The Benefits of Pair Programming applied to the Problems in Linux Kernel Development

Scott Thomas
California Polytechnic State University, San Luis Obispo
sbthomas@calpoly.edu

December 9, 2002

Abstract

The Linux kernel is an ever-changing piece of software that is growing at an incredible rate. It is unique in the fact that it is being developed by hundreds of people all around the world. Most development is done by solo programmers, then submitted via a mailing list and incorporated into the next kernel release. Although Linus (the inventor of Linux) reviews the code before it is incorporated into the kernel, for much it is only the second time the code has been reviewed. It is a dangerous thing to incorporate code into Linux that has not been designed and reviewed thoroughly.

Pair programming is the process of two programmers working together on a single task or product. The output of the process will be a single result belonging to both programmers involved. Pair programming is one of the more controversial components of extreme programming and the costs and benefits are not well established. Using existing studies and papers, I organized a table showing the proposed costs and benefits of pair programming.

To determine if pair programming would be valuable in Linux kernel development, I analyzed the costs and benefits advertised by pair programming and applied them to the Linux kernel domain. Using a chart created showing the disadvantages of Linux, I matched each disadvantage with at least one advantage provided by pair programming. Using pair programming to develop the Linux kernel would decrease the number of bugs submitted and increase the readability of the Linux code through simpler designs thought out by a pair of programmers.

Introduction

The Linux kernel has been in development for over 10 years now. It is unique in the fact that no one company takes responsibility of it, nor are people in it to make money from it. Rather, hundreds of computer programmers all over the world have taken the responsibility of adding features, fixing bugs, and making a better kernel for us all. Most of the programmers write Linux kernel code in their spare time. Most of this code is solo programmed (programmed by one developer sitting at his computer) and submitted for addition to the Linux kernel once he or she has deemed it good. Since the code is developed by a single programmer, bugfixes are subject to the age old saying about causing more bugs than fixed. A single programmer with his head in deep tends not to see all the different things the change will affect. Also new features added may not have been thought out to the appropriate length and all affected parts of the kernel may not have been analyzed. The kernel source code has also been patched hundreds if not thousands of times with hacks that are difficult to see and understand. These problems will only get worse with time and need a solution.

Pair programming is programming performed by two or more people. While one programmer is at the keyboard and mouse controlling, the other is observing, thinking about design alternatives, and asking questions. XP insists that all production code be pair programmed. This may sound extremely inefficient at first. Clearly, having two people do the work of one person is a waste of valuable resources. However, programming is not a laborious job, rather it is an intellectual job. The decisions that are made require much thought, and in this situation, two brains are better than one. With two people each being assigned to different tasks, there is less chance of the partners neglecting tests, or forgetting tasks. It is fairly easy for a person working by himself to be distracted and have an extremely low level of productivity. With pair programming frequent email and web surfing breaks are kept to a minimum and the time spent working produces more, high quality output.

The studies which have been performed on pair programming haven't seem to applied to any specific domain of software. They have just been studies done at universities mostly on classroom-based assignments. These studies have advertised many advantages to using pair programming over solo programming. I am interested to see if these advantages are applicable in the Linux kernel development domain. To determine if pair programming can be applied, I will take each problem with Linux and see if I can find a benefit of pair programming that can be applied. Once this analysis has been accomplished I will be able to determine whether or not pair programming is a benefit when applied to Linux kernel development. As a future project, it would be interesting to see if the rest of the extreme programming practices can be applied as well. It would also be interesting to see if implementing pair programming as a requirement in submitting Linux code would be feasible. This paper will not address these issues; rather it will only see if the benefits of pair programming can solve the problems in the Linux kernel development domain.

Linux Kernel Development

What is an operating system?

> "An operating system is a collection of programs that control the use of computer resources, provide standard communication interfaces, and provide for continuous operation of a computer" [1].

The operating system acts as a home for all other software to be run on a computer. Without an operating system, a computer would be an incredible pain to work with, if not impossible. All programs would need to have their own operating systems to run in, in order to access the underlying hardware devices. This would make developing software an even tougher task than it is today. Another feature operating systems provide is the ability to run multiple programs simultaneously. The operating system keeps track of the resources each program uses and can ensure that these resources are not used inappropriately.

A standard operating system consists of four parts: process management, input/output, memory management, and the file system [2]. An explanation of how these for areas are significant follows.

A process is a program that is running underneath the operating system. Many different processes can be run on a computer. The operating system is required to schedule each process according to the scheduling algorithm being used. It is its job to make sure that all processes get the chance to use the CPU. The operating system is also in charge of creating and removing processes dynamically. Lastly, the operating system must allow processes to communicate with each other. This process, called inter-process communication, is a difficult job for an operating system as there are many variables to consider.

Input/output (I/O) is the second major job of an operating system. The operating system is in charge of all I/O devices attached to the computer system. These devices fall into two categories and must be handled accordingly. Block devices are simple devices such as hard disks. Data must be written to and read from these devices in fixed size blocks. Character devices, such as a keyboard, deliver or accept streams of characters without any regard to a block structure. The operating system needs to provide an interface to all running processes that require the use of I/O. There are many problems associated with I/O including deadlocks. A deadlock occurs when two programs are waiting each waiting on a resource the other process has. They will, by default, wait forever, unless the operating system intervenes.

Next, the operating system is in charge of all memory management. Memory is central to the operation of modern computer systems [3]. Each process that is running requires its own set of memory. The entire program need not be in memory at the same time, but the next few instructions should be to maximize performance. There are many different algorithms available to maximize performance. It is up to the implementer of the operating system as to which he chooses to use. The goal of memory management is to

provide all processes with whatever they require, keeping the no or please wait response to a minimum.

Lastly, the operating system is in charge of interacting with a file system. The file system is the most visible part of an operating system [2]. The file system consists of two parts: files and directories. The files are the collections of blocks that contain data. The operating system is in charge of keeping track of all files on a disk and making sure they are saved appropriately. The directories are used as a way of sorting and organizing the files. A user will use directories to sort files that contain similar data. The operating system must know the proper method of interaction with the current file system, as the file system may be taken somewhere else (ex. Floppy disk) and will need to be readable there.

History of Linux

Linux was inspired by an educational operating system called Minix [4]. The Minix operating system was developed by Andrew Tanenbaum, as a purely educational, instructional operating system. Students in the past had been studying the source code for Unix version 6 in operating systems classes across the US [2]. However, with version 7, AT&T decided to market Unix and thus releasing the source code was no longer advantageous. This caused many universities to stop teaching the application and implementation of operating systems, and revert to only teaching the theory. Tanenbaum did not feel this was adequate and in [2] introduced his new operating system called Minix. Minix was designed with the number one goal being: teach students about operating systems. To avoid possible copyright problems, Tanenbaum did not use a any code from AT&T's distribution of Unix. Keeping the fact that this was to be an instructional operating system in mind, Tanenbaum made the operating system modular, as well as making it easy to read and understand by using over 3000 comments inline with the code [2].

Many people started to take an interest in Minix and wanted to use it as more than an educational operating system. People were requesting many features that would definitely improve Minix, but would make the operating system larger and more complex. Tanenbaum had to say no, in order to keep his operating system small enough to be taught in a one-semester course [5]. People kept complaining and Tanenbaum kept saying no until someone finally decided to design a new operating system that would implement all the features lacking in Minix.

The idea of Linux first sprang up in Finland in 1991. A student at the University of Helsinki named Linux Torvalds, decided he would implement a version of Minix with all the features people wanted. His goal was to make a Unix clone, without using any of the existing AT&T code. He tried to make modifications to the Minix code, but determined it was too much of an academic operating system and started writing this new operating system from scratch [4]. This new operating system would be called Linux, a cross between Minix and Linus. The biggest decision Linus made was to make the Linux source code freely available over the Internet. Linux has evolved rapidly with Linus still remaining the lead developer. The following table from [4] shows the progress of Linux over the last 10 years.

August 1991	Kernel Version 0.01
October 1991	Official announcement 0.02
November 1993	First Slackware Distribution with kernel 0.99
March 1994	Kernel Version 1.0 released
June 1995	First port to the Alpha architecture
June 1996	Kernel Version 2.0 released
October 1996	Debian Linux is used on the space shuttle in orbit (on an IBM laptop)
January 1999	Kernel Version 2.2 released
January 2001	Kernel Version 2.4 released

Table 1: Linux Kernel Progress

If the current trend continues, kernel version 3.0 should be out about 2005 or so. With each distribution, the size of the code-base, and the number of features included has

increased. The following chart shows the number of lines of code in each release. The data for this chart comes from [6] and [5]

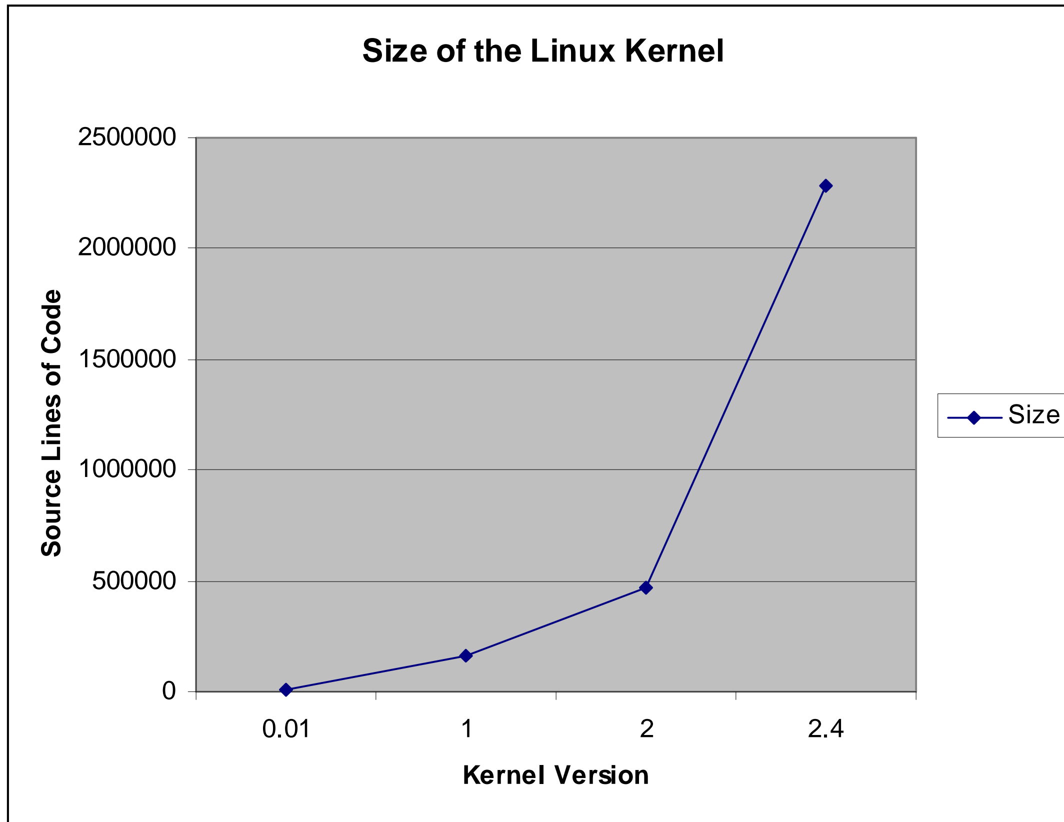

Chart 1: Size of the Linux Kernel

Advantages of Linux

The following advantages of Linux are taken from [7]. They will attempt to explain what is good about Linux and why people want to use it. First, Linux is free. There is no charge for the source code and it can be obtained online at any time. There are also no restrictions placed on the software. A person is free to change whatever they want so long as they include the source with anything they distribute.

Second, Linux is popular. The word "Linux" is being flashed more and more now, and people all over are starting to become interested in what it has to offer. There are also many different flavors (distributions) as well as ports that will run on many different

processors. Linux is extremely powerful. It can run on the slowest 386 found and clunk along better than windows. It can also be run on extremely fast servers replacing the high cost of Unix servers.

Linux is good quality software and runs high-quality software applications. With hundreds of developers all around the world writing the Linux kernel and software to run on it, this creates a huge base of software. If there is a task that needs to be done on a computer, the probability is high that there is already software available for Linux. Linux has full Unix features. The majority of system calls available on a standard Unix machine are available and behave similarly in Linux. It is also compatible with the graphical user interface X-windows.

Next, Linux is compatible with the different Microsoft operating systems. Linux can read and write from FAT partitions, and (currently) has read-only support for NTFS partitions. It can also access Windows shares and be seen in a Windows computer's network neighborhood. The Linux kernel itself is lightweight and can run on a very basic system. However, many distributions come packed with gigabytes of extra applications that can be installed if desired. The choice is all up to the user to determine what he or she needs or wants.

There are many support options available for Linux. Much information is available freely on the web, as well as companies who, for a fee, will provide technical support. Lastly, Linux is well documented. There are tons of books available for purchase, as well as many support sites online. A Linux documentation project has been started and contains many tutorials on how to perform different operations on a Linux machine.

Disadvantages of Linux

Despite all its advantages, Linux is not perfect and contains its share of downfalls. Most of these downfalls stem from the way Linux is developed. First, a single company does not develop Linux. When a company is contracted a software product, they can control

all aspects of the process. They choose who works on what part of the software as well as what features are incorporated into the software. Linux is in a whole different realm of software. First, there are no real customers. The customer is everyone in the world who might ever need or want to use the Linux operating system. There is also no distinct set of requirements. There are the basic requirements of an operating system, but beyond that it is up to the developers as to what is included. Therefore it can be extremely hard to control which direction Linux progresses in.

Without control of who is adding features to the Linux kernel, it is difficult to establish the quality of the code that is submitted. The code could come from someone who just started learning how to program, and wanted a method to read in a string of unknown length, or it could be a feature from someone who has programmed operating systems their entire life. This all leads to bugs being submitted in new features added to the Linux kernel. Most people program by themselves and it is difficult to see what things they are affecting around them.

When people see a bug in the kernel, they most likely add a quick fix. Many times this quick fix doesn't fix the problem, or adds many other new problems. As seen in the following clip from a recent kernel changelog [8]:

```
Kai Germaschewski <kai@tp1.ruhr-uni-bochum.de>:
  o ISDN: Fix error path in isdn_ppp.c

Kai Germaschewski <kai@tp1.ruhr-uni-bochum.de>:
  o ISDN: Fix the fix
```

Code is submitted and sometime is briefly reviewed, but almost any code can get into the kernel if it looks like the author knows what he or she is doing. There is no way to tell how much thought or design has gone into the code patch being submitted.

What one programmer sees as the ultimate simplest solution to a problem, may be completely foreign to another. When programmers write code alone, they try and make it the sleekest code possible. Most programmers when programming alone don't bother to document what they are doing because documentation isn't fun. This all leads to difficult

to read and understand code being incorporated into the Linux kernel. This problem is not evident immediately, rather when someone needs to make a change, they cannot understand what the previous code is trying to do. A good example of this comes from fork.c found in the kernel subdirectory of the Linux kernel [9].

```
111     if(++last_pid >= next_safe) {
112         if(last_pid & 0xffff8000)
113             last_pid = 300;
114         next_safe = PID_MAX;
115     }
```

This snippet of code is just an example of some of the really hard to understand pieces of code found in the Linux kernel. This probably made complete sense to the person programming, however for the general programmer, this is difficult if not impossible to decipher. There are magic number used, which most likely no one knows what they mean.

Another problem with Linux development is that the people working on the project are spread across the world. In common software projects in a company at least a group of developers know each other and work close together. If one of the programmers has a question about some of the code or a certain design he is creating, he has to find someone who is willing to spend time and help him. A more likely scenario is that the programmer will just guess as to what the code does and design his solution around the guess. This can cause large problems in the future if the accusation is incorrect. Also, the overall amount of people desiring to work on the Linux kernel is actually decreasing according to [5].

Another tough thing about the Linux kernel is that there is an extreme amount of code. Who knows how precise the existing code is. The solutions designed have been installed to get Linux working. Not much time has been spent on making the code smaller, or easier to read and understand. Because there is so much code, it is hard for someone new to get involved in hacking the Linux kernel. There are so many different places to start, each with an incredibly steep learning curve.

Summary

The following table shows the advantages and disadvantages of Linux. This table will be used again when we try and apply pair programming to solve the problems with the Linux kernel.

Advantages	**Disadvantages**
Free	No company in charge
Popular	Bugs
Powerful	Patches may not be reviewed
Good quality, high quality software apps	Difficult to understand code
Unix features	Programmers physically spread out
Highly compatible with Microsoft OS	Extremely large amount of code
Lightweight	Hard to incorporate new developers
Large or small installation possible	Number of developers working on kernel decreasing
Well Supported	
Well Documented	

Table 2: Advantages and Disadvantages of Linux

Pair Programming

History of pair programming

Pair programming is not a new process. People have been programming together for as long as programming has been around. Dijkstra even reports that he and Zonneveld "combated the writing errors by coding together" [10]. Dijkstra learned how to program in the 60s and quickly realized that by working in pairs, many errors never make it past the first stages of software engineering. Constantine in his book *Constantine on Peopleware* also describes a type of pair programming which he calls "dynamic duo" development [11]. Though, programmers working together has existed for a long time, it had not been formalized until recently.

What is Extreme Programming?

Pair programming is a key element in the software engineering process known as eXtreme Programming (XP). According to the father of XP, Kent Beck, eXtreme programming is “a lightweight, efficient, low-risk, flexible, predictable, scientific, and fun way to develop software” [12]. The XP process is a lightweight process as there is not a lot of paperwork and overhead involved in implementing XP on a project. Efficiency is a goal of XP as it pushes for continuous development. It advocates a plan that has been put together, but says that most of the design will be done on the fly. It stresses not to make decisions now that can be put off until later, counting on the fact that the decision may be removed later on down the line. XP is low-risk as it simply incorporates many of the programming practices that have been around for decades. XP simply puts a fancy name on a package of solid programming practices. Included in the flexible approach is the fact that XP stresses continuous product development. Programmers are encouraged to be designers throughout the software engineering process. The customer is expected to be an active part throughout the development process as XP allows for and encourages change in business needs. XP is also predictable as it relies on continuous updates and feedback from all involved in the design process. Short release times are suggested to allow this feedback to be possible. The process of XP is scientific in the fact it stresses to get something small working first. Once this has been confirmed to work, perform an update or add a feature, then test until all tests have been completed successfully. This is a very structured method for engineering software. Lastly XP is fun. Most programmers’ favorite part of software engineering is writing code to do cool stuff. XP allows the programming to start almost immediately.

There are twelve basic principles of XP [13]. Many of these parts are independent of each other. The parts included on project development can be adjusted on the fly to maximize efficiency. If a certain component of XP isn’t working, take it out, evaluate the problem with the component, solve, and insert it back in. The first step refers to the initial design. The customer writes the desired features each on separate business cards. The development team then takes the cards and estimates the amount of time that will need to

be spent in order to implement each feature. Using the time estimations, the customer then develops an order they would like to see the software implemented in. They order the cards with the most important features on the top. Once the requirements have been determined, implementation begins. XP's second principle is small releases. After the first few features have been implemented, the company should provide a release of the software. Third, each XP programming project has a metaphor associated with it. A system metaphor provides a similar object to the one being created in order to establish the proper vocabulary to be used [14]. The next aspect of XP pushes to find the simplest solution to the problem. XP counts on the fact that the requirements are going to change so don't worry about stuff that is off in the distance. Continuous testing is a major aspect of XP. Before the new feature is implemented, the programmer must write the test cases for the module. This can be frustrating at times, but will pay off in the end. Sixth, refactoring is required in order to use XP. Refactoring is the process of restructuring and re-writing code to simplify design and reduce duplicate code. This can ordinarily be a risky procedure, but XP requires continuous testing and this guards against breaking non-related objects. The seventh point is a collective ownership of all code. This means that there should be no experts in a specific area and all developers should be able to work on all areas of the code. Next, XP stresses continuous integration. All changes should be integrated into the code-base at least once a day. Keeping in mind the test cases need to pass in order to check it in. If this is not possible, then there has been a problem dividing up the features, as they should be small pieces of the project. Ninth, XP specifies a 40-hour workweek. According to XP policies, too much overtime indicates a flaw in the estimation and this should be re-worked. Because the customer is incorporated into the development process, there should always be someone on site who can make decisions regarding the software on behalf of the company. Next XP needs a common coding standard in order to work. The author should not be apparent just from the format of the code. This can be achieved by having people use the same software, setting tab widths, etc. Lastly, one of the most controversial parts of XP is pair programming [15]. XP states that all production code incorporated into the software must be written in pairs. One of the reason's pair programming is so controversial is that it is extremely easy to do wrong.

When all the components of pair programming aren't followed, the entire process falls apart quickly.

What is pair programming?

Pair programming is a process involving two software engineers working together on the same software artifact. This object can be a piece of code, UML diagram, algorithm design, test case, etc. We will refer to the person with current control of the keyboard as the driver. The driver does the "driving." He has control of the mouse and keyboard and the observer should not touch either until the two agree to switch roles.

The driver is not expected to behave or perform differently than he usually would while programming. However, he is expected to listen and discuss with the observer, any comments, complaints, and suggestions. The driver is going to be the person making many mistakes. It can be very difficult to adapt to the position of being the driver. There is constantly someone asking questions or suggesting alternatives. The driver needs to learn not to take the comments personally. He or she needs to realize that the person not at the keyboard will be able to see other things and find things faster on the screen. The driver needs to keep in mind that if the roles were reversed, the situation would be the same [16]. It is just a given fact that the person not typing will be more observant.

The observer plays a key role in pair programming. Without the observer, it wouldn't be pair programming. If you have a crummy observer, you will not receive any of the benefits of pair programming. Now that I have stressed the importance of the pair, we will examine just what the pair can do to be the best possible partner. First, get to know your partner and adjust interruptions accordingly. Some people have habits that will not go away. For example, I cannot press the 'p' key the first time to save my life. I always manage to hit another key first, but press backspace and correct it almost immediately. This is something that the observer should not try and fix. For the most part it doesn't cause any problems and can be ignored. Bringing it up would probably just cause tension between the pair. This is a small exception, but for the most part the observer should

bring up anything that comes to his attention. Anything and everything: If it is being put into the code, it is worth explaining. Syntax errors also may be pointed out, however this is not the main purpose of the observer. The compiler will catch most syntax errors, but the pair should point out errors noticed. The observer needs to be patient with the driver, as the observer will see things easier and faster than the driver. The driver is up to his ears in code, while the observer is standing back a few feet on solid ground. The observer should never take control of the keyboard or mouse until the time has come to switch positions.

There are also common things both the driver and the observer must do. Both must practice “ego-less programming”[17]. Excess ego can cause two huge problems in pair programming. He can choose not to listen to others as he feels that his way is the best. Or the programmer can become defensive when questioned or receiving criticism and determine people don’t like or trust him. On the other hand, a partner who is too passive also creates trouble for the situation. Pair programming relies on the fact that it produces a lower number of defects than individual coding, about 15% fewer. [19]. If one programmer is just passively agreeing with anything the other says, the detection of defects is drastically reduced. The observer’s effect is drastically reduced if he is too unsure of himself and only catches syntax or spelling errors. Another requirement of the pair is that they need to have a good attitude going in. This also seems obvious, but at least one of the two needs to believe that pair programming is the way to solve the problem or it will not work. The pair will decide to split and then compare code, which isn’t nearly as effective and takes more time to explain what each person has done. Pair programming needs to be an interactive process that takes place in front of a single computer. The two programmers also owe it to each other to stay on task while working on an assignment. The driver should not use the computer for other things such as checking email or surfing the web. The observer should also pay attention the whole time the driver is coding. The observer shouldn’t leave for coffee or any other reasons. It should be as if he were actually coding it himself.

Pair programming requirements

In order for pair programming to be productive there are many key items that need to be followed. The first and most important thing to remember is that pair programming is based on communication. This involves many different aspects: the physical location, the primary languages spoken, the attitudes of the programmers, the skill levels of the programmers, etc. First, in order for pair programming to be effective in a given location, a number of things are desired. Figure 1 is a diagram of an ideal situation for programmers on a team to pair program in.

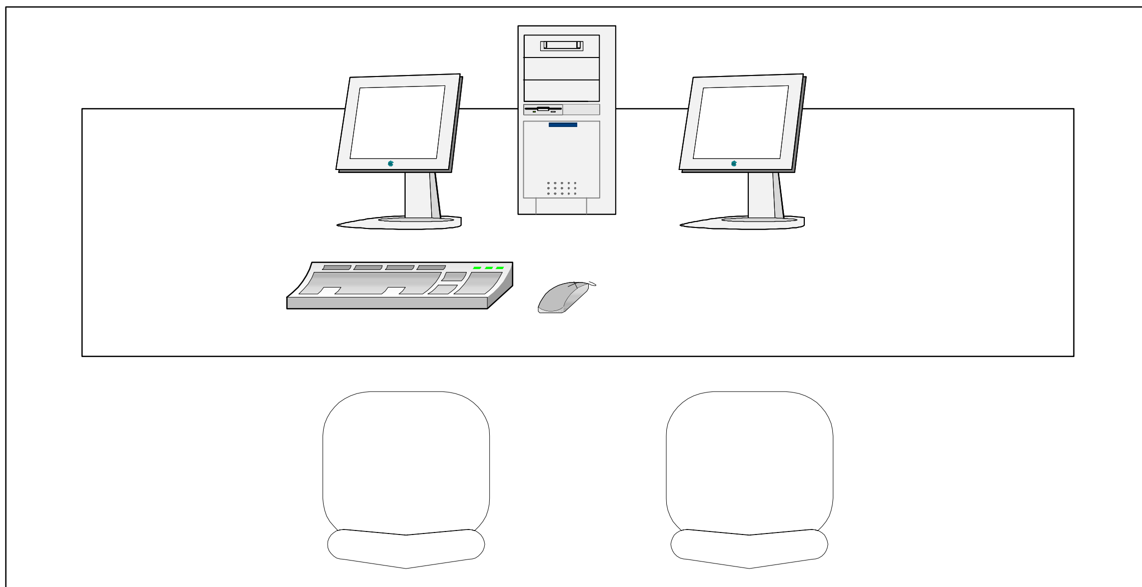

Figure 1: Ideal Pair Programming Desk

First, notice there is plenty of free space on the desk. This allows for both programmers to have scratch notepads, or other items they would like to have close by while programming. Also notice the desk is not in a corner. This arrangement will not work according to [16]. It is much to touch to get two people into a corner and have them be comfortable. Each programmer has his own chair. This is an important requirement, as you don't want one person hunched over the other person's shoulder, sitting on the floor, or constantly pacing back and forth behind the person coding. In an ideal situation, each programmer has his own monitor. Although there are two monitors, there is only one computer. Both monitors are hooked up to the same computer. This allows for each

programmer to see what is happening without feeling crowded. It is easy for the driver to start feeling cramped if the observer is straining to see right next to him. This causes irritation, which reduces production and can cause disaster. Next there are a wireless keyboard and mouse. You want the keyboard and mouse to move on a switch, not the people. Most switches will occur when the observer notices the driver is stuck and has a great idea that just popped into his head. Time shouldn't be wasted having to switch places and making sure everyone gets comfortable again. It is important in pair programming not to disrupt the thinking process.

As mentioned above, a key to pair programming is communication. If the two people trying to pair cannot communicate properly, pair programming will not serve its purpose. There are obvious things to avoid such as people who speak different languages. This obviously will not work in a pair-programming situation. However, even if the two programmers speak the same native language, this doesn't guarantee success. The two programmers will need to take time to establish a common vocabulary. People all come from different backgrounds and use different words to reference different things. Another thing to keep in mind are experience levels. Pairs with different levels of experience create the most discussion and provide the best means of learning from each other [17]. The two people need not be intimidated by each other's experience level in programming. Rather, the less experienced programmer should look at this as a great opportunity to increase his programming ability. Just like in sports, the best way for a person to get better is to hang out and interact with better people. For the more experienced programmer, it causes him to continually question the basics his entire knowledge foundation is built on. This may be frustrating at first, but soon the more experienced programmer will realize he can make errors and will begin to see the benefits of having an observer. Ron Jeffries points out in [18] that he himself learned to appreciate the younger programmer constantly questioning and pointing out possible mistakes and suggestions. Ron said that he had been prepared to give the kid a lesson, and ended up learning just as much as he taught.

Another important key to successful pair programming is to take breaks often [16]. Pair programming is a constant thought process. Each individual knows the other is depending on him to continue working and therefore the productivity increases. However, this increased productivity is physically and mentally draining. Pairs should take breaks often, ranging from full coffee breaks, to simply discussing anything not related to the software at hand. During these breaks is a good time to really get to know your pair. The better you know a person, the easier it is to take suggestions and criticism. It can also help ease the tension created when two people start pairing. If it just really does not seem to be working, it should be apparent to both partners, switch partners and don't let the bad experience spoil pair programming. Chances are most experiences will be positive and you will come out feeling more confident in the product produced [17]. Since XP stresses common ownership of all code with code standards, chances are all people will use the same development environment. It is a good idea for each partner to become familiar with the particular tool before starting the pair programming session to minimize frustration and maximize output. There is nothing worse than trying to work with a tool you don't know, but the person right next to you does. Also according to the principles of XP found in [13] design discussions should be limited. The goal of pair programming is to find the simplest solution to the task at hand. There shouldn't be lengthy discussions going back to the design. If there a major design flaw has been discovered, that should be dealt with separately.

How often should the pairs switch positions while pair programming is a difficult question to answer. In [16] it says to switch pairs often. However, it also says not to switch in the middle of a task. Ideally the tasks will be small enough that it is possible to both switch often and at the end of tasks. However, in reality this may not be the case the programmers should use their own judgment. The main complaint would be that the driver hasn't accomplished anything if he is always starting the tasks, but never gets to be the one to finish and check the finished product in. One of the most important things in pair programming is the sense of common responsibility that should be present. When conversing with the other programmer, words such as we and us should replace the you and I. Blame should not be placed on either of the persons individually, rather both

should be complimented on the success of the project. Even though all production code should be pair programmed, there are certain situations in which it is appropriate to do work alone. Reference [16] names two unique situations: When exploring something new or when there are multiple ideas about what may be causing a bug. In both of these situations it seems that it would be more productive to split and solo code for a while. However, when each is satisfied the produced code should be flushed and the code going into production should be pair programmed. It may seem like a waste to re-write the code just produced, but there is not reason that code should be exempt from the normal pair programming process. The following table shows different situations in which people might want to work alone, and whether it is appropriate or not.

Reason I want to work alone:	**Yes**	**No**
I want to be the hero		X
I'm afraid of exposing my deficiencies		X
What I'm doing is pure speculation	X	
I'm having a bad day and in a bad mood		X
I miss the meditative time alone I used to have		X

Table 3: Is it okay to program solo?

As the table shows, in most situations pair programming should be used according to [16]. If a person has the attitude of wanting to be the hero, he won't do the team any good. The code he cranks out sitting alone will most likely be tough to read which won't do anyone any good later when it needs to be changed. It also hasn't gone through the constant review all other code has, and is more likely to contain errors. If a person is afraid that his deficiencies will be exposed, they should think about the alternative. If the errors aren't found now, they will come out later in the development cycle. It's much less embarrassing for a partner to catch an error and correct it than for the error to come out later in QA. As mentioned in the previous paragraph, if someone wants to try a solution that may or may not work just for kicks; that should be done alone. It can be tough to work with a partner if the day just doesn't seem to be going right. Something bad happens on the way to work and you just want to get there and lock yourself in a corner and code your frustration away. This may make a person feel better, however the code produced in this situation will likely not be very good quality. It is really tough to think

clearly enough to code properly when in a bad mood. Rather, take a few minutes to settle down, maybe talk to you pair about the morning, then get started programming with a buddy. It will help get the spirits up too. Another complaint may be that people miss the time they used to spend alone. However, along with XP comes the 40-hour workweek, which is most likely going to be a decrease in the amount of time worked per week. The employee should use this extra time as the time he needs to himself. It should also be done outside the workplace in order to be truly relaxing.

Advantages of pair programming

Since pair programming is such a new method of software development, there is not much hard evidence of the costs and benefits of its implementation in software projects. However, a few studies have been conducted on the effects of pair programming. In [19], Cockburn and Williams show the main arguments for pair programming. The first advantage to pair programming is that it saves money. The overhead is approximately 15%, but the end product consistently has shown to have 15% fewer defects [16]. At first it seems weird that the overhead wasn't 100%. Before there was only one person working on the problem, now there are two. However, it is more than just two people working on the same problem. They are now working together which helps the problem-solving process [18]. Also, the fact that there are two working creates the pressure to keep working and do less doodling around. This helps to increase the efficiency of pair programming. The fact that you are squeezing more work out of employees is good enough, but results also show that the programmers enjoy the work more when working on it with a partner [17]. In [19] a student is quoted as saying, "It's nice to celebrate with somebody when something works." This is a feeling shared by most people. It's almost frustrating to accomplish a huge software project, especially in academia, and not be allowed to share the solution with anyone. However, the satisfaction did not come without an adjustment period, but in the end, most programmers were happy with the new method. Figure 2 copied directly from [15] shows the satisfaction received by employees pair programming compared to those solo programming.

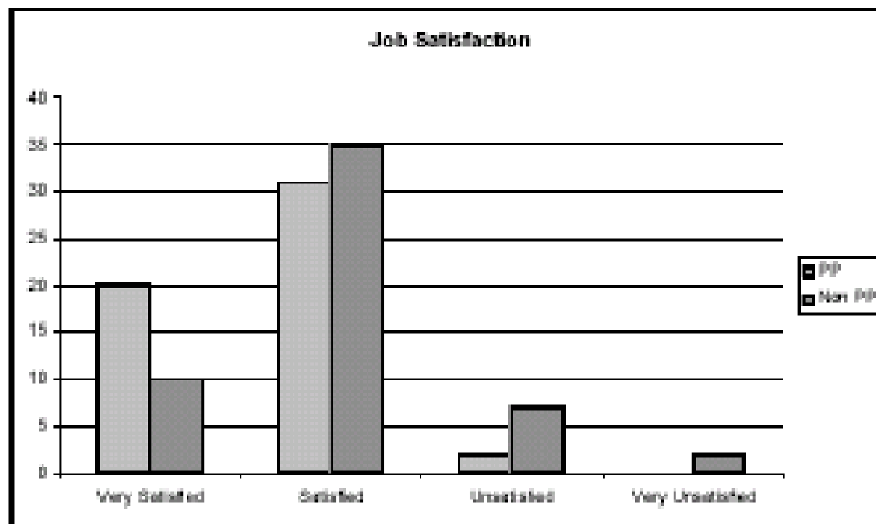

Figure 2: Job Satisfaction

The chart shows that twice as many pair programmers were very satisfied with their job compared to those not using pair programming. There is also a noticeable gap in the lower end of the chart where the employees didn't like their there job. There are very few pair programmers who were unsatisfied with the experience.

Overall quality improves when pair programming is used. The design decisions are made between two programmers and thus twice the thought goes into each. Also, because more time is spent designing the solution, the resulting code is of higher quality and is usually fewer lines than the solo-programmed code. The code quality is also higher when produced through pair programming. Because the code is constantly being reviewed, mistakes are found sooner. Also, with two people there is more pressure to stick to the existing code standards established by XP. This makes the resulting code easier to read, reducing the cost of modification in the future. Readability is also one of the key advantages of good code. Pair programming also reduces the amount of actual code, as two people usually find a faster more efficient way to program. In the study performed by Cockburn and Williams [19], they found that pairs generally wrote between ten and twenty percent less code than those programming the same piece of software by individually.

The pairs will learn to solve problems better and faster than individuals would. With two people brainstorming for ideas, the process will move faster and produce better thought out results. Pairs will also learn faster, and can be brought up to speed on a project much faster than individually. The experienced programmer is forced to introduce the new programmer and the new person is thrown right into the mix without worry of him doing something wrong. The experienced person will be right there questioning and suggesting ideas.

Pair programming also improves team building and communication while facilitating management. The pairs are forced to communicate constantly. Communication is an easy thing to lose when all programmers are locked up in separate cubicles. Without communication, any task immediately becomes much more difficult. Communication between members of the team improves workflow and overall worker moral. People are happier when they know the people they are working with and can trust them to make educated decisions. When pairs are rotated, the information cannot help but spread to everyone. If someone discovers a really cool trick or idea, he is going to want to spread it to whomever he partners with. If the tasks are short like they are supposed to be, the information will spread very rapidly. The management side of programming also becomes easier when using pairs. First, new programmers brought on to the project are thrown right in and can get up to speed much faster than if they were assigned a task and sent off into their own cubicle. Also, due to the fact that the pairs are constantly rotating and picking different tasks, there is less chance of losing a programmer key to the project. Everyone knows more about everything if pair programming is being implemented correctly. This makes for a much stronger base behind the software produced, allowing for flexibility in the work force.

Disadvantages of pair programming

There are however, arguments against pair programming. First, history teaches us that programming is a solitary activity. From academia we are taught to keep our code secret and not to ever look at anyone else’s code. The idea of common code and sharing of ideas

can be a difficult problem to overcome when choosing to implement pair programming. However it can be overcome, as one student wrote,

> "The adjustment period from solo programming to collaborative programming was like eating a hot pepper. The first time you try it, you might not like it because you are not used to it. However, the more you eat it, the more you like it" [19].

It is easy to take the naïve standpoint that two people are now doing the job of one person, which seems like quite a waste. We can see how managers would have a hard time dealing with the fact that now they are paying twice as much for the same position. However, research has shown that programming in pairs is actually more efficient than programming alone [19]. Lastly, many programmers feel their code is private and are reluctant to share it with others. This is a problem that can be overcome with time, and through a gradual introduction to pair programming. Lastly, Hugh Smith has proposed the argument in [20] that eventually pairs would get used to working with each other. Pair programming relies on the idea that the programmers have different thought processes to catch bugs. If the programmers start thinking the same, the number of design flaws caught by pair programming will decrease.

Summary

The following table shows a breakdown of all claimed advantages and disadvantages of using pair programming. These criteria will be used in determining whether or not pair programming is worthwhile in the Linux kernel development domain.

Advantages	Disadvantages
Saves the company money	Initial hurdle of sharing code
15-60 percent fewer defects	Managers don't want to take the risk
More efficient workers	Pairs can get begin thinking alike
Programmers enjoy working more with pairs	
Better design decisions made	
Less code	
More efficient code	
Easier to read code produced	
New programmers brought up to speed faster	
Improves team building	
Eases management's responsibility	
Code is automatically reviewed once	

Table 4: Advantages and Disadvantages of Pair Programming

Application

In this section I will take all the disadvantages of Linux and try and apply the advantages of pair programming to solve the problems. The following chart matches the problems and solutions. Following the table is an explanation on how the specific benefit(s) of pair programming solve the problem with Linux kernel development.

Problem with Linux	Benefit of Pair Programming
No company in charge	Improves team building
Bugs	1. Better design decisions made 2. 15-60% fewer defects
Patches may not be reviewed	Code is automatically reviewed once
Difficult to understand code	Easier to read code produced
Programmers physically spread out	Distributed pair programming
Extremely large amount of code	1. Less code 2. More efficient code
Hard to incorporate new developers	New programmers brought up to speed faster
Number of developers working on kernel decreasing	Programmers enjoy working more with pairs

Table 5: Benefits of Pair Programming Applied

As stated earlier, the most unique problem associated with Linux kernel development is the fact that there is no company in charge. There are no project meetings to ensure

people or on the same page, or project barbeques to help developers to get to know each other. The developers are simply thrown in to development and most likely have never met any of the people they are developing with. Pair programming has shown that it will increase team building [19]. If the programmers feel like they know the people they are working with, the uncomfortable edge goes away and the developers gain a sense of trust. This produces better code and happy developers.

As with most software, Linux contains its share of bugs. Bugs are unavoidable and occur in all software. There is no way to produce large-scale software that does not contain bugs. However, pair programming helps reduce the number of bugs in developed software. In the study mentioned in [18], the code developed by a pair of programmers passed 10-15% more test cases than the code developed individually. Producing fewer bugs is a definite advantage for the Linux kernel development domain.

Solo programmers do most Linux kernel programming. The output they produce may or may not be reviewed before it is incorporated into the kernel. As mentioned previously, Linux reviews much of the code, however he can only do the most important sections. The rest may or may not be reviewed by anyone. Pair programming code is all reviewed by at least one person. Even if more than one person reviews the resulting code, an extra review on account of pair programming does not hurt anything. Rather, pair programming ensures that all code has been reviewed.

Much of the Linux source code is extremely difficult to decipher. It is hard enough to understand, let alone try and make modifications or add features. The programmer who submitted the code may have understood what he was doing and thought that his method was obvious. With pair programming the code must be understood by at least two people. This ensures that the code is easier to read and understand.

The problem of programmers being in different physical locations is a tough one. Pair programming suggests the programmers be in the same room and have an easy method of communication. It doesn’t seem that pair programming can be applied to developers all

over the world. However, there has been some initial research on distributed pair programming. The study described in [21] compares two groups of programmers each with developers in different locations. One group used pair programming via PC sharing software and audio support and the other group performed normal programming via splitting up assignments and assigning to individual developers. The distributed pairs performed slightly better according to letter grades, but the enjoyment experienced by the distributed pairs was higher than those not using pair programming. Therefore, initial research proves that distributed pair programming is a benefit over traditional means of distributed programming.

The Linux kernel is by no means a small project. The current kernel contains a few million lines of code. When a project starts getting this big, the less code added, the better. More code can become repetitive and confusing. Inflated source code may also hint and inefficiencies. Pair programming has been proven to produce more efficient code thus reducing the number of lines of code produced.

With the Linux kernel source being as huge and cryptic as it is, it is hard for new developers to get involved. By pairing a new programmer with an existing programmer, the code is introduced and is less scary. The new programmer has someone to discuss the code with and work with in developing. Even if the programmer is the observer for the first couple weeks, he will be brought up to speed, all while provide the other benefits of pair programming to the resulting code.

Lastly, with the number of developers actually working on the Linux kernel decreasing, something needs to be done to keep Linux kernel development pushing forward. Perhaps people don’t enjoy programming the hardcore Linux kernel stuff after a hard day of work doing programming. That definitely doesn’t seem like an appetizing after dinner treat to me. However, programmers who program using pair programming enjoy the experience more. Applying pair programming to the Linux kernel domain may improve the enjoyment people have and people will look forward to writing coding.

Conclusions

The benefits of pair programming can be applied to many of the problematic areas of Linux kernel development domain. It appears that using pair programming as the means of developing the Linux kernel would solve many of the problems. First, the team of developers would improve their relationships with each other. New developers could be initiated into the development team faster, and hopefully new developers would be attracted due to the pair programming. Pair programming has been proven to produce fewer bugs than solo programming, which would definitely be an asset to the Linux kernel. The code composing the Linux kernel would become more understandable as two people must understand it up front rather than only one. Code submitted will have to have been reviewed at least once due to the nature of pair programming. Pair programming will help many of the problems in the Linux kernel domain.

Future Work

An area of interest to study in the future would be to see how the other fundamentals of XP apply to the Linux kernel domain. Some of the other fundamentals such as a common coding standard and collective ownership of code seem to apply rather well. However, some of the others such as refactoring may be difficult due to the nature of the code being produced. Also, XP requires constant customer involvement, which would be tough as there is no specific customer Linux is being developed for. Rather it is being developed for purposes from office workstations to backend servers.

Another area of interest would be to take the advantages of pair programming and apply them to other areas of code. Perhaps studying to see if the affects of pair programming are dependent on the type of programming language. The study regarding distributed pairs [21] referenced above mentions it uses and object-oriented language. It would be interesting to see if pair programming is more or less beneficial when procedural languages are used.

References

[1] R. Turner. *Operating Systems: Design and Implementation*. Macmillan Publishing Company, New York, NY. 1986.

[2] A. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-hall Inc., Englewood Cliffs, NJ. 1987.

[3] A. Silberschatz and P. Galvin. *Operating System Concepts: Fourth Edition*. Addison-Wesley, Reading, MA. 1994.

[4] M. Bar. *Linux Internals.* McGraw-Hill, New York, NY. 2000

[5] A. Tanenbaum. *Modern Operating System: Second Edition*. Prentice-hall, Upper Saddle River, NJ. 2001.

[6] D. Wheeler. *More than a Gigabuck: Estimating GNU/Linux's Size*. June 30, 2001. Available at: http://www.dwheeler.com/sloc/redhat71-v1/redhat71sloc.html. Viewed December 8, 2002.

[7] M. Welsh, M. Dalheimer, and L. Kaufman. *Running Linux*. O'Reilly & Assicuates, Inc., Sebastopol, CA. 1999.

[8] Linux Kernel version 2.4.20 Changelog. Available at http://www.kernel.org/pub/linux/kernel/v2.4/ChangeLog-2.4.20. Viewed December 8, 2002.

[9] Fork.c from the Linux kernel version 2.4.19. Found online at http://lxr.linux.no/source/kernel/fork.c. Viewed December 8, 2002.

[10] Edsger W. Dijkstra. *EWD1308-0*. From the Manuscripts of Edsger W. Dijkstra, University of Texas, Austin, TX. 2001

[11] Larry L. Constantine. *Constantine on Peopleware*. Yourdon Press, Englewood Cliffs, NJ. 1995.

[12] K. Beck. *Extreme Programming Explained: Embrace Change*, Addison-Wesley, Boston, MA 2000.

[13] John Brewer. *Extreme Programming FAQ*. http://www.jera.com/techinfo/xpfaq.html. Jera Design. Viewed November 2002.

[14] D. West, *Metaphor, Architecture, and XP*. Third International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP2002).

[15] G. Succi, W. Pedrycz, M. Marchesi, and L. Williams. *Preliminary Analysis of the Effects of Pair Programming on Job Satisfaction*, Third International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP2002).

[16] K. Auer, R. Miller. *Extreme Programming Applied: Playing to Win*. Addison-Wesley, Boston, MA. 2002.

[17] L. Williams and R. Kessler. *All I Really Need to Know about Pair Programming I Learned In Kindergarten*, Communications of the ACM, 2000.

[18] L. Williams, R. Kessler, W. Cunningham, and R. Jeffries. *Strengthening the Case for Pair-Programming*. IEEE Software, vol. 17 (2000), No. 4, 19-25.

[19] A. Cockburn, and L. Williams. *The Costs and Benefits of Pair Programming*, In Proceedings of the First International Conference on Extreme Programming and Flexible Processes in Software Engineering. Italy, 2000.

[20] Personal Interview with Prof. Hugh Smith. California Polytechnic State University. December 2, 2002.

[21] P. Baheti, L. Williams, E. Gehringer, and D. Stotts. *Exploring Pair Programming in Distributed Object-Oriented Team Projects*.