1 Understanding and Conception

1.1 Introduction of Linux

What is Linux?

As Microsoft windows, Linux is an operation system. But as the view from user Linux is quite different from MS windows and other OS we are familiar with. Mainly in three ways:

- Linux is a cross-platform operating system that runs on many computer models. Of course MS windows dose it now. Linux is free, in two senses. First, you may pay nothing to obtain and use Linux. On the other hand, you may choose to purchase Linux from a vendor who bundles Linux with special documentation or applications, or who provides technical support. That's why it is popular all over the world now.
- Second, and more important, Linux and many Linux applications are distributed in source form. This makes it possible for you and others to modify or improve them. You're not free to do this with most operating systems, which are distributed in binary form Because of this freedom; Linux is being constantly improved and updated, far outpacing the rate of progress of any other operating system. For example, Linux will likely be the first operating system to support Intel's forthcoming Merced 64-bit CPU.
- Linux has attractive features and performance. Free access to Linux source code lets programmers around the world implement new features, and tweak Linux to improve its performance and reliability. The best of these features and tweaks are incorporated in the standard Linux kernel or made available as kernel patches or applications. Not even Microsoft can mobilize and support a software development team as large and dedicated as the volunteer Linux software development team, which numbers in the hundreds of thousands, including programmers, code reviewers, and testers.

1.2 Debian GNU/Linux

Since Linux is free, it has many different versions so that can meet the need of different user. Like Redhat or Konipox, Debian is one of the members of Linux family. And this OS is also a tool to help me to fulfill my task in Germany.

The kernel version of Debian:

Linux host 2.6.8-2-386 #1 Thu May 19 17:40:50 JST 2005 i686 GNU/Linux

The compiler for me to compile my C++ code is GCC 3.3

1.3 Conceptions about Scheduling

Before given the conception of Scheduling, the process and thread should be introduced first.

1.3.1 What is process?

A process is a running instance of a program or running program. Processes are an abstraction created to embody the state of a program during its execution. This means keeping track of the data that is associated with a thread or thread execution, which includes variables, hardware state, and the contents of an address space. Every task run on Linux can be called a task, every user task or system management process can also be called process. The mechanism Linux used to share the system resource for all task is time-share management.
Processes differ from programs though they are seemed to be similar with each other at first glance. Program is only a static commands collection and does not occupy the running resource of system while Process does use the system resource, it is dynamic, vary with time. Processes own resources allocated by the operating system. Most of all processes are generated from programs and one program can create multiple processes.
Let's take IE one of the most familiar internet browsers to us as an example. Any veteran knows how to open a web page or access to website by imitate IE, and you can browse different webpage in different windows. And this is it. IE is a program and system compile the source code then generates a process and in turns of user a window display and user can see the webpage. Each time initiating of IE a process is generated. That's why you could switch WebPages among multiple windows.

1.3.2 States of process

In a multitasking computer system, processes may occupy a variety of states. The following process states are possible to most operation system.

Create

When a process is first created, it occupies the "created" or "new" state. In this state, the process waits for switch to the "ready" state. This admission will be approved or delayed by a long-term, or admission, scheduler. Typically in most desktop computer systems, this admission will be approved automatically, however for real time operating systems this admission may be delayed. In a real time system, admitting too many processes to the "ready" state may lead to over-saturation and over-contention for the systems resources, leading to an inability to meet process deadlines.

Ready

A "ready" or "waiting" process has been loaded into main memory and is awaiting execution on a CPU (to be context switched onto the CPU by the dispatcher, or short-term scheduler). There may be many "ready" processes at any one point of the systems execution - for example, in a one processor system, only one process can be executing at any one time, and all other "concurrently executing" processes will be waiting for execution.

Running

A "running", "executing" or "active" process is a process which is currently executing on a CPU. From this state the process may exceed its allocated time slice and be context switched out and back to "ready" by the operating system, it may indicate that it has finished and be terminated or it may block on some needed resource (such as an input / output resource) and be moved to a "blocked" state.

Blocked

Should a process "block" on a resource (such as a file, a semaphore or a device), it will be removed from the CPU (as a blocked process cannot continue execution) and will be in the blocked state. The process will remain "blocked" until its resource becomes available, which can unfortunately lead to deadlock. From the blocked state, the operating system may notify the process of the availability of the resource it is blocking on (the operating system itself may be alerted to the resource availability by an interrupt). Once the operating system is aware that a process is no longer blocking, the process is again "ready" and can from there be dispatched to its "running" state, and from there the process may make use of its newly available resource.

Terminated

A process may be terminated, either from the "running" state by completing its execution, or by being explicitly killed. In either of these cases, the process moves to the "terminated" state.

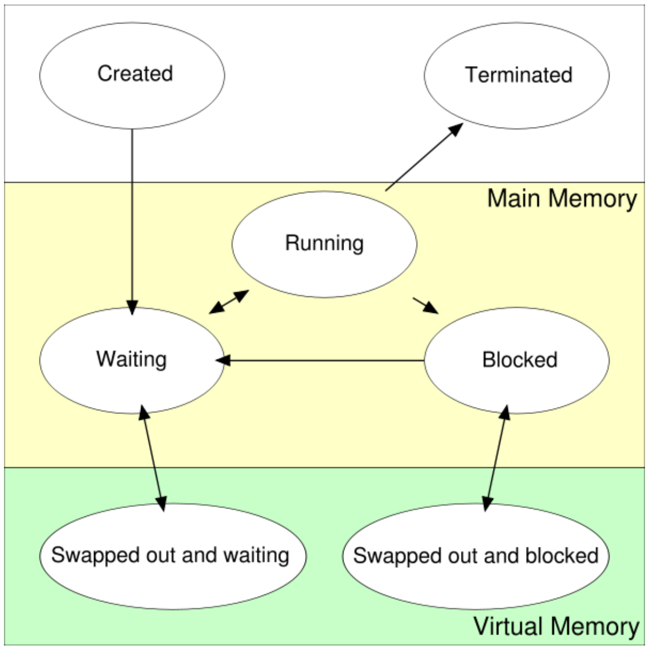

Figure 1-1

1.3.3 Process States in Linux

As Linux is a multitasking operating system. It of course has own process states.

TASK_RUNNING: The processes are running (current process) or get ready to be run(ready)

WAITING: At this state processes are waiting for events happen or system resource. In Linux there are two kinds of waiting processes, one is Task-interruptible another is Task-uninterruptible. The Task-interruptible process can be interrupt by a Signal and the later one can’t, it will change its state until some hardware states change happen.

TASK_STOPPED: Processes are terminated, normally by receiving a signal. Processes which can be debugged might remain this state.

TASK_ZOMBIE: Processes are terminated for some reasons, however the control-structure of process (task_struct) still remain.

The following graphic show how processes switch their states in Linux:

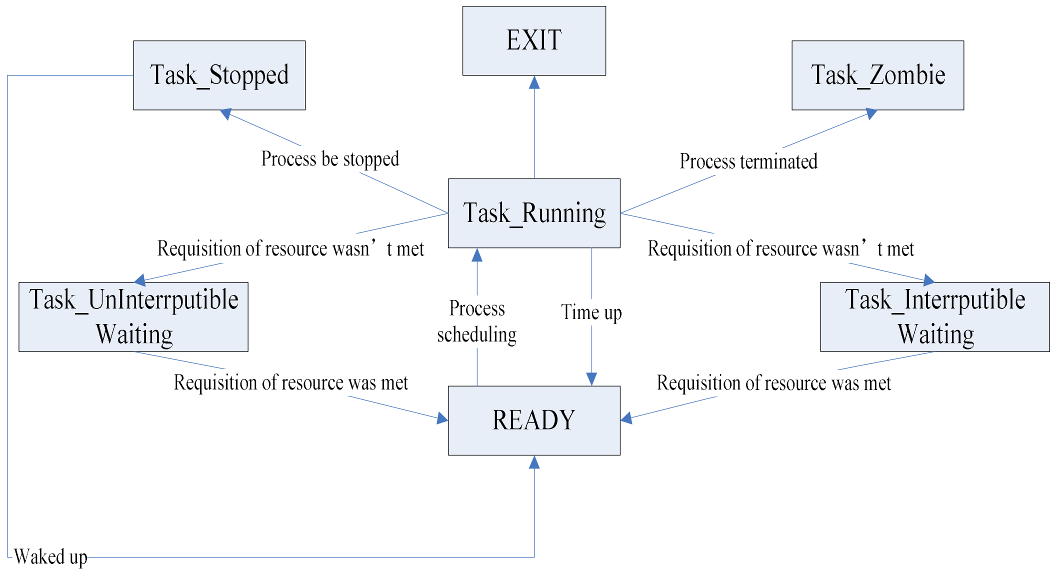
EXIT
Task_Stopped
Task_Zombie
Process be stopped
Process terminated
Task_Running
Requisition of resource wasn’t met
Requisition of resource wasn’t met
Task_UnInterruptible Waiting
Process scheduling
Time up
Task_Interrputible Waiting
Requisition of resource was met
Requisition of resource was met
READY
Waked up

Figure 1-2

1.3.4 What is Tread?

A thread in computer science is short for a thread of execution. Threads are a way for a program to split itself into two or more simultaneously running tasks. Multiple threads can be executed in parallel on many computer systems. This multithreading generally occurs by time slicing (where a single processor switches between different threads) or by multiprocessing (where threads are executed on separate processors). Threads are similar to processes, but differ in the way that they share resources. Threads allow a program to do multiple things concurrently. Since the threads, spawned by a program, share the same address space, one thread can modify data that is used by another thread easily and efficiently.

Linux is multi-processes and multi-threads operating system. Threads in Linux are defined as processes’ “context”.

1.3.5 Difference between processes and threads

Threads are distinguished from traditional multi-tasking operating system processes in that processes are typically independent, carry considerable state information, have separate address spaces, and interact only through system-provided inter-process communication mechanisms. Multiple threads, on the other hand, typically share the state information of a single process, and share memory and other resources directly.

A thread is the "lightest" unit of kernel scheduling. At least one thread exists within each process. If multiple threads can exist within a process, then they share the same memory and file resources. Threads are pre-emptively multitasked if the operating system's process scheduler is pre-emptive. Threads do not own resources except for a stack and a copy of the registers including the program counter.

On the contrary, a *process* is the "heaviest" unit of kernel scheduling. Processes own resources allocated by the operating system. Resources include memory, file handles, sockets, device handles, and windows. Processes do not share address spaces or file resources except through explicit methods such as inheriting file handles or shared memory segments, or mapping the same file in a shared way. Processes are typically pre-emptively multitasked.

1.3.6 Relationship between Processes and threads

The conception of threads and processes are very close to each other. The operating system creates a process for the purpose of running a program. Every process has at least one thread. On some operating systems, a process can have more than one thread. A kernel must keep track of each thread's stack and hardware state, or whatever is necessary to track a single flow of execution within a process. However, what most important is that only one thread may be executing on a CPU as any given time, which is basically the reason kernels have CPU scheduler.

In Linux, every process should contain an executable program, dedicated stack, private "process control model" (task_struct—a data structure) and owns private address spaces. However, threads in Linux are the combination of the former three parts —executable program, dedicated stack, private "process control model".

1.3.7 What is Scheduling?

At last we come to the point –- Scheduling. Scheduling is the process of assigning tasks to a set of resources. Scheduling is a key concept in multitasking and multiprocessing operating system design, and in real-time operating system design. It refers to the way processes are assigned priorities in a priority queue. This assignment is carried out by software known as a scheduler.

In general-purpose operating systems, the goal of the scheduler is to balance processor loads, and prevent any one process from either monopolizing the processor or being starved for resources.

In real-time environments, such as devices for automatic control in industry (for example robotics), the scheduler also must ensure that processes can meet deadlines; this is crucial for keeping the system stable.

1.3.8 Types of operating system schedulers:

Operating Systems may feature up to 3 distinct types of schedulers: a long-term scheduler (also known as an admission scheduler), a mid-term or medium-term scheduler and a short-term scheduler (also known as a dispatcher). This time we merely focus on short-term scheduler.

The short-term scheduler (also known as the dispatcher) decides which of the ready, in memory processes are to be executed (allocated a CPU) next, following a clock interrupt, an I/O interrupt, an operating system call or another form of signal. Thus the short-term scheduler makes scheduling decisions much more frequently than the long-term or mid-term schedulers - a scheduling decision will at a minimum have to be made after every time slice and these are very short. This scheduler can be preemptive, implying that it is capable of forcibly removing processes from a CPU when it decides to allocate that CPU to another process, or non-preemptive, in which case the scheduler is unable to "force" processes off the CPU.

1.3.9 Scheduling disciplines

Scheduling disciplines are algorithms used for distributing resources among parties which simultaneously and asynchronously request them. Scheduling disciplines are used in routers (to handle packet traffic) as well as in operating systems (to share CPU time among threads and processes). The main purposes of scheduling algorithms are to minimize resource starvation and to ensure fairness amongst the parties utilizing the resources.

1.4 Some of Scheduling algorithm

The Scheduling algorithms can be divided into two categories with respect to how they deal with clock interrupts.

Non-preemptive Scheduling

A scheduling discipline is non-preemptive if, once a process has been given the CPU it cannot be taken away from that process. Following are some characteristics of non-preemptive scheduling:

Short jobs are made to wait by longer jobs but the overall treatment of all processes is fair.

Response times are more predictable because incoming high priority jobs can not displace waiting jobs.

A scheduler executes jobs in the following two situations:

a. When a process switches from running state to the waiting state.

b. When a process terminates.

Preemptive Scheduling

A scheduling discipline is preemptive if, once a process has been given the CPU can taken away.

The strategy of allowing processes that are logically runable to be temporarily suspended is called Preemptive Scheduling and it is contrast to the "run to completion" method.

Starvation

Starvation is a multitasking-related problem, where a process is perpetually denied necessary resources. Starvation occurs when one program holds resources the other needs, but is unwilling to give them up.

Round-robin: Round-robin algorithm is one of the scheduling algorithms which assign time slices to each process in equal portions and order, handling all processes as having the same priority in order to prevent starvation of system resource. In prioritized scheduling systems, processes on an equal priority are often addressed in a round-robin manner. This algorithm starts at the beginning of the list of PDBs (Process Descriptor Block), giving each application in turn a chance at the CPU when time slices become available. It can decide who to run next by following the array or chain of PDBs to the next element, for the operating system must have a reference to the start of the list, and a reference to the current application, and if the end of the array or list be reached, reset back to the start. In all, Round-robin and Token-Ring in network area are much alike.

FCFS (first come, first serve): This is a non-preemptive technique. A single queue of ready processes is maintained, and the dispatcher always picks the first one. This method does not emphasis throughput, since long processes are allowed to monopolise CPU, For the same reason, the response time with FCFS can be high (with respect to execution time), especially if there is a high variance in process execution times. It's fairly obvious that this method penalizes short processes following long ones, though no starvation is possible. For there always a single, the starvation is impossible under this algorithm.

SJF (short job first): Shortest-Job-First (SJF) is a non-preemptive discipline in which waiting job (or process) with the smallest estimated run-time-to-completion is run next. SJF is a scheduling algorithm that assigns to each process the length of its next CPU burst/execution time. CPU is then given to the process with the minimal CPU burst from the waiting queue. SJF is provably optimal, in that for a given set of processes and their CPU bursts/execution times it gives the least average waiting time for each process. The SJF scheduling is especially appropriate for batch jobs for which the run times are known in advance. The SJF algorithm favors short jobs (or processors) at the expense of longer ones.

However, the obvious problem with SJF scheme is that it requires precise knowledge of how long a job or process will run and there is the potential for starvation in SJF. For the long processes have to await the short processes be executed first the long ones may have no chance to be executed.

1.4.1 what is real-time scheduling

Real-time in computer area means that correctness of result depends on both functional correctness and time that the result is delivered. In most situation Real-time is separate into two branch---- **Soft Real-time** and **Hard Real-time**.

A system is said to be **soft real-time** if the correctness of an operation depends not only upon the logical correctness of the operation but also upon the time at which it is performed. Tasks completed after their respective deadlines are less important than those whose deadlines have not yet expired. This leads to some tasks not being performed if the system load is too high.

A system is said to be **hard real-time** if the correctness of an operation depends not only upon the logical correctness of the operation but also upon the time at which it is performed. An operation performed after the deadline is, by definition, incorrect, and usually has no value. In a soft real-time system the value of an operation declines steadily after the deadline expires. Dynamic schedule computed at run-time based on tasks really executing. Static schedule done at compile time for all *possible* tasks.

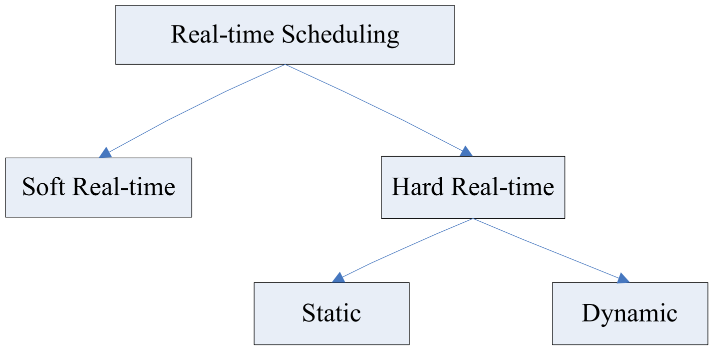

Figure 1-3

Linux scheduler supports soft Real-time scheduling. However, this means the Linux 2.6.x kernel is usually capable of meeting very strict RT scheduling deadline, it doesn’t guarantee that deadline will be met. Real-time tasks are assigned special scheduling modes and the scheduler gives them priority over any other task on the system. RT scheduling modes include a FIFO mode which allows RT tasks to run to completion on FCFS basis, and a RR scheduling mode that schedules RT tasks in a RR Fashion while essentially ignoring non-RT tasks on the system.

2 The Performance of Measure and the Criterion

2.1 Performance metrics

Performance metrics include:

Throughput: -Number of processes completed/time unit or the quantity of

	Bytes Reading&Writing.
RunningTime:	-The execution time of the process.
StartTime:	-The time that indicate when process be initiated.
AverageRunningTime:	-The average running time of all processes.
	(Average=All process/the number of processes)
The max-RunningTime:	-The MaxTime is the longest execution time of processes.
The min-RunningTime:	-The MinTime is the shortest execution time of processes.

On whole the average running time and the throughput are what we care about. Because short average-running-time under heavy throughput leads to quick performance. In another word the shorter the average time your machine is the quicker your machine is.

2.2 Tools are used in the measurement

To implement the measurement, some tools must be used besides C++ language. They are **Bash Shell**, **Cygwin** and **Gnuplot.**

2.2.1 What are CPU-Burst and I/O- burst

Generally CPU-Burst is a time interval when CPU is occupied by one process only. Normally loop is used to create the CPU burst cycle. Eventually a CPU burst will end with process termination

Actually there is no solid I/O-Burst. The definition of I/O-Burst like CPU-Burst that a time interval used by one process to deal with I/O operation. Nevertheless CPU is also involved in the I/O operation. Figures shown below describe how I/O-Burst works. Obviously it is CPU-I/O Burst cycle but the CPU has to wait a long time for the I/O operation.

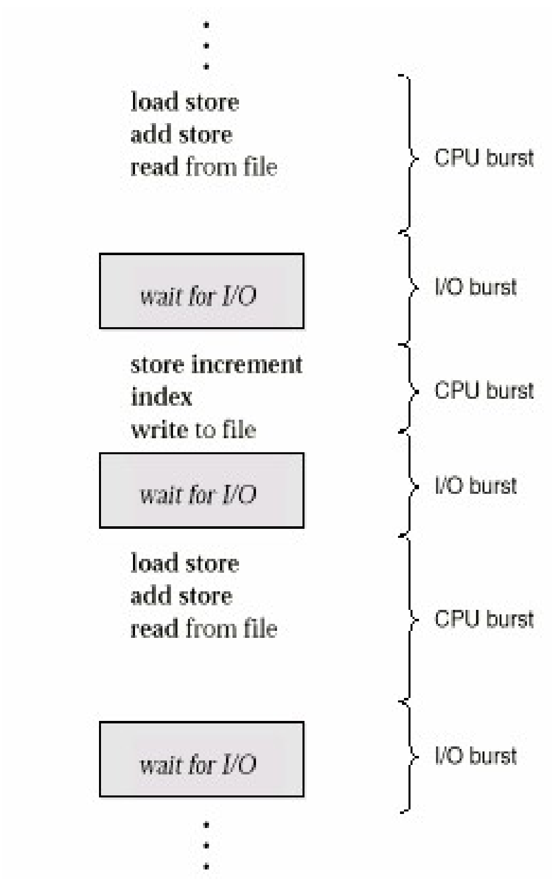

Figure 2-1

2.2.2 What is Bash Shell

Shell essentially provides a kind of interface for end-users. Typically, the term refers to an operating system shell which provides access to the services of a kernel. The name 'shell' originates from shells being an outer layer of interface between the user and the innards of the operating system (the kernel). Figure displayed below demonstrate the functionality of Shell. In another word Shell not only provide way for user to manipulate the operating system, also prevents user from accessing to the operation system kernel directly, like an egg, the shell of egg protect the albumen and yolk. also the shell is convenient for picking up because it is impossible grasp liquid in your hand.

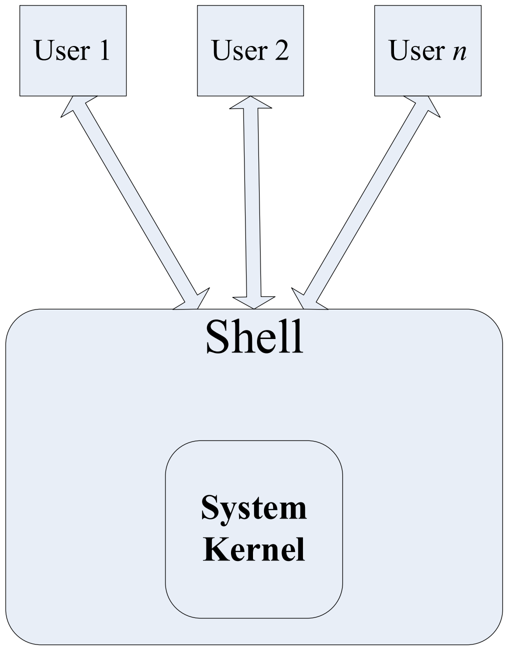

Figure 2-2

Operating system shells generally fall into one of two categories: command line and graphical. Command line shells provide a command line interface (CLI) to the operating system, while graphical shells provide a graphical user interface (GUI).

Bash is the shell, or command language interpreter, that will appear in the GNU operating system. Bash is a sh-compatible shell that incorporates useful features from the Korn shell (ksh) and C shell (csh). It is intended to conform to the IEEE POSIX P1003.2/ISO 9945.2 Shell and Tools standard. It offers functional improvements over sh for both programming and interactive use. In addition, most sh scripts can be run by Bash without modification.

In this task Bash is used to write script for the Shell and actually it include file manipulation, program execution, and printing text.

2.2.3 What is Gnuplot

Gnuplot is a freely distributed plotting tool with ports available for nearly every major platform. It can be operated in one of two modes: when you need to adjust and prettify a graph to "get it just right," you can operate it in interactive mode by issuing commands at the gnuplot prompt. Alternately, gnuplot can read commands from a file and produce graphs in batch mode. Batch-mode capability is especially useful if you are running a series of experiments and need to view graphs of the results after each run. In this case gnuplot version 4.0 is implemented.

2.2.4 What is Cygwin

Cygwin is a Linux-like environment for Windows. It consists of two parts:

1. A DLL (cygwin1.dll) which acts as a Linux API emulation layer providing substantial Linux API functionality.

2. A collection of tools, which provide Linux look and feel.

Cygwin's main job in this case is to provide a environment in Windows XP, so that measurement can be done under Windows.

2.3 How to measure

Actually we measure the CPU-burst and I/O-burst under parallel condition to measure Linux kernel 2.6 and kernel 2.4

Before continue introducing this condition. The signal and PCB should be explained first.

PCB--Process Control Block is a data structure that contains basic information and representing the state of a given process. Contents of Process Control Block are Process identification; Process status (HOLD, READY, RUNNING, WAITING);Process state (process status word, register contents, main memory info, resources, process priority); Accounting (CPU time, total time, memory occupancy, I/O operations, number of input records read, etc.)

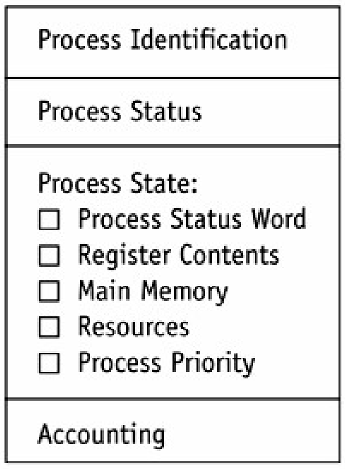

Figure 2-3

Every process has its own PCB.

Signal is a short message sent to a process, or group of processes, containing the number identifying the signal and used to notify a process or thread of a particular event.

In this case just like athlete wait for the “Pang” of signal gun. Processes are created and put in shared memory (talk about later) waiting for a signal which defined by user. Then system will schedule them. (The red spot in the Figure 2-4 means signal is received)

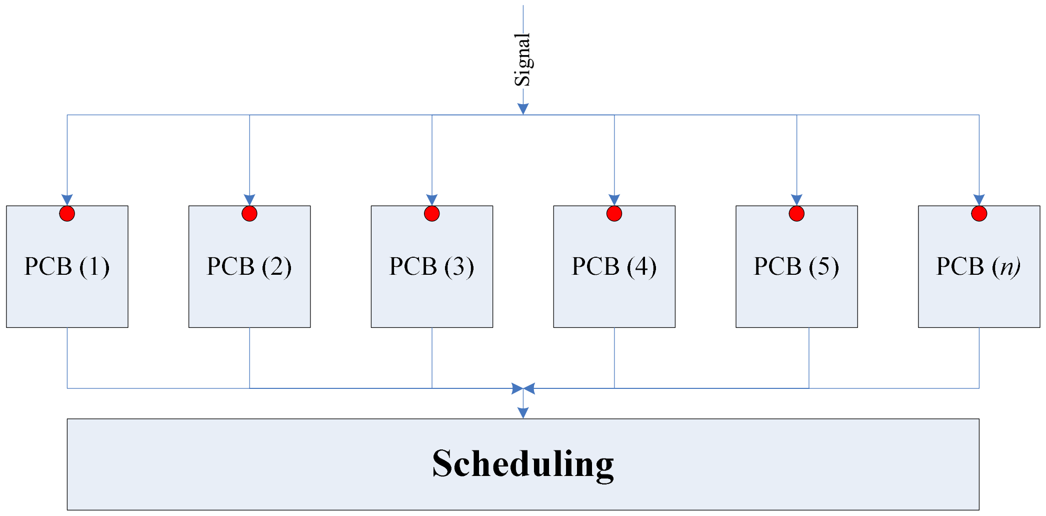

Figure 2-4

Measurement in Parallel condition:

Under this condition many processes are created and wait for the signal to be sent and then be scheduled at the same time.

As the figure shows below: T0 is the time that signal is sent. After received the signal all the processes generated will be scheduled by the operation system. According to the scheduling algorithm, processes are initiated at different time (T*n*0, n=A, B, C, D, E), though the signal is sent at the same time (T0) as far as all processes are concerned. The gap between T0 and T*n*0 is the waiting time. Waiting time is needed because the system scheduler schedules the processes into a priority queue. And time are different due to the preemptive scheduling; some system I/O operation with high priority will interrupt the current process let them be executed by the CPU first.

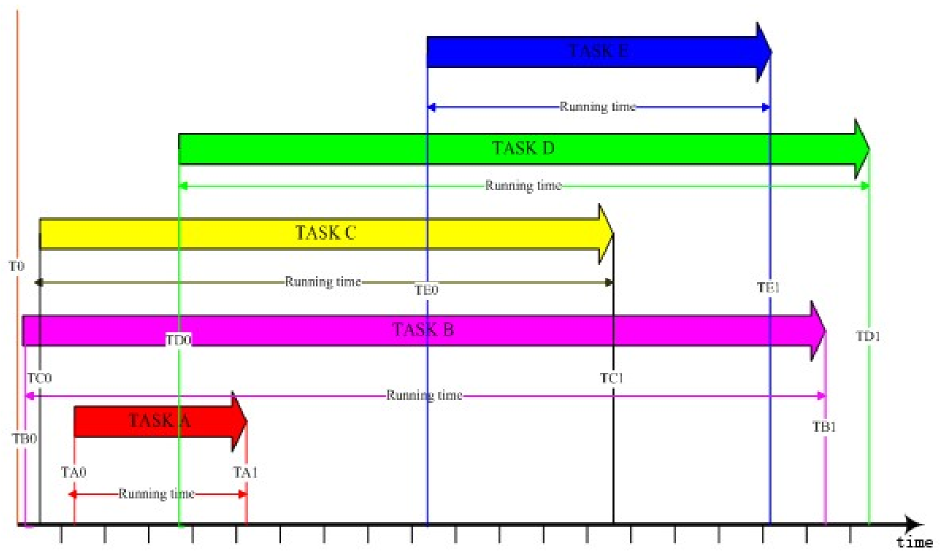

Figure 2-5

Context Switch is the computing process of storing and restoring the state of a CPU (the context) such that multiple processes can share a single CPU resource. The context switch is an essential feature of a multitasking operating system. A context switch is the computing process of storing and restoring the state of a CPU (the context) such that multiple processes can share a single CPU resource. The context switch is an essential feature of a multitasking operating system.

Actually before the signal is send, in context switch processes are sleeping and signal is used to waken them up.

2.3.1 Shared memory

To do the measurement in this condition shared memory should be used.

Shared memory refers to memory that is accessible by more than one process, where a process is a running instance of a program. In this context, shared memory is used to facilitate inter-process communication. See the entry for thread for more information about processes and threads.

2.3.2 Parallel condition of CPU-burst

In this case there are 2 bash shell scripts needed----**DoCPUfinal.sh** and **DrawGraphic.sh**. **DoCPUfinal.sh** takes charge of receiving arguments for **cpu-burst** generated by

cpu_burst2.cpp and redirect the result outcome to a temporally file **cpuplot.csv**. **DrawGraphic.sh** is responsible for getting the data from **cpuplot.csv** and generating the analysis graphic then save the picture to disk then remove the temp files.

In **cpu_burst2.cpp** first of all we receive 2 arguments form console, and then change the signal and create a shared memory segment.

```
if (argc <=2 )
    {
        ...    // argument missing output error message
    }
    else
    {
        ...    //receive arguments and define variable
        ...    //change the Signalhandler
        change_signalhandler (SIGUSR1, signal_sigusr1);
    // create a shared memory with the array writable for all processes
    // create a new SHM segment
    if ((shm_id = shmget(IPC_PRIVATE, sizeof(measure)*(loop1+1),
IPC_CREAT | SHM_R | SHM_W)) < 0)
        {
            ... // error message display SHM not created correctly
        }
        else
        {
            ... // output SHM_ID (shared memory segment ID)
        }
    }
```

Second of all, The *fork()* function contained in *<unistd.h>* header to create child process and child process shall be an exact copy of the calling process (parent process).

Upon successful completion, *fork()* shall return 0 to the child process and shall return the process ID of the child process to the parent process. Both processes shall continue to execute from the `fork()` function. Otherwise, -1 shall be returned to the parent process, no child process shall be created, and error shall be set to indicate the error. `If` condition is used to grantee the children processes to be created correctly.

```
if (pid != 0)            //no child then create
{
   pid=fork();
}
if (pid == -1)               //error happened
{
   cerr <<"cloud not fork" << endl;
   exit (-1);
}
if (pid == 0)            //child creation is successful
{
   ... // connect to the shared memory
   ... // start measurement
   ... // finish measurement and detach the shared memory
}
```

Function *`shmget();`* is used to create a shared memory and function *`void *shmat(int shmid, const void *shmaddr, int shmflg);`* is used to attach a shared memory segment or we can say connect to the shared memory on the contrary the *`coid *shmdt(const void *shmaddr);`* is used for detach or disconnect the shared memory. These functions are defined in *`<sys/shm.h>`* and *`<sys/sem.h> which`* are used for share memory creation and attach and detach.

After *`fork(),`* both the parent and the child processes shall be capable of executing independently before either one terminates and they are waiting for the “Pang” of operating system’ signal gun. (Like Figure 2-4 shows above). A signal is send though *`kill(0,SIGUSR1)`* to let all processes to be executed and the running order is according to scheduling mechanism.

Then a block of shared memory is opened to store children processes until the last child is created. The structure of shared memory in this case is below:

```
struct measure
{
    ourtime starttime;       //to preserve the start time
    ourtime stoptime;        //to preserve the stop time
    pid_t pid;               //to preserve the Process ID
};
```

`ourtime` is `defined by typedef struct timeval.`

<signal.h> defines the symbolic constants like SIG_ERR, pid_t, SIGUSR1 for user defined signal. The *kill()* function will send a signal to a process or a group of processes specified by pid. The signal to be sent is specified by *sig* and is either one from the list given in *<signal.h>* or 0.

If *sig* is 0 (the null signal), error checking is performed but no signal is actually sent. The null signal can be used to check the validity of pid.

If pid is greater than 0, `sig` will be sent to the process whose process ID is equal to pid.

If pid is 0, `sig` will be sent to all processes (excluding an unspecified set of system processes) whose process group ID is equal to the process group ID of the sender, and for which the process has permission to send a signal.

If pid is -1, sig will be sent to all processes (excluding an unspecified set of system processes) for which the process has permission to send that signal.

Finally the result of the measurement will be picked up from the structure Array and displayed on the screen unless it is redirected to a file.

Function `void process_routine(long double *loop2)` is used for CPU-Burst.

```
void process_routine(long double *loop2)
{
    unsigned int k;
    for (k=1; k<=*loop2; k++)
    {
        sqrt(k*(*loop2));
    }
}
```

`<sys/wait.h>` is declarations for waiting. The *`wait()`* function will suspend execution of the calling thread until status information for one of its terminated child processes is available, or until delivery of a signal whose action is either to execute a signal-catching function or to terminate the process.

Function *`static void signal_sigusr1 (int signum)`* is used for display which process is running. Function *int shm_status (int shm_id)* take charge of share memory creation and display the share memory information on the screen.

The signal state is changed by function *`static sighandler_t change_signalhandler (int signum, sighandler_t signalhandler).`*

DoCPUfinal.sh takes responsibility for arguments receiving and checking then redirect data to the temp file. The graphic drawing and store the graphic to disk is done by **DrawGraphic.sh**, and redirect the final result to *.csv file.

2.3.3 Parallel condition of I/O-burst

Mostly the **I/O-burst** and **CPU-burst** are much alike especially shared memory part and signal handle. It also composed by 3 files: **io_burst.cpp**, **DoIOfinal.sh** and **DrawDraphic-IO.sh**.

In **io_burst.cpp** processes are created the same way as **cpu_burst2.cpp**. But the arguments we need are 3 instead of 2. The 3 arguments are the number of processes we want to generate, the size of file we want to write into or read from the disk and the optional of the operator. In this case (1 means read; 2 means write). The *`<fstream>`* header is included, class fstream for I/O stream manipulation. The constructor of fstream takes 2 parameters. One is the file path and the second is the File Open mode. There are several open modes, each of them with a different purpose. The modes used in this case are *in* means open for input, `trunc` means truncate an existing stream when opening it and *out* means open for output.

Header *`<string>`* is standard ANSC lib responsible for deal with character inputting problem. In this case it is used for convert some other type of data to string type and get char type data from string. For instance, *string .c_str* returns a standard C character array version of the string.

I/O-burst operation in example is not the same as the formal condition. There is function like `process_routine(long double *)` use loop to generate the burst. Instead every process have to create a *.txt file, for the file is locked. Lock is a synchronization mechanism for enforcing limits on access to a resource in an environment where there are many threads of execution. Locks are one way of enforcing concurrency control policies. If file is locked, there is no possible for other processes to read or write it. The most typical example can be provided by Microsoft Office Word and Wordpad in Ms Windows. When you open a *.doc file by using Word, at the mean time you want to open it by Wordpad, then an error message will show up said that the *.doc file is opened by another program and can't be opened by Wordpad. This is also result from the monopolization of preemptive scheduling algorithm. By this reason every process is designate a *.txt file.

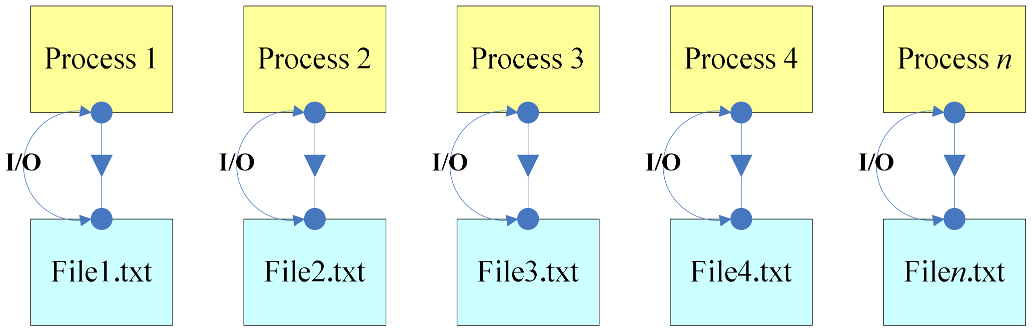

Figure 2-6

After given the number of the processes will be created. The size of every file should be inputted. Actually the size is 4**n* KB (n=the number received through I/O stream, the reason of multiply 4 is the characters be written into the file is "Jack" –my English name). Before creation of I/O-burst, options are needed for reading or writing. Object `readfile` defined by `ifstream` is used for reading bytes, vice verse `writefile` defined by `ofstream` is used for writing bytes.

`readfile.open( name.c_str(), ifstream::in | ifstream::trunc ),` `readfile.close()` are open and close operation toward the file. Function `readfile.read(block,sizeof(name.c_str()))` is used to read byes according to 2 arguments, the first is a pointer point to a memory block, the second is the total number of bytes will be read. `writefile.open( name.c_str(), ofstream::out |`

`ofstream::trunc )` and `writefile.close()` are used the same way as `read.open().` However bytes are written into file through "<<" operator.

DoIOfinal.sh is mostly alike **DoCPUfinal.sh** and **DrawDraphic-IO.sh** is the same as **DrawGraphic.sh**.

3 Experiments and Conclusion

3.1 Experiment environment

In this chapter, different machine will be used for testing the scheduling of different OS. We will implement the parallel condition on Linux kernel 2.6 and 2.4.

3.1.1 Linux Kernel 2.6.x

Linux Kernel 2.6 use *O (1)* algorithms make a lot of enhancement such as it allows the Linux kernel to efficiently handle massive numbers of tasks without increasing overhead costs as the number of tasks grows. However what put Kernel 2.6 under focus lamp is runqueues and priority arrays.

Essentially, a runqueue keeps track of all runnable tasks assigned to a particular CPU. As such, one runqueue is created and maintained for each CPU in a system. Each runqueue contains two priority arrays. All tasks on a CPU begin in one priority array, the active one, and as they run out of their timeslices they are moved to the expired priority array. During the move, a new timeslice is calculated. When there are no more runnable tasks in the active priority arrays, it is simply swapped with the expired priority array. The Linux 2.6.8.1 scheduler always schedules the highest priority task on a system, and if multiple tasks exist at the same priority level, they are scheduled round-robin with each other. Priority arrays allow the scheduler's algorithm to find the task with the highest priority in constant time. When a task is added to a priority array, it is added to the list within the array for its priority level. The highest priority task in a priority array is always scheduled first, and tasks within a certain priority level are scheduled round-robin.

3.1.2 Linux Kernel 2.4.x

The Linux kernel 2.4 scheduling `O (n)` algorithm works by dividing the CPU time into epochs. In a single epoch, every process has a specified time quantum whose duration is computed when the epoch begins. In general, different processes have different time quantum durations. The time quantum value is the maximum CPU time portion assigned to the process in that epoch. When a process has exhausted its time quantum, it is preempted and replaced by another runnable process. Of course, a process can be selected several times from the scheduler in the same epoch, as long as its quantum has not been exhausted--for instance, if it suspends itself to wait for I/O, it preserves some of its time quantum and can be selected again during the same epoch. The epoch ends when all runnable processes have exhausted their quantum; in this case, the scheduler algorithm recomputes the time-quantum durations of all processes and a new epoch begins.

3.2 Testing and Conclusion

CPU-Burst and I/O-Burst will be tested by using the source code which has been introduced briefly in previous chapter and will attached in Appendix

3.2.1 CPU-Burst Test & Conclusion

To ensure the accuracy of the data, every operation system will be tested under different amount of processes and different machine. But here according to the length only some typical data and figure will be cited. Every process will do 100,000 loops.

Data of kernel 2.6

Linux kernel version: 2.6.11-1.1369_FC4smp

CONFIG_SCHED_SMT=y

CONFIG_IOSCHED_NOOP=y

CONFIG_IOSCHED_AS=y

CONFIG_IOSCHED_DEADLINE=y

CONFIG_IOSCHED_CFQ=y

CONFIG_NET_SCHED=y

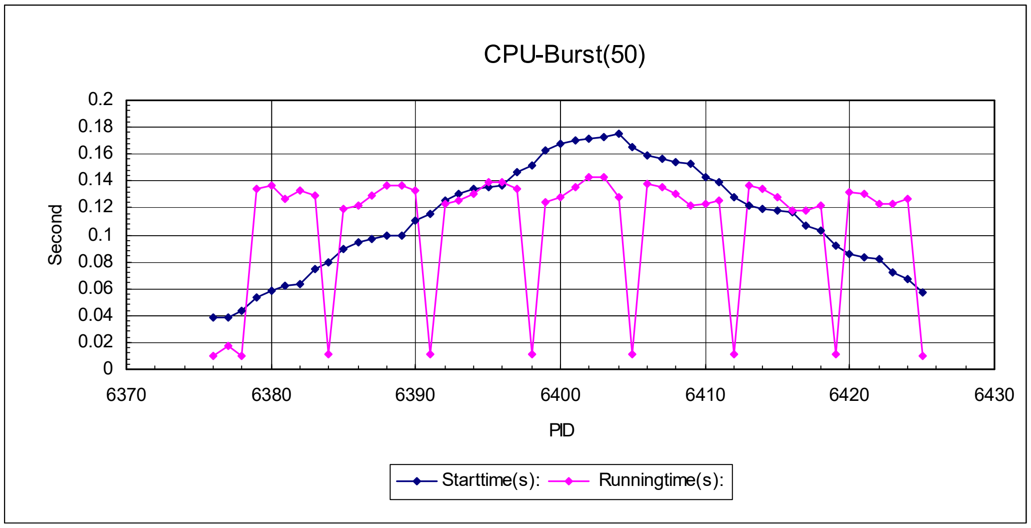

Figure 3-1-1

(the number in the Title is processes number)

Total Processes:	50	**Loop number:**	100,000
Average time:	0.106143	**Deviation:**	0.037958374
Max executing time:	0.142984	**Min executing time:**	0.0104878

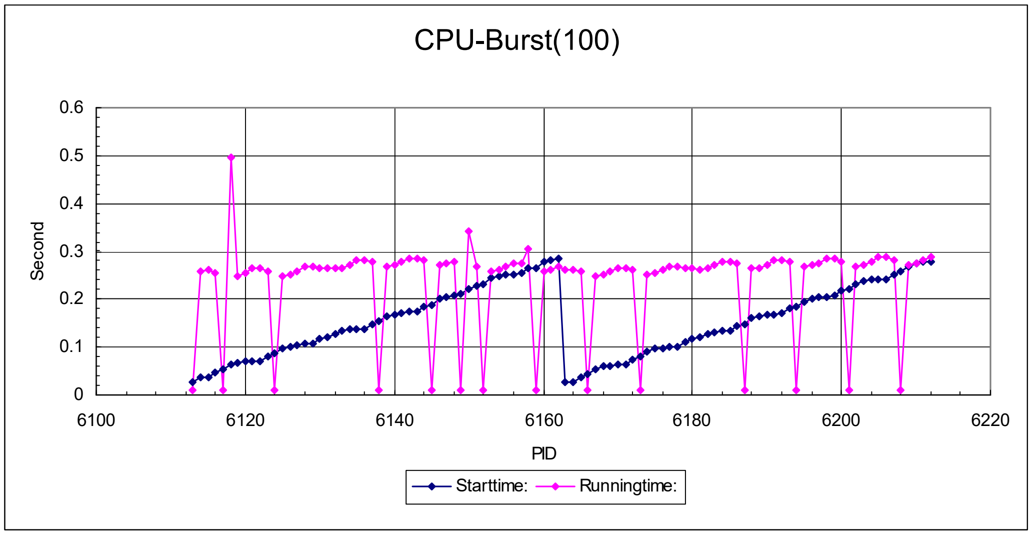

Figure 3-1-2

(the number in the Title is processes number)

Total Processes:	100	**Loop number:**	100,000
Average time:	0.235872	**Deviation:**	0.063085414
Max executing time:	0.495189	**Min executing time:**	0.010505

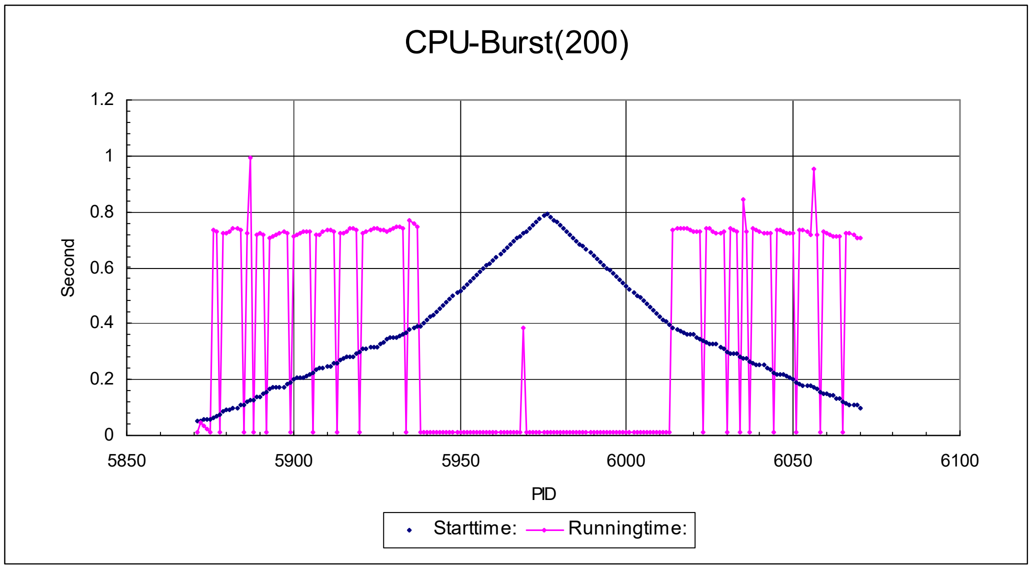

Figure 3-1-3

(the number in the Title is processes number)

Total Processes:	200	**Loop number:**	100,000
Average time:	0.382377	**Deviation:**	0.35989594
Max executing time:	0.994236	**Min executing time:**	0.0104818

Data of Kernel 2.4

Linux kernel version:

Linux Knoppix 2.4.47 #2 SMP Mo Aug 9 00:39:37 CEST 2004 i686 GNU/Linux

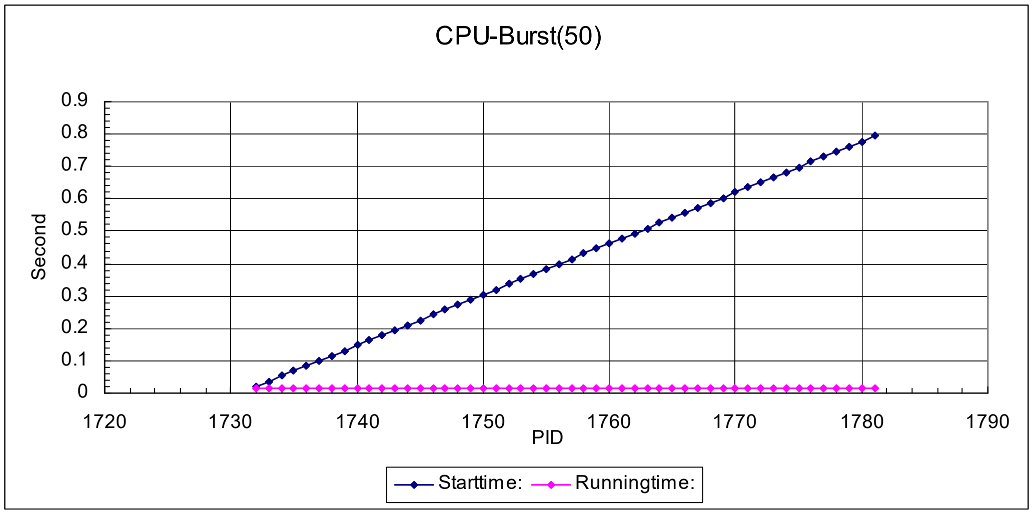

Figure 3-1-4

(the number in the Title is processes number)

Total Processes:	50	**Loop number:**	100,000
Average time:	0.0152406	**Deviation:**	4.81709E-05
Max executing time:	0.0159252	**Min executing time:**	0.0151999

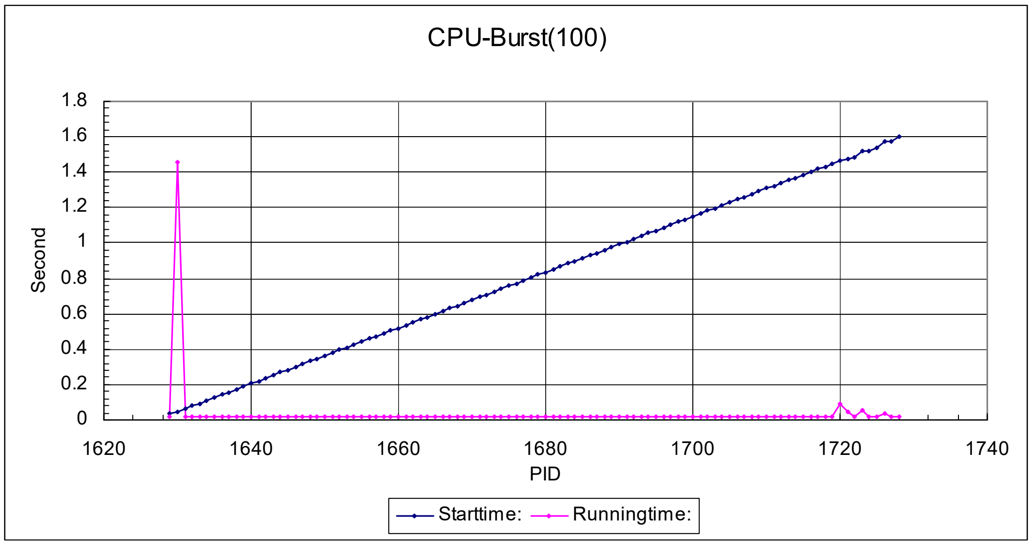

Figure 3-1-5

(the number in the Title is processes number)

Total Processes:	100	**Loop number:**	100,000
Average time:	0.0311988	**Deviation:**	0.030361663
Max executing time:	1.45535	**Min executing time:**	0.0151989

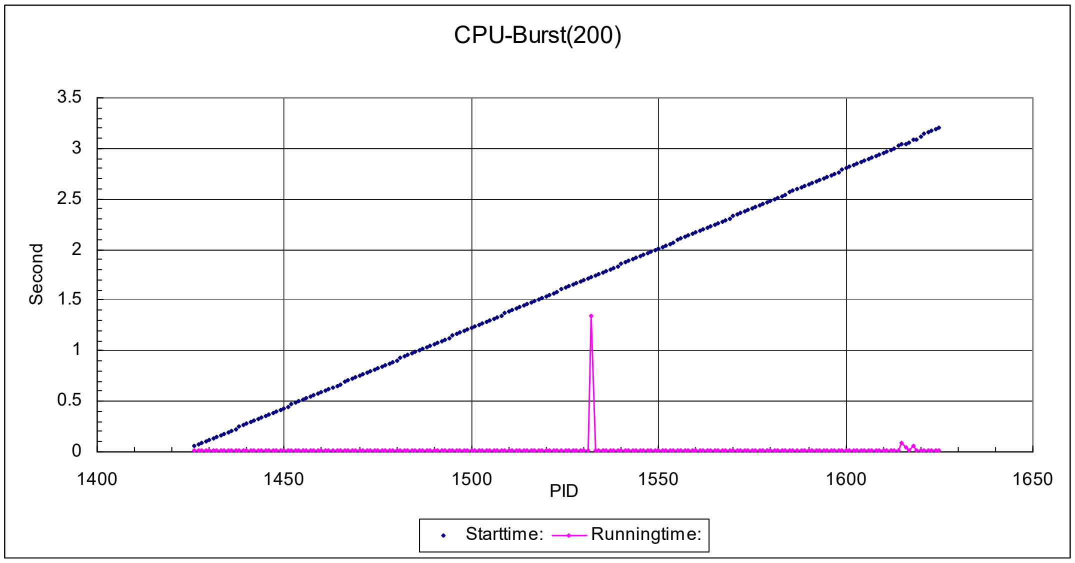

Figure 3-1-6

(the number in the Title is processes number)

Total Processes:	200	**Loop number:**	100,000
Average time:	0.0226109	**Deviation:**	0.014438323
Max executing time:	1.34559	**Min executing time:**	0.0151989

By compare the data and charts between Linux kernel 2.6 and kernel 2.4. It can be seen that the average running time of kernel 2.4 is quite shorter than kernel 2.6, however in kernel 2.4 the start time seen to be linearity with a big slop. In other words, though the running time is short, processes should wait a long time to be executed.

Look at the chart and the figure below:

Average run time			Average start time		Average stop time	
Process number:	**k 2.6**	**k 2.4**	**k 2.6**	**k 2.4**	**k 2.6**	**k 2.4**
50	0.106143	0.0152406	0.1124648	0.406844	0.2186082	0.4220846
100	0.235872	0.0311988	0.1548625	0.811306	0.390735	0.8425052
200	0.382377	0.0226109	0.3655435	1.625385	0.74792065	1.6479965
300	0.474293	0.0379512	0.583350133	2.443479	1.057642967	2.4814302

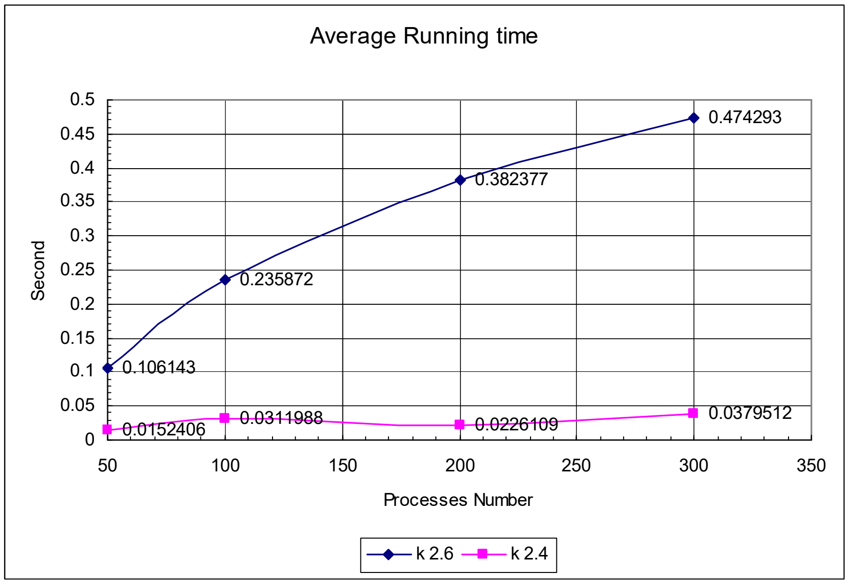

Figure 3-1-7 (k means kernel)

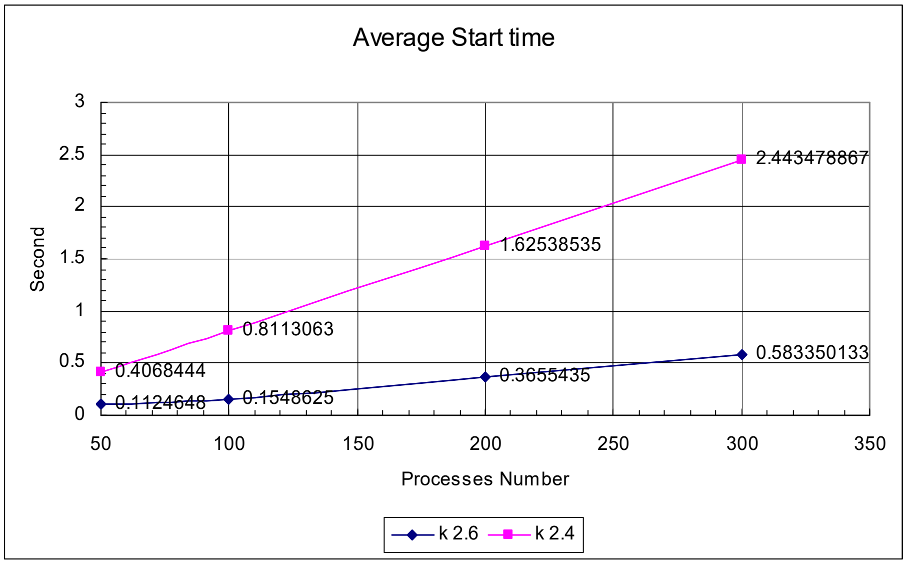

Figure 3-1-8 (k means kernel)

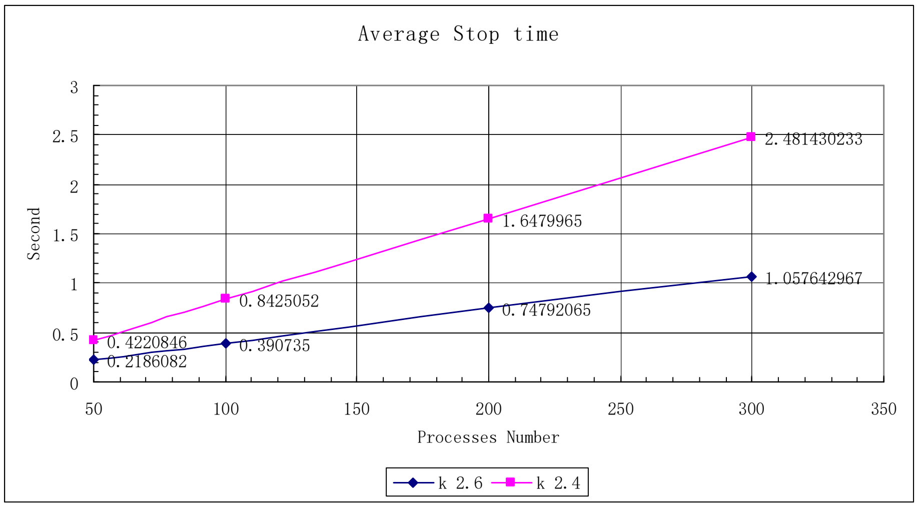

Figure 3-1-9 (k means kernel)

From those shown above kernel 2.6 optimize processes scheduling by using *O (1)*. Normally it seems that the time-lasting processes will be set a higher priority to let them be executed at first, meanwhile the shorter processes will be given a lower priority. It can be easily seen by comparing between start-time and running time and the gap shown on the figure that the operating system set priority to processes, especially with 200 processes' burst. It is also shown by the graphic when the processes number is 50 and 100 that the same priority processes are scheduled round-robin reciprocally. That why the start time looks like a mountain or saw-tooth alike and the running time are well-regulated without any big gap.

Nevertheless the schedule algorithm of kernel 2.4 is `O (n)`, which proved to be several weaknesses. We can see it from figure 3-1-1 to figure 3-1-6 or compare with the data and graphic of kernel 2.6 and kernel 2.4. The Linux 2.4.x scheduler executes in `O (n)` time, which means that the scheduling overhead on a system with many tasks can be dismal. During each call to `schedule ()`, every active task must be iterated over at least once in order for the scheduler to do its job. The obvious implication is that there are potentially frequent long periods of time when no "real" work is being done. Interactivity performance perception may suffer greatly from this. That's why the processes shown in these chats that has to wait much longer than kernel 2.6 (From the aspect of start-time), though the running time is much shorter. This is the aspect of Scalability.

The average timeslice assigned by the Linux 2.4.x scheduler is about 210ms.This is quite high (the average timeslice in the Linux 2.6.x scheduler is 100ms), and such large timeslices can cause the time between executions of low-priority tasks (or simply unlucky ones if all priorities are equal) to grow quite large. For example - with 100 threads using all of their 210ms timeslices without pause, the lowest priority thread in the group might have to wait more than 20 seconds before it executes. This is the result shown by figure 3-1-8 and figure 3-1-9.

3.2.2 I/O-Burst Test & Conclusion

Same kernel and machine like **CPU-Burst Test** are used in this example (Only a part of data shown below). Every process will write 400KB to disk.

Data of kernel 2.6

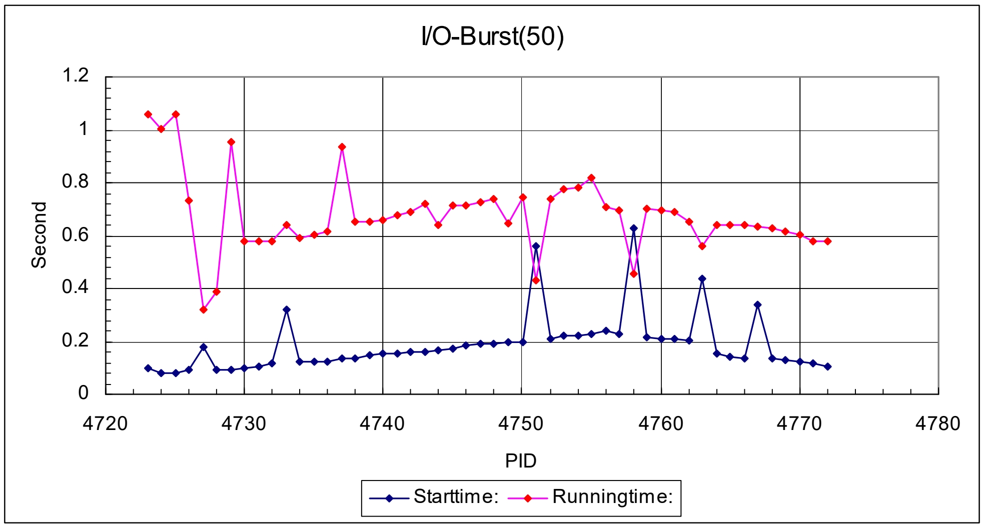

Figure 3-2-1

(the number in the Title is processes number)

Total Processes:	50	**Bytes number:**	400K
Average time:	0.67738	**Deviation:**	0.099668248
Max executing time:	1.05967	**Min executing time:**	0.318777

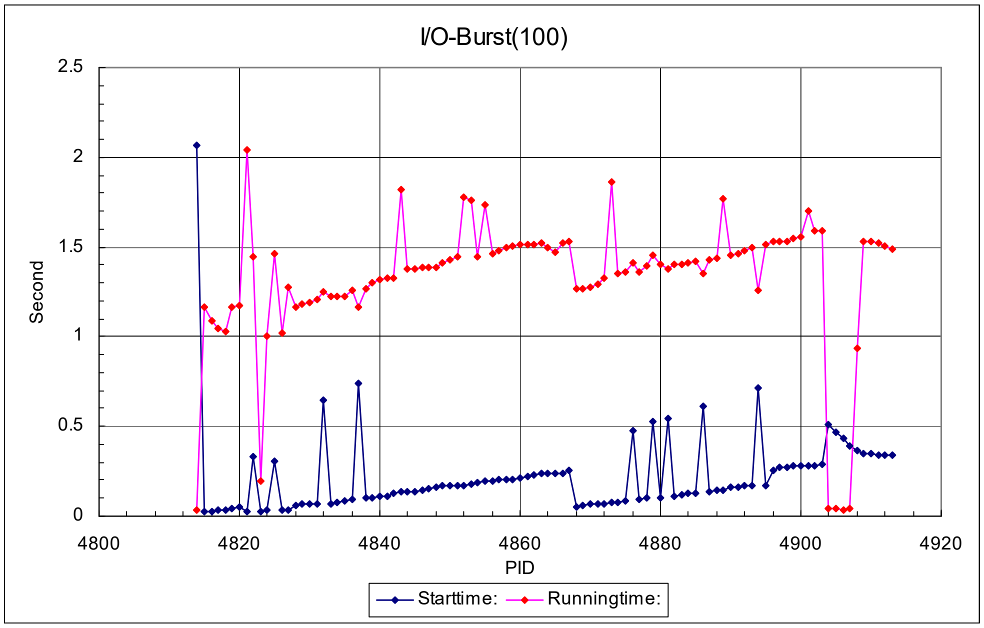

Figure 3-2-2

(the number in the Title is processes number)

Total Processes:	100	**Bytes number:**	400K
Average time:	1.32484	**Deviation:**	0.228167928
Max executing time:	2.03772	**Min executing time:**	0.0307438

Data of kernel 2.4

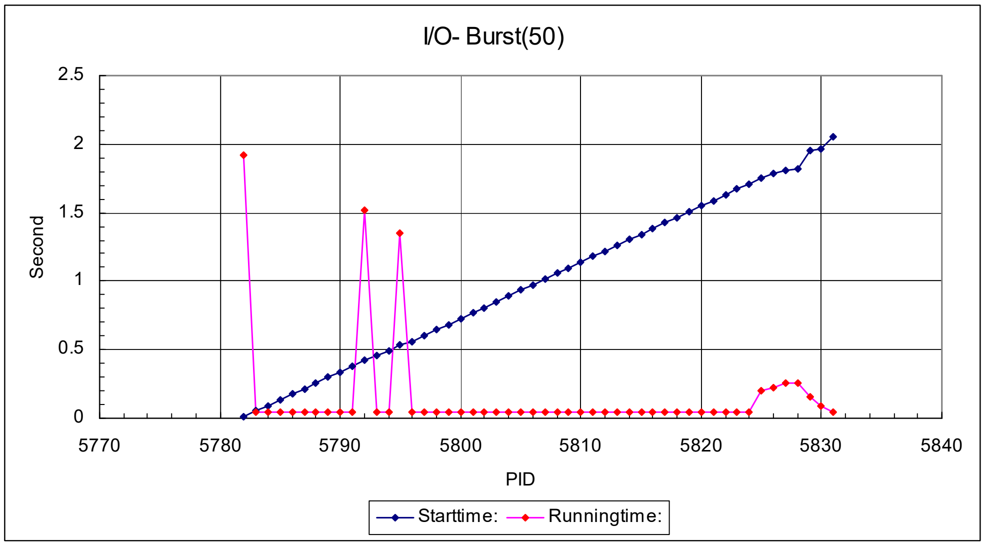

Figure 3-2-3

(the number in the Title is processes number)

Total Processes:	50	**Bytes number:**	400K
Average time:	0.15209	**Deviation:**	0.186206912
Max executing time:	1.9237	**Min executing time:**	0.0398982

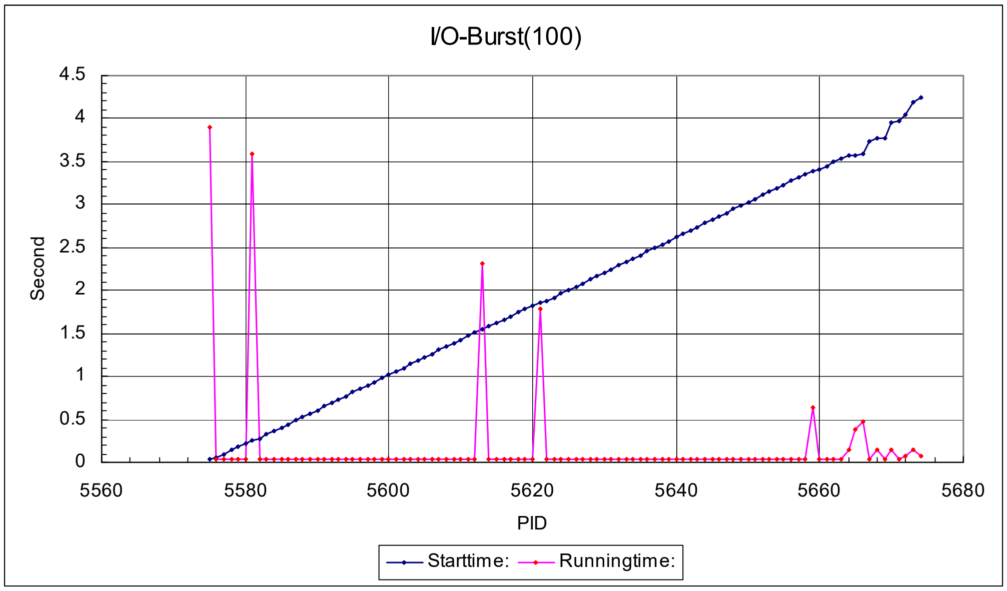

Figure 3-2-4

(the number in the Title is processes number)

Total Processes:	100	**Bytes number:**	400K
Average time:	0.172246	**Deviation:**	0.236838885
Max executing time:	3.89432	**Min executing time:**	0.0398591

From the data proved above. The difference of the data and figures between I/O and CPU is that I/O-Burst need much more running-time because CPU operation is faster than I/O and have to wait I/O operation and there are more peaks in the figure of both kernel 2.6 and kernel 2.4, it also due to the I/O operation towards Hardware. The running time of kernel 2.4 is much shorter than kernel 2.6 while the start-time of process is completely different. it can be further proved that kernel 2.6 used priority and RR is used when the processes' priority are the same.

The statistics blew will open out more information

Average run time			**Average start time**		**Average stop time**	
Processes Number:	**K2.6**	**K2.4**	**K2.6**	**K2.4**	**K2.6**	**K2.4**
50	0.67738	0.15209	0.185158	0.999758	0.862538	1.151848
100	1.32484	0.172246	0.219383	2.006158	1.544223	2.178405
200	1.93336	0.187511	0.774685	4.091754	2.708045	4.279265
300	2.56818	0.0963162	1.318406	6.215571	3.886584	6.311887

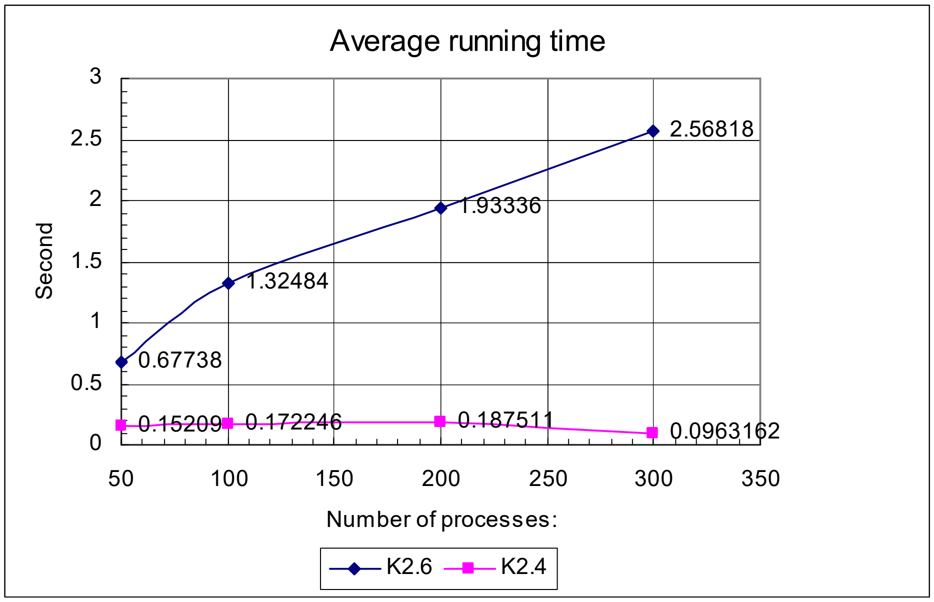

Figure 3-2-5 (k means kernel)

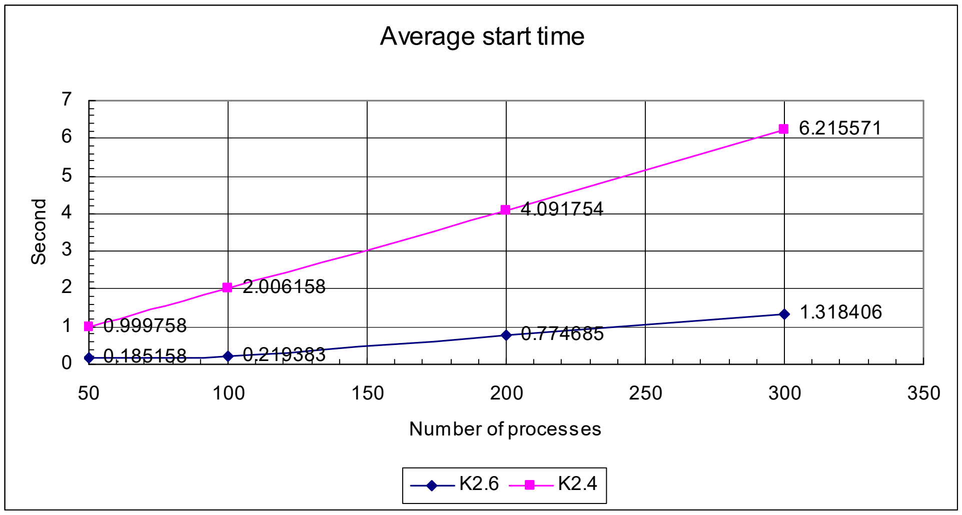

Figure 3-2-6 (k means kernel)

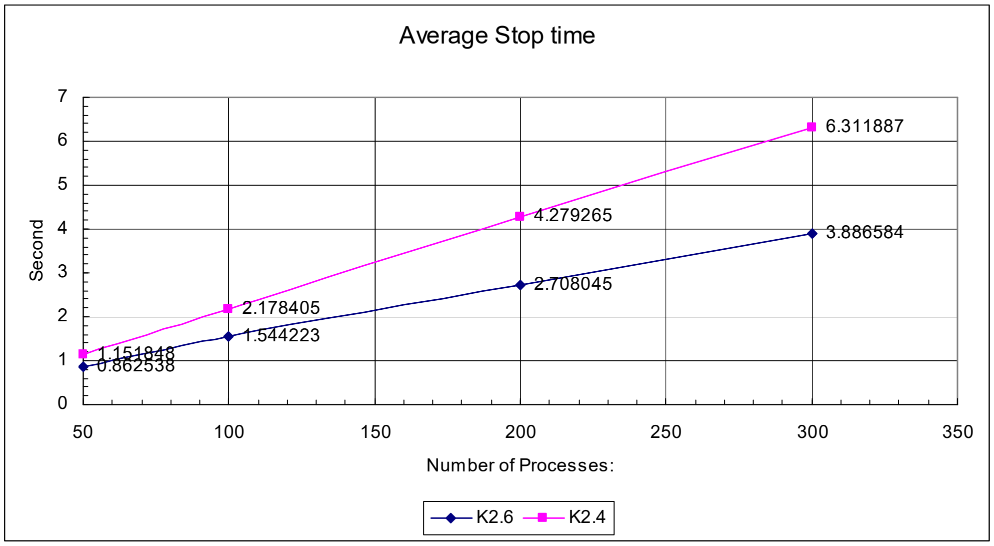

Figure 3-2-7 (k means kernel)

From the picture and chart, same conclusion can be drawn that O (1) is used to optimize the processes scheduling. Kernel 2.4 form the running time aspect is faster than kernel 2.6, but actually it spend more time than kernel 2.6 for a process starting execution, so that more time is needed in kernel to fulfill the Task—Burst.

The examples shown above just proved part of kernel 2.6 is superior to kernel 2.4. In fact there are 4 major flaws of kernel 2.4 when compare with kernel 2.6. There are scalability, large average timeslices, a less-than-optimal I/O-bound task priority boosting strategy, and weak RT-application support.

The scalability and large average timeslices have been covered in the example above.

Summary

Finally according to the data and figure provided upwards, in my personal opinion, the Linux kernel 2.6 is suitable for overhead and high performance or real-time application, while if demanding is not very rigorous or the quantity of task is not very large the Linux kernel is a good choice.

4 Works should be continue

The static analysis proved hereinbefore did not cove every aspect of comparison between Linux kernel 2.6 and kernel 2.4. So that the continue work should touch upon the RT-application support and the others. Due to the time limitation the Sequential condition (Processes are created and started one by one) is not tested, but the source code has been done and debugged. We can use this source code to compare windows with Linux. To let bash script be run in windows, Cygwin can be used to provide the environment. And the parallel condition's code about windows is not provided. In my opinion in order to get accurate comparison data the parallel condition should be done as well.

4.1 Final Note

All C++ source code and Bash Script code is provided in Appendix. The list shown below is reference.

4.2 Reference

[1] http://en.wikipedia.org/wiki/Scheduling_%28computing%29

[2] http://en.wikipedia.org/wiki/Process_states

[3] http://josh.trancesoftware.com/linux/

[4] C++ Primer, Fourth Edition by Stanley B. Lippman, Josée Lajoie, Barbara E. Moo

[5] http://www.infosys.com

[6] http://en.wikipedia.org/wiki/Signal_%28computing%29

[7] http://www.oreilly.com/catalog/linuxkernel/chapter/ch10.html

[8] http://www.cygwin.com/

[9] http://www.gnuplot.info/documentation.html

[10] http://fscked.org/writings/SHM/shm.html

[11] http://mathbits.com/MathBits/CompSci/Files/Sample2.htm

[12] http://www.codersource.net/cpp_file_io.html

[13] http://www.debian.org/doc/

[14] http://www.opengroup.org/onlinepubs/

[15] http://www.pathname.com/fhs/

[16] http://www.cs.jhu.edu/~yairamir/cs418/os2/sld001.htm

[17] Learning Debian GNU/Linux by Bill McCarty

[18] Using Linux by Bill Ball

Appendix A

Parallel condition code

CPU-BURST

The cpu_burst2.cpp

```
//Heads Declare

#include <unistd.h>

#include <iostream>

#include <stdlib.h>

#include <sys/wait.h>

#include <math.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <time.h>

#include <sys/sem.h>
```

```
#include <sys/time.h>

#include <math.h>

#include <iomanip>

#include <ios>

#include <signal.h>

//Definition

using namespace std;


//declare the function

void process_routine(long double *loop2);

int shm_status (int shm_id);

static void signal_sigusr1 (int signum) ;

static sighandler_t change_signalhandler (int signum, sighandler_t

signalhandler);


//define the structure

typedef struct timeval ourtime;

struct measure

{

   ourtime starttime;

   ourtime stoptime;

   pid_t pid;

};


//##MAIN FUNCTION##

int main (int argc, char ** argv)

{

   if (argc <=2 )

   {

      cerr << argv[0] << ": to less arguments!" <<endl;
```

```
        cerr << "Please use "<< argv[0] << " 10 100!"<<endl;
    }
    else
    {
        unsigned int loop1 = atoi (argv[1]);
        long double loop2 = atoi (argv[2]);
        unsigned int i;
        double reference_time,start_time,stop_time;
        //
        double
MaxRun,MinRun,AverageRunningTime,MaxStart,MinStart,MaxStop,MinStop;
        int MaxPID,MinPID;
        //
        pid_t  pid;
        // Shared Memory
        int shm_id, data;
        key_t key;
        measure *MT;      //the measure time!!!
        struct shmid_ds status;
        //change the Signalhandler
        change_signalhandler (SIGUSR1, signal_sigusr1);
        // create a shared memory with the array writable for all processes
        // create a new SHM segment (Array[0] is father)
        if ((shm_id = shmget(IPC_PRIVATE, sizeof(measure)*(loop1+1),
IPC_CREAT | SHM_R | SHM_W)) < 0)
        {
            cerr << "SHM sgement not created!" << endl;
            exit (-1);
        } else
```

```
       // ID of the segment
        cout << "SHM_ID: " << shm_id <<endl;
        shm_status(shm_id);
        cout<<endl;             //for Gnuplot to gernate Graphic
        cout<<endl;

       // attach a SHM segment
       MT = (measure *) shmat (shm_id, NULL, 0);
       if (MT == (void *) -1)
       {
           cerr <<  "SHM segment (%d) could not attached!" << shm_id <<
endl;
           exit (-1);
       }
       // entry 0 is reserved for the father - overall time
       MT[0].pid=getpid();
       gettimeofday(&MT[0].starttime,NULL);
//##Processes Gernation##
       cout << "##Generating all processes" << endl ;
       for (i=1;i<=loop1;i++)          //(Array[0] is father)
       {
           if (pid != 0)
           {
               pid=fork();
           }
           if (pid == -1)
           {
               cerr <<"cloud not fork" << endl;
               exit (-1);
           }
```

```
            if (pid == 0)
            {
                // connect to the shared memory
                measure *MT2;
                MT2 = (measure *) shmat (shm_id, NULL, 0);
                    if (MT2 == (void *) -1)
                    {
                    cerr <<" SHM segment could not attached!"<<
shm_id<<endl;
                    exit (-1);
                }
                MT2[i].pid=getpid();
                // wait for the start signal
                pause();
                // measure starts
                gettimeofday(&MT2[i].starttime,NULL);
                process_routine(&loop2);
                gettimeofday(&MT2[i].stoptime,NULL);
                // measure ends
                // all work is done - disconect to the shared meomry
                if (shmdt(MT2)<0)
                {
                    cerr << "SHM segment could not detached!\n" << shm_id
<< endl;
                }
                exit (0);
            }
        }
        // ready to start the measurement
        cout << "##--- START MEASUREMENT! ---"<< endl;
```

```
       sleep(2);                //to let all children to receive the
signal
       kill (0,SIGUSR1);
       while (wait(NULL)>=0);
       sleep(2);                //to make sure all signal were delived
       cout << "##--- FINISHED MEASUREMENT ---"<< endl;
       cout<<endl;              //for Gnuplot
       cout<<endl;
//##output of all information&Caculation##
       reference_time=(double) MT[0].starttime.tv_sec + (double)
MT[0].starttime.tv_usec/1e6;
       cout << "PID:\t\tStarttime:\tStoptime:\tRunningtime:"<<endl;
       MaxRun=0;
       MinRun=10000;
       AverageRunningTime=0;
       for (i=1;i<=loop1;i++)
       {
           start_time=(double) MT[i].starttime.tv_sec + (double)
MT[i].starttime.tv_usec/1e6;
           stop_time=(double) MT[i].stoptime.tv_sec + (double)
MT[i].stoptime.tv_usec/1e6 ;
           cout << MT[i].pid <<"\t\t"<<start_time-reference_time-2
           <<"\t\t"<< stop_time-reference_time -2<<"\t\t"<<
stop_time-start_time << endl;
           //Caculate the Average Running Time

   AverageRunningTime=AverageRunningTime+stop_time-start_time;
           if ((stop_time-start_time)<=MinRun)       //Find the Min
           {
               MinPID=MT[i].pid;
```

```
              MinStart=start_time-reference_time-2;

              MinStop=stop_time-reference_time-2;

              MinRun=stop_time-start_time;

          }

          if ((stop_time-start_time)>=MaxRun)       //Find the Max

          {

              MaxPID=MT[i].pid;

              MaxStart=start_time-reference_time-2;

              MaxStop=stop_time-reference_time-2;

              MaxRun=stop_time-start_time;

          }

      }

      AverageRunningTime=AverageRunningTime/(i-1);

      //display the MAX;MIN;AVERAGE

      cout<<endl;          //for gnuplot

      cout<<endl;

      //

      cout <<"##Average Running Time 
is:"<<"\t"<<AverageRunningTime<<endl;

      cout <<"##The Most Time-consuming Process is:"<<endl;

      cout<< "PID:\t\tStarttime:\tStoptime:\tRunningtime:"<<endl;


   cout<<MaxPID<<"\t\t"<<MaxStart<<"\t\t"<<MaxStop<<"\t\t"<<MaxRun<<
endl;

      cout <<"##The Lest Time-consuming Process is:"<<endl;

      cout<< "PID:\t\tStarttime:\tStoptime:\tRunningtime:"<<endl;


   cout<<MinPID<<"\t\t"<<MinStart<<"\t\t"<<MinStop<<"\t\t"<<MinRun<<
endl;

      // detach a SHM segment
```

```
        if (shmdt(MT)<0)

            fprintf (stderr,"SHM segment (%d) could not detached!\n",

shm_id);

        // delete shm segment

        if ((shmctl (shm_id, IPC_RMID, &status)) < 0)

            fprintf (stderr,"SHM segment (%d) could not deleted!\n",

shm_id);

    }

    exit (0);

}


//##Functions##

void process_routine(long double *loop2)

{

    unsigned int k;

    for (k=1; k<=*loop2; k++)

    {

        sqrt(k*(*loop2));

    }

}


int shm_status (int shm_id)

{

    struct shmid_ds status;

    if ((shmctl (shm_id, IPC_STAT, &status))==-1 )

    {

        fprintf (stderr,"SHM segment (%d) staus could determined!\n",

shm_id);

        return -1;

    }
    cout<<endl;
    printf ("segment status: %d\n",shm_id);
    printf ("pid of creator: %d\n",status.shm_cpid);
    printf ("pid of last operator: %d\n",status.shm_lpid);
    printf ("acces mode: %u\n",status.shm_perm.mode);
    printf ("size of segment (bytes): %d\n",status.shm_segsz);
    printf ("last attach time: %d
- %s",status.shm_atime,ctime(&status.shm_atime));
    printf ("last detach time: %d
- %s",status.shm_dtime,ctime(&status.shm_dtime));
    printf ("last change time: %d
- %s",status.shm_ctime,ctime(&status.shm_ctime));
    cout<<endl;
}

static void signal_sigusr1 (int signum)
{
    fprintf(stdout,"Process: %d is now running!\n",getpid());
}

static sighandler_t change_signalhandler (int signum, sighandler_t
signalhandler)
{
    struct sigaction mod_signal;

    //read prevoius signal action
    if (sigaction (signum, NULL, &mod_signal) < 0)
       return SIG_ERR;
    //modify sigaction structure values
    mod_signal.sa_handler = signalhandler;
    //add signal signum to the set
    sigaddset (&mod_signal.sa_mask, signum);

    //making certain system calls restartable across signals
    mod_signal.sa_flags = SA_RESTART;

    if (sigaction (signum, &mod_signal, NULL) < 0)
    {
        return SIG_ERR;
    }
    return mod_signal.sa_handler;
}
```

DoCPUfinal.sh

```
#!/bin/sh
#Get information:
echo -e " \t*********************This is for CPU-BURST
TESTING**********************"
echo -n "Input the number of processes you want to generate: "
read number
if [ "$number" = "" ]; then
    echo -n "argument missing!!!!!"
    exit 1
fi
echo -n "Please give the first number : "
read loop
if [ "$loop" = "" ]; then
    echo -n "argument missing!!!!!"
    exit 1
fi
echo -n -e " \t**************Press ENTER to Start*************"
read
echo -e " \t*****************Testing Start!****************"
./cpu-burst $number $loop  > cpuplot.csv
echo -e " \t*********************Done!!!*********************"
```

DrawGraphic.sh

```
#!/bin/sh
#GrawGraphic
gnuplot -persist << +endYeJieNi &
##save the graphic(PID & Running-Time)
set output "CPU_Burst1.png"
set terminal png
set title "CPU-Burst-TEST(2D)"
set xlabel "Processes ID [PID]"
set xtics 50
set ylabel "Run-Time [s]"
#set ytics 0.05
set grid
plot "./cpuplot.csv" index 2. using 1:4 title "Analysis" with impulses
+endYeJieNi
##Display the graphic
gnuplot -persist << +endYeJieNi &
set title "CPU-Burst-TEST(2D)"
set xlabel "Processes ID [PID]"
set xtics 50
set ylabel "Run-Time [s]"
```

```
#set ytics 0.05

set grid

plot "./cpuplot.csv" index 2. using 1:4 title "Analysis" with impulses

+endYeJieNi

#######################################

gnuplot -persist << +endYeJieNi &

##save the graphic(PID & Start-Time)

set output "CPU_Burst2.png"

set terminal png

set title "CPU-Burst-TEST(2D)"

set xlabel "Processes ID [PID]"

set xtics 50

set ylabel "Start-Time [s]"

#set ytics 0.05

set grid

plot "./cpuplot.csv" index 2. using 1:2 title "Analysis" with impulses

+endYeJieNi

##Display the graphic

gnuplot -persist << +endYeJieNi &

set title "CPU-Burst-TEST(2D)"

set xlabel "Processes ID [PID]"

set ylabel "Start-Time [s]"

set xtics 50

#set ytics 0.05

set grid

plot "./cpuplot.csv" index 2. using 1:2 title "Analysis" with impulses

+endYeJieNi

#Save the Data File

FILENAME=`date +%Y%m%d-%H%M%S`_cpu-burst.csv

uname -a >> $FILENAME
```

```
date >> $FILENAME

echo -e "PID:\tStarttime:\tStoptime:\tRunn-time:" >> $FILENAME

cat   ./cpuplot.csv >> $FILENAME

cat $FILENAME

rm -rf cpuplot.csv
```

I/O-BURST

The io_burst.cpp

```
//Heads Declare

#include <stdio.h>

#include <unistd.h>

#include <iostream>

#include <stdlib.h>

#include <sys/wait.h>

#include <math.h>

#include <sys/types.h>

#include <sys/ipc.h>

#include <sys/shm.h>

#include <time.h>

#include <sys/sem.h>

#include <sys/time.h>

#include <math.h>

#include <iomanip>

#include <ios>

#include <fstream>

#include <string>

#include <signal.h>
```

```
//Definition
using namespace std;

//declare the function
std::string itoa(int value, unsigned int base) ;
int shm_status (int shm_id);
static void signal_sigusr1 (int signum) ;
static sighandler_t change_signalhandler (int signum, sighandler_t
signalhandler);

//define the structure
typedef struct timeval ourtime;
struct measure
{
   ourtime starttime;
   ourtime stoptime;
   pid_t pid;
};

//##MAIN FUNCTION##
int main (int argc, char ** argv)
{
   if (argc <=3 )
   {
      cerr << argv[0] << ": to less arguments! Please input 3 numbers!!"
<<endl;
   }
   else
   {
      unsigned int loop1 = atoi (argv[1]);
      long int ByteNumber = atoi (argv[2])*1024;
      int sel=atoi(argv[3]);// 1 means read;2 means write
      unsigned int i;
      double reference_time,start_time,stop_time;
      ifstream readfile;
      ofstream writefile;
      //
      double
MaxRun,MinRun,AverageRunningTime,MaxStart,MinStart,MaxStop,MinStop;
      int MaxPID,MinPID;
      //
      pid_t  pid;
      // Shared Memory
      int shm_id, data;
      key_t key;
      measure *MT;      //the measure time!!!
      struct shmid_ds status;

      //change the Signalhandler
      change_signalhandler (SIGUSR1, signal_sigusr1);
      // create a shared memory with the array writable for all processes
      // create a new SHM segment (Array[0] is father)
      if ((shm_id = shmget(IPC_PRIVATE, sizeof(measure)*(loop1+1),
IPC_CREAT | SHM_R | SHM_W)) < 0)
      {
         cerr << "SHM sgement not created!" << endl;
         exit (-1);
      } else
```

```
       // ID of the segment
        cout << "SHM_ID: " << shm_id <<endl;
        shm_status(shm_id);
        cout<<endl;//for Gnuplot to gernate Graphic
        cout<<endl;

       // attach a SHM segment
       MT = (measure *) shmat (shm_id, NULL, 0);
       if (MT == (void *) -1)
       {
           cerr <<  "SHM segment (%d) could not attached!" << shm_id <<
endl;
           exit (-1);
       }
       // entry 0 is reserved for the father - overall time
       MT[0].pid=getpid();
       gettimeofday(&MT[0].starttime,NULL);
//##Processes Gernation##
       cout << "##Generating all processes" << endl ;
       for (i=1;i<=loop1;i++)          //(Array[0] is father)
       {
           if (pid != 0)
           {
               pid=fork();
           }
           if (pid == -1)
           {
               cerr <<"cloud not fork" << endl;
               exit (-1);
           }
```

```
          if (pid == 0)
          {
             // connect to the shared memory
             measure *MT2;     //children processes!alias
             MT2 = (measure *) shmat (shm_id, NULL, 0);
                if (MT2 == (void *) -1)
                {
                cerr <<" SHM segment could not attached!"<<
shm_id<<endl;
                exit (-1);
             }
             MT2[i].pid=getpid();
             // wait for the start signal
             pause();
             // measure starts
             //operation option
             if (sel==1)
             {
                sync();
                string file=( "test-" );
                string pid =string ( itoa(i,10) );
                string ext = ( ".txt\0" );
                string name = file+pid+ext;
                readfile.open( name.c_str(), ifstream::out |
ifstream::trunc );
                if (!readfile)
                {
                   cerr<<"Can't open the file! please use argument 3
option 2 to create file!!";
                   exit(-1);
```

```
                    }
                    char *block;
                    block= new char [ByteNumber*4];
                    int j;
                    gettimeofday(&MT2[i].starttime,NULL);
                    {
                        readfile.read(block,sizeof(name.c_str()));
                    }
                    gettimeofday(&MT2[i].stoptime,NULL);
                    readfile.close();
                }
                else if (sel==2)
                {
                    string file=( "test-" );
                    string pid =string ( itoa(i,10) );
                    string ext = ( ".txt\0" );
                    string name = file+pid+ext;
                    writefile.open( name.c_str(), ofstream::out |
ofstream::trunc );
                    if (!writefile)
                    {
                        cerr<<"Can't open the file!"<<endl;
                        exit(-1);
                    }
                    int j;
                    gettimeofday(&MT2[i].starttime,NULL);
                    for (j=0;j<ByteNumber;j++)
                    {
                        writefile<<"Jack";
                    }
                    gettimeofday(&MT2[i].stoptime,NULL);
                    writefile.close();
                }
                else
                {
                    cerr << "Incorrect argument!!!Please input 1 or 2
(1=read;2=write)" << endl;
                    exit (-1);
                }
                // measure ends
                // all work is done - disconect to the shared meomry
                if (shmdt(MT2)<0)
                {
                    cerr << "SHM segment could not detached!\n" << shm_id
<< endl;
                }
                exit (-1);
            }
        }
        // ready to start the measurement
        cout << "##--- START MEASUREMENT! ---"<< endl;
        sleep(2);                //to let all children to receive the
signal
        kill (0,SIGUSR1);
        while (wait(NULL)>=0);
        sleep(2);                //to make sure all signal were delived
        cout << "##--- FINISHED MEASUREMENT ---"<< endl;
        cout<<endl;//for Gnuplot
        cout<<endl;
//##output of all information&Caculation##
```

```
       reference_time=(double) MT[0].starttime.tv_sec + (double)
MT[0].starttime.tv_usec/1e6;
       cout << "PID:\t\tStarttime:\tStoptime:\tRunningtime:"<<endl;
       MaxRun=0;
       MinRun=10000;
       AverageRunningTime=0;
       for (i=1;i<=loop1;i++)
       {
           start_time=(double) MT[i].starttime.tv_sec + (double)
MT[i].starttime.tv_usec/1e6;
           stop_time=(double) MT[i].stoptime.tv_sec + (double)
MT[i].stoptime.tv_usec/1e6 ;
           cout << MT[i].pid <<"\t\t"<<start_time-reference_time-2
           <<"\t\t"<< stop_time-reference_time -2<<"\t\t"<<
stop_time-start_time << endl;
           //Caculate the Average Running Time

   AverageRunningTime=AverageRunningTime+stop_time-start_time;
           if ((stop_time-start_time)<=MinRun)       //Find the Min
           {
               MinPID=MT[i].pid;
               MinStart=start_time-reference_time-2;
               MinStop=stop_time-reference_time-2;
               MinRun=stop_time-start_time;
           }
           if ((stop_time-start_time)>=MaxRun)       //Find the Max
           {
               MaxPID=MT[i].pid;
               MaxStart=start_time-reference_time-2;
               MaxStop=stop_time-reference_time-2;
```

```
              MaxRun=stop_time-start_time;

           }

       }

       AverageRunningTime=AverageRunningTime/(i-1);

       //display the MAX;MIN;AVERAGE

       cout<<endl;                            //for gnuplot

       cout<<endl;

       cout <<"##Average Running Time
is:"<<"\t"<<AverageRunningTime<<endl;

       cout <<"##The Most Time-consuming Process is:"<<endl;

       cout<< "PID:\t\tStarttime:\tStoptime:\tRunningtime:"<<endl;


    cout<<MaxPID<<"\t\t"<<MaxStart<<"\t\t"<<MaxStop<<"\t\t"<<MaxRun<<
endl;

       cout <<"##The Lest Time-consuming Process is:"<<endl;

       cout<< "PID:\t\tStarttime:\tStoptime:\tRunningtime:"<<endl;


    cout<<MinPID<<"\t\t"<<MinStart<<"\t\t"<<MinStop<<"\t\t"<<MinRun<<
endl;

       // detach a SHM segment

       if (shmdt(MT)<0)

           fprintf (stderr,"SHM segment (%d) could not detached!\n",
shm_id);

       // delete shm segment

       if ((shmctl (shm_id, IPC_RMID, &status)) < 0)

           fprintf (stderr,"SHM segment (%d) could not deleted!\n",
shm_id);

    }

    exit (0);

}
```

```
//##Functions##

int shm_status (int shm_id)

{

    struct shmid_ds status;

    if ((shmctl (shm_id, IPC_STAT, &status))==-1 )

    {

        fprintf (stderr,"SHM segment (%d) staus could determined!\n",
shm_id);

        return -1;

    }

    cout<<endl;

    printf ("segment status: %d\n",shm_id);

    printf ("pid of creator: %d\n",status.shm_cpid);

    printf ("pid of last operator: %d\n",status.shm_lpid);

    printf ("acces mode: %u\n",status.shm_perm.mode);

    printf ("size of segment (bytes): %d\n",status.shm_segsz);

    printf ("last attach time: %d
- %s",status.shm_atime,ctime(&status.shm_atime));

    printf ("last detach time: %d
- %s",status.shm_dtime,ctime(&status.shm_dtime));

    printf ("last change time: %d
- %s",status.shm_ctime,ctime(&status.shm_ctime));

    cout<<endl;

}


static void signal_sigusr1 (int signum)

{

    fprintf(stdout,"Process: %d is now running!\n",getpid());

}
```

```
static sighandler_t change_signalhandler (int signum, sighandler_t
signalhandler)
{
    struct sigaction mod_signal;
    //read prevoius signal action
    if (sigaction (signum, NULL, &mod_signal) < 0)
       return SIG_ERR;
    //modify sigaction structure values
    mod_signal.sa_handler = signalhandler;
    //add signal signum to the set
    sigaddset (&mod_signal.sa_mask, signum);
    //making certain system calls restartable across signals
    mod_signal.sa_flags = SA_RESTART;
    if (sigaction (signum, &mod_signal, NULL) < 0)
    {
        return SIG_ERR;
    }
    return mod_signal.sa_handler;
}

// C++ version std::string style "itoa":
std::string itoa(int value, unsigned int base)
{
    const char digitMap[] = "0123456789abcdef";
    std::string buf;
    // Guard:
    if (base == 0 || base > 16) {
        // Error: may add more trace/log output here
        return buf;
    }
    // Take care of negative int:
    std::string sign;
    int _value = value;
    // Check for case when input is zero:
    if (_value == 0)
    {
        return "0";
    }
    if (value < 0)
    {
        _value = -value;
        sign = "-";
    }
    // Translating number to string with base:
    for (int i = 30; _value && i ; --i)
    {
        buf = digitMap[ _value % base ] + buf;
        _value /= base;
    }
    return sign.append(buf);
}
```

DoIOfinal.sh

```
#!/bin/sh
#Get information:
echo -e " \t*******************This is for IO-BURST
TESTING*********************"
echo -n "Input the number of processes you want to generate: "
```

```
read number
if [ "$number" = "" ]; then
    echo -n "argument missing!!!!!"
    exit 1
fi
echo -n "Please give the Byte number you want to read or write(1KB*4):
"
read Bytes
if [ "$Bytes" = "" ]; then
    echo -n "argument missing!!!!!"
    exit 1
fi
echo -n "Please give the option (1: read; 2 :write): "
read arg
if [ "$arg" = "" ];  then
    echo-n  "argument missing!!!!!"
    exit 1
fi
echo -n -e " \t*************Press ENTER to Start*************"
read
echo -e " \t****************Testing Start!****************"
./io-burst $number $Bytes $arg >> ioplot.csv
echo -e " \t*******************Done!!!********************"
```

DrawDraphic-IO.sh

```
#!/bin/sh
#GrawGraphic
gnuplot -persist << +endYeJieNi &
##save the graphic(PID & Running-Time)
```

```
set output "IO_Burst1.png"

set terminal png

set title "IO-Burst-TEST(2D)"

set xlabel "Processes ID [PID]"

set ylabel "Run-Time [s]"

set grid

plot "./ioplot.csv" index 2. using 1:4 title "Analysis" with impulses

+endYeJieNi

##Display the graphic

gnuplot -persist << +endYeJieNi &

set title "IO-Burst-TEST(2D)"

set xlabel "Processes ID [PID]"

set ylabel "Run-Time [s]"

set grid

plot "./ioplot.csv" index 2. using 1:4 title "Analysis" with impulses

+endYeJieNi

#######################################

gnuplot -persist << +endYeJieNi &

##save the graphic(PID & Start-Time)

set output "IO_Burst2.png"

set terminal png

set title "IO-Burst-TEST(2D)"

set xlabel "Processes ID [PID]"

set ylabel "Start-Time [s]"

set grid

plot "./ioplot.csv" index 2. using 1:2 title "Analysis" with impulses

+endYeJieNi

##Display the graphic

gnuplot -persist << +endYeJieNi &

set title "IO-Burst-TEST(2D)"
set xlabel "Processes ID [PID]"

set ylabel "Start-Time [s]"

set grid

plot "./ioplot.csv" index 2. using 1:2 title "Analysis" with impulses

+endYeJieNi

#Save the Data File

FILENAME=`date +%Y%m%d-%H%M%S`_io-burst.csv

uname -a >> $FILENAME

date >> $FILENAME

echo -e "PID:\tStarttime:\tStoptime:\tRunn-time:" >> $FILENAME

cat   ioplot.csv >> $FILENAME

cat $FILENAME

rm -rf ioplot.csv
```

Appendix B

Sequential condition

CPU-Burst

getreference_time.cpp

```
#include <sys/time.h>
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include <iomanip>
#include <ios>

using namespace std;using std::endl;
using std::setw;
typedef struct timeval our_time;
// main
int main(int argc, char *argv[])
{
   our_time time;
   gettimeofday(&time,NULL);
   unsigned long int reference_time=time.tv_sec;
```

```
    std::cout.setf (ios::fixed);
                     std::cout<<reference_time<<endl;
    return 0;
}
```

CPU-BURST.cpp

```
#include <sys/time.h>
#include <iostream>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <iomanip>
#include <ios>

long int getstart_time();                    //Dsiplay the start time
long int getstop_time();                     //Display the stop time
void normal_routine(long double *loop2);     //CPU-burst test

using namespace std;using std::endl;
using std::setw;

typedef struct timeval our_time;

// main
int main(int argc, char *argv[])
{
    if (argc<=3)
    {
       cerr<<"Argument missing! You must input 2 number devided by 
space!"<<endl;
       cerr<<"nomally the 2 number should be within 10000,"
       <<" because it'll take such a long time to execuate if the number 
is so big"
       <<endl;
       exit(-1);
    }
    else
    {

//Define the Varibale"
       our_time starttime,stoptime;
       unsigned  int i,loop1;
       double start_time,stop_time;
       long double loop2,reference_time;

       loop1=atoi(argv[1]);
       loop2=atoi(argv[2]);
       reference_time=atoi(argv[3]);
//begin:
       gettimeofday(&starttime,NULL);            //Get the start time
       for (i=0;i<=loop1;i++)
       {
          normal_routine(&loop2);
       }
       gettimeofday(&stoptime,NULL);             //Get the stop time

//Output the Value:
       start_time=(double) starttime.tv_sec + (double)
starttime.tv_usec/1e6;
       stop_time=(double) stoptime.tv_sec + (double)
stoptime.tv_usec/1e6 ;

       std::cout.setf (ios::fixed);                 //Set the Format
       std::cout <<getpid()<<"\t"<<(start_time-reference_time)<<"\t"
       <<(stop_time-reference_time)<<"\t"<< (stop_time-start_time)
<<endl;
    }
    return 0;
}

//test function:
void normal_routine( long double *loop2)
{
    unsigned int k;
    for (k=1; k<=*loop2; k++)
    {
       sqrt(k*(*loop2));
    }
}
```

cpu-burst-sechduling(2D).sh

```
#!/bin/sh

#Get information:

echo -e " \t*******************This is for CPU-BURST
```

```
TESTING*********************"
echo -n "Input the number of processes you want to generate: "
read number
if [ "$number" = "" ]; then
   echo -n "argument missing!!!!!"
   exit 1
fi
echo -n "Please give the number of loop1: "
read loop1
if [ "$loop1" = "" ]; then
   echo -n "argument missing!!!!!"
   exit 1
fi
echo -n "Please give the number of loop2: "
read loop2
if [ "$loop2" = "" ]; then
   echo -n "argument missing!!!!!"
   exit 1
fi
reference_time=`./gettime`
echo -n -e " \t*************Press ENTER to Start*************"
read
echo -e " \t****************Testing Start!***************"
{
for (( i=1; i<=$number; i=i+1 ))
do
   echo -e "Process: $i \tExecuating ......!" &
   #echo -n "$i ">>plot.data
   ./cpu-burst $loop1 $loop2 $reference_time  >> plot.data
done
}
wait 2
echo -e " \t********************Done!!!********************"
########################################
gnuplot -persist << +endYeJieNi &
##save the graphic(PID & Running-Time)
set output "CPU_Burst1.png"
set terminal png
set title "CPU-Burst-TEST(2D)"
set xlabel "Processes ID [PID]"
set ylabel "Run-Time [s]"
set ytics 0.2
set yrange [0:2]
set grid
```

```
plot "./plot.data" using 1:4 title "Analysis" with impulses
+endYeJieNi
##Display the graphic
gnuplot -persist << +endYeJieNi &
set title "CPU-Burst-TEST(2D)"
set xlabel "Processes ID [PID]"
set ylabel "Run-Time [s]"
set ytics 0.2
set yrange [0:2]
set grid
plot "./plot.data" using 1:4 title "Analysis" with impulses
+endYeJieNi
#######################################
gnuplot -persist << +endYeJieNi &
##save the graphic(PID & Start-Time)
set output "CPU_Burst2.png"
set terminal png
set title "CPU-Burst-TEST(2D)"
set xlabel "Processes ID [PID]"
set ylabel "Start-Time [s]"
set grid
plot "./plot.data" using 1:2 title "Analysis" with impulses
+endYeJieNi
##Display the graphic
gnuplot -persist << +endYeJieNi &
set title "CPU-Burst-TEST(2D)"
set xlabel "Processes ID [PID]"
set ylabel "Start-Time [s]"
set ytics 0.5
set grid
plot "./plot.data" using 1:2 title "Analysis" with impulses
+endYeJieNi
#Save the Data File
FILENAME=`date +%Y%m%d-%H%M%S`_cpu-burst.csv
uname -a >> $FILENAME
date >> $FILENAME
echo -e "PID\tStarttime\tStoptime\tRunn-time" >> $FILENAME
cat  plot.data >> $FILENAME
cat $FILENAME
rm -rf plot.data
```

I/O-Burst

getreference_time.cpp

```
#include <sys/time.h>
#include <iostream>
#include <stdlib.h>
#include <time.h>
#include <iomanip>
#include <ios>

using namespace std;using std::endl;
using std::setw;
typedef struct timeval our_time;
// main
int main(int argc, char *argv[])
{
   our_time time;
   gettimeofday(&time,NULL);
   unsigned long int reference_time=time.tv_sec;
   std::cout.setf (ios::fixed);
   std::cout<<reference_time<<endl;
   return 0;
}
```

I/O-BURST.cpp

```
#include <sys/time.h>
#include <iostream>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <iomanip>
#include <ios>
#include <fstream>
#include <string>

void normal_IOw(long int *loop2);
void normal_IOr(long int *loop2);
   //IO-burst test
using namespace std;
typedef struct timeval our_time;
ofstream wfile;
ifstream rfile;
// main
int main(int argc, char *argv[])
{
   if (argc<=4)
   {
       cerr<<"Argument missing! You must input 2 number devided by 
space!"<<endl;
       cerr<<"nomally the 2 number should be within 10000,"
       <<" because it'll take such a long time to execuate if the number 
is so big"
       <<endl;
       exit(-1);
   }
   else
   {
       //Creat or Open a file
       //Define the Varibale"
       our_time starttime,stoptime;
       unsigned  long int i,loop1,option;
       long int loop2;
       double start_time,stop_time;
       long double reference_time;
       loop1=atoi(argv[1]);
       loop2=atoi(argv[2]);
       option=atoi(argv[3]);
       reference_time=atoi(argv[4]);
//begin:
       if (option==1)//read
       {
           rfile.open ("test.txt",ios::in);
           gettimeofday(&starttime,NULL);         //Get the start time
           for (i=0;i<=loop1;i++)
           {
              normal_IOr(&loop2);
           }
           gettimeofday(&stoptime,NULL);

       }
       else if (option==2)//write
       {
           wfile.open ("test.txt",ios::out| ios::app);
           gettimeofday(&starttime,NULL);         //Get the start time
           for (i=0;i<=loop1;i++)
           {
              normal_IOw(&loop2);
           }
           gettimeofday(&stoptime,NULL);
       }
       else
       {
           cerr<<"Please input the option 1 or 2. 1=read;2=write!"<<endl;
           exit(-1);
       }

//Output the Value:
       start_time=(double) starttime.tv_sec + (double)
starttime.tv_usec/1e6;
       stop_time=(double) stoptime.tv_sec + (double)
stoptime.tv_usec/1e6 ;

       std::cout.setf (ios::fixed);                //Set the Format
       std::cout <<getpid()<<"\t"<<(start_time-reference_time)<<"\t"
       <<(stop_time-reference_time)<<"\t"<< (stop_time-start_time)
<<endl;
   }
   return 0;
}

//test function:
void normal_IOw( long int *loop2)
{
   int i;
   for (i=0;i<=(*loop2);i++)
   {
       wfile<<"Jack";
   }
   wfile.close();
}
void normal_IOr(long int *loop2)
{
   char *block;
   block= new char [int(*loop2)];
   int i;
   for (i=0;i<=(*loop2);i++)
   {
       rfile.read(block,*loop2);
   }
   rfile.close();
}
```

IO-burst-sechduling(2D).sh

```
#!/bin/sh

#Get information:

echo -e " \t*******************This is for CPU-BURST
TESTING********************"
echo -n "Input the number of processes you want to generate: "
read number
if [ "$number" = "" ]; then
    echo -n "argument missing!!!!!"
    exit 1
fi
echo -n "Please give the first number : "
read loop1
if [ "$loop1" = "" ]; then
    echo -n "argument missing!!!!!"
    exit 1
fi
echo -n "Please give the second number: "
read loop2
if [ "$loop2" = "" ]; then
    echo -n "argument missing!!!!!"
    exit 1
fi
echo -n "Please give the operator 1:means read while 2: means write: "
read operator
if [ "$operator" = "" ]; then
    echo -n "argument missing!!!!!"
    exit 1
fi
reference_time=`./gettime`
echo -n -e " \t**************Press ENTER to Start*************"
read
echo -e " \t*****************Testing Start!****************"
{
for (( i=1; i<=$number; i=i+1 ))
do
    echo -e "Process: $i \tExecuating ......!$$" &
    ./io-burst $loop1 $loop2 $operator $reference_time  >> ioplot.data &
done
} ####it gernate the processes at background
#wait 2
echo -e " \t********************Done!!!********************"
```

```
########################################
gnuplot -persist << +endYeJieNi &
##save the graphic(PID & Running-Time)
set output "IO_Burst1.png"
set terminal png
set title "I/O-Burst-TEST(2D)"
set xlabel "Processes ID [PID]"
set ylabel "Run-Time [s]"
set grid
plot "./ioplot.data" using 1:4 title "Analysis" with impulses
+endYeJieNi
##Display the graphic
gnuplot -persist << +endYeJieNi &
set title "I/O-Burst-TEST(2D)"
set xlabel "Processes ID [PID]"
set ylabel "Run-Time [s]"
set grid
plot "./ioplot.data" using 1:4 title "Analysis" with impulses
+endYeJieNi
#######################################
gnuplot -persist << +endYeJieNi &
##save the graphic(PID & Start-Time)
set output "IO_Burst2.png"
set terminal png
set title "I/O-Burst-TEST(2D)"
set xlabel "Processes ID [PID]"
set ylabel "Start-Time [s]"
set grid
plot "./ioplot.data" using 1:2 title "Analysis" with impulses
+endYeJieNi
##Display the graphic
gnuplot -persist << +endYeJieNi &
set title "I/O-Burst-TEST(2D)"
set xlabel "Processes ID [PID]"
set ylabel "Start-Time [s]"
set grid
plot "./ioplot.data" using 1:2 title "Analysis" with impulses
+endYeJieNi
#Save the Data File
FILENAME=`date +%Y%m%d-%H%M%S`_io-burst.csv
uname -a >> $FILENAME
date >> $FILENAME
echo -e "PID:\tStarttime:\tStoptime:\tRunn-time:" >> $FILENAME
cat  ioplot.data >> $FILENAME
cat $FILENAME
rm -rf ioplot.data
```