Lab 13: Multi Processor Systems II

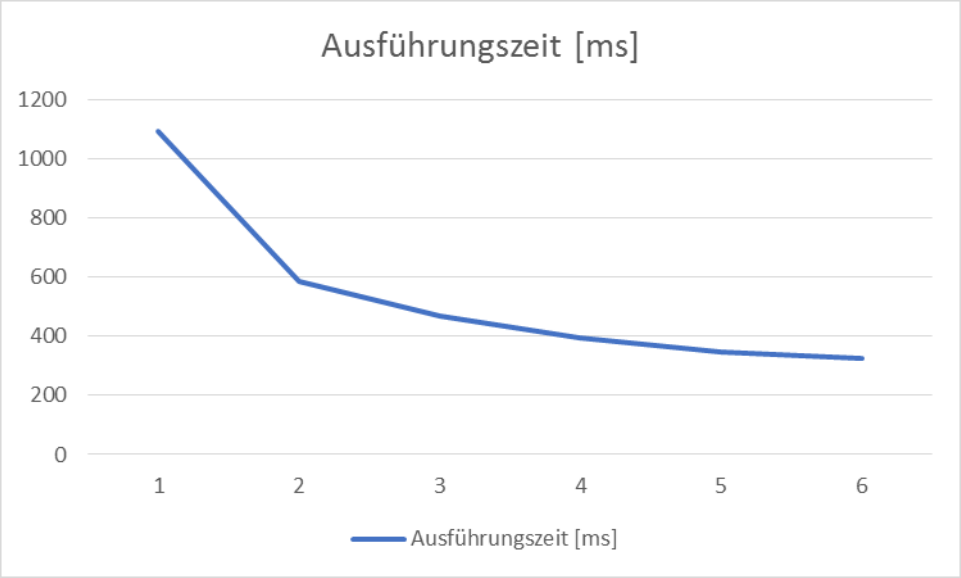
1. Können Sie sich erklären warum die Summe nicht 200% ergibt? Warum entspricht die Auslastung nicht 100% pro Prozessor?
   100% ist die gesamte Auslastung vom System – alle Prozessoren zusammen gerechnet.

2. Lassen wir das Mandelbrot Programm erneut laufen, erzeugen aber zwei Threads welche parallel die Berechnung durchführen. Was erwarten Sie in Bezug auf die Ausführungszeit?
   Sie sollte sich halbieren, somit also doppelt so schnell sein.

```
tdrohrer@ssm-mp17:~/ssm/gss_3/exercise1 $ ./mandelbrot
Thread  0 calculating -2.000000 .. 1.000000 (real) and -1.000000 .. 0.000000 (imaginary)
Thread  1 calculating -2.000000 .. 1.000000 (real) and 0.000000 .. 1.000000 (imaginary)
Execution finished. Time required for calculation = 586ms.
```

3. Erstellen Sie eine Tabelle und Grafik mit den Ausführungszeiten in Abhängigkeit zu den gestarteten Threads.
   Im System installierte CPUs: 4

Anzahl Threads	1	2	4	8	16	32
Ausführungszeit	*1092 ms*	*586 ms*	*467 ms*	*393 ms*	*347 ms*	*325 ms*



4. Erkennen Sie mit wie vielen Threads die Berechnung am effizientesten abläuft?
   Grundsätzlich je mehr umso besser – bei rund 4 Threads ist das Verhältnis am Besten.

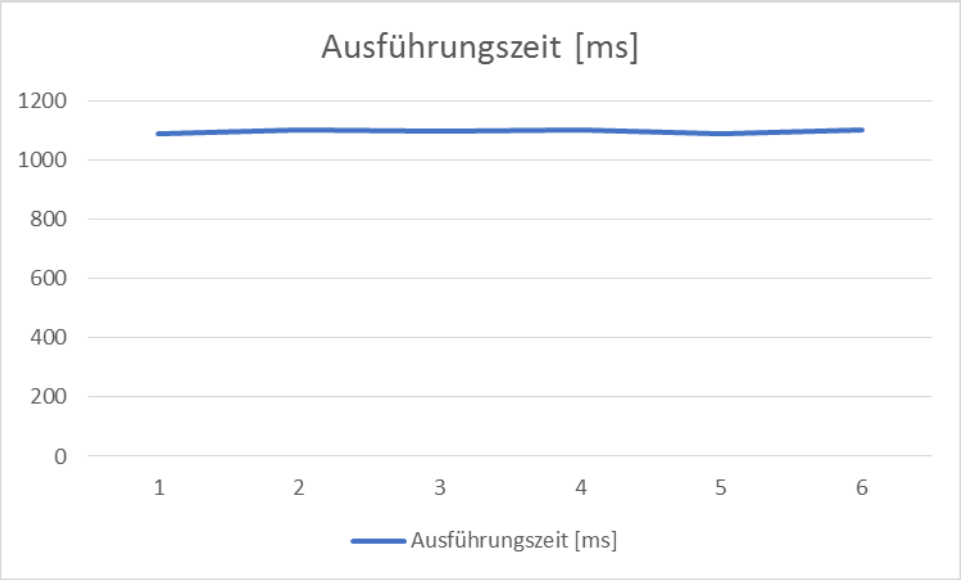
5. Welche Aussage können Sie machen im Bezug auf die Anzahl Threads in Abhängigkeit auf die installierten CPUs?
   Mein System hat 4 CPUs, entsprechend ist es bei 4 Threads optimal.

6. Nicht benötigte CPUs deaktivieren

```
root@ssm-mp17:~ # psrinfo
0       on-line   since 09/26/2013 07:55:34
1       off-line  since 10/10/2013 11:17:09
2       off-line  since 10/10/2013 11:17:11
3       off-line  since 10/10/2013 11:17:12
```

7. Erstellen Sie eine Tabelle und Grafik mit den Ausführungszeiten in Abhängigkeit zu den gestarteten Threads.
*Im System installierte CPUs: **1***

Anzahl Threads	1	2	4	8	16	32
Ausführungszeit	*1090 ms*	*1100 ms*	*1097 ms*	*1100 ms*	*1090 ms*	*1100 ms*



8. Erkennen Sie mit wie vielen Threads die Berechnung am effizientesten abläuft?
Es spielt keine Rolle, das System kann nicht mehr skalieren.

9. Stimmt Ihre Aussage noch im Bezug auf Threads und CPUs?
Nein, die Ausgangslage hat sich verändert.

10. Schalten Sie nun die CPU wieder online:

```
root@ssm-mp17:~ # psradm -n 1
root@ssm-mp17:~ # psradm -n 2
root@ssm-mp17:~ # psradm -n 3
root@ssm-mp17:~ # psrinfo
0       on-line   since 09/26/2013 07:55:34
1       on-line   since 10/10/2013 11:27:15
2       on-line   since 10/10/2013 11:27:21
3       on-line   since 10/10/2013 11:27:23
```

Auslastung des Systems

Special Report – Microstate Accounting

Unlike other operating systems that gather CPU statistics every clock tick or every fixed time interval (typically every hundredth of a second), Solaris 10 incorporates a technology called microstate accounting that uses high-resolution timestamps to measure CPU statistics for every event, thus producing extremely accurate statistics.

The microstate accounting system maintains accurate time counters for threads as well as CPUs. Thread-based microstate accounting tracks several meaningful states per thread in addition to user and system time, which include trap time, lock time, sleep time and latency time. prstat reports the per-process (option -m) or per-thread (option -mL) microstates.

11. Analysieren Sie die Ausgabe von „prstat –mL“. Beachten Sie die Anzeige in der LAT Spalte. Erklären Sie:

```
   PID USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIG PROCESS/LWPID
  4534 tdrohrer  58 0.3 0.0 0.0 0.0 0.0 0.0  42   0 214  39   0 mandelbrot/2
  4534 tdrohrer  41 0.2 0.0 0.0 0.0 0.0 0.0  59   0 158  23   0 mandelbrot/3
```

LAT = CPU latency

Es zeigt an zu wie viel % der Prozess, resp. Thread darauf wartet Rechenzeit zu bekommen.

In diesem Fall 42% resp, 59%

Prozesse und Interprozesskommunikation

12. Schreiben Sie ein Programm, welches zwei neue Prozesse erzeugt, welche jeweils „Hello World“ schreiben, und dann beendet.

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

#define NUM_PROCESSES 2

void HelloWorld( void )
{
  printf( "Hello World from child process: %d\n", getpid() );
}

int main( void )
{
  int i, status;
  pid_t pid[NUM_PROCESSES]; /* array of process IDs */
  for( i = 0; i < NUM_PROCESSES; i++ )
  {
          /* fork and save PID as pid[i] */
          pid[i] = fork();

          if ( pid[i] == 0 )
          {
                  /* Code Executed by Child Process */
                  /* start child process */
                  HelloWorld();
                  exit( EXIT_SUCCESS );
          }
          if ( pid[i] < 0 )
          {
                  /* Fork failed! */
                  exit( EXIT_FAILURE );
          }
  }
  /* Only Parent is here, due to exit() of the children */
  printf( "Parent (%d) is waiting for child processes.\n", getpid() );
  waitpid( (pid_t)-1, &status, 0 );
  return( EXIT_SUCCESS );
}
```

```
tdrohrer@ssm-mp17:~/ssm/gss_3/exercise2 $ ./hello_fork
Hello World from child process: 5996
Hello World from child process: 5997
Parent (5995) is waiting for child processes.
```

13. Wollen diese Prozesse nun miteinander kommunizieren, so wird Interprozess Kommunikation eingesetzt. Erstellen Sie eine Liste der Methoden zur Interprozesskommunikation:
    - *Pipes / Named Pipes*
    - *Sockets, Netzwerk*
    - *Message Queues*
    - *Shared Memory*
    - *Filesystem*

```
tdrohrer@ssm-mp17:~/ssm/gss_3/exercise2 $ ./with_synchronization
3 = SemidParent (6269) is waiting for child processes.
3 = Semid3 = SemidThe sum is 16236974
3 = SemidMin is 24
Max is 32757
```

Persönliche Lernkontrolle

Markieren Sie die richtigen Antworten:

- ☑ Compileroptionen haben einen wesentlichen Einfluss auf die Effizienz beim Ausführen eines Programms.
- ☐ Wenn ein System an seine Leistungsgrenzen stösst, kann in jedem Fall durch das Hinzufügen von weiteren CPUs eine schnellere Ausführungzeit eines Prozesses erreicht werden.
- ☐ Um eine CPU ein oder auszuschalten muss ein Sparc System neu gestartet werden.
- ☑ Mit „prstat –mL“ kann die Latenzzeit eines Threads dargestellt werden.

Nennen Sie (mindestens) vier Möglichkeiten um die Ausführungszeit eines Programms (zum Beispiel einer mathematischen Berechnung) zu verkürzen

- *Parallelisieren, Verteilung auf mehrere CPUs*
- *Zusätzliche Ressourcen Einbauen im Server (CPU, RAM, etc)*
- *Algorithmus verbessern / anderer verwenden*
- *Sauberer Programmieren, Ressourcen schonender Programmieren*
- *Programmcodeoptimierung durch den Compiler*
- *Optimale Verwendung der CPU Architektur (e.g. 64 bit)*

Nennen Sie die Posix Funktionen zum Erzeugen und Synchronisieren von Threads

fork() um ein neuer Prozess zu erzeugen
waitpid() um auf den Prozess zu warten

Erklären Sie, warum beide CPUs belastet werden, auch wenn nur ein Thread auf dem System eine hohe Last erzeugt? (Beispielsweise bei der Anzeige von prstat)

Infolge Task-Switch kann der Thread von einem auf den anderen CPU wechseln. Ebenfalls laufen noch diverse System-Threads die ebenfalls CPU load verursachen.

Erklären Sie wie Sie feststellen ob ein System optimal Ausgelastet ist. Wann ist ein System über 100% ausgelastet?

Mittels „vmstat“ kann die Auslastung abgefragt werden. Dadurch ist erkennbar bei einem Multi CPU System ob alle CPUs am Rechnen sind oder nur einzelne davon. Zusätzlich kann mit „prstat –mL“ angezeigt werden wie gross die Latenzzeit für die Prozesse sind.

Einen einfachen überblick ist z.B. mit „uptime“ möglich, dort wird die „load average“ (1min, 5min, 15min Durchschnitt) angezeigt.

with_synchronization.c

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
#include <time.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#define SHARED_FILE "daten.txt"
#define NUMSEMS 2

static int semid;


union semun {
         int val;
         struct semid_ds *buf;
         ushort_t *array;
};

key_t key;

int get_sem_val( int sid, int semnum )
{
        return( semctl(sid, semnum, GETVAL, 0));
}

int passSemaphore()
{
         struct sembuf b;

         b.sem_num = 0;
         b.sem_op = -1;     //P(), i.e. down()
         b.sem_flg = SEM_UNDO;

         if (semop( semid, &b, 1) == -1)
         {
                  printf("Semaphore DOWN() failed!");
                  return 0;
         }

         return 1;
}

int leaveSemaphore()
{
         struct sembuf b;

         b.sem_num = 0;
         b.sem_op =  1;        //V(), i.e. UP()
         b.sem_flg = SEM_UNDO;

         if (semop(semid, &b, 1) == -1)
         {
                  printf("Semaphore UP() failed!");
                  return 0;
         }

         return 1;
}



void RandomCounter( void )
{
         FILE *file;
         int number, counter;

         if (!passSemaphore())
         {
                  exit ( EXIT_FAILURE );
         }

         /* Open a file in write mode */
         file = fopen( SHARED_FILE, "w");

         if( file == NULL )
                  exit ( EXIT_FAILURE );

         srand48( time(NULL) );

         for ( counter = 0; counter < 1000; counter++ )
         {
                  /* create random number */
                  number = (int)(drand48() * 32768);
```

```
                /* write random number to shared file */
                fprintf(file, "%d\n", number );
        }

        fclose(file);
        leaveSemaphore();

        exit( EXIT_SUCCESS );
}

void AddNumbers( void )
{
        FILE *file;
        long sum;
        int number;

        if (!passSemaphore())
        {
                exit ( EXIT_FAILURE );
        }


        /* Open a file in read mode */
        file = fopen( SHARED_FILE, "r");

        if( file == NULL )
                exit ( EXIT_FAILURE );

        sum = 0;

        while ( fscanf( file, "%d", &number ) > 0 )
        {
                sum += number;
        }

        leaveSemaphore();

        printf( "The sum is %d\n", sum );
        exit( EXIT_SUCCESS );
}

void MinMaxNumber( void )
{
        FILE *file;
        int number, min, max;

        if (!passSemaphore())
        {
                exit ( EXIT_FAILURE );
        }

        /* Open a file in read mode */
        file = fopen( SHARED_FILE, "r");

        if( file == NULL )
                exit ( EXIT_FAILURE );

        min = 65535;
        max = 0;

        while ( fscanf( file, "%d", &number ) > 0 )
        {
                min = min > number ? number : min;
                max = max < number ? number : max;
        }

        leaveSemaphore();

        printf( "Min is %d\n", min );
        printf( "Max is %d\n", max );

        exit( EXIT_SUCCESS );
}


int main( int argc, char* argv[] )
{
        union semun arg;

        if ((key = ftok( argv[0], 'E' )) == -1)
        {
                perror( "ftok" );
                exit( EXIT_FAILURE );
        }

        // create a semaphore set with 1 semaphore:
        if ((semid = semget(key, 1, 0666 | IPC_CREAT )) == -1)
        {
                perror( "semget" );
                exit( EXIT_FAILURE );
        }
```

```
	/* initialize semaphore #0 to 1: */
	arg.val = 1;
	if ( semctl( semid, 0, SETVAL, arg ) == -1 )
	{
		perror( "semctl" );
		exit( EXIT_FAILURE );
	}

	printf("%d = Semid", semid);

	if ( fork() == 0 ) /* Fork and run child function */
		RandomCounter();

	if ( fork() == 0 ) /* Fork and run child function */
		AddNumbers();

	if ( fork() == 0 ) /* Fork and run child function */
		MinMaxNumber();

	/* Only Parent Process come here, due to exit() of the children */
	printf( "Parent (%d) is waiting for child processes.\n", getpid() );

	/* wait for 3 children */
	waitpid( (pid_t)-1, NULL, 0 );
	waitpid( (pid_t)-1, NULL, 0 );
	waitpid( (pid_t)-1, NULL, 0 );

	/* remove semaphore set from kernel */
	if ( semctl( semid, 0, IPC_RMID, arg ) == -1)
	{
		perror( "semctl" );
		exit( EXIT_FAILURE );
	}

	return( EXIT_SUCCESS );
}
```