Tutorial 8: **Signals and Data Structures Part II**

Faculty of Engineering and Applied Science

SOFE 3950U: Operating Systems | CRN: 74171

Due: March 18th, 2024

Group 8

Daniel Amasowomwan [100787640]
daniel.amasowomwan@ontariotechu.net

Stanley Watemi [100648403]
stanley.watemi@ontariotechu.net

Fayomi Toyin [100765921]
oluwatoyin.fayomi@ontariotechu.net

Conceptual Questions

1. What is an Abstract Data Type (ADT)?
    - Abstract Data type (ADT) is a type (or class) for objects whose behavior is defined by a set of values and a set of operations.

2. Explain the difference between a queue (FIFO) and a stack (LIFO).
    - A stack is a linear data structure in which elements can be inserted and deleted only from one side of the list, called the top while a queue is a linear data structure in which elements can be inserted only from one side of the list called rear, and the elements can be deleted only from the other side called the front.

3. Name and briefly explain three types of data structures.
    - Heap - A heap is a tree-based structure in which each parent node's associated key value is greater than or equal to the key values of any of its children's key values.
    - Tree - A tree, also known as a keyword tree, is a data structure that stores strings as data items that can be organized in a visual graph.
    - Linked list - A linked list stores a collection of items in a linear order. Each element, or node, in a linked list contains a data item, as well as a reference, or link, to the next item in the list.

4. Explain what a binary tree is, what are some common operations of a binary tree?
    - A binary tree is a tree data structure comprising of nodes with at most two children i.e. a right and left child. The node at the top is referred to as the root. A node without children is known as a leaf node. Some common operations that can be conducted on binary trees include insertion, deletion, and traversal.

5. Explain what a hash table (dictionary) is, what are common operations of a hash table?
    - Hash table - A hash table -- also known as a hash map -- stores a collection of items in an associative array that plots keys to values. A hash table uses a hash function to convert an index into an array of buckets that contain the desired data item. Hash tables are used to quickly store and retrieve data (or records).

Application problems

1. Create a program that does the following.
    - Create a structure called **proc** that contains the following
        - **parent,** character array of 256, name of the parent process
        - **name**, character array of 256 length
        - **priority,** integer for the process priority
        - **memory**, integer for the memory in MB used by process
    - Create a binary tree data structure called **proc_tree** which contains the proc data structure.
    - Create the necessary functions to interact with your binary tree data structure, you will need to add items to your tree and iterate through it.
    - Your program then reads the contents of a file called **process_tree.txt (7 LINES)**, which contains a **comma separated** list of the parent, name, priority, and memory.
    - Read the contents of the file and create your binary tree, add the children to the parent based on the name of the parent.
    - Print the contents of your binary tree (you likely need to use **recursion**!) displaying the contents of each parent, and the children of each parent.

```
#include <stddef.h>
```

```
#include <stdlib.h>

#include <stdio.h>

#include <stdbool.h>

#include <unistd.h>

#include <signal.h>

#include <sys/types.h>

#include <sys/wait.h>

#include <string.h>

// Define a structure for process

typedef struct {

    char parent[256];

    char name[256];

    int priority;

    int memory;

} proc;
```

```c
// Define a structure for binary tree node

typedef struct node {

    proc data;

    struct node *left;

    struct node *right;

} node;

// Function to create a new node

node *createNode(proc data) {

    node *newNode = (node *)malloc(sizeof(node));

    if (newNode == NULL) {

        printf("Memory allocation failed\n");

        exit(EXIT_FAILURE);

    }

    newNode->data = data;

    newNode->left = newNode->right = NULL;

    return newNode;

}

// Function to insert a node into the binary tree

node *insertNode(node *root, proc data) {

    if (root == NULL)

        return createNode(data);

    // Traverse left or right based on comparison with parent name

    if (strcmp(data.parent, root->data.name) < 0)

        root->left = insertNode(root->left, data);

    else

        root->right = insertNode(root->right, data);

    return root;

}

// Function to print the binary tree using inorder traversal

void printTree(node *root) {
```

```c
    if (root != NULL) {

        printTree(root->left);

        printf("Parent: %s, Name: %s, Priority: %d, Memory: %d\n", root->data.parent, root->data.name,
root->data.priority, root->data.memory);

        printTree(root->right);

    }

}

int main(void) {

    FILE *file = fopen("processes_tree.txt", "r");

    if (file == NULL) {

        printf("Failed to open file\n");

        return EXIT_FAILURE;

    }

    // Initialize root of the binary tree

    node *root = NULL;

    char line[512];

    // Read file line by line

    while (fgets(line, sizeof(line), file) != NULL) {

        proc newProc;

        sscanf(line, "%[^,],%[^,],%d,%d", newProc.parent, newProc.name, &newProc.priority, &newProc.memory);

        root = insertNode(root, newProc);

    }

    fclose(file);



    // Print the contents of the binary tree

    printf("Binary Tree Contents:\n");

    printTree(root);

    return 0;

}
```

```
danielamas@Linux22:~/school/opsystems/tutorials/tut8$ ./q1
Binary Tree Contents:
Parent: NULL, Name:  kernel, Priority: 0, Memory: 128
Parent: kernel, Name:  bash, Priority: 1, Memory: 64
Parent: kernel, Name:  zsh, Priority: 1, Memory: 64
Parent: bash, Name:  sublime, Priority: 3, Memory: 256
Parent: bash, Name:  gedit, Priority: 3, Memory: 128
Parent: zsh, Name:  eclipse, Priority: 3, Memory: 1024
Parent: zsh, Name:  chrome, Priority: 3, Memory: 2048
```

2. Create a simple host dispatch shell that does the following.
    - Create a structure called **proc** that contains the following
        - **name**, character array of 256 length
        - **priority,** integer for the process priority
        - **pid**, integer for the process id
        - **address** integer index of memory in **avail_mem** allocated
        - **memory,** integer for the memory required
        - **runtime,** integer for the running time in seconds
        - **suspended,** boolean indicating process has been suspended
    - Create a **FIFO** queue called **priority** which will be populated with real time priority processes (priority 0).
    - Create a second **FIFO** queue called **secondary,** which will be populated with secondary priority processes.
    - Create an array of **length 1024** called **avail_mem**, use #define MEMORY 1024, **initialize it to 0** to indicate all memory is free.
    - Read in the processes from the file **processes_q2.txt**, the file contains a comma separated list of the **name, priority, memory,** and **runtime** you must initialize the **pid and address to 0** in your process structure, it is set when you execute the processes.
    - When reading the file **processes_q2.txt** add each process with a priority of 0 to the **priority** queue, add the remaining processes to the **secondary** queue.
    - Iterate through all of the processes in the **priority** queue first, **pop()** each item from the queue and execute the **process** binary using fork and exec.
        - Mark the memory needed in the **avail_mem** array as used **(1)**, set the **address** member of the struct to the starting index where the memory is allocated in the avail_mem array.
        - Before **process** is executed, print the **name, priority, pid, memory, and runtime** of the process.
        - Run the process for the specified **runtime** and then send it the signal **SIGTSTP** to terminate it.

    - Ensure that you use the **waitpid** function to wait until the process has terminated.
    - Free the memory in **avail_mem** used by the process **(set the array entries to 0).**
- Then iterate through all of the processes in the **secondary** queue, **pop()** each item from the queue and execute the **process** binary using fork and exec.
    - If there is **enough memory** available in **avail_mem** array then proceed, otherwise **push()** it back on the queue.
    - Mark the memory needed in the **avail_mem** array as used **(1)**, set the **address** member of the struct to the starting index where the memory is allocated in the avail_mem array.
    - Before **process** is executed, print the **name, priority, pid, memory, and runtime** of the process.
    - **If** the process has already been suspended (**suspended** is true, and **pid** set) then send **SIGCONT** to the process to resume it.
    - Run the process for **1 second** then send **SIGTSTP** to the process to suspend it.
    - If the process was just created, set the **pid** member in the process struct that was returned from **pop()** to the process id returned from **exec().**
    - Decrement the **runtime** member in the process struct by **1.**
    - Set the **suspended** member in the process struct to **true**, indicating the processes has been suspended.
    - Add the process back to the **secondary** queue using **push()**
    - Repeat this for every process in the **secondary queue.**
- For any item in the **secondary** queue that **only has 1 second** of **runtime** left
    - Run the process for the specified **runtime** and then send it the signal **SIGINT** to terminate it.
    - Ensure that you use the **waitpid** function to wait until the process has terminated.
    - **Do not** add the process back to the queue.
    - Free the memory in **avail_mem** used by the process **(set the array entries to 0).**
- Once all of the processes have been executed the main program terminates.

```
danielamas@Linux22:~/school/opsystems/tutorials/tut8$ ./q2
Executing process:
Name: systemd
Priority: 0
Memory: 256
Runtime: 5
Executing systemd
systemd completed.
Executing process:
Name: bash
Priority: 0
Office Writer
Runtime: 8
Executing bash
bash completed.
Executing process:
Name: vim
Priority: 3
Memory: 128
Runtime: 4
Executing vim
vim completed.
```

```
Executing process:
Name: emacs
Priority: 3
Memory: 256
Runtime: 4
Executing emacs
emacs completed.
Executing process:
Name: chrome
Priority: 1
Memory: 512
Runtime: 2
Executing chrome
chrome completed.
Executing process:
Name: chrome
Priority: 1
Memory: 512
Runtime: 3
Executing chrome
chrome completed.
```

```
Executing chrome
chrome completed.
Executing process:
Name: chrome
Priority: 1
Memory: 1024
Runtime: 5
Executing chrome
chrome completed.
Executing process:
Name: gedit
Priority: 2
Memory: 128
Runtime: 4
Executing gedit
gedit completed.
Executing process:
Name: eclipse
Priority: 2
Memory: 1024
Runtime: 3
Executing eclipse
eclipse completed.
```

```
Executing process:
Name: clang
Priority: 1
Memory: 512
Runtime: 3
Executing clang
clang completed.
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <string.h>

#define MEMORY 1024

typedef struct {
    char name[256];
    int priority;
    int pid;
    int address;
    int memory;
    int runtime;
    bool suspended;
} Proc;

typedef struct {
    Proc array[MEMORY];
    int front, rear, size;
} Queue;

void initializeQueue(Queue *q) {
    q->front = q->rear = q->size = 0;
}

bool isEmpty(Queue *q) {
    return q->size == 0;
}

void push(Queue *q, Proc data) {
    if (q->size == MEMORY) {
        printf("Queue is full\n");
        return;
    }
    q->array[q->rear++] = data;
    q->rear %= MEMORY;
```

```c
    q->size++;
}

Proc pop(Queue *q) {
    if (isEmpty(q)) {
        printf("Queue is empty\n");
        exit(EXIT_FAILURE);
    }
    Proc data = q->array[q->front++];
    q->front %= MEMORY;
    q->size--;
    return data;
}

void executeProcess(Proc process, int avail_mem[]) {
    printf("Executing process:\n");
    printf("Name: %s\n", process.name);
    printf("Priority: %d\n", process.priority);
    printf("Memory: %d\n", process.memory);
    printf("Runtime: %d\n", process.runtime);

    pid_t pid = fork();
    if (pid < 0) {
        perror("Fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) { // Child process
        // Execute process binary here using exec
        printf("Executing %s\n", process.name);
        sleep(process.runtime);
        printf("%s completed.\n", process.name);
        exit(EXIT_SUCCESS);
    } else { // Parent process
        process.pid = pid;
        process.address = -1; // Update with the actual memory address
        avail_mem[process.address] = 1; // Mark memory as used
        waitpid(pid, NULL, 0); // Wait for child process to finish
        avail_mem[process.address] = 0; // Free the memory
    }
}

int main() {
    Queue priority, secondary;
    initializeQueue(&priority);
    initializeQueue(&secondary);
    int avail_mem[MEMORY] = {0};

    FILE *file = fopen("processes_q2.txt", "r");
    if (!file) {
        printf("Error opening file.\n");
        return EXIT_FAILURE;
    }

    Proc temp;
    while (fscanf(file, "%255[^,], %d, %d, %d\n", temp.name, &temp.priority, &temp.memory, &temp.runtime) == 4) {
        temp.pid = 0;
        temp.address = 0;
        temp.suspended = false;
        if (temp.priority == 0)
            push(&priority, temp);
        else
            push(&secondary, temp);
    }
    fclose(file);

    while (!isEmpty(&priority)) {
```

```c
        Proc process = pop(&priority);
        executeProcess(process, avail_mem);
    }

    while (!isEmpty(&secondary)) {
        Proc process = pop(&secondary);
        // Check if enough memory is available
        if (process.memory <= MEMORY) {
            executeProcess(process, avail_mem);
        } else {
            printf("Insufficient memory for %s, pushing it back to the queue.\n", process.name);
            push(&secondary, process);
        }
    }

    return 0;
}
```