**Dynamic vCPU Allocation Module**

**- Cloud Computing Project –**

2011140138

황 태림

**개요**

**과제 구현 목표**

본 과제는 가상화 환경 구축과 가상 환경의 VCPU를 효율적으로 관리하여 최대 효율을 끌어내는 것을 목표로 한다. DVAM은 가상머신의 VCPU를 각 가상머신의 CPU 이용률에 따라 동적으로 재 할당하여 하이퍼바이저의 자원을 효율적으로 사용할 수 있도록 하는 모듈이다. 이러한 DVAM을 만들어 Linux 프로그래밍에 대한 지식 습득과 활용을 목표로 한다.

평가 방식 :

scheduler의 성능을 평가하기 위해 프로세스 개별 평가 요소는 다음과 같다.

Response time : 요청 후 응답까지 걸리는 시간

Waiting time : 프로세스가 ready queue에서 대기하는 시간

Turnaround time : 프로세스의 수행 시작부터 끝날 때까지 걸리는 시간

위의 요소를 토대로 CPU에 적용된 scheduler 성능은 다음 5가지 요소로 측정한다.

CPU utilization : 전체 시스템 시간 중 CPU가 작업을 처리하는 시간의 비율

Throughput : CPU가 단위 시간당 처리하는 프로세스의 개수

Average response time : 프로세스들의 response time 평균 시간. 짧으면 짧을수록 좋다.

Average Waiting time : 프로세스들의 waiting time 평균 시간. 짧으면 짧을수록 좋다.

Average Turnaround time : 프로세스들의 turnaround time 평균 시간. 짧을수록 좋다.

**본문**

**시뮬레이션 핵심 모듈 설명**

시뮬레이션의 진행 과정과 모듈의 구성 및 기능 설명은 위에서 진행하였다. 지금부터는 scheduler의 핵심 부분인 scheduling algorithm의 구현과 작동 방식을 pseudo code 등을 이용하여 설명한다.

// time quantum과 I/O 작업 시작 시간을 전달받는다. algorithm은 scheduler에서 사용할 알고리즘의 종류를 넘겨받는다

Func Scheduler(time\_quantum, IO\_start\_time, algorithm):

time = 0 // 시간 초기화

idle\_time = 0 // cpu의 유휴 시간 초기화

// 작업 종료 까지 반복하는 반복문에서 작업이 진행된다

while(time >= 0):

// Job queue에 process가 없을 때 까지 반복한다

if(job\_queue has process):

while(job\_queue.process is arrived):

ready\_queue <- job\_queue.process // Job queue의 process를 ready\_queue에 넣는다

// 본 기능은 매 시간 프로세스의 작업 처리가 실행되기 전에 Ready queue가 말그대로 준비된 상태 (I/O가 끝난 후의 process, Job queue에서 도착한 process, preemptive로 양보된 process를 담아 알고리즘 수행을 준비한다. 추가로, Waiting queue에서 CPU burst, I/O burst가 모두 끝난 process는 Ready queue를 거쳐 Running state에서 Terminated queue로 옮겨지게끔 구성되어 있다) 가 끝난 후에 process의 작업을 처리한다

// Waiting queue에 이전 시간 처리가 완료된 process를 Ready queue에 옮겨 넣는다

if(waiting\_queue has process AND process has no remain\_io):

ready\_queue <- waiting\_queue.process

switch(algorithm):

case FCFS:

do nothing

case Non\_preemptive\_SJF:

sort(ready\_queue, SHORTEST)

// 남은 cpu 작업 시간이 적은 순으로 정렬

case Preemptive\_SJF:

sort(ready\_queue, SHORTEST)

case Non\_preemptive\_Priority:

sort(ready\_queue, PRIORITY) // 우선순위가 높은 순으로 정렬

case Preemptive\_Priority:

sort(ready\_queue, PRIORITY)

case Round\_robin:

do nothing

case HRRN:

sort(ready\_queue, HRRN) // HRRN algorithm에서 사용하는 기준으로 정렬한다

case Shortest\_io\_first:

sort(ready\_queue, IO) // IO가 적은 순으로 정렬한다

// Running state가 비어있지 않은 경우를 따져

// process의 scheduling을 수행한다

if(running\_state is not null):

if(running\_state reaches at IO\_start\_time):

waiting\_queue <- running\_state

// Round robin 알고리즘은 이 때 최초의 time quantum으로 초기화된다

reset(time\_quantum)

// process의 작업이 이전 시간에 마쳐졌다면 확인 후 처리한다

// 이 때, waiting queue에서 넘어온 I/O 작업과

// CPU 작업이 모두 끝난 process가 일시적으로

// ready queue에 있는 상태이므로, 이 process를

// terminated queue로 보내야 한다

while(running\_state is done AND ready\_queue has process that is done):

terminated\_queue <- running\_state

// Round robin 알고리즘은 이 때 최초의 time quantum으로 초기화된다

reset(time\_quantum)

// 알고리즘에 preemptive가 가능한 경우에는

// preemptive 여부를 따져 process를 처리한다

// preemptive algorithm들과 HRRN, Round robin,

// shortest I/O first가 이에 해당한다

if(algorithm is preemptive ):

ready\_queue <- running\_state

running\_state <- ready\_queue.process

// Round robin 알고리즘은 이 때 최초의 time quantum으로 초기화된다

reset(time\_quantum)

// 위의 과정을 거쳐서 running state가 여전히

// 비어있다면, 모두 종료된 process는 다시

// terminated queue로 보내고, 작업 처리를 위한

// process를 ready queue에서 가져온다

if(running\_state is null):

while(running\_state is done AND ready\_queue has process that is done):

terminated\_queue <- running\_state

// Round robin 알고리즘은 이 때 최초의 time quantum으로 초기화된다

reset(time\_quantum)

// ready queue에 도착한 process가 있을 경우

if(ready\_queue has process):

running\_state <- ready\_queue.process

//처음 작업을 위해 들어온 process는

// process의 정보(도착 시간, 대기 시간, 시작 시간 등)

// 처리를 진행한다

time\_set(process)

// ready queue가 아무것도 없다면 cpu 유휴 시간을 늘린다

else:

idle\_time++

// 여기까지 진행 후 running state에 있는 process는

// CPU 작업을 시간 단위로 진행할 수 있다

if(running\_state is not null):

CPU\_work(running\_state)

// Round robin의 경우 time quantum의 남은 시간을 감소 키킨다

time\_quantum--

// Waiting queue에 있는 process도 작업을 수행한다

if(waiting\_queue has process):

IO\_work(waiting\_queue.process)

// 모든 작업이 종료됐는지 확인 후 종료됐으면

// 평가를 진행한다

if(terminated\_queue.count has ALL processes):

evaluate()

// 알고리즘 종료

break

// time 변수를 증가시켜 scheduler의 시간 단위를 증가 시킨다

time++

// 모든 작업이 끝난 후 queue를 할당 해제 한다

free(ALL queue)

**CPU scheduling simulator 구동 및 성능 평가**

**맺음**

**시뮬레이터 정리 및 프로젝트 소감, 향후 나의 발전방향…**

이론으로 학습한 CPU scheduling을 직접 프로그래밍 하여 구현하면서 프로그래밍 실력 향상과 설계 단계부터 최종 테스트까지 발생한 문제를 해결하면서 얻은 해결능력은 쉽게 얻지 못할 좋은 기회였다고 본다. 데이터를 입력 후 결과를 비교하면서 scheduling algorithm의 성능을 분석하는 과정을 진행한 결과, 학습 내용을 더 깊이 이해할 수 있었다. 특히, I/O 작업을 포함한 복잡한 scheduling의 경우 이론적으로만 이해하는 데 어려움이 있는데, 시각적으로 확인을 하니 이해가 훨씬 수월하였다.

아쉬운 점은, 최초의 계획대로 GUI로 제작하여 사용자에게 더 직관적인 사용 유도와 깔끔한 결과 분석을 진행하고 싶었는데, 시간과 실력의 부족으로 인하여 그러지 못했다는 점이다. 전공 수업이 끝나더라도 GUI로 본 프로그램을 다시 만들어보고 싶은 열정을 심어주는 계기라 생각한다.

더 많은 scheduling algorithm을 구현하지 못한 것도 아쉽다. Multi-level queue나 서론부에서 소개한 rate monotonic, earliest deadline first등 구현 시 재미를 주며 지식의 깊이를 더할 수 있는 알고리즘이 있었지만 시간 부족으로 진행하지 못하였다. 추후 프로젝트의 시간 산정을 면밀히 하여 넉넉한 기간에 프로그램을 작성할 수 있도록 노력할 것이다.

Longest I/O first 의 알고리즘이 Shortest I/O first보다 훨씬 성능이 좋았지만, 안일하게 코드 작성 후 비교가 아닌, 어림 짐작으로 ‘이게 더 성능이 좋을 것이다’ 추측을 통해 둘 다 구현하지 않고 shortest I/O first만 구현한 점은 반성할 부분이다. 섣불리 확정하지 말고 설계 전 분석을 더 신중히 해야 함을 배울 수 있었다.

이 외에도 I/O를 random time에 시작하도록 하는 기능 미구현 등 아쉬움이 많은 프로젝트이다. 추후에 진행될 프로젝트에는 체크리스트 등을 작성하고, 코드 작성 전 시스템 설계나 분석에 더 많은 시간을 쏟는 등의 방식을 사용하며 아쉬움을 줄여나갈 수 있을 것이다.

**부록**

**소스코드**

**process.h**

#pragma once

#include <time.h>

// evaluation process structure

typedef struct EvalProcess{

int time\_start; // start time

int time\_pause; // paused time

int time\_end; // end time

int time\_res; // response time, 음수로 시작(첫 한 번만 하기 위해)

int time\_wait; // waiting time

int time\_turn; // turnaround time

int remain\_cpu; // remaining cpu time

int remain\_io; // remaining I/O time

} EvalProcess;

// process structure

typedef struct Process\* PtrProcess;

typedef struct Process{

int pid; // process id

int burst\_cpu; // CPU burst, 1 이상

int burst\_io; // I/O burst, 0 이상, 기본적으로 1초 뒤에 작업 수행

int arr\_time; // arrival time, 0 이상

int priority; // priority, 1 이상

EvalProcess eval\_info; // evaluation information

} Process;

// evaluation total structure

typedef struct EvalTotal\* PtrEvalTotal;

typedef struct EvalTotal {

int time\_start; // start time

int time\_end; // end time

double util\_cpu; // CPU utilization

double throughput; // throught

double awt; // average waiting time

double att; // average turnaround time

double art; // average response time

} EvalTotal;

PtrEvalTotal init\_evaluation();

void free\_evaluation(PtrEvalTotal et);

PtrProcess init\_process();

void free\_process(PtrProcess pc);

PtrProcess ran\_process(int pid, int burst\_cpu, int burst\_io, int arr\_time, int priority);

PtrProcess gen\_process(int pid, int burst\_cpu, int burst\_io, int arr\_time, int priority);