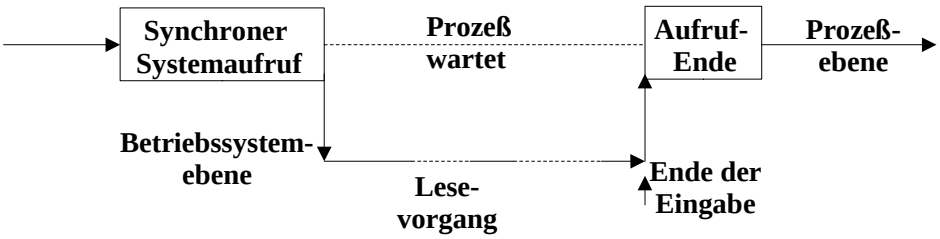
Anhang A: Asynchrone Systemaufrufe

A.1. Einleitung

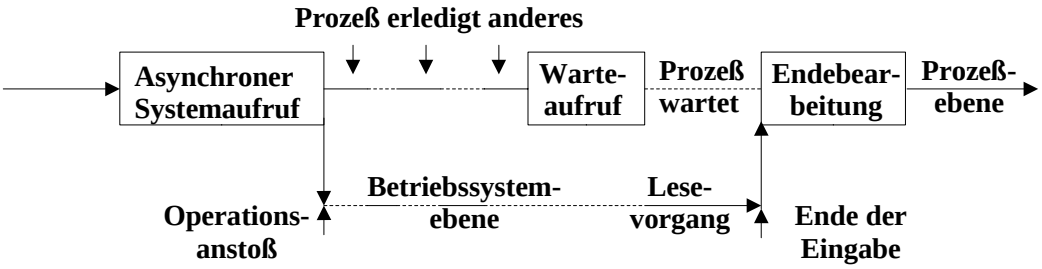
In Unix wie bei anderen Betriebssystemen gibt es eine Reihe von Systemaufrufen, die den aufrufenden Prozeß bis zum Eintritt eines bestimmten Ereignisses blockieren. In manchen Anwendungen ist es aber wünschenswert, normalerweise blockierende Systemaufrufe asynchron abgeben zu können, d. h., daß der aufrufende Prozeß zu den durch den Systemaufruf veranlaßten Aktionen des Betriebssystems parallel weiterarbeiten kann, also eben nicht blockiert wird. Er muß dann in geeigneter Weise von der Beendigung des asynchronen Systemaufrufs (= Eintritt des erwarteten Ereignisses) benachrichtigt werden. Anschließend muß er die Ergebnisse des asynchronen Systemaufrufs (z. B. gelesene Daten, Fehlercode) übernehmen können.

Man unterscheidet hierbei zwischen *implizitem* und *explizitem Warten*: Ein normaler (synchroner) Systemaufruf (z. B. ein Leseaufruf) führt zu *implizitem* Warten, denn der Prozeß wird automatisch bis zur Beendigung der eingeleiteten Operation blockiert, ohne daß der Programmierer sich weiter darum kümmern muß:



Systemaufruf mit implizitem Warten

Dies ist der Normalfall unter Unix bzw. Linux. - Bei asynchronen Systemaufrufen hingegen muß nach Anstoß der asynchronen Operation irgendwann *explizit* gewartet werden, d. h., der Prozeß muß explizit an das Betriebssystem einen Warteaufruf und einen Aufruf zur Endebearbeitung sowie Ergebnisübernahme abgeben, sonst erfährt er nicht, ob und mit welchem Ergebnis (z. B. gelesene Daten) die eingeleitete asynchrone Operation beendet wurde:



Systemaufruf mit explizitem Warten, Operationsende nach Warteaufruf

Wenn die angestoßene asynchrone Operation schon vor Abgabe des Warteaufrufs beendet wurde, wird der Prozeß durch den Warteaufruf natürlich nicht mehr blockiert, und er kann das Ergebnis sofort abholen.

A.2. Benötigte Systemdienste

Wie bei der Unterscheidung zwischen implizitem und explizitem Warten schon angeklungen ist, werden zur Realisierung von asynchronen Operationen gegenüber dem Synchronfall zusätzliche Systemdienste benötigt. Dabei handelt es sich grundsätzlich um die nachfolgend aufgeführten Dienste, die in den folgenden Abschnitten im einzelnen behandelt werden und zu denen nach Bedarf weitere kommen können. Sie sollen im Praktikum realisiert werden:

-- Start der asynchronen Operation (also Anstoß eines Ausgabevorgangs, Vorbereitung des Betriebssystems auf eine Eingabe, Erzeugung eines Subprozesses usw.),
-- Gleichzeitiger Start mehrerer asynchroner Operationen durch Übergabe einer Liste solcher Operationen,
-- Abbruch (Rücknahme) einer gestarteten asynchronen Operation,
-- Resynchronisation von Prozeß und asynchroner Operation mit drei Varianten:
   -- durch expliziten Warteaufruf,
   -- durch Status-Abfrage (bedingten Aufruf),
   -- durch Auslösung eines asynchronen Unterprogramms (ähnlich wie eine Signalbehandlung),
-- Übernahme des Ergebnisses bzw. einer Fehlerkennung aus der asynchronen Operation.

In (Standard-)UNIX arbeitet unter den Systemaufrufen nur *fork ()* für die Prozeßverzweigung asynchron, wenn man von der Ausnahme bestimmter Ein/Ausgabe-Aufrufe unter eng eingegrenzten Umständen (BSD-Versionen) absieht. In allen anderen Fällen muß die Asynchronität eines Systemaufrufes durch "Verpacken" in einen anderen Prozeß (und damit indirekt wieder durch *fork*) erreicht werden.

A.2.1. Start der asynchronen Operation

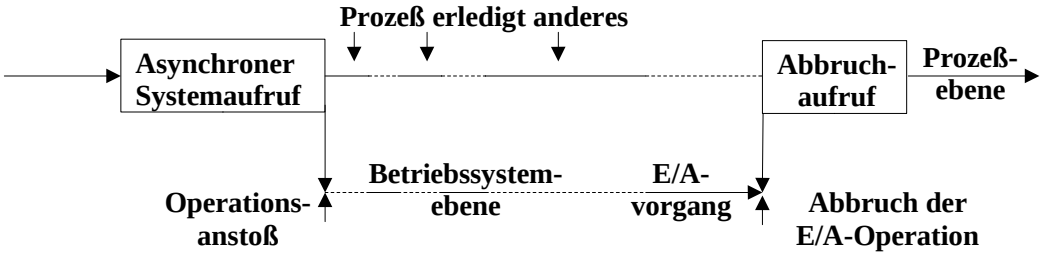
Durch einen Systemaufruf, der gegenüber der synchronen (also den Prozeß möglicherweise blockierenden) Variante des Systemaufrufs entsprechend modifiziert ist, wird die gewünschte asynchrone Operation ausgelöst (z. B. Ausgabe eines Datenpuffers auf ein peripheres Gerät) oder zur Ausführung angemeldet (z. B. eine Eingabeoperation). Dabei bekommt der aufrufende Prozeß für spätere Referenzzwecke entweder vom Betriebssystem eine Aufrufkennung mitgeteilt (ganze Zahl oder Bitmuster), oder er gibt dem Aufruf eine selbst gewählte Kennung an das Betriebssystem mit. Hierfür verwendet man eine eindeutig bestimmte, für den Aufruf charakteristische Konstante, etwa die Adresse des Parameterblocks für den Systemaufruf oder die Identifikation eines Betriebsmittels zur Interprozeßkommunikation und Resynchronisation (wie Semaphor, Briefkasten, *Pipe*), mit dessen Hilfe später die Resynchronisation durchgeführt werden soll.

Im Gegensatz zur synchronen Variante des jeweiligen Systemaufrufs wird der aufrufende Prozeß im Asynchronfall nach Aufrufabgabe sofort fortgesetzt, auch wenn die angestoßene Operation zu Wartezuständen führt. Für die Ausführung eines asynchronen Systemaufrufs wird parallel zum aufrufenden Prozeß Code des Betriebssystems abgearbeitet.

A.2.2. Abbruchaufruf

Bei Bedarf (z. B. nach Ablauf einer Zeitschranke) kann mit einem Abbruchaufruf *(CANCEL, ABORT)* der Abbruch (die Rücknahme) einer angestoßenen asynchronen Operation veranlaßt werden, zu deren Identifikation dem Abbruchaufruf die Aufrufkennung mitzugeben ist. Sinnvollerweise sollte nach Ausführung des Abbruchs mit einem weiteren Systemaufruf das Ergebnis der ab-

gebrochenen Operation abgeholt werden, etwa um eine Teilausführung erkennen oder mittels einer Fehlermeldung das Problem diagnostizieren zu können, das zum Abbruch geführt hat. Ein Abbruchaufruf, der nach Beendigung des zugehörigen asynchronen Aufrufs abgegeben wird, wird ignoriert.



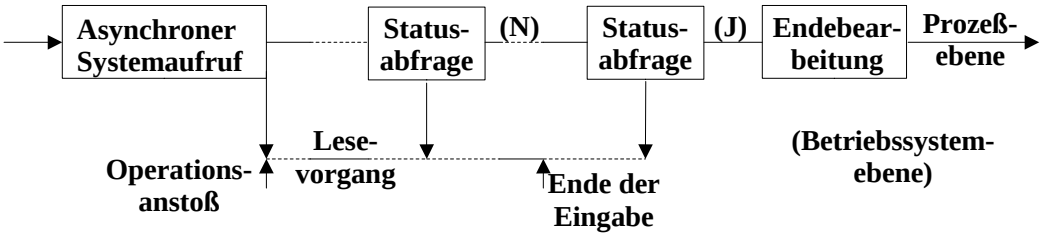
Abbruch einer asynchronen Operation

A.2.3. Resynchronisation

Wenn die asynchrone Operation nicht zwischenzeitlich abgebrochen, sondern normal beendet wurde, muß sich der Prozeß mit der asynchronen Operation resynchronisieren. Solange die Operation noch nicht abgeschlossen ist, bedeutet das, daß der Prozeß doch irgendwann auf den Abschluß warten muß. Für die Resynchronisation gibt es mehrere Möglichkeiten. - Im folgenden wird unter Abschluß-Ereignis der Abschluß einer vorher angestoßenen asynchronen Operation verstanden.

A.2.3.1. Resynchronisation durch Status-Abfrage (bedingter Aufruf)

Nach der Aufrufabgabe für eine Status-Abfrage wird der aufrufende Prozeß unmittelbar fortgesetzt, auch wenn die angestoßene Operation noch andauert. Er erfährt aus dem Aufrufergebnis den Operationsstatus, also, ob die in der Aufrufkennung angegebene Operation abgeschlossen ist oder nicht, und kann sich bei Bedarf bis zur nächsten Status-Abfrage anderen Tätigkeiten widmen. Wenn mit der Status-Abfrage im Falle des Operationsabschlusses auch die Ergebnis-Übernahme verbunden ist, spricht man von einem bedingten Abschluß-Aufruf.



Resynchronisation durch Status-Abfragen

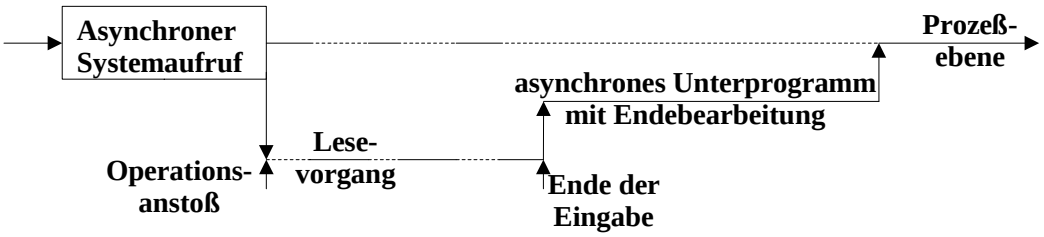
Grundsätzlich kann durch wiederholte Status-Abfragen ein Warten des Prozesses auf den Operationsabschluß vermieden werden, aber das ist nur so lange sinnvoll, wie der Prozeß zwischen zwei Abfragen andere sinnvolle Aufgaben erledigen kann, andernfalls kommt es zum aktiven Warten durch laufende Status-Abfragen innerhalb einer Schleife, das ja gerade vermieden werden soll.

A.2.3.2. Resynchronisation durch Warteaufruf

Hierbei wird der aufrufende Prozeß nach Aufrufabgabe bis zum Abschluß der in der Aufrufkennung angegebenen Operation blockiert. Wenn die Operation jedoch schon abgeschlossen ist, wird er unmittelbar fortgesetzt. Diese Variante ist in der Abbildung mit dem expliziten Warten in Abschnitt A.1 illustriert. Falls das Aufrufergebnis mittels Interprozeßkommunikation übermittelt wird, bieten sich als Warteaufrufe die entsprechenden blockierenden Empfangsaufrufe an.

A.2.3.3. Resynchronisation durch asynchrones Unterprogramm

Diese Resynchronisationsform entspricht dem Aufruf eines Signalbehandlungsprogramms unter Unix. Das asynchrone Unterprogramm läuft im Adreßraum des aufrufenden Prozesses ab und wird im allgemeinen den Aufruf zur Ergebnisübernahme abgeben, den aufrufenden Prozeß z. B. durch Ablegen einer Kennung im gemeinsamen Adreßraum verständigen und vielleicht weitere anwendungsabhängige Aktionen durchführen. Diese Variante kann mit der Abgabe eines Warteaufrufes kombiniert werden; beim Deblockieren des wartenden Prozesses wird zunächst das vorgesehene asynchrone Unterprogramm ausgeführt und danach erst der eigentliche Prozeß fortgesetzt.



Resynchronisation durch Auslösung eines asynchronen Unterprogramms

In den *BSD*-Varianten kann von einem E/A-bereiten Gerät *SIGIO* ausgelöst werden (geht nur mit *Terminals* und Netzverbindungen); dies muß mit einem *fcntl*-Aufruf (Konstante *F_SETOWN*) veranlaßt werden. In *System V.4* wird *SIGPOLL* gesendet; dies funktioniert nur mit *Streams*-Geräten (s. Abschnitt 7.2.3) und wird mit einem *ioctl*-Aufruf (Konstante *I_SETSIG*) angefordert. Die durch *SIGIO* bzw. *SIGPOLL* ausgelöste Signalbehandlungsfunktion entspricht dem hier behandelten asynchronen Unterprogramm.

A.2.4. Übernahme des Ergebnisses aus der asynchronen Operation

Erst dieser Systemaufruf schließt die mit der Aufrufkennung identifizierte asynchrone Operation endgültig ab und gibt die dafür allozierten Betriebsmittel wieder frei. Der Prozeß bekommt das Operationsergebnis mitgeteilt. In den obigen Abbildungen ist dieser Aufruf als "Endebearbeitung" gekennzeichnet worden. Er kann zwar mit den Resynchronisationsaufrufen oder dem Abbruchaufruf kombiniert, sollte aber gedanklich von diesen getrennt betrachtet werden, da das zu übernehmende Ergebnis beträchtlich von der gestarteten asynchronen Operation abhängt und Resynchronisationsaufruf bzw. Abbruchaufruf auch in der Lage sein müssen, eigene Ergebnisse zurückzugeben (z. B. ungültige Aufrufkennung).

Das zurückgelieferte Ergebnis entspricht im wesentlichen dem der zugehörigen synchronen Operation (z. B. Anzahl der übertragenen Zeichen, Datenpuffer in Form von Anfangsadresse und Länge,

Fehlerkennungen usw.); zusätzlich sollte die Möglichkeit bestehen, asynchronitätsspezifische Meldungen zurückzugeben (z. B. Aufrufkennung unbekannt oder Operation noch nicht beendet, falls der Ergebnis-Aufruf vor der Resynchronisation abgegeben wurde).

Die Ergebnisübergabe kann auch mit Hilfe eines der Verfahren für die Interprozeßkommunikation geschehen. Hierzu richtet der aufrufende Prozeß z. B. einen Botschaftskanal ein, dessen Kennung dem Startaufruf für die asynchrone Operation mitgegeben und über den ihm das Operationsergebnis mitgeteilt wird. Die Resynchronisation läßt sich dann auf die entsprechenden Empfangsaufrufe für Botschaftskanäle (bedingt oder wartend) zurückführen.

A.3. Mehrfachwartestellen

Ein zentrales Konzept bei der Programmierung paralleler Vorgänge stellt das der Mehrfachwartestelle dar. Ein Prozeß, der mehrere asynchrone Operationen angestoßen hat, muß in der Lage sein, die anschließende Resynchronisation effizient durchführen zu können. Es reicht nämlich nicht, daß der Prozeß auf den Abschluß nur einer der Operationen wartet; in der Zwischenzeit kann ja eine andere Operation beendet worden sein und muß mit einer möglicherweise zeitkritischen Reaktion bedient werden. Das Betriebssystem muß also die Möglichkeit vorsehen, einen Prozeß auf den Abschluß mehrerer Operationen gleichzeitig warten zu lassen. Lösungsansätze:

A.3.1. Abtasten der Ereignisse (Polling)

Der Prozeß läuft nach der Abgabe des asynchronen Systemaufrufs weiter und fragt von Zeit zu Zeit mit einem Statusaufruf (also einem bedingten Aufruf) ab, ob das durch den asynchronen Systemaufruf angekündigte oder angestoßene Ereignis inzwischen eingetreten ist (Abtasten oder *Polling*).

Solange dies nicht der Fall ist, wird der Statusaufruf mit einer entsprechenden Meldung sofort beendet. Ist das Ereignis eingetreten, muß der angestoßene Vorgang durch eine Endebearbeitung mit der Ergebnis-Übergabe abgeschlossen werden, und der Prozeß kann anschließend auf das Ereignis in der erforderlichen Weise reagieren.

Diese Vorgehensweise ist brauchbar, sofern der Prozeß nur *ein* Ereignis angestoßen hat: Solange er noch andere Aufgaben zu erledigen hat, kann er den Ereignisstatus gelegentlich mit dem bedingten Systemaufruf abfragen; wenn er nichts anderes mehr zu tun hat, kann er bis zum Eintreten des Ereignisses mit dem entsprechenden expliziten Warteaufruf den Prozessor abgeben und anschließend darauf reagieren.

Sobald der Prozeß jedoch mehrere Ereignisse angestoßen hat, muß er diese der Reihe nach durch laufendes Abtasten zyklisch überwachen (-> aktives Warten!).

A.3.2. Einrichtung einer Mehrfachwartestelle

Die Nachteile der Vorgehensweise mit Statusaufrufen (bedingten Systemaufrufen) lassen sich vermeiden, wenn das Betriebssystem Dienste zur Einrichtung einer Mehrfach-Wartestelle bietet. Der Prozeß stößt eine Reihe von Ereignissen an und gibt, sobald er nichts anderes mehr zu erledigen hat, dem Betriebssystem die Kennungen aller oder einiger angestoßener Ereignisse bekannt und wartet dann auf alle angegebenen Ereignisse gleichzeitig. Er wird fortgesetzt, wenn *mindestens eines* der Ereignisse eingetreten ist. Mit der Fortsetzung bekommt er die Kennung des bzw. der

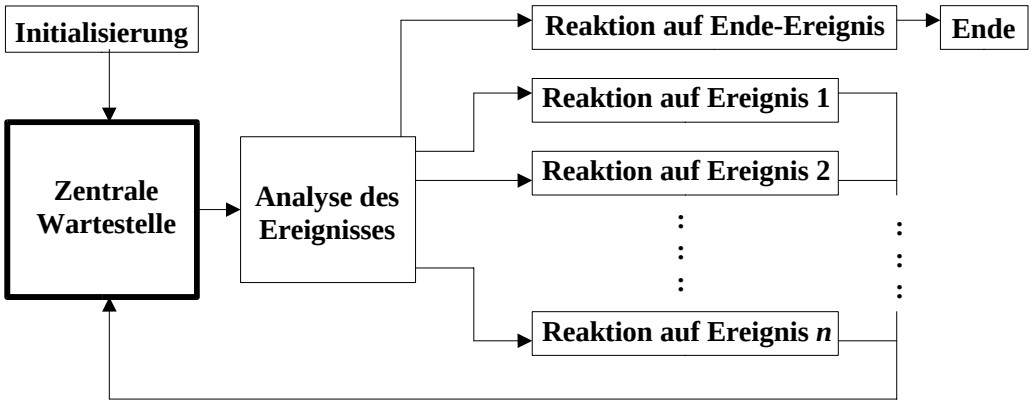
eingetretenen Ereignisse mitgeteilt und kann darauf reagieren. Danach kann er auf die übrigen Ereignisse weiterwarten. Mit einem Systemaufruf, der das Warten auf mehrere Ereignisse gleichzeitig gestattet, wird eine *Mehrfachwartestelle* eingerichtet.

War vor dem Anlaufen der Mehrfachwartestelle mindestens eines der angegebenen Ereignisse bereits eingetreten, so wird der Prozeß natürlich nicht angehalten, sondern sofort unter Angabe der Kennung des eingetretenen Ereignisses fortgesetzt.

A.4. Die zentrale Wartestelle

Bei der Programmierung mit Mehrfachwartestellen ist unbedingt darauf zu achten, daß ein Prozeß nicht mehr als eine, nämlich *die* zentrale Wartestelle besitzt. Andernfalls kann es passieren, daß er auf einer (Mehrfach-)Wartestelle steht, während eben bei einer anderen Wartestelle ein Ereignis eintritt. Er kann dann nicht rechtzeitig darauf reagieren.

Ein Prozeß mit zentraler Wartestelle hat immer folgende (vereinfachte) Grundstruktur (s. Abbildung): Nach einer Initialisierungsphase, in der auch die ersten asynchronen Operationen angestoßen werden, läuft er auf die zentrale Wartestelle und gibt damit den Prozessor ab. Wenn er daraufhin wieder fortgesetzt wird, analysiert er zunächst das eingetretene Ereignis und reagiert darauf. Dabei kann er noch weitere asynchrone Operationen veranlassen. Nach dem Abschluß der Reaktion läuft er wieder auf die zentrale Wartestelle, und der Vorgang wiederholt sich in ähnlicher Weise. Wenn während der letzten Ereignisreaktion bereits weitere Ereignisse eingetreten sind, wird er an der Wartestelle gar nicht blockiert, sondern kann sofort reagieren.



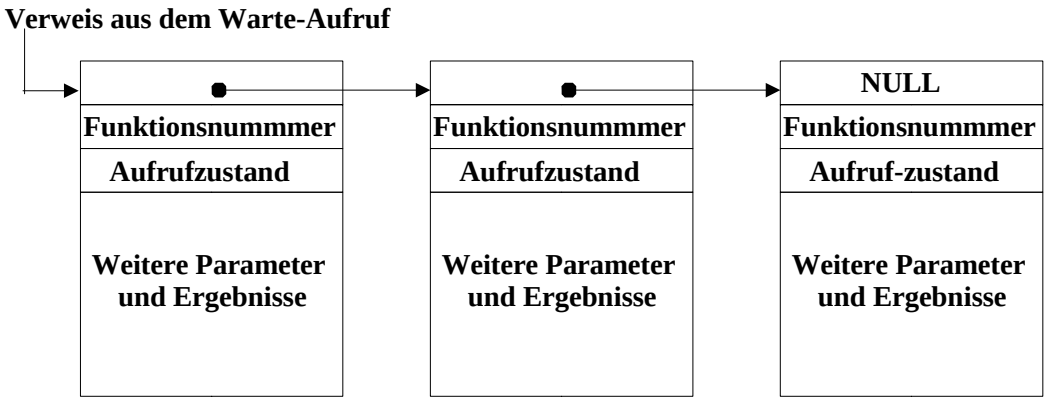
Grundstruktur eines Prozesses mit zentraler Wartestelle

A.5. Realisierungsformen für Mehrfachwartestellen

Mehrfachwartestellen lassen sich auf recht verschiedene Weise realisieren, und das läßt sich auch in der Praxis bei marktgängigen Betriebssystemen feststellen. Grundsätzlich muß dem Betriebssystem mit dem Warteaufruf mitgeteilt werden, auf welche Ereignisse gewartet werden soll. Der Warteaufruf blockiert den Prozeß nur dann, falls bisher keines der gemeldeten Ereignisse eingetreten ist, und setzt ihn andernfalls unmittelbar fort. Nach der Fortsetzung bekommt der Prozeß die Aufrufkennung eines der eingetretenen Ereignisse mitgeteilt, manchmal auch alle.

A.5.1. Verkettete Aufrufkennungen

Hier werden Aufrufkennungen der asynchronen Operationen, auf die gewartet werden soll, miteinander verkettet. Dies läßt sich am einfachsten in der Form realisieren, daß die Parameterblöcke der beteiligten Systemaufrufe in eine verzeigerte Liste eingehängt oder in einer Tabelle aufgeführt werden. Diese Variante hat den Vorteil, daß der wartende Prozeß von allen eingetretenen Ereignissen gleichzeitig benachrichtigt werden kann. Dazu werden in der Liste die Parameterblöcke der abgeschlossenen asynchronen Operationen im Feld "Aufrufzustand" entsprechend gekennzeichnet, deren Resultate ebenfalls in den Parameterblöcken vermerkt werden können, so daß sich eventuell ein Übernahmeaufruf (s. oben) erübrigt. Im Diagramm unten ist im Feld "Funktionsnummer" eine Codierung des jeweiligen Systemaufrufs, also z. B. für *open, close, read, write* usw., angegeben.



Verkettete Parameter-Blöcke von asynchronen Systemaufrufen

A.5.2. Bitvektoren

Beim Start einer asynchronen Operation wird dem Prozeß als Aufrufkennung ein Bitvektor einer bestimmten Länge (z. B. 32 bit) zurückgegeben, in dem ein der Operation zugeordnetes Bit gesetzt ist. Sie wird mit diesem Bit bei Folgeaufrufen (Abbruch, Status-Abfrage, Resynchronisation, Ergebnisrückgabe) identifiziert. Nach der Endebearbeitung wird ein benutztes Bit wieder frei.

Dem Warteaufruf wird vom Prozeß ein Bitvektor (ähnlich wie beim Systemaufruf *select ()*) mit einem Muster übergeben, in dem für jede asynchrone Operation, auf die gewartet werden soll, das zugehörige Bit gesetzt ist. Damit kann durch entsprechende logische Verknüpfungen auch auf eine definierte Teilmenge der ausstehenden Ereignisse gewartet werden. Vom Warteaufruf wird dem Prozeß als Resultat ein Bitmuster übergeben, in dem für jedes eingetretene der erwarteten

Ereignisse ein Bit gesetzt ist. Diese Methode funktioniert auch analog, wenn gezielt mehrere Aufrufe z. B. abgebrochen oder nach ihrem Status abgefragt werden sollen, nicht jedoch für die Ergebnisrückgabe. Der Nachteil dieses ansonsten recht eleganten Verfahrens liegt in der eher niedrigen Grenze bei der Kennzeichnung von ausstehenden asynchronen Operationen (= Länge des Bitvektors, vermindert um 1 für ein Bit zur Fehlerkennzeichnung). Die Beschränkung durch die vorgegebene Bitmusterlänge gilt allerdings nur für einen Prozeß, nicht systemweit.

A.5.3. Interprozeßkommunikation und -synchronisation

Eine Reihe von Realisierungsvarianten von Mehrfachwartestellen stützt sich auf Methoden der Interprozeßkommunikation und -synchronisation. Im Gegensatz zu den beiden vorgenannten Verfahren muß hier bereits bei Anstoß der asynchronen Operation die Kombination der Ereignisse, auf die später gemeinsam gewartet werden soll, festgelegt werden. Ob die Resynchronisation durch Abfragen oder durch Warteaufruf erreicht werden soll, wird durch die Wahl eines geeigneten Kommunikations-Aufrufes festgelegt.

A.5.3.1. Semaphore

Hierbei wird jedem Startaufruf einer asynchronen Operation die Identifikation eines Semaphors übergeben, dessen Zähler mit 0 initialisiert und das vom Betriebssystem bei Aufruf-Abschluß mittels einer V-Operation freigegeben wird. Der Prozeß kann mit einer P-Operation auf die zugehörigen Ereignisse warten.

Dabei muß zusätzlich eine Vorkehrung dafür getroffen werden, wie der abfragende Prozeß erfährt, welche der erwarteten Ereignisse eingetreten sind. Dies kann durch einen zusätzlichen Aufruf, Zugriff auf einen gemeinsamen Speicher oder Semaphore mit Datenübergabe geschehen. Auf jeden Fall macht diese Notwendigkeit den Mechanismus eher kompliziert.

A.5.3.2. Botschaftskanäle

Dem Aufruf zum Start einer asynchronen Operation wird die Identifikation eines Briefkastens übergeben, an dem nur der einrichtende Prozeß wartet. Da das Betriebssystem nach Abschluß der Operation darin nicht nur die Aufrufkennung, sondern sogar auch das gesamte Ergebnis ablegen kann, ist diese Variante gegenüber der mit Semaphoren überlegen.

A.5.3.3. Signale

Unix-spezifische Formen von Mehrfachwartestellen verwenden meistens den *pause*Aufruf, der den aufrufenden Prozeß bis zum Eintreffen eines Signals blockiert. Unix-Signale werden durch den Systemaufruf *kill()*, durch das Ende eines Sohnprozesses oder auch durch den Abschluß der in Abschnitt A.2.3.3 erwähnten Ein/Ausgabe-Operationen ausgelöst.

A.6. Realisierungsvorschläge unter Unix/Linux

Unter Unix/Linux bieten sich die nachfolgend skizzierten Realisierungsvarianten an:

A.6.1. Sohnprozeß nur mit fork ()

Der aufrufende Prozeß erzeugt einen Sohnprozeß mit *fork ()*, der automatisch die Parameter des asynchronen Systemaufrufs erbt. Der Sohnprozeß führt im Auftrag des Vaterprozesses den entsprechenden Systemaufruf synchron (also mit Blockieren) durch und resynchronisiert sich später auf eine der angegebenen Weisen mit dem Vaterprozeß. Nachteil: wenn der Vaterprozeß sehr groß ist, wird durch *fork ()* eine ebenso große Kopie erzeugt, bei mehreren asynchronen Systemaufrufen sogar mehrere Kopien.

A.6.2. Sohnprozeß mit fork () und exec ()

Den letztgenannten Nachteil kann man vermeiden, indem man den Sohnprozeß mit *exec ()* durch einen kleineren Prozeß überlagert, der nur den gewünschten Systemaufruf synchron durchführt. So ein Prozeß wird nicht sehr groß sein. Hier entsteht aber das Problem, wie die Aufrufparameter und (daneben auch) das Aufrufergebnis miteinander ausgetauscht werden, da *exec ()* ja den Datenbereich überschreibt, in dem die Aufrufparameter stehen (Abhilfe: Interprozeßkommunikation).

A.6.3. Threads (Subprozesse)

Ähnlich wie der Ansatz 6.1 funktioniert eine Lösung mit *Threads*: Statt aufwendig einen kompletten Sohnprozeß zu erzeugen, wird hier nur ein *Thread* abgespalten, der im Auftrag des Haupt*threads* den blockierenden Systemaufruf abgibt und sich nach dessen Beendigung wieder mit dem Haupt*thread* resynchronisiert. Allerdings sind die *Thread*-Implementierungen in den verschiedenen Unix-Versionen recht unterschiedlich und teilweise inkompatibel; in Linux gibt es die *Native POSIX Thread Library (NPTL)*, die im Grundsatz *Threads* nur auf Prozesse abbildet, so daß deren Verwendung daher kaum Vorteile gegenüber den Ansätzen 6.1 bzw. 6.2 ergeben würde.

A.6.4. Allgemeiner Stellvertreter-Prozeß

Hierzu wird systemweit ein allgemeiner Stellvertreter-Prozeß eingerichtet, der von anderen Prozessen mit der Ausführung von asynchronen Systemaufrufen beauftragt wird. Die Aufrufparameter und die Ergebnisse sind mittels Interprozeßkommunikation auszutauschen. Für jeden Auftrag muß der Stellvertreter-Prozeß einen eigenen Sohnprozeß erzeugen. - Problem: ein auftraggebender Prozeß kann die zu bearbeitende Datei (bzw. das Gerät) gegenüber dem Stellvertreter-Prozeß nicht mittels der prozeßeigenen Dateinummer identifizieren. Außerdem benötigt er die Rechte des Auftraggeber-Prozesses für die gewünschte Aktion.

A.7 Aufruf-Schnittstelle nach POSIX 1003.1b (Überblick)

Als Aufruf-Schnittstelle wird die von POSIX 1003.1b (Echtzeit-Erweiterungen für Unix) für asynchrone Systemaufrufe vorgeschlagen:

A.7.1. Parameterblock für asynchrone E/A

```
struct aiocb {
   int              aio_fildes;    /* Dateinummer fd */
   off_t            aio_offset;    /* Zugriffsposition */
   void *           aio_buf;       /* Datenpuffer */
   size_t           aio_nbytes;    /* Pufferlaenge/Anzahl Bytes im Puffer */
   int              aio_reqprio;   /* Auftragsprioritaet */
   struct sigevent  aio_sigevent;  /* Signalspezifikation */
   int              aio_lio_opcode;/* Opcode bei Verwendung als Listenelement,
                                      also: LIO_READ, LIO_WRITE, LIO_NOP */
   /* weitere Komponenten sind erlaubt */
};
```

Die Adresse des Parameterblocks *(struct aiocb)* wird bei den meisten Systemaufrufen zur Identifikation eines Auftrags benutzt. Nur zur Information:

```
struct sigevent { /* nur fuer Echtzeit-Signale */
   int           sigev_notify; /* = SIGEV_NONE oder SIGEV_SIGNAL */
   int           sigev_signo;  /* Signalnummer */
   union sigval  sigev_value;  /* Signalwert */
};
```

A.7.2. Asynchrones Lesen: aio_read ()

```
   int aio_read (struct aiocb *aiocbp);
```

Rückgabewerte: 0 bei Erfolg, -1 bei Fehler.

A.7.3. Asynchrones Schreiben: aio_write ()

```
   int aio_write (struct aiocb *aiocbp);
```

Rückgabewerte: 0 bei Erfolg, -1 bei Fehler.

A.7.4. Listengesteuerte E/A: lio_listio ()

```
   int lio_listio (int mode,              /* blockierend oder nicht */
                   struct aiocb *list[],  /* Auftragsliste/-tabelle */
                   int nent,              /* Laenge der Auftragsliste */
                   struct sigevent *sig); /* Signal: alle erledigt */
```

Rückgabewerte: 0 bei Erfolg, -1 bei Fehler.

A.7.5. Operationsabbruch: aio_cancel ()

```
   int aio_cancel (int fildes, struct aiocb *aiocbp);
/* falls aiocbp == NULL, alle Auftraege unter dieser Dateinummer abbrechen! */
```

Rückgabewerte: -1, falls Fehler bei *aio_cancel ()*; sonst
-- *AIO_CANCELED* (alle Aufträge abgebrochen),
-- *AIO_NOTCANCELED* (mindestens ein Auftrag nicht abgebrochen, da bereits erledigt),
-- *AIO_ALLDONE* (alle Aufträge bereits erledigt).

A.7.6. Ermittlung des Rückgabewerts: aio_return ()

```
   size_t aio_return (struct aiocb *aiocbp);
```

Rückgabewerte: wie die Rückgabewerte der (a)synchronen Funktion des Auftrags,
also *aio_read ()* bzw. *read ()*, dementsprechend *aio_write ()* bzw. *write ()*;
-1, falls Fehler bei *aio_return ()*.

A.7.7. Ermittlung des Abschluß- bzw. Fehlerstatus: aio_error ()

```
   int aio_error (struct aiocb *aiocbp);
```

Rückgabewerte:
-- *EINPROGRESS*, falls in `*aiocbp` referierter Auftrag noch in Bearbeitung,
-- 0, falls Auftrag erfolgreich abgeschlossen,
-- *errno* der zugrundeliegenden Operation, falls Auftrag fehlerhaft abgeschlossen;
-- -1, falls Fehler bei *aio_error ()* selbst.

A.7.8. Warten auf Operationsende: aio_suspend ()

```
   int aio_suspend (struct aiocb *list[], int nent, struct timeval *timeout);
```

Es wird gewartet, bis *mindestens einer* der abgegebenen Aufträge fertig ist.
Rückgabewerte: 0 bei Erfolg, -1 bei Fehler.

Anmerkung: von POSIX 1003.1b wird `struct timespec` statt `struct timeval` gefordert.