IPC—Message–Queue

Spezifikation

Hans–Jürgen Ficker, Sönke Jarré

Universität Bremen, 9. September 1999

Inhaltsverzeichnis

1 Einleitung

Diese Spezifikation entstand im Rahmen des studentischen Projektes LiVE!, das vom WS 1997/1998 bis SS 1999 an der Universität Bremen durchgeführt wurde. Der Sinn und Inhalt des Projektes ist am besten unter `http://aerobee.informatik.uni-bremen.de/start.html` nachzulesen. Da dieses Dokument im Rahmen des Projektes LiVE! entstand, findet es sich auch im Anhang an den Projektbericht wieder.

In unserer Teilgruppe von LiVE!, nach dem Inter-Process-Communication Paket IPC genannt, beschäftigen wir uns mit der Spezifikation und Verifikation der Message-Queues in Linux.

Zuerst sollten wir kurz erklären, warum wir uns überhaupt mit Linux und dem IPC-Package beschäftigen, ohne zu sehr auf LiVE! einzugehen. Linux ist ein modernes Betriebssystem, das zudem immer mehr in Mode kommt. Es hat den großen Vorteil, daß nicht nur der Source-Code vorliegt sondern es unter der Open-Source-Licens steht und somit der Source-Code auch beliebig abwandelbar ist. Der immer größeren Anhängerschar von Linux wollen wir uns anschließen, ebenfalls am Code herumbasteln und einen sinnvollen Beitrag für die Linux-Gemeinde leisten.

Entsprechend den Zielen des Projektes LiVE! wollten wir in unserem Teilprojekt einen Teil von Linux verifizieren. Hierfür benöigten wir einen Abschnitt aus Linux, der zum einen nicht zu lang ist, schon über einen längeren Zeitraum nicht mehr verändert worden ist, zudem weiterhin benötigt wird und auch nicht mehr großen Veränderungen unterliegt.

Hier sind wir im IPC-Package mit den Message-Queues fündig geworden. Der Message-Queue Source-Code ist 494 Zeilen lang (entspricht 11598 Bytes), und ist zuletzt am 20.11.1998 verändert worden. Die Kommunikation zwischen den einzelnen Prozessen wird auch in Zukunft sicher benötigt, zudem arbeiten viele Programme mit den Message-Queues und gehen von einem korrekten Funktionieren aus.

Im normalen Software-Entwicklungsprozeß sollte erst eine Spezifikation erstellt werden und anschließend entsprechend implementiert werden. Bei vielen Open-Source-Projekten verläuft dies jedoch anders, es wird implementiert und bestehende Fehler werden durch ausprobieren gefunden und behoben, eine Spezifikation findet niemals statt.

Die Message-Queues sind also bisher nicht formal spezifiziert. Es gibt den Source-Code, in dem einige spärliche Kommentare zum besseren Verständnis stehen, die online-Hilfe, vor allem die man-pages, die eine informelle Beschreibung versuchen, sowie einige Bücher, die eine teilweise Beschreibung der Funktionsweise der Message-Queues geben.

Entgegen dem normalen Software-Entwicklungsprozeß verwenden wir zur

Erstellung der formalen Spezifikation eine informelle Spezifikation und in Zweifelsfällen den Source-Code.

An verschiedenen Stellen waren wir uns über das Systemverhalten nicht ganz im klaren. Um das tatsächliche Verhalten der Message-Queues auszutesten schrieben wir kleine Testprogramme, die an der Linux-Implementierung angewendet worden.

Für die Spezifikation haben wir als Spezifikationssprache Z ausgewählt. Message-Queues könnte man als Kernel-Datenstrukturen bezeichnen. Um solche Datenstrukturen sowie deren Veränderungen am besten darzustellen erschien uns eine mengenbasierte Spezifikationssprache, wie Z eine ist, am besten geeignet zu sein.

Wir wollen mit unserer Spezifikation keine Einführung in die Sprache Z geben. Diejenigen Leser, die sich so etwas noch wünschen, seien auf die entsprechende Literatur verwiesen (...).

Als wir mit der Spezifikation anfingen beschränkten wir uns vorerst auf einige wesentliche Teile. Trotzdem fabrizierten wir auch hier schon Fehler, die erst ersichtlich wurden, als wir die restlichen Teile nach und nach ergänzten. Diese Fehler ließen sich teilweise leicht entfernen, bei einigen mußten wir aber auch die gesamte Spezifikation überarbeiten und umschreiben.

Insgesamt wurde die Spezifikation viel aufwendiger und langwieriger, als zuerst gedacht. Und natürlich ist sie inzwischen wesentlich komplexer als die kleinen konstruierten Beispiele aus den Lehrbüchern.

Unsere Spezifikation findet sich wie folgt in diesem Dokument wieder:

Im Kapitel 2 führen wir die für die Spezifikation benötigten Datenstrukturen ein. Wir widmen uns erst den benötigten Basistypen, anschließend wenden wir uns den Konstanten zu bevor wir die verwendeten Basisfunktionen einführen.

Anschließend, im Kapitel 3, widmen wir uns dem Zustandsraum, auf denen die Message-Queues arbeiten. Wir erläutern die dort verwendeten Objekte und erklären auch die Konsistenzbedingungen, die immer gelten müssen.

Das Kapitel 4, mit der Systeminitialisierung, ist ziemlich kurz. Hier versuchen wir auch den mit der Spezifikationssprache Z nicht so bewandetem Leser einige grundlegende Dinge kurz näher zu bringen.

Es gibt insgesamt vier verschiedene Message-Queue-Operationen, denen wir jeweils ein eigenes Kapitel widmen (Kapitel 5 bis Kapitel 8). Es wird jeweils zuerst der vom Benutzer gewünschte Erfolgsfall der Operation beschrieben, dann werden sämtliche Fehlerfälle aufgeführt. Zum Abschluß wird jeweils das User-Interface erstellt, das der Benutzer zu sehen bekommt.

Im letzten Abschnitt (Kapitel 9) werden wir dann die Ergebnisse vorstellen sowie die Erfahrungen schildern, die wir im Laufe der Zeit erlebt haben.

Inzwischen sind wir zu der Ansicht gelangt, daß die hier vorliegende aktuelle Spezifikation vollständig und korrekt ist. Teile von ihr haben wir auch verifiziert (siehe Projektbericht des Projektes LiVE! oder das Dokument “IPC Message-Queue, Verifikation“). Sollten noch Unklarheiten auftreten, so bitten wir um Benachrichtigung.

2 Datenstrukturen

In diesem Kapitel widmen wir uns zuerst dem Grundlegenden: die Datenstrukturen, mit denen wir später arbeiten.

Sowohl in diesem, als auch in den folgenden Kapiteln, werden wir versuchen für die bessere Verständlichkeit möglichst aussagekräftige Namen zu verwenden. Hierbei haben meist die im Source-Code verwendeten Namen als Vorlage gedient. Allerdings geht an einigen Stellen diese besser Verständlichkeit zu Lasten der Lesbarkeit, da sich lange Wörter mit mathematischen Zeichen gemischt nur schwer lesen lassen. Wir hoffen aber einen guten Kompromis gefunden zu haben.

Unsere formale Spezifikation soll möglichst abstrakt gehalten werden, auch wenn wir den Source-Code teilweise bei der Erstellung hinzugezogen haben. Wir wollen schließlich eine Beschreibung des nach Außen sichtbaren Verhaltens erstellen und keine Wiedergabe interner Abläufe.

Wie das übliche Vorgehen bei einer Z-Spezifikation ist, können wir nicht genau sagen. Die kleinen Beispiele aus den Z-Lehrbüchern konnten dies nicht ganz wiedergeben.

Wir haben bei der Spezifikation mit den grundlegenden Teil der Message-Queues angefangen, entsprechende Datenstrukturen uns erdacht, auf diesen Funktionen definiert und betrachtet, ob somit die Message-Queue-Operationen dargestellt waren. Anschließend haben wir alles verfeinert, mehr Features der Message-Queues hinzugenommen und haben dabei teilweise die Datenstrukturen abwandeln oder ergänzen müssen.

Entsprechend abstrakt sind einige Datenstrukturen und Funktionen, vor allem solche, die Systemdienste oder Systemstrukturen darstellen. Diese könnten jedoch in anderen Teilen des Betriebssystems, in denen sie benutzt oder bereitgestellt werden, genauer definiert werden, jedoch ist es hier nicht nötig.

Dieses Kapitel beginnt mit den Basistypen, die grundlegend vorhanden sein müssen und mit denen, oder auf denen, die Message-Queues arbeiten.

Sodann werden die Konstanten eingeführt, die Systemweit feststehen sollen.

Daran schließen sich die Basisfunktionen an, die im Prinzip jedem Prozeß zur Verfügung stehende Systemdienste darstellen.

2.1 Basistypen

Bei den Basistypen kann man zwei verschieden Arten unterscheiden.

Mit der einen Art wird nicht weiter gerechnet, die einzelnen Elemente müssen nur einem bestimmten Typen entsprechen. Also kann man sie wunderschön abstrakt halten.

Bei der anderen Art sieht es schon schwieriger aus, mit ihnen soll auch gerechnet werden. Dementsprechend muß eine Struktur mit spezifiziert werden, eventuell sogar noch entsprechende Rechenoperationen auf dieser Struktur. Da es sich aber bei der Struktur häufig um Zahlen handelt sind diese Rechenoperationen schon implizit mitgegeben.

In einem System gibt es zuerst einmal mehrere Benutzer. Diese haben eine eindeutige Identifikation, die User-ID aus der Menge UID.

$[\mathit{UID}]$

Da wir uns in UNIX-Systemen aufhalten gibt es für jeden User auch eine zugehörige Group-ID, diesmal aus der Menge GID.

$[\mathit{GID}]$

Ein Benutzer (user), eine Gruppe (group), oder „die ganze Welt" (other) können das Recht bekommen, auf eine Message-Queue zu schreiben, beziehungsweise von ihr zu lesen. Diese Rechte sind ähnlich den UNIX-Dateirechten, nur das execute-Bit hat keine Auswirkungen, und wurde deshalb auch weggelassen. Man kann es im realen System sogar setzen, nur hat es keinerlei Auswirkungen.

$\mathit{PERM} ::= \mathit{u\_write} \mid \mathit{u\_read} \mid \mathit{g\_write} \mid \mathit{g\_read} \mid \mathit{o\_write} \mid \mathit{o\_read}$

Ein Benutzer startet Prozesse. Diese müssen teilweise miteinander oder mit anderen Prozessen kommunizieren. Jeder Prozeß hat daher systemweit eine eindeutige Identifikation, den Prozeßidentifikator.

$[\mathit{PID}]$

Für die Kommunikation zwischen Prozessen können Message-Queues verwendet werden. Damit verschiedene Prozesse über eine Message-Queue miteinander kommunizieren können, muß dieselbe Message-Queue adressiert werden. Dies geschieht über den KEY.

$\mathit{KEY} : \mathbb{P}(\mathbb{N} \cup \{0\})$

Für die einzelnen Operationen wird die Message-Queue jedoch über IDs adressiert.

$$ID : \mathbb{P}(\mathbb{N} \cup \{0\})$$

In einer Message-Queue werden Messages gesendet. Diese bestehen aus einer Folge von Bytes und einem Typ für die Message.

Da die Strktur der Folgen von Bytes nicht weiter relevant ist, nehmen wir eine Folge von Bytes als einen abstrakten Datentypen. Dies ist ein schönes Beispiel für die Abstraktion, denn eigentlich hätte man hier einzelne Bytes als Datentypen nehmen müssen, die sich wiederum aus Bits zusammensetzen.

$$[BYTES]$$

Den Typ einer Nachricht wird in Form einer ganzen Zahl angegeben. Zu beachten ist, daß negative Zahlen zwar nicht als Typ einer Nachricht erlaubt sind, sie aber bei der Message-Queue Operation *msgrcv* eine spezielle Bedeutung haben.

$$TYPE : \mathbb{P}\,\mathbb{Z}$$

Mit einer Message-Queue können verschiedene Operationen ausgeführt werden, wie sie in den folgenden Kapiteln noch ausführlich beschrieben werden.

Bei den Operationen *msgrcv* und *msgget* können dabei einige Flags als Option mitgegeben werden.

$$\begin{aligned} FLAG ::= {} & MSG\_EXCEPT \mid IPC\_NOWAIT \mid MSG\_NOERROR \mid \\ & IPC\_CREAT \mid IPC\_EXCL \end{aligned}$$

Der Operation *msgctl* können dagegen bestimmte Kommandos als Argumente mitgegeben werden.

$$CMD ::= IPC\_STAT \mid IPC\_SET \mid IPC\_RMID$$

Message-Queue Operationen können eine Anzahl von Fehlern verursachen:

$$\begin{aligned} ERROR ::= {} & EACCES \mid EAGAIN \mid EEXIST \mid EINVAL \mid ENOENT \mid \\ & ENOMSG \mid E2BIG \mid ENOSPC \mid EPERM \mid NOERROR \end{aligned}$$

Und als Abschluß kann einem Prozeß als Return-Wert mitgeteilt werden, ob ein Fehler vorliegt (Returnwert -1) oder es eine erfolgreiche Operation war (Returnwert 0 oder eine positive ganze Zahl).

$$RETURN : \mathbb{P}(\mathbb{N} \cup \{0, -1\})$$

2.2 Konstanten

Es gibt eine Reihe von Konstanten, die einmalig systemweit festgelegt sind. Diese sind insoweit konstant, als daß sich Änderungen erst bei einem Neustart auswirken.

Die wirklich reale Größe, also die Definition der einzelnen Konstanten, ist an dieser Stelle nicht wichtig und sollte an anderer Stelle im System erfolgen.

Um eine mißbräuchliche Nutzung des Systems zu verhindern bzw. um an keine Systemgrenzen zu stoßen gibt es gewisse Begrenzungen für die Message-Queues.

In einem System sollte es eine Obergrenze für die Anzahl der Message-Queues geben. Daher gibt es eine maximale Anzahl von Message-Queues.

$$MSGMNI : \mathbb{N}$$

Damit eine Message-Queue nicht beliebig voll geschrieben werden kann, gibt es eine maximale Länge einer Message-Queue:

$$MSGMNB : \mathbb{N}$$

Auch Nachrichten sollten eine gewisse Länge nicht überschreiten.

$$MSGMAX : \mathbb{N}$$

Zudem gibt es einige spezielle Elemente aus den im vorherigen Abschnitt deklarierten Basistypen die für einige Operationen benannt sein müssen.

So wird für *msgget* ein spezielles Element aus KEY benötigt, um eine neue Message-Queue mit einem beliebigen KEY zu erzeugen.

$$IPC\_PRIVATE : KEY$$

Auch wird ein spezielles Element aus UID benötigt, daß den Superuser (root) kennzeichnet:

$$SUPERUSER : UID$$

2.3 Basisfunktionen

Im Folgendem werden einige Basisfunktionen eingeführt, die Systemdienste beschreiben, die den einzelnen Prozessen jeweils zur Verfügung stehen.

Einige dieser Systemdienste kommen aus anderen Bereichen des Betriebsystems und werden von uns nicht näher spezifiziert. Dieses müßte bei der Spezifikation dieser Bereiche geschehen.

Es werden zuerst einmal einige Funktionen benötigt, die sich um die Rechtevergabe kümmern.

So gibt es eine Funktion, die zu einer *UID* alle *GID*s der Gruppen zuordnet, in denen der jeweilige User ist:

$$getallgroup : UID \rightarrow \mathbb{P}\, GID$$

Auch gibt es Funktionen, die einer *PID* die *UID* bzw. die *GID* des Prozesses zuordnet. Dieses sind partielle Funktionen, da nur verwendeten *PID* ein Wert zugeordnet ist.

$$\begin{array}{l} getuid : PID \nrightarrow UID \\ getgid : PID \nrightarrow GID \end{array}$$

Zum Lesen und Schreiben auf einer Message-Queue werden ebenfalls Funktionen benötigt.

So ermittelt eine allgemeinen Funktion, ob ein User auf eine Message-Queue schreiben darf, wobei diese Message-Queue bestimmte Rechte, eine bestimmte *UID* und eine bestimmte *GID* hat.

$$\_\ \mathrm{maywrite}\ \_ : UID \leftrightarrow (\mathbb{P}\, PERM \times UID \times GID)$$

$$\begin{array}{l} \forall\, u1, u2 : UID;\ p : \mathbb{P}\, PERM;\ g : GID \bullet \\ u1\ \mathrm{maywrite}\ (p, u2, g) \Leftrightarrow (u1 = u2 \land u\_write \in p) \lor \\ \qquad (g \in getallgroup(u1) \land g\_write \in p) \lor \\ \qquad o\_write \in p \end{array}$$

Eine andere allgemeinen Funktion ermittelt, ob ein User von einer Message-Queue lesen darf, welche bestimmte Rechte, eine bestimmte *UID* und eine bestimmte *GID* hat.

$$\_\ \mathrm{mayread}\ \_ : UID \leftrightarrow (\mathbb{P}\, PERM \times UID \times GID)$$

$$\begin{array}{l} \forall\, u1, u2 : UID;\ p : \mathbb{P}\, PERM;\ g : GID \bullet \\ u1\ \mathrm{mayread}\ (p, u2, g) \Leftrightarrow (u1 = u2 \land u\_read \in p) \lor \\ \qquad (g \in getallgroup(u1) \land g\_read \in p) \lor \\ \qquad o\_read \in p \end{array}$$

Betrachten wir eine einzelne Message, so benötigen wir noch eine Funktion, die nur das Anfangsstück der Länge n der $BYTES$ zurückgibt. Da wir $BYTES$ sehr abstrakt gehalten haben können wir an dieser Stelle diese Funktion nicht näher spezifizieren.

$$truncate : BYTES \times \mathbb{N} \rightarrow BYTES$$

3 Message-Queue Zustandsraum

Nach unserer Sichtweise sind Message-Queues Kernel-Datenstrukturen. In diesem Kapitel geht es nun darum, den Zustandsraum, in dem sich diese Datenstrukturen befinden, zu beschreiben.

Hierfür gibt es einige Objekte, die den Zustand der Menge der Message-Queues charakterisieren. Für diese gelten gewisse Konsistenzbedingungen.

Um nicht zu sehr von der gegebenen Implementierung zu abstrahieren haben wir die im Source-Code vorhandenen Strukturen teilweise übernommen.

Insgesamt denken wir mit folgenden zehn Objekten den Zustandsraum hinreichend charakterisieren zu können:

- *getkey* gibt zu jeder *ID* einen eindeutigen *KEY* zurück.
- *msgq* weist einer *ID* eine Message-Queue zu.
- *creator* gibt den Erzeuger einer Message-Queue an.
- *owner* gibt den Besitzer einer Message-Queue an.
- *group* gibt die Gruppe einer Message-Queue an.
- *perm* zeigt die Rechte an, die ein User für die Message-Queue besitzt.
- *qbytes* liefert die maximale Länge einer Message-Queue.
- *cbytes* gibt die Länge einer Message-Queue wieder.
- *msg_ts* gibt für jede Nachricht aus einer Message-Queue die Länge dieser Nachricht an.
- *errno* gibt die Art des Fehlers aus.

Diese zehn Objekte, ihr aktuelle Zustand und ihr Veränderungen, sind von nun an bei allen Schemata von Bedeutung.

Die Abstraktion des Zustandraumes für die Kommunikation mit Hilfe von Message-Queues sieht nun folgendermaßen aus:

$$
\begin{array}{|l}
\hline
\multicolumn{1}{l}{\mathit{MsgQStateSpace}} \\
\mathit{getkey} : \mathit{ID} \pfun \mathit{KEY} \\
\mathit{msgq} : \mathit{ID} \pfun \mathrm{seq}(\mathit{TYPE} \times \mathit{BYTES}) \\
\mathit{creator} : \mathit{ID} \pfun \mathit{UID} \\
\mathit{owner} : \mathit{ID} \pfun \mathit{UID} \\
\mathit{group} : \mathit{ID} \pfun \mathit{GID} \\
\mathit{perm} : \mathit{ID} \pfun \mathbb{P}\,\mathit{PERM} \\
\mathit{qbytes} : \mathit{ID} \pfun \mathbb{N} \\
\mathit{cbytes} : \mathit{ID} \pfun \mathbb{N} \\
\mathit{msg\_ts} : \mathit{ID} \pfun \mathrm{seq}\,\mathbb{N} \\
\mathit{errno} : \mathit{ERROR} \\
\hline
\#\,\mathrm{dom}\,\mathit{getkey} \leq \mathit{MSGMNI} \quad (1) \\
\mathrm{dom}\,\mathit{msgq} = \mathrm{dom}\,\mathit{getkey} \quad (2) \\
\mathrm{dom}\,\mathit{msgq} = \mathrm{dom}\,\mathit{creator} \quad (3) \\
\mathrm{dom}\,\mathit{msgq} = \mathrm{dom}\,\mathit{owner} \quad (4) \\
\mathrm{dom}\,\mathit{msgq} = \mathrm{dom}\,\mathit{group} \quad (5) \\
\mathrm{dom}\,\mathit{msgq} = \mathrm{dom}\,\mathit{perm} \quad (6) \\
\mathrm{dom}\,\mathit{msgq} = \mathrm{dom}\,\mathit{qbytes} \quad (7) \\
\mathrm{dom}\,\mathit{msgq} = \mathrm{dom}\,\mathit{cbytes} \quad (8) \\
\mathrm{dom}\,\mathit{msgq} = \mathrm{dom}\,\mathit{msg\_ts} \quad (9) \\
\forall\, i : \mathit{ID} \mid i \in \mathrm{dom}\,\mathit{msgq} \bullet \mathrm{dom}(\mathit{msgq}(i)) = \mathrm{dom}(\mathit{msg\_ts}(i)) \quad (10) \\
\forall\, i1, i2 : \mathit{ID} \mid i1 \in \mathrm{dom}\,\mathit{msgq} \land i2 \in \mathrm{dom}\,\mathit{msgq} \bullet \quad (11) \\
\qquad \mathit{getkey}(i1) = \mathit{getkey}(i2) \Rightarrow i1 = i2 \lor \mathit{getkey}(i1) = \mathit{IPC\_PRIVATE} \\
\hline
\end{array}
$$

Dabei haben die in der unteren Hälfte aufgestellten Konsistenzbedingungen folgende Bedeutung:

- (1): Die Anzahl der Message-Queues im System dürfen nicht mehr sein als im System insgesamt mit *MSGMNI* erlaubt ist.
- (2) bis (9): Wenn es für eine bestimmte *ID* eine Message-Queue gibt, dann gibt es für diese *ID* auch *creator*, *owner*, *group*, *perm*, *qbytes*, *cbytes* und *msg_ts*.
- (10): Jeder Nachricht in einer Message-Queue wird auch eine Länge zugeordnet. Dies bedeutet, daß wenn einer *ID* eine Message-Queue zugeordnet ist, *msg_ts* diese *ID* auf eine Sequenz von natürlichen Zahlen abbildet, die jeweils die Länge der einzelnen Messages beschreiben.

- (11): Für alle Elemente aus KEY, außer für $IPC\_PRIVATE$, gilt, daß er immer eindeutig ist. Dies bedeutet, daß niemals zwei Message-Queues über ein und denselben KEY angesprochen werden können, mit der einzigen Ausnahme, $IPC\_PRIVATE$.

4 Systeminitialisierung

Nachdem wir nun den Zustandsraum für die Message-Queues festgelegt haben, geht es in diesem Kapitel nun um die Startbedingungen, sprich die Systeminitialisierung.

Beim Start des Betriebssystems gibt es noch keine Prozesse, diese müssen erst noch erzeugt werden. Daher existieren auch noch keine Message-Queues, diese werden erst später initialisiert.

Entsprechend einfach stellt sich somit die Initialisierung dar: Da noch keine Message-Queues existent sind, ist der Domain der benutzten Funktionen leer. Nur die Funktion, die auf bestimmte Werte abgebildet wird, ist auf den Default-Wert gesetzt.

Init

$MsgQStateSpace'$

$$\begin{array}{l}
\mathrm{dom}\, getkey' = \emptyset \\
\mathrm{dom}\, msgq' = \emptyset \\
\mathrm{dom}\, creator' = \emptyset \\
\mathrm{dom}\, owner' = \emptyset \\
\mathrm{dom}\, group' = \emptyset \\
\mathrm{dom}\, perm' = \emptyset \\
\mathrm{dom}\, qbytes' = \emptyset \\
\mathrm{dom}\, cbytes' = \emptyset \\
\mathrm{dom}\, msg_ts' = \emptyset \\
errno' = NOERROR
\end{array}$$

Für die mit Z noch nicht so bewanderten Leser:

Dieser Kasten ist ein Schema, auf der oberen Linie steht der Name des Schemas, in diesem Falle *Init*. In der oberen Hälfte des Schemas, also über dem Querstrich, werden sämtliche Ein- und Ausgabewerte deklariert sowie andere Schemata eingebunden. Mit einem Quote (') wird ein Zustand, der nach dem Schema gelten muß, gekennzeichnet.

Die untere Hälfte haben wir noch einmal aufgeteilt, zuerst stellen wir immer die Vorbedingungen und Nachbedingungen, die für dieses Schema gelten, dar. In diesem Falle sind keine vorhanden.

Anschließend werden sämtliche Objekte des Zustandsraum und ihre eventuellen Veränderungen aufgezeigt.

5 Message-Queue Operation *msgget*()

Die Operation *msgget*() ermöglicht es einem Prozeß eine Message-Queue für sich zugänglich zu machen.

Dabei Teilen wir die Operation in verschiedene Fälle auf. Dieses hat mehrere Gründe.

Zum einen gibt es die natürliche Aufteilung in den erfolgreichen und den nicht erfolgreichen Fall. Diese beiden Fälle werden aber noch jeweils in mehrere Einzelfälle aufgeteilt.

Hierdurch ist eine bessere Lesbarkeit gegeben und auch das Verständnis wird vereinfacht. Auch werden im Source-Code an diesen Stellen Fallunterscheidungen getroffen und somit ein Aufsplitten quasi vorgegeben.

Es gibt verschiedene Möglichkeiten bei *msgget*() erfolgreich zu sein, je nachdem welche Vorbedingungen vorliegen und welche Nachbedingungen man erreichen will.

Zuerst kann man natürlich eine Message-Queue mit einem bestimmten *KEY* einrichten, wobei der *KEY* bisher noch nicht existieren darf.

Dann ist es möglich sich bei einer Message-Queue, die schon existiert und deren *KEY* einem bekannt ist, sich einzuklinken.

Und zuletzt ist es möglich definitiv eine neue Message-Queue einzurichten, indem man den besonderen *KEY* *IPC_PRIVATE* vorgiebt.

Natürlich ist es auch möglich, daß ein Fehler auftritt. Dies bedeutet nicht, daß im System etwas fehlerhaft implementiert ist, sondern daß der vom Benutzer gewünschte Erfolgsfall nicht möglich ist.

So kann es vorkommen, daß der Benutzer für die Message-Queue keine Zugriffsrechte hat, daß ein *FLAG* falsch oder nicht gesetzt ist oder daß die Systemgrenzen überschritten würden.

In allen diesen Fällen wird keine Message-Queue für den Benutzer zur Verfügung gestellt und er erhält statt dessen eine entsprechende Fehlermeldung.

Insgesamt gibt es zwei Fehlerfälle, die wir nicht in unserer Spezifikation darstellen.

Da wäre die Möglichkeit, daß die Message-Queue zwischenzeitlich gelöscht wurde. Die andere Möglichkeit ist, daß nicht mehr genügend Speicher für den Inhalt der Message-Queue vorhanden ist.

Diese beiden Fälle müssen in anderen Teilen des Betriebsystems abgefangen werden, unsere Spezifikation der Message-Queues ist daher immer noch vollständig.

Für den Benutzer existieren diese Unterscheidungen in verschiedene Fehler- und Erfolgsfälle nicht, sie sind für ihn auch nicht weiter relevant.

Daher gibt es das User-Interface, in dem alle Fälle zusammengeführt werden und das nach Aussen sichtbare Verhalten von *msgget*() simuliert wird.

In diesem Kapitel führen wir zuerst sämtliche Erfolgsfälle auf. Hier versuchen wir die Unterschiede besonders hervorzuheben.

Anschließend stellen wir die Fehlerfälle vor und erklären jeden möglichst ausführlich.

Im letzten Abschnitt erstellen wir das User-Interface, indem wir die einzelnen Schema miteinander kombinieren.

Insgesamt erhalten wir somit eine formale abstrakte Beschreibung der Message-Queue Operation *msgget*() durch das Schema *msgget*

5.1 Erfolgsfall

Fangen wir mit dem erfolgreichem Einrichten einer Message-Queue, die noch nicht existiert, an. Dabei wird ein KEY vorgegeben, der noch nicht an eine andere Message-Queue vergeben ist.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Msgget1}} \\
\hline
\Delta \mathit{MsgQStateSpace} \\
k? : \mathit{KEY} \\
p? : \mathit{PID} \\
i! : \mathit{ID} \\
\mathit{perm}? : \mathbb{P}\, \mathit{PERM} \\
\mathit{flag}? : \mathbb{P}\, \mathit{FLAG} \\
r! : \mathit{RETURN} \\
\hline
k? \notin \operatorname{ran} \mathit{getkey} \land k? \neq \mathit{IPC\_PRIVATE} \\
i! \notin \operatorname{dom} \mathit{msgq} \\
\#(\operatorname{dom} \mathit{msgq}) < \mathit{MSGMNI} \\
\mathit{IPC\_CREAT} \in \mathit{flag}? \\
r! = i! \\
\\
\mathit{getkey}' = \mathit{getkey} \oplus \{i! \mapsto k?\} \\
\mathit{msgq}' = \mathit{msgq} \cup \{i! \mapsto \langle\rangle\} \\
\mathit{creator}' = \mathit{creator} \oplus \{i! \mapsto \mathit{getuid}(p?)\} \\
\mathit{owner}' = \mathit{owner} \oplus \{i! \mapsto \mathit{getuid}(p?)\} \\
\mathit{group}' = \mathit{group} \oplus \{i! \mapsto \mathit{getgid}(p?)\} \\
\mathit{perm}' = \mathit{perm} \oplus \{i! \mapsto \mathit{perm}?\} \\
\mathit{qbytes}' = \mathit{qbytes} \oplus \{i! \mapsto \mathit{MSGMNB}\} \\
\mathit{cbytes}' = \mathit{cbytes} \oplus \{i! \mapsto 0\} \\
\mathit{msg\_ts}' = \mathit{msg\_ts} \oplus \{i! \mapsto \langle\rangle\} \\
\mathit{errno}' = \mathit{errno} \\
\hline
\end{array}
$$

Hierbei wird zuerst der Zustandsraum in das Schema mit eingebunden, auf dem auch Veränderungen ausgeführt werden können.

Um später eine Schema-Komposition aller Einzelfälle von $msgget$ durchführen zu können haben alle diese Fälle dieselben Ein- und Ausgaben, die aus der Aufrufsyntax von $msgget()$ und den Rückgabewerten resultieren.

Interessant wird es erst bei den Vor- und Nachbedingungen, da sich hier die Schemata der einzelnen Fälle unterscheiden.

- Bei $Msgget1$ wird ein KEY als Eingabe verlangt, der bisher keiner Message-Queue zugeordnet ist und der auch nicht $IPC\_PRIVATE$ entspricht.

- Die ID, der diese Message-Queue zugewiesen wird, darf noch nicht vorhanden sein.

- Auch darf die Höchstgrenze der maximal im System zulässigen Message-Queues nicht überschritten werden.

- Als Flag muß IPC_CREAT gesetzt sein.

- Zurückgegeben wird als Returnwert die ID.

Auch die Objekte des Zustandsraumes unterliegen gewissen Änderungen, wobei die Konsistenzbedingungen des Zustandraumes natürlich weiterhin gelten müssen.

$errno$ verändert sich nicht. Bei allen anderen Objekten wird die durch die neue ID gekennzeichnete Message-Queue eingefügt.

Weiter geht es mit dem erfolgreichem einklinken in eine schon existierende Message-Queue. Hierfür muß dem Prozeß der KEY, mit dem die Message-Queue angesprochen wird, bekannt sein.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Msgget2}} \\
\hline
\Delta \mathit{MsgQStateSpace} \\
k? : \mathit{KEY} \\
p? : \mathit{PID} \\
i! : \mathit{ID} \\
\mathit{perm}? : \mathbb{P}\, \mathit{PERM} \\
\mathit{flag}? : \mathbb{P}\, \mathit{FLAG} \\
r! : \mathit{RETURN} \\
\hline
k? \in \operatorname{ran} \mathit{getkey} \land k? \neq \mathit{IPC\_PRIVATE} \\
i! = (\mu\, x : \mathit{ID} \mid \mathit{getkey}(x) = k?) \\
(\mathit{getuid}(p?) \text{ maywrite } (\mathit{perm}(i!), \mathit{owner}(i!), \mathit{group}(i!)) \lor \\
\quad \mathit{getuid}(p?) \text{ mayread } (\mathit{perm}(i!), \mathit{owner}(i!), \mathit{group}(i!))) \\
\neg\, (\mathit{IPC\_CREAT} \in \mathit{flag}? \land \mathit{IPC\_EXCL} \in \mathit{flag}?) \\
r! = i! \\
\\
\mathit{getkey}' = \mathit{getkey} \oplus \{i! \mapsto k?\} \\
\mathit{msgq}' = \mathit{msgq} \\
\mathit{creator}' = \mathit{creator} \\
\mathit{owner}' = \mathit{owner} \\
\mathit{group}' = \mathit{group} \\
\mathit{perm}' = \mathit{perm} \\
\mathit{qbytes}' = \mathit{qbytes} \\
\mathit{cbytes}' = \mathit{cbytes} \\
\mathit{msg\_ts}' = \mathit{msg\_ts} \\
\mathit{errno}' = \mathit{errno} \\
\hline
\end{array}
$$

Der erste Hälfte des Schemas $Msgget2$ entspricht der von $Msgget1$. Dies geschieht, um eine spätere Komposition der Schemata zu ermöglichen. Bei allen weiteren Schemata in diesem Kapitel ist dies übriegens auch aus demselben Grund der Fall und wird nicht weiter erwähnt werden.

Für die Vor- und Nachbedingungen ergibt sich bei $Msgget2$ folgendes:

- Es wird ein KEY als Eingabe verlangt, der schon eine Message-Queue bezeichnet, der aber nicht $IPC\_PRIVATE$ entsprechen darf.

- Als ID wird dann diejenige ID zurückgegeben, die die Message-Queue für den Prozeß adressiert.

- Der Prozeß muß das Recht haben Nachrichten in diese Message-Queue zu schreibe oder Nachrichten aus dieser Message-Queue zu lesen.

- Zusätzlich dürfen nicht gleichzeitig die Flags IPC_CREAT und IPC_EXCL gesetzt sein.

- Der zurückgegebene Returnwert entspricht der ID.

Gleichzeitig wirkt sich dieses Schema einzig und allein auf *getkey* aus dem Zustandsraum aus. Alle anderen Objekte des Zustandraumes bleiben unverändert.

Der hauptsächliche Unterschied zum vorherigen Schema ist, daß der KEY dem Prozeß schon bekannt ist. Dies bedeutet, es wird keine neue Message-Queue erzeugt, daher ist auch die Überprüfung, die maximale Anzahl der Message-Queues im Sysetem zu überschreiten, nicht notwendig. Statt muß für die schon bestehende Message-Queue ein Lese- oder Schreibrecht vorhanden sein.

Somit ändert sich der Zustandsraum nicht allzu sehr, es gibt lediglich eine weitere ID, der der KEY zugeordnet ist

Nun kommt noch das erfolgreiche Erzeugen einer neuen Message-Queue, wobei man durch Vorgabe von $IPC\_PRIVATE$ ganz sicher geht eine neue Message-Queue zu erzeugen.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Msgget3}} \\
\hline
\Delta \mathit{MsgQStateSpace} \\
k? : \mathit{KEY} \\
p? : \mathit{PID} \\
i! : \mathit{ID} \\
\mathit{perm}? : \mathbb{P}\, \mathit{PERM} \\
\mathit{flag}? : \mathbb{P}\, \mathit{FLAG} \\
r! : \mathit{RETURN} \\
\hline
k? = \mathit{IPC\_PRIVATE} \\
i! \notin \operatorname{dom} \mathit{msgq} \\
\#(\operatorname{dom} \mathit{msgq}) < \mathit{MSGMNI} \\
r! = i! \\
\\
\mathit{getkey}' = \mathit{getkey} \oplus \{i! \mapsto k?\} \\
\mathit{msgq}' = \mathit{msgq} \oplus \{i! \mapsto \langle\rangle\} \\
\mathit{creator}' = \mathit{creator} \oplus \{i! \mapsto \mathit{getuid}(p?)\} \\
\mathit{owner}' = \mathit{owner} \oplus \{i! \mapsto \mathit{getuid}(p?)\} \\
\mathit{group}' = \mathit{group} \oplus \{i! \mapsto \mathit{getgid}(p?)\} \\
\mathit{perm}' = \mathit{perm} \oplus \{i! \mapsto \mathit{perm}?\} \\
\mathit{qbytes}' = \mathit{qbytes} \oplus \{i! \mapsto \mathit{MSGMNB}\} \\
\mathit{cbytes}' = \mathit{cbytes} \oplus \{i! \mapsto 0\} \\
\mathit{msg\_ts}' = \mathit{msg\_ts} \oplus \{i! \mapsto \langle\rangle\} \\
\mathit{errno}' = \mathit{errno} \\
\hline
\end{array}
$$

Hierbei müssen wiederum gewisse Vor- und Nachbedingungen vorherschen:

- Der KEY muß auf $IPC\_PRIVATE$ gesetzt sein.
- Die ID, der diese Message-Queue zugewiesen wird, darf noch nicht vorhanden sein.
- Auch darf die Höchstgrenze der maximal im System zulässigen Message-Queues nicht überschritten werden.
- Es wird als Returnwert die ID zurückgegeben

Bei den Objekten des Zustandraumes verändert sich *errno* nicht. Bei allen anderen Objekten wird die durch die neue *ID* gekennzeichnete Message-Queue eingefügt.

Dieses Schema unterscheidet sich nicht sehr von $Msgget1$ und man hätte sie tatsächlich zusammenfassen können. Dies geschah aber aus zwei Gründen nicht.

Zum einen, da auch im Source-Code eine entsprechende Fallunterscheidung gegeben ist. Zum anderen, um das bessere Verständnis zu erleichtern und eine gewisse Trennung dieser beiden Fälle zu verdeutlichen. Bei $Msgget1$ wird ein KEY vorgegeben, von dem man nicht sicher weiß, ob er schon im System existiert. Daher muß auch das Flag IPC_CREAT gesetzt sein, um sich nicht versehentlich in eine andere Message-Queue mit einzuklinken. Bei $Msgget3$ ist man sich hingegen ziemlich sicher stets eine neue Message-Queue zu erzeugen. Die Folge ist allerdings, daß im System mehrere Message-Queues mit demselben KEY existieren können und in dem Fall eine korrekte Identifikation nur über die ID möglich ist.

$Msgget2$ unterscheidet sich von $Msgget3$ genau entsprechend wie von $Msgget1$.

5.2 Fehlerfall

Für den Fall, daß ein vom Benutzer nicht erwünschtes Ergebnis auftritt, ergibt sich einer der Fehlerfälle. Hierbei ist zu beachten, daß es einige Fälle gibt, die wir in unserer Spezifikation leider nicht mitmodellieren können. Dies wären:

- EIDRM: Die Message-Queue ist zwischenzeitlich gelöscht worden. Dies wird nicht im IPC-Paket abgefangen.
- ENOMEM: Das Betriebssystem hat nicht genügend Speicher, um den gesamten Inhalt der Message-Queue zu speichern. Hierfür ist der Kernel zuständig.

Bei den spezifizierten Fehlerfälle gibt es einige Gemeinsamkeiten.

Die erste Hälfte dieser Schemata sind alle gleich, da sie die selben Ein- und Ausgaben haben. Dies hat den schon weiter oben angeführten Grund, daß spätere Schemakompositionen ermöglicht werden sollen.

Der Returnwert $r!$ wird stets auf den für Fehlerfälle vorgesehene Wert -1 gesetzt.

Im Zustandsraum wird *errno* stets auf den Namen des Fehlers gesetzt. Dies ist auch der Name des Schema ohne das vorgesetzte msgget.

Für die restlichen Objekte des Zustandraumes ergeben sich keinerlei Veränderungen. Daher werden sie nicht explizit aufgeführt, da dieses durch das Einbinden des Zustandraumes implizit mit vorgegeben ist.

Der erste hier beschriebene Fehlerfall ist der erfolglose Versuch sich in eine Message-Queue einzuklinken, da der Prozeß keine entsprechenden Zugriffsrechte hat.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{msggetEACCES}} \\
\hline
\Delta MsgQStateSpace \\
k? : KEY \\
p? : PID \\
i! : ID \\
perm? : \mathbb{P}\, PERM \\
flag? : \mathbb{P}\, FLAG \\
r! : RETURN \\
\hline
k? \in \operatorname{ran} getkey \land k? \neq IPC\_PRIVATE \\
(\textbf{let}\ i == (\mu\, x : ID \mid getkey(x) = k?) \bullet \\
\quad \neg\, (getuid(p?)\ \text{maywrite}\ (perm(i), owner(i), group(i))) \land \\
\quad \neg\, (getuid(p?)\ \text{mayread}\ (perm(i), owner(i), group(i)))) \\
r! = -1 \\
errno' = EACCES \\
\hline
\end{array}
$$

Die hierbei vorherschenden Vor- und Nachbedingungen sehen wie folgt aus:

- Es wird ein KEY als Eingabe verlangt, der schon eine Message-Queue bezeichnet, der aber nicht $IPC\_PRIVATE$ entspricht.
- Für diese Message-Queue darf der Prozeß weder Schreibrechte noch Leserechte besitzen.
- Als Returnwert gibt es den für die Fehlermeldung entsprechenden Wert -1.

Der zweite Fehlerfall ist der erfolglose Versuch sich in eine Message-Queue einzuklinken, da die beiden Flags IPC_CREAT und IPC_EXCL gesetzt sind.

$$
\begin{array}{|l}
\hline
\textit{msggetEEXIST} \\
\Delta MsgQStateSpace \\
k? : KEY \\
p? : PID \\
i! : ID \\
perm? : \mathbb{P}\, PERM \\
flag? : \mathbb{P}\, FLAG \\
r! : RETURN \\
\hline
k? \in \operatorname{ran} getkey \land k? \neq IPC\_PRIVATE \\
IPC\_CREAT \in flag? \\
IPC\_EXCL \in flag? \\
r! = -1 \\
\\
errno' = EEXIST \\
\hline
\end{array}
$$

Wie aus der informellen Beschreibung des Schemas schon fast zu ersehen ist, ergeben sich folgende Vor- und Nachbedingungen:

- Es wird ein KEY als Eingabe verlangt, der schon eine Message-Queue bezeichnet, der aber nicht $IPC\_PRIVATE$ entspricht.
- Das Flag IPC_CREAT muß gesetzt sein.
- Ebenso muß das Flag IPC_EXCL gesetzt sein.
- Es gibt den Returnwert -1.

Bei diesem Fehlerfall gibt es zwei Sichtweisen. Entweder wird versucht sich in eine bestehende Message-Queue einzuklinken, es existiert aber für den angegebenen KEY gar keine Message-Queue, oder man versucht eine neue Message-Queue zu erzeugen, vergißt aber das Flag IPC_CREAT zu setzen.

$$
\begin{array}{|l}
\hline
\textit{msggetENOENT} \\
\Delta MsgQStateSpace \\
k? : KEY \\
p? : PID \\
i! : ID \\
perm? : \mathbb{P}\, PERM \\
flag? : \mathbb{P}\, FLAG \\
r! : RETURN \\
\hline
k? \notin \operatorname{ran} getkey \land k? \neq IPC\_PRIVATE \\
IPC\_CREAT \notin flag? \\
r! = -1 \\
errno' = ENOENT \\
\hline
\end{array}
$$

Die Vor- und Nachbedingungen dieses Schemas ergeben sich wieder von selbst:

- Es wird ein KEY als Eingabe verlangt, der schon eine Message-Queue bezeichnet, der aber nicht $IPC\_PRIVATE$ entspricht.
- Das Flag IPC_CREAT ist nicht gesetzt.
- Es gibt den Returnwert -1.

Als letzten Fehlerfall führen wir den Versuch zur Erzeugung einer neuen Message-Queue an, der erfolglos bleibt, da die Systemgrenze *MSGMNI* der maximalen Anzahl von Message-Queues in einem System überschritten würde.

$$
\begin{array}{|l}
\hline
\textit{msggetENOSPC} \\
\Delta \textit{MsgQStateSpace} \\
k? : \textit{KEY} \\
p? : \textit{PID} \\
i! : \textit{ID} \\
\textit{perm}? : \mathbb{P}\,\textit{PERM} \\
\textit{flag}? : \mathbb{P}\,\textit{FLAG} \\
r! : \textit{RETURN} \\
\hline
k? = \textit{IPC\_PRIVATE} \lor \\
(\textit{IPC\_CREAT} \in \textit{flag}? \land k? \notin \operatorname{ran} \textit{getkey}) \\
\#(\operatorname{dom} \textit{msgq}) \geq \textit{MSGMNI} \\
r! = -1 \\
\\
\textit{errno}' = \textit{ENOSPC} \\
\hline
\end{array}
$$

Bei diesen Vor- und Nachbedingungen wird es noch einmal interessanter:

- Es wird gefordert, daß entweder der *KEY* *IPC_PRIVATE* entspricht, oder der *KEY* noch nicht vorhanden ist und gleichzeitig das Flag *IPC_CREAT* gesetzt ist.
- Die Anzahl der vorhandenen Message-Queues müssen schon größer oder gleich der maximalen Anzahl von Message-Queues, *MSGMNI*, sein.
- Es gibt den Returnwert -1.

5.3 User-Interface

Für die Sichtweise des Benutzers stellen wir die verschiedene Fälle zu einem einzigen Schema zusammen, da ein Benutzer nicht zwischen diesen Fällen unterscheidet.

Hierfür benutzen wir die Schema-Komposition, bei der das Schema durch die nichtdeterministische Verknüpfung anderer Schemata entsteht. Um diesen Nichtdeterminismus aufzulösen haben wir die Vorbedingungen der einzelnen Schemata stets unterschiedlich gehalten, so daß immer eindeutig ist, welches Schema zum tragen kommt.

So kann man erst einmal sämtliche Erfolgsfälle zusammenfassen. Tritt einer dieser Fälle ein, so erhält der Benutzer ein Ergebnis, wie er es sich erwünscht hat.

$$Msgget \mathrel{\widehat=} Msgget1 \lor Msgget2 \lor Msgget3$$

Für den Fehlerfall ergibt sich die Gesamtzahl der Fehler aus der Schema-disjunktion der vier Fehler-Schemata:

$$\begin{aligned} Msggetfault \mathrel{\widehat=} {} & msggetEACCES \lor msggetEEXIST \lor msggetENOENT \\ & \lor msggetENOSPC \end{aligned}$$

Für den Benutzer besteht die Aufteilung zwischen Erfolgs- und Fehlerfall nicht. Er benutzt das System und erwartet eine entsprechende Rückmeldung. Somit sollten der Erfolgs- und der Fehlerfall zusammengefaßt werden:

$$msgget \mathrel{\widehat=} Msgget \lor Msggetfault$$

Somit ist *msgget* eine abstrakte formale Spezifizierung des nach Aussen sichtbaren Verhalten der Message-Queue Operation *msgget*().

6 Message-Queue Operation *msgsnd*()

Die Operation *msgsnd*() wird von den Prozessen dazu benutzt, um in einer vorhandenen und ihnen bekannten Message-Queue Nachrichten zu senden. Eine Nachricht besteht dabei aus einem Typ und einer Anzahl von Bytes.

Auch hier gibt es wieder eine Aufteilung, diesmal in einen erfolgreichen Fall und in mehrere Fehlerfälle. Die Begründung ist immer noch dieselbe wie bei *msgget*().

Im erfolgreichen Fall muß der Prozeß eine bestehende Message-Queue angeben, in die er seine Nachricht senden will, und für die er natürlich Schreibrechte besitzen muß. Von der Nachricht selbst muß er den Typ und die Länge der Nachricht wissen.

Die hier genannten Fehlerfälle sind keine Fehler im Sinne eines Systemfehlers sondern das gewünschte Verhalten tritt für den Benutzer nicht ein.

Der erste solche Fehlerfall ist der Versuch in eine Message-Queue zu senden, die schon voll ist.

Sodann kann es passieren, daß der Prozeß für die Message-Queue keine Schreibrechte besitzt.

Und es ist noch möglich, daß es eine falsche Eingabe gibt. Solch eine Eingabe könnte eine *ID* sein, die überhaupt nicht existiert, oder der Typ einer Nachricht, der kein gültiger Wert ist, oder aber die Länge der Nachricht, die nicht einem korrektem Wert entspricht.

Es gibt auch wieder einige Fehlerfälle, die wir in unserer Spezifikation nicht anführen. Hierfür gibt es zweierlei Gründe.

Zum einen werden einige dieser Fehlerfälle in anderen Teilen des Betriebsystems abgefangen. Hierunter fallen die Situationen, daß ein Pointer auf eine nicht existierende Adresse zeigt, die Message-Queue inzwischen gelöscht wurde und das das System nicht genügend Speicher hat, um den gesamten Inhalt der Message-Queue zu speichern.

Zum anderen existiert in Z keine Zeitkomponente, ein zeitlicher Verlauf und insbesondere das Warten auf ein Ereignis kann daher nicht dargestellt werden. Und daher kann in unserer Spezifikation das Warten auf eine Message-Queue und gleichzeitige Erhalten eines Interruptes nicht angeführt werden.

Für den Benutzer ist dies wieder völlig unwichtig, er benutzt *msgsnd*() und erwartet ein Ergebnis. Um dieses abstrakt darzustellen erstellen wir ein User-Interface, in dem wir das nach Außen sichtbare Verhalten spezifizieren.

In diesem Kapitel fangen wir mit dem Erfolgsfall an, beschreiben anschließend die verschiedenen Fehlerfälle und erstellen zum Schluß das User-Interface. In dem letzten Abschnitt gehen wir auch noch mal näher auf die nicht spezifizierten Fehlerfälle ein.

Somit erhalten wir abschließend eine abstrakte formale Spezifikation des Verhaltens der Message-Queue Operation $msgsnd()$.

6.1 Erfolgsfall

Der Erfolgsfall ist das erfolgreiche Senden einer Nachricht in einer Message-Queue. Es muß natürlich eine existierende Message-Queue adressiert sein und der Prozeß muß für diese Message-Queue auch Schreibrechte besitzen.

$$
\begin{array}{|l}
\hline
\textit{Msgsnd} \\
\Delta \textit{MsgQStateSpace} \\
t? : \textit{TYPE} \\
b? : \textit{BYTES} \\
\textit{msgsz}? : \mathbb{N} \\
p? : \textit{PID} \\
i? : \textit{ID} \\
f? : \mathbb{P}\,\textit{FLAG} \\
r! : \textit{RETURN} \\
\hline
i? \in \operatorname{dom} \textit{msgq} \\
\textit{getuid}(p?) \text{ maywrite } (\textit{perm}(i?), \textit{owner}(i?), \textit{group}(i?)) \\
\textit{cbytes}(i?) + \textit{msgsz}? \leq \textit{qbytes}(i?) \\
\textit{msgsz}? \leq \textit{MSGMAX} \\
\textit{msgsz}? \geq 0 \\
t? > 0 \\
r! = 0 \\
\\
\textit{getkey}' = \textit{getkey} \\
\textit{msgq}' = \textit{msgq} \oplus \{ i? \mapsto \textit{msgq}(i?) \frown \langle (t?, b?) \rangle \} \\
\textit{creator}' = \textit{creator} \\
\textit{owner}' = \textit{owner} \\
\textit{group}' = \textit{group} \\
\textit{perm}' = \textit{perm} \\
\textit{qbytes}' = \textit{qbytes} \\
\textit{cbytes}' = \textit{cbytes} \oplus \{ i? \mapsto \textit{cbytes}(i?) + \textit{msgsz}? \} \\
\textit{msg\_ts}' = \textit{msg\_ts} \oplus \{ i? \mapsto \textit{msg\_ts}(i?) \frown \langle \textit{msgsz}? \rangle \} \\
\textit{errno}' = \textit{errno} \\
\hline
\end{array}
$$

Die erste Hälfte aller Schemata für *msgsnd*() sieht gleich aus, um bei der Erstellung des User-Interfaces sämtliche Schemata miteinander kombinieren zu können.

Zuerst wird stets der Zustandsraum eingebunden, auf dem Veränderungen stattfinden können.

Dann gibt es als Eingaben die Prozeßidentifikation *PID* sowie die Identifikation einer Message-Queue *ID*. Um eine Nachricht zu Senden benötigt man auch eine Nachricht, insbesondere ihren Typen und die Nachricht selbst, die dabei eine bestimmte Länge besitzt. Eventuell können auch Flags vorgegeben werden.

Als Rückgabe erhält man einen Returnwert, bei den Fehlerfällen wird dieser stets auf -1, beim Erfolgsfall auf 0 gesetzt.

In der zweiten Hälfte des Schema folgen zunächst diese Vor- und Nachbedingungen:

- Die *ID* muß eine schon existierende Message-Queue adressieren.
- Der Prozeß muß auf der durch die *ID* bezeichneten Message-Queue Schreibrechte besitzen.
- Durch die zu sendende Nachricht darf die Länge der Message-Queue die maximale Länge der Message-Queue nicht überschreiten.
- Die Länge der Nachricht darf nicht die maximale Länge einer Nachricht überschreiten.
- Genauso darf die Länge einer Nachricht nicht einem negativem Wert entsprechen.
- Der Typ der Nachricht muß positiv sein.
- Daraufhin gibt es als Returnwert die 0 als Zeichen des erfolgreichen Sendens der Nachricht.

Auch die Objekte des Zustandraumes unterliegen gewissen Änderungen, wobei die Konsistenzbedingungen des Zustandraumes nicht verletzt werden.

Da wäre zuerst *msgq*, bei der an die existierende Message-Queue die Nachricht mit Angabe des Typen hinten dran gehängt wird.

Bei *cbytes* wird die Länge der Message-Queue um die Länge der gesendeten Nachricht erhöht.

Und bei *msg_ts* wird auch noch die Länge für die letzte Nachricht eingefügt.

Die restlichen Objekte bleiben vom Senden einer Nachricht unberüht und somit unverändert bestehen.

6.2 Fehlerfall

Einige der möglichen Fehlerfälle sind in unserer Spezifikation nicht enthalten. Dieses hat natürlich sein Gründe. Entweder wird der Fehlerfall nicht in dem von uns spezifizierten Teil des Betriebssystems, den Message-Queues, abgefangen oder er ist in Z nicht darstellbar.

Gehört ein Fehlerfall nicht in den Bereich der Message-Queues, so ist unsere Spezifikation natürlich weiterhin vollständig. Dieser Fall würde bei der Spezifikation des entsprechenden Teils des Betriebsystem mit zu erfassen sein.

Leider gibt es auch den Fall, daß die Sprache Z nicht mächtig genug ist, insbesondere durch das Fehlen einer zeitlichen Komponente. Diesen Fall müßten wir eigentlich mit spezifizieren, können dies aber nicht. Unsere Spezifikation ist an dieser Stelle also nicht mehr vollständig, treten entsprechende Vorbedingungen auf, so geht sie in einen unspezifizierten Zustand über.

Dieses läßt sich leider nicht vermeiden, in anderen Spezifikationssprachen hätten wir größere Probleme bekommen.

Hier sind die von uns nicht mitspezifizierten Fehlerfälle:

- EFAULT: Ein Pointer zeigt auf eine nicht existierende Adresse. Hierfür ist der Kernel zuständig

- EIDRM: Die Message-Queue ist zwischenzeitlich gelöscht worden. Dies wird nicht in IPC abgefangen.

- EINTR: Der Prozeß wartet auf eine Message-Queue und erhält einen Interrupt. Dies kann wegen der fehlenden Zeitkomponente in Z nicht dargestellt werden.

- ENOMEM: Das System hat nicht genügend Speicher um den gesamten Inhalt der Message-Queue zu speichern. Hierfür ist der Kernel zuständig.

Es gibt bei allen Fehlerfällen Gemeinsamkeiten, die wir hier einmal anführen wollen, um sie nicht stets ansprechen zu müssen.

Die erste Hälfte des Schemas entspricht immer den schon beschriebenen üblichen Ein- und Ausgaben bei $msgsnd()$.

Der Returnwert $r!$ wird auf den für Fehlerfälle üblichen Wert von -1 gesetzt.

Für den Zustandsraum gilt allgemein, daß *errno* auf den Namen des Fehlers gesetzt wird, also den Namen des Schema ohne das vorgefügte msgsnd.

Die restlichen Objekte des Zustandraumes unterliegen keinerlei Veränderungen, werden deshalb hier auch nicht aufgeführt. Durch das Einbinden von dem Zustandsraum ist dieses schon implizit mit vorgegeben.

Nun kann der Fall eintreten, daß das Senden einer Nachricht in einer Message-Queue erfolglos ist, da die Message-Queue schon voll ist. Gleichzeitig muß das Flag IPC_NOWAIT gesetzt sein.

$msgsndEAGAIN$

$$\Delta MsgQStateSpace$$
$$t? : TYPE$$
$$b? : BYTES$$
$$msgsz? : \mathbb{N}$$
$$p? : PID$$
$$i? : ID$$
$$f? : \mathbb{P}\, FLAG$$
$$r! : RETURN$$

$$i? \in \operatorname{dom} msgq$$
$$cbytes(i?) + msgsz? > qbytes(i?)$$
$$IPC\_NOWAIT \in f?$$
$$r! = -1$$
$$errno' = EAGAIN$$

Damit dieses Schema ausgeführt werden kann, muß eine ID vorgegeben werden, für die eine Message-Queue existiert. Auch muß die Länge der Message-Queue mit der Länge der zu sendenden Nachricht die maximale Länge dieser Message-Queue überschreiten. Zudem muß das Flag IPC_NOWAIT gesetzt sein.

Weiter geht es mit dem Fall, daß das Senden einer Nachricht erfolglos ist, da der Prozeß für die angegebene Message-Queue keine Schreibrechte besitzt.

$$
\begin{array}{|l}
\hline
\textit{msgsndEACCES} \\
\hline
\Delta \textit{MsgQStateSpace} \\
t? : \textit{TYPE} \\
b? : \textit{BYTES} \\
\textit{msgsz}? : \mathbb{N} \\
p? : \textit{PID} \\
i? : \textit{ID} \\
f? : \mathbb{P}\,\textit{FLAG} \\
r! : \textit{RETURN} \\
\hline
i? \in \operatorname{dom} \textit{msgq} \\
\neg\ (\textit{getuid}(p?) \text{ maywrite } (\textit{perm}(i?), \textit{owner}(i?), \textit{group}(i?))) \\
r! = -1 \\
\\
\textit{errno}' = \textit{EACCES} \\
\hline
\end{array}
$$

Für den Eintritt dieses Schema muß wieder ein gültige *ID* angegeben sein. Der Prozeß darf aber keine Schreibrechte für diese Message-Queue besitzen.

Dieser Fall tritt bei verschiedenen Möglichkeiten ein, stets ist es jedoch eine falsche Eingabe vorgegeben.

$$
\begin{array}{l}
\textit{msgsndEINVAL} \\
\hline
\Delta \mathit{MsgQStateSpace} \\
t? : \mathit{TYPE} \\
b? : \mathit{BYTES} \\
\mathit{msgsz}? : \mathbb{N} \\
p? : \mathit{PID} \\
i? : \mathit{ID} \\
f? : \mathbb{P}\,\mathit{FLAG} \\
r! : \mathit{RETURN} \\
\hline
i? \notin \operatorname{dom} \mathit{msgq} \lor \\
\quad t? \leq 0 \lor \\
\quad \mathit{msgsz}? < 0 \lor \\
\quad \mathit{msgsz}? > \mathit{MSGMAX} \\
r! = -1 \\
\mathit{errno}' = \mathit{EINVAL} \\
\hline
\end{array}
$$

Entweder wird hier eine falsche Message-Queue-ID vorgegeben, oder der Typ einer Nachricht ist nicht richtig, oder aber die Länge der Nachricht ist zu klein oder zu groß.

6.3 User-Interface

Die Aufteilung in die verschiedenen Fälle ergibt sich für den Benutzer nicht, er sieht nur das Verhalten einer einzelnen Operation und erhält entsprechend seinen Vorgaben entsprechende Rückgaben und Veränderungen. Um dieses nach Außen sichtbare Verhalten wiederzuspiegeln fügen wir alle erstellten Schema zusammen.

Dies geschieht mit einer Schemadisjunktion, die ermöglicht wird, da wir bei allen Schemata dieselben Ein- und Ausgaben benutzen.

Da hier der seltene Fall eintritt, daß wir den Erfolgsfall nicht weiter unterteilen mußten, brauchen wir auch keine Schemadisjunktion durchführen.

Tritt ein für den Benutzer nicht gewünschtes Ereignis ein, so haben wir hierfür die Fehlerfälle. Weiter vorne haben wir einige Fälle angeführt, die wir in unserer Spezifikation nicht darstellen.

Ansonsten ergibt sich die Gesamtzahl der Fehler aus der Schemadisjunktion:

$$Msgsndfault \mathrel{\widehat{=}} msgsndEAGAIN \lor msgsndEACCES \lor msgsndEINVAL$$

Für den Benutzer ergibt sich die Unterteilung in Erfolgs- und Fehlerfall nicht, er ruft nur eine einzige Operation auf. Daher führen wir auch noch diese beiden Fälle zusammen:

$$msgsnd \mathrel{\widehat{=}} Msgsnd \lor Msgsndfault$$

Somit entspricht das formal spezifizierte abstrakte *msgsnd* dem nach Aussen sichtbaren Verhalten der Message-Queue Operation $msgsnd()$.

7 Message-Queue Operation *msgrcv*()

Bei der Operation *msgrcv*() kann ein Prozeß aus einer ihm bekannten Message-Queue eine Nachricht auslesen.

Dieses ist die am kompliziertesten darzustellende Operation. Daher müssen wir die meisten Fallunterscheidungen aller Message-Queue Operationen treffen, um die Verständlichkeit und Lesbarkeit noch halbwegs zu gewährleisten.

Im erfolgreichen Fall ist natürlich stets vorausgesetzt, daß eine korrekte Message-Queue adressiert ist und der Prozeß auch Leserechte besitzt. Nun gibt es allerdings noch vier verschiedene Möglichkeiten eine Nachricht aus dieser Message-Queue zu lesen.

Zuerst wäre dies das Lesen der ersten Nachricht, die in der Message-Queue existiert. Dabei ist es egal, welchen Typ diese besitzt.

Dann wäre noch das Lesen der ersten Nachricht, die einem bestimmten Typ entspricht.

Weiter geht es mit der Möglichkeit die erste Nachricht zu lesen, die nicht einem bestimmten Typ besitzt.

Und als letztes existiert noch die Möglichkeit die erste Nachricht des kleinsten Typen der Message-Queue zu lesen, wobei diese allerdings kleiner oder gleich einem vorgegebenen Typ sein muß.

Durch den Aufbau der Erfolgsfälle ist es aber leider so, daß einige Fehlerfälle nicht allgemein für die Operation spezifiziert werden können, da sie vom Typ der Nachricht abhängen.

Dies ist dann gleich der Fall, falls die Länge einer Nachricht größer als Erlaubt ist und das Flag *IPC_NOERROR* nicht gesetzt ist. Hierbei ergeben sich also insgesamt vier Fehlerfälle.

Hat der Prozeß hingegen für die Message-Queue keine Leserechte, so ist dies ein allgemeines Problem und es gibt einen weiteren Fehlerfall.

Genauso allgemein ist es, wenn ein fehlerhafter Wert angegeben wird. Dies kann eine nicht existierende *ID* oder eine negative Nachrichtenlänge sein.

Und wieder vom Typ der Nachricht abhängig ist der Fall, daß keine Nachricht durch den angegebenen Typen gekennzeichnet wird. Also entstehen bedingt durch die Erfolgsfällen vier Fehlerfälle.

Aus den schon weiter vorne angeführten Gründen gibt es wieder einige Fehlerfälle, die wir nicht spezifizieren.

Diese Aufteilung in verschiedene Fälle entstanden wie schon erwähnt zum besseren Verständnis und besseren Lesbarkeit der Spezifikation. Für den Benutzer eines Systems geht es aber nur um das Resultat seiner Eingaben. Daher erstellen wir für ihn ein User-Interface.

Doch zuerst beschreiben wir die verschiedenen Erfolgsfälle, anschließend die Fehlerfälle und anschließend wird dann das User-Interface erstellt.

Somit erhalten wir abschließend eine abstrakte formale Spezifikation des Verhaltens der Message-Queue Operation $msgrcv()$.

7.1 Erfolgsfall

Um später die verschiedenen Erfolgsfälle wieder zusammenfügen zu können, haben diese alle dieselben Ein- und Ausgaben.

Es wird der Zustandsraum eingebunden, auf dem Veränderungen ausgeführt werden können. Mit der *ID* wird eine Message-Queue bezeichnet und mit der *PID* herausgefunden, ob ein Leserecht vorliegt. Es wird natürlich ein Typ für die Nachricht vorgegeben, der aber verschiedene Bedeutung haben kann. Die zu lesende Nachricht darf eine gewisse Länge nicht überschreiten. Und es können Flags gesetzt werden.

Zurückgegeben wird der Typ einer Nachricht, der von dem vorgegbenen Typen je nach dem entsprechenden Fall, unterschiedlich sein kann. Die Nachricht selbst wird natürlich auch zurückgegeben genauso wie ein Returnwert, der im Erfolgsfall immer der Länge der gelesenen Nachricht entspricht.

Um das Verständnis zu erleichtern und die einzelnen Erfolgsfälle besser miteinander vergleichen zu können haben wir sie alle gleich aufgebaut. Vor allem der erste Erfolgsfall $Msgrcv1$ ließe sich wesentlich einfacher darstellen, allerdings wäre ein Vergleich mit den anderen Erfolgsfällen nicht mehr so einfach. Zudem muß man nur einmal den Aufbau verstehen, schon hat man die vier Erfolgsfälle verstanden.

Die Vorbedingungen gleichen sich auch teilweise. Die mit der *ID* angegebene Message-Queue muß existieren, diese darf nicht leer sein sondern muß mindestens eine Nachricht enthalten und der Prozeß muß für diese Message-Queue Leserechte besitzen.

Die Schemata unterscheiden sich bei dem vorgegebenen Typ für die Nachricht und den zu setzenden Flags.

Auch der Zustandsraum unterliegt bei allen Schemata denselben Veränderungen. Dieses ähnelt zudem sehr $msgsnd()$, nur wird diesmal eine Nachricht aus der Message-Queue gelöscht.

Bei *msgq* wird in der durch die *ID* vorgegebenen Message-Queue die durch den Typ und die Flags angegebene Nachricht gelöscht und die verbleibenden Nachrichten in der Message-Queue in die richtige Reihenfolge gebracht.

Um die Länge dieser Nachricht muß auch die Länge der Message-Queue in *cbytes* geändert werden.

Da nun eine Nachricht weniger in der Message-Queue existiert muß auch die Länge dieser Nachricht in msg_ts gelöscht werden.

Alle anderen Objekte des Zustandraumes bleiben unverändert.

Doch nun zu den vier Erfolgsfällen, wobei wir versuchen werden die Unterschiede deutlich darzustellen. Beim ersten Erfolgsfall wollen wir dann auch probieren unsere Spezifikation noch etwas näher zu erläutern.

Das erfolgreiche Lesen der ersten Nachricht in einer Message-Queue.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\, Msgrcv1 \,\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta MsgQStateSpace \\
p? : PID \\
i? : ID \\
t? : TYPE \\
msgsz? : \mathbb{N} \\
f? : \mathbb{P}\, FLAG \\
t! : TYPE \\
b! : BYTES \\
r! : RETURN \\
\hline
i? \in \operatorname{dom} msgq \\
t? = 0 \\
getuid(p?) \text{ mayread } (perm(i?), owner(i?), group(i?)) \\
msgq(i?) \neq \langle\rangle \land \\
(\textbf{let } j == \{b1 : BYTES;\ t1 : TYPE \bullet (t1, b1)\} \bullet \\
\quad msgq(i?) \upharpoonright j \neq \langle\rangle \land \\
\quad (MSG\_NOERROR \in f? \lor \\
\qquad msg\_ts(i?)(min(\operatorname{dom}(msgq(i?) \rhd j))) \leq msgsz?) \land \\
\quad (t!, b!) = (\mu\, t2 : TYPE;\ b2 : BYTES \mid (t2, b2) = (msgq(i?) \upharpoonright j)(1) \bullet \\
\qquad\quad (t2, truncate(b2, msgsz?))) \land \\
\quad r! = min\{msg\_ts(i?)(min(\operatorname{dom}(msgq(i?) \rhd j))), msgsz?\} \land \\
\\
\quad getkey' = getkey \land \\
\quad msgq' = msgq \oplus \{i? \mapsto \\
\qquad squash(\{min(\operatorname{dom}(msgq(i?) \rhd j))\} \lhd msgq(i?))\} \land \\
\quad creator' = creator \land \\
\quad owner' = owner \land \\
\quad group' = group \land \\
\quad perm' = perm \land \\
\quad qbytes' = qbytes \land \\
\quad cbytes' = cbytes \oplus \\
\qquad \{i? \mapsto cbytes(i?) - msg\_ts(i?)(min(\operatorname{dom}(msgq(i?) \rhd j)))\} \land \\
\quad msg\_ts' = msg\_ts \oplus \\
\qquad \{i? \mapsto squash(\{min(\operatorname{dom}(msgq(i?) \rhd j))\} \lhd msg\_ts(i?))\} \land \\
\quad errno' = errno) \\
\hline
\end{array}
$$

Die Gemeinsamkeiten mit den anderen Schemata sind oben schon beschrieben worden. Nun wenden wir uns den Unterschieden zu.

Um das Lesen der ersten Nachricht einer Message-Queue zu erreichen wird der Typ 0 vorgegeben.

Hierbei ist es nun unwichtig, ob das Flag MSG_EXCEPT gesetzt ist oder nicht. Daher brauch es in diesem Schema nicht aufgeführt zu werden.

Sodann werden zwei lokale Variablen deklariert, die der zu lesenden Nachricht und ihrem Typ entsprechen. Diese Variablen müssen einigen Anforderungen genügen.

Für die Variable des Typen kann es vergleichende Bedingungen mit dem vorgegebenen Typen $t?$ geben. Dies ist in diesem Fall nicht gegeben.

Die Message-Queue muß eine Nachricht dieses Typens besitzen.

Sodann muß entweder das Flag $MSG\_NOERROR$ gesetzt sein oder die Länge der zu lesenden Nachricht muß kleiner oder gleich der Länge sein, die als maximale Länge für die Nachricht angegeben wurde.

Da immer noch mehrere Nachrichten vielleicht sogar unterschiedlichen Typs (hier sieht man die Verallgemeinerung, da man in diesem speziellen Fall diesen Teil vereinfachen könnte) diesen Anforderungen gerecht werden können, muß für die Ausgabe eine spezielle Nachricht aus dieser Menge genommen werden.

Da es eine geordnete Menge ist können wir die erste entsprechende Nachricht nehmen und deren Typ zurückgeben sowie die Bytes, wobei eventuell nur die ersten Bytes bis zu der gewünschten Länge der Nachricht genommen werden.

Als Returnwert erhält man die Länge der gelesenen Nachricht, also die Anzahl der gelesenen Bytes. Hierfür müssen in der Spezifikation einige Verrenkungen vorgenommen werden.

$msgq(i?)$ entspricht einer Sequenz von Tupeln aus Typen und Bytes. $msgq(i?) \rhd j)$ ergibt eine Sequenz, bei der nur noch Typen und Bytes, die den lokalen Variablen entsprechen, vorhanden sind. $\mathrm{dom}(msgq(i?) \rhd j)$ ergibt den Domain dieser Sequenz, wobei eine Sequenz eine Abbildung der natürlichen Zahlen in eine Menge entspricht. Mit $min(\mathrm{dom}(msgq(i?) \rhd j))$ wird das kleinste Element dieser Menge gefunden. Diese Zahl entspricht der Stellen, an der die zu lesende Nachricht in der Message-Queue positioniert ist.

Im vorliegendem Falle ist dies immer die erste Stelle, man hätte es sich also wesentlich einfacher machen können.

Für den Returnwert erhält man also die Länge der Nachricht, die an dieser Stelle in der Message-Queue eingetragen ist oder die vorgegebene maximale Länge einer Nachricht, je nachdem, welcher Wert kleiner ist.

Für die Objekte des Zustandraumes arbeitet man ebenfalls mit diesem Wert.

Bei *msgq* wird die Nachricht, die an dieser Stelle steht per Domain Substraktion gelöscht und die Message-Queue mit *squash* neu durchnummeriert.

Bei *cbytes* wird die Länge der Nachricht, die an ausgerechneter Stelle sich befindet, von der bisherigen Gesamtlänge abgezogen.

Und bei *msg_ts* wird die Länge, die an der ausgerechneten Stelle verzeichnet ist, ebenfalls per Domain-Substraktion gelöscht und die Sequenz mit *squash* neu geordnet.

Bei diesem Fall geht es um das erfolgreiche Lesen der ersten Nachricht, die einen bestimmten Typen besitzt.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\ \mathit{Msgrcv2}\ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta \mathit{MsgQStateSpace} \\
p? : \mathit{PID} \\
i? : \mathit{ID} \\
t? : \mathit{TYPE} \\
f? : \mathbb{P}\,\mathit{FLAG} \\
\mathit{msgsz}? : \mathbb{N} \\
t! : \mathit{TYPE} \\
b! : \mathit{BYTES} \\
r! : \mathit{RETURN} \\
\hline
i? \in \mathrm{dom}(\mathit{msgq}) \\
t? > 0 \\
\mathit{MSG\_EXCEPT} \notin f? \\
\mathit{getuid}(p?)\ \mathrm{mayread}\ (\mathit{perm}(i?), \mathit{owner}(i?), \mathit{group}(i?)) \\
\mathit{msgq}(i?) \neq \langle\rangle \land \\
(\mathbf{let}\ j == \{b1 : \mathit{BYTES};\ t1 : \mathit{TYPE} \mid t1 = t? \bullet \\
\quad (t1, b1)\} \bullet \\
\qquad \mathit{msgq}(i?) \upharpoonright j \neq \langle\rangle \land \\
\qquad (\mathit{MSG\_NOERROR} \in f? \lor \\
\qquad\quad \mathit{msg\_ts}(i?)(\mathit{min}(\mathrm{dom}(\mathit{msgq}(i?) \rhd j))) \leq \mathit{msgsz}?) \land \\
\qquad (t!, b!) = (\mu\, t2 : \mathit{TYPE};\ b2 : \mathit{BYTES} \mid (t2, b2) = (\mathit{msgq}(i?) \upharpoonright j)(1) \bullet \\
\qquad\quad (t2, \mathit{truncate}(b2, \mathit{msgsz}?))) \land \\
\qquad r! = \mathit{min}\{\mathit{msg\_ts}(i?)(\mathit{min}(\mathrm{dom}(\mathit{msgq}(i?) \rhd j))), \mathit{msgsz}?\} \land \\
\\
\qquad \mathit{getkey}' = \mathit{getkey} \land \\
\qquad \mathit{msgq}' = \mathit{msgq} \oplus \{i? \mapsto \\
\qquad\quad \mathit{squash}(\{\mathit{min}(\mathrm{dom}(\mathit{msgq}(i?) \rhd j))\} \ndres \mathit{msgq}(i?))\} \land \\
\qquad \mathit{creator}' = \mathit{creator} \land \\
\qquad \mathit{owner}' = \mathit{owner} \land \\
\qquad \mathit{group}' = \mathit{group} \land \\
\qquad \mathit{perm}' = \mathit{perm} \land \\
\qquad \mathit{qbytes}' = \mathit{qbytes} \land \\
\qquad \mathit{cbytes}' = \mathit{cbytes} \oplus \{i? \mapsto \mathit{cbytes}(i?) - \\
\qquad\quad \mathit{msg\_ts}(i?)(\mathit{min}(\mathrm{dom}(\mathit{msgq}(i?) \rhd j)))\} \land \\
\qquad \mathit{msg\_ts}' = \mathit{msg\_ts} \oplus \{i? \mapsto \\
\qquad\quad \mathit{squash}(\{\mathit{min}(\mathrm{dom}(\mathit{msgq}(i?) \rhd j))\} \ndres \mathit{msg\_ts}(i?))\} \land \\
\qquad \mathit{errno}' = \mathit{errno}) \\
\hline
\end{array}
$$

In diesem Schema ist der vorgegebene Typ größer als 0.

Gleichzeitig darf das Flag MSG_EXCEPT nicht gesetzt.

Und als Bedingung für die lokale Variable ergibt sich, daß der Typ gleich dem vorgegebenen Typ sein muß.

Alle restlichen Gegebenheiten entsprechen den schon beschriebenen allgemeinen Bedingungen für die Erfolgsfälle und den Beschreibungen der Notation.

Der Unterschied bei $Msgrcv2$ zu $Msgrcv1$ besteht also darin, daß hier ein konkreter Typ für die zu lesende Nachricht vorgegeben wird. Daraufhin wird auch die Menge der lokalen Variablen, die der Menge aller möglichen passenden Nachrichten entspricht, eingeschränkt.

Nun kann es auch noch zum erfolgreichen Lesen der ersten Nachricht einer Message-Queue kommen, die nicht einem bestimmten Typ entspricht.

$$
\begin{array}{l}
\textit{Msgrcv3} \\
\hline
\Delta \mathit{MsgQStateSpace} \\
p? : \mathit{PID} \\
i? : \mathit{ID} \\
t? : \mathit{TYPE} \\
f? : \mathbb{P}\, \mathit{FLAG} \\
\mathit{msgsz}? : \mathbb{N} \\
t! : \mathit{TYPE} \\
b! : \mathit{BYTES} \\
r! : \mathit{RETURN} \\
\hline
i? \in \operatorname{dom} \mathit{msgq} \\
t? > 0 \\
\mathit{MSG\_EXCEPT} \in f? \\
\mathit{getuid}(p?) \text{ mayread } (\mathit{perm}(i?), \mathit{owner}(i?), \mathit{group}(i?)) \\
\mathit{msgq}(i?) \neq \langle\rangle \land \\
(\mathbf{let}\ j == \{b1 : \mathit{BYTES};\ t1 : \mathit{TYPE} \mid t1 \neq t? \bullet \\
\quad (t1, b1)\} \bullet \\
\qquad \mathit{msgq}(i?) \upharpoonright j \neq \langle\rangle \land \\
\qquad (\mathit{MSG\_NOERROR} \in f? \lor \\
\qquad\quad \mathit{msg\_ts}(i?)(\mathit{min}(\operatorname{dom}(\mathit{msgq}(i?) \rhd j))) \leq \mathit{msgsz}?) \land \\
\qquad (t!, b!) = (\mu\, t2 : \mathit{TYPE};\ b2 : \mathit{BYTES} \mid (t2, b2) = (\mathit{msgq}(i?) \upharpoonright j)(1) \bullet \\
\qquad\quad (t2, \mathit{truncate}(b2, \mathit{msgsz}?))) \land \\
\qquad r! = \mathit{min}\{\mathit{msg\_ts}(i?)(\mathit{min}(\operatorname{dom}(\mathit{msgq}(i?) \rhd j))), \mathit{msgsz}?\} \land \\
\\
\qquad \mathit{getkey}' = \mathit{getkey} \land \\
\qquad \mathit{msgq}' = \mathit{msgq} \oplus \{i? \mapsto \\
\qquad\quad \mathit{squash}(\{\mathit{min}(\operatorname{dom}(\mathit{msgq}(i?) \rhd j))\} \ndres \mathit{msgq}(i?))\} \land \\
\qquad \mathit{creator}' = \mathit{creator} \land \\
\qquad \mathit{owner}' = \mathit{owner} \land \\
\qquad \mathit{group}' = \mathit{group} \land \\
\qquad \mathit{perm}' = \mathit{perm} \land \\
\qquad \mathit{qbytes}' = \mathit{qbytes} \land \\
\qquad \mathit{cbytes}' = \mathit{cbytes} \oplus \{i? \mapsto \mathit{cbytes}(i?) - \\
\qquad\quad \mathit{msg\_ts}(i?)(\mathit{min}(\operatorname{dom}(\mathit{msgq}(i?) \rhd j)))\} \land \\
\qquad \mathit{msg\_ts}' = \mathit{msg\_ts} \oplus \{i? \mapsto \\
\qquad\quad \mathit{squash}(\{\mathit{min}(\operatorname{dom}(\mathit{msgq}(i?) \rhd j))\} \ndres \mathit{msg\_ts}(i?))\} \land \\
\qquad \mathit{errno}' = \mathit{errno})
\end{array}
$$

Auch in diesem Fall ist der vorgegebene Typ größer als 0.

Allerdings muß das Flag MSG_EXCEPT unbedingt gesetzt sein.

Die zusätzliche Bedingung für die lokale Variable des Types ist diesmal, daß sie ungleich dem vorgegebenen Typ zu sein hat.

Der Unterschied zu $Msgrcv2$ besteht also darin, daß durch Setzen des Flag MSG_EXCEPT der Typ der Nachricht ungleich dem vorgegebenen Typ sein muß.

Der Unterschied zu $Msgrcv1$ ist der Typ, der einem positiven Wert entspricht und nicht auf 0 gesetzt ist.

Beim letzten Erfolgsfall wird die erste Nachricht des kleinsten Typen aus einer Message-Queue gelesen, die kleiner oder gleich einem vorgegebenen Typ ist.

$$
\begin{array}{l}
\textit{Msgrcv4} \\
\hline
\Delta \textit{MsgQStateSpace} \\
p? : \textit{PID} \\
i? : \textit{ID} \\
t? : \textit{TYPE} \\
f? : \mathbb{P}\,\textit{FLAG} \\
\textit{msgsz}? : \mathbb{N} \\
t! : \textit{TYPE} \\
b! : \textit{BYTES} \\
r! : \textit{RETURN} \\
\hline
i? \in \mathrm{dom}\,\textit{msgq} \\
t? < 0 \\
\textit{getuid}(p?)\ \mathrm{mayread}\ (\textit{perm}(i?), \textit{owner}(i?), \textit{group}(i?)) \\
\textit{msgq}(i?) \neq \langle\rangle \wedge \\
(\mathbf{let}\ j == \{b1 : \textit{BYTES};\ t1 : \textit{TYPE} \mid t1 \leq -t? \wedge \\
\quad t1 = \textit{min}(\{t2 : \textit{TYPE};\ b2 : \textit{BYTES} \mid (t2, b2) \in \mathrm{ran}(\textit{msgq}(i?)) \bullet t2\}) \bullet \\
\qquad (t1, b1)\} \bullet \\
\qquad\quad \textit{msgq}(i?) \upharpoonright j \neq \langle\rangle \wedge \\
\qquad (\textit{MSG\_NOERROR} \in f? \vee \\
\qquad\quad \textit{msg\_ts}(i?)(\textit{min}(\mathrm{dom}(\textit{msgq}(i?) \rhd j))) \leq \textit{msgsz}?) \wedge \\
\qquad (t!, b!) = (\mu\, t3 : \textit{TYPE};\ b3 : \textit{BYTES} \mid (t3, b3) = (\textit{msgq}(i?) \upharpoonright j)(1) \bullet \\
\qquad\quad (t3, \textit{truncate}(b3, \textit{msgsz}?))) \wedge \\
\qquad\quad r! = \textit{min}\{\textit{msg\_ts}(i?)(\textit{min}(\mathrm{dom}(\textit{msgq}(i?) \rhd j))), \textit{msgsz}?\} \wedge \\
\\
\qquad\quad \textit{getkey}' = \textit{getkey} \wedge \\
\qquad\quad \textit{msgq}' = \textit{msgq} \oplus \{i? \mapsto \\
\qquad\qquad \textit{squash}(\{\textit{min}(\mathrm{dom}(\textit{msgq}(i?) \rhd j))\} \ntriangleleft \textit{msgq}(i?))\} \wedge \\
\qquad\quad \textit{creator}' = \textit{creator} \wedge \\
\qquad\quad \textit{owner}' = \textit{owner} \wedge \\
\qquad\quad \textit{group}' = \textit{group} \wedge \\
\qquad\quad \textit{perm}' = \textit{perm} \wedge \\
\qquad\quad \textit{qbytes}' = \textit{qbytes} \wedge \\
\qquad\quad \textit{cbytes}' = \textit{cbytes} \oplus \{i? \mapsto \textit{cbytes}(i?) - \\
\qquad\qquad \textit{msg\_ts}(i?)(\textit{min}(\mathrm{dom}(\textit{msgq}(i?) \rhd j)))\} \wedge \\
\qquad\quad \textit{msg\_ts}' = \textit{msg\_ts} \oplus \{i? \mapsto \\
\qquad\qquad \textit{squash}(\{\textit{min}(\mathrm{dom}(\textit{msgq}(i?) \rhd j))\} \ntriangleleft \textit{msg\_ts}(i?))\} \wedge \\
\qquad\quad \textit{errno}' = \textit{errno}) \\
\hline
\end{array}
$$

Dieses mal wird ein Typ erwartet, der kleiner als 0 ist, also ein negativer Wert.

Als Bedingung für die lokale Variable ergibt sich diesmal, daß der Typ kleiner oder gleich dem negativem des vorgegebenen Typen ist. Da der vorgegebene Typ ein negativer Wert ist, muß der Typ also kleiner oder gleich dem absolutwert des vorgegebenen Typen sein.

Und nun ergibt sich in diesem speziellen Fall doch noch einmal eine Abweichung vom üblichen Aufbau der Erfolgsschema. Da nun mehrere verschiedene Typen möglich sind müssen zuerst die Nachrichten mit dem kleinsten Typen genommen werden. Dies ist also eine zusätzliche Bedingung an die lokale Variable.

Durch die Vorgabe eines negativen Wertes für den Typ unterscheidet sich dieser Fall von den restlichen drei Erfolgsfällen. Diesmal muß zuerst noch der Typ gefunden werden, bevor man die erste Nachricht aus der Menge betrachten kann.

Als abschließende Bemerkung wollen wir hier noch einmal erwähnen, daß die ähnliche Darstellung der verschiedenen Fälle für die Verständlichkeit und Vergleichbarkeit wohl doch positiv ist. Im ersten Moment erscheinen die Schemata sehr kompliziert und ähnlich, unterscheiden tun sie sich tatsächlich nur in drei Punkten. Doch hierdurch können Unterschiede wesentlich besser erfaßt werden.

7.2 Fehlerfall

Auch bei dieser Message-Queue Operation ist es wieder so, daß es einige Fehlerfälle gibt, die nicht in unserer Spezifikation enthalten sind.

Diese sind in den vorherigen Kapiteln schon erklärt worden, seien hier aber noch einmal aufgeführt.

- EFAULT: ein Pointer zeigt auf eine nicht exitstierende Adresse. Für dies Behandlung ist der Kernel zuständig.
- EIDRM: Die Message-Queue ist zwischenzeitlich gelöscht worden. Dies wird nicht im IPC-Paket abgefangen.
- EINTR: Der Prozeß wartet auf eine Message-Queue und erhält einen Interrupt. Dies kann in Z wegen der fehlenden Zeitkomponente nicht dargestellt werden.

In diesem Abschnitt werden die spezifizierbaren Fehlerfälle aufgeführt. Es gibt insgesamt vier solcher Fälle, von denen allerdings zwei von der Vorgabe des Typen abhängig sind. Diese beiden Fehlerfälle müssen noch einmal aufgeteilt werden in dieselben vier Unterscheidungen, die wir beim Erfolgsfall schon getroffen haben. Ansonsten würden diese Schemata zu unübersichtlich werden.

Prinzipiell gemeinsam ist dem Aufbau der Schemata, daß dieselben Ein- und Ausgaben wie bei den Erfolgsfällen vorhanden sind. Ebenso gibt es stets mit -1 den für Fehlerfälle typischen Rückgabewert. Auch bleibt der Zustandsraum unverändert mit Ausnahme von des Objektes *errno*, welches auf den Namen des Fehlers gesetzt wird.

Der erste Fehlerfall ist die Möglichkeit, daß die Länge einer Nachricht größer ist, als die vorgegebene maximale Länge ist, und gleichzeitig das Flag $MSG\_NOERROR$ nicht gesetzt ist.

Da dieser Fall von der Nachricht abhängig ist, man diese also erst finden muß, müssen dieselben Fallunterscheidungen wie bei dem Erfolgsfall getroffen werden.

Zuerst also der Fehler $E2BIG$ für den Fall, daß die erste Nachricht gelesen werden sollte und diese zu lang ist.

$msgrcv1E2BIG$

$$
\begin{array}{l}
\Delta MsgQStateSpace \\
p? : PID \\
i? : ID \\
t? : TYPE \\
f? : \mathbb{P}\, FLAG \\
msgsz? : \mathbb{N} \\
t! : TYPE \\
b! : BYTES \\
r! : RETURN \\
\hline
i? \in \mathrm{dom}\, msgq \\
t? = 0 \\
(\mathbf{let}\ j == \{b1 : BYTES;\ t1 : TYPE \bullet (t1, b1)\} \bullet \\
\quad msgq(i?) \upharpoonright j \neq \langle\rangle \land \\
\quad msg\_ts(i?)(min(\mathrm{dom}(msgq(i?) \rhd j))) > msgsz?) \\
MSG\_NOERROR \notin f? \\
r! = -1 \\
\\
errno' = E2BIG
\end{array}
$$

Für die Schema des Fehlers $E2BIG$ sind einige Vorbedingungen immer gleich. So muß stets eine korrekte Message-Queue adressiert sein und das Flag $MSG\_NOERROR$ ist gesetzt. Die Message-Queue hat mindestens eine Nachricht des gewünschten Typens zu enthalten und die erste Nachricht, die dem gewünschten Typen entspricht muß größer sein, als die vorgegebene maximale Länge einer Nachricht.

Die Schema unterscheiden sich auch stets in denselben Punkten. So muß in diesem Fall der vorgegebene Typ 0 sein und es findet für die Menge der möglichen Typen keine Einschränkung statt. Dies entspricht den Vorgaben beim Erfolgsfall $Msgrcv1$.

Wird jedoch das Lesen der ersten Nachricht eines bestimmten Typs aus einer Message-Queue gewünscht, und diese Nachricht ist länger, als die vorgegebene maximale Länge einer Nachricht, so tritt das zweite Schema des Fehlers $E2BIG$ ein.

$$
\begin{array}{|l}
\hline
\textit{msgrcv2E2BIG} \\
\Delta \textit{MsgQStateSpace} \\
p? : \textit{PID} \\
i? : \textit{ID} \\
t? : \textit{TYPE} \\
f? : \mathbb{P}\,\textit{FLAG} \\
\textit{msgsz}? : \mathbb{N} \\
t! : \textit{TYPE} \\
b! : \textit{BYTES} \\
r! : \textit{RETURN} \\
\hline
i? \in \operatorname{dom} \textit{msgq} \\
t? > 0 \\
\textit{MSG\_EXCEPT} \notin f? \\
(\mathbf{let}\ j == \{b1 : \textit{BYTES};\ t1 : \textit{TYPE} \mid t1 = t? \bullet (t1, b1)\} \bullet \\
\quad \textit{msgq}(i?) \upharpoonright j \neq \langle\rangle \land \\
\quad \textit{msg\_ts}(i?)(\textit{min}(\operatorname{dom}(\textit{msgq}(i?) \rhd j))) > \textit{msgsz}?) \\
\textit{MSG\_NOERROR} \notin f? \\
r! = -1 \\
\\
\textit{errno}' = \textit{E2BIG} \\
\hline
\end{array}
$$

Bei diesem Schema entsprechen die Vorgaben dem Erfolgsfall $Msgrcv2$. Es wird ein Typ größer 0 verlangt und gleichzeitig darf das Flag MSG_EXCEPT nicht gesetzt sein. Die Menge der passenden Nachrichten wird durch die Einschränkung auf den vorgegebenen Typen definiert.

Tritt dieser Fehler auf, wenn die erste Nachricht, die nicht einem bestimmten Typ entspricht ausgelesen werden soll, so ist dieses Schema relevant.

$$
\begin{array}{|l}
\hline
\textit{msgrcv3E2BIG} \\
\Delta \textit{MsgQStateSpace} \\
p? : \textit{PID} \\
i? : \textit{ID} \\
t? : \textit{TYPE} \\
f? : \mathbb{P}\,\textit{FLAG} \\
\textit{msgsz}? : \mathbb{N} \\
t! : \textit{TYPE} \\
b! : \textit{BYTES} \\
r! : \textit{RETURN} \\
\hline
i? \in \operatorname{dom} \textit{msgq} \\
t? > 0 \\
\textit{MSG\_EXCEPT} \in f? \\
(\mathbf{let}\ j == \{b1 : \textit{BYTES};\ t1 : \textit{TYPE} \mid t1 \neq t? \bullet (t1, b1)\} \bullet \\
\quad \textit{msgq}(i?) \upharpoonright j \neq \langle\rangle \land \\
\quad \textit{msg\_ts}(i?)(\textit{min}(\operatorname{dom}(\textit{msgq}(i?) \rhd j))) > \textit{msgsz}?) \\
\textit{MSG\_NOERROR} \notin f? \\
r! = -1 \\
\\
\textit{errno}' = \textit{E2BIG} \\
\hline
\end{array}
$$

Entsprechend dem Erfolgsfall *Msgrcv*3 wird hier ein Typ vorausgesetzt, der größer als 0 ist. Dabei muß das Flag *MSG_EXCEPT* gesetzt sein, was der Unterschied zum vorherigen Fall darstellt. Demzufolge wird die Menge der passenden Nachrichten dadurch eingeschränkt, daß sie nicht dem vorgegebenen Typen entsprechen.

Ist die Nachricht nun länger als erlaubt bei den Vorgaben, die $Msgrcv4$ entsprechen, so tritt dieser Fall ein.

$$
\begin{array}{l}
\textbf{schema } msgrcv4E2BIG \\
\Delta MsgQStateSpace \\
p? : PID \\
i? : ID \\
t? : TYPE \\
f? : \mathbb{P}\, FLAG \\
msgsz? : \mathbb{N} \\
t! : TYPE \\
b! : BYTES \\
r! : RETURN \\
\hline
i? \in \operatorname{dom} msgq \\
t? < 0 \\
(\mathbf{let}\ j == \{b1 : BYTES;\ t1 : TYPE \mid t1 \leq -t? \land \\
\quad t1 = min(\{t2 : TYPE;\ b2 : BYTES \mid \\
\qquad (t2, b2) \in \operatorname{ran}(msgq(i?)) \bullet t2\}) \bullet \\
\qquad\quad (t1, b1)\} \bullet \\
\qquad\qquad msgq(i?) \upharpoonright j \neq \langle\rangle \land \\
\qquad\qquad msg\_ts(i?)(min(\operatorname{dom}(msgq(i?) \rhd j))) > msgsz?) \\
MSG\_NOERROR \notin f? \\
r! = -1 \\
errno' = E2BIG
\end{array}
$$

In diesem Fall wird also ein Typ vorausgesetzt, der kleiner als 0 zu sein hat. Die Menge der passenden Nachrichten wird diesmal dadurch eingeschränkt, daß der Typ kleiner als der Absolutbetrag des vorgegebenen Typen zu sein hat.

Auch bei diesem Schema wird ebenso wie bei $Msgrcv4$ die Menge der passenden Nachrichten noch weiter eingeschränkt, indem sie auf die Nachrichten mit dem kleinsten Typen beschränkt wird.

Der zweite Fehlerfall ist das erfolglose Lesen einer Nachricht, da der Prozeß keine Leserechte für die vorgegebene Message-Queue besitzt.

Hierbei ist der vorgegebene Typ nicht weiter von Bedeutung, also muß auch keine weitere Fallunterscheidung gemäß den Erfolgsfällen getroffen werden.

$$
\begin{array}{|l}
\hline
\textit{msgrcvEACCES} \\
\Delta \textit{MsgQStateSpace} \\
p? : \textit{PID} \\
i? : \textit{ID} \\
t? : \textit{TYPE} \\
f? : \mathbb{P}\,\textit{FLAG} \\
\textit{msgsz}? : \mathbb{N} \\
t! : \textit{TYPE} \\
b! : \textit{BYTES} \\
r! : \textit{RETURN} \\
\hline
i? \in \operatorname{dom} \textit{msgq} \\
\neg\,(\textit{getuid}(p?) \text{ mayread } (\textit{perm}(i?), \textit{owner}(i?), \textit{group}(i?))) \\
r! = -1 \\
\\
\textit{errno}' = \textit{EACCES} \\
\hline
\end{array}
$$

Auch bei diesem Fall ist stets Voraussetzung, daß eine korrekte Message-Queue angegeben wurde. Daraufhin darf der Prozeß jedoch keine Leserechte für die Message-Queue besitzen.

Auch dieser Fehlerfall ist von den vorhergehenden Message-Queue Operationen her bekannt. Er tritt ein, wenn ein falscher Wert vorgegeben wurde, sei es für die *ID* oder für die Länge einer Nachricht.

$$
\begin{array}{|l}
\hline
\textit{msgrcvEINVAL} \\
\Delta \textit{MsgQStateSpace} \\
p? : \textit{PID} \\
i? : \textit{ID} \\
t? : \textit{TYPE} \\
f? : \mathbb{P}\,\textit{FLAG} \\
\textit{msgsz}? : \mathbb{N} \\
t! : \textit{TYPE} \\
b! : \textit{BYTES} \\
r! : \textit{RETURN} \\
\hline
i? \notin \operatorname{dom} \textit{msgq} \lor \\
\quad \textit{msgsz}? < 0 \\
r! = -1 \\
\textit{errno}' = \textit{EINVAL} \\
\hline
\end{array}
$$

Die Vorbedingungen sind diesmal ganz einfach. Entweder ist ein Message-Queue Identifier angegeben worden, der nicht vorhanden ist, oder es wurde als maximale Nachrichtenlänge ein negativer Wert angegeben.

Der letzte spezifizierte Fehlerfall ist wieder von der zu lesenden Nachricht abhängig, dementsprechend muß wieder dieselbe Fallunterscheidung wie beim Erfolgsfall getroffen werden.

Auch dieser Fehler ist von den anderen Message-Queue Operationen her bekannt. *NOMSG* kennzeichnet, daß keine Nachricht des vorgegebenen Typen in der angegebenen Message-Queue vorhanden ist.

Fangen wir mit dem ersten Fall an, daß man die erste Nachricht aus dem Schema auslesen wollte. Damit dieser Fehler eintritt muß die Message-Queue in diesem Falle leer sein.

$$
\begin{array}{l}
\textit{msgrcv1NOMSG} \\
\hline
\Delta \textit{MsgQStateSpace} \\
p? : \textit{PID} \\
i? : \textit{ID} \\
t? : \textit{TYPE} \\
f? : \mathbb{P}\, \textit{FLAG} \\
\textit{msgsz}? : \mathbb{N} \\
t! : \textit{TYPE} \\
b! : \textit{BYTES} \\
r! : \textit{RETURN} \\
\hline
i? \in \operatorname{dom} \textit{msgq} \\
t? = 0 \\
\textit{IPC\_NOWAIT} \in f? \land \\
\quad (\textit{msgq}(i?) = \langle\rangle \lor \\
\quad (\mathbf{let}\ j == \{b1 : \textit{BYTES};\ t1 : \textit{TYPE} \bullet (t1, b1)\} \bullet \\
\qquad \textit{msgq}(i?) \upharpoonright j = \langle\rangle)) \\
r! = -1 \\
\textit{errno}' = \textit{ENOMSG} \\
\hline
\end{array}
$$

Bei diesem Fehler gibt es wieder einige Vorbedingungen, die bei allen Fällen vorzuliegen haben. Dies wäre zuerst die Angabe einer korrekten *ID*. Dann muß das Flag *IPC_NOWAIT* gesetzt sein, wäre es nicht gesetzt träte übriegens der nicht spezifizierte Fall des Wartens ein. Die Message-Queue ist entweder leer oder sie darf keine Nachricht des vorgegebenen Typens enthalten.

Zusätzlich gibt es für die Fallunterscheidung wieder die üblichen unterschiedlichen Vorbedingungen. In diesem Fall muß als Typ die 0 vorgegeben sein, eine weiter Einschränkung der Menge der passenden Nachrichten kann daraufhin entfallen.

Tritt nun dieser Fehler ein, wenn man die erste Nachricht eines bestimmten Typen lesen möchte, ist dieses Schema entscheidene.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{msgrcv2NOMSG}} \\
\hline
\Delta MsgQStateSpace \\
p? : PID \\
i? : ID \\
t? : TYPE \\
f? : \mathbb{P}\, FLAG \\
msgsz? : \mathbb{N} \\
t! : TYPE \\
b! : BYTES \\
r! : RETURN \\
\hline
i? \in \operatorname{dom} msgq \\
t? > 0 \\
MSG\_EXCEPT \notin f? \\
IPC\_NOWAIT \in f? \land \\
\quad (msgq(i?) = \langle\rangle \lor \\
\quad (\mathbf{let}\ j == \{b1 : BYTES;\ t1 : TYPE \mid t1 = t? \bullet (t1, b1)\} \bullet \\
\qquad msgq(i?) \upharpoonright j = \langle\rangle)) \\
r! = -1 \\
errno' = ENOMSG \\
\hline
\end{array}
$$

Die unterschiedlichen Vorbedingungen gegenüber den anderen Schemate dieses Fehlers entsprechen wieder dem Erfolgsfall *Msgrcv*2, wobei ein Typ vorgegeben wird, der größer als 0 sein muß und gleichzeitig darf das Flag *MSG_EXCEPT* nicht gesetzt sein. Daraufhin wird die Menge der passenden Nachrichten durch das Testen auf Gleichheit mit dem vorgegebenen Typ beschränkt.

Der dritten Fall, entsprechend $Msgrcv3$, ist der Fall, wenn die erste Nachricht, die nicht einem bestimmten Typ entspricht, gelesen werden soll, und diese länger als die vorgegebene maximale Länge ist.

$$
\begin{array}{|l}
\hline
\textit{msgrcv3NOMSG} \\
\Delta \textit{MsgQStateSpace} \\
p? : \textit{PID} \\
i? : \textit{ID} \\
t? : \textit{TYPE} \\
f? : \mathbb{P}\, \textit{FLAG} \\
\textit{msgsz}? : \mathbb{N} \\
t! : \textit{TYPE} \\
b! : \textit{BYTES} \\
r! : \textit{RETURN} \\
\hline
i? \in \mathrm{dom}\, \textit{msgq} \\
t? > 0 \\
\textit{MSG\_EXCEPT} \in f? \\
\textit{IPC\_NOWAIT} \in f? \land \\
\quad (\textit{msgq}(i?) = \langle\rangle \lor \\
\quad (\mathbf{let}\ j == \{b1 : \textit{BYTES};\ t1 : \textit{TYPE} \mid t1 \neq t? \bullet (t1, b1)\} \bullet \\
\qquad \textit{msgq}(i?) \upharpoonright j = \langle\rangle)) \\
r! = -1 \\
\textit{errno}' = \textit{ENOMSG} \\
\hline
\end{array}
$$

Der Unterschied zum vorherigen Fall ist wieder, daß ein Typ größer 0 vorausgesetzt wird und das Flag MSG_EXCEPT gesetzt sein muß. Entsprechend wird die Menge der passenden Nachrichten durch das Testen auf Ungleichheit beschränkt.

Zum Schluß noch der Versuch eine Nachricht entsprechend dem Fall $Msgrcv4$ zu lesen, wobei diese länger als erlaubt ist.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{msgrcv4NOMSG}}} \\
\Delta MsgQStateSpace \\
p? : PID \\
i? : ID \\
t? : TYPE \\
f? : \mathbb{P}\, FLAG \\
msgsz? : \mathbb{N} \\
t! : TYPE \\
b! : BYTES \\
r! : RETURN \\
\hline
i? \in \operatorname{dom} msgq \\
t? < 0 \\
IPC\_NOWAIT \in f? \land \\
\quad (msgq(i?) = \langle\rangle \lor \\
\quad (\mathbf{let}\ j == \{b1 : BYTES;\ t1 : TYPE \mid t1 \leq -t? \land \\
\qquad t1 = min(\{t2 : TYPE;\ b2 : BYTES \mid \\
\qquad\quad (t2, b2) \in \operatorname{ran}(msgq(i?)) \bullet t2\}) \bullet \\
\qquad\qquad (t1, b1)\} \bullet \\
\qquad\qquad\quad msgq(i?) \upharpoonright j = \langle\rangle)) \\
r! = -1 \\
errno' = ENOMSG \\
\hline
\end{array}
$$

Im Gegensatz zu den vorherigen Fällen wird hier ein Typ vorausgesetzt, der kleiner als 0 ist. Die Menge der passenden Nachrichten wird zuerst auf die eingeschränkt, die kleiner als der Absolutbetrag des vorgegebenen Typens sind und anschließend auf die Nachrichten mit dem kleinsten Typen dieser Menge.

7.3 User-Interface

Die von uns vorgenommene Aufteilung in verschiedene Fälle ist “willkürlich“ und diente wie schon Beschrieben der besseren Lesbarkeit und Verständlichkeit. Nun müssen wir diese Fälle wieder zusammenführen, um das Verhalten der Message-Queue Operation *msgrcv*() zu beschreiben.

Fangen wir mit dem Erfolgsfall an und führen zuerst diese Fälle zusammen. Die Gesamtwirkung der Operation *Msgrcv*, die sämtliche Erfolgsfälle beinhaltet, ergibt sich aus folgender Schemadisjunktion:

$$Msgrcv \mathrel{\widehat{=}} Msgrcv1 \lor Msgrcv2 \lor Msgrcv3 \lor Msgrcv4$$

Sodann ist es sinnvoll alle Fälle zusammenzuführen, die nicht das vom Benutzer gewünschte Ergebnis darstellen. Dieses ergibt sich aus folgender Schemadisjunktion:

$$\begin{aligned} Msgrcvfault \mathrel{\widehat{=}} & msgrcv1E2BIG \lor msgrcv2E2BIG \lor msgrcv3E2BIG \\ & \lor msgrcv4E2BIG \lor msgrcvEACCES \lor msgrcvEINVAL \\ & \lor msgrcv1NOMSG \lor msgrcv2NOMSG \lor msgrcv3NOMSG \\ & \lor msgrcv4NOMSG \end{aligned}$$

Da in *msgrcv*() sowohl der Erfolgsfall als auch der Fehlerfall enthalten ist, müssen wir auch diese beiden Fälle zusammenfügen. Auch hier ist wieder eine Schemadisjunktion möglich, da sämtliche unterschiedlichen Schemata dieselben Ein- und Ausgaben besitzen.

$$msgrcv \mathrel{\widehat{=}} Msgrcv \lor Msgrcvfault$$

Somit entspricht das abstrakt formal spezifizierte *msgrcv* dem nach Aussen sichtbaren Verhalten der Message-Queue Operation *msgrcv*().

8 Message-Queue Operation *msgctl*()

Mit der Message-Queue Operation *msgctl*() können die Daten über eine Message-Queue manipuliert werden.

Auch bei dieser Operation haben wir uns entschlossen zur besseren Lesbarkeit und zur einfacheren Verständlichkeit diese Operation in den Erfolgsfall und den Fehlerfall aufzuteilen und diese jeweils noch weiter zu unterteilen.

Im Erfolgsfall ist es zum einen denkbar eine Message-Queue zu löschen. Desweiteren können Informationen über die Message-Queue ausgegeben werden oder diese Informationen können sogar verändert werden.

Auch die Fehlerfälle haben wir weiter unterteilt. Dabei entstanden drei Fehlerfälle, die aus den anderen Operationen schon bekannt sein dürften.

Zuerst *EACCESS*, der Prozeß hat keine Leserechte für die angegebenen Message-Queue. Weiter geht es mit *EINVAL*, es gab eine falsche Angabe für ein Kommando oder eine falsche Angabe beim Message-Queue Identifier. Und als drittes gibt es noch *EPERM*, der Prozeß hat nicht die entsprechenden Rechte zur Ausführung der Aktion.

Desweiteren gibt es wieder zwei Fehlerfälle, die wir nicht mitspezifiziert haben. Auch diese Fälle sind aus vorherigen Kapiteln schon bekannt und die Begründung für das Nichtspezifizieren ist dort schon mehrfach genannt.

- EFAULT: ein Pointer zeigt auf eine nicht exitstierende Adresse. Für dies Behandlung ist der Kernel zuständig.

- EIDRM: Die Message-Queue ist zwischenzeitlich gelöscht worden. Dies wird nicht im IPC-Paket abgefangen.

Nachdem wir wieder so viele Fallunterscheidungen getroffen haben ist es anschließend notwendig die verschiedenen Fälle zusammenzuführen um das Verhalten von *msgctl*() darzustellen.

Dieses Kapitel beginnt also wieder mit den Erfolgsfällen, die diesmal nicht sehr viele Gemeinsamkeiten aufweisen, woraufhin die Beschreibungen etwas länger werden. Weiter geht es mit den Fehlerfällen, bei denen wir den letzten Fehlerfall der Übersichtlichkeit halber in zwei Schemata aufgeteilt haben. Abschließend wird das User-Interface erstellt.

Insgesamt erhalten wir in diesem Kapitel eine abstrakte formale Spezifikation der Message-Queue Operation *msgctl*().

8.1 Erfolgsfall

Um eine spätere Schemadisjunktion zu ermöglichen haben wieder alle Erfolgsfälle dieselben Eingaben.

Zuerst wird wie üblich der Zustandsraum eingebunden, der Veränderungen unterliegen kann. Die *PID* wird bei der Rechtevergabe benötigt, die *ID* um die Message-Queue zu kennzeichnen.

Mit dem *CMD* wird ein Kommando vorgegeben, das angibt, welcher der drei Erfolgsfälle eintritt.

Desweiteren können die Informationen über die Message-Queue entweder als Eingaben oder als Ausgaben benutzt werden. Hierdurch werden die Schemata auch so lang, aber nicht kompliziert.

Voraussetzung für den Erfolgsfall ist stets, daß mit der *ID* eine existierende Message-Queue angegeben ist.

Zurückgegeben wird stets ein Returnwert, der bei dieser Operation im Erfolgsfalle stets 0 ist.

Die Vor- und Nachbedingungen und die Veränderungen des Zustandraumes sind derart unterschiedlich, daß dieses erst bei den speziellen Schemata erklärt wird.

Dieser Fall ist das erfolgreiche Löschen einer Message-Queue, entsprechende Rechte werden natürlich vorausgesetzt.

Die Folge dieses Schemas ist demnach, das eine Message-Queue mit sämtlichen noch enthaltenen Nachrichten sowie sämtliche Informationen über diese Message-Queue, wie Zugriffsrechte oder Angabe des Erzeugers oder Besitzers, gelöscht werden.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Msgctl1}} \\
\hline
\Delta \mathit{MsgQStateSpace} \\
p? : \mathit{PID} \\
i? : \mathit{ID} \\
\mathit{cmd}? : \mathit{CMD} \\
\mathit{msg\_uid}? : \mathit{UID} \\
\mathit{msg\_gid}? : \mathit{GID} \\
\mathit{msg\_perm}? : \mathbb{P}\, \mathit{PERM} \\
\mathit{msg\_qbytes}? : \mathbb{N} \\
\mathit{msg\_uid}! : \mathit{UID} \\
\mathit{msg\_gid}! : \mathit{GID} \\
\mathit{msg\_perm}! : \mathbb{P}\, \mathit{PERM} \\
\mathit{msg\_qbytes}! : \mathbb{N} \\
r! : \mathit{RETURN} \\
\hline
i? \in \operatorname{dom} \mathit{msgq} \\
\mathit{cmd}? = \mathit{IPC\_RMID} \\
\mathit{getuid}(p?) = \mathit{SUPERUSER} \lor \\
\qquad\qquad \mathit{getuid}(p?) = \mathit{owner}(i?) \lor \\
\qquad\qquad \mathit{getuid}(p?) = \mathit{creator}(i?) \\
r! = 0 \\
\\
\mathit{getkey}' = \mathit{getkey} \setminus \{i? \mapsto \mathit{getkey}(i?)\} \land \\
\mathit{msgq}' = \mathit{msgq} \setminus \{i? \mapsto \mathit{msgq}(i?)\} \land \\
\mathit{creator}' = \mathit{creator} \setminus \{i? \mapsto \mathit{creator}(i?)\} \land \\
\mathit{owner}' = \mathit{owner} \setminus \{i? \mapsto \mathit{owner}(i?)\} \land \\
\mathit{group}' = \mathit{group} \setminus \{i? \mapsto \mathit{group}(i?)\} \land \\
\mathit{perm}' = \mathit{perm} \setminus \{i? \mapsto \mathit{perm}(i?)\} \land \\
\mathit{qbytes}' = \mathit{qbytes} \setminus \{i? \mapsto \mathit{qbytes}(i?)\} \land \\
\mathit{cbytes}' = \mathit{cbytes} \setminus \{i? \mapsto \mathit{cbytes}(i?)\} \land \\
\mathit{msg\_ts}' = \mathit{msg\_ts} \setminus \{i? \mapsto \mathit{msg\_ts}(i?)\} \land \\
\mathit{errno}' = \mathit{errno} \\
\hline
\end{array}
$$

Um auf diesen Fall zu kommen, muß das Kommando $\mathit{IPC\_RMID}$ angegeben sein.

Zudem muß der Prozeß entweder Superuser sein, oder der Eigentümer der Message-Queue oder aber der Erzeuger der Message-Queue.

Daraufhin müssen bei sämtlichen Objekten des Zustandraumes, außer bei $errno$, die Abbildungen mit der ID als Ursprungswert gelöscht werden.

Bei diesem Fall geht es um die erfolgreiche Ausgabe von Informationen der Message-Queues.

Falls man ein Leserecht für die Message-Queue besitzt kann man sich somit den Besitzer, die Gruppe, die Zugriffsrechte sowie die maximale Länge der Message-Queue anzeigen lassen.

$$
\begin{array}{|l}
\hline
\textit{Msgctl2} \\
\Xi \textit{MsgQStateSpace} \\
p? : \textit{PID} \\
i? : \textit{ID} \\
\textit{cmd}? : \textit{CMD} \\
\textit{msg\_uid}? : \textit{UID} \\
\textit{msg\_gid}? : \textit{GID} \\
\textit{msg\_perm}? : \mathbb{P}\, \textit{PERM} \\
\textit{msg\_qbytes}? : \mathbb{N} \\
\textit{msg\_uid}! : \textit{UID} \\
\textit{msg\_gid}! : \textit{GID} \\
\textit{msg\_perm}! : \mathbb{P}\, \textit{PERM} \\
\textit{msg\_qbytes}! : \mathbb{N} \\
r! : \textit{RETURN} \\
\hline
i? \in \operatorname{dom} \textit{msgq} \\
\textit{cmd}? = \textit{IPC\_STAT} \\
\textit{getuid}(p?) \text{ mayread } (\textit{perm}(i?), \textit{owner}(i?), \textit{group}(i?)) \\
r! = 0 \\
\\
\textit{msg\_uid}! = \textit{owner}(i?) \\
\textit{msg\_gid}! = \textit{group}(i?) \\
\textit{msg\_perm}! = \textit{perm}(i?) \\
\textit{msg\_qbytes}! = \textit{qbytes}(i?) \\
\hline
\end{array}
$$

Für diesen Fall ist das Kommando IPC_STAT notwendig.

Genauso wird vorausgesetzt, daß ein Leserecht für die Message-Queue existiert.

Bei diesem Schema ist es nicht möglich, und auch nicht nötig, die Objekte des Zustandraumes zu verändern, schließlich sollen nur Informationen ausgegeben werden. Somit wird der Zustandsraum entsprechend eingebunden und die Objekte brauchen an dieser Stelle nicht aufgeführt zu werden.

Dieser Fall unterscheidet sich von $Msgctl1$ dadurch, daß die Objekte des Zustandraumes nicht verändert werden und statt dessen einige Werte über die Message-Queue zurückgegeben werden.

Der dritte Erfolgsfall ist das Verändern der Informationen über die Message-Queue.

Um solche Veränderungen zu bewirken müssen natürlich entsprechende Rechte vorliegen.

$$
\begin{array}{l}
\textit{Msgctl3} \\
\hline
\Delta \textit{MsgQStateSpace} \\
p? : \textit{PID} \\
i? : \textit{ID} \\
\textit{cmd}? : \textit{CMD} \\
\textit{msg\_uid}? : \textit{UID} \\
\textit{msg\_gid}? : \textit{GID} \\
\textit{msg\_perm}? : \mathbb{P}\, \textit{PERM} \\
\textit{msg\_qbytes}? : \mathbb{N} \\
\textit{msg\_uid}! : \textit{UID} \\
\textit{msg\_gid}! : \textit{GID} \\
\textit{msg\_perm}! : \mathbb{P}\, \textit{PERM} \\
\textit{msg\_qbytes}! : \mathbb{N} \\
r! : \textit{RETURN} \\
\hline
i? \in \operatorname{dom} \textit{msgq} \\
\textit{cmd}? = \textit{IPC\_SET} \\
\textit{getuid}(p?) = \textit{SUPERUSER} \lor \\
\qquad\qquad \textit{getuid}(p?) = \textit{owner}(i?) \lor \\
\qquad\qquad \textit{getuid}(p?) = \textit{creator}(i?) \\
\textit{msg\_qbytes}? \leq \textit{MSGMNB} \lor \textit{getuid}(p?) = \textit{SUPERUSER} \\
r! = 0 \\
\\
\textit{getkey}' = \textit{getkey} \\
\textit{msgq}' = \textit{msgq} \\
\textit{creator}' = \textit{creator} \\
\textit{owner}' = \textit{owner} \oplus \{ i? \mapsto \textit{msg\_uid}? \} \\
\textit{group}' = \textit{group} \oplus \{ i? \mapsto \textit{msg\_gid}? \} \\
\textit{perm}' = \textit{perm} \oplus \{ i? \mapsto \textit{msg\_perm}? \} \\
\textit{qbytes}' = \textit{qbytes} \oplus \{ i? \mapsto \textit{msg\_qbytes}? \} \\
\textit{cbytes}' = \textit{cbytes} \\
\textit{msg\_ts}' = \textit{msg\_ts} \\
\textit{errno}' = \textit{errno}
\end{array}
$$

Für diesen Fall muß als Kommando $\textit{IPC\_SET}$ vorgegeben sein.

Auch muß der Prozeß der Superuser, der Besitzer der Message-Queue oder der Erzeuger der Message-Queue sein.

Möchte man dabei die maximale Länge der Message-Queue verändern, so muß diese entweder kleiner als die vom System vorgegebene maximale Länge sein oder der Prozeß muß Superuser sein.

Je nachdem, welche Eingaben existieren, werden die Objekte des Zustandraumes verändert.

Soll der Besitzer der Message-Queue verändert werden, so wird *owner* beeinflußt, soll die Gruppe der Message-Queue verändert werden, wird bei *group* der entsprechende Wert durch die Vorgabe ersetzt. Bei einer Veränderung der Zugriffsrechte wird *perm* beeinflußt, bei einer Veränderung der maximalen Länge der Message-Queue hingegen *qbytes*.

Die restlichen Objekte bleiben stets unverändert.

Bei diesem Schema geht es im Unterschied zu *Msgctl*2 darum Werte zu verändern, demnach werden die Objekte des Zustandraumes mehr oder weniger verändert.

8.2 Fehlerfall

Auch bei dieser Message-Queue Operation gibt es wieder einige Fehlerfälle, die wir nicht spezifiziert haben. Die Begründung wurde in den vorherigen Kapiteln mehrfach genannt, daher hier nur ein kurzes Aufführen der ebenfalls schon aus den vorherigen Kapiteln bekannten nicht spezifizierten Fehler:

- EFAULT: ein Pointer zeigt auf eine nicht exitstierende Adresse. Für dies Behandlung ist der Kernel zuständig.
- EIDRM: Die Message-Queue ist zwischenzeitlich gelöscht worden. Dies wird nicht im IPC-Paket abgefangen.

Die spezifizierten Fehlerfälle erscheinen ziemlich lang, was daran liegt, daß in allen Fällen dieselben Ein- und Ausgaben wie bei den Erfolgsfällen verwendet werden. Dies ermöglicht eine spätere Schemadisjunktion.

Allen Fehlerfällen ist wieder gleich, daß sie als Returnwert die -1 besitzen. Genauso wird aus dem Zustandsraum einzig *errno* auf den Namen des Fehlers gesetzt, alle anderen Objekte des Zustandraumes bleiben unverändert.

Beim Fehler $EACCESS$ ist das Kommando IPC_STAT gesetzt, der Prozeß möchte also die Informationen über die Message-Queue ausgegeben bekommen. Leider besitzt er keine Leserechte für die Message-Queue.

$$
\begin{array}{|l}
\hline
\textit{msgctlEACCES} \\
\Delta \textit{MsgQStateSpace} \\
p? : \textit{PID} \\
i? : \textit{ID} \\
\textit{cmd}? : \textit{CMD} \\
\textit{msg\_uid}? : \textit{UID} \\
\textit{msg\_gid}? : \textit{GID} \\
\textit{msg\_perm}? : \mathbb{P}\,\textit{PERM} \\
\textit{msg\_qbytes}? : \mathbb{N} \\
\textit{msg\_uid}! : \textit{UID} \\
\textit{msg\_gid}! : \textit{GID} \\
\textit{msg\_perm}! : \mathbb{P}\,\textit{PERM} \\
\textit{msg\_qbytes}! : \mathbb{N} \\
r! : \textit{RETURN} \\
\hline
i? \in \operatorname{dom} \textit{msgq} \\
\textit{cmd}? = \textit{IPC\_STAT} \\
\neg\ (\textit{getuid}(p?) \ \mathrm{mayread}\ (\textit{perm}(i?), \textit{owner}(i?), \textit{group}(i?))) \\
r! = -1 \\
\textit{errno}' = \textit{EACCES} \\
\hline
\end{array}
$$

Voraussetzung ist, daß eine korrekte Message-Queue adressiert ist und das Kommando IPC_STAT gesetzt ist.

Desweiteren darf der Prozeß keine Leserechte für die Message-Queue besitzen.

EINVAL ist der schon bekannte Fall, daß eine falsche Angabe gemacht wird, entweder existiert die Message-Queue nicht oder das Kommando existiert nicht. Für diesen Fall ist dieses Schema entsprechend.

msgctlEINVAL

$\Delta MsgQStateSpace$
$p? : PID$
$i? : ID$
$cmd? : CMD$
$msg\_uid? : UID$
$msg\_gid? : GID$
$msg\_perm? : \mathbb{P}\, PERM$
$msg\_qbytes? : \mathbb{N}$
$msg\_uid! : UID$
$msg\_gid! : GID$
$msg\_perm! : \mathbb{P}\, PERM$
$msg\_qbytes! : \mathbb{N}$
$r! : RETURN$

$i? \notin \operatorname{dom} msgq \lor$
$cmd? \notin CMD$
$r! = -1$

$errno' = EINVAL$

Als Vorbedingung für dieses Schema ist entweder die angegebenen *ID* nicht vorhanden oder das angegebene Kommando ist nicht korrekt.

Für die bessere Verständlichkeit haben wir uns entschieden den Fehler *EPERM* in zwei Schemata aufzuteilen. Er beschreibt die Situation, das ein Prozeß nicht die notwendigen Rechte für die Ausführung der Aktion besitzt.

Ein weiterer Grund für diese Aufteilung ist der Source-Code, in dem ebenfalls zwei Fallunterscheidungen für diesen Fehler vorhanden sind.

Dieser Fall tritt ein, wenn bei *Msgctl*1 oder *Msgctl*3 der Prozeß nicht die notwendigen Rechte besitzt, er also weder Superuser noch Eigentümer noch Erzeuger der Message-Queue ist.

$$
\begin{array}{|l}
\hline
\textit{msgctl1EPERM} \\
\Delta \textit{MsgQStateSpace} \\
p? : \textit{PID} \\
i? : \textit{ID} \\
\textit{cmd}? : \textit{CMD} \\
\textit{msg\_uid}? : \textit{UID} \\
\textit{msg\_gid}? : \textit{GID} \\
\textit{msg\_perm}? : \mathbb{P}\, \textit{PERM} \\
\textit{msg\_qbytes}? : \mathbb{N} \\
\textit{msg\_uid}! : \textit{UID} \\
\textit{msg\_gid}! : \textit{GID} \\
\textit{msg\_perm}! : \mathbb{P}\, \textit{PERM} \\
\textit{msg\_qbytes}! : \mathbb{N} \\
r! : \textit{RETURN} \\
\hline
i? \in \operatorname{dom} \textit{msgq} \\
\textit{cmd}? = \textit{IPC\_SET} \lor \textit{cmd}? = \textit{IPC\_RMID} \\
\textit{getuid}(p?) \neq \textit{SUPERUSER} \land \\
\quad \textit{getuid}(p?) \neq \textit{owner}(i?) \land \\
\quad \textit{getuid}(p?) \neq \textit{creator}(i?) \\
r! = -1 \\
\\
\textit{errno}' = \textit{EPERM} \\
\hline
\end{array}
$$

Entsprechend dem einführendem Text muß das Kommando *IPC_SET* oder *IPC_RMID* gesetzt sein.

Und der Prozeß darf keine Rechte als Superuser, Message-Queue Eigentümer oder Message-Queue Erzeuger besitzen.

Ist hingegen das Kommando IPC_SET gesetzt, msg_qbytes ist größer als $MSGMNB$, und der Prozeß hat keine $SUPERUSER$–Rechte tritt dieser Fall ein.

$$
\begin{array}{|l}
\hline
\textit{msgctl2EPERM} \\
\Delta \mathit{MsgQStateSpace} \\
p? : \mathit{PID} \\
i? : \mathit{ID} \\
cmd? : \mathit{CMD} \\
msg\_uid? : \mathit{UID} \\
msg\_gid? : \mathit{GID} \\
msg\_perm? : \mathbb{P}\,\mathit{PERM} \\
msg\_qbytes? : \mathbb{N} \\
msg\_uid! : \mathit{UID} \\
msg\_gid! : \mathit{GID} \\
msg\_perm! : \mathbb{P}\,\mathit{PERM} \\
msg\_qbytes! : \mathbb{N} \\
r! : \mathit{RETURN} \\
\hline
i? \in \operatorname{dom} msgq \\
cmd? = \mathit{IPC\_SET} \\
getuid(p?) \neq \mathit{SUPERUSER} \\
msg\_qbytes? > \mathit{MSGMNB} \\
r! = -1 \\
errno' = \mathit{EPERM} \\
\hline
\end{array}
$$

Als Kommando wird IPC_SET vorausgesetzt, der Prozeß hat keine Superuser-Rechte und die gewünschte maximale Länge der Message-Queue ist größer als die vom System vorgegebene maximale Länge.

Dieses unterscheidet sich in soweit vom vorherigem Schema, als daß es um ein anderes Kommando geht und der Eigentümer und der Erzeuger der Message-Queue dieses Recht nicht besitzen.

8.3 User-Interface

In diesem Abschnitt setzen wir die Fallunterscheidungen wieder zu dem einen Fall zusammen. Dies wird durch die Verwendung derselben Ein- und Ausgaben ermöglicht.

Mit *Msgctl* wollen wir wieder den Erfolgsfall der Message-Queue Operation *msgctl*() bezeichnen, der sich aus der Schemadisjunktion der einzelnen Erfolgsfälle ergibt:

$$Msgctl \mathrel{\widehat{=}} Msgctl1 \lor Msgctl2 \lor Msgctl3$$

Die Gesamtzahl der Fehler ergibt sich aus der Schemadisjunktion:

$$\begin{aligned} Msgctlfault \mathrel{\widehat{=}} {} & msgctlEACCES \lor msgctlEINVAL \lor msgctl1EPERM \lor \\ & msgctl2EPERM \end{aligned}$$

Für den Benutzer des Systems stellt sich selbst diese Aufteilung in Erfolgs- und Fehlerfall nicht. Demnach führen wir diese auch noch zusammen:

$$msgctl \mathrel{\widehat{=}} Msgctl \lor Msgctlfault$$

Somit entspricht das abstrakte formal spezifizierte *msgctl* dem nach Aussen sichtbaren Verhalten der Message-Queue Operation *msgctl*().

9 Ergebnis

Wir haben während der Spezifikation sehr viel gelernt über die Spezifikationssprache Z und Spezifikation im allgemeinen. Und wir haben die üblichen Probleme einer formalen Spezifikation kennengelernt.

Es entsteht eigentlich immer ein zeitliches Problem, die formale Spezifikation wird immer wesentlich aufwendiger, als man es sich am Anfang erdacht hat.

Auch sind die Vorgaben nicht immer eindeutig. Obwohl es bei uns anders war, als die Erstellung einer Spezifikation normalerweise abläuft, so waren doch einige Aussagen in der online-Dokumentation nicht eindeutig und wir mußten den Source-Code zu rate ziehen bzw. das Systemverhalten testen.

Wir hatten *msgq* als eine partielle Abbildung von der Menge der keys in die Menge der Message-Queues spezifiziert. Dabei hatten wir angenommen, daß bei einem *msgget* mit dem KEY *IPC_PRIVATE* ein neuer, noch nicht vorhandener Key gewählt wird. In der Implementierung von Linux und Solaris ist dies jedoch nicht so, sondern es wird als Key *IPC_PRIVATE* bzw. 0 gesetzt. Dieses haben wir über das Testen mit kleinen Programmen herausgefunden.

Diese war nur ein Beispiel für einen Fehler, der zu kleinen Änderungen führte. Allerdings hatten wir auch einen Fehler mit eingebaut, für dessen Behebung die ganze Spezifikation umgeschrieben werden mußte.

Wir waren davon ausgegangen, daß die *ID*, mit denen man in *msgsnd*, *msgrcv* und *msgctl* die Message-Queue adressiert, jeweils für einen Prozeß vergeben wird. Dies ist nicht so, sondern die *ID* für eine Message-Queue wird systemweit einheitlich vergeben. Somit kann ein Prozeß sogar auf eine Message-Queue zugreifen, wenn er gar kein *msgget* vorher aufgerufen hat, ihm muß nur die *ID* bekannt sein und er muß die nötigen Rechte besitzen.

Demnach mußten wir viele Objekte des Zustandraumes abändern, denn bisher waren wir davon ausgegangen, daß Message-Queues stets über ihren *KEY* adressiert würden. Tatsächlich wird die Message-Queue aber über die systemweit einheitlich vergebene *ID* angesprochen. Dies hatte natürlich Auswirkung auf die gesamte Spezifikation und diese mußte vollständig überarbeitet werden.

Diese und weitere Fehler sind inzwischen vollständig eleminiert. Natürlich haben wir dann auch überprüft, ob unsere Spezifikation denn auch korrekt und vollständig ist.

Zuerst haben wir unsere Spezifikation auf syntaktische Korrektheit geprüft. Hierfür gibt es Hilfsmittel, wir haben das Tool Fuzz benutzt. Dieses ist ein Typchecker, der auf dem LaTeX-Source eines Dokumentes arbeitet und die in bestimmten Umgebungen gesetzten Z-Spezfikation checkt.

Nach einem Durchlauf der hier vorliegenden Spezifikation teilt uns Fuzz mit, daß die Spezifikation syntaktisch korrekt ist.

Fuzz ist aber ein Typechecker, er prüft also zusätzlich, ob an sämtlichen Stellen die Typen korrekt verwendet werden. Auch dieses ist der Fall.

Nun könnte es natürlich noch sein, daß ein Fall existiert, den wir nicht spezifiziert haben und somit unsere Spezifikation nicht vollständig wäre. Leider gibt es hierfür keine Toolunterstützung, somit haben wir selbst Tabellen erstellt, die die Vorbedingungen der einzelnen Schemata darstellen. Bei einer Überprüfung haben wir erkannt, daß lediglich die Fälle, die wir in Z wegen der fehlenden Zeitkomponente nicht zu spezifizieren waren, fehlen. Diese sind in den einzelnen Kapiteln jeweils aufgeführt, insbesondere handelt es sich um den Fall, wenn das Flag $IPC_N OWAIT$ nicht gesetzt ist.

Diese Tabellen wollen wir natürlich nicht vorenthalten und stellen sie hier vor.

Es folgt zuerst die Tabelle mit den Vorbedingungen für die verschiedenen Msgget-Operationen, dann geht es mit den Msgsnd-Operationen weiter bevor wir die Vorbedingungen für die Msgrcv-Operationen darlegen und wir schließen mit der Tabelle über die Msgctl-Operationen.

Operation	KEY	FLAGS
Msgget1	nicht vorhanden $\wedge \neq$ IPC_PRIVATE	IPC_CREATE
Msgget2	$\neq$ IPC_PRIVATE	$\neg$ (IPC_CREATE $\wedge$ IPC_EXCL)
Msgget3	IPC_PRIVATE	
msggetEACCESS	$\neq$ IPC_PRIVATE	
msggetEEXIST	$\neq$ IPC_PRIVATE	IPC_CREATE $\wedge$ IPC_EXCL
msggetENOENT	nicht vorhanden $\wedge \neq$ IPC_PRIVATE	$\neg$ IPC_CREATE
msggetENOSPC	IPC_PRIVATE	
msggetENOSPC	nicht vorhanden $\wedge \neq$ IPC_PRIVATE	IPC_CREATE

Operation	SIZE	WRITE/READ
Msgget1	$<$ MSGMNI	
Msgget2		maywrite $\vee$ mayread
Msgget3	$<$ MSGMNI	
msggetEACCESS		$\neg$ maywrite $\wedge$ $\neg$ mayread
msggetEEXIST		
msggetENOENT		
msggetENOSPC		
msggetENOSPC	$\geq$ MSGMNI	

Operation	ID	msgsz	qbytes
Msgsnd	vorhanden	$(msgsz \geq 0) \wedge (msgsz \leq MSGMAX)$	$msgsz + cbytes \leq qbytes$
msgsndEAGAIN	vorhanden		$msgsz + cbytes > qbytes$
msgsndEACCESS	vorhanden		
msgsdEINVAL	nicht vorhanden		
msgsdEINVAL	vorhanden	$(msgsz < 0) \vee (msgsz > MSGMAX)$	
msgsdEINVAL	vorhanden		

Operation	TYPE	WRITE/READ
Msgsnd	$TYPE > 0$	maywrite
msgsndEAGAIN		
msgsndEACCESS		$\neg$ maywrite
msgsdEINVAL		
msgsdEINVAL		
msgsdEINVAL	$TYPE \leq 0$	

Operation	ID	FLAGS	SIZE
Msgrcv1	vorhanden		MSG_NOERROR $\vee$ length msg $\leq$ msgsz
Msgrcv2	vorhanden	$\neg$ MSG_EXCEPT	MSG_NOERROR $\vee$ length msg $\leq$ msgsz
Msgrcv3	vorhanden	MSG_EXCEPT	MSG_NOERROR $\vee$ length msg $\leq$ msgsz
Msgrcv4	vorhanden		MSG_NOERROR $\vee$ length msg $\leq$ msgsz
msgrcv1E2BIG	vorhanden		$\neg$ MSG_NOERROR $\wedge$ length msg $>$ msgsz
msgrcv2E2BIG	vorhanden	$\neg$ MSG_EXCEPT	$\neg$ MSG_NOERROR $\wedge$ length msg $>$ msgsz
msgrcv3E2BIG	vorhanden	MSG_EXCEPT	$\neg$ MSG_NOERROR $\wedge$ length msg $>$ msgsz
msgrcv4E2BIG	vorhanden		$\neg$ MSG_NOERROR $\wedge$ length msg $>$ msgsz
msgrcvEACCESS	vorhanden		
msgrcvEINVAL msgrcvEINVAL	nicht vorhanden vorhanden		 msgsz $<$ 0
msgrcv1NOMSG	vorhanden	IPC_NOWAIT	
msgrcv2NOMSG	vorhanden	IPC_NOWAIT $\wedge$ $\neg$ MSG_EXCEPT	
msgrcv3NOMSG	vorhanden	IPC_NOWAIT $\wedge$ MSG_EXCEPT	
msgrcv4NOMSG	vorhanden	IPC_NOWAIT	

Operation	TYPE	msgq(i)	msgq $\upharpoonright$ j	WRITE/READ
Msgrcv1	TYPE = 0	$\neq \{\}$	$\neq \{\}$	mayread
Msgrcv2	TYPE > 0	$\neq \{\}$	$\neq \{\}$	mayread
Msgrcv3	TYPE > 0	$\neq \{\}$	$\neq \{\}$	mayread
Msgrcv4	TYPE < 0	$\neq \{\}$	$\neq \{\}$	mayread
msgrcv1E2BIG	t = 0	$\neq \{\}$		
msgrcv2E2BIG	t > 0	$\neq \{\}$		
msgrcv3E2BIG	t > 0	$\neq \{\}$		
msgrcv4E2BIG	t < 0	$\neq \{\}$		
msgrcvEACCESS				$\neg$ mayread
msgrcvEINVAL msgrcvEINVAL				
msgrcv1NOMSG	t = 0	$= \{\}$	$= \{\}$	
msgrcv2NOMSG	t > 0	$= \{\}$	$= \{\}$	
msgrcv3NOMSG	t > 0	$= \{\}$	$= \{\}$	
msgrcv4NOMSG	t < 0	$= \{\}$	$= \{\}$	

Operation	ID	CMD	UID
Msgctl1	vorhanden	IPC_RMID	Superuser $\vee$ owner $\vee$ creator
Msgctl2	vorhanden	IPC_STAT	
Msgctl3	vorhanden	IPC_SET	Superuser $\vee$ owner $\vee$ creator
msgctlEACCESS	vorhanden	IPC_STAT	
msgctlEINVAL	nicht vorhanden		
msgctlEINVAL		nicht vorhanden	
msgctl1EPERM	vorhanden	IPC_RMID $\vee$ IPC_SET	$\neg$ (Superuser $\vee$ owner $\vee$ creator)
msgctl2EPERM	vorhanden	IPC_SET	

Operation	MSGMNB	WRITE/READ
Msgctl1		
Msgctl2		mayread
Msgctl3	(msgq_qbytes $\leq$ MSGMNB) $\vee$ Superuser	
msgctlEACCESS		$\neg$ mayread
msgctlEINVAL		
msgctlEINVAL		
msgctl1EPERM		
msgctl2EPERM	(msgq_qbytes $>$ MSGMNB) $\wedge$ $\neg$ Superuser	

Eine Überprüfung dieser manuell erstellten Tabellen ist auch nur manuell möglich, aber wie schon Beschrieben, es werden sämtliche möglichen Fälle abgedeckt.

Ein Fazit für uns ist, daß wir sehr viel gelernt haben über Spezifikation im Allgemeinen und speziell über die Spezifikationssprache Z, dabei sind uns weder die negativen noch die positiven Seiten verborgen geblieben. Natürlich haben wir auch sehr viel über das IPC-Package und speziell die Message-Queues gelernt.

Als Ergebnis für die Allgemeinheit bleibt ein vielleicht sinnvoller Beitrag für die Linux-Gemeinde.

Und natürlich eine vollständige und korrekte abstrakte formale Spezifikation der Message-Queues.