

Four. Politics of Source Code

Now we are going to deal with the alternating history of freedom of source code. Once it was innocently free, in the then world of hackers, that we called ‘primitive FLOSS’ in the earlier chapters. Then this freedom was taken away by the rules of market. The world of free source code got confined into proprietary restrictions. But this was only a period of waiting. Once again this lost freedom was regained, the closed down source code got reopened. And this time it was done in such a way that no one can take it away any more. This was accomplished by GPL and FLOSS. In this chapter we traverse this history, from primitive freedom to confinement, and then the start of the movement that led to GPL and FLOSS, and thus the regaining of this freedom. But, as it happens with any recent history, marked more with a plenitude of data than any dearth of it: the problem is to decide, where to start from. Shall we start from birth of computer, or from the emergence of OS or Operating System? Or, from Unix, with its continuity into the GNU-Linux phase? Or, the start of FSF and then the FLOSS movement, or even later, from the birth of the Linux kernel? As a solution, let us be plural, let us start from all of them, a bit from each in turn. This chapter tries to give a feel of the hacking environment and primitive freedom of knowledge in the world of computing.

1. Prehistory of Computers

The history of computer science is a history of developing abstractions, actually layers of abstraction, progressively more dense and elaborate, to reach the abstract virtual space within a modern day working computer. The epitome of this is the metaphor of a computer working within a computer, like when we click the icon ‘computer’ on our desktop. Obviously it is a very elaborate symbolic space containing symbolic links to many of the real components of the real computer. The action of a click on this icon allows us to access this real space in many predefined ways. Actually, this happens in so many ways that we tend to forget about the symbolic nature of this space. In the last chapter we discussed how the kernel works as the principal translator in this elaborate metamorphosis of the symbolic space into the real. But in the final sense, it is nothing but symbolic: a long chain of very well-defined metaphors working through many layers of abstraction. Some of these we mentioned in the last chapter. We can say, in a way, the history of development of computing, if not science as a whole, is a history of development of these abstractions.

And so, we start from an woodcut illustration in the book *Margarita Philosophica* by Gregor Reisch. This illustration actually depicts a two layer abstraction, natural to numerical, and then numerical to symbolic. *Margarita Philosophica*, the encyclopedia of philosophy, was vastly in use in universities of medieval Europe as a textbook sixteenth century onwards. In the right half, Pythagoras, the famous Greek mathematician is doing calculations with stone beads on a counting-board. In the left half is Boethius, the translator of Aristotle, sometimes credited with the introduction of Hindu-Arabic numbers to the Western world. He is doing calculations too, but with symbols. *Arithmetica*, the goddess of arithmetic, is supervising the race. And as the countenances of the contenders suggest, it is not exactly going the Pythagorean way. This is Reisch’s reading of the history of mathematics. The stone beads on the board represent countable real objects, and transfer them to a numerical world, performing the first abstraction layer from the real to the numerical. And the left half works with symbols, transferring mathematics into the logical

world of abstracted knowledge. There are many interesting steps in the gradual progress of abstraction – Stephenson 1999 elaborated some of them, the role of abstraction and metaphor in the realm of OS or operating system. In the last chapter we described a few steps about the progressive unfolding of abstraction and metaphor in the virtual user space created in computer OS.



Anyway, after those abstractions in the woodcut from Reisch, the next jump of abstraction in computing was the digital representation of numbers operating within computer, that we discussed in the last chapter. The jump from numerical to digital in this age of computers, in reality, consists of millions of discoveries over hundreds of years. And let us make a jump-cut here through that prehistory to the mechanical number-crunching machines emerging during the birth of the modern technological age. In the march of these inventions of calculating machines, one of the very starting name is that of John Napier with his logarithm and his calculating gadget better known as *Napier's Bones*. Then came Edmund Gunter, William Oughtred, and Amédée Mannheim with their different versions of *Slide Rules*. William Schickard's *Calculating Clock*, Blaise Pascal's *Pascaline*, Gottfried Wilhelm Leibnitz's *Stepped Reckoner*, and Charles Xavier Thomas de Colmar's *Arithmometer* were some important ones in the long line of calculating machines.

In this long march, the numerical methods and machines were all a kind of concomitant with the fostering of calculations and yet more calculations, as it happened with the inception of the technological era. It came with a revolutionary increase in the total number of roads and constructions, maps and navigational charts, business and commerce. The history of these number crunchers culminated in *Difference Engine* and *Analytical Engine* by Charles Babbage and Ada Byron Lovelace. The problem of Babbage and Lovelace was that the make of their machines was too advanced for the technology of its time. And the leap of abstraction involved in these engines is adequately elaborated in the works of Ada

Lovelace. No one can help but agree that Ada understood and discussed the working principles of a stored-program computer long before they were built. The notes of Ada Lovelace does give a kind of an eerie feeling of reading an inhumanly correct predictive science fiction. She anticipated quite a lot of things in the realm of modern-day high-level computer languages. Though it may not be exactly relevant for the purpose of this book, we can quote here a paragraph from Lovelace 1842. This is “Sketch of the Analytical Engine” by L. F. Menabrea, translated and augmented with extensive commentary by Ada Byron Lovelace. This paragraph belongs to that commentary: a paragraph that prefigures programming on modern digital computers at least hundred years before they came.

The bounds of arithmetic were, however, outstepped the moment the idea of applying cards had occurred; and the Analytical Engine does not occupy common ground with mere “calculating machines.” It holds a position wholly its own; and the considerations it suggests are most interesting in their nature. In enabling mechanism to combine together general symbols, in successions of unlimited variety and extent, a uniting link is established between the operations of matter and the abstract mental processes of the most abstract branch of mathematical science. A new, a vast and a powerful language is developed for the future use of analysis, in which to wield its truths so that these may become of more speedy and accurate practical application for the purposes of mankind than the means hitherto in our possession have rendered possible. Thus not only the mental and the material, but the theoretical and the practical in the mathematical world, are brought into intimate connexion with each other. We are not aware of its being on record that anything partaking of the nature of what is so well designated the Analytical Engine has been hitherto proposed, or even thought of, as a practical possibility, any more than the idea of a thinking or a reasoning machine.

These notes show one of the high-points in the history of abstraction in computing, as we said, long before the real history commences. The way she dealt with Bernoulli numbers in Lovelace 1842 is considered by many as the first program in the history of programmable computers, with the concept of ‘loop’ in-built into it. Anyway, if we want to go into the process of blooming and growth of these abstractions, we will have to delve into works of many more talents in the history of computers, like George Boole, Von Neumann, Alan Turing and so on. But that will be definitely outside the scope of this book. Anyway, this prehistory of computers comprised of mechanical machines and methods. The electrical age in calculating machines dawned with the introduction of vacuum tubes during the First Generation, as it is called. One thing must be kept in mind, this categorization into generations is more of a categorization of convenience, made for discussions, without any internal and intrinsic meaning.

2. First Generation 1945-55

Electronic computer was a direct derivative of World War II. Different kinds of war necessities, from bombardment schedules to intelligence and counter-intelligence, were

bidding for a lot of computing power unattainable with the then available methods. The birth of the First Generation digital computer came as a result of the works of extremely talented people like Howard Eiken and Grace Hopper from Harvard with Mark I, John Presper Eckert and John Mauchly from Pennsylvania with ENIAC, John Von Neumann and Claude Shannon with their scientific and mathematical theories, Konrad Zuse with his *Plankalkül*, and so many other people from so many walks of life.

Not always the history of computing mentions Zuse with the importance he deserves. Konrad Zuse from Germany came with his machine and *Plankalkül*, a method of programming computers. This *Plankalkül* or “Plan Calculus” was actually a very different sort of programming language than the high-level languages like C, Fortran or Java that we know today. It was different in the sense that, unlike these languages, “Plan Calculus” did not presuppose a Von Neumann Architecture that we discussed in the last chapter. The glory of Zuse’s works is that he did all these, helplessly alone. The lonely Engineering student built the machine of his dreams in his own living room, by cutting simple metal strips with a zig-saw, strips that will be used in place of electric relays of the later machines. He was completely unaware of the other contemporary developments in the field. The works of this brilliant German student were all lost in bombing during World War II, and thus, Zuse lost the opportunity of influencing the developments that followed.

In this first generation, machines were made of vacuum tubes and plug-boards. In the start, electronic relays were yet to come. In the early years of the first generation, slow-working mechanical relays were used in place of electronic relays. Vacuum tubes brought in quite a revolutionary change. In this period, vacuum tubes started replacing the relays of the electrical circuitry of the earlier machines like Mark I or ENIAC. A relay is an electrical instrument that opens or closes a particular electrical circuit. And this opening or closing is controlled by another circuit. And so, a relay can relay action from one circuit to another – without which the chain of computer’s logic circuits cannot operate. Concepts like high level programming language or OS, at that time, were more of science fiction. It was even before the birth of assembly language that we mentioned in the last chapter. So the instructions into these machines were all fed through the lowest level: machine language, directly to the architecture of the machine. In this generation, a lot of works were done with plug-boards, things that did their work not in an electronic but an electrical way.

Running a program on a machine of these days meant composing and recomposing of the connections on a plug-board, and then inserting this plug-board into a machine that would fill an auditorium. This plug-board carried the program, physically, in terms of the cables and plugs. And then these commands imbibed in this plug-board would get resolved and executed by thousands of vacuum tubes. The board was filled with sockets, which were plugged with connecting cables, thus controlling the electrical connections, and thus, the lowest level instructions. This work was done by the plug-boards, by directly controlling the hardware of the circuits and wires. The instructions of the program that are supposed to run on the machine were held by these plug-boards. Then the vacuum tubes within the machine just operated on these instructions. The computer work was almost solely concerned with calculation, that is, crunching of a very large number of very large numbers. What a computer does today is just the same thing, with the complexity boosted up to billions of times.

The plug-boards got replaced by punched cards in the early fifties. Here the instructions were given by making some holes on a card, and feeding this card to the machine. These holes carried all the raw data and instruction data necessary for the program to run. But, other than this change in method of IO, input/output, of data, the working principles of the machines remained almost the same till the second generation computers started landing on this planet. The first generation of computers gave way to the second generation with the invention and introduction of transistors.

3. Second Generation 1955-65, FMS, Bus

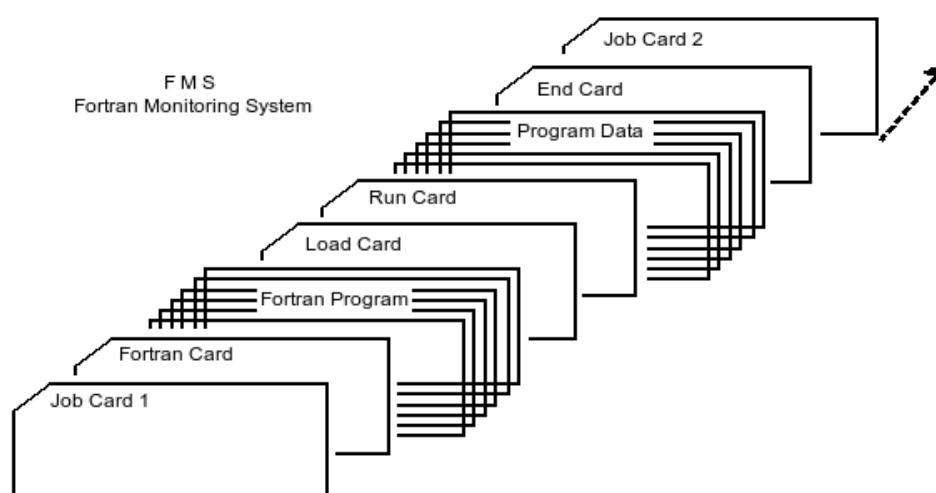
Three physicists Walter Houser Brattain, John Bardeen, and William Bradford Shockley came in 1948 with their discovery of transistors. This actually revolutionized the history of digital machines during this second generation of computers. Bipolar-Junction-Transistors, usually called BJT, replaced the earlier vacuum tubes. Transistors now controlled the current in the logic circuits. Transistors dramatically reduced the cost, size and operational cost of computers. The juggernaut of computer science started moving. Brattain, Bardeen and Shockley got the Nobel Prize in 1956 for this discovery. And within the turn of the fifties decade, vacuum tubes became obsolete. Starting from the first one named TX-0, Transistorized-Experimental-Computer-0, it became an age of transistorized digital computers.

Transistors improved not just the CPU, but the peripherals too. New kind of storage devices started to come up with a much higher ability of storing data. This age of transistorized machines first saw the emergence of Teletype, remote terminals at a distance connected to a mainframe by cables. The PDP series of machines was introduced by DEC, Digital Equipment Corporation, in 1957, as a commercial make of machine, much like the TX-0. On these PDP series machines, Dennis Richie and Ken Thompson will build Unix in the same Bell Laboratories, where Brattain, Bardeen and Shockley discovered transistors. This history of Unix we will need to explore later, in fuller details.

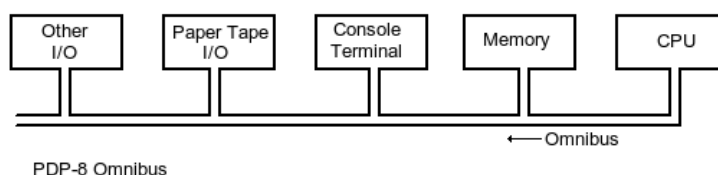
By this time, Fortran, Formula-Translation, was ruling the world of computing. It was the first high-level computer language, developed by John Backus in 1954-58. In Fortran we first witness the ancestors of the modern day armory of high-level languages: the concepts like *variables*, *expressions*, *statements*, *iterative* and *conditional statements*, separate *subroutines*, and formatting of IO. FMS, Fortran-Monitoring-System, the progenitor of a modern day OS, was born in this age of second generation machines. This was the time of punched-card programs. To run a punched-card program on a mainframe computer of this time involved quite a few repetitive identical steps. FMS, the ancestor of OS, came by, in the form of a program that looks after the repetitive steps involved in running programs, in the way of the element of 'control' mentioned in the discussion of Von Neumann Architecture.

These steps were like: loading the Fortran compiler into the machine (involving quite a few cards), then feeding the Fortran program, created by the programmer into the machine (again involving quite a few cards more). And if the computer does not run the full circle correctly in one go, a thing that was quite common, the programmer will have to feed the compiler cards once again. And then the program cards too once again, and like this it went

on, till the compiler fully read and translated the high-level Fortran program into machine executable code. And now, into this prepared machine, the programmer would load the machine language cards, and cards for all the subroutines that will be needed for the program to run. At last the program will run and the results will be printed in another set of punched cards, if only there was no error. In which case the same steps will be repeated once again. The motive of FMS was to look after all these repetitive steps. In order to reduce the immense operating time lost in repetitive tasks, that is, to automate the operator's job, this special program FMS emerged, in the sixties decade. The programmer's work was now to provide certain control cards and the program cards as and when needed to run the special program in the machine. These cards were read and carried out by the this special program, FMS.



In PDP-8, one of the later models of the series, DEC introduced a major innovation, a single bus called 'omnibus' that connected all the components: CPU, memory, terminal, and IO, all in one single routed connection. This was a major deviation from the earlier memory-centric models. This gradually became the system for all computers to follow. Compared to the first generation machines, these second generation transistorized machines had quite a few advantages like, less cost to produce, a lot more speed, a lot lower volume, and a higher reliability due to a much lower operating temperature. These new machines held and operated on tens of thousands of binary logic circuits. This was a scale higher by leaps and bounds than that first generation machines.



Section 4. Third Generation 1965-80, Portability, Multitasking

Robert Noyce and Jack Kilby invented IC, Integrated-Circuit, also called 'microcircuit', 'microchip', 'silicon chip' or 'chip', in 1958. The IC technology allowed the producers to go into mass-production, with an ensured reliability of the ready-made circuits. Let us mention here, IC is a very small miniaturized electronic circuit made of semiconductor

devices or electronic components that use semiconductor materials like silicon, germanium, or gallium arsenide. These semiconductor materials are special in the sense that they have an electrical conductivity more than insulators and less than good conductors. And some impurities can be added to these semiconductor materials that can act as a controlling device on the nature of conductivity and the kind and amount of charge it contains. For example, a small amount of impurity like phosphorus or boron in silicon greatly increases the positive charge content, and are called p-type semiconductor, and on the other hand the other type of semiconductor device is called n-type which contain a higher number of electrons, thus vesting them a higher negative charge content. The interpretation of these characteristics into logic circuits of binary nature is just one step away.

IC-s are used in every kind of electronic equipment in the modern world. It has revolutionized electronic industry as a whole. Combining together a large number of small transistors into a tiny chip was quite big a boost in comparison to the earlier process of manual assembly of circuits using separate individual electronic components. This led to new ways of doing things where new designs now adopted standard IC-s without going into the intricate details of building designs with individual transistors. Due to special technologies that prints the whole chip as a single unit, the cost of producing IC is low compared to any other alternative technology. And because this puts all the individual components so close together, the circuit can switch between components as and when necessary for doing the work very quickly and consuming very little power, in advantage over all the other alternatives.

IC made it possible to build computers much smaller in size, much faster in performance and much cheaper than its contemporary alternatives. And, it triggered off another spontaneous development in the world of computer building. Computers started getting organized into two main divisions. One of these was heavy machines, doing big jobs of mathematical nature like number-crunching and so on, used in the realm of science and technology. The focus of this kind of machines was on the concept of ‘word’ or units of memory. And the other variety was small commercial ones. This second type of machines was focused on IO of data – reading data and writing it back in terms of ‘stream of characters’. For IBM, the leading computer company of that time, the example of this division was the combination of two highly successful machines, 7094 and 1401. 7094 was a high-speed number cruncher using parallel binary arithmetic, and 1401 was hardly more than an IO processor. It was used in writing raw data on tape to be read by the big ones, or, for reading data from tape, where it was kept by them, or, taking prints from them. 1401 kind of machine was used mainly by the insurance or banking companies. Now, with the advent of IC, the cost of building machines came down. All this time, civilization’s demand for crunching numbers or data was increasing at great speed. And so, IBM started making a new kind of machine, System/360 by name. 360 could perform both the number crunching and character flow kind of job, quite efficiently for their times.

The most important thing about 360 was that, it was the inception of the concept of uniformity of machine structures, enabling an user to run one program developed in one machine on another one with the same kind of Hardware structure. This was true for System/360, and the later additions to this group like 370, 4303, 3080, or 3090 and so on,

some of which were being used even into very recent times. This feature, as we will see later, will have a very deep impact on the history of programming, OS, and computing as a whole, in the form of the concept of ‘portability’. As we know from the last chapter, *portability* of software means its ability to be reused on another machine, from the basic level of OS to the level of user applications. In case of 360, this portability created a problem too. System/360 machines varied wildly in terms of size, ability and scope. So, there were quite a large number of peripheral devices that were used together with one or other machine in this series. Hence, portability will imply that, all the drivers of all the devices would have to be there in every instance of OS in every machine. And this was a tall order for the technology of that time, even for a company like IBM. For System/360 range, IBM put ‘microcode’ to commercial use, in order to achieve this portability among machines. And for every driver the bulk of microcode would go on inflating, progressively rendering it quite unmanageable. Microcode is just a very low level code, even lower than machine code – it specifies what a CPU does when it executes a machine-code-instruction. Accumulation of all these microcode for the System/360 range led to proliferation of bugs, and finally gave birth to a very non-optimal and buggy OS, to say the least.

But, this very System/360 range gave the concept of ‘multiprogramming’ too. It was one of the most novel ideas that were generated by third generation machines. For the very big, costly and resource-intensive machines of this generation, there was a concept called ‘IO Intermission’. This signified the periods of time when the CPU was sitting idle while some IO job is getting done, say while the tape was being changed, or while a massive print job was going on, or something like that. Obviously there were very few IO Intermissions when some really voluminous scientific calculation was going on, if any at all. But these intermissions started getting quite numerous in case of big chunks of commercial data processing. This kind of data being very IO intensive, the CPU was just sitting idle for sufficiently long periods of time. And as we have already mentioned, for computers of that time this meant big time wastage of money. For really big jobs of commercial data processing, the activity time of the CPU would plunge down to as low as only ten percent.

As a solution to this problem of wastage of precious CPU time, there emerged the concept of ‘multiprogramming’. To go into that, let us get familiar with the role of *multitasking* performed by OS. These days this concept of multitasking has become so much a way of computing, we are so much used to this being done by OS, and that too so seamlessly, that we are hardly conscious of it. The concept of multitasking is a superset of the concept of *multiplexing*. *Multiplexing* or *muxing*, an import from the realm of telecommunication and computer networks, in short, is a technique of combining communications or IO operations for transporting multiple signals simultaneously over a single channel or line. Now think of an active OS where a number of programs are running simultaneously. It maybe a very simple combination of programs happening every moment on any PC, like say, a multimedia program playing music, a browser surfing the Net, a word-processor working on text, and so on. Or, it maybe some very special programs running on very special kind of machines. But, in every case, all the simultaneously running programs are working on the same machine with the same CPU, with the same resources attached with the machine, like say the same hard-disk, or the same RAM, or things like that. Now, the question is, how they are doing it, without muddling up the whole thing by one program trying to use the same memory while some other one is already using it? Here the concept of

multitasking comes in, a technique that is used by OS on the two separate planes of time and space.

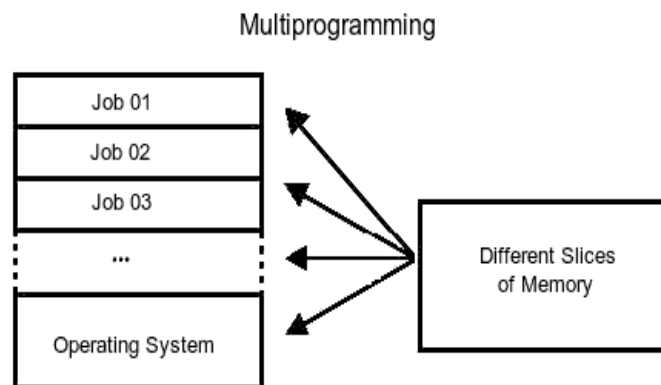
The process of multitasking, we said, is a superset of the concept of multiplexing. It enables a computer system to go on with different channels of multiple tasks, also called as ‘processes’, all these tasks sharing a common pool of resources like the CPU, the RAM and so on. Let us remember it for sure that in the truest sense of the term, there is *no* multitasking of the resources, because one and only one job can occupy a CPU at any single point in time: the CPU can carry out one and only one instruction at a time. This last statement, though sounds quite simple, actually, is not so simple at all. There is much inherent complexity, but, that obviously does not concern the simple things that we want to say here. And, like the CPU, the same thing is true in terms of RAM use too. One single bit of RAM, at any point of time, can never hold more than one piece of binary digit. Obviously one bulk of RAM bytes can be divided into smaller pieces, but one piece of RAM at one moment can hold information from only one process, anything else will violate the laws of physics. So, the only way out is to make the system multitasking for the user, while the CPU or RAM remain in their single-entity being.

A modern OS solves the problem of multitasking by scheduling. This schedule decides, which task will be the one running and thus using the resources at any moment, and when, after how much time the next process waiting in queue will get its turn of employing the CPU and other resources. This is for one CPU systems. In a system with multiple CPU-s, this system of multitasking gets generalized for all the CPU-s. There are more than one scheduling strategies, and an OS can choose which one of these it is going to adopt. These strategies are in general of three kinds, the *multiprogramming* schedule, the *time-sharing* schedule, and the *real-time* schedule. In *multiprogramming* schedule, the running task keeps running until it requires an external event for another task to take its place. This external event maybe reading/writing from/to a storage device, like the IO Intermission that we were talking about. Until the OS’s scheduler forcibly ejects this process from the CPU and replaces it with another, this task goes on running. In *time-sharing* schedule, a running process has to let the CPU and other resources go, either on its own or due to a hardware interrupt. And in *real-time* scheduling, an waiting task is guaranteed to get the CPU and the resources when an external event occurs, particularly in systems designed to control mechanical devices like industrial robots, which require processing of an event-driven job occurring in real time.

So, as we said, *multitasking* is executed on two separate planes, the axes of time and space. When processes are multitasked on the axis of time, all the processes are waiting in queue. The very moment the earlier process relinquishes the CPU and other resources, it is time for the next process to claim and possess it. And then it goes on employing the resource till the time is up, and then releases it for its possession by the next process in the queue. So, time is broken into small slices, and every process is getting one time slice in turn. But every time slice is so small, at least one hundredth of a second for a normal OS, that it is never possible on part of the user to understand the constantly rotating activity cycle of the processes. This is not true for the so-called ‘tickless’ kernels of very recent times, mainly meant for power-saving, but, once again, that is just not relevant for us.

The other kind of multitasking happens on the axis of space. If we consider the whole store

of resource as a plot of land, then this piece of land is getting partitioned into a number of smaller plots. This number is directly dependent on the number of processes running within the system and the size of the plot is directly proportional to the individual requirement of a process. Usually, RAM or swap-file space is allocated to each of the running processes in this way. So, in a real OS, if multitasking on the axis of time is to take place, there must be some kind of multitasking on the axis of space always ready activated. When the time-slice comes, if the new process has to take the charge of the CPU, there must be some place where the process was already there in a dormant state. So, whenever multitasking is operating, the operation happens on both the axes of time and space. In a very simplified way this is what multitasking is, the very inception of which took place during this third generation.



To solve the drainage of precious CPU time through IO Intermissions, System/360 range of machines started *multiprogramming* schedule. Let us assume that the whole memory is cut down into small slices. Now, say every running job is allocated one small slice of memory each. While Job 1 is waiting for the IO Intermission to end, Job 2 is employing the precious CPU time, and so on. So, all these slices of memory, under the dictatorship of OS, are keeping the CPU employed. And when to allow which job to contact the CPU with which slice of memory – that is entirely under the jurisdiction of OS. This ploy of slicing the memory is pushing the CPU towards full-employment. For the technologies of that time, multiprogramming demanded a newer kind of hardware and system, where it is possible to slice memory in this way and keep the slices separate. The arrival of IC fulfilled this demand on technology to a long way. And so, a lot of costly CPU time was resurrected. And some of that came from using multiprogramming in order to reduce the unwanted IO Intermissions for reading/writing from/to punched cards. When one program was reading/writing from/to punched card, the CPU was employed by another program. A moniker for this process was SPOOL: Simultaneous-Peripheral-Operation-On-Line, a word still in use in computing literature.

The first really functional time-sharing was successful in MIT in 1961. It was called CTSS, Compatible-Time-Sharing-System. And, after this, a modified form of this time-sharing was implemented in IBM 7094. But, time sharing in a big and popular way was yet to come. With the innovation of newer kinds of hardware during the third generation, all major computer concerns like MIT, Bell Laboratories, General Electric, and so on, started using time-sharing systems. Gradually a new dream was starting to form, that of a gigantic system, hundreds of thousands of users connected to it, each on his own tele-terminal. This

projected scheme of numerous users dispersed over a vast geographical space, like the city of Boston, connected to a single OS, getting all their jobs done, gave rise to the idea of Multics: MULTiplexed-Information-and-Computing-Service.

But, to a long way, this plan was an anachronism. The computer scientists of that time wanted everyone to get the service of a computer in the context of the then scenario of computing. It was a time when hardly a single individual could afford to have a computer of his own, and so a big machine able to handle thousands of users could obviously generate an economy of scale – this was the idea of the computer scientists. But, the situation, soon, was going to change in such a way, that nobody, no computer scientist of that time could foresee it. If someone told them that, within two decades or so, computing machines will get so cheap that machines million times powerful than the most able one of that time would sell at a price that even a middle-income individual can afford, it would seem to them some outrageously wild sci-fi.

The experience of Multics bred some mixed results. The work on Multics started in 1964 and the last active Multics system was shut down in 2000. With a computing power just above an Intel 486 and serving hundreds of users simultaneously, Multics showed real efficiency in optimized programming, primarily demonstrating the skill of the developers who built the programs under so restraining a resource base. And that too with a difficult language like PL/I, the one they used. Though not any epitome of success in itself, to say the least, Multics left some footprints on the shape of computing to come. Though Bell Laboratories and General Electric deserted the fight midway, MIT pushed the Multics project on, and Multics was finally installed in more than eighty machines in different big companies and universities, a lot of which continued working till the late nineties, that is, till more than three decades after the Multics project started.

The big marks on the computer science of the coming times that Multics left is very well discussed in standard literature. Let us mention a very important one of them here. In Unix or GNU-Linux, any physical device is dealt as a file, and this approach is a legacy of Multics. And the man who was one of the creators of Unix, Ken Thompson, came from Multics. One major source of the failure of Multics was the language PL/I, as we mentioned. The PL/I compiler was developed by IBM in the early sixties. As we discussed in chapter three, this PL/I compiler is that which enabled PL/I programming. Programming in PL/I needed a PL/I compiler that would translate the human-understandable and human-editable PL/I codes into machine-executable PL/I programs. Anyway, the goal of PL/I was to create a single language for both the scientific and the commercial users. And hence, features from three of the then popular programming languages were brought together. These were Fortran (FORmula-TRANslation, a language meant for scientific works), COBOL (COmmon-Business-Oriented-Language, a language for business computation, still in use), and ALGOL (ALGOrithmic-Language, another language for scientific works that intended to rectify some flaws in Fortran). With these features were added capabilities for extensive automatic corrections of syntax errors and the concept of multitasking. As a language PL/I was not very difficult to learn. But, implementing the PL/I compiler became so complex, time-consuming and difficult, that it never became popular. But it had very definite contributions into the coming of C and Unix, we are coming to that later.

5. Later Generations

In the heading of Section 4, we showed the time-span of Third Generation machines as 1965-80. But there are differences in opinion about this, and this is pretty natural. Computers were not built according to time-frame of generations. The frame is actually added later, to categorize something extremely heterogeneous, diverse, polymorphic and multi-directional in nature: the real history of machines. And so different historians may very well choose different landmarks to mark their periods. There are historians of computing who specified the span of third generation as 1964-71. Opinions are there, that specify a fourth generation from 1971 onwards. Or, even a fifth one that is yet to happen, machines that will work in terms of AI, artificial intelligence, till now more or less a category from sci-fi. Though this alternative terminal year for the third generation, 1971, actually represents a real break. It was the year of birth of a fully-functional Unix, though the process of birth started in 1969, as we mentioned before. Anyway, in this book we have mainly followed the calendar given in Tanenbaum 2005.

Tanenbaum 2005, after the third generation, specifies a fourth one from 1980 onwards, where it chooses VLSI, Very-Large-Scale-Integration, as the landmark. While Large-Scale-Integration could fit hundreds of components into the small area of a chip, VLSI, that emerged around 1980, enabled a chip to hold hundreds of thousands of components. This led to a very dramatic reduction in the size and price of computer, simultaneously hiking the reliability, power, and efficiency. Before the coming of the PDP machines, that we are going to discuss in a somewhat detailed way in our next section, computers were so big and expensive that companies or universities using them would have to run full-fledged computer centers for them. With the advent of new kinds of machines, now a single department of a company or university could have them. From 1980 onwards the price came down to a level where even an individual could afford it. Thus began the era of PC, personal computers. It was a birth of a new age that happened with Apple and IBM working as midwives. And this opened up, at that time, new areas of computer use: things like word-processing, spreadsheets, and so on. Many highly interactive user level programs, meant solely for the PC-s, not the big machines any more, started to materialize.

After the introduction of PC in 1981 by IBM, the number of PC-s in use went from two millions in 1981 to more than five millions in 1982, and sixty-five millions in 1991, and this figure goes on rising till date. And all this time computers continue getting cheaper, and more than that, they go on getting smaller, from desktop to laptop to palmtop to embedded systems. After getting introduced in 1981, IBM PC became the best-selling computer in history within almost no time. In terms of PC history, IBM now did something quite revolutionary in terms of the things to come, and this gesture was so very different from the practices of Apple, the other pioneer of PC age. In place of keeping the design a trade-secret, or, at least keeping it guarded by some patent, IBM published the complete design, together with all the circuit diagrams. They wanted other smaller companies to start manufacturing the components, thus increasing the flexibility and popularity of PC. And so, with the whole design in hand and components available in the market, many other companies started producing clones of IBM PC. Thus happened the PC revolution. And a whole industry and market of PC and its components were generated. The initial version of IBM PC came with MS-DOS, Disk-Operating-System supplied by Microsoft. But, this

started a new era. This history is extremely necessary for the purpose of our book, and we are coming back to it in fuller details, shortly. In this context, the essay, Stephenson 1999, that we mentioned in the discussion of metaphors and abstraction, is important.

By the middle of eighties a new thing came up, called RISC, Reduced-Instruction-Set-Computer. RISC is a new concept of CPU design that enables the CPU to ‘do less’ and thus ‘do it better, simpler and faster’, optimizing CPU performance. RISC computers can obey multiple instructions together, often in a different order than given in a program. These RISC machines started replacing the CISC, Complex-Instruction-Set-Computer, ones. Another direction of change was the Superscalar CPU architecture that works through a kind of parallelism, called ‘instruction set parallelism’ within a single processor, thus enabling the CPU to work faster. Another trend in the recent times is using multiple-core CPU-s in place of single-core ones. A multi-core processor is a combination of more than one independent cores, usually CPU-s, into a single package made of a single IC. A dual-core processor contains two cores, and a quad-core contains four. A multi-core processor implements multiprocessing or multiple processes, as if with multiple CPU-s, from within a single physical body. But, let us come back to our main strand of discussion.

6. PDP Minicomputers, Unix and C

Very dramatic changes started happening in the computer market during the third generation. Minicomputers in large numbers started landing there, with a much lower resource ability but a price tag of even less than one-twentieth of the large systems. Users not meant for scientific computing were the target group of these mass-produced machines. DEC, Digital Equipment Corporation started producing PDP, Programmable-Data-Processor, in 1959. This name ‘PDP’ too suggests the changing times. Naming the machines as ‘PDP’ involved avoiding the use of the term ‘computer’, because this term at that time was associated with images of voluminous, complicated and costly machines. A PDP 1 machine would cost even less than five percent of a 7094, with almost the same kind of performance in jobs other than number-crunching. And the last one in this PDP series was PDP 11 on which Ken Thompson, Dennis Ritchie, together with Douglas McIlroy, Joe Ossanna and others, created Unix, in Bell Laboratories. There are, mathematically speaking, ‘countably infinite’ number of resources on the birth of Unix and related things, both in the form of hard-printed books and electronic ones. But, the chief ones on which this book builds are Raymond 2004 and Tanenbaum 2002.

And there was a Multics continuity working here in Unix. Ken Thompson was there in the Multics team in Bell Laboratories. And apparently, he got a PDP 7 to work on, where he tried to build a trimmed down version of Multics, with a size that will be much smaller and trying to serve one user in place of many. This Multics was not written on PL/I, but directly in terms of assembly code. And this minuscule and cut-down version of Multics did really operate on the PDP 7. Unix got its name by Brian Kernighan, a co-worker with Thompson and Ritchie in Bell Laboratories, who contributed a lot to make Unix what it became. Kernighan started calling this system as ‘Unix’ as a joke-name, with a reference to Multics, as a version of Multics that was ‘one of whatever Multics was many of’, or, more subversively speaking, ‘Multics without testicles’, with a pun to ‘eunuchs’.

This OS in the making was soon transferred from the PDP 7 to a PDP 11, a newer machine in the same PDP series, higher in capabilities. Dennis Ritchie joined Thompson in the job, and gradually many other colleagues in Bell Laboratories followed. Now the problem was to decide on the language in which to build the OS. And one of the most important issues for them was that of ‘portability’, the concept of which we have already discussed. They wanted an OS that could be portable, and thus could be ported to another machine with another set of hardware and another set of device drivers. The developers of that time were already fed up with the process of writing everything anew once they had to change the machine and all. So they direly wanted a system that could run on every machine. By this time Thompson already had a language of his own, named ‘B’. This ‘B’ came from BCPL, Basic-Combined-Programming-Language.

BCPL was originally developed by Martin Richards, for writing compilers for other languages, and is no longer in common use. CPL, Combined-Programming-Language, the predecessor of BCPL, was created in 1960. Its plan was to cope with all kinds of problems, but was never fully implemented. Then in mid-sixties, in Multics project for experimental time-sharing systems, PL/I was used. In late sixties came BCPL or basic-CPL. BCPL, the inspiration of both the languages B and C, went on being used till late eighties, when it was replaced with C. In 1970, Thompson wrote a language called B, and first planned to write the whole Unix in B. B was a systems-language written for Unix on PDP 11, a language that was derived from BCPL and very similar to it, except syntax, and was the predecessor of C. But this implementation of Unix in B got full of problems due to some internal glitches of B. And so came C, the obvious logical name if the predecessor is B. After the changeover from PDP 7 to PDP 11, Dennis Ritchie built on B to create the new C language, in which most of the components of Unix were rewritten, including rewriting the kernel, that happened in 1973.

Very soon Unix became popular among scientists in different universities. They were asking for copies of Unix from Ritchie-Thompson, or, in other words, from Bell Laboratories. And due some legal hassles the company behind Bell Laboratories, AT&T, at that point in time, did not have any authority to make any business in software. So, just for a token fee, this new OS called Unix reached these universities. And the PDP machines were always already scientist-friendly. And so, for the scientist community in the universities, it was automatically a system of their choice with Unix running smoothly on PDP machines. And these systems, in a way, stand for one of the most dramatic inception in the history of computers, that was the double birth of Unix and C together.

This is true about every major event in history, it seems, that so many different, apparently unrelated factors come together, and make something happen. Here the most important of these factors were the PDP series, and Unix and C. This scenario made everything so simple for this new kind of OS. And because this whole system was written in C, as we have discussed in length in chapter three, the portability was so simple to achieve – just compile the system on a new machine with a new set of device drivers. Thus began the Unix era. And more or less here, the first part of this chapter ends, where we described some features of the environment where the process of primitive accumulation of the industry of electronic knowledge was bred. This age came to its full bloom in the birth of Unix, the process of the birth starting in 1969. Then came the era of taking away the

primitive freedom and cooperation operating in this world by the rules of market and capital. In resistance to this era all those supplements will start to happen, supplements that would finally lead to the text of GPL, and then create the very context of FLOSS.

7. Unix, Minix, Linux

In the last chapter we mentioned ‘closed source proprietary software’, like Microsoft Windows and so on. For the time being, let this phrase ‘closed source proprietary software’, stand for all the programs that many of us deploy on our machines, purchasing from companies like Microsoft, Adobe or others. All the applications that we run on our systems are programs. And the base they work and build upon, the OS or operating system, that is a program too. As we already know, these programs are not created ad-lib, that is, then and there. First, we write some high-level human-understandable code, that is, source code. Second, we edit and rectify this code from errors, that is, debug it. And finally, we compile it through a compiler in order to make it a low-level machine-executable program, transforming the source code text file into an executable binary file. This code-edit-compiler-binary cycle goes on rotating till we are satisfied enough about the binary, and then it finally becomes a program that works as an application on computers.

All the programs we run on our systems do pass through this cycle, at least once. And when we purchase a software from companies like Microsoft, be it an OS like Microsoft Windows 95 or XP or Vista, or be it an application like Microsoft Office Suit, they are all in binary. We are not even allowed to know the source-code behind these binaries, leave alone changing it, or adapting it to our custom demands, or redistributing the changed version. In a way, these ‘closed source proprietary software’ is in a final form – it is unalterable for us. When we are purchasing such ‘final’ software on CD or DVD or something, they carry the compiled binaries of the original source codes, and then we install these software. The action of installation means putting all those binaries and necessary library files and configuration files in a given predetermined format and path, within the machine’s hard-disk or elsewhere. This is done in such a predetermined way that enables the binaries to run, getting every necessary file in its proper expected place.

In contrast, software like GNU-Linux are called FLOSS, Free-Libre-Open-Source-Software, because, precisely, the sources are free and open. They are free and open to read, modify, redistribute and all. We will know the details of this freedom and openness later in this book. But, the point here is that, this kind of software is not ‘final’ or ‘unalterable’ in the sense of the last paragraph, which is not true for ‘closed source proprietary software’. If someone is using some closed-source software, and wants to change it, according to need or whim, or wants to create something new inspired by this one, he cannot do it. He has to start it anew, from scratch, from before the point from where the software companies themselves started their journey. Because everything within the journey of the ‘closed source proprietary software’ is closed, unalterable and given for anyone outside the company from whom it was purchased. And as we will discuss in later parts of this chapter, there is a very deep and painful irony working here.

The point from where these software companies started their ‘closed proprietary’ journey, that point in history was pretty open and free. And then these companies closed and

monopolized the whole thing. What the software companies tapped as primary resource was precisely the world of hacking: the primitive FLOSS, ruled by relations of freedom, friendship and cooperation. And then these companies tried to take away this freedom, and close it down by monopolizing the whole thing. As we said, they now wanted to produce pieces of software in which this free flow of knowledge is thwarted. And hence, anybody, wanting to do anything about any software, has to start from scratch. This process of starting from scratch is called ‘reinventing the wheel’ in FLOSS literature. In contrast, if someone is using FLOSS, he can use the available source code as his starting point, or his launching pad. In the form of this source code he is actually getting the continuity of knowledge in the field of software, his inheritance from what the monopoly companies are trying to disown him.

Let us mention and remind one important point here. Unix that gets created by Thompson, Ritchie, Kernighan and others in Bell Laboratories, is actually FLOSS. It is primitive FLOSS, long before even the invention of the term ‘FLOSS’. All the pieces of software involved here were free, libré, and open-source. So, obviously they were FLOSS, though without the name. Now, shortly after the birth of Unix, when different brand-name versions of Unix started to flood the market, not all of them were FLOSS. Each and every one of these different Unix versions, like BSD, SunOS, Solaris, HP-UX, AIX, Digital Unix, Irix, Ultrix and many more, could get created at all, because the original thing was FLOSS. And so, they had the liberty to customize and modify it according to their own taste, preference or need. Though, in a way, all this variety brought into being a kind of rigmarole that now called for a kind of standardization. And POSIX, Portable-Operating-System-Interface, served this purpose. POSIX is a family of related standards, defining all the necessary things – the API, Application-Programming-Interface, along with the shell and utilities interface, for all software compatible to different variants of Unix. API specifies the ways in which an OS supports the requests made by the running programs. In other words, POSIX standards mean interoperability between different brands of Unix. By POSIX, one piece of code written on one brand of Unix, can become portable to another brand – it can be compiled and run on the second system too.

After a long process of creation that started in 1969, the first public edition of Unix was released in 1971, where it was named as ‘Unix’. In the introduction to the Manual of this Unix, Thompson wrote, “... there are only ten installations of UNIX os, but more to come ...”. This Thompson-speak could be an all-time-great example of understatement, if only Thompson himself did really know beforehand what is going to happen: how Unix is going to explode. We are going to that in a bit, but before that let us mention here one interesting but unnecessary thing about the continuity and tradition flowing through these POSIX standards. This Unix Version 1 had only sixty commands, quite a few of them we almost get in exactly the same way on any recent versions of Unix or Linux, like the system on which this book is getting written, giving rise to a kind of personal pleasure for me: I am working on a piece of living history. I think any Unix or GNU-Linux user shares this fun, in the regular everyday commands that we type in on a terminal.

Anyway, two years before Unix was going to be born in 1971, in 1969, four events took place, which in a way changed the face of computing science, or science as a whole. Actually Peter Salus, in one of his lectures, mentioned about this surprising coincidence.

Here we are lifting this whole string from that lecture, Salus 2001, given on the occasion of the tenth anniversary of Linux, about the history of Unix and Linux. Four apparently unrelated events took place in the year 1969. One, reportedly, man landed on moon. “Reportedly”, because in recent times serious doubts have been uttered about whether it took place at all. Two, in August-September, Thompson and Ritchie generated the first structure of the would-be Unix operating system. Three, in December, four IMP-s were implemented. IMP or Interface-Message-Processor was the name of first generation routers, the connecting veins of ARPANET, Advanced-Research-Project-Agency-Network, or the skeleton of the embryo of the Internet that was yet to come. And, finally, on December 28, Linus Torvalds, the man behind the first Linux kernel was born.

Let us come back to Unix. Unix version two was released in 1972, and C was created the same year. In 1975 Unix version six was released. The total number of sites in the then ARPANET was even less than a hundred, and this Net thing will have to increase manifold before an event like Linux could happen, after sixteen more years, that is, in 1991. BSD, Berkeley-Software-Distribution, version one, was released in 1977. This was also known as BSD Unix. This was another version of Unix that built on the Bell Laboratories version. This 1977 version of BSD Unix, together with a text editor called ‘Ex’ and a Berkeley edition of a computer language called ‘Pascal’ – this whole package did cost only twenty dollars. Very soon the situation was going to change as big companies start to land in the software market, not only hiking the prices at a cosmic scale, but truncating the flow of human knowledge from one generation to another. They, once again tried to the same thing that once the alchemists did, partitioning and prohibiting free flow of knowledge, in their era in human history.

1979 saw the Unix version seven from Bell Laboratories. This was, in the true sense, the first really portable OS. It ran like anything on machines of the PDP series and many other models built by Interdata. Interdata was a computer company founded in 1960 and producing machines loosely based on IBM 360 architecture at that time. And in this very year, in June 1979, at a remove of only two years from the release of BSD Unix in 1977, a crucial USENIX meeting took place. USENIX was a Unix-Users-Group founded in 1975 for the study and development of Unix and similar systems. In this USENIX meeting, AT&T announced that from then on they are going to charge from twenty thousand to forty thousand dollars for Unix per CPU. Just compare this price with the paltry sum of twenty dollars only two years back. So, Andrew Tanenbaum, who already planned for a syllabus of OS in the coming year, discovered it to be an impossibility to take OS classes that year with Unix as the OS, and had nothing else had nothing else to do but something on his own. He created a new OS called Minix. Minix was a Unix-like Operating System based on a special kind of kernel called microkernel-architecture, created by Tanenbaum. This Minix would later inspire the creation of Linux kernel.

1980 saw the release of version four of BSD. From then to 1990, one by one came releases of BSD numbered in a peculiar way, 4.1, 4.1A, 4.1B, 4.1C, and so on. This was another symptom of the cramping of the software world under the pressure of the emerging licensing system of the big companies. AT&T allowed Berkeley University to release version 4 for academic research, but did not allow any new version to be released, other than minor updates. And hence, BSD versions had to be named like that, to show that they

were just minor updates to the version 4, not any major change.

In January 1984, Richard Stallman started the GNU project. And this very year he released his GNU-version of Emacs, a text editor and developer tool endowed with many other functions. In the 1985 May issue of 'Dr. Dobbs's Journal', Stallman published the manifesto of GNU. In this manifesto, he looked forward to a free OS, and all the tools that one needed to create a thing like that. These tools were, chiefly, the compiler and other developer tools. In this world of software strained under the weight of Unix and other proprietary licenses, later, a dramatic change would be brought in by GCC, the GNU-Compiler-Collection. But, it would take three more years, from the publication of the GNU manifesto, for the first beta version of GCC to release, in March, 1987. Though this version was far less than today's GCC, which can compile programs in almost all languages, and is a standard part of all Linux Distributions. However small compared to the GCC of today, the birth of it would emerge as a very dramatic break at that time. But, this would happen three years later.

For now, in 1984, AT&T's way of closing down the freedom in software world was already becoming prevalent. The open space of computer science and computing was getting closed, under enclosures of proprietary and prohibitive licenses. Sun Microsystem, that till then sold all their software together in a package, now for more profit, started selling them separately. And for the C compiler, Sun fixed a price abnormally high. The organization called FSF, Free Software Foundation, was not still there, but, the works of Stallman in favor of freedom of software had already started by then. Richard Stallman started working on a compiler from 1984, as we said. We will elaborate this idea later: that the movement for free software came actually from a restorative gesture, because the till then open space was getting closed down and partitioned by proprietary motives. Maybe this will remind some of us about the Marxian concept of 'primitive accumulation' and the role of land-enclosure in that field: how the artisan producers got expropriated.

Minix was released in 1986, and the whole source code of Minix was released too with Tanenbaum's book, "Operating System Design and Implementation" in 1987. Now the whole source code of Minix is available on the Net. But, till then, the Net was yet to come in a meaningful way, and hence, the source code was released on floppy disk. By this time, different Unix-like OS were getting built. And the number of such systems was increasing. In 1990 IBM released AIX version 3, built on Unix System V release 3. AIX, Advanced-Interactive-Executive, was IBM's own version of Unix. Before this AIX version 3, different versions of AIX started coming out 1986 onwards. They were releases built on prior versions of Unix System V. This Unix System V, commonly abbreviated as SysV, is one of the major flavors of Unix, in terms of use by software developers of that time. And it was so influential in all other later flavors, brands and makes of Unix, that even today we get footprints of this SysV in different flavors of FLOSS Unix, that is, Linux.

1991 is the year Linux was born. On August 26, 1991, at 11:12 AM, Linus Torvalds wrote a mail to a Usenet newsgroup comp.os.minix. Let us quote some portions from that mail.

I'm doing a (free) operating system (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since april, and is starting to get ready. I'd like any feedback on things people like/dislike in

```
minix, as my OS resembles it somewhat (same physical layout
of the file-system (due to practical reasons) among other
things). ... I've currently ported bash(1.08) and
gcc(1.40), and things seem to work. ...
```

Let us quote from portions from another mail by Linus Torvalds to the same newsgroup on October 5, 1991, at 8:53 PM.

```
I'm working on a free version of a minix-lookalike for AT-
386 computers. ... I've successfully run bash/gcc/gnu-
make/gnu-sed/compress etc under it. ... Full kernel source
is provided ... These sources still need minix-386 to be
compiled (and gcc-1.40, possibly 1.37.1, haven't
tested) ... I can (well, almost) hear you asking yourselves
"why?". Hurd will be out in a year (or two, or next month,
who knows), and I've already got minix. This is a program
for hackers by a hacker. I've enjoyed doing it, and
somebody might enjoy looking at it and even modifying it
for their own needs. It is still small enough to
understand, use and modify, and I'm looking forward to any
comments you might have. ...
```

‘386(486) AT clones’ in the first mail, and ‘AT-386’ in the second, refer to a particular kind of architecture. This was IBM PC AT (Advanced Technology), also called as PC AT or PC/AT, primarily designed around Intel 80286 CPU. The ‘386’ refers to Intel 80386 CPU, while the ‘486’ refers to the next one in this CPU series by Intel. We discussed a little about this x86 architecture in the last chapter. The terms ‘gnu’ in the first mail, and ‘Hurd’ in the second, refer to GNU Hurd, the FLOSS OS that Stallman already planned for. GNU as an organization at that time was working on it. We should note that, Torvalds creates his kernel, with Minix as the model in his mind. And, he refers to GCC and other GNU developer tools for the building of the system and the binaries. We should also note that, the whole source code is open. And Torvalds calls for the cooperation of the hackers’ community. We mentioned earlier about the positive aura of the word ‘hacker’ in the world of computing at that time, the aura that was exactly inverted, vesting it with negative surplus meanings, by the illiteracy and political nature of media.

In his seminal essay Raymond 2000, ‘The Cathedral and the Bazaar’, Raymond did aptly catch the dynamics of this community method of software building, working through multiple layers of cooperation. But, we must not forget here the presence of the Net as a major ingredient. The Net actually made all this cooperation and community possible at all. Without the Net being and becoming what it is, the Linux process could never take place. In 1975, the total number of sites on the Internet was less than a hundred. In 1981 this number grew to 213, and in 1994 it became 7.5 million. At the moment Linus Torvalds was sending this mail, this number was more than 0.5 million. So, in a way, Linux grew together with the Net. The internal dynamics of the Linux process of community cooperation is inherently integrated with the workings of the Net. We will raise this point once again, later in this book.

1992 saw a long debate between Andrew Tanenbaum and Linus Torvalds. Others from the computing community got involved in it. This was a debate with quite some degree of bitterness. It was all about Linux in particular and the characteristics and features of OS in

general. This debate is quite interesting, and important too in understanding the Linux process. A very good annotated account of this debate is given in the Appendix of DiBona 1999. Anyway, history has proved that Tanenbaum, the legendary teacher of computer science and the maker of Minix, was actually wrong in his evaluation of the possibility of Linux. Around 1992, different Linux distributions or ‘distro’-s started to emerge. These distributions, often called ‘distro’-s on different Linux mailing lists, consist of large collection of FLOSS applications like word-processors, spreadsheet applications, Internet browsers, mail clients, and different multimedia applications, and so on. All these applications in a distro come together inside an OS, which is built on top of the Linux kernel. And usually the main libraries necessary for the working of these application binaries come from the GNU project. The GUI or the Graphical User Interface in these distro-s are derivatives of the works of X.Org foundation, formerly called XFree86. All these components are FLOSS, as we said before. These distros are like different flavors of the same old ice-cream, that is, the Linux kernel and a POSIX-compliant system of applications woven around it. POSIX looks after their Unix-ness. So, they are a FLOSS continuity of the Unix tradition.

Obviously the distro-s that started emerging around 1992 had quite a long way to go to reach the state what they are in, at the moment of writing this book in 2010. The oldest Linux distros include “Interim Linux”, “TAMU” and “SLS”. Interim Linux came from MCC, Manchester-Computing-Center, a body of the University of Manchester, England. Interim Linux was released in February 1992. TAMU Linux came from people in TAMU, Texas A&M University, USA. SLS, Softlanding-Linux-System, was released in mid-1992. These distro-s contained more than the Linux kernel and basic utilities. They included TCP/IP, basic tools of networking, and later X-Windows, the GUI too. One of the earliest Linux distributions still maintained is “Slackware”, a descendant of SLS. Slackware was first released in 1993. SuSE, *Software-und-System-Entwicklung*, “Software and System Development”, was releasing German translation of SLS/Slackware software from 1992. First commercial version of SuSE was released in 1993. Red Hat Linux, which evolved into “Fedora”, on which OS this book is getting written, released its version 1.0 in November, 1994. That same year X-Windows started getting shipped into Linux distributions.

In September 1983, Richard Stallman wrote the original announcement of GNU – a project that finally started in January 1984. This announcement carries a vision of Stallman about a complete Unix-compatible software system. This vision has now come true, in the living reality of all the Linux distro-s. But, the contribution of GNU was much more than this vision and the libraries and development tools integral to the Linux system, that we mentioned earlier. In fact, the continuity around Linux is the Unix continuity that started around 1969. And, the historical break about Linux, if there is any, is more in the GNU GPL than even the Linux kernel that was born in 1991. In the coming chapter we are going to discuss the details of this intricate historical interplay of power politics and computing.

One point to note here is that, both Minix and Linux were Unix-compatible. In fact, they had to be like that. At that point of time, any new OS had to be like that, like another one in the series of Unix-clones, where the commands and crafts of Unix will apply as they do in Unix. Because, the whole community of hackers, developers of the system, and the people

that knew computing – they only knew Unix. If this similarity with Unix was a bit less than true for Minix, due to its stringent spectrum of functionality, it was a total truth for Linux. Keeping aside all the legal questions concerning marketing of software, names, trademarks, licenses and all, Linux is Unix. There are debates here, concerning the very definition of Unix. But here we are talking about the living Unix tradition of the people who did it. In terms of that tradition, Linux is the continuity of the same thing. Though, Linux very deeply transformed the Unix tradition while taking it up. Linux brought in a very large user base into it, in a scale unthinkable in terms of Unix proportions. And non-hacker users became a part of this community too, together with the hackers. By the time Linux emerged as one of the major determinants of the history of computing, the scenario had undergone a dramatic change in terms of hardware, and more importantly, in terms of the Internet. The PC age had come, together with instant transmission of copying anything electronic, through the omnipresent link of the Net. Linux, as it stands today, is a full-fledged highly functional OS.

In the last chapter we have seen, how the layers of software reside one above another in an OS. In terms of that scheme, in a modern Linux OS, the Linux kernel resides in the lowest layer above hardware. This kernel deals with all the device drivers, device files, and other system things like memory-handling and all. Above this kernel layer is the central layer of system programs. And almost everything in this layer in a Linux system is from GNU. The GCC, the core system utilities, the Bash shell, and even the libraries that these binaries operate with, are predominantly from GNU. Over this central layer is the layer of applications, which is now filled up with, in a super-abundant way, with lots and lots of FLOSS. Obviously, the GUI things are very important here. With the base supplied from X-Windows, there are lots of Graphical User Interfaces like Gnome, KDE, Xfce and so on. Then come the application software groups. One important one of which is LAMP, Linux-Apache-Mysql-PHP, integral pieces of FLOSS. And come lots of applications of many kinds like office, graphics, multimedia, network, games, and so on. Linux, or better, GNU-Linux, was all through an evolving cannon. GNU-Linux was a cannon that grew and developed 1991 onwards. Each element of this cannon was woven around the Linux kernel, developed with GNU tools. And more importantly, they carried in their heart an entirely new order of property. It was no more private right, private property now started protecting public right – GPL made it happen that way. But, we need to traverse a lot more before we start handling the property relations in terms of Hegel's philosophy of right.

The very Linux kernel itself is a good marker of the community contribution in Linux. We know, Linux kernel was born in 1991, the size of this kernel by Torvalds was 63 Kilobytes. Version 1.2 of Linux kernel came in 1995. Its size was just above 250 Kilobytes. And as we said, it went on evolving and growing. The number of Linux users was hiking, and they were bringing in newer kinds of hardware into Linux OS, adding newer kinds of device driver. This went on hiking the portability of Linux kernel, and its size too. The size of the Linux kernel in its crunched form, for a recent distro is near 3 Megabytes. The collective effort of the Linux community shows there in the difference of the size, from 250 to 3000. A lot of developers labor went there into the added size, a lot of real history of real use of real systems – a history that was changing all along the line.

As the concluding comment of this very long section, let us cite here an infamous lawsuit

coming from the birthplace of Unix. This lawsuit contributed a lot in popularizing Linux and bringing it into mainstream computing. The lawsuit took place between USL and BSDi. USL, Unix-System-Laboratories, was a division of Bell Laboratories responsible for further development of Unix. BSDi, Berkeley-Software-Design-Incorporated, was a corporation created for developing software, selling licenses, and supporting BSD/OS, a commercial and partially proprietary variant of BSD Unix. In CSRG, Computer-Science-Research-Group, in UCB, University-of-California-Berkeley, a lot of activity within the hacker community was going on at that time. CSRG had a license from AT&T on the source code of Unix. After a lot of extension and modification of the AT&T Unix source code, the BSD community of hackers started removing the original AT&T source code and replacing it with their own. The net result was released in 1991 as “NET-2” under BSD license, which was an almost complete Unix system. BSDi filled in the missing pieces in the “NET-2” source code, and ported it to Intel 386 architecture, selling it as “BSD/386”. In the 1990 lawsuit, USL claimed that, UCB violated the license, infringing on copyright and divulging trade-secret. The case was finally settled outside the court, in 1993, with USL and BSDi agreeing not to fight any more on BSD. Henceforth BSD flourished into quite a few flavors like FreeBSD, OpenBSD, NetBSD, DragonFly BSD and so on, each with some specific features meant for specific user groups. Seen this way, AT&T was a major contributor to the success of Linux in the primary years of 91 to 94, when the future of BSD became doubtful due to the lawsuit. It was extremely damaging to the BSD hackers, and this helped the early popularization of Linux as an alternative in the free and open hackers’ world.

8. The ~~Unix~~ FLOSS Tradition

Some elements of this section and the later sections to follow in this chapter are highly indebted to the book Raymond 2004, “Art of Unix Programming”. It is one of the best books ever written about Unix, Unix culture, and Unix way of doing things. We borrow from this book the viewpoint of the hackers’ community. Then we retell from this viewpoint, the story of closing down the source, and the struggle for liberating this source by reopening the closure. Raymond 2004 gives a brief exegesis of the strength of Unix. Let us lift a few ideas from there. And remember, here Unix means a tradition. Linux is a part of that same tradition, as we told in the last section. In terms of POSIX specifications, Linux is another brand Unix, only a *different* brand that is free – free as in freedom of speech. This tradition of Unix is the tradition of Linux too. We have already called this tradition as primitive FLOSS, or a FLOSS before the birth of the name of ‘FLOSS’. We mentioned in brief, how the free and open community of hackers resorted to and zeroed in on Linux, under the crisscross of pulls and pushes of different proprietary licenses and all. In the next chapter we will discuss these in details. Now let us summarize the points of strength of this hackers’ tradition, call it Unix or call it FLOSS. The important point is that it *is* a tradition, in the truest sense of the term. It has a continuity of knowledge that is never fully captured in the texts of the discipline. This is transmitted from one generation to the next, together with some values and some emotions. And Linux hackers and users together, go on carrying this continuity.

We already mentioned – this *new* kind of community is a Linux phenomenon: unification

of the space of hackers and users. Actually a simple inertia of scale worked here. The Unix tradition, before Linux came in, consisted only of hackers who used the system and worked on it. But, a massive manifold increase in the user-base generated this new space by including non-hacker users on a scale that was just unthinkable some ten years back. On innumerable occasions, the more technical users or hackers come to the help of the users, and the users give them the feedback, thus streamlining the directions of development. In the billions of pages of Linux Howto-s, we can always get the confluence of hackers and users. But, the one area where it becomes felt more than anywhere else is the space of mailing lists. The innumerable mailing lists in Linux carry and hold an important part of this tradition. As we said, Linux could never happen this way without the Net. This is a very living tradition. And it is continuing from the time before the birth of Unix, and is now carried forward by FLOSS. We will come back to many more details of this community, once again, in the last chapter. Now let us summarize the strengths of this FLOSS system and tradition, borrowing a lot from Raymond 2004.

One, the extreme durability of FLOSS tradition shows up in its continuity from 1969 till date, from a time when there was no PC or Workstation or even microprocessor chips. Two, FLOSS enjoys maximum diversity in the sense that no kind of hardware is there on which FLOSS does not run, and no kind of use is there that is not possible on FLOSS systems. No other OS can come even near to it in these two respects. Three, the C language, a central presence in FLOSS, has become widely naturalized on all kinds of systems. C has become almost the sole agency of system programming, and obviously something that software engineering cannot replace with anything else. C may very well be called the mother of almost all other programming languages. Together with C, two other things are now omnipresent in all kinds of computer systems. These two are contributions of this FLOSS tradition too. They are the tree-shaped file ‘namespace’ with directory nodes, and the concept of pipelines for connecting between programs. This FLOSS tradition now spans over quite a long history. So many kinds of hardware have passed, and so many systems, but all these years FLOSS tradition is running on, doing all kinds of works on all kinds of hardware. FLOSS is the only unchanging thing in this computer science world of dramatic changes, where, roughly speaking, a half of what one knows and does becomes obsolete in every 18 months.

This FLOSS tradition of freedom of knowledge and openness of cooperation was all-pervading in the beginning. This was later closed in, curbed, cordoned off and suppressed. This tradition was to return and revert back to, once again, later, in the form of GNU-Linux. And that is why when we call this tradition as a FLOSS one. This name true in two senses. One, it was an Unix tradition that was FLOSS from the word go, much before the birth of the word ‘FLOSS’. Two, then this Unix no more remained in this tradition in the sense that it no more remained ‘FLOSS’. And now the onus was on FLOSS to regain this tradition. And that it did, in the form of GNU-Linux. So, in every sense of the term it *is* FLOSS tradition, and we call it that way. Calling it Unix would be misreading history.

9. FLOSS Tradition and Counter-Culture

We already know how it all started around 1969. Multics-return Thompson was inventing the Unix system on a PDP-7, when his flank was strengthened by Ritchie, followed later

by McIlroy, Kernighan and others. The interesting point is, the primary focus of Thompson was a game called ‘Space Travel’, that he developed on Multics. And now he needed a new OS to play this game of simulated space-travel. First he ported the game to Fortran on a GECOS system. GECOS or General-Electric-Comprehensive-Operating-Supervisor was an OS for mainframe systems developed by General Electric around 1962. Thompson then proceeded to port this game to a PDP 7, when Ritchie joined him. This game, Space Travel, was an important circumstantial factor that accelerated the development of Unix. In the process of porting the game to the assembly language of PDP 7, they wrote the code that later grew into the embryo of Unix. The journey of Unix thus started, and an army of talented hackers started to gather around them.

Maybe, the fact that it all started with a game was not entirely pointless. “The Unix Programming Environment”, Kernighan and Pike 2001, is one of the greatest books in the cannon of FLOSS tradition. Both Kernighan and Pike are from the team that developed Unix. This book is a representative of the FLOSS culture in its true entirety and a must read for all would be FLOSS developers and active users. Anyway, the smallest subsection of the book is named ‘Games’ in Chapter 1 of the book. Let us quote the entire subsection, with particular attention on the first sentence.

It’s not always admitted officially, but one of the best ways to get comfortable with a computer and a terminal is to play games. The UNIX system comes with a modest supply of games, often supplemented locally. Ask around, or see Section 6 of the manual.

None can help but notice the mischievous tone. When the first sentence is commenting about the ‘official’ way, it must be ‘unofficial’ in itself. And this ‘unofficial’ undertone is accentuated by the informal use of ‘It’s’ in place of ‘It is’. And this quotation is not from a students’ leaflet, this is one of the most brilliant books of the FLOSS tradition. Everything in this subsection, the size of it, the tone, the choice of words, the syntax – represents the ‘unofficial’ playfulness. This unofficial tone and playfulness are still a part of FLOSS tradition, and the community it represents. Any representative text of FLOSS tradition carries this tone. And in a way, this tone is very integral to the FLOSS community. This playfulness, informality and some obtuse questioning of authority (that maybe we know better than the ‘official’) – the elements of the FLOSS community all are already there.

We can read the footprints of *counter-culture* here, by back-tracking into the question: whom does this ‘we’ represent – this playful, informal and authority-challenging ‘we’? It will become gradually more distinct through the coming pages of this book why we are assigning such a special name, ‘counter-culture’, to these attributes. Actually there are lots of elements of *counter-culture* involved here, elements that grew in size and depth during the sixties. And these things then evolved and grew in diversity and intensity through the FLOSS tradition, things that acquired a theoretical basis in the works of GNU through GPL, and actualized itself through the building of Linux kernel and the hordes of FLOSS thereafter. This tone is, very certainly, one of the first major supplements that accumulated around the FLOSS community, together with, obviously, the principles of freedom and cooperation in the process of knowledge accumulation.

Even the elements of *playfulness* and *informality* represent the iconoclasm too. While the

authority likes to be serious, 'we' like to be playful. While the authority is formal, 'we' break the forms by getting informal. If we now question ourselves, whom does this signifier 'we' readily brings to our mind, it will not be long to get the answer: a student community. A community within which the phrase from the last sentence of the quotation, 'ask around', readily gets its meaning. This answer, a student community pursuing free flow of knowledge, gets progressively more meaningful as we become conscious about the backdrop against which this FLOSS thing was happening. What a decade it was – the sixties, the decade of student unrest, a decade full of different kinds of unrest. A decade that in America continued sending predominantly unwilling young men to a nationally unwanted war. A decade through which student movements fostered all through Europe and America. A decade that ended with the historical May 1968 in Paris, and the inception of Unix in the very next year. And this was the decade of primitive FLOSS too. Unix was nothing but the epitome of this primitive FLOSS. And then this primitive FLOSS came under larceny by the rules of market. And it would take a GPL and a declared FLOSS to regain this freedom and cooperation in such a way that nobody can take it away any more.

To get all these in perspective we have to recapitulate some elements of American history, things that were happening around the birth of Unix, and thereafter, around the making of FLOSS tradition and community. But let us remind ourselves one thing from the very start of the discussion, that not even a single component of this whole process was ever conscious. The people involved in could never know it from before, what functional value any component is going to acquire in a much larger frame of space and time, in terms of the final context of FLOSS that they are going to create, the context that we witness today. They did all what they did, just because it was the call of the time, and they were honest and earnest enough to respond to the call. Each response of each respondent was often very small and insignificant in a stand-alone way, generating an endless series of minuscule supplements. Then, over time, these small and apparently insignificant supplements did merge and mingle together to generate a very coherent and meaningful mural to readers like us, reading the history of their time and actions. The birth of Unix started in 1969. But the preparation for it was going on for years before it. This year 1969 is at the tip end of this decade. And the earlier decade ended in 1959, the year the Vietnam war started. Then came this decade of sixties. And through this decade a lot of things happened that actually went on giving fuel to the *counter-culture*.

Unlike the Linux kernel, the *counter-culture* thing is nothing monolithic and well-defined ever at all. At its best it is a conglomeration of some tendencies. These tendencies, anti-power ones, go on generating a series of scattered bastard texts in the form of lectures, songs, pictures, graffiti, and so on. And in real life all these bastard texts merge and change over from the discursive space to the real, that is, they become components of a lifestyle. A lifestyle that challenged the old society and its values and the traditional morality. The persistent focus of all *counter-culture* activities has always been the Education System. Youth was the dominant domain of definition for all the counter-cultural functions. And this youth is usually the educated youth that can relate to all the academic things and their underlying ideological moorings.

May 1968 in France was the biggest event in Europe since the World War II. It led to a chain of student struggles that brought the then Charles de Gaulle Government on the brink

of collapse. In May 68, the *counter-culture* activities under the leadership of the students got mixed up with workers' struggle too. But, the Communists were always in an uneasy relationship with these things. In most of the cases they consider the elements of *subversion* involved there as degenerate or something. And, the French Communist Party too considered these students as anarchic and "false revolutionaries that must be energetically unmasked", Cliff and Birchall 2001. The term *counter-culture*, as a sociological term, came to prominence around the American society and other societies in sixties and seventies. An important component of it was the 'hippie' or 'flower-power'. This hippie or flower-power components meant a very different way of life, and some protests and movements around civil rights, racial rights, sexual rights, women rights – protests against war in particular, and social norms of power in general. Let us remember the lack of endorsement and the lack of camaraderie to *counter-culture* by the so-called 'official' resistance to power, the communist and Marxist ideology, though they have some tendencies in common. This point is going to be very important in our later chapters.

So, rejected by the mainstream resistance, these *counter-culture* things were absolute bastards, without any genealogy of their own, neither from the Father, that is, power, nor from the Anti-Father, that is, the Communist Resistance led by the workers. Let us remember the things we said in chapter two in the theoretical model of context-text-supplement politics. We discussed the theme of bastard texts searching for a father, that would generate a surrogate genealogy to all these bastard texts, and thus create the very context of reading the text that becomes the surrogate father. But, that will take time, more than one and half decades before the text starts getting written, in the form of GPL, and more than twenty years before the context generated by the surrogate father starts to materialize and actualize in the form of the Linux kernel and FLOSS movement. And this phenomenon of getting disowned by the Anti-Father, the Marxists, is going to be extremely important in understanding the moment of *differend* in context of GPL.

Let us come back to the primitive FLOSS context with its high point residing in Unix. All through the decade that in the final year provided us with Unix, some of these *counter-culture* issues were very important social ones in America. Hippies flourished, and so many kinds of dynamics started to emerge. These dynamics contributed to and strengthened *counter-culture*. As we have said, Vietnam War supplied the political-social-ethical backdrop all through this decade. Against this backdrop came the assassination of John F Kennedy in 1963. For so many Americans it was actually the biggest shock since World War II. Still more shocks were to come. In 1968, two more major assassinations took place, those of Martin Luther King Jr. and Robert Kennedy – assassination of the two leaders of the two movements that were shaking America all through this decade. Martin Luther King Jr. was the leader of the struggle for the rights of the African-Americans, and Robert Kennedy led the fight against American participation in Vietnam War. Both these movements became so strong that the system could not take it any more.

This decade was full of very active student movements from the very year 1960. And one of the major sites of students struggle was UCB, University of California Berkeley, the place that would later become a mecca of FLOSS tradition through BSD and all. Particularly from 1964 onwards UCB became the center of anti-war and other *counter-culture* resistance to such a massive scale that was never witnessed before in American

history. It is meaningless here to go into the details of the *counter-culture* and other forms of resistance, particularly in the universities of America all through this decade, myriads of authentic texts are there around it. Let us quote here a passage from one of the best books in this line, “The Year the Dream died: revisiting 1968 in America”, Witcover 1997.

... American intensification of the war had ignited campuses from New England and New York to California. It triggered scores of protest marches and the public burning of thousands of draft cards* by students demonstrating their commitment to stop the fighting, even at the risk of government prosecution. Many wore their hair long, dressed in hippie garb and used marijuana or stronger drugs, to the annoyance and often open hostility of their sedate elders and blue-collar, "straight" contemporaries. The nonviolent ones were called "flower children" who preached love as the ultimate answer to every problem. ...

This passage represents in a vivid way the things we just said, and contains both the elements, enmity and love, among the *counter-culture* activists, in their struggle against the power of the Father. we will see these elements to acquire great importance in our analysis of GNU GPL from a Hegelian viewpoint, in our later chapters. The context of ‘flower power’ and anti-war struggle are very crucial for the way we are going to interpret ‘love and competition’ among human beings. The context of ‘flower children’ deserves a few words. “Flower Power” or “Flower Children” were slogans used by the hippies in sixties and seventies. Flower is the counter of power, standing for the ideology of friendship and love, in context of the opposition towards Vietnam War, and any war in general. Flower stands for the ambiance of friendship in face of the prevailing enmity. Later, in the context of Hegel’s theory of Right, we would see these two apparently binary opposites of enmity and friendship coming together and generating new categories built through GPL in the broader context of market and civil society.

The ambiance of counter-culture in the late-sixties America is very sensitively captured in the film ‘Zabriskie Point’ by the master filmmaker Michelangelo Antonioni. Remember, this was the time when Ken Thompson and Dennis Ritchie were going into labor before the birth of Unix. Zabriskie Point released in February 1970. During 1969, both Unix and Zabriskie Point were in the making. The counter-culture factor in American universities and elsewhere had got so strong that MGM, Metro-Goldwyn-Mayer, the movie giant, wanted to appropriate the market, and invited Antonioni to make this movie on the counter-culture activities in America in the late sixties. Let us mention a sequence from ‘Zabriskie Point’ here that will be extremely relevant for our theoretical model. In the police-ruled campus of the university, one student is captured by police, and the officer asks him for his name. The officer asks, “Name?”, the student answers, “Karl Marx.” The officer asks, “How do you write it”? The young man replies, “M. A. R. X.”, and we see the camera to follow the typing by the officer, where he writes the ‘Marx’ part as directed by the student, and then the ‘Karl’ part as ‘Carl’. This visual metaphor dramatizes the moment of Americanization of the word, and thus stands for the very cultural distance between power and counter-culture activist. In the pathos of this distance, one’s words, even the symbols,

* Let us mention here, meant for all those not familiar, the ‘draft card’ thing stands for the official document of conscription: compulsory enrollment in military service.

get defaced, that too so unknowingly. But, this very metaphor represents an error too, on part of the filmmaker, in reading the history of these times. How much maybe the prevalence of Mao, Lenin and Marx in the conversation of the students, this counter-culture was something very different from Marxism. In that sense, ‘Marx’ is an erroneous metaphor here. And this gets more interesting, when we see the same error, though much more blatantly, committed by one of the brightest programmers of all times, Eric Raymond, when discussing about GNU GPL and other efforts by Richard Stallman. But we will come to that later.

As we said, it would take a lot more time, for the elements of *counter-culture* to coalesce and coagulate around GPL at such a scale that it may be discussed vis-a-vis counter-hegemony. But, the elements of *counter-culture* were always there, in the culture of comradeship and community among the hackers, in the tendencies to ignore the rules of market and capital, the rules of Father. Or, even in the subversive playfulness of the cultural ambiance of FLOSS. But, all these cultural, political and ideological factors could reside there in the FLOSS tradition due to the very primary condition that source code was free and open and the whole hacking community together was nurturing, developing, and sharing it. This FLOSS dimension was repressed by the workings of capital, to return once again as a possibility through GPL, and then to get actualized in the unending and till-continuing chain of events: the Linux kernel, the Linux distro-s, and FLOSS as a whole.

Forrest Gump by Robert Zemeckis, Zemeckis 1994, is an excellent film, narrating the continuities of American social history around the Vietnam war and later, in a very sensitive way. Interestingly, it deals with many of the elements that we mentioned in this chapter. It is natural too. Vietnam war is the central motif of Zemeckis 1994. The very name of Forrest Gump carries the footprint of Klu-Klux-Klan, or the history of apartheid and racial hatred. Elvis Presley, the King, one of the biggest cultural symbols in history, lodged as a paying guest of Forrest’s mother in Forrest’s childhood. And it was the King himself who encouraged Forrest to move and dance, a momentum that this crippled boy needed so much, in order to not remain crippled any more. The girlfriend of Forrest starts singing Dylan’s ‘how many roads’, that too in nude, as a part of the process of discovering herself. There are so many of these different elements weaved in an artistic and sensitive way all through the film. Now, the point is, Forrest Gump, after the war is over, invests his money in Apple and becomes a billionaire. We are going to witness in the coming chapter that there were more to this history than Apple or Microsoft. We are going to read the history of repression under these monopoly powers, and how GPL made happen an entirely new kind of political economic resistance that mankind has never seen before.