CS 4400
Computer Systems

LECTURE 21

Garbage collection

Memory-related bugs in C

Garbage Collector

- A dynamic storage allocator that automatically frees allocated blocks no longer needed by the program
  - such blocks are known as ***garbage***
- Applications explicitly allocate heap blocks, but never explicitly free them

Garbage Collector

- A large number of approaches for garbage collection exist
  – We'll discuss only the ***Mark & Sweep algorithm***
- Mark & Sweep can be built on top of an existing malloc package to provide garbage collection for C and C++

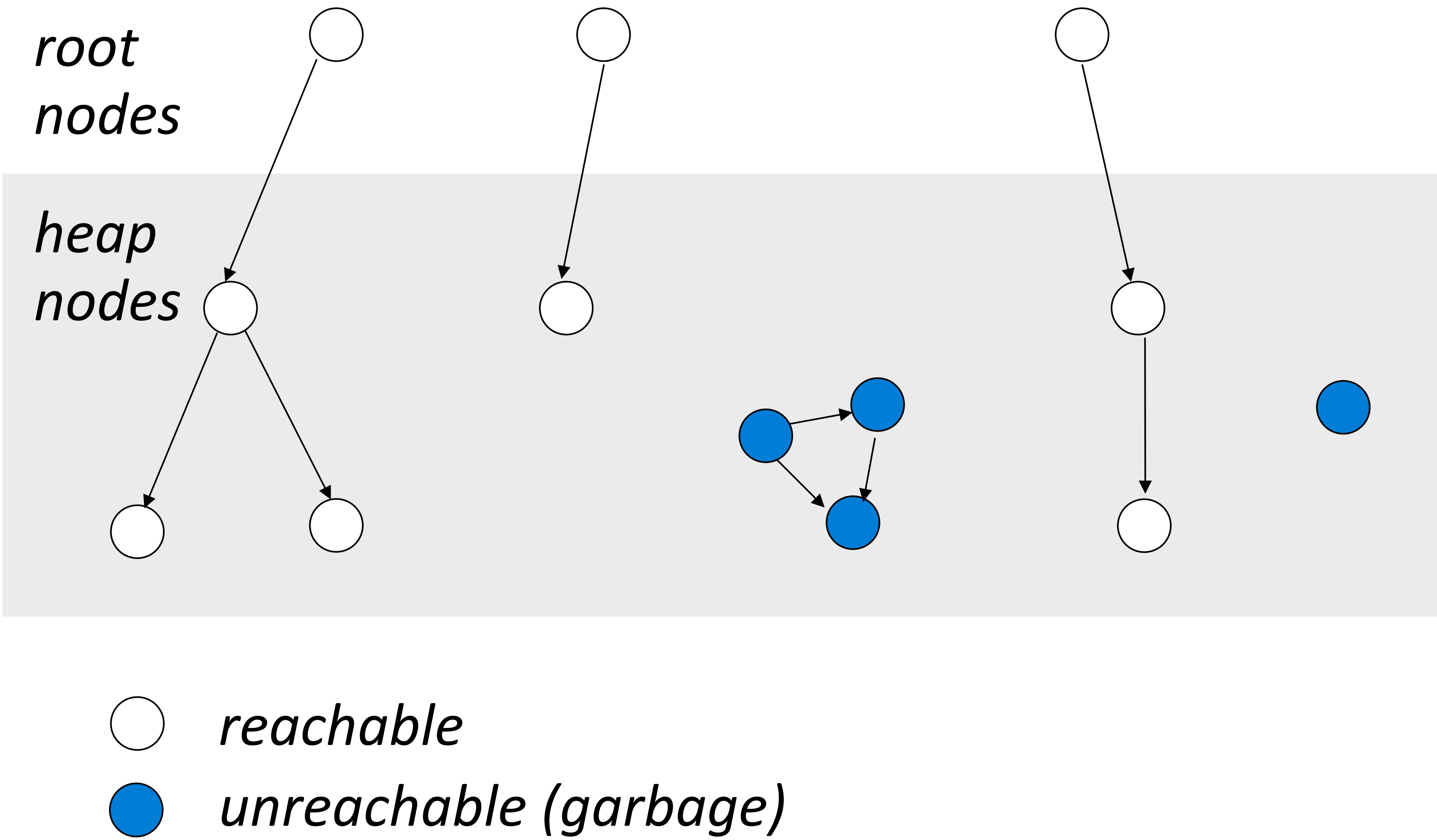
Reachability Graph

- ***Heap nodes*** correspond to an allocated heap block
  - $p \rightarrow q$ means that some location in block p points to some location in block q
- ***Root nodes*** correspond to locations not in the heap
  - can be registers, stack variables, ...

Reachability Graph

- A node *p* is reachable if there exists a directed path from any root node to *p*
  - Any unreachable node is garbage
- The garbage collector must maintain the graph and periodically reclaim unreachable nodes by freeing them

Example: Reachability Graph



Conservative Garbage Collectors

- Each reachable block is correctly identified as reachable
- Some unreachable blocks may be incorrectly identified as reachable

Conservative Garbage Collectors

- C/C++ cannot maintain an exact representation of the reachability graph, in general
  - Thus, collectors for such languages are conservative
- Collectors can provide their service on demand, or they may run as separate threads in parallel with the program
  - How can we incorporate a collector into the `malloc` package?

Mark & Sweep

- ***Mark phase***—marks all reachable and allocated descendants of the root nodes
- Calls `mark(p)` for every root node `p`
  - returns immediately if `p` does not point to an allocated and unmarked heap block
  - otherwise, marks the block and calls itself recursively on each word in the block

Mark & Sweep

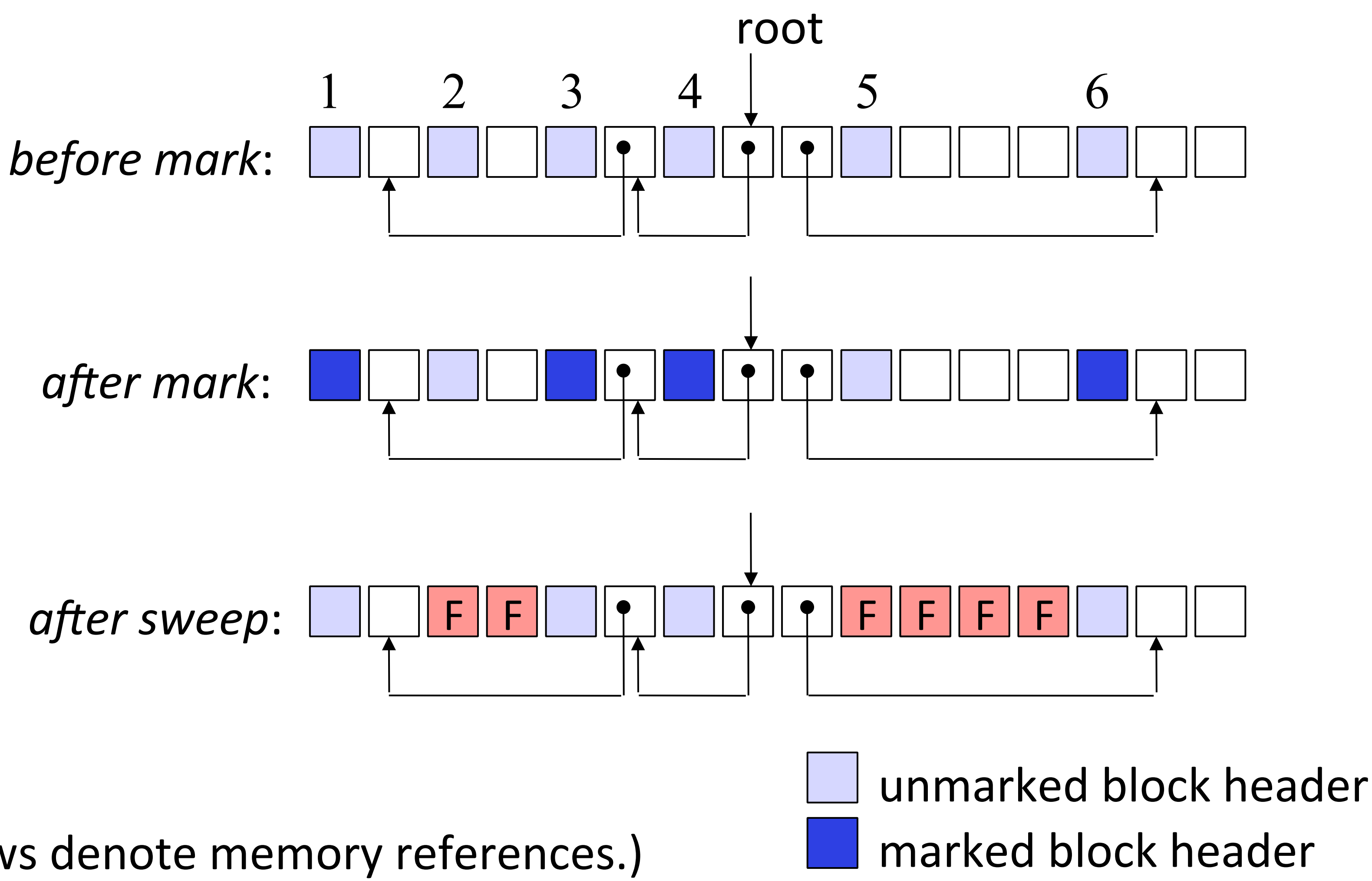
- ***Sweep phase***—frees each unmarked allocated block
- Calls `sweep(begin,end)` to iterate over every block in the heap, freeing any unmarked allocated blocks

Mark & Sweep Pseudocode

```
void mark(ptr p) {
  if((b = isPtr(p)) == NULL)    /* if p points to some */
    return;                     /* word in an allocated */
  if(blockMarked(b))            /* block, isPtr returns */
    return;                     /* a pointer to the */
  markBlock(b);                 /* beginning of that block */
  len = length(b);
  for(i = 0; i < len; i++)      /* for every word ... */
    mark(b[i]);
}

void sweep(ptr b, ptr end) {
  while(b < end) {
    if(blockMarked(b))
      unmarkBlock(b);
    else if(blockAllocated(b))
      free(b);
    b = nextBlock(b);
  }
}
```

Example: Mark & Sweep



(Arrows denote memory references.)

Initially, the heap consists of 6 unmarked allocated blocks.
After the mark phase, all nodes reachable from the root are marked.
After the sweep phase, unreachable blocks are reclaimed.

Conservative Mark & Sweep

- Implementing `isPtr(p)` in C is a challenge
- C does not tag memory with any type info
  - no obvious way to determine if `p` is a pointer
  - what if `int p` has the same value as an allocated address?

Conservative Mark & Sweep

- No obvious way to determine if `p` points to some location in the payload of an allocated block
- A balanced binary search tree of allocated blocks (ordered by address) can help to determine if `p` falls within the extent of an allocated block

GC Pros and Cons

- **Pros**:
  - Eliminates possibility of many kinds of memory bugs
  - Simplifies software
- **Cons**:
  - May add overhead
  - Causes pauses
  - Not necessarily compatible with very low-level software like OS kernels

Faster GC

- Observation: Most allocated objects are either very short-lived or very long-lived
- Fast GCs are *generational*: objects are first allocated in a nursery, and only promoted to the real heap if they survive for a while

Faster GC

- The nursery can be scanned rapidly and profitably, while the main heap does not need to be scanned very often
- Analyses done in the compiler can avoid heap allocations

Dereferencing Bad Pointers

- Typically, most of a 64-bit address space (or even 32-bit) is not mapped to any meaningful data.
  - dereferencing a pointer into unmapped memory causes a seg fault
- Some areas of virtual memory are read-only
  - writing to such an area causes a protection fault

Dereferencing Bad Pointers

- *Common bug*:

```
scanf("%d", val); // need &
```

  - contents of `val` are interpreted as an address
  - best case—program terminates with an exception
  - worst case—contents of `val` correspond to a valid read/write area of virtual memory (baffling consequences later)

Reading Uninitialized Memory

- Unlike `.bss` memory locations, heap and stack memory are not initialized to zero

```
/* returns y = Ax */

int* matvec(int** A, int* x, int n) {
  int i, j;

  int* y = Malloc(n * sizeof(int));

  for(i = 0; i < n; i++)
    for(j = 0; j < n; j++)
      y[i] += A[i][j] * x[j];

  return y;
}
```

Allowing Stack Buffer Overflows

- Recall that buffer overflow is caused by writing to a target buffer on the stack or heap without examining the size

```
void bufoverflow() {
  char buf[64];

  gets(buf);
  return;
}
```

- Better to use `fgets(stdin, 64, buf);`

Pointers & Objects Same Size?

- Assuming that pointers and the objects they point to are the same size is a common mistake

```
/* array of n ptrs, each points to m-int array */
int** makeArray1(int n, int m) {
  int i, **A = Malloc(n * sizeof(int));

  for(i = 0; i < n; i++)
    A[i] = Malloc(m * sizeof(int));

  return A;
}
```

- Runs fine if `int` and `int*` are same size.
- What happens if `int*` is larger?

Off-by-One Errors

```
/* array of n ptrs, each points to m-int array */
int** makeArray2(int n, int m) {
  int i, **A = Malloc(n * sizeof(int*));

  for(i = 0; i <= n; i++)
    A[i] = Malloc(m * sizeof(int));

  return A;
}
```

- What happens when we initialize `A[n]`?

Confusing Object & Pointer

- To avoid manipulating a pointer instead of the object it points to, be mindful of operator precedence/associativity

```
/* remove the first item in a binary heap of
 * size items, then reheapify remaining items */
int* binheapDelete(int** binheap, int* size) {
  int* packet = binheap[0];
  binheap[0] = binheap[*size-1];
  *size--;          /* should be (*size)--; */
  heapify(binheap, *size, 0);
  return packet;
}
```

- What is the consequence of decrementing the pointer instead of the actual size?

Misunderstanding Pointer Arithmetic

- Arithmetic operations on pointers are performed in units that are the size of the objects they (are intended to) point to, not necessarily 1 byte

```
/* search a 0-terminated array of ints and
 * return the first occurrence of val */
int* search(int* p, int val) {
  while(*p && *p != val)
    p += sizeof(int);   /* should be what? */

  return p;
}
```

Referencing Nonexistent Vars

```
int* stackref() {
  int val;

  return &val;
}
```

- The function returns a pointer to the local variable and then pops its stack frame. `p=stackref()` remains a valid memory address.
- What happens if the program later assigns some value to `*p`?
- Is there a problem if `val` is `static`?

Referencing Data in Free Blocks

```
int* heapref(int n, int m) {
  int i, *x, *y;

  x = Malloc(n * sizeof(int));
  ...
  free(x);
  ...

  y = Malloc(m * sizeof(int));
  for(i = 0; i < m; i++)
    y[i] = x[i]++;

  return y;
}
```

What are the values in `x`?

Introducing Memory Leaks

- Memory leaks occur when programmers forget to free allocated blocks, inadvertently creating garbage (i.e., unreachable nodes).

```
void leak(int n) {
  int* x = Malloc(n * sizeof(int));

  return;    /* the block at x is now garbage */
}
```

- If this function is called frequently, the heap will gradually fill with garbage (possibly consuming the entire virtual address space).
  - especially important for programs that never terminate

Memory Debugging Tools

- These are very useful, and should be used on all non-trivial C/C++
  - Valgrind
    - Dynamic binary rewriting, slow but super easy to use
  - Address sanitizer in Clang and recent versions of GCC
    - Checker is integrated with the compiler
    - More of pain to use but a lot faster than Valgrind
  - Purify – static binary rewriting
    - Probably not as good as Valgrind (and not free)
  - Insure++ – source code rewriting

Memory Debugging Tools

- Memory debuggers find lots of bugs but also miss lots of bugs
  - Catching all bugs is too expensive – no checker does this

Memory Management Summary

- Virtual memory is an abstraction of main memory
  - DRAM as a cache for disk memory
  - requires translation from virtual address to physical address using page tables, whose contents are maintained by the OS
  - simplifies memory management and protection

Memory Management Summary

- Even though virtual memory is provided automatically by the system, it is a finite resource
  - managing VM involves subtle time and space trade-offs
- The difficulty of memory-related errors is an important motivation for Java and C#
  - eliminate the ability to take addresses of variables
  - implicit dynamic storage allocator (no `free` or `delete`)