Producer & Consumer

Multithreading

HW2

32181928 박찬호

Dankook University

Mobile Systems Engineering

2022 Fall / Operating Systems

Left freedays : 0

Contents 목차

서론

Multi-programming & Process

사용자는 여러 프로그램을 동시에 실행하는 환경을 원한다. 이러한 상황을 Multi-Programming 이라고 부른다. 그러나 multi-programming 을 수행하기 위한 CPU 는 하나 또는 적은 개수 밖에 존재하지 않고, 경우에 따라 하나의 CPU 는 한 번에 하나의 프로그램을 실행할 수밖에 없다. CPU 가 하나 혹은 적은 개수로 존재하는 환경에서 OS 는 CPU 를 가상화하여 CPU 가 여러 개 존재하는 것 같은 illusion 을 만들어 낸다. 이때 꼭 필요한 개념이 바로 Process 이다.

Process 는 실행 중인 프로그램의 instance 이다. 프로그램 그 자체는 실행되기 이전에는 실행되기 위한 데이터의 집합으로 존재하고, 프로그램이 실행되어 process 가 생성되면 process 는 memory, register 등 여러 하드웨어의 상태와 연관 관계가 생기고 또한 Process 는 자체의 상태를 갖는다. OS 는 이러한 요소를 바탕으로 Process 의 실행 및 중단을 반복하며 시분할로 여러 프로그램이 동시에 실행되는 것처럼 illusion 을 형성하고, 이 과정에서 scheduler 는 scheduling policy 를 바탕으로 합리적인 시분할을 제공하기 위해 노력한다.

Thread

Multi-programming 을 구현하기 위해 시분할을 바탕으로 한 CPU 가상화와, 이를 뒷받침하는 메모리 가상화를 사용하였다. 그러나 현대적 컴퓨터 시스템에서 OS 는 CPU 가 한 순간에 하나의 명령어를 실행하는 것에서 벗어남으로써, parallelism 을 추구한다. 이를 위해 필요한 개념이 바로 Thread 이다. Thread 는 Process 와 유사한 상태를 가진다. 그러나 '한 Process 의 다른 여러 Thread'는 '다른 여러 프로세스'의 경우와 달리 동일한 주소 공간을 사용하고, Thread 별로 스택을 갖는다. 이로 인해 Thread 는 데이터를 공유하는 과정에서 고려할 점이 많다.

Multi-Threaded program 에서 특정 코드 영역을 실행할 때, 여러 thread 가 같은 코드를 실행하는 상황이 생긴다. 이때 단일 thread 에서 순차적으로 코드를 실행하는 것과 달리, 여러 thread 에서 코드를 실행하면 명령어의 실행 순서에 따라 전혀 다른 결과가 발생한다. 이러한 상황을 race condition 이라고 하고, 이러한 코드 영역을 critical section 이라고 부른다.

Mutex & Semaphore

여러 thread 가 critical section 에 접근하는 상황에서, race condition 이 발생하지 않도록 하기 위해 하나의 thread 만 해당 영역에 진입하도록 통제하는 것이 필요하다. Mutex 는 mutual exclusion 의 약자로 이러한 race condition 을 해결할 수 있다. Mutex 를 구현하기 위해서는 thread 가 critical section 에 진입할 때 lock 을 걸고 critical section 을 빠져나오면서 unlock 을 수행하면 된다. 이러한 과정에서 mutex 를 올바르게 구현하기 위해 test-and-set 등의 atomic instruction 이 사용된다. 또한 lock 이 걸린 시점에서 critical section 에 진입하지 못한 thread 가 busy wait 을 하지 않고 대기하다가 signal 을 받아 실행하도록 처리하기 위해 condition variable 을 사용할 수 있다.

이와 같은 상황에서 Semaphore 는 좋은 도구가 된다. Semaphore 는 mutex 와 Turing equivalent 하다. 즉, Semaphore 로 mutex 를 구현할 수 있고, mutex 로 semaphore 를 구현할 수 있다. Semaphore 는 critical section 에 진입할 수 있는 count 를 가지고 있고, critical section 에 진입한 thread 의 개수를 atomic instruction 으로 증가 또는 감소시키며 공유자원에 대한 접근을 관리하는 방법이다.

구현 목표

위의 개념과 프로젝트의 요구사항을 바탕으로, 구현 목표를 정해보자

1. prod_cons.c 를 수정하여 1 producer 1 consumer 상황에서 동작하도록 만든다.
2. Multiple consumer 가 동작하도록 수정한다.
3. Consumer 가 파일의 문자들에 대해 statistic 을 분석하고 출력하도록 한다.
4. Semaphore 를 사용하여 multiple producer – multiple consumer 상황에서 동작하도록 만든다.
5. Thread 의 증가에 따라 concurrency 가 더 향상되도록 수정한다.
6. 결과를 분석하고 insight 를 얻는다.

실행 방법

1. https://github.com/mobile-os-dku-cis-mse/2022-os-hw2 clone 받는다.
2. 32181928 branch 로 checkout 한다.

3-1. make all 을 입력하여 build 한 후 ./main {실행 파일} #producer #consumer #buffer size 꼴로 실행한다.

3-2. 또는 test.sh 를 bash 에서 실행한다.

본론

Pseudocode

본 프로젝트에서 Producer-Consumer 문제를 해결하기 위해 semaphore 를 사용한 pseudocode 는 아래와 같다.

Producer

1 Initialize variables

2 while

    1 sem_wait(empty)

    2 mutex_lock(lock)

    3 //critical section

    4 mutex_unlock(lock)

    5 sem_post(full)

3 end while

Consumer

1 Initialize variables

2 while

    1 sem_wait(full)

    2 mutex_lock(lock)

    3 //critical section

    4 mutex_unlock(lock)

    5 sem_post(empty)

    6 //consumer work

3 end while

위의 pseudocode 에서는 empty 와 full 이라는 별개의 count 를 가진 semaphore 를 사용하여, producer 가 buffer 에 item 을 추가하는 것과 consumer 가 buffer 에서 item 을 빼는 것을 통제한다. Producer 와 consumer 의 호출 이전에 empty 는 thread 의 개수로, full 0 으로 초기화 된 상태이다.

Producer 와 Consumer 는 sem_wait 를 통해 count 를 감소시키고, sem_post 를 통해 count 를 증가시킨다. 또한 critical section 진입 시에 mutex lock 을 사용한다.

Producer 는 critical section 진입 전에 empty 의 count 를 감소시키고 critical section 을 나오며 full 의 count 를 증가시킨다. Consumer 는 반대로 진입 전에 full 의 count 를 감소시키고 critical section 을 나오며 empty 의 count 를 증가시킨다. 이러한 방식을 통해, producer 는 empty 의 개수만큼 buffer 에 item 을 추가할 수 있고, critical section 을 빠져나오며 full 의 개수를 증가시켜 consumer 에게 buffer 에 consume 할 수 있는 item 이 있음을 알리는 역할을 수행한다. Consumer 는 반대로 full 의 개수만큼 item 을 consume 하고, empty 의 값을 증가시켜 buffer 에 추가할 수 있는 item 의 개수를 증가시킨다.

Pthread & Semaphore functions

실제 구현을 위해 사용되는 pthread & semaphore function 들은 아래와 같다.

pthread.h

int pthread_create(pthread_t		*thread,	Create thread
	const pthread_attr_t	*attr,	
	void	*(*start_routine)(void*),	
	void	*arg);	
int pthread_join(pthread_t		thread,	Wait for thread termination
	void	**value_ptr)	
void pthread_exit(void *value_ptr);			Terminates thread
int pthread_mutex_init(pthread_mutex_t		*mutex,	Mutex initialization
	const pthread_mutex_attr_t	*attr);	
int pthread_mutex_lock(pthread_mutex_t		*mutex);	Mutex lock
int pthread_mutex_unlock(pthread_mutex_t		*mutex);	Mutex unlock
int pthread_mutex_destroy(pthread_mutex_t		*mutex);	Mutex

semaphore.h

int sem_init(	sem_t	*sem,	Initialize semaphore
	int	pshared,	
	unsigned int	value)	
int sem_wait(	sem_t	*sem);	Lock semaphore
int sem_post(	sem_t	*sem)	Unlock semaphore
int sem_destroy(	sem_t	*sem)	Destroy semaphore

구현 결과

이를 통해 구현한 코드는 아래와 같다.

main.c

```
#include "prod_cons.h"
#include <pthread.h>

int main (int argc, char *argv[])
{
    pthread_t* prod;
    pthread_t* cons;
    int Nprod, Ncons;
    int buf_size;
    int rc;
    int *ret;
    int i;

    so_t *share = malloc(sizeof(so_t));
    memset(share, 0, sizeof(so_t));
    int stat[ASCII_SIZE] = { 0, };

// argument 관련 생략

    int** stats = malloc(sizeof(*stat) * Ncons);

    prod = malloc(sizeof(*prod) * Nprod);
    cons = malloc(sizeof(*cons) * Ncons);

    share->blocks = 512;
    share->nprod = Nprod;
    share->ncons = Ncons;

    FILE* rfile = fopen((char *) argv[1], "r");
    if (rfile == NULL) {
        perror("rfile");
        exit(0);
    }
    share->rfile = rfile;

    share->line = malloc(sizeof(*(share->line)) * share->blocks);
```

```
    share->buf_size = buf_size;

    sem_init(&(share->empty), 0, share->nprod);
    sem_init(&(share->full), 0, 0);

    pthread_mutex_init(&(share->mutex), NULL);

    for (i = 0 ; i < Nprod ; i++)
        pthread_create(&prod[i], NULL, producer, share);
    for (i = 0 ; i < Ncons ; i++)
        pthread_create(&cons[i], NULL, consumer, share);

// test 시 작업 없이 thread 의 비용만 측정하기 위한 no_work_prod 호출
    // for (i = 0 ; i < Nprod ; i++)
    //  pthread_create(&prod[i], NULL, no_work_prod, share);
    // for (i = 0 ; i < Ncons ; i++)
    //  pthread_create(&cons[i], NULL, no_work_cons, share);

    for (i = 0 ; i < Nprod ; i++) {
        rc = pthread_join(prod[i], (void **) &ret);
    }

    for (i = 0 ; i < Ncons ; i++) {
        rc = pthread_join(cons[i], (void **) &ret);
        stats[i] = (int*)ret;
    }

    for (i = 0; i < Ncons; i++) {
        for (int j = 0; j < ASCII_SIZE; j++) {

            stat[j] += stats[i][j];
        }
    }


// 출력 생략


    pthread_exit(NULL);
    pthread_mutex_destroy(&(share->mutex));
    sem_destroy(&(share->empty));
    sem_destroy(&(share->full));\

    exit(0);
}
```

prod_cons.c

```c
#include "prod_cons.h"

void *producer(void *arg) {

    so_t *so = arg;

    int *ret = malloc(sizeof(int));
    FILE *rfile = so->rfile;
    int buf_size = so->buf_size;
    char *line = malloc(sizeof(*line) * buf_size);
    int blocks = so->blocks;
    ssize_t read = 0;

    while (1) {

        sem_wait(&(so->empty));
        pthread_mutex_lock(&(so->mutex));
        read = fread(line, 1, buf_size, rfile);

        if (read == 0) {
            so->line[so->p_idx] = NULL;
            pthread_mutex_unlock(&(so->mutex));
            sem_post(&(so->full));
            break;
        }
        so->line[so->p_idx] = strdup(line);
        so->p_idx = (so->p_idx + 1) % blocks;
        pthread_mutex_unlock(&(so->mutex));
        sem_post(&(so->full));
    }
    free(line);

    *ret = 0;
    pthread_exit(ret);
}

void *consumer(void *arg) {
    so_t *so = arg;
    int *ret = malloc(sizeof(int));
    int *stat = calloc(ASCII_SIZE, sizeof(int));
    int len;
    char *line;
```

```
    int blocks = so->blocks;
    int arr[5] = { 1, 1, 1, 1, 1 };

    while (1) {
        sem_wait(&(so->full));
        pthread_mutex_lock(&(so->mutex));

        line = so->line[so->c_idx];

        if (line == NULL) {
            pthread_mutex_unlock(&(so->mutex));
            sem_post(&(so->full));
            sem_post(&(so->empty));
            break;
        }
        char* copy = strdup(line);

        free(so->line[so->c_idx]);
        so->line[so->c_idx] = NULL;
        so->c_idx = (so->c_idx + 1) % blocks;

        pthread_mutex_unlock(&(so->mutex));
        sem_post(&(so->empty));

// 실제 작업을 수행하는 영역
        char *cptr = copy;
        char *substr = NULL;
        char *brka = NULL;
        char *sep = "{}()[],;\" \n\t^";

        for (substr = strtok_r(cptr, sep, &brka); substr; substr =
strtok_r(NULL, sep, &brka)) {

            len = strlen(cptr);

            for (int i = 0 ; i < len ; i++) {
                int c = *cptr;
                if (c < 256 && c > 1) {
                    stat[c]++;
                }

                // overhead 를 더 크게 부여하기 위한 코드
                for (int j = 0; j < c; j++) {
                    arr[(len + j) % 5] *= c;
                }
```

```
                // more overhead

                cptr++;
            }
            cptr++;
            if (*cptr == '\0') break;

        }
    }

    *ret = stat;
    pthread_exit(ret);
}

void *no_work_cons(void *arg) {
// consumer 와 거의 동일한 코드이기 때문에 생략
}

void *no_work_prod(void *arg) {
// producer 와 거의 동일한 코드이기 때문에 생략
}
```

Test method

본 프로젝트에서는 thread 개수, buffer size, test 파일에 대해 다른 조건으로 여러 번 test 를 수행하기 위해 shell script 를 사용했다.

```
#!/bin/bash
make clear
make all

for filename in "/opt/android.tar" "/opt/FreeBSD9-orig.tar"
do
    for nthreads in {1..1001}
    do
        for buf_size in 4096 131072 1048576 32768 131072 524288 1048576
        do
```

```
        #echo "$filename - $nthreads prods - $nthreads cons - Buffer $buf_size
Bytes"
            echo "$filename - $nthreads prods - $nthreads cons - Buffer
$buf_size Bytes :" >> result
            { time ./main $filename $nthreads $nthreads $buf_size ; } 2>>
result
            echo " " >> result
        done
    done
done

echo -ne '\007'
```

결론

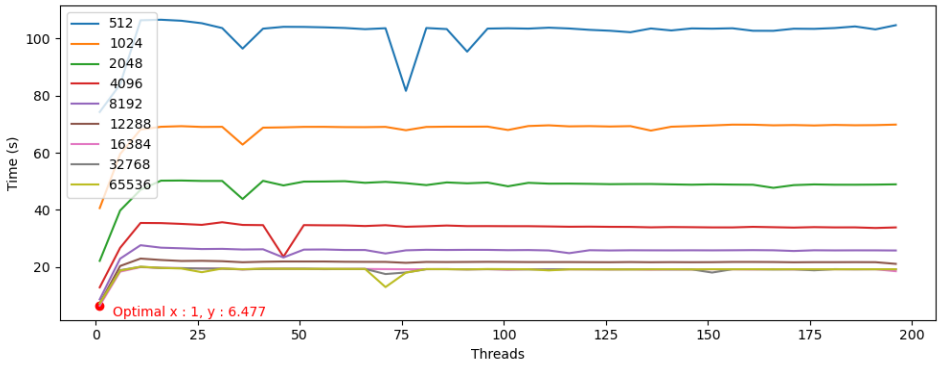
Test Result & Analysis

본 프로젝트에서는 많은 횟수의 test 를 수행했다. 모든 test 의 결과는 github 에 있으며, 여기서는 그 결과를 그린 graph 를 통해 확인한다. 모든 graph 의 x 축은 thread 의 개수, y 축은 실행 시간(s)이다. 여러 그래프가 하나의 plot 에 존재할 경우 각 그래프의 구분은 우상단에 표시하였다.

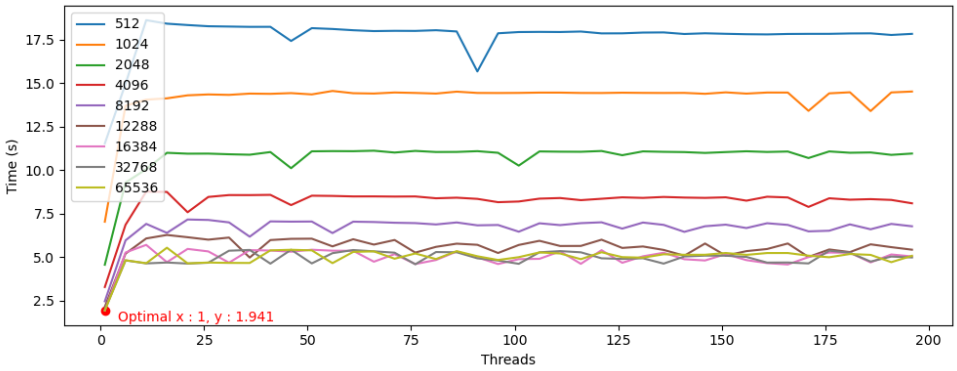
먼저 semaphore 등을 이용하여 multi thread 환경에서 동작이 가능하도록 구현한 후, critical section 내에 stat 을 분석하는 코드가 포함됐을 때의 test 결과는 아래와 같다.

Test-bad-result

/opt/android.tar



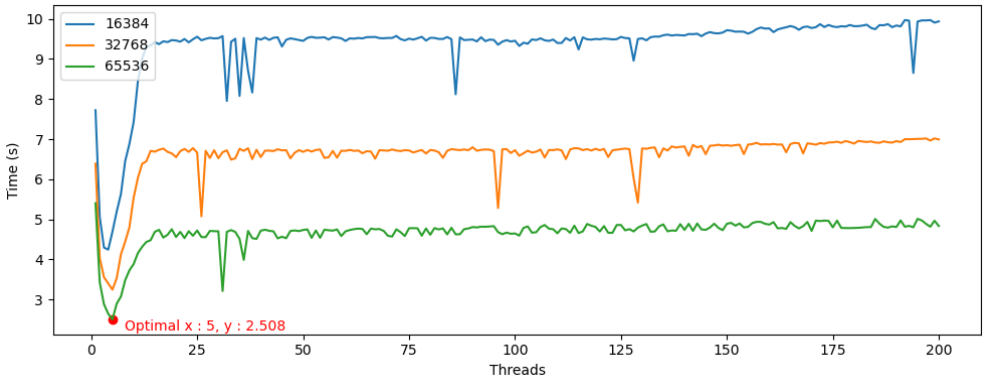
/opt/FreeBSD9-orig.tar



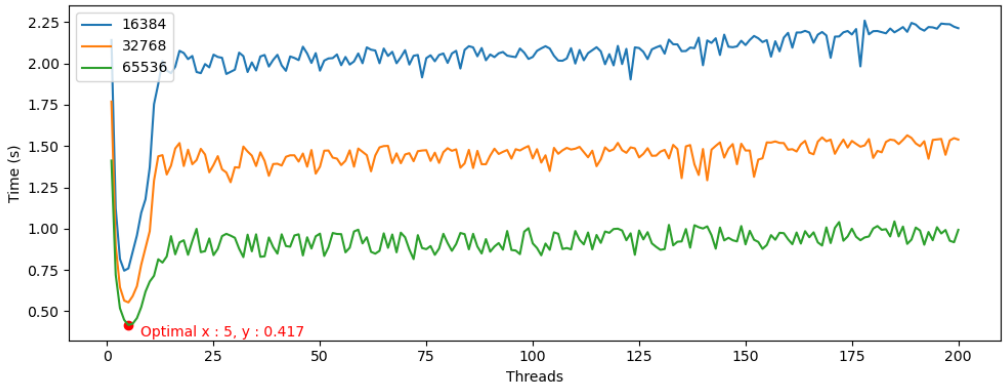
위의 두 결과를 통해 buffer 의 size 가 증가할수록 실행 시간이 감소한다는 것을 알 수 있었다. 그러나 thread 개수의 증가가 작업의 실행시간 단축에 전혀 도움이 되지 않았다.

Test-good-result

/opt/android.tar



/opt/FreeBSD9-orig.tar



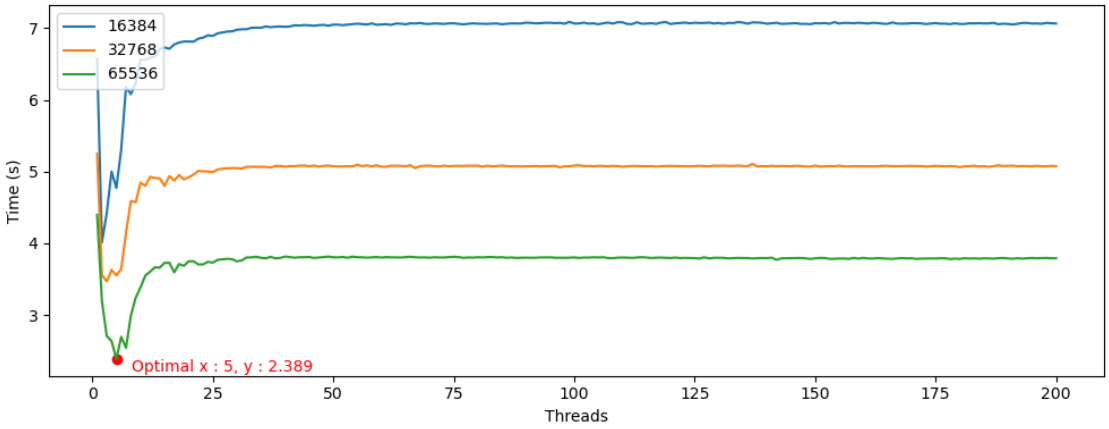
Critical section 에서 처리되는 일의 양을 줄이고, 이를 critical section 밖으로 빼내어 처리하였을 때 결과가 향상되었다. Thread 의 개수가 5 개가 될 때까지 실행시간이 급격하게 감소하였으나, 이후에는 실행 시간이 증가하였다.

코드에서의 주요한 변경사항은 아래와 같다.

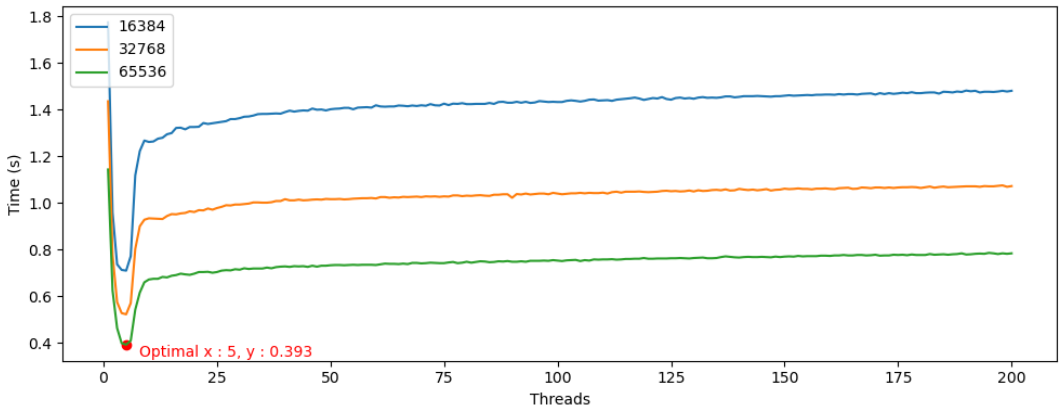
```
len = strlen(cptr);
for (int i = 0 ; i < len ; i++) {
        char c = cptr[i];
        if (c < 256 && c > 1) {
                stat[c]++;
        }
}
```

이 부분은 실제 수행되는 작업이다. 그런데, 이 부분이 shared object 의 stat 을 사용하여 수행되는 경우, critical section 안에서 작업이 수행되어야 한다. 이를 thread 내에서 stat 을 동적할당한 것으로 대체하고, stat 을 thread 의 join 시에 넘겨준 후 main 에서 결과를 합하면 위처럼 concurrency 의 향상을 이룰 수 있다. 동일한 test 를 단일 사용자만 존재하는 조금 더 stable 한 환경에서 수행한 결과는 아래와 같다.

/opt/android.tar



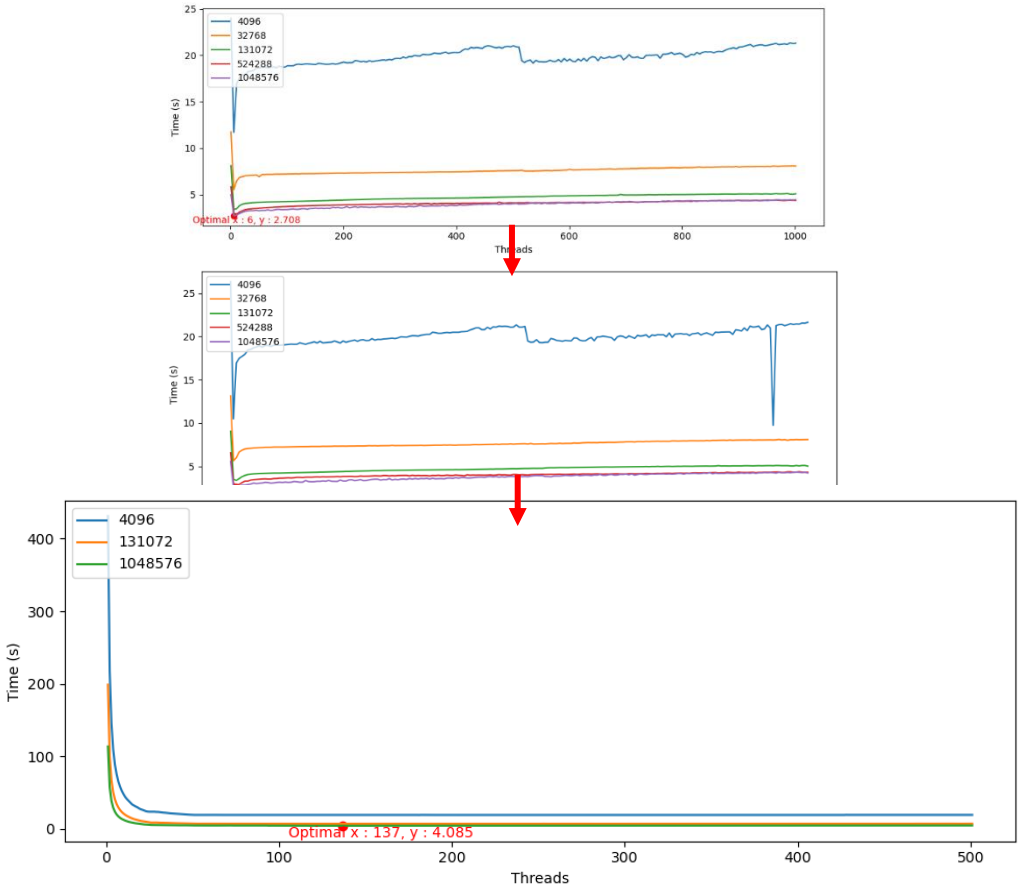
/opt/FreeBSD9-orig.tar



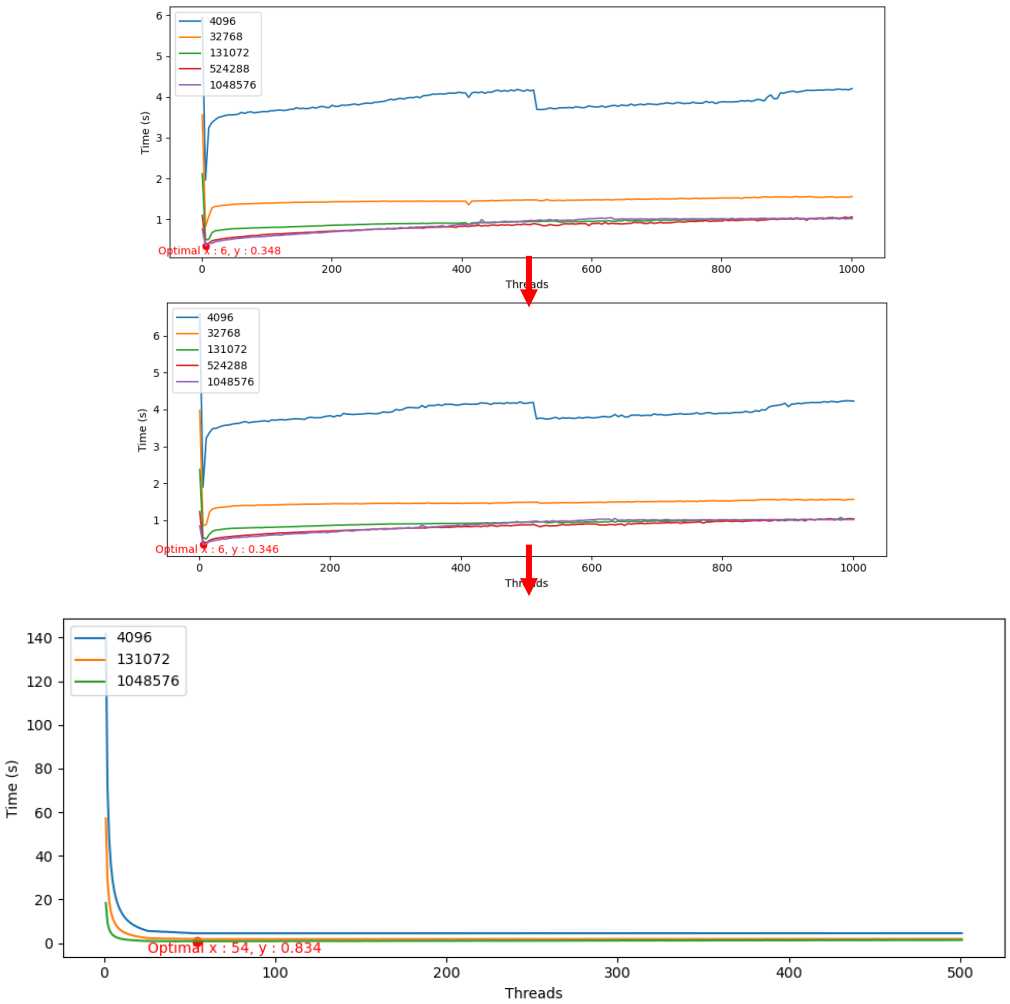
Test-result-increasing-overhead

이후에는 overhead 를 증가시키며 test 를 수행하였다.

/opt/android.tar



/opt/FreeBSD9-orig.tar



이러한 방식의 test 를 통해 확인할 수 있었던 것은 thread 의 개수가 optimal 일 때보다 많아질 때 증가하는 실행 시간이 선형에 가깝다는 것이었다. 그런데 전체 작업 양을 증가시키며 overhead 를 점차 증가시키자 optimal 한 thread 의 개수보다 많아질 때의 실행 시간 곡선이 점차 아래로 내려오는 것을 확인할 수 있었다. 따라서 최종적으로 아래의 코드와 같이, 작업 시의 overhead 를 증가시키자 thread 개수의 증가가 실행 시간 향상에 크게 도움이 됨을 확인할 수 있었다.

```
        char *cptr = copy;
        char *substr = NULL;
        char *brka = NULL;
        char *sep = "{}()[],;\" \n\t^";

        for (substr = strtok_r(cptr, sep, &brka); substr; substr =
strtok_r(NULL, sep, &brka)) {

            len = strlen(cptr);

            for (int i = 0 ; i < len ; i++) {
                int c = *cptr;
                if (c < 256 && c > 1) {
                    stat[c]++;
                }

                // more overhead
                for (int j = 0; j < c; j++) {
                    arr[(len + j) % 5] *= c;
                }
                // more overhead

                cptr++;
            }
            cptr++;
            if (*cptr == '\0') break;

        }
```

consumer 에서 위와 같이 각 알파벳의 정수 값만큼 어떠한 연산을 수행할 때, 이 연산은 thread 의 개수와 상관없이 전체 프로그램에 큰 overhead 가 된다. 이를 multi-threaded 구현으로 여러 thread 가 나눠서 수행할 경우 실행 시간 단축에 유의미한 개선을 만들 수 있다.

이를 간단하게 수식화하였다.

만약 전체 프로그램의 실행이 완료되는 시간을 thread 의 개수와 관련하여 생각하면, 실행시간은 크게 T 와 J 로 나눠서 생각할 수 있다. T 는 thread 의 생성과 공유 자원 접근 등 thread 의 개수에 따라 증가하는 시간이고, thread 의 개수에 비례한다. J 는 전체 작업에 걸리는 시간으로, thread 의 개수가 증가하면 일이 나눠서 수행되므로 thread 의 개수에 반비례한다. 또한 thread 의 실행과 무관하게 프로그램의 실행에 소모되는 시간 U 역시 존재한다. 따라서 수식은 아래와 같다.

$$Result\ Time$$
$$= T_t + J_t + U$$
$$= f(N_t) = N_t * C_t + J_1 / N_t + U$$

$$T_t = N_t * C_t$$

$$J_t = (N_j / N_t) * C_j = J_1 / N_t$$

T_t : $Time\ consumed\ to\ make\ and\ use\ threads$

N_t : $number\ of\ threads$

C_t : $Time\ cost\ to\ make\ a\ thread$

J_t : $Time\ threads\ consume\ to\ finish\ all\ jobs$

$J_1 = N_j * C_j$

N_j : $Number\ of\ jobs\ to\ do$

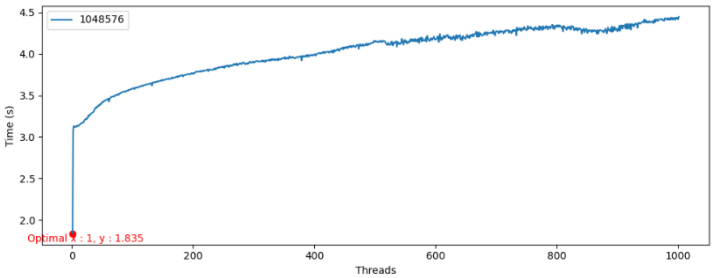
C_j : $Time\ cost\ to\ do\ a\ job$

U : $Time\ not\ related\ with\ threads$

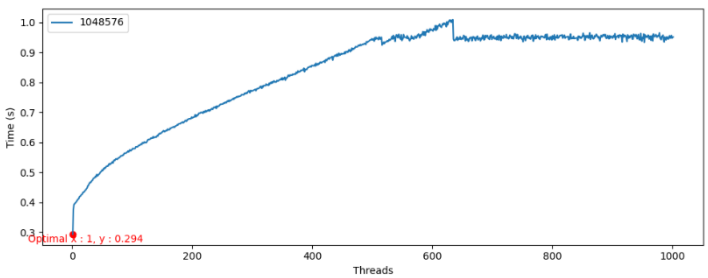
이러한 수식을 이용하여 overhead 가 큰 작업의 경우 thread 의 개수를 증가시키며 test 를 진행하지 않고, 실제 thread 를 증가시키며 test 를 진행하는 것과 비슷한 그래프를 근사할 수 있다. 수식에서 최소한으로 얻어야 하는 값은 C_t, U, J_1 이다.

이를 위해서 먼저 작업을 제외하고 thread 를 증가시키며 T_t 를 구한다. 이 시행은 overhead 가 작으므로 빠르게 결과를 확인할 수 있다. 그 결과는 아래와 같다.

/opt/android.tar

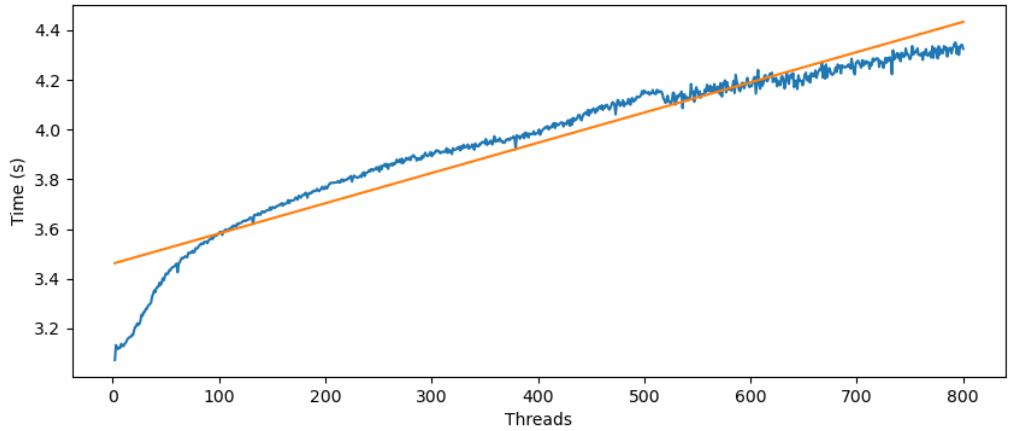


/opt/FreeBSD9-orig.tar

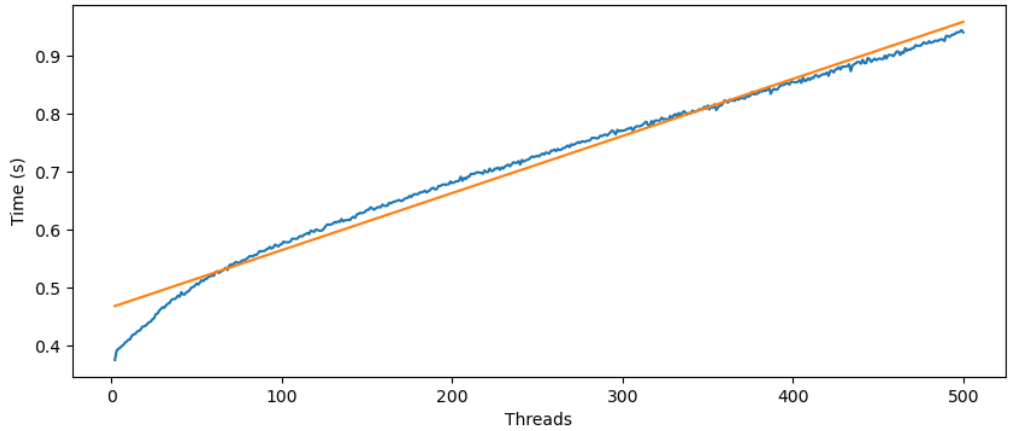


이를 이용하여 그래프의 값이 튀는 부분을 제외하고 선형 회귀를 수행하면 아래의 직선을 얻을 수 있다. 직선을 통해 기울기와 절편을 구하면, 이는 각각 C_t와 U가 된다.

/opt/android.tar



/opt/FreeBSD9-orig.tar



이후 작업을 포함해서 1 개의 thread 에 대해 측정을 수행하면 아래와 같이 J_1을 구할 수 있다.

```
 / ~/2022-os-hw2 / on  32181928 !1                                                    base  / with chanho@lab310 / at 01:00:31
❯ time ./main /opt/android.tar 1 1 1048576
./main /opt/android.tar 1 1 1048576  111.96s user 1.82s system 100% cpu 1:52.98 total

 / ~/2022-os-hw2 / on  32181928 !1                                  took 1m 53s  / base  / with chanho@lab310 / at 01:02:26
❯ time ./main /opt/FreeBSD9-orig.tar 1 1 1048576
./main /opt/FreeBSD9-orig.tar 1 1 1048576  18.28s user 0.26s system 101% cpu 18.342 total
```

이를 통해 필요한 값을 모두 구할 수 있다

/opt/android.tar

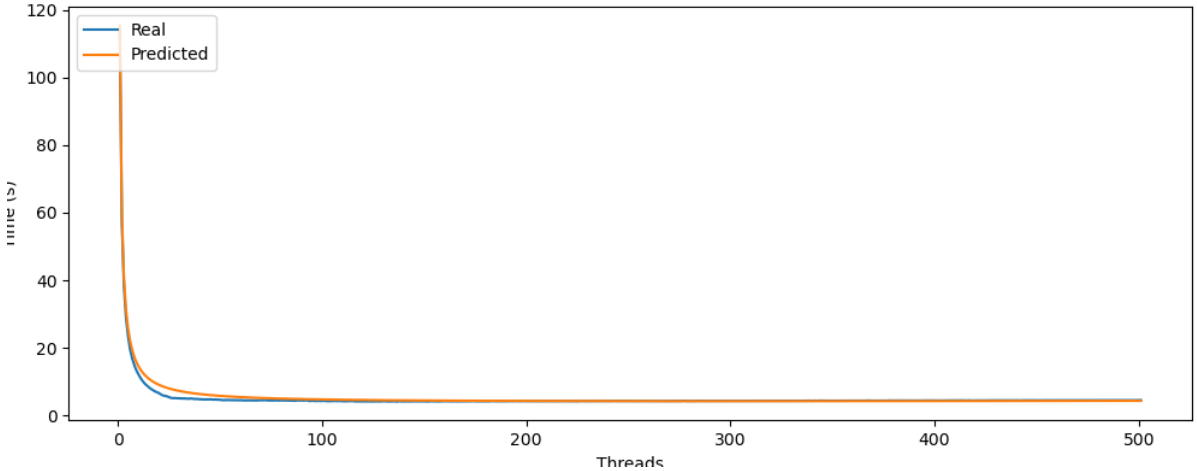
{'U': 3.460347914686592, 'C_t': array([0.00121625]), 'J_1': 111.96}

/opt/FreeBSD9-orig.tar

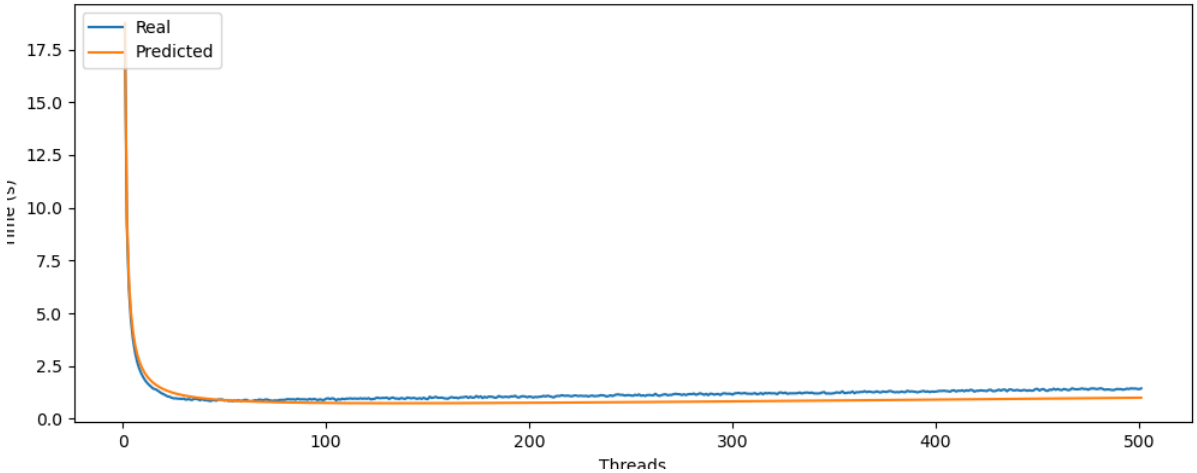
{'U': 0.4670290933674578, 'C_t': array([0.00098433]), 'J_1': 18.28}

이를 통해 수식을 그래프로 나타내면 아래와 같다. Real 은 실제 측정 결과, Predicted 는 수식을 나타낸 그래프이다.

/opt/android.tar



/opt/FreeBSD9-orig.tar



U 값을 정확하게 측정할 수 없어 thread 개수가 1 일 때의 값이 높게 그려지기는 하지만 실제 overhead 가 큰 작업을 수행하지 않고도 원래의 test 를 근사할 수 있으며, 이는 optimal thread 개수를 예측하는 등의 작업에 참고할 수 있는 결과이다.

참고자료

Operating Systems : Three easy pieces – Remzi H. Arpaci-Dusseau

감사합니다