Linux kernel hacking

Process containers - cgroups

LinuxDay 2009 – Siena (Oct 24), Andrea Righi <righi.andrea@gmail.com>

OS requirements

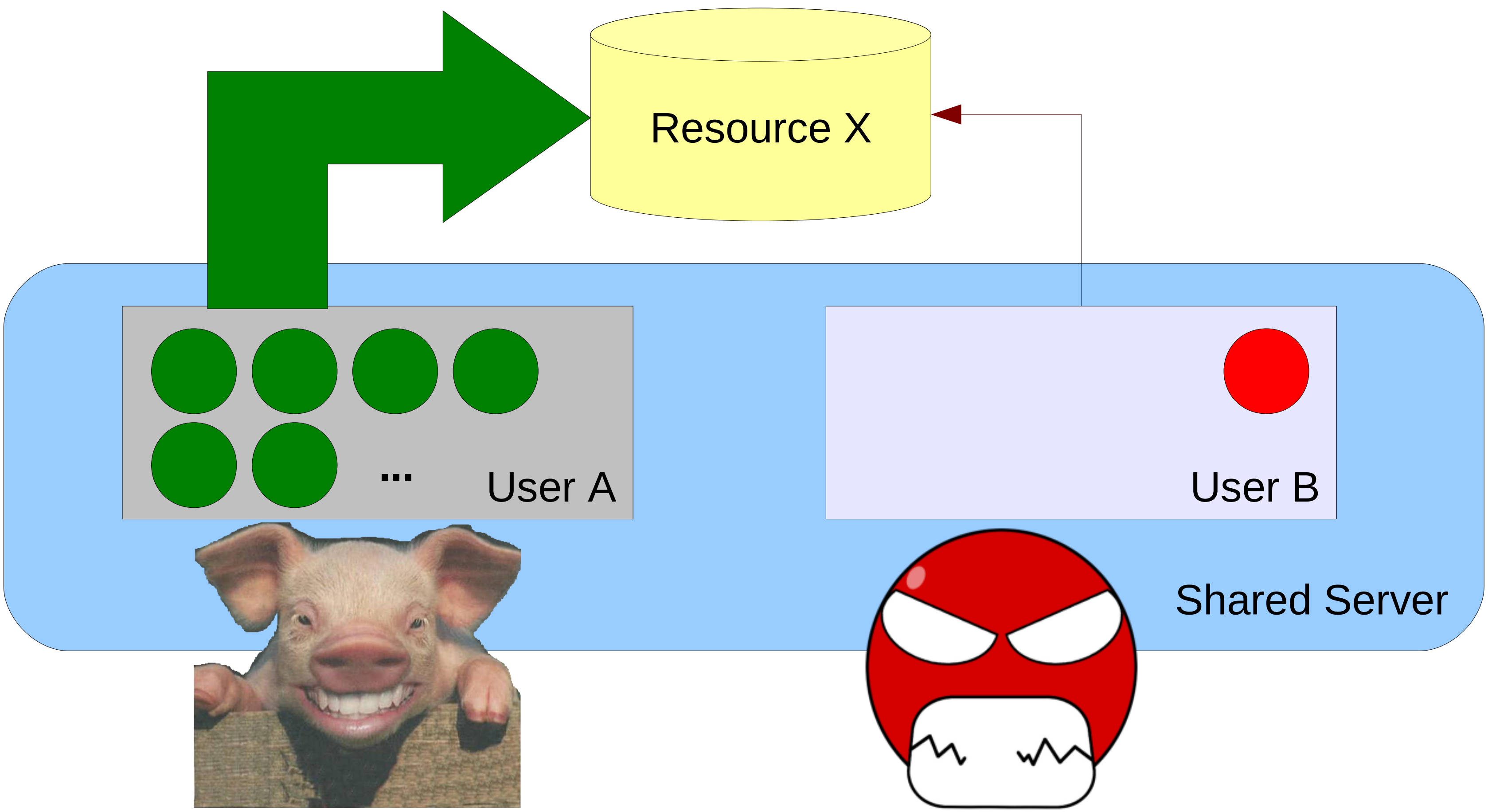
- Fair allocation of resources
  - Equal bandwidth to logical groups
- Operating systems must provide fair allocation of:
  - CPU Management
  - Task management
  - I/O management
  - Memory management
  - Network management
  - ...
- The concept of *task*, *user* and *group* (POSIX) may be not enough...

A typical scenario

- You're the sysadmin of a large hosting company
- Hundreds of users grouped in different *pay-per-use* classes (QoS)
- All need to get their fair share on single servers

Cheat: how to break the fairness on a shared host?

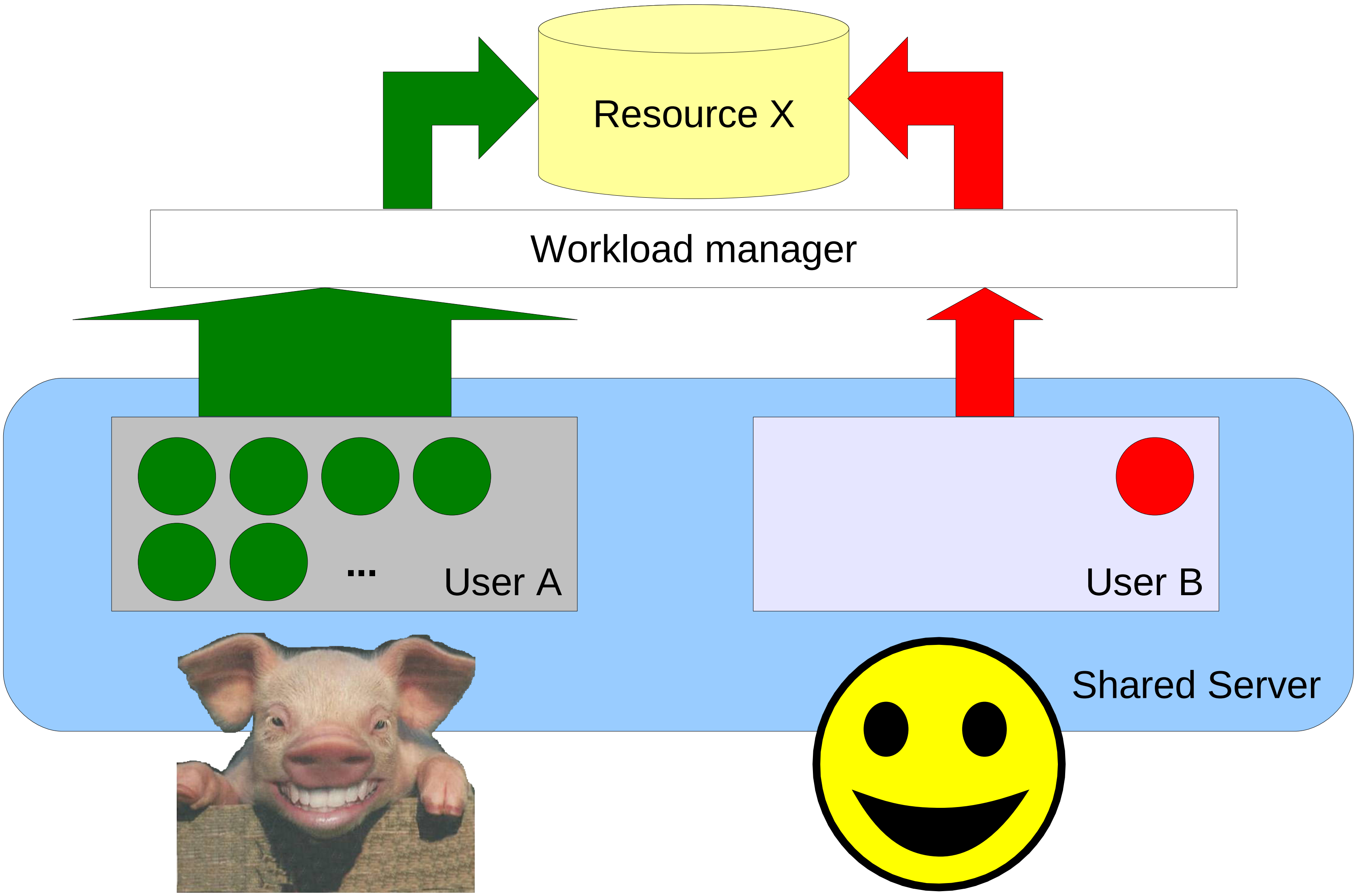
- *ulimit* affects the current shell execution environment
- Create many shells with many heavy processes



Solutions

- One physical server per user ← too much expensive!
- One virtual server per user – VPS ← difficult to maintain!
- OS resource management/partitioning ← OK!
  - Monitor consumed resources per user or class of user
  - Perform immediate actions on policy enforcement

Fair resource allocation



Cgroup: process container

- From *Documentation/cgroups/cgroups.txt*:
  - A **cgroup** associates a set of tasks with a set of parameters for one or more subsystems
  - A **subsystem** is a module that makes use of task grouping facilities provided by cgroups to treat groups of tasks in particular ways
- The *cgroup infrastructure* offers only the grouping functionality
- The *cgroup subsystems* apply the particular accounting/control policies to the group of tasks

Where are these “cgroups”?

- Part of the core Linux kernel (vanilla)
  - Linux >= 2.6.24
- Subsystems:
  - cpu, cpuacct, cpuset, memory, devices, freezer
- Source code:
  - kernel/cgroup.c
  - include/linux/cgroup.h
  - include/linux/cgroup_subsys.h
  - + *various cgroup subsystems implementation...*

Userspace interface: cgroup filesystem

- Mount the cgroup filesystem
  - **`mkdir /cgroup`**
    `mount -t cgroup -o` ***`subsys1,subsys2,...`*** **`none /cgroup`**
- Configure the cgroup subsystems using virtual files:
  - ```
    ls /cgroup
    subsys1.*
    subsys2.*
    ...
    tasks
    notify_on_release
    release_agent
    ```
- Create a cgroup instance *“foo”*:
  - **`mkdir /cgroup/foo`**
- Move a task (i.e. the current shell) into cgroup *“foo”*:
  - **`echo $$ > /cgroup/foo/tasks`**
    ```

Task selection

- Show the list of PIDs contained in a cgroup, reading the file “tasks” in the cgroup filesystem
  - PIDs in the root cgroup:

```
$ cat /cgroup/tasks
1
2
3
...
```

  - PIDs in the cgroup “*foo*”:

```
$ cat /cgroup/foo/tasks
2780
2781
```

Task selection: examples

- Example #1 – count the number of PIDs in cgroup "foo":

```
# wc -l /cgroup/foo/tasks
4
```

- Example #2 – kill all the PIDs contained in the cgroup "*bar*":

```
# kill $(cat /cgroup/bar/tasks)
```

- Example #3 – set the nice level of the processes contained in cgroup "*baz*" to 5:

```
# renice 5 -p $(cat /cgroup/baz/tasks)
```

Resource management

- Account/control the usage of system resources:
  - CPU
  - Memory
  - I/O bandwidth
  - Network bandwidth
  - Access permission to particular devices
  - ...
- We need a cgroup subsystem for each resource

Cgroup vs Virtualization

- Cgroups are a form of lightweight virtualization
  - While virtualization creates a new virtual machine upon which the guest system runs, cgroups implementation work by making walls around groups of processes
  - The result is that, while virtualized guests each run their own kernel (and can run different operating systems than the host), cgroups all run on the same host's kernel
  - Cgroups lack the complete isolation provided by a full virtualization solution, but they tend to be more efficient!

CPU management

- Cgroup CPU subsystem
  - Controlled by the Completely Fair Scheduler – CFS
- Give the same CPU bandwidth to the cgroup "*multimedia*" and the cgroup "*browser*":
  - `echo 1024 > /cgroup/browser/cpu.shares`
  - `echo 1024 > /cgroup/multimedia/cpu.shares`
- Q: is it really fair?

Without CPU cgroup subsystem
(10 tasks in “*multimedia*” and 5 tasks in “*browser*”)

```
%CPU %MEM    TIME+  COMMAND
  7  0.0   0:00.82 cpuhog-multimedia
  7  0.0   0:00.80 cpuhog-multimedia
  7  0.0   0:00.86 cpuhog-browser
  7  0.0   0:00.88 cpuhog-browser
  7  0.0   0:00.86 cpuhog-browser
  7  0.0   0:00.89 cpuhog-browser
  7  0.0   0:00.81 cpuhog-multimedia
  7  0.0   0:00.82 cpuhog-multimedia
  7  0.0   0:00.78 cpuhog-multimedia
  7  0.0   0:00.80 cpuhog-multimedia
  7  0.0   0:00.81 cpuhog-multimedia
  7  0.0   0:00.82 cpuhog-multimedia
  6  0.0   0:00.87 cpuhog-browser
  6  0.0   0:00.80 cpuhog-multimedia
  6  0.0   0:00.81 cpuhog-multimedia
```

multimedia => 66.66%
browser   => 33.33%

With CPU cgroup subsystem
(10 tasks in “*multimedia*” and 5 tasks in “*browser*”)

```
%CPU %MEM    TIME+  COMMAND
 10  0.0   0:00.51 cpuhog-browser
 10  0.0   0:00.50 cpuhog-browser
 10  0.0   0:00.51 cpuhog-browser
 10  0.0   0:00.50 cpuhog-browser
 10  0.0   0:00.50 cpuhog-browser
  5  0.0   0:00.24 cpuhog-multimedia
  5  0.0   0:00.24 cpuhog-multimedia
  5  0.0   0:00.24 cpuhog-multimedia
  5  0.0   0:00.23 cpuhog-multimedia
  5  0.0   0:00.23 cpuhog-multimedia
  5  0.0   0:00.22 cpuhog-multimedia
  5  0.0   0:00.23 cpuhog-multimedia
  5  0.0   0:00.23 cpuhog-multimedia
  5  0.0   0:00.23 cpuhog-multimedia
  5  0.0   0:00.22 cpuhog-multimedia
```

multimedia => 50.00%
browser => 50.00%

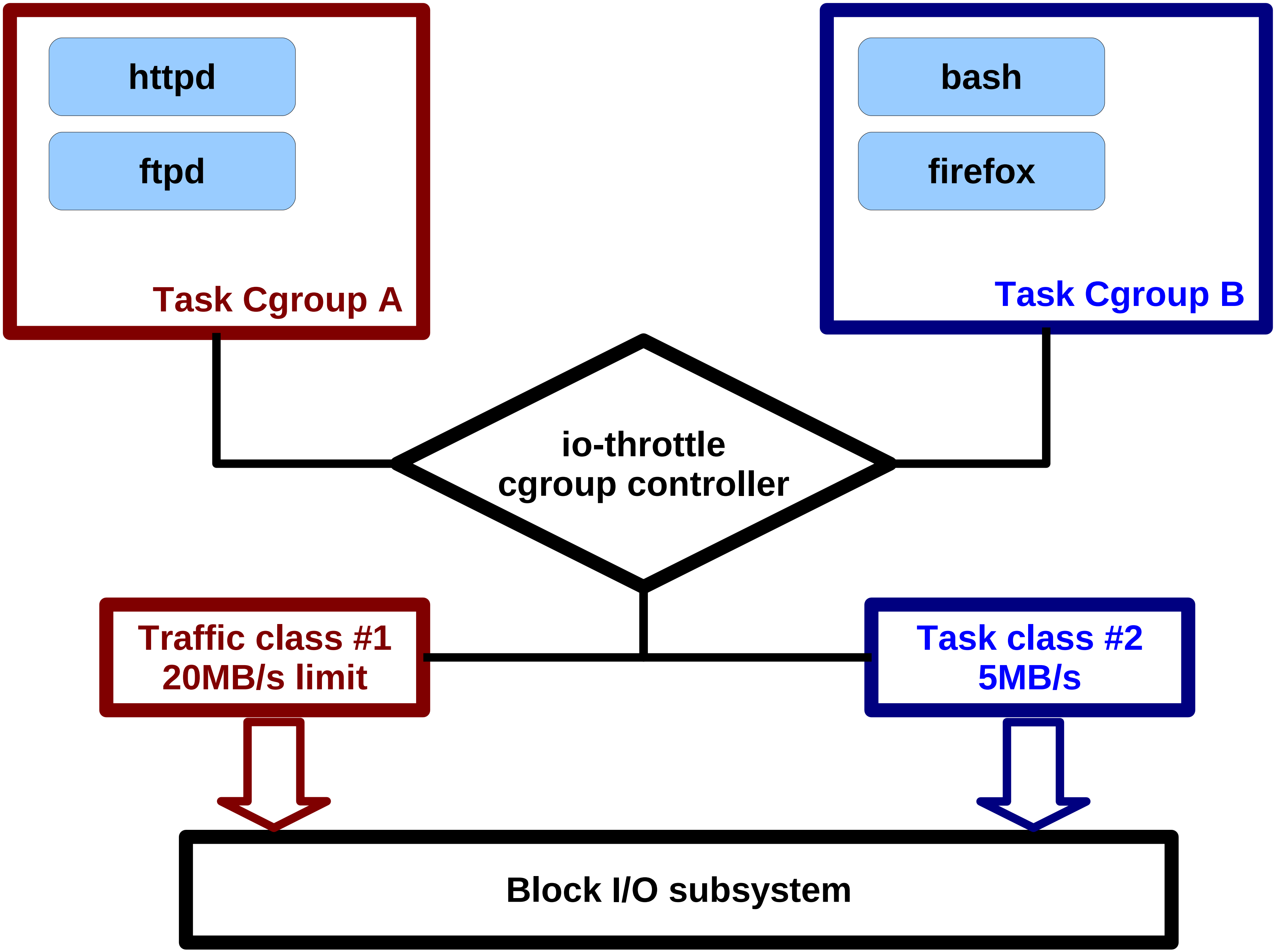
Memory management

- Enable control of anonymous, page cache (mapped and unmapped) and swap memory pages
  - Memory hungry applications can be limited to a smaller amount of memory
  - No more downtime due to global OOM in shared hosts!
- Configuration:
  - `echo 128M > /cgroup/browser/memory.limit_in_bytes`
  - `echo 256M > /cgroup/multimedia/memory.limit_in_bytes`

I/O management: io-throttle patch

- Under development: not yet included in the mainline kernel!
- Approach: block I/O requests if a cgroup exceeds its own ration of bandwidth
- Uses the cgroup virtual filesystem to configure block device BW and iops limit:
  - `echo /dev/sda:$((10 * 1024 * 1024)):0 > /cgroup/browser/blockio.bandwdith`
  - `echo /dev/sda:1000:0 > /cgroup/browser/blockio.iops`

Cgroup io-throttle: overview

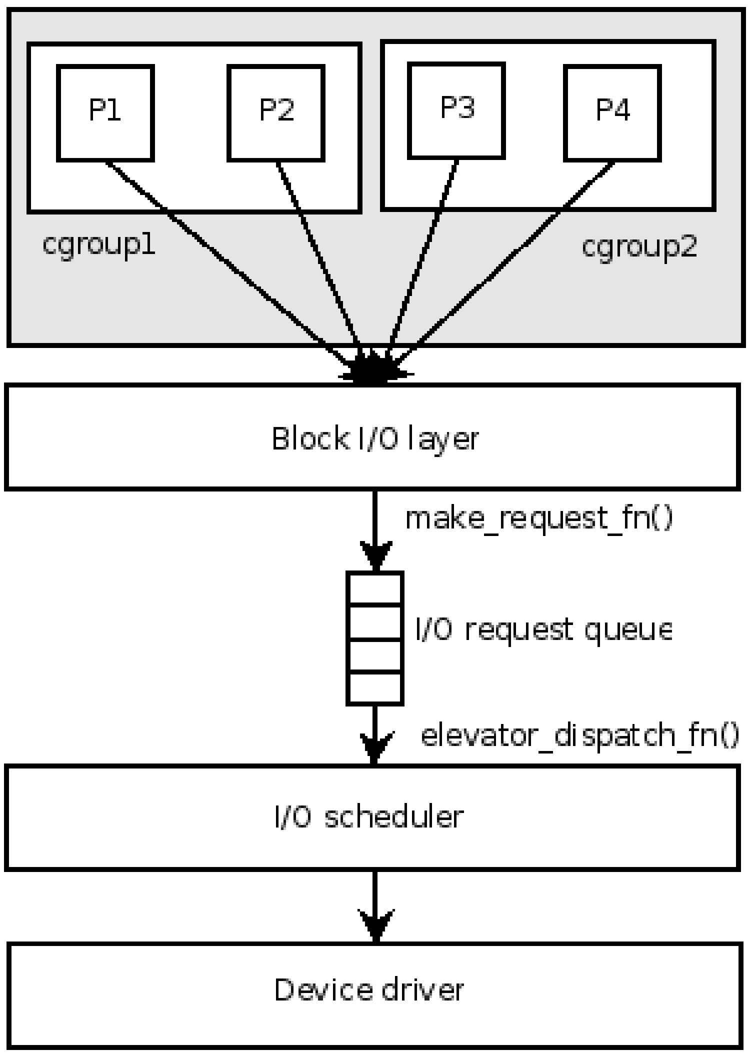


Is throttling an effective approach?

Advantages of throttling

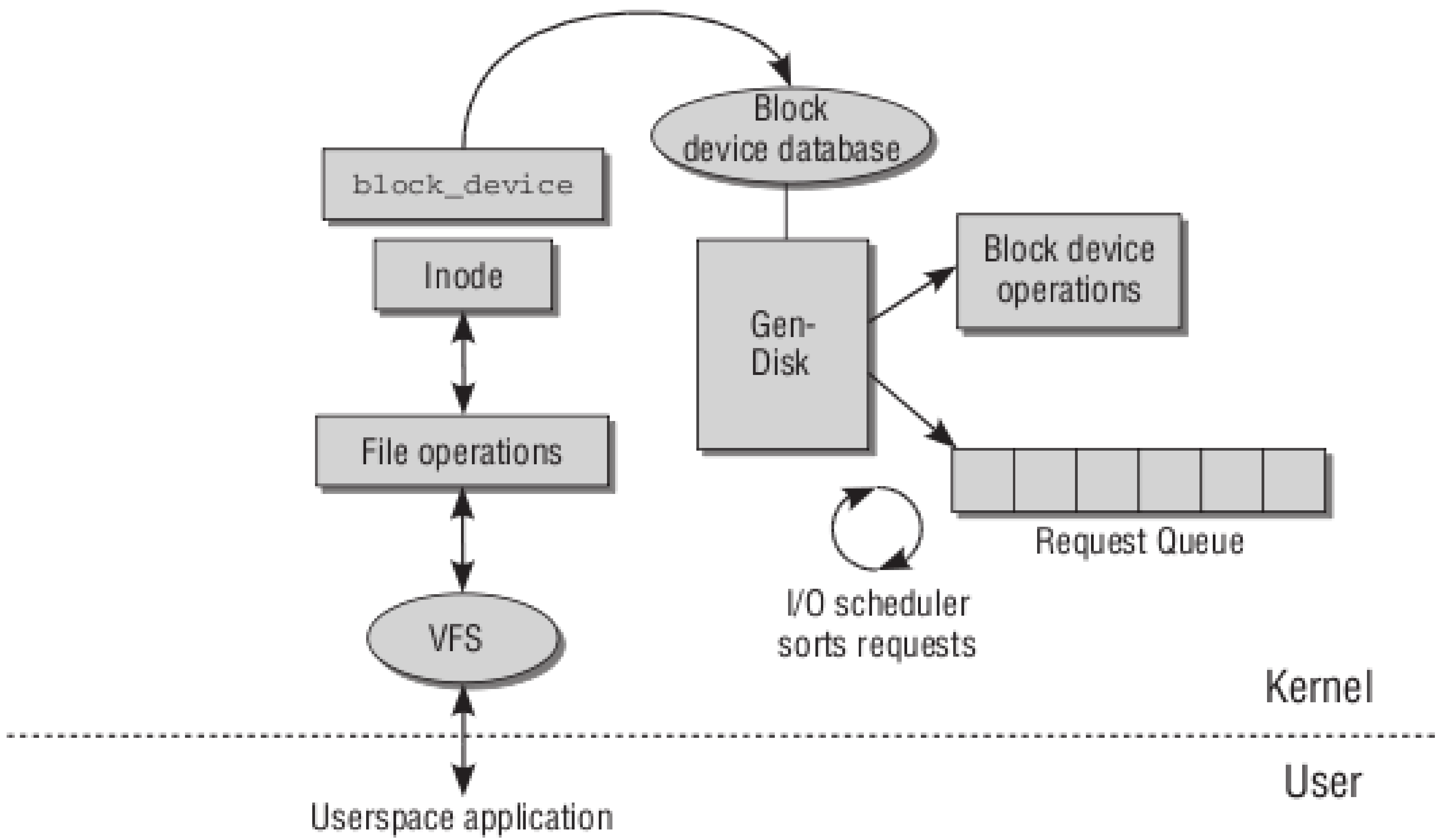
- Interactivity (i.e. desktop)
- QoS / Pay-per-use services
- Prevent resource hog tasks (typically in hosted environments the cause of slowness are due to the abuse of a single task / user)
- Reduce power-consumption
- More deterministic performance (real-time)

Linux I/O subsystem (overview)

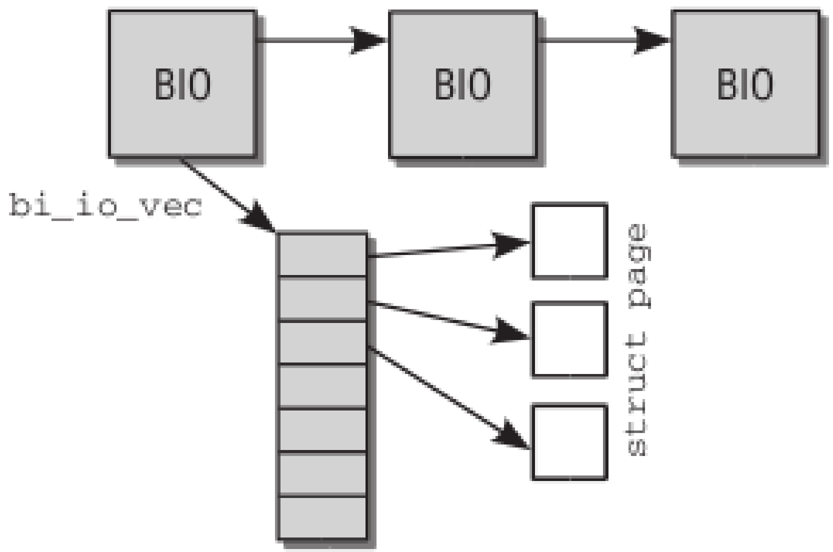


- Processes submit I/O requests using one (or more) queues
- The block I/O layer saves the context of the process that submit the request
- Requests can be merged and reordered by the I/O scheduler
  - Minimize disk seeks, optimize performance, provide fairness among processes

Block device I/O in Linux



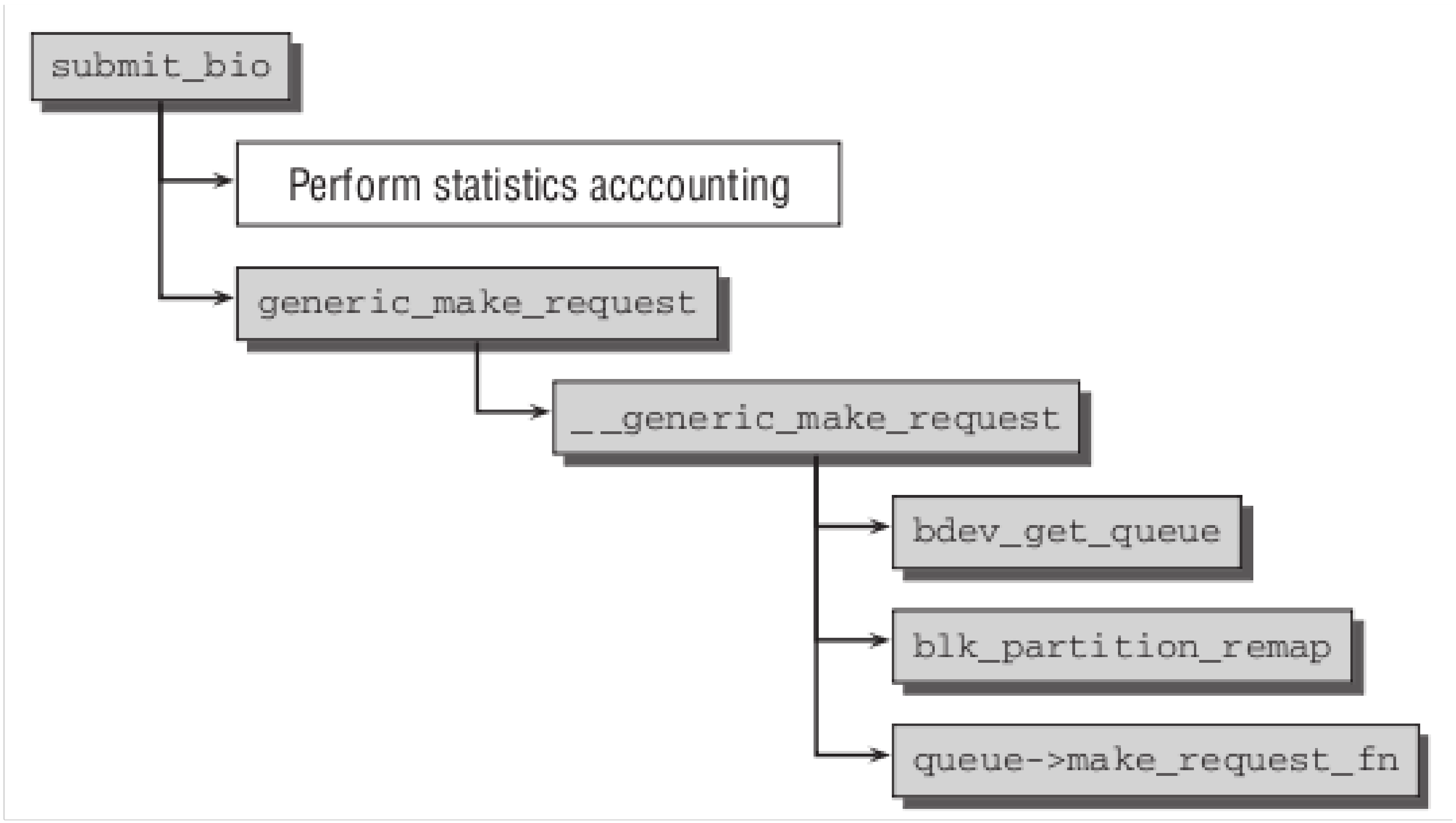
BIOs



- The kernel submits I/O requests in two steps
  - Create a *bio* instance to describe the request placed on a request queue (the *bio* points to the pages in memory involed in the I/O operation)
  - Process the request queue and carries out the actions described by the *bio*

Submit I/O requests: code flow

Kernel (high-level layers)



Elevator (elv_merge, plug/unplug queue, ...)

I/O scheduler (noop, deadline, anticipatory, CFQ)

Dispatch I/O requests

```
struct elevator_ops
{
        elevator_merge_fn *elevator_merge_fn;
        elevator_merged_fn *elevator_merged_fn;
        elevator_merge_req_fn *elevator_merge_req_fn;

        elevator_dispatch_fn *elevator_dispatch_fn;
        elevator_add_req_fn *elevator_add_req_fn;
        elevator_activate_req_fn *elevator_activate_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;

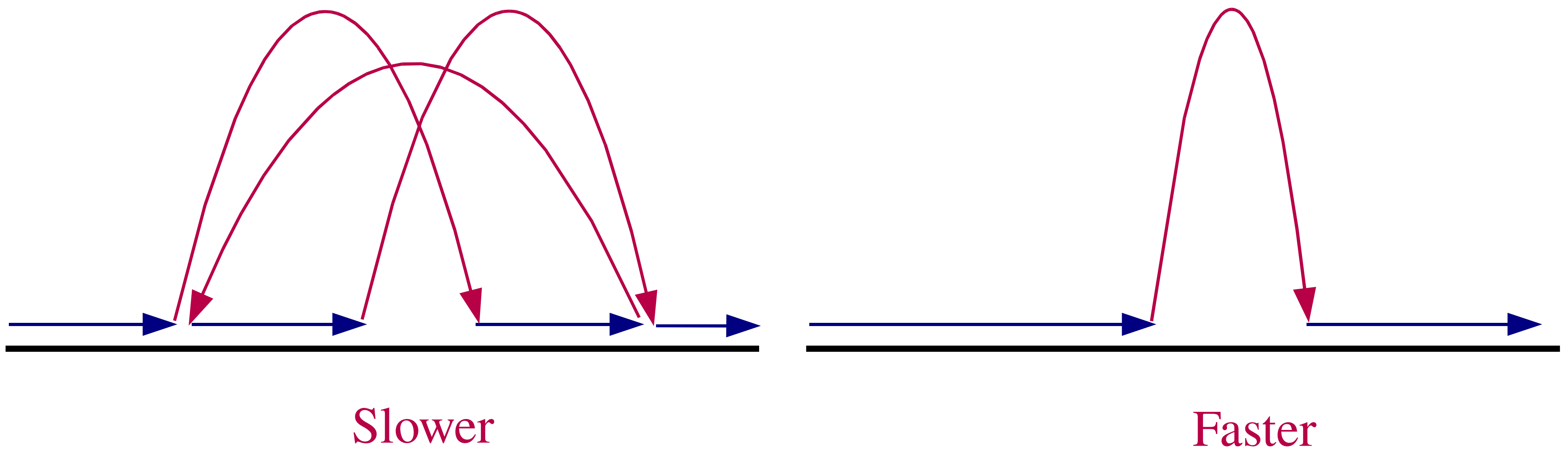
        elevator_may_queue_fn *elevator_may_queue_fn;

        elevator_init_fn *elevator_init_fn;
        elevator_exit_fn *elevator_exit_fn;
};
```

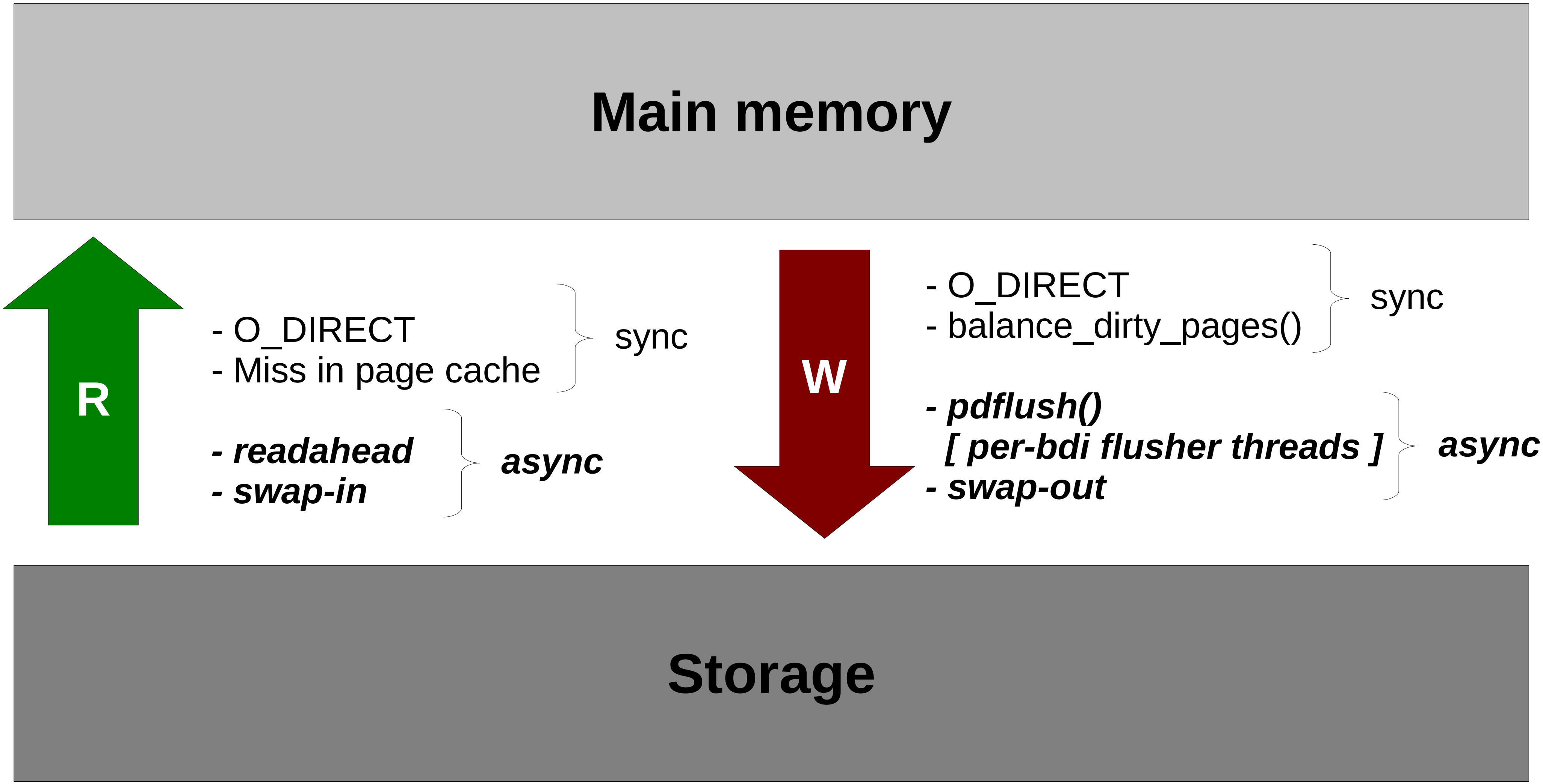
- I/O schedulers: complete management of the request queues (merge + reordering)
- Available I/O schedulers:
  - noop (FIFO)
  - Deadline
  - Anticipatory
  - CFQ

I/O schedulers

- Mission of I/O schedulers: re-order reads and writes to disk to minimize head movements

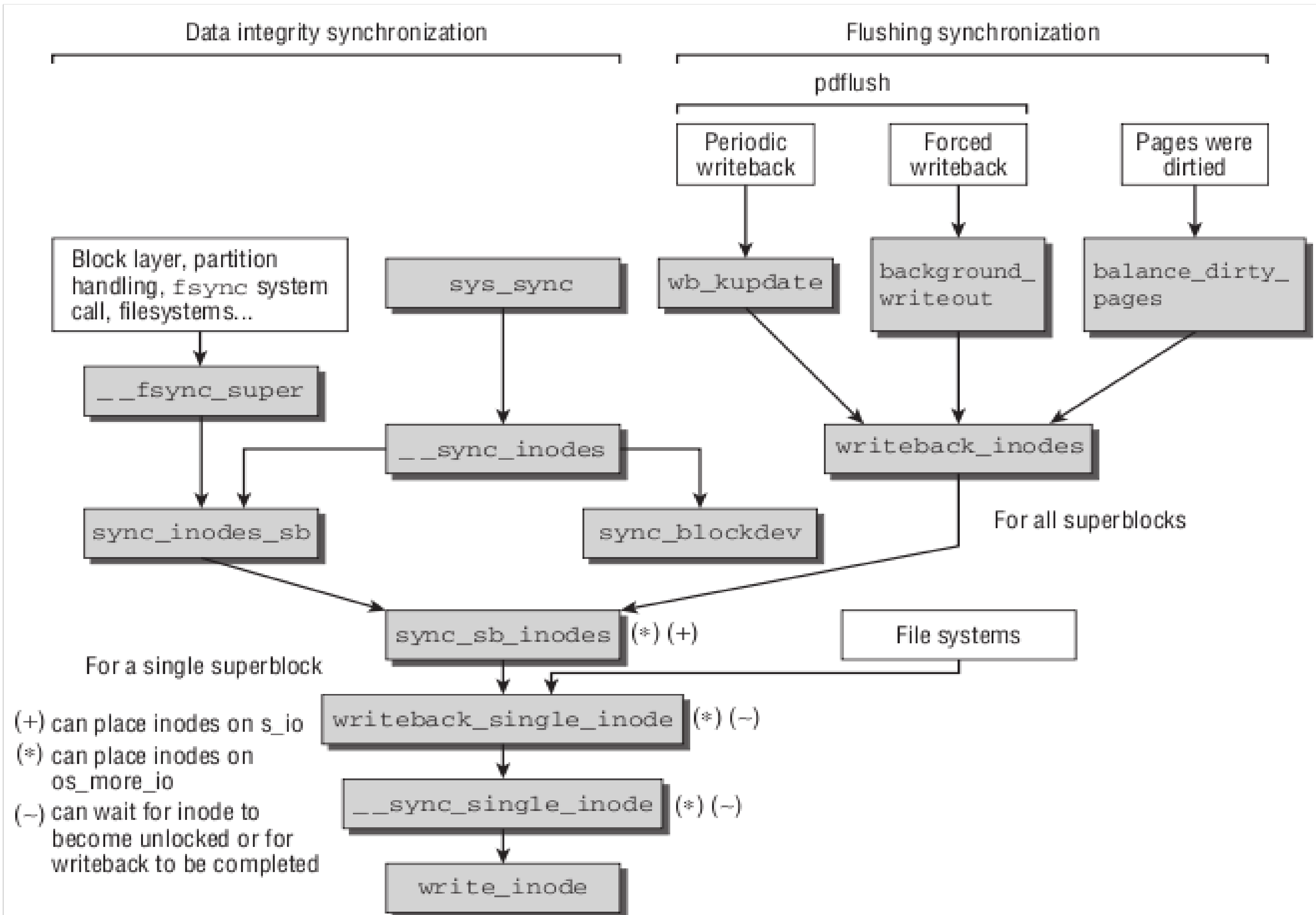


Memory pages <-> disk blocks



sync = same I/O context of the userspace task
async = different I/O context

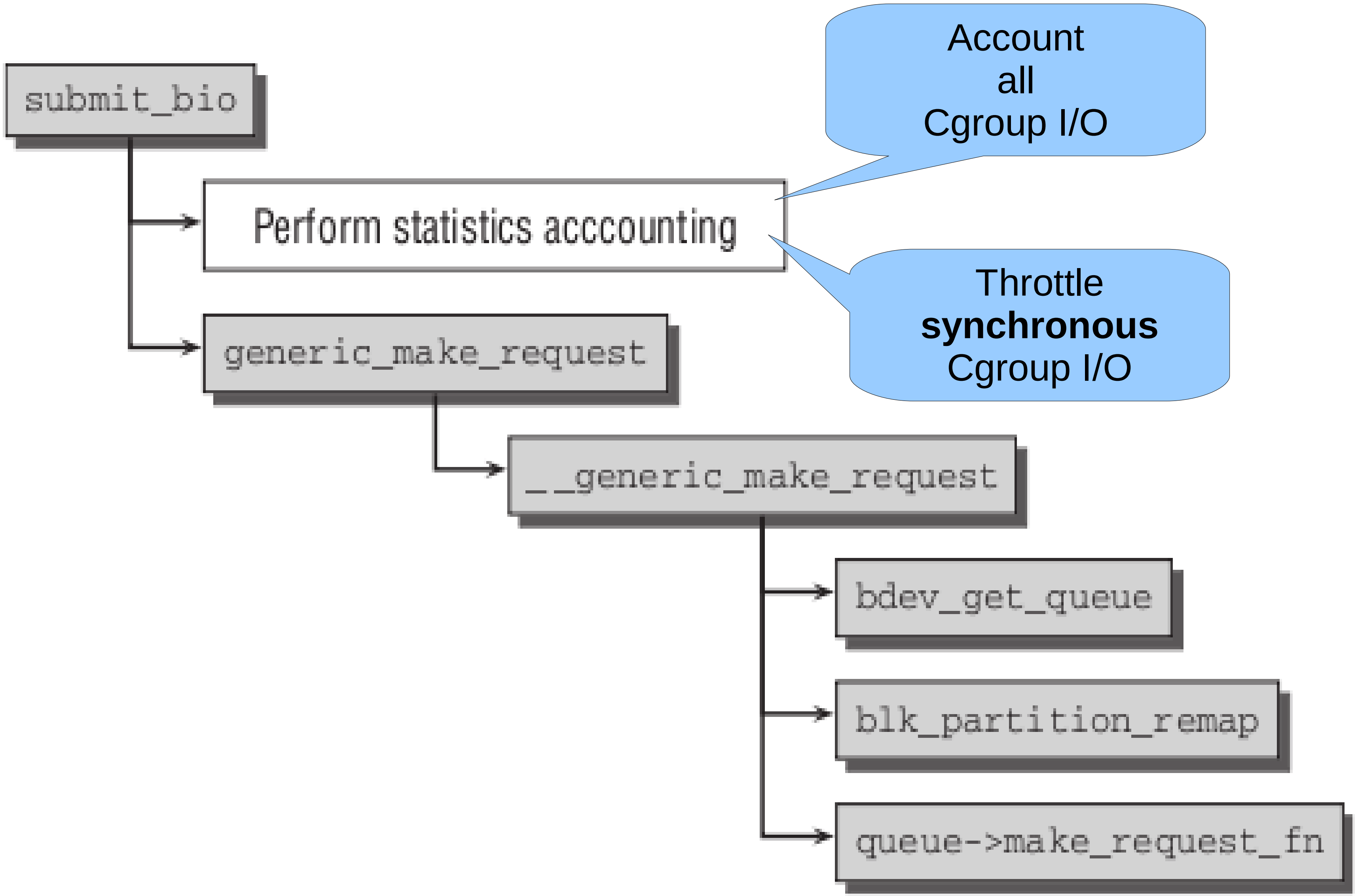
Data synchronization



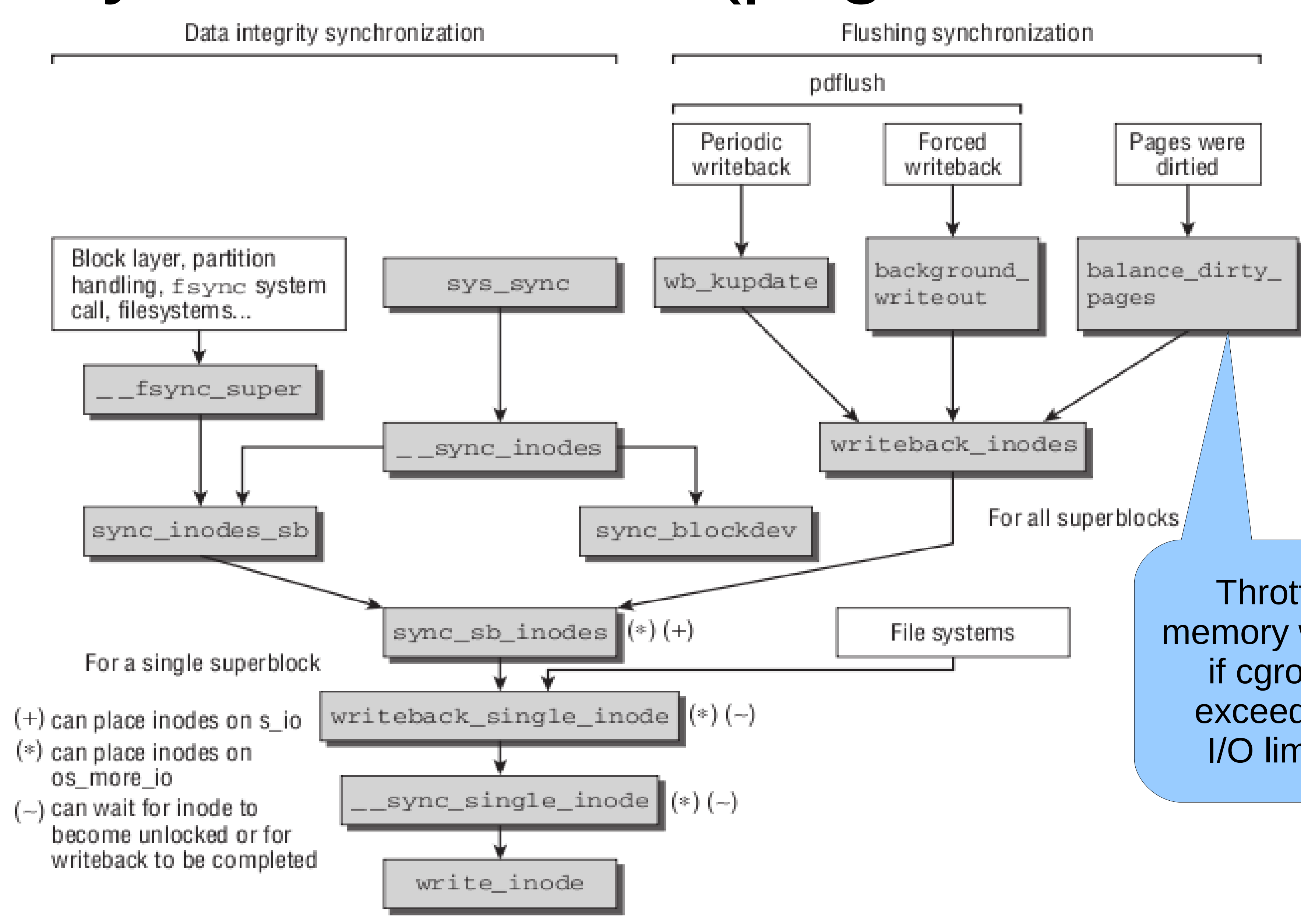
How does io-throttle work?

- Two type of I/O:
  - Synchronous I/O (O_DIRECT + read)
  - Asynchronous I/O (writeback)
- Two stages:
  - I/O accounting (sensor)
  - I/O throttling (actuator)

Synchronous I/O



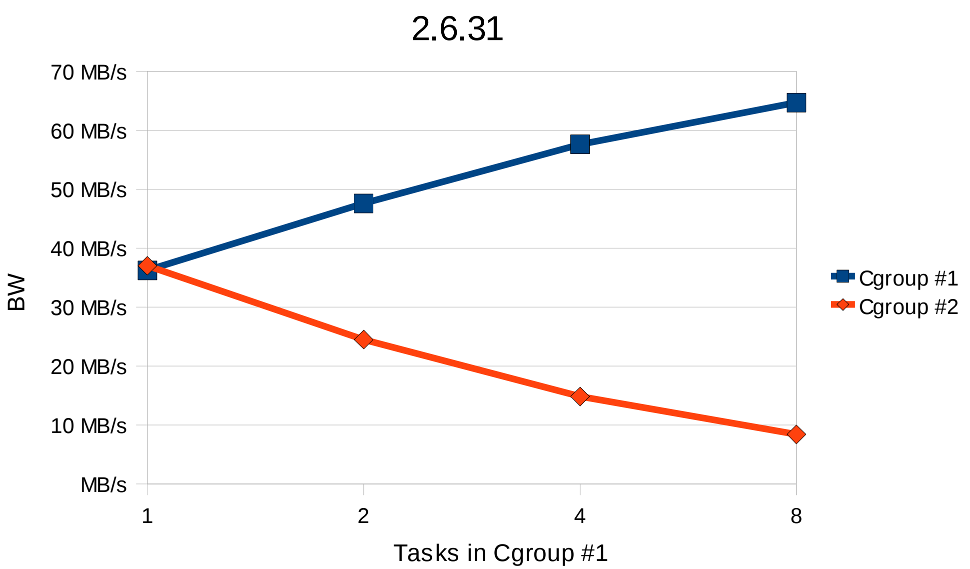
Asynchronous I/O (page writeback)

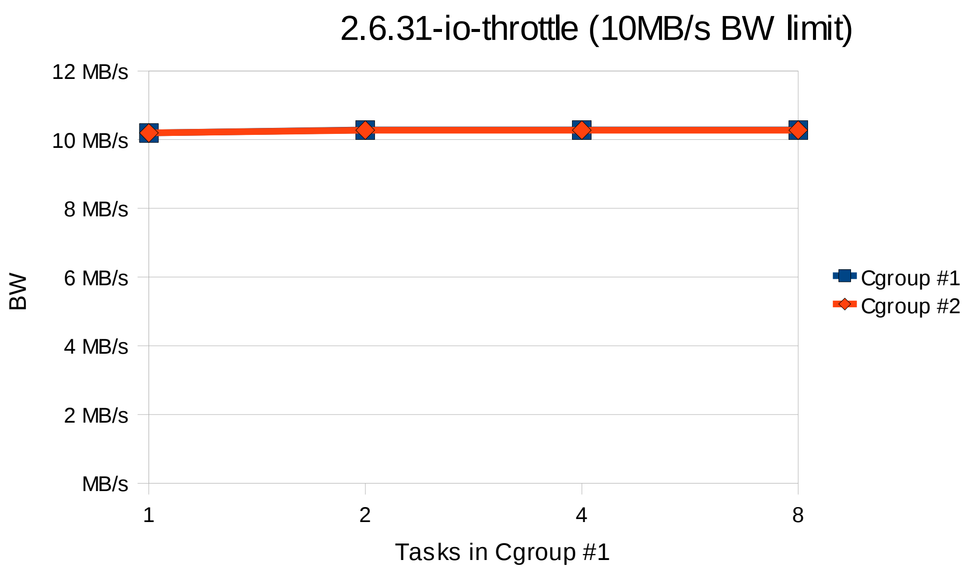


Throttle memory writes if cgroup exceeded I/O limits

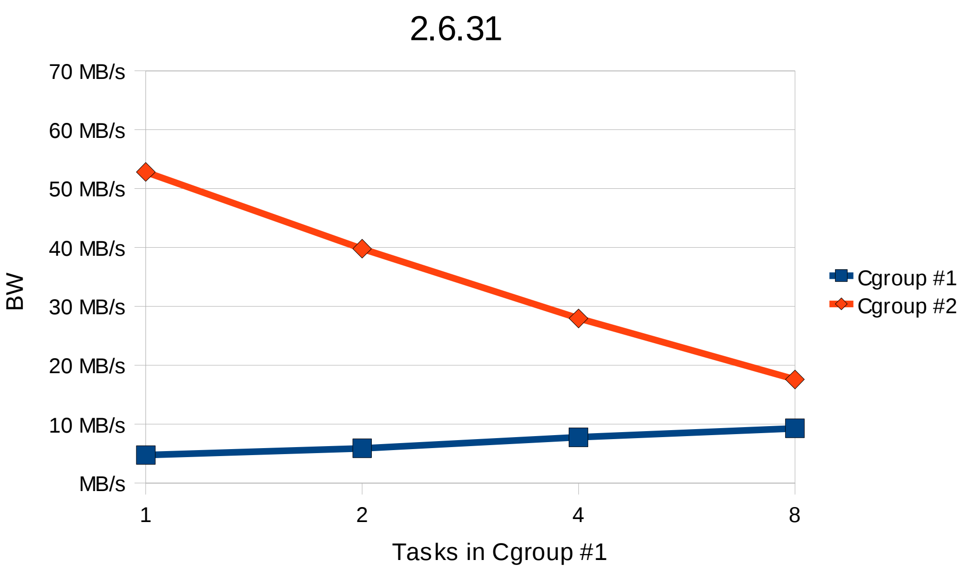
Some numbers

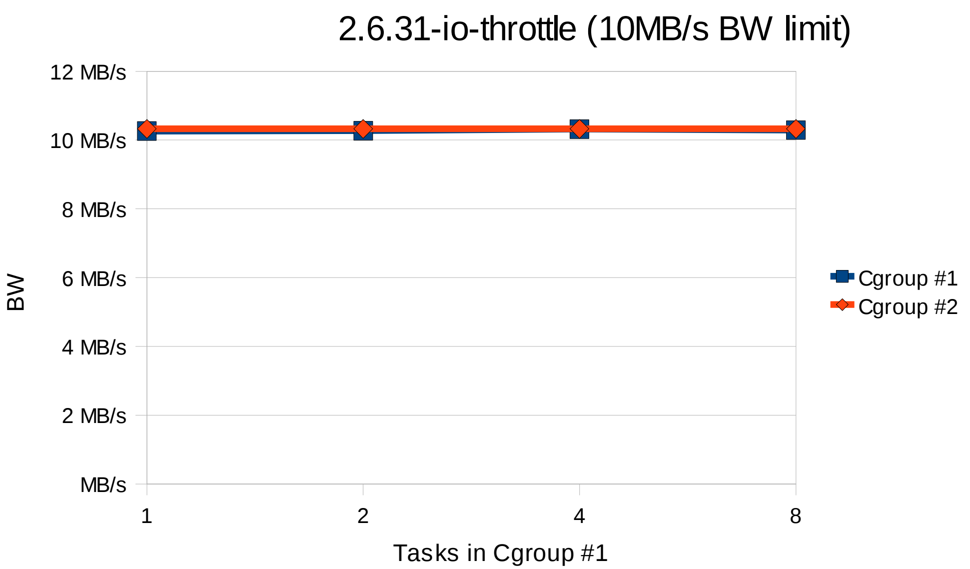
Sequential-readers (cgroup #1) VS Sequential-reader (cgroup #2)



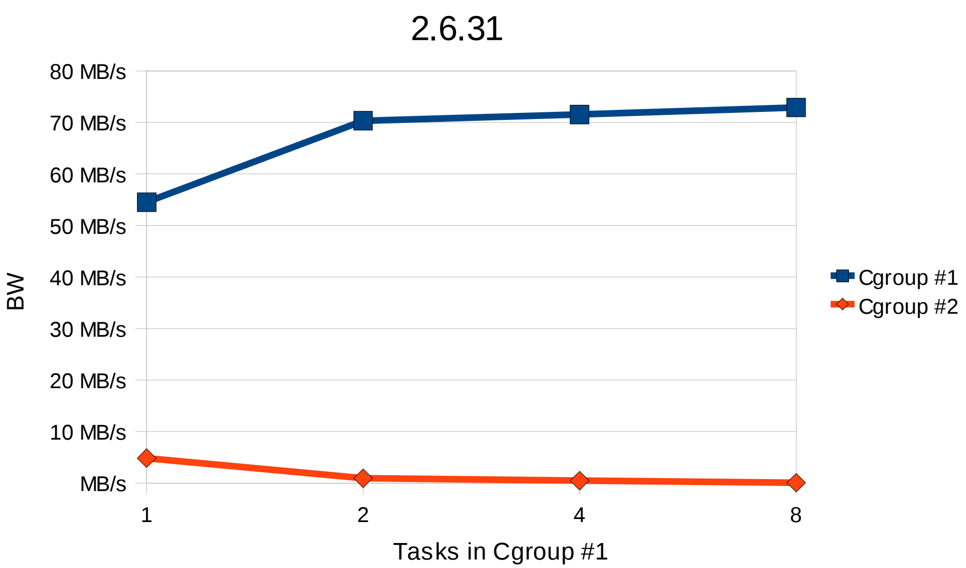


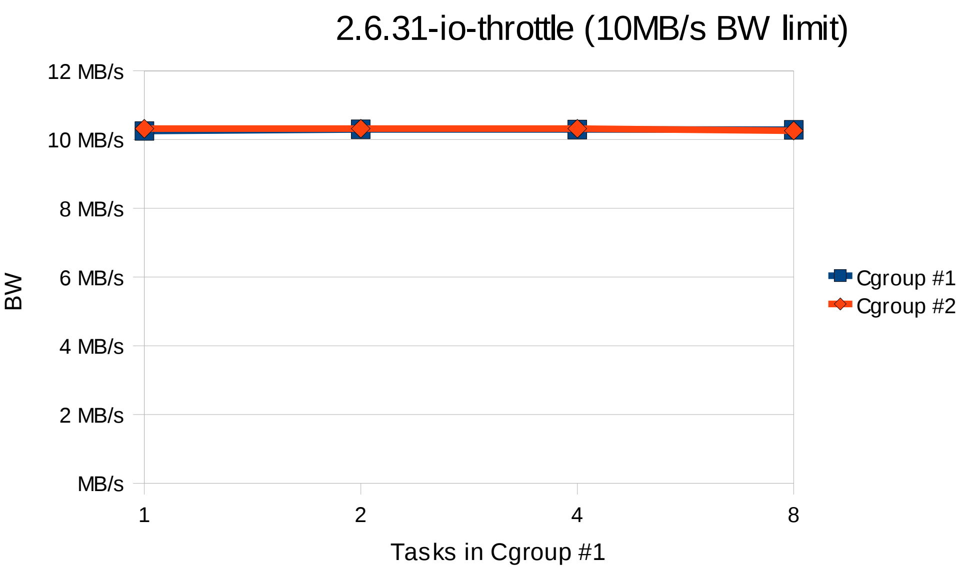
Random-readers (cgroup #1) VS Sequential-reader (cgroup #2)



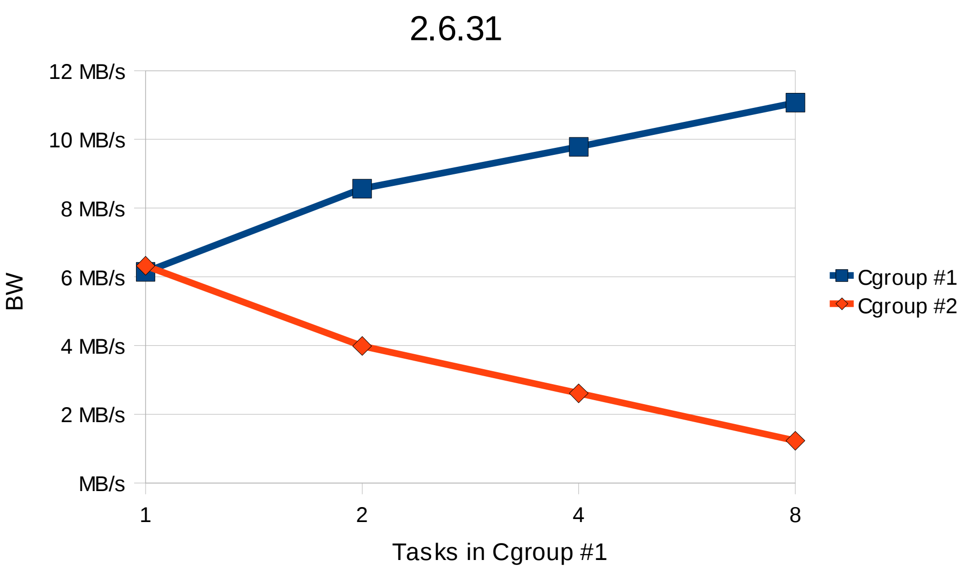


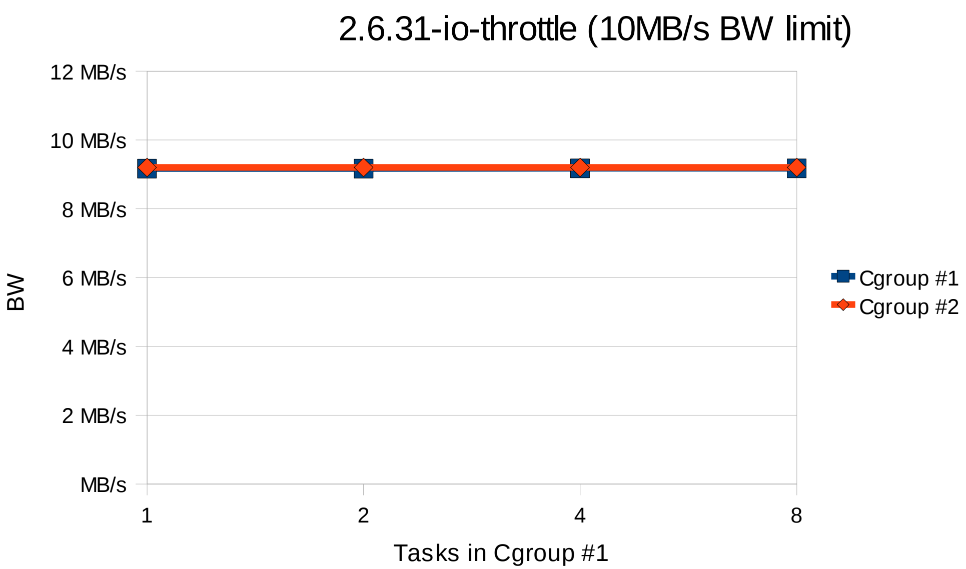
Sequential-readers (cgroup #1) VS Random-reader (cgroup #2)





Random-readers (cgroup #1) VS Random-reader (cgroup #2)





Conclusion

- cgroup framework
  - Put processes in logical containers
- cgroup subsystem
  - Resource accounting and control

- *Advantages*: lightweight isolation, simplicity
- *Disadvantages*: no way to run different kernels/ OS (like a real virtualization solution)

References

- Linux cgroups documentation
  - http://lxr.linux.no/#linux+v2.6.31/Documentation/cgroups/
- Which I/O controller is the fairest of them all?
  - http://lwn.net/Articles/332839/
- cgroup: io-throttle controller (v16)
  - http://thread.gmane.org/gmane.linux.kernel/831329
- io-throttle patchset
  - http://www.develer.com/~arighi/linux/patches/io-throttle/
- For any other question:
  - mailto:righi.andrea@gmail.com

Appendix: How to write your own cgroup subsystem?

- Basically we need to change the following files:
  - **init/Kconfig**: kernel configuration parameters (general setup)
  - **include/linux/cgroup_subsys.h**: cgroup subsystem definition
  - **kernel/cgroup_example.c**: cgroup subsystem implementation
  - **kernel/Makefile**: Makefile of the core kernel components
  - Finally, add the appropriate *hooks* into the kernel

Questions?