# KDB: Concurrent Debugger

# Reference Manual

## Version 1.1

## University of Waterloo

Peter A. Buhr, Martin Karsten, Jun Shih and Oliver Schuster


July 26, 2000

# Contents

# 1 Introduction

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 a 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 [MH89]. A **symbolic debugger** provides the additional capability to refer to data using variable names, and to code using file names and statement numbers from the original source program.

When debugging a **sequential program**, the process is synchronous between the debugger and the application, i.e., when the program is running the debugger is not running, and vice versa. However, when debugging a **concurrent program**, the process is asynchronous between the debugger and the threads running the application, i.e., when the debugger is running some or all of the application may continue to run. As a result, the control and manipulation mechanisms available for sequential debugging must be provided independently for every thread of control, and addition operations are needed to manage and manipulate multiple threads. Unfortunately, most debuggers do not work with concurrent programming languages or environments, or treat the concurrent environment synchronously.

Of those that do deal with concurrency, most work only with **kernel threads** provided by the operating system. Kernel threads are controlled and scheduled by the operating system, not by the runtime environment of the application using them. While kernel threads are essential, so too are **user threads** [ABLL92], which subdivide a kernel thread's execution among user threads in an application. User threads have the potential to be significantly less expensive than kernel threads in many cases because the language runtime system has specific knowledge about the form of concurrency and its implementation.

Given that user threads are important, some mechanism must exist to debug concurrent programs using them. The reason is straightforward: each language and/or thread library is different, and hence, each requires individual debugging support. The KDB debugger [BKS96] shows that it is possible to build very powerful and flexible debugging support for user threads.

Finally, debugging affects the execution of a concurrent program, called the **probe effect** [Gai86]. This influence is partly determined by the commands to the debugger while debugging the application. Therefore, users must be aware that a concurrent program may execute quite differently when being debugged.

# 2 KDB

KDB (Kalli's DeBugger) [Kar95, Shi96] is a multithreaded debugger for debugging multithreaded $\mu$C++ applications. $\mu$C++ [BS99] is an extended version of C++ providing light-weight tasking facilities, i.e., user-level threads, using a shared-memory model. Both uniprocessor and multiprocessor $\mu$C++ 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.

# 3 Before Starting KDB

In order to debug a $\mu$C++ program with KDB, *all* program components should be compiled with the compilation flag -debug and -g on the u++ command line; -debug is the $\mu$C++ default, so normally only -g has to be specified. The -debug flag inserts additional code into the application to interact with the debugger, and the -g flag inserts symbol table information needed for symbolic debugging. If a program compiled *without* -debug is run under KDB, the debugger display simply remains blank. If a program compiled *without* -g is run under KDB, the debugger can provide only very limited debugging capabilities.

# 4 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`

A symbolic link or alias to /u/usystem/software/MVD/bin is insufficient because the bin directory contains several other executables needed to run KDB.

## 5 User Interface

The user 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 sash line with a sash button on the right-hand side. By dragging on the sash button, it is possible to adjust the vertical size of a pane within a window. At times, a button may appear "greyed out", which means the button is **inactive** (insensitive), so pressing it does nothing (see Figure 2(a), where New Target is inactive). Inactive buttons are meaningless in that particular context or mode; a button becomes **active** (sensitive) again when an appropriate mode change occurs.

## 6 Starting KDB

KDB operates as a server at which an application registers as a client (multiple clients are allowed). Figure 1 shows this relationship, and some of the internal structure of KDB and the application. Both server and client can be started together using the shell command:

% kdb *application-name [application-argument-list]*

The debugger's **main window** appears when the debugger starts (see Figure 2(a)) and is discussed in Section 9. Connecting the client (application) to an already running server (KDB) is discussed in Section 8. Connecting the server (KDB) to an already running client (application) is discussed in Section 9.4.

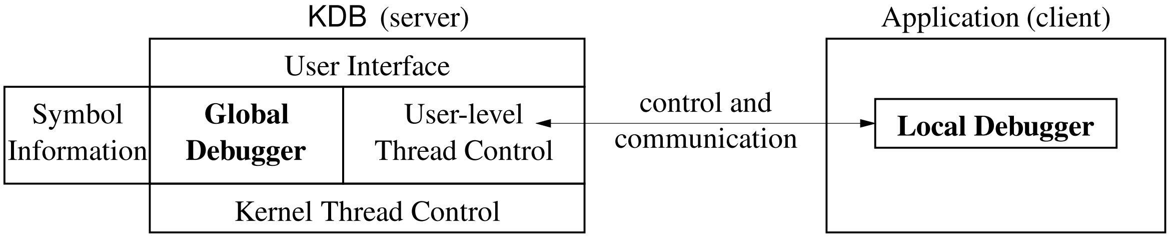



Figure 1: Debugging Structure

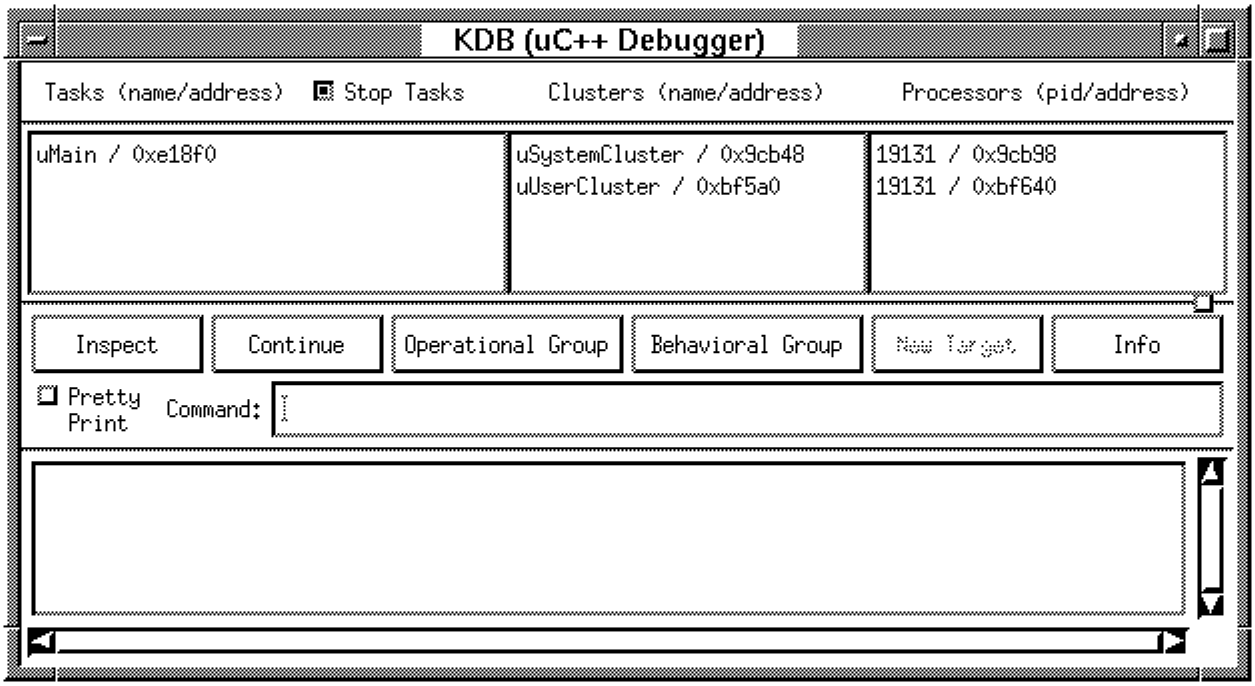
## 7 Terminating KDB

To terminate KDB, close the main window 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 closes a window, and when KDB's main window is closed, it terminates. Alternatively, typing <Ctrl-C> anywhere in the main window terminates KDB. 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 or by typing <Ctrl-C> anywhere in the window to remove them from the display without terminating KDB.

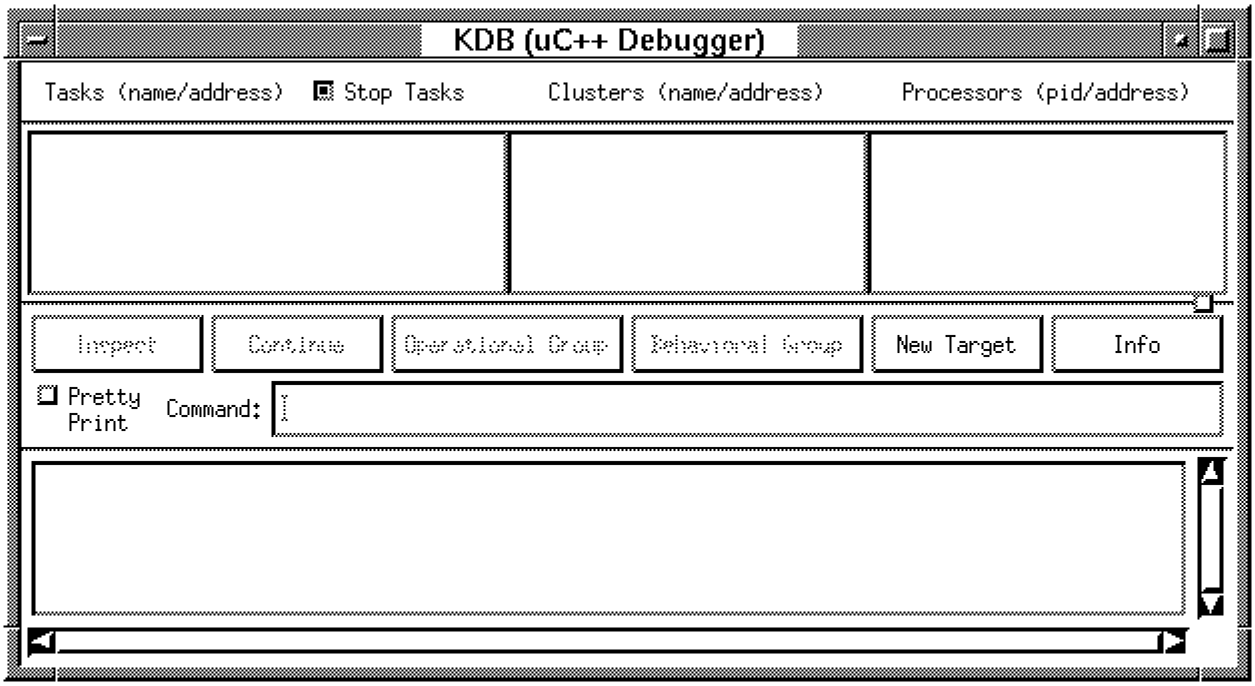
## 8 Reusing KDB

After an application finishes execution, KDB is still running and can proceed with a new debugging session for the same or a 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 2(b). The New Target button is now active, and correspondingly, the buttons dealing with the application are inactive. To start debugging a new application, a user must do *both* of the following in *either* order:




(a) Startup




(b) Termination

Figure 2: KDB main window

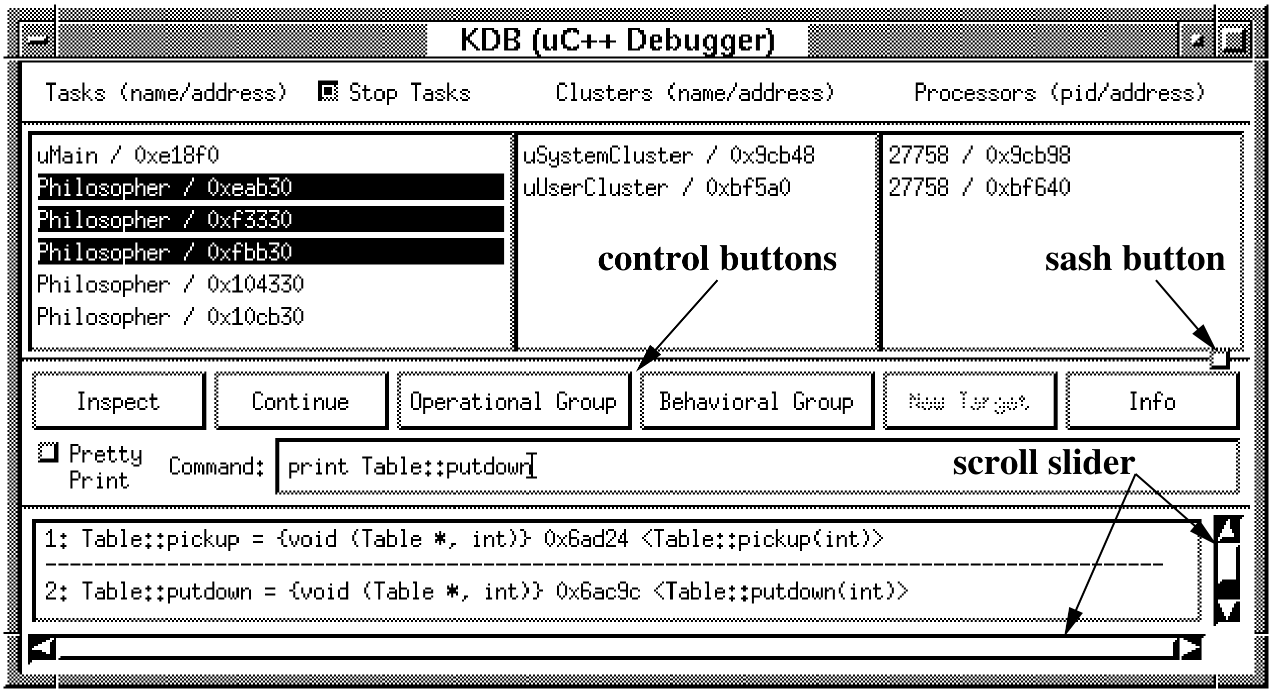



Figure 3: Main Window Showing Symbol Lookup

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

2. connect another application to KDB using the shell command:

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

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

## 9 Main Window

When the debugger starts, the main window appears and has 2 panes (see Figure 3).

1. The upper pane contains dynamic lists of tasks, clusters and processors[1] currently active in the application.

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

The two panes are separated by a **sash line**; there is a **sash button** on the right-hand side of the pane separator (labelled in Figure 3). By moving the mouse cursor over the sash button and pressing the left mouse button, the sash line can be dragged up or down, changing the size of the upper and lower panes. When the mouse button is released, the information is each pane is adjusted to the new pane size; however, there is a minimum pane size. In some cases, **scroll bars** appear for lists that can no longer be completely displayed. For example, the bottom window of the lower pane has scroll bars for both horizontal and vertical control. By moving the mouse cursor over the scroll slider (labelled in Figure 3) and pressing the left mouse button, it is possible to drag the scroll slider, changing the text displayed in the window associated with the scroll bar.

[1] Clusters and processors are artifacts of $\mu$C++ and not discussed here. See the *$\mu$C++ Annotated Reference Manual* [BS99] for details.

### 9.1 Tasks list

The left column of the upper pane is a list of all tasks currently active in the $\mu$C++ application. A task is identified by its name and memory address in the target application. The default task name in $\mu$C++ is the type name for a task. While $\mu$C++ allows a user-specified name for a task to be set at any time, the debugger only uses the type name given when a task is created.

When debugging, the default action for each task in the application is to stop at the beginning of its main member routine, unless the Stop Tasks button is toggled, where the Stop Tasks button is located on the first line of the upper pane. When Stop Tasks is off, newly created tasks continue execution *immediately*.

It is possible to select one or more tasks in the task list by moving the mouse cursor over a task, pressing the left mouse button, possibly dragging to select a group of tasks, and then releasing the mouse button. The selected tasks are highlighted (reverse video) to confirm the selection. For example, the three highlighted Philosopher tasks in Figure 3 form a group. To select a large contiguous set of tasks, especially when tasks are scrolled off the list, select the start task by moving the mouse cursor over a task, pressing the left mouse button, and then releasing the mouse button. Then move to the end task, possibly by scrolling with the slider, hold down the <Shift> key, and select the end task in the same way as the start task. All tasks between the start and end task become selected. To select a non-contiguous set of tasks, hold down the <Ctrl> key during the selection, make any number of selections by clicking and dragging, and raise the <Ctrl> key to end the selection. As long as the <Shift> or <Ctrl> key are used during selection, the selections accumulate, so it is possible to perform a series of contiguous and non-contiguous selections to form the desired group of tasks.

### 9.2 Clusters list

The centre column of the upper pane is a list of all clusters currently active in the $\mu$C++ application. A cluster is identified by its name and memory address in the target application. The default cluster name in $\mu$C++ is the type name for a cluster.

There is one operation available for clusters. By clicking on an entry in the cluster list, a window pops up (see Figure 4) 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.

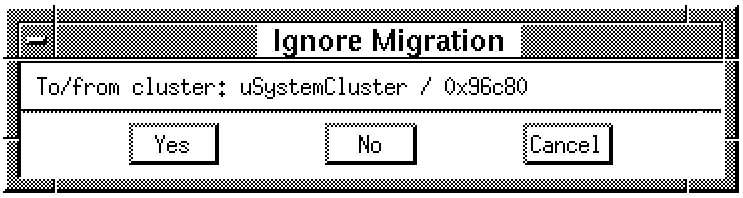



Figure 4: Task Migration Dialog

Because every migration of a task involves additional overhead during debugging, there can be situations where ignoring migration improves performance substantially. However, this mechanism should be used only by experienced $\mu$C++ programmers, since the debugger and application are no longer coordinated, i.e., what appears on the debugger display may not reflect the actual state of the application.

### 9.3 Processors list

The right column of the upper pane is a list of all processors (UNIX processes) currently active in the $\mu$C++ application. A processor is identified by its UNIX process id and memory address in the application. In Figure 3, both processors have the same UNIX process id, which means the $\mu$C++ application is running uniprocessor.

The only operation available for processors is a mechanism to ignore migration of processors among clusters. As for clusters, this is an insecure optimization mechanism that should only be used by experienced $\mu$C++ programmers.

### 9.4 Control Buttons

The **control buttons** (labelled in Figure 3) are the set of buttons at the top of the lower pane:

**Inspect button** Individual task information and control is provided through a task window (see Section 10), which is created by selecting a task(s) in the main window and typing <Return> or clicking on Inspect. Double-clicking on a single entry of the task list does the same as selecting a task and clicking on Inspect.

**Continue button** 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.

This operation is the only group operation directly accessible from the main window because it is commonly used. All other group operations are provided via the Operational and Behavioural buttons.

**Operational and Behavioural buttons** Group operations on the selected tasks are accessed through the popup group-dialogs available from buttons Operational Group and Behavioural Group. Both are discussed in detail in Section 11.

**New Target button** The New Target button is discussed in Section 8.

**Info button** The Info button pops up a copyright notice.

**Pretty Print button** If the Pretty Print button is toggled, data is printed in an easier to read way but results in longer output.

**Command area** 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 an executable that is already running.

Positioning the cursor in the command box makes it active so a command can be entered; typing <Return> executes the command. The result of the command appears in the command output-area of the lower pane. Each output in the command output-area 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 command output-area.

## 10 Task Window

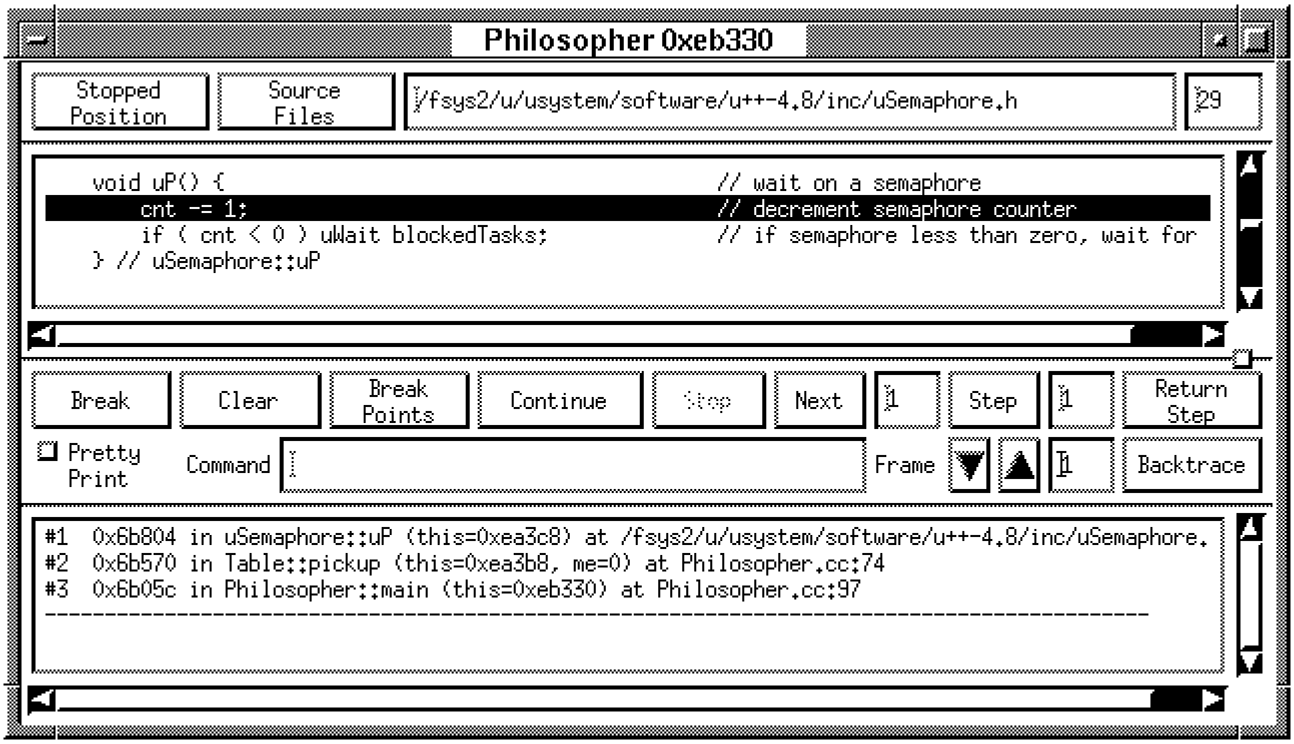
A **task window**[2] appears when a task is inspected from the main window or a breakpoint is encountered for a task; it has 2 panes (see examples in Figure 5).

1. The upper pane displays information about source code. The highlighted line (reverse video) marks the current statement about to be executed when stepping through a program, or by clicking on a line, that highlighted line becomes the operand for a command like setting or clearing a breakpoint.
2. The lower pane contains controls for sequential debugging of a thread and an output area for the results of certain operations.

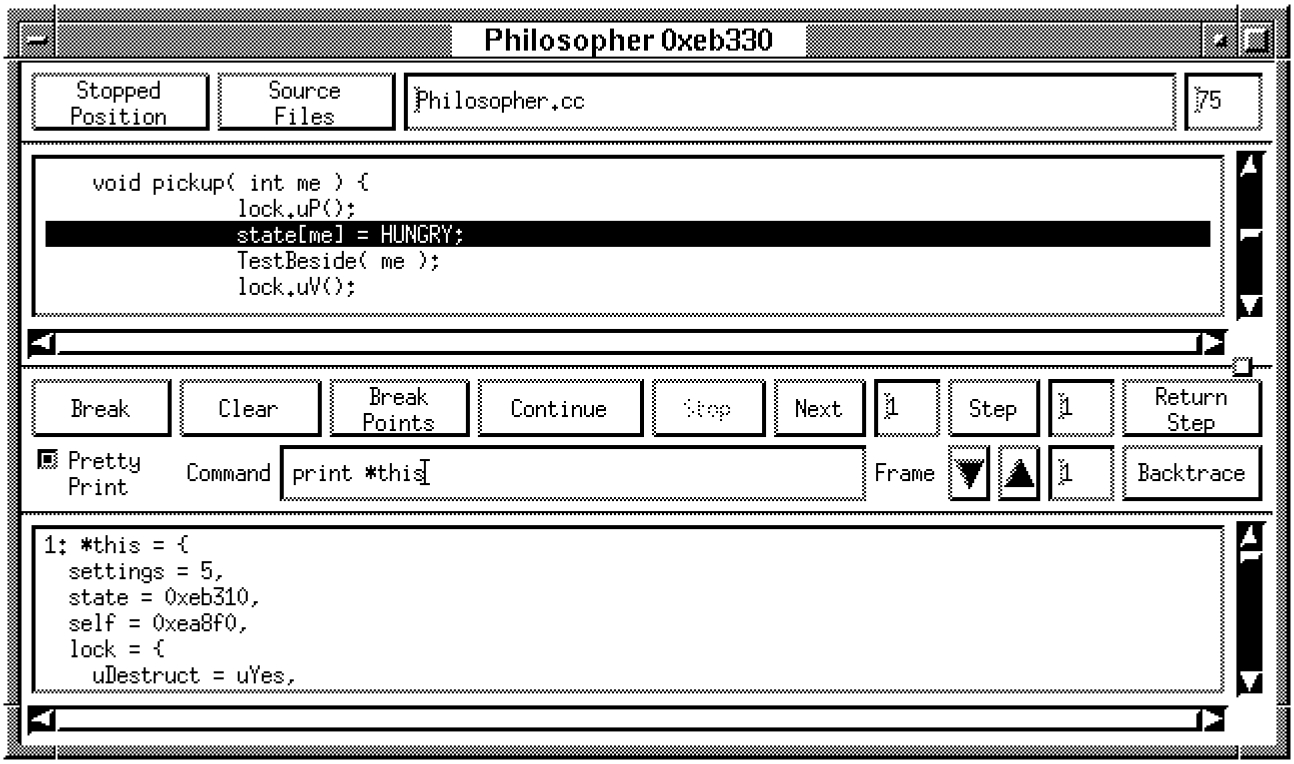
A task window begins by showing the last known position of the corresponding task by presenting the source file-name and line number on the first line of the upper pane, and displaying that source-code line in the source-code area with the particular line highlighted. It is possible to look at any part of the file in the source-code area by scrolling using the scroll bars on the right and bottom, or by entering a number into the line-number field on the first line and typing <Return>. It is possible to look at any file by entering a file name into the source-file field on the first line and typing <Return>. Clicking the Source Files button pops up a list of all source files for a program (see Figure 6). To select a file for display in the source-code area, either double-click on the name or click on both the name and OK. Finally, clicking the Stopped Position button returns to the file and line number of the last known execution position.

□ While it is possible to view different source files when a task is running, as soon as the task encounters a breakpoint, the source-code area of the upper pane changes immediately to show the breakpoint location. □

[2] This window is similar to the main debugging window of xxgdb [CW94].




(a) Backtrace




(b) Symbol Lookup

Figure 5: Task Windows

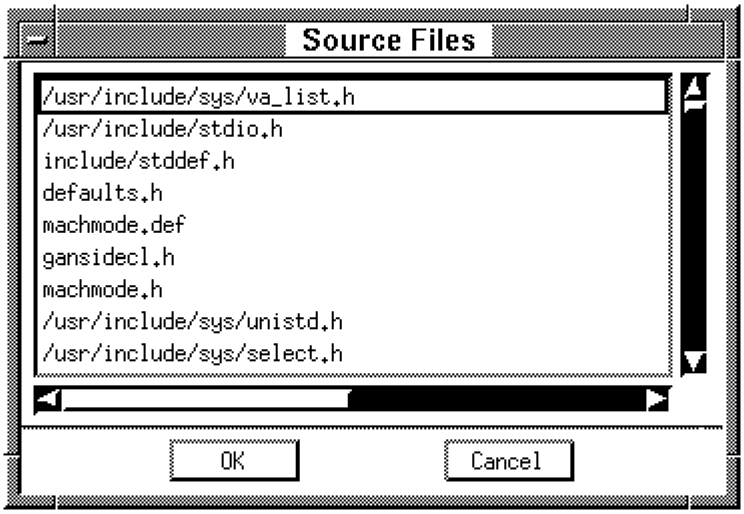



Figure 6: Source File List

### 10.1 Control Buttons

The **control buttons** are the set of buttons at the top of the lower pane (see Figure 5), and 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 only block if they become **application blocked** on resources that are held directly or indirectly by debugger blocked tasks.

**Break button** To add a breakpoint, click on the desired line of code, which becomes highlighted, and then click on Break.

**Clear button** To remove a breakpoint, click on the desired line of code, which becomes highlighted, and then click on Clear. If there is no breakpoint set at the line, a warning-beep sounds. If the line is already highlighted, just click on Clear.

**Break Points button** The Break Points button pops up a list of all breakpoints currently set for this task (see Figure 7). Breakpoints can be cleared by selecting one or more breakpoints from the list and then pressing the Clear button. (See section 9.1 for details on selecting a contiguous and/or non-contiguous group of items from a list.) It is possible to view the source-code location of a breakpoint by double-clicking on the breakpoint in the list. The source-code area of the upper pane in the task window changes to display the line of code containing the selected breakpoint.

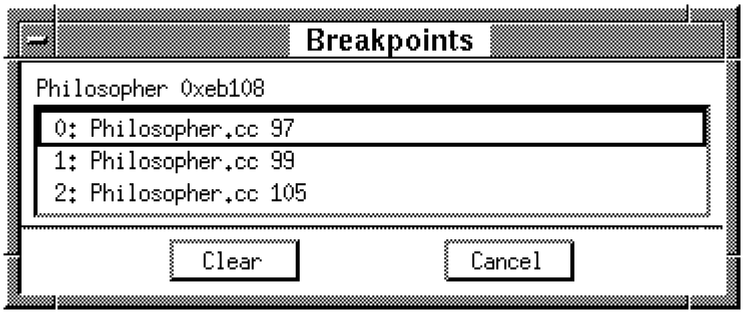



Figure 7: Breakpoint List

**Continue button** Clicking the Continue button resumes execution of the task.

**Stop button** Clicking the Stop button stops the task at the next possible location. If a task is currently blocked in the application, e.g., if a task is waiting on the entry queue of a mutex object, or executing in a system routine, e.g., reading or writing to a file, the stop request does not take effect until the task eventually becomes active again or returns to user code.

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

The $\mu$C++ translator inserts code into multiple locations in a $\mu$C++ application. In general, KDB is aware of all inserted code and automatically ignores it during debugging. However, there is one case where hiding could not be accomplished. $\mu$C++ creates a constructor and destructor for all mutex objects. There is no way for KDB to tell if a constructor or destructor is $\mu$C++ or user generated. Therefore, a Step operation at the declaration or termination of a mutex object always transfers to the object. If there is no user supplied constructor or destructor, this action seems peculiar. Performing another Step operation returns back to the previous point of execution.

Just as KDB is aware of $\mu$C++ inserted code, it is also aware of new $\mu$C++ forms of control flow. In particular, a Step operation at a **uResume** or **uSuspend** statement steps from one execution-state to another, just as a Step operation at a call or **return** statement steps into a routine or back to a caller.

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

**Pretty Print button** If the Pretty Print button is toggled, as in Figure 5(b), complex data is printed in an easier to read structured way but results in significantly longer output in the command output-area.

**Command area** The Command area is for typing in the following commands:

- [ print | p ] *expression*: print local variable names or expressions.
- [ break | b ] [ [ *source-file* ]:*line-no* | *function-name* ] [ if *simple-expression* ]: add a breakpoint at specified line in source file or at start of function.

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

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

  where the variable forms allowed are: V, *V, V->F and V.F
- [ clear | c ] [ [ *source-file* ]:*line-no* | *function-name* ]: remove a breakpoint at specified line in source file or at start function.

Positioning the cursor in the command box makes it active so a command can be entered; typing <Return> executes the command. The result of the command appears in the command output-area of the lower pane. Each output in the command output-area 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 command output-area.

For all commands, variable names are evaluated in the scope of the current stack location for the specific task associated with the window. For example, in Figure 5(b), the print command of variable **this** refers to the object for which the class method pickup is invoked.

**Frame/Backtrace buttons** The Backtrace button produces a backtrace of the calling stack for a task, which is shown in the command output-area of the lower pane (see Figure 5(a)). The arrow buttons to the left 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 source-code area of the upper pane. The frame position is displayed beside the arrow buttons; it is possible to go directly to a particular frame by entering the frame number in this field and typing Return. While moving up and down the calling stack, routines may be encountered that are compiled *without* -g, e.g., system libraries, UNIX signals, $\mu$C++ kernel. In this case, a message appears in the source-code area of the upper pane stating this fact; keep pressing the Frame arrows until routines are found that are compiled with the -g flag.

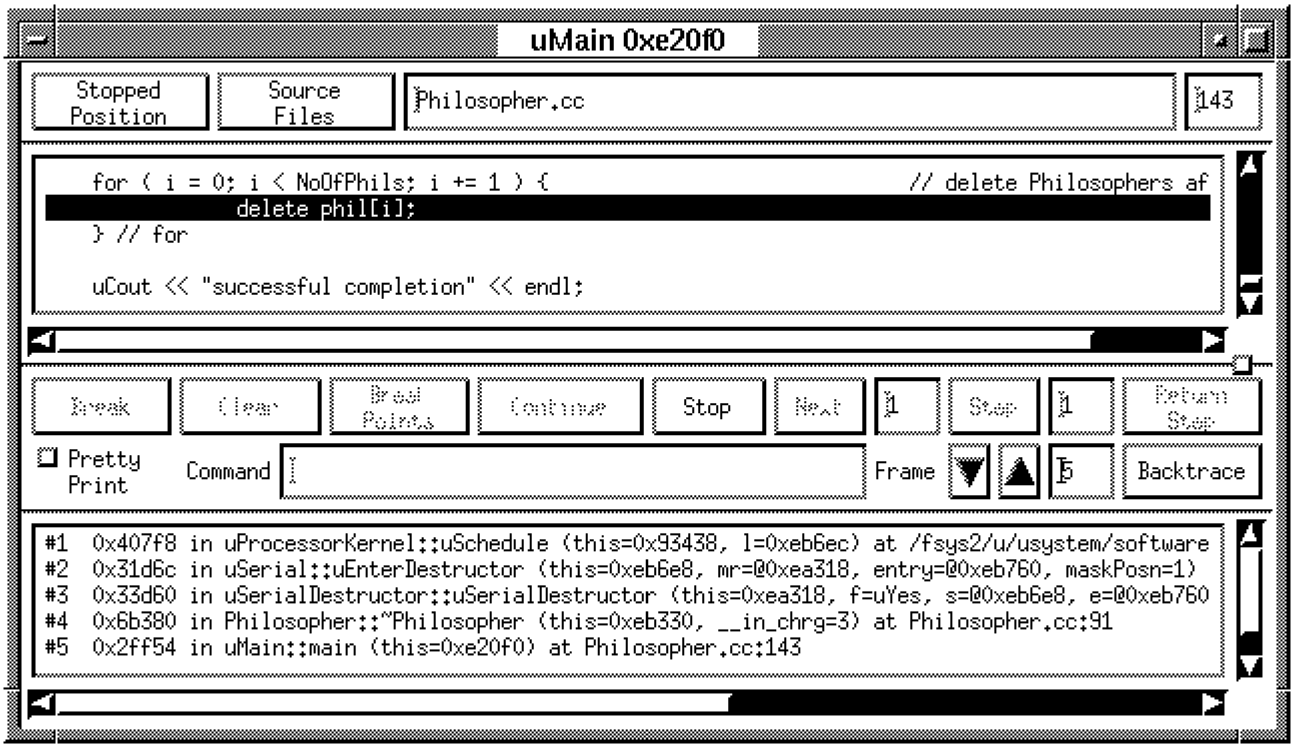



Figure 8: Task Window Showing Running Task

### 10.2 Click to Print

One of the most common operations performed during debugging is printing the value of a variable; therefore, this operation is optimized so that text can be selected directly from the source-code area of the upper pane. There are two ways to select text for printing:

**Double clicking** To select simple variable names and expressions, position the mouse cursor over the desired block of text and double click with the left mouse button. KDB attempts to select the *longest* C++-style variable/expression *not separated by whitespace*. For example, double clicking anywhere over the text "state[ me ]" selects the entire string.

**Click and drag** To select text that contains whitespace or less text than selected with a double-click, position the mouse cursor at the start of the text, press the left mouse button down, drag the cursor to the end of the text, and release the left mouse button. For example, positioning the mouse cursor at the start of the text "a + b / c", pressing the left mouse button down, and dragging to the end of the text, selects the entire string.

A print command with the selected text is then inserted into the command prompt verifying the command executed. This command can be subsequently edited in the command area, which is useful for printing slight variants of selected text, e.g., to print *phil[ i ] after printing phil[ i ].

### 10.3 Examining a Running Task

When a task is running, the task-window mode changes to the state shown in Figure 8: only the Stop and Backtrace buttons are active. Clicking the Stop button stops the task at the next possible location, which may not occur immediately.

It is possible to monitor the execution of a running task by clicking on the Backtrace button, which generates a snapshot of the current execution stack for a task (see lower pane of Figure 8). For example, if an application terminates with an error, such as a deadlock or segment fault, $\mu$C++ prints an appropriate error message in the shell where the application is connected to the debugger, such as:

> uC++ Runtime error (UNIX pid:12768) : no ready or pending tasks. Possible cause is tasks in a synchronization or mutual exclusion deadlock. Error occurred while executing task 0xbf500 (uSystemTask).

and informs the debugger of the error. The debugger stops all tasks in the application (but not instantly in multiprocessor applications) and a window pops up (see Figure 9) to control when the application terminates and possibly produces a core file. Before pressing OK, KDB can be used to examine all of the tasks in the application. In particular, walking the stack of some or all tasks provides detailed information about the state of the application leading to the error.


Figure 9: Application Abort Dialog

However, for a deadlock, tasks are still running from the debugger's perspective, but blocked from the thread system's perspective, which means stack walking is disabled, i.e., the frame arrows are disabled. *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.** For the deadlock situation, all of the tasks are application blocked, and hence, none can continue execution. If a task should begin execution again while pressing the frame arrows, it may cause incorrect information to be printed or even cause the debugger to fail. **Be careful when using this feature.** Finally, after examination of the application is complete, clicking OK in the abort window to release the application or it can terminate.

## 11 Thread Groups

Different kinds of possibly overlapping task-groups can be formed. This capability is an important feature when scaling to medium or large numbers of threads. Once a group of tasks is formed, an operation can be applied to all members of the group. Instead of interacting with a large number of individual threads, a user can interact with a small number of groups. However, grouping tasks does not affect the ability to control tasks individually; furthermore, a task's reactions to the commands issued to a group, such as setting a breakpoint or continuing execution, become visible in each task's window.

### 11.1 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. However, certain commands may only be meaningful to all tasks in the group if they all execute the same source code. For instance, setting a common breakpoint is meaningless for threads that do not execute common code. When an operational group window is closed, that group is terminated.

Clicking the Operational Group button forms a group of all tasks currently selected in the main window (see selected Philosopher tasks in Figure 3), and pops up a window (see Figure 10) 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] [ [*source-file* ]:*line-no* | *function-name*] [if *simple-expression*]: add a breakpoint at specified line in source file or at start of function.
- [clear | c] [ [*source-file* ]:*line-no* | *function-name*]: remove a breakpoint at specified line in source file or at start function.
- stop: stop task execution

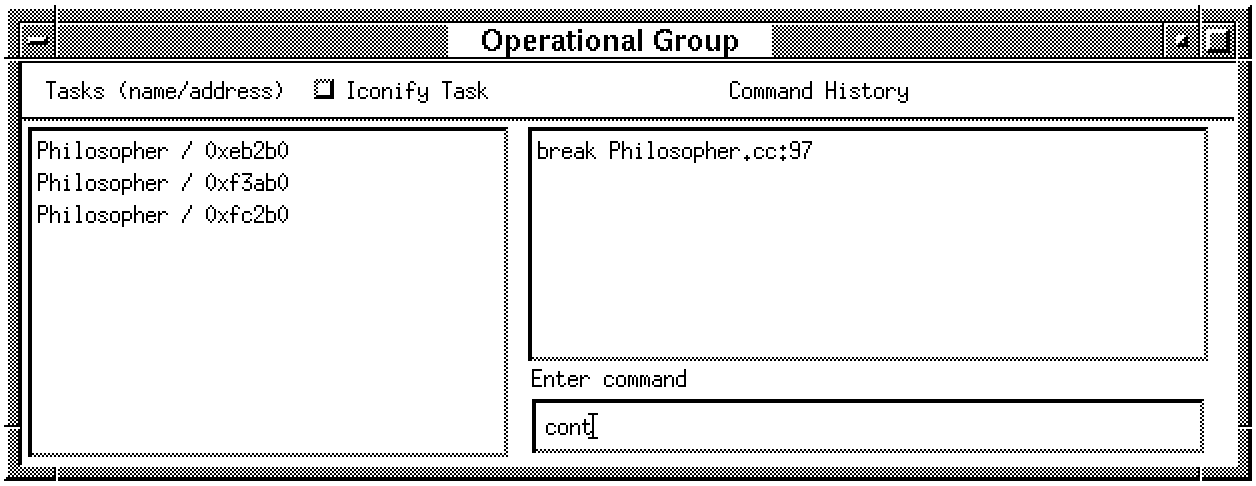



Figure 10: Operational Group Window

- 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

Positioning the cursor in the command box makes it active so a command can be entered; typing <Return> executes the command. 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 command area, where it can be edited if necessary before executing. Figure 10 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.

### 11.2 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. For example, when one 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, such as Stop and Continue operation.

Clicking the Behavioural Group button forms a group of all tasks currently selected in the main window (see selected Philosopher tasks in Figure 3), and pops up a window where an event and action can be issued for all tasks that belong to the group (see Figure 11). The following commands can be entered in the event window:

- [ break | b ] [ [ *source-file* ]:*line-no* | *function-name* ] [ if *simple-expression* ]: add a breakpoint at specified line in source file or at start of function.

The following commands can be entered in the operation window:

- stop: stop task execution

Figure 11 shows a behavioural group window where a breakpoint is about to be set for each philosopher task, and each task in the group is stopped when one of the tasks reaches the breakpoint.

## 12 Reducing Window Clutter

When debugging a large number of threads, a computer screen does not have enough space to show many thread windows, especially when working with operational/behavioural groups with a large number of threads. When a

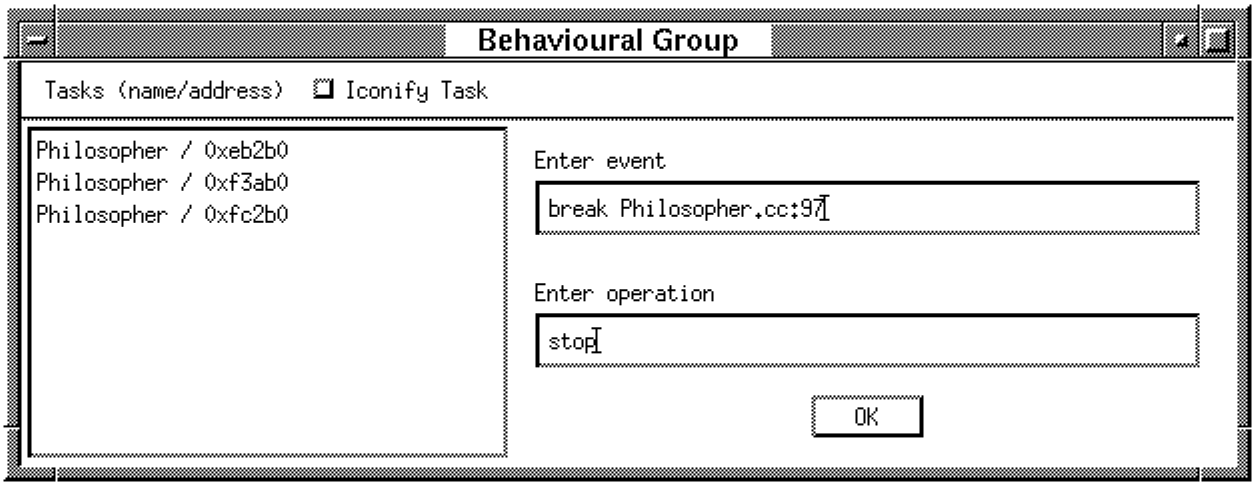



Figure 11: Behavioural Group Window

breakpoint is encountered in a thread, the default action is to create a thread window on the screen. This semantics can be very inconvenient if a breakpoint hit occurs in multiple threads within a group, causing multiple windows to pop-up and flood the screen.

To partially solve this problem, the thread windows of a group can be created in iconic state, if one does not already exist, by toggling the Iconify Task button in the group window.

## 13 Programmatic Interface

KDB provides a programmatic interface, which allows it to be operated by commands sent through a communication channel rather than the graphical user interface presented earlier. The purpose of the programmatic interface is to allow KDB to be used by other tools as part of a more sophisticated concurrent debugging and analysis environment. The communication channel is an INET socket so that remote (distributed) interaction is possible. The protocol is a simple machine level protocol for controlling KDB and obtaining output.

Two additional command line options are introduced:

**-api** enables the programmatic interface, an INET socket is created, and the socket port number is printed on the terminal.

**-nointerface** disables the Motif debugger interface so KDB is started without any window interface.

To use the programmatic interface in a C++ program, include the file:

**#include** <uDebuggerAPI.h>

at the beginning of each source file. This file contains constant and type declarations to construct communication messages (see Figure 12 and the next section).

### 13.1 Supported Messages

The following messages are supported by the programmatic interface, where each message name has a corresponding **#define** in the programmatic interface include file.

#### 13.1.1 BP_SET

Set a breakpoint in the specified user-thread. The message specifies the user thread id and the breakpoint location, which is parsed by KDB, e.g.,

- 82 *// breakpoint at line 82*
- Table::pickup *// breakpoint in the function pickup*

```
#define MAX_CMD_LEN      256
#define MAX_VAR_LEN      64
#define MAX_COND_LEN     128
#define MAX_PATH_LEN     256
#define MAX_PRINT_LEN    1024

typedef enum {
    BP_SET,
    BP_CLEAR,
    CONTINUE,
    STOP,
    PRINT,
    ATTACH,
    CLUSTER_LIST,
    PROCESSOR_LIST,
    THREAD_LIST,
    BP_HIT,
    PROGRAM_TERMINATED,
    TERMINATE,
} MessageType;

typedef int     NotifyMsg;
typedef void    *ThreadId;
typedef void    *ListId;
```

Figure 12: Programmatic Interface: Constants and Types

- Philosopher.cc:82 if k1 <= 10 *// breakpoint in file Philosopher.cc at line 82 if k1 <= 10*

The structure of the message from the controlling program to KDB is:

```
struct BP_SET_MSG {
    MessageType msg;
    ThreadId thread_id;
    char break_cmd[MAX_CMD_LEN + MAX_COND_LEN];
};
```

The structure of the notification message from KDB to the controlling program is:

```
struct GENERAL_NOTIFY {
    MessageType msg;
    NotifyMsg   nmsg;
};
```

nmsg is set to one of the following values:

0 ⇒ success
1 ⇒ thread does not exist
2 ⇒ thread is not in stopped state
3 ⇒ breakpoint command error
4 ⇒ other error

#### 13.1.2 BP_CLEAR

Clear a breakpoint in the specified user thread. The message specifies the user thread id and the breakpoint location, which is parsed by KDB. Clear commands are similar to breakpoint commands except there is no conditional clause. For example:

- 82 *// breakpoint at line 82*
- Table::pickup *// breakpoint in the function pickup*
- Philosopher.cc:82 *// breakpoint in file Philosopher.cc at line 82*

The structure of the message from the controlling program to KDB is:

```
struct BP_CLEAR_MSG {
    MessageType msg;
    ThreadId thread_id;
    char clear_cmd[MAX_CMD_LEN];
};
```

The structure of the notification message from KDB to the controlling program is:

```
struct GENERAL_NOTIFY {
    MessageType msg;
    NotifyMsg  nmsg;
};
```

nmsg is set to one of the following values:

0 ⇒ success
1 ⇒ thread does not exist
2 ⇒ thread is not in stopped state
3 ⇒ clear command error
4 ⇒ other error

### 13.1.3 CONTINUE

Continue a user thread, which may be previously stopped or has encountered a breakpoint. The message specifies the user-thread id of the user-thread to be continued. The structure of the message from the controlling program to KDB is:

```
struct CONTINUE_MSG {
    MessageType msg;
    ThreadId thread_id;
};
```

The structure of the notification message from KDB to the controlling program is:

```
struct GENERAL_NOTIFY {
    MessageType msg;
    NotifyMsg  nmsg;
};
```

nmsg is set to one of the following values:

0 ⇒ success
1 ⇒ thread does not exist
2 ⇒ thread is not in stopped state (thread is in running already)

### 13.1.4 STOP

Stop a specified user thread. The message specifies the user thread id to be stopped. The structure of the message from the controlling program to KDB is:

```
struct STOP_MSG {
    MessageType msg;
    ThreadId thread_id;
};
```

The structure of the notification message from KDB to the controlling program is:

```
struct GENERAL_NOTIFY {
    MessageType msg;
    NotifyMsg  nmsg;
};
```

nmsg is set to one of the following values:

0 ⇒ success

1 ⇒ thread does not exist
2 ⇒ thread is in stopped state (thread is stopped already)

### 13.1.5 PRINT

Print a variable in the specified user thread. The message specifies the user thread id and the variable name. The result of print is sent back to the controlling program. The structure of the message from the controlling program to KDB is:

```
struct PRINT_MSG {
    MessageType msg;
    ThreadId thread_id;
    char var_name[MAX_VAR_LEN];
};
```

The structure of the notification message from KDB to the controlling program is:

```
struct PRINT_NOTIFY {
    MessageType msg;
    NotifyMsg nmsg;
    char print[MAX_PRINT_LEN];
};
```

The raw output is put in field print. nmsg is set to one of the following values:

0 ⇒ success
1 ⇒ thread does not exist
2 ⇒ thread is not in stopped state

### 13.1.6 ATTACH

Attach the debugger to the specified process. The message specifies the process id and the relative or absolute path of the executable. The structure of the message is shown below:

```
struct ATTACH_MSG {
    MessageType msg;
    int  pid;
    char path[MAX_PATH_LEN];
};
```

The structure of the notification message from KDB to the controlling program is:

```
struct GENERAL_NOTIFY {
    MessageType msg;
    NotifyMsg  nmsg;
};
```

nmsg is set to one of the following values:

0 ⇒ success
1 ⇒ process does not exist
2 ⇒ executable file does not exist
3 ⇒ executable file is not compiled with -debug option

### 13.1.7 CLUSTER_LIST, THREAD_LIST, PROCESSOR_LIST

Send the current cluster list, thread list, or processor list, depending on the msg, to the controlling program. The structure of the message is shown below:

```
struct LIST_MSG {
    MessageType msg;
};
```

The structure of the notification message from KDB to the controlling program is:

```
struct LIST_NOTIFY {
    MessageType msg;
    NotifyMsg nmsg;
    ListId id;
};
```

Members of a list are sent one at a time. nmsg is set to one of the following values:

0 ⇒ current list requested is completely sent
1 ⇒ more list items are to be sent

### 13.1.8 BP_HIT

Notification only message. Notify the controlling program that a breakpoint is encountered in a thread. The structure of the message is shown below:

```
struct BP_HIT_NOTIFY {
    MessageType msg;
    int thread_id;
};
```

### 13.1.9 PROGRAM_TERMINATED

Notification only message. Notify the controlling program that the program has terminated. The structure of the message is shown below:

```
struct PROGRAM_TERMINATED_NOTIFY {
    MessageType msg;
};
```

### 13.1.10 TERMINATE

Close the connection. KDB removes any outstanding breakpoints from the application and terminates. At this point, the application continues execution normally. The structure of the message is shown below:

```
struct TERMINATE_MSG {
    MessageType msg;
};
```

There is no notification.

## 14 Contributors

While many people have made numerous suggestions, the following people were instrumental in turning this project from an idea into reality. Rory Jacobs [Jac95] wrote the prototype concurrent debugger for the $\mu$System [BMS94]. "Totally Cool" Martin Karsten designed and wrote version 1.0 of KDB for $\mu$C++, and Jun Shih extended it with many essential features. Oliver Schuster added support for replay [Sch99]. Peter Buhr did sundry coding as needed in versions 1.0, and rewrote much of the code for version 1.1.

## References

[ABLL92] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[BKS96] Peter A. Buhr, Martin Karsten, and Jun Shih. KDB: A Multi-threaded Debugger for Multi-threaded Applications. In *Proceedings of SPDT'96: SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 80–87, Philadelphia, Pennsylvania, U.S.A., May 1996. ACM Press.

[BMS94] Peter A. Buhr, Hamish I. Macdonald, and Richard A. Stroobosscher. $\mu$System Annotated Reference Manual, Version 4.4.3. Technical report, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, September 1994. ftp://plg.uwaterloo.ca/pub/uSystem/uSystem.ps.gz.

[BS99] Peter A. Buhr and Richard A. Stroobosscher. $\mu$C++ Annotated Reference Manual, Version 4.7. Technical report, Dept. of Computer Science, University of Waterloo, August 1999. ftp://plg.uwaterloo.ca/pub/-uSystem/uC++.ps.gz.

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

[Gai86] J. Gait. A Probe Effect in Concurrent Programs. *Software Practice and Experience*, 16(3):225–233, March 1986.

[Jac95] Rory Alan Jacobs. A Debugger for Multi-Threaded Applications. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, July 1995.

[Kar95] Martin Karsten. A Multi-Threaded Debugger for Multi-Threaded Applications. Diplomarbeit, Universität Mannheim, Mannheim, Deutschland, August 1995. ftp://plg.uwaterloo.ca/pub/MVD/-KarstenThesis.ps.gz.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging Concurrent Programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[Sch99] Oliver Schuster. Replay of Concurrent Shared-Memory Programs. Diplomarbeit, Universität Mannheim, Mannheim, Deutschland, April 1999. ftp://plg.uwaterloo.ca/pub/MVD/SchusterThesis.ps.gz.

[Shi96] Jun Shih. Debugging Concurrent Programs. Master's thesis, University of Waterloo, Waterloo, Ontario, Canada, N2L 3G1, December 1996. ftp://plg.uwaterloo.ca/pub/MVD/ShihThesis.ps.gz.

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

## Index