Betriebssysteme **WS 2007/2008**
Dr. Henrik Brosenne Georg-August Universität Göttingen
Institut für Informatik

Übung 2

Aufgabe 1

1. Gegeben seien die Prozesse P_1 bis P_5 mit folgenden *Rechenzeiten* t_i, *Ankunftszeiten* a_i und *Prioritäten* p_i:

Prozesse	P_1	P_2	P_3	P_4	P_5
Ankunftszeit a_i	0	2	3	14	18
Rechenzeit t_i	4	10	8	5	4
Priorität p_i	mittel	niedrig	hoch	mittel	hoch

   Die Zeitscheibenlänge sei 4. Skizziere für die folgenden Scheduling-Verfahren den Schedule und berechne die *mittlere Verweilzeit* und die *mittlere Wartezeit*:

   a) *First Come First Serve (FCFS)*,
   b) *Shortest Job First (SJF)*,
   c) *Shortest Remaining Time First (SRTF)*,
   d) *Round Robin*,
   e) *Round Robin mit Prioritäten*.

2. Bei dem obigen theoretischen Beispiel wäre es denkbar, dass mehrere Prozesse gleichzeitig eintreffen oder zeitgleich sowohl eine Zeitscheibe zu Ende geht als auch ein neuer Prozess eintrifft. Wieso muss dies in der Praxis nicht berücksichtigt werden?
   Hinweis. Es soll eine Einprozessormaschine betrachtet werden.

Aufgabe 2

1. Schreibe für eine Linux/Unix-Umgebung ein *C*-Programm, das einen *POSIX-Thread* erzeugt[1]. In diesem Thread sollen in einer Endlosschleife in aufsteigendender Reihenfolge Primzahlen berechnet und ausgegeben werden.[2] Im Hauptprogramm soll parallel dazu in einer Endlosschleife ein einzelnes Zeichen per `getchar` eingelesen werden. Wenn die Enter-Taste gedrückt wurde, soll das Hauptprogramm den erzeugten Thread per `pthread_cancel` beenden und selber terminieren.

[1] Beim Erzeugen des Threads kann als Zeiger auf die `pthread_attr_t`-Struktur `NULL` verwendet werden. Bitte auch das in der man-Page angegebene `#include` beachten. Da die POSIX-Threads nicht Teil der Standardbibliothek sind, muss die POSIX-Thread-Bibliothek explizit beim Linken (Option `-lpthread`) mit angegeben werden.

[2] Zur Primzahlberechnung kann ein simples Brute-Force-Verfahren verwendet werden. Als Datentyp soll `unsigned long int` zum Einsatz kommen. Es kann auch irgendwas anderes als Primzahlen berechnet werden – Hauptsache rechenintensiv!

2. Nun soll die gleiche Funktionalität ohne Threads realisiert werden. D.h., Primzahlen sollen endlos berechnet werden und das Programm soll mittels Enter-Taste terminiert werden können. Verwende hierzu nicht-blockierende Systemaufrufe.
   Hinweis. Mittels folgender Zeilen kann nicht-blockierend geprüft werden, ob auf der Standardeingabe ein Zeichen eingegeben wurde. Die Funktion liefert 0 zurück, falls keine Zeichen vorliegen, 1 falls Zeichen, die anschließend eingelesen werden können, vorliegen:

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/select.h>

int check_stdin()
{
  struct timeval timeout;
  fd_set readmask;
  int no_of_descriptors;

  timeout.tv_sec = 0;
  timeout.tv_usec = 0;
  FD_ZERO(&readmask);
  FD_SET(fileno(stdin), &readmask);

  no_of_descriptors = select(FD_SETSIZE, &readmask, NULL, NULL, &timeout);

  if (no_of_descriptors >= 0)
    {
      return no_of_descriptors;
    }
  else
    {
      perror("select error");
      exit(EXIT_FAILURE);
    }
}
```

3. Vergleiche (rein nach Augenmaß) die Geschwindigkeit der beiden Lösungen: welche Variante ist schneller? Worin könnte der Grund für den Geschwindigkeitunterschied liegen?

4. **Optionale Zusatzfrage.** Wenn man in der oben aufgeführten Funktion statt der generischen Fehlermeldung "`select error`" einen String ausgeben wollte, der dem in der Variable `errno` enthalten Fehlercode entspricht, könnte man den zugehörigen Text mittels der Betriebssystemaufrufe `strerror` bzw. `strerror_r` erhalten. Nur einer von beiden Aufrufe ist jedoch *thread-safe* – woran könnte das liegen?
   Hinweis. Betrachte die Unterschiede bei den Parametern der beiden Aufrufe.