Operating Systems and Concurrency. Lab 4.

Daniyar Nazarbayev.

Part 1.

1. pipeline_template_1.c (gcc ./pipe_template_1.c -o pipeline -lpthread)

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

#define N_THREADS 3
#define BUFFER_SIZE 200
#define N_DATA 100000
#define WORKLOAD1 100000
#define WORKLOAD2 100000
#define WORKLOAD3 100000
//#define OUTPUT

/********************************************************************************
 **
 ** Here, the buffer implementation:
 **
 ********************************************************************************/

typedef struct buffer buffer_t;

struct buffer {
  volatile int head;
  volatile int tail;
  int size;
  volatile int *elems;
};

buffer_t *createBuffer( int size)
{
  buffer_t *buf;

  buf = (buffer_t *)malloc( sizeof(buffer_t));
  buf->head = 0;
  buf->tail = 0;
  buf->size = size+1;
  buf->elems = (int *)malloc( (size+1)*sizeof(int));

  return( buf);
}

int pop( buffer_t* buf, int *data)
{
  int res;

  if(buf->head == buf->tail) {
      res = 0;
  } else {
    *data = buf->elems[buf->head];
    buf->head = (buf->head+1) % buf->size;
    res = 1;
  }

  return( res);
}
```

```
int push( buffer_t* buf, int data)
{
  int nextTail;
  int res;

  nextTail = (buf->tail + 1) % buf->size;
  if(nextTail != buf->head)   {
    buf->elems[buf->tail] = data;
    buf->tail = nextTail;
    res = 1;
  } else {
    res = 0;
  }

  return( res);
}

/*******************************************************************************
 **
 ** Now, the thread functions for the pipelining:
 **
 ******************************************************************************/

typedef struct threadArgs threadArgs_t;

struct threadArgs {
  int tid;
  buffer_t *in_buf;
  buffer_t *out_buf;
  int workload;
};

int workUnit( int data)
{
  if( data < 0)
    data++;

  return( data);
}

int process( int tid, int data, int  workload)
{
  int i;

#ifdef OUTPUT
    printf( "[%d] processing item %d!\n", tid, data);
#endif

  for( i=0; i<workload; i++)
    data = workUnit( data);

#ifdef OUTPUT
    printf( "[%d] item %d done!\n", tid, data);
#endif

  return( data);
}

void * pipeline( void *arg)
{
  int data;
  int workload;
  int suc;
  buffer_t *in;
  buffer_t *out;
  int tid;

  in = ((threadArgs_t *)arg)->in_buf;
  out = ((threadArgs_t *)arg)->out_buf;
  tid = ((threadArgs_t *)arg)->tid;
  workload = ((threadArgs_t *)arg)->workload;

while(1){
      while(pop(in, &data) == 1){
            process(tid, data, workload);
            while(push(out, data)==0){}
      }
}

}

int main()
{

  int i, suc;
  int data;

  threadArgs_t args[N_THREADS];
  pthread_t threads[N_THREADS];
  buffer_t *in, *inter1, *inter2, *out;

  in = createBuffer( N_DATA+1);
  inter1 = createBuffer( BUFFER_SIZE);
  inter2 = createBuffer( BUFFER_SIZE);
  out = createBuffer( N_DATA+1);

printf("Starting threads\n");

args[0].tid = 1;
args[0].in_buf = in;
args[0].out_buf = inter1;
args[0].workload = WORKLOAD1;

args[1].tid = 2;
args[1].in_buf = inter1;
args[1].out_buf = inter2;
args[1].workload = WORKLOAD2;

args[2].tid = 3;
args[2].in_buf = inter2;
args[2].out_buf = out;
args[2].workload = WORKLOAD3;

int x = 0;
while(x<N_THREADS){
pthread_create(&threads[x], NULL, pipeline, (void*) &args[x]);
x++;
}

x = 0;
srand(time(NULL));
while(x < N_DATA){
      data = rand();
      if(push(in, data) == 1){
      printf("input buffer : data %d is %d\n", x+1, data);
      x++;
      }
}
```

```
x = 0;
while(x < N_DATA){
      if(pop(out, &data) == 1){
      printf("out buffer : data %d is %d\n", x+1, data);
      x++;
      }
}

  return(0);
}
```

Part 2.

1. pipeline_template_2.c

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

#define N_THREADS 3
#define BUFFER_SIZE 200
#define N_DATA 100000
#define WORKLOAD1 100000
#define WORKLOAD2 100000
#define WORKLOAD3 100000
#define OUTPUT

/*******************************************************************************
 **
 ** Here, the buffer implementation:
 **
 ******************************************************************************/

struct timespec start[N_DATA], stop[N_DATA];
#define BILLION 1E9

typedef struct buffer buffer_t;

struct buffer {
  volatile int head;
  volatile int tail;
  int size;
  volatile int *elems;
};

buffer_t *createBuffer( int size)
{
  buffer_t *buf;

  buf = (buffer_t *)malloc( sizeof(buffer_t));
  buf->head = 0;
  buf->tail = 0;
  buf->size = size+1;
  buf->elems = (int *)malloc( (size+1)*sizeof(int));

  return( buf);
}

int pop( buffer_t* buf, int *data)
{
  int res;

  if(buf->head == buf->tail) {
      res = 0;
  } else {
    *data = buf->elems[buf->head];
    buf->head = (buf->head+1) % buf->size;
    res = 1;
  }

  return( res);
```

```
}


int push( buffer_t* buf, int data)
{
  int nextTail;
  int res;

  nextTail = (buf->tail + 1) % buf->size;
  if(nextTail != buf->head)   {
    buf->elems[buf->tail] = data;
    buf->tail = nextTail;
    res = 1;
  } else {
    res = 0;
  }

  return( res);
}

/*******************************************************************************
 **
 ** Now, the thread functions for the pipelining:
 **
 ******************************************************************************/

typedef struct threadArgs threadArgs_t;

struct threadArgs {
  int tid;
  buffer_t *in_buf;
  buffer_t *out_buf;
  int workload;
};

int workUnit( int data)
{
  if( data < 0)
    data++;

  return( data);
}

int process( int tid, int data, int  workload)
{
  int i;

#ifdef OUTPUT
    printf( "[%d] processing item %d!\n", tid, data);
#endif

  for( i=0; i<workload; i++)
    data = workUnit( data);

#ifdef OUTPUT
    printf( "[%d] item %d done!\n", tid, data);
#endif

  return( data);
}


void* input(buffer_t * in){
      int data = 1;
      int x = 0;
      srand(time(NULL));
      while(x < N_DATA){
            data = rand();
            if(push(in, data) == 1){
            printf("input buffer : data %d is %d\n", x+1, data);
            x++;
            }

      }
}

void* output(buffer_t * out){
int data;
int x = 0;
      while(x < N_DATA){
            if(pop(out, &data) == 1){
            printf("out buffer : data %d is %d\n", x+1, data);
            x++;
            }
      }
}

void * pipeline( void *arg)
{
  int data;
  int workload;
  int suc;
  buffer_t *in;
  buffer_t *out;
  int tid;

  in = ((threadArgs_t *)arg)->in_buf;
  out = ((threadArgs_t *)arg)->out_buf;
  tid = ((threadArgs_t *)arg)->tid;
  workload = ((threadArgs_t *)arg)->workload;

int x = 0;
while(x < N_DATA){
      while(pop(in, &data) == 1){
            if(tid == 1){clock_gettime( CLOCK_REALTIME, &start[x]);}
            data = process(tid, data, workload);
            while(push(out, data)==0){}
            if(tid == 3){clock_gettime( CLOCK_REALTIME, &stop[x]);}
            x++;

      }
}

}




/********************************************************************************
 **
 ** main
 **
 *******************************************************************************/

//gcc ./pipe_template.c -o pipeline -lpthread
int main()
{
  int i, suc;
  int data;


  threadArgs_t args[N_THREADS];
  pthread_t threads[N_THREADS];
  buffer_t *in, *inter1, *inter2, *out;

  in = createBuffer( N_DATA+1);
  inter1 = createBuffer( BUFFER_SIZE);
  inter2 = createBuffer( BUFFER_SIZE);
  out = createBuffer( N_DATA+1);

  /**
   *
   * First, we start our threads:
   */

printf("Starting threads\n");

args[0].tid = 1;
args[0].in_buf = in;
args[0].out_buf = inter1;
args[0].workload = WORKLOAD1;

args[1].tid = 2;
args[1].in_buf = inter1;
args[1].out_buf = inter2;
args[1].workload = WORKLOAD2;

args[2].tid = 3;
args[2].in_buf = inter2;
args[2].out_buf = out;
args[2].workload = WORKLOAD3;

int x = 0;
while(x<N_THREADS){
pthread_create(&threads[x], NULL, pipeline, (void*) &args[x]);
x++;
}

printf("Filling first buffer\n");

pthread_t input_thread, output_thread;

pthread_create(&input_thread, NULL, input, in);

pthread_create(&output_thread, NULL, output, (void*) out);

pthread_join(input_thread, NULL);
pthread_join(output_thread, NULL);

  /**
   * Finally, we observe the output in the buffer "out":
   */

x=0;
double lat, avg_lat, min_lat, max_lat;
while(x <N_DATA){
    lat = ( stop[x].tv_sec - start[x].tv_sec )
          + (double) ( stop[x].tv_nsec - start[x].tv_nsec )
            / (double) BILLION;
if(x == 0){
min_lat = lat;
max_lat = lat;
}

      avg_lat =+ lat;
      if(lat < min_lat){ min_lat = lat;}
      if(lat > max_lat){ max_lat = lat;}

    printf( "%.9g\n", lat );
x++;
}

avg_lat = avg_lat / N_DATA;

// avg_lat is the average time it takes to process one item
// 1 divided by avg_lat should give the amount of times it can do in a second
double throughput = 1 / avg_lat;

    printf( "%.9g\n", min_lat );
    printf( "%.9g\n", max_lat );
    printf( "%.9g\n", avg_lat );
    printf( "%.9g\n", throughput );





  return(0);
}
```

Part 3.

1. pipeline_template_3.c

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

#include <stdbool.h>

#define N_THREADS 3
#define BUFFER_SIZE 200
#define N_DATA 100000
#define WORKLOAD1 100000
#define WORKLOAD2 100000
#define WORKLOAD3 100000
#define OUTPUT

/********************************************************************************
 **
 ** Here, the buffer implementation:
 **
 ********************************************************************************/

typedef struct buffer buffer_t;

struct buffer {
  volatile int head;
  volatile int tail;
  int size;
  volatile int *elems;
bool mutex;
};

buffer_t *createBuffer( int size)
{
  buffer_t *buf;

  buf = (buffer_t *)malloc( sizeof(buffer_t));
  buf->head = 0;
  buf->tail = 0;
  buf->size = size+1;
  buf->elems = (int *)malloc( (size+1)*sizeof(int));
buf->mutex = false;

  return( buf);
}

int pop( buffer_t* buf, int *data)
{
  int res;

  if(buf->head == buf->tail || buf->mutex == true) {
      res = 0;
  } else {
      buf->mutex = true;
    *data = buf->elems[buf->head];
    buf->head = (buf->head+1) % buf->size;
    res = 1;
      buf->mutex = false;
  }
  return( res);
}


int push( buffer_t* buf, int data)
{
  int nextTail;
  int res;

  nextTail = (buf->tail + 1) % buf->size;
  if(nextTail != buf->head && buf->mutex == false)   {
      buf->mutex = true;
    buf->elems[buf->tail] = data;
    buf->tail = nextTail;
    res = 1;
      buf->mutex = false;
  } else {
    res = 0;
  }

  return( res);
}

/*******************************************************************************
 **
 ** Now, the thread functions for the pipelining:
 **
 ******************************************************************************/

typedef struct threadArgs threadArgs_t;

struct threadArgs {
  int tid;
  buffer_t *in_buf;
  buffer_t *out_buf;
  int workload;
};

int workUnit( int data)
{
  if( data < 0)
    data++;

  return( data);
}

int process( int tid, int data, int  workload)
{
  int i;

#ifdef OUTPUT
    printf( "[%d] processing item %d!\n", tid, data);
#endif

  for( i=0; i<workload; i++)
    data = workUnit( data);

#ifdef OUTPUT
    printf( "[%d] item %d done!\n", tid, data);
#endif

  return( data);
}
```

```
void * pipeline( void *arg)
{
  int data;
  int workload;
  int suc;
  buffer_t *in;
  buffer_t *out;
  int tid;

  in = ((threadArgs_t *)arg)->in_buf;
  out = ((threadArgs_t *)arg)->out_buf;
  tid = ((threadArgs_t *)arg)->tid;
  workload = ((threadArgs_t *)arg)->workload;

while(1){
      while(pop(in, &data) == 1){
            data = process(tid, data, workload);
            while(push(out, data)==0){}
      }
}

}

/*******************************************************************************
 **
 ** main
 **
 ******************************************************************************/

int main()
{

  int i, suc;
  int data;

  threadArgs_t args[N_THREADS];
  pthread_t threads[N_THREADS];
  buffer_t *in, *inter1, *inter2, *out;

  in = createBuffer( N_DATA+1);
  inter1 = createBuffer( BUFFER_SIZE);
  inter2 = createBuffer( BUFFER_SIZE);
  out = createBuffer( N_DATA+1);

printf("Starting threads\n");

args[0].tid = 1;
args[0].in_buf = in;
args[0].out_buf = inter1;
args[0].workload = WORKLOAD1;

args[1].tid = 2;
args[1].in_buf = inter1;
args[1].out_buf = inter2;
args[1].workload = WORKLOAD2;

args[2].tid = 3;
args[2].in_buf = inter2;
args[2].out_buf = out;
args[2].workload = WORKLOAD3;

int x = 0;
while(x<N_THREADS){
pthread_create(&threads[x], NULL, pipeline, (void*) &args[x]);
x++;
}

x = 0;
data = 1;
srand(time(NULL));
while(x < N_DATA){
      data = rand();
      if(push(in, data) == 1){
      printf("input buffer : data %d is %d\n", x+1, data);
      x++;
      }
}

x = 0;
while(x < N_DATA){
      if(pop(out, &data) == 1){
      printf("out buffer : data %d is %d\n", x+1, data);
      x++;
      }
}

  return(0);
}
```

Part 4.

1. pipeline_template_4.c

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

#include <stdbool.h>

#define N_THREADS 3
#define BUFFER_SIZE 200
#define N_DATA 100000
#define WORKLOAD1 100000
#define WORKLOAD2 100000
#define WORKLOAD3 100000
#define OUTPUT

/*******************************************************************************
 **
 ** Here, the buffer implementation:
 **
 ******************************************************************************/

struct timespec start[N_DATA], stop[N_DATA];
#define BILLION 1E9

typedef struct buffer buffer_t;

struct buffer {
  volatile int head;
  volatile int tail;
  int size;
  volatile int *elems;
bool mutex;
};

buffer_t *createBuffer( int size)
{
  buffer_t *buf;

  buf = (buffer_t *)malloc( sizeof(buffer_t));
  buf->head = 0;
  buf->tail = 0;
  buf->size = size+1;
  buf->elems = (int *)malloc( (size+1)*sizeof(int));
buf->mutex = false;

  return( buf);
}

int pop( buffer_t* buf, int *data)
{
  int res;
      // || buf->mutex == true
  if(buf->head == buf->tail) {
      res = 0;
  } else {
      buf->mutex = true;
    *data = buf->elems[buf->head];
    buf->head = (buf->head+1) % buf->size;
    res = 1;
      buf->mutex = false;
  }

  return( res);
}


int push( buffer_t* buf, int data)
{
  int nextTail;
  int res;

  nextTail = (buf->tail + 1) % buf->size;
  if(nextTail != buf->head )   {
    buf->elems[buf->tail] = data;
    buf->tail = nextTail;
    res = 1;
  } else {
    res = 0;
  }

  return( res);
}


/********************************************************************************
 **
 ** Now, the thread functions for the pipelining:
 **
 *******************************************************************************/

typedef struct threadArgs threadArgs_t;

struct threadArgs {
  int tid;
  buffer_t *in_buf;
  buffer_t *out_buf;
  int workload;
};

int workUnit( int data)
{
  if( data < 0)
    data++;

  return( data);
}

int process( int tid, int data, int  workload)
{
  int i;

#ifdef OUTPUT
    printf( "[%d] processing item %d!\n", tid, data);
#endif

  for( i=0; i<workload; i++)
    data = workUnit( data);

#ifdef OUTPUT
    printf( "[%d] item %d done!\n", tid, data);
#endif
```

```
  return( data);
}

void * pipeline( void *arg)
{
  int data;
  int workload;
  int suc;
  buffer_t *in;
  buffer_t *out;
  int tid;

  in = ((threadArgs_t *)arg)->in_buf;
  out = ((threadArgs_t *)arg)->out_buf;
  tid = ((threadArgs_t *)arg)->tid;

int x = 0;
while(1){
      while(pop(in, &data) == 1){
            data = process(tid, data, WORKLOAD1);
            data = process(tid, data, WORKLOAD2);
            data = process(tid, data, WORKLOAD3);
            while(push(out, data)==0){}
      }
}

}




/*******************************************************************************
 **
 ** main
 **
 ******************************************************************************/

//gcc ./pipe_template.c -o pipeline -lpthread
int main()
{

  int i, suc;
  int data;


  threadArgs_t args[N_THREADS];
  pthread_t threads[N_THREADS];
  buffer_t *in, *out;

  in = createBuffer( N_DATA+1);
  out = createBuffer( N_DATA+1);

printf("Starting threads\n");

int x = 0;
while(x<N_THREADS){
args[x].tid = x+1;
args[x].in_buf = in;
args[x].out_buf = out;

pthread_create(&threads[x], NULL, pipeline, (void*) &args[x]);
x++;
}
```

```
printf("Filling first buffer\n");

x = 0;
srand(time(NULL));
while(x < N_DATA){
      data = rand();
      if(push(in, data) == 1){
      printf("input buffer : data %d is %d\n", x+1, data);
      x++;
      }
}

x = 0;
while(x < N_DATA){
      if(pop(out, &data) == 1){
      printf("out buffer : data %d is %d\n", x+1, data);
      x++;
      }
}

  return(0);
}
```