CS 423 MP3 Design and Analysis Document

Group Members:

1) Furquan Shaikh(fmshaik2)
2) Gurmeet Singh(gurmeet2)
3) Puneet Chandra(pchandr2)

Design and Implementation Decisions:

1) Initialization:

The module loading and initialization of procfs entries for providing registration/deregistration access to user processes (work processes) is similar to the earlier MPs. A memory buffer is allocated using vzalloc (to obtain virtually contiguous and zero'ed memory pages). In order to ensure that these pages are not swapped out of memory by the MMU, SetPageReserved is used to set the PG_RESERVED bit for every page. vmalloc_to_page is used to convert the starting virtual address of every page in the buffer to the corresponding physical page.

2) Work Queue:

Additionally, a delayed work queue is created on the registration of the first work process. This delayed work queue is scheduled every 50ms to capture the work process statistics. The work queue handler scans each registered process to capture the accumulated statistics for all processes i.e. major fault, minor fault and the CPU time. These statistics are written to the memory buffer allocated during initialization as a sample headed by the jiffies. When there are no more processes registered, the work queue is cleared and deleted.

3) Character device:

A major number for the character device is allocated dynamically using alloc_chrdev_region. A character device cdev is allocated and initialized with function pointers for the device. The function pointers contain NULL for open and close, which ensures that these operations on the device always succeed. The most important function pointer is the memory mapping function, which maps the memory buffer

allocated during initialization to the virtual address space of the user process(monitor process). The device is then added using cdev_add. This device can be checked in the user space using cat /proc/devices. The device with the allocated major number can be seen under character devices. To remove the device, the cdev_del call is used and unregister_chrdev_region to remove the major number association.

4) Memory mapping:

Memory mapping is done to map the memory buffer allocated in the kernel into the virtual address space of the user process (monitor process). Starting virtual address of every page in memory buffer is used to obtain the corresponding page frame number using vmalloc_to_pfn. This page frame number is used to remap the page in the virtual address space of the process using remap_pfn_range.

5) Cleanup:

On unloading the module, procfs enteries are removed. If there are any registered processes, the memory allocated for these processes is freed. The work queue if present is flushed and deleted. The character device is unregistered. Additionally, an important step to be performed before freeing the memory allocated using vfree is to clear the PG_RESERVED bit of the pages using ClearPageReserved.

Testing:

The following tests have been performed:

1) Execute 2 work processes with random memory access and obtain statistics using monitor process (case 1a)

2) Execute 2 work processes – one with random memory access and other with localized memory access – and obtain statistics using monitor process (case 1b)

3) Plot the graphs for page fault rate for both case1a and case1b (Details in analysis section)

4) Execute N work processes (with N = 1,2,3,…11) all with random memory access to study the effect of multi-programming on CPU utilization(case 2)

Submission Files:

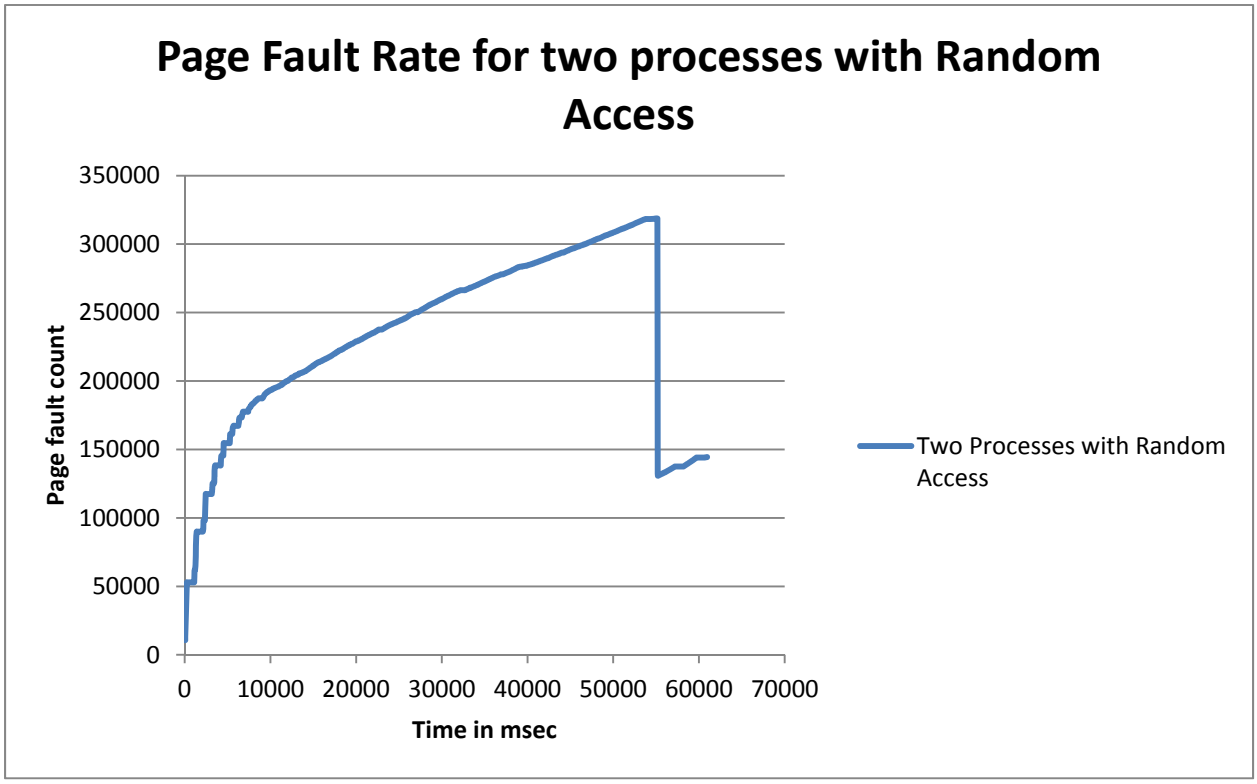
1) mp3_kernel_mod.c – MP3 kernel module implementation

2) monitor.c – Monitor process (Given)
3) work.c – Work process (Given)
4) mp3_given.h – Header file (Given)
5) Makefile – Makefile to compile kernel and user level processes
6) Design and Analysis document – This document

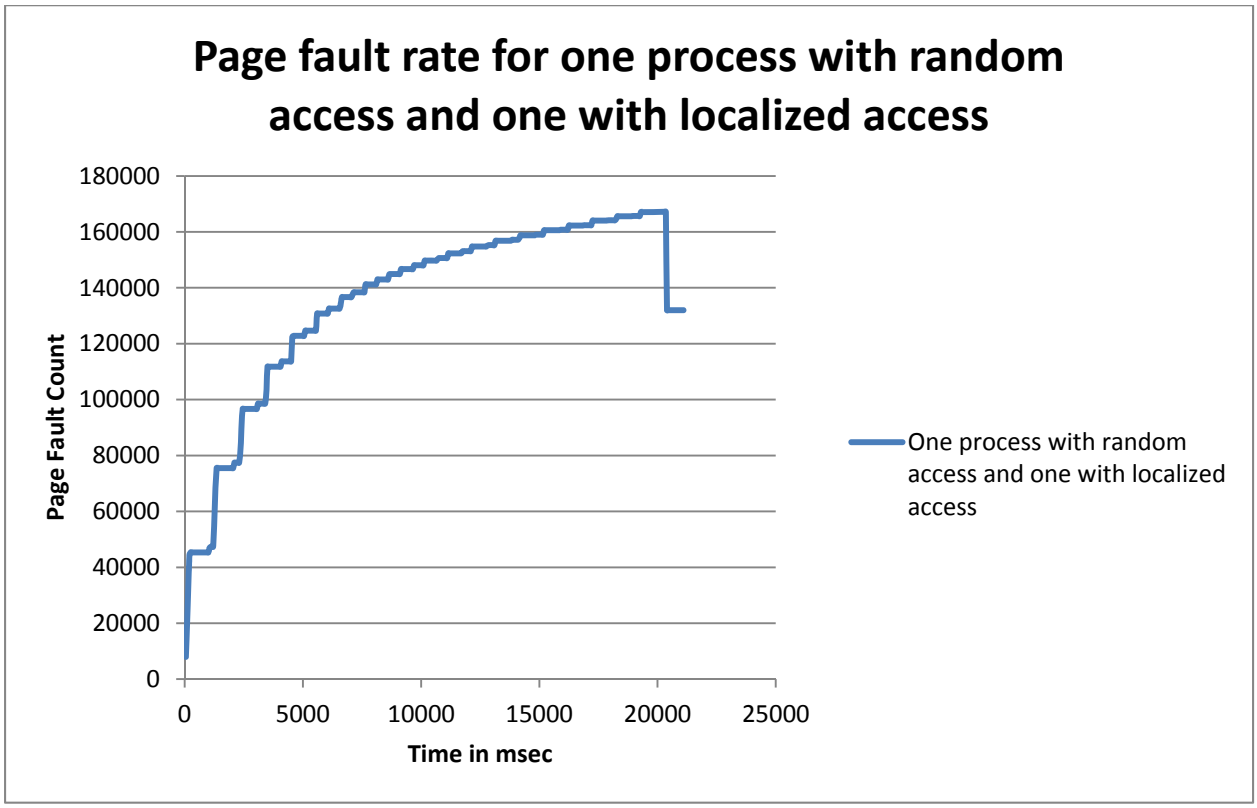
Analysis:

Case Study 1: Thrashing and locality

1a) Two work processes are executed both with random memory access. This is the graph obtained for the page fault rate of the processes



1b) Two work processes are executed – one with random memory access and other with localized memory access. This is the graph obtained for the page fault rate of the processes.



Analysis:

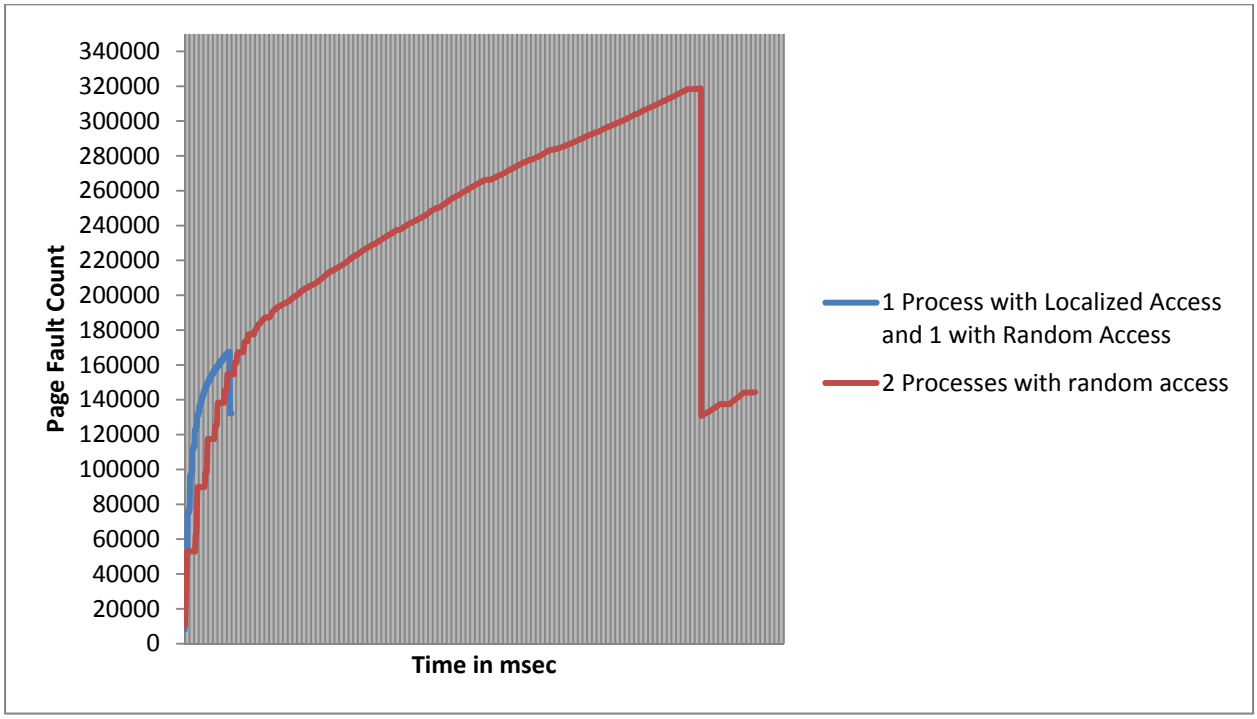
1) From the above two graphs it is clear that with two random processes the page fault count is much higher as compared with one process having random access and other localized access.

2) This shows that, the process which takes advantage of localized access, results in very less page faults.

3) Also, the time required for the two processes to reach completion is very less with a localized memory access process than a process with random memory access.

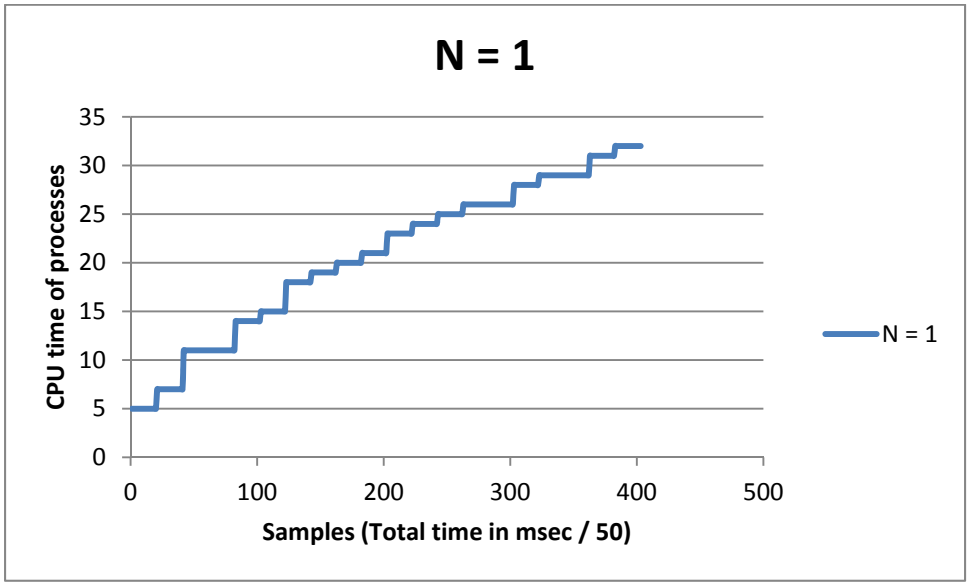
4) Thus, it is clearly evident that processes taking advantage of localized access result in less page faults and less completion time, whereas processes with completely random memory access can lead to thrashing.

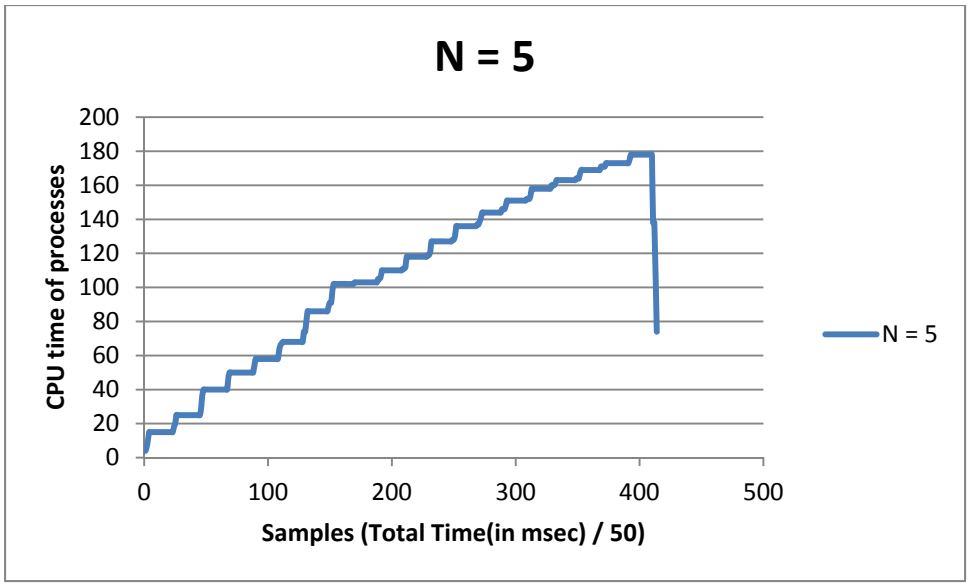
5) These analysis results can be confirmed from the combined graph of the two measures above:

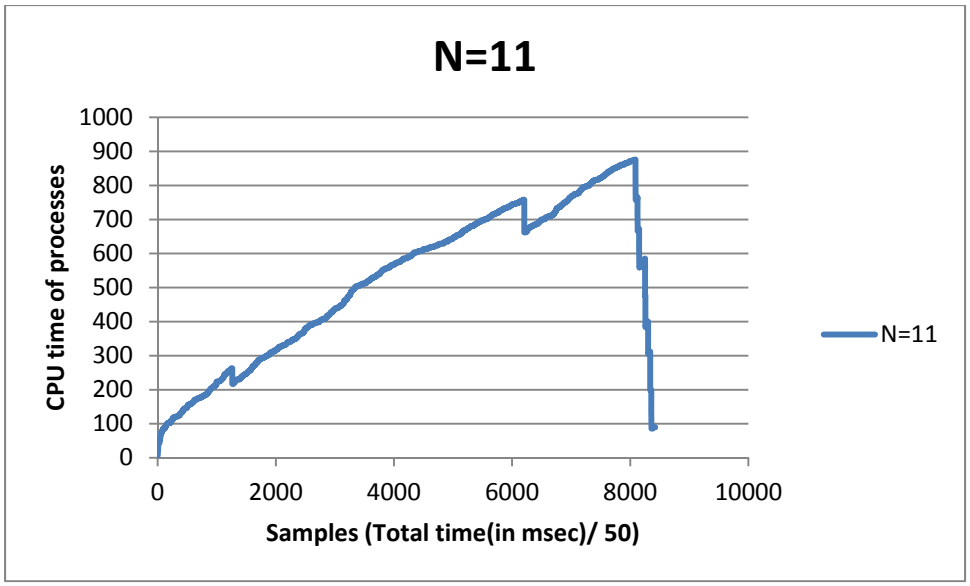


Case Study 2: Multiprogramming

N (1,5,11) processes having random memory access are executed and statistics are collected for the cpu time of the processes. The following graphs are obtained:







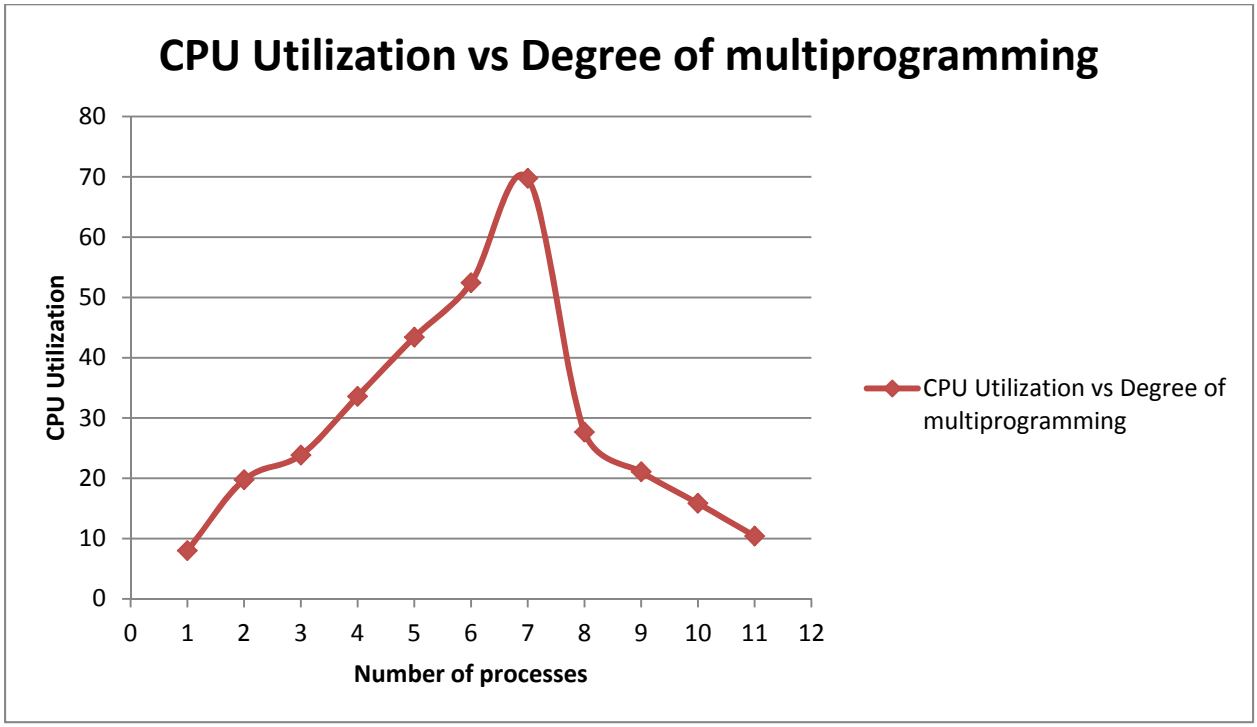
Analysis:

1) As seen from the above graphs, it is clear that the time taken to complete for 5 processes is slightly more than the time taken for 1 process. This shows that the CPU utilization for 5 processes is better than that for 1 process.

2) However, for 11 processes, the time taken to converge has increased significantly. This shows that beyond a certain limit, the CPU utilization falls instead of increasing.

3) This can be proved by plotting a graph of CPU utilization of N=1,2,3..11. This CPU utilization is calculated as a ratio of the maximum CPU time taken by the combination

of the processes to the actual completion time required for the processes. Graph is as shown below:



4) This graph clearly indicates that as degree of multiprogramming is increased, the CPU utilization goes on increasing till a point beyond which it reduces.
5) Possible cause of this behavior is **thrashing**. The degree of multiprogramming increases to such an extent that most of the time is spent in swapping out and in of pages rather than performing useful work.

Case Study 3: Synchronization between kernel and user

Providing concurrent access of the memory buffer to user process and kernel module is nothing but a typical example of producer-consumer problem. We need to ensure that:
1) Producer does not produce more than the capacity of the buffer
2) Consumer does not consume from an empty buffer
3) Produce does not over-write data that is not yet consumed by the consumer

The most straightforward solution to this problem is locking and flags. Basically, the kernel module can set a flag to indicate to the user process that the buffer is ready to be read. Once the user process is done reading the data, it can indicate to the kernel module that it has completed reading. Thus, the kernel module can start writing to it.

The flag can be either a reserved portion of the buffer (starting byte of the buffer) which can be set/reset to achieve synchronization. Another possible solution could be that the flag is present in the procfs like /proc/mp3/syncbuff. The user process can read this value to check with the kernel if buffer is ready to be read. Once done it can write to it to indicate that reading is done and kernel can continue writing.

The pseudo code for this approach could be something like:

```
monitor.c:
while(1) {
while (read_flag() == 0);

// Read data from the buffer

write_flag();

// Sleep for 30 seconds
}

kernel_module:
// Writing to buffer done
// Pause work queue
// Set flag to indicate to user process

// Write from user process to flag
// Continue with work queue
```

One major drawback with this approach is that the kernel needs to wait until the user process finishes reading from the buffer. In order to avoid this, the buffer could be divided into 2 sub-buffers such that while the kernel is writing to one sub-buffer the user process is reading from the other. Thus, the flag can be updated to instead contain the address of the buffer to read from. If there is no address returned the user process waits.

monitor.c:

```
while(1) {
while(read_addr() == NULL)
// Sleep for 30 seconds

// Read data from the buffer

write_done();
}
```

kernel_module:

```
// Maintain the buffer address completed writing to
// Start writing to free buffer address, if available. Else wait
// When user process asks for address, return address of the first completed buffer
// When user process done reading, add buffer to free buffer list. Continue writing if waiting for free buffer.
```

This method can be extended to maintain n sub-buffers instead of a single buffer. This way we can ensure that the kernel waiting time is reduced to minimum. Size of each sub-buffer needs to be sufficient enough to hold samples generated during 30 seconds. Also, we need to ensure that no sample is written across two sub-buffers. This way, synchronization can be achieved between the user process and kernel module