

JASON GREGORY
LEAD PROGRAMMER
NAUGHTY DOG, INC.

Dogged Determination

Technology and Process at Naughty Dog, Inc.

“The University of Naughty Dog”

- ✱ **Paul Keet**, Lead Programmer, Medal of Honor Electronic Arts





Naughty Dog Games

- ✱ Believable **characters**, compelling **stories**
- ✱ Jaw-dropping **visuals**
- ✱ Some of the best **animation** in the biz
- ✱ Rich and immersive **soundscapes**
- ✱ Touching **vocal performances** and memorable **music**
- ✱ World-class **technology**

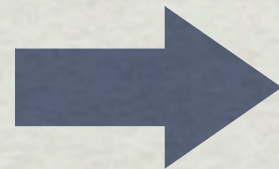
So What's Our Secret?

- * People

- * Culture

- * Process

- * Technology



CONTENT!

People

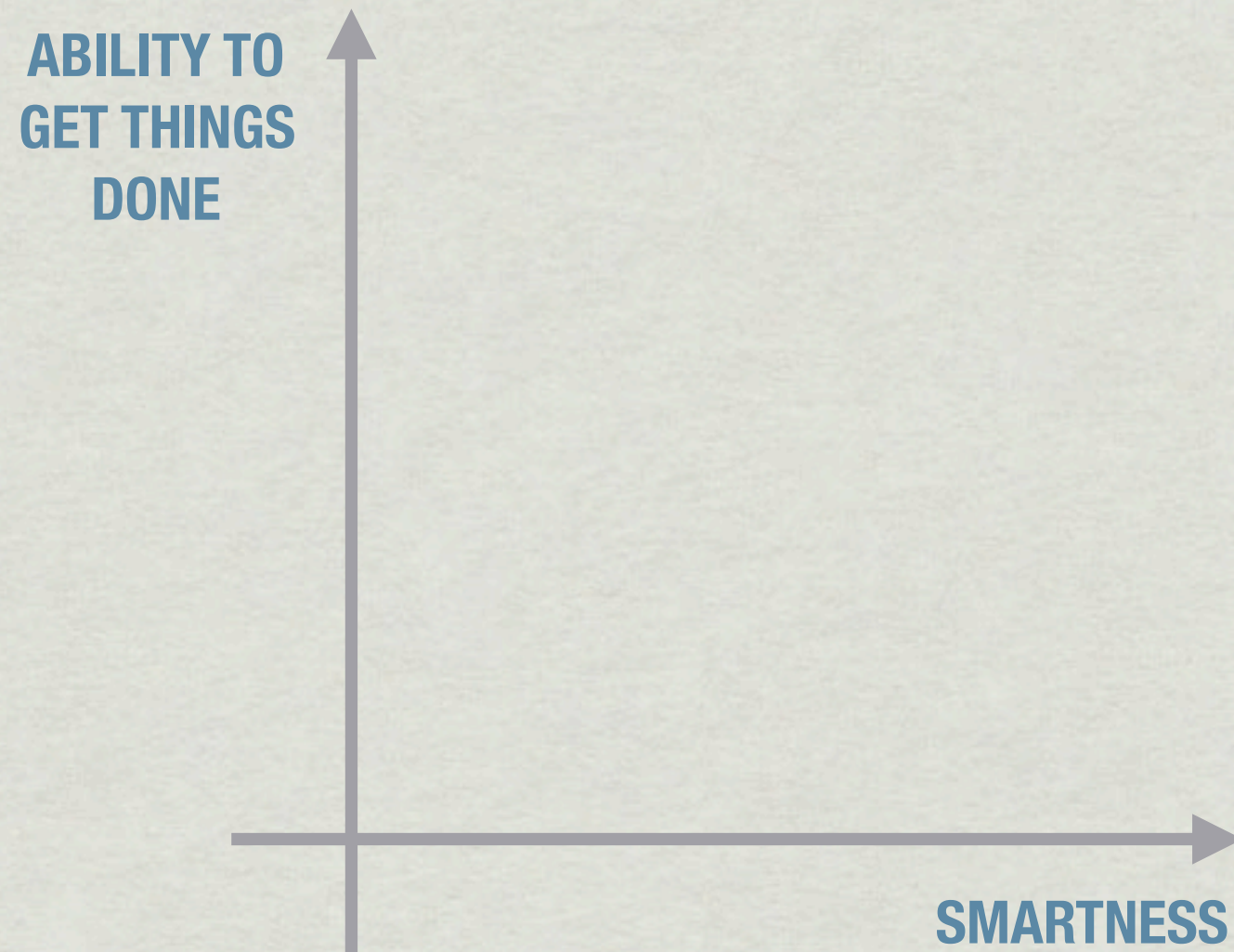
People

- * Making games (and most commercial products!) is a **team effort**
- * What makes a great team?
 - * Each **individual** is of the highest caliber
 - * Team **as a whole** “gels” and operates effectively
- * Effective **hiring** is the key

Axes of Hiring

- ✱ Joel Spolsky, of Microsoft and Fog Creek fame, speaks of the **two axes of hiring**
- ✱ Is the candidate **smart**?
- ✱ Can the candidate **get things done**?

Axes of Hiring



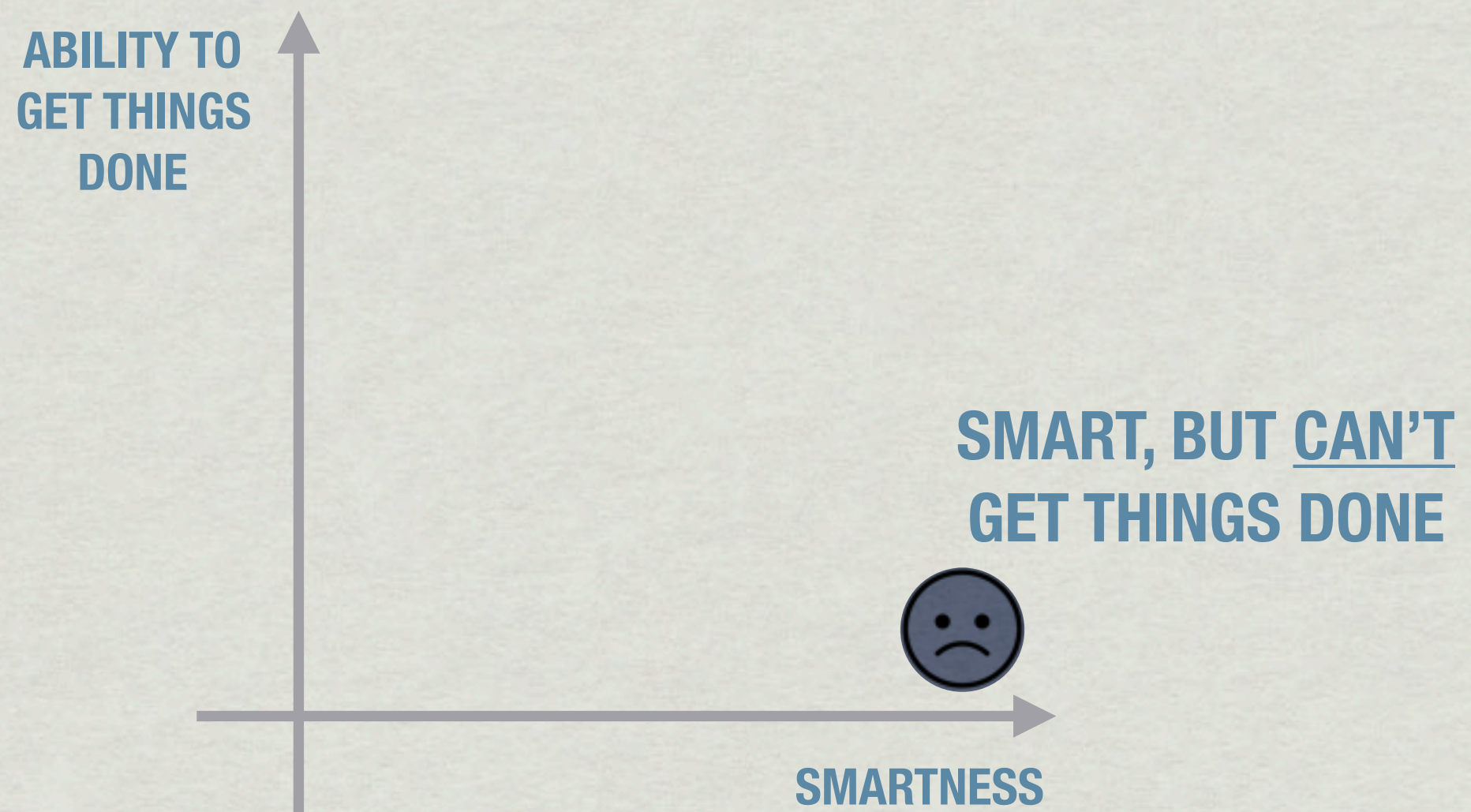
Axes of Hiring

**ABILITY TO
GET THINGS
DONE**

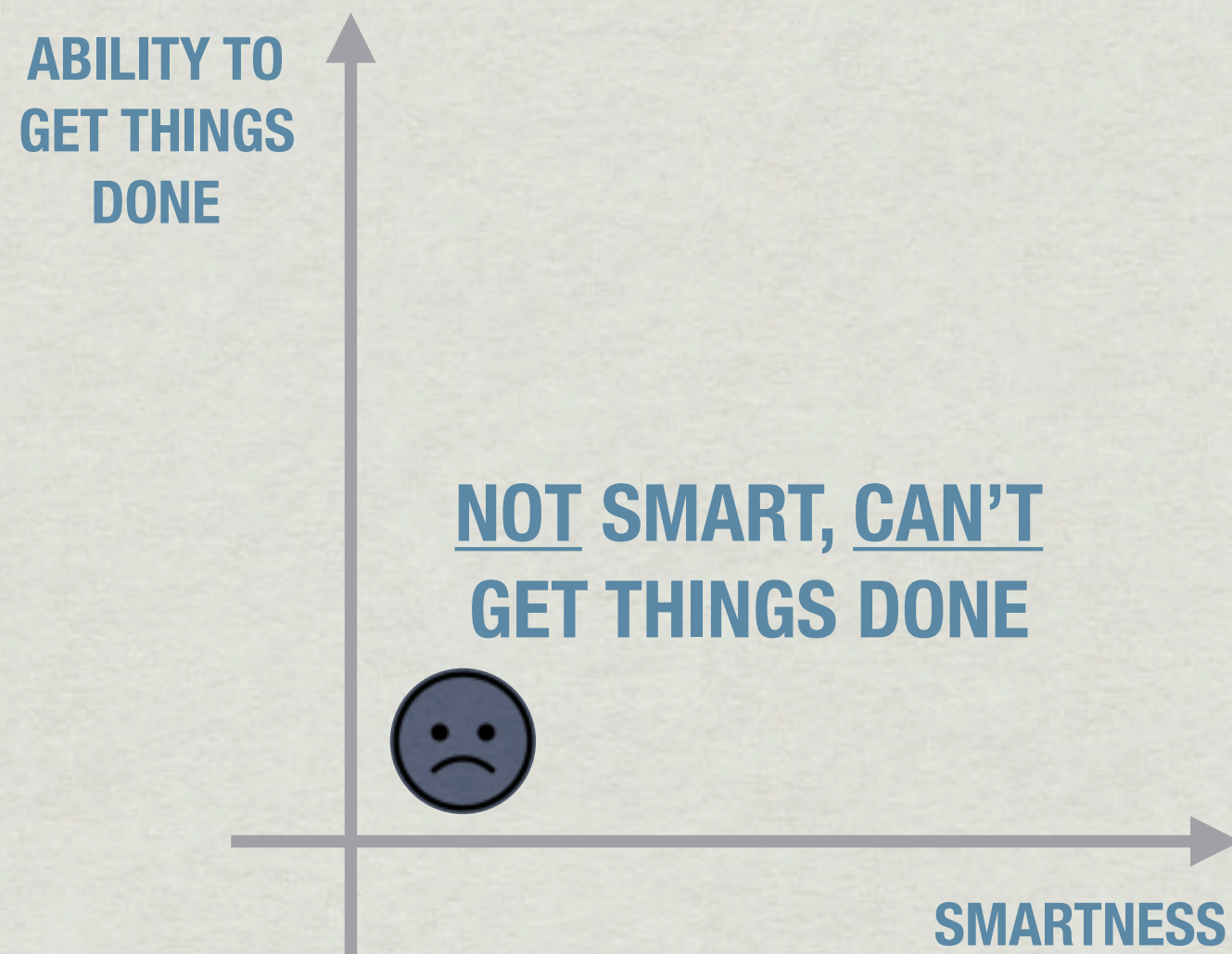

**SMART AND
GETS THINGS DONE**

SMARTNESS

Axes of Hiring



Axes of Hiring



Axes of Hiring

ABILITY TO
GET THINGS
DONE



NOT SMART AND
GETS THINGS DONE!

SMARTNESS

Hiring Programmers

- ✱ At Naughty Dog, programmer candidates are evaluated for:
 - ✱ **math** and **problem-solving** skills,
 - ✱ knowledge of **low-level hardware** and **optimization** techniques,
 - ✱ general **computer science** (data structures, algorithms, languages)
 - ✱ ... in **that order**!

Culture

Retaining Your Talent

- * **Trust** your people
- * Give them **creative freedom** and **responsibility**
- * Build a culture of **mutual respect** and **continual learning**
- * Effort translates directly to **rewards**
- * Don't be afraid to let a **bad apple** go

Flying by the Seat of Our Pants

- ✱ Richard Lemarchand, Co-Lead Designer on Uncharted series gave a great talk on this topic

“How to Fly by the Seat of Your Pants (Without Crapping Them)”

- D.I.C.E. Summit, 2010

<http://www.g4tv.com/videos/44276/dice-2010-naughty-dog-presentation/>

Culture

- * Environment of **mutual respect** and **trust**
- * **Open door** policy; **collaboration** encouraged
- * Aggressively criticize **ideas**, but never let it get personal
- * **No producers** ⇒ **everyone** is a producer!
- * Literally **every employee contributes** to making the game...
 - * ... even our receptionist... and the two co-presidents!

Process

Development Process

- ✱ Each team at Naughty Dog has its own process
- ✱ We'll focus on **technology development process** in this talk (the process used by the programmers)
- ✱ We'll also talk a bit about the **overall process** within the studio

Core Philosophies

- * **KISS** -- keep it simple (stupid)
- * **Rapid iteration**
- * Keep the game **running at all times**
- * **Everyone** always runs the **latest version** of the game
- * **Minimal meetings** (maximum communication!)
- * Manage **scope** of the project to **maximize quality**

KISS

- * Not everything we do at Naughty Dog is **rocket science**...
- * ... only ***some*** of it is!
- * In fact, we *usually* select the **simplest**, most straightforward solution that gets the job done
 - * e.g., simple text files, not flashy GUIs
 - * e.g., command-line build tools
 - * Don't reinvent the wheel

Rapid Iteration

- * Focus on achieving **results on-screen...**
 - * ... not architectural perfection or “religious” dogma
- * **Rapid iteration!**
 - * Get a **rough prototype** up and running ASAP
 - * **Leverage existing systems** to prototype new ones
 - * **Refine and iterate** many, many (many!) times

Keep It Running

- ✱ A central pillar of our rapid iteration approach:

Keep the game running at all times

- ✱ When developing a new system, we usually either:
 1. **gradually evolve** an existing system, or
 2. bring the new system up **in parallel** to the old

Revision Control

- ✱ **Revision control** is central to the studio
- ✱ Code, script, text data files: **Perforce**

Revision Control

- ✱ **Revision control** is central to the studio
- ✱ Code, script, text data files: **Perforce**



Revision Control

- * **Revision control** is central to the studio
- * Code, script, text data files: **Perforce**

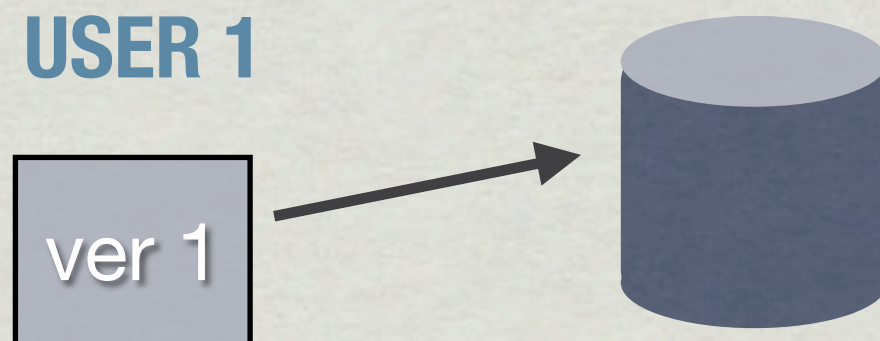
USER 1

ver 1



Revision Control

- ✱ **Revision control** is central to the studio
- ✱ Code, script, text data files: **Perforce**



Revision Control

- * **Revision control** is central to the studio
- * Code, script, text data files: **Perforce**

USER 1

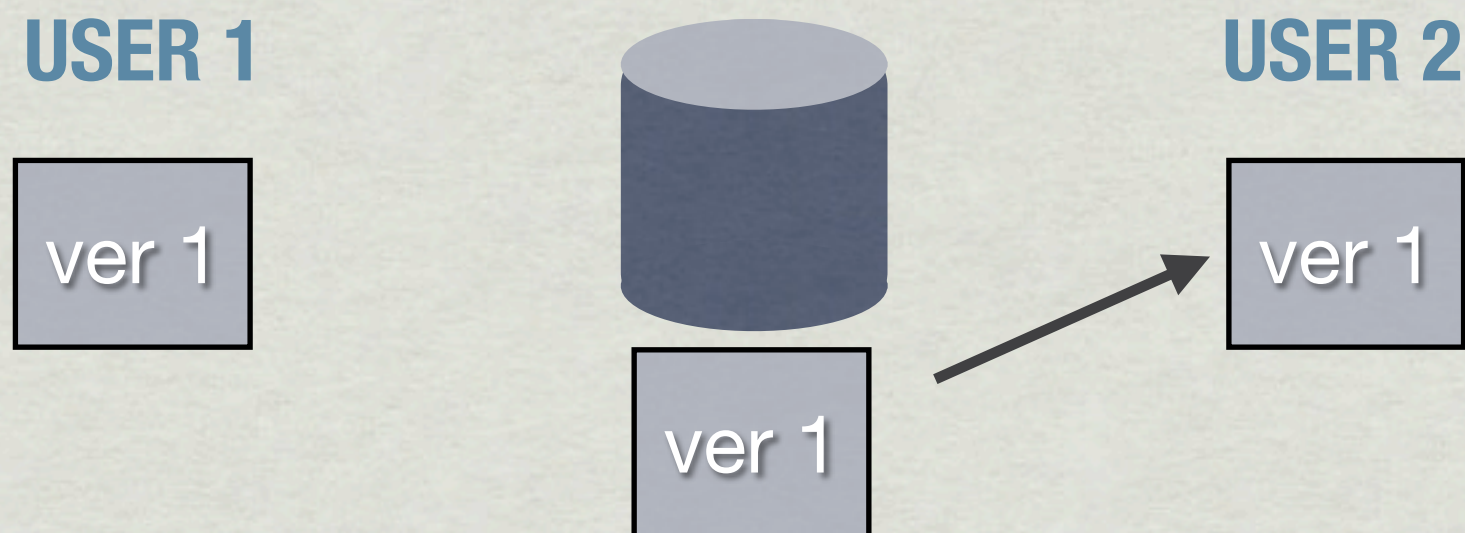
ver 1



ver 1

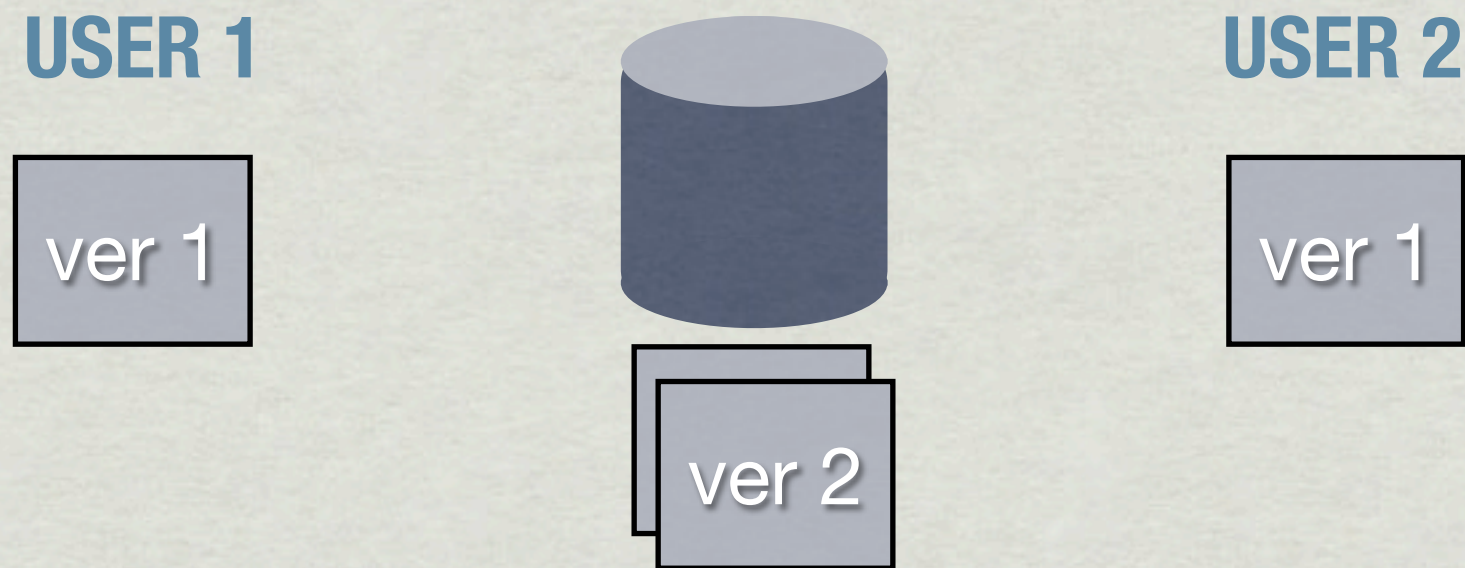
Revision Control

- * **Revision control** is central to the studio
- * Code, script, text data files: **Perforce**



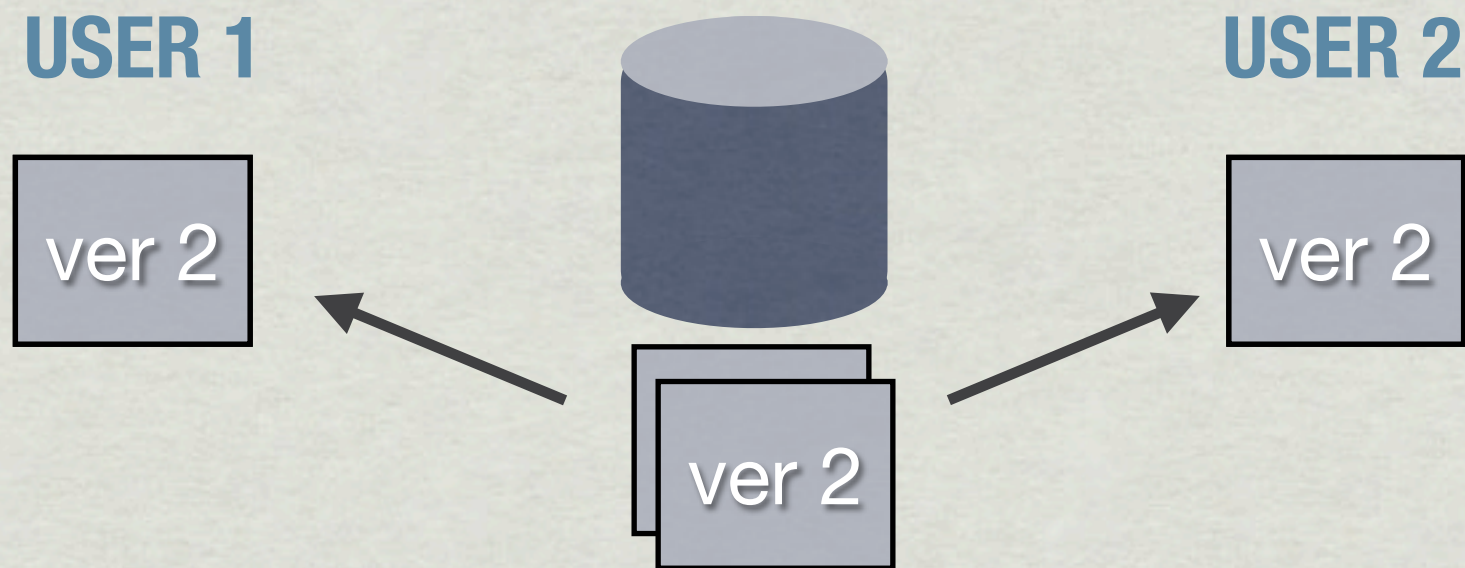
Revision Control

- * **Revision control** is central to the studio
- * Code, script, text data files: **Perforce**



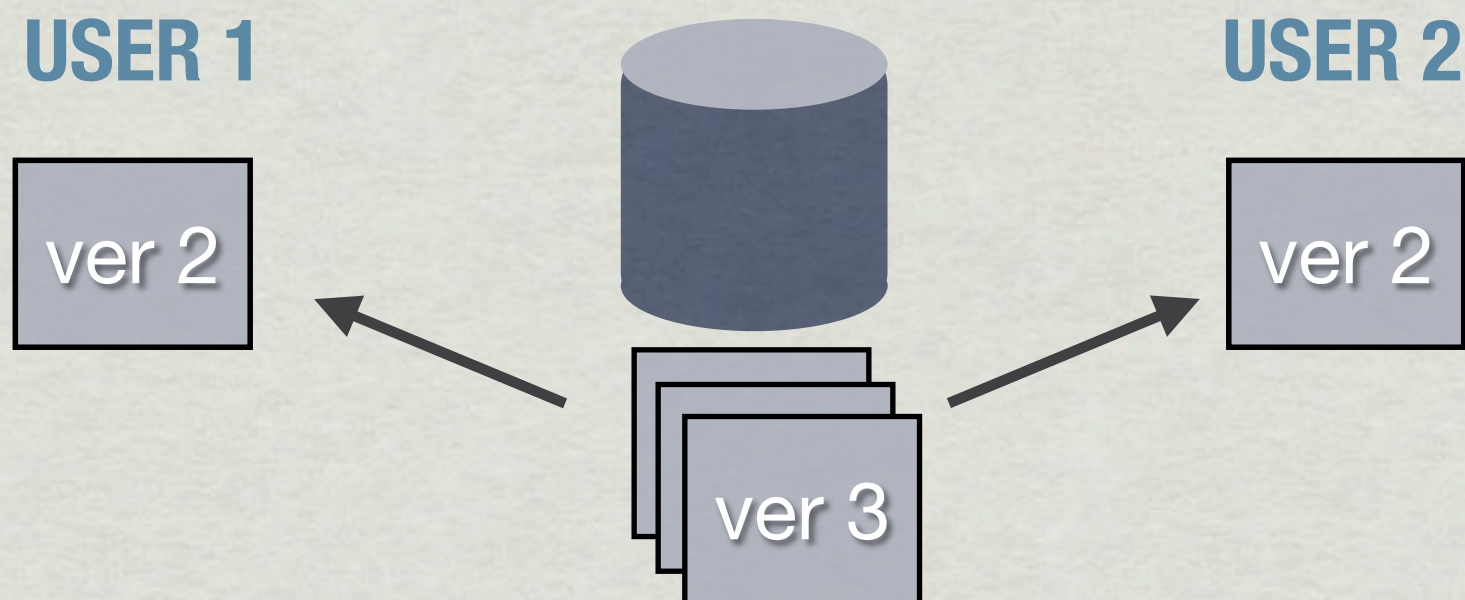
Revision Control

- * **Revision control** is central to the studio
- * Code, script, text data files: **Perforce**



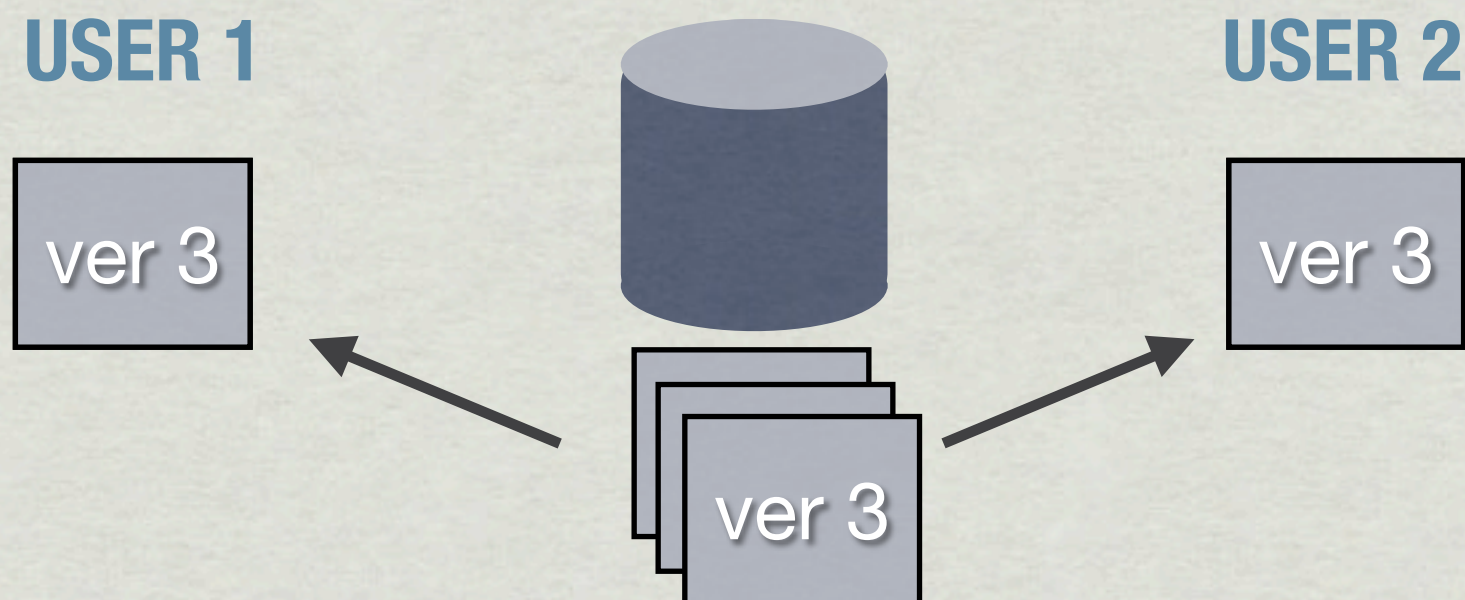
Revision Control

- * **Revision control** is central to the studio
- * Code, script, text data files: **Perforce**



Revision Control

- * **Revision control** is central to the studio
- * Code, script, text data files: **Perforce**

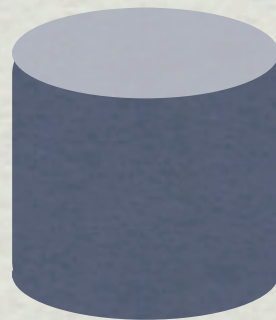


Revision Control

- ✱ Art assets: **proprietary in-house tool** (“BAM”)

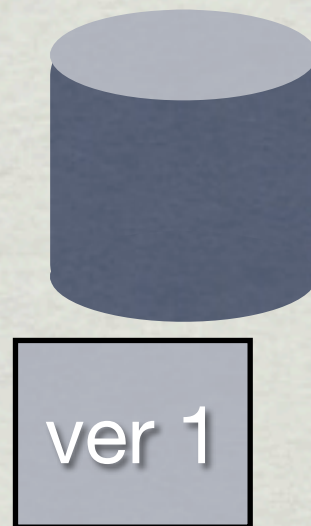
Revision Control

- ✱ Art assets: **proprietary in-house tool** (“BAM”)



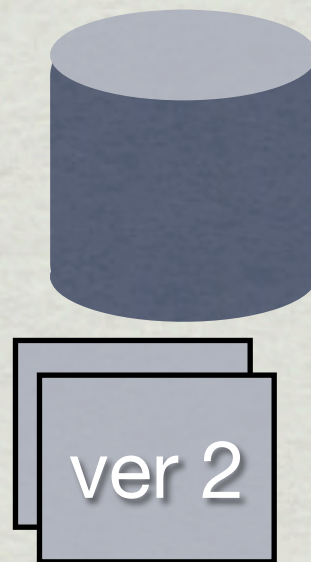
Revision Control

- ✱ Art assets: **proprietary in-house tool** (“BAM”)



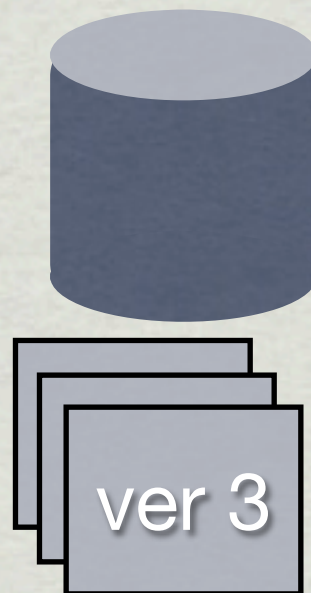
Revision Control

- ✱ Art assets: **proprietary in-house tool** (“BAM”)



Revision Control

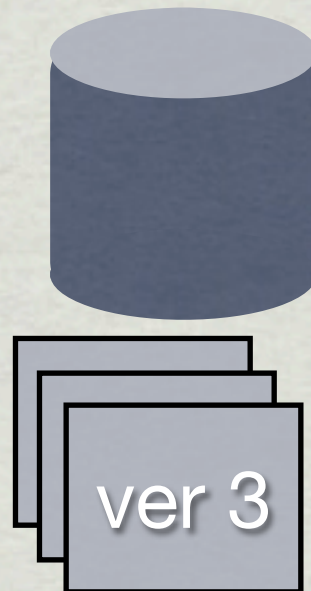
- ✱ Art assets: **proprietary in-house tool** (“BAM”)



Revision Control

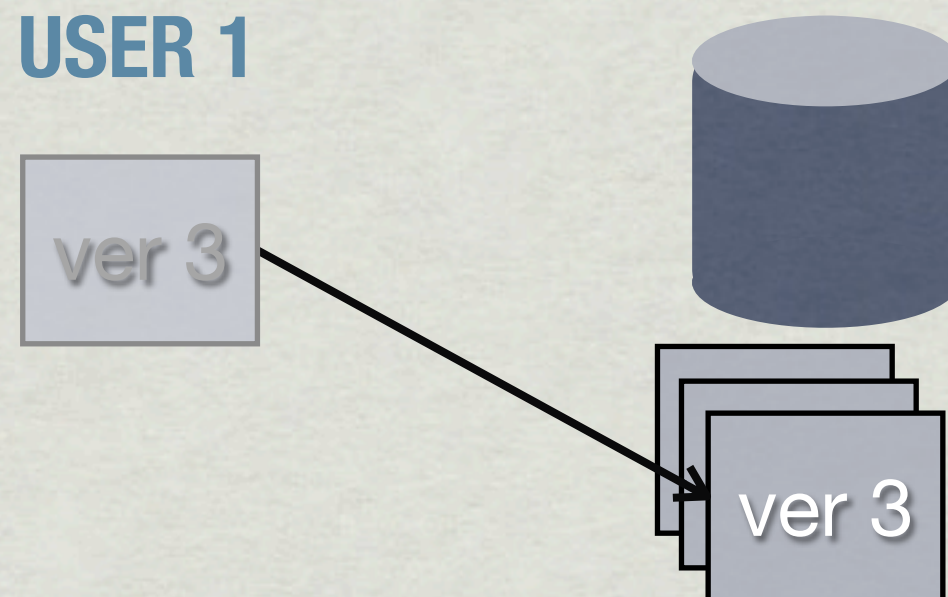
- * Art assets: **proprietary in-house tool** (“BAM”)

USER 1



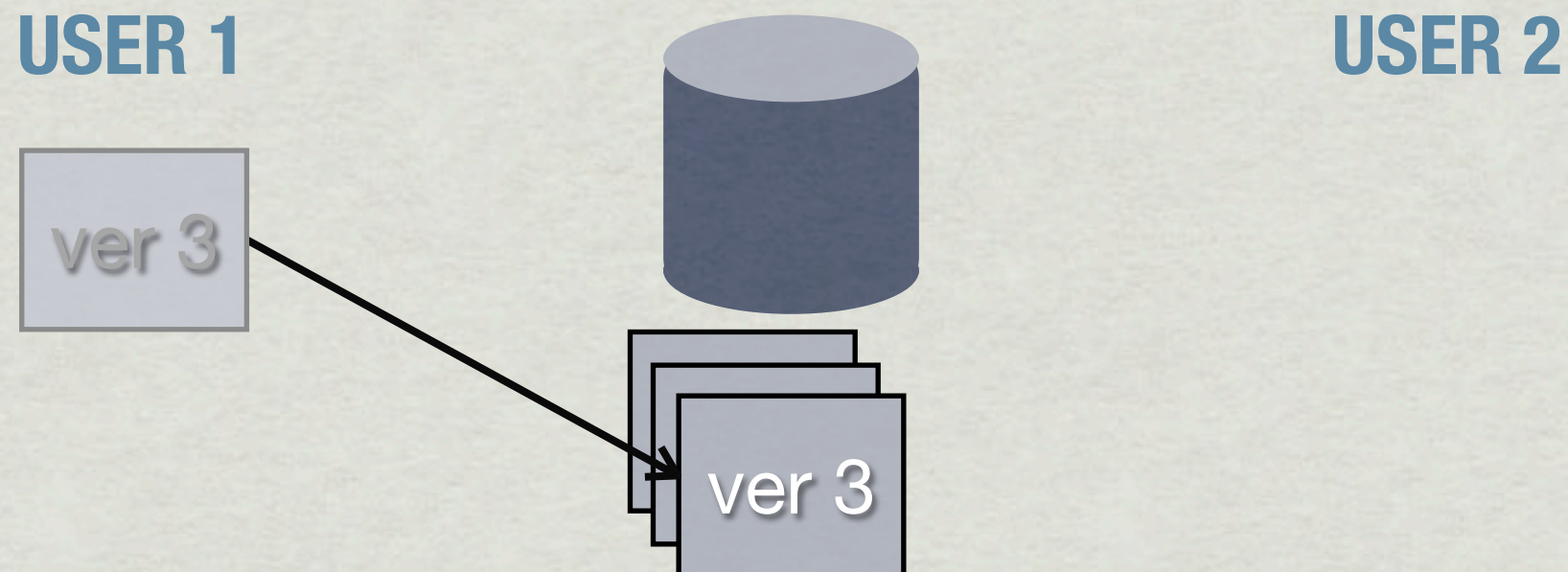
Revision Control

- * Art assets: **proprietary in-house tool** (“BAM”)



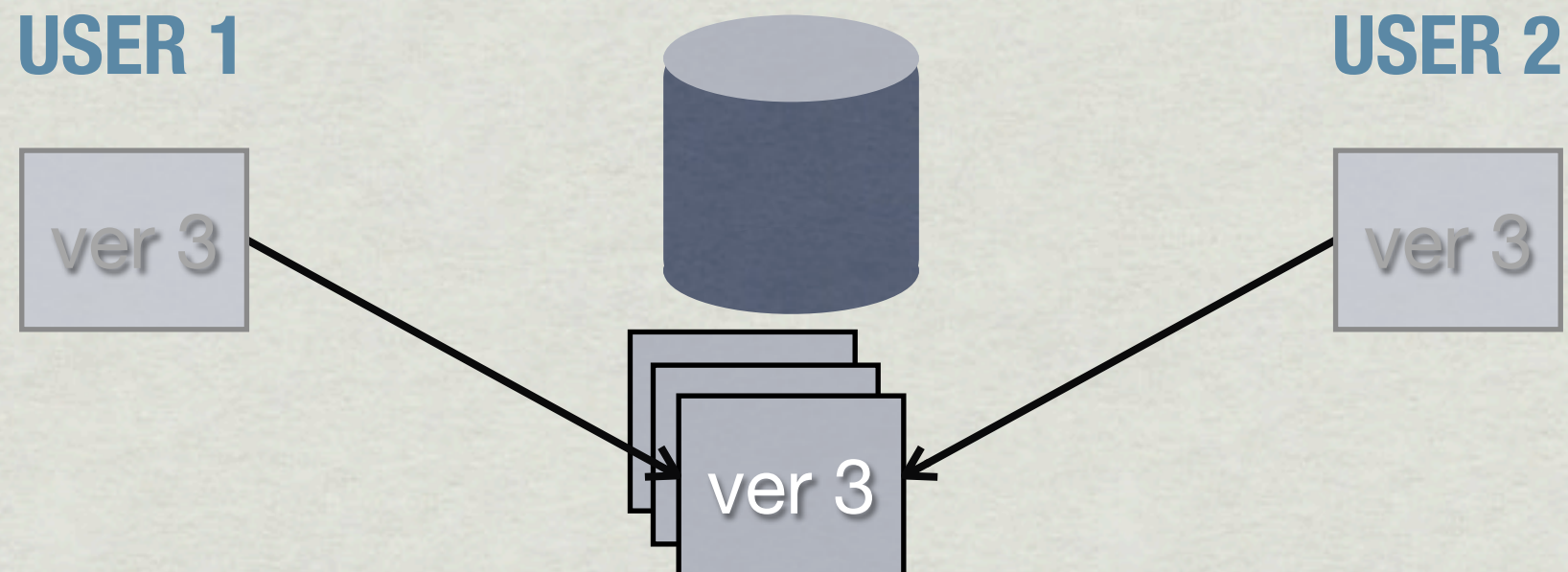
Revision Control

- * Art assets: **proprietary in-house tool** (“BAM”)



Revision Control

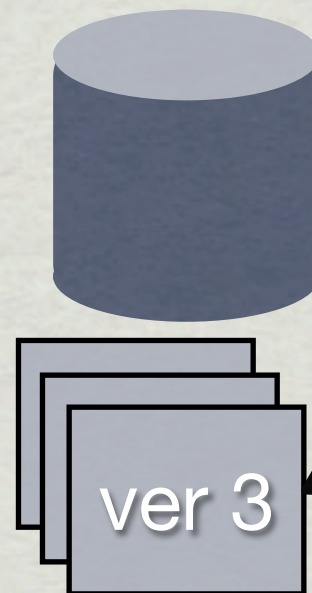
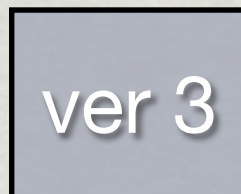
- * Art assets: **proprietary in-house tool** (“BAM”)



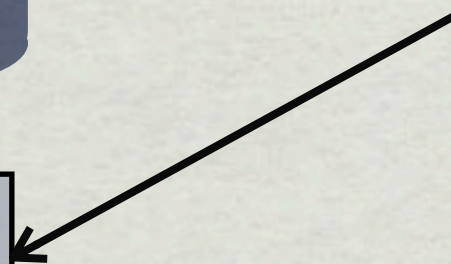
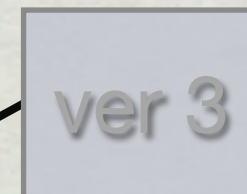
Revision Control

- * Art assets: **proprietary in-house tool** (“BAM”)

USER 1



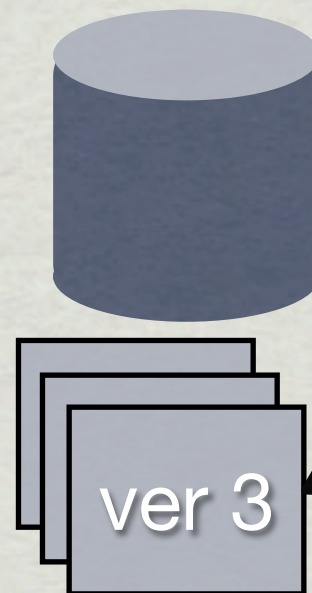
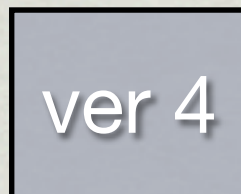
USER 2



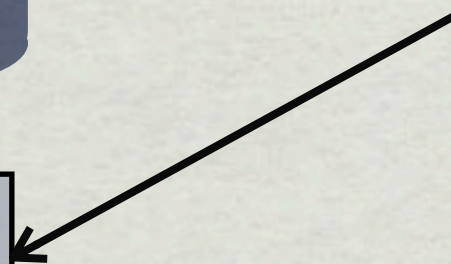
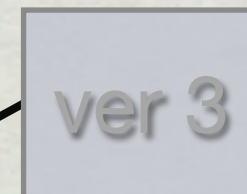
Revision Control

- * Art assets: **proprietary in-house tool** (“BAM”)

USER 1

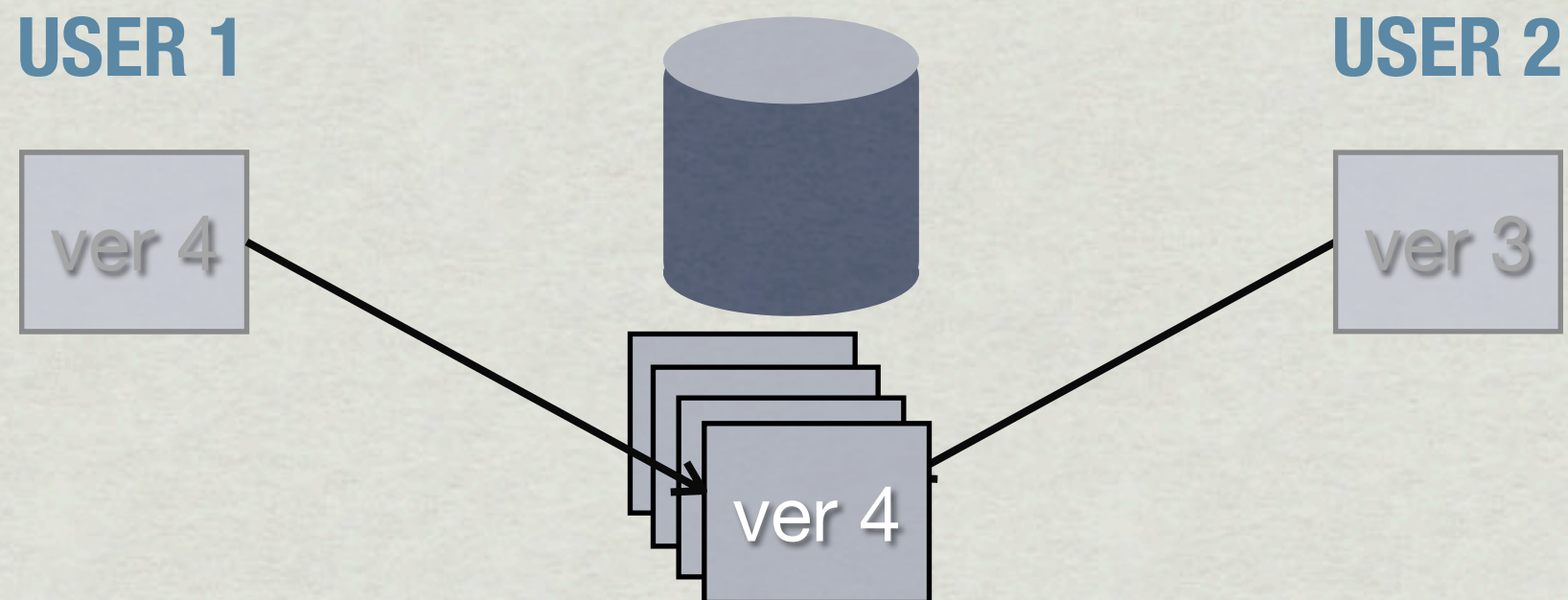


USER 2



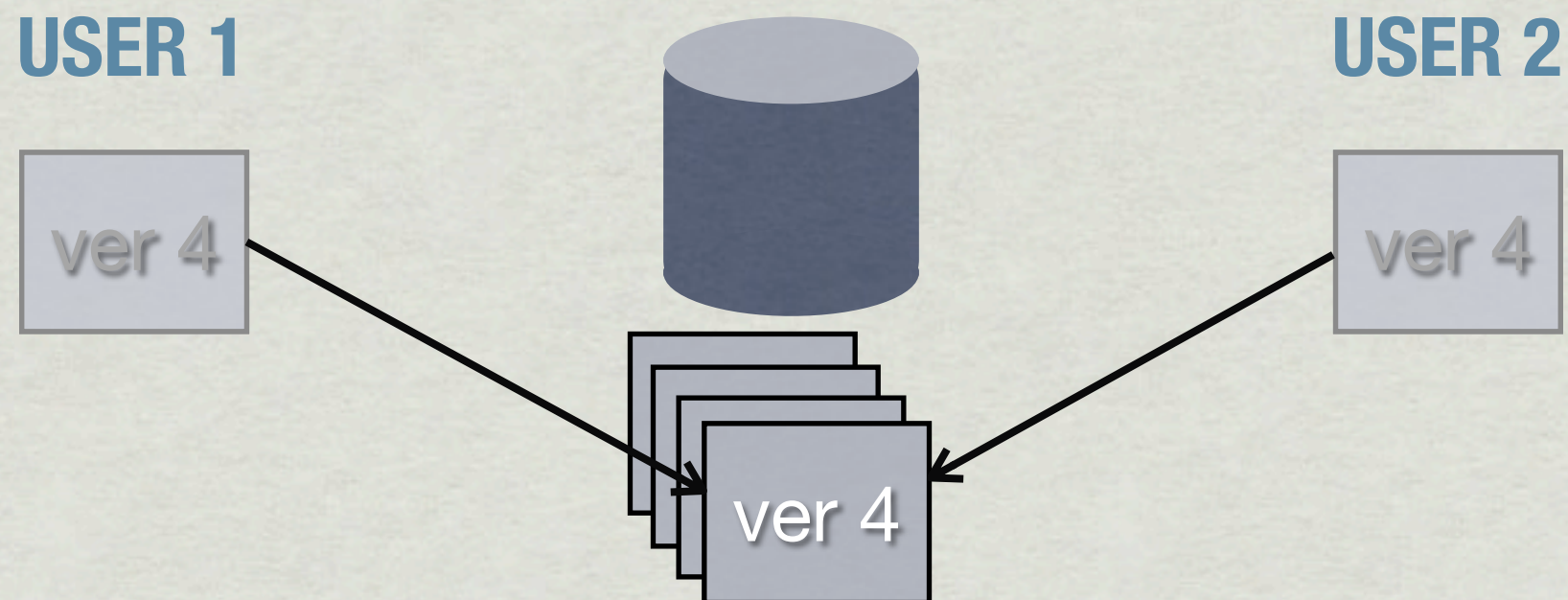
Revision Control

- * Art assets: **proprietary in-house tool** (“BAM”)



Revision Control

- * Art assets: **proprietary in-house tool** (“BAM”)



The Bleeding Edge

- * Everyone works on **latest** code/assets at all times
- * **Automated build script** called the **buildbot** builds *every change* checked into Perforce
 - * If the build **succeeds**, a new version of the game is **published** to entire studio
 - * If the build **fails**, **email** is sent to entire studio
- * As **assets** are changed, artists build them **globally**

The Bleeding Edge

“But how can you work like that?”

- ✱ Sounds **dangerous**, right?
- ✱ But actually it's one of the **best aspects** of Naughty Dog's development process

The Bleeding Edge

- * Having **latest code** published immediately means that **errors are discovered immediately**
- * The **easiest bug to fix** is the one that is **freshest** in your mind
- * Impossible for anyone to get **months behind** the team
- * There is exactly **one version** of the game that we all run
 - * No more “... but it works on *my* machine!”
- * Discourages **dangerous programming practices**

The Bleeding Edge


- * Same goes for **publishing assets globally**:
 - * **Problems** are discovered **quickly**
 - * Forces people to **think carefully** about their changes
 - * Exactly **one version** of the game assets, seen by everyone
 - * **Rapid iteration** encouraged; easy to share/collaborate
 - * No more “... but it works on *my* machine!”

Minimal Meetings


- * Very few long, formal meetings...
- * ... and a LOT of impromptu, short, informal meetings!
- * Just pop by someone's desk and discuss
- * Go grab other people if their input is needed
- * Summarize results in an email or on the wiki
- * Manage via our online task system, "Tasker"



Tuesday, March 4, 14

Task Project: T1 ▼

Task  **add more granularity to actor load dependency debug data**


Tuesday, March 4, 14

Description	
<p>Currently, we can debug why an actor is loading by dumping level dependencies. Unfortunately, that only gives us data to the granularity of what level is loading the actor. Further granularity would be helpful for debugging -- to the level of what spawner or what look in the level is loading the actor as a dependency.</p> <p>For instance, if I see anim-melee-common loaded by level X, I then have to check every artgroup on every npc in the level and compare it against the looks file in dc to see which look(s) bring in that actor.</p> <p>It would be nice if I could see which looks used in that level have anim-melee-common as a dependency without having to guess and check each look until I find the right ones(s).</p>	

Team					
	Name/Viewed	Status	Hours Left		Due Date
	<div>★ Dave Smith</div> <div>Tue Jan 7 at 10:53</div> <div>NEXT</div>	<input type="text" value="-"/>	<input type="text" value="0"/>	<input type="text" value="▼"/>	<input type="text" value=""/>
	<div>★ Jason Gregory</div> <div>Thu Jan 2 at 11:15</div>	<input type="text" value="-"/>	<input type="text" value="0"/>	<input type="text" value="▼"/>	<input type="text" value=""/>
	<div>✓ Travis McIntosh</div> <div>Wed Dec 11 at 12:11</div>	<input type="text" value="-"/>	<input type="text" value="0"/>	<input type="text" value="▼"/>	<input type="text" value=""/>
	<div>★ Jerome Durand</div> <div>Fri Sep 27 at 18:04</div>	<input type="text" value="-"/>	<input type="text" value="0"/>	<input type="text" value="▼"/>	<input type="text" value=""/>
	<div>★ Christian Gyrling</div> <div>Tue Sep 24 at 11:31</div>	<input type="text" value="-"/>	<input type="text" value="0"/>	<input type="text" value="▼"/>	<input type="text" value=""/>
	<div>★ Jacob Minkoff</div> <div>Wed Jan 22 at 14:40</div>				<input type="text" value=""/>
Add team members...					


Activity

Show Auto Comments




Type comment here... (Enter to submit)

Upload Screenshot Screenshot(Debug)



Christian Gyrling Tue Sep 24 at 11:30
Sorry, jumped the gun slightly there, but we can still hook this up through the LoadSets.



Christian Gyrling Tue Sep 24 at 11:30

Tuesday, March 4, 14

Balance

- * Careful, thoughtful balance between **story** and **gameplay**
- * ... and between **systemic gameplay** and one-off **set pieces**
- * Attention to **detail**
- * **Prioritizing** well / knowing what's **important** (and what's not)

Managing Scope

- * Make the game we want to make
 - * Project schedule always in service to the game (not vice versa)
 - * May cut some content towards the **end** of the project in order to hit our ship date
- * Use various **scheduling tools** to plan ahead
 - * ... but not too far ahead!

Technology

Key Foundational Technologies

- * Efficient, fragmentation-free **memory allocation**
- * Effective use of **multicore** computing resources
- * Careful **code and data optimization** based on deep understanding of the hardware
- * Powerful in-game **debugging** and **profiling** facilities
- * Empower content creators through **script** and **data-driven** systems

Memory Management

Memory Allocation

- * Memory allocation is not free!
 - * General-purpose `new/malloc()` needs to handle every possible request -- slow!
 - * Lowest-level memory allocation routines require a **context switch** into the OS -- **super** slow!
- * Memory **fragmentation** is the enemy
- * Know **how** available memory is being used

Memory Fragmentation



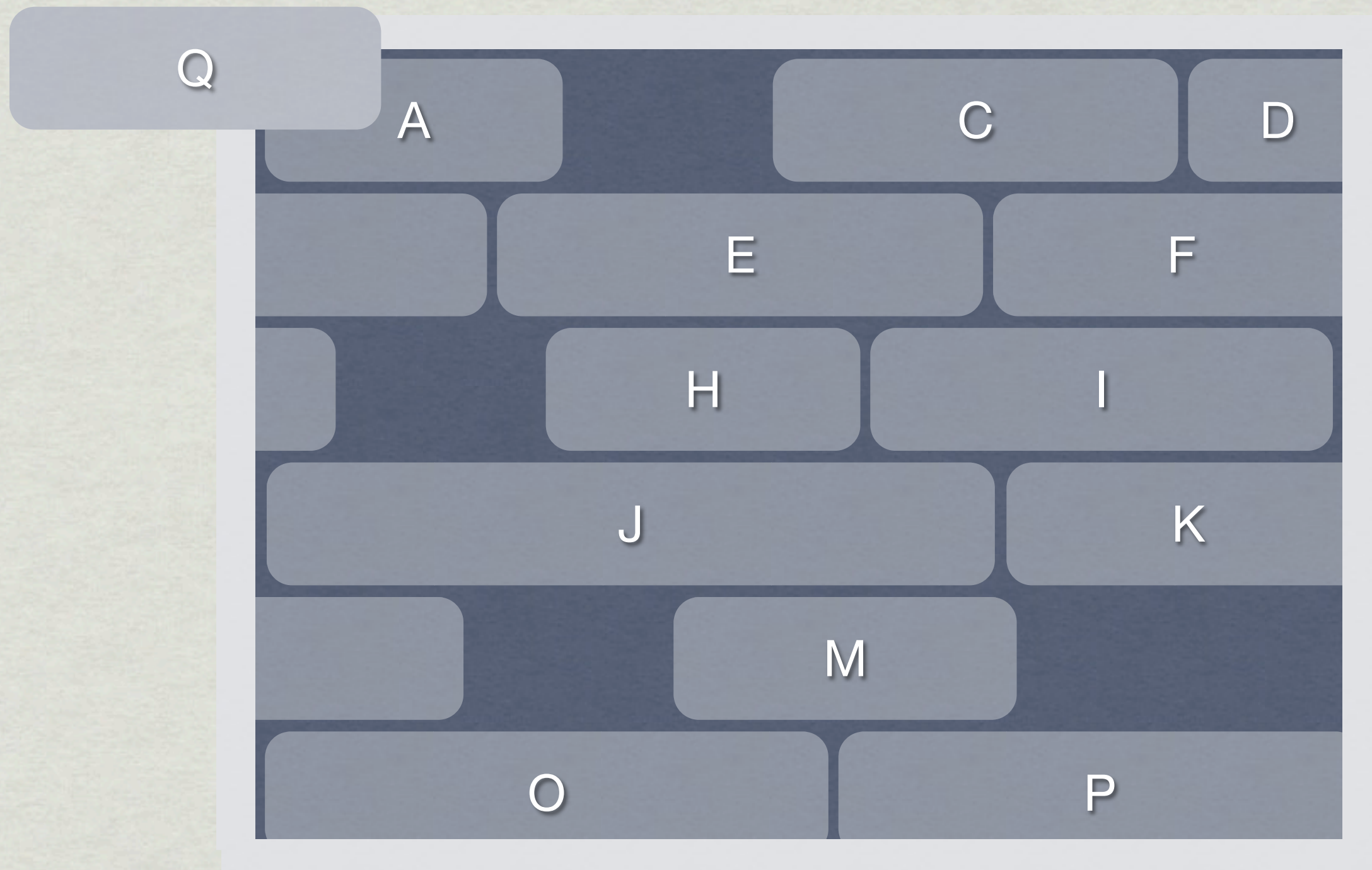
Memory Fragmentation



Memory Fragmentation



Memory Fragmentation



Memory Fragmentation



Memory Fragmentation



Memory Fragmentation



Memory Fragmentation



Memory Fragmentation



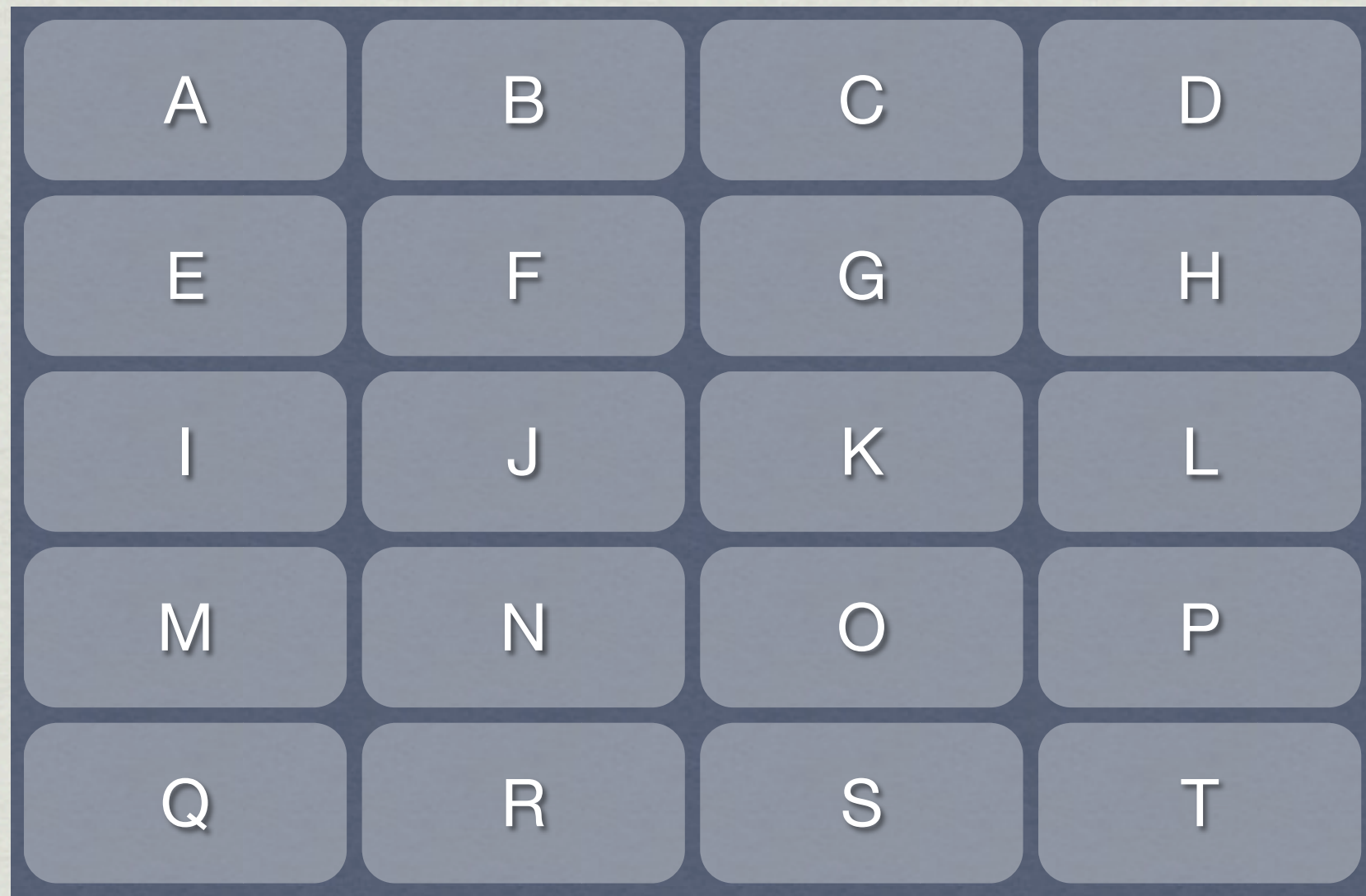
Memory Allocation

- * Always better to do allocation yourself:
 - * Side-step the **OS**
 - * **Custom-tailor** allocators to match your software's **allocation patterns**
 - * **Avoid** memory **fragmentation** entirely
 - * Control your **memory map** explicitly

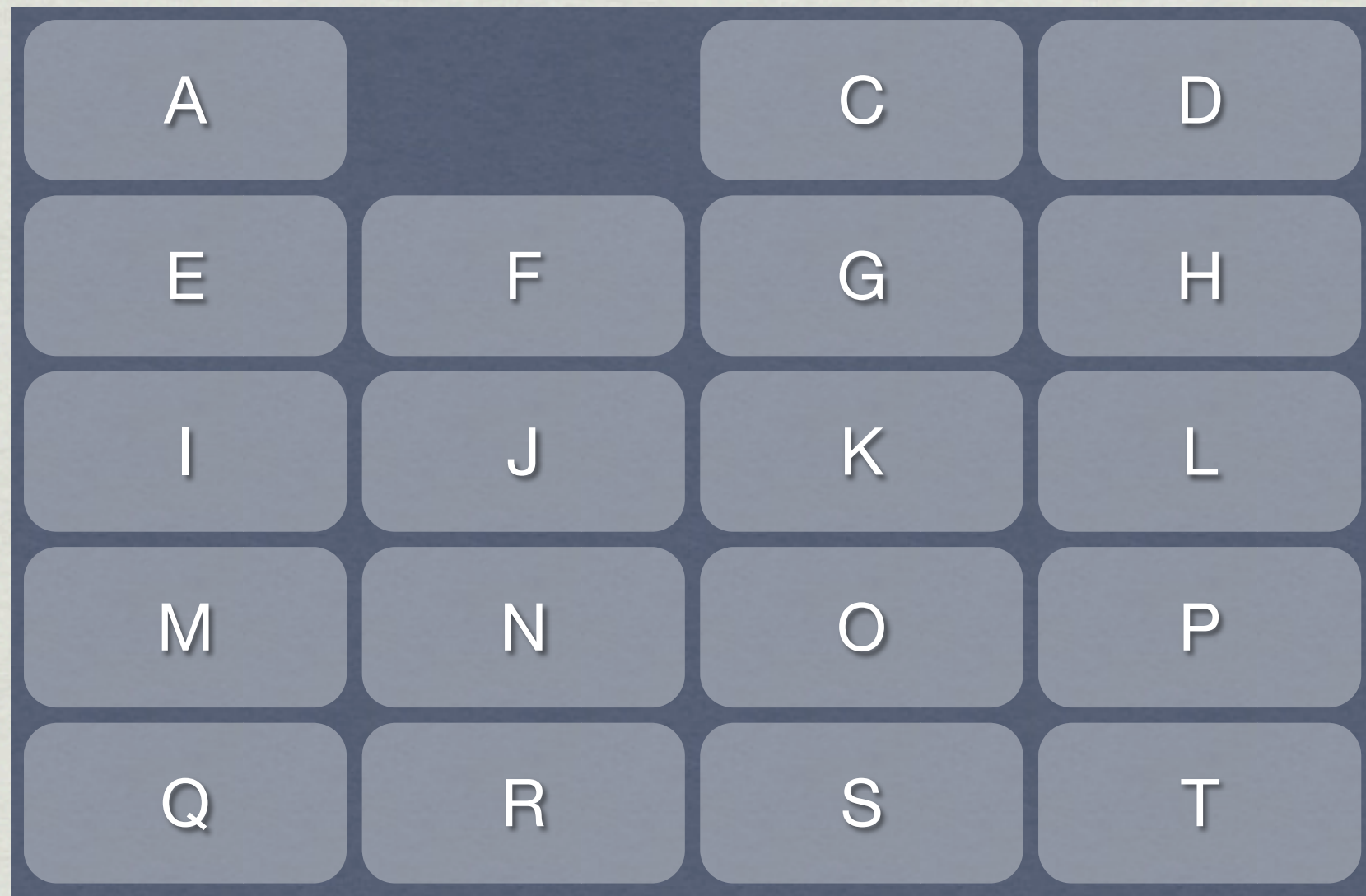
Pool Allocator



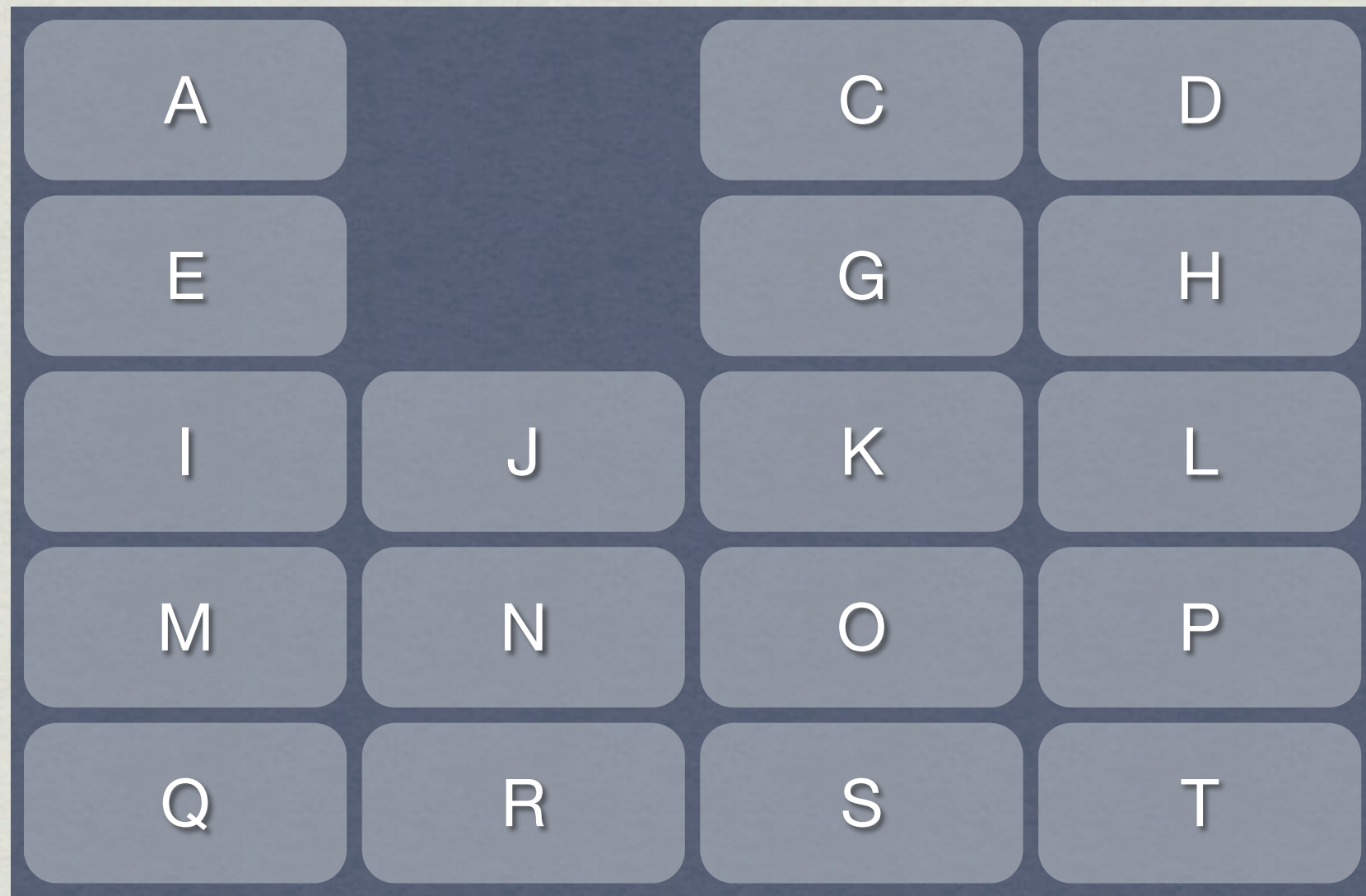
Pool Allocator



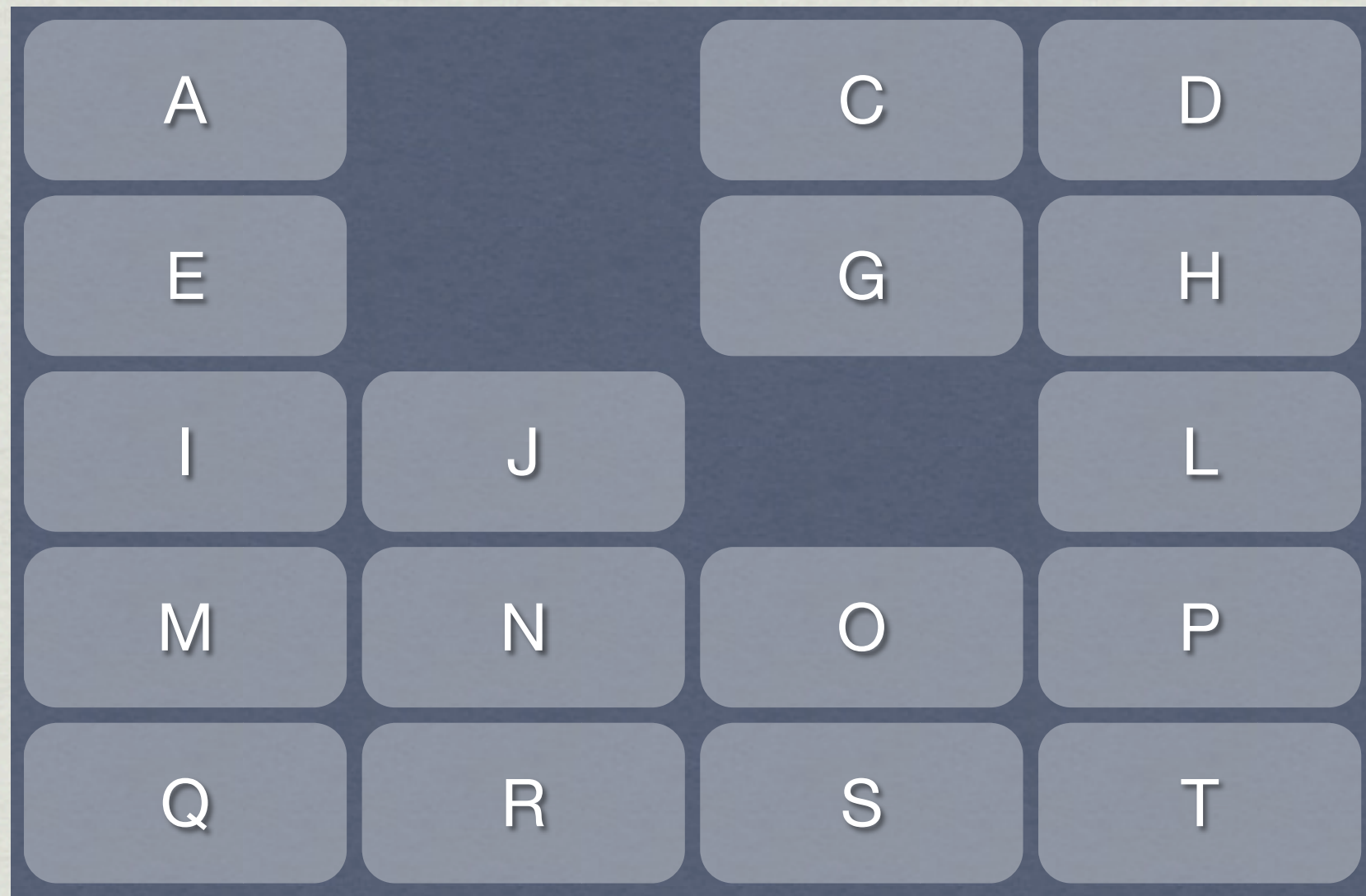
Pool Allocator



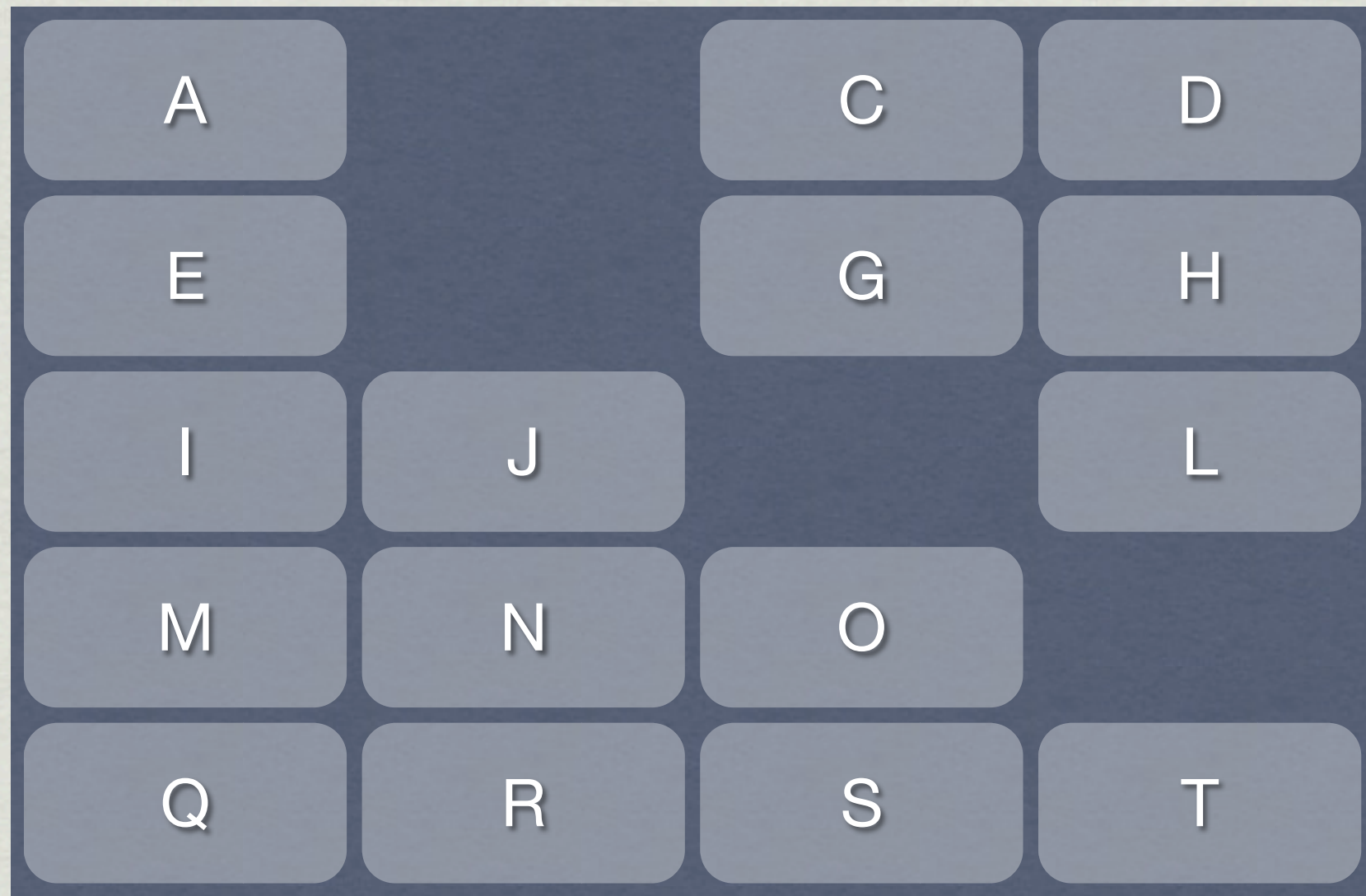
Pool Allocator



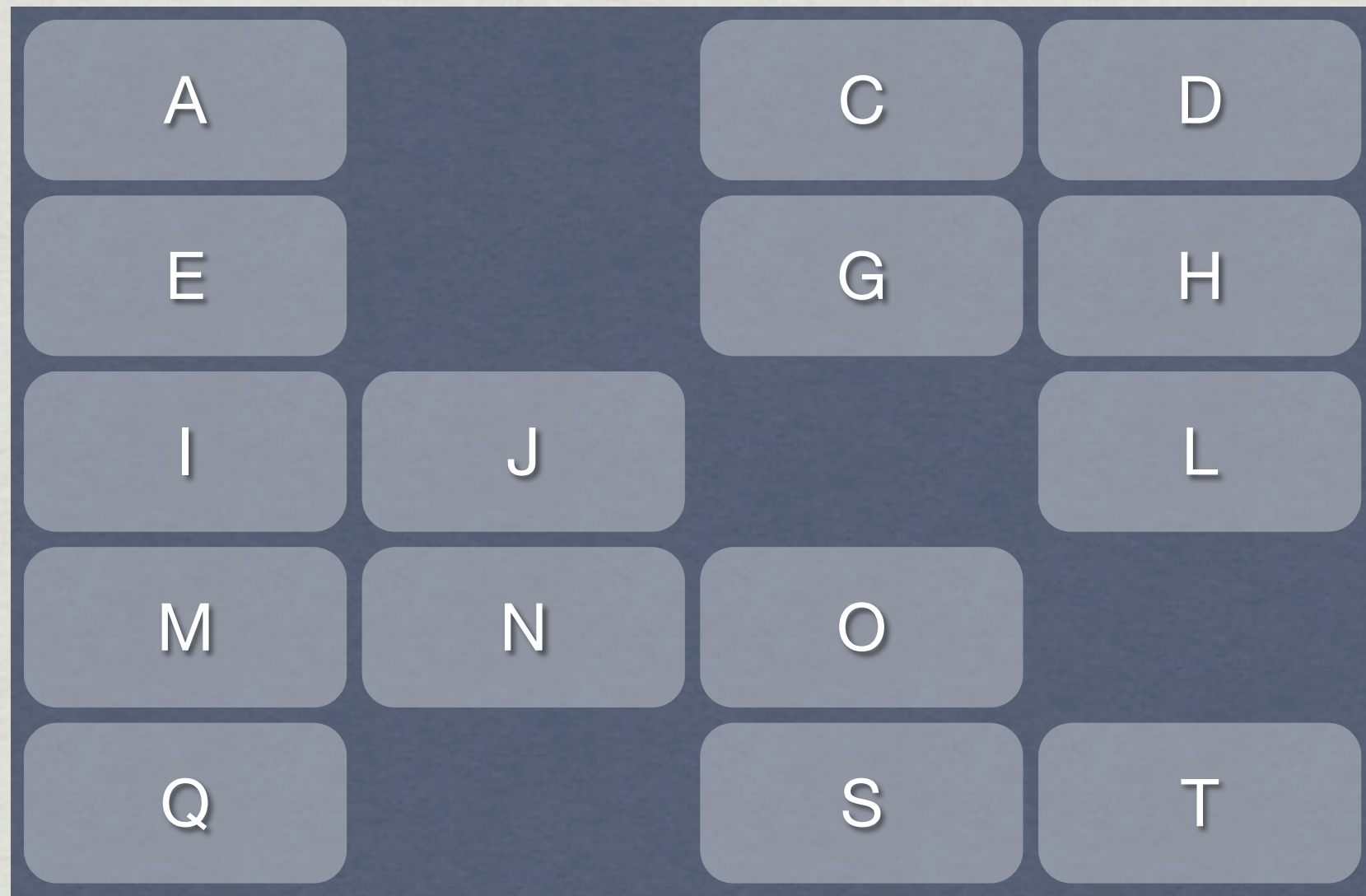
Pool Allocator



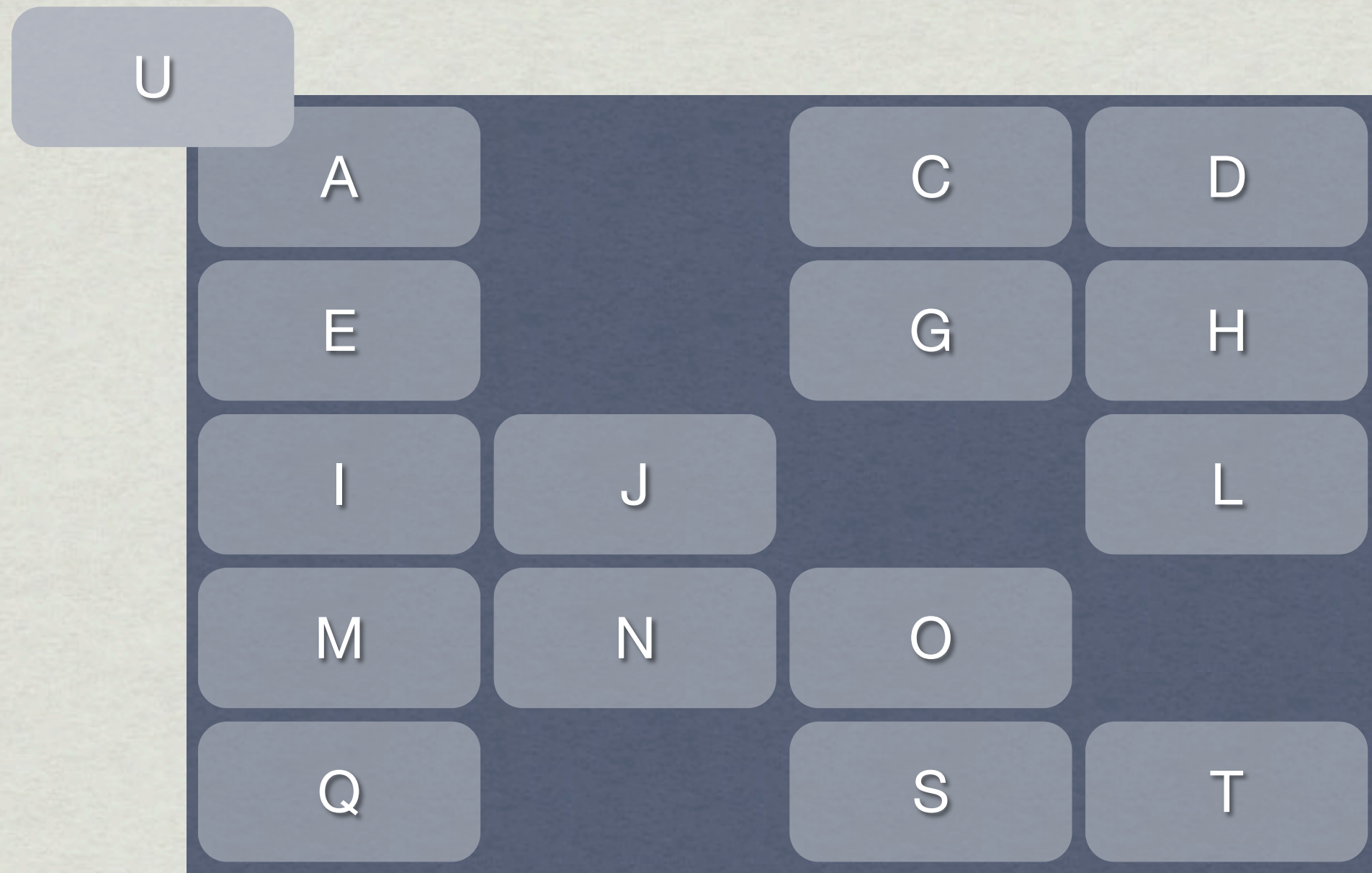
Pool Allocator



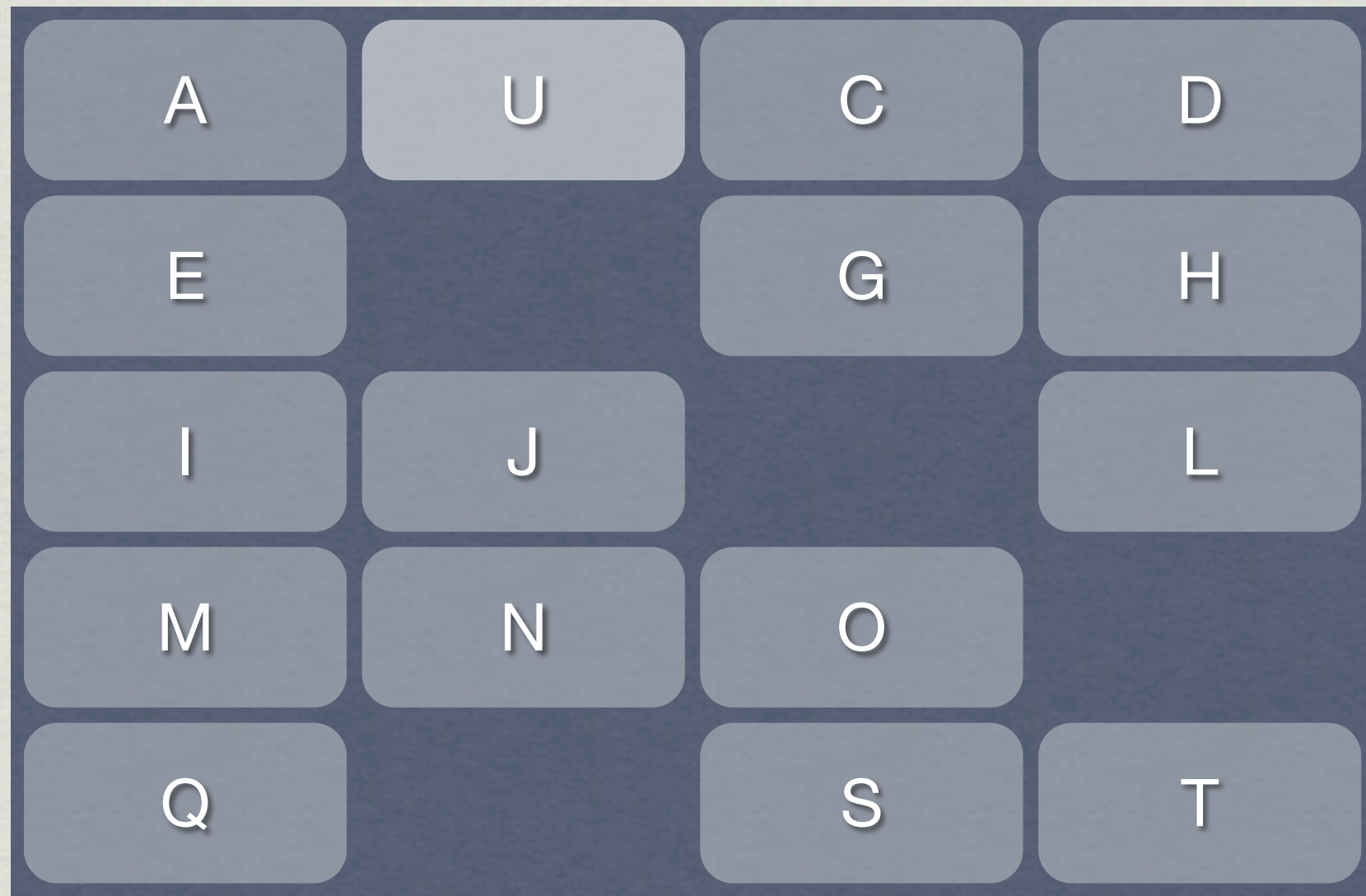
Pool Allocator



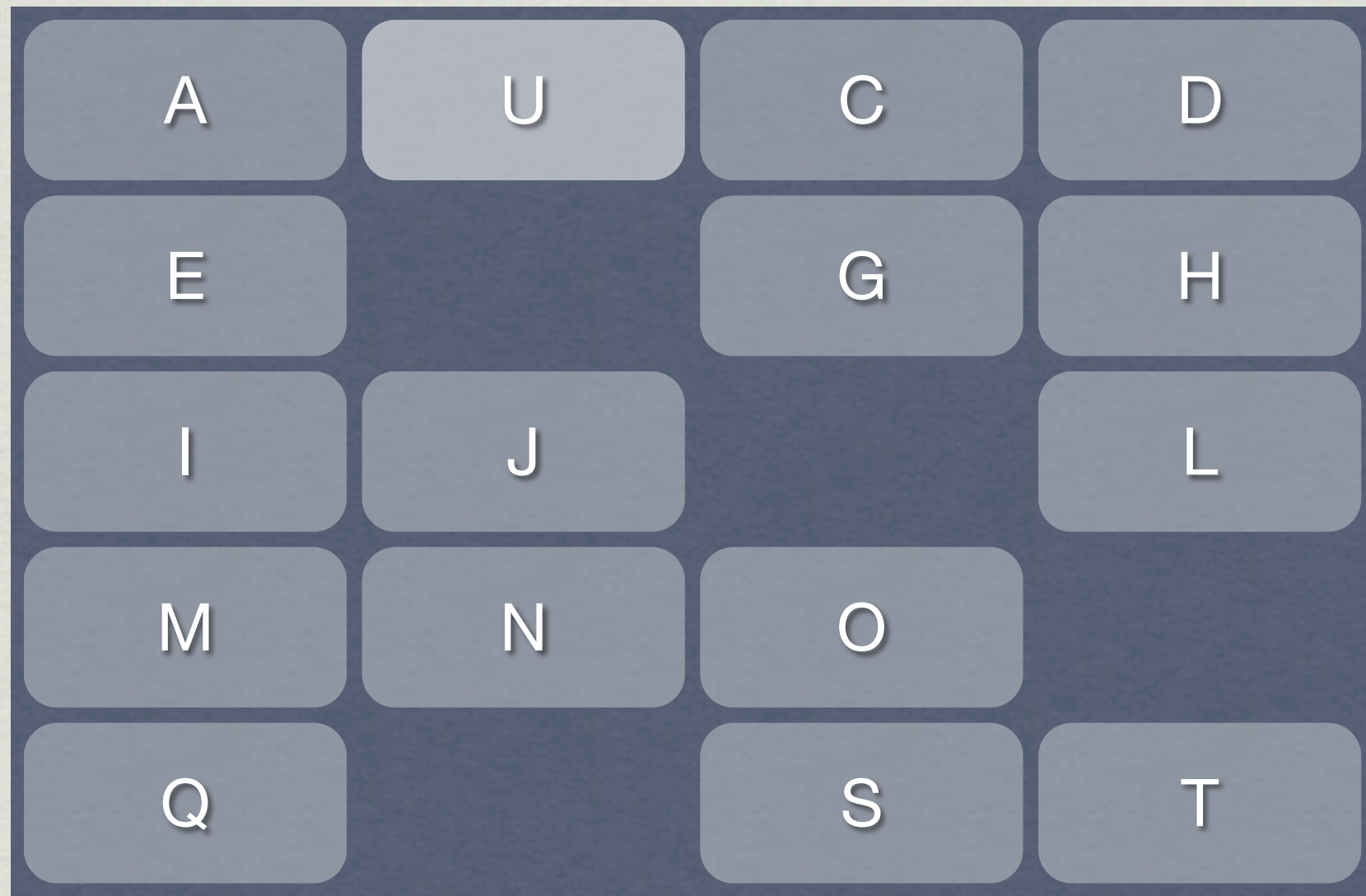
Pool Allocator



Pool Allocator

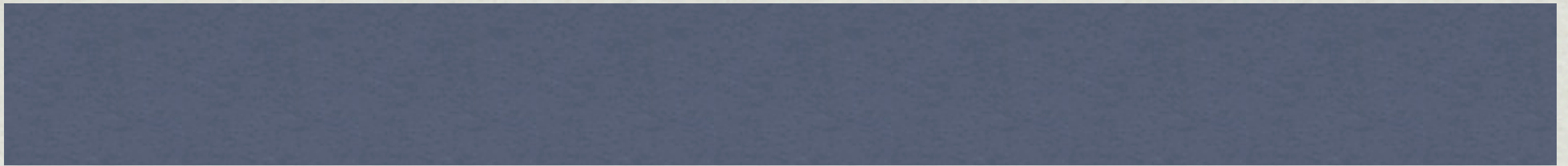


Pool Allocator



FRAGMENTATION HAPPENS, BUT NOT A PROBLEM

Stack Allocator



Stack Allocator



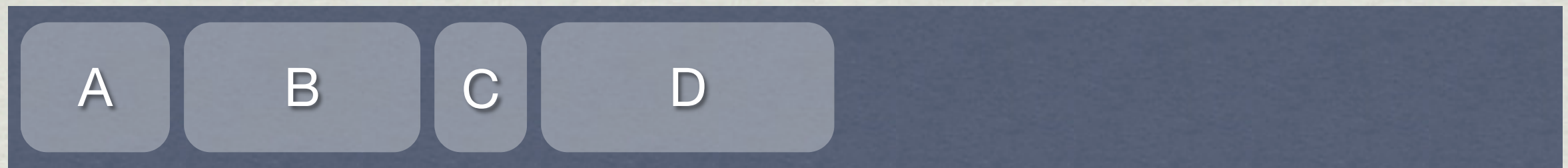
Stack Allocator



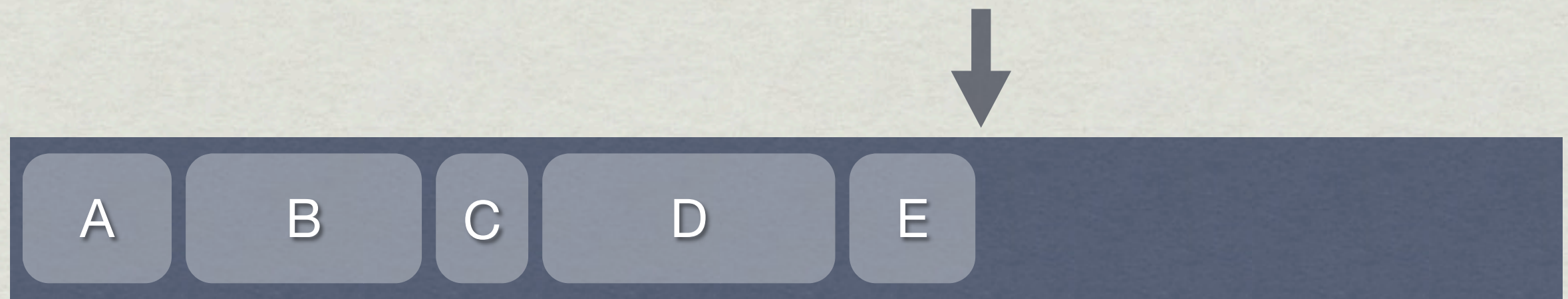
Stack Allocator



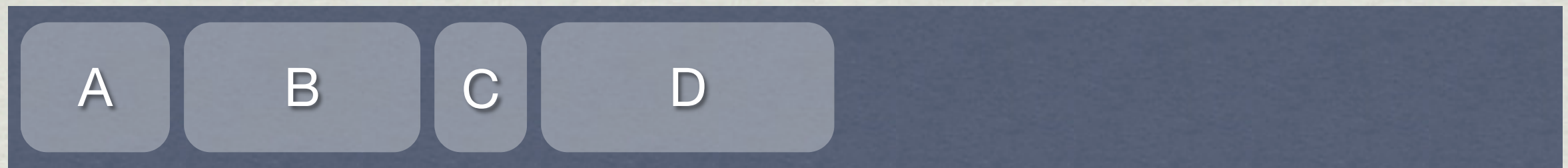
Stack Allocator



Stack Allocator



Stack Allocator



Stack Allocator



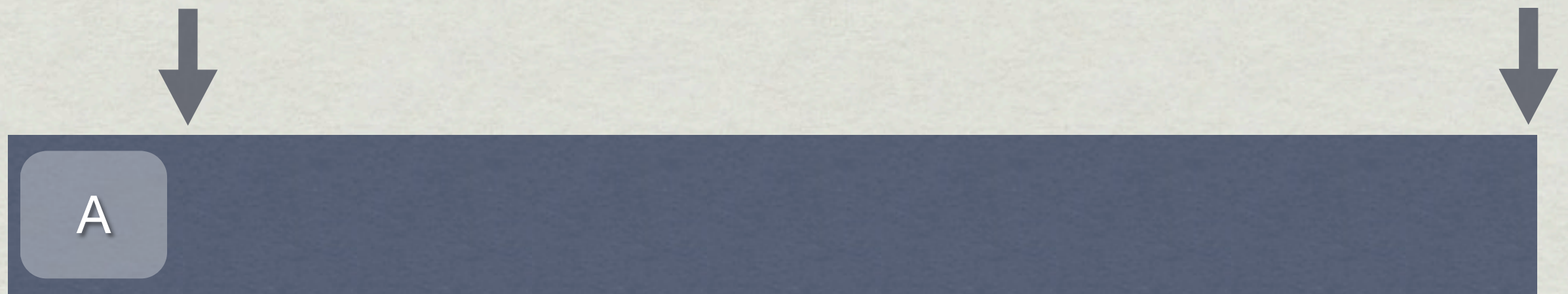
Stack Allocator



Stack Allocator



Stack Allocator



Stack Allocator



Stack Allocator



Stack Allocator



**NO FRAGMENTATION
(AS LONG AS WE ALWAYS
FREE IN REVERSE ORDER)**

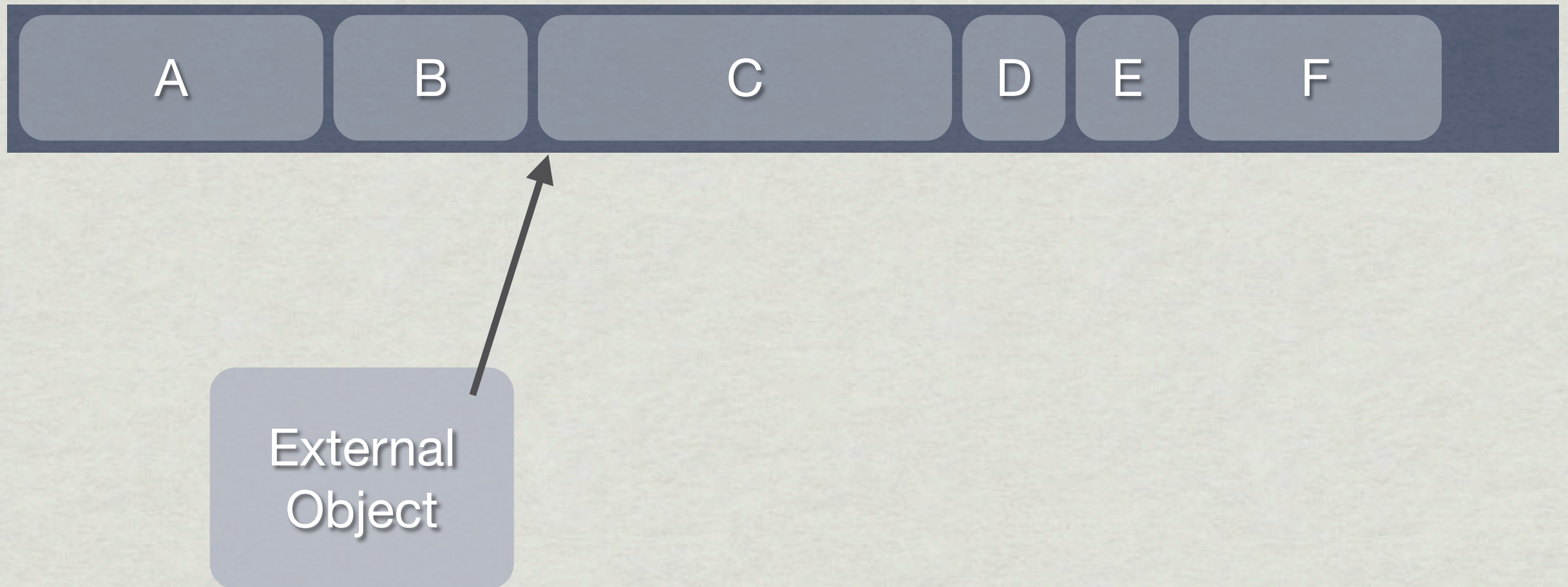
Relocatable Heap



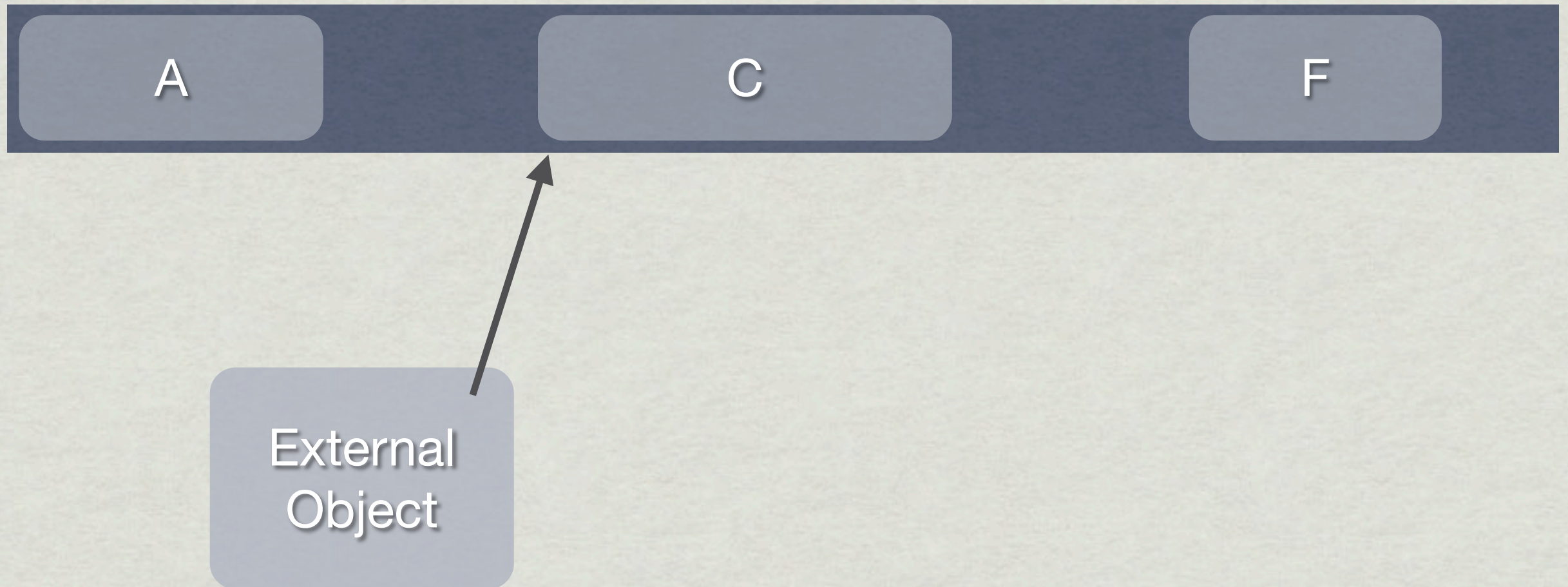
Relocatable Heap



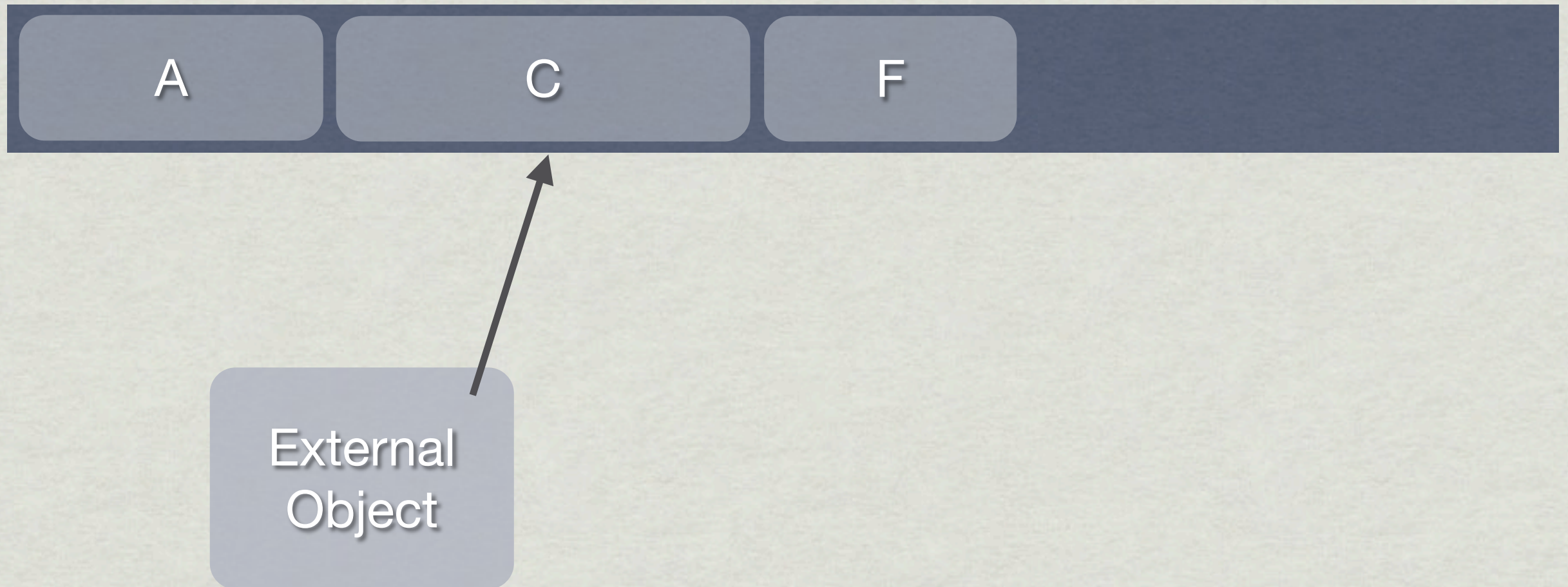
Relocatable Heap



Relocatable Heap



Relocatable Heap



Relocatable Heap



Relocatable Heap



**NO FRAGMENTATION
(THANKS TO RELOCATION)**

Mapping Your Memory

- ✱ We employ an **explicit memory map** to manage and track our allocations

```
MemoryMapEntry g_memoryMap[] = {
    { GLOBAL,          SYSTEM, SIZE_MB(256) },
    { VRAM,            SYSTEM, SIZE_MB(256) },
    { DEBUG_GLOBAL,    SYSTEM, SIZE_MB(128) },
    { DEBUG_VRAM,      SYSTEM, SIZE_MB(128) },

    // GLOBAL allocators
    { PHYSICS,          GLOBAL, SIZE_MB( 5) },
    { OBJECTS,          GLOBAL, SIZE_MB( 32) },
    // ...
};
```


Multicore Hardware

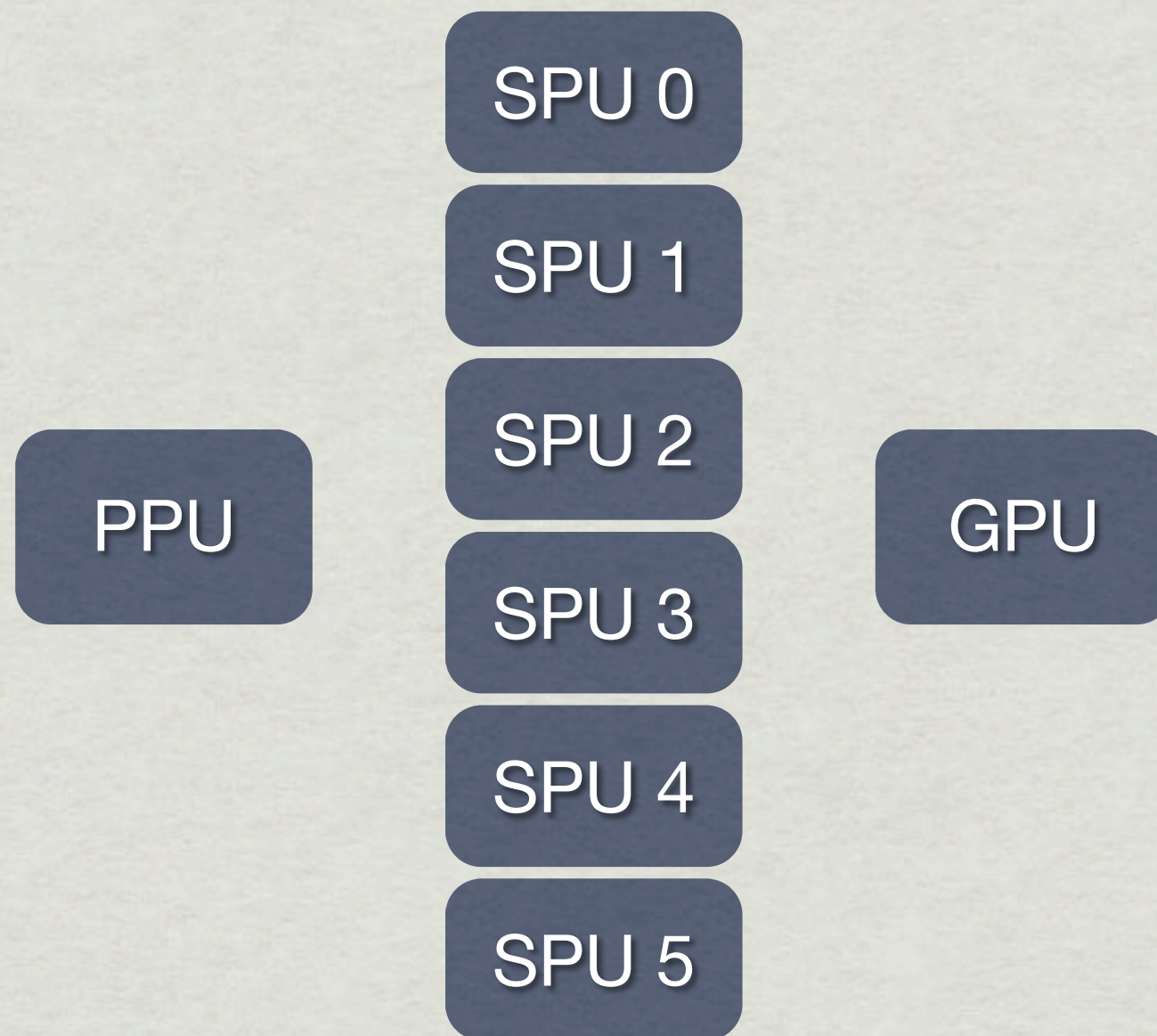
Multicore Hardware

- * PS2, PS3 and PS4 are **multicore** architectures
- * **PS3:** one central CPU (PPU) + 6 synergistic processing units (SPUs) + GPU
- * **PS4:** 8 CPU cores organized into two clusters + GPU with powerful general-purpose compute engine
- * Crucial to take **full advantage** of **all** processing resources!

Multicore: Job System

- * On PS3, we developed a highly efficient job system in conjunction with the ICE team
- * **Job** = input, scratch, and output **buffer(s)** + **code**
- * Jobs are **kicked** by the PPU (or by other jobs) and scheduled to run on the SPU
- * **Gather** results of job(s) later in this frame, or during the next frame
- * Fairly granular -- thousands of jobs / frame

PS3 Job System



PS3 Job System



PS3 Job System

PPU

SPU 0

SPU 1

SPU 2

SPU 3

SPU 4

SPU 5

PS3 Job System

PPU

Main Thread

SPU 0

SPU 1

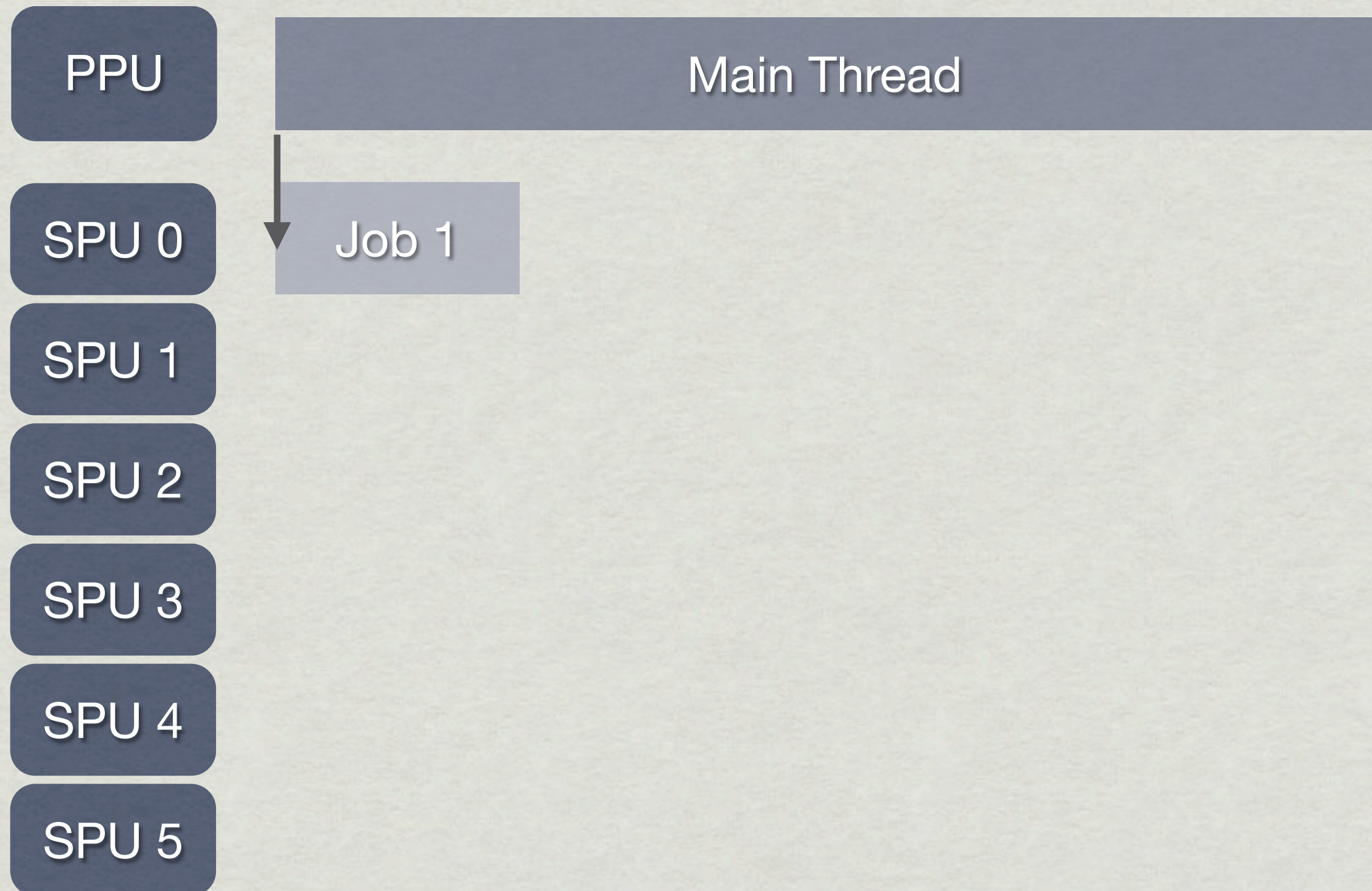
SPU 2

SPU 3

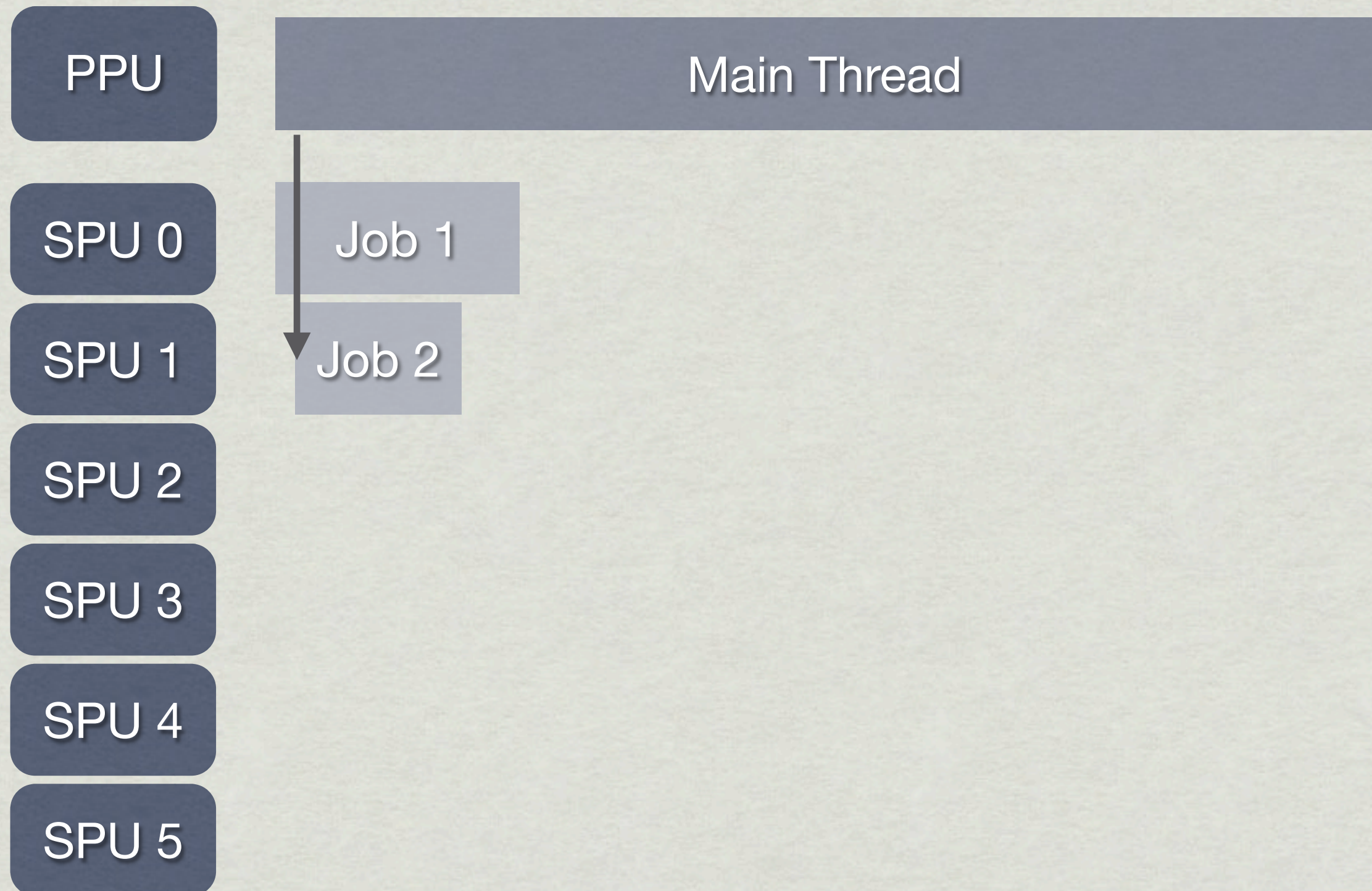
SPU 4

SPU 5

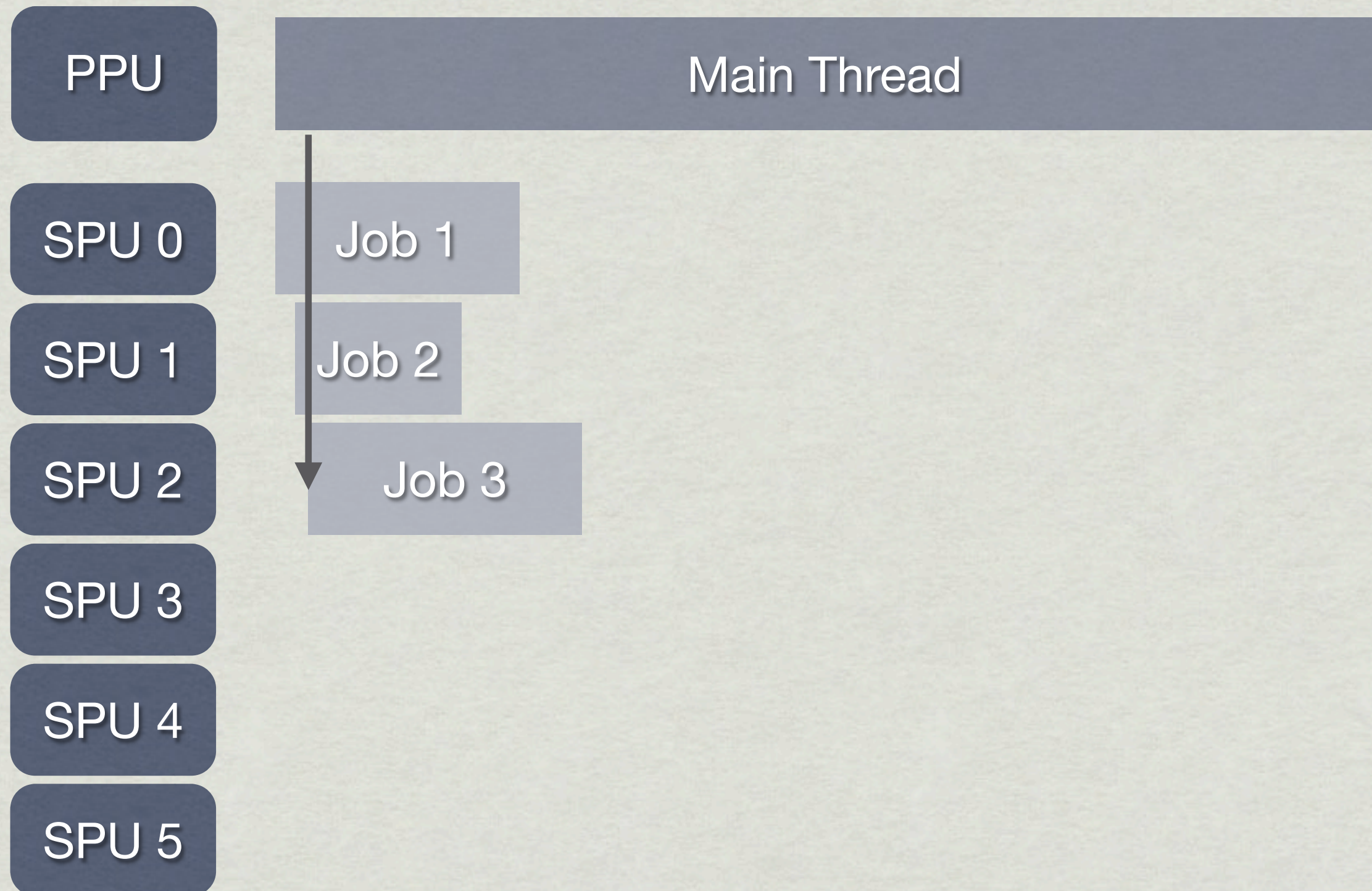
PS3 Job System



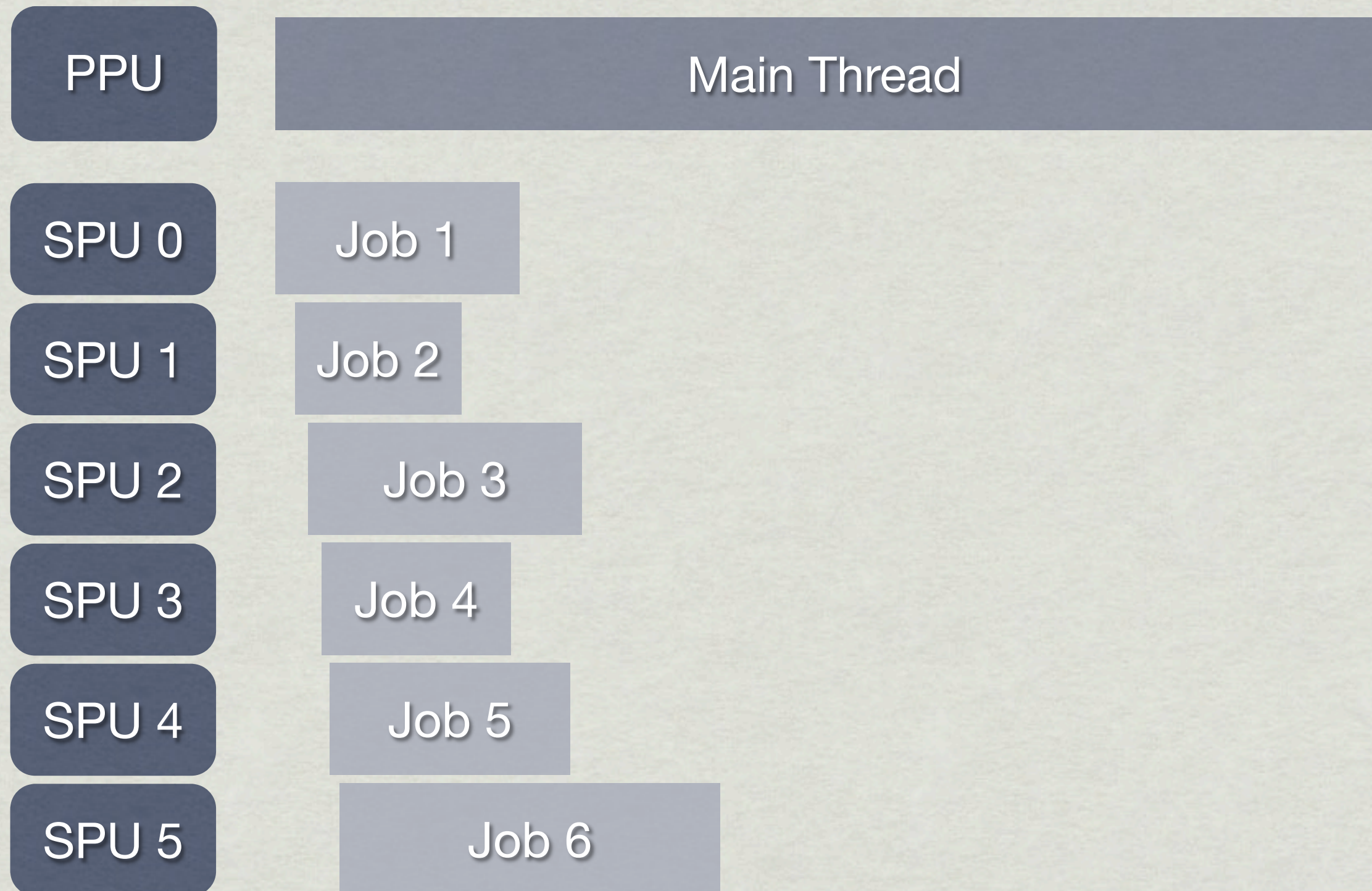
PS3 Job System



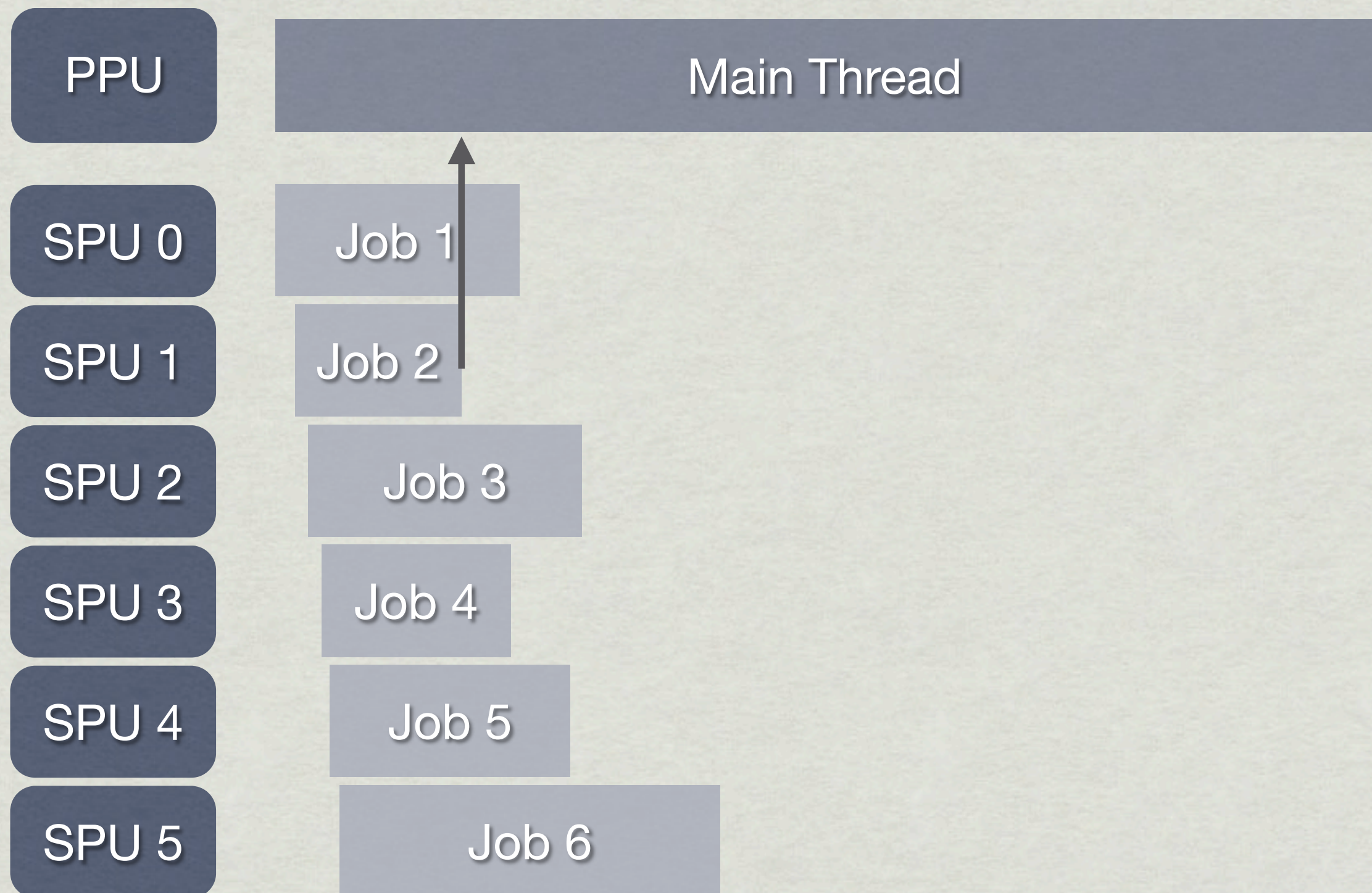
PS3 Job System



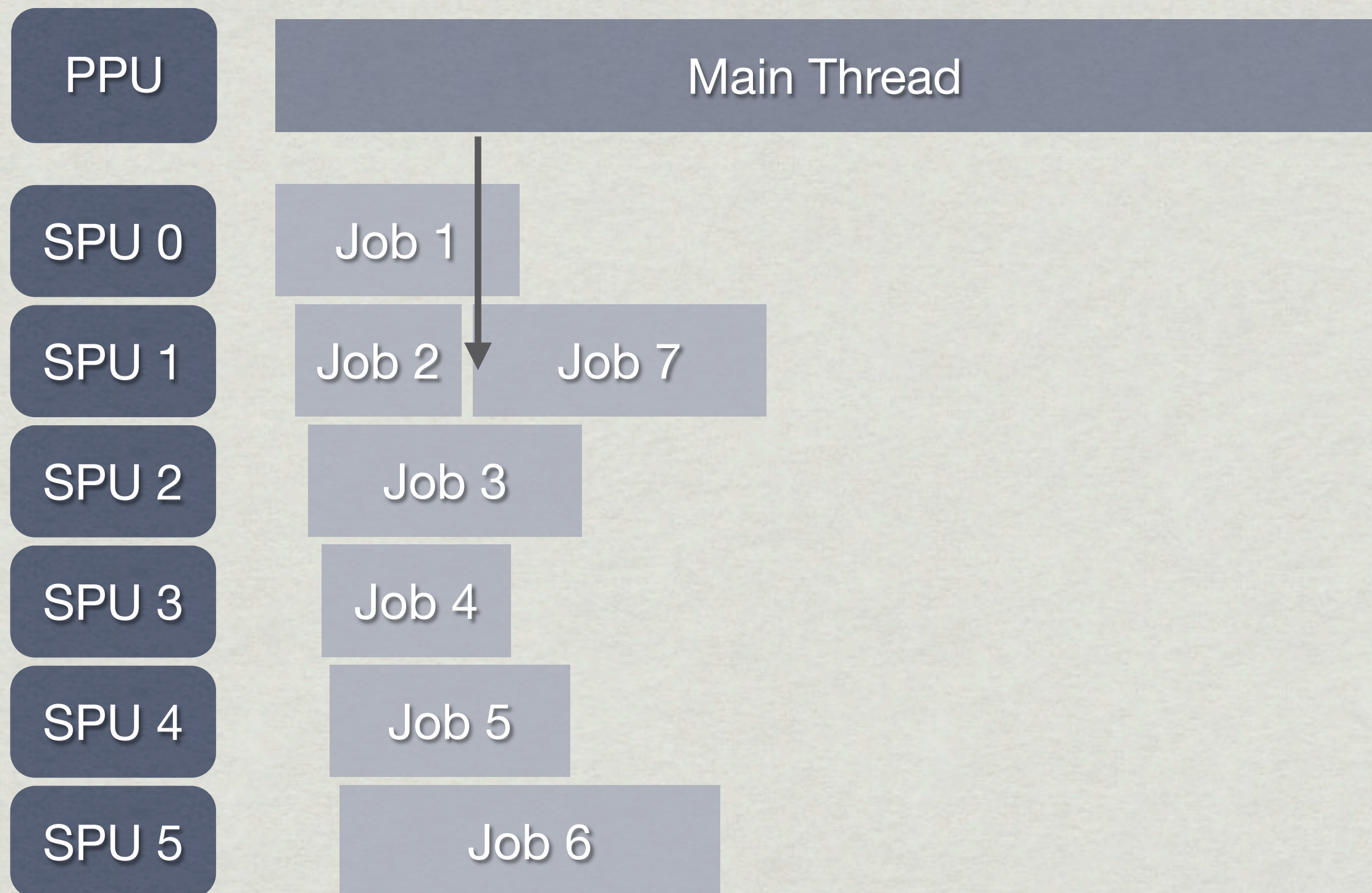
PS3 Job System



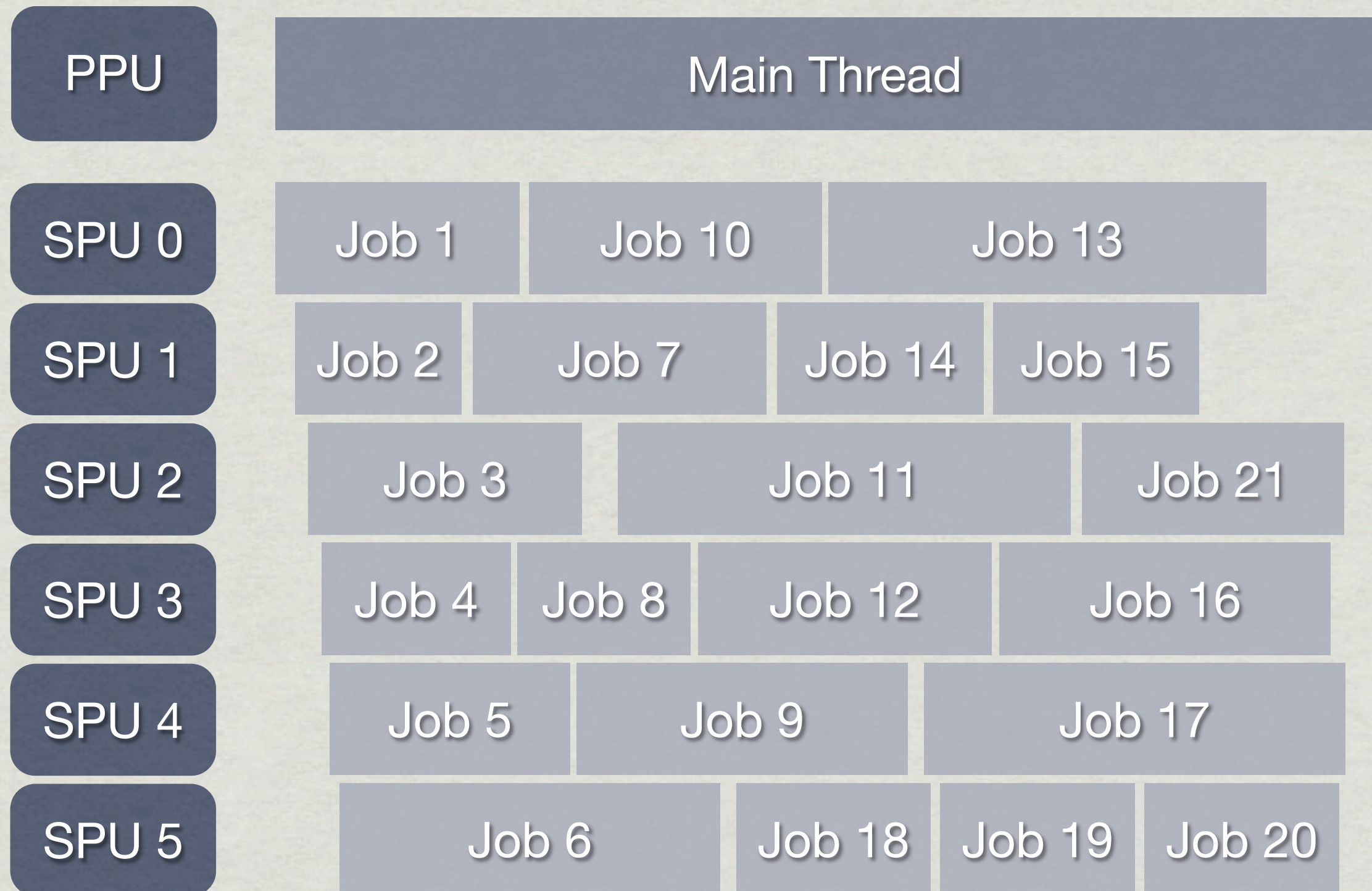
PS3 Job System



PS3 Job System



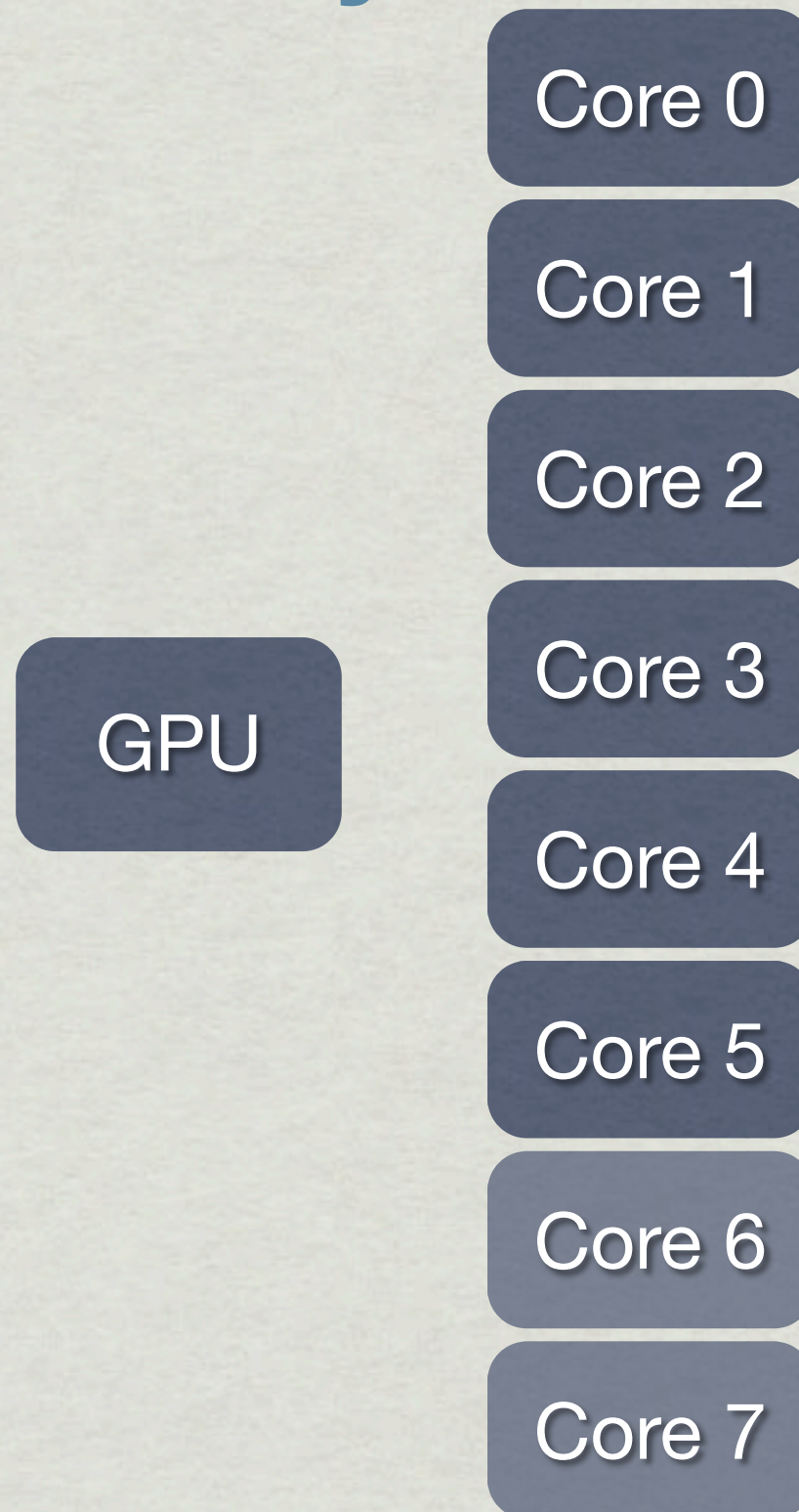
PS3 Job System



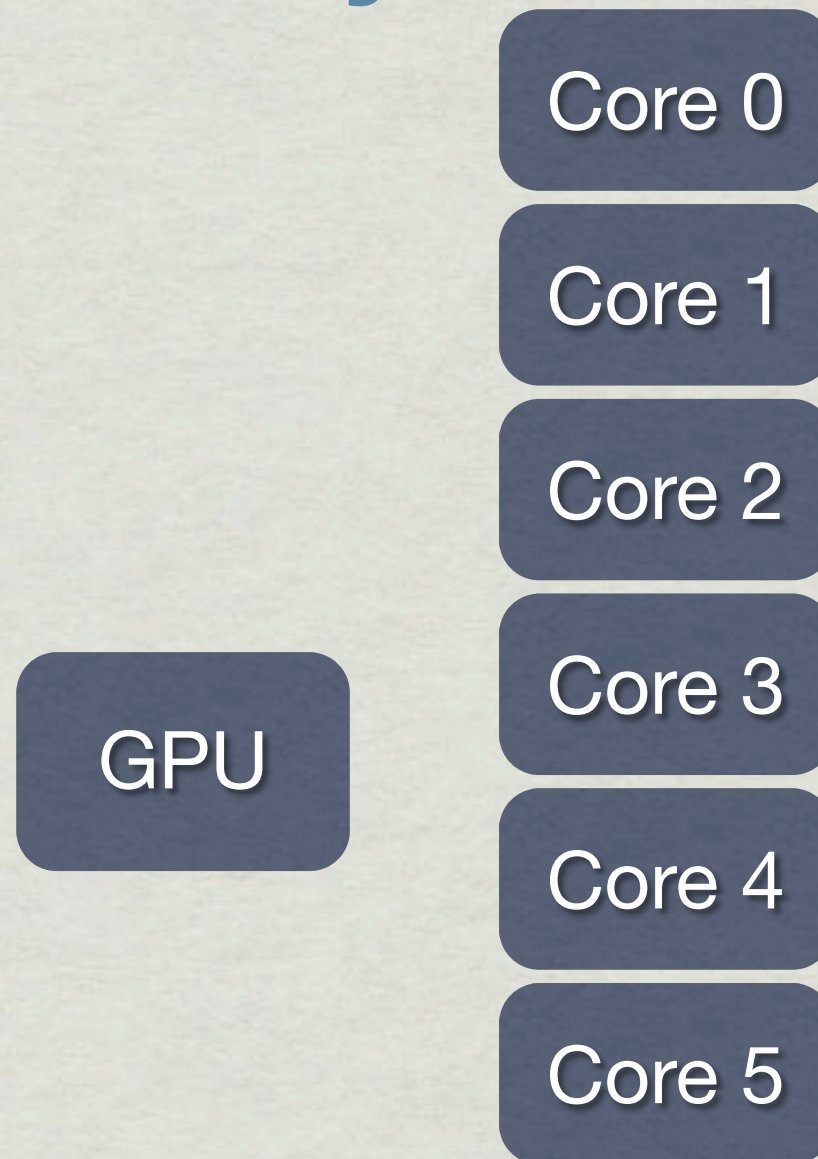
Multicore: Job System

- * On PS4 we implemented a new job system
 - * Similar concept to SPU job system on PS3
- * **6 CPUs, 6 worker threads**
 - * Jobs are **kicked**, picked up by available worker
 - * Each job acts like a lightweight **fiber**
 - * **Shared memory**, but can retain input, scratch and output buffer(s) for legacy code migration

PS4 Job System



PS4 Job System



PS4 Job System

GPU

Core 0

Core 1

Core 2

Core 3

Core 4

Core 5

PS4 Job System

GPU

Core 0 Worker Thread 0

Core 1 Worker Thread 1

Core 2 Worker Thread 2

Core 3 Worker Thread 3

Core 4 Worker Thread 4

Core 5 Worker Thread 5

PS4 Job System

GPU

Core 0

Job 1: Main Game Loop

Core 1

Core 2

Core 3

Core 4

Core 5

PS4 Job System

GPU

Core 0

Job 1: Main Game Loop

Core 1

Job 2

Job 7

Job 12

Job 13

Core 2

Job 3

Job 10

Job 19

Core 3

Job 4

Job 8

Job 11

Job 14

Core 4

Job 5

Job 9

Job 15

Core 5

Job 6

Job 16

Job 17

Job 18

PS4 Job System

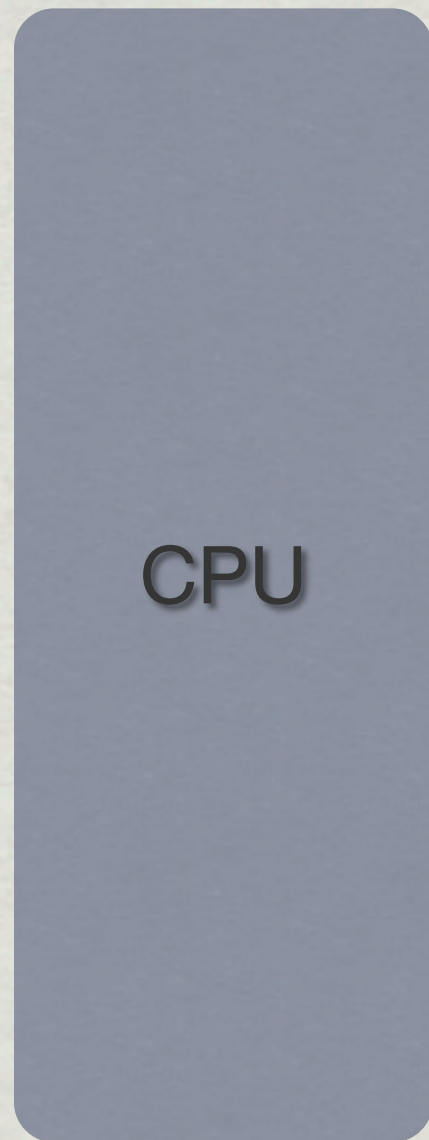


Optimization

Understand Your Hardware

- ✱ The secret to writing highly efficient code is **mastery of your hardware** (PlayStation1 through PlayStation4)
- ✱ Memory caching and its implications
- ✱ Superscalar CPU architecture and pipelining
- ✱ Branch prediction, data dependencies, load-hit-store, ...
- ✱ Assembly language downcoding
- ✱ Optimization of data layouts as well as code

Memory Caching



Memory Caching



CPU

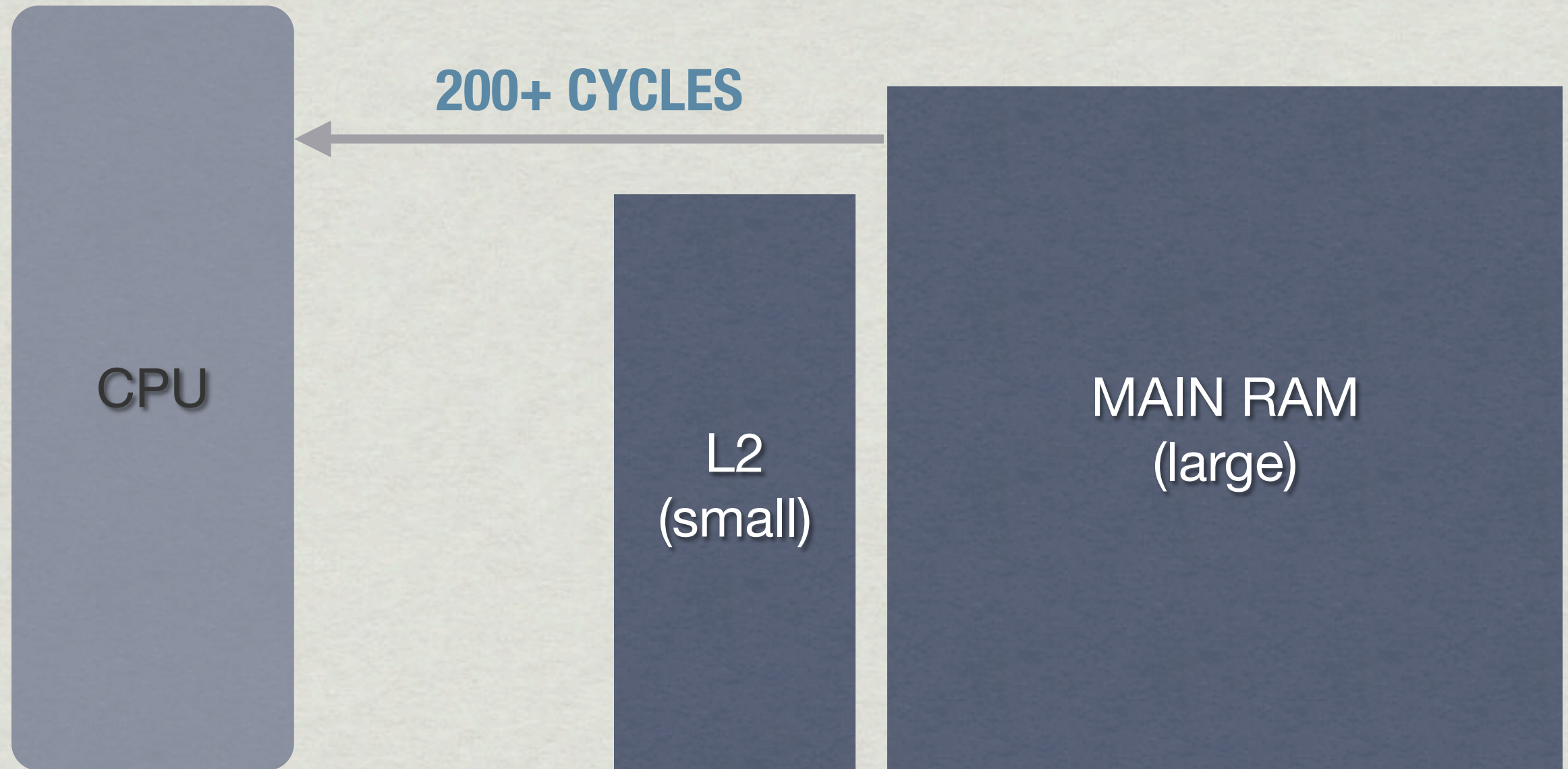
The diagram consists of two main components: a light blue rounded rectangle on the left labeled 'CPU' and a dark blue square on the right labeled 'MAIN RAM (large)'. The CPU rectangle is positioned on the left side of the slide, and the RAM square is on the right side. There are no connecting lines or arrows between them.

MAIN RAM
(large)

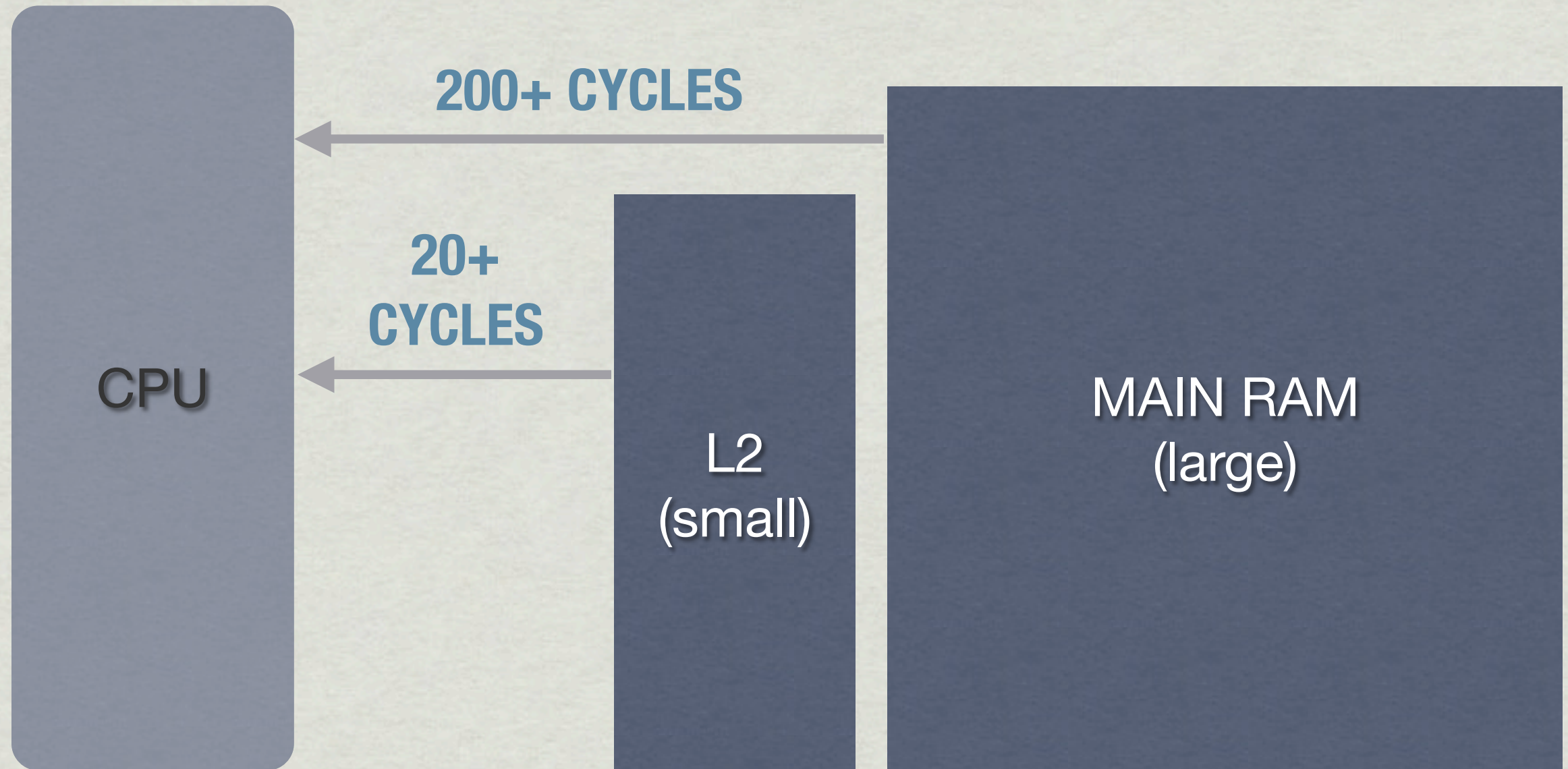
Memory Caching



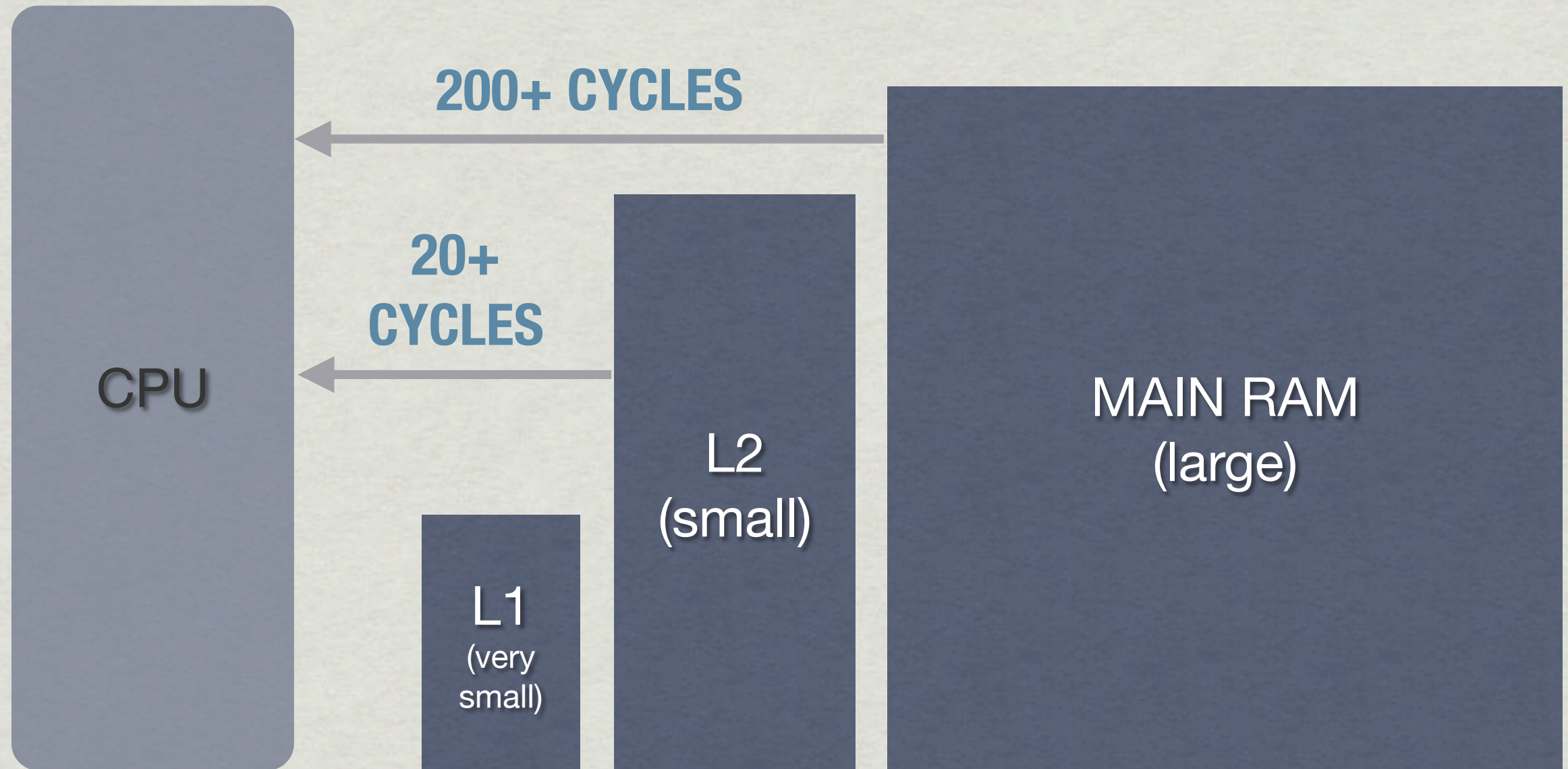
Memory Caching



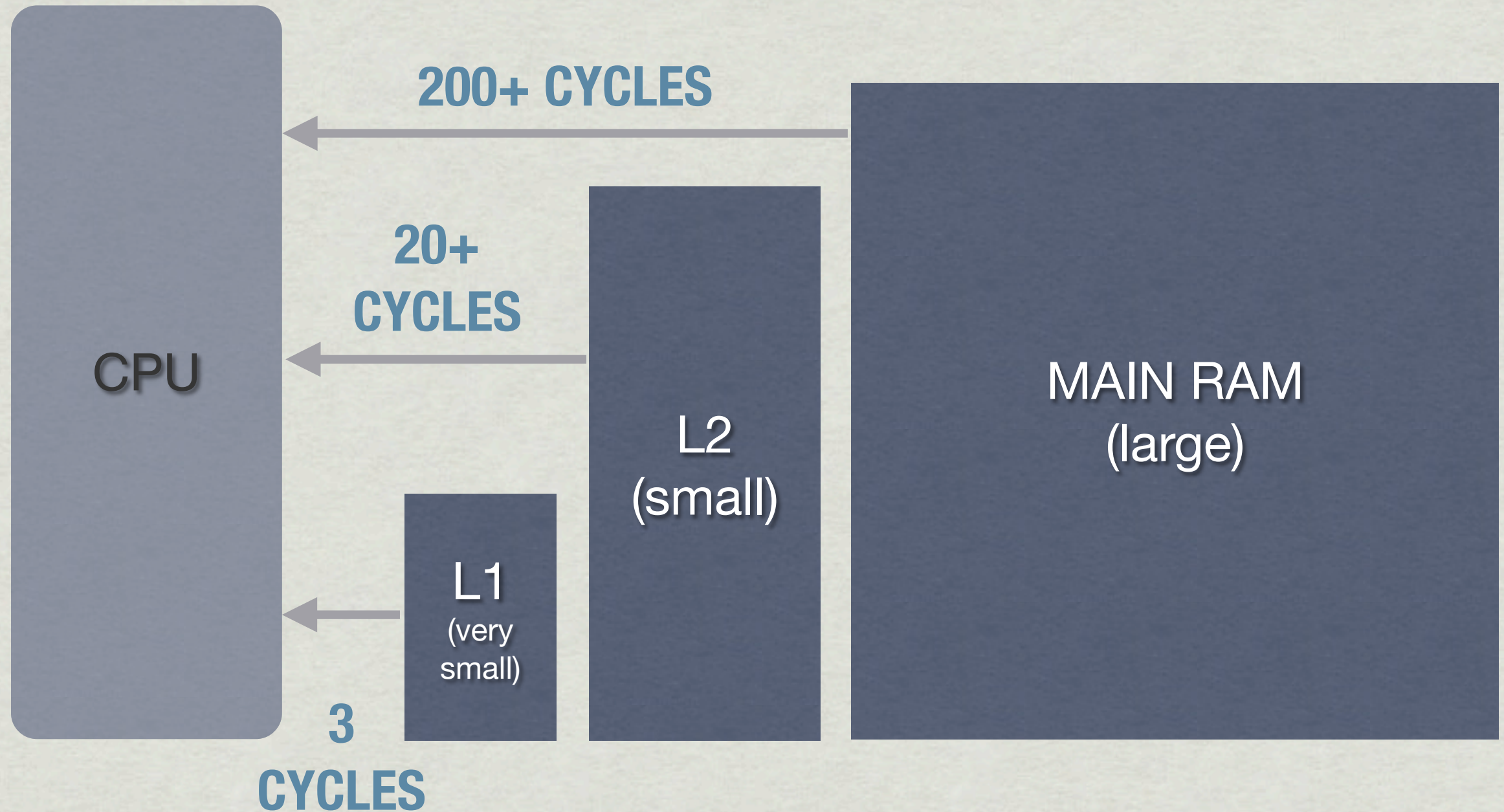
Memory Caching



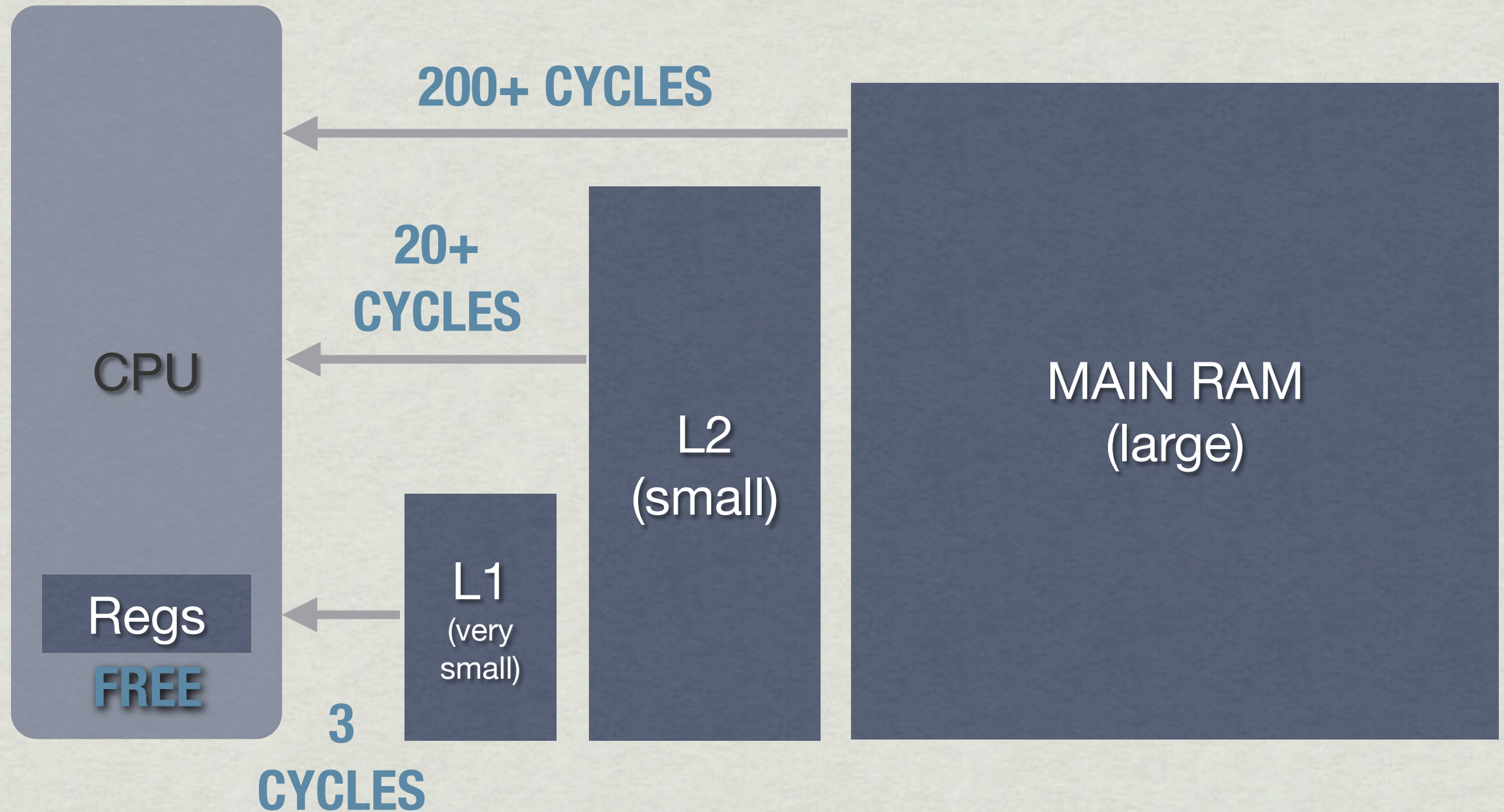
Memory Caching



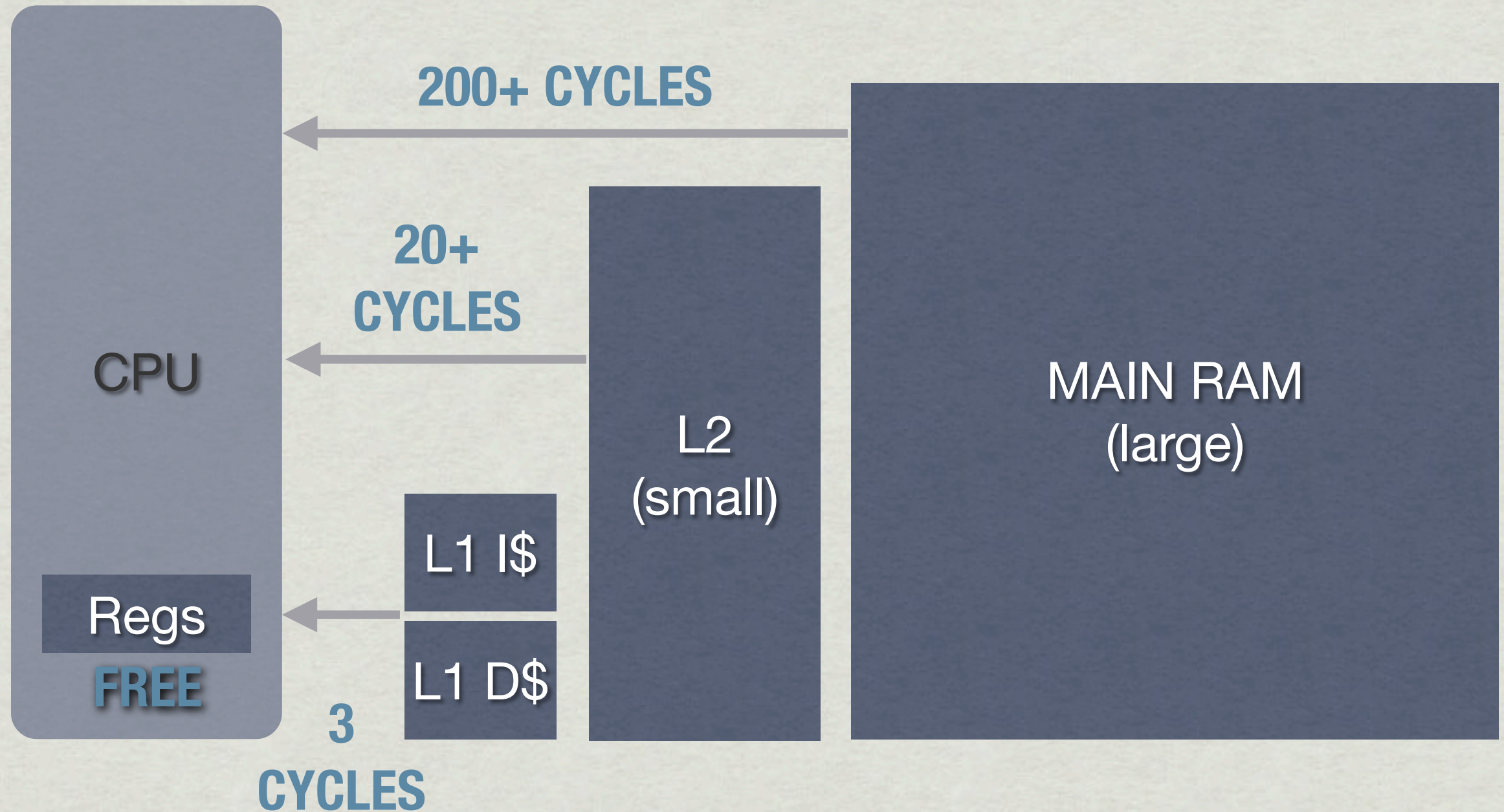
Memory Caching



Memory Caching



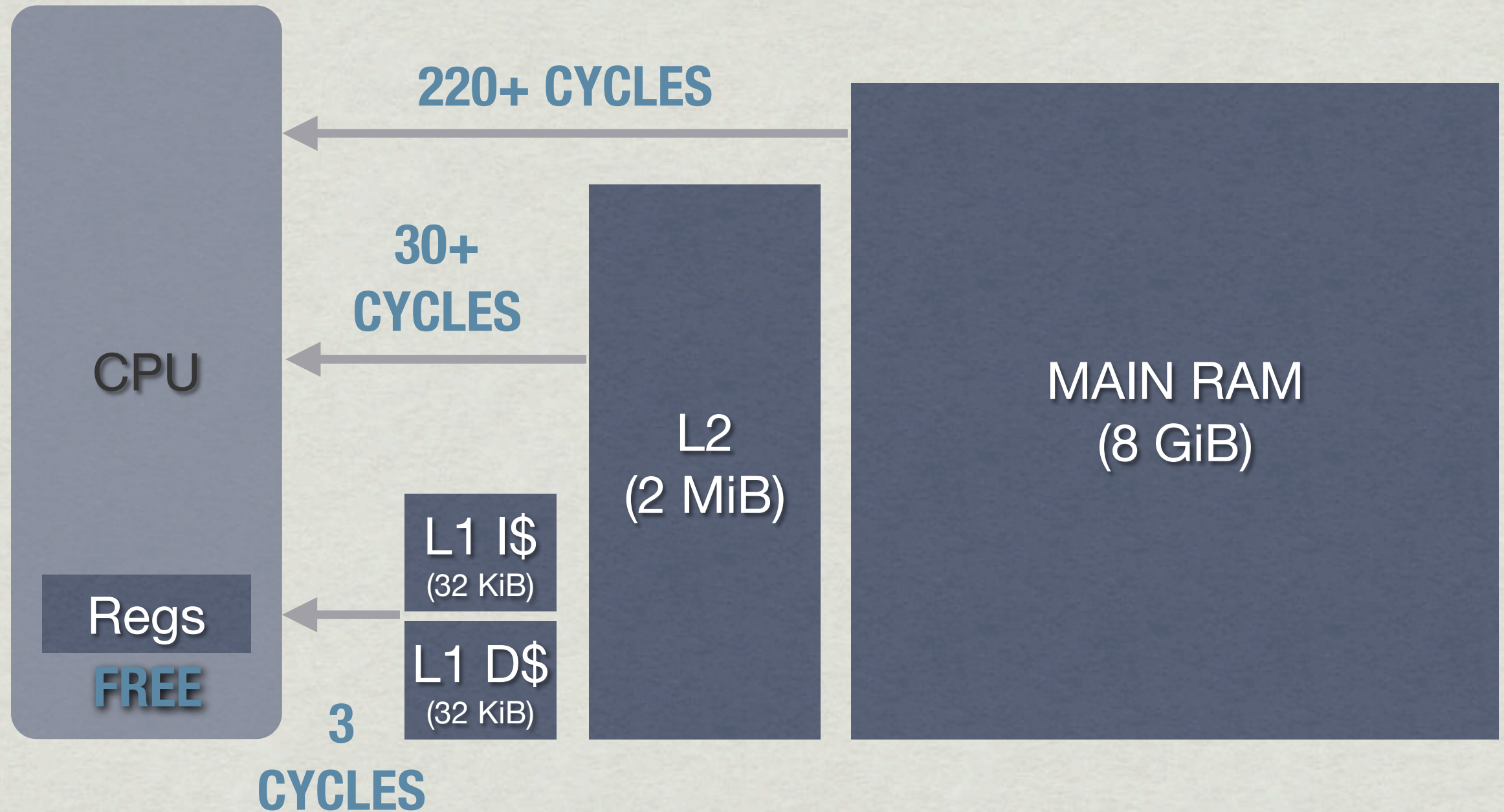
Memory Caching



Memory Caching

- * Understanding the cache allows you to **optimize**
 - * Don't forget the 80/20 rule
- * General rules of thumb:
 - * Keep high-performance code small (fit in I\$)
 - * Keep high-performance data small and contiguous (fit in D\$)

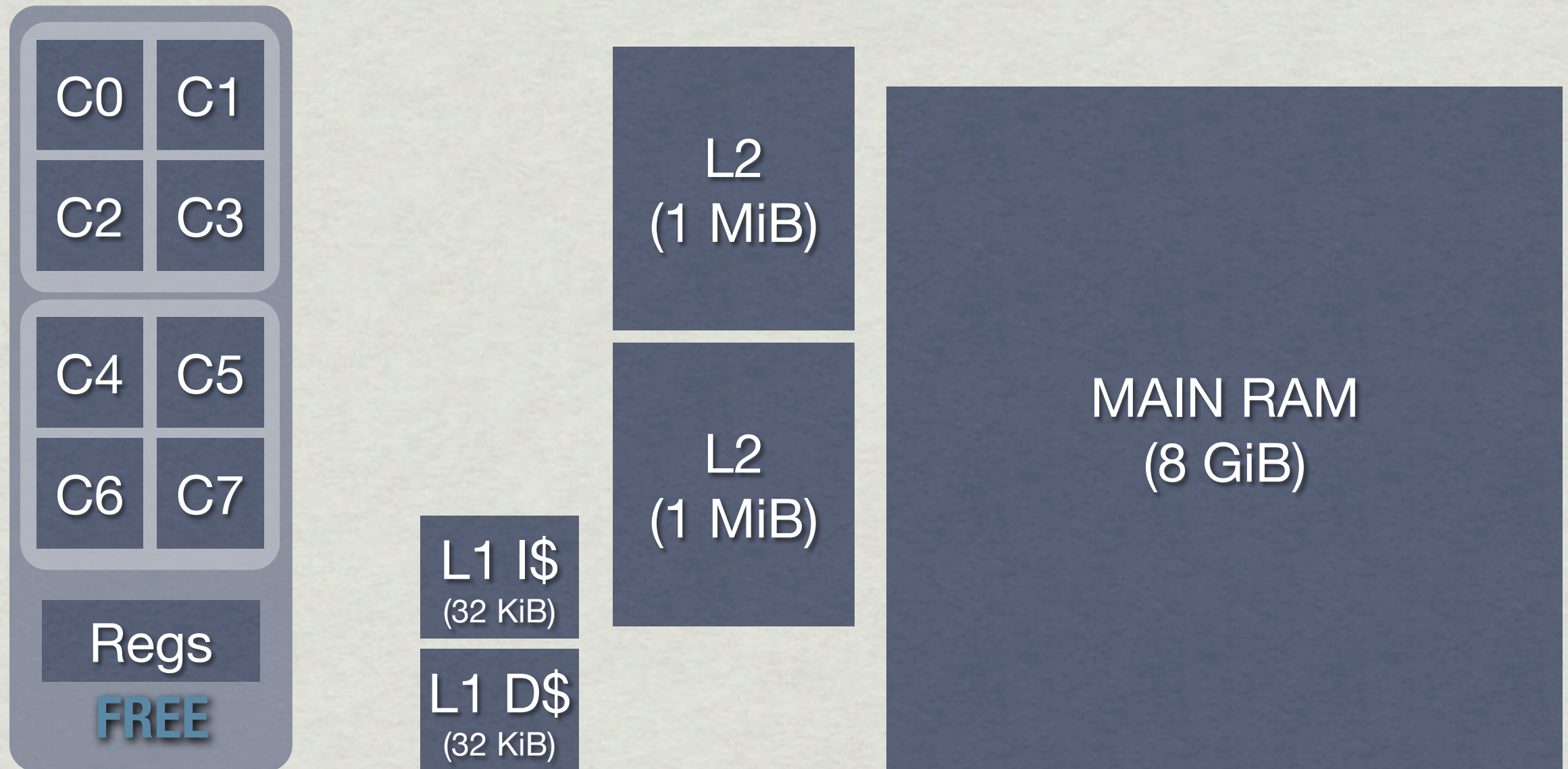
PS4 Cache Architecture



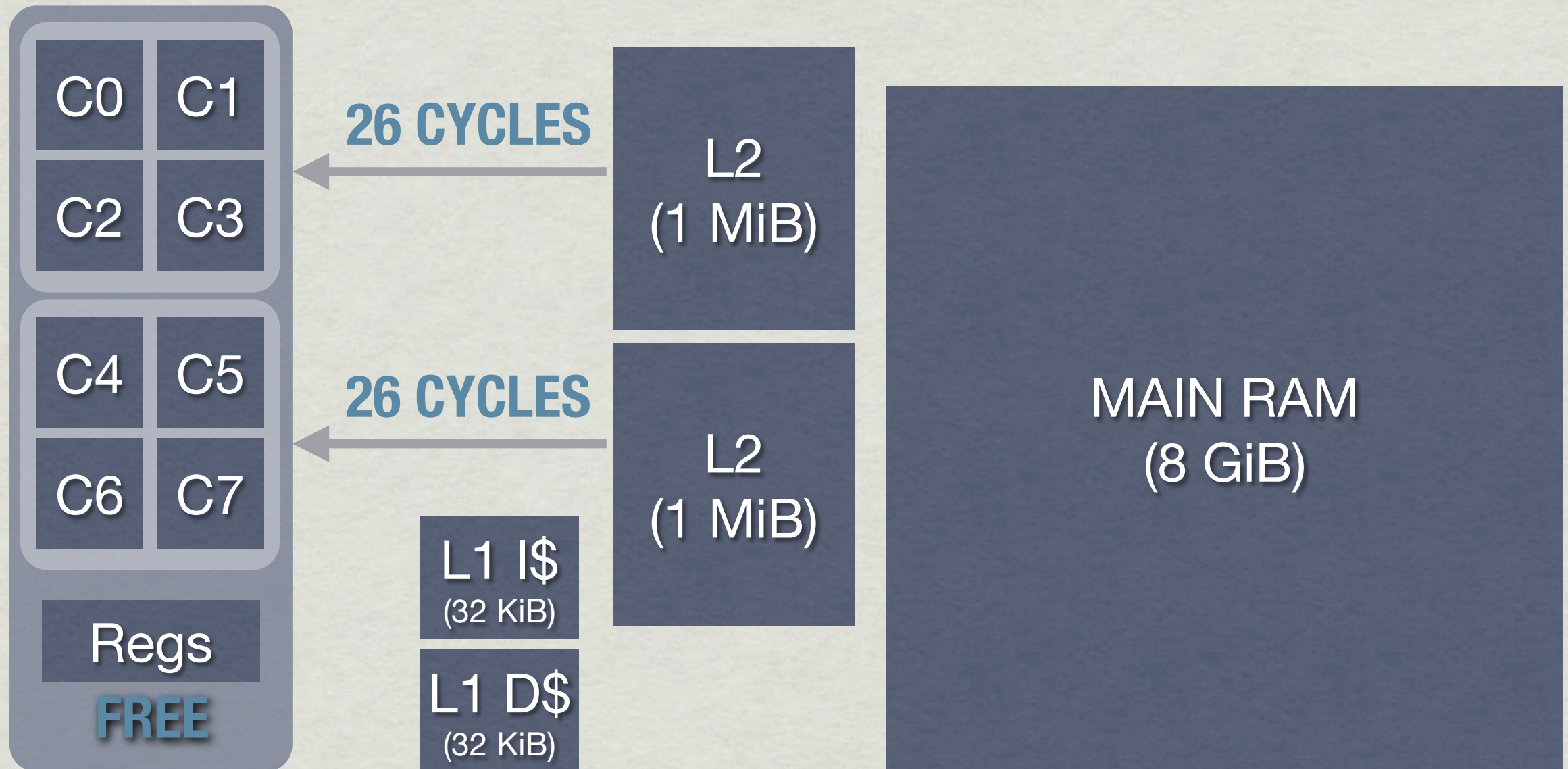
PS4 Cache Architecture



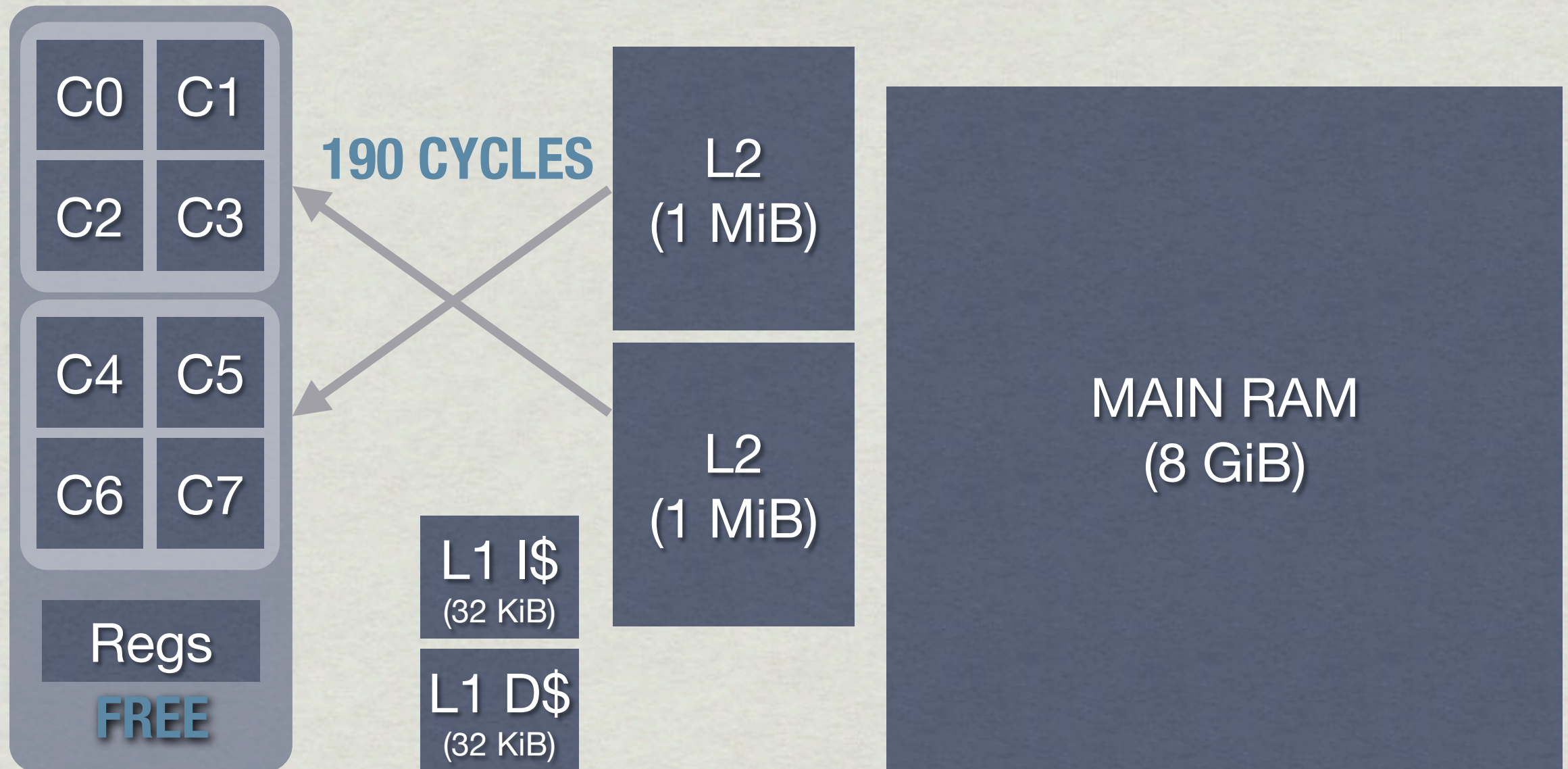
PS4 Cache Architecture



PS4 Cache Architecture



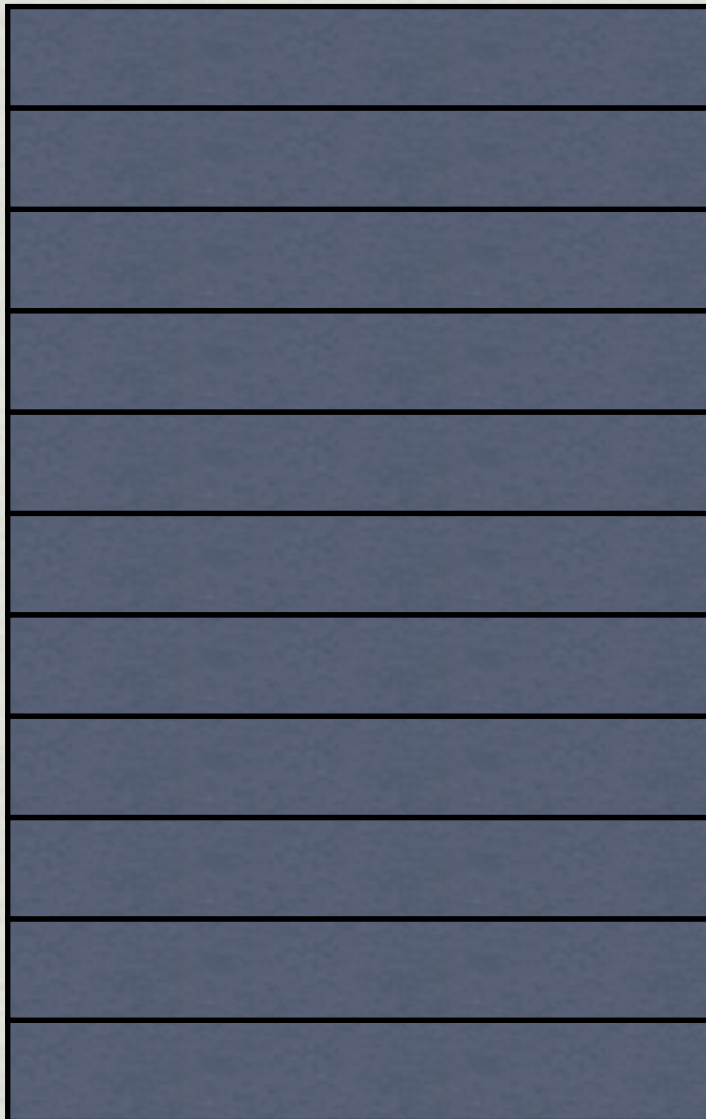
PS4 Cache Architecture



PS4 Cache Architecture

MAIN RAM

0x5280
0x5240
0x5200
0x51C0
0x5180
0x5140
0x5100
0x50C0
0x5080
0x5040
0x5000



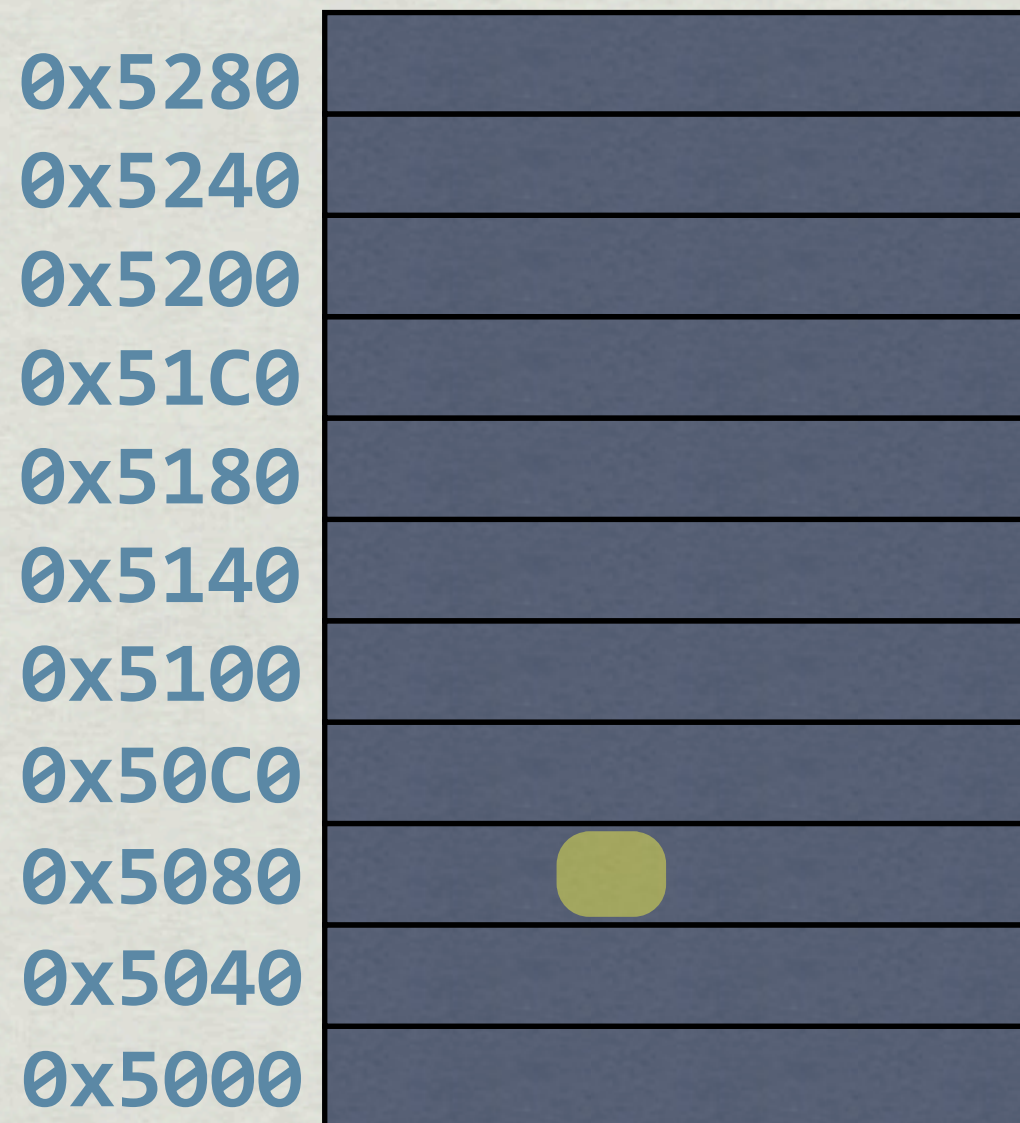
CACHE

0x0280
0x0240
0x0200
0x01C0
0x0180
0x0140
0x0100
0x00C0
0x0080
0x0040
0x0000

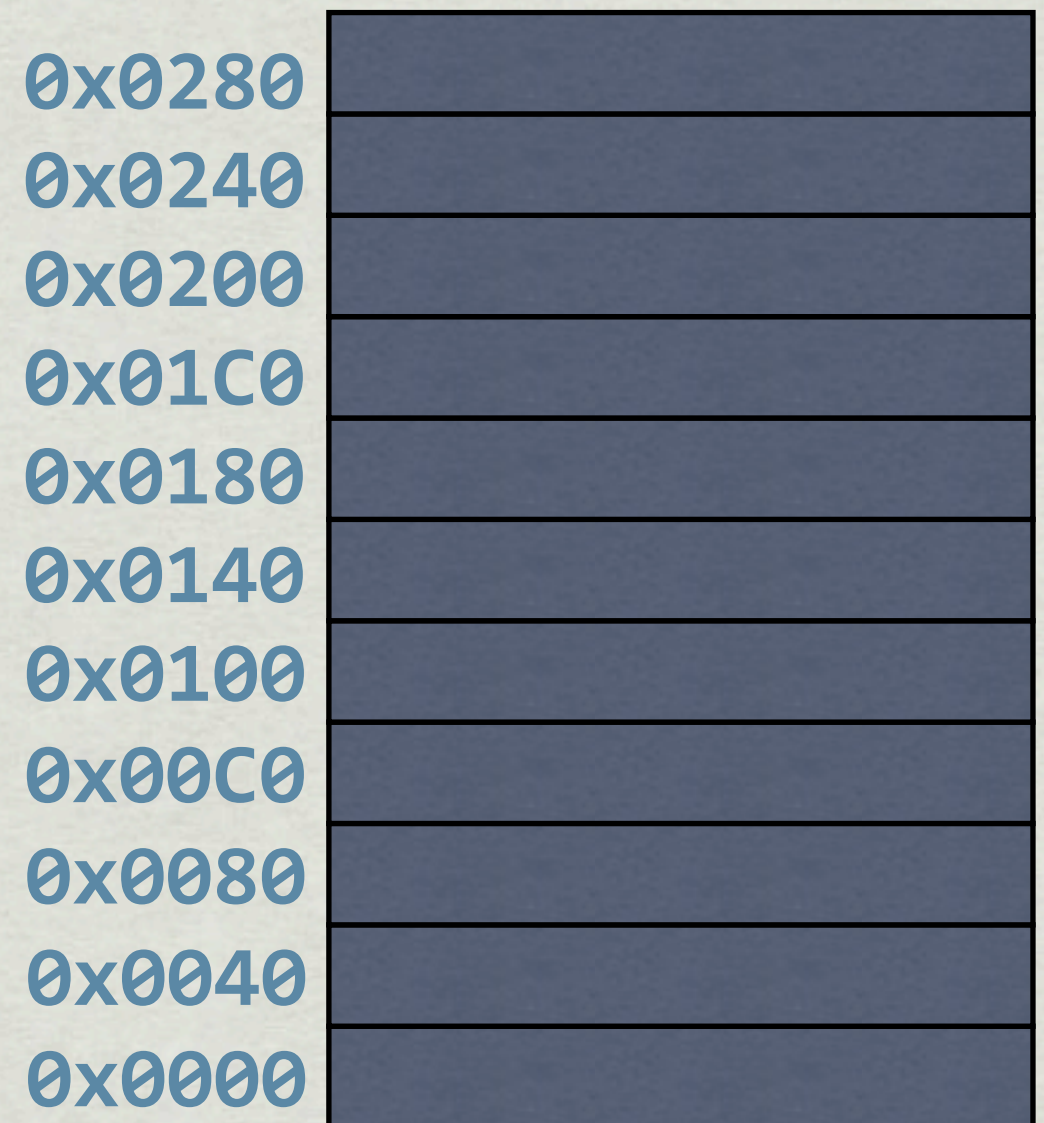


PS4 Cache Architecture

MAIN RAM



CACHE

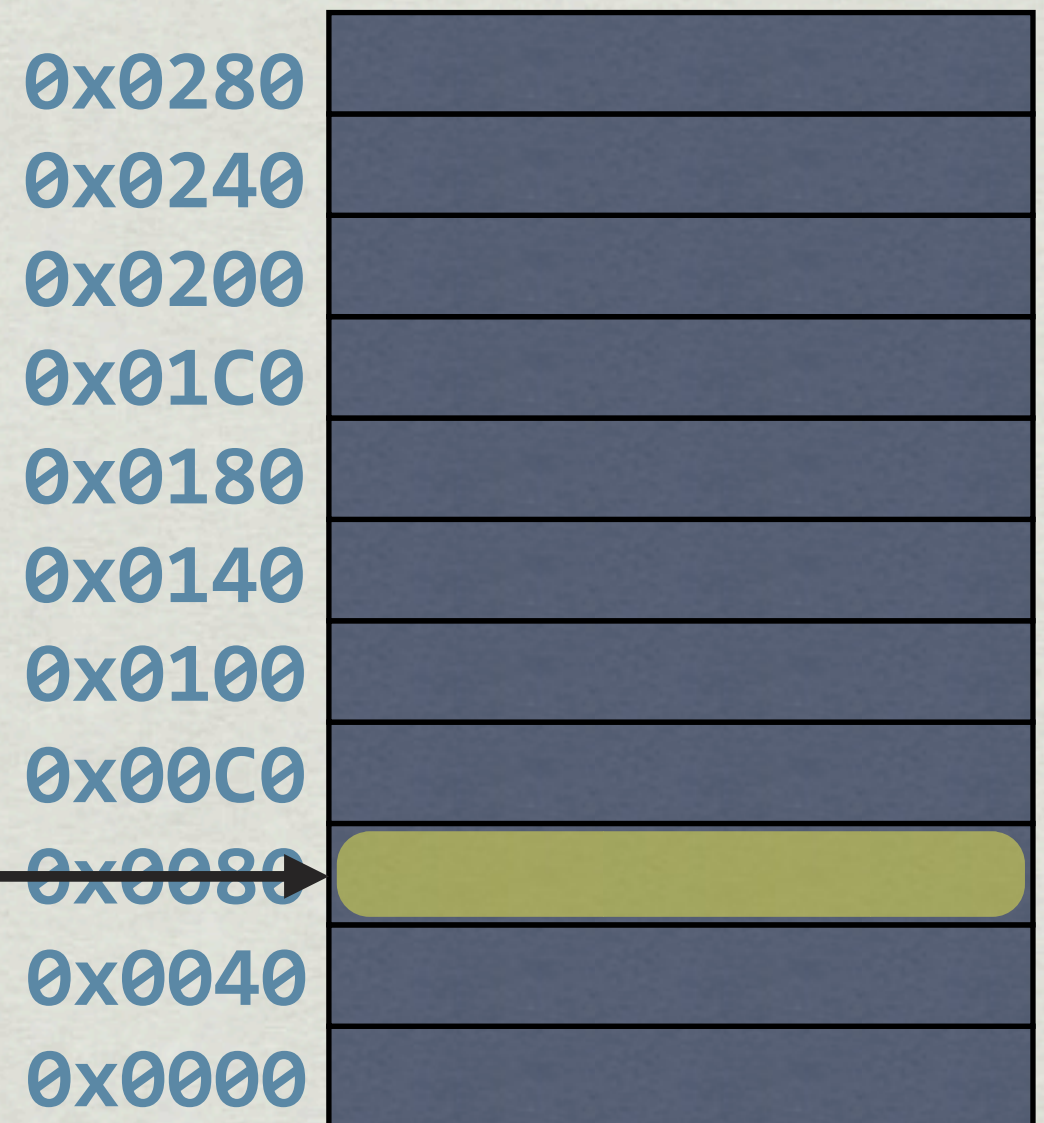


PS4 Cache Architecture

MAIN RAM



CACHE



PS4 Optimization

- ✱ ***PS4-specific:*** avoid cross-cluster L2 cache line sharing (190 cycles versus 26 cycles)!

```
U32 g_jobCount[6]; // one per core
```


PS4 Optimization

- ✱ ***PS4-specific:*** avoid cross-cluster L2 cache line sharing (190 cycles versus 26 cycles)!

```
struct JobCount
{
    U32 m_count;
    U8  m_padding[60];
};
JobCount g_jobCount[6]; // one per core
```


Pipelined CPU



Pipelined CPU

INSTRUCTION FETCH



Pipelined CPU

**INSTRUCTION DECODE/
REGISTER FETCH**



Pipelined CPU

EXECUTION



Pipelined CPU

MEMORY ACCESS



Pipelined CPU



Pipelined CPU



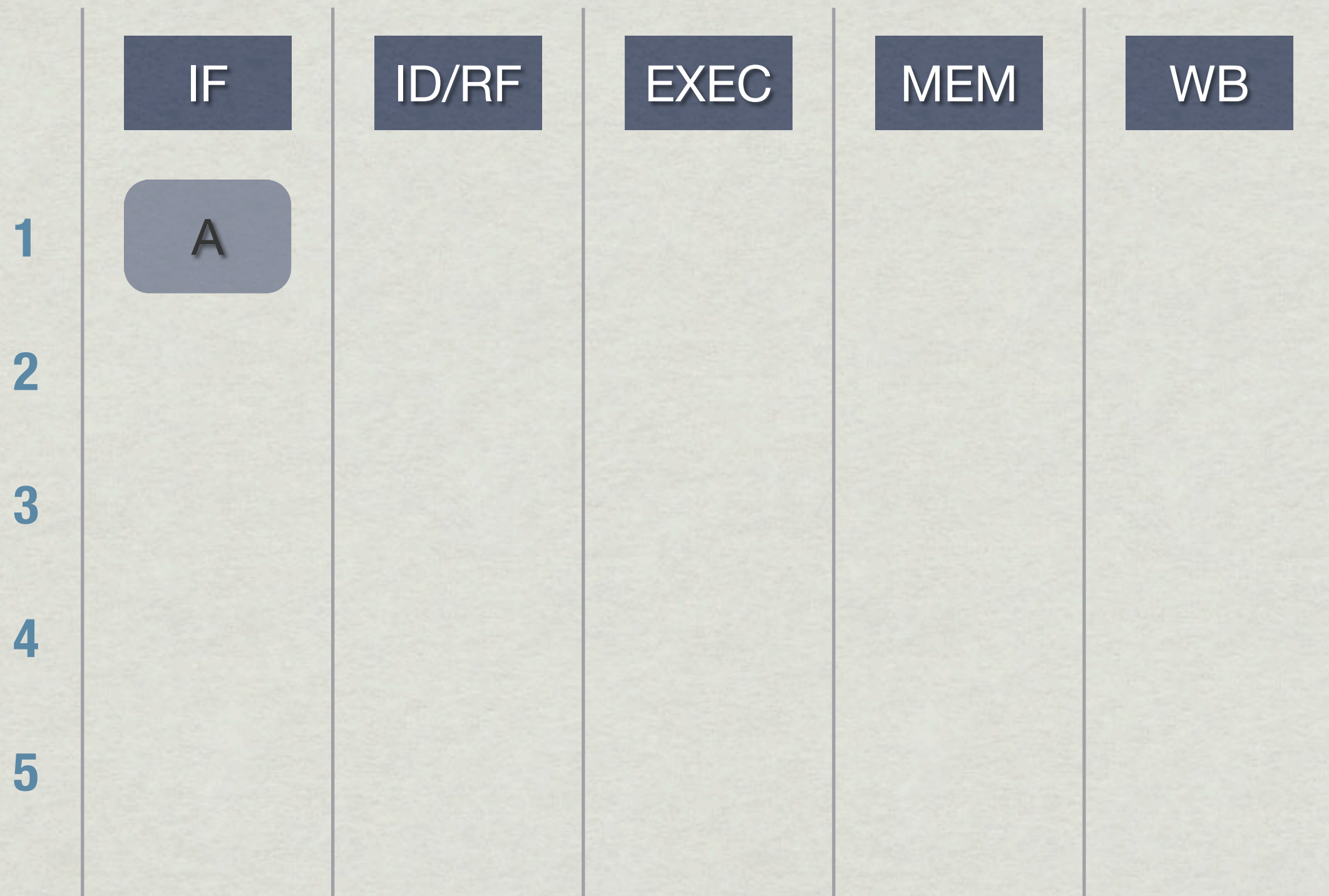
Pipelined CPU



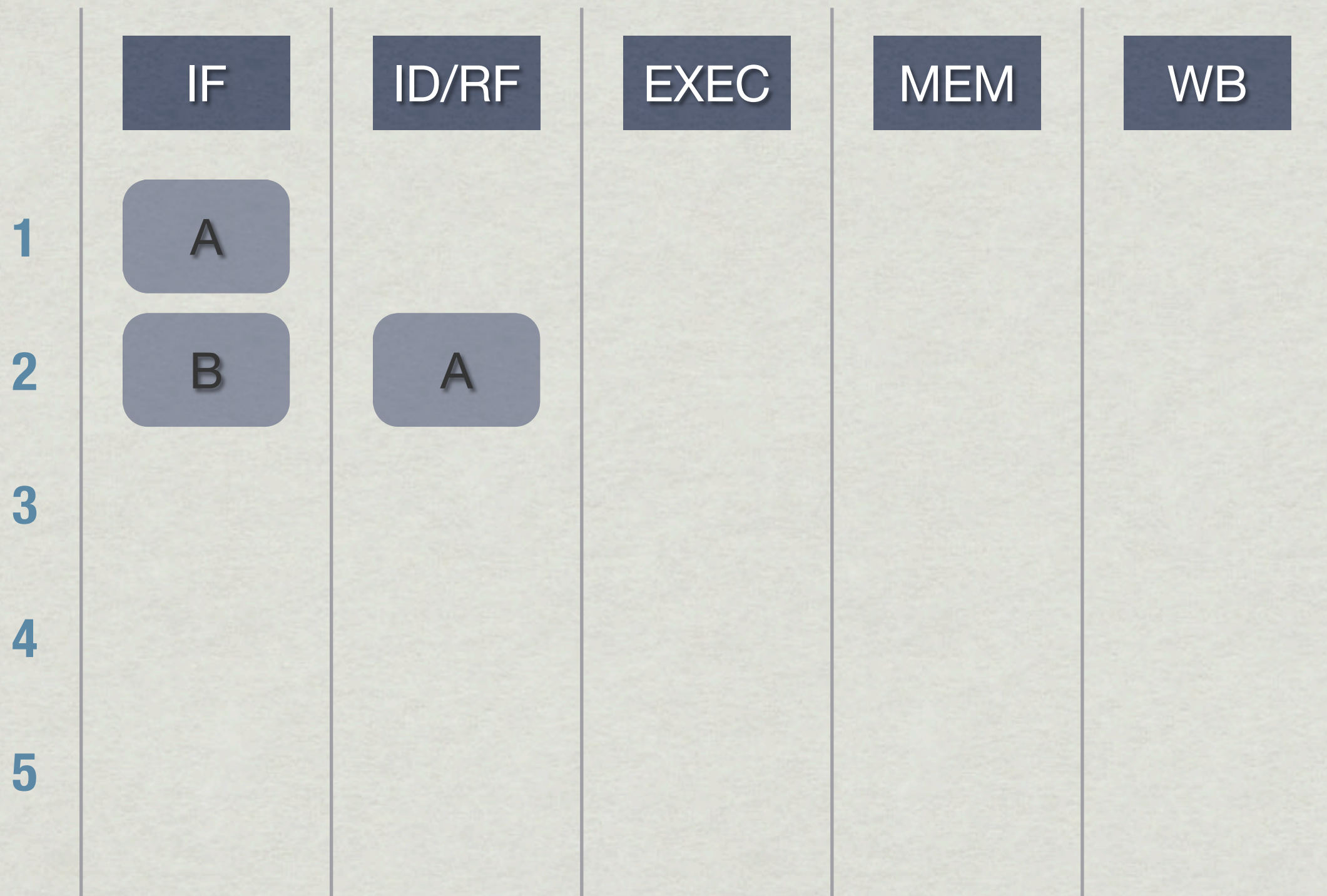
Pipelined CPU



Pipelined CPU



Pipelined CPU



Pipelined CPU



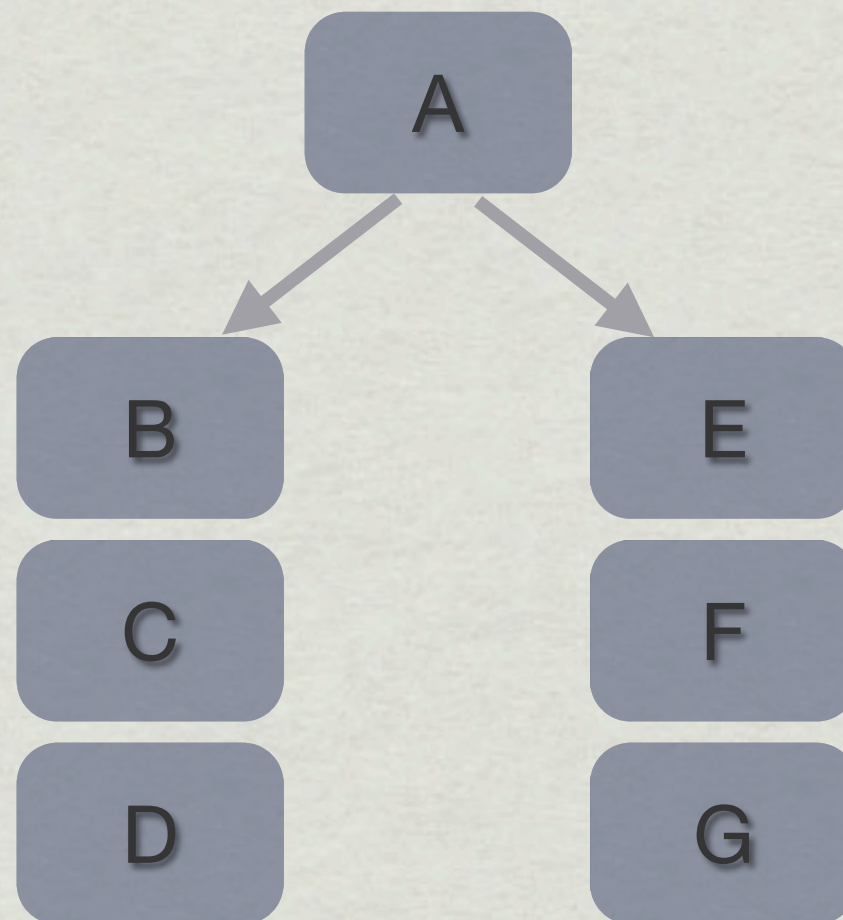
Pipelined CPU



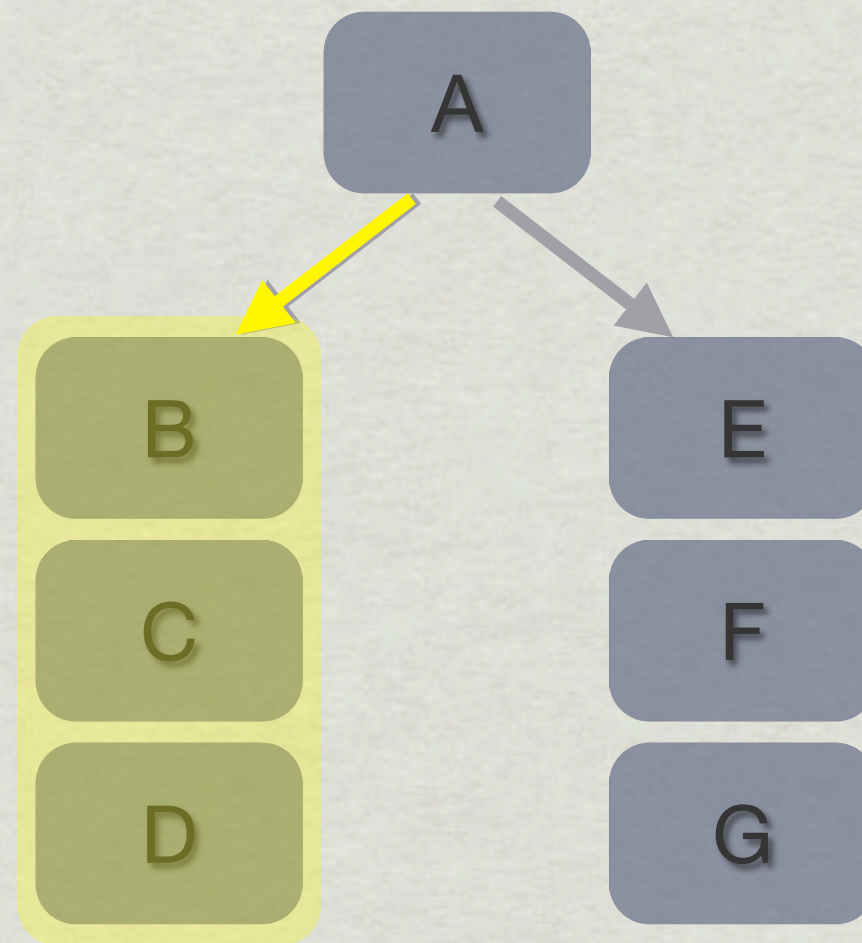
Pipelined CPU



Mispredicted Branch



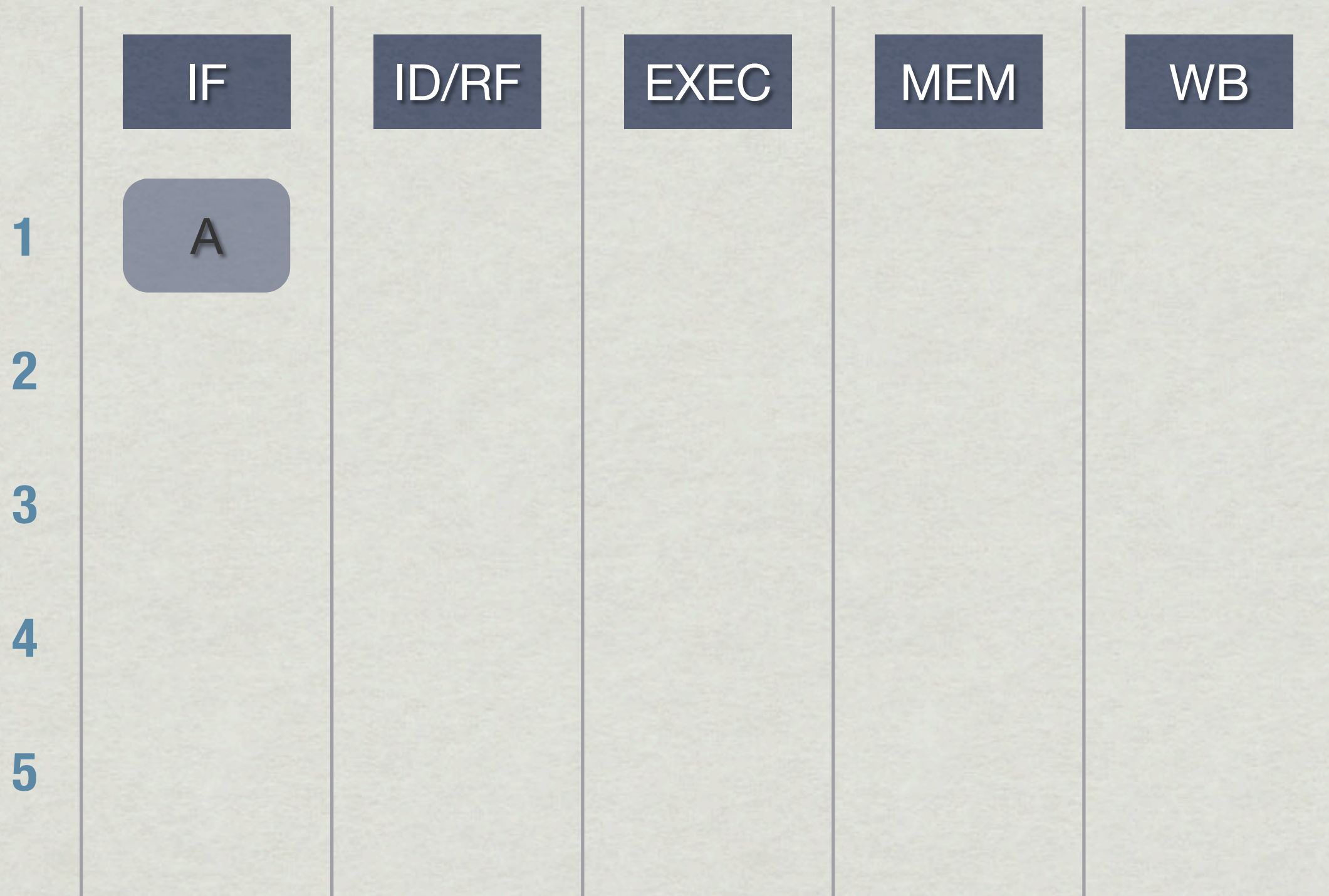
Mispredicted Branch



Mispredicted Branch



Mispredicted Branch



Mispredicted Branch



Mispredicted Branch



Mispredicted Branch



Mispredicted Branch



Mispredicted Branch



Mispredicted Branch

- ✱ How can we take advantage of this knowledge?
 1. **How good** is your CPU's **branch prediction HW**?
 2. If **not good** (e.g. PS3 PPU!), avoid branches in high-performance code:
 - ✱ Calculate both results and use **fsel**
 - ✱ Split branchy loops into separate cases
 - ✱ Select simpler, less branchy algorithms (e.g. insertion sort over quicksort) where applicable

Tools

In-Engine Debugging and Profiling Tools

- * Crucial to build **in-engine tools** to aid in development
- * In-game **development menus** and **shortcut keys**
- * **Debug drawing** facilities
- * In-engine **profiling** tools
- * Useful to **all disciplines**
 - * Programmers, designers, artists, sound team, ...

In-Game Menus

0.000000

Debug Draw Options

Insert Debug Strings into Push Buffer
Enable Wireframe
Enable Wireframe See-Through
Enable Point Mode

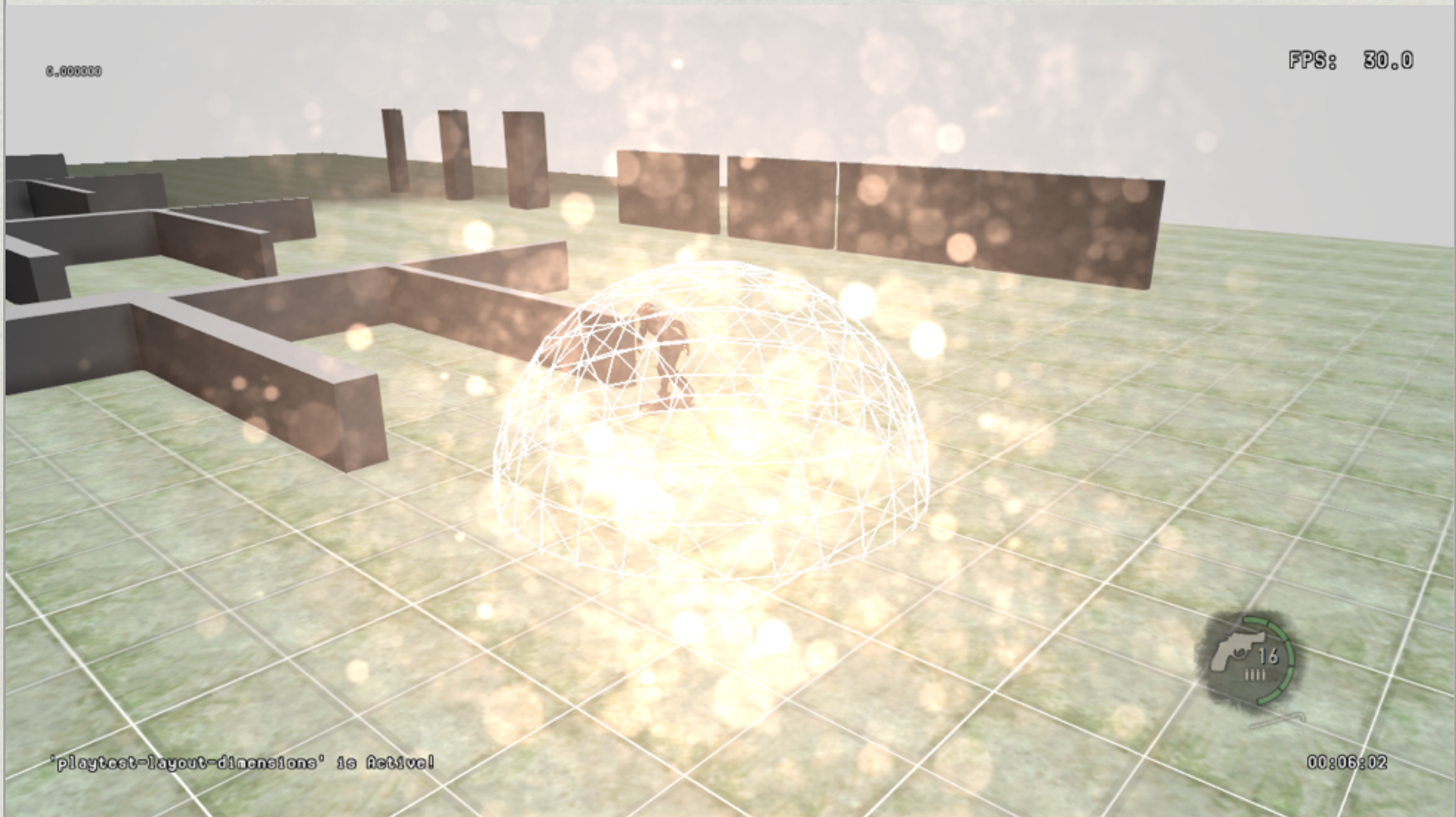
Disable Surface Reflections
Disable Sunlight ShadowBuffer Draw
Disable Local Light ShadowBuffer Draw
Disable Snow/Rain Buffer Draw
Disable Shader Lod Buckets
> Disable Background Meshes
Disable Foreground Meshes
Disable Alpha blend Meshes
Disable Shrub Meshes
Disable Foreground Reflections
Disable Subsurface pass

Draw Shadow Occluders
Draw Reflection Geometry
Draw Local Shadow Occluders
Draw Rain/Snow Occluders
Draw Velocity Mblur Geometry
Draw Flashlight Bouncers

Show Instance Bounding Sphere
Show Fg Instance Bounding Box
Show Sunlight Intensity
Show Pixel Shader Allocations
Show instance texture debug info

Global Near Dist 0.24
Reflection Far Distance 30.00

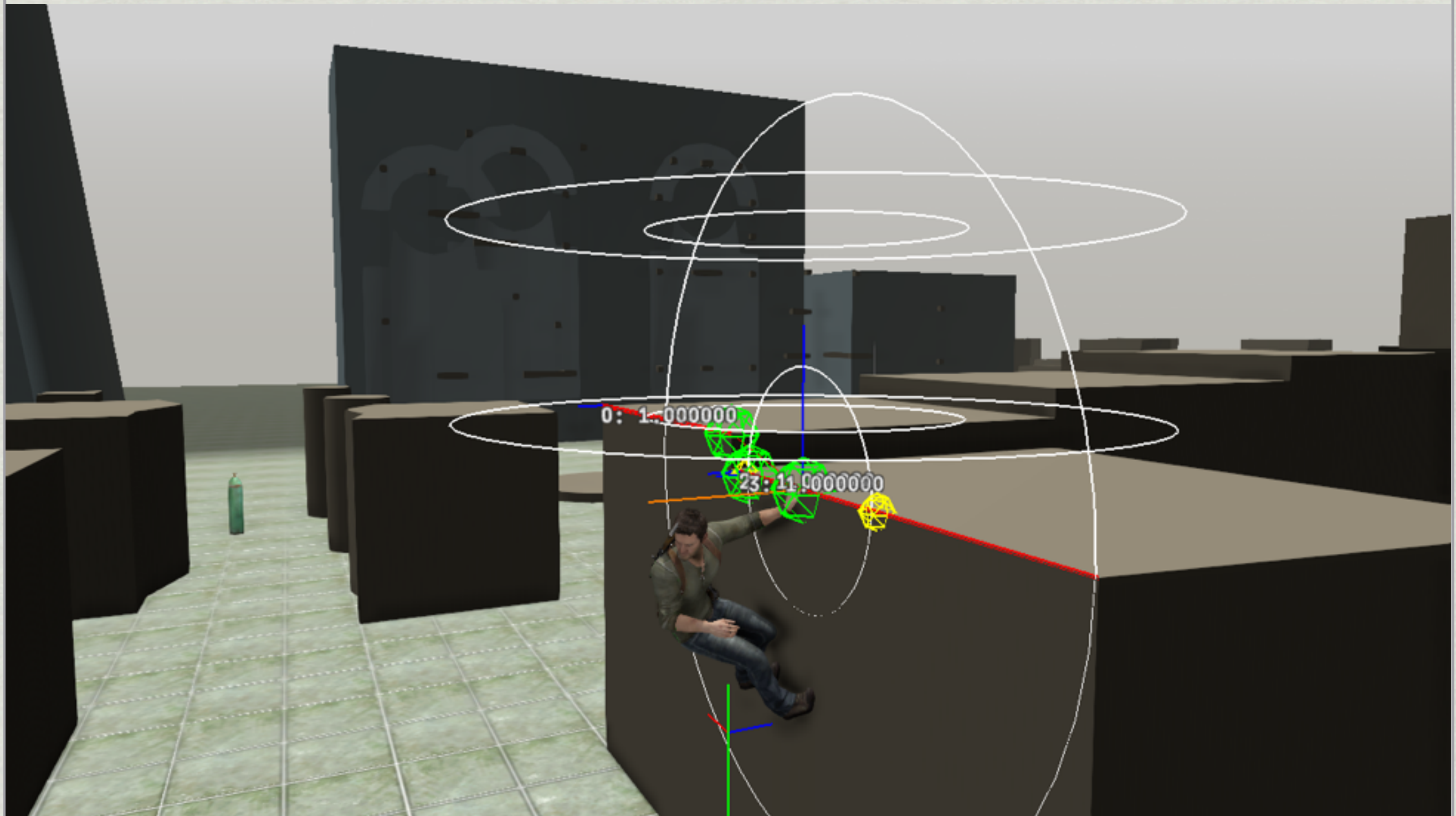
Debug Drawing



Debug Drawing



Debug Drawing



Profiling Tools



The screenshot shows a game environment with a dam and a character. Overlaid on the left side is a memory profiling menu. The menu is divided into three sections: Retail Memory, Debug Memory, and Stack Memory. Each section lists various memory components and their usage. The background shows a character standing on a bridge or walkway, looking at a dam with water flowing over it. The sky is overcast, and there are mountains in the distance.

FPS: 30.0

-[Retail Memory]-----

OS Memory	:	43.00	/	43.00	MB
Code And Static Data	:	23.13	/	23.13	MB
PRXs	:	2.19	/	2.38	MB
Debug PRXs	:	0.00	/	3.00	MB
Global Memory	:	0.01	/	66.00	MB
IO Memory	:	119.91	/	120.00	MB
Threads Contexts	:	1.07	/	1.50	MB (Max 1.11 MB)
Available Sys Memory	:			26.49	MB (Min 26.46 MB)
Total:		256.00 MB			

-[Debug Memory]-----

Extra OS Memory	:	62.00	/	62.00	MB
Code/Data Overflow Mem	:	6.12	/	10.00	MB
IO Debug Memory	:	86.20	/	92.00	MB
Physics Overflow Mem	:	0.00	/	7.88	MB
Debug Memory	:	61.13	/	69.00	MB
Total:		233.00 MB			

-[Stack Memory]-----

Max Stack Usage	:	117.47	/	256.00	KB
-----------------	---	--------	---	--------	----

'tommys-dan-tom-path-sluiice-gate-start' is Active!

00:02:15

Profiling Tools



Profiling Tools



Profiling Tools



Profiling Tools



Profiling Tools



Profiling Tools



Data-Driven Design

Data-Driven Design

- * Put the **power to create** into the hands of the **content creators!**
- * Reduce dependencies on **programming team**
- * At Naughty Dog, we do this via:
 - * **Data-driven** systems with easy-to-edit data files
 - * Runtime **scripting language** for use by designers and artists

Data-Driven Design

- ✱ Both data scripts and runtime scripts written in **Scheme** (a **Lisp** variant)
- ✱ Rich Lisp history at Naughty Dog!
- ✱ Scheme offers powerful language customization tools via **hygienic macros**
- ✱ Allows you to easily **customize the language** to suit your needs

Data-Driven Design

- ✱ Example of simple data definition script:

```
(define-physics-sound
  :models      ('tin-cans 'rusty-cans)
  :joint       'root
  :light-hit   'sfx-cans-hit-light
  :hard-hit    'sfx-cans-hard-hit
  :roll        'sfx-cans-roll
  :slide       'sfx-cans-slide
  :light-force 2.0
  :hard-force  10.0
)
```


Data-Driven Design

- * Users can edit data-definition scripts in any text editor
- * Re-build the data and **hot-swap** into the running game via a command-line Scheme interpreter

```
> (mr "physics-sounds.dc")
```



Runtime Scripts

- ✱ Example of runtime script:

```
(define-state-script 'ss-kick-gate
  (state ('idle)
    (on (start)
      (animate 'self 'gate-idle)
    )
    (on (event 'kick)
      (animate 'player 'player-gate-kick)
      (wait-animate 'self 'gate-open)
      (send-event 'opened 'self)
      (go 'open)
    )
  )
```


Runtime Scripts

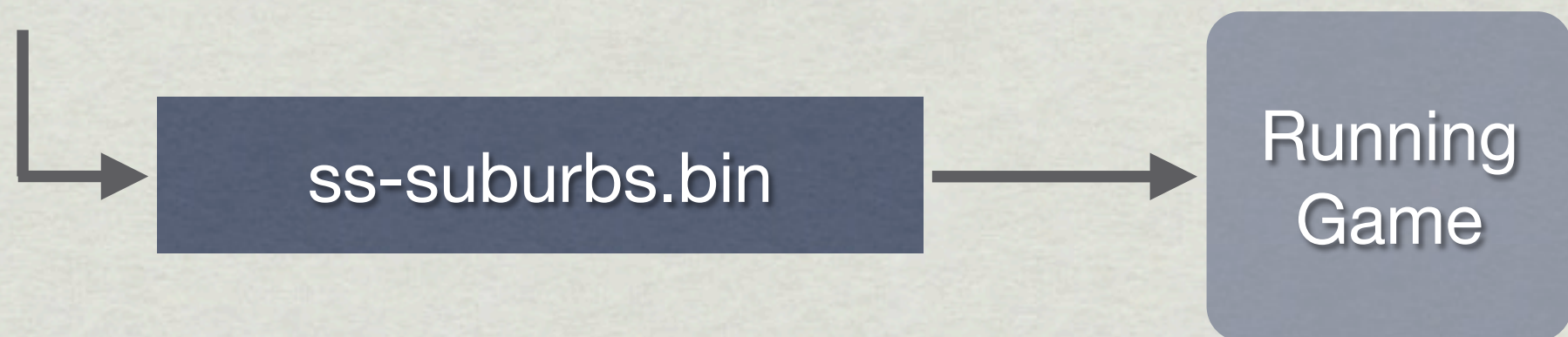
```
;; ...
```

```
(state ('open)
  (on (start)
    (animate 'self 'gate-idle-open)
  )
  (on (event 'close)
    (wait-animate 'self 'gate-close)
    (go 'idle)
  )
)
)
```


Runtime Scripts

- * Users can edit runtimes scripts in any text editor
- * Re-build the script and **hot-swap** into the running game via a command-line Scheme interpreter

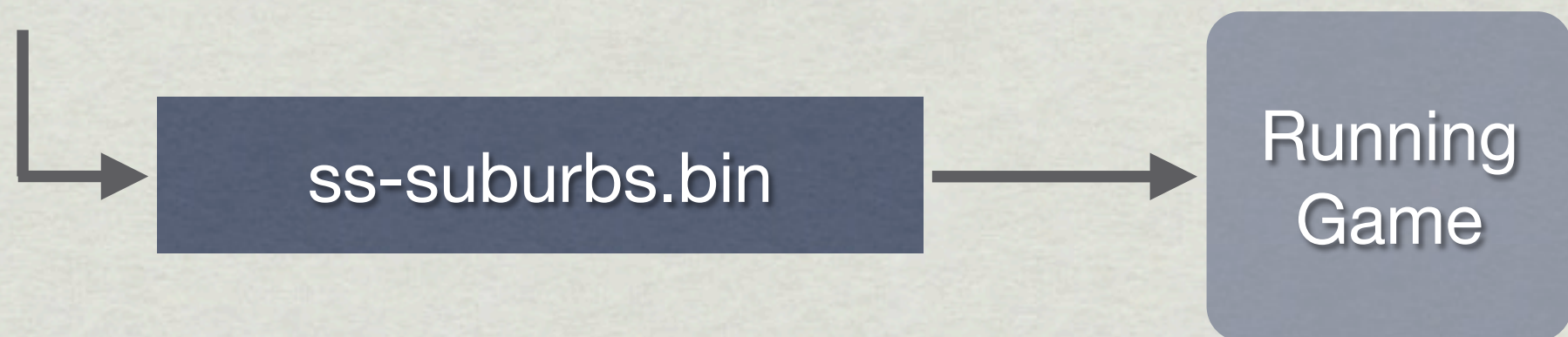
```
> (mr "ss-suburbs.dc")
```



Runtime Scripts

- * Users can edit runtimes scripts in any text editor
- * Re-build the script and **hot-swap** into the running game via a command-line Scheme interpreter

```
> (mr "ss-suburbs.dc")
```

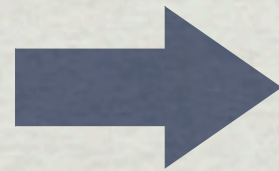


SCRIPT CODE IS DATA!

Conclusion

Conclusion

- * People
- * Culture
- * Process
- * Technology



CONTENT!

Lighting and Shading



Visual Effects



Audio



Animation



Thanks for Listening!

✱ Questions?

jason_gregory@naughtydog.com

www.gameenginebook.com