As of version 2.6.24, the mainline Linux tree provides four block I/O schedulers: Noop, Deadline, Anticipatory (AS) and Completely Fair Queuing (CFQ). For benchmarking purposes, we have developed the First-In First-Out (FIFO) scheduler.

1 Background

1.1 Block Layer Structure

The Linux block I/O subsystem is shown in Figure 1. As far as I/O scheduling is concerned, the key components are the elevator layer, the I/O scheduler itself, and the dispatch queue.

The elevator layer connects the scheduler with the generic block I/O layer via an interface exported by the scheduler. It also provides basic functionality common to many I/O schedulers, such as merge searching (see Section 1.3).

The I/O scheduler is responsible for taking requests from the elevator layer, rearranging them according to some scheduling policy, and then populating the dispatch queue with requests which are consumed by the device driver and ultimately issued to the underlying device.

1.2 Merging

Two I/O requests are said to be logically adjacent when the end-sector of one request corresponds to the sector immediately before the start of the other. If these requests are in the same direction (that is, read or write), they can then be merged into a single larger request.

When searching for merge opportunities while adding a new request, two types of merges are possible: front- and back-merges. A front-merge occurs when the new request falls before an existing adjacent request, according to

---

1 Although the term “elevator” typically refers to a specific scheduling algorithm, Linux uses this term to refer generically to I/O schedulers.
start-sector order. When the new request falls after an existing request, a back-merge occurs.

Figure 2 shows merging graphically; a request for sector 3 can be front-merged with an existing request for sectors 4 and 5, while the request for sector 6 can be back-merged with the same existing request.

1.2.1 Coalescing

After an existing request has been involved in a merge, it is possible that the growth of the request causes it to become adjacent to another existing request. As with merging, these requests can then be coalesced into a single larger request. In Figure 2, after back-merging request 6 with the existing request for sectors 4 and 5, the resulting request (now for sectors 4, 5 and 6) can be coalesced with request 7.

Figure 2: Request merging and coalescing

1.3 Elevator-layer Merging

The Linux generic elevator layer maintains a hash table of requests indexed by end-sector number, which is used to discover back-merge opportunities with an ideally constant-time hash table lookup. No front-merging functionality is provided, and so must be implemented by each I/O scheduler if desired.
The elevator layer also provides a “one-hit” merge cache which stores the last request involved in a merge. This cache is checked for both front- and back-merge possibilities before performing a more general merge search.

Requests involved in a merge are automatically coalesced using the `elevator_former_req_fn` and `elevator_latter_req_fn` functions which are provided by the I/O scheduler.

### 1.4 Synchronicity

**TODO:** This section needs a better name

Block requests can be classified into two types: synchronous and asynchronous.

Synchronous requests are those which have a client process blocked waiting on the result of the request. These are usually the result of a read operation, such as through the `read` POSIX function. On the other hand, asynchronous requests do not have a client process blocked waiting on the result. Writes via calls to the `write` function usually result in asynchronous requests. **TODO:** This is strictly incorrect, but maybe close enough for practical purposes.

The correlation between data direction (read and write) and synchronicity is not always the case. The `aio.*` family of functions, for instance, allows issuing on asynchronous reads. `open` parameters, such as `O_DIRECT` and `O_ASYNC`, can also affect the synchronicity.

### 1.5 Tunables

**TODO:** This section

Describe what a tunable is and where to find them.

### 1.6 Scheduler API

**TODO:** Do I want this?
2 Noop

The Noop (no-operation) I/O scheduler provides minimal functionality, performing back-merging, limited coalescing, and sorting of requests only on the dispatch queue.

Requests are organised in a simple first-in, first-out (FIFO) queue. Before adding a new request to the queue, back-merge opportunities are discovered and performed using the sector-indexed hash table provided by the generic elevator layer, but no front-merging is attempted. The scheduler will only coalesce requests that are adjacent both in sector position and in request order, since it does not know the sector-order of requests (no such data-structure is maintained). When dispatching a request, Noop will perform a sector-sorted insertion of the request on the front of its internal queue into the dispatch queue.

Noop has no tunable parameters.

3 FIFO

The FIFO, or First-In First-Out I/O scheduler issues requests to the disk in exactly the form they are received, without any merging or re-ordering. This simulates a system with no I/O scheduling capacity, and so can be used as a basis for benchmarking.

FIFO has no tunables.
4 Deadline

The goal of the Deadline I/O scheduler is to maximise global throughput while bounding time requests spend in the queue.

4.1 Architecture

Requests in the Deadline scheduler are organised into two types of logical queues. The first, sort_list, is a Red-Black (RB) tree sorted according to request sector number. It is used to traverse pending requests in increasing sector-order and for lookup of merge opportunities. Read and write requests are stored in separate sort_lists.

The second type of queue is the fifo_list, which stores requests in FIFO order; that is, the oldest request is at the head of the queue. As with sort_list, there are separate read and write queues which are implemented as doubly-linked lists since only the front and back of the queue is of interest.

New requests are added to both the fifo_list and sort_list corresponding to that request’s data direction (read or write).

4.2 Batching

Requests are normally added to the dispatch queue from the sort_list queues in increasing-sector order, similar to the CSCAN algorithm, in batches. A batch consists of up to fifo_batch contiguous (with respect to sector-number) requests of the same data direction which are treated as a single unit by the I/O scheduler.

A batch is terminated when one of the following conditions occur:

- No more requests are available for the current batch’s data direction;
- The next request is not sequential with respect to the last request dispatched;
- The current batch has dispatched fifo_batch requests.

4.3 Deadlines

Each request in the Deadline scheduler is assigned an expiry time (its deadline), which is determined by the read_expire and write_expire tunable (see Section 4.6), and dictates the maximum time this request should be allowed to go unserviced. Note these are soft deadlines and do not provide any sort of real-time performance guarantees; the actual service latency may be significantly greater (or indeed less) than the corresponding expiry time.

When the request on the front of the fifo_list expires (that is, its expiry time elapses), Deadline will expedite dispatch of that request.
4.4 Dispatch

The following procedure is executed each time a new request is to be selected for dispatch:

1. Dispatch from the current batch until one of the conditions in Section 4.2 occur;

2. Select a new data direction. Choose reads unless there are no reads pending, or there are pending write requests that we have starved for too long, according to the writes_starved tunable;

3. Check for an expired request in the fifo_list queue of the selected data direction. If there is one, start a new batch with that request;

4. If we selected the same direction as the previous batch, start a new batch where the previous one left off;

5. Start a new batch at the request with the earliest expiry time in the chosen direction.

New requests are always added to the tail of the dispatch queue.

4.5 Merging

Deadline performs back-merging via the sector-sorted hash table provided by the elevator layer, along with optional front-merging (specified with the front_merge tunable) which utilises sort_list to discover adjacent requests.

Requests that are made adjacent as a result of a merge are discovered and coalesced.

4.6 Tunables

The Deadline scheduler exports five tunables to user-space. Table 1 summarises their function and default values.
<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_expire</td>
<td>500ms</td>
<td>Expiry time for read requests.</td>
</tr>
<tr>
<td>write_expire</td>
<td>5s</td>
<td>Expiry time for write requests.</td>
</tr>
<tr>
<td>writes_starved</td>
<td>2</td>
<td>Number of read batches to issue before switching to a write batch.</td>
</tr>
<tr>
<td>front_merge</td>
<td>true</td>
<td>Enables searching for front-merge opportunities.</td>
</tr>
<tr>
<td>fifo_batch</td>
<td>16</td>
<td>Maximum number of requests per batch.</td>
</tr>
</tbody>
</table>

Table 1: Summary of Deadline tunable parameters

5 Anticipatory

The Anticipatory scheduler (AS) aims to improve throughput by exploiting locality in I/O requests using a technique called *anticipation*.

5.1 Overview

Structurally AS is similar to Deadline; requests are inserted into both sorted and FIFO queues. It also implements similar policies. AS always performs full back- and front-merging and coalescing, and implements request deadlines which are tuned with `read_expire` and `write_expire`. AS also issues requests in batches, but unlike Deadline, the batches are time-based. Requests in AS batches do not need to be contiguous or even monotonic.

The key difference between the Anticipatory and Deadline schedulers is the use of anticipation, discussed in Section 5.3.

Strictly speaking, AS does not differentiate between read and write requests like Deadline, but rather synchronous and asynchronous requests. However, much of the existing documentation uses the terms read and write interchangeably with synchronous and asynchronous.

5.2 Batching

A batch in the AS scheduler is a collection of requests in the same direction (not necessarily contiguous) over a fixed period of time: `read_batch_expire` ms for synchronous requests and `write_batch_expire` ms for asynchronous requests. Asynchronous batches are additionally limited to a number of requests which varies at run-time. This batch limit is adjusted according to measured time for previous batches to get actual batch times close to the specified time, `write_batch_expire`.

New synchronous batches are started from the earliest entry in the FIFO queue. New asynchronous batches, however, continue from where the previous
asynchronous batch completed.

5.3 Anticipation

Anticipation aims to defeat a phenomenon called deceptive idleness [ID01]. When a process issues synchronous, sequential I/O requests, there is a period before the next request is issued, but after the previous one completes, where the process has no outstanding requests (the think-time). The scheduler may assume that the process has no further requests, and switch to servicing other processes. As a result, it may miss opportunities to issue sequential requests.

If anticipation is likely to improve performance, AS will insert a pause of up to \texttt{antic\_expire} ms waiting for a suitable request. Anticipation will only occur during synchronous batches, and the following conditions must hold for a process if the scheduler will anticipate a request from it:

1. It has no outstanding requests;
2. It is still running, or it is likely cooperating with another process and either not running or likely to exit soon;
3. The mean think-time is less than \texttt{antic\_expire};
4. There are no other synchronous requests available (from any process) that are closer than the one expected from this process.

If the scheduler receives the request it was anticipating or a new synchronous request that is closer than the one anticipated, it will leave the anticipation state and immediately dispatch the request. If the request anticipated does not arrive within \texttt{antic\_expire} ms, the scheduler simply continues with the next request.

5.4 Dispatch

AS alternates between dispatching read and write batches. Within each batch, requests are selected in a CSCAN-like manner. However, backward seeks are allowed if:

1. The nearest backward-request is more than twice as close as the nearest forward-request; and
2. The backward-request is no more than 512 MiB away from the current position.

Before dispatching a request, and up to once per expiry period (\texttt{read\_expire} or \texttt{write\_expire}, depending on the current batch direction), the FIFO queue is checked for expired requests. If there is an expired request in the current data direction, it is immediately dispatched.
5.5 Tunables

The tunables available for the Anticipatory scheduler are summarised in Table 2.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_expire</td>
<td>125ms</td>
<td>Expiry time for synchronous requests.</td>
</tr>
<tr>
<td>write_expire</td>
<td>250ms</td>
<td>Expiry time for asynchronous requests.</td>
</tr>
<tr>
<td>read_batch_expire</td>
<td>500ms</td>
<td>How long to batch synchronous requests before switching to an asynchronous batch.</td>
</tr>
<tr>
<td>write_batch_expire</td>
<td>125ms</td>
<td>How long to batch asynchronous requests before switching to a synchronous batch.</td>
</tr>
<tr>
<td>antic_expire</td>
<td>6.7ms</td>
<td>Maximum time to anticipate a synchronous request.</td>
</tr>
</tbody>
</table>

Table 2: Summary of AS tunable parameters
6 CFQ

The Completely Fair Queueing (CFQ) I/O scheduler tries to provide each competing process with an equal time share of access to the underlying device. It also provides many of the features of the Deadline and Anticipatory schedulers.

6.1 Priorities

CFQ is currently the only scheduler that is aware of I/O priorities. Each process has an associated priority class, which defines what type of service is received. The priority classes in order of priority are real-time (RT), best-effort (BE), and idle. The RT and BE classes are further divided into eight priority levels that indicate the level of service required within the class, with zero being the highest, and seven the lowest. If a process doesn’t have its priority explicitly set, the class defaults to best-effort, and the level is derived from its nice value.

6.2 Structure

The fundamental abstraction in CFQ is the cfq_queue. Each queue acts as an independent source of requests, and the queues are serviced in a manner that allows priority-based fair queuing.

Each process has its own cfq_queue for synchronous requests. For asynchronous requests, there is one queue per priority—eight RT queues, eight BE queues, and an idle queue. These are shared among all processes. As a consequence, per-process fairness only applies to synchronous requests. Priority is still respected.

When a queue has requests pending, it is entered into the service tree, which stores the queues in the order in which they will be serviced. The order is a function of the queue’s priority.

Within each cfq_queue, requests are stored in both a sector-sorted queue and a FIFO list.

6.3 Fairness

CFQ

6.4 Dispatch

Choosing the next request to dispatch is divided into two separate tasks: selecting the queue to service, and choosing a request within the queue. Before starting a dispatch, the following algorithm is used to select a queue:

1. If the current queue has requests pending and has not used its time slice, dispatch from that queue;
2. If the current queue has no requests pending but is idling or has requests in-flight, wait for completion before selecting a new queue;

3. Select the next queue in service order from the service_tree. If it is an idle queue, only service it if slice_idle ms have elapsed.

Idle-class queues are limited to one request per round. All other queues are limited to quantum requests, and asynchronous queues have an additional request limit which is derived from its priority.

### 6.5 Tunables

The tunables available for the Completely Fair Queuing scheduler are summarised in [Table 3](#).

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>back_seek_max</td>
<td>16384 KiB</td>
<td>Maximum allowed size of backward seeks.</td>
</tr>
<tr>
<td>back_seek_penalty</td>
<td>2</td>
<td>Factor by which forward seeks are considered better than backward seeks.</td>
</tr>
<tr>
<td>fifo_expire_async</td>
<td>250ms</td>
<td>Expiry time of asynchronous requests.</td>
</tr>
<tr>
<td>fifo_expire_sync</td>
<td>125ms</td>
<td>Expiry time of synchronous requests.</td>
</tr>
<tr>
<td>quantum</td>
<td>4</td>
<td>Maximum requests per-queue to service each round.</td>
</tr>
<tr>
<td>slice_async</td>
<td>40ms</td>
<td>Base time slice for asynchronous requests.</td>
</tr>
<tr>
<td>slice_async_rq</td>
<td>2</td>
<td>Base number of asynchronous requests per round.</td>
</tr>
<tr>
<td>slice_idle</td>
<td>8ms</td>
<td>Time to wait for a queue to produce more I/O before switching to a new queue.</td>
</tr>
<tr>
<td>slice_sync</td>
<td>100ms</td>
<td>Base time slice for synchronous requests.</td>
</tr>
</tbody>
</table>

Table 3: Summary of CFQ tunable parameters
References