**Project-2 Document**

**Sharanya Gottimukkula**

**Ramya Kalvakuntla**

**Implementation:**

**Question 1:**

In this part we have designed a process generator that generates 50 processes with different memory requirements in the range 10KB to 2MB and cycles in range 200 to 2500 cycles. Each process will arrive in the system for every 50 cycles. Once the process is arrived, it is allocated with the required memory and is ready to run until it finishes its cycles. The memory is allocated using malloc and de-allocated using free ( C lib functions).

Below is the structure which represents the process with cycles and memory

typedef struct Process {

int max\_cycle;

unsigned char \*memory;

int id;

int cycle;

int status;

} process;

The process comes in the system for every 50 cycles and after that it is assigned with the cycles in the range 200 to 2500 and memory in the range 10KB to 2MB

if(t%50==0 && p<MAXPROCESS){ // Creating new process for every 50 cycles

process\_list[p] = (process \*) malloc(sizeof (process));

process\_list[p]->id = p+1;// process id

// Setting cycles for a process between 200 to 2500

int cycle = gen\_random(200,2500);

process\_list[p]->max\_cycle = cycle;

process\_list[p]->cycle = 0;

// Setting memory of process between 10KB to 2MB

int memory = gen\_random(10000,2000000);

process\_list[p]->memory = malloc(memory);

memory\_sum += memory;//total memory

process\_list[p]->status = 0;

printf("New process ID: %2d Memory: %7d[Bytes] Cycles: %4d\n",p,memory,cycle);

p++;

}

The required memory for each process is allocated using malloc and de-allocated using free

**Allocating memory**

process\_list[p]->memory = malloc(memory);

memory\_sum += memory;//total memory

**De-allocating memory**

for(i=0;i<p;i++){

if(process\_list[i]->cycle == process\_list[i]->max\_cycle && process\_list[i]->status==1){

// Free process memory

free(process\_list[i]->memory);

process\_list[i]->status = 2;

printf("Process %d has finished\n",process\_list[i]->id);

cp++;

}

}

**Note:** While using malloc there will be a context switch from user space to system space only if the requested memory is more than the memory available for user.

**Question 2:**

This question is also same as the first question whereas instead of using malloc and free for allocating de-allocating memory, we used my\_malloc( ) and my\_free( ) to allocate and de-allocate memory from the initially acquired memory block of size 100MB.

#define MAXMEMORY 100000000

unsigned char \*main\_memory = malloc(MAXMEMORY);//allocating 100mb memory

The context switch will be done only once to get 100mb memory after which the 100mb is used to allocate memory.

**my\_malloc():**

When a new process arrives , it is assigned memory from the initial memory of 100 MB. For this we used a method my\_malloc which has a 2-dimensional array to keep track of allocated and free partitions. This array will record the memory block’s start and end addresses. When a process comes in this method will search for a free partition and allocate if 1 found. The process memory address is then assigned to be the location of partition. It will also resize the adjacent empty blocks if the middle block is freed

int my\_malloc(process \*cprocess, unsigned char \*main\_memory){

int j;

for(j=0;j<MAXINDEX;j++){

if(partition[j][0]==-1) continue;

// Found empty block

if(partition[j][1] - partition[j][0] >= cprocess->memory\_size){

// Set memory address on partition

cprocess->address = partition[j][0];

// Set memory buffer

cprocess->memory = cprocess->address + main\_memory;

// Shrink current memory block

partition[j][0] += cprocess->memory\_size;

return 1;

}

}

return 0; // No available memory

}

**my\_free():**

When a process completes its cycles, we will de-allocate the memory using my\_free memory, i.e., this method dynamically resizes the adjacent partition to match the empty hole.

void my\_free(process \*cprocess){

int j;

int pR=-1,pL=-1;

// Find pR and pL

for(j=0;j<MAXINDEX;j++){

if(partition[j][0] == cprocess->address + cprocess->memory\_size) pR = j;

if(partition[j][1] == cprocess->address -1) pL = j;

}

// Resize partition according to freed memory

if(pL != -1 && pR != -1){

// Both left and right blocks are empty

partition[pL][j] = partition[pR][1];

partition[pR][0] = -1; partition[pR][1] = -1;

}else if(pL != -1 && pR == -1){

// Left block is empty

partition[pL][1] += cprocess->memory\_size;

}else if(pL == -1 && pR != -1){

//Right block is empty

partition[pR][0] += cprocess->address;

}else{

// Find empty partition

for(j=0;j<MAXINDEX;j++){

if(partition[j][0]==-1){

partition[j][0] = cprocess->address;

partition[j][1] = cprocess->address + cprocess->memory\_size -1;

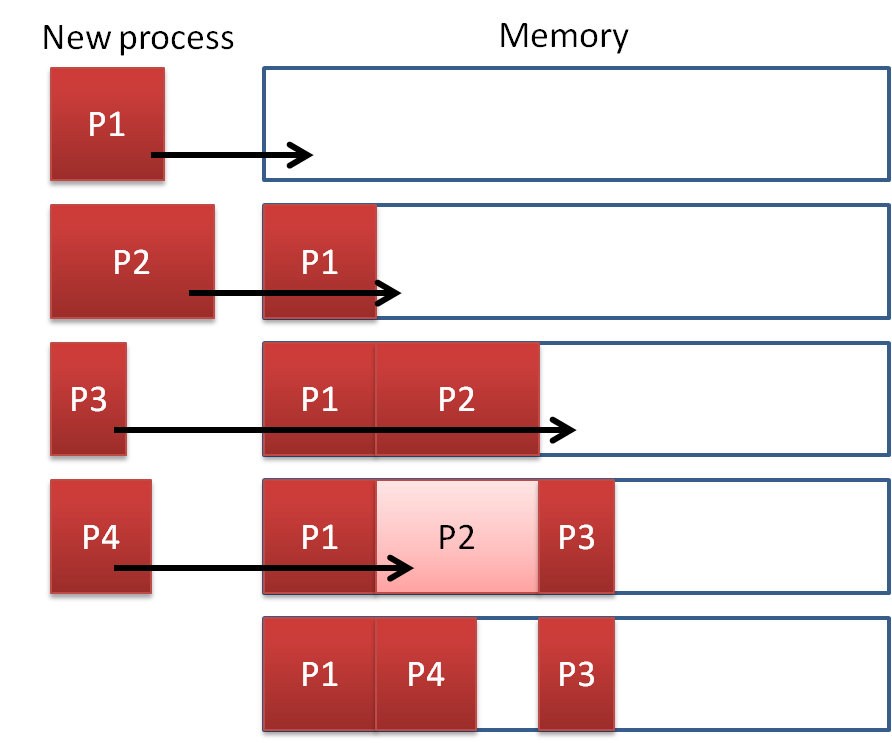
break;

}

}

}

}



**Question 3:**

In this part the memory partition is static. We initially assigned the memory for 100MB. Here we use my\_malloc( ) and my\_free( ) to allocate and de-allocate memory. The memory of 100MB is divided into 20 partitions of 5 MB each.

**my\_malloc:**

Once the process arrives, the process will be allocated with anyone of 20 partitions made in the 100MB memory.

int my\_malloc(process \*cprocess, unsigned char \*main\_memory){

int j;

for(j=0;j<MAXPARTITION;j++){

if(partition[j]==0){

cprocess->address = j;//setting memory address partition

cprocess->memory = main\_memory + j \* BLOCKSIZE;

partition[j] = 1;// Set current memory block as used

return 1;

}

}

return 0;

}

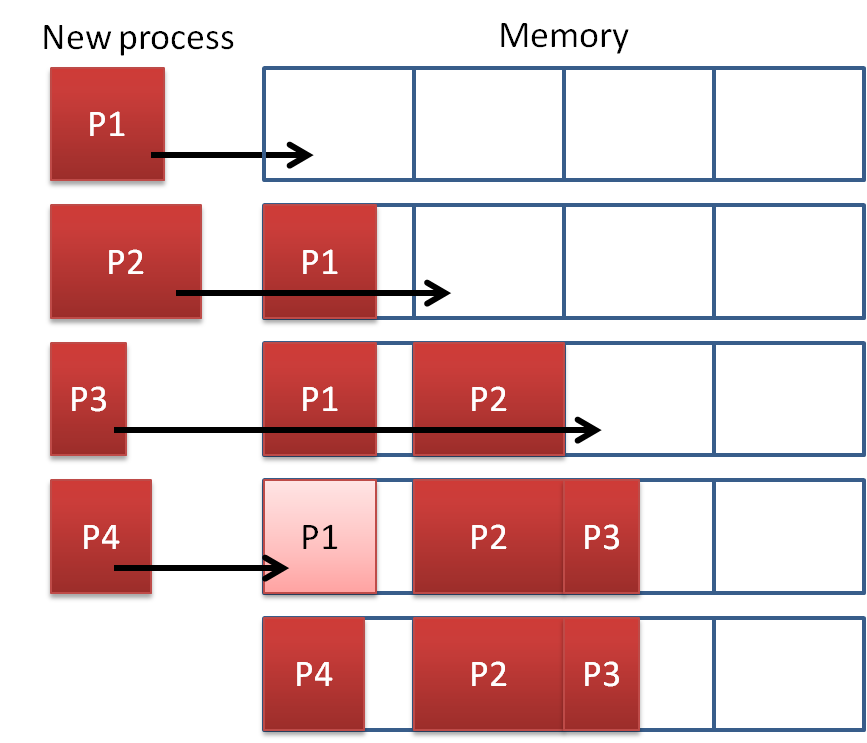
**my\_free:**

Once the process completes its cycles, the memory partition will be set as free again.

void my\_free(process \*cprocess){

partition[cprocess->address]=0;// Setting current partition as free

}



**Limitations:**

As for question 2, the total process memories are designed to be less or equal to initial allocated memory. (50 process x 2Mb max memory per process <= 100 Mb initial memory). Thus, there will always available memory for new process. For this reason, process swapping and memory defragmentation was not implemented. Moreover, the partition record is implemented as simple 2D array. For better efficiency, a linked list can be used instead, e.g. adding in-between elements is faster. But In this case the overhead due to context switching from user to system space will be reduced.

Similarly, in question 3 the new process will always have available memory block to be used, thus the process does not have to wait. There will be internal fragmentation because the memory of size 5MB is allocated even for memory request of size 13KB.

Both implementations just assign the process memory location to an address on initially allocated memory block. Thus, if the process uses even slightly more memory than it should, the adjacent process memory will be affected.

**Experiments:**

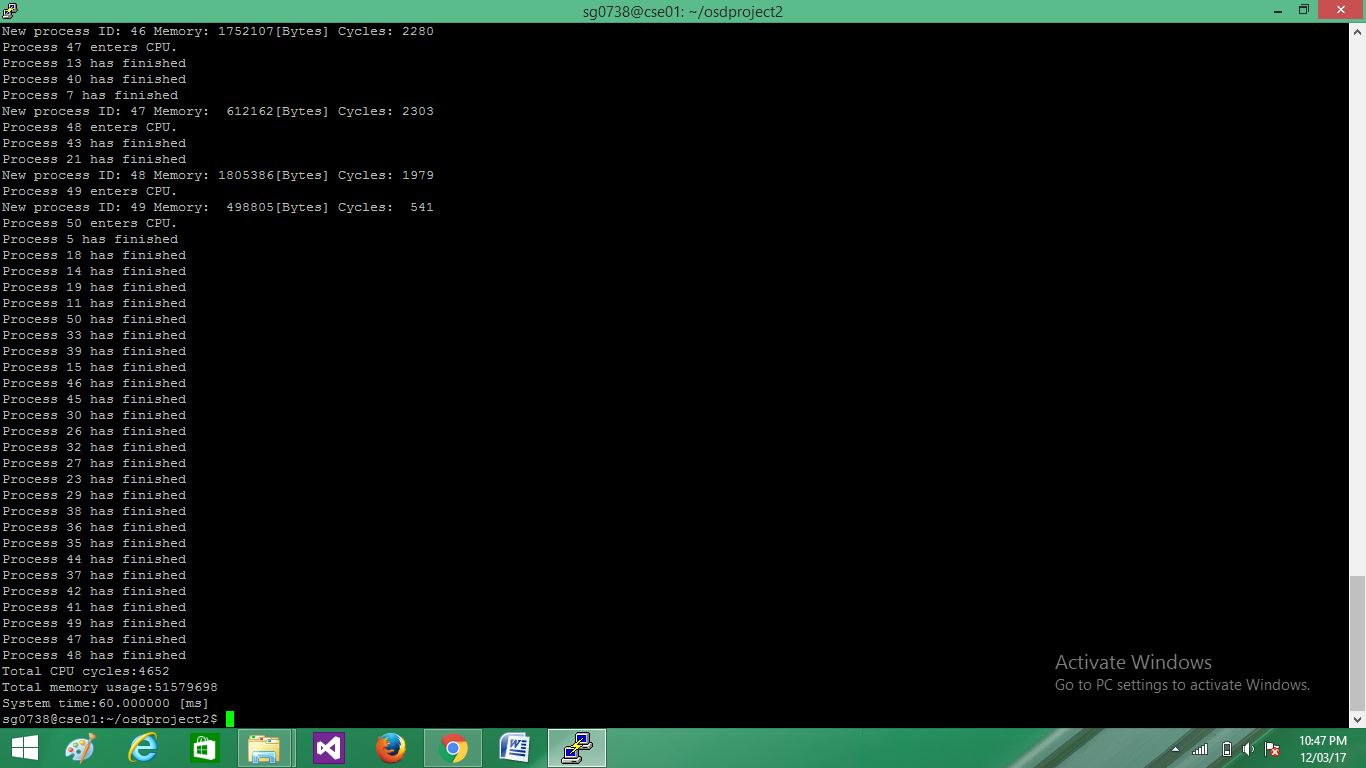
While trying to implement memory manager by using malloc only once to get 100mb memory so a sto reduce overhead due to context switches. we taught of two approaches initially to allocate memory for 50 processes dynamically from the initial 100mb block i.e using array or linked list to keep track of free blocks, finally used array because if we use linked list we need to call malloc every time which was not the requirement.

First when we started using array it was taking more time to search for the free blocks, then we enhanced the array search to make it faster. The other issue was the malloc will perform context switch only if the user space available is not sufficient i.e it will switch to the system and request memory only when the available system space is fully utilized, but 100mb was very less when compared to the CSE machine user space memory. We could observe the overhead due context switching clearly in our own laptop.

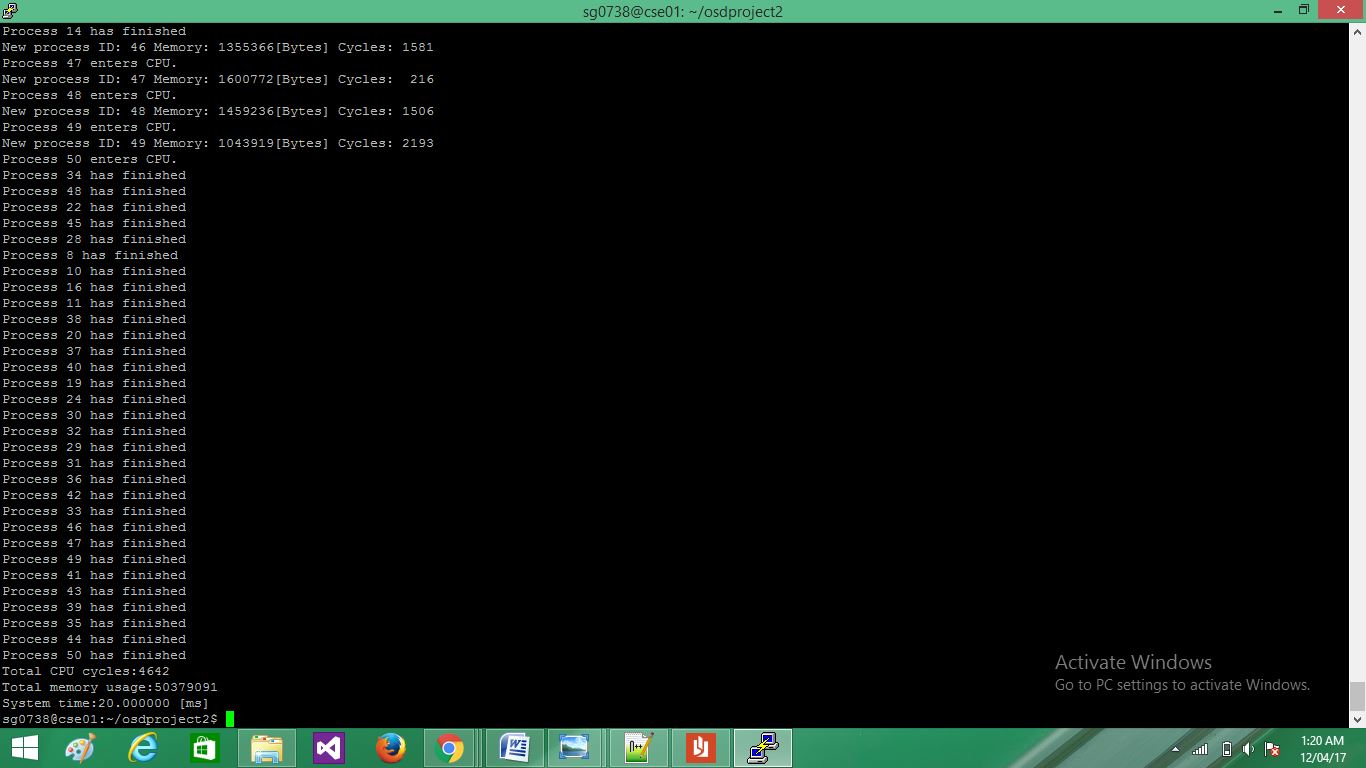
When we increment the processes number the memory requirement will be more than 100mb as the memory of processes is in the range 10kb to 2mb and the memory manager slows down because 100mb will be very small block but the overhead due to context switching will be reduced. The new processes, which come in, should wait for the executing processes to complete their cycles to get a free block.

**RESULTS:**

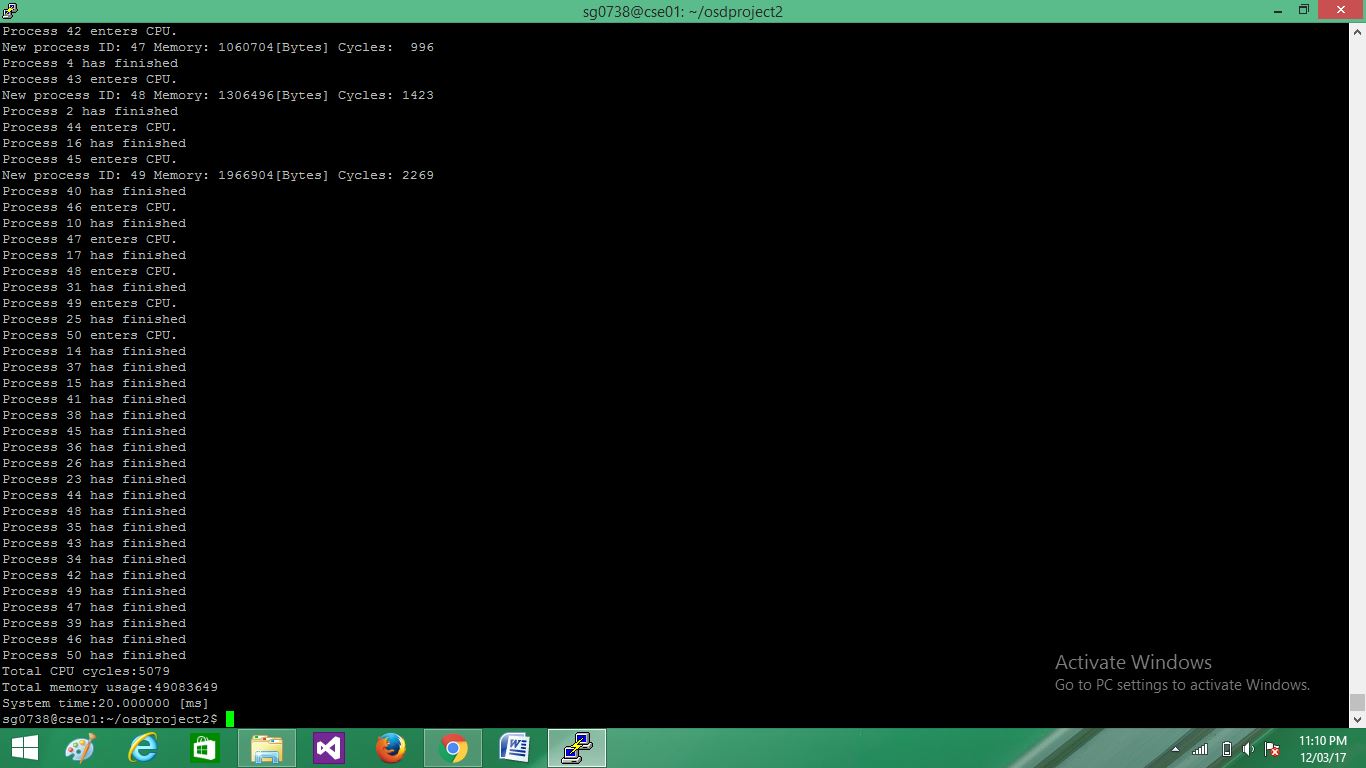
**Question 1 output:**



**Question 2 output:**

****

**Question 3 output:**

****