Introduction to Operating Systems

COS 316 Lecture 4

Amit Levy

What is an Operating System?

What is an Operating System?

operating system
/ˈäpəˌrādiNG ˌsistəm/

noun

1. software that manages a computer’s resources for users and their applications [1]
2. provides applications with perfomant, safe and flexible access to hardware and software services

What is an Operating System?

- **Naming** - storage, devices, applications, users
- **Caching** - file caching, memory heirarchy, name translation
- **Resource Management** - shared network interfaces
- **Virtualization** - processes, virtual machines, device drivers
- **Access Control** - users, capabilities

Naming: IPC with UNIX Pipe vs Binder

Inter-Process Communication (IPC)

- Applications want to talk to each other
    - Why? Decouple responsibilities
- IPC an specific mechanism tailored for applications on same computer
    - Simple abstraction: send messages from one application to another
- Key question: how do we *name* other applications?

UNIX Pipes (a caricature)

- Parent process passes file descriptors to each child
  - Parent process knows the "name" of each end points process (just a PID)
- Pipe file descriptors are *local names*
  - A file descriptor is just a 32-bit number
  - Different namespace for each process

```
$ cat grades.csv |
    cut -d "," --output-delimeter="\t" -f 2,3,4 |
    awk '$1 == assignment1' | tee grades_assignment_1.tsv
```

Binder (a caricature)

- Android's IPC mechanism
    - See also dbus, Windows COM
- Applications expose services with unique global name
    - e.g. `edu.princeton.cos316.GradeService`
- Client applications can bind to services through special system calls

```
sp<IBinder> binder =
  defaultServiceManager()
    ->getService("edu.princeton.cos316.GradeService");
ASSERT(binder != 0);
sp<IGradeService> gs =
  interface_cast<IGradeService>(binder);
ASSERT(gs != 0);
Grade grades[] = gs.get_assignment1_grades();
do_stuff_with_grades(grades);
```

Revision: Unix Pipes vs Binder

	Unix Pipes	Android Binder
Namespace		
Discovery		

Thought Experiment 2: Unix Pipes vs Binder

Our application:

- Parses the grades database and finds all grades for a particular assignment

Need to output the result somewhere. Let the user choose!

- File on local machine or NFS, or cloud store
- Another application that processes the data further

How would we implement this with pipes? With Binder? Which is better?

Thought Experiment 2: Unix Pipes vs Binder

Our application:

- Parses the grades database and finds all grades for a particular assignment

Want to ensure output only to secure cloud storage, e.g. Dropbox folder accessible only to instructors.

How would we implement this with pipes? With Binder? Which is better?

Takeaway: Naming in Operating Systems

Choices of resource names impacts: discoverability, performance, flexibility, security.

Caching: Rethink the Sync

Reading and writing durably to disk is *very* expensive:

- ~10ms on a HDD, ~0.1 on a SSD
- Compare with 100ns for main memory

Caching: Rethink the Sync

Reading and writing durably to disk is *very* expensive:

- ~10ms on a HDD, ~0.1 on a SSD
- Compare with 100ns for main memory

```
FILE *out = fopen("squares.txt", "a");
for (int i = 0; i < 100; i++) {
  fprintf(out, "%d: %f\n", pow((double)i, 2.0));
}
```

How long does this take on a HDD? on a SSD? in a tmpfs (in memory)?

Asyncrhonous File System Semantics

Many OSs combat latency by *caching* I/O in memory and flushing to disk asynchronously

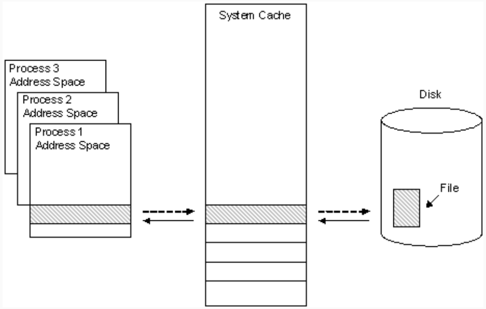


Figure 1: File Caching ([2])

How long with a buffer cache?

```c
FILE *out = fopen("squares.txt", "a");
for (int i = 0; i < 100; i++) {
  fprintf(out, "%d: %f\n", pow((double)i, 2.0));
}
```

Trading Durability for Performance

```
for (int i = 0; i < 100; i++) {
  fprintf(out, "%d: %f\n", pow((double)i, 2.0));
  printf("Finished %d of 100\n", i);
}
```

```
$ ./myprogram
Finished 1 of 100
Finished 2 of 100
...
Finished 97 of 100
# -- computer loses power here --
```

Trading Durability for Performance

`sync()` and `fsync()` guarantee durability again

```c
FILE *out = fopen("squares.txt", "a");
for (int i = 0; i < 100; i++) {
  fprintf(out, "%d: %f\n", pow((double)i, 2.0));
  sync();
  printf("Finished %d of 100\n", i);
}
```

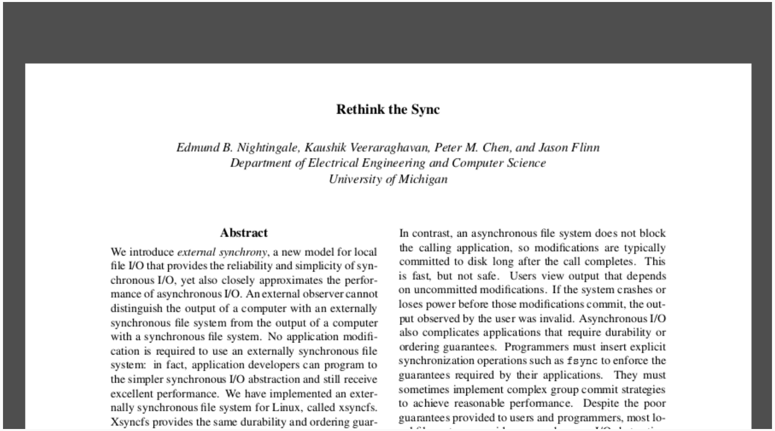
Trading Durability for Performance

`sync()` and `fsync()` guarantee durability again

```c
FILE *out = fopen("squares.txt", "a");
for (int i = 0; i < 100; i++) {
  fprintf(out, "%d: %f\n", pow((double)i, 2.0));
  sync();
  printf("Finished %d of 100\n", i);
}
```

But... performance suffers: 10ms to run on my laptop

Getting the best of both worlds: Rethink the Sync



Rethink the Sync

Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn
Department of Electrical Engineering and Computer Science
University of Michigan

Abstract

We introduce *external synchrony*, a new model for local file I/O that provides the reliability and simplicity of synchronous I/O, yet also closely approximates the performance of asynchronous I/O. An external observer cannot distinguish the output of a computer with an externally synchronous file system from the output of a computer with a synchronous file system. No application modification is required to use an externally synchronous file system: in fact, application developers can program to the simpler synchronous I/O abstraction and still receive excellent performance. We have implemented an externally synchronous file system for Linux, called xsyncfs. Xsyncfs provides the same durability and ordering guar-

In contrast, an asynchronous file system does not block the calling application, so modifications are typically committed to disk long after the call completes. This is fast, but not safe. Users view output that depends on uncommitted modifications. If the system crashes or loses power before those modifications commit, the output observed by the user was invalid. Asynchronous I/O also complicates applications that require durability or ordering guarantees. Programmers must insert explicit synchronization operations such as `fsync` to enforce the guarantees required by their applications. They must sometimes implement complex group commit strategies to achieve reasonable performance. Despite the poor guarantees provided to users and programmers, most lo-

Figure 2: Rethink the Sync, OSDI 2006

Rethink the Sync

Key Idea: Only flush to disk on *externalizable* events.

Externalizable: someone outside the process can see it happened.

Rethink the Sync

Key Idea: Only flush to disk on *externalizable* events.

Externalizable: someone outside the process can see it happened.

```
for (int i = 0; i < 100; i++) {
  fprintf(out, "%d: %f\n", pow((double)i, 2.0));
  // not "externalized"
}
```

Can flush asynchronously

Rethink the Sync

Key Idea: Only flush to disk on *externalizable* events.

Externalizable: someone outside the process can see it happened.

```
for (int i = 0; i < 100; i++) {
  fprintf(out, "%d: %f\n", pow((double)i, 2.0));
  printf("Finished %d of 100\n", i);
}
```

Have to flush on every write

Rethink the Sync

Key Idea: Only flush to disk on *externalizable* events.

Externalizable: someone outside the process can see it happened.

```
for (int i = 0; i < 100; i++) {
  fprintf(out, "%d: %f\n", pow((double)i, 2.0));
}
printf("Finished %d of 100\n", i);
```

Only have to flush at the end of the loop

Rethink “Rethink the Sync”

Why not leave caching to applications?

Takeaway: Caching in operating systems

Caching used ubiquitously to mask performance bottlenecks. Choosing how and where to apply caching can impact performance, durability, correctness, security, flexibility...

Up Next

Assignment 1

- Due TOMORROW @ 11pm

Next Time: Naming

References

[1] Anderson, T. and Dahlin, M. *Operating Systems: Principles & Practice*.

[2] *https://docs.microsoft.com/en-us/windows/win32/fileio/file-caching*.