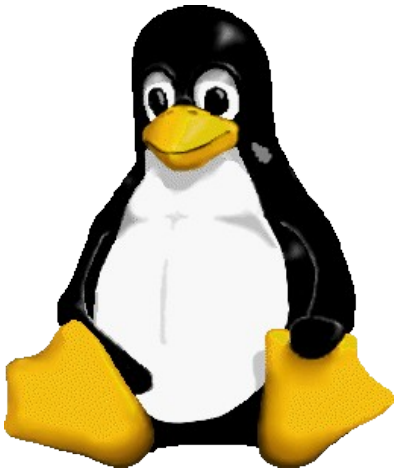
# Conoscere e ottimizzare l'I/O su Linux

betterembedded2014



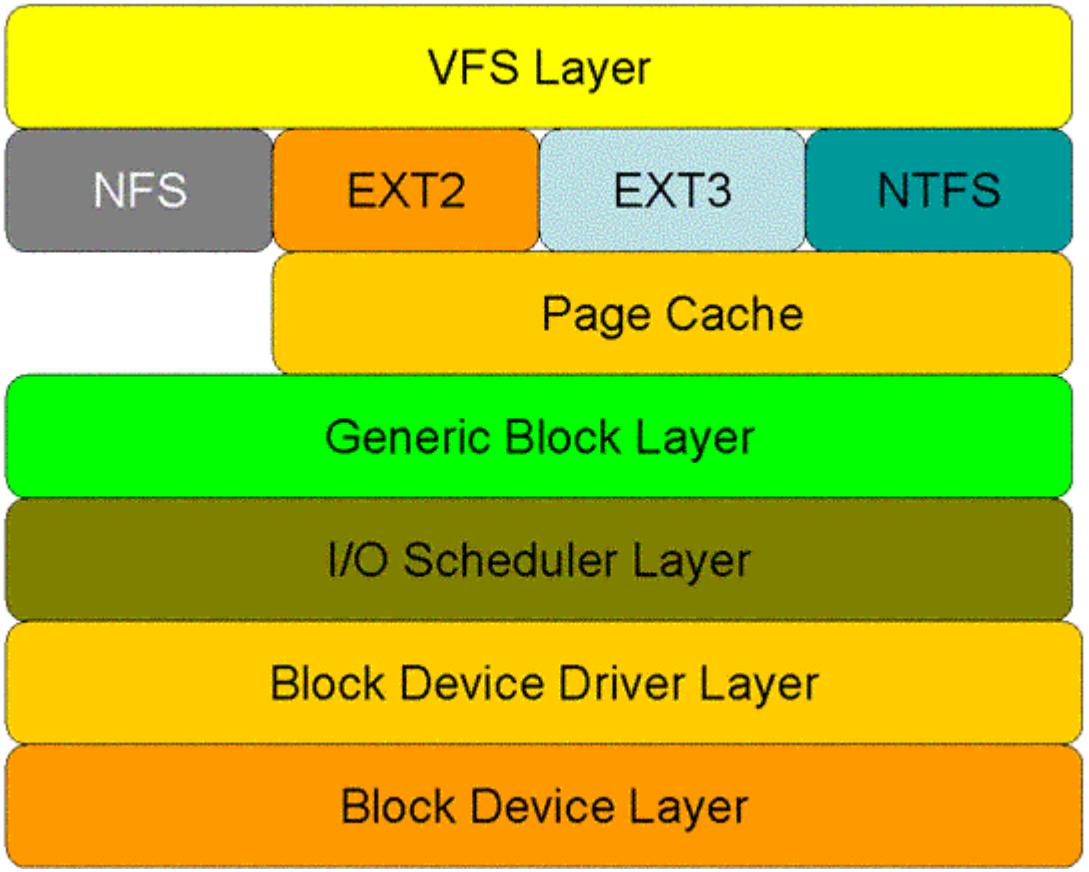
Andrea Righi - andrea@betterlinux.com

# Agenda

- Overview
- I/O Monitoring
- I/O Tuning
- Reliability
- Q/A

# Overview

# File I/O in Linux




Andrea Righi - andrea@betterlinux.com

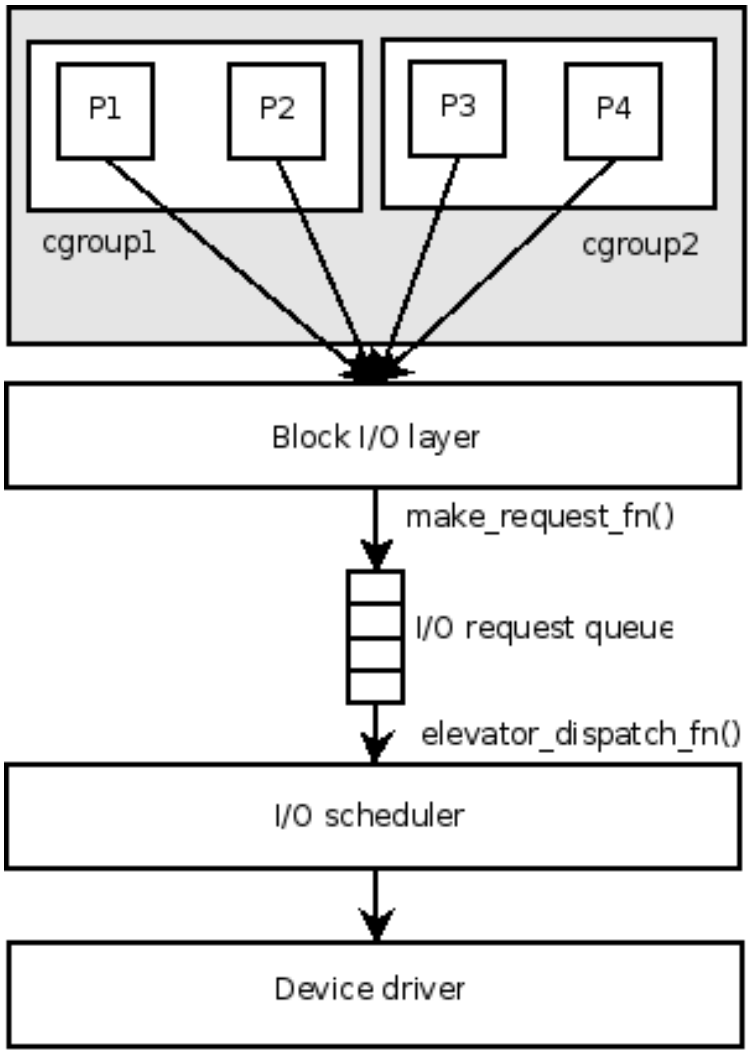
# READ vs WRITE

- READ
  - synchronous: CPU needs to wait the completion of the READ to continue
  - cached pages are easy to reclaim
- WRITE
  - asynchronous: CPU doesn't need to wait the completion of the WRITE to continue
  - cached pages are hard to reclaim (require I/O)

# SYNC vs ASYNC

- SYNC I/O READ: kernel queues a read operation for the data and returns only after the entire block of data is read back, process is in waiting for I/O state (D)
- SYNC I/O WRITE: kernel queues a write operation for the data and returns only after the entire block of data is written, process is in waiting for I/O
- ASYNC I/O READ: process repeatedly call read() with the size of the data remaning, until the entire block is read (use select()/poll() to determine when some data is available)
- ASYNC I/O WRITE: kernel updates the corresponding pages in page-cache and marks them dirty; then the control quickly returns to the process which can continue to run; the data is flushed later from a different context in more optimal ways (i.e., sequential vs seeky blocks)

# Block I/O subsystem (simplified view)




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

# Plug / unplug

- When I/O is queued to a device that device enters a plugged state
- I/O isn't immediately dispatched to the low-level device driver
- When a process is going to wait on the I/O to finish, the device is unplugged
- Allow merging of sequenial requests (writing and reading bigger chunks of data allows to save re-writes of the same hardware blocks and improves I/O throughput)

# Flash memory

- Limited amount of erase cycles
- Flash memory blocks have to be explicitly erased before they can be written to
- Writes decrease flash memory lifetime
- Wear leveling: logical mapping to distribute writes evenly among the available physical blocks

# I/O Monitoring

Andrea Righi - andrea@betterlinux.com

# iostat

- Informations about request queues associated with specific block devices
  - Number of blocks read/written, average I/O wait time, disk utilization %, ...
- It does not provide detailed informations per-I/O based (pid? uid? ...)

# iotop

- top-like I/O monitoring tool
- Disk read, write, I/O wait time percentage
- Still does not provide enough informations on a per-I/O basis:
  - per block device statistics are missing
  - no statistics about the nature of each request

# blktrace

- Low-overhead monitoring tool
- detailed per user / cgroup / thread and block device statistics
- allow to trace events for specific operations performed on each I/O entering the block I/O layer

# blktrace events

- Request queue entry allocated
- Sleep during request queue allocation
- Request queue insertion
- Front/back merge
- Re-queue of a request
- Request issued to underlying block device
- Request queue plug/unplug
- I/O remap (DM / MD)
- I/O split/bounce operation
- Request completed
- ...

# blktrace operations

- RWBS
  - 'R' - read
  - 'W' - write
  - 'D' - discard
  - 'B' - barrier
  - 'A' - ahead
  - 'S' - synchronous
  - 'M' - meta-data
  - 'N' - No data

```
static void fill_rwbs(char *rwbs, struct blk_io_trace *t)
{
        int i = 0;

        if (t->action & BLK_TC_DISCARD)    rwbs[i++] = 'D';
        else if (t->action & BLK_TC_WRITE) rwbs[i++] = 'W';
        else if (t->bytes)            rwbs[i++] = 'R';
        else                       rwbs[i++] = 'N';

        if (t->action & BLK_TC_AHEAD)      rwbs[i++] = 'A';
        if (t->action & BLK_TC_BARRIER)    rwbs[i++] = 'B';
        if (t->action & BLK_TC_SYNC)       rwbs[i++] = 'S';
        if (t->action & BLK_TC_META)       rwbs[i++] = 'M';

        rwbs[i] = '\0';
}
```

# blktrace actions

- Actions
  - C -- complete
  - D -- issued
  - I – inserted
  - Q -- queued
  - B -- bounced
  - M – back merge
  - F -- front merge
  - G -- get request
  - S -- sleep
  - P -- plug
  - U -- unplug
  - T -- unplug due to timer
  - X -- split
  - A -- remap
  - m -- message

# blktrace output

```
 # btrace /dev/sda
 ...
 8,0    1       26     0.054596889   228  Q  WS 237891152 + 8 [jbd2/sda3-8]
 8,0    1       27     0.054597204   228  M  WS 237891152 + 8 [jbd2/sda3-8]
 8,0    1       28     0.054597816   228  A  WS 237891160 + 8 <- (8,3) 230983256
 8,0    1       29     0.054598137   228  Q  WS 237891160 + 8 [jbd2/sda3-8]
 8,0    1       30     0.054598457   228  M  WS 237891160 + 8 [jbd2/sda3-8]
 8,0    1       31     0.054599094   228  A  WS 237891168 + 8 <- (8,3) 230983264
 8,0    1       32     0.054599399   228  Q  WS 237891168 + 8 [jbd2/sda3-8]
 8,0    1       33     0.054599725   228  M  WS 237891168 + 8 [jbd2/sda3-8]

Device, CPU, seq.num., timestamp,    PID, Action, RWBS, Start block + # of blocks, PID
```

# I/O Tuning

Andrea Righi - andrea@betterlinux.com

# Dirty pages writeback

- Writeback is the process of writing pages back to persistent storage
- Dirty pages (grep Dirty /proc/meminfo)
- Slow down tasks that are creating more dirty pages than the system can handle balance_dirty_pages()
  - direct reclaim (bad I/O pattern)
  - pause
  - IO-less dirty throttling (>= 3.2)
- pdflush vs per backing device writeback (>= 2.6.32)

# Background vs direct cleaning

- From Documentation/sysctl/vm.txt:
  - Background cleaning (kernel flusher threads):
    - /proc/sys/vm/dirty_background_ratio
    - /proc/sys/vm/dirty_background_bytes
  - Direct cleaning (normal tasks generating disk writes):
    - /proc/sys/vm/dirty_ratio
    - /proc/sys/vm/dirty_bytes

# Flusher thread tuning

- /proc/sys/vm/dirty_writeback_centisecs
  - Wake up kernel flusher threads every dirty_writeback_centisecs
- /proc/sys/vm/dirty_expire_centisecs
  - Define when dirty data is old enough to be eligible for writeout by kernel flusher threads

# Swap I/O

- /proc/sys/vm/swappiness
  - anonymous vs file LRU scanning ratio
    - high value: aggressive swap
    - low value: aggressive file pages reclaim

# Filesystem I/O

- ext3: data=journal | ordered | writeback
  - journal: meta-data + data committed in the journal
  - ordered: data committed before its meta-data
  - writeback: meta-data and data committed out-of-order
- ext4: delayed allocation
  - block allocation deferred until background writeback
  - improve chances of using contiguous blocks (threads writing at different offsets simultaneously)
  - xfs, ext4, zfs, …
  - zero-length file problem:
    - open-write-close-rename

# Filesystem I/O tuning

- noatime, nodiratime:
  - do not update inode access times
- relatime:
  - access time is updated if the previous access time was earlier than the current modify or change time (doesn't break applications like mutt that needs to know if a file has been read since the last time it was modified)
- commit=N
  - sync data and meta-data every N seconds (default = 5s)

# I/O tuning at different layers

- Applications
  - LD_PRELOAD
- VM
  - caching
- Filesystem
  - mount options / filesystem tuning
- Block device
  - caching

# Reliability

Andrea Righi - andrea@betterlinux.com

# I/O data flow

- Application to library buffer
  - fwrite(), fprintf(), etc.
- Library to OS buffer
  - write()
- OS buffer to disk
  - paged out, periodic flush (5 sec usually)
  - fsync(), fdatasync(), sync(), sync_file_range()

# Simple use case

- User hits “Save” in Word Processor
  - Expects that data to be on disk when saved
- If power goes out
  - The last saved version of my data is there
  - If there isn't an explicit save, some recent version of my data should be okay

# Buggy implementation

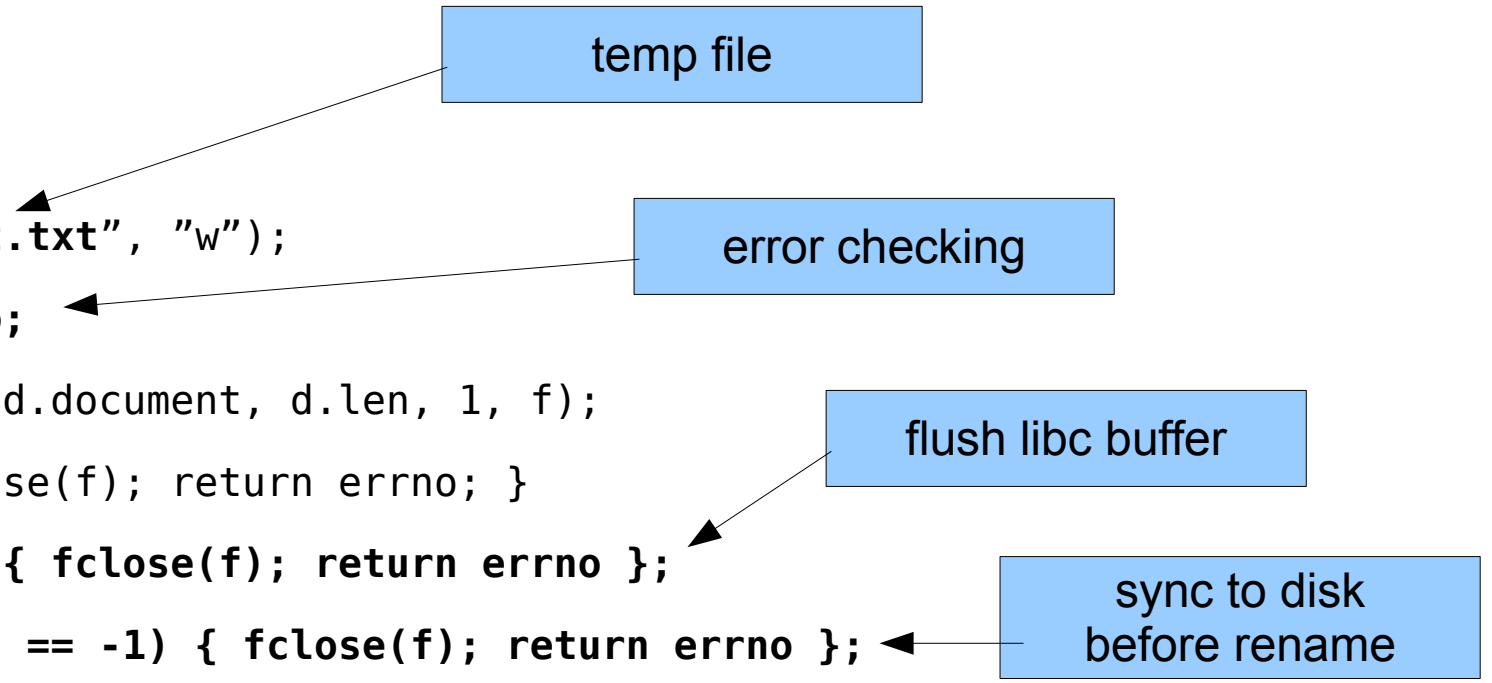
```
struct wp_doc {
   char *document;
   size_t len;
};
struct wp_doc d;
...
FILE *f;
f = fopen(“document.txt”, ”w”);
fwrite(d.document, d.len, 1, f);
fclose(f);
```

# Bugs

- No error checking
  - fopen (did we open the file?)
  - fwrite (did we write the entire file?)
- Crash in the middle of fwrite()
  - document corrupted
- No sync
  - close **does not** imply sync()!

# Reliable implementation

```
struct wp_doc {
    char *document;
    size_t len;
};
struct wp_doc d;
...
FILE *f;
size_t len;
f = fopen(“.document.txt”, ”w”);
if (!f) return errno;
size_t len = fwrite(d.document, d.len, 1, f);
if (len != 1) { fclose(f); return errno; }
if (fflush(f) != 0) { fclose(f); return errno };
if (fsync(fileno(f)) == -1) { fclose(f); return errno };
fclose(f);
rename(“.document.txt”, ”document.txt”);
```



# References

- Block I/O layer tracing - blktrace:
  http://www.mimuw.edu.pl/~lichota/09-10/Optymalizacja-open-source/Materiały/10%20-%20Dysk/gelato_ICE06apr_blktrace_brunelle_hp.pdf
- Eat my data:
  http://www.flamingspork.com/talks/2007/06/eat_my_data.odp
- fsync() problems with Firefox:
  http://shaver.off.net/diary/2008/05/25/fsyncers-and-curveballs/
- Linux documentation
  - Documentation/sysctl/vm.txt
  - Documentation/laptops/laptop-mode.txt

# Q/A

- You're very welcome!
- Twitter
  - @arighi
  - #bem2014