

Graceful Degradation under Resource Overbooking

Anupam Chanda (*anupamc@rice.edu*),
Amit Saha (*amsaha@rice.edu*)

06th December, 2002

1 Introduction

It is trivial to write malicious code that hogs up a particular resource, for example, memory, disk, etc. Such malicious code can be written by intent (in order to degrade the system performance considerably) or by mistake (for example, memory leaks). As an effect of running such code the other applications running on the same machine suffer and their performance degrade considerably. The systems response time to any user input becomes very slow, thus making it almost impossible to stop the malicious process. Such kind of resource overbooking is a prevalent problem. In this project we present two approaches to reduce the problem of resource overbooking. We consider the specific case of the resource being main memory. However, our approaches are general enough to be applied to other resources such as disk, CPU, network bandwidth, etc.

2 Problem Description

In an attempt to test how severe the problem of resource overbooking can get, we wrote the following pathological program:

```
#include <stdio.h>

int main()
{
    while(1)
    {
        char *p;
        p = (char*)malloc(sizeof(char)*4096);

        /* touch the page so that the page is
         * brought into physical memory
         */
    }
}
```

```
        memset ( p , 0 , 4096 ) ;  
    }  
}
```

Following were our observations on running the code, as a normal user (as opposed to running as *root*) on a FreeBSD 4.3 and a FreeBSD 4.7 machine:

1. The system started swapping and hence hit the disk as the program *malloced* more and more memory.
2. The system's response time became very slow. At the time of running this test code, KDE was being run as the desktop environment. All windows as well as the mouse took forever to respond.
3. Eventually the system ran out of swap space and the process with the largest memory usage (resident set size + swap size) was killed by the kernel. In many scenarios the KDE server happened to be the largest process and was hence killed by the kernel, thus killing all X sessions on the desktop.

We traced down the events that took place before the machine started to behave in such an erratic manner. We came up with the following chronology of events:

1. The kernel tries to satisfy the memory requested by the malicious process. In trying to do so, the kernel has to page out from the physical memory pages that were held by other, presumably non - malicious, processes. However, the kernel does not pay any attention to either the rate at which the malicious process is requesting memory or the resident set size of the process.
2. As the malicious process keeps on requesting more and more memory, the kernel starts swapping out processes which are currently present in physical memory. Again, the kernel does not try to identify which process is actually causing the memory hog. Once the kernel starts swapping processes the performance of the system degrades considerably and since the kernel has to keep on swapping processes out of physical memory (because of continues memory requests from the malicious code), the degradation in the performance of the system gets worse. This is the time when the user realizes that something bad is going on in the system. However, by this time it is too late to act upon it as the user interface of the system stops responding.
3. As the memory requests from the malicious code keeps rising, even the swap space gets exhausted and the kernel takes the drastic step of finding the process which has the largest memory footprint (i.e. the sum of the resident set size and the swap size of a process) and subsequently killing it. The largest such process can very well be a perfectly non - malicious process that takes a lot of memory but whose rate of increase in memory is very low (for example, X server, Netscape, etc).

The important thing to notice is that even though the kernel has information about the resident set size of each process as well as the page faults caused by each process, the kernel does not utilize these information to try and prevent the degradation of the system. Our work tries to use the information already maintained by the kernel to prevent resource overbooking by a particular process. In Section 6 we describe how our mechanisms can be extended to apply to units of execution other than the process for example, users, groups, etc.

3 Design

We monitor the processes running on the system to catch any possible indication of memory hogging. Once we identify such indications, we take corrective actions against the processes which have caused the same. We present two such schemes for the above strategy.

3.1 Page Fault Scheme

We observed that, when a process *mallocs* and subsequently touches that page, a page fault is generated, and this causes the kernel to allocate a page for that process in physical memory. Now, if the sum of resident size of all processes on the system is approaching the size of physical memory, such allocation of a page will be at the cost of paging out of some other page (victim). If the victim page happens to belong to some other process, which subsequently tries to touch it, it will suffer a page fault, and will incur performance degradation. So we argue, that page fault rate can be used as an indicator to whether a process is actually trying to hog up memory. The higher the page fault rate, the more likely that the process is malicious.

For our pathological test code, we actually measured the number of page faults, and found them considerably higher than the rate of page faults for normal legitimate applications running on the system. However, we don't have a detailed study of page fault rate distributions for normal applications, like Netscape, emacs, etc. So we had to fix some ad hoc threshold for page fault rate, and if some process crossed that, we infer that the process is malicious.

We present our algorithm for identifying and penalizing malicious process in the following pseudo code.

```
/*
 * THRESHOLD = allowable threshold for page fault rate
 * WINDOW = measurement window interval
 * HIGH_WATER_MARK = fraction of physical memory,
 * if sum of resident size of
 * processes go above this, then take action
 */

procedure monitor_pagefault_rate(Proc p)
{
    /* p is the process being monitored */
```

```

pf = no. of page faults for p in the last WINDOW;
/* p.timeout is the timeout for which
 * this process was suspended last time,
 * 0, if it was not suspended */

if (pf > THRESHOLD) {
    /* Process may be malicious */
    total_rss_size = sum of resident set size of all processes;
    if (total_rss_size >
        HIGH_WATER_MARK * physical_memory_size) {
        /* need to take action */
        if (p.timeout == 0) {
            /* first time */
            /* start with some default timeout */
            p.timeout = default_timeout;
        }
        else {
            /* nth. time, grow timeout */
            p.timeout += delta(p.timeout);
            /* delta is inversely proportional to the priority
             * of the process and proportional to p.timeout
             */
        }
        /* suspend p for time p.timeout */
        suspend(p, p.timeout);
    }
}
}

```

3.2 Resident Size Scheme

As an alternative scheme, we base identification of malicious processes on how much resident size they have. Normal applications, pass through phases of memory allocations and doing computations and touching them. The resident sizes of such applications are not insanely huge, and moreover they don't request memory at such alarming rate that their resident size almost fills up the entire physical memory. So as an alternative scheme, we measure the resident set size of processes when the page out daemon kicks in, and suspend the process with the largest resident size for a timeout value. We present the algorithm in the following pseudo code.

```

/*
 * THRESHOLD = fraction of physical memory,
 * if a process uses more than this

```

```

* amount, take action
*/

procedure rss_take_action()
{
    /* called by page out daemon */
    size = size of process with largest rss size;
    big_proc = the proc with the largest rss size;
    /* big_proc.timeout is the timeout for which
    * this process was suspended last time,
    * 0, if it was not suspended */
    if (big_proc.timeout == 0) {
        /* not suspended before */
        /* start with some default timeout */
        big_proc.timeout = default_timeout;
    }
    else {
        /* nth. time, grow timeout */
        big_proc.timeout += delta(big_proc.timeout);
        /* delta is inversely proportional to the priority
        * of the process and proportional to p.timeout
        */
    }
    suspend(big_proc, big_proc.timeout);
}

```

4 Implementation

We implemented our policies in the FreeBSD 4.7 kernel [1] and ran the modified kernel on a Pentium II 450MHz machine with 128MB main memory. We modified the *struct proc* data structure to include entries for *timeout* and *page fault rate*. Since we modified the *struct proc* data structure we had to recompile all libraries which used *struct proc* and also programs which statically linked to these libraries.

For the *Page Fault Scheme* we had to add around 200 lines of code. We did not have to change any line of code. Besides, our additional code can be turned off by a compile time flag. Apart from the additions in *struct proc* made in */usr/src/sys/include/proc.h* all other additions were in the file */usr/src/sys/vm/vm_fault.c*.

For the *Resident Set Scheme* we had to add around 250 lines of code. As in the implementation of the previous scheme, there was no change in existing code and the additional code of this scheme can also be turned off by a compile time flag. All the additional code is in a single file */usr/src/sys/vm/vm_pageout.c*. Since our code kicks in once the *pageout* daemon starts functioning, we had to do a bit more bookkeeping compared to the the page fault scheme as the

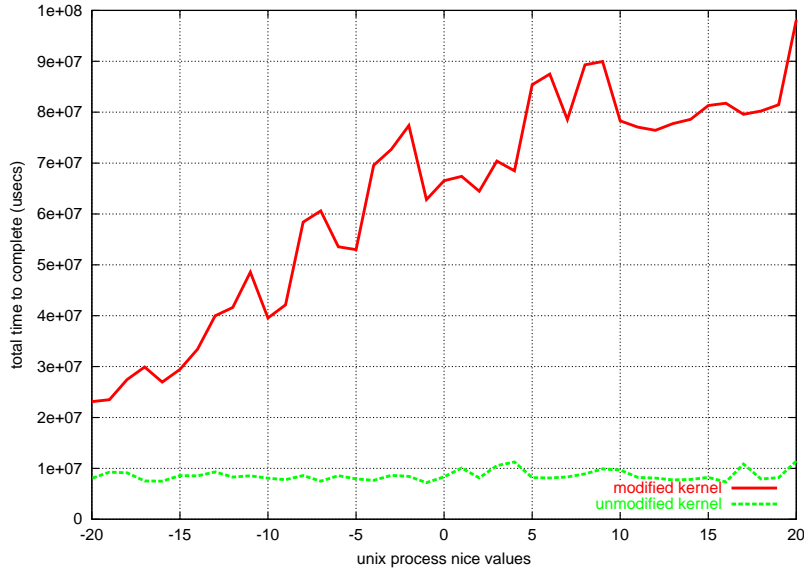


Figure 1: Total Time to execute vs. process nice values

daemon is a process in itself, and we needed to find the process with the largest resident set size. In the previous scheme the bookkeeping effort was minimal since the initial code for handling page faults still runs in the context of the faulting process. This accounted for the difference in the lines of code between the two schemes.

In both the mechanisms we used kernel threads to handle a suspended process (and wake it up after the timeout for that process expires). Since, there is only additional code it can be added to any existing FreeBSD 4.x kernel as a kernel patch.

5 Evaluation

5.1 Page Fault Scheme

We test our Page Fault policy by first running a malicious process which is similar to our pathological test case presented in Section 2. However, this program runs for a finite number of iterations instead of the *while(1)* present in the previous test case. Our mechanism starts suspending the process only after the total resident set size (considering all processes) exceeds a particular threshold, in this case 20000 pages, which is around 70% of the main memory size.

As seen from Figure 1 the total time taken to execute a memory hogging process changes with the nice value of the process. The lower the priority of the process (higher nice value) higher is the time required to complete execution since the system suspends processes with lower priority (higher nice value) longer than processes with higher priority (lower nice value). This becomes more clear in Figure 2. The increase in the timeout value for successive suspensions is much steeper for processes with lower priority. Also, as shown in Figure 1, the total time taken to execute the malicious process in an unmodified kernel is much smaller since, unlike the

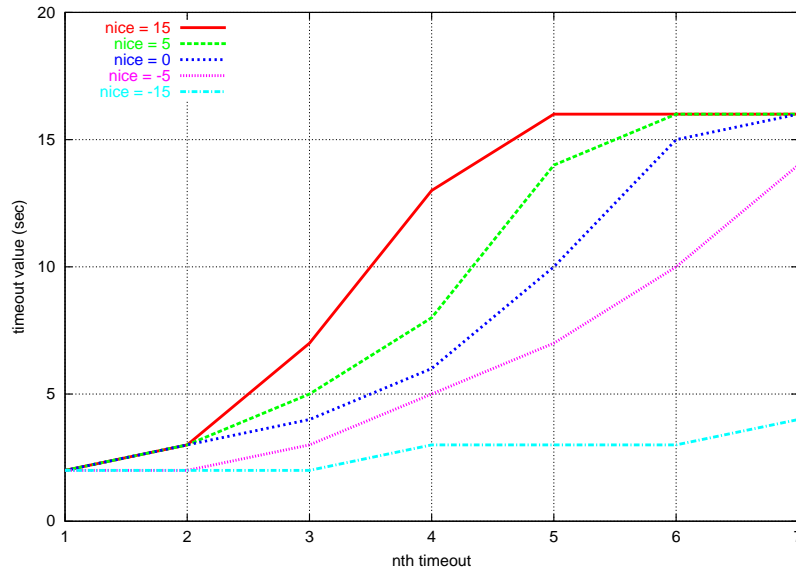


Figure 2: Timeout values on successive timeouts vs. process nice values

modified kernel, the program is never actually suspended by the kernel. However, running such a malicious process in the unmodified kernel runs the risk of running out of swap space and thus getting the largest process killed.

We then tested our Page Fault scheme by running a malicious process along with another non malicious process (i.e. a process which has a constant memory footprint). We run the malicious process at 3 different nice values of -20, -10 and 0 respectively. We do not run the malicious process for nice values higher than 0 because the non malicious process was being run at a nice value of 0 and hence any other process running with a lower priority (i.e. with a higher nice value) will anyway not be able to affect the non malicious process to the extent of degrading its performance noticeably. As shown in Figure 3 the higher the priority of the malicious process (i.e. the lower the nice value), the more the non malicious process is degraded. However, even with a nice value of -20 the malicious process is not able to degrade the performance of the non malicious process. This is in stark contrast with the degradation experienced by the non malicious process in the unmodified kernel. Another important thing to note is that in the unmodified kernel there is a huge penalty in the response time which subsides only after the malicious process dies. However, in the modified kernel the degradation is staggered over time since the malicious is suspended from time to time.

5.2 Resident Size Scheme

For this scheme we started two very large memory hogging processes staggered in time. As shown in Figure 4 the first process reaches a plateau where it is suspended by our scheme since the resident set size of the process exceeds the high water mark. At this point the other process starts and the kernel pages out the suspended process's pages and allows the second process to grow, bringing down the resident set size of the first process. Since the resident set size

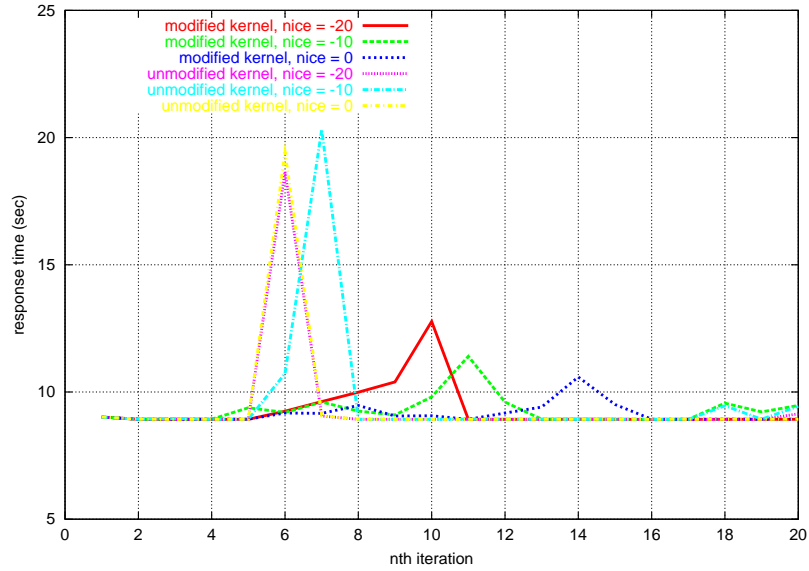


Figure 3: Response time of non malicious process

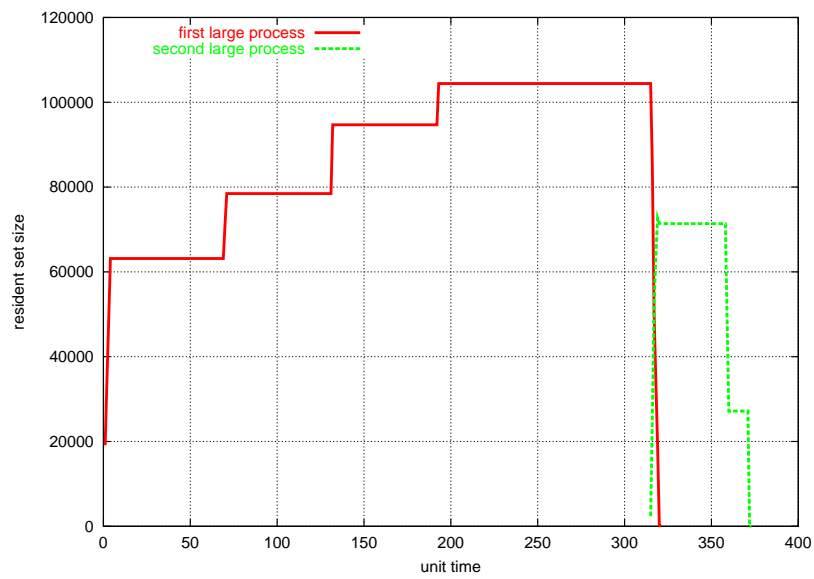


Figure 4: Resident set size of two memory hogging process started staggered in time

of the first process goes below the low water mark the first process can again be resumed and goes to completion. The second process is also suspended when it crosses the high water mark. However, in order to be confident that this mechanism will always work we have to do many more experiments under different workloads. We plan to do this in a future extension to this work.

From our evaluation we come to the conclusion that the page fault scheme is more robust and is also able to take the decision at an earlier stage thus better preventing degradation. The resident size scheme is also effective but is less robust and requires much more experimentation.

6 Applying the design to other resources

Even though we have presented our design for a specific resource, namely *main memory*, and have thus implemented to prevent main memory overbooking yet, our design mechanisms are easily applicable to any other computer resource. In this section we handle some of the more common resources.

CPU

Overbooking of CPU is possibly the least hazardous since UNIX kernels have a very elaborate mechanism by which it prevents process starvation even for processes with very low priorities. Scrutiny of the process priority mechanisms employed by 4.4 BSD kernels [2] reveals that a process's user mode priority is decreased linearly based on recent CPU utilizations. The user can change the *nice* value of the process but the kernel prioritizes a process based upon the process's nice value as well as the process's recent CPU utilization. This is a policy that is very similar to the approach that we have taken. What is surprising is that even though the kernel does prioritization based upon CPU utilization yet, the kernel ignores priorities and utilization in case of other resources such as memory.

Disk Access

As in the case of hogging main memory a malicious process can hog access to the disk. Even though the malicious process will probably block when doing an I/O from disk yet, the process can generate so many disk requests that the process can completely hog the bandwidth to the disk. This would degrade the performance of any other process trying to access the disk.

Our design can be easily extended to work for disk access. Fundamentally disk accesses and memory accesses are not too different. A process's rate of disk access can very easily be monitored and this rate can be used to prioritize the process's disk access requests. This approach is similar to our mechanism which uses page fault rate to rate limit a process's memory consumption. A mechanism that is similar to our second approach is already present in present day UNIX systems. Following this approach users are given quotas of disk space and once the user's disk usage is greater than the allocated amount, the kernel takes necessary and appropriate actions.

Network bandwidth

In the same way that a process can hog main memory, a connection can hog the network bandwidth. Our approach of rate limiting the memory consumption of a process can be applied to connections to ensure that the network bandwidth is used fairly among all requesting connections.

7 Conclusion

In this work we have provided two simple and effective mechanisms to help non malicious processes to gracefully degrade in the presence of non malicious, memory hogging processes. Our results show that a malicious process, even when run with a very high priority, is not able to degrade the performance of the malicious code too much. Also, when a malicious process is being run in the absence of memory pressure, the process is allowed to run normally (as in an unmodified kernel) until the memory is nearing overload condition (which we identify by the total resident set size of all processes crossing a threshold) when the kernel starts suspending the process. Our mechanisms also take into consideration the nice value of the processes.

Though we have provided the implementation for the specific case of memory yet, our mechanisms are quite general and can be applied to other resources such as CPU, disk, network bandwidth. From our experiments, the page fault mechanism is more robust whereas the resident set size mechanism requires extensive experimentation.

References

- [1] <http://www.frebsd.org>.
- [2] McKusick, Bostic, Karels, and Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1999.