Simulation of Scheduling

CSC311 Systems Software

30 November 2015

queue.c

```
// Leon Tabak
// CSC311 Systems Software
// 30 November 2015

#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

// This is that start of a program to simulate the
// first-come/first-served scheduling of processes
// for uninterrupted execution in a CPU
// (or customers in a bank who line up in
// front of a teller's window).

// There are two principle parameters in this
// simulation: MEAN_SERVICE_TIME and MEAN_INTERARRIVAL_TIME.
// The relationship between these two parameters determines
// the performance of the system. If processes (customers)
// arrive faster than they can be served, the length of
// the queue (waiting line) will grow and grow.

// MEAN_SERVICE_TIME is measure of the
// amount of time needed to execute each process
// (or take care of each customer).
#define MEAN_SERVICE_TIME   2.0

// MEAN_INTERARRIVAL_TIME is a measure of
// the time that elapses between the arrival
// in the system of successive processes
// (or customers).
```

```
#define MEAN_INTERARRIVAL_TIME 3.0

// Create aliases for the 3 data structures
// that this program defines and uses.
// Also, create aliases for pointers to these
// data structures.
typedef struct process Process, *ProcessPointer;
typedef struct node Node, *NodePointer;
typedef struct queue Queue, *QueuePointer;

// A process is the basic unit of work in
// the system.
// It is a program to be executed or a customer
// in a bank who requires help from a teller.
struct process {
  // id is a unique integer identifier for the process
  int id;

  // serviceTime is a measure of the time required
  // from the CPU for this process if the process
  // is a program (or from the teller if the process
  // is a customer in a bank)
  double serviceTime;

  // interarrivalTime is a measure of the time that
  // elapses between the arrival of the previous
  // process and the arrival of this process
  double interarrivalTime;

  // arrivalTime is the time at which this process
  // enters the system--it is the sum of the interarrival
  // times of this process and all previous processes
  double arrivalTime;

  // serviceStartTime is the time at which this
  // process begins execution in the CPU (or receiving
  // service from the teller if the process is a customer
  // in a bank)
  double serviceStartTime;

  // serviceCompleteTime is the time at which the
  // execution of this process ends (or the time
  // at which the teller finishes whatever tasks
  // the customer has requested in the case that
  // the process is a customer in a bank)
  double serviceCompleteTime;
```

```
}; // process

// We can represent a queue with a doubly-linked
// list.
// A node is one element in the linked list.
// It contains a means of finding information
// about a single process and the means of finding
// what lies immediately ahead and immediately behin
// in the queue.
struct node {
  ProcessPointer processPointer;
  NodePointer pointerToPrevNode;
  NodePointer pointerToNextNode;
}; // node

// A queue is a waiting line.
// Processes (or customers) join the
// waiting line at one end and exit
// at the other end.
struct queue {
  int length;
  NodePointer pointerToHead;
  NodePointer pointerToTail;
}; // queue

void seedRandomNumberGenerator() {
  // Seed the random number generator
  // with the time measured in seconds.
  // "time_t" is a just another name for
  // a long (64 bit) integer.
  time_t t = time(NULL) ;
  srand( t ) ;
} // seedRandomNumberGenerator()

// Service times and interarrival times
// will be random numbers drawn from an
// exponential distribution.
// All values will be positive.
// Smaller values will be more likely than
// larger values.
// There is no upper bound on the values.
double exponentialRandom( double mean ) {
  return -mean * log(((double) rand())/RAND_MAX);
} // exponentialRandom()

int numberOfProcessesCreated = 0;
```

```
ProcessPointer createProcess() {
  ProcessPointer pp = (ProcessPointer) malloc(sizeof(Process));
  pp->id = numberOfProcessesCreated++;
  pp->serviceTime = exponentialRandom( MEAN_SERVICE_TIME );
  pp->interarrivalTime = exponentialRandom( MEAN_INTERARRIVAL_TIME );

  // At the time of the process' creation,
  // the values of the arrivalTime, serviceStartTime,
  // and serviceCompleteTime are unknown.
  pp->arrivalTime = 0.0;
  pp->serviceStartTime = 0.0;
  pp->serviceCompleteTime = 0.0;
  return pp;
} // createProcess()

void printProcess( ProcessPointer pp ) {
  printf( "process #%3d: (%8.4f, %8.4f, %8.4f, %8.4f, %8.4f)\n",
          pp->id,
          pp->serviceTime,
          pp->interarrivalTime,
          pp->arrivalTime,
          pp->serviceStartTime,
          pp->serviceCompleteTime );
} // printProcess( ProcessPointer )

// Print the id numbers of the...
//   * process that is referenced in the node
//     that is behind the given node
//   * process that is referenced in the given node
//   * process that is referenced in the node
//     that is ahead of a the given node
void printNode( NodePointer np ) {
  int previous = -1;
  int current = -1;
  int next = -1;

  if( np != NULL )  {
    current = np->processPointer->id;
    if( np->pointerToPrevNode != NULL ) {
      previous = np->pointerToPrevNode->processPointer->id;
    } // if
    if( np->pointerToNextNode != NULL ) {
      next = np->pointerToNextNode->processPointer->id;
    } // if
  } // if
```

```
  printf( "(%3d:%3d:%3d)", previous, current, next );
} // printNode( NodePointer )

QueuePointer createQueue() {
  // Create an empty queue.
  // The number of elements in the
  // empty queue is 0.
  // The pointers to its (non-existant)
  // head and tail are NULL.
  QueuePointer qp = (QueuePointer) malloc(sizeof(Queue));
  qp->length = 0;
  qp->pointerToHead = NULL;
  qp->pointerToTail = NULL;
  return qp;
} // createQueue()

void printQueue( QueuePointer qp ) {
  printf( "[%3d  ", qp->length );
  NodePointer np = qp->pointerToTail;
  while( np != NULL ) {
    printNode( np );
    np = np->pointerToNextNode;
  } // while
  printf( "]\n" );
} // printQueue( QueuePointer )

// Print a complete description of the
// process referenced in each element (node)
// of the queue.
// The complete description includes id, service time,
// interarrival time, arrival time, time at which
// service begins, and time at which service is completed.
void printProcessesInQueue( QueuePointer qp ) {
  NodePointer np = qp->pointerToHead;
  while( np != NULL ) {
    printProcess( np->processPointer );
    np = np->pointerToPrevNode;
  } // while
} // printProcessInQueue( QueuePointer )

bool isQueueEmpty( QueuePointer qp ) {
  return (qp->pointerToHead == NULL) &&
    (qp->pointerToTail == NULL);
} // isQueueEmpty( QueuePointer )
```

```
// Take a look at the process that is at
// the head of the line.
ProcessPointer peek( QueuePointer qp ) {
  ProcessPointer pp = NULL;

  if( qp->pointerToHead != NULL ) {
    pp = qp->pointerToHead->processPointer;
  } // if

  return pp;
} // peek( QueuePointer )

// Add a process at the end of the line.
void enqueue( QueuePointer qp, ProcessPointer pp ) {
  NodePointer np = (NodePointer) malloc(sizeof(Node));
  np->processPointer = pp;

  if( qp->pointerToTail != NULL ) {
    qp->pointerToTail->pointerToPrevNode = np;
  } // if

  np->pointerToNextNode = qp->pointerToTail;
  np->pointerToPrevNode = NULL;

  qp->pointerToTail = np;
  if( qp->pointerToHead == NULL ) {
    qp->pointerToHead = np;
  } // if

  // increment count of number of elements in queue
  qp->length++;
} // enqueue( QueuePointer, ProcessPointer )

// Remove a process from the front of the line.
ProcessPointer dequeue( QueuePointer qp ) {
  ProcessPointer pp = NULL;
  if( qp->pointerToHead != NULL ) {
    pp = qp->pointerToHead->processPointer;
    qp->pointerToHead = qp->pointerToHead->pointerToPrevNode;
    if( qp->pointerToHead == NULL ) {
      qp->pointerToTail = NULL;
    } // if
    else {
      free( qp->pointerToHead->pointerToNextNode );
      qp->pointerToHead->pointerToNextNode = NULL;
    } // else
```

```
    // decrement count of number of elements in queue
    qp->length--;
  } // if

  return pp;
} // dequeue( QueuePointer )

// Verify that the elements of the doubly-linked
// list are correctly linked.
void testQueue( int numberOfProcesses ) {
  seedRandomNumberGenerator();

  QueuePointer qp = createQueue();

  printf( "\n\nBegin adding elements to the queue.\n\n" );
  printQueue( qp );

  double elapsedTime = 0.0;
  int i;
  for( i = 0; i < numberOfProcesses; i++ ) {
    ProcessPointer pp = createProcess();
    elapsedTime += pp->interarrivalTime;
    pp->arrivalTime = elapsedTime;
    enqueue( qp, pp );
    printQueue( qp );
  } // for

  printf( "\n" );
  printProcessesInQueue( qp );

  printf( "\nBegin removing elements from the queue.\n\n" );
  printQueue( qp );

  while( !isQueueEmpty( qp ) ) {
    ProcessPointer pp = dequeue( qp );
    printQueue( qp );
    free( pp );
  } // while
} // testQueue( int )

// Create a queue and fill it with a specified
// number of processes.
QueuePointer buildQueue( int numberOfProcesses ) {
  seedRandomNumberGenerator();
```

```
  QueuePointer qp = createQueue();

  double elapsedTime = 0.0;
  int i;
  for( i = 0; i < numberOfProcesses; i++ ) {
    ProcessPointer pp = createProcess();
    elapsedTime += pp->interarrivalTime;
    pp->arrivalTime = elapsedTime;
    enqueue( qp, pp );
  } // for

  return qp;
} // buildQueue( int )

int main( int argc, char** argv ) {

  testQueue( 6 );

  exit(0);
} //  main( int, char** )
```