

Doing the right thing

in

systems

programming

education

Thomas F. Piatkowski

Professor of Computer Science
Professor of Electrical and Computer Engineering
Department of Computer Science
Western Michigan University
Kalamazoo MI 49008

(616) 387-5974

thomas.piatkowski@wmich.edu

<http://www.cs.wmich.edu/~piat>

Abstract

During the past three years the author has taught an experimental version of **CS 224 Systems Programming Concepts**, a traditional sophomore core course in computer science at Western Michigan University. One of the objectives of the course has been to **improve the way systems programming is taught** by introducing a more professional style and methodology into the computer science curriculum; namely:

1. **Adding more rigor** to the technical content of the course with the creation and documentation of a **semi-formal reference model of the UNIX operating system**. [This effort has been very successful, resulting in an innovative elegant detailed teaching model of UNIX as a specialized discrete-state machine.]
2. **Modifying the teaching/learning environment** in the course to make it more productive and more like a professional environment. More emphasis is placed on **collaborative learning, formal analysis of case studies, written and oral communication, cleanroom software engineering, a formal requirements/proposal/implementation/documentation project cycle, and undergraduate research**.

The approach has led to the collaborative creation by professor and students of a significant amount of useful detailed technical material that is not to be found elsewhere in the literature.

The principal purpose of this poster presentation is to overview:

1. The historical setting for the course offering.
2. The structure and syllabus of the course.
3. The principal innovative features of the course, especially those relating to the formal teaching model of UNIX.

The poster presentation concludes with comments on the status of this effort and possible directions for future work.

PROLOGUE

I have had a long active professional interest in software engineering and the use of formal methods in the computer-system life-cycle. I have been involved in the production, and in managing the production, of really large important computer systems; I was the technical director at IBM for the formal specification of the first release of IBM's System Network Architecture (SNA) and I was manager of the group within Burroughs (now Unisys) responsible for the initial design of Burroughs Network Architecture (BNA).

In the fall of 1996 I began teaching CS 224 Systems Programming Concepts for the first time. While making my initial preparations to teach the course and during the first two semesters of offering it, I began to develop some critical opinions as to how systems programming, UNIX, and the C language are presented to our students (computer science majors for the most part). My insights were sharpened by the fact that I was initially 1) seeing the course from the same point of view as my students (not being a UNIX tyro, by any means); and 2) seeing the course from the point-of-view of a professional software engineer.

The experiment of introducing some formalism and an architectural model for UNIX, and the decision to utilize tools and techniques of industrial software engineering thus began.

This poster presentation illustrates a few of the important highlights from this ongoing effort. Teaching CS 224 has afforded me a tremendously satisfying teaching experience and I hope for my students an equally satisfying learning experience.

To the interested reader, I hope this poster presentation provides some insights and ideas of value.

Thomas F. Piatkowski
Kalamazoo, Michigan
1999 October 14

Objectives of the course

For many years the traditional objective of this course has been to master most of the material in pp. 1- 469 of the course text [Stevens]. The major topics include:

- An overview of the SUN architecture and data networks
- The history of UNIX and UNIX standards
- Unbuffered I/O and related system calls
- Files and directories and related system calls
- Buffered I/O and related function calls
- The UNIX process and related system calls
- Signals and interprocess communication and related system calls

The course enhancements overviewed in this poster presentation include:

- Documentation types: executive summary, analysis report, requirements, preproposal, implementation report, testing report, manpages
- UNIX-oriented block diagramming and flowcharting
- Anthology of block diagrams and flowcharts describing programs presented and system calls described in the course text [Stevens]
- Timing features of UNIX
- The waterfall system development process, including one-on-one technical reviews

Philosophy and rationale of the course enhancements

To improve our traditional systems programming course in three significant ways, we

PUT THE RIGHT MATERIAL

high quality tutorial material — exemplary paradigms for all the document types our students are expected to master

high quality student work — the better examples of student work are selected for refinement and incorporation into an evolving course pack that comprises a legacy from the students of each class to their successors

IN THE RIGHT PLACE

the introductory course in systems programming is an ideal place to start introducing *good software engineering practice*. It follows two semesters of learning the ropes as novice solo programmers, and one semester of introductory computer architecture where students are introduced to the details of a modern hardware environment and to the rudiments of hardware concurrency

the systems programming course is next — it provides the first opportunity to take an inside look at *programming-in-the-large* and *how the experts did it* :

- ! how operating systems are architected and programmed
- ! how system and library call interfaces are architected and how some of them might be programmed
- ! how an industrial-strength operating system is documented — both for tutorial and reference purposes

IN THE RIGHT WAY

formal modeling is good — clear rigorous mathematically manipulable notation (as opposed to natural language) used to specify systems in all their details

- ! enhances clear thinking about systems and their behavior
- ! facilitates effective communication on design and behavioral issues
- ! leads to concise unambiguous designs
- ! opens the way to computer-aided toolsets

cleanroom software engineering is good — a proven approach to software development that emphasizes defect prevention. Promotes system design based on implementation-independent constructs. Does not allow access to the implementation environment in the design specification stage

collaborative learning is good — a style of learning where students assist each other in planning how to address learning objectives and in executing their plans. Collaborative learning skills are essential in the engineering workplace and can be an enhancing feature of undergraduate computer science courses (such as this one).

Course syllabus highlights

CS 224 Systems Programming Concepts

Catalog Description

Topics include program development tools, basic testing, timing, profiling and benchmarking, characteristics of physical devices, memory management, device drivers, pseudo-devices, file structures, file I/O (both buffered and unbuffered), processes, shells, inter-process communications, signals, exceptions, pipes, sockets, shared memory and file and record locking, All topics are viewed from a UNIX systems programming perspective. (3 hrs.)

Prerequisites

CS 112 (Computer Science II) and
CS 223 (Computer Organization and Assembly Language)

Objective

To master most of the material in pp. 1- 469 of the course text [Stevens] and to begin mastering software engineering techniques for programming in the large.

Required texts (2)

W.R. Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley, Reading MA, 1992.

CS 224 Course pack for Fall 1999

Commentary

1. This course encourages working together and helping one another in several ways:
 - a. Grading will be on an absolute scale -- not curved.
 - b. Good pedagogical material can be created by students during the course and we all benefit from sharing such material.
 - c. **All student homework, examination, or project submissions are subject to annotation, modification, reproduction and distribution by the instructor.**
2. Homework and projects must be of reproducible quality to be eligible for highest grades.
3. Homework and project assignments may be done in groups *when explicitly allowed in the assignment statement*. *Group submissions must explicitly acknowledge all participants*. All group participants will receive the same grade on collaborative work.
4. Copying the work of others is generally permitted **if it is properly cited**. Failure to acknowledge sources will be treated as plagiarism .
5. Homework and project assignments that are assigned to be done individually *must be done **totally** individually*.
6. All exams must be done individually by each student without assistance from any other person or source unless explicitly allowed in the exam statement.

Analysis component

being able to take a systems program from the text (or other source) and completely determine its structure and behavior from a detailed analysis of its code and other references

Synthesis component

being able to take a general problem requirements statement and design an elegant systems program that meets the requirements

Research component

being able to independently locate, learn, and utilize information related to the objectives of this course

being able to instrument programs and empirically measure and evaluate their performance attributes

Documentation component

being able to effectively use informal and formal descriptive techniques to record and teach how a given systems program is constructed and behaves

Demonstration component

being able to effectively run and test systems programs (both those designed by the student and those provided from other sources)

Technical concepts and notation

A suite of thoughtfully designed technical concepts and notational conventions support the use of a formal and architecturally based specification of UNIX and the UNIX environment.

Techniques that have been adapted to fit the UNIX environment in an especially symbiotic manner include:

- Viewing UNIX as a discrete-state machine

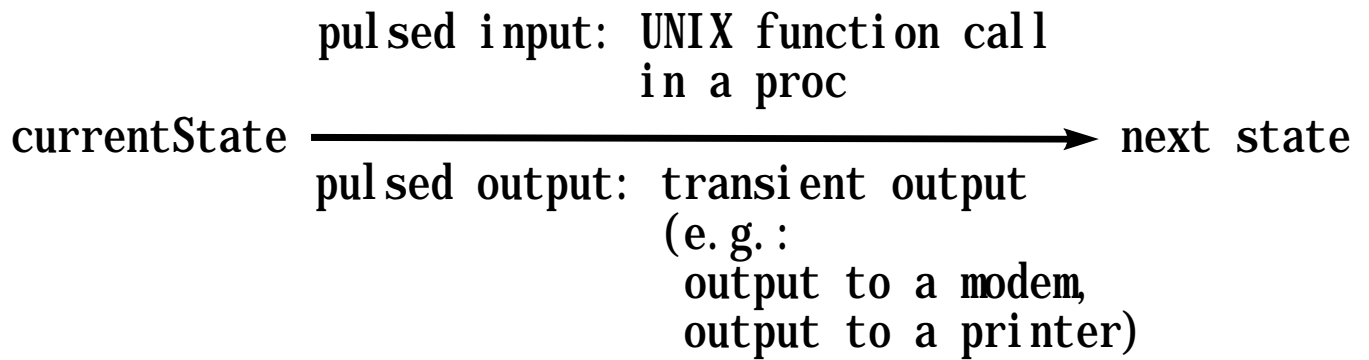
- The state-structure graph of UNIX

- UNIX-oriented naming conventions

- Block diagrams with sequencing

- UNIX-oriented flowcharting

UNIX as a discrete-state machine




```

{ proc }
+-- errno
+-- cwd } environment variables
+-- name }
+-- { FD }
+-- { ( FP, FILE* ) }
+-- { FILE }
+--   FD
+--   buf*
+--   size_buf
+--   n_char
+--   error_flag
+--   eof_flag
+--   f_pos
+--   mode: { _IOFBF,
+--          _IOLBF, _IONBF }
+--   type: { r,   rb,   w,
+--          wb,  a,   ab,
+--          r+,  r+b,  rb+,
+--          w+,  w+b,  wb+,
+--          a+,  a+b,  ab+ }
+-- { buf }
+--   { parameters }
+--   data
+-- ( exitHandler* )
+-- code
+-- rlimit_core } [ resource limits, S180 ]
+-- rlimit_cpu  }

```

processes

```

{ ker }
+-- HZ
+-- TICK
+-- time [ seconds since EPOCH ]
+-- hrestime
+-- localFileSystemElement*
+-- { ( PID , procTable* ) }
+-- { procTable }
+-- activityState : { terminated, ready, running,
                    blocked, waiting, paused, zombie }
+-- alarmClock
+-- proc*
+-- uid
+-- gid
+-- euid
+-- egid
+-- { suppgid }
+-- ssuid [ saved set-user-ID ]
+-- ssgid [ saved set-group-ID ]
+-- parentPID
+-- p_utime
+-- p_stime
+-- p_cutime
+-- p_cstime
+-- cmask
+-- { ( SIGNO , sigQ , sigMask, sigAction ) }
+-- { ( FD , fileTable* ) }
+-- { fileTable }
+-- offset [ Byte basis ]
+-- vnode*
+-- f_count [process references][GC337]
+-- read
+-- write
+-- append
+-- sync
+-- non-blocking
+-- close-on-exec
} status bits

```

kernels

```

nfs [GC335...]
+--{ vnode }
+-- v_count [ fileTable references ] [GC348]
+-- { lock }
+-- l_type : { F_RDLCK, F_WRLCK, F_UNLOCK }
+-- l_whence : short
+-- l_start : off_t
+-- l_len : off_t
+-- l_sysid : long
+-- l_pid : pid_t
+-- pad : long
+-- stat
+-- st_mode
+-- type : { regular, directory, character special,
+--         block special, FIFO, socket, symbolic link } [S75]
+-- set_user_ID
+-- set_group_ID
+-- userRead
+-- userwrite
+-- userExecute
+-- groupRead
+-- groupwrite
+-- groupExecute
+-- otherRead
+-- otherwrite
+-- otherExecute } permissionBits
+-- st_ino
+-- st_dev
+-- st_rdev
+-- st_nlink [ directory references ]
+-- st_uid
+-- st_gid
+-- st_size
+-- st_atime
+-- st_mtime
+-- st_ctime
+-- st_blksize [ Byte basis ]
+-- st_blocks [ Block basis ]
+-- inode
+-- ( dataPtrBlock* )10 [typically the initial allocation]
+-- singleIndirectDataPtrBlock*
+-- doubleIndirectDataPtrBlock*
+-- { doubleIndirectDataPtrBlock }
+--   +-- ( singleIndirectDataPtrBlock* )128
+-- { singleIndirectDataPtrBlock }
+--   +-- ( dataPtrBlock* )128
+-- { dataPtrBlock }
+--   +-- ( dataBlock* )128
+-- { dataBlock }
+--   +-- ( byte )512
+-- { directoryBlock }
+--   +-- { ( filename , nfs.vnode* ) }

```

nfs

UNIX-oriented naming conventions

Reserved terms for UNIX entities and operations

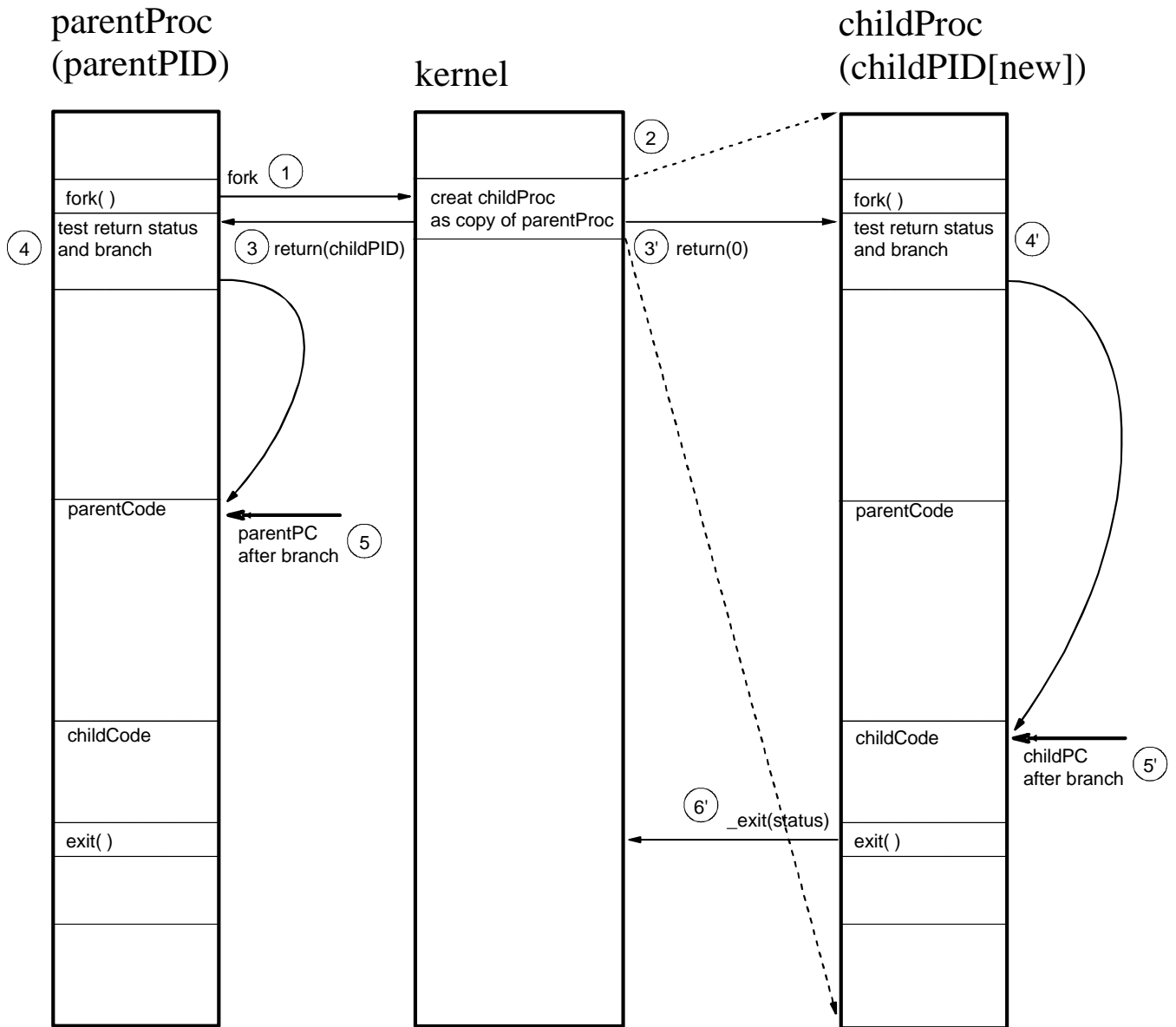
e.g.: FD, fileTable, ker, nfs, PID, proc,
procTable, vnode, ...

path, abs(path), directory(path), filename(path), ...

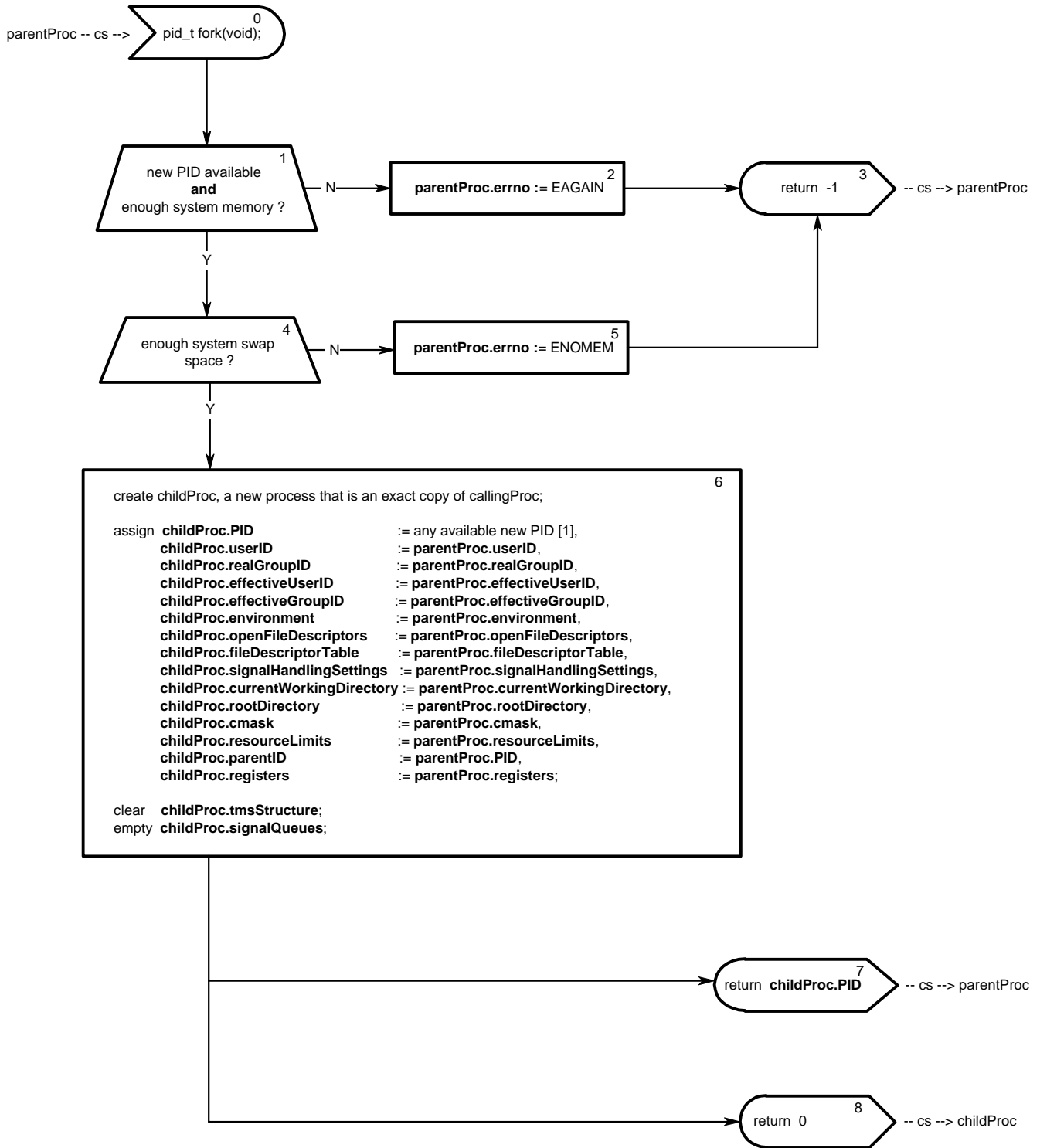
Structured naming that is automorphous to the UNIX state structure

e.g.: proc(ker.procTable(PID).proc*)
ker.fileTable(procTable(PID).fileTable*(FD))
nfs.vnode(ker.fileTable(PID, FD).(nfs.vnode)*)

Block diagram with sequencing for *fork()*



Flowchart for *fork()*



[1] the PID assignment algorithm is implementation dependent.

Course pack

incorporates material developed by the instructor and successive generations of students

Contents

Preface

Acknowledgments

Scope and Organization

Contents

Introduction

- An outline of the history of UNIX
- Cleanroom Software Engineering
- Terminology

The architecture of UNIX and its environment

NFS (Network File System)

LINUX

System and library function calls

Bibliography

Appendix 1 — Timing issues

Appendix 2 — Draft sketches of selected function flowcharts

Appendix 3 — Guide to the manpages

Appendix 4 — Miscellaneous

- Citation and acknowledgment guide for CS 224

- Executive summary style guide for CS 224

- Steven's sourcecode tarfile

- Unix vs. WindowsNT

- sh manpages

- csh manpages

- tcsh manpages

- Taking a tour on the tcsh express

cs224Lib: A Learner's UNIXLibrary

The waterfall system development process

Course library

- ! contains a collection of source code files, object code files, shell scripts and manpages for user commands and library functions; developed by the students and instructor
- ! items demonstrate one or more interesting design features or provide a functionality related to the course
- ! a dynamic document; upgraded from time to time; appearing in a sequence of improving and growing versions
- ! presented to the students of CS 224

that they will find its contents helpful in their study of UNIX and systems programming

that they will find some commands and functions that make them more productive in CS 224 and other courses

and

that they in their turn will find here a model and a vehicle for contributing to a better understanding of UNIX and systems programming on the part of their colleagues and teachers both now and in the future.

Possible directions for future effort

1. Complete the development of block diagrams, flowcharts, and other technical material to cover all course topics from the text book [Stevens].
2. Carefully validate all the block diagrams and flowcharts; this will require 1) careful study of the UNIX operating system documentation and 2) testing.
3. Expand the scope to include:
 - a. LINUX — this will allow direct study of implementation C-code.
 - b. WindowsNT (or Windows2000)
 - c. Tornado — or some other important real-time operating system.
4. Measure the effectiveness and quality of this more formal method of instruction against the more traditional and less formal approach.
5. Prepare a proper text book on UNIX using the formal modeling techniques developed in CS 224 as a principal expository vehicle.

Bibliography

T.F. Piatkowski (ed), *cs224Lib: A Learner's UNIX Library* (3e), Bronco Books, Kalamazoo MI, August 1999.

T.F. Piatkowski (ed), *UNIX: A Learner's Reference Model* (5e), Bronco Books, Kalamazoo MI, August 1999.

L.J. Arthur and T. Burns, *UNIX Shell Programming*, John Wiley, New York, 1994.

P. DuBois, *csch & tcsh*, O'Reilly & Associates, Sebastopol CA, 1995.

B. Goodheart and J. Cox, *The Magic Garden Explained*, Prentice-Hall, New York, 1994.

K. Haviland and B. Salama, *UNIX System Programming*, Addison-Wesley, Reading MA, 1987.

Information technology -- Portable Operating System Interface (POSIX) -- Part 1: System Application Program Interface (API)[C Language], ISO/IEC 9945-1:1996(E), ANSI/IEEE Std 1003.1, 1996 Edition, IEEE, New York, 1996.

B.W. Kernighan and D.M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs, 1988.

A. Oram and S. Talbott, *Managing Projects with make*, O'Reilly & Associates, Sebastopol CA, 1991.

R.P. Paul, *SPARC Architecture, Assembly Language Programming, & C*, Prentice-Hall, Englewood Cliffs NJ, 1994.

W.R. Stevens, *Advanced Programming in the UNIX Environment*, Addison-Wesley, Reading MA, 1992.

The Solaris manpages as provided on the systems of the Western Michigan University, Department of Computer Science, Computer Lab Network, Dunbar 4460, Fall 1999.

A. Tanenbaum, *Operating Systems : Design and Implementation*, Prentice-Hall, Englewood Cliffs NJ, 1987.

A. Tanenbaum, *Structured Computer Organization*, Prentice-Hall, Englewood Cliffs NJ, 1990.

P. Wang, *An Introduction to Berkeley UNIX*, PWS Publishing, Boston MA, 1993.

M. Waite, D. Martin, and S. Prata, *The Waite Group's UNIX Primer Plus (2e)*, Sams, Carmel IN, 1994.

Where to get more information

An extended version of this poster presentation

and a supporting document

Terminology and notation for UNIX-oriented objects

are collected under

<http://www.cs.wmich.edu/~piat/professions>

For more information about the author see

<http://www.cs.wmich.edu/~piat>