Operating Systems

Report of Design Decisions
of our library

of

Assignment-6

by

Harsha, Vachan, Yuvraj, Chandu
20CS10071,20CS10070,20CS10025,20CS10017

COMPUTER SCIENCE DEPARTMENT
INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR

0.1 Structure of Internal Page Table

```
typedef struct pageTableEntry{
    unsigned int address : 30;
    unsigned int valid : 1;
    unsigned int scope : 1;
    void init();
    void print();
    }  pageTableEntry;
```

This is user defined data type for page table entry. Each Page table entry is a 32-bit unsigned number. The most significant 30 bits are for storing the offset of our main memory in **address** field, the next bit is used for the valid **field** which denotes whether we currently have a valid list at this page table entry (if it is valid then 1 else 0) and last bit is used for the **scope** field which denotes the scope of that entry

Functions:
init() : It initializes all data members to 0
print() : It displays the entry

```
typedef struct{
    pageTableEntry tableEntry[PGSIZE];
    int head;
    int tail;
    sizeT size;
    int insert(unsigned int address);
    void init();
    void print();
    int remove(int index);
    }  pageTable;
```

This is user defined data type for Page Table and its data members are **tableEntry** which essentially contains the array of page table entries. Here when we need to add

a new entry to the page table we find an empty entry in the array whose valid bit is 0 and insert the element at that position.In this process of finding an empty set **head** and **tail** are used. For maintaining which indices are free in the page table are free we used implicit free list (it does not provide any space overhead). Here head stores the first index which is free, and tail stores the last index that is free. Now, the address field is utilised only when that position is filled with some entry. So, for every free index, starting from head, we store the next index that is free in the address field itself, creating an implicit list. Thus, whenever a new entry has to be inserted, it is inserted at the index head, and head is moved one place forward by using head = tableEntry[head].address. Also, whenever a spot in the page table is freed, it is added to this implicit list after tail. This allows us to find an empty spot and insert a new entry in O(1) time.**size** field is used to store the current number of filled entries.

Functions:
init(): Initializes all the entries in the Page Table.
insert(unsigned int address): Adds the new entry to the Page Table.
remove(int index): Removes that particular index entry from the Page table.
print() : It displays all the entries of the Page Table

0.2 Additional data structures/functions used

Memory :

```
typedef struct{
    int *start;
    int *end;
    sizeT words;
    sizeT freeWords;
    int numFreeBlocks;
    sizeT maxFreeBlockSize;
    int init(sizeT size);
    int *findFreeBlock(sizeT size);
```

void allocateBlock(int *block, sizeT size);
void freeBlock(int *block);
int getoffset(int *block);
int *getaddress(int offset);
void printMem();
} memory;

This is user defined Data Type for Memory and we used implicit free list concept to access different blocks of the Memory. Detailed Explaination is below

Data Members:

start: The start address of the malloc memory segment.

end: The address just after the last block of the memory segment.

words: The size of the memory segment (in words).

freeWords: Total amount of free space in the memory segment (in words).

numFreeBlocks: The number of free blocks in memory.

maxFreeBlockSize: Size of the largest free block.

Functions:

init(): Allocates the required amount of memory

getOffset(int *block): Converts the absolute address in the malloc memory segment to offset from the start.

getAddr(int offset): Converts the offset to the absolute address in the malloc memory segment and returns that address.

findFreeBlock(sizeT size): Finds a free block of memory for size words.

allocateBlock(int *block, sizeT size): Allocates memory for size words at address block and sets the appropriate headers and footers.

freeBlock(int *block): Deallocates the memory block at address block and sets the appropriate headers and footers.

printMem(): Displays the current memory blocks along with all its info

Finding Free Block

We use the first-fit strategy for finding an empty memory block of atleast the required size. We keep on traversing the memory segment, and advancing the current pointer using block = block + (*block right shift by 1 bit), till we find a free block that

satisfies our requirements. This might take O(N) time, where N is the number of blocks. However, it is actually much less because we have large blocks in the memory, and the pointer advances to the next block.

Allocating a Block

Since the allocated space might be smaller than the free space, we might have to split the block into an allocated and a free block. This requires changing 2 headers and 2 footers, which can be done in O(1) time.

Freeing a Block

For freeing a block, just changing the allocated bit in the header and footer to 0 is not enough. We need to coalesce the block with the previous and next blocks if they are empty. This can also be done in O(1) time.

Advantages of the Implicit Free List

We do not require any external data structure to keep track of the free and used blocks, and hence it is very simple and clean to implement.

Compare it to a linked list implementation for the free and used blocks. So, each node would have to store an address, size of the block, and a pointer to the next node, requiring a total of 12 bytes. In the implicit free list, we have an overhead of 8 bytes per block (header and footer). Also, finding a free block would take linear case in a linked list, and freeing would also take linear time if we were to coalesce as then we would have to keep the linked list sorted by address. We achieve freeing a block in O(1) time using the implicit free list.

Global Stack :

```
typedef struct{
    char* name;
    int length;
    int index;
    }  stackEntry;
```

This is user defined datatype for StackEntry which stores the list name and length of the name and index of that list address to the page table

```
typedef struct{
    stackEntry stack[STSIZE];
    sizeT size;
    void init();
    stackEntry top();
    int push(stackEntry index);
    stackEntry pop();
    void print();
    int find(char* name);
    }  Stack;
```

This is user defined datatype for Stack and it stores the array of stackentries and this keeps track of which lists are in scope currently and are accessible. Whenever a new stack is created it is pushed into stack and when it goes out of the scope it is popped and size field is used to store the current number of stack entries.

Functions:
init(): Initializes the stack.
top(): Returns the element at the top of the stack.
push(int v): Pushes an element onto the stack.
pop(): Pops the element at the top of the stack.
print(): Displays the contents of the stack.

List :

```
typedef struct list{
    int element;
    struct list *next;
     struct list *prev;
    }  list;
```

This is the user defined datatype for list

0.3 impact of freeElem() for merge sort

memoryfootprint and run time with and without freeElem (average over 100 runs).

Memory Footprint AVG without freeElem() : 6.355935% of memory is used.

Run Time AVG without freeElem() : 31096604 micro seconds.

Memory Footprint AVG with freeElem() : 5.989721% of memory is used.

Run Time AVG with freeElem() : 34638105 micro seconds.

Less memory is used when we use freeElem because we free space when we use freeElem
While using freeElem runtime will increase because of the extra computation.

0.4 Memory Footprint AVG

All the above described data structures used for this assignment are declared globally, so that all the 3 threads can access them.
More information is in .txt files in submission

Performance Maximise: while using Recursive code structure our performance is maximised due to continous freeing of the blocks of memory.

Performance Minimise: while using Iterative code structure our performance is minimised because freeing of blocks of memory is done at the end.

0.5 Memory Footprint AVG

We did not use any locks because we have only one thread of execution.