LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Grundlagenpraktikum: Rechnerarchitektur

Arbeitsblatt 7
06.06.2022 - 12.06.2022

T7.1 File IO

In dieser Aufgabe wollen wir uns dem Öffnen und Lesen von Dateien widmen. Hierzu werden wir das Programm `toupper_simd` so erweitern, dass der Nutzer auch eigene Eingabedateien spezifizieren kann.

Hinweis: Ziehen Sie ggf. die relevanten `man` Pages für weitere Informationen heran. Achten Sie auch stets auf eine geeignete Fehlerbehandlung und hilfreiche Fehlermeldungen.

Vorlage: `https://gra.caps.in.tum.de/m/toupper_simd.tar` – Für die Bearbeitung der Aufgabe müssen Sie lediglich die Funktionen `read_file` und `write_file` bearbeiten.

1. Implementieren Sie zunächst die Funktion `read_file`. Öffnen Sie hierzu mit `fopen` die Datei, deren Pfad der Funktion übergeben wurde.

```
14 static char* read_file(const char* path) {
15     char* string = NULL;
16      file;
17
18     if (!(file = fopen(path, "r"))) {
19         perror("Error opening file");
20         goto cleanup;
21     }
22
23 cleanup:
24     if (file)
25       fclose(file);
26
27     return string;
28 }
```

2. Finden Sie heraus, wie Sie die Funktion `fstat` verwenden können, um die Größe der Datei zu bestimmen (`man 2 fstat`). Wie können Sie zudem feststellen, ob es sich bei der Datei um eine reguläre Datei handelt?

   Das Feld `st_size` des `statbuf` Out-Parameters wird auf die Größe der Datei gesetzt. Ob es sich um eine gültige Datei handelt lässt, sich mit dem `S_ISREG` Makro und dem `statbuf` Feld `st_mode` überprüfen. Da `fstat` anstatt eines File Pointer allerdings einen File Descriptor als ersten Parameter erwarten, müssen wir die zuvor geöffnete Datei noch mit `fileno` in einen solchen umwandeln.

```
22      ...
23      struct stat statbuf;
24      if (fstat(fileno(file), &statbuf)) {
25          perror("Error retrieving file stats");
26          goto cleanup;
27      }
28      if (!S_ISREG(statbuf.st_mode) || statbuf.st_size <= 0) {
29          fprintf(stderr, "Error processing file: Not a regular file
                or invalid size\n");
30          goto cleanup;
31      }
32
33  cleanup:
34      ...
```

3. Allozieren Sie nun genügend Speicher für den Dateiinhalt und lesen Sie die Datei mit `fread` ein. Was müssen Sie hierbei mit Hinblick auf das terminierende `NULL`-Byte beachten?

   Neben dem eigentlichen Dateiinhalt muss auch noch ein abschließendes `NULL`-Byte in den Rückgabestring geschrieben werden; für dieses muss ein extra Byte Speicher alloziert werden. Weitherhin schreibt `fread` selbst kein `NULL`-Byte in den String, was wir demnach manuell machen müssen.

```
32      ...
33      if (!(string = malloc(statbuf.st_size + 1))) {
34          fprintf(stderr, "Error reading file: Could not allocate
                enough memory\n");
35          goto cleanup;
36      }
37
38      if (!fread(string, 1, statbuf.st_size, file)) {
39          perror("Error reading file");
40          free(string);
41          string = NULL;
42          goto cleanup;
43      }
44
45      string[statbuf.st_size] = '\0';
46
47  cleanup:
48      ...
```

4. Bevor sie den String zurückgeben, denken Sie daran, die geöffnete Datei wieder mit `fclose` zu schließen. Achten Sie darauf, dass die Datei auch im Fehlerfall geschlossen wird!

5. Implementieren Sie nun die Funktion `write_file`. Öffnen Sie hierzu auch in dieser zunächst die Ausgabedatei.

```
54 static void write_file(const char* path, const char* string) {
55      file;
56
57     if (!(file = fopen(path, "w"))) {
58         perror("Error opening file");
59         return;
60     }
61 }
```

6. Schreiben Sie nun den String *ohne* das terminierende `NULL`-Byte in die Datei. Denken Sie auch hier daran, die Datei in jedem Fall zu schließen.

```
61     ...
62     if (!fwrite(string, 1, strlen(string), file)) {
63         perror("Error writing to file");
64     }
65
66     fclose(file);
67 }
```

S7.1 Zeitmessung

Um verschiedene Implementierungen zur Lösung eines Problems vergleichen zu können, wird oftmals die benötigte Laufzeit als Kriterium herangezogen. Bei diesem vermeintlich einfachen Problem gibt es jedoch einige Fallstricke, welche beim Benchmarking beachtet werden sollten[1].

- Die C-Standardbibliothek (bzw. Erweiterungen aus dem POSIX-Standard und Linux-spezifische Funktionen) enthält verschiedene Funktionen, um die Zeit zu messen. Wie unterscheiden sich die Funktionen `clock_gettime`, `timespec_get` (Achtung: hat noch keine `man`-Page), `gettimeofday` und `clock`? Welche von diesen Möglichkeiten ist empfehlenswert?
  - *`clock_gettime` (POSIX) gibt die Zeit einer Betriebssystem-Uhr zurück. Je nach Betriebssystem gibt es mehrere Uhren zur Auswahl:*
    - *`CLOCK_REALTIME` beinhaltet mögliche Sprünge der Zeit u.A. durch Schaltsekunden oder anderweitige Korrekturen und gibt die aktuelle Uhrzeit an (was i.d.R. nicht nötig ist).*
    - *`CLOCK_PROCESS_CPUTIME_ID`, `CLOCK_THREAD_CPUTIME_ID` – misst die CPU-Zeit, die jedoch nicht identisch mit der real benötigten Zeit ist und eine deutlich geringere Genauigkeit hat.*

[1] Das trifft selbstverständlich auch auf Ihre Projektaufgabe zu.

  - * `CLOCK_MONOTONIC` ***garantiert, dass die Uhr kontinuierlich fortschreitet, und zwar mit einer Sekunde pro Sekunde, ohne dass jemand dies verändern kann.***
  - * `CLOCK_MONOTONIC_RAW` ***ist im Gegensatz dazu nicht durch NTP-Synchronisierungen beeinflusst, also möglicherweise bei längeren Abständen ungenauer. Ein weiterer Nachteil ist, dass dies Linux-spezifisch ist, also auf anderen Plattformen wie BSD nicht funktionieren muss.***

- `timespec_get` ***(C11) ist die C-Standardisierung von*** `clock_gettime`***; derzeit ist aber nur*** `TIME_UTC` ***spezifiziert, welches*** `CLOCK_REALTIME` ***entspricht. Es existiert ein Proposal[2], für C23 auch*** `TIME_MONOTONIC` ***zu standardisieren.***
- `gettimeofday` ***hat eine geringere Genauigkeit und desweiteren dieselben Probleme wie*** `CLOCK_REALTIME` ***– und beinhaltet Zeitzonen-Veränderungen (und andere Veränderungen an der Uhr).***
- `clock` ***misst die CPU-Zeit, die nicht der real verstrichenen Zeit entspricht (z.B. bei Multi-Threading, System-Calls, etc.), und ist zudem weniger genau. Die Funktion gibt die CPU-Zeit in Mikro-Sekunden zurück – auf 32-Bit Systemen lassen sich damit nicht mehr als 72 Minuten messen, weshalb die Funktion nicht zu empfehlen ist.***

Folglich sollte also `clock_gettime` ***mit*** `CLOCK_MONOTONIC` ***verwendet werden, alternativ wäre auch*** `CLOCK_MONOTONIC_RAW` ***möglich.***

- Warum ist die Verwendung des Befehls `time` oftmals nicht empfehlenswert?

  Dieser misst auch die Zeit, die benötigt wird, um das Programm und Shared-Libraries zu laden sowie andere, unrelevante Operationen wie I/O.

- Weshalb ist das Messen von I/O-Operationen wie `printf` problematisch? Lesen Sie hierzu auch `man 7 pipe` und bedenken Sie, dass auch `stdout` nicht notwendigerweise ein Terminal (auch eine *pipe*) ist, sondern auch ein anderes Programm, ein Socket oder eine Datei sein kann.

  Wenn der pipe-Buffer im Kernel voll ist, wird die I/O-Operation das Programm blockieren. Non-blocking I/O verlagert die Logik zur Behandlung dieses Falls lediglich in das Programm und ist daher i.A. keine geeignete Alternative, um das Problem zu umgehen.

- Überlegen Sie, aus welchen Gründen es zu Ungenauigkeiten bei der Zeitmessung kommen kann – beispielsweise, wenn die Laufzeiten mehrerer Durchläufe stark schwanken. Wie kann man diese Probleme vermeiden?

  - ***Problem: Falsche Ergebnisse – diese kann man nämlich in beliebig kurzer Zeit berechnen.***

    Vermeidung: Auf jeden Fall (!) überprüfen, dass die berechneten Ergebnisse korrekt sind!

[2] `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2647.pdf`

- ***Problem: Inferenz durch Hintergrundprozesse.***

  Vermeidung: Gänzlich vermeiden lässt sich das Problem nicht – das Betriebssystem kann im Hintergrund (unkontrollierbar) andere Aufgaben erledigen und der Zustand des Prozessors ist nicht immer optimal (z.B. wegen Stromsparfunktion). Daher sollte man Zeitmessungen immer min. 3 Mal wiederholen und schauen, dass es keine stark abweichenden Ergebnisse gibt; im Zweifel sind die Messungen öfter zu wiederholen.

  Achten Sie darauf, dass niemand außer Ihnen zur gleichen Zeit Ressourcen auf der zur Zeitmessung verwendeten Maschine in Anspruch nimmt. Die `lxhalle` ***und virtuelle Maschinen sind für Zeitmessungen*** **nicht** ***geeignet!***

- ***Problem: Zu kurzer Zeitabstand zwischen Messpunkten.***

  Vermeidung: Zeitlichen Abstand zwischen Messpunkten erhöhen, i.d.R. min. auf eine Sekunde. Außerdem benötigt die Zeitmessung selbst auch Zeit!

- ***Problem: I/O-Operationen wurden mit gemessen.***

  Vermeidung: I/O-Operationen nicht mit messen.

P7.1 SIMD-Anwendung: ToUpper [3 Pkt.]

In dieser Einheit betrachten wir die Funktion `toupper`, welche in einem String sämtliche Klein-Buchstaben durch die entsprechenden Groß-Buchstaben ersetzt. Optimieren Sie diese Funktion mit den *SSE*-Erweiterungen.

```
1  toupper_simd:
2      test edi, 15
3      jz .Laligned
4  .Lunaligned_loop: // just an adoption of toupper_asm.
5      mov al,[rdi]
6      cmp al,0
7      je .Lend
8      sub al,'a'
9      cmp al,25
10     ja .Lunaligned_cont
11     sub byte ptr [rdi],0x20
12 .Lunaligned_cont:
13     inc rdi
14     test edi, 15
15     jnz .Lunaligned_loop
16     // rdi is now 16-byte aligned
17 .Laligned:
18     movdqa  xmm8, xmmword ptr [rip+end_upper]       // 16 * 0x7b
19     pxor    xmm6, xmm6
20     movdqa  xmm7, xmmword ptr [rip+begin_lower]     // 16 * 0x60
21     movdqa  xmm5, xmmword ptr [rip+diff]            // 16 * 0x20
22     jmp     .Lload
23 .Lconvert:
24     movdqa  xmm2, xmm8
```

```
25      pcmpgtb xmm2, xmm1  // 0xff if 0x7b > c
26      movdqa  xmm3, xmm1
27      pcmpgtb xmm3, xmm7  // 0xff if c > 0x60
28      pand    xmm2, xmm3  // 0xff if 0x7b > c > 0x60
29      pand    xmm2, xmm5  // 0x20 if 0x7b > c > 0x60
30      psubb   xmm1, xmm2  // c -= 0x20 if 0x7b > c > 0x60
31      add     rdi, 0x10
32      movdqa  xmmword ptr [rdi-0x10], xmm1
33  .Lload:
34      movdqa  xmm1, xmmword ptr [rdi]
35      movdqa  xmm4, xmm1
36      pcmpeqb xmm4, xmm6  // 0xff if c == 0
37      pmovmskb eax, xmm4
38      test    eax, eax    // sets ZF if xmm4 is all-zero
39      jz      .Lconvert
40      jmp     toupper_asm
```

P7.2 SIMD-Anwendung: strlen [3 Pkt.]

SIMD kann, wie Sie bereits in den Videos gesehen haben, mit General Purpose Registern durchgeführt werden. Berechnen Sie mit Hilfe von SIMD auf General Purpose Registern die Länge des Strings `str`.

```
size_t strlen(const char* str);
```

Sie haben für diese Aufgabe keinen Zugriff auf SSE. Außerdem ist die Anzahl an verfügbaren Zyklen begrenzt.

```
1  strlen:
2    xor rax, rax
3
4    //To not accidentally run over a page boundary, our address needs to be
         a multiple of 8. Run in a loop until that's the case
5    .LBeginSISDLoop:
6    mov cl, [rdi]
7    jmp 2f
8      1:
9      inc rax
10     inc rdi
11     mov cl, [rdi]
12   2:
13   //If Byte is already zero, stop
14   test cl, cl
15   jz .Lend
16   //If lower three bits are zero, implicitly fallthrough to our SIMD-loop
17   test rdi, 0x7
18   jnz 1b
19
20   mov r8, 0x7F7F7F7F7F7F7F7F
21
```

```
22  .LSIMDLoop:
23  //Begin by now moving 8 Byte into rcx - rdi is already set correctly
24  mov rcx, [rdi]
25  jmp 2f
26  1:
27    add rax, 8
28    add rdi, 8
29    mov rcx, [rdi]
30  2:
31  //Do the comparison - see https://graphics.stanford.edu/~seander/
        bithacks.html#ZeroInWord
32  mov rdx, rcx
33  and rdx, r8
34  add rdx, r8
35  or rdx, rcx
36  or rdx, r8
37  not rdx
38  test rdx, rdx
39  jz 1b
40
41  //A normal SISD loop
42  .LSISDLoop:
43  mov cl, [rdi]
44  jmp 2f
45    1:
46    inc rax
47    inc rdi
48    mov cl, [rdi]
49  2:
50  test cl, cl
51  jnz 1b
52
53  .Lend:
54  ret
```

Q7.1 Quiz [4 Pkt.] (siehe Praktikumswebsite)