Model Checking Linux's Virtual File System

Joint work by Andy Galloway, Jan Tobias Mühlberg,

Radu Siminiceanu and Gerald Lüttgen

Jan Tobias Mühlberg
muehlber@cs.york.ac.uk

University of York, UK

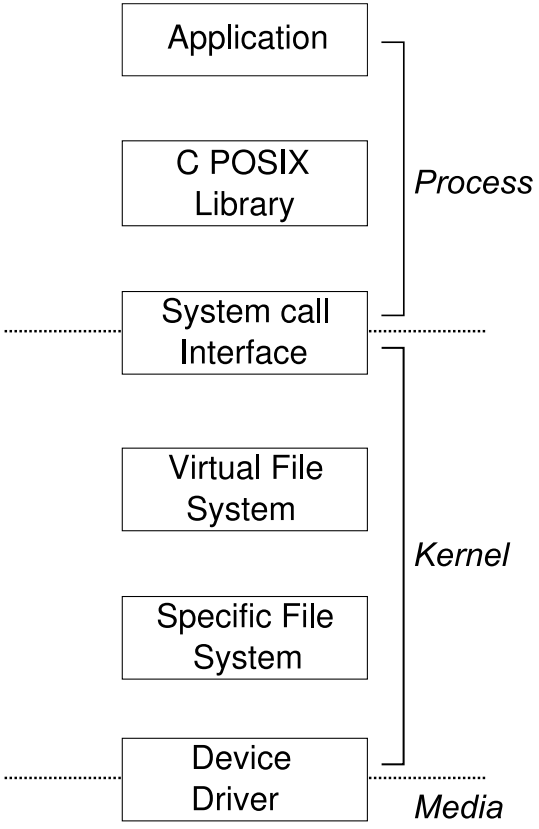
Tuesday, 18th December 2007

Aims

- Produce a large case study for measuring the performance of the SMART (Ciardo et al., 2006) model-checker
  - State-space generation using Saturation (Ciardo et al., 2001)
- Assess feasibility of analytic program verification on part of the Linux kernel
- Find (potential) errors in implementation

The **V**irtual **F**ile **S**ystem

- Indirection layer: abstract interface allowing many specific file systems to coexist
- System calls for accessing files, caching
- Various locking mechanisms



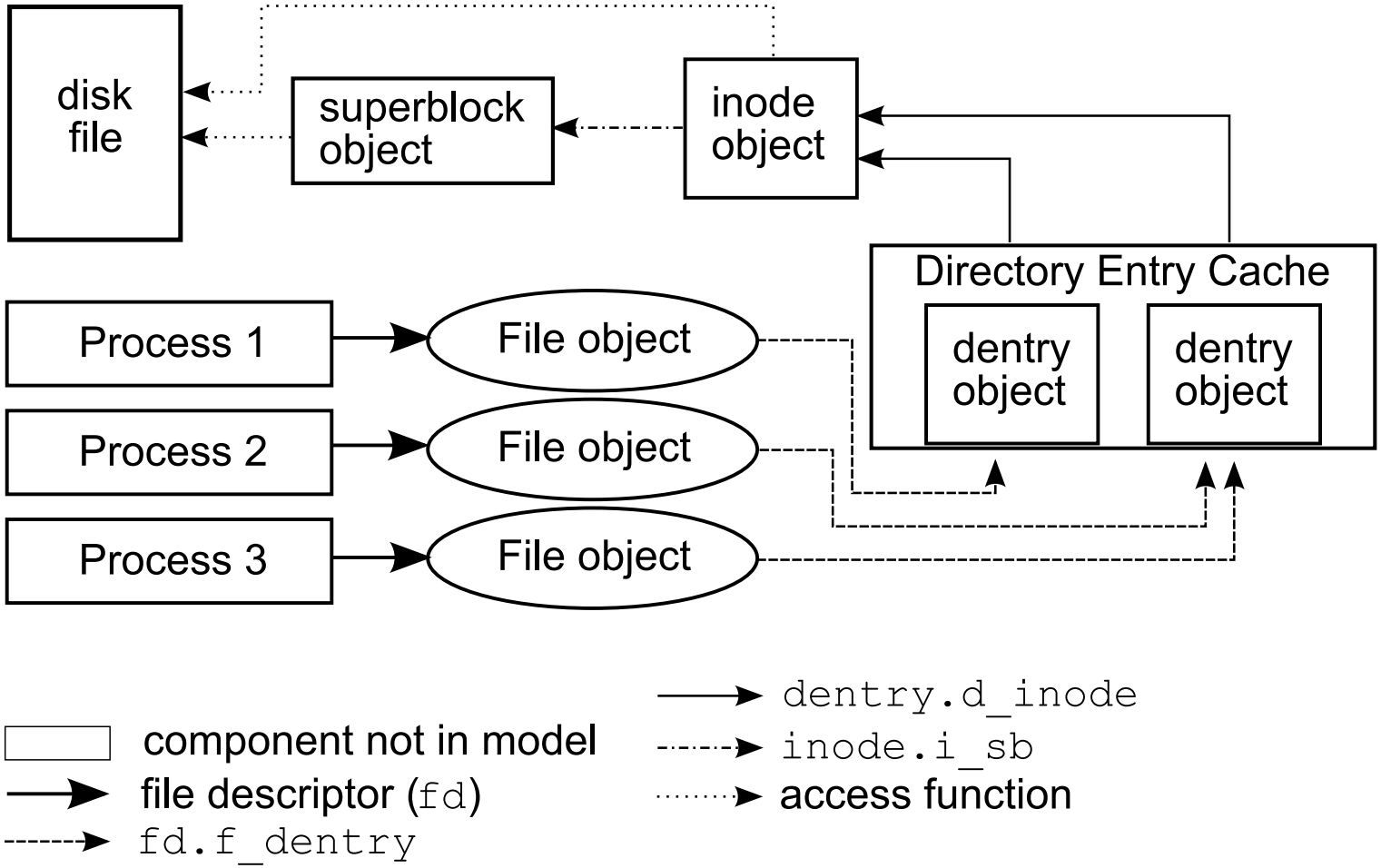
Data Structures

- `super_block`: Abstract properties (size, physical device, mount point, etc.), pointer to root `dentry`
- `dentry`: Logical structure of the file system, including file name, parent directory, children and siblings, reference counter, operations, pointer to `inode`
- `inode`: Physical properties of a file, such as size, permissions, file type, operations

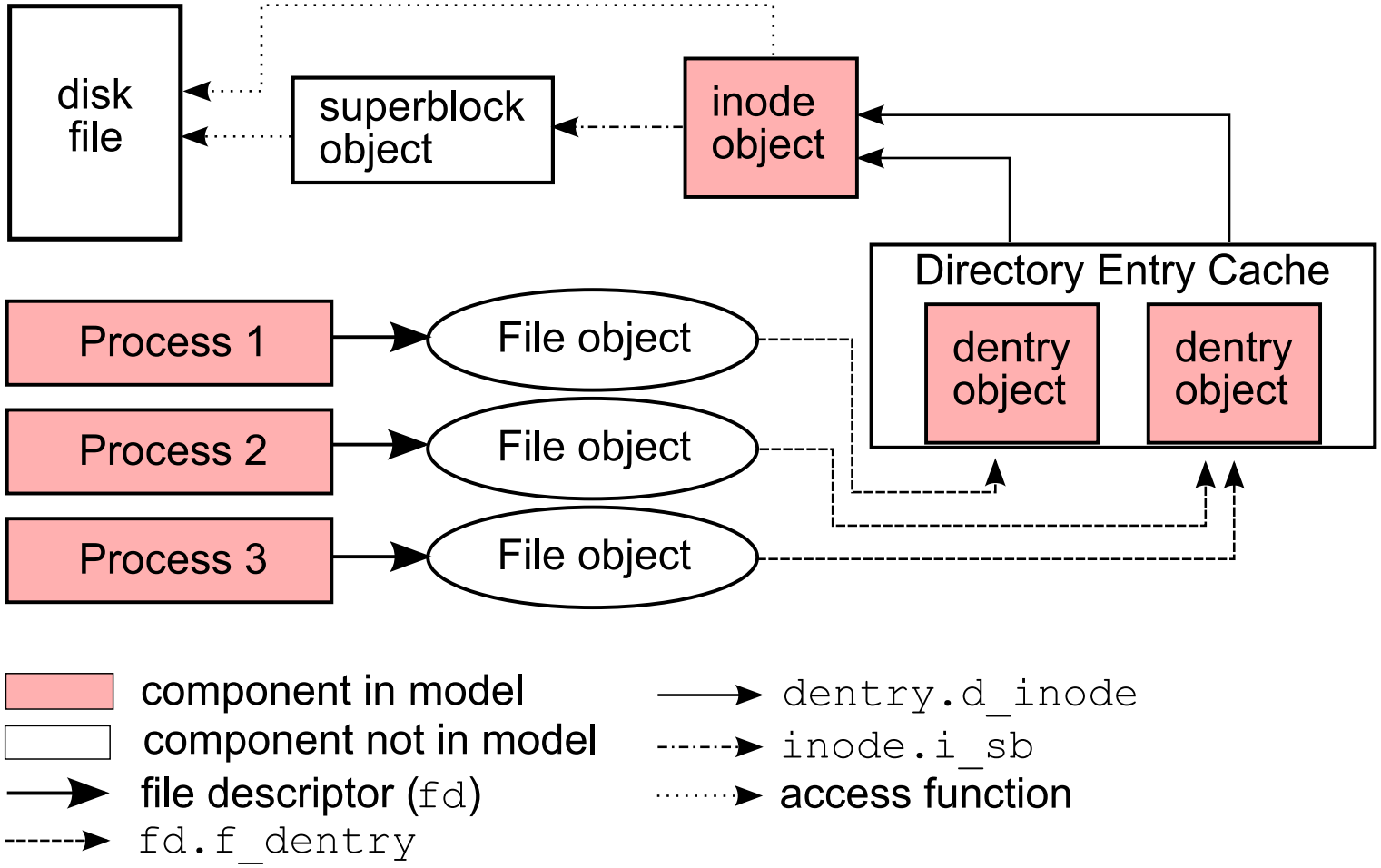
Concurrency Issues

- Arise from highly concurrent environment the VFS runs in
- Multiple processes may read or modify `dentry` and `inode` objects concurrently
- VFS employs several global and per-object locks in order to sequentialise access

Interaction Overview



Our Scope (1)


disk file
superblock object
inode object
Directory Entry Cache
dentry object
dentry object
Process 1
Process 2
Process 3
File object
File object
File object
component in model
component not in model
file descriptor (fd)
fd.f_dentry
dentry.d_inode
inode.i_sb
access function

Our Scope (2)

- Focus on generic aspects such as VFS, eventually evolve towards including specific FS and media layer
- Consider a limited number of single host processes
- Basic operations: `creat()`, `mkdir()`, `unlink()`, `rmdir()` and `rename()`
- Abstract data structures
- Small file system of 8 dentries and 8 inodes, no hard links, no symbolic links

Outline

1. Identifying key variables and structure components
2. Abstracting from the Linux code
3. Building a SPIN model
4. Building a SMART model
5. Conclusions and future work

1. Identifying Key Variables

Identifying Key Variables (1)

- Complex data structures – which parts of the logical structure and the locking mechanisms are required for our restricted model
- Which ones are interesting for verification properties

$\longrightarrow$ Manual analysis of the header files and available documentation

Identifying Key Variables (2)

```
struct dentry {
 atomic_t d_count;
 unsigned int d_flags;
 spinlock_t d_lock;
 struct inode *d_inode;

 struct hlist_node d_hash;
 struct dentry *d_parent;
 struct qstr d_name;

 struct list_head d_lru;
 union {
  struct list_head d_child;
  struct rcu_head d_rcu;
 } d_u;
 struct list_head d_subdirs;
 struct list_head d_alias;
 unsigned long d_time;
 struct dentry_operations *d_op;
 struct super_block *d_sb;
 void *d_fsdata;
#ifdef CONFIG_PROFILING
 struct dcookie_struct *d_cookie;
#endif
 int d_mounted;
 unsigned char
  d_iname[DNAME_INLINE_LEN_MIN];
};
```

Identifying Key Variables (3)

- `d_inode` and `d_parent`: 3 bits each
- `d_child` and `d_subdirs`: 8 bits each; marking dentries rather than having lists
- `d_iname`: 8 bits; 8 different names
- `d_count`: 3 bits; max. 6 operations
- `d_lock`: 3 bits; status, process id, waiting process
- `d_rcu`: 4 bits (not used)

Identifying Key Variables (4)

- Information space:

  (35 bits per dentry + 26 bits per inode) * 8

  + 9 bits for a superblock

  = 2^{497}

Identifying Key Variables (5)

- Allowing for verification of structural properties:
  - `d_inode` pointing to valid inode, `i_dentry` pointing to valid dentry
  - siblings and children are valid, `d_parent` is set correctly
- Also reference properties: `d_count`
- Also API usage rules: locking

2. Abstracting From Code

Abstracting From Code (1)

- VFS of Linux 2.6.18 consists of mainly 3 public header files (`fs.h`, `namei.h`, `dcache.h`) and 4 source files (`dcache.c`, `namei.c`, `inode.c`, `stat.c`)
- $\approx$ 70k LOC
- References many other parts of the kernel (allocation/deallocation, locking, etc.)
- References to process' context needs to be considered

Abstracting From Code (2)

- We tried Modex (Holzmann and Smith, 2002):
  - Failed parsing the code
  - It's not all ANSI C; function pointers; dynamic allocation

Abstracting From Code (3)

- We developed a pseudocode implementation of the VFS, based on our modelling decisions:
  - Generated call traces from the kernel for `mount()`, `umount()`, `creat()`, `open()`, `close()`, `unlink()`, `mkdir()`, `rmdir()`, `rename()`
  - Then manual inspection of the traces and code

Abstracting From Code (4)

```
sys_creat
|  sys_open
|  |  getname
|  |  get_unused_fd
|  |  filp_open
|  |  |  open_namei
|  |  |  |  path_lookup
|  |  |  |  |  link_path_walk
|  |  |  |  |  |  __link_path_walk
|  |  |  |  |  |  |  permission
|  |  |  |  |  |  |  do_lookup
|  |  |  |  |  |  |  |  __d_lookup
|  |  |  |  |  |  |  |  __follow_mount
|  |  |  |  |  |  |  |  |  lookup_mnt
|  |  |  |  |  |  |  |  |  dput
|  |  |  |  |  |  |  |  |  |  _atomic_dec_and_lock
|  |  |  |  |  |  |  dput
|  |  |  |  |  |  |  |  _atomic_dec_and_lock
|  |  |  |  |  |  |  permission
|  |  |  |  |  |  dput
|  |  |  |  |  |  |  _atomic_dec_and_lock
```

Abstracting From Code (5)

- Results:
  - $\approx$ 3k LOC of pseudocode
  - Tedious and error prone
  - Developed in parallel with the SPIN model; simulation and verification runs helped debugging
  - Had to abstract intent rather than implementation in several cases

SPIN Model

SPIN Model (1)

- Why SPIN?
  - Valuable basis for comparison
  - Provides vital intermediate representation of requirements; mainly due to C-like syntax of Promela
  - Maturity and ability to support compound data structures and concurrency

SPIN Model (2)

- Assertions made up about 3% of the model and helped to spot errors in the model and the pseudocode
- Use of assertions forced stylistic changes in the model
- Great benefits due to assertions, simulation, and a large set of systematic tests
- Focus on correctness of data, operations rather than concurrency

SPIN Model (3)

- Results:

Nodes :	4 dentries, 4 inodes
Approx Time :	2 minutes
Approx Memory Used :	700 MBytes
State Vector :	356 Bytes
Compressed State Vector:	39 Bytes (+12 overhead)
Compression Ratio :	13%

- Model checker quickly ran out of memory for 5 nodes

SMART Model

SMART Model (1)

- SMART:
  - Designed for logical and stochastic analysis
  - Petri nets as modelling language, not adequate for software
  - Program counters had to be introduced manually; no compound data structures, no recursion, no dynamic allocation, no lists
  - Saturation requires Kronecker consistency

SMART Model (2)

- Results (1 process):

#D	#I	# states	time(s)	mem (KBytes)
4	4	80461	14.42	16776
5	5	5604562	76.92	184571
6	5	87900191	382.34	1430972
7	4	37223248	255.54	502398
8	4	170672245	507.87	1059037

SMART Model (3)

- Results (2 processes):

#D	#I	# states	time(s)	mem (KBytes)
2	2	18934	3.43	4331
2	3	18934	4.02	4488
3	2	485587	93.28	78968
3	3	2992118	1139.61	784659

5. Conclusions and Future Work

Conclusions (good)

- Outlined experience in modelling and model checking parts of Linux's VFS implementation
- Presented a straightforward approach to abstracting and modelling complex data structures and program's control flow
- We were able to verify the resulting models for SPIN and SMART

Conclusions (not so good, 1)

- Restrictions in SMART modelling language led to less faithful model
- SPIN model was sequential, SMART model concurrent – hard to compare
- SPIN model contained lots of assertions while SMART model focused on concurrency issues such as deadlocks

Conclusions (not so good, 2)

- Verification with SPIN was restricted to 4-node model – doesn't say much about the correctness of model, pseudocode or implementation
- Manual abstraction of pseudocode makes scientific conclusions hard to infer

Future Work (1)

- We need better tools for automated abstraction:
  - Tools such as BLAST (Henzinger et al., 2002) are leading into that direction; we show in (Mühlberg and Lüttgen, 2006) that they don't deal very well with OS code yet
  - Problems: compiler and architecture specific code; multiple programming languages; function pointers; concurrency
  - But: can manual abstractions ever be faithful?

Future Work (2)

- Bring SPIN model in line with SMART model
- Add multi-process concurrency to SPIN model
- Run verification in comparable setting
- Review and extend models

Questions?

This talk is based on (Galloway et al., 2007). It's available at

http://www.cs.york.ac.uk/ftpdir/reports/YCS-2007-423.pdf.

References

Ciardo, G., Jones, R. L., Miner, A. S., and Siminiceanu, R. I.: 2006, *Perform. Eval.* **63(6)**, 578

Ciardo, G., Lüttgen, G., and Siminiceanu, R.: 2001, Saturation: An efficient iteration strategy for symbolic state-space generation, in *7th Intl. Conf, on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2001)*, Vol. 2031 of *Lecture Notes in Computer Science*, pp 328 – 342, Springer-Verlag, Genova, Italy

Galloway, A., Mühlberg, J. T., Siminiceanu, R., and Lüttgen, G.: 2007, *Model-checking Part of a Linux File System*, Technical Report YCS-2007-423, Department of Computer Science, University of York, UK

Henzinger, T. A., Jhala, R., Majumdar, R., Necula, G. C., Sutre, G., and Weimer, W.: 2002, Temporal-safety proofs for systems code, in *CAV '02: Proceedings of the 14th International Conference on Computer Aided Verification*, pp 526 – 538, Springer-Verlag, London, UK

Holzmann, G. J. and Smith, M. H.: 2002, *IEEE Trans. Softw. Eng.* **28(4)**, 364

Mühlberg, J. T. and Lüttgen, G.: 2006, Blasting linux code, in *FMICS 2006*, No. 4346 in LNCS, pp 211 – 226