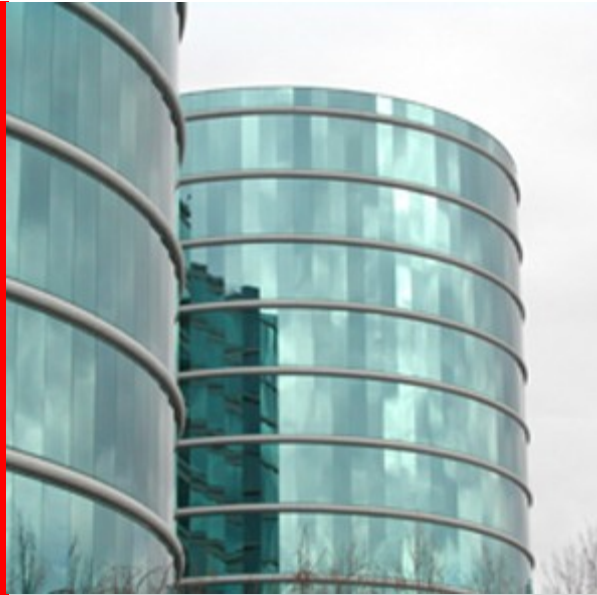




ORACLE®



**ORACLE<sup>®</sup>**

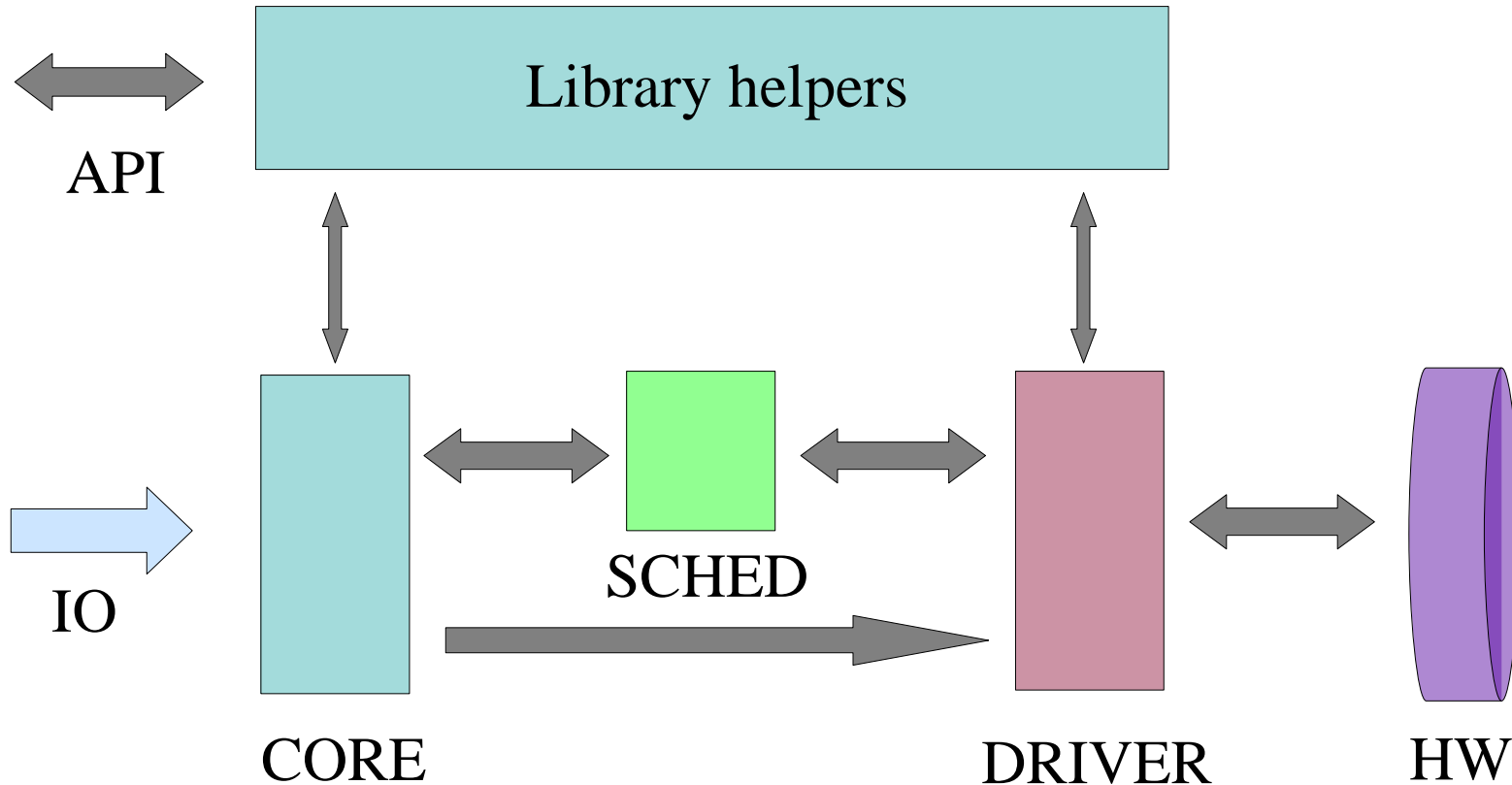
## **Linux IO stack and schedulers**

Jens Axboe <jens.axboe@oracle.com>  
Consulting Member of Staff

# Outline

- Block layer
- IO schedulers
- CFQ
- Some testing and benchmark results

# Block layer parts



# Library helpers

- sysfs interface
- Barrier handling
- IO context handling
- Tagged command queuing
- Hardware restriction handling
- Various utility functions

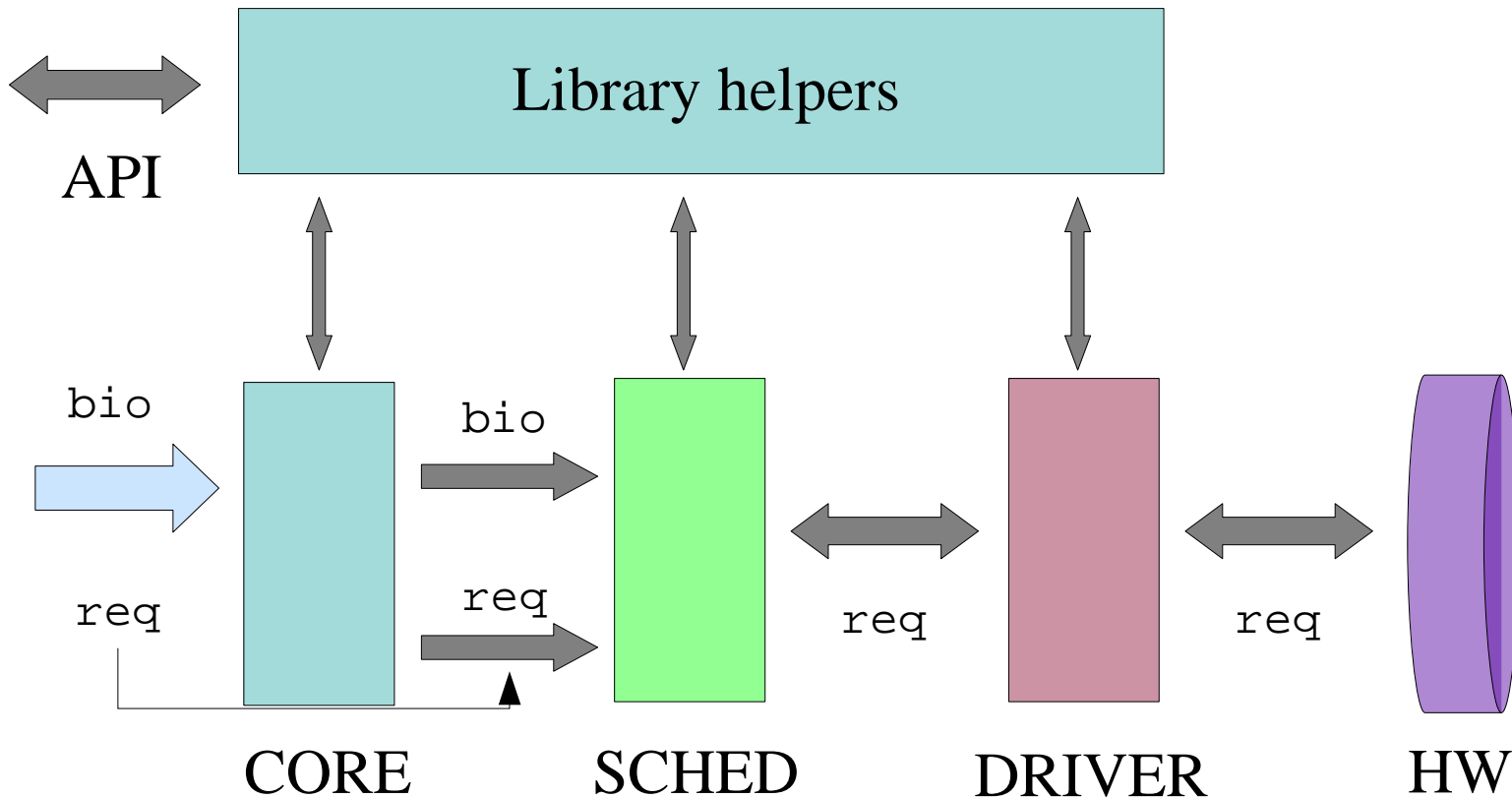
# BIO

- `struct bio` is the main IO unit
- Referenced counted object
- Contains meta data regarding direction, offset, etc
- Holds state information (uptodate, error, etc)
- Holds a `struct bio_vec` array
  - Each `bio_vec` is a page/length/offset tuple
- Maximum data size of `bio` restricted by size of array
  - Typically 256 pages, or 1MiB on 4kb page size archs

# Request

- `struct request` is the *message* passed
- Holds an arbitrary number of sequential `bio`'s
- Holds state information for driver, block layer, and IO scheduler

# Block layer parts, units



# Building a bio

- `bio = bio_alloc(gfp_mask, nr_iovecs);`
  - `bio->bi_bdev = bdev; /* destination dev */`
  - `bio->bi_sector = sector; /* dest offset */`
  - `bio->bi_end_io = func; /* end io function */`
- `bio_add_page(bio, page, len, off);`
  - Adds another page at the end of bio
- `bio = bio_clone(orig, gfp_mask);`
  - Sets up a clone of orig
- Various modifier flags available

# IO interface (queue to complete)

- `submit_bio()` queues bio
  - `generic_make_request()` unrolls IO stack
    - `__make_request()`
      - io scheduler merge or insert
- `elv_next_request()` returns next request to driver
  - `elv_dequeue_request()` removes request
  - `elv_requeue_request()` requeues request
- `blk_complete_request()` signals completion
  - `blk_end_request()` ends all or parts of request
- The struct `request_queue` is the transport

# Plugging

- Think bath tub drain plug
- Unplug on condition
  - X number of requests ready
  - Y period of time has passed
  - Somebody asks for a page in flight
- Advantages
  - Better merging
  - Better ordering

# IO Ordering

- Linux does guarantee any ordering by default
- IO barriers
  - Soft barrier
  - Hard barrier
- Drive/driver implementation
  - Cache type restrictions
  - FUA
  - Tag ordering

# New developments

- Keyword: CPU local
- IO CPU affinity
- Single block device maps to multiple queues
- Explicit device plugging
- SSD optimizations
  - Queue carries a device “profile”
  - High IOPS rate appears not to be a problem



# IO Schedulers

# Drive performance

## Characteristics

- How the drive can help us
  - Command queuing (NCQ, TCQ)
    - Optimal seek pattern
    - Eliminate/reduce rotational latency
- Where the drive is mostly helpless
  - Associated/dependent requests
  - Competing IO streams
  - Fairness



# Current drive performance

	Avg seek	Avg rot. delay	Transfer speed
5400RPM	14 msec	5.6 msec	40 MiB/s
7200RPM	9 msec	4.2 msec	60 MiB/s
15000 RPM	4 msec	2.0 msec	90 MiB/s

	4K xfer	4K random read	64K random read
5400RPM	98 $\mu$ sec	203 K/sec	3020 K/sec
7200RPM	65 $\mu$ sec	301 K/sec	4490 K/sec
15000 RPM	43 $\mu$ sec	662 K/sec	9600 K/sec

# Linux IO Scheduling

## Schedulers

- Currently includes 4 IO schedulers
  - noop
    - No sorting (strict FIFO), does request merging
    - Very simple, ~60 lines of effective code
  - deadline
    - Assigns deadlines to requests
    - Otherwise CSCAN with a few twists
      - Direction batching
  - anticipatory (“as”)
    - Basic functional algorithm is like deadline
    - Adds request anticipation
  - CFQ
    - We'll get to that

# Loading and switching schedulers

- Configurable in kernel config
- elevator= boot parameter
  - Only for builtin schedulers
- modprobe deadline-iosched
- Online switching
  - `$ cat /sys/block/sda/queue/scheduler`  
noop [cfq] anticipatory deadline
  - `# echo noop > /sys/block/sda/queue/scheduler`
  - `$ cat /sys/block/sda/queue/scheduler`  
[noop] cfq anticipatory deadline

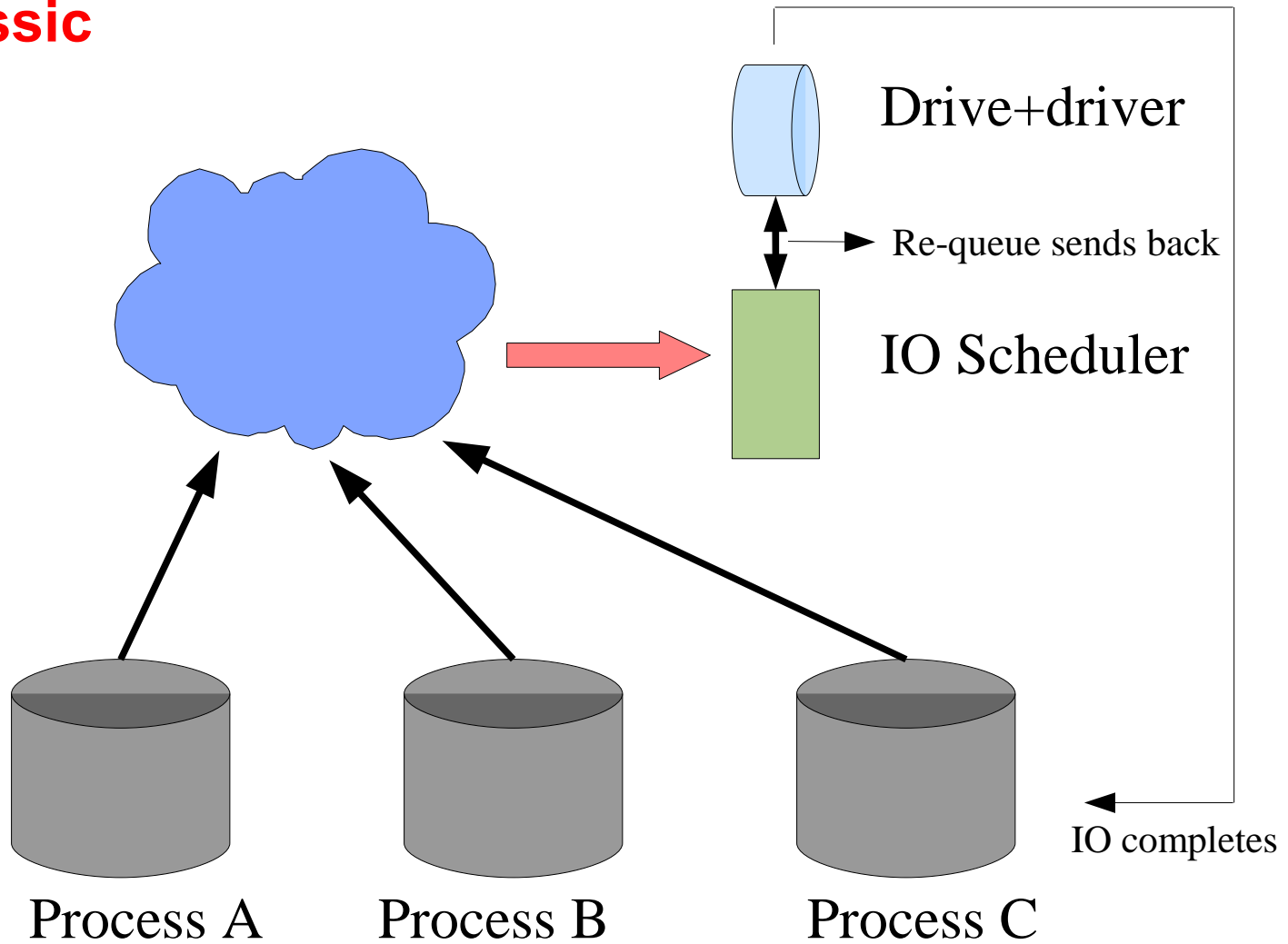
# Completely Fair Queuing

## CFQ v3

- First scheduler to tie process and IO
  - Persistent IO contexts
- Time slice design
  - Time based accounting, not request
  - Fairness across different io patterns
- Support for IO priorities
- Approx. 2300 lines of code
- Merged in 2.6.13 (v1 in 2.6.3, v2 in 2.6.10)
- Default scheduler since 2.6.18

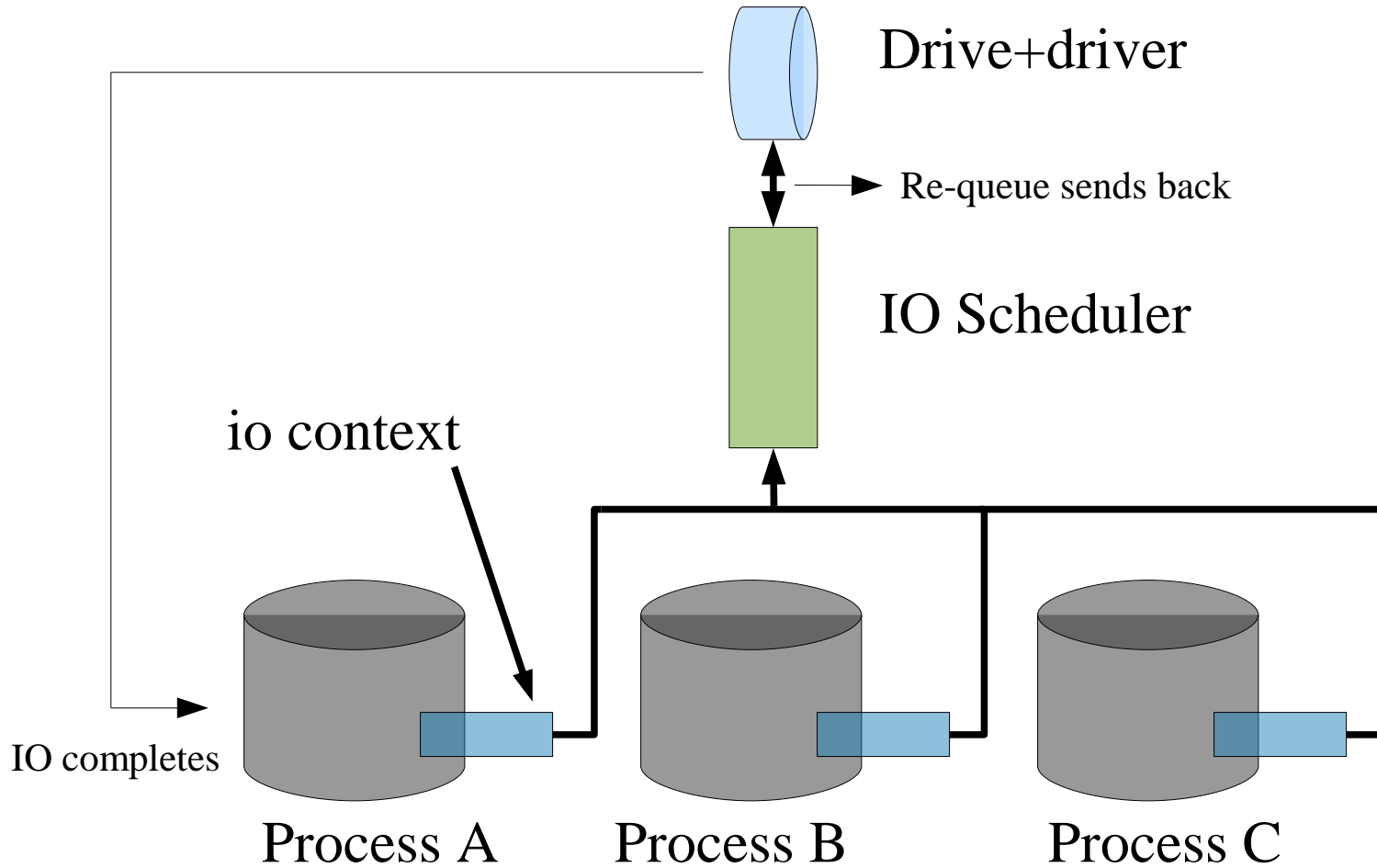
# IO Hierarchy

## Classic

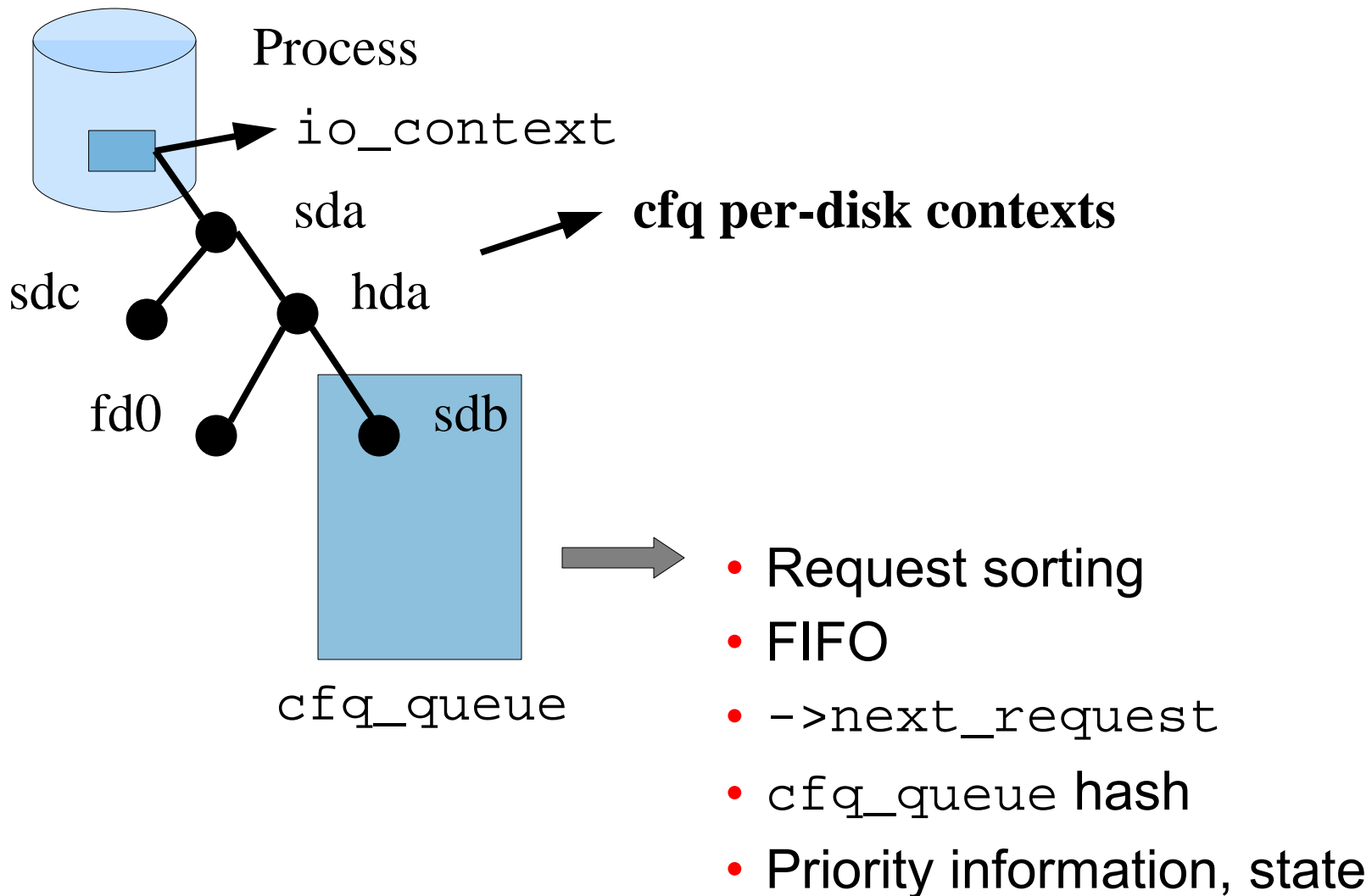


# IO Hierarchy

## CFQ

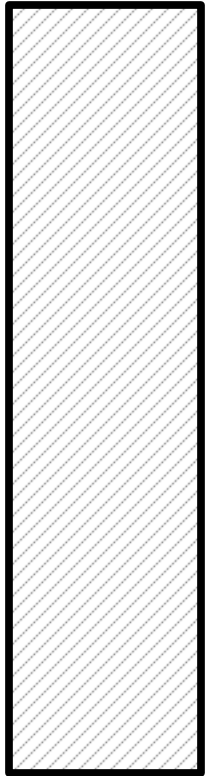


# Process $\leftrightarrow$ CFQ mapping

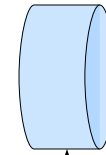


# CFQ per-queue data

cfq\_data



- best-effort lists
- busy list
- current list
- idle list
- hash of cfq\_queue
- io\_context pointer
- state and settings



Drive



Driver



Dispatch list

# Time slices

## Concept

- Simple to understand
  - Each process gets priority access to the disk for a given period of time
- Fair
  - Occasional/sync issuer gets as much time as queue flooder
  - Defined latency
- Synchronous slices
  - Time bounded only (100 msec at prio 4)
  - May idle
- Asynchronous slices
  - Time bounded (40 msec at prio 4)
  - Request bounded (2 at prio 4)
  - May not idle

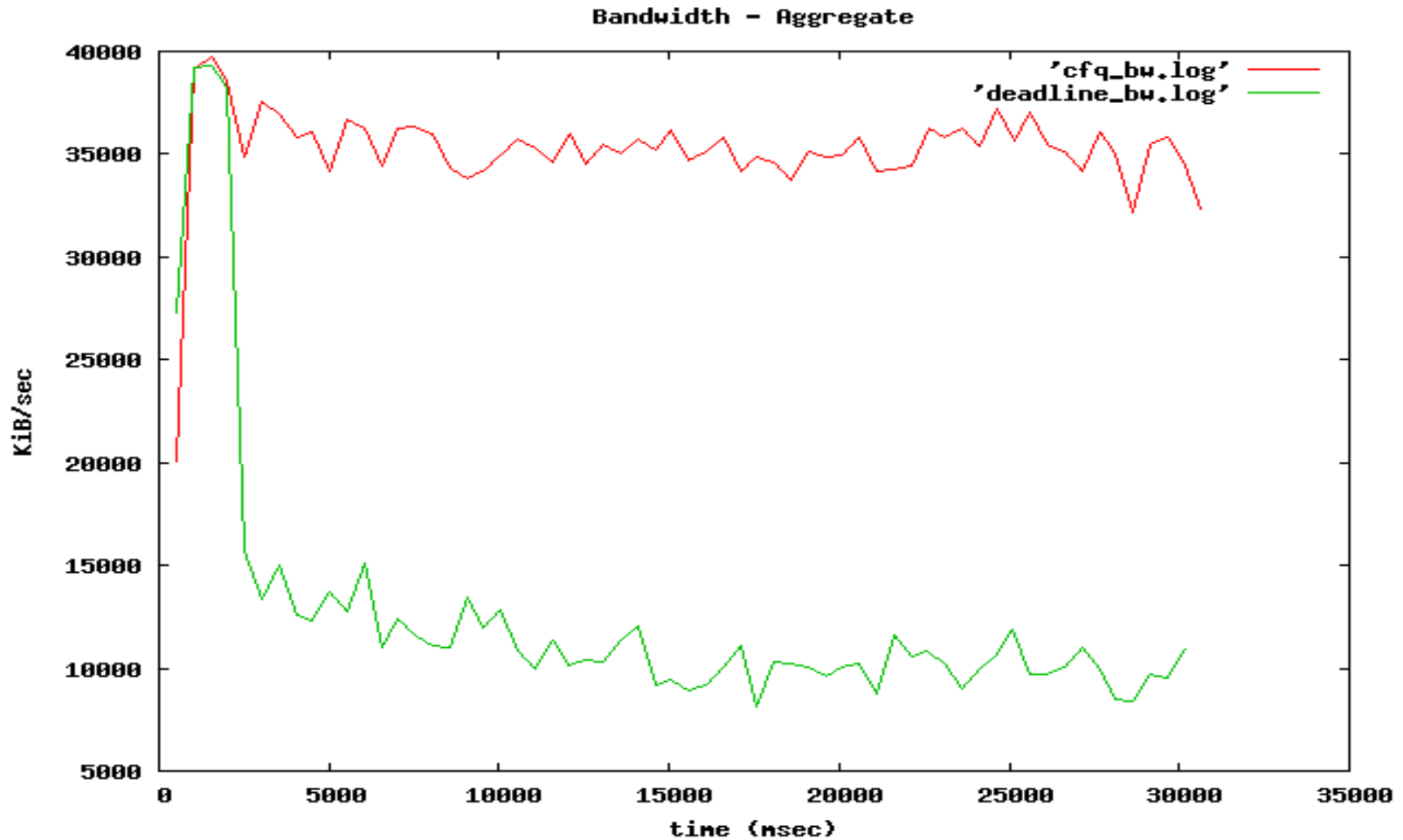
# Time slices

## Idling

- Not a work conserving scheduler
  - May decide to idle drive, even with work pending
- Why?
  - Expect close request
  - Seek time reduction
  - Several processes
- When not to idle
  - Process IO pattern is “seeky”
  - Process “waits” for too long between requests
  - Command queuing

# Streamed readers

## Example, CFQ vs DEADLINE



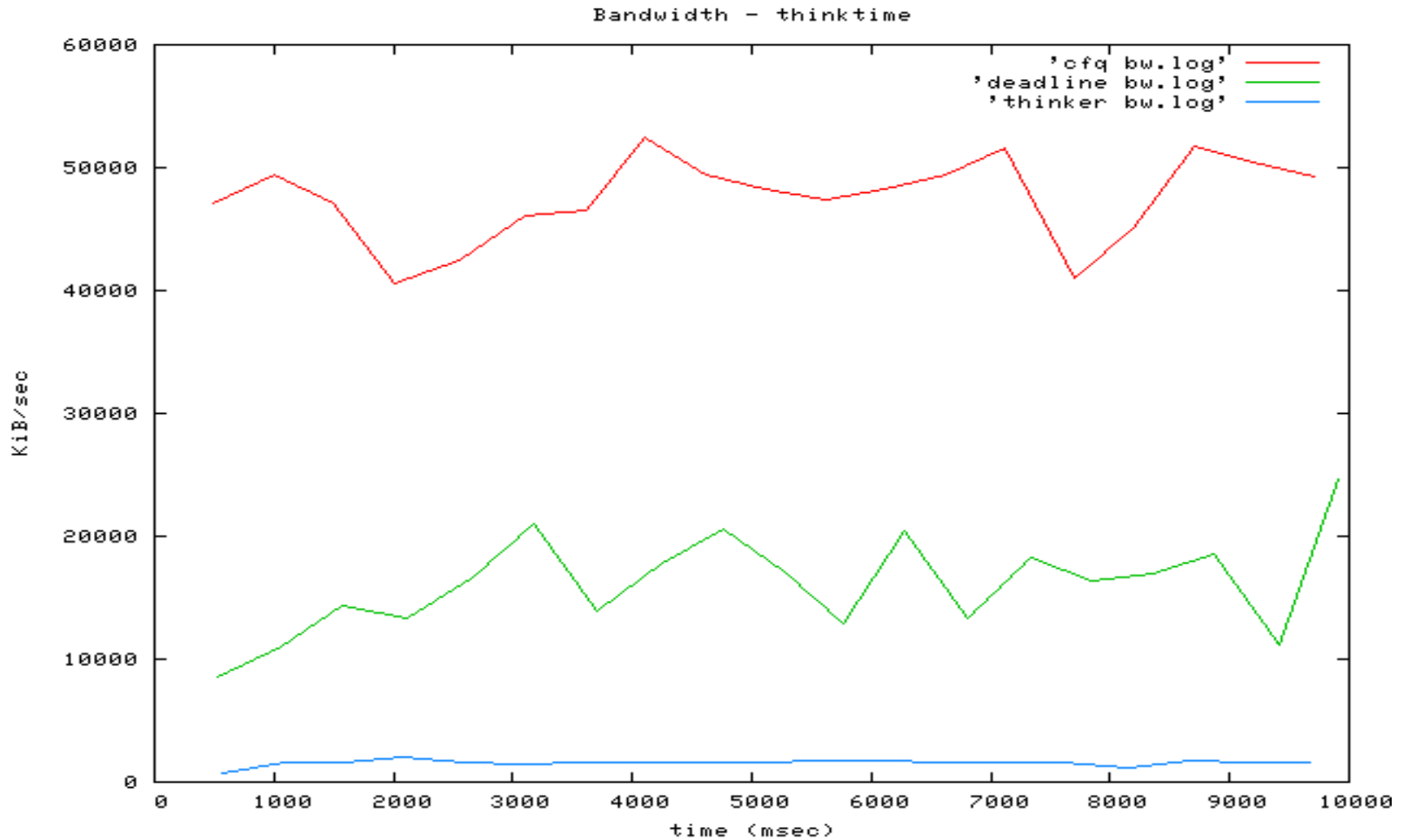
# Dependent reads

How many reads to load a 300 byte file?

- Monitor IO activity with blktrace
- exec: `vi /path/to/some/file`
- Total of 5 reads (28KiB)
  - Meta data + file data
- Undisturbed
  - Takes 66 msec
  - Imagine a delay between each operation

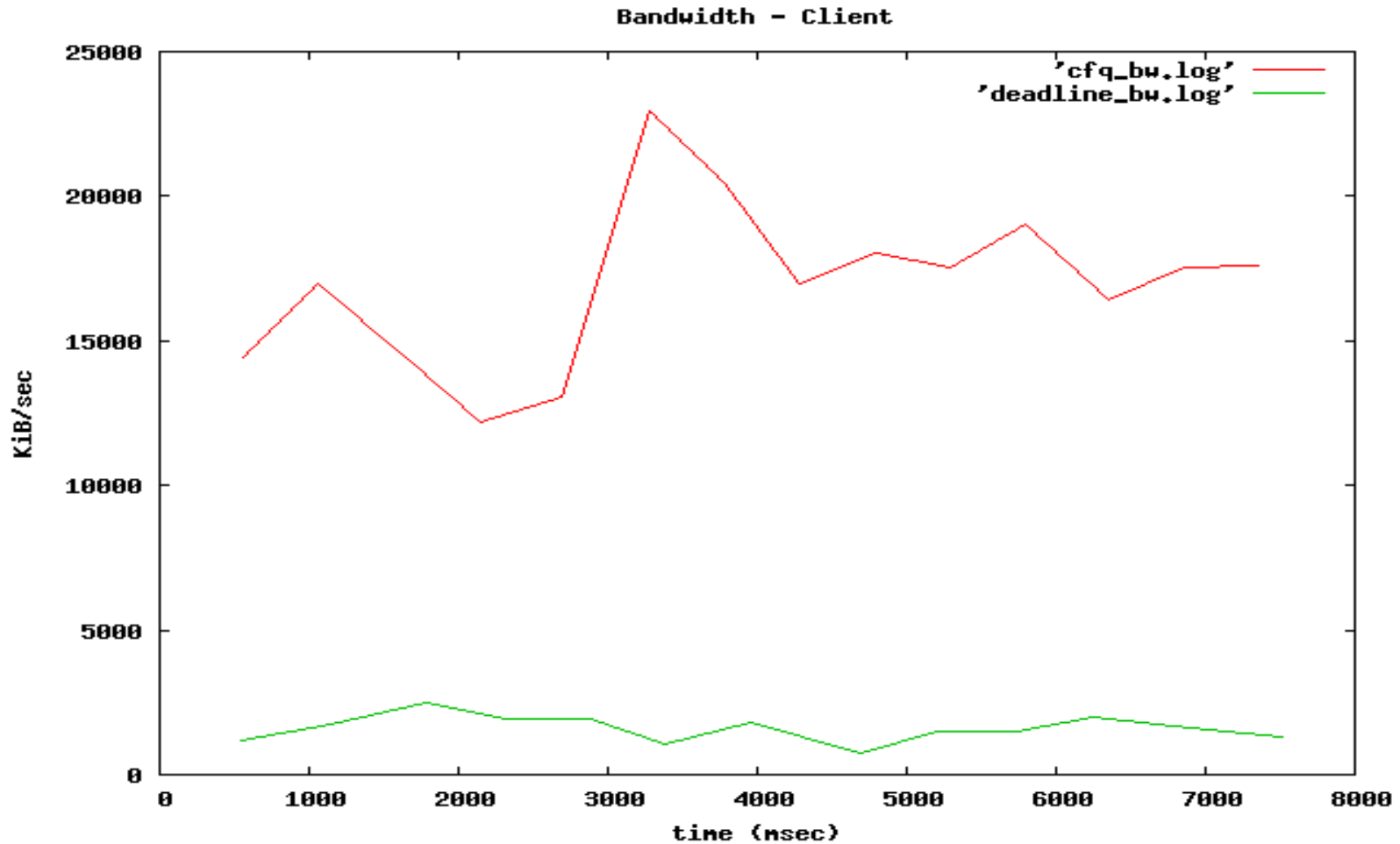
# Dependent reads

## Example, CFQ/DEADLINE vs thinker



# Reader vs writer

## Example, CFQ vs DEADLINE



# IO Priority Classes

- 3 default IO scheduling classes
  - Each with 8 sub levels, [0 – 7]
- Idle
  - Only gets access to disk, when nobody else uses it
  - Soft grace period
    - With hard grace period, used to be root only
- Best effort
  - Default class
- Real time
  - Always gets priority access to disk (goes straight to `cur_rr`)
  - Otherwise like best effort (same slice lengths)
  - root only

# IO Priorities

- 8 default levels
  - 0 the highest, 7 the lowest. Default is 4.
- Simple extension to time slices
  - Just scale slice with priority
- Can be set explicitly with `ionice`
  - `$ ionice -n <level> -c <class> [ -p <pid> ] [command]`
  - Inherited across forks
- Otherwise, follows `cpu nice`
  - Best effort class
  - `nice -20...-16: ionice 0`
  - `nice 0...4 ionice 4`
  - `nice 15...19: ionice 7`

# Benchmark, competing readers

## fio job file

- 8 simultaneous readers
  - 128 MiB files
- 4 KiB block size
- Write bandwidth log
  - 500 msec window average

```
[global]
bs=4k
buffered=1
rw=read
ioengine=sync
iodepth=1
size=128m
write_bw_log
```

```
[files]
numjobs=8
```

# Benchmark, competing readers

## Results

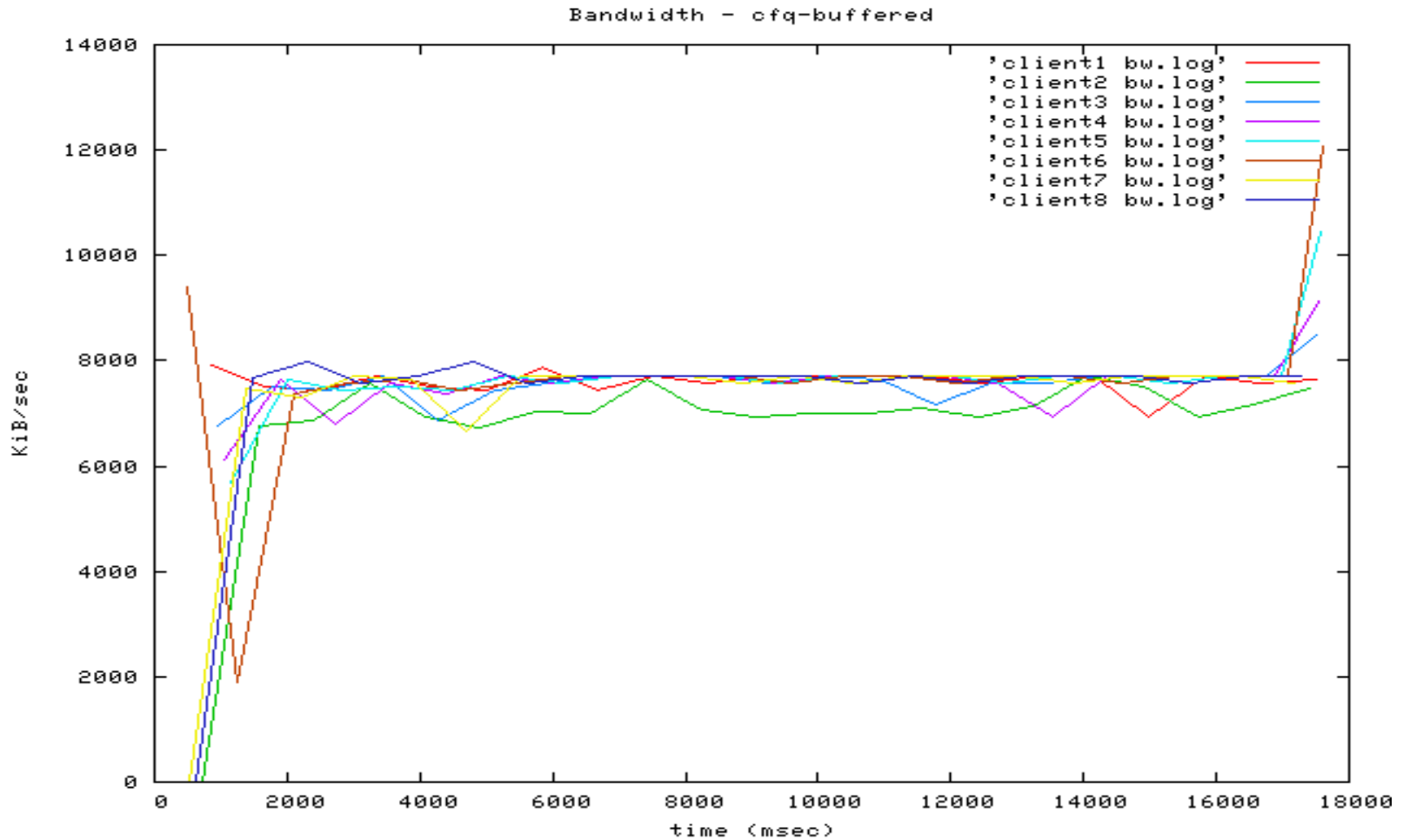
Single thread: ~62MiB/sec

	Min bw	Max bw	Aggr bw	% of max
CFQ	7535 K/s	7727 K/s	60295 K/s	94.9%
AS	7240 K/s	9451 K/s	57921 K/s	91.2%
DEADLINE	2462 K/s	2488 K/s	19700 K/s	31.0%

	Runtime	Max latency	Avg lat	Deviation
CFQ	17.8 secs	746 msec	0.5 msec	19 msec
AS	18.5 secs	890 msec	0.5 msec	17 msec
DEADLINE	54.5 secs	240 msec	1.7 msec	13 msec

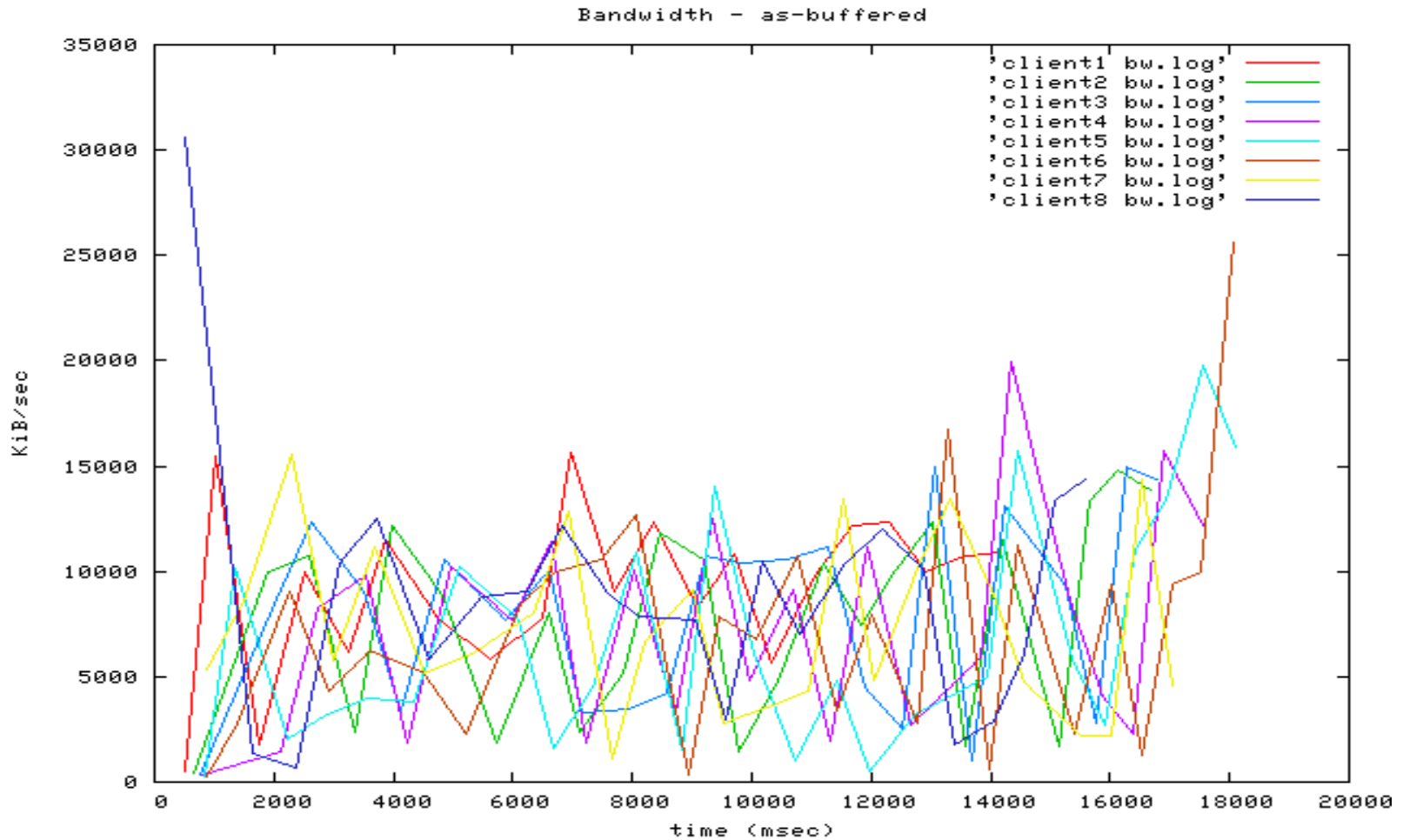
# Benchmark, competing readers

## Graphed bandwidth, CFQ



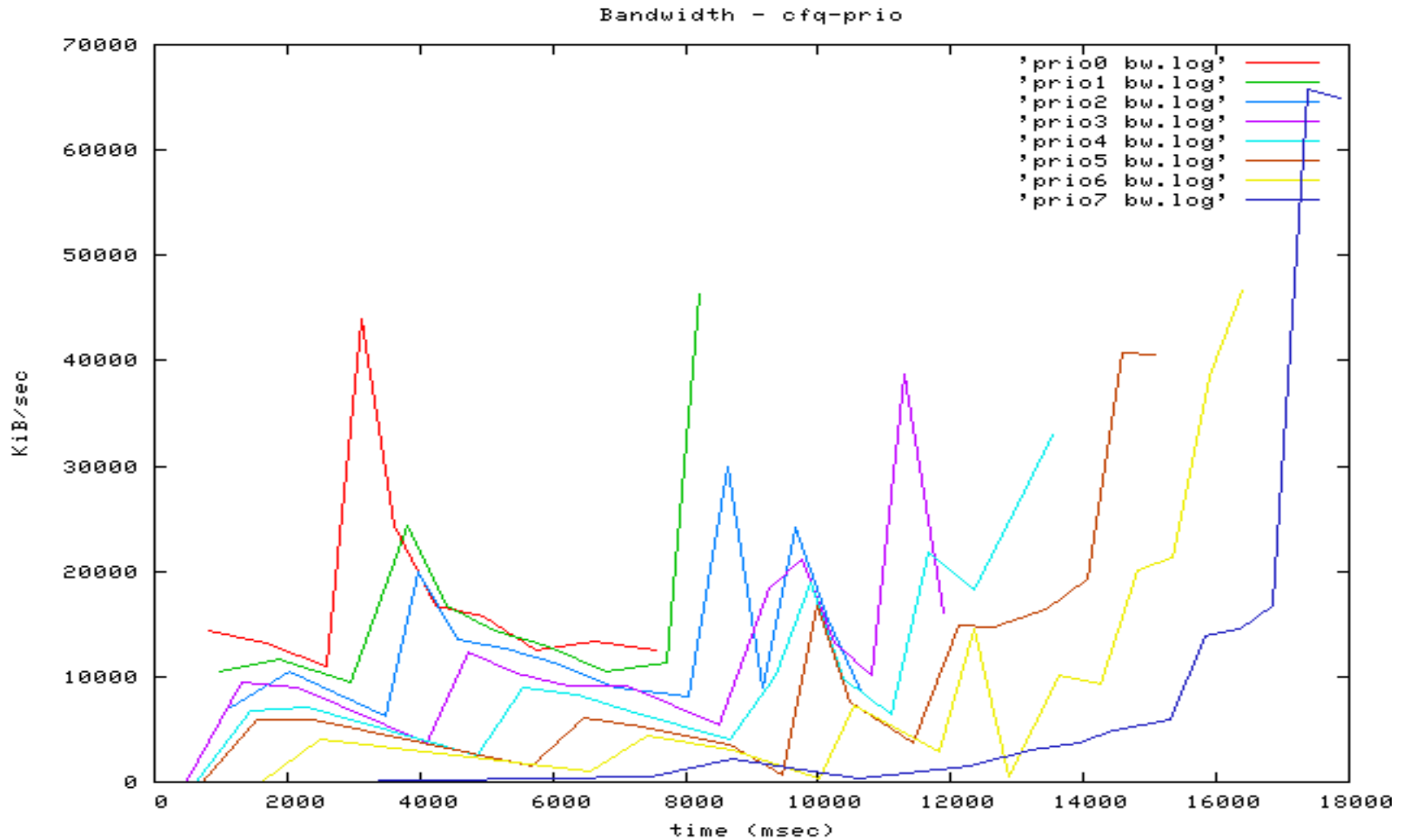
# Benchmark, competing readers

## Graphed bandwidth, AS



# Benchmark, competing readers

## Graphed bandwidth, CFQ priorities



# Resources

- Kernel files
  - `block/cfq-iosched.c`
  - `block/elevator.c`, `include/linux/elevator.h`
  - `block/blk-*.c`, `include/linux/blkdev.h`
  - `fs/bio.c`, `include/linux/bio.h`
- fio
  - `git clone git://git.kernel.dk/fio.git`
  - ... or get tar ball from <http://brick.kernel.dk/snaps/>
- blktrace
  - `git clone git://git.kernel.dk/blktrace.git`
  - Kernel 2.6.17 or newer
  - ... or get tar ball from <http://brick.kernel.dk/snaps/>

# Questions?



**ORACLE IS THE INFORMATION COMPANY**