How to run C++ applications on a bare PC?

Dr. Ramesh K. Karne
Towson University
rkarne@towson.edu

Karthick V. Jaganathan
Towson University
kvenka1@towson.edu

Tufail Ahmed
Towson University
tahmed3@towson.edu

Abstract

Most of the computer applications today run on a given operating system environment. The application programs written in a programming language such as C++ are intertwined with operating system and environment to run on a given machine. Thus, a C++ program requires a processor such as an Intel Pentium and an Operating system such as a Microsoft Windows. Why do we have to run applications in such a constrained environment? It may be because, that is how evolution of computing happened since the inception of personal computers in the 80's. In this paper, we describe details on how to run C++ applications on a bare machine. We provide some benefits of running applications on a bare machine without any operating system. We present some sample applications that are built to demonstrate the capability of running C++ applications on a bare machine. Finally, we describe our future research direction that may potentially offer a revolution in computing architecture and application development.

1. Introduction

Running computer applications on a bare machine is not a new concept. That is how computing started decades ago, where a program is loaded manually with toggle switches, and run by pushing a run button. We are not talking about that in this paper! However, our computer systems are complex and they have grown out-of proportion creating a large semantic-gap between applications and hardware. For example, latest Microsoft XP operating system (OS) has 40 million lines of code. It is time now to revisit the computing evolution and seek solutions using new paradigms. The OS and environments have been changing rapidly since last 30 years thus making things obsolete before they can be even productive in their life span. For example, Microsoft released more than 20 primary releases of OS in last 25 years. Each language and environment goes through a new release every six months. When an OS, or its environment changes, it has ripple effect resulting in legacy applications. How could we stop such proliferation of products and technologies and make applications stable? Without providing any validation, one possible approach is to avoid operating system and its environments. This is being done slowly in some areas of computing without being noticed. For example, Exokernel [2] approach indicates that current operating systems are inefficient and some applications such as server operating systems [4] run faster when kernel code is moved into applications. Microkernel [3], nano-kernel [14], OSKit [11], and other similar approaches were also motivated by improving performance by making the OS lean. However, none of these techniques volunteer to remove OS completely!

In addition to the pioneering research of Exokernel architecture [2], there are many efforts in research to develop efficient operating environment for running applications. Most recently, *sandboxing* mechanisms [10] are proposed that allow applications to configure and deploy services at user level. Individual efforts [1] provided free operating system resources and code to de-emphasize the role of commercial OS, but they were not successful in eliminating the centralized OS architecture. The OS-Kit [11] approach advocates running applications on a bare PC, but still requires a minimal kernel to assist the applications and they do not provide C++ API for the user. The Choices [12] project indicates that, it is possible to develop an object-oriented operating system that runs on a bare machine across many hardware platforms including: Intel and Sun. In Choices, the object-oriented OS provides the resource sharing and multiplexing of applications. Similar ideas have been around with Embedded ADA programming language [13], where the language provides OS constructs such as process, timer, and memory management functions. This approach reduces the dependency of centralized OS, thus providing programming capabilities to the user at the language level. The SWILL [8] project developed a lightweight programming library that adds a simple embedded web server capability to C and C++

programs, but still requires OS. This programming library provides efficient API to build server applications.

When an OS is completely removed, now a computer application can directly communicate with the hardware. There are many applications that need such computing paradigm where they have a total control of their program. Some of the common applications such as email, Web server, text processing, spreadsheets, VoIP Phone, gateways, routers, and so on can be run on bare machine. We have presented an application object (AO) architecture [5] to build computer applications that run on a bare machine. Our research was partially funded by NSF to explore possibility to develop the application-oriented object architecture (AOA) [5, 6]. The original concept of this idea was presented in [7]. In this paper, we focus our discussion on how to run C++ applications on a bare PC. The rest of the paper is organized as follows. The Section 2 will provide a full description of our approach, Section 3 describes the benefits of running bare PC applications, Section 4 illustrates some sample applications, Section 5 narrates our current and future research, Section 6 mentions acknowledgements, and Section 7 provides conclusions of this paper.

2. Description

Running C++ applications on a bare machine is not a trivial task. The bare machine used in this work a desktop or a personal computer (PC). You will run into numerous problems and issues in attempting such a daunting task. To simply put this problem in perspective, first you need to avoid all *.h files in your C++ program to run your application with out using an OS. How about loading your program into the machine? You need to build your own boot and load program as there is OS running in the system. The following subsections present some of these problem areas and provide a brief overview on how we accomplished running some applications such as sorting, and email using C++ programming language.

2.1. Boot and Load

In order to get a control of the machine without loading a conventional OS, you need to write your own boot program in assembly or C. This boot program when it is complete will enable you to jump to your own program after the boot. You need to follow standard boot procedures for PCs, preferably from a floppy disk (some PCs today don't have floppy), and power up the machine with this boot disk in the floppy drive. This is simpler to do than booting from a hard disk, as hard disk requires more sophistication in its organization and file system. We have used Turbo assembler to create our own boot code using a floppy disk. There are many samples of boot code available on the Web, but most of them need some modifications and they may not work! Once your system is booted, you need to load your own program from another disk which may be written either in assembly or C. After the boot, you need to jump (long JMP) to this program so that you can accomplish what you want to do with the PC after the boot. In our case, after the boot, we jump to an assembly program which displays a menu where some operations can be performed on the PC such as: loading a program, running a program, checking memory, and so on. You also need a loader program which is part of your boot to load this initial program. This loader has to be small (called a mini-loader) so that it fits in one boot sector code including the boot code. You also need a sophisticated or bigger loader later, if you want to load your large C++ programs in memory.

2.2. Modes of PC

Once the PC is booted, it is in a real mode, where the addressing is limited to 1MB and there is no protection of your code. It is simply in the disk operating system (DOS) mode. In order to load and run larger programs and have access to bigger memory, you need to learn how to switch to protected mode using assembly language instructions. However, all basic input/output system (BIOS) calls are only available in real mode. If you want to use BIOS for I/O, you need to have a mechanisms to switch from protected to real mode to perform BIOS calls. Thus, your C++ program or some sort of control program needs to do this switching that is transparent to the user. If you plan to use software interrupts (255 of them), instead of BIOS interrupts, then you need to write your own code in assembly to address all I/O operations. We have used both BIOS and software interrupts and have written a protected to real mode switch to operate in both modes. This is not a trivial thing to do in assembly which requires you to be familiar with Intel's architecture specification document, which is available on their Web site. This document is a useful resource for understanding and implementing interrupts, addressing schemes, task facilities, traps, and exceptions.

2.3. Input/Output

Instead of switching back and forth to real mode, it is more efficient to write software interrupts for all I/O. There are 255 software interrupts available in a PC environment, where you need to setup this in a table called interrupt descriptor table (IDT). Hardware interrupts are also part of these and they are automatically invoked by the hardware. For example, when a timer interrupt comes, it will be pointed to IDT location 08h. All the software interrupts will be invoked by your program in assembly language. Each IDT entry will be pointing to a global descriptor table (GDT) which keeps the address of the interrupt to be executed. Interrupts can also be used to run applications. Intel architecture specification document provides more detailed description of the architecture facilities. This is a very long learning process for software people as it involves understanding of some hardware facilities, and architecture facilities in the Intel microprocessor.

2.4. Compiling Environments

In a conventional C++ program, we include *.h files, which are OS specific libraries that will be included in the executable. If we use any C++ builders, then they will automatically include these libraries which have no meaning in an OS less environment. Thus, the compiling environment should be using batch files to compile and link the needed modules. As application programmer codes at C++ language level, we have developed an interface file which has all the interfaces needed for these programs. This file has API including: input/output, tasks, memory, timer, shared memory, locks, and so on. We use Visual Studio C++ compiler (batch mode), MASM 6.11 assembler, and Turbo assembler compilers to create executable modules. We have written all batch files to do compilation and linking for boot, initial programs, and for application programs. These batch files are simple and easy to understand and use.

The API is a C++ call, which in turn invokes a C call, and that in turn invokes an assembly call. A C++ programmer only sees the API. If an interface can be accomplished at C++ level, then the code will be implemented in the C++ level itself. We avoid using any direct assembly calls in the C++ program; however, such invocation is permitted by the compiler.

2.5. Memory Map

As you are running applications on a bare machine with out OS, you need to keep track of your own memory so that your program runs properly. Memory map should organize initial program, application program, stack, network data area, shared memory area, task area, and so on. The memory size interface provides the limits of the memory available, which can be used to map memory areas.

2.6. Memory and Disk Usage

We assume that all the memory available in the PC is available to the application. We do not provide any file management other than read and write a sector from a floppy disk. The application program has to manage its own disk space and memory space in the program. As application programmer has full knowledge of PC hardware resources, he/she can allocate and de-allocate them appropriately based on the available storage. Assuming full address-space of memory (for example, 4GB in 32 bit architecture), an AO programmer has to manage this volatile memory during execution, and any persistent storage should be stored on the network (server disk). The AOs could also be stored on the network server and loaded upon demand at a client site.

2.7. Tasks and Scheduling

We developed an innovative task interface to the user. A user can define a C++ class as task object, and define methods inside the class which represent individual tasks. Each task address (or member function address) must be passed to the task object during the creation of a task. The created tasks can be placed in a list or queue and a given scheduling algorithm can be used to run these tasks. We provide a hardware timer to the application programmer to use it for scheduling at a given time intervals. When hardware timer occurs, the IDT will return the control to a timer task in C++ which in turn can manage the scheduler and scheduling of processes in a given period. Due to space limitations in this paper, we could not provide more details of our approach with examples..

2.8. Debugging

Debugging a C++ program is very difficult task in a bare machine environment without any OS support. We have written several tools to help debug the programs. A user can write a trace string to memory in his/her program and later print this string in case of an

error. A user can also invoke an interrupt directly in his/her program, which can be used to print registers, memory, and other flags to identify the source of an error. We provide a facility to dump memory onto screen and also search a string in memory. All the error and exception conditions are handled over to a C++ application program by the hardware by setting up appropriate IDT entry and a procedure in the program. The above debug facilities are quite useful in software development. In addition, a programmer can still write their own trace and debugging messages in their program..

2.9. Ethernet

Ethernet interfaces provide direct communication to network card to check for a message arrival, to read a message, and to write a message. This is an Ethernet frame interface directly from a C++ program, which avoids all other layers including TCP (transmission control protocol), and IP (internet protocol). This allows two AO to communicate at Ethernet level. Some messages such as ARP (address resolution protocol), which are small and within the packet size limit of Ethernet (~1500), they can be directly received by a C++ program. We have written a polling task do exactly that, which polls the Ethernet card and discards all unnecessary messages and responds to valid ARP requests.

2.10. IP

IP interface provides send, and receive facilities. The TCP/UDP (unreliable datagram protocol) layer accesses these interfaces.

2.11. TCP/UDP

TCP interface provides connect, send, receive, and close connection facilities. Similarly, UPD provides send, and receive facilities. These interfaces are accessed by applications.

2.12. ARP

ARP interface provides functions to resolve and process ARP requests. These interfaces can be used by applications or TCP/UDP/IP layers.

2.13. SMTP

The SMTP (simple message transmission protocol) client interfaces provide open-connection, send mail, and close connection for an application. These interfaces are used to build an Email client application. We have also built an email receive client application, which connects to a mail server, and enables you to read messages received for a given user.

3. Benefits

There are many advantages of running a computer application on a bare machine. Some of the advantages can be illustrated as follows.

1. Simplicity: As programs communicate directly to the hardware, there is no middle layer such as OS, thus the program can run on any Intel Pentium based processor. A user can simply put a disk or a CD and run his/her program.

2. Closed System: When you run an application directly on a bare machine, it is a closed system. It does not depend on any other systems or environments. It is harder for an intruder to get into such a closed system, where as in an open system, one could always find a hole in some layer to exploit the system.

3. Less Prone to Errors: All the errors are predictable and containable with in the program. Hardware and software exceptions are handled with in the same program. In a conventional system, OS errors are not visible to application programmers. For example, an application programmer has no knowledge of memory exception until it happens, which may be too late to correct it.

4. No Layers: Application program is one layer. There is no overhead in communicating between layers. There is also no need for intermediate API to communicate from one layer to another.

5. No Environments: There is a single programming language environment. If we choose, C++ as the language of choice for applications, then that is the only environment needed to develop and run applications. There is no need for separate execution environment to run applications.

6. Forces Extensibility: When applications are written to run directly on hardware, software and hardware vendors are forced to provide extension from previous releases, instead of abandoning the previous release and replace it with a new one. When vendors follow extensibility instead of an evolutionary path where extensibility may or may not be provided, products can last longer and less prone to obsolescence.

7. Pervasive: When application programs are directly communicating with hardware, they can be used commonly in many application domains. For

example, if a cell phone and PC has Intel Pentium based processors (may vary in size, and performance), then the same application can run in both domains. Today, an email application is same in a cell phone or in a PC, but the actual software is different. A bare PC and a bare cell phone with the same email application object can offer a homogeneous architecture for pervasive computing [9].

8. Autonomous: When an application program is a closed system, it can be more autonomous or self-controllable and self-manageable.

9. More Secure: When an application is self-contained as there is no other OS or environment, then its security may be better controlled. Secure authentication and privacy mechanisms can be encapsulated right into the application object. As the current systems are complex, open and layered, it is very difficult to enforce security in many component systems and layers. Many of the security problems today point to OS and its internals.

10. More Mobile: When applications are written as self-contained entities and autonomous, then they can be mobile. An application can run on any PC, and it can be anywhere in the network. One does not have to be concerned about the platform issues. Mobility can be achieved physically or through the Internet.

4. Sample Applications

Using the C++ API presented above and some additional API such as string functions, type conversion functions, random number generator, we have developed a variety of application objects including: bubble sort, text editor, send/receive messages using UDP, TCP, Ethernet/within a local network, and send mail using SMTP (Email Client). The I/O devices used in our applications are keyboard, monitor, floppy drive, and Pentium processor with 64MB of memory. The Email client has 2666 lines of code without including comments, and the size of the executable is 32,768 bytes. Notice that the AO is very small and it does not require any other environment or libraries. We did not implement fragmentation in TCP/IP as most of the email messages are small.

Developing bare PC applications was a daunting task for us as it requires in depth knowledge and experience with PC hardware and architecture. Most of all, there are no debugging tools available that can run on a bare PC. Thus, we first started our mission with a development of "hello" program and slowly progressed to Email Client. Now, we have a comprehensive C++ API, which can be used to develop complex applications such as a Web server, which is currently in progress.

The construction of Email client was also a difficult problem due to the network layers and communication between an OS less PC and a conventional mail server. Initially, we developed communication between two PCs where both of them are bare machines. Our first step and hurdle was development of a device driver for a network card running Ethernet. Ethernet device drivers are specific to a particular OS. We developed an Ethernet driver for 3Com 509B card that has basic functions to send and receive messages in synchronous manner. Finally, we have developed the full-blown functions to write the Email Client AO.

As the TCP/IP communication protocols are robust and implementation independent, we are able to build a lean and mean network stack that is needed for the email client. Due to space limitations, the details of these protocols are not described here.

When an application is built, it includes all the necessary program and data needed for its execution. For example, sorting application consists of random number generator, data in a file, keyboard input and output display. The size of this code module is 16 KB. Similarly, an Email application consists of all communication protocols, device driver, and all C++ API. Notice that, most of these applications are small in size and easy to manage them on a floppy drive.

5. Current and Future Research

Currently, we are developing a Web server that can run on a bare machine. This presents challenging problems and issues related to our networking code, which should be extended to run Web server application. Most of the changes are needed in TCP/IP and a new protocol HTTP (hypertext transmit protocol). We are also investigating a computer architecture that is amicable to application objects and bare machine environment. Once the Web server is built, we will be able to conduct performance benchmarks on the bare PC server and compare it with other commercially available servers such as Apache. We are also seeking to develop new application objects that make our approach suitable for other application domains. Finally, we are investigating securing aspects of the application objects, bare machine computing, and their role in commercial computing.

6. Acknowledgements

We sincerely thank NSF and in particular late Dr. Frank Anger who initially supported this work by funding SGER CCR-0120155. With out this support, our work could not have reached to this level.

7. Conclusions

We described detailed knowledge that is required to build bare PC applications using C++ programming language. The C++ API illustrated consists of many interfaces including: boot-up, loading, timer, memory, keyboard, display, disk, tasks, TCP/IP stack, SMTP, and others. We have presented the benefits of running computer applications on a bare PC. Current state of research and further work needed is also briefly outlined. When the Web server under construction is operational, this may be the first Web server on the Internet as per our knowledge, which will be running on a bare machine with no conventional OS. When this concept becomes successful, it will spur research in pervasive computing devices that can use our approach to build homogeneous devices that run on bare machines. Semiconductor vendors such as Intel and others should be interested in this idea as they can now build microprocessors that are common to a variety of pervasive devices.

8. References

[1] "A complete 32-bit C/C++ development system for Intel 80386 (and higher) PCs running DOS," http://www.delorie.com/djgpp.

[2] Engler, D. R., The Exokernel Operating System Architecuture, Ph.D. thesis, MIT, October 1998.

[3] Ford, B., Hibler, M., Lepreau, J., McGrath, R., and Tullman, P., "Interface and execution models in the Fluke Kernel," Proceedings of the Third Symposium on Operating Systems Design and Implementation," 1999, pp. 101-115.

[4] Kaashoek, M.F., Engler, D.R., Ganger, G.R., and Wallach, D.A. Server Operating Systems, In the Proceedings of the 7th ACM SIGOPS European Workshop: Systems support for worldwide applications, Connemara, Ireland, September 1996, pages 141-148.

[5] Karne, R.K., Gattu, R., Dandu, R., and Zhang, Z., "Application-oriented Object Architecture: Concepts and Approach," 26th Annual International Computer Software and Applications Conference, IASTED International Conference, Tsukuba, Japan, NPDPA 2002, October 2002.

[6] Karne, R.K. Application-oriented Object Architecture: A Revolutionary Approach, 6th International Conference, HPC Asia 2002, December 2002.

[7] Karne, R.K., "Object-oriented Computer Architectures for New Generation of Applications," Computer Architecture News, December 1995, Vol. 23, No. 5, pp. 8-19.

[8] Lampoudi, S., and Beazley. D.M. SWILL: A Simple Embedded Web Server Library, Proceedings of the FREENIX Track: 2002 USENIX Annual Technical Conference, Monterey, California, June 10-15, 2002.

[9] Oxygen Project, MIT, http://oxygen.lcs.mit.edu.

[10]Qi, X., Parmer, G., West, R. An Efficient End-host Architecture for Cluster Communication Services, to appear in Cluster 04.

[11] "The OS Kit Project," http://www.cs.utah.edu/flux/oskit.

[12] The Choices: Object-Oriented Operating System, http://choices.uiuc.edu/choices/choices.html.

[13] Turn any IBM compatible PC into a Preliminary Target Board, High Quality, Real-Time, Embedded Ada for PCs, http://www.ddci.com/products_pcbare.shtml.

[14] Tan, See-Mong., Raila, D.K., and Campbell, R.H. An Object-oriented nano-kernel for operating system hardware support, Object-orientation in Operating Systems, 1995, Fourth International Workshop, p220-223.