Code-Dateien finden Sie in `swf-sp-ws2023-ue03.tgz`, Download mit:

```
wget swf.hgesser.de/bs-sp/ws2023/prakt/swf-sp-ws2023-ue03.tgz
```

6. Prozessliste: ein ps-Klon

a) In dieser Aufgabe geht es darum, eine Prozessliste auszugeben. Dabei greifen Sie auf die Informationen zurück, die Sie aus dem `/proc`-Dateisystem (aus den Einträgen `/proc/PID/stat`) auslesen können. Ihre Version von `ps` soll tabellarisch die folgenden Informationen für alle Prozesse ausgeben:

- Prozess-ID
- Status (ein Buchstabe)
- Parent-Process-ID
- Prozessgruppen- und Session-IDs
- Terminal (nur numerisch, es ist keine Umwandlung in `tty1`, `tty2`, `pts/1` etc. nötig)
- Kommando in Langfassung – das erhalten Sie nicht aus `/proc/PID/stat`, sondern über die Datei `/proc/PID/cmdline`.

Das Hauptprogramm besteht aus einer Schleife, die von 1 bis 32768 läuft und für jede dieser potenziellen Prozess-IDs (diese Grenze finden Sie über `cat /proc/sys/kernel/pid_max` heraus) prüft, ob es ein Verzeichnis `/proc/pid/` gibt:

1. Öffnen Sie das Verzeichnis `/proc/pid` mit `open()`.
2. Im Erfolgsfall schließen Sie den File Descriptor direkt wieder und wissen, dass es dieses Verzeichnis gibt.
3. Dann öffnen Sie die Datei `/proc/pid/stat` und kopieren den Inhalt dieser Datei in einen Puffer. (500 Byte Puffergröße reichen aus.)
4. Der Pufferinhalt besteht aus mehreren Feldern, die durch Leerzeichen getrennt sind. Sie können sich eine kleine Routine `getvalue()` schreiben, die das `i`-te Feld zurück gibt (also `i` Leerzeichen überspringt).
5. Mit den folgenden `#define`-Anweisungen

   ```
   #define PROC_PID 0
   #define PROC_NAME 1
   #define PROC_STATE 2
   #define PROC_PPID 3
   #define PROC_PGRP 4
   #define PROC_SESS 5
   #define PROC_TTYNO 6
   #define PROC_UTIME 13
   #define PROC_STIME 14
   #define PROC_NICE 18
   #define PROC_THREADS 19
   ```

   finden Sie schnell die richtigen Felder und übertragen die Inhalte in dafür vorbereitete String-Variablen, die Sie schließlich mit

   ```
   printf ("%7s  %-2s%7s  %7s  %7s  %4x %s\n",
           prid, state, ppid, pgrp, sess, ttyno, longcmd);
   ```

   ausgeben. Fügen Sie vorab noch eine Überschriftszeile ein.

Im Code-Archiv finden Sie eine Quellcode-Datei (`myps.c`), die schon die obigen Definitionen enthält und alle benötigten Header-Dateien einbindet.

Um aus einer Integer-Zahl *pid* den String "/proc/*pid*/" bzw. "/proc/*pid*/stat" zu erzeugen, benötigen Sie die Funktion `sprintf()`, die ähnlich arbeitet wie `printf` (siehe `man sprintf`).

Nach ersten Experimenten werden Sie feststellen, dass `/proc/pid/cmdline` bei einigen Prozessen leer ist. Diesen Fall sollen Sie erkennen und dann stattdessen das `cmd`-Feld aus `/proc/pid/stat` ausgeben; dann geben Sie den Kommandonamen in eckigen Klammern (z. B. `[kthreadd]`) aus, wie es auch das Kommando `ps` tut.

b) Starten Sie Firefox und öffnen Sie ein paar Webseiten; das Programm besteht aus mehreren Threads. Was fällt Ihnen (anhand von Firefox) auf, wenn Sie die Prozessliste mit Ihrer `ps`-Implementation anzeigen?

7. Adress-Datenbank

a) Betrachten Sie die Datei `record.c` aus dem Aufgabenarchiv. Testen Sie das Programm und verstehen Sie, wie es arbeitet. Hilfestellung gibt die Manpage zu `lseek` (`man lseek`).

b) Erweitern Sie das Programm um eine Aktion `new` (aufzurufen mit `new 0`, weil die Eingabe immer zwei Argumente erwartet). Das Programm soll dann (wie beim Ändern) die Werte für einen neuen Record erfragen und diesen am Ende der Datei ergänzen. Geben Sie danach die ID des neuen Eintrags aus. Auch für das Anhängen eines neuen Eintrags benötigen Sie `lseek`.

c) Erweitern Sie das Programm um eine Suchfunktion `search` (aufzurufen mit `search 0`). Das Programm soll dann einen Suchbegriff einlesen und die IDs aller Einträge anzeigen, bei denen der Suchbegriff mit dem Namen oder Vornamen identisch ist (keine Suche nach Teilen).

8. Shell mit Job-Kontrolle

In der Datei `spsh.c` finden Sie eine Musterlösung zur Mini-Shell-Aufgabe 2 (Übungsblatt 2) – mit einer Erweiterung: Wenn das letzte Argument & ist, wird der neue Prozess im Hintergrund gestartet. Zu Beginn der Hauptschleife im Programm prüft die Mini-Shell, ob es beendete Hintergrundprozesse gibt. Falls ja, zeigt sie deren PIDs an. Außerdem wird ein internes Kommando `exit` interpretiert. Lesen und verstehen Sie das Programm.

Hinweis: Die Musterlösung verwendet die Bibliotheksfunktion `strtok()`, die in einer Schleife die einzelnen, durch Leerzeichen getrennten Argumente aus einem String holt.

9. Shell mit internem `ls`-Kommando

a) In dieser Aufgabe geht es darum, die Mini-Shell (`spsh.c` aus Aufgabe 8) um ein internes `ls`-Kommando zu ergänzen, das nur dann zum Einsatz kommt, wenn Sie in der Shell `ls` und einen einzelnen Dateinamen als Argument eingeben. (Diese Einschränkung dient dazu, dass Sie nicht Fälle mit mehreren Argumenten behandeln müssen.) Schreiben Sie eine Funktion `ls()`, die Sie mit dem Dateinamen als Argument aufrufen.

Ein Programmrumpf steht im Archiv als `spsh-mit-ls.c` bereit, darin sind schon alle benötigten `#include`-Direktiven und Vorschläge für die zu verwendenden Variablen (sowie die Mini-Shell ohne Aufruf der `ls()`-Funktion) enthalten.

Die Ausgabe soll so aussehen wie ein Aufruf von `ls -ild datei`; bei der Anzeige von Datum und Uhrzeit können Sie das Format `YYYY/MM/DD hh:mm:ss` verwenden.

Wenn Sie die Shell aus Übung erfolgreich implementiert haben, können Sie auch Ihre eigene Lösung als Basis verwenden.

Zur Implementierung benötigen Sie die Funktion `lstat()` (siehe `man 2 lstat`), die Informationen über eine Datei in ein Struct vom Typ `struct stat` schreibt. Um z. B. zu testen, ob `/tmp` ein Verzeichnis ist, könnten Sie folgenden Code verwenden:

```
struct stat s;
lstat ("/tmp", &s);
if (S_ISDIR(s.st_mode)) printf ("/tmp ist ein Verzeichnis");
```

Lesen Sie auch die Hinweise zum Unterschied zwischen `lstat()` und `stat()`.

Analog zu `S_ISDIR()` gibt es weitere Makros, mit denen Sie auf verschiedene Eigenschaften prüfen können. Ihre `ls()`-Funktion soll prüfen, ob es sich um eine normale Datei (`-`), ein Verzeichnis (`d`), einen symbolischen Link (`l`), eine Zeichen- (`c`) oder Blockgerätedatei (`b`) oder einen FIFO (`f`) handelt. Beschreibungen der dafür verfügbaren Makros finden Sie in der Manpage.

Für erste Tests können Sie die Benutzer- und Gruppen-IDs als Zahlen ausgeben. `ls` zeigt aber stattdessen die Benutzer- und Gruppennamen an, die in den Dateien `/etc/passwd` und `/etc/group` stehen. Sie müssen nicht selbst in diese Dateien schauen: Es stehen die Funktionen `getpwuid()` und `getgrgid()` zur Verfügung, deren Manpages verraten, wie sie zu nutzen sind.

Für die Ausgabe von Datum und Zeit müssen Sie zunächst den in der `stat`-Struktur gespeicherten Datumswert konvertieren. Dazu nutzen Sie die Funktion `localtime()`, die als Rückgabewert einen `struct tm *` hat. Darin gespeichert sind u. a. die Anzahl Jahre seit 1900, der Monat (0..11), Tag, Stunde, Minute und Sekunde. Bei der Ausgabe müssen Sie also z. B. zur Jahreszahl 1900 addieren, um den richtigen Wert zu erhalten.

Bei Gerätedateien zeigt `ls` statt der Dateigröße zwei durch Komma getrennte Zahlen (die sog. Major- und Minor-Geräte-IDs) an. Ihre `ls`-Version kann diesen Sonderfall ignorieren und die Größe ausgeben (die bei Gerätedateien immer 0 ist).

Für den Rechte-String (z. B. `-rwxrw-rw-`) verwenden Sie am besten einen zehn Zeichen langen String, den Sie schrittweise zusammensetzen. Die Zugriffsrechte sind in `st_mode` (in `stat`) gespeichert. Um z. B. zu testen, ob der Besitzer der Datei Leserechte hat, können Sie diesen Code verwenden:

```
rights[1] = (s.st_mode & 0400) ? 'r' : '-';  // -r-------- ?
```

Die Konstruktion mit `? … :` ist eine Kurzschreibweise: `a = cond ? b : c` steht für

```
if (cond)
  a = b;
else
  a = c;
```

Etwas schwieriger ist die Überprüfung der Attribute SUID, SGID und Sticky-Bit. Deren Bedeutungen hängen davon ab, ob bestimmte Ausführrechte gesetzt sind oder nicht. So soll in der Ausgabe z. B. ein `s` erscheinen, wenn die Datei ausführbar ist und das SUID-Bit gesetzt ist. Ist sie nicht ausführbar (aber das SUID-Bit gesetzt), erscheint hingegen ein großes `S`.

b) Für den Fall, dass Sie einen symbolischen Link finden, sollen Sie nun auch das Link-Ziel ausgeben (wie es auch das normale `ls`-Kommando tut). Dafür verwenden Sie die Bibliotheksfunktion `readlink()`.

Beispielausgaben einer Musterlösung:

```
spsh$ ls -ild a.out
Process 88958 launched: foreground
8300579 -rwxr-xr-x  1 user  user  13776  6 Mai 11:30 a.out
spsh$ ls a.out
8300579 -rwxr-xr-x 1 user user 13776 2013/05/06 11:30:29 a.out
spsh$ ls /dev/fd/0
88980 lrwx------ 1 root root 64 2013/05/06 11:41:37 /dev/fd/0 -> /dev/pts/1
spsh$ ls /dev/pts/1
4 crw--w---- 1 user tty 0 2013/05/06 11:41:37 /dev/pts/1
```

(Der erste `ls`-Aufruf hat zwei Argumente, darum startet die Shell das externe `ls`-Programm.)