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 Ansatzt den DMA-Controller den Datentransfer auszuführen. Allerding 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 enthaltenden 7. Das Programm verwendet DMA-Ansatz um die Datei nach den siebenen 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 dan 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 Awiting 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-Adress und einem Port zu einem Server. Dann gibt es die 4098Bit großen Datenpackete, die der Server (insbesondere der Beispielserver) 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 benachrichtig, wenn der Server ein neues Datenpaket gesendet hat und kann dies dann ausgeben.

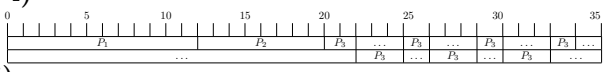
Die Implementation von `up.c` arbeitet äußerst ineffizient, da sie ständig auf neuen Input prüft anstatt sich benachrichtigen zu lassen, wenn ein neuer Input erfolgt ist. Man könnte das Programm verbessern, indem ich einen Interrupt gesteuerten Ansatz verwenden würde. Dann würde keine auf Input prüfende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

Aufgabe 4.2

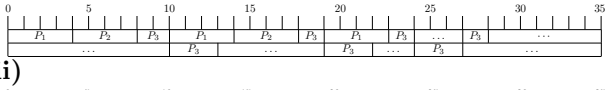
a)

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

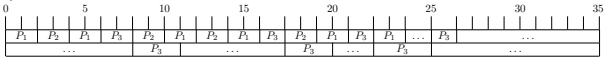
i)



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 paralell

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

t	Kl.0 FIFO(1)	Kl.1 $RR_2(4)$	Kl.2 $RR_6(16)$	Kl.3 FIFO	Incoming	Running
0	-	-	-	-	A(7),B(6)	-
1	-	A(7)	B(6)	-	C(1)	A(7)
2	C(1)	A(6)	B(6)	-	D(2)	C(1)
3	-	A(6),D(2)	B(6)	-	-	A(6)
4	-	D(2),A(5)	B(6)	-	-	D(2)
5	-	D(1),A(5)	B(6)	-	E(17)	D(1)
6	-	A(5)	B(6)	E(17)	-	A(5)
7	-	A(4)	B(6)	E(17)	-	A(4)
8	-	-	B(6),A(3)	E(17)	-	B(6)
9	-	-	B(5),A(3)	E(17)	-	B(5)
10	-	-	B(4),A(3)	E(17)	F(3)	B(4)
11	-	F(3)	B(3),A(3)	E(17)	-	F(3)
12	-	F(2)	B(3),A(3)	E(17)	-	F(2)
13	-	F(1)	B(3),A(3)	E(17)	G(5)	F(1)
14	G(5)	-	B(3),A(3)	E(17)	-	G(5)
15	-	G(4)	B(3),A(3)	E(17)	H(3)	G(4)
16	-	G(3),H(3)	B(3),A(3)	E(17)	I(3)	G(3)
17	-	H(3),G(2),I(3)	B(3),A(3)	E(17)	-	H(3)
18	-	H(2),G(2),I(3)	B(3),A(3)	E(17)	-	H(2)
19	-	G(2),I(3),H(1)	B(3),A(3)	E(17)	-	G(2)
20	-	G(1),I(3),H(1)	B(3),A(3)	E(17)	-	G(1)

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 eigene 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,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 lässst 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)

Messagesysteme stellen ein Erzeuger-Verbraucher-Problem dar, da die zwei kommunizierenden Systeme auf jeden Fall einen geteilten Speicher haben müssen, auch wenn dieser nicht direkt Erreichbar ist, sondern vom Betriebssystem Verwaltet wird. Dieser Geteilte Speicher ist a) Begrenzt, kann also vollständig gefüllt sein, wenn ein Prozess dauerhaft schreibt und der andere nicht ließt (bzw. das gelesene nicht wieder löscht) und kann b) leer sein, wenn kein Prozess etwas Schreibt, sondern der eine bspw. auf eine User Eingaben wartet, die er dem anderen weiterleiten will.

Festplattencache stellt ein Erzeuger-Verbraucher-Problem dar, da unter anderem Prozesse Daten schneller verarbeiten können, als Festplatten sie schreiben oder lesen können. Dadurch entsteht beim Schreiben von großen Datenmengen unter Umständen ein komplett gefüllter Puffer oder beim Lesen von Daten ein komplett Leerer Puffer.

Tastaturpuffer stellt ein Erzeuger-Verbraucher-Problem dar. Der Tastaturpuffer kann entweder voll sein, wenn Prozesse die Eingaben nicht verarbeiten oder leer sein, wenn der User lange keine Tasten drückt.

b)

Bei der Verwendung eines Ringpuffers treten keine Konsistenzprobleme mehr auf, da bei einem leeren beziehungsweise vollem Puffer der Verbraucher beziehungsweise Erzeugerprozess die Ausführung pausieren kann bis der andere Prozess die Daten verarbeitet hat. Prinzipiell funktioniert dies auch mit mehreren Erzeugern und Verbrauchern, dies ist z.B. sinnvoll, wenn Erzeuger Aufgaben auf mehrere Verbraucher delegieren wollen (IO-Ausgaben etc.). Es muss allerdings sichergestellt werden, dass nicht zwei Erzeuger gleichzeitig eine neues Elemten in den Puffer einreihen beziehungsweise zwei Verbraucher gleichzeitig ein Element entfernen. Dies kann zum Beispiel erreicht werden, in dem die In und Out Zeiger bereits vor der Operation verschoben werden. In diesem Fall muss nur sichergestellt werden, dass das Element welches gerade durch einen Verbraucher bearbeitet wird nicht durch einen Erzeuger überschrieben wird.