Vorlesung Betriebssysteme WS 07/08
Aufgabenblatt 3 vom 6. Dezember 2007
(Vorstellung der Lösungen bei den Tutoren bis zum 20.12.07)

Aufgabe 3.1: (15 Punkte)
Beantworten Sie folgende Fragen und erläutern Sie die Antworten Ihrem Tutor.

- Was ist „*busy waiting*“?
- Welche anderen Möglichkeiten „zu warten“ gibt es in einem Betriebssystem?
- Kann „*busy waiting*“ immer vermieden werden?
- An welcher Stelle verwendet der Windows-Kernel „*busy waiting*“?

Aufgabe 3.2: (20 Punkte)
In der Vorlesung wurde der „Bakery“-Algorithmus zur Realisierung von wechselseitigem Ausschluss vorgestellt. Implementieren Sie den Algorithmus innerhalb des bereitgestellten Programmrahmens unter Windows!

- `bakery.h` enthält die Schnittstellendeklarationen der zu implementierenden Funktionen.
- `main.c` enthält ein Testprogramm.

Erläutern Sie Ihrem Tutor Ihre Implementierung und führen Sie Ihr Programm vor!
Erstellen Sie ein Makefile für **`nmake`** mit dem Ziel **`bakery.exe`**. Verpacken Sie alle Source-Code Dateien (inklusive Makefile) in eine ZIP-Datei (blatt32.zip). Diese ZIP-Datei muss mindestens 24 Stunden vor dem Tutoriumstermin in das Abgabesystem eingestellt werden.

Aufgabe 3.3: (30 Punkte)
Schreiben Sie in der Programmiersprache C eine Kommando-Shell für Windows. Die Shell soll Kommandos aus einer Datei (1. Übergabe-Parameter) oder von der Standardeingabe zeilenweise einlesen. Jede Zeile enthält genau ein Kommando: ein ausführbares Programm oder ein Shellkommando. Ein ausführbares Programm zeichnet sich durch einen Pfad zum Programm oder durch einen Pfad zum Programm gefolgt von mit einem Leerzeichen getrennten „Und“ (&) aus:

- `C:\windows\system32\notepad.exe` startet das Programm notepad.exe als neuen Prozess und wartet blockierend auf dessen Beendigung.
- `c:\windows\system32\calc.exe &` startet das Programm calc.exe als neuen Prozess und steht danach für weitere Kommandos bereit.

Ein Shellkommando beginnt mit einem Doppelpunkt gefolgt vom eigentlichen Kommando. Es sollen folgende Kommandos unterstützt werden:

- `:wait` wartet blockierend auf die Beendigung aller nebenläufig (also mit &) gestarteten Prozesse
- `:exit` beendet die Shell

Beispiel:

```
> type skript.txt
C:\windows\system32\notepad.exe
C:\windows\system32\calc.exe &
C:\windows\system32\charmap.exe &
:wait
C:\windows\system32\sol.exe
:exit
> min_shell.exe script.txt
```

Verwenden Sie zur Implementierung die folgenden API-Funktionen: *CreateProcess(), WaitForSingleObject(), WaitForMultipleObjects(), GetLastError(), FormatMessage(), fgets(), fopen(), fclose(), ferror(), perror(), strerror(), strlen()* und *strcmp()*. Auftretende Fehler sollen durch die Ausgabe der Systemfehlermeldung, wenn verfügbar, auf die Standardfehlerausgabe dem Benutzer angezeigt werden.

Erläutern Sie Ihrem Tutor Ihre Implementierung und führen Sie Ihr Programm vor!

Erstellen Sie ein Makefile für **`nmake`** mit dem Ziel **`min_shell.exe`**. Verpacken Sie alle Source-Code Dateien (inklusive Makefile) in eine ZIP-Datei (blatt33.zip). Diese ZIP-Datei muss mindestens 24 Stunden vor dem Tutoriumstermin in das Abgabesystem eingestellt werden.

Aufgabe 3.4: (15 Punkte)
In der Vorlesung wurden Algorithmen zur Synchronisation von mehreren Threads/Prozessen besprochen. In dieser Aufgabe geht es um die Erzeugung von Prozessen. Machen Sie sich mit der Semantik des **`fork`**- und **`wait`**-Systemrufs vertraut. Betrachten Sie anschließend das folgende Programm. Welche Prozesse existieren zu den Zeitpunkten t={10s, 30s, …, 110s} im System? Fertigen Sie zusätzlich einen Prozessgraph (Abbildung 1) an. Zur Vereinfachung wird angenommen, dass für die Systemrufe keinerlei Zeit beansprucht wird.

```
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

void main(void)
{
        sleep(20);
        fork();
        sleep(20);
        fork();
        wait(NULL);
        sleep(20);
        fork();
        sleep(20);
}
```

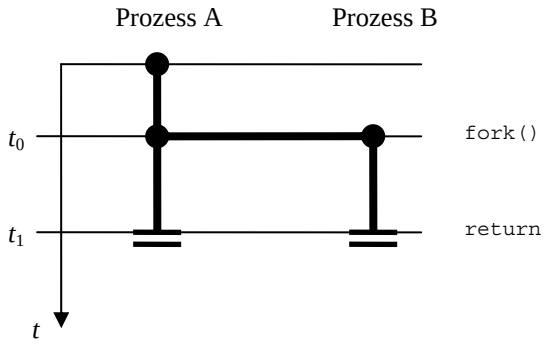

Abbildung 1: Prozessgraph

Bitte bringen Sie das Diagramm in Papierform zum Tutoriumstermin mit und erläutern Sie es Ihrem Tutor!

Aufgabe 3.5: (20 Punkte)
Zwei parallel ablaufende Prozesse (der Erzeuger und der Verbraucher) verwenden einen gemeinsamen, begrenzten Speicherbereich. Der Erzeuger schreibt Daten in den gemeinsamen Speicher, der Verbraucher entnimmt Daten.
Kann der Erzeuger kein neues Datenelement speichern, verwendet er die **`sleep`**-Funktion und wartet, bis er vom Verbraucher wieder aufgeweckt wird nachdem dieser ein Datenelement entfernt hat.
Analog wartet der Verbraucher wenn keine Datenelemente vorhanden sind bis er vom Erzeuger geweckt wird.

Nachfolgend sind (Pseudo-Code) Implementierungen für den Erzeuger und den Verbraucher dargestellt:

```
#define N 100  // Puffergroesse
int count = 0; // Anzahl der Puffereintraege
```

```
void producer(void) {

  int item;

  while (TRUE) {
    produce_item(&item); // neuer Eintrag
    if (count == N)      // Puffer voll?
      sleep();
    insert_item(item);   // schreibe Eintrag
    count = count+1;     // Anzahl erhoehen
    if (count == 1)      // war Puffer leer?
      wakeup(consumer);
  }
}
```

```
void consumer(void) {

  int item;

  while (TRUE) {
    if (count == 0)      // Puffer leer?
      sleep();
    remove_item(&item); // Eintrag lesen
    count = count-1;    // Anzahl vermindern
    if (count == N-1)   // war Puffer voll?
      wakeup(producer);
    consume_item(item); // Eintrag bearbeiten
  }
}
```

Analysieren Sie das Programmfragment und beantworten Sie Ihrem Tutor folgende Fragen:

- Da der Erzeuger- und der Verbraucherprozess parallel ablaufen, können sie vom Scheduler unterbrochen werden. Finden Sie zwei unterschiedliche parallele Programmabläufe, die dazu führen, dass sowohl der Erzeuger- als auch der Verbraucherprozess mittels **`sleep`** warten!
- Markiert man den gesamten „while(TRUE)“-Schleifenrumpf als kritischen Abschnitt wird dieses Problem nicht gelöst. Warum?
- Beschreiben und begründen Sie, wie dieses Problem mit Hilfe von Semaphoren gelöst werden kann!