PVM Light Weight Process Package

by
Weihaw Chuang

December 14, 1994

Abstract:
PVM Light Weight Process package or PLWP is a multi-threaded environment built over the Parallel Virtual Machine (PVM). Its original purpose was to furnish a laboratory for James Hoe to experiment with thread scheduling strategies in his FUNi project. PLWP has since become more flexible, and now permits user-definable thread scheduling and object-passing for general computation. This is done through a software specification using virtual-specification classes, which permits run-time selectable C++ classes that are able to determine the correct member functions from their class hierarchy. Written in this fashion, run-time selectable scheduler classes can save and restore shared contexts, and manage thread selection. Similarly, object-passing classes can send themselves between threads on the same task or on separate tasks transparently. PLWP's environment enables preemptive context switching, simplifies tasks naming, and is portable through the use of QuickThreads abstract thread type.

Acknowledgments

Thanks to James Hoe for his advice, assistance, and patience for this project and many others. His insights have been very useful and are much appreciated.

Many thanks to the close friends that I've had at the Institute and Lexington High School. They made this interminable ordeal called "school" survivable. Thank you Juhan, Peter, Albert and Jin.

Thanks to my brother Warren for commiserating with me, and then making me laugh.

Most importantly I give my thanks to my parents for their foresight, patience, support, and love. Only now can I fathom the sacrifices they made to raise two kids, and put them through school. I dedicate this project to them.

Contents

Figures and Examples

1 Introduction

PVM Light Weight Process package or PLWP provides a multi-threaded environment over the Parallel Virtual Machine (PVM). Its purpose originally was to provide a laboratory for testing thread-scheduling-strategies on James Hoe's FUNi network, and is still meant to do this. PLWP has since become more generalized to allow experimentation of thread-scheduling-strategies in the broader context of distributed computing. Further, its design is modular for future enhancement. Scheduling is performed by a user-definable scheduler class, with context-switching handled by David Keppel's QuickThreads. Messages in PLWP are C++ objects that are user-definable. PLWP protects the non-reentrant PVM function from interrupts, allowing preemptive context-switching. If PVM message-passing semantics are desired, the `pmsg` class provides a similar send and receive interface, plus capabilities to pass common non-C++ class data-types. PLWP is written in C++ to facilitate formal structure, and cleaner interfaces.

1.1 Format of the Report

The intent of this report is to introduce PLWP version 0.3 (alpha), and to provide information needed for setting up and using this environment. Key ideas are backed by examples to demonstrate their use. Unfortunately time does not permit every feature of PLWP to be elucidated upon, particularly internal details[1]. Some knowledge of PVM and C++ is assumed.

This report uses a notation convention that should be clarified. Text written in courier font like `ptask` are C++ variables, functions, keywords, or source code. The word `this` refers to the class object being discussed, and the meaning is analogous to "itself". Its usage comes from the `this` implicit parameter given in all C++ member functions, and is a pointer to the class-object, in other words "itself".

---

[1] I apologize for any potential confusion, and if there are questions about the report or PLWP, contact the author at `wech@lcs.mit.edu`.

The report will follow this format: The first section is the introduction, covering the motivation behind the project, an outline of PLWP strengths, and a description of related work. Background information is covered in the second section, which describes the foundation libraries- PVM and QuickThreads- and the scaffolding between the libraries- the C++ language. The third section introduces the software interface and general details about using PLWP. The fourth section describes how to write PLWP message and scheduler classes. Described here is the virtual-specification class paradigm. The fifth section characterizes the consequences of PLWP's implementation viz. where it can run and its limitations. Concluding is the sixth section.

1.2 History

The original motivation for this project is to create a thread context-switcher for James Hoe's Fast User-level Network Interface project (FUNi) [Hoe]. Two features of PLWP are derived from FUNi's characteristics, and are worth mentioning. FUNi provides an interface into a high bandwidth network called ARTIC, allowing clusters of commercial workstations to be harnessed for distributed computation. Being based on stock workstations, communication over Ethernet is possible, allowing complementary message-passing capability through PVM. PLWP takes advantage of this. FUNi's message-passing interface is given as send and receive FIFO buffers in user memory, where a user process can place or retrieve messages sent over the network. Threads share the registers managing the FIFO's and FUNi state, and consequently the registers need to be saved and restored during context switches. The current version of PLWP can do this.

For version 0.1 of PLWP, I created a context switcher with fixed round-robin scheduling using PVM as the communication mechanism. After finishing the initial version of PLWP, James wanted to generalize the package. He said there were three areas that needed additional work- make PLWP's structure amenable to future changes, generalize message-passing, and provide a flexible scheduler. Performance of the scheduler, he said was a secondary issue as context switching is infrequent. This allowed for some freedom of design. The result is version 0.2 of PLWP. The current version, 0.3, incorporates bug fixes and documentation changes, and is described in this report.

1.3 Environment

PLWP creates an environment that's even simpler to use than PVM by reducing the complexity of initialization, task[2] naming, and critical sections. The user need not specify PVM initialization as PLWP takes care of this by a C++ trick– a `ptask` global object called `gTask` is created automatically upon startup. This object's creator makes the necessary initialization calls. Also if the PLWP task is spawned, `gTask` obtains task-naming information from the parent task, and informs the PLWP "world" that "it's alive". PLWP task-naming binds an Abstract ID (AbsID) integer to a PVM Task ID (TID). This simplifies writing code when dealing with task names, to using a constant specified before compilation, as opposed to manipulating a PVM TID indirectly through a variable or an array element. Thinking about and programming groups of tasks by ranges of constants is easier than arrays, hopefully making complex task-control-hierarchies simpler to deal with. Some PVM calls are not reentrant such as `pvm_send` and `pvm_brecv`, and are dangerous if preemptive context switching is to occur. This is addressed by encapsulating the PVM calls in critical sections within PLWP, in a manner transparent to the user.

1.4 Object-Passing

PLWP is structured towards sending C++ class or struct objects as messages. It also can send data types like integers and character arrays if encapsulated in an object provided for that purpose. In keeping PLWP's interface consistent with PVM semantics, the sender specifies a task and thread for the destination, and the receiver specifies a task and thread for the source. A tag is provided to differentiate between different messages. This is especially useful for insuring that messages of a certain type are received correctly. It is also possible to specify a wild card[3] on the receiving end for the Abstract ID, Thread

---

[2] A UNIX process enrolled in the Parallel Virtual Machine environment by making a PVM library call.
[3] In a special case, the destination Thread ID can also be given as a wild card.

ID, and tag. Messages between threads on the same task are sent and received in the same manner as messages between tasks.

Object-passing allows for better software engineering. The first reason is encapsulation of send and receive member functions helps isolate the send and receive critical sections. Though the user could code them himself, it becomes burdensome to manage masking and unmasking interrupts. This means the user writes less code, which in turn reduces the probability of bugs. The second reason is that if C++ is used, then there should be a way to send and receive C++ objects. In C++ the primary data structure is the class object, and the ability to pass it was necessary to continue development in C++. The third reason is that isolating a message's PVM dependencies to pvm_pkxxx and pvm_upkxxx calls makes it easy to convert messages to use FUNi's message-passing interface. All James will have to do is write the equivalent pack and unpack functions for FUNi, modify two sets of send and receive functions in PLWP, and recompile.

PLWP does not exclude the direct use of PVM message passing. If the user desires this, he must protect the PVM functions from interrupts, particularly the send and receive calls, and differentiate the message-tag format of the PVM direct message from that used by PLWP. It may also be necessary to uncomment-out code that protects old PVM message buffer state during PLWP actions.

1.5 Scheduling

Thread scheduling is a complex topic that James wanted to experiment with. The complexity arises from dealing with loosely coupled distributed processes that have a variety of control hierarchies like master-slave or SPMD, combined with multitudes of thread scheduling strategies. These combinations are not well understood, so James wanted a flexible mechanism to explore several different strategies. Examples he mentioned are gang scheduling, priority scheduling, and load-balancing scheduling. My solution was to create a scheduler object that could easily be swapped at run-time. I followed David Keppel's design methodology for QuickThreads [Keppel], resulting in a clear interface between the scheduler code and PLWP code. In other words, all the responsibility of scheduling is pushed into the scheduler object, and encapsulated with a uniform interface. The scheduler object has complete control on how it selects its threads, communicates with other tasks' schedulers, sets up the timer interrupt, and saves and restores shared contexts. Regarding to the task of scheduling, PLWP manages the underlying context switching, thread tables, state information, and interrupts.

1.6 Related Work

There are several multi-threaded PVM packages available or in development: TPVM[4], (DTh[5]), PVMt[6], and NewThreads[7]. Documentation for all these packages was obtained, however neither their code nor performance was evaluated. PLWP, TPVM, and NewThreads all appear to follow the "pod" paradigm where the task acts as a pod for threads running inside of it. PVM in these cases is not modified. TPVM is still under development, and information was based on a preliminary draft. It has an interesting thread control mechanism based on a set of rules, when satisfied, triggers (starts execution) a thread's execution[8]. In addition it has normal thread synchronization mechanisms, and a unique remote-memory copying feature. NewThread provides a thread package with a simple interface, and cooperative multi-threading. Curiously they initially wrote their package in C++, but later switched to C for performance reasons. PVMt, still under construction, is a comprehensive attempt at making PVM multi-threading. Its author obtained source code for PVM, and made the necessary modification to the daemon and library to provide each thread with an active message buffer. It appears to have a sophisticated mechanism for dealing with blocking receives to increase performance. If the blocking-receive does not find the message, PVMt context switches and places the

---

[4] Contact Adam Ferrari email: `ajf2j@cs.virginia.edu` to obtain TPVM documentation and prelimary source code.

[5] Contact Farhad Arbab email: `farhad@cwi.nl` to obtain the DTh library package.

[6] Contact Emmanuel Ackaouy email: `ack@cs.wisc.edu` to obtain PVMt documentation. PVMt status is not known.

[7] ftp `ftp.cs.washington.edu` in `/pub/dylan`. Ackaouy says that NewThreads has been updated recently by someone at U. Wisconsin.

[8] It is claimed that this is analogous to dataflow driven threads.

unreceived message in a unreceived message queue. When the requested message arrives, PVMt returns control to the original thread. Rumors exist of an official multi-threaded PVM package coming out eventually.

2 Background

PLWP builds upon two libraries– PVM provides the interprocess message communications, and QuickThreads provides the light weight threads primitives with C++ providing the scaffolding. Concepts from these two packages and C++ influenced the design of PLWP, as described below. Two other packages are used as well: the hash tables from JCOOL C++ package, and several segments of code from James Hoe's LWP package. A modified version of the hash tables is discussed in section three.

2.1 PVM

Parallel Virtual Machine (PVM) is a distributed environment that provides uniform communication mechanisms like message passing and signaling with an "easy-to-use" interface. PVM simplifies the semantics of message- passing, and abstracts many of the hardware and network details [Sunderam]. Consequently PVM is available for most workstations and several supercomputers, and programs written over PVM can easily be ported from one PVM-supported platform to another. Message-passing in PVM is geared towards sending large arrays (vectors) between machines of differing architectures (heterogeneous environment) with automatic conversion for different numerical representation. PVM's unit of computation is the task viz. a UNIX process, and limits it to large grain parallelism.

2.2 Quick Threads

While looking for a suitable thread package, especially one that would work on the SPARC architecture, it was suggested[9] I use QuickThreads by David Keppel [Keppel]. It is available for: Intel 80x86, KSR-1, KSR-2, Motorola 88x00, MIPS Rx000, Sun SPARC, DEC VAX, DEC Alpha AXP, and HP-PA platforms. Other architectures maybe forth coming. Keppel's package is designed for others to build thread packages around his thread Abstract Data Type (ADT) with mechanisms for: thread creation, initialization, context switching, and aborting. From this project's perspective, QuickThreads was

[9] In reply to a query on comp.parallel.

especially useful in dealing with vagaries of stack growth direction[10], and context switching SPARC register windows. As the Thread ADT is minimal, its context switching is very fast, approximately the cost of a function call[11]. QuickThreads intentionally does not provide mechanisms for synchronization or architecture/system dependent features. This allowed the Thread ADT to be minimal and uncomplex, so that the user could easily customize QuickThreads to their requirements.

2.3 C++

Good software engineering is a primary requirement of this project, and one strength of C++ is its structured syntax. C++ allows a form of programming called "data abstraction" [Stroupstrup, Liskov] which allows the programmer to create modular structure with clear interfaces. This allows greater readability through formal structure and makes procedural dependencies more apparent. Consequently future modifications should be easier to make. The cost is additional function calls- a performance burden. However, future C++ features should ameliorate some of the costs, and improve structure. C++ will provide function inlining[12] which in theory should eliminate the cost of function call. Clu like features such as templates[13] and exception handling will further improve structure and allow graceful error recovery.

C++ can perform run-time member function selection that obeys inheritance by declaring the function to be virtual. [Winston] This is useful for specifying classes to have a common feature set and interface, yet provide different behaviors. PLWP's use of virtual functions in the context of virtual-base classes is described in detail in section four.

---

10 Stacks can grow up or down depending on the architecture.
11 A quick performance test revealed context switch was about 40 µS on CSG Sparcs.
12 As of the writing of the report, it was not certain whether G++ has function inlining
13 In Clu its called parameterized clusters.

3 Programming Interface

This section is an introduction to the PLWP software interface. Overall this explains how to get PLWP running, and covers topics like its environment, control, and basic message passing through PLWP. A number of secondary member functions and features are left out– principally debugging and gratuitous reader functions. They are unnecessary for getting a program running, and can be found in code if needed.

3.1 Overview

PLWP consists of C++ classes and helper functions (see diagram 1). The three principle classes are: `ptask`, `plwp`, and `pmsg`. `ptask` acts as the primary interface for controlling PLWP, maintaining its state information, and providing low level PLWP object-passing services. `plwp` class encapsulates QuickThreads, plus stores thread related information like performance. `pmsg` is an application oriented object (message) passing class that can send arrays, integers, and virtual message objects. This object-passing class is strict about specifying a destination thread to prevent confusion over the recipient. In contrast, a message-passing class called `mmsg` is used by PLWP and schedulers to send a message to a wildcard (any) destination thread on another task. Besides these principle classes, PLWP also has a set of helper functions and miscellaneous classes that deal with strings, masking and unmasking interrupts, hash tables, etc.

These classes and procedures are currently available as a set of C++ files and headers that must be compiled with the application[14]. At that time, the PVM and QuickThreads libraries must be linked to the application as well. A makefile template is provided to simplify this. Eventually, if PLWP becomes stable enough, a library will be built.

[14] PLWP source code is located in `/home/prj3/wech/PLWPv02/Code at LCS CSG.`

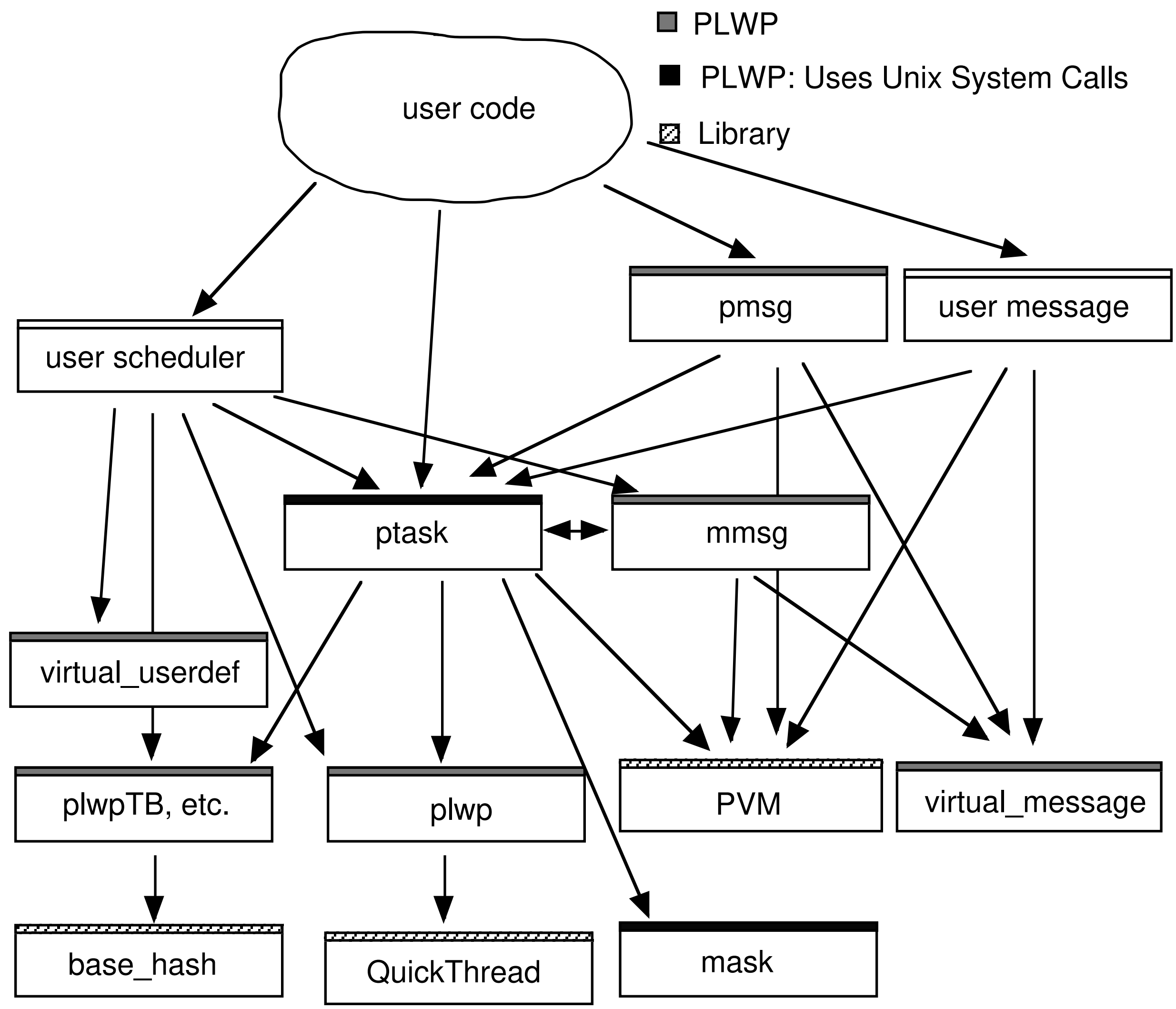

Figure 1. Software Dependency Diagram (Abridged)

Task and thread naming are abstracted and simplified into integers. Every PLWP task has a constant integer name called "Abstract ID" which is bound to an associated PVM TID. The Abstract ID simplifies task identification to a constant specified before compilation, and lets PLWP worry about PVM naming details. Normally the Abstract ID is assigned in source code as a parameter to the task spawning function– the exception being the command line launched task which defaults to 0. In a fashion similar to Tasks, Threads are named using a constant integer called the "Thread ID". The permissible Thread ID values are between 0 and 127, with the main thread assigned the default value 0. Name binding for each task are stored in the `ptask` database object called `gTask`.

3.2 Ptask

ptask class has two functions– it is the database for task information, and supports actions like creating and maintaining threads, and spawning tasks. To help control the database, reader and writer functions are provided to manipulate some of ptask member variables. However a number of variables must be accessed directly, in particular the plwpTB, pTaskTB, and pSpawnedTB hash tables pointers. This is done because encapsulating the hash table's large interface is too troublesome. Note that ptasks is the only class in PLWP whose member variables should be accessed like this– all other classes have a complete set of reader and writer functions.

A PLWP thread's existence is marked by its creation, context switches, and its abort. plwp_create_lwp member function creates a new thread given a pointer to a function f and a pointer to a scheduler. QuickThread stipulates that f must be a non-class member function with an integer pointer argument. The scheduler must be a subclass of virtual-specification class virtual_userdef and can be user defined (described later). When plwp_create_lwp returns, it leaves the thread in a ready-to-run state in the thread table plwpTB. When this thread is context switched to, for the first time, f is called, and runs until it context switches or returns. In the case that f returns, plwp_abort is called killing the thread. plwp_start perform the initial context switch from the main thread to a thread specified by the main thread's scheduler. The user can pass a thread scheduler through plwp_start, otherwise ptask uses the default cooperative_userdef scheduler. It also sets up PLWP state to allow both cooperative and preemptive context switches. plwp_yield is used to cooperatively context switch. plwp_abort kills the current thread and context switches to a new thread. For all thread context-switches, the scheduler associated with the current thread is used to determine the next thread to run.

Tasks are spawned and to a limited degree controlled through ptask. plwp_spawn_task spawns a task assuming it to have PLWP incorporated. If it is not, the child task might get confused by the ensuing initialization messages from the parent task. `plwp_exit` terminates itself (task) with the appropriate notification to other PLWP tasks, and to PVM. `plwp_print_current` gives a string with the long version of task status information. `plwp_print_all` asynchronously[15] asks its spawned tasks to give strings with the short version of their status information, which is then given to the user. Currently the string status information is undefined, but can be assumed to have thread information, and the task name bindings in human readable form.

3.3 Pmsg

Although this section describes pmsg, it is also applicable to any other PLWP message class that uses the `virtual_message` virtual-specification class. The class is meant to provide message passing services to help prototype applications. Actually `pmsg` is an umbrella description for two classes– a `send_msg` class and a `recv_msg` class, which are derived from `virtual_message` through `base_msg` class. `pmsg` is a general tool, and the user will want to eventually create custom message classes specific to his needs. This process is described later. `pmsg` can encapsulate four different types of data: an integer, integer array, memory buffer (void pointer), and a virtual_message object. The first two are self explanatory. The third sends arbitrary objects like structs without data conversion and type protection. The fourth is described several paragraphs below.

To create a `send_msg` object, data is passed to the creator function and is encapsulated. `send_msg` class insures data cannot be modified (easily) after encapsulation by providing no writer functions. This semantic mimics the PVM message buffer functionality which, after the data has been packed in a PVM send buffer, cannot be modified. A send or mcast operation then performs the actual send(s). These functions are given a destination thread which can be in another task, in the same task, or even itself, and a message tag.

`recv_msg` semantics follows from `send_msg`. Initially an empty `recv_msg` is created. The one exception is when the data to be received, is an object– this is described

[15] The code for this function provides a useful template for asynchronous requests. See also the associated interrupt handler for this function.

in the next paragraph. Assume for now that the data is either an integer, integer array, or void pointer. At some point after creation, the object receives data using either blocking receive or non blocking receive from a user specified thread and message tag (or wildcards). . As their name implies blocking receive busy waits[16] until a message arrives, while non-blocking receives return a Boolean indicating whether it receives a message or not. After a message is sent and received, the `recv_msg` object is updated with data. This data can then be accessed using reader functions.

Message passing with objects using `pmsg` differs from the data types given above. First the class of the object to be sent, must be a derived class of `virtual_message` and have certain virtual-member functions overloading those found in `virtual_message`. Although this sounds troublesome, keep in mind that any arbitrary class with these attributes can be used, and that writing overloading member-functions is simpler than writing out the equivalent PVM code. This is described in section four. Second, the `recv_msg` creator requires a `virtual_message` objects, empty or otherwise, as a template describing how messages are to be received. This is covered in section four as well.

Messages are forwarded by their destination address and message tag. Abstract (Task) ID and Thread ID act as references to the source and destination addresses. These integer ID's are described above. Sometime Abstract ID and Thread ID are encapsulated into an `lwp_ID` class object, creating the notion of a thread address. This is a relic of the previous version of PLWP and should not be used. Messages can be further differentiated by a message tag which is useful for message type differentiation. In this implementation the number of tags are limited to the range of 0 and 2047. There is an additional low level tag called `code` which is not visible to user code. It has a range of 0-15 and is used to distinguish messages classes– for example `mmsg` from `pmsg`.

[16] The thread repeatedly loops checking the condition of the buffer.

3.4 Hash Tables

Hash tables in PLWP are a modified version of the one found in the "JCOOL" C++ library. They feature automatic table re-sizing, and are accessed like a circular list. The later feature is especially useful for implementing round-robin schedulers. The original parameterized tables are modified into four different tables[17]. Each `ptask` (one in each task) has a `plwp_Table` thread table called `plwpTB`, and two `Task_Tables` called `pTaskTB` and `pSpawnedTB` which are used for world name-binding, and spawned-task information. A fourth `pmsg_Table` table called `pmsgTB` is found in each thread to store inter-thread messages. The user need only be concerned with the `plwpTB` table as it is used in scheduling. It binds Thread ID key to a `plwp` thread-pointer value.

The interfaces to the hash table member-functions are identical. The appendix has a listing of the member functions, and describe in greater detail the interface, and behavior. Key-value pairs are stored, retrieved,and deleted through the respective use of `put`, `get`, and `remove` member function. The hash tables can act like a circular list, with a current key-value pointer that references some element if there is valid data. The pointer can also be defined invalid when the table is empty. This is manipulated through the `next`, `prev`, `key`, `value`, and `find` functions– `next` increments the current pointer to the successive element, `prev` decrements the pointer to the previous element, `key` returns the key for the current pair, `value` returns the value for the current pair, and `find` moves the pointer to match the key passed as a parameter. The size of the hash table can be found through the `length` member function.

3.5 Example

The following multi-threaded program is called "test_timer". The threads send int messages to a slave called "test_timer_slave", busy wait for a response, and prints out the slave's string reply. "test_timer" preemptively context switches itself and its slave every

---

[17] Unfortunately G++ gets confused by some of the advanced parameter features found in these libraries thus three different sets of code.

second using the `preemptive` scheduler. For clarity some source code has been removed.

```
/* thread */
void* print1(void *baz)
{
  int i, cc;
  int foo= *((int *) baz);
  /* continuously prints out messages sent by slaves */
  while(1) {
    i++;
    send_msg smsg(i);
    lwp_ID whoID(1, foo);
    smsg.send(whoID, WORD_MSG);
    string bar;
    recv_msg rmsg(&bar);
    cc = rmsg.brecv(-1,-1,STRING_MSG);
    mask_plwp();
    mvprintw(foo,1,"mask: %d %d\tdmsg: \t%s",gMaskPlwpLevel,
             gMaskParticularLevel, bar.get_string());
    refresh();
    unmask_plwp();
  }
}


/* main thread */
main()
{
  mask_plwp();
  initscr();
  clear();
  refresh();
  unmask_plwp();
  preemptive* CuPtr;
  CuPtr = new preemptive(1.0);
  int one =1, where;
  gTask.plwp_spawn_task("test_timer_slave", (char **) 0,
                    &one, &where);
  cout << "task one: " << one << " where: " << where <<endl;
  int onePtr = 1;
  gTask.plwp_create_lwp(onePtr, CuPtr,
                  &print1, (void *) &onePtr);
  CuPtr = new preemptive(1.0);
  int twoPtr = 2;
  gTask.plwp_create_lwp(twoPtr, CuPtr, &print1,
                  (void *) &twoPtr);
  string bar;
  gTask.plwp_print_current(bar);
  CuPtr = new preemptive(1.0);

  /* makes sure slave is alive before preemptive context switch */
  while(gTask.plwp_alive(1)==0);
  cout << "Its Alive!" << endl;

  gTask.plwp_start(CuPtr);
  cerr << "main done\n";
```

```
}
```

Example 1. test_timer.cc Code

4 Virtual Class Interface

This section covers the implementation of new message-passing and scheduler classes. Both use the virtual-specification[18] class concept to implement run-time member function selection. The action of scheduling and message-passing requires several behaviors (procedure calls) and some state information which C++ classes satisfy. Unfortunately C++ figures out which member function to call during compile time, and is strict about type checking. This posses difficulty for run-time assignment of objects, and calling the object's member function to perform an action. Virtual-specification classes discard this strictness and allows run-time assignment and member function selection. To implement this, member functions in a base class (see figure 2) are declared `virtual`. Now all member functions in the class hierarchy of exactly the same name and interface become virtual. When run-time member function selection is required, the member function closest in class hierarchy to the object, is selected[19]. The type checker will allow assignment of objects to pointers which are its superclass. This means that any object declared subclass of a virtual-specification class can be assigned to a pointer of type virtual-specification class. Applying this paradigm to the message class and scheduler class is described below in the following two subsections.

[18] The name "virtual-specification" class was chosen because it bests characterizes the concept, and does not conflict with pre-existing official C++ names.
[19] More formally it picks the method that shadows the other methods.

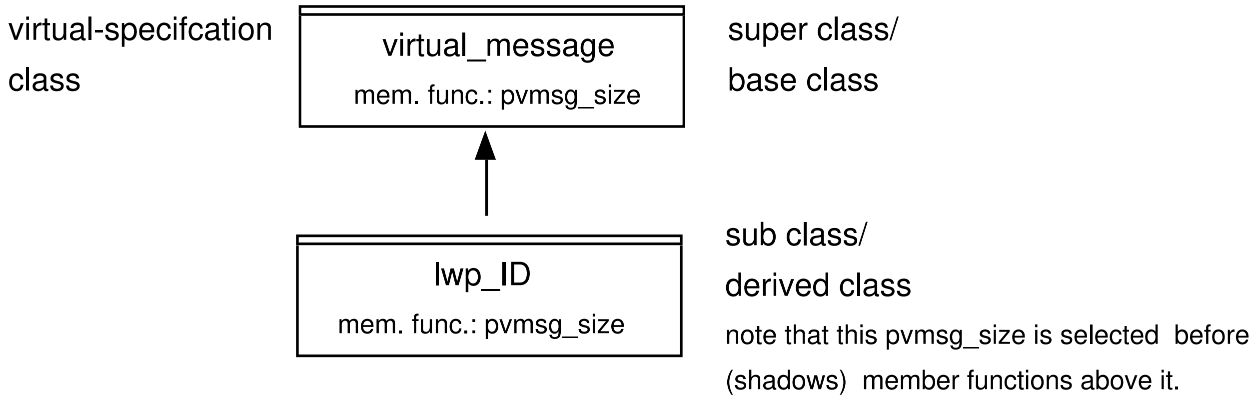

Figure 2. Example of Class Hierarchy (with Virtual-Specification Class)

```
class virtual_message {      /* base class */
 virtual int pvmsg_size() { /* define this function to be virtual */
   return sizeof(virtual_message);
   }
  ...
  }


class lwp_ID : public virtual_message {
 /* derived class of virtual_message */
 int pvmsg_size() {
   return sizeof(lwp_ID);   /* becomes virtual */
   }
  ...
  }
```

Example 2. Code Snippet for Figure 2

4.1 Virtual_message

A C++ classes can be converted into a PLWP message by inheriting the `virtual_messages` virtual-specification class, and adding the necessary member functions. This provides the actions necessary for sending messages between tasks through PVM, and between threads on the same task. Five member functions, made virtual through `virtual_messages`, should be provided as given in the appendix.

The following is a summary of `virtual_messages` member-function behavior. To implement object-passing over PVM, two functions called `pvmsg_pack` and `pvmsg_unpack` are needed to pack and unpack the PVM buffer. Specifically, `pvmsg_pack` writes data from itself into the send-buffer using PVM packings calls. When done, PLWP sends the buffer through PVM to the recipient. The message is received by the destination task which then calls `pvmsg_unpack`. Here PVM unpackings calls are used to store data into either a newly created object which is returned, or itself (`this` object, which is not returned). The later case allows the class to match the semantics of the PVM unpack calls where data is written into a pre-existing data structure. Objects passed between threads use `pvmsg_replicate` and `pvmsg_copy` functions. `pvmsg_replicate` returns a duplicate of itself which is forwarded to the correct message table. Duplication prevents the recipient thread from modifying the sending thread's object. On the receive side, `pvmsg_copy` takes the object found in the message table, and either returns it or stores the data in `this` (itself). `pvmsg_copy` "return/store" behavior parallels `pvmsg_unpack`. The fifth function `pvmsg_size` returns the size of the object in bytes.

Receiving an object requires an object of the same type be passed as a parameter into `ptask::help_nrecv_msg` (or `ptask::help_nrecv_main`) to be used as a template. This is necessary whether the template object receives the sent object's data, or the template object is used as a "pattern" to create a new object. The reason is that the underlying `ptask::help_nrecv_msg` mechanism needs the class-object to figure out which class's member function to call. If the passed object's class differs from the one being received, the program will crash[20]. Appropriate use of message tags can prevent this. This dangerous behavior is no different than PVM's– if PVM unpack-calls are used incorrectly, the task will crash as well.

Altogether the five member functions provide a safe means of sending messages. However the specified behavior of the member functions need not be followed. For example if a user wants to make a fast but unsafe inter-thread object-passing class, only `pvu_replicate` and `pvu_copy` need be specified. `pvu_replicate` and `pvu_copy` returns a pointer to itself (`this`), circumventing the safety mechanism described above, while saving object duplication and updating time. The other virtual methods need not be

[20] What happens is that incorrect pvm_upkXXX's of the wrong sizes are performed on the memory buffer, resulting in catastrophic failure.

specified, and if called, C++ will default to the `virtual_messages` class member function which are empty.

Sending and receiving messages are handled by `ptask's help_send_msg`, `help_send_main`, `help_nrecv_msg`, and `help_send_main` member functions. They allow for exacting, though complex, control of how a message is to be sent and received. Though these ptask member functions can be used directly, it is advisable to add a member function that simplifies the send and receive interface in the user defined message class. Another idea is to create an encapsulating class that extends the functionality of `ptask` send and receive functions. The `pmsg` classes are examples of this where `mcast` sends multiple copies of itself, while `brecv` provides the functionality of a blocking receive.

The following is an example of a class modified to be a PLWP message. `lwp_ID` is edited for clarity, leaving out the creator, reader, and writer member functions. `It` is used in PLWP as a thread identification object containing information on the Abstract ID and the Thread ID.

```
class lwp_ID : public virtual_message {
  /* note virtual_message is a public base class */
  int iAbstractID;    /* user defined Task ID */
  int iThreadID;
 public:
  lwp_ID(int AbstractID, int ThreadID);  /* with search for TaskID*/
  lwp_ID();
  int AbstractID() const;
  int ThreadID() const;
  /* virtual_message methods */
  void pvmsg_pack() {
    pvm_pkint(&iAbstractID, 1, 1);        /* PVM data packing calls */
    pvm_pkint(&iThreadID, 1, 1);
  }
  virtual_message* pvmsg_unpack() {
    pvm_upkint(&iAbstractID, 1, 1);       /* PVM data unpacking calls */
    pvm_upkint(&iThreadID, 1, 1);
    return 0;                             /* returns nothing */
  }
  virtual_message* pvmsg_replicate() {
    lwp_ID* next = new lwp_ID(*this);  /* editted out this creator */
    return next;
  }
  int pvmsg_size() {
    return sizeof(lwp_ID);
  }
  virtual_message* pvmsg_copy(virtual_message *) {
```

```
    lwp_ID* msg  = (lwp_ID *) message;
    iThreadID = msg->iThreadID;
    iAbstractID = msg->iAbstractID;
    return 0;                            /* returns nothing */
};
```

Example 3. lwp_ID.h Code

4.2 Virtual_userdef

Objects of this class provide scheduling for threads, and save and restore shared-contexts. Each thread is assigned a scheduler-object. When `ptask` needs one of these context-switching services, it calls upon the current thread's scheduler to handle it. In the fashion described in section 4, a class can be used as a scheduler if it is derived from `virtual_userdef` virtual-specification class.

The primary function of the scheduler is thread selection. `pvu_context_switch` is called upon to handle this. In trying to decide the next thread, it can access information in `pTask` such as the thread table `plwpTB`, the main thread through `pMain` pointer, and the current thread through `pCurrent` pointer. The thread table `plwpTB` does not include the main thread, which allows the thread table to act as a circular list for only user-defined threads. This makes it possible to use the hash table's `next` call to select the next-thread and guarantee it to be user-defined. Information about the place where the scheduler is called, is passed to `pvu_context_switch` as an integer `cond` parameter. The `cond` codes for `ptask`, are described in "pdefs.h.". `cond` codes can also be passed as a parameter of `plwp_yield` to the scheduler so that the user code can communicate with the scheduler. QuickThreads requires the next thread to be other than the current executing thread. If this is not satisfied, QuickThreads will crash the program, which makes checking whether the next thread is the current thread a good idea. When a thread is found, it should be returned by the scheduler to `ptask`. `ptask` subsequently context switches to it. If the scheduler is unable to find another thread, it can return `0`. `ptask` will try to accommodate by not context switching, but may crash in doing so, as in the case of `plwp_abort`. If synchronization and control of threads on other tasks is desired, the scheduler must take care of this by synchronizing the other tasks' schedulers and communicating with them. This is described in the following paragraph.

Coordinating thread scheduling between tasks requires synchronization. First there must be a prearranged hierarchy that specifies a task to be the controller like the master in the master-slave(s) hierarchy. Borrowing this example's terminology, the master task has an alarm clock that preempts the current thread which, upon expiring, calls `plwp_yield from` SIGALRM interrupt handler. In selecting the next thread `plwp_yield` calls the current-thread scheduler in preparation for the context switch. Abbreviating the "master task current-thread scheduler" to "master scheduler", the master scheduler then determines the next thread for itself and its slaves. It initiates a preemptive context-switch in its slaves via the SIGUSR1 signal in a process described in the next paragraph. The slave tasks are interrupted, and control is passed to their interrupt handler. The master scheduler tells the slave handler to execute the slave's current-thread's scheduler. This is done by sending a `mmsg` integer message with the three parts of the destination specified as the slave task, wildcard (any) destination thread, and USR1HANDLER_MSG tag (from "pdefs.h"). The integer value of the message is CONTEXT_SWITCH which indexes the slave's interrupt subhandlers to `plwp_yield` which is called. `plwp_yield` then calls the slave's current-thread scheduler. At this point, the master scheduler and the slave scheduler are synchronized, and can engage in further communication to determine the next thread for the slave task. When done the master scheduler resets the alarm clock using `setitimer` and returns the master's next-thread, while the slave scheduler returns the slave's next-thread.

PLWP provides two tools to reduce the complexity of using signals and interrupts. First by default all PLWP tasks have a SIGUSR1 interrupt handler that can service asynchronous requests by running subhandlers. This mechanism services events like updating name binding after a task is spawned, or providing task status information. If the user wants to add asynchronous services to PLWP, it is recommended that this be done through the SIGUSR1 handler mechanism for consistency. After the client sends the

SIGUSR1 signal to the server, it must send a integer mmsg message (described in the previous paragraph) to select a service viz. a subhandlers. The second tool, abstracts the complexity of sending a signal and message into the usr1Sig class. All the user need do is create a usr1Sig object with an integer representing the request, and send the usr1Sig object to the intended destination.

Variations on the task control hierarchy are possible. Gang scheduling can be implemented as an "intelligent" version of the master-slaves hierarchy given above. One improvement is to load balance the contexts, and select the context with the most unread messages. The ptask member function scan_unread_msg updates for each thread in that task, characteristics about unread messages. Upon finishing, the member variables plwp::GetUnreadMsgCount and plwp::GetUnreadMsgSize represents the number of unread messages and the total amount of memory in bytes the unread messages represent. Another statistic is the number of context switches performed which is given in plwp::GetContextSwitch.

The following example is a "round robin" cooperatively context-switching scheduler which PLWP uses as the default main thread scheduler. The class keeps a static, meaning only one exists per task, Thread ID variable. This is used to keep track of the current thread and insure that the next thread is other than the current thread. Some code is deleted to make the example clearer.

```
/* default context switcher */
/* this version does not save and restore contexts */
class cooperative_userdef: public virtual_userdef {
  static plwp* CU_Current;   /* default 0 */
  static int CU_key;         /* default 0 */
 public:
  cooperative_userdef() {
    /* setup user context */
  }
  /* context switcher */
  plwp* pvu_context_switch(ptask *GlobalTask, plwp *RecommendedTh,
                           int Cond) {
    plwp* newThread;
    if (GlobalTask->plwpTB->length() >= 2) {
      /* need at least two threads */
      GlobalTask->plwpTB->find(CU_key);
```

```
      if (!GlobalTask->plwpTB->next())
      goto foobar;
      /*      do (CU_Current ==
       *      GlobalTask->plwpTB->safe_value(newThread))); */
      GlobalTask->plwpTB->safe_key(CU_key);
      GlobalTask->plwpTB->safe_value(newThread);
      CU_Current = newThread;
      return newThread;
    }
    else {    /* can't find a thread... handle error */
    foobar:
      if (GlobalTask->pCurrent==GlobalTask->pMain)
      return 0;
      else
      return GlobalTask->pMain;
    }
  }
  int pvu_size() {
    return sizeof(cooperative_userdef);
  }
  void pvu_string(string& StrStuff) {
    StrStuff || " cooperative_userdef- CU_key: " || CU_key;
  }
  ~cooperative_userdef() {}
};
```

Example 4. cooperative_userdef.h Code

5 Implementation Details

This section covers the consequences of this implementation. The first part covers which platforms PLWP can run on, the second covers known bugs, and the third part covers limitations and possible solutions.

5.1 Platforms Supported

PLWP currently supports Sun SPARC. It should be easily ported to other platforms that have the needed UNIX system calls, and QuickThreads supports. The ubiquitous PVM is not a constraint, as it only requires a rebuild of the library to make it linkable for almost any UNIX platform.

PLWP requires two POSIX compliant[21] UNIX system calls `sigaction` and `sigprocmask`, and a BSD call `setitimer`. The manner that the POSIX calls are used are backwards compatible with their BSD counterparts– so if the POSIX calls are swapped with the BSD calls and masked properly from interrupts, PLWP can run on BSD[22] UNIX.

As listed above, QuickThreads is available for: Intel 80x86, KSR-1, KSR-2, 88000, MIPS Rx000, Sun SPARC, DEC VAX, DEC Alpha AXP, and HP-PA platforms. The notable missing architecture is IBM POWER (PowerPC). If a missing architecture is needed, there is documentation on extending QuickThreads. Another option is to use a different thread package which is done by swapping out the current QuickThread `plwp` class with another `plwp` class using the new thread package.

5.2 Bugs

Two functional bugs are known. First PLWPv0.3 alpha is believed to be memory leaky because of loose memory management in the message-passing classes. Memory usage was determined by using the "top" UNIX command where I observed for one test program, constant memory resource growth. Although this test program "crashed"

[21] This may be System V system calls in which case many more UNIX operating systems can be used.
[22] PLWPv0.1 was implemented using BSD calls.

because of extreme message-passing loading[23], other test programs with expected message-passing loading ran without problems. Thus it is believed that PLWP should be safe for its intended use as a thread-scheduling package. The second bug is that `ptask::scan_unread_msg` cannot always determine the number of unread messages sent between Tasks. This is because PVM does not immediately update the unread message queue after a message is sent. Each time `pvm_nrecv` is called to probe the message queue, one unread message is added to the queue until all message that have been sent but not received are in the queue. Thus several calls to `ptask::scan_unread_msg` are likely necessary to update the unread message count.

5.3 Limitations (aka. Future Work)

This section describes limitations caused by implementation choices. The range of tags, threads, and code identifiable by the message-passing `ptask` functions are 0-2047 tags, 0-127 threads ID's, and 0-15 code's, setting a practical limit on the number of messages and threads possible. This is caused by truncation of the message tags and ID's from sixteen bytes of data[24] into four bytes. One possible solution is to modify the message to contain the tags and ID's in the buffer as part of the packed message. Under the new scheme, a message is received, partially unpacked to determine the tags and ID's, forwarded to the correct destination thread, and stored until a PLWP receive call is made. The second limitation is PLWP's lack of thread synchronization primitives. Barriers and monitors are not implemented because initially it was believed unnecessary for PLWP's function as a thread scheduler. This maybe added in the future.

---

[23] It maybe that PVM's message buffers overflowed as the receive side could not keep up the asynchronous sends.

[24] Source and Destination Thread ID, Message Tag, and Code.

6 Conclusion

PLWP provides support for experimenting with threads and thread-scheduling over the PVM environment. There are several key ideas which make this project different from other mutli-threaded PVM packages. First a specification method called the virtual-specification class is introduced, which defines an interface for run-time selectable member functions. Second, PLWP separates and encapsulates the scheduling mechanism into user-defined objects which are assigned at run-time. This allows the user to create his own scheduler as needed, and to swap the schedulers during run-time. The third idea is to provide a mechanism for sending C++ objects over PVM in a simple manner with minimal additional code. Fourth, messages between threads on the same task are send in identical fashion to messages between tasks. These ideas, hopefully, will simplify the development of thread-schedulers, and enable further work towards understanding distributed thread-scheduling-strategies.

7 Bibliography

E. Ackaouy. Multiplexing PVM for Multi-Threaded Tasks. Graduate Class Paper, 13 May 1994.

E Felton and D. McNamee. NewThreads 2.0 User's Guide. NewThreads package documentation. 24 August 1992.

J. Hoe. Effective Parallel Computation on Workstation Cluster with a User-level Communication Network. Master Thesis, MIT, 1994.

Geist, A et al. PVM 3 User's Guide and Reference Manual. Tech Report ORNL/TM-12187, Oak Ridge National Laboratory, May 1993.

D. Keppel. Tools and Techniques for Building Fast Portable Threads Packages. Tech Report UWCSE 93-05-06, University of Washington, 1993

B. Liskov and J. Guttag. Abstraction and Specification in Program Development. MIT Press: Cambridge Massachusetts, 1986.

B. Stroustrup. The C++ Programming Language. 2nd Ed. AT&T, June 1993.

V. Sunderam. PVM: A Framework for Parallel Distributed Computing. Tech Report, Emory University. Available at mathcs.emory.edu under directory /pub/vss

TPVM: A Threads-Based Interface and Subsystem for PVM. Draft. 28 June 1994. Contact A. Ferrari for further details.

P. Winston. On To C++. Addison-Wesley: Reading Massachusetts, 1994.

Appendix

ptask plwp_abort()

Synopsis

```
void  ptask::plwp_abort()
```

Discussion

This kills the current thread. It then context switches to a new thread chosen by the aborted thread's scheduler. `plwp_abort` will also terminate the main thread, though PLWP behavior is undefined afterwards.

ptask plwp_alive()

Synopsis

```
int  info  =  ptask::plwp_alive(int  AbsID)
```

Parameters

AbsID - a task's abstract ID (integer)

info - if this task recognizes a task via AbsID then it returns true (1), else false (0)

Discussion

This can determine whether a task has announced that it is alive. Before the announcement, if an unannounced `AbsID` is used, PLWP's behavior is undetermined. Thus `plwp_alive` can be used to determine whether the `AbsID` is valid.

ptask plwp_create_lwp()

Synopsis

```
int info = ptask::plwp_create_lwp(int ThreadID,
              virtual_userdef *UserPtr, qt_userf_t
              *f, void *arg1, int
              StackSize=0x100000)
```

Parameters

ThreadID - a unique integer thread identifier. This number is limited from 1-127. Note 0 is main Thread ID.

UserPtr - a pointer to the thread scheduler. Each thread has its own scheduler.

f - a pointer to a (non-method) function of the form qt_userf_f which is starting point of the thread.

arg1 - a void pointer that is the argument to f

StackSize - the integer allocated area of memory for thread stack. If not specified this is defaulted via C++ to 0x100000 (arbitrarily size).

info - returns 1 on success, and -1 if ThreadID is 0.

Discussion

This creates, and initializes a thread. Upon return of `plwp_create_lwp`, the thread is ready to run. The argument `arg1` is passed to `f`. The scheduler `UserPtr` is used by PLWP when context switching from this thread. When the thread returns from `f`, meaning it has finished, `plwp_abort` is called.

typedef `void *(qt_userf_t)(void *u)`

ptask plwp_exit()

Synopsis

```
void ptask::plwp_exit()
```

Discussion

This kills the task including all of its threads.

Synopsis

```
virtual_message*  data  =
        ptask::help_nrecv_main(virtual_messa
        ge* msg,  int  Tid,  int  ThreadID,  int
        msgtag,  int*  cc,  int  code=1)
```

Parameters

data - either 0 if PVM behavior, or pointer to a virtual_message object

msg - a virtual_message object meaning it has virtual_message as a super class, and the necessary virtual functions

Tid - integer PVM for source task or wildcard -1.

ThreadID - integer source thread name (0-127) or wildcard -1.

msgtag - integer tag (0-2047) or wildcard -1.

code - integer msg code (0-15) default to 1.

cc - PVM condition code

Discussion

This ptask helper procedure receives PLWP main messages. The source is specified by a `Tid`, `ThreadID`, `msgtag`, and `code` integers where `AbsID`, `ThreadID`, and `msgtag` can be specified by a wildcard. A source thread can be specified as itself, a thread on the same task, or a thread on another task. Depending on the behavior of the virtual message, it may put the data in the `msg` or return the data.

Synopsis

```
virtual_message* data =
        ptask::help_nrecv_msg(virtual_message
        * msg, int AbsID, int ThreadID, int
        msgtag, int* cc, int code=0)
```

Parameters

data - either 0 if PVM behavior, or pointer to a virtual_message object

msg - a virtual_message object meaning it has virtual_message as a super class, and the necessary virtual functions

AbsID - integer Abstract ID for source task or wildcard -1.

ThreadID - integer source thread name (0-127) or wildcard -1.

msgtag - integer tag (0-2047) or wildcard -1.

code - integer msg code (0-15) defaulted to 0.

cc - PVM condition code

Discussion

This ptask helper procedure receives PLWP messages. The source is specified by a `AbsID`, `ThreadID`, `msgtag`, and `code` integers where `AbsID`, `ThreadID`, and `msgtag` can be specified by a wildcard. A source thread can be specified as itself, a thread on the same task, or a thread on another task. Depending on the behavior of the virtual message, it may put the data in the `msg` or return the data.

Synopsis

```
int cc = ptask::help_send_main(virtual_message*
             msg, int tid, int ThreadID, int
             msgtag, int code=1)
```

Parameters

msg - a virtual_message object meaning it has virtual_message as a super class, and the necessary virtual functions

tid - integer PVM Tid for destination task.

ThreadID - integer destination thread name (0-127).

msgtag - integer tag (0-2047).

code - integer msg code (0-15) defaulted to 1.

Discussion

This ptask helper procedure sends PLWP main messages. The destination is specified by a `Tid`, `ThreadID`, `msgtag`, and `code` integer. A destination is a thread that can be specified as itself, a thread on the same task, or a thread on another task. Note that a wildcard -1 can be specified for destination `ThreadID`, which means that any thread in that task is the intended recipient.

ptask help_send_msg()

Synopsis

```
int cc = ptask::help_send_msg(virtual_message* msg,
                 int AbsID, int ThreadID, int msgtag,
                 int code=0)
```

Parameters

msg - a virtual_message object meaning it has virtual_message as a super class, and the necessary virtual functions

AbsID - integer Abstract ID for destination task.

ThreadID - integer destination thread name (0-127).

msgtag - integer tag (0-2047).

code - integer msg code (0-15) defaulted to 0.

Discussion

This ptask helper procedure sends PLWP user messages. The destination is specified by a `AbsID`, `ThreadID`, `msgtag`, and `code` integer. A destination is a thread that can be specified as itself, a thread on the same task, or a thread on another task.

ptask plwp_print_all()

Synopsis

```
void ptask::plwp_print_all(string& Str)
```

Parameters

Str - a string object that is updated with plwp status

Discussion

`plwp_print_all` returns a string with a short version of this task's status, and all its childrens' task status. Included is information on task state, thread state, and task name bindings.

ptask plwp_print_current()

Synopsis

```
void  ptask::plwp_print_current(string&  Str)
```

Parameters

Str - a string object that is updated with plwp status

Discussion

`plwp_print_current` returns a string with a long version of this task status. Included is information on task's state, threads information (long version), spawned tasks ID, and task name bindings.

ptask scan_unread_msg()

Synopsis

```
void  ptask::scan_unread_msg()
```

Discussion

This procedure can be used by a load balancing scheduler to update the number and size of unread messages for each thread. Messages when sent, but not received, are considered to be unread. Each thread has two variables counting the number of unread messages waiting, and the size of the unread messages in bytes. This procedure updates those counters. There is a slight message size difference between messages sent between tasks via PVM and within the same task.

Note: There is a bug. PVM doesn't update its unread message buffer after a message is sent. The solution is to call a series of dummy non-blocking PVM receives that update the buffer until all outstanding messages are in the unread message buffer. Each one of these "probes" updates the buffer with one additional unread message. It was not known whether this fix was needed so scan_unread_msg has not been updated. This bug does not affect scan_unread_msg ability to quantify within-task unread messages.

Synopsis

```
int info = ptask::plwp_spawn_task(char* Task, char
                  ** argv, int* abstractids, int*
                  childtids, ntask=1, int flag=0,
                  char* where=0)
```

Parameters

Task - executable name (file name) of task to be spawned. The path is modified by PVM's ep (executable path) environment variable.

argv - task parameters, which is passed to pvm_spawn without modification.

abstractids - pointer to an array of integers representing the task's Abstract ID name.

childtids - pointer to an array (must be allocated) with PVM Tids stored in locations corresponding the abstractids.

ntask - number of tasks to be spawned. If not specified, this is defaulted via C++ to be 1.

flag - PVM spawn options. See the PVM manual. This is defaulted to be 0,

where - a char pointer which is an argument to the flag options. This is defaulted to be 0.

Discussion

This spawns a PVM task (or tasks) that is assumed to be incorporated with PLWP. Many of the options available for `pvm_spawn` are allowed here, and are accessible via the `flag` and `where` parameters (See PVM documentation). Each task to be spawned is assigned a unique Abstract ID (`AbsID`) via the `abstractids` integer array. This `AbsID` can be given before compilation in source code, and is used by PLWP to find the PVM Tid. If the users wants to deal with PVM directly, he can find the Tid that corresponds to the `AbsID`, in `childtids` integer array at the same index. If there are any error messages (given as PVM codes) they are stored in `childtids`.

ptask plwp_start()

Synopsis

```
void ptask::plwp_start(virtual_userdef *UserPtr=0)
```

Parameters

UserPtr - a pointer to the per thread, thread scheduler.

Discussion

This is the mechanism to start context switching threads, and turns on the context switching "switch". The "switch" is a hack to eliminate the need for barriers by not allowing pre-emption interrupts from other tasks to be acted upon if it is turned "off." A scheduler for the main thread is passed via `UserPtr`. Note if no scheduler is passed, then by default a "round robin" scheduler is used.

ptask plwp_yield()

Synopsis

```
int info = ptask::plwp_yield(int flag=YIELD_SWITCH)
```

Parameters

flag - This tells plwp_yield the circumstance for which it is context switching.

info - This returns 1 if context switch successful, or some negative error code if context switching was not possible.

Discussion

This procedure cooperatively context switches, by giving control to PLWP which finds a new thread using the current thread's scheduler, and context switches to the new thread. Note that PLWP, and the scheduler runs with the current thread's ID, a potential source of confusion for the source and destination thread ID's. The flag is optional, and is defaulted to YIELD_SWITCH. The flag is passed to the context switcher's `cond` parameter.

base_hash next()

Synopsis

```
Boolean bool = base_hash::next()
```

Parameters

bool - boolean (int)

Discussion

If entries exist in the queue, `next` increments the hash table pointer to the next entry in the queue and returns TRUE. If no entries exist in the queue, `next` returns FALSE.

base_hash prev()

Synopsis

```
Boolean  bool  =  base_hash::prev()
```

Parameters

bool - boolean (int)

Discussion

If entries exist in the queue, `prev` decrements the hash table pointer to the previous entry in the queue and returns TRUE. If no entries exist in the queue, `prev` returns FALSE.

base_hash top()

Synopsis

```
Boolean  bool  =  base_hash::top()
```

Parameters

bool - boolean (int)

Discussion

If entries exist in the table, `top` changes the hash table pointer in the queue to the "top" entry, and returns TRUE. If no entries exist in the queue, `top` returns FALSE. The "top" of the queue is invariant, and can be used as a starting point for iterators.

plwpTable find()

Synopsis

```
Boolean bool = find(const int& key)
```

Parameters

bool - boolean (int)

key - integer key.

Discussion

This find the entry with the same key as `key` parameter. If the key is found, `find` sets the hash table pointer to that entry, and returns TRUE. Otherwise it returns FALSE.

plwpTable get()

Synopsis

```
Boolean bool = get(const int& key, int& val)
```

Parameters

bool - boolean (int)

key - integer key.

val- integer value.

Discussion

This finds the entry with the same key as `key` parameter. If the key is found, `get` sets the hash table pointer to that entry, reads the entry's value into `val`, and returns TRUE. Otherwise it returns FALSE.

plwpTable put()

Synopsis

```
Boolean bool = put(const int& key, const int& val)
```

Parameters

bool - boolean (int)

key - integer key.

val- integer value.

Discussion

This finds the entry with the same key as `key` parameter. If the key is found, `put` sets the hash table pointer to that entry, writes `val` into the entry's value, and returns FALSE.

Otherwise it creates a new entry with key `key` and value `val`, sets the pointer to this entry, and returns TRUE.

plwpTable safe_key()

Synopsis

```
Boolean  bool  =  safe_key(int&  key)
```

Parameters

bool - boolean (int)

key - integer key.

Discussion

If the hash table pointer references an entry, `safe_key` reads the entry's key into `key`, and returns TRUE. Otherwise it returns FALSE.

plwpTable safe_value()

Synopsis

```
Boolean  bool  =  safe_value(int&  val)
```

Parameters

bool - boolean (int)

val - integer value.

Discussion

If the hash table pointer references an entry, `safe_value` reads the entry's value into `val`, and returns TRUE. Otherwise it returns FALSE.

Synopsis

```
send_msg::send_msg(int*  intdata,  int  size)
send_msg::send_msg(void*  data,  int  size)
send_msg::send_msg(virtual_message*  object)
send_msg::send_msg(int  Word)
```

Parameters

intdata - int pointer to a buffer. size is in terms of int.

size - buffer size in terms of the data type.

data - void pointer (byte) to a buffer. size is in terms of bytes.

object - pointer to object.

word - integer data.

Discussion

This is the creator for send_msg class. Using the C++ operator overloading feature, a set of creators can be used to create send_msg's which encapsulate different data types.

send_msg send()

Synopsis

```
int info = send_msg::send(lwp_ID to, int tag)
int info = send_msg::send(int AbsID, int thread,
                    int tag)
```

Parameters

to - a lwp_ID representing a particular thread on a particular task

tag - a user defined integer that can be used or identification. The allowable range is 0-2047.

AbsID - non-negative integer task's abstract ID.

thread - integer thread ID. Range is from 0-127.

info - return the send status in terms of PVM codes and: MSG_CANT_SEND_TO_SELF, MSG_CODE_NO_PLWP.

Discussion

Send message will send the encapsulated data to the destination provided by the destination arguments. One version of send takes as the destination argument lwp_ID `to`, while the other version takes int `AbsID` and int `thread`. The results are identical. Tag usage is user defined, but it is strongly suggested tag be used for type differentiation.

send_msg mcast()

Synopsis

```
int info = send_msg::mcast(lwp_ID to[], int size,
            int tag)

int info = send_msg::send(int AbsID[], int
            thread[], int size, int tag)
```

Parameters

to - a lwp_ID array of all the intended message recipients

AbsID - an integer array of the Abstract ID's of the intended message recipients

thread - an integer array of the thread ID's of the intended message recipients. The values corresponds to Abs ID's of the same index, and are limited to the range 0-127.

size - size of arrays

tag - a user defined integer that can be used for identification. The allowable range is 0-2047.

info - return the send status in terms of PVM codes and: MSG_CANT_SEND_TO_SELF, MSG_CODE_NO_PLWP. This returns the condition code of the last send.

Discussion

`mcast` will send multiple copies of the encapsulated data to the destinations provided by destination arguments. One version of send takes an array of `lwp_ID`'s, and the other takes two int arrays of `AbsID` and thread types. It sends each destination a copy of the data. `tag` usage is user defined, but it is strongly suggested `tag` be used for type differentiation.

recv_msg recv_msg()

Synopsis

```
recv_msg::recv_msg()

recv_msg::recv_msg(virtual_message* object)
```

Parameters

object - pointer to object.

Discussion

This is the creator for `recv_msg` class. The second version allows the created `recv_msg` to receive a `virtual_mesage` object as the message. In this case `recv_msg` needs the `virtual_mesage` object for its PVM unpacking method, thus requires a (empty) copy of the `virtual_mesage` object during creation.

recv_msg nrecv()

Synopsis

```
int info = recv_msg::nrecv(lwp_ID from, int tag)
int info = recv_msg::nrecv(int AbsID, int thread,
                int tag)
```

Parameters

from - a lwp_ID representing the source thread in a particular task

tag - a user defined integer that can be used or identification. The allowable range is 0-2047 or -1 wild card.

AbsID - non-negative integer task's abstract ID or -1 wild card.

thread - integer thread ID ranging from 0-127 or -1 wild card.

info - return the send status in terms of PVM codes and: MSG_CODE_NO_MSG (failure) and MSG_CODE_FOUND (intra-thread success).

Discussion

This is the non-blocking receive method for `recv_msg`. If the `recv_msg` has a `virtual_message` object encapsulated, upon return from this procedure and if a message has been received, the object contains new data. For other possible data types, a copy is created which can be accessed by selector member functions.

recv_msg brecv()

Synopsis

```
int info = recv_msg::brecv(lwp_ID from, int tag)
int info = recv_msg::brecv(int AbsID, int thread,
                           int tag)
```

Parameters

from - a lwp_ID representing the source thread in a particular task

tag - a user defined integer that can be used or identification. The allowable range is 0-2047 or -1 wild card.

AbsID - non-negative integer task's abstract ID or -1 wild card.

thread - integer thread ID or -1 wild card.

info - return the send status in terms of PVM codes and: MSG_CODE_NO_MSG (failure) and MSG_CODE_FOUND (intra-thread success).

Discussion

This is the blocking receive method for `recv_msg` and is interruptable. The member function will only return if a message matching its parameters is found. If the `recv_msg` has a `virtual_message` object encapsulated, upon return from this procedure, the object contains new data. For the other possible data type, a copy is created which can be accessed by selector member functions.

recv_msg (inherited) get_data()

Synopsis

```
int* data = get_data(int* size)
```

Parameters

data - void pointer pointing to memory buffer (bytes)

size - gets updated with the size of the buffer in bytes.

Discussion

This returns a pointer to a memory buffer allocated during receive and contains the data. If empty, it returns 0.

recv_msg (inherited) get_int_array()

Synopsis

```
int*  data  =  get_int_array(int*  size)
```

Parameters

data - void pointer pointing to a integer array.

size - gets updated with the size of the buffer in terms of integers.

Discussion

This returns pointer to a integers array received. If empty, returns 0.

recv_msg (inherited) get_word()

Synopsis

```
int  word  =  get_word()
```

Parameters

word - encapsulated data.

Discussion

This returns the integer data received. If empty returns 0.

usr1Sig usr1Sig()

Synopsis

```
usr1Sig::usr1Sig(int  handlerTag)
```

Parameters

handlerTag- integer index into SIGUSR1's subhandler table.

Discussion

This creates a `usr1Sig` object. The object when sent will send a SIGUSR1 signal to the destination, requesting the service represented by `handlerTag`. See "pdef.h" for the integer values for services.

usr1Sig send()

Synopsis

```
void send(int destTask)
```

Parameters

destTask - integer Abstract ID of destination task .

Discussion

This sends the signal and service request message to task specified by `destTask`.

usr1Sig mcast()

Synopsis

```
void mcast(int destTask_array[], int number)
```

Parameters

destTask_array - integer array of destination task's Abstract ID.

number - number of destinations in destTask_array

Discussion

This sends the signal and service request message to tasks specified by `destTask_array`.

virtual_userdef pvu_context_switch()

Synopsis

```
plwp* nextTh virtual_userdef::pvu_context_switch(
                ptask* GlobalTask, plwp*
                RecommendTh, int Cond)
```

Parameters

GlobalTask - a ptask pointer.

RecommendTh - ptask may make a suggestion on the recommended thread (plwp pointer), otherwise this is zero.

Cond - This passes integer condition data of the context switching e.g.: START_SWITCH, YIELD_SWITCH, ABORT_SWITCH, SIGUSR1_SWITCH, YIELD_RECV_SWITCH.

nextTh - returns the next thread i.e. plwp pointer. It can return 0 if no thread was found.

Discussion

This is used by ptask as the scheduler. Given the above arguments, the scheduler should select the next thread to context switch to. The scheduler must not return the same thread as the one running. If it cannot avoid this (or some similar circumstance arises), it may return a 0. In this case the `ptask` will try not to context switch if possible though it may crash. If multiple tasks require synchronization during thread scheduling, the scheduler handles all signaling and communication. Any hierarchy is defined by the user in scheduler. pvu_context_switch also saves any user context from shared memory.

virtual_userdef pvu_restore()

Synopsis

```
void  virtual_userdef::pvu_restore()
```

Discussion

If this is specified, this member function restores any user context from shared memory.

virtual_userdef pvu_string()

Synopsis

```
void  virtual_userdef::pvu_string(string&  StrStuff)
```

Parameters

StrStuff - This parameter is modified to contain text information about the state of the virtual_userdef.

Discussion

This method is used by ptask to print out information about the `virtual_userdef` which is useful for debugging. For display consistency, the last line of the string should not be terminated with a newline. Also lines following the first should be indented two spaces to the right. The string appears in the `ptask::plwp_print_current()`.

virtual_message pvmg_copy()

Synopsis

```
void  virtual_message::pvm_copy(virtual_message*
                 source)
```

Parameters

source - copies this objects data into this object.

Discussion

This is used by PLWP to copy the contents of one `virtual_message` object into another. It was felt that messages should behave similarly to the way PVM unpack. To simulate this for inter-thread message-passing, this function is needed. The object and the source must be the same type. If they are not, this will likely crash the program. If the PVM like behavior is not prefered, one could easily make a dummy `pvm_copy` function.

virtual_message pvmsg_pack()

Synopsis

```
void  pvmsg_pack::pvmsg_pack()
```

Discussion

This procedure packs the object's data into a PVM message buffer. Since the details of managing the buffer are taken care of by ptask, all this procedure has to do is use the appropriate pvm_pkxxx call [Geist]. This paradigm is especially useful for deeply nested records viz C++ classes. If the object contains another `virtual_message` object, all one has to do is call the nested object's pvmsg_pack function.

virtual_message pvmg_replicate()

Synopsis

```
virtual_message*  obj  =
                  virtual_message::pvm_replicate()
```

Parameters

obj - returns a copy of this object (or 0 if replication is not desired).

Discussion

This is a virtual creator, and returns a duplicate of this object. This is used by PLWP to create a duplicate object which is inserted in the intra-task message buffer. The reason for this is to prevent the source and destination threads from having conflicts as the object exists in both threads. Of course other behavior can be given; `plwp_replicate` can return the pointer to itself to hasten message creation..

virtual_message pvmg_size()

Synopsis

```
int  size  =  virtual_message::pvm_size()
```

Parameters

size - returns the size of the message.

Discussion

This method should returns the size of the object, and is used by `ptask::scan_unread_msg` to calculate the size of the message.

Synopsis

```
virtual_message*  data  =
                    virtual_message::pvm_unpack()
```

Parameters

data - either 0 if PVM behavior, or pointer to a virtual_message object

Discussion

This unpacks the data from the current receive buffer into this or returns a created object. Data in the buffer is presented in the same order that it was packed, thus pvm_upkxxx calls should be used in the same order as their pvm_pkxxx counterparts [Geist]. Ptasks handles other details of PVM message handling.