μC++ Monitoring, Visualization and Debugging Annotated Reference Manual

Preliminary Draft

Version 1.0

Peter A. Buhr and Martin Karsten ©1993, 1995

November 15, 1996

Contents

Preface

The goal of this work is to provide a framework and a set of tools in that framework to help understand the dynamic execution of concurrent programs. The dynamic execution of concurrent programs is significantly more complex than in sequential programs; understanding that complexity is aided by monitoring, visualization and debugging tools. Furthermore, a concurrent program introduces new kinds of errors, such as race-conditions, livelock and deadlock, that do not occur in a sequential program. This work provides tools to monitor and visually display how much parallelism is actually occurring in a concurrent program, and other tools to determine how and why a concurrent program failed.

This manual is strictly a reference manual for monitoring, visualization and debugging in μC++. A reader should have an intermediate knowledge of X/Intrinsic/Motif windows and concurrency to understand all the ideas presented in this manual as well as some experience programming in μC++.

This manual contains annotations set off from the normal discussion in the following way:

□ Annotation discussion like this is quoted with quads. □

An annotation provides rationale for design decisions or additional implementation information. Also a chapter or section may end with a commentary section, which contains major discussion about design alternatives and/or implementation issues.

Each chapter of the manual does *not* begin with an insightful quotation. Feel free to add your own.

Chapter 1

Framework

1.1 Introduction

A framework is presented and a set of tools in that framework to help understand the dynamic execution of concurrent programs and debug them. A holistic view is taken to understanding a concurrent program by providing a number of different capabilities. Furthermore, by adopting a toolkit approach, both users and implementors of the concurrency system can use the same tools to build monitoring, visualization and debugging capabilities.

The toolkit is built around μC++ [BS95], which is an extended version of C++ providing light-tasking facilities using a shared-memory model. Most of the ideas presented here are language independent, and thus, are applicable to a large number of concurrent and non-concurrent languages. Finally, this work is based in UNIX[1], using X/Intrinsic/Motif because of their current popularity and availability.

1.2 Framework

The following three components form the framework:

Monitoring is the process of asynchronously (or synchronously) collecting information about a program's execution, which can be displayed during the program's execution or afterwards. When monitoring a sequential program, it is generally true that the monitoring does not change the data values generated or the order that the values are generated. However, the non-determinism and temporal aspects of concurrency make it impossible to monitor a concurrent program without affecting its execution. Hence, as soon as a probe is attached to monitor behaviour, the probe affects the system it is measuring so that the system may not behave as it did before the probe was attached, called the probe effect. At best, a designer of monitoring tools can only strive to minimize probe effects.

Both statistical and exact monitoring capabilities are needed. Statistical monitoring is done by periodically probing a target program's memory from a separate task. This form of monitoring provides imprecise information about a program, but has a lower probe effect than exact monitoring. Exact monitoring requires that the target program generate events whenever interesting operations occur. This form of monitoring provides precise information about the program, but has a much higher probe effect.

Monitoring may be on data or events, where data monitoring show a variable as it changes and event monitoring show when an event has occurred. Data monitoring can be considered as event monitoring,

[1]UNIX is a registered trademark of AT&T Bell Laboratories

where the event is assignment to the variable; however, it is the value that is important and not the fact that the assignment occurred.

Both statistical and exact monitoring can be done implicitly and explicitly, i.e., the monitoring is performed automatically by the underlying runtime system or the user explicitly is involved in specifying the monitoring.

Visualization is the process of displaying monitored information in a concise and meaningful way. The raw monitored data can be presented directly as a sequence of data values; however, there is usually too much data for people to interpret and comprehend. Instead, the raw data is transformed into another form that allows large amounts of data to become comprehendable. For example, displaying data in the form of a graph may allow large amounts of information to be understood at a glance. Just as some graphs can represent multiple dependent and possibly independent variables [Tuf83, p. 40], so can a visualization display represent multiple values and events. As more information is represented by a single display, the more complex the display becomes, and usually, the more difficult it is for a user to understand the display and comprehend its meaning. Colour is often an essential element of visualization because it provides yet another mechanism for condensing and differentiating information in a display.

Debugging is an interactive process where a program can be stopped, restarted and executed stepwise. The program's data can be looked up and modified when the program is stopped. When debugging a sequential program, this process is synchronous between the debugger and the target. However, for debugging a concurrent program, control and manipulation mechanisms have to be provided independently for every thread of control. Debugging affects the execution of a concurrent program in the same way monitoring does. This influence is partly determined by the requests of the programmer debugging the application. Hence, there is a similar motivation to minimize the influence that is caused by user requests.

It is true that monitoring and visualization can be used to help debug a program. However, both monitoring and visualization can be used for other purposes, e.g., to help optimize an algorithm or program, and therefore, are orthogonal issues to debugging.

1.3 Statistical Monitoring

In general, a program generates too many data values and too many events occur during execution to be of interest to or understood by a programmer. A programmer can often tell if a program is working according to its design simply by examining its general behaviour. For example, are approximately the right number of tasks created, blocked, or destroyed at approximately the right time? A program's general behaviour can be shown by statistically sampling its execution. For example, program profilers, like gprof [GKM82], provide information about execution "hot spots" by sampling the current execution location. Execution hot spots can then be examined as possible locations for optimization to improve program performance. Statistical information may be valuable for debugging a new algorithm or simply to understand the differences among algorithms (that are already debugged). Statistical sampling can often be done without having to insert code in an application so sampling can be done on existing execution modules. Furthermore, the probe effect may be lessened on multiprocessor shared-memory architectures as the sampling can occur on a separate processor (special sampling hardware can also be used). The drawback to statistical sampling is that important events may occur between sampling, and hence, be missed.

In a concurrent program, a statistical examination of tasks that form a program, can tell a programmer about the level of concurrency that is provided by a particular algorithm or by different data manipulated by an algorithm. Statistical feedback about the communication among tasks can also be extremely informative.

1.4 Exact Monitoring

Exact monitoring is sampling with the guarantee that all events of a particular kind are observed. Exact monitoring are often necessary to find concurrency errors like race conditions or event sequences that lead to deadlock. As a result, exact monitoring is often considered a debugging technique; however, it is useful for other purposes. For example, knowing an exact sequence of events can explain why one algorithm provides more concurrency than another. The drawback with exact monitoring is that enormous amounts of data may be generated. Handling this data invariably increases the probe effect, hence making it more difficult to observe just those events that are trying to be captured by collecting exact events (catch-22).

Because large amounts of information are generated, it is usually post-processed. Post-processing has advantages like multiple replay and reverse replay. However, because of the volume of information, facilities must exist to elide irrelevant information or search for particular kinds or patterns of event sequences.

1.5 Implicit/Explicit Monitoring and Visualization

In general, it does not seem possible to provide totally automated monitoring and visualization. That is, no visualization system can anticipate all the different ways that users will want to collect and display their information. Therefore, tools must be directly available to users so they have the freedom to experiment with different approaches. On the other hand, some predefined facilities can be provided so that casual users do not have to become experts to produce simple animation components. Therefore, a monitoring/visualization system is best designed as a toolkit, which provides capabilities at a variety of levels.

This toolkit allows users to participate in the design activity by providing their own tools. Users can easily build their own monitoring and displaying tools from existing tools. Furthermore, these same tools are available to the system implementors to build low-level monitoring and visualization tools for the runtime kernel.

1.6 Debugging/Debugger

The definition of debugging used here encompasses debugging by a traditional interactive debugger, such as dbx or gdb. This process involves an interactive session where the debugger is used to control the execution of an application program by stopping its execution, examining it, possibly changing it, and restarting its execution so the cycle can begin again. A *symbolic debugger* provides the additional capability to refer to data storage using variable names, and refer to code using file names and statement numbers from the original source program.

Unfortunately, most interactive debuggers do not work with concurrent programming languages or environments (some exceptions are pdbx on the Sequent and dbx on the SGI machines). With shared memory concurrency, the debugger must know about the multiple stacks that exist for the objects with independent execution states and deal with setting breakpoints in a single code image shared by these objects. Debugging each of these objects is done sequentially, but the debugger must be able to concurrently present sessions for multiple threads of control and provide mechanisms to handle them independently. With non-shared memory concurrency, the debugger must handle multiple independent address spaces, possibly on different machines, which often requires interaction with the operating system.

1.7 μC++ MVD Toolkit

The μC++ Monitoring, Visualization and Debugging (MVD) toolkit supports the following facilities:

- There is a general facility to construct statistical monitoring tools. These monitoring tools can be connected to a (currently small) set of Motif visualization tools to display the monitored results. The statistical monitoring and visualization can occur in real-time or monitored data can be written to a file for post-processing visualization. A user may explicitly indicate in a program the variables that are to be monitored and possibly how they are visualized. There is also an implicit visualization capability to monitor the μC++ runtime kernel.
- There is an exact monitoring facility that is connected to a powerful post-processing event-replay facility [Tal92]. All μC++ objects can have certain critical events traced by simply compiling the program with an appropriate flag.
- There is a symbolic debugger that understands multiple shared-memory execution states. It concurrently presents sessions for an arbitrary number of tasks in a μC++ program and allows independent control of each task. As well, it provides facilities to look up local data relative to any task's execution stack. Currently, it has limited support to control non-shared memory applications.

1.8 Organization

This manual discusses implicit statistical and implicit exact monitoring first to allow users to use the monitoring and visualization tools as quickly as possible. Only minimal effort is required to learn how to manipulate the displays that are implicitly generated. Next explicit statistical and explicit exact monitoring are covered. These concepts require substantially more understanding because of the additional user involvement. Finally, the debugger is discussed.

Chapter 2

Implicit Statistical Monitoring

By including file:

```
#include "/u/usystem/software/MVD/src/uKernelSplr.h"
```

after <uC++.h>, implicit monitoring and visualization of the μC++ kernel is presented. The runtime monitor is built using the same facilities detailed in Chapter 5. The only additional capability of the μC++ runtime sampler is that it is allowed to reference variables internal to the runtime system, which are normally hidden.

2.1 Kernel Display

The μC++ runtime system groups tasks and processors together into entities called *clusters* (see "μC++ Annotated Reference Manual"). When the application begins execution, a list of clusters in displayed (left window in Figure 2.1). On a uniprocessor there is only one cluster, called the uSystemCluster. The slider at the top of the window dynamically controls the sampling and display frequency of the information in the window. The polling and display frequencies are combined because there is no state that is saved on each poll; the cluster list simply traversed and displayed.

2.2 Cluster Display

By selecting a cluster from the list of clusters, detailed information about that cluster's execution is presented. For example, cluster uSystemCluster has been selected (indicated in reverse video in the left window) and its detailed information is shown in the right window in Figure 2.1. The lower portion of the right window shows the tasks currently executing on the cluster (left column) and which tasks are actually executing on processors (right column). The first five tasks are administrative tasks and tasks fred, mary, and john are user tasks. Task uWatcher is currently executing. (The example was run on a uniprocessor). The slider at the top of the window dynamically controls the sampling and display frequency of the information in the window. The polling and display frequencies are combined because there is no state that is saved on each poll; the cluster list simply traversed and displayed.

2.3 Task Display

By selecting a task from the list of clusters, detailed information about that task's execution is presented. For example, tasks uMain and uWatcher have been selected (indicated in reverse video in the left column) and their detailed information is shown in the right window in Figure 2.2. The sliders at the top of the window dynamically controls the sampling and display frequency of the information in the window. The polling and display frequencies are separate because there is state, stack high-water mark and task status information,

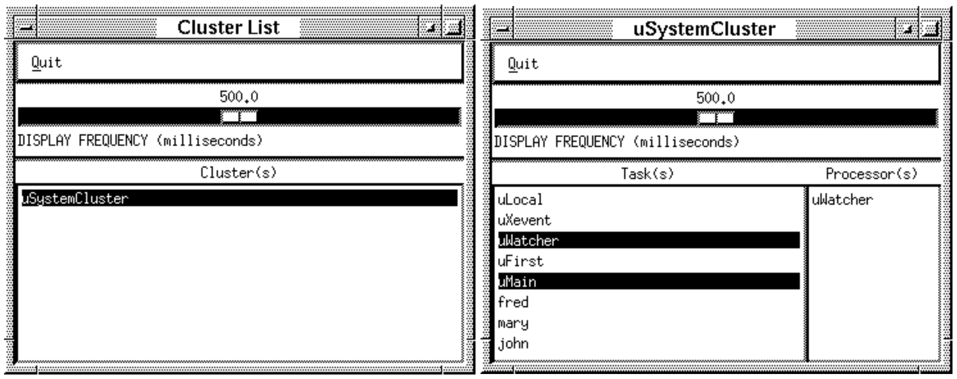


Figure 2.1: Cluster Displays

that is saved on each poll. By polling at a high frequency, more precise stack high-water mark and task status information is acquired.

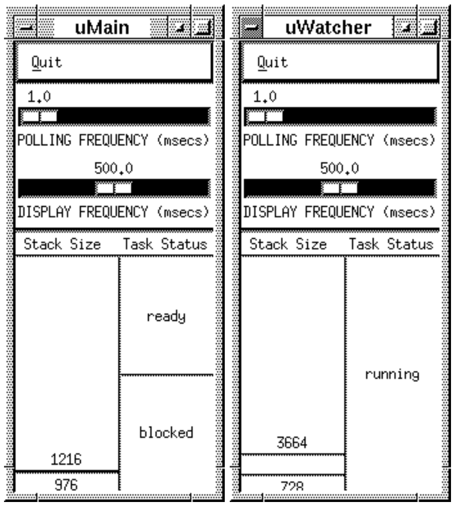


Figure 2.2: Task Displays

A task display shows the status of the task's stack and execution status; more information may be added. The stack display shows the size of the stack the last time the task performed a context switch (bottom number in the column). This value is displayed in a range from 0 to the task's maximum stack size. As well, there is a high-water mark indicator showing the maximum amount of stack space observed thus far (top number in the column). This information is useful for setting the amount of stack space for a task. If a task is using most of its stack, it could be increased. A task can exceed its stack limit between samples so just because a task's high-water mark has not exceeded the maximum do not assume the stack size is sufficient.

The status display shows the percentage of time a task spends in the running, ready and blocked states, respectively. This information indicates which tasks are performing most of the work and can be used to determine if a group of tasks are achieving expected levels of concurrency. Task uMain is spending approximately half of its time in the ready and blocked state. Task uWatcher is spending all of its time in the running state. On a uniprocessor, all tasks except uWatcher will have zero running state because the kernel watcher is always running when it invokes the task samplers so a task sampler cannot see its task running. On a multiprocessor, where the kernel watcher is on a separate processor, task displays will show all three states.

Chapter 3

Exact Monitoring

By compiling an *entire* program with the -trace compilation flag, i.e., all translation-units, events are generated for several fundamental μC++ operations. These events are implicitly generated from the user's perspective because no code has to be written to cause their generation. Events for a particular object form a *trace* of the operations performed by or to that object. These events are collected and possibly filtered by a collector task and then sent to the μC++ event tracer [Tal92] (see Figure 3.1).

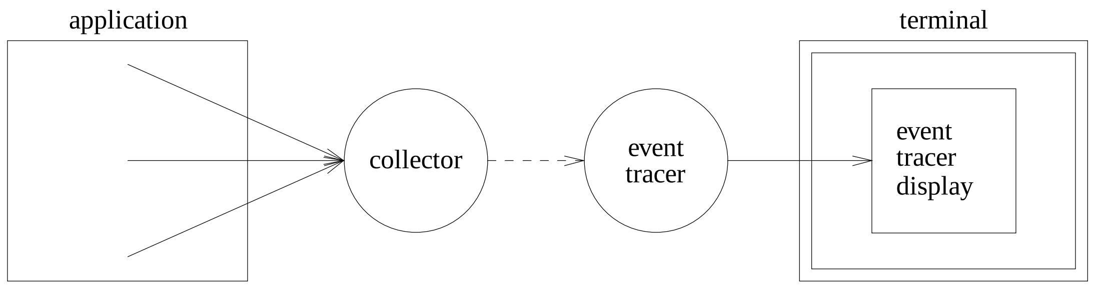


Figure 3.1: Event Tracer Structure

The event tracer may be on the same or a different computer; hence the link to the event trace may be through shared memory or over a communication channel. The event tracer organizes the events it receives and displays the event information in a graphical form, with traced objects on one axis and relative time along the other axis. No knowledge of the event tracer is required to understand this discussion. (See the document "Hermes Debugger Prototype–User Documentation" for a detailed explanation of all the features and options available for displaying event information.)

3.1 Model

Everything that happens in a μC++ program is done by the threads of tasks; the current tracing model focuses on the activities of the threads within a program. In particular, thread movement among certain objects are closely followed. For example, when one μC++ task calls a member routine in another, the callee is seen blocking to accept the call, and the caller is seen petitioning for entry, moving into and then out of the callee, after which the callee's thread is seen to continue execution.

Currently, the following objects are traced in μC++: tasks, monitors, semaphores, coroutines and mutex coroutines. Only certain operations on these objects generate events: for mutex objects, all interactions with mutex members are indicated in the trace; for coroutines, all interactions with the coroutine body are traced. Calls to non-mutex members and members of coroutines that do not resume the coroutine do not generate events nor do calls to free routines. Generating events for all calls would provide too much information in an object's trace, much of it unrelated to concurrent or coroutine execution.

Figure 3.2 shows part of an example trace for a disk, disk scheduler and several disk clients. Fundamentally, the current tracing model follows the activities of threads from creation, as they move through traceable objects, and finally destruction. Each traceable object has its own trace-line in the event display. Along any particular section of the trace-line, the form of the line indicates the status of threads, if any, within the traceable object. When a traceable object contains a ready thread, its trace-line is solid. When a traceable object contains only threads that are blocked, its trace line is dashed. A trace line is blank when its object contains no threads.

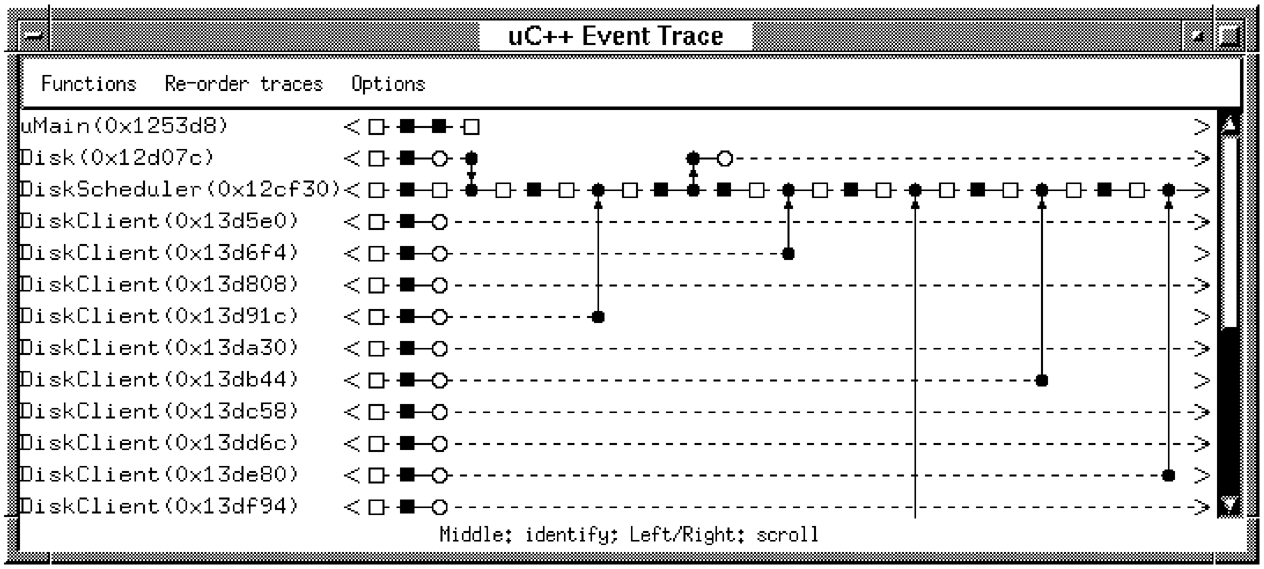


Figure 3.2: Disk Scheduler Trace

In general, events that cause a thread to block are displayed with an empty square □, and events that cause a thread to be made ready are displayed with a filled square ■. Special cases exist for the creation and destruction of traceable objects; both creation and destruction events are indicated with an empty square, which is consistent for a task because its thread is blocked at these times. Also, the initial starting and final stopping of a task's thread are indicated with a filled square, stating that the thread is ready to execute its main() member routine or its destructor routine, respectively.

A thread moves from one object to another by invoking a member routine of the other object; for mutex objects, a thread may be blocked at the call to a member routine because another thread is using that object. Whenever a call is made to a mutex object, a petition to enter is indicated with an empty circle ○. Once a thread is admitted, it is seen to transfer from one trace line to another; it transfers back when the member routine returns. Thread transfers are indicated by an arrow perpendicular to the trace-lines. An arrow originates and ends with filled circles ●.

One traceable object is always present in every display, uMain. The uMain task is internal to μC++; a user always supplies the body of this task, uMain::main(), and thus the uMain thread is always present.

3.2 Event Ordering

Events are places along a trace line based on relative, not absolute time. That is, events to the left of a particular event occur before it and events to the right (or below) occur after it. In general, the events are evenly spaced along a trace line and the distance between them does *not* reflect the amount of time between the events. Similarly, events on different trace lines have no time relationship. For example, task uMain appears to terminate at the same time or before all other traced objects, when in fact it is always the last object to be terminated. By imagining the trace lines as being elastic, it is possible to imagine stretching the trace line for uMain so that it is always longer than any other trace lines.

Partial orderings between trace lines occur because of synchronization events (● event and the arrows between the trace lines). At a synchronization point, all events to the left of that point along *both* trace lines occurred before the synchronization. Similarly, all events to the right of a synchronization point along *both* trace lines must occur after the synchronization.

3.3 Producer-Consumer Example

The following four programs illustrating tracing of tasks, monitors, semaphores and coroutines, respectively. The examples, and in particular their explanations, are intended to be read in the order in which they are presented. All four example programs are solutions to a producer-consumer problem. The first three programs all use a bounded buffer between the producer and consumer tasks to provide some asynchrony. Since the producer and consumer tasks are the same for all three of the different bounded buffers, that code is shown only once in Figure 3.3. The fourth producer-consumer solution uses coroutines, which have no concurrency, so there is no buffer. In all the examples, a single element is generated by the producer and given to the consumer.

```
uTask prod {
    buffer &buf;                        // reference to shared buffer
    void main() {
        buf.query();                    // check status of buffer
        buf.insert( 3 );                // insert an element
    }
  public:
    prod( buffer &b ) : buf( b ) {}
};

uTask cons {
    buffer &buf;                        // reference to shared buffer
    void main() {
        buf.remove();                   // remove an element
    }
  public:
    cons( buffer &b ) : buf( b ) {}
};

void uMain::main() {
    buffer b;                           // create bounded buffer task
    cons c( b );                        // create consumer task
    prod p( b );                        // create producer task
}
```

Figure 3.3: Producer-Consumer Tasks

3.4 Task Trace

The first example program (see Figure 3.4) illustrates interactions among tasks, which is visualized in Figure 3.5. Tasks c, of type cons, and p, of type prod, make calls to task b, of type buffer. Notice that the trace-lines for tasks c, p, and b are named according to the object type, rather than according to the variable itself; hence, multiple instances of the same traceable type have the same trace-line name (unless each object is given an individual name using uSetName). To aid in debugging, the address of each object is printed after it. These addresses can be used with a debugger to examine the individual instances.

The creation of tasks, and the starting of their threads, is indicated by the empty and filled squares at the beginning of the trace-lines. The trace-line is dashed between these two events, indicating the thread is blocked. After the filled square, the trace-line is solid, indicating the thread is ready. During the ready period, tasks c and p call task b's members buffer::insert() and buffer::remove(), respectively. Since b is a mutex object and insert and remove are mutex routines, entry is not immediate. Task c and p must first petition for entry, and wait for admission; the petition is indicated by the empty circle on the cons and prod trace-lines. Following the empty circle, the line is dashed, indicating that the thread is blocked waiting for entry. Finally, notice that prod::main() makes a call to a non-mutex member, buffer::query(). Because query is a non-mutex member, it cannot affect a thread, and hence, there is nothing in the trace about this call.

Task b's first action is to accept a call to its members insert and remove. (The accept of ~buffer is discussed later.) The acceptance is indicated by the empty square on the buffer trace-line, and following the acceptance event, the trace-line is dashed since b's thread is blocked. Eventually, task p gains admission to task b, which is indicated by an arrow from the prod trace-line to the buffer trace-line. This arrow indicates the transfer of task p's thread into task b.

Having gained permission to execute task b, task p's thread executes the member routine buffer::insert(), which is indicated by the solid line along the buffer trace-line. Using the caller's thread to execute a mutex member is an artifact of the implementation of μC++ [HN80]. The completion of buffer::insert() by p's thread is indicated by an arrow from the buffer trace-line to the prod trace-line, where p continues to execute (continued solid line), terminates its main() routine, has its destructor executed, and is destroyed.

Task b now continues execute after p's call has completed. This continuation is indicated by first re-stating that the task is blocked (dashed line along the buffer trace-line), then the re-start (filled square) and finally continued execution (solid trace-line). Task b now accepts calls to either its insert or remove member routines, which blocks it (indicated by the empty square). The process is now repeated for task c, which has been blocked since its petition to enter b (dashed line along the cons trace-line). After the call to remove by c, task c stops and is destroyed. Finally, task b accepts a call to its destructor (made implicitly at the end of the block in which b is declared), and it too stops and is destroyed.

Note that the task uMain is created, started, stopped and finally destroyed, as indicated by the top trace-lines in the display, but took no part in the rest of the program.

3.5 Monitor Trace

The second example program (see Figure 3.6) illustrates interactions of tasks with a monitor, which is visualized in Figure 3.7. In this case, tasks c and p make calls to a monitor b, of type buffer. As before, note that traceable objects are labelled with their class types rather than their variable names.

The creation of the tasks, and the starting of their threads, is indicated as before by the empty and filled squares at the beginning of the trace-lines. However, a monitor does not have a thread so its creation is marked with an empty square and there is a blank trace-line and no starting indicator. Both tasks c and p petition to enter the monitor, but the mutual exclusion property of the monitor only allows one thread in the monitor at a time.

```
#include <uC++.h>

uTask buffer {
    int front, back;                    // position of front and back of queue
    int count;                          // number of used elements in the queue
    int elems[5];
  public:
    buffer() { front = back = count = 0; }
    uNoMutex int query() { return count; }
    void insert( int elem ) { elems[back] = elem; }
    int remove() { return elems[front]; }
  protected:
    void main() {
        for ( ;; ) {
            uAccept( ~buffer ) {
                break;
            } uOr uWhen( count != 5 ) uAccept( insert ) {
                back = ( back + 1 ) % 5;
                count += 1;
            } uOr uWhen( count != 0 ) uAccept( remove ) {
                front = ( front + 1 ) % 5;
                count -= 1;
            } // uAccept
        } // for
    }
};

#include "ProdConsTasks.cc"
```

Figure 3.4: Task Buffer

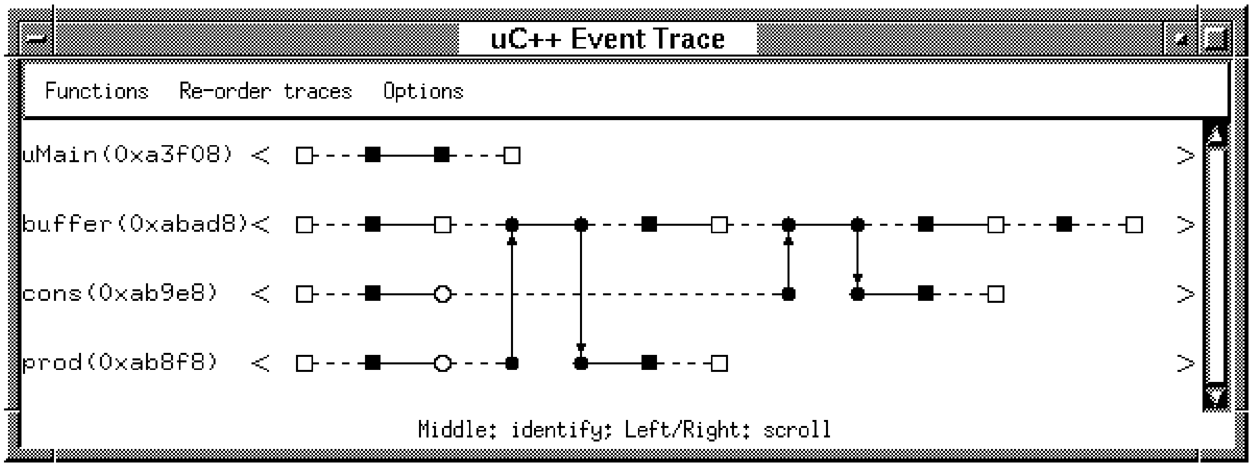


Figure 3.5: Task Trace

```
#include <uC++.h>

uMonitor buffer {
      int front, back;                        // position of front and back of queue
      int count;                              // number of used elements in the queue
      int elems[5];
   public:
      buffer() { front = back = count = 0; }
      uNoMutex int query() { return count; }
      void insert( int elem );
      int remove();
};

void buffer::insert( int elem ) {
      if ( count == 20 ) uAccept( remove );          // no calls to insert
      elems[back] = elem;
      back = ( back + 1 ) % 20;
      count += 1;
}

int buffer::remove() {
      if ( count == 0 ) uAccept( insert );           // no calls to remove
      int elem = elems[front];
      front = ( front + 1 ) % 20;
      count -= 1;
      return elem;
};

#include "ProdConsTasks.cc"
```

Figure 3.6: Monitor Buffer

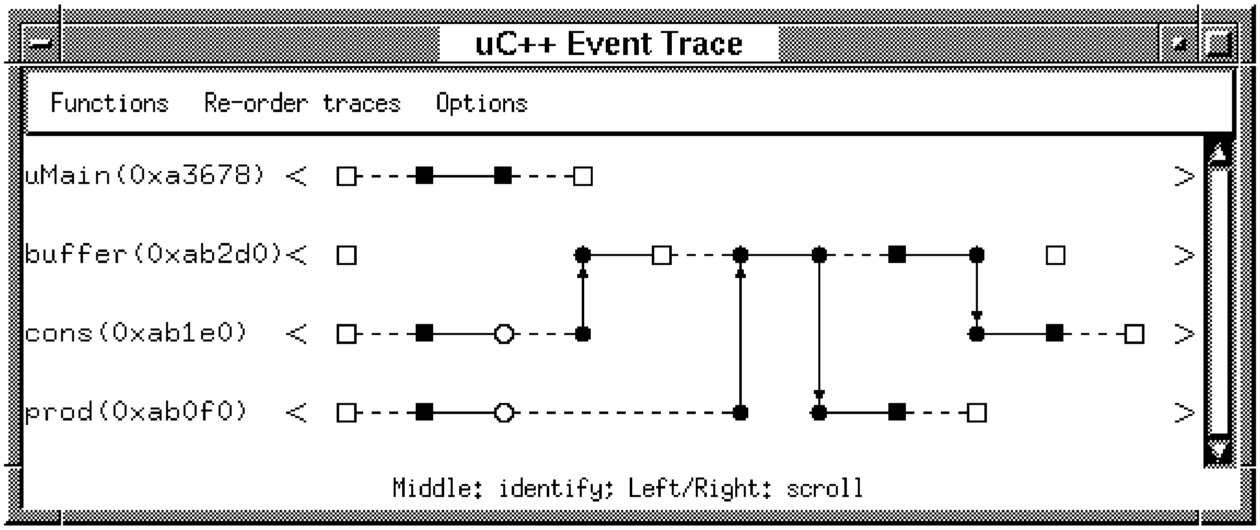


Figure 3.7: Monitor Trace

Task c gains entry first (indicated by an arrow from the cons trace-line to the buffer trace-line), checks if there is a data value (indicated by the solid line), and then blocks waiting for a call to insert because there is no data in the buffer (indicated by the empty square). Task p now gains entry (indicated by an arrow from the prod trace-line to the buffer trace-line), drops off a value (indicated by the solid line), leaves the monitor (indicated by an arrow from the buffer trace-line to the prod trace-line), stops execution and is destroyed. Task c now continues in the monitor, which is indicated by first restating that the task is blocked (dashed line along the buffer trace-line), then the re-start (filled square) and finally continued execution (solid trace-line), where task c picks up the data left by task p. Finally, task c leaves the monitor (indicated by an arrow from the buffer trace-line to the cons trace-line), stops execution and is destroyed.

3.6 Semaphore Trace

The third example program (see Figure 3.8) illustrates interactions of tasks with counting semaphores, which is visualized in Figure 3.9. In this case, tasks c and p make calls to a buffer b using semaphores to provide mutual exclusion and synchronization. As before, note that traceable objects are labelled with their class types rather than their variable names.

The creation of the tasks, and the starting of their threads, is indicated as before by the empty and filled squares at the beginning of the trace-lines. However, class buffer is not a mutex object so it does not have a trace-line; instead, the two semaphores within it have trace-lines. When a task calls the P or V routines of a semaphore, it petitions for entry into the semaphores as for a mutex member of a task or monitor. Task c calls the P routine of semaphore full (indicated by an arrow from the cons trace-line to the uSemaphore trace-line), executes in the P routine (solid trace-line), and blocks because the semaphore count is 0 (indicated by the empty square).

Task p calls the P routine of semaphore empty (indicated by an arrow from the prod trace-line to the uSemaphore trace-line), executes in the P routine (solid trace-line), returns (indicated by an arrow from the uSemaphore trace-line to the prod trace-line), and continues execution (solid trace-line). Because semaphore empty is non-zero, task p does not block. Next task p calls the V routine of semaphore full (indicated by an arrow from the prod trace-line to the uSemaphore trace-line), executes in the V routine (solid trace-line), returns (indicated by an arrow from the uSemaphore trace-line to the prod trace-line), and continues execution (solid trace-line). Because the V routine never blocks, calls to it always have this form. Task p then stops and is destroyed.

Task c now continues execution after task p unlocks the semaphore full on which it was blocked. This continuation is indicated by first restating that the task is blocked (dashed line along the uSemaphore trace-line), then the re-start (filled square) and finally continued execution (solid trace-line). Task c now returns (indicated by an arrow from the uSemaphore trace-line to the cons trace-line), and continues execution (solid trace-line). Next task c calls the V routine of semaphore empty (indicated by an arrow from the cons trace-line to the uSemaphore trace-line), executes in the V routine (solid-trace-line), returns (indicated by an arrow from the uSemaphore trace-line to the cons trace-line), and continues execution (solid-trace-line). Task c then stops and is destroyed.

3.7 Coroutine Trace

The fourth example program (see Figure 3.10) illustrates interactions of coroutines, which is visualized in Figure 3.11. (Actually, there are two program traces in this visualization and each is discussed separately). Like a monitor, a coroutine does not have its own thread, but it does have an execution-state, like a task. In the case of task interactions with other tasks or monitors, the task's thread always executes using its own execution-state. When a task interacts with a coroutine, the task's thread may execute with the coroutine's

```
#include <uC++.h>
#include <uSemaphore.h>

class buffer {
    int front, back;                    // position of front and back of queue
    int count;                          // number of used elements in the queue
    uSemaphore full, empty;             // synchronize for full and empty buffer
    int elems[5];
  public:
    buffer() : full( 0 ), empty( 5 ) { front = back = count = 0; }
    int query() { return count; }
    void insert( int elem ) {
        empty.uP();                     // wait if queue is full
        elems[back] = elem;
        back = ( back + 1 ) % 5;
        count += 1;
        full.uV();                      // signal a full queue space
    }
    int remove() {
        int elem;

        full.uP();                      // wait if queue is empty
        elem = elems[front];
        front = ( front + 1 ) % 5;
        count -= 1;
        empty.uV();                     // signal empty queue space
        return( elem );
    }
};

#include "ProdConsTasks.cc"
```

Figure 3.8: Semaphore Buffer

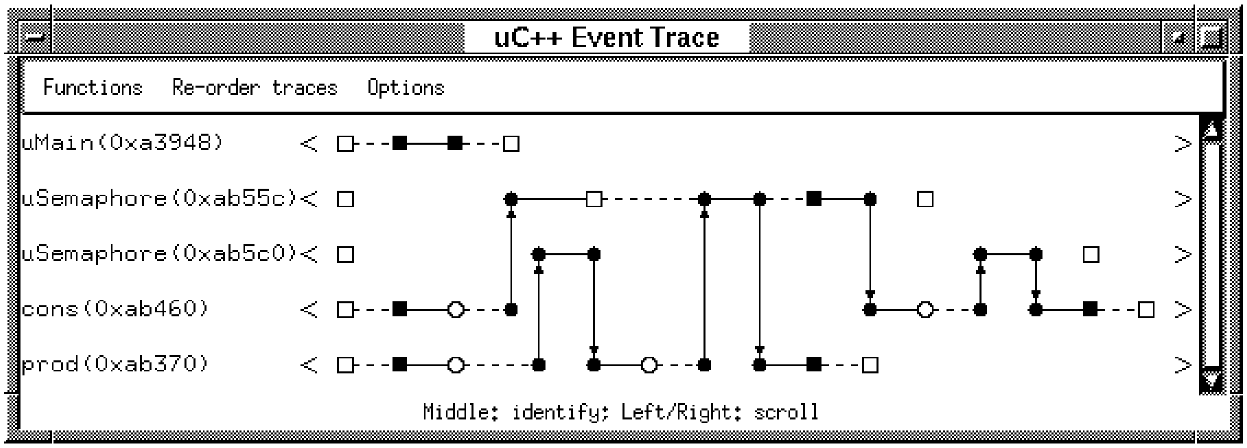


Figure 3.9: Semaphore Trace

execution-state. It is the transfers of a thread to and from a coroutine's execution-state, called a context switch, that is visualized for coroutines.

A context switch from a task's execution-state to the coroutine's execution-state occurs when a coroutine's member-routine executes a uResume statement and the thread transfers to the coroutine's last suspension point (often in main() routine). The thread then continues executing from that point using the coroutine's execution-state. Another context switch occurs when the coroutine's main() routine executes a uSuspend statement; control returns back to the point of the last resumption and the thread of the calling task is now executing on its own execution-state. A context switch between execution-states is visualized by an arrow between trace-lines. For example, an arrow from an object to the trace-line of a coroutine indicates that the coroutine has been resumed.

Mutex coroutines are more complicated because the call to the coroutine's member routine has to first petition for entry, then it can attempt to resume the coroutine's main() routine. While the petition to enter the mutex coroutines is indicated in the event trace (empty circle), only the context switch is indicated by an arrow between trace lines. Hence, if the mutex coroutine's member-routine does not resume the coroutine, the trace shows a petition to enter a mutex object but no transfer of thread to the particular mutex coroutine. Currently, there is no obvious way to display both a mutex entry call and a coroutine resumption without having coroutine member routines as traceable entities.

The first four trace lines of Figure 3.11 describe a no-mutex coroutine example. For the first time, task uMain's is explicitly involved. After all the objects have been created and started, the uMain task calls the start member of coroutine p, which resumes the coroutine (indicated by an arrow from the uMain trace-line to the prod trace-line). p's main() routine calls the member routine delivery in coroutine c, which resumes coroutine c (indicated by an arrow from the prod trace-line to the cons trace-line).

c's main() routine suspends back to the point of last resumption implicitly (indicated by an arrow from the cons trace-line to the prod trace-line) when its main() routine terminates (indicated by the empty square). p's main() routine suspends back to the point of last resumption implicitly (indicated by an arrow from the prod trace-line to the uMain trace-line) when its main() routine terminates (indicated by the empty square). Task uMain now stops and is destroyed.

The next four trace-lines of Figure 3.11 describe the execution of the coroutine example program changed so that coroutine cons is a mutex coroutine. The lower traces are identical to the upper traces except for the addition of a petition (indicated by the empty circle) when coroutine p calls cons::delivery(), since cons::delivery() is now a mutex member.

3.8 Event Information

Each event display element (□, ■, ○, ●) can be clicked on and a popup window appears containing additional information about that particular event. For example, clicking on the start-event (■) at the beginning of the trace line for the consumer task in Figure 3.7:

```
cons(0xab1e0) <  □---■ thread start [cons(0xab1e0), #1]
```

The event type is thread start, the object associated with the event is buffer, and this event is the second event for buffer (numbered from zero).

3.8.1 Thread-Ready Event

A thread-ready event is generated when a blocked task is restarted. For example, clicking on a thread-ready event (■) along the trace line for monitor bounded-buffer in Figure 3.7 produces:

```
●---■ thread ready "cons(0xab1e0)" [buffer(0xab2d0), #5]
```

```
#include <uC++.h>

uCoroutine cons {
      int elem;

      void main() { /* consume elem */ }
   public:
      void delivery( int e ) {
            elem = e;
            uResume;                          // restart cons::main
      }
};

uCoroutine prod {
      cons &c;

      void main() { c.delivery( 5 ); }
   public:
      prod( cons &c ) : c( c ) {}
      void start() { uResume; }              // restart prod::main
};

void uMain::main() {
      cons c;                                 // create consumer
      prod p( c );                            // create producer
      p.start();
}
```

Figure 3.10: Coroutine Producer/Consumer

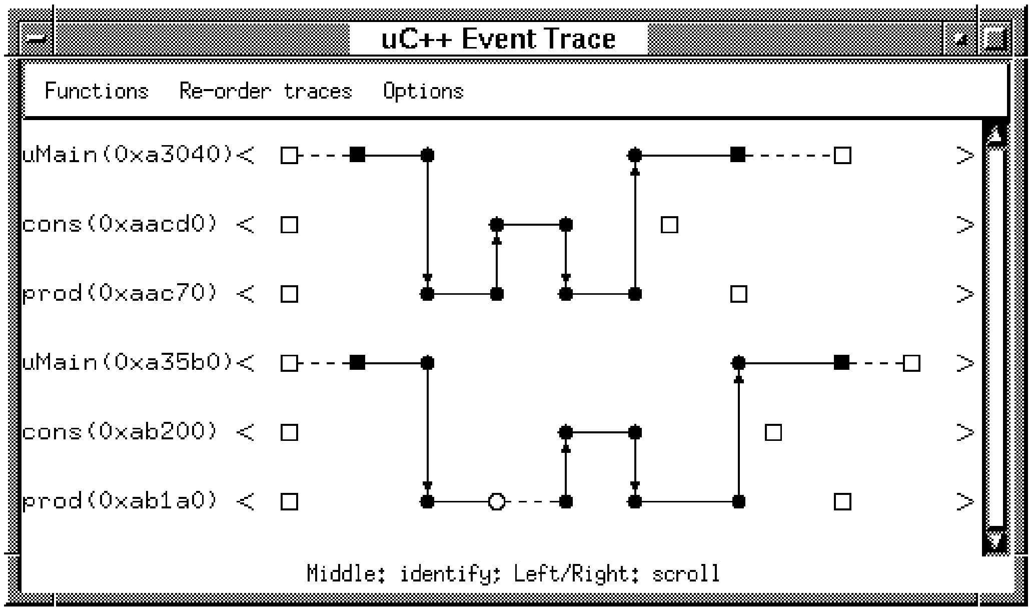
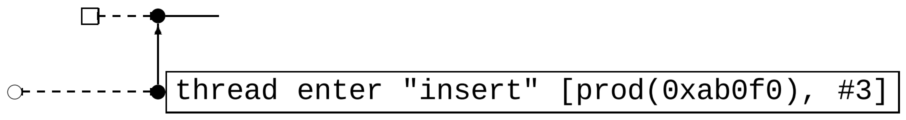


Figure 3.11: Coroutine Producer/Consumer Traces

The additional information in quotes is the name of the object restarted in the monitor. Thus, even if an object's delay point is off the trace window, it is possible to determine the particular object that restarted.

3.8.2 Synchronization Event

A synchronization event is generated when a thread transfers to and from a mutex object. For example, clicking on a synchronization event (●) along the trace line for the producer task in Figure 3.7 produces:



The additional information in quotes is the name of the mutex member routine in the buffer called by the consumer task. Clicking on any synchronization event that begins a thread transfer (thread enter or leave) indicates the name of the mutex member routine called or being returned to.

Chapter 4

Working with X Window/Motif

X Window/Motif were chosen for the basic visualization platform because of their current popularity and availability. Most of the ideas presented here would work equally well with other visualization toolkits, e.g., the xview toolkit or the athena widget set.

4.1 Structure of X

Version 11, release 6 of the X Window System is the first release intended to support client applications that work in a multi-threaded environment. See "Xlib - C Language X Interface" and "X Toolkit Intrinsics - C Language Interface", which are both distributed with the X Window System, for a detailed description on how to create concurrent client applications. X provides an abstract locking interface that can be implemented by any system- or user-level thread library. In general, locks in X are owner locks, i.e., they can be re-acquired by the thread of control that currently holds the lock, but they have to be released exactly as often as they were acquired.

At the *Xlib* level, multiple connections to the X server can be created and used concurrently, and multiple threads of control can access the same connection. Automatic fine-grain locking is done internally by the *Xlib* library. At the *X Toolkit Intrinsics* level (Xt), multiple application contexts can be created, each of which has internal display connections. In general, locking at this level is done on the basis of application contexts. Additionally, a global lock for the whole application can be used by widget developers, if global data has to be protected.

The Xt is designed to use an event loop and callbacks. That is, an application program passes control to X (by calling XtAppMainLoop), after it has built its windows and installed its callback routines. X then waits for events either from the user's terminal or other sources, such as timers, and it either invokes internal code, e.g., puts down a menu, and/or it interacts with the application code by invoking the specified callback routines.

The new X locking mechanisms can be used in various ways but the most common approach is the following. A dedicated thread of control is used to receive events from the X server and dispatches them to other objects. Concurrently, other threads can call X/Xt library routines to change application values.

4.2 X Window System and μC++

While the X client libraries are configured to work with the thread libraries of multiple vendors, some additional changes were required to make them compatible with μC++. As a first step, wrapper routines are planted, so that the X libraries, which are programmed in C, can internally use the uLock and the uCondition classes of μC++. This enables cooperation with the basic scheduling and lightweight-blocking mechanisms of μC++.

When using a vendor's kernel thread package, blocking I/O only blocks the thread of control that calls the appropriate operating system routine, for example select. Since μC++ provides user-level threads, its special lightweight-blocking I/O library must be used to achieve the same effect. Therefore, calls to the uSelect member routine of the uIOCluster are implemented in the X source code (by using wrappers from μC++ to C) at places where usually select is called. A dedicated uIOCluster object (see [BS95] for a description) is created in an X application, because the X libraries use data that is private to each UNIX process, like file descriptors, etc. A class uXwrapper is provided which migrates a task to this cluster when an object is created, and migrates it back when the object is destroyed. This class can be used for automatic migration in every routine that calls X library routines, like in:

```
#include <uXlib.h>

Display *dpy;

void createInterface() {
    uXwrapper dummy;            // automatic migration
    dpy = XOpenDisplay( NULL );
    // . . .
}                               // automatic migration back on destruction of dummy
```

In a μC++ application, timer interrupts are used to realize preemptive scheduling. Certain UNIX system calls return a failure value and set the error number, if a timer interrupt occurs while the system call is executed. This is partly handled in the Xlib, except for initialization of the socket connection with the X server. To prevent obscure error messages, preemption of μC++ is turned off, whenever a connection is established. Again, this mechanism is planted into the *Xlib* using wrappers from μC++ to C.

Unfortunately, to this end, the Motif widget library is not reentrant. Therefore, every call that accesses a Motif widget has to be explicitly made mutual exclusive for the whole application and the internal locking mechanisms of the Intrinsics library are largely obsolete. This can be seen in the the following example:

```
#include <uC++.h>
#include <uXlib.h>
#include <X11/Intrinsic.h>

Widget my_widget;
XtApplicationContext app;

void changeValue( int x ) {
    XtAppLock( app );
    XtProcessLock();
    XtVaSetValues( my_widget, XmNvalue, x, NULL );
    XtProcessUnlock();
    XtAppUnlock( app );
}
```

4.3 X Window Callbacks and C++

The following coding convention was developed for working with X callbacks in C++. Callbacks are C routines (C linkage) that are dynamically registered with X and subsequently invoked by X when particular X-window events occur, such as a button press on the mouse or screen. This is the mechanism that X uses to interact with an application. The following coding conventions allow C++ applications that interact with

X to still be written in an object-oriented style.

The conventions are as follows:

- All widget definitions should be defined as a class and that widget's callback routines should be private static members, as in:

  ```
  class widget {
      int foo;
      void bar() { . . . }
      static xxxCB( Widget widget, widget *This, XmAnyCallbackStruct *call_data ) {
          This->foo = 3;
          This->bar();
      }
    public:
      . . .
  };
  ```

  Making the callback routines static members reduces global name-space pollution and a static member routine is treated as a non-member routine so its address can be passed to X routines that are written in C (i.e., static member routines do not have an implicit this parameter).

- When installing a callback routine, as in:

  ```
  widget w;
  XtAddCallback( w, XmNactivateCallback, xxxCB, this );
  ```

  the last argument is a pointer to client data that is passed to the callback routine's second parameter. The convention requires that the last argument must always be the value of this, and hence, the second parameter to a callback routine is always a pointer to the type of class that contains the callback routine. Furthermore, the name of the second parameter should be This. Following this convention allows the callback routine to access all the data and routines in the containing widget class as if it was actually a member routine of the widget class (albeit explicitly through the This pointer). For example, in callback routine xxxCB, members foo and bar are both accessed through the This pointer.

Figure 4.1 shows a program that uses the coding conventions to handle its callback routines.

4.4 The μX Package

A collection of classes is available to simplify the task of interacting with X and Motif under μC++. By including file:

```
#include <uXlib.h>
```

after <uC++.h>, the basic cooperation is enabled and the class uXwrapper can be used for automatic migration (see Section 4.2). Additionally, the file

```
#include <uXmlib.h>
```

can be included to access classes to support using Motif: uXmwrapper and uXmCBwrapper. As stated previously, Motif is not thread-safe and mutual exclusion must be acquired before any operation can safely by invoked on Motif widgets. Additionally, if a task calls any X routine to send a request to the X server, this request is buffered in the library. If changes shall become visible immediately, the buffer has to be flushed. All this complexity is hidden within uXmwrapper, as well as migration. That is, if an object of uXmwrapper is created at the beginning of a routine, the task is migrated, the necessary locks are acquired and on destruction of the object the library's event buffer is flushed. The above example then looks like:

```
#include <uC++.h>
#include <uXlib.h>
#include <uXmlib.h>

Widget my_widget;

void changeValue( int x ) {
    Xmwrapper dummy( my_widget );
    XtVaSetValues( my_widget, XmNvalue, x, NULL );
}
```

When a callback routine is called, the lock for the application context and the global lock are acquired. If a callback contains a call to a mutex member of a task, this can lead to deadlock situations, if the task also tries to acquire the X locks to perform any changes in the interface. To handle this situation, the uXmCBwrapper class can be used to release the owner locks temporarily and re-acquire them on destruction of the object as in the following example:

```
#include <uC++.h>
#include <uXlib.h>
#include <uXmlib.h>

uTask fred {
    void main();
public:
    void request( int from );
}

fred f;

class MyWidget {
    int id;
    static void anyCallback( Widget w, MyWidget *This, XmAnyCallbackStruct *call_data) {
        XmCBwrapper dummy( w );
        f.request( This->id );
    }
```

One should be aware that between creation and destruction of a callback wrapper no lock is held and therefore, no call can safely be made that accesses Motif data. Additionally, a routine uXmAppMainLoop is provided that performs the necessary locking when Motif is used and replaces the usual XtAppMainLoop. In traditional X programming, the single thread of control passes control to X after it has initialized X and built a series of widgets. There are many kinds of problems that are *not* amenable to the event-loop programming style. In fact, virtually all concurrent applications are not amenable to this style because the application does not want to block waiting for X events. Instead, the concurrent application has other work to do and the X interface is only one component of this work. Therefore, the routine Uxmappmainloop can be called from a dedicated task object.

For convenience, the μX toolkit provides an additional server, called uServerXtShell, which is an instance of:

```
class uXtShellServer {
public:
    uXtShellServer();
    ~uXtShellServer();

    Widget XtCreateApplicationShell( const char *title, String fallBacks[] = NULL );

    void XtMainLoopBlock();
    void XtMainLoopNonBlock();
    void XtMainLoopWait();
}; // uXtShellServer

extern uXtShellServer *uServerXtShell;
```

This server is globally available to μC++ programs that interact with X. The server shell operations are:

XtCreateApplicationShell – creates a new shell widget under which a new widget hierarchy can be created. This routine handles the special case of initializing X/Intrinsic on the first call.

XtMainLoopBlock – acts similarly like the usual XtAppMainLoop. Control is passed to X and the routine eventually returns, when the user interface is destroyed.

XtMainLoopNonBlock – create a dedicated task that executes the main loop. The caller task continues execution.

XtMainLoopWait – waits for the task created with XtMainLoopNonBlock to finish execution. This can be used to synchronize with the destruction of the user interface.

```
#include <uC++.h>
#include <uIOStream.h>
#include <uXmlib.h>
#include <uXtShellServer.h>

#include <Xm/Xm.h>
#include <Xm/MainW.h>
#include <Xm/PushB.h>

class uPushMeWidget {
    static void uQuitCB( Widget widget, uPushMeWidget *This, XmAnyCallbackStruct *call_data ) {
        XtDestroyWidget( This->shell );                    // destroy parent and all its subwidgets
    } // uPushMeWidget::uQuitCB

    static void uPushMeCB( Widget widget, uPushMeWidget *This, XmPushButtonCallbackStruct *call_data ) {
        uCout << "Please don´t tread on me" << endl;
    } // uPushMeWidget::uPushMeCB

    Widget shell;
  public:
    uPushMeWidget( Widget shell ) : shell( shell ) {
        Widget mainWindow = XtVaCreateManagedWidget( "main", xmMainWindowWidgetClass, shell,
                                        XmNwidth, 150, XmNheight, 75,
                                        NULL );
        XmString quitTitle = XmStringCreateSimple( "Quit" );
        Widget menuBar = XmVaCreateSimpleMenuBar( mainWindow, "menuBar",
                                        XmVaCASCADEBUTTON, quitTitle, ´Q´,
                                        NULL );
        XmStringFree( quitTitle );
        XtManageChild( menuBar );
        XtAddCallback( XtNameToWidget( menuBar, "button_0" ), XmNactivateCallback,
                                        (XtCallbackProc)uQuitCB, this );
#       define uPushMeTitle "Tread on me"
        Widget uPushMe = XtVaCreateManagedWidget( "uPushMe", xmPushButtonWidgetClass, mainWindow,
                                        XtVaTypedArg, XmNlabelString, XmRString, uPushMeTitle, sizeof(uPushMeTitle),
                                        NULL );
        XtAddCallback( uPushMe, XmNactivateCallback, (XtCallbackProc)uPushMeCB, this );
    } // uPushMeWidget::uPushMeWidget
}; // uPushMeWidget

void uMain::main() {
    Widget shell = uServerXtShell->XtCreateApplicationShell( "uPushMeTest" );
    {
        uXmwrapper dummy( shell );
        uPushMeWidget pmw( shell );
        XtRealizeWidget( shell );
    }
    uServerXtShell->XtMainLoopBlock();
    uCout << "Quitter!" << endl;
} // uMain::main

// Local Variables: //
// tab-width: 4 //
// compile-command: "u++-work -g uPushMe.cc -I/u/usystem/software/visualization/inc -I/u/usystem/software/visualization/X11R6/include -I/
// End: //
```

Figure 4.1: C++ Callback Coding Style

Chapter 5

Explicit Statistical Monitoring

Explicit statistical monitoring requires a detailed understanding of the low-level structure provided by the μC++ MVD toolkit. Explicit monitoring is performed by two kinds of objects: watcher and sampler objects. Figure 5.1 shows how watchers and samplers cooperate to produce a display. Both the watcher and sampler objects need to be fairly small and fast because they may be probing an application at a high frequency.

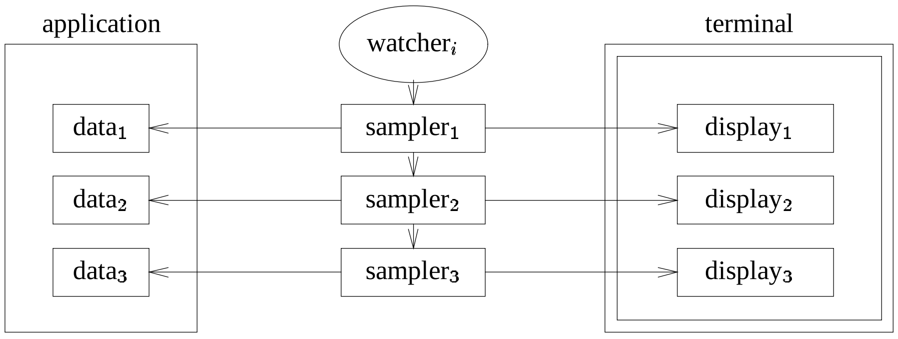


Figure 5.1: Watcher/Sampler Structure

5.1 Watcher Object

A *watcher* is a task that manages an event list of user-specified sampler objects. At specified intervals, the watcher invokes a sampler object on its event list, and that sampler then inspects part of the application and possibly displays the inspected data. Multiple watchers can be created if the number of samplers is large and/or the sampling frequencies are high; normally a single watcher is sufficient. A watcher is least invasive on machines with multiple processors where the watcher can execute on a separate processor independent of the application. On a single processor machine, a watcher can only be expected to produce a coarse-grained view of the application. Nevertheless, this still can be useful in many cases. The watcher is provided by the μC++ monitoring toolkit and its interface is:

```
uTask uWatcher {
  public:
     void add( uBaseSampler &sampler );        // add sampler to event list
     void remove( uBaseSampler &sampler );     // remove sampler from event list
};
```

The member routines add and remove a sampler to/from the watcher's event list, respectively. Normally, these routines are called by the constructor and destructor of a sampler, respectively.

5.2 Sampler Object

A *sampler* inspects data values in an application and possibly displays the inspected data. A sampler has two basic operations: poll memory in a particular way to determine some relevant information, and display that information in some useful format. These two operations are invoked by the watcher object that a sampler is associated with. In detail, a sampler indicates to the watcher the frequency in microseconds at which its polling and display operations are invoked. These two frequencies may be different. A sampler must be derived from class uBaseSampler to ensure it has the necessary member routines that are called by the watcher:

```
class uBaseSampler {
  protected:
     int pollFreq, displayFreq;                // poll and display frequency
  public:
     uBaseSampler( uWatcher &w, const int pollFreq, const int displayFreq );
     virtual void poll() = 0;                  // called by watcher
     virtual void display() = 0;               // called by watcher
     void pause();                             // stop sampling temporarily
     void resume();                            // restart sampling
};
```

The sampler is associated with watcher w, which calls the sampler's poll member at frequency pollFreq and display member at frequency displayFreq.

The following generic sampler preforms simple sampling of a numeric memory value:

```
template<class T> class uNumSampler : public uBaseSampler { // T requires: 0, =, >, <, +, /, <<
  public:
     uNumSampler( uWatcher &w,                 // watcher where sampler resides
          int pollFreq,                        // polling frequency
          int displayFreq,                     // display frequency
          T &locn,                             // location to be sampled
          char *name,                          // widget title
          T dialMin,                           // absolute minimum
          T dialMax,                           // absolute maximum
          T zero,                              // zero value for type T
          uNumGaugeTL::GaugeType gauge         // initial gauge type
     );
     void poll();
     void display();
}; // uNumSampler<T>
```

This sampler examines a numeric object at memory location locn. To make the numeric sampler generic it is necessary to pass a value corresponding to zero for the type T.[1] This simple sampler maintains a running average of the sampled values at location locn as well as the minimum and maximum values. Appendix A shows the complete generic numeric sampler.

The generic sampler uNumSampler is used in the following way to sample numeric values in a program:

[1]This is an artifact of the C++ type system.

```
void uMain::main() {
    uWatcher w;                        // create watcher
    int i;                             // target variable
    uNumSampler<int> sd( w, 50000, 500000, i, "i", 0, 50000, 0, uNumGaugeTL::digital ),
    sb( w, 50000, 500000, i, "i", 0, 50000, 0, uNumGaugeTL::bargraph ),
    ss( w, 50000, 500000, i, "i", 0, 50000, 0, uNumGaugeTL::speedometer );

    for ( i = 0; i <= 50000; i += 1 ) uDelay();           // delay necessary on a uniprocessor
} // uMain::main
```

In this example, a watcher is created with three samplers for variable i each displaying the value of i in a different format. (On a multiprocessor, the watcher is automatically created on a separate processor to reduce the probe effect.) When the program is run, the variable i is displayed in three different formats: digitally, as a bar graph, and as a speedometer, as illustrated in Figure 5.2. In the bargraph, the minimum, average and

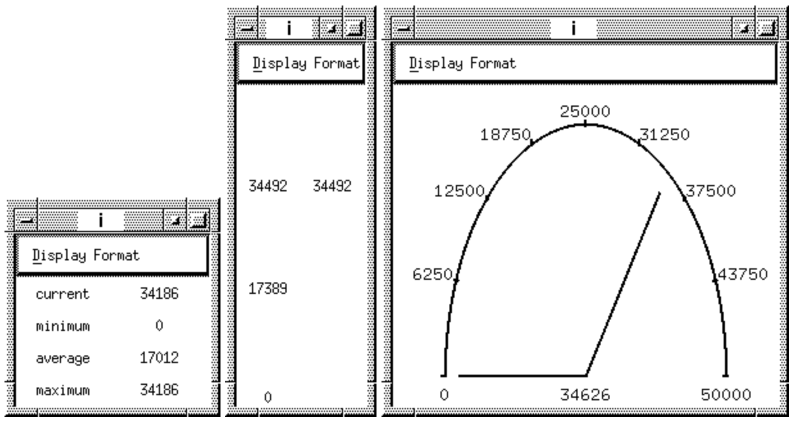


Figure 5.2: Different Numeric Displays: digital, bar graph, speedometer

maximum values appear in the left column, while the current value appears in the right column. The spacing of the values between the top and bottom of the window shows the relative location of the values between the absolute minimum and maximum (0-50000). In the speedometer, there are pointers to the minimum, current, and maximum values and the current value is also displayed digitally at the bottom-centre of the speedometer (the average value is not displayed). Since, the current and maximum values are the same for i, the current and maximum pointer are displayed on top of one another, and hence, appear as a single pointer line. Normally, it is unnecessary to create three samplers to obtain three views of a variable because the pull-down menu, `Display Format`, allows any of the three display forms to be changed dynamically to digital, bar graph or speedometer.

A sampler can be written to sample any data structure and generic samplers can be constructed for common types of data structures so many users will never have to write a sampler. For example, uNumSampler samples any type that has the operations, = > < + / <<, and a 0 (zero) constant. One important aspect of this design is that the sampling frequency and the display frequency are independent. This feature allows filtering of large numbers of values while still being able to capture peak values. For example, the minimum and maximum values can be sampled at a high rate, but writing to the display may only be done every 1/30 of second because that is the refresh rate of the display device. ([] report that a refresh rate as slow as 1/10 of a second is sufficient for people to perceive smooth motion.) A sampler can also greatly reduce the display updates by not sending an event if the value(s) have not changed. Furthermore, a sampler can dynamically control its poll and display frequency by adjusting the values of the protected variables pollFreq and displayFreq of the base class uBaseSampler. For example, the longer it has been since a change occurred in a value being probed, the less urgent it is to probe again. Finally, it is a sampler's responsibility to deal with non-atomic data structures. For example, when updating a binary tree, there may be a short time interval when the tree is disconnected and a sampler that is probing the tree must be prepared to deal with this. Such samplers must be very pessimistic about the object they are examining and very robust in their probing algorithms. For example, a simple precautionary measure when sampling is to check that a pointer is within the address space before dereferencing it.

Thus, users can write their own samplers or use one of the the predefined samplers. We hope to enlarge the current suite of samplers and displayers as time permits.

5.3 Coding Conventions

Additional coding conventions are used in the construction of samplers and their display widgets. These conventions follow directly from the X/C++ conventions discussed in Chapter 4. A sampler passes its address to its display widget, just as and the widget does to its callbacks, and the callbacks to X. This approach allows the object receiving the address to communicate back to its creator. In Figure 5.3, these pointers allow the top communication lines from right to left. (even though the pointers go the other direction). As well, each creator maintains a pointer to the object it created to allow the bottom communication lines from left to right. Hence, there is bidirectional communication among the objects used as follows: output flows from the sampler to the widget to X; input flows from X to the callback to the widget to the sampler and possibly to the watcher.

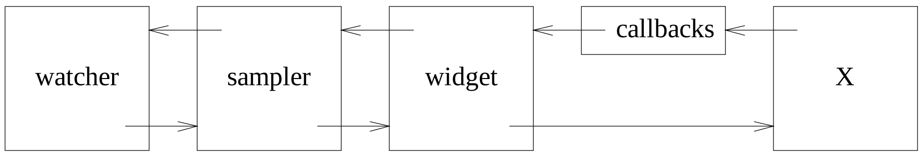


Figure 5.3: Coding Conventions

5.4 Animated Bounded Buffer Example

The following example visualizes the effect of inserting a bounded buffer between a producer and consumer task using synchronous communication (see Figure 5.4). The buffer is size 100 and the number of elements in the buffer is displayed by a horizontal slider (0 on the left). Feedback controls are provided to adjust the speed of the producer and consumer tasks. These controls adjust random delay cycles, with the random number generated around an average with a particular standard deviation. By playing with the delay controls, it is possible to see that the buffer fills or empties if the speed of the producer and consumer varies only slightly or their speed fluctuates significantly. There are 4 basic components to the animated bounded buffer: display widget, buffer sampler, visual bounded buffer, driver; each is discussed separately.

5.4.1 Display Widget

While the display widget is the most complex of the components, this results from all of the calls to X to build the display. Since this manual is not an X manual, the details of the display only appear in Appendix B and are not discussed. Only the interface to the display widget is Figure 5.5 is relevant to this discussion.

There are 5 callback routines corresponding to the quit button and 4 sliders. While the quit button halts the program, the slider values have to be available to the driver so it can adjust its execution. Each callback

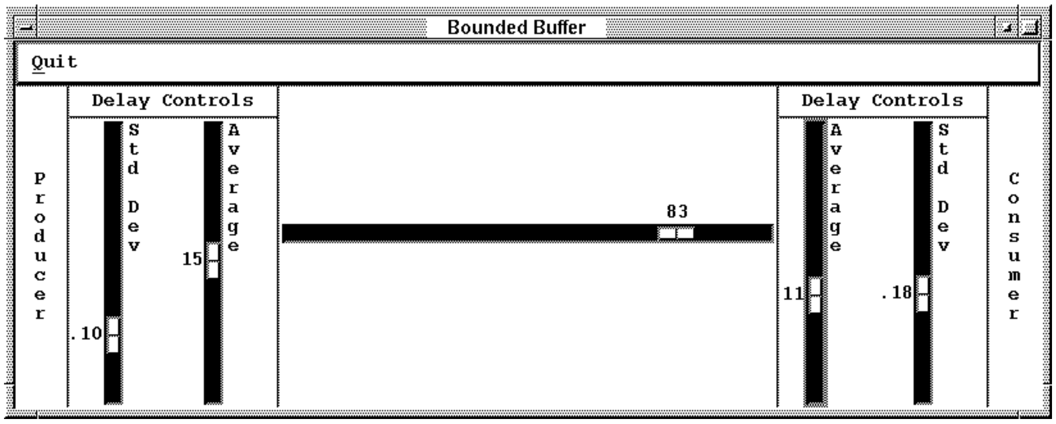


Figure 5.4: Animated Bounded Buffer

routine has a This parameter, whose value is an instance of the class that contains them. The constructor receives the title for the display and the size of the buffer, which is used to set the maximum size of the buffer slider (horizontal slider). There are 4 get routines to retrieve the values returned from the sliders to adjust the producer and consumer tasks and one set routine to move the buffer slider to the current number of items in the buffer.

5.4.2 Buffer Sampler

Figure 5.6 shows the sampler that periodically samples the size variables in the monitor and updates the display widget accordingly. The sampler's constructor has the same first 5 parameters as the numeric sampler, uNumSampler, followed by the size of the monitor's buffer to set the horizontal slider. The constructor uses these parameters to initialize the base sampler, location to be sampled, and display widget, respectively. The constructor and destructor add and remove the sampler to/from the watcher in the appropriate manner. The poll routine makes a copy of the buffer's current size and the display routine displays this value in the widget's horizontal slider. Finally, the 4 get routines return the values of the sliders directly from the display widget.

5.4.3 Visual Bounded Buffer

Figure 5.7 shows the enhancements to an existing generic bounded buffer implemented as a monitor. The visual bounded buffer inherits the insert and remove routine from the base monitor, and all the variables needed to manage the buffer. The base monitor is extended with watcher and sampler variables, along with 4 get routines to obtain the values from the display sliders. The visual buffer's constructor initializes the base buffer, widget and sampler, respectively. (Notice that the watcher is initialized in the constructor's initialization list even though it does not require any initialization arguments. This is because the sampler depends on the watcher, and hence, the watcher must be initialized first. By appropriately ordering the variables in the constructor's initialization list, it is possible to control the ordering of initialization.) The monitor variable count is passed to the sampler, pcsampler, as the variable to be sampled. This variable contains the number of items in the bounded buffer.

```
#include "uXserver.h"

class uProdConsWidget {
    static void QuitCB( Widget widget, uProdConsWidget *This, XmAnyCallbackStruct *call_data );
    static void prodStdWDCB( Widget widget, uProdConsWidget *This, XmScaleCallbackStruct *call_data );
    static void prodAvgWDCB( Widget widget, uProdConsWidget *This, XmScaleCallbackStruct *call_data );
    static void consStdWDCB( Widget widget, uProdConsWidget *This, XmScaleCallbackStruct *call_data );
    static void consAvgWDCB( Widget widget, uProdConsWidget *This, XmScaleCallbackStruct *call_data );

    Widget parent, bufScaleWD;
    const int NumDecPts = 2;
    float prodStd, prodAvg, consStd, consAvg;
  public:
    uProdConsWidget( const char *title, int bufSize );
    virtual ~uProdConsWidget();

    float getProdStd() {
        return prodStd;
    } // uProdConsWidget::getProdStd

    float getProdAvg() {
        return prodAvg;
    } // uProdConsWidget::getProdAvg

    float getConsStd() {
        return consStd;
    } // uProdConsWidget::getConsStd

    float getConsAvg() {
        return consAvg;
    } // uProdConsWidget::getConsAvg

    void setValue( int val );
}; // uProdConsWidget
```

Figure 5.5: Display Widget

5.4.4 Driver

Figure 5.8 shows the driver program using the visual bounded buffer between a producer and consumer task. The only additions over a non-visual bounded buffer are the two calls to uDelay before and after the calls to insert and remove, respectively. These two delay calls read the current values of the appropriate sliders associated with the buffer display and then delay the producer or consumer task for some appropriate period of time.

```
#include "uBaseSplr.h"
#include "uProdConsWD.h"

class uProdConsSampler : public uBaseSampler {
    const int &locn;
    int curr;
    uProdConsWidget pcWD;
  public:
    uProdConsSampler( uWatcher &w,          // watcher where sampler resides
            const int pollFreq,             // polling frequency
            const int displayFreq,          // display frequency
            const int &locn,                // location to be sampled
            const char *title,              // widget title
            int bufSize ) :                 // buffer size
            uBaseSampler( w, pollFreq, displayFreq ), locn( locn ), pcWD( title, bufSize ) {
        w.add( *this );                     // add sampler to watcher′ s event list
    } // uProdConsSampler::uProdConsSampler

    ~uProdConsSampler() {
        if ( !paused ) {                    // check if already removed from watcher′ s event list
            w.remove( *this );              // remove sampler from watcher′ s event list
        } // if
    } // uProdConsSampler::~uProdConsSampler

    void poll() {
        curr = locn;                        // get current value
    } // uProdConsSampler::poll

    void display() {
        pcWD.setValue( curr );              // display current value
    } // uProdConsSampler::display

    float getProdStd() {
        return pcWD.getProdStd();
    } // uProdConsSampler::getProdStd

    float uProdConsSampler::getProdAvg() {
        return pcWD.getProdAvg();
    } // uProdConsSampler::getProdAvg

    float uProdConsSampler::getConsStd() {
        return pcWD.getConsStd();
    } // uProdConsSampler::getConsStd

    float uProdConsSampler::getConsAvg() {
        return pcWD.getConsAvg();
    } // uProdConsSampler::getConsAvg
}; // uProdConsSampler
```

Figure 5.6: Buffer Sampler

```
#include "/u/usystem/software/collection/src/uBoundedBuffer.h"

template<class ELEMTYPE> uMonitor VisualBoundedBuffer : public uBoundedBuffer<ELEMTYPE> {
    uWatcher w;
    uProdConsSampler pcsampler;
  public:
    VisualBoundedBuffer( const int size = 10 ) :
            uBoundedBuffer<ELEMTYPE>( size ),
            w(), pcsampler( w, 50000, 100000, count, "Bounded Buffer", size ) {
    } // VisualBoundedBuffer::VisualBoundedBuffer

    uNoMutex float getProdAvg() {
        return pcsampler.getProdAvg();
    } // VisualBoundedBuffer::getProdAvg

    uNoMutex float getProdStd() {
        return pcsampler.getProdStd();
    } // VisualBoundedBuffer::getProdStd

    uNoMutex float getConsAvg() {
        return pcsampler.getConsAvg();
    } // VisualBoundedBuffer::getConsAvg

    uNoMutex float getConsStd() {
        return pcsampler.getConsStd();
    } // VisualBoundedBuffer::getConsStd
}; // VisualBoundedBuffer
```

Figure 5.7: Visual Bounded Buffer

```
#include <math.h>

inline double UniformRand( void ) {
    return (double)( rand() % 10000 + 1 ) / (double)10000;
} // UniformRand

inline double ExpRand( double Average ) {
    return - (double)Average * log( UniformRand( ) );
} // ExpRand

inline double HypExpRand( double Average, double Std ) {
    if ( UniformRand() < (double)Std ) {
        return ExpRand( 1.0 ) * ( (double)Average / (2 * (double)Std ) );
    } else {
        return ExpRand( 1.0 ) * ( (double)Average / ( 2 * ( 1 - (double)Std ) ) );
    } // if
} // HypExpRand

uTask producer {
    VisualBoundedBuffer<int> &buf;
    void main() {
        uFloatingPointContext fpcxt;
        for ( ;; ) {
            uDelay( (int)HypExpRand( buf.getProdAvg(), buf.getProdStd() ) ); // spend time producing
            buf.insert( rand() % 100 + 1 );          // insert item into queue
        } // for
    } // producer::main
  public:
    producer( VisualBoundedBuffer<int> &buf ) : buf( buf ) {
    } // producer::producer
}; // producer

uTask consumer {
    VisualBoundedBuffer<int> &buf;
    void main() {
        uFloatingPointContext fpcxt;
        for ( ;; ) {
            int item = buf.remove();                 // remove from front of queue
            uDelay( (int)HypExpRand( buf.getConsAvg(), buf.getConsStd() ) ); // spend time consuming
        } // for
    } // consumer::main
  public:
    consumer( VisualBoundedBuffer<int> &buf ) : buf( buf ) {
    } // consumer::consumer
}; // consumer

void uMain::main() {
    srand( getpid() );                               // set random number seed
    VisualBoundedBuffer<int> buf( 100 );             // create a buffer monitor
    {
        consumer cons( buf );                        // create a consumer tasks
        {
            producer prods( buf );                   // create producer tasks
        }
    }
} // uMain::main
```

Figure 5.8: Animated Bounded Buffer: Driver

Chapter 6

Debugger

This chapter describes kdb (kalli's debugger) [Kar95], a multithreaded debugger for debugging multi-threaded μC++ [BS95] applications. Both uniprocessor and multiprocessor μC++ applications may be debugged. kdb is based on parts of the gdb debugger [SP95], and therefore, some of the commands and their syntax are the same. While knowledge of gdb is an advantage, it is not essential.

6.1 Before Starting kdb

In order to debugged with kdb, a μC++ application must be compiled with the compilation flags -debug and -g; -debug is the μC++ default, so normally only -g has to be specified.

6.2 Accessing kdb

To access kdb, the path:

```
/u/usystem/software/MVD/bin
```

must be added to the PATH environment variable. This variable is usually initialized in the .cshrc file in a user's home directory. Usually, there is a line in .cshrc that looks like:

```
setenv PATH '/bin/showpath $HOME /bin standard'
```

This line can be augmented to:

```
setenv PATH '/bin/showpath /u/usystem/software/MVD/bin $HOME /bin standard'
```

6.3 Interface

The interface for kdb is based on the *X Window System*, the *X Toolkit Intrinsics* and the *Motif* widget set. All windows in the interface can be resized from the window manager's border and the size of components in a window adapt automatically. Additionally, some windows are subdivided into stacked *panes*, separated with a horizontal line with a small box on the right-hand side. By dragging on a pane separator, it is possible to adjust the vertical size of a pane within a window.

6.4 Starting kdb

The debugger operates as a server at which the application registers as a client (multiple clients are allowed). Both server and client can be started together using the command:

```
kdb application-name [application-argument-list]
```

Connecting the client (application) to an already running server (kdb) is discussed in Section 6.6. Connecting the server (kdb) to an already running client (application) is discussed in Section 6.7.1.

The debugger's main window appears when the debugger starts (see Figure 6.1 (a)) and is discussed in Section 6.7. Notice that button NewTarget is "grayed out" (the affect is more noticeable on a computer screen), which means that the button is *inactive* so pressing it does nothing. When the debugger is in different modes, certain buttons are inactive (insensitive) because they are meaningless in that mode, and become active (sensitive) again when the debugger's mode changes.

6.5 Terminating kdb

To terminate kdb, the main window may be closed at any time by using the appropriate mechanism of the window manager. Each window manager has a pulldown or popup menu with a close option that removes a window, and when the main window is removed, kdb is terminated. If an application is executing when kdb terminates, it is released to continue execution normally. Other kdb windows may be closed at any time through the window manager to remove them from the display without terminating kdb.

6.6 Reusing kdb

After an application finishes execution, kdb is still running and can proceed with a new debugging session for the same or new application. In fact, because the cost of starting the debugger is fairly large, it is efficient to use the same debugger instance for multiple debugging sessions.

When the application is finished, the debugger's main window enters the state shown in Figure 6.1 (b). The button New Target is now active, and correspondingly, most of the other buttons that deal with the application are inactive. To start debugging a new application, a user must do *both* of the following in *either* order:

1. click on the New Target button to prepare the debugger for a new application.

2. connect another application to kdb using the command:

   kdb_target *application-name [application-argument-list]*

   □ WARNING: It is very common to forget to press the New Target button when trying to start a new debugging session. □

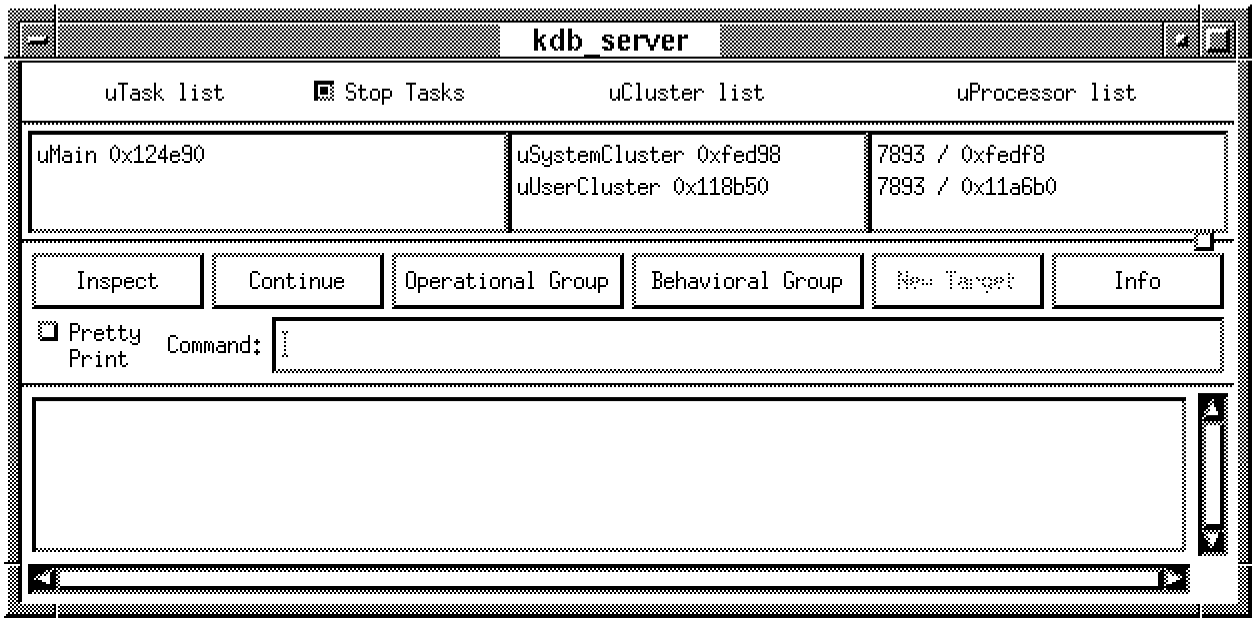
6.7 Main Window

When the debugger starts, the main window appears and has 2 panes (see Figure 6.2).

1. The top pane contains dynamic tables of all tasks, clusters and processors[1] currently active in the application.

2. The bottom pane contains controls for debugger interactions. Most of the operations at this level manage task groups or query global variables.

[1]Clusters and processors are artifacts of μC++ and not discussed here (see "μC++ Annotated Reference Manual").

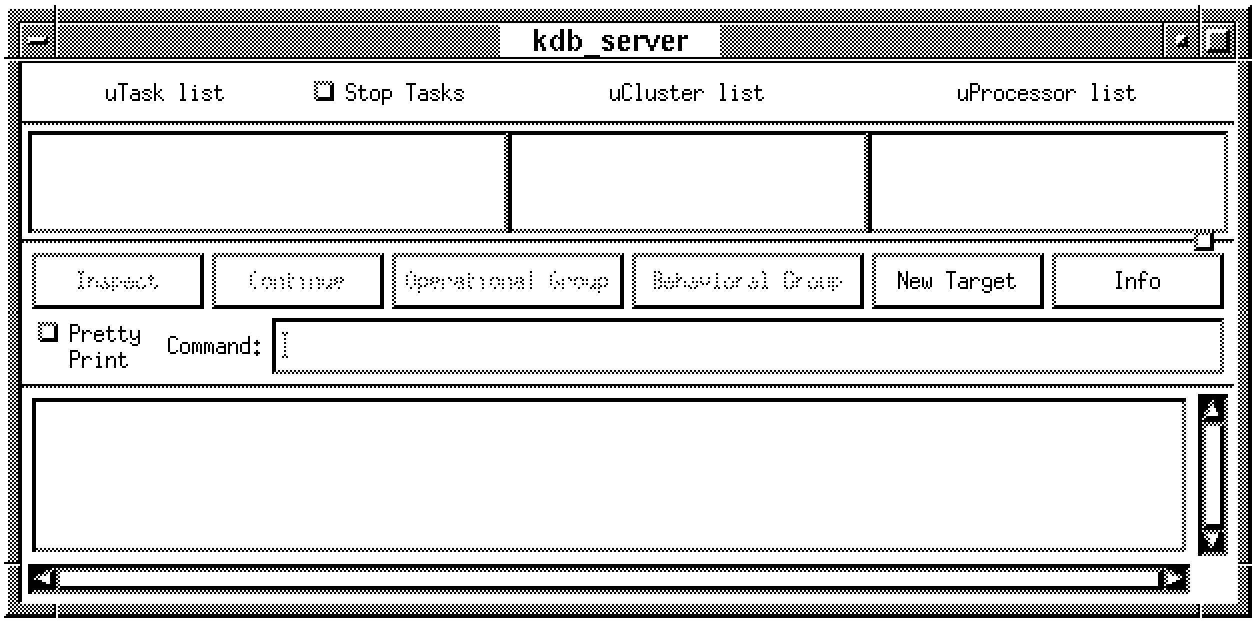


```
uMain 0x124e90
```

```
uSystemCluster 0xfed98
uUserCluster 0x118b50
```

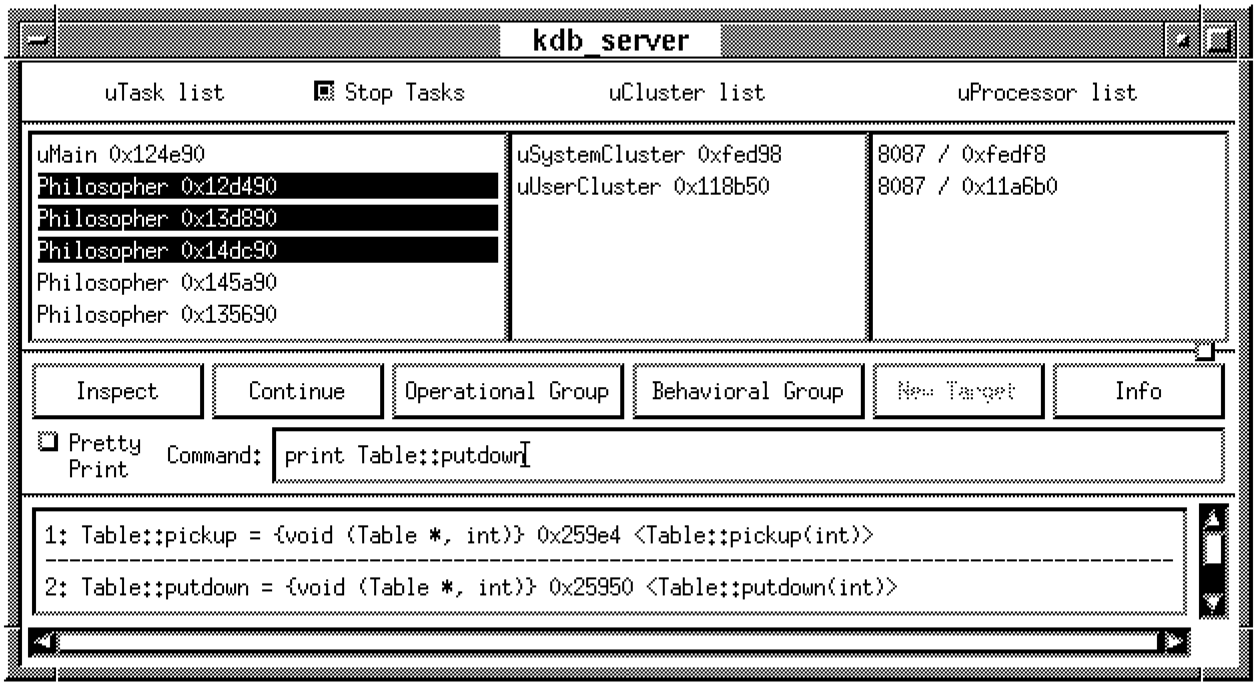
```
7893 / 0xfedf8
7893 / 0x11a6b0
```

(a) Startup Mode



(b) End Mode

Figure 6.1: kdb Modes



```
uMain 0x124e90
Philosopher 0x12d490
Philosopher 0x13d890
Philosopher 0x14dc90
Philosopher 0x145a90
Philosopher 0x135690
```

```
uSystemCluster 0xfed98
uUserCluster 0x118b50
```

```
8087 / 0xfedf8
8087 / 0x11a6b0
```

```
print Table::putdown
```

```
1: Table::pickup = {void (Table *, int)} 0x259e4 <Table::pickup(int)>
-----------------------------------------------------------------------------------------
2: Table::putdown = {void (Table *, int)} 0x25950 <Table::putdown(int)>
```

Figure 6.2: Main Window Showing Symbol Lookup

6.7.1 uTask list

The left column in the top pane is a list of all tasks currently active in the μC++ application. Tasks are identified by their name and memory address in the target application. The default name for a task is its type name; μC++ allows a user-specified name for a task to be set at any time, but the debugger only uses the name given when a task is created (either the default type name or a user-specified name passed to the constructor uBaseTask).

When connected to the debugger, the default action is for each task in the application to stop at the beginning of its main member routine, unless the Stop Tasks button is toggled. When Stop Task is off, newly created tasks continue execution immediately.

It is possible to select one or more tasks in the task list by clicking on a task and dragging to select a group (the Philosopher tasks highlighted are selected in Figure 6.2).

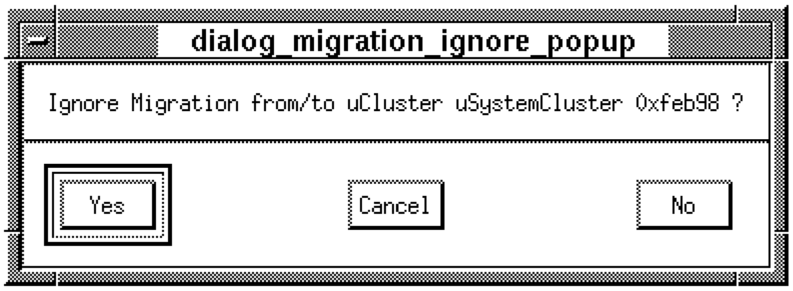
□ Currently, it is not possible to select a non-contiguous set of objects to form a group. This capability will be provided shortly. □

6.7.2 uCluster list

The centre column in the top pane is a list of all clusters currently active in the μC++ application. Clusters are identified by their name and memory address in the application.

There is one operation available for clusters. By double-clicking on an entry in the cluster list, a window pops up (see Figure 6.3) to control whether the debugger is notified of task migration to and from this cluster. Selecting Yes means the debugger ignores further task migration and No means the debugger is notified about further task migration.

Because every migration of a task involves additional overhead during debugging, there can be situations where ignoring migration improves performance. However, this mechanism should be used only



Ignore Migration from/to uCluster uSystemCluster 0xfeb98 ?

Figure 6.3: Task Migration Notification Dialog

by experienced μC++ programmers to prevent problems, since the debugger and application are no longer coordinated.

6.7.3 uProcessor list

The right column in the top pane is a list of all processors (UNIX processes) currently active in the μC++ application. Processors are identified by their UNIX process id and memory address in the application. (Both processors have the same UNIX process id in Figure 6.2, which means the μC++ application is running uniprocessor.)

The only operation available for processors is a similar mechanism to ignore migration of processors among clusters. Again, this is an insecure optimization mechanism that should only be used by experienced μC++ programmers.

6.7.4 Control Buttons

Individual task information and control is provided through a task window (see Section 6.8), which is created by selecting a task(s) in the main window and clicking on Inspect, or implicitly pops up when a breakpoint is encountered for a task. Double-clicking on a single entry of the task list does the same as selecting a task and clicking on Inspect. The group operation, Continue, continues execution of all tasks of a selected group. If some of the tasks are already running, the continue request is ignored for these tasks. All other group operations are accessed through the popup group windows available from buttons Operational Group and Behavioural Group. Both kinds are discussed in detail in Section 6.9. The New Target button was discussed in Section 6.6. The Info button pops up a short copyright note. The Command area is for typing in the following commands:

- [print|p] *expression*: print global variable names or expressions.
- [attach|a] *executable-file process-id*: attach kdb to the executable that is already running.

Positioning the cursor in the command box makes it active so a command can be entered; press the Return key to execute the command. The result of the command appears in the bottom of the pane. If the Pretty Print button is toggled, complex data is shown in an easier to read structured way but results in longer output. Each output in the bottom pane is numbered on the left and separated with a row of dashes, so a history of all output can be scrolled through using the scroll bar on the right of the bottom pane.

6.8 Task Window

A task window appears when a task is inspected from the main window, or a breakpoint is encountered for a task; it has 2 panes (see examples in Figure 6.4). (Note, this window is similar to the main debugging window of xxgdb [CW94].)

1. The top pane displays information about source code. The highlighted line (reverse video) marks the current statement being executed when stepping through a program, or by clicking on a line, that line becomes the operand for a command like setting or clearing a breakpoint.

2. The bottom pane contains controls for sequential debugging of a thread and an output area for the results of certain operations.

A task window begins by showing the last known position of the corresponding task by giving the source file name and line number (top right of the top pane), and displaying the source code (bottom of the pane) with the particular line highlighted. Button Stopped Position returns to the last known execution position from the current selection.

Button Source Files pops up a list of all source files for the program, which can be selected for display in the top pane (see Figure 6.5). To select a source file, either double-click on its name or click on its name and OK.

6.8.1 Control Buttons

The control buttons in the bottom pane (see examples in Figure 6.4) are used to control execution of a task. While some tasks may be blocked by the debugger during the debugging process, *all other tasks continue to execute in real time*; these tasks may become application blocked on resources that are held by the debugger blocked tasks.

To set a breakpoint, position the highlighting bar by clicking on the desired line of code and then click on Break. A breakpoint is removed by clicking on Clear when the the breakpoint's location is highlighted. If there is no breakpoint set at the line, a warning beep is sounded. Clicking the Continue button resumes execution of the task.

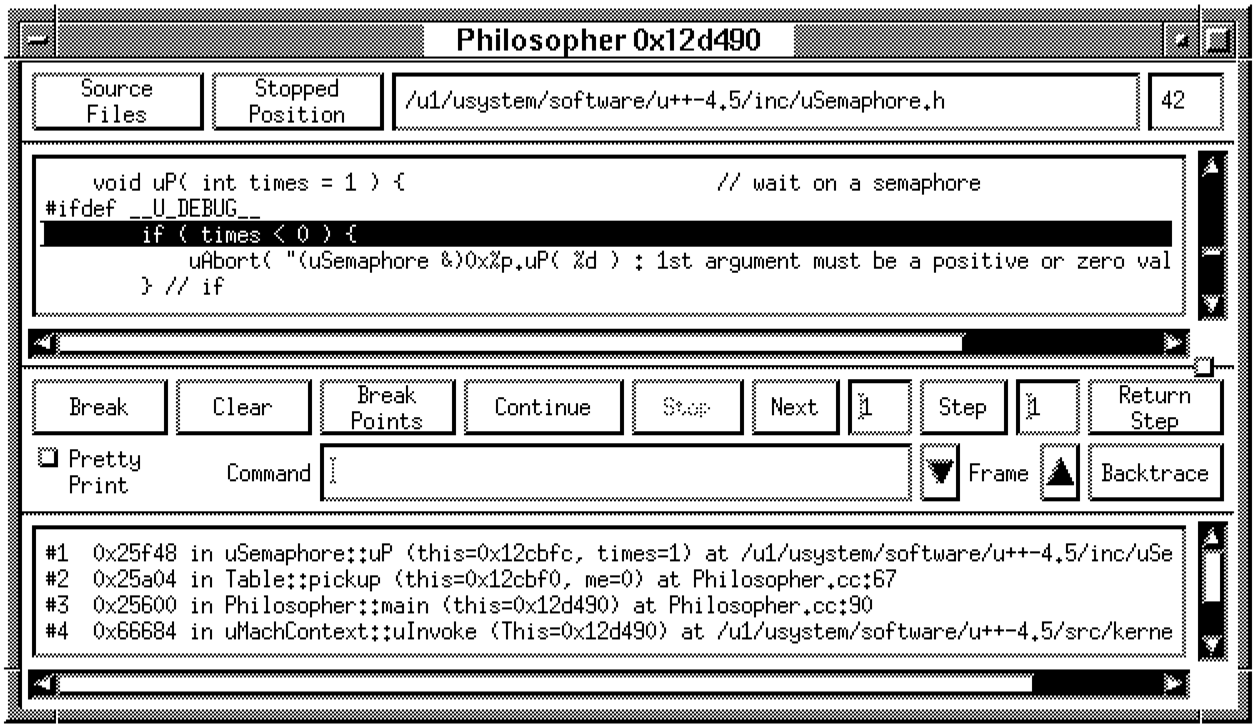
Clicking Next executes a line of source code. If this line contains routine calls, the routines are completely executed. This is different from Step, which also executes a line of source code, but steps into each routine call and stops execution at the beginning of the routine. For both Next and Step buttons, a number can be specified in the corresponding text field beside each button to perform multiple operations.

Clicking on Return Step causes the task to continue execution until the end of the current routine is reached. Execution stops after the routine call to the just completed routine.

□ μC++ inserts code in a μC++ application at the start and end of each mutex member, and the constructors and destructors of coroutines, mutex objects, and tasks. Therefore, use Next at the start of these routines to avoid entering the inserted code, which is normally irrelevant during debugging. If you enter one of the μC++ routines by mistake, use Return Step to finish the inserted code and continue in the desired routine. □

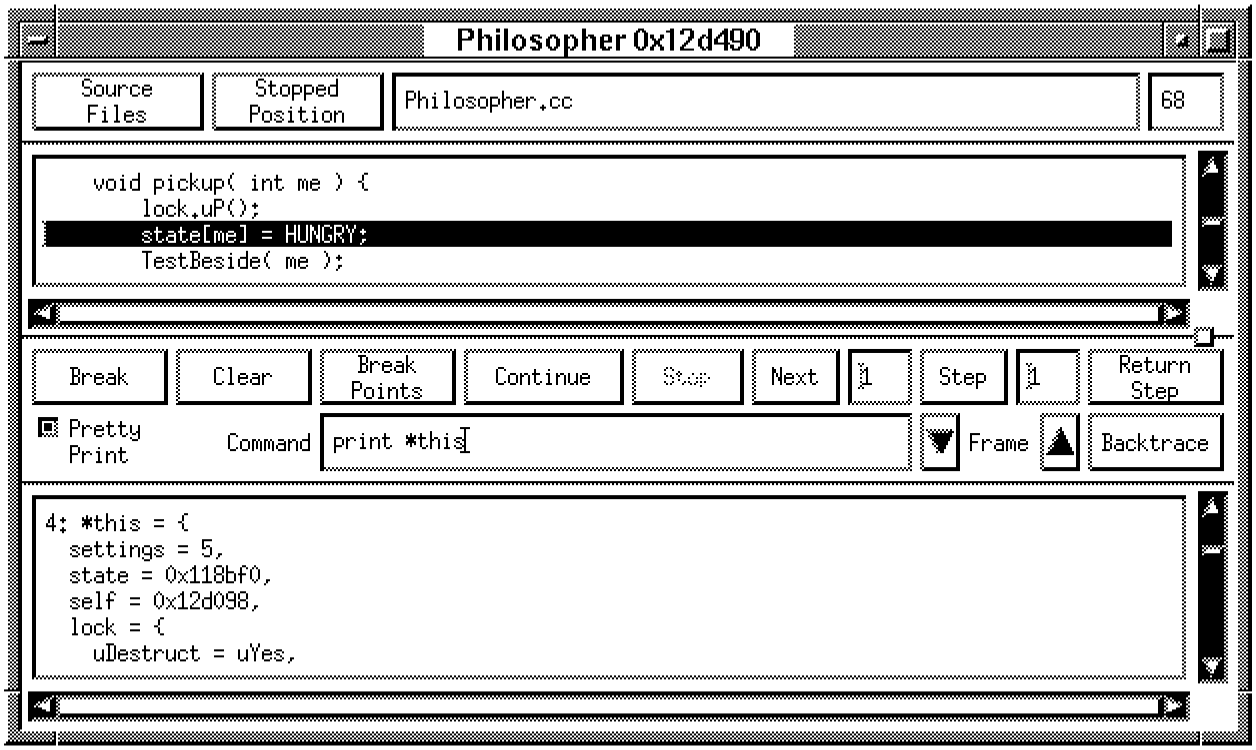
6.8.2 Backtrace

The Backtrace button produces a backtrace of the calling stack for the task, which is shown at the bottom of the pane (see Figure 6.4 (a)). The arrow buttons to the left and right of the label Frame step up and down the calling stack, respectively, which also causes the appropriate source code for the stack frame to be displayed in the top pane. While moving up and down the calling stack, routines may be encounters that were compiled *without* -g, e.g., system libraries, UNIX signals, μC++ kernel. In this case, a message appears in the top pane stating this fact; keep pressing the Frame arrows until routines are found that were compiled with the -g flag.



```
/u1/usystem/software/u++-4.5/inc/uSemaphore.h    42

    void uP( int times = 1 ) {                          // wait on a semaphore
#ifdef __U_DEBUG__
        if ( times < 0 ) {
            uAbort( "(uSemaphore &)0x%p.uP( %d ) : 1st argument must be a positive or zero val
        } // if

#1  0x25f48 in uSemaphore::uP (this=0x12cbfc, times=1) at /u1/usystem/software/u++-4.5/inc/uSe
#2  0x25a04 in Table::pickup (this=0x12cbf0, me=0) at Philosopher.cc:67
#3  0x25600 in Philosopher::main (this=0x12d490) at Philosopher.cc:90
#4  0x66684 in uMachContext::uInvoke (This=0x12d490) at /u1/usystem/software/u++-4.5/src/kerne
```

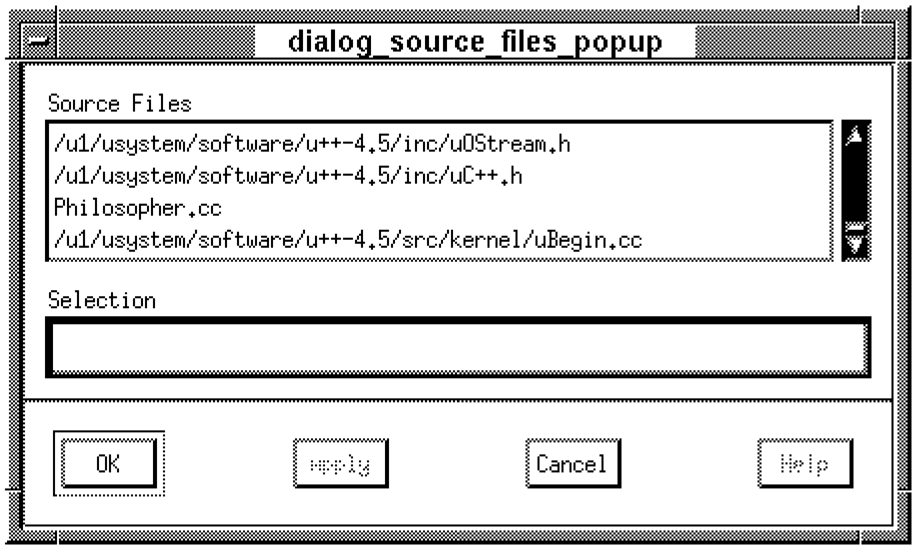
(a) Backtrace



```
Philosopher.cc    68

    void pickup( int me ) {
        lock.uP();
        state[me] = HUNGRY;
        TestBeside( me );

print *this

4: *this = {
  settings = 5,
  state = 0x118bf0,
  self = 0x12d098,
  lock = {
    uDestruct = uYes,
```

(b) Symbol Lookup

Figure 6.4: Task Windows



```
/u1/usystem/software/u++-4.5/inc/uOStream.h
/u1/usystem/software/u++-4.5/inc/uC++.h
Philosopher.cc
/u1/usystem/software/u++-4.5/src/kernel/uBegin.cc
```

Figure 6.5: Source File List

6.8.3 Commands

The Command area is for typing in the following commands:

- [print|p] *expression*: print global variable names or expressions.
- [break|b] [[*executable-file*]:*line-no* | *function-name*] [if *simple-expression*]: set breakpoint at specified line (in source file) or function.

  The conditional clause, i.e., the if clause, means the breakpoint is only triggered if an applicable task encounters the breakpoint *and* the conditional expression is true. The *simple-expression* is of the form:

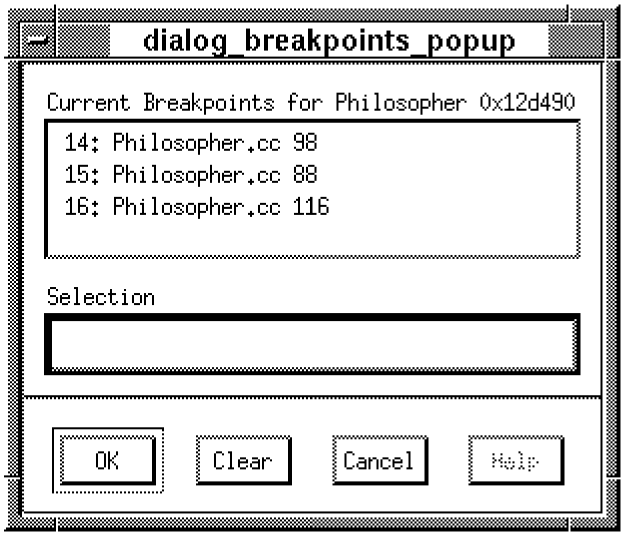
  integer-[variable | *constant]* [== | != | >= | <= | > | <] *integer-[variable* | *constant]*

- [clear|c] [[*executable-file*]:*line-no* | *function-name*]: clear breakpoint at specified line (in source file) or function.

To look up symbol information, the same mechanism is used as in the main window. The difference is that variable names (or expressions) are evaluated in the scope of the current stack location. For example, in Figure 6.4 (b), the name this refers to the object on which the class method pickup is invoked. If the Pretty Print button is toggled, complex data is shown in an easier to read structured way but results in longer output. Each output in the bottom pane is numbered on the left and separated with a row of dashes, so a history of all output can be scrolled through using the scroll bar on the right of the bottom pane.

6.8.4 Breakpoint Selection

The Breakpoints button pops up a list of all breakpoints currently set for this task (see Figure 6.6). A breakpoint is selected by double-clicking on its list entry and the source file containing it appears in the top pane, positioned at the breakpoint's location. Additionally, breakpoints can be deleted by clicking on a breakpoint's entry in the list and the Clear button of the breakpoint window.



```
Current Breakpoints for Philosopher 0x12d490
14: Philosopher.cc 98
15: Philosopher.cc 88
16: Philosopher.cc 116
```

Figure 6.6: Breakpoint List

6.8.5 Examining a Running Task

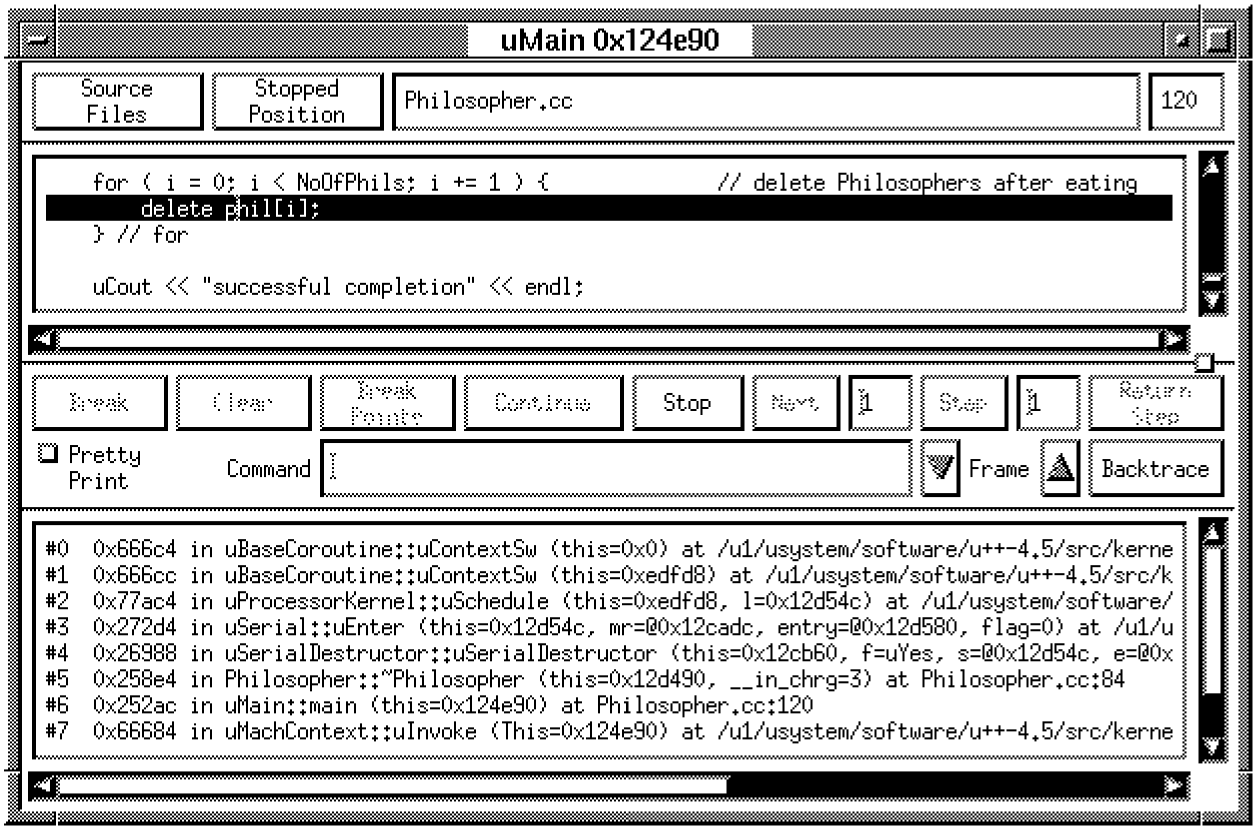
When a task is running, the task window mode changes to the state shown in Figure 6.7: only the Stop and Backtrace buttons are active. Clicking Stop stops the task at the next possible location. However, if a task is currently blocked in the application, e.g., if a task is waiting on the entry queue of a mutex object, the stop request does not take effect until the task is eventually made active again.

It is possible to monitor the execution of a running task by clicking on the Backtrace button, which gives a snapshot of the current execution stack for a task (see bottom of Figure 6.7). This capability is especially helpful if an application runs into a deadlock. If a deadlock occurs, tasks are still running from the debugger's perspective, but blocked from the thread system's perspective. Double clicking on the Backtrace button for a task enables the frame arrow buttons, making it possible to move up and down a running task's stack frame. **This operation is safe only when a task is application blocked.** Variables of a running task can be looked up relative to the last known position in the same way as for a stopped task. In this way, it is possible to find call cycles that lead to the deadlock.

This same facility can be used when an application terminates with an error, such as a task producing a segment fault. When the error occurs, μC++ prints an error message in the shell where the application is connected to the debugger, such as:

```
plg[3]% kdb a.out
philosopher 0 will eat 30 noodles
philosopher 0 is eating 4 noodles leaving 26 noodles
uC++ Run Time Error (UNIX pid:26958) uSigHandlerModule::uSigSegvBusHandler,
                        addressing error (sigsegv) at location 0x1004730
Error occurred while executing task "Philosopher" (0xd66d0)
```

As well, the debugger stops all tasks in the application and a window pops up (see Figure 6.8) to control when the application terminates and produces a core file. Before pressing OK, kdb can be used to examine all the tasks in the application. In particular, the error message states that the error occurred in "Philosopher" (0xd66d0), so that task should be inspected. The task is running at the time of the error, so double clicking on the Backtrace button is necessary to allow moving up the task's stack. Since the debugger has stopped all tasks, this is a safe operation. When examination of the application is complete, clicking OK in the abort window terminates the application.



```
    for ( i = 0; i < NoOfPhils; i += 1 ) {          // delete Philosophers after eating
        delete phil[i];
    } // for

    uCout << "successful completion" << endl;
```

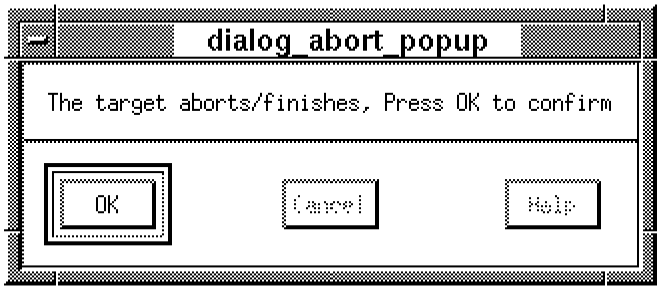
```
#0  0x666c4 in uBaseCoroutine::uContextSw (this=0x0) at /u1/usystem/software/u++-4.5/src/kerne
#1  0x666cc in uBaseCoroutine::uContextSw (this=0xedfd8) at /u1/usystem/software/u++-4.5/src/k
#2  0x77ac4 in uProcessorKernel::uSchedule (this=0xedfd8, l=0x12d54c) at /u1/usystem/software/
#3  0x272d4 in uSerial::uEnter (this=0x12d54c, mr=@0x12cadc, entry=@0x12d580, flag=0) at /u1/u
#4  0x26988 in uSerialDestructor::uSerialDestructor (this=0x12cb60, f=uYes, s=@0x12d54c, e=@0x
#5  0x258e4 in Philosopher::~Philosopher (this=0x12d490, __in_chrg=3) at Philosopher.cc:84
#6  0x252ac in uMain::main (this=0x124e90) at Philosopher.cc:120
#7  0x66684 in uMachContext::uInvoke (This=0x124e90) at /u1/usystem/software/u++-4.5/src/kerne
```

Figure 6.7: Task Window Showing Running Task

□ While it is possible to view different source files when a task is running, as soon as the task encounters a breakpoint, the top pane shows the current stopped location. □

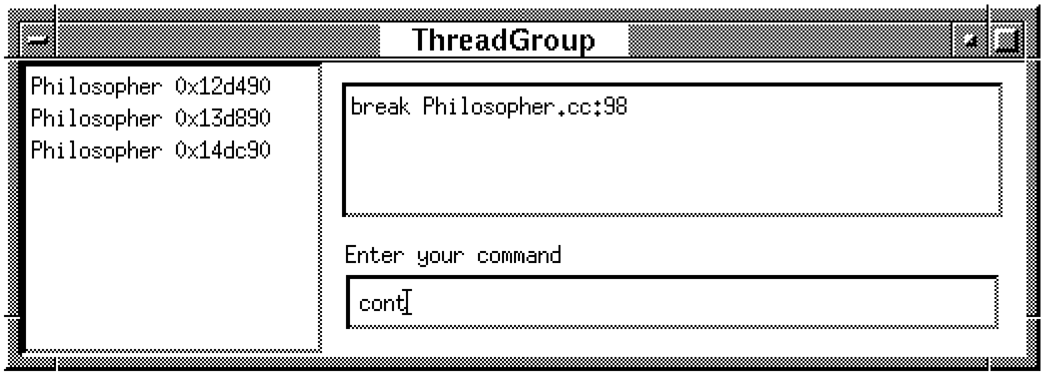
6.9 Thread Groups

Different kinds of possibly overlapping groups can be formed from selected tasks. However, grouping tasks together does not affect the ability to control tasks separately; furthermore, task reactions to the commands issued to a group, such as setting a breakpoint or continuing execution, become visible in each task's window.



The target aborts/finishes, Press OK to confirm

Figure 6.8: Application Abort Dialog



```
Philosopher 0x12d490
Philosopher 0x13d890
Philosopher 0x14dc90
```

```
break Philosopher.cc:98
```

```
cont
```

Figure 6.9: Operational Group Window

6.9.1 Operational Group Window

Tasks can be grouped together and operations can be issued on a group of tasks as a user convenience, rather than entering multiple commands for each task. Multiple operational group windows can be created, each defining a different group. A task may appear in any number of operational groups. When an operational group window is closed, that group is terminated.

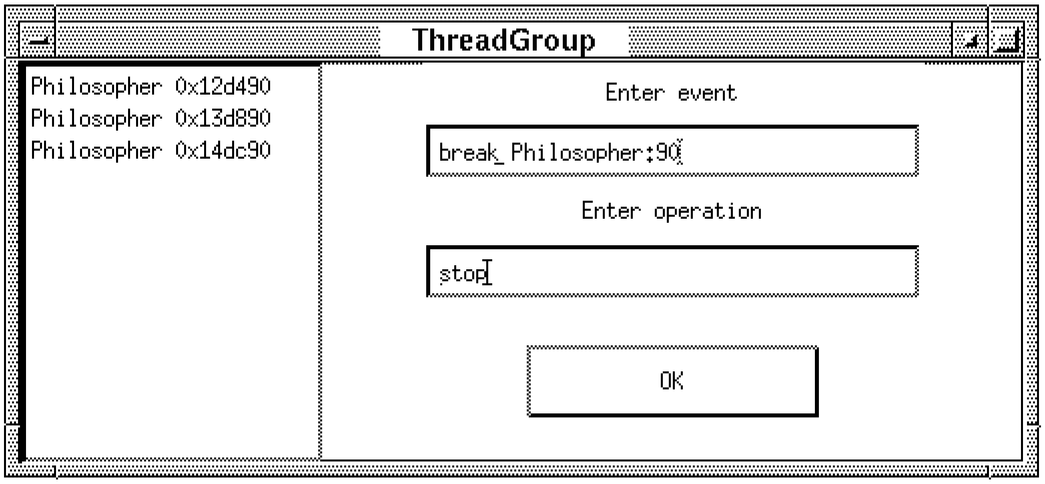
Clicking on Operational Group forms a group of all tasks currently selected in the main window (see selected Philosopher tasks in Figure 6.2), and pops up a window (see Figure 6.9) where commands can be issued on all tasks that belong to the group. The following commands can be entered in this window and the corresponding operation is performed on each task in the group:

- [break|b] [[*executable-file*]:*line-no* | *function-name*] [if *simple-expression*]: set breakpoint at specified line (in source file) or function.
- [clear|c] [[*executable-file*]:*line-no* | *function-name*]: clear breakpoint at specified line (in source file) or function.
- stop: stop task execution
- cont: continue task execution
- next *[number]*: execute *number* source lines of code (default value for *number* is 1) not entering routine calls
- step *[number]*: execute *number* source lines of code (default value for *number* is 1) entering routine calls

In general, if a command is not applicable to one of the tasks, e.g., stop for an already stopped task, the command is silently ignored. The upper right area of the group window shows a history of group commands entered for this operational group. Clicking on a previous command copies it into the enter command area. Figure 6.9 shows an operational group window where a breakpoint was previously set for each philosopher task (upper right), and each task is about to be continued.

6.9.2 Behavioural Group Window

A behavioural group is a set of tasks whose behaviour is linked to some event. In other words, if an event occurs for any task in a behavioural group, an action is applied to all the tasks in the group, e.g., when one



Philosopher 0x12d490
Philosopher 0x13d890
Philosopher 0x14dc90

break Philosopher:90

stop

Figure 6.10: Behavioural Group Window

task triggers a breakpoint, all tasks in the group are stopped. Hence, a behavioural group must have an event and operation associated with it. Furthermore, a task can appear in only one behavioural group at a time because actions in one group might cause inconsistent behaviour in another group.

Clicking on Behavioural Group forms a group of all tasks currently selected in the main window (see selected Philosopher tasks in Figure 6.2), and pops up a window where an event and action can be issued for all tasks that belong to the group (see Figure 6.10). The following commands can be entered in the event window:

- [break|b] [[*executable-file*]:*line-no* | *function-name*] [if *simple-expression*]: set breakpoint at specified line (in source file) or function.

The following commands can be entered in the operation window:

- stop: stop task execution

Figure 6.10 shows a behavioural group window where a breakpoint is about to be set for each philosopher task, and each task in the group will be stopped when one of the tasks reaches the breakpoint.

Chapter 7

Miscellaneous

7.1 Contributors

While many people have made numerous suggestions, the following people were instrumental in turning this project from an idea into reality. Peter Buhr got μC++ and X to speak on intimate terms and forced the coding conventions on others; he also wrote the statistical sampling code, the documentation, and did other sundry coding as needed. David Taylor wrote the event tracer and made many changes as we whined and complained. Scott Zinn and Rob Good kept the event tracer going on plg. Johan Larson wrote the first version of the event generation code for μC++ and Peter Buhr wrote the second version. Rory Jacobs wrote the prototype concurrent debugger for the μSystem, and “Totally Cool” Martin Karsten wrote the concurrent debugger, kdb, for μC++. Jun Shih extended kdb with lots of additional features.

Appendix A

Numeric Sampler

```
// -*- Mode: C++ -*-
//
// Visualization 1.0, Copyright (C) Peter A. Buhr 1992
//
// uNumSplr.h – generic sampler for numeric types
//
// Author : Peter A. Buhr
// Created On : Sun Feb 9 20:34:28 1992
// Last Modified By : Peter Buhr
// Last Modified On : Sat Jul 30 08:55:10 1994
// Update Count : 148
//


#ifndef __U_NUMSPLR_H__
#define __U_NUMSPLR_H__


#include "uBaseSplr.h"
#include "uNumGauge.h"


// T must have operations: 0, =, >, <, +, /

template<class T> class uNumSampler : public uBaseSampler {
    const T &locn;
    int count, firstMin, firstMax;
    T curr, min, avg, max, sum;
    uNumGauge<T> numgauge;                              // displayer object
  public:
    uNumSampler( uWatcher &w, const int pollFreq, const int displayFreq, const T &locn, const char *name,
            const T dialMin, const T dialMax, const T zero, uNumGaugeTL::GaugeType gauge = uNumGaugeTL::digital ) :
            uBaseSampler( w, pollFreq, displayFreq ), locn( locn ), numgauge( name, gauge, dialMin, dialMax ) {
        firstMax = firstMin = 1;
        sum = zero;
        count = 0;

        // The sampler must be added to the watcher′ s event list at the end of the derived
        // sampler because the moment it is added to the event list, the poll and display
        // routine can be invoked by the watcher; therefore, the derived class must be
        // completely initialized. Furthermore, the sampler cannot be added to the watcher′ s
        // event list in the base sampler′ s constructor because the base sampler′ s poll and display
        // routine would be invoked instead of the derived sampler′ s routines, which is incorrect.

        watcher.add( *this );                         // add sampler to watcher′ s event list
    } // uNumSampler<T>::uNumSampler

    ~uNumSampler() {
        watcher.remove( *this );                      // remove sampler from watcher′ s event list
    } // uNumSampler<T>::~uNumSampler

    void poll() {
        curr = locn;                                  // get current value

        if ( firstMin ) {                             // maintain minimum value
            min = curr;
            firstMin = 0;
        } else {
            if ( curr < min ) {
                min = curr;
            } // if
        } // if

        count += 1;                                   // calculate running average
        sum += curr;
        avg = sum / count;

        if ( firstMax ) {                             // maintain maximum value
            max = curr;
            firstMax = 0;
        } else {
            if ( curr > max ) {
                max = curr;
            } // if
        } // if
    } // uNumSampler<T>::poll

    void display() {
        numgauge.display( curr, min, avg, max );
    } // uNumSampler<T>::display
}; // uNumSampler<T>


#endif __U_NUMSPLR_H__


// Local Variables: //
// compile-command: "dmake" //
// End: //
```

Appendix B

Visual Bounded Buffer Widget

```
// -*- Mode: C++ -*-
//
// Visualization 1.0, Copyright (C) Peter A. Buhr 1992
//
// uProdConsWD.h –
//
// Author : Peter A. Buhr
// Created On : Tue Sep 29 13:30:45 1992
// Last Modified By : Peter A. Buhr
// Last Modified On : Sat Jul 22 16:48:08 1995
// Update Count : 39
//


#ifndef __U_PRODCONSWD_H__
#define __U_PRODCONSWD_H__


#include <uXmlib.h>
#include "uXtShellServer.h"


class uProdConsSampler;                                        // forward declaration


// The widget structure for a ProdConsWidget:
//
// * Top-level Shell
// * Main Window Widget
// * Menubar Widget
// * Cascade Widget (quit)
// * Form Widget
// * Label Widget (Producer)
// * Separator Widget
// * Scale Widget (Producer Average)
// * Separator Widget
// * Scale Widget (Producer Std Dev)
// * Separator Widget
// * Label Widget (Delay)
// * Separator Widget
// * Scale Widget (Consumer Std Dev)
// * Separator Widget
// * Scale Widget (Consumer Average)
// * Separator Widget
// * Label Widget (Consumer)


class uProdConsWidget {
    static void QuitCB( Widget widget, uProdConsWidget *This, XmAnyCallbackStruct *call_data );
    static void pollingFreqCB( Widget widget, uProdConsWidget *This, XmScaleCallbackStruct *call_data );
    static void displayFreqCB( Widget widget, uProdConsWidget *This, XmScaleCallbackStruct *call_data );
    static void prodStdWDCB( Widget widget, uProdConsWidget *This, XmScaleCallbackStruct *call_data );
    static void prodAvgWDCB( Widget widget, uProdConsWidget *This, XmScaleCallbackStruct *call_data );
    static void consStdWDCB( Widget widget, uProdConsWidget *This, XmScaleCallbackStruct *call_data );
    static void consAvgWDCB( Widget widget, uProdConsWidget *This, XmScaleCallbackStruct *call_data );

    uProdConsSampler &prodconsSampler;
    Widget shell, pollingFreqWD, displayFreqWD, bufScaleWD;
    const int NumDecPts = 2;
    float prodStd, prodAvg, consStd, consAvg;
  public:
    uProdConsWidget( uProdConsSampler &prodconsSampler, const char *title, int bufSize );
    virtual ~uProdConsWidget();

    float getProdStd() {
        return prodStd;
    } // uProdConsWidget::getProdStd

    float getProdAvg() {
        return prodAvg;
    } // uProdConsWidget::getProdAvg

    float getConsStd() {
        return consStd;
    } // uProdConsWidget::getConsStd

    float getConsAvg() {
        return consAvg;
    } // uProdConsWidget::getConsAvg

    void setValue( int val );
}; // uProdConsWidget


#endif __U_PRODCONSWD_H__


// Local Variables: //
// compile-command: "dmake" //
// End: //
```

Bibliography

[BS95] Peter A. Buhr and Richard A. Stroobosscher. μC++ Annotated Reference Manual, Version 4.4. Technical Report Unnumbered: Available via ftp from plg.uwaterloo.ca in pub/uSystem/uC++.ps.gz, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, September 1995.

[CW94] P. Cheung and P. Willard. *XXGDB – X Window System Interface to the GDB Debugger*, November 1994. Distributed with XXGDB.

[GKM82] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a Call Graph Execution Profiler. *SIGPLAN Notices*, 17(6):120–126, June 1982. Proceedings of the SIGPLAN'82 Symposium on Compiler Construction, June 23–25, 1982, Boston, Massachusetts, U.S.A.

[HN80] A. N. Habermann and I. R. Nassi. Efficient Implementation of Ada Tasks. Technical Report CMU-CS-80-103, Carnegie-Mellon University, 1980.

[Kar95] Martin Karsten. A Multi-Threaded Debugger for Multi-Threaded Applications. Diplomarbeit, Universität Mannheim, Mannheim, Deutschland, August 1995.

[SP95] Richard M. Stallman and Roland H. Pesch. *Debugging with GDB*. Free Software Foundation, 675 Massachusetts Avenue, Cambridge, MA 02139 USA, 1995.

[Tal92] David Talyor. A Prototype Debugger for Hermes. In *CASCON'92*. IBM, November 1992.

[Tuf83] Edward R. Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 1983.

Index