Name: Stephen Reilly
CSUN ID: 203764658
Email: stephen.reilly.029@my.csun.edu

**README:** I decided to implement the PCB array, using an array of pointers to PCBs. Each PCB has a linked list to all its children. Create a new PCB, requires you to pass in which parent is creating the new PCB so that the new PCB's parent can be assigned. Destroying a PCB destroys everything but the caller itself. So DestroyManager makes sure everything is deleted when the program terminates. When you print all the PCBs, it also prints the children of the CPB indicated by an indentation and "->" in the terminal screenshot.

Core Methods from Assignment Implemented:

1. Create(PCB), adds a new PCB to the Manager's array
2. Destroy(Manager[0]), Destroys all descendants of 0, but doesn't destroy 0 itself

**Code is written in C. You can copy code below and paste it into IDE with a C compiler. The main method calls the assignment implementation. It prints the following output to terminal:**

# Output from terminal:

```c
int main( ) {
    // create a dynamic array of structs to store all the PCB processes from PCB[0] to PCB[n]
    //  made it a struct so I could include size attribute
    struct PcbManager*  manager = CreatePcbManager();

    //FIRST PCB HAS Manager as PARENT, SO I PASS NULL AS PARRENT, State = Ready (enum)
    CreatePCB( manager, NULL, Ready);                  //create[0]     // create parent process
    CreatePCB( manager, manager->pcbarray[0], Ready);  //create[1]     // creates 1st child of PCB[0] at PCB[1]
    CreatePCB( manager, manager->pcbarray[0], Ready);  //create[2]     // creates 2nd child of PCB[0] at PCB[2]
    CreatePCB( manager, manager->pcbarray[2], Ready);  //create[3]     // creates 1st child of PCB[2] at PCB[3]
    CreatePCB( manager, manager->pcbarray[0], Ready);  //create[4]     // creates 3rd child of PCB[0] at PCB[4]

    printf("\nPRINTING ALL PCB'S AND CHILDREN AFTER CREATION\n");
    PrintAllPcb( manager);

    DestroyPCB( manager->pcbarray[0],manager); // destroy[0] destroys all descendent of PCB[0],which includes processes PCB[1] through PCB[4]
    printf("CALLED DESTROY ON PCB[0] PRINTING WHAT'S LEFT\n");
    PrintAllPcb( manager);

    //free anything left in the manager array, at this point just PCB[0]
    DestroyPcbManager( manager );
    return 0;
}
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL                                  bash

Stephens-MacBook-Pro-2:proj2 stephenreilly$ gcc -g -Wall project2.c -o project2
Stephens-MacBook-Pro-2:proj2 stephenreilly$ ./project2

PRINTING ALL PCB'S AND CHILDREN AFTER CREATION

==================CURRENT PCBs (their children after ->)=================
PCB_ID: 0 | PARENT_ID: -1 | PCB_STATE: Ready | PCB_CHILDREN: 3
   ->PCB_ID: 1 | PARENT_ID: 0 | PCB_STATE: Ready | PCB_CHILDREN: 0
   ->PCB_ID: 2 | PARENT_ID: 0 | PCB_STATE: Ready | PCB_CHILDREN: 1
   ->PCB_ID: 4 | PARENT_ID: 0 | PCB_STATE: Ready | PCB_CHILDREN: 0
PCB_ID: 1 | PARENT_ID: 0 | PCB_STATE: Ready | PCB_CHILDREN: 0
PCB_ID: 2 | PARENT_ID: 0 | PCB_STATE: Ready | PCB_CHILDREN: 1
   ->PCB_ID: 3 | PARENT_ID: 2 | PCB_STATE: Ready | PCB_CHILDREN: 0
PCB_ID: 3 | PARENT_ID: 2 | PCB_STATE: Ready | PCB_CHILDREN: 0
PCB_ID: 4 | PARENT_ID: 0 | PCB_STATE: Ready | PCB_CHILDREN: 0
====================================================================
CALLED DESTROY ON PCB[0] PRINTING WHAT'S LEFT

==================CURRENT PCBs (their children after ->)=================
PCB_ID: 0 | PARENT_ID: -1 | PCB_STATE: Ready | PCB_CHILDREN: 0
====================================================================
```

```c
#include <stdio.h>
#include <stdlib.h>


//initial capacity of PCB array
#define INITIAL_CAPACITY 16

// different possible states for each PCB
typedef enum Pcb_States {
    Ready =0 ,
    Blocked = 1,
    Running = 2
} PcbStates;

//defined in order to translate the enums from number to printable text
//used to translate enums to text when printing out the PCB's state
const char* PcbStateNames[] = {[Ready] = "Ready", [Blocked] = "Blocked", [Running] =
"Running" };

// effectively the PCB array that gets resized as we add too many PCBs
//initalized with 16 elements
struct PcbManager {
    struct PCB** pcbarray;
    int size;
    int capacity;
};

//the pcb itself with id, state, a linked list of children, and a state
struct PCB {
    int indexId;
    struct PCB * parent;
    struct ChildList * children;
    PcbStates state;
};

//the linked list that is assigned to each PCB once it gets a single child
struct ChildList {
    struct ChildNode * head;
    struct ChildNode * tail;
    int size;
};

//a node on the linked list which points to a PCB listed in the array.
struct ChildNode {
    struct ChildNode * next;
    struct ChildNode * prev;
    struct PCB* pcb;
};
```

```
/* This is to create the Manager to store the Manager of PCB processes from PCB[0] to
PCB[n] */
struct PcbManager* CreatePcbManager();

/*adds PCB to the end of the array struct's PCBlist */
void AddPCBtoManager(  struct PcbManager* manager, struct PCB* pcb );

/* only should be called by CreatePCB, if the allocated list is more than the intial
capacity of PCB[n]
   then createPCB calls this method which doubles the capacity of the Manager
*/
void ResizePcbManager( struct PcbManager* );

/*Free all the memory allocated to store the Manager of PCBs */
void DestroyPcbManager( struct PcbManager* Pcb );

/* This creates a new PCB. If argument passed is NULL, then the new PCB is the root
PCB.
If a non null PCB is passed, then the new PCB returned is the child
The PCB parent passed will have an updated child list with the new PCB.
*/
struct PCB* CreatePCB(struct PcbManager* manager, struct PCB *parent, PcbStates
pcbstate );


/* helper function for create PCB,
 adds the newly created PCB (child) to the child list of the parent if it's not null
*/
void AddChildToPCB( struct PCB* parent, struct PCB* child);

/* for the parent PCB, it will recursively remove all the descendants and deallocate
all the memory of each PCB */
void DestroyPCB( struct PCB *parent, struct PcbManager* manager);

/* loops through the Manager of PCB and calls print PCB for the PCB[n] and then for
PCB[n]'s children*/
void PrintAllPcb( struct PcbManager * );

/* prints the PCB index, it's status, and it's state. */
void PrintPcbHeader( struct PCB * );

/*Loop through each child of the pcb and print out all of it's children's info*/
void PrintPcbChildren( struct PCB * singlePcb);


/* This is to create the array to store all the PCB processes from PCB[0] to PCB[n] */
struct PcbManager* CreatePcbManager() {
```

```
    struct PcbManager * manager = malloc( sizeof( struct PcbManager));
    struct PCB ** pcbs = malloc( sizeof( struct PCB *) * INITIAL_CAPACITY);

    manager->pcbarray = pcbs;
    manager->capacity = INITIAL_CAPACITY;
    manager->size =  0;

    return manager;
}

/* only should be called by CreatePCB, if the allocated list is more than the intial
capacity of PCB[n]
   then createPCB calls this method which doubles the capacity of the array
*/
void ResizePcbManager( struct PcbManager* manager ) {
    //double capacity and copy array to new array
    int capacity = manager->capacity;
    manager->capacity *=2;

    struct PCB** resized_arry = malloc( sizeof(struct PCB*) * manager->capacity );

    for( int i = 0; i < capacity; i++ ) {
        resized_arry[i] = manager->pcbarray[i];
    }
    free( manager->pcbarray );
    manager->pcbarray = resized_arry;
}

/*Free all the memory allocated to store the array of PCBs */
void DestroyPcbManager( struct PcbManager* manager ){
    for( int i = 0; i < manager->size; i++) {
        //if manager array slot is null, it means the PCB allocated to that spot has
already been freed by some parent of it
        if( manager->pcbarray[i] != NULL ){
            //recursively free all the dynmically allocated children and their PCB
decendants;
            DestroyPCB( manager->pcbarray[i], manager );

            //destroy the parent itself
            free(manager->pcbarray[i]);
        }
        //printf( "after free PCB \n");
    }
    //free array of pointer structs allocated to manage the pcb
    free( manager->pcbarray);

    //free the pcb struct itself
```

```
    free( manager);
    manager = NULL;
}

/*adds PCB to the end of the array struct's PCBlist */
void AddPCBtoManager(  struct PcbManager* manager, struct PCB* pcb ) {

    //if we've run out of space in the originally allocated array. copy all PCB to new
bigger array
    if( manager->size  == manager->capacity ){
        ResizePcbManager( manager);
    }
    //printf( "adding to manager at p[%d]\n", pcb->indexId);

    manager->pcbarray[pcb->indexId] = pcb;

   // printf( "printing state at manager: %d \n", manager->pcbarray[pcb->indexId]-
>state);

    manager->size = manager->size + 1;

}

/* This creates a new PCB. If argument passed is NULL, then the new PCB is the root
PCB.
If a non null PCB is passed, then the new PCB returned is the child
The PCB parent passed will have an updated child list with the new PCB.
*/
struct PCB* CreatePCB( struct PcbManager* manager, struct PCB *parent, PcbStates
pcbstate ){

    struct PCB* newpcb = malloc( sizeof(struct PCB ) );
    //printf( "after ready state %d \n", pcbstate);
    newpcb->state = pcbstate;
    newpcb->indexId = manager->size;

    //struct ChildList * children = malloc( sizeof(struct ChildList ));
    //ONLY allocate child list if the parent actually is going to have children
    newpcb->children = NULL;

    //if the parent is not null, then the parent has created this PCB, so the pCB
should be added to parent's child list
    if( parent != NULL){
        newpcb->parent = parent;
        //printf( "parent is not null \n");
        //helper function to add pcb to parent's linked list
        AddChildToPCB( parent, newpcb);
    }
```

```
    //we need to add the newly created PCB to the PCB manager (ie array of PCBs)
    AddPCBtoManager(manager, newpcb);

    return newpcb;
}

//add the given child to the children list of the parent pcb
void AddChildToPCB( struct PCB* parent, struct PCB* child){
    //need to create a new node to add to the parent's linked list
    struct ChildNode * node = malloc( sizeof(struct ChildNode ));
    node->pcb = child;
    node->next = NULL;
    node->prev = NULL;


    //only allocate a child list for parent if parent doesn't have one already
    if( parent->children == NULL ){
        parent->children = malloc( sizeof(struct ChildList ));
        parent->children->head = NULL;
        parent->children->tail = NULL;
        parent->children->size = 0;
    }

    //if parent has no children yet we need to update the head and tail
    if( parent->children->size == 0){
        parent->children->head = node;
        parent->children->tail = node;
    } else if(parent->children->size == 1 ){
        //parent has one child, so we need to update the prev and next
        node->prev = parent->children->head;
        parent->children->tail = node;
        parent->children->head->next =  node;
    } else{
        //parent has multiple childs so we only need to update at the tail
        parent->children->tail->next = node;
        node->prev = parent->children->tail;
        parent->children->tail = node;
    }
    parent->children->size = parent->children->size +1;
}


/* Recursively destroy the descendants of a PCB, leaving the parent intact */
void DestroyPCB(struct PCB *parent, struct PcbManager* manager) {

    // BASE CASE if parent no longer has descendants the destroy is finished
    if ( parent->children == NULL) {
```

```
        //printf( "base case free PCB_ID children: %d | \n", parent->indexId);
        return;
    }
    //printf( "PCB_ID to delete: %d | \n", parent->children->head->pcb->indexId);

    struct ChildNode *childNode = parent->children->head;
    struct ChildNode *cleanUp = childNode;
    while (childNode != NULL) {
        //set array to null so that Manager knows location is free and existing PCB is
cleared
        manager->pcbarray[childNode->pcb->indexId] = NULL;

        // Recursively destroy the current child PCB and its descendants
        DestroyPCB(childNode->pcb, manager);
        //Use this to free previous head
        cleanUp =childNode;
        //advance to the next child to delete/recurse over
        childNode = childNode->next;
        // Update the head of the child list since we've deleted the previous
        parent->children->head = childNode;

        //free the pcb of the child
        free( cleanUp->pcb);
        //free the node of the parent linked list to the child
        free(cleanUp);
    }

    //for the parents that had children (including caller), we need to free it's
children link list
    struct ChildList * cleanupChildList = parent->children;
    parent->children = NULL;
    free(cleanupChildList);
}


/* loops through the array of PCB and calls print PCB for the PCB[n] and then for
PCB[n]'s children*/
void PrintAllPcb( struct PcbManager * manager ){

    if( manager == NULL) { return; }

    printf( "\n==================CURRENT PCBs (their children after -
>)=================\n");
    for( int i = 0; i < manager->size; i++) {
        //since when we delete we set the manager array spot to 0, but don't update
the size for reuse
        //we need to check if it's null. the reason we don't reuse spots is for index
creation consistency
```

```
        if( manager->pcbarray[i] != NULL) {
            PrintPcbHeader( manager->pcbarray[i]);
            PrintPcbChildren( manager->pcbarray[i]);
        }
    }
    printf(
"================================================================================\n");
}

/* prints the PCB index, it's status, and it's state. */
void PrintPcbHeader( struct PCB * singlePcb ){

    //check if parent is NULL (and therefore the root), if it is than the parent ID is
-1;
    int parentID  = singlePcb->parent == NULL ? -1 : singlePcb->parent->indexId;
    int size = singlePcb->children == NULL ? 0 : singlePcb->children->size;

    printf( "PCB_ID: %d | PARENT_ID: %d | PCB_STATE: %s | PCB_CHILDREN: %d\n",
        singlePcb->indexId,
        parentID,
        PcbStateNames[singlePcb->state], // convert enum, to text name using an array
of possible names for each enum number.
        size
    );
}


void PrintPcbChildren( struct PCB * singlePcb) {

    //no children so nothing to print.
    if( singlePcb == NULL || singlePcb->children == NULL) {
        return;
    }

    //get the first child of the PCB
    struct ChildNode* child = singlePcb->children->head;

    //loop and print PCB headers of all the children
    while( child != NULL ){
        printf("   ->");
        PrintPcbHeader( child->pcb);
        child = child->next;
    }
}



int main( ) {
```

```
    // create a dynamic array of structs to store all the PCB processes from PCB[0] to
PCB[n]
    //  made it a struct so I could include size attribute
    struct PcbManager*  manager = CreatePcbManager();

    //FIRST PCB HAS Manager as PARENT, SO I PASS NULL AS PARRENT, State = Ready (enum)
    CreatePCB( manager, NULL, Ready);                  //create[0]     // create
parent process
    CreatePCB( manager, manager->pcbarray[0], Ready);  //create[1]     // creates 1st
child of PCB[0] at PCB[1]
    CreatePCB( manager, manager->pcbarray[0], Ready);  //create[2]     // creates 2nd
child of PCB[0] at PCB[2]
    CreatePCB( manager, manager->pcbarray[2], Ready);  //create[3]     // creates 1st
child of PCB[2] at PCB[3]
    CreatePCB( manager, manager->pcbarray[0], Ready);  //create[4]     // creates 3rd
child of PCB[0] at PCB[4]

    printf("\nPRINTING ALL PCB'S AND CHILDREN AFTER CREATION\n");
    PrintAllPcb( manager);

    DestroyPCB( manager->pcbarray[0],manager); // destroy[0] destroys all descendent
of PCB[0],which includes processes PCB[1] through PCB[4]
    printf("CALLED DESTROY ON PCB[0] PRINTING WHAT'S LEFT\n");
    PrintAllPcb( manager);

    //free anything left in the manager array, at this point just PCB[0]
    DestroyPcbManager( manager );
    return 0;
}
```