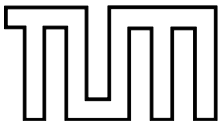


FAKULTÄT FÜR INFORMATIK
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Analyse von Softwarefehlern

Seminar Sommersemester 2021

Mars Pathfinder: Priority Inversion

Achilleas Tsakpinis

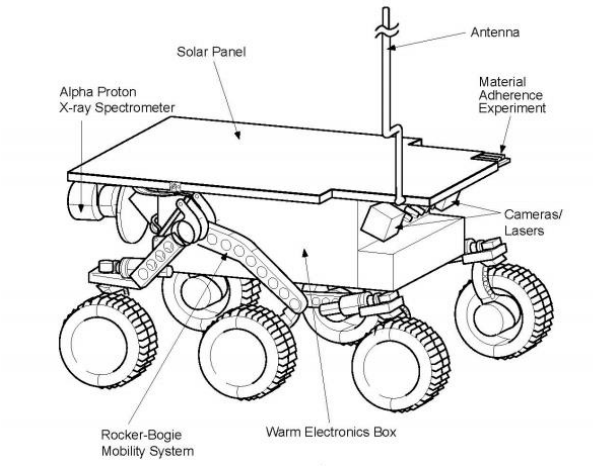
Inhalt

1. Einleitung

Am 4.12.1996 um 1:58 (EST) startete die Rakete des Mars Pathfinder Richtung Mars. Diese Mission war in jeder Hinsicht eine Neuheit für die Raumfahrt. Zum ersten Mal war ein Rover inklusive eines Landemoduls an Board der Rakete. Des Weiteren versuchte man, das erste Mal mit einem Rover auf der Marsoberfläche zu landen und anschließend damit auch Forschung zu betreiben. Wegen begrenzter Mittel wurden das Budget und die Dauer der Mission stark eingeschränkt und man entschied sich, das Projekt mit der „faster/better/cheaper"-Strategie zu realisieren. Dabei priorisiert man verschiedene Aufgaben des Projekts und plant mit Fehlern, die eventuell erst bei der Nutzung auftreten und auch erst dann behoben werden. Erschwert wurde die Mission aber auch durch äußerliche Faktoren. Durch die enorme Entfernung zur Erde war es unmöglich den Rover oder die Landung in Echtzeit zu steuern, wodurch auf autonome Systeme gesetzt wurde. Auch die Bandbreite war durch die Entfernung eingeschränkt und lag bei ungefähr 10 Mbit pro Marstag[1] [1] [2] [3].

Auf dem Mars gab es zudem weitere Herausforderungen. Der Rover wurde mit einer Kombination aus Solarpanels und Batterien betrieben. Um die Elektronik vor den nächtlichen Temperaturen von bis zu -100 Grad zu schützen, wurde diese in einer Box eingelagert, in der die Temperatur angepasst werden konnte. Die Reifen des Rovers waren ebenfalls eine technische Leistung. Diese mussten den spitzen Felskanten standhalten, aber auch Vortrieb garantieren und gleichzeitig leicht sein. Am 4.7.1997 gelang die komplexe Landung, die aus mehreren Schritten bestand. Kurz bevor das Landemodul auf der Oberfläche ankam, wurde es wie geplant abgeworfen und landete sicher auf Luftkissen, die das Landemodul ummantelten. Am 5.7.1997 wurde der Rover das erste Mal ausgefahren, womit die NASA und somit Amerika das erste Land war, das einen autonomen Rover auf den Mars gebracht hat [1] [2].

2. Der Marsrover



- 16kg
- 65cm x 48cm x 30cm
- Autonom gesteuert
- 0.6 m/min
- 10m Operationsradius
- Solar/Batteriestrom betrieben [1]

Abbildung 1: Bild des Rovers mit wissenschaftlichen Messgeräten

Quelle: [1]

[1] Ein Marstag ist in etwa 24 h, 39 min und 35 sec lang.

Der Pathfinder wird von einem IBM RS/6000 Prozessor mit einem proprietären Echtzeitbetriebssystem namens VxWorks betrieben. Das Betriebssystem wird oft in Kleingeräten in der Luft- und der Raumfahrt verwendet. Des Weiteren gibt es eine Reihe von wissenschaftlichen Geräten auf dem Rover [1]:

Alphapartikel-Röntgenspektrometer (APXS): Analysiert die chemische Zusammensetzung von entnommenen Bodenproben.

Stereoskopische Kamera (IMP): Erzeugt eine Art 3D-Effekt durch die Aneinanderreihung von mindestens zwei Kameralinsen.

Messstation (ASI/MET): Wurde genutzt, um Temperatur, Wind und Druck auf dem Mars zu messen.

Darüber hinaus wurden drei weitere Außenkameras angebracht [1].

3. Technischer Aufbau

Das Betriebssystem VxWorks verwaltet verschieden priorisierte Prozesse (siehe Abbildung 3). Der BC_Sched Prozess hat die höchste Priorität 4 und achtet darauf, dass der Zyklus richtig ausgeführt wird und wer als nächster Zugriff auf den Bus haben darf. BC_Dist verarbeitet entstandene Daten mit Priorität 3 und ist der sogenannte Konsument der Pipe. ASI/MET sammelt Daten und schreibt sie auf die Pipe mit Priorität 1 und ist der sogenannte Produzent für die Pipe. Des Weiteren gibt es diverse andere Prozesse wie den Kommunikationsprozess mit Priorität 2 [1] [2].

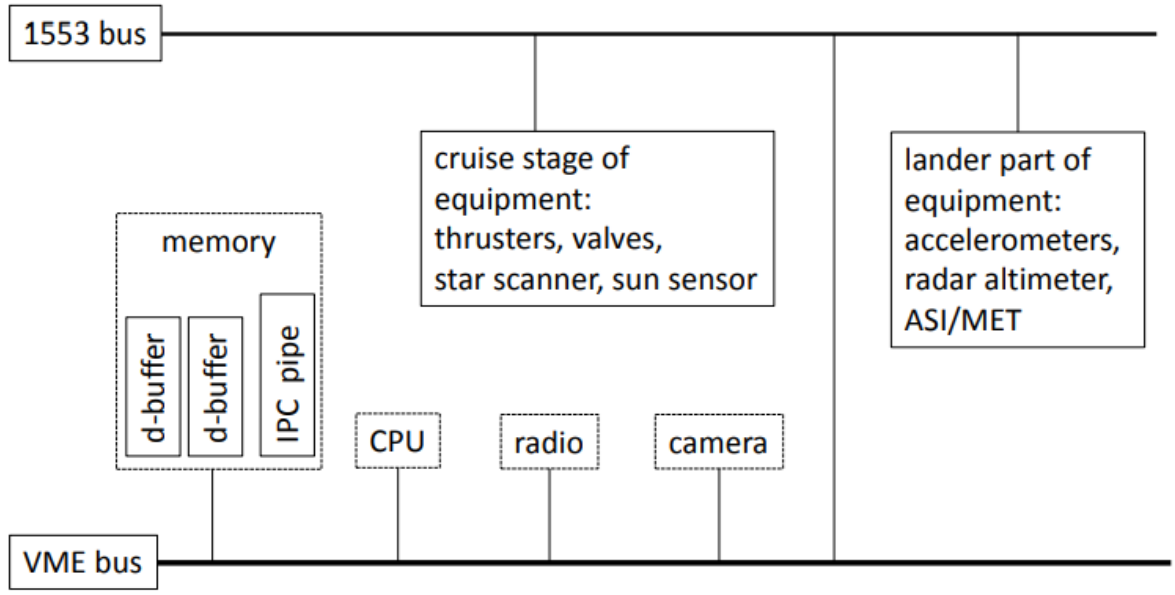


Abbildung 2: Technischer Aufbau der Hardware

Quelle: [1]

In Abbildung 2 sieht man, dass es zwei verschiedene Busse gibt. Der 1553 Bus bringt Daten der Messstation zur Main Memory und bindet verschiedene Sensoren (z.B. zur Steuerung) an. Der VME Bus verbindet CPU und Hauptspeicher, aber auch Instrumente zur Kommunikation und eine Kamera [1].

Der Bus hat einen Zyklus, der folgendermaßen aussieht [1]:

1. ASI/MET Prozess erzeugt Daten und schreibt sie in die Pipe.
2. BC_dist liest die Daten aus der Pipe und verarbeitet sie gegebenenfalls.
3. Weitere Prozesse wie der Kommunikationsprozess erhalten auch kurze Rechenzeit.
4. BC_Sched überprüft, ob der letzte Zyklus reibungslos abgelaufen ist.

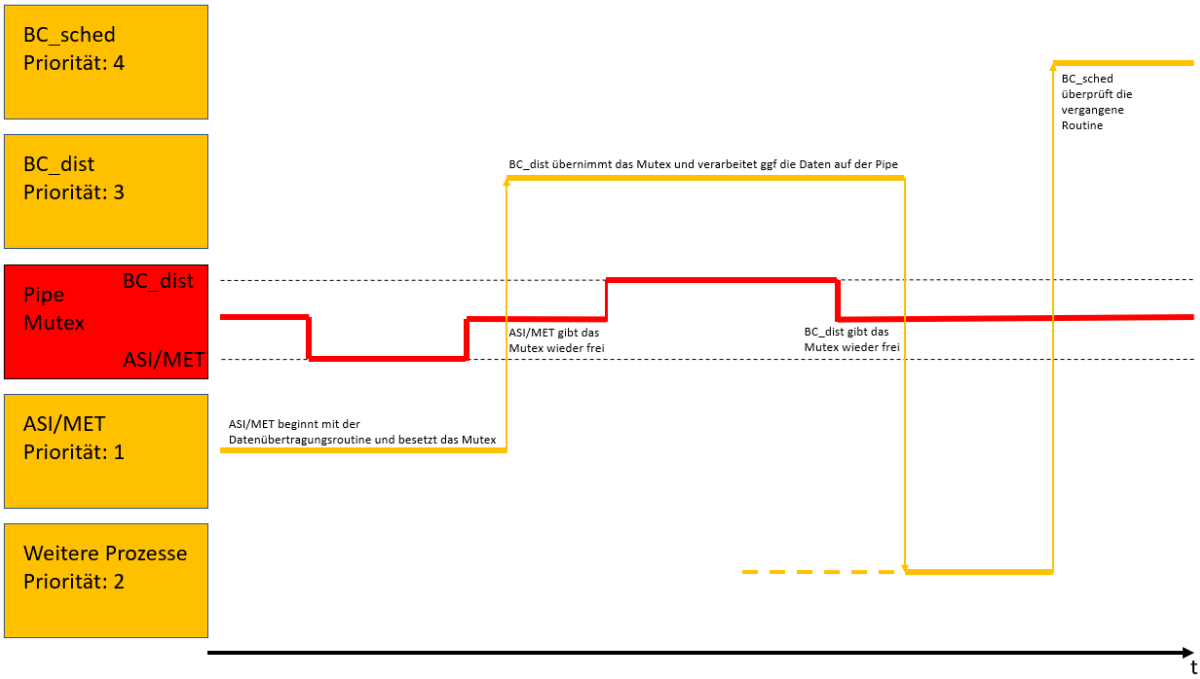


Abbildung 3: Fehlerfreie Busroutine

Inspiration: [4]

4. Softwarebug und mögliche Lösungen

Das Problem hat sich bemerkbar gemacht durch zufällige, nicht reproduzierbare Systemzurücksetzungen, die beim Datensammeln entstanden sind und einen Datenverlust bewirkt haben. Schon in der Testphase sind ähnliche Probleme aufgetreten, die aber aufgrund von niedriger Priorität vernachlässigt wurden. Um den aufgetretenen Fehler zu beschreiben, wird der Begriff der Prioritätsinversion eingeführt [2].

4.1 Prioritätsinversion

Eine Prioritätsinversion tritt ein, wenn ein Prozess hoher Priorität durch einen Prozess mit niedriger Priorität blockiert wird. Prinzipiell gibt es zwei Arten von Prioritätsinversionen [5]:

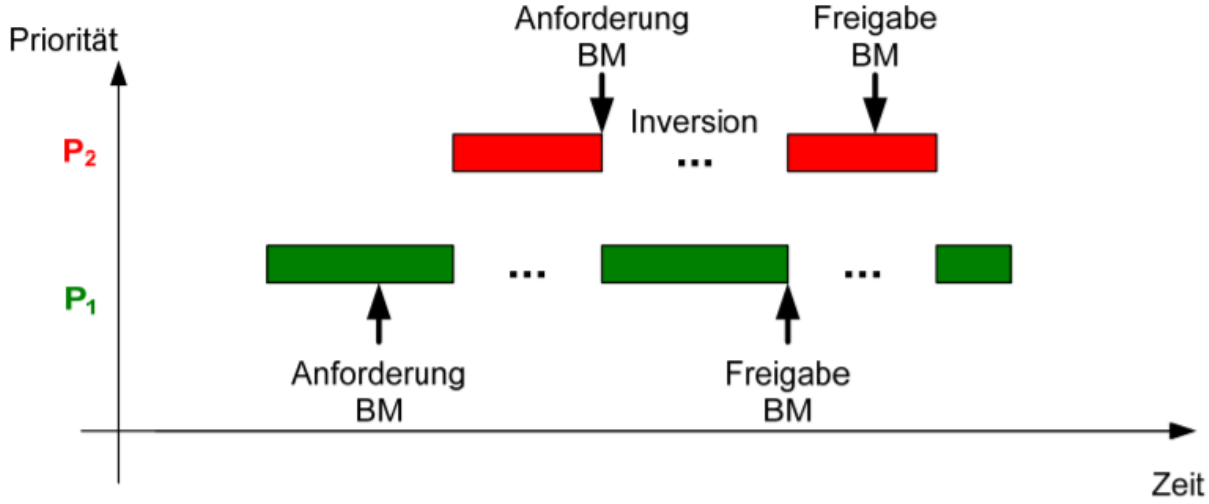


Abbildung 4: Begrenzte Prioritätsinversion

Quelle: [5]

Die begrenzte Prioritätsinversion ist weniger problematisch, hat aber zur Folge, dass der Prozess höherer Priorität (in Abbildung 4 in Rot dargestellt) für die Zeit des kritischen Abschnitts blockiert wird, da Grün das Mutex besitzt und Rot erst auf die Freigabe warten muss, um arbeiten zu können [5].

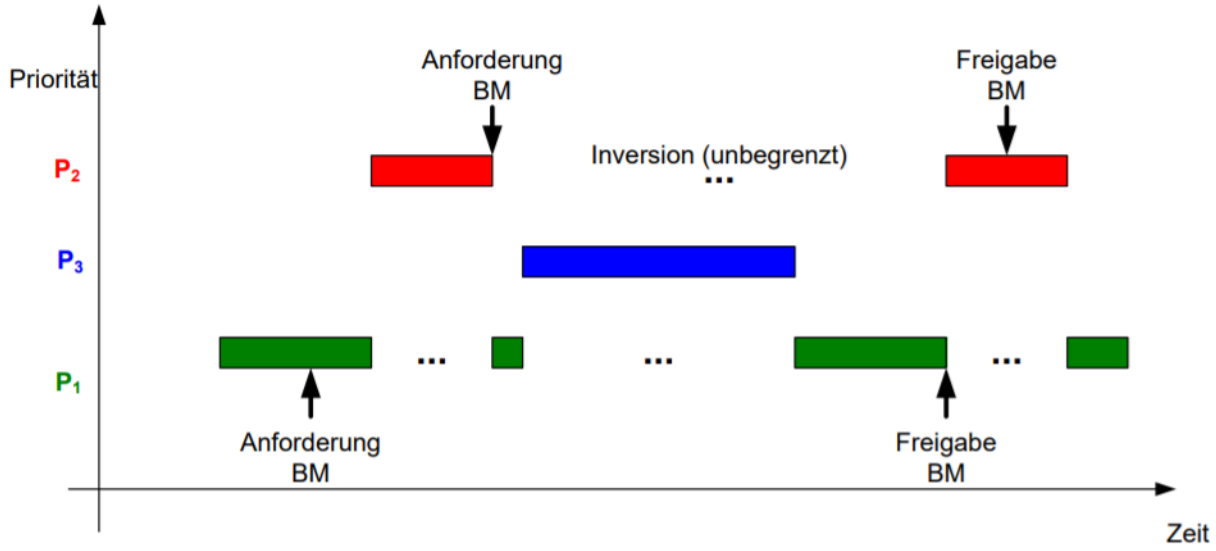


Abbildung 5: Unbegrenzte Prioritätsinversion

Quelle: [5]

Demgegenüber ist die unbegrenzte Prioritätsinversion sehr kritisch und sollte auf jeden Fall vermieden werden [5]. Diese entsteht, wenn es mindestens 3 Prozesse (hier Rot, Blau, Grün) mit jeweils absteigender Priorität gibt. Grün besetzt das Mutex, wird aber durch Rot unterbrochen. Rot fordert dann das besetzte Betriebsmittel (BM) an, wodurch Rot schlafen gelegt wird und Blau Prozessorzeit bekommt. Blau und auch andere Prozesse mit Priorität zwischen Rot und Grün können jetzt theoretisch unendlich lang weiterarbeiten, weil Grün das Mutex nicht freigeben kann und Rot auf das Mutex warten muss.

4.2 Softwarefehler beim Mars Pathfinder

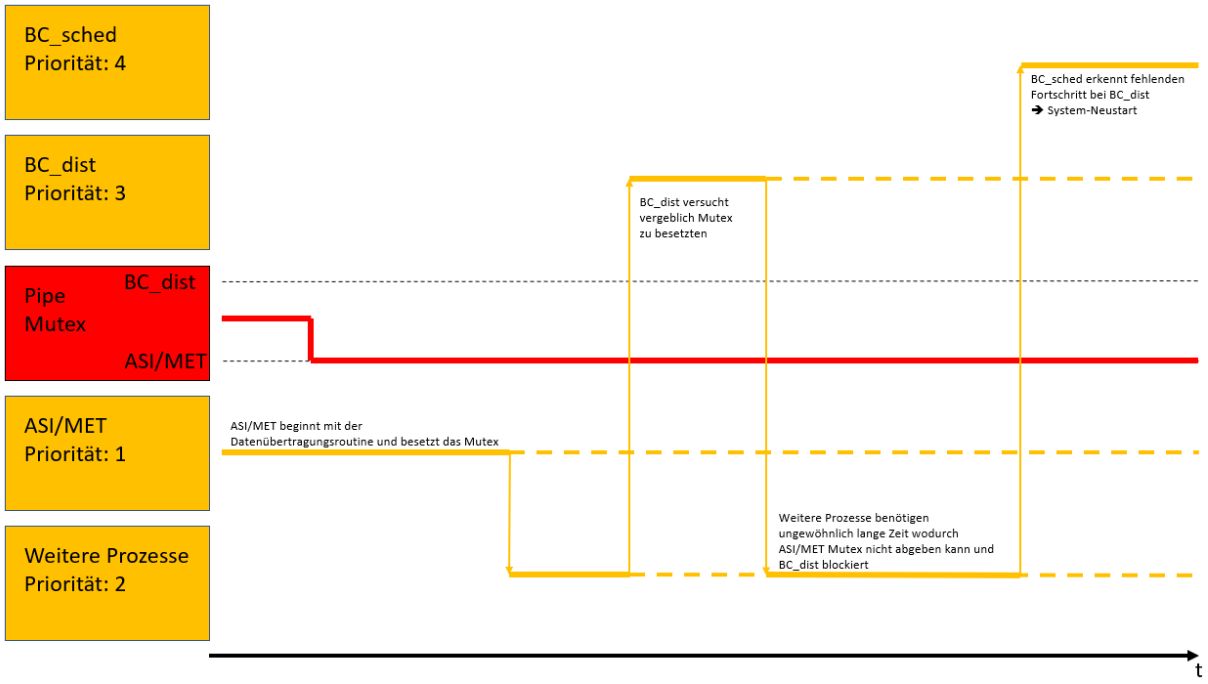


Abbildung 6: Fehlerhafte Busroutine

Inspiration: [4]

Die Prioritätsinversion ist auch beim Mars Pathfinder zu beobachten. Der ASI/MET Prozess sammelt Daten und schreibt diese auf die Pipe. Dafür besetzt er das Mutex. Dieser Prozess wird aber von den weiteren Prozessen unterbrochen, die Rechenzeit beanspruchen. Anschließend will BC_Dist auf die Pipe zugreifen und die Daten verarbeiten. Dieser wird aber unterbrochen, weil das Mutex besetzt ist. Da die weiteren Prozesse noch nicht fertig mit ihren Aufgaben sind, erhalten sie weitere Prozessorzeit. Am Ende einer Routine überprüft BC_Sched den Fortschritt von BC_Dist, merkt, dass es Unregelmäßigkeiten bei BC_Dist gab und setzt deshalb das System zurück.

Analysiert hat man das Problem mit Hilfe von Trace/Log Funktionen und mit Hilfe des Debuggers. Zudem gab es auf der Erde ein baugleiches Duplikat namens Marie Curie. Auch VxWorks selbst hatte einen speziellen Analysemodus, bei dem es möglich war, Prozesswechsel und Interrupts anzuzeigen [2]. Diese Tools zum Fehlererkennen sind typisch für die oben genannte „faster/better/cheaper“-Strategie, indem man gezielt kleine, niedrig priorisierte Fehler in Kauf nimmt, weil man sie im Nachhinein mit wenig Aufwand beseitigen kann [3].

4.3 Prioritätsvererbung

Die Prioritätsvererbung bietet eine einfache Lösung für das Problem. Sie ist leicht implementierbar, verhindert unbegrenzte Prioritätsinversion und die Dauer der Blockade entspricht der Dauer des kritischen Abschnitts. Deadlocks sind jedoch nicht ausgeschlossen [5].

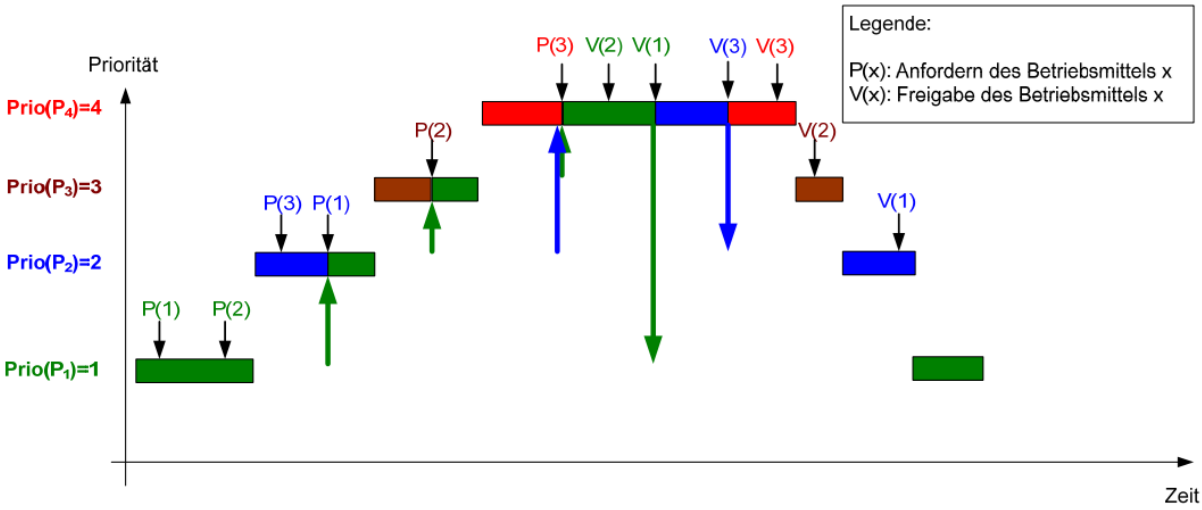


Abbildung 7: Routine einer Prioritätsvererbung

Quelle: [5]

In Abbildung 7 startet der grüne Prozess und besetzt Mutex 1 und 2. Grün wird von Blau unterbrochen und Blau besetzt Mutex 3 und würde gerne Mutex 1 besetzten. Jetzt wird die Priorität von Blau an Grün vererbt. Grün kann mit erhöhter Priorität weiterarbeiten. Braun unterbricht Grün und versucht Mutex 2 zu besetzen. Das hat zur Folge, dass Grün auf Priorität 3 hochgestuft wird. Rot unterbricht Grün und fordert Mutex 3 an. Da Mutex 3 von Blau besetzt wurde, wird Blau hochgestuft. Blau wartet auf Grün und somit wird auch Grün auf Priorität 4 hochgestuft und darf als erstes arbeiten. Grün verliert seine vererbte Priorität, nachdem es beide Mutexe freigegeben hat. Als nächstes beendet sich Blau, gibt auch das Mutex ab und wird wieder runtergestuft. Somit kann Rot seine Routine beenden und alle anderen Prozesse ebenso.

Zusammengefasst wird also eine Priorität vererbt, wenn ein höher priorisierter Prozess ein Betriebsmittel anfordert, das von einem niedriger priorisierten Prozess besetzt wird. Der niedriger priorisierte Prozess gibt die Priorität ab, sobald das Betriebsmittel frei wird. Bei der Verwendung von Prioritätsvererbung gab es aber auch durchaus problematische Szenarien, die im Folgenden genauer betrachtet werden [5].

4.3.1 Mehrere Ressourcen-Problem

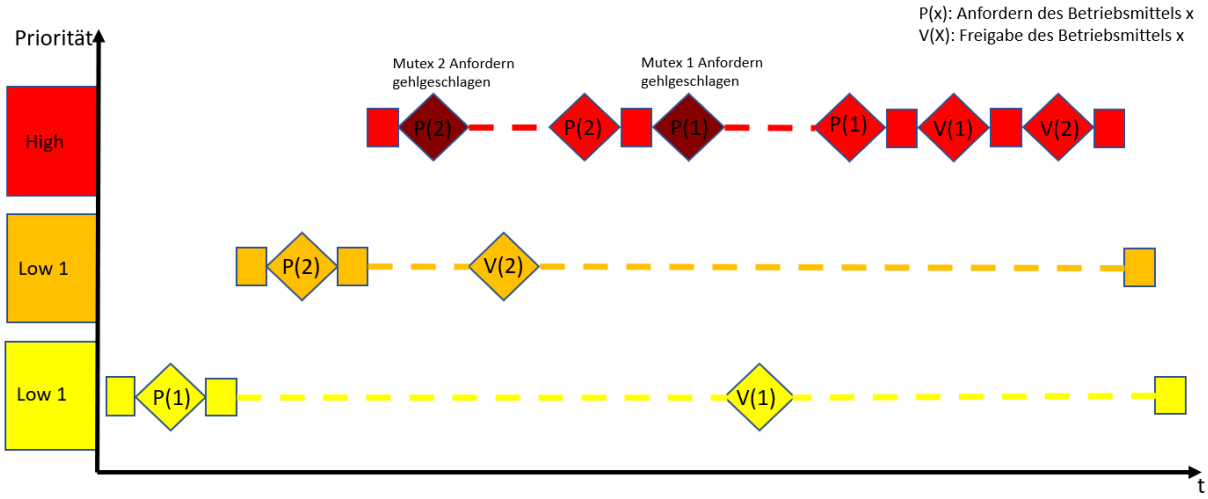


Abbildung 8: Mehrere Ressourcen-Problem

Inspiration: [6]

Dieses Problem ist nicht kritisch, zeigt jedoch sehr gut die Grenzen dieses Lösungsansatzes auf. Gegeben sind nun zwei niedriger priorisierte Prozesse L1 und L2 und ein hoch priorisierter Prozess H (siehe Abbildung 8). Zudem hat das System zwei Betriebsmittel 1 und 2. L1 startet anfangs und besetzt Mutex 1. L2 unterbricht dann L1 und besetzt das zweite Mutex. H erhält im Anschluss Prozessorzeit und fordert erst Mutex 2 und anschließend Mutex 1 an. Dies hat zur Folge, dass der eigentlich deutlich höher priorisierte Prozess H erst auf die anderen beiden Prozesse warten muss, bis diese die Betriebsmittel abgegeben haben. Erst dann kann H seine Routine beenden. Da der Mars Pathfinder nur eine Ressource besitzt ist das nicht so problematisch. Dennoch ist zu berücksichtigen, dass BC_Dist ausgebremst werden kann.

4.3.2 Deadlock-Problem

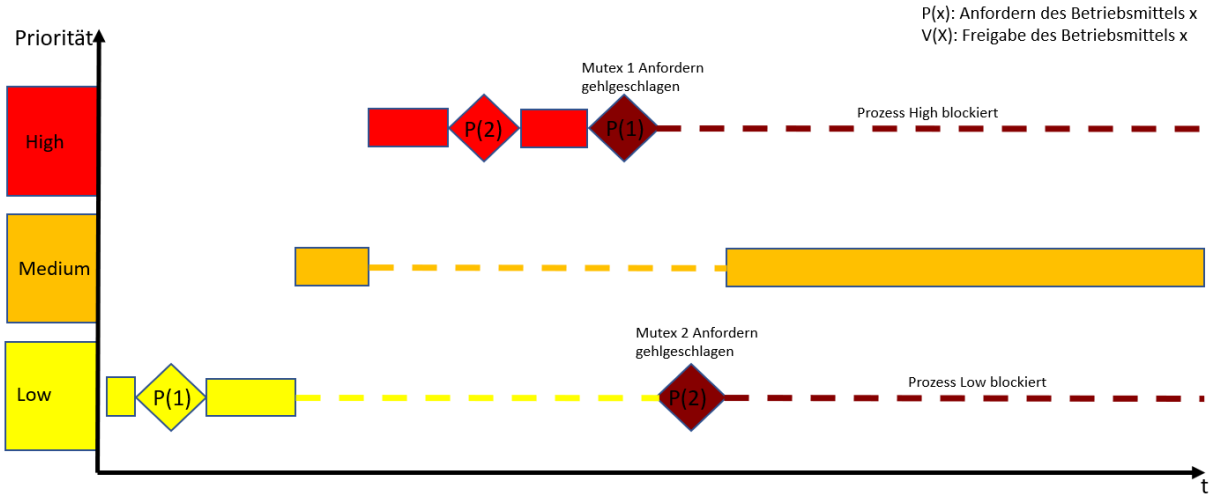


Abbildung 9: Deadlock-Problem

Inspiration: [6]

Deutlich gravierender ist das Deadlock-Problem, das einen Systemabsturz oder zwei sich blockierende Prozesse zur Folge haben kann. Dabei gibt es auch hier drei verschieden priorisierte Prozesse (High, Medium und Low) und zwei Mutexe. L startet auch hier und besetzt Mutex 1. Daraufhin unterbricht M und arbeitet bis er von H unterbrochen wird. H besetzt nun Mutex 2 und will als nächstes Mutex 1 besetzten. Dies hat zur Folge, dass L auf die Priorität von H hochgestuft wird und arbeiten darf. L will in seiner Routine als nächstes Mutex 2 akquirieren, wird aber auch schlafen gelegt, weil dieses von H belegt ist. Das Resultat ist, dass L Mutex 1 hat und auf Mutex 2 wartet, während H Mutex 2 hat und auf Mutex 1 wartet. Ein klassischer Deadlock ist entstanden. Entweder das System ist klug genug, erkennt den Deadlock und führt eine Routine zum Beheben des Fehlers durch, oder H und L sind für die Dauer des Programms blockiert und nur M kann noch weiterarbeiten.

4.4 Prioritätsobergrenzen (Priority Ceiling)

Ein etwas komplexerer Ansatz zum Lösen der Prioritätsinversion bietet das Verfahren mit Prioritätsobergrenzen. Das Verfahren blockiert Prozesse nur für die Dauer des kritischen Abschnittes und verhindert zudem Deadlocks [7]. Jedoch wurde es einerseits wegen seiner Komplexität nicht angewandt und andererseits, da mit einem Mutex keine Verklemmung auftreten kann (siehe 4.3.2). Bei Prioritätsgrenzen erhält jedes Betriebsmittel s (z.B. Semaphoren oder Mutexe) eine Prioritätsgrenze ceil(s). Diese Prioritätsgrenze entspricht der maximalen Priorität der Prozesse, die die Semaphore s besetzen können. Die Routine sieht laut Prof. Dr. Alois Knoll und Dr. Christian Buckl (Vorlesung „Echtzeitsysteme“) wie folgt aus:

1. *„Ein Prozess p darf ein BM nur blockieren, wenn er von keinem anderen Prozess, der andere BM besitzt, verzögert werden kann.*
2. *Die aktuelle Prioritätsgrenze für Prozess p ist aktceil = max{ ceil(s) | s ∈ locked} mit locked = der Menge aller von anderen Prozessen blockierten BM.*
3. *Prozess p darf Betriebsmittel s benutzen, wenn für seine aktuelle Priorität aktprio gilt: aktprio(p) > aktceil(p).*
4. *Andernfalls gibt es genau einen Prozess, der s besitzt. Die Priorität dieses Prozesses wird auf aktprio gesetzt.“* [7]

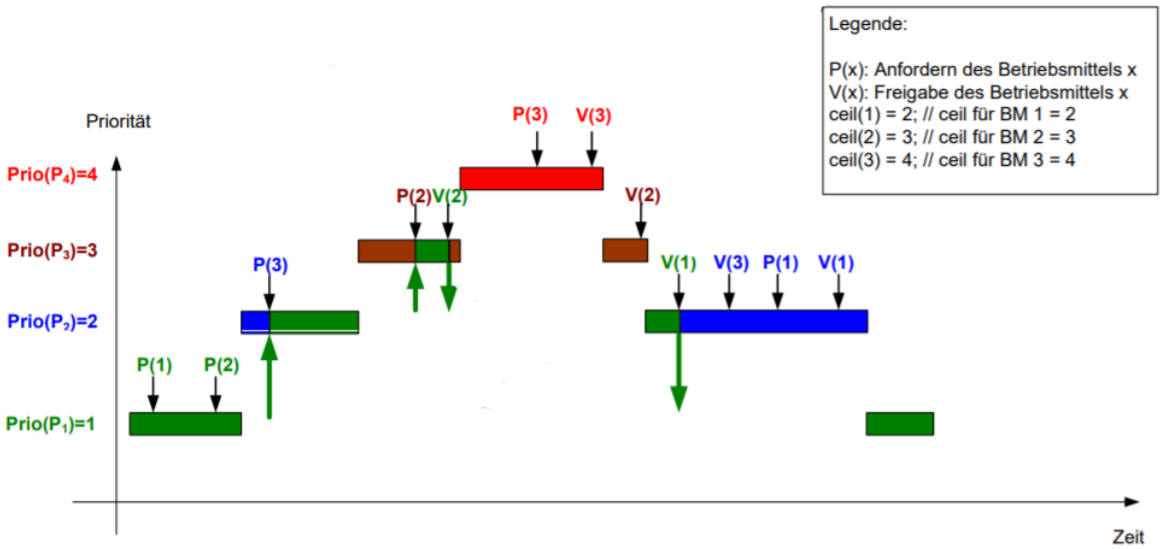


Abbildung 10: Prozessverhalten mit Prioritätsobergrenzen für Betriebsmittel

Quelle: [7]

Priority Ceiling wäre zwar das sicherere Verfahren, jedoch nicht rentabel gewesen, was den Aufwand für die Umsetzung angeht. Des Weiteren gibt es das sogenannte Immediate Priority Ceiling [8], bei dem Prozesse direkt die Priorität ceil(s) zugewiesen bekommen, worauf in dieser Arbeit aber nicht genauer eingegangen wird [5].

Im nachfolgenden Kapitel 5 wird analysiert, wie der Fehler behoben wurde und welche Auswirkungen der Fehler auf die Mission hatte.

5. Umgang mit dem Bug

5.1 Fehlerbehebung

Nachdem der Fehler durch die Debugging-/Log-Tools gefunden werden konnte, wurde die Prioritätsvererbung umgesetzt. Man hatte den Mechanismus bereits implementiert, jedoch deaktiviert, weil man den Prozess BC_Dist nicht verzögern wollte. Die Prioritätsvererbung wurde durch globale, boolesche Variablen blockiert. VxWorks arbeitet mit einem C-Language Interpreter, der es erlaubte, während des Debuggings C Expressions und Funktionen auszuführen. Zudem waren die Adressen der Variablen in Symboltabellen gespeichert, die für den Interpreter zugänglich waren. Letztendlich wurde ein C Programm hochgeladen, welches die Variablen auf True setze und somit die Prioritätsvererbung aktivierte [1] [2].

5.2 Auswirkungen

Der Bug trat vier Mal zwischen dem 5. und 14. Juli 1997 auf, bis er behoben wurde. Glücklicherweise wurden nur Systemneustarts erzwungen. Dies hatte zur Folge, dass die wissenschaftlichen Daten, die sich im Moment auf der Pipe befanden, verloren gingen. Die Daten wurden neu erhoben, was lediglich zu einer Zeitverzögerung geführt hat. Sonst gab es keinerlei andere Auswirkungen durch den Fehler [1].

5.3 Fazit

Dieser Bug zeigt sehr gut, warum White Box Testing essentiell für die Softwareentwicklung und Fehlerbehebung ist. Ausschließlich über Ein- und Ausgaben wäre es unmöglich gewesen, den Fehler zu identifizieren. Zudem hatte man nicht mit der riesigen Menge an Daten gerechnet und dass die weiteren Prozesse so viel Zeit in Anspruch nehmen und somit BC_Dist verzögern würden. Diese Szenarien konnten in der Testphase auf der Erde nicht realitätsgetreu rekonstruiert werden [2].

6. Zusammenfassung

Der Mars Pathfinder ist ein gutes Beispiel für einen positiven Umgang mit Fehlern und der Berücksichtigung von Fehlersituationen in der Softwareentwicklung. Obwohl das Team mit der „faster/better/cheaper“-Strategie unter enormem Zeitdruck stand und auch finanziell eingeschränkt war, wurde von Anfang an viel Wert auf das Testen gelegt. Außerdem wurde auch darauf geachtet, dass man nicht erkannte Fehler durch entsprechende Hilfsmittel zu einem späteren Zeitpunkt noch erkennen und beheben kann. Auch wenn der Fehler nicht gravierend war, wurde er innerhalb weniger Stunden gefunden, analysiert und anschließend behoben. Dieses Fehlermanagement hatte unter anderem zur Folge, dass der Pathfinder die dreimonatige Mission erfüllen konnte und darüber hinaus mehr als das zehnfache der geplanten Zeit funktionsfähig war. Durch das Vorhaben konnten wertvolle Daten erhoben und wichtige Erfahrungen für zukünftige Marsmissionen gesammelt werden [1] [2] [3].

7. Ausblick

Der Mars ist im Moment so interessant, wie nie zuvor. Dabei spielt nicht nur die Wissenschaft eine Rolle, sondern die Länder wollen durch ihren technischen Fortschritt auch Macht demonstrieren. Die USA starteten 2020 mit dem Perseverance Rover und der Drohne Ingenuity eine erneute Mars-Mission. Der Rover basiert auf der Technologie des Curiosity Rovers und sucht primär nach Spuren von Leben. Zudem sollen Proberückholungen zur Erde vorbereitet werden. Außerdem gab es bereits die ersten erfolgreichen Flugtests mit der Drohne. Der chinesische Rover Zhurong ist vor kurzem auf dem Mars gelandet und soll ebenfalls nach Leben suchen. Aber auch kleinere Nationen wie Japan planen derzeit eine Marsmission, in der sie von Marsmonden Proben entnehmen wollen, um diese anschließend auf der Erde auszuwerten [9] [10] [11].

8. Literaturverzeichnis

[1] T. Huckle und N. Tobias, Bits and Bugs: A Scientific and Historical Review on Software Failures in Computational Science, Society for Industrial and Applied Mathematics, 2019.

[2] M. Jones, „What really happened on Mars Rover Pathfinder," 1997.

[3] B. Murihead und P. Pritchett, „The Mars Pathfinder: Approach to "Faster-Better'Cheaper"," Pritchett & Associates, 1998.

[4] P. Sändig, „Mars Pathfinder: Priority Inversion," https://balancer.bbb.rbg.tum.de/playback/presentation/2.3/8cc9316d99973cf94aca20f79fd75848670ad7e7-1590687823450?meetingId=8cc9316d99973cf94aca20f79fd75848670ad7e7-1590687823450, 2020.

[5] K. Prof. Dr. Alois und B. Dr. Christian, „Scheduling-Echtzeitsysteme," http://archive.www6.in.tum.de/www6/pub/Main/TeachingWs2010Echtzeitsysteme/echtzeit_20110118.pdf, 2010.

[6] K. Florian, S. Alex und M. Marcel, „Softwarefehler in der Raumfahrt," https://www5.in.tum.de/lehre/seminare/semsoft/unterlagen_02/soj1/website/index.htm, 2002.

[7] K. Prof. Dr. Alois und B. Dr. Christian, „Prioritätsobergrenzen (priority ceiling) - Echtzeitsysteme," http://archive.www6.in.tum.de/www6/pub/Main/TeachingWs2010Echtzeitsysteme/echtzeit_20110119.pdf, 2010.

[8] B. Dr. Christian, „Immediate Priority Ceiling-Echtzeitsysteme," http://archive.www6.in.tum.de/www6/pub/Main/TeachingWs20123chtzeitsysteme/Echtzeit-Gesamt.pdf, 2013.

[9] Z. Abbany, „Japan's Mars moons mission leads to human spaceflight," https://www.dw.com/en/japans-mars-moons-mission-leads-to-human-spaceflight/a-54961901, 2020.

[10] J.-C. Hanika, H. Westram und O. Huber, „Perseverance und Ingenuity auf dem Mars," https://www.br.de/wissen/perseverance-ingenuity-mars-rover-hubschrauber-nasa-100.html, 2021.

[11] A. Jones, „China is about to try a high-stakes landing on Mars," https://www.nationalgeographic.co.uk/space/2021/05/china-is-about-to-try-a-high-stakes-landing-on-mars, 2021.