Lab Assignment 1 Report: Process Creation Hierarchy

Erik Gonzalez
COMP 322\L Operating Systems
Professor: David Freedman
September 27, 2025

This report explains my C program in the order it actually runs: starting from `main`, through the menu loop, selection dispatch, array initialization, process creation, descendant destruction, and quitting/freeing memory.

main

```c
int main(void){
    pcb *dynaArr = NULL;
    int max = 0;
    int sel;
    do {
        sel = getSel();
        selFunc(sel, &dynaArr, &max);
    } while (sel != 4);
    return 0;
}
```

The `main` function is the entry point of the program.

- **PCB array pointer:** It begins by declaring `pcb *dynaArr = NULL`. This pointer will eventually hold the address of the dynamically allocated PCB array, but it starts as `NULL` to signal that no array has been created yet.
- **Maximum size:** It declares `int max = 0`, which will store the maximum number of processes chosen by the user when option 1 (Enter parameters) is selected. Until then, the value remains 0.
- **Menu selection:** It declares `int sel`, which will keep track of the user's most recent menu choice.
- **Looping structure:** The `do-while` loop ensures that the menu will always be displayed at least once. Inside the loop:
  - `getSel()` is called to print the menu and return the user's choice.
  - The user input, along with the PCB array pointer and max size, is then passed to `selFunc()`, which holds a switch statment to select the correct function (initialize, create, destroy, or quit).
- **Exit:** The loop continues to run until the user enters option 4. At that point, `selFunc()` handles cleanup by freeing all memory, and the loop condition `(sel != 4)` fails, exiting the program.

menuFunc

```
void menuFunc(void){
    puts("\nProcess creation and destruction");
    puts("-----------------------------------");
    puts("1) Enter parameters");
    puts("2) Create a new child process");
    puts("3) Destroy all descendants of a process");
    puts("4) Quit program and free memory\n");
}
```

The `menuFunc` function prints the program's user menu.

- It prints four options:
    1. Enter parameters (to set the maximum number of processes).
    2. Create a new child process.
    3. Destroy all descendants of a process.
    4. Quit program and free memory.
- The function uses `puts()` to output. I chose puts as it was faster and easier to write than printf for this use case.
- It does not return any value

getSel

```
int getSel(void){
    int sel;
    menuFunc();
    printf("Enter selection: ");
    if (scanf("%d", &sel) != 1) return 4;
    return sel;
}
```

The `getSel` calls the menu function (menuFunc), and asks for the user's input

- Calls `menuFunc()` to display the menu options are printed before asking for input.
- Variable `sel`, stores the user's input.
- Prompts the user with `"Enter selection:"`.
- `scanf()` to read an integer into `sel`. Also includes a check to make sure user input is within bounds.

selFunc

```c
void selFunc(int sel, pcb **arr, int *pmax){
    switch (sel) {
        case 1: {
            int newMax = maxProc();
            freeMem(*arr, *pmax);
            *arr = initArr(newMax);
            *pmax = newMax;
            break;
        }
        case 2:
            if (!*arr) { printf("Pick 1 first.\n"); break; }
            create(*arr, *pmax);
            break;
        case 3:
            if (!*arr) { printf("Pick 1 first.\n"); break; }
            destroyDesc(*arr, *pmax);
            break;
        case 4:
            printf("Quitting program...\n");
            freeMem(*arr, *pmax);
            *arr = NULL;
            *pmax = 0;
            break;
        default:
            printf("Invalid selection.\n");
    }
}
```

The `selFunc` function takes the user's menu selection and uses a `switch` statement to run the correct logic:

- **Case 1:** Calls `maxProc()` to read a max process count from the user. It then frees any existing array with `freeMem`, initializes a new PCB array with `initArr`, and updates the max size.
- **Case 2:** Ensures that the PCB array has been initialized. If so, it calls `create()` to add a new child process to the hierarchy.
- **Case 3:** Ensures that the PCB array has been initialized. If so, it calls `destroyDesc()` to recursively delete all descendants of a given process.
- **Case 4:** Prints "Quitting program...", frees all memory with `freeMem`, and resets pointers to `NULL` before exiting the program loop.
- **Default:** Handles invalid user input by printing an error message.

maxProc

```c
int maxProc(void){
    int max;
    printf("Enter the maximum number of processes: ");
    if (scanf("%d", &max) != 1 || max < 1) max = 1;
    return max;
}
```

The `maxProc` function asks the user for the max amount of processes.

- Declares the integer variable `max`.
- Prompts the user to enter the maximum number of processes.
- Uses `scanf()` to attempt to read an integer and has a input bounds check
- Returns the max value so that memory for the PCB array can be allocated.

initArr

```c
pcb* initArr(int max){
    pcb *dynaArr = malloc(max * sizeof(pcb));
    if (!dynaArr) { perror("malloc"); exit(1); }
    for (int i = 0; i < max; i++) {
        dynaArr[i].isFree = 0;
        dynaArr[i].parent = -1;
        dynaArr[i].children = NULL;
    }
    dynaArr[0].isFree = 1;
    dynaArr[0].parent = -1;
    return dynaArr;
}
```

The `initArr` function sets up the PCB array for all processes.

- Dynamically allocates memory for an array of `pcb` structures
- Loops over every index in the array:
  - Marks each PCB as free by setting `isFree = 0`.
  - Resets the parent index to `-1` (no parent).
  - Initializes the children pointer to `NULL`.
- iniitializes `PCB[0]` as the root process
- Returns the pointer to the newly allocated array so it can be used for the program.

create

```c
void create(pcb *arr, int max){
    if (!arr) { printf("Not initialized.\n"); return; }
    int parentIndex;
    printf("Enter the parent process index: ");
    if (scanf("%d", &parentIndex) != 1) { printf("Bad input.\n"); return; }
    if (parentIndex < 0 || parentIndex >= max || arr[parentIndex].isFree ==
    0) {
        printf("Invalid parent index.\n");
        return;
    }
    int childIndex = -1;
    for (int i = 1; i < max; i++) {
        if (arr[i].isFree == 0) { childIndex = i; break; }
    }
    if (childIndex == -1) { printf("No free PCB available.\n"); return; }
    arr[childIndex].isFree = 1;
    arr[childIndex].parent = parentIndex;
    arr[childIndex].children = NULL;
    child *node = malloc(sizeof(child));
    if (!node) { perror("malloc"); exit(1); }
    node->childIndex = childIndex;
    node->next = NULL;
    if (arr[parentIndex].children == NULL) {
        arr[parentIndex].children = node;
    } else {
        child *cur = arr[parentIndex].children;
        while (cur->next) cur = cur->next;
        cur->next = node;
    }
    printParentChildren(arr, max);
}
```

The `create` function adds a new process to the hierarchy as a child of an existing parent.

- Checks to make sure the PCB array is initialized
- Prompts for user input for a parent process index.
- Searches the PCB array for the first available free slot
- Initializes the new child PCB by marking it active, recording its parent, and setting its children list to `NULL`.
- Allocates a new `child` node and sets its index to the new child PCB.
- Appends this node to the parent's child linked lists
- Calls `printParentChildren()` to print the hierarchy.

printParentChildren

```
void printParentChildren(pcb *arr, int max){
    for (int i = 0; i < max; i++) {
        if (arr[i].isFree && arr[i].children) {
            printf("PCB[%d] is the parent of: ", i);
            child *c = arr[i].children;
            while (c) {
                printf("PCB[%d] ", c->childIndex);
                c = c->next;
            }
            printf("\n");
        }
    }
}
```

The `printParentChildren` function displays the parent–child relations in the PCB hierarchy:

- Loops through every PCB in the dynamic array.
- For each PCB that is active and has at least one child, it prints a line showing that PCB's index as a parent.
- Traverses the linked list of children for that parent and prints each child's index in the order.

destroyDesc

```c
void destroyDesc(pcb *arr, int max){
    if (!arr) { printf("Not initialized.\n"); return; }
    int p;
    printf("Enter the index of the process whose descendants are to be
   destroyed: ");
    if (scanf("%d", &p) != 1) { printf("Bad input.\n"); return; }
    if (p < 0 || p >= max || arr[p].isFree == 0) {
        printf("Invalid process index.\n");
        return;
    }
    destroySubTree(arr, max, p);
    printParentChildren(arr, max);
}
```

The `destroyDesc` function asks as the user for the index of the process, which will result in the descendants from that index to be destroyed recursively.

- Prompts the user for a process index `p`.
- Calls `destroySubTree` on that index, which performs the recursive destruction of all descendants.

—

destroySubTree

```c
void destroySubTree(pcb *arr, int max, int p){
    child *cur = arr[p].children;
    while (cur) {
        int q = cur->childIndex;
        destroySubTree(arr, max, q);
        arr[q].isFree = 0;
        arr[q].parent = -1;
        arr[q].children = NULL;
        child *next = cur->next;
        free(cur);
        cur = next;
    }
    arr[p].children = NULL;
}
```

The `destroySubTree` function performs the recursive deletion of descendants:

- Begins with the linked list of children belonging to process `p`.
- For each child node:
  - Recursively calls `destroySubTree` to ensure all of that child's descendants are deleted first.
  - Marks the child PCB as free by resetting `isFree`, `parent`, and `children`.
  - Frees the current child node from the parent's linked list.
- After all children are processed, sets `arr[p].children = NULL`, leaving the parent with no descendants.

freeMem

```
void freeMem(pcb *arr, int max){
    if (!arr) return;
    for (int i = 0; i < max; i++) {
        child *c = arr[i].children;
        while (c) { child *n = c->next; free(c); c = n; }
        arr[i].children = NULL;
    }
    free(arr);
}
```

On quitting, this function deallocates all child lists and frees the PCB array.