A fork() in the road

Andrew Baumann
Microsoft Research

Jonathan Appavoo
Boston University

Orran Krieger
Boston University

Timothy Roscoe
ETH Zurich

The Open Group Base Specifications Issue 7, 2018 edition
IEEE Std 1003.1-2017 (Revision of IEEE Std 1003.1-2008)

NAME

fork - create a new process

SYNOPSIS

```
#include <unistd.h>

pid_t fork(void);
```

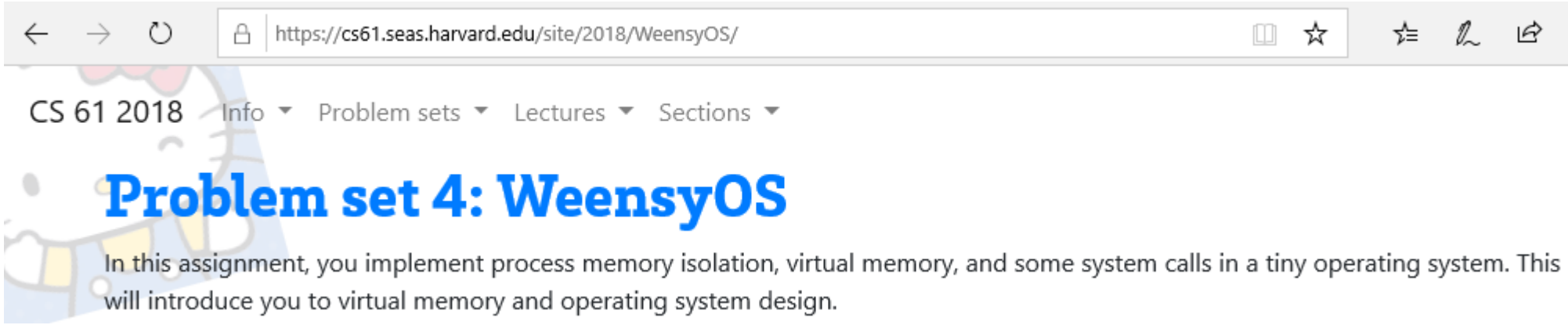
DESCRIPTION

The *fork*() function shall create a new process. The new process (child process) shall be an exact copy of the calling process (parent process) except as detailed below:

Motivation

- We've first-hand experience of many research OSes
  L4, Wanda, Nemesis, Mungi, Hurricane, Tornado, K42, Barrelfish, Drawbridge …
- Supporting fork, or choosing not to, repeatedly tied our hands
- This is common wisdom among those who have built non-Unix OSes
- And yet…

Motivation

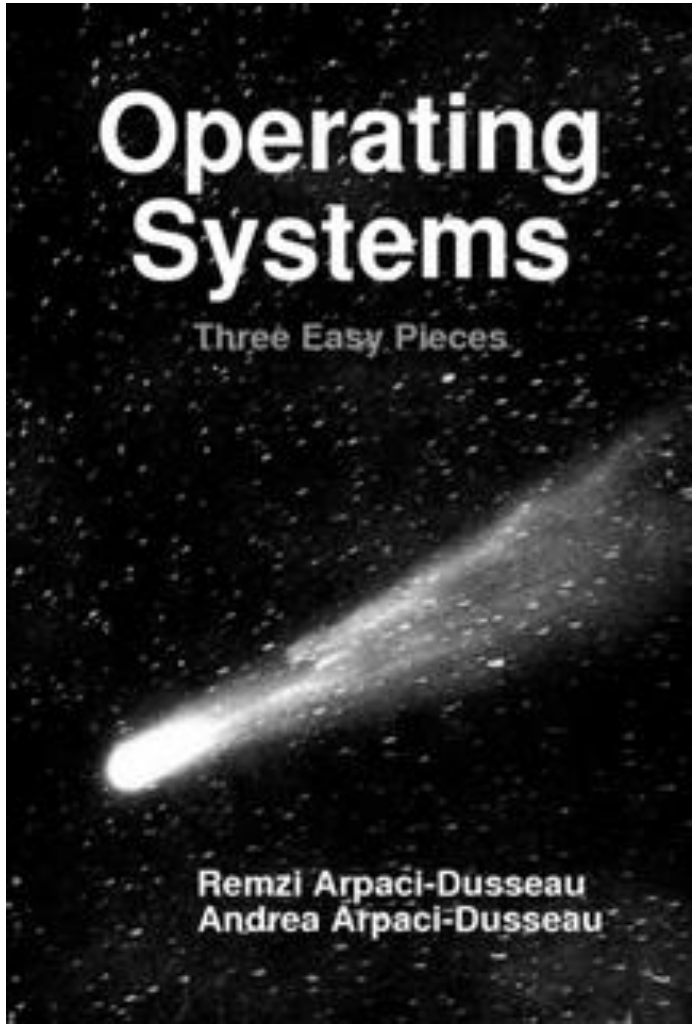


...

Step 5: Fork

The `fork` system call is one of Unix's great ideas. It starts a new process as a *copy* of an existing process. The `fork` system call appears to return twice, once to each process. To the child process, it returns 0. To the parent process, it returns the child's process ID.

Motivation



5.4 Why? Motivating The API

Of course, one big question you might have: why would we build such an odd interface to what should be the simple act of creating a new process? Well, as it turns out, the separation of `fork()` and `exec()` is essential in building a UNIX shell, because it lets the shell run code *after* the call to `fork()` but *before* the call to `exec()`; this code can alter the environment of the about-to-be-run program, and thus enables a variety of interesting features to be readily built.

TIP: GETTING IT RIGHT (LAMPSON'S LAW)

As Lampson states in his well-regarded "Hints for Computer Systems Design" [L83], "**Get it right**. Neither abstraction nor simplicity is a substitute for getting it right." Sometimes, you just have to do the right thing, and when you do, it is way better than the alternatives. There are lots of ways to design APIs for process creation; however, the combination of `fork()` and `exec()` are simple and immensely powerful. Here, the UNIX designers simply got it right. And because Lampson so often "got it right", we name the law in his honor.

Why do people like fork?

- It's *simple*: no parameters!
  - cf. Win32 CreateProcess
- It's *elegant*: fork is orthogonal to exec
  - System calls that a process uses on itself also initialise a child
  - e.g. shell modifies FDs prior to exec
- It eased concurrency
  - Especially in the days before threads and async I/O

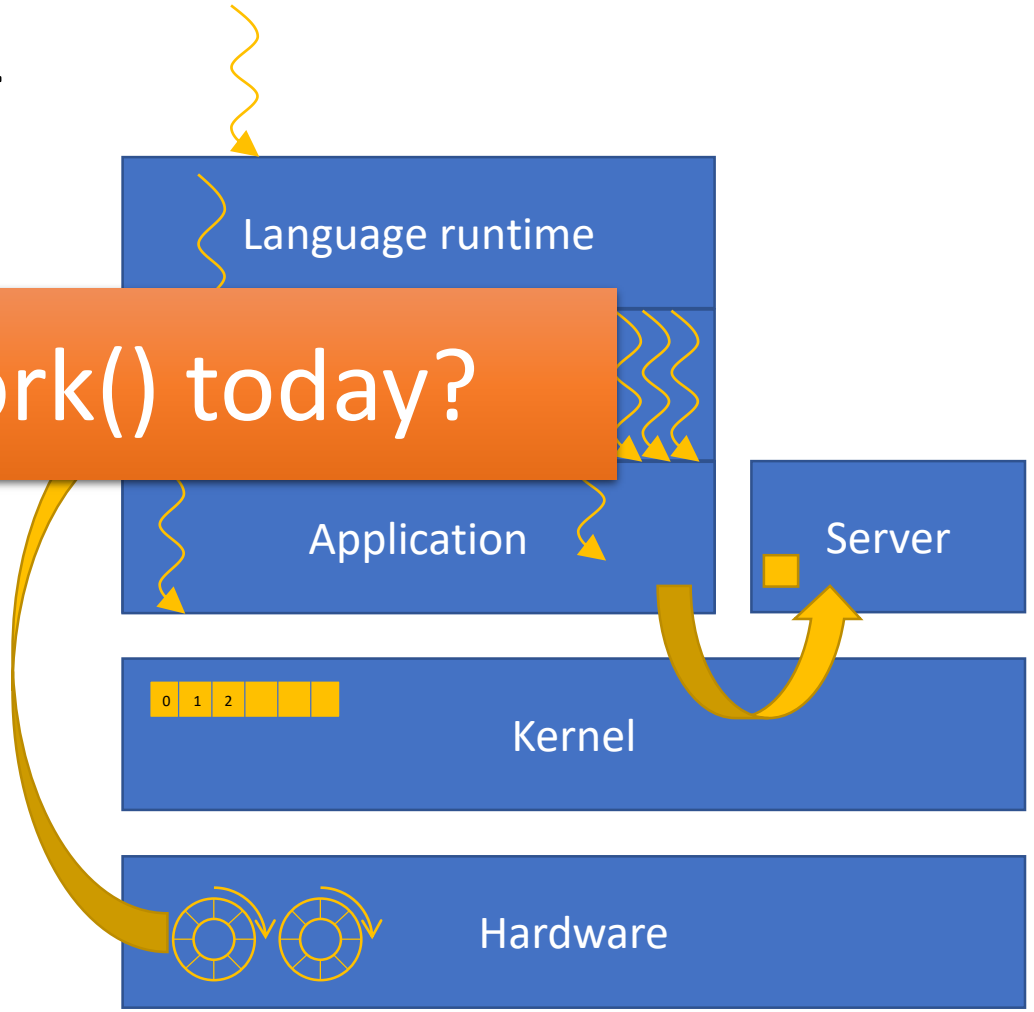
Fork today

- Fork is no longer simple
- Fork doesn’t compose
- Fork isn’t thread-safe
- Fork is insecure
- Fork is slow
- Fork doesn’t scale
- Fork encourages memory overcommit
- Fork is incompatible with a single address space
- Fork is incompatible with heterogeneous hardware
- Fork infects an entire system

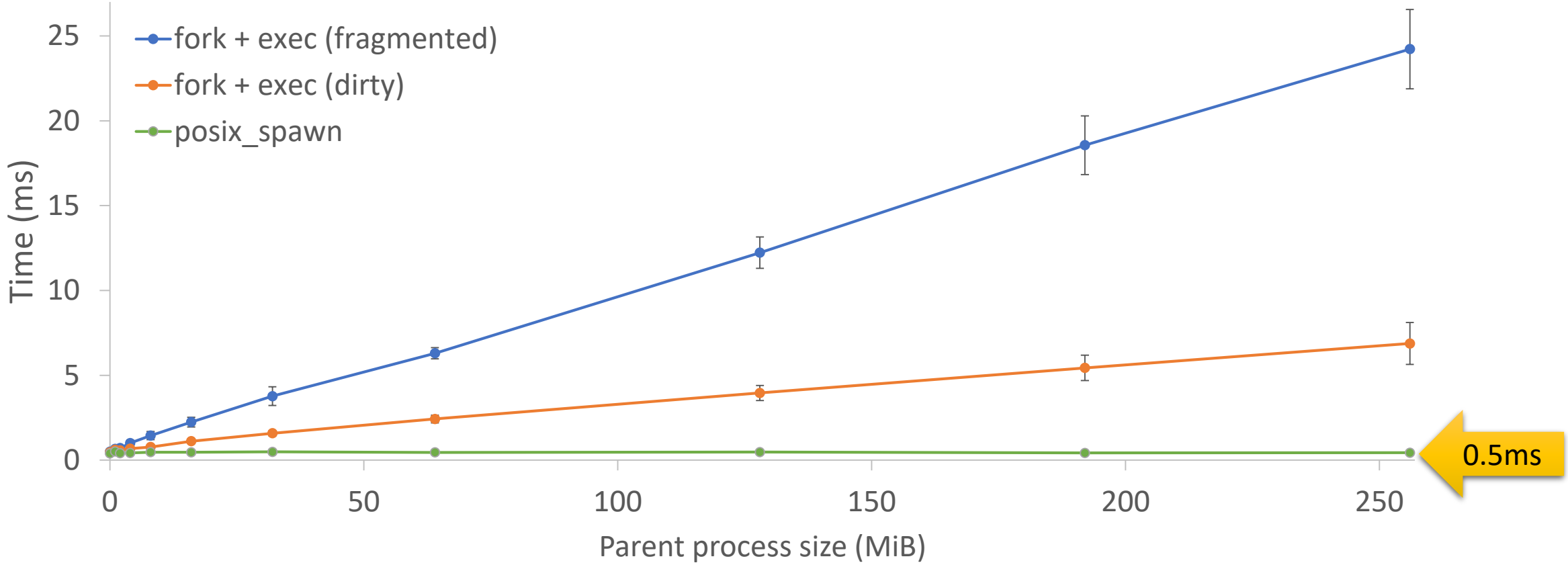
Fork doesn’t compose

- Fork creates a process by cloning another
- Where is the state of a process?
  - In classic Unix:
    - C
  - Toda
    - User-mode libraries
    - Threads
    - Server processes
    - Hardware accelerator context
- Every component must support fork
  - Many don’t → undefined behaviour

Who would accept fork() today?



Fork is slow


fork + exec (fragmented)
fork + exec (dirty)
posix_spawn
Time (ms)
25
20
15
10
5
0
0
50
100
150
200
250
Parent process size (MiB)
0.5ms

Fork infects a system: the K42 experience

- Scalable multiprocessor OS, developed at IBM Research
- Object-oriented kernel and libraries
  - Separation of concerns between files, memory management, etc.
  - Multiple implementations (e.g. single-core, scalable)
- Aimed to support multiple OS personalities
  - However, competitive Linux performance demanded efficient fork…
- Efficient fork requires:
  - Centralised state → lack of modularity, poor scalability
  - Lazy copying → complex object relationships
- Result: every interface, and every object, must support fork
  - Made a mess of the abstractions
  - Led to abandoning other OS personalities


So Ken, where did fork come from anyway?

Origins of fork

Unix designers credit Project Genie (Berkeley, 1964-68)

> *"The fork operation, essentially as we implemented it, was present in the GENIE time-sharing system"*

[Ritchie & Thompson, CACM 1974]

The UNIX Time-Sharing System

Dennis M. Ritchie and Ken Thompson
Bell Laboratories

UNIX is a general-purpose, multi-user, interactive operating system for the Digital Equipment Corporation PDP-11/40 and 11/45 computers. It offers a number of features seldom found even in larger operating systems, including: (1) a hierarchical file system incorporating demountable volumes; (2) compatible file, device, and inter-process I/O; (3) the ability to initiate asynchronous processes; (4) system command language selectable on a per-user basis; and (5) over 100 subsystems including a dozen languages. This paper discusses the nature and implementation of the file system and of the user command interface.

Key Words and Phrases: time-sharing, operating system, file system, command language, PDP-11
CR Categories: 4.30, 4.32

This is a revised version of a paper presented at the Fourth ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15-17, 1973. Authors' address: Bell Laboratories, Murray Hill, NJ 07974.

1. Introduction

There have been three versions of UNIX. The earliest version (circa 1969–70) ran on the Digital Equipment Corporation PDP-7 and -9 computers. The second version ran on the unprotected PDP-11/20 computer. This paper describes only the PDP-11/40 and /45 [1] system since it is more modern and many of the differences between it and older UNIX systems result from redesign of features found to be deficient or lacking.

Since PDP-11 UNIX became operational in February 1971, about 40 installations have been put into service; they are generally smaller than the system described here. Most of them are engaged in applications such as the preparation and formatting of patent applications and other textual material, the collection and processing of trouble data from various switching machines within the Bell System, and recording and checking telephone service orders. Our own installation is used mainly for research in operating systems, languages, computer networks, and other topics in computer science, and also for document preparation.

Perhaps the most important achievement of UNIX is to demonstrate that a powerful operating system for interactive use need not be expensive either in equipment or in human effort: UNIX can run on hardware costing as little as $40,000, and less than two man-years were spent on the main system software. Yet UNIX contains a number of features seldom offered even in much larger systems. It is hoped, however, the users of UNIX will find that the most important characteristics of the system are its simplicity, elegance, and ease of use.

Besides the system proper, the major programs available under UNIX are: assembler, text editor based on QED [2], linking loader, symbolic debugger, compiler for a language resembling BCPL [3] with types and structures (C), interpreter for a dialect of BASIC, text formatting program, Fortran compiler, Snobol interpreter, top-down compiler-compiler (TMG) [4], bottom-up compiler-compiler (YACC), form letter generator, macro processor (M6) [5], and permuted index program.

There is also a host of maintenance, utility, recreation, and novelty programs. All of these programs were written locally. It is worth noting that the system is totally self-supporting. All UNIX software is maintained under UNIX; likewise, UNIX documents are generated and formatted by the UNIX editor and text formatting program.

2. Hardware and Software Environment

The PDP-11/45 on which our UNIX installation is implemented is a 16-bit word (8-bit byte) computer with 144K bytes of core memory; UNIX occupies 42K bytes. This system, however, includes a very large number of device drivers and enjoys a generous allotment of space for I/O buffers and system tables; a minimal system

Project Genie aka

SDS 940 Time-Sharing Computer

3. FORKS AND JOBS

CREATION OF FORKS

A fork may create new, dependent, entries in the PAC table by executing BRS 9. This BRS takes its argument in the A register, which contains the address of a seven-word panic table with the format given in Table 2.

Table 2. Panic Table

Word	Contents
0	Program counter
1	A register
2	B register
3	X register
4	First relabeling register
5	Second relabeling register
6	Status

The status word may be

-2 Dismissed for input/output

-1 Running

0 Dismissed on escape or BRS 10

1 Dismissed on illegal instruction panic

2 Dismissed on memory panic

The panic table address must not be the same for two dependent forks of the same fork, or overlap a page boundary. If it is, BRS 9 is illegal. The first six bits of the A register have the following significance as shown in Figure 3.

Bit	Significance
0	Make fork Executive if current fork is Executive.
1	Set fork relabeling from panic table. Otherwise, use current relabeling.
2	Propagate escape assignment to fork (see BRS 90).
3	Make fork fixed memory. It is not allowed to obtain any more memory than it is started with.
4	Make fork local memory. New memory will be assigned to it independently of the controlling fork.
5	Make fork Exec type 1 if current fork is Exec.

Figure 3. Significance of Bits in A Register

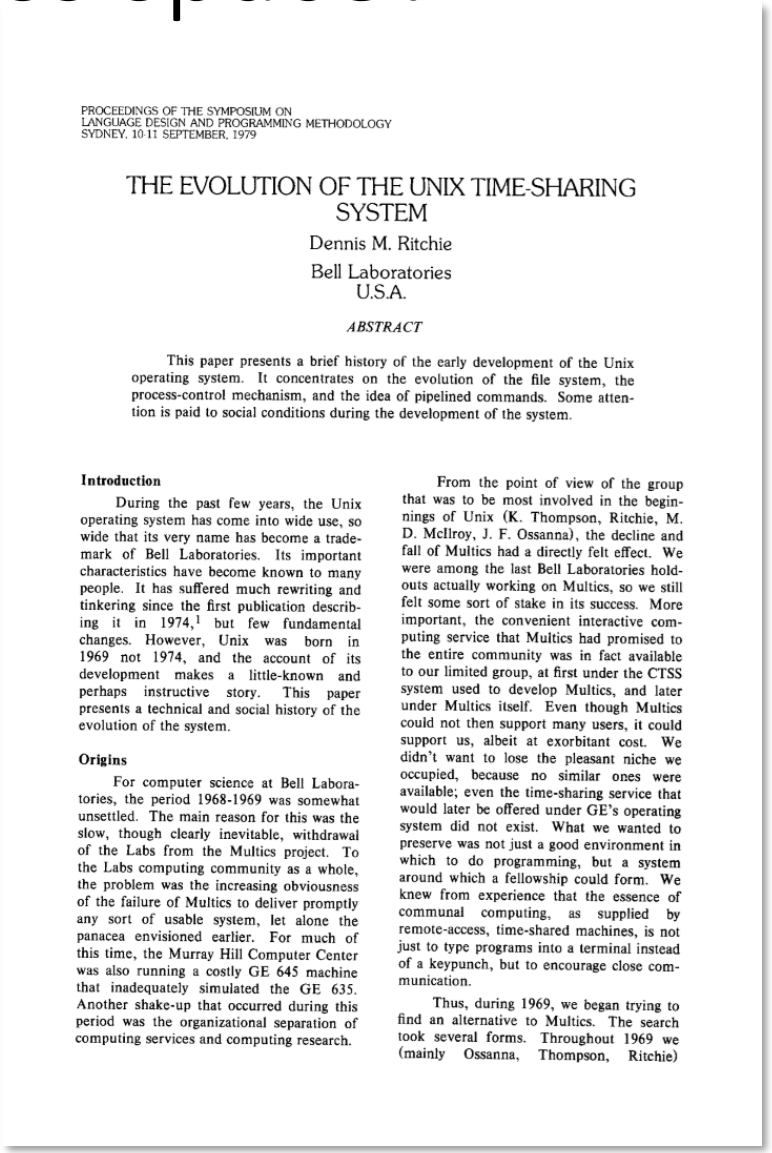
When BRS 9 is executed, a new entry in the PAC table is created. This new fork is said to be a fork of the fork creating it. This is called the controlling fork. The fork is said to be lower in the hierarchy of forks than the controlling fork. The latter may itself be a fork of some still higher fork. A job may have a maximum of eight forks including the executive. The A, B and X registers for the fork are set up from the current contents of the panic table. The address at which execution of the fork is to be started is also taken from the panic table. The relabeling registers are set up either from the current contents of the panic table or from the relabeling registers of the currently running program. An executive fork may change the relabeling. A user fork is restricted to changing relabeling in the manner permitted by BRS 44. The status word is set to -1 by BRS 9. The fork number that is assigned is kept in PIM. This number is an index to the fork parameters kept in the TS block.

The fork structure is kept track of by pointers in PACT. For each fork PFORK points to the controlling fork, PDOWN to one of the subsidiary forks, and PPAR to a fork on the same level. All the subsidiary forks of a single fork are chained in a list.

Why did Unix fork copy the address space?

For implementation expedience [Ritchie, 1979]

- fork was 27 lines of PDP-7 assembly
  - One process resident at a time
  - Copy parent's memory out to swap
  - Continue running child
- exec didn't exist – it was part of the shell
  - Would have been more work to combine them



PROCEEDINGS OF THE SYMPOSIUM ON
LANGUAGE DESIGN AND PROGRAMMING METHODOLOGY
SYDNEY, 10-11 SEPTEMBER, 1979

THE EVOLUTION OF THE UNIX TIME-SHARING SYSTEM

Dennis M. Ritchie

Bell Laboratories
U.S.A.

ABSTRACT

This paper presents a brief history of the early development of the Unix operating system. It concentrates on the evolution of the file system, the process-control mechanism, and the idea of pipelined commands. Some attention is paid to social conditions during the development of the system.

Introduction

During the past few years, the Unix operating system has come into wide use, so wide that its very name has become a trademark of Bell Laboratories. Its important characteristics have become known to many people. It has suffered much rewriting and tinkering since the first publication describing it in 1974,[1] but few fundamental changes. However, Unix was born in 1969 not 1974, and the account of its development makes a little-known and perhaps instructive story. This paper presents a technical and social history of the evolution of the system.

Origins

For computer science at Bell Laboratories, the period 1968-1969 was somewhat unsettled. The main reason for this was the slow, though clearly inevitable, withdrawal of the Labs from the Multics project. To the Labs computing community as a whole, the problem was the increasing obviousness of the failure of Multics to deliver promptly any sort of usable system, let alone the panacea envisioned earlier. For much of this time, the Murray Hill Computer Center was also running a costly GE 645 machine that inadequately simulated the GE 635. Another shake-up that occurred during this period was the organizational separation of computing services and computing research.

From the point of view of the group that was to be most involved in the beginnings of Unix (K. Thompson, Ritchie, M. D. McIlroy, J. F. Ossanna), the decline and fall of Multics had a directly felt effect. We were among the last Bell Laboratories holdouts actually working on Multics, so we still felt some sort of stake in its success. More important, the convenient interactive computing service that Multics had promised to the entire community was in fact available to our limited group, at first under the CTSS system used to develop Multics, and later under Multics itself. Even though Multics could not then support many users, it could support us, albeit at exorbitant cost. We didn't want to lose the pleasant niche we occupied, because no similar ones were available; even the time-sharing service that would later be offered under GE's operating system did not exist. What we wanted to preserve was not just a good environment in which to do programming, but a system around which a fellowship could form. We knew from experience that the essence of communal computing, as supplied by remote-access, time-shared machines, is not just to type programs into a terminal instead of a keypunch, but to encourage close communication.

Thus, during 1969, we began trying to find an alternative to Multics. The search took several forms. Throughout 1969 we (mainly Ossanna, Thompson, Ritchie)

Fork was a hack!

- Fork is not an inspired design, but an accident of history
- Only Unix implemented it this way
- We may be stuck with fork for a long time to come
- But, let’s not pretend that it’s still a good idea today!

Get the fork out of my OS!

- Deprecate fork!
- Improve the alternatives
  - posix_spawn(), cross-process APIs
- **Please,** stop teaching students that fork is good design
  - Begin with spawn
  - Teach fork, but include historical context

- See our paper for:
  - Alternatives to fork, specific use cases, war stories, and more