FAST MULTI-THREADING ON SHARED MEMORY MULTIPROCESSORS

A THESIS SUBMITTED IN
PARTIAL FULFILMENT OF THE
REQUIREMENTS FOR THE DEGREE
OF BACHELOR OF SCIENCE (HONS.)

University of Malta
Department of Computer Science and A.I.

By
Joseph Cordina
June 2000

Abstract

This thesis implements a fast multi-threaded shared memory multiprocessor scheduler that runs on Linux machines. The main concern is efficiency in scheduling and in inter-thread and network communication, giving good performance when executing fine grain user level threads. Also emphasis was made to create a scheduler that is scalable with the number of processors and still provide transparency to the application developer.

This thesis starts off with a version of MESH that runs on a uniprocessor machine and modifies it into SMP-MESH, a version that runs on shared memory multiprocessor machines, the system being easily scalable to any number of processors. MESH is a tightly coupled fine grain user level thread scheduler providing very fast scheduling and communication rates. Our multiprocessor implementation allows the concurrent execution of user level threads, through the use of a shared run queue together with concurrent inter-thread communication. Shared resources exist that are protected through the use of spin-locks.

SMP-MESH implements all the advantages of parallel processing. The system caters for load balancing and well behaved idling. Also specific race hazards that might increase latency or lead to deadlock are investigated.

An investigation of the overheads introduced in the new implementation is presented. It is seen how a variety of benchmarks assess different parts of the scheduler and communication sections of SMP-MESH. Light is shed on the thread granularity limit beyond which contention for shared resources starts to influence performance adversely. This thesis shows that an SMP implementation of the uniprocessor MESH is the obvious step in the evolution of MESH resulting in considerable speedup while still maintaining all the fast techniques present originally.

Acknowledgements

I would like to express my gratitude to Dr. Kevin Vella for his patience and perseverance to make this project possible. I am quite aware of the many difficulties I made him face and for this I am eternally grateful. I would also like to thank Kurt Debattista for being an excellent partner in discussing practical matters about SMP implementations.

I am indebted to the University of Malta and especially the Computer Science and A.I. Department for providing all the necessary hardware and expertise to realise this work.

I would also like to sincerely thank my parents and my girlfriend without whose support I would surely not have pulled through so well. Lastly but not least I would like to thank all those who have been patient with me in this past year and those with whom I celebrated in more joyous times.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

The use of multiple threads of execution on modern operating systems that support multiprocessor architectures presents the possibility of highly efficient parallel execution. Thread scheduling on a uniprocessor machine gives apparent concurrency especially when using overlapping I/O operations. Yet this is only an impression of concurrency, since the scheduler is merely shifting rapidly from one thread to the next and no two threads are being processed in the same point in time. Such systems introduce added latencies over non-threaded schedulers due to context switching and preemptive interruption of execution. Making use of user level threads, that is threads that do not use kernel system calls, makes scheduling much faster than kernel level threads.

The next obvious step is to take several independent threads and execute them on different processors achieving true concurrency. One cost effective way to build a multiprocessor architecture is to place multiple processors on the same address and data bus, incurring much less cost than having more sophisticated interconnections between independent systems. Executing threads on different processors makes execution time of the whole task at hand shorter and making use of shared memory also reduces space complexity, even though care must be exercised to avoid overloading the shared resources.

This thesis implements a multiprocessor version of MESH. MEssaging and ScHeduling for Fine Grain Parallel Processing on Commodity Platforms (MESH) is a tight integration of a user-level thread scheduler with 32 levels of thread priority and a zero copy messaging system that enables efficient fine-grain parallel processing. It targets *Intel Pentium* processor-based Linux operating systems. Developed at CERN, it is designed to efficiently schedule threads on uniprocessor machines. Making use of several innovative techniques it provides very fast scheduling and inter-thread communication. This project ports MESH to a shared memory multiprocessor system.

Making use of a shared memory symmetric multiprocessing model, this new version of MESH, SMP-MESH, allows true concurrent execution of several user level threads. The extent of such concurrency in executing user level threads is only limited by the number of processors available on the system. SMP-MESH uses a minimum amount of memory space by accessing shared memory for code sections. A kernel level process runs on each processor so that each processor executes user level threads from a shared run queue. User level threads are not guaranteed to be run on the same processor all throughout their execution, rather threads may migrate between processors, automatically balancing processor workloads. This implementation enables concurrent inter-thread communication and also protects concurrent access to the network card. Full parallelism may be exploited apart from special areas where shared data is accessed and there is the possibility of data corruption. These areas are protected through the use of low latency spinlocks.

Starting off with a very fast user level thread scheduler such as MESH, SMP-MESH takes advantage of all the original techniques while incurring minimal additional overheads. Special care was taken to keep critical sections to a minimum size and also to keep newly added algorithms as fast as possible. Overheads were obviously introduced but these are well balanced by the advantages gained through the use of multiprocessors.

1.1 Organisation

This dissertation is split up into a series of logical sections. The first chapter introduces Shared Memory Multiprocessing together with a variety of complementary hardware and software techniques. It also introduces the logical hierarchy within the operating system on which our threads are implemented. The next chapter goes into the details of uniprocessor MESH with its fast scheduling techniques and several other design and implementation issues. It also gives a brief introduction to the MESH API presented to the application developer. In addition, it discusses the different run queue configurations available to SMP schedulers and also the various implementations of the spin lock synchronisation primitive.

The next chapter investigates several problems faced in implementing SMP-MESH and the techniques and algorithms used to tackle them. It also discusses the reason for certain implementation decisions whenever many design options were available. It also describes certain additions to the uniprocessor MESH that enable well-behaved execution of the scheduler. Chapter 5 then describes the MESH inter-thread communication system with its API and how this system has been ported on to the multi-processor.

Finally a series of benchmarks are introduced and described. A series of tests that were performed on the system are explained and each result discussed. The overheads added on to the uniprocessor version are investigated together with the effect of fine thread granularity on parallel execution efficiency. Some speedup values are also given.

Chapter 2

Shared Memory Multiprocessing

2.1 Introduction

Symmetric Multiprocessing (SMP) is a series of two or more processors combined together to form a single computer system. They are symmetric in that each processor has equal and fair access to each of the system's resources. Several processes are able to execute simultaneously, each executing on different processors and when each process makes part of a single task, this task will be completed in less time than on a uniprocessor machine. SMP removes the need to design faster processors to achieve faster execution. Yet multiple processor systems require the use of special hardware models, operating systems and programming constructs to achieve faster execution efficiently and without data corruption. Gottlieb gives the rather crisp definition of SMP as “A large collection of processing elements that can communicate and co-operate to solve problems fast” [1]. Several SMP configurations exist with the *tightly coupled shared memory* SMP system being the one mostly in use today. This system can be used with comparatively small changes to the overlying uniprocessor operating system and also has quite an attractive price tag. It consists of several processors inter-connected by a high speed bus and with every processor having access to a single area of memory. Each processor does not have an area of private memory apart from its cache. Support for SMP systems can be found in most common operating systems including the SunOS UNIX implementation [25].

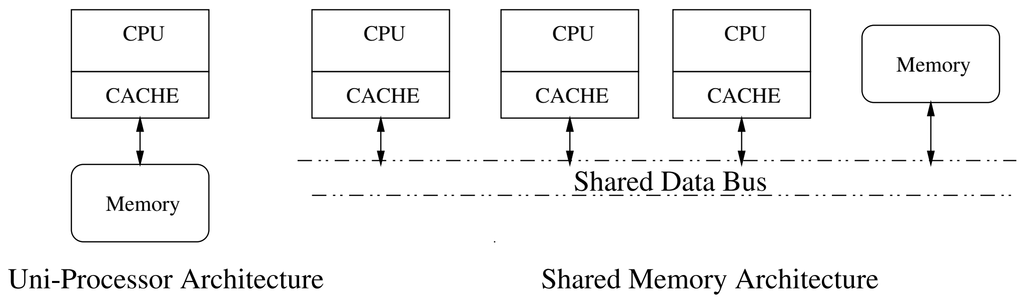


Figure 2.1: Typical Hardware Architecture

2.2 SMP Architecture

The modern generation of the SMP computational model consists of several processors executing concurrently and independently. Each Processing Element (PE) processes a stream of instructions that process some data. Under a widely used taxonomy, this is known as the Multiple Instruction Multiple Data architecture (MIMD) [1]. Shared Memory Processing (as seen by Treleaven [46]) is a conceptual view of MIMD where the same area of memory is accessed by the several PEs. Most commercial examples are instances of tightly coupled, shared bus, shared memory, uniform access multiprocessors with caching put to use in hiding memory latency and lowering bus traffic [51]. Many variations on these exist which are able to give better performances at an obvious increase in market price. These include Ring-Based Processors (with each processor having access to some memory that is split up into a private part and a part which is shared between all the processors, an implementation of which is **MEMNET**) and Switched Multiprocessors (which use a number of communication buses and organize processors in a hierarchical fashion) [43]. A typical SMP machine topology can be seen in Figure 2.1 in comparison to a Von Neumann machine which is the most in use to date. One of the main problems in SMP is that all PEs have access to shared memory and simultaneous access to a common area of memory by more than one processor would lead to data corruption.

The *Intel Pentium* based Shared Memory architecture is a typical implementation of the SMP architecture. It is normally equipped with two to four processors accessing memory on a shared bus with certain features aimed to achieve good speedups while maintaining a low price tag.

2.3 Caches And Processor Instructions

Normally a small high speed access memory is attached in between the bus and the processor. It tends to be quite costly to manufacture and is called the cache. It uses the Principle of Locality that states that processes tend to reference storage in non-uniform, highly localized patterns [15]. In multiprocessor systems the cache has to be more sophisticated to cope for data which is being altered by another processor. Several implementations exist for cache coherence amongst which are [1]:

- Non-cacheable variables where shared variables are simply not cached,
- Write-Through where any writes to memory go all the way through to main memory. Uses write invalidation techniques where a flag signals other caches that part of their contents is out-dated,
- Write-Back where only the present cache is updated allowing memory to become stale, and
- Directory Based where a data structure is kept notifying the invalid entries in each cache.

Some systems (including the *Pentium II* processor based architectures) actually incorporate more than one strategy and one can specify how each area of memory is cache treated through the use of MTTRs (Memory Type and Range Registers) [6]. MTTRs is a hardware support for physically addressed contiguous regions giving total freedom of access strategies. The *Intel Pentium* processor is equipped with a two level cache, one being internal and the other external to the processor, each achieving varying degrees of access time.

In addition, the processor normally provides atomic primitives, the number and complexity of which dictate the data structures and associated algorithms that may be implemented in a non-blocking manner [51]. Of special significance are the atomic Test-And-Set, Compare-And-Exchange and Fetch-And-Store instructions. More complex instructions have been suggested by Segall and Rudolph [39] using Test-And-Test-And-Set which considers cache coherence. Also it has been suggested that the Operating System can provide higher level atomic instructions where hardware support is lacking, such as Compare-And-Swap where only Test-And-Set and Exchange are available [9]. The inclusion of the Test-And-Set instruction on board the *Intel* manufactured processors has started since the *386* processor. In addition a special prefix instruction, `lock`, gives exclusive use to the processor executing such an instruction to some area of memory. This instruction's scope is limited to one instruction following `lock` [45, 47]

2.4 SMP Programming

Gottlieb [1] defines three necessary constructs for parallel programming:

1. Define a set of subtasks to be executed in parallel,
2. Start and Stop their execution, and
3. Coordinate and specify their interaction while they are executing.

A subtask, also known as a thread of execution, is a series of one or more instructions which are expected to be executed in series and result in a well specified task to be performed. Having several threads running concurrently in memory, they are bound to try to access the same area of memory at the same moment in time. This can lead to non-deterministic behavior as seen in Figure 2.2.

An ever worse situation is when a cycle of resource dependencies is created which makes the system prone to deadlock as shown in Table 2.1. One can immediately see that if these two programs are run concurrently with each instruction terminating execution simultaneously, then both programs will block indefinitely waiting for the resource to be released. Some operating systems cater for this by

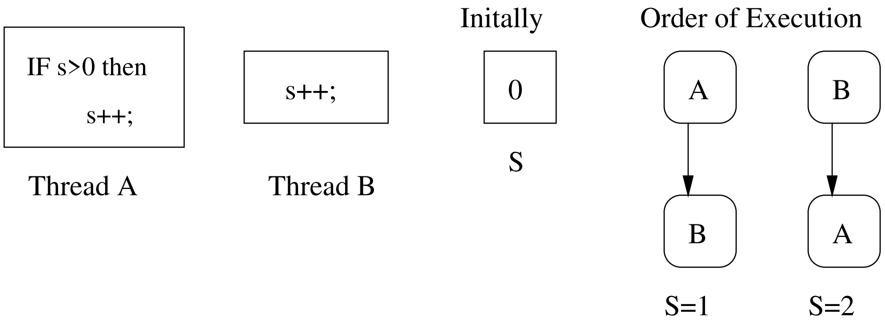


Figure 2.2: Non Deterministic Runtime Execution

trying to avoid the occurrence of deadlock, by inspecting a system to determine whether a deadlock occurred and by implementing a recovery approach when it happens [40]. A common way to catch out dead lock situations is by finding a cycle of dependencies.

To synchronise the access of resources, use is made of special hardware level support (discussed on page 7) to provide the following software constructs:

Semaphores Introduced by Dijkstra, it is an integer-valued variable that supports two basic operations P() and V() shown in Algorithm 2.4 [49]. It is assumed that these operations are performed atomically and they are normally supported through kernel based functions by the Operating System.

Spin-Locks A variable which regulates access to some resource. When the resource is required, a processing element checks this variable and if found set (resource locked), it simply waits until variable is reset (resource unlocked). When the variable is found reset, it is set and resource accessed. Upon usage of the resource, the lock is reset again. Refer to page 24.

A section of code which accesses a shared resource is known as a **critical section** and is normally protected (for example by the use of a **P()** call before

1	Obtain Resource A	Obtain Resource B
2	Obtain Resource B	Obtain Resource A

Table 2.1: Deadlock Prone Code

Algorithm 2.1 Semaphore Operations

```
Acquire semaphore
P(semaphore)
    while (semaphore = 0)
        repeat
    decrement semaphore by 1
Release Semaphore
V(semaphore)
    increment semaphore by 1
```

it is executed and a **V()** upon termination). This ensures that only one thread accesses the resource at any one time. A different semaphore is used per resource.

Parallelism is feared by many programmers since it introduces non-determinism in the execution of the code that makes execution patterns hard to predict and cope with. Also many shy away from it due to its inherent complexity to put it in a mathematically sound context. Normally a new programming approach has to be undertaken taking into considerations the division of certain tasks into manageable independent sub-tasks. One way to achieve this is through the use of multi-threading. A thread is a unit of execution that can be developed and executed independently of any other thread. Allocating a sub-task to each of these threads results in these threads executing the whole task in separate chunks. In a uniprocessor machine, the scheduler's task is to execute each thread in turn on the available processor. Placed into an SMP perspective, this results in a scalable situation where each thread will be executed by a different processor giving a decrease in total execution time. Increasing the number of threads, thus splitting the overall tasks into finer sub-tasks and increasing the number of processors will result in more concurrency and faster overall execution. Higher level abstractions for inter-thread communication can also be provided, hiding away the intricacies of synchronising the processors in the SMP machine.

2.5 Scheduling

Processor scheduling is the assignment of processes to processors to accomplish work [15]. It involves the creation, destruction and the fair execution of all processes on the system giving the impression of concurrency even on a single processor. The process of changing from one process to another consists of one process being stopped, selecting the next appropriate one to be handed to the CPU and then performing the change over to the new process All this is known as the **context switch**. This process can be preemptive where one process is interrupted and another process proceeds in the specified fashion, or self yielding where the process itself gives up execution or blocks on some input or output. Most common technique in use is Round Robin preemptive scheduling where each process gets to run for the duration of a specified time slice. A process can be of three types distinguished by its level of abstraction from the Operating System: **Kernel Level Processes**, **Light Weight Processes** or **User Level Processes**. Seen differently, these provide a sort of scheduling hierarchy [13].

2.5.1 Kernel Processes

A thread of execution created, controlled, monitored, and destroyed in kernel space. Kernel space is the set of functions provided by the operating system giving the caller exclusive access. It has total autonomy from other processes owning all the necessary data structures and has a high priority of execution. The kernel threads all have a unique identifier stored in the kernel and can be accessed by the Operating System on a one to one basis. They are able to block in the kernel, and can thus give up execution to other processes. Creation of such threads is in the order of milliseconds yet one has available a series of facilities to synchronise and communicate with other threads [51]. On the UNIX system a Kernel Process is created through the use of the **fork()** function call [42].

2.5.2 Light Weight Processes

A *Light Weight Process (LWP)* is a basic unit of CPU utilization and consists of a program counter, a register set and a stack space [40]. It shares with peer LWPs its code section, data section and operating system resources. Such processes are separately dispatched by the kernel, may perform independent system calls, incur independent page faults and run in parallel on a multiprocessor machine [36]. On the *Linux System* an LWP is the same as a kernel process. On an SMP machine, using the SunOS system, one is able to bind LWPs to specific processors. Extensions to the Linux kernel to allow such threads to be bounded to specific CPUs is being investigated, see [6, 28]. On the Linux System, an LWP is created by calling the **clone()** function which provides different levels of resource sharing between the threads. Other implementations of LWPs have been suggested [41]. Threads are also offered through the kernel as POSIX threads, a standard implementation in most UNIX operation systems implementing multi-threading [35, 36, 24].

2.5.3 User Level Threads

These are threads whose scheduler normally runs on a kernel thread or a light weight process. The operating system is unaware of such threads running [51]. Scheduling from one thread to another is done on the user level. The fact that, on scheduling operations, a vertical switch in the kernel is no longer necessary decreases the scheduling time drastically, this being the main advantage of such implementations. Preemptive scheduling on the user level can be achieved through the use of signals yet this still requires a kernel vertical entry. Thus most modern User Level Thread packages make use of *cooperative scheduling* where the thread either explicitly yields execution or the compiler enters automatically yield instructions in appropriate places. Low Latency Techniques for thread scheduling will be investigated in more detail in Chapter 2. Figure 2.3 shows the layout of a typical operating system which supports all the above process levels [51]. Note that this hierarchy of processes is not so well defined in most implementations, with work also being done to place user level threads on the kernel level even though at the dispute of some who state that these (the kernel level and the user level) are inherently not compatible [4, 30].

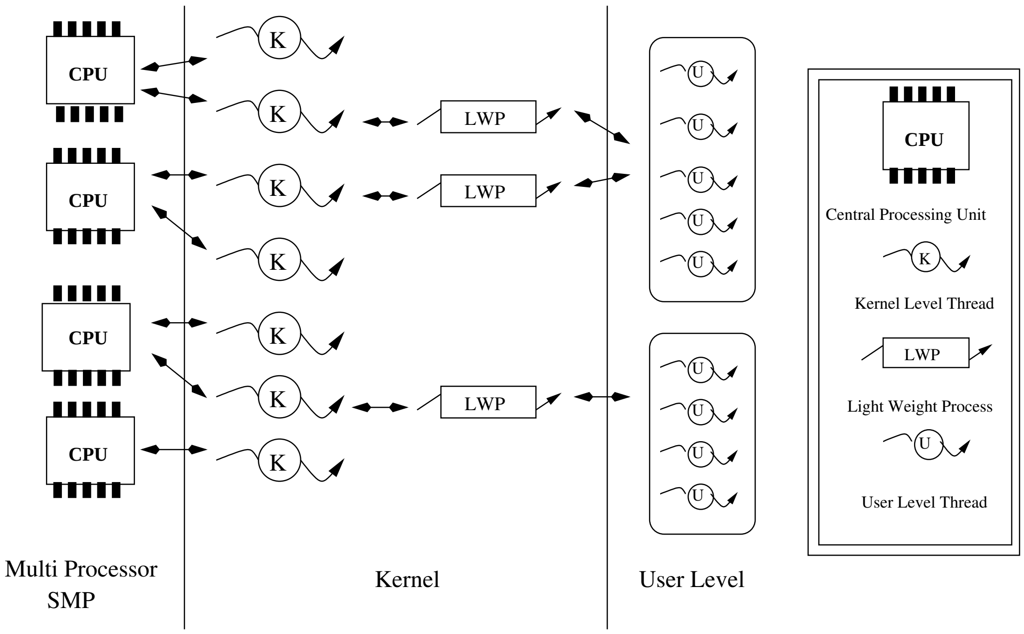


Figure 2.3: Two Level Scheduling

2.5.4 Further SMP Considerations

All the software techniques discussed so far could be applied to uniprocessor machines. An SMP machine merely requires that the scheduler at the kernel or at the user level cater for several situations such as *processor starvation* and *race conditions*. A processor when running a thread which is waiting to enter a critical section is, as such, doing no useful work. Also one cannot allow a situation when one processor is running a large number of threads while another is simply idling. One technique is implementing a system of well-behaved idling say by the process sleeping in the kernel with minimum superfluous wake-ups [52]. Whenever one processor gets too many processes allocated to it, a solution would be using a process called *migration*. This involves moving one process from one processor to another processors which is less busy. Yet the **Principle of Locality** should be kept in mind since one does not want to lose out on the advantages of an optimized processor cache. Also process migration takes up computation time. Processes who are all contending for the same resource should merely be run sequentially. Further considerations should be considered when a thread blocks in the kernel. This may lead to situations of deadlock.

In fact, SMP scheduling in the kernel can be summarized as an automatic task trying its best to maintain a well balanced system to obtain optimum performance.

Chapter 3

Low Latency Scheduling Techniques

3.1 Introduction

This chapter will discuss recent advances in techniques for fast-thread scheduling. This discussion will start off with an introduction to MESH, a fast thread package developed by M. Boosten at CERN [12]. An analysis of the main techniques used in MESH will follow. This discussion will also introduce other techniques which enable efficient SMP scheduling which we will apply to MESH in the forthcoming chapters.

3.2 MESH

MESH (MEssaging and ScHeduling for Fine Grain Parallel Processing on Commodity Platforms) is a tight integration of a user-level thread scheduler and a zero-copy messaging system that enables efficient fine-grained processing on commodity platforms with efficient support for fault tolerance [12]. It is aimed to be used in the parallel computer that constitutes the second stage trigger of the ATLAS detector, a hadronic tile calorimeter that measures the energy of charged and neutral particles, generated in the Large Hadron Collider (LHC). The LHC is a new hadron collider being built at CERN and expected to be functional by 2005. It is aimed to investigate phenomena in particle physics [11], namely the search for the Higgs Boson predicted by the *standard model* and particles predicted by *super-symmetry theory* [37].

Many thread package implementations are available [23, 18, 10], MESH being able to provide context switching times of 55ns on a uniprocessor machine and a throughput of 115MBytes/s with a CPU load of 6% on a Gigabit Ethernet card using jumbo frames. Such results are quite phenomenal compared to other packages yet one immediately notices certain aspects that had to be sacrificed, such as the fact that the programmer is forced to work hand in hand with the scheduler which does not provide the user abstraction level one is accustomed to. The success of MESH is based on:

- User-Level Scheduling,
- Active Context Saving,
- Inline Context Switching,
- User Level Hardware Access,
- Local Polling, and
- Reducing I/O Bus Accesses.

These will be discussed in the sections that follow.

3.3 Process Run Queues

A process run queue is often implemented as a structure containing a linked list of thread descriptors, organized in FIFO fashion, each of which contains the *context data* of each thread. The context of a thread is the minimum amount of data which would allow that thread to continue executing at the point where it was interrupted. This is usually the set of processor registers since it is the processor's registers that are corrupted when another process is executed. Whenever a new thread of execution is created, its descriptor is simply added to the queue. Then when a thread yields execution, terminates or blocks on some input or output, the scheduler takes over and chooses the next thread to run. While some schedulers remove the currently executing thread from the run queue, MESH chooses to skip this step and never removes the current thread for efficiency reasons. It was necessary to change this in our SMP-MESH implementation to avoid multiple processors all choosing the thread at the top of the run queue as the current thread. Several levels of run queue (32 levels in all) are implemented in MESH, each containing threads with a different priority of execution. MESH also makes the run-queue doubly linked to allow faster enqueue and dequeue operations of thread descriptors from the queue (refer to Figure 3.4). Only one pointer is held for each queue, pointing to the head of the queue. The enqueue operation consists of placing an element in front of the queue and making the head pointer point to this newly added element. The dequeue operation finds the last element using the head pointer and the links between the elements (it being doubly linked and circular) and removes this element. All links are adjusted accordingly in both operations.

On an SMP machine things complicate themselves since each processor accessing the run queue cannot safely alter it concurrently with other processors. We therefore have to define critical sections around portions of scheduler code (see page 8). Various run queue configuration options are available to the designer of an SMP thread scheduler.

The one investigated most thoroughly in this thesis is the **shared run queue**. One run queue exists and all new threads are merely placed upon this structure. This provides a centralized system yet moves away from any cache affinity. Also fine granularity causes problems in that the critical section is accessed quite often leading to idling processors (busy waiting at best). A shared run queue also provides automatic load balancing in that *processor starvation* never occurs. Processor starvation refers to the state when a processor spends most of its time waiting for things to do while all the other processors are full to the brim with jobs allocated to them

One can also have a **multiple run queue** structure where each thread is

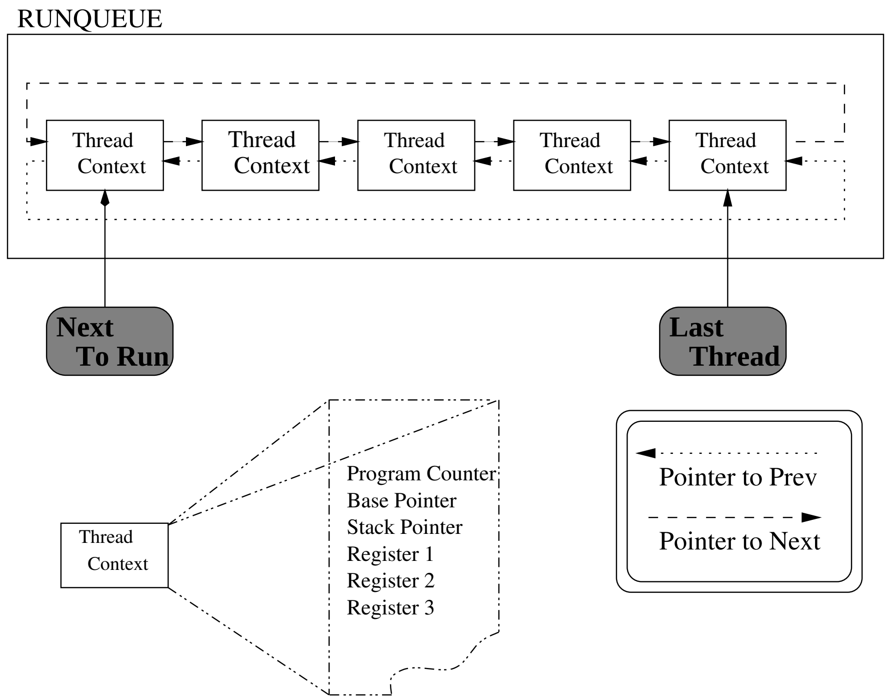


Figure 3.4: Circular FIFO Run Queue on a Uniprocessor Machine

created on the run queue belonging to the processor that created it or placed into another processor's run queue depending on its capacity and current work load. Careful allocation of user level threads to the run queues tackles *processor starvation*, yet only through a higher computational cost. Multiple run queues respect cache affinity yet in the case of a starving processor, migration tends to be quite costly.

A quite novel compromise has been investigated by Vella [51] with the use of **Batch Queues** (see Figure 3.5). In the figure some pointers have been left out to aid clarity. Here the run queue becomes a queue of batches, where each batch in turn consist of a queue of thread contexts that has a fixed limited size (further improvements would be to make this size variable). Now each processor is allocated one batch run queue and also an overflow run queue. There also exists a central shared queue of batches. Whenever a processor runs out of threads to run it brings in a new batch from the shared queue. When new threads are created they are placed in the current run queue and if the maximum count is exceeded, they

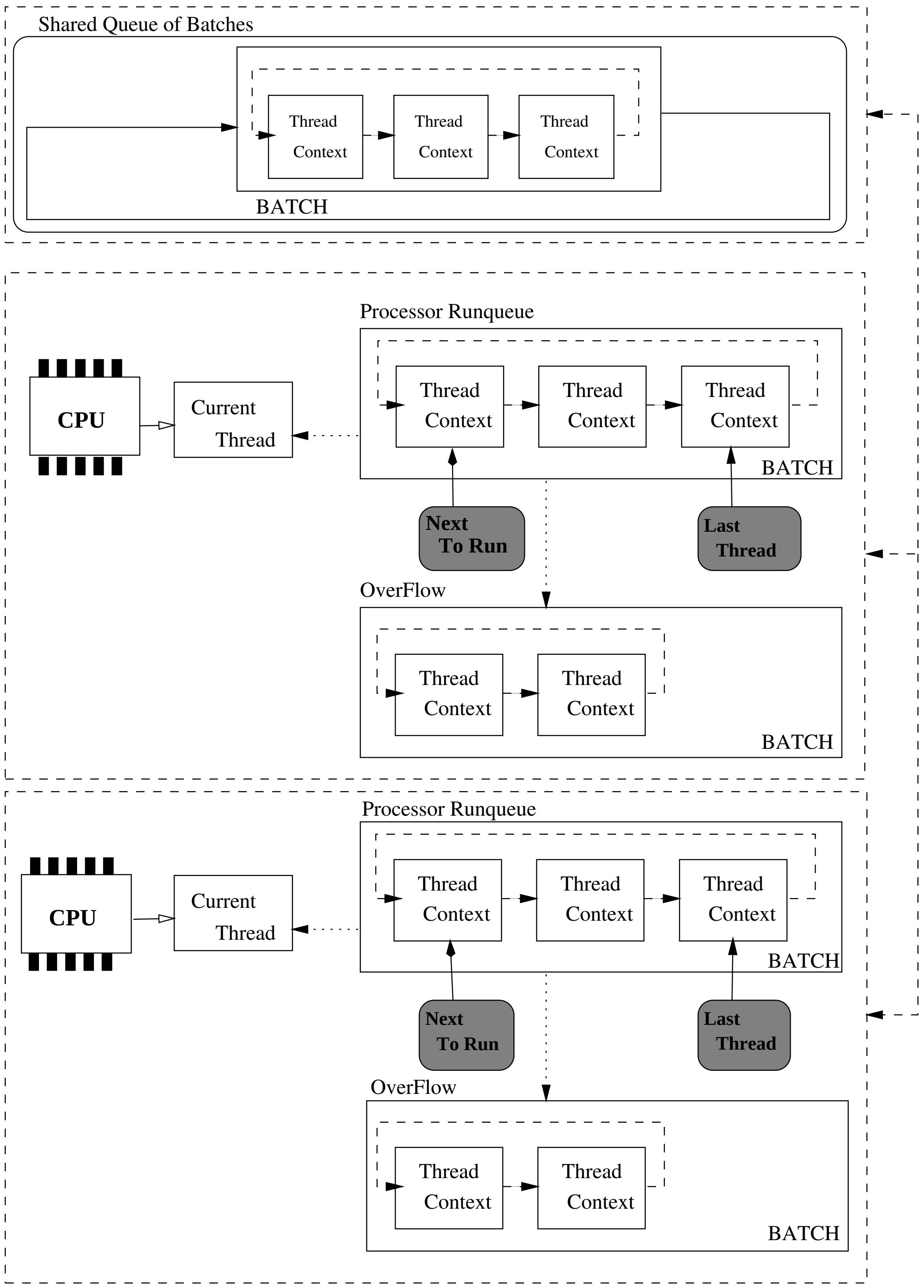


Figure 3.5: Batch Run Queue

are placed in an overflow queue. This overflow queue actually becomes the new batch which is moved to the shared queue when convenient. This creates coarser granularity in that whole batches are migrated and not single thread descriptors thus making migration not as expensive and less frequent and maintains cache affinity as long as the whole batch's workspace does not exceed the cache size.

3.4 Active and Inline Context Switching

In a context switch, the scheduler stores all the registers that will change while the thread is not running. Yet it would take less time and instructions if *only* the registers *in use* by the thread are stored since all the rest are would not be used on context switching back. A further improvement would be obtained if we force the thread to use as little as possible of the processor's registers close to the context switch. This is the idea behind *active context switching* developed by J. Moores for Occam processes [34]. The compilation stage is the only stage that has complete knowledge of which registers are in use at what point prior to the execution. Using a special feature of the `gcc` compiler [48], the current context is stored through the use of inline assembly and the compiler is told that such context switching code corrupts all the registers being used. This forces the compiler not to use any of the processors registers at such a point, thus removing the need to store such registers. This technique provides smaller data structures and an incredible decrease in context switching time. The context switch needs to be placed as inline code, making the code section larger, yet a non-inline entry makes the compiler generate function call code slowing down the context switch. The context switch code for the MESH system follows :

```
asm volatile (
  "movl %%esp,(%1)\n\t"    /* Store Stack Pointer   */
  "movl %%ebp,4(%1)\n\t"   /* Store Base Pointer    */
  "movl $$0f,8(%1)\n\t"     /* Store Program Counter */
  : :
  "r" (_the_jmp_buf) :     /* variable storing context */
```

```
/* list of registers that will instruct gcc to store*/
 "eax","ebx","ecx","edx","esi","edi","cc","memory",
 "st", "st(1)", "st(2)", "st(3)", "st(4)",
 "st(5)", "st(6)", "st(7)"); }
```

The Stack and Base Pointer need to be stored due to a bug in the `gcc` compiler that ignores the `-fomit-frame-pointer` instruction [34]. Active context switching results in a reduction of 222clks compared to the normal full register context switch [12].

On a function call, the function's parameters are pushed onto the stack and a new stack pointer value set up. To avoid these extra instructions all scheduler functions are placed as inline code, resulting into a physical copy of the function at each function call point. This makes the code bigger and faster. This is performed by the *inline* directive in `gcc`.

In MESH a context switch occurs whenever a thread terminates, blocks upon an I/O instruction or yields execution explicitly. Avoiding to use floating point numbers at these points would certainly make context switching much faster (otherwise all floating point registers require to be saved in memory prior to the context switch), especially considering that floating point numbers normally span several bytes of memory.

Use of the compiler's knowledge about runtime conditions has been taken further in a novel research where a whole thread's package with all its synchronisation scheduling and storage management is placed under compiler control. This resulted in TAM giving optimal control of the machine instruction layout yet not resorting to machine code optimisation by hand [14].

3.5 Thread Join in MESH

A thread join is the process where a thread suspends execution and resumes only when one or more other specified threads have terminated execution. This is an essential synchronisation construct normally known in literature as the **PAR** programming construct [52]. It is the only consrtuct which will ensure the termination

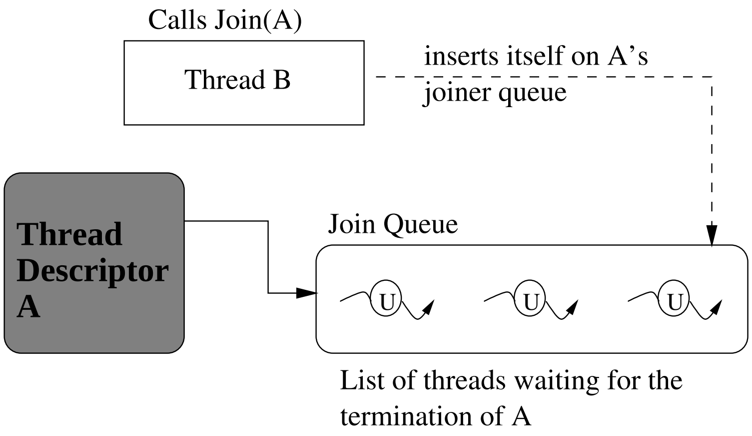


Figure 3.6: User level thread B makes a Thread_Join(A) call

of all user level threads before the termination of the main program itself, and is essential in making parallel programs deterministic. In MESH this construct is provided as yet another API call.

In MESH, whenever a thread executes a join specifying a single thread as a parameter, it is placed on a joiner queue which belongs to the thread waited for (the one specified). If a set of threads are given in the join construct, then first the thread is added on the joiner queue of the first thread specified in the list of parameters and when this terminates, it is added to the next joiner queue of the next thread, and so on. Figure 3.6 shows the case of a join when specifying a single thread.

When a thread terminates execution, it checks to see if there is any thread upon its joiner queue. If this is non-empty, then each thread on the joiner queue is set as runnable by inserting them onto the run queue. Prior to being placed inside the joiner queue, each thread stores its context and thus upon being placed again on the run queue, each thread will resume exactly at the point after the thread join call.

3.6 Communication Techniques in MESH

This topic will be briefly skimmed through since it does not play a major part in this discussion. MESH implements a system of *user-level hardware access* where there are no vertical kernel calls to invoke communication to the network card but all drivers are implemented at the user level. It minimizes latency upon entry to the kernel, yet makes managing such hardware more laborious [8]. To completely move away from the kernel no signals are used. Yet this means that the network card cannot notify MESH when data arrives or is sent to destination. This is solved by implementing a system of *local polling*. When the system enters the scheduler it always checks for the presence of some data from the card. This turns out to be much more advantagoues than using signals since access to the card is made using local memory. It is still not optimal to check continuously for the presence of incoming data whenever one accesses the scheduler, especially since one expects a very fine granularity of threads. F. Barnes has worked on a pre-compiler that automatically inserts instructions that poll for any pending communication instructions [7].

MESH uses a new generation of network cards that have direct access to local memory and in fact can notify of things happening using memory flag locations allocated to the card. Whenever data is received or needs to be transferred, an area in memory which is pre-allocated to the card is used. This memory area is a sort of temporary repository for data that is accessed by the scheduler or the card whenever they are ready to process such data. This method removes the need to copy received data from the card to another part in memory wasting computational resources or copy data to be sent from the scheduler on to the card. See Figure 3.7. This is a technique called *zero copy communication* [12]. To achieve further improvements in MESH communication, bus access is reduced by transferring groups of data-descriptors instead of single data descriptors. Details of the communications logical picture presented to the user will be discussed in subsequent chapters.

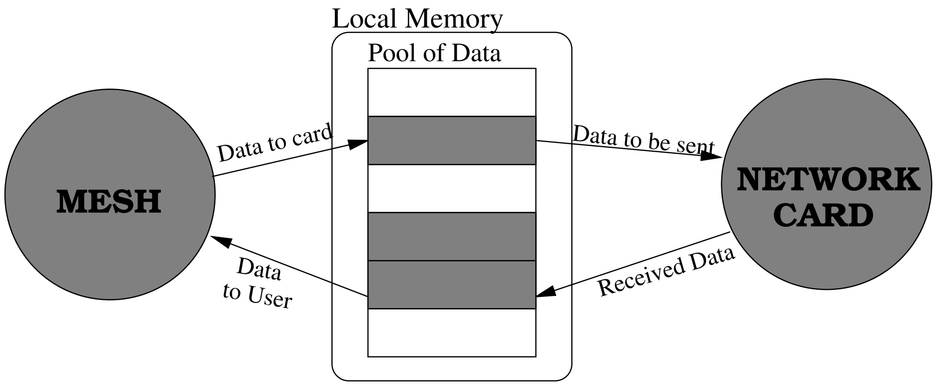


Figure 3.7: Zero Copy Communication

3.7 SMP Shared Resource Access

Whenever more than one thread tries to access some shared resource, conflicts could occur with the result of data corruption or deadlock. The order of access and update is crucial as a starting point in the search for a solution. One solution is the use of critical sections protected by the use of semaphores and spin-locks to sequentialize access. Other methods rely on the use of Remote Procedure Calls, Monitors and a Server/Client perspective, i.e. other techniques to protect critical sections [15]. Another approach is that of regulating the order of access to a resource, making away with the use of spin-locks.

3.7.1 Spin-Locks

Whenever a thread tries to enter a lock protected critical section which is already owned by another thread, then the former thread has two course of action:

1. Busy Waiting, or
2. Blocking in the Kernel.

Blocking in the kernel refers to when the processor is released to perform other tasks while the original kernel level thread is suspended until a condition is met in the kernel space itself. This involves a vertical switch in the kernel where in a fine granularity situation, such as MESH is expected to have, this is not

Algorithm 3.2 Spin Lock Acquisition, not safe

```
1    1:   mov ax,lock      ; move semaphore in accumulator
2         cmp ax,0         ; compare accumulator to zero
3         jz 2:            ; jump if zero
4         jmp 1:           ; repeat
5    2:   mov lock, 1      ; set lock, lock acquired
```

feasible time-wise. This will be discussed later on when we introduce the concept of sleeping when the kernel thread runs out of user level threads to execute (refer to Section 4.4). A further extension is that of having a dynamic system that decides on execution which synchorinization protocol and waiting mechanism to apply based upon run time factors. This has been investigated, yet is still a research field at its infancy [27].

Busy Waiting involves looping on a specific lock until the lock is found open, that is having the value of zero. A lock is simply a variable which controls a specific shared resource. As is common in the literature, we will assume that whichever thread is successful in setting the lock to the value of one from the value of zero, will own exclusive access to the resource and when releasing the resource, the same thread resets the lock to the value of zero. This ensures that only one thread can enter the critical section at any one time. The semaphore, discussed previously, is simply a variant of the lock. One simply checks repeatedly until the lock becomes zero and then proceeds in the critical section. This construct is also knows as a *spin lock*. Refer to Algorithm 3.2 for an example of a busy wait on a spin lock.

Examining the above code (Algorithm 3.2) it is immediately realized that running such a code segment on an SMP machine if one thread has just executed line 1 and another thread is actually running any line from 2 to line 4, then both will have a value of zero for the lock and thus both will enter the critical section together! This defeats the scope of the spin-lock which should give exclusive access to a resource to one and only one thread at any one time. The only way to ensure this, is ensuring that all threads have an up-to-date value of the lock and that setting and resetting the value of the lock is performed atomically. Through the

Algorithm 3.3 Safe spin-lock Acquisition

```
1:  lock            ;obtain lock on the bus for btsl
    btsl lock,0     ;copies bit 0 of lock into carry and sets bit 0
    jnc 2           ;jump if carry is 0
    jmp 1           ;spin
2:  nop             ;no operation, lock acquired
```

Algorithm 3.4 Spin-Lock Release

```
lock            ;obtain lock
btrl lock,0     ;reset bit zero of lock atomically
```

use of the atomic instruction Test-And-Set discussed previously, one can restate the spin lock acquisition code as such (see Algorithm 3.3).

Here we see that only one thread can enter the critical section, since only one thread will be able to set the lock from the value of 0 to 1 at any one time. All the others will simply spin until they find the value of the lock equal to 0. Obviously to release the lock one simply resets its value with another equivalent atomic instruction (see Algorithm 3.4).

As noted by Vella [51] and others [2, 53], whenever a thread busy waits using the above code, it is continuously setting the value of the lock to one. On an SMP machine this will tend to continuously invalidate the caches of all processors, having an invalidate on write cache mode. This leads to a drastic degradation of performance as more kernel level threads busy wait on the same lock (i.e. with finer granularity of user level threads or increased number of processors). On every loop of each executing thread, the cache will be updated resulting in the cache being a performance bottleneck. This is known as the *thundering herd problem* [49]. An alternative approach, solving this problem, is obtained by merging the above two algorithms to give Algorithm 3.5

Thus now the lock is trying to be set from 0 to 1 and if this is not successful, the kernel thread will simply loop until the lock has the value of 0. Note that the spinning is performed on a read instruction and not on a write instruction and

Algorithm 3.5 Spin Lock Acquisition with Cache Considerations

```
1:  lock           ;obtain lock
    btsl lock,0    ;copy bit 0 on carry and set bit 0 of lock
    jnc 3:         ;jump if carry is zero
2:  btl lock,0     ;copy bit 0 on carry
    jc 2:          ;if carry is set loop back
    jmp 1:         ;otherwise try to acquire lock again
3:  nop            ;no operation, lock acquired
```

thus the cache will be invalidated only when the lock is released by some thread or when the lock is re-acquired by the same thread.

The above spin locks have one major problem, if a thread acquires a lock and then subsequently crashes, then all other threads will simply busy wait for ever. Implementing a time-out is simple, using a counter that decrements itself for every loop in the busy wait until it reaches the value of zero, indicating that a time-out has occured. The value of the counter is initally set according to some forecast of the delay in busy waiting. Setting such a value to a small value would spur false alarms and thus this counter is always used to initiate a series of checks about the health of the system and not seen as a final arbiter of some crashed thread.

3.7.2 Lock-free and Non-Blocking Alternatives

Lock-free synchronisation requires no lock to be held while accessing the resource. Non-blocking synchronisation takes lock-free synchronisation further with no faster processor having to wait for a slower one to perform some operation [51]. The careful design of the order of access to such resources achieves this. These solutions require either the use of advanced and highly complex atomic instructions, which not all processors support, or the use of modified data structures with complex methods of accessing them. Techniques exist to implement non-blocking data structures such as shared queues [33, 3, 19], stacks, priority queues, counters [50, 32] and lock-protected shared queues that allow concurrent enqueue and dequeue operations [31]. Generalized techniques to convert lock bound data structures to lock free data structures have been developed yet still considered computationally expensive [50]. Pre-emption safe locking can also be improved through kernel support. A comprehensive comparison of hardware support to busy waiting can be found in [3].

3.8 Blocking in the Kernel

Whenever a user level thread requests an I/O function through the kernel, unless this data is readily available, the user level thread will be suspended by the kernel. This also results in the whole kernel level thread blocking in the kernel. On an SMP machine with a limited number of processors, each having an associated kernel level thread (i.e. one thread per processor) such a suspension would imply that the whole kernel level thread will stop from running any other user level thread, making the whole processor idle. It makes sense to let only the user level thread to block and in the meantime the kernel thread continues running other user level threads from the run queue. Also it is advantageous to flush to memory the current user level thread's stack space since another thread will probably be run instead on the same processor [22]. The user level thread's actions cannot be constantly monitored and it cannot be foretold if a kernel system call will or will not lead to a suspension of the thread.

Alternatively, one can construct **kernel call wrappers**. This means that some extra code is placed around the I/O kernel calls, with the suspension prone kernel function call preceded and followed with some extra code shown in Algorithm 3.8.

Practically the original number of kernel level threads executing user level threads always remains the same within the scheduler package. The extra ones are simply in suspension waiting to be unblocked in the kernel or in the process of cloning or dying. One major botteneck to this is that an extra vertical switch in the kernel is performed during the **clone()** call which incidentally is also a very computationally expensive operation in all UNIX Operating Systems. Blocking in the kernel for some input or output normally takes a longer time than this and thus such a delay on cloning the kernel level thread is quite acceptable. Figure 3.8

Algorithm 3.6 Kernel Provided Function Call Wrapper

1. Create a new clone of the current thread (using the **clone()** in Linux).
2. (a) Parent thread proceeds by blocking in the kernel on making the kernel function call.
   (b) Child thread proceeds by retreiving the next user level thread from the run queue and executing it.
3. When the parent thread unblocks from the kernel, it pushes the user level thread again on the run queue and terminates.

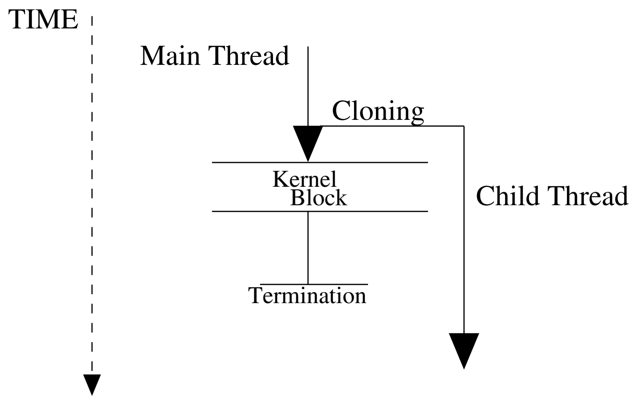


Figure 3.8: Blocking in the Kernel

shows this process in action.

All this provides a test bed for experimentation of just how many and which operations should be left under the kernel's normal supervision and which should be reimplemented at the user level. A compromise needs to be reached since both offer their own advantages and disadvantages [17, 5]. An adaptable implementation of threads package would make such decisions superflous since all options would be implemented, and then left to the discretion of the supervisor of the package to enable the optimal ones [20].

Chapter 4

Scheduling for SMP-MESH

This chapter discusses the various design and implementation issues that were applied to the uniprocessor MESH scheduler for it to execute on multiprocessor systems. The shared run queue is discussed together with the adequate protection of several shared resources from corruption. The introduction of well-behaved idling and load balancing is also investigated. Some case studies that could have caused SMP-MESH to fail are introduced and the solutions discussed.

4.1 Shared Run Queue Access

SMP-MESH implements a shared run queue for the storage of the currently runnable suite of user level threads. Each kernel level thread tries to obtain one of such user level thread descriptors and execute it. During MESH's execution, very frequent access to such a data structure occurs due to the very fine grain nature of each user level thread. Also such access is proportional to the number of threads currently running on the processors available on the SMP machine.

A lock protecting the shared run queue guarantees sequential access to enqueue and dequeue operations. The shared run queue is accessed by each kernel level thread whenever:

- A user level thread has finished executing and the kernel level thread running it needs another user level thread to execute,

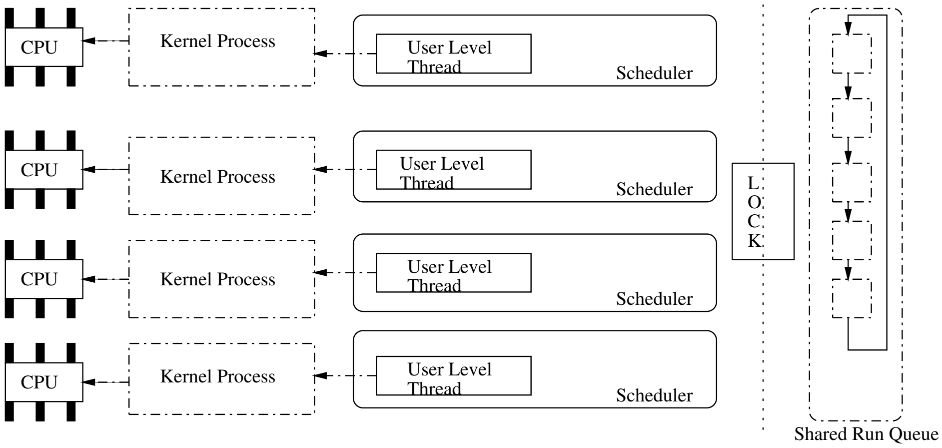


Figure 4.9: Lock sequentialising shared run queue access

- A user level thread has yielded execution explicitly and the current user level thread needs to be enqueued and another user level thread obtained for execution,

- A user level thread has requested some communication operation and this is not readily available, thus it blocks and the kernel level thread running it needs to run another user level thread,

- A new user level thread has been created and such a thread needs to be placed on the run queue for execution, or

- A user level thread has finished executing and by so doing has satisfied another thread that was waiting for it to terminate. Such a thread needs to be newly placed on the shared run queue.

Whenever any of these occur, a lock needs to be acquired, the operation performed and the lock released again. Use of hardware atomic instructions is made through the use of the `lock` instruction that guarantees that access to the run queue lock is exclusive to the first arrived kernel level thread. Figure 4.9 shows the lock protecting the run queue.

Instructions to enqueue and dequeue user level thread descriptors were kept to a minimum thus reducing the time spent holding the lock. This is imperative, since the time spent busy waiting is wasted processor time. Such a delay becomes reasonable, if it is guaranteed that it is much less than an operation for the kernel level thread to sleep in the kernel and give way for other processes to run.

4.2 Load Balancing

A uniform division of tasks is desirable at all times. Whenever a user level thread terminates or cannot be executed at that moment in time, the kernel level thread on which it is executing will find the system (shared run queue and the other kernel level threads) in one of four states:

- The run queue having user level threads inside it,
- The run queue being empty, yet other kernel level threads are all executing a user level thread,
- The run queue empty and some other kernel threads are idling, or
- The run queue empty and all the other kernel threads are idling.

These are portrayed in Figure 4.10. No other run queue and kernel level thread state is allowed is this system.

If the system is in the first state, then the kernel level thread simply proceeds to withdraw one user level thread from the shared run queue and context switch to it, thus executing it. If the system is found in the second or third state, then the kernel level thread cannot execute any other thread and will have to fall in an idle state. Later on we will see that is it desirable for such a kernel level thread to sleep in the kernel, yet it might alternatively spin until a new thread becomes available on the shared run queue. For the system in the fourth state, it would mean that the kernel level thread has reached termination point since no further new user level thread would be created and as such this kernel level thread cannot fall into an idle state. Thus finding all other kernel level threads idling should entail that the end of the program has been reached. This is not expected to happen in MESH, yet is left to the application programmer's skills that it does not happen.

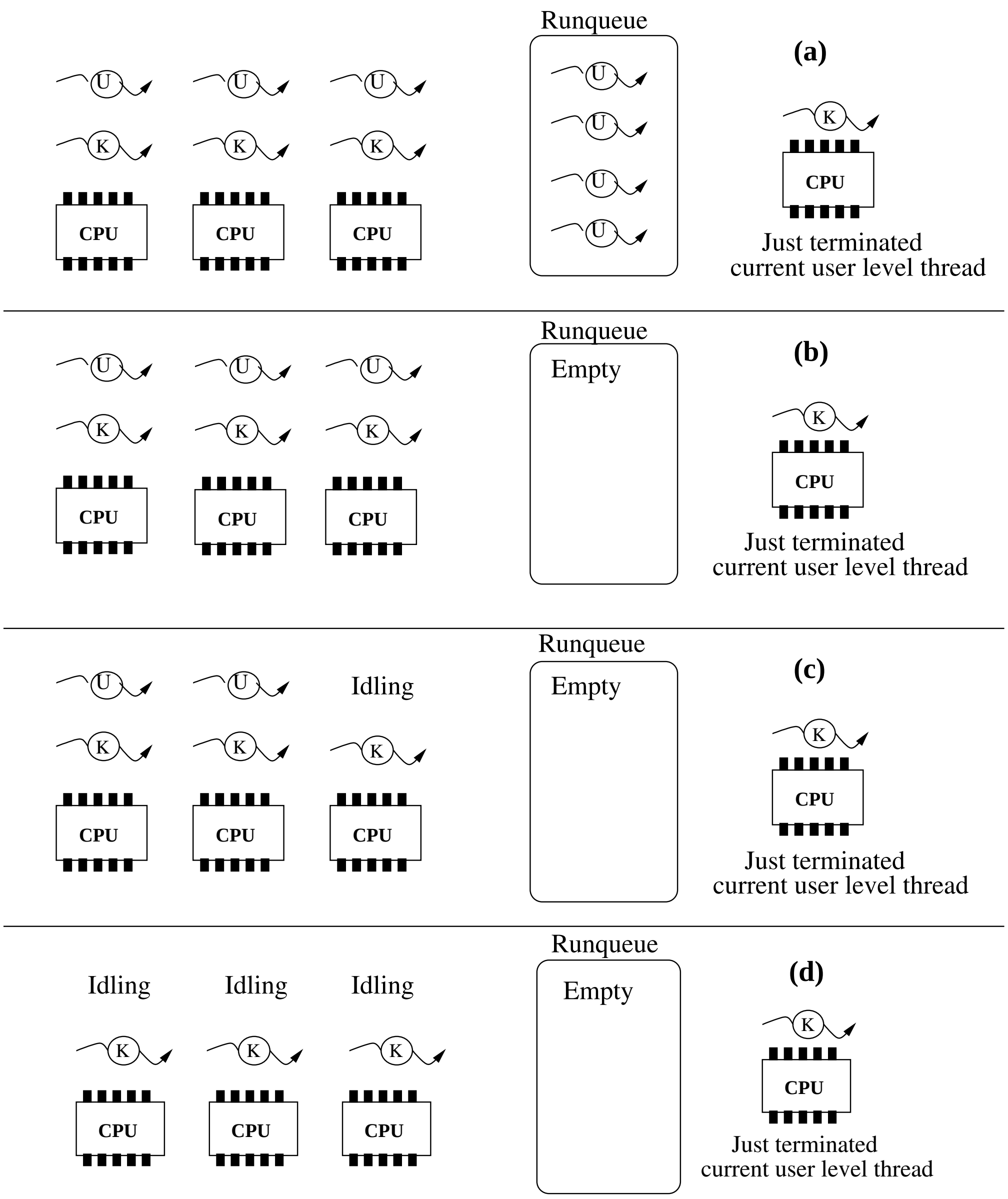


Figure 4.10: Possible states of run queue and kernel level threads when another kernel thread obtains a lock to the shared run queue.(a) threads available on run queue and all other kernel threads are running user level threads; (b) run queue empty yet all other kernel level threads are executing threads; (c) run queue empty yet some other kernel level threads are executing threads; (d) run queue empty and all kernel threads are idling

The system supports such a state by executing the scheduler's terminating and exit code segment.

For the system in the third or fourth state, whenever a new user level thread is created, then a new thread should be passed on to an idling kernel level thread. This would give an execution priority to the idling kernel threads directly. It would ensure that the same kernel thread would not re-acquire the same user level thread and result in the balancing of tasks between all the kernel level threads. Also this ensures that the run queue only has threads upon it when all kernel threads are busy executing user level threads.

4.3 Kernel Level Thread Identity

A kernel level thread continuously accesses the shared run queue to obtain new user level threads to execute and to place new threads on it. Whenever a new user level thread is executed, it is removed from the shared run queue (to avoid other kernel level threads finding it there), and it is placed on a variable indicating the currently running thread. This variable is essential since many a time there is the need to place back the currently running user level thread back to the run queue (for example on an explicit thread yield instruction or when the thread blocks on communication).

In a shared memory SMP context, where there is no concept of private memory for each kernel thread (which is the case for Linux threads, unlike LWPs on the SunOS system), one cannot access a specific variable containing the currently running user level thread since such a variable would be indistinguishable from one kernel level thread to the next. For a number of kernel level threads having a memory workspace which is all shared, one cannot access a specific variable that can tell you the descriptor of the current user level thread being executed, since such a variable would be common to all. All kernel level threads are not knowledgeable of their own identity, since they all share the same workspace to work in.

A means towards self identification is provided through the kernel call `getpid()`. This call would give the caller the process id number and would then be possible to compare to a list of running kernel level thread identifiers to determine self identification. While this is a perfectly working solution, the kernel level thread requires knowledge of self-identification every time that the scheduler is invoked from the user level thread. In a situation of fine granularity and low latency context switching, a call to `getpid()` would be very expensive, it being a kernel system call, incurring a vertical switch with large overheads and substantial delays.

Another effort towards self-identification is passing a variable giving the kernel level thread's index on to the user. When a kernel thread is initiated (through the `clone()` call), knowledge is present of the thread's identity. This can be maintained by passing it on to each user level thread. When exiting the scheduler code, this index is passed on to the user level thread and when entering the scheduler, this index is passed back into the scheduler. This requires the user's API functions to change and would break even further any level of abstraction between the user and the scheduler that currently exists. This is quite undesirable, with the user having enough problems with programming details without also having to worry with scheduler maintenance.

Identification is best achieved using the user level thread's stack. When a kernel level thread schedules to a user level thread, the kernel level thread's stack area changes to an area of memory that is the user level thread's stack space. This is stored before hand by the user level thread on creation. On creation it also stores inside its descriptor a pointer to the start of the stack and the end of the stack (achieved by start of stack plus size of stack). Furthermore, a series of variables exist that point to the currently running user level thread's descriptors, each executing on a specific kernel level thread (access to these variables is the aim of this self identification problem). By verifying that the current stack pointer lies within the memory area of one of the currently running user level thread stack areas, one can deduce the currently running kernel level thread. The algorithm can be seen in Algorithm 4.7 and Figure 4.11 portrays this for a system having 4 kernel level threads running concurrently.

Using stack space is a compromise between using a `getpid()` kernel call and using the user level thread as a carrier of the current kernel level thread index. Each time the scheduler is entered, the current kernel level thread's stack pointer

Algorithm 4.7 Kernel Level Thread Self-Identification

```
index = 0;
Repeat
  ult = user_level_thread_descriptor[index];
  if stack start of ult <= current stack pointer
   and current stack pointer<=stack start of ult + stack size
  {
    identified = true;
    current index = index;
  }
  else
    index=index+1;
Until identified
index is current kernel level thread
```

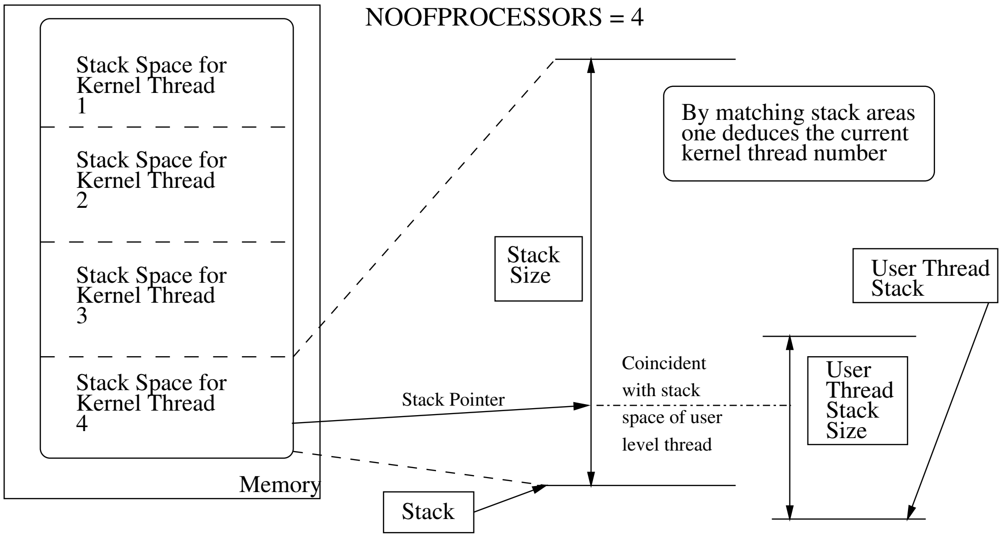


Figure 4.11: Self Identification through Stack Space Bisection

needs to be read and this compared to each kernel level's thread currently running user level thread. For a system with n processors, one expects n kernel level threads, thus this involves a worst case of O(n) with an average of O($n/2$). This is acceptable for a system with a small amount of processors. Once the current kernel level thread is identified, the procedure of self identification should not be repeated until the scheduler is exited and then entered again (through some explicit call to scheduler facilities in the user level thread code). Complete knowledge of self identity is also maintained through the proper storage of the user level thread begin executed in the variable associated with the proper kernel level thread.

A new way to avoid using such a complicated algorithm for self identification is through the use of a special register available to the new families of the Intel processor. This special variable is guaranteed not to be used by the kernel for execution yet is still saved on the context switch by the kernel. Also *gcc* guarantees not to use such a register. Thus the kernel level thread can store a value for self identity in this register allowing it to be retrieved when required in an O(1) operation. This alternative is being investigated to be included later on in SMP-MESH.

4.4 Well Behaved Idling

Whenever a kernel level threads tries to obtain a new user level thread to execute from the shared run queue and the run queue is empty, then the thread cannot do anything but wait for a newly available user level thread to run. It can busy wait, yet busy waiting is still processor intensive. Thus each kernel level thread is made to sleep in the kernel through the use of a semaphore block, thus releasing the processor for use to other running applications and kernel level threads.

4.4.1 Sleeping Threads

A kernel level thread sleeps whenever it finds the shared run queue devoid of user level threads to execute. Access to the shared run queue occurs when:

- User level thread terminates and a new user level thread is needed for execution, and when
- User level thread is made non runnable (blocking on communication, placed upon join queue, and others) and a new one needs to be fetched.

Access is first made to the shared run queue status to verify its occupancy. This occurs after managing to obtain the lock protecting the use of the shared run queue. If the run queue is found empty, the kernel level thread sleeps. When threads sleep, they cannot sleep holding the lock blocking all further access to the shared run queue. After incrementing a global variable denoting the number of sleeping threads (valid to verify if this thread is the last to sleep see page 31) the lock is released and the thread proceeds to block in the kernel. This is performed through the function `safe_pause()` where a kernel provided function (`semop(ownsema)`) is used that is made to decrement the value of a semaphore by a value of one. This semaphore is a variable provided by the kernel that is assigned to each and every kernel level thread on the scheduler's initialization. On sleeping, such a semaphore is accessed knowing which kernel level thread is performing this operation (see page 33). This semaphore is initially initialized to the value of zero, thus such an operation would block this kernel level thread until the value of the semaphore is one, when the semaphore can be deducted the value of one.

Using a semaphore operation ensures that when the lock is released, if another thread had to try to wake the kernel level thread before the former thread blocks in the kernel, then the semaphore operation would still succeed, without the former kernel level thread never blocking. Using a different scheduling technique (say signals) could result in the thread sleeping in the kernel without ever receiving the wake up signal. This is the *lost wake up* problem [49, 51], as seen in Figure 4.12. A sleeping status flag is also set up for each kernel thread. Figure 4.13 shows a typical situation with active and sleeping threads.

Whenever a kernel level thread places a new user level thread on the queue, it checks to see if there are idling threads that can execute this new user level thread. This can be performed by scanning through the sleeping status flags of each kernel level thread, apart from one's own, and waking up the first sleeping

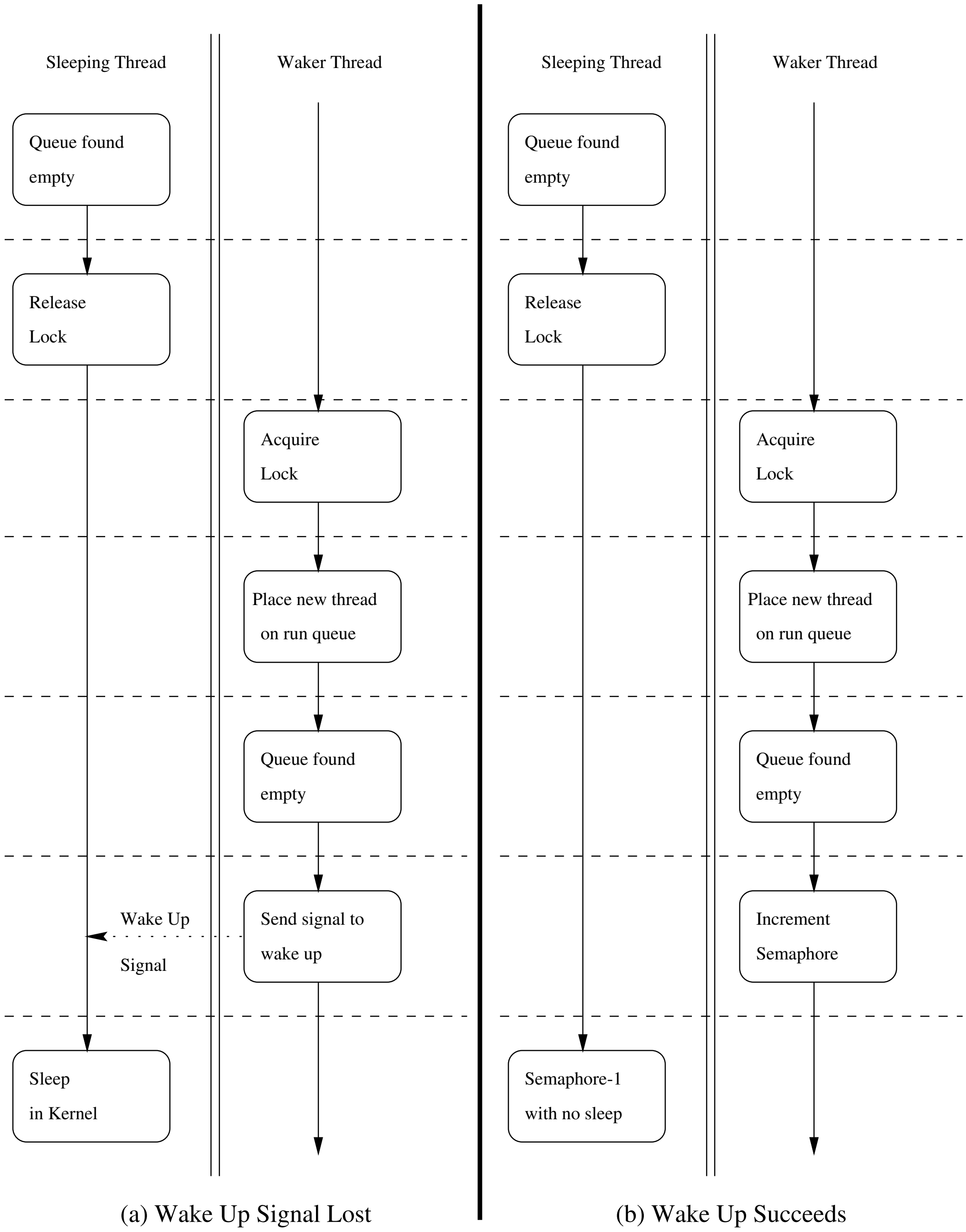


Figure 4.12: Thread Sleeping and Waking; (a) *lost wake up* using signals; (b) proper wake up using semaphores

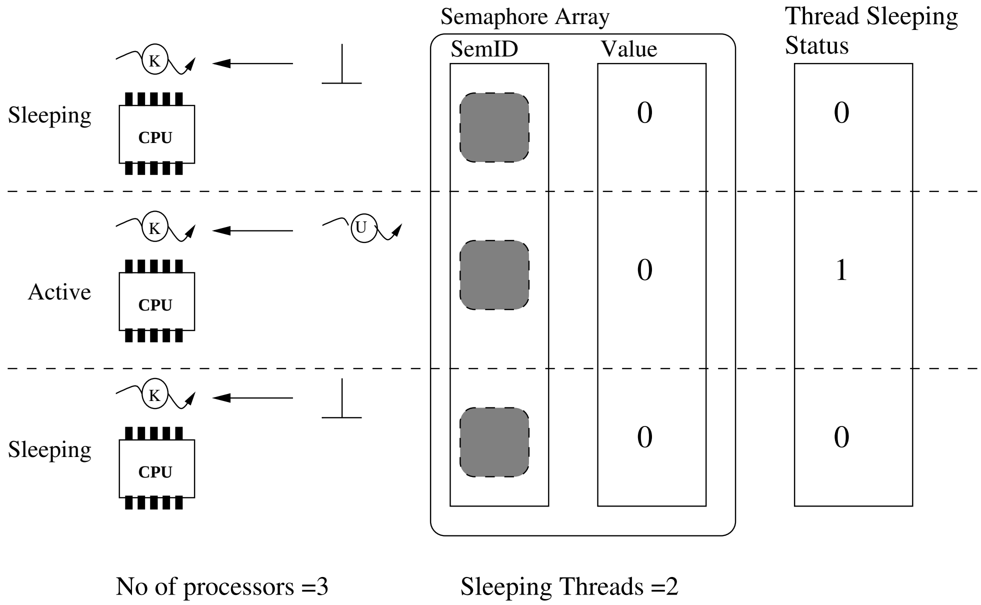


Figure 4.13: Kernel Level Thread Sleep Configuration

kernel level thread found. Examining Figure 4.10 and Figure 4.13 gives us an alternative solution. Whenever a kernel thread is sleeping, the run queue must contain no user level threads inside it. If run queue is not empty, then the user level thread is merely enqueued and the thread proceeds. If run queue is empty, the number of sleeping threads dictates if it should proceed (if discovered to be zero) or scan through the kernel thread sleeping states for the first occurring sleeping thread and wake it up (if found with value larger than zero). When a sleeping thread is found, the number of sleepers is decremented, the sleeping kernel level thread's status is set to active and waking up is performed by incrementing the sleeping thread's semaphore by one using a `semop(sleepthreadsema)` and then proceeding. The newly woken up thread proceeds to run the newly available user level thread. An alteration on this algorithm is necessary to decrease the number of superfluous wake-ups (for more details see page 40).

The newly woken up thread has no knowledge of the kernel level thread that woke it up, yet such knowledge is superfluous in this context. If necessary, knowledge of the waker thread can be obtained using a structure assigned to each kernel level thread on which a message is left by the waker, since the waker thread has perfect knowledge of who it is waking up. Note that thread wake ups can also be safely performed concurrently. The waking up instruction is a kernel system call with the associated delays of a kernel vertical switch. Thus the lock for the queue is released prior to waking up any thread, and re-acquired immediately afterwards, giving time for other kernel level threads to access the run queue during this delay. Non-corruption of the sleeping status flags is possible through the correct use of the lock to the shared run queue. In the sleeping thread and the waker thread, the lock is only released after the thread's sleeping status flag has been set. Preference on which thread is woken up is given to the kernel level thread with the lowest index. This might be seen as opposing the principle of load balancing, yet sleeping threads are not a burden to the system (within reasonable amounts) and no preference should be made at any one time to any one thread blocked on a semaphore in the kernel. In the situation of a processor being faster than another on the same system, the faster ones should be mapped to the lower index numbered kernel level threads.

4.4.2 Optimisation of Thread Idling

First Measure

From the time a thread is woken up from a kernel semaphore block to it acquiring the lock to access the shared run queue (thus retrieving the user level thread it was woken up for), another kernel level thread might acquire this lock and empty again the run queue. The newly woken up thread would thus revert back to sleeping on its kernel semaphore. This entails two kernel operations, that of waking up and again sleeping again, with no apparent useful work being done.

To avoid these superfluous wake-ups, the new user level thread to execute is rerouted to the sleeping kernel level thread via a variable assigned to it and not through the run queue. Further more, such a variable can be the variable used to point to the currently running user level thread. The newly woken up thread then simply context switches to this. The user level thread is placed there by the thread waking up the kernel level thread. Such an algorithm presents a further advantage for the thread doing the waking up: there is no need for an enqueue operation on a wake-up. This method also enables a more reliable forecast of the system state, since the newly created user level thread will surely be executed by the newly woken up thread and by no other. This cannot be guaranteed if the new thread was routed through the run queue.

Second Measure

Whenever the shared run queue is empty and there are sleeping threads in the system, a user level thread which is destined to go on the run queue should cause some kernel level thread to wake up and execute it. Cases exist where the user level thread being placed on the run queue is not a new thread but a descheduled thread, such as when the user level thread executes a thread yield.

A thread yield is an explicit user operation that gives up execution to allow other thread to execute or otherwise to allow polling of communication. In the scheduler it consists of the currently running user level thread being placed on the run queue, the next one on the queue retrieved and this executed.

If there are sleeping threads, in a yield the user level thread is passed on to a sleeping thread followed by an operation to wake it up and the kernel level thread that passed it on would proceed to interrogate the run queue for a new user level thread to execute. Yet the run queue is obviously found empty. Such a thread would then proceed to sleep on a semaphore. Thus two very expensive operations are performed superfluously, that of waking up a thread and another one that of sleeping in the kernel.

A solution is to interrogate the status of the run queue on each thread yield operation and if found empty, simply re-execute the same user level thread. With 32 levels of priority, this takes 32 interrogations with the lock held throughout. Careful analysis of the possible states available to the system allows a special situation to be operated for a thread yield whereby in all cases no kernel level thread is woken up, and the user level thread is placed on the run queue. If a higher priority thread exists on the run queue, then no kernel level thread can be in a sleeping state (see Figure 4.10). If the kernel level thread has no other thread to run, it would simply dequeue the same user level thread again. This can give a temporary illegal state of the system where threads exist on the queue and there are also sleeping kernel level threads, yet the lock is not released until the whole operation is terminated, thus the system does not fail. It has been calculated that it takes much less operations to enqueue and dequeue the same user level thread than querying the shared run queue for size on all 32 priority levels. Also one expects to find such a small number of user level threads running with most kernel level threads sleeping only at the beginning and at the end of the program's execution.

Third Measure

Whenever the currently running user level thread is scheduled out (it is not run anymore and is not inserted into the shared run queue again), a further user level thread is fetched from the queue. There are instances, such as in the `thread_par()` algorithm (it consists of placing new threads on the run queue and then joining on them), whereby the currently running thread is stored away and scheduled out. Following this step, the algorithm proceeds to place new threads on the run queue. The usual steps to retrieve a new user level thread are executed. If the shared run queue is found empty, this would result in the kernel level thread sleeping on a kernel semaphore and the new threads never placed on the run queue for execution. Thus in such an algorithm, it is advantageous to stop the kernel level thread from sleeping even in cases where the shared run queue is found devoid of user level thread to run. The newly placed user level threads would, in such case, be the next threads to execute.

The above modification is due to an inherent defect in the uniprocessor MESH algorithm. A perfectly SMP compliant solution, would have required that the placement of the new user level threads occurred prior to the scheduling out of the current thread. Yet such a change would have changed drastically the code of the uniprocessor MESH having major repercussions on other algorithms. An SMP-MESH built from scratch would have catered for this.

4.5 Other Shared Resource Protection

Accessing a shared resource concurrently on a Shared Memory model of an SMP machine may lead to corruption. While reading data presents no problem, when updating data each thread must be informed timely of such an update. Conversely each thread of execution should not be allowed to read information when another thread is updating such a resource. Such sequential modification of data is achieved through the use of spin-locks in this SMP-MESH implementation. The shared data structures to be protected are the shared-run queue, dynamic memory allocation, thread termination and the thread join queue. Also the self-identification algorithm needs to be protected. The shared run queue has already been discussed in Section 4.1.

4.5.1 Memory Allocation

Whenever a new user level thread is created, new memory needs to be requested through the kernel. A call is made to the `malloc()` function which as such is a library call that makes access to the `sbrk()` kernel call. While one is guaranteed that the kernel call is SMP protected, one does not have such a guarantee for the library provided section of the code. It is noted that on the Linux Operating system, protecting calls to `malloc()` using a spin-lock access protocol rids one of certain segmentation faults. Also thread creation is not expected to be very frequent in MESH and thus a spin-lock protects this memory allocation.

A more proper treatment is the implementation of a specific SMP version of `malloc()` or implementing a kernel wrapper that has busy waiting instructions to the uniprocessor version of `malloc()`. Otherwise user access validity to memory allocation cannot be guaranteed. Such a kernel wrapper access is shown in Figure 4.14.

4.5.2 Thread Join

A thread join is the process where a thread suspends execution until one or more other threads have terminated executing. In SMP-MESH whenever a join call is

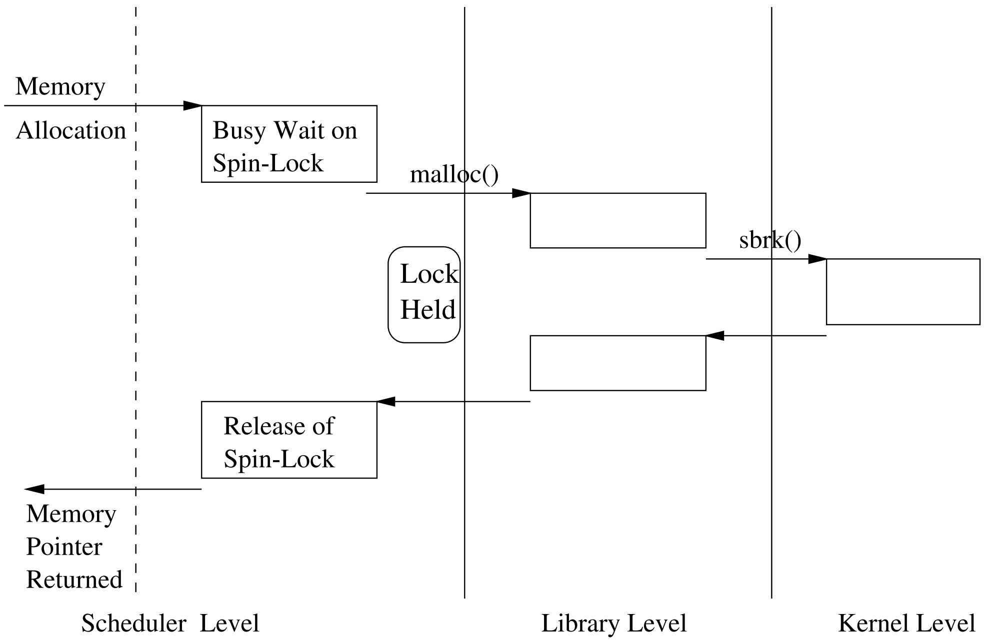
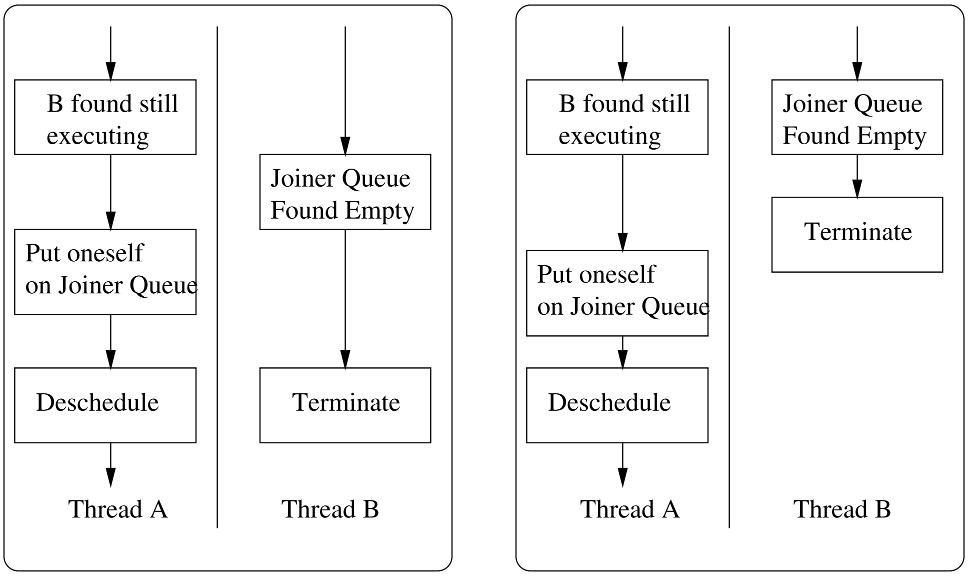


Figure 4.14: Concurrent Memory Allocation Protection

made, the thread joined upon might be in three states:

- User-level thread is executing,
- User-level thread is in the process of terminating, or
- User level thread has already terminated.

If user level thread joined upon has terminated, then the calling user level thread is simply not added to any joiner queue. Then the join proceeds to the next joiner thread given, or if no more user level threads remain to be joined upon, execution can resume on to the next instruction. If the user level thread is executing, then the joining thread is placed on to the joiner queue of the waited-upon thread. Then the kernel thread proceeds to retrieve the next thread from the shared run queue, removing the thread that initiated the join from the list of runnable threads. When the thread waited upon terminates, before scheduling out, it places all the threads on its joiner queue back on to the shared run queue, thus making them runnable again by any available kernel level thread.



Thread A is lost never to be run again

Thread A accesses Thread B after it has terminated

Figure 4.15: Thread A executes Thread_Join(B)

A problem arises when the thread is in the process of terminating and at the same time another thread is placing itself on to its join queue. There are two possible sequence of execution, both of which are equally drastic. First, if the terminating user level thread finds its joiner queue empty, it proceeds on to terminate by scheduling to another user level thread. Yet before this deschedule occurs, another might place itself on to its joiner queue, then the thread that called the join construct will be lost and never run. Secondly if the thread that called the join construct checks if the thread has terminated, and found still executing, then it would proceeds to add itself on its joiner queue. Yet if such a thread terminates before the enqueue operation terminates, the joining thread would try to access the joiner queue of a thread which has terminated, causing a memory fault. These two paths can be seen in Figure 4.15 where thread A makes the join call to wait for thread B, both of which are user-level threads.

While such occurrences are not expected to occur frequently, to avoid such a fault, a lock is needed to protect this happening. Whenever a thread is terminating, just before it checks to see if any thread is upon its joiner queue, it busy waits on such a lock. Upon acquiring the lock it proceeds to check its joiner queue and places any threads upon it on the shared run queue. Then the lock is released. On the other hand when a thread makes a join call, before checking the status of the thread to be joined upon, it acquires this lock on the joiner queue and then proceeds to add itself to this thread's joiner queue or otherwise, after which the lock is released. This will ensure that the above circumstances never occur. Note that due to the rarity of such a complex occurrence of circumstances, busy waiting would rarely occur and the only overheads are the instructions to obtain such a lock. The lock protects the joiner queue and the status of the user level thread since the sequential access of both these resources is essential to prevent the above sequence of events happening.

4.5.3 Kernel Level Thread Identity

Whenever a scheduler function executes, knowledge of the kernel level thread entering the scheduler is indispensable (Section 4.3). Prior to a context switch, the kernel level thread changes the value of the variable pointing to the currently running user level thread. Each kernel level thread has such a variable. There is a short period of time, between the time of placing the new user level thread descriptor and the context switch proper where this variable does not have a correct value inside it. If in such a period of time, some other kernel level thread accesses this variable, it might deduce the wrong self identification index. This occurs if the latter is running the user level thread that the other kernel level thread is context switching from.

To protect from this quite rare circumstance, a further spin lock is implemented that protects access to the self identification procedure. This increases latency on such a highly used procedure, and thus widens the gap between the non-SMP version of MESH and the SMP version.

4.6 Joining Threads in an SMP Context

Thread Join allows the synchronisation of user level threads with one thread suspending execution until a number of given threads terminate. It is an essential construct in threads programming and presents certain problems in an SMP perspective. Whenever a `Thread_Join(list of threads)` is executed, the scheduler places the currently running thread on the first given thread's join queue and schedules in the next available thread from the run queue. Refer to page 20 for more details.

In an shared memory SMP perspective, having the lock protecting access to the joiner queue of each thread would ensure that no thread is terminating while another is placing a link to itself on the joiner queue. A most intricate and peculiar problem is seen to arise from this algorithm. First let us examine on which stack the kernel level threads are operating and at what times.

4.6.1 Operating Stack

Whenever a kernel level thread is executing a user level thread, then the kernel level thread's stack is the same as that of the user level thread. This is crucial for the Kernel Identity algorithm (see page 33). When a user level thread terminates or is scheduled out (it is not run anymore and not inserted into the shared run queue again), the kernel level thread executes all its operations, to the moment of obtaining a new user level thread and performing the context switch to it, on the old user level thread's stack space. From the context switch onwards then it executes on the stack space of the new user level thread. No problem is posed in most circumstances, as long as the kernel level thread holds the run queue lock to the context switch point, preventing any other kernel level thread access to this same stack space. Upon sleeping in the kernel finding no thread on the run queue, the stack is left intact as long as the old user level thread is not allowed to execute.

4.6.2 Joining Stack

An outline will be given of a certain number of steps which lead to a situation where a kernel level thread runs in a forbidden stack space after User Level Thread A makes a `Thread_Join(Thread B)` call on kernel level thread 1.

On the thread join, the kernel level thread (1) after placing its user level thread (A) onto the joiner queue of the given thread (B) proceeds to obtain another user level thread from the shared run queue. It finds the run queue empty and the kernel level thread (1) proceeds to sleep on a semaphore (see Section 4.4). This only occurs if the user level thread (B) given in `thread_join(B)` is already running on another kernel level thread (2). Upon termination of this user level thread (B) it proceeds to place the original user level thread (A) on to the run queue. Upon finding more user level thread sleeping, another kernel level thread (3) will execute this thread (A). All this is well and good until at some later stage, the original kernel level thread (1) re-awakens to execute some other user level thread. To the point of the context switch, the kernel level thread (1) is still running on the original user level thread's (A) stack space. This is a memory fault since this stack space contains outdated and incorrect data (from the execution on thread 3). Note that thread 2 and thread 3 could be separate or the same thread, both leading to the same situation. The original kernel level thread (1) needs to make use of the thread's stack, most importantly since it contains the index of self identity, information that cannot be obtained from usual means since the algorithm given in 4.3 is no longer valid here (an incorrect stack space is being used). Figure 4.16 makes this clearer.

Such a situation occurs since a thread is allowed to execute even if scheduled out. This stems out from the semantics of the `Yield()` construct. This problem is not a property of sleeping in the kernel, it occurs also if busy waiting is used whenever there is an idle state. Shifting to a new stack which cannot be corrupted before proceeding to sleep in the kernel solves this. Every kernel level thread is assigned a special user level thread which has its own stack space. Whenever a `Thread_Join()` is executed and the run queue is found empty, the kernel level thread proceeds to context switch to its own special thread (called the *join thread*).

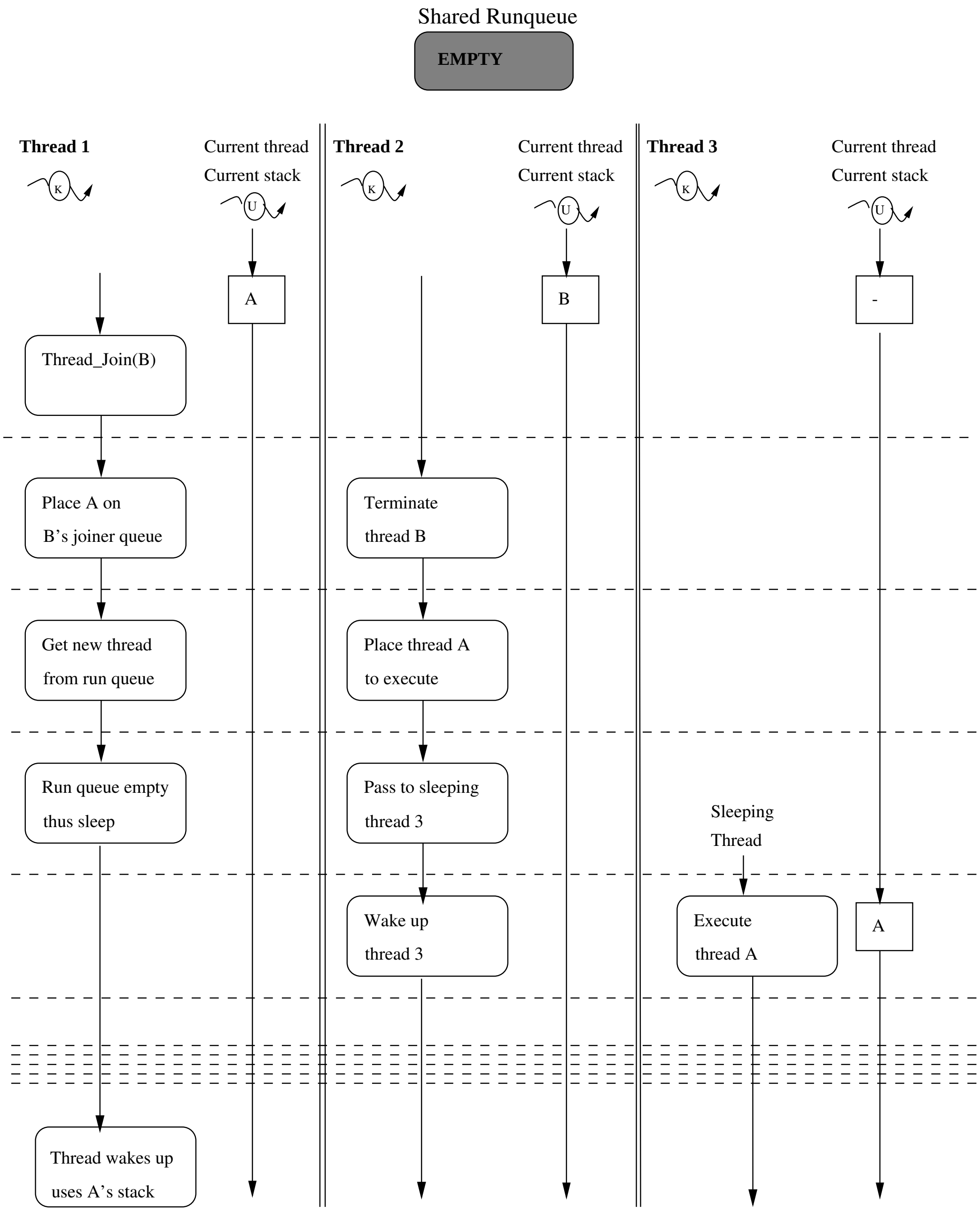


Figure 4.16: Invalid Stack Access

Algorithm 4.8 Safe Join_Thread Execution

1. Obtain lock to access shared run queue
2. If found empty, release lock and sleep and then re-acquire lock when woken up
3. Obtain newly available user level thread
4. Release lock to shared run queue
5. Context switch to newly available user level thread

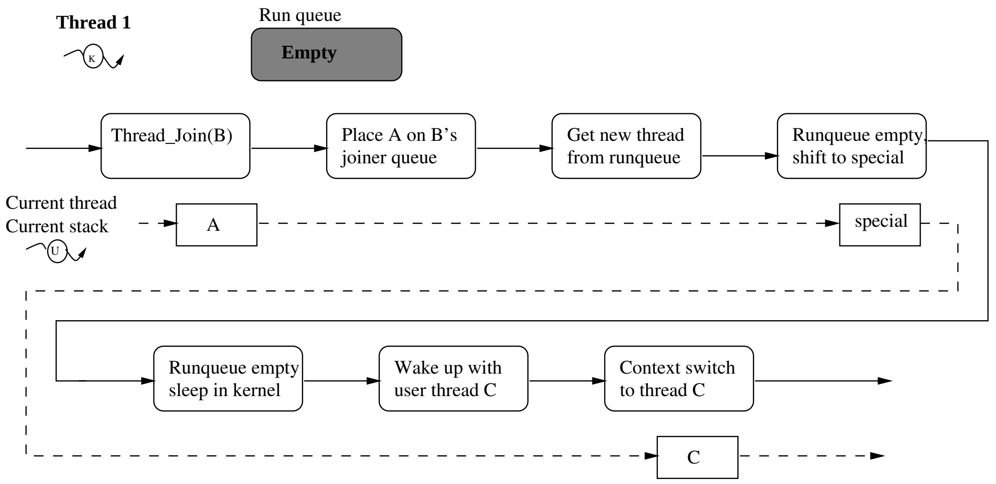


Figure 4.17: Proper Stack Access on Thread Join

This thread runs the code in Algorithm 4.8 repeatedly.

This stack will never be corrupted, it is specific to each kernel level thread. It also stores information of self identity, it being placed there on scheduler initialization. This thread acts as an intermediary stage in between the old user level thread and the new one. See Figure 4.17.

4.7 Scheduler Initialization

Before the threads start executing, the scheduler is setup for running. This includes:

- Setting the number of sleepers to 0,
- Setting the sleeping state of each kernel level thread to active,
- Requesting the sleeping semaphores from the kernel and placing them as shared semaphores,
- Setting the value of each semaphore to zero,
- Redefining certain signals such as SIGINT to execute proper functions that terminate cleanly all kernel level threads, and
- Setting all spin-locks to the value of zero.

In addition, a number of user level threads (equal to the number of kernel level threads) are set up. These are the *dummy threads* which are assigned to each kernel level thread. These dummy threads supply the stack space on which the variables of kernel level threads are initially stored.

After setting up the items above, the scheduler proceeds to execute repeatedly the `clone()` function that spawns a number of kernel level threads. The kernel level thread is set to use the same memory area as the parent and then directed to a special function. This special function consists of accessing the shared run queue and trying to access the first available user level thread. On failure each kernel thread sleeps on its semaphore until a new user level thread becomes available for execution. In the mean time, each kernel level thread executes on the dummy thread's stack mentioned above, up to the point of the context switch. The parent thread proceeds to execute the main function of the available main thread. It is noted that, most of the time, the kernel level threads start off by simply sleeping in the kernel.

4.7.1 Number of Kernel Level Threads

The number of kernel level threads should be on average always equal to the number of processors available on the system. Having a smaller number of kernel level threads results in under-utilization of the system. Excessive number of kernel level threads forces the operating system to continuously preempt these threads to context switch from one thread to the next, resulting in unnecessary delays. With the number of kernel level threads equal to the number of processors, a slight under utilization still occurs when the thread blocks in the kernel for I/O operations and other kernel provided functions. With each kernel level thread not bound to a specific processor, the best number of kernel level threads is one or two threads larger than the number of processors, leaving it to the system to schedule the extra thread or two on the idling processor when the former blocks in the kernel. Alternatively one can introduce kernel function wrappers (see Section 3.8) that make perfect use of blocked kernel thread at the expense of the creation of a new thread altogether.

4.8 Scheduler Termination

The `clone()` function spawns off kernel level threads that are immediately killed by the Linux kernel in the event of the parent terminating. The MESH semantics enforce that the main user level thread waits for the other user level thread to terminate before terminating itself (using the `Thread_Join()` or `Thread_Par()` construct). A common exit point is provided by such semantics. With only one user level thread still active (the main one), one expects all other kernel level threads to be in a sleeping state. The scheduler proceeds to delete each semaphore from the system. Each sleeping kernel level thread would in turn be woken up abruptly and noticing the removal of the semaphore (that can only occur explicitly) will proceed to terminate (using `exit()`). Finally the last remaining kernel level thread (that is the same one which deleted the semaphores) will proceed with the normal uniprocessor termination procedures and finally exit.

Chapter 5

Inter-Thread Communication

Inter-thread Communication in MESH is achieved using a complex hierarchy of data structures and constructs. This communication system implements both inter thread communication on the same system (same computer) and inter-system communication (between different computers). Inter system communication is provided by extensive driver implementations of two high bandwidth Network Interface Cards (NIC), the Alteon Gigabit and the Intel 100 Megabit Ethernet cards. The user is not abstracted from the system's complex communication implementation thus reducing even further the abstraction layer between the user and the system's internals. The API provides a myriad of steps to be carefully followed to allow inter-thread and inter-system communication and the user holds no guarantee that the data has been transmitted correctly, and received by the receiving end.

Data that needs to be transmitted by the user level thread has to be placed into a system-provided *message*. The memory space containing the message is requested from a *message pool* through a *poolport*. Such a request is a blocking operation. The message's destination is set up by specifying the *destination port* into the message's *buffer descriptor*. The message is sent off to another thread or through the NIC using a *source port*.

Data reception is performed by the user level thread by making a request for a message through a receive port. This is a blocking operation. After the data is used, the message's memory space is recycled back into the pool through a

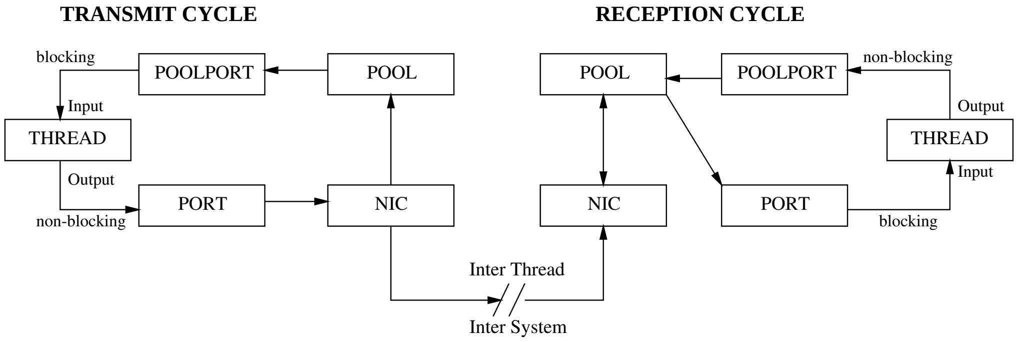


Figure 5.18: Data Transmission and Reception. Courtesy of M. Boosten [11]

poolport using a non-blocking operation. Internally when a message is received, the message space to hold the data in it is borrowed from the pool.

Data reception and data transmission is shown in Figure 5.18. In the above discussion, a blocking operation is an operation that forces the thread's execution to be suspended until certain conditions in the system have been met to allow such an operation to terminate. These conditions may be the arrival of some message or the accessibility to some area of main memory. With the communication constructs provided in MESH, the application programmer can assign tasks to specific port numbers and merely dump task related data onto such ports. On the other hand, task solving user level threads can listen on their specific port number for any data. Thus the consumer and the producer of data can be implemented separately as long as a common port is used. The fastest user level thread listening on a specific port will receive the data. This task related partitioning is implemented on a one system basis, with network communication being a one-to-one operation fixed by the Ethernet address of the destination port.

5.1 Communication in MESH

This section will outline in considerable detail the intricate relationships of the communication data structures found in MESH. This is necessary to justify the use of locks to protect or otherwise some of these structures.

Message pools, poolports and ports are structures that only exist within MESH and are not accessed by the NIC. They are used to provide a virtual communication environment for the user to work in.

5.1.1 Consecutive Memory

Consecutive memory is an area of memory where each element (a byte) is ordered in a physically consecutive way. The reason this memory area is used and not conventional memory is to allow concurrent access by the Network Interface Card and MESH. It is used to store data packets aimed for inter-thread communication. Using special drivers, the kernel is prevented any access to this memory area for conventional allocation and use. The kernel is made to see this area as just another I/O device through the use of the `bigphysarea` patch to the kernel code and special kernel drivers called `consequetive`. Allocation of this special area of memory is achieved by initializing this area before the Linux operating system itself is initialized. When MESH initializes, this memory is mapped into user-space and therefore made easily accessible from the application. This area allows MESH to quickly calculate the physical address of memory locations within this consecutive memory. These physical addresses are then used by the NICs hardware DMA to access data packets directly.

Inside MESH, a pointer exists that points to the start of this consecutive area of memory and another pointer points to the point where all memory allocated ends and the still unallocated memory starts. These pointers allow runtime dynamic allocation of consecutive memory. Each allocated area is physically aligned to suit the requirements of the NIC card in use, since normal NIC DMA cannot access every location. By exchanging pointers between MESH and the NIC, packets stored in this memory area are identified and used. On data transmission, MESH passes a pointer to the NIC that points to the area of consecutive memory that holds the data to be sent. On reception, the NIC places data in this area and passes a pointer to this data for MESH to use. This enables *zero copy communication* with data transfer being of order O(*message length*). In fact the user is also passed a pointer to this area and there is never the need for data to be transferred physically from one location to another in inter-thread communication.

5.1.2 Device Independent Packet Interface

The system makes use of a structure called the Device Independent Packet Interface (DIPI). It is used to store in a logical fashion a series of Buffer Descriptors that are directly accesses by the NIC. A list of DIPIs can be present on each system. The DIPI consists of a series of four queues each having a related consumer and a producer:

Queue	Producer	Consumer
Send	user	NIC
Sent	NIC	user
Receive	user	NIC
Received	NIC	user

Table 5.2: DIPI Queue Structure

Each queue is a list of elements called Buffer Descriptors (BD). Each BD contains hardware related information such as the start of the buffer (see below) in physical memory, the length of the buffer, the address of the next BD and a value marking the buffer as being the last one or not. This last attribute allows a packet consisting of more than one buffers to be transferred between MESH and the NIC making use of system data-bus burst mode upon data transfer.

On receiving a packet over the network the NIC takes a BD from the receive queue, fills it with packet data and places the BD into the received queue. On data transmission, MESH places the BD into the send queue while the NIC, after transmitting the data over the network to its destination, places the BD into the sent queue. All these BDs are in turn re-used making a completely closed cycle with a fixed number of BDs system wide..

A buffer is essentially a message (refer to Section 5.1.3). Each BD with its associated message is actually stored unto consecutive memory. Figure 5.19 shows a DIPI structure with the queues that MESH can access whenever it plays the role of data producer.

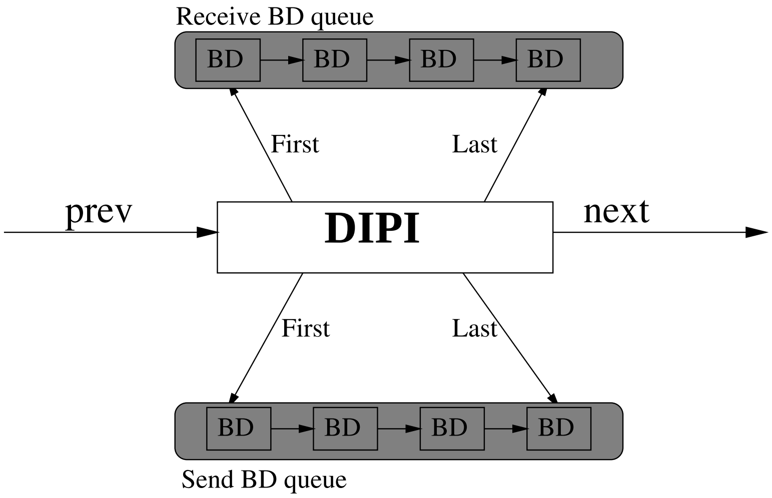


Figure 5.19: The Device Independent Packet Interface as seen by MESH being the producer

5.1.3 Messages

A message is some data with a header for the use of communication. It holds information to be used by the MESH system, and also information directed for the NIC's use. It holds all the information necessary to allow complete transmission from one user level thread to the next including source port, destination port, and others. The maximum length of a message is dictated by the NIC being used, that in its turn determines the Ethernet's maximum packet size in use.

Every new message is created with maximum size and exists in consecutive memory. Each NIC had associated to it a DIPI. Upon initialization, the full maximum complement of messages is created and allocated to the send and receive queue of the NIC's DIPI. The number of messages in the system always remains constant, with the consecutive memory being re-used cyclicly.

The application programmer is abstracted from the internal details of each message by being passed a pointer to the start of the data area, that lies after the message header in memory. MESH operates on the message level and performs the inclusion of the BD prior to transmission to the NIC and strips it off upon receiving the message from the NIC. Thus the NIC has a different view of each packet than MESH and in fact may see a number of messages as a single unit. The

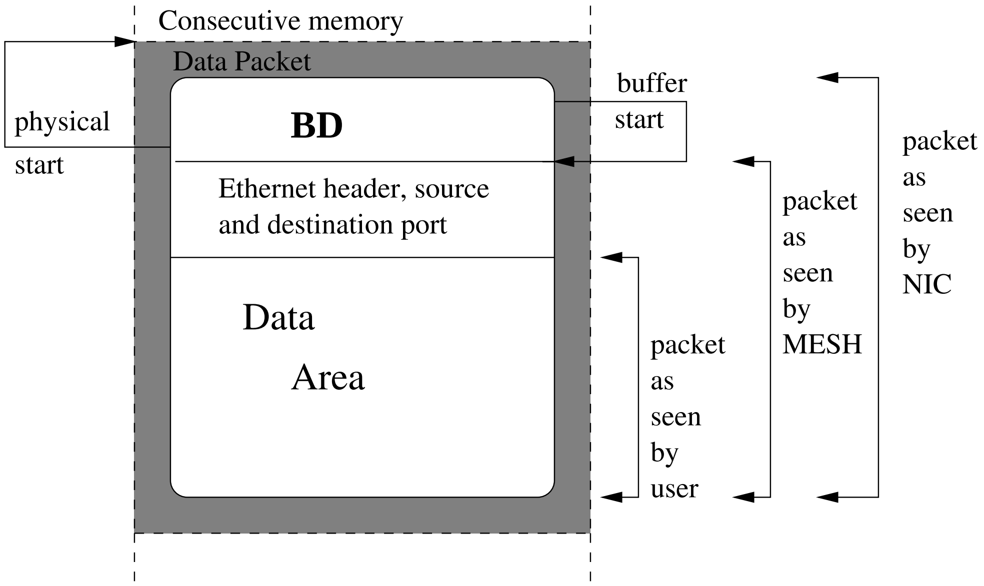


Figure 5.20: Data Packets

layout of each data packet inside consecutive memory can be seen in Figure 5.20. Note that the BD is used to map the packet into physical consecutive memory and the message header deals with the communication details of the packet. In network communication, only the message is transferred, with the BD header not making any sense upon leaving the local system.

The message header contains many attributes. The source pool indicates where the message originated from. The source and destination port are used for local inter-thread communication occurs while the destination and source port numbers are used for network communication. Port numbers are used since on systems connected by a network, pointers to data structures make no sense, thus the reference number of such ports are relayed through. The Ethernet Header is also stored here, giving information about the Ethernet address of the destination NIC. If this attribute has the Ethernet address equal to the local Ethernet address, then communication occurs between local user level threads. In addition a priority is stored that is used by the NIC to indicate the priority of transmission of such a message, allowing the use of multiple priority levels for transmission of messages.

5.1.4 Message Pools

A message pool is a repository of buffers that is used by the MESH system. Structured as a queue of buffers, each buffer is in turn mapped on to consecutive memory and from a different conceptual perspective subsequently mapped into one of the four queues in a DIPI.

A message pool (referred here simply as pool) is created by the user with a specified number of buffers. The maximum number of buffers are immediately created and placed in the pool. This occurs provided enough memory space exists in consecutive memory to create these buffers, otherwise pool creation fails. A concept used throughout the whole communication system is that of the *conservation of buffers*. At no particular time during MESH's operation is a buffer destroyed with space being reclaimed to consecutive memory and buffers are only created in pool creation. Whenever a buffer is placed in a queue, its only movement is that of being removed from one queue and placed in another. In local inter-thread communication, when a buffer is received, a buffer is extracted from the the destination pool and returned into the source pool. In network communication, the NIC queues will provide the substitute buffer that goes in the source pool. In MESH, only pointers are swapped during communication requiring few operations to communicate. Considering the fixed and limited amount of consecutive memory unto which buffers are mapped, conservation of buffer space also becomes a necessity.

5.1.5 Poolports

A poolport is simply a repository for one buffer. Whenever a poolport is created, it is associated to a given source pool. On poolport initialization, a buffer is retrieved from this source pool and attached to the poolport. Whenever a user level thread requires access to a buffer, it requests a message from the poolport and the buffer within it is returned. In other words, the poolport acts as an interface between the user level thread and the pool, with the user never allowed direct access to the pool. Upon giving up the buffer within it, the poolport retrieves another buffer from the source pool. If the source pool is found empty, the poolport merely stores a null pointer inside it.

When a user requests a buffer from the poolport and the poolport is empty, the poolport attaches itself on to a special queue belonging to a pool. This queue of poolports, called waiter queue, contains all those poolports which are currently waiting to be supplied with a buffer. In addition, the requesting user level thread is descheduled and stored inside this specific poolport. When a new buffer becomes available to the source pool, the pool passes the buffer on to the poolport at the top of the queue and reschedules the thread within it. The poolport is also removed from this waiter queue. Figure 5.21 shows a user level thread requesting the system for a buffer. Examples are shown of the system in different states.

A buffer that has been used by the user level thread and is not needed anymore is returned to the source pool through a poolport. Note that the poolport can be devoid of a buffer yet the buffer is still passed on to the source pool, allowing other poolports to receive this buffer, thus giving preference to poolports on the waiter queue rather than those outside this queue.

5.1.6 Ports

A port stores a queue of messages and acts as an interface between the user level thread and the NIC or between any two user level threads. Buffers are transmitted from port to port and the user level thread apply operations to retrieve these buffers from these ports. Ports play the most important role is setting up the message header. First of all, a port contains a reference number which is unique for a particular system and as such should be unique to all systems connected by a network. It also holds a multitude of buffers organized as a FIFO queue.

Multiple ports are implemented using an array of ports with the array index corresponding to the port number. There exist two types of ports, transmission ports and reception ports. There is no implementation difference yet depending on the particular setup of the communication primitives by the user level thread, this will make the ports behave accordingly. Every port has associated to it a pool which acts as the source of buffers.

When transmitting buffers, MESH compares the port's Ethernet address (set

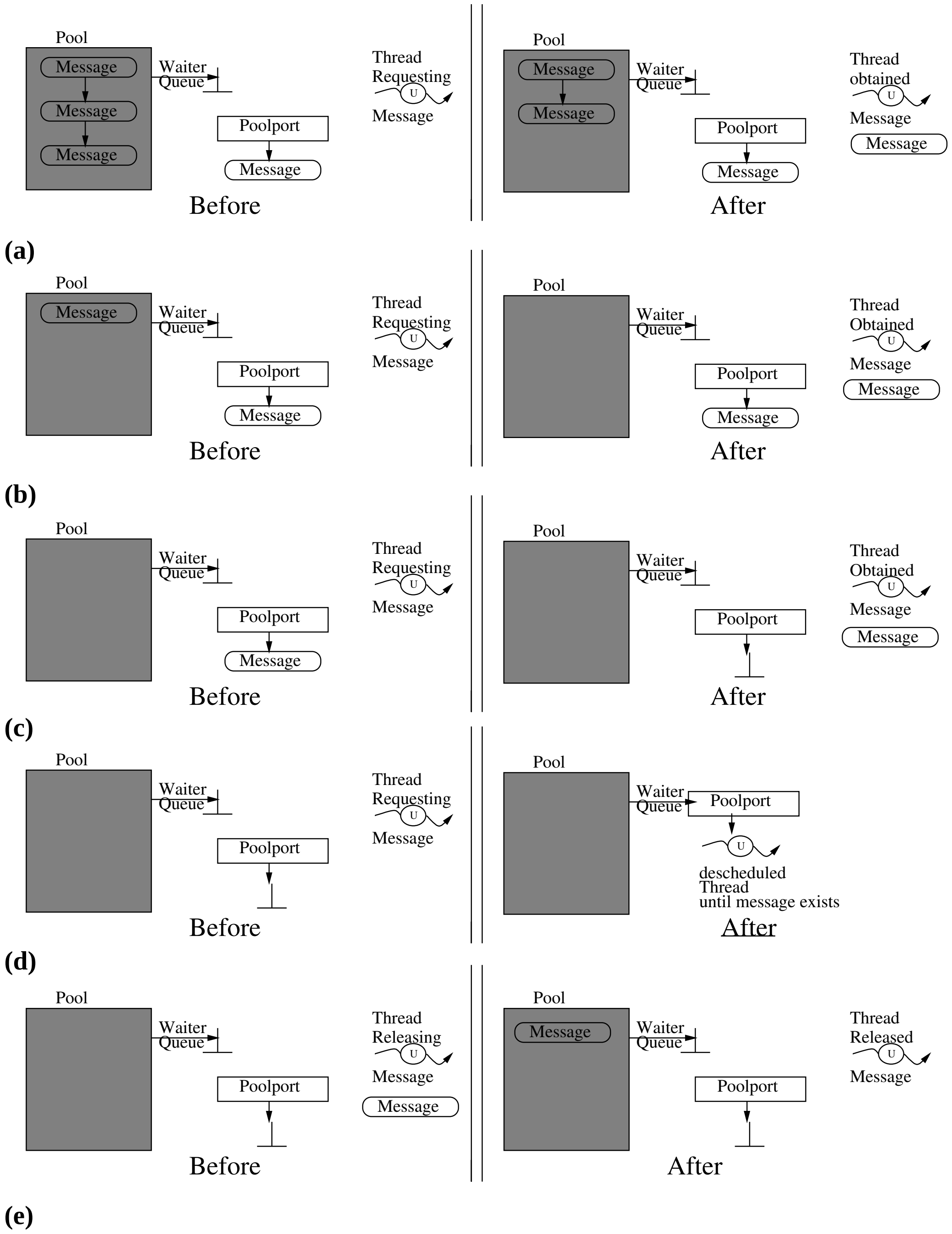


Figure 5.21: Pool and Poolport configurations; (a-d) user level thread request for a buffer; (e) user level thread releasing buffer

up by the user) to that of the local Ethernet address. If different, then transmission of the buffer is sent to the NIC after recording the source port number. This is through the send queue of the DIPI assigned to a particular NIC. The source and destination port numbers are also set up prior to transmission. If transmission is local i.e. within the same system, then the buffer is simply sent to the port with the given port number on the same system. Upon transmission, the NIC or the destination port (depending if communication is non-local or local respectively) returns a buffer that is added to the source's port pool.

When a buffer is received on the NIC's receive queue, the buffer is added to the received queue of the destination port. This is only done if a buffer can be taken from the destination port's receive pool. If so, the buffer is added to the NIC's receive queue to allow further buffers to be received by the NIC. If not, MESH performs a *selective discard*: the buffer is simply left in the NIC's receive queue and associated with data that is received in the future directed to the same port. A receive overflow warning message is reported to the user stating that all future buffers will be discarded unless more buffers become available on the source pool. After a limited number of buffers available to the NIC are used up, received data will start to be discarded. In essence the source is providing a buffer (containing the message) and in turn the destination needs to provide a buffer to replace the one lost (see Figure 5.22). This ensures that the number of buffers in the queues at different levels (DIPI level, pool level, and others) remains always constant. If more than one message is received on a particular port and replacement messages are available, they are simply added on to the message queue.

A user level thread might try to input a message from a port at any time. It cannot request selectively the source port and the first message on the port's message queue is returned. If the message queue is empty, the user level thread is descheduled and placed inside the port. Thus an input on a port is a potentially blocking operation. When a message arrives addressed to the port hosting the descheduled user level thread, the thread is rescheduled by being enqueued on the run queue again. Note that only one thread at a time can block on a port input since the port does not implement a thread queue. When more than one thread tries to input on the port and the port does not have messages on it, only the last

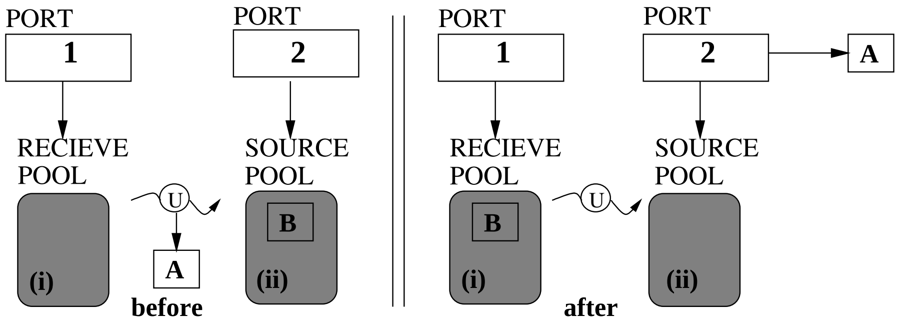


Figure 5.22: User Level Thread sending Buffer A to port 2 through port 1

thread will be stored inside the port, the rest will be lost and corrupted. While this is strange, the implementation was not changed trusting there might be a purpose for this configuration of affairs.

From a user's point of view, the philosophy behind MESH's *overflow control by selective discard* is that the application programmer should set a limit to how many buffers can be processed at any one time depending on the amount of communication expected. Receiving too many buffers on such a port would normally lead to system overloading. Buffering these messages causes system hogging and blocking the sender causes network congestion. Thus the messages are simply discarded when too many arrive at any one time. It is left to the application programmer to deal with such eventualities.

5.2 Communication API

In MESH, the application user has to follow a certain number of predefined tasks to ensure proper reception and transmission of a message.

To relay a message to another thread, first of all the application has to be designed properly, setting up port numbers to specific tasks. Then the user creates a message pool and from it obtains a message through a poolport. This poolport has to be associated with such a pool on creation. Upon return, which might be a blocking operation, the message is set up by the user with the relevant size and data within it. Finally the destination port number is recorded in the message header. Beforehand a source and destination port have to be created on the local system, with the source port having the Ethernet address of the local NIC and the destination port having the Ethernet address of the destination NIC. The destination port number should be the same as that recorded inside the message to be sent. For network communication a port will exist on another system having the same port number to which the message will be sent. Finally the message is sent on the source port. The source port will then relay the message to the destination port wherever that might be. After the ports have been set up, from the user's point of view, there exists no difference between local and non-local ports.

To receive a message, the application creates the destination port assigned to a specific receive message pool. This port has to be global to MESH since only one port with the same port number can exist at any one time in the whole system. The application then requests a message from the source port and when a message becomes available the application can use it. After using the message, the buffer space is returned to the system through a poolport associated with the Receive pool.

Very careful application design is required by MESH's system of communication. Whenever messages arrive, only the scheduler can re-schedule the blocked threads. Now MESH does not support pre-emptive scheduling and message receipt is detected through polling. Thus explicit scheduler entry points are necessary on a frequent basis to poll for communication and perform the necessary operations. This process becomes automatic if fine grain threads are used since every time a thread terminates, the scheduler is entered and communications polled. One does not expect more than one input poolport or input port per user level thread since MESH does not support more than one thread blocking at any one time on message input. Figure 5.23 shows the process of message transmission.

5.3 SMP-MESH Communication

With such a complex structure for inter-thread communication and with communication being crucial to the proper operation of parallel threads, one expects a

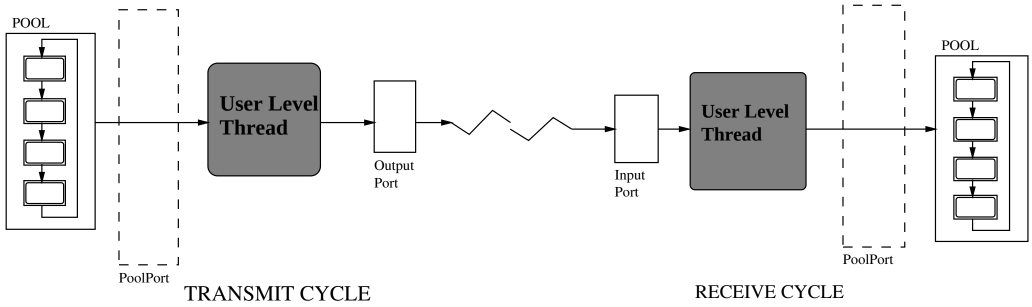


Figure 5.23: Message Transmission

large amount of operations related to communication occurring concurrently in a multiple processor shared memory implementation of MESH. Thus a synchronised method of accessing these structures is necessary to avoid corruption of data. Care must be taken not to introduce too many overheads, the slightest added latency having a large effect especially considering the highly optimised code of the uniprocessor MESH. The structures are protected using spin locks with the communicating threads having to busy wait at times before accessing some shared structure.

Almost all API constructs considered above have to be protected during concurrent access. Fortunately the sections to be protected consist mainly of enqueue and dequeue operations that do not involve time consuming operations, thus resulting in small critical sections that release the lock promptly.

Due to the intricate configuration of these communication constructs, it is common that one operation accesses a multiple number of data structures. A typical case is that of a port which accesses a message pool. Recourse can be made to one of three alternatives to synchronise access (see Figure 5.24) to these data structures:

- Hold a single global lock whenever a communication operation needs to be performed, or

- Hold a lock for each structure, such a lock being global to the structure in use, or

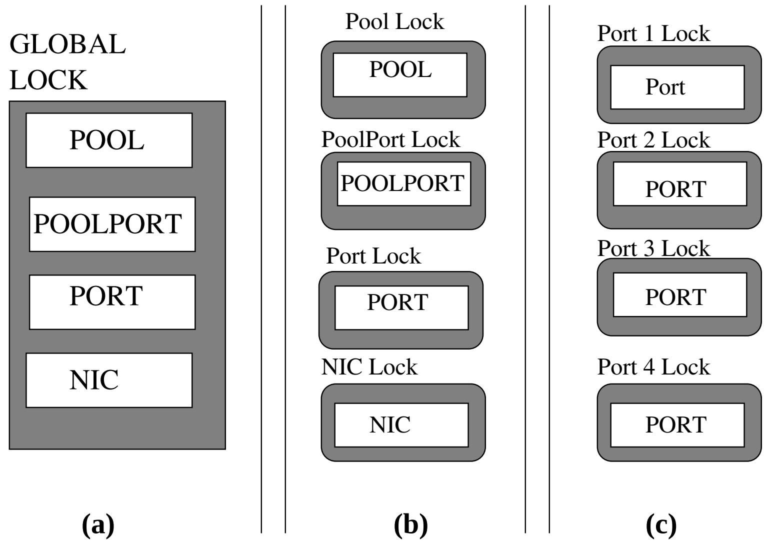


Figure 5.24: Possible lock granularity. (a) Global Communications Lock; (b) Lock protecting each structure; (c) Lock protecting each instance of each structure. Note that not all communication data structures are shown.

- Hold a lock for each instance of each structure.

The first alternative would give the simplest way out and would also ensure no nesting of locks. Yet due to the relative long sections of code per communication operation, the lock would be held for a long period of time and thus would increase contention for communication access. Also it would sequentialise all communication, removing any benefits of concurrent communication that is available on shared memory multiprocessors.

The second alternative allows operations on different structures to occur concurrently yet stops any operations on the same structure. Each lock protects the structure system wide. While this is a better solution than the above, it still does not provide enough granularity to allow the optimum amount of concurrency but is the best alternative whenever one does not expect frequent access to communication facilities, also reducing space complexity.

The third alternative might be seen as too fine a granularity. While it allows a high level of concurrency with the lock only protecting each instance, it requires that at some instances the scheduler holds multiple locks. It allows concurrent accesses to different instances of the structure and thus such a configuration only makes sense in threads employing a large number of such instances. The fact that queues of threads blocked on port input and poolport input are not supported entails that MESH will be used in a situation of frequent communication. SMP-MESH thus implements this configuration.

Interestingly a finer granularity of locks can be used where input and output on the same structure would use different locks, allowing these two operations to occur concurrently. This would require very specific enqueue and dequeue algorithms to avoid data corruption. The structures that cannot be accessed concurrently without the risk of data corruption are:

- Consecutive Memory,
- Message pools,
- Ports,
- Poolport, and
- Network Interface Card access.

User level threads block on an input operation from a port or a poolport whenever the requested data is not available. Whenever a thread deschedules it cannot hold the lock of the structure it sleeps on, since this will prevent other user level threads to access this structure and place data upon it and awaken the blocked user level thread. Thus this lock is released prior to descheduling. This lock is always released after the thread has been removed from a runnable state and added to the structure it blocks on, be it a poolport or a port. If another kernel level thread accesses the structure before the blocking user level thread has actually scheduled out, we end up with two concurrent version of the same user level thread. Yet one user level thread will simply proceed to block and the other will continue on from the deschedule point, thus causing no conflict. The blocked one will not be recorded anywhere and thus will never be rescheduled again. Also these is no increase in space complexity to these two threads since they run in the same memory space. Whenever a thread is re-scheduled after blocking it proceeds to retrieve the data that it blocked on. Thus the lock is retrieved again. In other words an input on a port or a poolport might at times consist of two busy waiting sessions, the first to access the structure and another time after the reschedule of the user level thread. This still causes very few latency problems since as such data is not expected to become available too quickly whenever a thread blocks unto it and when such data finally arrives, there will rarely be more than one thread trying to access this structure, each holding the lock. This is not expected apart from the case when a burst of data keeps the structure busy, yet this is still rarely expected to happen in SMP-MESH.

5.3.1 Communication Data Structures

Whenever a message pool is created, a number of messages are claimed from the consecutive memory area. All this occurs on the user level and thus creation of pools concurrently would lead to corruption of the message space. Creation of message pools is thus protected by a special global lock. There is only one area of consecutive memory in MESH and more than one lock is redundant.

Another instance when consecutive memory is requested for space is in the creation of a DIPI. Yet one does not expect more than one DIPI to be created concurrently since DIPI initialization forms part of the initialisation of MESH and occurs prior to the creation of different kernel threads, thus removing any concurrency problems.

All accesses to message pools are protected with a lock which is private to the specific message pool being accesses. Operations on message pools include adding and removing messages. These results in very small critical sections and one does not expect high contention to access message pools since the lock is not held for a long period of time. A separate lock per pool is used allowing several message pools to be created, accessed and destroyed concurrently.

Whenever accessing poolports, a lock needs to be held. In MESH we expect poolports not to be shared in between user level threads yet the system itself accesses these poolports to place messages into them and this can occur concurrently. On an input operation by a user level thread, two locks will have to be held at one time, the poolport's lock and the message pool's lock since a poolport tries to obtain its message from the message pool.

Accessing ports is also safe for concurrent use using a lock specific to each port. Ports are normally accessed by user level threads on input and output. They are also accessed by the NIC whenever a message is received on the system. All need to be holding the lock before changing any data inside the port. Ports also make access to the NIC for transmitting messages and this can also occur concurrently. Not more than one or two NICs are expected per system, thus a global lock is used that protects any access to the NIC. More than one lock per NIC was deemed superfluous since concurrent access to the NIC is not expected to occur frequently.

Ports are used to transfer data from one port to the next. Yet if a user level thread is transferring buffers through a source port to a destination port and the source and destination ports are the same, this leads to conflicts. In this case, the source port while holding its own lock will try to access the destination port by busy waiting on the lock it is holding. This will lead to a deadlock situation. In SMP-MESH this situation is not catered and corrected for since one does not expect such a situation to occur unless its a badly designed application and thus beyond the scope of the scheduler. A solution would be that of holding a single lock for all ports (as opposed to a lock for every instance of the port) yet this would stop any concurrent inter-thread communication on different ports. Another alternative is an extra check to realize the state of the system and thus deduce if the destination and source port are the same and act accordingly. Yet this would introduce further latencies and with the high speed communication present in MESH, any instruction which is not strictly necessary should be avoided.

Chapter 6

Results

6.1 Introduction

In this section, an outline of a series of benchmarks and the results obtained will be investigated. Some explanation of the values obtained will also be examined. Three versions of MESH were used: UNI-MESH, the original uniprocessor scheduler; SMP-MESH(1), the modified MESH with all the features to run on SMP machines yet still using one kernel level thread; and SMP-MESH(2), the modified MESH with a total of two kernel level threads. All results were executed on the SMP version of Linux 2.2.12. When SMP-MESH(2) was used, it was up to the Linux kernel to position each thread on different processors, which it did successfully at all times. Hardware-wise a dual Pentium-II 400MHz machine was used, with the machine equipped with 512 KB of Cache and 128MB of RAM. A minimum number of applications were run concurrently with the test runs, with the most computationally expensive application being X Windows. For time measurements the in-built kernel function `gettimeofday(...)` was used giving an accuracy of up to $\pm 1\mu s$. Each result quoted here is an average of around 10 measurements. Each subsequent section will concentrate on specific differences between the uniprocessor MESH and the SMP version.

The main aim of parallel computers is to achieve **speedup** which is the ratio of execution time of some code on a uniprocessor computer to the execution time of the same code on a multiprocessor computer. If T_s is the time taken to execute the code on a single processor and T_p is the corresponding time to execute it on a multiprocessor, then we have

$$SpeedUp \quad S = \frac{T_s}{T_p} \tag{6.1}$$

Intuition will make us believe that the obvious answer to this is that having n processors one should expect $T_p = \frac{T_s}{n}$ achieving a speedup of n . Yet as *Amdahl* properly noted, each task consists of parts which are parallelizable and others which must be run sequentially and thus ideal speedup is never reached.

6.2 Thread Construction

Some modification to UNI-MESH was made allowing new thread to be created on SMP machines. The space allocated to the new thread is obtained through `malloc()` which is protected by an appropriate lock. Thus when running SMP-MESH a delay due to the acquisition of the lock and its subsequent release is expected. 10,000 threads were created and the time for creation measured (see Algorithm 6.9). Only one user level thread is allowed to create new threads thus processor number does not affect the result. The time spent in acquiring the lock and then releasing it without any busy waiting, since the lock is always found free, is given. The results follow:

MESH Version	Total Time (μs)	Time/thread (μs)
UNI-MESH	306863	30.68
SMP-MESH(1)	309967	31.00
SMP-MESH(2)	308903	30.89

Table 6.3: Thread Creation Results

The difference between the SMP-MESH(1) and the SMP-MESH(2) results are not indicative of any speed-up since the second thread is left idle. The difference given probably arises from the fact that a kernel call is made when using the `malloc()` call and thus one is accessing the kernel is different states depending on the other running processes in the system. Note how on average the acquisition and releasing of the lock takes around $0.2\mu s$ which is quite a good result considering

Algorithm 6.9 Thread Creation Benchmark

```
Thread *thread[10000];
int i;
gettimeofday(tv1,NULL);
for (i=0;i<10000;i++) {
   thread[i] = Thread_Constructor(sample, 65536, 0);
}
gettimeofday(tv2,NULL);
```

that the bottleneck of user level thread creation is the kernel entry point through `malloc()` and this amount of time depends on the kernel state on entry. Also threads are not expected to be created often in MESH, with each thread having a substantial lifetime, even though not necessarily a longer execution time.

6.3 Thread Initialization

Whenever a new thread is created, it is normally placed into the run queue and by doing so, made executable by any available processor. To place the thread onto the run queue in the SMP-version first of all the lock has to be acquired. After placing the thread, the lock is released. Also if another kernel level thread is sleeping in the kernel on a semaphore, the thread placing the threads on to the run queue will have to notify the other thread to wake up before releasing the lock. Using Algorithm 6.10, 10,000 user level threads were created and placed on to the run queue. The time returned by the benchmark comprises only the placement on to the run queue and is not inclusive of creation time. Each thread placed into the run queue was made to loop infinitely thus making sure that the second thread of execution does not interfere with the results. Results can be seen in Table 6.4.

First of all the difference between the SMP-MESH(1) and SMP-MESH(2) total execution time can be attributed to the fact that in SMP-MESH(2) there is an entry into the kernel through the `semop()` call to notify the second thread to wake up and execute. Comparing the SMP-MESH(1) values and that of UNI-MESH we see that the time for run queue placement practically doubled (97% increase in

MESH Version	Total Time (μs)	Time/thread (μs)
UNI-MESH	7250	0.72
SMP-MESH(1)	14276	1.42
SMP-MESH(2)	14562	1.46

Table 6.4: Thread placement on Run Queue Results

operation time). Yet note that this length of time time does not only comprise the acquisition of the run queue lock. The operations involved are:

- The kernel level thread calculates self identity as discussed in Section 4.3,
- The run queue lock is acquired and then a check is made to see if there are any sleeping threads in the system. In SMP-MESH(1) there are never other sleeping threads,
- The thread is enqueued into the run queue,
- SMP-MESH checks to see if there are any other higher priority threads on the run queue (something that UNI-MESH does not do since only one kernel thread can place higher priority threads and will know when to deschedule itself), and
- Finally the lock is released.

Actually all these operations are all quite expensive computationally since they access data which might not reside in the cache and values are changing often, thus invalidating the cache. It is no surprise that such latency is introduced.

6.4 Context Switching

The most common event in a fine grained user level thread scheduler are context switches. These are expected to occur often in MESH and in fact is one of the highly optimized sections of MESH. Algorithm 6.11 performs a test on context switching. The traditional way of measuring context switching time is placing one thread that

Algorithm 6.10 Thread Run Queue Placement Benchmark

```
Thread *thread[10000];
int i;
for (i=0;i<10000;i++) {
   thread[i] = Thread_Constructor(sample, 65536, 0);
}
gettimeofday(tv1,NULL);
for (i=0;i<10000;i++) {
   Thread_Run(thread[i]);
}
gettimeofday(tv2,NULL);
```

yields a number of times. Yet in our configuration this is not advantageous since whenever the run queue is empty, the kernel level thread sleeps in the kernel and with one user level thread available a number of expensive wakeups and sleeping operation would happen often. Thus 20 threads were placed on the run queue removing the possibility of having the run queue empty apart from the beginning and end of the test when the threads are initiated or terminated. The time to place the threads into the run queue has to be included in the benchmark time result since upon placement, other kernel level threads will commence executing the new threads and thus would affect the context switching time. Each one of these 20 threads thus yields execution (by entering the MESH scheduler) 20,000 times before terminating. The results can be seen in the following table:

MESH Version	Total Time (μs)	Time/thread (μs)
UNI-MESH	27337	0.068
SMP-MESH(1)	208909	0.52
SMP-MESH(2)	798913	2.00

Table 6.5: Thread Context Switching time

Comparing the UNI-MESH results and the SMP-MESH results one immediately notices an increase of 665% in context switching time! This value is worrying but yet cannot be avoided. The fact that UNI-MESH was designed for uniprocessor use and had that aim all along, several optimizations were made to make this scheduler faster. Yet these optimizations are not portable in any way on to multiprocessor versions. One of such conflicting optimization in UNI-MESH is the fact that the currently running thread is always left on the run queue, something which is not possible in multiprocessor implementations. Also the fact that a context switch in UNI-MESH consists of so little machine instruction, it is very easy to produce a multi-fold increase in latency. Yet note that the context switch is still below the $1\mu s$ mark making it faster than any kernel context switch. While making use of optimization techniques introduced and used in UNI-MESH, SMP-MESH has to contend with other possibilities and situations. The time of execution of each context switch in SMP-MESH consists of first identifying the kernel level thread making the context switch, placing the user level thread on the run queue (in UNI-MESH the thread is simply there), then removing the next thread from the run queue and updating the priority value of the highest available thread to execute next (an O(16) operation). Finally the lock is released and the context switch proper executed (the latter being the only operation in common with UNI-MESH). Also the `Thread_Join()` instruction forces the joiner lock to be held for some time, thus introducing another latency. The above in fact consists mainly of the same operations as those used in the previous benchmark (refer to Section 6.3) apart from the removal of the next thread which is not expected to take many machine instructions. A context switch in fact takes 27% ((0.71-0.52)/0.71) less time to execute than the placement of new threads on to the run queue, mainly because of the same user level thread space being used over and over again in this case and there being a small amount of threads (20 in all), optimum use is made of the cache space, reducing the context switch time.

Comparing the SMP-MESH(1) and SMP-MESH(2) results we note that the dual processor version takes much more time than the single processor version which is the total opposite of what we aim to achieve. In fact there is an increase of 284% over the SMP-MESH(1) value of the context switch. Note that the benchmark is constructed to created a situation of the finest granularity for each thread. In fact all the user level threads will enter the scheduler very often due to the frequent context switches. The chance of some kernel level thread finding the lock held increases and so does the amount of busy waiting. On SMP-MESH(1) there

Algorithm 6.11 Context Switching Benchmark

```
void sample(Thread *p)
{
  int count=0;
  while (count < 20000) {
    count++;
    Thread_Yield();
  }
}

void Main(Thread *p, int argc, char** argv)
{
   Thread *thread[20];
   int i;
   for (i=0;i<20;i++) {
      thread[i] = Thread_Constructor(sample, 65536, 0);
   }

   gettimeofday(tv1,NULL);
   for (i=0;i<20;i++) {
      Thread_Run(thread[i]);
   }
   for (i=0;i<20;i++) {
      Thread_Join(thread[i],NULL);
   }
   gettimeofday(tv2,NULL);
}
```

is no circumstance where busy waiting will occur yet with more multiprocessor usage, we expect more threads to be busy waiting . This is a typical case of contention for a spin-lock and it is inherent in the use of spin-locks. In fact with such an increase in context switching we can safely assume that most of the time of the kernel level thread operation is spent busy waiting. This is a waste of processor time and thus a compromise will have to be found.

6.5 Granularity

So far no benefit has resulted from using an SMP version of MESH. Yet making use of less finely grained threads immediately yields its results. Using Algorithm 6.13 and Algorithm 6.12 we can measure the speedup achieved in concurrent processing. A total of 50 threads are created and placed on the run queue. Each thread in turn executes a loop having the size of this loop regulated by the Granularity g. Thus increasing g will make each thread execute for longer time. Inside the loop a simple calculation is executed causing further delay per iteration. After the loop, the thread yields execution until the user level thread is re-scheduled again where the same process is repeated to the point where all threads terminate. Each thread yields 10,000 times giving a total value of 500,000 context switches, all separated by a period of time proportional to the granularity. Increasing g gives an increase in execution time. This can be seen in Figure 6.25.

Note how the dual processor MESH execution is on average half that of the single processor MESH. Also note how the execution of UNI-MESH is proportional to MESH(1) with barely any difference apart from a small proportion of latency that is introduced by the implementation. Note that this applies for most granularities but not all. Figure 6.26 shows the same graph this time focusing on fine granularity. Here we can see how at very fine granularities the dual processor version seems to break down and actually takes longer to execute the same code than the uniprocessor version. The worst case of finest granularity is reached when the granularity is set to zero, i.e. for the case seen above when calculating context switching. Here there is a large increase in contention for the spin-lock and we see how a saturation of execution time is reached where the kernel level threads

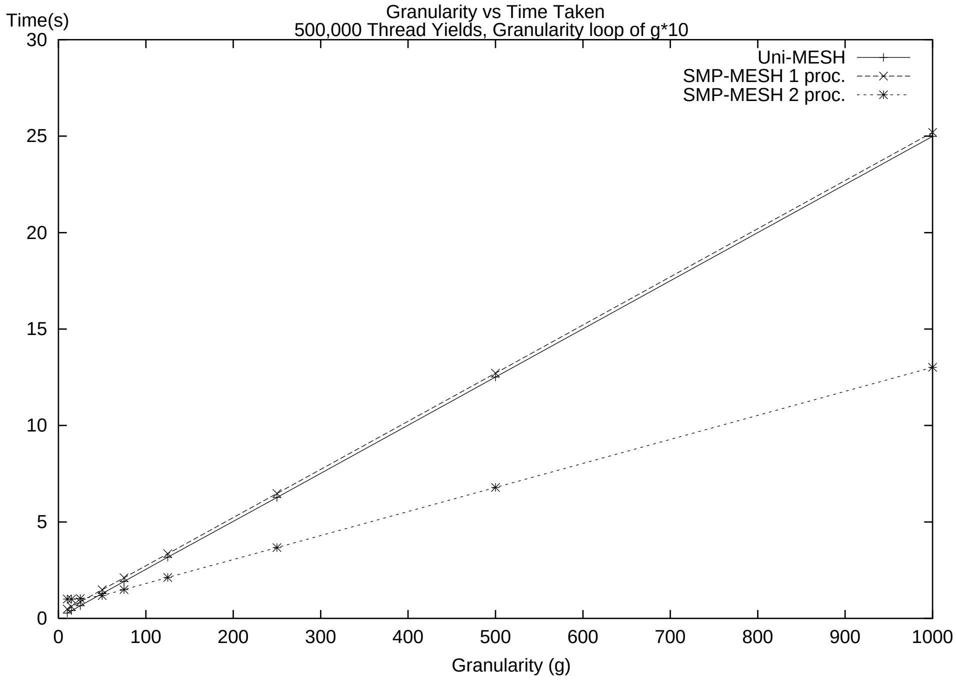


Figure 6.25: Total time of execution for different granularity

spends most of the time busy waiting and thus starts taking constant execution time regardless of decreased granularity. Also note how SMP-MESH(1) and UNI-MESH still take less time to execute at finer granularity indicating that this must be a problem inherent only in concurrent execution.

The figures shown previously do give some indication of what is happening, yet do not give a very clear picture of speedup. By dividing the time taken to execute all the iterations with the granularity amount, the time taken to for each granularity unit can be compared at different granularities. These can be seen in Figure 6.27. Note how at large granularities the execution time of the SMP-MESH(2) is actually half of that of SMP-MESH(1).

Examining the graph (Figure 6.27) one notices that while for most granularities UNI-MESH and SMP-MESH maintain a constant time per granularity unit, the SMP-MESH actually varies and even approaches a point where execution time equals that of the uniprocessor. Figure 6.28 shows in a more direct manner how speedup varies with granularity. At large granularities a speedup of 1.9 or more is

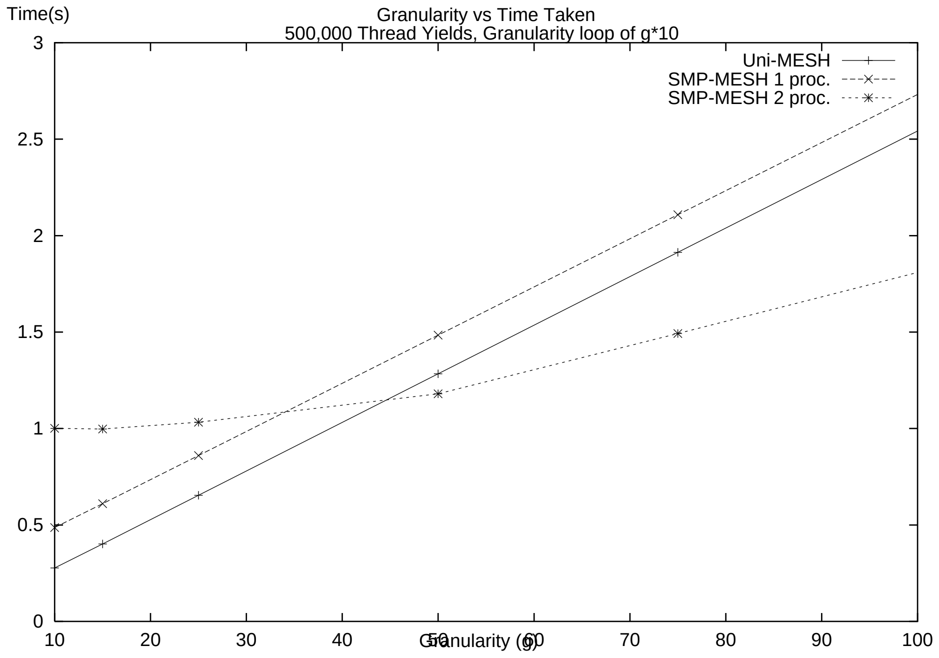


Figure 6.26: Total time of execution for different granularities at breakdown

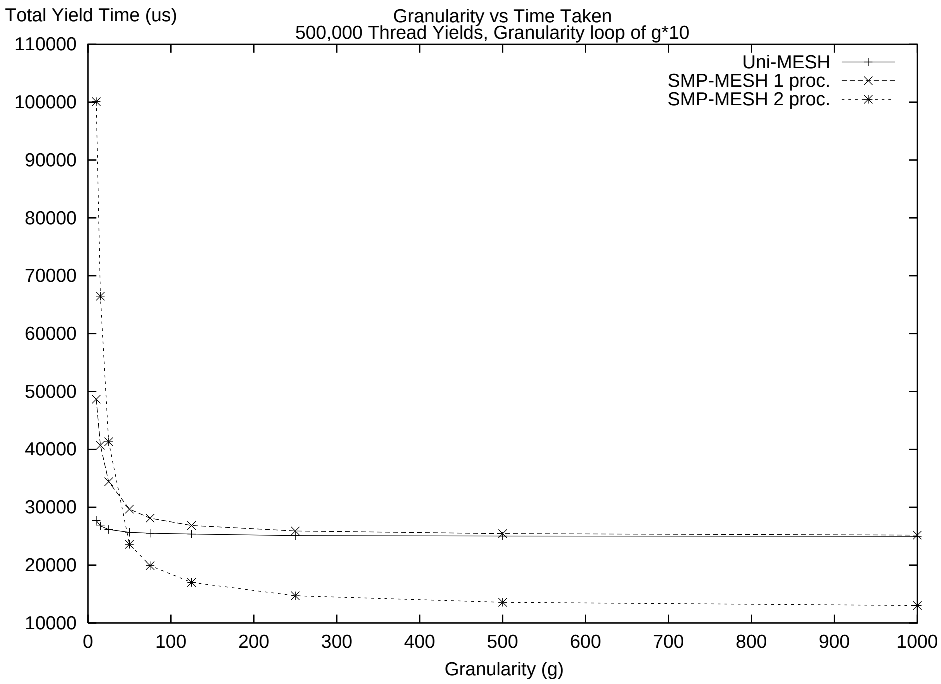


Figure 6.27: Execution time per Granularity Unit for different granularities

Algorithm 6.12 Granularity Thread

```
// smaller value makes each thread finer grained
#define g 1000

void sample(Thread *p)
{
  int i,n=0,count=0;
  while (count < 10000) {
    for (i=0;i<(g*10);i++)
           n=n+1;
    count++;
    Thread_Yield();
  }
}
```

Algorithm 6.13 Granularity Main Program

```
void Main(Thread *p, int argc, char** argv)
{
   Thread *thread[50];
   int i;

   for (i=0;i<50;i++) {
      thread[i] = Thread_Constructor(sample, 65536, 0);
   }

   gettimeofday(tv1,NULL);
   for (i=0;i<50;i++) {
      Thread_Run(thread[i]);
   }
   for (i=0;i<50;i++) {
      Thread_Join(thread[i],NULL);
   }

   gettimeofday(tv2,NULL);
}
```

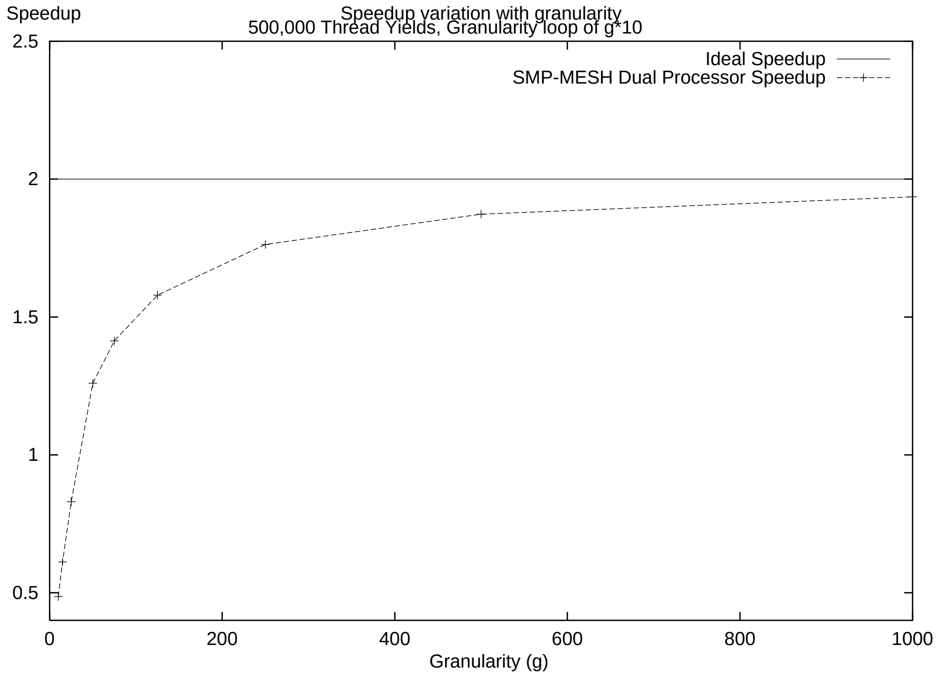


Figure 6.28: Variation of Speedup with Granularity

easily achieved. Yet at around a granularity of 200 and less most of the benefits of SMP implementations are lost. This arises from increased contention for the spin-locks.

Examining the time taken per granularity unit at fine granularities we discover some interesting facets (refer to Figure 6.29). First of all even for the UNI-MESH and SMP-MESH(1) versions we detect a slight increase in time. This arises mainly due to hardware. Since user level threads are being scheduled in and out at a very fast rate, the hardware will be constantly invalidating the data in the cache and thus issuing requests on the memory bus. This causes a performance bottleneck since data will be unavailable when the cache is being updated. Also in the SMP-MESH(1) results we also have the locking of the data bus on the motherboard through the use of specific atomic instructions when acquiring the lock and thus increases the bottlenecks.

The SMP-MESH(2) results clearly show that the added latency is introduced due to spin-lock contention. In fact it is not advantageous to use a double processor

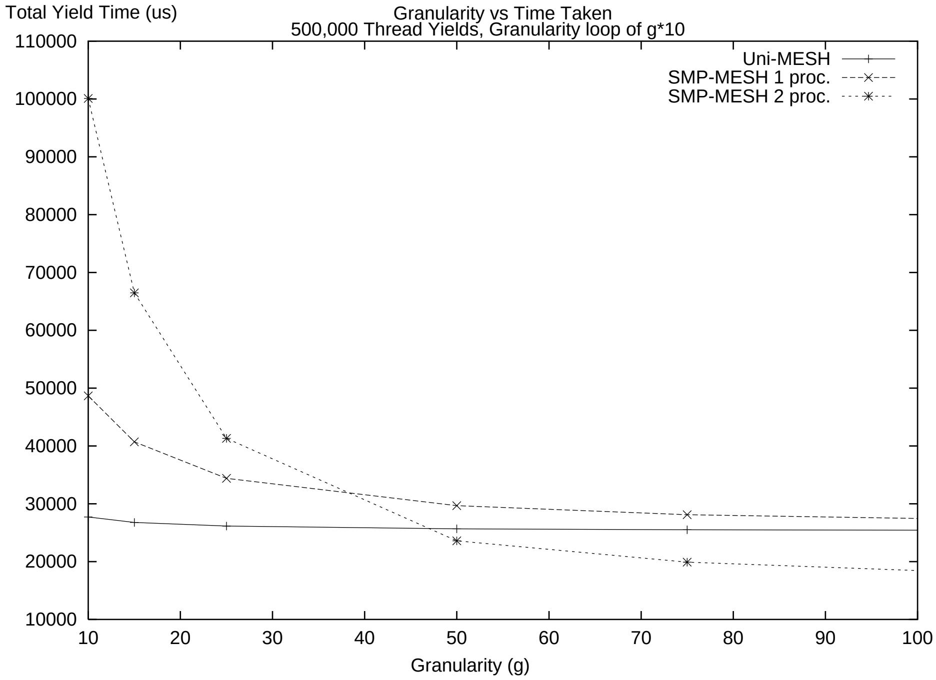


Figure 6.29: Execution time per Granularity Unit for Fine Granularities

below the granularity of 50. Yet this rarely happens in practice since such a granularity barely gives the thread any time to perform useful operations.

6.6 Cache Bottleneck

In Section 3.7.1, we saw how a busy wait is performed by spinning on a read and then only when there is the possibility of acquiring the lock, one performs a write operation on to the lock. This ensures that the multiprocessor caches are not being continuously invalidated and thus have to be updated. An experiment was performed in which the same algorithm as that used for the granularity test (Algorithm 6.13) was executed after setting g to 50. This granularity setting gives quite a large number of busy wait occurrences (as indicated by graphs discussed above) but still is not the worst case of latency. Two sets of data were recorded, the first by spinning on write and the other by spinning on read every time using the SMP-MESH(2) version. The results follow where the time for each granularity unit is given:

Cache Disabled (μs)	Cache Enabled (μs)
24067	23568

Table 6.6: Spinning on Read Comparison

When using Spin-On-Read an improvement of $499\mu s$ (2% decrease in time) over using Spin-On-Write is observed. The major effect that cache updating can have on execution is clearly visible.

6.7 Communication

SMP-MESH has modifications over UNI-MESH to allow concurrent communication. Yet a proper benchmark giving speedup cannot be constructed. This is due to the fact that having two threads communicating in a ping-pong fashion (one sends, other receives and send back, and others) on a multiple kernel level thread situation, after executing one of the two user level threads the kernel level thread will find the run queue empty and sleep in the kernel. This will be repeated countless times and will give an erratic result due to the expensive kernel function call to change the semaphore. Another benchmark could be that of sending a huge amount of data to another thread while the other thread simply receives this data. Yet the *selective discard* (where messages are discarded if too many are received) technique in unsynchronised user level threads results in messages being discarded, thus giving incorrect results.

Latencies were introduced upon implementing communication in an SMP perspective. This involves the acquisition of several locks for each operation. A ping-pong program was implemented and run with the total of 10,000,000 packet transmission. On message input, if the user level thread blocks on communication, it is placed on a special structure when descheduled and not placed again in the run queue, thus making this benchmark different from the case of a context switch with extra latencies and thus cannot be compared. The execution time for every data packet communication follows :

SMP-MESH(1) (μs)	UNI-MESH (μs)
7.20	1.02

Table 6.7: Ping-Pong Communication

An increase in execution time of 605% is recorded over the UNI-MESH communication. This value should simply be used in an indicative way since it still includes run queue access and kernel level thread identity values which shadow the true latency introduced due to communication synchronisation primitives.

Chapter 7

Conclusions and Further Work

This thesis has presented SMP-MESH, a scheduler able to make use of multiprocessor machines effectively.

Some preliminary concepts have been investigated in the first chapter. Reference was made to shared memory multiprocessor architectures enabling processes to execute concurrently. Also primitives available for synchronising access to shared resources were examined. In the next chapter, MESH in its original form was introduced and presented in considerable detail. The techniques used to provide very low latency scheduling and inter-thread communication were also discussed. Scheduling on SMP machines in general was then introduced together with the obvious problems encountered under such circumstances.

The following chapter then treated the design and implementation details of SMP-MESH, describing both scheduling, inter-thread and network communication details. Several other major problems encountered were discussed and a variety of solutions were introduced together with the reasons for choosing a particular solution in the implementation. The use of spin-locks to access various shared resources and other issues such as well behaved processor idling and load balancing were thoroughly discussed.

Several benchmarks were introduced and explained in Chapter 6. Comparative results between uniprocessor MESH and SMP-MESH were given together with speedup results on a dual processor computer. The reasons for the increase in latency in some situations were explained and the effect of thread granularity on scheduling was examined.

7.1 Further Work

A large of amount of work can still be made on SMP-MESH to make it even more efficient. Most of these tasks have already been discussed and designed, yet lack of time has prevented their implementation. Below some of the major possible improvements are mentioned.

SMP-MESH is fully operational. Some problems exist with the `Thread_Join()` command especially when this command is nested with other `Thread_Join()` commands. This problem is also present in the uniprocessor MESH and thus no effort to rectify this problem has been made, preferring to contact the original author of MESH in the event that the API would have to be changed. Also an SMP version of threads blocking on a common shared timer has not been implemented.

One of the tasks still to be implemented is manual mapping of kernel level threads on to their respective processors. With the current implementation of SMP-MESH, the Linux kernel dynamically reallocates the kernel level threads to processors during execution. Extensions to the kernel itself are necessary where a new kernel system call is provided that binds each kernel level thread to a specific processor. This also improves cache reuse for each kernel level thread.

There is much space for optimization of the scheduler to further improve performance. The kernel level thread identification algorithm can be replaced by introducing access to a special register reserved for such use. Also contention to spin locks can be reduced by implementing concurrent enqueue and dequeue operations, or even lock-free data structures where contention would drop substantially. In the event of still having critical sections, they should all be coded in assembly language thus reducing even further any latency in these sections and further optimizing such operations.

An ideal situation would be that where the user is not required to provide explicit manual entry points to the scheduler but pre-processing by the compiler would enable automatic scheduler entry points to be introduced. This would also provide automatic communication polling instructions. All this creates a more proper abstraction layer between the user and MESH.

While results returned are satisfactory, more testing is required especially in the case of systems with more than two processors on board. The unavailability of such machines has made such tests impossible.

Another interesting investigation would be to implement different run queue configurations such as batch run queues making better use of caches consistency for user level threads. This would provide a series of results that can be compared for efficiency.

7.2 Closing Note

This project performed the next obvious step in the evolution of MESH, that of bringing it into the shared memory multiprocessor architecture. Care was taken to preserve all its low latency scheduling techniques together with its high speed communication constructs. Overheads were inevitable yet the benefit of added multiprocessor speedups are evident.

SMP-MESH achieves the true aim of SMP, that of executing tasks in the least amount of time, using the least amount of space and at minimal financial costs. The end result is a very fast user level thread package for the C language, running on the Linux operating system, making use of all the features inherent in multiprocessor systems with shared memory.

Bibliography

[1] Almasi, G.S. and Gottlieb, A. *Highly Parallel Computing.* (2nd ed.). Benjamin/Cummings Publishing Company, 1994.

[2] Anderson, T., Lazowska, E. and Levy, H. *The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors.* Technical Report 88-09-04, University of Washington, September 1988.

[3] Anderson, T. *The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors.* Technical Report 89-04-03, University of Washington, 1989.

[4] Anderson, T., Bershad, B., Lazowska, E. and Levy, H. *Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism*, Technical Report 90-04-02, University of Washington, April 1990.

[5] Anderson, T., Bershad, B., Lazowska, E. and Levy, H. *Thread Management for Shared-Memory Multiprocessors.* University of Washington, Seattle, 1991.

[6] Barnes, F., *Symmetric Multiprocessing: Locking Processes to Processors, Synchronization and Communication.* available Email: Fredrick.Barnes@cern.ch, 1999.

[7] Barnes, F. *Automated Poll Insertion.* CERN, August 1999.

[8] Bershad, B., Anderson, T., Lazowska, E. and Levy, H. *User-Level Interprocess Communication for Shared Memory Multiprocessors.* University of Washington, 1990.

[9] Bershad, B. *Practical Considerations for Non-Blocking Concurrent Objects.* Carnegie Mellon University, May 1993.

[10] Bhoedjang, R., Ruhl, T., Hofman, R., Langendoen, K. Bal, H. *Panda: A portable Platform to Support Parallel Programming Languages.* Vrije Universiteit Amsterdam, Holland.

[11] Boosten, M. *MESH: Messaging and Scheduling for Fine Grained Parallel Processing on Commodity Platforms.* Eindhoven University of Technology, 1999

[12] Boosten M. *Fine Grained Processing on a Commodity Platform: a Solution for the ATLAS Second Level Trigger.* Ph.D. Thesis, Eindhoven University of Technology, 1999

[13] Culler, D.E., Schauser, K.E. and Eicken, T. *Two Fundamental Limits on Dataflow Multiprocessing.* University of California, Berkeley.

[14] Culler, D.E., Sah, A., Schauser, K.E., Eicken, T. and Wawrzynek, J. *Fine-grain Parallelism with Minimal Hardware Support : A Compiler-Controlled Threaded Abstract Machine.* University of California, Berkeley, April 1991.

[15] Deitel, H.M. *Operating Systems.* (2nd ed.). Addison Wesley, 1990.

[16] Dennis, J.B. and Gao, G.R. *Multithreaded Architectures: Principles, Projects and Issues.* ACAPS Technical Memo 29, February 1994.

[17] Feeley, M., Chase, J. and Lazowska, E. *User-Level Threads and Interprocess Communication.* Technical Report 93-02-09, Department of Computer Science and Engineering, University of Washington, February 1993.

[18] Foster, I., Kesselman, C. and Tuecke, S. *Nexus: Runtime Support for Task-Parallel Programming Languages*, Argonne National Laboratory, Argonne, Illinois, 1996.

[19] Goodeve, D.M., Davy, J.R., Dew, P.M and Nash, J.M. *The Consistency Properties of a Scalable Concurrent Queue.* Research Report Series 96.35, University of Leeds, December 1996.

[20] Haines, M. *On Designing Lightweight Threads for Subtrate Software.* Proceedings of the USENIX 1997 Annual Technical Conference, Anaheim, California, January 1997

[21] Herlihy, M. *Wait-free Synchronisation. ACM Transactions on Programming Language and Systems.* January 1991.

[22] Keppel, D. *Register Windows and User-Space Threads on the SPARC.* University of Washington, Seattle, August 1991

[23] Keppel, D. *Tools and Techniques for Building Fast Portable Threads Packages.* University of Washington, Seattle, 1993.

[24] Kleiman, S., Smaalders, B., Stein, D. and Shah, D. *Writing Multithreaded Code in Solaris.* SunSoft Inc.

[25] Kleiman, S., Voll, J., Eykholt, J., Shivalingiah, A. Williams, D., Smith, M., Barton, S., and Skinner, G. *Symmetric Multiprocessing in Solaris 2.0.* SunSoft Inc., California.

[26] Knezo, E. *Implementation of SMP-MESH, design decisions and refined work plan*, CERN, Switzerland, 1999.

[27] Lim, B.H. *Reactive Synchronisation Algorithms for Multiprocessor.* Massachusetts Institute of Technology, October 1994.

[28] Markatos, E., Crovella, M. and Das, P. *The Effects of Multiprogramming on Barrier Synchronisation.* University of Rochester, 1991.

[29] Markatos, E. *Scheduling for Locality in Shared Memory Multiprocessors.* Ph.D. thesis, University of Rochester, 1993.

[30] Marsh, B., Scott, M., LeBlanc, T. and Markatos, E. *First-Class User-Level Threads.* In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pages 110-121. October 1991.

[31] Michael, M. and Scott, M. *Simple, Fast and Practical Non-Blocking and Blocking Concurrent Queue Algorithms.* Technical Report TR600, University of Rochester, 1995.

[32] Michael, M. and Scott M. *Relative Performance of Preemption-Safe Locking and Non-Blocking Synchronisation on Multiprogrammed Shared Memory Multiprocessors.* University of Rochester, 1997.

[33] Michael, M. and Scott, M. *Non-Blocking Algorithms and Preemption-Safe Locking on Multiprogrammed Shared Memory Multiprocessors. Journal of Parallel and Distributed Computing*, 1998.

[34] Moores, J. *CCSP - A portable CSP-based run-time system supporting C and Occam, Architectures, Languages and Techniques for Concurrent Systems* . WoTUG-22: 147-168, 1999.

[35] Mueller, F. *A Library Implementation of POSIX threads under UNIX.* Florida State University, January 1993.

[36] Powell, M.L., Kleiman, S.R., et al.,*Solaris SunOS 5.0, SunOS5.0 Multi-Thread Architecture.* Sun Microsystems, 1991.

[37] Rodgers, P. *CERN prepares for the LHC and beyond.* In *Physics World*, vol 13, no. 5, p 10-11, May 2000.

[38] Scott, M.L. and Mellor-Crummey, J.M. *Fast, Contention-Free Combining Tree Barriers for Shared-Memory Multiprocessors. International Journal of Parallel Programming.* August 1994.

[39] Segall, Z. and Rudolph, L. *Dynamic Decentralised Cache Schemes for a MIMD Parallel Processor.* Proceedings of the 11th Annual International Symposium on Computer Architectures, 1984.

[40] Silberschatz, A. and Galvin, P. *Operating Systems Concepts.* (4th ed.), Addison Wesley, 1994.

[41] Stein, D. and Shah, D. *Implementing Light Weight Threads.* Summer'92 USENIX, June 1992.

[42] Stevens, W.R.*Advanced Programming in the UNIX Environment.* Addison Wesley Professional Series, 1993.

[43] Tanenbaum, A.S.*Distributed Operating Systems.* Prentice-Hall International, 1995.

[44] Theobald, K. *Advanced Compilers, Architectures and Parallel Systems.* In *Panel Sessions of the 1991 Workshop on Multithreaded Computers*, ACAPS Technical Memo 30, p3., April, 1993.

[45] Thorne, M. *Computer Organization and Assembly Language Programming.* (2nd ed.), Benjamin/Cummings Publishing Company, 1994.

[46] Treleaven, P.C. *Control-Driven, Data-Driven, and Demand-Driven Computer Architecture (Abstract).* Parallel Computing vol. 2, 1985.

[47] *Turbo Assembler 3.0, Quick Reference Guide.* Borland International, 1991.

[48] *Using and Porting GNU cc.* WWW.
`URL:http://www.delorie.com/gnu/docs/gcc`.

[49] Vahalia, U. *UNIX Internals.* Prentice Hall, 1996.

[50] Valois, J.D. *Lock-Free Data Structures.* Ph.D. Thesis. Rensselaer Polytechnic Institute, Troy, New York, May 1995.

[51] Vella, K. *Seamless Parallel Computing on Heterogeneous Networks of Multiprocessor Workstations.* Ph.D. Thesis, Department of Computer Science, University of Kent, 1998.

[52] Vella, K. and Welch, P. *CSP/Occam on Shared Memory Multiprocessor Workstations.* In *Architectures, Languages and Techniques.* B.M. Cook (Ed.), IOS Press, 1999.

[53] Wisniewski, R., Leonidas, K. and Scott, M. *High Performance Synchronisation Algorithms for Multiprogrammed Multiprocessors.* University of Rochester, December 1994.

Appendix I

Diskette Contents

The diskette supplied with this document contains all the source code to execute and develop applications on SMP-MESH. It should run on any official release of the Linux operating system. To make use of multiple processors, the Linux kernel has to be compiled with the SMP feature enabled.

Just like its predecessor, MESH, this package requires the *bigphysarea* patch to be applied to the Linux kernel and the *consequetive* driver to be installed. In addition, it requires an area of memory to be allocated upon Linux initialization and also the use of the Alteon or the Intel fast Ethernet cards for network communication.

Linking the applications to SMP-MESH is transparent to the user and is no different to linking applications to MESH. A settings file exists called `Global.h` where `NOOFPROCESSORS` should be set to the number of processors present on the system and `SMP` should be defined. `THREADSTACKSIZE` defines the stack space allocated to each kernel level thread apart from the parent thread whose stack is controlled by the operating system. All other variables in this file are for testing purposes.