2A Basistechniken: Lösungen

2A.1 Wissens- und Verständnisfragen

1.a.) Application Programming Interface

1.b.) Multiprogramming

1.c.) Mehreren Prozessoren, die über einen Bus miteinander verbunden sind

1.d.) Stapelverarbeitungssystemen mit Spooling, Clustern

1.e.) Ist ein Programm, das ausgeführt wird; hat in UNIX eine eindeutige Nummer

1.f.) An den Vater die PID des Sohns und an den Sohn eine 0

1.g.) Ein Prozess kann mehrere Threads enthalten

1.h.) start()

2.a.) Schnittstellen

2.b.) Spooling

2.c.) Process Control Block (PCB)

2.d.) Kontext

2.e.) wait()

2.f.) sleep()

3.a.) Falsch, denn beim Mehrprogrammbetrieb werden mehrere Prozesse auf einem Prozessor ausgeführt.

3.b.) Wahr; er kann mehrere fork()-Aufrufe durchführen.

3.c.) Falsch; der Vater ist (eindeutig) der Prozess, der ihn mit fork() erzeugt hat.

3.d.) Falsch. UNIX-Prozesse haben voneinander getrennte Speicherbereiche (sofern nicht explizit Shared Memory vereinbart wurde).

3.e.) Wahr; sie warten auf die Terminierung eines Prozesses bzw. Threads.

4.a.) CPU, denn sie ist keine Schnittstelle

4.b.) Multiprogrammingsystem, denn es realisiert Pseudonebenläufigkeit, während die anderen Begriffe echte Nebenläufigkeit betreffen

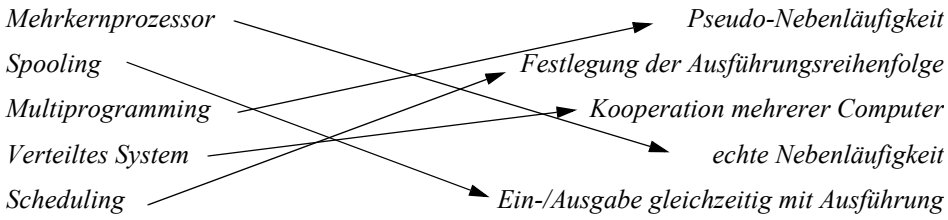
4.c.) Scheduling, denn es ist keine Betriebsart

4.d.) synchronisiert, denn es ist kein Prozesszustand

c.) ps, denn es ist keine Funktion der UNIX/Linux-C-Schnittstelle, sondern ein Benutzerkommando

4.e.) start, denn es ist eine Methode für Java-Threads (nicht, wie die anderen, eine Funktion für UNIX/ Linux-Prozesse)

5.)Begriffe und Eigenschaften:



6.a.) Benutzerschnittstelle und Programmierschnittstelle (API). Über die Benutzerschnittstelle greifen menschliche Benutzer auf Dienste des Betriebssystem zu, über die Programmierschnittstelle greifen Anwendungsprogramme zu.

6.b.) Bei echter Nebenläufigkeit können mehrere Aktionen echt gleichzeitig (also zum selben Zeitpunkt) stattfinden. Bei Pseudonebenläufigkeit kann zu jedem Zeitpunkt immer nur einer Aktion ablaufen; der Prozessor wird aber derart rasch zwischen Aktivitäten (= Folgen von Aktionen) umgeschaltet, dass ein menschlicher Beobachter den Eindruck der gleichzeitigen Ausführung hat. Echte Nebenläufigkeit wird durch nebenläufig arbeitenden Hardware-Komponenten realisiert, Pseudonebenläufigkeit in Software (z.B. durch Prozessen und Threads) mit Unterstützung des Betriebssystems.

6.c.) Programm und Kontext. Die Befehle des Programms werden ausgeführt, wobei die Bestandteile des Kontexts benutzt werden.

6.d.) Prozesse haben jeweils ihre eigenen Kontexte, d.h. sind im Allgemeinen voneinander isoliert. Ein Prozess kann mehrere Threads enthalten, die im gemeinsamen Prozess-Kontext aktiv sind.

6.e.) Registerinhalte inkl. Programmzähler, direkt zugreifbare Speicherbereiche, zugeodnete Peripheriegeräte, geöffnete Dateien.

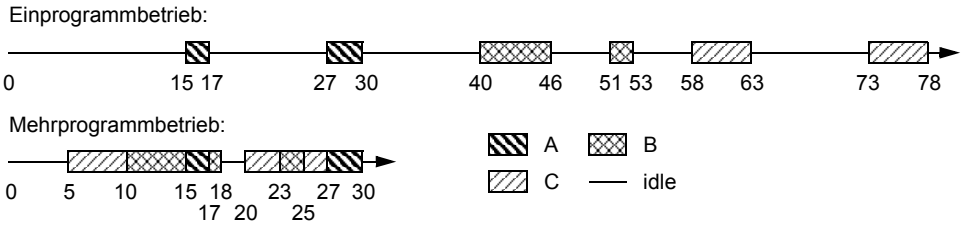
6.f.) Siehe BILD 2.23.

6.g.) Der Parameterwert, den ein terminierender Prozess an exit() übergibt, taucht im Rückgabewert des (Referenzaufruf-)Parameters des zugehörigen wait()-Aufrufs wieder auf.

6.h.) run() definiert die Aktionen, die ein Thread ausführen soll, durch Aufruf von start() wird die nebenläufige Ausführung des Threads gestartet.

2A.2 Sprachunabhängige Anwendungsaufgaben

1.)Zeitlinien:



Tipp für die Zeitlinie des Mehrprogrammbetriebs: Zeichnen Sie zunächst die Ausführungsphasen des Auf-

trags mit der höchsten Priorität (hier: A), denn er wird den Prozessor immer sofort bekommen, wenn er ihn benötigt. Legen Sie dann in die verbleibenden Lücken die Ausführungsphasen des Auftrags mit der nächsthöheren Priorität (hier: B) usw.

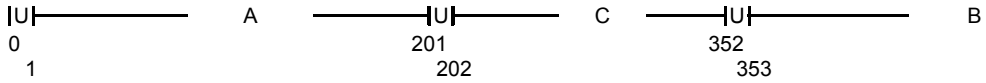
Einprogrammbetrieb:
Aufenthaltsdauer A: 30, B: 53, C: 78 → Mittel: ca. 53,67, Auslastung: ca. 29,5%

Mehrprogrammbetrieb:
Aufenthaltsdauer A: 30, B: 25, C: 27 → Mittel: ca. 27,33, Auslastung: ca. 76,7%

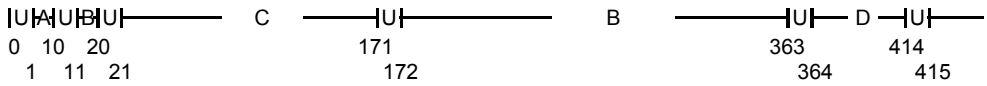
2.)Zeitlinien:

Skizzen nicht maßstabsgerecht. U = Umschaltung.

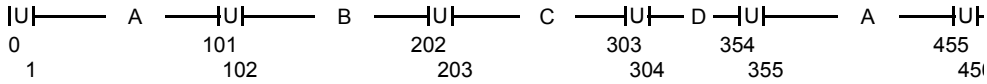
Prioritäten, nichtunterbrechend:



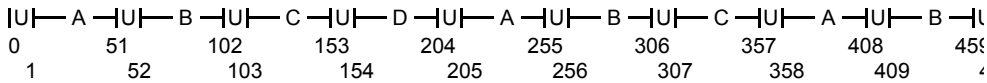
Prioritäten, unterbrechend:



Round Robin, Zeitscheibenlänge 100:



Round Robin, Zeitscheibenlänge 50:



Bei unterbrechendem prioritätengesteuertem Scheduling werden die Aufträge in der Reihenfolge ihrer Prioritäten fertig; bei nichtunterbrechendem Scheduling ist der später eintreffende, hochpriore Auftrag C gegenüber dem niederprioren Auftrag A benachteiligt. Bei Round Robin werden Aufträge mit kürzeren Ausführungszeiten tendenziell schneller fertig als Aufträge mit längeren Ausführungszeiten. Besonders ausgeprägt ist dies bei kurzen Zeitscheibenlängen, wobei allerdings der Aufwand für Umschaltungen größer ist.

2A.3 Programmierung unter UNIX/Linux

1.)Start → bereit (Warten auf Zuteilung des Prozessors) → rechnend (erstes printf) → blockiert (sleep) → bereit (Warten auf erneute Zuteilung des Prozessors) → rechnend (zweites printf) → Terminierung

2.)ps -f

Prozess für die Shell, die die Ausführung der eingegebenen Benutzerkommandos steuert

Zum Beispiel:

BS01 2700 2690 -bash
BS01 2701 2700 prog
BS01 2702 2701 prog

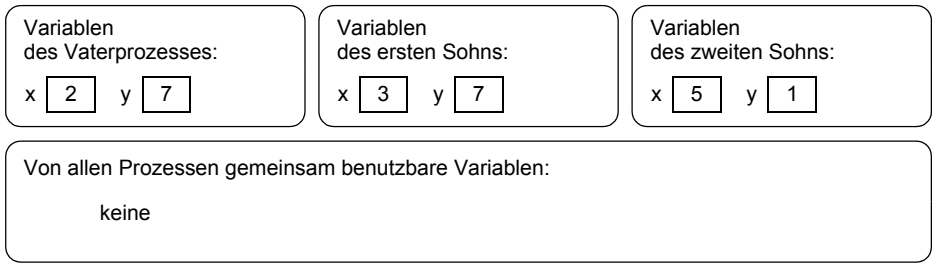
3.)101 100 100 101

0

4.)"if fork()==0" statt "if fork()!=0", "sleep(2)" statt "wait(2)", "exit(0)" am Ende des Sohns ergänzen, "wait(&s)" statt "sleep(&s)".

5.)Die Ausgabe ist eindeutig; sie lautet "i=10". Da Vater und Sohn getrennte Variablen haben, haben die Operationen des Sohns auf i keinen Einfluss auf den Wert von i im Vaterprozess.

6.)Speicherskizze:



Der Vaterprozess wird als letzter fertig, denn er wartet in den wait()-Aufrufen auf das Ende seiner Söhne und terminiert erst dann.

Durch Aufruf von getppid().

Indem er den Rückgabewert von fork() in einer Variablen speichert ("if (sohnpid=fork())==0) ...").

(Mindestens) drei – für jeden Prozess ein eigener.

```
7.)#include <stdio.h>
#include <stdlib.h>
main() {
  int i, status;
  if (fork()==0) {
    sleep(2);
    printf("Hier ist Sohn 1\n");
    exit(0); }
  wait(&status);
  if (fork()==0) {
    for (i=1;i<=20;i++)
      printf("%d ",i);
    exit(0); }
}
```

```
8.)#include <stdio.h>
#include <stdlib.h>
```

```
#include <signal.h>
main() {
 int i, enkel, status;
 if (fork()==0) {
   if((enkel=fork())==0) {
     for (i=0;;i++) {
       printf("%d\n",i);
       sleep(1);
     }
   }
   sleep(15);
   kill(enkel,SIGKILL);
   exit(0);
 }
 wait(&status);
}
```

9.)Lösung zu 7.:

```
#include <pthread.h>
#include <stdio.h>

void *sohn1prog() {
 sleep(2);
 printf("Hier ist Sohn 1\n");
 pthread_exit(0);
}

void *sohn2prog() {
 int i;
 for (i=1;i<=20;i++)
  printf("%d ",i);
 pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
 pthread_t sohn1, sohn2;
 printf("Main erzeugt Sohn 1\n");
 pthread_create(&sohn1, NULL, sohn1prog, NULL);
 printf("Main wartet auf Sohn 1\n");
 pthread_join(sohn1, NULL);
 printf("Main: Sohn1 beendet\n");
 printf("Main erzeugt Sohn 2\n");
 pthread_create(&sohn2, NULL, sohn2prog, NULL);
 pthread_join(sohn2, NULL);
 printf("\nMain: Sohn2 beendet\n");
 printf("Main endet\n");
}
```

Lösung zu 8.:

```
#include <pthread.h>
#include <stdio.h>

void *enkelprog() {
 int i;
 for (i=0;;i++) {
  printf("%d\n",i);
  sleep(1);
 }
}

void *sohnprog() {
 pthread_t enkel;
 printf("Sohn erzeugt Enkel\n");
 pthread_create(&enkel, NULL, enkelprog, NULL);
 sleep(15);
 printf("Sohn terminiert Enkel\n");
 pthread_cancel(enkel);
 pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
 pthread_t sohn;
 printf("Main erzeugt Sohn\n");
 pthread_create(&sohn, NULL, sohnprog, NULL);
 printf("Main wartet auf Sohn\n");
 pthread_join(sohn, NULL);
 printf("Main: Sohn beendet \n");
}
```

Lösung zu 2A.4.5.:

```
#include <pthread.h>
#include <malloc.h>
#include <stdio.h>
#include <string.h>

char string[500];
char  woerter[10][20]  =  {  "Fischers","Fritz","fischt","frische","Fische","frische","Fische","fischt","Fi-
schers","Fritz"};

struct parameter {
 char wort[20];
 int nummer;
 pthread_t wartenAuf;
};

void *threadprog(void *p) {
 struct parameter *pstruct = (struct parameter *) p;
```

```
 if (pstruct->wartenAuf!=0)
  pthread_join(pstruct->wartenAuf, NULL);
 sleep(1);
 printf("Nr. %d haengt an: %s\n", pstruct->nummer, pstruct->wort);
 strcat(string,pstruct->wort);
 strcat(string," ");
 pthread_exit(NULL);
}
main() {
 int i;
 pthread_t t = 0;
 struct parameter *p;
 strcpy(string,"");
 for (i=0;i<10;i++) {
  p = (struct parameter *) malloc(sizeof(struct parameter));
  strcpy(p->wort,woerter[i]);
  p->nummer = i;
  p->wartenAuf = t;
  pthread_create(&t, NULL, threadprog, (void *)p);
 }
 pthread_join(t, NULL);
 printf("%s",string);
}
```

2A.4 Programmierung in Java

1.)Der Thread muss mit t.start() als nebenläufige Aktivität gestartet werden. Mit t.run() wird die run()-Methode des Objekts t durch den Thread des Hauptprogramms ausgeführt.

2.)Die Variable muss als static deklariert werden.

3.)

```
class MyThread extends Thread {
 private Thread wartenAuf;
 private String ausgabe;
 MyThread(Thread wartenAuf, String ausgabe) {
  this.wartenAuf = wartenAuf;
  this.ausgabe = ausgabe;
 }
 public void run() {
  if (wartenAuf!=null)
   try {
    wartenAuf.join();
   } catch (InterruptedException e) {}
  for (int i=0; i<3; i++) {
   try { sleep(500); } catch (InterruptedException e) {  }
```

```
   System.out.println(ausgabe);
  }
 }
}
public class MyProgram {
 public static void main(String[] args) {
  MyThread t1 = new MyThread(null,"Thread 1"),
           t2 = new MyThread(t1,"Thread 2");
  t1.start();
  t2.start();
  try {
   t2.join();
  } catch (InterruptedException e) {}
  System.out.println("Beide Threads fertig");
 }
}
4.)class Enkel extends Thread {
 public void run() {
  for (int i=0;!isInterrupted();i++) {
   System.out.println(i);
   try {
    sleep(1000);
   } catch (InterruptedException e) { this.interrupt(); }
  }
 }
}
class Sohn extends Thread {
 public void run() {
  Enkel enkel = new Enkel();
  enkel.start();
  try {
   sleep(15000);
  } catch (InterruptedException e) {}
  enkel.interrupt();
 }
}
public class Hauptprogramm {
 public static void main(String[] args) {
  Sohn sohn = new Sohn();
  sohn.start();
  try {
   sohn.join();
  } catch (InterruptedException e) {}
 }
```

```
}
5.)class MyThread extends Thread {
 String anzuhaengen;
 Thread wartenAuf;
 MyThread(String anzuhaengen, Thread wartenAuf) {
   this.anzuhaengen = anzuhaengen;
   this.wartenAuf = wartenAuf;
 }
 public void run() {
  if (wartenAuf!=null)
   try {
    wartenAuf.join();
   } catch (InterruptedException e) {}
  MyProgram.satz = MyProgram.satz+" "+anzuhaengen;
 }
}
public class MyProgram {
 static String satz = "";
 public static void main(String[] args) {
  String woerter[] = { "Fischers","Fritz","fischt","frische","Fische","frische","
                                                    Fische","fischt","Fischers","Fritz"};
  MyThread t = null;
  for (int i=0;i<10;i++) {
    t = new MyThread(woerter[i],t);
    t.start();
  }
  try {
   t.join();
  } catch (InterruptedException e) {}
  System.out.println(satz);
 }
}
6.)import java.util.concurrent.atomic.AtomicInteger;
class ThreadA extends Thread {
 public void run() {
  while (!isInterrupted())
   MyProgram.a.incrementAndGet();
 }
}
class ThreadB extends Thread {
 public void run() {
  while (!isInterrupted())
   MyProgram.a.decrementAndGet();
 }
```

```
}
public class MyProgram {
 static AtomicInteger a = new AtomicInteger();
 public static void main(String[] args) {
  ThreadA ta = new ThreadA();
  ThreadB tb = new ThreadB();
  ta.setPriority(Thread.MIN_PRIORITY);
  tb.setPriority(Thread.MIN_PRIORITY);
  ta.start();
  tb.start();
  System.out.println("Prio A: "+ta.getPriority()+"  Prio B: "+tb.getPriority());
  for (int i=0;i<50;i++) {
   try {
    Thread.currentThread().sleep(200);
   } catch (InterruptedException e) {  }
   System.out.println(a.get());
  }
  ta.interrupt();
  tb.interrupt();
  try  {
   ta.join();
   tb.join();
  } catch (InterruptedException e) {  }
  System.out.println("------------------");
  a.set(0);
  ta = new ThreadA();
  tb = new ThreadB();
  ta.setPriority(Thread.MIN_PRIORITY+4);
  tb.setPriority(Thread.MIN_PRIORITY);
  System.out.println("Prio A: "+ta.getPriority()+"  Prio B: "+tb.getPriority());
  ta.start();
  tb.start();
  for (int i=0;i<50;i++) {
   try {
    Thread.currentThread().sleep(200);
   } catch (InterruptedException e) {  }
   System.out.println(a.get());
  }
  ta.interrupt();
  tb.interrupt();
  try  {
   ta.join();
   tb.join();
  } catch (InterruptedException e) {  }
  System.out.println("------------------");
```

```
    }
}
```

Bei gleicher Priorität würde man erwarten, dass sich der Wert von a bei 0 einpendelt. Das ist aber nicht immer so – offensichtlich erhalten die Threads trotz gleicher Priorität nicht immer denselben Anteil an Prozessorzeit. Erhöht man jedoch die Priorität eines Threads stark gegenüber der des anderen, so wird er tatsächlich bevorzugt.