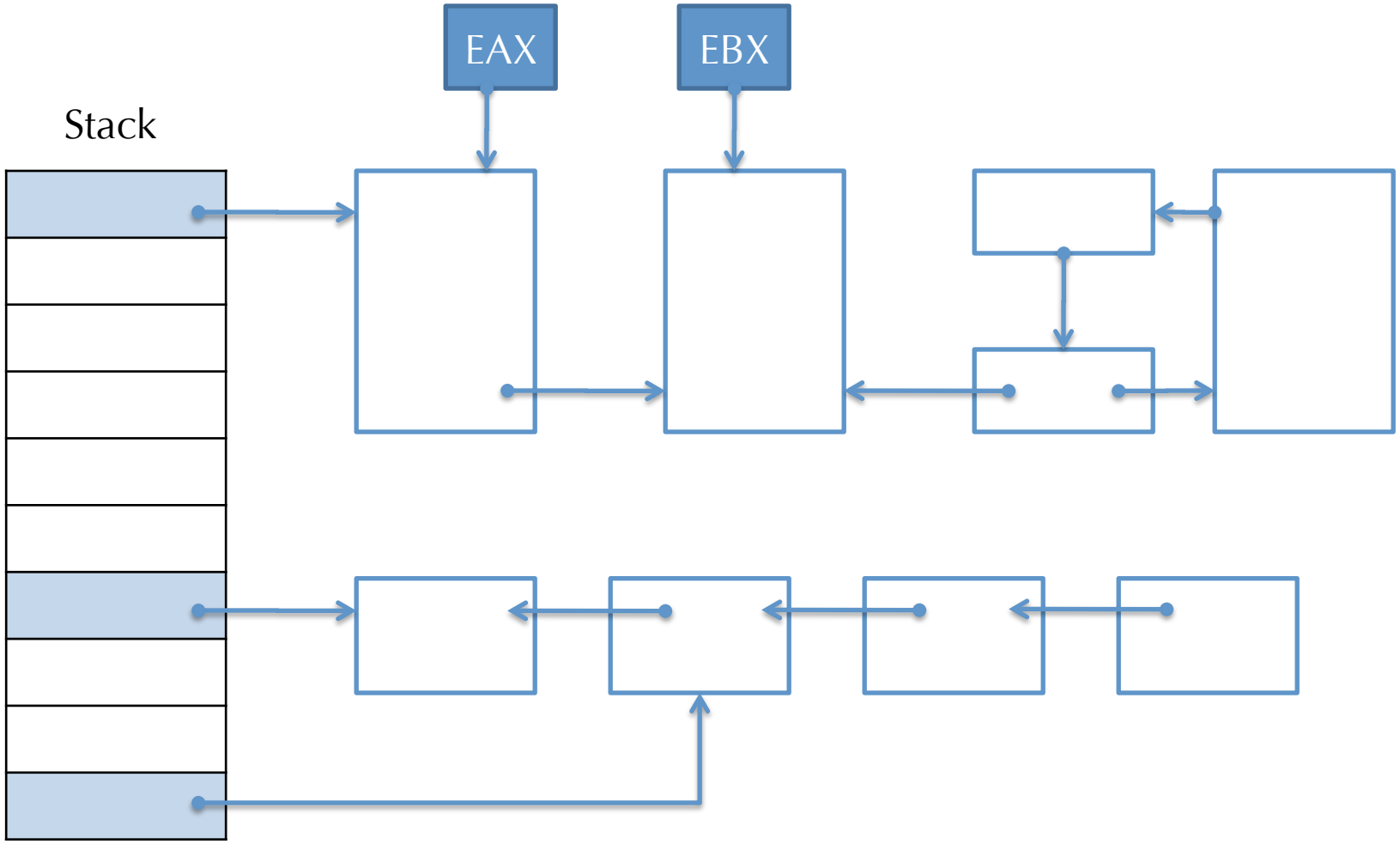
- We can approximate this information by freeing memory that can't be reached from any *root* references.
  - A *root pointer* is one that might be accessible directly from the program (i.e. they're not in the heap).
  - Root pointers include pointer values stored in registers, in global variables, or on the stack.
- If a memory cell is part of a record (or other data structure) that can be reached by traversing pointers from the root, it is *live*.
- It is safe to reclaim all memory cells not reachable from a root (such cells are *garbage*).

Reachability & Pointers

- Starting from stack, registers, & globals (*roots*), determine which objects in the heap are reachable following pointers.
- Reclaim any object that isn't reachable.
- Requires being able to distinguish pointer values from other values (e.g., ints).
- Type safe languages:
  - OCaml, SML/NJ use the low bit:
    1 it's a scalar, 0 it's a pointer. (Hence 31-bit ints in OCaml)
  - Java puts the tag bits in the object meta-data (uses more space).
  - Type safety implies that casts can't introduce new pointers
  - Also, pointers are abstract (references), so objects can be moved without changing the meaning of the program
- Unsafe languages:
  - Pointers aren't abstract, they can't be moved.
  - Boehm-Demers-Weiser *conservative* collector for C use heuristics: (e.g., the value doesn't point into an allocated object, pointers are multiples of 4, etc.)
  - May not find as much garbage due to conservativity.

Example Object Graph

- Pointers in the stack, registers, and globals are *roots*



MARK & SWEEP GC

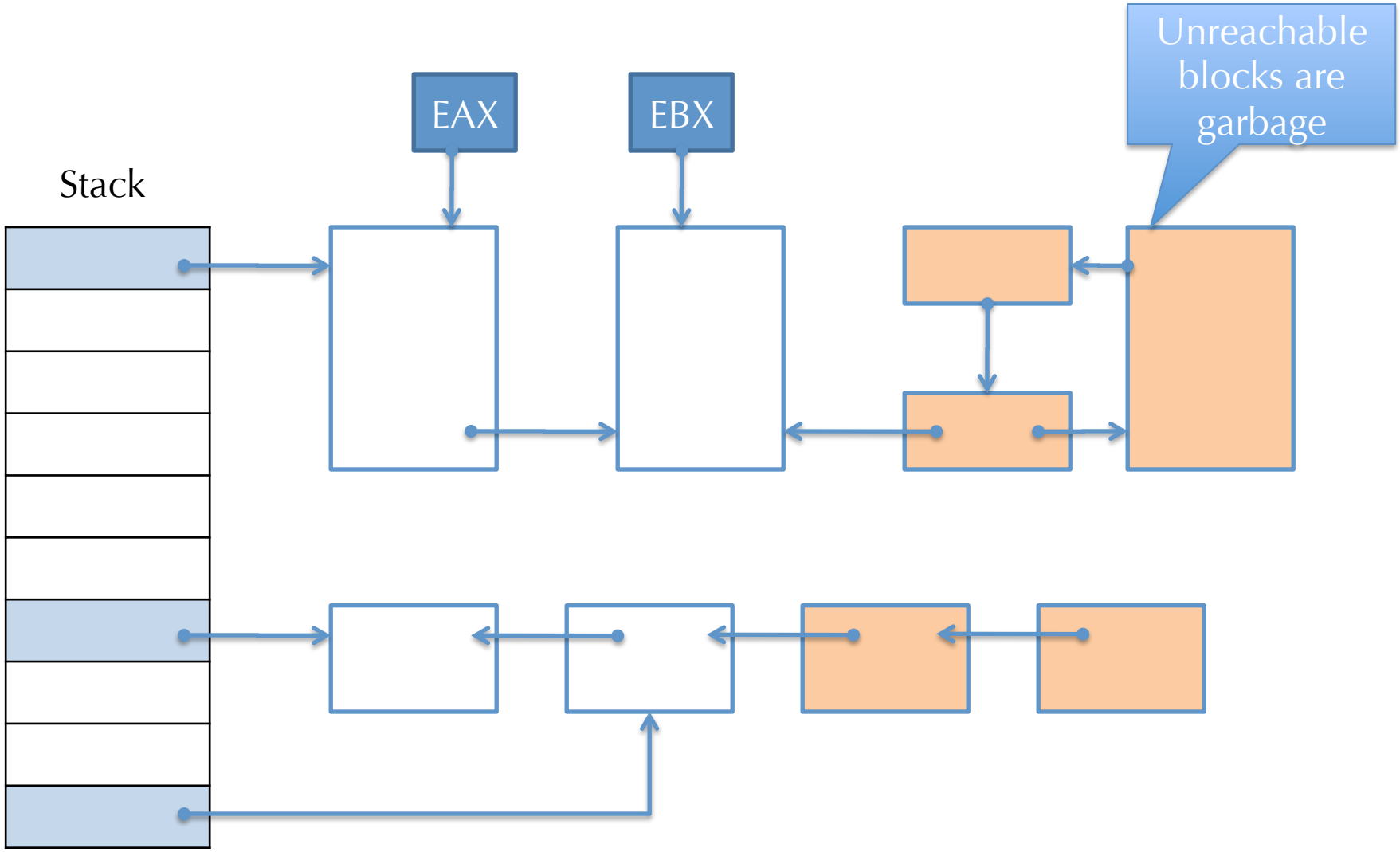
Mark and Sweep Garbage Collection

- Classic algorithm with two phases:

- Phase 1: Mark
  - Start from the roots
  - Do depth-first traversal, marking every object reached.

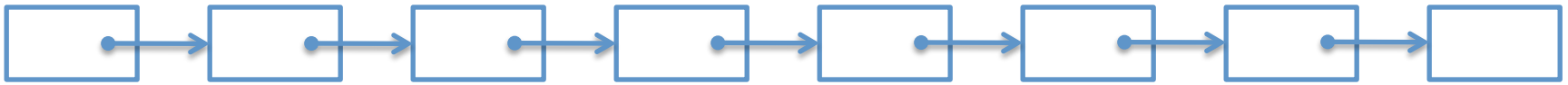
- Phase 2: Sweep
  - Walk over *all* allocated objects and check for marks.
  - Unmarked objects are reclaimed.
  - Marked objects have their marks cleared.
  - Optional: compact all live objects in heap by moving them adjacent to one another. (needs extra work & indirection to "patch up" pointers)

Results of Marking Graph



Implementing the Mark Phase

- Depth-first search has a natural recursive algorithm.
- Question: what happens when traversing a long linked list?



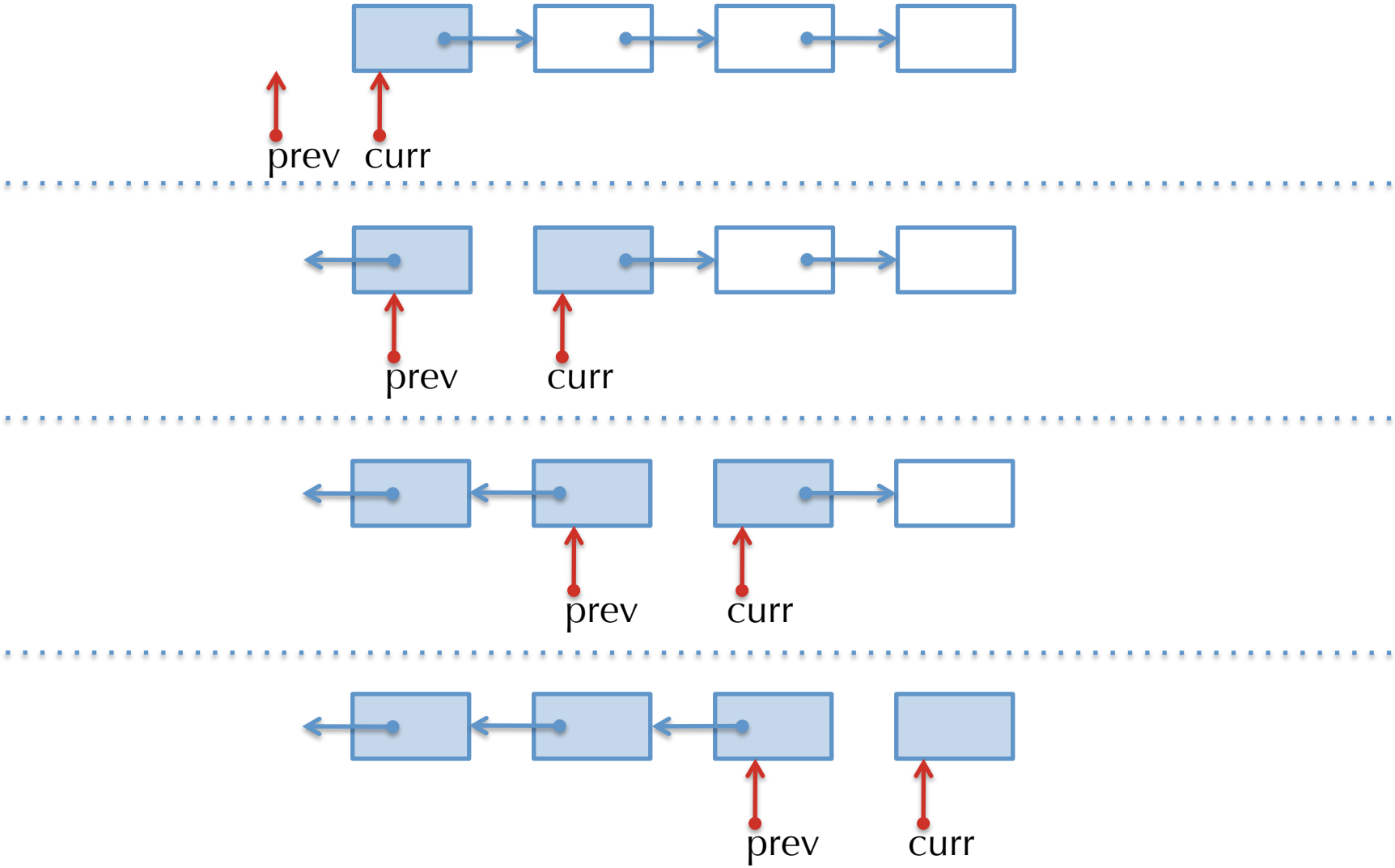
- Where do we store the information needed to perform the traversal?
  - (In general, garbage collectors are tricky to implement because if they allocate memory who manages that?!)

Deutsch-Schorr-Waite (DSW) Algorithm

- No need for a stack, it is possible to use the graph being traversed itself to store the data necessary…
- Idea: during depth-first-search, each pointer is followed only once. The algorithm can reverse the pointers on the way down and restore them on the way back up.
  - Mark a bit on each object traversed on the way down.

- Two pointers:
  - curr: points to the current node
  - prev points to the previous node
- On the way down, flip pointers as you traverse them:
  - tmp := curr
    curr := curr.next
    tmp.next := prev
    prev := curr

Example of DSW (traversing down)



Costs & Implications

- Need to generalize to account for objects that have multiple outgoing pointers.
- Depth-first traversal terminates when there are no children pointers or all children are already marked.
  - Accounts for cycles in the object graph.
- The Deutsch-Schorr-Waite algorithm breaks objects during the traversal.
  - All computation must be halted during the mark phase. (Bad for concurrent programs!)
- Mark & Sweep algorithm reads all memory in use by the program (even if it's garbage!)
  - Running time is proportional to the total amount of allocated memory (both live and garbage).
  - Can pause the programs for long times during garbage collection.