How Does An Operating System Manage Memory?

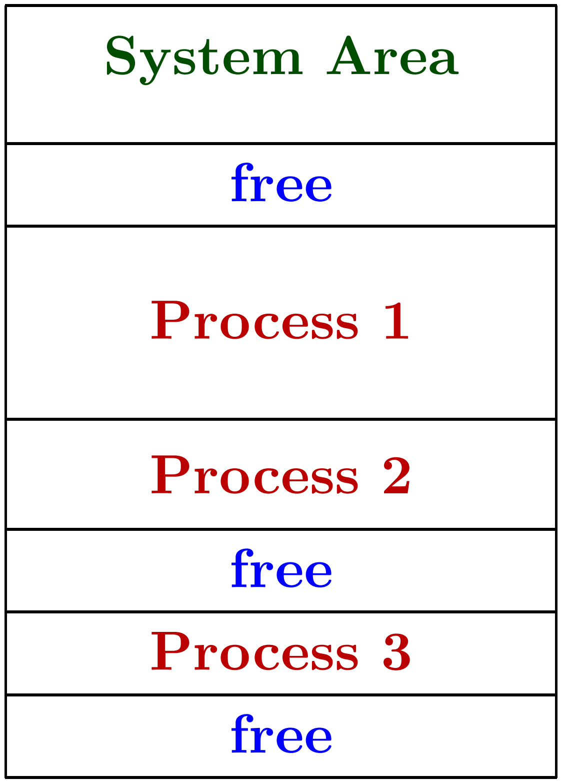
John Franco
Electrical Engineering and Computing Systems
University of Cincinnati

Memory Hierarchy

Type	**Capacity**	**Latency**
Registers	A few `KB`	1 ns
L1 Cache	About 1 `MB`	3 cycles
L2 Cache	About 5 `MB`	15 cycles
RAM	About 8 `GB`	50 ns
Hard drive	About 1 `TB`	1 ms

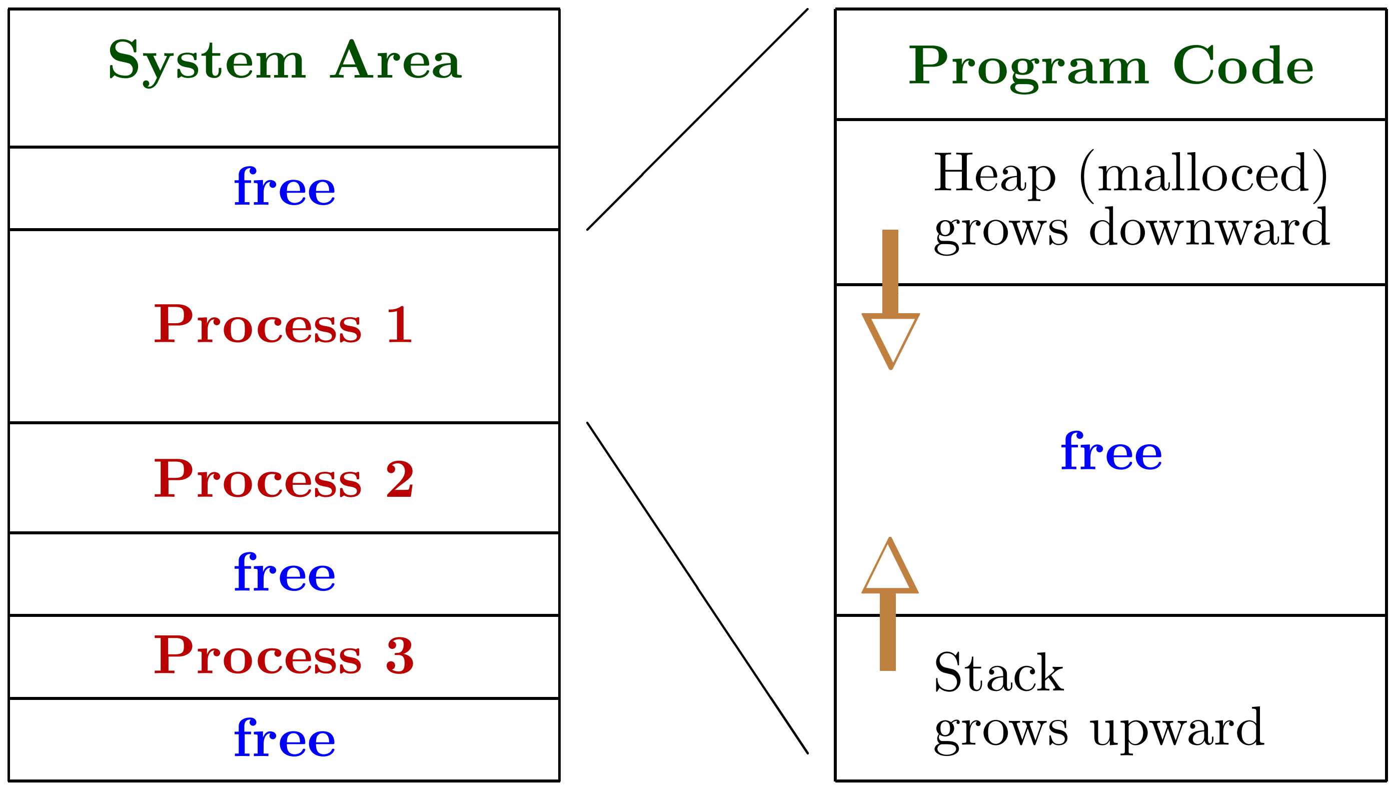
- It is desired to have as much of currently relevant data in the higher memory types as possible.
- But this is not feasible so data is constantly moved between memory types using some sophisticated algorithms.
- The hard drive is typically used for file storage but a partition of the hard drive may be used to act like RAM. This is called virtual memory.
- Data structures are employed for efficient data handling.
  For example: red-black trees, fibonacci heaps, B-trees.
- The OS handles all of the above.

Processes in Memory



- The OS occupies space in low memory (System Area)
- Each process occupies some space, possibly interspersed with free space
- There are usually many processes but few processors, so some scheduler must switch a processor to run on a collection of processes
- But if there is one set of registers that hold state, filling those registers for every context switch will cost a lot of time

Address Space



- Stack: keeps track of function call chain, parameter passing
- Heap: dynamically allocated, user-managed memory, static variables
- What to do with multiple threads in the same address space?
- Virtualization: a process thinks it has lots of contiguous memory starting at address 0 but in reality this is not the case

Program code segment

- start
  - `%esi` gets `argc`, `%ecx` gets `argv`
- init
  - section of code that is run before `main`
- plt
  - Procedure Linkage Table: makes dynamic loading easier
- text

  Actual machine instructions that your CPU going to execute.
  This is sharable - hence read only
  - `get_pc_thunk.bx`: loads location of the code into the `%ebx` register allows global objects (which have a fixed offset from the code) to be accessed as an offset from that register.
  - `(de)register_tm_clones`: transaction memory - a concurrency control mechanism for controlling access to shared memory in concurrent computing
  - `do_global_[c|d]tors_aux`: call [con|de]structors of static objects
  - `frame_dummy`: sets up exception handling
  - `main`: the `main` procedure
- fini
  - executed after `main` finishes normally

Program code segment

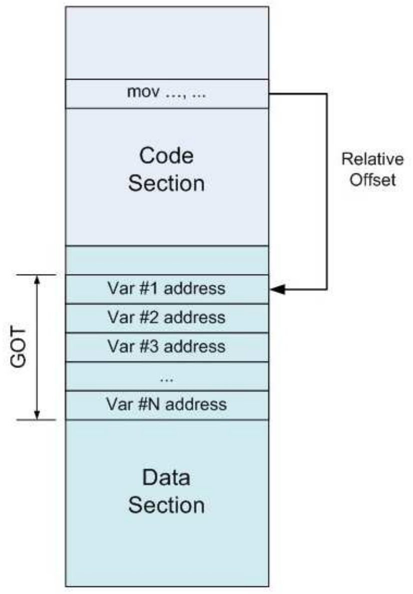
- data

  All initialized, declared variables in a program, for example:

  `float salary = 123.45;`

  There is a read-only subsection and a read-write subsection

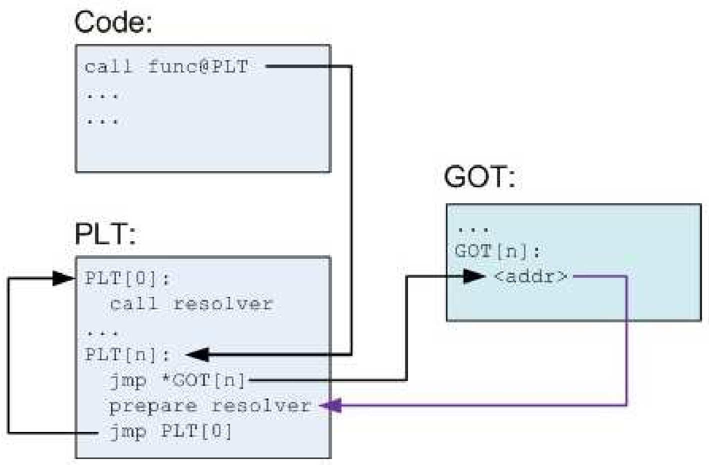
  Includes the Global Offset Table (GOT), a table of addresses, residing in the data section. When an instruction in the code section refers to a variable, instead of referring to it directly by absolute address (which would require a relocation), it refers to an entry in the GOT. Since the GOT is in a known place in the data section, this reference is relative and known to the linker. The GOT entry contains the absolute address of the variable.



Program code segment

- bss
  Uninitialized data including arrays. No initial values have been set

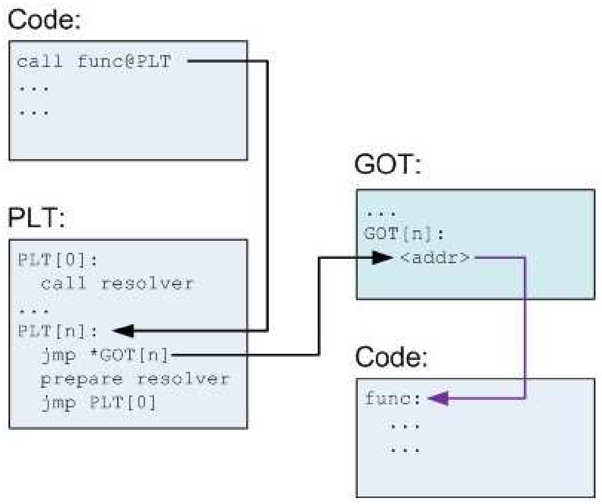
Procedure Linkage Table



- Initially, function calls have not yet been resolved
- A function `f` is called, it translates to `f@plt`, an entry in the PLT
- The `PLT` consists of an entry containing:
  1) a jump to a location which is specified in a corresponding `GOT` entry
  2) preparation of arguments for a "resolver" routine
  3) call to the resolver routine (first entry of the PLT)
- The 1st routine resolves the actual address of the function

1. PLT[n] is called and jumps to the address pointed to in GOT[n]
2. This address points into PLT[n] itself, to the preparation of arguments for the resolver
3. The resolver is then called
4. The resolver resolves the actual address of `f`, places the actual address into GOT[n] and calls `f`

Procedure Linkage Table

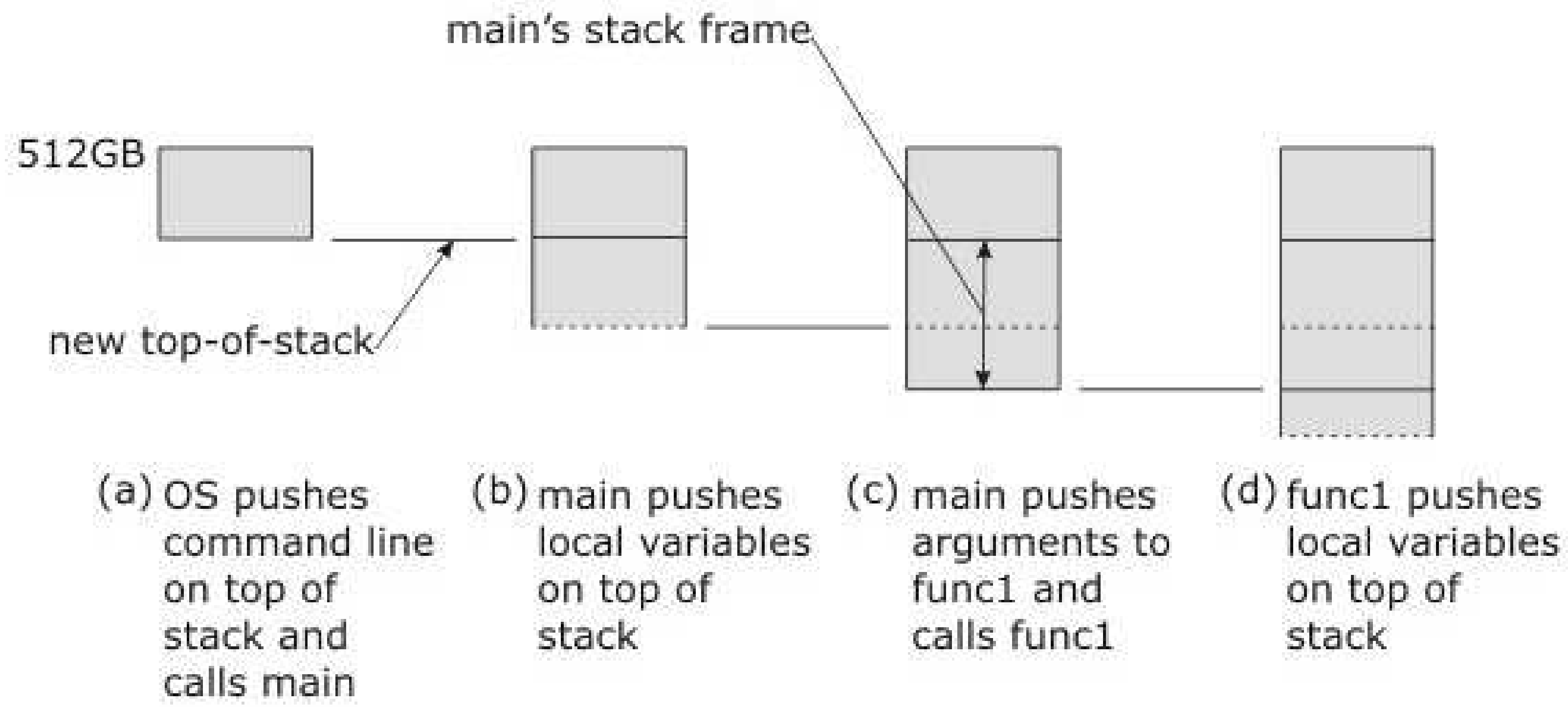


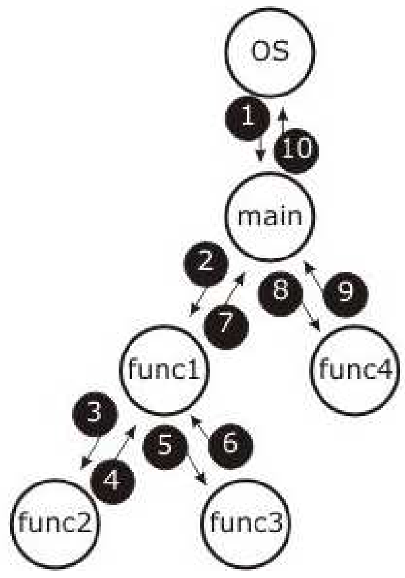
GOT[n] now points to the actual `f` instead of back into the `PLT`; when `f` is called again:

1. PLT[n] is called and jumps to the address pointed to in GOT[n]
2. GOT[n] points to `f`, so this just transfers control to `f`

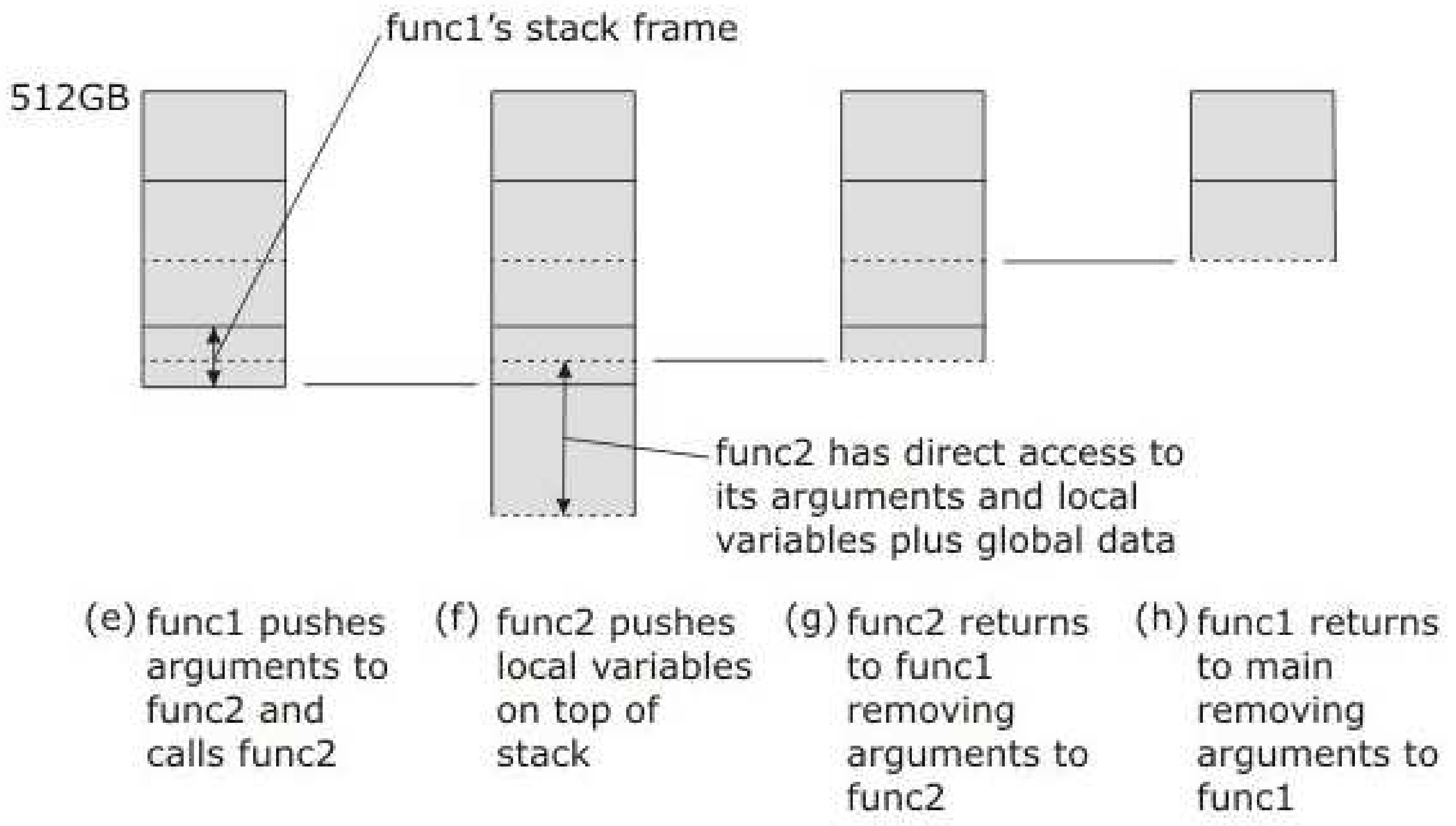
- Now `f` is actually being called, not the resolver
- This mechanism allows lazy resolution of functions, and no resolution at all for functions that are not actually called
- It also leaves the code/text section of the library completely position independent: the only place where an absolute address is used is the GOT, which resides in the data section and will be relocated by the dynamic loader.

Evolution of the stack segment

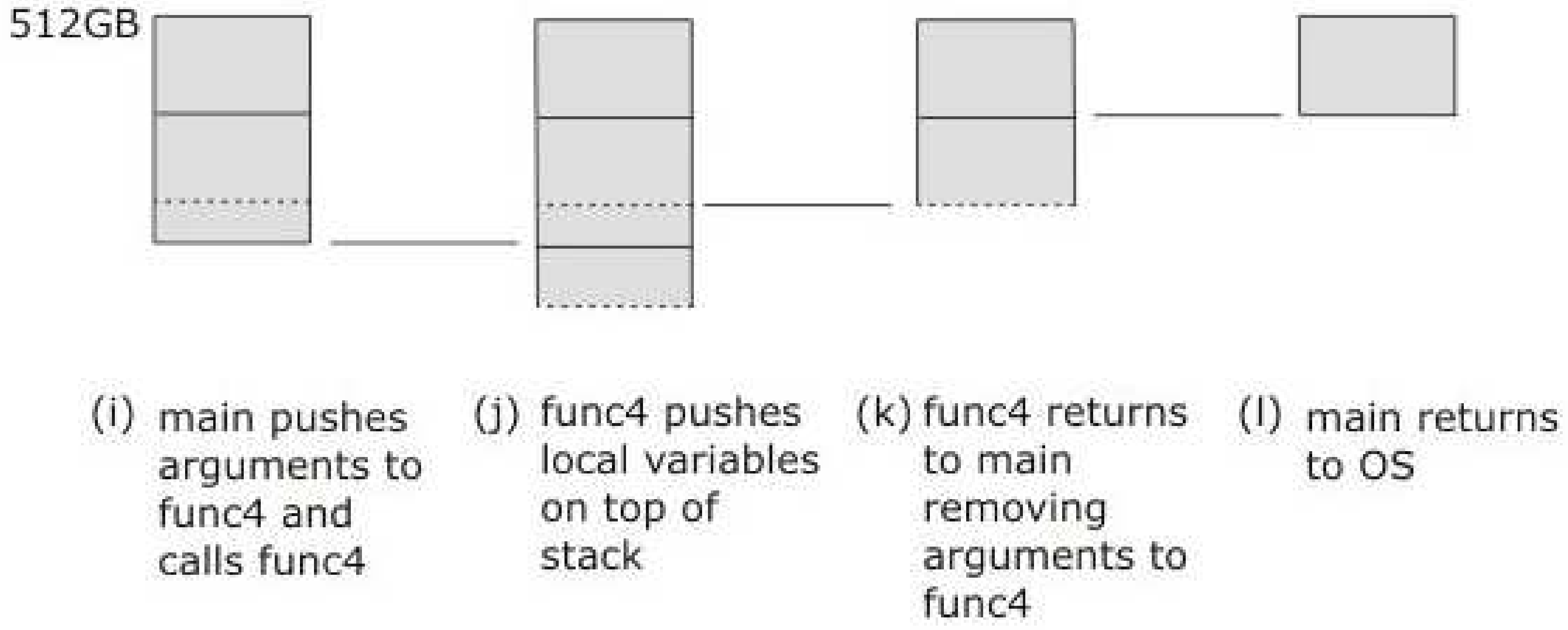

main's stack frame
512GB
new top-of-stack
(a) OS pushes command line on top of stack and calls main
(b) main pushes local variables on top of stack
(c) main pushes arguments to func1 and calls func1
(d) func1 pushes local variables on top of stack



Evolution of the stack segment


func1's stack frame
512GB
func2 has direct access to its arguments and local variables plus global data
(e) func1 pushes arguments to func2 and calls func2
(f) func2 pushes local variables on top of stack
(g) func2 returns to func1 removing arguments to func2
(h) func1 returns to main removing arguments to func1

Evolution of the stack segment


512GB
(i) main pushes arguments to func4 and calls func4
(j) func4 pushes local variables on top of stack
(k) func4 returns to main removing arguments to func4
(l) main returns to OS

Virtualization

Locate the address space of a process:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
   printf("location of code : %p\n", (void*) main);
   printf("location of heap : %p\n", (void*) malloc(1));
   int x = 3;
   printf("location of stack : %p\n", (void*) &x);
   return x;
}
```

- All of the above addresses are reported as virtual - not physical

Goals:

- **Transparency**: OS needs to set virtual addresses so that process thinks it owns the machine
- **Efficiency**: OS needs to do this with minimum overhead (time & space)
- **Protection**: OS needs to prevent unauthorized access of memory

Address Translation

- **The hardware changes virtual addresses to physical addresses**
  - during instruction fetch, load, store
  - translation occurs on every memory reference
  - registers are provided to save a *base* address and a *bounds* limit
  - the physical address is the addition of the *base* to the virtual
  - efficient since this is done directly in hardware

- **OS manages the translation**
  - keeps track of which locations are free and which are owned by whom
  - process thinks it has its own private memory
  - but multiple processes share the same physical memory at the same time
  - only intervenes when a process is created, terminated, and switched

- **Example**:

```
int main () { int x = 3; }

<main>:
  push %ebp                <- push base pointer
  mov  %esp,%ebp           <- base pointer becomes stack address
  sub  $0x10,%esp          <- subtract 16 from stack pointer
  movl $0x3,-0x4(%ebp)     <- put 3 into 4 byte space atop base
  leave
  ret
```

Static Relocation

- **Address translation is done *before* execution of code**

- Hardware instruction cycle:

```
loop {
    w = M[instr_ctr]; /* fetch instruction */
    oc = Opcode(w);
    adr = Address(w);
    instr_ctr += 1;
    switch (oc) {
      case 1:  reg += M[adr];             /* add */
      case 2:  M[adr] = reg;              /* store */
      case 3:  instr_ctr = adr;           /* branch */
      .  .  .
    }
}
```

Dynamic Relocation

- **Address translation is done *during* execution of code**
  - Relationship between physical address (pa) and virtual address (va):

```
pa = Name_Location_map(va)
```

  - Hardware instruction cycle:

```
loop {
    w = M[NL_map(instr_ctr)]; /* fetch instruction */
    oc = Opcode(w);
    adr = Address(w);
    instr_ctr += 1;
    switch (oc) {
      case 1:  reg += M[NL_map(adr)];      /* add */
      case 2:  M[NL_map(adr)] = reg;       /* store */
      case 3:  instr_ctr = NL_map(adr);    /* branch */
      .  .  .
    }
}
```

But what if address spaces are large?

- **Problems**:
  - a large *free* space is wasteful
  - the address space may be larger than the physical space

- **A Possible Solution**:
  - *segment* the address space
  - a segment is a fixed-size contiguous portion of the address space
  - employ base and bounds registers for each segment
  - OS is free to decide where each segment will be placed in physical memory

- **Segments**:
  - at least a base-bounds pair for each of code, heap, stack
  - *segmentation fault* occurs if access is attempted outside the segment
  - *explicit segmentation*: top 2 bits identify segment, rest the offset

- **Example**:
  segments: code=(32K,2K), heap=(34K,2K), stack=(24K,2K) - (base,bound)
  suppose virtual address of heap starts at 4K
  heap address 4200 translates to 4200-4096+34816 = 34920

Free Space

- **Segmentation**:
  - fixed size segments of contiguous memory are easy to manage
    use a simple table mapping owner/virtual address to physical address
  - but most segments are not fixed size - thus...
  - request for non-existing free segment size may be made - thus...
  - compaction is difficult as a segment cannot be "moved" when owned by a user until the user returns it with `free`

- **A Possible Solution**:
  - split and coalesce
  - maintain linked list of free segment specs
  - concept example:

    ```
    head->[addr:0,len:10]->[addr:20,len:10]->NULL
    ```

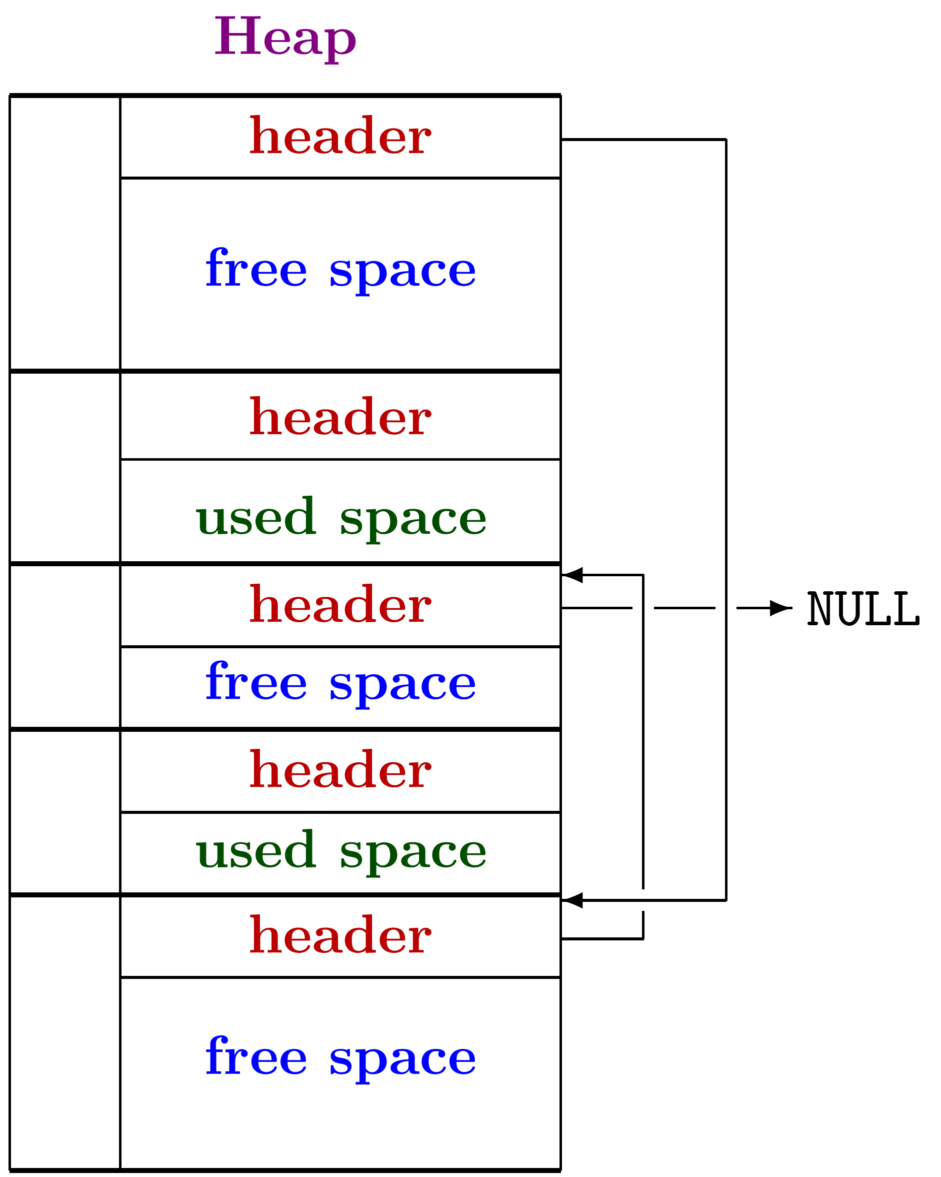
    split: request 3 bytes -

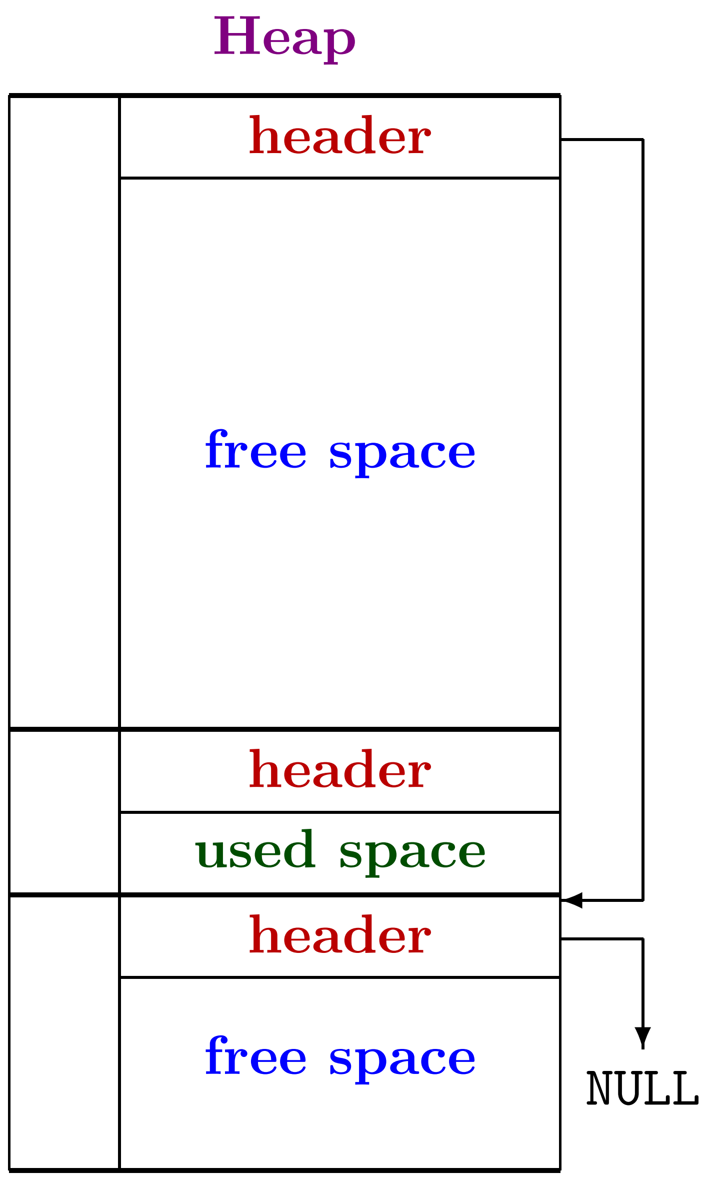
    ```
    head->[addr:0,len:10]->[addr:23,len:7]->NULL
    ```

    coalesce: return 10 bytes from address 10 -

    ```
    head->[addr:0,len:20]->[addr:23,len:7]->NULL
    ```
  - actually, free space is included in the nodes, headers also have extra information such as a magic number for sanity check

Heap Management





Suppose topmost used space on the left is returned to the OS
Then an optimal result for the heap is as shown on the right

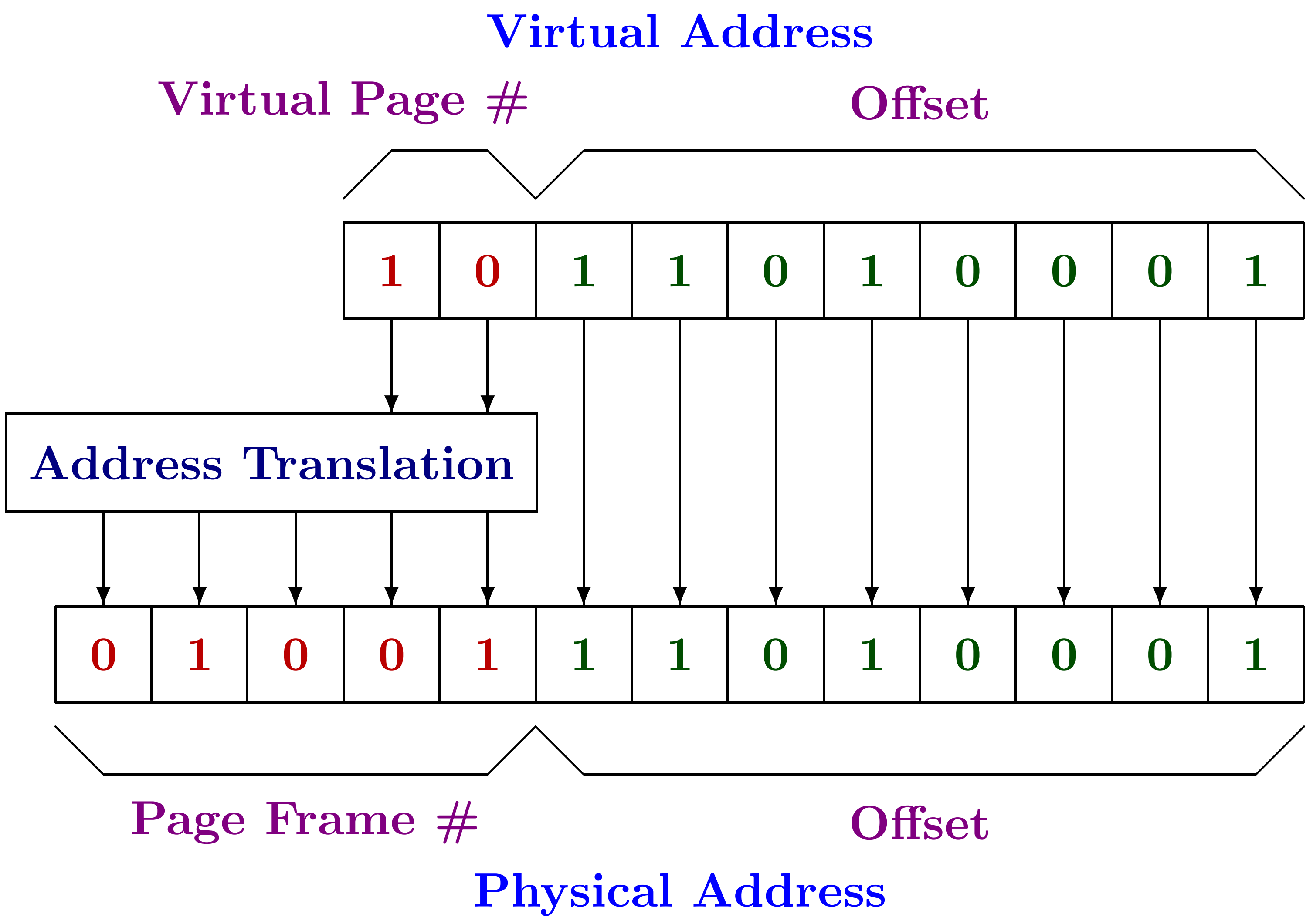
Heuristics for Allocating Space

- **Best Fit**
  - Of all contiguous chunks big enough to satisfy the request return the memory from the smallest chunk
  - The block to return is found in one pass through the free list
  - In theory: attempts to reduce wasted space
- **Worst Fit**
  - Return the memory from the chunk with the most wasted space
  - Empirically bad performance: high overhead, high fragmentation
- **First Fit**
  - Return the memory from the first chunk that satisfies the request
  - Much less overhead: no need to search the entire free list
  - But then the beginning of the free list has lots of small pieces which may inhibit optimal defragmentation
- **Next Fit**
  - Like First Fit except begin looking from the last allocated block
  - Overhead is similar to First Fit
  - Prevents "splintering" of the beginning of the free list

Memory Pages

- **Rationale**:
  - Since de-fragmentation is easy with fixed sized segments, use memory units that are all the same size
  - These units might be 4K in size (12 bits - offset)
- **Properties**:
  - Both physical and virtual address space are paged
  - A page of physical memory is called a page frame
  - The OS supports the abstraction of an address space effectively, regardless of how processes use the address space
  - No assumptions about how the heap and stack grow and how they are used need be made
  - An address space may have many pages and they do not have to be in address-increasing order
- **How it works**:
  - There is a per-process page table in memory that maps address space pages to physical space frames
  - The page table address is in a page table base address register (PTBR)

Paging Translation



Page Table Entry

31 … 12	11 … 9	8	7	6	5	4	3	2	1	0
Page Frame #	Avail	G	PAT	D	A	PCD	PWT	U/S	R/W	P

P	- Present bit: is this page in memory (1) or swapped out (0)
R/W	- Read/write: read only (0) or read/write (1)
U/S	- User (1) or superuser (0) owned
PWT	- Page level write-through: write to memory and cache together
PCD	- Page can (0) cannot (1) be cached (page cache disable)
A	- Accessed bit: set to 1 by the CPU on any write or read
D	- Dirty bit: set to 1 by the CPU on any write
PAT	- Page Attribute Table: 4KB pages (0) or 4MB pages (1)
G	- Global: this (frequently accessed) PTE kept under task switch
Avail	- Available to the user

Example Page Table Entry for some x86 processors

http://books.google.com/books?id=aJFVCnwNbMEC&pg=PA98&lpg=PA98&dq=page+table+attribute+bit&source=bl&ots=iTYGBYSSMB&sig=bVZXa_Ivvs_Xup1XDadYu3aZVHU&hl=en&sa=X&ei=A2JtUrDSEo_IkAfqooHoDg&ved=0CCgQ6AEwADgK#v=onepage&q=page%20table%20attribute%20bit&f=false

Address Translation Computation

```
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE))
// Fetch the PTE
PTE = AccessMemory(PTEAddr)
// Check if process can access the page
if (PTE.Present == False) // Page frame exists in memory?
    RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.User, PTE.R/W) == False)
    RaiseException(PROTECTION_FAULT)
else
    // Access is OK: form physical address and fetch it
    offset = VirtualAddress & OFFSET_MASK
PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
Register = AccessMemory(PhysAddr)
```

PTBR:	Page table base register
VPN:	Virtual page number
PFN:	Physical frame number

Address Translation Computation

```
physical_address NL_map (virtual_page p, offset o) {
    return PFN(M[PTBR + p*PTE_size]) + o;
}
```

where `M[PTBR + p*PTE_size]` is a page table entry and `PFN()` is a function that extracts the physical frame number from the input page table entry, shifts it to the left by 12 bits, and returns the result.

Translation Look-aside Buffer

- **Rationale**:
  - Paging can be slow due to the complexity of memory access
  - Putting some of this in hardware (MMU) can speed things up
- **Outline**:
  - TLB is a cache of popular virtual-to-physical address translations
  - On a virtual memory access, the hardware looks at the TLB first
  - **Spatial locality**: if memory location x is accessed then it is likely that a location near x will be accessed soon
  - **Temporal locality**: if memory location is accessed, it is likely that it will be accessed again soon (e.g. loop instructions)
  - Implemented as a hash table for fast access
  - May have 128 entries (use `x86info -c` to find out)
  - Only applies to the running process
  - Which to evict on a miss? Least-recently-used?
- **Performance**:
  - The performance boost is so fantastic that TLBs actually make virtual memory possible
  - TLB lookup is about 5 ns, memory lookup is about 60 ns
  - Hit ratio is commonly 80-98% depending on size of TLB

Smaller Paging Tables

- **Problem**:
  - Page tables are potentially large
  - Consider: 32 bit addresses, 4KB pages

    Address space has ($2^{32}/2^{12} = 2^{20}$) pages; PT size = 4MB

    100 processes could require 400MB of memory (see /proc/meminfo)

- **Solutions**:
  - Use larger page sizes (run `pagesize -a`)

    Increases internal fragmentation
  - Multi-level page tables

    a page table is partitioned into pages (page directory)

    if all PTEs of a page are invalid then do not allocate that page

    use a page directory to refer to cached page tables

    register `CR3` contains the page directory base address

    upper 10 bits of virtual address indexes into page directory

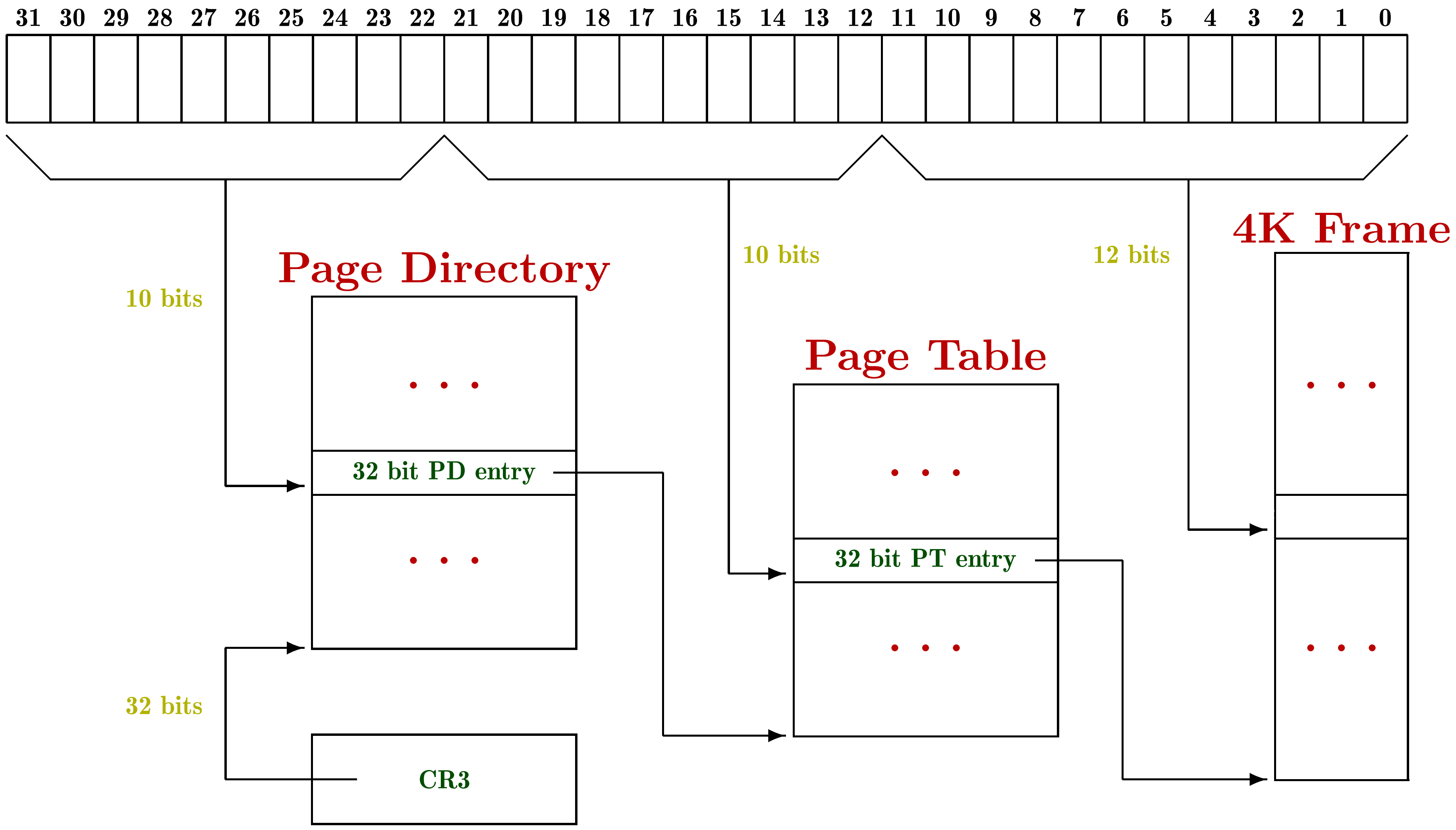
    the value returned is the base of some page table

    next 10 bits of virtual address indexes into that page table

    result is the physical frame number

    last 12 bits of virtual address is the frame offset

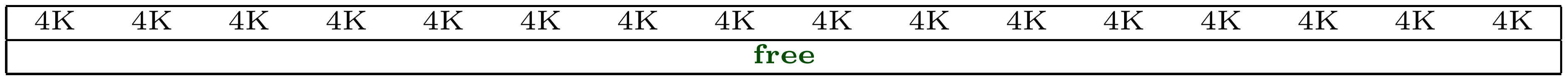
Multi-level Page Table



- space is allocated for page tables in proportion to the amount used by running processes - empty parts of page table disappear
- the page table need not occupy contiguous memory - only the frames need to be contiguous (large chunks are not allocated - this is more feasible)

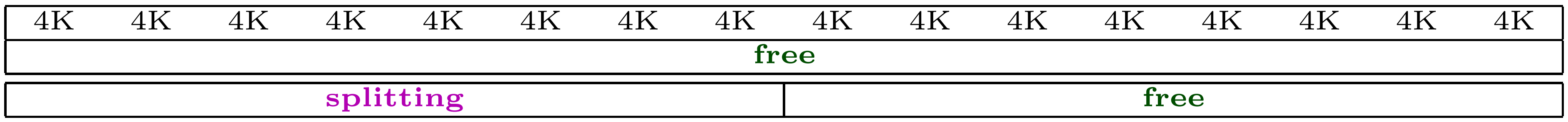
Buddy Memory Allocation

Start with completely free heap space



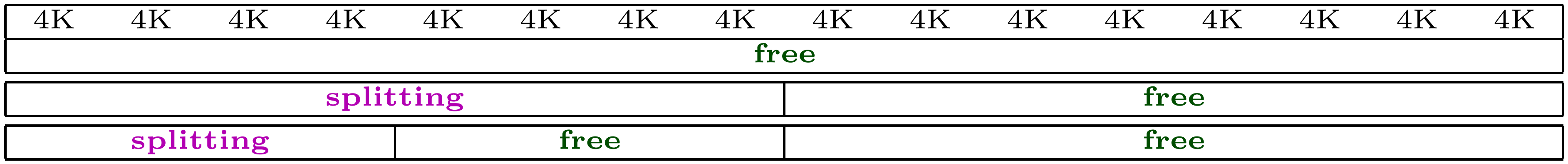
Buddy Memory Allocation

Start with completely free heap space
Satisfy request for 5.2K of space, program A
Split into two chunks of equal size



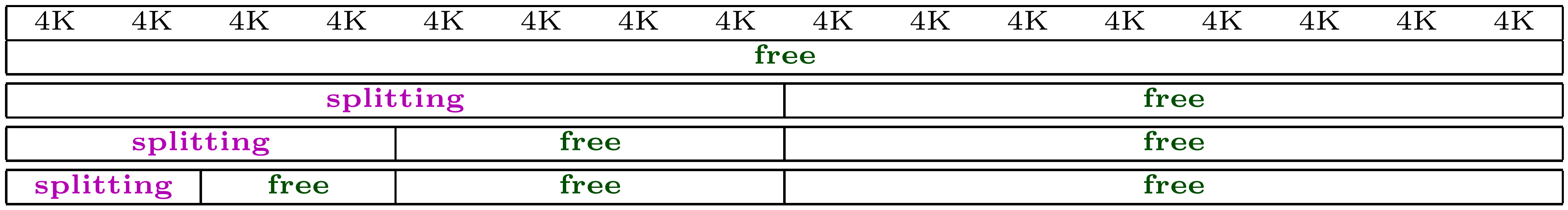
Buddy Memory Allocation

Start with completely free heap space
Satisfy request for 5.2K of space, program A
Can split again into two chunks of equal size



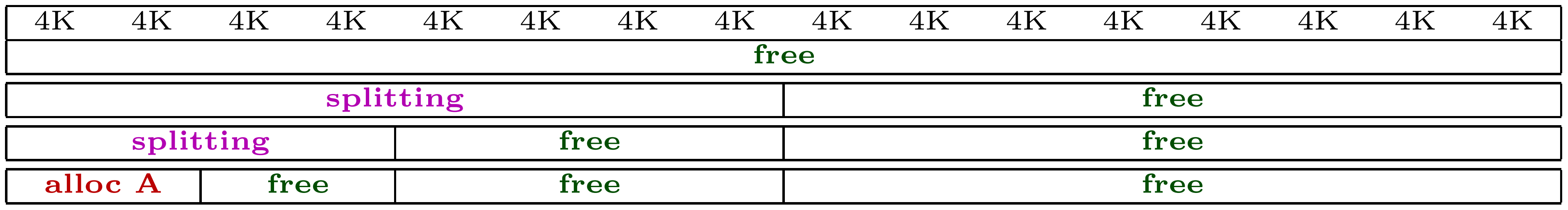
Buddy Memory Allocation

Start with completely free heap space
Satisfy request for 5.2K of space, program A
Cannot split again



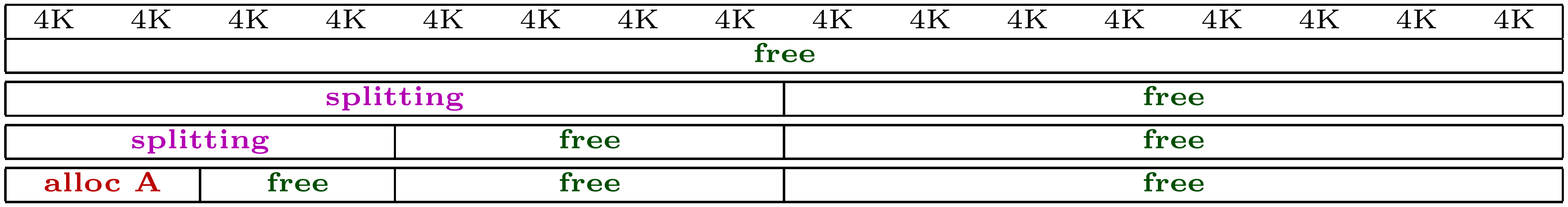
Buddy Memory Allocation

Start with completely free heap space
Satisfy request for 5.2K of space, program A
Cannot split again so allocate a piece of the heap



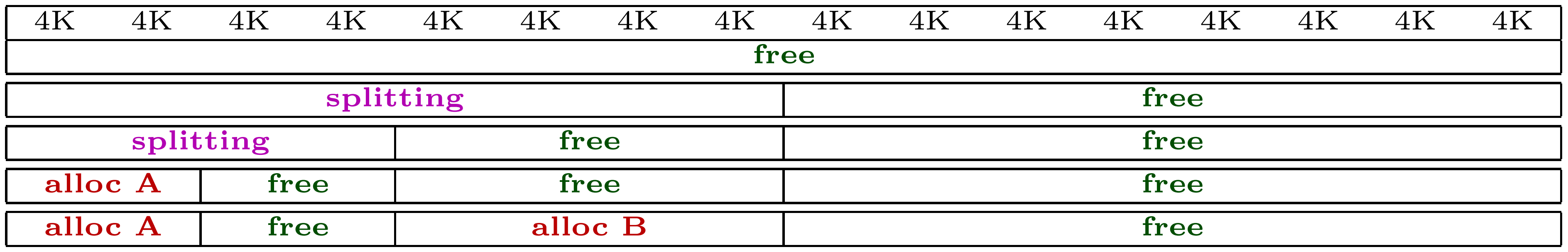
Buddy Memory Allocation

Start with completely free heap space
Satisfy new request for 14.4K of space, program B



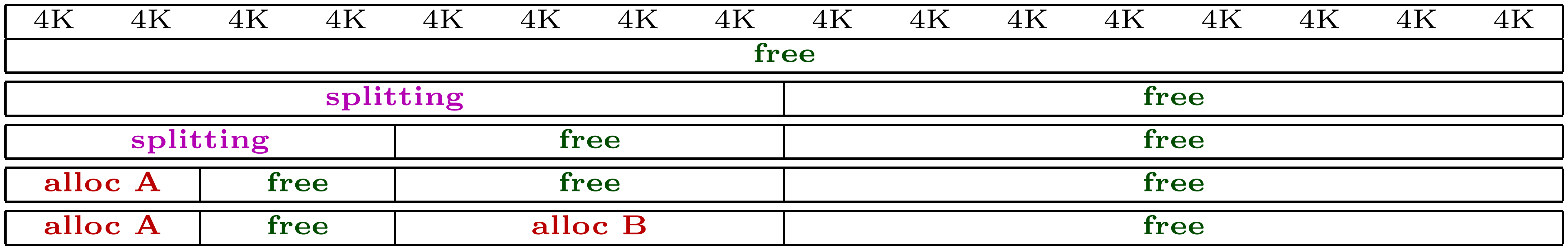
Buddy Memory Allocation

Start with completely free heap space
Satisfy new request for 14.4K of space, program B
A block of size 16K is available, allocate it



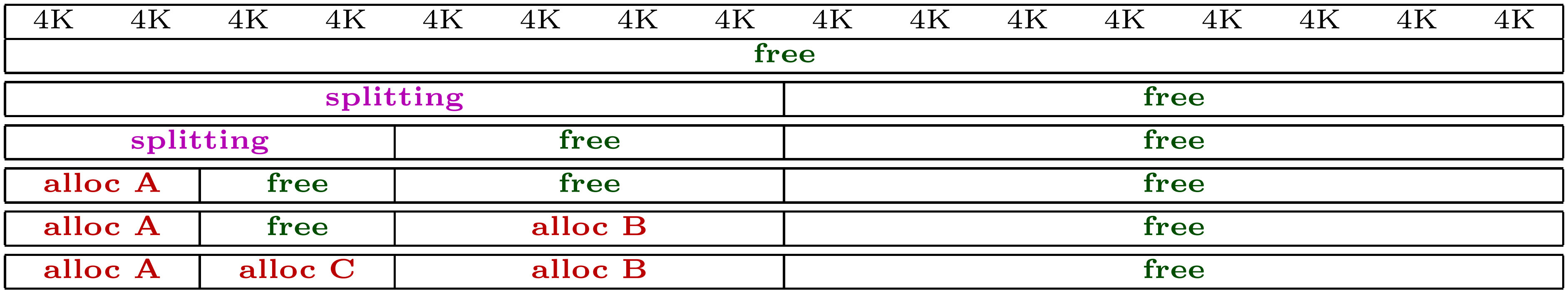
Buddy Memory Allocation

Start with completely free heap space
Satisfy new request for 4.4K of space, program C



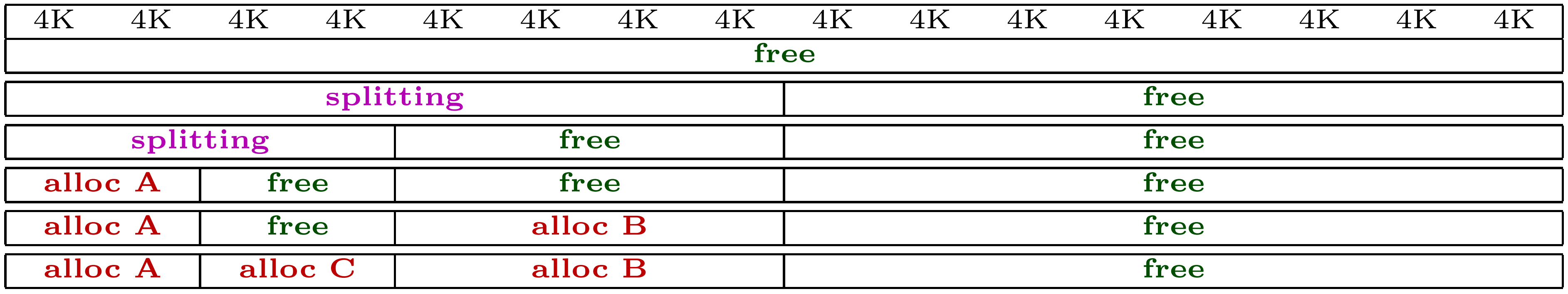
Buddy Memory Allocation

Start with completely free heap space
Satisfy new request for 4.4K of space, program C
A block of size 8K is available, allocate it


4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K
free
splitting | free
splitting | free | free
alloc A | free | free | free
alloc A | free | alloc B | free
alloc A | alloc C | alloc B | free

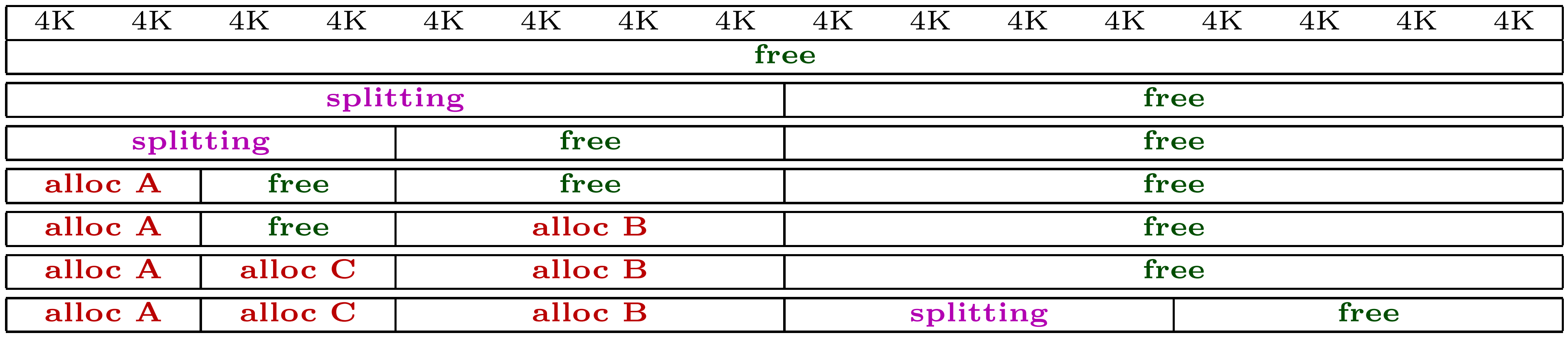
Buddy Memory Allocation

Start with completely free heap space
Satisfy new request for 4.4K of space, program D



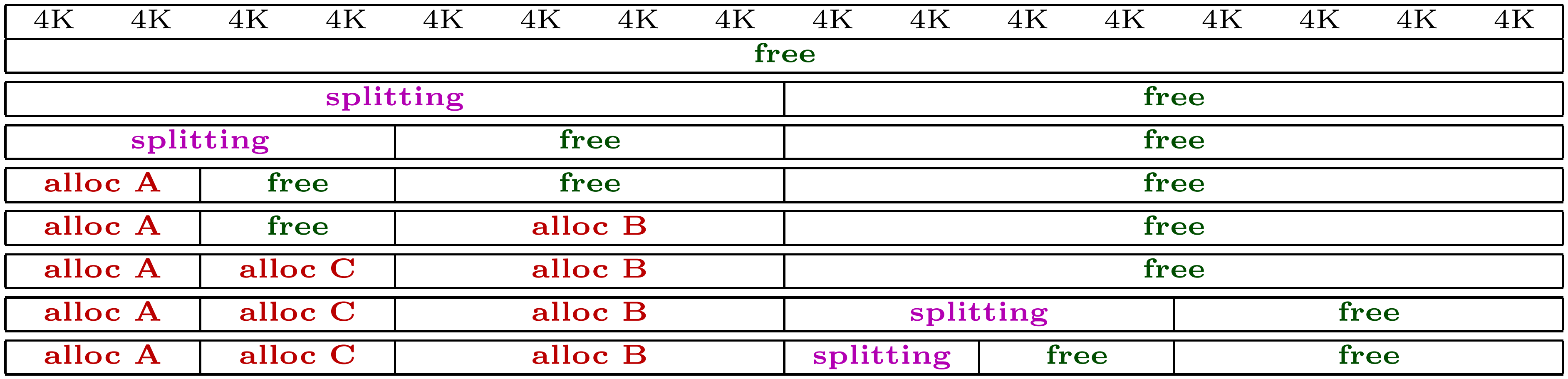
Buddy Memory Allocation

Start with completely free heap space
Satisfy new request for 4.4K of space, program D
Split smallest available chunk


4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K
free
splitting | free
splitting | free | free
alloc A | free | free | free
alloc A | free | alloc B | free
alloc A | alloc C | alloc B | free
alloc A | alloc C | alloc B | splitting | free

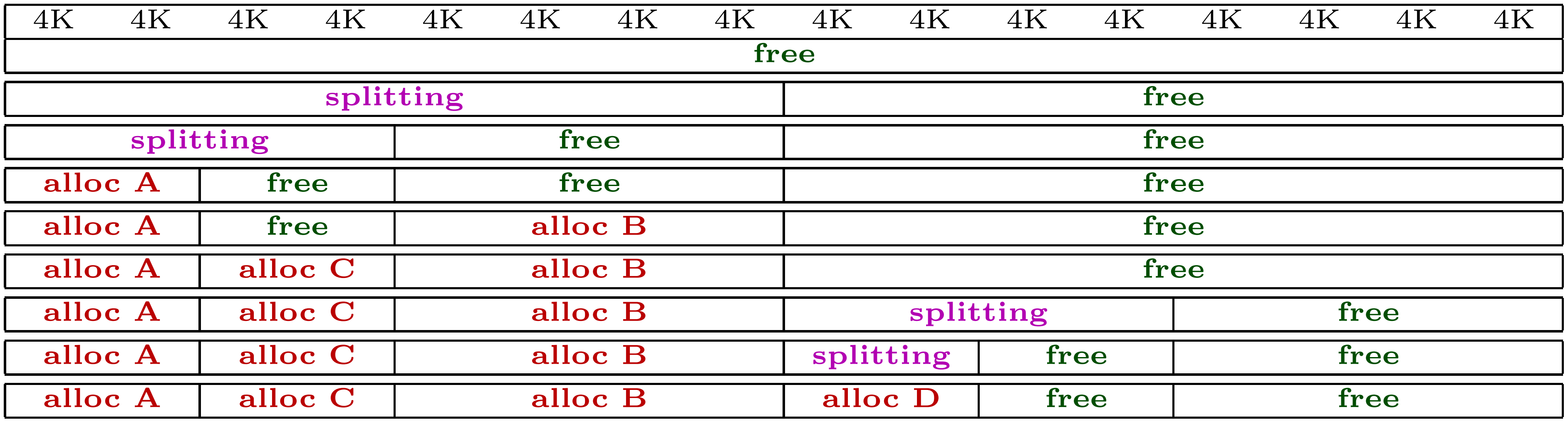
Buddy Memory Allocation

Start with completely free heap space
Satisfy new request for 4.4K of space, program D
Split again


4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K
free
splitting | free
splitting | free | free
alloc A | free | free | free
alloc A | free | alloc B | free
alloc A | alloc C | alloc B | free
alloc A | alloc C | alloc B | splitting | free
alloc A | alloc C | alloc B | splitting | free | free

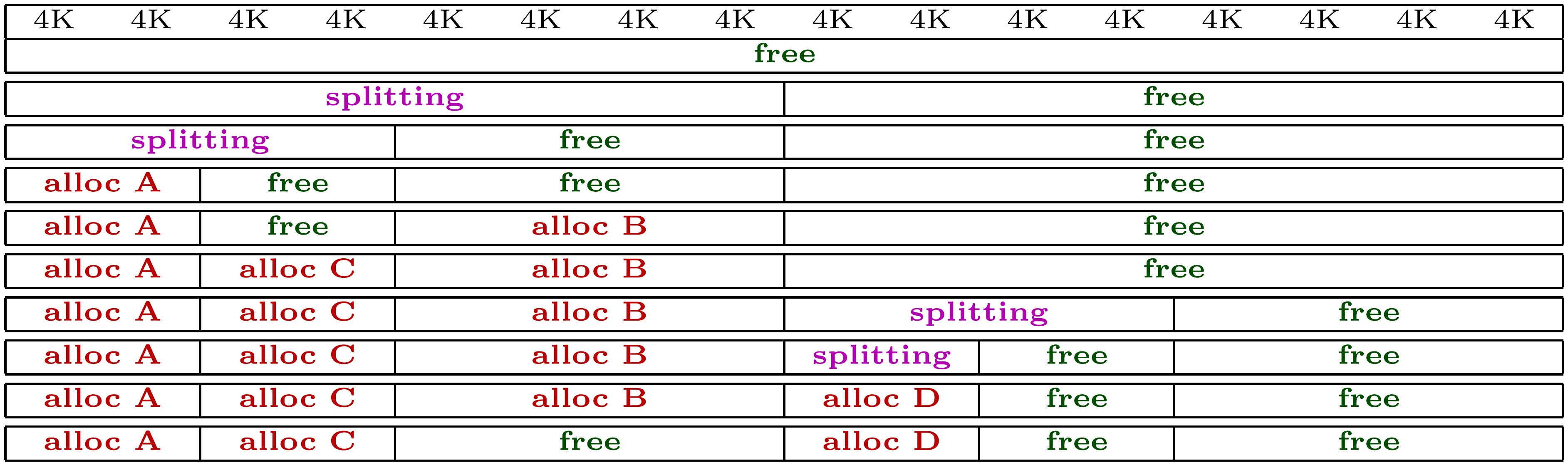
Buddy Memory Allocation

Start with completely free heap space
Satisfy new request for 4.4K of space, program D
Allocate to D


4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K
free
splitting | free
splitting | free | free
alloc A | free | free | free
alloc A | free | alloc B | free
alloc A | alloc C | alloc B | free
alloc A | alloc C | alloc B | splitting | free
alloc A | alloc C | alloc B | splitting | free | free
alloc A | alloc C | alloc B | alloc D | free | free

Buddy Memory Allocation

Start with completely free heap space
B returns space to OS


4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K
free
splitting | free
splitting | free | free
alloc A | free | free | free
alloc A | free | alloc B | free
alloc A | alloc C | alloc B | free
alloc A | alloc C | alloc B | splitting | free
alloc A | alloc C | alloc B | splitting | free | free
alloc A | alloc C | alloc B | alloc D | free | free
alloc A | alloc C | free | alloc D | free | free

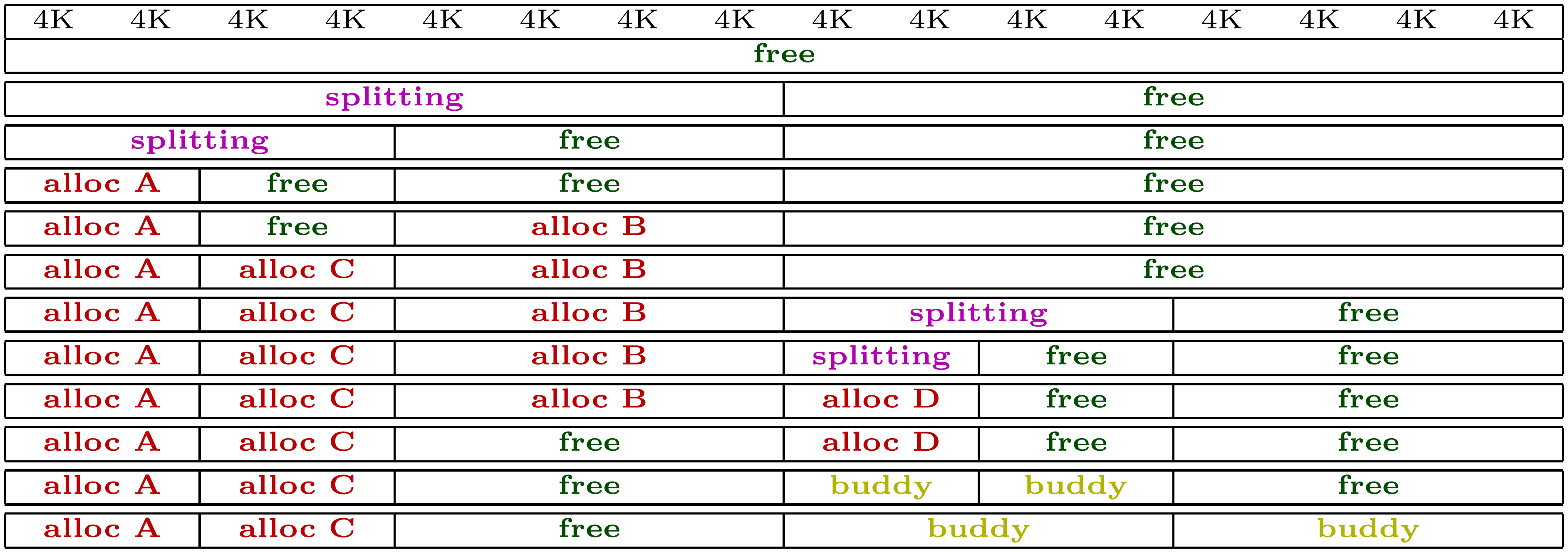
Buddy Memory Allocation

Start with completely free heap space
D returns space to OS
Its buddy is free, the two are coalesced

4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K
free															
splitting								**free**							
splitting				**free**				**free**							
alloc A		**free**		**free**				**free**							
alloc A		**free**		**alloc B**				**free**							
alloc A		**alloc C**		**alloc B**				**free**							
alloc A		**alloc C**		**alloc B**				**splitting**				**free**			
alloc A		**alloc C**		**alloc B**				**splitting**		**free**		**free**			
alloc A		**alloc C**		**alloc B**				**alloc D**		**free**		**free**			
alloc A		**alloc C**		**free**				**alloc D**		**free**		**free**			
alloc A		**alloc C**		**free**				**buddy**		**buddy**		**free**			

Buddy Memory Allocation

Start with completely free heap space
D returns space to OS
Another buddy is free, the two are coalesced


4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K
free
splitting free
splitting free free
alloc A free free free
alloc A free alloc B free
alloc A alloc C alloc B free
alloc A alloc C alloc B splitting free
alloc A alloc C alloc B splitting free free
alloc A alloc C alloc B alloc D free free
alloc A alloc C free alloc D free free
alloc A alloc C free buddy buddy free
alloc A alloc C free buddy buddy

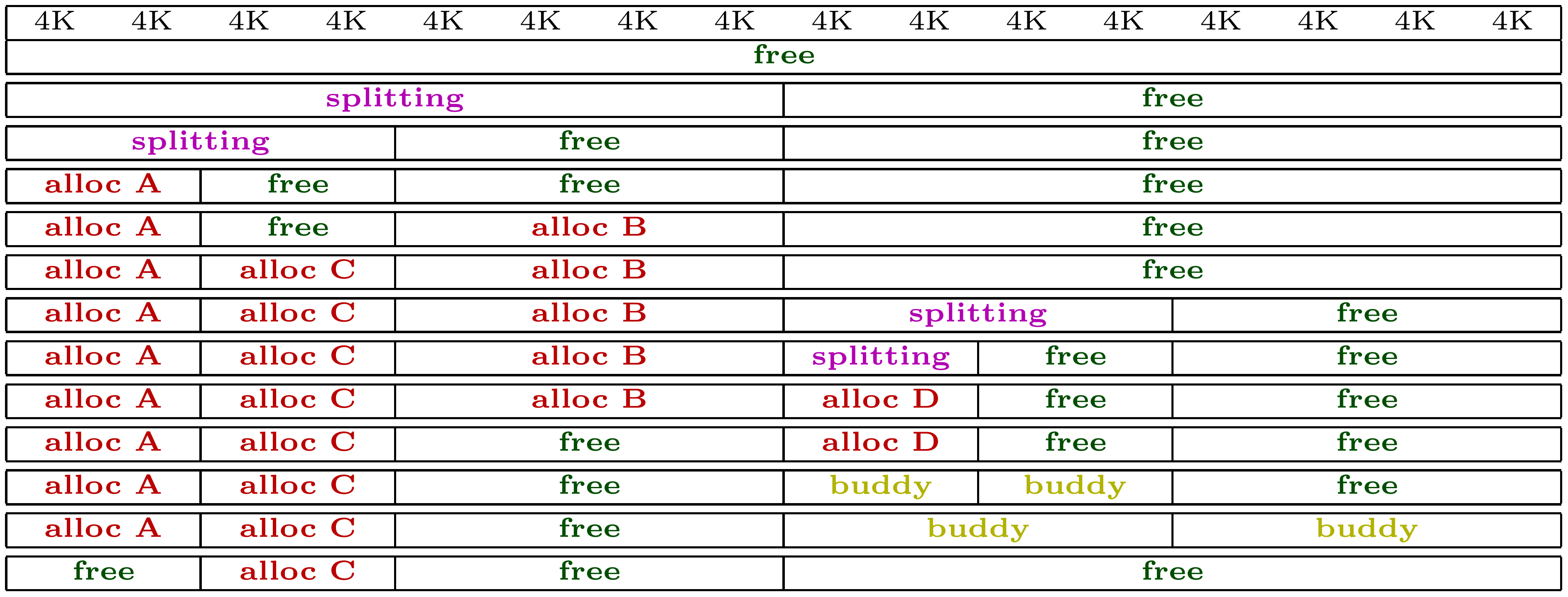
Buddy Memory Allocation

Start with completely free heap space
D returns space to OS

4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K
free															
splitting								free							
splitting				free				free							
alloc A		free		free				free							
alloc A		free		alloc B				free							
alloc A		alloc C		alloc B				free							
alloc A		alloc C		alloc B				splitting				free			
alloc A		alloc C		alloc B				splitting		free		free			
alloc A		alloc C		alloc B				alloc D		free		free			
alloc A		alloc C		free				alloc D		free		free			
alloc A		alloc C		free				buddy		buddy		free			
alloc A		alloc C		free				buddy				buddy			
alloc A		alloc C		free				free							

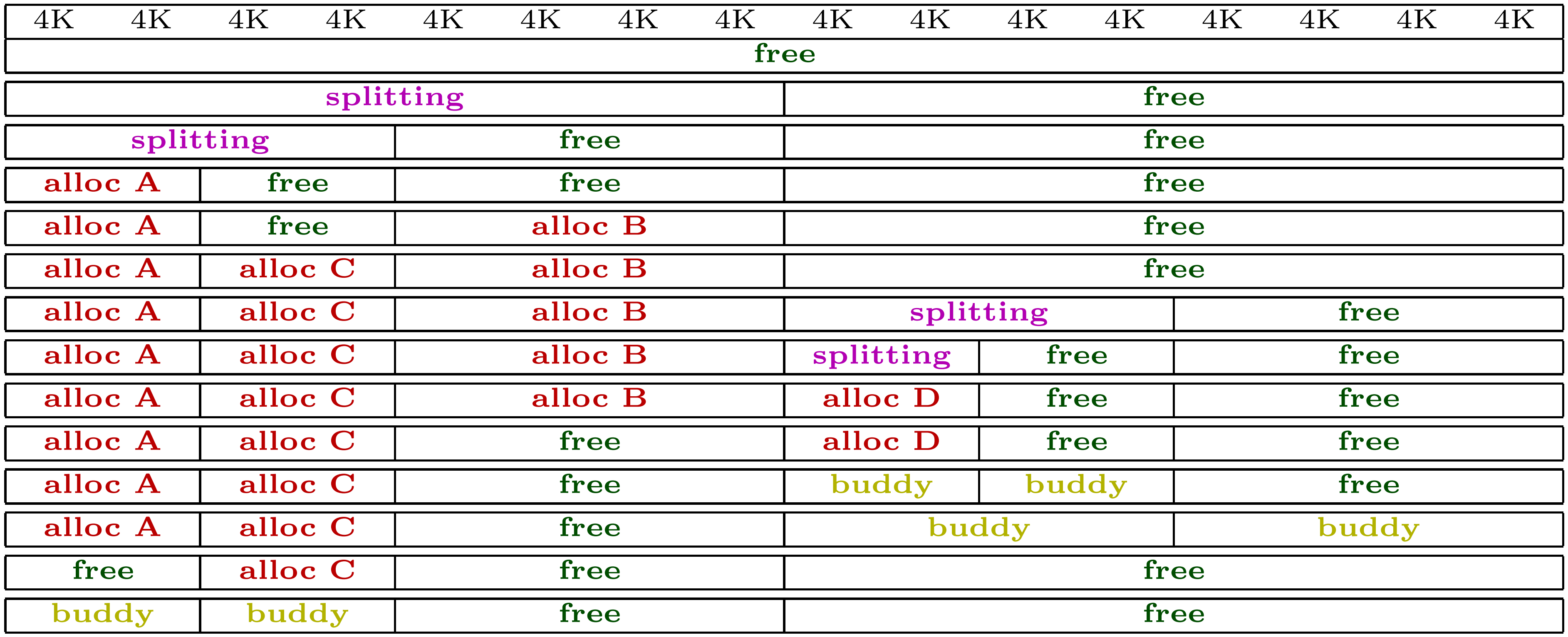
Buddy Memory Allocation

Start with completely free heap space
A returns space to OS


4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K
free
splitting | free
splitting | free | free
alloc A | free | free | free
alloc A | free | alloc B | free
alloc A | alloc C | alloc B | free
alloc A | alloc C | alloc B | splitting | free
alloc A | alloc C | alloc B | splitting | free | free
alloc A | alloc C | alloc B | alloc D | free | free
alloc A | alloc C | free | alloc D | free | free
alloc A | alloc C | free | buddy | buddy | free
alloc A | alloc C | free | buddy | buddy
free | alloc C | free | free

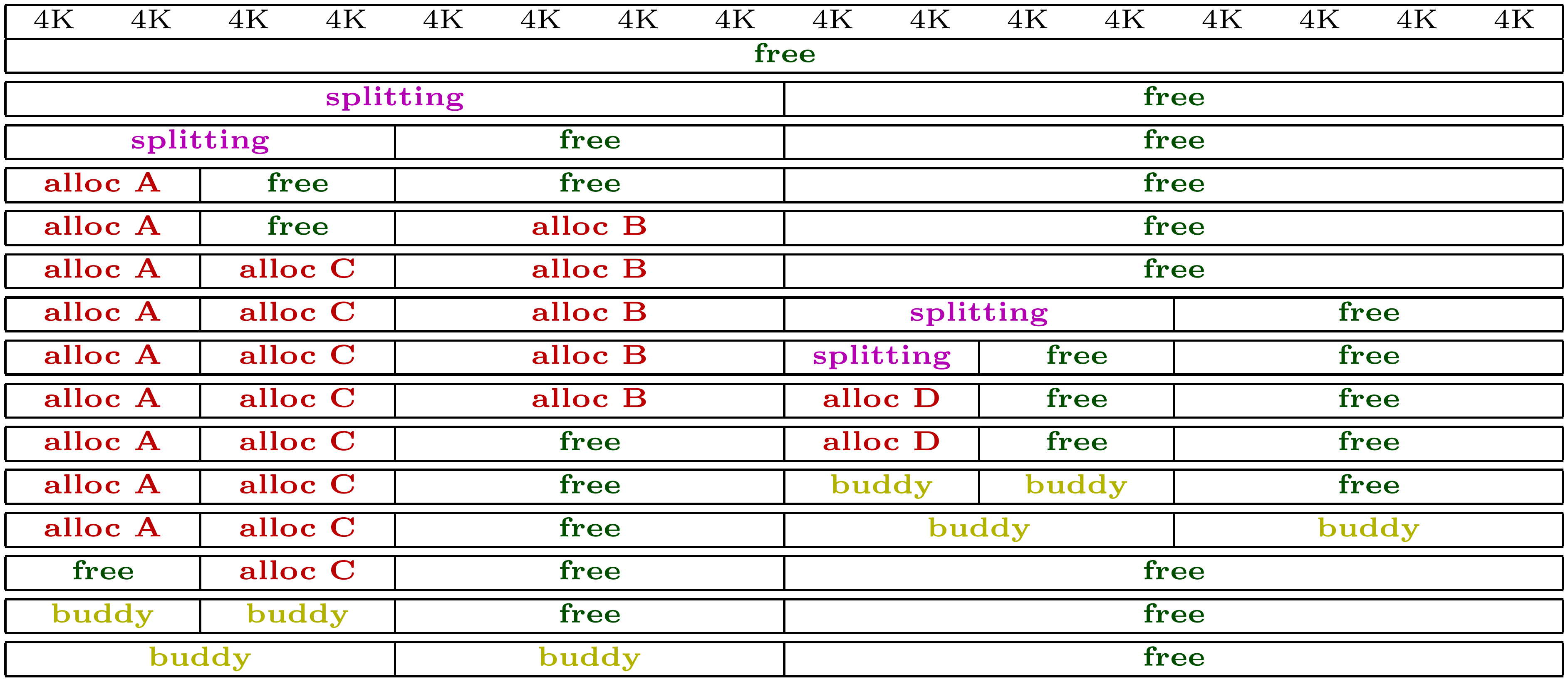
Buddy Memory Allocation

Start with completely free heap space
C returns space to OS


4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K
free
splitting | free
splitting | free | free
alloc A | free | free | free
alloc A | free | alloc B | free
alloc A | alloc C | alloc B | free
alloc A | alloc C | alloc B | splitting | free
alloc A | alloc C | alloc B | splitting | free | free
alloc A | alloc C | alloc B | alloc D | free | free
alloc A | alloc C | free | alloc D | free | free
alloc A | alloc C | free | buddy | buddy | free
alloc A | alloc C | free | buddy | buddy
free | alloc C | free | free
buddy | buddy | free | free

Buddy Memory Allocation

Start with completely free heap space
C returns space to OS
Two buddies are coalesced


4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K
free
splitting | free
splitting | free | free
alloc A | free | free | free
alloc A | free | alloc B | free
alloc A | alloc C | alloc B | free
alloc A | alloc C | alloc B | splitting | free
alloc A | alloc C | alloc B | splitting | free | free
alloc A | alloc C | alloc B | alloc D | free | free
alloc A | alloc C | free | alloc D | free | free
alloc A | alloc C | free | buddy | buddy | free
alloc A | alloc C | free | buddy | buddy
free | alloc C | free | free
buddy | buddy | free | free
buddy | buddy | free

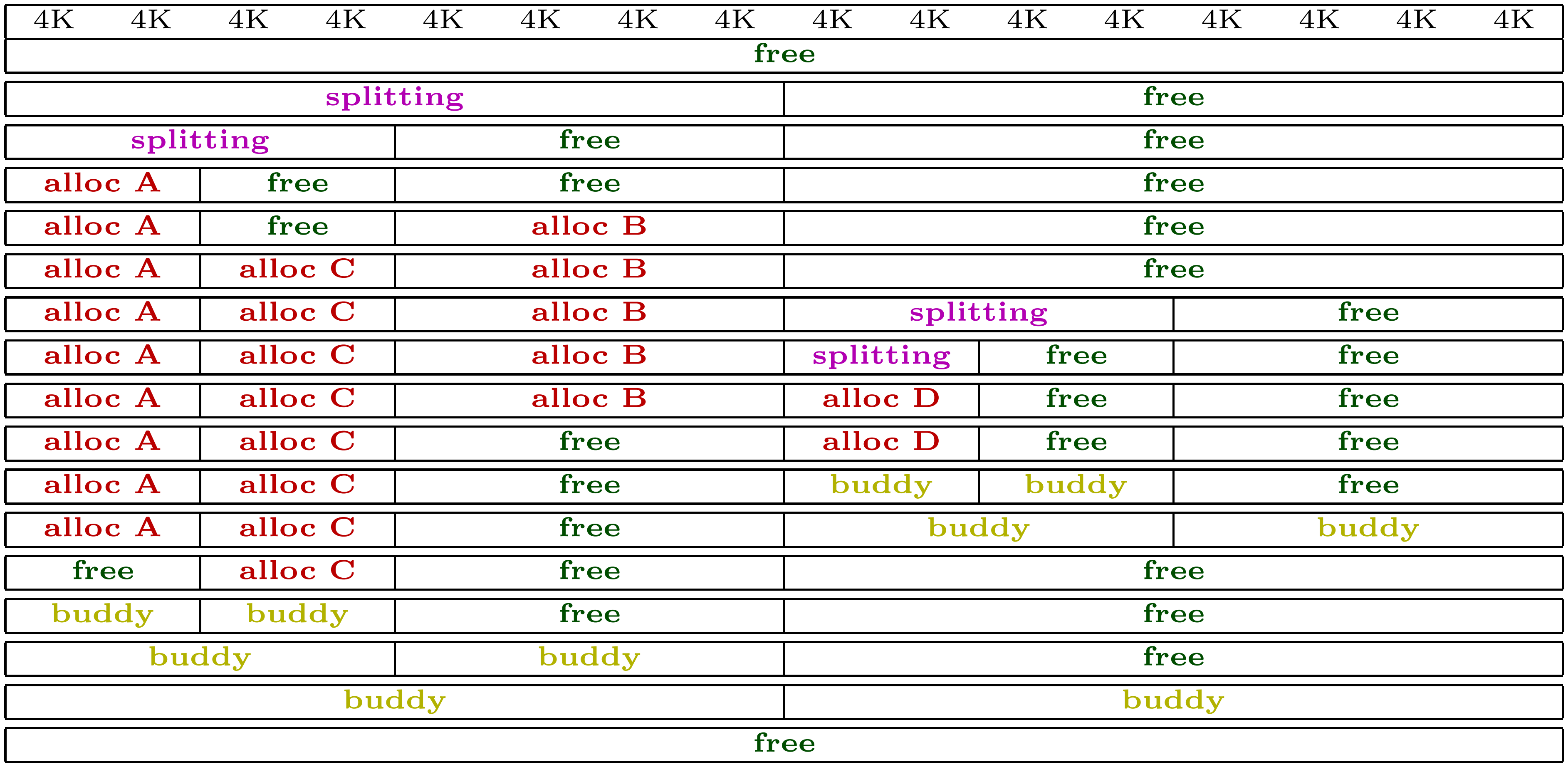
Buddy Memory Allocation

Start with completely free heap space
C returns space to OS
Two more buddies are coalesced

4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K	4K
free															
splitting								free							
splitting				free				free							
alloc A		free		free				free							
alloc A		free		alloc B				free							
alloc A		alloc C		alloc B				free							
alloc A		alloc C		alloc B				splitting				free			
alloc A		alloc C		alloc B				splitting		free		free			
alloc A		alloc C		alloc B				alloc D		free		free			
alloc A		alloc C		free				alloc D		free		free			
alloc A		alloc C		free				buddy		buddy		free			
alloc A		alloc C		free				buddy				buddy			
free		alloc C		free				free							
buddy		buddy		free				free							
buddy				buddy				free							
buddy								buddy							

Buddy Memory Allocation

Start with completely free heap space
C returns space to OS
Two more buddies are coalesced


4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K 4K
free
splitting | free
splitting | free | free
alloc A | free | free | free
alloc A | free | alloc B | free
alloc A | alloc C | alloc B | free
alloc A | alloc C | alloc B | splitting | free
alloc A | alloc C | alloc B | splitting | free | free
alloc A | alloc C | alloc B | alloc D | free | free
alloc A | alloc C | free | alloc D | free | free
alloc A | alloc C | free | buddy | buddy | free
alloc A | alloc C | free | buddy | buddy
free | alloc C | free | free
buddy | buddy | free | free
buddy | buddy | free
buddy | buddy
free

Buddy Memory Allocator

- **Properties**:
  - Worst case allocation/deallocation is slow: coalescence is too eager although the number of such ops is $\log_2 n$ where n is # primitive blocks
  - External fragmentation is low: small allocation holes are not persistent
  - High internal fragmentation: lots of memory allocated that is not needed

    a 66K request may be satisfied with 128K memory!
  - Compaction in the classical sense is not necessary
  - Small data structures are used to keep track of which buddies exist
  - Can be used as a course-grain allocator on top of a Slab allocator
  - Used in linux

    Try `cat /proc/buddyinfo`

Slab Allocator

- **Rationale**:
  - Some data objects always have the same size (examples: semaphores)
  - Preallocate a number of open slots of each type object in a cache, find an open slot in the cache for a desired request

- **Goals**:
  - Make effective use of the buddy system.
    - help eliminate internal fragmentation due to the buddy system.
    - reduce coalescence inefficiency with long slab lifetime
  - Cache commonly used objects and keep them in an initialized state so system does not waste time allocating, initializing, deallocating them.
  - Preserve basic structure of freed objects between uses
  - Better utilize hardware caches by aligning objects to the L1 and L2 caches.

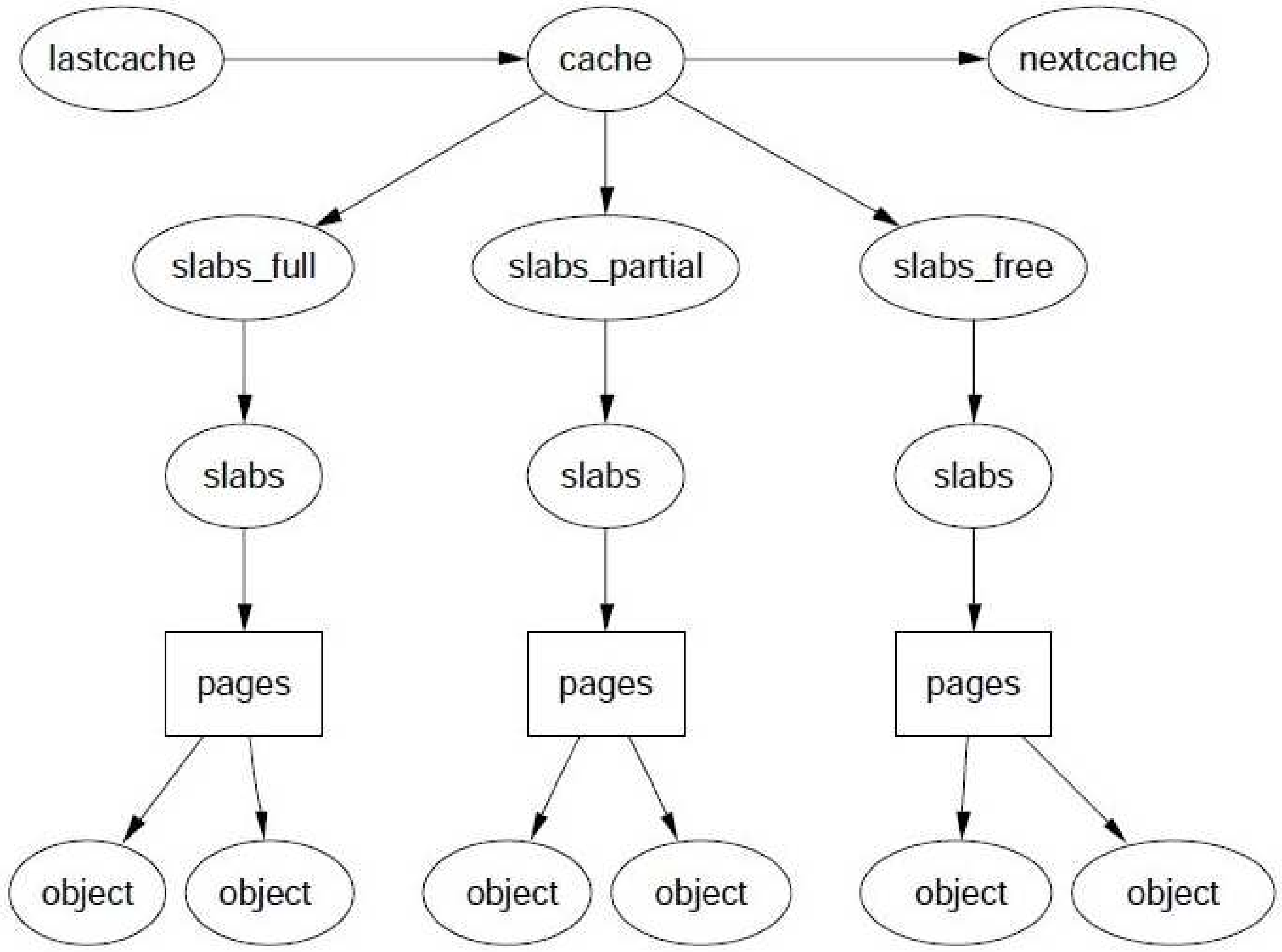
- **Components**:
  - **Slab**: a contiguous piece of memory - one or more frames.
  - **Cache**: A cache consists of one or more slabs.
  - A slab is the amount a cache can grow or shrink by.

Slab Allocator

- **Caches**:
  - Many caches are maintained in doubly linked lists.
  - Some caches are reserved for specific CPUs to avoid having to use spin locks on access.
  - The slabs are carved into small chunks for the data structures and objects the cache manages
  - Each cache has three lists of slabs by type:
    - `slabs_full`: all objects are used
    - `slabs_partial`: some objects are free - may be allocated
    - `slabs_free`: all objects are free - may be destroyed
  - See `/proc/slabinfo` for information on slab usage

- **Operation**:
  - Initially, all slabs are empty and unmarked
  - OS tries to find and return an open slot in an unmarked slab
  - If the last open slot of a slab is taken, that slab becomes marked
  - If no open slots exist, the OS allocates a new slab from contiguous physical memory and assigns it to a cache and returns a slot from it

Slab Allocator



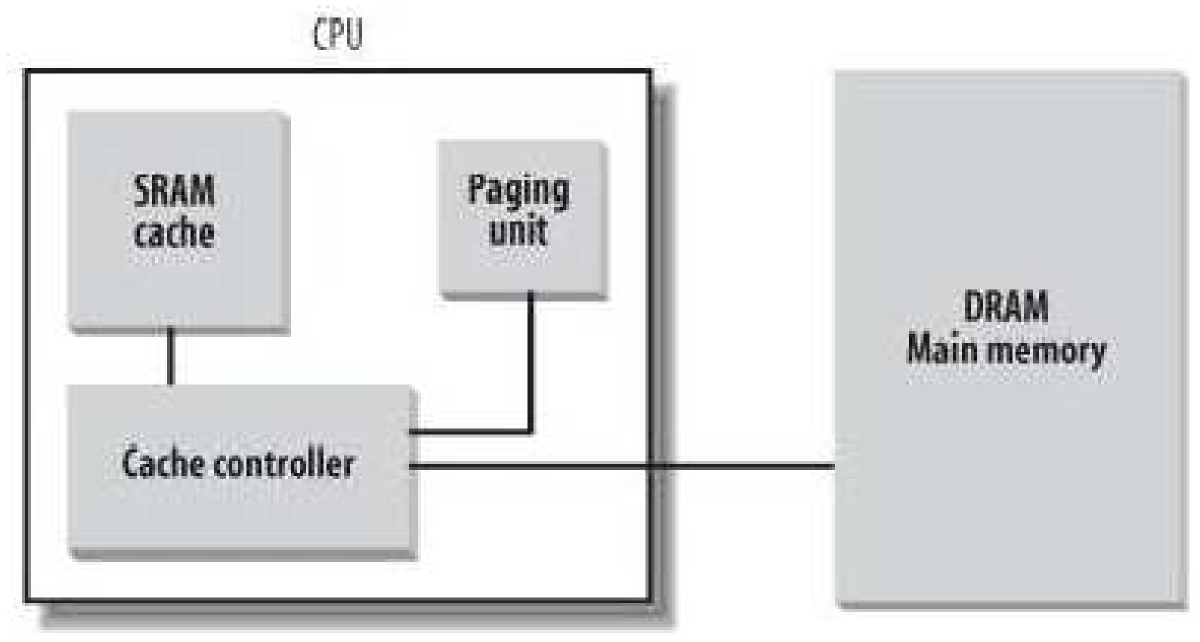
Slab Allocator

- **Cache descriptor**:

```
struct kmem_cache {
   struct list_head slabs_partial;
   struct list_head slabs_full;
   struct list_head slabs_free;
   unsigned long free_objects;  /* idx of first free slot */
   unsigned int free_limit;     /* max number slots */
   size_t colour;               /* number of colors */
   unsigned int colour_off;     /* color offset */
   unsigned int colour_next;    /* cache coloring */
   spinlock_t list_lock;        /* protect members */
   struct array_cache *shared;  /* hand out cache-warm objects from */
   struct array_cache **alien;  /* alloc: pointers to free objects */
   unsigned long next_reap;     /* skip reap if jiffies < next_reap */
   unsigned int gfporder;       /* # pages/slab (power of 2) */
   unsigned int growing;        /* do not deallocate */
   void (*ctor)(void*);         /* constructor - initialize object */
   void (*dtor)(void*);         /* destructor - cleanup object */
   struct list_head next        /* next cache in chain */
};
```

Slab Allocator

- **Hardware cache**:
  - To avoid CPU wait for instruction fetches/execs - not the TLB
  - A *line*: a few dozen bytes - transferred as a burst from DRAM
  - The cache: subdivided into many lines
  - *direct mapping*: main memory line always transferred to same cache line
  - *fully associative*: main memory line can go to any line in cache
  - *N-way*: a main memory line can go to one of N cache lines
  - *SRAM*: stores the lines of main memory
  - *Controller*: array of entries - each has a tag (mml id) and status



Slab Allocator

- **Hardware cache - CPU accesses RAM**:

  - CPU looks at bits of physical address representing a tag

  - Searches tags in the controller for a match

  - On a match (cache hit):

    read op:

      data transferred from cache line to a register

      access to memory is avoided

    write op:

      write-through: controller writes to RAM and cache line

      write-back: only the cache line is updated - controller updates RAM when CPU executes an instruction requiring a flush

  - No match (cache miss):

    a cache line is selected to be written to memory (if dirty)

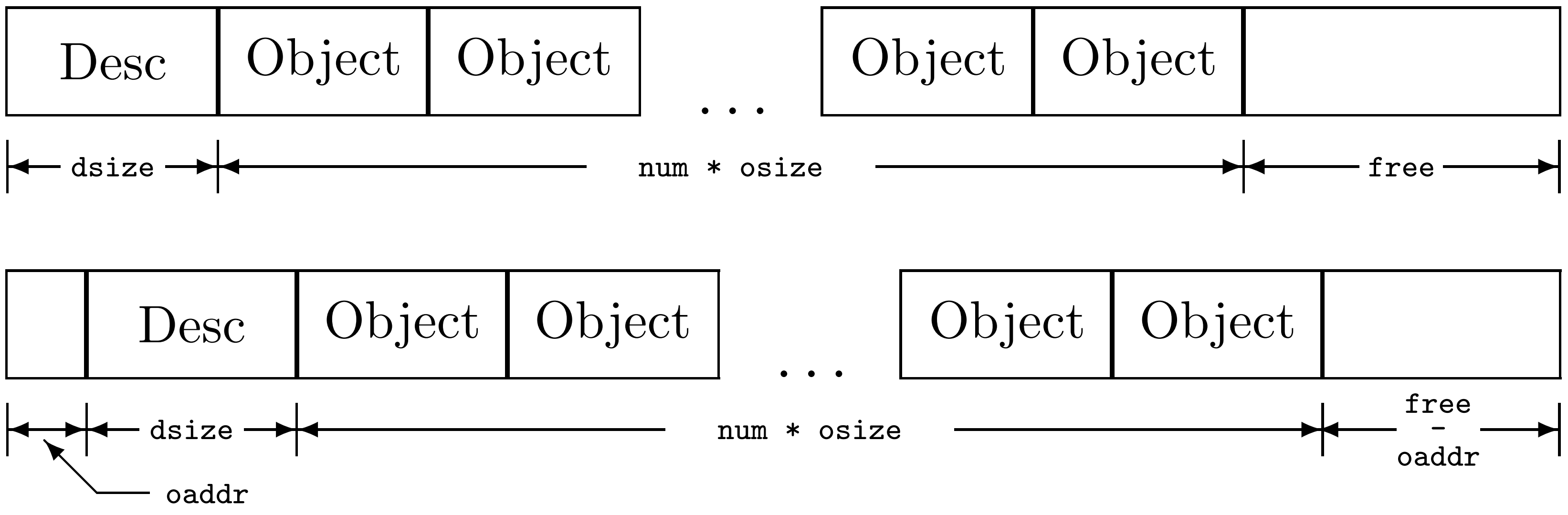
    the line is replaced by the needed one

Slab Allocator

- **Hardware cache - evaluation**:
  - Assume:
    - get 4 bytes at a time from main memory
    - each fetch from main memory takes 20 clock cycles
    - each cache line fetch takes 5 nsec
  - Then:
    - to fill a 32 byte cache line takes 160 clock cycles = 160 nsec
    - to read 4 bytes takes 20 clock cycles = 20 nsec
  - So: for hit rate $r > 90.3\%$, cache times are less
    $r * 5 + (1 - r) * 160 = 20$
  - If memory lines align with DRAM rows, then need only 10 cycles to fetch
  - Crossover is now at hit rate = 80%
    $r * 5 + (1 - r) * 80 = 20$

Slab Allocator

- **Colors**:
  - Same size objects end up stored at the same offset within a cache
  - Objects with same offset from different slabs map to same cache line
  - Colors move free space to front to break up this contention
  - Let `oaddr` be used to evenly divide object address (i.e. align in RAM)
  - Let `osize` be the size of each object in the slab
  - Let `num` be the number of objects in the slab
  - Let `dsize` be the size of the slab descriptor
  - Let `free` be the size of the free space
  - Number of colors = `free/oaddr`
  - Example:



Slab Allocator

- **Slab descriptor**:

```
struct slab {
  union {
    struct {
      struct list_head list;    /* keeps track of where slab is */
      unsigned long colouroff;  /* color offset in the slab  */
      void *s_mem;            /* pointer to 1st object in slab */
      unsigned int inuse;     /* num objects active in slab */
      kmem_bufctl_t free;     /* idx of next free object in slab */
      unsigned short nodeid;
    };
    struct slab_rcu __slab_cover_slab_rcu;
  };
};
```

May be place in the slab or in the “sizes cache” depending on the size of the slab object.

Slub Allocator

- **Rationale**:
  - Drop-in replacement for the Slab allocator
  - Reduces space needed by the Slab allocator to keep track of objects

- **Outline**:
  - There are many Slab object lists
  - For large systems, the Slab lists become exponentially large
  - Gigabytes of Slab overhead is not unheard of
  - Each Slab contains some metadata that makes alignment difficult

    http://www.devx.com/tips/Tip/13265
  - The Slub allocator eliminates the need for many object lists by moving Slab metadata to existing elements of the OS, such as the memory map

- **Benefits**:
  - Is 5-10% faster than Slab allocation
  - Much less memory is used for overhead
  - Now the default allocator for linux

Slob Allocator

- **Rationale**:
  - Drop-in replacement for the Slab allocator
  - Low overhead, useful for small systems, embedded systems
- **Outline**:
  - Uses first fit allocation in a conventional way
  - But emulates Slab allocation, hence can replace it
- **Comparison**:
  - More memory efficient than Slab
  - The code is significantly smaller
  - May save 1/2 MB RAM (remember, this is for small systems)
  - Suffers greatly from fragmentation problems on larger systems