CIS 341: Compilers

Lecture 34

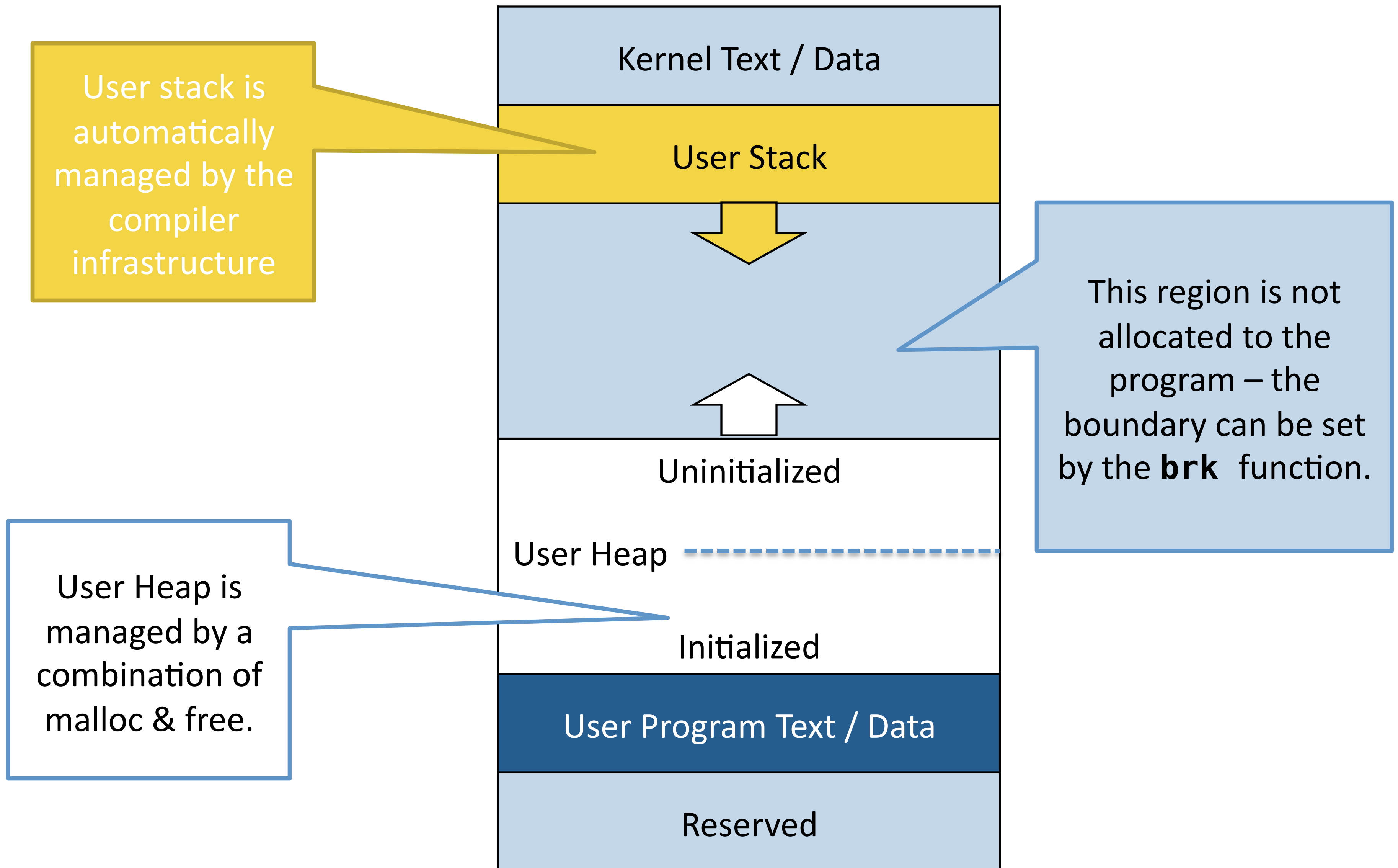
The Plan

- Today:
  - Memory management
- Project 7: Register Allocation for OAT
  - Due: Friday, December 5th at 11:59 pm
- Announcements:
  - Final Exam: Wednesday, December 10th noon – 2:00pm CHEM 514

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.

Unix Memory Layout



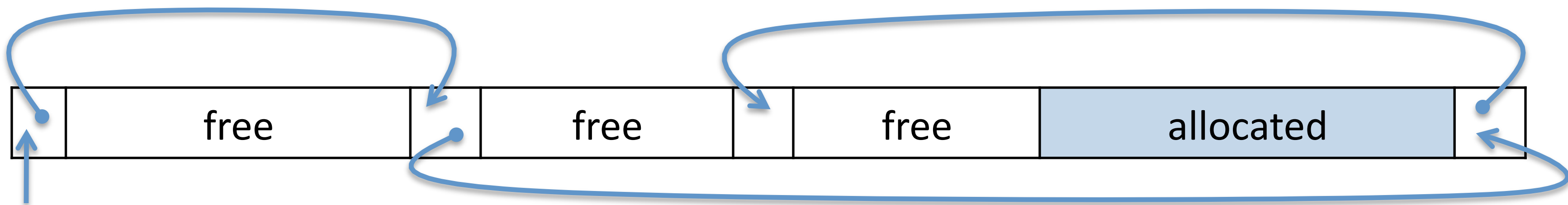
Explicit Memory Management

- 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%

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

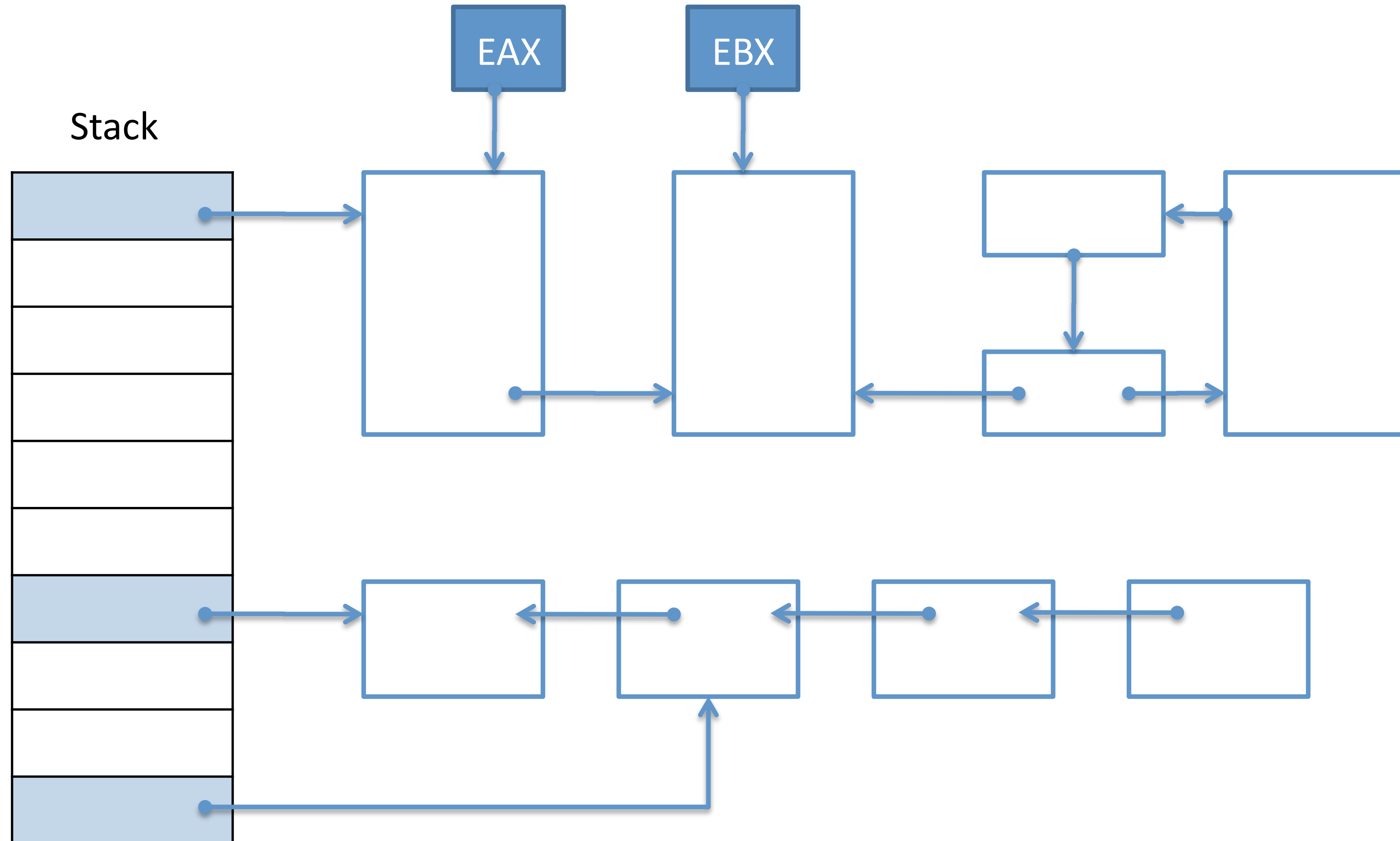
- When is a chunk of memory no longer needed?
  - In general, this problem is undecidable.
- We can approximate this information by freeing memory that can't be reached from any *root* references.
  - A *root pointer* is one that might be accessible directly from the program (i.e. they're not in the 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 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

Unreachable blocks are garbage

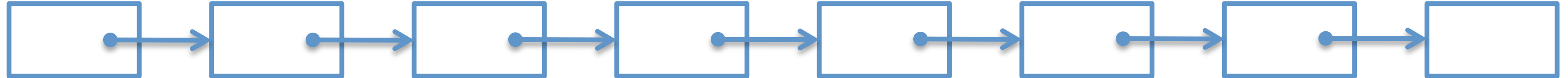
Stack

EAX

EBX

Implementing the Mark Phase

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

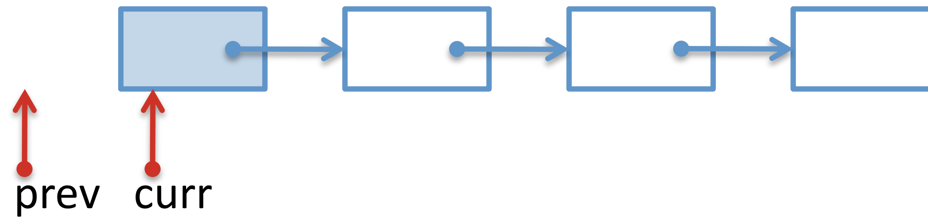


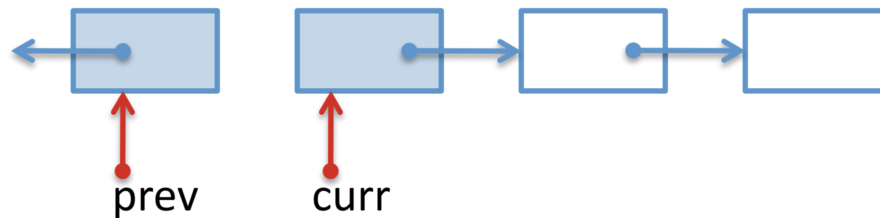
- Where do we store the information needed to perform the traversal?
  - (In general, garbage collectors are tricky to implement because if they allocate memory who manages that?!)

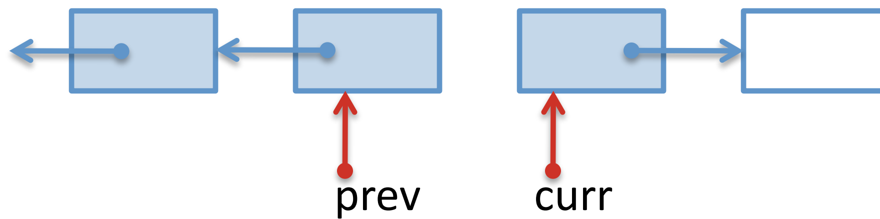
Deutsch-Schorr-Waite (DSW) Algorithm

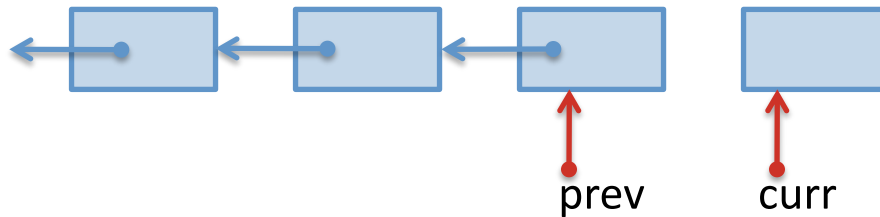
- No need for a stack, it is possible to use the graph being traversed itself to store the data necessary…
- Idea: during depth-first-search, each pointer is followed only once. The algorithm can reverse the pointers on the way down and restore them on the way back up.
  - Mark a bit on each object traversed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