CSC718 Parallel Programming Midterm

Total: 100 points

Due by Midnight, Oct 16, 2022

(Midterm is a take-home exam. It includes four questions. The last one is a programming assignment. Please submit all your solutions and source code through D2L midterm Dropbox.)

1) (20 points) Binary semaphores only allow the integer to hold the values 0 and 1. However, binary semaphore are easier to implement than general semaphores. Here we demonstrate that a general semaphore can be implemented from a binary semaphore.
   A structure of general semaphore is defined as below. It utilizes two binary semaphores. The binary semaphores support two functions.
   b_wait(s): decrements s if s>0. If not, the process executing is blocked.
   b_signal(s): increments s by 1 (does not change the value if s = 1). After the increment, a process waiting on s resumes its execution.

```
struct semaphore
{
        int value;
        int waiting = 0;
        b_semaphore queue = 0; // initialized to 0
        b_semaphore mutex = 1; // initialized to 1
}
```

```
void wait(struct semaphore *s)
{
        b_wait(s->mutex);
        if (s->value == 0)
        {
                s->waiting++;
                b_signal(s->mutex);
                b_wait(s->queue);
        }
        else
        {
                s->value--;
                A. _______________
        }
}
```

```
void signal(struct semaphore *s)
{
        b_wait(s->mutex);
        if (s->waiting > 0)
        {
                s->waiting--;
                B._______________
                C._______________
        }
        else
        {
                s->value++;
                b_signal(s->mutex);
        }
}
```

Figure 2

Two semaphore functions, wait(s) and signal(s) are given in Figure 2. However, there are three lines of code are missing. Fill the missing code in A, B, and C.

A. `b_signal(s->mutex);`

B. `b_signal(s->mutex);`
C. `b_signal(s->queue);`

2) (24 points) Assume n pieces of work forming into a one-dimensional array (0 to n-1), p processes, and block allocation (scatters larger blocks among processes) is used. Process rank i starts from 0.

   a. What is the most pieces of work any process has?

$$\left\lceil \frac{n}{p} \right\rceil$$

   b. What is the least pieces of work any process has?

$$\left\lfloor \frac{n}{p} \right\rfloor$$

   c. How many processes have the most pieces of work?

$$r = n \, mod \, p$$

   d. What is the first element controlled by process i?

$$\left\lfloor \frac{in}{\rho} \right\rfloor$$

   e. What is the last element controlled by process i?

$$\left\lfloor \frac{(i+1)n}{p} \right\rfloor - 1$$

   f. Which process will have the control on element j?

$$\left\lfloor \frac{(p(j+1)-1)}{n} \right\rfloor$$

3) (36 points) Programming Assignment: The value of the definite integral

$$\int_0^1 \frac{4}{1+x^2} dx$$

is π . We can use numerical integration to compute π by approximating the area under the curve. A simple way to do this is called the rectangle rule (Figure 4.7). We divide the interval [0, 1] into *k* subintervals of equal size. We find the height of the curve at the midpoint of each of these subintervals. With these heights we can construct k rectangles. The area of the rectangles approximates the area under the curve. As *k* increases, the accuracy of the estimate also increases.

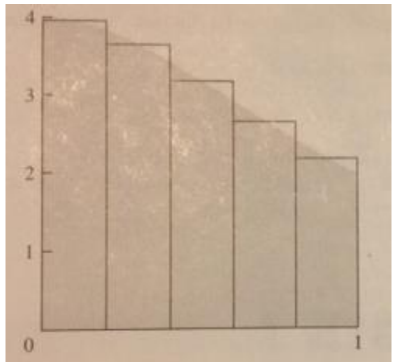

Figure 4.7 The rectangle rule is a simple way to approximate the area under a curve. In this example the function is $4/(1+x^2)$, and the area under the curve between 0 and 1 is π.

A C program that uses the rectangle rule to approximate π appears in Figure 4.8. The source code, pi.c, can be found in the class website. You can compile and run the program in Linux.

```
#include <stdio.h>
#define INTERVALS 1000000
int main(int argc, char* argv[])
{
  double area; /* The final answer */
  double ysum; /* Sum of rectangle heights */
  double xi;   /* Midpoint of interval */
  int i;
  ysum= 0.0;
  for(i=0;i<INTERVALS;i++)
    {
     xi= ((1.0/INTERVALS)*(i+0.5)); /* Midpoint of interval */
     ysum += 4.0/(1.0 + xi*xi);
    }
     area=ysum*(1.0/INTERVALS);
     printf("Area is %13.11f\n",area);
  return 0;
}
```

Figure 4.8 A C program to compute the value of π using the rectangle rule.

a. (10 points) Let INTERVALS=1000000, what is pi? Show your calculation result.

$$\int_0^1 \frac{4}{1+x^2}dx = 4\int \frac{1}{1+x^2}dx = 4\tan^{-1}x + C$$

- pi is 3.14159265359

```c
#include <stdio.h>
#define INTERVALS 1000000

int main(int arc, char* argv[])
{
    double area;    /* The final anser */
    double ysum;    /* Sum of rectangle heights */
    double xi;  /* Midpoint of interval */
    int i;

    ysum = 0.0;

    for (i=0; i < INTERVALS; i++)
    {
        xi=((1.0/INTERVALS)*(i+0.5));
        ysum+=4.0/(1.0+xi*xi);
    }

    area = ysum * (1.0/INTERVALS);
    printf( format: "pi is %13.11f\n", area);
    return 0;
}
```

main

Run: scratch

```
C:\cPrograms\CLionProjects\scratch\cmake-build-debug\scratch.exe
pi is 3.14159265359

Process finished with exit code 0
```

-

b. (16 points) The program can be converted to a multithread program. A sample program, mth-pi.c, has been provided as below. There are two lines, A and B, which are incomplete. Fill the blanks and complete the program.

ptharg[i].low = ______________A____________________;
pthargi[i].high = ______________B____________________;

A. BLOCK_LOW(i, THNUMS, INTERVALS)
B. BLOCK_HIGH(i, THNUMS, INTERVALS)

c. (10 points) Fill the missing lines in mth-pi.c. Compile and run the program. Let INERVAL=1000000, what is pi? Is it the same value as your calculation result in a).

- pi is 3.14159265359, it is the same value as my calculation result in a

```
for (i = 0; i < THNUMS; i++)
{
    ptharg[i].id = i;
    ptharg[i].low = BLOCK_LOW(i, THNUMS, INTERVALS);
    ptharg[i].high = BLOCK_HIGH(i, THNUMS, INTERVALS);
```

```
n
all ×
"/home/silence/Documents/dsu/2022b - fall/CSC 718 - Operating
Area is 3.14159265359

Process finished with exit code 0
```

-

```
#include <pthread.h>
#include <stdio.h>
#define INTERVALS 1000000

#define THNUMS		3

// thread parametes
struct ThreadParams
{
	int id;			// id
	int low;		// start
	int high;		// end
	double ysum;		// return partial sum
};

#define BLOCK_LOW(id, p, n)	((id)*(n)/(p))
#define BLOCK_HIGH(id, p, n)	(BLOCK_LOW((id)+1,p,n)-1)
#define BLOCK_SIZE(id, p, n)	(BLOCK_HIGH(id, p, n)-BLOCK_LOW(id, p, n)+1)

// calculate pi partial sum
void *calcpi(void *arg)
{
	double ysum;
	double xi;
	int i;
```

```
    struct ThreadParams *pm = (struct ThreadParams*)arg;

    ysum = 0.0;

    for (i = pm->low; i <= pm->high; i++)
    {
        xi=((1.0/INTERVALS)*(i+0.5));
        ysum=ysum+4.0/(1.0+xi*xi);
    }

    pm->ysum = ysum;
}

int main(int arc, char* argv[])
{
    struct ThreadParams ptharg[THNUMS];
    double area;
    double ysum;
    int i;
    pthread_t tid;
    void *status;

    // create multithreads to calculate partial sum
    ysum = 0.0;

    for (i = 0; i < THNUMS; i++)
    {
        ptharg[i].id = i;
        ptharg[i].low = ______________A__________________;
        ptharg[i].high = _____________B__________________;

        if (pthread_create(&tid, NULL, calcpi, (void*)&ptharg[i]) != 0)
        {
            perror("error creating child");
            return -1;
        }

        pthread_join(tid, &status);
    }

    // calculate total area
    area = 0;

    for (i = 0; i < THNUMS; i++)
```

```
        {
                area = area + ptharg[i].ysum;
        }

        area = area * (1.0/INTERVALS);
        printf("Area is %13.11f\n", area);
        return 0;
}
```

4) (20 points) Programming Assignment: Write a parallel program using MPI that computes the sum 1+2+...+p in the following manner: Each process i assigns the value i+1 to an integer, and then the processes perform a sum reduction of these values. Process 0 should print the result of the reduction. As a way of double checking the result, process 0 should also compute and print the value p(p+1)/2. Please benchmark the running time of the parallel program using the Rushmore cluster as discussed in the class.

	Np=1	Np=2	Np=3	Np=4
Execution time	0.000020	0.000165	0.000111	0.000281

```
mpiuser@Jefferson:~/Desktop/KC$ mpicc sum.c -o sum
mpiuser@Jefferson:~/Desktop/KC$ scp /home/mpiuser/Desktop/KC/sum mpiuser@Washington:/home/mp
iuser/Desktop/KC/sum
sum                                                          100%   17KB  13.2MB/s   00:00
mpiuser@Jefferson:~/Desktop/KC$ scp /home/mpiuser/Desktop/KC/sum mpiuser@Lincoln:/home/mpius
er/Desktop/KC/sum
sum                                                          100%   17KB  14.5MB/s   00:00
mpiuser@Jefferson:~/Desktop/KC$ scp /home/mpiuser/Desktop/KC/sum mpiuser@Roosevelt:/home/mpi
user/Desktop/KC/sum
sum                                                          100%   17KB  13.6MB/s   00:00
mpiuser@Jefferson:~/Desktop/KC$ mpirun -np 1 -machinefile machinefile.dsu ./sum
Lincoln [Processor 0 of 1] has completed.

The sum of all values between 1 and 10 is 55
Elapsed time: 0.000020.
mpiuser@Jefferson:~/Desktop/KC$ mpirun -np 2 -machinefile machinefile.dsu ./sum
Washington [Processor 1 of 2] has completed.
Lincoln [Processor 0 of 2] has completed.

The sum of all values between 1 and 10 is 55
Elapsed time: 0.000165.
mpiuser@Jefferson:~/Desktop/KC$ mpirun -np 3 -machinefile machinefile.dsu ./sum
Washington [Processor 1 of 3] has completed.
Roosevelt [Processor 2 of 3] has completed.
Lincoln [Processor 0 of 3] has completed.

The sum of all values between 1 and 10 is 55
Elapsed time: 0.000111.
mpiuser@Jefferson:~/Desktop/KC$ mpirun -np 4 -machinefile machinefile.dsu ./sum
Jefferson [Processor 3 of 4] has completed.
Washington [Processor 1 of 4] has completed.
Roosevelt [Processor 2 of 4] has completed.
Lincoln [Processor 0 of 4] has completed.

The sum of all values between 1 and 10 is 55
Elapsed time: 0.000281.
```