Chapter 6: Run-Time Environments

Yepang Liu

liuyp1@sustech.edu.cn

Outline

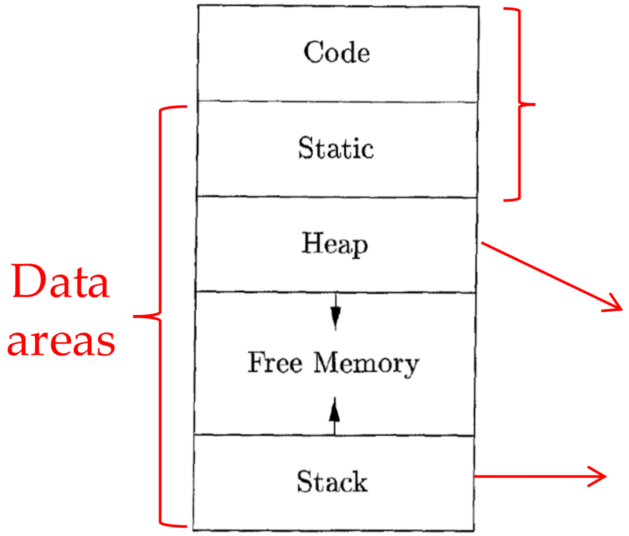
- Storage Organization
  - Static Storage
  - Dynamic Storage
- Stack Space Allocation
  - Activation Trees
  - Activation Records
  - Calling Sequences
- Heap Management
  - Memory Manager
  - Program Locality
  - Fragmentation

Run-Time Environment

- A compiler must accurately implement the abstractions in the source-language definition
  - Names, scopes, data types, operators, procedures, parameters, flow-of-control constructs…
- To do so, the compiler manages a run-time environment (运行时刻环境) and deals with:
  - Layout and allocation of storage locations for data in the source program
  - Mechanisms to access variables
  - Linkages between procedures, the mechanisms for passing parameters

Storage Organization

- Typical subdivision of run-time memory into code and data areas (in the logical address space)



Statically determined areas:

- *Code*: target code
- *Static*: information known at compile time

Long-lived data (e.g., data in the storage allocated by `malloc`)

Short-lived *activation records* (*stack frame*) generated during procedure calls

Static vs. Dynamic Storage Allocation

- *Static*: the storage-allocation decision can be made by the compiler by looking only at the program text
  - Global constants, global variables
- *Dynamic*: a decision can be made only while the program is running
  - ***Stack storage:*** The space for names local to a procedure is allocated on a stack. The lifetime of the data is the same as that of the called procedure
  - ***Heap storage:*** Hold data that may outlive the call to the procedure that created it
    - Manual memory deallocation (手工回收内存, e.g., by calling the function `free`)
    - Automatic memory deallocation (a.k.a., garbage collection, 垃圾回收)

Outline

- Storage Organization
- Stack Space Allocation
  - Activation Trees
  - Activation Records
  - Calling Sequences
- Heap Management

Activation of Procedures

- Procedure calls, or *activation* of procedures, nest in time
    - If a procedure P calls Q, the callee Q returns before its caller P
    - This makes it possible to use stack (first in/called, last out/return) to manage the run-time memory used by procedure calls

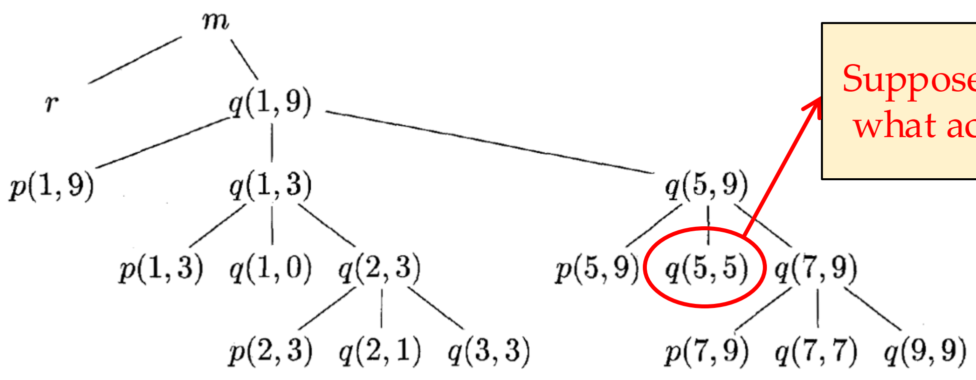
```
int a[11];
void readArray() { /* Reads 9 integers into a[1], ..., a[9]. */
    int i;
    ...
}
int partition(int m, int n) {
    /* Picks a separator value v, and partitions a[m..n] so that
       a[m..p-1] are less than v, a[p] = v, and a[p+1..n] are
       equal to or greater than v. Returns p. */
    ...
}
void quicksort(int m, int n) {
    int i;
    if (n > m) {
        i = partition(m, n);
        quicksort(m, i-1);
        quicksort(i+1, n);
    }
}
main() {
    readArray();
    a[0] = -9999;
    a[10] = 9999;
    quicksort(1,9);
}
```

Possible activations:

```
enter main()
    enter readArray()
    leave readArray()
    enter quicksort(1,9)
        enter partition(1,9)
        leave partition(1,9)
        enter quicksort(1,3)
            ...
        leave quicksort(1,3)
        enter quicksort(5,9)
            ...
        leave quicksort(5,9)
    leave quicksort(1,9)
leave main()
```

Activation Trees (活动树)

- We can represent the activations of procedures during the running of an entire program by a tree, called *activation tree*
  - Each node corresponds to one activation (children nodes are ordered)
  - The root is the activation of the "`main`" procedure
- **Two interesting properties (FILO):**
  - The sequence of procedure calls corresponds to a preorder traversal
  - The sequence of returns corresponds to a postorder traversal



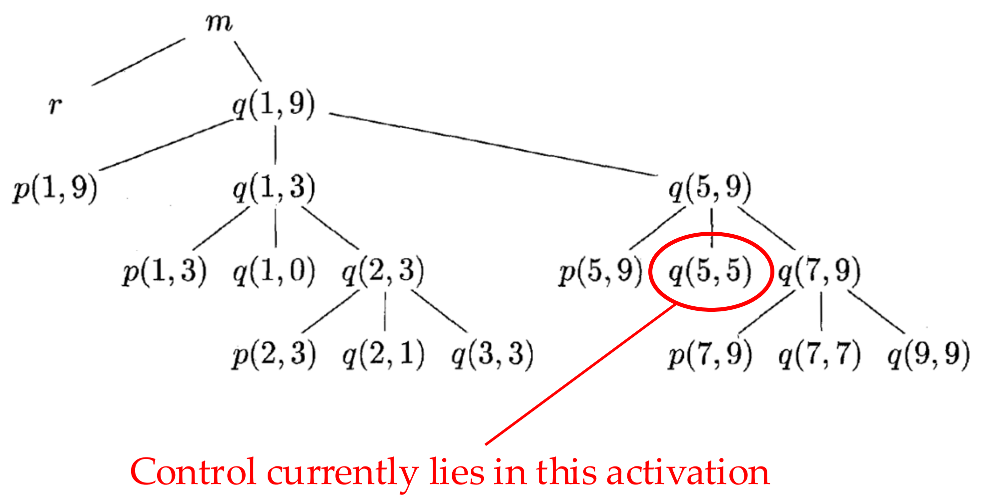
Suppose control lies in this activation, then what activations are currently open (*live*)?

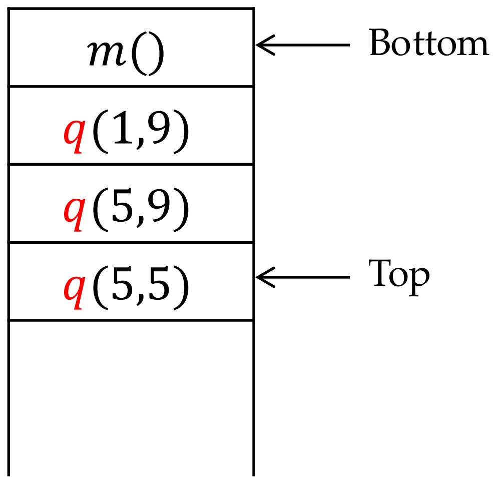
Outline

- Storage Organization
- Stack Space Allocation
  - Activation Trees
  - Activation Records
  - Calling Sequences
- Heap Management

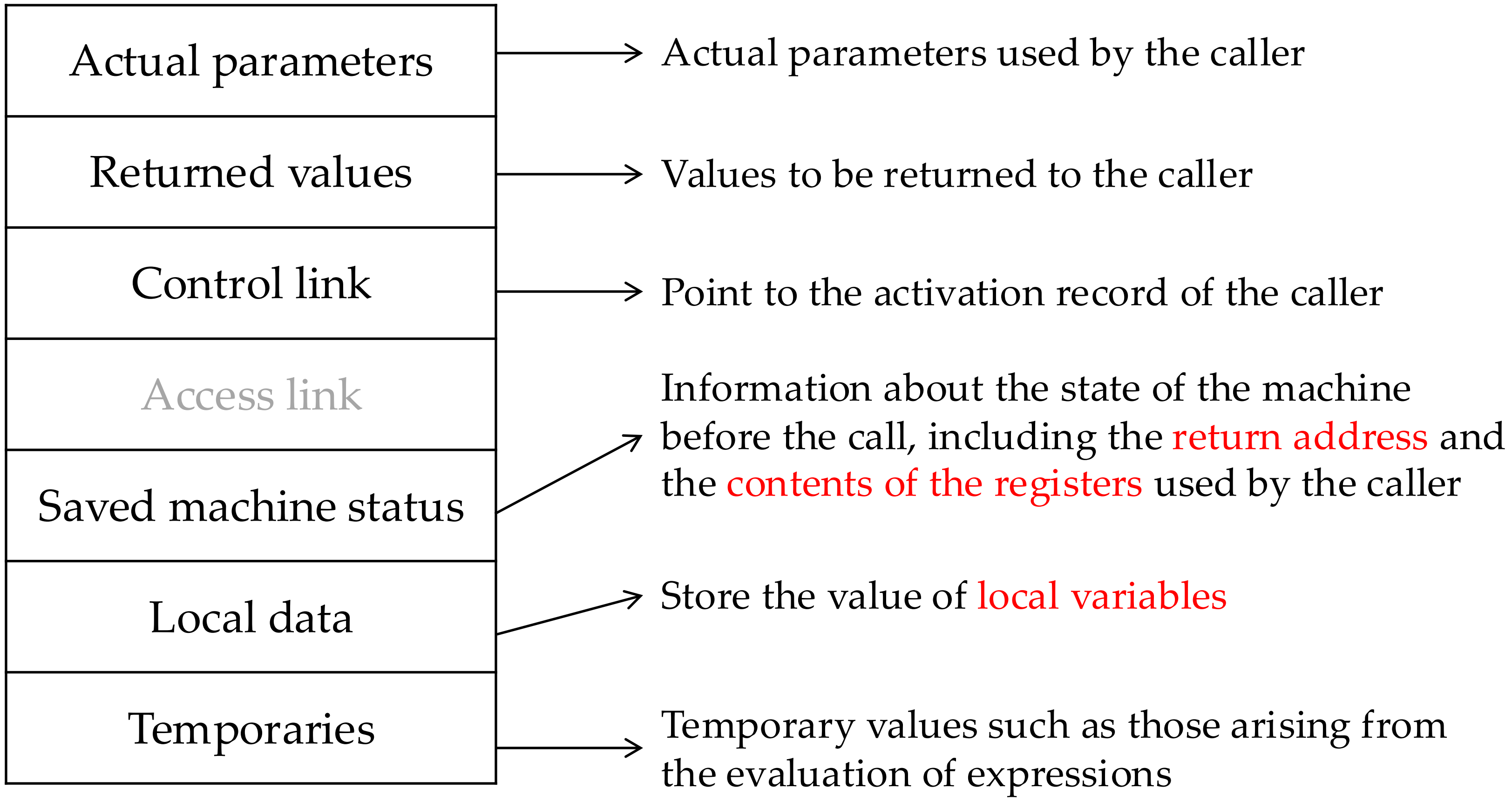
Activation Record (活动记录)

- Procedure calls and returns are usually managed by a run-time stack called the *control stack* (or *call stack*)
- Each live activation has an *activation record* (or *stack frame*) on the control stack





What's in An Activation Record?



Actual parameters used by the caller

Values to be returned to the caller

Point to the activation record of the caller

Information about the state of the machine before the call, including the return address and the contents of the registers used by the caller

Store the value of local variables

Temporary values such as those arising from the evaluation of expressions

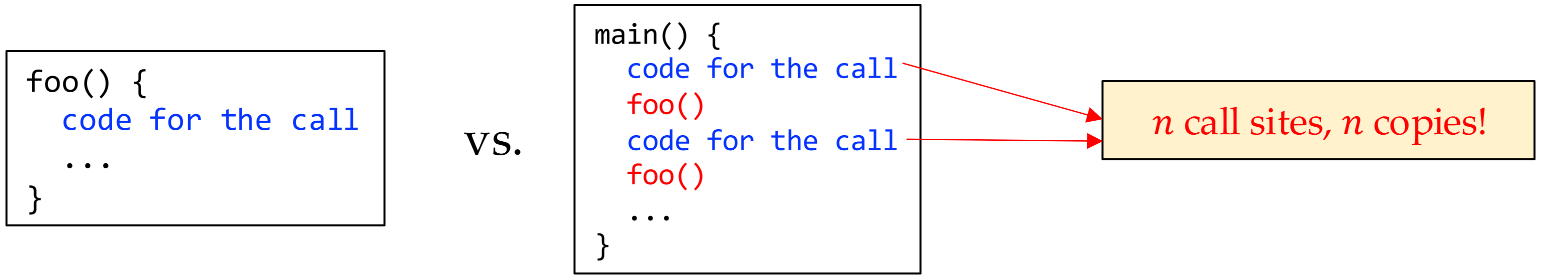
Each activation record may occupy a different amount of memory. The total size of the call stack is often small as compared with heap. For example, in JVM, the default size of the call stack for each thread is 1MB.

Outline

- Storage Organization
- Stack Space Allocation
  - Activation Trees
  - Activation Records
  - Calling Sequences
- Heap Management

Calling Sequences (方法调用代码序列)

- *Calling sequences*, consisting of code that (1) allocates an activation record on the stack and (2) enters information into its fields
- A *return sequence* (返回代码序列) restores the state of the machine so that the caller can continue its execution after the call
- The code in a calling sequence is divided between the caller and the callee
  - There is no exact division; the source language, target machine, and the OS may impose requirements that affect the solution
  - A general rule is to put as much code as possible into the callee



```
foo() {
  code for the call
  ...
}
```

vs.

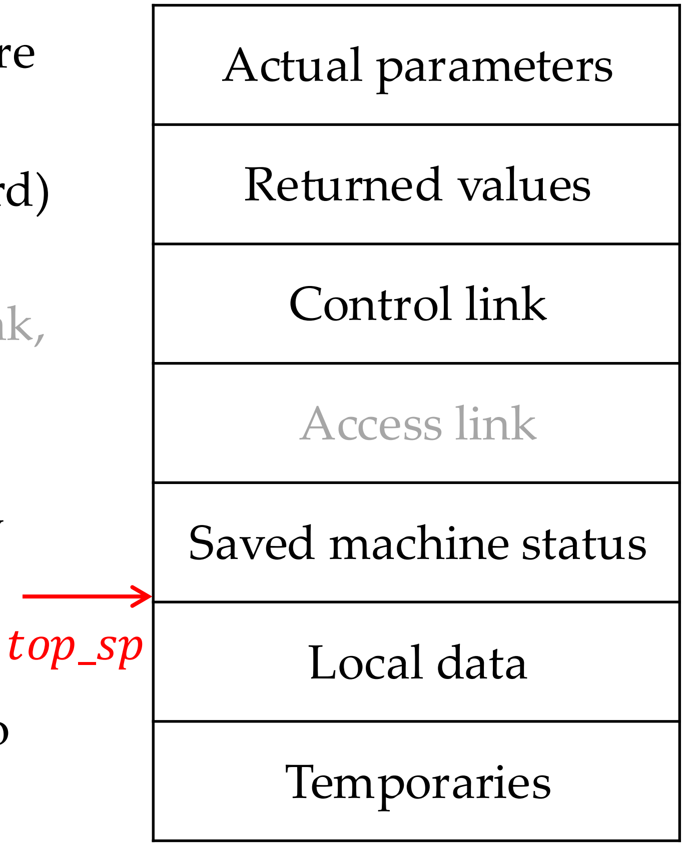
```
main() {
  code for the call
  foo()
  code for the call
  foo()
  ...
}
```

Calling/Return Sequences' Job

- From the data perspective
  - Correctly pass arguments to the callee
  - Correctly pass the return values to the caller
- From the control perspective
  - Correctly transfer the control to the first instruction of the callee
  - Correctly transfer the control back to the caller so that it can continue with the instruction immediately after the procedure-call statement

Layout of Activation Records
General Principles

- Values passed between caller and callee are put at the beginning of the callee's activation record (next to the caller's record)
- Fixed-length items (control link, access link, saved machine status) are in the middle
- Items whose size may not be known early enough are placed at the end
- The top-of-stack (*top_sp*) pointer points to the end of the fixed-length fields



Steps of A Calling Sequence

1. The caller evaluates the actual parameters

2. The caller stores the return address and the old value of *top_sp* into the callee's activation record

3. The caller increments *top_sp* accordingly

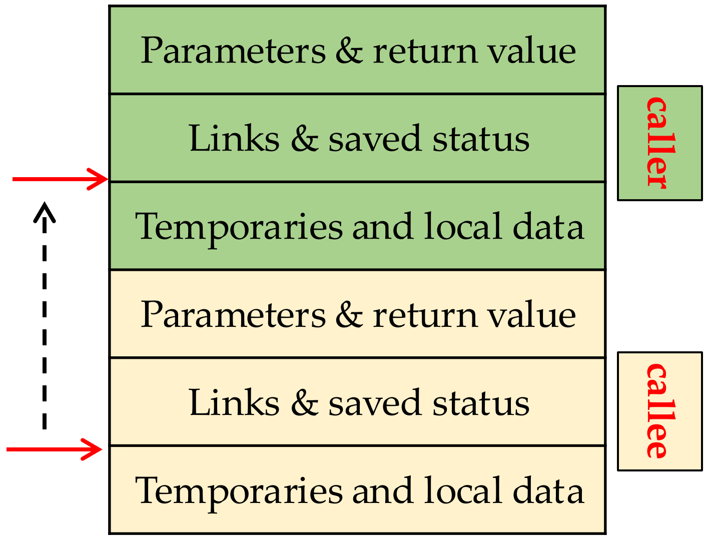
4. The callee saves the register values and other status info

5. The callee initializes its local data and begins execution

Steps of A Return Sequence

- The callee places the return value next to the actual parameters fields in its activation record
- Using information in the machine-status field, the callee restores *top_sp** and other registers
- Go to the return address set by the caller

* Although *top_sp* has been decremented, the caller knows where to find the return value and use it (recall that caller places the actual parameters)


Parameters & return value
Links & saved status
Temporaries and local data
Parameters & return value
Links & saved status
Temporaries and local data
caller
callee

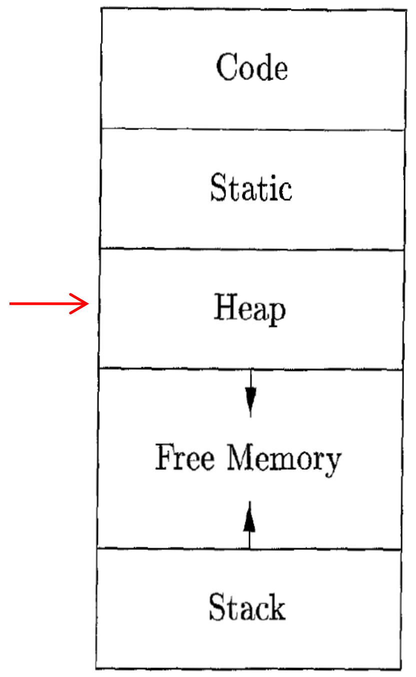
Outline

- Storage Organization
- Stack Space Allocation
- Heap Management

$\rightarrow$

- The Memory Manager
- Leveraging Program Locality
- Reducing Fragmentation

Heap (堆)



- The *heap* is used for data that lives indefinitely, or until the program explicitly deletes it

- Many languages enable us to create objects/data whose existence is not tied to the procedure activation that creates them
  - In Java, the objects created with `new` can exist long after the procedure that created them is gone
  - In C, the memory space allocated with `malloc` is accessible until the `free` operation is applied

The Memory Manager

- Memory manager allocates and deallocates space within the heap
  - It serves as an interface between application programs and the OS

- The mechanism depends on languages
  - For C/C++, developers need to deallocate memory manually (using `free` or `delete`). The memory manager is responsible for implementing deallocation
  - For Java, garbage collector (a subsystem of memory manager) finds spaces within the heap that are no longer needed and reallocates such spaces to hold other data items

Basic Functions

- **Allocation (分配内存):** Provide contiguous heap memory when a program requests memory for a variable or object
  - If possible, it satisfies the request using free space in the heap
  - Otherwise, it seeks to increase the heap storage space by getting consecutive bytes of virtual memory from the OS

- **Deallocation (回收内存):** Return deallocated space to the pool of free space for reuse
  - Typically, memory managers do not return memory to the OS, even if the program's heap usage drops

Desirable Properties

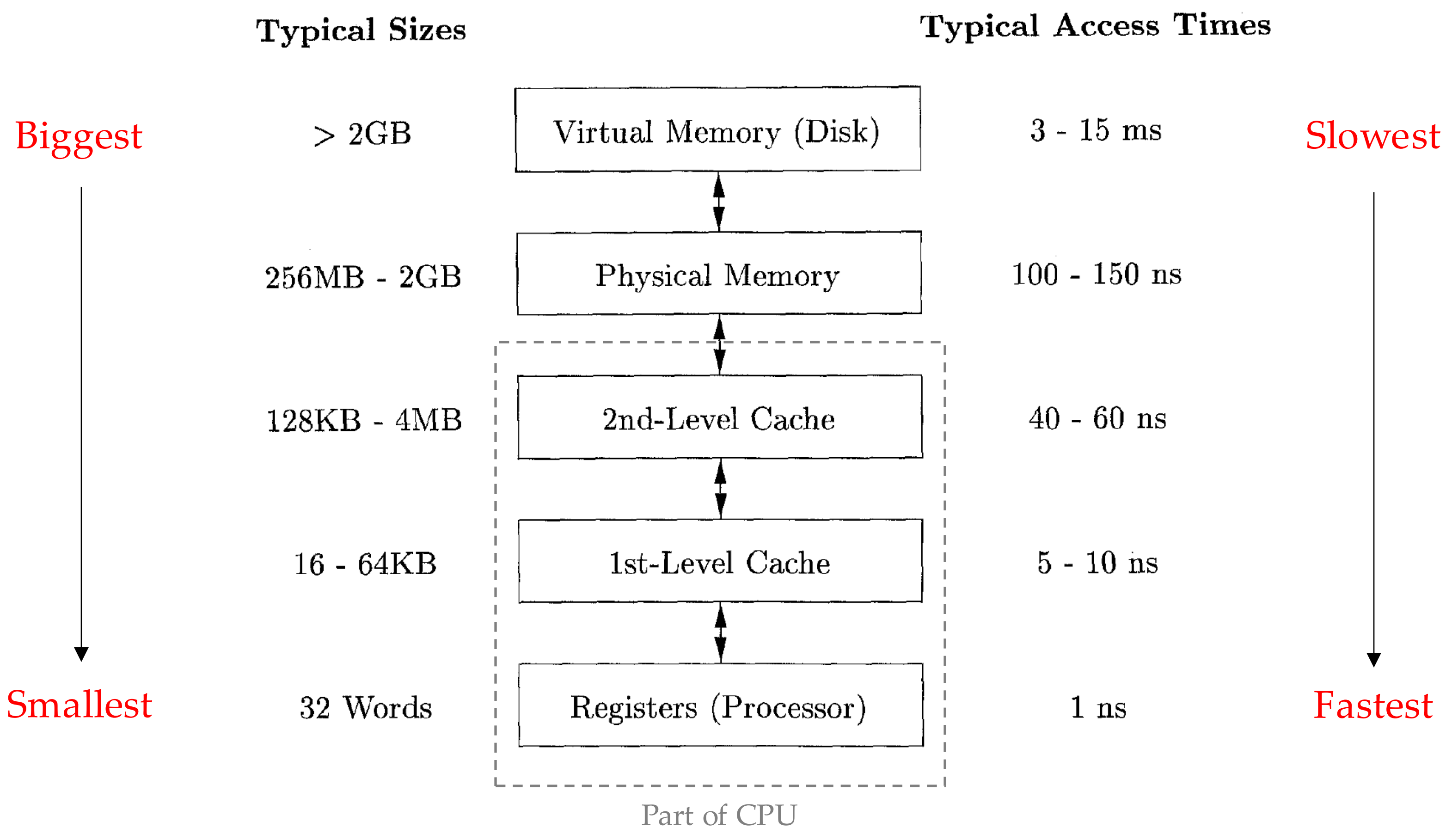
- **Space efficiency (空间效率):** Minimize the total heap space needed by a program, i.e., minimizing "fragmentation" (碎片化)
- **Program efficiency (程序效率):** Make better use of the memory hierarchy to allow programs to run faster*
- **Low overhead (低开销):** Minimize the execution time spent on memory allocation/deallocation

* The time taken to execute an instruction can vary widely depending on where objects are placed in memory

Outline

- Storage Organization
- Stack Space Allocation
- Heap Management
  - The Memory Manager
  - Leveraging Program Locality
  - Reducing Fragmentation

Memory Hierarchy



Program Locality (程序局部性)

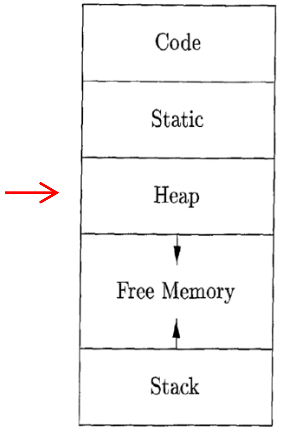
- Most programs exhibit a high degree of locality:
    - **Temporal locality (时间局部性):** the memory locations accessed are likely to be accessed again within a short period of time
    - **Spatial locality (空间局部性):** memory locations close to the locations accessed are likely to be accessed within a short period of time
- Programs spend 90% of their time executing 10% of the code (80/20 or Pareto's Principle in programming)

Locality allows us to take advantage of the memory hierarchy to lower the average memory-access time: place the most common instructions and data in the fast-but-small storage, while leaving the rest in the slow-but-large storage

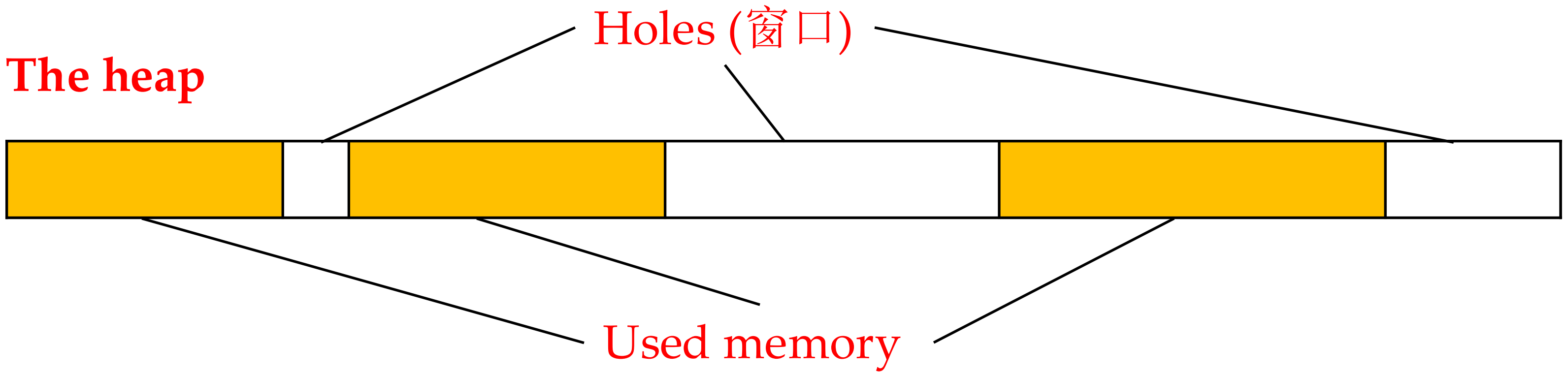
Outline

- Storage Organization
- Stack Space Allocation
- Heap Management
  - The Memory Manager
  - Leveraging Program Locality
  - Reducing Fragmentation

The Fragmentation Problem (1)



- At the beginning of program execution, the heap is one contiguous unit of free space
- As the program allocates/deallocates memory, the heap is broken up into free and used chunks

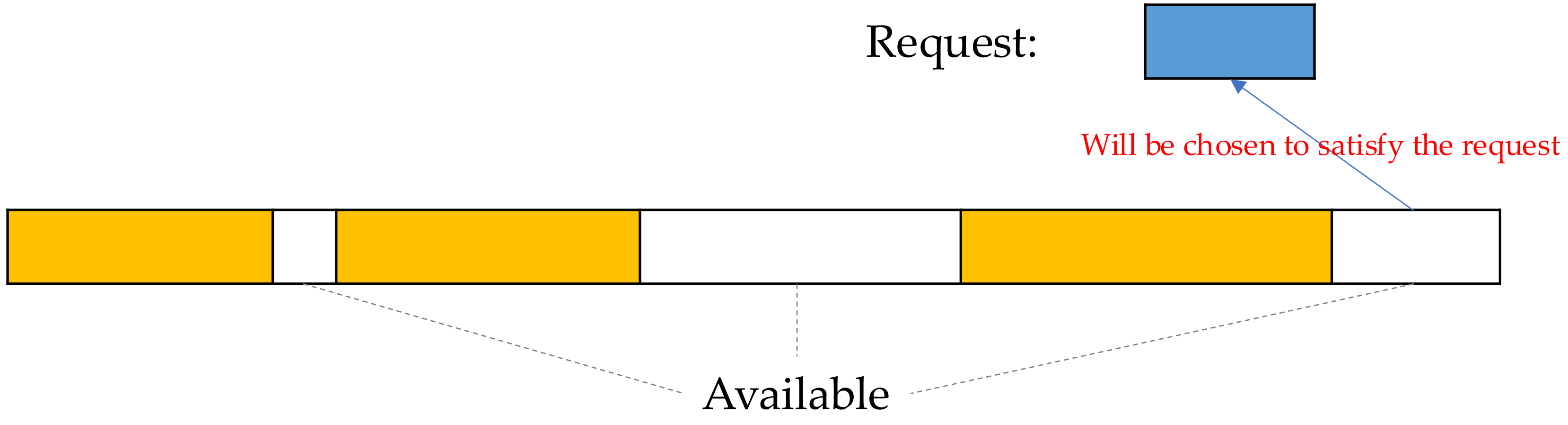


The Fragmentation Problem (2)

- With each allocation request, the memory manager must place the requested memory into a large-enough hole
    - Typically, it needs to split some hole, creating a yet smaller hole
- With each deallocation request, the freed memory are added back to the pool of free space
    - Should combine contiguous holes into larger ones. Otherwise, the holes can only get smaller
- Without a good strategy, the free memory may get *fragmented*, consisting of a large number of small, noncontiguous holes

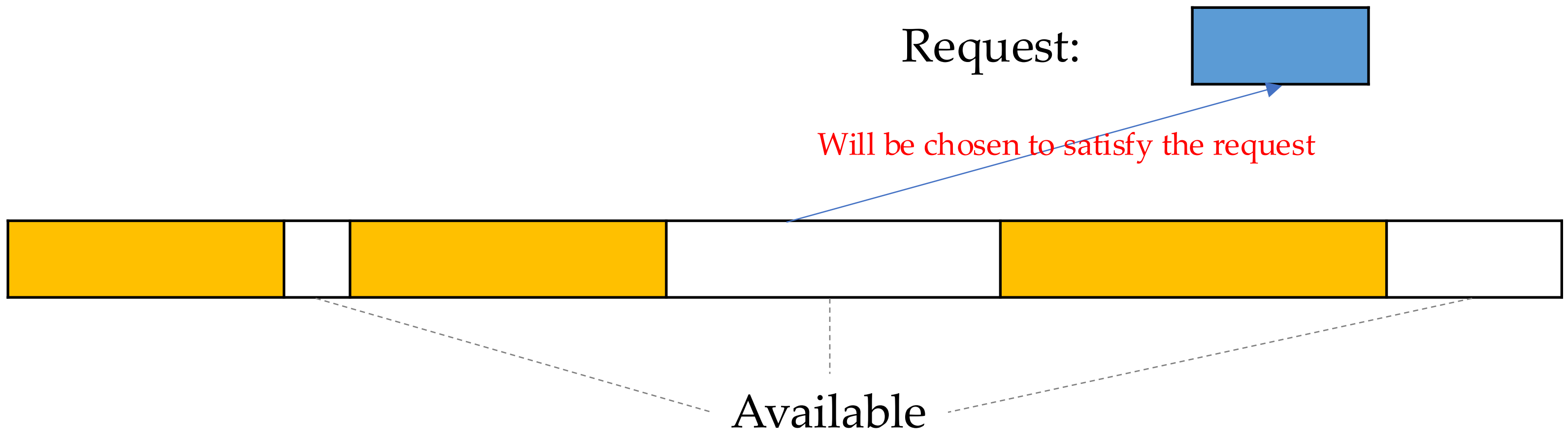
Object Placement Strategies

- **Best-fit algorithm**
  - Allocate the requested memory in the smallest available hole that is large enough
  - This algorithm spares the large holes to satisfy subsequent, larger requests, which tends to improve space utilization



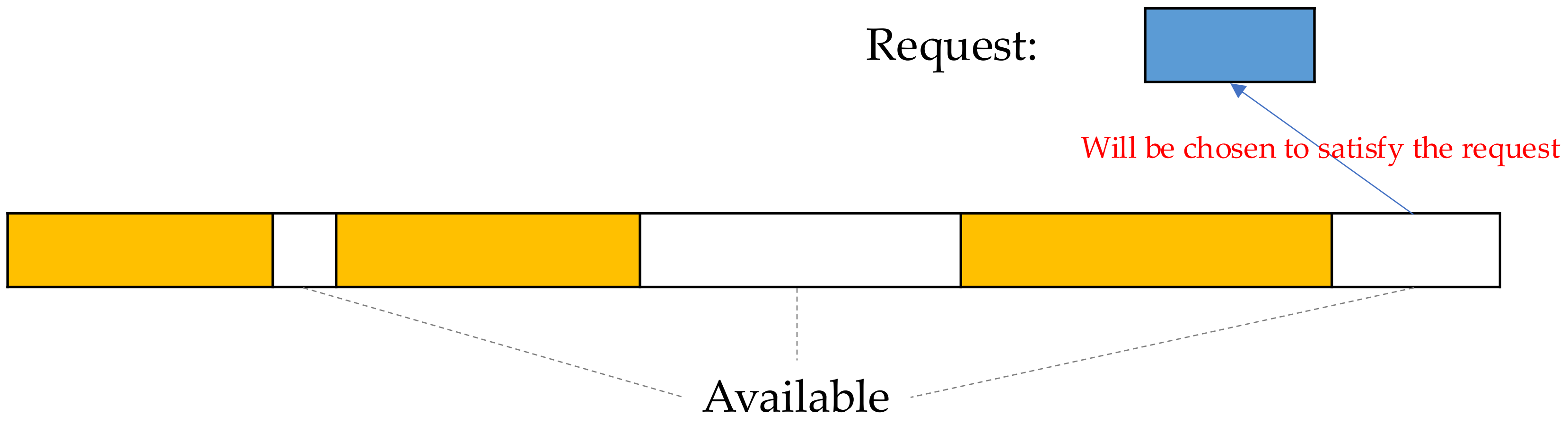
Object Placement Strategies

- **First-fit algorithm**
  - An object is placed in the first (lowest-address) hole in which it fits
  - It takes less time to place objects and improves spatial locality, but is inferior to best-fit in overall performance



The Binning Strategy for Best-Fit

- **Motivation:** The naïve best fit algorithm is not efficient enough, may need to traverse and check all available memory chunks

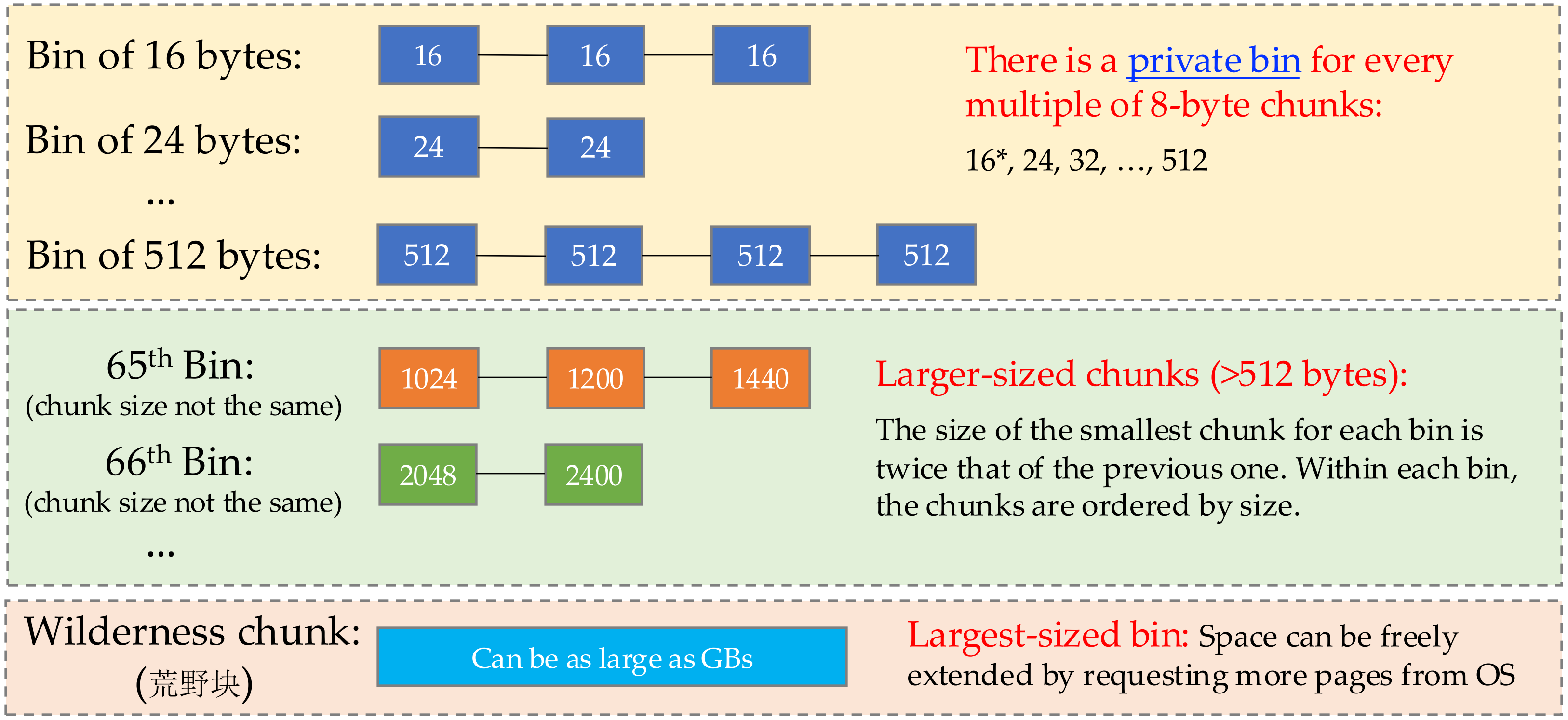


- **Basic idea of the binning strategy:**
  - Organize free chunks into separate *bins* (容器), according to chunk sizes
  - Have more bins for smaller-sized chunks (objects are often small)

Learn more about Doug Lea's memory allocator: https://cw.fel.cvut.cz/old/_media/courses/a4m33pal/04_dynamic_memory_v6.pdf, https://gee.cs.oswego.edu/dl/html/malloc.html

The Doug Lea's Strategy

- The Lea memory manager of the GNU C compiler `gcc` aligns all chunks to 8-byte boundaries (i.e., chunk size is always a multiple of eight)



There is a private bin for every multiple of 8-byte chunks:

16*, 24, 32, …, 512

Larger-sized chunks (>512 bytes):

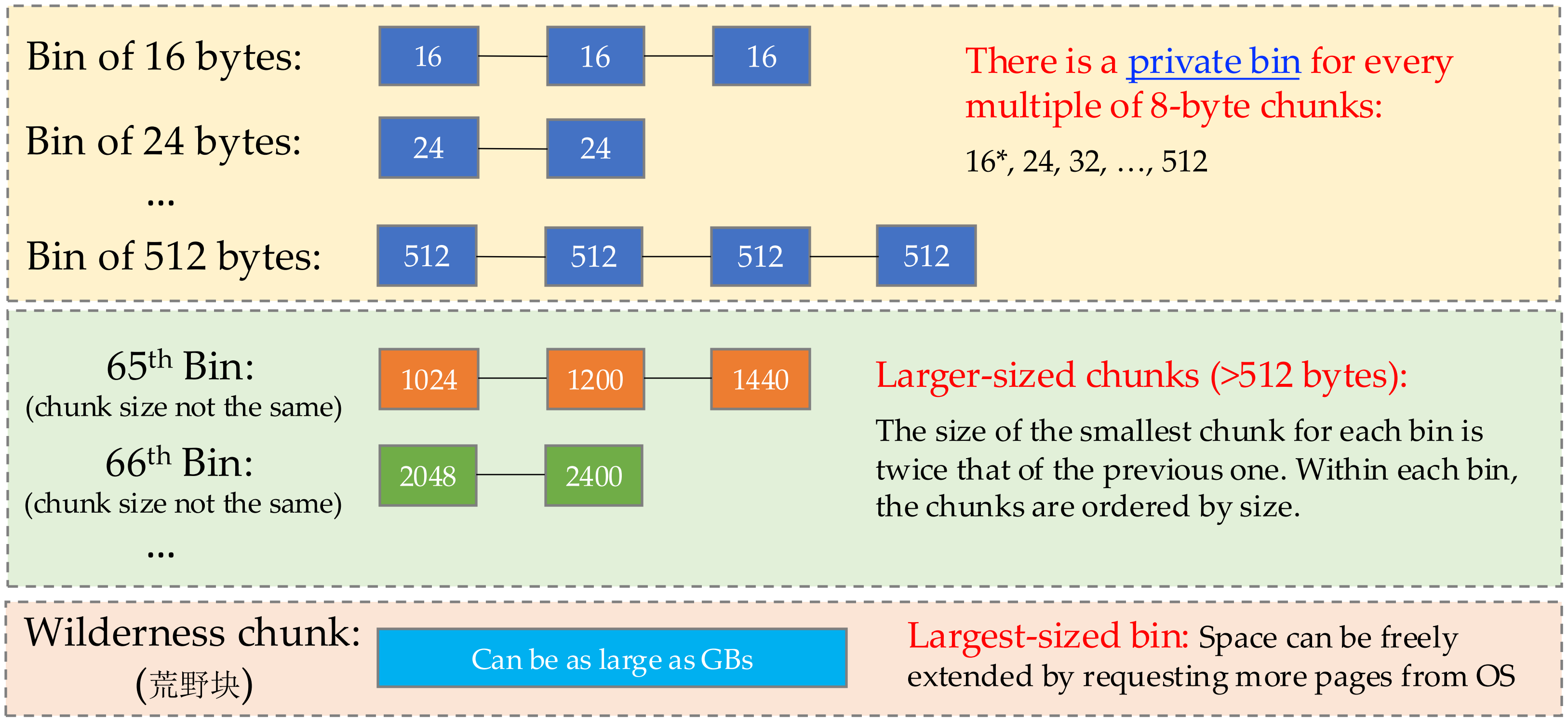
The size of the smallest chunk for each bin is twice that of the previous one. Within each bin, the chunks are ordered by size.

Largest-sized bin: Space can be freely extended by requesting more pages from OS

* Not starting from 8 because of malloc bookkeeping overhead

Binning Makes It Easy to Find the Best-Fit Chunk

- For small sizes requests (e.g., 48 bytes), if there is a bin for chunks of that size, we may take any chunk from the bin (the chunks are of the same size)



There is a private bin for every multiple of 8-byte chunks:

16*, 24, 32, …, 512

Larger-sized chunks (>512 bytes):

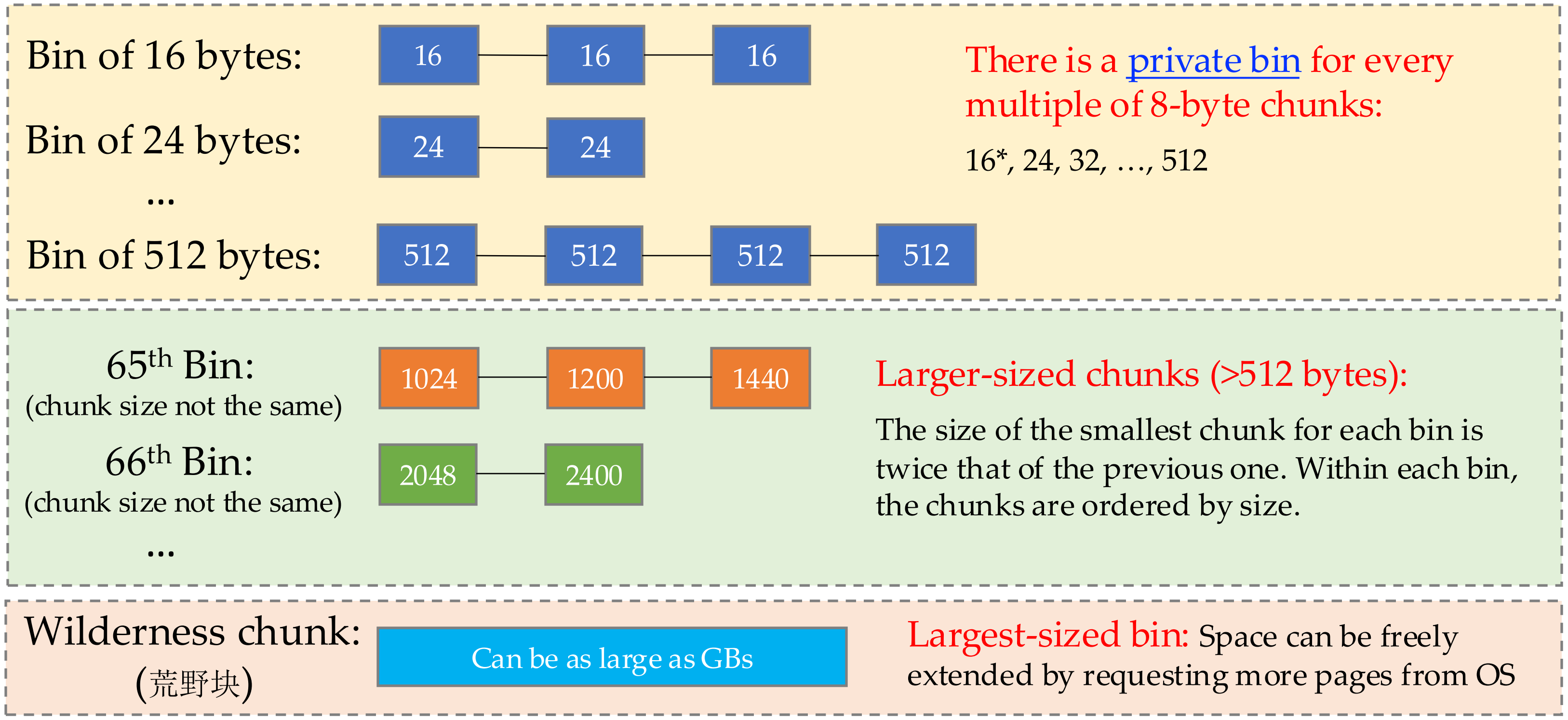
The size of the smallest chunk for each bin is twice that of the previous one. Within each bin, the chunks are ordered by size.

Largest-sized bin: Space can be freely extended by requesting more pages from OS

* Not starting from 8 because of malloc bookkeeping overhead

Binning Makes It Easy to Find the Best-Fit Chunk

- For sizes that do not have a private bin (e.g., 1224 bytes), we find the bin that is allowed to include chunks of the size (within the bin, where the chunks may have different size, we can use either a first-fit or a best-fit algorithm*)



There is a private bin for every multiple of 8-byte chunks:

16*, 24, 32, …, 512

Larger-sized chunks (>512 bytes):

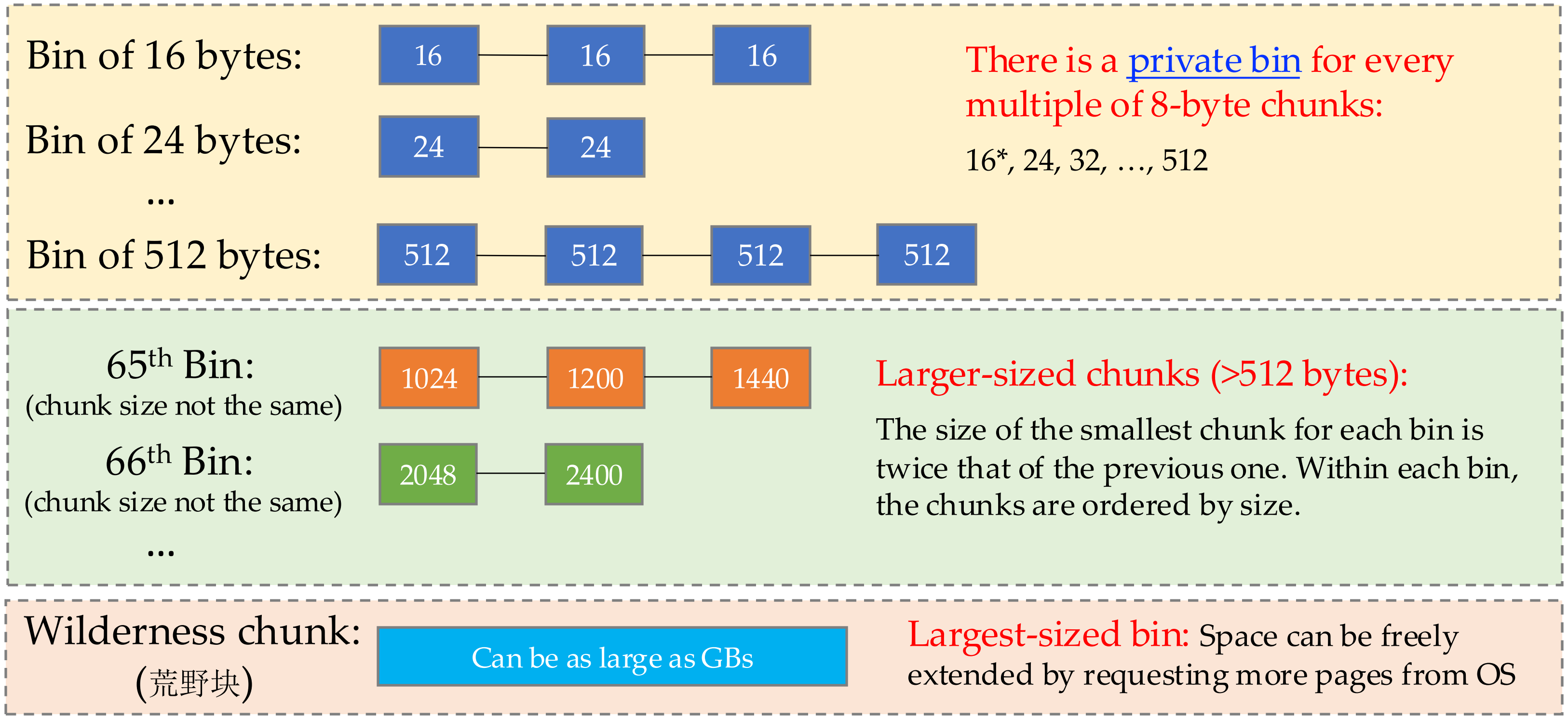
The size of the smallest chunk for each bin is twice that of the previous one. Within each bin, the chunks are ordered by size.

Largest-sized bin: Space can be freely extended by requesting more pages from OS

* When the fit is not exact, the remainder of the chunk will genreally need to be placed in a bin with smaller sizes

Binning Makes It Easy to Find the Best-Fit Chunk

- When the above searches fail, we continue to check the bin for the next larger size(s). Eventually, we may reach wilderness chunk.



Bin of 16 bytes:
16
16
16
Bin of 24 bytes:
24
24
...
Bin of 512 bytes:
512
512
512
512
There is a private bin for every multiple of 8-byte chunks:
16*, 24, 32, …, 512
65th Bin:
(chunk size not the same)
1024
1200
1440
66th Bin:
(chunk size not the same)
2048
2400
...
Larger-sized chunks (>512 bytes):
The size of the smallest chunk for each bin is twice that of the previous one. Within each bin, the chunks are ordered by size.
Wilderness chunk:
(荒野块)
Can be as large as GBs
Largest-sized bin: Space can be freely extended by requesting more pages from OS

Reading Tasks

- Chapter 7 of the dragon book
  - 7.1 Storage Organization
  - 7.2 Stack Allocation of Space (7.2.1 – 7.2.3)
  - 7.4 Heap Management (7.4.1 – 7.4.4)