Parallel Programming Support within the Linux Operating System

Jens Langner (4199928)
Supervisor: Dr. Joaquin Sitte

(ITB 434) Parallel Computing
Queensland University of Technology
School of Software Engineering

May 27, 2002
Contents

1 Introduction 2
  1.1 Parallel Programming ........................................... 2
  1.2 Technical Report Tasks ........................................ 2

2 Synchronous Multi Processing (SMP) 3
  2.1 Concepts .......................................................... 3
  2.2 Available Hardware ............................................. 3
  2.3 Shared Memory Programming .................................... 4
    2.3.1 Sharing concepts .......................................... 4
  2.4 Atomic operations ............................................... 5
  2.5 Cache Line Size ................................................ 6
  2.6 Linux scheduler issues ......................................... 6
  2.7 Linux Threads .................................................. 7
    2.7.1 Program Structure ......................................... 8
  2.8 System V Shared Memory ...................................... 10
    2.8.1 Program Structure ......................................... 10
  2.9 Memory Map Calls ............................................. 12
  2.10 Summary ....................................................... 13

3 Linux Clusters 14

4 Conclusion 15
Chapter 1

Introduction

1.1 Parallel Programming

The fundamental concept of parallel programming is to speed up the execution of a program by dividing it into smaller pieces (fragments). This fractions will than be spread over an amount of processors available to the underlying operating system.

A program that can be spread over \( N \) processors might execute \( N \) times faster than it would on a single processor system. Of course analysis has shown that this ideal assumptions can not be made because of occuring overhead by the operating system or the used parallel programming environment. Moreover the maximum speedup is dependent on the size of the fraction that can only be executed sequentiel, what is defined in the so-called Amdahl’s Law:\(^1\):

\[
S(n) = \frac{1}{f + (1 - f)/n}
\]

This shows that the sequentiel fraction \( f \) has to be almost zero to really obtain a maximum speedup of \( N \). Another common estimation for the speedup is, that the maximum obtainable speedup scales at \( N/\log_2 N \). Also former practial experiences has shown that, for example on a 4 processor machine a maximum speedup of \( S(n) = 1.9 \) is a reasonable speedup one can assume while thinking of parallelism within a program.

1.2 Technical Report Tasks

As one has decided to introduce some sort of parallism within a program this technical report should give a short overview of the different available mechanisms within and available for the Linux operating system.

This document will discuss two different approaches for parallel computing available to Linux developers: *Synchronous Multi Processing (SMP)*, *Cluster Systems* and will also take a short view into the advantages of each of this approach with some discussions of the underlying processor architecture.

Chapter 2

Synchronous Multi Processing (SMP)

The following chapter about SMP support within the Linux operating system should give the reader a short overview of the different possibilities. Especially available SMP Hardware and how to use this hardware with modern threads libraries should be discussed here to show that Linux is highly capable of synchronous multiprocessing and can compete with other platforms that formerly introduced SMP.

2.1 Concepts

To give a overview about the possibilities in programming for SMP systems within the Linux operating system, it is necessary to understand the concepts of SMP. Synchronous Multi Processing describes the concept of having more than one processor available to a operating system where mainly the OS itself handles the distribution of tasks automatically. The scheduler of this operating system will automatically spread out the different tasks/processes to all available processors. So for the programmers view this will be transparent and not visible at all. Also the operating system itself will run on different processors where so-called ”critical tables” and ”mutexes” make sure that ”critical regions” of the operating system will run only on one processors. Of course, the difficulty in programming such an operating system is, that it has to be divideable into independent parts that can be spread over a bunch of processors.

2.2 Available Hardware

In early times SMP Hardware wasn’t available at all or just too expensive. But as time went by, nowadays SMP Hardware systems are nearly available everywhere. Of course this fact is the result of standarization within the SMP Hardware platforms. Linux itself also profited of that standarization which was mainly initiated by Intel through the MPS¹ 1.1 and 1.4 specification. [4]

¹MultiProcessor Specs: http://www.intel.com/design/pro/datashts/242016.htm
Today Linux supports many SMP Hardware platforms that are mainly based on that specification of Intel. The only non-Intel SMP Hardware platform also supported by the Linux Operating System is the Sun4m multiprocessor platform for SPARC machines. Also the new AMD-MP SMP Platform comes into shape since some months but the primary way in which systems that comply with MPS differ is in how they implement access to physically shared memory.

2.3 Shared Memory Programming

As discussed before, within a SMP platform all processors are sharing the same public memory, even if they have their own cache memories. To fully understand the concepts behind SMP it is necessary to know how shared memory communication between processors really work.

One can assume that simply one processor stores a value into memory and another processor loads it; but unfortunately that isn’t quite that simply at all. For example, the relationship between processes and processors is very blurry; however, if we have no more active processes than there are processors, the terms are roughly interchangeable.

2.3.1 Sharing concepts

There are two main concepts concerning shared memory programming. Shared Everything and Shared Something.

Both of these models allow processors to communicate by loads and stores from/into shared memory; the distinction comes in the fact that shared everything places all data structures in shared memory, while shared something requires the user to explicitly indicate which data structures are potentially shared and which are private to a single processor.

So when people speak of shared memory programming they often implicitly mean some kind of mixture of this both concepts, because other mechanisms like the System V Shared Memory support within Linux/unix depends on the concepts of shared something where the other main mechanism of parallel programming by using reentrant threads library (i.e. POSIX PThreads Library\textsuperscript{2}) uses the concept of a shared everything environment.

Shared Everything

So Shared Everything for example provides the programmer with an easy way to incremental port already existing serial implementations of a program in an shared everything parallel way, because you do not need to analyse which data need to be accessed by more than one processor.

As discussed before the most common type of shared everything programming support is a reentrant threads library like the well-known POSIX PThreads library package. Threads are essentially ”light-weight” processes and, most importantly, share access to a single memory map.

\textsuperscript{2}as defined by the POSIX 1003.1c threads standard
It also is fairly obvious that shared everything only works if all processors are executing the exact same memory image; you cannot use shared everything across multiple different code images.

**Shared Something**

The main idea behind the concept of *Shared Something* is to share only what is really needed to be shared. Especially it makes it easier to predict and tune performance. The only few problems that arise with introducing *Shared Something* mechanisms within a program is, that you initially have to know beforehand what really needs to be shared and that you can’t easily port existing serial implementations with that approach.

Once you have decided to go for the much safer way of shared something programming, Linux provides you with two similar mechanisms to introduce *Shared Something* memory support. The first one is also the most well-known possibility to use the standardized *System V Shared Memory* support.

The other possibility is a memory mapping facility where no defined standardization really exists and therefore also the implementation varies widely across different UNIX systems. In a following section we will only briefly describe that `mmap()` system call, because it is really not widely used and mainly the *System V Shared Memory* mechanism is preferred for shared something approaches.

### 2.4 Atomic operations

Independent of the memory sharing concept you have choosen for a particular implementation of a parallel program, the result will pretty much be the same. You will get a pointer to a memory area that is accessible by all of your processes. So one might think that he or she just access this area like it were located in a local memory area. But this is really not quite the correct approach for this, because there is still the possibility that two processes are accessing the same memory area in a manner that it is not sure what will be the end state of that area. Therefore **Atomicity** refers to the concept to assure, a operation on an object is always valid and that two processes never can access that object at the same specified time. So, unless special precautions are taken, only simple load and store operations that occur within a single bus transaction are really atomic.

In addition to this, modern compilers like gcc will often perform strong optimizations to the code that will automatically eliminate special memory operations that are needed to ensure that other processors can see what a another processor has done.

There exists many approaches to solve that atomicity problem while programming a parallel program, but beside approaches like introducing `volatile` symbols in front of every variable where atomicity should be ensured and `LOCK` statements within special assembly inline functions of gcc, there is also the more

---

3. see also "man shmop" within the manpages of Linux
4. i.e. aligned 8, 16 or 32-bit operations
5. refer gcc manual for more detailed information about Volatility mechanisms
elegant variant to use special *mutual exclusion* mechanisms provided by either the *PThreads library* or the semaphore system of the *System V IPC* concept.

### 2.5 Cache Line Size

Of course the cache line size of a processor system can have a dramatical impact on the performance of a SMP system. The MPS standard itself normally defines that references to be coherent no matter what type of caching is used. In real life it seems to be a bit different from the standardization efforts of Intel. If one processor writes to a particular line of memory, every cached copy of the old line must be invalidated or updated. So if more than one processor writes some data to different portions of the same cache line, a lot of cache and bus traffic may result, effectively to pass the line from one processor cache to the other one. To solve that problem one has to organize his data so that every portion that is accessed in parallel tends to come from another cache line.

If you assume that the cache line size is $c$ and the addresses of two different items are $a$ and $b$, you should make sure that:

$$((\text{int}) a) \& \neg (c - 1)) \neq ((\text{int}) b) \& \neg (c - 1))$$

Otherwise both items will be on the same cache line which bring up the above discussed disadvantages.

### 2.6 Linux scheduler issues

As already mentioned in an earlier section, the relationship between processors and processes can be quite blurry. This is the point where the scheduler of a certain operating system comes into discussion. In this section we would like to bring some light into this blurry situation by explaining the main tasks of the Linux scheduler and the new support mechanisms that were introduced to Linux in kernel 2.0 to support SMP hardware.

As the name of the Linux scheduler itself implies, the main task of it is to schedule processes within the operating system for execution time. When also SMP comes into that scheme, the scheduler also has to distribute this processes on different processors accordingly to some scheduling algorithms. This scheduling is mainly done transparent to the user. That means that a programmer normally has no chance to define on which processor a process should run. We use the term ”normally” here, because there are of course some possibilities to control/tune the behaviour of the scheduler, but this mechanisms can more be counted as hacks than official supported standarizated functions and so we will not take a deeper look into that mechanisms and focus only on the standard possibilities for parallel programming under Linux.

First of all, Linux doesn’t really support the ability to create real kernel level threads that will automatically be distributed over all processors. Therefore

---

6 such as `pthread_mutex()` or the `semop()` operations
7 also known as *false sharing*
8 normal Pentium L1 cache is 32 bytes long, while a L2 cache normally consists of 256 bytes
the kernel developers used a trick to include threadability within Linux, by introducing a new kernel function called \texttt{clone()}. This clone function can be compared to the \texttt{fork()} function as it does nearly the same instead of creating threads. But also not really, because of the trick the kernel developers used, threads within the Linux operating system are just normal processes that share their address space and in addition to that also can use the same signal handlers, file descriptor tables etc. This is not possible by forking a process, because normally \texttt{fork()} doesn’t allow to use the same descriptor tables or signal handlers. In addition to that hybrid solution of \texttt{clone()}, an ordinary programmer should normally never use that function directly to introduce threadability to a program, because it is a Linux only solution to integrate threads and it will obviously better to use official standardized functions like the one out of the \textit{POSIX Threads Library} to assure portability to other operating systems.

So if you create threads from within Linux you will notice that this new "threads" will appear as normal processes within the global process table and therefore have also different process identifiers. In addition to that it is possible to set different priorities to every thread. The same transparency applies to the Linux scheduler itself, because threads will be treated like normal processes and therefore the scheduler will try to distribute this threads on different processors like they were normal processes. Again this distribution can not be manipulated by the ordinary programmer and he can just rely on the rule that for best performance, \textit{the number of processes/threads in a parallel program should be equal to the expected number of a program’s processes/threads that simultaneously can be running on different processors}. For example, if a four-processor SMP typically has one process actively running for some other purpose (e.g. a Mail server), then a parallel program should use only three processes/threads. You can get a rough idea of how many other processes are active on a system by looking at the "load average" quoted by the \texttt{uptime} command.

\subsection*{2.7 Linux Threads}

Now that we have discussed the implementation of the \texttt{clone()} function and why it was introduced in Kernel 2.0 of Linux, we want to spent some time looking at the actual implementation of the \textit{POSIX Threads Library} available to Linux. As mentioned earlier, it is really not adviceable to use the \texttt{clone()} function directly, because it will make it hard later to port this application to another platform and another point why one shouldn’t use that function is, that a thread library like the \textit{PThreads Library} comes already with reentrant functions for \textit{mutual exclusions}, so one haven’t to mix up \texttt{clone()} with other mutual exclusion mechanisms like the \textit{System V IPC} semaphore mechanisms, which we will discuss later.

By looking at the \textit{PThreads Library} implementation on Linux\footnote{which is simply called ”Linux Threads”} it is easy to recognize that the \textit{Linux Threads} library itself provides a very good and efficient way to develop large-scale threaded programs. Not only the portability and
that many tutorial material is available for that POSIX standard is the main benefit here. Moreover this pthread library provides all mechanisms needed to completely implement threadability and parallel processing under every unix environment.

2.7.1 Program Structure

By following the POSIX standard for thread programming such a program will have to use standardized functions for introducing parallelism. As discussed before, the pthread library distributed with Linux provides mutual exclusion mechanisms. This mechanisms give the programmer the ability to introduce atomicity within his parallel program. By surrounding critical sections with pthread_mutex_lock() and pthread_mutex_unlock() calls, he or she can make sure that this section will only be executed at one processor at a time.

In addition to that the pthread_create() and pthread_join() calls are the main functions for creating and destroying threads. This four functions of the pthreads library, together with calling the gcc compiler with the -D_REENTRANT define option, gives the programmer the ability to quickly introduce threads within his program.

The following source code of a numerical approximation of π (PI) should give an impression how easy it is to introduce parallelism into a program by using the powerful pthreads library:

```c
#include <stdio.h>
#include <stdlib.h>
#include "pthread.h"

volatile double pi = 0.0; // shared Approximation to pi
pthread_mutex_t pi_lock; // mutual exclusion variable
volatile double intervals; // number of intervals

void *process(void *arg)
{
    register double width, localsum;
    register int i;
    register int iproc = (*((char *) arg) - '0');

    // width of the approximation
    width = 1.0 / intervals;

    // local computation
    localsum = 0;
    for(i = iproc; i < intervals; i += 2)
```
double x = (i + 0.5) * width;
localsum += 4.0 / (1.0 + x * x);
localsum *= width;

// Lock & Unlock before accessing PI
pthread_mutex_lock(&pi_lock);
pi += localsum;
pthread_mutex_unlock(&pi_lock);

return(NULL);

int main(int argc, char **argv)
{
    pthread_t thread0, thread1;
    void * retval;

    // number of wished intervals is parsed by command line
    intervals = atoi(argv[1]);

    // init the mutual exclusion variable
    pthread_mutex_init(&pi_lock, NULL);

    // create two threads for the approximation now
    if (pthread_create(&thread0, NULL, process, "0") ||
        pthread_create(&thread1, NULL, process, "1"))
    {
        fprintf(stderr, "%s: cannot make thread\n", argv[0]);
        exit(1);
    }

    // wait & rejoin the two threads
    if (pthread_join(thread0, &retval) ||
        pthread_join(thread1, &retval))
    {
        fprintf(stderr, "%s: thread join failed\n", argv[0]);
        exit(1);
    }

    // print the result
    printf("Estimation of pi is \%f\n", pi);
    exit(0);
}
2.8 System V Shared Memory

Already mentioned before, beside the easy and elegant way of using the POSIX threads library approach, a developer has also the possibility to use mechanisms of the System V IPC\textsuperscript{13} concept. While comparing the System V IPC and the Linux Threads package, already discussed above, one will recognize that both systems provides mechanisms for mutual exclusion. However, by introducing parallelism it is not adviceable to use the semaphore or message queue system out of the System V IPC. The reason for that is that it was mainly designed for uniprocessor systems and therefore isn’t that fast like pthreads. Especially the semaphore possibilities within this IPC system shouldn’t used at all within a program using a threads library package because the functions don’t react really atomical over multiple processors.

The reason why we still introduced that section is, that at least the shared memory concept of the System V IPC could be interesting for parallel programming. The advantages of this is that once a shared memory segment is created and attached to a thread, any change to a certain value by one processor will automatically visible to all other processors. But the most important advantage is, that each communication operation will occur without the overhead of a system call.

2.8.1 Program Structure

Like the program structure of programs using the pthreads library, also the System V Shared Memory mechanism comes with some set of support functions. Mainly the \texttt{shmget()}, \texttt{shmctl()} and \texttt{shmat()} functions are responsible for introducing shared memory support.

By calling \texttt{shmget()}, one can reserverate a special memory area that should be shared by several processes or threads. Afterwards every process that wants to use this particular memory segment, have to use the \texttt{shmat()} function to attach this area to the process table itself. After attaching and using that particular memory segment it has to be detached by calling the \texttt{shmdt()} function and then signaled to be removed with a \texttt{shmctl()} call.

But still after attaching a shared memory segment to a process the programmer has to take care that only one thread at a time accesses this segment by using the mutual exclusion mechanisms discussed in the Linux Threads library package. So a transformed version of the example source code within the pthreads section after introducing shared memory support with System V Shared Memory support functions would look like:

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include "pthread.h"
```

\textsuperscript{13}Inter-Process Communication
volatile struct shared // the shared memory segment
{
    double pi;
    pthread_mutex_t pi_lock;
}*shared;

volatile double intervals; // interval variable

void *process(void *arg)
{
    register double width, localsum;
    register int i;
    register int iproc = (*((char *) arg) - '0');

    // width of the approximation
    width = 1.0 / intervals;

    // local computation
    localsum = 0;
    for (i=iproc; i<intervals; i+=2)
    {
        register double x = (i + 0.5) * width;
        localsum += 4.0 / (1.0 + x * x);
    }
    localsum *= width;

    // Lock & Unlock before accessing PI
    pthread_mutex_lock(&(shared->pi_lock));
    shared->pi += localsum;
    pthread_mutex_unlock(&(shared->pi_lock));

    return(NULL);
}

int main(int argc, char **argv)
{
    pthread_t thread0, thread1;
    void * retval;
    register int shmid;

    // Allocate System V shared memory segment
    shmid = shmget( IPC_PRIVATE,
                  sizeof(struct shared),
                  (IPC_CREAT | 0600)
    );
    shared = ((volatile struct shared *)shmat(shmid, 0, 0));
// immediately schedule it for removal.
shmctl(shmid, IPC_RMID, 0);

// initialize PI within the shared memory segment
shared->pi = 0.0;

// number of wished intervals is parsed by command line
intervals = atoi(argv[1]);

// init the mutual exclusion variable
pthread_mutex_init(&(shared->pi_lock), NULL);

// create two threads for the approximation now
if (pthread_create(&thread0, NULL, process, "0") ||
    pthread_create(&thread1, NULL, process, "1"))
{
    fprintf(stderr, "%s: cannot make thread\n", argv[0]);
    exit(1);
}

// wait & rejoin the two threads
if (pthread_join(thread0, &retval) ||
    pthread_join(thread1, &retval))
{
    fprintf(stderr, "%s: thread join failed\n", argv[0]);
    exit(1);
}

// print the result
printf("Estimation of pi is %f\n", shared->pi);

exit(0);

2.9 Memory Map Calls

As file I/O operations can be very time consuming there are of course buffered
I/O functions like getchar() or fwrite(). The only problem with that system
calls is, that as soon as multiple processors are introduced they will be broken
if two processes on different processors are trying to write to the same file. Also
the overhead for that functions will increase significant.

BSD Unix systems and also Linux therefore fixed that problem by providing
some special memory mapping calls that will allow a portion of a file to be
mapped into user memory. The Linux implementation of that mmap()\textsuperscript{14} call

\textsuperscript{14}please see the manpages of the \texttt{mmap()} call function for detailed information on argument
space and usability.
is mainly the same like generating and attaching a shared memory segment with the System V Shared Memory concept described above. Just to mention, nowadays the \texttt{mmap()} was obsoleted by the more modern and supported System V Shared Memory calls.

So there is really no big difference in using \texttt{mmap()} or \texttt{shmget()} to introduced shared memory in a parallel programs but one may still prefer that call because of portability reasons to BSD.

\section*{2.10 Summary}

After reading over this SMP section, one should get a very good impression about the very good possibilities available to the Linux operating system for parallel programming. The point that Linux uses well-documented standardised system calls like the one in the POSIX Threads Library or the System V IPC makes it a very powerfull tool to implement highly scalable parallelism. Also the Linux scheduler, which is responsible to distribute over multiple processors was reworked with latest 2.4 versions of the linux kernel. So by latest versions, Linux really haven’t to fear competing operating systems\textsuperscript{15} where SMP is said to be more stable.

\textsuperscript{15}like SUN Solaris or Windows
Chapter 3

Linux Clusters

As this report should only cover the parallel programming support available within the Linux operating system, we just want to give a short overview to other alternative approaches to provide parallelism. One of this alternatives, also available to Linux, is the ability to build Cluster systems. Clusters are an amount of servers that are connected via a specified network. This network can be any media, but nowadays speaking of Clusters implies to have servers connected via a LAN or Internet. Beside that, Clusters can also be built via other bus systems like SCSI or FireWire. But what all Cluster systems unifies is, that they run special software that will allow them to distribute computation onto several connected servers.
To really discuss on the full possibilities and software available for building Cluster system would really fill another own report and therefore we just want to mention the names of popular systems available to Linux:

- **PVM - Parallel Virtual Machine [6]**
a freely-available, portable, message-passing library generally implemented on top of sockets. It is clearly established as the de-facto standard for message-passing cluster parallel computing.

- **MPI - Message Passing Interface [7]**
equal to PVM, but established as the new official standard for a message parsing library for clusters.

- **AFAPI - Aggregate Function API**
a very hardware-specific low-level support library for PAPERS (Purdue’s Adapter for Parallel Execution and Rapid Synchronization)

- **Beowulf [8]**
centers on production of software for using off-the-shelf clustered workstations based on commodity PC-class hardware, a high-bandwidth cluster-internal network, and the Linux operating system
Chapter 4

Conclusion

That there are many possibilities for parallel programming available to the Linux operating system should be clear now. Taking the briefly discussed Cluster systems apart, Linux still provides a wide range of support functions for parallelism. The main advantage here is clearly the very successful implementation of the *POSIX threads library* included in the latest *glibc* library as discussed in section 2.7 of this report.

Not only the fact, that it is relatively simple to implement parallel programs with the support functions provided by the *pthreads* library makes it a strong concept. Moreover the fact that together with the SMP abilities of the Linux kernel and scheduler itself, it allows to write high scalable application where the scheduler of Linux successfully distributes each thread onto separate processors. In addition to that, the discussed *System V Shared Memory* implementation allow the programmer to assure that all threads can access a certain memory segment at no time. All this possibilities lets easily forget the fact that Linux itself doesn’t support real kernel level threads and uses a ”trick” to provide threadability.

Also the standardized way of the implementation of the parallel support makes Linux very powerful in parallel computing, because it is very easy to port existing applications from one operating system to another. In contrast to that, the SMP support within Windows uses system calls that are different from the standard calls other operating systems are using for creating threads and therefore portability isn’t easy under Windows, but that isn’t really a new fact.

So putting it together, Linux really provides all types of developers with a highly advanced and modern framework for parallel programming, so it mainly should depend on your underlying hardware how fast you will get out your results of your program.
Bibliography


[3] Reevaluating Amdahl’s Law
   http://joda.cis.temple.edu/∼shi/docs/amdahl/amdahl.html

[4] Intel Multi Processor Specification
   http://www.intel.com/design/pro/datasheets/242016.htm

[5] Programming POSIX Threads
   http://www.humanfactor.com/pthreads/

[6] Parallel Virtual Machine
   http://www.epm.ornl.gov/pvm/pvm_home.html

   http://www.mcs.anl.gov/mpi/

[8] The Beowulf project
   http://www.beowulf.org/