Assessed Coursework

Managing Resources without Deadlock

1 Question 1: Locking

Our operating system implements locks using the following code:

```
LOCK: CLI ; disable interrupts
TEST L
BNZ LOCK
MOV #1, L
STI ; enable interrupts

UNLOCK: MOV #0, L
```

i. Using *pseudocode* write a procedure to explain how these locks will protect a critical region in process P, which carries out the following calculation: \( V = V - 10 \).

(0.5 marks)

ii. In less than 50 words, say why we disable and re-enable interrupts at the beginning and end of the lock code.

(0.5 marks)

iii. In less than 200 words, comment on the behaviour of this lock when many processes \( P_1, P_2, \ldots, P_n \) are run on the same processor. Assume the time-slice quantum for each process is the equivalent of one *pseudocode* instruction from your answer in (i), and show where the processes and the lock are interrupted.

(2 marks)

iv. Our operating system also allows semaphores to be used to protect critical regions. Process K is the code running on a door sensor that is counting the number of people in a room. The room has a maximum capacity of 100 people and the door can only let one person enter, or leave, the room at a given time. Using *pseudocode*, define the semaphore(s) needed for K’s enter and exit procedures for the door’s operation. Your total lines of pseudocode should not exceed 30!

(2 marks)

*Please note that answers breaking the code or word maximums will have marks deducted.*
2 Question 2: Fixing the Priority Inversion Problem in Simple Kernel

A straightforward use of synchronisation tools such as semaphores can sometimes lead to “priority inversion” in scheduling protocols, where a high-priority job ends up being blocked for an indefinite period of time by a lower-priority job. This was the underlying issue that caused some of the software problems experienced by NASA’s Mars Pathfinder mission (see attached reports). Your job in this assessed exercise is to make changes to the scheduling scheme used in Simple Kernel which fix this problem.

2.1 Getting Started

Use as your starting point for this part of the exercise the result of the unassessed exercise 6 (“Some Simple Modifications to Simple Kernel”). You can obtain a sample solution for this unassessed exercise as follows:

```
cp -r ~/ob3/SimpleKernelExercise2-Start.tar.gz .
```

Don’t forget the “.” at the end of that command. Unpack that directory as described in the handout for the first Simple Kernel exercise.

Examine the file `user.c` which comes with that directory. You will see that this implements three processes: `enterRoom`, `leaveRoom` and `medium_priority`. The priorities of these processes are as follows: `enterRoom`: high, `leaveRoom`: low, `medium_priority`: medium.

2.2 What to do

i. Your first task is to implement the locking mechanism (using semaphores) from item (iv) in Question 1 in this user program. Declare any necessary semaphores, and surround the print statements in the `enterRoom` and `leaveRoom` processes with the required calls to `P` and `V`. You should not have to add much more than 15 lines of code!

(1 mark)

You can slow things down by uncommenting the `delay(2)` calls in these processes. These should stay inside the inner critical region together with the print statement.

If you have done this part of the exercise correctly, the output should look something like this:

```
EMEMMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEMLMEML
iii. Create a new type of semaphore called resource. Modify sem.h and sem.c to add a declaration for the new type and to implement the following functions that operate on this type of semaphore: initResource, Pr, Vr.

Resources at this stage are exactly like semaphores, except that they are understood to represent a resource which is accessed by exactly one producer and exactly one consumer. This is done by storing two additional fields in the structure for this type: an integer for the PID of the consumer and an integer for the PID of the producer. The initResource function will take two additional parameters compared to initSem, representing the producer and consumer PIDs.

(1 mark)

iv. Add the following function prototype to queue.h:

```c
void removeQ( queueP q, int pid );
```

Implement this function in queue.c. This function should remove the process with PID pid from the given queue q. You may assume that the process is present exactly once in the queue (this will be guaranteed by the way that this function is used).

(1 mark)

v. Add a new field int own_priority to the PCB structure in procP.h. This will be used to store a processes’ “own” (original) priority while it has temporarily inherited another processes’ priority in order to resolve a priority inversion issue. own_priority should be initialised to the same value as the priority field pr in the create function in proc.c. Also, add the following function prototypes to procP.h:

```c
void inherit_priority( const int producer, const int consumer );
void reset_priority( const int pid );
```

vi. Implement the functions inherit_priority and reset_priority in proc.c.

inherit_priority should cause the producer process to inherit the priority of the consumer process. As discussed in the lecture, this means that if the current state of the producer is READY, you have to remove the producer from its current priority queue and add it to its new priority queue.

reset_priority resets the priority of a process to its original value.

Modify the Pr function in sem.c to call the inherit_priority function when a process blocks on a resource semaphore. Modify Vr to call reset_priority when a process that is running on a priority other than its own lifts a resource semaphore.

Modify your user program in user.c so as to use a resource semaphore (rather than an ordinary semaphore) to represent the room. You should now have solved the priority inversion problem in (ii).

(2 marks)

2.3 What to hand in

For question 2, hand in a tar archive of the system directory in your modified Simple Kernel OS. You can produce such a tar file from the ICOS directory by typing

```
tar cvf system.tar system
```

Olav Beckmann, Imperial College London, December 5, 2005
Assessed Tutorial Exercise

Priority Inversion Problem

• Assume a system has processes with 3 priority levels: high, medium and low.
• Let a high-priority process H share a resource with a low-priority process L, for example L producing data that H consumes.
• What happens when H has to wait for a semaphore that will be released by L?
• It all depends on what the medium-priority processes do!

Notes on Solution to the Priority Inversion Problem

Theory
• Let the low-priority process inherit the priority of the process that is waiting for it.
• To keep things simple, we will implement a special type of semaphore ("resource") that has exactly 1 producer and 1 consumer.
• We need to register producer and consumer with "resource" semaphores when they are created.
• The idea is then that when a producer blocks on a resource semaphore, the priority of the consumer is set to that of the producer.

In Simple Kernel...
• Add a mechanism for PIDs (process identifiers).
• You can copy the existing implementation of semaphores and add to it.
• typedef struct Semaphore {
  int count;
  int producer, consumer;
  Queue waiting;
} Semaphore;
• Make new versions of initSema, P and V for resource-type semaphores.

Changing the Priority of a Process

• Do we want to change the priority of the producer process in all states?
• In running, what do we have to do?
• In delayed?
• In suspended?
• In ready?
• Add a mechanism for keeping track of process state.
From Mike Jones <mjones@microsoft.com>
Sent: Sunday, December 07, 1997 6:47 PM
Subject: What really happened on Mars?

The Mars Pathfinder mission was widely proclaimed as "flawless" in the early days after its July 4th, 1997 landing on the Martian surface. Successes included its unconventional "landing" -- bouncing onto the Martian surface surrounded by airbags, deploying the Sojourner rover, and gathering and transmitting voluminous data back to Earth, including the panoramic pictures that were such a hit on the Web. But a few days into the mission, not long after Pathfinder started gathering meteorological data, the spacecraft began experiencing total system resets, each resulting in losses of data. The press reported these failures in terms such as "software glitches" and "the computer was trying to do too many things at once."

This week at the IEEE Real-Time Systems Symposium I heard a fascinating keynote address by David Wolter, Chief Technical Officer of Wind River Systems. Wind River makes VxWorks, the real-time embedded systems kernel that was used in the Mars Pathfinder mission. In his talk, he explained in detail the actual software problems that caused the total system resets of the Pathfinder spacecraft, how they were diagnosed, and how they were solved. I wanted to share his story with each of you.

VxWorks provides preemptive priority scheduling of threads. Tasks on the Pathfinder spacecraft were executed as threads with priorities that were assigned in the usual manner reflecting the relative urgency of these tasks.

Pathfinder contained an "information bus", which you can think of as a shared memory area used for passing information between different components of the spacecraft. A bus management task ran frequently with high priority to move certain kinds of data in and out of the information bus. Access to the bus was synchronized with mutual exclusion locks (mutexes).

The meteorological data gathering task ran as an infrequent, low priority thread, and used the information bus to publish its data. When publishing its data, it would acquire a mutex, do writes to the bus, and release the mutex. If an interrupt caused the information bus thread to be scheduled while this mutex was held, and if the information bus thread then attempted to acquire this same mutex in order to retrieve published data, this would cause it to block on the mutex, waiting until the meteorological thread released the mutex before it could continue. The spacecraft also contained a communications task that ran with medium priority.

Most of the time this combination worked fine. However, very infrequently it was possible for an interrupt to occur that caused the (medium priority) communications task to be scheduled during the short interval while the high priority information bus thread was blocked waiting for the (low priority) meteorological data thread. In this case, the long-running communications task, having higher priority than the meteorological task, would prevent it from running, consequently preventing the blocked information bus task from running. After some time had passed, a watchdog timer would go off, notice that the data bus task had not been executed for some time, conclude that something had gone drastically wrong, and initiate a total system reset.

This scenario is a classic case of priority inversion.

HOW WAS THIS DEBUGGED?

VxWorks can be run in a mode where it records a total trace of all interesting system events, including context switches, uses of synchronization objects, and interrupts.

For the record, the paper was:

This note was widely circulated after I sent it to a few friends in the systems community. Among other places, it appeared in Peter G. Neumann's moderated Risks Forum (comp.risks) on Tuesday, 9 December 1997 in issue RISKS-19.49. Be sure to also read the follow-up message from Glenn Reeves of JPL, who led the software team for the Mars Pathfinder spacecraft.

ANALYSIS AND LESSONS

First and foremost, diagnosing this problem as a black box would have been impossible. Only detailed traces of actual system behavior enabled the faulty execution sequence to be captured and identified.

Secondly, leaving the "debugging" facilities in the system saved the day. Without the ability to modify the system in the field, the problem could not have been corrected.

Finally, the engineer's initial analysis that "the data bus task executes very frequently and is time-critical - we shouldn't spend the extra time in to perform priority inheritance" was exactly wrong. It is precisely in such time critical and important situations where correctness is essential, even at some additional performance cost.

HUMAN NATURE, DEADLINE PRESSURES

David told us that the JPL engineers later confessed that one or two system resets had occurred in their months of pre-flight testing. They had never been reproducible or explainable, and so the engineers, in a very human-nature response of denial, decided that they probably weren't important, using the rationale "it was probably caused by a hardware glitch".

Part of it too was the engineers' focus. They were extremely focused on ensuring the quality and flawless operation of the landing software. Should it have failed, the mission would have been lost. It is entirely understandable for the engineers to discount occasional glitches in the less-critical land-mission software, particularly given that a spacecraft reset was a viable recovery strategy at that phase of the mission.

THE IMPORTANCE OF GOOD THEORY/ALGORITHMS

David also said that some of the real heroes of the situation were some people from CMU who had published a paper he'd heard presented many years ago who first identified the priority inversion problem and proposed the solution. He apologized for not remembering the precise details of the paper or who wrote it. Bringing things full circle, it turns out that the three authors of this result were all in the room, and at the end of the talk were encouraged by the program chair to stand and be acknowledged. They were Lui Sha, John Lehoczky, and Raj Rajkumar. When was the last time you saw a room of people cheer a group of computer science theorists for their significant practical contribution to advancing human knowledge?:) It was quite a moment.

POSTLUDE
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 sec... 125 sec...

----------
|<-----|<-----|<-----|<-----|<-----|<-----|<-----|
|<->|<->|<->|<->|<->|<->|<->|
|<-------------|----------------|--------------------|--------|---|---|-------|
|<- bc_dist active ->|    bc_sched active
|<------------- .125 seconds ------------------------>|
```

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

- 11 - house bus starts via hardware control on the S band. The transactions for this cycle had been set up by the previous execution of the bc_dist task.
- 12 - 1553 traffic is complete and the bc_dist task is awakened.
- 13 - bc_dist task has completed all of the data distribution
- 14 - bc_sched task is awakened to setup transactions for the next cycle
- 15 - bc_sched activity is complete

The bc_sched and bc_dist tasks check each cycle to see that the other had completed its execution. The bc_sched task is the highest priority task in the system (except for the vxWorks "hExec" task). The bc_dist task 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 ASIMET 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 ASIMET task which is delivered its information via a interprocess communication 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 ASIMET 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 descriptions 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 pipel process is such a device and the IPC mechanism we use is based on pipes. The ASIMET task had called select, which had called sxNode:Add(), 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 semWait() the newest ASIMET data via the IPC mechanism which called writePipe()() which was blocked, taking the mutex semaphore. Most of the medium priority tasks ran, still not allowing the bc_dist 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 (as bad 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 of some 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 tracing facility which was originally developed to find a bug in an early version of the vxWorks process (Wind River ported vxWorks to the RS6000 processor for us for this mission). This tracing facility was built by David Cummings who was one of the software engineers on the task. Lou Stanley, of Wind River, took this facility and instrumented the pipe services, msg queues, interrupt handling, select services, and the tExec task. The facility initializes at startup and continues to collect data (in msg queues) 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 seeded so as to stop the tracing collection and dump the data (even though we knew we could not get the dump to 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 the priority inheritance. The Wind River folks, for many of their services, supply global configuration variables for parameters such as the "options" parameter for the semCreate() 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 semaphore creators affected after that point will be created with the new options. There was no easy way to our initialization logic to only modify the semaphore associated with the pipe used for bc_dist task to ASIMET 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 inversion option was deliberately left by Wind River in the default selectLib() service for optimum performance. How will performance degrade if we turn the priority inversion on?

4) Was there some intrinsic behavior of the select mechanism itself that would change if the priority inversion was enabled?

We did not apply modifying the global variable to include the priority inversion. This corrected the problem. We asked Wind River to analyze the potential impacts for (1) and (4). They concluded that the performance impacts would be minimal and that the behavior of (1) 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 the answer is: "the first task on the list who cares on whether the priority inversion option 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 ASIMET data was being collected and intermediate
tasks were heavily loaded. One 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 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 inversion 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 Witmer 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

I would like to thank 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 Achutz
Kim Gostelow
Dave Smyth
Steve Stolper

Also,

Miguel San Martin
Sam Sefian
Brian Lazarra (WRS)
Mike Deliman (WRS)
Lisa Stanley (WRS)

---------------

Glenn Reeves
Mars Pathfinder Flight Software Cognizant Engineer
glenn.e.reeves@jpl.nasa.gov

Related Links:

- Mike Jones’ initial “What really happened on Mars?” message
- Top-level “What really happened on Mars” page

Mike Jones’ home page