Channels, Visualization, and Topology Editor*

Steve Carr, Ping Chen, Timothy R. Jozwowski, Jean Mayo and Ching-Kuang Shene†
Department of Computer Science
Michigan Technological University
Houghton, MI 49931–1295
Email: {carr,pichen,trjozwow,jmayo,shene}@mtu.edu

ABSTRACT

This paper presents our effort in designing pedagogical tools for teaching message passing using channels. These tools include a class library that supports channels, a visualization system that helps students see the execution behavior of threads and message passing, and a topology editor that provides an environment for students to design network topologies. Moreover, since we have made sure the uniformity of the channel definition across the thread, parallel and distributed environments, porting a threaded program to a parallel/distributed environment is easy.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; D.4.1 [**Operating Systems**]: Process Management; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer Science Education*

General Terms

Design

Keywords

threads, channels, visualization

1. INTRODUCTION

Message passing was first used in Brinch Hansen's RC 4000 operating system in 1969 [7], and formalized by Hoare into the concept of a channel in 1978 [9]. Since then, message passing has become an indispensable tool in interprocess communication and operating system design. Many operating systems support message passing in various forms, and some languages (*e.g.*, Java) include message passing as a language feature. In operating systems textbooks, message passing is a standard topic in the discussion of synchronization primitives. This paper discusses our effort in teaching message passing with channels in a junior level introduction to operating systems course using two tools ThreadMentor and mtuTopology. ThreadMentor is designed to help students visualize the execution behavior of threads and synchronization primitives (*e.g.*, mutex locks, semaphores, monitors, barriers, reader-writer locks, and channels), while mtuTopology provides students with an environment to design connection topologies (intra-nets) and then connect a number of topologies into a network (inter-net). Because of the uniformity of the channel definition in our class libraries, a message passing program using channels can be ported to the thread, parallel and distributed environments with minimal effort. mtuTopology is independent of the underlying environment and can be used in concurrent programming and related courses.

*This work was supported by the National Science Foundation under grants DUE-9752244 and DUE-9952509. The fourth author is also supported by an NSF CAREER grant CCR-9984682.
†Corresponding author.

ITiCSE'02, June 24-26, 2002, Aarhus, Denmark.

2. WHAT IS A CHANNEL?

ThreadMentor employs a more general approach to channels than Hoare's original definition. A channel is a *bi-directional* communication link for threads to send messages to or receive messages from another thread. The *capacity* of a channel is the buffer size. If the capacity is zero, no message can be waiting in a channel. Therefore, a sender must wait until a receiver retrieves the message, and the two involved are synchronized for a message transfer to occur. Channels with zero capacity are usually referred to as *synchronous* channels (*i.e.*, blocking send and blocking receive), and the synchronization is a *rendezvous*. A telephone line is a good example as the connection must be established before any conversation can take place.

If the capacity is non-zero, messages may wait in the buffer. A sender (*resp.*, receiver) waits if the buffer is full (*resp.*, empty), and the channel is *asynchronous*. In a thread environment, an asynchronous channel is simply a bounded-buffer. A telephone with an answering machine is an example. As long as the recording tape has space, a caller (*i.e.*, sender) can call and leave a message. ThreadMentor does not support unbounded capacity channels.

3. THREADMENTOR SUPPORT

ThreadMentor consists of a class library [8] that supports thread management and all commonly used synchronization primitives, a user-level kernel mtuThread that supports non-

preemptive multithreaded programming [3], and a visualization system [4] that helps students see the execution behavior of threads and synchronization primitives (Figure 1). The visualization system runs as a separate process to minimize possible interferences to the running threads. The class library and the visualization system communicate with each other with a message queue.

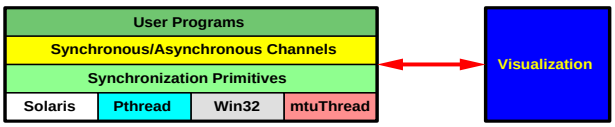


Figure 1: ThreadMentor Architecture

ThreadMentor supports synchronous and asynchronous channels. Class `SynOneToOneChannel` (*resp.*, `AsynOneToOneChannel`) defines a one-to-one synchronous (*resp.*, asynchronous) channel, and class `ManyToManyChannel` defines a many-to-many channel. A many-to-many channel is always asynchronous, and multiple senders and receivers can send messages to and receive messages from the same channel. Senders and receivers use `Send()` and `Receive()` methods to send and receive messages, respectively.

4. EXAMPLES AND ASSIGNMENTS

The concept of channels is covered at the end of the discussion of synchronization primitives in an operating systems course. A number of examples (*e.g.*, exchange sort and sorting network) are discussed in class. One of our favorites is the sieve method. Each prime number found so far is represented by a thread `SieveThread` that is an instance of the `Thread` class in which method `ThreadFunc()` contains the executable code (Figure 2). These `SieveThread`s are created dynamically and chained to the end of a linear array. Each thread creates a synchronous channel in its constructor, and keeps retrieving a number from this channel. If the received number is the `STOP` message, this thread exits the loop. If the received number is a multiple of the number this thread has, the received number is ignored and this thread retrieves the next number from the channel. There are two cases to consider if the received number is not a multiple of the number this thread has. If this thread is not the last in the thread chain, this thread passes the received number to the next one. Otherwise, this thread creates a new thread, appends it to the end, and sends the number to this newly created thread. The new thread "memorizes" the first number it receives and starts the loop.

There are two programming assignments for students to practice asynchronous and synchronous channels. The workcrew model was used for the asynchronous communication assignment. A master thread assigns work to workcrew threads through a channel. Workcrew threads retrieve the assigned work from this channel and report the results to the master through a second channel. This process continues until the master thread has gathered all necessary information. The partition step of the quicksort fits into this scheme very well.

The systolic matrix multiplication was used as a synchronous communication assignment. Recently, we follow Brinch Hansen's RC 4000 and Mach's approach of using message passing for operating system design. A synchronous channel is built between a simulated operating system component (*e.g.*, disk head scheduling or page replacement algorithm) and user threads. A user thread uses this channel to request system service. We used disk head scheduling as the simulated system service [1, 2]. Threads send track numbers to the channel as I/O requests. The disk head scheduler monitors the channel and retrieves messages with polling (Figure 3). Then, it serves the requests and releases the requesting thread when the simulated operation completes. A detailed description of these assignments can be found in [11].

```
SieveThread::SieveThread(int prime) // constructor
{
  next = NULL;              // next sieve thread
  this->prime = prime;      // save the prime
  channel = new SynOneToOneChannel(/* name here */);
  cout << " prime number: " << prime << endl;
}

void SieveThread::ThreadFunc() // thread body
{
  Thread::ThreadFunc();
  int             number;
  Thread_t self = GetID();

  while (true) {          // get a message
    channel->Receive(&number, sizeof(int));
    if (number == STOP)   // is it a STOP?
      break;              // yes, bail out
    if (number % prime != 0) {  // a composite?
      if (next != NULL)   // have a successor?
        next->channel->Send(&number,sizeof(int));
      else {              // no succ. create one
        next = new SieveThread(number);
        next->Begin();    // run my succ
      }
    }
  }
  if (next != NULL) {     // am I the last?
    next->channel->Send(&number,sizeof(int));
    next->Join();         // pass STOP and wait
  }
  Exit();
}
```

Figure 2: A sieve thread

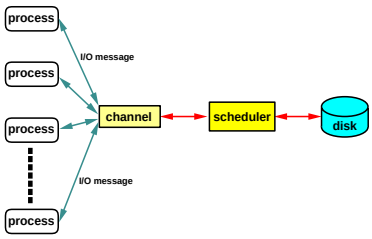


Figure 3: Disk head scheduling

5. VISUALIZING CHANNELS

ThreadMentor includes a visualization subsystem for helping students visualize the execution behavior of every involved thread and synchronization primitive (Figure 1). See [4] for the details. This subsystem is extended to include channel visualization.

When a user program runs, it automatically brings up the visualization system, which displays its Main Window. With

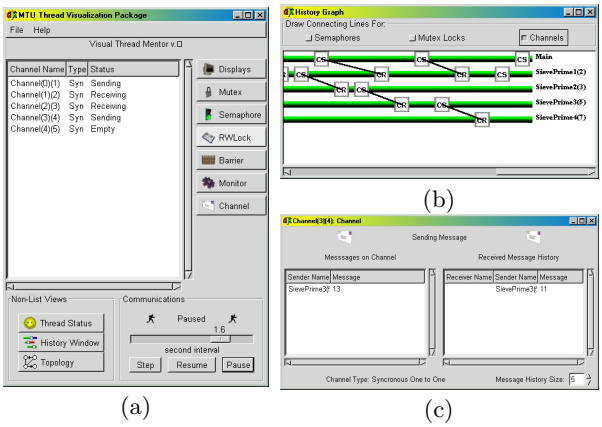


Figure 4: ThreadMentor Channel Visualization

the Main Window, a user can set the execution speed of a program, step through the synchronization primitives in a program, and click on one of the six buttons to display the top-level information of the corresponding synchronization primitive in a display area. Figure 4(a) shows the display of channel name, type (*i.e.*, synchronous or asynchronous), and the current status (*i.e.*, sending, receiving, and empty) of each channel created so far.

Click on the `History Window` button to activate the History Window. The execution history of the running program is shown as horizontal bars on which all events that have occurred are "tagged" with various tags. For example, tags `CS` and `CR` indicate a thread has sent and received a message (Figure 4(b)). A user may click on the `Channel` button of the History Window to see the sender and receiver of each communication, which is shown with a line segment connecting a `CS` tag of a thread and a `CR` tag of another thread. Moreover, a user may click on a tag to bring up a Source Window to see the source program with the line containing the corresponding send/receive highlighted. To learn more about a channel, a user may click on its name shown in the Main Window to bring up a channel window (Figure 4(c)). This window provides a history of all activities, including the actual messages, of a channel.

Students can trace their programs step-by-step, identify important elements of a channel, and visualize on-going message passing activities. Hence, this helps students recognize the dynamic behavior of threaded programs, understand the subject, and debug their programs.

6. A TOPOLOGY EDITOR

In parallel and distributed computing, the connections among nodes are frequently established before computation starts, instead of being dynamically created like the sieve program discussed in Section 4. This means a programmer may have to build the connection topology in her/his program. In our case (*i.e.*, simulating parallel/distributed computing with threads), a student must declare and create all channels. This is a very tedious but simple process. To help students build the connection topology so that they can concentrate on programming, we developed a topology editor mtuTopology.

mtuTopology is a stand-alone program of ThreadMentor. It provides the users with a two-level topology construction. A user may construct several topologies, which are treated as nodes in another topology, and use this capability to simulate an inter-network and a number of intra-networks. After a topology is constructed, a user may save the topology description to a set of `*.h` and `*.cpp` files to be included into her/his programs. Moreover, the description of the topology is also saved for the visualization to use. Because the definitions of channels are all the same in ThreadMentor and in the parallel and distributed modules that are being developed in our concurrent project [5], mtuTopology can be used for threads as well as for parallel and distributed programming.

7. CREATING A TOPOLOGY

When mtuTopology starts, it shows its Topology Net Editor Window, the rear one in Figure 5. With the buttons in the lower-left corner, a user can create a new, delete an existing, and edit a selected topology. Click on the `New Topology` button to add a topology to the canvas of the Topology Net Editor Window. Once this is done, a topology icon with its ID is shown on the canvas, and, at the same time, the default name and the ID are shown in the display area. To edit a topology, click on one of the topology icons followed by the `Edit Topology` button. See the middle window in Figure 5. The selected (*i.e.*, clicked) topology is highlighted. Moreover, this brings up the Topology Editor Window. See the front window in Figure 5.

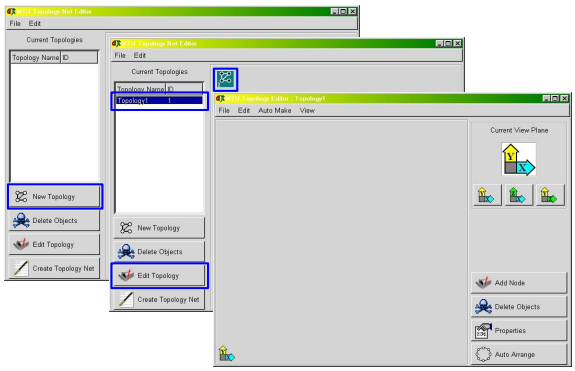
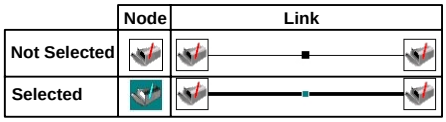


Figure 5: mtuTopology Main Windows

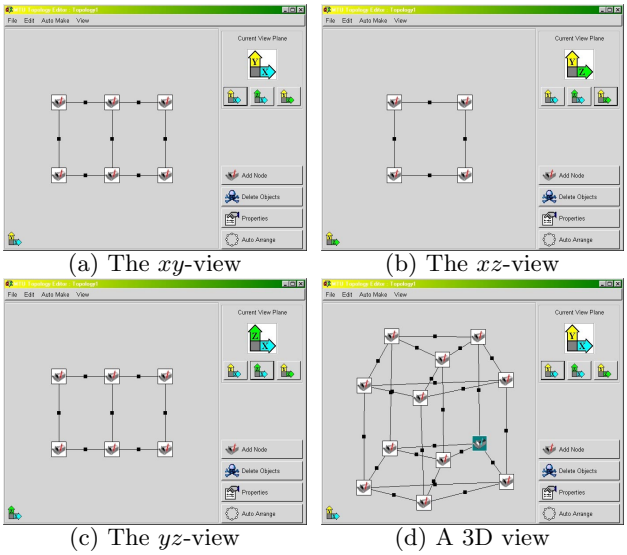
A connection topology is built with nodes and links, where nodes may be threads or processes and links are channels. A user may use the `Add Node` button to add new nodes, and `Delete Objects` to delete nodes and links. A node and a link are represented by a mailbox and a line segment between two mailboxes, respectively. To select a node (*resp.*, link), click on the node (*resp.*, the small black square on the link). The selected node and link are shown with inverted color and thick line segment, respectively, as shown below.



A user may right-drag a node to another to create a link, and left-drag a node to change its position. If some nodes and links are no longer needed, select one and click on `Delete Objects` to delete. More nodes and links can be

added until the topology meets the need.

mtuTopology is capable of generating linear array, ring, star, grid, torus, and fully connected topologies. Moreover, the grid and torus topologies can be in 3D. Click on the `Auto Make` menu button to bring up a menu for selecting a topology. Once a topology type is selected, a user supplies additional input such as the width, height, and depth of a 3D grid or 3D torus, or the number of nodes for the other types. Then, mtuTopology constructs a topology that meets the specification. Because grid and torus can be 2D or 3D, mtuTopology provides the xy-, xz- and yz- views in the upper-right corner of the Topology Editor Window. Click on one of the three buttons to switch views. Figure 6 is a 3D torus of width 3, height 2 and depth 2. A user may move the nodes to create a better view as shown in the Figure 6(d).



(a) The xy-view (b) The xz-view

(c) The yz-view (d) A 3D view

Figure 6: A 3D Torus Topology

Combined with the class library, mtuTopology can create a topology in an *indefinite* form for linear array, ring, star, grid and torus topologies. More precisely, the topology editor creates a topology of the indicated type, and allows the user program to specify the actual dimension. Figure 7 shows a linear array in which a node is marked with a special node icon, indicating an indefinite form.

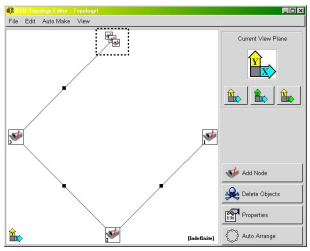


Figure 7: A Linear Array Topology in Indefinite Form

The indefinite form is very useful. For example, if the systolic matrix multiplication algorithm is used to compute the product $C = A \cdot B$, where A and B are $m \times k$ and $k \times n$ matrices, we need a $(m+1) \times (n+1)$ grid (Figure 8). The upper-left node may be used for regulating the heartbeat rhythm, and each of the remaining nodes of the first column (*resp.*, row) pumps the data of the corresponding row (*resp.*, column) of matrix A (*resp.*, B) to its neighbor. Each of the remaining nodes in the grid receives data from its left and top neighbors, performs some operations, and pumps the data it receives to its right and bottom neighbors. Since before the matrices are read in we do not know the number of rows and columns, the indefinite form is required.

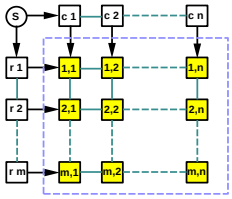


Figure 8: Grid for Systolic Matrix Multiplication

Figure 9 shows a simplified example of the use of mtuTopology. mtuTopology generates a grid topology for the systolic matrix multiplication algorithm as shown in Figure 8 and saves its definition in `Grid.h`. A user program simply includes this header file and creates the topology. Each thread receives its position in the grid when it is created. Then, threads use `TopologySend()` and `TopologyReceive()` to send and receive messages.

```
#include "Grid.h"
GridTopology  *Top;         // the grid topology
Top = new GridTopology();   // in main()

WorkerThread::WorkerThread(int r, int c)
{
  row = r; column = c;
}

void WorkerThread::ThreadFunc() // thread body
{
  Thread::ThreadFunc();
  int  r, c, sum = 0;
  int  left, right, up, down, ID = getMyID();

  left = ID-1; right = ID+1;
  up   = ID - GRID_SIZE; down = ID + GRID_SIZE;
  for (i = 0; i < MAX_SIZE; i++) {
    Top->TopologyReceive(left, &r, sizeof(int));
    Top->TopologyReceive(up, &c, sizeof(int));
    sum += r * c;
    Top->TopologySend(right, &r, sizeof(int));
    Top->TopologySend(down, &c, sizeof(int));
    // other computation
  }
  C[row][column] = sum;  // save result
  Exit();
}
```

Figure 9: Systolic matrix multiplication

8. CONNECTING TOPOLOGIES

Two or more topologies can be connected into a topology network. Thus, the topologies are like intra-networks, and the topology network is an inter-network. Similar to

the topology editor, we can right-drag a topology icon on top of the other to establish a link. In fact, the topology net editor permits multiple connections between topologies (Figure 10); however, a link must connect to two nodes in two topologies. Thus, for each link, we need to select a node from each topology to complete the connection. To do so, select the link by clicking on the black square of that link. A dialogue window appears in which the IDs of the two involved topologies are shown (the inset window of Figure 10). We then fill in the node ID in each topology.

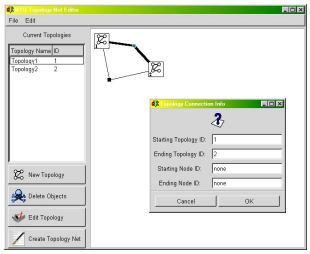


Figure 10: The Topology Net Editor

After the desired topology network is created, we can save it by clicking on the `Create Topology Net` button in the lower-left corner of the Topology Net Editor Window. This will generate a set of `*.h` and `*.cpp` files to be used with our class libraries. In addition, we can also load a topology back for further work. This process continues until a satisfactory topology net is obtained.

9. CONCLUSIONS

We have presented the channel concept, its class library, and two pedagogical tools ThreadMentor and mtuTopology for teaching concurrency based on message passing using threads. The concept of channels has been used in our operating systems course [10, 11], and ThreadMentor and mtuTopology are being site tested in a number of schools. Reactions from site testers and participants of our workshops [6] were very positive and encouraging. Based on channels, we are currently developing ConcurrentMentor for data-parallel programming and distributed programming [5]. The interested readers may find more about our work, software availability, course materials, and future announcements at the following site:

```
http://www.cs.mtu.edu/~shene/NSF-3
```

The URLs of the home pages of ThreadMentor and ConcurrentMentor are

```
http://www.cs.mtu.edu/ThreadMentor
http://www.cs.mtu.edu/ConcurrentMentor
```

A tutorial of ThreadMentor is available at

```
http://www.cs.mtu.edu/~shene/NSF-3/e-Book/index.html
```

10. REFERENCES

[1] Andrews, G. R., *Concurrent Programming: Principles and Practice*, Benjamin/Cummings, 1991.

[2] Andrews, G. R., *Foundations of Multithreaded, Parallel, and Distributed Programming*, Addison-Wesley, 2000.

[3] Bedy, M. J., Carr, S., Huang X. and Shene, C.-K., The Design and Construction of a User-Level Kernel for Teaching Multithreaded Programming, *29th ASEE/IEEE Frontiers in Education*, Vol. II, 1999, pp. (12b3-1)–(12b3-6).

[4] Bedy, M. J., Carr, S., Huang X. and Shene, C.-K., A Visualization System for Multithreaded Programming, *31st SIGCSE Technical Symposium*, 2000, pp. 1–5.

[5] Carr, S., Fang, C., Jozwowski, T., Mayo, J. and Shene, C.-K., A Communication Library to Support Concurrent Programming Courses, *ACM 33rd SIGCSE Technical Symposium*, 2002, pp. 360–364.

[6] Carr, S., Mayo, J. and Shene, C.-K., Teaching Multithreaded Programming Made Easy (workshop abstract), *The Journal of Computing in Small Colleges*, Vol. 17 (2001), No. 1 (October), pp. 156–157; and *ACM 33rd SIGCSE Technical Symposium*, 2002, p. 420.

[7] Brinch Hansen, P., RC 4000 Software: Multiprogramming System, in *Classic Operating Systems: from Batch Processing to Distributed Systems*, edited by Per Brinch Hansen, Springer-Verlag, 2001, pp. 237–281.

[8] Carr, S. and Shene, C.-K., A Portable Class Library for Teaching Multithreaded Programming, *ACM 5th ITiCSE 2000 Conference*, 2000, pp. 124–127.

[9] Hoare, C. A. R., *Communicating Sequential Processes*, Prentice-Hall, 1984.

[10] Shene, C.-K., Multithreaded Programming in an Introduction to Operating Systems Course, *29th SIGCSE Technical Symposium*, 1998, pp. 242–246.

[11] Shene, C.-K., Multithreaded Programming Can Strengthen an Operating Systems Course, to appear in *Computer Science Education Journal*.