

- 1.) [12 points] Short Answer Questions, Write a short paragraph – complete sentences and thoughts -- to answer each of the questions given here.

GRADING(page 2): 3 points each

- a) Describe the parts of and the purpose of the Process Control Block (PCB).

ANSWER: Stores all of the information about the process – so that when context switching happens, the information about this process stays with this process.

- process state: new, ready, etc.
- processor registers
- Memory Management Information
 - page tables, and limit registers for segments
- CPU scheduling information
 - process priority
 - pointers to process queues
- Accounting information
 - time used (and limits)
 - files used
 - program owner
- I/O status information
 - list of open files
 - pending I/O operations

Grading (3 points): They must somehow state that it stores information about a process (1 point)
They must then list at least two things contained in the process (1 point each)

- b) Describe Ahmdal's Law and how it can be used to make your code more efficient.

Answer: Ahmdal's law tells how much time will be gained by improving a portion of the code. The formula is based on "alpha" the percentage of time taken by the portion that is being improved and k the amount of improvement on that section. Both "alpha" and "k" must be maximized in order to see large improvements. It helps because it lets you know which sections to work on with what portion of your time because you can know what kind of improvement to expect.

Grading: (3 points) Must mention it is a formula to tell the amount of improvement achieved (1 point)
Must mention that it is based on "alpha" and "k" (not necessarily by name (1 point)
Must mention how it helps you in improving your code (1 point)

- c) Describe the purpose of the "Retirement Unit", which is part of the Instruction Control Unit.

Answer: The retirement unit is in charge of making sure tasks are completed in the correct order. It takes care of the register bank. It is the last part of the series of tasks completed for a single command so it takes care of updating the registers correctly if this task was indeed supposed to take place at this time.

Grading: (3 points) Must mention that it is the last task (1 point)
Must mention that it controls the updating of the registers (2 points)

- d) Show how the following code could be made more efficient for instruction level pipelining assuming you want at least 3 instructions before accessing the same register so that you can keep the pipeline filled. Also, explain why it is more efficient.

```
int    sum=0, arr[100], i;
/*arr is completely filled here */
for (i = 0; i < 100; i++) {
    sum += arr[i];
}
```

Answer:

```
int sum=0, arr[100], i;
for (i = 0; i <= 97; i+= 3){
    sum1 += arr[i];
    sum2 += arr[i+1];
    sum3 += arr[i+2];
}
sum = sum1+sum2+sum3 + arr[99];
```

This is more efficient because by using three different summing places, there are no stalls which would be caused by one sequential instruction waiting until after the register writeback of the previous instruction before it can do the instruction decoding phase.

Grading: (3 points) Must give or at least describe code as shown above (1 point)
amount of loop unrolling does not matter (must be at least 2)
Must describe with concept of stall and writeback (1 point)
-- If they just did the unrolling without the different sum variable on each line and described the unrolling benefit they get 1 point of the 3.

2.) [19 points] Write functions to do each of the following tasks using the following types:

<pre>#define TSIZE 27 typedef struct Node { int *key; int *value; struct Node *next; } Node; typedef Node* Pntr;</pre>	<pre>typedef Pntr[TSIZE] TArr; typedef struct{ int size; /* total number of elements in table */ TArr data; /* actual data of the table */ } Table; /* an empty table has a 0 for the size and all pointers in the array as NULL */</pre>
--	---

- a) Write the function which will completely empty the hash table (leaving an empty table). You must be sure there are no memory leaks, and you do not have any helper functions provided that will clean up parts of the table. The function returns 0 if there were nodes to delete or -1 if the table is empty or any error occurred.

FUNCTION CALLED: Table x; /*other code here filling the table*/ cleantable(&x);

PROTOTYPE: int cleantable(Table*);

ANSWER:

```
int cleantable(Table *tpntr){
    int i;
    Pntr curr = NULL, nextone = NULL;

    if (tpntr->size == 0) return -1;
    for (i = 0; i < TSIZE; i++){
        curr = tpntr->data[i];
        while (curr){
            nextone = curr ->next;
            if (!curr->key) return -1;
            else free(curr->key);
            if (!curr->data) return -1;
            else free(curr->data);
            free(curr);
            curr= nextone;
        }
        tpntr->size = 0;
        return 0;
    }
}
```

Grading:

9 points total

- -2 points for not checking if it is empty and returning -1
- -1 points for not setting the size of the cleaned up table
- -1 if tpntr set to NULL at the end
- -1 point for not freeing key from each node
- -1 point for not freeing the data from each node
- -2 points for not freeing the nodes themselves
- -2 if used curr->next after curr was freed
- -3 up to 3 points lost if the array of the table is not traversed to all lines in the table
- -3 up to 3 points lost for not correctly traversing the linked lists within a element of the table

GradingNotes:

- checking if table pointer is NULL is not needed but should not be marked as incorrect (because we showed how we called it and it couldn't be NULL with that call)
- not setting the elements of the array to NULL (t->data[i]) is -1
- syntax errors and non-declared variables up to -2 points off for blatant syntax errors
- freeing t->data (the array) that is obviously not dynamically allocated is -2
- or setting t->data to NULL is also -2 (but don't take off both)
- ok to not check if curr->key and curr->value are not NULL since we announced they can assume they are allocated
- not setting curr->key and curr->value to NULL is also ok since you're going to free the node next

- b) Write a function that creates a new chained (bucket) hash table which is an exact independent copy of the original. The first parameter passes in the hash table to be copied while the second passes back the new copy that you must create. The function should not modify the original table. It should insert all elements into the same hash line on the table (it does not need to rehash), but the value field should be increased by one. Since the hash was based on the key field rather than the value, changing the value field will not change where it is hashed to. It should return as the return value of the function 0 if the table is completely empty (but any empty copy exists anyway), or the number of elements in the largest chain of the hash table if there are one or more elements in the table, or a -1 for any type of failure.

FUNCTION CALLED: Table x,*y=NULL; /*other code filling x; y is pointing to an empty table;*/ copyhash(x,y);

PROTOTYPE: int copyhash(Table, Table *);

ANSWER:

```
int copyhash(Table old, Table * npntr){
    int i; Pntr curr, nnode;
    npntr->size = 0;
    for (i = 0; i < TSIZE; i++){
        curr = old.data[i];
        while (curr){
            nnode = (Pntr) malloc(sizeof(Node));
            if (!nnode) {
                cleantable(npntr);
                return -1;
            }
            nnode->key = (int *)malloc(sizeof(int));
            if (!nnode->key) {
                free(nnode);
                cleantable(npntr);
                return -1;
            }
            nnode->value = (int *)malloc(sizeof(int));
            if (!nnode->value) {
                free(nnode->key);
                free(nnode);
                cleantable(npntr);
                return -1;
            }
            nnode->key = curr->key;
            nnode->value = curr->value;
            nnode->next = npntr->data[i];
            npntr->data[i] = nnode;
            curr = curr->next;
            (npntr->size)++;
        }
        return (npntr->size);
    }
}
```

Grading:

10 points total

- -2 up to 2 points lost if the array of the table is not traversed to all lines in the table
- -2 up to 2 points lost for not correctly traversing the linked lists within a element of the table
- -4 up to 4 points lost for not checking and correctly handling all returns from malloc (freeing as needed)
 - freeing the key if the value could not be allocated
 - freeing the node if either the key or value could not be allocated
 - freeing the rest of the table (that was already put in if something could not be allocated).
- -1 not returning the current size when everything was copied correctly
- -3 up to 3 points lost for not inserting the nodes into the correct places of the table

GradingNOTES:

- malloc of the table – based on what we told them is wrong – several still doing it -2??
- not casting malloc/calloc is wrong but our compiler allows them to get away with it -0?
- syntax errors and variables not declared – up to -2 points

3.) [12 points] Write a makefile that builds the programs with the following dependencies. A solid line indicates a file included by another file. You must write the makefile in such a way that typing the command make at the command line will create both executable files based on the correct versions of the files in the current directory. You must make sure you compile so that the code is optimized as much as possible, that you can use the debugger and so all warnings are noted and treated as errors.

ANSWER:

```
CC = gcc
CFLAGS = -O3 -g -Wall -Werror
all: prog1 prog2
first.o: one.h two.h three.h
second.o: one.h two.h three.h four.h
third: four.h
prog1: first.o second.o
      $(CC) $(CFLAGS) -o prog1 first.o second.o
prog2: third.o
      $(CC) $(CFLAGS) -o prog2 third.o
```

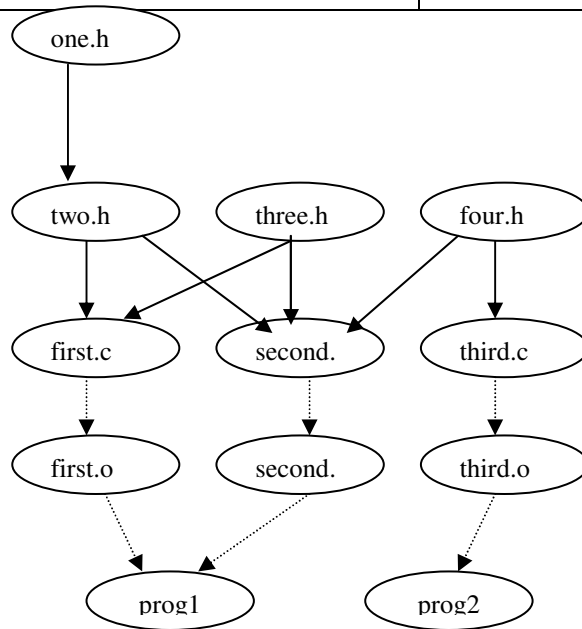
Grading:

12 points total

- -4 up to 4 points lost for options
 - ok if not done with macros as long as options are there
- -2 up to 2 points lost for the “all” line
- -3 up to 3 points lost for the .o lines
 - don’t have to use implied dependencies or actions
 - this includes if they put a “touch” in for one.h
- -3 up to 3 points lost for the executable compilation lines

GradingNOTES:

- macros used but incorrectly should receive -1
- order – “all must be first” others don’t matter?
-



4) Assume you wish to use a binary search tree to store a list of three functions for each of several different people.

- a) [8 points] You need to define the types needed to do this. Each function returns a float, and takes as parameters an array of integers and an integer that specifies the number of used elements in that array. You only need to create the types necessary for storage as described below.

GRADING:

General – if any one of the things listed for that point are incorrect or missing they lose that one point.

1. Create a type named FPntr that is the function pointer to an individual function of the type described above.

ANSWER:

```
typedef float (*Fpnr) (int arr[],int sz);
```

GRADING:

2 points – 1 point for the typedef and (*name) and 1 point for the return type and parameter list

2. Create a type named FArr that is the array of three function pointers.

ANSWER:

```
typedef Fpnr[3] FArr;
```

GRADING:

2 points – 1 point for the typedef and name and 1 point for the type and size of array.

3. Create a type named DataTy that is the data (name and array of function pointers).

ANSWER:

```
typedef struct _dataTy{
    char *name;
    FArr Funct;
} DataTy;
```

GRADING:

2 points – 1 point for typedef, struct and name and 1 point for contents (tag is not required)

4. Create a type named Node that is appropriate for storing the DataTy instances in a binary search tree.

ANSWER:

```
typedef struct _node{
    DataTy data;
    struct _node *left, *right;
} Node
```

GRADING:

2 points – 1 point for typedef, struct, tag and name and 1 point for contents

b) [12 points] These questions continue from the previous page. Using the types you just defined, give prototypes for the following functions:

1. NameHere() - parameters are a character array and a pointer to the root of the tree (as defined above) ; returns a 1 if the name passed as the character array appears in the tree or a 0 if it does not.

ANSWER:

```
int NameHere(Node *root, char *name);
```

GRADING:

3 points – 1 point for return type and name of function, 2 points for the parameter list

2. AddData() - takes a DataTy as the first argument and a second parameter that allows the root pointer to be read and modified by the function. The contents of the DataTy should be added into the tree in the appropriate place for a binary search tree. The function's return type should report the value -1 if there was any type of error during the insert or 0 if everything was fine.

ANSWER:

```
int AddData(Node **root, DataTy data);
```

GRADING:

3 points – 1 point for the return type and name of the function, 2 points for the parameter list

3. NewData() - parameters are a character array and an array of function pointers; returns a pointer to a DataTy. The data contained in the parameters is be used to create the return value.

ANSWER:

```
DataTy* NewData(char *name, FArr functs);
```

GRADING:

3 points – 1 point for the return type of the function, 1point for the name and char array, 1 point for the FArr

- c) [15 Points] These questions continue the previous page. Based on the types defined and the prototypes from the previous pages implement the functions defined below. You may not assume any other functions have already been implemented when you are writing each part below.
1. Implement the code for NewData(), which you prototyped on the previous page. Make sure all data is a true copy of what is passed in through the parameters. The character string passed will be a valid character string, and the array of function pointers is also good.
(You may not use strdup in implementing this function.)

ANSWER:

```
DataTy* NewData(char *name, FArr functs){
    DataTy* temp;
    temp = (DataTy *) malloc(sizeof(DataTy));
    if (!temp) exit (-1);
    temp->name = (char*)calloc(strlen(name)+1,sizeof(char));
    if (!temp->name) exit (-1);
    strcpy(temp->name,name);
    for (i = 0; i<3; i++)
        temp->data[i] = functs[i];
    return temp;
}
```

GRADING:

8 points

- -4 for passing back the address of a statically allocated structure
- -2 up to 2 points lost for not allocating space
- -2 up to 2 points lost if name is not copied correctly (enough space allocated etc)
- -2 up to 2 points lost if function array is not copied correctly (memcpy is fine here)
- -1 point lost for not testing return values of malloc/calloc

GRADINGNOTES:

- leaving off the +1 on the strlen for how much space is allocated is a -1
- strcpy the wrong direction is a -1
- calloc takes its arguments in either order
- If they didn't use the FArr and instead used a pointer to a (function pointer) they have to dynamically allocate that array – no extra points for doing this but -1 if they needed it and didn't do it or didn't do it correctly
- In general allocating the wrong amount of space (other than the off by 1 above) is a -2 to -3 (if often)

Write a function named TestThis(), which will return a 0 if the function above works or a -1 if the function above does not work (you can still assume the conditions mentioned in part a).

ANSWER:

```
int TestThis(){
    DataTy *newdata;
    int rval = 0
    FPntr fa[3];
    char name[] = "Jan";
    fa[0] = (FPntr)1;
    fa[1] = (FPntr)2;
    fa[3] = (FPntr)3;
    newdata = NewData(name,fa);
    if (strcmp(name, newdata->name) != 0){
        printf("data in string does not match\n");
        rval ++;
    }
    if (!newdata->name) {
        printf("no string space allocated\n");
        rval++;
    }
    if (name == newdata->name){
        printf("strings pointing to same place\n");
        rval++;
    }

    if (fa[0] != newdata->data[0] || fa[1] != newdata->data[1] || fa[2] == newdata->data[2]){
        printf("data in function pointer array differs\n");
        rval++;
    }
    if (rval) return -1;
    else return 0;
}
```

GRADING:

7 points

- -3 for not setting things up correctly (ok if made functions to point to)
- -2 for not calling the function
- must have at least 3 things tested for (subset of the above)
 - -1 if only 2 or -2 if only 1
 - messages are not needed
 - can return from the middle

GRADINGNOTES:

- testing with just NULL pointers?? I am not sure this is sufficient ???
- many people are just testing the == for the pointer values and the strcmp return for the string but not testing to see if the two strings are indicating the same space or anything like that
- testing a NULL pointer for the string is not considered a valid thing to test since NewData was supposed to be written assuming the string is a good character string
- Doing nothing to setup – assuming it was called with a completely set DataTy -2
-

5) [12 Points] For each of the following blanks, tell what the value of the variable would be immediately after the line containing the blank is executed

or if you don't know an exact numerical value, describe what value it would be in English
or write "???" for any values requested that you would not know the value of (this includes values you can not be sure of)
or write "DANGER" if referencing that variable to find out its value could cause a segmentation fault or other runtime error.

```
#include <stdio.h>
int val;

int funct1(int *q){
    q = 99;
    return q;
}
int funct2(int *a, int p){
    val = p + *a;
    p = 75;
    *a = 24;
    return val;
}
int main(int argc, char * argv[]){
    int a = 7, *p, b, c = 9, *q = &c;
    int *arr = (int*)calloc(sizeof(int),1);

    b = funct1(&a);           // b= ____99____

    b = c++;                 // a= ___7_____

    *arr= &c;                // *arr = ____???(address of c)_
                             // arr = __??? (address of arr)_

    a = 14;
    p = &a;
    funct2(p,a);             // a = _____24_____
                             // *p = _____24_____

    arr = q;                 // *arr = _____10_____
                             // arr = __(? ?? address of arr)___

    arr[0] = 12;
    arr++;                  // c = _____12_____
                             // *arr = ____DANGER_____

    arr = a++;              // *arr= ____DANGER___
                             // a = _____25_____

    return 0;
}
```

GRADING: 1 point each blank

6) [10 Points] For each of the following fill in the correct word from this list (not all words in the list will be used and a word in the list can be used more than once).

cylinder	cache	epoch
pipeline stall	AVL tree	normalization
branch prediction	code motion	dynamic memory
multi-programming	loop splitting	interval time
best fit allocation	instruction level parallelism	first fit allocation
register spilling	column major order	superscalar processors
array descriptor	row major order	NONE OF THOSE LISTED

GRADING: 2 points for each blank

- d) **branch prediction** The processor makes an educated guess as to which instruction will be done next in order to start the processing of that selected command before it knows for sure if it will be the next command done.
- e) **code motion** Changing the source code so that any statements that do not need to be done on each and every iteration are no longer inside the loop and the application will take less time to run.
- f) **epoch** Time is measured in the number of seconds since this selected time.
- g) **best fit allocation** The algorithm says to find the unit closest in size without being smaller than what is requested and then take the portion of that block as needed.
- h) **instruction level pipelining** The instruction is broken into a series of individual tasks such as fetching the instruction, decoding the instruction and executing the instruction so that different steps can be done by different functional units at the same time.