Linux I/O Schedulers

Hao-Ran Liu

Why I/O scheduler?

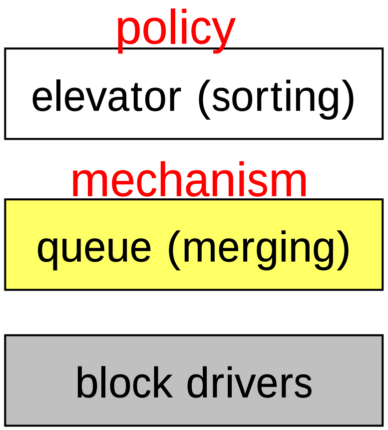
- Disk seek is the slowest operation in a computer
  - A system would perform horribly without a suitable I/O scheduler
- I/O scheduler arranges the disk head to move in a single direction to minimize seeks
  - Like the way elevators moves between floors
  - Achieve greater global throughput at the expense of fairness to some requests

What do I/O schedulers do?

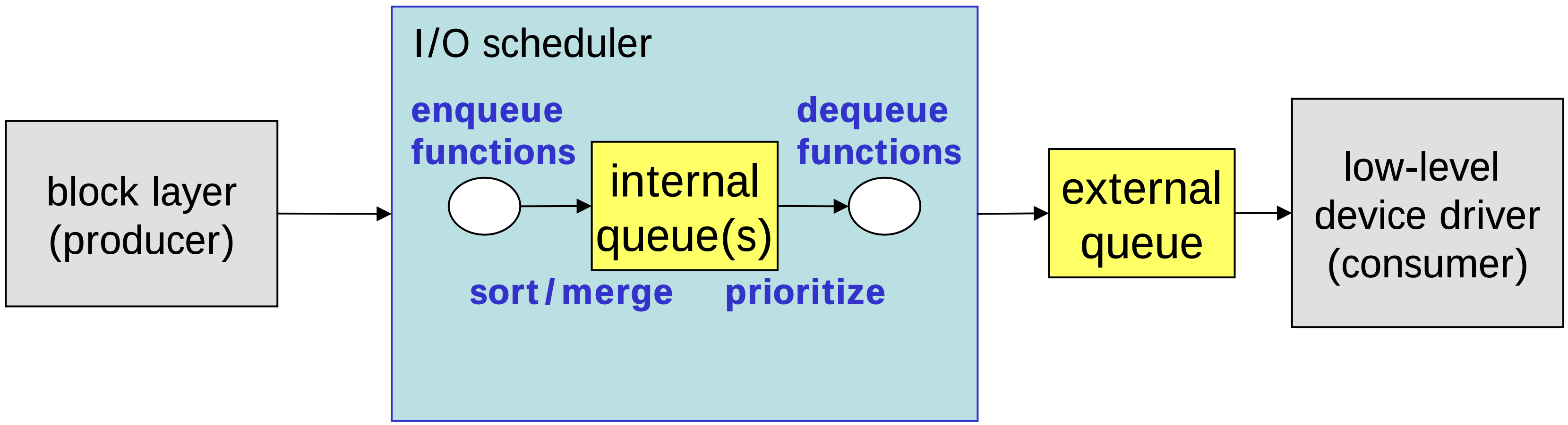
- Improve overall disk throughput by
  - Reorder requests to reduce the disk seek time
  - Merge requests to reduce the number of requests
- Prevent starvation
  - submit requests before deadline
  - Avoid read starvation by write
- Provide fairness among different processes

Linux I/O scheduling framework

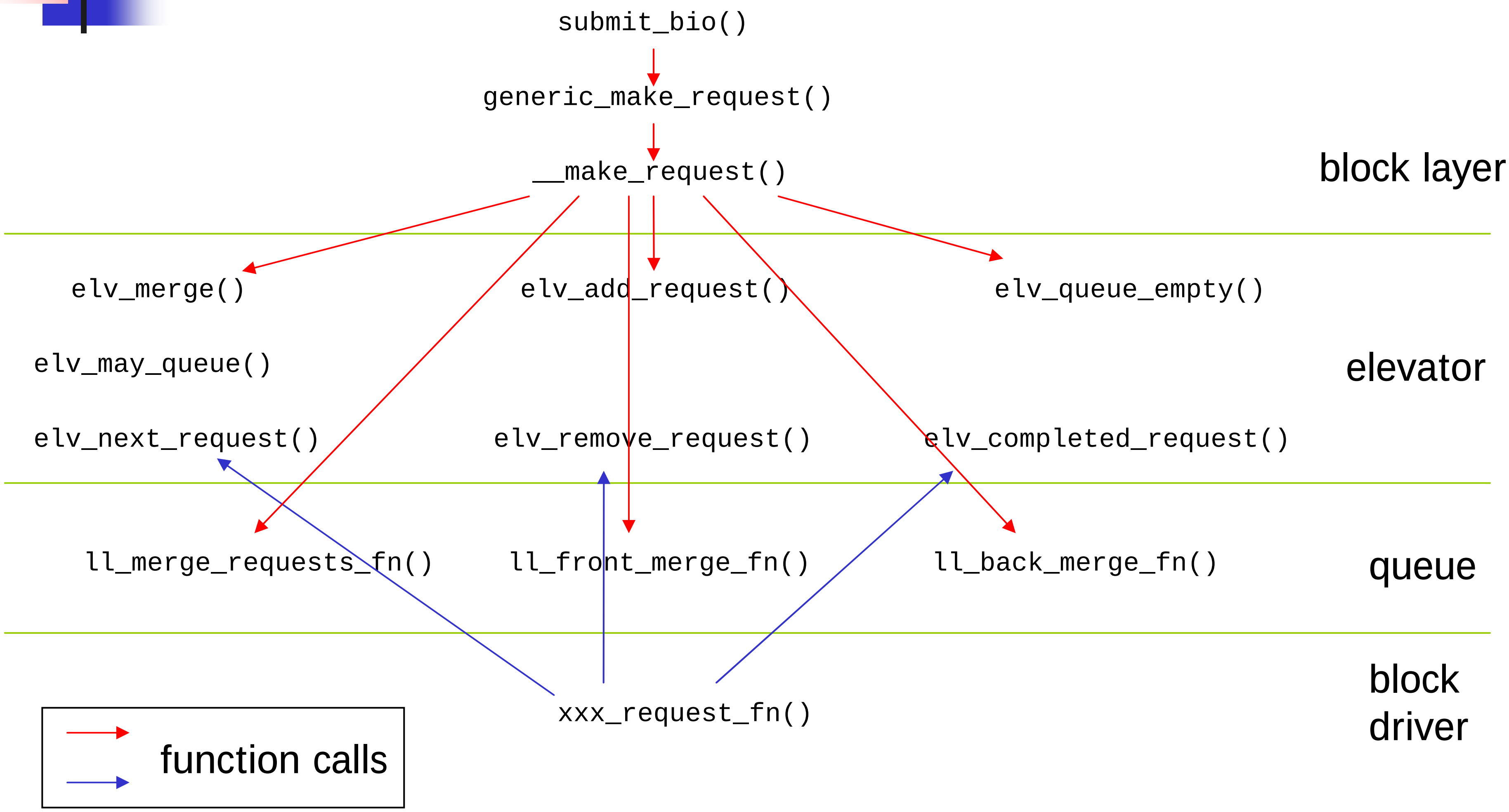
- Linux elevator is an abstract layer to which different I/O scheduler can attach
- Merging mechanisms are provided by request queues
  - Front or back merge of a request and a bio
  - Merge two requests
- Sorting policy and merge decision are done in elevators
  - Pick up a request to be merged with a bio
  - Add a new request to the request queue
  - Select next request to be processed by block drivers



Abstraction of Linux I/O scheduler framework



The relationship of I/O scheduler functions

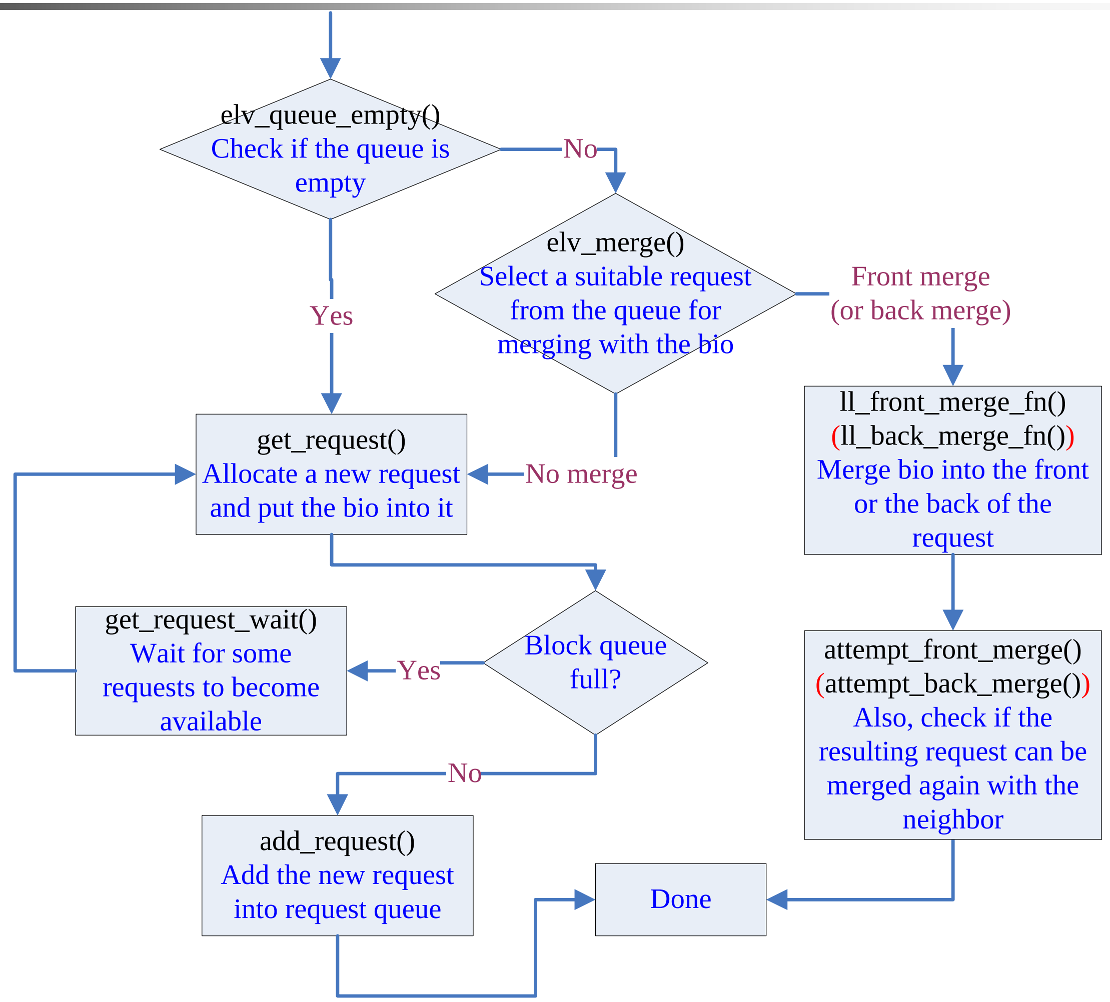


Description of elevator functions

Most functions are just wrappers. The actual implementation are elevator-specific

Type	Description
`elv_merge`	Find a request in the request queue to be merged with a bio. The function's return value indicate front merge, back merge or no merge.
`elv_add_request`	Add a new request to the request queue
`elv_may_queue`	Ask if the elevator allows enqueing of a new request
`elv_remove_request`	Remove a request from the request queue
`elv_queue_empty`	Check if the request queue is empty
`elv_next_request`	Called by the device drivers to get next request from the request queue
`elv_completed_request`	Called when a request is completed
`elv_set_request`	When a new request is allocated, this function is called to initialize elevator-specific variables
`elv_put_request`	When a request is to be freed, this function is called to free memory allocated for some elevator.

Flowchart of `__make_request()`



Merge functions at request queue

```
struct request_queue
{
        struct list_head queue_head;
        struct elevator_queue *elevator;
        ...
        merge_request_fn *back_merge_fn;
        merge_request_fn *front_merge_fn;
        merge_requests_fn *merge_requests_fn;
        ...
}
```

A list of requests (external queue)

Elevator queue of this request queue

Pointers to merge functions:

```
ll_back_merge_fn()     : back merge a request and a bio
ll_front_merge_fn()    : front merge a request and a bio
ll_merge_requests_fn() : merge two requests
```

`ll_xxx_fn()` is the default set of functions for merge

The structure of elevator type

Each request queue is associated with its own elevator queue of certain type

```
struct elevator_queue
{
        struct elevator_ops *ops;
        void *elevator_data;
        struct kobject kobj;
        struct elevator_type *elevator_type;
};
```

the private data of the elevator

```
struct elevator_type
{
        struct list_head list;
        struct elevator_ops ops;
        struct kobj_type *elevator_ktype;
        char elevator_name[ELV_NAME_MAX];
        struct module *elevator_owner;
};
```

A list of all available elevator types

elevator functions

the name of the elevator

The structure of elevator operations

These pointers point to the functions of a specific elevator

```
struct elevator_ops {
        elevator_merge_fn *elevator_merge_fn;
        elevator_merged_fn *elevator_merged_fn;
        elevator_merge_req_fn *elevator_merge_req_fn;
        elevator_next_req_fn *elevator_next_req_fn;
        elevator_add_req_fn *elevator_add_req_fn;
        elevator_remove_req_fn *elevator_remove_req_fn;
        elevator_requeue_req_fn *elevator_requeue_req_fn;
        elevator_deactivate_req_fn *elevator_deactivate_req_fn;
        elevator_queue_empty_fn *elevator_queue_empty_fn;
        elevator_completed_req_fn *elevator_completed_req_fn;
        elevator_request_list_fn *elevator_former_req_fn;
        elevator_request_list_fn *elevator_latter_req_fn;
        elevator_set_req_fn *elevator_set_req_fn;
        elevator_put_req_fn *elevator_put_req_fn;
        elevator_may_queue_fn *elevator_may_queue_fn;
        elevator_init_fn *elevator_init_fn;
        elevator_exit_fn *elevator_exit_fn;
};
```

Elevators in Linux 2.6

- All elevator types are registered in a global linked list `elv_list`
- Request queues can change to a different type of elevator online
  - This allows for adaptive I/O scheduling based on current workloads
- I/O schedulers available
  - noop, deadline, CFQ, anticipatory

NOOP I/O scheduler

- Suitable for truly random-access device, like flash memory card
- Requests in the queue are kept in FIFO order
- Only the last request added to the request queue will be tested for the possibility of a merge

NOOP: Registration

```
static struct elevator_type elevator_noop = {
        .ops = {
                .elevator_merge_fn          = elevator_noop_merge,
                .elevator_merge_req_fn      = elevator_noop_merge_requests,
                .elevator_next_req_fn       = elevator_noop_next_request,
                .elevator_add_req_fn        = elevator_noop_add_request,
        },
        .elevator_name = "noop",
        .elevator_owner = THIS_MODULE,
};

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

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

module_init(noop_init);
module_exit(noop_exit);
```

This structure stores the name of the noop elevator and pointers to **noop** functions. Use **elv_register()** function to register the structure with the plugin interfaces of the elevator

NOOP: add request and get next request

```
static void elevator_noop_add_request(request_queue_t *q, struct request *rq,
                                                int where) {
        if (where == ELEVATOR_INSERT_FRONT)
                list_add(&rq->queuelist, &q->queue_head);
        else
                list_add_tail(&rq->queuelist, &q->queue_head);

        /*
         * new merges must not precede this barrier
         */
        if (rq->flags & REQ_HARDBARRIER)
                q->last_merge = NULL;
        else if (!q->last_merge)
                q->last_merge = rq;
}

static struct request *elevator_noop_next_request(request_queue_t *q) {
        if (!list_empty(&q->queue_head))
                return list_entry_rq(q->queue_head.next);

        return NULL;
}
```

Add a new request to the request queue

Called by the device driver to get the next request to be submitted. If the request queue is not empty, return the request at the head of the queue

NOOP: request merge

```
/*
 * See if we can find a request that this buffer can be coalesced with.
 */
static int elevator_noop_merge(request_queue_t *q, struct request **req,
                                  struct bio *bio) {
        int ret;

        ret = elv_try_last_merge(q, bio);
        if (ret != ELEVATOR_NO_MERGE)
                *req = q->last_merge;

        return ret;
}

static void elevator_noop_merge_requests(request_queue_t *q,
                          struct request *req, struct request *next) {
        list_del_init(&next->queuelist);
}
```

Given a bio, find a adjacent request in the request queue to be merged with.

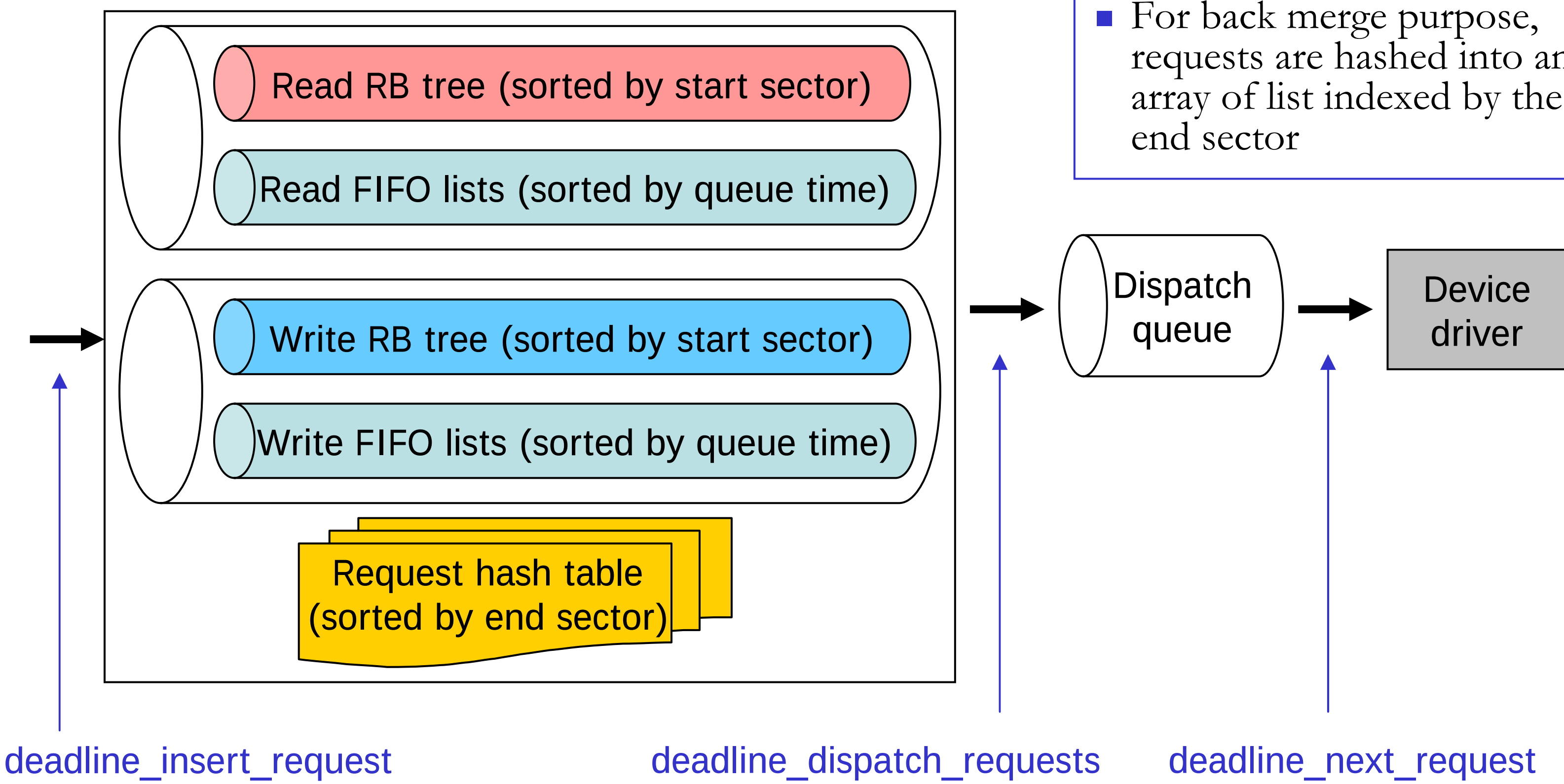
This function simply remove `next` request from the request queue. It is called after `next` are merged into `req`.

Deadline I/O scheduler

- Goal
  - Reorder requests to improve I/O performance while simultaneously ensuring that no I/O request is being starved
  - Favor reads over writes
- Each requests is associated with a expire time
  - Read: 500ms, write 5sec
- Requests are inserted into
  - A sorted-by-start-sector queue (two queues! for read and write)
  - A FIFO list (two lists too!) sorted by expire time
- Normally, requests are pulled from sorted queues. However, if the request at the head of either FIFO queue expires, requests are still processed in sorted order but started from the first request in the FIFO queue

Architecture view of Deadline I/O scheduler

- The sorted queues are built on red-black trees
- For back merge purpose, requests are hashed into an array of list indexed by the end sector



Deadline: dispatching requests

1. If [next_req] is in the batch (adjacent to previous request and batch count < 16), set it as [dispatch_req] and jump to step 5
2. Here, we are not in a batch. If there are read reqs and write is not starved, select read dir and jump to step 4
3. If there are write reqs, select write dir. Otherwise, return 0
4. If the first req in the fifo of the selected data direction expired, set it as [dispatch_req] and set batch count = 0. Otherwise, set [next_req] as [dispatch_req]
5. Increase batch count and dispatch the [dispatch_req].
6. Search forward from the end sector of [dispatch_req] in the RB tree of selected dir. Set the next request as [next_req]

Anticipatory scheduling
Background

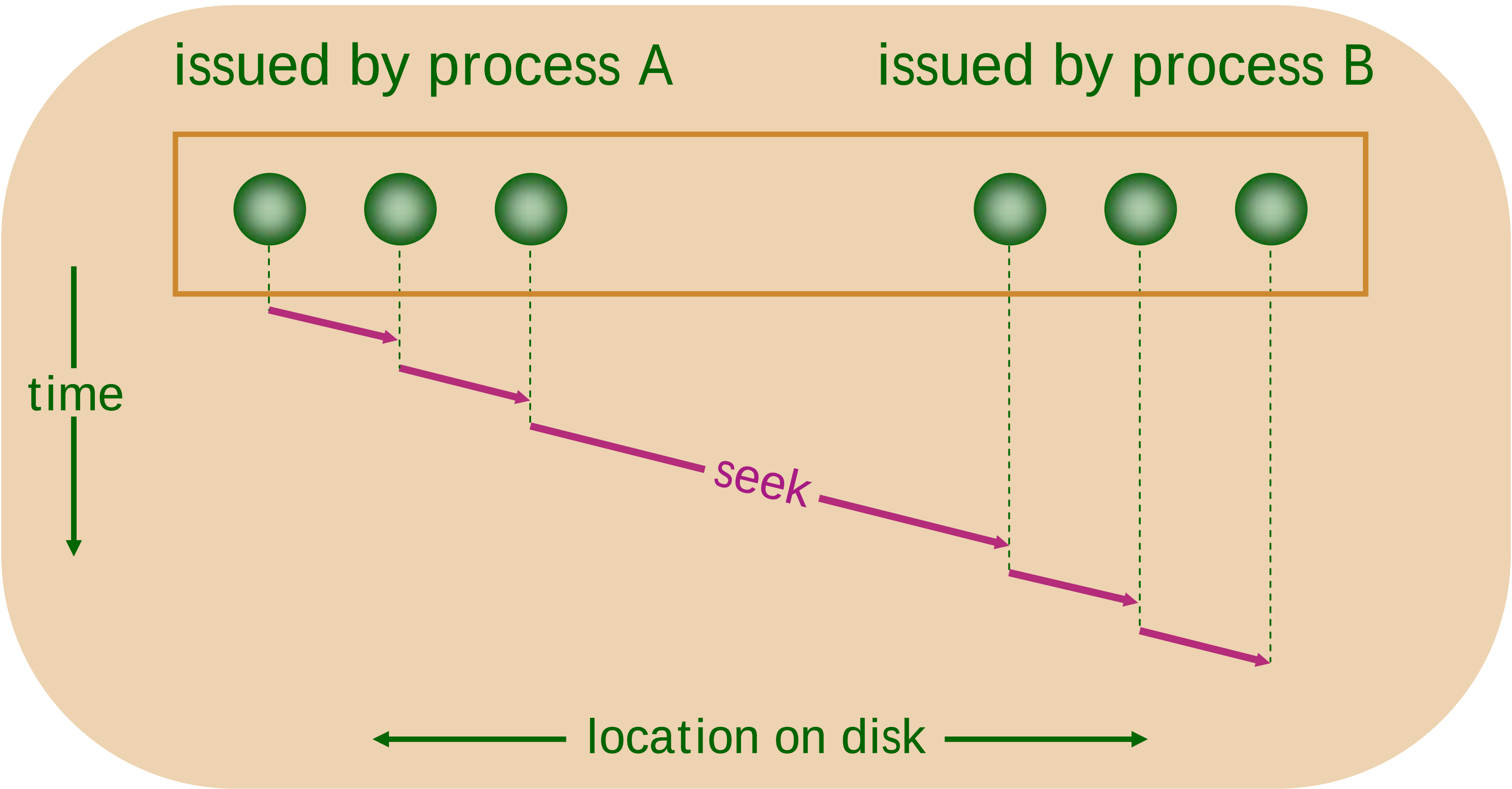
Disk schedulers reorder available disk requests for

- performance by seek optimization,
- proportional resource allocation, etc.

Any policy needs multiple outstanding requests to make good decisions!

from http://www.cs.rice.edu/~ssiyer/r/antsched/

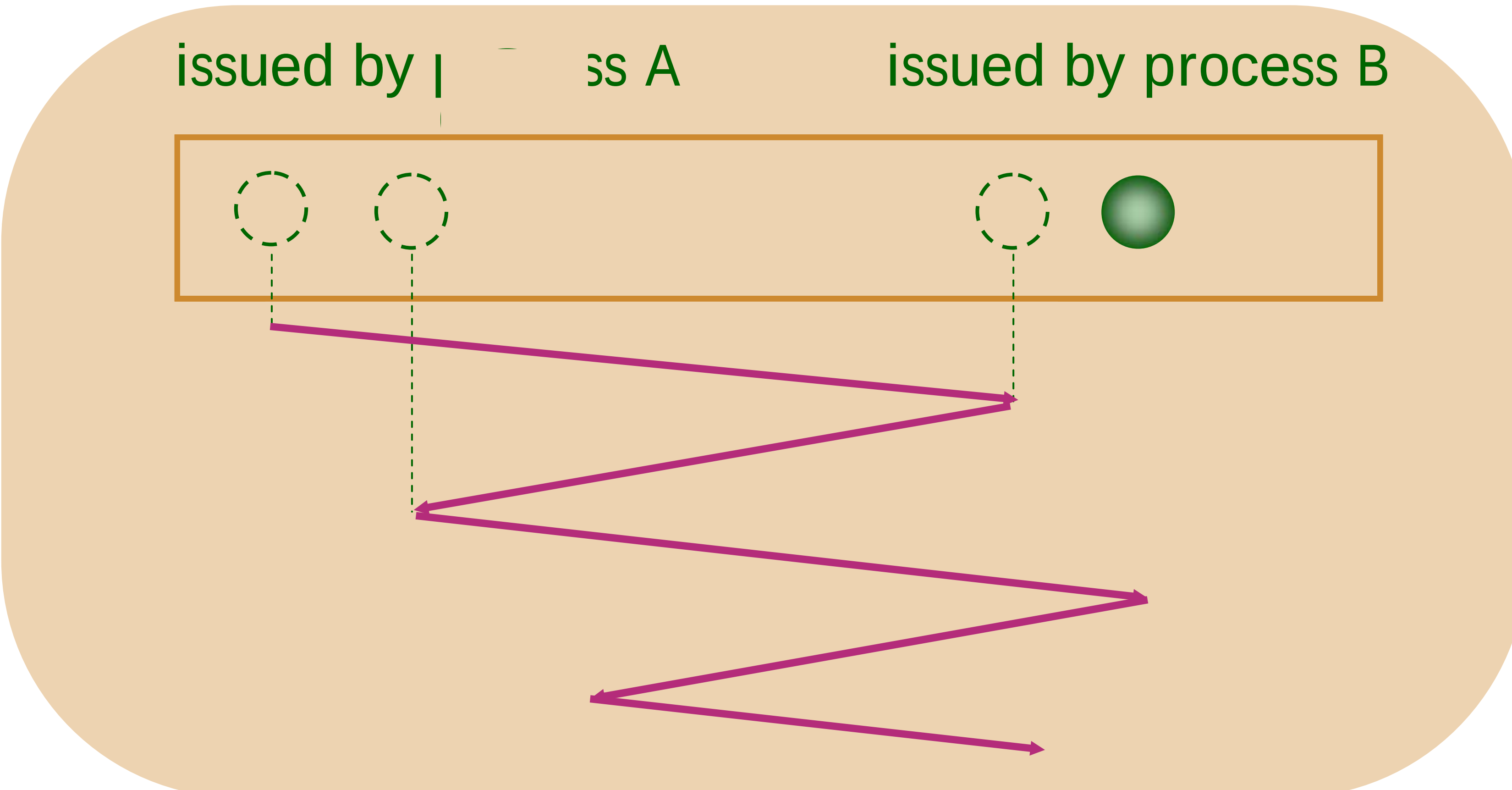
With enough requests...



E.g., Throughput = 21 MB/s (IBM Deskstar disk)

from http://www.cs.rice.edu/~ssiyer/r/antsched/

With synchronous I/O...



E.g., Throughput = 5 MB/s

from http://www.cs.rice.edu/~ssiyer/r/antsched/

Deceptive idleness

Process A is about to issue next request.

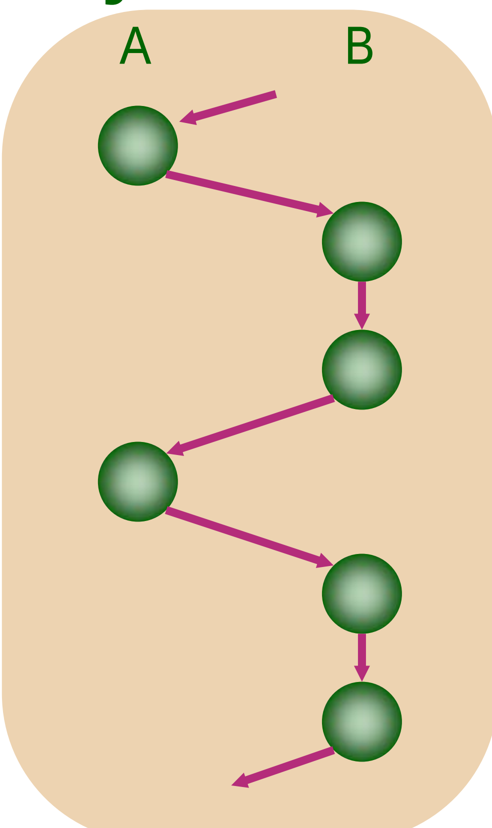
but

Scheduler hastily assumes that process A has no further requests!

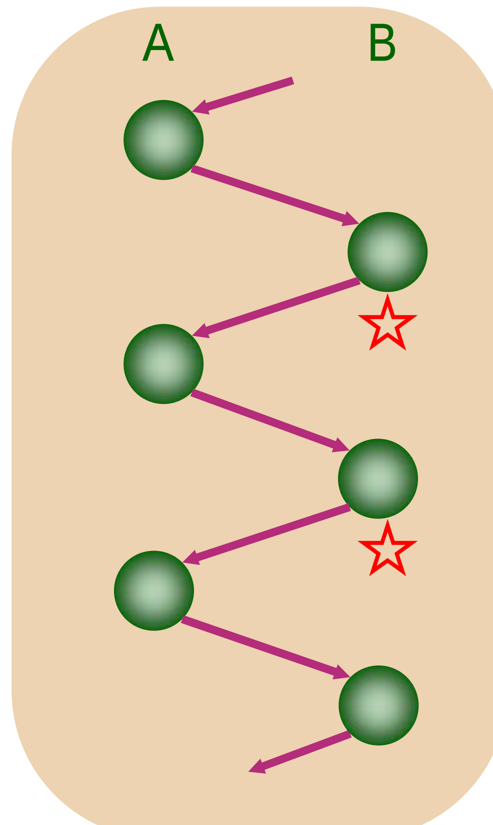
from http://www.cs.rice.edu/~ssiyer/r/antsched/

Proportional scheduler

Allocate disk service in say 1:2 ratio:



Deceptive idleness causes 1:1 allocation:



from http://www.cs.rice.edu/~ssiyer/r/antsched/

Anticipatory scheduling

Key idea: Sometimes wait for process whose request was last serviced.

Keeps disk idle for short intervals.

But with informed decisions, this:

- Improves throughput
- Achieves desired proportions

from http://www.cs.rice.edu/~ssiyer/r/antsched/

Cost-benefit analysis

Balance expected benefits of waiting
against cost of keeping disk idle.

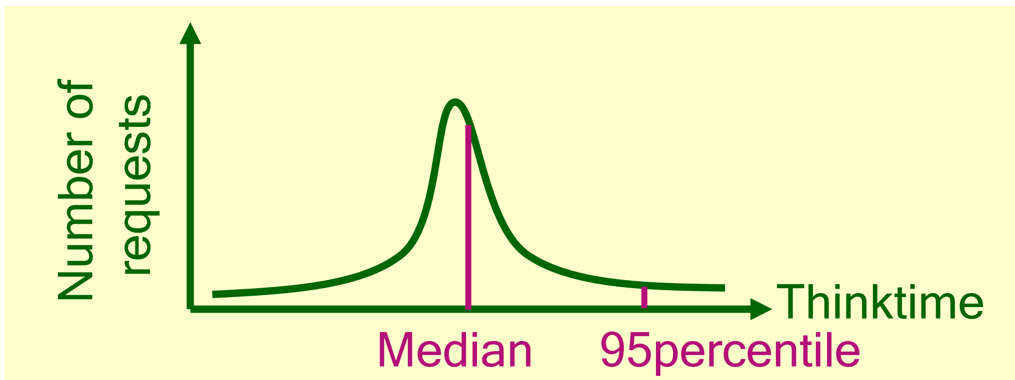
Tradeoffs sensitive to scheduling policy
e.g., 1. seek optimizing scheduler
2. proportional scheduler

from http://www.cs.rice.edu/~ssiyer/r/antsched/

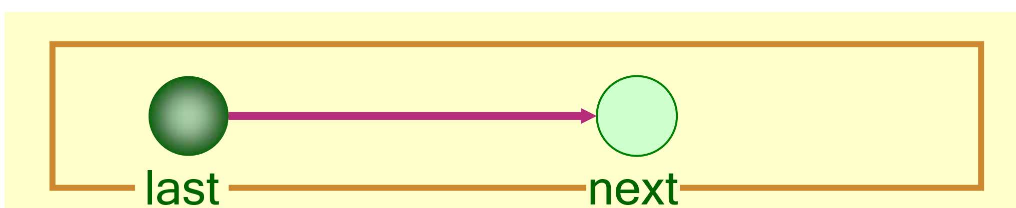
Statistics

For each process, measure:

1. Expected median and 95percentile thinktime



2. Expected positioning time



from http://www.cs.rice.edu/~ssiyer/r/antsched/

Cost-benefit analysis for seek optimizing scheduler

best := best available request chosen by scheduler

next := expected forthcoming request from process whose request was last serviced

Benefit =
best.positioning_time — next.positioning_time

Cost = next.median_thinktime

Waiting_duration =
(Benefit > Cost) ? next.95percentile_thinktime : 0

from http://www.cs.rice.edu/~ssiyer/r/antsched/

Proportional scheduler

Costs and benefits are different.

e.g., proportional scheduler:

Wait for process whose request was last serviced,
1. if it has received less than its allocation, and
2. if it has thinktime below a threshold (e.g., 3ms)

Waiting_duration = next.95percentile_thinktime

from http://www.cs.rice.edu/~ssiyer/r/antsched/

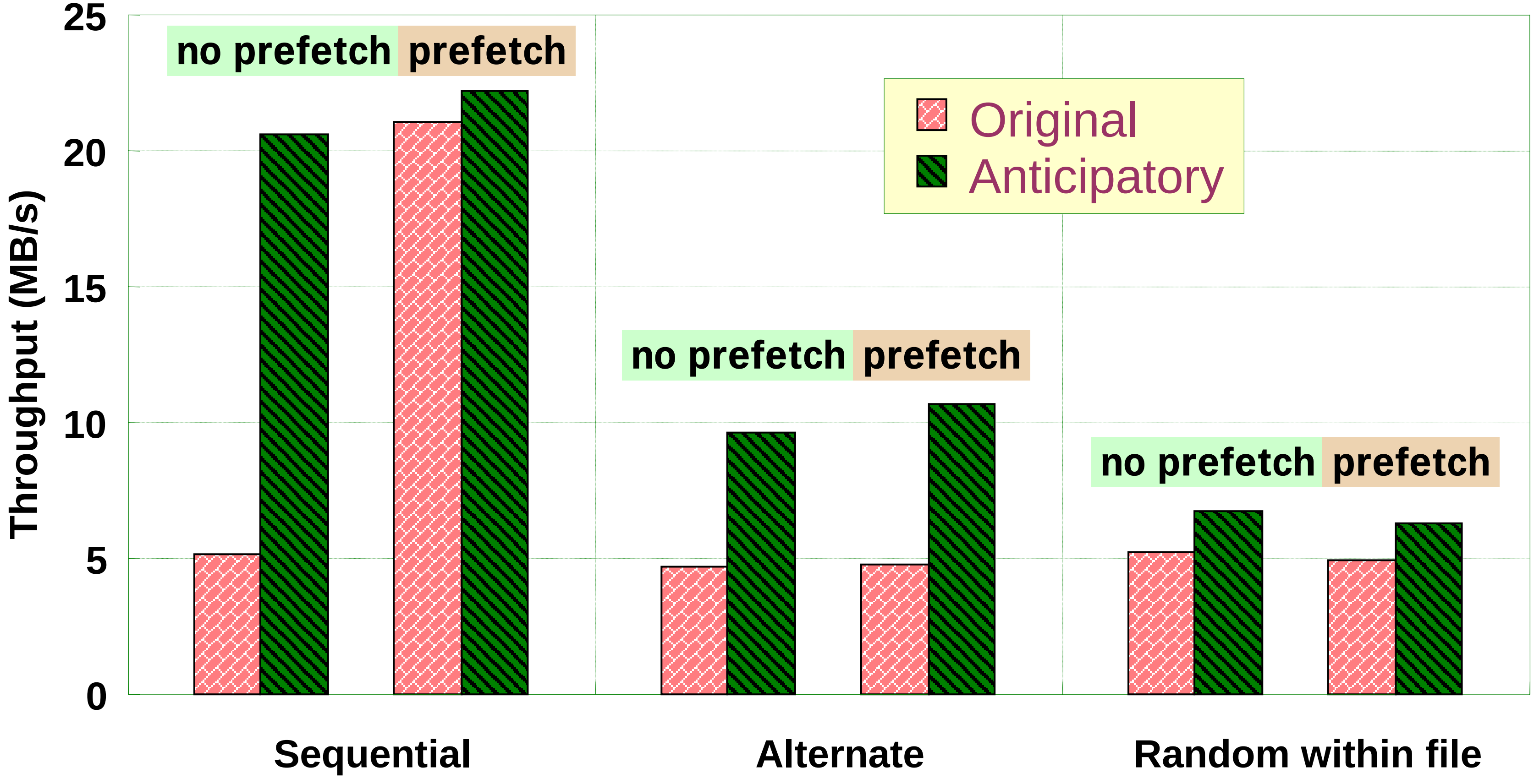
Prefetch

Overlaps computation with I/O.

Side-effect:
avoids deceptive idleness!

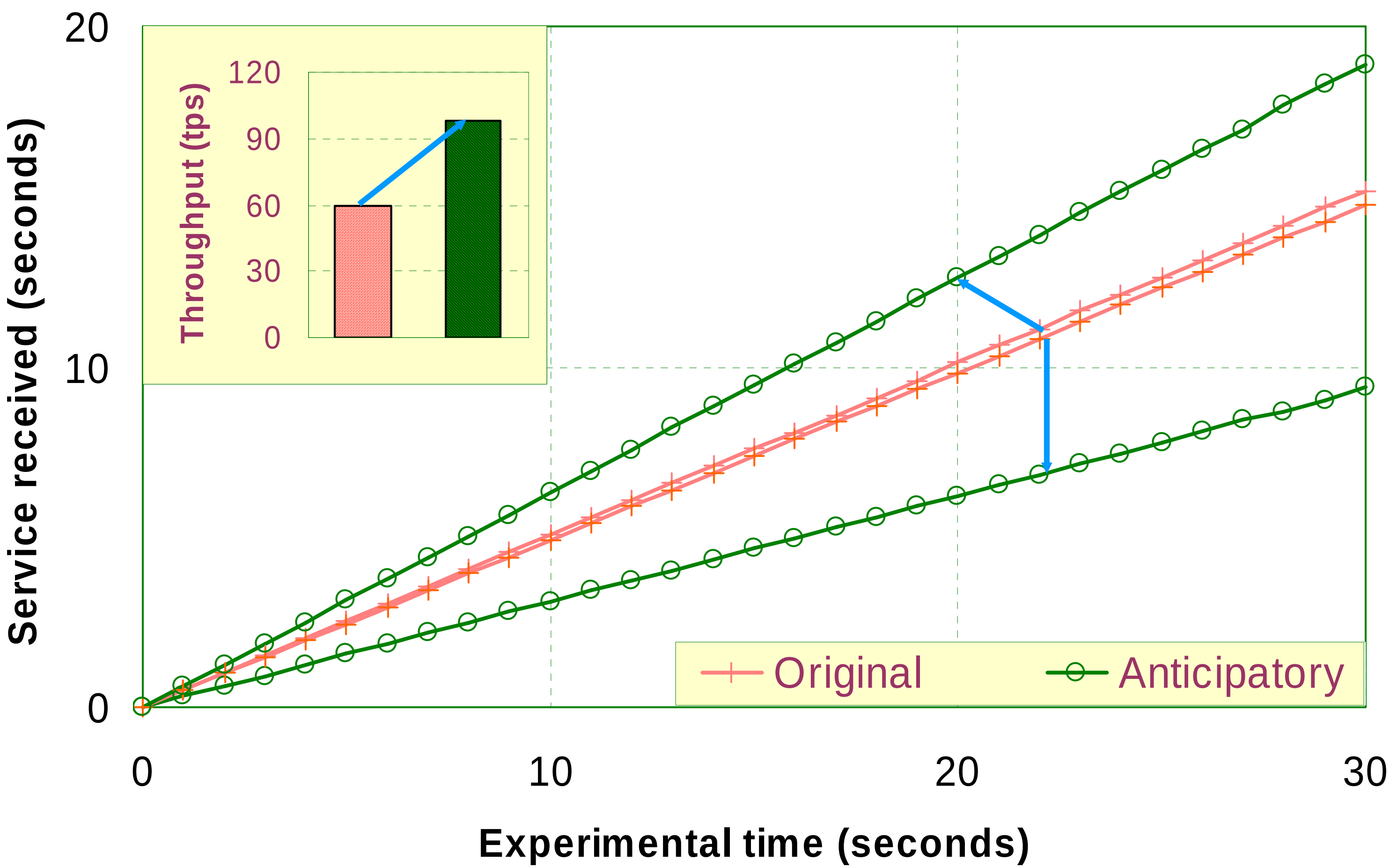
- Application-driven
- Kernel-driven

Microbenchmark



from http://www.cs.rice.edu/~ssiyer/r/antsched/

Proportional scheduler



Database benchmark: two databases, select queries

from http://www.cs.rice.edu/~ssiyer/r/antsched/

Work-conserving vs. non-work-conserving

- Work-conserving scheduler
  - If the disk is idle or a request is completed, next request in the queue is scheduled immediately
- Non-work-conserving scheduler
  - the disk stands idle in the face of nonempty queue
- Anticipatory scheduler are non-work-conserving

Anticipatory I/O scheduler in Linux

- Based on deadline I/O scheduler
- Suitable for desktop, good interactive performance
- Design shortcomings
  - Assume only 1 physical seeking head
    - Bad for RAID devices
  - Only 1 read request are dispatched to the disk controller at a time
    - Bad for controller that supports TCQ
  - Read anticipation assumes synchronous requests are issued by individual processes
    - Bad for requests issued cooperatively by multiple processes
- Rough benefit-cost analysis
  - Anticipate a better request if mean thinktime of the process < 6ms and mean seek distance of the process < seek distance of next request

Anticipatory IO scheduler policy

- One-way elevator algorithm
  - Limited backward seeks
- FIFO expiration times for reads and for writes
  - When a requests expire, interrupt the current elevator sweep
- Read and write request batching
  - Scheduler alternates dispatching read and write batches to the driver. The read (write) FIFO timeout values are tested only during read (write) batches.
- Read Anticipation
  - At the end of each read request, the I/O scheduler examines its next candidate read request from its sorted read list and decide whether to wait for a “better request”

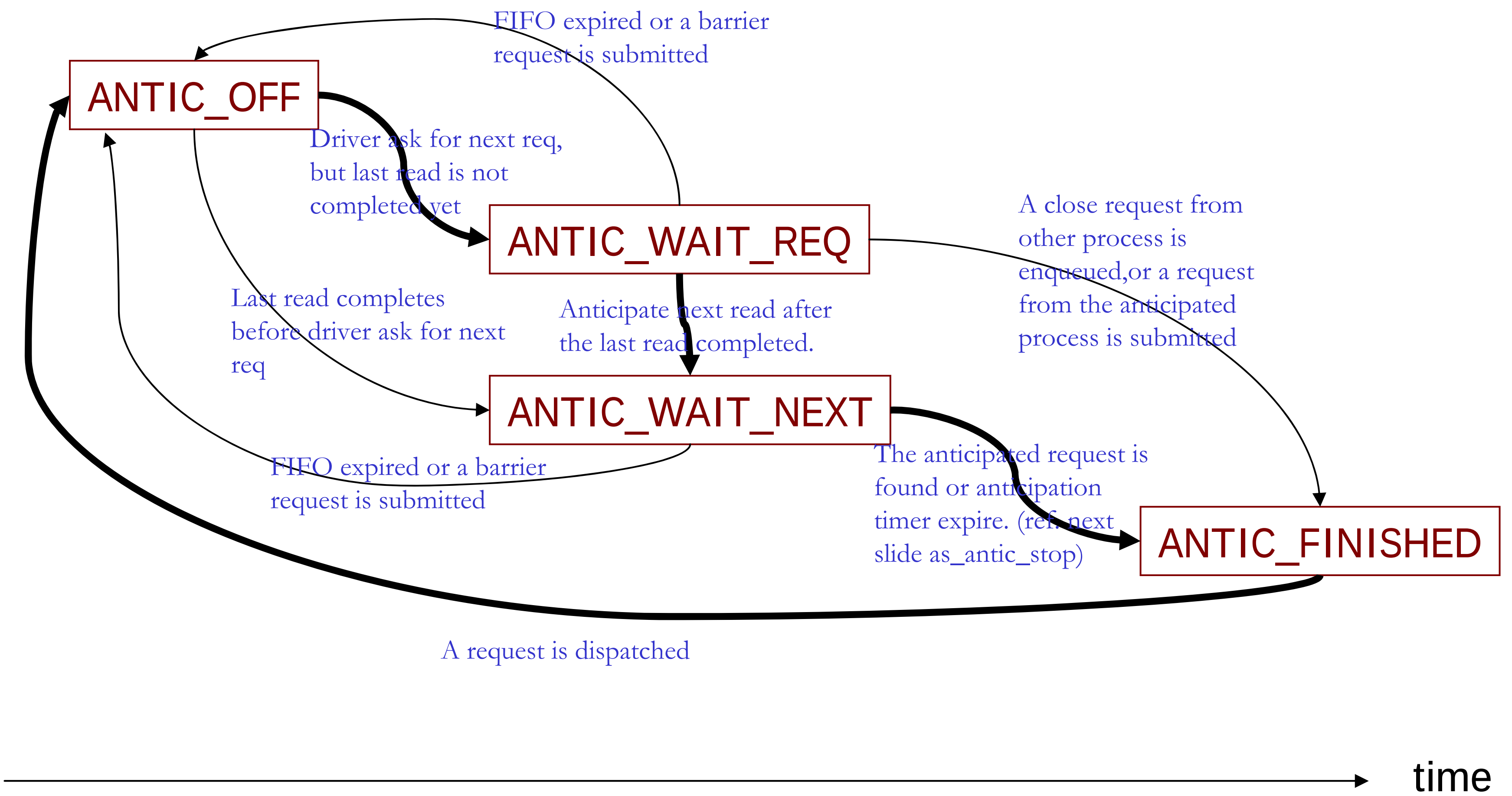
I/O statistics for anticipatory scheduler

- Per request queue (as_data)
  - The last sector of the last request
  - Exit probability
    - Probability a task will exit while being waited on
- Per process (as_io_context)
  - Last request completion time
  - Last request position
  - Mean think time
  - Mean seek distance

Anticipation States

- ANTIC_OFF
  - Not anticipating (normal operation)
- ANTIC_WAIT_REQ
  - The last read has not yet completed
- ANTIC_WAIT_NEXT
  - Currently anticipating a request vs last read (which has completed)
- ANTIC_FINISHED
  - Anticipating but have found a candidate or timed out

State transitions of request anticipation



Functions executed during the anticipation of requests

`as_completed_request`

executed when a request is completed. If the completed request is a read, record completion time and call `as_antic_waitnext`

`as_add_request`

executed when a new request is queued. If this is a read, update the corresponding process's I/O statistics.Conditionally call `as_antic_stop`

`as_antic_waitnext`

Start a timer (expire time: 6ms) to wait the next read request.

`as_antic_stop`

Stop anticipation timer and scheduler a call to `request_fn`
Called in 4 conditions:
1. FIFO queue expired
2. The anticipated process submit a read or write
3. The anticipated process exited
4. A close request is submitted from other process

`as_antic_timeout`

Timer expired. Schedule a call to `request_fn`

time

I/O statistics – thinktime & seek distance

- These statistics are associated with each process, but not with a specific I/O device
  - The statistics will be a combination of I/O behavior from all actively-use devices (It seems bad!)
- Thinktime
  - At enqueuing of a new read request, thinktime = current jiffies – completion time of last read request
- seek distance
  - At enqueuing of a new read request, seek distance = |start sector of the new request – last request end sector|

I/O statistics – average thinktime and seek distance

- Previous I/O history decays as new request are enqueued
- Fixed point arithmetic (1.0 == 1 << 8)

Mean thinktime of a process

$$tsamples = \frac{7 \times tsamples + 256}{8}$$

$$ttotal = \frac{7 \times ttotal + 256 \times thinktime}{8}$$

$$tmean = \frac{ttotal + 128}{tsamples}$$

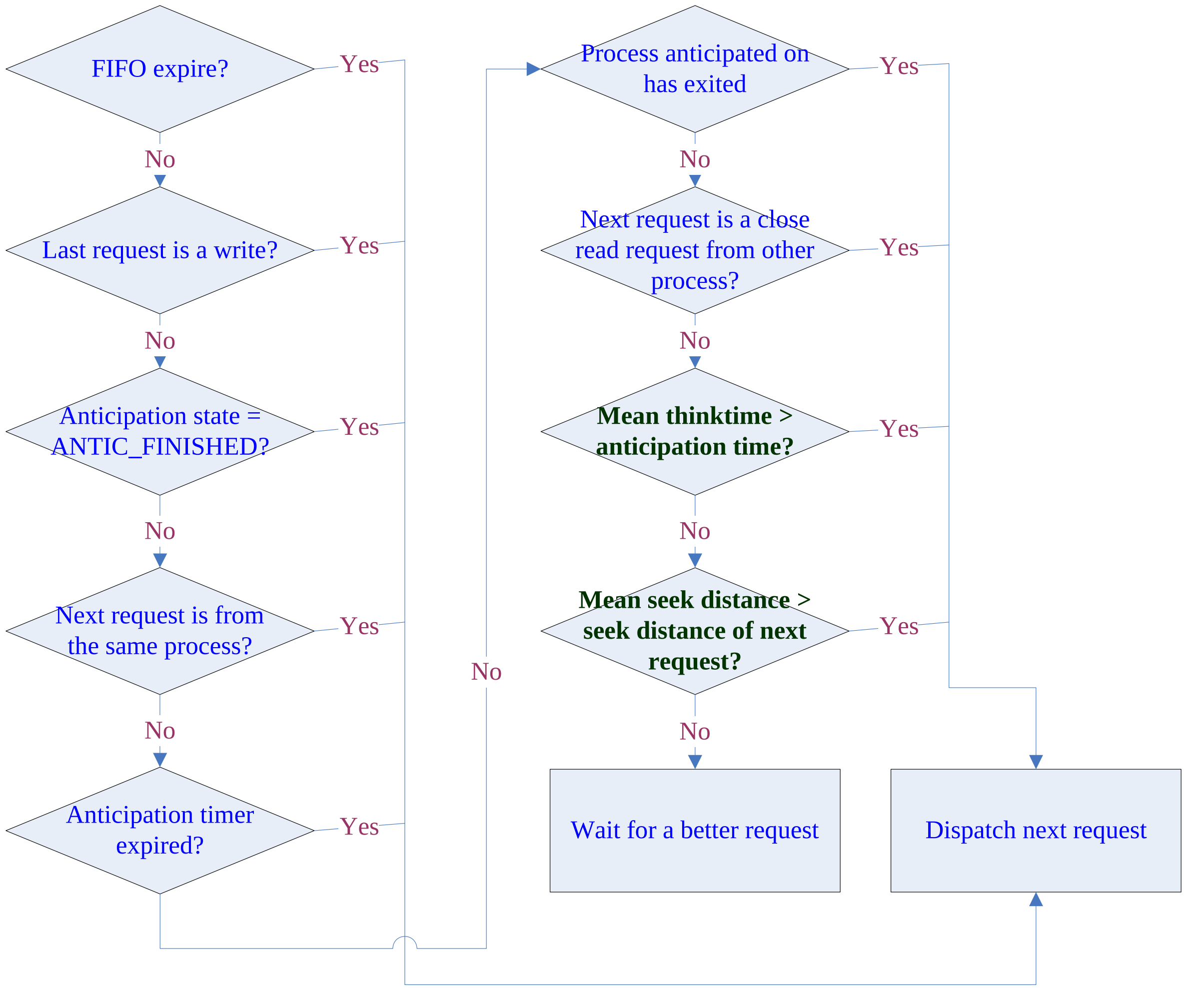
Mean seek distance of a process

$$ssamples = \frac{7 \times ssamples + 256}{8}$$

$$stotal = \frac{7 \times stotal + 256 \times seekdist}{8}$$

$$smean = \frac{stotal + {ssamples}/{2}}{ssamples}$$

Make a decision –
Shall we anticipate a “better request”?



Cooperative Anticipatory Scheduler

- Proposed in this paper: Enhancements to Linux I/O scheduler, OLS2005
- The problems of anticipatory scheduler
  - Anticipation works only when requests are issued by the same process
- Solution
  - Keep anticipating even when the anticipated process has exited
  - Cooperative exit probability: existence of cooperative processes related to dead processes

AS failed to anticipate chunk reads

AS works too well for Program 1. Program 2 starved.

CAS: Performance Evaluation

Streaming writes and reads

Program 1:
```
while true
do
    dd if=/dev/zero of=file \
       count=2048 bs=1M
done
```

Program 2:
```
time cat 200mb-file > /dev/null
```

Scheduler	Execution time (sec)	Throughput (MB/s)
Deadline	129	25
AS	10	33
CAS	9	33

Streaming and chunk reads

Program 1:
```
while true
do
    cat big-file > /dev/null
done
```

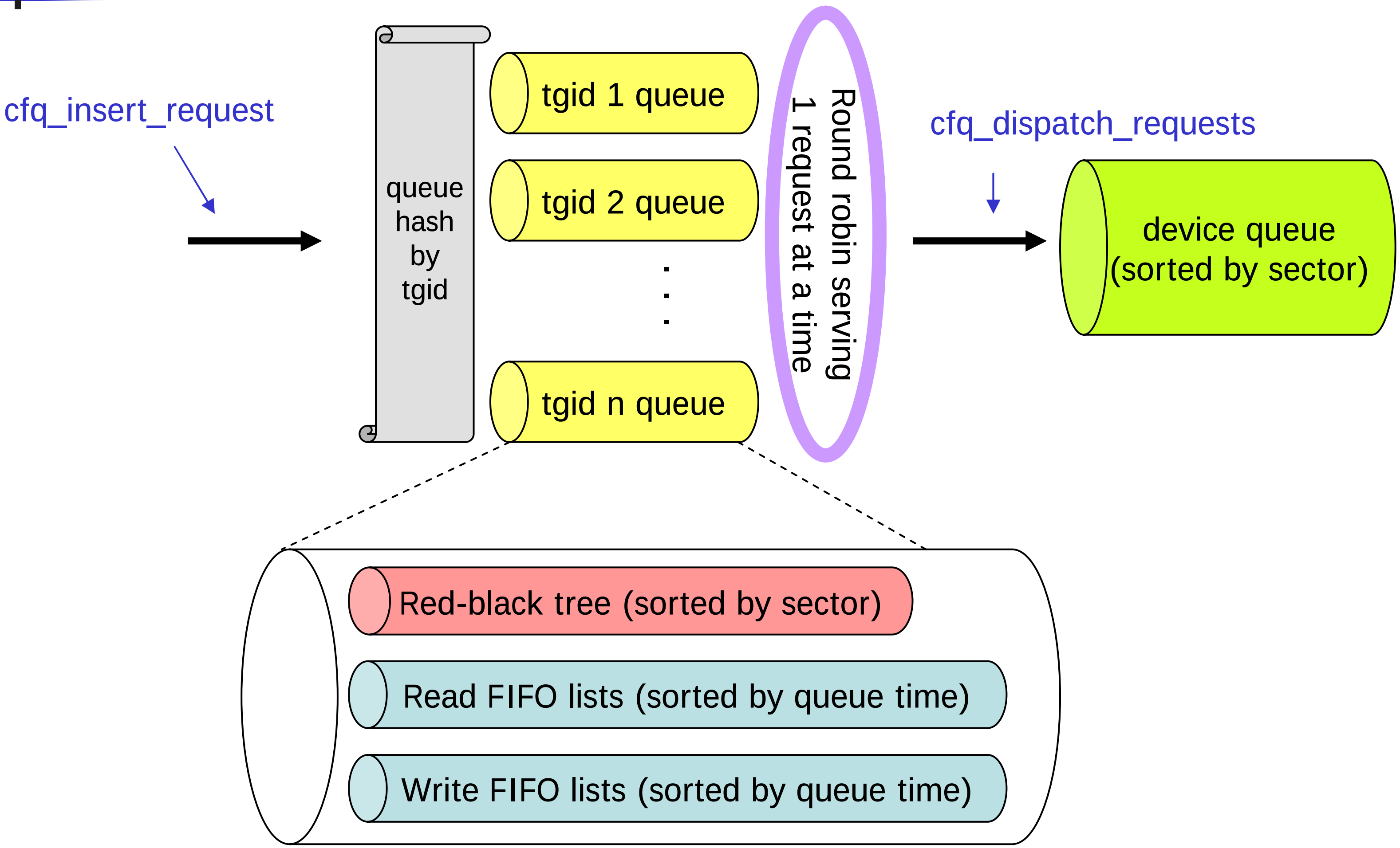
Program 2:
```
time find . -type f -exec \
    cat '{}' ';' > /dev/null
```

Scheduler	Execution time (sec)	Throughput (MB/s)
Deadline	297	9
AS	4767	35
CAS	255	34

CFQv2 (Complete Fair Queuing) I/O scheduler

- Goal
  - Provide fair allocation of I/O bandwidth among all the initiators of I/O requests
- CFQ can be configured to provide fairness at per-process, per-process-group, per-user and per-user-group levels.
- Each initiator has its own request queue and CFQ services these queues round-robin
  - Data writeback is usually performed by the ***pdflush*** kernel threads. That means, all data writes share the alloted I/O bandwidth of the ***pdflush*** threads

Architecture view of CFQv2



References

- Anticipatory scheduling: A disk scheduling framework to overcome deceptive idleness in synchronous I/O, Sitaram Iyer, ACM SOSP'01
- Enhancements to Linux I/O scheduling, Seetharami Seelam, OLS'05
- Linux 2.6.12 kernel source
- Linux Kernel Development, 2nd edition, Robert Love, 2005