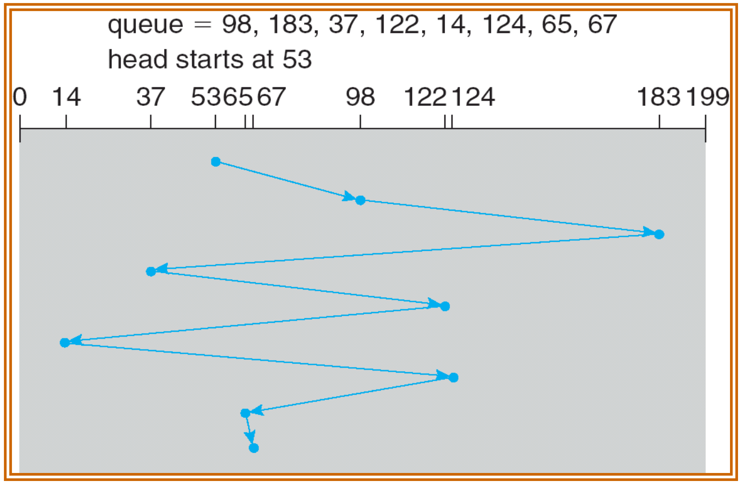
Unit 6: I/O Scheduler

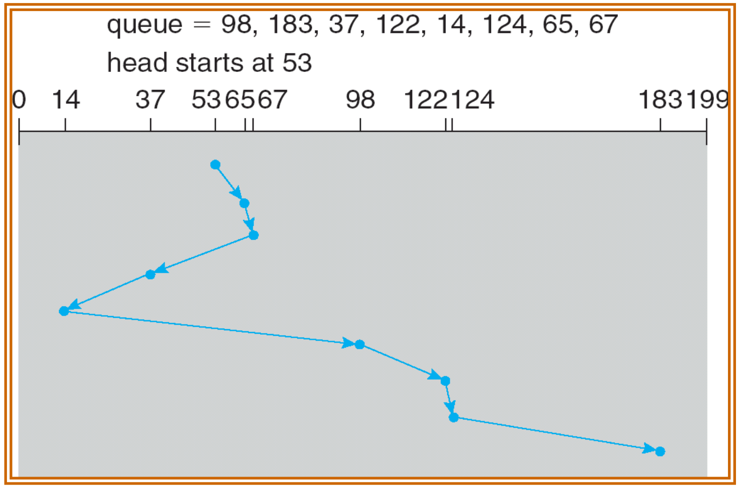
Prof. Li-Pin Chang
ESSLab@NCTU

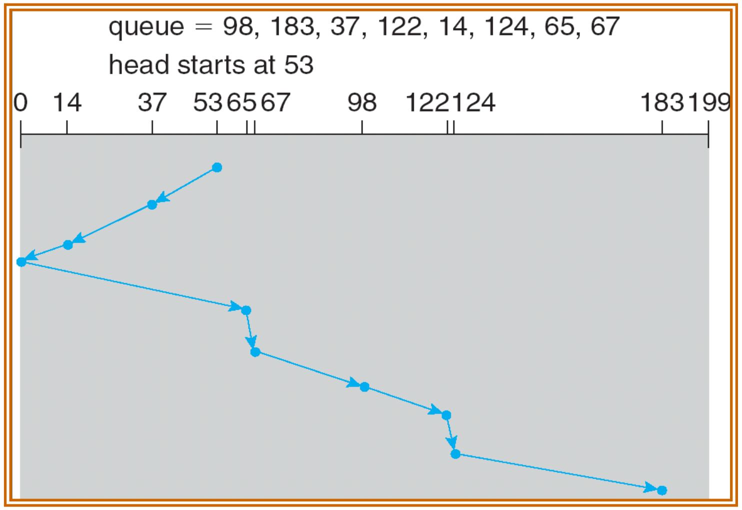
I/O Scheduler

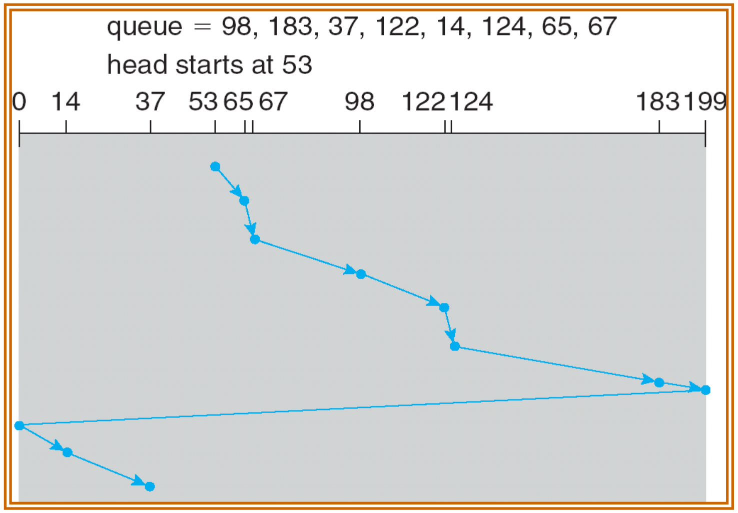
- The service time of a request depends on the disk location of the last completed request
- Deliberately delaying the service of disk requests may improve the global throughput
  - Merge adjacent requests to reduce I/O overhead
  - Reorder requests to minimize seek time
- Other consideration:
  - Request waiting time
  - Process-level share of the disk bandwidth
  - Work-conservative vs. non work-conservative

Review


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0 14 37 53 65 67 98 122 124 183 199


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0 14 37 53 65 67 98 122 124 183 199


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0 14 37 53 65 67 98 122 124 183 199


queue = 98, 183, 37, 122, 14, 124, 65, 67
head starts at 53
0 14 37 53 65 67 98 122 124 183 199

Terminology

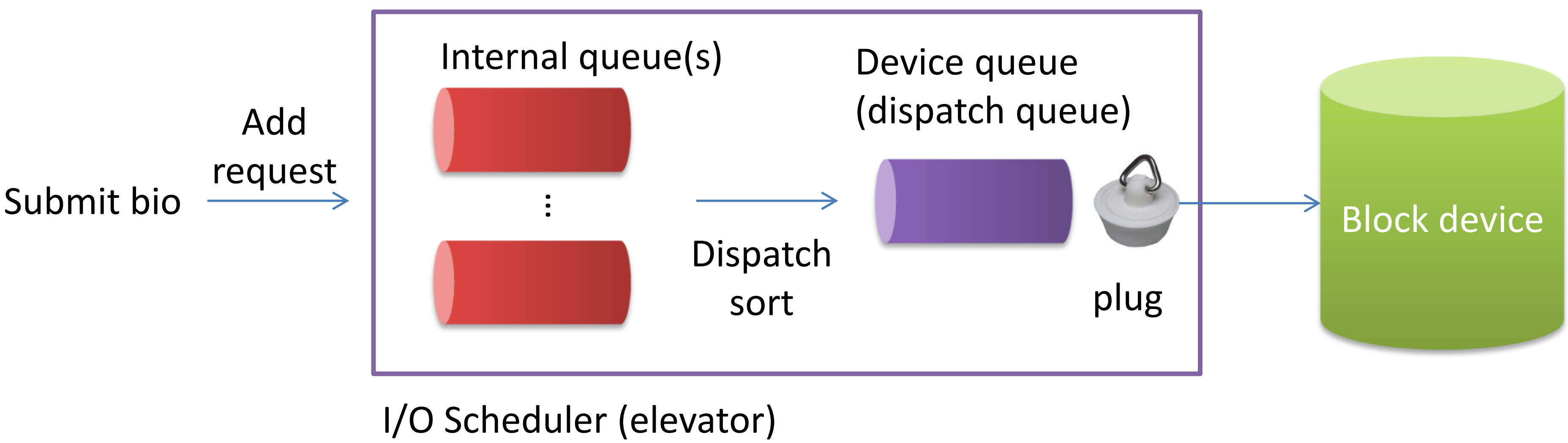
- I/O request = block request = disk request
- Elevator = I/O scheduler
- Block device Q= dispatch Q
- Elevator private Q= elevator internal Q

Request Characteristics

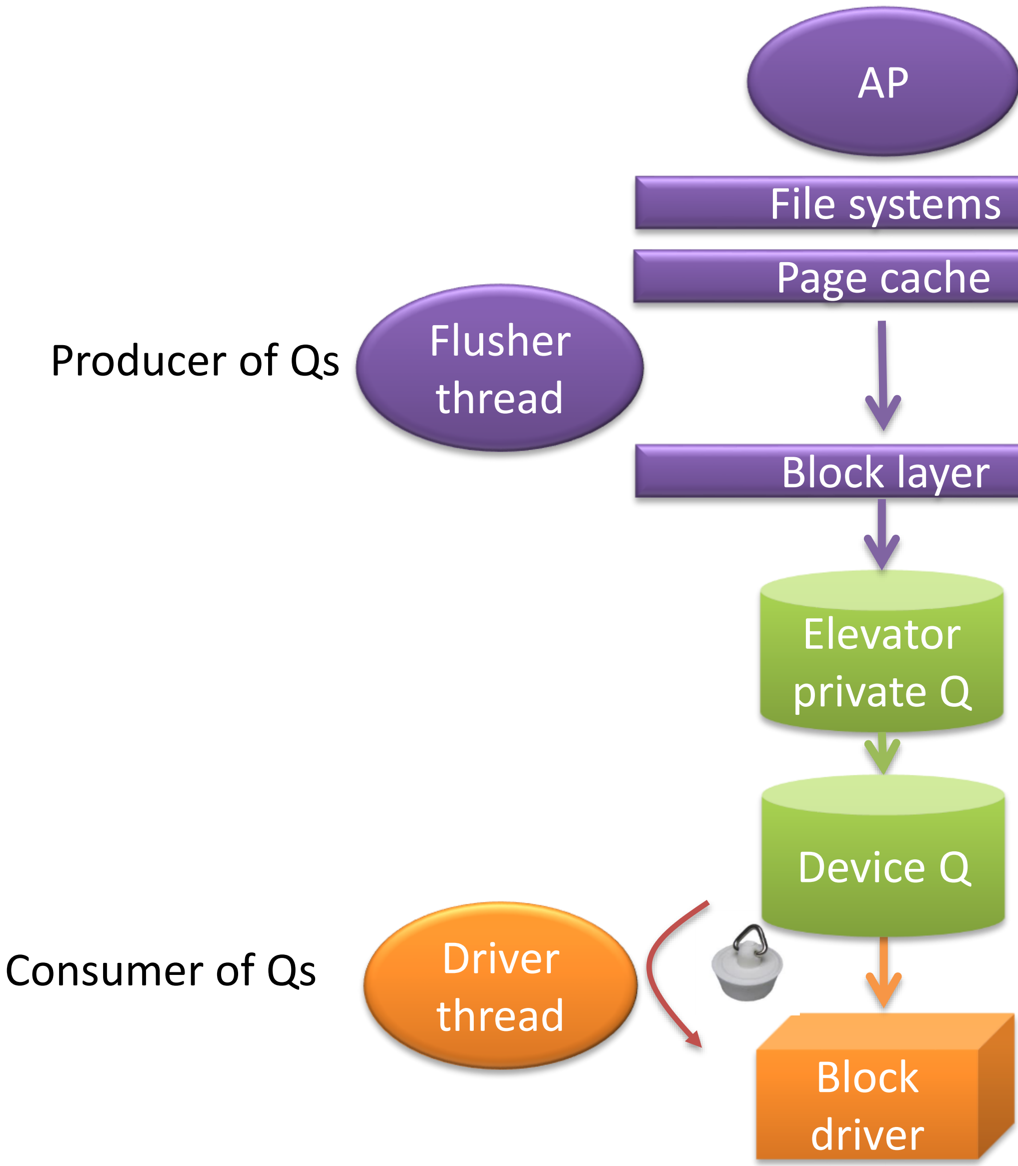
- Unless explicitly specified otherwise,
  - Read requests are synchronous
  - Write requests are asynchronous
- In the block layer, there is (are)
  - One read request per process
  - Many asynchronous write requests per process

Queue, Queue, and Queues

- The scheduler maintains its private queue(s)
  - High-level scheduling; considering bandwidth share or request deadlines
  - Low-level scheduling; considering seek-time minimization
- Each block device is associated with a dispatch queue
  - Usually has 1 or zero requests
  - Pull a request from the scheduler Q if the dispatch Q is empty



Producer-Consumer Paradigm



Elevator

- Schedulers are called “elevators” in Linux
  - Bad terminology
  - I/O schedulers has nothing to do with the elevator algorithm
- Four schedulers (elevators) in Linux 2.6
  - NOOP
  - Complete Fair Queuing (CFQ)
  - Deadline
  - Anticipatory (AS); the default scheduler

Binding Elevators to Block Devices

- Change the system default elevator
  - Add “elevator=xxx” in /boot/grub/grub.conf
  - cat /sys/block/DEV/queue/scheduler" - the list of valid names
  - as, noop, deadline, cfq

- Every block device can select a different elevator
  - Show the current scheduler of a block device via sysfs
    - cat /sys/block/ram0/queue/scheduler
    - cat /sys/block/sda/queue/scheduler

- Runtime change a block device’s elevator
  - echo “xxx” > cat /sys/block/ram0/queue/scheduler
  - Elevator change
    - generic_make_request() calls block_wait_queue_running()
    - wait until all pending requests complete
    - switch the elevator

Elevator Operations

- Add requests
  - Insert a request to the elevator queue
    - called by upper block layer
- Dispatch requests
  - Insert a request to the device request queue (dispatch queue)
    - called by a kernel thread associated with the block device driver

Function Name	functionallity
elevator_merge_req_fn	called when two requests get merged. the one which gets merged into the other one will be never seen by I/O scheduler again. IOW, after being merged, the request is gone.
elevator_merged_fn	called when a request in the scheduler has been involved in a merge. It is used in the deadline scheduler for example, to reposition the request if its sorting order has changed.
elevator_allow_merge_fn	called whenever the block layer determines that a bio can be merged into an existing request safely. The io scheduler may still want to stop a merge at this point if it results in some sort of conflict internally, this hook allows it to do that.
elevator_dispatch_fn*	fills the dispatch queue with ready requests. I/O schedulers are free to postpone requests by not filling the dispatch queue unless @force is non-zero. Once dispatched, I/O schedulers are not allowed to manipulate the requests - they belong to generic dispatch queue.
elevator_add_req_fn*	called to add a new request into the scheduler
elevator_queue_empty_fn	returns true if the merge queue is empty. Drivers shouldn't use this, but rather check if elv_next_request is NULL (without losing the request if one exists!)

Function Name	functionallity
elevator_former_req_fn/ elevator_latter_req_fn	These return the request before or after the one specified in disk sort order. Used by the block layer to find merge possibilities.
elevator_completed_req_fn	called when a request is completed.
elevator_may_queue_fn	returns true if the scheduler wants to allow the current context to queue a new request even if it is over the queue limit. This must be used very carefully!!
elevator_set_req_fn/ elevator_put_req_fn	Must be used to allocate and free any elevator specific storage for a request
elevator_activate_req_fn	Called when device driver first sees a request. I/O schedulers can use this callback to determine when actual execution of a request starts.
elevator_deactivate_req_fn	Called when device driver decides to delay a request by requeueing it.
elevator_init_fn* elevator_exit_fn	Allocate and free any elevator specific storage for a queue.

Block Drivers and Schedulers

- A block device driver can use I/O schedulers
  - Ramdisk does not, since scheduling has no effect on it
  - If a block driver wishes to use schedulers, it calls blk_init_queue() to register a callback function
    - E.g., hd.c registers do_hd_request()
- A block driver can create a private kernel thread that moves pulls requests from the dispatch Q for final processing
  - If the dispatch Q is empty, it call blk_fetch_request() to move requests from the scheduler Q to the dispatch Q

Request Arrivals

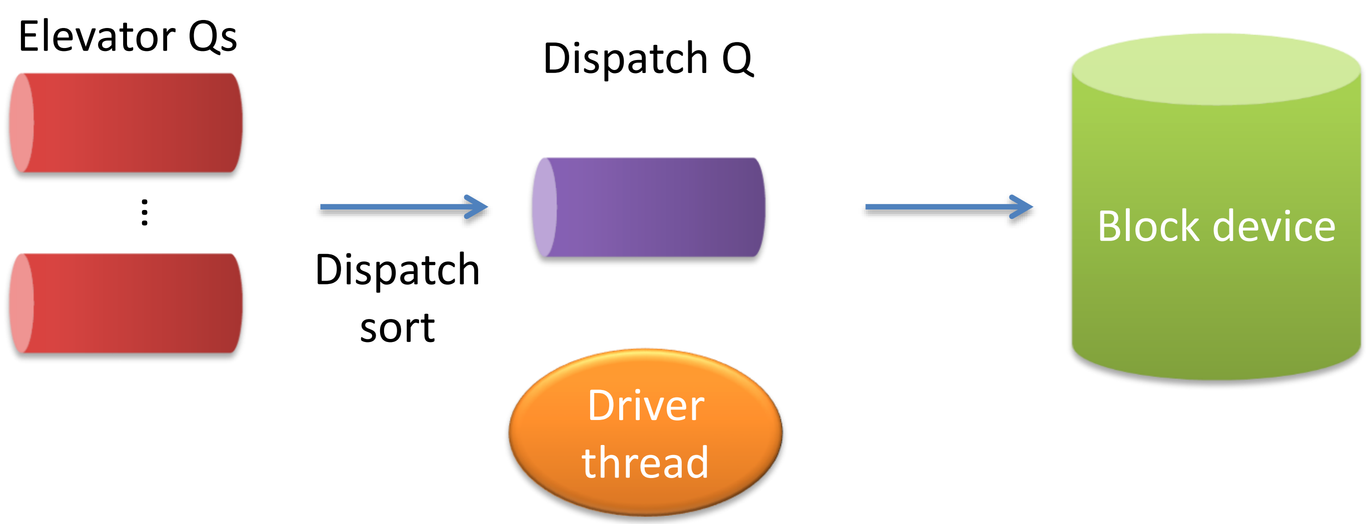
- The page cache flusher thread inserts the buffer heads of dirty pages to the scheduler queue

- < 2.6.32
  - A pool of flusher threads
  - A thread submits the BHs of 1024 dirty pages to the scheduler Q in a regular period or when the # of dirty pages > a threadhold

- >= 2.6.32
  - Every block device has a flusher thread
  - Behave like the threads in old kernel versions

Request Departures

- The block driver thread removes a request from the dispatch queue for final processing
  - If the dispatch Q is empty, the thread calls *blk_fetch_request*() and the scheduler will dispatch a request (from the scheduler Q) to the dispatch Q



NOOP Scheduler

- block/noop-iosched.c
- The elevator internal queue is FIFO
- Designed for random-access block devices, such as USB sticks and perhaps solid-state disks
  - Seek-time minimization has no effects on such devices
  - Ramdisk does not use NOOP, it registers its own *make_request*()

NOOP Private Queue & Device Queue

```
request_queue {
        struct list_head *queue_head;   // device Q (dispatch Q)
        elevator_queue *elevator;       // ptr to elv (not Q)
        …
}
struct elevator_queue {
        …
        void *elevator_data;            // elevator private data
}
struct noop_data {
        struct list_head queue;         // NOOP private Q
}
```

Notice: there are only one NOOP private Q and many device Qs

Initialization (NOOP)

```c
static struct elevator_type elevator_noop = {
	.ops = {
		.elevator_merge_req_fn		= noop_merged_requests,
		.elevator_dispatch_fn		= noop_dispatch,
		.elevator_add_req_fn		= noop_add_request,
		.elevator_queue_empty_fn	= noop_queue_empty,
		.elevator_former_req_fn		= noop_former_request,
		.elevator_latter_req_fn		= noop_latter_request,
		.elevator_init_fn		= noop_init_queue,
		.elevator_exit_fn		= noop_exit_queue,
	},
	.elevator_name = "noop",
	.elevator_owner = THIS_MODULE,
};

static int __init noop_init(void)
{
	elv_register(&elevator_noop);

	return 0;
}

static void __exit noop_exit(void)
{
	elv_unregister(&elevator_noop);
}
```

Request Arrivals (NOOP)

- Upper layers submit a bio to low-level driver

ll_rw_block() *block layer
__generic_make_request()
q →make_request_fn() *req Q callback
__make_request() *generic mkreq
add_request()
__elv_add_request() *elevator layer
q →elevator →ops →add_request_fn() *elv callback
noop_add_request() * NOOP

NOOP Add Request

```
static void noop_add_request(struct request_queue *q, struct request *rq)
{
	struct noop_data *nd = q->elevator->elevator_data;

	list_add_tail(&rq->queuelist, &nd->queue);
}
```

- Add the new request the tail of the NOOP private queue
  - Requests will later be removed from the list head

Request Departures (NOOP)

- The kernel thread associated with a block driver calls *blk_fetch_request*() at proper timings

  blk_fetch_request()
  blk_peek_request()
  __elv_next_request()
  q→*elevator*→*ops*→*elevator_dispatch_fn*()
  noop_dispatch()

- If a bio is not serviced immediately, then when the bio is completed the block driver calls *__blk_end_request*()

NOOP Dispatch Request

```
static int noop_dispatch(struct request_queue *q, int force)
{
	struct noop_data *nd = q->elevator->elevator_data;

	if (!list_empty(&nd->queue)) {
		struct request *rq;
		rq = list_entry(nd->queue.next, struct request, queuelist);
		list_del_init(&rq->queuelist);
		elv_dispatch_sort(q, rq);
		return 1;
	}
	return 0;
}
```

- Removing a pending bio from the head of the NOOP private queue
- Inserting the removed bio to the dispatch Q by *elv_dispatch_sort*()

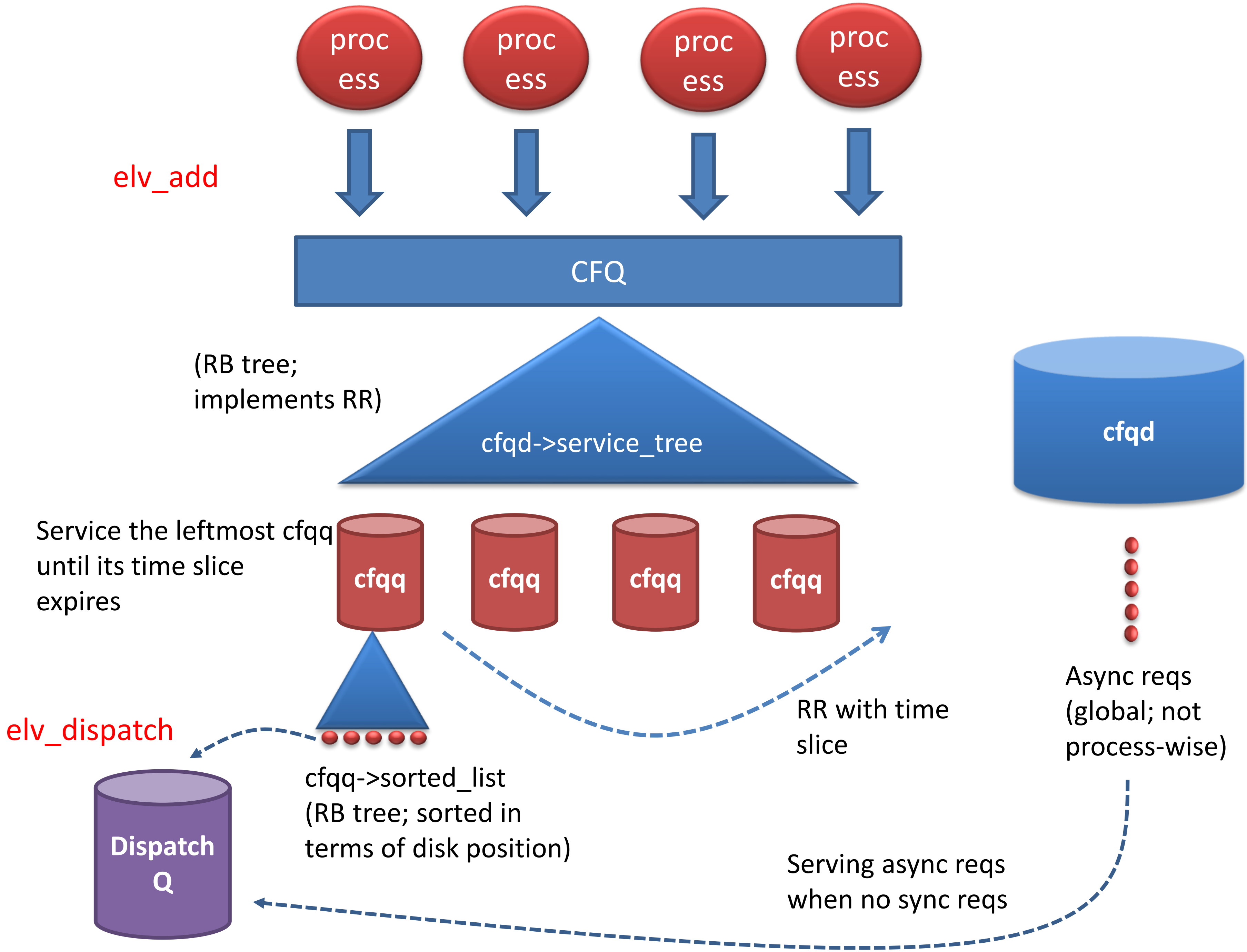
```c
void elv_dispatch_sort(struct request_queue *q, struct request *rq)
{
	sector_t boundary;
	struct list_head *entry;
	int stop_flags;

	...

	boundary = q->end_sector;
	stop_flags = REQ_SOFTBARRIER | REQ_HARDBARRIER | REQ_STARTED;
	list_for_each_prev(entry, &q->queue_head) {
		struct request *pos = list_entry_rq(entry);

		if (blk_discard_rq(rq) != blk_discard_rq(pos))
			break;
		if (rq_data_dir(rq) != rq_data_dir(pos))
			break;
		if (pos->cmd_flags & stop_flags)
			break;
		if (blk_rq_pos(rq) >= boundary) {
			if (blk_rq_pos(pos) < boundary)
				continue;
		} else {
			if (blk_rq_pos(pos) >= boundary)
				break;
		}
		if (blk_rq_pos(rq) >= blk_rq_pos(pos))
			break;
	}

	list_add(&rq->queuelist, entry);
} ? end elv_dispatch_sort ?
```

Break when seeing barrier

Sort by disk pos

CFQ Scheduler

- Complete Fair Queuing
  - block/cfq-iosched.c
  - Its scheduling algorithm changed a lot in the recent kernel versions
- Every process receives a fair share of disk utilization
  - Preventing asynchronous requests of a process from overwhelming a block device
  - Improves fairness at the cost of access latency


proc
ess
proc
ess
proc
ess
proc
ess
elv_add
CFQ
(RB tree;
implements RR)
cfqd->service_tree
cfqd
Service the leftmost cfqq
until its time slice
expires
cfqq
cfqq
cfqq
cfqq
elv_dispatch
RR with time
slice
Async reqs
(global; not
process-wise)
cfqq->sorted_list
(RB tree; sorted in
terms of disk position)
Dispatch
Q
Serving async reqs
when no sync reqs

CFQ Process Queues

```
request_queue {
        …
        elevator_queue   *elevator;          // ptr to elv (not Q)
        …
}
struct elevator_queue {
        …
        void *elevator_data;                 // cfq_data *cfqd (only 1)
}
                   @service_tree
struct request{
        …
        void  *elevator_private2;            // cfq_queue *cfqq (per process)
}
```

Request Arrivals (CFQ)

- Insert the incoming request into various queues
  - *cfqq→fifo*
    - FIFO list. Timed-out requests is serviced first
  - *cfqq→sorted_list*
    - RB tree. Sorted in terms of disk positions
  - *cfqd→prio_trees*[8]
    - Arrays of RB trees. Defines 8 level of request priority; inherit from process priority class (RT, BE, IDLE)
    - For asynchronous requests, shared by all processes

```
static void cfq_insert_request(struct request_queue *q, struct request *rq)
{
        struct cfq_data *cfqd = q->elevator->elevator_data;
        struct cfq_queue *cfqq = RQ_CFQQ(rq);

        cfq_log_cfqq(cfqd, cfqq, "insert_request");
(1) →   cfq_init_prio_data(cfqq, RQ_CIC(rq)->ioc);

(2) →   cfq_add_rq_rb(rq);

        rq_set_fifo_time(rq, jiffies + cfqd->cfq_fifo_expire[rq_is_sync(rq)]);
(3) →   list_add_tail(&rq->queuelist, &cfqq->fifo);

        cfq_rq_enqueued(cfqd, cfqq, rq);
}
```

1. Initialize request priority information (mostly inherit from process's CPU priority class) and insert the request into *cfqd*→*prio_trees*[8]
2. Insert the request to *cfqq*→*sorted_list*
3. Insert the request to *cdqq*→*fifo*

Request Departures (CFQ)

- If the time slice of the current *cfqq* does not yet expire
  - If a request in *cfqq*→*fifo* has timed out, choose it
  - Otherwise, choose a request from *cfqq*→*sorted_list* in the SCAN manner
- Otherwise, choose the next *cfqq*
- Calls *elv_dispatch_sort*() to insert the chosen request to the device queue

```
static int cfq_dispatch_requests(struct request_queue *q, int force)
{
        struct cfq_data *cfqd = q->elevator->elevator_data;
        struct cfq_queue *cfqq;

        ...

(1) →   cfqq = cfq_select_queue(cfqd);
        if (! cfqq)
                return 0;

        /*
         * Dispatch a request from this cfqq, if it is allowed
         */
(2) →   if (! cfq_dispatch_request(cfqd, cfqq))
                return 0;

        ...

} ? end cfq_dispatch_requests ?
```

cfqq: per-process Q

1. If the current (the leftmost in the service tree) *cfqq* used up its time slice, replenish its slice and re-insert to the service tree and pick up the next cfqq from the service tree
2. Calls *cfq_find_next_rq*() to select the next request in the SCAN manner, and then calls *elv_dispatch_sort*() to insert the request to the device queue.

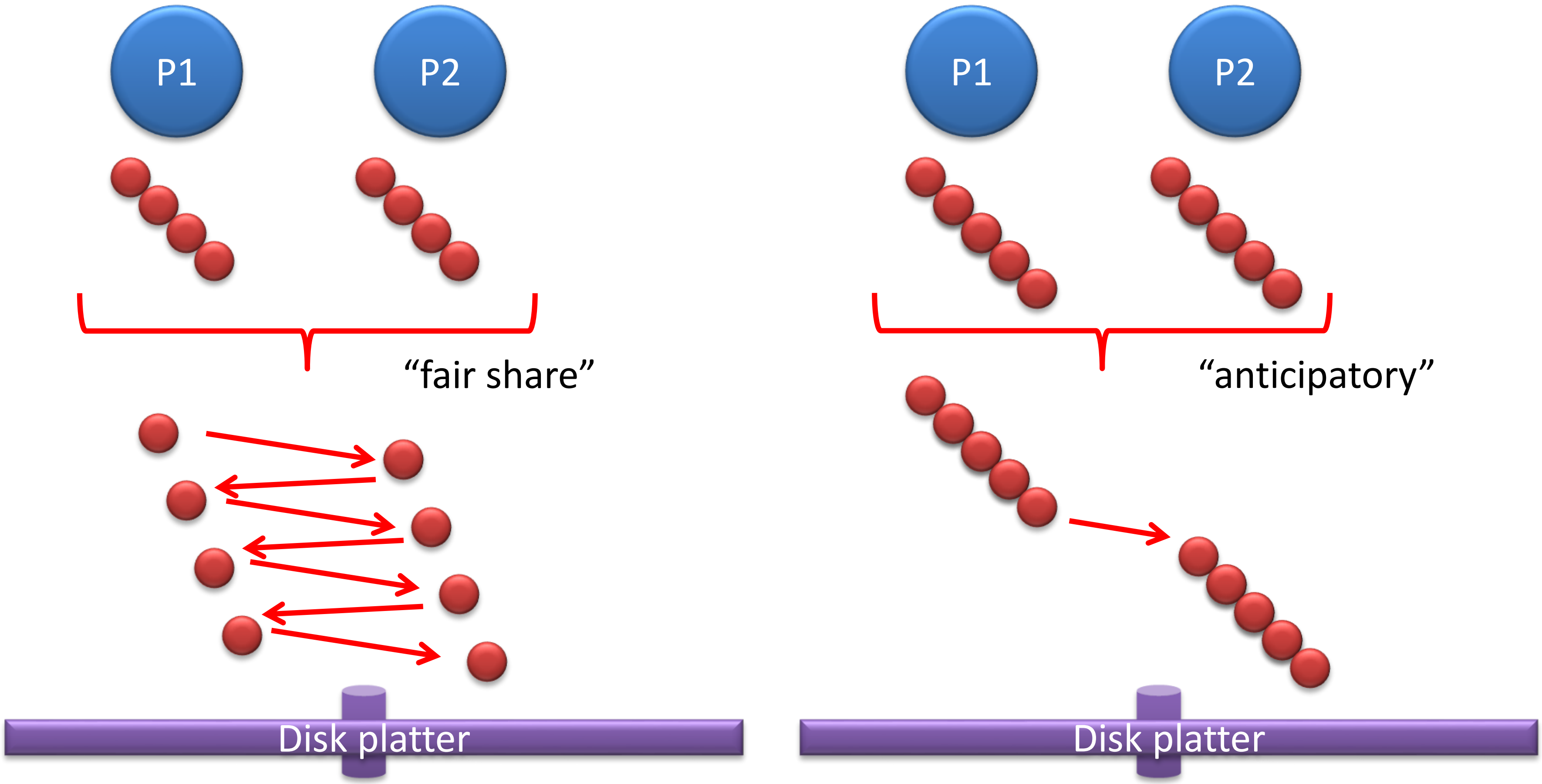
AS Scheduler

- Anticipatory
- Non work-conserving
  - Work-conserving schedulers dispatch as long as its queue is not empty
    - NOOP, deadline, and early versions of CFQ
  - Non work-conserving schedulers may pause even if the queue is backlogged
    - Idle time insertion may improve the performance!
      - !?

AS Scheduler

- The major problem is on read processing, because processes issue one read request, wait its completion, and then issue the next
  - The scheduler does not know whether a process will issue a next read request!
  - In contrast, a process issues a bunch of write requests that can be processed asynchronously
  - Efficient read processing requires *prediction*!

Why Idling Helps



Locality of access: If a process access a disk block, then it is very possible that

- It accesses the next disk block (temporal locality)
- It accesses a nearby disk block (spatial locality)

The Need for Idleness

- Most read requests are synchronous requests
  - The scheduler sees only one read request in the queue, but further read may come
  - Make a “guess”; wins if the process issues a nearby request, but loses otherwise
  - How long should we wait??
    - Long wait: penalty will be large
    - Short wait: pessimistic

The Need for Idleness

- The disk seek time is a function of the seek distance

$$a_1+a_2(d)^{0.5}$$

  - a_1 and a_2 are HDD-specific constants
  - Non-conserving scheduling receives benefit if

$$a_1+a_2(d_1)^{0.5}+t< a_1+a_2(d_2)^{0.5}$$

$$t<(d_2/d_1)^{0.5}$$

    - If $d_2/d_1 = 16$ then AS should not wait longer than 4ms
  - The AS scheduler has a simpler choice: ~6ms
    - Average seek time
      - Notebook HDD: 14 ms (ST9500325AS)
      - Desktop HDD: 10 ms (ST3000DM001)

Request Handling (AS)

- AS maintains three queues (inherit from deadline)
  - Read queue, write queue, and FIFO queue
  - RQ and WQ sort their requests in terms of sector #
  - FIFO Q evicts timed-out requests to avoid starvation
- After AS dispatches a request from the RQ to the devQ, it pauses for a while
  - If a new request arrives nearby the current head position, service it immediately
  - If the idleness expires, resume processing pending requests

Recent Development of AS and CFQ

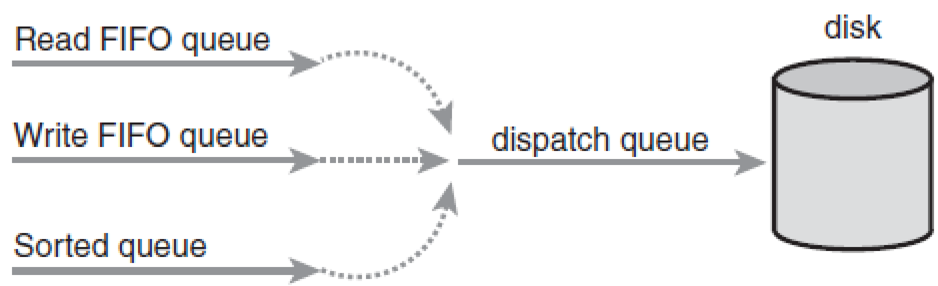
- AS spins off from deadline, while CFQ is independently developed
- AS outperformed CFQ in many tests
  - Prior kernel releases use AS as the default
- Newer version of CFQ implements anticipation
  - Each cfqq is associated with a time slice, and idling consumes the time slice
    - A nature implementation of anticipatory
  - Recent kernel releases use CFQ as the default scheduler
    - Hmmm… competition is a good thing

Recent Development of AS and CFQ

- Challenges
  - RAIDs or LVM-volumes have multiple LUNs behind the disk drive
    - LUNs are transparent to schedulers (a bad thing)
    - Many RAID drivers calls make_request() to bypass the scheduler and schedule requests itself
  - Disk utilization accounting becomes difficult when the hard drives support NCQ or tagged commands, especially for SSDs
    - Allowing multiple outstanding requests
    - Requests are completed out of order

Deadline Scheduler

- It is basically a SCAN scheduler with the following enhancements
  – Starvation prevention
  – Read latency optimization
- It adopts 3 queues, one sorted in terms of disk position and the other two are FIFOs



Deadline Scheduler

- A new request is inserted to the sorted queue and the RQ or WQ (depending on its type)
  - The deadline (largest waiting time) of a read request and a write request are 500ms and 5000ms, respectively
- On dispatch, check deadline expiration in RQ and WQ
  - The scheduler prefers RQ over WQ unless WQ has been ignored too many times
  - If RQ and WQ have no expiration, pull a request from the sorted queue

Why Prefer Read over Write?

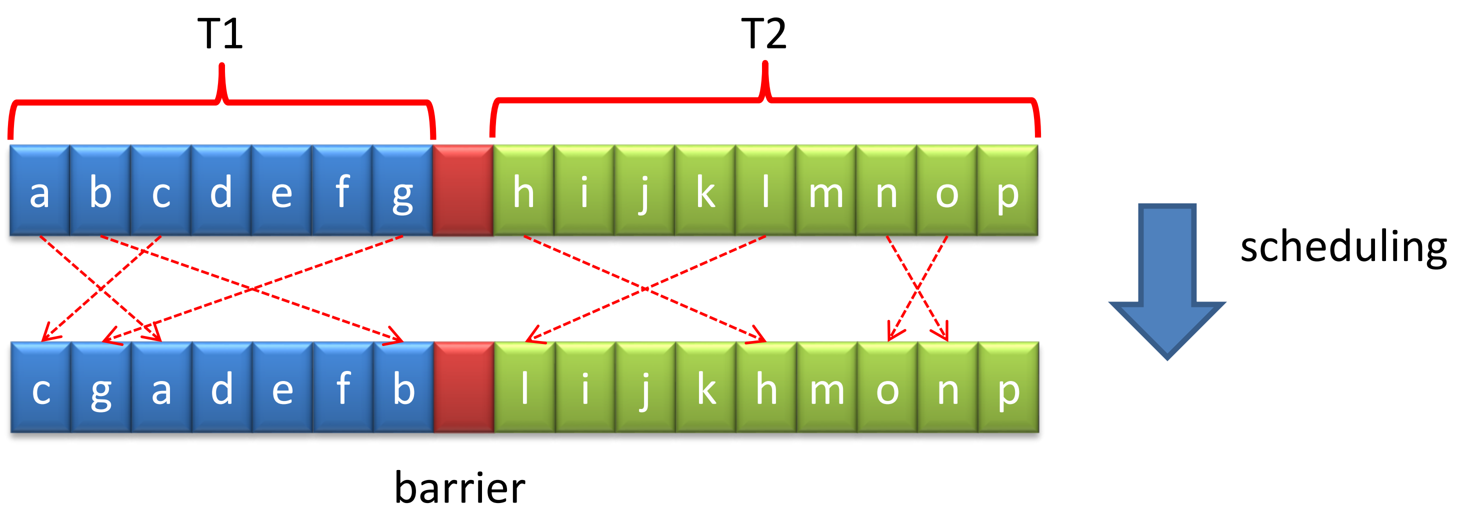
- As mentioned earlier, a process is blocked until its read request is fulfilled (i.e., sync)
  - Delaying a read request further delays the next read request from the same process
  - Read latency has higher impact to user experience
  - Write latency is de-coupled from user processes as a write request is "completed" as soon as it enters the page cache

Request Merge

- If two requests are adjacent in terms of disk location, merge them into one
  – Saves I/O overhead and exploits spatial locality
- Front merge and back merge
  – 99.999% back merge
- The block layer calls .elevator_merge_fn after it has merged two requests
  – The elevator can delete the merged request

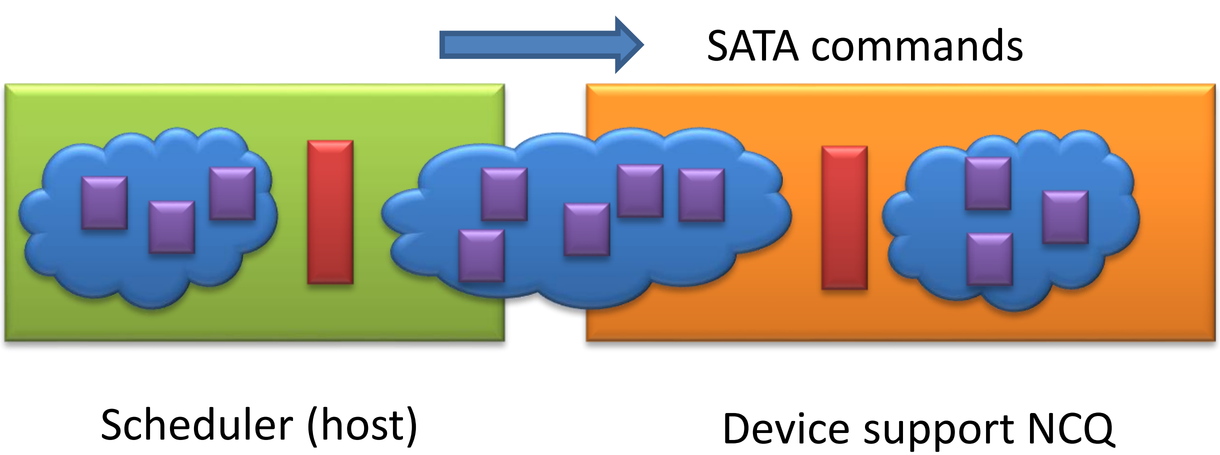
I/O Barrier

- Journaling file system must make sure that the current transaction has been "committed" to disk before proceeding to the next transaction
  - The order of requests of the same transaction does not matter, but the re-ordering must not come across the boundaries of transactions
  - The file system issues a "barrier" request to confine the scope of request scheduling



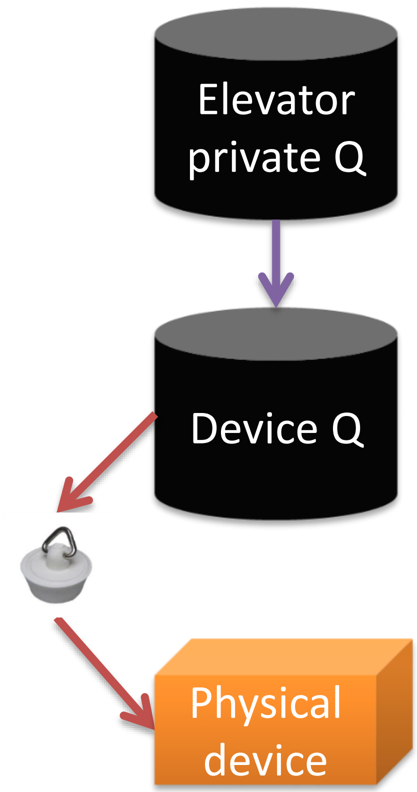
I/O Barrier

- Barrier becomes more complex when the block device supports NCQ or tagged commands
  - The device will internally re-order the requests!!
  - Send a “barrier” command if the device supports
  - Otherwise, the scheduler stops dispatching requests until all precedent requests complete



Plug/Unplug

- Keep as many pending requests in the scheduler's private queue as possible, and dispatch requests to the device queue in batches
  - Increase the chance of request merging and re-ordering
  - Also a kind of anticipation
- blk_plug_device() and blk_remove_plug(), they must be paired with each other



Plug a Block Device

- A block device is plugged when there is no request in scheduler queue
  - Sets QUEUE_FLAG_PLUGGED in q→queue_flags using test-and-set
  - Reset the un-plug timer

```
void blk_plug_device(struct request_queue *q)
{
	WARN_ON(!irqs_disabled());

	/*
	 * don't plug a stopped queue, it must be paired with blk_start_queue()
	 * which will restart the queueing
	 */
	if (blk_queue_stopped(q))
		return;

	if (!queue_flag_test_and_set(QUEUE_FLAG_PLUGGED, q)) {
		mod_timer(&q->unplug_timer, jiffies + q->unplug_delay);
		trace_block_plug(q);
	}
}
```

Unplug a Block Device

- A block device is unplugged when
  - There are pending requests in the scheduler's private queue (default is 4)
  - The plug timer has expired (default is 3 ms)
    - Wakes up kblockd, who in turn calls blk_remove_plug()

```
int blk_remove_plug(struct request_queue *q)
{
	WARN_ON(!irqs_disabled());

	if (!queue_flag_test_and_clear(QUEUE_FLAG_PLUGGED, q))
		return 0;

	del_timer(&q->unplug_timer);
	return 1;
}
```

Request Queue, re-visited

```
struct request_queue
{
    /*
     * Together with queue_head for cacheline sharing
     */
    struct list_head        queue_head;
    struct request          *last_merge;
    struct elevator_queue   *elevator;

    ...

    make_request_fn         *make_request_fn;
    unplug_fn               *unplug_fn;
    merge_bvec_fn           *merge_bvec_fn;

    struct timer_list   unplug_timer;
    int                 unplug_thresh;   /* After this many requests */
    unsigned long       unplug_delay;    /* After this many jiffies */
    struct work_struct  unplug_work;

    ....
} ? end request_queue ? ;
```

Called by kblockd

Wakes up kblockd

blktrace

- Block trace: a block-layer tracing tool
  - A part of the kernel release; default is on
  - Need a user program "blktrace" to turn on/off tracing and another program "blkparse" to convert the trace to human readable format
  - Captures events for I/O scheduler and block devices

```
struct cfq_data *cfqd = q->elevator->elevator_data;
struct cfq_queue *cfqq = RQ_CFQQ(rq);

cfq_log_cfqq(cfqd, cfqq, "insert_request");
```

```
cfq_log_cfqq(cfqd, cfqq, "dispatched a request");
return 1;
```

Sample output

% blktrace -d /dev/sda -o - | blkparse -i –

```
8,0   0   1295   34.301179551  2898  A  WS 2090487 + 8 <- (8,1) 2090424
8,0   0   1296   34.301181856  2898  Q  WS 2090487 + 8 [firefox]
8,0   0   1297   34.301187117  2898  G  WS 2090487 + 8 [firefox]
8,0   0   1298   34.301191156  2898  P   N [firefox]
8,0   0   1299   34.301193470  2898  I   W 2090487 + 8 [firefox]
8,0   0   1300   34.301214331  2898  U   N [firefox] 1
8,0   0   1301   34.301234469  2898  D   W 2090487 + 8 [firefox]
8,0   0   1302   34.360634963     0  C   W 2090487 + 8 [0]
```

process

Sequence Number

CPU ID

timestamp

PID

event

RWBS

Start block + number of blocks

Dev(mjr,mnr)

Sample output

- Explain for event

Event	description
A	(Remap) The remap action details what exactly is being remapped to what. *remapping address of a logical partition to a physical disk.
Q	(Queued) This notes intent to queue at the given location. *going to insert a request to the scheduler queue
G	(Get request) To send any type of request to a block device, a struct request container must be allocated first. *get a request for inserting into the scheduler queue
P	(Plug) When I/O is queued to a previously empty block device queue, Linux will plug the queue in anticipation of future I/O being added before this data is needed. *plug a device because the scheduler Q is empty

Sample output

I	(Inserted) A request is being sent to the I/O scheduler for addition to the internal queue and later service by the driver. The request is fully formed at this time. *A request has been inserted to the scheduler Q
U	(Unplug) Some request data already queued in the device, start sending requests to the driver. This may happen automatically if a timeout period has passed or if a number of requests have been added to the queue. *scheduler Q is not empty so unplug the device Q
D	(Dispatch) A request that previously resided on the block layer queue or in the I/O scheduler has been sent to the driver. *A request has been dispatched to the block driver for processing
C	(Complete) A previously issued request has been completed. The output will detail the sector and size of that request, as well as the success or failure of it. *A request has been completed

Usage

- blktrace -d /dev/sda -o - | blkparse -i -
  - The same as “btrace /dev/sda”
- blktrace -d /dev/sda -o sda
- blkparse -i sda -s -o output.txt

Lab 9: SSTF Disk Scheduler

SSTF

- Shortest Seek Time First
  - Service the request whose disk position is nearest to the current head position
  - Has natural appeal, but is sub-optimal in terms of global throughput
    - Starvation
    - Total seek distance is not guaranteed to be minimal

Implementation

- We recommend that you modify the NOOP scheduler to implement SSTF
  - Modify noop_add_request()
    - Re-use q->elevator->elevator_data->queue; but now employ a new insertion policy
    - Insert the new request to the nearest existing request
  - Modify elv_dispatch_sort(), which is called by noop_dispatch()
    - Its original behavior is to sort requests in the dispatch queue in terms of disk position
    - A cleaner way is write a new function to replace elv_dispatch_sort()

Validate Your Implementation

- Create a new virtual disk, e.g., sdb
  - Create a partition sdb1, and format it (optional)
- Write a program whose disk access pattern sufficiently validates your SSTF
  - We will provide you a tiny kernel module that directly write to logical sectors of /dev/sdb1
    - Writing to /dev/sdb is fine too, whatever you like it
  - Notice: be careful about request merge

Validate Your Implementation

- Switch your write position back and forth between two clusters and observe the order of request add/dispatch
- E.g., request add order a1, b1, a2, b2, ……
  - NOOP dispatch: a1, b2, a2, b2
  - SSTF dispatch: a1, a2, …, b1, b2, …

References

- Robert love, “Linux Kernel Development 3rd Edition,” 2010
- Daniel B. Bovet et al., “Understanding the Linux Kernel 3rd Edition,” 2005
- Jonathan Corbet et al., “Linux Device Drivers 3rd Edition,” 2005
  - Ch16, http://lwn.net/Kernel/LDD3/
- Linux kernel source tree 2.6.34