Simulation of Multilevel scheduling algorithm

과목명 : 운영체제(공통반)

교수명 : 양승민

학과 : 산업정보시스템공학과

학번 : 20131847

이름 : 박호정

제출일자 : 2019.10.18

목차

1. 소개
2. 관련 연구

2.1 소스 분석

2.2 사용한 시스템 콜, API 함수 설명

1. 문제해결방법 구현 방법
2. 결론

1.소개

텍스트 파일 형식의 입력 파일을 프로그램의 첫 번째 인자로 받아들여 프로세스 정보를 입력 받고 이를 이용하여 멀티 레벨 큐 스케줄링을 시뮬레이션 하는 프로그램이다. 입력 파일에는 한 줄마다 프로세스의 스케줄링 정보가 존재하며 “id process-type arrive-time service-time priority” 의 5개 정보가 존재한다. 이 정보를 기반으로 멀티 레벨 큐 스케줄링을 활용하여 시뮬레이션한다. 시뮬레이션 결과로 나온 프로세스 별 스케줄링 결과를 간트차트 형태로 출력하며 평균 turnaround time, 평균 wait time을 출력한다.

본 과제는 입력 파일 처리, 멀티 레벨 큐 스케줄링을 이용한 시뮬레이션을 모두 구현한다.

2.관련 연구

2.1 소스 분석

2.1.1 전체적인 프로그램 구조

본 프로그램은 main 함수에서는 크게 함수 2개를 호출하고 종료된다. 즉, read\_config 함수와 simulate 함수를 호출하도록 구현되었다. 이후 결과를 출력하는 함수를 호출하면 main 함수가 종료된다.

read\_config 수행 이전에 인자의 예외처리를 진행하였다. 인자가 없거나 인자가 잘못된 경우에는 예외 메세지를 출력하고 종료하도록 하였다.

인자의 예외처리를 통과 한 후 read\_config함수를 호출한다. read\_config 함수에서 과제1에서와 유사하게 string parsing을 이용하며 텍스트 한 줄에 대해서 한번의 반복문을 수행한다. 한 줄에는 5개의 정보가 있으며 각각의 정보 마다 입력 조건에 맞는지 체크한다. 5개의 정보가 예외없이 통과하였다면 프로세스 스케줄링 정보로 활용한다. 이 정보는 process\_array라는 배열에 저장된다. process\_array는 Process 타입을 원소로 갖는 배열로 모든 프로세스 정보를 가지고 있는 데이터이다. 프로세스 정보를 등록하기 위해 add\_process라는 함수를 이용한다. 위 과정을 반복하여 텍스트 파일을 전부 읽고 나면 모든 프로세스 스케줄링 정보가 process\_array에 등록된다.

이 정보를 기준으로 멀티 레벨 큐 알고리즘을 적용한 시뮬레이션을 진행한다. simulate 함수는 실행할 프로세스가 있다면 계속 진행된다. 즉, remain\_process\_num이 아직 완료되지 않은 프로세스의 개수를 의미하는데 0보다 큰 경우에 반복이 진행된다.

한 번의 반복 동안에는 크게 3가지 파트가 있다. 첫 번째 파트는 High type을 위한 파트이고 두 번째 파트는 Mid type을 위한 파트이다. 마지막 파트는 파트1이나 2에서 선택된 프로세스에 cpu time을 할당하는 작업이다.

첫 번째 파트에서는 mod 연산을 통하여 구현하였다. cpu time을 10 단위로 끊은 후 6:4 의 비율을 사용하였다. 이를 통해 60%의 cpu time을 H에 할당하도록 하였다. 이 시간 동안은 H, M, L 순서로 큐를 확인하면서 작업을 할당한다. 즉, H 큐에 작업이 존재하면 H 큐에서 가장 높은 우선순위 작업에 cpu time을 할당하는 것이다.

두 번째 파트에서는 첫번째 파트와 유사하다. 다만 40%로 바뀌고 M, H, L 순서로 큐를 확인한다.

세 번째 파트에서는 첫 번째 파트나 두 번째 파트에서 할당된 작업에 cpu time 1을 할당한다. cpu time 1을 할당함으로써 해당 작업이 끝나게 되는 경우에는 처리가 남은 프로세스의 수를 1 감소시킨다. read\_config 결과로서 처리해야할 프로세스의 수가 정해지기 때문에 모든 작업을 완료 해야만 반복문이 끝난다.

반복문에 들어가서 첫 번째 파트 시작 전에 항상 arrive time <= cpu time 인 작업을 멀티레벨 큐에 할당해주는 작업을 진행한다. 이 때 큐에 할당되는 기준은 각 큐의 기준에 따른다. H 큐인 경우엔 process의 priority를 기준으로 순서가 정해진다.

다음은 코드 상에 존재하는 목록이다. #define, global variable, function definition 부분이 존재한다. 순서대로 설명하도록 한다.

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

출력을 위한 지정을 하였다.

#define PROCESS\_MAX\_NUMBER (26 \* 10)

대문자 하나와 숫자 하나를 조합하여 만들 수 있는 가지 수 이다. 만들 수 있는 프로세스 개수와 동일하다.

#define ID\_MAX 2

id의 길이 제한이다.

#define ARRIVE\_TIME\_MIN 0

#define ARRIVE\_TIME\_MAX 30

도착 시간에 대한 제한 사항이다.

#define SERVICE\_TIME\_MIN 1

#define SERVICE\_TIME\_MAX 30

서비스 시간에 대한 제한 사항이다.

#define PRIORITY\_MIN 1

#define PRIORITY\_MAX 10

우선 순위에 대한 제한 사항이다.

#define SLOT\_MAX ARRIVE\_TIME\_MAX + (SERVICE\_TIME\_MAX \* PROCESS\_MAX\_NUMBER)

최대로 cpu가 동작할 수 있는 시간이다.

#define TRUE 1

#define FALSE 0

조건문에서 사용하기 위해 정의하였다.

**typedef struct \_Process Process;**

프로세스 타입을 정의하기 위해 타입을 정의하였다. input data와 simulation 동안 사용할 data, 출력을 위한 data를 저장하는 필드들이 존재한다.

**typedef** **struct** \_Process\_queue Process\_queue;

멀티 레벨 큐에서 큐 한 개를 관리하기 위한 타입이다. linked list 형태로 관리하기 위해 Process\* 타입의 head, tail, cur 가 존재한다.

전역 변수는 다음과 같이 선언하였다.

Process process\_array[PROCESS\_MAX\_NUMBER];

read\_config 이후 스케줄링 대상이 되는 모든 프로세스를 담는 배열이다.

Process\_queue high\_process\_queue; *// H type queue*

Process\_queue mid\_process\_queue; *// M type queue*

Process\_queue low\_process\_queue; *// L type queue*

멀티 레벨 큐를 관리하기 위한 변수 세 개이다.

**static** **int** num\_of\_all\_process;

프로그램에서 처리해야할 프로세스의 총 개수를 저장할 변수이다.

**static** **int** last\_assigned\_process\_order;

process\_array 배열에서 최근에 멀티 레벨 큐에 할당된 프로세스의 인덱스를 저장할 변수이다. 한 번만 멀티 레벨 큐에 할당하기 위해 필요한 변수이다.

**static** **void** prn\_split\_line()

**static** **void** prn\_split\_str(**char** \* s)

디버깅 시에 구분 선을 출력을 하기 위한 함수이다.

**static** **void** prn\_multilevel\_queue()

멀티레벨 큐를 모두 출력해주는 함수이다. 큐 별로 우선순위가 높은 순서대로 출력한다.

출력 예시)

**H queue ) [P4]-> [P1]**

**M queue ) [P3]**

**L queue ) [P2]**

**static** **void** prn\_process\_array(**void**)

모든 process가 저장되어 있는 배열을 전부 출력해주는 함수이다. read\_config함수 호출 이후 정상적으로 프로세스 정보가 저장되었는지 확인하기 위해 구현하였다.

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

공백을 제거하기 위한 함수이다.

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

id가 유효한지 체크하는 함수이다. 길이는 2이고 대문자와 숫자가 순서대로 오는지 확인한다. 유효한 경우 1, 아닌경우 -1 리턴

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

이미 존재하는 id인지 확인한다.

존재하는 경우 해당 프로세스 주소를 리턴, 아닌 경우 NULL 리턴

**static** **void** add\_process (Process \*process)

process\_array에 프로세스를 추가한다.

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

입력파일을 한 줄씩 파싱하며 5개의 정보가 정상적으로 읽어진 경우에 add\_process를 수행한다.

정상적으로 읽은 경우 1, 아닌 경우 -1 리턴

**static** **void** assign\_process\_to\_multi\_level\_queue(**int** cpu\_time)

cpu time을 기준으로 멀티 레벨 큐에 할당되어야 하는 시점이 된 것들을 확인한다. 확인된 프로세스들은 각각 add\_multilevel\_process\_queue(type, process)를 호출해준다.

**static** **int** isEmptyQueue(Process\_queue \*pq)

큐가 비었는지 확인해준다.

빈 경우 1, 아닌 경우 -1을 리턴

**static** **int** add\_multilevl\_process\_queue(**char** process\_type, Process \* add\_process)

타입과 프로세스가 주어지면 해당 큐에 추가해준다. 링크드 리스트 형태로 큐를 구현했으며 add 시에 타입별로 ascending criteria가 다르다. 각각 H는 priority , M은 service time, L은 arrive time이 기준이 된다.

정상적인 경우 1 리턴

**static** **int** simulate ()

완료해야할 프로세스의 개수만큼 수행할 때까지 cpu scheduling 작업을 반복한다. 소스 분석에서 설명한 바와 같이 assign\_process\_to\_multi\_level\_queue를 수행한 후 세 가지 파트로 나누어 진다. 첫 번째, 60%의 cpu time동안은 H, M, L 순으로 큐에 작업을 확인하며 cpu time을 할당한다. 두 번째, 40%의 cpu time동안은 M, H, L 순으로 큐에 작업을 확인하며 cpu time을 할당한다. 세 번째, 해당 프로세스를 cpu time 1만큼 감소시키며 이 프로세스가 종료 되었을 경우 큐에서 제거하고 처리해야할 프로세스의 수를 1 감소시킨다. 위 과정을 처리해야할 프로세스가 0이 될 때까지 반복한다.

**static** **int** inititalize\_multilevel\_queue()

멀티 레벨 큐 세개를 초기화 한다. head node와 tail node를 생성하고 head->next가 tail node를 가리키도록 초기화 시킨다.

정상적으로 완료한 경우 1, 아닌 경우 -1 리턴

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

입력 값이 정상적인지 확인하고 멀티레벨 큐 초기화 함수, read\_config함수, simulate함수, result 출력함수를 호출한다.

**static** **void** prn\_result(**int** cpu\_time)

결과를 출력하는 함수이다. 과제의 결과를 출력하는 부분도 포함되어 있다. 디버깅 시에 활용한 다양한 출력문들이 존재한다.

2.2 사용한 시스템 콜, API 함수 설명

2.2.1 fclose()

int fclose (FILE \*stream)

성공시 0 return, 실패시 -1 return

개방된 스트림을 닫는다. 버퍼에 남은 출력용 데이터는 파일로 기록, 입력용 데이터는 지워짐(버퍼 비워짐)

에러발생시 NULL 반환

2.2.2 malloc()

void\* malloc (size\_t size);

동적으로 size 만큼 메모리를 heap 영역에 할당한 뒤 반환.

2.2.3 memset()

void memset (void \*s, int c, size\_t n);

\* s : 설정할 메모리 시작 주소

\* c : 8bit 짜리 값

\* n : 개수

\* s가 가리키는 메모리를 값 c로 n개 채움

2.2.4 isupper()

int isupper(int x);

인수로 받은 x가 대문자인지 판별하는 함수이다. 대문자인 경우 1, 아닌경우 0보다 큰 값을 리턴

2.2.5 isdigit()

int isdigit(int c)

인수로 받은 문자가 숫자 문자인 (‘0’~’9’)지를 판별한다. 숫자인 경우 1, 아닌 경우 0보다 큰 값을 리턴

2.2.6 strchr()

char \* strchr(const char \* string, int c);

read\_config(char \*)함수에 있는 함수로 문자열에 공백이 있는지 검사하는데 쓰이는 함수.

문자열에서 임의의 문자가 처음으로 발견된 위치를 구한다.

문자가 발견된 위치를 포인터로 반환한다.

2.2.7 long strtol(const char \*restrict str, char \*\*endptr, int base)

숫자 문자열을 long형 숫자로 변환.

Atoi()나 atoll()과는 달리 변환 하려는 진수를 선택 가능하다.

숫자가 아닌 문자를 만나면 그 포인터 위치를 구한다.

char \*str : 정수로 변환할 문자열

char \*\*endptr : 숫자로 변경하지 못하는 문자열의 시작

정상적인 경우 long, 아닌 경우 0이 리턴

3.문제해결방법 구현 방법

구현 방법을 생각하는 과정에서 자료구조를 정하는 것이 핵심이었다. process의 정보는 파일에 주어져 있으며 이 정보를 가지고 관리해야 하기 때문이다. 보다 관리하기 쉽고 성능도 좋은 형태로 자료구조를 고려하였다. queue에 append 작업시 배열로 할 경우 sort 비용이 크기 때문에 ascending order로 append해주는 linked list를 사용하기로 하였다. 입력 파일의 정보는 최초에 한 번 읽고 모두 저장되기 때문에 전역변수의 배열에 저장하는 것이 좋다고 판단하였다.

정리하면, data가 관리되는 형태는 다음과 같다. 입력 파일을 바탕으로 프로세스의 정보를 전역변수 process\_array에 저장한다. 이 배열은 프로그램이 끝날 때까지 존재한다. 또한, 배열의 원소가 Process 타입이므로 시뮬레이션 과정에서 큐 연결 작업 시 배열에 있는 데이터를 통해 진행한다. 논리적으로는 process\_array와 queue 3개가 독립적으로 작동하는 것으로 생각하지만 물리적으로는 queue에 있는 process들은 process\_array에 존재한다. 즉, 멀티 레벨 큐를 각각의 큐로 관리하고 이를 c의 linked list 형태로 관리하는 것이다.

위와 같은 자료구조를 바탕으로 cpu time이 1씩 증가하며 short term 스케줄링을 진행하게 된다. 이 때 cpu time 1은 1 msec를 의미한다. 이 때 단위를 10으로 잡아서 앞의 6은 H 큐를 스케줄링하고 뒤의 4는 M 큐를 스케줄링 하도록 설계하였다. 이렇게 선택한 이유는 장기적으로 보았을 때 cpu time이 골고루 쓰일 것이라고 판단했기 때문이다. 왜냐하면 H, M 타입의 프로세스가 높은 빈도로 존재 하였을 때 10단위에서 6과 4를 연속적으로 분배하는 것과 특정 기준에 맞추어 60%,40%를 연속적이지 않게 분배하는 것 모두 상황에 따라 성능이 다를 것이기 때문이다.

예시)

P1 M 2 5 1

P2 H 3 5 1

본 과제 구현 방식은 다음과 같다. 시작과 동시에 H큐에 60%가 할당된다. 그 후 40%가 할당되고 종료된다. 따라서 다음과 같은 차트가 된다.

P1 \* \*\*\*\*\*

P2. \*\*\*\*\*\*

본 과제 구현 방식과 다른 알고리즘 k 방식은 다음과 같다. H, M에 해당하는 작업이 없을 때 L을 수행하고 있다가 H나 M 이 들어온 경우 먼저 들어온 type에 따라 6:4, 4:6을 결정한다. 할당된 10 단위가 끝나게 되면 위의 로직을 반복하는 알고리즘도 있다. 이 알고리즘을 k 방식이라 하자. 이 경우에는 cpu가 L을 수행하거나 수행하지 않고 있다가 H보다 M이 미세한 차이로 먼저 들어왔을 경우 H의 우선권이 사라지기 때문에 이런 경우에 좋지 않다.

예시)

P1 M 2 5 1

P2 H 3 5 1

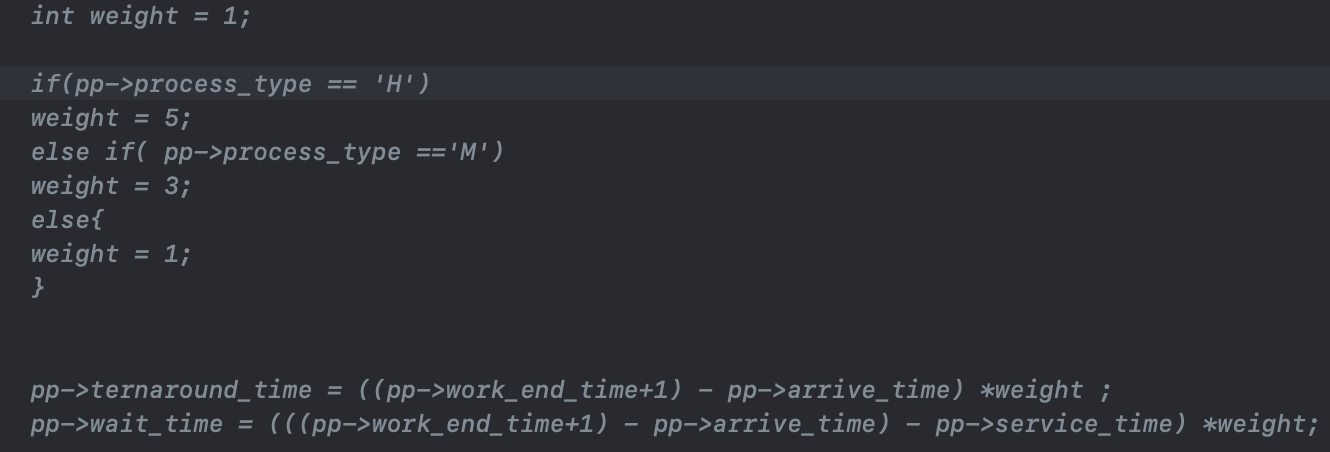
k 방식 결과

P1 \*\*\*\* \*

P2. \*\*\*\*\*

예시와 같은 경우에 본 과제 구현 방식이라면 H가 우선적으로 수행된다. 하지만 k 방식의 경우에는 P1을 기준으로 cpu time 4가 할당되고 그 후 H 큐에 cpu time이 할당되기 때문에 H 큐가 기다리는 상황이 발생한다.

성능을 테스트해보기 위해서 L을 수행하고 있다가 H, M이 들어온 경우 최대 10의 cpu time을 할당하는 방식(k 방식)과 본 과제를 구현한 방식과 비교해보았다. 테스트 데이터는 2개로 각 테스트 데이터에는 100개의 프로세스를 임의로 만들었다. 또한 H는 가중치 5, M은 가중치 3을 주어서 turnaround time과 wait time에 패널티를 주도록 하였다.



첫 번째 데이터에 대한 결과

본 과제 구현 방식:

CPU TIME: 1460

AVERAGE TURNAROUND TIME: 1387.30

AVERAGE WAITING TIME: 1342.80

k 방식

CPU TIME: 1460

AVERAGE TURNAROUND TIME: 1419.30

AVERAGE WAITING TIME: 1374.80

두 번째 데이터에 대한 결과

본 과제 구현 방식:

CPU TIME: 1460

AVERAGE TURNAROUND TIME: 1372.83

AVERAGE WAITING TIME: 1329.47

k 방식

CPU TIME: 1460

AVERAGE TURNAROUND TIME: 1406.08

AVERAGE WAITING TIME: 1362.72

위 테스트 케이스의 경우에는 본 과제 구현 방식이 더 좋은 성능을 나타냈다. 프로세스의 데이터 분포에 따라서 두 알고리즘의 성능차이가 생길 것으로 판단된다. 따라서, 프로세스의 입력 정보에 따라서 최적의 알고리즘은 시뮬레이션을 통해 결정되어야 할 것이다.

4. 결론

멀티 레벨 큐를 통해서 프로세스의 타입에 따른 cpu time scheduling을 다양한 방식으로 구현할 수 있음을 파악하였다. 실제 OS에서 이것보다 더 많은 프로세스의 종류를 스케줄링하므로 이론적 알고리즘 성능 분석과 시뮬레이션을 통하여 프로세스 데이터에 따른 최적의 알고리즘을 결정할 수 있음을 파악하였다.

또한, short term scehduling도 중요하지만 mid term , long term scheduling을 함께 생각하지 않으면 프로세스 관리의 효율성이 떨어질 수 있음을 느꼈다. 이번 과제를 통해 OS의 성능 예측이나 상황에 따른 스케줄링 기법에 대한 고찰을 할 수 있는 시간이었다. 산업공학과에서 시뮬레이션 과목을 수강하였는데 여기에서 시뮬레이션을 직접 구현할 때 여러 요소를 고려할 수 있음을 파악하였다.

실행결과

