

Betriebssysteme

Tutorium 3

Philipp Kirchhofer

`philipp.kirchhofer@student.kit.edu`

`http://www.stud.uni-karlsruhe.de/~uxbtt/`

**Lehrstuhl Systemarchitektur
Universität Karlsruhe (TH)**

11. November 2009

Was machen wir heute?

1 Organisatorisches

2 Tutorien Übungsblatt

- Processes
- Processes in Unix
- Race conditions and Deadlocks

Programmieraufgaben

Programmieraufgabe 1 jetzt verfügbar.

Deadline: 25. November 20 Uhr

Prozesskontrollblock (PCB)

Der PCB speichert Informationen über den Zustand eines Prozesses. Diese Informationen werden benötigt, um nach einer Unterbrechung den Zustand des Prozesses nahtlos wiederherstellen zu können.

Frage 3.1.a

Welche Daten enthält ein Prozesskontrollblock?

Antwort

- Befehlszähler
- Stackpointer
- CPU Register
- Weitere Verwaltungsinformationen
 - Speicherverwaltung
 - Offene Dateien
 - Scheduling

Frage 3.1.b

Was muss bei einem Kontextwechsel zwischen verschiedenen Prozessen durchgeführt werden?

Antwort

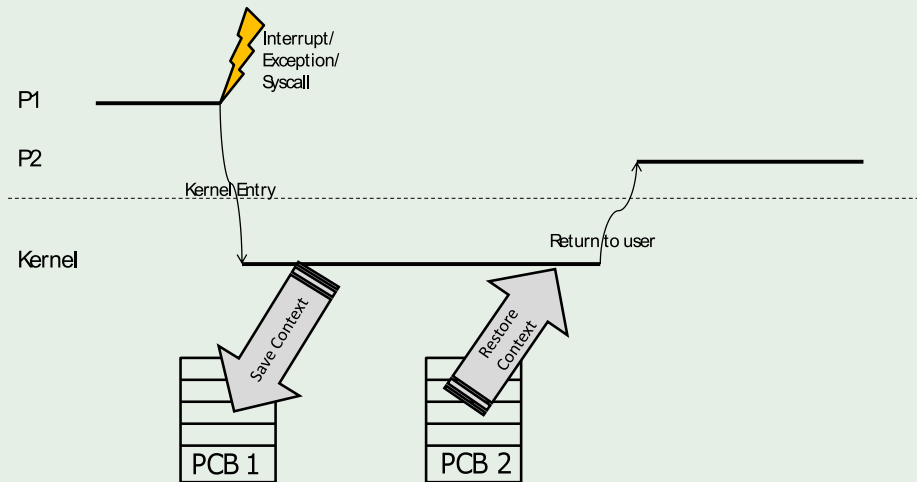
Der Kern muss den Zustand des unterbrochenen Prozess im zugehörigen PCB speichern. Anschließend wird der Zustand des Ziel-Prozesses durch Auslesen des PCB wiederhergestellt.

Zusätzlich können hardwareabhängige Aktionen notwendig sein: Beim Wechsel des Adressraum müssen z.B. Caches (TLB) geleert oder spezielle Register im Prozessor angepasst werden.

Der gesamte Vorgang ist für die beteiligten Prozesse vollständig transparent.

Hinweis: Bei einem Systemaufruf kann eine Anwendung einen Teil des Zustandes (z.B. Register) sichern und nach der Rückkehr in das Programm wiederherstellen. Bei einem Kontextwechsel ist dies nicht möglich.

Beispiel: Kontextwechsel



Frage 3.1.c

Was ist der Unterschied zwischen kurz-, mittel- oder langfristigem Scheduling?

Antwort

Scheduling Strategien

- Kurzfristig - Wählt den nächsten laufenden Prozess aus
- Mittelfristig - Optimiert die Multiprogrammierung durch Ein- und Auslagern von nicht benötigten Prozessen
- Langfristig - Wählt neue Prozesse aus

Frage 3.1.d

Was sind die Unterschiede zwischen Prozessor- und E/A-lastigen Prozessen?

Antwort

Prozessorlastig

- Selten E/A-Operationen
- Verbrauchen viel Rechenzeit
- Geringe Wahrscheinlichkeit zu blocken

E/A-lastig

- Viele E/A-Operationen
- Kurze Rechenphasen
- Hohe Wahrscheinlichkeit zu blocken

Frage 3.1.e

Weshalb ist eine gute Mischung aus Prozessor- und E/A-lastigen Prozessen für die Auslastung des Rechners optimal?

Antwort

Durch eine gute Mischung können alle Ressourcen eines Systems (Speicher, Festplatte, Prozessor) gut ausgelastet werden.

(siehe auch Multiprogrammierung)

Frage 3.2.a

Was macht ein *fork* Systemaufruf?

Antwort

fork erstellt eine Kopie des Ursprungsprozesses. Die Kopie läuft unabhängig weiter. Kindprozess übernimmt von Elternprozess

- Maschinencode
- Daten
- Befehlszähler
- Adressraum Layout
- Offene Dateien

Kindprozess erhält

- Prozessnummer
- Adressraum
- Eltern-ID wird auf ID des Elternprozess gesetzt

Frage 3.2.b

Demo Programm zu *fork*

Antwort

Demo **fork.c**

Frage 3.2.c

Wir möchten eine Shell schreiben. Kann man mit dem *fork* Systemaufruf die gewünschte Funktionalität umsetzen?

Antwort

fork ist für das Starten eines neuen Programms nicht geeignet, da es nur eine Kopie eines schon gestarteten Programms erstellt.

Zur Lösung des Problems nutzt man den Systemaufruf *execve*. *execve* ersetzt das laufende Programm durch ein anderes, behält aber den Adressraum bei.

Übliches Vorgehen beim Starten von neuen Programm

Die Shell erzeugt eine Kopie von sich selber mit Hilfe des *fork* Systemaufrufs, anschließend ersetzt der Kindprozess seinen eigenen Programmcode durch das zu startende Programm.

Frage 3.3.a

Was sind Race Conditions?

Antwort

Ergebnis einer Operation hängt vom zeitlichen Verhalten verschiedener Einzeloperationen ab.

Bedingungen für Eintritt einer Race Condition

Kritischer Abschnitt

Programmabschnitt bearbeitet globale Daten in mehreren Schritten

Parallelverarbeitung

Parallele Abarbeitung eines kritischen Abschnitts

Beispiel: Kritischer Abschnitt

```
current_money = get_balance();  
current_money += delta;  
set_balance (current_money);
```

Beispiel: Race Condition

t	AKTION 1	AKTION 2
0	current_money = 100;	
1	current_money = get_balance();	
2	current_money += 50;	current_money = get_balance();
3	set_balance (current_money);	current_money -= 100;
4		set_balance (current_money);
5	current_money = 0;	

Frage 3.3.b

Wie können Race Conditions vermieden werden?

Antwort

Atomare Ausführung von kritischen Abschnitten

Mutex Verfahren

- Lock
- Semaphore
- Monitor

Tutorien Übungsblatt - Race conditions and Deadlocks

Beispiel: Locking

t	AKTION 1	AKTION 2
0	current_money = 100;	
1	lock(L);	
2	current_money = get_balance();	
3	current_money += 50;	lock(L);
4	set_balance (current_money);	~blocked~
5	unlock(L);	~blocked~
6		current_money = get_balance();
7		current_money -= 100;
8		set_balance (current_money);
9		unlock(L);
10	current_money = 50;	

Frage 3.3.c

Was sind Deadlocks?

Antwort

Mehrere Prozesse warten darauf, dass jeweils ein anderer Prozess beendet wird.

Ein Deadlock kann eintreten wenn folgende vier Bedingungen zutreffen:

- No Preemption
Die Betriebsmittel werden ausschließlich durch die Prozesse freigegeben
- Hold and Wait
Die Prozesse fordern Betriebsmittel an, behalten aber zugleich den Zugriff auf andere
- Mutual Exclusion
Der Zugriff auf die Betriebsmittel ist exklusiv
- Circular Wait
Mindestens zwei Prozesse besitzen bezüglich der Betriebsmittel eine zirkuläre Abhängigkeit

Frage 3.3.d

Wie können Deadlocks vermieden werden?

Antwort

Es kann kein Deadlock mehr auftreten sobald mindestens eine der vier Bedingungen nicht erfüllt ist.

- No Preemption
- Hold and Wait
- Mutual Exclusion
- Circular Wait

Mindestens zwei Prozesse besitzen bezüglich der Betriebsmittel eine zirkuläre Abhängigkeit

Lösung hier: Ordnen der benötigten Ressourcen in einer beliebigen Reihenfolge und anschließende Vergabe in aufsteigender Reihenfolge.

Fragen & Kommentare?



Traffic Deadlock