Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 File-IO-Konstellationen

1.1 Konsistenz und Synchronisation

Alle folgende Konstellationen beziehen sich auf die Laufzeit eines untersuchten Programms. Zugriffe auf Dateien vor Programmstart und nach Programmende werden nicht betrachtet. Eine Unterscheidung in Threads und Prozesse wird bei der Beschreibung der einzelnen Konstellationen nicht vorgenommen, da beide Varianten sich bezüglich der Synchronisation gleich verhalten. Vereinfachend steht der Begriff Prozess daher für Thread oder Prozess.

1.1.1 Prozess 0 schreibt einmalig in eine Datei

Ein Prozess schreibt einmalig in eine Datei. Die Datei wird durch keinen anderen Prozess beschrieben und durch keinen Prozess gelesen.

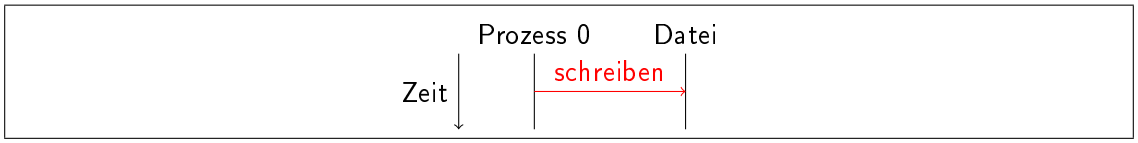

Abbildung 1.1: Prozess 0 schreibt einmalig in eine Datei

Um diese Konstellation zu erkennen, muss für alle Dateizugriffe des Programms die *Datei* und die *Art des Zugriffs* protokolliert werden. Dadurch ist es möglich zu überprüfen, ob nur einmalig schreibend auf eine Datei zugegriffen wird.

- Datei: Pfad und Dateiname
- Art des Zugriffs: öffnen, schreiben, lesen

Bei dieser Konstellation ist keine Synchronisation während der Laufzeit des Programms erforderlich. IO kann gefahrlos optimiert werden. Lediglich vor *Programmende* muss eventuell geprüft werden, ob der *Schreibvorgang abgeschlossen* ist, damit nachfolgenden Programmen die Daten zur Verfügung stehen. Hierfür muss auch ein Schließen der Datei (POSIX close) protokolliert werden.

- Art des Zugriffs: schließen

1.1.2 Prozess 0 schreibt wiederholt in eine Datei

Ein Prozess schreibt wiederholt in eine Datei. Die Datei wird durch keinen anderen Prozess beschrieben und durch keinen Prozess gelesen.

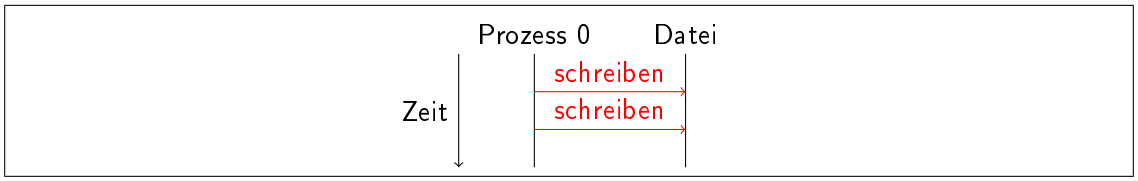

Abbildung 1.2: Prozess 0 schreibt wiederholt in eine Datei

Um diese Konstellation zu erkennen sind, die gleichen Daten wie bei einem einmaligen Schreiben in eine Datei notwendig (siehe Abschnitt 1.1.1, Seite 3). Neben der *Datei* selbst und der *Art des Zugriffs* ist hier allerdings auch noch die *Zeit* von Interesse. Für Optimierungen des Zugriffs ist die *Reihenfolge* der Zugriffe entscheidend. Nur wenn die Reihenfolge protokolliert wird, kann bei einer Optimierung das ursprüngliche Ergebnis sichergestellt werden. Hierfür ist auch die genaue *Position* der geschriebenen *Bytes* in der Datei wichtig. Nur mit dieser Information kann ein Überschreiben in der Datei oder ein sequenzielles Anhängen an ein Dateiende erkannt und bei einer Optimierung beachtet werden. Um Möglichkeiten zur Optimierung zu erkennen, ist zusätzlich die *Dauer* eines einzelnen Schreibvorgangs notwendig. Über diese Information kann geprüft werden, ob ein nachfolgender Schreibvorgang auf den vorhergehenden warten muss. Hierfür muss zu den Schreibvorgängen auch noch das eigentliche Öffnen (POSIX open) protokolliert werden. Dies betrifft allerdings die Art des Zugriffs und entspricht daher den oben bereits erwähnten Informationen.

- Datei: Pfad und Dateiname
- Art des Zugriffs: öffnen, schreiben, lesen, schließen
- Zeit: Start- und Endzeitpunkt des Zugriffs
- Position: Byteposition und-länge des Zugriffs (bei schreiben und lesen)

Bei dieser Konstellation ist eine *Synchronisation* zwischen den einzelnen Schreibvorgängen notwendig, sofern diese sich gegenseitig *überschreiben* oder ein vorhergehender Schreibzugriff das *Dateiende* verschiebt und der aktuelle Vorgang an dieses anknüpft. Im zweiten Fall kann eventuell auf eine Synchronisation über Blockieren nachfolgender Schreibzugriffe verzichtet werden, wenn die Bytelänge und Anzahl der vorhergehenden Schreibvorgänge bekannt ist.

1.1.3 Prozess 0 liest einmalig aus einer Datei

Ein Prozess liest einmalig aus einer Datei. Die Datei wird durch keinen anderen Prozess gelesen und durch keinen Prozess beschrieben.

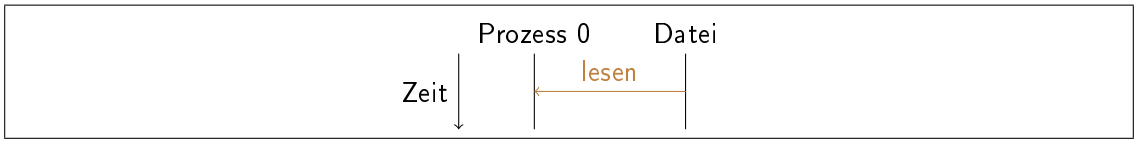

Abbildung 1.3: Prozess 0 liest einmalig aus einer Datei

Um diese Konstellation zu erkennen, sind die gleichen Daten wie bei einem einmaligen Schreiben in eine Datei notwendig (siehe Abschnitt 1.1.1, Seite 3). Das Schließen der Datei ist dabei ebenfalls nur zur Sicherung des Zugriffs durch nachfolgend arbeitende Programme notwendig.

- Datei: Pfad und Dateiname
- Art des Zugriffs: öffnen, schreiben, lesen, schließen

Bei dieser Konstellation ist keine Synchronisation während der Laufzeit des Programms erforderlich. IO kann gefahrlos optimiert werden.

1.1.4 Prozess 0 liest wiederholt aus einer Datei

Ein Prozess liest wiederholt aus einer Datei. Die Datei wird durch keinen anderen Prozess gelesen und durch keinen Prozess beschrieben.

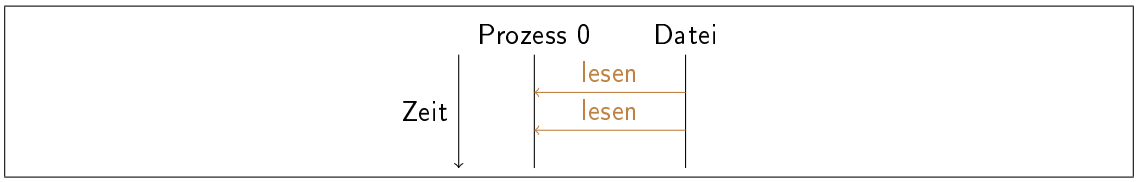

Abbildung 1.4: Prozess 0 liest wiederholt aus einer Datei

Diese Konstellation verhält sich analog zum einmaligen Lesen aus einer Datei (siehe Abschnitt 1.1.3, Seite 4), da das mehrfache Lesen einer sich nicht verändernden Datei weder beim Erkennen der Konstellation noch zur Synchronisation zusätzliche Anforderungen stellt.

- Datei: Pfad und Dateiname
- Art des Zugriffs: öffnen, schreiben, lesen, schließen

1.1.5 Prozess 0 liest und schreibt eine Datei

Ein Prozess liest und schreibt wiederholt eine Datei. Die Datei wird durch keinen anderen Prozess gelesen und durch keinen anderen Prozess beschrieben.

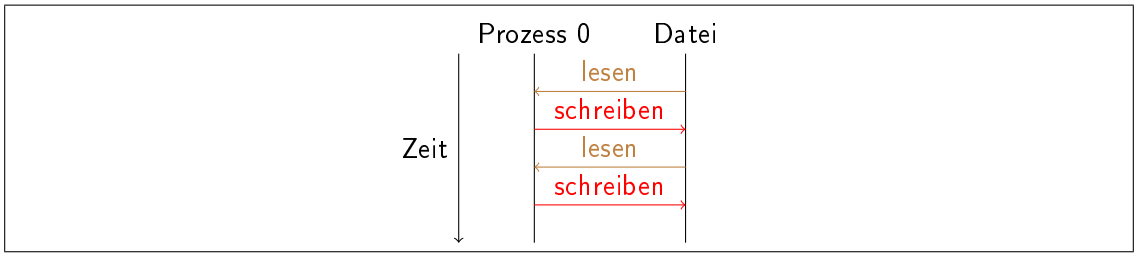

Abbildung 1.5: Prozess 0 liest und schreibt eine Datei

Um zwischen Lesen und Schreiben unterscheiden zu können, sind mindestens die Informationen zum einmaligen Lesen (siehe Abschnitt 1.1.3, Seite 4) und zum einmaligen Schreiben (siehe Abschnitt 1.1.1, Seite 3) notwendig. Um konkurrierende und sich gegenseitig blockierende Schreibvorgänge zu identifizieren sind zudem die gleichen Daten wie beim wiederholten Schreiben nötig (siehe Abschnitt 1.1.2, Seite 4).

- Datei: Pfad und Dateiname
- Art des Zugriffs: öffnen, schreiben, lesen, schließen
- Zeit: Start- und Endzeitpunkt des Zugriffs
- Position: Byteposition und-länge des Zugriffs (bei schreiben und lesen)

Bei dieser Konstellation ist eine *Synchronisation* sowohl zwischen den einzelnen Schreibvorgängen als auch zwischen Schreibvorgängen und darauf folgenden Lesevorgängen notwendig. Hier muss also zusätzlich zu den im Abschnitt über wiederholtes Schreiben (siehe Abschnitt 1.1.2, Seite 4) genannten Prüfungen noch überprüft werden, ob ein lesender Zugriff auf zuvor durch Schreibvorgänge veränderte Bytes erfolgt.

1.1.6 Prozess 0 und Prozess 1 schreiben in eine Datei

Mehrere Prozesse schreiben wiederholt in die gleiche Datei. Die Datei wird durch keinen Prozess gelesen.

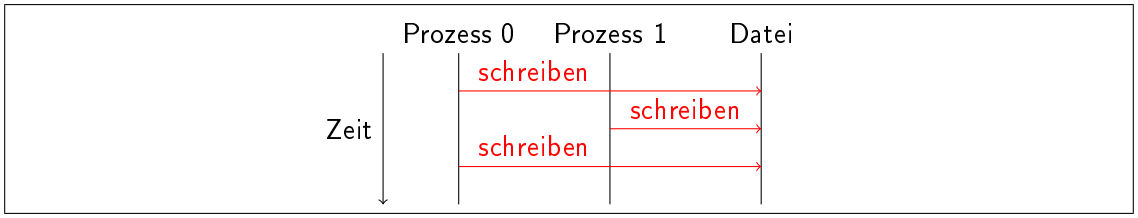

Abbildung 1.6: Prozess 0 und Prozess 1 schreiben in eine Datei

Neben den Daten zum Erkennen mehrerer Schreibvorgänge (siehe Abschnitt 1.1.2, Seite 4) ist noch eine Information über den jeweiligen *Prozess* notwendig.

- Datei: Pfad und Dateiname

- Art des Zugriffs: öffnen, schreiben, lesen, schließen
- Zeit: Start- und Endzeitpunkt des Zugriffs
- Position: Byteposition und-länge des Zugriffs (bei schreiben und lesen)
- Prozess/Thread: Prozess-ID und Thread-Nummer

Wie beim wiederholten Schreiben durch einen Prozess (siehe Abschnitt 1.1.2, Seite 4) ist auch in dieser Konstellation eine *Synchronisation* nötig.

1.1.7 Prozess 0 und Prozess 1 lesen aus einer Datei

Mehrere Prozesse lesen wiederholt aus der gleichen Datei. Die Datei wird durch keinen Prozess beschrieben.

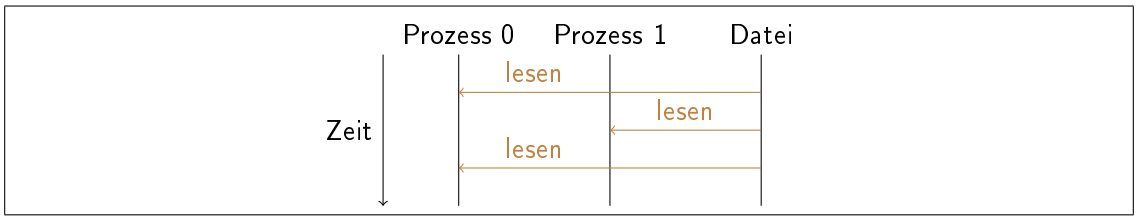

Abbildung 1.7: Prozess 0 und Prozess 1 lesen aus einer Datei

Neben den Daten zum Erkennen mehrerer Lesevorgäng (siehe Abschnitt 1.1.4, Seite 5) ist noch eine Information über den jeweiligen *Prozess* notwendig.

- Datei: Pfad und Dateiname
- Art des Zugriffs: öffnen, schreiben, lesen, schließen
- Prozess/Thread: Prozess-ID und Thread-Nummer

1.1.8 Prozess 0 und Prozess 1 lesen und schreiben eine Datei

Mehrere Prozesse lesen und schreiben wiederholt eine Datei.

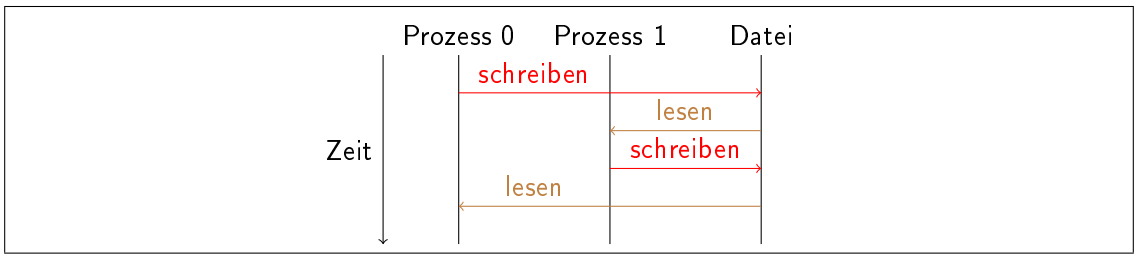

Abbildung 1.8: Prozess 0 und Prozess 1 lesen aus einer Datei

Diese Konstellation stellt eine Kombination aller vorhergehenden Konstellationen dar. Dementsprechend werden alle Daten der einzelnen Konstellationen benötigt.

- Datei: Pfad und Dateiname
- Art des Zugriffs: öffnen, schreiben, lesen, schließen
- Zeit: Start- und Endzeitpunkt des Zugriffs
- Position: Byteposition und-länge des Zugriffs (bei schreiben und lesen)
- Prozess/Thread: Prozess-ID und Thread-Nummer

Hier ist eine *Synchronisation* zwischen den Schreibvorgängen aller beteiligter Prozesse notwendig. Zudem müssen Lesevorgänge nach Schreibvorgängen gegebenenfalls auch synchronisiert erfolgen (wenn ein zuvor geschriebener Dateiinhalt ausgelesen wird).

1.2 Metadaten

Werden Metadaten von einem Programm ausgelesen? Falls dies nicht der Fall ist, so kann möglicherweise auf die Erstellung und Aktualisierung von Metadaten verzichtet werden. Dementsprechend können Methoden und Dateisysteme ohne Metadaten genutzt werden.

1.3 Dateisystem

2 Wrapper

Eine Liste der vom Linux-Kernel angebotenen Systemfunktionen kann der Dokumentation in den „man-Pages“ entnommen werden [4]. Hier finden sich auch die von POSIX definierten Funktionen, welche ebenfalls in der glibc-Bibliothek als C-Funktionen zur Verfügung stehen.

2.1 POSIX über glibc

Die POSIX Implementierung in Linux stellt Systemfunktionen entsprechend der einzelnen POSIX-Funktionen bereit. Diese werden meist nicht direkt, sondern über die entsprechende c-Bibliothek glibc aufgerufen [2]. Für die Fälle, in denen die Systemfunktionen über glibc aufgerufen werden, können Wrapper für die einzelnen Funktionen bereitgestellt werden.

Dabei ist zu beachten, dass einige Funktionen innerhalb der glibc wiederum andere Funktionen aufrufen. So ruft „printf“ zunächst „puts“ auf. In „puts“ wird wiederum „write“ aufgerufen. Da mit den folgend beschriebenen Mitteln nur Aufrufe von außen an die glibc-Bibliothek gewrappt werden können, Aufrufe innerhalb von glibc aber nicht, müssen für das Wrappen aller „write“-Aufrufe auch „puts“ und „printf“ gewrappt werden.

Die folgenden Vorgehensweisen setzen voraus, dass die glibc genutzt wird. Dies ist abhängig von der Programmiersprache und dem jeweiligen Compiler. „Go“ nutzt nach dem Kompilieren beispielsweise direkt die Systemfunktionen ohne die glibc zu verwenden.

2.1.1 dynamische gelinkt

Unter Linux existiert die Umgebungsvariable „LD_PRELOAD“. Diese kann zur Angabe eines Pfades zu einer shared library genutzt werden. Die entsprechende Bibliothek wird dann vor allen anderen Bibliotheken geladen. Werden in dieser Bibliothek Funktionen der glibc-Bibliothek (C-Interface zu Systemfunktionen unter Linux) [4] überschrieben, so werden anstelle der glibc-Funktionen die überschriebenen Funktionen ausgeführt.

Um innerhalb der überschriebenen Funktionen die ursprünglich gerufene Funktion aus der glibc-Bibliothek aufzurufen, kann nicht direkt der Funktionsname genutzt werden, da dies zu einem Namenskonflikt mit der überschriebenen Funktion führt. Anstelle eines Aufrufs über den Namen kann allerdings mit der Funktion dlsym [1] die Adresse der gewünschten Funktion ermittelt werden. Über diese Adresse kann dann die Funktion in glibc ausgeführt werden.

Die Funktion dlsym muss mit der Kponstante „RTLD_NEXT“ (findet erste Routine innerhalb der geladenen Module) aufgerufen werden. Dabei muss über einen Init Hook (Linker -inti) sichergestellt sein, dass einmalig nach dem Laden von glibc der jeweilige Funktionspointer ermittelt wird.

2.1.2 statisch gelinkt

Werden Funktionen nicht zur Laufzeit dynamisch ermittelt, sondern sind statisch fest eingebunden, so kann ein Wrapper nur zum Zeitpunkt des Linkens eingebunden werden. Hierfür kann im GNU Linker die Option „ld –wrap=symbol“ [3] genutzt werden. Über den gcc kann die Option „-Wl“genutzt werden, damit intern der Linker mit „ld –wrap“ aufgerufen wird.

2.2 POSIX über Kernel Entry Point

Im Projekt nicht relevant, da üblicherweise über glibc und nicht direkt über System Calls gearbeitet wird.

In diesem Zusammenhang interessante Schlagworte:

- ptrace
- system call interposition (overhead: context switch on every system call?)
- Beispiele: Plash, Systrace, Subterfugue, Chrome sandbox, Pink trace
- User Mode Linux

2.3 Daten

Für die im Abschnitt zu Konsistenz und Synchronisation (siehe Abschnitt 1.1, Seite 3) aufgeführten Konstellationen müssen durch die Wrapper unterschiedliche Informationen protokolliert werden. Aus diesen Unterschieden ergeben sich folgende Satzarten:

Art des Zugriffs	Prozess-ID	Thread-Nr.	Pfad und Datei	Startzeit	Endzeit	Position	Länge
öffnen	✓	✓	✓	✓	✓	✗	✗
schließen	✓	✓	✓	✓	✓	✗	✗
lesen	✓	✓	✓	✓	✓	✓	✓
schreiben	✓	✓	✓	✓	✓	✓	✓

✓ Datum wird benötigt
✗ Datum wird nicht benötigt

Tabelle 2.1: Übersicht Satzarten

Aus diesen Satzarten ergeben sich mehrere möglich Varianten zur Protokollierung. Um möglichst wenig Speicherplatz zu verbrauchen und die benötigte Zeit beim Schreiben in einen Buffer beziehungsweise in eine Datei minimal zu halten, können verschiedene Satzarten mit unterschiedlicher Länge geschrieben/protokolliert werden. So können die Satzarten für Öffnen und Schließen kürzer als die anderen abgebildet werden. Dieses Vorgehen macht allerdings eine Unterscheidung in verschiedene Satzarten zur Laufzeit des Programms nötig. Nur durch eine Unterscheidung können die einzelnen Daten so protokolliert werden, dass sie später wieder unterschieden werden können. Hierfür ist zusätzliche Logik beim Schreiben in einen Buffer beziehungsweise beim Schreiben aus einem Buffer in eine Logdatei notwendig. Um dies zu vermeiden stehen zwei Alternativen zur Auswahl. Zum einen können alle Satzarten in gleicher Länge gespeichert/geschrieben werden. Hierdurch geht jedoch sowohl der Vorteil beim benötigten Speicherplatz, als auch der Vorteil bei der benötigten Zeit verloren. Zum anderen kann jeder Satz mit einem eindeutigen Kennzeichen begonnen oder abgeschlossen werden. So kann die am Beginn eines Satzes stehende Satzart einfach von anderen Daten unterschieden werden. Hierdurch können die Vorteile ohne aufwändige Logik genutzt werden.

Es werden neben den oben aufgeführten noch weitere Daten benötigt (z.B. Optionen beim Öffnen einer Datei oder MPI-spezifische Parameter).

2.4 Architektur

2.4.1 Thread Local Storage

Über Thread Local Storeage (TLS) sicherstellen, dass beim Schreiben in den Speicher keine Synchronisation notwendig ist und somit auch keine Wartezeiten anfallen. Im TLS für jeden Thread einer Anwendung einen Buffer zum Schreiben reservieren. Sobald ein Buffer voll ist, die enthaltenen Daten in eine eigene Datei schreiben. Dabei die Prozess-ID und die Thread-Nummer im Dateinamen vermerken. Auf diese Weise ist auch beim Schreiben in die Datei

keine Synchronisation notwendig.

Um den TLS beim Start eines Threads zu reservieren und ihn vor dem Ende des jeweiligen Threads abschließend in eine Datei zu übernehmen, muss das Starten und das Beenden eines Threads erkannt werden. Dies gilt ebenso für das Öffnen und Schließen der jeweiligen Datei.

Da die Funktionalität zum Überprüfen des Buffers (ist genügend Platz vorhanden?) und zum Leeren des Buffers in eine Datei, in jedem Wrapper einer POSIX-/MPI-IO-Funktion benötigt wird, muss sie in separate Funktionen ausgelagert werden. Dabei kann die Funktion zum Prüfen des Buffers die Funktion zum Leeren intern nutzen/aufrufen. Somit muss in den Wrappern nur eine Funktion genutzt werden und das Leeren steht dennoch als separate Funktion für das Beenden eines Threads zur Verfügung.

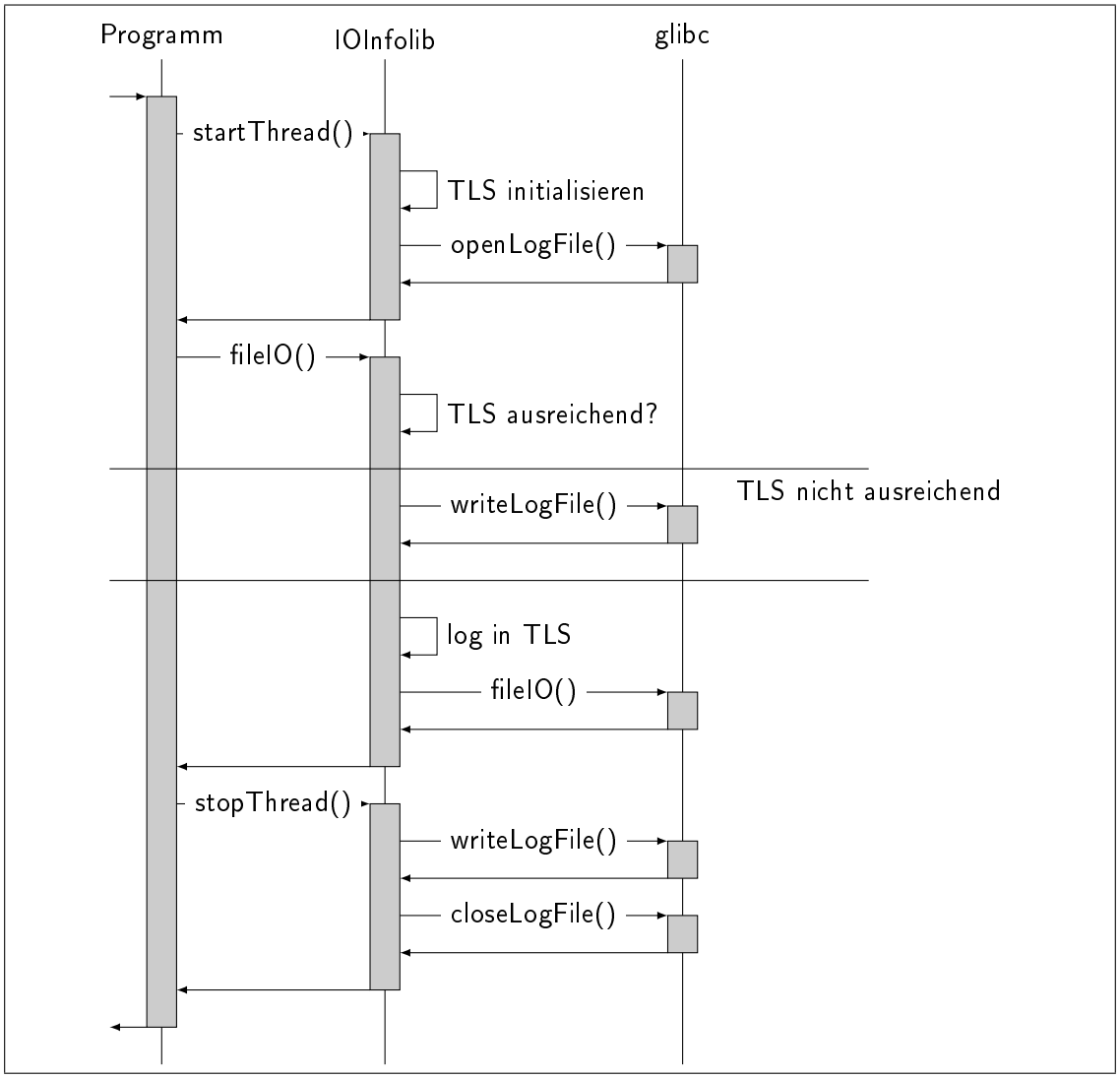

Abbildung 2.1: Sequenzdiagramm Wrapper mit TLS

Je nach Art der Parallelisierung unter Linux kann das Starten und das Stoppen eines nebenläufigen Vorgangs unterschiedlich abgefangen werden. Bei einem Vorgehen über fork() und exit() werden separate Prozesse (heavy-weight process) gestartet. Geschieht dies über die entsprechenden Funktionen in der glibc, so kann es über Wrapper abgefangen werden. Zusätzlich müssen noch alle nicht mittels fork() sondern über einen Befehl zum Ausführen einer Datei gestarteten Prozesse abgefangen werden. Dies betrifft die Funktionen execl(), execlp(), execv() und execvp(). Wird anstelle von fork() die Funktion clone() genutzt, so wird kein Prozess, sondern ein Thread (light-weight process) gestartet. Daher muss dies Funktion ebenfalls abgefangen werden. In diesem Fall gibt es keinen exit() für die einzelnen Threads. Stattdessen kann ein waitpid() mit der Prozess-ID des Threads genutzt werden, um das Ende des Threads zu erkennen. In diesem Fall ist der Speicher des Threads allerdings bereits freigegeben und nicht mehr sicher nutzbar. Das finale schreiben des Buffers in eine Datei kann also auf diesem Weg nicht ermöglicht werden. Das gleiche gilt für exit_group(). Über diese Funktion werden mehrere laufende Prozesse/Threads in einer Prozess-Gruppe gemeinsam beendet. Auch hierbei ist nicht sichergestellt, ob der Speicher eines Threads nicht bereits freigegeben wurde.

Wurde alternativ über die POSIX-Threads in der glibc parallelisiert, so kann über das Makro pthread_cleanup_push() eine weitere Funktion zum finalen Cleanup übergeben werden. Da dieses Makro allerdings immer mit einem weiteren Makro (pthread_cleanup_pop()) innerhalb der gleichen umschließenden Klammern kombiniert werden muss, lässt sich dieser Ansatz bei einer event-orientierten Vorgehensweise nicht nutzen. Zudem funktioniert dieses Vorgehen nur bei Threads, die mit den entsprechenden Funktionen der glibc erstellt wurden. Somit lassen sich beispielsweise über openMP parallelisierte Programme so nicht instrumentieren.

Bei dieser Architektur muss also zwischen unterschiedlichen Parallelisierungsarten unterschieden werden. Zudem gibt es Vorgehensweisen, die nicht über die glibc-Funktionen gehen. So nutzt openMP offensichtlich direkt die Funktionen des Kernels und reduziert auf diese Weise den Overhead der Parallelisierung auf ein Minimum. Leider gibt es bei openMP somit keine Funktionen in der glibc oder einer anderen Bibliothek, die durch einen Wrapper abgefangen werden können.

thrd_create() thrd_exit() pthread_exit()

2.4.2 zentraler Buffer

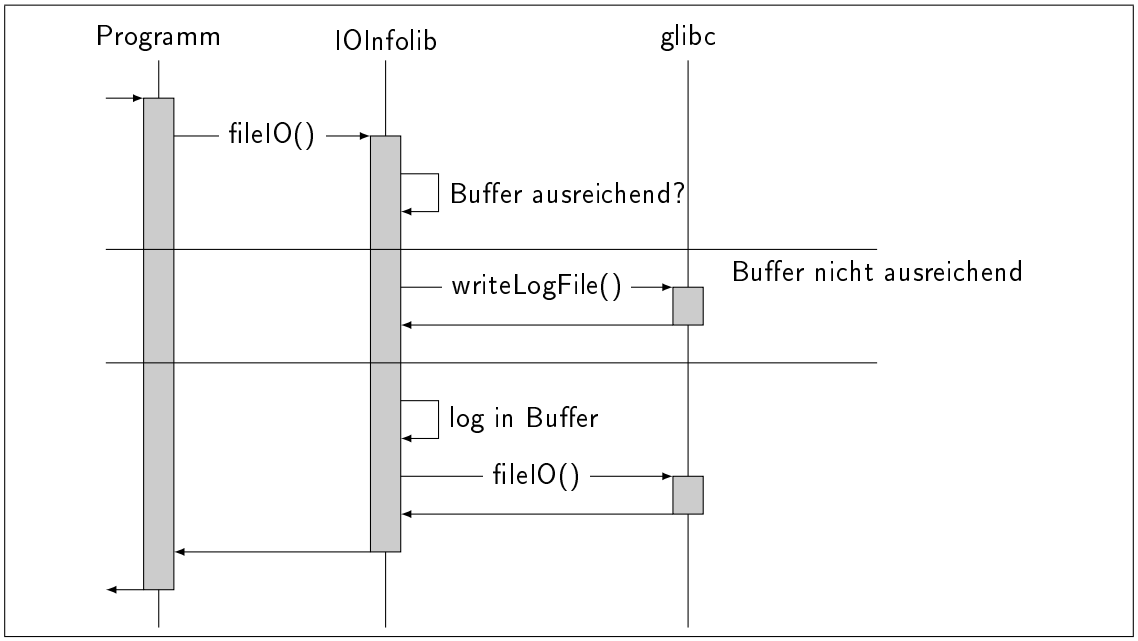

Abbildung 2.2: Sequenzdiagramm Wrapper mit zentralem Buffer

```
1  #define _GNU_SOURCE
2  #include <dlfcn.h>
3  #include <stdio.h>
4  #include <string.h>
5  #include <unistd.h>
6  #include <sys/types.h>
7  #include <stdarg.h>
8  #include <pthread.h>
9
10 static void init() __attribute__((constructor));
11 static void cleanup() __attribute__((destructor));
12
13 /* Function pointers for glibc functions */
14 static ssize_t (*real_write)(int fd, const void *buf, size_t count) =
       NULL;
15 static int (*real_puts)(const char* str) = NULL;
16 static int (*real_printf)(const char *__restrict format, ...) = NULL;
17 static int (*real_vprintf)(const char *__restrict format, _G_va_list
       arg) = NULL;
18
19 /* Buffer */
20 #define BUFFER_SIZE 400
21 static char data_buffer[BUFFER_SIZE];
22 static char* endpos;
23 static char* pos;
```

```
24
25  /* Mutex */
26  static pthread_mutex_t lock;
27
28  void printData() {
29          real_printf(data_buffer);
30          pos = data_buffer;
31          *pos = '\0';
32  }
33
34  void writeData(char *data) {
35          int tmp_pos;
36          char print[100];
37          sprintf(print, "pid:%lu;pts:%lu;", getpid(), pthread_self());
38          strcat(print, data);
39          strcat(print, "\n");
40
41          /* write (synchronized) */
42          pthread_mutex_lock(&lock);
43          if (pos + strlen(print) > endpos) {
44                  printData();
45          }
46          strcpy(pos, print);
47          pos += strlen(print);
48
49          pthread_mutex_unlock(&lock);
50  }
51
52  void cleanup() {
53          printData();
54
55          pthread_mutex_destroy(&lock);
56  }
57
58  static void init() {
59          real_write = dlsym(RTLD_NEXT, "write");
60          real_puts = dlsym(RTLD_NEXT, "puts");
61          real_printf = dlsym(RTLD_NEXT, "printf");
62          real_vprintf = dlsym(RTLD_NEXT, "vprintf");
63
64          endpos = data_buffer + BUFFER_SIZE - 1;
65          pos = data_buffer;
66
67          pthread_mutex_init(&lock, NULL);
68  }
69
70  ssize_t write(int fd, const void *buf, size_t count) {
71          char print[40];
72          sprintf(print, "write:chars#:%lu", count);
```

```
73          writeData(print);
74
75          return real_write(fd, buf, count);
76  }
77
78  int puts(const char* str) {
79          char print[40];
80          sprintf(print, "puts:chars#:%lu", strlen(str));
81          writeData(print);
82
83          return real_puts(str);
84  }
85
86  int printf(const char *__restrict format, ...) {
87          va_list args;
88          int retval;
89
90          char print[40];
91          sprintf(print, "printf:chars#:%lu", strlen(format));
92          writeData(print);
93
94          va_start(args, format);
95          /* use vprintf instead of printf because of the variable
               parameter-list */
96          retval = real_vprintf(format, args);
97          va_end(args);
98          return retval;
99  }
```

Literaturverzeichnis

[1] *dlsym3*. http://man7.org/linux/man-pages/man3/dlsym.3.html, o. J.. – [Online; Zugriff am 07.11.2018]

[2] *intro2*. http://man7.org/linux/man-pages/man2/intro.2.html, o. J.. – [Online; Zugriff am 07.11.2018]

[3] *ld1*. http://man7.org/linux/man-pages/man1/ld.1.html, o. J.. – [Online; Zugriff am 07.11.2018]

[4] *syscalls2*. http://man7.org/linux/man-pages/man2/syscalls.2.html, o. J.. – [Online; Zugriff am 07.11.2018]