ECE 391 Final Review

You’re almost done, hang in there!

Reminders

- Final is Tuesday Dec 12, 7-10 PM
- 3 sheets of notes
- Topics Covered
  - Signals
  - I-Mail
  - Memory Allocation
  - Scheduling
  - Memory Maps
  - MP3
- Final is not cumulative

Signals

What are Signals?

- User “Interrupts”

- Signals can be generated by user processes or kernel
  - Signals can be handled either by kernel or user processes
  - Some signals can be masked (Non maskable signals exist and you should know them)
- Signals vs Interrupts:
  - When do we check for signals – returning from:
    - Interrupt, Exception, systemcall
  - Who manages them: Signals -> Kernel, Interrupts -> Processor (IDT)
  - Who handles them: Signals -> Kernel/User, Interrupts -> Kernel

Important Signals

- SIGINT (Ctrl-C)
- SIGSTP (Ctrl-Z)
- SIGCONT
- SIGKILL
- SIGSTOP

- Which two signals cannot have their default behavior changed?

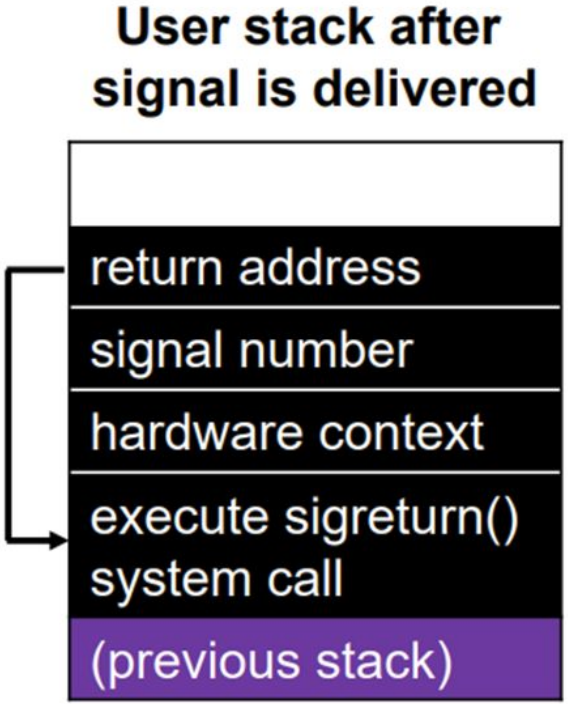
Default Actions

- Terminate
  - Ends the program completely
- Dump Core
  - Terminates and dumps core
- Stop
  - Stops a program, program can be resumed later
- Ignore
  - Signal is discarded
- Continue
  - Continues a currently stopped process

How does this sigreturn work?

- Default handler goes through kernel
- If handler is assigned, kernel will “help” the user with returning to kernel
- Building the user return argument
    - Must call another system call to get back to sigreturn
    - Kernel will set this up, rather than having user handler do the syscall
- Swapping hardware context on return
    - Want: Get to signal code, Problem: sigreturn system call context not correct to get there
    - Solution: Store hardware context from first signal call interrupt, restore in sigreturn

Sigreturn User Stack



Things to Know

- Signal User stack
  - Possible coding question here
- Who creates signals?
- Signal control flow (Signal generator -> Kernel -> handler (user or kernel) -> sigreturn -> initial signal generator)
- Non-Maskable signals
- Signal related functions

I-Mail

What is a device driver?

- Kernel interacts with I/O devices through a device driver
- Why?
  - Hides implementation details of how device works
  - Allows dynamic loading/unloading of device drivers
  - Creates a standardize API for interacting with the device

Examples

- Block devices:
    - Data is accessible in blocks of a fixed size (Few kB)
    - Data transfers are buffered and cached
- Character devices:
    - Data is access at a byte level

Overview

- What is Blocking?
  - Waits until information is received to return
- Functions that can block
  - Wait for a new message
    - Read
    - Poll
  - Write
    - Doesn't have to wait for device
      - Unique to I-mail

Overview (pt. 2)

- Sleeping
  - Step 1: Release All locks
  - Step 2: Ensure Conditions without locks are in secured state
  - Step 3: Go to Sleep
  - Step 4: Wake up
    - Reacquire locks
    - Check validity

Overview (pt. 3)

- Important Data Structures
  - Wait Queues
    - Efficient way to wait for previous tasks to finish
    - Formed via doubly linked list
    - Reduces CPU cycles
    - Could Create Race conditions
  - I-mail Read
    - Check for readable information & available semaphores
    - Read information as necessary
    - Release used sempahores

Dynamic Memory Allocation

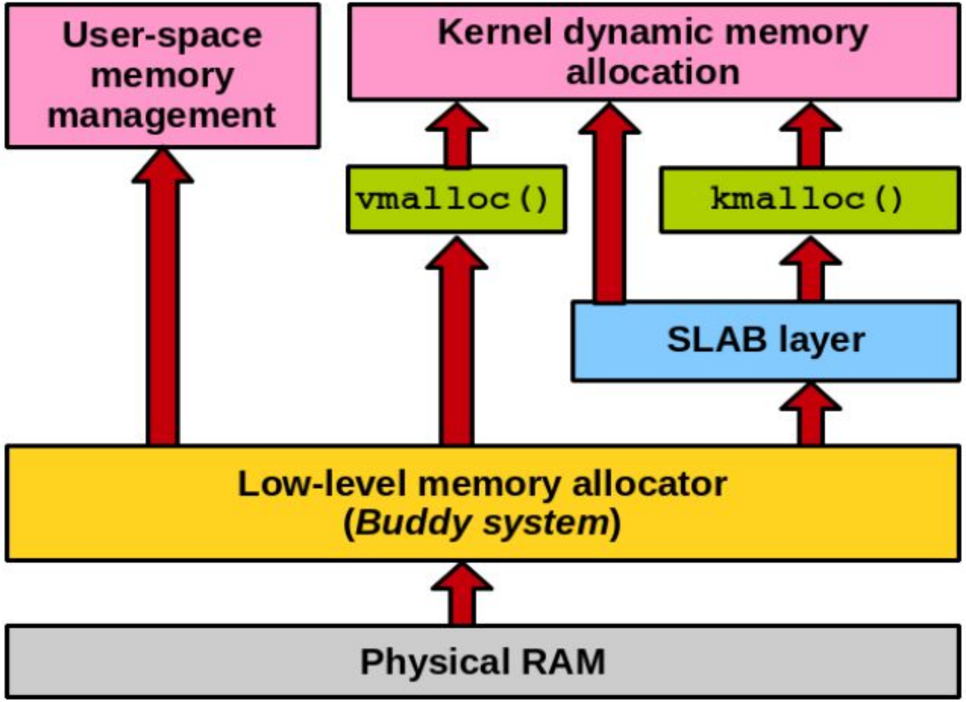
- User creates data in driver
- Deleted by:
    - I-mail
    - Delete user function
- Deleting user could create function if user is in the process of using I-mail

Synchronization

- Functions need to be synchronized appropriately
- Think about which functions interact with each other
- Think about order that operations will be conducted in
  - Use semaphores appropriately

Memory Allocation

Overview of allocation in Linux



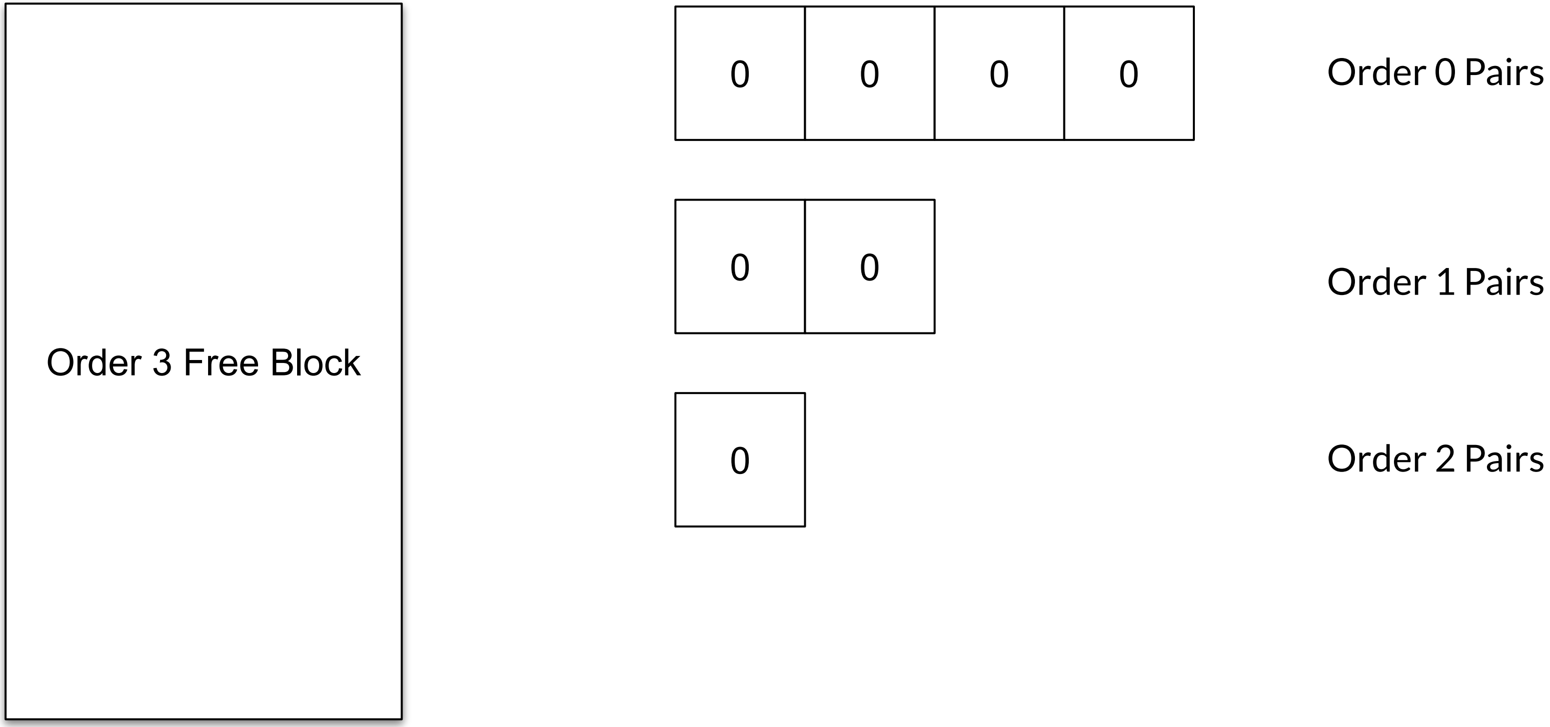
Overview of allocation in Linux

- Physically contiguous:
  - Buddy allocator.
  - Slab allocator.
  - kmalloc()
- Non-physically contiguous (but virtually contiguous): vmalloc().

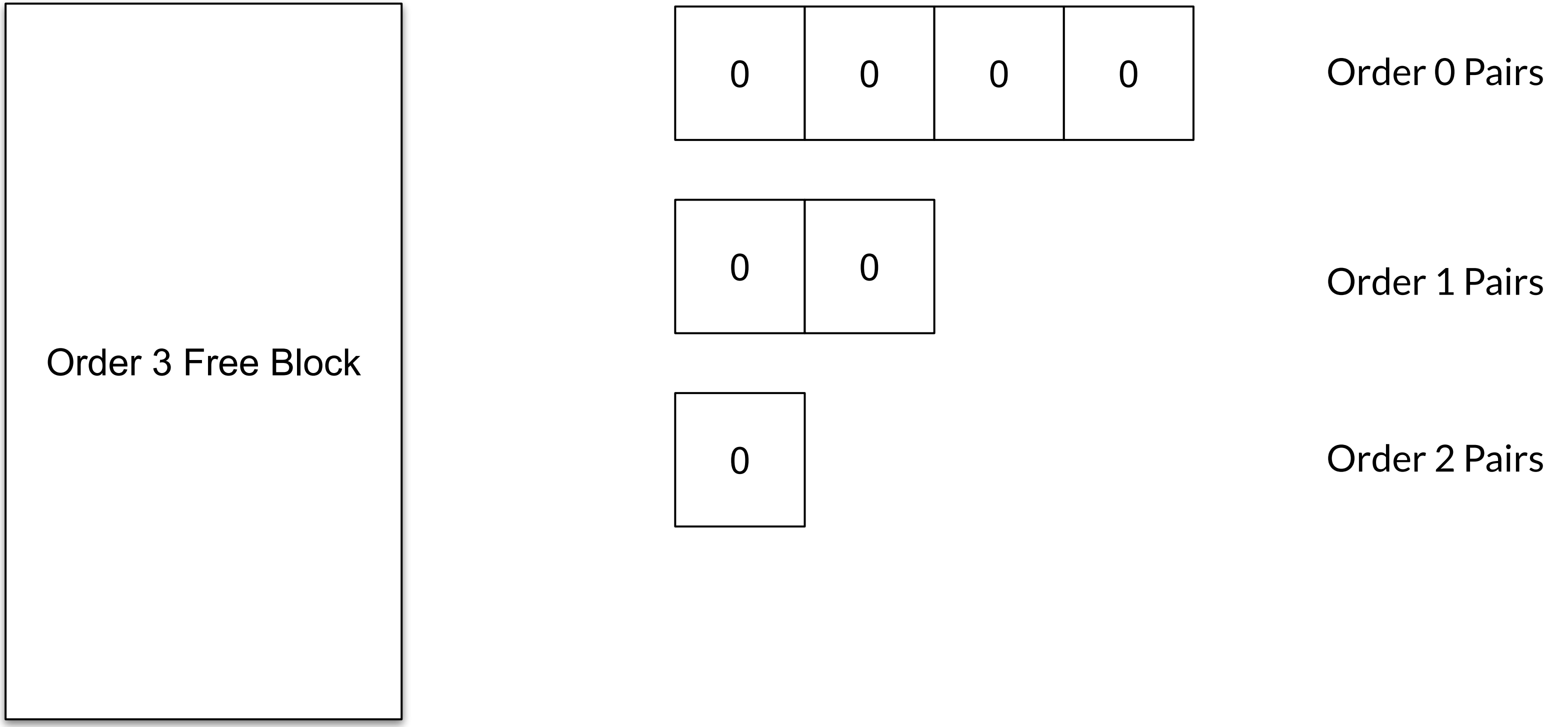
Buddy allocator interface

- No need to memorize function names, just know that buddy allocator serves up requested number of pages:
- unsigned long **__get_free_page** (unsigned int gfp_mask)
  - Allocate a single page and return a virtual address
- unsigned long **__get_free_pages** (unsigned int gfp_mask, unsigned int order)
  - Allocate 2^{order} number of pages and return a virtual address

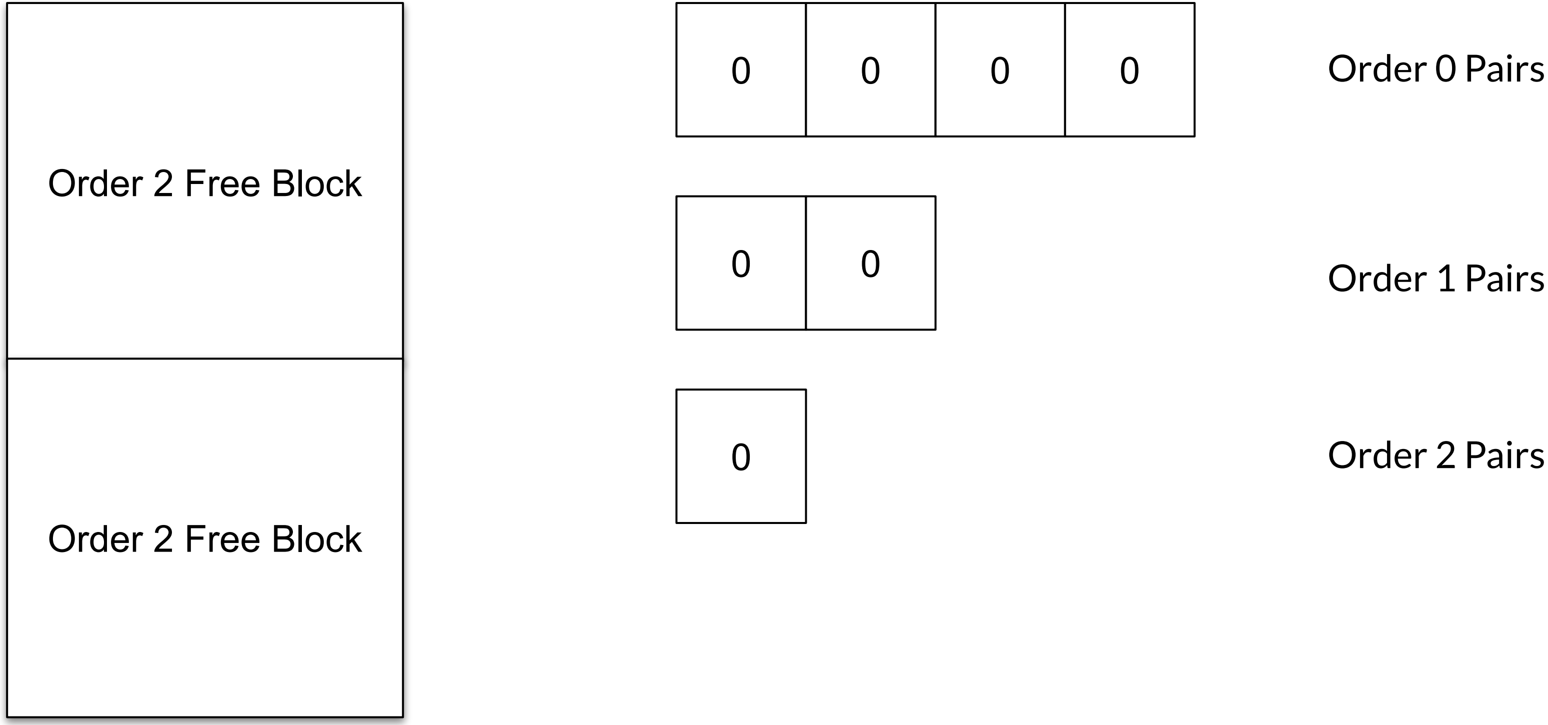
Buddy allocator



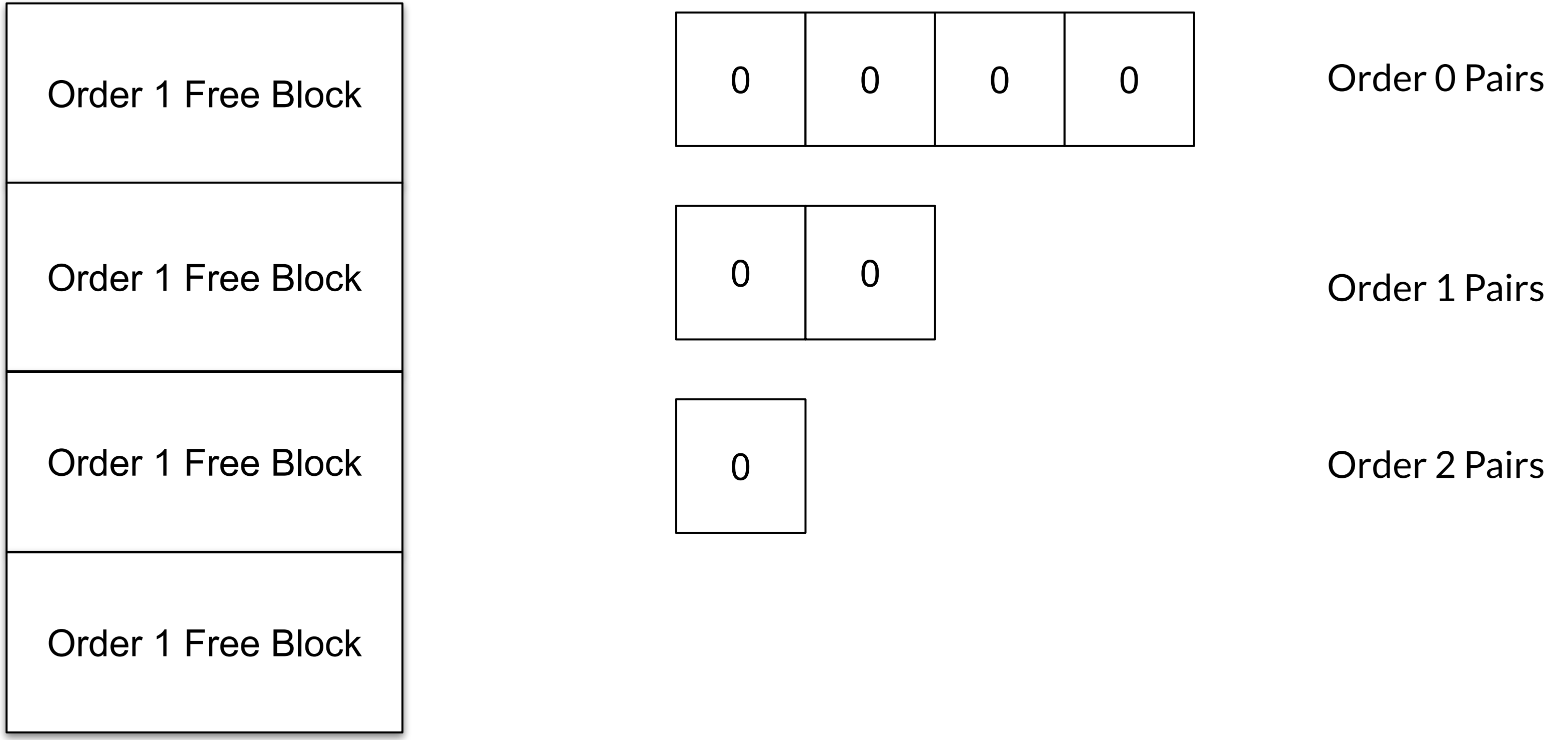
Allocate Order 0 Block



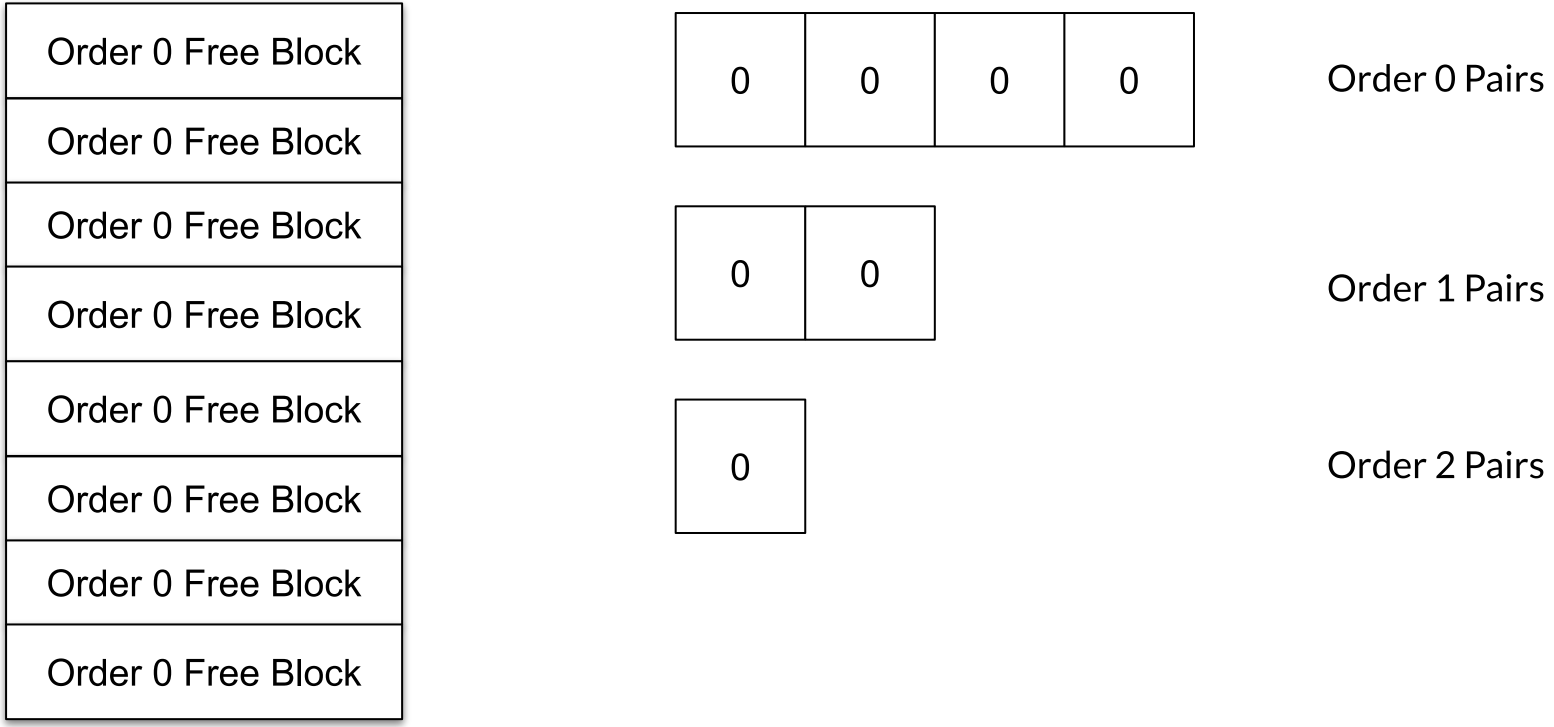
Allocate Order 0 Block



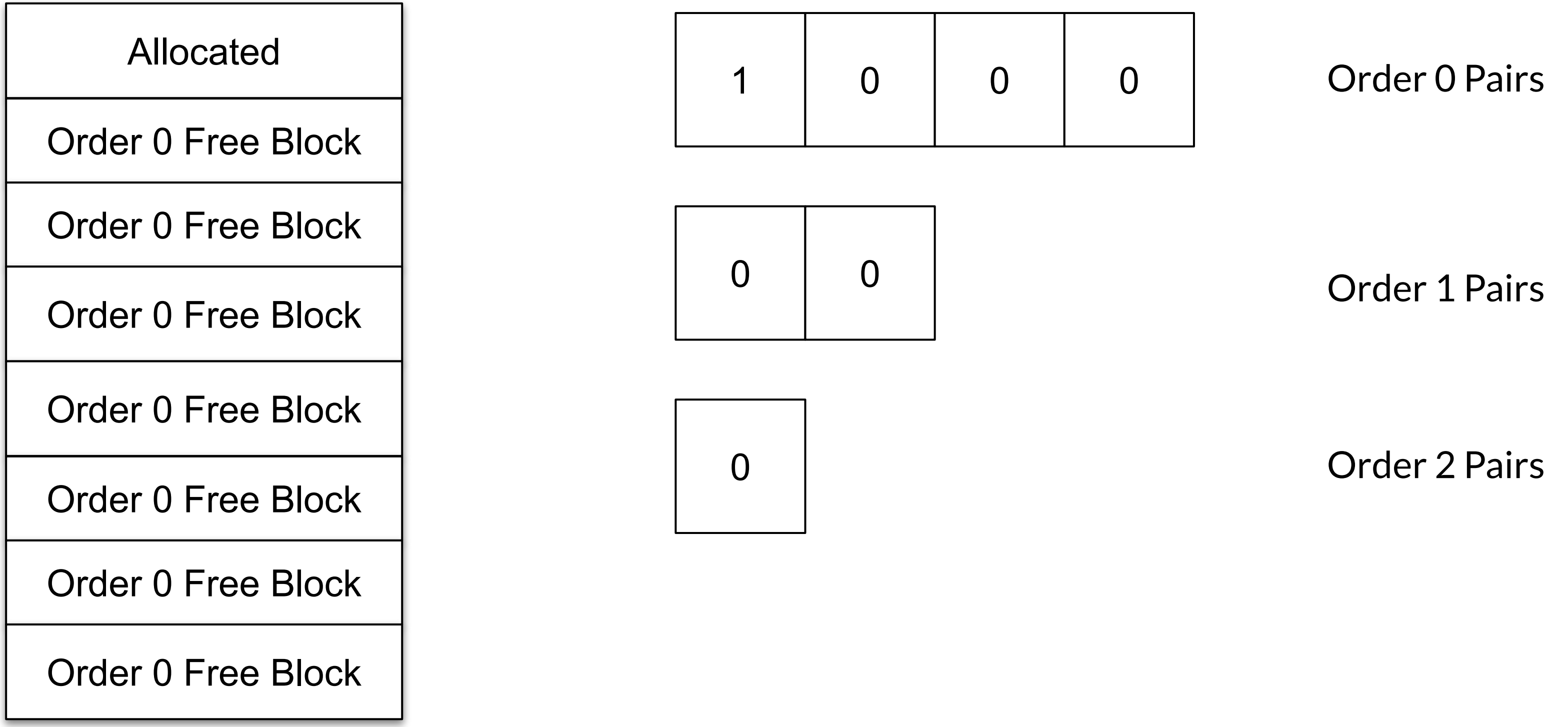
Allocate Order 0 Block



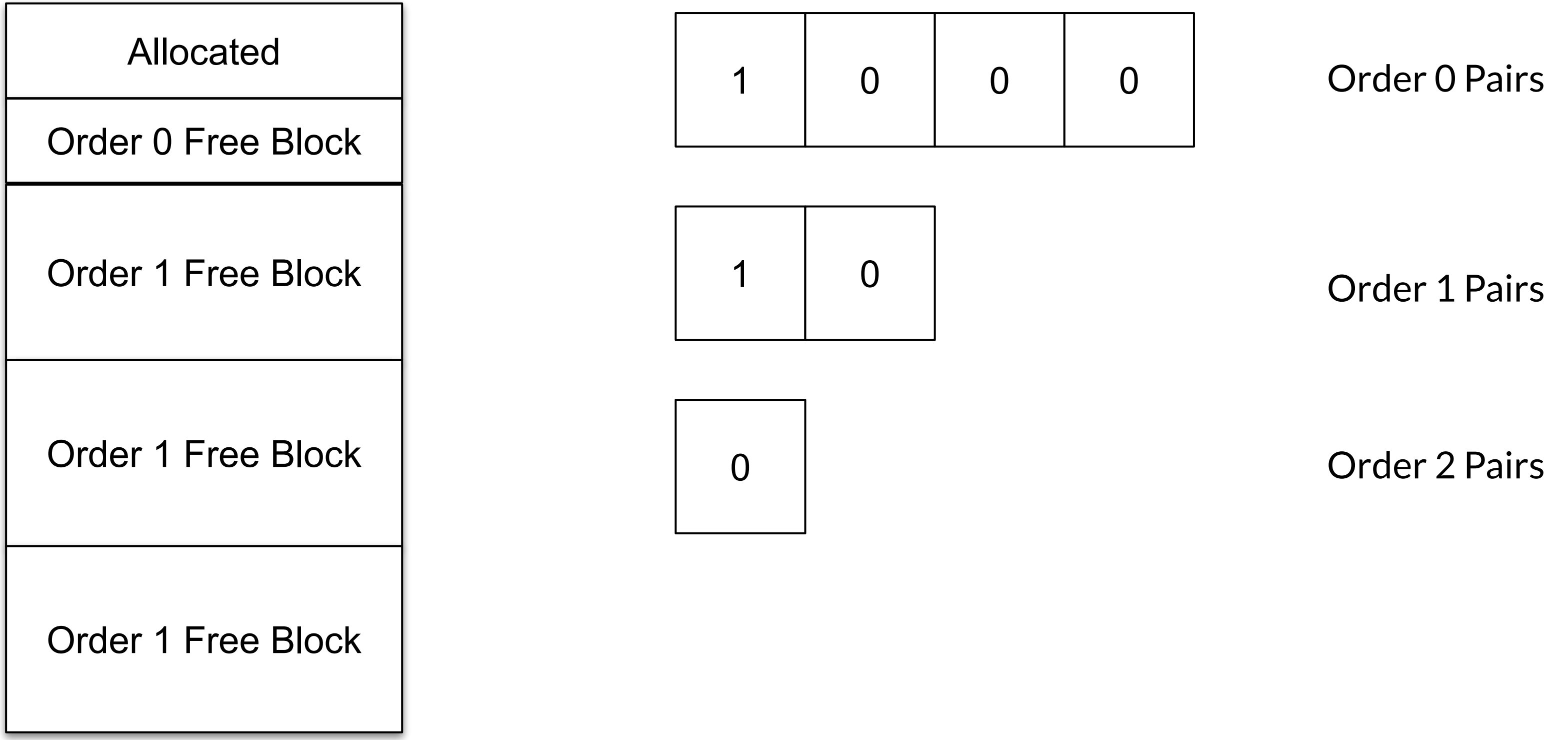
Allocate Order 0 Block



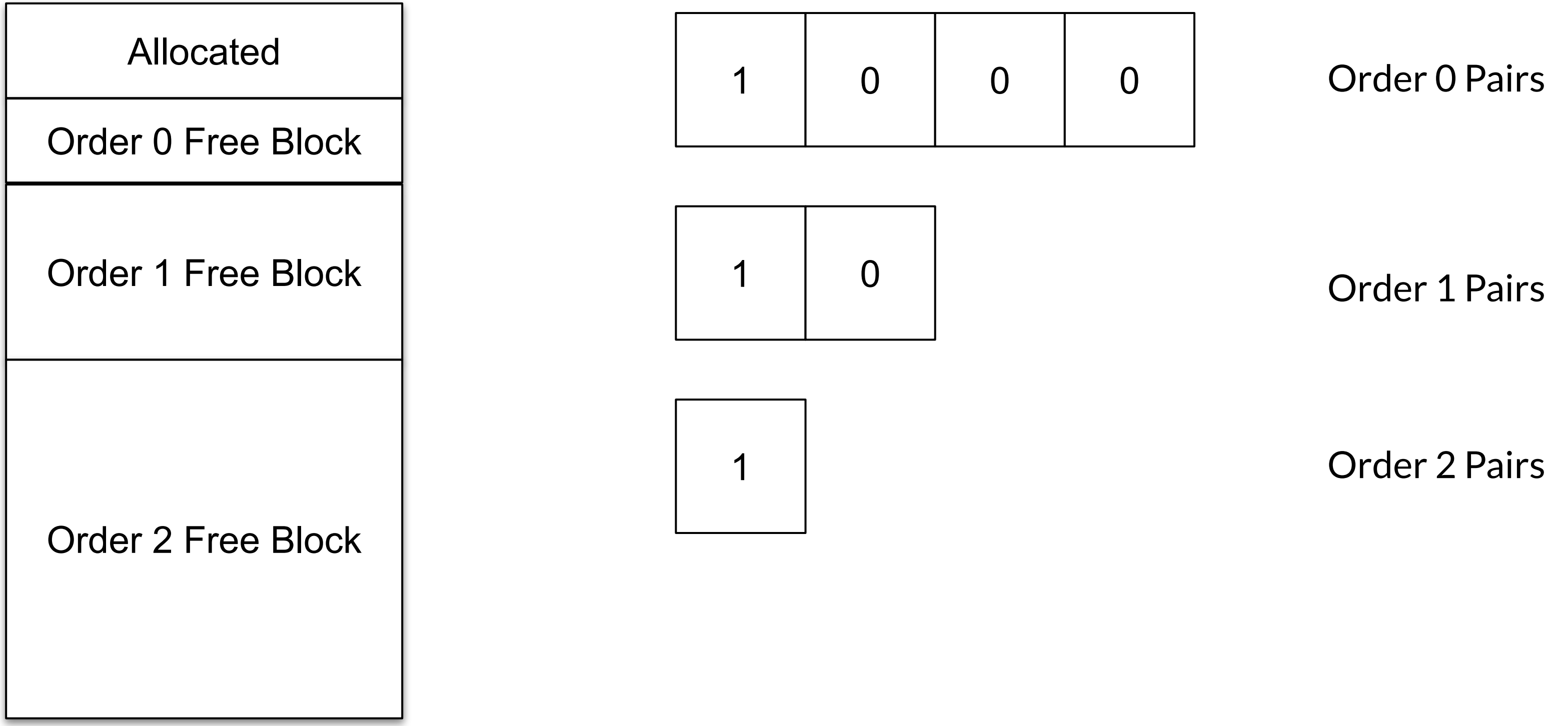
Allocate Order 0 Block



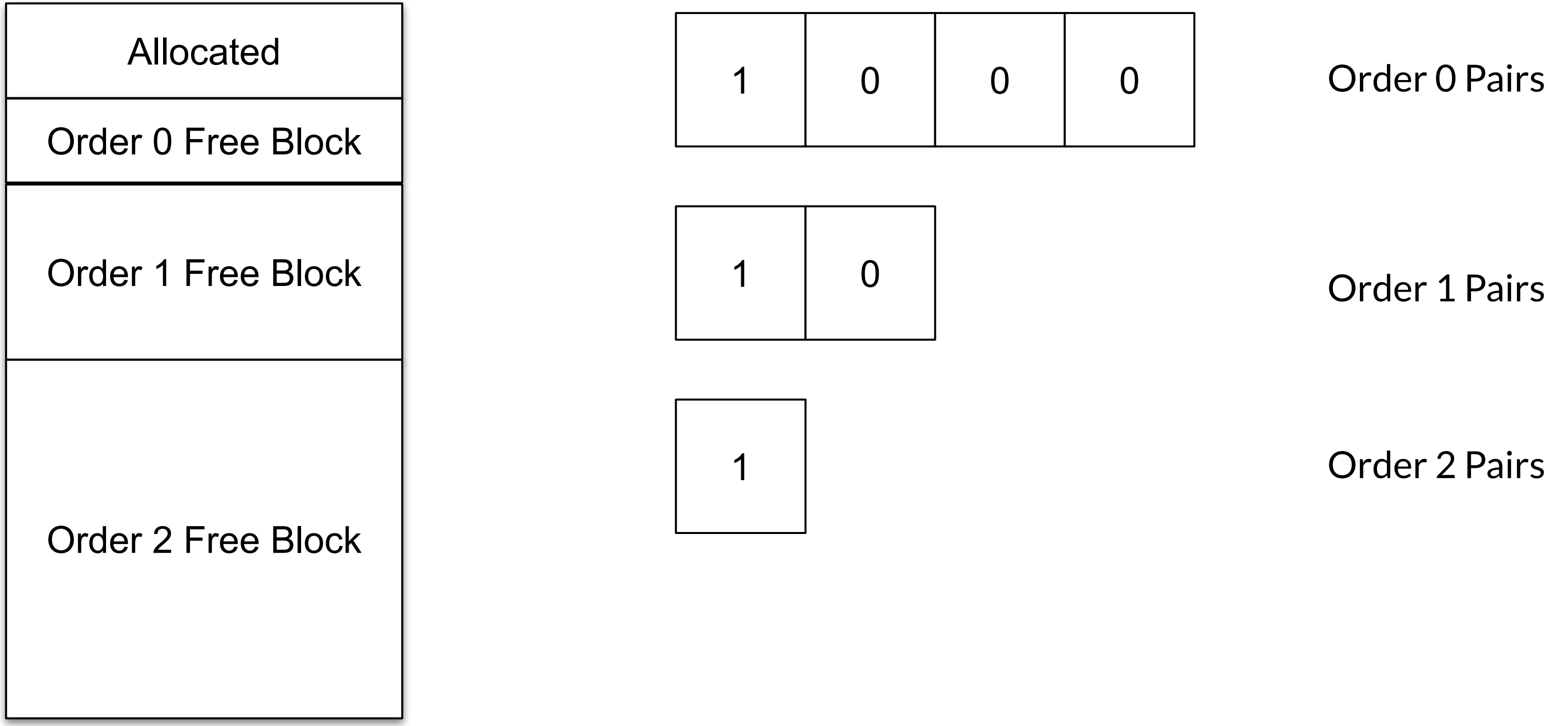
Allocate Order 0 Block



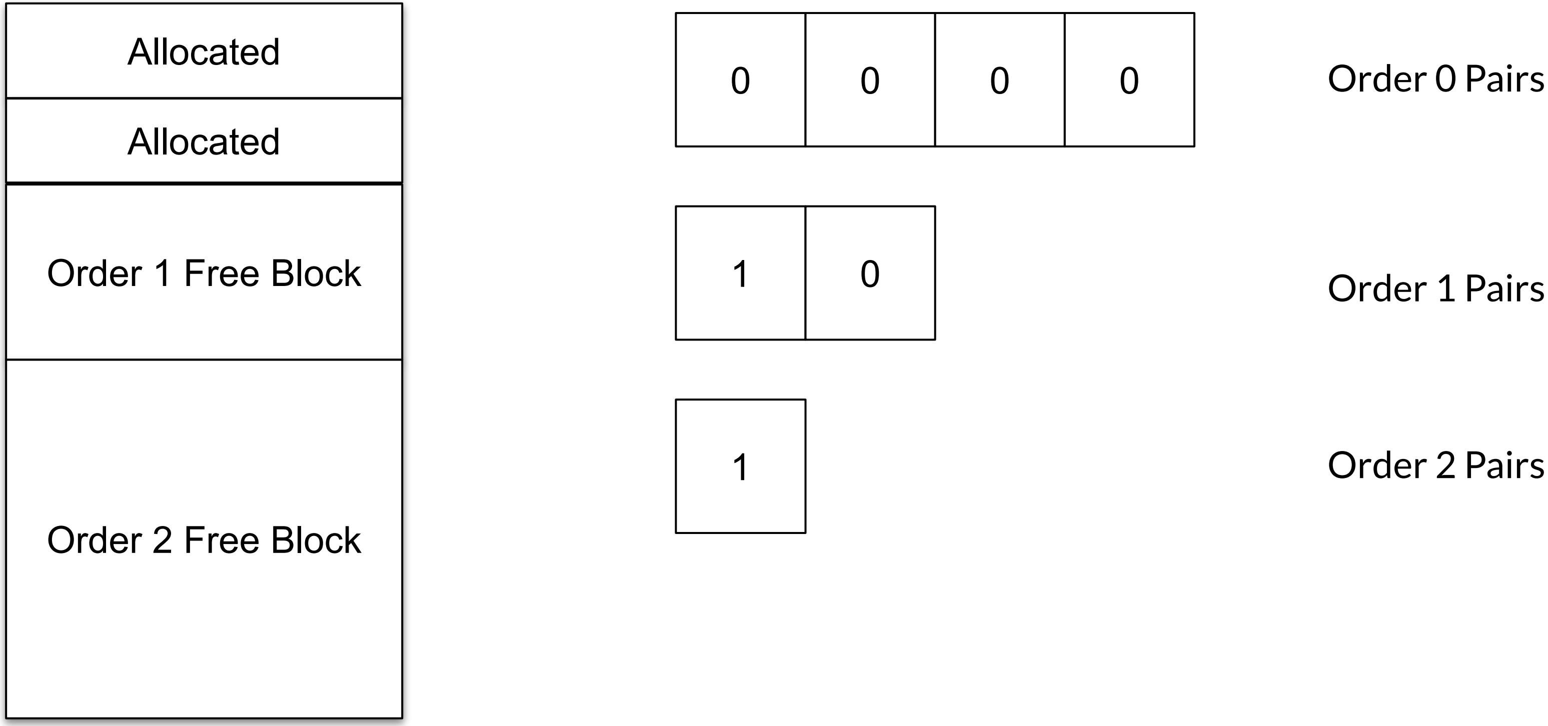
Allocate Order 0 Block



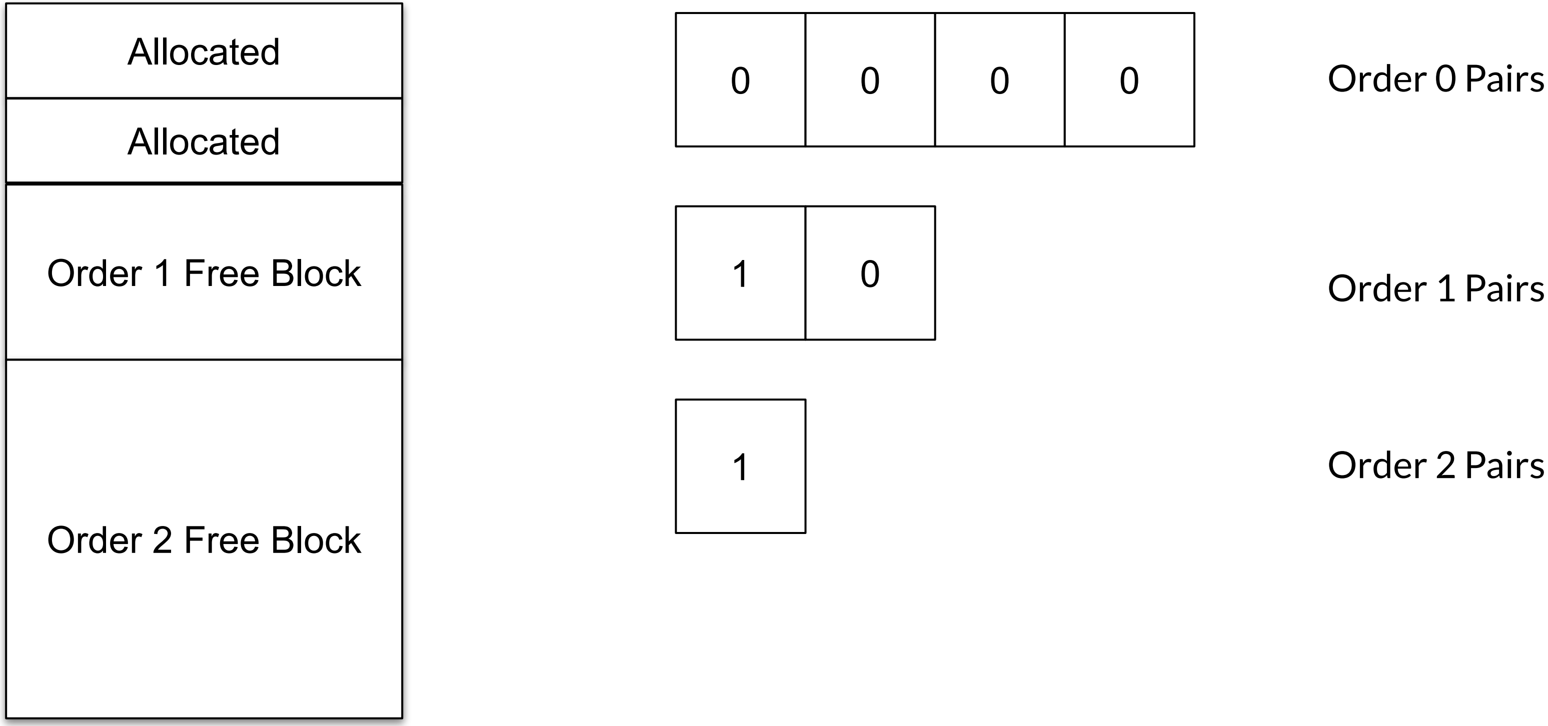
Allocate Another Order 0 Block



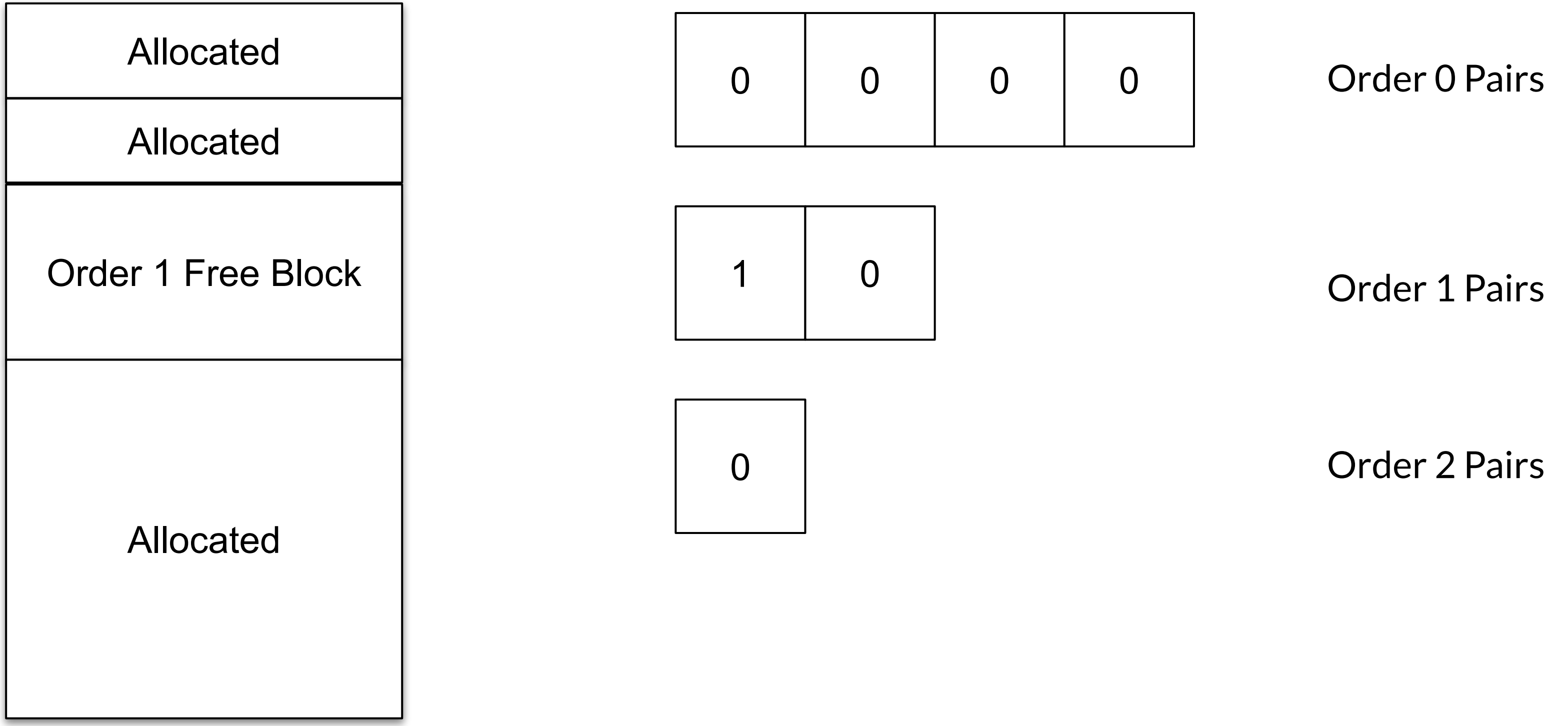
Allocate Another Order 0 Block



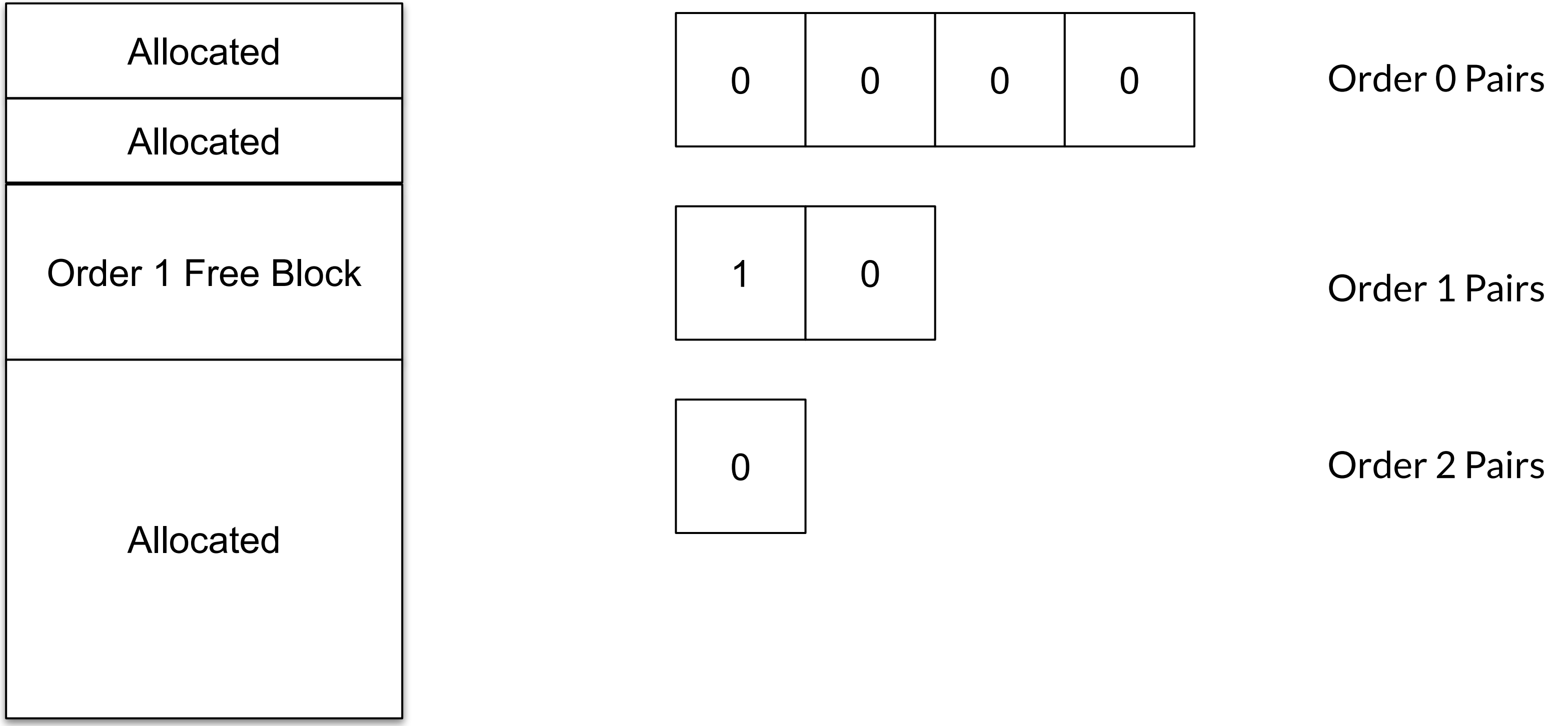
Allocate Order 2 Block



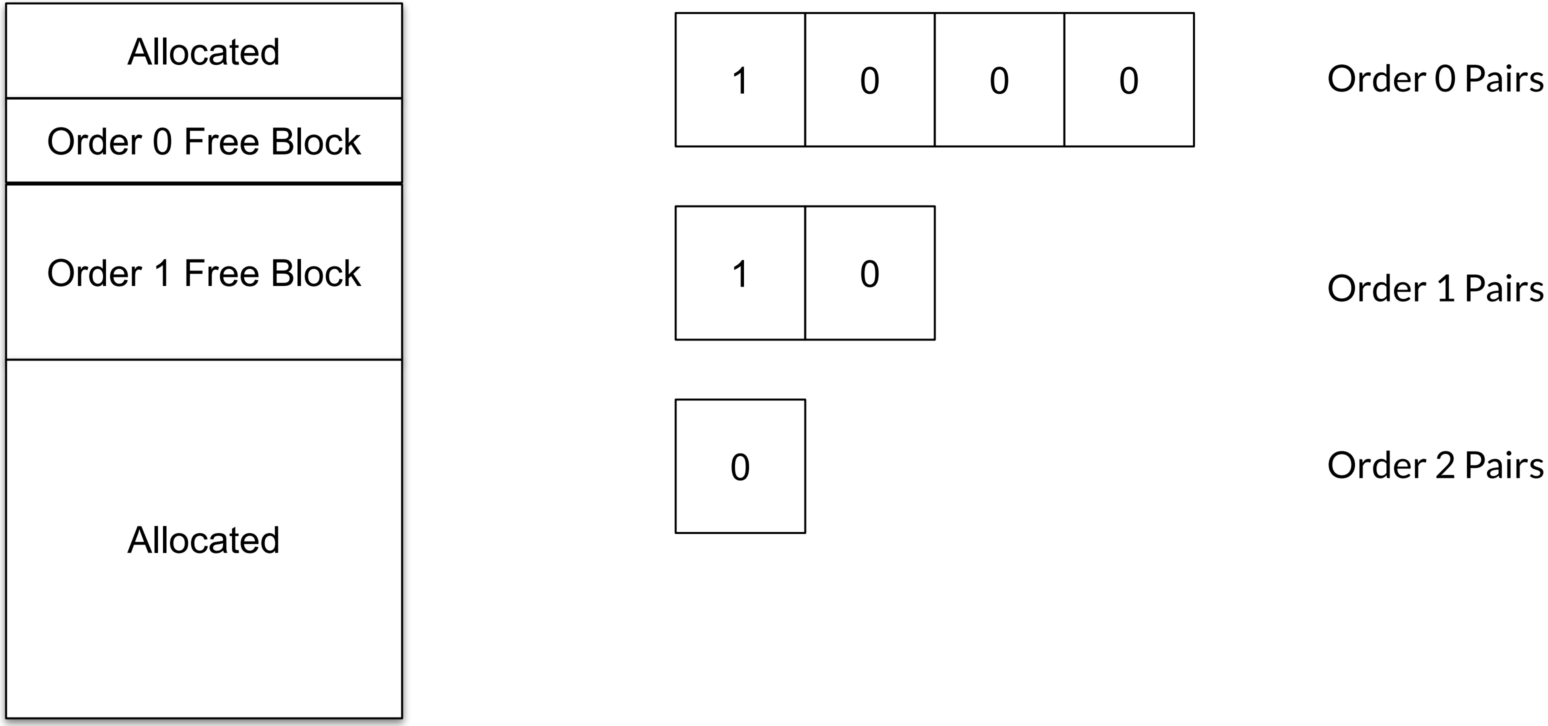
Allocate Order 2 Block



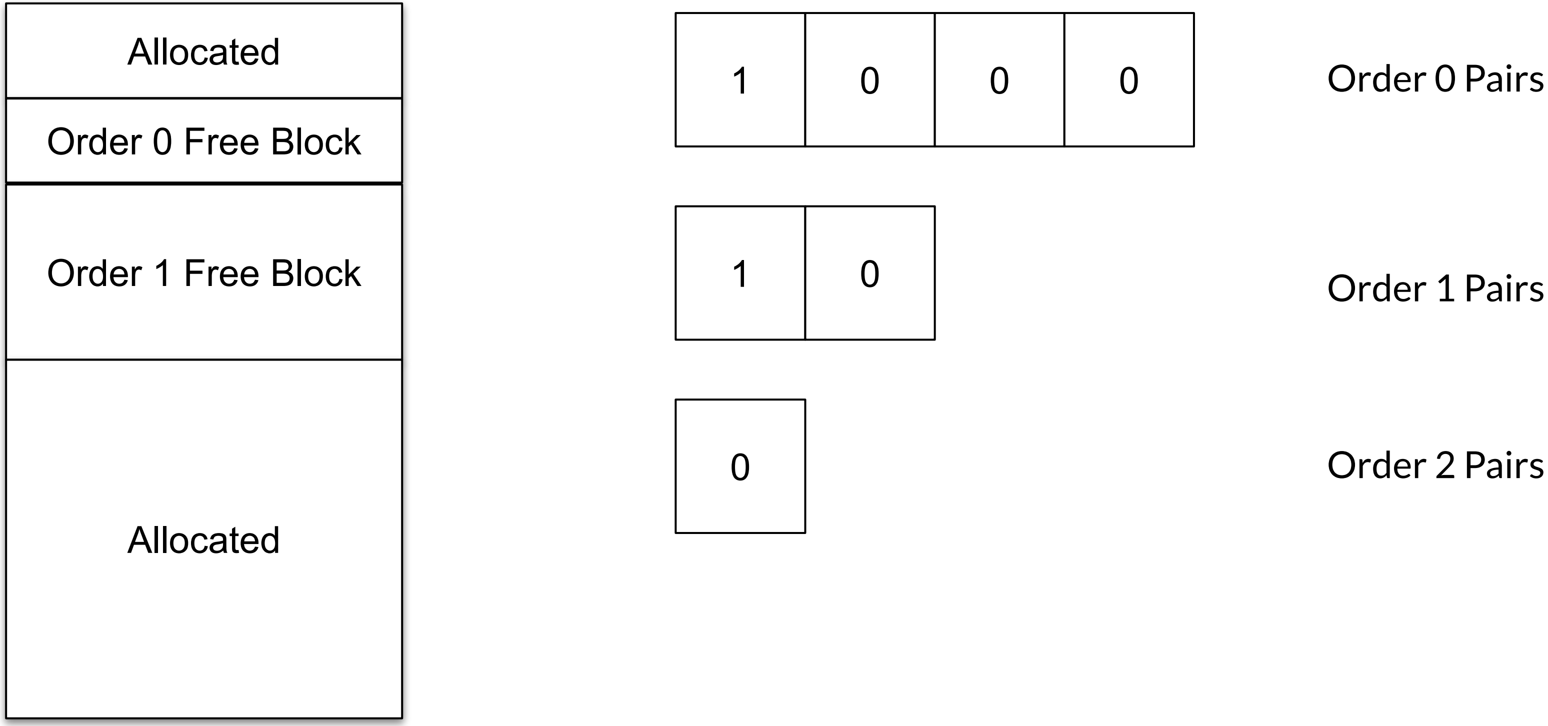
Free Order 0 Block



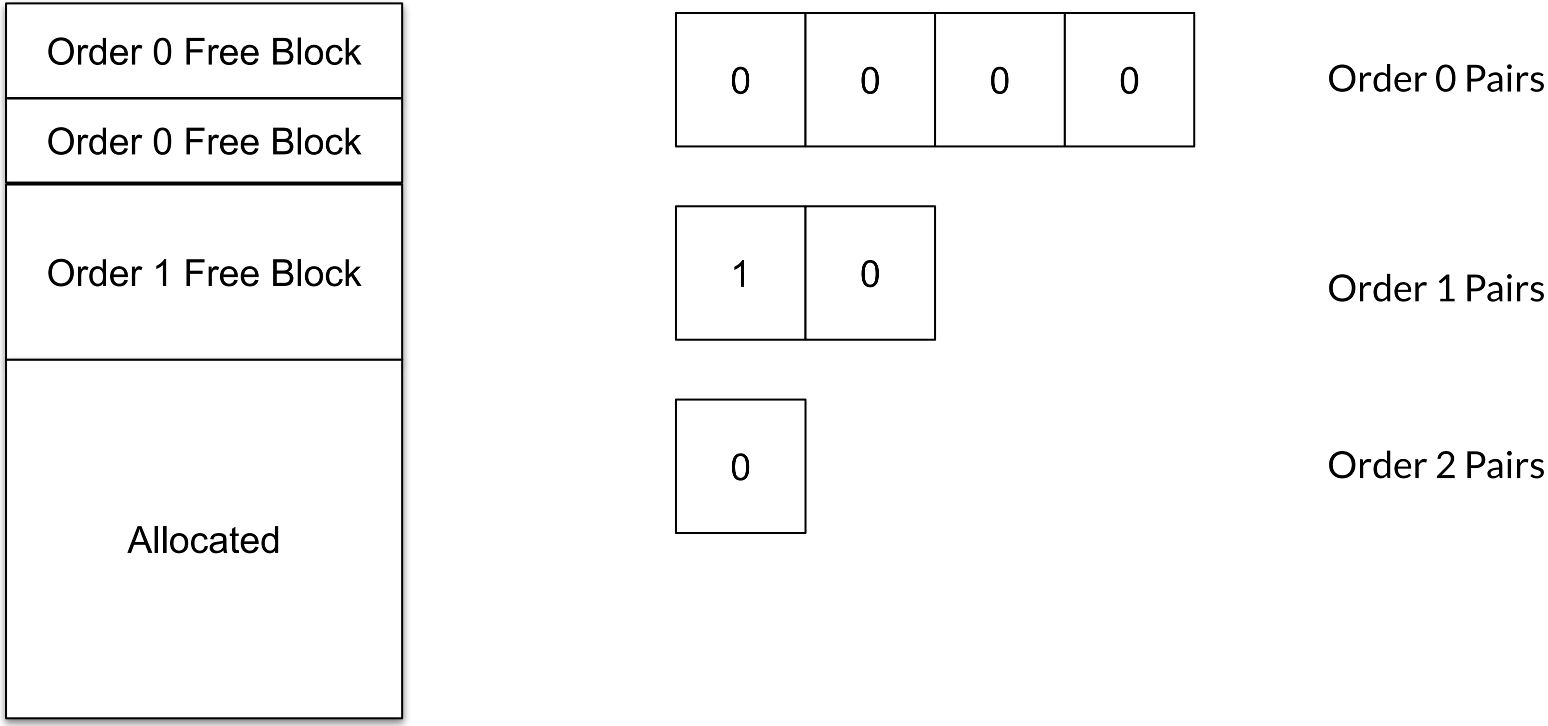
Free Order 0 Block



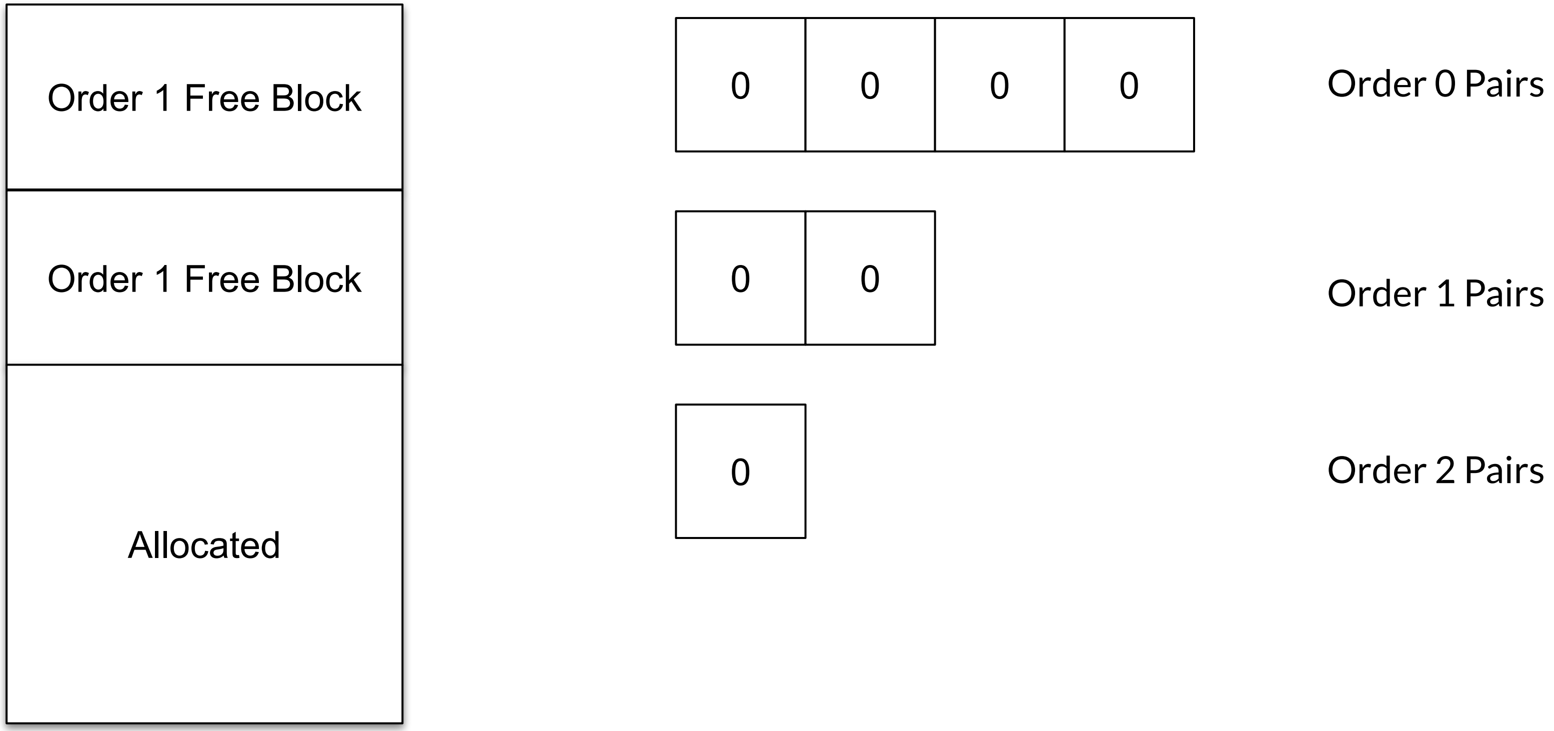
Free Other Order 0 Block



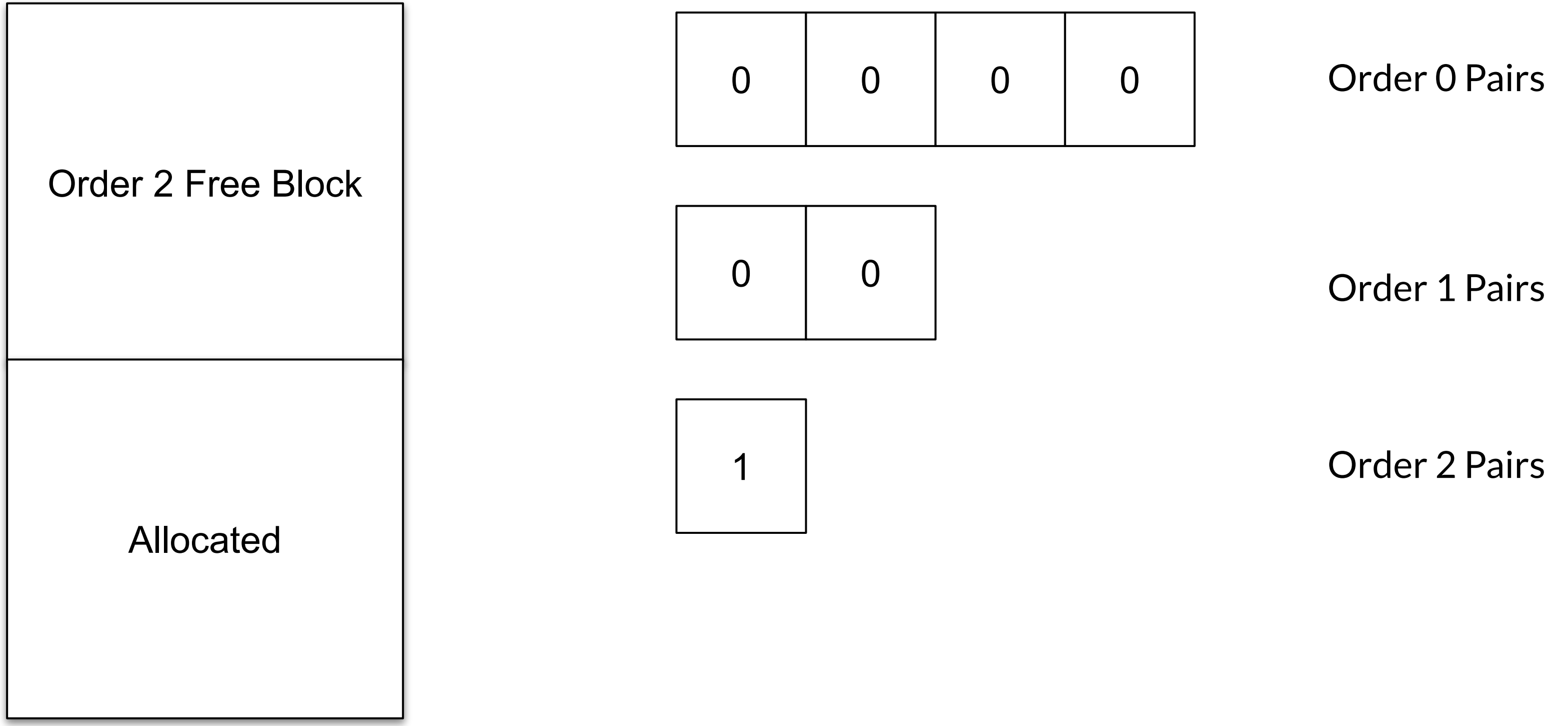
Free Other Order 0 Block



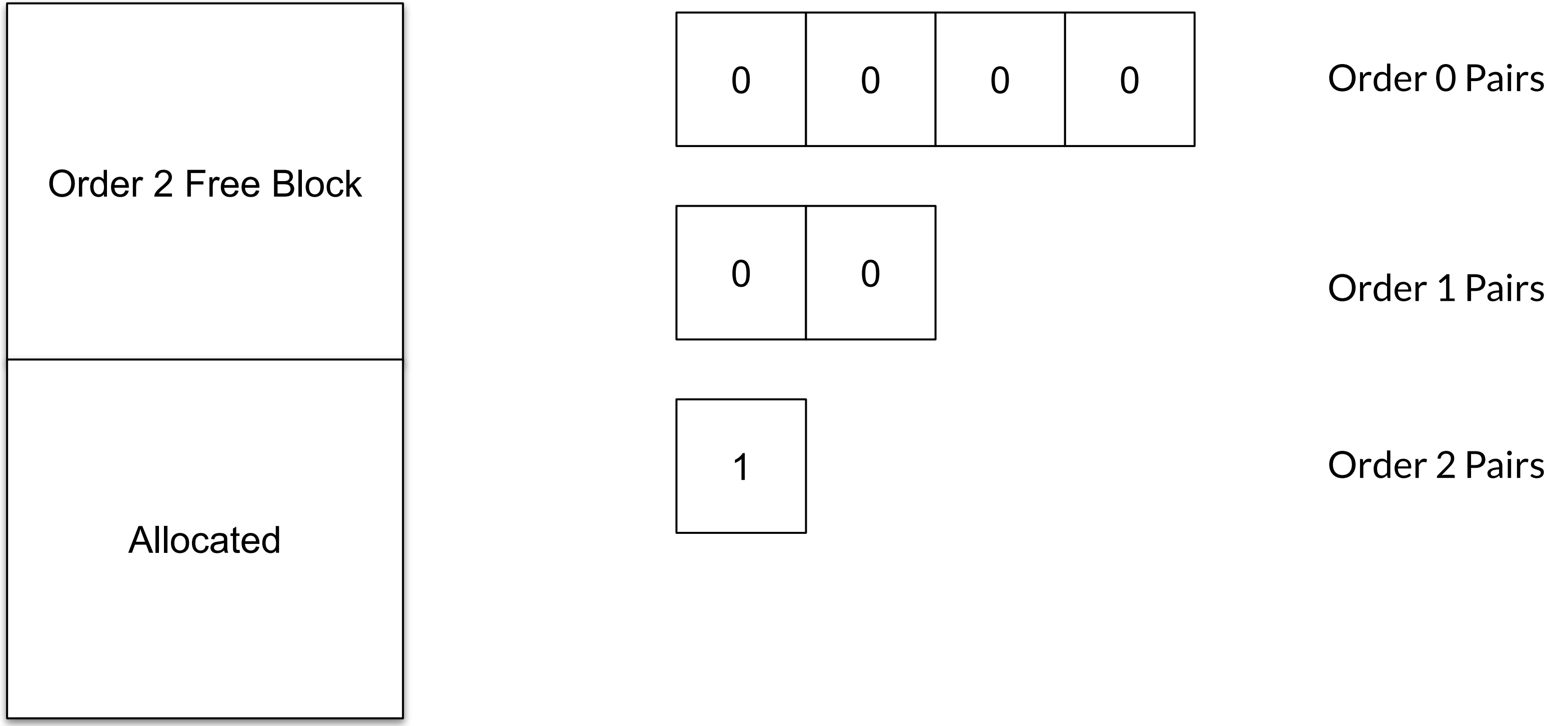
Free Other Order 0 Block



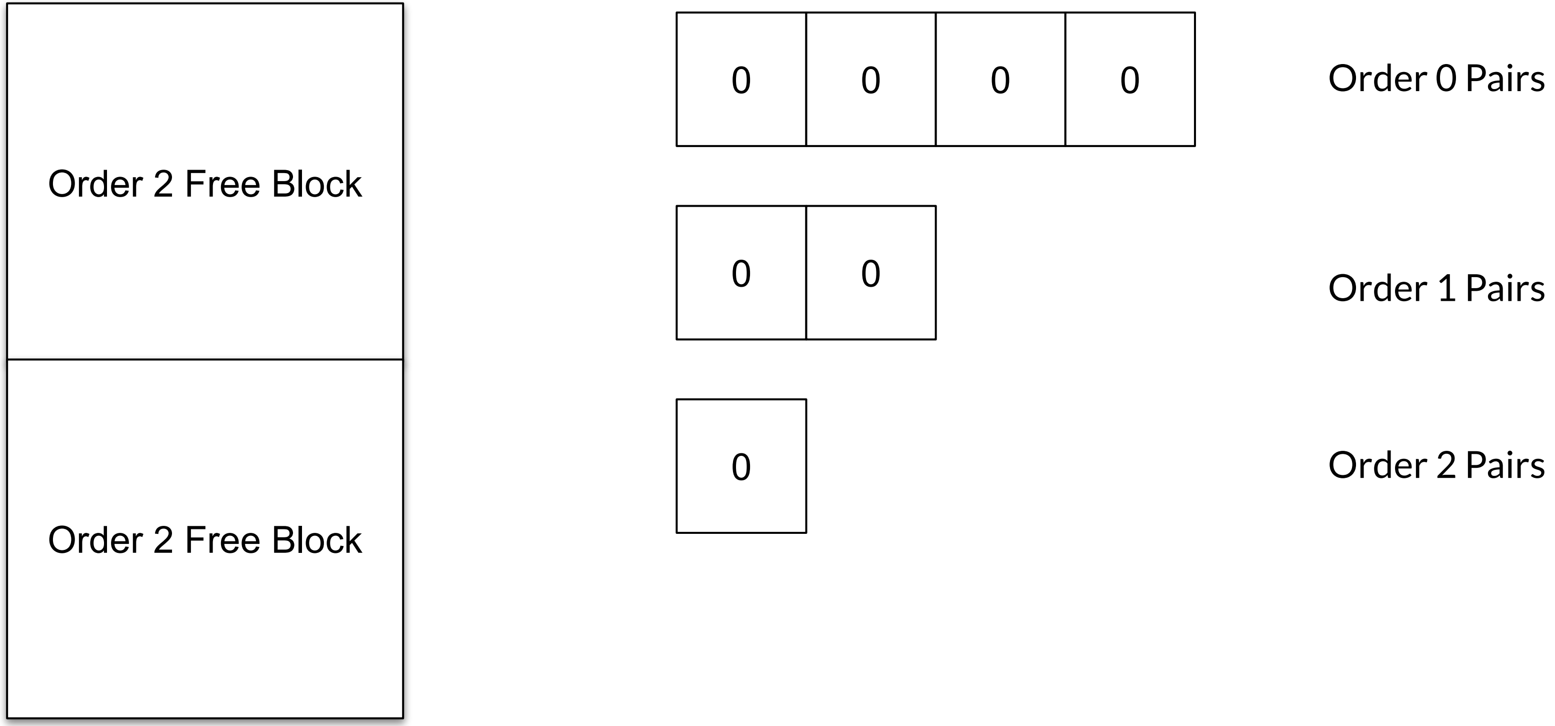
Free Other Order 0 Block



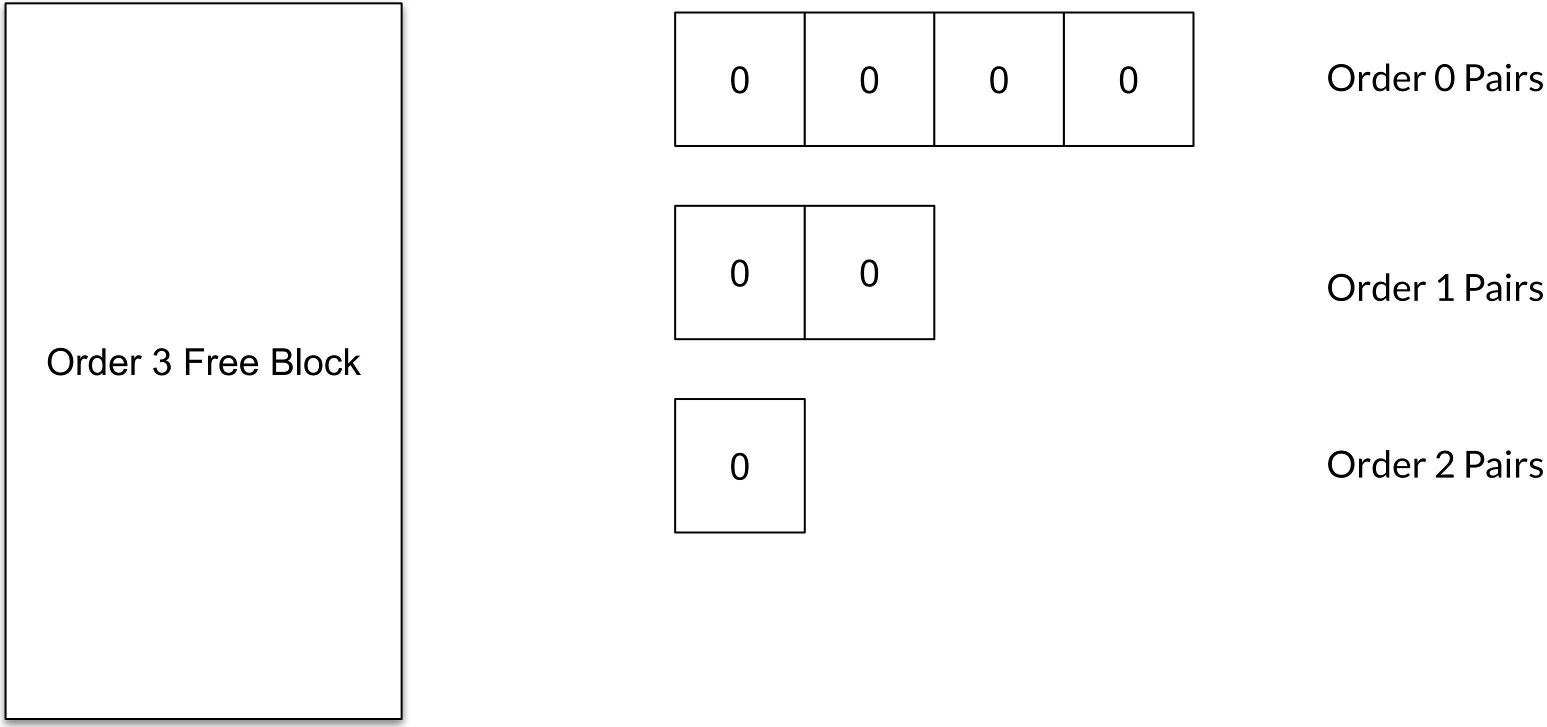
Free Order 2 Block



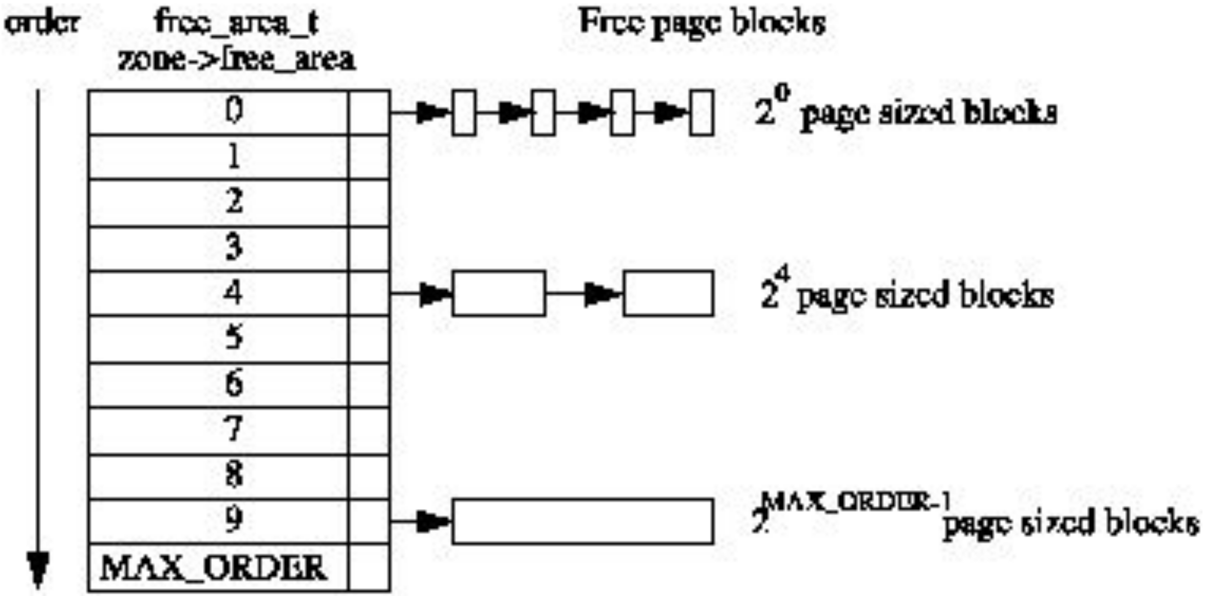
Free Order 2 Block



Free Order 2 Block



Free Page List



Buddy allocation issues

- Buddy allocation **reduces external fragmentation** (how?)
- However, it **doesn't prevent internal fragmentation** (why?)
  - Request 33 pages, best block it'll give you is 64 pages. 31 pages = 48% wasted.
- Linux's solution: layer "slab allocator" on top of buddy allocator to solve internal fragmentation.

Slab allocation: the big idea

- Some structs are allocated and freed very often. Examples:
  - task_struct, mutex’s, inodes, dentries
- There’s an overhead to destructing and constructing these objects over and over.
- Solution: **cache**! [Bonwick94]

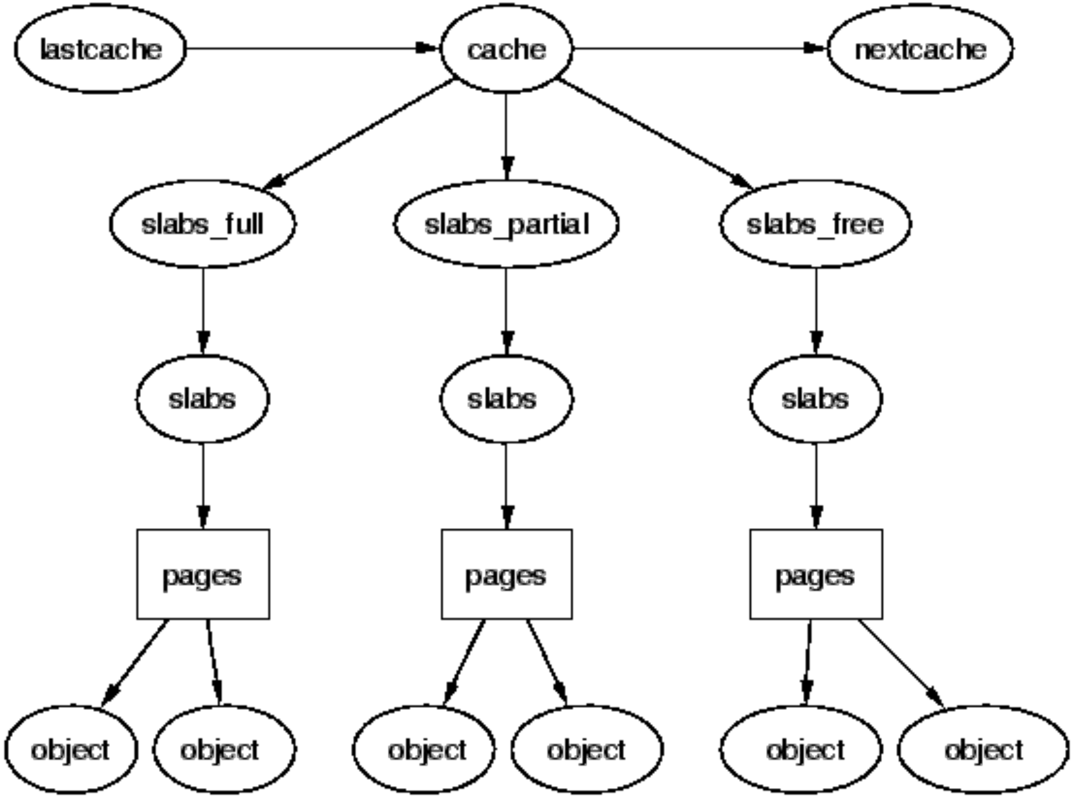
Slab allocation: 2 principal goals

- Allow allocation of small blocks of memory to help eliminate internal fragmentation that would be otherwise caused by the buddy system.
- Have caches of commonly used objects kept in an initialised state available for use by the kernel.
- Additional principle: HW cache-align objects to speed up access.

Object caching: basic algorithm

```
// Allocate an object
if (there’s an object in the cache)
  take it (no construction required);
else {
  allocate memory;
  construct the object;
}

// Free an object
return it to cache (no destruction --
                    simply return it
                    to init state);
```

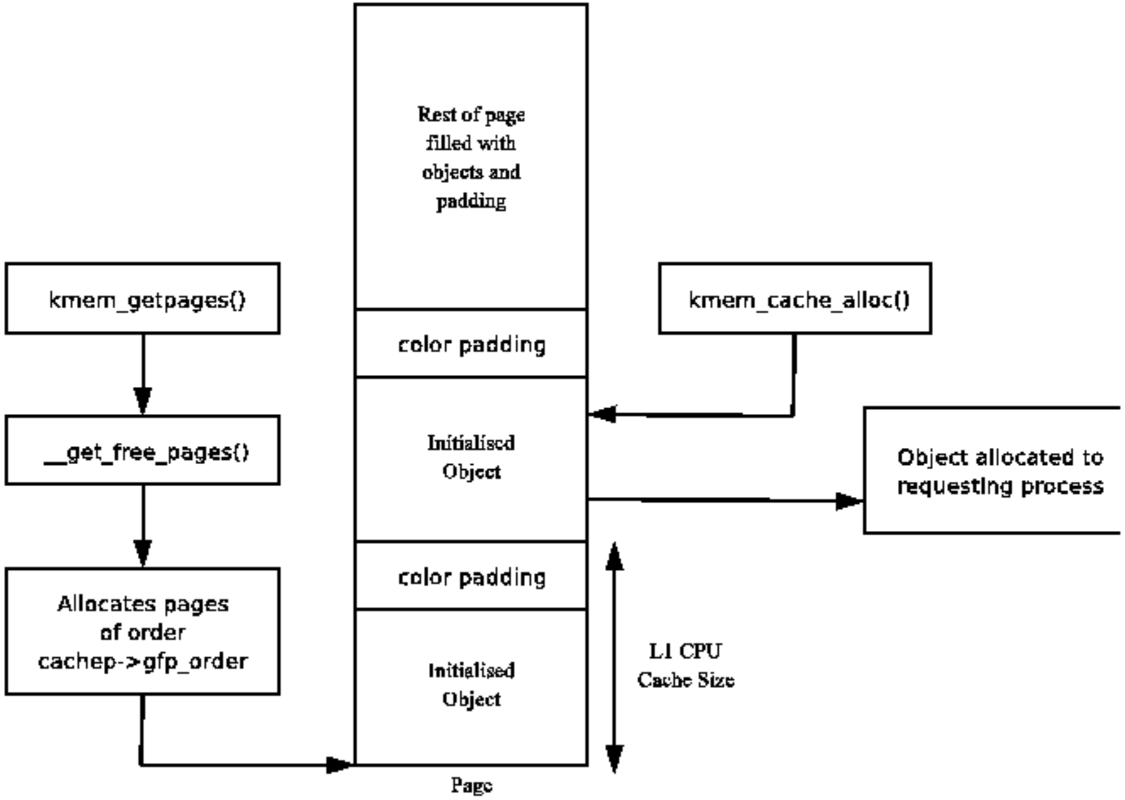
Slab cache structure: one cache per object type



Definitions:

- Slab: one or more pages of physically contiguous memory carved up into equal-size chunks.
- Slab cache: collection of slabs that correspond to a single object type.

Slab allocation: slightly more detail



Ex: your object type is a 24B struct. Your slab is 8192B (2 pages). Assuming no color padding, how many structs can you fit in the slab?

floor(8192/24) = 341.

kmalloc: lean on slab cache.

- Kernel has a separate bunch of caches specifically for kmalloc.
    - On a Linux box, try: `sudo vmstat -m | grep kmalloc.`
- These range from ~8K to 8B. (depends on arch).
- When kmalloc is called, it uses one of the dedicated slab caches to fulfill the request.
- Leaning on slab caches helps minimize internal fragmentation within a page, and allows dynamic allocation of small buffers.

```
$ sudo vmstat -m | grep "kmalloc"
Cache                        Num  Total   Size  Pages
...
kmalloc-8k                   297    312   8192      4
kmalloc-4k                  1028   1072   4096      8
kmalloc-2k                  1655   1664   2048     16
kmalloc-1k                  2711   2720   1024     32
kmalloc-512                34017  34592    512     32
kmalloc-256                13146  13344    256     32
kmalloc-192                 3309   3402    192     42
kmalloc-128                 3609   3712    128     32
kmalloc-96                  6621   7854     96     42
kmalloc-64                 20124  21120     64     64
kmalloc-32                 15363  15744     32    128
kmalloc-16                 28787  31232     16    256
kmalloc-8                  15323  15360      8    512
...
```

vmalloc

- If we want lots of memory, finding physically contiguous memory may be tricky despite buddy allocator.
- Linux provides vmalloc, which gets a large virtually contiguous chunk of memory but not necessarily contiguous physically.
- `vmalloc`/`vfree` is the interface.

Scheduling

Why do we need scheduling?

- Scheduling provides the *illusion* of multiple processes running simultaneously without additional hardware requirements
- Ensures fairness of CPU usage
- Make computers more useful
    - Imagine if your computer froze for hours every time you started a simulation

Scheduling Policies

First In-First Out: Process are executed in the order they arrive, and allowed to run to completion

Round Robin: Schedule waiting processes in a circular manner, with each process running for some time slice before being interrupted and rescheduled

Shortest Job First: Of the waiting processes, the one estimated to take the *least amount of time* is allowed to run to completion. Can be preempted if a shorted process arrives.

Priority Levels

- Why should schedulers care about task priority? Isn't this unfair?
  - Some tasks are time sensitive
  - Some tasks are significantly more important than others
- Interactive applications typically need fast response times to be useful
  - Want to react fast to input
- CPU bound applications will typically have lower priority
  - Simulations

Memory Maps

How do memory maps work?

- What’s shared?
  - C Library
- What’s not shared?
  - Heap, stack, Executable code (Most important, executable for lib is shared but the actual executable code running is not)
- Permissions:
  - R – read, W – write, X – execute
- Total memory between processes
  - Sum together pages, making sure to only count libraries once over all processes

MP3 + etc.

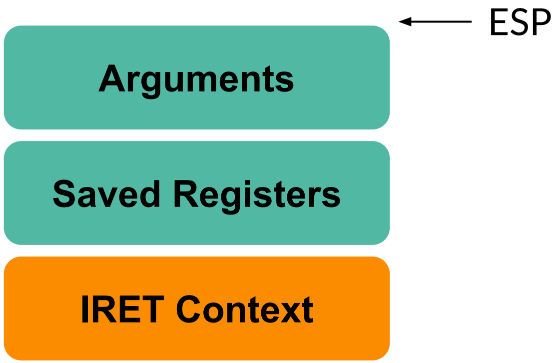
Implementing execute()

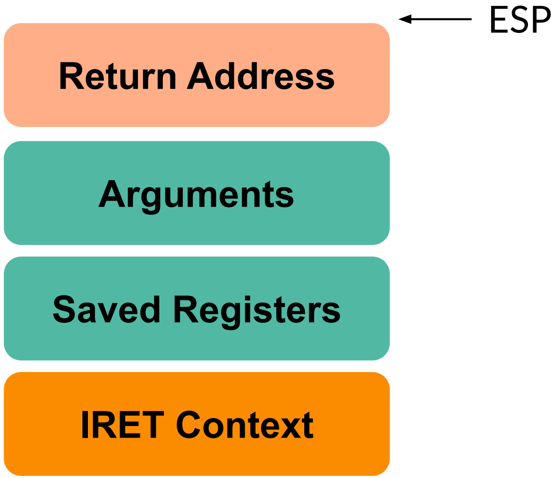
1. Generate a PCB
    - PID, parent PID,
2. Modify paging
    - New user page for program data
    - Flush TLB
3. Load user program into memory
4. Modify TSS (esp0)
    - Why don’t we touch anything else?
5. Setup userspace IRET context
    - SS, ESP, EFLAGS, CS, EIP

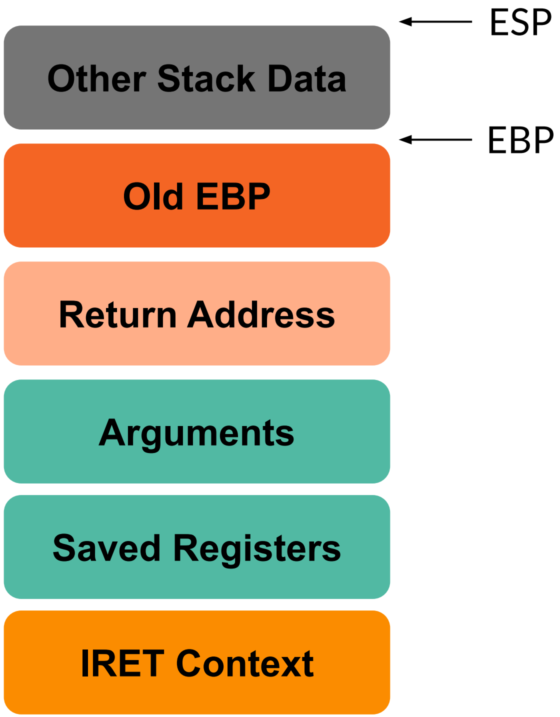
```
ece391> cat frame0.txt

/\/\/\/\/\/\/\/\/\/\/\/\
         o
           o    o
       o
             o
        o     O
    _    \
 |\/.\   | \/  /  /
 |=  _>   \|   \ /
 |/\_/    |/   |/
----------M----M--------


ece391> ls_
```

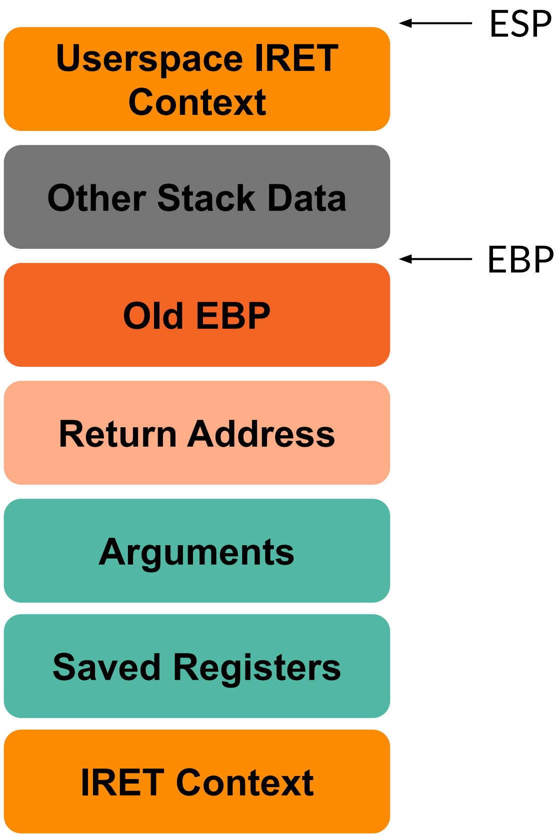

ece391> cat frame0.txt
ece391> ls
INT
$0x80

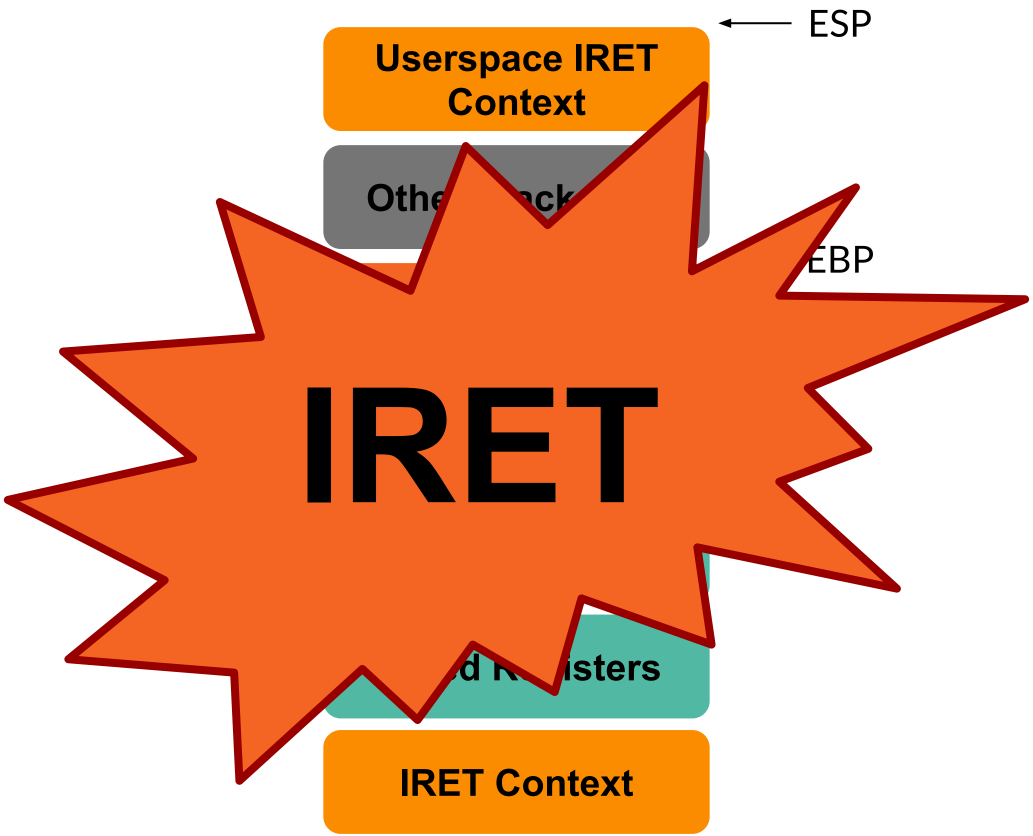

ESP
IRET Context

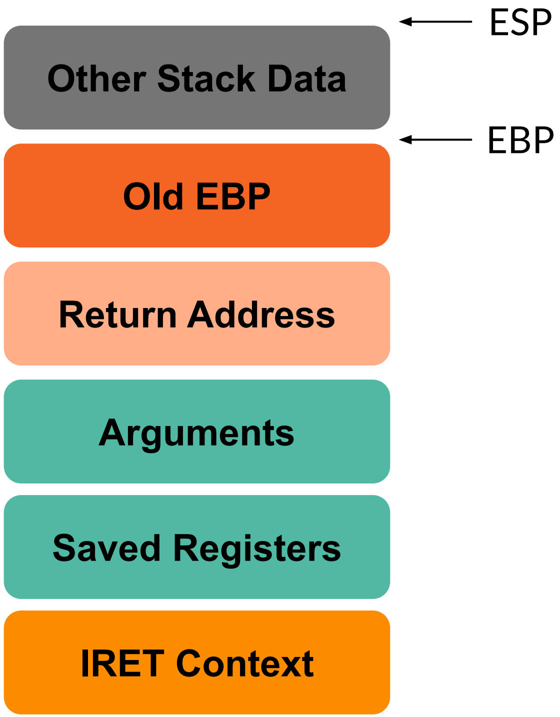

ESP
Arguments
Saved Registers
IRET Context

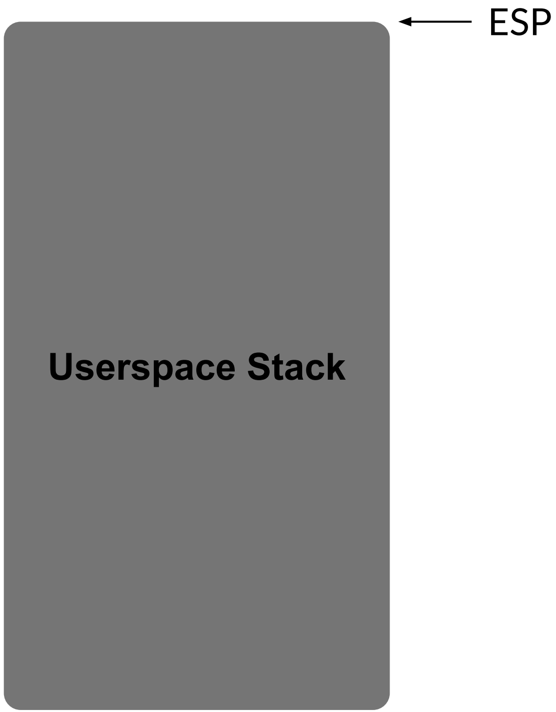

ESP
Return Address
Arguments
Saved Registers
IRET Context


ESP
Other Stack Data
EBP
Old EBP
Return Address
Arguments
Saved Registers
IRET Context

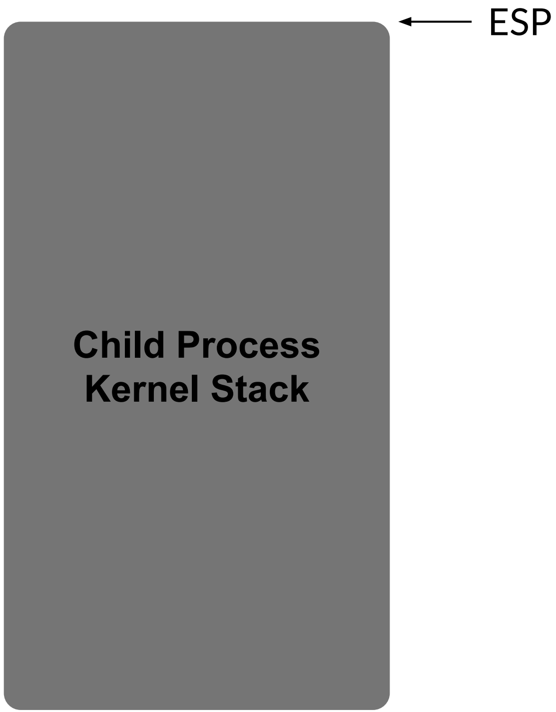

ESP
Userspace IRET Context
Other Stack Data
EBP
Old EBP
Return Address
Arguments
Saved Registers
IRET Context


ESP
Userspace IRET Context
EBP
IRET
IRET Context

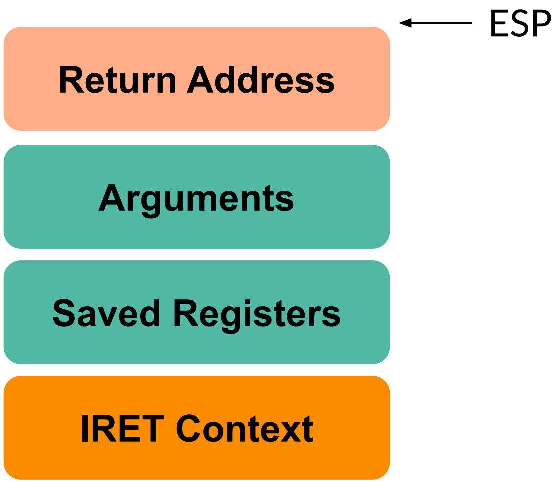

ESP
Other Stack Data
EBP
Old EBP
Return Address
Arguments
Saved Registers
IRET Context

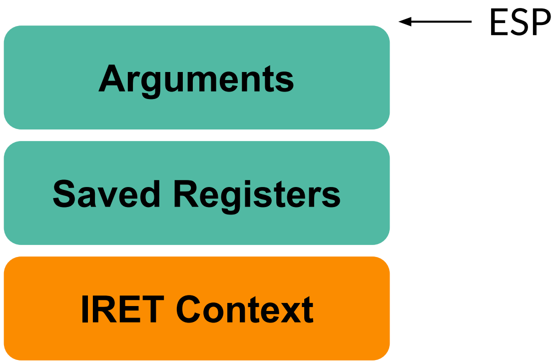

ESP
Userspace Stack


ESP
INT
$0x80


ESP
Child Process
Kernel Stack


EBP
Old EBP
Return Address
Arguments
Saved Registers
IRET Context


ESP
Return Address
Arguments
Saved Registers
IRET Context



ESP
Arguments
Saved Registers
IRET Context

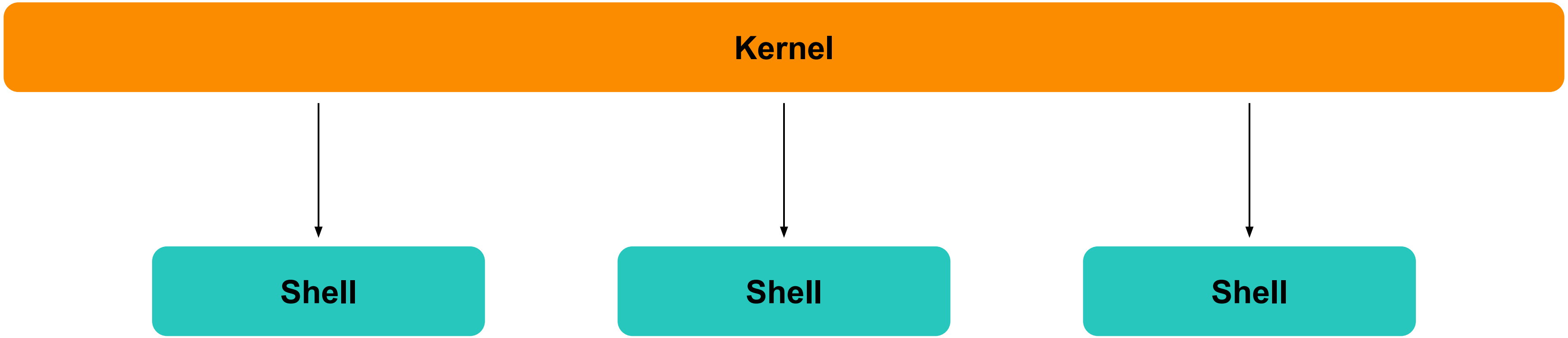

ESP
IRET Context

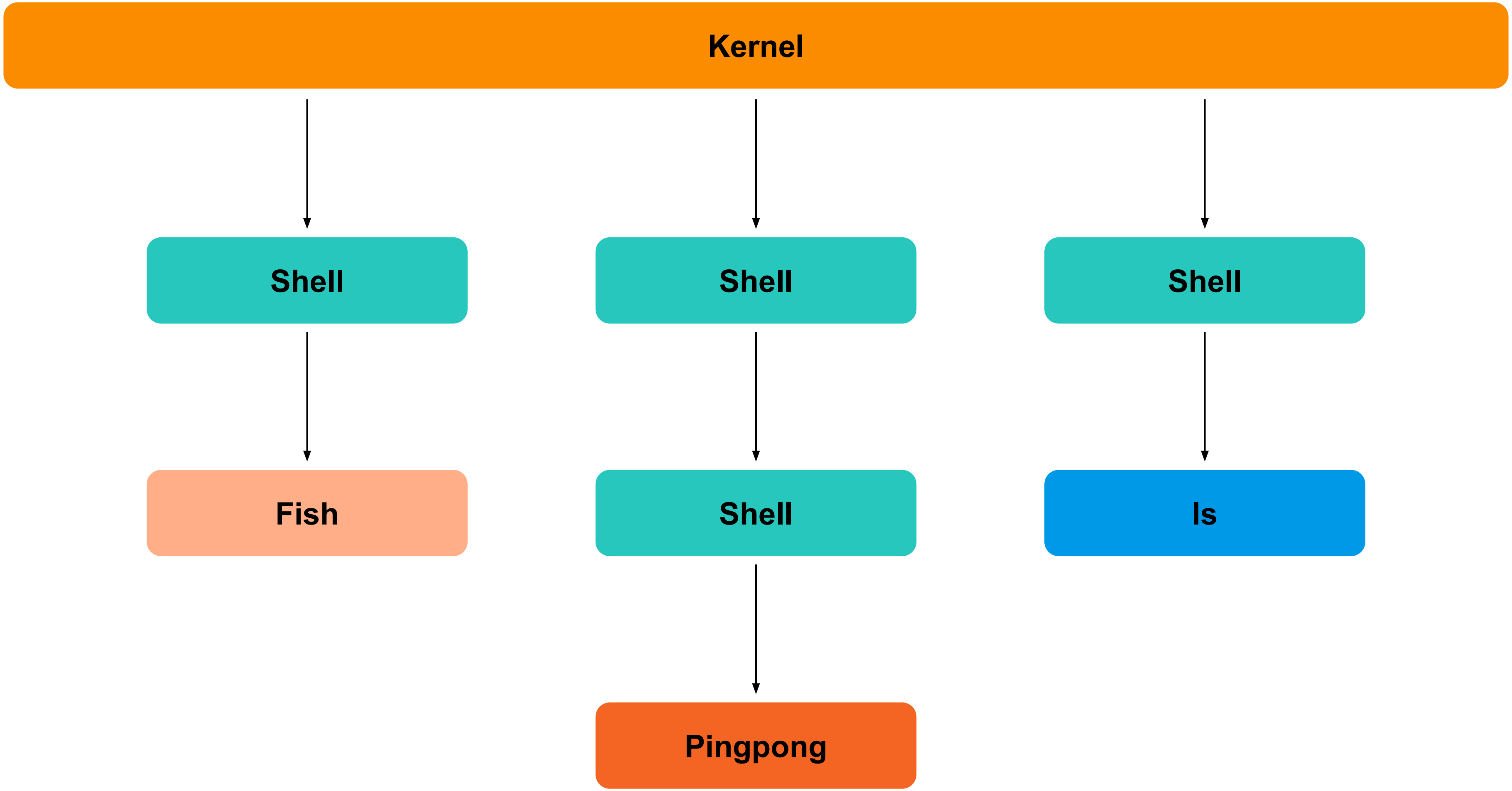
```
ece391> cat frame0.txt

/\/\/\/\/\/\/\/\/\/\/\/\
         o
           o    o
       o
             o
        o     O
    _    \
 |\/.\   | \/  /  /
 |=  _>   \|   \ /
 |/\_/    |/   |/
----------M----M--------


ece391> ls
.
hello
frame0.txt
...
```

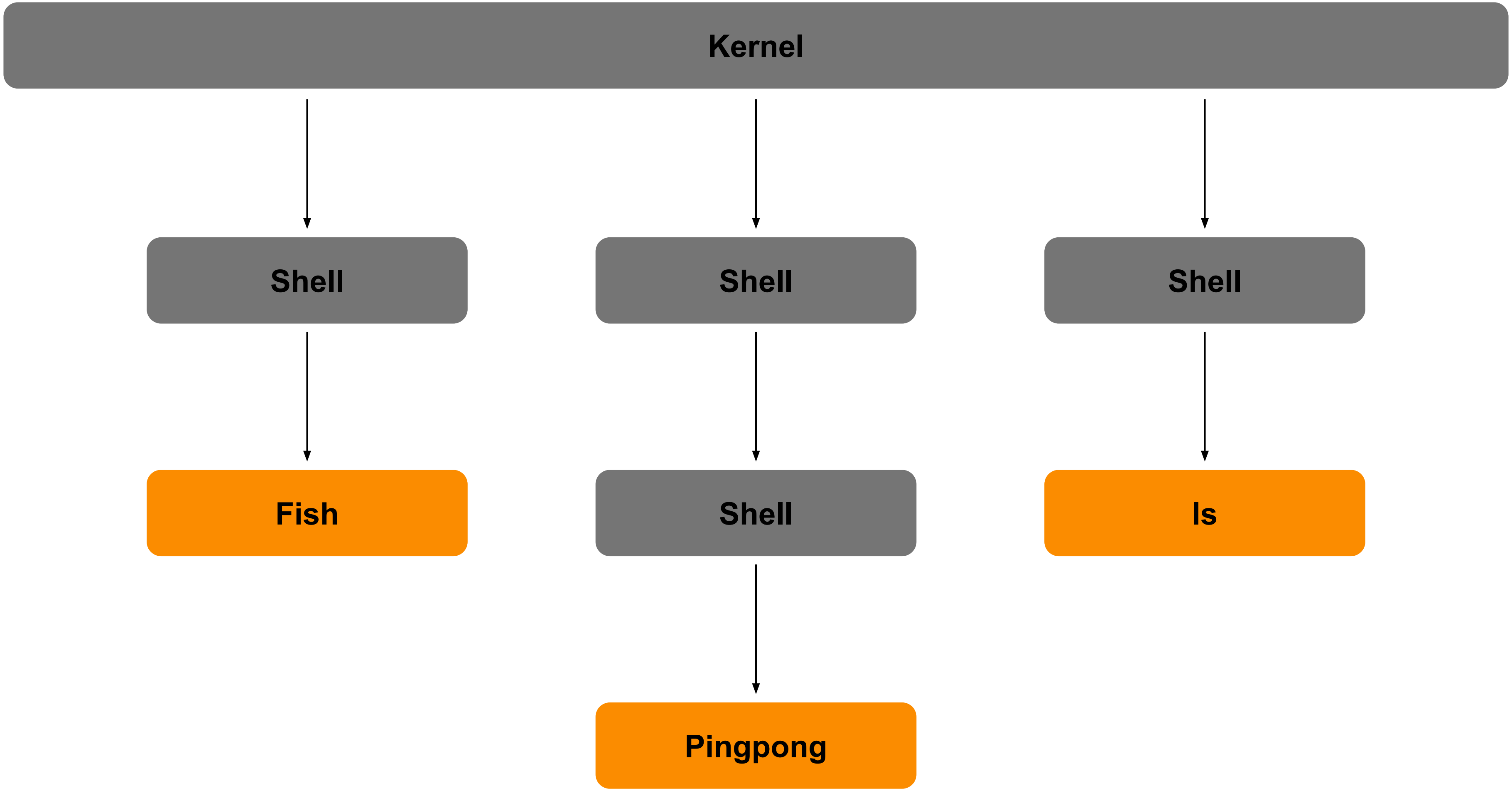
Terminals + Scheduling

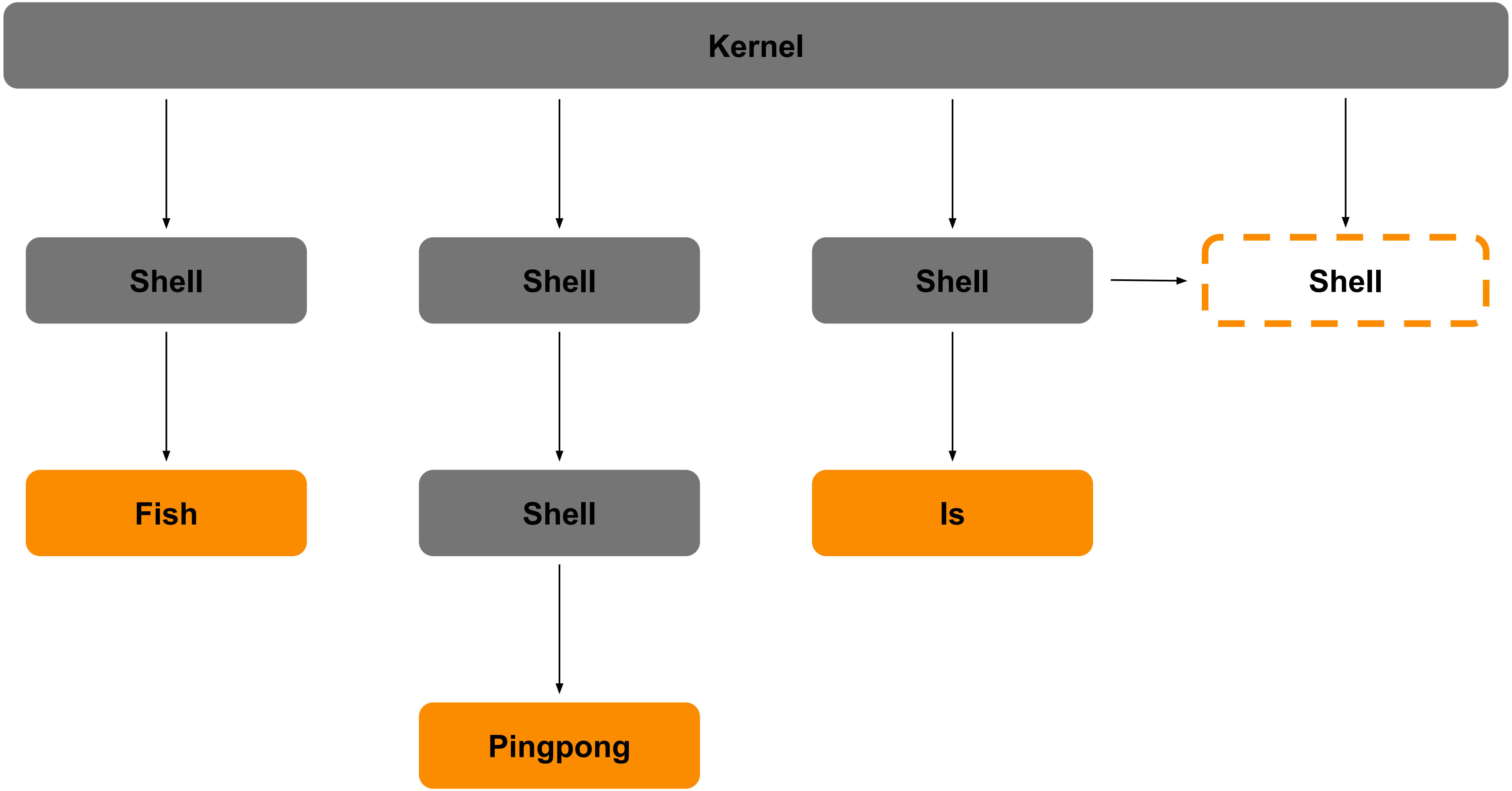
- Three terminals -> three leaf processes
  - Up to 6 *total* processes
- Switch between terminals using keyboard interrupts
  - Remap video memory
- Round robin scheduling
  - Switch processes on PIT interrupts
  - Takes advantage of IRET context set up by processor

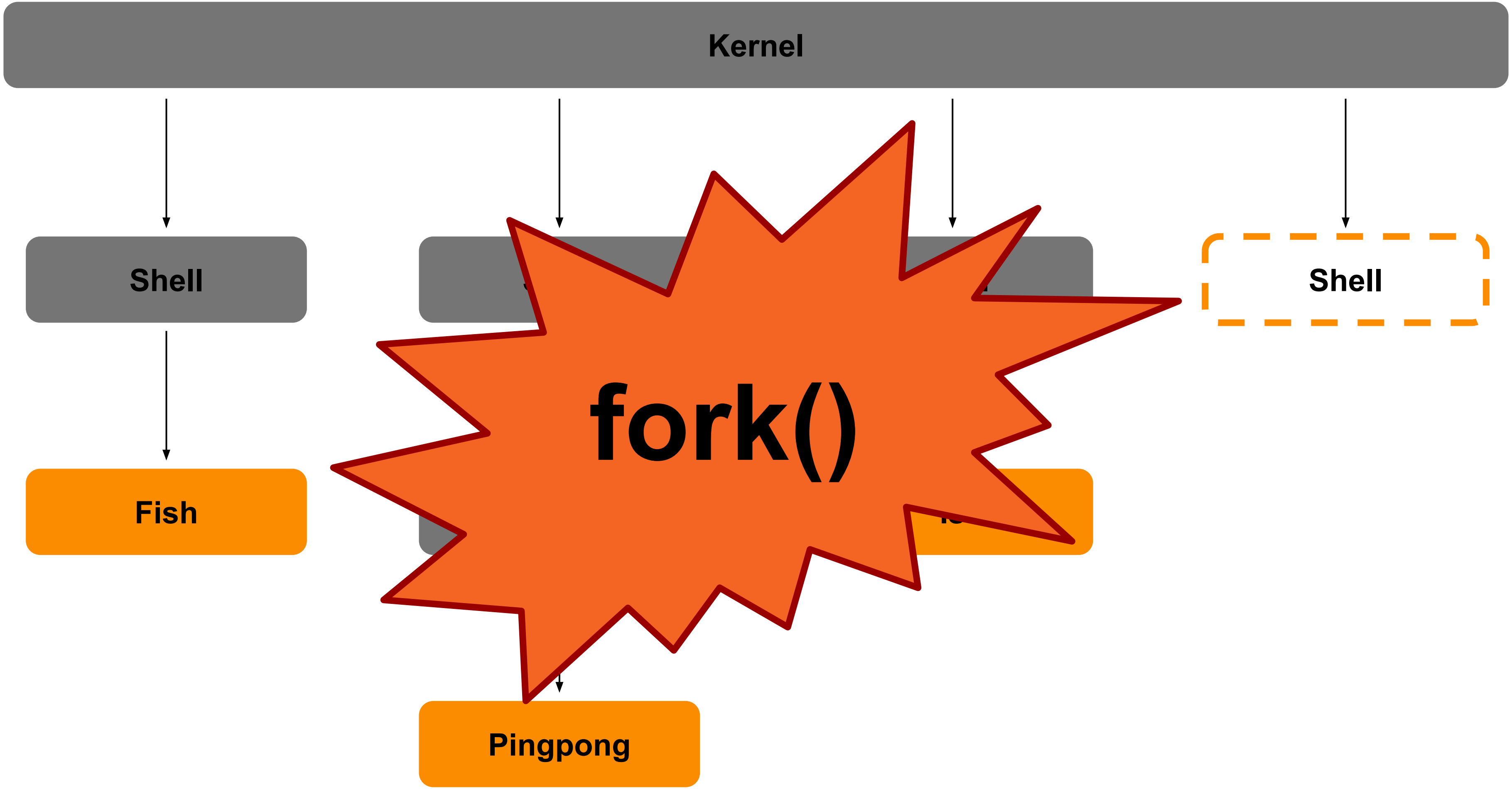
Kernel


Kernel
Shell
Shell
Shell


Kernel
Shell
Shell
Shell
Fish
Shell
ls
Pingpong


Kernel
Shell
Shell
Shell
Fish
Shell
ls
Pingpong


Kernel
Shell
Shell
Shell
Shell
Fish
Shell
ls
Pingpong


Kernel
Shell
Shell
Fish
fork()
Pingpong

The fork() System Call

- Why is fork useful?
  - Create new ‘leaf process’
- Duplicates the process + any metadata associated with it
  - Copy on write
- Different return value for parent / child
  - Parent gets Child PID, child gets 0
- Often used in conjunction with exec()