Operating Systems (coe628) Lecture Notes – Week 1

Table of Contents

1 Preamble

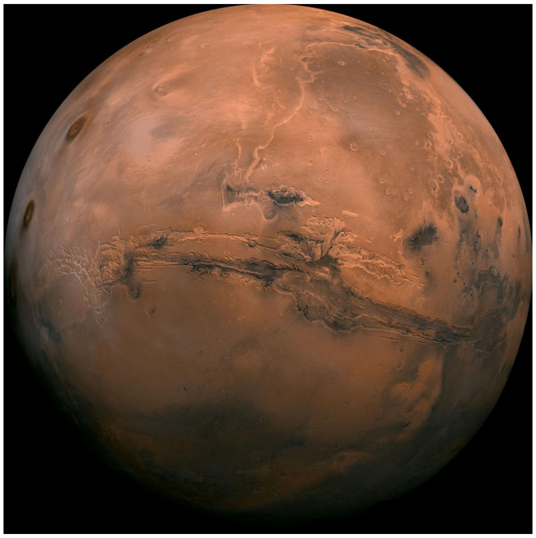


Illustration 2: Planet Mars

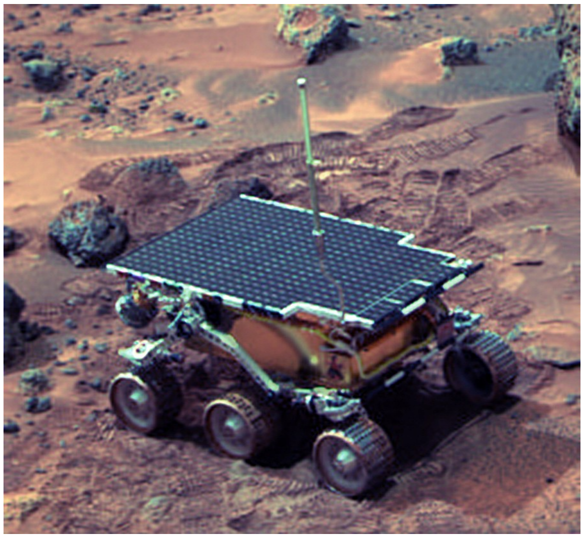


Illustration 1: Sojourner

A previous mission to Mars failed spectacularly due to a software error (mixing up miles and kilometres.)

The next mission (Sojourner) also had software bugs; a report discussing the bugs is included as an appendix to these notes.

Some of the report is easy to understand:

“Yes, we were concentrating heavily on the entry and landing software...”

Other parts address some issues that are directly related to this course. For example:

“The ASI/MET task had called select, which had called pipeIoctl(), which had called selNodeAdd(), which was in the process of giving the mutex semaphore. The ASI/ MET task was preempted and semGive() was not completed. Several medium priority tasks ran until the bc_dist task was activated.”

By the end of this course you will understand what a “mutex semaphore” is. And (possibly) you would not have introduced the kind of bug that the highly professional programmers at JPL did.

2 Announcements:

- Course management information distributed in class.
- Labs start the week of January 16, 2017 (No labs this week.)

- Counselling hours:
    - Monday: 1:00 pm – 2:00 pm
    - Friday: 2:00 pm – 3:00 pm
    - ENG-449, 979-5000 x6099, kclowes@ryerson.ca

3 What you will learn in this course

You will learn how operating systems work, the algorithms used and how to use the OS.

Topics include:

- How multitasking works (i.e. how the CPU can appear to run several programs simultaneously.)
- How *virtual memory* can allow a program (or programs) to use more memory than actually exists.
- How CPU hardware is needed and used to make these capabilities secure.
- Various scheduling algorithms.
- How concurrent tasks can share resources safely.
- How networked *client-server* (including *cloud computing*) work and how to use them.
- How machine virtualization works.

Although this not a programming course, you will write software in the lab.

Most of the lab projects use the **C** programming language. However, you will also learn a bit about **shell** programming. There will also be at least one lab that uses **Java**. (Multithreading is deeply built into the Java language but this aspect was not covered at all in your Java course.)

4 What is an Operating System?

For the purposes of this course, an Operating System is software that acts as an abstraction layer between the physical resources of the computer system and the application programs that run on the system. The OS manages the resources.

The resources managed by an OS include:

- Memory
- CPU
- Tasks (Processes and Threads)
- Input/Output (I/O)
- Files and file system

The OS manages these resources so that they are used efficiently. For example, “busy-waiting loops” (aka “spin-locks”) are rarely needed.

5 View of an OS

5.1 End user (non-technical)

- Users (for example, someone who writes documents with Microsoft *Word*) experience operating system differences at a "cosmetic" level.
- For example, if they change from Windows to Mac OS X, they quickly adapt and learn that the functionality of the square red "X" button on the top right of a widow used in Windows to close a window is accomplished with the red round button on the top left in Mac OS X.

5.2 Application programmer

- Non-GUI applications can often be written so that they can be compiled and run on different OSes with few changes.
- Example: a C programmer who restricts themselves to the standard C library.
- Applications (including GUI apps) written with an operating system independent language (such as Java) need no changes for different systems.
- GUI applications will generally be tightly integrated with the windowing system supported by the operating system.
- Application programmers should strive to isolate the OS-dependent code into small modules. To do this, the programmer needs to know the APIs for different systems.
- Programmers writing sophisticated software will also need to learn how to use some OS features directly. This is particularly important when designing apps that exploit the concurrent use of multiple threads or processes and ensure reliable and safe communication between these tasks.

5.3 System programmer/administrator

- These professionals need the greatest knowledge of operating systems including:
    - How operating systems work internally
    - How to write device drivers.
    - How to tune and/or modify the kernel.

6 Operating Systems covered

- Microsoft Windows.
- UNIX and UNIX-based including:
    - Linux
    - Solaris
    - Mac OS X

- ○ Android (Google)
- ○ Chromium (and Chrome—the OS, not the browser) (Google)
- ○ BSD Unix
- ○ Apple iOS (iPad, iPhone, iPod touch)
- ○ POSIX (IEEE standard. All Unix OSes support it and Microsoft Windows is almost completely compliant.)

7 The hardware/software interface

- The OS abstracts away the details of the underlying hardware from the programmer.
- Modern OSes also provide *security* but this requires additional hardware capabilities from the CPU. Specifically:
  - At least 2 modes of operation (user and supervisor[1]) enforced by the hardware. Only OS code is allowed to operate in supervisor mode.
  - Hardware memory management (paged and/or segmented *virtual memory*) that makes it possible for the OS to ensure that processes do not corrupt other processes.

8 Using the UNIX shell

Refer to the local guide (http://www.ee.ryerson.ca/guides/user/#SECTION00400)

9 Introduction to processes, I/O redirection, piping and the Shell

Consider the following simple programs which read characters from *stdin*, transform them and write the possibly modified character to *stdout*.

The “files” *stdin* and *stdout* are opened by the Operating System before a program begins. By default, *stdin* is the keyboard and *stdout* is the screen. However, the command line interface (in Windows, Linux, Mac OS, Android or any POSIX-compliant OS) allow the user to redefine to be any file.

This is possible because:

- POSIX systems have a very simple definition of a “file”. It is simply a sequence of bytes.
- Prior to UNIX, OSes defined several different types of files: ISAM, text, object code, etc. Which each had their own format. It was meaningless, for example, to talk of a keyboard as a “file”.
- With the UNIX definition, any real file is indeed “a sequence of bytes”.
- And a keyboard is also “a sequence of bytes” (although read-only).
- A screen is also a write-only “sequence of bytes”.

[1] Note: “supervisor” mode is also called “kernel” or “system” mode.

- We shall see later on how a POSIX OS function (called `dup` or `dup2`) can change the definition of what "file" *stdin* or *stdout* correspond to.

9.1 Convert to uppercase:

```
/*
 * Description:  Reads characters from stdin and writes each character
 * as uppercase to stdout.  (Note, "caseless" characters
 * such as numbers or punctuation are unchanged.)
 */
int main(int argc, char** argv) {

    int ch;
    while((ch = getchar()) != EOF) {
        putchar(toupper(ch));
    }
    exit(0);
}
```

You can download the source code

9.1.1 Remarks:

- Note that "ch" is an int, not a char. Why? Because "stdin" can be any file which can contain any characters (bytes). So a "special character" cannot be used to indicate "end-of-file". (Otherwise the file could not contain that byte!) The OS knows how many bytes are in the file and how many have been read. It must return something that is NOT a byte to indicate end of file.

9.2 Reverse the case

```
/*
 * File:   main.c
 * Author: Ken Clowes
 */

#include <stdio.h>
/*
 * Description:  Reads characters from stdin and writes
 * each character reversing its case (i.e. lower case is
 * converted to upper case and vice versa)
 * to stdout.  (Note, "caseless" characters such
 * as numbers or punctuation are unchanged.)
 */

int main(int argc, char** argv) {
    int ch;

    while((ch = getchar())!= EOF) {
        if(isupper(ch)) {
            putchar(tolower(ch));
        } else {
            putchar(toupper(ch));
        }
    }
    return (EXIT_SUCCESS);
}
```

9.3 Using redirection:

- Given the command “toupper” we can write:
  - toupper #read keyboard, echo as uppercase to screen
  - toupper < foo #Display file “foo” in uppercase on screen
  - toupper > goo #read from keyboard, write uppercase to file “goo”
  - toupper < foo > goo #readfile “foo”; uppercase to “goo”
  - toupper < foo > tmp1; reverse < tmp1 > goo2; #convert “foo” to lowercase in file “goo”

9.4 Using pipes (|)

- Cmd1 | Cmd2 connects the stdout of Cmd1 to the stdin of Cmd2
- Thus “toupper | reverse” converts stdin to lowercase on stdout.
- This is much more efficient that creating temporary intermediate files.

9.5 Using shell scripts:

Consider the “shell script” file “mycopy”:

```
#!/usr/bin/sh
./reverse < $1 | ./reverse > $2
```

A file can now be copied with: “mycopy foo goo” to copy file “foo” to “goo”.

10 Memory spaces

Programs need memory for different purposes:

- Memory for the program instructions (in machine language, usually generated by a compiler of source code written by the programmer.) (The “text” segment.)
- Memory for local variables and passed parameters. (The “stack”.)
- Memory for (initialized) global variables. (The “global data” segment.)

- Memory that is allocated dynamically (for example using “malloc”). (The “heap”.)
- Possibly global read-only memory.

The operating system is responsible for allocating and managing these memory segments.

10.1 A program to illustrate memory spaces (628MemorySpacesWeek1)

The following program illustrates that these kind of things so use different sections of memory.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//Global variables
int g1;
int g2;
char * gcp1;
char * gcp2;

// global function
void foo(int p, int q) {
    printf("Address of parameter p:   %p\n", &p);
    printf("Address of parameter q:   %p\n", &q);
    printf("\n");
}

//Another global function AND entry point
int main(int argc, char** argv) {
    int local_1;
    int local_2;

    gcp1 = "abc";
    printf("Address of main:          %p\n", main);
    printf("Address of foo:           %p\n", foo);
    printf("\n");

    printf("Address of g1:            %p\n", &g1);
    printf("Address of g2:            %p\n", &g2);
    printf("Address of gcp1:          %p\n", &gcp1);
    printf("Address of gcp2:          %p\n", &gcp2);
    printf("\n");

    printf("Address of local_1:       %p\n", &local_1);
    printf("Address of local_2:       %p\n", &local_2);
    printf("\n");

    foo(local_1, 5);
    printf("Address of string 'abc':  %p\n", gcp1);
    printf("\n");
    gcp2 = (char *) malloc(strlen(gcp1) + 1);
    strcpy(gcp2, gcp1);
    printf("Address of copied string: %p\n", gcp2);
    printf("\n");


    *gcp2 = 'x';
    printf("%s\n", gcp2);

/*What happens when the next line is uncommented? */
//    *gcp1 = 'x';
    printf("Goodbye\n");
    return (EXIT_SUCCESS);
}
```

You can obtain the source code here

10.2 Analysis of output

The output (on my Lenovo laptop under Windows 10 compiled with gcc) is:

```
Address of main:          0x100401122
Address of foo:           0x1004010e0

Address of g1:            0x1004071a0
Address of g2:            0x1004071b0
Address of gcp1:          0x1004071a8
Address of gcp2:          0x1004071b8

Address of local_1:       0x5fcb1c
Address of local_2:       0x5fcb18

Address of parameter p:   0x5fcaf0
Address of parameter q:   0x5fcaf8

Address of string 'abc':  0x10040306c

Address of copied string: 0x1004071b8

xbc
Goodbye
```

NOTES:

1. All of the global variables are in the area 0x100407100–0x1004071ff. This called the “global data” area.
2. The functions (main and foo) are in the area 0x100407100–0x100407ff. This is called the “text” area.
3. The local variables and passed parameters are in the area 0x5fca00–0x5cbff. This is the “stack”.
4. The dynamically allocated space (&gcp2[0]) is 0x6000483f0. This is in the “heap”.
5. The string “abc” is in a global **read-only** area.

11 Posix process creation with `fork()`

```c
/*
 * File:   main.c
 * Author: Ken Clowes
 *
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int global = 5;

int main(int argc, char** argv) {
    int local = 5;
    int i;

    if (fork() == 0) {
        for (i = 0; i < 10; i++) {
            local++;
            global++;
            printf("CHILD local: %d, global: %d\n", local, global);
            usleep(500);
        }
        exit(EXIT_SUCCESS);
    }
    for (i = 0; i < 10; i++) {
            local--;
            global--;
            printf("PARENT local: %d, global: %d\n", local, global);
            usleep(500);
        }

    return (EXIT_SUCCESS);
}
```

The source code is available here

12 Post lecture study guide

1. Read the Wikipedia article on operating systems.
2. Suppose you have to write 3 C compilers:
    - one for a PC running Windows 10

- one for a MacBook Pro running Mac OS X.
- one for an iPad running iOS 3 on an ARM 32-bit microprocessor

Which two compilers would be the most similar? Why?

13 Prep for next lecture (Monday, Jan 16, 2017)

- Try to remember how to use the C programming language. Can you still write the “Hello World” program?
- We will be using the Netbeans Integrated Development Environment in this course. It is free open-source and runs on all popular operating systems. I suggest you download and install it on your own computer.
- If you are using Microsoft Windows (XP, Vista or 7, 8 or 10), download and install cygwin (which allows most Unix commands and system calls to work under Windows.)

14 Some puzzles

1. See if you can compile and run the following short C program. Can you explain the output?

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char * argv[], char **envp) {
   int x = 1;
   char ch[4];
   printf("x = %d\n", x);
   ch[12] = '1';
   printf("x = %d\n", x);
   return EXIT_SUCCESS;
}
```

2. What about this program. What is the output?

```c
main() { char *s="main() { char *s=%c%s%c; printf(s,34,s,34); }"; printf(s,34,s,34); }
```

Appendix

14.1 Appendix A Mars

12/7/2016 What really happened on Mars? Authoritative Account

From: Glenn E Reeves[SMTP:Glenn.E.Reeves@jpl.nasa.gov]
Sent: Monday, December 15, 1997 10:28 AM
To: jack@telerobotics.jpl.nasa.gov; mishkin@telerobotics.jpl.nasa.gov; rmanning@mail1.jpl.nasa.gov; miked@wrs.com; Mike Jones;
gregg@wrs.com; dlre@chevron.com; David.Cummings@andrew.com; pky@appsig.com; karl_schneider@radixtek.com; lisa@wrs.com;
dave@wrs.com; John Krumm; bilzabub@pclink.com; mike_conrad@atk.com; elf@wavemark.com; mlisa@erols.com; Thomas Blumer;
shep@sunne.East.Sun.com; kmarks@xionics.com; frederick_hallock@lotus.com; athomas@xenergy.com; andy@harlequin.com;
don@wavemark.com; bostic@bostic.com; samiller@red.primextech.com; higgins@daffy.fnal.gov; intersci@smof.demon.co.uk;
kay@rsvl.unisys.com; glen@qnx.com; nev@bostic.com; jharris@mail1.jpl.nasa.gov; Richard A Volpe; Richard V Welch; Henry W Stone;
Allen R Sirota; Kim P Gostelow; Robert D Rasmussen; David J Eisenman; Daniel E Erickson; Udo Wehmeier; Peter R Gluck; David E
Smyth; Robert C Barry; Steven A Stolper; Alfred D Schoepke; David C Gruel; Guy S Beutelschies; Robert M Manning; Friedrich A
Krug; Samuel H Zingales; Steven M Collins; J Morgan Parker
Subject: Re: [Fwd: FW: What really happened on Mars?]

What really happened on Mars?

By now most of you have read Mike’s (mbj@microsoft.com) summary of Dave Wilner’s comments given at the IEEE RealTime Systems Symposium. I don’t know Mike and I didn’t attend the symposium (though I really wish I had now) and I have not talked to Dave Wilner since before the talk. However, I did lead the software team for the Mars Pathfinder spacecraft. So, instead of trying to find out what was said I will just tell you what happened. You can make your own judgments.

I sent this message out to everyone who was a recipient of Mike’s original that I had an email address for. Please pass it on to anyone you sent the first one to. Mike, I hope you will post this wherever you posted the original.
Since I want to make sure the problem is clearly understood I need to step through each of the areas which contributed to the problem.

THE HARDWARE

The simplified view of the Mars Pathfinder hardware architecture looks like this. A single CPU controls the spacecraft. It resides on a VME bus which also contains interface cards for the radio, the camera, and an interface to a 1553 bus. The 1553 bus connects to two places : The "cruise stage" part of the spacecraft and the "lander" part of the spacecraft. The hardware on the cruise part of the spacecraft controls thrusters, valves, a sun sensor, and a star scanner. The hardware on the lander part provides an interface to accelerometers, a radar altimeter, and an instrument for meteorological science known as the ASI/MET. The hardware which we used to interface to the 1553 bus (at both ends) was inherited from the Cassini spacecraft. This hardware came with a specific paradigm for its usage : the software will schedule activity at an 8 Hz rate. This **feature** dictated the architecture of the software which controls both the 1553 bus and the devices attached to it.

THE SOFTWARE ARCHITECTURE

The software to control the 1553 bus and the attached instruments was implemented as two tasks. The first task controlled the setup of transactions on the 1553 bus (called the bus scheduler or bc_sched task) and the second task handled the collection of the transaction results i.e. the data. The second task is referred to as the bc_dist (for distribution) task. A typical timeline for the bus activity for a single cycle is shown below. It is not to scale. This cycle was constantly repeated.

```
  |< ------------- .125 seconds ------------------------>|
    |<***************|                   |********|   |**>|

                     |<- bc_dist active ->|    bc_sched active
    |< -bus active ->|                            |<->|

----|----------------|--------------------|--------|---|---|-------
    t1               t2                   t3       t4  t5  t1
```

The *** are periods when tasks other than the ones listed are executing. Yes, there is some idle time.

t1 - bus hardware starts via hardware control on the 8 Hz boundary. The transactions for the this cycle had been set up by the previous execution of the bc_sched task.
t2 - 1553 traffic is complete and the bc_dist task is awakened.
t3 - bc_dist task has completed all of the data distribution
t4 - bc_sched task is awakened to setup transactions for the next cycle
t5 - bc_sched activity is complete

The bc_sched and bc_dist tasks check each cycle to be sure that the other had completed its execution. The bc_sched task is the highest priority task in the system (except for the vxWorks "tExec" task). The bc_dist is third highest (a task controlling the entry and landing is second). All of the tasks which perform other spacecraft functions are lower. Science functions, such as imaging, image compression, and the ASI/MET task are still lower. Data is collected from devices connected to the 1553 bus only when they are powered. Most of the tasks in the system that access the information collected over the 1553 do so via a double buffered shared memory mechanism into which the bc_dist task places the latest data. The exception to this is the ASI/MET task which is delivered its information via an interprocess communication mechanism (IPC). The IPC mechanism uses the vxWorks pipe() facility. Tasks wait on one or more IPC "queues" for messages to arrive. Tasks use the select() mechanism to wait for message arrival. Multiple queues are used when both high and lower priority messages are required. Most of the IPC traffic in the system is not for the delivery of realtime data. However, again, the exception to this is the use of the IPC mechanism with the ASI/MET task. The cause of the reset on Mars was in the use and configuration of the IPC mechanism.

THE FAILURE

The failure was identified by the spacecraft as a failure of the bc_dist task to complete its execution before the bc_sched task started. The reaction to this by the spacecraft was to reset the computer. This reset reinitializes all of the hardware and software. It also terminates the execution of the current ground commanded activities. No science or engineering data is lost that has already been collected (the data in RAM is recovered so long as power is not lost). However, the remainder of the activities for that day were not accomplished until the next day.

The failure turned out to be a case of priority inversion (how we discovered this and how we fixed it are covered later). The higher priority bc_dist task was blocked by the much lower priority ASI/MET task that was holding a shared resource. The ASI/MET task had acquired this resource and then been preempted by several of the medium priority tasks. When the bc_sched task was activated, to setup the transactions for the next 1553 bus cycle, it detected that the bc_dist task had not completed its execution. The resource that caused this problem was a mutual exclusion semaphore used within the select() mechanism to control access to the list of file descriptors that the select() mechanism was to wait on.

The select mechanism creates a mutual exclusion semaphore to protect the "wait list" of file descriptors for those devices which support select. The vxWorks pipe() mechanism is such a device and the IPC mechanism we used is based on using pipes. The ASI/MET task had called select, which had called pipeIoctl(), which had called selNodeAdd(), which was in the process of giving the mutex semaphore. The ASI/ MET task was preempted and semGive() was not completed. Several medium priority tasks ran until the bc_dist task was activated. The bc_dist task attempted to send the newest ASI/MET data via the IPC mechanism which called pipeWrite(). pipeWrite() blocked, taking the mutex semaphore. More of the medium priority tasks ran, still not allowing the ASI/MET task to run, until the bc_sched task was awakened. At that point, the bc_sched task determined that the bc_dist task had not completed its cycle (a hard deadline in the system) and declared the error that initiated the reset.

HOW WE FOUND IT

The software that flies on Mars Pathfinder has several debug features within it that are used in the lab but are not used on the flight spacecraft (not used because some of them produce more information than we can send back to Earth). These features were not "fortuitously" left enabled but remain in the software by design. We strongly believe in the "test what you fly and fly what you test" philosophy.

One of these tools is a trace/log facility which was originally developed to find a bug in an early version of the vxWorks port (Wind River ported vxWorks to the RS6000 processor for us for this mission). This trace/log facility was built by David Cummings who was one of the software engineers on the task. Lisa Stanley, of Wind River, took this facility and instrumented the pipe services, msgQ services, interrupt handling, select services, and the tExec task. The facility initializes at startup and continues to collect data (in ring buffers) until told to stop. The facility produces a voluminous dump of information when asked.

After the problem occurred on Mars we did run the same set of activities over and over again in the lab. The bc_sched was already coded so as to stop the trace/log collection and dump the data (even though we knew we could not get the dump in flight) for this error. So, when we went into the lab to test it we did not have to change the software.
In less that 18 hours we were able to cause the problem to occur. Once we were able to reproduce the failure the priority inversion problem was obvious.

HOW WAS THE PROBLEM CORRECTED

Once we understood the problem the fix appeared obvious : change the creation flags for the semaphore so as to enable priority inheritance. The Wind River folks, for many of their services, supply global configuration variables for parameters such as the "options" parameter for the semMCreate used by the select service (although this is not documented and those who do not have vxWorks source code or have not studied the source code might be unaware of this feature). However, the fix is not so obvious for several reasons :

1) The code for this is in the selectLib() and is common for all device creations. When you change this global variable all of the select semaphores created after that point will be created with the new options. There was no easy way in our initialization logic to only modify the semaphore associated with the pipe used for bc_dist task to ASI/MET task communications.
2) If we make this change, and it is applied on a global basis, how will this change the behavior of the rest of the system ?
3) The priority inheritance option was deliberately left out by Wind River in the default selectLib() service for optimum performance. How will performance degrade if we turn priority inheritance on?
4) Was there some intrinsic behavior of the select mechanism itself that would change if priority inheritance was enabled ?
We did end up modifying the global variable to include the priority inheritance option. This corrected the problem. We asked Wind River to analyze the potential impacts for (3) and (4). They concluded that the performance impact would be minimal and that the behavior of select() would not change so long as there was always only one task waiting for any particular file descriptor. This is true in our system. I believe that the debate at Wind River still continues on whether the priority inheritance option should be on as the default. For (1) and (2) the change did alter the characteristics of all of the select semaphores. We concluded, both by analysis and test, that there was no adverse behavior. We tested the system extensively before we changed the software on the spacecraft.

HOW WE CHANGED THE SOFTWARE ON THE SPACECRAFT

No, we did not use the vxWorks shell to change the software (although the shell is usable on the spacecraft). The process of "patching" the software on the spacecraft is a specialized process. It involves sending the differences between what you have onboard and what you want (and have on Earth) to the spacecraft. Custom software on the spacecraft (with a whole bunch of validation) modifies the onboard copy. If you want more info you can send me email.

WHY DIDN'T WE CATCH IT BEFORE LAUNCH?

The problem would only manifest itself when ASI/MET data was being collected and intermediate tasks were heavily loaded. Our before launch testing was limited to the "best case" high data rates and science activities. The fact that data rates from the surface were higher than anticipated and the amount of science activities proportionally greater served to aggravate the problem. We did not expect nor test the "better than we could have ever imagined" case.

HUMAN NATURE, DEADLINE PRESSURES

We did see the problem before landing but could not get it to repeat when we tried to track it down. It was not forgotten nor was it deemed unimportant. Yes, we were concentrating heavily on the entry and landing software. Yes, we considered this problem lower priority. Yes, we would have liked to have everything perfect before landing. However, I don't see any problem here other than we ran out of time to get the lower priority issues completed.

We did have one other thing on our side; we knew how robust our system was because that is the way we designed it. We knew that if this problem occurred we would reset. We built in mechanisms to recover the current activity so that there would be no interruptions in the science data (although this wasn't used until later in the landed mission). We built in the ability (and tested it) to go through multiple resets while we were going through the Martian atmosphere. We designed the software to recover from radiation induced errors in the memory or the processor. The spacecraft would have even done a 60 day mission on its own, including deploying the rover, if the radio receiver had broken when we landed. There are a large number of safeguards in the system to ensure robust, continued operation in the event of a failure of this type. These safeguards allowed us to designate problems of this nature as lower priority.

We had our priorities right.

ANALYSIS AND LESSONS

Did we (the JPL team) make an error in assuming how the select/pipe mechanism would work ? Yes, probably. But there was no conscious decision to not have the priority inheritance option enabled. We just missed it. There are several other places in the flight software where similar protection is required for critical data structures and the semaphores do have priority inversion protection. A good lesson when you fly COTS stuff make
sure you know how it works. Mike is quite correct in saying that we could not have figured this out **ever** if we did not have the tools to give us the insight. We built
many of the tools into the software for exactly this type of problem. We always planned to leave them in. In fact, the shell (and the stdout
stream) were very useful the entire mission. If you want more detail send me a note.

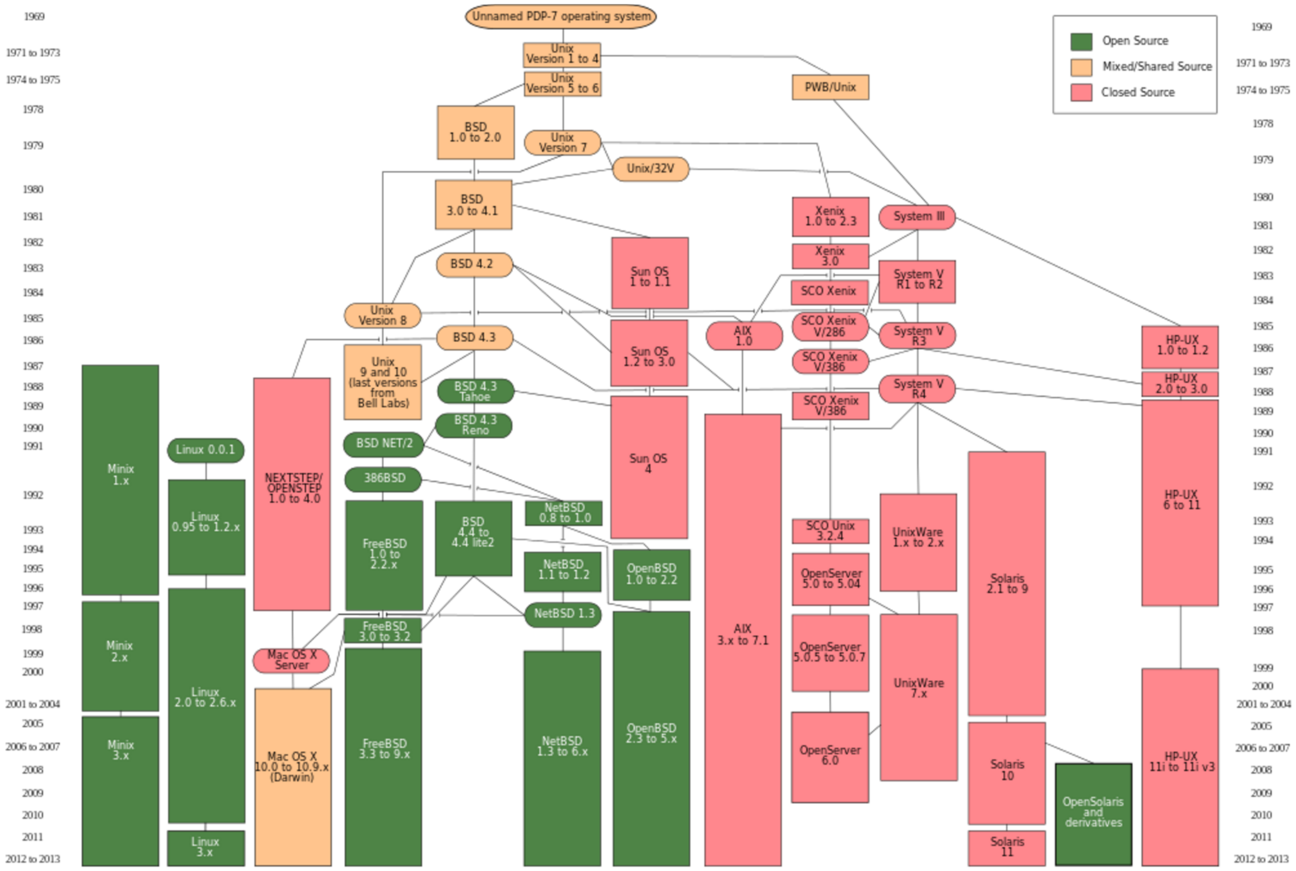
SETTING THE RECORD STRAIGHT

First, I want to make sure that everyone understands how I feel in regard to Wind River. These folks did a fantastic job for us. They were enthusiastic and supported us when we came to them and asked them to do an affordable port of vxWorks. They delivered the alpha version in 3 months. When we had a problem they put some of the brightest engineers I have ever worked with on the problem. Our communication with them was fantastic. If they had not done such a professional job the Mars Pathfinder mission would not have been the success that it is.

Second, Dave Wilner did talk to me about this problem before he gave his talk. I could not find my notes where I had detailed the description of the problem. So, I winged it and I sure did get it wrong. Sorry Dave.

ACKNOWLEDGMENTS

First, thanks to Mike for writing a very nice description of the talk. I think I have had probably 400 people send me copies. You gave me
the push to write the part of the Mars Pathfinder EndofMission
report that I had been procrastinating doing.
A special thanks to Steve Stolper for helping me do this.
The biggest thanks should go to the software team that I had the privilege of leading and whose expertise allowed us to succeed :
Pam Yoshioka
Dave Cummings
Don Meyer
Karl Schneider
Greg Welz
Rick Achatz
Kim Gostelow
Dave Smyth
Steve Stolper
Also ,
Miguel San Martin
Sam Sirlin
Brian Lazara (WRS)
Mike Deliman (WRS)
Lisa Stanley (WRS)
Glenn Reeves


Open Source
Mixed/Shared Source
Closed Source
1969
1971 to 1973
1974 to 1975
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001 to 2004
2005
2006 to 2007
2008
2009
2010
2011
2012 to 2013
Unnamed PDP-7 operating system
Unix Version 1 to 4
Unix Version 5 to 6
PWB/Unix
BSD 1.0 to 2.0
Unix Version 7
Unix/32V
BSD 3.0 to 4.1
Xenix 1.0 to 2.3
System III
Xenix 3.0
Sun OS 1 to 1.1
BSD 4.2
System V R1 to R2
SCO Xenix
Unix Version 8
SCO Xenix V/286
AIX 1.0
System V R3
BSD 4.3
Sun OS 1.2 to 3.0
SCO Xenix V/386
HP-UX 1.0 to 1.2
Unix 9 and 10 (last versions from Bell Labs)
BSD 4.3 Tahoe
System V R4
HP-UX 2.0 to 3.0
SCO Xenix V/386
BSD 4.3 Reno
Minix 1.x
Linux 0.0.1
NEXTSTEP/ OPENSTEP 1.0 to 4.0
BSD NET/2
Sun OS 4
386BSD
HP-UX 6 to 11
Linux 0.95 to 1.2.x
NetBSD 0.8 to 1.0
BSD 4.4 to 4.4 lite2
FreeBSD 1.0 to 2.2.x
SCO Unix 3.2.4
UnixWare 1.x to 2.x
NetBSD 1.1 to 1.2
OpenBSD 1.0 to 2.2
OpenServer 5.0 to 5.04
Solaris 2.1 to 9
NetBSD 1.3
Minix 2.x
FreeBSD 3.0 to 3.2
AIX 3.x to 7.1
OpenServer 5.0.5 to 5.0.7
Mac OS X Server
Linux 2.0 to 2.6.x
UnixWare 7.x
Mac OS X 10.0 to 10.9.x (Darwin)
FreeBSD 3.3 to 9.x
NetBSD 1.3 to 6.x
OpenBSD 2.3 to 5.x
Minix 3.x
OpenServer 6.0
Solaris 10
OpenSolaris and derivatives
HP-UX 11i to 11i v3
Linux 3.x
Solaris 11