Wool -- A C library for OSE PowerPC Multi-Core system

Yang Wang

Master's thesis at: System on-chip design
Royal Institute of Technology

Conducted at: ENEA

Supervisors: Qiang Chen,
ICT, Royal Institute of Technology
Detlef Scholle, Barbro Claesson
ENEA

Examiner: Lirong Zheng
ICT, Royal Institute of Technology

Stockholm, May, 2012

Abstract

With the requirement of high property processor rapidly increasing, ordinary single-core processors can hardly deal with the task it faces due to the limitation of frequency, power consumption, heat dissipation and etc. Under this circumstance multi-core processor technology turns out to be a reasonable solution for this problem. While multi-core processors will improve the performance like clock frequency, parallel threads and etc, a certain research illustrate that the improvement of the performance is not linear with the number of cores. This is because the overhead of scheduler, unbalanced load and the architecture of the multi-core processors. When operating system gives good model like SMP, AMP and industry like X86 and PowerPC design kind of architecture for multi-core, author's direction naturally focus on a light overhead scheduler which is a C language scheduler "Wool" researched in this paper originally designed by Mr. Karl-Filip Faxen in SICS. Wool is a library providing lightweight tasks on top of pthreads. [4]

In order to understand the principal and structure of "Wool", author does several pre-study about multi-core schedule and decides the platform and operating system to design some tests. The first part of this master thesis report give an overview of modern technology of multi-core architecture, parallel programming, task parallelism strategy and etc. The second part specified the "Wool" scheduler. PowerPC e500 core and Enea OSE operating system for the test design and use case. The last part conclude the results of the tests and point out the probably direction for future research.

Table of Contents

Chapter 1

Introduction

1.1Background

As the Moore's Law stated, the performance of computer doubled every 18 months which is concluded in David House's reversion [5] in the past decade years until the single core come to its bottleneck. Then naturally people turn to multi-core to find a solution and inspiringly the multi-core processor run at a lower frequency and perform better than single core because "two heads are better than one."[6] At the same time new problem about reducing overhead of concurrent program and keeping load balance for parallelism task begin to confuse engineer when parallel software start to widely be used on multi-core architecture.

Wool is a library providing lightweight tasks on top of pthreads [4] in other words it is a C task scheduler for concurrent program especially on multi-core architecture. The corporation between Karl-Filip Faxen, the author developed Wool and Enea give birth to my master thesis work which is running "Wool" on OSE operating system under PowerPC to derive characteristics of "Wool".

1.2 Problem statement

This master thesis concentrates on the "Wool" task parallelism strategy, PowerPC assembly instruction set and OSE operating system. One problem is to investigate how "Wool" reduce overhead and execute an efficient balance load. The other one is that "Wool" only works on Linux under CISC (Complex Instruction Set Computing) architecture like X86. If author want to launch it on OSE operating system under RISC (Reduced Instruction Set Computing) architecture—PowerPC, the primary thing to do is getting familiar with the OSE architecture and having an overview of the difference of assembly code between CISC and RISC.

Questions that should be answered:

Q1 What kind of task parallelism strategy does "Wool" use?

Q2 What is the difference in program under CISR and RISC architecture?

Q3 Can "Wool" be modified to run under PowerPC architecture?

1.3 Method

The whole thesis work is 20 weeks in which pre-study, test, report and presentation should be involved. First 10 week theoretical papers, manual and instruction will be primary to study which aim to solve the questions mentioned above. After then deeper investigation on "Wool" structure and Macro inside could discover more detail and secret of it. When it comes to the second half, more practical work with PowerPC assembly code, research on OSE operating system and the chosen platform should be done. In the end, come out with design and test to make some conclusion of the characteristics of "Wool". A new modified "Wool" then will be born for the combination of two features—OSE operating system and PowerPC architecture.

1.4 Delimitations

There will be a lot of theoretical material to read and also a bunch of practical problem to solve. Since the whole thesis work is 20 weeks, author can only use QorIQ P4080 platform with OSE operation system on it as research target.

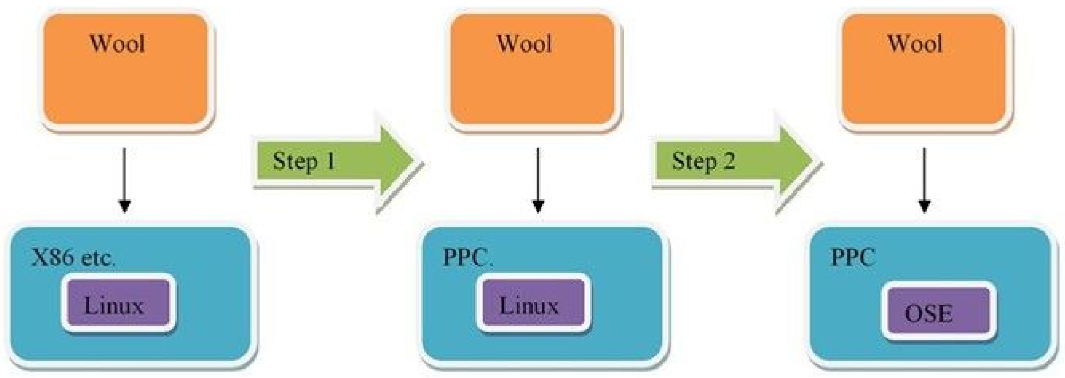
1.5 Purpose

The purpose for this master thesis is to transplant "Wool" onto OSE operating system in PowerPC architecture. Furthermore, investigate the property of "Wool" running in that environment and give some conclusions.

1.6 Goals

G1. Make "Wool" fit for PowerPC instruction set.

G2 Modify "Wool" to run on OSE operating system



Chapter 2

Multi-core processors

2.1 Introduction

The trend of increasing a processor's speed to get a boost in performance is a way of the past. [6] The heat dissipation problem, clock frequency limit and also the general trend of parallelism software program force engineer developing new solution. As multi-core processor is a brand new architecture to increase the performance without the limited clock frequency, it begins to be widely used in industry. A multi-core processor is an integrated device with more than one independent computing units and all of these units are able to execute instructions.

2.2 Property of multi-core

The advantage of multi-core processors is obvious when compare the computing speed, clock frequency and data rate. The benefit could be divided into two parts, immediate benefit and long term benefit. Immediate benefit is especially significant in areas like processor-intensive tasks, concurrent program with multi-threads, and diversity of structure for industry. A new direction for next generation software and hardware design is the long term benefit. Like every micro-processor, multi-core processor has its own advantages and drawback. The disadvantages include the complexity of the architecture, overhead in software program, hardly handling the communication between cores and load balance.

But as L. Peng referred in his paper, due to advances in the circuit technology and performance limitation in wide-issue, super-speculative processors, Chip-Multiprocessors (CMP) or multi-core technology has become the mainstream in CPU design. [7]

2.3Architecture

Multi-core processor architecture include caches/shared caches, memory, interconnection and cores. In this thesis multi-core processor can be considered a processor with multi-cores inside. Figure 1 identifies a normal structure of a multi-core processor. In this figure, the cores work with two levels of cache, which L1 cache is private while L2 cache is shared. A system bus connects processor 0, processor 1 and a system memory. The interconnection transfers data and instruction between processors/caches and system memory. These cores here can have either the same position or a master-slave relationship.

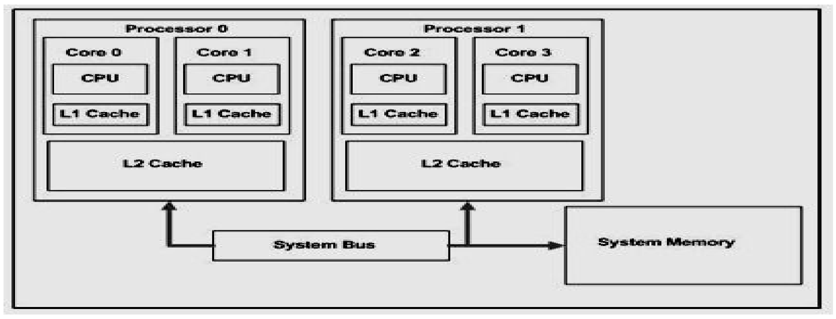


Figure 1 A multi-core multi-processor system architecture [8]

2.3.1 Cache

Cache is designed to reduce the memory access time. Its access time is much faster than memory and it could save data or instruction which will be used frequently or iterative, so it is located between memory and core as kind of buffer. Usually two or three levels of cache are used, i.e. L1 cache, L2 cache and L3 cache and L1 cache is private for one core while L2, L3 cache are normally public. Cache is made of SRAM which is expensive and especially L1 cache is smallest, faster and most expensive. Because of the value of size of cache, L2 and L3 caches are always shared by cores. The benefit is obvious which is to reduce cache under-utilization, cache coherency complexity, and false sharing penalty, data storage redundancy at the L2 and L3 cache level and front-side bus traffic. [8]

2.3.2 Memory

A memory is connected to processors by system bus (interconnection) and much larger than caches. A memory normally store big files which are currently executing but too big to store in cache. Compared to hard disk, the access time to memory is still less and when it comes to multi-core processor the memory is necessarily required. Because a global memory is a good and easy solution for communication between cores by writing and reading the memory. It is useful but also leaves problem to interconnection when there are approximately 32 cores. [6]

2.3.3 Interconnection networks

The interconnection work plays a central role in determining the overall performance of a multi-computer system. If the network cannot provide adequate performance, for a particular application, nodes will frequently be forced to wait for data to arrive. [9] This will affect the whole system vitally because communication between cores is the primary condition to remain the system smooth and stable. For a given number of cores, the “best” interconnection architecture in a given chip multiprocessing environment depends on a myriad of factors, including performance objectives, power/area budget, bandwidth requirements, technology, and even the system of software.[10]

2.3.4 Homogeneous and heterogeneous processors

Homogeneous cores are all exactly the same: equivalent frequencies, cache sizes, functions, whereas each core in heterogeneous environment could have a specific function and run its own specialized instruction set. [6]

There always is an argument about which is better and it still remain suspended. Homogeneous cores are easy to use because the structure and the instruction set are all the same. The work load can be allocated at any of the core so the scheduler, load balance and parallelism task are convenient to achieve. But when it comes to specialized task this environment may be not the most efficient one. A special tailored task runs on a specific core with the proper structure and instruction set will performance much better. At the same, the complexity is the vital technical barrier for common purpose tasks. So both these two environment have their own special ability for variety tasks, like two sides of a coin.

2.4 Summary

Multi-core processor opens a new window to solve the frequency limit, heat dissipation and power consumption problems of traditional single-core processor. On the other hand, because of using shared cache, memory, the requirement of interconnection bandwidth and complexity increases rapidly. To control the growth of the complexity of the structure, a homogeneous core environment shares the same hardware structure and instruction set is widely used in most of recent systems. However, a tradeoff always exists between complexity and efficiency. At some special cases, increasing the complexity by utilizing heterogeneous core environment will enhance the capability of the system significantly.

Chapter 3

Parallel programming

3.1 Introduction

Parallel programming is normally based on the structure of multi-core processors as it breaking a program into several task/threads which can execute simultaneously. Running several parts of the program parallel will shorten the run-time and speed up the system response. Parallel programming makes good use of the multi-core hardware architecture and keeps every core busy. This technology has been widely used in supporting scientific research and even in our daily life like weather forecast. But as the particularity of parallel programming author mentioned above, only a problem satisfies some requirement is able to be parallel programming.

a) Be able to be broken apart into discrete pieces of work that can be solved simultaneously.

b) Be able to execute multiple program instruction at any moment in time.

c) Be able to be solved in less time with multiple compute resources than with a single compute resource.[11]

It seems like that the more cores a CPU has the faster the program will be executed. Unfortunately the result is not ideal and actually with number of cores the increasing the performance draws an opposite “U” curve. This is because an overhead is imposed which includes software overhead, breaking program down, synchronization, communication between tasks and assembling program back. Now we can see parallel programming is a tricky problem and tradeoff often occurs between different models and solutions.

3.2 Parallel programming models

Parallel programming models exist as an abstraction above hardware and memory architecture. [11] Different models have their own advantage and of course drawbacks. Here present several common models and give a brief introduction.

3.2.1 Shared Memory (without threads)

Shared memory is a certain memory space which is able to be accessed by different software in order to communicate with each other and save the memory space. A same storage unit like global variable can be visited by a collection of individual tasks. By writing and reading the unit, data and communication instruction exchange between tasks. Indeed, it presents a simple communication method, since ownership of this data does not have a notion. However there are a few of drawbacks discussed below.

3.2.1.1 Racing problem

Race condition happens when several processes or threads try to access a shared state. Because no notion exists, any of task win the race can write or read the data. If one task changes a variable unexpectedly, it will result in another task fetching wrong data without noticing. Avoiding this problem is essentially important in the situation of data dependency, which is difficult to debug.

3.2.1.2 Deadlock

Usually locks/semaphores are imposed to solve the racing problem. Locks/semaphores are kind of resource that only the task has the resource is able to deal with the data and other tasks are not permit to access until the resource is released or recycled. Well-constructed locks/semaphores dominate if the program works efficient, otherwise a deadlock will occur.

When two or more threads or processes endlessly waiting for each others to release certain shared resource, this scenario is a deadlock.

In order to avoid the deadlock happening, four conditions should be treated seriously.

a) Mutual exclusion: Each resource is either currently allocated to exactly one processors or it is available.

b) Hold and wait: Processes currently holding resources, it cannot be taken away by another process or the kernel.

c) No preemption: Once a process holds a resource, it cannot be taken away by another process or the kernel.

d) Circular wait: Each process is waiting to obtain a resource which is held by another process.[12]

3.2.2 Threads

In a threads model, a process is divided into several threads. These threads share the same resource of the process possess of. The communication between threads is sharing a global memory and they are able to call, synchronize and terminate other threads. Once the threads finish, they release the clock slice and give the resource back to the process. POSIX defines a standard for C language threads and APIs are supported in the pthread.h library.

3.2.3 Distributed memory/ Message passing

Message passing is a concept of communication between two processes with only local memory. Normally a sending operation always is matched with a receiving operation by another process. This model fits multi-core hardware best since every core has its own memory like caches and they are connected by on-chip bus or network. At the same time, it requires that the programmer build an even clearer model about the interaction between tasks and the whole architecture of the problem to avoid race condition, dead lock and etc.

Point-to-point communication and collective communication are two basic communication models. P2P communication naturally the first idea comes in mind because the idea is simple and easy to handle. But if a collect of tasks send massage at the same time, a traffic problem will occur in the interconnection which bottlenecked the speed of the system. Sometimes a same massage is sent over and over and this causes unnecessary overlap and decrease the efficiency. Collective communication is based on a concept that a member collects massage first and sends them until some condition is satisfied. Three types of collective operations include synchronization, data movement and collective computation. Broadcast, scatter and all-to-all are three mainstream technology of data movement. [13]

3.3 Designing parallel

The efficiency and speed of the parallel programming accompany a collector of problems. Some problems only impact on the efficiency, and others may cause vital error which could hard be debugged.

3.3.1 Partitioning

Partitioning is breaking down a problem into smaller tasks which can be executed on several cores and this is the first step for parallel programming. Meanwhile, this partitioning work is the main overhead of start-up time. Domain decomposition and functional decomposition are two kind of partitioning methods depend on the specified problems. Domain decomposition concentrates on the tasks that have similar data structure for example array and cube. However, functional decomposition focuses on the instruction set and collect similar ones together to a core.

3.3.2 Communications

In the case of communications, data is shared between tasks and also instructions are sent.

3.3.2.1 Blocking and non-blocking

In non-blocking communication, a task continues to do its work as soon as it sends a message. This kind of communication sending a lot of small massages add the latency up and lead to heavy overhead. To reduce the latency and increase the efficiency of bandwidth, collecting the message and sending them once is a good solution, and this brings out the concept of blocking communication. In blocking communication, tasks need to "handshaking" before they continue to work and also the waiting time for "handshaking" cause another kind of overhead

3.3.3 Data dependencies

A dependency exists between program statements when the order of statement execution affects the results of the program and a data dependence results from multiple use of the same location in storage by different tasks.[11] Data dependencies need to be carefully treated in partitioning and it is the primary inhibitor in parallel programming.

3.3.4 Load balancing

The high performance of multi-core processor is because every core takes a part of a problem. In the other word, if some cores are just standing by, the multi-core will become meaningless. Load balancing tries to guarantee that every core is busy. If the problem is simple for instance array/matrix and loop, allocating same size of work to each core is able to work efficiently. Otherwise, some complex or unpredictable tasks need use dynamic scheduling. Here introduce two familiar strategies.

Task-pool: No task is assigned to cores at beginning, and all cores go and fetch a task from the task-pool. If one core finish its own work, is will go and fetch another work until the task-pool empty.

Task-stealing: Each core owns a stack of tasks at first. If a certain core has done with its stack, it will steal tasks from other cores by random or other algorithm.

3.3.5 Granularity

Granularity is the ratio of computation/communication and it can be divided into fine-grain parallelism & coarse-grain.

Fine-grain	**Coarse-grain**
Low ratio of computation/communication	High ratio of computation/communication
Relatively easy for load balancing	Relatively hard for load balancing
High communication overhead	Low communication overhead
Low potential to enhance performance	High potential to enhance performance

This table shows a tradeoff between load balancing and communication overhead. A final decision should depend on the algorithm and the hardware environment. In other words, compare load balancing overhead and communication overhead for the system.

3.4 Summary

In this chapter, parallelism coding seems to be a difficult job. Choosing the most suitable model for specific hardware environment is the first step. Then in practical programming like partitioning, communication, load balancing and granularity, a decision of tradeoff need to be given which require a solid knowledge for a platform and instruction flow.

Chapter 4

CPU architecture

4.1 Introduction

CPU architecture is given to the same series of CPU products by manufacturer. Recently, Intel CPU and AMD CPU occupy the CPU market and they present different architecture. CPU architecture can be divided in two ways. One depends on the operation bits (32-bit & 64-bit) and the other depends on instruction set (CISC & RISC).

4.2 Operation bits architecture

32-bit architecture is the mainstream in CPU market. 64-bit architecture breaks some traditional IA32 architecture limit and owns a position in high performance computer. IA-32, x86-32, x86-64 all belong to x86, i.e. Intel 32-bit x86 architecture. IA-64 is a 64-bit architecture developed by Intel and Hp in order to fully enhance the performance of formal IA-32bit CPU. It abandons x86 architecture since it restricts the enhancement of CPU, but it is not able to solve the problem of compatibility for 32-bit application.

4.3 Instruction set architecture

Instruction set architecture is part of computer system structure and relates to program design. It includes the management for data type, instruction, registers, addressing mode, memory architecture, interrupt, exception handling and I/O. Instruction architecture works based on micro-architecture which exists in CPU. Without micro-architecture, instruction set architecture is nothing but opcode. Different micro-architecture can execute a same instruction for example Intel Pentium and AMD Ahtlon, because they use the same x86 instruction set. Depending on the difference of instruction set, instruction set architecture can be divided into CISC and RISC.

4.3.1 CISC

In a long term, the enhancement of computation performance relies on increasing the complexity of hardware. Under this background, hardware engineers continuously add complex functional instruction and flexible addressing mode. In order to achieve complicated operation, micro-processors provide programmer not only register and machine instruction but also micro programs which are stored in ROM.

Micro-processors execute a serious of elementary instruction after analyzing every instruction and this kind of design is called Complex Instruction Set Computer-CISC. In general, CISC models include at least 300 instructions and some have even more than 500 instructions.

4.3.2 RISC

A computer using CISC have the strong capability to deal with high level language which is benefit for increasing the performance of the computer. However, somebody refuse to go with the stream when the design of computer develops along this way. In 1975, IBM Jhomasl Wason research center starts to organize the investigation of the rationality of CISC, because at that moment, scientists notice that more and more complicated instruction system is difficult to achieve and can reduce the performance of the system. In 1979, one research carried out by a group of scientists from UC Berkeley lead by Professor Paterson shows that CISC has a collection of drawbacks.

Firstly, the usage of varieties of instructions differs massively: 80% instructions in a typical program are only 20% in the processor instruction system. Actually, the most frequently used instructions are simple instructions like fetching, writing, adding and etc. In other words, the design of CISC instruction system people concentrate for long term is indeed an instruction system processor which is rarely used in practical work. Meanwhile, complex instruction system will bring the complexity of structure. This increases the cost of design and the opportunity of mistake as well.

Secondly, in despite of the high level technology of VLSI, it is difficult to package all hardware of CISC on one chip which affects the development of single-chip micro-computer. In CISC, many complicated instructions require special complicate operation which are kind of copy of some high level language, so the compatibility is not satisfied. Aiming at these problem, Professor Paterson came up with the concept of reduced instruction which is a instruction system should include small amount of high frequently used instructions and provide a few necessary instructions to support operating system and high level language. Computers developed based on this principle are called Reduced Instruction Set Computer-RISC.

4.4 Familiar CPU architecture

In general, several CPU architectures are the mainstream in industry as kind of development model and thousands of application based on them.

	X86	**SPARC**	**PowerPC**
Designer	Inter, AMD	Sun Microsystems	AIM
Bits	16-bit, 32-bit, and/or 64-bit	64-bit (32→64)	32-bit/64-bit (32→64)

Design	CISC	RISC	RISC
Type	Register-Memory	Register-Register	Load-Store
Encoding	Variable	Fixed	Fixed/Variable
Branching	Status Register	Condition Code	Condition Code

4.5 Summary

Generally CPU architecture mainly means the operation bits and instruction set. 32-bit architecture is primary used right now and 64-bit architecture owns more capability to enhance performance. However the compatibility problem remains to be fixed.

The stable and fast instruction execution and low cost of manufacture are the advantages of RISC while most software supports CISC at present. There is also a compatibility demand. Three types of most common CPU architectures have different operation bits and instruction sets. The type of instruction sets matters the software program most and the compatibility between different X86 and PowerPC is hard to handle.

Chapter 5

Task Queue

5.1Introduction

In order to implement problems parallel, many of them need to breakdown into small executable units. However, the problems today have more and more amount of tasks and become more efficiently concurrency-dependent. These kinds of applications have the same characteristics which is dynamic task number and size. To handle these tasks, a powerful scheduler is required to assign tasks to every core and keep load balancing. Task queue is a well-known mechanism that is primarily designed to address the load imbalance problem. [1] Task Queue is defined as a mechanism to synchronously distribute a sequence of tasks among parallel threads of execution. [14]

5.2 Basic concept

5.2.1 Parallel pattern

Loop-Parallelism: A loop problem is able to breakdown into iteration of small tasks which can be executed parallel. This kind of task has the property of statically length.

Task-Parallelism: Split a normal problem into small executable units. It widens the concept of loop-parallelism in which the task length could be variable and a data dependency exists between tasks.

Memory-Parallelism: Based on the idea of assign tasks shared same data to one processor's memory, this can increase the locality of tasks and reduce the communication overhead.

These three parallel patterns based on different models and come out with corresponding strategy to achieve low overhead and more scalability.

5.2.2 Centralized and distributed task queue

Centralized and distributed task queues are two methods to enqueue and dequeue tasks.

Centralized task queue is described as one processor own a global task queue and others processor call for tasks from it. This scheduling operation of assign tasks is handled by operating system. As one can imagine, it is a simple way to assign tasks without considering much about the task dependency because this is the only global queue the system have. Meanwhile, the interaction between processors is simple too since every processor only communicates with the master processor. However, the negative influence is obvious at the same time. A bottleneck will form when the communication is too busy. To conclude the usability of centralized task queue, three factors should be considered: task size, communication latency and number of processors.[14] A big task size can hide the overhead of communication while small number of processors have less possibility to form a bottleneck.

Distributed task queue present each processor its own queue and the tasks have been assigned at first. A well-performed scalability is the most significant advantage, i.e. it is more suitable for large amount of processors. At the same time, the communication overhead is relatively huge when some processor's queue is empty and it asks for new tasks. Another problem is the difficulty to handle the task dependency, so sometimes the system need to synchronize between tasks which bring waiting time and communication overhead too. Here illustrates the importance of locality mentioned in memory parallelism because it guarantees less synchronization and deals with task dependency locally. To distinguish the dependency of tasks, an addition priority marked on tasks is a solution. Meanwhile it brings more software overhead and increase the complexity of scheduling.

Hierarchical task queue is designed by combining both the concept of centralized task queue and distributed task queue. Each processor owns a relatively small local queue and the whole system owns a global queue. First enqueue tasks into each local queue and then enqueue the rest tasks into the global queue. When the local queue is empty, the processor takes tasks from the global queue. If the amount of available parallelism is limited or the amount of local memory is small, hierarchical task queue can sometimes perform better. [1]

5.2.3 Eager and lazy scheduling

Eager scheduling and lazy scheduling are two completely opposite scheduling methods. Eager scheduling can be described as the operating system keeps assigning tasks to processors while observing the status of each processor. Apparently, eager scheduling performance well in load balancing since no processor will be idle or starvation. On the other hand, this also impose overhead of checking every processor. In contrast, lazy scheduling will not give tasks to any processor until someone is idle and ask for tasks. Without checking the status all the time, low overhead is imposed by lazy scheduling. But the load balancing is not efficient compared to eager scheduling. In extreme situation, due to the variety of task size and big number of processors, some processor will ask for tasks and waste lot of time waiting.

5.3 Software and hardware task queue implementation

Software task queue implementation presents both portability and a high overhead of task management. The impact of the overhead will be significant unless each task is big enough to hide the latency. If a specific hardware has a task queue structure, the implementation will be accelerated massively. Also, the support software is able to be simplified which will reduce the overhead of software queue management. However, a specified program needs to be produced for a different hardware platform

5.4 Task stealing

Task stealing is a strategy for distributed task queue. In this model, the underlying work is to guarantee the locality of private queue, because the most significant overhead of this strategy is communication between processors.

Processors enqueue and dequeue a task from the head of the queue, as known FILO (First In Last Out). If a processor has finished its private queue, it will try to steal a task from other processor at the tail place. Generally the stealing target is random.

The advantages are described below:

a) It has been shown to be provably efficient both in terms of execution time and space usage.[1]

b) By scheduling the child tasks on the same processor as the parent task we get better cache locality since it supports significant data sharing between parent and child tasks.[2]

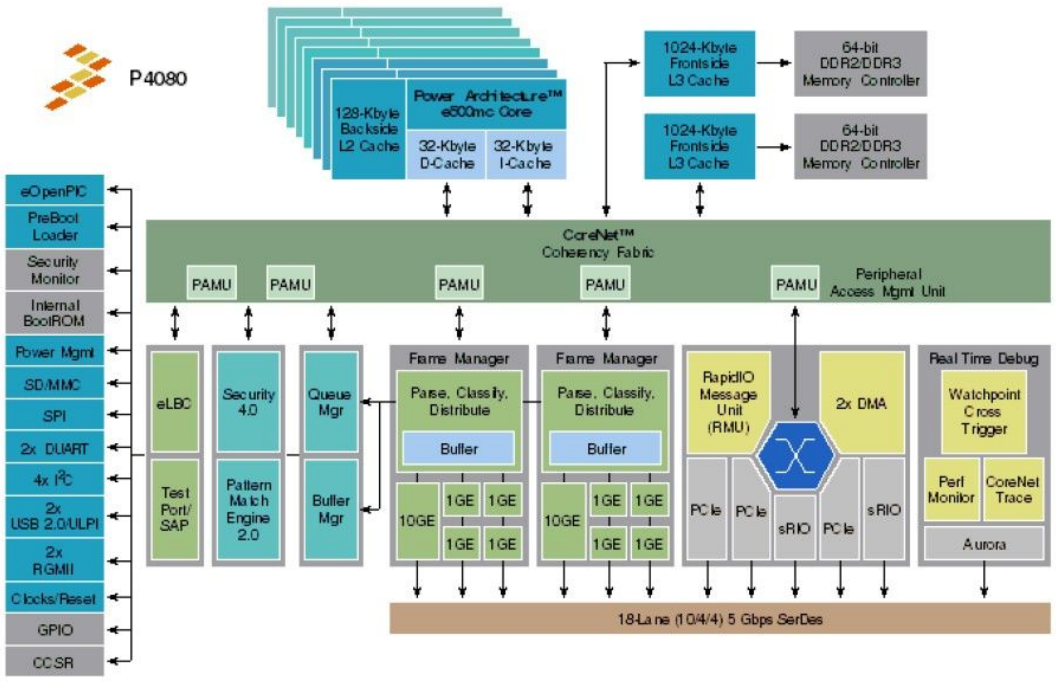
5.5 Summary

Task queue presents a good solution for parallel task management. According to the task size, communication latency and number of processors, a tradeoff requires to be balanced between centralized and distributed task queue. Because of the increasing processors number on platform design, distributed task queue will be the trend promisingly. Task stealing is a strategy based on distributed task queue in which locality is the most importance factor. By using a stack, the executing order and stealing order are well organized.

Chapter 6

QorIQ P4080 platform

The P4080 QorIQ communications processor combines eight Power Architecture processor cores with high-performance Data Path Acceleration Architecture (DPAA), CoreNet fabric infrastructure, and network and peripheral bus interfaces required for networking, telecom/datacom, wireless infrastructure, and mil/aerospace applications.[3] It is a powerful environment and can process in routers, switches, base station controllers, and general-purpose embedded computing systems.[3] The performance is really satisfied compared to multiple discrete devices and the improvement of board design is significant.



P4080 block diagram

6.1 Queue manager

P4080 platform supports hardware queue manager which can improve the performance compared to common software manager and also increase the complexity for porting. The queue manager (QMan) is the main component in the DPAA that allows for simplified sharing of network interfaces and hardware accelerators by multiple CPU cores. [2] It also provides a simple and consistent message and data passing mechanism for dividing processing tasks among multiple CPU cores. [2] Considering OSE signal passing communication pattern, this kind of hardware architecture is able to improve the performance of the operating system.

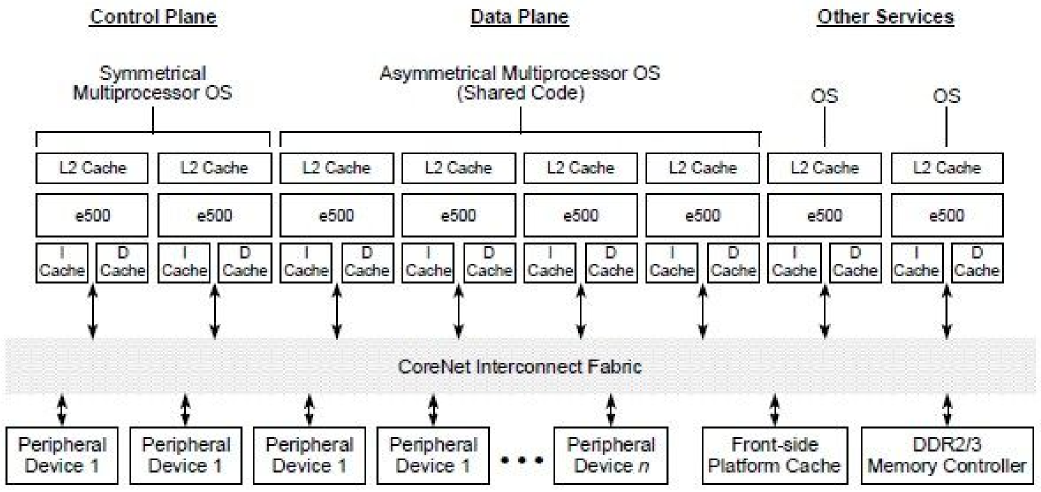
6.2 Resource partitioning

Porting discrete CPUs into a single system-on-chip raises both unexpected problems and newly advantages. A multi-core system might work unstable if no effectively resource partitioning and sharing methods are used. However, P4080 supports a new level of hardware partitioning, in order to ensure the software access the resource they are authorized to access. This method is relatively easier in P4080 than other environments, due to P4080 consists eight e500mc cores which build up a SMP environment. Otherwise, when it comes to be a SMP environment, the complexity of OS on different cores or even multi-OSes on different cores may increase rapidly. In the other words, the difficulty of distinguishing authorized software might be a bottleneck for OS protection.

6.3 e500mc core

Defined by Power ISA, e500mc cores are low-power embedded processors and work on 32-bit implementation which consist 32 32-bit general purpose registers. In every clock cycle, the superscalar processor assigns and operates two instructions. Furthermore, the core is capable to execute six instructions in parallel. Generally, the storage is in order and there also exists an optional out of order mode. The Load/Store unit (LSU) supports 32-bit integer and 64-bit floating-point operands. At the same time, it includes L1 cache which is independent on-chip, 32-kbyte, eight-way set associative, physically addressed and a unified 128-kbyte, eight-way set associative, physically addressed, backside L2 cache for instructions and data.

The design of the e500mc aims at multi-core integrated devices. The features ensure multi-core implementation and multi operating systems within a integrated devices by partitioning the cores.



6.4 Summary

The P4080 platform is an SMP system which includes eight e500mc cores and applies RISC instruction set. It also consists hardware Queue manager, resource partitioning and three level caches. Therefore, in this case, this platform is a typical multi-core platform on which task scheduling test will be suitable.

Chapter 7

Wool

7.1 Introduction

Wool provides lightweight tasks based on pthreads and it is also defined as a work stealing C library. The overhead of scheduling is reduced by the Wool macro and inline functions. Generally, the Wool library initials itself with one thread per physical processor. However, it is optional in the command line when the program starts. In Wool, a thread is called worker and each worker own a data structure which particularly in Wool is a task pool. The pool consists tasks that are ready to be executed and shared equally between workers.

7.2 Parallelism

The parallelism is achieved by spawning and sync tasks which are likely to be an asynchronous function call.

```
TASK_1( int, pfib, int, n )
{
    if( n < 2 ) {
        return n;
    } else {
        int m,k;
        SPAWN( pfib, n-1 );
        k = CALL( pfib, n-2 );
        m = SYNC( pfib );
        return m+k;
    }
}


TASK_2( int, main, int, argc, char **, argv )
{
    int n,m;
    if( argc < 2 ) {
        fprintf( stderr, "Usage: fib <Woolopt>... <arg>\n" ),
        exit( 2 );
    }
    n = atoi( argv[ 1 ] );
    m = CALL( pfib, n );
    printf( "%d\n", m );
    return 0;
}
```

example fib.c

In this fib example:

- TASK_1(int, pfib, int, n): Initialize fib (n)
- SPAWN(pfib, n-2): Do fib (n-1) in parallel
- m = SYNC(pfib): Ensure that SPAWNed task done. Otherwise do it.
- a = CALL(fib, n-1): Optimization of SPAWN+SYNC.

In this way, the task pool will grow in the shape of a tree with balanced sub-trees.

7.3 Work stealing in Wool

In Wool, work stealing is based on spawning and sync. Spawning take care of allocating space for tasks, initializing it and inserting it into the task pool.(efficient work stealing for fine grained parallelism) When tasks need to be sync, Wool will check the if the task is stolen. If not, Wool will finish this task and remove it from the task pool. Otherwise, the worker has to wait. What to do during this time is significant and may affect the performance seriously. Therefore, here raises two strategies.

7.3.1 Parking

In modern treading implementation scenario, workers might be more than cores. In this way, another worker will implement while a worker wait. However, problems occurs when several running in a single core. The scheduling overhead and cache misses increase while the locality decreases. Therefore, parking is a strategy to avoid this problem. The basic idea is in the most of time to keep the same number of workers and cores. In other words, the active workers are equal to cores and other worker are parking or sleeping. Once an active worker waiting, it will wake up a sleeping one. Sometimes, the number of worker may be more than cores. At this moment, as soon as a worker finish its task and notice that there are more active workers than cores, it will park itself.

7.3.2 Leapfrogging

In leapfrogging, when a worker A needs to wait for joining a task stolen from worker B, it could possibly steal task only from worker B. The task from pool of B must be done before the task from A. In this way, A is not occupied when B finishes.

7.4 Test

In order to investigate the effect of Wool, a test is running on a Linux single-core PC. The test is a compare of simple loop function with and without Wool.

The test supply loop2.c with Wool and myloop.c without Wool. However, the algorithm is unique and the structure is the same. Therefore, the result will present the difference obviously.

Loop2.c

```
#include "Wool.h"
#include <stdlib.h>
#include <stdio.h>
extern int loop(int);
LOOP_BODY_1( work, 100, int, i, int, n )
{
   loop( n );
}
TASK_2( int, main, int, argc, char **, argv )
{
   int grainsize = atoi( argv[1] );
   int p_iters = atoi( argv[2] );
   int s_iters = atoi( argv[3] );
   int i;
   for( i=0; i<s_iters; i++ ) {
      FOR( work, 0, p_iters, grainsize );
```

```
    }
    printf( "%d %d %d\n", grainsize, p_iters, s_iters );
    return 0;
}
```

Myloop.c

```
#include <stdlib.h>
#include <stdio.h>
extern int loop(int);
LOOP_BODY_1(int i, int n )
{
      int t;
      for(t=0;t<i; t++)
          {
              loop( n );
          }
}
main(int argc, char ** argv )
{
    int grainsize = atoi( argv[1] );
    int p_iters = atoi( argv[2] );
    int s_iters = atoi( argv[3] );
    int i;
    for( i=0; i<s_iters; i++ ) {
        LOOP_BODY_1( p_iters, grainsize );
    }
    printf( "%d %d %d\n", grainsize, p_iters, s_iters );
    return 0;
}
```

Myloop	Loop2	Myloop	Loop2	Myloop	Loop2
(10,10,1	(10,10,1	(100,100,10	(100,100,10	(1000,1000,100	(1000,1000,100

0)	0)	0)	0)	0)	0)
Real 0m0.002s	Real 0m0.002s	Real 0m0.004s	Real 0m0.003s	Real 0m2.424s	Real 0m0.548s
User 0m0.000s	User 0m0.000s	User 0m0.000s	User 0m0.000s	User 0m2.376s	User 0m0.536s
Sys 0m0.000s	Sys 0m0.000s	Sys 0m0.004s	Sys 0m0.000s	Sys 0m0.004s	Sys 0m0.000s

Loop function is able to create a balanced tree and in this result we can see the advantage of Wool increase with the tree growing. The reason is Wool arrange the threads in low overhead by increasing the locality and cache hits.

7.5 Summary

The Wool library is hardware dependent and the macros in Wool are also big amount. In spite of the complexity of Wool seemed like, it is based on pitheads. In other words, all the parallelism and work stealing come from the threads. Recently, the Wool supports several hardware, i.e. X86, SPARC and IA64. The Operating Systems are more flexible, besides Linux, OSes that support POSIX pthreads will work.

Chapter 8

Implementation

8.1 Wool for PowerPC

The target board P4080 is a product of PowerPC. Thus, since the Wool is hardware dependent, a modification of Wool requires to be done for the specific target board. Moreover, the hardware dependency is built on several macros defined in Wool.h and these macros are replaced by hardware based assembler code.

8.1.1 Memory and storage fence

Memory and storage fence ensure the complier adding an ordering constraint before and after the barrier. This method takes place because sometimes the compiler optimizes the code out-of-order. In other words, the order of code is reorganized to improve the performance. Although this optimization can not be noticed in one thread code, it may cause unpredictable problem in multi-thread code or multi-core system. Wool is a working stealing library and based on threads, therefore it is a serious problem in concurrency programming with Wool. Also, the memory fence is hardware dependent as well and always defined in architecture's memory ordering model. These codes are low level machine code which include synchronization primitives and lock-free data structures.

```
#if defined(__sparc__)
   #define SFENCE            asm volatile( "membar #StoreStore" )
   #define MFENCE             asm volatile( "membar #StoreLoad|#StoreStore" )

 #elif defined(__i386__)
   #define SFENCE            asm volatile( "sfence" )
   #define MFENCE             asm volatile( "mfence" )

#elif defined(__x86_64__)
   #define SFENCE            asm volatile( "sfence" )
   #define MFENCE             asm volatile( "mfence" )
```

In the code above, Wool defines memory and store fence for Sparc, i386 and x86. Similarly, assembler codes for P4080 require to be defined in the same place.

Pseudo code for powerpc:

```
defined(__powerpc__)
    #define SFENCE              /*      */
    #define MFENCE               asm volatile( "sync" )
                                // wait for the memory to be updated
```

8.1.2 Exchange and CAS

In Wool, the task stealing basically is an exchange operation between memory and task pool. Since it is also a low level operation, it is defined with assembler code as well. An advanced operation called CAS (compare and swap) is an improvement of an Exchange operation. The thieves using CAS have to compare the value it read with the previous value before an exchange operation.

```
defined(__i386__)
    #define EXCHANGE(R,M) asm volatile ( "xchg      %1, %0" : "+m" (M), "+r" (R) )


#elif defined(__x86_64__)
    #define EXCHANGE(R,M) asm volatile ( "xchg      %1, %0" : "+m" (M), "+r" (R) )
    #define CAS(R,M,V)    asm volatile ( "lock cmpxchg %2, %1" \
                                                  : "+a" (V), "+m"(M) : "r" (R) : "cc" )
```

In the code above, Wool defines exchange for i386 and exchange and CAS for x86. Similarly, assembler codes for P4080 require to be defined in the same place.

Pseudo code for powerpc:

```
defined(__powerpc__)
    #define EXCHANGE(R,M)
  asm volatile( "loop: lwarx r5,o,r3 //Load and reserve
                          stwcx r4,0,r3 //Store new value if still reserved
                         bne-    loop" ) // Loop if lost reservation
```

Explanation: Assume r4 contains the new value to be stored and r3 contains the address of the word. Then in the end, the new value is stored and old value returns to r4.

```
    #define CAS(R,M,V)
asm volatile( "Loop: lwarx    r6,0,r3 //Load and reserve
  cmpw    r4,r6 //Compare the first two operands
  bne- exit // Go to exit if not equal
stwcx r5,0,r3 //Store new value if still reserved
bne-    Loop //Loop if lost reservation
exit:      mr r4, r6" ) //Return value from storage
```

Explanation: Assume r5 contains the new value to be stored if match and r3 contains the address of the word. The value to be compared with the value in memory is in r4. Then in the end, the new value is stored after a successful match and old value returns to r4.

8.2 Wool for OSE

OSE is an operating system with powerful feature of OSE process and massage passing communication. However, in order to satisfy costumers to transplant their code directly on OSE, it also supplies OSE pthreads. Therefore, the possibility for Wool on OSE depends on the difference between OSE pthreads and POSIX pthreads. OSE now supports most of the pthreads APIs which cover the Wool, therefore Wool is able to run on OSE directly.

8.3 Modify Wool with OSE process

In Wool, the worker always steals the task from the bottom of victim task pool. The task pool is made of fixed size task descriptors. Although it will simplify the memory management, this kind of method is not as usual as TBB and Cilk++. TBB and Cilk++ have a list of free allocation task memory size which fit the OSE task pool management. Thus, it could be the first step to transfer Wool worker into an OSE process. Moreover, the stolen task is probably to be modified as an OSE signal and push into the OSE signal queue. In this way, the process is able to even select the potential victim by the massage filter.

8.4 Summary

Modifying Wool on P4080 and OSE is definitely possible. The hardware defining work in Wool.h is massive and complex. Low level machine knowledge requires to be investigated. The Wool's structure and working algorithm are the significant problem to understand. However, Wool for OSE is rather simple since OSE supports most of pthreads APIs. In the future, it will be a trend to fix Wool worker into a OSE process.

Chapter 9

Discussion and future work

Wool is a C library rather than a compiler code generator and preprocessors. Comparing with Cilk and TBB, it is competitive in light overhead and growing new technology utilizing. OSE are also a powerful Operating System and so far less potential in OSE signals are developed.

In the future, here gives some directions for information:

- Investigate the possibility of modifying the OSE process to a Wool worker and the Wool task data to an OSE message. And also allocate the task pool directly in the OSE user's pool.
- Adding hardware debugging feature on the System which will help to increase the transparency of the system.
- Improve the Wool's victim selection method. For instance, check the timestamp of a worker instead of random select.
- Figure out the possibility for Wool on a heterogeneous system and how is the performance.

Reference:

[1] Kumar, S., Hughes, C. J., and Nguyen, A. 2007. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. SIGARCH Comput. Archit. News 35, 2 (Jun. 2007), 162-173.

[2] P4080 reference manual

[3] P4080 QorIQ intergrated multicore communication processor family reference manual

[4] Wool user's guide

[5] Video Transcript, "Excerpts from a Conversation with Gordon Moore: Moore‟s Law", Intel Corporation, 2005

[6] Multicore Processors – A Necessity By Bryan Schauer

[7] L. Peng et al, "Memory Performance and Scalability of Intel‟s and AMD‟s Dual-Core Processors: A Case Study", IEEE, 2007

[8] Effective use of the shared cache in multi-core architectures By Tian Tian, Intel Corp., January 23, 2007

[9] Interconnection Networks www.cs.cf.ac.uk/Parallel/Year2/section5.html

[10] Interconnections in multi-core architecture: understanding mechanisms, overhead and scaling ;Rakesh Kumar

[11] Introduction to parallel computing Author: Blaise Barney, Lawrence Livermore national laboratory

[12] CSCI. 4210 Operating systems Deadlock http://www.cs.rpi.edu/academics/courses/fall09/os/c15/

[13] MHPCC(Maui High Performance Computing Center) SP Parallel programming workshop message passing overview

[14] Task Queue Implementation Pattern Ekaterina Gonina (Author), Jike Chong (Shepherd), UC Berkeley ParLab

Appendix:

Enea OSE

Enea OSE is a widely used real time operating system, optimized for distributed, fault-tolerant and embedded systems.

1 OSE fundamentals

There are two kernel types in the OSE family. One is called the OSE Real Time Kernel, designed for embedded systems. The other, which is similar to the Real Time Kernel, is called the OSE Soft Kernel and is intended to run in a host computer, such as a UNIX workstation, Linux or PC. Both kernels are developed for use in single-CPU and multi-CPU systems. Much effort has been put into OSE to make an application look the same irrespective of the number of CPUs involved.

1.1 Native versus Cross development

Native Development:

- Application is developed and executed on the same computer
- Usually using the same operating system environment for both

Cross Development

- Application is developed on one computer
- Will run on another computer

	Soft Kernel		**Real Time Kernel**	
	Windows	Solaris	Target/Freeze Mode or Run Mode	
Compiler	VisualC++	gcc	DIAB	GreenHills
SCD	VisualC++	DDD(gdb)	SingleStep	Multi

SCD-Source Code Debugger

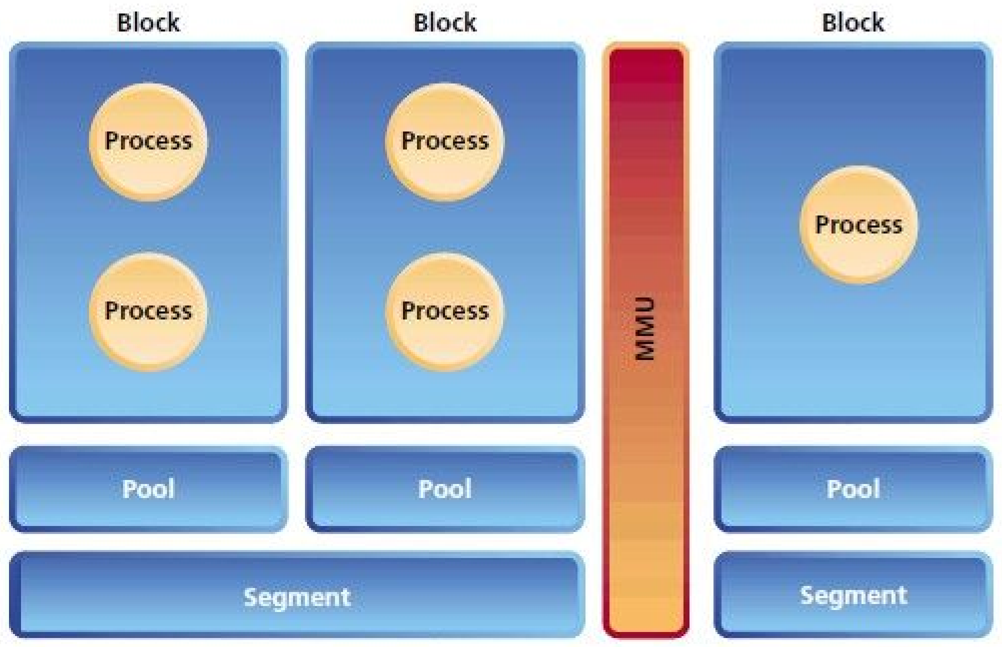
1.2 OSE Memory pool

- In OSE, the basic type of memory area is the *pool*. It is an area of memory from which signal buffers, stacks and kernel areas are allocated.

- Always one global memory pool: *system pool*. System processes and data reside in this pool, which must be located in kernel memory.

- Possible to create “local” pools using the same memory space as the processes they support. Pools can be created dynamically by a call to *create_pool ()* or statically by declaring it in *OSEMAIN.CON*.

- *ose_set_pool_max_size()* added from OSE5.5 for dynamic change of pool max size and fragment size.



A signal containing a pointer to the memory pool of sender is usually fast but dangerous. To avoid this danger, one or several pools can be grouped in a separate “domain”. This way, while sending a signal across segment boundaries the user is able to choose to copy the signal buffer from the sender segment to receive segment.

2 OSE architecture

OSE Architecture includes Init, Kernel, Core, core extensions and Platform.

The Init layer architectural responsibilities are:

- Target initialization, which brings the system to a known, safe state
- Run-time interface for persistent configuration management
- Run-time interface for low-level logging

The kernel layer architectural responsibilities are:

- Process scheduling services
- Logical and physical memory management
- Process management and synchronizations services
- Inter-process communication (IPC) services (using OSE signals)
- Basic time and time-out service
- Debug and kernel monitoring services
- Centralized error handler plug-in interface
- Link handler plug-in interface

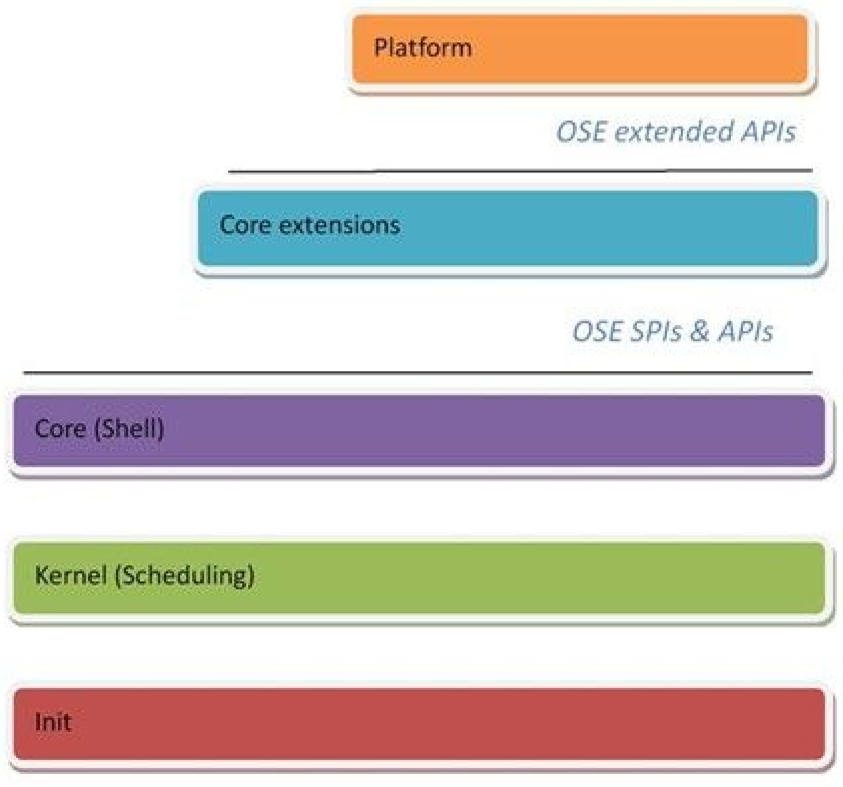
The core layer architectural responsibilities are:

- C/C++ run-time environment
- Heap memory service
- Clock and calendar services
- File system plug-in interface
- Login shell service
- Program management plug-in interface

The core extension layer extends the core services for different markets and customer applications

The platform layer consists of a number of OSE platform components that use the OSE API and SPI to provide the application layer with additional interfaces.

- Higher-level network protocols, such as FTP or HTTP
- Internet routing algorithms, such as RIP or OSPF
- Java and other virtual machines



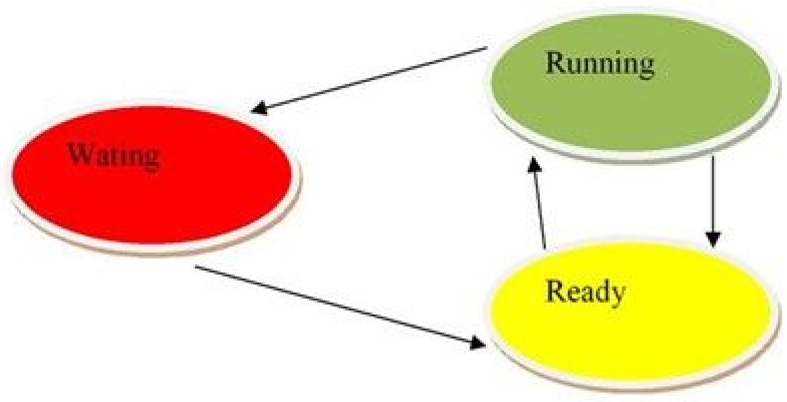
OSE architecture

3 OSE processes and IPC

OSE process is the fundamental building block within OSE which is a function with its “context”, i.e. its own stack and s set of specific variables and register values. Processes share the CPU time allocated by the kernel and they are divided into categories (static and dynamic) and types (interrupt, time-interrupt, prioritized, background and phantom). Therefore, in one word, an OSE process is a thread with some special features.

3.1 Process states

Since CPU time has to be shared by all the existing processes within a system, a process is not always running. Therefore, there exists three states for an OSE process: Ready, Waiting, Running.



States diagram

3.2 Process categories

Static processes are created at the system start by the kernel, or at the start of a load module. They are globally visible, supposed to exist for all the life of the system or load module and it is not allowed to kill a static process.

As opposed to static processes, there are dynamic processes which can be created, configured and killed freely in run-time, using system calls. Furthermore, dynamic processes can be created as multiple instances of the same code.

3.3 Process types

Interrupt processes

- Called in response to a hardware interrupt or software event (trap)
- Run from beginning to end each time
- May be interrupted by another interrupt process with higher priority

This process is designed for response-time critical task.

Timer interrupt processes

- Called in response to change in the system timer
- Act in exactly the same way as ordinary interrupt processes

This process is designed for high priority cyclic task.

Both processes above are Non-blocking system calls.

Prioritized processes

- Common process type
- Written as infinite loops
- Run as long as no interrupt process or another prioritized process with higher priority is ready to run, or it is blocking itself by a system call, or its time slice is over.

This process is designed for longer task than interrupt processes, not directly tied to some external events.

Background processes

- Run in a strict time-sharing mode at the lowest priority level (Usually Round-Robin)
- Written as infinite loops
- It may use blocking system calls

This process is designed for lowest priority level processes, used to spend leftover CPU time.

Phantom processes

- Not really a “process”
- Contain no code
- Unable to receive signals
- Contain only a signal redirection table
- Mainly used as part of a logical channel when communicating across target boundaries

This process is designed for special purpose, e.g. local “proxy” for remote processes, used by link handles.

3.4 IPC concepts

General OSE/IPC usually based on signals passing. OSE signals are not like Unix signals and it is more like a message passing communication. A signal is a message that is sent from one process to another. The signal contains some form of information that the originating process wishes to convey to the destination process. The signal also has some additional attributes that are set by the operating system. The signal attributes keep track of which process sent the signal (the sender), which process it was sent to (the addressee), which process owns the signal (the owner), its size, signal number and a few other things. These signals will be popped into the massage queue of the receiver process.

A process has only one signal queue which is created and administered by the kernel. The kernel also owns the signal queue and every signal in the queue are treated equally. The order, in which the signals are retrieved from the signal queue, is decided from within the receiving process and only the owner of the signal can modify the contents of the buffer.

In OSE, inter process synchronization and communication is done perfectly with signals. A signal is a synchronization primitive which can be used as a notification or acknowledge message as well as a data-carrying message. Signals can be used to implement other RT primitives such as semaphores, barriers or monitors.

4 Summary

OSE has its own specific development track and it utilizes message passing for inter communication which is quite different with Unix/Linux. It does support threads; however the OSE process is more powerful for embedded system. This feature adapts to multi-core platform and task parallelism which are the most important concentrated point in this research project.