Rechnerstrukturen und Betriebssysteme 2
Übung (Sommersemester 2025)
Prof. Dr. Erik Kamsties
Dipl.-Inform. Michael Hoffmann

Fachhochschule Dortmund
University of Applied Sciences and Arts
Fachbereich Informatik

Übung 6

Aufgabe 1

Die Prozesse P_A, P_B und P_C greifen mit den nachstehend angegebenen Operationen auf die zählenden Semaphore S_A, S_B und S_C zu.

Prozess P_A	Prozess P_B	Prozess P_C
$P(S_A)$	$P(S_B)$	$P(_{SC})$
$P(S_A)$	.	$P(_{SC})$
$P(S_A)$	.	$P(S_C)$
.	.	.
.	.	.
.	$V(S_C)$	$V(S_B)$
$V(S_B)$	$V(S_A)$	$V(S_B)$
END	END	END

Die Semaphore sind für die vier Fälle wie in der Tabelle angegeben initialisiert.

Fall	a)	b)	c)	d)
S_A	2	3	2	0
S_B	0	0	1	0
S_C	2	2	1	3

Geben Sie für die Fälle a), b), c) und d) den Ablauf der drei Prozesse an. Nehmen Sie an, dass die Prozesse nach dem Round-Robin Verfahren in der Reihenfolge P_A, P_B, P_C ausgeführt werden. Eine Umschaltung zum nächsten Prozess erfolge erst, wenn der aktuelle Prozess terminiert oder blockiert.

a)

S_A	S_B	S_C	P_A	Zustand	P_B	Zustand	P_C	Zustand
2	0	2		R				
1	0	2	$P(S_A)$	R				
0	0	2	$P(S_A)$	R				
0	0	2	$P(S_A)$	B				

S_A	S_B	S_C	P_A	Zustand	P_B	Zustand	P_C	Zustand
0	0	2			$P(S_B)$	B		
0	0	1					$P(S_C)$	R
0	0	0					$P(S_C)$	R
0	0	0					$P(S_C)$	B

Alle Prozesse blockieren.

b) A⇢B⇢C
c) B⇢A | C Blockiert (3. Anweisung)
d) C⇢B | A Blockiert (2. Anweisung)

Aufgabe 2

Gegeben sind zwei Prozesse P_A und P_B, die auf eine gemeinsame Variable `V` zugreifen, wobei P_A nur schreibend und P_B nur lesend auf die Variable zugreift.

Ergänzen Sie den nachfolgenden Pseudocode um geeignete Semaphor-Operationen (`P()`, `V()`), um den Zugriff auf die Variable zu synchronisieren.

```
Prozess PA                      Prozess PB
semaphore lock = 1;
P(lock);                        P(lock);
V = ... ;                       printf(V);
V(lock);                        V(lock);
```

Aufgabe 3

Wie muss der Semaphor initialisiert werden?

Der Semaphor muss mit 1 initialisiert werden.

Aufgabe 4

Nun soll der lesende Prozess solange blockiert werden, bis ein neuer Wert in die Variable geschrieben wurde und der schreibende Prozess, bis die Variable ausgelesen wurde.

Ergänzen Sie das Programm durch die entsprechenden Semaphor-Operationen.

```
Prozess PA                  Prozess PB
semaphore leer = 1;
semaphore voll = 0;
P(leer);                    P(voll);
```

```
V = ... ;                    printf(V);
V(voll);                     V(leer);
```

Aufgabe 5

Wie müssen die Semaphore initialisiert werden?

Der Semaphor `leer` mit 1, der Semaphor `voll` mit 0.

Aufgabe 6

Das angegebene Code-Fragment startet einen POSIX Thread und wartet auf seine Terminierung:

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

void *zeige(void *arg);

int main (void) {
    int status;
    int data=3;
    pthread_t thread;

    /* Starte den Thread */
    status=pthread_create(&thread,NULL,zeige,&data);
    if(status!=0) {
        fprintf(stderr,"Error: Konnte Thread nicht erzeugen\n");
        exit(-1);
    }

    /* Warte auf Terminierung */
    pthread_join(thread,NULL);
    exit(0);
}

void *zeige(void *arg) {
    int *dat = (int*)arg;
    printf("Habe folgende Daten erhalten: %d\n",*dat);
    return NULL;
}
```

Was tut dieses Programm genau? Welche Bildschirmausgabe wird erzeugt?

Das Programm erzeugt einen Thread, der die Funktion zeige ausführt und wartet auf dessen Terminierung. Dem Thread wird in der Variablen data der Wert 3 übergeben. Diesen Wert schreibt der Thread auf den Bildschirm. Die Bildschirmausgabe lautet:

```
Habe folgende Daten erhalten 3
```

Aufgabe 7

Wann terminiert das Hauptprogramm?

Das Hauptprogramm wartet mit `pthread_join(thread,NULL)` auf die Terminierung des Threads. Das Hauptprogramm terminiert also nach der Beendigung des Threads, d.h. nach der Bildschirmausgabe.

Aufgabe 8

Worin liegen die wesentlichen Vorteile von Threads gegenüber nebenläufigen Prozessen (mittels `fork()` erzeugt)? Wo liegen die Gefahren?

Da beim Aufruf eines Threads (leichtgewichtiger Prozess) im Gegensatz zum (schwergewichtigen) Prozess keine Kopie der Umgebung erstellt wird, erfolgt die Erzeugung von Threads wesentlich schneller. Damit eignen sich Threads auch für eine sehr fein-granulare Parallelisierung, während Prozesse eher für umfangreiche Aufgaben geeignet sind.

Ein Thread kann auch sehr einfach mit dem aufrufenden Programm kommunizieren, da es im selben Speicherbereich abläuft und Zugriff auf die lokalen Daten des aufrufenden Programms hat. Damit vereinfacht sich die Interprozess-Kommunikation.

Durch den gemeinsamen Speicherbereich entsteht aber auch die Gefahr, dass ein Thread durch Programmierfehler die Daten des aufrufenden Programms beschädigt. Diese Gefahr steigt bei komplexen, unübersichtlichen Aufgaben, die deshalb weniger für die Implementierung mit Threads geeignet sind.