Contextual Debugging and Analysis of Multi–threaded Applications

M. Bednorz A. Gwozdowski K. Zieliński
Institute of Computer Science
University of Mining & Metallurgy
Al.Mickiewicza 30, 30-059 Kraków, Poland
kz@ics.agh.edu.pl

September 29, 1994

1 Introduction

The reasons for supporting multiple threads of control fall into two categories [2], those motivated by efficient utilization of multiprocessor hardware and those motivated by application concurrency. The both reasons justify an effort to develop multi–threaded software and put enough impact to force programmers to rewrite many existing applications [1, 2].

As is true of many system services today, the programmers view of the multiple threads of control service is not always identical to what the kernel implements. The software view is created by a combination of the kernel, run-time libraries, and the compilation system. This approach increases the portability of applications and systems, by hiding some details of implementation, while providing better performance, by allowing library code to do some work without involving the kernel. So the threads library level is the most interesting for an application programmer.

Multi–threaded programs are specially difficult to test and debug. They suffer from the same problems as concurrent and parallel programming paradigms. Multiplicity of threads, concurrency of their execution, synchronization and communication between them, make an identification of the program state very sophisticated. In this situation monitoring and tracing of the program behavior is a very complicated task. Usually an execution of one thread depends on the context or is related to the conditions manipulated by other threads. It is very difficult to follow a history of the program components, those related to programming with threads such as mutexes, condition variables, locks or semaphores usage. Unfortunately an access to modern analysis and debugging tools for multi–threaded applications is still very limited [12, 8].

The aim of this paper is to present a new concept of a multi–threaded programs analysis and debugging based on contextual visualisation of the program components that influence threads execution. For this purpose a dedicated software package called **MTV** (Multi-Threading Viewer) has been designed and implemented. It performs on the run-time library level, hence only a programmers' view of multiple threads control execution may be analyzed. The proposed idea is general and could work for any threads library [4, 8, 11] but the investigated in this paper version of software runs under Solaris 2.3 and exploits SunOS 5.3 Multi–thread Architecture. This architecture has been chosen not only for availability reason but it has modern two-level structure and well defined interface to user threads library.

The remainder of this paper is divided into five sections. The first gives a discussion of the major design assumptions and the model of activity represented by **MTV**. This covers a tested program code instrumentation, communication and synchronization between the instrumented program and **MTV**. In Section 2 general concept of contextual visualisation of multi–threaded program have been elaborated. MTV cooperation with monitored program implementation details have been discussed in Section 3. In Section 4 **MTV** user interface has been described. Next, in Section 5 representation of multi–threaded program state has been shown, capability of **MTV** for certain classes of error recognition has been specified and illustrated in more details by a few examples. These examples are not intended to be exhaustive, but they indicate the opportunities to exploit **MTV** for analysis of complex applications. Short evaluation of the proposed contextual visualisation techniques with application to multi–threaded programs analysis concludes the paper.

2 General concept of contextual visualization

There are a few related problems in the analysis of multi–threaded application behavior. The most fundamental one concerns a definition of the program state and representation of the execution history in the form suitable for correctness of the program analysis. It influences a choice of the program under test control technique and the method of collecting monitored information.

The proposed approach depends also on the general assumptions about a class of the investigated analysis tool such as:

1. Multi–threaded program should be analyzed on-line and the tool should provide mechanisms for its execution step by step, setting breakpoints or animation with the chosen speed.
2. Influence of the tool on a program under test behavior should be limited as much as possible. It means in particular that relative timings between the independent threads of execution should be preserved. This condition is crucial for correctness of concurrent systems activity analysis.
3. Code instrumentation should be performed automatically without any manual intervention of a programmer and should provide any relevant information about threads execution.
4. Information collected during a program execution should be presented in the form easy to analyse and the system should navigate its review in the most appropriate way.

The definition of the program state in this paper covers only its multi–threaded aspects. It is represented by five lists containing the following information about so called threads elements:

1. Threads of the program,
2. Mutual exclusion (mutex) locks,
3. Condition variables,
4. Counting semaphores,
5. Multiple readers, single writer (readers/writer) locks.

The last four lists correspond to the synchronization primitives typically available in many threads libraries, in particular they are provided by the Solaris 2.3 threads library. Each entry on these lists represents a record containing the whole accessible information about the relevant item and the context in which a given item exists.

For instance the record which describes a mutual exclusion lock contains: name of the source program file and line number where the lock was created, pointer to the lock, type of the lock, descriptor of the thread that created the lock, pointer to the list of threads that possess the lock, pointer to the list of threads that are suspended on this lock.

The idea of contextual analysis proposed in this paper may be shortly described as a process of automatic association of the given inspected item from one list with related items from remaining lists. The association process provides additional information difficult to find manually, and as will be shown in Section 5 makes if possible to identify many errors. This conclusion together with assumptions pointed out in the beginning of this section had a substantial impact on **MTV** architecture.

According to the assumption a multi–threaded program is monitored on-line during its execution. Taking into account the definition of multi–threaded program state, its changes may occur only when a call to the threads library function is performed. So the state transition is associated with the following events:

- thread creation,
- thread destruction,
- another call to the threads library function.

These events are reported by a special functions that are put into the monitored program code during so called instrumentation process. The instrumented monitored program and **MTV** are two separate processes. Interactions between these processes consist of the following general steps:

1. Inside the global monitor the instrumented code collects information about thread's function call.
2. The packet with the collected information is sent to **MTV**,
3. **MTV** freezes the monitored program and starts received information processing,
4. when **MTV** is ready for a new event processing it resumes the monitored program activity.

The program under test is "frozen" by **MTV** any time these events occur. It means that all threads in the program are stopped. When the program is frozen, **MTV** has time to do any necessary processing and visualization, without any risk that some threads are executed in parallel. This preserves the relative timing between threads in the tested process.

On multiprocessor more than one thread may call a thread library function in the same time. Here rises a general problem of visualization such truely parallel state. Because **MTV** is single threaded process any concurrent requests from tested program are, in fact, handled sequentially. To solve this problems out, messages are sent to **MTV** inside of global monitor. The sequential order of sent messages in that case is achieved. If it is possible, the thread function call is made also inside the global monitor to preserve the real process state seen by **MTV** (see next: type 2 of code instrumentation). In this case some of the threads library functions in a multi–threaded program can lose their ability of really parallel execution on multiprocessor architecture.

3 MTV cooperation with monitored program – implementation details

3.1 Code program instrumentation

The monitored program is instrumented by the preprocessor, which replaces each thread function call by the corresponding function from the **MTV** library.

Depending on the thread function type, the instrumentation function has one of the three possible structure shown in Table 1.

Type	Function structure

1

```
int MTV_thr_function(filename, line, arg1,...,argn);
{
  int err;
  Before_Thr_Function(filename, line, arg1,...,argn);
  err = thr_function(arg1,...,argn);
  After_Thr_Function_1(filename, line, arg1,...,argn);
  return(err)
}
```

2

```
int MTV_thr_function(filename, line, arg1,...,argn);
{
  int err;
  MT_VIEWER_LOCK;
  err = thr_function(arg1,...,argn);
  After_Thr_Function_2(filename, line, arg1,...,argn);
  KILL_MTV;
  STOP_SELF;
  MT_VIEWER_UNLOCK;
  return(err);
}
```

3

```
int MTV_thr_function(filename, line, arg1,...,argn);
{
  run_thread_arg_ptr_t run_thread_arg_ptr;
  Before_Thr_Exit(filename, line, arg1,...,argn);
  thr_getspecific( _MT_SPECIFICDATA_KEY, (void **)&run_thread_arg_ptr);
  free(run_thread_arg_ptr);
  thr_exit(status);
}
```

Table 1: Instrumentation function structures types

The most common situation is when a thread function call may suspend the calling thread. It means that the state before and after the function call is interesting. In that case the instrumentation function of structure number one is used. The body of this instrumentation function is built of three basic elements: the prologue section (**Before_Thr_Function()**), the thread function call (**thr_function**) and the epilogue section (**After_Thr_Function()**). In the prologue section all information about the thread just before call of a given thread function is collected. The epilogue plays a very similar role but it analyses the thread state just after the function call. A skeleton of the function **Before_Thr_Function()** may be specified as follows:

```
static void Before_Thr_Function(filename, line,
arg1,...,argn)
```

```
{
thr_function_rec_t thr_function_rec;

   MTV_VIEWER_LOCK;
   Set_Header(&thr_function_rec.header, file.name, line,
op_thr_function, BEFORE, 0);

   /* block of code which collects the thread state */

*(thr_function_rec_ptr_t)_MT_Viewer_Buf =
thr_function_rec;

   KILL_MTV;
   STOP_SELF;
   MTV_VIEWER_UNLOCK;
}   /* Before_Thr_Function */
```

The function body is implemented as a critical section guarded by global monitor **MT_VIERWER_LOCK** entry and **MT_VIERWER_UNLOCK** exit macro respectively. It means that in any moment of time only one thread may execute this function. Function **Set_Header()** creates a header of the packet sent to **MTV**. It contains all data which identify the called function in the tested program. Next, this packet is supplemented by any accessible information about the thread state.

The packet is send to **MTV** by the macro **KILL_MTV** and the thread is suspended by the macro **STOP_SELF**. This communication and synchronization mechanism will be described in more detail in the next section. The general structure of the function **After_Thr_Function_1()** is exactly the same as the already described **Before_Thr_Function()** function.

The **thr_function** is not in the global monitor because it is possible that a thread can be blocked after it calls that function. Real state of process and state known by **MTV** is not guarantied to be really consistent (e.g.: before **thr_function** returns system may yield execution to another thread)

The function of structure 2 is used in case, when the thread function call has not a chance to suspend the thread. The **thr_function** is called in the scope of global monitor. In this case the function call and appropriate massage, which describes that function, is sent to **MTV** sequentially. Real state of process and state known by MTV is guaranteed to be really consistent.

The last investigated structure is used only for the **thr_exit()** function call. The state of the thread is impossible to be catched after that function execution, because the calling thread disappears.

The function **After_Thr_Function_2()** is very similar to **After_Thr_Function_1()** except that there are no macros in its body because it is already called in the scope of global synchronization monitor. The skeleton of the function **After_Thr_Function_2()** is as follow:

```
static void After_Thr_Function(filename, line,
arg1,...,argn)
{
   thr_function_rec_t thr_function_rec;

   Set_Header(&thr_function_rec.header, file.name, line,
op_thr_function, AFTER, 0);

   /* block of code which collects the thread state */
```

```
*(thr_function_rec_ptr_t)_MT_Viewer_Buf =
thr_function_rec;
}   /* After_Thr_Function */
```

3.2 Communication and synchronization

As has been pointed out before, monitored program and **MTV** are two separate processes. After user's choice, which program should be monitored, **MTV** is forked by function **fork()**. Next, using the function **exec**, the child process is substituted by the monitored multi–threaded process. As a basic communication mechanism between **MTV** and tested program, FIFO queue has been chosen. This type of communication is free of deadlock danger, even when the monitored program hangs.

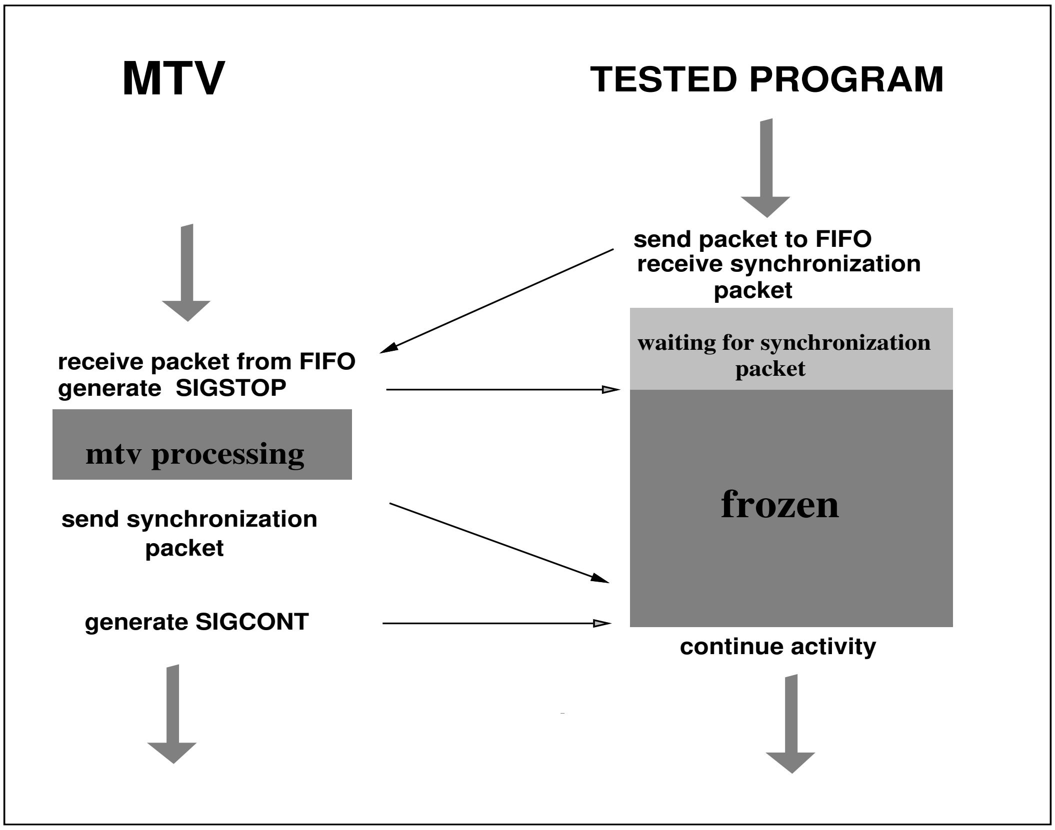


Figure 1: Synchronization between **MTV** and tested program

The proposed communication and synchronization schema is shown in Fig. 3.2. The tested program sends packet of collected data (macro **KILL_MTV**) to the FIFO handled by **MTV**. Next the tested program suspends, waiting for synchronization packet delivered by **MTV**, using the FIFO queue (macro **STOP_SELF**).

After receiving the data packet, **MTV** sends the signal SIGSTOP. This packet plays a very important role, because the tested program hangs in the critical section, in spite of possible delay in the signal SIGSTOP generation. The tested program is resumed in two steps. First, the synchronization packet is put to the FIFO, next, the SIGCONT signal is generated. The proposed mechanism has been very carefully tested on four–processor SUN 514 and works without any problems.

3.3 Mltithreaded program state collection

According to the assumption the aim of **MTV** is to completely control the state of the monitored process. It requires continuous accumulating and updating information about monitored process state. We must control all of the threads elements and their mutual relationships. Each thread element in the monitored process is connected to one record in **MTV**, which describes the current state of that element. Records of the same type form lists. We have five main types of lists in **MTV**: lists of threads, mutexes, condition variables, rw_locks and semaphores. Records are updated on the basis of the information from the monitored process (packets from FIFO). When a new thread, mutex, condition variable, rw_lock or semaphore is initiated, its record is subsequently created and added to the corresponding list. When any thread element is destroyed, its record is destroyed, too. This record contains detailed information about all elements of the thread e.g.: name, address, flags, etc. In order to control the mutual connections between all thread elements, each record owns a number of lists of the so called *small* records, where the information about the connections between thread elements is stored. For example, when a thread is waiting for a mutex, a new element with its descriptor is added to the list of *small* records in the record representation of this mutex. Simultaneously, in the so called *great* record representing that waiting thread, a new element with the address of the mutex at which that thread is blocked is added to the list of *small* records. When the thread becomes the owner of the mutex (and also of the critical section) the corresponding *small* record is removed from the list of waiting threads (in *great* record), and added to the owning list of *small* records. Simultaneously, in the *great* record representing that thread, the *small* record containing the information about the thread waiting at the mutex is removed from the waiting list of *small* records. This *small* record containing the information that, the thread has become the owner of the mutex, is added to the owning list of *small* records. The information in the lists of *small* records allows to obtain a context for all thread elements and is essential for the proposed contextual visualisation. The address of mutex, condition variable, rw_lock or semaphore is the only key to unique identification of this thread element. The descriptor is such a key to threads. Additionally each thread is associated with a serial number. When a new thread is created, its descriptors are not known, but it is possible, that thread will run immediately after having been created - that is the main reason for introduction of the serial number.

The lists of records are the main data structures in the program. User interface refers to specific elements of the lists in order to obtain the information about the state of the monitored process.

4 MTV user interface

Nowadays modern applications have a very clear modular functional structure. There is a logical layer responsible for the user interface on top of **MTV**. It has two main tasks:

- representing the dynamic state of the monitored program;
- conducting a dialogue with end user.

The user interface has been developed in X Window System environment based on OSF/Motif library [6] and that is why it is called GUI (Graphic User Interface).

4.1 GUI software structure

Programming techniques must conform to specific criteria in order to use X11 environment [3]. One of the most important assumptions is that the main loop is not in the user code but in X11 based library. The application is event driven. When an event has accrued, X11 calls one of user hook functions. In fact application consists of functions that create interface elements (collection of widgets) and action functions (callbacks).

MTV has two different logical classes of input sources:

1. events which are generated by X-server,
2. events which are generated by monitored program.

Both of them are asynchronous. The problem is: how can **MTV** — which is a single threaded process — be waiting for these two types of events. A well structured X11 based application should not transfer control to the user code for unspecified period of time. It may lead to situation, when the whole application hangs in a callback function. In our system such a potential danger exists, when the monitored program hangs because of a deadlock.

The solution of this problem is to handle both input sources of events in a uniform way. In X11 system it is possible to register a new source of input events with the XtAppAddInput() function. In **MTV** we register FIFO pipe from the monitored process to **MTV** as such a new source of input.

4.2 Monitored program execution modes

The monitored program may be executed in the following modes: **Step, Continue** and **Animate**. They may be described as follows:

Step In this mode we can watch step by step events generated by the instrumented code of the multi–thread monitored process. When the *Step* button is pressed, **MTV** sends a synchronization byte and SIGCONT signal to the monitored process provided that it is the first *Step* operation just after a message from this process has been received. Otherwise the Step action is ignored (an empty action). So *Step* button simply forwards execution to the next event. It is a basic operation used for other modes of execution implementation.

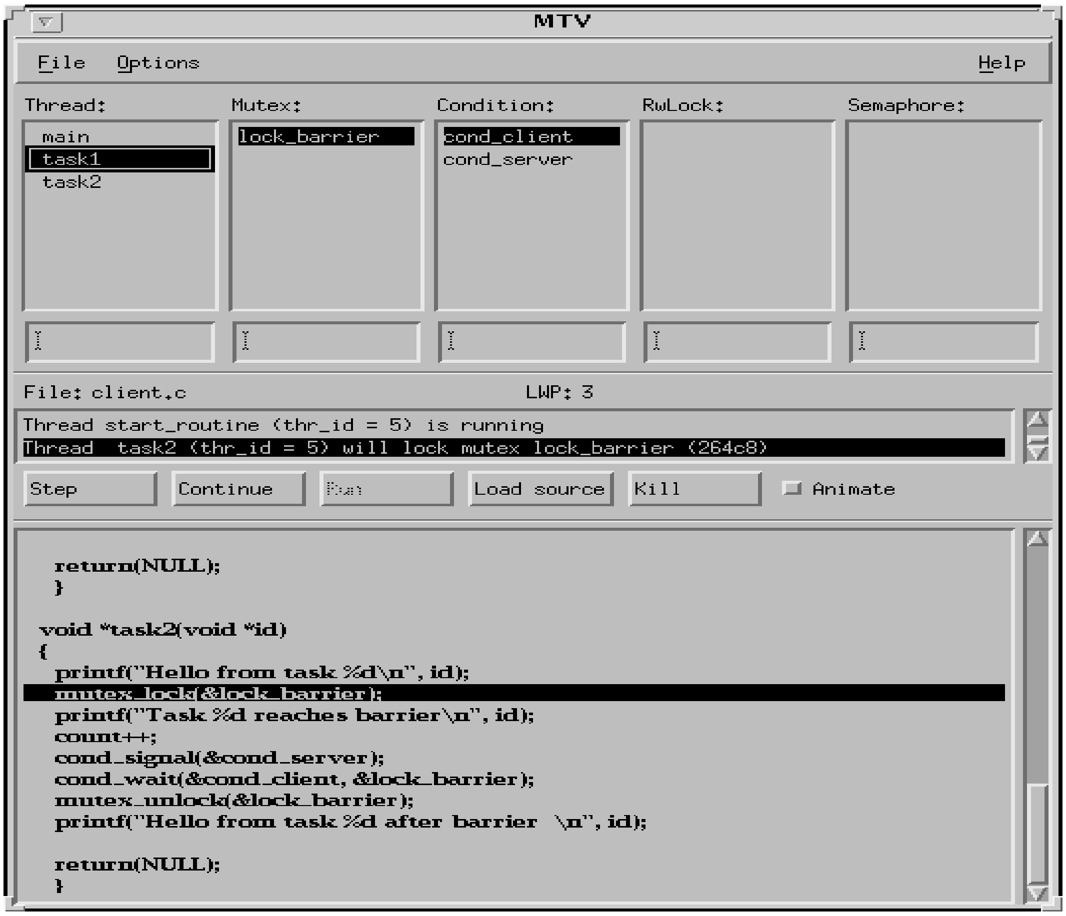
Continue **MTV** provides facilities for setting traps (breakpoints). The traps can be set in the program lines containing thread functions only. They are set and canceled by a double mouse click (or by marking and pressing *Enter*) on the chosen line of the source code in the main **MTV** window. Pressing *Continue* button will force the monitored program to stop on the nearest trap. It means, that the monitored process is running until a breakpoint is reached. It is implemented with XtAppAddWorkProc() function, which is used for Step action registration as so called work procedure. In other words when there is no event in **MTV** event queue, Step action is called until a breakpoint is reached.

Animate Animate is very similar to Continue concept. Toggle *Animate* allows monitoring of the thread program in the animation mode. The option *Animate Rate* accessible in *Option* menu, changes the animation time slice in a continuous way from 0.25 to 2 seconds per Step. It means Step action is called at each slice of time instead of the user pressing the Step button. Step action is called as timed procedure after it is registered with XtAppAddTimeOut() function.

It is possible to switch among the modes during monitoring processes. The button *Kill* makes it possible to finish the monitored process at any time. GUI was developed with BuilderXcessory application generator [14]. This tool allows a very efficient design of application skeleton.

5 Representation of multi–threaded program state

The visualisation subsystem is built of main **MTV** window and five detailed information window one for each thread's element. In the main window each executed thread's operation (thread's function) is illustrated with a matching line of the source code (Fig.2).



File: client.c LWP: 3

```
Thread start_routine (thr_id = 5) is running
Thread  task2 (thr_id = 5) will lock mutex lock_barrier (264c8)
```

```
  return(NULL);
  }

void *task2(void *id)
{
  printf("Hello from task %d\n", id);
  mutex_lock(&lock_barrier);
  printf("Task %d reaches barrier\n", id);
  count++;
  cond_signal(&cond_server);
  cond_wait(&cond_client, &lock_barrier);
  mutex_unlock(&lock_barrier);
  printf("Hello from task %d after barrier  \n", id);

  return(NULL);
  }
```

Figure 2: Main **MTV** main window

The source code of the monitored process is visualized in the lower part of the main window of **MTV**. Dynamic changes of the thread's elements context are displayed in the upper part of the window. They have a form of five lists with the names of thread's elements of the monitored process.

The mutually related thread elements are highlighted. Additionally, context related to any element can be highlighted by clicking on the chosen thread element of any list at any time. For example it is possible to see, which thread is suspended on a condition variable in the critical section created by mutexes, or to recognize the causes of deadlocks.

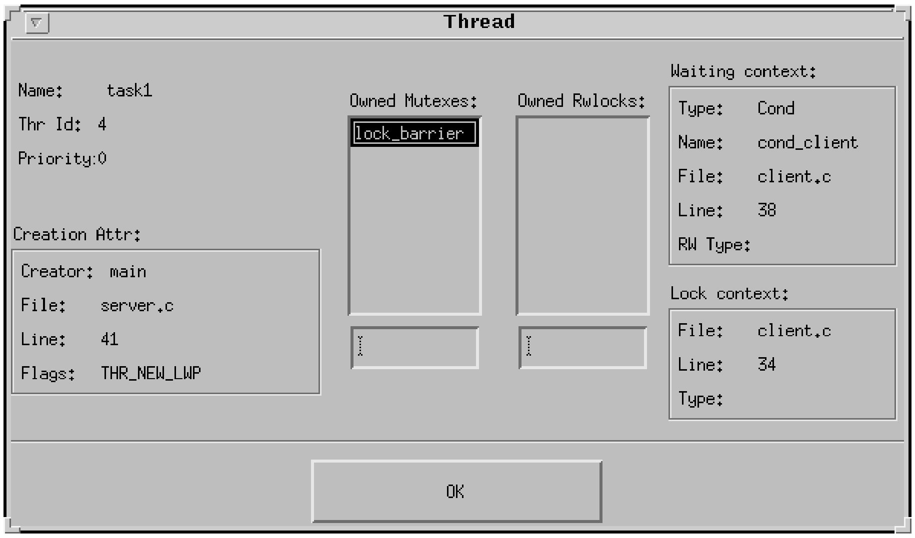
Information such as the name of the source file and the number of LWP (degree of concurrence) are also displayed in **MTV** main window.

There is also a window in the central part of **MTV** main window, where messages about thread events taking place in the monitored process are reported. Also warning and error messages appear in this window. This will be elaborated in the next section.

The windows with detailed information about each element from five threads' elements lists in main window can be obtained by double clicking (or marking and pressing *Enter*) that element. There is at least one list in each detailed window. These lists concern the thread blocking situations. Highlighting of any element of the list causes completing of the proper fields in the window with such information as: the name of file and number of the line where the thread was blocked (in case there is a situation related to r/w_locks additionally the type (read or write) is displayed).

In the detailed information window we also have information about that thread elements such as name, address or descriptor, flags and creation attributes, and additionally for threads - priority, and for semaphores - a counter.

In the threads detailed information window (Fig.3), the monitor which creates the critical section is prefixed, by the character # on the list of the monitors, owned by the thread if it left the critical section being suspended on the condition variable.



Name: task1
Thr Id: 4
Priority:0

Creation Attr:
Creator: main
File: server.c
Line: 41
Flags: THR_NEW_LWP

Owned Mutexes:
lock_barrier

Owned Rwlocks:

Waiting context:
Type: Cond
Name: cond_client
File: client.c
Line: 38
RW Type:

Lock context:
File: client.c
Line: 34
Type:

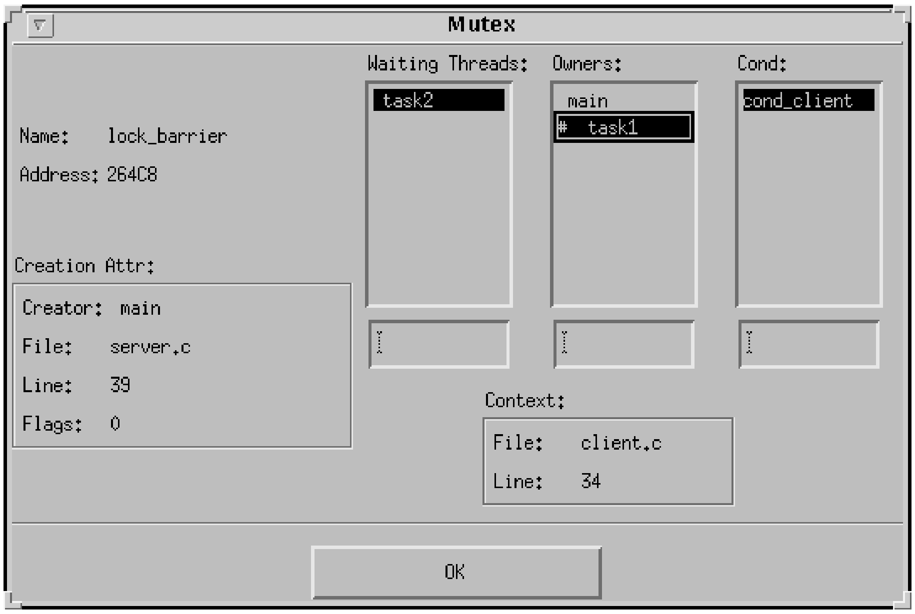
Figure 3: Threads' detailed information window

Also in the monitor detailed information window (see Fig.4), the name of the thread owning the monitor is prefixed by the character #, if the thread is suspended on the condition variable.

The proposed visualisation system seems to be enough powerful for representation of a multi–threaded program state and illustrates well our idea of the contextual debugging. It has been proved by many experiments and elaborated in the next section.

5.1 List of reported errors

Beside monitoring activity of the error free multi–threaded applications **MTV** can also play a role of a kind of debugger. **MTV** is capable of finding and warning of many logical errors, which can be found in multi–threaded programs. All classes of errors are illustrated in DEBUG



Name: lock_barrier
Address: 264C8

Creation Attr:
Creator: main
File: server.c
Line: 39
Flags: 0

Waiting Threads: task2
Owners: main, # task1
Cond: cond_client

Context:
File: client.c
Line: 34

Figure 4: Mutexs' detailed information window

directory of mtv.tar.Z packet (available by ftp on galaxy.uci.agh.edu.pl). Some of errors are very crucial for multi–thread state, e.g.: leaving critical section without unlocking associated monitor. In this case there is no sense to continue such applications, so the only thing **MTV** allows for, is to kill the monitored program. There are also some less important warnings e.g.: uses of uninitialized synchronization variables. It is also possible for **MTV** to find some kind of deadlocks. There is a list of all possible errors detected by **MTV**:

1. Misuse of exiting critical section. A thread exits critical section without unlocking the owned monitor.
2. Misuse of **xxx_destroy()** functions A thread destroys a synchronization elements (monitor, condition variable, r/w lock, semaphore) while it is being used by other thread.
3. Misuse of **xxx_init()** functions A thread reinitializes synchronization elements (monitor, condition variable, r/w lock, semaphore) while it is being used by other thread.
4. Misuse of **cond_wait()** functions A thread, when it is not in the proper critical section, calls **cond_xxxwait()** function.
5. Misuses of **cond_signal()** and **cond_broadcast()** functions. A thread signals condition variable, when there are no threads blocked on it, or the signalling thread is not the owner of the proper monitor.
6. Misuses of **mutex_unlock()** function A thread calls mutex_unlock, while mutex does not exist, or the thread in not the owner of the proper critical section.
7. Uses of uninitialized synchronization elements (monitor, condition variable, r/w lock, semaphore).
8. Misuses of **mutex_trylock()** and **rw_tryxxx()** functions. A thread enters critical section without permission.

9. Checking for:

   - local deadlock - some threads are blocking one another, but there may be some running threads in the process,
   - global deadlock - all threads are blocked in the process for ever,
   - probable global deadlock - probably all threads are blocked in the process; among blocked threads there is at least one thread, which is blocked on timed condition variable.

5.2 Examples

It is difficult to present in this paper every situation when **MTV** is able to identify an error. Just to give only a general idea about, the following three case studies have been investigated.

Case study 1 This program demonstrates local deadlocks. Threads `main`, `task1`, and `task2` are in a local deadlock. The thread `task3` is the only running thread.

```
mutex_t    lock_1;
mutex_t    lock_2;
mutex_t    lock_3;

void *task1(void *);
void *task2(void *);
void *task3(void *);

thread_t th_id1;
thread_t th_id2;

main(int argc, char **argv)
{
   mutex_init(&lock_1, USYNC_THREAD, NULL);
   mutex_init(&lock_2, USYNC_THREAD, NULL);
   mutex_init(&lock_3, USYNC_THREAD, NULL);
   thr_create(NULL, 0, task1, (void *) 1, THR_SUSPENDED, &th_id1);
   thr_create(NULL, 0, task2, (void *) 2, THR_SUSPENDED, &th_id2);
   thr_create(NULL, 0, task3, (void *) 3, THR_NEW_LWP, NULL);
   mutex_lock(&lock_1);
   thr_continue(th_id1);
   sleep(1);
   mutex_lock(&lock_2);
   mutex_unlock(&lock_2);
   mutex_unlock(&lock_1);
   return(0);
}

void *task1(void *id)
{
   mutex_lock(&lock_2);
   thr_continue(th_id2);
   sleep(1);
   mutex_lock(&lock_3);
   mutex_unlock(&lock_2);
   mutex_unlock(&lock_3);
   return(NULL);
}
```

```
void *task2(void *id)
{
   mutex_lock(&lock_3);
   mutex_lock(&lock_1);
   mutex_unlock(&lock_1);
   mutex_unlock(&lock_3);
   return(NULL);
 }

 void *task3(void *id)
 {
    while(1)
    {
       printf("Hello from task  %d\n", id);
       sleep(1);
    }
  return(NULL);
  }
```

Case study 2 This program demonstrates misuses of `con_signal()` and `cond_broadcast()` functions. The thread `main` signals condition variable `cond_1` when there are no threads blocked on it. The thread `task1` calls `cond_broadcast()`, when it is not the owner of monitor `lock_1`.

```
mutex_t    lock_1;
mutex_t    lock_2;
cond_t     cond_1;

void *task1(void *);

main(int argc, char **argv)
{
   cond_init(&cond_1, USYNC_THREAD, NULL);
   mutex_init(&lock_1, USYNC_THREAD, NULL);
   mutex_init(&lock_2, USYNC_THREAD, NULL);

   thr_create(NULL, 0, task1, (void *) 1, THR_NEW_LWP, NULL);

   mutex_lock(&lock_1);
   cond_signal(&cond_1);
               /* WRONG - no threads are blocked on cond_1 */
      cond_wait(&cond_1, &lock_1);
   mutex_unlock(&lock_1);
   thr_join(0, NULL, NULL);
   return(0);
}

void *task1(void *id)
{
   printf("Hello from task %d\n", id);
   mutex_lock(&lock_2);
      cond_broadcast(&cond_1);
               /* WRONG - calling
cond_broadcast() not within
                  the scope of monitor
lock_1 */
```

```
   mutex_unlock(&lock_2);
   return(NULL);
}
```

Case study 3 This program demontrates a misuse of critical section. The thread `task1` exits without unlocking the monitor `lock1`.

```
mutex_t    lock1;
cond_t     cond1;

void *task1(void *);

main(int argc, char **argv)
{
   cond_init (&cond1, USYNC_THREAD, NULL);
   mutex_init(&lock1, USYNC_THREAD, NULL);

   thr_create(NULL, 0, task1, (void *) 1, THR_NEW_LWP, NULL);

   mutex_lock(&lock1);
   printf("Hello from main\n");
   cond_wait(&cond1, &lock1);
   mutex_unlock(&lock1);
   printf("End main\n");
   return(0);
 }

void *task1(void *id)
{
   printf("Hello from task %d\n", id);
   mutex_lock(&lock1);
   printf("Hello from task %d after lock lock1\n", id);
   cond_signal(&cond1);
   thr_exit(0);
              /* WRONG - task1 is still the owner of lock_1 */
   return(NULL);
}
```

6 Conclusions

The proposed idea of the contextual debugger seems to be very attractive for multi–threaded applications. The complexity of such programs state analysis requires suitable representation of information about threads' elements and support for automatic identification relation between them. It is a good platform for automatic recognition of certain typical but difficult to find in larger programs bugs. The basic feature of the proposed debugger have been successfully proved. The most challenging area such as synchronization between a multi–threaded program running on symmetric multiprocessor machine and monitoring environment needs still a further investigation.

The proposed sequential visualisation of events is not satisfactory solution. The more global approach with multi–threaded monitoring program and the same support from kernel functions may be necessary to gain better results on this field.

Bibliography

[1] S. Kleiman, J. Voll, *Symmetric Multiprecessing in Solaris 2.0* SunSoft Inc. Mountain View, California.

[2] S. Kleiman, B. Smmalder, D. Stein, D. Shah, Writing Multithreaded Code in Solaris, SunSoft Inc. Mountain View, California.

[3] *Motif 1.2 Programmer's Reference Manual* - Sun Microsystems, 1992.

[4] F. Mueller, *Pthreads Library Interface.* Department of Computer Science, B-173, Florida State University.

[5] F. MuelLer, Implementing POSIX Threads under UNIX: Description of Work in Progress, Proc. of the second Software Eng. Research Forum, Nov. 1992, Melbourne, Florida.

[6] A. Nye, T. O'Reilly *X Toolkit Intrinsics Reference Manual.* O'Reilly & Associates, Inc., 1990.

[7] M.L. Powell, S.R. Kleinman, S. Barton, D. Shah, D. Stein, M. Weeks, SunOs Multi-thread Architecture, USENIX-Conference Winter'91, Dallas, TX.

[8] *SunOS 5.3 System Services* - Sun Microsystems, 1992.

[9] D. Stein, D. Shah, Implementing Lightweight Threads, Summer'92 USENIX - June 8-12, 1992 - San Antonio, TX.

[10] *SunOS 5.3 Guide to Multi–thread* - Sun Microsystems, 1992.

[11] *SunOS Reference Manual.* Sun Microsystems, 1990.

[12] *Sparc Works version2.0 - Debugging a Program.* SunPro, 1992.

[13] A.S. Tanenbaum, *Modern Operating System.* Amsterdam, The Netherland 1992: by Princie-Hall, Inc.

[14] *The BuilderXcesory User's Guide* 1988-1992 Integrated Computer Solutions, Inc.