

Betriebssysteme und Softwaretechnik

Abgabe: 25.05.2017

Adrian C. Hinrichs Matr.Nr. 367129

Jeremias Merten Matr.Nr. 367626

Georg C. Dorndorf Matr.Nr. 366511

Aufgabe 4

Aufgabe 4.1

a)

Beim Busy Waiting I/O Ansatz übernimmt die CPU den Datentransfer. Während das I/O Gerät arbeitet wartet dabei die CPU bis sie weitere Daten transferieren kann.

Beim Interrupt I/O Ansatz beauftragt die CPU einen Device Controller, der dann den Datentransfer übernimmt. Die CPU steuert dabei die genauen Einzelheiten. Der Device Controller sendet nach jeder beendeten I/O Aktion ein Interrupt und ein Interrupt Handler der CPU unterbricht den aktuellen Prozess und weist dem Device Controller die nächste I/O Aktion zu. Danach nimmt die CPU den vorher unterbrochenen Prozess wieder auf.

Beim Ansatz der Direct Memory Access (DMA) wird ein zusätzlicher DMA-Controller in der Hardware benötigt. Hier beauftragt die CPU ähnlich wie beim Interrupt I/O Ansatz den DMA-Controller den Datentransfer auszuführen. Allerdings programmiert die CPU hierfür den gesamte I/O Vorgang in dem DMA-Controller und wird dann lediglich nach Beendigung des Vorgangs mittels eines Interrupts informiert.

Die Unterschiede zwischen den Ansätzen bestehen darin in wie weit die CPU direkt mit dem Datentransfer beschäftigt ist.

Bei den ersten beiden Ansätzen tritt das Problem auf, dass die CPU beim Transfer großer Datenmengen nicht mehr effizient arbeiten kann. Dies liegt daran, dass sie entweder ständig wartet (Busy Waiting I/O) oder häufig von ihrer Arbeit unterbrochen wird (Interrupt I/O). Der Vorteil des DMA Ansatzes ist, dass die CPU während den I/O Aktionen nicht ständig unterbrochen wird und weiterhin die anderen Prozesse bedienen kann. Trotzdem behält sie die Kontrolle über die Details des Transfers, da sie den DMA für jede I/O Aktion programmiert.

b)

`.\count <file>` zählt die Anzahl der in der Datei enthaltenen 7. Das Programm verwendet DMA-Ansatz um die Datei nach den sieben zu durchsuchen. Umgesetzt ist der Ansatz mithilfe des `mmap()` Befehls, der den Physikalischen Speicher in den virtuellen Speicher des Prozesses mappt. Dadurch kann das Programm dann über die programmierten virtuellen Adressen direkt auf den Speicher zugreifen.

`.\up` ändert jeden Kleinbuchstaben, den man in das

Terminal eingibt zu einem Großbuchstaben und gibt dann nur den Großbuchstaben aus. Es nutzt den Busy Waiting Ansatz und fragt in einer Endlosschleife den Input des Terminals ab. Es setzt dabei den Busy Waiting ansatz um indem es mithilfe eines speziellen File-Descriptors und einem besonderen Asynchronen Terminal zugriff ständig auf Änderungen prüft.

`.\tcpconnect <ip> <port>` verbindet sich mit einer IPv4-Adresse und einem Port zu einem Server. Dann gibt es die 4096 Bit großen Datenpakete, die der Server (insbesondere der Beispielservers) sendet auf der Konsole aus. Falls der Server nichts sendet wird auch nichts ausgegeben. Diese Funktion setzt das Programm mithilfe des Interrupt gesteuerten Ansatzes um. Das Programm setzt den Ansatz mithilfe der `signal()`-Funktion um. Durch diese Funktion wird der Prozess vom Prozessor benachrichtigt, wenn der Server ein neues Datenpaket gesendet hat und kann dies dann ausgeben.

Die Implementation von `up.c` arbeitet äußerst ineffizient, da sie ständig auf neuen Input prüft anstatt sich benachrichtigen zu lassen, wenn ein neuer Input erfolgt ist. Man könnte das Programm verbessern, indem ich einen Interrupt gesteuerten Ansatz verwenden würde. Dann würde keine auf Input prüfende Endlosschleife die CPU-Zeit beanspruchen.

Aufgabe 4.2

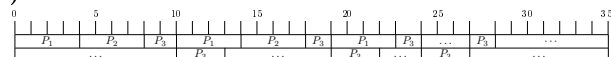
a)

Die erste Zeile stellt die CPU-Bedienzeit dar und die zweite die I/O-Zeit der Prozesse.

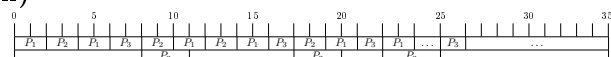
i)



ii)



iii)



b)

Die dritte Scheduling-Strategie maximiert die CPU-Auslastung, da durch das kurze Quantum für einen fertigen I/O-Prozess viel öfter die Möglichkeit auftritt sich in die FIFO-Warteschlange einzureihen. Umgekehrt sorgt das kurze Quantum aber auch dafür, dass ein Prozess nicht von anderen Prozessen aufgehalten wird obwohl er selbst nur kurze CPU-Zeit benötigt. Dadurch kann dieser Prozess dann seine I/O Zeit parallel

zu anderen Prozessen starten. Dadurch reduzieren sich die Phasen in denen ein Prozess abwechselnd CPU- und I/O-Zeit nutzt ohne, dass ein anderer Prozess die CPU nutzt. Dadurch ist mithilfe dieser Scheduling-Strategie die CPU-Auslastung im Vergleich zu i) und ii) immer optimaler.

Aufgabe 4.3

Aufgabe 4.4

a)

Zunächst initialisiert der Vaterprozess einen geteilten Speicherblock mit dem Befehl `shmget(IPC_PRIVATE ↪ , SEGSIZE, IPC_CREAT|0644);`, diesen bindet er dann mittels der Codezeile `shar_mem = (int*)shmat(id, 0, 0);` an seinen eigenen Speicherbereich, so dass er darauf zugreifen kann. Der Vaterprozess forkt nun `NUM_OF_CHILDS` viele Kind-Prozesse, welche gemeinsam den im geteilten Speicherbereich abgelegten Counter auf `MAXCOUNT` hochzählen. Jeder Kind-Prozess zählt dabei wie oft er den geteilten Counter erhöht hat, und gibt dies anschließend aus.

Der Vaterprozess wartet nun mittels `waitpid(pid[i], NULL ↪ , 0);` ($\forall i \in [0, \text{NUM_OF_CHILDS}]$) auf die Terminierung aller Kindprozesse.

Zum Schluss wird noch der geteilte Speicher mittels `shmdt(shar_mem);` vom Prozess getrennt und via `shmctl(id ↪ , IPC_RMID, 0);` wieder freigegeben.

b)

Beispielaufrufe:

```
$ ./ipc
132586 Durchläufe wurden benötigt.
364568 Durchläufe wurden benötigt.
0 Durchläufe wurden benötigt.
863507 Durchläufe wurden benötigt.
$ ./ipc
120833 Durchläufe wurden benötigt.
0 Durchläufe wurden benötigt.
0 Durchläufe wurden benötigt.
879168 Durchläufe wurden benötigt.
$ ./ipc
1000000 Durchläufe wurden benötigt.
0 Durchläufe wurden benötigt.
0 Durchläufe wurden benötigt.
0 Durchläufe wurden benötigt.
```

Es fällt auf, dass

1. Nicht alle Kind-Prozesse gleich viel Arbeit verrichten
2. Unter Umständen werden über 1000001 Durchläufe benötigt, um die Zahl 1000000 zu überschreiten.

Der erste Punkt lässt sich trivialerweise durch das in der Realität nicht gerecht verteilende Scheduling erklären.

Der zweite Punkt ist die Folge einer RACE-CONDITION. Die Kindprozesse greifen (quasi) gleichzeitig auf die geteilte Variable zu und »überschreiben« dabei den Fortschritt der anderen Prozesse

Aufgabe 4.5

a)

b)