Inhaltsverzeichnis

1. POSIX–IPC: Named Pipes (4)

1.1. mkfifo - Named Pipes bei nicht verwandten Prozessen

Nicht miteinander **verwandte** Prozesse können durch gewöhnliche Pipes nicht kommunizieren. Bei diesen werden zB. Named Pipes (FIFOs) verwendet.

Ein FIFO ist letztlich eine Pipe, die aber über einen Namen im Dateisystem zu erreichen ist.

Ein FIFO wird mittels des Systemaufrufs (man 3 mkfifo) erzeugt.

```
NAME
       mkfifo - make a FIFO special file (a named pipe)

SYNOPSIS
       #include <sys/types.h>
       #include <sys/stat.h>

       int mkfifo(const char *pathname, mode_t mode);
```

Die FIFOs werden im Dateisystem abgelegt und müssen dort wieder entfernt werden.

Wurde ein FIFO erzeugt, kann dieser mit den Standard-I/O-Funktionen **open(),close(), read(), write() und unlink()** benutzt werden.

1.1.1. Beispiel: demo_fifo_echo.c – 1 Reader und N Schreiber

- (Reader 1) Terminal 1: ./demo_fifo_echo.exe
- (Schreiber 1) Terminal 2: cat > /tmp/myFIFO-echo
- (Schreiber 2) Terminal 3: cat > /tmp/myFIFO-echo

```
// demo_mkfifo_echo.c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <fcntl.h>
#include <signal.h>

#define FIFO_NAME "/tmp/myFIFO-echo"

int main(){
      int fd, ret;
      char c;

      ret=mkfifo(FIFO_NAME, S_IRUSR | S_IWUSR);
      if (ret==-1){ perror(FIFO_NAME); exit(0); }

      fd=open(FIFO_NAME,O_RDONLY);
      if (fd<0 ) { perror(FIFO_NAME); exit(0); }

      while (read(fd,&c,1)==1)
            fputc(c, stdout);

      close(fd);

      unlink(FIFO_NAME);
      return 0;
 }
 // gcc demo_mkfifo_echo.c -o demo_mkfifo_echo.exe
 // ./demo_mkfifo_echo.exe
 //
 // in 2 anderen Terminals
 // cat > /tmp/myFIFO-echo
```

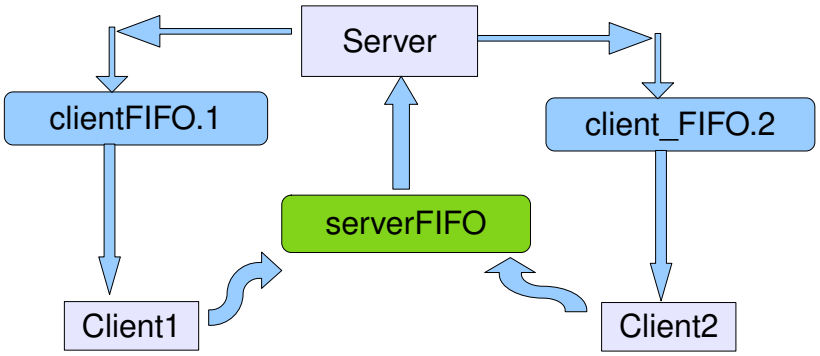
1.1.2. Beachte im Umgang mit FIFOs

- Bei der Verwendung von FIFOs ist es üblich, dass mehrere Schreiber in einen FIFO schreiben. Damit die Daten der einzelnen Schreiboperationen nicht vermischt werden, müssen atomare Schreiboperationen verwendet werden. Also müssen bei jeder Schreiboperation weniger als PIPE_BUF Bytes geschrieben werden.
- Wenn man einen FIFO zum Schreiben mit close oder fclose schließt, bedeutet dies für den FIFO zum Lesen ein EOF.
- Mehrere Schreiber und 1 Reader: (vgl. Druckerspooler)
  Haben alle Schreiber eines FIFO ihre Verbindung abgebaut und wurden alle Daten aus der FIFO gelesen, liefert der nächste read(2)-Aufruf 0 Bytes gelesen, ohne zu blockieren.

1.1.3. Beispiel: fifo-client-server

Mit FIFOs lassen sich zB. mittels Shellprogrammierung sehr leicht Client/Server-Anwendungen implementieren. Als Beispiel implementieren wir einen Server, der ein Telefonbuch verwaltet. Die zugehörigen Clienten, von denen es viele geben kann, können vom Server Telefondaten abrufen und ihn auffordern, neue Telefondaten in die Datenbank einzutragen. Der Kommunikationsfluss:

Wenn der Server den verschiedenen Clients Antworten schicken soll, dann muss für jeden Client ein eigener FIFO angelegt werden.



Jeder Client nimmt über eine bekannte FIFO Kontakt mit dem Server auf, die Antwort liefert der Server über einen nur ihm und dem Klient bekannt FIFO.

Datei: fifo-client.sh

```
#!/bin/sh
# fifo-client.sh
# $$ ... liefert die PID
#
CLIENT_FIFO=/tmp/clientFIFO.$$
SERVER_FIFO=/tmp/serverFIFO

#
mkfifo $CLIENT_FIFO

# SIGTERM SIGINT abfangen und CLIENT_FIFO löschen und ende
trap "trap '' 0 1 2 3 15;
      rm -f $CLIENT_FIFO; exit 0;" 0 1 2 3 15

# infos
echo " Kommandos:"
echo "            add   Name  Number"
echo "            get   Name"
echo "            get   Number"

# Anfrage lesen und an Server schicken
while read question
do
      echo $question $CLIENT_FIFO > $SERVER_FIFO
      read answer < $CLIENT_FIFO
      echo $answer > /dev/tty
done
```

Der Client

1. legt eine nur ihm bekannte CLIENT_FIFO an und
2. liest von der Standardeingabe die Anfragen.
3. Die Anfrage (question) und der Name der CLIENT_FIFO -welche der Server für die Antwort verwendet- werden über die SERVER_FIFO an der Server geschickt.
4. Die Antworten werden aus der CLIENT_FIFO gelesen und aufs Terminal ausgegeben.
5. Falls der Client terminiert, wird die CLIENT_FIFO durch die trap aus dem Dateisystem entfernt.

Datei: fifo-server.sh

```
#!/bin/sh
# fifo-server.sh
#    Telefonserver
#                $1    $2          $3          $4
#    Kommandos:
```

```
#                add   Name  Number      FIFO
#               get Name    FIFO
#               get Number  FIFO

SERVER_FIFO=/tmp/serverFIFO
DATA=fifo-server.db         # Name Number

mkfifo $SERVER_FIFO
#
trap 'trap "" 0 1 2 3 15;
      rm -f $SERVER_FIFO; exit ' 0 1 2 3 15

while true
do
      (  while read question < $SERVER_FIFO
            do
                  set $question
                  case  $1
                        in a*) echo "$2 $3" >>  $DATA
                                    sort -u -o $DATA $DATA
                                    echo added > $4
                        ;; g*) found="`grep "$2" $DATA`"
                                    echo "$found" > $3
                        ;; *)  echo "What?" > /dev/console
                  esac
            done
      )
done
```

1.2. popen - bidirektionale Kommunikation mit Komfort

Die Subroutine popen() führt das im Parameter (String) angegebene Kommando mit Hilfe von system() aus. Dabei wird zusätzlich eine Pipe angelegt, die von popen zurückgegeben wird.Auf diese pipe kann dann geschrieben (mode = "w") bzw. gelesen (mode = "r") werden. Die Subroutine pclose schließt den als Argument anzugebenden Stream wieder und liefert den Status der Shell als Resultat. Falls sich der Stream nicht schließen läßt, ist das Resultat gleich EOF.

Zum Lesen bzw. Schreiben des Streams dienen die alltäglichen Standard-I/O-Funktionen wie fgets() oder fputs().

popen, pclose - process I/O

1.2.1. Beispiel: demo_popen.c

```
// demo_popen.c

#include <stdio.h>
#include <stdlib.h>

#define BUF_SIZE 1024

int main()
{
    char buf[BUF_SIZE];
```

```
    FILE *pstream;


    pstream = popen("sort /etc/passwd", "r");
    while (fgets(buf, BUF_SIZE, pstream))
    {
        fputs(buf, stdout);
    }
    fclose(pstream);


    pstream = popen("pstree", "r");
    while (fgets(buf, BUF_SIZE, pstream))
    {
        fputs(buf, stdout);
    }
    fclose(pstream);


    char *cmd = "curl -s -L http://www.zamg.ac.at/ogd | grep Salzburg | awk -
F\";\" '{print $2 \";\" $4 \";\" $5 \";hat \" $6 \" Grad celsius\"}'";

    pstream = popen(cmd, "r");
    while (fgets(buf, BUF_SIZE, pstream))
    {
        fputs(buf, stdout);
    }
    fclose(pstream);

    return 0;
}

// gcc demo_popen.c -o demo_popen.exe; ./demo_popen.exe
```

1.2.2. Aufgabe: popen_starter.c

Schreiben Sie ein Programm, das aus der Kommandozeile als 1. Argument einen Dateinamen und als 2. Argument ein Shellkommando erhält. Führen Sie dieses Kommando mit popen() aus und schreiben Sie das Ergebnis in die Datei.

Beispiel:
./popen_starter.exe erg.txt "find /etc -type f | grep -i localhost"

Hinweis:
Da der Befehl spaces enthalten kann, muss er mit " " geklammert werden.

1.3. Synchronisation mit Lockfiles

Wir betrachten in diesem Abschnitt eine einfache und altbewährte Methode zur Erreichung des **gegenseitigen Ausschlusses bei kritischen Abschnitten**.

Mit Hilfe der Eigenschaften des Systemaufrufs open() implementieren sogenannte Lock-Files eine Art von **Semaphoren**.

☑ P-Operation: in den kritischen Abschnitt eintreten
Als Simulation der P-Operation wird wiederholt versucht, das Lock-File mit open() zu kreieren.

☑ Die V-Operation: den kritischen Abschnitt verlassen
wird auf das Löschen des Lock-Files mit unlink zurückgeführt.

☑ Die beiden Operationen P und V heißen in unserem Zusammenhang lock und unlock.

Die wesentliche Idee bei dieser Vorgehensweise ist die gemeinsame Verwendung der Zugriffs-Flags O_CREAT und O_EXCL bei open (man open). Wenn das File schon existiert, kann ein Prozess nicht erfolgreich damit sein. Unter der Voraussetzung, dass der Systemaufruf open atomar ist, also nicht unterbrochen werden kann, ist damit die Semaphor-Eigenschaft sichergestellt.

Wir wollen in der Folge zwei Funktionen zum Betreten und Verlassen des kritischen Abschnittes implementieren:

☑ **int lock(char* name);**

☑ **void unlock(char* name);**

Die Funktionen haben beide das Argument name, einen String-Pointer, der den Namen des Lock-Files im Directory /tmp angibt. Das Resultat von lock ist im Erfolgsfalls die Anzahl der Versuche, die zur Inbesitznahme des Lock-Files nötig waren, minimal 1. Im Fehlerfall ist das Resultat negativ, absolut gleich der Anzahl der Fehlversuche.

1.3.1. Beispiel: demo_lock.c

```
// demo_lock.c
// implement critical section using lock-files

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>

#define LOCKDIR "/tmp/"
#define MAX_LOCKNAME 5+64+1
#define MAX_ATTEMPTS 10
#define SLPTIME (unsigned int)1


/* Lock-File in Besitz bringen */
int lock(char* name){
      int fd;
      int nAttempts;
      char full_lockname[MAX_LOCKNAME];

      // absolute path of the lock-file:
      // /tmp/..... erstellen
      strcpy(full_lockname, LOCKDIR);
      strncat(full_lockname, name, 64);

      // try to create lock-file
      nAttempts = 1;
      fd = open(full_lockname, O_WRONLY|O_CREAT|O_EXCL,0666);
      while (fd==-1 && errno==EEXIST){
            // lockfile exists-> so try again

            if (nAttempts >= MAX_ATTEMPTS)
                  return -nAttempts;
```

```
            sleep(SLPTIME);
            nAttempts++;
            fd = open(full_lockname, O_WRONLY|O_CREAT|O_EXCL,0666);
      }

      return nAttempts;
}

/* Lock-File freigeben */
void unlock(char* name) {
      char full_lockname[MAX_LOCKNAME];

      // absolute path of the lock-file:
      // /tmp/..... erstellen
      strcpy(full_lockname, LOCKDIR);
      strncat(full_lockname, name, 64);

      if (unlink(full_lockname) == -1) {
            fprintf(stderr, "Error: unlock().\n");
            exit(1);
      }
}


int main(int argc, char* argv[]){
      char lockname[128];
      int  pid, i, l, nLoops;
      int ret_lock;

      if (argc != 4) {
            fprintf(stderr, "usage: %s processname lock-filename nLoops\n",
argv[0]);
            exit(1);
      }

      strcpy(lockname, argv[2]);
      nLoops = atoi(argv[3]);


      for (i=0; i<nLoops; i++) {
            // TRY LOCK
            printf("\n%s: TRY to ENTER critical section", argv[1]);
            printf(" (nLoop=%2i Lock-File=%s)", i, lockname);
            fflush(stdout);

            ret_lock = lock(lockname);
            if (ret_lock < 0) {
                  fprintf(stderr,"%2i attempts failed to create Lock-File %s\n",
                        ret_lock, lockname);
                  exit(1);
            }

            // critical section
            printf("\n%s: IN critical section", argv[1]);
            printf(" (nLoop=%2i locking-attempts=%2i)",  i, ret_lock);
            fflush(stdout);

            //UNLOCK
            unlock(lockname);
            printf("\n%s: LEFT critical section", argv[1]);
            printf(" (nLoop=%2i Lock-File=%s)", i, lockname);
```

```
            fflush(stdout);

            sleep(2);
      }/*for*/

      printf("\n\n");
      return 0;
}
```

Wir compilieren demo_lock.c und starten zwei Versionen zum konkurrenten Ablauf. Dies sieht dann so aus:

gcc demo_lock.c -o demo_lock.exe
./demo_lock.exe "AAA" LOCK 10 & ./demo_lock.exe "……ZZZ" LOCK 10 &

Typische Beispiele der Anwendung von Lockfiles innerhalb UNIX sind zu finden bei relativen langsamen Vorgängen, wie der Synchronisierung von Druckjobs.