

Relazione sul Progetto dell'Esame di
Sistemi Operativi
Anno Accademico 2016/17

Bacciarini Yuri - **xxxxxxx** - *yuri.bacciarini@stud.unifi.it*
Bindi Giovanni - **5530804** - *giovanni.bindi@stud.unifi.it*
Puliti Gabriele- **5300140** - *gabriele.puliti@stud.unifi.it*

June 24, 2017

Contents

| | | |
|----------|---|-----------|
| 1 | Primo Esercizio | 2 |
| 1.1 | Descrizione dell'implementazione | 2 |
| 1.2 | Evidenza del corretto funzionamento | 3 |
| 1.3 | Codice | 3 |
| 2 | Secondo Esercizio | 9 |
| 2.1 | Descrizione dell'implementazione | 9 |
| 2.2 | Evidenza del corretto funzionamento | 9 |
| 2.3 | Codice | 9 |
| 3 | Terzo Esercizio | 11 |
| 3.1 | Descrizione dell'implementazione | 11 |
| 3.2 | Evidenza del corretto funzionamento | 11 |
| 3.3 | Codice | 11 |

1 Primo Esercizio

Simulatore di chiamate a procedura

1.1 Descrizione dell'implementazione

L'obiettivo del primo esercizio è quello di implementare uno scheduler di processi. Quest'ultimo deve permettere all'utente di poter creare, eseguire ed eliminare i processi stessi secondo una politica di priorità od esecuzioni rimanenti.

Abbiamo organizzato il codice in tre files: due librerie *config.h* e *taskmanager.h* ed un programma, *scheduler.c*. All'interno di *config.h* vengono unicamente definite due stringhe utilizzate nella formattazione dell'output. All'interno di *taskmanager.h* abbiamo invece definito la **struct** *TaskElement*, ovvero l'elemento **Task**, descritto da 5 campi fondamentali che rappresentano un processo all'interno della nostra implementazione:

1. *ID* : Un numero intero univoco che viene automaticamente assegnato alla creazione del task.
2. *nameTask* : Nome del task, di massimo 8 caratteri, scelto dall'utente alla creazione.
3. *priority* : Numero intero che rappresenta la priorità del task.
4. *remainingExe* : Numero intero che rappresenta il numero di esecuzioni rimanenti (burst) del task.
5. **nextTask* : Puntatore al task successivo

Sempre all'interno di *taskmanager.h* vi sono le implementazioni delle operazioni che il nostro scheduler sarà in grado di effettuare, definite dalle seguenti funzioni:

- **setExeNumber(void)** : Permette l'inserimento del numero di esecuzioni rimanenti *n*, effettuando i controlli sulla legalità dell'input ($1 < n < 99$).
- **setPriority(void)** : Permette l'inserimento della priorità *p*, effettuando i controlli sulla legalità dell'input ($1 < p < 9$).
- **setTaskName(Task*)** : Permette l'inserimento del nome del task, effettuando i controlli sulla lunghezza massima della stringa inserita (al massimo 8 caratteri).
- **isEmptyTaskList(Task*)** : Esegue il controllo sulla lista di task, restituendo 0 nel caso sia vuota.
- **selectTask(Task*)** : Restituisce il task con il PID richiesto dall'utente, dopo aver eseguito la ricerca nella lista.
- **modifyPriority(Task*)** : Permette di modificare la priorità del task selezionato.
- **modifyExecNumb(Task*)** : Permette di modificare il numero di esecuzioni rimanenti del task selezionato.
- **newTaskElement(Task*,int)** : Permette la creazione di un nuovo task, allocandolo in memoria con l'utilizzo di una `malloc`.
- **printTask(Task*)** : Esegue la stampa degli elementi del task coerentemente con la richiesta nella specifica dell'esercizio.
- **printListTask(Task*)** : Esegue la stampa dell'intera lista dei task, richiamando la funzione `printTask`.
- **deleteTask(Task*, Task*)** : Permette l'eliminazione di un task dalla lista, semplicemente collegando il puntatore *nextTask* dell'elemento precedente al task successivo a quello che deve essere eliminato.
- **executeTask(Task*)** : Esegue il task in testa alla coda, eseguendo i controlli sul numero di esecuzioni rimanenti.

Le operazioni legate allo scheduling sono state poi affidate a *scheduler.c3*, il quale contiene le funzioni:


```

30 void printTask(Task*);
31 void printListTasks(Task*);
32 Task* deleteTask(Task*, Task*);
33 int executeTask(Task*);
34
35 int setExeNumber() {
36     int exeNum = 0;
37     do {
38         printf("\n\rInsert the number of remaning executions : ");
39         scanf("%i", &exeNum);
40         if ((exeNum < 0) || (exeNum > 99)) {
41             printf("\n\rError! It must be a number between 1 and 99. \n\r");
42         }
43     } while ((exeNum <= 0) || (exeNum > 99));
44     return exeNum;
45 }
46
47 int setPriority() {
48     int priority = 0;
49     do {
50         printf("\n\rInsert the priority (ascending order): ");
51         scanf("%i", &priority);
52         if ((priority < 0) || (priority > 9)) {
53             printf("\n\rError! It must be a number between 1 and 9\n\r");
54         }
55     } while ((priority < 0) || (priority > 9));
56     return priority;
57 }
58
59 void setTaskName(Task *actualTask) {
60     char name[8];
61     printf("\n\rName this task (max 8 chars) : ");
62     scanf("%8s", name);
63     strcpy(actualTask->nameTask, name);
64     return;
65 }
66
67 int isEmptyTaskList(Task *actualTask) {
68     if (!(actualTask->ID)) {
69         printf("\n\rList is empty! Please insert a task first...\n\r");
70         return 1;
71     }
72     return 0;
73 }
74
75 Task* selectTask(Task* actualTask) {
76     int id;
77     printf("Select the task...\n\rInsert the ID : ");
78     scanf("%d", &id);
79     while (actualTask->ID != id) {
80         actualTask = actualTask->nextTask;
81         if (actualTask == NULL) {
82             printf("\n\rError! No tasks with this ID!\n\r");
83             return actualTask;
84         }
85     }
86     return actualTask;
87 }
88
89 void modifyPriority(Task *thisTask) {
90     thisTask = selectTask(thisTask);
91     if (thisTask == NULL) {
92         return;
93     }
94     thisTask->priority = setPriority();
95     return;
96 }
97
98 void modifyExecNumb(Task *thisTask) {
99     thisTask = selectTask(thisTask);
100     if (thisTask == NULL) {
101         return;
102     }

```

```

103     thisTask->remainingExe = setExeNumber();
104     return;
105 }
106
107 Task* newTaskElement(Task *actualTask, int idT) {
108     actualTask->ID = idT;
109     setTaskName(actualTask);
110     actualTask->priority = setPriority();
111     actualTask->remainingExe = setExeNumber();
112     (*actualTask).nextTask = malloc(sizeof(Task));
113     return (*actualTask).nextTask;
114 }
115
116 void printTask(Task *thisTask) {
117     printf("| %d + %d + %s + %d | \n\r",
118         thisTask->ID, thisTask->priority, thisTask->nameTask,
119         thisTask->remainingExe);
120     printf(SEPARATOR);
121 }
122
123 void printListTasks(Task *first) {
124     printf(SEPARATOR);
125     printf("| ID + PRIORITY + TASK NAME + REMAINING EXEC | \n\r");
126     printf(SEPARATOR);
127     Task* tmp = first;
128     while (tmp->ID != 0) {
129         printTask(tmp);
130         tmp = (*tmp).nextTask;
131     }
132 }
133
134 Task* deleteTask(Task *first, Task *thisTask) {
135     if (thisTask != NULL) {
136         Task *tmpTask = first;
137         if (thisTask == first) {
138             tmpTask = thisTask->nextTask;
139             thisTask->ID = thisTask->priority = thisTask->remainingExe = 0;
140             strcpy(thisTask->nameTask, "\0");
141             thisTask->nextTask = NULL;
142             return tmpTask;
143         } else {
144             while (tmpTask->nextTask == NULL) {
145                 if (tmpTask->nextTask == thisTask) {
146                     tmpTask->nextTask = thisTask->nextTask;
147                     thisTask->ID = thisTask->priority = thisTask->remainingExe =
148                         0;
149                     strcpy(thisTask->nameTask, "\0");
150                     thisTask->nextTask = NULL;
151                     return first;
152                 }
153                 tmpTask = tmpTask->nextTask;
154             }
155         }
156     }
157     printf("There is no task to delete!\n\r");
158     return first;
159 }
160
161 int executeTask(Task *thisTask) {
162     if (thisTask != NULL) {
163         thisTask->remainingExe -= 1;
164         return thisTask->remainingExe;
165     } else if (thisTask->remainingExe == 0) {
166         printf("This task has no more executions to be done\n\r");
167         return 0;
168     }
169     printf("There is no task to execute!\n\r");
170     return 0;
171 }

```

Listing 2: Task Manager

```

2  * scheduler.h
3  *
4  *   Created on: 25 mag 2017
5  *   Authors: wabri, pagano
6  */
7  #include <string.h>
8  #include <stdio.h>
9  #include <stdlib.h>
10 #include "taskmanager.h"
11
12 int getChoice(void);
13 char switchPolicy(char pol);
14 Task* sortListByPriority(Task*);
15 Task* sortListByExecution(Task*);
16 Task* swapTask(Task*, Task*, Task*);
17
18 int main() {
19     int idTraker = 1;
20     int flag = 1;
21     char policy = 'p';
22     Task *firstTask = malloc(sizeof(Task));
23     Task *lastTask = NULL; // the last Task is always empty
24     Task *tmpTask;
25     printf(POINTSHEAD);
26     printf("                This is a process scheduler\n\r");
27     printf(POINTSHEAD);
28     while (flag == 1) {
29         switch (getChoice()) {
30             case 0:
31                 printf("Bye!\n\r");
32                 return 0;
33             case 1:
34                 if (firstTask->ID == 0) {
35                     lastTask = newTaskElement(firstTask, idTraker);
36                 } else {
37                     lastTask = newTaskElement(lastTask, idTraker);
38                     if (policy == 'p') {
39                         firstTask = sortListByPriority(firstTask);
40                     } else if (policy == 'e') {
41                         firstTask = sortListByExecution(firstTask);
42                     }
43                 }
44                 idTraker += 1;
45                 break;
46             case 2:
47                 printf("\n\rHow many execution do you want to do: ");
48                 scanf("%d", &flag);
49                 while (flag != 0) {
50                     if (executeTask(firstTask) == 0) {
51                         firstTask = deleteTask(firstTask, firstTask);
52                     }
53                     flag -= 1;
54                 }
55                 flag = 1;
56                 printf("\n\r");
57                 break;
58             case 3:
59                 tmpTask = selectTask(firstTask);
60                 if (executeTask(tmpTask) == 0) {
61                     firstTask = deleteTask(firstTask, tmpTask);
62                 }
63                 break;
64             case 4:
65                 firstTask = deleteTask(firstTask, selectTask(firstTask));
66                 break;
67             case 5:
68                 modifyPriority(firstTask);
69                 if (policy == 'p') {
70                     firstTask = sortListByPriority(firstTask);
71                 }
72                 break;
73             case 6:
74                 policy = switchPolicy(policy);

```

```

75     if (policy == 'p') {
76         firstTask = sortListByPriority(firstTask);
77     } else if (policy == 'e') {
78         firstTask = sortListByExecution(firstTask);
79     }
80     break;
81 case 7:
82     modifyExecNumb(firstTask);
83     if (policy == 'e') {
84         firstTask = sortListByExecution(firstTask);
85     }
86     break;
87 default:
88     flag = 0;
89     break;
90 }
91 if (!isEmptyTaskList(firstTask)) {
92     printf("\n\rScheduling Policy: ");
93     if (policy == 'p') {
94         printf("PRIORITY \n\r");
95     } else if (policy == 'e') {
96         printf("REMAINING EXECUTIONS \n\r");
97     }
98     printListTasks(firstTask);
99 }
100 }
101 return 0;
102 }
103
104 int getChoice() {
105     printf("\n\rPlease select an option:\n\r");
106     printf(" 0) Exit\n\r 1) Create a new task\n\r 2) Execute the task on the top of the\n\r\n\r");
107     printf(" 3) Execute a task\n\r 4) Delete a task\n\r 5) Modify the PRIORITY of a task\n\r\n\r");
108     printf(" 6) Switch policy (default : PRIORITY)\n\r 7) Modify the REMAINING EXECUTIONS of a\n\r\n\r");
109     printf(" task");
110     int res = 0;
111     printf("\n\r> ");
112     scanf("%i", &res);
113     return res;
114 }
115 }
116
117 char switchPolicy(char pol) {
118     printf("\n\rYou switched the policy of scheduling from ");
119     if (pol == 'p') {
120         printf("PRIORITY to REMAINING EXECUTIONS\n\r");
121         return 'e';
122     } else if (pol == 'e') {
123         printf("REMAINING EXECUTIONS to PRIORITY \n\r");
124         return 'p';
125     }
126     return 'p';
127 }
128
129 Task* sortListByPriority(Task *headTask) {
130     Task *tempTask = headTask;
131     Task *previousTempTask = tempTask;
132     int flag = 0;
133     while (!flag) {
134         flag = 1;
135         tempTask = headTask;
136         previousTempTask = tempTask;
137         while (tempTask->ID != 0) {
138             if (tempTask->priority < tempTask->nextTask->priority) {
139                 if (tempTask == headTask) {
140                     headTask = swapTask(headTask, tempTask, tempTask->nextTask);
141                 } else {
142                     previousTempTask = swapTask(previousTempTask, tempTask,
143                                                 tempTask->nextTask);
144                 }

```

```

145         flag = 0;
146     }
147     previousTempTask = tempTask;
148     tempTask = tempTask->nextTask;
149 }
150 }
151 return headTask;
152 }
153
154 Task* sortListByExecution(Task* headTask) {
155     Task *tempTask = headTask;
156     Task *previousTempTask = tempTask;
157     int flag = 0;
158     while (!flag) {
159         flag = 1;
160         tempTask = headTask;
161         previousTempTask = tempTask;
162         while (tempTask->ID != 0) {
163             if ((tempTask->remainingExe > tempTask->nextTask->remainingExe)
164                 && (tempTask->nextTask->remainingExe != 0)) {
165                 if (tempTask == headTask) {
166                     headTask = swapTask(headTask, headTask, headTask->nextTask);
167                 } else {
168                     previousTempTask = swapTask(previousTempTask, tempTask,
169                                                 tempTask->nextTask);
170                 }
171                 flag = 0;
172             }
173             previousTempTask = tempTask;
174             tempTask = tempTask->nextTask;
175         }
176     }
177     return headTask;
178 }
179
180 Task* swapTask(Task *previousTask, Task *taskSwap1, Task *taskSwap2) {
181     if (previousTask != taskSwap1) {
182         previousTask->nextTask = taskSwap2;
183         taskSwap1->nextTask = taskSwap2->nextTask;
184         taskSwap2->nextTask = taskSwap1;
185         return previousTask;
186     }
187     taskSwap1->nextTask = taskSwap2->nextTask;
188     taskSwap2->nextTask = taskSwap1;
189     return taskSwap2;
190 }

```

Listing 3: Scheduler

2 Secondo Esercizio

Esecutore di comandi

2.1 Descrizione dell'implementazione

L'obiettivo del secondo esercizio é quello di creare un esecutore di comandi UNIX che scriva, sequenzialmente o parallelamente, l'output dell'esecuzione su di un file.

Tutte le funzionalità del programma sono incluse all'interno della libreria *cmd.h4* e fanno uso a loro volta della libreria *unistd.h*. La funzione `initDataFolder()` si occupa di creare la cartella ed inserirvi il file di output. Essa viene generata all'interno della directory `"../commandexe/data/[pid]"` dove il *pid* é il process ID del chiamante in questione, ritornato dal `getpid()`. Il comando inserito dall'utente viene poi eseguito attraverso una `popen()`, la quale apre uno stream di scrittura/lettura su di una pipe, inserendovi l'output del comando. La funzione `execCommandAndLog(char,int)` genera due *char//*, rispettivamente il *path* ed il *filename*, quest'ultimo viene nominato attraverso il pid e l'indice di esecuzione, come richiesto dalla specifica di implementazione. Viene poi eseguito il comando ed il log dell'output: l'esecuzione viene affidata ancora una volta ad una `popen()` mentre la scrittura dell'output viene eseguita mediante le usuali funzioni dello `stdin` attraverso il descrittore di file generato precedentemente.

2.2 Evidenza del corretto funzionamento

Quí andranno gli screenshot

2.3 Codice

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <unistd.h>
5
6 #define MAX_CMD_LEN 100
7
8 // init folder ../commandexe/data/[pid] to store the logs
9 int initDataFolder() {
10     char cmd[30];
11     FILE *fp;
12     sprintf(cmd, "%s%i", "mkdir -p ../commandexe/data/", getpid());
13     fp = popen(cmd, "r");
14     if (fp == NULL) {
15         printf("[Error] - Error initialing process folder\n");
16         exit(1);
17     }
18     return 0;
19 }
20
21 // function that execute the c command and log the output in ../commandexe/data/[pid]/out
22 // [index]
23 int execCommandAndLog(char* c, int index) {
24     FILE *fp;
25     char path[1035];
26     char filename[7];
27
28     sprintf(filename, "../commandexe/data/%i/%s.%i", getpid(), "out", index);
29     FILE *f = fopen(filename, "w");
30     if (f == NULL) {
31         printf("[Error] - Error opening file!\n");
32         exit(1);
33     }
34
35     // command open to read
36     sprintf(c, "%s %s", c, "2>&1");
37     fp = popen(c, "r");
38
39     if (fp == NULL) {
40         printf(f, "[Error] - Error executing the command\n");
41     }
42 }
```

```

42 // read the output a line at a time - output it.
43 while (fgets(path, sizeof(path) - 1, fp) != NULL) {
44     fprintf(f, "%s", path);
45 }
46
47 // closing files
48 pclose(fp);
49 fclose(f);
50
51 return 0;
52 }
53
54 int cmd_out() {
55     int k = 1;
56     initDataFolder();
57     while (1) {
58         char cmd[MAX_CMD_LEN] = "";
59         printf("\nEnter the %d-cmd: ", k);
60         //read chars until \n
61         scanf("%[^\\n]", cmd);
62         getchar();
63         printf("Cmd entered : %s\\n", cmd);
64         if (strlen(cmd) == 0) {
65             printf("Bye!\\n");
66             exit(1);
67         }
68         execCommandAndLog(cmd, k);
69         k = k + 1;
70     }
71
72     return (0);
73 }

```

Listing 4: Executor

3 Terzo Esercizio

Message passing

3.1 Descrizione dell'implementazione

3.2 Evidenza del corretto funzionamento

Qui andranno gli screenshot

3.3 Codice