Paper: ASAT-14-258-CE

14th International Conference on
AEROSPACE SCIENCES & AVIATION TECHNOLOGY,
ASAT - 14 – May 24 - 26, 2011, Email: asat@mtc.edu.eg
Military Technical College, Kobry Elkobbah, Cairo, Egypt
Tel: +(202) 24025292 –24036138, Fax: +(202) 22621908

Application Migration from Ultrix to Linux Operating System

M. Sobhy*, A.M. Salem†, K. Abdel-Salam*

Abstract: During the recent twenty years there has been substantial investments aimed to migrate legacy application hosted on legacy operating systems to modern applications to be host on modern operating systems, This process can be expensive, both in terms of actual cost and the risk, the migration will lead to changing core applications and can significantly change the way the entire organization operates. Legacy systems are successful and therefore mature, and likely have been in existence for a long period of time. A consequence is that legacy software is built using technologies available at the time it was constructed, as opposed to the most modern software technologies. Older technologies are more difficult to maintain, and this is a key point of pain for many legacy system owners. The aim of this paper is to get a generic methodology for migrating application hosted on legacy UNIX (Ultrix) or its derivative to be hosted on modern Linux operating system. We have already got a legacy application hosted on Ultrix operating system as a case study and we will perform a successful migration to be host on Linux operating system through our proposed migration methodology.

Simply the idea of our proposed method lies in getting the architectural and functional differences between Ultrix and Linux in order to get the best method for migration. The assumption is that RedHat and other Linux variants, being similar to Ultrix isn't strictly true. After we had understood the functional differences between Ultrix and Linux, we had mapped system calls to a correct corresponding system calls in new modern platform in order to maintain the same behavior of application and overcoming the problem of endian. Moreover, there are differences in compiler options, building options, the synchronization methods used and signals. Furthermore, we got a methodology to migrate legacy Fortran code and Ingres database to modern ones.

This paper organize into five sections, section one, introduction and overview on the Ultrix and Linux history, section two, challenges facing legacy application and why most organizations nowadays concerning with migration, section three, our proposed migration methodology and how overcoming the issues mentioned above in abstract, section four, summary and conclusion, and section five, references.

1. Introduction

For computer science at Bell Laboratories, the period 1968-1969 was somewhat unsettled. The main reason for this was the slow, though clearly inevitable, withdrawal of the Labs from the Multics project. To the Labs computing community as a whole, the problem was the increasing obviousness of the failure of Multics to deliver promptly any sort of usable system, let alone the panacea envisioned earlier. For much of this time, the Murray Hill Computer Center was also running a costly GE 645 machine that inadequately simulated the GE 635. Another shake-up that occurred during this period was the organizational separation of

* Egyptian Armed Forces, Egypt.
† Ain Shams University, Cairo, Egypt.

computing services and computing research [1]. In 1969, Bell Laboratories developed UNIX as a "timesharing" system, a term used to describe a multitasking operating system that supports multiple users at each terminal. Although the first implementation was written in assembly language, the designers always intended to write UNIX in a higher-level language. Therefore, Bell Labs invented the C language so that they could rewrite UNIX. UNIX has evolved into a popular operating system that runs on computers ranging in size from personal computers to mainframes [2]. Figure 1 depicts the evolution of UNIX from a single code base into the wide variety of UNIX systems available today. In fact, this is only a summary—there are more than 50 flavors of UNIX in use today. The codes in the diagram refer to the brands and versions of UNIX that are in common use, including: AIX from IBM, Solaris from SUN Microsystems, HP-UX and Tru64 from Hewlett Packard, UnixWare from Caldera, FreeBSD (open source).

The first native VAX UNIX product from DEC was Ultrix-32, based on 4.2BSD with some non-kernel features from System V, and was released in June 1984. Ultrix-32 was primarily the brainchild of Armando Stettner. Its purpose was to provide a DEC-supported native UNIX for VAX. It also incorporated several modifications and scripts. Later, Ultrix-32 incorporated support for DECnet and other proprietary DEC protocols such as LAT. It did not support VAX clustering, last major release of Ultrix was version 4.5 in 1995.

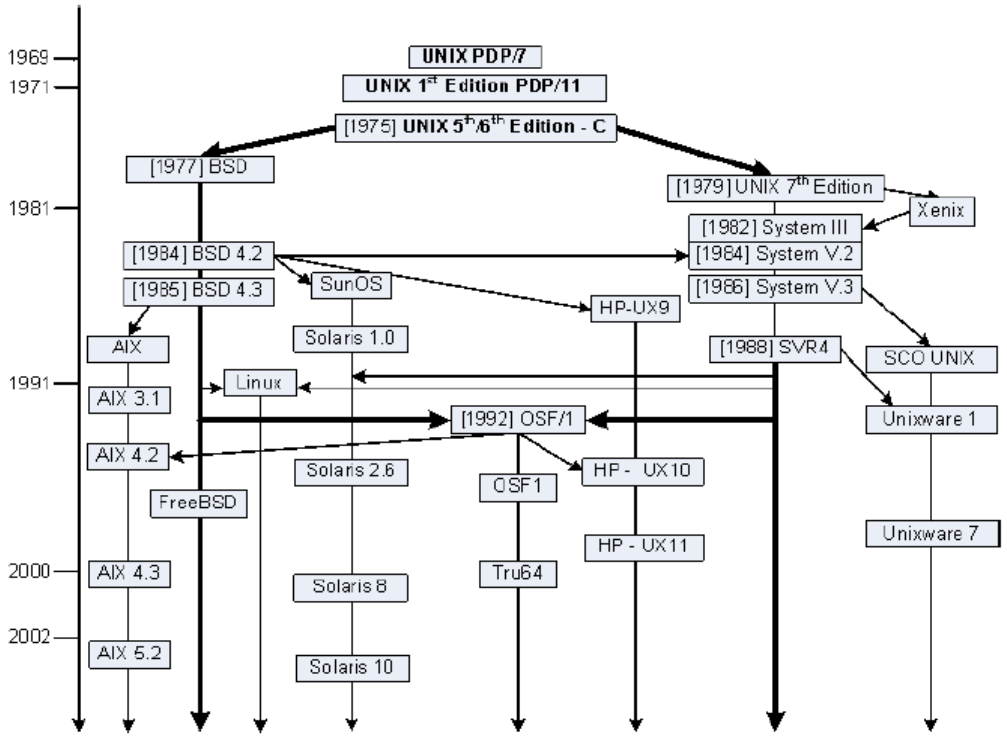

Figure 1 Evolution of the UNIX operating system [1]

The History of Linux began in 1991 with the commencement of a personal project by a Finnish student, Linus Torvalds, to create a new operating system kernel. Since then the resulting Linux kernel has been marked by constant growth throughout its history. The UNIX operating system was conceived and implemented in the 1960s and first released in 1970. Its availability and portability caused it to be widely adopted, copied and modified by academic institutions and businesses. Its design became influential to authors of other systems. In 1983, Richard Stallman started the GNU project with the goal of creating a free UNIX-like operating system [2]. As part of this work, he wrote the GNU General Public License (GPL).

By the early 1990s there was almost enough available software to create a full operating system. However, the GNU kernel, called Hurd, failed to attract enough attention from developers leaving GNU incomplete. MINIX, a Unix-like system intended for academic use, was released by Andrew S. Tanenbaum in 1987. While source code for the system was available, modification and redistribution were restricted. In addition, MINIX's 16-bit design was not well adapted to the 32-bit features of the increasingly cheap and popular Intel 386 architecture for personal computers. These factors and the lack of a widely-adopted, free kernel provided the impetus for Torvalds's starting his project. He has stated that if either the GNU or 386BSD kernels were available at the time, he likely would not have written his own, In 1991, in Helsinki, Linus Torvalds began a project that later became the Linux kernel. It was initially a terminal emulator, which Torvalds used to access the large UNIX servers of the university. He wrote the program specifically for the hardware he was using and independent of an operating system because he wanted to use the functions of his new PC with an 80386 processor. Development was done on MINIX using the GNU C compiler, which is still the main choice for compiling Linux today (although the code can be built with other compilers, such as the Intel C Compiler).

2. Issues and Challenges Facing Legacy Systems

Maintaining and upgrading legacy systems is one of the most difficult challenges CIOs face today. Constant technological change often weakens the business value of legacy systems, which have been developed over the years through huge investments. CIOs struggle with the problem of modernizing these systems while keeping their functionality intact. Despite their obsolescence, legacy systems continue to provide a competitive advantage through supporting unique business processes and containing invaluable knowledge and historical data.

Despite the availability of more cost-effective technology, about 80% of IT systems are running on legacy platforms. International Data Corporation estimates that 200 billion lines of legacy code are still in use today on more than 10,000 large mainframe sites. The difficulty in accessing legacy applications is reflected in a December 2001 study by Hurwitz Group that found only 10% of enterprises have fully integrated their most mission-critical business processes [3] .

Driving the need for change is the cost versus the business value of legacy systems, which according to some industry polls represent as much as 85-90% of an IT budget for operation and maintenance. Monolithic legacy architectures are antitheses to modern distributed and layered architectures. In addition, IT departments find it increasingly difficult to hire developers qualified to work on applications written in languages no longer found in modern technologies.

Several options exist for modernizing legacy systems, defined as any monolithic information system that's too difficult and expensive to modify to meet new and constantly changing business requirements. Techniques range from quick fixes such as screen scraping and legacy wrapping to permanent, but more complex, solutions such as automated migration or replacing the system with a packaged product.

In our paper, the problem arises because of we have an application which is running on DEC 5500 machine that has Ultrix operating system hosted on it, and we want to migrate this application to be run on X86 processors which host Linux operating system.

We are interested in this migration because of the underlying platform (Ultrix) is hard to support and running on legacy hardware systems (DEC 5500). Such hardware systems are becoming more expensive to maintain or obsolete, and personnel that know these systems are also more difficult to find. These legacy applications are self-contained and have little or no flexibility to adapt them to changing business requirements. However, some of our current

business models were not even conceived of at the time these applications were developed. In general, legacy application maintenance is very cumbersome and represents a major fixed cost at a time of reduced IT spending. The high cost of maintenance is also compounded by the fact that many of the original developers are reaching retirement age and there is a critical shortage of skilled resources to replace them.

3. Proposed Methodology

Today's information system executives are primarily concerned with ways to increase bottom line results quickly with cost cutting programs that additionally improve IT service levels on a global basis. One of such major undertakings is "updating legacy systems", based on the beliefs that existing systems are valuable assets to be improved, especially at a time when, for many companies, the cost of implementing new systems is unacceptably high. IT and financial executives alike are losing sleep over the contradictory challenge of significantly cutting cost of IT operations and at the same time increasing operational effectiveness and service quality to the organization.

In light of these and other factors, we have many choices:

Replace legacy applications with packaged software; Rewrite the applications using new technology and tools and Application migration.

We believe the most important aspect of application migration is the preservation of existing organization business processes and practices and minimal business disruption. There is a much lower risk with application migration than with rewriting projects.

Our approach is predicated on application migration, focusing on the application code, touch points with the old infrastructure and building to an alternate environment such as Linux. This preserves the integrity of the application with no risk to the embedded business knowledge while integrating the new platform capabilities.

We introduce our proposed migration methodology, which is simply based on studying the behavior of legacy application in order to get the critical parts in this application to be migrating which often be system calls, big-little endian, semaphores, compiler options and building options. Also database and obsolete programming languages migration, in our proposed migration we get a generic methodology in order to migrate any Ultrix application to be run on Linux platform by overcome all above critical issues mentioned above.

3.1 Endianness

In computing, endianness is the ordering of individually addressable sub-units (words, bytes, or even bits) within a longer data word stored in external memory. The most typical cases are the ordering of bytes within a 16-, 32-, or 64-bit word, where endianness is often simply referred to as byte order [5]. The usual contrast is between most versus least significant byte first, called big-endian and little-endian respectively. Mixed forms are also possible; the ordering of bytes within a 16-bit word may be different from the ordering of 16-bit words within a 32-bit word, for instance; although rare, such cases are sometimes collectively referred to as mixed-endian or middle-endian.

Although Linux was developed on an Intel platform, which is primarily a little-endian environment, it has been ported to several other hardware platforms that support big-endian environments. Endianness, or byte ordering, refers to how a data element and its individual bytes are stored. In a big-endian (BE) environment, the lowest address is associated with the most-significant or leftmost byte of a multibyte value. In a little-endian (LE) environment, the lowest address is associated with the least-significant or rightmost byte of a multibyte value. In general, bit 0 is associated with the most-significant bit (MSB) for BE, but with the least-

significant bit for LE. List 1 shows a sample program that prints data differently when compiled and run on a BE or LE environment.

List 1 Code Listing of endian_sample.c

```
1   #include <stdio.h>
2    main( ) {
3    int i;  /* Loop variable */
4    long x = 0xAABBCCDD;
5    unsigned char *ptr = (char *) &x;
6    printf("x in hex: %x\n", x);
7    printf("x by bytes: ");
8    for (i=0; i < sizeof(long); i++)
9   printf("%x\t", ptr[i]);
10   printf("\n");}
```

When compiled and run on an Intel server, which is an LE environment, the following is printed:

```
x in hex: aabbccdd
x by bytes: dd cc   bb   aa
```

When compiled and run on an IBM Power server, which is a BE environment, the following is printed:

```
x in hex: aabbccdd
x by bytes: aa bb   cc   dd
```

Note that the preceding example was compiled with gcc, which creates a 32-bit executable by default.

Most RISC-based computers, including IBM PowerPC servers, and the Internet Protocol (IP) use a BE layout, whereas Intel and Alpha architectures utilize an LE layout. When porting software, it is recommended to watch out for endian issues, because most if not all of these issues may go undetected, resulting in hard-to-find problems when they occur.

Nonuniform data reference occurs more often in user space application code, whereas the latter two categories are difficulties encountered at lower code levels (for example, device drivers). Nonuniform data reference arises from improper data type reference with regard to endianness, usually dealing with unions or pointers. Endian-friendly code should incorporate definitions to determine whether the platform is LE or BE. It is considered a good programming habit to never cast a pointer to an int and to explicitly reference data type and byte values during conversion.

Software professionals porting applications from UNIX or other platforms to Linux on POWER must remember the following:

Linux on POWER is a big-endian platform

In order to overcome this problem we implement code that converts any data type from BE to LE and vice versa by define variable (rev) which has value 0 or 1 depend on converting from BE to LE or vice versa, List 2 shows a sample of this code

After implementing the above code and testing it, then we highlights all critical sections in the code to be migrated that will suffer from BE-to-LE problem and then we passes these section to the code in List 2 in order to convert all messages server send or receive it, List 3 shows a sample of these critical sections.

List 2 Code Listing of convert_BE_LE.c

```
#define rev 1
void rev_arr_ (char* cc,int len)
{ int i=0;
        for(i=0;i<len/2;i++)
        {         char temp=cc[i];
                cc[i]=cc[len-1-i];
                cc[len-1-i]=temp;    }}
void rev_data_array(char* x,int data_type_len,int total_len)
{    int i=0;
        if(rev)
        for(i=0;i<total_len*data_type_len;i+=data_type_len)
                        rev_data_type(x+i,data_type_len);}
```

List 3 Code Listing of convert_messages_BE_LE.c

```
#include <include/pc.h>
#include "pc_extern.h"
void invers_astt_msg_X_BUFFER (X_BUFFER* x)
{
        rev_data_type_(&x->sw,sizeof(x->sw));
        rev_data_type_(&x->bs[0].size,sizeof(x->bs[0].size));
        rev_data_type_(&x->bs[1].size,sizeof(x->bs[1].size));
}
```

3.2 Semaphores in Ultrix and Linux

There are two implementations of semaphores, traditional SystemV semaphores and the newer POSIX semaphores. For the traditional one you need <sys/sem.h> like in Ultrix and for the new POSIX one <semaphore.h> like in Linux. The functions they supply can be distinguished easily: all the functions for SysV semaphores have no underscore in their names (so it's actually not sem_get() but semget()) while the POSIX semaphore functions all have one.

Table 1 shows some functions they supply:

Table 1 SysV and POSIX comparison

SysV	POSIX
semctl()	sem_getvalue()
semget()	sem_post()
semop()	sem_timedwait()
	sem_trywait()
	sem_wait()
	sem_destroy()
	sem_init()
	sem_close()

In our application to be migrated there are four types of mode, one of them called sync playback mode the system do not wait if in sync playback mode the implementation of this condition appear in file called sm_shell.c in SM module like this :
if (cd->simulation_mode != sync_playback) Wait_Alarm();

The problem appears because of the implementation of (Wait_Alarm function) as shown in list 4.

List 4 Implementation of Wait_Alarm function

```
void Wait_Alarm()
{ int smask;
 /* use sigblock to obtain the current masks 'smask' */
 smask = sigblock(1<<(SIGALRM));
 /* wait for the alarm signal to sequence the next loop */
 sigpause(smask);}
```

In order to overcome the wait_Aralm problem we replace it by using signal as shown in List 5:

List 5 Implementation of Wait_Alram in Liunx

```
If (cd->simulation_mode != sync_playback)
                        {       signal(SIGALRM,Alarmclock);
                                sigpause(0L);
                        }
```

Another difference is that with POSIX semaphores you can have named semaphores, i.e. semaphores that have a file associated with them, with the last three functions used for creating, closing and deleting such a named semaphore while sem_init() and sem_destroy() are only to be used with unnamed semaphores. We don't know exactly when POSIX semaphores were introduced in Linux but they weren't available in 2.2 kernels. They are also not mandated by POSIX but only as a Real-time extension, so they need not be available on all systems that claim POSIX conformance.

3.3 System Calls and Compiler/Building Options

The simplest definition of system calls is that the mechanism used by an application program to request service from the operating system. System calls often use a special machine code instruction which causes the processor to change mode (e.g. to "supervisor mode" or "protected mode"). This allows the operating system to perform restricted actions such as accessing hardware devices or the memory management unit. The only way for an application in User Space to explicitly trigger a switch to Kernel Mode is to issue a *system call*. Therefore system calls constitute the interface between processes and the operating system. Each system call provides a basic operation such as opening a file, getting the current time, creating a new process, or reading a character. In this way system calls can be viewed as regular function calls, if it were for the fact that they transfer control to the Ultrix Kernel. System calls essentially are synchronous calls to the operating system.
When a program invokes a system call, it is interrupted [6] and the system switches to Kernel space as shown in Figure 3. We must take the behavior of fork in our consideration when we migrate to from Unix, Ultrix and its earlier version to Linux, The behavior of fork() in Unix,

Ultrix and earlier releases is different from fork() in POSIX threads. In POSIX threads, fork() creates a new process, duplicating the complete address space in the child. However, it duplicates only the calling thread in the child process. The Unix threads API also provides the replicate all fork semantics, forkall(). This function duplicates the address space and all the threads in the child. This feature is not supported by the POSIX thread standard. System calls differ from operating system to another also from hardware to another, Table 2 showing sample of system calls on Unix and corresponding in Linux.

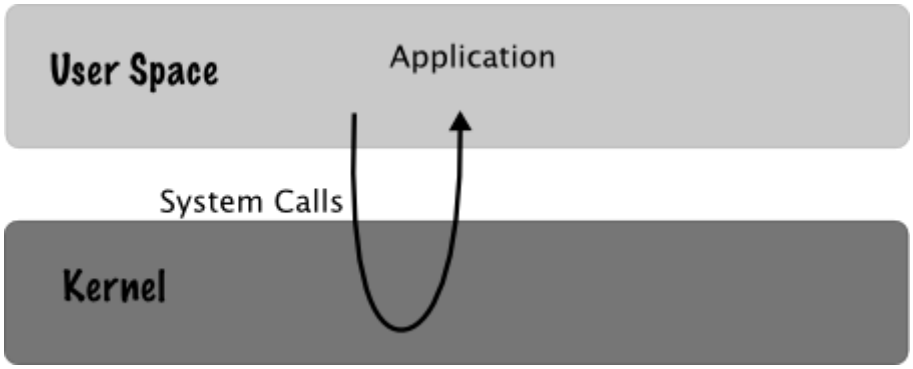

Figure 3 Switching between User space and Kernel

Table 2 System Calls Sample Comparison

Ultrix Library Calls	Linux Equivalent	Description
perror, errno, sys_errlist, sys_nerr	perror, errno	System error messages.
l64a_r	None	Convert between long integer and base-64 ASCII string.
clock_getres	clock_gettime	Clock operations.
fdatasync	Fsync	Synchronize a file's in-core state with its state on disk.
getprivgrp, setprivgrp	None	Get and set special attributes for group.
thr_create()	pthread_create()	Creates a new thread of control.
thr_exit()	pthread_exit()	Terminates the execution of the calling thread.

3.4 Compiler/Building Options

As the porting effort starts, differences in compiler locations and compiler options between Ultrix and Linux become apparent. Although the compilers on both Ultrix and Linux claim conformance or close conformance to ANSI and ISO standards, porting personnel may encounter platform-specific programming syntax differences. The use of compiler extensions and platform-specific features are common programming practices that require some effort when porting from Ultrix to Linux.

Table 3 Ultrix and Linux Compiler Comparison Table

Compiler	Ultrix Path	Linux Path
ANSI C	/opt/ansic/bin/cc	/usr/bin/gcc
C89	/opt/ansic/bin/c89	/usr/bin/gcc -std=c89
C++	/opt/aCC/bin/aCC	/usr/bin/g++

```
One of the famous errors when you rebuilding an application written to be
run on Ultrix system on Linux system are for example:
$ nasm -f elf -o example2.o example2.asm
$ ld -s -o example2 example2.o -lX11 -L/usr/X11R6/lib
$ ./example2
 bash: ./example: /usr/lib/libc.so.1: bad ELF interpreter: No such file or
directory
```

In order to run our case study application, we have first to compile eight folders, which contain the main modules after compiling each folder individual merging object files (.o file) in each folder in one library file (.a file) using the suitable makefile, and then link all these library files together with suitable dynamic linker to get executable file to be run. All these processes wrote in a build file, List 6 show sample of the contents of this file. In legacy system we type ./build_file_name on shell to run, we try this on Linux but the above bolded error appear to us too.

List 6 Build file in Legacy system

```
#Build procedure for host software
cd /usr/users1 /source/md
make
.
.
echo - SC build complete
cd ../sm
make
echo - SM build complete
link_file
echo
echo *** build complete***
```

From the above list its clear that after building all files in the eight main folder, the build file call another file called link file which is reasonable for dynamic linking at this step, problem arise here again because we are now in front of different compilers and different dynamic linker List 7 show the old link file and the bolded line show the reasons of this problem.

On Ultrix, /usr/lib/libc.so.1 is an alternate interpreter for SVID ABI compatibility (SVID == System V Interface Definition). I suppose that the same is possible for Linux. This suggests to us that ld is incorrectly compiled or configured. It should know the proper dynamic linker without being told. For some reason it seems to think that libc.so.1 is the correct dynamic linker, which it apparently is not. gcc seems to always specify the dynamic linker, so this error, if it is an error, would only show up if one were invoking ld directly and using dynamic linking, which is not often done.

List 8 shows how we can overcome linking problem, then it works just fine with no complaints, no errors so, obviously, GCC is automatically setting some parameter or some "default" differently to LD...we could, of course, just continue to use GCC and everything would be fine.

List 7 The old link file on Ultrix machine

```
echo Linking AA program...
cp /usr/users1 /execute  /usr/users1/ /execute/ _exe.old
ld /lib/crt0.o -o aa\
$AA_sc/si.o\
$AA_sc/sg.o\
$II_SYSTEM/sql/lib/libsql.a\
-L$AA_sm -lsm\
-L$AA_md -lmd\
-L$AA_dy -ldy\
-L$AA_gr -lgr\
-L$AA_rc -lrc\
-L$AA_pc -lpc\
-lm\
$AA_src/util/memory/libmem.a\
-lUfor -lfor -lutil -li -lots -nocount -lc
echo Load complete - Type aa to run.
```

List 8 The new link file on Linux machine

```
echo Linking AA program...
cp /usr/users1 /execute  /usr/users1/ /execute/ _exe.old
gcc -o aa\
$AA_sc/si.o\
$AA_sc/sg.o\
/opt/Ingres/IngresII/ingres/lib/libingres.a\
/usr/lib/gcc-lib/i386-redhat-linux/3.2.3/libgcc.a\
-L$AA_sm -lsm\
-L$AA_md -lmd\
-L$AA_dy -ldy\
-L$AA_gr -lgr\
-L$AA_rc -lrc\
-L$AA_pc -lpc\
-lm\
$AA_src/util/memory/libmem.a\
-ldl -lpthread -lcrypt -lrt -lc
echo Load complete - Type astt to run.
```

3.5 Database Migration

Break a dependency chain before it breaks you and the migration process. Given the scenario of migrating from ingres 6.4 to ingres 9.3, changing the underlying operating system to Linux from Ultrix, modifying major tables within a schema, and running newer/modified versions of related applications. Since databases are more difficult to migrate than a set of files, the migration process and resource allocation have to be carefully planned and customized to the specific environment to minimize production downtime.

There are various types of database migration such as:

Database Upgrade : Installing the latest database release

Hardware Upgrade : Moving to a newer hardware / software release on the same platform

New Platform : Moving to the different hardware / software platform
OS Upgrade : Upgrading OS release on the current system
All these types of data migration are independent of each other, but some are often combined together to make it more efficient and to take advantage of the overall process. One of the most common types of migration is the database upgrade to the latest release. This is usually done in conjunction with hardware upgrade and / or OS upgrade such as in our case we will perform Database upgrade from ingres 6.4 to ingres 9.3, Hardware upgrade from DEC 5500 to Intel x86 and new platform from Ultrix to Linux. Any of these data migration paths involve installation of the new database server on the target system. Usually, it is the latest database release. In order to migrate database, we have first to migrate database engine in our case we have to migrate database ingres 6.4 which running on Ultrix to ingres 9.3 which run on Linux, this step seems to be trivial but actually there are so many important issues to be in consideration in this phase. We have to change some paths of system environment, for example after installation of ingres on Linux and trying to perform any database action like createdb, this error will appear
createdb: could not connect to database template1 : could not connect to server : No such file or directory
Is the server running locally and accepting connections on Unix domain socket
"tmp/.s.PGSQL.5432
This error arise because there is something called the postgresql createdb and not the ingres createdb which is Postgres is another DBMS shipped with Linux it shares many of the same utility names as Ingres and default in Linux not like Unix, to overcome this error we have to set environment variable to be Instead of this:
export PATH=$PATH:$II_SYSTEM/ingres/bin:$II_SYSTEM/ingres/utility;
you need to have this:
export PATH=$II_SYSTEM/ingres/bin:$II_SYSTEM/ingres/utility:$PATH;
Which will put the Ingres directories ahead of Postgres, and so you will get the Ingres versions of those utilities. In our application which is client/server application client is hosted on Windows platform and server hosted on Linux, there are must be connection between database hosted on Linux server and Windows client. The client application is written in C#, so we have to provide a connection between .NET and database, one of the most errors appear while connecting is invalid user name and password even it's correct. The cause of this problem was incorrect permissions (connection string) on file 'ingvalidpw'.
The installation is leaving ingvalidpw as:
-rwxr-xr-x 1 ingres ingres 11665 2009-02-25 18:31 ingvalidpw
So, to fix the problem, as root:
chown root ingvalidpw
chmod u+s ingvalidpw
which results in:
-rwsr-xr-x 1 root ingres 11665 2009-02-25 18:31 ingvalidpw

At first we export all dump files from legacy database (ingres 6.4) to import it again in modern database (ingres 9.3), after we got about 18 dump files (.hp) of all necessary data, we created SQL file to import these files into modern database, List 9 and List 4-10 Show sample of SQL file that export and import data from and in the system.

List 9 Sample of SQL file for exporting data

```
copy breakpoint_details(
        breakpoint_id= varchar(0)tab with null(']^NULL^['),
        breakpoint_num= c0tab with null(']^NULL^['),
        directives= varchar(0)tab with null(']^NULL^['),
        parameters= varchar(0)nl with null(']^NULL^['),
        nl= d0nl)
into 'C:/breakpoint_details.hp';
copy breakpoint_directives(
        directive= varchar(0)nl with null(']^NULL^['),
        nl= d0nl)
into 'C:/breakpoint_directives.hp';
copy breakpoint_names(
        breakpoint_id= varchar(0)tab with null(']^NULL^['),
        breakpoint_num= c0nl with null(']^NULL^['),
        nl= d0nl)
into 'C:/breakpoint_names.hp';
```

List 10 Sample of SQL file for importing data

```
create table breakpoint_details(
        breakpoint_id char(16),
        breakpoint_num integer,
        directives char(4),
        parameters char(10)
)
with duplicates,
location = (ii_database);
copy breakpoint_details(
        breakpoint_id= varchar(0)tab with null(']^NULL^['),
        breakpoint_num= c0tab with null(']^NULL^['),
        directives= varchar(0)tab with null(']^NULL^['),
        parameters= varchar(0)nl with null(']^NULL^['),
        nl= d0nl)
from '/migration/database creation/breakpoint_details.hp';
create table breakpoint_directives(
        directive char(4)
)
with duplicates,
location = (ii_database);
from '/migration/database creation/breakpoint_names.hp';
grant   select   on breakpoint_details to public;
grant   update   on breakpoint_details to public;
grant   delete   on breakpoint_details to public;
grant   insert   on breakpoint_details to public;
commit;
```

3.6 Migration of FORTRAN 77 to FORTRAN 95

With the release of the Sun ONE Studio 7 Compiler Collection, the Fortran 95 compiler, f95, now accepts many of the features of the FORTRAN 77 compiler, f77, including various language extensions. The following non-standard FORTRAN 77 extensions were supported by the **f77** compiler but are not supported by **f95**:

f95 limits the number of continuation lines to 99. f77 had no such limit on continuation lines.
Variable Format Expressions (VFEs) are not available in f95.
f95 does not recognize the legacy f77 "R" format edit descriptor.
Arrays and character strings with variable lengths are not allowed on Fortran 95 NAMELIST statements.
f95 reports illegal I/O specifiers as errors. f77 gave only warnings.
f77 allowed up to 20 array subscripts; f95 allows only 9.
f95 does not allow non-constants in PARAMETER statements.
Integer values cannot be used in the initializer of a CHARACTER type declaration.
f95 will not allow array elements in boundary expressions before the array is declared. List 13 showing the following gets an error:

List 11 Error from incompatibility

```
subroutine s(i1,i2)
integer i1(i2(1):10)
dimension i2(10)
...->ERROR: "I2" has been used as a function, therefore it must not be declared with the explicit-shape DIMENSION
```

The maximum length for names is 31 characters.
Debugging comments (comments lines with "D" in column one) are always treated as comments. There is no option for turning them into live statements.
f95 does not recognize the following f77 compiler options:

- `-arg=local -dbl -oldstruct -i2 -i4 -r4 -r8 -vax`

Library Routines Not Supported by f95:

- `The POSIX library.`
- `The IOINIT ( ) library routine.`

When moving source code from one platform to another, you need to first look at the general structure and location of the various files that need to be moved. Ultrix applications typically go in the /usr directory. Many Ultrix administrators further segregate system programs from added user programs. Commonly, user programs are added to /usr/local, and system programs go in the /usr root.
Developers ordinarily find source code files in /usr/local on the Ultrix platform. Within this directory, common directories include:
• /usr/local/bin…….. for executable files.
• /usr/local/include…….. for header files.
• /usr/local/lib…….. for library files.

4. Summary and Conclusions

Migration is transactions that transfer an application that already run on a specific platform to another and getting minimum change in code, instead of redeveloping the code again to suit

the new platform. The result of this continuing progress is that you as the IT decision maker are caught in a difficult situation.
You can make no changes and risk that your systems will slip into obsolescence.
Or you can make a change and risk joining a computing trend that turns out to be an evolutionary dead end (Migration).
So what reasons are driving us to consider moving from UNIX environment that has served us so faithfully all these years? Perhaps because we have the following goals:
Reducing costs
Increasing flexibility
Improving performance
We study many techniques for application migration but we focused on the major differences between legacy and modern platform in order to get successful proposed migration methodology.
As we mentioned earlier that some techniques like hardware emulator [7] and software simulator [8] cannot deal or get fully successful migration with large applications. In our paper, we introduced a large application as case study and applied our proposed method and got fully successful migration.
We conclude that the little-big endian, system calls, semaphores and signals are more promising in our problem especially that you can get technical successful migration but fail in getting the same behavior of legacy application. Furthermore, we got methods to migrate database and legacy Fortran. Indeed, since we got a better result by using our proposed methodology. But, every application has its own features so we have to get good assessment in order to get successful migration. A future direction would be in getting encapsulating our proposed method in a middle ware which will be run intermediate between modern operating system and legacy application.
Finally there is no comparison between the performance of application before and after migration, that's because there is huge gap between Ultrix and Linux performance, moreover, Intel X86 and DEC machines.

5. References

[1] Andrew S. Tanenbaum and Albert S Woodhull, Operating Systems Design and Implementation", Prentice Hall, 3rd edition, 2006.
[2] W. Richard Stevens and Stephen A. Rago, "Advanced Programming in the UNIX Environment", Addison Wesley Professional, 2nd edition, 2005.
[3] Bruce Claremont, "Understanding the business aspects of software migrations", July 1992
[4] "Linux System Calls" by Mark Mitchell and Jeffrey Oldham for a good description of common system calls.
[5] Jonathan Lewis, "How UNIX Works", Mc Graw Hill, 2005
[6] M. Tim Jones, "Kernel command using Linux system calls", IBM developerWorkers, March 2007
[7] Microsoft Corporation, "UNIX Custom Application Migration Guide", 2006.
[8] David Schafer, "Making a Choice between an Emulator or Software Migration", IEEE AUTOTESTCON Proceedings, San Antonio, TX, USA, 2004.
[9] Moeini, A. Rafe, V. Mahdian, F., "An Approach to Refactoring Legacy Systems", 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE), 2010.