Faculty of Engineering and Applied Science

SOFE 3950U Tutorial 8

March , 2024

Group Members

Inder Singh – 100816726

Justin Fisher – 100776303

Okiki Ojo – 100790236

Conceptual Questions

1. Abstract Data type (ADT) is an abstraction of a data structure that provides only the interface for a data structure. The interface does not provide any specific details about how something should be implemented.

2. A stack is a data structure where the elements are inserted or deleted (pushed or popped) from one side, or the top of the list. It follows a LIFO principle, meaning that the last element to be added to the stack is the first element to be removed. A queue however, is a data structure where elements are inserted from one end of the list and removed from the opposite end. Each element is inserted from the rear of the list and elements are only deleted from the other side, or the front of the list. It follows a FIFO principle, meaning that the first element inserted into the queue is supposed to be the first element when remove is called.

3. A static data structure is a type of data structure which has a fixed memory size. An example of this data structure is an array, which stores a collection of elements of the same data type in fixed memory locations. Another data structure type is a dynamic data structure, where the size is not fixed. The size can be randomly updated during runtime, which is efficient for space complexity. These include queues, stacks and linked lists. The last type of data structure is a non-linear data structure, where data elements are not placed sequentially or linearly. Each element can connect to many other elements, rather than forming a linear sequence. This includes graphs and trees. Graphs consist of a set of nodes connected by edges, and trees represent more of a hierarchical structure where each parent node contains a set of child nodes.

4. A binary tree is a tree where each node only has at most two children. Some common operations on a binary tree include insertion, where a node is inserted while still maintaining the binary tree properties. Deletion removes a node while still maintaining the binary tree properties. Traversal is used to visit all the nodes in specific orders. Inorder traversal visits the left subtree, the root node, and the right subtree. Preorder traversal visits the root node, the left subtree, and the right subtree. Postorder traversal visits the left subtree, the right subtree, and then the root. The maximum element operation returns the biggest element in the binary tree. The minimum element operation returns the smallest element in the binary tree. Another operation is to find the height of the binary tree. And lastly, another operation performed is to ensure the tree remains balanced.

5. A hash table stores certain values with a certain key, and this key is determined with a specific hashing function to generate a unique key. It then maps these keys to indexes in an array. This key is then used to search for the value. Common operations include inserting values, searching for values, deleting values, and updating values.

Application Questions

1.

```c
#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>

#include "tree.h"
#include "queue.h"
#include "process.h"

int main() {
    process_t data; // Temporary variable to hold the data read from the file.
    // Attempt to open the file containing process information.
    FILE *file = fopen("processes_tree.txt", "r");
    if (file == NULL) {
        fprintf(stderr, "File not found\n");
        exit(EXIT_FAILURE);
    }

    // Root node of the binary tree.
    tree_t* root = NULL;

    // Read each line from the file and insert the process data into a queue.
    while (fscanf(file, "%[^,], %[^,], %d, %d\n", data.parent, data.name, &data.priority, &data.memory) == 4) {
        if (strcmp(data.parent, "NULL") == 0) {
            root = create_node(data);
        } else {
            root = insert_proc(root, data);
        }
    }

    // Close the file after reading the contents.
    fclose(file);

    // Print the tree
    printf("Binary Tree Contents:\n");
    print_tree(root, 0);

    // Free the allocated memory for the binary tree.
    free_proc_tree(root);
    return 0;
}
```

```c
#ifndef PROCESS_H_
#define PROCESS_H_

#define MAX_NAME_SIZE 256

// Process struct, stores the process state
typedef struct process_t {
    char parent[MAX_NAME_SIZE]; // Name of the parent process.
    char name[MAX_NAME_SIZE];   // Name of the process.
    int priority;               // Priority of the process.
    int memory;                 // Memory in MB used by the process.
} process_t;

#endif /* PROCESS_H_ */
```

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

#ifndef TREE_H_
#define TREE_H_

// Tree struct, stores the process state
typedef struct tree_t {
    process_t process;
    struct tree_t *left;
    struct tree_t *right;
} tree_t;

extern tree_t* create_node(process_t new_proc);
extern tree_t* insert_proc(tree_t* node, process_t new_proc);
extern void print_tree(tree_t* node, int level);
extern void free_proc_tree(tree_t *tree);

#endif /* TREE_H_ */
```

```c
#include <string.h>
#include "tree.h"

/**
 * Creates a new tree node with the given process data.
 *
 * @return Pointer to the newly created tree node.
 */
extern  tree_t* create_node(process_t new_proc) {
    tree_t* newNode = (tree_t*)malloc(sizeof(tree_t));
    if (newNode == NULL) {
        fprintf(stderr, "Memory allocation error\n");
        exit(1);
    }
    newNode->process = new_proc;
    newNode->left = newNode->right = NULL;
    return newNode;
}

/**
 * Inserts a new node into the tree.
 *
 * @param tree The tree to insert the new node into.
 * @param process The process data to be stored in the new node.
 * @return Pointer to the newly created tree node.
 */
extern tree_t* insert_proc(tree_t* node, process_t new_proc) {
    if (node == NULL) {
        return create_node(new_proc);
    }
    // Simplified comparison; real-world applications may require more complex logic
    if (strcmp(new_proc.parent, node->process.name) == 0) {
        if (node->left == NULL) {
            node->left = create_node(new_proc);
        } else {
            node->right = create_node(new_proc);
        }
    } else {
        insert_proc(node->left, new_proc);
        insert_proc(node->right, new_proc);
    }
    return node;
}

/**
 * Recursively prints the binary tree, showing each process's name, priority, and memory usage.
 *
 * @param tree The root of the binary tree to print.
 * @param space The indentation level for pretty printing.
 */

// Print the tree
extern void print_tree(tree_t* node, int level) {
    if (node != NULL) {
        for (int i = 0; i < level; i++) {
            printf("  ");
        }
        printf("%s (Priority: %d, Memory: %dMB)\n", node->process.name, node->process.priority, node->process.memory);
        print_tree(node->left, level + 1);
        print_tree(node->right, level + 1);
    }
}

/**
 * Frees the memory allocated for the binary tree.
 *
 * @param tree The root of the binary tree to free.
 */
extern void free_proc_tree(tree_t *tree) {
    if (tree == NULL) {
        return;
    }
    free_proc_tree(tree->left);
    free_proc_tree(tree->right);
    free(tree);
}
```

```
root@Okiki-PC ➜ Tutorial 8 gcc -Wall -Wextra -std=c99 q1.c tree.c tree.h process.h queue.c queue.h -o q1
root@Okiki-PC ➜ Tutorial 8 ./q1
Binary Tree Contents:
kernel (Priority: 0, Memory: 128MB)
  bash (Priority: 1, Memory: 64MB)
    sublime (Priority: 3, Memory: 256MB)
    gedit (Priority: 3, Memory: 128MB)
  zsh (Priority: 1, Memory: 64MB)
    eclipse (Priority: 3, Memory: 1024MB)
    chrome (Priority: 3, Memory: 2048MB)
root@Okiki-PC ➜ Tutorial 8 git:(main)
```

2.

```c
You, 1 second ago | 1 author (You)
#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 "queue.h"
#include "process.h"      You, 1 hour ago • chore: ...

#define MAX_PROCESSES 100
#define MAX_NAME_LEN 256
#define MEMORY 1024

queue_t *priority_queue = NULL, *secondary_queue = NULL;
int avail_mem[MEMORY] = {0};

// Placeholder for exec_process function
int main() {
    // Attempt to open the file containing process information.
    FILE *file = fopen("processes_q2.txt", "r");
    if (file == NULL) {
        fprintf(stderr, "File not found\n");
        exit(EXIT_FAILURE);
    }

    // `input_process_list` is a temporary list that holds the processes read from the file.
    process_t input_process_list[MAX_PROCESSES];
    process_t data; // Temporary variable to hold the data read from the file.

    // `len` is the length of the data stored in `input_process_list` array from the file.
    int len = 0;

    // Read each line from the file and insert the process data into a queue.
    while (fscanf(file, "%[^,], %d, %d, %d\n", data.name, &data.priority, &data.memory, &data.runtime) == 4) {
        data.pid = 0;
        data.address = 0; // Indicating not yet allocated
        data.suspended = false;
        input_process_list[len++] = data;
    }

    // Close the file after reading the contents.
    fclose(file);

    // For efficient memory management, shrink the dispatch list to the number of processes actually read from the file.
    process_t dispatch_list[len];
    for (int i = 0; i < len; i++) {
        dispatch_list[i] = input_process_list[i];
    }

    printf("Processes:\n");

    // Sort the dispatch list by arrival time.
    int dispatch_list_len = sizeof(dispatch_list) / sizeof(dispatch_list[0]);

    // Push the processes into the appropriate queues based off of their priority.
    for (int i = 0; i < dispatch_list_len; i ++) {
        process_t *proc = &dispatch_list[i];
        if (proc->priority == 0) {
            push(&priority_queue, proc);
        } else {
            push(&secondary_queue, proc);
        }
    }

    int mem_index = 0;
    process_t *current_process;

    int status;
    pid_t pid = fork();

    // Priority queue
    while ((current_process = pop(&priority_queue)) != NULL) {
        for (int i = mem_index; i < mem_index + current_process->memory; i++) {
            avail_mem[i] = 1;
        }

        current_process->address = mem_index;
        mem_index += current_process->memory;

        // Forking to create a child process
        if (pid < 0) {
            fprintf(stderr, "Fork failed\n");
            return 1;
        } else if (pid == 0) {
            // Child process
```

```c
int main() {
        } else if (pid == 0) {
            // Child process
            // Replace child's image with `./proceess`
            // Execute the pre-compiled program './process'
            execl("./process", "./process", (char *)NULL);

            // execl only returns on error
            perror("execl");
            // printf("Child process\n");
            exit(1);
        } else {
            // Parent process
            current_process->pid = pid;
            printf("Name: %s, Priority: %d, PID: %d, Address: %d, Runtime: %d\n", current_process->name, current_process->priority, current_process->pid, current_process->memory, current_process->runtime);

            // Sleep for runtime of current_process in seconds before sending SIGTSTP
            sleep(current_process->runtime);

            // SIGSTP sent, process is suspended
            kill(pid, SIGTSTP);

            // SIGINT sent, process is terminated
            kill(pid, SIGINT);

            // Wait for the child to terminate
            waitpid(pid, &status, 0);
            if (WIFEXITED(status)) {
                printf("Child exited with status %d\n", WEXITSTATUS(status));
                pid = fork();
            }
        }

        for (int i = current_process->address; i < current_process->address + current_process->memory; i++) {
            // Deallocate the memory
            avail_mem[i] = 0;
        }

        // Reset the memory index
        mem_index = current_process->address;
    }

    if (pid > 0) {
        kill(pid, SIGINT); // Terminate the process
    }


    // Forking to create a child process
    pid = fork();

    // Print mem_index
    printf("mem_index: %d\n", mem_index);

    // Secondary queue
    while ((current_process = pop(&secondary_queue)) != NULL) {
        int status;

        if (pid < 0) {
            fprintf(stderr, "Fork failed\n");
            return 1;
        } else if (pid == 0) {
            // Child process
            // Replace child's image with `./proceess`
            // Execute the pre-compiled program './process'
            execl("./process", "./process", (char *)NULL);

            // execl only returns on error
            perror("execl");
            exit(1);
        } else {
            // Parent process
            current_process->pid = pid;
            printf("Name: %s, Priority: %d, PID: %d, Address: %d, Runtime: %d\n", current_process->name, current_process->priority, current_process->pid, current_process->memory, current_process->runtime);

            if (!current_process->suspended) {
                if ((mem_index - current_process->memory) > MAX_PROCESSES) {
                    printf("Insufficient memory for process %s\n", current_process->name);
                    push(&secondary_queue, current_process);
                    continue;
                }

                // Allocate the memory
                for (int i = mem_index; i < mem_index + current_process->memory; i++) {
                    avail_mem[i] = 1;
                }

                current_process->address = mem_index;
                mem_index += current_process->memory;
            } else {
```

```c
int main() {
    pid = fork();

    // Print mem_index
    printf("mem_index: %d\n", mem_index);

    // Secondary queue
    while ((current_process = pop(&secondary_queue)) != NULL) {
        int status;

        if (pid < 0) {
            fprintf(stderr, "Fork failed\n");
            return 1;
        } else if (pid == 0) {
            // Child process
            // Replace child's image with `./proceess`
            // Execute the pre-compiled program './process'
            execl("./process", "./process", (char *)NULL);

            // execl only returns on error      You, 1 hour ago • chore: ...
            perror("execl");
            exit(1);
        } else {
            // Parent process
            current_process->pid = pid;
            printf("Name: %s, Priority: %d, PID: %d, Address: %d, Runtime: %d\n", current_process->name, current_process->priority, current_process->pid, current_process->memory, current_process->runtime);

            if (!current_process->suspended) {
                if ((mem_index + current_process->memory) > MAX_PROCESSES) {
                    printf("Insufficient memory for process %s\n", current_process->name);
                    push(&secondary_queue, current_process);
                    continue;
                }

                // Allocate the memory
                for (int i = mem_index; i < mem_index + current_process->memory; i++) {
                    avail_mem[i] = 1;
                }

                current_process->address = mem_index;
                mem_index += current_process->memory;
            } else {
                kill(pid, SIGCONT);
                current_process->suspended = false;
            }

            if (current_process->runtime > 1) {
                // Sleep for 1 seconds before sending SIGTSTP
                sleep(1);
                kill(pid, SIGTSTP);

                current_process->runtime--;
                current_process->suspended = true;

                push(&secondary_queue, current_process);
            } else {
                // Process completes its execution
                sleep(current_process->runtime);
                kill(pid, SIGINT); // Terminate the process

                // Wait for the child to terminate
                waitpid(pid, &status, 0);
                if (WIFEXITED(status)) {
                    printf("Child exited with status %d\n", WEXITSTATUS(status));
                    pid = fork();
                }
            }
        }

        for (int i = current_process->address; i < current_process->address + current_process->memory; i++) {
            // Deallocate the memory
            avail_mem[i] = 0;
        }
    }

    if (pid > 0) {
        kill(pid, SIGINT); // Terminate the process
    }

    return 0;
}
```

```c
#include <stdbool.h> // Add this to use bool, true, and false

#ifndef PROCESS_H_
#define PROCESS_H_

#define MAX_NAME_SIZE 256

// Process struct, stores the process state
You, 1 hour ago | 1 author (You)
typedef struct process_t {
    char parent[MAX_NAME_SIZE]; // Name of the parent process.
    char name[MAX_NAME_SIZE];   // Name of the process.
    int priority;               // Priority of the process.
    int memory;                 // Memory in MB used by the proces
    int pid; // Process ID
    int address; // Memory address index
    int runtime; // Runtime in seconds
    bool suspended; // Indicates if the process is suspended
} process_t;

#endif /* PROCESS_H_ */
```

You, 15 hours ago | 1 author (You)

```c
#include "queue.h"      You, 15 hours ago • chore: os-tut-8

/**
 * Creates a new queue.
 *
 * @param queue Pointer to the head of the queue.
 *
 * Dynamically allocates memory for a new node and assigns it to the queue pointer.
 * Assumes that the passed 'queue' pointer points to a dummy head node to keep things simple.
 */
extern queue_t *create_queue() {
    // Dynamically allocate memory for a new node of the query.
    node_t *new_node = (node_t*) malloc(sizeof(node_t));
    if (!new_node) {
        fprintf(stderr, "Memory allocation failed to create a new queue\n");
        exit(EXIT_FAILURE);
    }

    new_node->process = NULL;
    new_node->next = NULL;
    new_node->prev = NULL;

    return new_node;
}

/**
 * Adds a new process to the end of the queue.
 *
 * @param queue Pointer to the head of the queue.
 * @param process Pointer to the process to be added to the queue.
 *
 * Dynamically allocates memory for a new node, assigns the process to it,
 * and inserts it at the end of the queue. Assumes that the passed 'queue'
 * pointer points to a dummy head node to keep things simple.
 */
extern void push(queue_t **queue, process_t *process) {
    // Dynamically allocate memory for a new node.
    node_t *new_node = (node_t*) malloc(sizeof(node_t));
    if (!new_node) {
        fprintf(stderr, "Memory allocation failed to add a new node to the queue\n");
        exit(EXIT_FAILURE);
    }

    new_node->process = process;
    new_node->next = NULL;
    new_node->prev = NULL;

    if (*queue == NULL || (*queue)->process == NULL) {
        // The queue is empty, so this new node is now the queue.
        *queue = new_node;
    } else {
        // Traverse the queue to find the last node.
        node_t *current = *queue;
        while (current->next != NULL) {
            current = current->next;
        }

        // Link the new node into the list.
        new_node->prev = current; // Set new_node's prev pointer to the last node.
        current->next = new_node; // Link the last node to the new node.
    }
}

/**
 * Removes and returns the current process from the specified node in the queue.
 *
 * This function takes a pointer to a node (within a queue) and removes that node from the queue.
 * It handles the connections of surrounding nodes to maintain the integrity of the queue.
 * Finally, it frees the memory allocated for the removed node and returns the process it contained.
 *
 * @param queue A double pointer to the node to be removed. This allows the function to modify
 *              the caller's pointer, particularly useful when removing the head of the queue.
 *              Give us flexibility to remove any node in the queue.
 * @return The process contained within the removed node. If the queue is empty or the pointer is
 *         NULL, it returns NULL.
 */
extern process_t *pop(queue_t **queue) {
    // Check if the queue or the target node is NULL. If so, there's nothing to remove.
    if (queue == NULL || *queue == NULL) {
        return NULL;
    }

    // Check if the node contains a process. If not, there's nothing to remove.
    // We don't want to remove the dummy head node, as if we do we would need to re-create the queue from scratch.
    if ((*queue)->process == NULL) {
        return NULL;
    }

    node_t *node_to_remove = *queue;
    process_t *return_process = node_to_remove->process;

    // If there's a node after the one we're removing, we need to update its 'prev' pointer
    // to skip the removed node, pointing to the previous node of the one being removed.
    if (node_to_remove->next != NULL) {
        node_to_remove->next->prev = node_to_remove->prev;
    }

```

nect   Live Share

```c
/**
 * Removes and returns the current process from the specified node in the queue.
 *
 * This function takes a pointer to a node (within a queue) and removes that node from the queue.
 * It handles the connections of surrounding nodes to maintain the integrity of the queue.
 * Finally, it frees the memory allocated for the removed node and returns the process it contained.
 *
 * @param queue A double pointer to the node to be removed. This allows the function to modify
 *              the caller's pointer, particularly useful when removing the head of the queue.
 *              Give us flexibility to remove any node in the queue.
 * @return The process contained within the removed node. If the queue is empty or the pointer is
 *         NULL, it returns NULL.
 */
extern process_t *pop(queue_t **queue) {
    // Check if the queue or the target node is NULL. If so, there's nothing to remove.
    if (queue == NULL || *queue == NULL) {
        return NULL;
    }

    // Check if the node contains a process. If not, there's nothing to remove.
    // We don't want to remove the dummy head node, as if we do we would need to re-create the queue from scratch.
    if ((*queue)->process == NULL) {
        return NULL;
    }

    node_t *node_to_remove = *queue;
    process_t *return_process = node_to_remove->process;

    // If there's a node after the one we're removing, we need to update its 'prev' pointer
    // to skip the removed node, pointing to the previous node of the one being removed.
    if (node_to_remove->next != NULL) {
        node_to_remove->next->prev = node_to_remove->prev;
    }

    // If there's a node before the one we're removing, we update its 'next' pointer to
    // skip the removed node, directly connecting to the next node of the one being removed.
    // If there's no previous node (meaning we're removing the head of the queue), we update
    // the head pointer to point to the next node.
    if (node_to_remove->prev != NULL) {
        node_to_remove->prev->next = node_to_remove->next;
    } else {
        // If removing the head, the next node becomes the new head of the queue.
        *queue = node_to_remove->next;
    }

    // Now that we've detached the node from the queue, we can safely free its memory.
    // What we're freeing here is the `node_t` struct that makes up the node in the queue and not the process itself.
    // As if we remove the process itself, we would lose the reference to it.
    free(node_to_remove);
    return return_process;
}
```

```c
You, 15 hours ago | 1 author (You)
#include <stdio.h>
#include <stdlib.h>
#include "process.h"

#ifndef QUEUE_H_
#define QUEUE_H_

You, 15 hours ago | 1 author (You)
typedef struct node_t {        You, 15 hours ago • chore: os-tut-8
    /** Pointer to the process data associated with this node. */
    process_t *process;
    /** Pointer to the next node in the queue, NULL if it's the last node. */
    struct node_t *prev;
    /** Pointer to the previous node in the queue, NULL if it's the first node. */
    struct node_t *next;
} node_t;

// Alias node_t as queue_t for clarity when used to represent a queue
typedef node_t queue_t;

/**
 * Creates a new queue.
 *
 * @param queue Pointer to the head of the queue.
 *
 * Dynamically allocates memory for a new node and assigns it to the queue pointer.
 * Assumes that the passed 'queue' pointer points to a dummy head node to keep things simple.
 */
extern queue_t *create_queue();

/**
 * Adds a new process to the end of the queue.
 *
 * @param queue Pointer to the head of the queue.
 * @param process Pointer to the process to be added to the queue.
 *
 * Dynamically allocates memory for a new node, assigns the process to it,
 * and inserts it at the end of the queue. Assumes that the passed 'queue'
 * pointer points to a dummy head node to keep things simple.
 */
extern void push(queue_t **queue, process_t *process);

/**
 * Removes and returns the current process from the specified node in the queue.
 *
 * This function takes a pointer to a node (within a queue) and removes that node from the queue.
 * It handles the connections of surrounding nodes to maintain the integrity of the queue.
 * Finally, it frees the memory allocated for the removed node and returns the process it contained.
 *
 * @param queue A double pointer to the node to be removed. This allows the function to modify
 *              the caller's pointer, particularly useful when removing the head of the queue.
 *              Give us flexibility to remove any node in the queue.
 * @return The process contained within the removed node. If the queue is empty or the pointer is
 *         NULL, it returns NULL.
 */
extern process_t *pop(queue_t **queue);

#endif /* QUEUE_H_ */
```

```
root@Okiki-PC ➜ Tutorial 8 git:(main) ✗ gcc -Wall -Wextra -std=c99 q2.c process.h queue.c queue.h -o q2.o && ./q2.o
q2.c: In function ‘main’:
q2.c:107:13: warning: implicit declaration of function ‘kill’ [-Wimplicit-function-declaration]
  107 |             kill(pid, SIGTSTP);
      |             ^~~~
Processes:
Name: systemd, Priority: 0, PID: 188591, Address: 256, Runtime: 5
 188591; START
 188591; tick 1
 188591; tick 2
 188591; tick 3
 188591; tick 4
 188591; tick 5
 188591; SIGINT
Child exited with status 0
Name: bash, Priority: 0, PID: 188658, Address: 64, Runtime: 8
 188658; START
 188658; tick 1
 188658; tick 2
 188658; tick 3
 188658; tick 4
 188658; tick 5
 188658; tick 6
 188658; tick 7
 188658; tick 8
 188658; SIGINT
Child exited with status 0
mem_index: 0
Name: vim, Priority: 3, PID: 188733, Address: 128, Runtime: 4
mem_index: 0
 188733; START
Name: emacs, Priority: 3, PID: 188733, Address: 256, Runtime: 4
 188733; tick 1
 188733; SIGTSTP
Name: chrome, Priority: 1, PID: 188733, Address: 512, Runtime: 2
Name: chrome, Priority: 1, PID: 188733, Address: 512, Runtime: 3
Insufficient memory for process chrome
Name: chrome, Priority: 1, PID: 188733, Address: 1024, Runtime: 5
Name: gedit, Priority: 2, PID: 188733, Address: 128, Runtime: 4
Insufficient memory for process gedit
Name: eclipse, Priority: 2, PID: 188733, Address: 1024, Runtime: 3
Insufficient memory for process eclipse
Name: clang, Priority: 1, PID: 188733, Address: 512, Runtime: 3
Insufficient memory for process clang
Name: vim, Priority: 3, PID: 188733, Address: 128, Runtime: 3
Ubuntu  main*  0 0  0  Connect  Live Share  Ln 162, Col 23 (254 selected)  Spaces: 4  UTF-8  LF
```

```
Name: vim, Priority: 3, PID: 188733, Address: 128, Runtime: 3
 188733; SIGCONT
Name: emacs, Priority: 3, PID: 188733, Address: 256, Runtime: 3
 188733; tick 2
Name: chrome, Priority: 1, PID: 188733, Address: 512, Runtime: 1
 188733; tick 3
 188733; SIGTSTP
```