

CPU thread scheduler

운영체제 팀프로젝트_3조

2023111394 정보보호학과 육은서

2023111410 정보보호학과 정수민

2023111417 정보보호학과 함은지

2023111420 정보보호학과 황선영

운영체제 김태호 교수님

제출일: 2024년 12월 9일

<목차>

1. 프로그램 설명

- A. Cpuschedule USAGE
- B. Cpuschedule main 함수
- C. Cpuschedule 함수
- D. FCFS
- E. SJF 비선점, 선점
- F. LJF 비선점, 선점
- G. Priority 비선점, 선점
- H. RR

2. 프로그램 실행 화면

3. 팀원별 역할 분담

4. 참고 문헌

1. 프로그램 설명

A. Cpuschedule USAGE

```
sficheu@BOOK-0ASDNN1RUN:~/OSScheduler$ ls
cpuschedule.c  thread.txt  thread_result.txt
sficheu@BOOK-0ASDNN1RUN:~/OSScheduler$ gcc cpuschedule.c -o cpuschedule
sficheu@BOOK-0ASDNN1RUN:~/OSScheduler$ ls
cpuschedule  cpuschedule.c  thread.txt  thread_result.txt
```

thread scheduling 시뮬레이션을 구현한 코드인 cpuschedule.c를 gcc한다.

```
sficheu@BOOK-0ASDNN1RUN:~/OSScheduler$ ./cpuschedule < thread.txt
```

./로 cpuschedule을 실행함과 동시에 thread.txt를 입력받는다.

```
sficheu@BOOK-0ASDNN1RUN:~/OSScheduler$ ls
cpuschedule  cpuschedule-test.c  cpuschedule.c  thread.txt  thread_result.txt
sficheu@BOOK-0ASDNN1RUN:~/OSScheduler$ gcc cpuschedule-test.c -o cpuschedule-test
sficheu@BOOK-0ASDNN1RUN:~/OSScheduler$ ls
cpuschedule  cpuschedule-test  cpuschedule-test.c  cpuschedule.c  thread.txt  thread_result.txt
```

시험 자동화를 위한 프로그램을 구현한 코드인 cpuschedule-test.c를 gcc한다.

```
sficheu@BOOK-0ASDNN1RUN:~/OSScheduler$ ./cpuschedule-test cpuschedule\<thread.txt thread_result.txt
```

./로 cpuschedule-test를 실행함과 동시에 cpuschedule에 thread.txt를 입력받고, 그 결과를 thread_result.txt와 비교한다.

이때 < 앞에 역슬래시(₩)가 없다면, 쉘 자체에서 리다이렉션 처리가 되어 thread.txt thread_result.txt가 입력 리다이렉션 되기 때문에 역슬래시를 넣어줬다.

B. Cpuschedule main 함수

```
int i = 0;
char input[256];
int threadnum = 0;
int time_slice = 1;
```

입력받은 스레드의 개수를 추적하기 위한 i 변수, 사용자로부터 입력받는 문자열을 저장하기 위한 input 배열, 입력된 총 스레드의 개수를 저장하기 위한 threadnum 변수와 RR에서 시간 할당량을 사용하기 위한 time_slice 변수를 선언 및 초기화한다.

```
while (fgets(input, sizeof(input), stdin)) {
    if (strcmp(input, "E₩n") == 0 || strcmp(input, "E") == 0) {
        break;
    }
}
```

fgets 함수를 통해 사용자의 입력을 읽어온다. 이때 들어오는 입력은 공백으로 구분되는 문자열이다. 이후 입력값이 E로 들어오면 입력을 종료한다.

```

char* token = strtok(input, " ");
if (token != NULL) {
    strncpy(th[i].tid, token, sizeof(th[i].tid) - 1);
    th[i].tid[sizeof(th[i].tid) - 1] = '\0';

    token = strtok(NULL, " ");
    if (token != NULL) {
        th[i].arrtime = atoi(token);
    } else {
        printf("Error: No data\n");
        continue;
    }

    token = strtok(NULL, " ");
    if (token != NULL) {
        th[i].exetime = atoi(token);
    } else {
        printf("Error: No data\n");
        continue;
    }

    token = strtok(NULL, " ");
    if (token != NULL) {
        th[i].priority = atoi(token);
    } else {
        printf("Error: No data\n");
        continue;
    }
}

```

strtok을 사용하여 문자열을 파싱한다. 공백으로 구분되는 입력값을 각각의 필드로 분리하여 tid, arrtime, exetime, priority의 각 스레드 정보를 th 배열에 저장한다.

```

else {
    printf("Error: No data\n");
    continue;
}

if (i >= MAX) {
    printf("WARNING: overflow\n");
    break;
}

```

만약 입력값이 부족하거나 올바르지 않다면 에러 메시지가 출력되고, 스레드의 개수가 최대 개수를 초과하면 경고 메시지와 함께 프로그램이 종료된다.

```
threadnum = i;
```

입력된 스레드의 총 개수를 저장한다.

```
FCFS(threadnum);  
SJF(threadnum);  
SJFpree(threadnum);  
LJF(threadnum);  
LJFpree(threadnum);  
PriorityNonPreemptive(threadnum);  
PriorityPreemptive(threadnum);  
RR(threadnum, time_slice);
```

FCFS, SJF 비선점형과 선점형, LJF 비선점형과 선점형, priority 비선점형과 선점형, 그리고 RR 알고리즘 함수를 호출한다.

```
const char* algo[] = {  
    "FCFS",  
    "SJF (non preemptive)",  
    "SJF (preemptive)",  
    "LJF (non preemptive)",  
    "LJF (preemptive)",  
    "priority(non preemptive)",  
    "priority (preemptive)",  
    "RR"  
};  
  
printf("\n\nResults\n\nCompleted Time\n\nTurnaround Time\n\nWaiting Time\n\n");  
printf("-----\n\n");  
  
for (int j = 0; j < 8; j++) {  
    printf("%-25s\n%.2f\n%.2f\n%.2f\n", algo[j], CT[j], TAT[j], WT[j]);  
}
```

각 알고리즘의 이름을 algo 배열에 저장하고, CT와 TAT, WT를 반복적으로 출력한다.

C. FCFS

```
int nn = 0;  
int time = 0;  
if (n == 0) {  
    return -1;  
}
```

처리된 스레드의 수를 저장하는 nn 변수, 현재 시간을 추적하는 time 변수를 선언 및 초기화한다. 이후 스레드의 개수가 0일 경우, 실패로 간주하고 -1을 반환한다.

```
Thread th_copy[MAX];  
memcpy(th_copy, th, sizeof(Thread) * n);
```

각 함수에서 독립적으로 실행이 가능하게 하기 위해, 원본 스레드 데이터인 th를 th_copy로 복사하여 사용한다.

```
while (nn < n)
```

처리된 스레드인 nn이 전체 스레드인 n의 수보다 적을 때까지 반복한다.

```
for (int i = 0; i < n; i++) {  
    if (th_copy[i].exetime <= 0) { continue; }  
    if (idx == -1 || th_copy[i].arrtime < th_copy[idx].arrtime) {  
        idx = i;  
    }  
}
```

for 문을 통해 이미 완료된 스레드는 건너뛰면서, 도착시간이 가장 이른 스레드를 선택하도록 한다.

```
if (th_copy[idx].arrtime > time) {  
    printf("- (%d)\n%d : ", th_copy[idx].arrtime, th_copy[idx].arrtime + time);  
    time += th_copy[idx].arrtime;  
}  
  
printf("%s (%d)\n%d : ", th_copy[idx].tid, th_copy[idx].exetime, time + th_copy[idx].exetime);
```

if 문을 통해 스레드의 도착 시간을 확인한다. 만약 스레드의 도착 시간이 현재 시간보다 큰 경우에는 대기 시간을 추가한다. 이후 현재 시간을 스레드 도착 시간만큼 더해 업데이트 한다.

스레드 ID와 실행 시간, 스레드가 실행되는 시간을 출력한다.

```
WT[0] += time - th_copy[idx].arrtime;  
TAT[0] += time - th_copy[idx].arrtime + th_copy[idx].exetime;  
CT[0] += time + th_copy[idx].exetime;
```

대기 시간(WT)과 반환 시간(TAT), 완료 시간(CT)를 구한다.

```
time += th_copy[idx].exetime;  
th_copy[idx].exetime = 0;  
nn += 1;
```

스레드 처리가 완료되었으면 time에 실행 시간을 더하고, 현재 스레드를 완료 처리한 다음 완료된 스레드(nn)의 수를 증가한다.

```
WT[0] /= n;  
TAT[0] /= n;  
CT[0] /= n;
```

대기 시간(WT)과 반환 시간(TAT), 완료 시간(CT)의 평균을 계산한다.

D. SJF 비선점, 선점

a. SJF (Non-Preemptive)

```
int nn = 0;
int time = 0;
```

처리된 스레드의 수를 저장하는 nn 변수, 현재 시간을 추적하는 time 변수를 선언 및 초기화한다.

```
Thread th_copy[MAX];
memcpy(th_copy, th, sizeof(Thread) * n);
```

각 함수에서 독립적으로 실행이 가능하게 하기 위해, 원본 스레드 데이터인 th를 th_copy로 복사하여 사용한다.

```
int idx = -1;
int min = MAX;
for (int i = 0; i < n; i++) {
    if (th_copy[i].exetime <= 0) { continue; }
    if (th_copy[i].arrtime <= time && th_copy[i].exetime < min) {
        min = th_copy[i].exetime;
        idx = i;
    }
}
```

실행하는 스레드의 인덱스를 나타내는 idx 변수와 탐색하는 작업 중 가장 짧은 실행 시간을 나타내는 min 변수를 선언하고 초기화한다.

for 문과 if 문을 통해 이미 완료된 스레드는 건너뛰면서, 실행 시간이 가장 짧은 스레드를 선택하도록 한다.

```
if (idx == -1) {
    time++;
    continue;
}
```

만약 실행 가능한 작업이 없다면, time을 증가시켜 다음 작업을 대기시킨다.

```
if (th_copy[idx].arrtime > time) {
    printf("- (%d)\n%d : ", th_copy[idx].arrtime, th_copy[idx].arrtime + time);
    time = th_copy[idx].arrtime;
}
printf("%s (%d)\n%d : ", th_copy[idx].tid, th_copy[idx].exetime, time + th_copy[idx].exetime);
```

if 문을 통해 스레드의 도착 시간을 확인한다. 만약 스레드의 도착 시간이 현재 시간보다 큰 경우에는 대기 시간을 추가한다. 이후 현재 시간을 스레드 도착 시간만큼 더해 업데이트 한다.

스레드 ID와 실행 시간, 스레드가 실행되는 시간을 출력한다.

```
WT[1] += time - th_copy[idx].arrtime;
```

```
TAT[1] += time - th_copy[idx].arrtime + th_copy[idx].exetime;  
CT[1] += time + th_copy[idx].exetime;
```

대기 시간(WT)과 반환 시간(TAT), 완료 시간(CT)를 구한다.

```
time += th_copy[idx].exetime;  
th_copy[idx].exetime = 0;  
nn += 1;
```

스레드 처리가 완료되었으면 time에 실행 시간을 더하고, 현재 스레드를 완료 처리한 다음 완료된 스레드(nn)의 수를 증가한다.

```
WT[1] /= n;  
TAT[1] /= n;  
CT[1] /= n;
```

대기 시간(WT)과 반환 시간(TAT), 완료 시간(CT)의 평균을 계산한다.

b. SJF (Preemptive)

```
int left[MAX];  
Thread th_copy[MAX];  
memcpy(th_copy, th, sizeof(Thread) * n);  
for (int i = 0; i < n; i++) {  
    left[i] = th_copy[i].exetime;  
}
```

남은 실행 시간을 저장하는 left 배열을 선언하고, if 문을 통해 실행하는 동안 남은 시간이 감소하며 0이 되면 완료되는 것으로 간주한다.

또한 각 함수에서 독립적으로 실행이 가능하게 하기 위해, 원본 스레드 데이터인 th를 th_copy로 복사하여 사용한다.

```
int time = 0;  
int completed = 0;  
int current_thread = -1;
```

현재 시간을 추적하는 time 변수와 완료된 작업 수를 나타내는 completed 변수, 현재 실행 중인 작업의 인덱스를 저장하는 current_thread 변수를 선언하고 초기화한다.

```
int idx = -1;  
int min = MAX;  
for (int j = 0; j < n; j++) {  
    if (left[j] > 0 && th[j].arrtime <= time && left[j] < min) {  
        min = left[j];  
        idx = j;  
    }  
}
```

실행하는 스레드의 인덱스를 나타내는 idx 변수와 탐색하는 작업 중 가장 짧은 실행 시

간을 나타내는 min 변수를 선언하고 초기화한다.

for 문과 if 문을 통해 이미 완료된 스레드는 건너뛰면서, 남은 실행 시간이 가장 짧은 스레드를 선택하도록 한다.

```
if (idx == -1) {  
    time++;  
    continue;  
}
```

만약 실행 가능한 작업이 없다면, time을 증가시켜 다음 작업을 대기시킨다.

```
if (current_thread != idx) {  
    printf("%s (%d)\\n", th_copy[idx].tid, left[idx]);  
    current_thread = idx;  
}
```

만약 현재 실행 중인 작업과 새로 선택되는 작업이 다르다면, 전환되는 새 작업의 ID와 남은 시간을 출력하고 current_thread를 새 작업으로 업데이트한다.

```
int duration = left[idx];  
time += duration;  
left[idx] = 0;  
completed++;
```

선택된 작업의 남은 실행 시간을 나타내는 duration 변수를 선언하고 초기화한다.

이후 현재 시간에 남은 실행 시간만큼 증가시키고, 스레드를 완료 처리한다.

```
CT[2] += time;  
TAT[2] += time - th_copy[idx].arrtime;  
WT[2] += time - th_copy[idx].arrtime - th_copy[idx].exetime;
```

대기 시간(WT)과 반환 시간(TAT), 완료 시간(CT)를 구한다.

```
if (completed < n) {  
    printf("%d : ", time);  
}  
printf("%d : #\\n", time);
```

완료된 작업이 남아있는 경우, 현재 시간을 출력하고 모든 작업이 완료되면 종료 메시지를 출력한다.

```
CT[2] /= n;  
TAT[2] /= n;  
WT[2] /= n;
```

대기 시간(WT)과 반환 시간(TAT), 완료 시간(CT)의 평균을 계산한다.

E. LJF 비선점, 선점

a. LJF (Non-Preemptive)

```
int nn = 0;
int time = 0;
```

처리된 스레드의 수를 저장하는 nn 변수, 현재 시간을 추적하는 time 변수를 선언 및 초기화한다.

```
Thread th_copy[MAX];
memcpy(th_copy, th, sizeof(Thread) * n);
```

각 함수에서 독립적으로 실행이 가능하게 하기 위해, 원본 스레드 데이터인 th를 th_copy로 복사하여 사용한다.

```
int idx = -1;
int max = -1;
for (int i = 0; i < n; i++) {
    if (th_copy[i].exetime > 0 && th_copy[i].arrtime <= time && th_copy[i].exetime > max) {
        max = th_copy[i].exetime;
        idx = i;
    }
}
```

실행하는 스레드의 인덱스를 나타내는 idx 변수와 탐색하는 작업 중 가장 긴 실행 시간을 나타내는 max 변수를 선언하고 초기화한다.

for 문과 if 문을 통해 이미 완료된 스레드는 건너뛰면서, 실행 시간이 가장 긴 스레드를 선택하도록 한다.

```
if (idx == -1) {
    time++;
    continue;
}
```

만약 실행 가능한 작업이 없다면, time을 증가시켜 다음 작업을 대기시킨다.

```
printf("%s (%d)\n%d : ", th_copy[idx].tid, th_copy[idx].exetime, time + th_copy[idx].exetime);
```

스레드 ID와 실행 시간, 스레드가 실행되는 시간을 출력한다.

```
WT[3] += time - th_copy[idx].arrtime;
TAT[3] += time - th_copy[idx].arrtime + th_copy[idx].exetime;
CT[3] += time + th_copy[idx].exetime;
```

대기 시간(WT)과 반환 시간(TAT), 완료 시간(CT)를 구한다.

```
time += th_copy[idx].exetime;
th_copy[idx].exetime = 0;
nn += 1;
```

스레드 처리가 완료되었으면 time에 실행 시간을 더하고, 현재 스레드를 완료 처리한 다음 완료된 스레드(nn)의 수를 증가한다.

```
WT[3] /= n;
```

```
TAT[3] /= n;
CT[3] /= n;
```

대기 시간(WT)과 반환 시간(TAT), 완료 시간(CT)의 평균을 계산한다.

b. LJF (Preemptive)

```
int findLongestJob(int n, int time) {
    int idx = -1;
    for (int i = 0; i < n; i++) {
        if (th[i].arrtime <= time && left[i] > 0) {
            if (idx == -1 || left[idx] < left[i]) {
                idx = i;
            }
        }
    }
    return idx;
}
```

findLongestJob 함수를 통해 남은 실행 시간이 가장 긴 작업을 찾는다.

실행하는 스레드의 인덱스를 나타내는 idx 변수를 선언하고 초기화한다.

for 문과 if 문을 통해 현재 시간까지 도착한 작업과 실행이 완료되지 않은 작업에서만 실행하고, 남은 실행 시간이 가장 긴 스레드를 선택하도록 한다.

선택된 작업의 인덱스 값인 idx를 반환하지만, 만약 실행 가능한 작업이 없다면 -1을 반환한다.

```
int time = 0;
int completed = 0;
int prev_idx = -1;
int prev_time = 0;
for (int i = 0; i < n; i++) {
    left[i] = th[i].exetime;
}
```

현재 시간을 나타내는 time 변수, 완료된 작업 수를 나타내는 completed 변수, 이전에 실행 중이었던 작업의 인덱스를 나타내는 prev_idx 변수와 이전 작업의 시작 시간을 나타내는 prev_time 변수를 선언하고 초기화한다.

if 문을 통해 실행하는 동안 남은 시간이 감소하며 0이 되면 완료되는 것으로 간주한다.

```
int idx = prev_idx;
int new_job_arrived = 0;
if (prev_idx == -1)
    new_job_arrived = 1;
```

이전 작업의 인덱스를 idx에 저장하고, 새로운 작업의 도착 여부를 나타내는

new_job_arrived 변수를 선언 및 초기화한다.

만약 현재 실행 중인 작업이 없으면 new_job_arrived를 1로 설정한다.

```
for (int i = 0; i < n; i++) {  
    if (th[i].arrtime == time) {  
        new_job_arrived = 1;  
        break;  
    }  
}
```

for 문과 if 문을 통해 새로운 작업이 현재 시간에 도착했는지 확인한다.

작업의 도착 시간이 현재 시간과 같다면, new_job_arrived를 1로 설정한다.

```
if (new_job_arrived || (prev_idx != -1 && left[prev_idx] == 0)) {  
    idx = findLongestJob(n, time);  
}
```

new_job_arrived가 1이거나, 이전에 실행하던 작업이 모두 완료됐을 때, 앞선 findLongestJob 함수를 통해 남은 작업 중 남은 실행 시간이 가장 긴 작업을 선택한다.

```
if (idx == -1) {  
    time++;  
    continue;  
}
```

만약 실행 가능한 작업이 없다면, time을 증가시켜 다음 작업을 대기시킨다.

```
if (idx != prev_idx) {  
    if (prev_idx != -1 && left[prev_idx] > 0) {  
        printf("%s (%d)\n%d : ", th[prev_idx].tid, time - prev_time, time);  
    }  
    prev_time = time;  
    prev_idx = idx;  
}
```

만약 현재 실행 중인 작업과 이전 작업이 다르다면, 이전 작업의 ID와 실행 시간, 남은 시간을 출력하고 prev_time과 prev_idx를 현재 작업으로 업데이트한다.

```
left[idx]--;  
time++;
```

선택된 작업의 남은 실행 시간을 감소하고, 현재 시간을 증가한다.

```
if (left[idx] == 0) {  
    completed++;  
    printf("%s (%d)\n%d : ", th[idx].tid, time - prev_time, time);  
}
```

작업이 완료되면, completed를 증가하고, 완료된 작업의 ID와 실행 시간, 종료 시간을 출

력한다.

```
WT[4] += time - th[idx].arrtime - th[idx].exetime;
TAT[4] += time - th[idx].arrtime;
CT[4] += time;
```

대기 시간(WT)과 반환 시간(TAT), 완료 시간(CT)를 구한다.

```
WT[4] /= n;
TAT[4] /= n;
CT[4] /= n;
```

대기 시간(WT)과 반환 시간(TAT), 완료 시간(CT)의 평균을 계산한다.

F. Priority 비선점, 선점

a. Priority (Non-Preemptive)

```
int nn = 0;
int time = 0;
```

처리된 스레드의 수를 저장하는 nn 변수, 현재 시간을 추적하는 time 변수를 선언 및 초기화한다.

```
Thread th_copy[MAX];
memcpy(th_copy, th, sizeof(Thread) * n);
```

각 함수에서 독립적으로 실행이 가능하게 하기 위해, 원본 스레드 데이터인 th를 th_copy로 복사하여 사용한다.

```
int idx = -1;
for (int i = 0; i < n; i++) {
    if (th_copy[i].exetime > 0 && th_copy[i].arrtime <= time) {
        if (idx == -1 || th_copy[i].arrtime < th_copy[idx].arrtime ||
            (th_copy[i].arrtime == th_copy[idx].arrtime && th_copy[i].priority < th_copy[idx].priority)) {
            idx = i;
        }
    }
}
```

실행되는 스레드의 인덱스를 나타내는 idx 변수를 선언하고 초기화한다.

for 문과 if 문을 통해 이미 완료된 스레드는 건너뛰면서, 도착 시간이 빠른 작업을 우선 선택하고, 만약 도착 시간이 같다면 우선 순위가 높은 작업을 선택하도록 한다.

```
if (idx == -1) {
    int any_task_left = 0;
    for (int i = 0; i < n; i++) {
        if (th_copy[i].exetime > 0) {
```

```

        any_task_left = 1;
        break;
    }
}
if (any_task_left == 0) {
    break;
}
time++;
continue;
}

```

만약 실행 가능한 작업이 없다면, 실행할 작업이 남아있는지 확인하는 any_task_left 변수를 선언하고 초기화한 후 time을 증가시켜 다음 작업을 대기시킨다.

이때 for 문과 if 문을 통해 작업 실행 시간이 남아있다면 any_task_left를 1로 설정하지만, 모든 작업이 완료되어 any_task_left가 0이라면 종료한다.

```
printf("%s (%d)\n%d : ", th_copy[idx].tid, th_copy[idx].exetime, time + th_copy[idx].exetime);
```

스레드 ID와 실행 시간, 스레드가 실행되는 시간을 출력한다.

```

WT[5] += time - th_copy[idx].arrtime;
TAT[5] += time - th_copy[idx].arrtime + th_copy[idx].exetime;
CT[5] += time + th_copy[idx].exetime;

```

대기 시간(WT)과 반환 시간(TAT), 완료 시간(CT)를 구한다.

```

time += th_copy[idx].exetime;
th_copy[idx].exetime = 0;
nn += 1;

```

스레드 처리가 완료되었으면 time에 실행 시간을 더하고, 현재 스레드를 완료 처리한 다음 완료된 스레드(nn)의 수를 증가한다.

```

WT[5] /= n;
TAT[5] /= n;
CT[5] /= n;

```

대기 시간(WT)과 반환 시간(TAT), 완료 시간(CT)의 평균을 계산한다.

b. Priority (Preemptive)

```

int findpriority(int n, int time) {
    int idx = -1;
    for (int i = 0; i < n; i++) {
        if (th[i].arrtime <= time && left[i] > 0) {
            if (idx == -1 || th[idx].priority < th[i].priority) {
                idx = i;
            }
        }
    }
}

```

```

    }
}
return idx;
}

```

findpriority() 함수는 현재 시간에 실행 가능한 스레드 중에서 우선순위가 가장 높은 스레드를 선택하도록 하는 함수다.

초깃값이 -1인 실행 가능한 스레드가 없는 상태로 모든 스레드를 순회하여 arrtime이 현재 시간보다 이하, 남은 실행시간(left[i])가 0보다 크고 기존 idx에 저장된 스레드보다 우선순위가 높은 걸 찾는다면 idx를 해당 스레드의 인덱스로 업데이트 하는 식으로 구성된다.

```

int PriorityPreemptive(int n) {
    printf("WnPriority (Preemptive)Wn");
    int time = 0;
    int completed = 0;
    int prev_idx = -1;
    int prev_time = 0;
    for (int i = 0; i < n; i++) {
        left[i] = th[i].exetime;
    }
}

```

PriorityPreemptive() 함수는 선점형 우선순위 알고리즘을 실행하는 함수다.

```

while (completed < n) {
    int idx = findpriority(n, time);
    if (idx == -1) {
        time++;
        continue;
    }
    if (idx != prev_idx) {
        if (prev_idx != -1) {
            printf("%d : %s (%d) Wn", prev_time, th[prev_idx].tid, time - prev_time);
        }
        prev_time = time;
        prev_idx = idx;
    }
    left[idx]--;
    time++;
    if (left[idx] == 0) {

```

```

        completed++;
        WT[6] += time - th[idx].arrtime - th[idx].exetime;
        TAT[6] += time - th[idx].arrtime;
        CT[6] += time;
    }
}

```

While문이 중요한 부분인데, 처음에 정의했던 findpriority()를 호출해 현재 실행할 수 있는 우선순위가 가장 높은 스레드의 인덱스를 선택하게 된다.

그리고 현재 실행하는 스레드(idx)와 이전에 실행한 스레드(prev_idx)가 다르다면 prev_idx의 실행 시간 정보를 출력하고 prev_time을 현재 시간으로 업데이트, prev_idx를 현재 실행 중인 스레드로 업데이트 하여 스레드 변경 처리를 해준다.

시간을 증가시키면서 선택된 스레드의 남은 실행이 0이 되면 해당 스레드의 WT, TAT, CT를 업데이트 하게 된다.

```

    if (prev_idx != -1) {
        printf("%d : %s (%d) %n", prev_time, th[prev_idx].tid, time - prev_time);
    }
    printf("%d : #%n", time);
    WT[6] /= n;
    TAT[6] /= n;
    CT[6] /= n;
    return 0;
}

```

그리고 마지막으로 실행된 스레드의 실행 정보를 출력한 후에 WT, TAT, CT의 6번째 인덱스 값에 누적 후 스레드 개수로 나눠서 평균을 계산한다.

G. RR

```

int left[MAX];
int completed = 0;
int time = 0;
for (int i = 0; i < n; i++) {
    left[i] = th[i].exetime;
}
printf("0 : ");

```

필요한 변수들을 초기화 후 실행시간을 left 배열에 옮겨준다. 시작 시간인 '0 : '을 먼저 출력한다.

```

while (completed < n) {
    for (int i = 0; i < n; i++) {
        if (left[i] > 0 && th[i].arrtime <= time) {

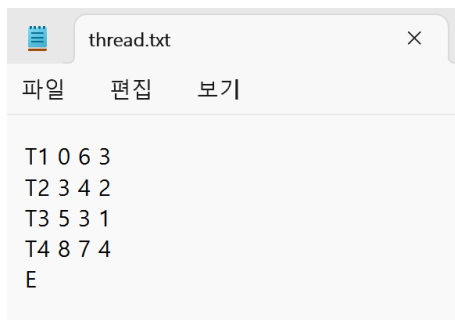
```


While문은 완료된 스레드가 기존 스레드 개수보다 작을 동안 진행된다. 스레드마다 검사를 하는데, 실행시간이 남아있고 현재 시간보다 도착시간이 같거나 작을 때(이미 도착해 있음) 조건문 내부로 들어간다.

```
if (left[i] > time_slice) {
    printf("%s(%d)\n", th[i].tid, time_slice);
    left[i] -= time_slice;
    time += time_slice;
}
else {
    printf("%s(%d)\n", th[i].tid, left[i]);
    time += left[i];
    left[i] = 0;
    completed++;
    CT[7] += time;
    TAT[7] += time - th[i].arrtime;
    WT[7] += time - th[i].arrtime - th[i].exetime;
}
```

조건문 내부는 위와 같다. Left[i]가 time_slice보다 크면 자른다. 현재 시간은 슬라이스만큼 더하고 남은 시간(left[i])에는 슬라이스만큼 빼준다. Time_slice보다 작거나 같으면 현재 시간은 left[i]만큼 더하고 left[i]는 0으로 돌린다. 이후 completed 변수를 +1해준다.

2. 프로그램 실행 화면



```
thread.txt
파일 편집 보기
T1 0 6 3
T2 3 4 2
T3 5 3 1
T4 8 7 4
E
```

입력은 위와 같다.

```

FCFS
0 : T1 (6)
6 : T2 (4)
10 : T3 (3)
13 : T4 (7)
20 : #

SJF (Non-Preemptive)
0 : T1 (6)
6 : T3 (3)
9 : T2 (4)
13 : T4 (7)
20 : #

SJF (Preemptive)
0 : T1 (6)
6 : T3 (3)
9 : T2 (4)
13 : T4 (7)
20 : #

LJF (Non-Preemptive)
0 : T1 (6)
6 : T2 (4)
10 : T4 (7)
17 : T3 (3)
20 : #

LJF (preemptive)
0 : T1 (3)
3 : T2 (2)
5 : T1 (3)
8 : T4 (7)
15 : T3 (3)
18 : T2 (2)
20 : #

Priority (Non-preemptive)
0 : T1 (6)
6 : T2 (4)
10 : T3 (3)
13 : T4 (7)
20 : #

Priority (Preemptive)
0 : T1 (6)
6 : T2 (2)
8 : T4 (7)
15 : T2 (2)
17 : T3 (3)
20 : #

RR
0 : T1(1)
1 : T1(1)
2 : T1(1)
3 : T2(1)
4 : T1(1)
5 : T2(1)
6 : T3(1)
7 : T1(1)
8 : T2(1)
9 : T3(1)
10 : T4(1)
11 : T1(1)
12 : T2(1)
13 : T3(1)
14 : T4(1)
15 : T4(1)
16 : T4(1)
17 : T4(1)
18 : T4(1)
19 : T4(1)
20 : #

```

Results	Completed Time	Turnaround Time	Waiting Time
FCFS	12.25	8.25	3.25
SJF (non preemptive)	12.00	8.00	3.00
SJF (preemptive)	12.00	8.00	3.00
LJF (non preemptive)	13.25	9.25	4.25
LJF (preemptive)	15.25	11.25	6.25
priority(non preemptive)	12.25	8.25	3.25
priority (preemptive)	14.50	10.50	5.50
RR	14.75	10.75	5.75

출력화면은 위와 같다.

```

hellomynewuninuxxxxxxx@BOOK-D15UEBQLJP:~/OSsche$ ./cpuschedule-test cpuschedule\<thread.txt thread_result.txt
FCFS : PASS, PASS, PASS, PASS
SJF (non preemptive) : PASS, PASS, PASS, PASS
SJF (preemptive) : PASS, PASS, PASS, PASS
LJF (non preemptive) : PASS, PASS, PASS, PASS
LJF (preemptive) : PASS, PASS, PASS, PASS
priority(non preemptive) : PASS, PASS, PASS, PASS
priority (preemptive) : PASS, PASS, PASS, PASS
RR : PASS, PASS, PASS, PASS

```

자동화 프로그램을 사용한 출력화면은 위와 같다.

3. 팀원별 역할 분담

팀원 이름	역할
육은서	FCFS 함수 LJF(preemptive) 함수 priority(preemptive) 함수 자동화 프로그램
정수민	SJF(non preemptive) 함수 SJF(preemptive) 함수 RR 함수
함은지	priority(non preemptive) 함수

	priority(preemptive) 함수 RR 함수 자동화 프로그램
황선영	LJF(non preemptive) 함수 LJF(preemptive) 함수

4. 참고 문헌

CPU 스케줄링 시뮬레이터(C언어) . (2023). <https://velog.io/@strn18/OS-CPU-%EC%8A%A4%EC%BC%80%EC%A4%84%EB%A7%81-%EC%8B%9C%EB%AE%AC%EB%A0%88%EC%9D%B4%ED%84%B0>.