Introduction of OpenMP and System Calls

Outline

- Parallel programming models
- Introduction of OpenMP
- OpenMP directive
- OpenMP clause
- Introduction of system calls
- Process & Thread
- CPU affinity
- Process priority

Parallel Programming Models

- Abstraction above hardware and memory architectures
- Common parallel programming models:
  - Shared memory

    E.g., Pthread, OpenMP
  - Message passing

    E.g., MPI
  - Data parallel (data decomposion)

    E.g., CUDA

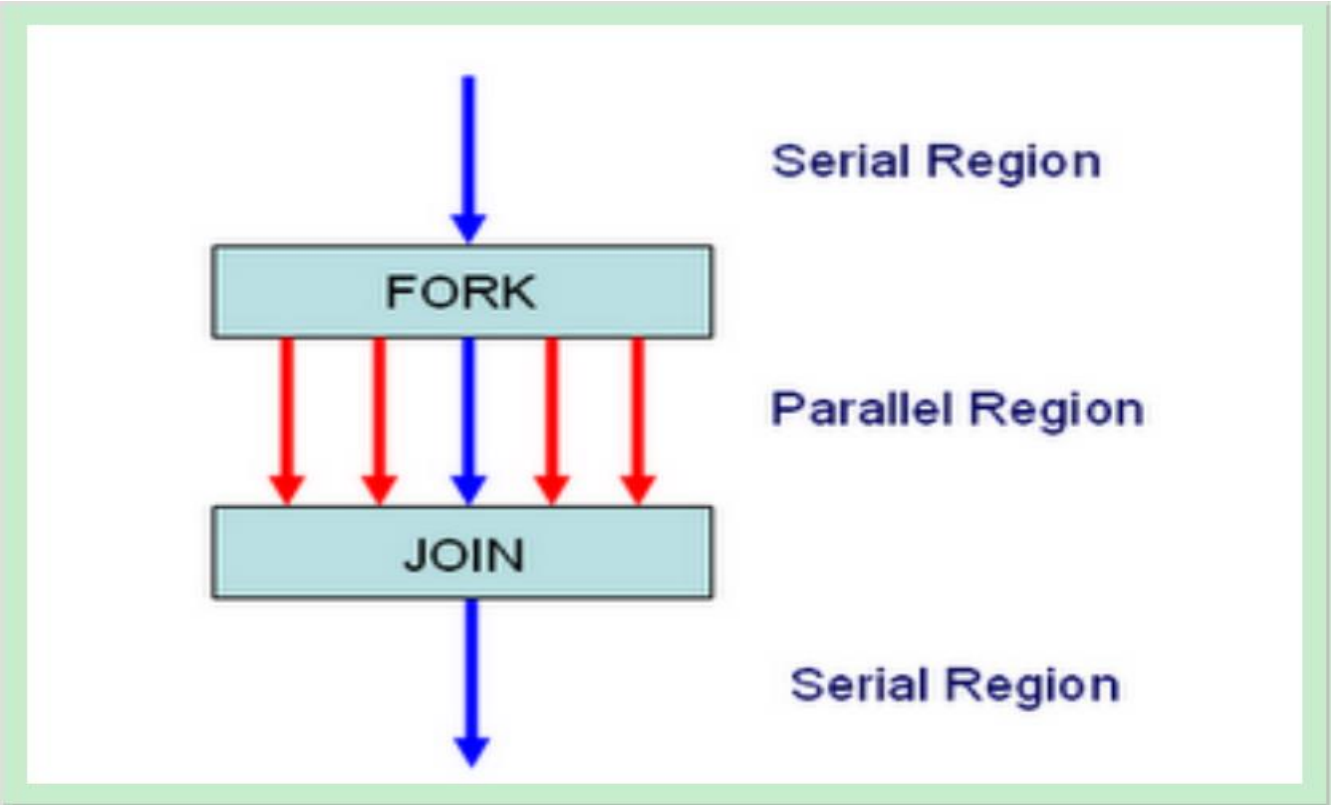
Outline

- Parallel programming models
- Introduction of OpenMP
- OpenMP directive
- OpenMP clause
- Introduction of system calls
- Process & Thread
- CPU affinity
- Process priority

Introduction

- The OpenMP provides API for developing parallel program on shared memory architecture.
- The OpenMP is for C/C++ and Fortran.
- It consists of directives, library functions, and environment variables.

Fork-join Model



Outline

- Parallel programming models
- Introduction of OpenMP
- OpenMP directive
- OpenMP clause
- Introduction of system calls
- Process & Thread
- CPU affinity
- Process priority

OpenMP Directive

- #include <omp.h>
- #pragma omp directive [clause,clause,…]
- OpenMP Directive Construct
  - Parallel Construct
  - Work-Sharing Constructs

Parallel Construct

- Parallel directive

```
void parallel_D()
{
	omp_set_num_threads(5);
	#pragma omp parallel
	{
		printf("%d¥n",omp_get_thread_num());
	}
}
```

Work-Sharing Constructs

- Loop directive
- Sections directive
- Single directive

Loop directive

```
void loop_D()
{
	omp_set_num_threads(2);
	int i,j;
	#pragma omp parallel
	{
		#pragma omp for
		for(i=0;i<10;i++)
		{
			printf("i = %d  thread id=
				%d¥n",i,omp_get_thread_num());
		}
	}
}
```

```
i =  0 thread id= 0
i =  1 thread id= 0
i =  5 thread id= 1
i =  2 thread id= 0
i =  6 thread id= 1
i =  3 thread id= 0
i =  4 thread id= 0
i =  7 thread id= 1
i =  8 thread id= 1
i =  9 thread id= 1
```

Sections directive

```
void sections_D()
{
      omp_set_num_threads(2);
      #pragma omp parallel{

            #pragma omp sections {
                  #pragma omp section {
                        printf("Execute First section by thread
                            %d¥n",omp_get_thread_num());
                  }

                  #pragma omp section{
                        printf("Execute second section by thread
                            %d¥n",omp_get_thread_num());
                  }
            }
      }
}
```

```
Execute First section by thread 0
Execute second section by thread 1
```

Single directive

```
void single_D()
{
      omp_set_num_threads(2);
       #pragma omp parallel {

            #pragma omp single{
                        delay();
                        printf("Execute first single region by thread
                        %d¥n",omp_get_thread_num());
             }

            printf("Execute parallel region by thread %d¥n",omp_get_thread_num());

            #pragma omp single
            printf("Execute second single region by thread
             %d¥n",omp_get_thread_num());
      }
}
```

```
Execute first single region by thread 1
Execute parallel region by thread  0
Execute parallel region by thread  1
Execute second single region by thread 1
```

Outline

- Parallel programming models
- Introduction of OpenMP
- OpenMP directive
- OpenMP clause
- Introduction of system calls
- Process & Thread
- CPU affinity
- Process priority

Clause Types

- Data-Sharing Attribute Clauses
  - default, reduction, shared
  - private
- Data Copying Clauses
  - copyin, copyprivate
- Other Clauses
  - if, nowait, num_threads, ordered, schedule

default and private

```
#pragma omp parallel
{
        #pragma omp for
        for( int i =  0 ; i < 3; ++ i )
                for( int j = 0; j < 3; ++X)
                        Test2( i , X );
}
```

```
<T:0> - 0, 0
<T:1> - 2, 0
<T:0> - 0, 1
<T:1> - 2, 1
<T:0> - 0, 2
<T:1> - 2, 2
<T:0> - 1, 0
<T:0> - 1, 1
<T:0> - 1, 2
```

```
Int I,j;
#pragma omp parallel
{
        #pragma omp for
        for( i =  0 ; i < 3; ++ i )
                for( j = 0; j < 3; ++X)
                        Test2( i , X );
}
```

```
<T:0> - 0, 0
<T:1> - 2, 0
<T:0> - 0, 1
<T:1> - 2, 2
<T:0> - 1, 0
<T:1> - 2, 1
<T:0> - 1, 2
```

default and private

```
//omp_shared_default.cpp
Int X
#pragma omp parallel default(none)
{
        #pragma omp for private( X )
        for( int i =  0 ; i < 3; ++ i )
                for( X = 0; X < 3; ++X)
                        Test2( i , X );
}


Void Test2( int n, int m)
{

        printf( “<T:%d> - %d, %dn”, omp_get_thread_num(),n,m);
}
```

reduction

```
omp_set_num_threads(4);
#pragma omp parallel reduction(+ : nCount)
{
        nCount += 1;
        printf_s("T:%d¥tnCount=%d¥n",omp_get_thread_num(),nCount);
        #pragma omp for reduction(+ : nSum)
        for (i = 1; i <= 10; ++i)
        {
                nSum += i;
        printf_s("T:%d¥ti=%d¥tnSum=%d¥n",omp_get_thread_num(),i,nSum);
        }
}
```

Sample output:

```
T:1    nCount=1
T:1    i=4    nSum=4
T:1    i=5    nSum=9
T:2    nCount=1
T:2    i=7    nSum=7
T:3    nCount=1
T:3    i=9    nSum=9
```

```
T:0    nCount=1
T:0    i=1    nSum=1
T:0    i=2    nSum=3
T:2    i=8    nSum=15
T:3    i=10   nSum=19
T:1    i=6    nSum=15
T:0    i=3    nSum=6
nCount=4      nSum=55
```

*Get thread number

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

void Test( int n )
{
	printf( "<T:%d> - %dn", omp_get_thread_num(), n );
}

int main(int argc, char* argv[])
{
	#pragma omp parallel { Test( 0 ); }
	system( "pause" );
}
```

```
<t:0> - 0
<t:1> - 0
```

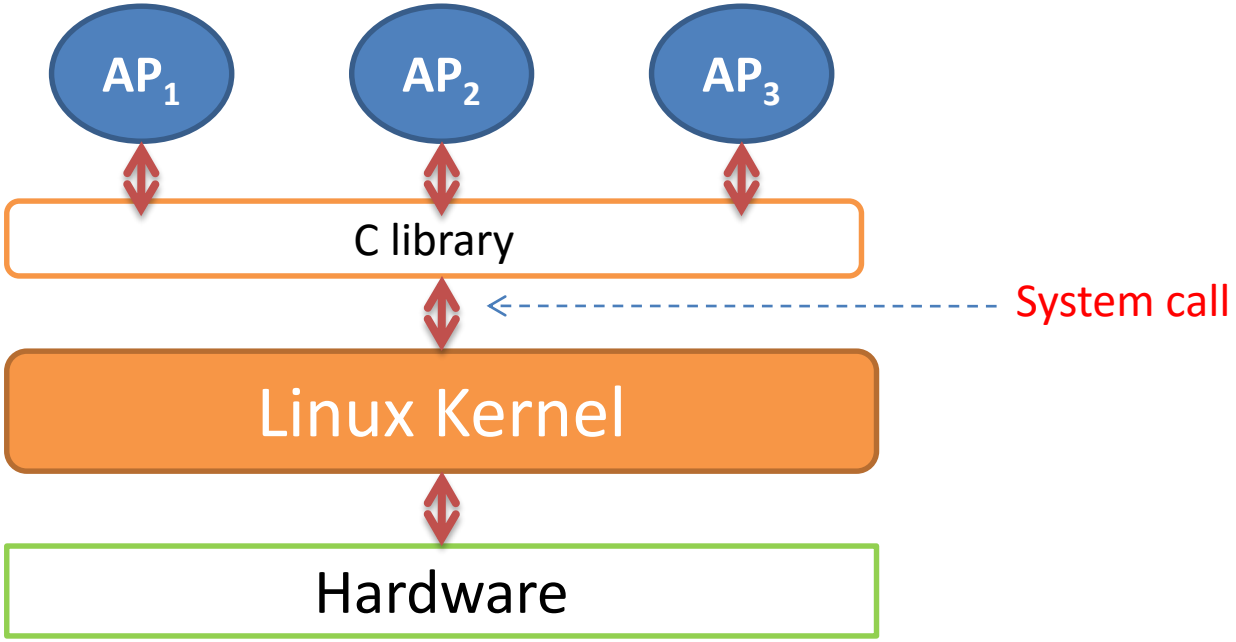
Outline

- Parallel programming models
- Introduction of OpenMP
- OpenMP directive
- OpenMP clause
- Introduction of system calls
- Process & Thread
- CPU affinity
- Process priority

System Call

Definition:

System calls provide an interface between the hardware and user-space processes.

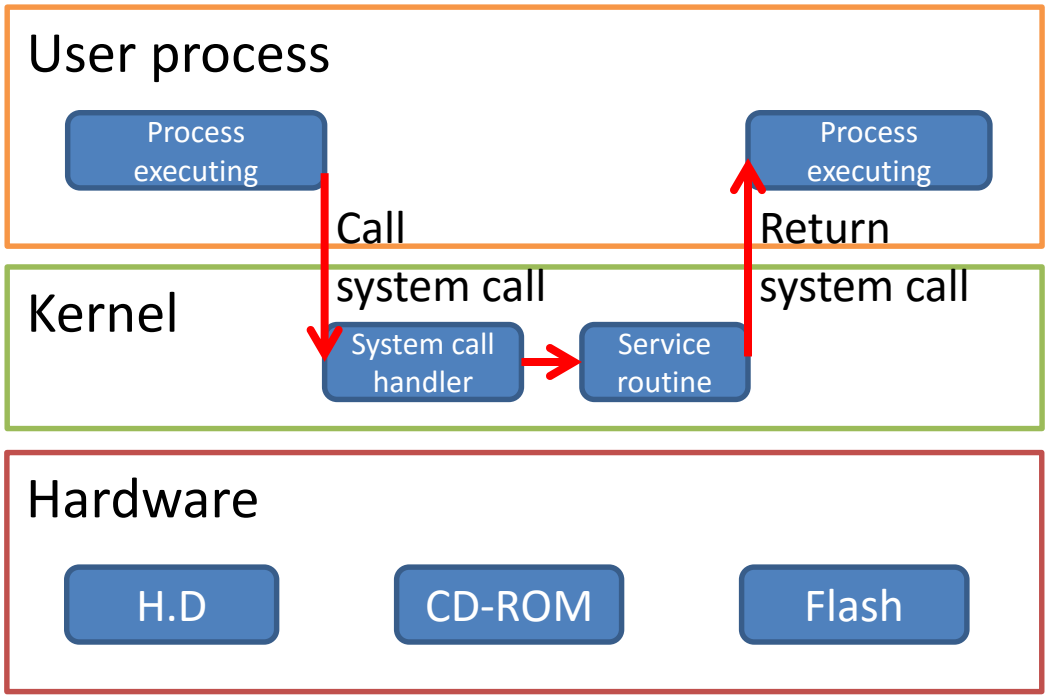


System Call

- Example:

  There is a file I/O program executing on user mode.

  – File name: c:¥students¥score.txt

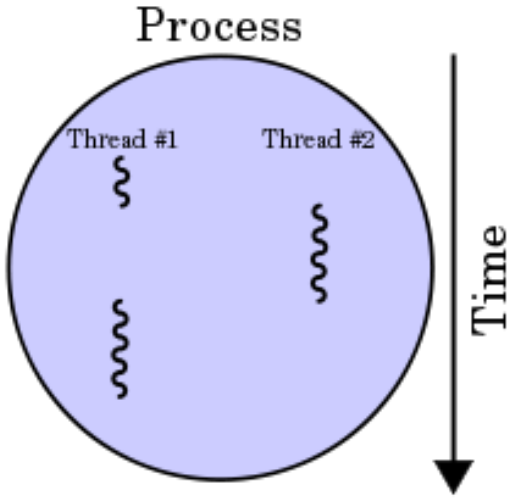


Outline

- Parallel programming models
- Introduction of OpenMP
- OpenMP directive
- OpenMP clause
- Introduction of system calls
- Process & Thread
- CPU affinity
- Process priority

Process & Thread

- In computing, a **process** is an instance of a computer program that is being executed. It contains the program code and its current activity .
- All processes are composed of one or more threads.



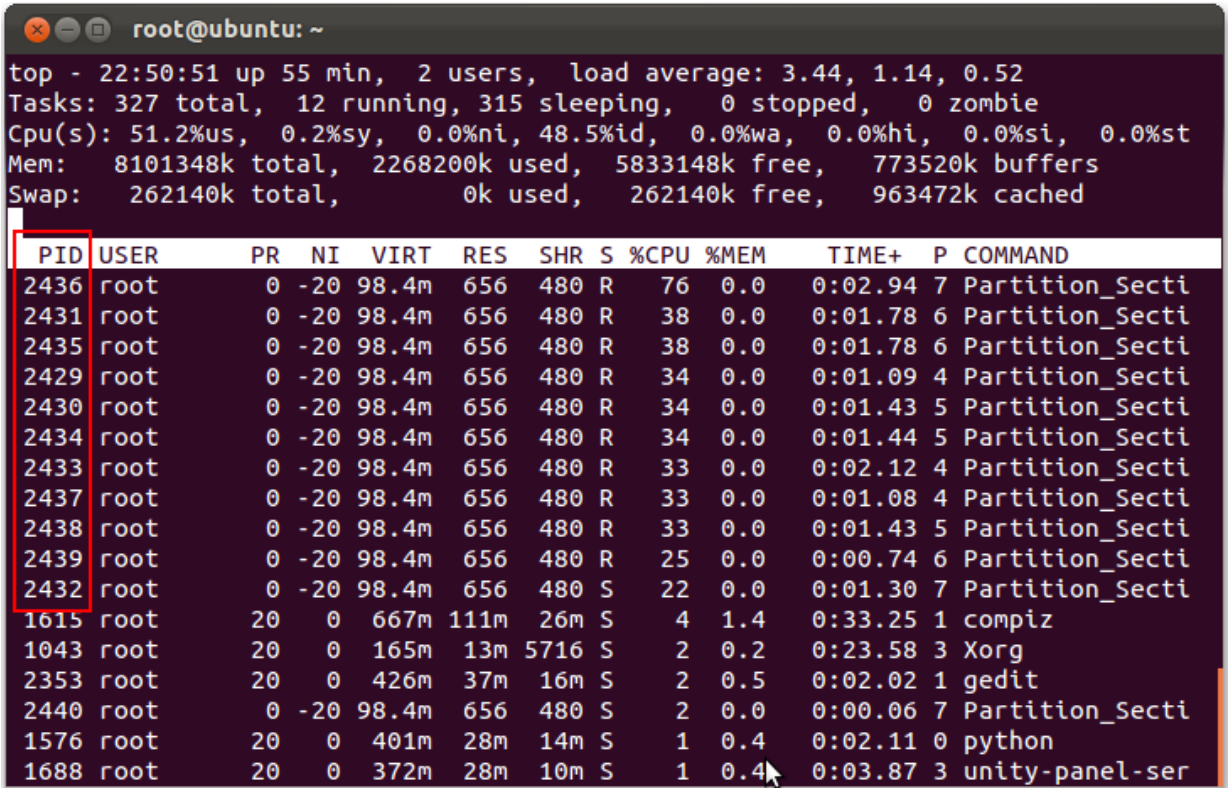
Process ID

```
/* getpid: print a process' process ID */
#include <stdlib.h> /* needed to define exit() */
#include <unistd.h> /* needed to define getpid() */
#include <stdio.h>   /* needed for printf() */

int main(int argc, char **argv) {

                        printf("my process ID is %d¥n", getpid());

                        exit(0);
}
```

Thread ID

- We can get the real thread ID of each thread by combining the function “omp_get_thread_num()” which is provided by OpenMP with getpid().

```
PID=getpid()+omp_get_thread_num();
```


root@ubuntu: ~
top - 22:50:51 up 55 min, 2 users, load average: 3.44, 1.14, 0.52
Tasks: 327 total, 12 running, 315 sleeping,   0 stopped,   0 zombie
Cpu(s): 51.2%us, 0.2%sy, 0.0%ni, 48.5%id, 0.0%wa, 0.0%hi, 0.0%si, 0.0%st
Mem: 8101348k total, 2268200k used, 5833148k free,   773520k buffers
Swap: 262140k total,   0k used,   262140k free,   963472k cached
PID USER PR NI VIRT RES SHR S %CPU %MEM TIME+ P COMMAND
2436 root 0 -20 98.4m 656 480 R 76 0.0 0:02.94 7 Partition_Secti
2431 root 0 -20 98.4m 656 480 R 38 0.0 0:01.78 6 Partition_Secti
2435 root 0 -20 98.4m 656 480 R 38 0.0 0:01.78 6 Partition_Secti
2429 root 0 -20 98.4m 656 480 R 34 0.0 0:01.09 4 Partition_Secti
2430 root 0 -20 98.4m 656 480 R 34 0.0 0:01.43 5 Partition_Secti
2434 root 0 -20 98.4m 656 480 R 34 0.0 0:01.44 5 Partition_Secti
2433 root 0 -20 98.4m 656 480 R 33 0.0 0:02.12 4 Partition_Secti
2437 root 0 -20 98.4m 656 480 R 33 0.0 0:01.08 4 Partition_Secti
2438 root 0 -20 98.4m 656 480 R 33 0.0 0:01.43 5 Partition_Secti
2439 root 0 -20 98.4m 656 480 R 25 0.0 0:00.74 6 Partition_Secti
2432 root 0 -20 98.4m 656 480 S 22 0.0 0:01.30 7 Partition_Secti
1615 root 20 0 667m 111m 26m S 4 1.4 0:33.25 1 compiz
1043 root 20 0 165m 13m 5716 S 2 0.2 0:23.58 3 Xorg
2353 root 20 0 426m 37m 16m S 2 0.5 0:02.02 1 gedit
2440 root 0 -20 98.4m 656 480 S 2 0.0 0:00.06 7 Partition_Secti
1576 root 20 0 401m 28m 14m S 1 0.4 0:02.11 0 python
1688 root 20 0 372m 28m 10m S 1 0.4 0:03.87 3 unity-panel-ser

Outline

- Parallel programming models
- Introduction of OpenMP
- OpenMP directive
- OpenMP clause
- Introduction of system calls
- Process & Thread
- CPU affinity
- Process priority

CPU affinity

- **Processor affinity** enables the binding and un-binding of a process or thread to a physical CPU or a range of CPUs.

- Scheduling algorithm implementations vary in adherence to processor affinity.

Example

```
#define _GNU_SOURCE
#include <sched.h>

int main(int argc, char** argv)
{
        cpu_set_t set;
        int ret, i;

        CPU_ZERO(&set);
        CPU_SET( 4 , &set);
        ret=sched_setaffinity(0, sizeof(cpu_set_t), &set);

}
```

Outline

- Parallel programming models
- Introduction of OpenMP
- OpenMP directive
- OpenMP clause
- Introduction of system calls
- Process & Thread
- CPU affinity
- Process priority

Process priority

- The scheduler arranges the processes in the ready queue in order of their priority.

- Lower priority processes get interrupted by incoming higher priority processes.

Example

```
#include <unistd.h>
#include <sys/resource.h>

int main(int argc, char** argv)
{
        int ret;
        int PID;
        int priority = -20;

        PID = getpid();
        ret = setpriority( PRIO_PROCESS, PID, priority );

}
```

Reference

- http://kheresy.wordpress.com/2006/09/15/
- http://en.wikipedia.org/wiki/
- OpenMP 教學, 謝仁偉 博士
- http://linux.die.net