The bug fixing process in proprietary and Free/Libre Open Source Software: A coordination theory analysis [1]

Kevin Crowston

Syracuse University School of Information Studies
348 Hinds Hall
Syracuse, NY 13244 USA

crowston@syr.edu

1st Revision

September 7, 2005

To appear as: Crowston, K. (in press). In V. Grover & M. L. Markus (Eds.), *Business Process Transformation*. Armonk, NY: M. E. Sharpe.

1 This research was partially supported by the Center for Coordination Science at the Massachusetts Institute of Technology, a fellowship from the Ameritech Foundation through the University of Michigan Institute for Public Policy Studies and by NSF Grants 03-41475 and 04-14468. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. An earlier version of the analysis of the proprietary software process appeared in (Crowston, 1997). Fieldwork and preliminary analysis was done with the assistance of Stephen Brobst. The paper has benefited from discussions with James Howison, Michael Cohen, Jin-tae Lee, Thomas Malone, Charlie Osborn, Brian Pentland and other members of the CCS Process Handbook Project, comments from Gerardine DeSanctis, M. Lynne Markus, Varun Grover and anonymous reviewers and from reading by Marie Williams.

The bug fixing process in proprietary and Free/Libre Open Source Software: A coordination theory analysis

Abstract

To support business process transformation, we must first be able to represent business processes in a way that allows us to compare and contrast them or to design new ones. In this paper, I use coordination theory to analyze the bug fixing processes in the proprietary operating system development group of a large mini-computer manufacturer and for the Free/Libre Open Source Software Linux operating system kernel. Three approaches to identifying dependencies and coordination mechanisms are presented. Mechanisms analyzed include those for task assignment, resource sharing and managing dependencies between modules of source code. The proprietary development organization assigned problem reports to engineers based on the module that appeared to be in error, since engineers only worked on particular modules. Alternative task assignment mechanisms include assignment to engineers based on workload or voluntary assignment, as in Linux. In the proprietary process, modules of source code were not shared, but rather "owned" by one engineer, thus reducing the need for coordination. In Linux, where multiple developers can work on the same modules, alternative resource sharing mechanisms have been developed to manage source code. Finally, the proprietary developers managed dependencies between modules informally, relying on their personal knowledge of which other engineers used their code. The Linux process allows developers to change code in multiple modules, but emphasizes modularity to reduce the need to do so.

A coordination theory analysis of bug fixing in proprietary and free/libre open source software

Introduction

To support business process transformation, we must first be able to represent business processes in a way that allows us to compare and contrast them or to design new ones (Malone *et al.*, 1999). Consider the software problem (bug) fixing process, a process that I will use as a source for examples in this paper. Customers having problems with a piece of software (a bug) report the problems to its developers, who (they hope) eventually provide some kind of solution (a bug fix). In this paper, I compare the bug fixing processes for a proprietary mini-computer operating system and for the Free/Libre Open Source Software (FLOSS)[2] Linux kernel project. The company I studied had an elaborate process to receive problem reports, filter out duplicates of known problems, identify which modules of the system are apparently at fault for novel problems and route the reports to the software engineers responsible for those modules. Along the way, an engineer might develop a workaround (i.e., a way to avoid the problem); the responsible software engineer might develop a change to the code of part of the system (i.e., a patch) to fix it. The patch is then sent to other groups who test it, integrate it into the total system and, eventually, send it to the customers who originally had the problem. (A more

[2] FLOSS is a broad term used to embrace software that is developed and released under some sort of free or open source license. The free software and the open source movements are distinct and have different philosophies but mostly common practices. The licenses they use allow users to obtain and distribute the software's original source, to redistribute the software, and to publish modified versions as source code and in executable form. While the open source movement views these freedoms pragmatically (as supporting a development methodology), the Free Software movement regards them as human rights, a meaning captured by the French/Spanish word 'libre' and by saying "think of free speech, not free beer". (See http://www.gnu.org/philosophy/ and http://opensource.org/ for more details.) This paper focuses on development practices, which we expect to be largely shared across the virtual teams in both movements. However, in recognition of these two communities, we use the acronym FLOSS, standing for Free/Libre and Open Source Software, rather than the more common OSS.

detailed description of this process appears below.) The Linux bug fixing process (which has evolved over time) has a similar but different set of steps. The description and comparison of these processes raises several questions that are key for business process transformation: Why is the process structured this way, with finely divided responsibility for different parts of the process? In what ways are the two processes (proprietary and FLOSS) similar or different and what are the implications of these similarities and differences? And more simply, how else could software development organizations approach problem fixing?

In the remainder of this paper, I present one approach to answering these questions. In the next section I briefly review coordination theory and show how it can guide the analysis and transformation of a process. The bulk of the paper presents a detailed example. The following sections describe the case sites—the software development division of a minicomputer manufacturer and the Linux kernel project—and the data collection and analysis methods. The analysis section presents the dependencies and coordination mechanisms identified in the cases. The paper concludes by briefly evaluating the coordination theory approach and discussing its application in other settings.

Theory: A coordination theory approach to business processes

In this paper, I use coordination theory as an approach to analyzing processes and for understand their diversity. If we examine many companies, we will observe a wide variety of approaches to the software bug fixing process. For example, in other companies (and other parts of the company I studied), when a problem report arrives, it is simply assigned to the next free engineer. If we examine many processes, we will see a similar range of possibilities. Individuals (or firms) may be either generalists who perform a wide variety of tasks, or specialists who perform only a few. Activities may be assigned to actors within a single organization, as with bug fixing; other assignments may take place in a market, as with auditing, consulting and an increasingly wide variety of services; and finally, assignments may be given to others in a network of corporations.

Despite this diversity, when we systematically compare processes, patterns emerge. Organizations that perform the same task often perform essentially the same basic activities. For example, organizations that fix software bugs must all diagnose the bug, write code for a fix and integrate the change with the rest of the system. Looking more broadly, many engineering change processes have activities similar to those for software.

While these general activities are often the same, the processes differ in important details: how these large abstract tasks are decomposed into activities, who performs particular activities and how the activities are assigned. In other words, processes differ in how they are coordinated. However, even with coordination there are common patterns: similar problems arise and are managed similarly. For example, nearly every organization must assign activities to specific actors and task assignment mechanisms can be grouped into a few broadly similar categories. Such mechanisms are the subject matter of coordination theory.

Coordination theory

To analyze these patterns of coordination, I use the framework developed by Malone and Crowston (1994), who define coordination as "managing dependencies between activities" (p. 90). They define coordination theory as the still developing body of "theories about how coordination can occur in diverse kinds of systems" (p. 87). Malone and Crowston analyze group action in terms of *actors* performing *interdependent* *activities* to achieve *goals*. These activities may also require or create *resources* of various types (defined broadly as anything necessary for or the product of an activity, including raw materials, tools, information and the effort of actors).

For example, in the case of software bug fixing, *activities* include diagnosing the bug, writing code for a fix and integrating it with the rest of the system, as mentioned above. *Actors* include the customers and various employees of the software company. In some cases, it may be useful to analyze a group of individuals as a collective actor (Abell, 1987). For example, to simplify the analysis of coordination within a particular subunit, the other subunits with which it interacts might all be represented as collective actors. The *goal* of software bug fixing appears to be eliminating problems in the system, but alternative goals—such as appearing responsive to customer requests—could also be analyzed. In taking this approach, we adopt Dennett's (1987) intentional stance: since there is no completely reliable way to determine someone's goals (or if indeed they have goals at all), we, as observers, can only impute goals to the actors and analyze how well the process accomplishes these goals. Finally, *resources* include the problem reports, information about known problems, computer time, software patches, source code and so on.

It should be noted that in developing this framework, Malone and Crowston describe coordination mechanisms as relying on other necessary group functions, such as decision making, communications and development of shared understandings and collective sense-making (Britton *et al.*, 2000; Crowston & Kammerer, 1998). To develop a complete model of some process would involve modelling all of these aspects: coordination, decision-making and communications. In practice, our analyses have tended to focus on the coordination aspects, bracketing the other phenomenon.

According to coordination theory, actors in organizations face *coordination problems* that arise from dependencies that constrain how tasks can be performed. These dependencies may be inherent in the structure of the problem (e.g., components of a system may interact with each other, constraining the kinds of changes that can be made to a single component without interfering with the functioning of others) or they may result from decomposition of the goal into activities or the assignment of activities to actors and resources (e.g., two engineers working on the same component face constraints on the kind of changes they can make without interfering with each other).

To overcome these coordination problems, actors must perform additional activities, which Malone and Crowston call *coordination mechanisms*. For example, a software engineer planning to change one module in a computer system must first check if the changes will affect other modules and then arrange for any necessary changes to modules that will be affected; two engineers working on the same module must each be careful not to overwrite the other's changes. Coordination mechanisms may be specific to a particular setting, such as a code management system to control changes to software, or general, such as hierarchical or market mechanisms to manage assignment of activities to actors or other resources.

The first key claim of coordination theory is that dependencies and the mechanisms for managing them are general, that is, a particular dependency and a mechanism to manage it will be found in a variety of organizational settings. For example, a common coordination problem is that a particular activity may require specialized skills, thus constraining which actors can work on it; this dependency between an activity and an actor arises in some form in nearly every organization.

Coordination theory thus suggests identifying and studying common dependencies and their related coordination mechanisms across a wide variety of organizational settings.

The second claim is that there are often several coordination mechanisms that could be used to manage a dependency, as the task assignment example illustrates. Possible mechanisms to manage the dependency between an activity and an actor include manager selection of a subordinate, first-come-first-served allocation and various kinds of markets. Again, coordination theory suggests that these mechanisms may be useful in a wide variety of organizational settings. Organizations with similar activities to achieve similar goals will have to manage the same dependencies, but may choose different coordination mechanisms, thus resulting in different processes.

Finally, the previous two claims taken together imply that, given an organization performing some task, one way to generate alternative processes is to first identify the particular dependencies and coordination problems faced by that organization and then consider what alternative coordination mechanisms could be used to manage them.

To summarize, according to coordination theory, the activities in a process can be separated into those that are necessary to achieve the goal of the process (e.g., that directly contribute to the output of the process) and those that serve primarily to manage various dependencies between activities and resources. This conceptual separation is useful because it focuses attention on the coordination mechanisms, which are believed to be a particularly variable part of a process, thus suggesting an approach to redesigning processes. Furthermore, coordination mechanisms are primarily information-processing activities and therefore, good candidates for support from information-technology.

The aim of coordination theory is not new: defining processes and attempting to improve performance has been a constant goal of business process transformation. The focus on dependencies is also a recurring theme. Even the idea of substitute mechanisms has been suggested; for example, Lawler (1989) argues that the functions of an organization's hierarchy, many of which are ways of coordinating lower level actions, can be accomplished in other ways, such as work design, information systems or new patterns of information distribution. However, coordination theory makes many of these earlier notions more precise by decomposing tasks and resources. For example, the classic distinction among sequential, interdependent and network processes of organizing can be decomposed into particular dependencies managed by particular mechanisms. In this view, a network (Powell, 1990), for example, is not a property of a collection of organizations *per se*, but rather a restriction on which actor is chosen to work on a particular task (i.e., how a task-actor dependency is managed). In a hierarchy, a task is assigned to an actor chosen from within the organization, e.g., based on specialization or managerial decision; in a market, from the set of suppliers active in the market, e.g., by bidding; and in a network, from the appropriate member of the network.

A typology of coordination mechanisms

As a guide to such analyses, Crowston (1991a, 2003) presents a typology of dependencies and associated coordination mechanisms. The typology of dependencies and examples of associated coordination mechanisms is shown in Table 1.

Insert Table 1 about here

The main dimension of the typology involves the types of objects involved in the dependency. To simplify the typology, we compress the elements of Malone and Crowston's (1994) framework into two groups: tasks (which includes goals and activities) and resources used or created by tasks (which here includes the effort of the actors). Logically, there are three kinds of dependencies between tasks and resources:

those between two tasks, those between two resources and those between a task and a resource. (As a further simplification, dependencies between more than two elements are decomposed into dependencies between pairs of elements.)

Some aspects of this typology are more developed than others; for example, task-task dependencies have been analyzed in some detail, while the others have not. Specifically, task-task dependencies have been further distinguished by considering what kinds of resources are shared by the two tasks (e.g., shareable or reusable resources), how these resources are used (as an input or as an output of the task) and whether the required uses conflict with each other (e.g., two different uses of a non-reusable resource).

For each dependency, a brief description of an associated coordination mechanism is given. For example, to manage a task-resource dependency, the typology notes that it is necessary first to identify required and available resources, then to choose a particular resource and finally to assign the resource. Managing a prerequisite dependency (a task-task dependency) requires ordering the tasks, ensuring that the output of the first is usable by the second and managing the transfer of the resource from the first to the second. These activities can be performed in many different ways. For example, a manager with a task to assign might know of the available resources or might have to spend time hunting them down. Usability might be managed reactively by testing the resource and returning problems or proactively by involving the user in the production of the resource.

Research setting and data collection

Coordination theory is intended to analyze organizations in a way that facilitates redesign. The question is, does this approach work; that is, can we find dependencies and coordination mechanisms in a real process? Does this analysis help explain commonly used alternative processes or suggest novel ones? I undertook two case analyses to answer these questions, in a setting where the precise processes for decomposing and completing tasks were observable. In the remainder of this paper, I present the application of coordination theory to a particular process, thus grounding the claims of coordination theory within a carefully specified organizational domain. In this section, I provide an overview of the research setting, the data collection and data analysis approaches.

Data collection

Proprietary software. The proprietary software organization in this example was the minicomputer division of a large corporation. In 1989, when the study started, the entire corporation had sales of approximately $10 billion and roughly 100,000 employees. The computer division produced several lines of minicomputers and workstations and developed system software for these computers. The group in this study was responsible for the development of the kernel of a proprietary operating system, a total of about one million lines of code in a high-level language.

The analysis of the proprietary software development process presented here was based on 16 interviews with 12 individuals, including six software engineers, two support group managers and three support group members and one marketing engineer. The interviews were carried out during six trips to the company's engineering headquarters; most were one to two hours long. Additionally, a former member of the software development group assisted in the data collection and analysis.

As discussed above, coordination mechanisms are primarily information-processing activities. Therefore, this study adopted the information processing view of organizations, which focuses on how organizations process information (Galbraith, 1977;

March & Simon, 1958; Tushman & Nadler, 1978). The goal of the data collection was to uncover, in March and Simon's (1958) terms, the programs used by the individuals in the group. March and Simon suggest three ways to uncover these programs: 1) interviewing individuals, 2) examining documents that describe standard operating procedures and 3) observing individuals. I relied most heavily on interviews. As March and Simon (1958) point out, "most programs are stored in the minds of the employees who carry them out, or in the minds of their superiors, subordinates or associates. For many purposes, the simplest and most accurate way to discover what a person does is to ask him" (p. 142).

I started the data collection by identifying different kinds of actors in the group. This identification was done with the aid of a few key informants, and refined as the study progressed. When available, formal documentation of the process was used as a starting point. For example, a number of individuals designed and coded parts of the operating system, all working in roughly the same way and using the same kinds of information; each was an example of a "software engineer actor." However, response centre or marketing engineers used different information, which they processed differently. Therefore they were analyzed separately.

Interview subjects were identified by the key informants, based on their job responsibilities; there was no evidence, however, that their reports were atypical. I then interviewed each subject to identify the type of information received by each kind of actor and the way each type was handled. Data were collected by asking subjects: 1) what kinds of information they received; 2) from whom they received it; 3) how they received it (e.g., from telephone calls, memos or computer systems); 4) how they processed the different kinds of information; and 5) to whom they sent messages as a result. When possible, these questions were grounded by asking interviewees to talk about items they had received that day.

I also collected examples of documents that were created and exchanged as part of the process or that described standard procedures or individual jobs. Not surprisingly, the process as performed often differed from the formally documented process. For example, there was a formal method for tracking which engineers used which interfaces, but in practice most engineers seemed to rely on their memories. It was this informal process (as well as the formal process surrounding it) that I sought to document.

The initial product of these studies was a model of the change process (presented in more detail below) that described the actors involved, which steps each performed and the information they exchanged. It should be noted that data were collected about only one group because my contact at this company worked in that group. My impression from interviews with individuals who had worked in or who interacted with other groups, was that processes were similar in other software development units, however, I have no direct information about other groups.

Relying on interviews for data can introduce some biases. First, people do not always say what they really think. Some interviews were conducted in the presence of another employee of the company, so interviewees may have been tempted to say what they think they should say (the "company line"), what they think I want to hear or what will make themselves or the company look best. Second, individuals sometimes may not know the answer or may be mistaken.

To control for interview bias, I cross-checked reported data with other informants. I also used the modelling process as a check on the data, applying the negative case study method (Kidder, 1981). In this method, researchers switch between data collection and model development, using predictions or implications of the model to guide the search for disconfirming evidence. When such data cannot be found, the model has been refined to agree with all available data.

Linux. As a comparison to the proprietary software company, I examined the bug fixing processes for the kernel of the Linux operating system. Linux is a FLOSS Unix-like operating system. The original release was created by Linus Torvalds in 1991, but it has since grown with the contributions of literally thousands of developers (Moon & Sproull, 2000) and is now a fully-featured and widely used system. The most recent release, 2.6 in 2004, is reported to have more than 4 million lines of code (Wheeler, 2005). A unique feature of Linux is the dual version approach to development, in which even-numbered versions (2.2, 2.4, 2.6, etc.) are intended to be stable for end users and odd-numbered versions (2.1, 2.3, 2.5, etc.) are for development and may therefore be unstable.

FLOSS development differs greatly from proprietary development in that it is not owned by a single organization. Developers contribute from around the world, meet face-to-face infrequently if at all, and coordinate their activity primarily by means of computer-mediated communications (CMC) (Raymond, 1998; Wayner, 2000). These teams depend on processes that span traditional boundaries of place and ownership. The research literature on software development and on distributed work emphasizes the difficulties of distributed software development, but the case of FLOSS development presents an intriguing counter-example. What is perhaps most surprising about the FLOSS process is that it appears to eschew traditional project coordination mechanisms such as formal planning, system-level design, schedules, and defined development processes (Herbsleb & Grinter, 1999). As well, many (though by no means all) programmers contribute to projects as volunteers, without working for a common organization or being paid. Characterized by a globally distributed developer force and a rapid and reliable software development process, effective FLOSS development teams somehow profit from the advantages and overcome the challenges of distributed work (Alho & Sulonen, 1998).

My analysis of the Linux bug change process is based primarily on published descriptions of the process (e.g., Moon & Sproull, 2000). These analyses have been extended by analysis of Linux kernel bug tracking logs. It is important to note that the Linux process has changed over time, with a particularly significant change for the 2.6 version (Larson, 2004). Therefore, I will discuss the process as it appeared at different points in time. For ease of reference, I will refer to these as the original and current change processes, respectively.

The bug fixing processes

In this section, I briefly describe the bug fixing processes as implemented in the proprietary and Linux development processes.

The proprietary bug fixing process. The proprietary software organization stated the following goals for the change process:

- ensure that all critical program parameters are documented: customer commitments, cross-functional dependencies.
- ensure that a proposed change is: reviewed by all impacted software development units/functions and formally approved or rejected.
- ensure that document status is made available to all users: stable (revision number and date); changes being considered; approved/rejected/withdrawn.
- ensure changes are made quickly and efficiently

In addition, the change process had two larger goals: maintain the quality of the software and to minimize the cost of changes. To maintain quality, the process ensured that changes were made by someone who understands the module involved, that changes are fully tested and that the module and its documentation were kept in agreement. To reduce the cost of changes, the change process required that changes be made only to fix a problem or add an authorized enhancement. As one manager put it, the "formal change control process is there to prevent changes."

The activities performed for a typical change in the proprietary process are summarized in the flowchart shown in Figure 1. Although no particular bug is necessarily treated in exactly this way, these activities were described as typical by my interviewees. Roles in bug fixing included customer, marketing engineer, engineering manager, software engineers and quality assurance. Actors involved in the process are listed at the top of the column of activities they perform. (To save space, the flow continues from the bottom right of the chart to the top left; the activities on the right follow rather than overlap those on the left.)

Insert Figure 1 about here

The software maintenance process started when a problem is encountered. When a customer called the customer support centre with a problem, the call handler tried to solve it using manuals, product descriptions, and the database of known problems. If the problem appeared to be a bug that was not already in the database, a new problem report was entered. Many problems were found during the development process by the testing group, who entered problem reports directly. (Note that the response centre was treated as a collective actor. As a result, internal centre activities are omitted from the flowchart of the process.)

A marketing engineer periodically reviewed new entries in the database. Marketing engineers reviewed problem reports for completeness and attempted to replicate the problem. The marketing engineer might decide that the problem is really a request for an enhancement, which was handled by a separate process. If the bug was genuine, the marketing engineer determined the location of the problem and assigned the problem report to the software development unit responsible for that module.

A coordinator in the development unit next assigned the problem report to the appropriate software engineer, who investigated the problem. If the problem turned out to be entirely in another module, then the engineer passed the request to the engineer responsible for the other module. If the problem was internal to a single module, the engineer just fixed the module. If the problem required changes to multiple modules, the engineer discussed the changes with the owners of the affected modules (as well as other interested engineers) and arranged for them to modify their modules appropriately. All changes required the approval of management. Changes to interfaces intended for general use required as well a design review and approval from a change review board.

When the engineer was satisfied with the change and it had been approved, he or she submitted the new code to the testing and integration group. The integration group then recompiled the changed code and relinked the system. The kernel was then tested; any bugs found were reported to the appropriate engineers, potentially starting another pass through the process. Customers were periodically sent the most recent release of the system. In a few cases, they received a patch for a single particularly important or relevant change.

The original Linux bug fixing process. We next consider the original bug fixing process for the Linux kernel. The goals of the Linux change management process are not explicitly stated, but can be inferred from the descriptions of the process. During development for kernels 2.4 and earlier, the concern seemed to be adding code to improve the kernel, while maintaining the quality of the system. Unlike the proprietary process, the FLOSS process does not necessarily distinguish between new features and bug fixes, nor attempt to control what is added beyond ensuring that the code is of good quality (though adding code that might impact the stability of even-numbered kernels, such as a significant new feature, is discouraged). Since developers are also users, the fact that code has been developed is generally justification enough for at least considering its inclusion, though there is no guarantee that a new piece of code will ever be incorporated in an official release.

As with the proprietary process, the original Linux kernel bug fixing process started with bug reports posted to a mailing list for kernel developers for a particular subsystem. Appendix 1 shows an example emailed bug report for the kernel along with follow-up messages and an eventual fix (note that to save space, the source code listings that were contained in the messages have been redacted). The kernel developer list gets thousands of such messages per month. Based on the report, a developer might have tried to characterize the bug or to develop a fix. The decision to work on a fix depended on interest and ability of individual. The work done is in turn sent to the kernel mailing list. Because multiple developers are working on the system in parallel, it is necessary for a new piece of code to be provided in a form that can quickly be integrated with the existing source code and any other new code that may have been created in the meantime (that is, another goal of the change process is to incorporate changes from multiple developers). The most common way to distribute new code is as a "patch", that is, a set of changes to the current files. Patch tools can integrate multiple patches and identify conflicting ones, that is, patches that make overlapping changes to the same lines of code.

Once a patch is developed and posted to the list, other users can provide feedback, e.g., on the coding style, design approach or experiences using the patch. Kernel testing depends heavily on peer review (indeed, Glance (2004) goes as far as to claim that no testing is done). If users encounter problems with the patch, then they provide feedback on the list, thus restarting the bug fixing process.

Linux's originator, Linus Torvalds, coordinates the overall maintenance of the kernel. If user experiences with a patch were positive, Torvalds might incorporate it into his kernel source code tree, often rewriting it in the process. Torvalds' acceptance of a patch was (and still is) the only way into an official Linux release. As Moon and Sproull note (2000),

> there is no confusion about who has decision authority in this project. Torvalds manages and announces all releases—all 569 of them to date. He acts as a filter on all patches and new features—rejecting, accepting, or revising as he chooses. He can single-handedly decide to redo something completely if he wishes.

Torvalds has stated that he is more likely to accept patches from the small number of developers he knows and trusts, so patch writers often send their patches to the maintainers for a particular module and count on them evaluating and forwarding it to Torvalds. Several other developers maintain their own copies of the sources with different policies about which patches they accept, and companies (and indeed, end users) that use Linux can also maintain their own trees with their own selection of patches

The current Linux bug fixing process. With the more recent versions of the kernel, the bug fixing process has changed to be less centered on Torvalds and to make more use of information and communications technologies (ICT) support, specifically, the

Bugzilla[3] bug tracker and a source code control system (first Bitkeeper and now Git[4]) (Larson, 2004). Again, we can infer some of the goals of the new bug changing process by noting the problems the new process is said to fix. Larson (2004) notes that with the original process, “where nobody really knew which changes were in, whether they were merged properly, or what new things to expect in the upcoming release,” while the new processes “ensure that patches aren't forgotten” and gives developers the “ability to always be working with the latest copy of the code” and “to update as soon as a feature or bug fix they need goes into the kernel” (Larson, 2004). We infer then that the current process is intended to keep track of bug reports and patches and to ensure the latter are widely available.

Rather than posting bug reports to the mailing list, in the current process bug reports can be posted on a bug tracking system (Larson, 2004). Use of the Bugzilla system provides a single source for all bug reports and enables tracking of status of each, fixed or not. However, not all developer use the system, as will be discussed below.

Linux kernel sources, which originally were maintained by Torvalds, are now kept in a source code control system, at first BitKeeper (Larson, 2004) and now a new system called Git (Shankland, 2005) Both systems allow patches (in particular, patches spanning multiple files) to be easily tracked and included or dropped in a release so different developers can more easily create patches that can be brought together into a final release. Torvalds is quoted as saying that use of the BitKeepeer sped up

[3] Bugzilla was originally developed for the Mozilla project, but is now widely used by FLOSS developers.

[4] BitKeeper is a proprietary source code control system whose developers provide a free license to Linux developers. Torvalds began using BitKeeper in 2002. After a dispute, the license was revoked in April 2005, after which Torvalds developed his own system, named Git, specialized for Linux kernel sources (Shankland, 2005).

development by a factor of 2 (LeClaire, 2005). Because some developers objected to using BitKeeper, which is proprietary software, kernel sources can be obtained in a variety of other ways and patches may still be posted to the mailing list. Git similarly allows easier maintenance of a set of patches, but it is not yet clear how its use will affect kernel development.

Finally, the testing process for the current release is somewhat more systematic (Larson, 2004). For example, the Linux Test Project (http://ltp.sourceforge.net/) has developed test suites for the kernel. Automated systems exist that allow a developer to provide a patch for the kernel and have the tests run automatically using the patched code. The system can also pull the current release of the kernel from the source code control system and run the regression tests on it. Official releases are still made by Torvalds.

Analysis approach

In this section, we describe a coordination theory analysis for the processes such as the ones described above by identifying the dependencies and coordination mechanisms in place. Three heuristics have been developed to identify dependencies and coordination mechanisms in different ways (Crowston & Osborn, 2003). First, we can examine activities in the current process, identify those that seem to be part of some coordination mechanism and then determine what dependencies they manage. Second, we can list the activities and resources involved in the process, consider what dependencies are possible between them and then determine how these dependencies are being managed. Finally, we can look for problems with the process that hint at unmanaged coordination problems and identify the underlying dependencies. In this section, we will describe each heuristic and give examples of the dependencies and coordination mechanisms identified with this approach. In the following section, we will systematically present the integrated results of the analysis.

Looking for coordination mechanisms

Taking the first approach, many of the activities in the bug fixing process appear to be instances of the coordination mechanisms discussed earlier. Table 2 lists the activities performed in the current bug fixing process. The dependency the activity manages, if any, is listed in the third column.

Insert Table 2 about here.

For example, one of the first things the customer service centre staff and marketing engineers in the proprietary firm do upon receiving a problem report is check if it duplicates a known problem listed in the change database. In the typology, *detect common output* is listed as a coordination mechanism for managing a dependency between two tasks that have duplicate outcomes. The organization can avoid doing the same work twice by looking for the common output in a database of known problems and reusing the result of one of the tasks (as happened in this example). Linux developers similarly mark bugs in Bugzilla as duplicates.

Task assignment is a coordination mechanism for managing the dependency between a task and an actor by finding the appropriate actor to perform the task. Such coordination mechanisms are performed repeatedly in this process: customers assign tasks to the customer service centre, the customer service centre assigns novel tasks to the marketing engineers, marketing engineers assign them to the software engineers and software engineers assign tasks to each other.

Looking for dependencies

The second approach to identifying coordination mechanisms is to list the tasks and resources involved in the process and then consider what dependencies are possible between them. It may be that some of the activities in a process are coordination mechanisms for managing those dependencies. As mentioned above, tasks necessary to respond to problem reports include noticing there is a problem, finding a workaround, reproducing and diagnosing the problem, designing a fix, writing new code and recompiling the system with the new code. These activities are shown in bold in Table 2. Resources include the problem reports, the efforts of a number of specialized actors and the code itself.

Task-task dependencies. Dependencies between tasks can be identified by looking for resources used by more than one task. For example, many tasks create some output, such as a bug report, a diagnosis or new code, which is used as input by some other task, thus creating a *prerequisite dependency* between the two. Malone and Crowston (1994) note that such dependencies often impose usability and transfer constraints. Some steps in the process appear to manage such constraints. For example, testing that a new module works correctly addresses the usability constraint between creating code and relinking and using the system; releasing the new system addresses the transfer constraint between the development organization and the final user of the system.

If there are two problems in the same module, then both bug fixing tasks need the same code, thus creating a *shared resource dependency*. In the proprietary development process, this dependency is managed by assigning modules of code to individual programmers and then assigning all problems in these modules to that programmer. This arrangement is often called “code ownership,” since each module of the system has a single owner who performs all tasks that modify that module. Such an arrangement allows the owner of the code to control all changes made to the code, simplifying the coordination of multiple changes.

Task-resource dependencies. The second category of dependencies is those between tasks and resources, which are managed by some kind of task or resource assignment. These coordination mechanisms were identified and discussed above, e.g., the assignment of bugs to developers by marketing engineers.

Resource-resource dependencies. Finally, there are dependencies between modules owned by different engineers, that is, resource-resource dependencies, that constrain what changes can be made. A module depends on another if the first makes use of services provided by the second. For example, the process manager code may call routines that are part of the file system; therefore, the process management code depends on the file system code. Such dependencies must be noticed or identified before they can be managed by arranging for coordinated changes. Interactions between different parts of a software system are not always obvious, since they are not limited to direct physical connections.

Looking for coordination problems

A final approach for identifying dependencies is to look for problems in the process that suggest unmanaged dependencies. For example, the proprietary developers occasionally found at system integration or during testing that a change made to one module was incompatible with others, despite the efforts to detect and avoid interactions described in the previous section. These problems occur because some dependency between the module being changed and other modules were not detected and managed.

In the case of Linux, the original process had the problem that patches would get dropped due to Torvalds becoming overloaded with submissions. The bug fixing process could be quite frustrating for a developer if a patch was not accepted, because it was hard to know if the decision was based on the quality of the code or because Torvalds was simply too busy to handle the submission. In case it was the later, individuals would often repost patches. As Larson notes, “bugs were often missed, forgotten, or ignored unless the person reporting the bug was incredibly persistent” (2004). As developer Rob Landley succinctly put it, “Linus doesn’t scale” (Barr, 2002).

To summarize, there are three heuristics that can be used to identify dependencies and coordination mechanisms in a process. The first approach is to match activities performed against known coordination mechanisms, such as searching for duplicate tasks or task assignment. The second approach is to identify possible dependencies between activities and resources and then search for activities that do manage these. In the example, we identified prerequisite, shared resource and resource-resource dependencies that were managed. The final approach is to look for problems that suggest unmanaged or incompletely managed dependencies. The coordination mechanisms identified are both generic, such as task assignments, and specific, such as code sharing systems.

Results: Dependencies and coordination mechanisms in software bug fixing

In this section, we discuss the dependencies and coordination mechanisms found in the bug fixing process using the analysis techniques presented above. We will then discuss differences between the FLOSS and proprietary processes as well as additional alternatives suggested by the analysis. Following the typology of dependencies in Table 1, we consider task-task, task-resource and resource-resource dependencies in turn. Table 3 presents a summary of this comparison.

Task-task

Task-task dependencies include shared output, shared resource and flow dependencies.

Duplicate (shared output). The first type of dependency is a common output dependency, meaning that two tasks create the same output. Avoiding *duplicate tasks* is difficult if there are numerous workers who could be working on the same task. For example, the same bug may be reported by many users; in a software development company, managers of the development group would prefer not to waste resources diagnosing and solving the same problem repeatedly.

In the proprietary bug fixing process, this dependency is managed by the marketing engineers, who search the databases for each bug report to identify possible existing solutions or duplicate reports. In the original Linux process, this dependency was managed by allowing users to see all bugs as they were reported or to search the mailing list, thus hopefully avoiding duplicates. However, it seems unlikely that average users are able to keep up with the volume of messages or to identify matching reports. Instead, the burden would fall on the developers to identify when a user report was a duplicate. In the current process, users are encouraged to search the Bugzilla database for duplicate problem reports before filing a new one. Nevertheless, there is no guarantee that users will search or that they will find a duplicate even if it exists, so duplicate bugs are reported and have to be identified and marked as such by developers.

Shared resource. The second type of task-task dependency is a shared resource dependency, which arises when multiple processes need the same resources. For example, multiple bug fixes may have to be made to the same module of code. As mentioned above, one approach to dealing with these dependencies is to have a single person handle all of the changes and the resulting dependencies. This approach does not eliminate the dependencies, but does allow a single user to see and thus manage all of them. In the original Linux process, Torvalds essentially performed this role, since he maintained the kernel source code. In the post-2.4 Linux process, the use of a source code control system helps to manage multiple changes to various files by facilitating integration of changes made by multiple developers and providing notification of conflicts when developers check code back into the system.

Flow. The third type of task-task dependency is a flow dependency, which arises when one task creates a resource that another task requires as an input. While in general a flow dependency implies the need for three different kinds of coordination mechanisms to manage transfer of the resource, ordering of the tasks and usability, we will focus on the third, since usability is the most significant issue in the processes studied. Usability means that there must be some mechanism in place to ensure that the output of the first task is usable by the second task. At the highest level, the entire bug fixing process might be viewed as a mechanism to ensure the usability for the users of the software being developed, thus managing a flow dependency from developers to users.

Within the bug fixing process, several activities ensure that the output of one task is usable by another. An interesting usability constraint, common to both cases, is that bug reports from the general public are often not detailed enough to be useful for developers (Glance, 2004). The low quality of bug reports discourages developers from relying on the system, thus further decreasing the quality of the bugs. In response, some developers may take on the role of filtering or improving reports. In the proprietary process, marketing engineers check that problem reports are detailed enough to be used by the engineers fixing the bugs. In the Linux process, developers might post follow up messages to a report (either on the email list or in the Bugzilla database) requesting additional information.

Similarly, bug fixes need to be tested to check that they correctly fix the problem and do not introduce new problems, so testing is a way to check that output of bug fixing is suitable for use. Testing was done through a rather formal process for the proprietary system, and by a combination of individual and peer review for the Linux process.

In the proprietary process, managers must approve changes before they can be implemented. In Linux, module managers and Torvalds play a similar role. Such approvals provide a check on the quality of the change, either directly, if the manager notices problems, or indirectly, if engineers are more careful with changes they have to show their managers. There are other possible interpretations of this approval process: managers might use the information to allocate resources among different projects, to track how engineers spend their time or even to demonstrate their political power. However, if approvals are a quality check, other mechanisms might be appropriate (in this and any other process). For example, if approvals are time consuming yet likely, it may be more effective to continue the change process without waiting for the approval. Most changes will be implemented more quickly; the few that are rejected will require additional rework, but the overall cost might be lower. Alternatively, managerial reviews could be eliminated altogether in favour of more intensive testing and tracking of test results. Interestingly, Linux allows developers to take this approach, since anyone can maintain a kernel source tree that includes the patches they want to test or use, approved or not.

Task-resource

The next type of dependencies I consider are dependencies between tasks and the resources they need. The primary resources in the bug fixing process are the modules of code and the time and effort of developers.

Task assignment. In the analysis, we noted numerous places where actors perform part of a task assignment process. For example, customers give problem reports to the service centre, which in turn assigns the problems to product engineers, who then assign them to software engineers. In addition, software engineers may assign reports or subtasks to each other. The typology points out that a key problem in task assignment is choosing the actor to whom to assign a task. For the proprietary process, the choice is made based on specialization. Specialization allows engineers to develop expertise in a few modules, which is particularly important when the engineers are also developing new versions of the system. Furthermore, since modules are assigned to engineers, the code sharing problem discussed above is minimized. However, there are also disadvantages. First, diagnosing the location of a problem can be difficult, because symptoms can appear to be in one module as a result of problems somewhere else. In the best case, an error message will clearly identify the problem; otherwise, the problem will be assigned to the most likely area and perhaps transferred later. In any event, making the assignment correctly requires a fair amount of work and experience for the assigner, as is evidenced by the multiple layers involved in making the assignment. A second problem is load balancing: one engineer might have many problems to work on, while others have none.

An alternative basis for choosing engineers was found in a new support group that was set up in the proprietary company during our study. Support engineers were not specialized by module, but were instead organized around change ownership, that is, an engineer assigned a particular problem report makes changes to any affected modules. As a result, task assignment can be done based on workload rather than specialization. In this case, a manager can make the assignment by tracking the status of individual engineers, or engineers can assign work to themselves whenever they finish a task. Many processes could be similarly redesigned to use generalists rather than specialists. For example, in a customer service process, the person who answers the phone could be enabled to resolve any problem rather than having to refer the problem to a specialist.

With change ownership, multiple engineers may have to work on the same module, thus creating a new shared resource dependency. This problem illustrates an important point: coordination mechanisms are themselves activities, and using a different kind of coordination mechanism to manage one dependency may create new dependencies that must in turn be managed. In this case, to manage these new task dependencies, the company implemented a source control system to maintain a copy of all source files. When engineers want to modify a file, they check it out of the system, preventing other programmers from modifying it (by contrast, the Linux source code control system does not lock files, thus permitting parallel development). When the modification is completed, the module is checked back in and the system records the changes made. The activities and analysis of this process are shown in Table 4.

Interestingly, the Linux process does not rely on explicit assignment but instead relies on developers to assign themselves to tasks. With the BugZilla system, depending on the module with the bug, bugs start off assigned to the default maintainer, however, default maintainers may not be the ones who actually fix the bug. This approach makes the assignment process more similar to the market approach suggested by Crowston (1997). In a market form of task assignment, a description of each task is sent to all available agents. Each evaluates the task and, if interested in working on it, submits a bid, saying how long it would take, how much it would cost or even what they would charge to do it. The task is then assigned to the best bidder, thus using information supplied by the agents themselves as the basis for picking who should work on the task. However, the Linux process differs in that there is often no explicit assignment of the task; rather, each developer chooses autonomously whether to work on a bug.

The self-selection process results in two significant differences from the commercial process. First, there is a very uneven distribution of tasks per developer, as shown in the plot of the number of change sets contributed to BitKeeper vs. the number of developers in Figure 2. This figure shows that a few developers contribute many patches, while most developers contribute only a few, something that would likely be undesirable in most companies. A second problem is that multiple developers may choose to work on the same code modules at the same time. Rather than preventing such conflicting uses, the source code control system used for Linux identifies and often resolves any problems created by the parallel uses as they are found (a process sometimes called "optimistic concurrency control").

Insert Figure 2 about here

The use of a market-like task assignment mechanism in the Linux process is consistent with predictions of the effect of the more extensive use of ICT. ICT makes it easier to gather information about available tasks and resources and to distribute the decision about which resources to use for a particular task. At a macro level, Malone, Yates and Benjamin (1987) suggest that decreased coordination costs favour more extensive use of markets, which usually have lower costs but require more coordination activities, over vertical integration, which makes the opposite trade-off.

Resource-resource

The final type of dependency in the typology is resource-resource dependencies. In the bug fixing process, these arise because modules of code may be interrelated, meaning that a change in one module can require changes in another. In principle, it should be easy to detect dependencies automatically by examining the code, as was done by Schach et al. (2003). In the proprietary software development company, however, there seemed to be no reliable mechanical means to determine the interactions between different modules. Instead, dependencies are tracked by manually tracking documents. The set of routines and data provided by a module make up what is called the interface to that module. Different interfaces are provided for different classes of users. Customer interfaces are described in published manuals and are therefore rarely changed. Service interfaces are provided for use by developers of other parts of the system software and are described in formal documents, called external specifications, which are circulated within the company but usually not to customers. Interfaces intended for use only within a single development group are described in an informally circulated internal specification, if they are documented at all.

Copies of manuals and external specifications are kept in a documentation library; internal specifications are maintained only by their developer. Programmers who request a document from the document library are tracked so they can be informed of any changes to the document. At the time of my study, there were 800-900 documents in the library, and about 1000 document requestors being tracked. A total of 15,000 copies of documents had been distributed.

In practice, however, programmers sometimes borrow a document or copy pieces of someone else's code and therefore do not realize that they should inform the developer that they are using the interface. Since the documentation subscriber lists are not reliable, the software engineer planning a change identifies the other affected engineers based mostly on their knowledge of the system's interactions and what other developers are doing.

Some coordination problems can be traced to the reliance on a heuristic mechanism to locate dependencies. In particular, because there is less informal communication between divisions, the mechanism does not work very well for dependencies between modules are developed in different divisions. For example, in the organization studied, a word processing program once became the source of mysterious system crashes. It turned out that the word processor's developers had used a very low-level system call that had been changed between releases of the operating system. However, because the word processor was developed in another unit from the operating system, the programmers of the two modules did not communicate. Thus, the developers of the word processor did not know they should avoid the system call nor did the developer of the system call know the word processor developers were using it. In other words, the usual social mechanism for managing the dependencies between modules failed, leading to the problems.

In the Linux environment, it is not clear how dependencies between modules are handled. A key goal of most FLOSS development efforts is to reduce the number of inter-module linkages since they make it harder for developers to understand and work with the code (Reis & Fortes, 2002). Still, there are reports that degree of linkage in the Linux kernel are increasing, raising concerns about maintainability (Schach et al., 2003). Understanding the nature of dependencies between modules and their management should be a fruitful area for future research on FLOSS development.

For *sharing information resources*, communications and database technologies may automate the necessary coordination mechanisms. For example, coordination is necessary if multiple tasks use common information stored on paper (a shared resource dependency). It may therefore be desirable to have a single individual handle all the data, to simplify the coordination. For example, a paper-based conference room schedule is usually kept in one central location because of the possibility of conflicting reservations and the prohibitive cost of updating copies every time a reservation is made. Data such as customer accounts or credit information are often handled similarly, resulting in specialization of actors based on their access to information. Database and communications systems enable multiple workers to access and make changes to data. By empowering workers and reducing the need for specialization, ICT can change the basis for assigning tasks. For example, if all workers are equally capable of performing a task, then tasks can be assigned on criteria such as workload or the customer involved, rather than on availability of data or specialization. Such a change was made to the Citibank letter of credit process when a group of specialists, each performing a single step of the process, were replaced by generalists who handle the entire process for particular customers (Matteis, 1979).

Conclusion

To conclude this paper, I consider the implications for a coordination theory approach for process transformation, discuss limitations of the approach and present ideas for future research.

Implications of Coordination Analysis for Process Transformation

Engineering change provides a microcosm of coordination problems and mechanisms to solve them. Successful implementation of a change requires management of numerous dependencies among tasks and resources. A variety of mechanisms are used to manage these dependencies. For example, the possibility of duplicate tasks may be ignored or may be investigated before engineers attempt to solve the problem. Dependencies between tasks and the resources needed to perform them are managed by a variety of task assignment mechanisms, such as managerial decision-making based on expertise or workload; those between modules of the system, by technological coordination mechanisms, such as source control systems. The choice of coordination mechanisms to manage these dependencies results in a variety of possible business processes, some already known (such as change ownership) and some novel (such as voluntaristic selection of work by developers). The relative desirability of mechanisms is likely to be affected by the use of ICT. For example, the use of a computer system such as Bugzilla makes it easier to find existing solutions to a problem. Such a system could reduce both duplicate effort and coordination costs.

The software change process has interesting parallels in other industries. Despite differences in the products, other engineering change processes studied by this author (Crowston, 1991b) had similar goals, activities, coordination problems and mechanisms. Further afield, there are parallels between diagnosing software bugs to assign them to engineers and diagnosing patients to assign them to medical specialists. An analysis similar to the one presented here might reveal interesting alternatives in the medical diagnosis domain as well.

Limitations of coordination theory analysis

Of course, our analysis has several limitations. Coordination theory, like all theories, is a simplification of the complexity of real processes in order to highlight possibilities for transformation. The single example presented here demonstrates the potential of this approach. A coordination theory analysis can identify important dependencies or coordination problems that must be addressed in some way for a process to be successful. However, the suggestions of the analysis need to be tempered by consideration of omitted factors. The technique focuses on how tasks are performed, rather than how employees are motivated to perform, come to understand their jobs or develop shared cultures. For example, a lower mechanism cost does not mean that that mechanism is always better or should be implemented. As mentioned above, market-like task assignment mechanisms may have certain cost benefits, but are also susceptible to agency problems that must be addressed if they are to succeed. Rather than dictating what must be done, the analysis suggests possibilities for transformation that an informed manager can consider and modify to fit the particulars of the organization.

Said alternately, coordination theory does not make strong predictions about what should happen to any single organization that implements a new communication system, although it does suggest what will happen in aggregate (Malone et al., 1987). Rather than the specific accuracy of its predictions, therefore, an appropriate test for the theory is its utility for organization designers. Coordination theory is a success if those attempting to understand or redesign a process find it useful to consider how various dependencies are managed and the implications of alternative mechanisms. As an example, we have used these techniques to compile a handbook of business processes at a variety of levels and in different domains (Malone *et al.*, 2003; Malone et al., 1999). Coordination theory makes the handbook feasible by providing a framework for describing more precisely how processes are similar and where they differ. Managers or consultants interested in redesigning a process can consult the handbook to identify likely alternatives and to investigate the advantages or disadvantages of each. A further advantage of this approach is its ability to suggest new processes by navigating through the dual hierarchies, abstraction and composition. As well, the work reviewed above on modelling techniques shows that coordination can be incorporated into many modelling techniques.

Future Research

A process transformation agenda suggests several additional projects for future research. First, development of the process handbook and general use of a coordination-theory analysis require more rigorous methods for recording processes and identifying dependencies in organizations. There are already many techniques for data collection that are relevant, but none focus explicitly on identifying dependencies. Researchers affiliated with the handbook project have proposed an approach that relies on basic techniques of ethnographic interviewing and observation to collect data and activity lists to identify dependencies and coordination mechanisms (Crowston & Osborn, 2003).

A second question is how to classify different coordination processes. The initial approach to this problem was to list coordination processes by the dependency they addresses. Goethals et al. (2005) extend the analysis given here, while Malone et al. (1999) take this approach further by proposing a hierarchy of coordination processes from general to specific. Other authors have proposed different organizations. For example, Etcheverry, Lopisteguy and Dagorret (2001a, 2001b) propose a catalogue of coordination patterns. A pattern is defined as "a solution to a problem in a given context", so the catalog is organized by coordination contexts. More work could be done to bring together and organize the mechanisms that have been studied.

A third question is about the generality of coordination mechanisms. Most of the work applying the typology of coordination mechanisms has assumed rather than tested the generality of the mechanisms. Nevertheless, the list of mechanisms does seem to have been useful in a variety of settings.

A final question is how to analyze specific coordination practices, e.g., resource allocation. Malone and Crowston (1994) asked, “Can we characterize an entire ‘design space’ for solutions to this problem and analyze the major factors that would favor one solution over another in specific situations?” Most applications of coordination theory are not very explicit about evaluation or factors that make particular coordination mechanisms more or less desirable. There has been some work addressing specific metrics for coordination. For example, Frozza & Alvares (2002) offer a list of criteria for comparing mechanisms: predictivity, adaptability, action control, communication mode, conflicts, information exchange, agents, applications, advantages and disadvantages. Albino, Pontrandolfo and Scozzi (2002) develop the notion of coordination load, “a quantitative index that measures the effort required to properly coordinate a given process”, based on an analysis of the workflow in the process. The goal of this index is to allow a comparison of alternative coordination modes. Nevertheless, it is clear that we are far from characterizing the design space for any of the identified dependencies or coordination mechanisms.

Even in its current stage of development though, coordination theory seems to provide a much needed underpinning for the study and design of new business processes. The result of these efforts will be a coordination-theory based set of tools for analysts and designers, thus enabling business process transformation.

References

Abbott, A. (1992). From causes to events: Notes on narrative positivism. *Sociological Methods and Research, 20*(4), 428–455.

Abell, P. (1987). *The syntax of social life: The theory and method of comparative narratives*. New York: Clarendon Press.

Albino, V., Pontrandolfo, P., & Scozzi, B. (2002). Analysis of information flows to enhance the coordination of production processes. *International Journal of Production Economics, 75*, 7–9.

Alho, K., & Sulonen, R. (1998). *Supporting virtual software projects on the Web.* Paper presented at the Workshop on Coordinating Distributed Software Development Projects, 7th International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE '98).

Barr, J. (2002, 11 February). Linus tries to make himself scale. *LinuxWorld.com* Retrieved 20 March, 2005, from http://www.linuxworld.com/story/32722.htm

Britton, L. C., Wright, M., & Ball, D. F. (2000). The use of co-ordination theory to improve service quality in executive search. *Service Industries Journal, 20*(4), 85–102.

Crowston, K. (1991a). Modelling coordination in organizations. In M. Masuch & G. Massimo (Eds.), *Artificial intelligence in organization and management theory*. Amsterdam: Elsevier.

Crowston, K. (1991b). *Towards a coordination cookbook: Recipes for multi-agent action.* Unpublished doctoral dissertation, MIT Sloan School of Management.

Crowston, K. (1997). A coordination theory approach to organizational process design. *Organization Science, 8*(2), 157–175.

Crowston, K. (2003). A taxonomy of organizational dependencies and coordination mechanisms. In T. W. Malone, K. Crowston & G. Herman (Eds.), *The process handbook* (pp. 85–108). Cambridge, MA: MIT Press.

Crowston, K., & Kammerer, E. (1998). Coordination and collective mind in software requirements development. *IBM Systems Journal, 37*(2), 227–245.

Crowston, K., & Osborn, C. S. (2003). A coordination theory approach to process description and redesign. In T. W. Malone, K. Crowston & G. Herman (Eds.), *Organizing business knowledge: The MIT process handbook*. Cambridge, MA: MIT Press.

Dennett, D. C. (1987). *The intentional stance*. Cambridge, MA: MIT Press.

Etcheverry, P., Lopisteguy, P., & Dagorret, P. (2001a). Pattern-based guidelines for coordination engineering. In *Database and expert systems applications* (Vol. 2113, pp. 155-164).

Etcheverry, P., Lopisteguy, P., & Dagorret, P. (2001b). Specifying contexts for coordination patterns. In *Proceedings of modeling and using context* (Vol. 2116, pp. 437-440).

Frozza, R., & Alvares, L. O. (2002). Criteria for the analysis of coordination in multi-agent applications. In *Coordination models and languages, proceedings* (Vol. 2315, pp. 158-165).

Galbraith, J. R. (1977). *Organization design*. Reading, MA: Addison-Wesley.

Glance, D. G. (2004). Release criteria for the Linux kernel. *First Monday, 9*(4).

Goethals, F., De Backer, M., Lemahieu, W., Snoeck, M., & Vandenbulcke, J. (2005, 22 June). *Identifying dependencies in business processes.* Paper presented at the Communication and Coordination in Business Processes (LAP-CCBP) Workshop, Kiruna, Sweden.

Herbsleb, J. D., & Grinter, R. E. (1999). Splitting the organization and integrating the code: Conway's law revisited. In *Proceedings of the international conference on software engineering (ICSE '99)* (pp. 85–95). Los Angeles, CA: ACM.

Kidder, L. H. (1981). *Research methods in social relations* (4th ed.). New York: Holt, Rinehart and Winston.

Larson, P. (2004). Kernel comparison: Improvements in kernel development from 2.4 to 2.6. Retrieved 19 March, 2005, from http://www-106.ibm.com /developerworks/Linux/library/l-dev26/

Lawler, E. E., III. (1989). Substitutes for hierarchy. *Organizational Dynamics, 163*(3), 39–45.

LeClaire, J. (2005, 22 April). Torvalds adopts new programming management system. *www.LinuxInsider.com* Retrieved 25 July, 2005, from http://www.macnewsworld.com/story/42548.html

Malone, T. W., & Crowston, K. (1994). The interdisciplinary study of coordination. *Computing Surveys, 26*(1), 87–119.

Malone, T. W., Crowston, K., & Herman, G. (Eds.). (2003). *Organizing business knowledge: The MIT process handbook*. Cambridge, MA: MIT Press.

Malone, T. W., Crowston, K., Lee, J., Pentland, B., Dellarocas, C., Wyner, G., et al. (1999). Tools for inventing organizations: Toward a handbook of organizational processes. *Management Science, 43*(3), 425–443.

Malone, T. W., Yates, J., & Benjamin, R. I. (1987). Electronic markets and electronic hierarchies. *Communications of the ACM, 30*, 484–497.

March, J. G., & Simon, H. A. (1958). *Organizations*. New York: John Wiley and Sons.

Matteis, R. J. (1979). The new back office focuses on customer service. *Harvard Business Review, 57*, 146–159.

McKelvey, B., & Aldrich, H. (1983). Populations, natural selection and applied organization science. *Administrative Science Quarterly, 28*, 101–128.

Mohr, L. B. (1982). *Explaining organizational behavior: The limits and possibilities of theory and research*. San Francisco: Jossey-Bass.

Moon, J. Y., & Sproull, L. (2000). Essence of distributed work: The case of Linux kernel. *First Monday, 5*(11).

Powell, W. W. (1990). Neither market nor hierarchy: Network forms of organization. *Research in Organizational Behavior, 12*, 295–336.

Raymond, E. S. (1998). The cathedral and the bazaar. *First Monday, 3*(3).

Reis, C. R., & Fortes, R. P. d. M. (2002). An overview of the software engineering process and tools in the Mozilla project. Retrieved 1 March, 2005, from http://opensource.MIT.edu/papers/reismozilla.pdf

Schach, S. R., Jin, B., Wright, D. R., Heller, G. Z., & Offutt, A. J. (2003). Maintainability of the Linux kernel. Retrieved 14 Dec, 2003, from http://www.vuse.vanderbilt.edu/%7Esrs/preprints/Linux.longitudinal.preprint.pdf

Shankland, S. (2005, 20 April). Torvalds unveils new Linux control system. *CNET News.com* Retrieved 25 July, 2005, from http://news.zdnet.com/2100-3513_22-5678651.html

Tushman, M., & Nadler, D. (1978). Information processing as an integrating concept in organization design. *Academy of Management Review, 3*, 613–624.

Wayner, P. (2000). *Free for all*. New York: HarperCollins.

Wheeler, D. A. (2005). Linux kernel 2.6: It's worth more! Retrieved 18 March, 2005, from http://www.dwheeler.com/essays/Linux-kernel-cost.html

Figures and Tables

Table 1. A typology of dependencies and associated coordination mechanisms from (Crowston, 2003).

Task uses resource

1. determine needs
2. identify resources
    - ads
    - prepared list
    - only one resource
3. collect information on resources
    - by bidding
    - manager knows
4. pick best
5. do assignment
    - mark resource in use
6. manage flow dependencies from acquiring resource to using resource

Task requires multiple resources simultaneously

1. pre-assign resources to simplify coordination problem
2. manage dependency on the fly
    - avoid or detect and resolve deadlock
    - detect and resolve starvation

Sharing: Multiple tasks use the same resource

- ensure same version of sharable resources
    - destroy obsolete versions
    - copy master prior to use
    - check versions prior to use
    - detect and fix problems after the fact
- schedule use of non-shareable but reusable resources
    1. check for conflict before using and then mark the resource as in-use
    2. manage flow of resource from one task to another
- allocate non-reusable resources
    - divide the resource among the tasks
    - abandon one task
    - get more resources

Flow: One task uses a resource created by another

1. usability (i.e., the right thing)
    - user adapts to resource as created
    - creator gets information from user to tailor resource
    - 3rd party sets standard, followed by both producer and consumer
2. prerequisite (i.e., at the right time)
    - producer produces first
        - follow plan
        - monitor usage
        - wait to be asked

        - standard reorder points
        - when out
        - just-in-time
    - consumer waits until produced
        - monitor
        - be notified
3. accessibility (i.e., in the right place)
    - physical goods
        - truck
    - information
        - on paper
        - verbally
        - by computer

Common output: Multiple tasks create the same output

1. Detect common output
    - database of known problems
2. Manage common outputs
    - effects overlap or are the same
        - eliminate one task (manage shared resource)
        - merge tasks take advantage of synergy
    - effects are incompatible
        - abandon one
        - don't try to achieve them at the same time

Composition of tasks

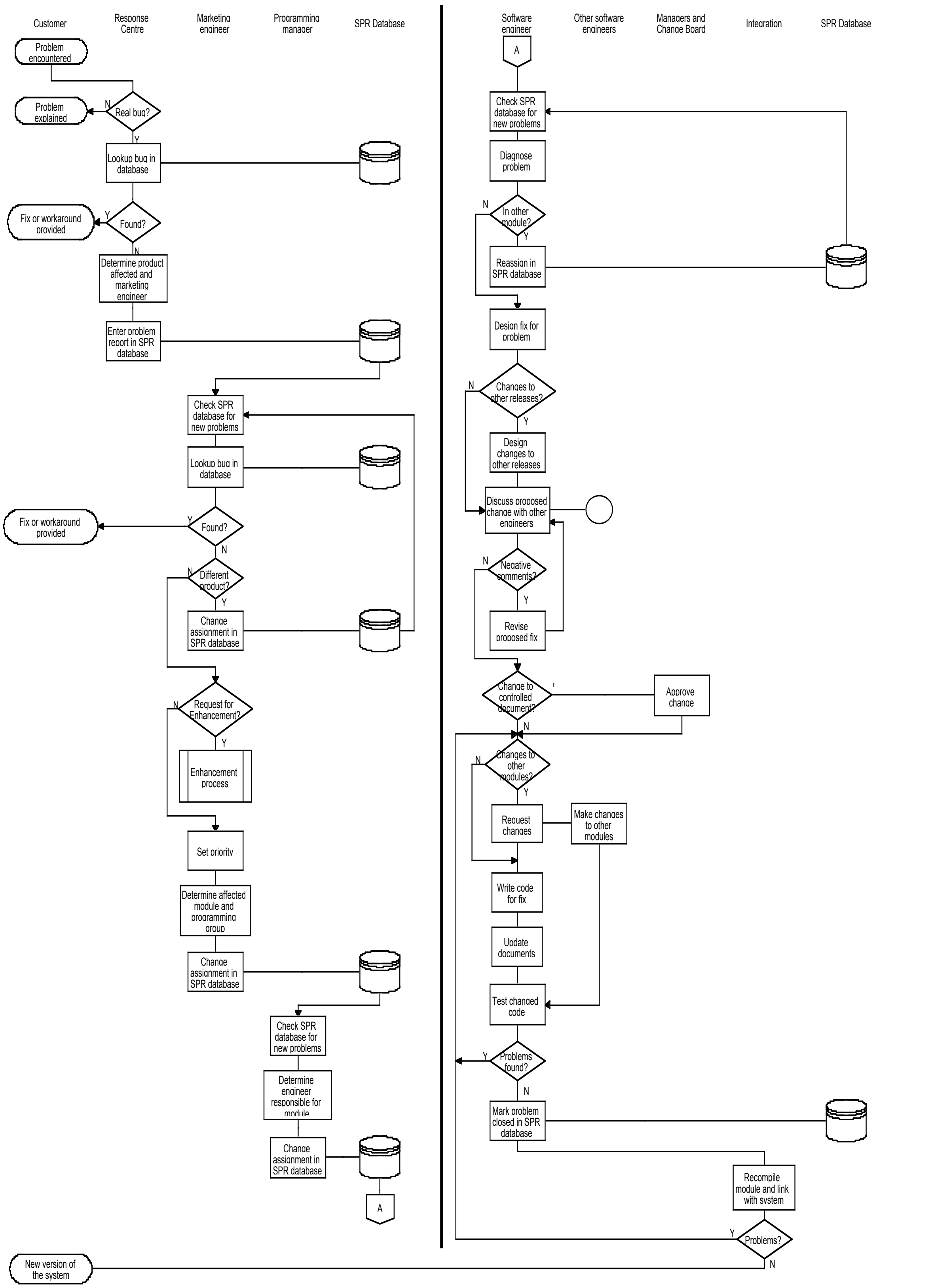
- choose tasks to achieve a given goal (a planning problem)

Composition of resources

- trace dependencies of between resources to determine if a coordination problem exists

Note: Dependencies are shown in bold. Numbered items are components of the coordination mechanism for managing the given dependency. Bulletted items are alternative mechanisms or components of the mechanism for the given dependency.

Figure 1. Flowchart of proprietary software bug fixing process (from Crowston, 1997).



Customer
Response Centre
Marketing engineer
Programming manager
SPR Database
Software engineer
Other software engineers
Managers and Change Board
Integration
SPR Database
Problem encountered
Real bug?
Problem explained
Lookup bug in database
Found?
Fix or workaround provided
Determine product affected and marketing engineer
Enter problem report in SPR database
Check SPR database for new problems
Lookup bug in database
Found?
Fix or workaround provided
Different product?
Change assignment in SPR database
Request for Enhancement?
Enhancement process
Set priority
Determine affected module and programming group
Change assignment in SPR database
Check SPR database for new problems
Determine engineer responsible for module
Change assignment in SPR database
A
Check SPR database for new problems
Diagnose problem
In other module?
Reassign in SPR database
Design fix for problem
Changes to other releases?
Design changes to other releases
Discuss proposed change with other engineers
Negative comments?
Revise proposed fix
Change to controlled document?
Approve change
Changes to other modules?
Request changes
Make changes to other modules
Write code for fix
Update documents
Test changed code
Problems found?
Mark problem closed in SPR database
Recompile module and link with system
Problems?
New version of the system
Y
N

NB: the flow continues from the bottom right of the chart to the top left; the activities on the right follow rather than overlap those on the left.

Table 2. Activities and coordination mechanisms in the proprietary bug fixing process (from Crowston, 1997).

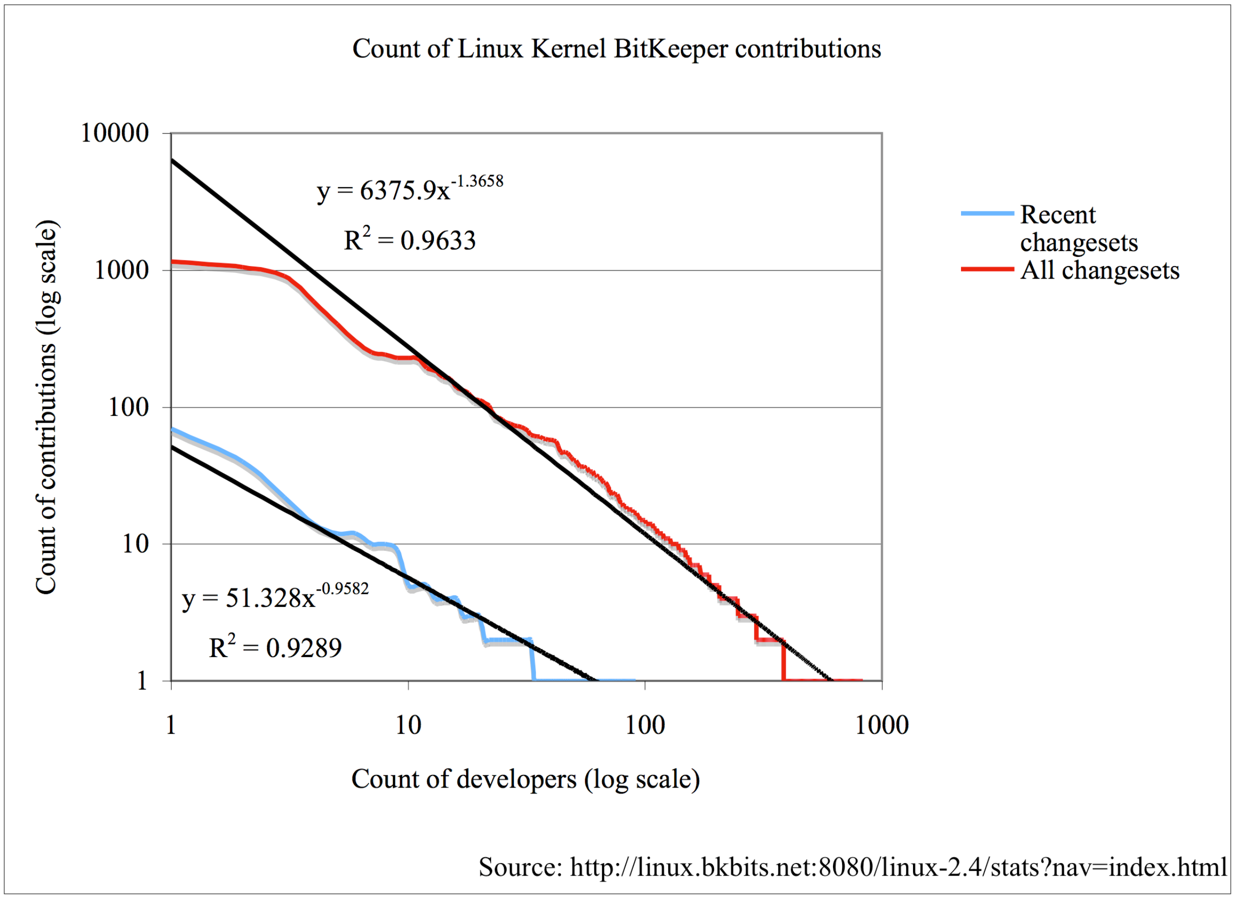
Actor	Activity	Dependency managed between…
Customer	**find a problem while using system**	
	report problem to response centre	problem fixing task and capable actor
Response Centre	look for bug in database of known bugs; if found, return fix to customer and stop	problem fixing task and duplicate tasks
	attempt to resolve problem	
	refer hardware problems to field engineers	problem fixing task and capable actor
	if problem is novel, determine affected product and forward bug report to marketing engineer	problem fixing task and capable actor
Marketing Engineer	look for bug in database of known bugs; if found, return fix to customer	problem fixing task and duplicate tasks
	request additional information if necessary	usability of problem report by next activity
	attempt to reproduce problem	usability of problem report by next activity
	attempt to find workaround	
	set priority for problem	problem fixing task and actor's time
	if the report is actually a request for an enhancement , then treat it differently	
	determine affected module	task and resources required by tasks
	if unable to diagnose, forward to SWAT Team	
	if bug is in another product, forward to appropriate product manager	problem fixing task and capable actor
	forward bug report to manager of group responsible for module	
Programming Manager or Designate	determine engineer responsible for module and forward bug report to that engineer	problem fixing task and capable actor
Software Engineer	pick the report with the highest priority, or the oldest, or the one you want to work on next	problem fixing task and actor's time
	diagnose the problem	
	if the problem is in another module, forward it to the engineer for that module	problem fixing task and capable actor
	design a fix for the bug	
	check if change is needed in other releases and make the change as needed	problem fixing task and capable actor
	send the proposed fix to affected engineers for their comments; if the comments are negative, then revise the bug and repeat the process	two modules
	if the change requires changes to a controlled document, then send the proposed change to the various managers and the change review board for their approval	management of usability task and capable actor
Managers	approve the change	usability of fix by next activity
Software engineer	**write the code for the fix**	
	determine what changes are needed to other modules	task and subtasks needed to accomplish it
	if necessary, ask the engineers responsible for the other modules to make any necessary changes	problem fixing task and capable actor
	test the proposed fix	usability of fix by next activity
	send the changed modules to the integration manager	task and capable actor
	release the patch to be sent to the customer	transfer to customer
Integration	check that the change has been approved	usability of fix by integration activity
	recompile the module and link it with the rest of the system	
	test the entire system	usability of entire system by next activity
	release the new software	transport to customers

Activities in bold are those logically necessary to respond to problem reports.

Table 3. Activities in the generalist form of task assignment (from Crowston, 1997).

Agent	**Activity**	**Dependency managed between…**
Customer	Use system, find a bug	
	report bug to response centre	problem fixing task and capable actor
Response Centre	lookup bug in database of known bugs; if found, return fix to customer and stop	problem fixing task and duplicate tasks
	determine affected product and forward bug report to marketing engineer	problem fixing task and capable actor
Marketing Engineer	lookup bug in database of known bugs; if found, return fix to customer and stop	problem fixing task and duplicate tasks
	attempt to reproduce the bug—part of diagnosing it	
	determine affected module; if can't diagnose, forward to SWAT Team; if other product, forward to appropriate product manager; put bug report in the queue of bugs to work on	problem fixing task and capable actor
Software Engineer	start work on the next bug in the queue	problem fixing task and actor's time
	diagnose the bug	
	if it's actually an enhancement request, then treat it differently	
	design a fix for the bug	
	if the change requires changes to a controlled document, then send the proposed change to the various managers and the change review board for their approval	management of usability task and capable actor
Managers	approve the change	usability of fix by subsequent activities
Software engineer	check out the necessary modules; if someone else is working on them, then wait or negotiate to work concurrently	problem fixing task and other tasks using the same module
	write the code for the fix	
	test the proposed fix	usability of fix by subsequent activities
	send the changed modules to the integration manager; check in the module	
Integration	check that the change has been approved	usability of fix by subsequent activities
	recompile the module and link it with the rest of the system	integration
	test the entire system	usability of entire system by next activity
	release the new software	

Figure 2. BitKeeper contributions vs number of developers.



Changes per developer follow a power law distribution: A small number of developers contribute many changes, while many developers contribute only a few changes. Black lines are fitted power law trend lines.

Appendix. Example of email bug report and patch in the Linux kernel (from http://lkml.org/lkml/2003/5/28/204).

```
Subject     2.4 bug: fifo-write causes diskwrites to read-only fs !
From  R
Date  Wed May 28 2003 - 13:02:20 EST

Hi all,

It turns out that Linux is updating inode timestamps of fifos (named pipes)
that are written to while residing on a read-only filesystem. It is not only
updating in-ram info, but it will issue *physical* writes to the read-only fs
on the disk !

I use a CompactFlash in an embedded application with a read-only root-fs on
it. There are several processes that communicate with each other via fifos.
This bug in Linux causes frequent writes to my CF and will shorten it's
lifetime enormously ..

I've posted a report on the "mysterious writes" before:
(http://www.ussg.iu.edu/hypermail/Linux/kernel/0303.2/1753.html)
(incorrectly) linking it to a possible bug in O_SYNC. Nothing came out of it.

But now I've completely tracked down the bug (logging all diskaccesses and
seeing it undoubtly write in disksectors containing time-stamp info of
fifo's). Looking back it would have been easier to prove that something is
wrong: the modified time-stamps survive power-cycles. This is not supposed to
happen on a read-only fs.

I've tried reading the kernel source to find where the bug lives, But I'm not
too familiar with it. Anyone out there who can pin it down ?

greetings, R

Sysinfo:
--------
- various 2.4 kernels including RH-2.4.20-13.9, but also straight 2.4(ac)
ones.
- CompactFlash (= IDE disk)
- Geode GX1 CPU (i586 compatible)
```

```
Date  Wed, 28 May 2003 15:17:38 -0400 (EDT)
From  D
Subject     Re: 2.4 bug: fifo-write causes diskwrites to read-only fs !

How does it 'know' it's a R/O file-system? Have you mounted it R/O, mounted
it noatime, or just taken whatever you get when you boot from a ramdisk?

FYI, I created a FIFO with mkfifo, remounted the file-system R/O, executed
`cat` with it's input coming from the FIFO, and then waited for a few
minutes. I then wrote to the FIFO. The atime did not change with 2.4.20.

Cheers, D
```

```
Subject     Re: 2.4 bug: fifo-write causes diskwrites to read-only fs !
Date  Wed, 28 May 2003 21:34:17 +0200
From  R

Hi D,

The kernel has the "ro" commandline-parameter. There is no remount after the
system boots. "touch /bla" gives a read-only fs error.

> FYI, I created a FIFO with mkfifo, remounted the file-system
> R/O, executed `cat` with it's input coming from the FIFO, and
> then waited for a few minutes. I then wrote to the FIFO.
> The atime did not change with 2.4.20.

Just did the same here (on my workstation). And the times *did* change ..
More precisely: the "modification" & "change" were updated, the "access" time
remained unchanged.

RH9, kernel-2.4.20-13.9

greetings, R
```

```
Date  Wed, 28 May 2003 16:22:47 -0400 (EDT)
From  D
Subject     Re: 2.4 bug: fifo-write causes diskwrites to read-only fs !

Okay. I can now verify the problem. There are two problems as this script
will show:

Script started on Wed May 28 16:10:13 2003
# cat xxx.c
#include <stdio.h>
… test program listing omitted …
# gcc -O2 -o xxx -Wall xxx.c
# ./xxx
atime = 3ed51750
mtime = 3ed517c5
ctime = 3ed517c5
…test program output omitted …

# >/alt/foo
bash: /alt/foo: Read-only file system
# exit
exit
Script done on Wed May 28 16:11:12 2003

As you can clearly see, access time (atime) is not changed. However, both
ctime and mtime are both changed with every FIFO access. Since this FIFO is
provably on a R/O file system, nothing should change.

Now, somebody will probably claim that this is the correct POSIX defined
behavior <sigh> so you might have to make some work-around like use a pipe or
socket instead of the FIFO??
```

Cheers, D

Subject   Re: 2.4 bug: fifo-write causes diskwrites to read-only fs !
Date Wed, 28 May 2003 22:52:30 +0200
From R

Hi D,

Yeah !, I'm no longer alone .. :-)

Note that the fact that you see the times changing in the fs while it is mounted doesn't imply a problem in itself: serial and tty device nodes get their time-stamps updated too on a read-only fs when they are written. But these changes are in ram only: when you reboot you get the old values back.

But with FIFOs the changes *do* get written out to the read-only fs !

Hmm, wonder what happens if you try it on a real read-only medium like a CDR. Maybe kernel errors/panic ..

> Now, somebody will probably claim that this is the correct
> POSIX defined behavior <sigh> so you might have to make some
> work-around like use a pipe or socket instead of the FIFO??

Seems very stupid to me if POSIX specifies this. I don't have the POSIX spec, but maybe it specifies what "read-only" is supposed to mean somewhere too ..

But let's wait & see .. :-)

greetings, R

Subject   Re: 2.4 bug: fifo-write causes diskwrites to read-only fs !
From S
Date Fri May 30 2003 - 08:27:17 EST

Hi,

On Wed, May 28, 2003 at 10:52:30PM +0200, R wrote:

> I don't have the POSIX spec, but maybe it specifies what "read-only"
> is supposed to mean somewhere too ..

SingleUnix says:

http://www.opengroup.org/onlinepubs/007904975/basedefs/xbd_chap04.html#tag_04_07

"Marks for update, and updates themselves, are not done for files on read-only file systems"

So we're wrong here. Patch below fixes it for me for 2.4.

Cheers, S

```
--- Linux-2.4-odirect/fs/inode.c.=K0004=.orig
+++ Linux-2.4-odirect/fs/inode.c
@@ -1194,6 +1194,24 @@ void update_atime (struct inode *inode)
mark_inode_dirty_sync (inode);
} /* End Function update_atime */

+/**
+ * update_mctime - update the mtime and ctime
+ * @inode: inode accessed
+ *
+ * Update the modified and changed times on an inode for writes to special
+ * files such as fifos. No change is forced if the timestamps are already
+ * up-to-date or if the filesystem is readonly.
+ */
+
+void update_mctime (struct inode *inode)
+{
+ if (inode->i_mtime == CURRENT_TIME && inode->i_ctime == CURRENT_TIME)
+ return;
+ if ( IS_RDONLY (inode) ) return;
+ inode->i_ctime = inode->i_mtime = CURRENT_TIME;
+ mark_inode_dirty (inode);
+} /* End Function update_mctime */
+

/*
* Quota functions that want to walk the inode lists..
--- Linux-2.4-odirect/fs/pipe.c.=K0004=.orig
+++ Linux-2.4-odirect/fs/pipe.c
@@ -230,8 +230,7 @@ pipe_write(struct file *filp, const char
/* Signal readers asynchronously that there is more data. */
wake_up_interruptible(PIPE_WAIT(*inode));

- inode->i_ctime = inode->i_mtime = CURRENT_TIME;
- mark_inode_dirty(inode);
+ update_mctime(inode);

out:
up(PIPE_SEM(*inode));
--- Linux-2.4-odirect/include/Linux/fs.h.=K0004=.orig
+++ Linux-2.4-odirect/include/Linux/fs.h
@@ -201,6 +201,7 @@ extern int leases_enable, dir_notify_ena
#include <asm/byteorder.h>

extern void update_atime (struct inode *);
+extern void update_mctime (struct inode *);
#define UPDATE_ATIME(inode) update_atime (inode)

extern void buffer_init(unsigned long);
```