

# Control Groups (cgroups)

Tyler Caraza-Harter

# Outline

Performance Isolation

**Mechanism** Examples

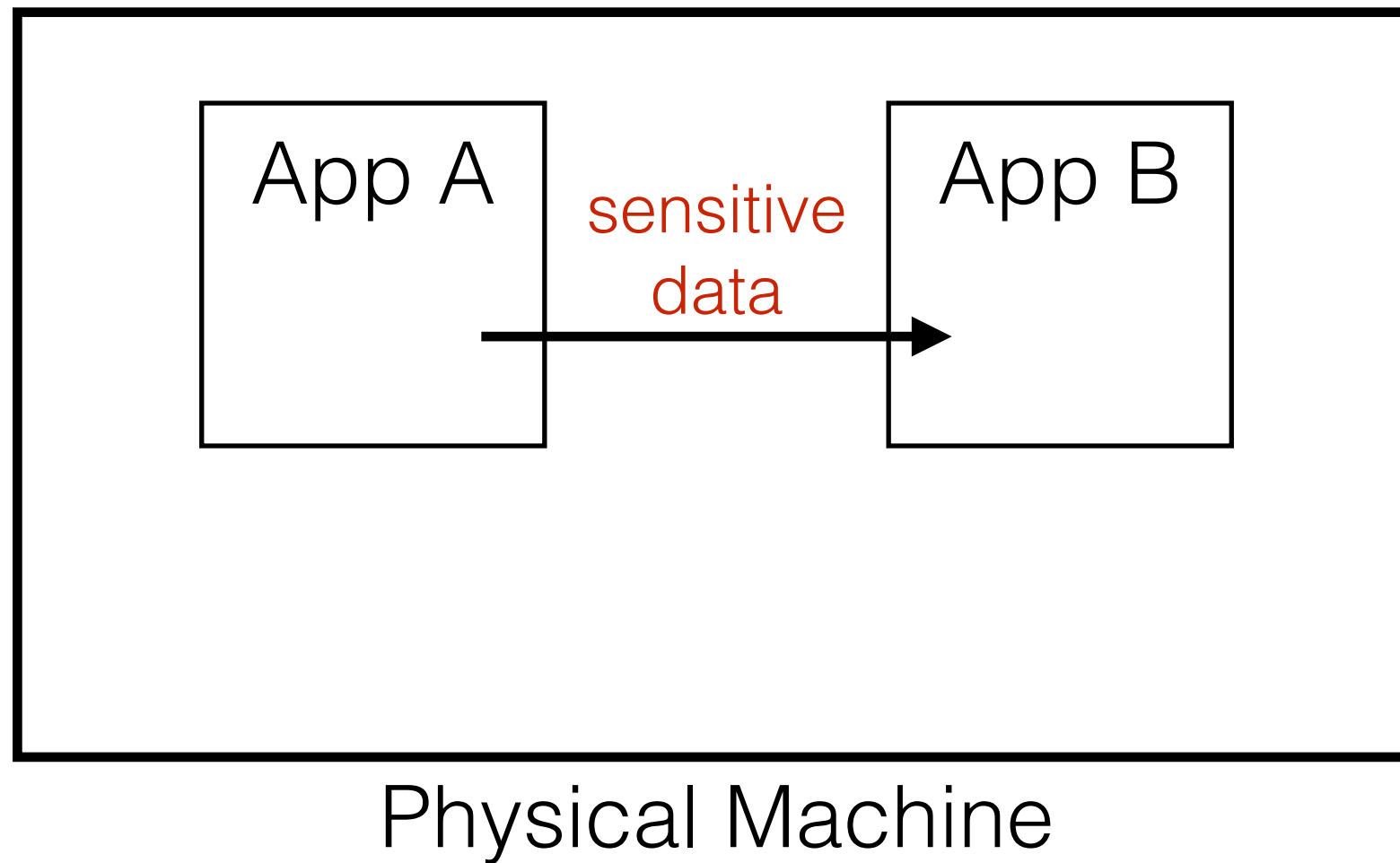
**Interface:** cgroup overview

Controllers

- freezer/cpu/cpuset
- memory
- io (disk)
- pids
- what about network?

Usage in OpenLambda

# Isolating Data

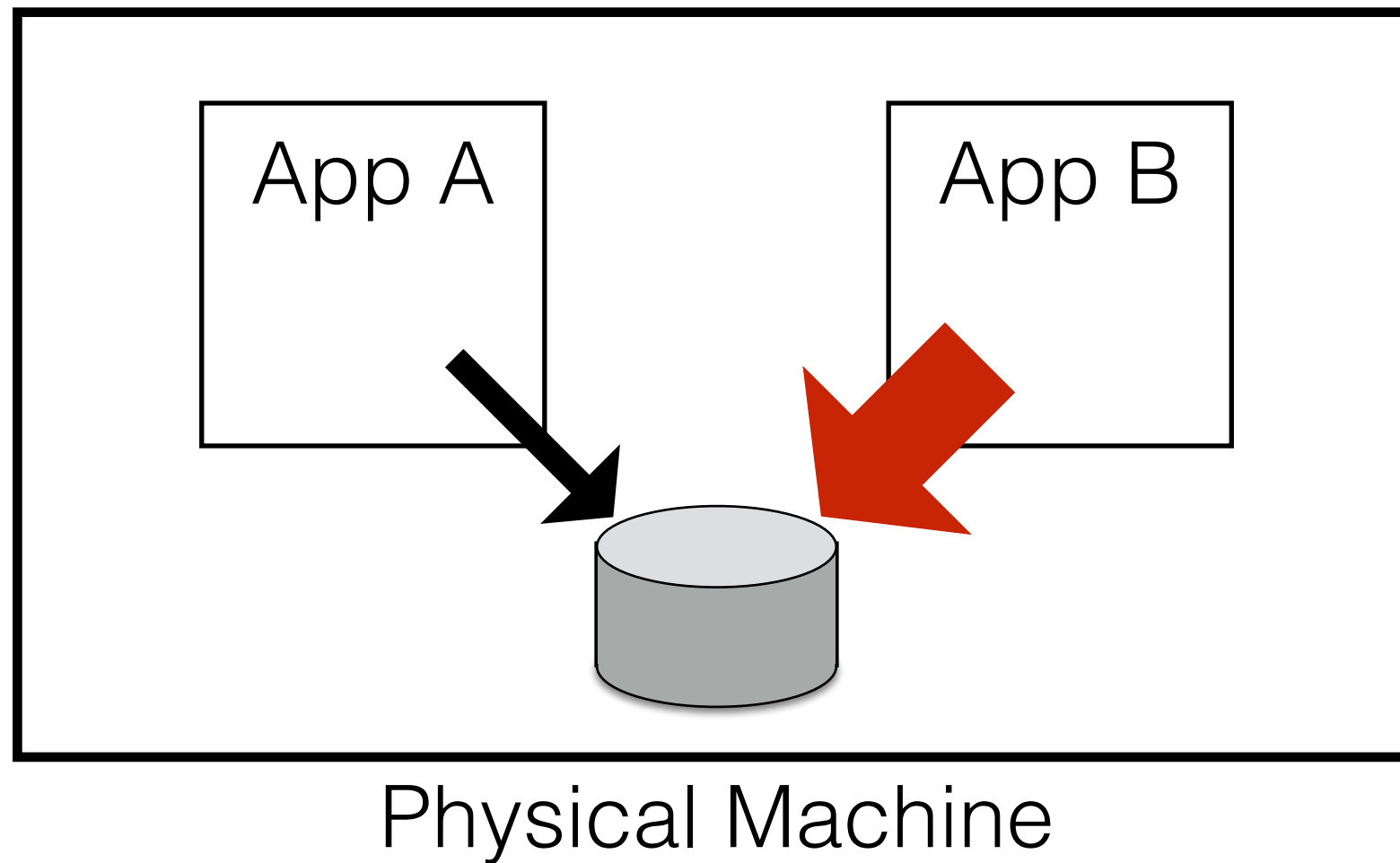


don't want: **leaks**

Linux

- older features: virtual memory, ACLs, etc
- newer features: namespaces (separate session)

# Isolating Performance



don't want: **unfairness**

This session focuses on **performance** isolation

# Perf Isolation: Interface vs. Mechanism

Linux has many **mechanisms** for isolating performance. Examples:

- **OOM (out of memory) killer** -- randomly kill a process when low on memory (relative to total RAM, or a set limit)
- **CPU scheduler**, such as CFS (completely fair scheduler)
- **Block I/O scheduler**, such as BFQ (budget fair queueing)

There are multiple **interfaces** for interacting with these mechanisms:

- nice, ionice
- cgroups v1 (used by OpenLambda when SOCK was published)
- **cgroups v2** (used by OpenLambda now)

**Observation:** cgroups can only be as good as the underlying schedulers/mechanisms.

# Outline

Performance Isolation

**Mechanism** Examples

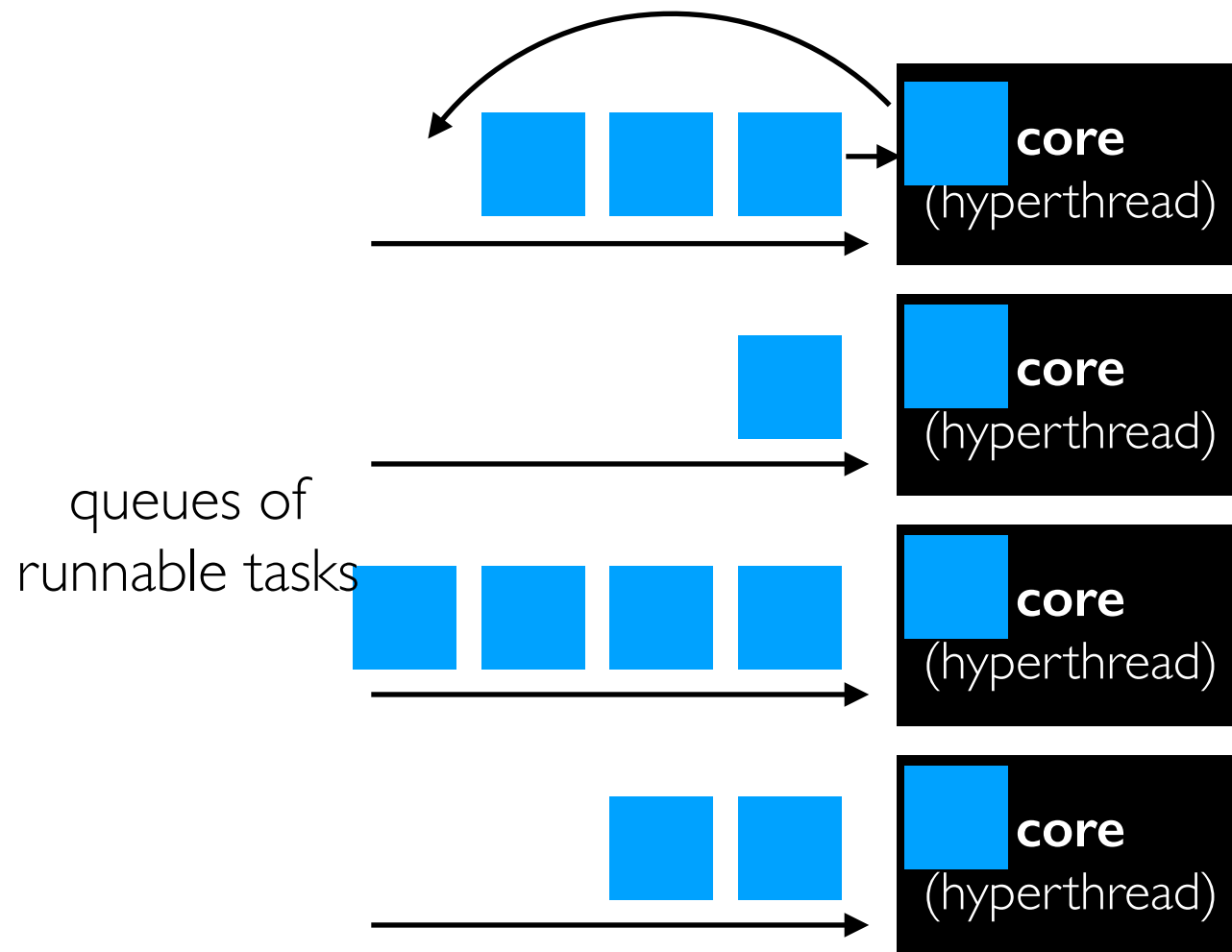
**Interface:** cgroup overview

Controllers

- freezer/cpu/cpuset
- memory
- io (disk)
- pids
- what about network?

Usage in OpenLambda

# CFS (Completely Fair Scheduler) - Default

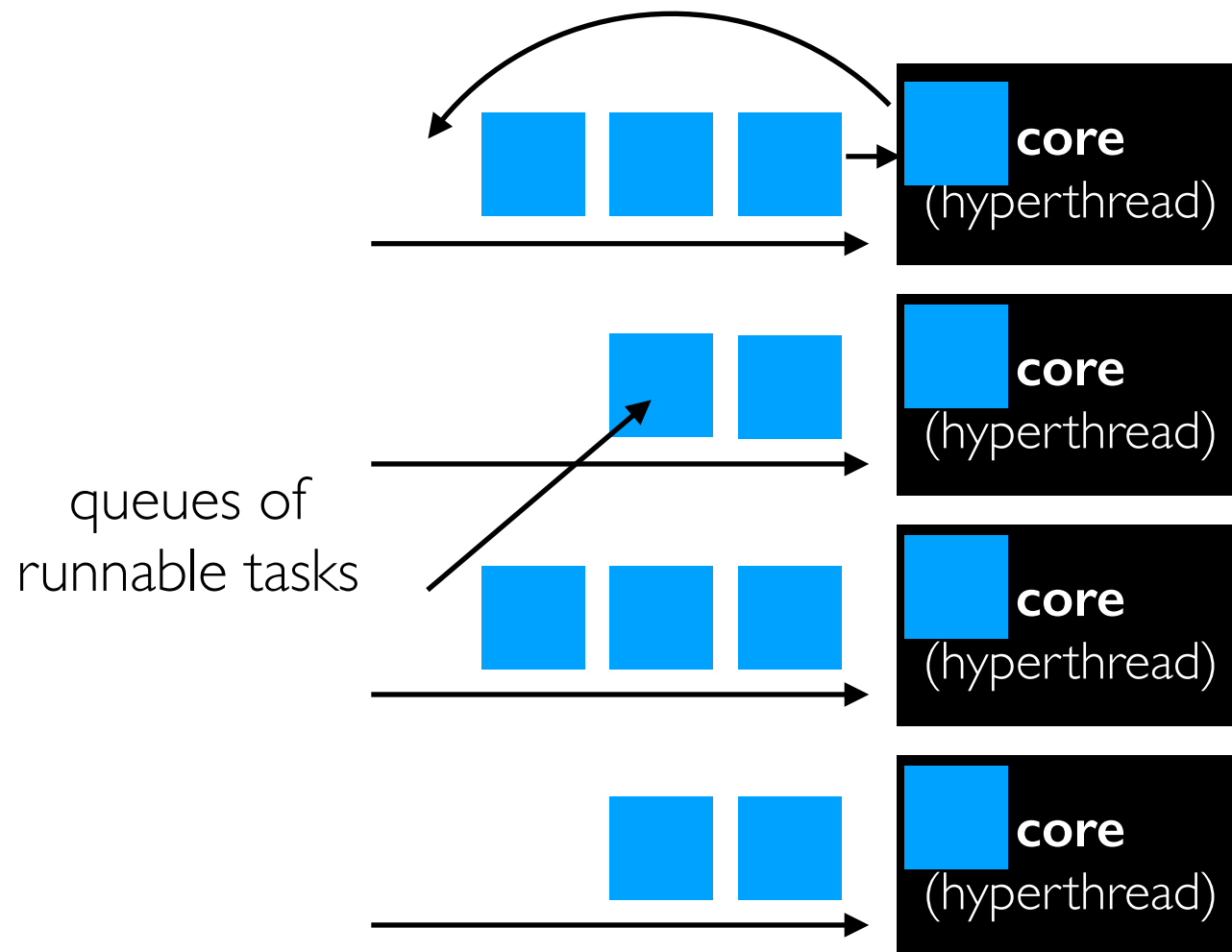


## Operation

- each core/hyperthread has its own **queue** (want to spend more time where CPU cache is warm)
- cycle through processes: higher **priority** = more time running

```
time nice -n 10 python3 -c 'sum(range(int(1e9)))'
```

# CFS (Completely Fair Scheduler) - Default

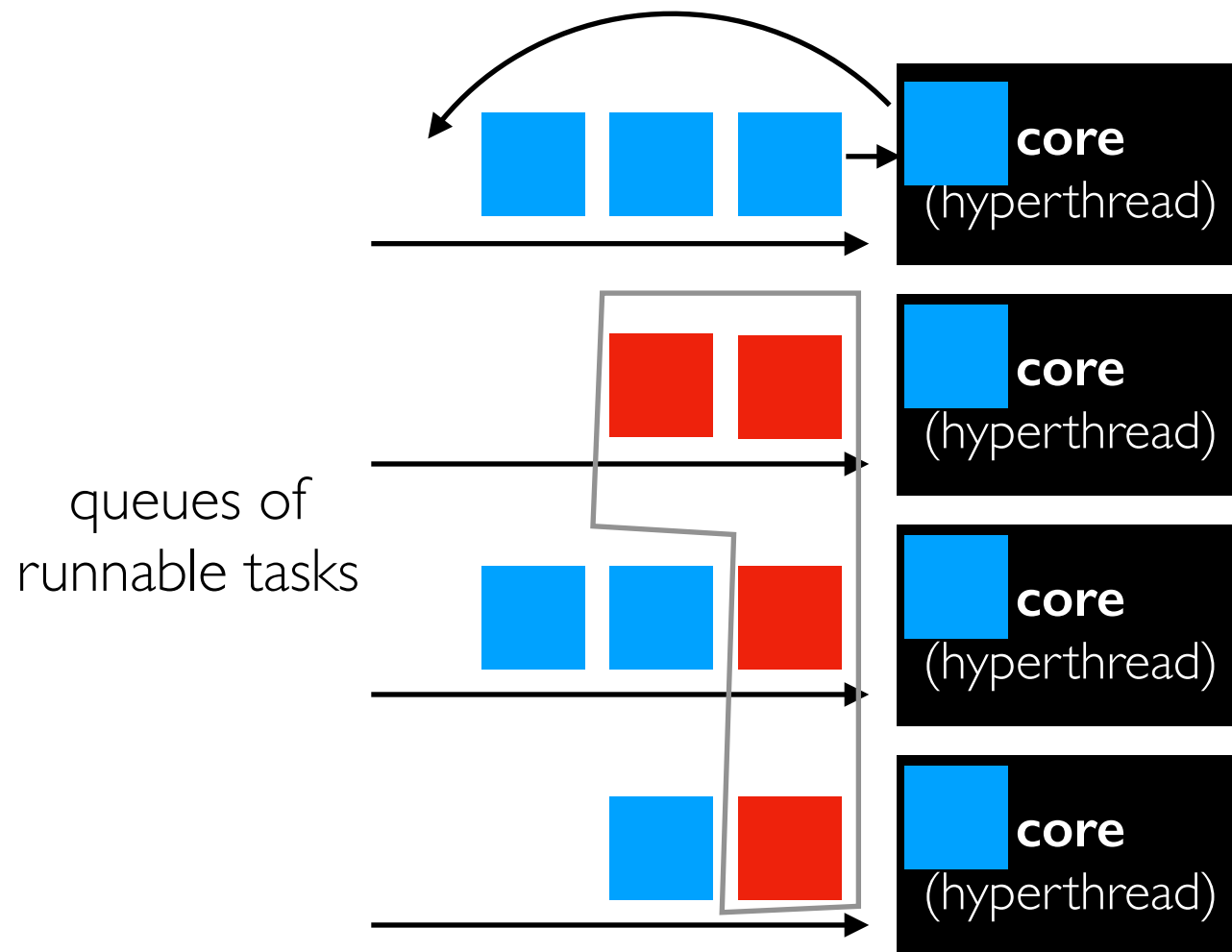


## Operation

- each core/hyperthread has its own queue  
(want to spend more time where CPU cache is warm)
- cycle through processes: higher priority = more time running
- occasionally **rebalance**



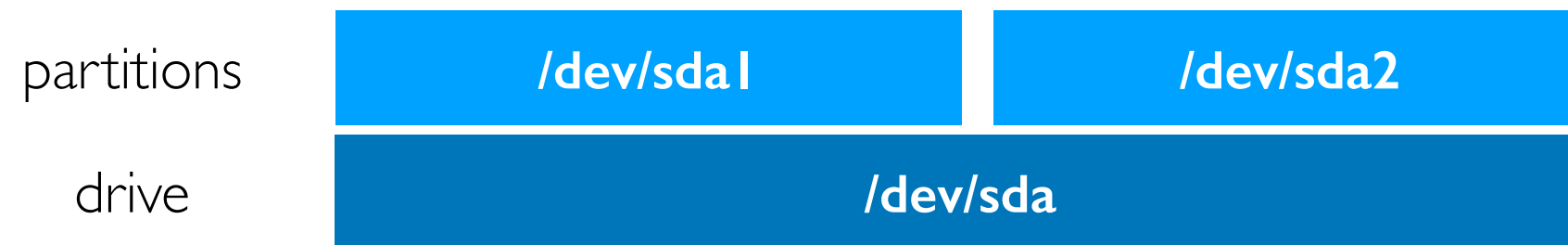
# CFS (Completely Fair Scheduler) - Default



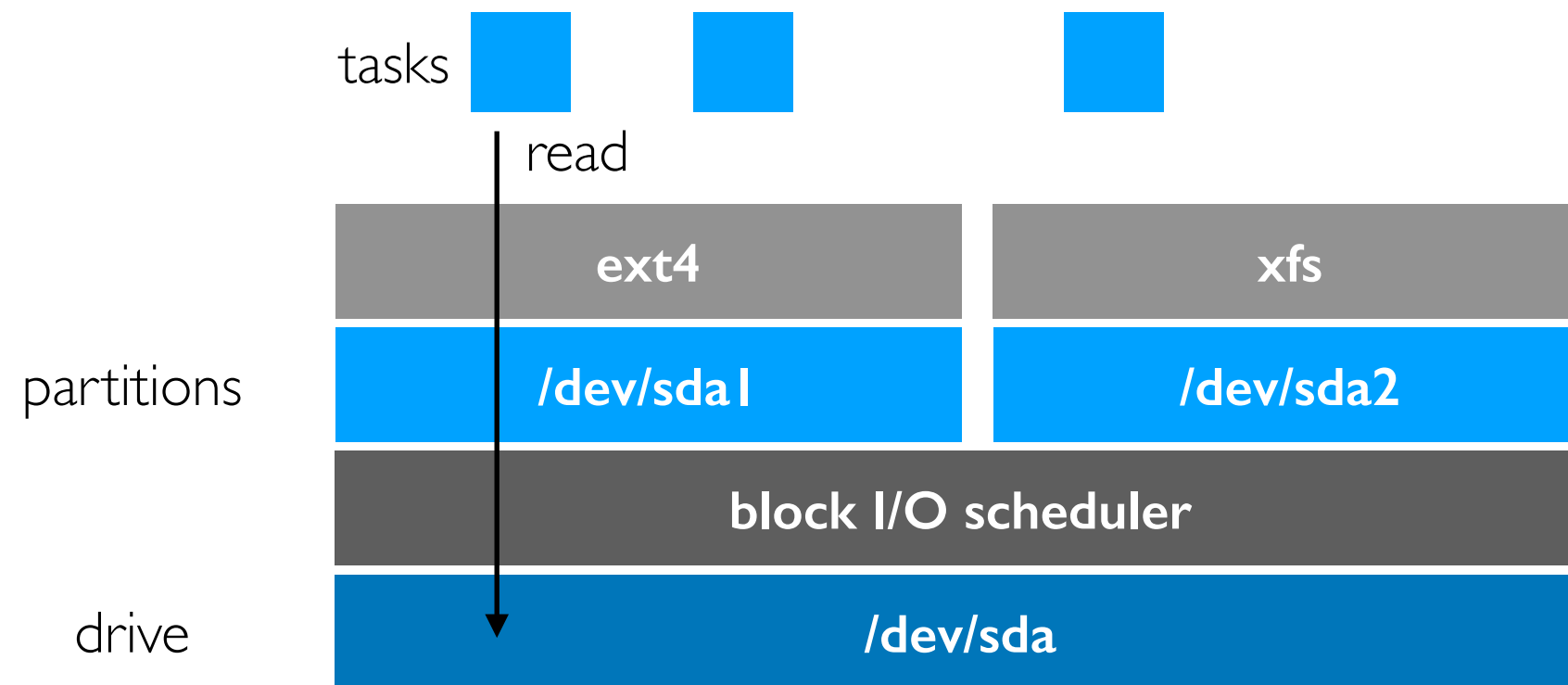
## Operation

- each core/hyperthread has its own queue  
(want to spend more time where CPU cache is warm)
- cycle through processes: higher priority = more time running
- occasionally rebalance
- **task groups**'s let multiple threads in a process (or in a multi-process app) share accounting

# Disk I/O: Block Scheduler



# Disk I/O: Block Scheduler



```
$ cat /sys/block/sda/queue/scheduler  
[mq-deadline] none  
$ echo none > /sys/block/sda/queue/scheduler
```

`ionice` works like `nice`, but for block I/O schedulers that support priorities

# Block I/O Scheduling Limitations

## Split-Level I/O Scheduling

Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik\*, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

*University of Wisconsin-Madison*

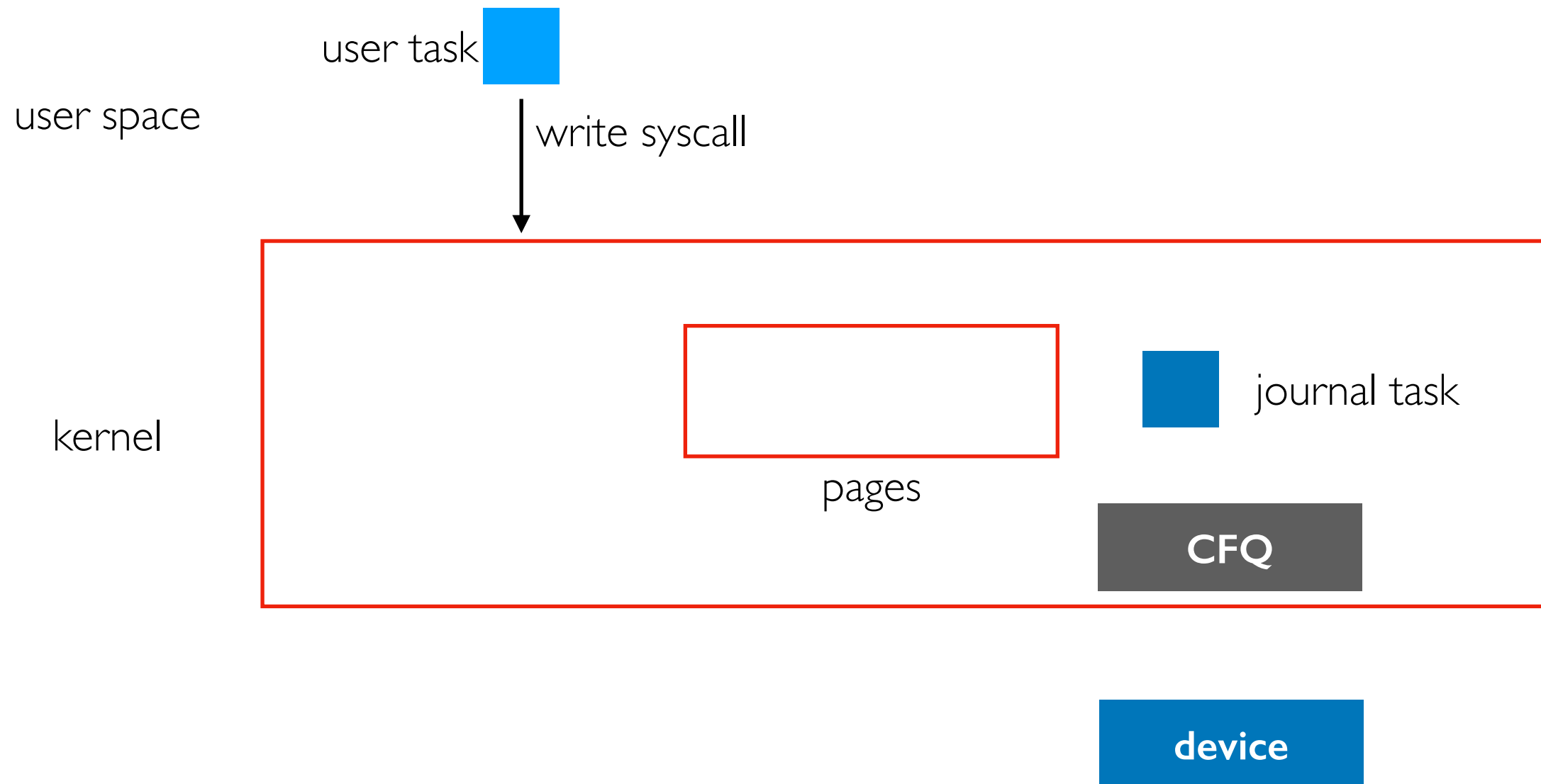
*IBM Research-Almaden\**

<https://research.cs.wisc.edu/wind/Publications/split-sosp15.pdf>

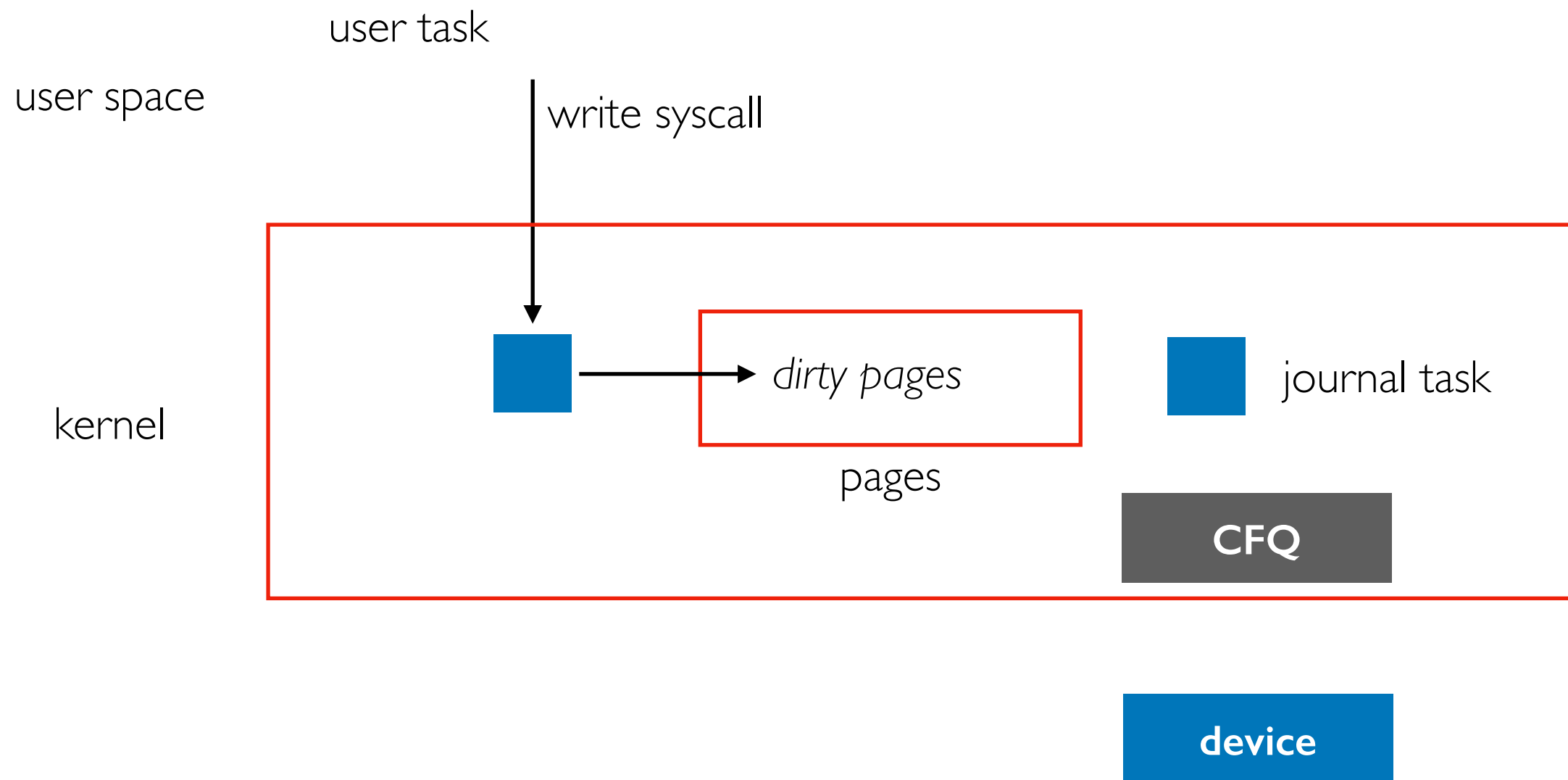
**Issue:** block schedulers associate I/O with tasks, but due to the page cache and journaling, blame is often misdirected.

**Remember:** cgroups can only be as good as the underlying schedulers/mechanisms.

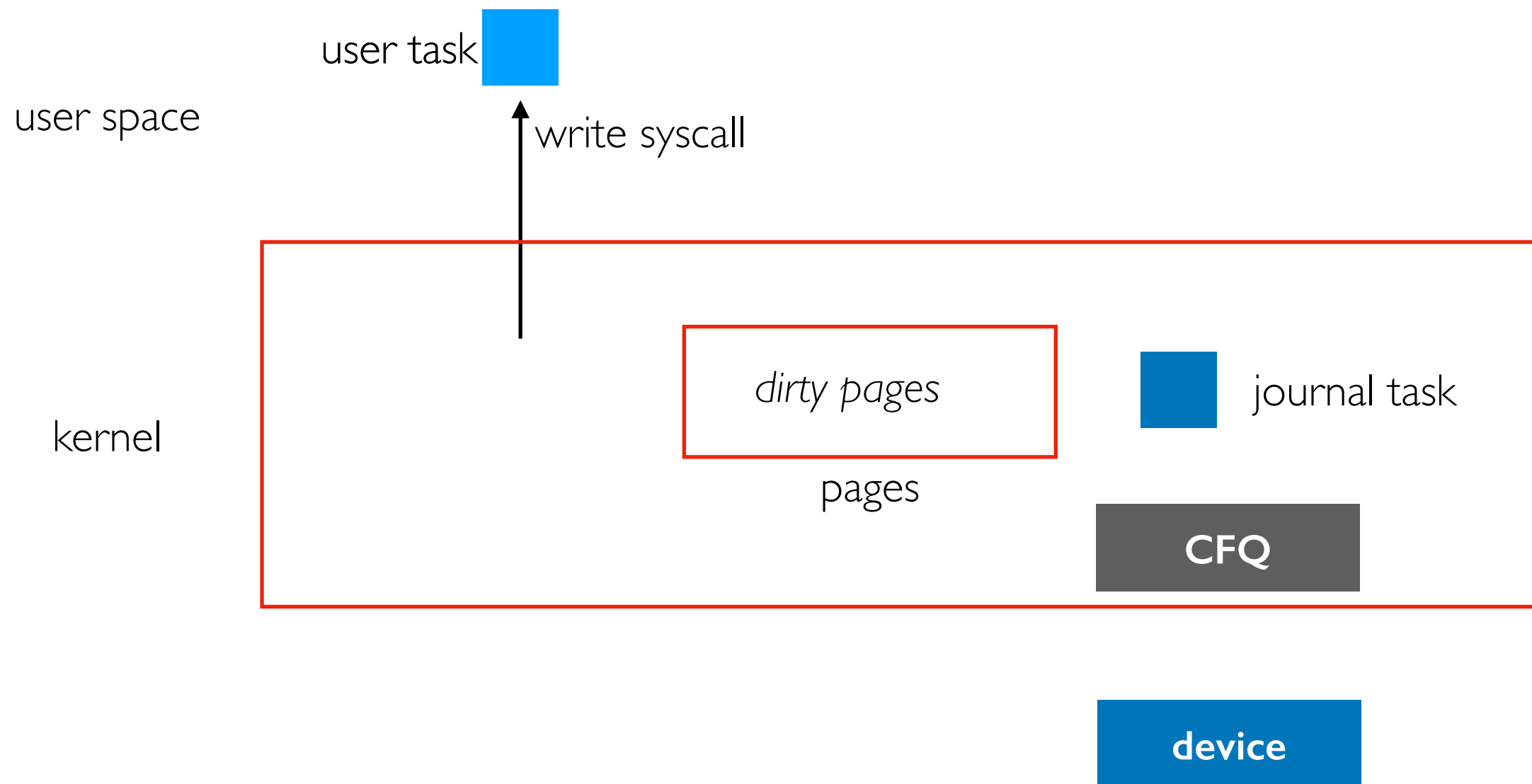
# Block I/O Scheduling Limitations



# Block I/O Scheduling Limitations

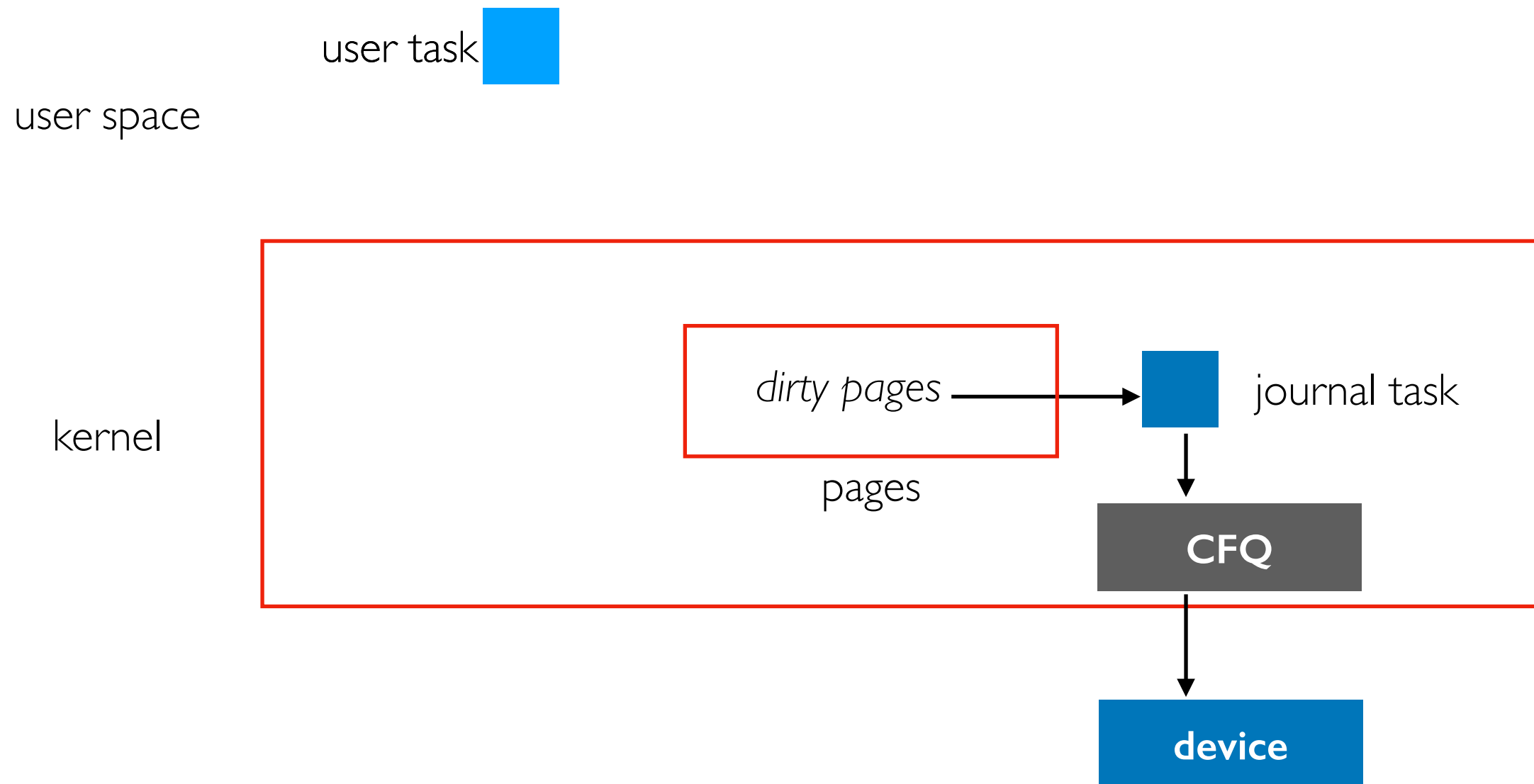


# Block I/O Scheduling Limitations



problem 1: user task has already created work that must be done but hasn't been accounted for

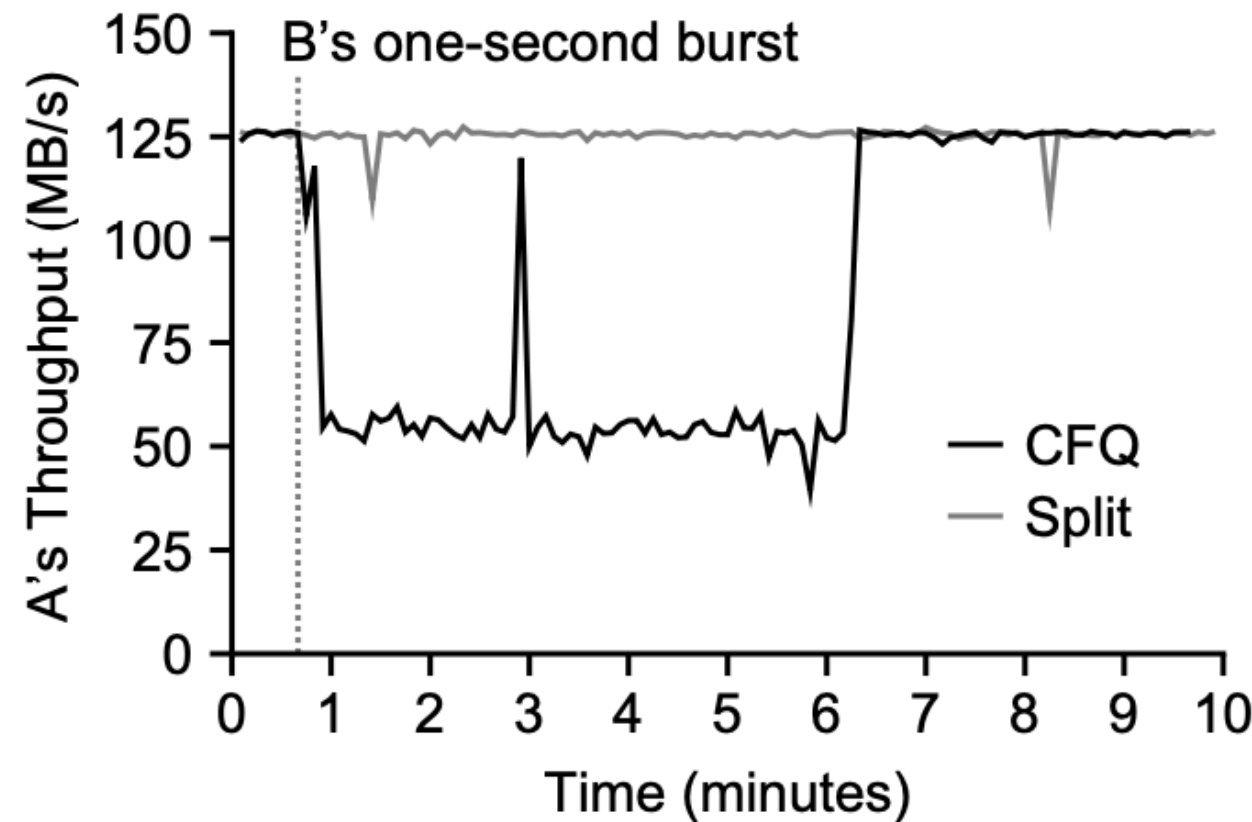
# Block I/O Scheduling Limitations



problem 2: from scheduler's perspective, all writes can be blamed on one task (journal task)



# Problem 1: Too Much Work Created before Accounting



**Figure 1: Write Burst.** *B's one-second random-write burst severely degrades A's performance for over five minutes. Putting B in CFQ's idle class provides no help.*

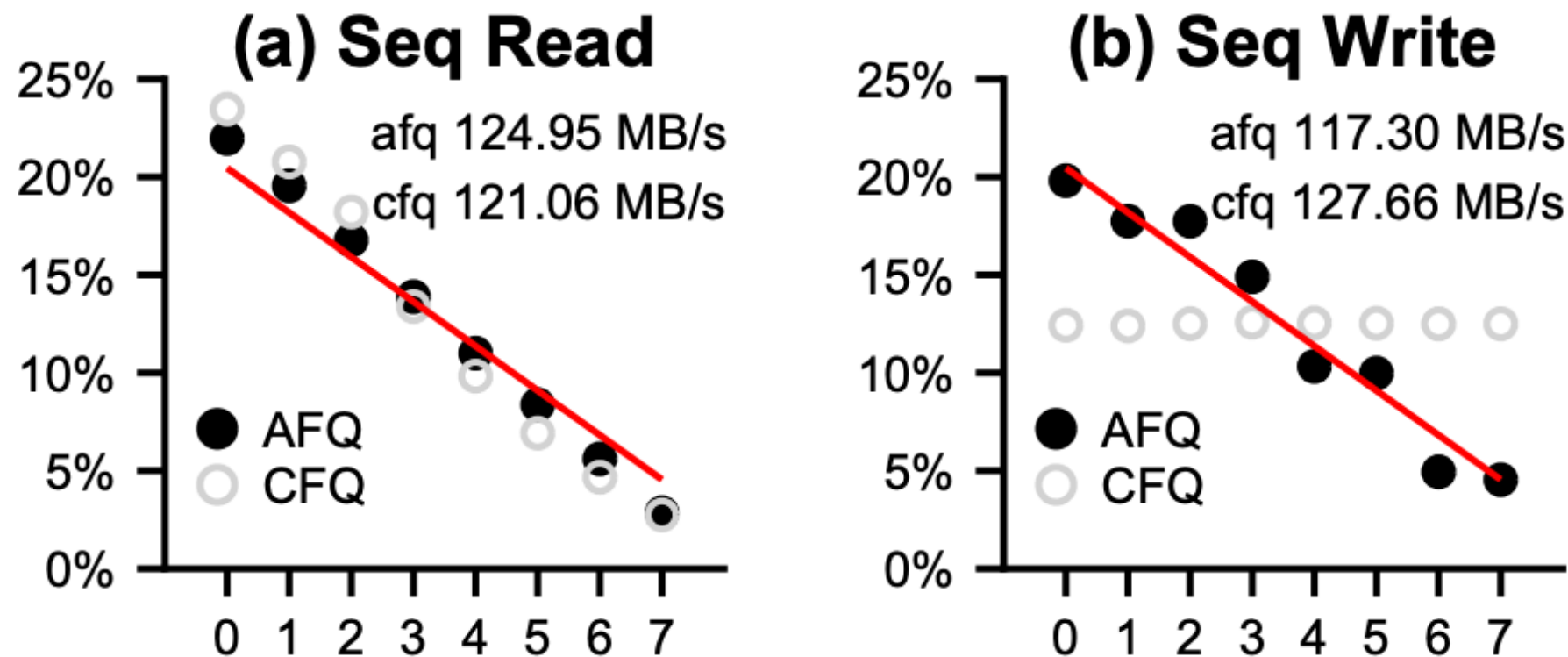
## Split-Level I/O Scheduling

Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik\*, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

University of Wisconsin-Madison

IBM Research-Almaden\*

# Problem 2: Write Priority is Meaningless



## Split-Level I/O Scheduling

Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T. Kaushik\*, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau

*University of Wisconsin-Madison*

*IBM Research-Almaden\**

# Outline

Performance Isolation

**Mechanism** Examples

**Interface:** cgroup overview

Controllers

- freezer/cpu/cpuset
- memory
- io (disk)
- pids
- what about network?

Usage in OpenLambda

# cgroups

## Control Groups



```
graph TD; CG[Control Groups] --> C[Controllers]; CG --> G[Groups of processes];
```

**Controllers** (per resource type) enable limiting/sharing consumption

- **compute:** `cpu`, `cpuset`, `freezer`
- **memory:** `memory`, `hugetlb`, `rdma`
- `io` (disk only)
- `pids`
- `perf_event`

**Groups** of processes (sometimes threads)

- groups are hierarchical
- different combinations of controllers/settings can be applied to groups
- implemented as a pseudo **file system** (for example, you mount the `cgroup2` file system, and create cgroups with `mkdir`)

## cgroups v1 vs. v2

- v1 is more flexible (probably too flexible)
- v2 just makes more "sense"
- both can be used simultaneously (but you probably shouldn't)

# cgroups

## Control Groups



```
graph TD; CG[Control Groups] --> C[Controllers]; CG --> G[Groups of processes];
```

**Controllers** (per resource type) enable limiting/sharing consumption

- **compute:** `cpu`, `cpuset`, `freezer`
- **memory:** `memory`, `hugetlb`, `rdma`
- `io` (disk only)
- `pids`
- `perf_event`

**Groups** of processes (sometimes threads)

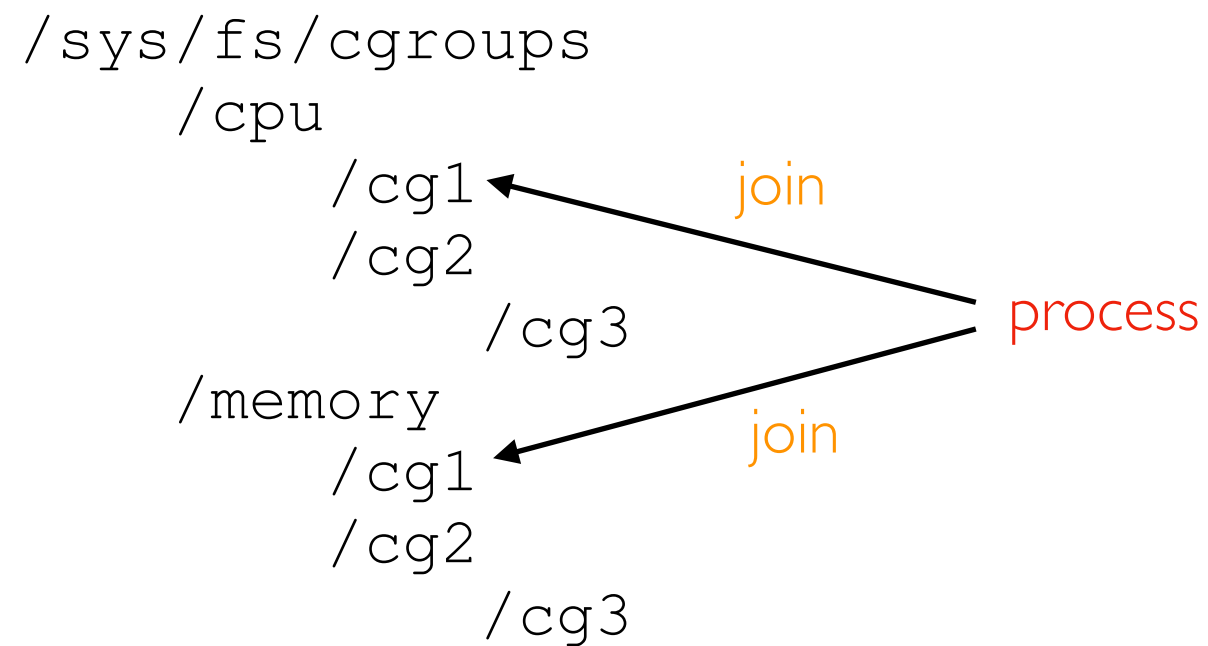
- groups are hierarchical
- different combinations of controllers/settings can be applied to groups
- implemented as a pseudo **file system** (for example, you mount the `cgroup2` file system, and create cgroups with `mkdir`)

## cgroups v1 vs. v2

- v1 is more flexible (probably too flexible)
- v2 just makes more "sense"
- both can be used simultaneously (but you probably shouldn't)

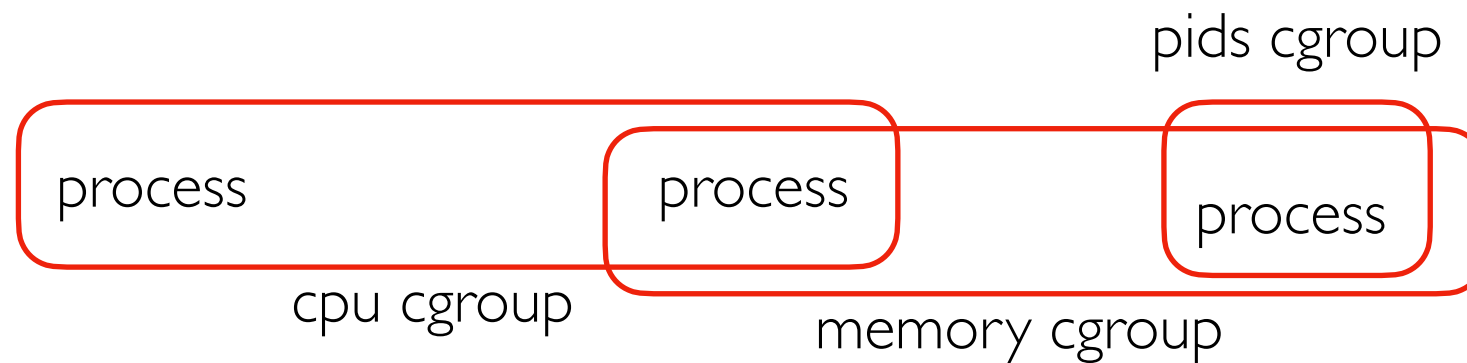
# cgroups **VERSION I** hierarchy

example hierarchy with two controllers (cpu and memory)  
and three groups (cg1, cg2, and cg3)



With v1, you had to join many different cgroups, for each type.

# cgroups **VERSION I** hierarchy



cgroups v1 allowed a lot of flexibility  
with almost no imaginable use case

# cgroups **VERSION 2** hierarchy

example hierarchy with two controllers (cpu and memory)  
and two groups (cg1 and cg2)

```
/sys/fs/cgroups
  /cg1 ← join process
    /cpu files...
    /memory files...
  /cg2
    /cpu files...
    /memory files...
  /cg3
    /cpu files...
    /memory files...
```



# Other Simplifications

## Mount Points

- **v1:** instances of the cgroup FS could be mounted in different places, with different controllers applied
- **v2:** all mount points are just identical views into the same hierarchy

## Nodes having processes

- **v1:** any node in the tree can have processes
- **v2:** only the root and leaves can have processes (some exceptions)

## Threads

- **v1:** threads OR processes can be in a group
- **v2:** only processes can be in a group (this evolved some...)

# Threads

Memory cgroups don't make sense for threads: heap space is not assigned to specific threads, and the kernel isn't involved on every malloc, so it couldn't apply different memory limits to different threads anyway.

crgroups V2 dropped support for threaded cgroups.

BUT! CPU cgroups are useful for threads in some case (threads doing background work should perhaps get less share).

Conclusion: V2 added back limited support by implementing different kinds of controller.

- **domain:** for processes (the default)
- **threaded:** for resources like CPU

# Controller Types: Domain vs. Threaded

example hierarchy with two controllers (cpu and memory)  
and two groups (cg1 and cg2)

```
/sys/fs/cgroups
  /myapp [domain]
    /cg1 [domain threaded]
      /memory files...
      /foreground [threaded]
        /cpu files...
      /background [threaded]
        /cpu files...
```

process is at this level,  
with a memory limit



# Controller Types: Domain vs. Threaded

example hierarchy with two controllers (cpu and memory)  
and two groups (cg1 and cg2)

```
/sys/fs/cgroups
  /myapp [domain]
    /cg1 [domain threaded]
      /memory files...
      /foreground [threaded] ← threads of the cg1 will
        /cpu files...          be in one of these
      /background [threaded] ←
        /cpu files...
```

# Joining a cgroup

## approach 1: after creation

each cgroup has a `cgroup.procs` and `cgroup.threads` entry. Write a PID or thread ID to one of these to join it.

## approach 2: during creation

these days `fork()` is just a wrapper around a clone call (`clone`, `clone2`, or `clone3`).

Recently (Linux 5.7, released 2020), `clone3` added `CLONE_INTO_CGROUP` flag.

# Outline

Performance Isolation

**Mechanism** Examples

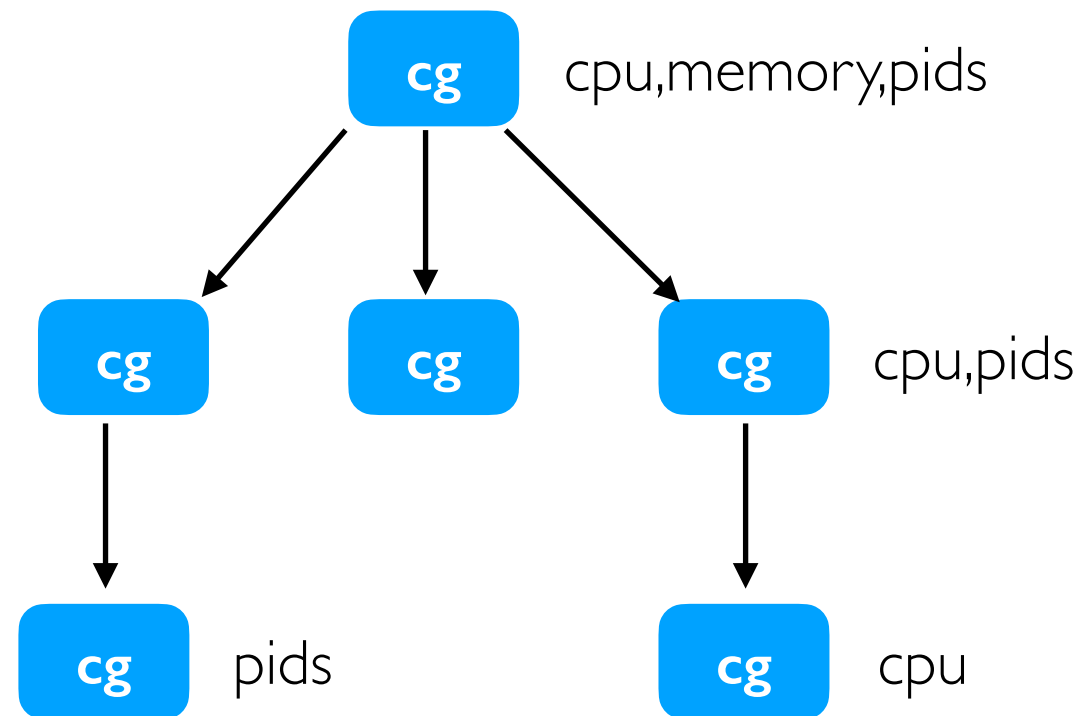
**Interface:** cgroup overview

## Controllers

- freezer/cpu/cpuset
- memory
- io (disk)
- pids
- what about network?

Usage in OpenLambda

# Controllers



- a cgroup will have a subset of controllers enabled
- control of some of these can be passed to children
- implication: as you go deeper, the set of controllers can only get smaller

```
/sys/fs/cgroups
```

```
  /cg1
```

```
    /cgroup.controllers
```

```
    /cgroup.subtree_control
```

```
    /cg2
```

```
      /cgroup.controllers
```

```
      /cgroup.subtree_control
```

# Outline

Performance Isolation

**Mechanism** Examples

**Interface:** cgroup overview

Controllers

- freezer/cpu/cpuset
- memory
- io (disk)
- pids
- what about network?

Usage in OpenLambda



# Controlling Compute Resources

Should a cgroup run? Where should it run? How much time should it run?

These are controlled by `freezer`, `cpuset`, `cpu` and respectively.

# Freeze

```
/sys/fs/cgroups
```

```
  /cg1
```

```
    /cgroup.freeze
```

```
    /cgroup.events
```

```
  /cg2
```

```
    /cgroup.freeze
```

```
    /cgroup.events
```

write 1 to freeze (make non-schedulable), 0 to unfreeze

note: this is not a regular controller, so you don't need to enable it.

Freezing/unfreezing is asynchronous. Use `poll()` or `inotify` on `cgroup.events` to observe when the action is complete. Note: OpenLambda should do this, but it doesn't yet.

Typical use in serverless: unfreeze when it's time to service an event. Bill for non-frozen time. Freeze after request (so user can't get free compute with a background thread).

# Anecdote: Early Version of Google Cloud Functions

## **Peeking Behind the Curtains of Serverless Platforms**

**Liang Wang, UW-Madison; Mengyuan Li and Yinqian Zhang, The Ohio State University;  
Thomas Ristenpart, Cornell Tech; Michael Swift, UW-Madison**

<https://www.usenix.org/conference/atc18/presentation/wang-liang>

<https://www.usenix.org/system/files/conference/atc18/atc18-wang-liang.pdf>

*"Background processes. We found in Google one could execute an external script in the background that continued to run even after the function invocation concluded. The script we ran posted a 10 M file every 10 seconds to a server under our control, and the longest time it stayed alive was 21 hours. We could not find any logs of the network activity performed by the background process and were not charged for its resource consumption."*

Perhaps the freezer wasn't being used correctly?

# cpuset

```
echo "2,3" > cpuset.cpus
```

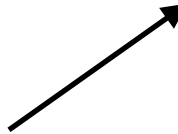
← this group of processes will  
only run on these cores

- guarantees will not run on other cores
- they are not guaranteed these cores -- other things might get scheduled here
- good: get warmer CPU cache
- bad: other cores might be idle, but we can't use them even if we're doing lots of compute

# cpu

- cpu is more flexible: specify share or limit of resource, without constraining WHERE it is achieved
- lots of configs:

<code>cpu.idle</code>	<code>cpu.stat</code>	<code>cpu.max.burst</code>	<code>cpu.pressure</code>
<code>cpu.max</code>	<code>cpu.uclamp.max</code>	<code>cpu.uclamp.min</code>	<code>cpu.weight.nice</code>
			<code>cpu.weight</code>



"When distributing CPU time to the children, all of their `cpu.weight` values are summed up and then each active child gets CPU in proportion to their weight relative to the total. This means that if all `cpu.weight` files have the same value, all children will get equal shares of the CPU time. The actual `cpu.weight` values only matter if they're different; if they're all the same, the value is arbitrary."

<https://utcc.utoronto.ca/~cks/space/blog/linux/CgroupV2FairShareScheduling>

- for sole tenant, you probably want to never waste cores when something is runnable
- when customers are billed per compute, you probably don't want to give away extra cores for free just to be nice

# cpu

- cpu is more flexible: specify share or limit of resource, without constraining WHERE it is achieved
- lots of configs:

cpu.idle	cpu.stat	cpu.max.burst	cpu.pressure
cpu.max	cpu.uclamp.max	cpu.uclamp.min	cpu.weight.nice
			cpu.weight

runtime    walltime (both in microseconds)

echo "50000 100000" > cpu.max

out of each 100ms period,  
only give 50ms of compute

echo "5000 10000" > cpu.max

same ratio, but over 10ms period. Shorter: more  
interactive, but less efficient (context switches)

echo "200000 100000" > cpu.max

runtime > walltime, meaning can use  
> 1 core (in this case, 2 cores).

# Outline

Performance Isolation

**Mechanism** Examples

**Interface:** cgroup overview

Controllers

- freezer/cpu/cpuset
- **memory**
- io (disk)
- pids
- what about network?

Usage in OpenLambda

# memory

memory.events.local    memory.low  
memory.min            memory.oom.group            memory.stat  
memory.swap.events            memory.high            memory.numa\_stat  
memory.pressure            memory.swap.current            memory.swap.highMechanism  
**memory.max**            **memory.swap.max**            **memory.current**            **memory.events**

hard limit on RAM usage.  
Exceed it and the OOM  
killer gets you. (bytes)

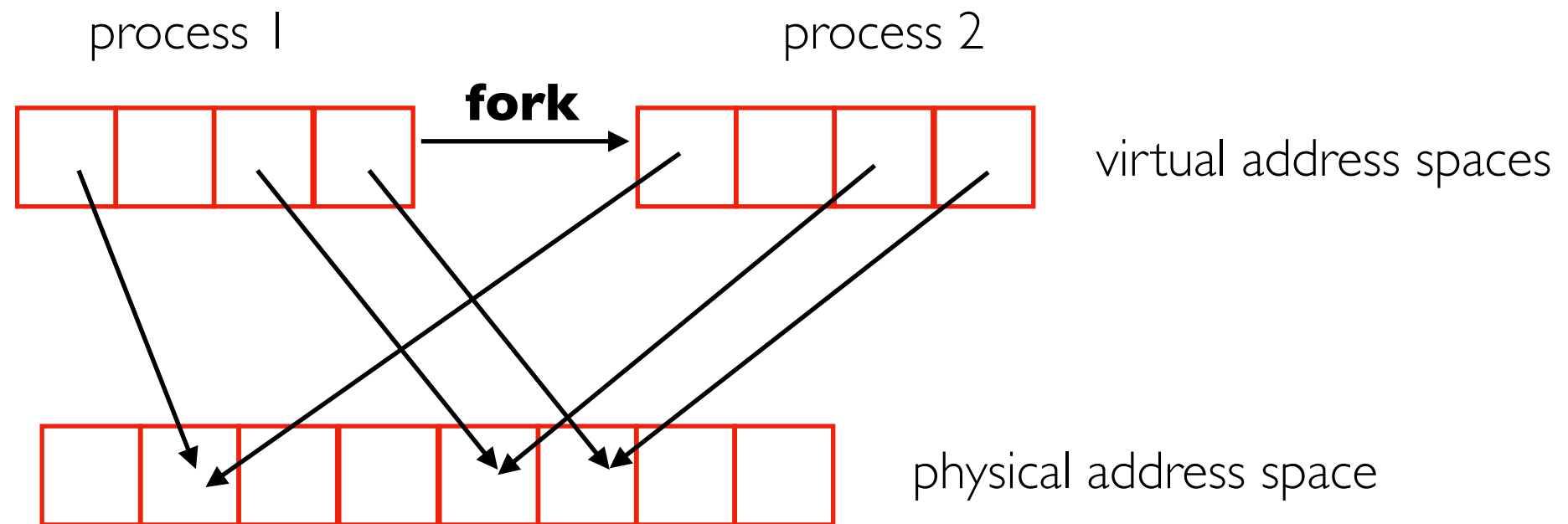
bytes of space that can be  
used in a swapfile.

current memory usage  
(in bytes).

can be read to determine  
whether the OOM killer  
terminated a process. Very  
useful for generating a useful  
error message for the user.



# Who to charge?

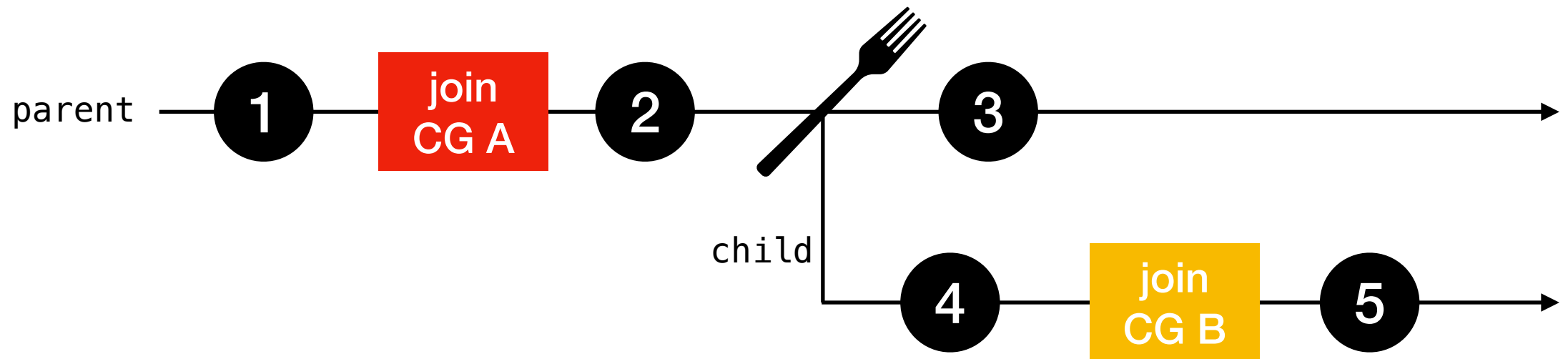


We're using 12 KB (3 pages) of physical memory total. How should we do accounting across the two processes?

- 6 KB each? If process 2 stops, does the memory accounting for process 1 pop, leading to a kill?
- 12 KB for process 1, since it allocated it first? Why should process 2 have a free ride?

No good answers here, and even if we know what we want, cgroups provide limited control.

# Accounting (cgroup V1 experiment)



**Each circle above is a memory allocation.  
What should be blamed on A? On B?**

cgroups V1 has a "memory.move\_charge\_at\_immigrate" setting to provide some control.  
cgroups V2 is simpler and lacks this!

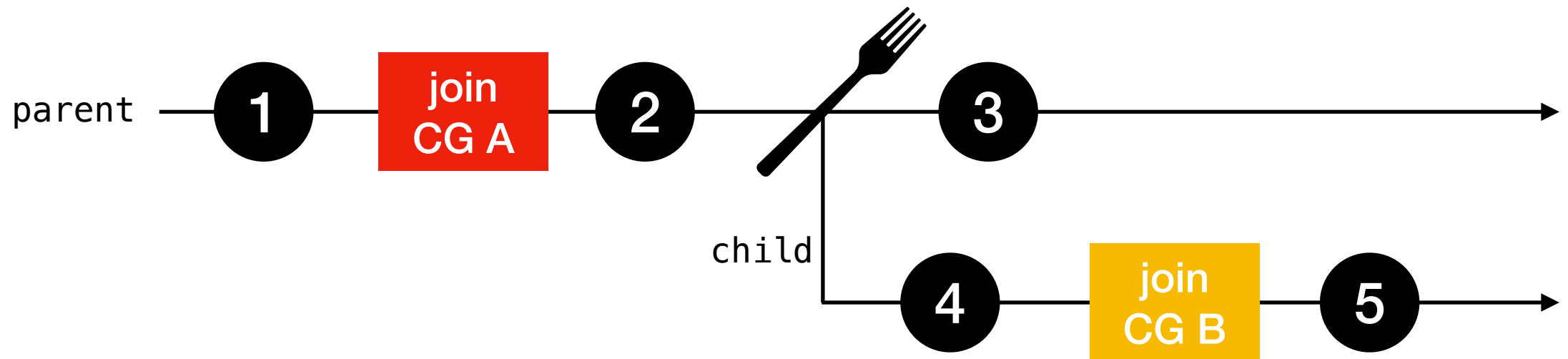
```
shell> bash
```

```
bash> mkdir /sys/fs/cgroup/memory/my-cg
```

```
bash> echo $$ > /sys/fs/cgroup/memory/my-cg/tasks # own PID
```

```
bash> echo 4M > /sys/fs/cgroup/memory/my-cg/memory.limit_in_bytes
```

# Accounting (cgroup V1 experiment)



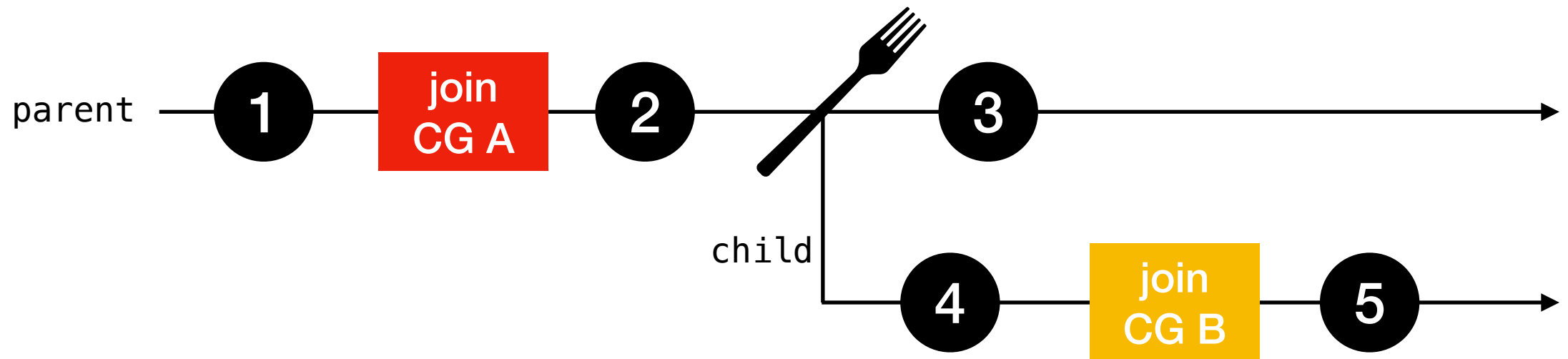
## Allocation Point

Move Charge on Immigrate

	1	2	3	4	5
neither	-	A	A	A	B
A	A	A	A	A	B
B	-	B	A	B	B
both	B	B [96%]	A	B [84%]	B

```
bash> echo 1 > /sys/fs/cgroup/memory/my-cg/memory.move_charge_at_immigrate
```

# Accounting (cgroup V1 experiment)



## Allocation Point

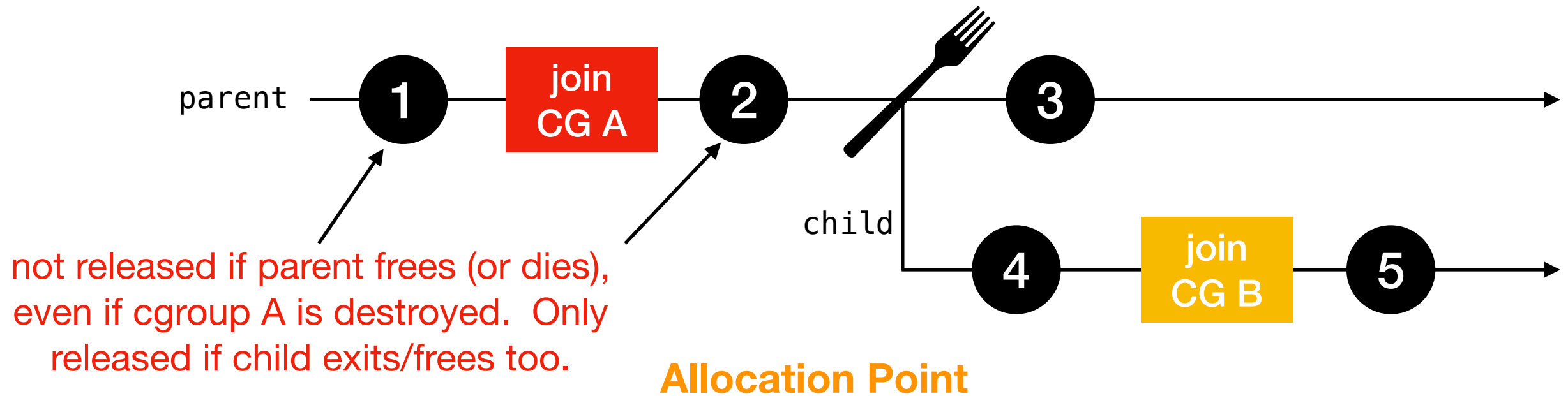
Move Charge on Immigrate

	1	2	3	4	5
neither	-	A	A	A	B
A	A	A	A	A	B
B	-	B	A	B	B
both	B	B [96%]	A	B [84%]	B

"leak" ~300KB  
per fork

**Implication:** ONLY use `move_charge_on_immigrate` for parentless sandboxes.  
Avoid accruing memory charges between fork and joining child CG.

# Accounting (cgroup V1 experiment)



Move Charge on Immigrate

	1	2	3	4	5
neither	-	A	A	A	B
A	A	A	A	A	B
B	-	B	A	B	B
both	B	B [96%]	A	B [84%]	B

**Implication:** never assume memory can be reclaimed from a dead sandbox until its children exit too.

# Outline

Performance Isolation

**Mechanism** Examples

**Interface:** cgroup overview

Controllers

- freezer/cpu/cpuset
- memory
- io (disk)
- pids
- what about network?

Usage in OpenLambda

# io

The io controller is for disk only.

`io.max` is like `cpu.max`, in that it sets a hard cap regardless of what other processes are doing. There is more granularity to differentiate reads/writes, etc.

`io.weight` is like `cpu.weight`.

Both these settings can indicate which device the configs apply to.

Remember the limitations of I/O scheduling mentioned earlier! It's hard to be fair across tasks if you don't know which task originated a write.

Note: I'm not aware of a cgroup controller for limited disk space used. Better to look into quota settings for particular local FS in use?

# Outline

Performance Isolation

**Mechanism** Examples

**Interface:** cgroup overview

Controllers

- freezer/cpu/cpuset
- memory
- io (disk)
- pids
- what about network?

Usage in OpenLambda



# pids

`pids.max` limits the number of processes in a cgroup. Prevent fork bombs from getting out of control.

# Outline

Performance Isolation

**Mechanism** Examples

**Interface:** cgroup overview

Controllers

- freezer/cpu/cpuset
- memory
- io (disk)
- pids
- what about network?

Usage in OpenLambda

# Network I/O

## V1

- net\_cls cgroup could specify a classid, used by firewall and traffic shaping config.
- net\_prio cgroup could specify priority per network interface

## V2

- network related controllers were removed

Extended Berkeley Packed Filters (eBPF): <https://en.wikipedia.org/wiki/EBPF>

- provide way to run safe code in the kernel to make decisions about blocking/prioritizing packets
- eBPF filters can be written that observe cgroup names and make decisions accordingly

# Outline

Performance Isolation

**Mechanism** Examples

**Interface:** cgroup overview

Controllers

- freezer/cpu/cpuset
- memory
- io (disk)
- pids
- what about network?

Usage in OpenLambda

# SOCK cgroup Usage

SOCK (Serverless Optimized Container) is main the sandbox engine in OpenLambda and uses cgroups.

SOCK paper (2018): we just created and joined all the cgroups (to get meaningful perf measurements) but didn't actually set any limits.

We have been using more settings over time. It's on ongoing process, but it's far from complete:

- disk/network I/O: we don't do anything
- pids controller: one configurable limit on proc count for all lambdas
- cpu controller: specify cpu.max for all (no cpuset)
- **memory** is where we have done the most work

# Admission + Eviction

## Version 1 (SOCK paper)

- Evict **idle** sandboxes if  $\text{cgroup\_usage\_sum} > \text{soft\_limit}$
- Only admit new sandboxes if  $\text{cgroup\_usage\_sum} < \text{hard\_limit}$

## Problems:

1. what if every sandbox is active?
2. containers growing their mem footprint aren't subject to admission

# Admission + Eviction

## Version 1 (SOCK paper)

- Evict **idle** sandboxes if `cgroup_usage_sum > soft_limit`
- Only admit new sandboxes if `cgroup_usage_sum < hard_limit`

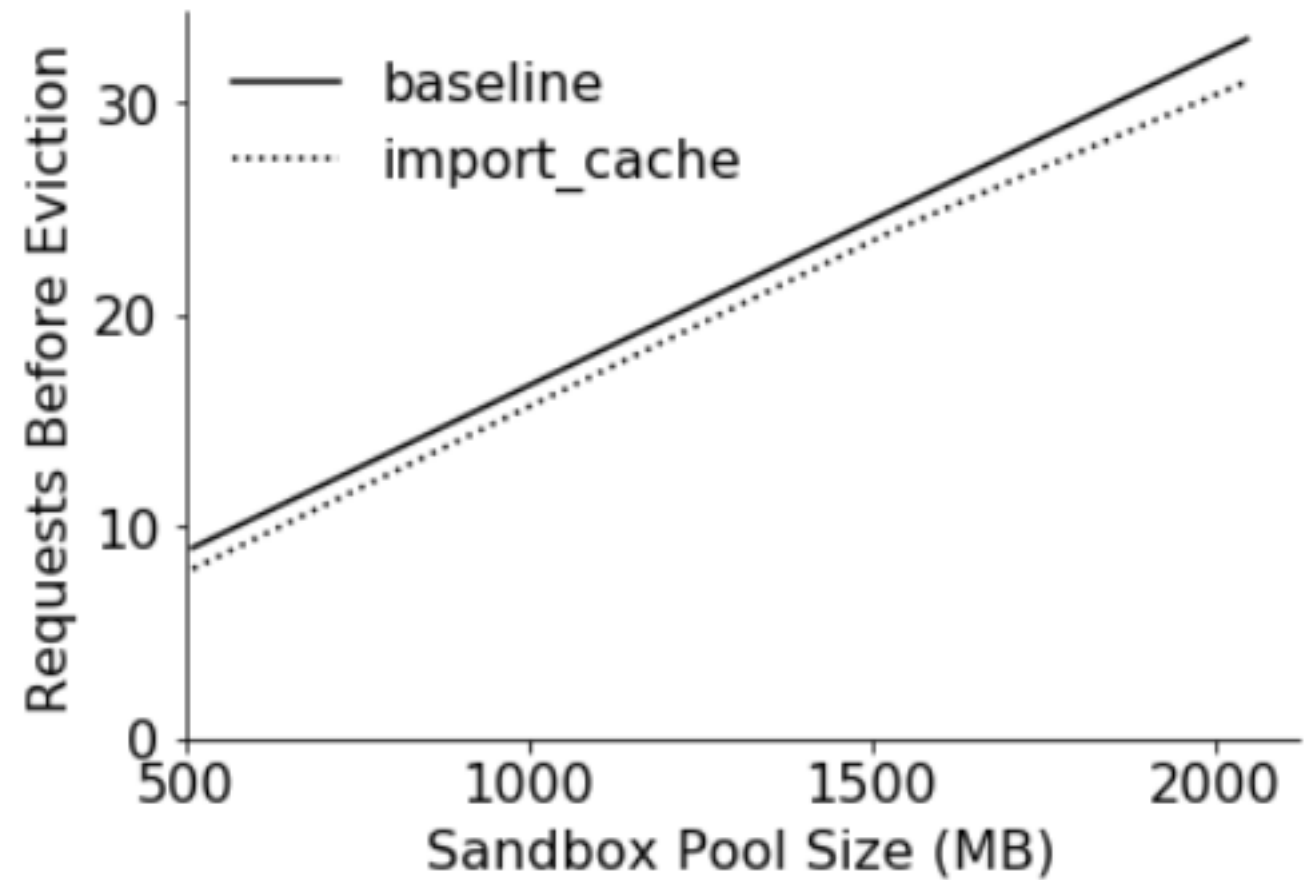
## Problems:

1. what if every sandbox is active?
2. containers growing their mem footprint aren't subject to admission

## Version 2

- Eviction policy: evict idle sandboxes that haven't been unfrozen for the longest time (basically LRU, where usage is unfreezing)
- Only admit new sandboxes if `cgroup_limit_sum < hard_limit`

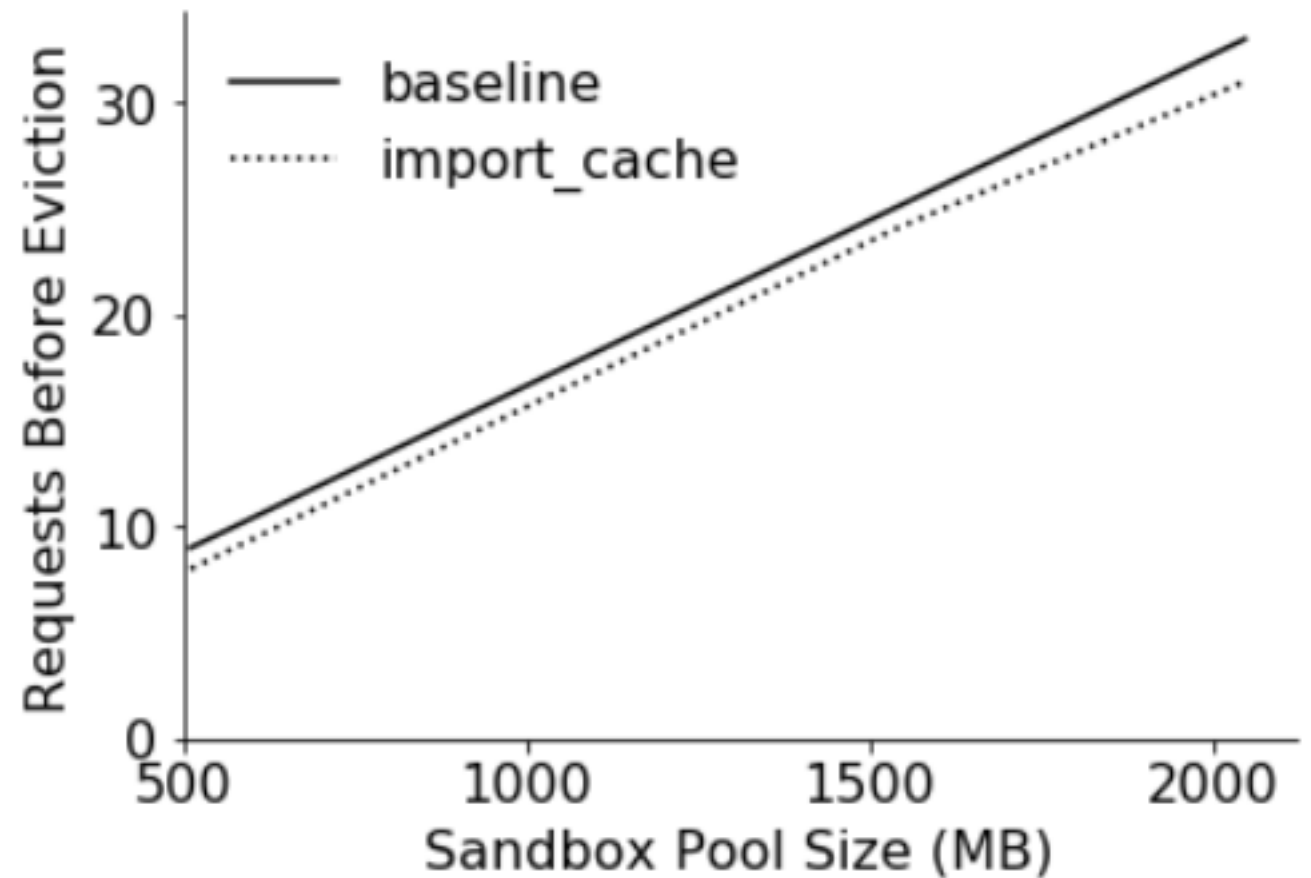
**no-op lambdas,  
call each once**



Both lines are the new (v2) admission+eviction policy,  
with and without the Zygote import cache enabled



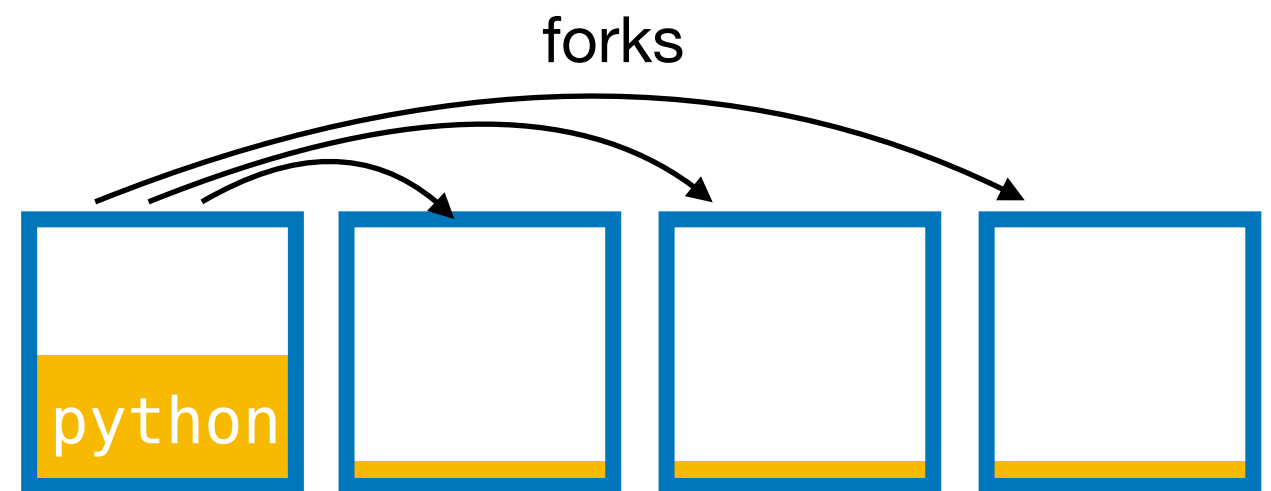
**Problem:** import cache reduces memory usage; that doesn't matter if we admit based on limits (not usage)



more usage, lower cumulative cap

**[without Zygoter]**

**vs.**



less usage, higher cumulative cap

**[with Zygoter]**

# Admission + Eviction

## Version 1 (SOCK paper)

- Evict **idle** sandboxes if  $\text{cgroup\_usage\_sum} > \text{soft\_limit}$
- Only admit new sandboxes if  $\text{cgroup\_usage\_sum} < \text{hard\_limit}$

## Problems:

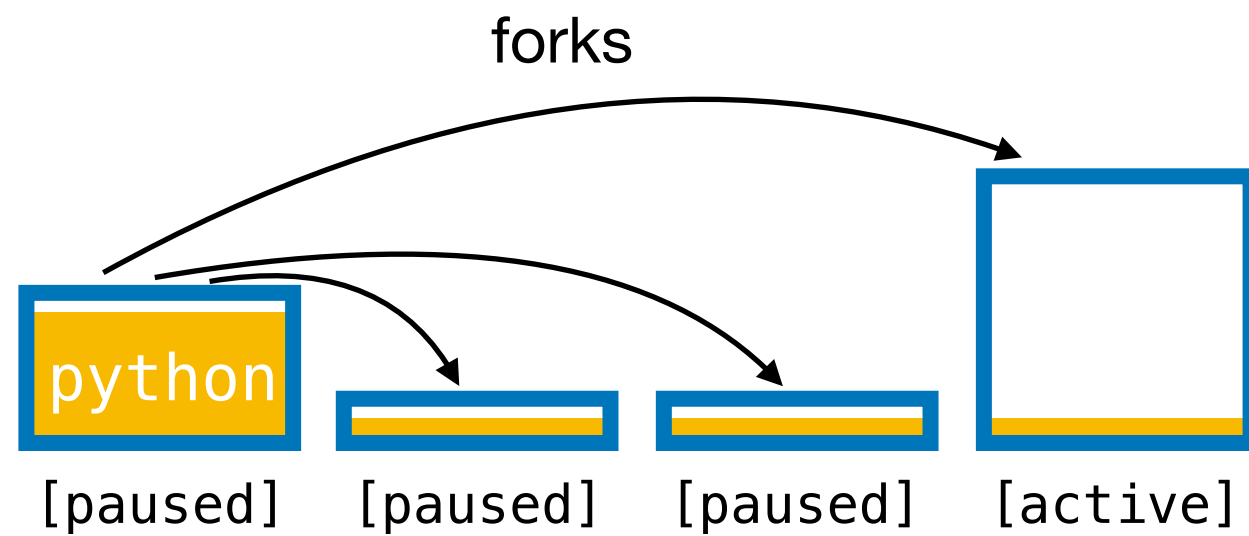
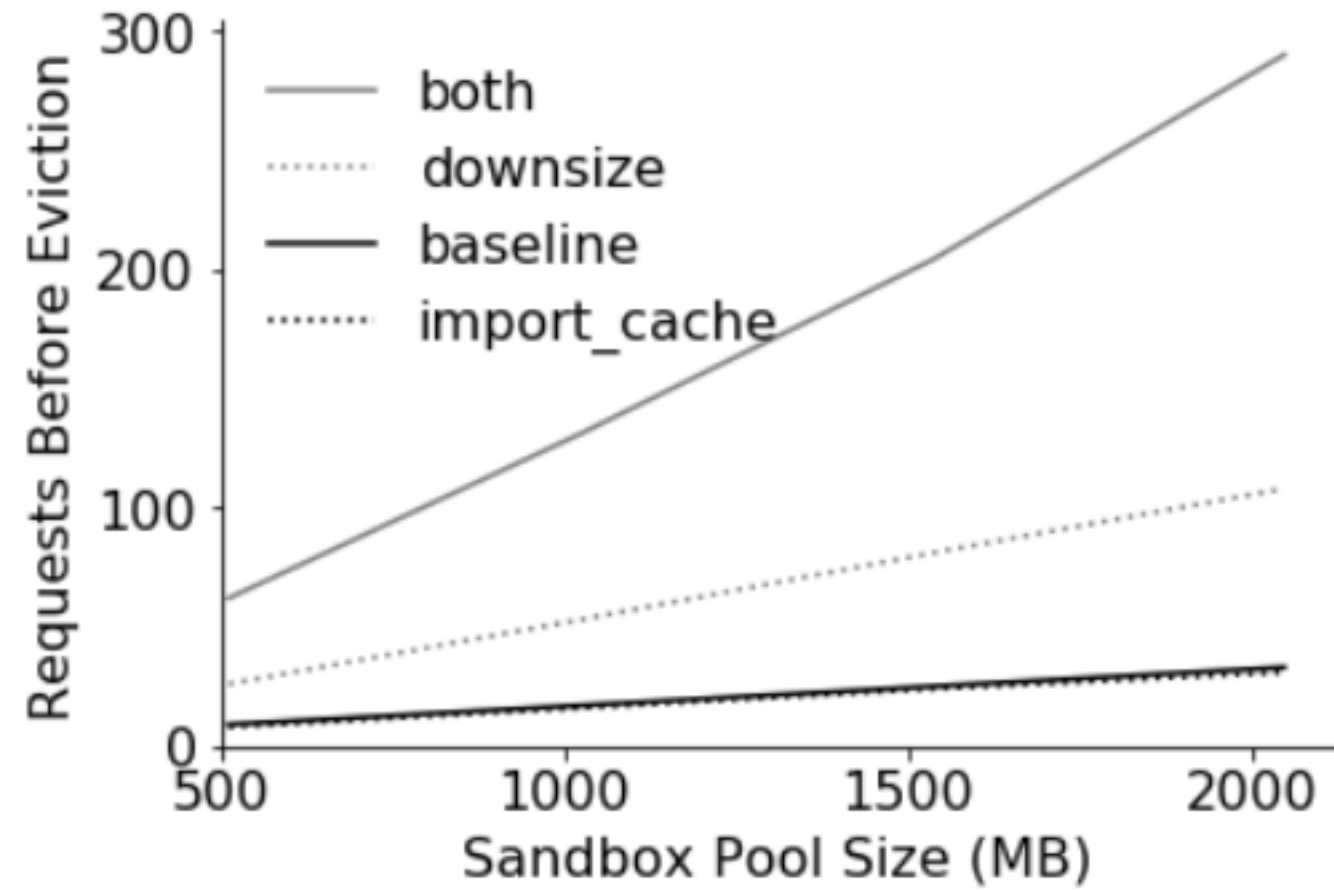
1. what if every sandbox is active?
2. containers growing their mem footprint aren't subject to admission

## Version 2

- Eviction policy described later...
- Only admit new sandboxes if  $\text{cgroup\_limit\_sum} < \text{hard\_limit}$

## Version 3

- Drop cgroup mem limit to actual usage when Sandbox is paused
- Otherwise same as V2



# Configs for cgroups

Relevant subset of worker's config.json file:

```
{
  "mem_pool_mb": 7449,
  "limits": {
    "procs": 10,
    "mem_mb": 50,
    "cpu_percent": 100,
    "max_runtime_default": 30,
    "swappiness": 0,
    "installer_mem_mb": 500
  },
  "features": {
    "reuse_cgroups": false,
  },
  "trace": {
    "cgroups": false,
    "memory": false,
    "evictor": false,
  },
}
```

sum of all cgroup memory limits will never exceed this (default depends on total system RAM available)

fork bomb will fail after 10 processes

memory per regular lambda (more than you think because the Python runtime gets accounted to a zygote)

installing packages with pip can use a lot of memory (because it often needs a compiler for C/Fortran dependencies)

this should probably always be off (will probably remove it)

# cgroups Going Forward in OpenLambda

See current code here: <https://github.com/open-lambda/open-lambda/tree/main/src/worker/sandbox/cgroups>

SOCK cgroups are probably due for a re-write

- configurable settings on a per-lambda/invocation basis
- use `cgroup.events` with `poll()` to wait for freeze and depopulate events
- use `CLONE_INTO_CGROUP` with `clone3()` so we don't need to move cgroups after creation (the lack of this leads to an ugly loop where we keep checking if new processes have been launched before the move so we don't leave any new ones behind)
- remove cgroup reuse (this subsystem is already brittle enough as is)
- set aside cgroups when `rmdir` fails (instead of panic'ing)
- use io limits; do something reasonable with swap
- add users with file quotes to accomplish space limits