*/\**

*\* OS Assignment #2*

*\*/*

#include <unistd.h>

#include <stdlib.h>

#include <stdio.h>

#include <string.h>

#include <signal.h>

#include <ctype.h>

#include <errno.h>

#include <sys/wait.h>

#define MSG(x...) fprintf (stderr, x)

#define STRERROR strerror (errno)

#define PROCESS\_MAX (26 \* 10) *// 지정할 수 있는 프로세스 개수는 최대 260개 알파벳(26개:A~Z) \* 숫자(10개:0~9)*

#define ID\_MAX 2 *// 알파벳 대문자, 숫자로 구성 (중복 허용 X)*

*/\* 프로세스 도착시간 0 이상 30 이하 정수\*/*

#define ARRIVE\_TIME\_MIN 0

#define ARRIVE\_TIME\_MAX 30

*/\* 프로세스 서비스시간 1 이상 30 이하 정수\*/*

#define SERVICE\_TIME\_MIN 1

#define SERVICE\_TIME\_MAX 30

*/\* 프로세스 우선순위 1 이상 10 이하 정수 (작을 수록 우선순위 높음)\*/*

#define PRIORITY\_MIN 1

#define PRIORITY\_MAX 10

#define SLOT\_MAX ((ARRIVE\_TIME\_MAX + SERVICE\_TIME\_MAX) \* PROCESS\_MAX) *// ((30+30) \* 260)*

**enum**

{

SCHED\_SJF = 0, *// SCHED\_SJF (Shortest Job First)의 값 = 0*

SCHED\_SRT, *// SCHED\_SRT (Shortest Remaining Time First)의 값 = 1*

SCHED\_RR, *// SCHED\_RR (Round Robin)의 값 = 2*

SCHED\_PR, *// SCHED\_PR (Priority)의 값 = 3 \*단, preemptive 이다.*

SCHED\_MAX *// SCHED\_MAX 의 값 = 4*

};

**typedef** **struct** \_Process Process; *// 구조체명 \_Process을 Process로 재정의*

**struct** \_Process

{

**int** idx; *// id번호*

**int** queue\_idx; *// 기다리는 값*

**char** id[ID\_MAX + 1]; *// id값 char형 원소 3개 갖음*

**int** arrive\_time; *// 도착시간*

**int** service\_time; *// 서비스 수행 시간*

**int** priority; *// 우선순위*

**int** remain\_time; *// 완료까지 남은 시간*

**int** complete\_time; *// 종료 시각*

**int** turnaround\_time; *// 완료 시간*

**int** wait\_time; *// 대기 시간*

};

**static** Process processes[PROCESS\_MAX]; *// Process 구조체 배열로, 원소 갯수는 260개임. \*전역 스태틱 변수*

**static** **int** process\_total; *// 기본값 0. 프로세스가 추가될 경우 ++1되는 현재 프로세스들 갯수 \*전역 스태틱 변수*

**static** Process \*queue[PROCESS\_MAX]; *// Process 구조체를 가리킬 수 있는 Process 포인터 변수의 모음 queue 배열. 원소 갯수 260개 \* 전역 스태틱 변수*

**static** **int** queue\_len; *// queue 배열의 갯수*

*/\*프로세스 시뮬레이션 시 시각화 해줄 배열 : 프로세스 최대 갯수만큼 행, 각 프로세스가 가질 수 있는 최대 소요시간 만큼 열(슬롯) 할당\*/*

**static** **char** schedule[PROCESS\_MAX][SLOT\_MAX]; *// 2차원 char형 배열 schedule.*

**static** **char**\* strstrip (**char**\* str) {

**char** \*start;

size\_t len;

len = strlen (str);

**while** (len--)

{

**if** (!isspace (str[len]))

**break**;

str[len] = '\0';

}

**for** (start = str; \*start && isspace (\*start); start++)

;

memmove (str, start, strlen (start) + 1);

**return** str;

}

**static** **int** check\_valid\_id (**const** **char**\* str)

{

size\_t len;

**int** i;

len = strlen (str);

**if** (len != ID\_MAX)

**return** -1;

**for** (i = 0; i < len; i++)

**if** (!(isupper (str[i]) || isdigit (str[i])))

**return** -1;

**return** 0;

}

**static** Process\* lookup\_process (**const** **char**\* id)

{

**int** i;

**for** (i = 0; i < process\_total; i++)

**if** (!strcmp (id, processes[i].id))

**return** &processes[i];

**return** **NULL**;

}

**static** **void** append\_process (Process \*process) *// 프로세스 추가*

{

processes[process\_total] = \*process;

*// process 배열인 processes의 마지막 원소, 즉 맨 마지막 원소에 actual param으로 받은 process\*가 가리키는 객체의 값 복사*

processes[process\_total].idx = process\_total;

*// processes 배열 맨 마지막 process 객체의 멤버 변수 idx에 process\_total 값 대입*

process\_total++;

*// process\_total 값 1 추가 (스태틱이라 계속 반영됨.)*

}

**static** **int** read\_config (**const** **char** \*filename)

{

FILE \*fp;

**char** line[256];

**int** line\_nr;

fp = fopen (filename, "r");

**if** (!fp)

**return** -1;

process\_total = 0;

line\_nr = 0;

**while** (fgets (line, **sizeof** (line), fp))

{

Process process;

**char** \*p;

**char** \*s;

size\_t len;

line\_nr++;

memset (&process, 0x00, **sizeof** (process));

len = strlen (line);

**if** (line[len - 1] == '\n')

line[len - 1] = '\0';

**if** (0)

MSG ("config[%3d] %s\n", line\_nr, line);

strstrip (line);

*/\* comment or empty line \*/*

**if** (line[0] == '#' || line[0] == '\0')

**continue**;

*/\* id \*/*

s = line;

p = strchr (s, ' ');

**if** (!p)

**goto** invalid\_line;

\*p = '\0';

strstrip (s);

**if** (check\_valid\_id (s))

{

MSG ("invalid process id '%s' in line %d, ignored\n", s, line\_nr);

**continue**;

}

**if** (lookup\_process (s))

{

MSG ("duplicate process id '%s' in line %d, ignored\n", s, line\_nr);

**continue**;

}

strcpy (process.id, s);

*/\* arrive time \*/*

s = p + 1;

p = strchr (s, ' ');

**if** (!p)

**goto** invalid\_line;

\*p = '\0';

strstrip (s);

process.arrive\_time = strtol (s, **NULL**, 10);

**if** (process.arrive\_time < ARRIVE\_TIME\_MIN

|| ARRIVE\_TIME\_MAX < process.arrive\_time

|| (process\_total > 0 &&

processes[process\_total - 1].arrive\_time > process.arrive\_time))

{

MSG ("invalid arrive-time '%s' in line %d, ignored\n", s, line\_nr);

**continue**;

}

*/\* service time \*/*

s = p + 1;

p = strchr (s, ' ');

**if** (!p)

**goto** invalid\_line;

\*p = '\0';

strstrip (s);

process.service\_time = strtol (s, **NULL**, 10);

**if** (process.service\_time < SERVICE\_TIME\_MIN

|| SERVICE\_TIME\_MAX < process.service\_time)

{

MSG ("invalid service-time '%s' in line %d, ignored\n", s, line\_nr);

**continue**;

}

*/\* priority \*/*

s = p + 1;

strstrip (s);

process.priority = strtol (s, **NULL**, 10); *// 현재 process의 priority를 s부터 NULL문자 까지 읽고, 10진수로 바꾼다.*

**if** (process.priority < PRIORITY\_MIN *// 유효한 priority 값이 아닌경우. 1~10사이의 정수가 아닌경우*

|| PRIORITY\_MAX < process.priority)

{

MSG ("invalid priority '%s' in line %d, ignored\n", s, line\_nr);

**continue**; *// 올라가서 다음 줄 읽기*

}

append\_process (&process); *// 프로세스 추가.*

**continue**;

invalid\_line:

MSG ("invalid format in line %d, ignored\n", line\_nr);

}

fclose (fp);

**return** 0;

}

**static** **void** simulate (**int** sched) *// process scheduling algorithm 시뮬레이션 하는 함수*

{

Process\* process; *// Process\* 형 변수 process (현재 실행되는 프로세스 나타냄)*

**int** p; *// int 형 변수 p (프로세스 갯수)*

**int** p\_done; *// int 형 변수 p (완료된 프로세스 갯수)*

**int** cpu\_time; *// int 형 변수 cpu\_time (cpu\_time 나타냄)*

**int** sum\_turnaround\_time; *// int 형 변수 sum\_turnaround\_time (소요시간(대기시간+서비스시간) 합계)*

**int** sum\_waiting\_time; *// int 형 변수 sum\_waiting\_time (대기시간 합계)*

**float** avg\_turnaround\_time; *// float 형 변수 avg\_turnaround\_time (평균 소요시간(대기시간+서비스시간))*

**float** avg\_waiting\_time; *// float 형 변수 avg\_waiting\_time (평균 대기시간)*

**for** (p = 0; p < PROCESS\_MAX; p++) *// 프로세스 최대갯수(260) 까지 반복*

{

**int** slot;

**for** (slot = 0; slot < SLOT\_MAX; slot++)

schedule[p][slot] = 0; *// 반복문으로 char 배열 schedule의 모든 원소 0으로 초기화*

queue[p] = **NULL**; *// 반복문으로 Process\* 배열 queue의 모든 원소가 아무것도 가리키지 않도록 초기화*

}

p = 0;

p\_done = 0;

queue\_len = 0;

process = **NULL**; *// Process\* 형 변수도 아무것도 가리키지 않도록 초기화*

**for** (cpu\_time = 0; p\_done < process\_total; cpu\_time++)

{

*/\* Insert arrived process into the queue. \*/*

**for** (; p < process\_total; p++)

{

Process \*pp;

pp = &processes[p];

**if** (pp->arrive\_time != cpu\_time) **break**;

pp->remain\_time = pp->service\_time;

pp->queue\_idx = queue\_len;

queue[queue\_len] = pp;

queue\_len++;

}

*/\* Pick a process according to scheduling algorithm. \*/*

**switch** (sched)

{

**case** SCHED\_SJF: *// shortest job first scheduling algorithm*

**if** (!process)

{

**int** i;

**int** shortest;

shortest = SERVICE\_TIME\_MAX + 1; *// shortest 값을 최대+1로 설정*

**for** (i = 0; i < queue\_len; i++)

**if** (queue[i]->service\_time < shortest) *// queue[i]의 service\_time이 shortest보다 작은 경우(더 짧은 service)*

{

process = queue[i]; *// 현재 process가 queue[i]가 가리키는 것과 같은 Process 가리키도록 함*

shortest = process->service\_time; *// shortest에 현재 process의 멤버 service\_time 값 복사*

}

}

**break**;

**case** SCHED\_SRT: *// shortest remaining time first scheduling algorithm*

{

**int** i;

**int** shortest;

shortest = SERVICE\_TIME\_MAX + 1; *// shortest 값을 최대+1로 설정*

**for** (i = 0; i < queue\_len; i++)

**if** (queue[i]->remain\_time < shortest) *// queue[i]의 service\_time이 shortest보다 작은 경우(더 짧은 service)*

{

process = queue[i]; *// 현재 process가 queue[i]가 가리키는 것과 같은 Process(구조체) 가리키도록 함*

shortest = process->remain\_time; *// shortest에 현재 process의 멤버 service\_time 값 복사*

}

}

**break**;

**case** SCHED\_RR: *// round-robin scheduling algorithm*

{

**int** i;

process = queue[0]; *// process는 queue의 첫번째 원소가 가리키는 것과 같은 Process(구조체) 가리키도록 함*

**for** (i = 0; i < (queue\_len - 1); i++) *// queue가 2개 이상일 때*

{

queue[i] = queue[i + 1]; *// queue[i] 원소가 queue[i+1]번째 원소가 가리키는 것과 같은 Process(구조체) 가리키도록 함*

queue[i]->queue\_idx = i; *// queue[i] 원소의 멤버 queue\_idx를 i 값으로 바꿈.*

}

queue[i] = process; *// queue[i] 원소에 현재 process가 가리키는 것과 같은 Process(구조체) 가리키도록 함*

queue[i]->queue\_idx = i; *// queue[i] 원소의 멤버 queue\_idx를 i 값으로 바꿈.*

}

**break**;

**case** SCHED\_PR: *// priority scheduling algorithm (preemptive)*

*/\**

*SCHED\_PR 구동 원리 :*

*1. priority 정렬을 위한 임시변수 설정*

*: 블럭 내의 int형 변수 tmp\_PR을 설정하고 PRIORITY\_MAX+1, 즉 priority 최대값(최하위) + 1 값 주었음*

*2. priority가 같을 경우의 임의 규칙 설정 (service\_time이 짧은 것을 먼저 실행하는 SJF scheduling algorithm 적용)*

*: 블럭 내의 int형 변수 tmp\_service을 설정하고 SERVICE\_TIME\_MAX+1, 즉 service\_time 최대값(30) + 1 값 주었음*

*3. queue 를 순회*

*: queue[i]가 가리키는 Process 구조체의 멤버 priority와 tmp\_PR을 비교*

*\* priority가 tmp\_PR보다 작은 경우*

*- 현재 process가 queue[i]가 가리키는 것과 같은 Process 구조체 가리키도록 함*

*- tmp\_PR에 process가 가리키는 Process 구조체의 멤버 priority 값 복사*

*- tmp\_service에 process가 가리키는 Process 구조체의 멤버 service\_time 값 복사*

*\* priority가 tmp\_PR과 같은 경우*

*- queue[i]가 가리키는 Process 구조체의 멤버 service\_time이 tmp\_service보다 작은 경우,*

*현재 process가 queue[i]가 가리키는 것과 같은 Process 구조체 가리키도록 함*

*- tmp\_service에 process가 가리키는 Process 구조체의 멤버 service\_time 값 복사*

*\*/*

{

**int** i; *// index 접근을 위한 int형 변수 i*

**int** tmp\_PR = PRIORITY\_MAX+1; *// priority 정렬을 위한 임시변수 tmp\_PR 선언 및 초기화*

**int** tmp\_service = SERVICE\_TIME\_MAX+1; *// priority가 같을 경우 SJF algorithm 정렬을 위한 임시변수 tmp\_service 선언 및 초기화*

**for** (i=0;i<queue\_len;++i)

{

**if**(queue[i]->priority < tmp\_PR) *// queue[i]의 priority가 tmp\_PR보다 작을 경우*

{

process = queue[i]; *// 현재 process가 queue[i]가 가리키는 것과 같은 Process 구조체 가리키도록 함*

tmp\_PR = process->priority; *// tmp\_PR에 process가 가리키는 Process 구조체의 멤버 priority 값 복사*

}

**else** **if** (queue[i]->priority == tmp\_PR) *// queue[i]의 priority가 tmp\_PR과 같은 경우*

{

**if** (queue[i]->service\_time < tmp\_service) *// queue[i]의 service\_time이 tmp\_service보다 작은 경우*

{

process = queue[i]; *// 현재 process가 queue[i]가 가리키는 것과 같은 Process 구조체 가리키도록 함*

tmp\_service = process->service\_time; *// tmp\_service에 process가 가리키는 Process 구조체의 멤버 service\_time 값 복사*

}

}

}

}

**break**;

**default**:

MSG ("invalid scheduing algorithm '%d', ignored\n", sched); *// sched 값에 SJF, SRT, RR, PR이외의 값이 들어온 경우*

**return**;

}

**if** (0)

MSG ("[%02d] %s[%d:%d] %d/%d\n",

cpu\_time,

process->id,

process->idx,

process->queue\_idx,

process->remain\_time,

process->service\_time);

*/\* no process to schedule. \*/*

**if** (!process)

**continue**;

schedule[process->idx][cpu\_time] = 1;

process->remain\_time--; *// 현재 프로세스 남은 시간 1씩 줄임.*

**if** (process->remain\_time <= 0)

{

**int** i;

**for** (i = process->queue\_idx; i < (queue\_len - 1); i++)

{

queue[i] = queue[i + 1];

queue[i]->queue\_idx = i;

}

queue\_len--;

process->complete\_time = cpu\_time + 1;

*// 프로세스의 완료 시각은 cpu\_time + 1 (완료를 정하는 시간까지 +1)*

process->turnaround\_time = process->complete\_time - process->arrive\_time;

*// 프로세스가 완료되는데 걸린 시간은 프로세스의 완료시각(complete\_time) - 프로세스의 도착시각(arrive\_time)*

process->wait\_time = process->turnaround\_time - process->service\_time;

*// 프로세스가 기다린 시간은 프로세스가 완료되는데 걸린시간(turnaround\_time) - 프로세스가 실제 실행된 시간(serviced\_time)*

process = **NULL**;

p\_done++; *// 완료된 프로세스 갯수 +1 증가*

}

}

printf ("\n[%s]\n",

sched == SCHED\_SJF ? "SJF" :

sched == SCHED\_SRT ? "SRT" :

sched == SCHED\_RR ? "RR" :

sched == SCHED\_PR ? "PR" : "UNKNOWN");

sum\_turnaround\_time = 0; *// sum\_turnaround\_time 값 0 으로 초기화*

sum\_waiting\_time = 0; *// sum\_waiting\_time 값 0 으로 초기화*

**for** (p = 0; p < process\_total; p++) *// Gantt Chart 출력*

{

**int** slot;

printf ("%s ", processes[p].id); *// processes 원소에 해당하는 id 값 출력*

**for** (slot = 0; slot <= cpu\_time; slot++)

putchar (schedule[p][slot] ? '\*' : ' '); *// process별로 schedule에 잡힌 시간 \*로 표시*

printf ("\n");

sum\_turnaround\_time += processes[p].turnaround\_time; *// sum\_turnaround\_timeprocesses(소요시간 합)에 원소의 turnaround\_time(소요시간) 계속 더해감*

sum\_waiting\_time += processes[p].wait\_time; *// sum\_waiting\_time(대기시간 합)에 processes 원소의 waiting\_time(대기시간) 계속 더해감*

}

avg\_turnaround\_time = (**float**) sum\_turnaround\_time / (**float**) process\_total; *// avg\_turnaround\_time에 소요시간 합 / 추가된 프로세스 갯수 연산한 값 복사*

avg\_waiting\_time = (**float**) sum\_waiting\_time / (**float**) process\_total; *// avg\_waiting\_time에 대기시간 합 / 추가된 프로세스 갯수 연산한 값 복사*

printf ("CPU TIME: %d\n", cpu\_time);

printf ("AVERAGE TURNAROUND TIME: %.2f\n", avg\_turnaround\_time);

printf ("AVERAGE WAITING TIME: %.2f\n", avg\_waiting\_time);

}

**int** main (**int** argc, **char** \*\*argv)

{

**int** sched;

**if** (argc <= 1) *// terminal에 실행시키는 문자열만 입력한 경우*

{

MSG ("usage: %s input-file\n", argv[0]);

**return** -1;

}

**if** (read\_config (argv[1])) *// 두번째 문자열(읽을 file) 읽었을 때 읽기 실패할 경우*

{

MSG ("failed to load config file '%s': %s\n", argv[1], STRERROR);

**return** -1;

}

**for** (sched = 0; sched < SCHED\_MAX; sched++) *// sched이 4보다 작을 때까지 반복*

simulate (sched); *// simulate 함수 실행*

**return** 0;

}