

# Paging Activity Tracing

July 17, 2002

## Abstract

The Memory Management (MM) sub-system is the most important parts of the operating system. This paper presents a *Swapping Tracing System* to view the memory activity in real time by changing the kernel code with a separate visualization tool. The system provides *Physical Memory View*, *Process View* and *Sector view* to inspect the page activity exactly in overall respective, which can help administrator analyze the system bottleneck or monitor the rush hour of swapping activity. With some statistic analyses, it can also help us to design the special swapping algorithm for the special application.

This paper also proposes the property view for each virtual memory page, physical memory page, or sector page in hard disk. These information can be known, such as what is process attached with this page, what is current status of this page. Especially, you can fetch the last ten activities information of a specific process to support detail analysis on the special virtual memory of a special process. It can demo the activity of special swap algorithm, such as LRU.

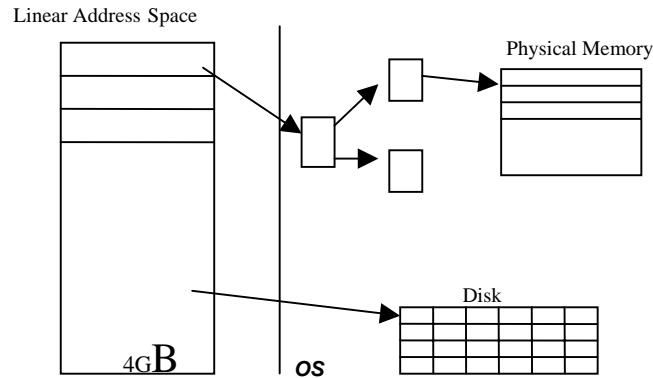
## 1. Introduction

The Memory Management subsystem is one of the most important parts of the operating system. And virtual memory (VM) is one of the strategies that make the system appear to have more memory than it actually have. In a VM system, the addresses are virtual addresses that will be converted into physical address based on the information in a set of tables. To make this translation easier, all of the addresses are divided into fixed size block with same size called page, whatever virtual address or physical address.

Because the physical memory is much less than the virtual memory, for saving the physical memory, we can only load virtual pages that are currently used by the applications. This is known as Demand Paging. Figure 1 is the abstract model of VM. In this function, page table plays an important role for memory mapping between virtual memory and physical memory.

Memory mapping is used to map image and data files into a processes address space. In memory mapping, the contents of a file are linked directly into the virtual address space of a process. Therefore, a virtual page can stay in a sector of the disk, or a physical memory page. Any attempt to access a page that is not in physical memory throws a page fault exception, which is also thrown for invalid memory access. Moreover, the kernel maintains that the number of free pages in physical memory must remain above a certain threshold. Some critical kernel processes needed to free resources must be able to obtain

memory to complete their task. The kernel wakes the thread “kswapd” to free pages every time this threshold is violated.



**Figure 1: Abstract Model of VM.**

In this process, if this discarded page has not been changed, it just need be discarded. However, if this page has been modified, the operating system must preserve the contents of this page so that it can be accessed at a later time. This type pages are called dirty page, and will be saved into a special sort of file called swap file after removing from physical memory. There are a lot of algorithms to decide which pages to discard or swap. Linux uses the Least Recently Used (LRU) algorithm to decide the discard pages. The basic element of this scheme is page aging. And for different application, these algorithms will have different action.

It is obvious that memory management sub-system is complex and essential. Proper implementation can generate the best result. If the algorithm selects wrong pages in most cases when the memory resource is exhausted, it will generate thrashing, and then hamper the performance. And a system maybe has the special requirement for the performance of an application in this system. Or like this sample we provide behind with the special memory using cycle. For these applications, the activities of the memory pages can help designer and developer to make the implementation more efficiently.

A virtual page will have its properties about the disk sector position or the physical memory position. And a physical memory can be mapped to a specific process and its virtual memory. In the same, a disk sector can be mapped to the specific process and the physical memory. For the running efficiency, it can be checked if the connective virtual pages are still connective in physical memory. There will be a pre-read algorithm with the discarding algorithm. With the page activities tracing, the implementation result of these result can be tested with dump information.

This paper provides a useful system to monitor the page activities, which changes the kernel source code, and provide separate virtualization application. It shows the swapping operation of pages in real time, and also provides the statistic analysis for the activity.

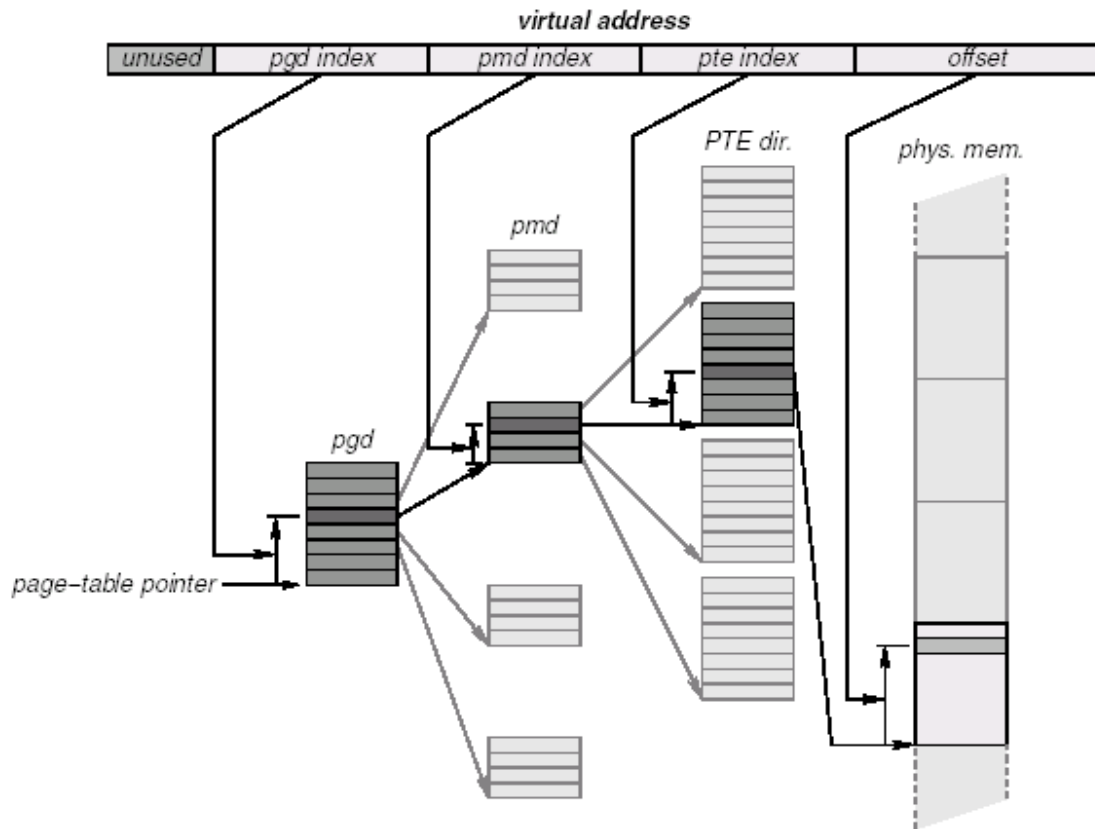
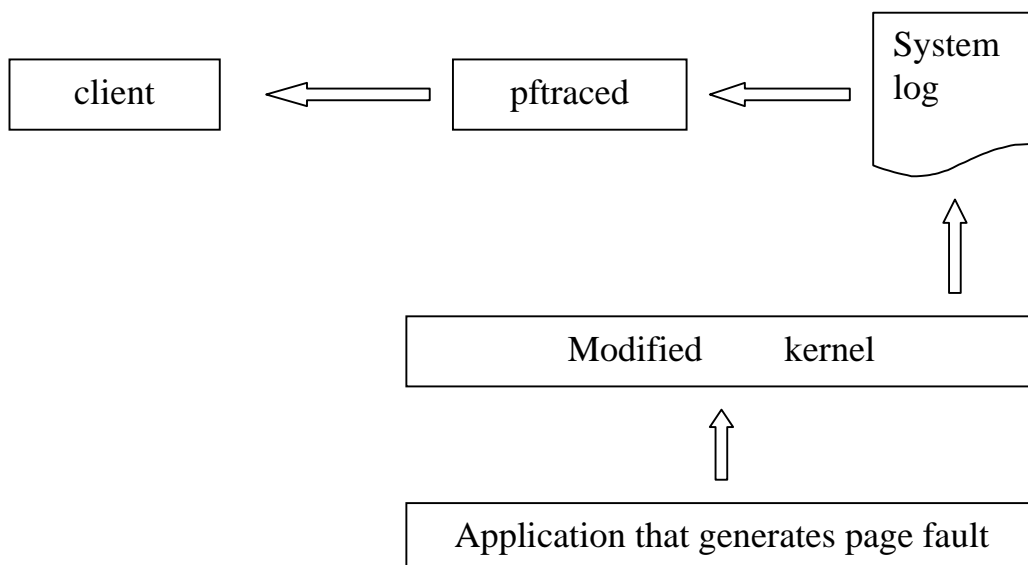


Figure 2: Memory Mapping by Page Table.

## 2. Design and Implementation

### 2.1 Design

As shown in the Figure 5 below, our work consists mainly of 4 parts:



**Figure 3 Design.****2.1.1 Modified kernel**

The kernel should be modified to be able to trace every page fault, whether it is a swap-in or swap-out, and put the concerning information about this pagefault into the system log such as the page number being swapped in/out, the sector from/to which the `kswapd` read/write the page contents, the current time that the swap is taking place; what's more, if it is a swap-in, we should know about the Process ID and the virtual address that caused this pagefault. We only trace the actual swapping operations. If a page is discarded because it is not dirty, we do not record it.

**2.1.2 Daemon that traces page faults**

The daemon should know what has been recorded by the modified kernel, and can send the information to remote client such that it can be visualized to the users

**2.1.3 Client doing visualization to the users**

At the remote site, the client program should always be running to retrieve the information sent by the daemon and then try to visualize it to the users.

**2.1.4 Applications that generate page faults**

First when we find a pagefault, we can get information about this pagefault by reading some kernel data structures defined in the system header file such as "`sched.h`", "`mm.h`", "`mmap.h`" and so on. After that we can use kernel print function "`printk`" to write whatever we get into system log, which actually is a ring buffer.

The first thing for the daemon to do is to read the "`/proc/meminfo`" file to get the size of both the main memory and each page. After that, the daemon will have to read the system log in realtime. To read from the system log, the daemon should use system call "`_syscall3()`" to initialize before using the system call "`syslog()`" to retrieve the contents of the system log.

Having retrieved the messages recorded by the kernel from the system log, the daemon will send out the information to the client via UDP datagrams. And since we are using UDP instead of TCP to send messages to the client, we may lose some datagrams, which means the first message we send to the client containing the sizes of page and memory may be lost and thus the client may not be aware of those useful information. So we will send out such messages periodically such that the client will always be able to know the configuration of the host.

The client will begin to record the pagefault information after it received the first message containing the sizes of memory and page. And for the sake of display, we will only show the changes of 2048 pages.

---

## 2.2 Kernel Hacking

In order to carry out the proposed functions, we need to modify the linux kernel, inserting our code to gather those required information and transfer the information out. Our modification needs to be minor such that the kernel is not affected much by our code. Also, our modification needs to be precise such that we get the correct information we want. So, before we start to make any changes in the kernel, we need to study and understand the kernel.

Fortunately, there are plenty of documents on Internet that helps us a lot to understand the swapping in and swapping out mechanism. In linux, swapping in and swapping out are implemented independently. When there is a legal page fault, the page containing the virtual memory address (VMA) that triggered the page fault will be swapped into the physical memory. However, swapping out is performed independently from the decision of swapping in. A kernel thread `kswapd` will swap out some pages when the free physical memory is below a predefined threshold. Hence we need to insert at least two pieces of code into the linux kernel, one piece to record swapping in information, the other to record swapping out information.

We also need to go through those data structures adopted by linux kernel that stores the information of tasks (processes), pages, swap-devices, and so on to dig out the required information.

In this section, we first present the function tree of both swapping in and swapping out. Then we present how to gather the required information.

### 2.2.1 Function Tree

In this section we present the function tree of both swapping in and swapping out. The knowledge of the control flow of the function tree is a necessity to fetch valid information. For example, some data fields are filled only after the call of a function. So if we print out the data field before the call, the information we get is meaningless. On the other hand, this knowledge also helps us to decide where to insert our code. As what we have done, we put only one piece of code for both swapping in and swapping out to record all required information at once. In this way, our change to the kernel is very small and still we fulfill our goal.

When a page fault, which is an exception, occurs, the page fault exception handler, `do_page_fault()` is invoked by the kernel. In `do_page_fault()`, the VMA that triggered the page fault is fetched. After an intensive checking and get rid of all possible illegal page faults, `do_page_fault()` invokes `handle_mm_fault()`. In `handle_mm_fault()`, it goes through the page table to get the corresponding *page table entry* of the VMA that triggered the page fault. Linux adopts a three-level page table, each has name as *page directory index*, *page middle directory* and *page table index* respectively. However, in i386 architecture, it actually implemented a two-level page table, i.e., the first two levels indices are in fact one level. After get a free page table entry (*pte*), `handle_mm_fault()` invokes `handle_pte_fault()` to validate the *pte* by swapping in the pages into physical memory and fill the page number into *pte*.

The structure of *pte* is very important in virtual memory management [1]. According to the status of whether the page is in physical memory or in the disk, the meaning of *pte* is different.

---

Pages in physical memory (*Present=1*)

Address	Avail			D	A	U/S	R/W	P	
31-12	11-9	8	7	6	4	3	2	1	0

Pages not in physical memory (*Present=0*)

Swap Offset	Swap Type	P
31-8	7-1	0

**D**-Dirty    **A**-Accessed    **U/S**-User/Supervisor    **R/W**-Read/Write    **P**-Present

**Figure 4 The Structure of *pte*. (From [1])**

As shown in Figure 6, if the page is in physical memory, then the first 20 bits are the page number of the physical memory where the page is stored. Otherwise, the page is in the swap-device, and the first 24 bits is the *Swap Offset*. As far as we can see from the source code, the *Swap Offset* \* 8 is actually the sector number starting from where the page is stored in the swap partition, provided we use swap partition but not swap files. In the next subsection, we will explain this in details.

The real swapping operation is carried out in `do_swap_page()` which is invoked by the `handle_pte_fault()`. In `do_swap_page()`, the page is read from the swap device and the *pte* is set to be *Present* in memory and the first 24 bits is filled with the page number of the physical memory storing the page. This is done by `set_pte()`. So it is safe to insert our piece of code right after `set_pte()` to record the page number by extract it from the valid *pte*. The whole process is shown in Figure 7.

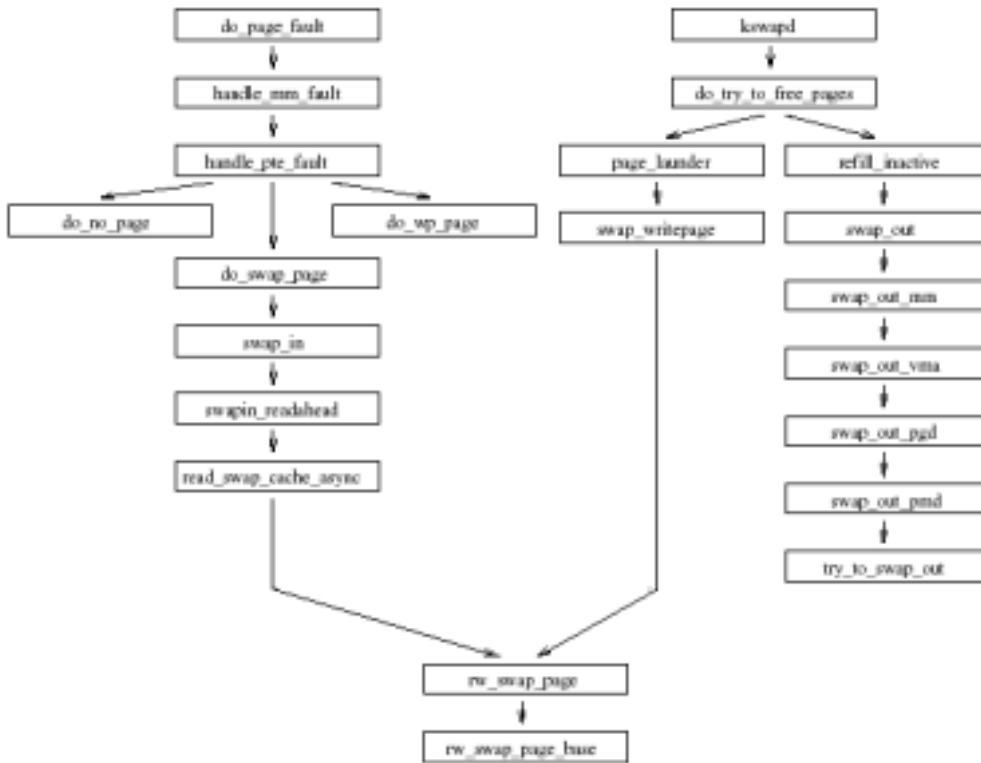


Figure 5 Function Tree. (From [1])

The process of the swapping out is also shown in Figure 7. A kernel thread `kswapd` will try to free some memory when the free memory is below a predefined threshold by swapping out some “useless” pages. The `kswapd` keeps checking the size of free physical memory and invokes `do_try_to_free_pages()` when necessary. The `do_try_to_free_pages()` in turn invokes `page_launder()` to do page “laundry”. It scans a *LRU* list of all *pages* and tries to either discard the page if it is not “dirty” (not written by any processes) or write it back to disk if it is “dirty”. If the *page* is dirty, it will pass the `PageDirty(page)` and finally reaches the `write_page(page)`, which carries out the swapping operation. After that, we can get the page number by `page - mem_map`, where `mem_map` is a global variable indicates the starting of the the table storing all *pages*. We will discuss it in details later.

In the following subsections we present the implementation of our recording function.

### 2.2.2 Swap In

We inserted a piece of code in `/usr/src/linux/mm/memory.c` to record the process ID, the VMA, the page number, the sector number and the time when swapping in a page. In function `do_swap_page()` we did the following modification.

```

*****
static int do_swap_page(struct mm_struct * mm,

```

```

    struct vm_area_struct * vma, unsigned long address,
    pte_t * page_table, swp_entry_t entry, int
    write_access)
{
    ...
    set_pte(page_table, pte);

    /* MODIFY START */
    if ( pte_present(*page_table) ) {
        do_gettimeofday(&tv);
        ncount ++;
        printk( KERN_INFO
            "pGswAP_iN %06ld %ld.%06d %05d %08X %08X %08x\n",
            ncount, tv.tv_sec, tv.tv_usec,
            current->pid, address,
            ((unsigned long)(pte.pte_low)) >> 12,
            ( entry.val >> 8 ) * 8 );
    }
    /* MODIFY END */
    ...
}
*****

```

As discussed in the previous subsection, after `set_pte()`, it is safe to fetch the page number from `pte`. In the following we will explain in details how to get each required value.

In order to write these information into system log, we need to use `printk()`, which is a kernel version of `printf()`. All the information sent to `printk()` is passed to system log, and can be fetched by other process by system call `syslog()`.

## Process ID

When switched into kernel space, there is a global variable `struct task_struct* current` to store the information of the current task (process). The field `current->pid` is the process ID of the current process.

## Page Number

As shown in the previous subsection, the highest 20 bits of a valid `pte` is the page number of the physical memory, provided the page is already *Present* in physical memory.

## VMA

This is the variable named as `address` passed into `do_swap_page()`.

---



## Time

A call to `do_gettimeofday()` will return the current system time. Actually `do_gettimeofday()` is how the kernel implement the system call `gettimeofday()`. Since now we are in kernel space, we can only access the `do_` version of a system call.

## Sector Number

This is the hardest part in our project, we did not find any documents on Internet discuss how does linux calculate this. As far as we can see from the source code, if the swap device is a swap partition, then  $sector = Swap\ Offset * 8$ , as explained below.

As shown in Figure 7, all read and write on swap-device is actually carried out in function `rw_swap_page()`. So we start from it and discovered the following calling path:

```
rw_swap_page() -> rw_swap_page() ->
brw_page() -> submit_bh()
```

The I/O operation is performed in `submit_bh()` where we found the following statement:

```
count = bh->b_size >> 9;
bh->br_sector = bh -> b_blocknr * count;
```

Hence, as we guessed, the `bh->br_sector` indicates the sector number where in the swap partitions we put the page. In `brw_page()`, we found that the `bh -> b_blocknr` is set as the *Swap Offset*. Also, in `create_empty_buffers()` we found that `bh->b_size` is set as  $PAGE\_SIZE / zones\_used$ , where *PAGE\_SIZE* is a macro defined as 4096, and for swap partition, *zones\_used* is set to be 1. This is because that a page is of 4096 bytes, and a sector is of 512 bytes. So linux groups consecutive 8 sectors as a zone to store a page. Hence we know that  $sector = Swap\ Offset * 8$  when swap partition is used.

In the kernel, we found that the *Swap Offset* is stored in two places other than *pte*, the *index* field of *struct page*, and the *val* field of *swp\_entry\_t*. Since there is a *swp\_entry\_t entry* passed into `do_swap_page()`, we simply use  $(entry.val >> 8) * 8$  to get the sector number.

The file that contains all functions discussed above is listed in Table 1.

<code>rw_swap_page()</code>	<code>/usr/src/linux/mm/page_io.c</code>
<code>rw_swap_page_base()</code>	<code>/usr/src/linux/mm/page_io.c</code>
<code>Brw_page()</code>	<code>/usr/src/linux/fs/buffer.c</code>
<code>submit_bh()</code>	<code>/usr/src/linux/drivers/block/ ll_rw_blk.c</code>
<code>create_empty_buffers()</code>	<code>/usr/src/linux/fs/buffer.c</code>

**Table 1 File and Functions**

### 2.2.3 Swap Out

We insert a piece of code in `/usr/src/linux/mm/vmscan.c` to record the page number, the sector number and the time when swapping out a page. In function `page_laundry()` we did the following modification.

```
*****
int page_laundry(int gfp_mask, int sync)
{
    ...
    writepage(page);

    /* MODIFY START*/
    do_gettimeofday(&tv);
    printk( KERN_INFO "pGswAPoUT %ld.%06d %08X %08x\n",
            tv.tv_sec, tv.tv_usec, page - mem_map,
            (page -> index >> 8) * 8 );
    /* MODIFY END */
    ...
}
*****
```

As discussed in previous subsection, after the call of `writepage()` we are sure that the page is written into swap partition and then it is safe to record the required information stored in the variable `page`. In the following, we explain in details how to get each required value.

#### Page Number

In linux kernel, `mem_map` is a global variable that is the starting address of an array of `struct page`. The interesting thing is that any `page` stored in the  $i$ th position of the array `mem_map` corresponds to a page in physical memory starting at the address  $i * 4096$ . Hence,  $i$  is the page number. So here we use `page - mem_map` to get  $i$ , which is the page number.

#### Time

This is the same with what we did in swap in.

#### Sector

As discussed in previous subsection, `page -> index >> 8` is *Swap Offset*, so we use `(page -> index >> 8) * 8` to get the sector number.

## 2.3 Daemon

### 2.3.1 The goal of *pftraced*

The main purpose of *pftraced* is to set up a real-time connection for the kernel and the client. And it's *pftraced*'s duty to tell the client program about the sizes of the memory and the pages.

### 2.3.2 What does *pftraced* do

#### a. Reading “/etc/pftraced.conf”

To get the ip address of client and server such that it can send messages between these two machines.

#### b. Initializing as daemon

To be initialized as a daemon, we should call `fork` once, and let the parent process exit. Then we call “`setsid()`” and “`umask(0)`”, and change the working directory of this process to root by “`chdir("/")`”. After these 3 steps, we can set this process as a daemon.

#### c. Reading `meminfo`

The second line of “`/proc/meminfo`” contains memory size and there is a system call “`getpagesize()`” that can return the page size, which can tell us the size of each page.

#### d. Fetching the result stored by the kernel

First we should print such a line at the very beginning of the daemon such that we can use “`syslog()`” to read the contents of the system log afterward:

```
_syscall3(int, syslog, int, type, char *, bufp, int, len);
```

When reading the system log via “`syslog()`”, we should specify the parameters as follows: `syslog(4, buf, MAXSIZE)`, in which “4” denotes read and clear it from the system log, “buf” specify the buffer to store the result, and `MAX_SIZE` tells about the maximum size to read.

#### e. Sending UDP packets to client site

- **resolve the content of the buffer**

Check if it is concerning pagefault.

Read contents of the buffer, including time and page number, and if it's a swap-in, also get the processID and virtual address.

- **compose the send message**

Convert `time(unsigned long)` into `localtime(hour:min:sec.ms)`

- **send out the packet**

Set a variable `mcount`, in each round `mcount = (mcount + 1) % 64`. When `mcount == 0`, we will compose and send a message containing the sizes of memory and page.

---

## 2.4 Visualization

In this project, we study and revise the kernel source code to implement the Page Active Tracing. For express our working, we implement the visualization part to show our result. In this section, we will explain and describe the visualization part. It is developed under

Operation System: Windows 2000

Development Tool: Visual Studio 6.0

Support Development Tool: BCG library

Actually, the demon application will gather the data periodically from the system log, and send the data by UDP packet at port 9091 to the client site where visualization application lies. And this application will bind and listen to this port. The data packet is organized by the fixed format. In the client site, the application will decode it, and send the data to the GUI. The GUI is divided into three basic parts:

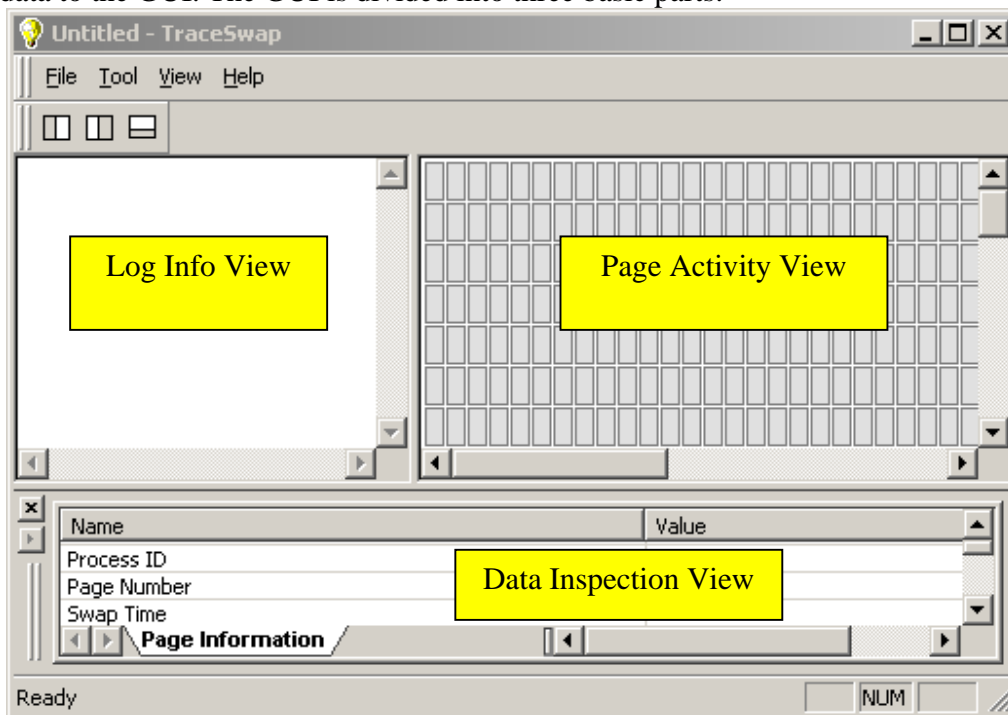
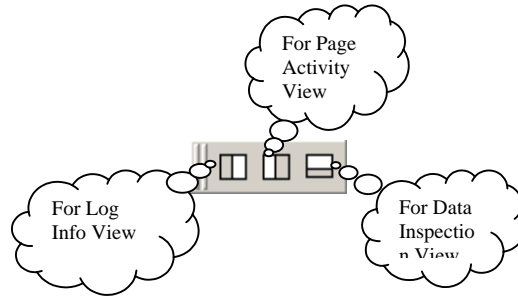


Figure 6: The GUI of Page Activity Trace Tool

In the log info view, it will show any information gotten from the server site. The messages use "IN" and "OUT" to separate the Swap-In and Swap-Out Operation. All of the data will be showed in this view. And in Page Activity View, the Swap-in and Swap-out operation will be shown by the animate. Blue color is used to express that there is swap-in operation, and Red color is for swap-out operation. Here, for distinguish the special condition that there are successive two swap-in and swap-out operation, whatever which is early. At the beginning, all of the color will be gray. It means all of these pages are not traced. And the relative data information will be 0. After tracing any pages, the normal color is white. Now, it should have the different data, and you can see the detail information by click the item. The selected status will be green same as the swap in&out operation. After clicking the item, the following Data Inspector View will show the

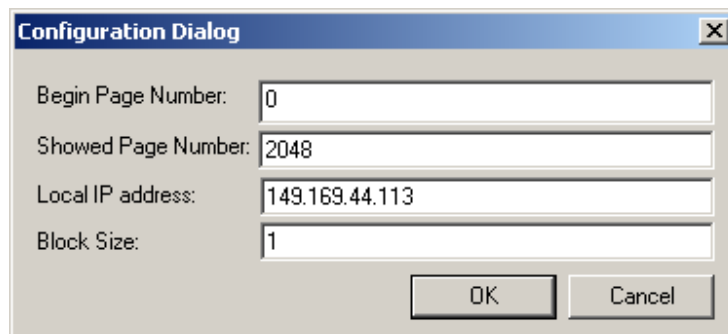
information of this page in detail. It includes the Process ID, Page Number, Swap Time, Virtual Memory Address, and Sector information.

Certainly, to satisfy the view requirement, each view can be showed or hide by the menu or tool bar. In the upper Figure, you can see that there are three tool buttons in the tool bar. Each one can hide / show one view.



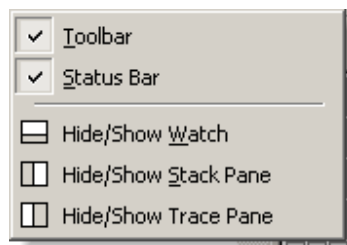
**Figure 7: Tool Bar**

Because the memory size of the testing computer system is bigger, and the size of each page is just 4K, we can't show whole pages in one view. So we separate the memory into several bigger blocks each of which has 2048 pages. For inspecting different block, we provide a Configuration Dialog to let you change the beginning page number. The dialog is simple listed in the following diagram.



**Figure 8: Configuration Dialog**

Certainly, you can hide the tool bar, menu bar, and status bar in the GUI to expand the inspector view. All of the command are listed in View menu, the pop up menu is



**Figure 9: Popup Menu of View Menu**

## 3. Testing and Evaluation

### 3.1 Testing

#### Pagefault.c (First Version)

```
#define TWOPAGES_SIZE 1024
int memory1[30000][TWOPAGES_SIZE];
int memory2[30000][TWOPAGES_SIZE];
int main()
{
    int i, j;
    while (1)
        for ( j = 0; j < TWOPAGES_SIZE; j++ ) {
            for ( i = 0; i < 30000; i++ )
                memory1[i][j] = memory2[i][j] = 0;
            sleep(1);
        }
    return 0;
}
```

In this sample, we provide two dimensions array, and iterate this array by the first dimension as the inner loop. Each unit in the first dimension will have bigger size than a page. So there will be one page between two connective swapping activities.

#### Pagefault.c (Second Version)

```
#include <stdio.h>
#include <sys/types.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

#define ARRAY_SIZE 10142720 //10142720
int memory1[ARRAY_SIZE];
int memory2[ARRAY_SIZE];

int main()
{
    int i, j, k, l, m;
    long n;
    pid_t first, second;

    first = fork();
    if(first == -1 ) {
        printf("fork error!\n");
    }

    if (first==0){
        //child
        printf("enter first child!\n");
        while(1) {
            for (n = 0; n < ARRAY_SIZE; n++){
                memory1[n] = memory2[n] = 0;
            }
        }
    }
}
```

```

    }
    else {
        //parent
        second = fork();
        if( second == -1 ) {
            printf("fork error!\n");
        }

        if(second == 0) {
            printf("enter second child!\n");
            //child
            while(1) {
                for (n = 0; n < ARRAY_SIZE; n++){
                    memory1[n] = memory2[n] = 0;
                }
            }
        }
        else {
            //parent
            i = 1;
            printf("enter parent process!\n");
            while(1) {
                if( i == 0 ) {
                    kill(first, SIGSTOP);
                    kill(second, SIGCONT);
                    i = 1;
                }
                else {
                    kill(second, SIGSTOP);
                    kill(first, SIGCONT);
                    i = 0;
                }
                sleep(30);
            }
        }
    }
    return 0;
}

```

We developed a very short but effective program to test. Each memory copy will cause a page fault, so there will be many page faults in a very short time. And what is more, since it is easy for us to know the rules of page faults generated by this program, it will be fair for us to know whether or not our work is correct or not.

## 3.2 Evaluation

### Small Size of Daemon

A major difficulty for this project is that our program may generate pagefault by itself and thus affect the result of testing. Our daemon has a small size such that it cannot generate too many (if any) page faults when monitoring the system.

## Realtime Visualization

By limiting the number of pages to be presented, we are able to realize visualization in realtime, which is a desired feature.

## 4. Conclusion and Future Work

We have fulfilled all the requirements of the project. Some possible future work is to further reduce the affection of the tracing facility; apply our project to practical applications.

## Reference

- [1] Brian Holinka, *Memory Management 2: Virtual Memory CSCI780: Linux Kernel Internals*, Fall 2001, [http://www.cs.wm.edu/~brian/research/papers/cs780f01\\_vmm.pdf](http://www.cs.wm.edu/~brian/research/papers/cs780f01_vmm.pdf)
  - [2] David A. Rusling, *The Linux Kernel*, [http://www.mil.ufl.edu/~number9/doc/linux\\_kernel.pdf](http://www.mil.ufl.edu/~number9/doc/linux_kernel.pdf)
  - [3] David Mosberger and Stephane Eranian, *IA-64 Linux Kernel: Design and Implementation*, Chapter 4, <http://vig.pearsoned.com/samplechapter/0130610143.pdf>
  - [4] Joe Knapka, *Virtual Memory Management Policy*, <http://home.earthlink.net/~jknappa/linux-mm/vmpolicy.html>
-